TiDB

TiDB 用户指南（企业版）

产品版本：v7.1 | 文档版本：20230531



PingCAP

北京平凯星辰科技发展有限公司

目录

1 关于 TiDB 11

1.1 TiDB 简介 ····· 11

1.1.1 五大核心特性 ····· 11

1.1.2 四大核心应用场景 ····· 11

1.1.3 另请参阅 ····· 12

1.2 TiDB 7.1.0 Release Notes ····· 12

1.2.1 功能详情 ····· 13

1.2.2 兼容性变更 ····· 17

1.2.3 改进提升 ····· 27

1.2.4 贡献者 ····· 27

1.3 TiDB 功能概览 ····· 28

1.3.1 数据类型，函数和操作符 ····· 28

1.3.2 索引和约束 ····· 29

1.3.3 SQL 语句 ····· 29

1.3.4 高级 SQL 功能 ····· 30

1.3.5 数据定义语言 (DDL) ····· 30

1.3.6 事务 ····· 30

1.3.7 分区 ····· 31

1.3.8 统计信息 ····· 31

1.3.9 安全 ····· 31

1.3.10 数据导入和导出 ····· 32

1.3.11 管理，可视化和工具 ····· 32

1.4 与 MySQL 兼容性对比 ····· 33
1.4.1 不支持的功能特性 ····· 34
1.4.2 与 MySQL 有差异的特性详细说明 ····· 34
1.5 使用限制 ····· 39
1.5.1 标识符长度限制 ····· 39
1.5.2 Databases、 Tables、 Views、 Connections 总个数限制 ····· 39
1.5.3 单个 Database 的限制 ····· 40
1.5.4 单个 Table 的限制 ····· 40
1.5.5 单行的限制 ····· 40
1.5.6 单列的限制 ····· 40
1.5.7 数据类型限制 ····· 40
1.5.8 SQL Statements 的限制 ····· 41
1.5.9 TiKV 版本的限制 ····· 41
2 快速上手 41
2.1 TiDB 数据库快速上手指南 ····· 41
2.1.1 部署本地测试集群 ····· 41
2.1.2 在单机上模拟部署生产环境集群 ····· 45
2.1.3 探索更多 ····· 49
2.2 HTAP 快速上手指南 ····· 50
2.2.1 基础概念 ····· 50
2.2.2 体验步骤 ····· 50
2.2.3 探索更多 ····· 54
2.3 SQL 基本操作 ····· 54
2.3.1 分类 ····· 54
2.3.2 查看、创建和删除数据库 ····· 54
2.3.3 创建、查看和删除表 ····· 55
2.3.4 创建、查看和删除索引 ····· 55
2.3.5 记录的增删改 ····· 56
2.3.6 查询数据 ····· 56
2.3.7 创建、授权和删除用户 ····· 57

2.4 HTAP 深入探索指南 · · · · · 57
2.4.1 HTAP 适用场景 · · · · · 58
2.4.2 HTAP 架构 · · · · · 58
2.4.3 HTAP 环境准备 · · · · · 58
2.4.4 HTAP 数据准备 · · · · · 59
2.4.5 HTAP 数据处理 · · · · · 59
2.4.6 HTAP 性能监控 · · · · · 59
2.4.7 HTAP 故障诊断 · · · · · 60
2.4.8 探索更多 · · · · · 60

3 教程 60
3.1 单区域多AZ 部署TiDB · · · · · 60
3.1.1 了解Raft 协议 · · · · · 60
3.1.2 同区域三AZ 方案 · · · · · 61
3.2 双区域多AZ 部署TiDB · · · · · 66
3.2.1 简介 · · · · · 66
3.2.2 架构 · · · · · 66
3.2.3 配置 · · · · · 68
3.3 单区域双AZ 部署TiDB · · · · · 72
3.3.1 简介 · · · · · 72
3.3.2 部署架构 · · · · · 72
3.3.3 配置 · · · · · 73
3.4 读取历史数据 · · · · · 79
3.4.1 使用 Stale Read 功能读取历史数据（推荐） · · · · · 79
3.4.2 通过系统变量tidb_snapshot 读取历史数据 · · · · · 90
3.5 最佳实践 · · · · · 95
3.5.1 TiDB 最佳实践 · · · · · 95
3.5.2 开发 Java 应用使用 TiDB 的最佳实践 · · · · · 100
3.5.3 HAProxy 在 TiDB 中的最佳实践 · · · · · 109
3.5.4 TiDB 高并发写入场景最佳实践 · · · · · 117
3.5.5 使用 Grafana 监控 TiDB 的最佳实践 · · · · · 126
3.5.6 PD 调度策略最佳实践 · · · · · 136
3.5.7 海量 Region 集群调优最佳实践 · · · · · 144

3.5.8 三节点混合部署的最佳实践 149
3.5.9 在三数据中心下就近读取数据 153
3.5.10 UUID 最佳实践 153
3.5.11 只读存储节点最佳实践 155
3.6 Placement Rules 使用文档 157
3.6.1 规则系统介绍 158
3.6.2 配置规则操作步骤 160
3.6.3 典型场景示例 164
3.7 Load Base Split 167
3.7.1 场景描述 167
3.7.2 实现原理 167
3.7.3 使用方法 168
3.8 Store Limit 169
3.8.1 实现原理 169
3.8.2 使用方法 170
3.8.3 Store Limit v2 原理 170
3.9 DDL 语句的执行原理及最佳实践 170
3.9.1 DDL 执行原理 170
3.9.2 最佳实践 174
3.9.3 DDL 相关的命令介绍 175
3.9.4 常见问题 175
4 参考指南 175
4.1 系统变量 175
4.1.1 变量参考 176
4.2 配置文件参数 292
4.2.1 TiDB 配置文件描述 292
4.2.2 TiKV 配置文件描述 317
4.2.3 TiFlash 配置参数 375
4.2.4 PD 配置文件描述 384
4.3 CLI 396
4.3.1 TiKV Control 使用说明 396
4.3.2 PD Control 使用说明 410
4.3.3 TiDB Control 使用说明 437
4.3.4 PD Recover 使用文档 443

4.3.5 binlogctl 工具 ····· 447
4.4 命令行参数 ····· 449
4.4.1 TiDB 配置参数 ····· 449
4.4.2 TiKV 配置参数 ····· 455
4.4.3 TiFlash 命令行参数 ····· 457
4.4.4 PD 配置参数 ····· 459
4.5 监控指标 ····· 462
4.5.1 Overview 面板重要监控指标详解 ····· 462
4.5.2 Performance Overview 面板重要监控指标详解 ····· 464
4.5.3 TiDB 重要监控指标详解 ····· 470
4.5.4 PD 重要监控指标详解 ····· 475
4.5.5 TiKV 监控指标详解 ····· 483
4.5.6 TiFlash 集群监控 ····· 502
4.5.7 TiCDC 详细监控指标 ····· 505
4.5.8 资源管控 (Resource Control) 监控指标详解 ····· 510
4.6 安全加固 ····· 512
4.6.1 为 TiDB 客户端服务端间通信开启加密传输 ····· 512
4.6.2 为 TiDB 组件间通信开启加密传输 ····· 516
4.6.3 生成自签名证书 ····· 520
4.6.4 静态加密 ····· 522
4.6.5 为 TiDB 落盘文件开启加密 ····· 530
4.6.6 日志脱敏 ····· 530
4.7 权限 ····· 531
4.7.1 与 MySQL 安全特性差异 ····· 531
4.7.2 权限管理 ····· 534
4.7.3 TiDB 用户账户管理 ····· 544
4.7.4 TiDB 密码管理 ····· 548
4.7.5 基于角色的访问控制 ····· 557
4.7.6 TiDB 证书鉴权使用指南 ····· 563
4.8 SQL ····· 571
4.8.1 SQL 语言结构和语法 ····· 571
4.8.2 SQL 语句 ····· 611
4.8.3 数据类型 ····· 952
4.8.4 函数与操作符 ····· 968
4.8.5 聚簇索引 ····· 1008
4.8.6 约束 ····· 1014

4.8.7 生成列 1020
4.8.8 SQL 模式 1024
4.8.9 表属性 1044
4.8.10 事务 1047
4.8.11 视图 1081
4.8.12 分区表 1086
4.8.13 临时表 1124
4.8.14 缓存表 1130
4.8.15 外键约束 1135
4.8.16 字符集和排序 1143
4.8.17 Placement Rules in SQL 1155
4.8.18 系统表 1161
4.8.19 元数据锁 1271
4.9 遥测 1274
4.9.1 哪些使用情况信息会被收集？ 1275
4.9.2 开启遥测功能 1276
4.9.3 禁用遥测功能 1279
4.9.4 查看遥测启用状态 1281
4.9.5 使用情况信息合规性 1281
4.10 错误码与故障诊断 1282
4.10.1 错误码 1282
4.10.2 故障诊断 1290
4.11 通过拓扑 label 进行副本调度 1290
4.11.1 根据集群拓扑配置 labels 1291
4.11.2 基于拓扑 label 的 PD 调度策略 1295
4.12 TiDB 白名单插件 1296
4.12.1 获取插件 1296
4.12.2 部署插件 1296
4.12.3 查看状态 1298
4.12.4 启用和禁用插件 1299
4.12.5 使用和验证插件 1299
4.13 TiDB 审计插件 1300
4.13.1 获取插件 1301
4.13.2 部署审计插件 1301

4.13.3 查看状态 1305
4.13.4 启用和禁用插件 1305
4.13.5 修改 TABLE_ACCESS 相关配置 1306
4.13.6 使用插件输出审计日志 1307
4.13.7 日志输出 1308
4.13.8 清除审计日志 1310
4.14 内部组件介绍 1310
4.14.1 TiDB 后端任务分布式框架 1310

5 常见问题解答 (FAQ) 1314
5.1 TiDB 产品常见问题 1314
5.1.1 1.1 TiDB 介绍及整体架构 1314
5.1.2 1.2 TiDB 原理 1316
5.2 SQL 操作常见问题 1316
5.2.1 TiDB 是否支持二级键？ 1316
5.2.2 TiDB 在对大表执行 DDL 操作时，性能表现如何？ 1317
5.2.3 如何选择正确的查询计划？是否需要使用优化器提示？还是可以使用提示？ 1317
5.2.4 如何阻止特定的 SQL 语句执行（或者将某个 SQL 语句加入黑名单）？ 1317
5.2.5 TiDB 对哪些 MySQL variables 兼容？ 1318
5.2.6 省略 ORDER BY 条件时 TiDB 中返回结果的顺序与 MySQL 中的不一致 1318
5.2.7 TiDB 是否支持 SELECT FOR UPDATE？ 1319
5.2.8 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？ 1319
5.2.9 一个事务中的语句数量最大是多少？ 1319
5.2.10 TiDB 中，为什么出现后插入数据的自增 ID 反而小？ 1319
5.2.11 如何在 TiDB 中修改 sql_mode？ 1320
5.2.12 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？ 1320
5.2.13 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？ 1320
5.2.14 TiDB 中删除数据后会立即释放空间吗？ 1320
5.2.15 删除数据后查询速度为何会变慢？ 1320
5.2.16 对数据做删除操作之后，空间回收比较慢，如何处理？ 1321
5.2.17 SHOW PROCESSLIST 是否显示系统进程号？ 1321
5.2.18 在 TiDB 中如何控制或改变 SQL 提交的执行优先级？ 1321
5.2.19 在 TiDB 中 auto analyze 的触发策略是怎样的？ 1322

5.2.20 可以使用 Optimizer Hints 控制优化器行为吗？ ····· 1322
5.2.21 DDL 执行 ····· 1322
5.2.22 执行 DDL 会慢的可能原因 ····· 1322
5.2.23 SQL 优化 ····· 1325
5.2.24 数据库优化 ····· 1326
5.3 高可用常见问题 ····· 1327
5.3.1 TiDB 数据是强一致的吗？ ····· 1327
5.3.2 TiDB 是否提供三中心跨机房多活部署的推荐方案？ ····· 1327
5.4 高可靠常见问题 ····· 1327
5.4.1 TiDB 是否支持数据加密？ ····· 1327
5.4.2 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？ ····· 1327
5.4.3 TiDB 支持哪些认证协议？过程是怎样的？ ····· 1327
5.4.4 如何修改用户名密码和权限？ ····· 1328
6 术语表 1328
6.1 A ····· 1328
6.1.1 ACID ····· 1328
6.2 B ····· 1328
6.2.1 BR ····· 1328
6.2.2 Batch Create Table ····· 1329
6.2.3 Baseline Capturing ····· 1329
6.2.4 Bucket ····· 1329
6.3 C ····· 1329
6.3.1 Cached Table ····· 1329
6.3.2 Coalesce Partition ····· 1329
6.3.3 Continuous Profiling ····· 1329
6.4 D ····· 1329
6.4.1 Dynamic Pruning ····· 1329
6.5 I ····· 1330
6.5.1 Index Merge ····· 1330
6.5.2 In-Memory Pessimistic Lock ····· 1330
6.6 L ····· 1330
6.6.1 Leader/Follower/Learner ····· 1330
6.7 O ····· 1330
6.7.1 Old value ····· 1330

6.7.2 Operator ····· 1330
6.7.3 Operator Step ····· 1330
6.8 P ····· 1331
6.8.1 Partitioning ····· 1331
6.8.2 Pending/Down ····· 1331
6.8.3 Point get ····· 1331
6.8.4 Predicate columns ····· 1331
6.9 Q ····· 1331
6.9.1 Quota Limiter ····· 1331
6.10 R ····· 1331
6.10.1 Raft Engine ····· 1331
6.10.2 Region/Peer/Raft Group ····· 1332
6.10.3 Region Split ····· 1332
6.10.4 Restore ····· 1332
6.11 S ····· 1332
6.11.1 Scheduler ····· 1332
6.11.2 Store ····· 1332
6.12 T ····· 1332
6.12.1 Top SQL ····· 1332
6.12.2 TSO ····· 1333
6.12.3 TTL ····· 1333

1 关于TiDB

1.1 TiDB 简介

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

关于TiDB 的关键技术创新，请观看以下视频。

1.1.1 五大核心特性

- 一键水平扩容或者缩容

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

- 金融级高可用

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

- 实时 HTAP

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

- 云原生的分布式数据库

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

- 兼容 MySQL 5.7 协议和 MySQL 生态

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

1.1.2 四大核心应用场景

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

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

- 对存储容量、可扩展性、并发要求较高的海量数据及高并发的 OLTP 场景

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

- Real-time HTAP 场景

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

- 数据汇聚、二次加工处理的场景

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

关于TiDB 典型应用场景和用户案例的介绍，请观看以下视频。

1.1.3 另请参阅

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

1.2 TiDB 7.1.0 Release Notes

TiDB 版本：7.1.0 (upcoming)

> **注意：**
>
> TiDB v7.1.0 尚未正式发布。此Release Notes 旨在帮助你提前了解即将发布的版本，但其中内容可能会有所调整。本文列出的功能并不保证会包含在最终发布的版本中。

在 7.1.0 版本中，你可以获得以下关键特性：

<th>分类</th>	<th>功能</th>	<th>描述</th>

<td rowspan="2">可扩展性与性能</td>
<td>
    <↪ 会话级别非 Prepare 语句执行计划缓存 (GA)</td>
<td>支持在会话级别自动重用执行计划缓存，减少查询计划时间，缩短相同 SQL 查询的时间，
    <↪ 而无需事先手动准备 Prepare Statement 语句。 </td>

<td><a href="https://docs.pingcap.com/zh/tidb/dev/troubleshoot-hot-spot-issues#打散读热点" target
    <↪ ="_blank">基于负载的副本读取</td>
<td>在读热点场景中，TiDB 可以将热点 TiKV 节点的读请求转发到副本。
    <↪ 该功能有效地打散了读热点并优化了集群资源的利用。你可以通过调整系统变量 <a href="https://
    <↪ docs.pingcap.com/zh/tidb/dev/system-variables#tidb_load_based_replica_read_threshold-从-
    <↪ v700-版本开始引入" target="_blank"><code>tidb_load_based_replica_read_threshold</code></a
    <↪ > 控制基于负载的副本读取的触发阈值。 </td>

<td rowspan="1">稳定性与高可用</td>
<td>资源管控
    <↪ (GA)</td>

支持基于资源组的资源管控，将数据库用户映射到对应的资源组中，并根据实际需求设置每个资源组的配额。

<td rowspan="2">SQL</td>
<td><a href="https://docs.pingcap.com/zh/tidb/dev/sql-statement-create-index#多值索引" target="
    <↪ _blank">多值索引 (GA)</td>
<td>引入 MySQL 兼容的多值索引，增强 JSON 类型，提升 TiDB 对 MySQL 8.0 的兼容性。
    <↪ 该功能提升了对多值列进行成员检查的效率。 </td>

<td>生成列 (
    <↪ GA)</td>
<td>生成列 (Generated Columns) 的值是通过实时计算列定义中的 SQL 表达式得到的。
    <↪ 该功能将一些应用逻辑推向数据库层，从而提升查询效率。 </td>

1.2.1 功能详情

1.2.1.1 性能

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

  在 v7.0.0 中，TiFlash 引入了延迟物化实验特性， 用于优化查询性能。该特性默认关闭（系统变量tidb_opt_enable_late_materialization 默认为 OFF）。当 SELECT 语句中包含过滤条件（WHERE 子句）时，TiFlash 默认会先读取该查询所需列的全部数据，然后再根据查询条件对数据进行过滤、聚合等计算任务。开启该特性后，TiFlash 支持下推部分过滤条件到 TableScan 算子，即先扫描过滤条件相关的列数据，过滤得到符合条件的行后，再扫描这些行的其他列数据，继续后续计算，从而减少IO 扫描和数据处理的计算量。

从v7.1.0 开始，TiFlash 延迟物化功能正式GA，默认开启（系统变量tidb_opt_enable_late_materialization 默认为 ON），TiDB 优化器会根据统计信息和查询的过滤条件，决定哪些过滤条件会被下推到 TableScan 算子。

更多信息，请参考用户文档。

- TiFlash 支持根据网络交换数据量自动选择 MPP 模式的Join 算法#7084 @solotzg

  TiFlash MPP 模式有多种Join 算法。在v7.1.0 之前的版本中，TiDB 根据变量tidb_broadcast_join_threshold_count <⟶> 和tidb_broadcast_join_threshold_size 以及实际数据量决定 TiFlash MPP 模式是否使用 Broadcast Hash Join 算法。

  在 v7.1.0 中，TiDB 引入变量tidb_prefer_broadcast_join_by_exchange_data_size，控制是否基于最小网络数据交换策略选择 MPP Join 算法。该变量默认关闭，表示默认保持 v7.1.0 之前的算法选择策略。如需开启，请设置该变量为 ON。开启后，你无需再手动调整tidb_broadcast_join_threshold_count 和tidb_broadcast_join_threshold_size 的阈值（此时这两个变量将不再生效），TiDB 会自动估算不同 Join 算法所需进行网络交换的数据量，然后选择综合开销较小的算法，从而减少网络流量，提升 MPP 查询性能。

  更多信息，请参考用户文档。

- 支持自适应副本读取缓解读热点#14151 @sticnarf @you06

  在读热点场景中，热点TiKV 无法及时处理读请求，导致读请求排队。但是，此时并非所有 TiKV 资源都已耗尽。为了降低延迟，TiDB v7.1.0 引入了负载自适应副本读取功能，允许从其他有可用资源的TiKV 节点读取副本，而无需在热点TiKV 节点排队等待。你可以通过tidb_load_based_replica_read_threshold 系统变量控制读请求的排队长度。当leader 节点的预估排队时间超过该阈值时，TiDB 会优先从 follower 节点读取数据。在读热点的情况下，与不打散读热点相比，该功能可提高读取吞吐量 70% ~ 200%。

  更多信息，请参考用户文档。

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

  TiDB v7.0.0 引入了非 Prepare 语句的执行计划缓存作为实验特性，以提升在线交易场景的并发处理能力。该功能在 v7.1.0 正式GA 并默认打开，支持缓存更多模式的 SQL。

  为了提升内存利用率，TiDB v7.1.0 将非 Prepare 与 Prepare 语句的缓存池合并。你可以通过系统变量tidb_session_plan_cache_size 设置缓存大小。原有的系统变量tidb_prepared_plan_cache_size 和tidb_non_prepared_plan_cache_size 被废弃。

  为保持向前兼容， 从旧版本升级到 v7.1.0 时， 缓存池大小 tidb_session_plan_cache_size 的值与 tidb_prepared_plan_cache_size 保持一致，tidb_enable_non_prepared_plan_cache 保持升级前的设置。经过性能测试后，你可通过 tidb_enable_non_prepared_plan_cache 开启非 Parepare 语句的执行计划缓存功能。对于新创建的 v7.1.0 集群，非 Parepare 语句的缓存功能默认打开。

  更多信息，请参考用户文档。

1.2.1.2 稳定性

- 资源管控成为正式功能(GA) #38825 @nolouch @BornChanger @glorv @tiancaiamao @Connor1996 @JmPotato @hnes @CabinfeverB @HuSharp

  TiDB 持续增强资源管控能力，在 v7.1.0 该功能正式GA。该特性将极大地提升TiDB 集群的资源利用率和性能表现。资源管控特性的引入对TiDB 具有里程碑的意义，你可以将一个分布式数据库集群划分成多个

逻辑单元，将不同的数据库用户映射到对应的资源组中，并根据实际需求设置每个资源组的配额。当集群资源紧张时，同一资源组内的会话所使用的全部资源将受到配额限制，防止某一资源组的过度消耗对其他资源组的会话造成影响。

该特性也可以将多个来自不同系统的中小型应用整合到同一个TiDB 集群中。即使某个应用的负载增加，也不会影响其他应用的正常运行。而在系统负载较低的时候，繁忙的应用即使超出设定的配额，仍可获得所需系统资源，实现资源的最大化利用。此外，合理利用资源管控特性可以减少集群数量，降低运维难度及管理成本。

在 TiDB v7.1.0 中，该特性增加了基于实际负载和硬件部署来估算系统容量上限的能力，为你进行容量规划提供更准确的参考。这有助于你更好地管理TiDB 的资源分配，从而满足企业级场景的稳定性需求。

更多信息，请参考用户文档。

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

  TiDB v7.1.0 引入Fast Online DDL 的检查点机制，可以大幅提升 Fast Online DDL 的容错性和自动恢复能力。即使 TiDB owner 因故障重启或者切换，TiDB 也能够通过自动定期保存的检查点恢复部分进度，从而让 DDL 执行更加稳定高效。

  更多信息，请参考用户文档。

- BR 备份恢复工具支持断点恢复#42339 @Leavrth

  快照恢复或日志恢复会因为一些可恢复性错误导致提前结束，例如硬盘空间占满、节点宕机等突发情况。在TiDB v7.1.0 之前，即使错误被及时处理，之前恢复的进度也会作废，你需要重新进行恢复。对大规模集群来说，会造成大量额外成本。

  为了尽可能继续上一次的恢复，从TiDB v7.1.0 起，备份恢复特性引入了断点恢复的功能。该功能可以在意外中断后保留上一次恢复的大部分进度。

  更多信息，请参考用户文档。

- 优化统计信息缓存加载策略#42160 @xuyifangreeneyes

  开启统计信息同步加载后，TiDB 可以大幅减少启动时必须载入的统计信息的数量，从而提升启动过程中统计信息的加载速度。该特性提升了TiDB 在复杂运行环境下的稳定性，并降低了部分TiDB 节点重启对整体服务的影响。

  更多信息，请参考用户文档。

1.2.1.3 SQL 功能

- 支持通过 `INSERT INTO SELECT` 语句保存TiFlash 查询结果(GA) #37515 @gengliqi

  从 v6.5.0 起，TiDB 支持下推 `INSERT INTO SELECT` 语句中的 `SELECT` 子句（分析查询）到TiFlash，你可以将 TiFlash 的查询结果方便地保存到 `INSERT INTO` 指定的TiDB 表中供后续分析使用，起到了结果缓存（即结果物化）的效果。

  在 v7.1.0 版本中，该功能正式 GA。当 TiDB 执行 `INSERT INTO SELECT` 语句中的 `SELECT` 子句时，优化器将根据SQL 模式及TiFlash 副本的代价估算自行决定是否将查询下推至TiFlash。因此，在实验特性阶段引入的系统变量 `tidb_enable_tiflash_read_for_write_stmt` 将被废弃。需要注意的是，TiFlash 对于 `INSERT INTO ... SELECT` 语句的计算规则不满足 `STRICT SQL Mode` 要求，因此只有当前会话的SQL 模式为非严格模式（即`sql_mode` 值不包含`STRICT_TRANS_TABLES` 和`STRICT_ALL_TABLES`），TiDB 才允许将`INSERT INTO SELECT` 语句中的 `SELECT` 子句下推至TiFlash。

  更多信息，请参考用户文档。

- MySQL 兼容的多值索引(Multi-Valued Index) 成为正式功能(GA) #39592 @xiongjiwei @qw4990 @YangKeao

  过滤 JSON 列中某个数组的值是一种常见操作，但使用普通索引无法加速此过程。在数组上创建多值索引可以大幅提升过滤性能。如果 JSON 列中的某个数组上存在多值索引，则函数 `MEMBER OF()`、`JSON_CONTAINS()` 和 `JSON_OVERLAPS()` 的检索条件可以利用该多值索引进行过滤，从而减少大量的 I/O 消耗，提升执行速度。

  在 v7.1.0 中，TiDB 多值索引成为正式功能(GA)，支持更完整的数据类型，并与TiDB 的工具链兼容。你可以在生产环境利用多值索引来加速对JSON 数组的检索操作。

  更多信息，请参考用户文档。

- 完善 Hash 分区表和 Key 分区表的分区管理功能#42728 @mjonss

  在 v7.1.0 之前，TiDB 中的 Hash 分区表和 Key 分区表只支持 `TRUNCATE PARTITION` 分区管理。从 v7.1.0 开始，Hash 分区表和 Key 分区表新增支持 `ADD PARTITION` 和 `COALESCE PARTITION` 分区管理。你可以根据需要灵活调整 Hash 分区表和 Key 分区表的分区数量。例如，通过 `ADD PARTITION` 语句增加分区数量，或通过 `COALESCE PARTITION` 语句减少分区数量。

  更多信息，请参考用户文档。

- Range INTERVAL 分区定义语法成为正式功能(GA) #35683 @mjonss

  在 v6.3.0 中引入的Range INTERVAL 的分区定义语法成为正式功能(GA)。通过该语法，你可以根据所需的间隔 (interval) 定义 Range 分区，不需要枚举所有分区，可大幅度缩短Range 分区表的定义语句长度。语义与原有Range 分区等价。

  更多信息，请参考用户文档。

- 生成列(Generated Columns) 成为正式功能(GA) @bb7133

  生成列是数据库中非常有价值的一个功能。在创建表时，可以定义一列的值由表中其他列的值计算而来，而不是由用户显式插入或更新。这个生成列可以是虚拟列(Virtual Column) 或存储列(Stored Column)。 TiDB 在早期版本就提供了与 MySQL 兼容的生成列功能，在 v7.1.0 中这个功能正式GA。

  使用生成列可以提升TiDB 对 MySQL 的兼容性，方便从 MySQL 平滑迁移到TiDB，同时也能简化数据维护复杂度，增强数据一致性并提高查询效率。

  更多信息，请参考用户文档。

1.2.1.4 可观测性

- 增加优化器诊断信息#43122 @time-and-fate

  获取充足的信息是 SQL 性能诊断的关键。在 v7.1.0 中，TiDB 持续为各种诊断工具增加优化器运行信息，以便更好地解释执行计划如何被选择，从而协助定位 SQL 性能问题。这些信息包括：

  - 在 `PLAN REPLAYER` 的输出中增加 `debug_trace.json` 文件。
  - 在 `EXPLAIN` 的输出中为 `operator info` 添加部分统计信息详情。
  - 为慢日志的 `Stats` 字段添加部分统计信息详情。

更多信息，请参考使用 `PLAN REPLAYER` 保存和恢复集群线程信息、使用 `EXPLAIN` 解读执行计划和慢日志查询。

1.2.1.5 安全

- 更换TiFlash 系统表信息的查询接口#6941 @flowbehappy

  从 v7.1.0 起，TiFlash 在向 TiDB 提供INFORMATION_SCHEMA.TIFLASH_TABLES 和INFORMATION_SCHEMA.TIFLASH_SEGMENTS 系统表的查询服务时，不再使用 HTTP 端口，而是使用 gRPC 端口，从而避免 HTTP 服务的安全风险。

1.2.2 兼容性变更

> **注意：**
>
> 以下为从 v7.0.0 升级至当前版本(v7.1.0) 所需兼容性变更信息。如果从 v6.6.0 或之前版本升级到当前版本，可能也需要考虑和查看中间版本Release Notes 中提到的兼容性变更信息。

1.2.2.1 行为变更

- 为了提高安全性，TiFlash 废弃了 HTTP 服务端口（默认 8123），采用 gRPC 端口作为替代

  如果你已经将 TiFlash 升级到 v7.1.0，那么在升级 TiDB 到 v7.1.0 的过程中，TiDB 无法读取 TiFlash 系统表（INFORMATION_SCHEMA.TIFLASH_TABLES 和INFORMATION_SCHEMA.TIFLASH_SEGMENTS）。

1.2.2.2 系统变量

变量名	修改类型	描述
tidb_enable_tiflash_read_for_write_stmt	废弃	从 v7.1.0 开始，该变量废弃并且默认值从 OFF 修改为 ON。当 tidb_allow_mpp = ON 时，优化器将根据SQL 模式及 TiFlash 副本的代价估算自行决定是否将查询下推至 TiFlash。
tidb_non_prepared_plan_cache_size	废弃	从 v7.1.0 起，该变量被废弃，你可以使用tidb_session_plan_cache_size 控制 Plan Cache 最多能够缓存的计划数量。

变量名	修改类型	描述
tidb_prepared_plan_cache_size <↩>	废弃	从 v7.1.0 起，该变量被废弃，你可以使用 tidb_session_plan_cache_size <↩> 控制 Plan Cache 最多能够缓存的计划数量。
tidb_ddl_dist_execute_reorg <↩>	删除	重命名为 tidb_enable_dist_task <↩>。
tidb_enable_non_prepared_plan_cache <↩>	修改	经进一步的测试后，该变量默认值从 OFF 修改为 ON，表示默认开启非 Prepare 语句执行计划缓存。

变量名	修改类型	描述
tidb_load_based_replica_read_threshold	修改	该变量从 v7.1.0 开始生效，用于设置基于负载的 replica read 的触发阈值。经进一步的测试后，该变量默认值从 "0s" 修改为 "1s"。
tidb_opt_enable_late_materialization	修改	默认值从 OFF 修改为 ON，代表 TiFlash 延迟物化功能默认开启。

变量名	修改类型	描述
tidb_enable_dist_task <↔>	新增	控制是否开启分布式执行框架。开启分布式执行后，DDL、Import 等支持的后端任务将会由集群中多个 TiDB 节点共同完成。该变量由 tidb_ddl_distribute_reorg <↔> 改名而来。
tidb_enable_non_prepared_plan_cache_for_dml <↔>	新增	控制非 Prepare 语句执行计划缓存是否支持 DML 语句。

变量名	修改类型	描述
tidb_opt_fix_control ⟷	新增	通过设置该变量，你可以更细粒度地控制优化器的行为，并且避免集群升级后优化器行为变化导致的性能回退。
tidb_plan_cache_invalidation_on_fresh_stats ⟷	新增	控制当某张表上的统计信息更新后，与该表相关的 Plan Cache 是否自动失效。
tidb_plan_cache_max_plan_size ⟷	新增	控制可以缓存的 Prepare 或非 Prepare 语句执行计划的最大大小。

变量名	修改类型	描述
tidb_prefer_broadcast_join_by_exchange_data_size <→	新增	控制是否使用最小网络数据交换策略。使用该策略时，TiDB 会估算 Broadcast Hash Join 和 Shuffled Hash Join 两种算法所需进行网络交换的数据量，并选择网络交换数据量较小的算法。该功能开启后，tidb_broadcast_join_threshold_size <→ 和tidb_broadcast_join_threshold_count <→ 将不再生效。

变量名	修改类型	描述
tidb_session_plan_cache_size	新增	控制 Plan Cache 最多能够缓存的计划数量。其中，Prepare 语句执行计划缓存和非 Prepare 语句执行计划缓存共用一个缓存。

1.2.2.3 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	force-init-stats	新增	用于控制 TiDB 启动时是否在统计信息初始化完成后再对外提供服务。
TiDB	lite-init-stats	新增	用于控制 TiDB 启动时是否采用轻量级的统计信息初始化。

配置文件	配置项	修改类型	描述
TiDB	timeout	新增	用于控制 TiDB 写日志操作的超时时间，当磁盘故障导致日志无法写入时，该配置可以让 TiDB 进程崩溃而不是卡死。默认值为 0，即不设定超时时间。
TiKV	optimize-

配置文件	配置项	修改类型	描述
TiKV	split.byte-threshold	修改	当region-split-size 大于等于 4 GB 时，默认值从 30MiB 修改为 100MiB。
TiKV	split.qps-threshold	修改	当region-split-size 大于等于 4 GB 时，默认值从 3000 修改为 7000。
TiKV	split.region-cpu-overload-threshold-ratio	修改	当region-split-size 大于等于 4 GB 时，默认值从 0.25 修改为 0.75。
PD	store-limit-version	新增	用于设置 store limit 工作模式。可选值为 "v1" 和 "v2"。
PD	schedule.enable-diagnostic	修改	默认值从 false 修改为 true，默认打开调度器的诊断功能。
TiFlash	http_port	删除	废弃 TiFlash HTTP 服务端口（默认 8123）。

配置文件	配置项	修改类型	描述
TiCDC	sink. <↩> enable <↩> - <↩> partition <↩> - <↩> separator <↩>	修改	默认值从 false 修改为 true，代表默认会将表中各个分区的数据分不同的目录来存储。建议保持该配置项为 true 以避免同步分区表到存储服务时可能丢数据的问题。

1.2.3 改进提升

- TiFlash
  - 提升 TiFlash 在存算分离架构下的性能和稳定性#6882 @JaySon-Huang @breezewish @JinheLin
  - 支持在 Semi Join 或 Anti Semi Join 中，通过选择较小的表作为 Build 端来优化查询性能#7280 @yibin87
- TiCDC
  - TiCDC 过滤了由有损 DDL 语句导致的数据变更，避免发送无效数据变更#43227 @hi-rustin

1.2.4 贡献者

感谢来自TiDB 社区的贡献者们：

- ethercflow
- hihihuhu
- jiyfhust
- L-maple
- lqs
- pingandb
- yorkhellen
- yujiarista

1.3 TiDB 功能概览

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

> 注意：
>
> PingCAP 不提供基于 DMR 版本的bug 修复版本，如有bug，会在后续版本中修复。如无特殊需求，建议使用最新 LTS 版本。
>
> 下表中出现的缩写字母含义如下：
>
> - Y：已GA 的功能，可以在生产环境中使用。注意即使某个功能在 DMR 版本中GA，也建议在后续 LTS 版本中将该功能用于生产环境。
> - N：不支持该功能。
> - E：未GA 的功能，即实验特性(experimental)，请注意使用场景限制。实验特性会在未事先通知的情况下发生变化或删除。语法和实现可能会在 GA 前发生变化。如果遇到问题，请在 GitHub 上提交issue 反馈。

1.3.1 数据类型，函数和操作符

数据类型，函数，操作符	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
数值类型	Y	Y	Y	Y	Y	Y	Y	Y	Y
日期和时间类型	Y	Y	Y	Y	Y	Y	Y	Y	Y
字符串类型	Y	Y	Y	Y	Y	Y	Y	Y	Y
JSON 类型	Y	Y	E	E	E	E	E	E	E
控制流程函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
字符串函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
数值函数与操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y
日期和时间函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
位函数和操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y
Cast 函数和操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y
加密和压缩函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
信息函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
JSON 函数	Y	Y	E	E	E	E	E	E	E
聚合函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
窗口函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
其他函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y
字符集和排序规则 [1]	Y	Y	Y	Y	Y	Y	Y	Y	Y
用户级别锁	Y	Y	Y	N	N	N	N	N	N

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

1.3.2 索引和约束

索引和约束	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0		
表达式索引[2]	Y	Y	E	E	E	E	E	E	E		
列式存储(TiFlash)	Y	Y	Y	Y	Y	Y	Y	Y	Y		
使用 FastScan 加速 OLAP 场景下的查询	Y	E	N	N	N	N	N	N	N		
RocksDB 引擎	Y	Y	Y	Y	Y	Y	Y	Y	Y		
Titan 插件	Y	Y	Y	Y	Y	Y	Y	Y	Y		
Titan Level Merge	E	E	E	E	E	E	E	E	E		
使用bucket 提高数据扫描并发度	E	E	E	N	N	N	N	N	N		
不可见索引	Y	Y	Y	Y	Y	Y	Y	Y	N		
复合主键	Y	Y	Y	Y	Y	Y	Y	Y	Y		
唯一约束	Y	Y	Y	Y	Y	Y	Y	Y	Y		
整型主键上的聚簇索引	Y	Y	Y	Y	Y	Y	Y	Y	Y		
复合或非整型主键上的聚簇索引	Y	Y	Y	Y	Y	Y	Y	Y	N		
多值索引	Y	N	N	N	N	N	N	N	N		
外键约束	Y	N	N	N	N	N	N	N	N		
TiFlash 延迟物化	Y	N	N	N	N	N	N	N	N		

1.3.3 SQL 语句

SQL 语句[3]	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
SELECT, INSERT, UPDATE, DELETE, REPLACE	Y	Y	Y	Y	Y	Y	Y	Y	Y
INSERT ON DUPLICATE KEY UPDATE	Y	Y	Y	Y	Y	Y	Y	Y	Y
LOAD DATA INFILE	Y	Y	Y	Y	Y	Y	Y	Y	Y
SELECT INTO OUTFILE	Y	Y	Y	Y	Y	Y	Y	Y	Y
INNER JOIN, LEFT|RIGHT [OUTER] JOIN	Y	Y	Y	Y	Y	Y	Y	Y	Y
UNION, UNION ALL	Y	Y	Y	Y	Y	Y	Y	Y	Y
EXCEPT 和 INTERSECT 运算符	Y	Y	Y	Y	Y	Y	Y	Y	N
GROUP BY, ORDER BY	Y	Y	Y	Y	Y	Y	Y	Y	Y
窗口函数	Y	Y	Y	Y	Y	Y	Y	Y	Y
公共表表达式(CTE)	Y	Y	Y	Y	Y	Y	Y	N	N
START TRANSACTION, COMMIT, ROLLBACK	Y	Y	Y	Y	Y	Y	Y	Y	Y
EXPLAIN	Y	Y	Y	Y	Y	Y	Y	Y	Y
EXPLAIN ANALYZE	Y	Y	Y	Y	Y	Y	Y	Y	Y
用户自定义变量	E	E	E	E	E	E	E	E	E
BATCH [ON COLUMN] LIMIT INTEGER DELETE	Y	Y	Y	N	N	N	N	N	N
BATCH [ON COLUMN] LIMIT INTEGER INSERT/UPDATE/REPLACE	Y	Y	N	N	N	N	N	N	N
ALTER TABLE ... COMPACT	Y	Y	E	N	N	N	N	N	N
表级锁(Table Lock)	E	E	E	E	E	E	E	E	E
物化列式存储的查询结果	Y	E	N	N	N	N	N	N	N

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

1.3.4 高级 SQL 功能

高级 SQL 功能	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Prepare 语句执行计划缓存	Y	Y	Y	Y	Y	E	E	E	E
非 Prepare 语句执行计划缓存	Y	N	N	N	N	N	N	N	N
执行计划管理(SPM)	Y	Y	Y	Y	Y	Y	Y	Y	Y
根据历史执行计划创建绑定	Y	E	N	N	N	N	N	N	N
下推计算结果缓存(Coprocessor Cache)	Y	Y	Y	Y	Y	Y	Y	Y	E
Stale Read	Y	Y	Y	Y	Y	Y	Y	N	N
Follower Read	Y	Y	Y	Y	Y	Y	Y	Y	Y
通过系统变量 tidb_snapshot 读取历史数据	Y	Y	Y	Y	Y	Y	Y	Y	Y
Optimizer hints	Y	Y	Y	Y	Y	Y	Y	Y	Y
MPP 执行引擎	Y	Y	Y	Y	Y	Y	Y	Y	N
MPP 执行引擎- compression exchange	Y	N	N	N	N	N	N	N	N
索引合并	Y	Y	Y	Y	E	E	E	E	E
基于 SQL 的数据放置规则	Y	Y	Y	E	E	N	N	N	N
Cascades Planner	E	E	E	E	E	E	E	E	E

1.3.5 数据定义语言(DDL)

数据定义语言(DDL)	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
CREATE, DROP, ALTER, RENAME, TRUNCATE	Y	Y	Y	Y	Y	Y	Y	Y	Y
生成列	Y	E	E	E	E	E	E	E	E
视图	Y	Y	Y	Y	Y	Y	Y	Y	Y
序列	Y	Y	Y	Y	Y	Y	Y	Y	Y
AUTO_INCREMENT 列	Y	Y[4]	Y	Y	Y	Y	Y	Y	Y
AUTO_RANDOM 列	Y	Y	Y	Y	Y	Y	Y	Y	Y
TTL (Time to Live)	Y	E	N	N	N	N	N	N	N
DDL 算法断言	Y	Y	Y	Y	Y	Y	Y	Y	Y
在单条语句中添加多列	Y	Y	E	E	E	E	E	E	E
更改列类型	Y	Y	Y	Y	Y	Y	Y	N	N
临时表	Y	Y	Y	Y	Y	N	N	N	N
并行 DDL	Y	Y	N	N	N	N	N	N	N
添加索引加速	Y	Y	N	N	N	N	N	N	N
元数据锁	Y	Y	N	N	N	N	N	N	N
FLASHBACK CLUSTER TO TIMESTAMP	Y	Y	N	N	N	N	N	N	N

1.3.6 事务

[4]从TiDB v6.4.0 开始，支持高性能、全局单调递增的 AUTO_INCREMENT 列

事务	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Async commit	Y	Y	Y	Y	Y	Y	Y	Y	N
1PC	Y	Y	Y	Y	Y	Y	Y	Y	N
大事务 (10 GB)	Y	Y	Y	Y	Y	Y	Y	Y	Y
悲观事务	Y	Y	Y	Y	Y	Y	Y	Y	Y
乐观事务	Y	Y	Y	Y	Y	Y	Y	Y	Y
可重复读隔离（快照隔离）	Y	Y	Y	Y	Y	Y	Y	Y	Y
读已提交隔离	Y	Y	Y	Y	Y	Y	Y	Y	Y

1.3.7 分区

分区	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Range 分区	Y	Y	Y	Y	Y	Y	Y	Y	Y
Hash 分区	Y	Y	Y	Y	Y	Y	Y	Y	Y
Key 分区	Y	N	N	N	N	N	N	N	N
List 分区	Y	Y	Y	E	E	E	E	E	N
List COLUMNS 分区	Y	Y	Y	E	E	E	E	E	N
EXCHANGE PARTITION	Y	Y	E	E	E	E	E	E	N
REORGANIZE PARTITION	Y	N	N	N	N	N	N	N	N
COALESCE PARTITION	Y	N	N	N	N	N	N	N	N
动态裁剪	Y	Y	Y	E	E	E	E	N	N
Range COLUMNS 分区	Y	Y	N	N	N	N	N	N	N
Range INTERVAL 分区	Y	E	N	N	N	N	N	N	N

1.3.8 统计信息

统计信息	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0
CM-Sketch	默认关闭	默认关闭	默认关闭	默认关闭	默认关闭	Y	Y	Y
直方图	Y	Y	Y	Y	Y	Y	Y	Y
扩展统计信息（多列）	E	E	E	E	E	E	E	E
统计反馈	N	N	已废弃	已废弃	E	E	E	E
统计信息自动更新	Y	Y	Y	Y	Y	Y	Y	Y
快速分析	E	E	E	E	E	E	E	E
动态裁剪	Y	Y	Y	E	E	E	E	N
收集部分列的统计信息	E	E	E	E	N	N	N	N
限制统计信息的内存使用量	E	E	E	N	N	N	N	N
随机采样约10000行数据来快速构建统计信息	E	E	E	E	E	E	E	E
锁定统计信息	E	E	N	N	N	N	N	N

1.3.9 安全

安全	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
传输层加密(TLS)	Y	Y	Y	Y	Y	Y	Y	Y	Y
静态加密(TDE)	Y	Y	Y	Y	Y	Y	Y	Y	Y
基于角色的访问控制(RBAC)	Y	Y	Y	Y	Y	Y	Y	Y	Y
证书鉴权	Y	Y	Y	Y	Y	Y	Y	Y	Y
caching_sha2_password 认证	Y	Y	Y	Y	Y	Y	N	N	N
tidb_sm3_password 认证	Y	Y	N	N	N	N	N	N	N
tidb_auth_token 认证	Y	Y	N	N	N	N	N	N	N
密码管理	Y	Y	N	N	N	N	N	N	N
与 MySQL 兼容的 GRANT 权限管理	Y	Y	Y	Y	Y	Y	Y	Y	Y
动态权限	Y	Y	Y	Y	Y	Y	Y	N	N
安全增强模式	Y	Y	Y	Y	Y	Y	Y	N	N
日志脱敏	Y	Y	Y	Y	Y	Y	Y	Y	N

1.3.10 数据导入和导出

数据导入和导出	7.1 (upcoming)	6.5	6.1	5.4	5.3	
快速导入(TiDB Lightning)	Y	Y	Y	Y	Y	
mydumper 逻辑导出	已废弃	已废弃	已废弃	已废弃	已废弃	已
Dumpling 逻辑导出	Y	Y	Y	Y	Y	
事务 LOAD DATA	Y[5]	Y	Y	Y	Y	
数据迁移工具	Y	Y	Y	Y	Y	
TiDB Binlog	Y	Y	Y	Y	Y	
Change data capture (CDC)	Y	Y	Y	Y	Y	
TiCDC 支持保存数据到存储服务(Amazon S3/GCS/Azure Blob Storage/NFS)	Y	E	N	N	N	
TiCDC 支持在两个TiDB 集群之间进行双向复制	Y	Y	N	N	N	
TiCDC OpenAPI v2	Y	N	N	N	N	

1.3.11 管理，可视化和工具

管理，可视化和工具	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
TiDB Dashboard 图形化展示	Y	Y	Y	Y	Y	Y	Y	Y	Y
TiDB Dashboard 持续性能分析功能	Y	Y	Y	E	E	N	N	N	N
TiDB Dashboard Top SQL 功能	Y	Y	Y	E	N	N	N	N	N
TiDB Dashboard SQL 诊断功能	Y	Y	E	E	E	E	E	E	E
TiDB Dashboard 集群诊断功能	Y	Y	E	E	E	E	E	E	E
Grafana 中的TiKV-FastTune 面板	E	E	E	E	E	E	E	E	E
Information schema	Y	Y	Y	Y	Y	Y	Y	Y	Y
Metrics schema	Y	Y	Y	Y	Y	Y	Y	Y	Y

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

管理，可视化和工具	7.1 (upcoming)	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Statements summary tables	Y	Y	Y	Y	Y	Y	Y	Y	Y
Statements summary tables - 持久化 statements summary	E	N	N	N	N	N	N	N	N
慢查询日志	Y	Y	Y	Y	Y	Y	Y	Y	Y
TiUP 部署	Y	Y	Y	Y	Y	Y	Y	Y	Y
Kubernetes operator	Y	Y	Y	Y	Y	Y	Y	Y	Y
内置物理备份	Y	Y	Y	Y	Y	Y	Y	Y	Y
Global Kill	Y	Y	Y	E	E	E	E	E	E
Lock View	Y	Y	Y	Y	Y	Y	E	E	E
SHOW CONFIG	Y	Y	Y	Y	Y	Y	Y	Y	Y
SET CONFIG	Y	Y	Y	E	E	E	E	E	E
DM WebUI	E	E	E	N	N	N	N	N	N
前台限流	Y	Y	E	N	N	N	N	N	N
后台限流	E	E	N	N	N	N	N	N	N
基于 EBS 的备份和恢复	Y	Y	N	N	N	N	N	N	N
PITR	Y	Y	N	N	N	N	N	N	N
全局内存控制	Y	Y	N	N	N	N	N	N	N
RawKV 跨集群复制	E	E	N	N	N	N	N	N	N
Green GC	E	E	E	E	E	E	E	E	N
资源管控(Resource Control)	Y	N	N	N	N	N	N	N	N
TiFlash 存算分离架构与 S3 支持	E	N	N	N	N	N	N	N	N

1.4 与 MySQL 兼容性对比

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

但TiDB 尚未支持一些 MySQL 功能，可能的原因如下：

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

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

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

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

1.4.1 不支持的功能特性

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 全文语法与索引#1793
- 空间类型的函数（即 GIS/GEOMETRY）、数据类型和索引 #6347
- 非 ascii、latin1、binary、utf8、utf8mb4、gbk 的字符集
- SYS schema
- MySQL 追踪优化器
- XML 函数
- X-Protocol #1109
- 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
- REPAIR TABLE 语法
- OPTIMIZE TABLE 语法
- HANDLER 语句
- CREATE TABLESPACE 语句
- “Session Tracker: 将 GTID 上下文信息添加到OK 包中”

1.4.2 与 MySQL 有差异的特性详细说明

1.4.2.1 自增ID

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

自增ID 详情可参阅AUTO_INCREMENT。

> 注意：

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

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

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

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

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

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

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

注意：

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

1.4.2.2 Performance schema

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

1.4.2.3 查询计划

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

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

详情参见理解TiDB 执行计划。

1.4.2.4 内建函数

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

1.4.2.5 DDL 的限制

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

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

更多详情，请参考分区表文档。

1.4.2.6 ANALYZE TABLE

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

更多信息统计的差异请参阅ANALYZE TABLE。

1.4.2.7 SELECT 的限制

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

详情参见SELECT。

1.4.2.8 UPDATE 语句

详情参见UPDATE。

1.4.2.9 视图

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

1.4.2.10 临时表

详见TiDB 本地临时表与 MySQL 临时表的兼容性。

1.4.2.11 字符集和排序规则

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

1.4.2.12 存储引擎

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

1.4.2.13 SQL 模式

TiDB 支持大部分SQL 模式。不支持的 SQL 模式如下：

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

1.4.2.14 默认设置

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

1.4.2.15 日期时间处理的区别

1.4.2.15.1 时区

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

1.4.2.16 类型系统

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

1.4.2.17 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。

1.4.2.18 CREATE RESOURCE GROUP，DROP RESOURCE GROUP 和 ALTER RESOURCE GROUP

TiDB 资源组创建与修改语句的语法与 MySQL 官方不同，详情参见：

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

1.5 使用限制

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

1.5.1 标识符长度限制

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

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

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

标识符类型	最大个数
Connections	unlimited

1.5.3 单个 Database 的限制

类型	最大限制
Tables	unlimited

1.5.4 单个 Table 的限制

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

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

1.5.5 单行的限制

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

1.5.6 单列的限制

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

1.5.7 数据类型限制

类型	最大限制
CHAR	256 字符
BINARY	256 字节
VARBINARY	65535 字节
VARCHAR	16383 字符

类型	最大限制
TEXT	默认为 6291456 字节（即 6 MiB），可调至 125829120 字节（即 120 MiB）
BLOB	默认为 6291456 字节（即 6 MiB），可调至 125829120 字节（即 120 MiB）

1.5.8 SQL Statements 的限制

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

1.5.9 TiKV 版本的限制

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

2 快速上手

2.1 TiDB 数据库快速上手指南

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

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

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

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

2.1.1 部署本地测试集群

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

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

1. 下载并安装TiUP。

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

安装完成后会提示如下信息：

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

2. 声明全局环境变量。

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

```
source ${your_shell_profile}
```

3. 在当前 session 执行以下命令启动集群。

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

```
tiup playground
```

- 也可以指定TiDB 版本以及各组件实例个数，命令类似于：

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

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

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

注意：

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

4. 新开启一个 session 以访问TiDB 数据库。

   - 使用TiUP `client` 连接TiDB：

     ```
     tiup client
     ```

   - 也可使用 MySQL 客户端连接TiDB：

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

5. 通过http://127.0.0.1:9090 访问TiDB 的 Prometheus 管理界面。
6. 通过http://127.0.0.1:2379/dashboard 访问TiDB Dashboard 页面，默认用户名为 `root`，密码为空。
7. 通过http://127.0.0.1:3000 访问TiDB 的 Grafana 界面，默认用户名和密码都为 `admin`。
8. （可选）将数据加载到TiFlash 进行分析。
9. 测试完成之后，可以通过执行以下步骤来清理集群：

   1. 按下 Control+C 键停掉上述启用的TiDB 服务。
   2. 等待服务退出操作完成后，执行以下命令：

      ```
      tiup clean --all
      ```

注意：

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

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

1. 下载并安装TiUP。

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

   安装完成后会提示如下信息：

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

2. 声明全局环境变量。

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

```
source ${your_shell_profile}
```

3. 在当前 session 执行以下命令启动集群。

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

     ```
     tiup playground
     ```

   - 也可以指定TiDB 版本以及各组件实例个数，命令类似于：

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

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

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

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

4. 新开启一个 session 以访问TiDB 数据库。

    - 使用TiUP client 连接TiDB：

    ```
    tiup client
    ```

    - 也可使用 MySQL 客户端连接TiDB：

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

5. 通过http://127.0.0.1:9090 访问TiDB 的 Prometheus 管理界面。
6. 通过http://127.0.0.1:2379/dashboard 访问TiDB Dashboard 页面，默认用户名为 root，密码为空。
7. 通过http://127.0.0.1:3000 访问TiDB 的 Grafana 界面，默认用户名和密码都为 admin。
8. （可选）将数据加载到TiFlash 进行分析。
9. 测试完成之后，可以通过执行以下步骤来清理集群：

    1. 按下 Control+C 键停掉上述启用的TiDB 服务。
    2. 等待服务退出操作完成后，执行以下命令：

```
tiup clean --all
```

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

2.1.2 在单机上模拟部署生产环境集群

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

本节介绍如何参照TiUP 最小拓扑的一个YAML 文件部署TiDB 集群。

2.1.2.1 准备环境

准备一台部署主机，确保其软件满足需求：

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

最小规模的TiDB 集群拓扑：

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

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

部署主机软件和环境要求：

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

2.1.2.2 实施部署

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

1. 下载并安装TiUP：

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

2. 声明全局环境变量：

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

```
source ${your_shell_profile}
```

3. 安装 TiUP 的 cluster 组件：

```
tiup cluster
```

4. 如果机器已经安装TiUP cluster，需要更新软件版本：

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

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

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

    ```
    service sshd restart
    ```

6. 创建并启动集群

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

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

    配置模板如下：

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

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

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

```
 tiflash:
   logger.level: "info"

pd_servers:
 - host: 10.0.1.1

tidb_servers:
 - host: 10.0.1.1

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

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

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

tiflash_servers:
 - host: 10.0.1.1

monitoring_servers:
 - host: 10.0.1.1

grafana_servers:
 - host: 10.0.1.1
```

7. 执行集群部署命令：

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

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

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

按照引导，输入”y”及 root 密码，来完成部署：

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

8. 启动集群：

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

9. 访问集群：

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

     ```
     yum -y install mysql
     ```

   - 访问TiDB 数据库，密码为空：

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

   - 访问TiDB 的 Grafana 监控：

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

   - 访问TiDB 的 Dashboard：

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

   - 执行以下命令确认当前已经部署的集群列表：

     ```
     tiup cluster list
     ```

   - 执行以下命令查看集群的拓扑结构和状态：

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

2.1.3 探索更多

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

2.2 HTAP 快速上手指南

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

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

2.2.1 基础概念

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

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

2.2.2 体验步骤

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

2.2.2.1 第 1 步：部署试用环境

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

```
tiup playground
```

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

2.2.2.2 第 2 步：准备试用数据

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

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

1. 使用以下命令安装数据生成工具：

```
tiup install bench
```

2. 使用以下命令生成数据：

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

当命令行输出 Finished 时，表示数据生成完毕。

3. 运行以下 SQL 语句查看生成的数据：

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

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

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

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

2.2.2.3 第 3 步：使用行存查询数据

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

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

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

2.2.2.4 第 4 步：同步列存数据

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

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

如需查询TiFlash 表的同步状态，请使用以下 SQL 语句：

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

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

以上查询结果中：

- 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。副本状态变为可用之后就不再改变。
- PROGRESS 字段代表同步进度，在 0.0~1.0 之间，1 代表TiFlash 副本已经完成同步。

2.2.2.5 第 5 步：使用 HTAP 更快地分析数据

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

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。如需查看实际是否选择了TiFlash 副本，可以使用 desc 或 explain analyze 语句，例如：

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

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

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

你可以对比两次的查询结果和查询性能。

2.2.3 探索更多

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

2.3 SQL 基本操作

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

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

2.3.1 分类

SQL 语言通常按照功能划分成以下的 4 个部分：

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

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

2.3.2 查看、创建和删除数据库

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

使用 SHOW DATABASES 语句查看系统中数据库列表：

```
SHOW DATABASES;
```

使用名为 mysql 的数据库：

```
USE mysql;
```

使用 SHOW TABLES 语句查看数据库中的所有表。例如：

```
SHOW TABLES FROM mysql;
```

使用 CREATE DATABASE 语句创建数据库。语法如下：

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

例如，要创建一个名为 samp_db 的数据库，可使用以下语句：

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

添加 IF NOT EXISTS 可防止发生错误。

使用 DROP DATABASE 语句删除数据库。例如：

```
DROP DATABASE samp_db;
```

2.3.3 创建、查看和删除表

使用 CREATE TABLE 语句创建表。语法如下：

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

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

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

使用 SHOW CREATE 语句查看建表语句，即 DDL。例如：

```
SHOW CREATE TABLE person;
```

使用 DROP TABLE 语句删除表。例如：

```
DROP TABLE person;
```

2.3.4 创建、查看和删除索引

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

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

或者：

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

对于值唯一的列，可以创建唯一索引。例如：

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

或者：

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

使用 SHOW INDEX 语句查看表内所有索引：

```
SHOW INDEX FROM person;
```

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

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

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

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

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

2.3.5 记录的增删改

使用 INSERT 语句向表内插入表记录。例如：

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

使用 INSERT 语句向表内插入包含部分字段数据的表记录。例如：

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

使用 UPDATE 语句向表内修改表记录的部分字段数据。例如：

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

使用 DELETE 语句向表内删除部分表记录。例如：

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

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

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

2.3.6 查询数据

使用 SELECT 语句检索表内数据。例如：

```
SELECT * FROM person;
```

在 SELECT 后面加上要查询的列名。例如：

```
SELECT name FROM person;
```

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

使用WHERE 子句，对所有记录进行是否符合条件的筛选后再返回。例如：

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

常用的DCL 功能是创建或删除用户，和对用户权限的管理。

2.3.7 创建、授权和删除用户

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

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

授权用户 tiuser 可检索数据库 samp_db 内的表：

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

查询用户 tiuser 的权限：

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

删除用户 tiuser：

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

2.4 HTAP 深入探索指南

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

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

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

2.4.1 HTAP 适用场景

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

以下是三种 HTAP 典型适用场景：

- 混合负载场景

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

- 实时流处理场景

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

- 数据中枢场景

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

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

2.4.2 HTAP 架构

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

更多架构信息，请参考 TiDB HTAP 形态架构。

2.4.3 HTAP 环境准备

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

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

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

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

2.4.4 HTAP 数据准备

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

- 如果TiDB 集群中还没有数据，请先迁移数据到TiDB。详情请参阅数据迁移。
- 如果TiDB 集群中已经有从上游同步过来的数据，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，详情请参阅使用TiFlash。

2.4.5 HTAP 数据处理

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

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

2.4.6 HTAP 性能监控

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

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

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

2.4.7 HTAP 故障诊断

在使用TiDB 的过程中如果遇到问题，请参阅以下文档：

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

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

2.4.8 探索更多

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

3 教程

3.1 单区域多AZ 部署TiDB

作为一栈式实时 HTAP 数据库，TiDB 兼顾了传统关系型数据库的优秀特性、NoSQL 数据库可扩展性以及跨可用区 (Availability Zone, AZ) 场景下的高可用。本文档旨在介绍同区域多AZ 部署TiDB 的方案。

本文中的区域指的是地理隔离的不同位置，AZ 指的是区域内部划分的相互独立的资源集合。本文描述的方案同样适用于一个城市内多个数据中心（同城多中心）的场景。

3.1.1 了解Raft 协议

Raft 是一种分布式一致性算法，在TiDB 集群的多种组件中，PD 和TiKV 都通过Raft 实现了数据的容灾。Raft 的灾难恢复能力通过如下机制实现：

- Raft 成员的本质是日志复制和状态机。Raft 成员之间通过复制日志来实现数据同步；Raft 成员在不同条件下切换自己的成员状态，其目标是选出leader 以提供对外服务。
- Raft 是一个表决系统，它遵循多数派协议，在一个Raft Group 中，某成员获得大多数投票，它的成员状态就会转变为leader。也就是说，当一个Raft Group 还保有大多数节点(majority) 时，它就能够选出leader 以提供对外服务。

遵循Raft 可靠性的特点，放到现实场景中：

- 想克服任意 1 台服务器(host) 的故障，应至少提供 3 台服务器。
- 想克服任意 1 个机柜(rack) 的故障，应至少提供 3 个机柜。
- 想克服任意 1 个可用区（AZ，也可以是同城的多个机房）的故障，应至少提供 3 个AZ。

- 想应对任意 1 个区域的灾难场景，应至少规划 3 个区域用于部署集群。

可见，原生Raft 协议对于偶数副本的支持并不是很友好，考虑跨区域网络延迟影响，同区域三AZ 可能是最适合部署 Raft 的高可用及容灾方案。

3.1.2 同区域三AZ 方案

同区域三AZ 方案，即同区域有三个机房部署TiDB 集群，AZ 间的数据在集群内部（通过 Raft 协议）进行同步。同区域三AZ 可同时对外进行读写服务，任意中心发生故障不影响数据一致性。

3.1.2.1 简易架构图

集群TiDB、 TiKV 和 PD 组件分别部署在 3 个不同的AZ，这是最常规且高可用性最高的方案。

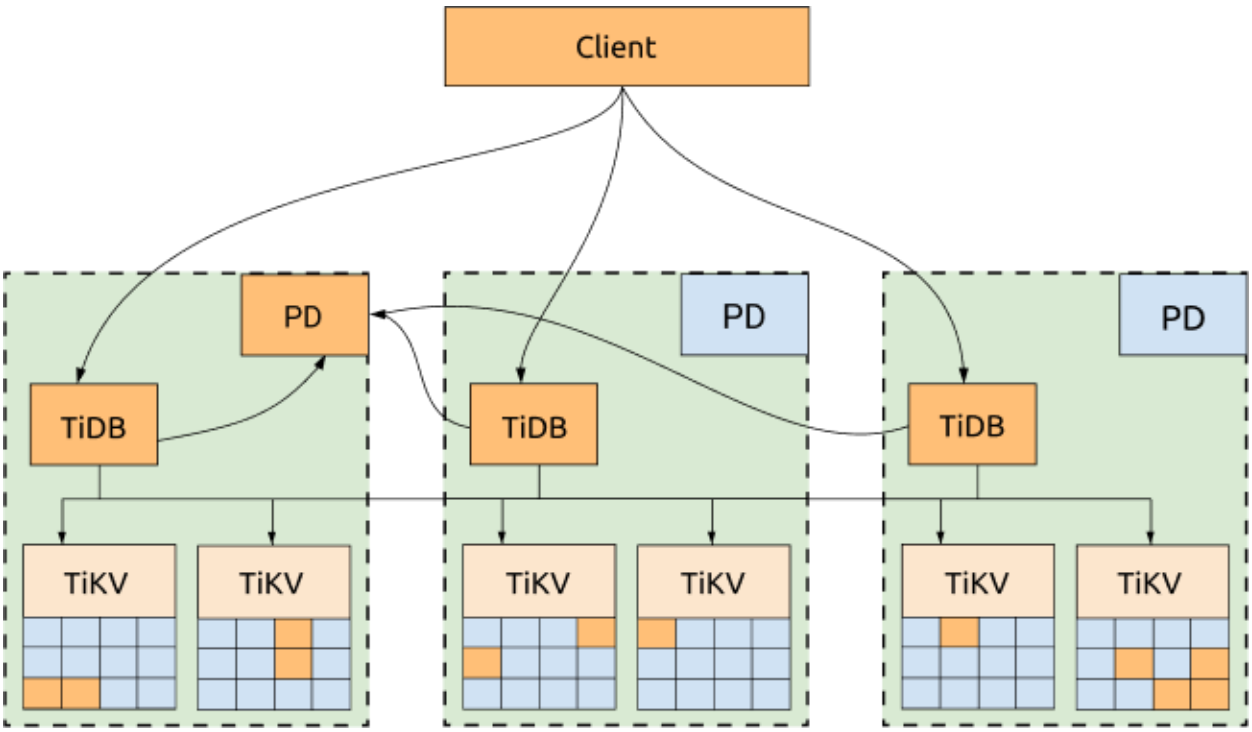


图 1: 三 AZ 部署

优点：

- 所有数据的副本分布在三个AZ，具备高可用和容灾能力
- 任何一个AZ 失效后，不会产生任何数据丢失(RPO = 0)
- 任何一个AZ 失效后，其他两个AZ 会自动发起leader election，并在一定时间内（通常 20s 以内）自动恢复服务

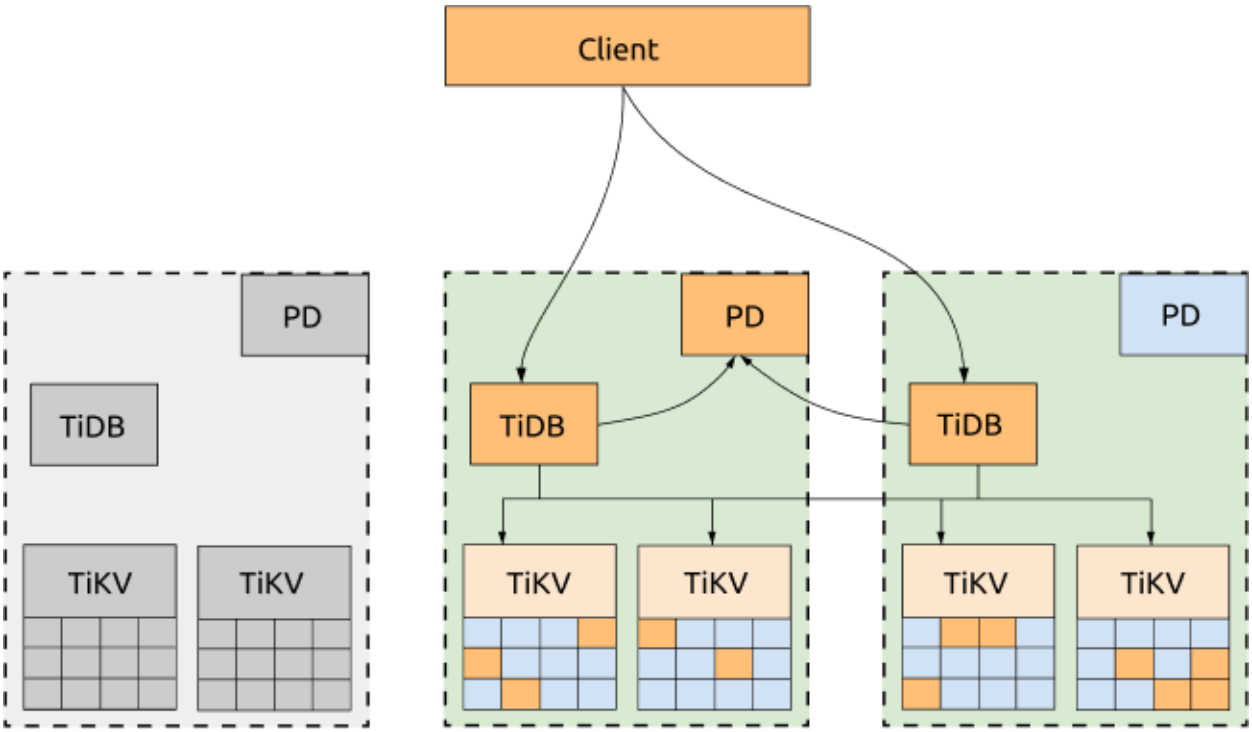


图 2: 三 AZ 部署容灾

缺点：

性能受网络延迟影响。具体影响如下：

- 对于写入的场景，所有写入的数据需要同步复制到至少两个AZ，由于TiDB 写入过程使用两阶段提交，故写入延迟至少需要两倍AZ 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AZ，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AZ 时，TiDB 上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。

3.1.2.2 架构优化图

如果不需要每个AZ 同时对外提供服务，可以将业务流量全部派发到一个AZ，并通过调度策略把Region leader和 PD leader 都迁移到同一个AZ。这样，不管是从 PD 获取TSO，还是读取 Region，都不会受AZ 间网络的影响。当该AZ 失效时，PD leader 和Region leader 会自动在其它AZ 选出，只需要把业务流量转移至其他存活的 AZ 即可。

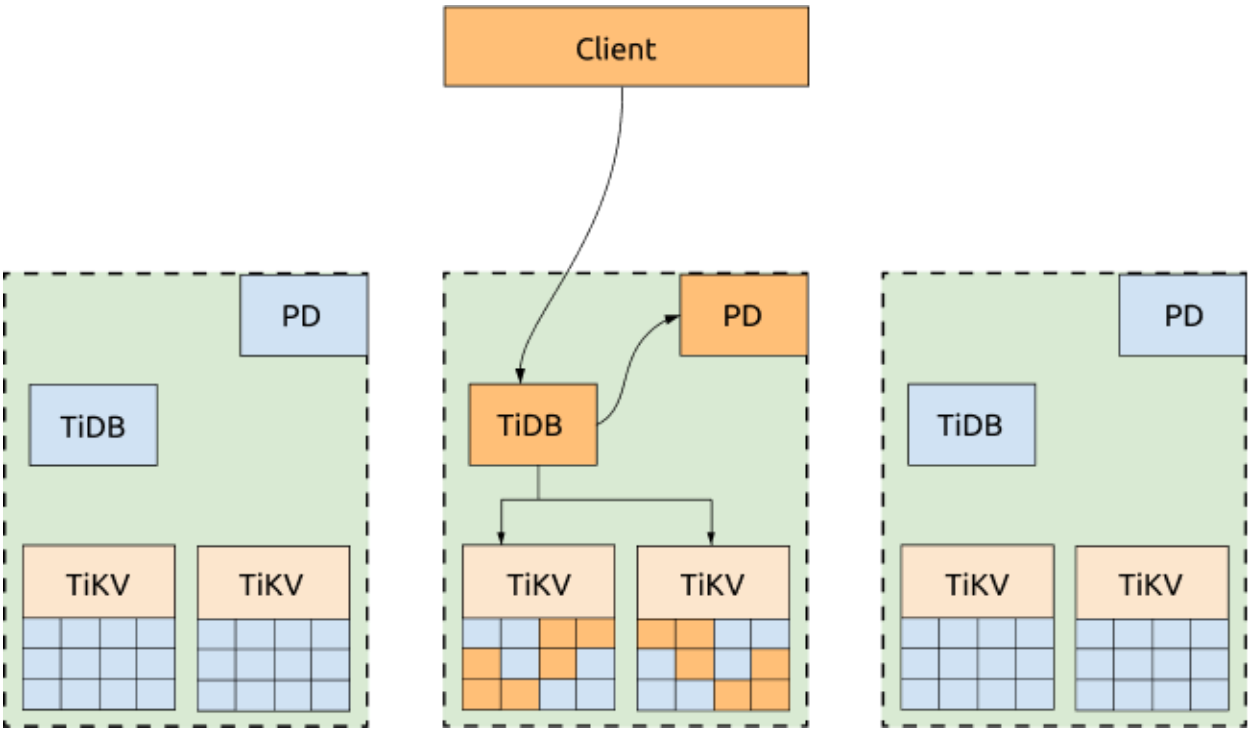


图 3: 三 AZ 部署读性能优化

优点：

集群TSO 获取能力以及读取性能有所提升。具体调度策略设置模板参照如下：

```
-- 其他 AZ 将 leader 驱逐至承载业务流量的 AZ

config set label-property reject-leader LabelName labelValue

-- 迁移 PD leader 并设置优先级
member leader transfer pdName1
member leader_priority pdName1 5
member leader_priority pdName2 4
member leader_priority pdName3 3
```

> 注意：
>
> TiDB 5.2 及以上版本默认不支持 `label-property` 配置。若要设置副本策略，请使用Placement Rules。

缺点：

- 写入场景仍受AZ 网络延迟影响，这是因为遵循Raft 多数派协议，所有写入的数据需要同步复制到至少两个AZ
- TiDB Server 是 AZ 级别单点
- 业务流量纯走单AZ，性能受限于单AZ 网络带宽压力
- TSO 获取能力以及读取性能受限于业务流量AZ 集群 PD、TiKV 组件是否正常，否则仍受跨AZ 网络交互影响

3.1.2.3 样例部署图

3.1.2.3.1 样例拓扑架构

假设某区域有三个AZ，AZ1、AZ2 和 AZ3。每个AZ 中有两套机架，每个机架有三台服务器，不考虑混合布署以及单台机器多实例部署，同区域三AZ 架构集群（3 副本）部署参考如下：

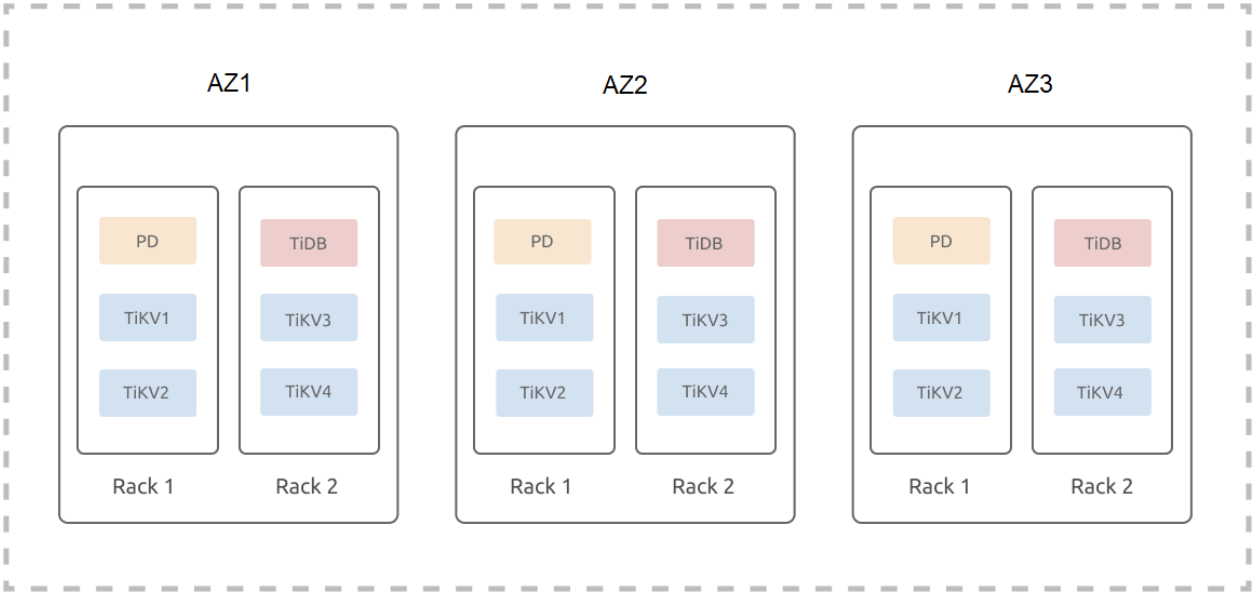


图 4: 同区域三AZ 集群部署

3.1.2.3.2 TiKV Labels 简介

TiKV 是一个 Multi-Raft 系统，其数据按Region（默认 96M）切分，每个Region 的 3 个副本构成了一个Raft Group。假设一个 3 副本TiDB 集群，由于Region 的副本数与TiKV 实例数量无关，则一个Region 的 3 个副本只会被调度到其中 3 个 TiKV 实例上，也就是说即使集群扩容N 个 TiKV 实例，其本质仍是一个 3 副本集群。

由于 3 副本的Raft Group 只能容忍 1 副本故障，当集群被扩容到N 个 TiKV 实例时，这个集群依然只能容忍一个 TiKV 实例的故障。2 个TiKV 实例的故障可能会导致某些Region 丢失多个副本，整个集群的数据也不再完整，访问到这些Region 上的数据的 SQL 请求将会失败。而N 个 TiKV 实例中同时有两个发生故障的概率是远远高于 3 个 TiKV 中同时有两个发生故障的概率的，也就是说 Multi-Raft 系统集群扩容TiKV 实例越多，其可用性是逐渐降低的。

正因为 Multi-Raft TiKV 系统局限性，Labels 标签应运而出，其主要用于描述TiKV 的位置信息。Label 信息随着部署或滚动更新操作刷新到TiKV 的启动配置文件中，启动后的TiKV 会将自己最新的 Label 信息上报给 PD，PD 根据用户登记的 Label 名称（也就是 Label 元信息），结合 TiKV 的拓扑进行Region 副本的最优调度，从而提高系统可用性。

3.1.2.3.3 TiKV Labels 样例规划

针对TiKV Labels 标签，你需要根据已有的物理资源、容灾能力容忍度等方面进行设计与规划，进而提升系统的可用性和容灾能力。并根据已规划的拓扑架构，在集群初始化配置文件中进行配置（此处省略其他非重点项）：

```
server_configs:
  pd:
    replication.location-labels: ["zone","az","rack","host"]

tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r1", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r2", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r2", host: "33" }

  - host: 10.63.10.34
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r1", host: "34" }
  - host: 10.63.10.35
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r1", host: "35" }
  - host: 10.63.10.36
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r2", host: "36" }
  - host: 10.63.10.37
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r2", host: "37" }

  - host: 10.63.10.38
    config:
      server.labels: { zone: "z3", az: "az3", rack: "r1", host: "38" }
  - host: 10.63.10.39
```

```
  config:
    server.labels: { zone: "z3", az: "az3", rack: "r1", host: "39" }
- host: 10.63.10.40
  config:
    server.labels: { zone: "z3", az: "az3", rack: "r2", host: "40" }
- host: 10.63.10.41
  config:
    server.labels: { zone: "z3", az: "az3", rack: "r2", host: "41" }
```

本例中，zone 表示逻辑可用区层级，用于控制副本的隔离（当前集群 3 副本）。

不直接采用az、rack 和 host 三层 Label 结构，是因为考虑到将来可能会扩容AZ，假设新扩容的AZ 编号是AZ2、AZ3 和 AZ4，则只需在对应可用区下扩容AZ，rack 也只需在对应AZ 下扩容。

如果直接采用 AZ、rack 和 host 三层 Label 结构，那么扩容 AZ 操作可能需重新添加 Label，TiKV 数据整体需要 Rebalance。

3.1.2.4 高可用和容灾分析

采用区域多AZ 方案，当任意一个AZ 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，各种调度策略主要用于优化性能，当发生故障时，调度机制总是优先考虑可用性而不是性能。

3.2 双区域多AZ 部署TiDB

本文档简要介绍双区域多可用区(Availability Zone, AZ) 部署的架构模型及配置。

本文中的区域指的是地理隔离的不同位置，AZ 指的是区域内部划分的相互独立的资源集合，本文描述的方案同样适用于在两个城市部署三个数据中心（两地三中心）的场景。

3.2.1 简介

双区域多AZ 架构，即生产数据AZ、同区域灾备AZ、跨区域灾备AZ 的高可用容灾方案。在这种模式下，两个区域的三个AZ 互联互通，如果一个AZ 发生故障或灾难，其他AZ 可以正常运行并对关键业务或全部业务实现接管。相比同区域多AZ 方案，双区域三AZ 具有跨区域级高可用能力，可以应对区域级自然灾害。

TiDB 分布式数据库采用Raft 算法，可以原生支持双区域三AZ 架构，并保证集群数据的一致性和高可用。而且，同区域AZ 网络延迟相对较小，可以把业务流量同时派发到同区域两个AZ，并通过控制Region Leader 和 PD Leader 分布实现同区域AZ 共同负载业务流量。

3.2.2 架构

本文以北京和西安部署集群为例，阐述TiDB 分布式数据库双区域三AZ 架构的部署模型。

假设北京有两个AZ，AZ1 和 AZ2，西安有一个AZ，AZ3。北京同区域两AZ 之间网络延迟低于 3 ms，北京与西安之间的网络使用ISP 专线，延迟约为 20 ms。

下图为集群部署架构图，具体如下：

- 集群采用双区域三AZ部署方式，分别为北京AZ1，北京AZ2，西安AZ3。
- 集群采用 5 副本模式，其中AZ1 和 AZ2 分别放 2 份副本，AZ3 放 1 份副本；TiKV 按机柜设置 Label，即每个机柜上有 1 份副本。
- 副本间通过Raft 协议保证数据的一致性和高可用，对用户完全透明。

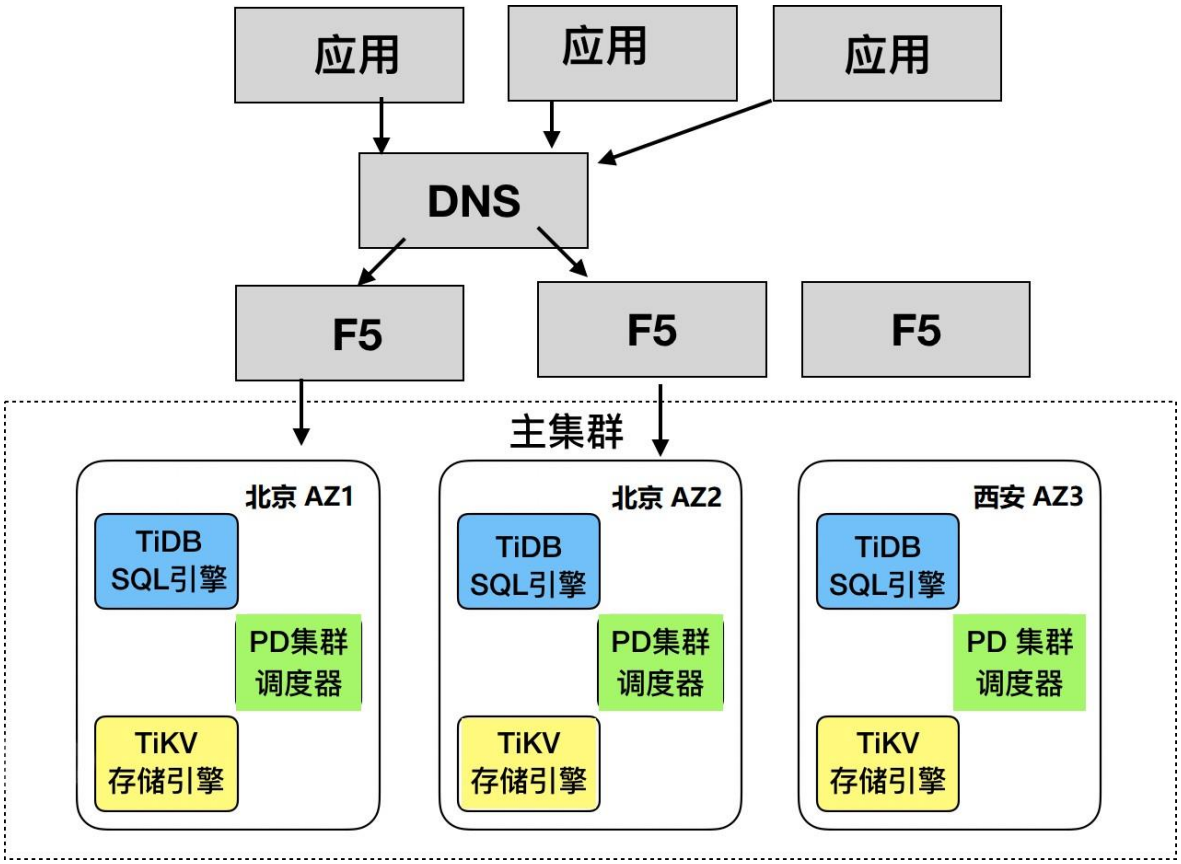


图 5: 双区域三AZ 集群架构图

该架构具备高可用能力，同时通过 PD 调度保证 Region Leader 只出现在同区域的两个 AZ。相比于三 AZ，即 Region Leader 分布不受限制的方案，双区域三AZ 方案有以下优缺点：

- 优点
  - Region Leader 都在同区域AZ，延迟低，数据写入速度更优。
  - 双AZ 可同时对外提供服务，资源利用率更高。
  - 任一AZ 失效后，另一AZ 接管服务，业务可用并且不发生数据丢失。
- 缺点

- 因为数据一致性是基于Raft 算法实现，当同区域两个AZ 同时失效时，因为远程灾备AZ 只剩下一份副本，不满足Raft 算法大多数副本存活的要求。最终将导致集群暂时不可用，需要从一副本恢复集群，丢失少部分还没同步的热数据。这种情况出现概率较小。
- 由于使用了网络专线，该架构下网络设施成本较高。
- 双区域三AZ 需设置 5 副本，数据冗余度增加，空间成本攀升。

3.2.2.1 详细示例

北京、西安双区域三AZ 配置详解：

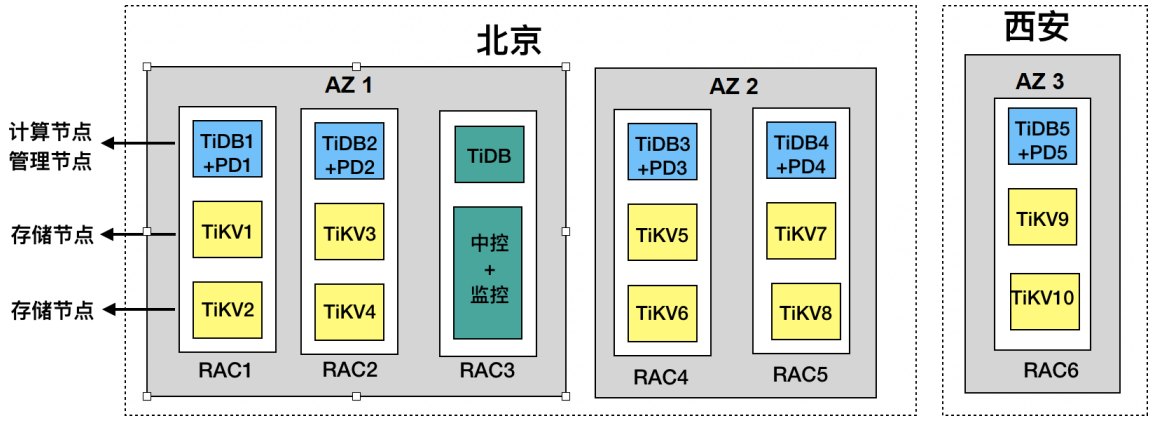


图 6: 双区域三AZ 配置详图

如上图所示，北京有两个可用区AZ1 和AZ2，可用区AZ1 有三套机架rac1、 rac2 和rac3，可用区AZ2 有两套机架rac4 和 rac5；西安可用区AZ3 有一套机架rac6。

AZ1 的 rac1 机架中，一台服务器部署了TiDB 和 PD 服务，另外两台服务器部署了TiKV 服务，其中，每台TiKV 服务器部署了两个TiKV 实例(tikv-server)，rac2、 rac4、 rac5 和 rac6 类似。

机架rac3 上部署了TiDB Server、中控及监控服务器。TiDB Server 用于日常管理维护和备份。中控和监控服务器上部署了 Prometheus、 Grafana 以及恢复工具。

另可增加备份服务器，其上部署 Drainer，Drainer 以输出 file 文件的方式将binlog 数据保存到指定位置，实现增量备份的目的。

3.2.3 配置

3.2.3.1 示例

以下为一个 `tiup topology.yaml` 文件示例：

```
## # Global variables are applied to all deployments and used as the default value of
## # the deployments if a specific deployment value is missing.
global:
  user: "tidb"
```

```
  ssh_port: 22
  deploy_dir: "/data/tidb_cluster/tidb-deploy"
  data_dir: "/data/tidb_cluster/tidb-data"

server_configs:
  tikv:
    server.grpc-compression-type: gzip
  pd:
    replication.location-labels:  ["dc","zone","rack","host"]

pd_servers:
  - host: 10.63.10.10
    name: "pd-10"
  - host: 10.63.10.11
    name: "pd-11"
  - host: 10.63.10.12
    name: "pd-12"
  - host: 10.63.10.13
    name: "pd-13"
  - host: 10.63.10.14
    name: "pd-14"

tidb_servers:
  - host: 10.63.10.10
  - host: 10.63.10.11
  - host: 10.63.10.12
  - host: 10.63.10.13
  - host: 10.63.10.14

tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { az: "1", replication zone: "1", rack: "1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { az: "1", replication zone: "2", rack: "2", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { az: "2", replication zone: "3", rack: "3", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { az: "2", replication zone: "4", rack: "4", host: "33" }
  - host: 10.63.10.34
    config:
      server.labels: { az: "3", replication zone: "5", rack: "5", host: "34" }
```

```
      raftstore.raft-min-election-timeout-ticks:  1000
      raftstore.raft-max-election-timeout-ticks:  1200

monitoring_servers:
  - host: 10.63.10.60

grafana_servers:
  - host: 10.63.10.60

alertmanager_servers:
  - host: 10.63.10.60
```

3.2.3.2 Labels 设计

在双区域三AZ 部署方式下，对于 Labels 的设计需要充分考虑到系统的可用性和容灾能力，建议根据部署的物理结构来定义AZ、 replication zone、 rack 和 host 四个等级。

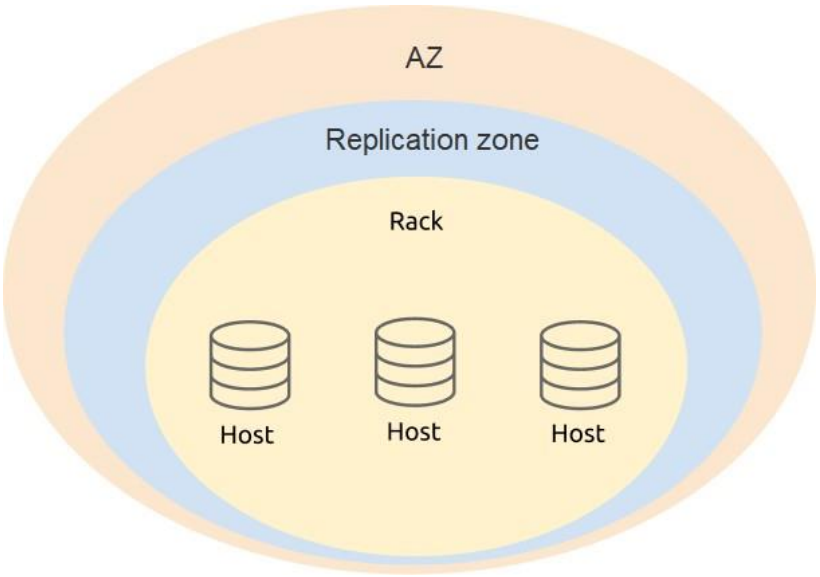


图 7: Label 逻辑定义图

PD 设置中添加TiKV label 的等级配置。

```
server_configs:
  pd:
    replication.location-labels:  ["az","replication zone","rack","host"]
```

tikv_servers 设置基于TiKV 真实物理部署位置的 Label 信息，方便 PD 进行全局管理和调度。

```
tikv_servers:
  - host: 10.63.10.30
    config:
```

```
    server.labels: { az: "1", replication zone: "1", rack: "1", host: "30" }
- host: 10.63.10.31
  config:
    server.labels: { az: "1", replication zone: "2", rack: "2", host: "31" }
- host: 10.63.10.32
  config:
    server.labels: { az: "2", replication zone: "3", rack: "3", host: "32" }
- host: 10.63.10.33
  config:
    server.labels: { az: "2", replication zone: "4", rack: "4", host: "33" }
- host: 10.63.10.34
  config:
    server.labels: { az: "3", replication zone: "5", rack: "5", host: "34" }
```

3.2.3.3 参数配置优化

在双区域三AZ 的架构部署中，从性能优化的角度，除了常规参数配置外，还需要对集群中相关组件参数进行调整。

- 启用TiKV gRPC 消息压缩。由于需要在网络中传输集群数据，可开启 gRPC 消息压缩，降低网络流量。

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

- 优化跨区域AZ3 的TiKV 节点网络，修改TiKV 的如下参数，拉长跨区域副本参与选举的时间，避免跨区域 TiKV 中的副本参与Raft 选举。

```
raftstore.raft-min-election-timeout-ticks: 1000
raftstore.raft-max-election-timeout-ticks: 1200
```

- 调度设置。在集群启动后，通过 `tiup ctl:v<CLUSTER_VERSION> pd` 工具进行调度策略修改。修改TiKV Raft 副本数按照安装时规划好的副本数进行设置，在本例中为 5 副本。

```
config set max-replicas 5
```

- 禁止向跨区域AZ 调度 Raft Leader，当 Raft Leader 在跨区域AZ 时，会造成不必要的本区域AZ 与远程AZ 间的网络消耗，同时，网络带宽和延迟也会对TiDB 的集群性能产生影响。

```
config set label-property reject-leader dc 3
```

> **注意：**
>
> TiDB 5.2 及以上版本默认不支持`label-property` 配置。若要设置副本策略，请使用Placement Rules。

- 设置 PD 的优先级，为了避免出现跨区域AZ 的 PD 成为 Leader，可以将本区域 AZ 的 PD 优先级调高（数字越大，优先级越高），将跨区域的 PD 优先级调低。

```
member leader_priority PD-10 5
member leader_priority PD-11 5
member leader_priority PD-12 5
member leader_priority PD-13 5
member leader_priority PD-14 1
```

3.3 单区域双AZ部署TiDB

本文介绍单区域双可用区(Availability Zone, AZ)的部署模式，包括方案架构、示例配置、副本方法及启用该模式的方法。

本文中的区域指的是地理隔离的不同位置，AZ 指的是区域内部划分的相互独立的资源集合，本文所描述的方案同样适用于同一城市两个数据中心（同城双中心）的场景。

3.3.1 简介

TiDB 通常采用多AZ 部署方案保证集群高可用和容灾能力。多AZ 部署方案包括单区域多AZ 部署模式、双区域多 AZ 部署模式等多种部署模式。本文介绍单区域双AZ 部署方案，即在同一区域部署两个AZ，成本更低，同样能满足高可用和容灾要求。该部署方案采用自适应同步模式，即 Data Replication Auto Synchronous，简称 DR Auto-Sync。

单区域双AZ 部署方案下，两个AZ 通常位于同一个城市或两个相邻城市（例如北京和廊坊），相距50 km 以内，AZ 间的网络连接延迟小于 1.5 ms，带宽大于 10 Gbps。

3.3.2 部署架构

本文以某市为例，市里有两个AZ，AZ1 和 AZ2，AZ1 为主用区域，AZ2 为从属区域，分别位于城东和城西。

下图为集群部署架构图，具体如下：

- 集群采用推荐的 6 副本模式，其中 AZ1 中放 3 个Voter，AZ2 中放 2 个 Follower 副本和 1 个 Learner 副本。TiKV 按机房的实际情况打上合适的 Label。
- 副本间通过Raft 协议保证数据的一致性和高可用，对用户完全透明。

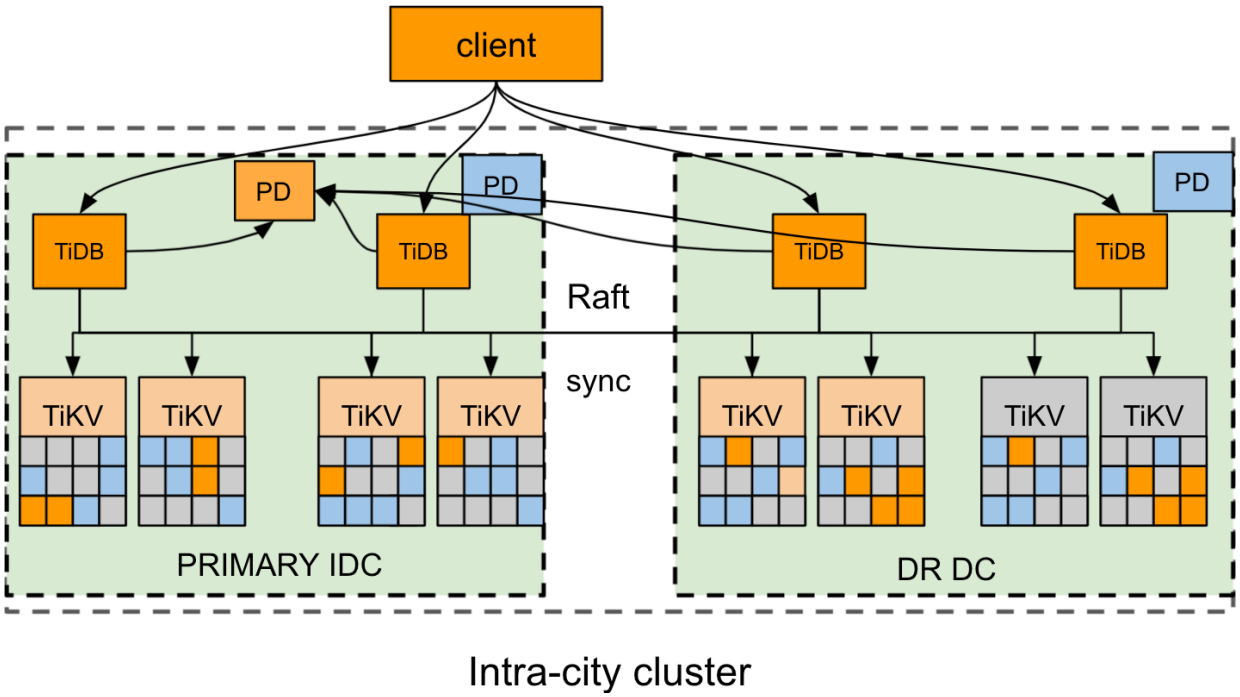


图 8: 单区域双AZ 集群架构图

该部署方案定义了三种状态来控制和标示集群的同步状态，该状态约束了TiKV 的同步方式。集群的复制模式可以自动在三种状态之间自适应切换。要了解切换过程，请参考状态转换。

- sync：同步复制模式，此时从 AZ (DR) 至少有一个副本与主AZ (PRIMARY) 进行同步，Raft 算法保证每条日志按 Label 同步复制到 DR。
- async：异步复制模式，此时不保证 DR 与 PRIMARY 完全同步，Raft 算法使用经典的majority 方式复制日志。
- sync-recover：恢复同步，此时不保证 DR 与 PRIMARY 完全同步，Raft 逐步切换成 Label 复制，切换成功后汇报给 PD。

3.3.3 配置

3.3.3.1 示例

以下 `tiup topology.yaml` 示例拓扑文件为单区域双AZ 典型的拓扑配置：

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

```
    replication.location-labels: ["az","rack","host"]
pd_servers:
  - host: 10.63.10.10
    name: "pd-10"
  - host: 10.63.10.11
    name: "pd-11"
  - host: 10.63.10.12
    name: "pd-12"
tidb_servers:
  - host: 10.63.10.10
  - host: 10.63.10.11
  - host: 10.63.10.12
tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { az: "east", rack: "east-1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { az: "east", rack: "east-2", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { az: "east", rack: "east-3", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { az: "west", rack: "west-1", host: "33" }
  - host: 10.63.10.34
    config:
      server.labels: { az: "west", rack: "west-2", host: "34" }
  - host: 10.63.10.35
    config:
      server.labels: { az: "west", rack: "west-3", host: "35" }
monitoring_servers:
  - host: 10.63.10.60
grafana_servers:
   - host: 10.63.10.60
 alertmanager_servers:
   - host: 10.63.10.60
```

3.3.3.2 Placement Rules 规划

为了按照规划的集群拓扑进行部署，你需要使用Placement Rules 来规划集群副本的放置位置。以 6 副本（3 个 Voter 副本在主AZ，2 个 Follower 副本和 1 个 Learner 副本在从AZ）的部署方式为例，可使用 Placement Rules 进行如下副本配置：

```
cat rule.json
```

```
[
  {
    "group_id": "pd",
    "group_index": 0,
    "group_override": false,
    "rules": [
      {
        "group_id": "pd",
        "id": "az-east",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 3,
        "label_constraints": [
          {
            "key": "az",
            "op": "in",
             "values":
              [ "east"
            ]
          }
        ],
        "location_labels":
          [ "az",
          "rack",
          "host"
        ]
      },
      {
        "group_id": "pd",
        "id": "az-west",
        "start_key": "",
        "end_key": "",
        "role": "follower",
        "count": 2,
        "label_constraints": [
          {
            "key": "az",
            "op": "in",
             "values":
              [ "west"
            ]
          }
        ],
        "location_labels": [
```

```
          "az",
          "rack",
          "host"
        ]
      },
      {
        "group_id": "pd",
        "id": "az-west",
        "start_key": "",
        "end_key": "",
        "role": "learner",
        "count": 1,
        "label_constraints": [
          {
            "key": "az",
            "op": "in",
             "values":
              [ "west"
            ]
          }
        ],
        "location_labels":
          [ "az",
          "rack",
          "host"
        ]
      }
    ]
  }
]
```

如果需要使用 `rule.json` 中的配置，你可以使用以下命令把原有的配置备份到 `default.json` 文件，再使用 `rule.json` 中的配置覆盖原有配置：

```
pd-ctl config placement-rules rule-bundle load --out="default.json"
pd-ctl config placement-rules rule-bundle save --in="rule.json"
```

如果需要回退配置，你可以还原备份的 `default.json` 文件或者手动编写如下的json 文件并将其覆盖到现有的配置文件中：

```
cat default.json
[
  {
    "group_id": "pd",
    "group_index": 0,
    "group_override": false,
```

```
    "rules": [
      {
        "group_id": "pd",
        "id": "default",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 5
      }
    ]
  }
]
```

3.3.3.3 启用自适应同步模式

副本的复制模式由 PD 节点控制。如果要使用 DR Auto-sync 自适应同步模式，需要按照以下任一方法修改 PD 的配置。

- 方法一：先配置 PD 的配置文件，然后部署集群。

```
[replication-mode]
replication-mode = "dr-auto-sync"
[replication-mode.dr-auto-sync]
label-key = "az"
primary = "east"
dr = "west"
primary-replicas = 3
dr-replicas = 2
wait-store-timeout = "1m"
```

- 方法二：如果已经部署了集群，则使用pd-ctl 命令修改 PD 的配置。

```
config set replication-mode dr-auto-sync
config set replication-mode dr-auto-sync label-key az
config set replication-mode dr-auto-sync primary east
config set replication-mode dr-auto-sync dr west
config set replication-mode dr-auto-sync primary-replicas 3
config set replication-mode dr-auto-sync dr-replicas 2
```

配置项说明：

- `replication-mode` 为待启用的复制模式，以上示例中设置为 `dr-auto-sync`。默认使用 majority 算法。
- `label-key` 用于区分不同的AZ，需要和 Placement Rules 相匹配。其中主AZ 为“east”，从AZ 为“west”。
- `primary-replicas` 是主AZ 上 Voter 副本的数量。
- `dr-replicas` 是从AZ 上 Voter 副本的数量。

- `wait-store-timeout` 是当出现网络隔离或者故障时，切换到异步复制模式的等待时间。如果超过这个时间还没恢复，则自动切换到异步复制模式。默认时间为 60 秒。

如果需要检查当前集群的复制状态，可以通过以下API 获取：

```
curl http://pd_ip:pd_port/pd/api/v1/replication_mode/status
```

```
{
  "mode": "dr-auto-sync",
  "dr-auto-sync": {
    "label-key": "az",
    "state": "sync"
  }
}
```

3.3.3.3.1 状态转换

简单来讲，集群的复制模式可以自动在三种状态之间自适应的切换：

- 当集群一切正常时，会进入同步复制模式来最大化地保障灾备AZ 的数据完整性。
- 当AZ 网络断连或灾备AZ 发生整体故障时，在经过一段提前设置好的保护窗口之后，集群会进入异步复制状态，来保障业务的可用性。
- 当 AZ 网络重连或灾备AZ 整体恢复之后，灾备AZ 的 TiKV 节点会重新加入到集群，逐步同步数据并最终转为同步复制模式。

状态转换的细节过程如下：

1. 初始化：集群在初次启动时处于 sync（同步复制）模式，PD 会下发信息给TiKV，所有TiKV 节点会严格按照 sync 模式的要求进行工作。
2. 同步切异步：PD 通过定时检查TiKV 的心跳信息来判断TiKV 是否宕机或断连。如果宕机数超过 Primary 和 DR 各自副本的数量 `primary-replicas` 和 `dr-replicas`，意味着无法完成同步复制，需要切换状态。当宕机时间超过了 `wait-store-timeout` 设定的时间，PD 将集群状态切换成async（异步复制）模式。然后 PD 再将async 状态下发到所有TiKV 节点，TiKV 的复制模式由双AZ 同步方式转为原生的Raft 大多数落实方式 (majority)。
3. 异步切同步：PD 通过定时检查TiKV 的心跳信息来判断TiKV 是否恢复连接，如果宕机数小于 Primary 和 DR 各自副本的数量，意味着可以切回同步。PD 会将集群复制状态先切换至 sync-recover，再将该状态下发给所有TiKV 节点。TiKV 的所有Region 逐步切换成双AZ 同步复制模式，切换成功后通过心跳将状态同步信息给 PD。PD 记录TiKV 上 Region 的状态并统计恢复进度。当TiKV 的所有Region 都完成了同步复制模式的切换，PD 将集群复制状态切换为 sync。

3.3.3.4 灾难恢复

本节介绍单区域双AZ 部署提供的容灾恢复方案。

当处于同步复制状态的集群发生了灾难，可进行 RPO = 0 的数据恢复：

- 如果主AZ发生故障，丢失了大多数Voter副本，但是从AZ有完整的数据，可在从AZ恢复数据。此时需要人工介入，通过专业工具恢复。如需获取支持，请联系 PingCAP 服务与支持。
- 如果从AZ发生故障，丢失了少数Voter副本，能自动切换成async异步复制模式。

当不处于同步复制状态的集群发生了灾难，不能保证满足 RPO = 0 进行数据恢复：

- 如果丢失了大多数Voter副本，需要人工介入，通过专业工具恢复（恢复方式请联系TiDB团队）。

3.4 读取历史数据

3.4.1 使用 Stale Read 功能读取历史数据（推荐）

3.4.1.1 Stale Read 功能的使用场景

本文档介绍 Stale Read 的使用场景。Stale Read 是一种读取历史数据版本的机制，读取TiDB中存储的历史数据版本。通过 Stale Read 功能，你能从指定时间点或时间范围内读取对应的历史数据，从而避免数据同步带来延迟。当使用 Stale Read 时，TiDB 默认会随机选择一个副本来读取数据，因此能利用所有副本。如果你的应用程序不能容忍读到非实时的数据，请勿使用 Stale Read，否则读到的数据可能不是最新成功写入的数据。

3.4.1.1.1 场景描述

- 场景一：如果一个事务仅涉及只读操作，并且一定程度上可容忍牺牲实时性，你可以使用 Stale Read 功能来读取历史数据。由于牺牲了一定的实时性，使用 Stale Read 后，TiDB 可以让请求分发到任一个副本上，使得查询的执行获得更大的吞吐量。特别是在一些小表的查询场景中，如果使用了强一致性读，Leader 可能集中在某一个存储节点上，导致查询压力集中在该节点，成为整个查询的瓶颈。通过 Stale Read，可以提升了查询整体的吞吐能力，从而显著提升查询性能。
- 场景二：在部分跨数据中心部署的场景中，如果使用了强一致性的Follower 读，为了读到的数据与 Leader 上的数据一致，会产生跨数据中心获取 Readindex 来校验的请求，导致整体查询的访问延迟增加。通过使用 Stale Read 功能，可以牺牲一定的实时性，就近访问对应数据所在当前中心的副本，避免跨数据中心的网络延迟，降低整体查询的访问延迟。详情参考在三数据中心下就近读取数据。

3.4.1.1.2 使用方法

TiDB 提供语句级别、会话级别以及全局级别的 Stale Read 使用方式，具体使用方法如下：

- 语句级别：
  - 指定一个精确的时间点（推荐）：如需 TiDB 读取一个时间点上保证全局事务记录一致性的数据并且不破坏隔离级别，你可以指定这个时间点对应的时间戳。要使用该方式，请参阅AS OF TIMESTAMP 语法文档。
  - 指定时间范围：如需TiDB 读取在一个时间范围内尽可能新的数据并且不破坏隔离级别，你可以指定一个时间范围。在指定时间范围内，TiDB 会选择一个合适的时间戳，该时间戳能保证所访问的副本上不存在开始于这个时间戳之前且还没有提交的相关事务，即能保证在所访问的可用副本上可执行读取操作而且不会被阻塞。要使用该方式，请参阅AS OF TIMESTAMP 语法文档和该文档中TIDB_BOUNDED_STALENESS 函数部分的介绍。

- 会话级别：
  - 指定时间范围：在会话级别中，如需TiDB 在后续的查询中读取一个时间范围内尽可能新的数据并且不破坏隔离级别，你可以通过设置一个 session 变量 `tidb_read_staleness` 来指定一个时间范围。要使用该方式，请参阅通过系统变量 `tidb_read_staleness` 读取历史数据。

除此以外，你也可以通过设置系统变量`tidb_external_ts` 来在某一会话或全局范围读取某一时间点前的历史数据。要使用该方式，请参阅通过系统变量 `tidb_external_ts` 读取历史数据。

减少 Stale Read 延时

Stale Read 功能会定期推进TiDB 集群的Resolved TS 时间戳，该时间戳能保证TiDB 读到满足事务一致性的数据。当 Stale Read 使用的时间戳（比如 `AS OF TIMESTAMP '2016-10-08 16:45:26'`）大于 Resolved TS 时，Stale Read 会先触发TiDB 推进Resolved TS，等待推进完成后再读取数据，从而导致延时上升。

通过调整下面TiKV 的配置项，你可以使TiDB 加快Resolved TS 推进，以减少 Stale Read 延时：

```
[resolved-ts]
advance-ts-interval = "20s" # 默认为 20 秒，可适当调小该值以加快 Resolved TS 推进，比如调整为 1
    ↪ 秒。
```

> 注意：
>
> 调小该参数会增加TiKV CPU 使用率和各节点之间的流量。

3.4.1.1.3 限制

当对表的 Stale Read 查询下推到TiFlash 时，如果该表在 Stale Read 所指定的读取时间戳之后执行过 DDL 操作，此查询将会报错。原因是TiFlash 只支持从最新的表结构读取数据。

例如：

```
create table t1(id int);
alter table t1 set tiflash replica 1;
```

一分钟后进行 DDL 操作：

```
alter table t1 add column c1 int not null;
```

然后使用 Stale Read 读取一分钟前的数据：

```
set @@session.tidb_enforce_mpp=1;
select * from t1 as of timestamp NOW() - INTERVAL 1 minute;
```

此时TiFlash 会报错：

```
ERROR 1105 (HY000): other error for mpp stream: From MPP<query:<query_ts:1673950975508472943,
    <—> local_query_id:18, server_id:111947, start_ts:438816196526080000>,task_id:1>: Code: 0, e.
    <—> displayText() = DB::TiFlashException: Table 323 schema version 104 newer than query
    <—> schema version 100, e.what() = DB::TiFlashException,
```

把 Stale Read 指定的读取时间戳改成 DDL 操作完成之后的时间，即可避免该错误。

3.4.1.2 使用 AS OF TIMESTAMP 语法读取历史数据

本文档介绍如何通过 `AS OF TIMESTAMP` 语句使用Stale Read 功能来读取TiDB 历史版本数据，包括具体的操作示例以及历史数据的保存策略。

TiDB 支持通过标准 SQL 接口，即通过 `AS OF TIMESTAMP` SQL 语法的形式读取历史数据，无需特殊的服务器或者驱动器。当数据被更新或删除后，你可以通过 SQL 接口将更新或删除前的数据读取出来。

> **注意：**
>
> 读取历史数据时，即使当前数据的表结构相较于历史数据的表结构已经发生改变，历史数据也会以当时的历史表结构来返回。

3.4.1.2.1 语法方式

你可以通过以下三种方式使用 `AS OF TIMESTAMP` 语法：

- `SELECT ... FROM ... AS OF TIMESTAMP`
- `START TRANSACTION READ ONLY AS OF TIMESTAMP`
- `SET TRANSACTION READ ONLY AS OF TIMESTAMP`

如果你想要指定一个精确的时间点，可在 `AS OF TIMESTAMP` 中使用日期时间和时间函数，日期时间的格式为：“2016-10-08 16:45:26.999”，最小时间精度范围为毫秒，通常可只写到秒，例如“2016-10-08 16:45:26”。你也可以通过 `NOW(3)` 函数获得精确到毫秒的当前时间。如果想读取几秒前的数据，推荐使用例如 `NOW()- INTERVAL <—> 10 SECOND` 的表达式。（推荐）

如果你想要指定一个时间范围，需要使用`TIDB_BOUNDED_STALENESS()` 函数。使用该函数，TiDB 会在指定的时间范围内选择一个合适的时间戳，该时间戳能保证所访问的副本上不存在开始于这个时间戳之前且还没有提交的相关事务，即能保证所访问的可用副本上执行读取操作而且不会被阻塞。用法为 `TIDB_BOUNDED_STALENESS <—> (t1, t2)`，其中 `t1` 和 `t2` 为时间范围的两端，支持使用日期时间和时间函数。

示例如下：

- `AS OF TIMESTAMP '2016-10-08 16:45:26'` 表示读取在 2016 年 10 月 8 日 16 点 45 分 26 秒时最新的数据。
- `AS OF TIMESTAMP NOW()- INTERVAL 10 SECOND` 表示读取 10 秒前最新的数据。
- `AS OF TIMESTAMP TIDB_BOUNDED_STALENESS('2016-10-08 16:45:26', '2016-10-08 16:45:29')` 表示读取在 2016 年 10 月 8 日 16 点 45 分 26 秒到 29 秒的时间范围内尽可能新的数据。

- AS OF TIMESTAMP TIDB_BOUNDED_STALENESS(NOW()- INTERVAL 20 SECOND, NOW()) 表示读取 20 秒前到现在的时间范围内尽可能新的数据。

> 注意：
>
> 除了指定时间戳，AS OF TIMESTAMP 语法最常用使用的方式是读几秒前的数据。如果采用这种方式，推荐读 5 秒以上的历史数据。
>
> 使用 Stale Read 时需要为TiDB 和 PD 节点部署 NTP 服务，防止TiDB 指定的时间戳超过当前最新的 TSO 分配进度（如几秒后的时间戳），或者落后于 GC safe point 的时间戳。当指定的时间戳超过服务范围，TiDB 会返回错误。
>
> 你可以通过调整TiKV 的 advance-ts-interval 配置项提高 Stale Read 数据的时效性（即减少延时）。详情参见减少 Stale Read 延时。

3.4.1.2.2 示例

本节通过多个示例介绍 AS OF TIMESTAMP 语法的不同使用方法。在本节中，先介绍如何准备用于恢复的数据，再分别展示如何通过 SELECT、START TRANSACTION READ ONLY AS OF TIMESTAMP、SET TRANSACTION READ ONLY AS OF TIMESTAMP 使用 AS OF TIMESTAMP。

准备数据

在准备数据阶段，创建一张表，并插入若干行数据：

```
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)
```

查看表中的数据：

```
select * from t;
```

```
+----------+
| c    |
+----------+
|    1 |
|    2 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

查看当前时间：

```
select now();
```

```
+---------------------------------+
| now()                           |
+---------------------------------+
| 2021-05-26 16:45:26 |
+---------------------------------+
1 row in set (0.00 sec)
```

更新某一行数据：

```
update t set c=22 where c=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

确认数据已经被更新：

```
select * from t;
```

```
+----------+
| c    |
+----------+
|    1 |
|   22 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

通过 SELECT 读取历史数据

通过SELECT ... FROM ... AS OF TIMESTAMP 语句读取一个基于历史时间的数据。

```
select * from t as of timestamp '2021-05-26 16:45:26';
```

```
+----------+
| c    |
+----------+
|    1 |
|    2 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

> **注意：**
>
> 通过 SELECT 语句读取多个表时要保证TIMESTAMP EXPRESSION 是一致的。比如：`select * from t as of timestamp NOW()- INTERVAL 2 SECOND, c as of timestamp NOW()- INTERVAL 2 SECOND;`。此外，在 SELECT 语句中，你必须要指定相关数据表的as of 信息，若不指定，SELECT 语句会默认读最新的数据。

通过 START TRANSACTION READ ONLY AS OF TIMESTAMP 读取历史数据

通过START TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以开启一个基于历史时间的只读事务，该事务基于所提供的历史时间来读取历史数据。

```
start transaction read only as of timestamp '2021-05-26 16:45:26';
```

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

```
select * from t;
```

```
+----------+
| c    |
+----------+
|    1 |
|    2 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

```
commit;
```

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

当事务结束后，即可读取最新数据。

```
select * from t;
```

```
+----------+
| c    |
+----------+
|    1 |
|   22 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

> 注意：
>
> 通过 START TRANSACTION READ ONLY AS OF TIMESTAMP 开启的事务为只读事务。假如在该事务中执行写入操作，操作将会被该事务拒绝。

通过 SET TRANSACTION READ ONLY AS OF TIMESTAMP 读取历史数据

通过SET TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以将下一个事务设置为基于指定历史时间的只读事务。该事务将会基于所提供的历史时间来读取历史数据。

```
set transaction read only as of timestamp '2021-05-26 16:45:26';
```

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

```
begin;
```

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

```
select * from t;
```

```
+----------+
| c    |
+----------+
|    1 |
|    2 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

```
commit;
```

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

当事务结束后，即可读取最新数据。

```
select * from t;
```

```
+----------+
| c    |
+----------+
|    1 |
|   22 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

> 注意：
>
> 通过 SET TRANSACTION READ ONLY AS OF TIMESTAMP 开启的事务为只读事务。假如在该事务中执行写入操作，操作将会被该事务拒绝。

3.4.1.3 通过系统变量 tidb_read_staleness 读取历史数据

为支持读取历史版本数据，TiDB 从 5.4 版本起引入了一个新的系统变量 tidb_read_staleness。本文档介绍如何通过该系统变量读取历史数据，其中包括具体的操作流程。

3.4.1.3.1 功能介绍

系统变量 tidb_read_staleness 用于设置当前会话允许读取的历史数据范围，其数据类型为int，作用域为 SESSION。设置该变量后，TiDB 会从参数允许的范围内选出一个尽可能新的时间戳，并影响后继的所有读操作。比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证TiKV 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

开启 tidb_read_staleness 后，你仍可以进行以下操作：

- 在当前会话中插入、修改、删除数据或进行 DML 操作。这些语句不会受到 tidb_read_staleness 的影响。
- 在当前会话开启交互式事务。在该事务内的查询依旧是读取最新版本的数据。

完成对历史版本数据的读取后，你可以通过以下两种方式来读取最新版本的数据。

- 结束当前会话。
- 使用 SET 语句，把 tidb_read_staleness 变量的值设为 ""。

> 注意：
>
> 你可以通过调整TiKV 的 advance-ts-interval 配置项提高 Stale Read 数据的时效性（即减少延时），详情参见减少 Stale Read 延时。

3.4.1.3.2 示例

本节通过具体操作示例介绍系统变量 tidb_read_staleness的使用方法。

1. 初始化阶段。创建一个表后，在表中插入几行数据：

```
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. 更新某一行数据：

```
update t set c=22 where c=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

4. 确认数据已经被更新：

```
select * from t;
```

```
+---------+
| c       |
+---------+
|       1 |
|      22 |
|       3 |
+---------+
3 rows in set (0.00 sec)
```

5. 设置一个特殊的环境变量 tidb_read_staleness。

   该变量的作用域为 SESSION，设置变量值后，TiDB 会读取变量值时间之前的最新一个版本的数据。

   以下设置表示TiDB 会从 5 秒前至现在的时间范围内选择一个尽可能新的时间戳，将其用作为历史数据读取的时间戳：

```
set @@tidb_read_staleness="-5";
```

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

> **注意：**
>
> 必须在 `tidb_read_staleness` 前使用 `@@`，而非 `@`。因为 `@@` 表示系统变量，`@` 则表示用户变量。你需要根据第 3 步到第 4 步所花费的时间，来设定你要读取的历史时间范围，即 `tidb_read_staleness` 的值。否则，查询结果中显示的会是最新数据，而非历史数据。因此，请根据自己的实际操作情况调整该时间范围。比如，在本示例中，由于设定的时间范围是 5 秒，你需要在 5 秒内完成第 3 步和第 4 步。

这里读取到的内容即为更新前的数据，也就是历史版本的数据：

```
select * from t;
```

```
+---------+
| c       |
+---------+
|       1 |
|       2 |
|       3 |
+---------+
3 rows in set (0.00 sec)
```

6. 清空这个变量后，即可读取最新版本数据：

```
set @@tidb_read_staleness="";
```

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

```
select * from t;
```

```
+---------+
| c       |
+---------+
|       1 |
|      22 |
|       3 |
+---------+
3 rows in set (0.00 sec)
```

3.4.1.4 通过系统变量 `tidb_external_ts` 读取历史数据

为了支持读取历史版本数据，TiDB 从 v6.4.0 起引入了一个新的系统变量`tidb_external_ts`。本文档介绍如何通过该系统变量读取历史数据，其中包括具体的操作流程。

3.4.1.4.1 场景介绍

通过配置让TiDB 能够读取某一固定时间点的历史数据对于TiCDC 等数据同步工具非常有用。在数据同步工具完成了某一时间点前的数据同步之后，可以通过设置下游TiDB 的 tidb_external_ts 系统变量，使得下游TiDB 的请求能够读取到该时间点前的数据。这将避免在同步过程中，下游TiDB 读取到尚未完全同步而不一致的数据。

3.4.1.4.2 功能介绍

系统变量tidb_external_ts 用于指定启用 tidb_enable_external_ts_read 时，读取历史数据使用的时间戳。

系统变量tidb_enable_external_ts_read 控制着是否在当前会话或全局启用读取历史数据的功能。默认值为 OFF，这意味着该功能关闭，并且设置 tidb_external_ts 没有作用。当该变量被全局地设置为 ON 时，所有的请求都将读取到 tidb_external_ts 指定时间之前的历史数据。如果 tidb_enable_external_ts_read 仅在某一会话被设置为 ON，则只有该会话中的请求会读取到历史数据。

当 tidb_enable_external_ts_read 被设置为 ON 时，TiDB 会进入只读模式，任何写请求都会失败并且返回错误 ERROR 1836 (HY000): Running in read-only mode。

3.4.1.4.3 示例

以下是一个使用该功能的示例：

1. 创建一个表后，在表中插入几行数据：

```
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. 将 tidb_external_ts 设置为 @@tidb_current_ts：

```
START TRANSACTION;
SET GLOBAL tidb_external_ts = @@tidb_current_ts;
COMMIT;
```

4. 插入新的一行并确认新的一行已经被插入：

```
INSERT INTO t VALUES (4);
```

```
Query OK, 1 row affected (0.001 sec)
```

```
SELECT * FROM t;
```

```
+----------+
| id   |
+----------+
|    1 |
|    2 |
|    3 |
|    4 |
+----------+
4 rows in set (0.00 sec)
```

5. 将 tidb_enable_external_ts_read 设置为 ON 后，再次查询表中的数据：

```
SET tidb_enable_external_ts_read = ON;
SELECT * FROM t;
```

```
+----------+
| c    |
+----------+
|    1 |
|    2 |
|    3 |
+----------+
3 rows in set (0.00 sec)
```

因为tidb_external_ts 被设置为插入这一行之前的时间，在启动tidb_enable_external_ts_read 后，将读取不到新插入的行。

3.4.2 通过系统变量tidb_snapshot 读取历史数据

本文档介绍如何通过系统变量 tidb_snapshot 读取历史数据，包括具体的操作流程以及历史数据的保存策略。

> 注意：
>
> 你还可以使用Stale Read 功能读取历史数据。更推荐使用 Stale Read 读取历史数据。

3.4.2.1 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。

> 注意：
>
> 读取历史数据时，即使当前数据的表结构相较于历史数据的表结构已经发生改变，历史数据也会使用当时的表结构来返回数据。

3.4.2.2 操作流程

为支持读取历史版本数据，TiDB 引入了一个新的系统变量tidb_snapshot：

- 这个变量的作用域为 SESSION。
- 你可以通过标准的 SET 语句修改这个变量的值。
- 这个变量的数据类型为文本类型，能够存储TSO 和日期时间。TSO 是从 PD 端获取的全局授时的时间戳，日期时间的格式为：“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 会按照设置的时间戳建立 Snapshot（没有开销，只是创建数据结构），随后所有的 SELECT 操作都会从这个 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。

3.4.2.3 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：

- 使用系统变量tidb_gc_life_time 可以配置历史版本的保留时间（默认值是 10m0s）。
- 使用 SQL 语句 SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point' 可以查询当前的 safePoint，即当前可以读的最旧的快照。在每次 GC 开始运行时，safePoint 将自动更新。

3.4.2.4 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。

3.4.2.5 历史数据恢复策略

在恢复历史版本的数据之前，需要确保在对数据进行操作时，垃圾回收机制(GC) 不会清除历史数据。如下所示，可以通过设置 tidb_gc_life_time 变量来调整GC 清理的周期。不要忘记在恢复历史数据后将该变量设置回之前的值。

```
SET GLOBAL tidb_gc_life_time="60m";
```

> **注意：**
>
> 将 GC life time 从默认的 10 分钟增加到半小时及以上，会导致同一行保留有多个版本并占用更多的磁盘空间，也可能会影响某些操作的性能，例如扫描。进行扫描操作时，TiDB 读取数据需要跳过这些有多个版本的同一行，从而影响到扫描性能。

如果想要恢复历史版本的数据，可以使用以下任意一种方法进行设置：

- 对于简单场景，在设置 tidb_snapshot 变量后使用 SELECT 语句并复制粘贴输出结果，或者使用 SELECT <—> ... INTO LOCAL OUTFLE 语句并使用 LOAD DATA 语句来导入数据。
- 使用Dumpling 导出TiDB 的历史数据快照。Dumpling 在导出较大的数据集时有较好的性能。

3.5 最佳实践

3.5.1 TiDB 最佳实践

本文档总结使用TiDB 时的一些最佳实践，主要涉及 SQL 使用和 OLAP/OLTP 优化技巧，特别是一些TiDB 专有的优化开关。

建议先阅读讲解TiDB 原理的三篇文章（讲存储，说计算，谈调度），再来看这篇文章。

3.5.1.1 前言

数据库是一个通用的基础组件，在开发过程中会考虑到多种目标场景，在具体的业务场景中，需要根据业务的实际情况对数据的参数或者使用方式进行调整。

TiDB 是一个兼容 MySQL 协议和语法的分布式数据库，但是由于其内部实现，特别是支持分布式存储以及分布式事务，使得一些使用方法和 MySQL 有所区别。

3.5.1.2 基本概念

TiDB 的最佳实践与其实现原理密切相关，建议读者先了解一些基本的实现机制，包括Raft、分布式事务、数据分片、负载均衡、SQL 到 KV 的映射方案、二级索引的实现方法、分布式执行引擎。下面会做一点简单的介绍，更详细的信息可以参考 PingCAP 公众号以及知乎专栏的一些文章。

3.5.1.2.1 Raft

Raft 是一种一致性协议，能提供强一致的数据复制保证，TiDB 最底层用Raft 来同步数据。每次写入都要写入多数副本，才能对外返回成功，这样即使丢掉少数副本，也能保证系统中还有最新的数据。比如最大 3 副本的话，每次写入 2 副本才算成功，任何时候，只丢失一个副本的情况下，存活的两个副本中至少有一个具有最新的数据。

相比 Master-Slave 方式的同步，同样是保存三副本，Raft 的方式更为高效，写入的延迟取决于最快的两个副本，而不是最慢的那个副本。所以使用Raft 同步的情况下，异地多活成为可能。在典型的两地三中心场景下，每次写入只需要本数据中心以及离得近的一个数据中心写入成功就能保证数据的一致性，而并不需要三个数据中心都写成功。但是这并不意味着在任何场景都能构建跨机房部署的业务，当写入量比较大时候，机房之间的带宽和延迟成为关键因素，如果写入速度超过机房之间的带宽，或者是机房之间延迟过大，整个Raft 同步机制依然无法很好的运转。

3.5.1.2.2 分布式事务

TiDB 提供完整的分布式事务，事务模型是在Google Percolator 的基础上做了一些优化。具体的实现可以参考《Percolator 和TiDB 事务算法》这篇文章。本文档只讨论以下几点：

- 乐观锁

  TiDB 的乐观事务模型，只有在真正提交的时候，才会做冲突检测。如果有冲突，则需要重试。这种模型在冲突严重的场景下，会比较低效，因为重试之前的操作都是无效的，需要重复做。举一个比较极端的例子，就是把数据库当做计数器用，如果访问的并发度比较高，那么一定会有严重的冲突，导致大量的重试甚至是超时。但是如果访问冲突并不十分严重，那么乐观锁模型具备较高的效率。在冲突严重的场景下，推荐使用悲观锁，或在系统架构层面解决问题，比如将计数器放在Redis 中。

- 悲观锁

  TiDB 的悲观事务模式，悲观事务的行为和 MySQL 基本一致，在执行阶段就会上锁，先到先得，避免冲突情况下的重试，可以保证有较多冲突的事务的成功率。悲观锁同时解决了希望通过 `select for update` 对数据提前锁定的场景。但如果业务场景本身冲突较少，乐观锁的性能会更有优势。

- 事务大小限制

  由于分布式事务要做两阶段提交，并且底层还需要做Raft 复制，如果一个事务非常大，会使得提交过程非常慢，并且会卡住下面的Raft 复制流程。为了避免系统出现被卡住的情况，我们对事务的大小做了限制：

  - 单个事务包含的 SQL 语句不超过 5000 条（默认）
    * 单条 KV entry 不超过 6MB（默认）
    * KV entry 的总大小不超过 10G

  在 Google 的 Cloud Spanner 上面，也有类似的限制。

3.5.1.2.3 数据分片

TiKV 自动将底层数据按照 Key 的Range 进行分片。每个Region 是一个 Key 的范围，从 `StartKey` 到 EndKey 的左闭右开区间。Region 中的 Key-Value 总量超过一定值，就会自动分裂。这部分用户不需要担心。

3.5.1.2.4 负载均衡

PD 会根据整个TiKV 集群的状态，对集群的负载进行调度。调度是以Region 为单位，以 PD 配置的策略为调度逻辑，自动完成。

3.5.1.2.5 SQL on KV

TiDB 自动将 SQL 结构映射为 KV 结构。具体的可以参考《三篇文章了解TiDB 技术内幕- 说计算》这篇文档。简单来说，TiDB 执行了以下操作：

- 一行数据映射为一个 KV，Key 以 `TableID` 构造前缀，以行ID 为后缀
- 一条索引映射为一个 KV，Key 以 `TableID+IndexID` 构造前缀，以索引值构造后缀

可以看到，对于一个表中的数据或者索引，会具有相同的前缀，这样在TiKV 的 Key 空间内，这些 Key-Value 会在相邻的位置。那么当写入量很大，并且集中在一个表上面时，就会造成写入的热点，特别是连续写入的数据中某些索引值也是连续的（比如update time 这种按时间递增的字段），会在很少的几个 Region 上形成写入热点，成为整个系统的瓶颈。同样，如果所有的数据读取操作也都集中在很小的一个范围内（比如在连续的几万或者十几万行数据上），那么可能造成数据的访问热点。

3.5.1.2.6 二级索引

TiDB 支持完整的二级索引，并且是全局索引，很多查询可以通过索引来优化。如果利用好二级索引，对业务非常重要，很多 MySQL 上的经验在TiDB 这里依然适用，不过TiDB 还有一些自己的特点，需要注意，这一节主要讨论在TiDB 上使用二级索引的一些注意事项。

- 二级索引是否越多越好

  二级索引能加速查询，但是要注意新增一个索引是有副作用的。上一节介绍了索引的存储模型，那么每增加一个索引，在插入一条数据的时候，就要新增一个 Key-Value，所以索引越多，写入越慢，并且空间占用越大。另外过多的索引也会影响优化器运行时间，并且不合适的索引会误导优化器。所以索引并不是越多越好。

- 对哪些列建索引比较合适

  上文提到，索引很重要但不是越多越好，因此需要根据具体的业务特点创建合适的索引。原则上需要对查询中需要用到的列创建索引，目的是提高性能。下面几种情况适合创建索引：

  - 区分度比较大的列，通过索引能显著地减少过滤后的行数
  - 有多个查询条件时，可以选择组合索引，注意需要把等值条件的列放在组合索引的前面

  这里举一个例子，假设常用的查询是 `select * from t where c1 = 10 and c2 = 100 and c3 > 10`，那么可以考虑建立组合索引 `Index cidx (c1, c2, c3)`，这样可以用查询条件构造出一个索引前缀进行 Scan。

- 通过索引查询和直接扫描 Table 的区别

  TiDB 实现了全局索引，所以索引和 Table 中的数据并不一定在一个数据分片上。通过索引查询的时候，需要先扫描索引，得到对应的行ID，然后通过行ID 去取数据，所以可能会涉及到两次网络请求，会有一定的性能开销。

  如果查询涉及到大量的行，那么扫描索引是并发进行，只要第一批结果已经返回，就可以开始去取 Table 的数据，所以这里是一个并行+ Pipeline 的模式，虽然有两次访问的开销，但是延迟并不会很大。

  以下情况不会涉及到两次访问的问题：

  - 索引中的列已经满足了查询需求。比如 Table t 上面的列c 有索引，查询是 `select c from t where c > 10;`，这个时候，只需要访问索引，就可以拿到所需要的全部数据。这种情况称之为覆盖索引(Covering Index)。所以如果很关注查询性能，可以将部分不需要过滤但是需要在查询结果中返回的列放入索引中，构造成组合索引，比如这个例子：`select c1, c2 from t where c1 > 10;`，要优化这个查询可以创建组合索引 `Index c12 (c1, c2)`。
  - 表的 Primary Key 是整数类型。在这种情况下，TiDB 会将 Primary Key 的值当做行ID，所以如果查询条件是在 PK 上面，那么可以直接构造出行ID 的范围，直接扫描 Table 数据，获取结果。

- 查询并发度

  数据分散在很多Region 上，所以 TiDB 在做查询的时候会并发进行，默认的并发度比较保守，因为过高的并发度会消耗大量的系统资源，且对于 OLTP 类型的查询，往往不会涉及到大量的数据，较低的并发度已经可以满足需求。对于 OLAP 类型的 Query，往往需要较高的并发度。所以 TiDB 支持通过 System Variable 来调整查询并发度。

  - tidb_distsql_scan_concurrency

  在进行扫描数据的时候的并发度，这里包括扫描 Table 以及索引数据。

  - tidb_index_lookup_size

  如果是需要访问索引获取行ID 之后再访问 Table 数据，那么每次会把一批行ID 作为一次请求去访问 Table 数据，这个参数可以设置 Batch 的大小，较大的 Batch 会使得延迟增加，较小的 Batch 可能会造成更多的查询次数。这个参数的合适大小与查询涉及的数据量有关。一般不需要调整。

- tidb_index_lookup_concurrency

  如果是需要访问索引获取行ID 之后再访问 Table 数据，每次通过行ID 获取数据时候的并发度通过这个参数调节。

- 通过索引保证结果顺序

  索引除了可以用来过滤数据之外，还能用来对数据排序，首先按照索引的顺序获取行ID，然后再按照行 ID 的返回顺序返回行的内容，这样可以保证返回结果按照索引列有序。前面提到了扫索引和获取 Row 之间是并行+ Pipeline 模式，如果要求按照索引的顺序返回Row，那么这两次查询之间的并发度设置的太高并不会降低延迟，所以默认的并发度比较保守。可以通过tidb_index_serial_scan_concurrency 变量进行并发度调整。

- 逆序索引

  目前TiDB 支持对索引进行逆序 Scan，目前速度比顺序 Scan 慢一些，通常情况下慢 20%，在数据频繁修改造成版本较多的情况下，会慢的更多。如果可能，建议避免对索引的逆序 Scan。

3.5.1.3 场景与实践

上一节我们讨论了一些TiDB 基本的实现机制及其对使用带来的影响，本节我们从具体的使用场景出发，谈一些更为具体的操作实践。我们以从部署到支撑业务这条链路为序，进行讨论。

3.5.1.3.1 部署

在部署之前请务必阅读TiDB 部署建议以及对硬件的需求。

推荐通过TiUP 部署TiDB 集群，这个工具可以部署、停止、销毁、升级整个集群，非常方便易用。非常不推荐手动部署，后期的维护和升级会很麻烦。

3.5.1.3.2 导入数据

为了提高导入数据期间的写入性能，可以对TiKV 的参数进行调优，具体的文档查看TiKV 性能参数调优。

3.5.1.3.3 写入

上面提到了TiDB 对单个事务的大小有限制，这层限制是在 KV 层面，反映在 SQL 层面的话，简单来说一行数据会映射为一个 KV entry，每多一个索引，也会增加一个 KV entry。

> 注意：
>
> 对事务的大小限制，要考虑TiDB 做编码以及事务额外 Key 的开销，在使用的时候，建议每个事务的行数不超过 200 行，且单行数据小于 100k，否则可能性能不佳。

建议无论是Insert，Update 还是 Delete 语句，都通过分 Batch 或者是加 Limit 的方式限制。

在删除大量数据的时候，建议使用 `Delete from t where xx limit 5000;` 这样的方案，通过循环来删除，用 `Affected Rows == 0` 作为循环结束条件。

如果一次删除的数据量非常大，这种循环的方式会越来越慢，因为每次删除都是从前向后遍历，前面的删除之后，短时间内会残留不少删除标记（后续会被GC 清理掉），影响后面的 Delete 语句。如果有可能，建议把 Where 条件细化。举个例子，假设要删除 2017-05-26 当天的所有数据，那么可以这样做：

```
for i from 0 to 23:
    while affected_rows > 0:
        delete from t where insert_time >= i:00:00 and insert_time < (i+1):00:00 limit 5000;
        affected_rows = select affected_rows()
```

上面是一段伪代码，意思就是要把大块的数据拆成小块删除，以避免删除过程中前面的 Delete 语句影响后面的 Delete 语句。

3.5.1.3.4 查询

看业务的查询需求以及具体的语句，可以参考TiDB 专用系统变量和语法这篇文档。可以通过 SET 语句控制 SQL 执行的并发度，另外通过 Hint 控制Join 物理算子选择。

另外 MySQL 标准的索引选择 Hint 语法，也可以用，通过 Use Index/Ignore Index hint 控制优化器选择索引。

如果是个 OLTP 和 OLAP 混合类型的业务，可以把 TP 请求和AP 请求发送到不同的tidb-server 上，这样能够减小 AP 业务对于 TP 业务的影响。承载AP 业务的tidb-server 推荐使用高配的机器，比如 CPU 核数比较多，内存比较大。

但彻底的隔离 OLTP 和 OLAP，推荐将 OLAP 的业务跑在TiFlash 上。TiFlash 是列存引擎，在 OLAP 的分析查询场景上，性能极具亮点，TiFlash 可以在存储层上做到物理隔离，并可做到一致性读取。

3.5.1.3.5 监控和日志

Metrics 系统是了解系统状态的最佳方法，建议所有的用户都部署监控系统。

TiDB 使用Grafana + Prometheus 监控系统状态。如果使用TiUP 部署集群，那么会自动部署和配置监控系统。

监控系统中的监控项很多，大部分是给TiDB 开发者查看的内容，如果没有对源代码比较深入的了解，并没有必要了解这些监控项。我们会精简出一些和业务相关或者是系统关键组件状态相关的监控项，放在一个独立的 overview 面板中，供用户使用。

除了监控之外，查看日志也是了解系统状态的常用方法。TiDB 的三个组件tidb-server/tikv-server/pd-server 都有一个 --log-file 的参数。如果启动的时候设置了这个参数，那么日志会保存着参数所设置的文件的位置，另外会自动的按天对 Log 文件做归档。如果没有设置 --log-file 参数，日志会输出在 stderr 中。

从 4.0 版本开始，从解决易用性的角度出发，提供了TiDB Dashboard UI 系统，通过浏览器访问 http://PD_IP:<⟷ PD_PORT/dashboard 即可打开TiDB Dashboard。TiDB Dashboard 可以提供集群状态、性能分析、流量可视化、SQL 诊断、日志搜索等功能。

3.5.1.3.6 文档

了解一个系统或者解决使用中的问题最好的方法是阅读文档，明白实现原理。TiDB 有大量的官方文档，希望大家在遇到问题的时候能先尝试通过文档或者搜索Issue list 寻找解决方案。官方文档查看docs-cn。如果希望阅读英文文档，可以查看docs。

其中的FAQ 和故障诊断章节建议大家仔细阅读。另外TiDB 还有一些不错的工具，也有配套的文档，具体的见各项工具的GitHub 页面。

除了文档之外，还有很多不错的文章介绍TiDB 的各项技术细节内幕，大家可以关注下面这些文章发布渠道：

- 公众号：微信搜索 PingCAP
- 知乎专栏：TiDB 的后花园
- 官方博客

3.5.1.4 TiDB 的最佳适用场景

简单来说，TiDB 适合具备下面这些特点的场景：

- 数据量大，单机保存不下
- 不希望做 Sharding 或者懒得做 Sharding
- 访问模式上没有明显的热点
- 需要事务、需要强一致、需要灾备
- 希望Real-Time HTAP，减少存储链路

3.5.2 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。

3.5.2.1 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。

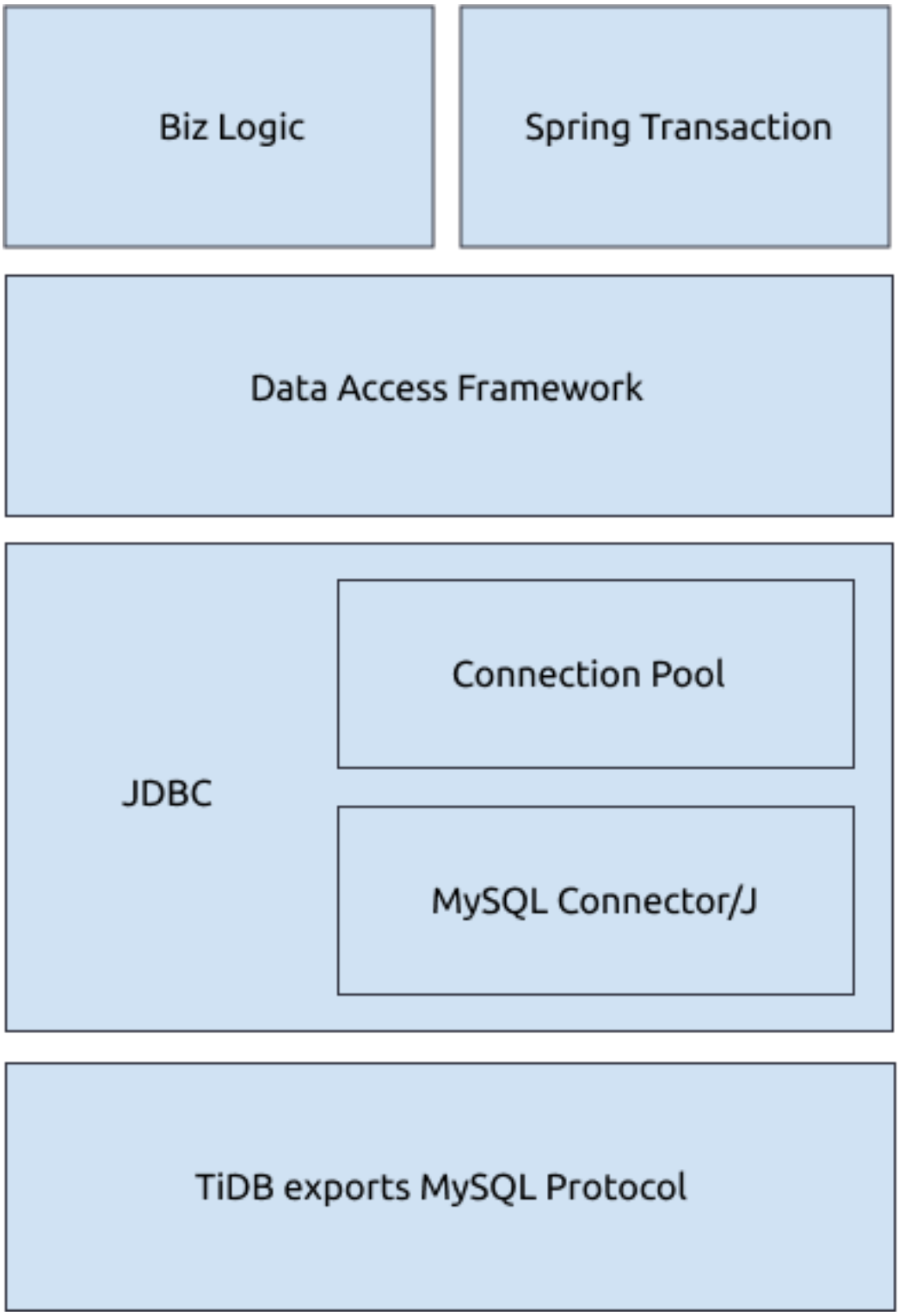


图 9: Java Component

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。

3.5.2.2 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。

3.5.2.2.1 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。

使用流式读取数据时，需要将 resultset 读取完成或close 后，才能继续使用该语句进行下次查询，否则会报错 No statements may be issued when any streaming result sets are open and in use on a <⟶ given connection. Ensure that you have called .close()on any active streaming result sets <⟶ before attempting more queries.。

如果需要在 resultset 读取完成或 close 前进行查询避免上述报错， 可在 URL 中添加配置参数 clobberStreamingResults=true，这样会自动 close resultset，但之前流式查询未被读取的结果集会丢失。

- 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。

3.5.2.2.2 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 > CPS 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 > CPS By Instance 查看请求命令类型，如果请求中 COM_STMT_EXECUTE 数目远远多于 COM_STMT_PREPARE 即生效。

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 > CPS 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 > CPS 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。

集成参数

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，可查看MySQL Connector/J 8.0 版本或 5.1 版本来确认当前MySQL Connector/J 中 maxPerformance 包含的具体配置。

配置后查看监控，可以看到多余语句减少。

超时参数

TiDB 提供两个与 MySQL 兼容的超时控制参数，wait_timeout 和 max_execution_time。这两个参数分别控制与 Java 应用连接的空闲超时时间和连接中 SQL 执行的超时时间，即控制TiDB 与 Java 应用的连接最长闲多久和最长忙多久。这两个参数的默认值都是 0，即默认允许连接无限闲置以及无限忙碌（一个 SQL 语句执行无限的长的时间）。

但在实际生产环境中，空闲连接和一直无限执行的 SQL 对数据库和应用都有不好的影响。你可以通过在应用的连接字符串中配置这两个参数来避免空闲连接和执行时间过长的 SQL 语句。例如，设置 sessionVariables <⟷ =wait_timeout=3600（1 小时）和 sessionVariables=max_execution_time=300000（5 分钟）。

3.5.2.3 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, druid, 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。

3.5.2.3.1 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。

3.5.2.3.2 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（由代理的idle 配置决定）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。

3.5.2.4 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。

3.5.2.4.1 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();
```

3.5.2.4.2 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。

3.5.2.5 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，方法返回结果前提交事务。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。

3.5.2.6 其他

3.5.2.6.1 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。

3.5.2.7 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。

3.5.3 HAProxy 在 TiDB 中的最佳实践

本文介绍 HAProxy 在 TiDB 中的最佳配置和使用方法。HAProxy 提供 TCP 协议下的负载均衡能力，TiDB 客户端通过连接 HAProxy 提供的浮动IP 即可对数据进行操作，实现TiDB Server 层的负载均衡。

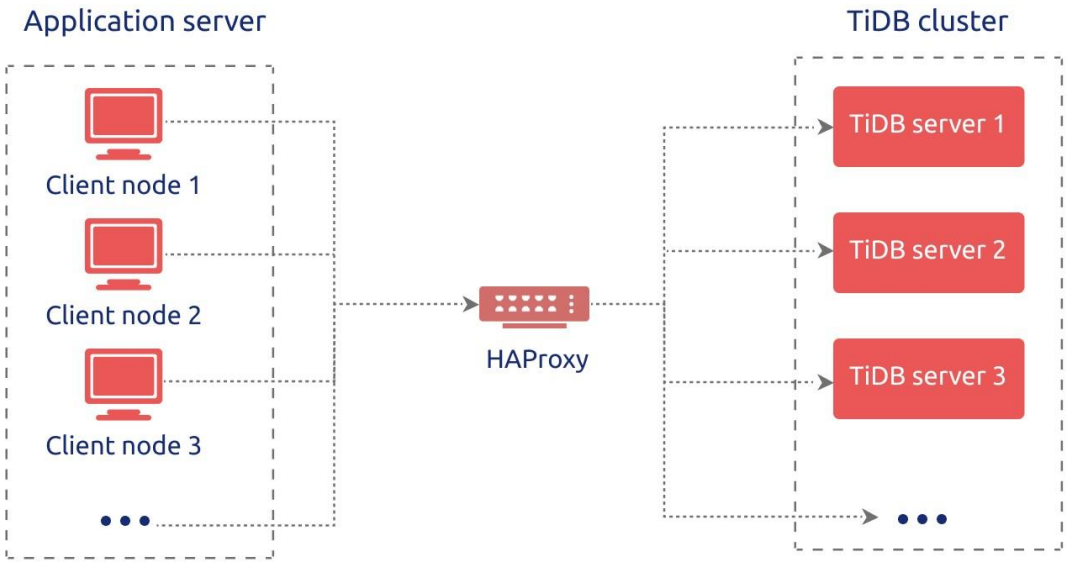


图 10: HAProxy 在 TiDB 中的最佳实践

注意：

TiDB 支持的最小 HAProxy 版本为 v1.5。使用 v1.5 到 v2.1 之间的 HAProxy 时，需要在 `mysql-check` 中配置 `post-41`。建议使用 HAProxy v2.2 或更高版本。

3.5.3.1 HAProxy 简介

HAProxy 是由C 语言编写的自由开放源码的软件，为基于TCP 和 HTTP 协议的应用程序提供高可用性、负载均衡和代理服务。因为 HAProxy 能够快速、高效使用 CPU 和内存，所以目前使用非常广泛，许多知名网站诸如 GitHub、 Bitbucket、 Stack Overflow、 Reddit、 Tumblr、 Twitter 和Tuenti 以及亚马逊网络服务系统都在使用 HAProxy。

HAProxy 由 Linux 内核的核心贡献者Willy Tarreau 于 2000 年编写，他现在仍然负责该项目的维护，并在开源社区免费提供版本迭代。本文示例使用 HAProxy 2.6。推荐使用最新稳定版的 HAProxy，详情见已发布的 HAProxy 版本。

3.5.3.2 HAProxy 部分核心功能介绍

- 高可用性：HAProxy 提供优雅关闭服务和无缝切换的高可用功能；
- 负载均衡：L4 (TCP) 和 L7 (HTTP) 两种负载均衡模式，至少 9 类均衡算法，比如 roundrobin，leastconn，random 等；
- 健康检查：对 HAProxy 配置的 HTTP 或者TCP 模式状态进行检查；
- 会话保持：在应用程序没有提供会话保持功能的情况下，HAProxy 可以提供该项功能；
- SSL：支持 HTTPS 通信和解析；
- 监控与统计：通过 web 页面可以实时监控服务状态以及具体的流量信息。

3.5.3.3 准备环境

在部署 HAProxy 之前，需准备好以下环境。

3.5.3.3.1 硬件要求

根据官方文档，对 HAProxy 的服务器硬件配置有以下建议，也可以根据负载均衡环境进行推算，在此基础上提高服务器配置。

硬件资源	最低配置
CPU	2 核，3.5 GHz
内存	16 GB
存储容量	50 GB（SATA 盘）
网卡	万兆网卡

3.5.3.3.2 依赖软件

根据 HAProxy 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

Linux 操作系统	版本
Red Hat Enterprise Linux	7 或者 8
CentOS	7 或者 8
Oracle Enterprise Linux	7 或者 8
Ubuntu LTS	18.04 或者以上版本

注意：

- 其他操作系统支持情况，详见 HAProxy 文档。

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
```

3.5.3.4 部署 HAProxy

HAProxy 配置 Database 负载均衡场景操作简单，以下部署操作具有普遍性，不具有特殊性，建议根据实际场景，个性化配置相关的配置文件。

3.5.3.4.1 安装 HAProxy

1. 下载 HAProxy 2.6.2 的源码包：

   ```
   wget  https://www.haproxy.org/download/2.6/src/haproxy-2.6.2.tar.gz
   ```

2. 解压源码包：

   ```
   tar zxf haproxy-2.6.2.tar.gz
   ```

3. 从源码编译 HAProxy 应用：

   ```
   cd haproxy-2.6.2
   make clean
   make -j 8 TARGET=linux-glibc USE_THREAD=1
   make PREFIX=${/app/haproxy} SBINDIR=${/app/haproxy/bin} install  # 将 `${/app/haproxy}` 和 `
       <-> ${/app/haproxy/bin}` 替换为自定义的实际路径。
   ```

4. 重新配置 `profile` 文件：

   ```
   echo 'export PATH=/app/haproxy/bin:$PATH' >> /etc/profile
   . /etc/profile
   ```

5. 检查 HAProxy 是否安装成功：

```
which 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。
-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）。

参数	说明
`-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。
`-dV`	禁止在服务器端使用 SSL。

参数	说明
`-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。

3.5.3.4.2 配置 HAProxy

yum 安装过程中会生成配置模版，你也可以根据实际场景自定义配置如下配置项。

```
global                                  # 全局配置。
   log         127.0.0.1 local2         # 定义全局的 syslog 服务器，最多可以定义两个。
   chroot      /var/lib/haproxy         # 更改当前目录并为启动进程设置超级用户权限，
       <-> 从而提高安全性。
   pidfile     /var/run/haproxy.pid     # 将 HAProxy 进程的 PID 写入 pidfile。
   maxconn     4096                     # 单个 HAProxy 进程可接受的最大并发连接数，
       <-> 等价于命令行参数 "-n"。
   nbthread    48                       # 最大线程数。线程数的上限与 CPU 数量相同。
   user        haproxy                  # 同 UID 参数。
   group       haproxy                  # 同 GID 参数，建议使用专用用户组。
   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
```

如要通过 SHOW PROCESSLIST 查看连接来源IP，需要配置使用 PROXY 协议连接TiDB。

```
   server tidb-1 10.9.18.229:4000 send-proxy check inter 2000 rise 2 fall 3
   server tidb-2 10.9.39.208:4000 send-proxy check inter 2000 rise 2 fall 3
   server tidb-3 10.9.64.166:4000 send-proxy check inter 2000 rise 2 fall 3
```

> **注意：**
>
> 使用 PROXY 协议时，你需要在tidb-server 的配置文件中设置proxy-protocol.networks。

3.5.3.4.3 启动 HAProxy

要启动 HAProxy，执行 haproxy 命令。默认读取 /etc/haproxy/haproxy.cfg（推荐）。

```
haproxy -f /etc/haproxy/haproxy.cfg
```

3.5.3.4.4 停止 HAProxy

要停止 HAProxy，使用 kill -9 命令。

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}
```

3.5.4 TiDB 高并发写入场景最佳实践

在TiDB的使用过程中，一个典型场景是高并发批量写入数据到TiDB。本文阐述了该场景中的常见问题，旨在给出一个业务的最佳实践，帮助读者避免因使用TiDB不当而影响业务开发。

3.5.4.1 目标读者

本文假设你已对TiDB有一定的了解，推荐先阅读TiDB原理相关的三篇文章（讲存储，说计算，谈调度），以及 TiDB Best Practice。

3.5.4.2 高并发批量插入场景

高并发批量插入的场景通常出现在业务系统的批量任务中，例如清算以及结算等业务。此类场景存在以下特点：

- 数据量大
- 需要短时间内将历史数据入库
- 需要短时间内读取大量数据

这就对TiDB提出了以下挑战：

- 写入/读取能力是否可以线性水平扩展
- 随着数据持续大并发写入，数据库性能是否稳定不衰减

对于分布式数据库来说，除了本身的基础性能外，最重要的就是充分利用所有节点能力，避免让单个节点成为瓶颈。

3.5.4.3 TiDB 数据分布原理

如果要解决以上挑战，需要从TiDB 数据切分以及调度的原理开始讲起。这里只作简单说明，详情可参阅谈调度。

TiDB 以Region 为单位对数据进行切分，每个 Region 有大小限制（默认 96M）。Region 的切分方式是范围切分。每个 Region 会有多副本，每一组副本，称为一个Raft Group。每个Raft Group 中由 Leader 负责执行这块数据的读&写（TiDB 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。

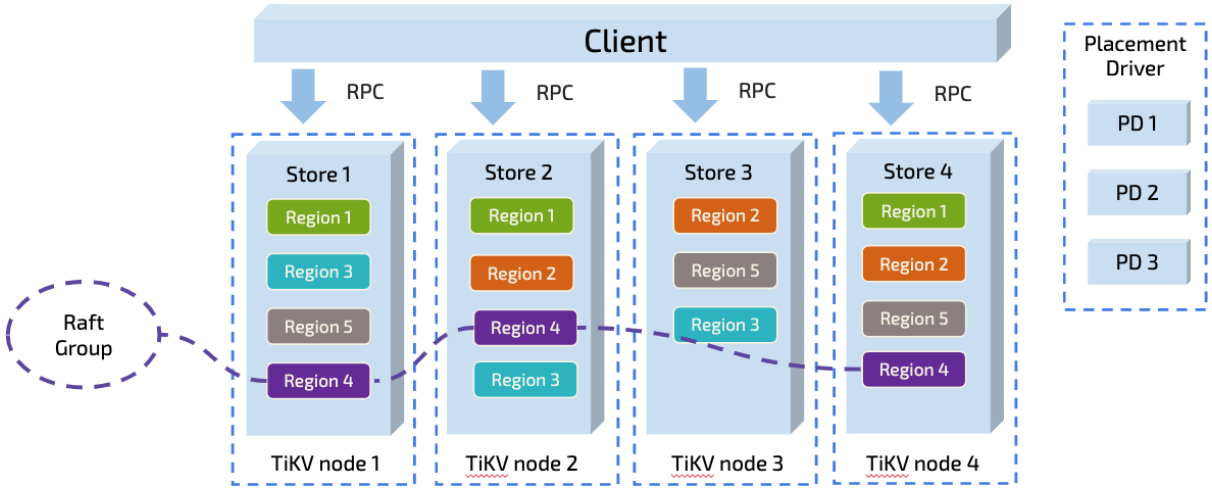


图 11: TiDB 数据概览

从原理上来说，只要没有业务上的写入热点（即业务写入没有 AUTO_INCREMENT 的主键和单调递增的索引，更多细节可参阅TiDB 正确使用方式），依靠这个架构，TiDB 不仅具备线性扩展的读写能力，也能够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。

3.5.4.4 热点产生的实例

以下为一张示例表：

```
CREATE TABLE IF NOT EXISTS
      TEST_HOTSPOT( id        BIGINT
      PRIMARY KEY,
      age         INT,
      user_name  VARCHAR(32),
      email       VARCHAR(128)
)
```

这个表的结构非常简单，除了 id 为主键以外，没有额外的二级索引。将数据写入该表的语句如下，id 通过随机数离散生成：

```
SET SESSION cte_max_recursion_depth = 1000000;
INSERT INTO TEST_HOTSPOT
SELECT
  n,                                        -- ID
  RAND()*80,                                -- 0 到 80 之间的随机数
  CONCAT('user-',n),
  CONCAT(
    CHAR(65 + (RAND() * 25) USING ascii),  -- 65 到 65+25 之间的随机数，转换为一个 A-Z 字符
    '-user-',
```

```
    n,
    '@example.com'
  )
FROM
  (WITH RECURSIVE nr(n) AS
    (SELECT 1                                    -- 从 1 开始 CTE
      UNION ALL SELECT n + 1                     -- 每次循环 n 增加 1
      FROM nr WHERE n < 1000000                  -- 当 n 为 1_000_000 时停止循环
    ) SELECT n FROM nr
  ) a;
```

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。

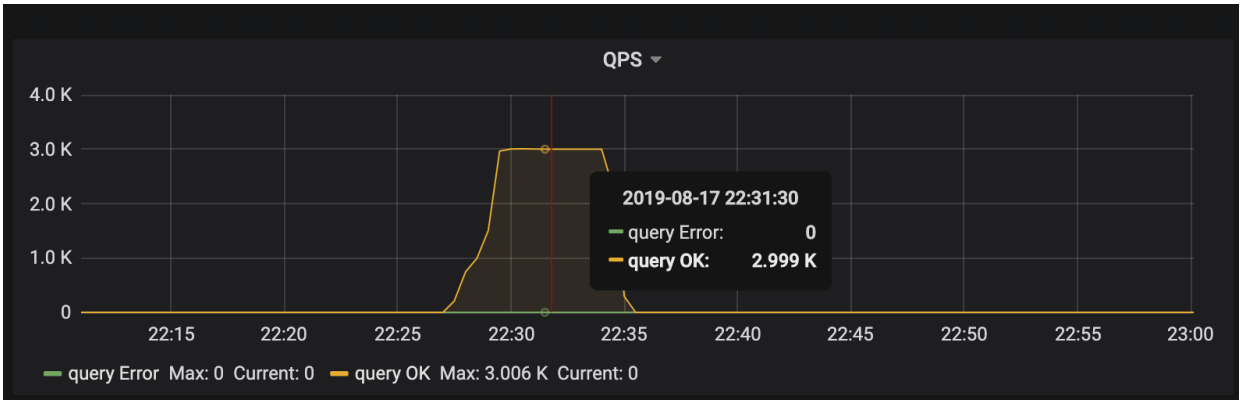


图 12: QPS1

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 ）：

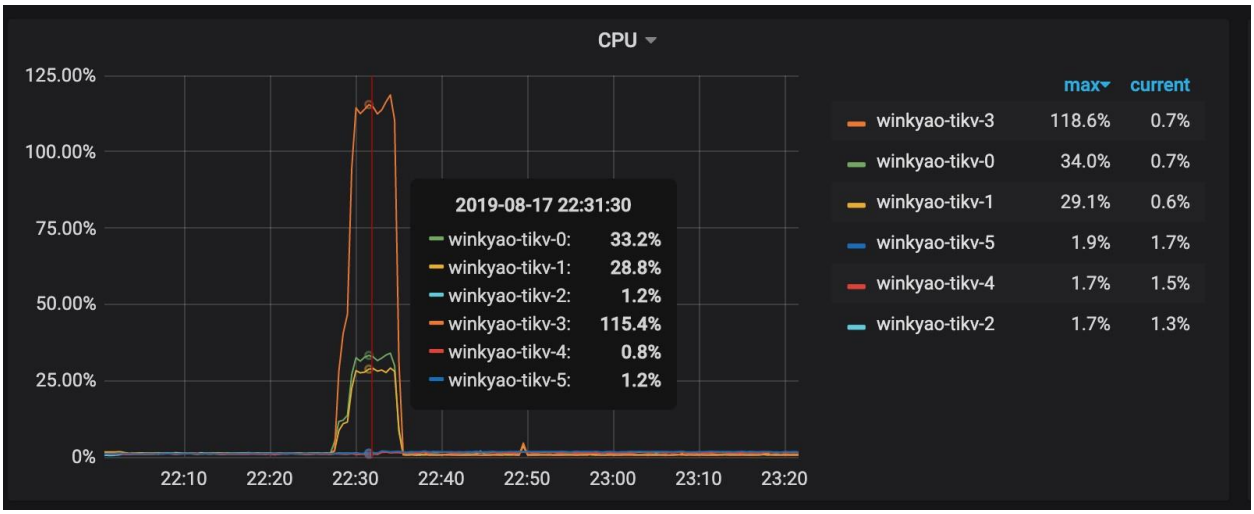


图 13: QPS2

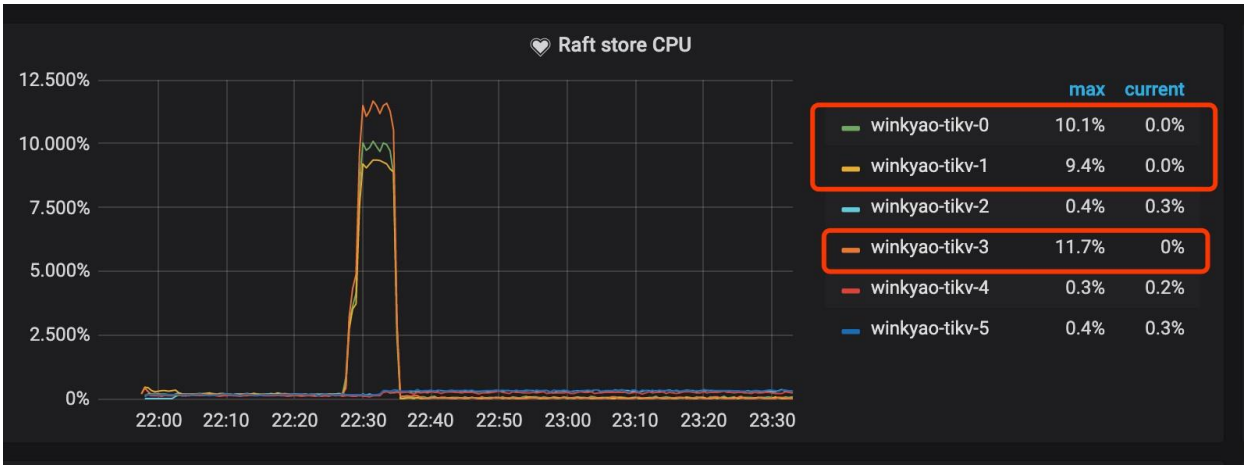


图 14: QPS3

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 15: QPS4

3.5.4.5 热点问题产生的原因

以上测试并未达到理论场景中最佳实践，因为刚创建表的时候，这个表在TiKV 中只会对应为一个Region，范围是：

```
[CommonPrefix + TableID, CommonPrefix + TableID + 1)
```

短时间内大量数据会持续写入到同一个Region 上。

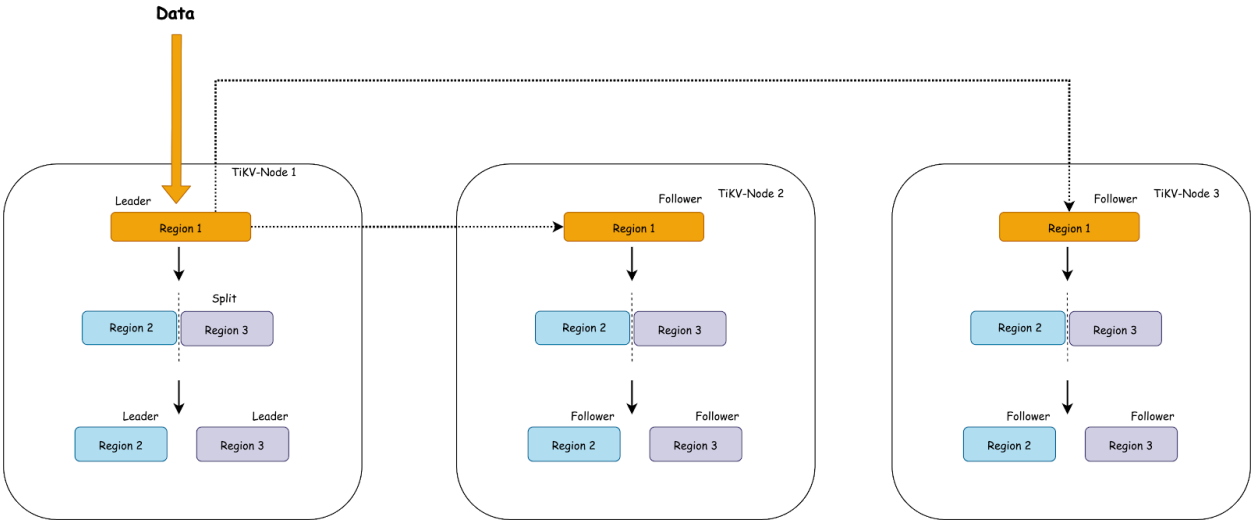


图 16: TiKV Region 分裂流程

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：

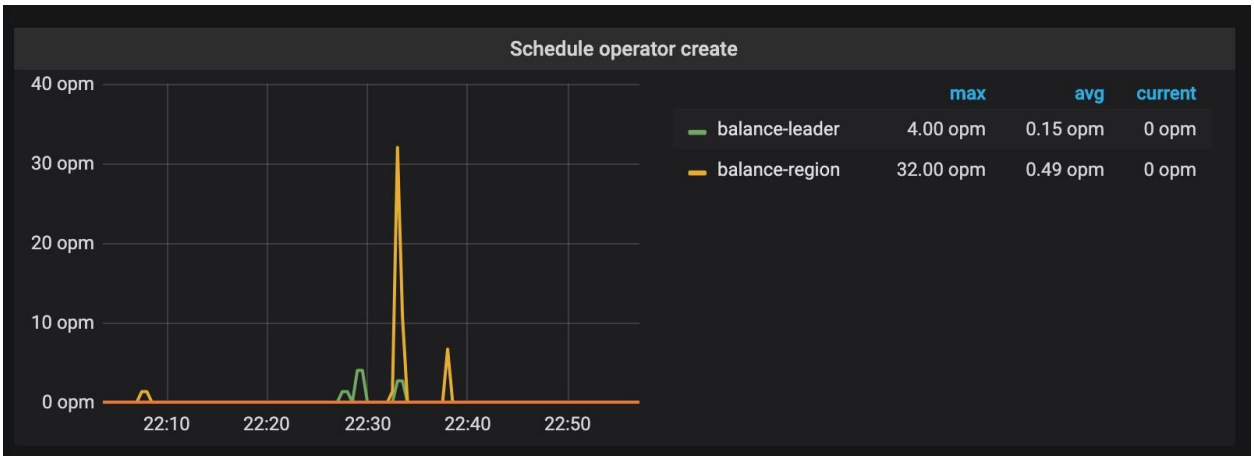


图 17: QPS5

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。

3.5.4.6 热点问题的规避方法

为了达到场景理论中的最佳性能，可跳过这个预热阶段，直接将Region 切分为预期的数量，提前调度到集群的各个节点中。

TiDB 在 v3.0.x 以及 v2.1.13 后支持一个叫Split Region 的新特性。这个特性提供了新的语法：

```
SPLIT TABLE table_name [INDEX index_name] BETWEEN (lower_value) AND (upper_value) REGIONS
    <↪ region_num
```

```
SPLIT TABLE table_name [INDEX index_name] BY (value_list) [, (value_list)]
```

但是TiDB 并不会自动提前完成这个切分操作。原因如下：

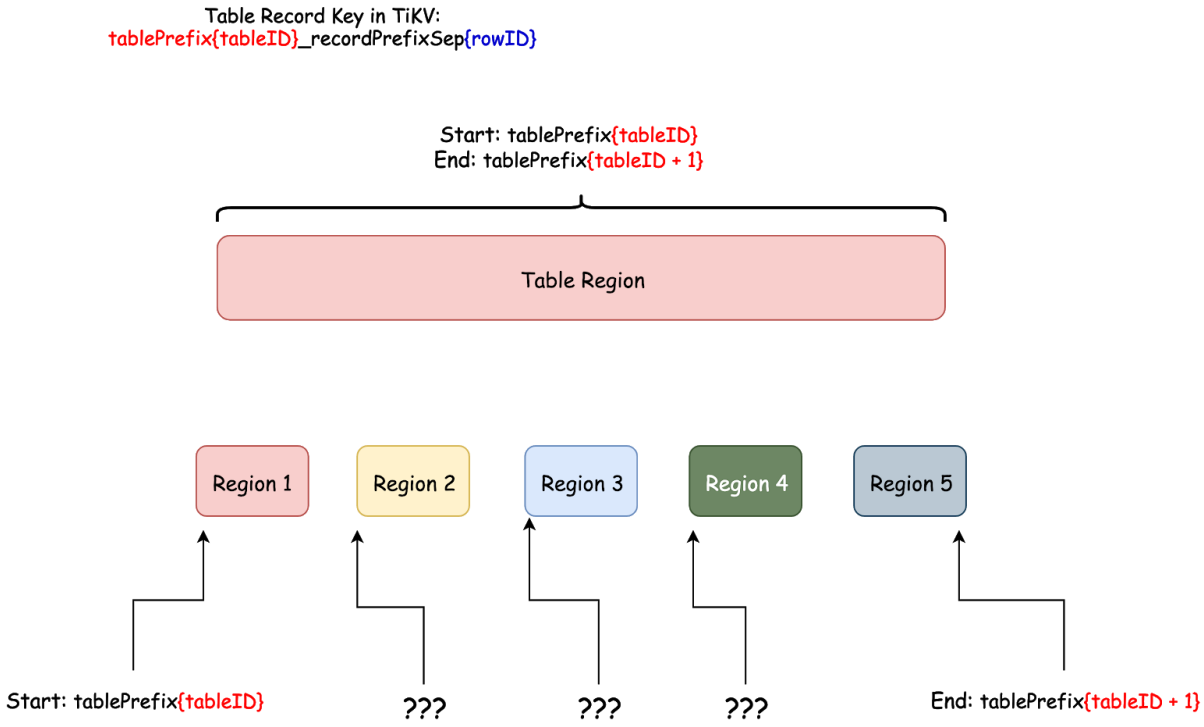


图 18: Table Region Range

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。

也可以通过以下 SQL 语句查看Region 的分布。你需要将 table_name 替换为实际的表名。

```
SELECT
    p.STORE_ID,
    COUNT(s.REGION_ID) PEER_COUNT
FROM
    INFORMATION_SCHEMA.TIKV_REGION_STATUS  s
    JOIN INFORMATION_SCHEMA.TIKV_REGION_PEERS p ON s.REGION_ID = p.REGION_ID
WHERE
    TABLE_NAME = 'table_name'
    AND p.is_leader = 1
GROUP BY
    p.STORE_ID
ORDER BY
    PEER_COUNT DESC;
```

再重新运行写入负载：

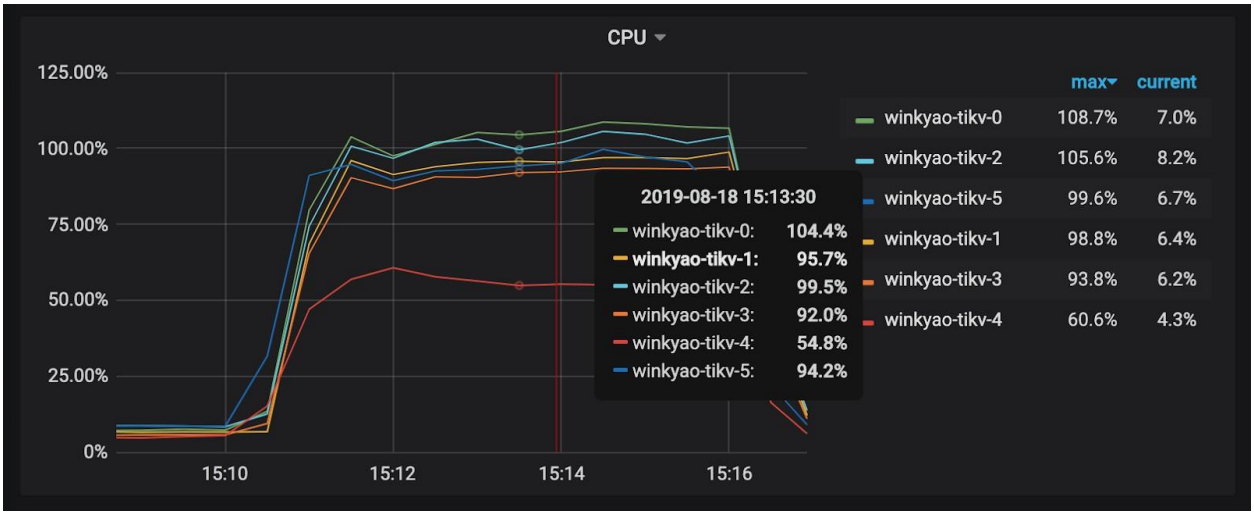


图 19: QPS6

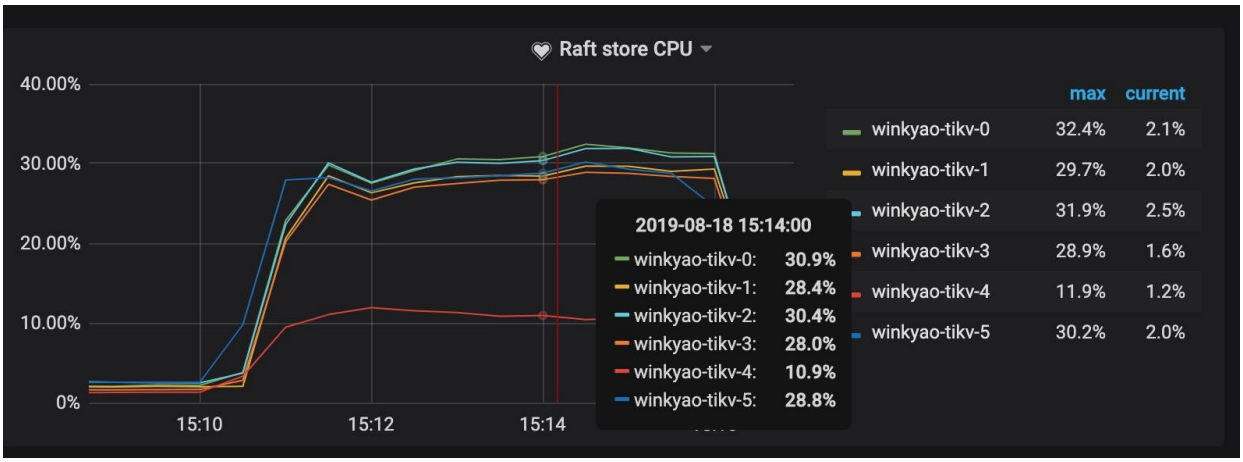


图 20: QPS7

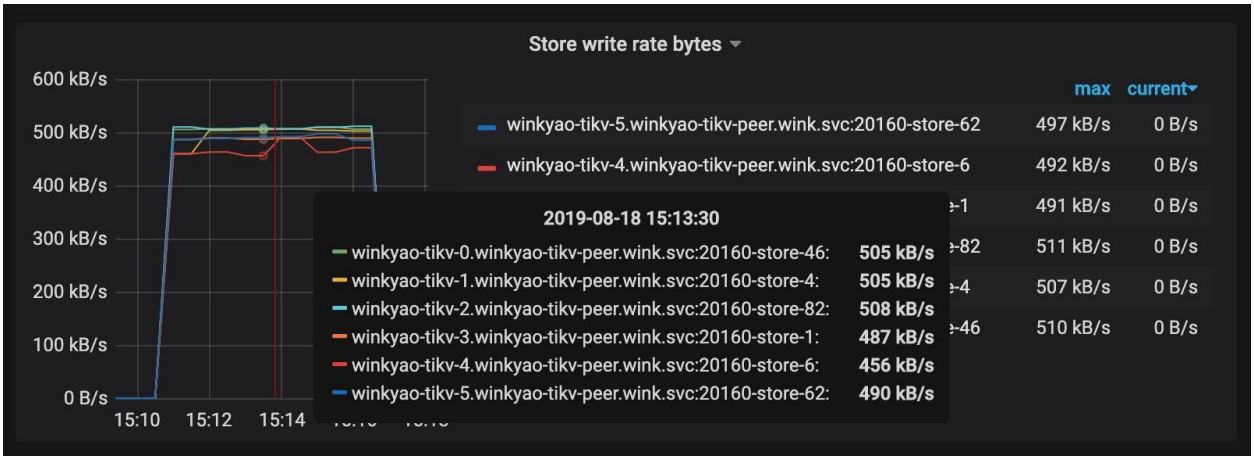


图 21: QPS8

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。

3.5.4.6.1 更复杂的热点问题

问题一：

如果表没有主键或者主键不是整数类型，而且用户也不想自己生成一个随机分布的主键ID 的话，TiDB 内部有一个隐式的 `_tidb_rowid` 列作为行ID。在不使用 `SHARD_ROW_ID_BITS` 的情况下，`_tidb_rowid` 列的值基本也为单调递增，此时也会有写热点存在（参阅SHARD_ROW_ID_BITS 的详细说明）。

要避免由 `_tidb_rowid` 带来的写入热点问题，可以在建表时，使用 `SHARD_ROW_ID_BITS` 和 `PRE_SPLIT_REGIONS` 这两个建表选项（参阅PRE_SPLIT_REGIONS 的详细说明）。

SHARD_ROW_ID_BITS 用于将 _tidb_rowid 列生成的行ID 随机打散。PRE_SPLIT_REGIONS 用于在建完表后预先进行 Split region。

> **注意：**
>
> PRE_SPLIT_REGIONS 的值必须小于或等于 SHARD_ROW_ID_BITS。

以下全局变量会影响 PRE_SPLIT_REGIONS 的行为，需要特别注意：

- tidb_scatter_region：该变量用于控制建表完成后是否等待预切分和打散 Region 完成后再返回结果。如果建表后有大批量写入，需要设置该变量值为 1，表示等待所有Region 都切分和打散完成后再返回结果给客户端。否则未打散完成就进行写入会对写入性能影响有较大的影响。

示例：

```
create table t (a int, b int) SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS=3;
```

- SHARD_ROW_ID_BITS = 4 表示tidb_rowid 的值会随机分布成 16 (16=2^4) 个范围区间。
- PRE_SPLIT_REGIONS=3 表示建完表后提前切分出 8 (2^3) 个 Region。

开始写数据进表t 后，数据会被写入提前切分好的 8 个 Region 中，这样也避免了刚开始建表完后因为只有一个 Region 而存在的写热点问题。

问题二：

如果表的主键为整数类型，并且该表使用了 AUTO_INCREMENT 来保证主键唯一性（不需要连续或递增）的表而言，由于TiDB 直接使用主键行值作为 _tidb_rowid，此时无法使用 SHARD_ROW_ID_BITS 来打散热点。

要解决上述热点问题，可以利用 AUTO_RANDOM 列属性（参阅AUTO_RANDOM 的详细说明），将 AUTO_INCREMENT 改为 AUTO_RANDOM，插入数据时让 TiDB 自动为整型主键列分配一个值，消除行ID 的连续性，从而达到打散热点的目的。

3.5.4.7 参数配置

TiDB 2.1 版本中在 SQL 层引入了latch 机制，用于在写入冲突比较频繁的场景中提前发现事务冲突，减少TiDB 和TiKV 事务提交时写写冲突导致的重试。通常，跑批场景使用的是存量数据，所以并不存在事务的写入冲突。可以把TiDB 的 latch 功能关闭，以减少为细小对象分配内存：

```
[txn-local-latches]
enabled = false
```

3.5.5 使用Grafana 监控TiDB 的最佳实践

使用TiUP 部署TiDB 集群时，如果在拓扑配置中添加了Grafana 和 Prometheus，会部署一套 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。

3.5.5.1 监控架构

Prometheus 是一个拥有多维度数据模型和灵活查询语句的时序数据库。Grafana 是一个开源的metric 分析及可视化系统。

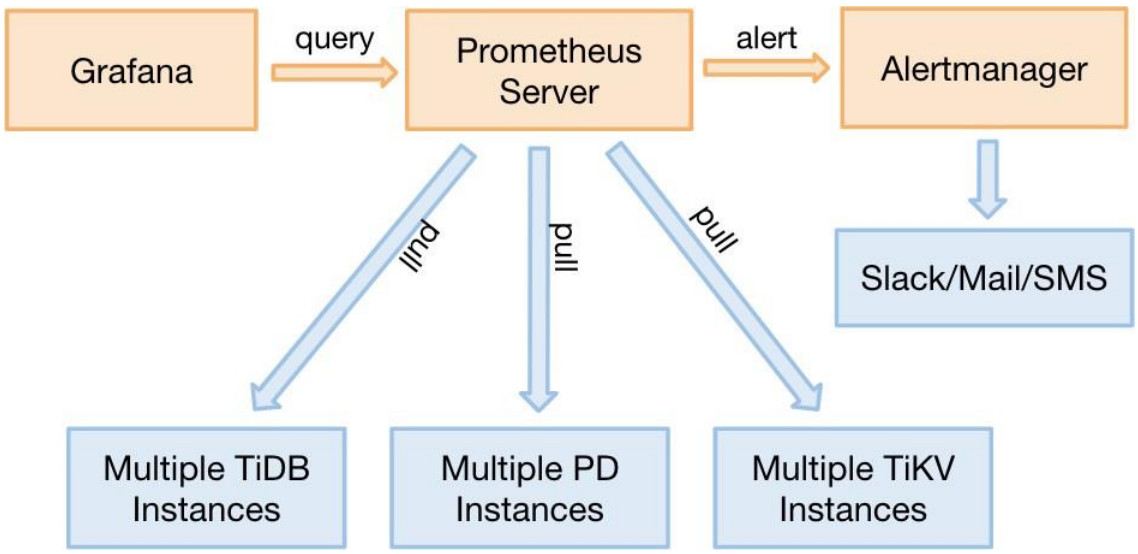


图 22: TiDB 监控整体架构

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。
- 去掉了 Pushgateway 这个单点组件。

3.5.5.2 监控数据的来源与展示

TiDB 的 3 个核心组件（TiDB server、TiKV server 和 PD server）可以通过 HTTP 接口来获取metric 数据。这些 metric 均是从程序代码中上传的，默认端口如下：

组件	端口
TiDB server	10080
TiKV server	20181
PD server	2379

下面以TiDB server 为例，展示如何通过 HTTP 接口查看一个语句的 QPS 数据：

```
curl http://__tidb_ip__:10080/metrics |grep tidb_executor_statement_total
```

```
### 可以看到实时 QPS 数据，并根据不同 type 对 SQL 语句进行了区分，value 是 counter 类型的累计值（
    <-> 科学计数法）。
tidb_executor_statement_total{type="Delete"} 520197
tidb_executor_statement_total{type="Explain"} 1
tidb_executor_statement_total{type="Insert"} 7.20799402e+08
tidb_executor_statement_total{type="Select"} 2.64983586e+08
tidb_executor_statement_total{type="Set"} 2.399075e+06
tidb_executor_statement_total{type="Show"} 500531
tidb_executor_statement_total{type="Use"} 466016
```

这些数据会存储在 Prometheus 中，然后在Grafana 上进行展示。在面板上点击鼠标右键会出现Edit 按钮（或直接按E 键），如下图所示：

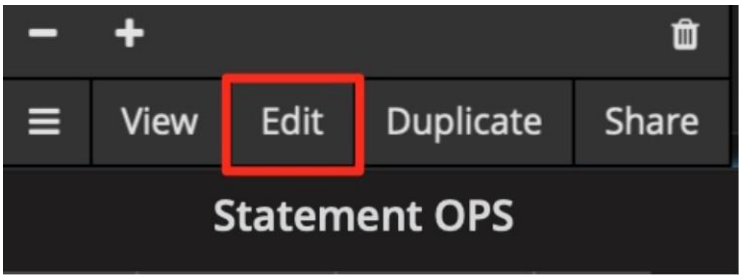


图 23: Metrics 面板的编辑入口

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 24: 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。

3.5.5.3 Grafana 使用技巧

本小节介绍高效利用Grafana 监控分析TiDB 指标的七个技巧。

3.5.5.3.1 技巧 1：查看所有维度并编辑表达式

在监控数据的来源与展示一节的示例中，数据是按照type 进行分组的。如果你想知道是否还能按其它维度分组，并快速查看还有哪些维度，可采用以下技巧：在query 的表达式上只保留指标名称，不做任何计算，`Legend format` 也留空。这样就能显示出原始的 metric 数据。比如，下图能看到有 3 个维度（`instance`、`job` 和 `type`）：

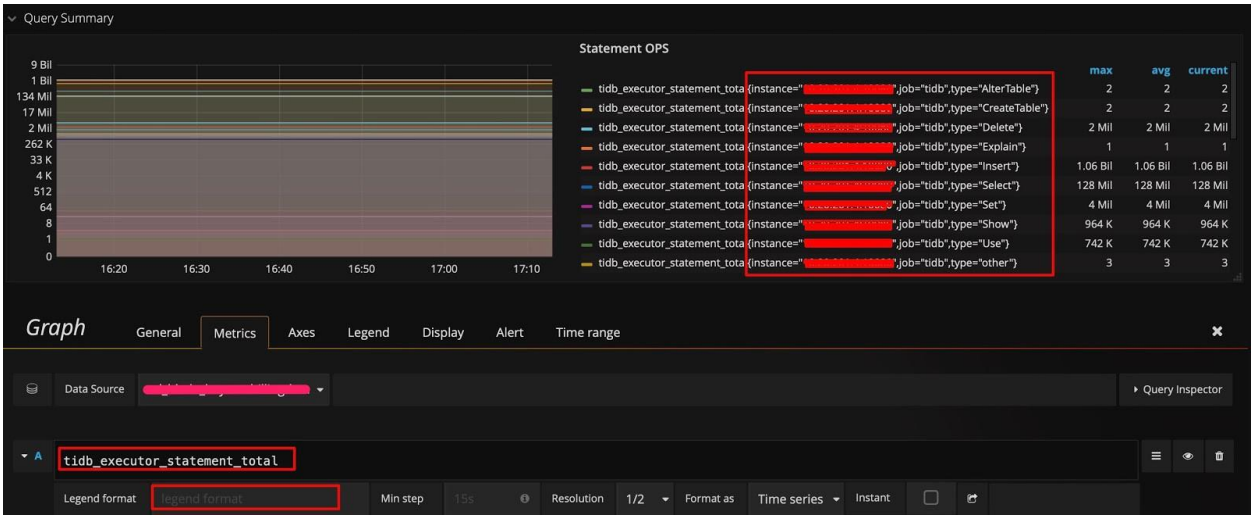


图 25: 编辑表达式并查看所有维度

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 26: 给表达式增加一个instance 维度

3.5.5.3.2 技巧 2：调整Y 轴标尺的计算方式

以 Query Duration 指标为例，默认的比例尺采用 2 的对数计算，显示上会将差距缩小。为了观察到明显的变化，可以将比例尺改为线性，从下面两张图中可以看到显示上的区别，明显发现那个时刻有个 SQL 语句运行较慢。

当然也不是所有场景都适合用线性，比如观察 1 个月的性能趋势，用线性可能就会有很多噪点，不好观察。

标尺默认的比例尺为 2 的对数：

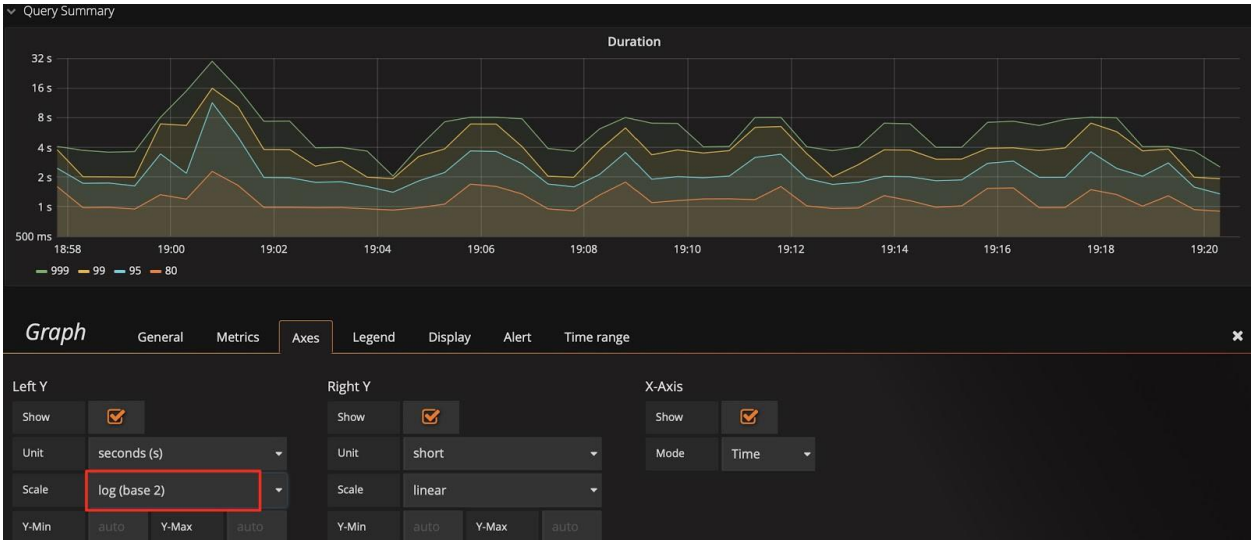


图 27: 标尺默认的比例尺为 2 的对数

将标尺的比例尺调整为线性：

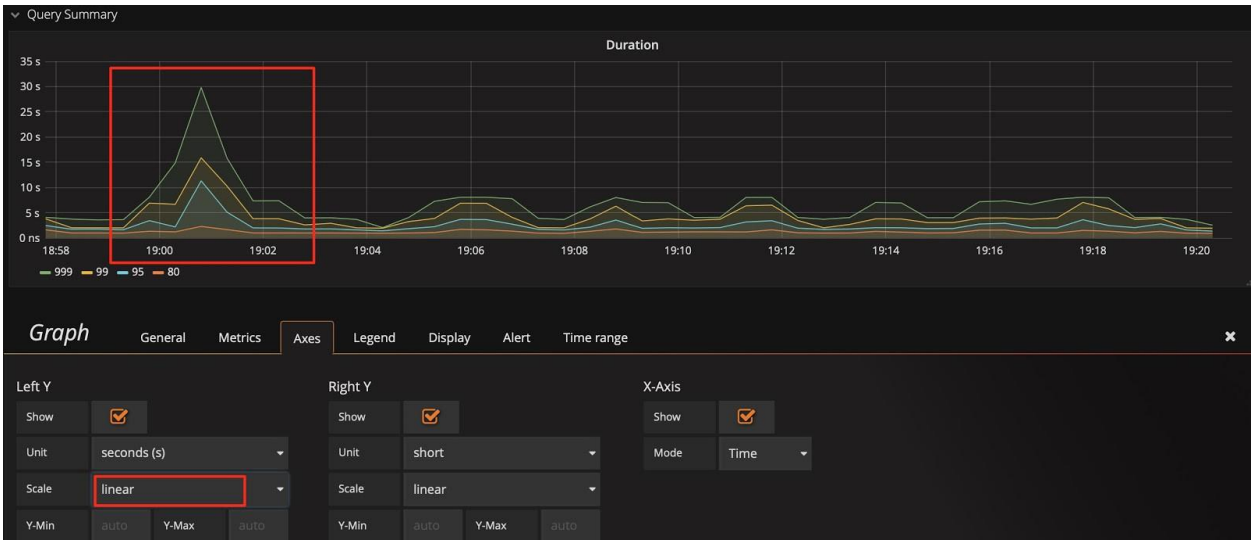


图 28: 调整标尺的比例尺为线性

> **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。

3.5.5.3.3 技巧 3：调整Y 轴基线，放大变化

有时已经用了线性比例尺，却还是看不出变化趋势。比如下图中，在扩容后想观察 `Store size` 的实时变化效果，但由于基数较大，观察不到微弱的变化。这时可以将Y 轴最小值从 `0` 改为 `auto`，将上部放大。观察下面两张图的区别，可以看出数据已开始迁移了。

基线默认为 `0`：

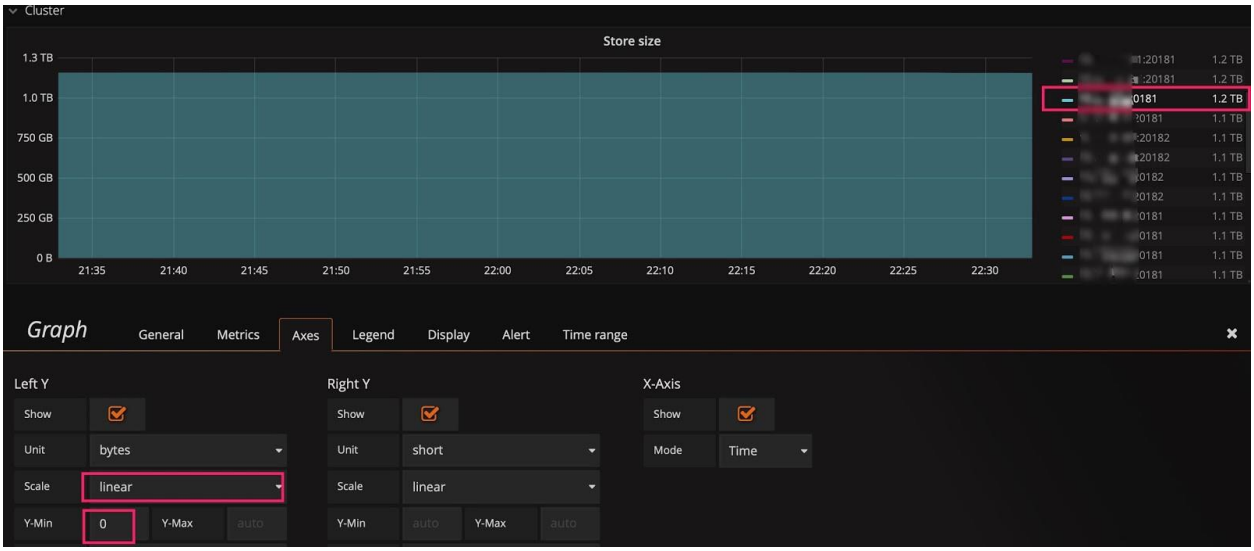


图 29: 基线默认为 0

将基线调整为 auto：

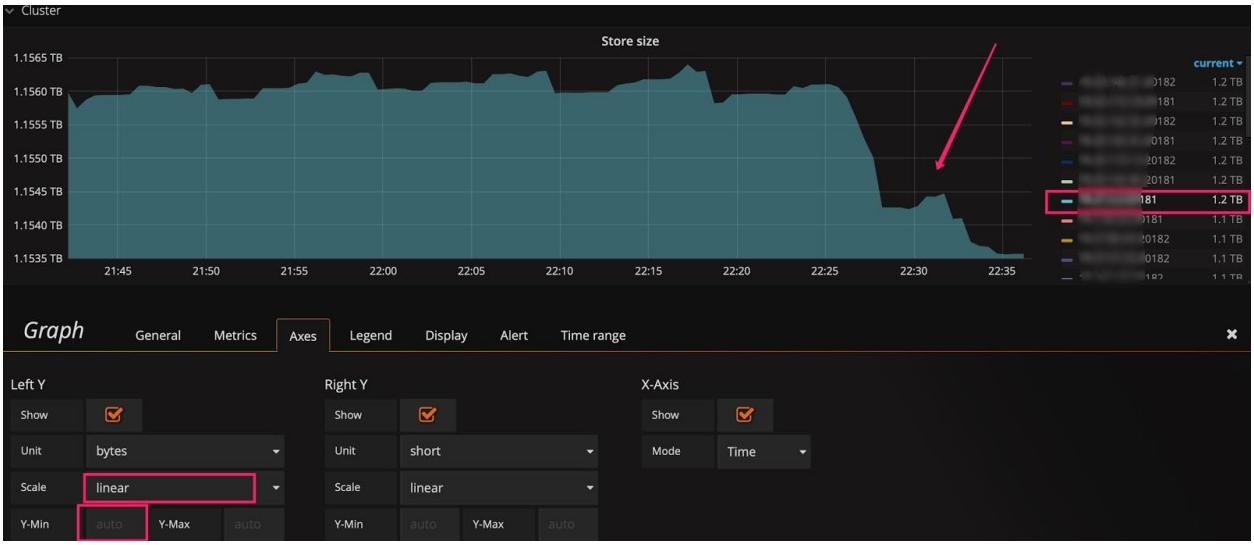


图 30: 调整基线为auto

3.5.5.3.4 技巧 4：标尺联动

在 Settings 面板中，有一个Graph Tooltip 设置项，默认使用 Default。

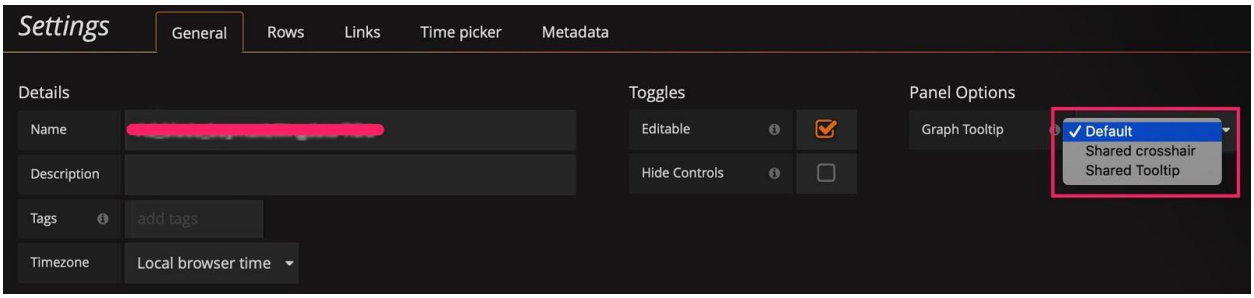


图 31: 图形展示工具

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：

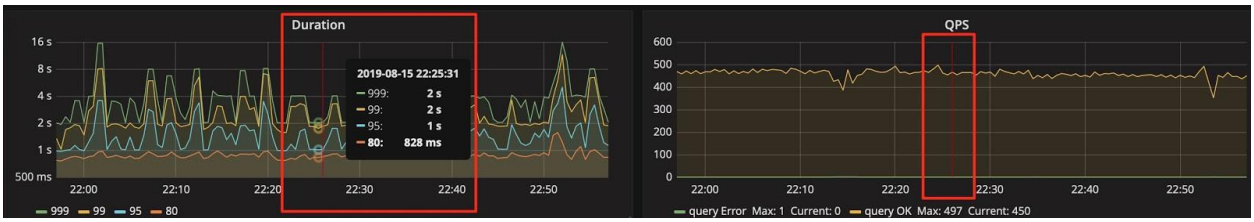


图 32: 调整图形展示工具为 Shared crosshair

将图形展示工具调整为 Shared Tooltip：

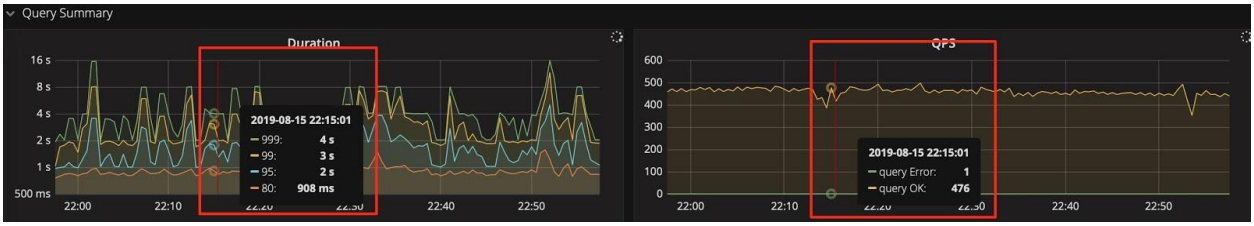


图 33: 调整图形展示工具为 Shared Tooltip

3.5.5.3.5 技巧 5：手动输入 ip: 端口号查看历史信息

PD 的 dashboard 只展示当前leader 的 metric 信息，而有时想看历史上 PD leader 当时的状况，但是 instance 下拉列表中已不存在这个成员了。此时，可以手动输入 ip:2379 来查看当时的数据。



图 34: 查看历史metric 信息

3.5.5.3.6 技巧 6：巧用 Avg 函数

通常默认图例中只有 Max 和 Current 函数。当指标波动较大时，可以增加 Avg 等其它汇总函数的图例，来看一段时间的整体趋势。

增加 Avg 等汇总函数：

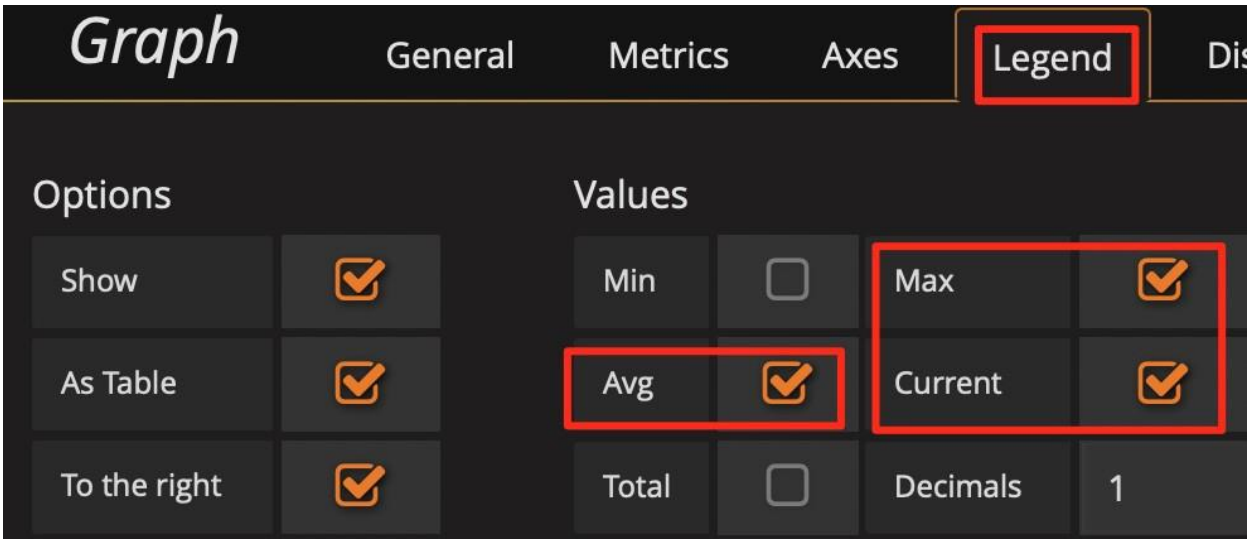


图 35: 增加Avg 等汇总函数

然后查看整体趋势：

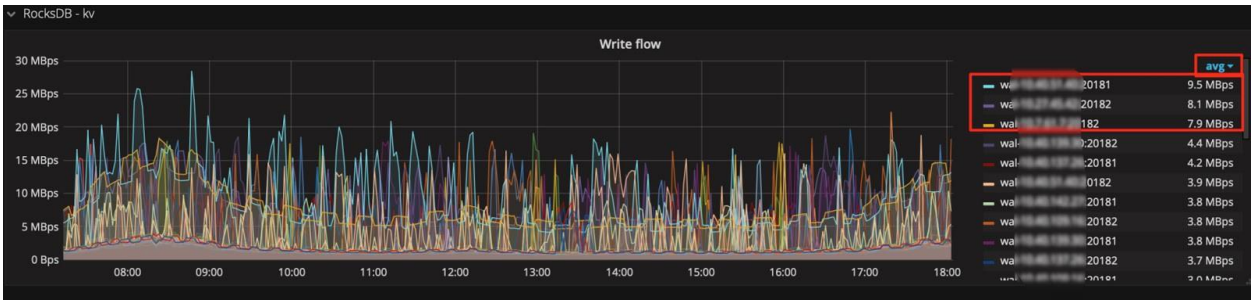


图 36: 增加 Avg 函数查看整体趋势

3.5.5.3.7 技巧 7：使用 Prometheus 的 API 接口获得表达式的结果

Grafana 通过 Prometheus 的接口获取数据，你也可以用该接口来获取数据，这个用法还可以衍生出许多功能：

- 自动获取集群规模、状态等信息。
- 对表达式稍加改动给报表提供数据，如统计每天的 QPS 总量、每天的 QPS 峰值和每天的响应时间。
- 将重要的指标进行定期健康巡检。

Prometheus 的 API 接口如下：



图 37: 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"
}
```

3.5.5.4 总结

Grafana + Prometheus 监控平台是一套非常强大的组合工具，用好这套工具可以为分析节省很多时间，提高效率，更重要的是，我们可以更容易发现问题。在运维TiDB 集群，尤其是数据量大的情况下，这套工具能派上大用场。

3.5.6 PD 调度策略最佳实践

本文将详细介绍 PD 调度系统的原理，并通过几个典型场景的分析和处理方式，分享调度策略的最佳实践和调优方式，帮助大家在使用过程中快速定位问题。本文假定你对TiDB，TiKV 以及 PD 已经有一定的了解，相关核心概念如下：

- Leader/Follower/Learner
- Operator
- Operator Step
- Pending/Down
- Region/Peer/Raft Group

- Region Split
- Scheduler
- Store

> **注意：**
>
> 本文内容基于TiDB 3.0 版本，更早的版本(2.x)缺少部分功能的支持，但是基本原理类似，也可以以本文作为参考。

3.5.6.1 PD 调度原理

该部分介绍调度系统涉及到的相关原理和流程。

3.5.6.1.1 调度流程

宏观上来看，调度流程大体可划分为 3 个部分：

1. 信息收集

   TiKV 节点周期性地向 PD 上报 `StoreHeartbeat` 和 `RegionHeartbeat` 两种心跳消息：

   - `StoreHeartbeat` 包含 Store 的基本信息、容量、剩余空间和读写流量等数据。
   - `RegionHeartbeat` 包含Region 的范围、副本分布、副本状态、数据量和读写流量等数据。

   PD 梳理并转存这些信息供调度进行决策。

2. 生成调度

   不同的调度器从自身的逻辑和需求出发，考虑各种限制和约束后生成待执行的 Operator。这里所说的限制和约束包括但不限于：

   - 不往处于异常状态中（如断连、下线、繁忙、空间不足或在大量收发 snapshot 等）的 Store 添加副本
   - Balance 时不选择状态异常的Region
   - 不尝试把 Leader 转移给 Pending Peer
   - 不尝试直接移除 Leader
   - 不破坏Region 各种副本的物理隔离
   - 不破坏 Label property 等约束

3. 执行调度

   调度执行的步骤为：

   a. Operator 先进入一个由 `OperatorController` 管理的等待队列。

   b. `OperatorController` 会根据配置以一定的并发量从等待队列中取出 Operator 并执行。执行的过程就是依次把每个 Operator Step 下发给对应Region 的 Leader。

   c. 标记 Operator 为 “finish” 或 “timeout” 状态，然后从执行列表中移除。

3.5.6.1.2 负载均衡

Region 负载均衡调度主要依赖 balance-leader 和 balance-region 两个调度器。二者的调度目标是将Region 均匀地分散在集群中的所有 Store 上，但它们各有侧重：balance-leader 关注Region 的 Leader，目的是分散处理客户端请求的压力；balance-region 关注Region 的各个 Peer，目的是分散存储的压力，同时避免出现爆盘等状况。

balance-leader 与 balance-region 有着相似的调度流程：

1. 根据不同 Store 的对应资源量的情况分别打分。
2. 不断从得分较高的 Store 选择 Leader 或 Peer 迁移到得分较低的 Store 上。

两者的分数计算上也有一定差异：balance-leader 比较简单，使用 Store 上所有 Leader 所对应的Region Size 加和作为得分。因为不同节点存储容量可能不一致，计算 balance-region 得分会分以下三种情况：

- 当空间富余时使用数据量计算得分（使不同节点数据量基本上均衡）
- 当空间不足时由使用剩余空间计算得分（使不同节点剩余空间基本均衡）
- 处于中间态时则同时考虑两个因素做加权和当作得分

此外，为了应对不同节点可能在性能等方面存在差异的问题，还可为 Store 设置负载均衡的权重。leader-<⟶weight 和 region-weight 分别用于控制 Leader 权重以及Region 权重（默认值都为“1”）。假如把某个 Store 的 leader-weight 设为“2”，调度稳定后，则该节点的 Leader 数量约为普通节点的 2 倍；假如把某个 Store 的 region-weight 设为“0.5”，那么调度稳定后该节点的Region 数量约为其他节点的一半。

3.5.6.1.3 热点调度

热点调度对应的调度器是 hot-region-scheduler。从 TiDB v3.0 版本开始，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。

3.5.6.1.4 集群拓扑感知

让 PD 感知不同节点分布的拓扑是为了通过调度使不同Region 的各个副本尽可能分散，保证高可用和容灾。PD 会在后台不断扫描所有Region，当发现Region 的分布不是当前的最优化状态时，会生成调度以替换 Peer，将 Region 调整至最佳状态。

负责这个检查的组件叫 replicaChecker（跟 Scheduler 类似，但是不可关闭）。它依赖于 location-labels 配置项来进行调度。比如配置 [zone,rack,host] 定义了三层的拓扑结构：集群分为多个 zone（可用区），每个 zone 下有多个rack（机架），每个 rack 下有多个host（主机）。PD 在调度时首先会尝试将Region 的 Peer 放置在不同的 zone，假如无法满足（比如配置 3 副本但总共只有 2 个 zone）则保证放置在不同的rack；假如rack 的数量也不足以保证隔离，那么再尝试host 级别的隔离，以此类推。

3.5.6.1.5 缩容及故障恢复

缩容是指预备将某个 Store 下线，通过命令将该 Store 标记为“Offline”状态，此时 PD 通过调度将待下线节点上的Region 迁移至其他节点。

故障恢复是指当有 Store 发生故障且无法恢复时，有 Peer 分布在对应 Store 上的Region 会产生缺少副本的状况，此时 PD 需要在其他节点上为这些Region 补副本。

这两种情况的处理过程基本上是一样的。`replicaChecker` 检查到Region 存在异常状态的 Peer 后，生成调度在健康的 Store 上创建新副本替换异常的副本。

3.5.6.1.6 Region merge

Region merge 指的是为了避免删除数据后大量小甚至空的Region 消耗系统资源，通过调度把相邻的小Region 合并的过程。Region merge 由 `mergeChecker` 负责，其过程与 `replicaChecker` 类似：PD 在后台遍历，发现连续的小 Region 后发起调度。

具体来说，当某个新分裂出来的Region 存在的时间超过配置项`split-merge-interval` 的值（默认 1h）后，如果同时满足以下情况，该Region 会触发 Region merge 调度：

- 该Region 大小小于配置项`max-merge-region-size` 的值（默认 20 MiB）
- 该Region 中 key 的数量小于配置项`max-merge-region-keys` 的值（默认 200000）

3.5.6.2 查询调度状态

你可以通过观察 PD 相关的 Metrics 或使用pd-ctl 工具等方式查看调度系统状态。更具体的信息可以参考PD 监控和PD Control。

3.5.6.2.1 Operator 状态

Grafana PD/Operator 页面展示了 Operator 的相关统计信息。其中比较重要的有：

- Schedule Operator Create：Operator 的创建情况
- Operator finish duration：Operator 执行耗时的情况
- Operator Step duration：不同 Operator Step 执行耗时的情况

查询 Operator 的 pd-ctl 命令有：

- `operator show`：查询当前调度生成的所有 Operator
- `operator show [admin | leader | region]`：按照类型查询 Operator

3.5.6.2.2 Balance 状态

Grafana PD/Statistics - Balance 页面展示了负载均衡的相关统计信息，其中比较重要的有：

- Store Leader/Region score：每个 Store 的得分
- Store Leader/Region count：每个 Store 的 Leader/Region 数量
- Store available：每个 Store 的剩余空间

使用pd-ctl 的 `store` 命令可以查询 Store 的得分、数量、剩余空间和 weight 等信息。

3.5.6.2.3 热点调度状态

Grafana PD/Statistics - hotspot 页面展示了热点Region 的相关统计信息，其中比较重要的有：

- Hot write Region’s leader/peer distribution：写热点Region 的 Leader/Peer 分布情况
- Hot read Region’s leader distribution：读热点Region 的 Leader 分布情况

使用pd-ctl 同样可以查询上述信息，可以使用的命令有：

- `hot read`：查询读热点 Region 信息
- `hot write`：查询写热点 Region 信息
- `hot store`：按 Store 统计热点分布情况
- `region topread [limit]`：查询当前读流量最大的 Region
- `region topwrite [limit]`：查询当前写流量最大的 Region

3.5.6.2.4 Region 健康度

Grafana PD/Cluster/Region health 面板展示了异常Region 的相关统计信息，包括 Pending Peer、Down Peer、Offline Peer，以及副本数过多或过少的Region。

通过pd-ctl 的 `region check` 命令可以查看具体异常的Region 列表：

- `region check miss-peer`：缺副本的 Region
- `region check extra-peer`：多副本的 Region
- `region check down-peer`：有副本状态为 Down 的 Region
- `region check pending-peer`：有副本状态为 Pending 的 Region

3.5.6.3 调度策略控制

使用pd-ctl 可以从以下三个方面来调整 PD 的调度策略。更具体的信息可以参考PD Control。

3.5.6.3.1 启停调度器

pd-ctl 支持动态创建和删除 Scheduler，你可以通过这些操作来控制 PD 的调度行为，如：

- `scheduler show`：显示当前系统中的 Scheduler
- `scheduler remove balance-leader-scheduler`：删除（停用）balance region 调度器
- `scheduler add evict-leader-scheduler 1`：添加移除 Store 1 的所有 Leader 的调度器

3.5.6.3.2 手动添加 Operator

PD 支持直接通过pd-ctl 来创建或删除 Operator，如：

- `operator add add-peer 2 5`：在 Store 5 上为Region 2 添加 Peer
- `operator add transfer-leader 2 5`：将Region 2 的 Leader 迁移至 Store 5
- `operator add split-region 2`：将Region 2 拆分为 2 个大小相当的Region
- `operator remove 2`：取消Region 2 当前待执行的 Operator

3.5.6.3.3 调度参数调整

使用 pd-ctl 执行 `config show` 命令可以查看所有的调度参数，执行 `config set {key} {value}` 可以调整对应参数的值。常见调整如下：

- `leader-schedule-limit`：控制 Transfer Leader 调度的并发数
- `region-schedule-limit`：控制增删 Peer 调度的并发数
- `enable-replace-offline-replica`：开启节点下线的调度
- `enable-location-replacement`：开启调整 Region 隔离级别相关的调度
- `max-snapshot-count`：每个 Store 允许的最大收发 Snapshot 的并发数

3.5.6.4 典型场景分析与处理

该部分通过几个典型场景及其应对方式说明 PD 调度策略的最佳实践。

3.5.6.4.1 Leader/Region 分布不均衡

PD 的打分机制决定了一般情况下，不同 Store 的 Leader Count 和 Region Count 不能完全说明负载均衡状态，所以需要从 TiKV 的实际负载或者存储空间占用来判断是否有负载不均衡的状况。

确认 Leader/Region 分布不均衡后，首先观察不同 Store 的打分情况。

如果不同 Store 的打分是接近的，说明 PD 认为此时已经是均衡状态了，可能的原因有：

- 存在热点导致负载不均衡。可以参考热点分布不均匀中的解决办法进行分析处理。
- 存在大量空Region 或小Region，因此不同 Store 的 Leader 数量差别特别大，导致Raftstore 负担过重。此时需要开启Region Merge 并尽可能加速合并。
- 不同 Store 的软硬件环境存在差异。可以参考负载均衡一节视实际情况调整 `leader-weight` 和 `region-weight` 来控制 Leader/Region 的分布。
- 其他不明原因。仍可以通过调整 `leader-weight` 和 `region-weight` 来控制 Leader/Region 的分布。

如果不同 Store 的分数差异较大，需要进一步检查 Operator 的相关 Metrics，特别关注 Operator 的生成和执行情况，这时大体上又分两种情况：

- 生成的调度是正常的，但是调度的速度很慢。可能的原因有：
  - 调度速度受限于limit 配置。PD 默认配置的limit 比较保守，在不对正常业务造成显著影响的前提下，可以酌情将 `leader-schedule-limit` 或 `region-schedule-limit` 调大一些。此外，`max-pending-peer-count` 以及 `max-snapshot-count` 限制也可以放宽。
  - 系统中同时运行有其他的调度任务产生竞争，导致balance 速度上不去。这种情况下如果balance 调度的优先级更高，可以先停掉其他的调度或者限制其他调度的速度。例如Region 没均衡的情况下做下线节点操作，下线的调度与Region Balance 会抢占 `region-schedule-limit` 配额，此时你可以调小 `replica-schedule-limit` 以限制下线调度的速度，或者设置 `enable-replace-offline-replica = false` 来暂时关闭下线流程。
  - 调度执行得太慢。可以通过 Operator step duration 进行判断。通常不涉及到收发 Snapshot 的 Step（比如 `TransferLeader`，`RemovePeer`，`PromoteLearner` 等）的完成时间应该在毫秒级，涉及到 Snapshot 的 Step（如 `AddLearner`，`AddPeer` 等）的完成时间为数十秒。如果耗时明显过高，可能是TiKV 压力过大或者网络等方面的瓶颈导致的，需要具体情况具体分析。

- 没能生成对应的balance 调度。可能的原因有：

  - 调度器未被启用。比如对应的 Scheduler 被删除了，或者limit 被设置为“0”。
  - 由于其他约束无法进行调度。比如系统中有 `evict-leader-scheduler`，此时无法把 Leader 迁移至对应的 Store。再比如设置了 Label property，也会导致部分 Store 不接受 Leader。
  - 集群拓扑的限制导致无法均衡。比如 3 副本 3 数据中心的集群，由于副本隔离的限制，每个Region 的 3 个副本都分别分布在不同的数据中心，假如这 3 个数据中心的 Store 数不一样，最后调度就会收敛在每个数据中心均衡，但是全局不均衡的状态。

3.5.6.4.2 节点下线速度慢

这个场景需要从 Operator 相关的 Metrics 入手，分析 Operator 的生成执行情况。

如果调度在正常生成，只是速度很慢，可能的原因有：

- 调度速度受限于 limit 配置。可以适当调大 `replica-schedule-limit`，`max-pending-peer-count` 以及 `max-snapshot-count` 限制也可以放宽。
- 系统中同时运行有其他的调度任务产生竞争。处理方法参考Leader/Region 分布不均衡。
- 下线单个节点时，由于待操作的 Region 有很大一部分（3 副本配置下约 1/3）的 Leader 都集中在下线的节点上，下线速度会受限于这个单点生成 Snapshot 的速度。你可以通过手动给该节点添加一个 `evict-leader-scheduler` 调度器迁走 Leader 来加速。

如果没有对应的 Operator 调度生成，可能的原因有：

- 下线调度被关闭，或者 `replica-schedule-limit` 被设为“0”。
- 找不到节点来转移Region。例如相同 Label 的替代节点可用容量都不足 20%，PD 为了避免爆盘的风险会停止调度。这种情况需要添加更多节点，或者删除一些数据释放空间。

3.5.6.4.3 节点上线速度慢

目前 PD 没有对节点上线特殊处理。节点上线实际上是依靠 balance region 机制来调度的，所以参考Leader/Region 分布不均衡中的排查步骤即可。

3.5.6.4.4 热点分布不均匀

热点调度的问题大体上可以分为以下几种情况：

- 从 PD 的 Metrics 能看出来有不少hot Region，但是调度速度跟不上，不能及时地把热点Region 分散开来。

解决方法：调大 `hot-region-schedule-limit` 并减少其他调度器的limit 配额，从而加快热点调度的速度。还可调小 `hot-region-cache-hits-threshold` 使 PD 对更快响应流量的变化。

- 单一Region 形成热点，比如大量请求频繁 scan 一个小表，这个可以从业务角度或者 Metrics 统计的热点信息看出来。由于单Region 热点现阶段无法使用打散的手段来消除，需要确认热点Region 后手动添加 `split-region` 调度将这样的Region 拆开。

- 从 PD 的统计来看没有热点，但是从TiKV 的相关 Metrics 可以看出部分节点负载明显高于其他节点，成为整个系统的瓶颈。这是因为目前 PD 统计热点Region 的维度比较单一，仅针对流量进行分析，在某些场景下无法准确定位热点。例如部分Region 有大量的点查请求，从流量上来看并不显著，但是过高的 QPS 导致关键模块达到瓶颈。

  解决方法：首先从业务层面确定形成热点的table，然后添加 `scatter-range-scheduler` 调度器使这个 table 的所有Region 均匀分布。TiDB 也在其 HTTP API 中提供了相关接口来简化这个操作，具体可以参考 TiDB HTTP API 文档。

3.5.6.4.5 Region Merge 速度慢

Region Merge 速度慢也很有可能是受到limit 配置的限制（`merge-schedule-limit` 及 `region-schedule-limit`），或者是与其他调度器产生了竞争。具体来说，可有如下处理方式：

- 假如已经从相关 Metrics 得知系统中有大量的空 Region，这时可以通过把 `max-merge-region-size` 和 `max-merge-region-keys` 调整为较小值来加快 Merge 速度。这是因为 Merge 的过程涉及到副本迁移，所以 Merge 的 Region 越小，速度就越快。如果生成 Merge Operator 的速度很快，想进一步加快Region Merge 过程，还可以把 `patrol-region-interval` 调整为“10ms”（从 v5.3.0 起，此配置项默认值为“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: 使用 PD Control，根据集群情况选择性地设置以下参数。
    * 如果集群中不存在TiDB 实例，将key-type 的值设置为 `raw` 或 `txn`。此时，无论 `enable-cross-table-merge` 设置如何，PD 均可以跨表合并Region。该参数支持动态修改。

```
config set key-type txn
```

    * 如果集群中存在TiDB 实例，将 `key-type` 的值设置为 `table`。此时将 `enable-cross-table-merge` 设置为 `true`，可以使 PD 跨表合并Region。该参数支持动态修改。

```
config set enable-cross-table-merge true
```

  如果修改未生效，请参阅FAQ - 修改TiKV/PD 的 toml 配置文件后没有生效。

> **注意：**
> 在开启 `placement-rules` 后，请合理切换 `key-type`，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。

3.5.6.4.6 TiKV 节点故障处理策略

没有人工介入时，PD 处理TiKV 节点故障的默认行为是，等待半小时之后（可通过 `max-store-down-time` 配置调整），将此节点设置为 Down 状态，并开始为涉及到的Region 补充副本。

实践中，如果能确定这个节点的故障是不可恢复的，可以立即做下线处理，这样 PD 能尽快补齐副本，降低数据丢失的风险。与之相对，如果确定这个节点是能恢复的，但可能半小时之内来不及，则可以把 `max-store-down-time` 临时调整为比较大的值，这样能避免超时之后产生不必要的副本补充，造成资源浪费。

自 v5.2.0 起，TiKV 引入了慢节点检测机制。通过对TiKV 中的请求进行采样，计算出一个范围在 1~100 的分数。当分数大于等于 80 时，该TiKV 节点会被设置为 slow 状态。可以通过添加`evict-slow-store-scheduler` 来针对慢节点进行对应的检测和调度。当检测到有且只有一个TiKV 节点为慢节点，并且该TiKV 的 slow score 到达上限（默认 100）时，将节点上的leader 驱逐（其作用类似于 `evict-leader-scheduler`）。

3.5.7 海量Region 集群调优最佳实践

在 TiDB 的架构中，所有数据以一定 key range 被切分成若干Region 分布在多个TiKV 实例上。随着数据的写入，一个集群中会产生上百万个甚至千万个Region。单个TiKV 实例上产生过多的Region 会给集群带来较大的负担，影响整个集群的性能表现。

本文将介绍TiKV 核心模块Raftstore 的工作流程，海量Region 导致性能问题的原因，以及优化性能的方法。

3.5.7.1 Raftstore 的工作流程

一个TiKV 实例上有多个Region。Region 消息是通过Raftstore 模块驱动Raft 状态机来处理的。这些消息包括 Region 上读写请求的处理、Raft log 的持久化和复制、Raft 的心跳处理等。但是，Region 数量增多会影响整个集群的性能。为了解释这一点，需要先了解TiKV 的核心模块Raftstore 的工作流程。

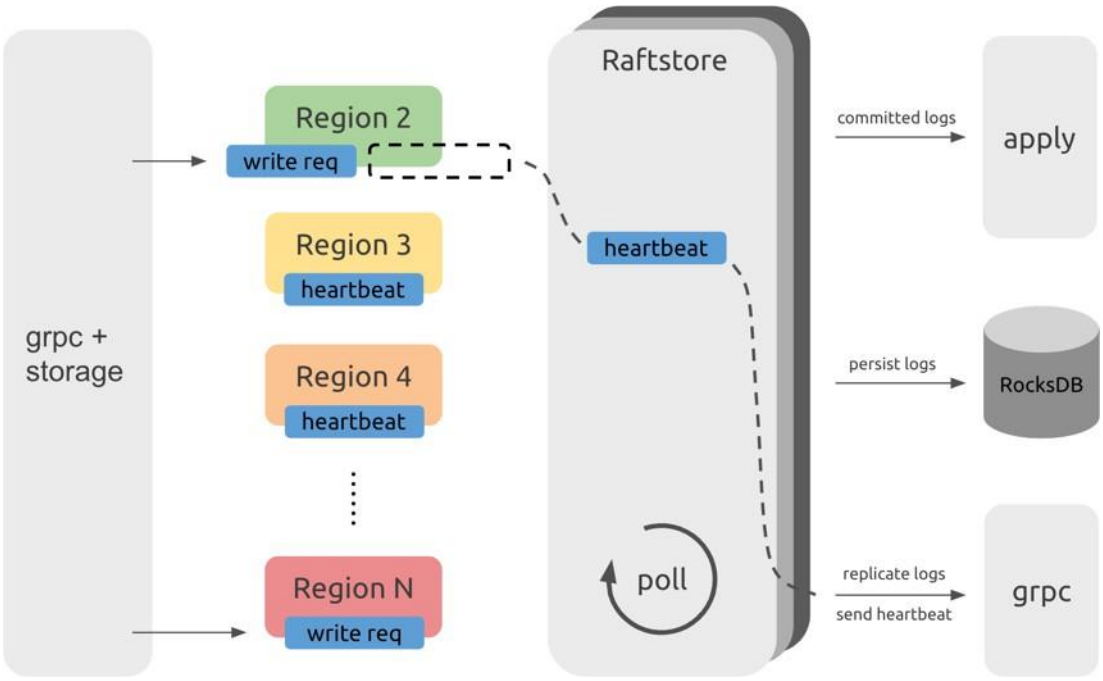


图 38: 图 1 Raftstore 处理流程示意图

> 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。

3.5.7.2 性能问题

从 Raftstore 处理流程示意图可以看出，需要依次处理各个Region 的消息。那么在Region 数量较多的情况下，Raftstore 需要花费一些时间去处理大量Region 的心跳，从而带来一些延迟，导致某些读写请求得不到及时处理。如果读写压力较大，Raftstore 线程的 CPU 使用率容易达到瓶颈，导致延迟进一步增加，进而影响性能表现。

通常在有负载的情况下，如果Raftstore 的 CPU 使用率达到了 85% 以上，即可视为达到繁忙状态且成为了瓶颈，同时 `propose wait duration` 可能会高达百毫秒级别。

> 注意：
>
> - Raftstore 的 CPU 使用率是指单线程的情况。如果是多线程 Raftstore，可等比例放大使用率。
> - 由于Raftstore 线程中有I/O 操作，所以 CPU 使用率不可能达到 100%。

3.5.7.2.1 性能监控

可在Grafana 的 TiKV 面板下查看相关的监控metrics：

- Thread-CPU 下的 `Raft store CPU`

  参考值：低于 `raftstore.store-pool-size * 85%`。

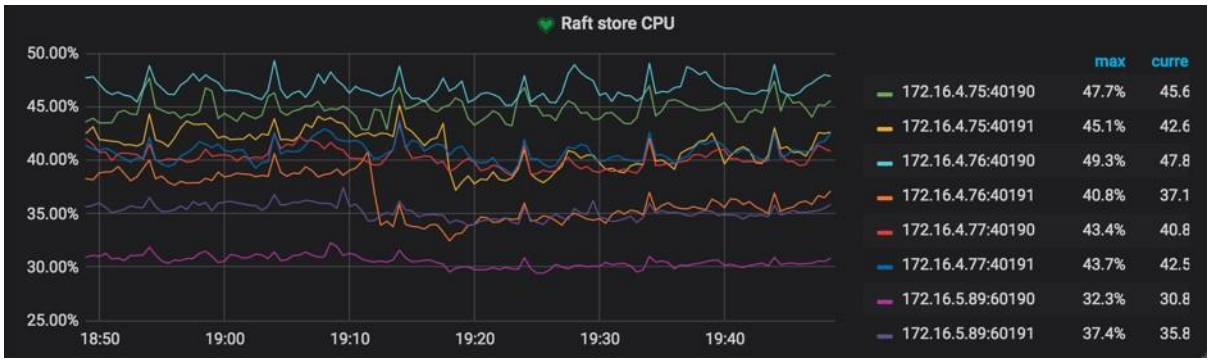


图 39: 图 2 查看Raftstore CPU

- 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。

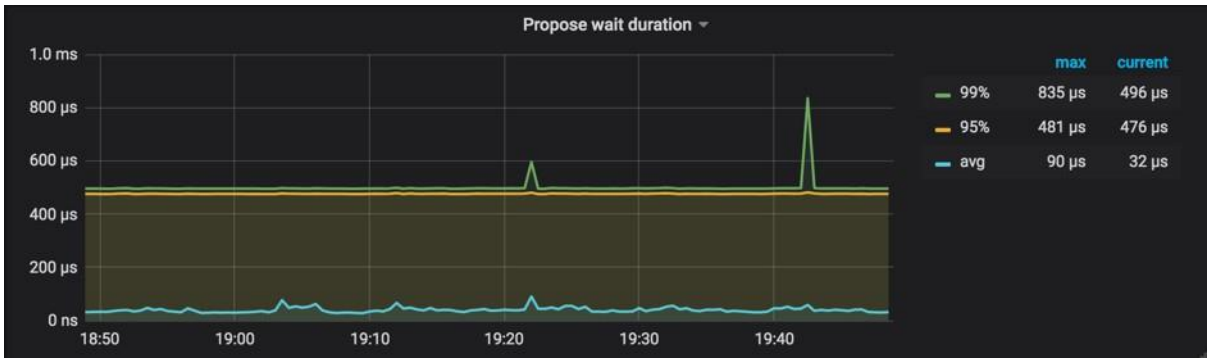


图 40: 图 3 查看 Propose wait duration

3.5.7.3 性能优化方法

找到性能问题的根源后，可从以下两个方向来解决性能问题：

- 减少单个TiKV 实例的Region 数
- 减少单个Region 的消息数

3.5.7.3.1 方法一：增加TiKV 实例

如果I/O 资源和 CPU 资源都比较充足，可在单台机器上部署多个TiKV 实例，以减少单个TiKV 实例上的Region 个数；或者增加TiKV 集群的机器数。

3.5.7.3.2 方法二：调整 `raft-base-tick-interval`

除了减少Region 个数外，还可以通过减少Region 单位时间内的消息数量来减小Raftstore 的压力。例如，在TiKV 配置中适当调大 `raft-base-tick-interval`：

```
[raftstore]
raft-base-tick-interval = "2s"
```

`raft-base-tick-interval` 是 Raftstore 驱动每个Region 的 Raft 状态机的时间间隔，也就是每隔该时长就需要向 Raft 状态机发送一个tick 消息。增加该时间间隔，可以有效减少Raftstore 的消息数量。

需要注意的是，该tick 消息的间隔也决定了 `election timeout` 和 `heartbeat` 的间隔。示例如下：

```
raft-election-timeout = raft-base-tick-interval * raft-election-timeout-ticks
raft-heartbeat-interval = raft-base-tick-interval * raft-heartbeat-ticks
```

如果 Region Follower 在 `raft-election-timeout` 间隔内未收到来自 Leader 的心跳，就会判断 Leader 出现故障而发起新的选举。`raft-heartbeat-interval` 是 Leader 向 Follower 发送心跳的间隔，因此调大 `raft-base-tick-interval` 可以减少单位时间内Raft 发送的网络消息，但也会让Raft 检测到 Leader 故障的时间更长。

3.5.7.3.3 方法三：提高Raftstore 并发数

从 v3.0 版本起，Raftstore 已经扩展为多线程，极大降低了Raftstore 线程成为瓶颈的可能性。

TiKV 默认将 `raftstore.store-pool-size` 配置为 2。如果Raftstore 出现瓶颈，可以根据实际情况适当调高该参数值，但不建议设置过高以免引入不必要的线程切换开销。

3.5.7.3.4 方法四：开启 Hibernate Region 功能

在实际情况中，读写请求并不会均匀分布到每个Region 上，而是集中在少数的Region 上。那么可以尽量减少暂时空闲的Region 的消息数量，这也就是 Hibernate Region 的功能。无必要时可不进行 `raft-base-tick`，即不驱动空闲Region 的 Raft 状态机，那么就不会触发这些Region 的 Raft 产生心跳信息，极大地减小了Raftstore 的工作负担。

Hibernate Region 在TiKV master 分支上默认开启。可根据实际情况和需求来配置此功能的开启和关闭，请参阅配置 Hibernate Region。

3.5.7.3.5 方法五：开启 Region Merge

> 注意：
>
> 从 TiDB v3.0 开始，`Region Merge` 默认开启。

开启Region Merge 也能减少Region 的个数。与Region Split 相反，Region Merge 是通过调度把相邻的小Region 合并的过程。在集群中删除数据或者执行 `Drop Table/Truncate Table` 语句后，可以将小Region 甚至空Region 进行合并以减少资源的消耗。

通过 pd-ctl 设置以下参数即可开启 `Region Merge`：

```
config set max-merge-region-size 20
config set max-merge-region-keys 200000
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。

3.5.7.3.6 方法六：调整Region 大小

Region 默认的大小约为 96 MiB，将其调大也可以减少 Region 个数，具体介绍可参考使用更大的Region。

> 警告：
>
> 自定义Region 大小是在TiDB v6.1.0 引入的实验特性，不建议在生产环境中配置。使用此特性的风险包括：

- 更容易发生性能抖动。
- 查询性能回退，尤其是大范围数据查询的性能会有回退。
- 调度变慢。

3.5.7.4 其他问题和解决方案

3.5.7.4.1 切换 PD Leader 的速度慢

PD 需要将Region Meta 信息持久化在 etcd 上，以保证切换 PD Leader 节点后 PD 能快速继续提供Region 路由服务。随着 Region 数量的增加，etcd 出现性能问题，使得 PD 在切换 Leader 时从 etcd 获取Region Meta 信息的速度较慢。在百万Region 量级时，从 etcd 获取信息的时间可能需要十几秒甚至几十秒。

因此从 v3.0 版本起，PD 默认开启配置项 `use-region-storage`，将 Region Meta 信息存在本地的 LevelDB 中，并通过其他机制同步 PD 节点间的信息。

3.5.7.4.2 PD 路由信息更新不及时

在 TiKV 中，pd-worker 模块将Region Meta 信息定期上报给 PD，在 TiKV 重启或者切换Region Leader 时需要通过统计信息重新计算Region 的 `approximate size/keys`。因此在 Region 数量较多的情况下，pd-worker 单线程可能成为瓶颈，造成任务得不到及时处理而堆积起来。因此 PD 不能及时获取某些Region Meta 信息以致路由信息更新不及时。该问题不会影响实际的读写，但可能导致 PD 调度不准确以及TiDB 更新Region cache 时需要多几次 round-trip。

可在TiKV Grafana 面板中查看 Task 下的Worker pending tasks 来确定pd-worker 是否有任务堆积。通常来说，pending tasks 应该维持在一个比较低的值。

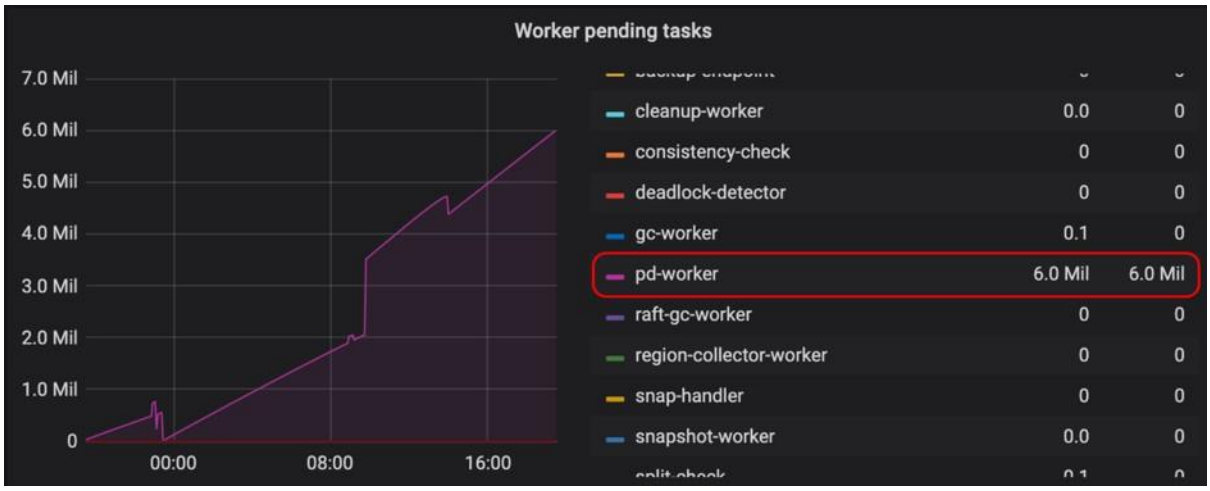


图 41: 图 4 查看pd-worker

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。

3.5.7.4.3 Prometheus 查询metrics 的速度慢

在大规模集群中，随着 TiKV 实例数的增加，Prometheus 查询metrics 时的计算压力较大，导致Grafana 查看metrics 的速度较慢。从 v3.0 版本起设置了一些metrics 的预计算，让这个问题有所缓解。

3.5.8 三节点混合部署的最佳实践

在对性能要求不高且需要控制成本的场景下，将TiDB、 TiKV、 PD 混合部署在三台机器上是一个可行的方案。

本文以 TPC-C 作为工作负载，提供一些在三节点混合部署场景下部署和参数调整的建议。

3.5.8.1 部署环境和测试方法

三台物理机，每台机器有 16 个 CPU 核心且内存为 32 GB。每节点（台机器）上混合部署 1 TiDB （实例）+ 1 TiKV （实例）+ 1 PD （实例）。

由于 PD 和TiKV 都会存储信息到磁盘，磁盘的写入读取延迟会直接影响到 PD 和TiKV 的服务延迟，为了防止 PD 和TiKV 对磁盘资源的争抢导致相互影响，推荐 PD 和TiKV 采用不同的磁盘。

在 TiUP bench 中使用 TPC-C 5000 Warehouse 数据，每次以 `terminals` 参数为 128 的并发度测试 12 小时。主要观察集群性能的稳定性指标。

下图是默认参数配置下，12 小时内集群的 QPS 监控，可以看倒有比较明显的抖动。



图 42: QPS with default config

调整参数后，稳定性得到了改善。



图 43: QPS with modified config

3.5.8.2 参数调整

上图中出现抖动的主要原因是，默认的线程池和后台任务资源分配针对的是资源比较充足的机器。在混合部署的场景下，因为可用资源需要被多个组件共享，所以需要通过配置参数进行限制。

本次测试最终采用的配置如下：

```
tikv:
    readpool.unified.max-thread-count: 6
    server.grpc-concurrency: 2
    storage.scheduler-worker-pool-size: 2
    gc.max-write-bytes-per-sec: 300K
    rocksdb.max-background-jobs: 3
    rocksdb.max-sub-compactions: 1
    rocksdb.rate-bytes-per-sec: “200M”

  tidb:
    performance.max-procs: 8
```

接下来会分别介绍这几个参数的意义和调整方法。

3.5.8.2.1 TiKV 线程池大小配置

本节的配置项会调整一些与前台业务有关的线程池的资源配比。缩减这些线程池会损失一些性能，但在混合部署场景下可用资源有限，本身也难以达到很高的性能，所以选择牺牲一小部分性能去换取整体的稳定性。

可以先以默认配置为基础运行一次实际负载的测试，观察各线程池的实际使用量，再修改这些配置项，缩减利用率不高的线程池大小。

`readpool.unified.max-thread-count`

该参数默认取值为机器线程数的 80%，因为是混合部署场景，你需要手动计算并指定该值。可以先设置成期望 TiKV 使用的 CPU 线程数的 80%。

`server.grpc-concurrency`

该参数默认为 4。因为现有部署方案的 CPU 资源有限，实际请求数也不会很多。可以把该参数值调低，后续观察监控面板保持其使用率在 80% 以下即可。

本次测试最后选择设置该参数值为 2，通过 gRPC poll CPU 面板观察，利用率正好在 80% 左右。

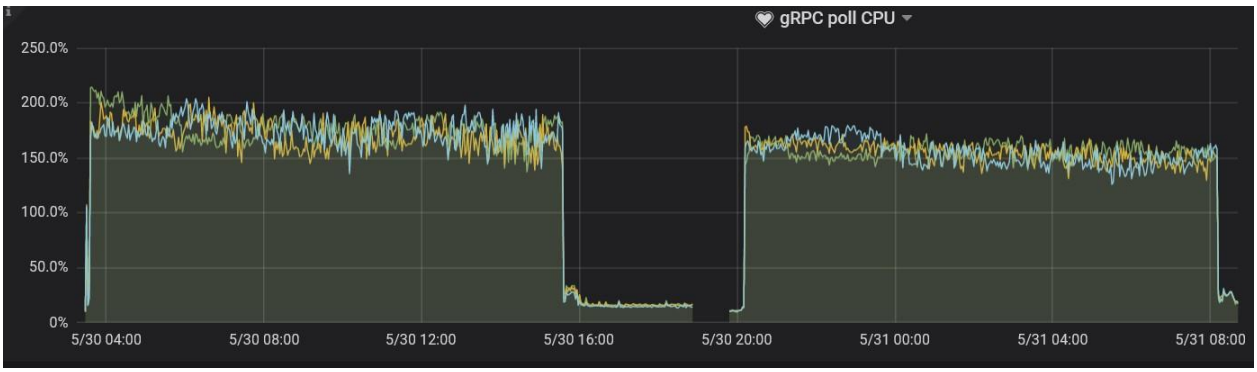


图 44: gRPC Pool CPU

`storage.scheduler-worker-pool-size`

在 TiKV 检测到机器 CPU 数大于或等于 16 时，该参数值默认为 8。CPU 数小于 16 时，该参数值默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 scheduler 线程池本身不进行任何写操作。

一般来说该线程池的利用率保持在 50% - 75% 之间是比较好的。和 gRPC 线程池情况类似，混合部署时该参数默认取值偏大，资源利用不充分。本次测试最后选择取值为 2，通过 Scheduler worker CPU 面板观察，利用率比较符合最佳实践。



图 45: Scheduler Worker CPU

3.5.8.2.2 TiKV 后台任务资源配置

除前台任务之外，TiKV 上还会持续地有后台任务进行数据整理和过期数据的清除。默认配置为这些后台任务分配了比较多的资源以应对大流量的写入。

混合部署场景下，默认配置就不是很合适了，需要通过以下参数对这些后台任务地资源使用量进行限制。

`rocksdb.max-background-jobs` 和 `rocksdb.max-sub-compactions`

RocksDB 线程池是进行 Compact 和 Flush 任务的线程池，默认大小为 8。这明显超出了实际可以使用的资源，需要限制。`rocksdb.max-sub-compactions` 是单个compaction 任务的子任务并发数，默认值为 3，在写入流量不大的情况下可以进行限制。

这次测试最终将 `rocksdb.max-background-jobs` 设置为 3，将 `rocksdb.max-sub-compactions` 设置为 1。在 12 小时的 TPC-C 负载下没有发生Write Stall，根据实际负载进行这两项参数的优化时，可以逐步调低这两个配置，并通过监控观察。

- 如果遇到了Write Stall，可以先调大 `rocksdb.max-background-jobs` 的取值。
- 如果还是存在问题，可将 `rocksdb.max-sub-compactions` 设置为 2 或者 3。

`rocksdb.rate-bytes-per-sec`

该参数用于限制后台compaction 任务的磁盘流量。默认配置下没有进行任何限制。为了避免compaction 挤占前台服务资源，可以根据硬盘的顺序读写速度进行调整，为正常的服务保留足够多的磁盘带宽。

类似compaction 线程池的调整方法，调整该参数值后，也根据是否存在Write Stall 来判断取值是否合理。

`gc.max_write_bytes_per_sec`

因为TiDB 使用的是 MVCC 的模型，TiKV 还需要周期性地在后台清除旧版本的数据。当可用资源有限的时候，这个操作会引起周期性的性能抖动。可以用 `gc.max_write_bytes_per_sec` 来限制这一操作的资源使用。



图 46: GC Impact

除了在配置文件中设置该参数之外，还可以通过tikv-ctl 动态调节，为调整该参数提供便利：

```
tiup ctl:v<CLUSTER_VERSION> tikv --host=${ip:port} modify-tikv-config -n gc.
    <-> max_write_bytes_per_sec -v ${limit}
```

> **注意：**
>
> 对于更新频繁的业务场景，限制GC 流量可能会导致 MVCC 版本堆积，进而影响读取性能。所以这个参数的取值需要进行多次尝试，在性能抖动和性能衰退之间找一个比较平衡的取值。

3.5.8.2.3 TiDB 参数调整

一般通过系统变量调整TiDB 执行算子的优化参数，例如tidb_hash_join_concurrency、tidb_index_lookup_join_concurrency <-> 等。

本测试中没有对这些参数进行调整。如果实际的业务负载测试中，出现执行算子消耗过多 CPU 资源的情况，可以针对业务场景对特定的算子资源使用量进行限制。这部分内容可以参考TiDB 系统变量文档。

performance.max-procs

该参数控制着整个Go 进程能使用的 CPU 核心数量，默认情况下为当前机器或者cgroups 的 CPU 数量。

Go 运行时会定期使用一定比例的线程进行GC 等后台工作。在混合部署模式下，如果不对这一参数进行限制，GC 等后台操作会过多占用 CPU 资源。

3.5.9 在三数据中心下就近读取数据

在三数据中心模式下，Region 的三个副本都会隔离在各个数据中心里。然而在强一致读的要求下，TiDB 的每一个查询都需要访问对应数据的 Leader 副本，而查询的来源可能和 Leader 所在的数据中心不一致，这就会引起跨数据中心的数据访问，从而造成访问的延迟上升。本文主要介绍使用Stale Read 功能，以牺牲数据实时性的方式，避免跨数据中心的访问，从而降低访问的延迟。

3.5.9.1 部署三数据中心的TiDB 集群

部署三数据中心的方法，参考同城多数据中心部署TiDB。

请注意，如果TiKV 和 TiDB 都有 `labels` 配置项，在给TiKV 和 TiDB 配置标签时，同一个数据中心下的TiKV 和 TiDB 应该配置相同的 `zone` 标签。假设TiKV 和TiDB 都在 `dc-1` 数据中心下，那么两者都需要配置如下标签：

```
[labels]
zone=dc-1
```

3.5.9.2 使用 Stale Read 就近读取数据

Stale Read 为用户提供了一种读取历史数据的一种机制。使用 Stale Read 功能，你能从指定时间点或时间范围内读取对应的历史数据，从而避免数据同步带来延迟。在部分跨数据中心部署的场景中使用 Stale Read 功能，通过牺牲一定的实时性，TiDB 可就近访问对应数据所在当前中心的副本，避免跨数据中心的网络延迟，降低整体查询的访问延迟。

当TiDB 收到Stale Read 查询时，假如对应的TiDB 配置了`zone` 标签，而且`tidb_replica_read` 为`closest-replicas`，就会将请求发送到对应数据副本所在TiKV 拥有相同的 `zone` 标签的节点上。

如何使用 Stale Read 查询，参考使用 AS OF TIMESTAMP 语法读取历史数据。

3.5.10 UUID 最佳实践

3.5.10.1 UUID 概述

通用唯一标识符（UUID）用作主键而不是 `AUTO_INCREMENT` 整数值时，可以提供以下好处：

- UUID 可以在多个系统生成，而不会产生冲突。某些情况下可以减少到TiDB 的网络往返次数，从而提高性能。
- 绝大多数编程语言和数据库系统都支持 UUID。
- 用在 URL 中时，UUID 不容易被枚举攻击。相比之下，使用 `auto_increment` 数字，则很容易让发票ID 或用户ID 被猜出。

3.5.10.2 最佳实践

3.5.10.2.1 二进制存储

UUID 文本是一个包含 36 字符的字符串，如 `ab06f63e-8fe7-11ec-a514-5405db7aad56`。使用 `UUID_TO_BIN()` 可将 UUID 文本格式转换为 16 字节的二进制格式。这样，你可以将文本存储在 `BINARY(16)` 列中。检索 UUID 时，可以使用 `BIN_TO_UUID()` 函数再将其转换回文本格式。

3.5.10.2.2 UUID 格式二进制顺序和聚簇主键

UUID_TO_BIN() 函数可以接收一个参数(UUID)或两个参数（第一个为 UUID，第二个为 swap_flag）。建议不要在 TiDB 中设置 swap_flag，以避免出现热点问题。

同时，你也可以在 UUID 主键上显式设置CLUSTERED 选项来避免热点问题。

为了演示 swap_flag 的效果，本文以表结构相同的两张表为例。区别在于，uuid_demo_1 表中插入的数据使用 UUID_TO_BIN(?, 0)，而 uuid_demo_2 表中使用 UUID_TO_BIN(?, 1)。

在如下的流量可视化页面，你可以看到写入操作集中在 uuid_demo_2 表的单个Region 中，而这个表中的二进制格式字段顺序被调换过。

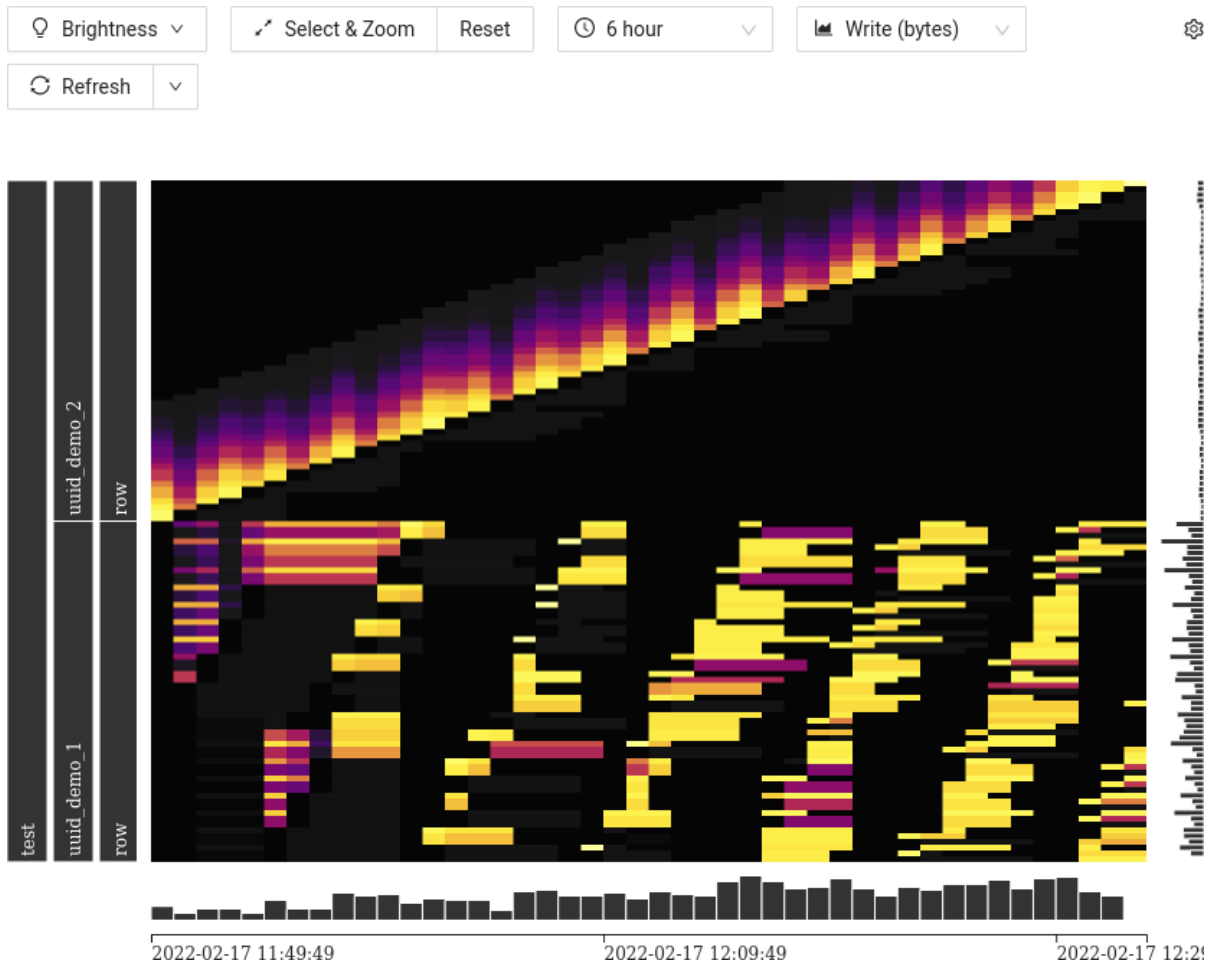


图 47: Key Visualizer

```
CREATE TABLE `uuid_demo_1` (
  `uuid` varbinary(16) NOT NULL,
  `c1` varchar(255) NOT NULL,
  PRIMARY KEY (`uuid`) CLUSTERED
```

```
)
```

```
CREATE TABLE `uuid_demo_2` (
  `uuid` varbinary(16) NOT NULL,
  `c1` varchar(255) NOT NULL,
  PRIMARY KEY (`uuid`) CLUSTERED
)
```

3.5.10.3 与 MySQL 兼容性

UUID 也可以在 MySQL 中使用。MySQL 8.0 引入了 `BIN_TO_UUID()` 和 `UUID_TO_BIN()` 函数。`UUID()` 函数在较早的 MySQL 版本中也可以使用。

3.5.11 只读存储节点最佳实践

本文档介绍如何配置只读存储节点，以及如何将备份、分析、测试等流量导向这些节点，使这些对延迟要求较低的负载与线上重要服务在物理上达到隔离的效果。

3.5.11.1 操作步骤

3.5.11.1.1 1. 将部分TiKV 节点指定为只读节点

通过给TiKV 节点标记特殊label（使用 `$` 作为 label key 的前缀）的方式，可以把部分节点指定为特殊只读节点。除非通过设置 Placement Rules 的方式显式指定这些节点存储某些数据，否则 PD 不会调度任何数据到这些节点上。

只读节点可通过执行 `tiup cluster edit-config` 命令进行配置：

```
tikv_servers:
  - host: ...
    ...
    labels:
      $mode: readonly
```

3.5.11.1.2 2. 将数据以learner 形式存储在只读节点

1. 使用如下 `pd-ctl config placement-rules` 命令导出默认 Placement Rules：

   ```
   pd-ctl config placement-rules rule-bundle load --out="rules.json"
   ```

   如果之前没有配置过 Placement Rules，那么会导出如下内容：

   ```
   [
     {
       "group_id": "pd",
   ```

```
    "group_index": 0,
    "group_override": false,
    "rules": [
      {
        "group_id": "pd",
        "id": "default",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 3
      }
    ]
  }
]
```

2. 将所有数据在只读节点以learner 方式存储一份。如下示例基于默认配置：

```
[
  {
    "group_id": "pd",
    "group_index": 0,
    "group_override": false,
    "rules": [
      {
        "group_id": "pd",
        "id": "default",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 3
      },
      {
        "group_id": "pd",
        "id": "readonly",
        "start_key": "",
        "end_key": "",
        "role": "learner",
        "count": 1,
        "label_constraints": [
          {
            "key": "$mode",
            "op": "in",
            "values": [
              "readonly"
            ]
          }
```

```
        ],
        "version": 1
      }
    ]
  }
]
```

3. 执行 pd-ctl config placement-rules 命令将上面的配置写入 PD：

```
pd-ctl config placement-rules rule-bundle save --in="rules.json"
```

注意：

- 当对已存在大量数据的集群进行如上操作时，整个集群可能需要一段时间才能将数据完全复制到只读节点上。在这期间，只读节点可能无法进行服务。
- 因为备份的特殊下推实现机制，每个label 所对应的learner 数量不能超过 1，否则会导致在备份时产生重复数据。

3.5.11.1.3 3. 使用 Follower Read 功能读取只读节点

3.1 在 TiDB 中使用 Follower Read

你可以将系统变量tidb_replica_read 设置为 learner 来读取只读节点上的数据：

```
set tidb_replica_read=learner;
```

3.2 在 TiSpark 中使用 Follower Read

你可以在 Spark 配置文件中设置 spark.tispark.replica_read = learner 来读取只读节点上的数据：

```
spark.tispark.replica_read learner
```

3.3 在备份集群数据时只备份 Follower 节点

你可以在br 命令行中添加 --replica-read-label 参数，来读取只读节点上的数据。注意，在 shell 中运行如下命令时需使用单引号包裹label，以防止$ 被 shell 解析。

```
br backup full ... --replica-read-label '$mode:readonly'
```

3.6 Placement Rules 使用文档

> **注意：**
>
> 本文介绍如何手动在 Placement Driver (PD) 中设置 Placement Rules。推荐使用Placement Rules in SQL，让你更方便地设置表和分区的放置。

Placement Rules 是 PD 在 4.0 版本引入的一套副本规则系统，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。

Placement Rules 特性在TiDB v5.0 及以上的版本中默认开启。如需关闭 Placement Rules 特性，请参考关闭 Placement Rules。

3.6.1 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，支持更灵活的方式来组织规则，还引入了分组(Group) 的概念。通常情况下，用户可根据规则的不同来源把规则放置在不同的Group。

Placement Rules 示意图如下所示：

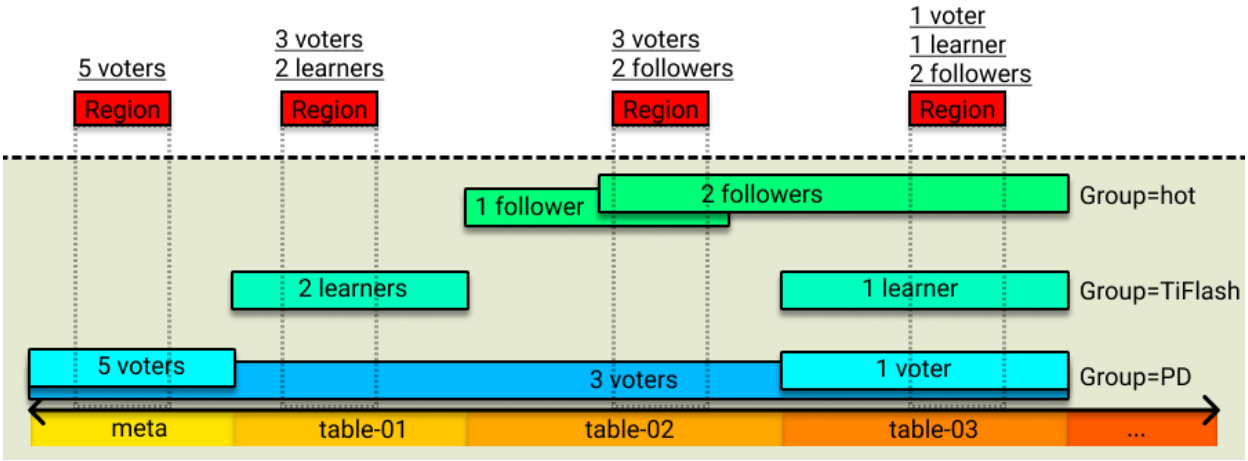


图 48: Placement rules overview

3.6.1.1 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 voter/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
IsolationLevel	string	用于设置最小强制物理隔离级别

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。

IsolationLevel 的意义和作用详细请参考配置集群拓扑。例如已配置 LocationLabels 为 [zone,rack,host] 的前提下，设置 IsolationLevel 为 zone，则 PD 在调度时会保证每个Region 的所有 Peer 均被放置在不同的zone。假如无法满足 IsolationLevel 的最小强制隔离级别限制（比如配置 3 副本但总共只有 2 个 zone），PD 也不会尝试补足，以满足该限制。IsolationLevel 默认值为空字符串，即禁用状态。

3.6.1.2 规则分组字段

以下是规则分组字段的含义：

字段名	类型及约束	说明
ID	string	分组ID，用于标识规则来源
Index	int	不同分组的堆叠次序
Override	true/false	是否覆盖index 更小的分组

如果不单独设置规则分组，默认 Override=false，对应的行为是不同分组之间相互不影响，不同分组内的规则是同时生效的。

3.6.2 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。

3.6.2.1 开启 Placement Rules 特性

Placement Rules 特性在TiDB v5.0 及以上的版本中默认开启。如需关闭 Placement Rules 特性，请参考关闭 Placement Rules。如需在关闭后重新开启该特性，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"],
  "isolation_level": ""
}
```

如果是已经初始化过的集群，也可以通过pd-ctl 进行在线开启：

```
pd-ctl config placement-rules enable
```

PD 同样将根据系统的 max-replicas 及 location-labels 生成默认的规则。

> **注意：**
>
> 开启 Placement Rules 后，原先的 max-replicas 及 location-labels 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

3.6.2.2 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。

3.6.2.3 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
```

3.6.2.4 使用pd-ctl 设置规则分组

- 查看所有的规则分组列表

```
pd-ctl config placement-rules rule-group show
```

- 查看指定ID 的规则分组

```
pd-ctl config placement-rules rule-group show pd
```

- 设置规则分组的index 和 override 属性

```
pd-ctl config placement-rules rule-group set pd 100 true
```

- 删除规则分组配置（如组内还有规则，则使用默认分组配置）

```
pd-ctl config placement-rules rule-group delete pd
```

3.6.2.5 使用pd-ctl批量更新分组及组内规则

使用 `rule-bundle` 子命令，可以方便地同时查看和修改规则分组及组内的所有规则。

该子命令中 `get {group_id}` 用来查询一个分组，输出结果为嵌套形式的规则分组和组内规则：

```
pd-ctl config placement-rules rule-bundle get pd
```

输出示例：

```
{
  "group_id": "pd",
  "group_index": 0,
  "group_override": false,
  "rules": [
    {
      "group_id": "pd",
      "id": "default",
      "start_key": "",
      "end_key": "",
      "role": "voter",
      "count": 3
    }
  ]
}
```

`rule-bundle get` 子命令中可以添加 `--out` 参数来将输出写入文件，方便后续修改保存。

```
pd-ctl config placement-rules rule-bundle get pd --out="group.json"
```

修改完成后，使用 `rule-bundle set` 子命令将文件中的配置保存至 PD 服务器。与前面介绍的 `save` 不同，此命令会替换服务器端该分组内的所有规则。

```
pd-ctl config placement-rules rule-bundle set pd --in="group.json"
```

3.6.2.6 使用pd-ctl查看和修改所有配置

用户还可以使用pd-ctl查看和修改所有配置，即把全部配置保存至文件，修改后再覆盖保存。该操作同样使用 `rule-bundle` 子命令。

下面的命令将所有配置保存至 `rules.json` 文件：

```
pd-ctl config placement-rules rule-bundle load --out="rules.json"
```

编辑完文件后，使用下面的命令将配置保存至 PD 服务器：

```
pd-ctl config placement-rules rule-bundle save --in="rules.json"
```

3.6.2.7 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。

3.6.3 典型场景示例

本部分介绍Placement Rules的使用场景示例。

3.6.3.1 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"]
}
```

3.6.3.2 场景二：5 副本按 2-2-1 的比例放置在 3 个数据中心，且第 3 个中心不产生 Leader

创建三条规则，分别设置副本数为 2、2、1，并且在每个规则内通过 `label_constraints` 将副本限定在对应的数据中心内。另外，不需要leader 的数据中心将 `role` 改为 `follower`。

```
[
    {
        "group_id": "pd",
        "id": "zone1",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone1"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone2",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone2"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone3",
        "start_key": "",
        "end_key": "",
        "role": "follower",
        "count": 1,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone3"]}
        ],
        "location_labels": ["rack", "host"]
    }
]
```

3.6.3.3 场景三：为某张表添加 2 个 TiFlash Learner 副本

为表的 row key 单独添加一条规则，限定数量为 2，并且通过 label_constraints 保证副本产生在 engine=tiflash 的节点。注意这里使用了单独的 group_id，保证这条规则不会与系统中其他来源的规则互相覆盖或产生冲突。

```
{
  "group_id": "tiflash",
  "id": "learner-replica-table-ttt",
  "start_key": "7480000000000000ff2d5f720000000000fa",
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "learner",
  "count": 2,
  "label_constraints": [
    {"key": "engine", "op": "in", "values": ["tiflash"]}
  ],
  "location_labels": ["host"]
}
```

3.6.3.4 场景四：为某张表在有高性能磁盘的北京节点添加 2 个 Follower 副本

这个例子展示了比较复杂的 label_constraints 配置，下面的例子限定了副本放置在bj1 或 bj2 机房，且磁盘类型为 nvme。

```
{
  "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": "in", "values": ["nvme"]}
  ],
  "location_labels": ["host"]
}
```

3.6.3.5 场景五：将某张表迁移至 SSD 节点

与场景三不同，这个场景不是要在原有配置的基础上增加新副本，而是要强制覆盖一段数据的其它配置，因此需要通过配置规则分组来指定一个足够大的index 以及设置 override 来覆盖原有规则。

规则：

```
{
  "group_id": "ssd-override",
  "id": "ssd-table-45",
  "start_key": "7480000000000000ff2d5f720000000000fa",
```

```
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "voter",
  "count": 3,
  "label_constraints": [
    {"key": "disk", "op": "in", "values": ["ssd"]}
  ],
  "location_labels": ["rack", "host"]
}
```

规则分组：

```
{
  "id": "ssd-override",
  "index": 1024,
  "override": true,
}
```

3.7 Load Base Split

Load Base Split 是 TiKV 在 4.0 版本引入的特性，旨在解决Region 访问分布不均匀造成的热点问题，比如小表的全表扫描。

3.7.1 场景描述

在TiDB 中，当流量集中在某些节点时很容易形成热点。PD 会尝试通过调度 Hot Region，尽可能让这些 Hot Region 均匀分布在各个节点上，以求获得更好的性能。

但是 PD 的调度的最小粒度是Region。如果集群的热点数目少于节点数目，或者说存在某几个热点流量远高于其他Region，对 PD 的热点调度来说，能做到的也只是让热点从一个节点转移到另一个节点，而无法让整个集群承担负载。

这种场景在读请求居多的 workload 中尤为常见。例如对小表的全表扫描和索引查找，或者是对一些字段的频繁访问。

在此之前解决此类问题的办法是手动输入命令去拆分一个或几个热点Region，但是这样的操作存在以下两个问题：

- 均匀拆分Region 并不一定是最好的选择，请求可能集中在某几个 Key 上，即使均匀拆分后热点可能仍然集中在其中一个Region 上，可能需要经过多次均匀拆分才能达到目标。
- 人工介入不够及时和易用。

3.7.2 实现原理

Load Base Split 会基于统计信息自动拆分Region。通过统计信息识别出读流量或 CPU 使用率在 10s 内持续超过阈值的Region，并在合适的位置将这些Region 拆分。在选择拆分的位置时，会尽可能平衡拆分后两个Region 的访问量，并尽量避免跨Region 的访问。

Load Base Split 后的 Region 不会被迅速 Merge。一方面，PD 的 MergeChecker 会跳过 hot Region，另一方面 PD 也会针对心跳信息中的 QPS 去进行判断，避免 Merge 两个 QPS 很高的 Region。

3.7.3 使用方法

目前的 Load Base Split 的控制参数如下：

- split.qps-threshold：表明一个 Region 被识别为热点的 QPS 阈值。当 region-split-size 小于 4 GB 时，默认为每秒 3000 QPS。当 region-split-size 大于或等于 4 GB 时，默认值为每秒 7000 QPS。
- split.byte-threshold：自 v5.0 引入，表明一个 Region 被识别为热点的流量阈值，单位为 Byte。当 region-split-size 小于 4 GB 时，默认值为每秒 30 MiB 流量。当 region-split-size 大于或等于 4 GB 时，默认值为每秒 100 MiB 流量。
- split.region-cpu-overload-threshold-ratio：自 v6.2.0 引入，表明一个 Region 被识别为热点的 CPU 使用率（占读线程池 CPU 时间的百分比）阈值。当 region-split-size 小于 4 GB 时，默认值为 0.25。当 region-split-size 大于或等于 4 GB 时，默认值为 0.75。

如果连续 10s 内，某个 Region 每秒的各类读请求之和超过了 split.qps-threshold、流量超过了 split.byte-threshold，或 CPU 使用率在 Unified Read Pool 内的占比超过了 split.region-cpu-overload-threshold-ratio，那么就会尝试对此 Region 进行拆分。

目前默认开启 Load Base Split，但配置相对保守。如果想要关闭这个功能，将 QPS 和 Byte 阈值全部调到足够高并将 CPU 占比阈值调为 0 即可。

目前有两种办法修改配置：

- 通过 SQL 语句修改，例如：

```
# 设置 QPS 阈值为 1500
SET config tikv split.qps-threshold=1500;
# 设置 Byte 阈值为 15 MiB (15 * 1024 * 1024)
SET config tikv split.byte-threshold=15728640;
# 设置 CPU 使用率阈值为 50%
SET config tikv split.region-cpu-overload-threshold-ratio=0.5;
```

- 通过 TiKV 修改，例如：

```
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.qps-
    ↪ threshold":"1500"}'
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.byte-
    ↪ threshold":"15728640"}'
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.region
    ↪ -cpu-overload-threshold-ratio":"0.5"}'
```

同理，目前也有两种办法查看配置：

- 通过 SQL 查看，例如：

```
show config where type='tikv' and name like '%split.qps-threshold%'
```

- 通过TiKV 查看，例如：

```
curl "http://ip:status_port/config"
```

> **注意：**
>
> 从 v4.0.0-rc.2 起可以使用 SQL 语句来修改和查看配置。

3.8 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。

3.8.1 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。

3.8.2 使用方法

Store Limit 相关的参数可以通过 pd-ctl 进行设置。

3.8.2.1 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 添加和删除 peer 的速度上限。
store limit add-peer                 // 显示所有 store 添加 peer 的速度上限。
store limit remove-peer              // 显示所有 store 删除 peer 的速度上限。
```

3.8.2.2 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 添加和删除 peer 的速度上限为每分钟 5 个。
store limit all 5 add-peer           // 设置所有 store 添加 peer 的速度上限为每分钟 5 个。
store limit all 5 remove-peer        // 设置所有 store 删除 peer 的速度上限为每分钟 5 个。
```

3.8.2.3 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 添加和删除 peer 的速度上限为每分钟 5 个。
store limit 1 5 add-peer             // 设置 store 1 添加 peer 的速度上限为每分钟 5 个。
store limit 1 5 remove-peer          // 设置 store 1 删除 peer 的速度上限为每分钟 5 个。
```

3.8.3 Store Limit v2 原理

当store-limit-version 设置为 v2 时，Store Limit v2 生效。在此模式下，Operator 调度限制将根据TiKV Snapshot 执行情况进行动态调整。当TiKV 积压的任务较少时，PD 会增加其调度任务。相反，PD 会减少对该节点的调度任务。此时，你无需关注如何设置 store limit 以加快调度进度。

在该模式下，TiKV 执行速度成为迁移进度的主要瓶颈。你可以通过TiKV Details > Snapshot > Snapshot Speed 面板判断当前调度速度是否达到TiKV 限流设置。通过调整 TiKV Snapshot Limit (snap-io-max-bytes-per-sec) 来增加或减少该节点的调度速度。

3.9 DDL 语句的执行原理及最佳实践

本文介绍了TiDB 中 DDL 语句的执行原理（包括 DDL Owner 模块和在线变更 DDL 的流程）和最佳实践。

3.9.1 DDL 执行原理

TiDB 采用在线异步变更的方式执行 DDL 语句，从而实现 DDL 语句的执行不会阻塞其他会话中的 DML 语句。因此，在业务执行过程中，你可以通过在线异步变更 DDL 对数据库对象定义进行变更。

3.9.1.1 DDL 语句类型简介

按照执行期间是否阻塞用户业务，DDL 语句可以划分为：

- 离线 DDL 语句：即数据库接收到用户的 DDL 语句后，会先对要修改的数据库对象进行加锁，再执行元数据变更，在 DDL 执行过程中将阻塞用户业务对数据的修改。
- 在线 DDL 语句：即数据库在执行 DDL 语句时，通过一定的方法，使得 DDL 执行不阻塞用户业务，且能够保证用户业务可在 DDL 执行期间提交修改，在执行过程中保证对应对象的数据正确性与一致性。

按照是否需要操作 DDL 目标对象所包括的数据来划分，DDL 语句可以划分为：

- 逻辑 DDL 语句：通常只修改数据库对象的元数据，不对变更对象存储的数据进行处理，例如变更表名或变更列名。

  在 TiDB 中，逻辑 DDL 语句又被称为General DDL。General DDL 的执行时间通常较短，只需要几十毫秒或者几秒。执行这类 DDL 语句几乎不消耗系统资源，因此不会影响业务负载。

- 物理 DDL 语句：不但会修改变更对象的元数据，同时也修改变更对象所存储的用户数据。例如，为表创建索引，不仅需要变更表的定义，同时也需要做一次全表扫描以构建新增加的索引。

  在 TiDB 中，物理 DDL 被称为Reorg DDL（Reorg 即 Reorganization）。目前物理 DDL 只包含 `ADD INDEX` 以及有损列类型变更（例如从 `INT` 转成 `CHAR` 类型）这两种类型。物理 DDL 的特点是执行时间较长，且执行时间与表的数据量、机器配置以及业务负载有关。

  执行物理 DDL 会影响业务负载，具体有两个方面。一方面需要从TiKV 中读取数据并写入新数据，因此会消耗TiKV 的 CPU 及 I/O 资源。另一方面，DDL Owner 所在的TiDB 节点需要进行相应的计算，因此会消耗更多的 CPU 资源。由于目前TiDB 还不支持分布式执行 DDL 语句，因此其他TiDB 节点不会占用更多的系统资源。

> 注意：
> DDL 语句对用户业务的影响通常都是由于执行物理 DDL 任务造成的。因此要优化 DDL 语句对于用户业务的影响，重点在于对物理 DDL 任务执行期间的设计，降低对于用户业务的影响。

3.9.1.2 TiDB DDL 模块

TiDB DDL 模块引入 DDL Owner（简称 Owner）角色来代理执行所有进入到 TiDB 集群中的 DDL 语句。对于当前 TiDB DDL 模块的实现，在同一时间，整个TiDB 集群中只有一个TiDB 节点能当选为 Owner。当选 Owner 后，TiDB 节点中启动的 worker 才能处理集群中的 DDL 任务。

TiDB 通过 etcd 的选举功能从多个TiDB 节点中选举出一个节点来担任 Owner 的宿主节点。默认情况下，每个 TiDB 节点都可能当选 Owner（你可以通过配置 `run-ddl` 控制某个TiDB 节点是否竞选 Owner）。Owner 节点是有任期的，并会主动维护自己的任期，即续约。当 Owner 节点宕机后，其他节点可以通过 etcd 感知到并重新选举出新的 Owner，在集群中继续担任 DDL 任务执行者的角色。

DDL Owner 的简单示意图如下：

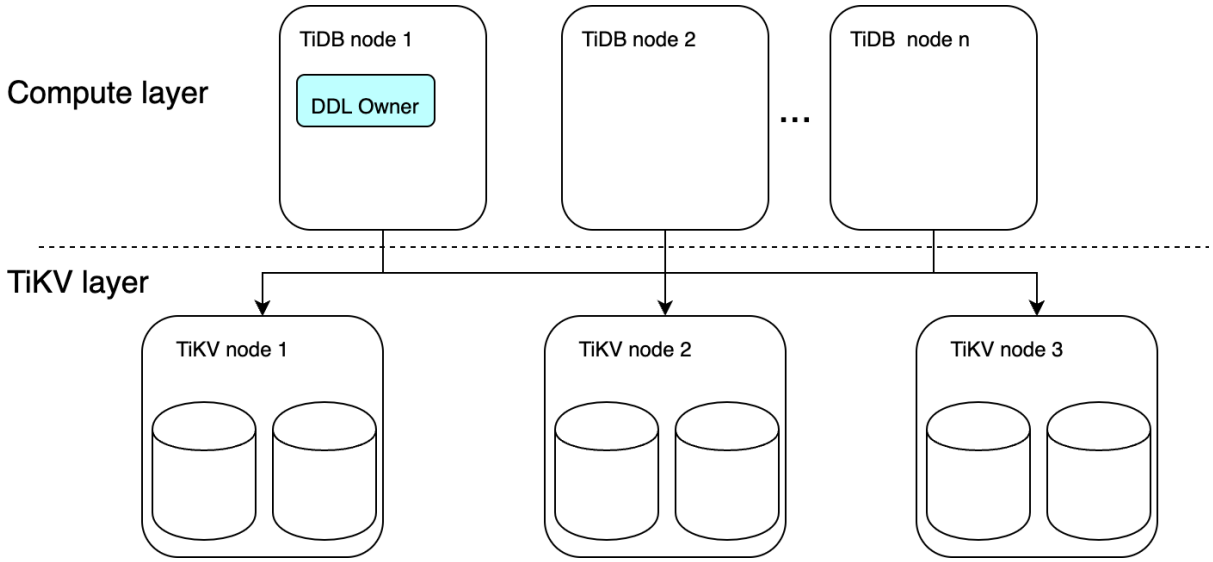


图 49: DDL Owner

你可以通过 ADMIN SHOW DDL 语句查看当前 DDL owner：

```
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)
```

3.9.1.3 TiDB 在线 DDL 异步变更的原理

TiDB DDL 模块从设计之初就选择了在线异步变更的模式，为TiDB 的用户提供了不停机变更业务的服务能力。

DDL 变更即两个状态之间的切换（变更前-> 变更后）。在线变更 DDL，是在两个状态之间引入多个相互兼容的小版本状态。同时，在同一个 DDL 语句在执行期间，对于同一集群中不同TiDB 节点，允许不同节点的变更对象小版本存在不同（集群中各TiDB 节点变更对象的小版本差距不超过两个版本），因为相邻两个小版本之间可以相互兼容。

通过多个小版本演进的方式，确保多个TiDB 节点能够正确同步元数据，并保证期间执行用户事务更改数据的正确性与一致性。

以 ADD INDEX 为例，整个变更状态流程如下：

```
absent -> delete only -> write only -> write reorg -> public
```

对于用户来说，新建的索引在 public 状态前都不可用。

在 v6.2.0 之前，TiDB SQL 层中处理异步 Schema 变更的基本流程如下：

1. MySQL Client 发送给TiDB server 一个 DDL 操作请求。
2. 某个TiDB server 收到请求（即TiDB server 的 MySQL Protocol 层对请求进行解析优化），然后发送到 TiDB SQL 层进行执行。

   TiDB SQL 层接到 DDL 请求后，会启动 start job 模块根据请求将请求封装成特定的 DDL Job（即 DDL 任务），然后将此Job 按语句类型分类，分别存储到 KV 层的对应 DDL Job 队列，并通知自身对应的 worker 有 Job 需要处理。
3. 接收到处理Job 通知的 worker，会判断自身是否处于 DDL Owner 的角色。如果是 Owner 角色则直接处理此 Job。如果没有处于 Owner 角色则退出不做任何处理。

   假设某台TiDB server 不是 Owner 角色，那么其他某个节点一定有一个是 Owner。处于 Owner 角色的节点的 worker 通过定期检测机制来检查是否有Job 可以被执行。如果发现有Job，那么 worker 就会处理该Job。
4. Worker 处理完Job 后，会将此Job 从 KV 层对应的Job queue 中移除，并放入 job history queue。之前封装 Job 的 start job 模块会定期在 job history queue 中查看是否有已经处理完成的Job 的 ID。如果有，则这个Job 对应的整个 DDL 操作结束执行。
5. TiDB server 将 DDL 处理结果返回至 MySQL Client。

在 TiDB v6.2.0 前，该 DDL 执行框架存在以下限制：

- TiKV 集群中只有 general job queue 和 add index job queue 两个队列，分别处理逻辑 DDL 和物理 DDL。
- DDL Owner 总是以先入先出的方式处理 DDL Job。
- DDL Owner 每次只能执行一个同种类型（逻辑或物理）的 DDL 任务，这个约束较为严格。

这些限制可能会导致一些“非预期”的 DDL 阻塞行为。具体可以参考SQL FAQ - DDL 执行。

在TiDB v6.2 之前，由于 Owner 每次只能执行一个同种类型（逻辑或物理）的 DDL 任务，这个约束较为严格，同时影响用户体验。

当 DDL 任务之间不存在相关依赖时，并行执行并不会影响数据正确性和一致性。例如：用户A 在 T1 表上增加一个索引，同时用户B 从 T2 表删除一列。这两条 DDL 语句可以并行执行。

为了提升 DDL 执行的用户体验，从 v6.2.0 起，TiDB 对原有的 DDL Owner 角色进行了升级，使得 Owner 能对 DDL 任务做相关性判断，判断逻辑如下：

- 涉及同一张表的 DDL 相互阻塞。
- DROP DATABASE 和数据库内所有对象的 DDL 互相阻塞。
- 涉及不同表的加索引和列类型变更可以并发执行。

- 逻辑 DDL 需要等待之前正在执行的逻辑 DDL 执行完才能执行。
- 其他情况下 DDL 可以根据 Concurrent DDL 并行度可用情况确定是否可以执行。

具体来说，TiDB 在 v6.2.0 中对 DDL 执行框架进行了如下升级：

- DDL Owner 能够根据以上判断逻辑并行执行 DDL 任务。
- 改善了 DDL Job 队列先入先出的问题。DDL Owner 不再选择当前队列最前面的 DDL Job，而是选择当前可以执行的 DDL Job。
- 扩充了处理物理 DDL 的 worker 数量，使得能够并行地添加多个物理 DDL。

  因为TiDB 中所有支持的 DDL 任务都是以在线变更的方式来实现的，TiDB 通过 Owner 即可对新的 DDL Job 进行相关性判断，并根据相关性结果进行 DDL 任务的调度，从而使分布式数据库实现了和传统数据库中 DDL 并发相同的效果。

并发 DDL 框架的实现进一步加强了TiDB 中 DDL 语句的执行能力，并更符合商用数据库的使用习惯。

3.9.2 最佳实践

3.9.2.1 通过系统变量来平衡物理 DDL 的执行速度与对业务负载的影响

执行物理 DDL（包括添加索引或列类型变更）时，适当调整以下系统变量可以平衡 DDL 执行速度与对业务负载的影响：

- `tidb_ddl_reorg_worker_cnt`：这个变量用来设置 DDL 操作reorg worker 的数量，控制回填的并发度。
- `tidb_ddl_reorg_batch_size`：这个变量用来设置 DDL 操作 `re-organize` 阶段的batch size，以控制回填的数据量。

  推荐值：

  - 在无其他负载情况下， 如需让 `ADD INDEX` 尽快完成， 可以将 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 的值适当调大，例如将两个变量值分别调为 20 和 2048。
  - 在有其他负载情况下，如需让`ADD INDEX` 尽量不影响其他业务，可以将`tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 适当调小，例如将两个变量值分别调为 4 和 256。

建议：

- 以上两个变量均可以在 DDL 任务执行过程中动态调整，并且在下一个事务批次中生效。
- 根据 DDL 操作的类型，并结合业务负载压力，选择合适的时间点执行，例如建议在业务负载比较低的情况运行 `ADD INDEX` 操作。
- 由于添加索引的时间跨度较长，发送相关的指令后，TiDB 会在后台执行任务，TiDB server 宕机不会影响继续执行。

3.9.2.2 并发发送 DDL 请求实现快速建大量表

一个建表的操作耗时大约 50 毫秒。受框架的限制，建表耗时可能更长。

为了更快地建表，推荐通过并发发送多个 DDL 请求以达到最快建表速度。如果串行地发送 DDL 请求，并且没有发给 Owner 节点，则建表速度会很慢。

3.9.2.3 在一条 ALTER 语句中进行多次变更

自 v6.2.0 起，TiDB 支持在一条 ALTER 语句中修改一张表的多个模式对象（如列、索引），同时保证整个语句的原子性。因此推荐在一条 ALTER 语句中进行多次变更。

3.9.2.4 检查读写性能

在添加索引时，回填数据阶段会对集群造成一定的读写压力。在 ADD INDEX 的命令发送成功后，并且在 write reorg 阶段，建议检查Grafana 面板上TiDB 和 TiKV 读写相关的性能指标，以及业务响应时间，来确定 ADD INDEX 操作对集群是否造成影响。

3.9.3 DDL 相关的命令介绍

- ADMIN SHOW DDL：用于查看 TiDB DDL 的状态，包括当前 schema 版本号、DDL Owner 的 DDL ID 和地址、正在执行的 DDL 任务和 SQL、当前 TiDB 实例的 DDL ID。详情参阅ADMIN SHOW DDL。
- ADMIN SHOW DDL JOBS：查看集群环境中的 DDL 任务运行中详细的状态。详情参阅ADMIN SHOW DDL JOBS。
- 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。详情参阅ADMIN SHOW DDL JOB QUERIES。
- ADMIN CANCEL DDL JOBS job_id [, job_id]：用于取消已经提交但未执行完成的 DDL 任务。取消完成后，执行 DDL 任务的 SQL 语句会返回 ERROR 8214 (HY000): Cancelled DDL job 错误。

  取消一个已经执行完成的 DDL 任务会在 RESULT 列看到 DDL Job:90 not found 的错误，表示该任务已从 DDL 等待队列中被移除。

3.9.4 常见问题

DDL 语句执行相关的常见问题，参考SQL FAQ - DDL 执行。

4 参考指南

4.1 系统变量

TiDB 系统变量的行为与MySQL 相似，变量的作用范围可以是会话级别有效(Session Scope) 或全局范围有效(Global Scope)。其中：

- 对 SESSION 作用域变量的更改，设置后只影响当前会话。

- 对 GLOBAL 作用域变量的更改，设置后立即生效。如果该变量也有 SESSION 作用域，已经连接的所有会话(包括当前会话)将继续使用会话当前的 SESSION 变量值。
- 要设置变量值，可使用SET 语句。

```
## 以下两个语句等价地改变一个 Session 变量
SET tidb_distsql_scan_concurrency = 10;
SET SESSION tidb_distsql_scan_concurrency = 10;

## 以下两个语句等价地改变一个 Global 变量
SET @@global.tidb_distsql_scan_concurrency = 10;
SET  GLOBAL tidb_distsql_scan_concurrency = 10;
```

注意：

部分 GLOBAL 作用域的变量会持久化到TiDB 集群中。文档中的变量有一个“是否持久化到集群”的说明，可以为“是”或者“否”。

- 对于持久化到集群的变量，当该全局变量被修改后，会通知所有TiDB 服务器刷新其系统变量缓存。在集群中增加一个新的TiDB 服务器时，或者重启现存的TiDB 服务器时，都将自动使用该持久化变量。
- 对于不持久化到集群的变量，对变量的修改只对当前连接的TiDB 实例生效。如果需要保留设置过的值，需要在 tidb.toml 配置文件中声明。

此外，由于应用和连接器通常需要读取 MySQL 变量，为了兼容这一需求，在TiDB 中，部分 MySQL 的变量既可读取也可设置。例如，尽管JDBC 连接器不依赖于查询缓存(query cache) 的行为，但仍然可以读取和设置查询缓存。

注意：

变量取较大值并不总会带来更好的性能。由于大部分变量对单个连接生效，设置变量时，还应考虑正在执行语句的并发连接数量。

确定安全值时，应考虑变量的单位：

- 如果单位为线程，安全值通常取决于 CPU 核的数量。
- 如果单位为字节，安全值通常小于系统内存的总量。
- 如果单位为时间，单位可能为秒或毫秒。

单位相同的多个变量可能会争夺同一组资源。

4.1.1 变量参考

4.1.1.1 allow_auto_random_explicit_insert 从 v4.0.3 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 是否允许在 INSERT 语句中显式指定含有 AUTO_RANDOM 属性的列的值。

4.1.1.2 authentication_ldap_sasl_auth_method_name 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：SCRAM-SHA-1
- 可选值：SCRAM-SHA-1、SCRAM-SHA-256、GSSAPI
- LDAP SASL 身份验证中，验证方法的名称。

4.1.1.3 authentication_ldap_sasl_bind_base_dn 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP SASL 身份验证中，搜索用户的范围。如果创建用户时没有通过 AS ... 指定 dn，TiDB 会自动在 LDAP Server 的该范围中根据用户名搜索用户 dn。例如 dc=example,dc=org。

4.1.1.4 authentication_ldap_sasl_bind_root_dn 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP SASL 身份验证中，TiDB 登录 LDAP Server 搜索用户时使用的 dn。

4.1.1.5 authentication_ldap_sasl_bind_root_pwd 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP SASL 身份验证中，TiDB 登录 LDAP Server 搜索用户时使用的密码。

4.1.1.6 authentication_ldap_sasl_ca_path 从 v7.1.0 版本开始引入

- 作用域：GLOBAL

- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP SASL 身份验证中，TiDB 对 StartTLS 连接使用的 CA 证书的路径。

4.1.1.7 `authentication_ldap_sasl_init_pool_size` 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：10
- 范围：`[1, 32767]`
- LDAP SASL 身份验证中，TiDB 与 LDAP Server 间连接池的初始连接数。

4.1.1.8 `authentication_ldap_sasl_max_pool_size` 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1000
- 范围：`[1, 32767]`
- LDAP SASL 身份验证中，TiDB 与 LDAP Server 间连接池的最大连接数。

4.1.1.9 `authentication_ldap_sasl_server_host` 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP SASL 身份验证中，LDAP Server 的主机名或地址。

4.1.1.10 `authentication_ldap_sasl_server_port` 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：389
- 范围：`[1, 65535]`
- LDAP SASL 身份验证中，LDAP Server 的 TCP/IP 端口号。

4.1.1.11 `authentication_ldap_sasl_tls` 从 v7.1.0 版本开始引入

- 作用域：GLOBAL

- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- LDAP SASL 身份验证中，是否使用 StartTLS 对连接加密。

4.1.1.12 authentication_ldap_simple_auth_method_name 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：SIMPLE
- 可选值：SIMPLE
- LDAP simple 身份验证中，验证方法的名称。现在仅支持 SIMPLE。

4.1.1.13 authentication_ldap_simple_bind_base_dn 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP simple 身份验证中，搜索用户的范围。如果创建用户时没有通过 AS ... 指定 dn，TiDB 会自动在 LDAP Server 的该范围中根据用户名搜索用户 dn。例如 dc=example,dc=org。

4.1.1.14 authentication_ldap_simple_bind_root_dn 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP simple 身份验证中，TiDB 登录 LDAP Server 搜索用户时使用的 dn。

4.1.1.15 authentication_ldap_simple_bind_root_pwd 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP simple 身份验证中，TiDB 登录 LDAP Server 搜索用户时使用的密码。

4.1.1.16 authentication_ldap_simple_ca_path 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是

- 类型：字符串
- 默认值：“”
- LDAP simple 身份验证中，TiDB 对 StartTLS 连接使用的 CA 证书的路径。

4.1.1.17 authentication_ldap_simple_init_pool_size 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：10
- 范围：[1，32767]
- LDAP simple 身份验证中，TiDB 与 LDAP Server 间连接池的初始连接数。

4.1.1.18 authentication_ldap_simple_max_pool_size 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1000
- 范围：[1，32767]
- LDAP simple 身份验证中，TiDB 与 LDAP Server 间连接池的最大连接数。

4.1.1.19 authentication_ldap_simple_server_host 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：“”
- LDAP simple 身份验证中，LDAP Server 的主机名或地址。

4.1.1.20 authentication_ldap_simple_server_port 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：389
- 范围：[1，65535]
- LDAP simple 身份验证中，LDAP Server 的 TCP/IP 端口号。

4.1.1.21 authentication_ldap_simple_tls 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是

- 类型：布尔型
- 默认值：OFF
- LDAP simple 身份验证中，是否使用 StartTLS 对连接加密。

4.1.1.22 auto_increment_increment

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 65535]
- 控制 AUTO_INCREMENT 自增值字段的自增步长。该变量常与 auto_increment_offset 一起使用。

4.1.1.23 auto_increment_offset

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 65535]
- 控制 AUTO_INCREMENT 自增值字段的初始值。该变量常与 auto_increment_increment 一起使用。示例如下：

```
mysql> CREATE TABLE t1 (a int not null primary key auto_increment);
Query OK, 0 rows affected (0.10 sec)

mysql> set auto_increment_offset=1;
Query OK, 0 rows affected (0.00 sec)

mysql> set auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (),(),(),();
Query OK, 4 rows affected (0.04 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t1;
+----+
| a  |
+----+
|  1 |
|  4 |
|  7 |
| 10 |
```

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

4.1.1.24 autocommit

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 用于设置在非显式事务时是否自动提交事务。更多信息，请参见事务概述。

4.1.1.25 block_encryption_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：aes-128-ecb
- 可选值：aes-128-ecb、aes-192-ecb、aes-256-ecb、aes-128-cbc、aes-192-cbc、aes-256-cbc、aes-128-ofb、aes-192-ofb、aes-256-ofb、aes-128-cfb、aes-192-cfb、aes-256-cfb
- 该变量用于设置 AES_ENCRYPT() 和 AES_DECRYPT() 函数的加密模式。

4.1.1.26 character_set_client

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 这个变量表示从客户端发出的数据所用的字符集。有关更多TiDB 支持的字符集和排序规则，参阅字符集和排序规则文档。如果需要更改字符集，建议使用SET NAMES 语句。

4.1.1.27 character_set_connection

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 若没有为字符串常量指定字符集，该变量表示这些字符串常量所使用的字符集。

4.1.1.28 character_set_database

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 该变量表示当前默认在用数据库的字符集，不建议设置该变量。选择新的默认数据库后，服务器会更改该变量的值。

4.1.1.29 character_set_results

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 该变量表示数据发送至客户端时所使用的字符集。

4.1.1.30 character_set_server

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 当 CREATE SCHEMA 中没有指定字符集时，该变量表示这些新建的表结构所使用的字符集。

4.1.1.31 collation_connection

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4_bin
- 该变量表示连接中所使用的排序规则。与 MySQL 中的 collation_connection 一致。

4.1.1.32 collation_database

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4_bin
- 该变量表示当前数据库默认所使用的排序规则。与 MySQL 中的 collation_database 一致。不建议设置此变量，当前使用的数据库变动时，此变量会被TiDB 修改。

4.1.1.33 collation_server

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4_bin
- 该变量表示创建数据库时默认的排序规则。

4.1.1.34 cte_max_recursion_depth

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1000
- 范围：[0, 4294967295]
- 这个变量用于控制公共表表达式的最大递归深度。

4.1.1.35 datadir

- 作用域：NONE
- 默认值：使用的组件和部署方式不同，默认值也不同。
  - `/tmp/tidb`：如果你将`--store`设置为 `"unistore"` 或没有设置 `--store`，则默认值为 `/tmp/tidb`。
  - `${pd-ip}:${pd-port}`：如果你设置的存储引擎是 TiKV（如果使用TiUP 和TiDB Operator 部署，则默认的存储引擎为TiKV），则默认值为 `${pd-ip}:${pd-port}`。
- 这个变量表示数据存储的位置，位置可以是本地路径 `/tmp/tidb`。如果数据存储在 TiKV 上，则可以是指向 PD 服务器的路径。变量值的格式为 `${pd-ip}:${pd-port}`，表示 TiDB 在启动时连接到的 PD 服务器。

4.1.1.36 ddl_slow_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：整数型
- 默认值：300
- 取值范围：`[0, 2147483647]`
- 单位：毫秒
- 耗时超过该阈值的 DDL 操作会被输出到日志。

4.1.1.37 default_authentication_plugin

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：`mysql_native_password`
- 可选值：`mysql_native_password`，`caching_sha2_password`，`tidb_sm3_password`，`tidb_auth_token`，`authentication_ldap_sasl` 或 `authentication_ldap_simple`。
- `tidb_auth_token` 认证方式仅用于TiDB Cloud 内部实现，不要设置为该值。
- 服务器和客户端建立连接时，这个变量用于设置服务器对外通告的默认身份验证方式。如要了解该变量的其他可选值，参见可用的身份验证插件。
- 若要在用户登录时使用 `tidb_sm3_password` 插件，需要使用TiDB-JDBC 进行连接。

4.1.1.38 default_password_lifetime 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 取值范围：`[0, 65535]`
- 该变量用于设置全局自动密码过期策略，默认值为 0，即禁用全局自动密码过期。如果设置该变量的值为正整数N，则表示允许的密码生存期为N，即必须在N 天之内更改密码。

4.1.1.39 default_week_format

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 取值范围：[0, 7]
- 设置 WEEK() 函数使用的周格式。

4.1.1.40 disconnect_on_expired_password 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 类型：布尔型
- 默认值：ON
- 该变量是一个只读变量，用来显示TiDB 是否会直接断开密码已过期用户的连接。当其值为 ON，表示 TiDB 会断开密码已过期用户的连接。当其值为 OFF，表示 TiDB 会将密码已过期用户的连接置于 “沙盒模式”，允许该用户建立连接并执行密码重置操作。
- 如果需要改变 TiDB 对密码已过期用户连接的处理方式，请在 TiDB 配置文件中的 [security] 部分修改disconnect-on-expired-password 选项。

4.1.1.41 error_count

- 作用域：SESSION
- 默认值：0
- 类型：整数型
- 表示上一条生成消息的 SQL 语句中的错误数。该变量为只读变量。

4.1.1.42 foreign_key_checks

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：在 v6.6.0 之前版本中为 OFF，在 v6.6.0 及之后的版本中为 ON。
- 表示是否开启外键约束检查。

4.1.1.43 group_concat_max_len

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1024
- 取值范围：[4, 18446744073709551615]
- 表示 GROUP_CONCAT() 函数中，项目的最大缓冲区大小。

4.1.1.44 have_openssl

- 作用域：NONE
- 类型：布尔型
- 默认值：DISABLED
- 用于 MySQL 兼容性的只读变量。当服务器启用 TLS 时，服务器将其设置为 YES。

4.1.1.45 have_ssl

- 作用域：NONE
- 类型：布尔型
- 默认值：DISABLED
- 用于 MySQL 兼容性的只读变量。当服务器启用 TLS 时，服务器将其设置为 YES。

4.1.1.46 hostname

- 作用域：NONE
- 默认值：（ 系统主机名 ）
- 这个变量一个只读变量，表示TiDB server 的主机名。

4.1.1.47 identity 从 v5.3.0 版本开始引入

- 该变量为变量last_insert_id 的别名。

4.1.1.48 init_connect

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 用户首次连接到TiDB 服务器时，init_connect 特性允许TiDB 自动执行一条或多条 SQL 语句。如果你有 CONNECTION_ADMIN 或者 SUPER 权限，这些 SQL 语句将不会被自动执行。如果这些语句执行报错，你的用户连接将被终止。

4.1.1.49 innodb_lock_wait_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：50
- 范围：[1, 3600]
- 单位：秒
- 悲观事务语句等锁时间。

4.1.1.50 interactive_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：28800
- 范围：[1, 31536000]
- 单位：秒
- 该变量表示交互式用户会话的空闲超时。交互式用户会话是指使用 CLIENT_INTERACTIVE 选项调用 mysql_real_connect() API 建立的会话（例如：MySQL shell 和 MySQL client）。该变量与 MySQL 完全兼容。

4.1.1.51 last_insert_id 从 v5.3.0 版本开始引入

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 取值范围：[0, 9223372036854775807]
- 返回由 INSERT 语句产生的最新 AUTO_INSCRENT 或者 AUTO_RANDOM 值，与 LAST_INSERT_ID() 的返回的结果相同。与 MySQL 中的 last_insert_id 一致。

4.1.1.52 last_plan_from_binding 从 v4.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量用来显示上一条执行的语句所使用的执行计划是否来自binding 的执行计划。

4.1.1.53 last_plan_from_cache 从 v4.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 这个变量用来显示上一个 execute 语句所使用的执行计划是不是直接从plan cache 中取出来的。

4.1.1.54 last_sql_use_alloc 从 v6.4.0 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 这个变量是一个只读变量，用来显示上一个语句是否使用了缓存的 Chunk 对象(Chunk allocation)。

4.1.1.55 license

- 作用域：NONE
- 默认值：Apache License 2.0
- 这个变量表示TiDB 服务器的安装许可证。

4.1.1.56 log_bin

- 作用域：NONE
- 类型：布尔型
- 默认值：OFF
- 该变量表示是否使用TiDB Binlog。

4.1.1.57 max_connections

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：整数型
- 默认值：0
- 取值范围：[0, 100000]
- 该变量表示TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认情况下，该变量值为 0 表示不限制客户端连接数。当本变量的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。

4.1.1.58 max_execution_time

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 单位：毫秒
- 语句最长执行时间。默认值(0) 表示无限制。

> 注意：
>
> - max_execution_time 目前对所有类型的语句生效，并非只对 SELECT 语句生效，与 MySQL 不同（只对SELECT 语句生效）。实际精度在 100ms 级别，而非更准确的毫秒级别。
> - 对于使用了MAX_EXECUTION_TIME Hint 的 SQL 语句，这些语句的最长执行时间将不受该变量限制，而是由该 Hint 进行限制。你也可以使用该 Hint 来创建 SQL 绑定，详情请参考SQL 操作常见问题。

4.1.1.59 max_prepared_stmt_count

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：-1
- 范围：[-1, 1048576]
- 指定当前实例中PREPARE 语句的最大数量。
- 值为 -1 时表示不对实例中的 PREPARE 语句数量进行限制。
- 如果将变量值设为超过上限 1048576，则使用上限值 1048576：

```
mysql> SET GLOBAL max_prepared_stmt_count = 1048577;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------------------------+
| Level   | Code | Message                                                         |
+---------+------+-----------------------------------------------------------------+
| Warning | 1292 | Truncated incorrect max_prepared_stmt_count value: '1048577' |
+---------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SHOW GLOBAL VARIABLES LIKE 'max_prepared_stmt_count';
+-------------------------+---------------+
| Variable_name           | Value   |
+-------------------------+---------------+
| max_prepared_stmt_count | 1048576 |
+-------------------------+---------------+
1 row in set (0.00 sec)
```

4.1.1.60 max_allowed_packet 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：67108864
- 取值范围：[1024, 1073741824]
- 该变量取值应为 1024 的整数倍。若取值无法被 1024 整除，则会提示 warning 并向下取整。例如设置为 1025 时，则TiDB 中的实际取值为 1024。
- 服务器端和客户端在一次传送数据包的过程中所允许最大的数据包大小，单位为字节。
- 该变量的行为与 MySQL 兼容。

4.1.1.61 mpp_exchange_compression_mode 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：UNSPECIFIED
- 可选值：NONE，FAST，HIGH_COMPRESSION，UNSPECIFIED

- 该变量用于选择 MPP Exchange 算子的数据压缩模式，当TiDB 选择版本号为 1 的 MPP 执行计划时生效。该变量值的含义如下：
  - UNSPECIFIED：表示未指定，TiDB 将自动选择压缩模式，当前TiDB 自动选择 FAST 模式
  - NONE：不使用数据压缩
  - FAST：快速模式，整体性能较好，压缩比小于 HIGH_COMPRESSION
  - HIGH_COMPRESSION：高压缩比模式

4.1.1.62 mpp_version 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：UNSPECIFIED
- 可选值：UNSPECIFIED，0，1
- 该变量用于指定不同版本的 MPP 执行计划。指定后，TiDB 会选择指定版本的 MPP 执行计划。该变量值含义如下：
  - UNSPECIFIED：表示未指定，此时 TiDB 自动选择最新版本 1。
  - 0：兼容所有TiDB 集群版本，MPP 版本大于 0 的新特性均不会生效。
  - 1：从 v6.6.0 版本开始引入，用于开启TiFlash 带压缩的数据交换，详情参见MPP Version 和 Exchange 数据压缩。

4.1.1.63 password_history 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 4294967295]
- 该变量用于建立密码重用策略，使TiDB 基于密码更改次数限制密码的重复使用。该变量默认值为 0，表示禁用基于密码更改次数的密码重用策略。当设置该变量为一个正整数N 时，表示不允许重复使用最近 N 次使用过的密码。

4.1.1.64 password_reuse_interval 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 4294967295]
- 该变量用于建立密码重用策略，使TiDB 基于经过时间限制密码重复使用。该变量默认值为 0，表示禁用基于密码经过时间的密码重用策略。当设置该变量为一个正整数N 时，表示不允许重复使用最近N 天内使用过的密码。

4.1.1.65 plugin_dir

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 默认值：“”
- 指定加载插件的目录。

4.1.1.66 plugin_load

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 默认值：“”
- 指定TiDB 启动时加载的插件，多个插件之间用逗号（,）分隔。

4.1.1.67 port

- 作用域：NONE
- 类型：整数型
- 默认值：4000
- 范围：[0, 65535]
- 使用 MySQL 协议时tidb-server 监听的端口。

4.1.1.68 rand_seed1

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 该变量用于为 SQL 函数 RAND() 中使用的随机值生成器添加种子。
- 该变量的行为与 MySQL 兼容。

4.1.1.69 rand_seed2

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 该变量用于为 SQL 函数 RAND() 中使用的随机值生成器添加种子。
- 该变量的行为与 MySQL 兼容。

4.1.1.70 require_secure_transport 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否所有TiDB 的连接都在本地 socket 上进行通信，或使用 TLS。详情见为 TiDB 客户端服务端间通信开启加密传输。
- 该变量设置为 ON 时，必须使用开启 TLS 的会话连接到TiDB，防止在 TLS 配置不正确时出现锁定的情况。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(security.require-secure-transport) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.71 skip_name_resolve 从 v5.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量控制 tidb-server 实例是否将主机名作为连接握手的一部分来解析。
- 当 DNS 不可靠时，可以启用该变量来提高网络性能。

> 注意：
>
> 当 skip_name_resolve 设置为 ON 时，身份信息中包含主机名的用户将无法登录服务器。例如：
>
> ```
> CREATE USER 'appuser'@'apphost' IDENTIFIED BY 'app-password';
> ```
>
> 该示例中，建议将 apphost 替换为IP 地址或通配符（%）。

4.1.1.72 socket

- 作用域：NONE
- 默认值：“”
- 使用 MySQL 协议时，tidb-server 所监听的本地unix 套接字文件。

4.1.1.73 sql_log_bin

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 表示是否将更改写入TiDB Binlog。

> 注意：
>
> 不建议将 sql_log_bin 设置为全局变量，因为TiDB 的未来版本可能只允许将其设置为会话变量。

4.1.1.74 sql_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO ↩,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
- 这个变量控制许多 MySQL 兼容行为。详情见SQL 模式。

4.1.1.75 sql_require_primary_key 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制表是否必须有主键。启用该变量后，如果在没有主键的情况下创建或修改表，将返回错误。
- 该功能基于 MySQL 8.0 的特性 sql_require_primary_key。
- 强烈推荐在使用TiCDC 时启用改变量，因为同步数据变更至 MySQL sink 时要求表必须有主键。

4.1.1.76 sql_select_limit 从 v4.0.2 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：18446744073709551615
- 范围：[0, 18446744073709551615]
- 单位：行
- SELECT 语句返回的最大行数。

4.1.1.77 ssl_ca

- 作用域：NONE
- 默认值：“”
- 证书颁发机构(CA) 文件的位置。若文件不存在，则变量值为空。该变量的值由TiDB 配置项ssl-ca 定义。

4.1.1.78 ssl_cert

- 作用域：NONE
- 默认值：“”
- 用于 SSL/TLS 连接的证书文件的位置。若文件不存在，则变量值为空。该变量的值由 TiDB 配置项ssl-cert定义。

4.1.1.79 ssl_key

- 作用域：NONE
- 默认值：“”
- 用于 SSL/TLS 连接的私钥文件的位置。若文件不存在，则变量值为空。该变量的值由TiDB 配置项ssl-key定义。

4.1.1.80 system_time_zone

- 作用域：NONE
- 默认值：（ 随系统 ）
- 该变量显示首次引导启动TiDB 时的系统时区。另请参阅time_zone。

4.1.1.81 tidb_adaptive_closest_read_threshold 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：4096
- 取值范围：[0, 9223372036854775807]
- 单位：字节
- 这个变量用于控制当replica-read 设置为 closest-adaptive 时，优先将读请求发送至TiDB server 所在区域副本的阈值。当读请求预估的返回结果的大小超过此阈值时，TiDB 会将读请求优先发送至同一可用区的副本，否则会发送至leader 副本。

4.1.1.82 tidb_allow_batch_cop 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2]
- 这个变量用于控制TiDB 向 TiFlash 发送coprocessor 请求的方式，有以下几种取值：
  - 0：从不批量发送请求
  - 1：aggregation 和 join 的请求会进行批量发送
  - 2：所有的cop 请求都会批量发送

4.1.1.83 tidb_allow_fallback_to_tikv 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 这个变量表示将TiKV 作为备用存储引擎的存储引擎列表。当该列表中的存储引擎发生故障导致 SQL 语句执行失败时，TiDB 会使用TiKV 作为存储引擎再次执行该 SQL 语句。目前支持设置该变量为“” 或者“tiflash”。如果设置该变量为“tiflash”，当TiFlash 返回超时错误（对应的错误码为ErrTiFlashServerTimeout）时，TiDB 会使用TiKV 作为存储引擎再次执行该 SQL 语句。

4.1.1.84 tidb_allow_function_for_expression_index 从 v5.2.0 版本开始引入

- 作用域：NONE
- 默认值：json_array、json_array_append、json_array_insert、json_contains、json_contains_path、json_depth、json_extract、json_insert、json_keys、json_length、json_merge_patch、json_merge_preserve <⟶ 、json_object、json_pretty、json_quote、json_remove、json_replace、json_search、json_set、json_storage_size、json_type、json_unquote、json_valid、lower、md5、reverse、tidb_shard、upper、vitess_hash
- 这个变量用于显示创建表达式索引所允许使用的函数。

4.1.1.85 tidb_allow_mpp 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否使用TiFlash 的 MPP 模式执行查询，可以设置的值包括：
  - 0 或 OFF，代表从不使用 MPP 模式
  - 1 或 ON，代表由优化器根据代价估算选择是否使用 MPP 模式（默认）

MPP 是TiFlash 引擎提供的分布式计算框架，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法。MPP 模式选择的详细说明参见控制是否选择 MPP 模式。

4.1.1.86 tidb_allow_remove_auto_inc 从 v2.1.18 和 v3.0.4 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 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(OFF) 为不允许。

4.1.1.87 tidb_analyze_partition_concurrency

> **警告：**
>
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 这个变量用于TiDB analyze 分区表时，对分区表统计信息进行读写的并发度。

4.1.1.88 tidb_analyze_version 从 v5.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：2
- 范围：[1, 2]
- 这个变量用于控制TiDB 收集统计信息的行为。
- 在 v5.3.0 及之后的版本中，该变量的默认值为 2，具体可参照统计信息简介文档。如果从 v5.3.0 之前版本的集群升级至 v5.3.0 及之后的版本，tidb_analyze_version 的默认值不发生变化。

4.1.1.89 tidb_auto_analyze_end_time

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：时间
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动ANALYZE 更新统计信息的结束时间。例如，只允许在 UTC 时间的凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

  ```
  - tidb_auto_analyze_start_time='01:00 +0000'
  - tidb_auto_analyze_end_time='03:00 +0000'
  ```

4.1.1.90 tidb_auto_analyze_partition_batch_size 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 1024]

- 用于设置TiDB 自动analyze 分区表（即自动收集分区表上的统计信息）时，每次同时analyze 分区的个数。
- 若该变量值小于分区表的分区数，则TiDB 会分多批自动analyze 该分区表的所有分区。若该变量值大于等于分区表的分区数，则TiDB 会同时analyze 该分区表的所有分区。
- 若分区表个数远大于该变量值，且自动analyze 花费时间较长，可调大该参数的值以减少耗时。

4.1.1.91 tidb_auto_analyze_ratio

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0.5
- 范围：[0, 18446744073709551615]
- 这个变量用来设置 TiDB 在后台自动执行ANALYZE TABLE 更新统计信息的阈值。0.5 指的是当表中超过 50% 的行被修改时， 触发自动 ANALYZE 更新。可以指定 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 来限制自动ANALYZE 的时间

> 注意：
> 当系统变量 tidb_enable_auto_analyze 设置为 ON 时，TiDB 才会触发 auto_analyze。

4.1.1.92 tidb_auto_analyze_start_time

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：时间
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动ANALYZE 更新统计信息的开始时间。例如，只允许在 UTC 时间的凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

```
- tidb_auto_analyze_start_time='01:00 +0000'
- tidb_auto_analyze_end_time='03:00 +0000'
```

4.1.1.93 tidb_auto_build_stats_concurrency 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 256]
- 这个变量用来设置执行统计信息自动更新的并发度。

4.1.1.94 tidb_backoff_lock_fast

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：10
- 范围：[1, 2147483647]
- 这个变量用来设置读请求遇到锁的backoff时间。

4.1.1.95 tidb_backoff_weight

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：2
- 范围：[0, 2147483647]
- 这个变量用来给TiDB的 backoff 最大时间增加权重，即内部遇到网络或其他组件(TiKV, PD)故障时，发送重试请求的最大重试时间。可以通过这个变量来调整最大重试时间，最小值为 1。

  例如，TiDB 向 PD 取 TSO 的基础超时时间是 15 秒，当 tidb_backoff_weight = 2 时，取TSO 的最大超时时间为：基础时间* 2 等于 30 秒。

  在网络环境较差的情况下，适当增大该变量值可以有效缓解因为超时而向应用端报错的情况；而如果应用端希望更快地接到报错信息，则应该尽量减小该变量的值。

4.1.1.96 tidb_batch_commit

> 警告：
>
> 不建议开启此变量。

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用已废弃的batch-commit特性。当该变量开启时，事务可能会通过分组一些语句被拆分为多个事务，并被非原子地提交。不推荐使用这种方式。

4.1.1.97 tidb_batch_delete

> **警告：**
>
> 该变量与废弃的batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用已废弃的batch-dml 特性中的batch-delete 特性。当该变量开启时，DELETE 语句可能会被拆分为多个事务，并被非原子地提交。要使该特性生效，还需要开启 tidb_enable_batch_dml 并将 tidb_dml_batch_size 的值设置为正数。不推荐使用这种方式。

4.1.1.98 tidb_batch_insert

> **警告：**
>
> 该变量与废弃的batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用已废弃的batch-dml 特性中的batch-insert 特性。当该变量开启时，INSERT 语句可能会被拆分为多个事务，并被非原子地提交。要使该特性生效，还需要开启 tidb_enable_batch_dml 并将 tidb_dml_batch_size 的值设置为正数。不推荐使用这种方式。

4.1.1.99 tidb_batch_pending_tiflash_count 从 v6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4000
- 范围：[0, 4294967295]
- 使用 ALTER DATABASE SET TIFLASH REPLICA 语句为TiFlash 添加副本时，能容许的不可用表的个数上限。如果超过该上限，则会停止或者以非常慢的速度为库中的剩余表设置TiFlash 副本。

4.1.1.100 tidb_broadcast_join_threshold_count 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL

- 是否持久化到集群：是
- 类型：整数型
- 默认值：10240
- 范围：[0，9223372036854775807]
- 单位：行
- 如果join 的对象为子查询，优化器无法估计子查询结果集大小，在这种情况下通过结果集行数判断。如果子查询的行数估计值小于该变量，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。
- 开启tidb_prefer_broadcast_join_by_exchange_data_size 功能后，该变量将不再生效。

4.1.1.101 tidb_broadcast_join_threshold_size 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：104857600 (100 MiB)
- 范围：[0，9223372036854775807]
- 单位：字节
- 如果表大小（字节数）小于该值，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。
- 开启tidb_prefer_broadcast_join_by_exchange_data_size 功能后，该变量将不再生效。

4.1.1.102 tidb_build_stats_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 单位：线程
- 默认值：4
- 取值范围：[1，256]
- 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。

4.1.1.103 tidb_capture_plan_baselines 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启自动捕获绑定功能。该功能依赖 Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。
- 开启该功能后会定期遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动创建绑定。

4.1.1.104 tidb_cdc_write_source 从 v6.5.0 版本开始引入

- 作用域：SESSION
- 是否持久化到集群：否
- 类型：整数型
- 默认值：0
- 范围：[0, 15]
- 当变量非 0 时，该 SESSION 写入的数据将被视为是由TiCDC 写入的。这个变量仅由TiCDC 设置，任何时候都不应该手动调整该变量。

4.1.1.105 tidb_check_mb4_value_in_utf8

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：布尔型
- 默认值：ON
- 设置该变量为 ON 可强制只存储基本多文种平面(BMP) 编码区段内的 utf8 字符值。若要存储 BMP 区段外的 utf8 值，推荐使用 utf8mb4 字符集。
- 早期版本的TiDB 中 (v2.1.x)，utf8 检查更为宽松。如果你的TiDB 集群是从早期版本升级的，推荐关闭该变量，详情参阅升级与升级后常见问题。

4.1.1.106 tidb_committer_concurrency 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：128
- 范围：[1, 10000]
- 在单个事务的提交阶段，用于执行提交操作相关请求的 goroutine 数量。
- 若提交的事务过大，事务提交时的流控队列等待耗时可能会过长。此时，可以通过调大该配置项来加速提交。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(performance.committer-concurrency) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.107 tidb_checksum_table_concurrency

- 作用域：SESSION
- 类型：整数型
- 默认值：4
- 取值范围：[1, 256]
- 单位：线程
- 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。

4.1.1.108 tidb_config

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 默认值：“”
- 这个变量是一个只读变量，用来获取当前TiDB Server 的配置信息。

4.1.1.109 tidb_constraint_check_in_place

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量仅适用于乐观事务模型。悲观事务模式中的行为由tidb_constraint_check_in_place_pessimistic <↔> 控制。
- 当这个变量设置为 OFF 时，唯一索引的重复值检查会被推迟到事务提交时才进行。这有助于提高性能，但对于某些应用，可能导致非预期的行为。详情见约束。
  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 OFF：

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

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

```
tidb> commit; -- 事务提交时才检查
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 ON：

```
set @@tidb_constraint_check_in_place=ON;
begin optimistic;
insert into t values (1);
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

4.1.1.110 tidb_constraint_check_in_place_pessimistic 从 v6.3.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：当配置项pessimistic-txn.constraint-check-in-place-pessimistic 为默认值 true 时，该变量的默认值为 ON。当配置项pessimistic-txn.constraint-check-in-place-pessimistic 为 false 时，该变量的默认值为 OFF。
- 该变量仅适用于悲观事务模型。乐观事务模式中的行为由tidb_constraint_check_in_place 控制。
- 当这个变量设置为 OFF 时，唯一约束检查会被推迟到下一次需要对这个索引加锁的语句执行时，或事务提交时才进行。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。
- 关闭该变量可能会导致悲观事务中返回 LazyUniquenessCheckFailure 报错。返回该错误时，TiDB 将会回滚当前事务。
- 关闭该变量后，悲观事务中不支持使用SAVEPOINT 功能。
- 关闭该变量时，commit 语句可能会报出 Write conflict 错误或 Duplicate entry 错误，两种错误都意味着事务回滚。
  - 悲观事务模型下将 tidb_constraint_check_in_place_pessimistic 设置为 OFF：

```
set @@tidb_constraint_check_in_place_pessimistic=OFF;
create table t (i int key);
insert into t values (1);
begin pessimistic;
insert into t values (1);
```

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

```
tidb> commit; -- 事务提交时才检查
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

  - 悲观事务模型下将 tidb_constraint_check_in_place_pessimistic 设置为 ON：

```
set @@tidb_constraint_check_in_place_pessimistic=ON;
begin pessimistic;
insert into t values (1);
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

4.1.1.111 tidb_cost_model_version 从 v6.2.0 版本开始引入

注意：

- 自 v6.5.0 开始，新创建的TiDB 集群默认使用 Cost Model Version 2。如果从 v6.4.0 及之前版本的集群升级到 v6.5.0 及之后的版本，`tidb_cost_model_version` 的值不发生变化。
- 切换代价模型版本可能会引起查询计划的变动。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：2
- 取值范围：`[1, 2]`
- 可选值：
  - 1：使用 Cost Model Version 1 代价模型。TiDB v6.4.0 及之前的版本默认使用 Cost Model Version 1。
  - 2：使用 Cost Model Version 2 代价模型。TiDB v6.5.0 正式发布了代价模型Cost Model Version 2，在内部测试中比Version 1 版本的代价模型更加准确。
- 代价模型会影响优化器对计划的选择，具体可见代价模型。

4.1.1.112 `tidb_current_ts`

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 取值范围：`[0, 9223372036854775807]`
- 这个变量是一个只读变量，用来获取当前事务的时间戳。

4.1.1.113 `tidb_ddl_disk_quota` 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：107374182400 (100 GiB)
- 范围：`[107374182400, 1125899906842624]` ([100 GiB, 1 PiB])
- 单位：字节
- 这个变量仅在`tidb_ddl_enable_fast_reorg` 开启的情况下生效，用于设置创建索引的回填过程中本地存储空间的使用限制。

4.1.1.114 `tidb_ddl_enable_fast_reorg` 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型

- 默认值：ON
- 这个变量用来控制是否开启添加索引加速功能，来提升创建索引回填过程的速度。开启该变量对于数据量较大的表有一定的性能提升。
- TiDB v7.1.0 引入了快速加索引功能的检查点机制，即使TiDB owner 因故障重启或者切换，也能够通过自动定期保存的检查点恢复部分进度。
- 要验证已经完成的 ADD INDEX 操作是否使用了添加索引加速功能，可以执行ADMIN SHOW DDL JOBS 语句查看 JOB_TYPE 一列中是否含有 ingest 字样。

> **警告：**
>
> 目前，PITR 恢复会额外处理日志备份时间段内通过索引加速功能创建的索引，以达到兼容效果。详细内容请参考索引加速功能为什么与 PITR 功能不兼容。

> **注意：**
>
> 在升级到 v6.5.0 及以上版本时，建议你检查TiDB 的temp-dir 路径是否正确挂载了 SSD 磁盘。该参数是TiDB 的配置参数，设置后需要重启TiDB 才能生效。因此，在升级前提前进行设置，可以避免再次重启。

4.1.1.115 tidb_enable_dist_task 从 v7.1.0 版本开始引入

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启TiDB 后端任务分布式框架。开启分布式框架后，DDL 和 Import 等后端任务将会由集群中多个TiDB 节点共同完成。
- 在 TiDB v7.1.0 中，只支持分布式执行分区表的 ADD INDEX。
- 该变量由 tidb_ddl_distribute_reorg 改名而来。

4.1.1.116 tidb_ddl_error_count_limit

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：512
- 范围：[0, 9223372036854775807]
- 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。

4.1.1.117 tidb_ddl_flashback_concurrency 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：64
- 范围：[1, 256]
- 这个变量用来控制FLASHBACK CLUSTER TO TIMESTAMP 的并发数。

4.1.1.118 tidb_ddl_reorg_batch_size

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：256
- 范围：[32, 10240]
- 单位：行
- 这个变量用来设置 DDL 操作 re-organize 阶段的batch size。比如 ADD INDEX 操作，需要回填索引数据，通过并发 tidb_ddl_reorg_worker_cnt 个 worker 一起回填数据，每个 worker 以 batch 为单位进行回填。
  - 如果 ADD INDEX 操作时有较多 UPDATE 操作或者 REPLACE 等更新操作，batch size 越大，事务冲突的概率也会越大，此时建议调小batch size 的值，最小值是 32。
  - 在没有事务冲突的情况下，batch size 可设为较大值（需要参考worker 数量，见线上负载与ADD INDEX 相互影响测试），这样回填数据的速度更快，但是 TiKV 的写入压力也会变大。

4.1.1.119 tidb_ddl_reorg_priority

- 作用域：SESSION
- 类型：枚举型
- 默认值：PRIORITY_LOW
- 可选值：PRIORITY_LOW、PRIORITY_NORMAL、PRIORITY_HIGH
- 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。

4.1.1.120 tidb_ddl_reorg_worker_cnt

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 DDL 操作 re-organize 阶段的并发度。

4.1.1.121 tidb_default_string_match_selectivity 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点型
- 默认值：0.8
- 范围：[0, 1]
- 这个变量用来设置过滤条件中的 like、rlike、regexp 函数在行数估算时的默认选择率，以及是否对这些函数启用TopN 辅助估算。
- TiDB 总是会尝试利用统计信息对过滤条件中的 like 进行估算，但是当 like 匹配的字符串太复杂时，或者面对 rlike 或 regexp 时，往往无法充分利用统计信息，转而使用 0.8 作为选择率，造成行数估算的误差较大。
- 该变量可以用于修改这个行为，当变量被设为 0 以外的值时，会使用变量的值而不是默认的 0.8 作为选择率。
- 如果将该变量的值设为 0，TiDB 在对上述三个函数进行行数估算时，会尝试利用统计信息中的TopN 进行求值来提高估算精度，同时也会考虑统计信息中的 NULL 数。求值操作预计会造成少量性能损耗。这个功能生效的前提是统计信息是在tidb_analyze_version 设为 2 时收集的。
- 当该变量的值被设为默认值以外的值的时候，会对 not like、not rlike、not regexp 的行数估算也进行相应的调整。

4.1.1.122 tidb_disable_txn_auto_retry

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来设置是否禁用显式的乐观事务自动重试，设置为 ON 时，不会自动重试，如果遇到事务冲突需要在应用层重试。

  如果将该变量的值设为 OFF，TiDB 将会自动重试事务，这样在事务提交时遇到的错误更少。需要注意的是，这样可能会导致数据更新丢失。

  这个变量不会影响自动提交的隐式事务和TiDB 内部执行的事务，它们依旧会根据 tidb_retry_limit 的值来决定最大重试次数。

关于是否需要禁用自动重试，请参考重试的局限性。

该变量只适用于乐观事务，不适用于悲观事务。悲观事务的重试次数由max_retry_count 控制。

4.1.1.123 tidb_distsql_scan_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：15
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 scan 操作的并发度。
- AP 类应用适合较大的值，TP 类应用适合较小的值。对于AP 类应用，最大值建议不要超过所有TiKV 节点的 CPU 核数。
- 若表的分区较多可以适当调小该参数（取决于扫描数据量的大小以及扫描频率），避免 TiKV 内存溢出 (OOM)。

4.1.1.124 tidb_dml_batch_size

> 警告：
>
> 该变量与废弃的batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用 batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 单位：行
- 这个变量的值大于 0 时，TiDB 会将 INSERT 语句在更小的事务中批量提交。这样可减少内存使用，确保大批量修改时事务大小不会达到 txn-total-size-limit 限制。
- 只有变量值为 0 时才符合ACID 要求。否则无法保证TiDB 的原子性和隔离性要求。
- 要使该特性生效， 还需要开启 tidb_enable_batch_dml， 以及至少开启 tidb_batch_insert 和 tidb_batch_delete 中的一个。

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

4.1.1.125 tidb_enable_1pc 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 指定是否在只涉及一个Region 的事务上启用一阶段提交特性。比起传统两阶段提交，一阶段提交能大幅降低事务提交延迟并提升吞吐。

> 注意：
>
> - 对于新创建的集群，默认值为 ON。对于升级版本的集群，如果升级前是 v5.0 以下版本，升级后默认值为 OFF。
> - 启用TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用TiCDC 替代 TiDB Binlog。
> - 启用该参数仅意味着一阶段提交成为可选的事务提交模式，实际由TiDB 自行判断选择最合适的提交模式进行事务提交。

4.1.1.126 tidb_enable_analyze_snapshot 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量控制 ANALYZE 读取历史时刻的数据还是读取最新的数据。当该变量设置为 ON 时，ANALYZE 读取 ANALYZE 开始时刻的历史数据。当该变量设置为 OFF 时，ANALYZE 读取最新的数据。
- 在 v5.2 之前，ANALYZE 读取最新的数据。v5.2 至 v6.1 版本 ANALYZE 读取 ANALYZE 开始时刻的历史数据。

> 警告：
>
> 如果 ANALYZE 读取 ANALYZE 开始时刻的历史数据，长时间的 AUTO ANALYZE 可能会因为历史数据被GC 而出现 `GC life time is shorter than transaction duration` 的报错。

4.1.1.127 tidb_enable_async_commit 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON

- 该变量控制是否启用Async Commit 特性，使事务两阶段提交的第二阶段于后台异步进行。开启本特性能降低事务提交的延迟。

> 注意：
>
> - 对于新创建的集群，默认值为 ON。对于升级版本的集群，如果升级前是 v5.0 以下版本，升级后默认值为 OFF。
> - 启用TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用TiCDC 替代 TiDB Binlog。
> - 启用该参数仅意味着Async Commit 成为可选的事务提交模式，实际由TiDB 自行判断选择最合适的提交模式进行事务提交。

4.1.1.128 tidb_enable_auto_analyze 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量控制TiDB 是否以后台操作自动更新表的统计信息。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(`performance.run-auto-analyze`)进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.129 tidb_enable_auto_increment_in_generated

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否允许在创建生成列或者表达式索引时引用自增列。

4.1.1.130 tidb_enable_batch_dml

> 警告：
>
> 该变量与废弃的batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用 batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：GLOBAL
- 是否持久化到集群：是

- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用废弃的batch-dml 特性。启用该变量后，部分语句可能会被拆分为多个事务执行，这是非原子性的，使用时需谨慎。使用 batch-dml 时，必须确保正在操作的数据没有并发操作。要使该变量生效，还需要为 tidb_batch_dml_size 指定一个正值，并启用 tidb_batch_insert 和 tidb_batch_delete 中的至少一个。

4.1.1.131 tidb_enable_cascades_planner

> 警告：
>
> 目前cascades planner 为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启cascades planner。

4.1.1.132 tidb_enable_chunk_rpc 从 v4.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：ON
- 这个变量用来设置是否启用 Coprocessor 的 Chunk 数据编码格式。

4.1.1.133 tidb_enable_clustered_index 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：ON
- 可选值：OFF，ON，INT_ONLY
- 这个变量用于控制默认情况下表的主键是否使用聚簇索引。“默认情况”即不显式指定 CLUSTERED <-> /NONCLUSTERED 关键字的情况。可设置为 OFF/ON/INT_ONLY。
  - OFF 表示所有主键默认使用非聚簇索引。
  - ON 表示所有主键默认使用聚簇索引。
  - INT_ONLY 此时的行为受配置项 alter-primary-key 控制。如果该配置项取值为 true，则所有主键默认使用非聚簇索引；如果该配置项取值为 false，则由单个整数类型的列构成的主键默认使用聚簇索引，其他类型的主键默认使用非聚簇索引。

4.1.1.134 tidb_enable_collect_execution_info

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否同时将各个执行算子的执行信息记录入 slow query log 中。

4.1.1.135 tidb_enable_column_tracking 从 v5.4.0 版本开始引入

> 警告：
>
> 收集 PREDICATE COLUMNS 的统计信息目前为实验特性，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启TiDB 对 PREDICATE COLUMNS 的收集。关闭该变量后，之前收集的 PREDICATE <-> COLUMNS 会被清除。详情见收集部分列的统计信息。

4.1.1.136 tidb_enable_ddl 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 默认值：ON
- 可选值：OFF，ON
- 用于设置该TiDB 实例是否可以成为 DDL owner。若当前 TiDB 集群中只有一台TiDB 实例，则不能禁止该实例成为 DDL owner，即不能设置为 OFF。

4.1.1.137 tidb_enable_enhanced_security

- 作用域：NONE
- 类型：布尔型
- 默认值：OFF
- 这个变量表示所连接的TiDB 服务器是否启用了安全增强模式(SEM)。若要改变该变量值，你需要在TiDB 服务器的配置文件中修改 enable-sem 项的值，并重启TiDB 服务器。
- 安全增强模式受安全增强式 Linux 等系统设计的启发，削减拥有 MySQL SUPER 权限的用户能力，转而使用细粒度的 RESTRICTED 权限作为替代。这些细粒度的 RESTRICTED 权限如下：
  - RESTRICTED_TABLES_ADMIN：能够写入 mysql 库中的系统表，能查看 information_schema 表上的敏感列。

- RESTRICTED_STATUS_ADMIN：能够在 SHOW STATUS 命令中查看敏感内容。
- RESTRICTED_VARIABLES_ADMIN：能够在 SHOW [GLOBAL] VARIABLES 和 SET 命令中查看和设置包含敏感内容的变量。
- RESTRICTED_USER_ADMIN：能够阻止其他用户更改或删除用户帐户。
- RESTRICTED_CONNECTION_ADMIN：能够阻止其它用户使用 KILL 语句终止连接。

4.1.1.138 tidb_enable_external_ts_read 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 当此变量设置为 ON 时，TiDB 会读取tidb_external_ts 指定时间戳前的历史数据。

4.1.1.139 tidb_external_ts 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 当tidb_enable_external_ts_read 设置为 ON 时，TiDB 会依据该变量指定的时间戳读取历史数据。

4.1.1.140 tidb_restricted_read_only 从 v5.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 可选值：OFF 和 ON
- tidb_restricted_read_only和tidb_super_read_only 的作用相似。在大多数情况下， 你只需要使用tidb_super_read_only 即可。
- 拥有 SUPER 或 SYSTEM_VARIABLES_ADMIN 权限的用户可以修改该变量。如果 TiDB 开启了安全增强模式，你还需要额外的 RESTRICTED_VARIABLES_ADMIN 权限才能读取或修改该变量。
- tidb_restricted_read_only 的设置将影响tidb_super_read_only 的值：
  - 当设置 tidb_restricted_read_only 为 ON 时，tidb_super_read_only 的将自动被设置为 ON。
  - 当设置 tidb_restricted_read_only 为 OFF 时，tidb_super_read_only 的值将不受影响。
  - 当 tidb_restricted_read_only 为 ON 时，tidb_super_read_only 的值无法设置为 OFF。

- 对于 TiDB 的 DBaaS 供应商，当 TiDB 为另一个数据库的下游数据库时，如果要将整个 TiDB 集群设置为只读模式，你需要开启安全增强模式 并将 tidb_restricted_read_only 设置为 ON，从而防止你的用户通过tidb_super_read_only 将 TiDB 集群设置为可写。实现方法：首先开启安全增强模式，然后由你（作为 DBaaS 的控制面）使用一个 admin 用户控制 tidb_restricted_read_only（需要拥有 SYSTEM_VARIABLES_ADMIN 和 RESTRICTED_VARIABLES_ADMIN 权限），由你的数据库用户使用 root 用户控制tidb_super_read_only（需要拥有 SUPER 权限）。
- 该变量可以控制整个集群的只读状态。开启后（即该值为 ON），整个集群中的 TiDB 服务器都将进入只读状态，只有 SELECT、USE、SHOW 等不会修改数据的语句才能被执行，其他如 INSERT、UPDATE 等语句会被拒绝执行。
- 该变量开启只读模式只保证整个集群最终进入只读模式，当变量修改状态还没被同步到其他TiDB 服务器时，尚未同步的TiDB 仍然停留在非只读模式。
- 在执行 SQL 语句之前，TiDB 会检查集群的只读标志。从 v6.2.0 起，在提交 SQL 语句之前，TiDB 也会检查该标志，从而防止在服务器被置于只读模式后某些长期运行的auto commit 语句可能修改数据的情况。
- 在变量开启时，对于尚未提交的事务：
  - 如果有尚未提交的只读事务，可正常提交该事务。
  - 如果尚未提交的事务为非只读事务，在事务内执行写入的 SQL 语句会被拒绝。
  - 如果尚未提交的事务已经有数据改动，其提交也会被拒绝。
- 当集群开启只读模式后，所有用户（包括 SUPER 用户）都无法执行可能写入数据的 SQL 语句，除非该用户被显式地授予了 RESTRICTED_REPLICA_WRITER_ADMIN 权限。

4.1.1.141 tidb_enable_exchange_partition

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量用于设置是否启用exchange partitions with tables 特性。默认值为 ON，即默认开启该功能。
- 该变量自 v6.3.0 开始废弃，其取值将固定为默认值 ON，即默认开启 exchange partitions with tables。

4.1.1.142 tidb_enable_extended_stats

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量指定TiDB 是否收集扩展统计信息来指导优化器。

4.1.1.143 tidb_enable_fast_analyze

> **警告：**
>
> 目前快速分析功能为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 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。

4.1.1.144 tidb_enable_foreign_key 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：在 v6.6.0 之前版本中为 OFF，在 v6.6.0 及之后的版本中为 ON。
- 这个变量用于控制是否开启 FOREIGN KEY 特性。

4.1.1.145 tidb_enable_gc_aware_memory_track

> **警告：**
>
> 该变量为TiDB 内部调试变量，可能会在未来版本中删除，请勿设置该变量。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启可感知到垃圾回收的内存追踪(GC-Aware memory track)。

4.1.1.146 tidb_enable_non_prepared_plan_cache

> **警告：**
>
> 非 Prepare 语句执行计划缓存目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否开启非 Prepare 语句执行计划缓存。

4.1.1.147 tidb_enable_non_prepared_plan_cache_for_dml 从 v7.1.0 版本开始引入

> 警告：
>
> 非 Prepare 语句执行计划缓存目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制非 Prepare 语句执行计划缓存是否支持 DML 语句。

4.1.1.148 tidb_enable_gogc_tuner 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量来用控制是否开启GOGC Tuner。

4.1.1.149 tidb_enable_historical_stats

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否开启历史统计信息。默认值为 ON 表示默认开启历史统计信息。

4.1.1.150 tidb_enable_historical_stats_for_capture

> 警告：
>
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制 PLAN REPLAYER CAPTURE 抓取的内容是否默认带历史统计信息。默认值为 OFF，表示默认不带历史统计信息。

4.1.1.151 tidb_enable_index_merge 从 v4.0 版本开始引入

> 注意：
>
> - 当集群从 v4.0.0 以下版本升级到 v5.4.0 及以上版本时，该变量开关默认关闭，防止升级后计划发生变化导致回退。
> - 当集群从 v4.0.0 及以上版本升级到 v5.4.0 及以上版本时，该变量开关保持升级前的状态。
> - 对于 v5.4.0 及以上版本的新建集群，该变量开关默认开启。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否开启index merge 功能。

4.1.1.152 tidb_enable_index_merge_join

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 表示是否启用 IndexMergeJoin 算子。
- 该变量为TiDB 内部变量，不推荐使用，否则可能会造成数据正确性问题。

4.1.1.153 tidb_enable_legacy_instance_scope 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于允许使用 SET SESSION 对 INSTANCE 作用域的变量进行设置，用法同 SET GLOBAL。
- 为了兼容之前的TiDB 版本，该变量值默认为 ON。

4.1.1.154 tidb_enable_list_partition 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来设置是否开启 LIST (COLUMNS)TABLE PARTITION 特性。

4.1.1.155 tidb_enable_local_txn

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于一个未发布的特性，请勿修改该变量值。

4.1.1.156 tidb_enable_metadata_lock 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来设置是否开启元数据锁特性。需要注意，在设置该变量时，集群中不能有 DDL 任务，以免造成非预期数据正确性、一致性问题。

4.1.1.157 tidb_enable_mutation_checker 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于设置是否开启mutation checker。 mutation checker 是一项在 DML 语句执行过程中进行的数据索引一致性校验，校验报错会回滚当前语句。开启该校验会导致 CPU 使用轻微上升。详见数据索引一致性报错。
- 对于新创建的 v6.0.0 及以上的集群，默认值为 ON。对于升级版本的集群，如果升级前是低于 v6.0.0 的版本，升级后默认值为 OFF。

4.1.1.158 tidb_enable_new_cost_interface 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON

- TiDB v6.2.0 对代价模型的实现进行了代码层面的重构，这个变量用来控制是否使用重构后的代价模型Cost Model Version 2。
- 重构后的代价模型使用完全一样的代价公式，因此不会引起计划选择的变动，此开关默认打开。
- 从 v6.1 升级至 v6.2 的用户，此开关保持升级前的 OFF 状态，此时建议直接打开；对于从 v6.1 之前版本升级至 v6.2 的用户，此开关默认为 ON。

4.1.1.159 tidb_enable_new_only_full_group_by_check 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于控制TiDB 执行 ONLY_FULL_GOUP_BY 检查时的行为。有关 ONLY_FULL_GROUP_BY 的信息可以参考 MySQL 文档。在 v6.1 中 TiDB 对该项检查做了更严格正确的处理。
- 由于可能存在版本升级造成的兼容性问题，在 v6.1 中该变量默认值是 OFF，即默认关闭。

4.1.1.160 tidb_enable_noop_functions 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：OFF
- 可选值：ON、OFF、WARN
- 默认情况下，用户尝试将某些语法用于尚未实现的功能时，TiDB 会报错。若将该变量值设为 ON，TiDB 则自动忽略此类功能不可用的情况，即不会报错。若用户无法更改 SQL 代码，可考虑将变量值设为 ON。
- 启用 noop 函数可以控制以下行为：
  - LOCK IN SHARE MODE 语法
  - SQL_CALC_FOUND_ROWS 语法
  - START TRANSACTION READ ONLY 和 SET TRANSACTION READ ONLY 语法
  - tx_read_only、transaction_read_only、offline_mode、super_read_only、read_only 以及 sql_auto_is_null 系统变量
  - GROUP BY <expr> ASC|DESC 语法

> **警告：**
>
> 该变量只有在默认值 OFF 时，才算是安全的。因为设置 tidb_enable_noop_functions=1 后，TiDB 会自动忽略某些语法而不报错，这可能会导致应用程序出现异常行为。例如，允许使用语法 START TRANSACTION READ ONLY 时，事务仍会处于读写模式。

4.1.1.161 tidb_enable_noop_variables 从 v6.2.0 版本开始引入

- 作用域：GLOBAL

- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 若该变量值为 OFF，TiDB 具有以下行为：
  - 使用 SET 设置 noop 的系统变量时会报 "setting *variable_name* has no effect in TiDB" 的警告。
  - SHOW [SESSION | GLOBAL] VARIABLES 的结果不显示 noop 的系统变量。
  - 使用 SELECT 读取 noop 的系统变量时会报 "variable *variable_name* has no effect in TiDB" 的警告。
- 你可以通过 SELECT * FROM INFORMATION_SCHEMA.CLIENT_ERRORS_SUMMARY_GLOBAL; 语句来检查TiDB 实例是否曾设置和读取 noop 系统变量。

4.1.1.162 tidb_enable_null_aware_anti_join 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：在 v7.0.0 之前版本中为 OFF，即默认关闭。在 v7.0.0 及之后的版本中为 ON，即默认开启。
- 这个变量用于控制TiDB 对特殊集合算子 NOT IN 和 != ALL 引导的子查询产生的ANTI JOIN 是否采用 Null Aware Hash Join 的执行方式。
- 从旧版本升级到 v7.0.0 及之后版本，该功能自动开启，即该变量的值修改为默认值 ON。

4.1.1.163 tidb_enable_outer_join_reorder 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 自 v6.1.0 起，TiDB 的Join Reorder 算法开始支持 Outer Join。该变量用于控制是否启用 Outer Join 的Join Reorder。
- 对于从较低版本升级到当前版本的TiDB：
  - 如果升级前TiDB 的版本低于 v6.1.0，升级后该变量的默认值为 ON。
  - 如果升级前TiDB 的版本等于或大于 v6.1.0，升级后该变量的默认值跟随升级前的设定值。

4.1.1.164 tidb_enable_inl_join_inner_multi_pattern 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于控制当内表上有 Selection/Projection 算子时是否支持Index Join。OFF 表示不支持。

4.1.1.165 tidb_enable_ordered_result_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 指定是否对最终的输出结果进行自动排序。
- 例如，开启该变量后，TiDB 会将 `SELECT a, MAX(b)FROM t GROUP BY a` 处理为 `SELECT a, MAX(b)FROM t GROUP BY a ORDER BY a, MAX(b)`。

4.1.1.166 tidb_enable_paging 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否使用分页(paging) 方式发送 Coprocessor 请求。对于 [v5.4.0, v6.2.0) 区间的TiDB 版本，该变量只对 IndexLookup 算子生效；对于 v6.2.0 以及之后的版本，该变量对全局生效。从 v6.4.0 版本开始，该变量默认值由 OFF 改成 ON。
- 适用场景：
  - 推荐在所有偏 OLTP 的场景下使用paging。
  - 对于使用 IndexLookUp 和 Limit 并且 Limit 无法下推到 IndexScan 上的读请求，可能会出现读请求的延迟高、TiKV 的 Unified read pool CPU 使用率高的情况。在这种情况下，由于 Limit 算子只需要少部分数据，开启tidb_enable_paging 能够减少处理数据的数量，从而降低延迟、减少资源消耗。
  - 对于 Dumpling 数据导出或者全表扫描这类的场景，开启paging 后可以有效降低TiDB 进程的内存消耗。

> 注意：
>
> 对于偏OLAP 的场景，并且以TiKV 而非TiFlash 作为存储引擎时，开启paging 可能导致部分场景下性能回退。此时，你可以考虑通过该变量关闭 paging 或者通过系统变量tidb_min_paging_size 和tidb_max_paging_size 调整paging size 的行数范围。

4.1.1.167 tidb_enable_parallel_apply 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF

- 这个变量用于控制是否开启Apply 算子并发，并发数由 `tidb_executor_concurrency` 变量控制。Apply 算子用来处理关联子查询且默认无并发，所以执行速度较慢。打开Apply 并发开关可增加并发度，提高执行速度。目前默认关闭。

4.1.1.168 `tidb_enable_pipelined_window_function`

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量指定是否对窗口函数采用流水线的执行算法。

4.1.1.169 `tidb_enable_plan_cache_for_param_limit` 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制 Prepared Plan Cache 是否缓存 `LIMIT` 后面带变量(`LIMIT ?`) 的执行计划。目前不支持缓存 `LIMIT` 后面带变量且变量值大于 10000 的执行计划。

4.1.1.170 `tidb_enable_plan_cache_for_subquery` 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制 Prepared Plan Cache 是否缓存包含子查询的查询。

4.1.1.171 `tidb_enable_plan_replayer_capture`

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否开启`PLAN REPLAYER CAPTURE` 功能。默认值 ON 代表开启 `PLAN REPLAYER CAPTURE` 功能。

4.1.1.172 `tidb_enable_plan_replayer_continuous_capture` 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否开启`PLAN REPLAYER CONTINUOUS CAPTURE` 功能。默认值 OFF 代表关闭功能。

4.1.1.173 tidb_enable_prepared_plan_cache 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否开启 Prepared Plan Cache。开启后，对 Prepare、 Execute 请求的执行计划会进行缓存，以便在后续执行时跳过查询计划优化这个步骤，获得性能上的提升。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(prepared-plan-cache.enabled) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.174 tidb_enable_prepared_plan_cache_memory_monitor 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否统计 Prepared Plan Cache 中所缓存的执行计划占用的内存。具体可见Prepared Plan Cache 的内存管理。

4.1.1.175 tidb_enable_pseudo_for_outdated_stats 从 v5.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制优化器在一张表上的统计信息过期时的行为。
- 统计信息过期的判断标准：最近一次对某张表执行 ANALYZE 获得统计信息后，该表数据被修改的行数大于该表总行数的 80%，便可判定该表的统计信息已过期。该比例可通过pseudo-estimate-ratio 配置参数调整。
- 默认情况下（即该变量值为 OFF 时），某张表上的统计信息过期后，优化器仍会使用该表上的统计信息。将该变量值设为 ON 时，当统计信息过期后，优化器认为该表上除总行数以外的统计信息不再可靠，转而使用pseudo 统计信息。
- 如果表数据修改较频繁，没有及时对表执行 ANALYZE，但又希望执行计划保持稳定，推荐将该变量值设为 OFF。

4.1.1.176 tidb_enable_rate_limit_action

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量控制是否为读数据的算子开启动态内存控制功能。读数据的算子默认启用tidb_distsql_scan_concurrency <⟶ 所允许的最大线程数来读取数据。当单条 SQL 语句的内存使用每超过tidb_mem_quota_query 一次，读数据的算子会停止一个线程。

- 当读数据的算子只剩 1 个线程且当单条 SQL 语句的内存使用继续超过tidb_mem_quota_query 时，该 SQL 语句会触发其它的内存控制行为，例如落盘。
- 该变量在单条查询仅涉及读数据的情况下，对内存控制效果较好。若还存在额外的计算操作（如连接、聚合等），打开该变量可能会导致内存不受 tidb_mem_quota_query 控制，加剧 OOM 风险。

4.1.1.177 tidb_enable_resource_control 从 v6.6.0 版本开始引入

> **警告：**
>
> 资源管控目前为实验性特性，此变量定义可能在之后发生变化或者删除。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：布尔型
- 该变量是资源管控特性的开关。该变量设置为 ON 时，集群支持应用按照资源组做资源隔离。

4.1.1.178 tidb_enable_reuse_chunk 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 可选值：OFF，ON
- 该变量用于控制TiDB 是否启用 Chunk 对象缓存。如果为 ON，则优先使用缓存中的 Chunk 对象，缓存中找不到申请的对象时才会从系统内存中申请。如果为 OFF，则直接从系统内存中申请 Chunk 对象。

4.1.1.179 tidb_enable_slow_log

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否开启 slow log 功能。

4.1.1.180 tidb_enable_stmt_summary 从 v3.0.4 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否开启 statement summary 功能。如果开启，SQL 的耗时等执行信息将被记录到系统表 information_schema.STATEMENTS_SUMMARY 中，用于定位和排查 SQL 性能问题。

4.1.1.181 tidb_enable_strict_double_type_check 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否可以用 DOUBLE 类型的无效定义创建表。该设置的目的是提供一个从TiDB 早期版本升级的方法，因为早期版本在验证类型方面不太严格。
- 该变量的默认值 ON 与 MySQL 兼容。

例如，由于无法保证浮点类型的精度，现在将DOUBLE(10) 类型视为无效。将tidb_enable_strict_double_type_check <⟶> 更改为 OFF 后，将会创建表。如下所示：

```
CREATE TABLE t1 (id int, c double(10));
ERROR 1149 (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
SET tidb_enable_strict_double_type_check = 'OFF';
Query OK, 0 rows affected (0.00 sec)
CREATE TABLE t1 (id int, c double(10));
Query OK, 0 rows affected (0.09 sec)
```

> 注意：
>
> 该设置仅适用于 DOUBLE 类型，因为 MySQL 允许为 FLOAT 类型指定精度。从 MySQL 8.0.17 开始已弃用此行为，不建议为 FLOAT 或 DOUBLE 类型指定精度。

4.1.1.182 tidb_enable_table_partition

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：枚举型
- 可选值：OFF，ON，AUTO
- 这个变量用来设置是否开启 TABLE PARTITION 特性。目前变量支持以下三种值：
  - 默认值 ON 表示开启TiDB 当前已实现了的分区表类型，目前Range partition、 Hash partition 以及 Range column 单列的场景会生效。
  - AUTO 目前作用和 ON 一样。
  - OFF 表示关闭 TABLE PARTITION 特性，此时语法还是保持兼容，只是创建的表并不是真正的分区表，而是普通的表。

4.1.1.183 tidb_enable_telemetry 从 v4.0.2 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于动态地控制TiDB 遥测功能是否开启，当前版本默认关闭TiDB 的遥测功能。当所有TiDB 实例都设置配置项enable-telemetry 为 false 时，将忽略该系统变量，并总是关闭TiDB 遥测功能。参阅遥测了解该功能详情。

4.1.1.184 tidb_enable_tiflash_read_for_write_stmt 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制包含增删改的 SQL 语句中的读取操作能否下推到TiFlash，比如：
  - INSERT INTO SELECT 语句中的 SELECT 查询（典型应用场景为TiFlash 查询结果物化）
  - UPDATE 和 DELETE 语句中的 WHERE 条件过滤
- 从 v7.1.0 开始，该变量废弃。当tidb_allow_mpp = ON 时，优化器将根据SQL 模式及 TiFlash 副本的代价估算自行决定是否将查询下推至TiFlash。需要注意的是，只有当前会话的SQL 模式为非严格模式（即 sql_mode 值不包含 STRICT_TRANS_TABLES 和 STRICT_ALL_TABLES）时，TiDB 才允许将包含增删改的 SQL 语句（如 INSERT INTO SELECT）中的读取操作下推至 TiFlash。

4.1.1.185 tidb_enable_tmp_storage_on_oom

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 设置是否在单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时为某些算子启用临时磁盘。
- 在 v6.3.0 之前这个开关可通过TiDB 配置文件中的 oom-use-tmp-storage 项进行配置。在升级到 v6.3.0 及更新的版本后，集群会自动使用原 oom-use-tmp-storage 的值来初始化该开关，配置文件中 oom-use-tmp-<⟷ storage 的新设置不再影响该开关。

4.1.1.186 tidb_enable_top_sql 从 v5.4.0 版本开始引入

> **警告：**
>
> Top SQL 目前是实验性功能，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启Top SQL 特性。

4.1.1.187 tidb_enable_tso_follower_proxy 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来开启TSO Follower Proxy 特性。当该值为 OFF 时，TiDB 仅会从 PD leader 获取TSO。开启该特性之后，TiDB 在获取TSO 时会将请求均匀地发送到所有 PD 节点上，通过 PD follower 转发 TSO 请求，从而降低 PD leader 的 CPU 压力。
- 适合开启TSO Follower Proxy 的场景：
  - PD leader 因高压力的TSO 请求而达到 CPU 瓶颈，导致TSO RPC 请求的延迟较高。
  - 集群中的TiDB 实例数量较多，且调高tidb_tso_client_batch_max_wait_time 并不能缓解TSO RPC 请求延迟高的问题。

> 注意：
>
> 如果 PD leader 的 TSO RPC 延迟升高，但其现象并非由 CPU 使用率达到瓶颈而导致（可能存在网络等问题），此时，打开 TSO Follower Proxy 可能会导致TiDB 的语句执行延迟上升，从而影响集群的 QPS 表现。

4.1.1.188 tidb_enable_unsafe_substitute 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否对生成列中表达式替换使用不安全的替换方式。默认值为 OFF，即默认关闭不安全的替换方式。详情见生成列。

4.1.1.189 tidb_enable_vectorized_expression 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否开启向量化执行。

4.1.1.190 tidb_enable_window_function

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 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 OFF。

4.1.1.191 tidb_enable_row_level_checksum 从 v7.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启TiCDC 单行数据正确性校验功能。

4.1.1.192 tidb_enforce_mpp 从 v5.1 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF（表示关闭）。如需修改此变量的默认值，请配置performance.enforce-mpp 参数。
- 这个变量用于控制是否忽略优化器代价估算，强制使用TiFlash 的 MPP 模式执行查询，可以设置的值包括：
  - 0 或 OFF，代表不强制使用 MPP 模式（默认）
  - 1 或 ON，代表将忽略代价估算，强制使用 MPP 模式。注意：只有当 tidb_allow_mpp=true 时该设置才生效。

MPP 是TiFlash 引擎提供的分布式计算框架，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法。MPP 模式选择的详细说明参见控制是否选择 MPP 模式。

4.1.1.193 tidb_evolve_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否启用自动演进绑定功能。该功能的详细介绍和使用方法可以参考自动演进绑定。
- 为了减少自动演进对集群的影响，可以进行以下配置：
  - 设置 tidb_evolve_plan_task_max_time，限制每个执行计划运行的最长时间，其默认值为 600s；
  - 设置tidb_evolve_plan_task_start_time 和 tidb_evolve_plan_task_end_time，限制运行演进任务的时间窗口，默认值分别为 00:00 +0000 和 23:59 +0000。

4.1.1.194 tidb_evolve_plan_task_end_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：时间
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动演进的结束时间。

4.1.1.195 tidb_evolve_plan_task_max_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：600
- 范围：[-1, 9223372036854775807]
- 单位：秒
- 该变量用于限制自动演进功能中，每个执行计划运行的最长时间。

4.1.1.196 tidb_evolve_plan_task_start_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：时间
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动演进的开始时间。

4.1.1.197 tidb_executor_concurrency 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：5
- 范围：[1, 256]
- 单位：线程

变量用来统一设置各个 SQL 算子的并发度，包括：

- index lookup
- index lookup join
- hash join
- hash aggregation（partial 和 final 阶段）
- window
- projection

tidb_executor_concurrency 整合了已有的系统变量，方便管理。这些变量所列如下：

- tidb_index_lookup_concurrency
- tidb_index_lookup_join_concurrency
- tidb_hash_join_concurrency
- tidb_hashagg_partial_concurrency
- tidb_hashagg_final_concurrency
- tidb_projection_concurrency
- tidb_window_concurrency

v5.0 后， 用户仍可以单独修改以上系统变量（会有废弃警告）， 且修改只影响单个算子。后续通过 tidb_executor_concurrency 的修改也不会影响该算子。若要通过 tidb_executor_concurrency 来管理所有算子的并发度，需要将以上所列变量的值设置为 -1。

对于从 v5.0 之前的版本升级到 v5.0 的系统， 如果用户对上述所列变量的值没有做过改动（即 tidb_hash_join_concurrency 值为 5， 其他值为 4 ）， 则会自动转为使用 tidb_executor_concurrency 来统一管理算子并发度。如果用户对上述变量的值做过改动，则沿用之前的变量对相应的算子做并发控制。

4.1.1.198 tidb_expensive_query_time_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：整数型
- 默认值：60
- 范围：[10, 2147483647]
- 单位：秒
- 这个变量用来控制打印 expensive query 日志的阈值时间，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。

4.1.1.199 tidb_force_priority

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：枚举型
- 默认值：NO_PRIORITY
- 可选值：NO_PRIORITY、LOW_PRIORITY、DELAYED、HIGH_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。
- 默认值 NO_PRIORITY 表示不强制改变执行语句的优先级。

4.1.1.200 tidb_gc_concurrency 从 v5.0 版本开始引入

- 作用域：GLOBAL

- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用于指定GC 在Resolve Locks（清理锁）步骤中线程的数量。默认值 -1 表示由TiDB 自主判断运行 GC 要使用的线程的数量。

4.1.1.201 tidb_gc_enable 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否启用TiKV 的垃圾回收(GC) 机制。如果不启用GC 机制，系统将不再清理旧版本的数据，因此会有损系统性能。

4.1.1.202 tidb_gc_life_time 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：Duration
- 默认值：10m0s
- 范围：[10m0s, 8760h0m0s]
- 这个变量用于指定每次进行垃圾回收(GC) 时保留数据的时限。变量值为Go 的 Duration 字符串格式。每次进行GC 时，将以当前时间减去该变量的值作为 safe point。

Note:

- 在数据频繁更新的场景下，将 tidb_gc_life_time 的值设置得过大（如数天甚至数月）可能会导致一些潜在的问题，如：
  - 占用更多的存储空间。
  - 大量的历史数据可能会在一定程度上影响系统性能， 尤其是范围的查询（如 select count(*)from t）。
- 如果一个事务的运行时长超过了 tidb_gc_life_time 配置的值，在GC 时，为了使这个事务可以继续正常运行，系统会保留从这个事务开始时间 start_ts 以来的数据。例如，如果 tidb_gc_life_time 的值配置为 10 分钟，且在一次GC 时，集群正在运行的事务中最早开始的那个事务已经运行了 15 分钟，那么本次GC 将保留最近 15 分钟的数据。

4.1.1.203 tidb_gc_max_wait_time 从 v6.1.0 版本开始引入

- 作用域：GLOBAL

- 是否持久化到集群：是
- 类型：整数型
- 默认值：86400
- 范围：[600, 31536000]
- 单位：秒
- 这个变量用于指定活跃事务阻碍GC safe point 推进的最大时间。每次进行GC 时，默认GC safe point 不会超过正在执行中的事务的开始时间。如果活跃事务运行时间未超过该值，GC safe point 会一直被阻塞不更新，直到活跃事务运行时间超过该值 safe point 才会正常推进。

4.1.1.204 tidb_gc_run_interval 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：Duration
- 默认值：10m0s
- 范围：[10m0s, 8760h0m0s]
- 这个变量用于指定垃圾回收 (GC) 运行的时间间隔。变量值为 Go 的 Duration 字符串格式，如"1h30m"、"15m"等。

4.1.1.205 tidb_gc_scan_lock_mode 从 v5.0 版本开始引入

> 警告：
>
> Green GC 目前是实验性功能，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：LEGACY
- 可设置为：PHYSICAL，LEGACY
  - LEGACY：使用旧的扫描方式，即禁用 Green GC。
  - PHYSICAL：使用物理扫描方式，即启用 Green GC。
- 这个变量用于指定垃圾回收(GC) 的Resolve Locks（清理锁）步骤中扫描锁的方式。当变量值设置为LEGACY 时，TiDB 以 Region 为单位进行扫描。当变量值设置为 PHYSICAL 时，每个TiKV 节点分别绕过Raft 层直接扫描数据，可以有效地缓解在启用Hibernate Region 功能时，GC 唤醒全部Region 的影响，从而提升Resolve Locks（清理锁）这个步骤的执行速度。

4.1.1.206 tidb_general_log

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例

- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否在日志里记录所有的 SQL 语句。该功能默认关闭。如果系统运维人员在定位问题过程中需要追踪所有 SQL 记录，可考虑开启该功能。
- 在 TiDB 配置项log.level 为 "info" 或 "debug" 时，通过查询 "GENERAL_LOG" 字符串可以定位到该功能在日志中的所有记录。日志会记录以下内容：
  - conn：当前会话对应的 ID
  - user：当前会话用户
  - schemaVersion：当前 schema 版本
  - txnStartTS：当前事务的开始时间戳
  - forUpdateTS：事务模式为悲观事务时，SQL 语句的当前时间戳。悲观事务内发生写冲突时，会重试当前执行语句，该时间戳会被更新。重试次数由max-retry-count 配置。事务模式为乐观事务时，该条目与 txnStartTS 等价。
  - isReadConsistency：当前事务隔离级别是否是读已提交 (RC)
  - current_db：当前数据库名
  - txn_mode：事务模式。可选值：OPTIMISTIC (乐观事务模式)，或 PESSIMISTIC (悲观事务模式)
  - sql：当前查询对应的 SQL 语句

4.1.1.207 tidb_non_prepared_plan_cache_size

> 警告：
>
> 从 v7.1.0 开始，该变量被废弃。请使用tidb_session_plan_cache_size 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：100
- 范围：[1, 100000]
- 这个变量用来控制非 Prepare 语句执行计划缓存最多能够缓存的计划数量。

4.1.1.208 tidb_generate_binary_plan 从 v6.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于指定是否在 slow log 和 statement summary 里包含以二进制格式编码的执行计划。
- 开启该变量后，即可在TiDB Dashboard 中查看查询的图形化执行计划。注意，TiDB Dashboard 只显示变量开启时产生的查询的执行计划。
- 用 select tidb_decode_binary_plan('xxx...') SQL 语句可以从编码后的执行计划解析出具体的执行计划。

4.1.1.209 tidb_gogc_tuner_threshold 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.6
- 范围：[0, 0.9)
- 这个变量用来控制GOGC Tuner 自动调节的最大内存阈值，超过阈值后GOGC Tuner 会停止工作。

4.1.1.210 tidb_guarantee_linearizability 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 此变量控制异步提交(Async Commit) 中提交时间戳的计算方式。默认情况下（使用 ON 值），两阶段提交从 PD 服务器请求一个新的时间戳，并使用该时间戳计算最终提交的时间戳，这样可保证所有并发事务可线性化。
- 如果将该变量值设为 OFF，从 PD 获取时间戳的操作会被省掉，这种情况下只保证因果一致性但不保证线性一致性。详情请参考 PingCAP 博文Async Commit 原理介绍。
- 对于需要只保证因果一致性的场景，可将此变量设为 OFF 以提升性能。

4.1.1.211 tidb_hash_exchange_with_new_collation

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该值表示是否在开启new collation 的集群里生成 MPP hash partition exchange 算子。true 表示生成此算子，false表示不生成。
- 该变量为TiDB 内部变量，不推荐设置该变量。

4.1.1.212 tidb_hash_join_concurrency

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[1, 256]

- 单位：线程
- 这个变量用来设置hash join 算法的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.213 tidb_hashagg_final_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 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。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.214 tidb_hashagg_partial_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 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。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.215 tidb_historical_stats_duration 从 v6.6.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是

- 类型：Duration
- 默认值：168h，即 7 天
- 这个变量用来控制历史统计信息在存储中的保留时间。

4.1.1.216 tidb_ignore_prepared_cache_close_stmt 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否忽略关闭 Prepared Statement 的指令。
- 如果变量值设为 ON，Binary 协议的 COM_STMT_CLOSE 信号和文本协议的DEALLOCATE PREPARE 语句都会被忽略。

4.1.1.217 tidb_index_join_batch_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：25000
- 范围：[1, 2147483647]
- 单位：行
- 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。

4.1.1.218 tidb_index_join_double_read_penalty_cost_rate 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0
- 范围：[0, 18446744073709551615]
- 这个变量用来设置是否给选择index join 增加一些惩罚性的代价，以降低优化器选择index join 操作的倾向，从而增加选择其他join 方式的倾向，例如如选择hash join 和 tiflash join 等。
- 优化器选择index join 可能触发较多的回表请求，造成较多的资源开销，此时可以通过设置这个变量，来减少优化器选择index join 的倾向。
- 这个变量只有在tidb_cost_model_version 设置为 2 时生效。

4.1.1.219 tidb_index_lookup_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1，256]
- 单位：线程
- 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。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.220 tidb_index_lookup_join_concurrency

> 警告：
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1，256]
- 单位：线程
- 这个变量用来设置index lookup join 算法的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.221 tidb_index_merge_intersection_concurrency 从 v6.5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1，256]
- 这个变量用来设置索引合并进行交集操作时的最大并发度，仅在以动态裁剪模式访问分区表时有效。实际并发度为 tidb_index_merge_intersection_concurrency 与分区表分区数目两者中较小的值。
- 默认值 -1 表示使用tidb_executor_concurrency 的值。

4.1.1.222 tidb_index_lookup_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：20000
- 范围：[1，2147483647]
- 单位：行
- 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。

4.1.1.223 tidb_index_serial_scan_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 256]
- 单位：线程
- 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。

4.1.1.224 tidb_init_chunk_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：32
- 范围：[1, 32]
- 单位：行
- 这个变量用来设置执行过程中初始chunk的行数。默认值是 32，可设置的范围是 1 ~ 32。

4.1.1.225 tidb_isolation_read_engines 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：tikv,tiflash,tidb
- 这个变量用于设置TiDB 在读取数据时可以使用的存储引擎列表。

4.1.1.226 tidb_last_ddl_info 从 v6.0.0 版本开始引入

- 作用域：SESSION
- 默认值：“”
- 类型：字符串
- 该变量为只读变量，TiDB 内部使用该变量获取当前会话中上一个 DDL 操作的信息。
  - “query”：上一个 DDL 查询字符串。
  - “seq_num”：每个 DDL 操作的序列号，用于标识 DDL 操作的顺序。

4.1.1.227 tidb_last_query_info 从 v4.0.14 版本开始引入

- 作用域：SESSION
- 默认值：“”
- 这是一个只读变量。用于在TiDB 内部查询上一条 DML 语句的事务信息。查询的事务信息包括：
  - txn_scope：事务的作用域，可能为 global 或 local。
  - start_ts：事务开始的时间戳。
  - for_update_ts：先前执行的 DML 语句的 for_update_ts 信息。这是TiDB 用于测试的内部术语。通常，你可以忽略此信息。
  - error：错误消息（如果有）。

4.1.1.228 tidb_last_txn_info 从 v4.0.9 版本开始引入

- 作用域：SESSION
- 类型：字符串
- 此变量用于获取当前会话中最后一个事务的信息。这是一个只读变量。事务信息包括：
  - 事务的范围
  - 开始时间戳和提交时间戳
  - 事务的提交模式，可能是两阶段提交，一阶段提交，或者异步提交
  - 事务从异步提交或一阶段提交到两阶段提交的回退信息
  - 遇到的错误

4.1.1.229 tidb_last_plan_replayer_token 从 v6.3.0 版本开始引入

- 作用域：SESSION
- 类型：字符串
- 这个变量是一个只读变量，用于获取当前会话中最后一个 PLAN REPLAYER dump 的结果。

4.1.1.230 tidb_load_based_replica_read_threshold 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值："1s"
- 范围：[0s, 1h]
- 类型：字符串
- 这个变量用来设置基于负载的replica read 的触发阈值。当leader 节点的预估排队时间超过阈值时，TiDB 会优先从 follower 节点读取数据。格式为时间，例如 "100ms" 或 "1s"。详情见TiDB 热点问题处理。

4.1.1.231 tidb_log_file_max_days 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 这个变量可以调整当前TiDB 实例上日志的最大保留天数。默认值是实例配置文件中指定的值，见配置项max-days。此变量只影响当前 TiDB 实例上的配置，重启后丢失，且配置文件不受影响。

4.1.1.232 tidb_low_resolution_tso

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否启用低精度TSO 特性。开启该功能之后，新事务会使用一个每 2s 更新一次的TS 来读取数据。
- 主要场景是在可以容忍读到旧数据的情况下，降低小的只读事务获取TSO 的开销。

4.1.1.233 tidb_nontransactional_ignore_error 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否在非事务语句中立刻返回错误。当设为 OFF 时，在碰到第一个报错的batch 时，非事务 DML 语句即中止，取消其后的所有batch，返回错误。当设为 ON 时，当某个batch 执行报错时，其后的batch 会继续执行，直到所有batch 执行完毕，返回结果时把这些错误合并后返回。

4.1.1.234 tidb_max_auto_analyze_time 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：43200
- 范围：[0, 2147483647]
- 单位：秒
- 这个变量用于指定自动ANALYZE 的最大执行时间。当执行时间超出指定的时间时，自动ANALYZE 会被终止。当该变量值为 0 时，自动ANALYZE 没有最大执行时间的限制。

4.1.1.235 tidb_max_bytes_before_tiflash_external_group_by 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1, 9223372036854775807]
- 这个变量用于指定TiFlash 中带有 GROUP BY 的 Hash Aggregation 算子的最大内存使用量，单位为byte，超过该值之后TiFlash 会触发 Hash Aggregation 算子的落盘。当该变量值为 -1 时，TiDB 不传递该变量给TiFlash。只有该变量值大于等于 0 时，TiDB 才会传递该变量给TiFlash。该变量为 0 时表示内存使用无限制，即TiFlash Hash Aggregation 算子不会触发落盘。详情见TiFlash 数据落盘。

> 注意：
>
> - 假设一个TiDB 集群有多个TiFlash 节点，Aggregation 通常会在多个TiFlash 节点上分布式执行。该变量控制的是单个TiFlash 节点中Aggregation 算子的最大内存使用量。
> - 当该变量设置为 -1 时，TiFlash 将根据自身配置项max_bytes_before_external_group_by 的值来决定Aggregation 算子的最大内存使用量。

4.1.1.236 tidb_max_bytes_before_tiflash_external_join 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1，9223372036854775807]
- 这个变量用于指定TiFlash 中带等值关联条件的 Hash Join 算子的最大内存使用量，单位为byte，超过该值之后TiFlash 会触发 Hash Join 算子的落盘。当该变量值为-1 时，TiDB 不传递该变量给TiFlash。只有该变量值大于等于 0 时，TiDB 才会传递该变量给TiFlash。该变量为 0 时表示内存使用无限制，即TiFlash Hash Join 算子不会触发落盘。详情见TiFlash 数据落盘。

> 注意：
>
> - 假设一个TiDB 集群有多个TiFlash 节点，Join 通常会在多个TiFlash 节点上分布式执行。该变量控制的是单个TiFlash 节点中Join 算子的最大内存使用量。
> - 当该变量设置为-1 时，TiFlash 将根据自身配置项max_bytes_before_external_join 的值来决定Join 算子的最大内存使用量。

4.1.1.237 tidb_max_bytes_before_tiflash_external_sort 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1，9223372036854775807]
- 这个变量用于指定TiFlash 中带topN 和 sort 算子的最大内存使用量，单位为byte，超过该值之后TiFlash 会触发topN 和 sort 算子的落盘。当该变量值为-1 时，TiDB 不传递该变量给TiFlash。只有该变量值大于等于 0 时，TiDB 才会传递该变量给TiFlash。该变量为 0 时表示内存使用无限制，即TiFlash topN 和 sort 算子不会触发落盘。详情见TiFlash 数据落盘。

> 注意：
>
> - 假设一个TiDB 集群有多个TiFlash 节点，TopN 和 Sort 通常会在多个TiFlash 节点中分布式执行。该变量控制的是单个TiFlash 节点中TopN 和 Sort 算子的最大内存使用量。
> - 当该变量设置为-1 时，TiFlash 将根据自身配置项max_bytes_before_external_sort 的值来决定TopN 和 Sort 算子的最大内存使用量。

4.1.1.238 tidb_max_chunk_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1024
- 范围：[32，2147483647]
- 单位：行
- 这个变量用来设置执行过程中一个chunk 最大的行数，设置过大可能引起缓存局部性的问题。

4.1.1.239 tidb_max_delta_schema_count

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1024
- 范围：[100，16384]
- 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。

4.1.1.240 tidb_max_paging_size 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：50000
- 范围：[1，9223372036854775807]
- 单位：行
- 这个变量用来设置coprocessor 协议中paging size 的最大的行数。请合理设置该值，设置过小，TiDB 与TiKV 的 RPC 交互会更频繁；设置过大，导数据和全表扫等特定场景会占用更多内存。该变量的默认值对于 OLTP 场景较友好，如果业务只使用了TiKV 作为存储引擎，当执行偏 OLAP 的负载时，可以考虑将变量值调大，有可能获得更好的性能。

4.1.1.241 tidb_max_tiflash_threads 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1，256]
- 单位：线程
- TiFlash 中request 执行的最大并发度。默认值为 -1，表示该系统变量无效。0 表示由TiFlash 系统自动设置该值。

4.1.1.242 tidb_mem_oom_action 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：CANCEL
- 可选值：CANCEL，LOG
- 该变量控制当单个查询使用的内存超过限制(tidb_mem_quota_query)且不能再利用临时磁盘时，TiDB 所采取的操作。详情见TiDB 内存控制。
- 该变量默认值为 CANCEL，但在 TiDB v4.0.2 及之前的版本中，默认值为 LOG。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(oom-action)进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.243 tidb_mem_quota_analyze 从 v6.1.0 版本开始引入

> **警告：**
>
> 目前限制ANALYZE 的内存使用量为实验特性，在生产环境中使用时可能存在内存统计有误差的情况。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 单位：字节
- 取值范围：[-1, 9223372036854775807]
- 这个变量用来控制TiDB 更新统计信息时的最大总内存占用，包括用户执行的ANALYZE TABLE 和 TiDB 后台自动执行的统计信息更新任务。当总的内存占用超过这个阈值时，用户执行的 ANALYZE 会被终止退出，并通过错误信息提示用户尝试更小的采样率或稍后重试。如果TiDB 后台自动执行的统计信息更新任务因内存超限而退出，且使用的采样率高于默认值，则会使用默认采样率重试一次。当该变量值为负数或零时，TiDB 不对更新统计信息的前后台任务进行内存限制。

> **注意：**
>
> 只有在 TiDB 的启动配置文件中开启了 run-auto-analyze 选项，该 TiDB 集群才会触发 auto_analyze。

4.1.1.244 tidb_mem_quota_apply_cache 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：整数型
- 默认值：33554432 (32 MiB)
- 范围：[0, 9223372036854775807]
- 单位：字节
- 这个变量用来设置 Apply 算子中局部 Cache 的内存使用阈值。
- Apply 算子中局部 Cache 用来加速 Apply 算子的计算，该变量可以设置 Apply Cache 的内存使用阈值。设置变量值为 0 可以关闭 Apply Cache 功能。

4.1.1.245 tidb_mem_quota_binding_cache 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：67108864 (64 MiB)
- 范围：[0, 2147483647]
- 单位：字节
- 这个变量用来设置存放 binding 的缓存的内存使用阈值。
- 如果一个系统创建或者捕获了过多的绑定，导致绑定所使用的内存空间超过该阈值，TiDB 会在日志中增加警告日志进行提示。这种情况下，缓存无法存放所有可用的绑定，并且无法保证哪些绑定存在于缓存中，因此，可能存在一些查询无法使用可用绑定的情况。此时，可以调大该变量的值，从而保证所有可用绑定都能正常使用。修改变量值以后，需要执行命令 admin reload bindings 重新加载绑定，确保变更生效。

4.1.1.246 tidb_mem_quota_query

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1073741824 (1 GiB)
- 范围：[-1, 9223372036854775807]
- 单位：字节
- 在 v6.1.0 之前的版本中，作用域为 SESSION。v6.1.0 及之后的版本，作用域变更为 SESSION | GLOBAL。
- 在 v6.5.0 之前的版本中，该变量用来设置单条查询的内存使用限制，如果单条查询执行过程中使用的内存量超过该阈值，会触发系统变量tidb_mem_oom_action 中指定的行为。
- 在 v6.5.0 及之后的版本中，该变量用来设置单个会话整体的内存使用限制，如果某个会话执行过程中使用的内存量超过该阈值，会触发系统变量tidb_mem_oom_action 中指定的行为。需要注意的是，自 v6.5.0 版本开始，会话的内存使用量包含会话中事务所消耗的内存。v6.5.0 及之后版本对事务内存的控制行为有所变化，详见txn-total-size-limit。
- 当变量值为 0 或 -1 时，表示内存阈值为正无穷。此外，当变量值小于 128 时，将默认被设置为 128。

4.1.1.247 tidb_memory_debug_mode_alarm_ratio

- 作用域：SESSION
- 类型：浮点型

- 默认值：0
- 该变量表示在TiDB memory debug 模式下，允许的内存统计误差值。
- 该变量用于TiDB 内部测试，不推荐修改该变量值。

4.1.1.248 tidb_memory_debug_mode_min_heap_inuse

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 该变量用于TiDB 内部测试，不推荐修改该变量值，因为开启后会影响TiDB 的性能。
- 配置此参数后，TiDB 会进入 memory debug 模式进行内存追踪准确度的分析。TiDB 会在后续执行 SQL 语句的过程中频繁触发 GC， 并将实际内存使用和内存统计值做对比。若当前内存使用大于 tidb_memory_debug_mode_min_heap_inuse 且内存统计误差超过 tidb_memory_debug_mode_alarm_ratio，则会输出相关内存信息到日志和文件中。

4.1.1.249 tidb_memory_usage_alarm_ratio

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0.7
- 范围：[0.0, 1.0]
- 这个变量用于设置触发tidb-server 内存告警的内存使用比率。默认情况下，当TiDB 内存使用量超过总内存的 70% 且满足报警条件时，TiDB 会打印报警日志。
- 当配置该变量的值为 0 或 1 时，表示关闭内存阈值报警功能。
- 当配置该变量为 0 到 1 之间的值时，表示开启内存阈值报警功能：
  - 如果系统变量tidb_server_memory_limit 等于0,则内存报警阈值为tidb_memory_usage_alarm_ratio <—> * 系统内存大小。
  - 如果系统变量tidb_server_memory_limit 被设置为大于0,则内存报警阈值为tidb_memory_usage_alarm_ratio <—> * tidb_server_memory_limit。

4.1.1.250 tidb_memory_usage_alarm_keep_record_num 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：5
- 范围：[1, 10000]
- 当 tidb-server 内存占用超过内存报警阈值并触发报警时，TiDB 默认只保留最近 5 次报警时所生成的状态文件。你可以通过该变量调整该次数。

4.1.1.251 tidb_merge_join_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 取值范围：[1, 256]
- 设置 MergeJoin 算子执行查询时的并发度。
- 不推荐设置该变量，修改该变量值可能会造成数据正确性问题。

4.1.1.252 tidb_merge_partition_stats_concurrency

> 警告：
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 这个变量用于TiDB analyze 分区表时，对分区表统计信息进行合并时的并发度。

4.1.1.253 tidb_metric_query_range_duration 从 v4.0 版本开始引入

- 作用域：SESSION
- 类型：整数型
- 默认值：60
- 范围：[10, 216000]
- 单位：秒
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的range duration。

4.1.1.254 tidb_metric_query_step 从 v4.0 版本开始引入

- 作用域：SESSION
- 类型：整数型
- 默认值：60
- 范围：[10, 216000]
- 单位：秒
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 step。

4.1.1.255 tidb_min_paging_size 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：128
- 范围：[1，9223372036854775807]
- 单位：行
- 这个变量用来设置coprocessor 协议中paging size 的最小的行数。请合理设置该值，设置过小，TiDB 与 TiKV 的 RPC 交互会更频繁；设置过大，IndexLookup 带 Limit 场景会出现性能下降。该变量的默认值对于 OLTP 场景较友好，如果业务只使用了TiKV 作为存储引擎，当执行偏 OLAP 的负载时，可以考虑将变量值调大，有可能获得更好的性能。

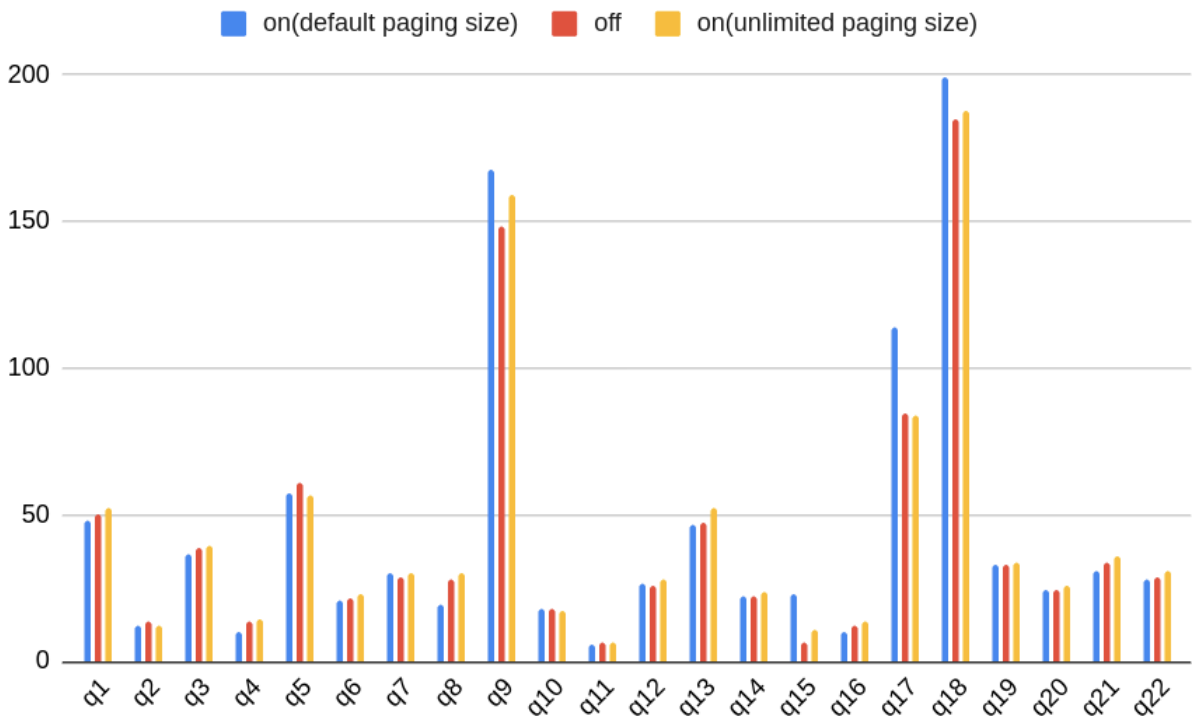


图 50: Paging size impact on TPCH

开启tidb_enable_paging 时，tidb_min_paging_size 和tidb_max_paging_size 对 TPCH 的性能影响如上图所示，纵轴是执行时间，越小越好。

4.1.1.256 tidb_mpp_store_fail_ttl

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：Duration
- 默认值：60s
- 刚重启的TiFlash 可能不能正常提供服务。为了防止查询失败，TiDB 会限制tidb-server 向刚重启的TiFlash 节点发送查询。这个变量表示刚重启的TiFlash 不被发送请求的时间范围。

4.1.1.257 tidb_multi_statement_mode 从 v4.0.11 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：OFF
- 可选值：OFF，ON，WARN
- 该变量用于控制是否在同一个 COM_QUERY 调用中执行多个查询。
- 为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。该变量可用作早期TiDB 版本的升级路径选项。该变量值与是否允许多语句行为的对照表如下：

客户端设置	tidb_multi_statement_mode 值	是否允许多语句
Multiple Statements = ON	OFF	允许
Multiple Statements = ON	ON	允许
Multiple Statements = ON	WARN	允许
Multiple Statements = OFF	OFF	不允许
Multiple Statements = OFF	ON	允许
Multiple Statements = OFF	WARN	允许+警告提示

> 注意：
>
> 只有默认值 OFF 才是安全的。如果用户业务是专为早期TiDB 版本而设计的，那么需要将该变量值设为 ON。如果用户业务需要多语句支持，建议用户使用客户端提供的设置，不要使用 tidb_multi_statement_mode 变量进行设置。

> - go-sql-driver (multiStatements)
> - Connector/J (allowMultiQueries)
> - PHP mysqli (mysqli_multi_query)

4.1.1.258 tidb_opt_agg_push_down

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置优化器是否执行聚合函数下推到Join，Projection 和 UnionAll 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 ON。

4.1.1.259 tidb_opt_broadcast_cartesian_join

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2]
- 表示是否允许 Broadcast Cartesian Join 算法。
- 值为0 时表示不允许使用Broadcast Cartesian Join 算法。值为1 时表示根据tidb_broadcast_join_threshold_count ⟷ 的行数阈值确定是否允许使用 Broadcast Cartesian Join 算法。值为 2 时表示总是允许 Broadcast Cartesian Join 算法，即使表的大小超过了该阈值。
- 该变量是TiDB 内部使用的变量，不推荐修改该变量的值。

4.1.1.260 tidb_opt_concurrency_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：3.0
- 表示在TiDB 中开启一个Golang goroutine 的 CPU 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.261 tidb_opt_copcpu_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：3.0
- 表示TiKV 协处理器处理一行数据的 CPU 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.262 tidb_opt_correlation_exp_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：1
- 范围：[0, 2147483647]
- 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。

4.1.1.263 tidb_opt_correlation_threshold

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0.9
- 范围：[0, 1]
- 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。

4.1.1.264 tidb_opt_cpu_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：3.0
- 表示TiDB 处理一行数据的 CPU 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.265 tidb_opt_derive_topn 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 表示是否开启从窗口函数中推导TopN 或 Limit 的优化规则。

4.1.1.266 tidb_opt_desc_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：3.0
- 表示降序扫描时，TiKV 在磁盘上扫描一行数据的开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.267 tidb_opt_disk_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：1.5
- 表示TiDB 往临时磁盘读写一个字节数据的I/O 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.268 tidb_opt_distinct_agg_push_down

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置优化器是否执行带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。当查询中带有 Distinct 的聚合操作执行很慢时，可以尝试设置该变量为 1。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从TiKV 读取所有数据，并在TiDB 侧执行 distinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增里一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    <-->                        +           +           +                +
    <-->
| id                        | estRows   | task       | access object | operator info
    <-->                              |
+--
    <-->                        +           +           +                +
    <--> ------------------------ ----------- ------------ -----------------
| StreamAgg_6               | 1.00      | root       |               | funcs:count(distinct test.t.a)
    <--> ->Column#4 |
| └─TableReader_10          | 10000.00 | root       |               | data:TableFullScan_9
    <-->                       |
|    └─TableFullScan_9      | 10000.00 | cop[tikv] | table:t        | keep order:false, stats:
    <--> pseudo             |
+--
    <-->                        +           +           +                +
    <-->
    ------------------------ ----------- ------------ -----------------
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> desc select count(distinct a) from test.t;
+--
    <⟷                          +          +           +               +
    <⟷
| id                          | estRows  | task      | access object | operator info
    <⟷                             |
+--
    <⟷-------------------------+----------+-----------+---------------+
    <⟷
| HashAgg_8                   | 1.00     | root      |               | funcs:count(distinct test.t.
    <⟷ a)->Column#3 |
| └─TableReader_9             | 1.00     | root      |               | data:HashAgg_5
    <⟷                              |
|   └─HashAgg_5               | 1.00     | cop[tikv] |               | group by:test.t.a,
    <⟷                           |
|     └─TableFullScan_7       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    <⟷ pseudo            |
+--
    <⟷                             +          +           +               +
    <⟷
4 rows in set (0.00 sec)
```

4.1.1.269 tidb_opt_enable_correlation_adjustment

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制优化器是否开启交叉估算。

4.1.1.270 tidb_opt_enable_late_materialization 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否启用TiFlash 延迟物化功能。注意在TiFlash Fast Scan 模式下，延迟物化功能暂不可用。
- 当设置该变量为 OFF 关闭 TiFlash 延迟物化功能时，如果 SELECT 语句中包含过滤条件（WHERE 子句），TiFlash 会先扫描查询所需列的全部数据后再进行过滤。当设置该变量为 ON 开启TiFlash 延迟物化功能时，TiFlash 会先扫描下推到 TableScan 算子的过滤条件相关的列数据，过滤得到符合条件的行后，再扫描这些行的其他列数据，继续后续计算，从而减少IO 扫描和数据处理的计算量。

4.1.1.271 tidb_opt_fix_control 从 v7.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：字符串
- 默认值：""
- 这个变量用来控制优化器的一些内部行为。
- 一部分优化器行为的选择依赖用户场景或 SQL 编写方式。通过设置该变量，你可以更细粒度地控制优化器的行为，并且避免集群升级后优化器行为变化导致的性能回退。
- 该变量支持设置多个控制项，不同控制项之间使用逗号(,) 分隔。格式形如 "<#issue1>:<value1>,<#issue2>:<value2>,...,<#issueN>:<valueN>"，其中 <#issueN> 代表控制项编号，对应链接 https://github.com/pingcap/tidb/issues/<#issue> 对该控制项进行说明，<valueN> 是为该控制项设置的目标值。

4.1.1.272 tidb_opt_force_inline_cte 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否强制inline CTE。默认值为 OFF，即默认不强制 inline CTE。注意，此时依旧可以通过 MERGE() hint 来开启个别 CTE 的 inline。如果设置为 ON，则当前 session 中所有查询的 CTE（递归 CTE 除外）都会inline。

4.1.1.273 tidb_opt_advanced_join_hint 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制包括HASH_JOIN() Hint、MERGE_JOIN() Hint 等用于控制连接算法的Join Method Hint 是否会影响Join Reorder 的优化过程，包括LEADING() Hint 的使用。默认值为 ON，即默认不影响。如果设置为 OFF，在一些同时使用 Join Method Hint 和 LEADING() Hint 的场景下可能会产生冲突。

> **注意：**
>
> v7.0.0 之前的版本行为和将该变量设置为 OFF 的行为一致。为确保向前兼容，从旧版本升级到 v7.0.0 及之后版本的集群，该变量会被设置成 OFF。为了获取更灵活的 Hint 行为，强烈建议在确保无性能回退的情况下，将该变量切换为 ON。

4.1.1.274 tidb_opt_insubq_to_join_and_agg

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来设置是否开启优化规则：将子查询转成join 和 aggregation。
  例如，打开这个优化规则后，会将下面子查询做如下变化：

```
select * from t where t.a in (select aa from t1);
```

  将子查询转成如下join：

```
select t.* from t, (select aa from t1 group by aa) tmp_t where t.a = tmp_t.aa;
```

  如果t1 在列 aa 上有unique 且 not null 的限制，可以直接改写为如下，不需要添加aggregation。

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

4.1.1.275 tidb_opt_join_reorder_threshold

- 作用域: SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 这个变量用来控制TiDB Join Reorder 算法的选择。当参与Join Reorder 的节点个数大于该阈值时，TiDB 选择贪心算法，小于该阈值时TiDB 选择动态规划(dynamic programming) 算法。
- 目前对于 OLTP 的查询，推荐保持默认值。对于 OLAP 的查询，推荐将变量值设为 10~15 来获得AP 场景下更好的连接顺序。

4.1.1.276 tidb_opt_limit_push_down_threshold

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：100
- 范围：[0, 2147483647]
- 这个变量用来设置将 Limit 和 TopN 算子下推到TiKV 的阈值。
- 如果 Limit 或者 TopN 的取值小于等于这个阈值，则 Limit 和 TopN 算子会被强制下推到TiKV。该变量可以解决部分由于估算误差导致 Limit 或者 TopN 无法被下推的问题。

4.1.1.277 tidb_opt_memory_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：0.001
- 表示TiDB 存储一行数据的内存开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.278 tidb_opt_mpp_outer_join_fixed_build_side 从 v5.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 当该变量值为 ON 时，左连接始终使用内表作为构建端，右连接始终使用外表作为构建端。将该变量值设为 OFF 后，外连接可以灵活选择任意一边表作为构建端。

4.1.1.279 tidb_opt_network_factor

- 作用域：SESSION | GLOBAL
- 是否持久化
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：1.0
- 表示传输 1 比特数据的网络净开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.280 tidb_opt_ordering_index_selectivity_threshold 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0
- 范围：[0, 1]
- 用于当 SQL 中存在 ORDER BY 和 LIMIT 子句且带有过滤条件时，控制优化器选择索引的行为。
- 对于此类查询，优化器会考虑选择对应的索引来满足 ORDER BY 和 LIMIT 子句（即使这个索引并不满足任何过滤条件）。但是由于数据分布的复杂性，优化器在这种场景下可能会选择不优的索引。
- 该变量表示一个阈值。当存在索引能满足过滤条件，且其选择率估算值低于该阈值时，优化器会避免选择用于满足 ORDER BY 和 LIMIT 的索引，而优先选择用于满足过滤条件的索引。
- 例如，当把该变量设为 0 时，优化器保持默认行为；当设为 1 时，优化器总是优先选择满足过滤条件的索引，避免选择满足 ORDER BY 和 LIMIT 的索引。
- 在以下示例中，t 表共有 1,000,000 行数据。使用 b 列上的索引时，其估算行数是大约 8,748 行，因此其选择率估算值大约是 0.0087。默认情况下，优化器选择了 a 列上的索引。而将该变量设为 0.01 之后，由于 b 列上的索引的选择率(0.0087) 低于 0.01，优化器选择了b 列上的索引。

```
> EXPLAIN SELECT * FROM t WHERE b <= 9000 ORDER BY a LIMIT 1;
+--
    <-> ---------------------------------+---------+-----------+----------------------+-------------------+
    <->
| id                                | estRows | task      | access object        | operator info
    <->     |
+--
    <->                                 +         +           +                      +                   +
    <-> ---------------------------------- --------- ----------- ---------------------- -------------------
    <->
| Limit_12                          | 1.00    | root      |                      | offset:0,
    <-> count:1  |
| └─Projection_25                    | 1.00    | root      |                       | test.t.a,
    <-> test.t.b |
|   └─IndexLookUp_24                 | 1.00    | root      |                       |
    <->                     |
|     ├─IndexFullScan_21(Build)      | 114.30  | cop[tikv] | table:t, index:ia(a) | keep order:
    <-> true     |
|     └─Selection_23(Probe)          | 1.00    | cop[tikv] |                       | le(test.t.b,
    <-> 9000) |
|       └─TableRowIDScan_22          | 114.30  | cop[tikv] | table:t               | keep order:
    <-> false    |
+--
    <->                                 +         +           +                      +                   +
    <-> ---------------------------------- --------- ----------- ---------------------- -------------------

> SET SESSION tidb_opt_ordering_index_selectivity_threshold = 0.01;

> EXPLAIN SELECT * FROM t WHERE b <= 9000 ORDER BY a LIMIT 1;
+--
    <->                                 +         +           +                      +
    <>---------------------------------- --------- ----------- ---------------------- ----------------------------
| id                                | estRows | task      | access object        | operator info
    <->                       |
+--
    <->                                 +         +           +                      +
    <>---------------------------------- --------- ----------- ---------------------- ----------------------------
| TopN_9                            | 1.00    | root      |                      | test.t.a,
    <-> offset:0, count:1          |
| └─IndexLookUp_20                   | 1.00    | root      |                       |
    <->                                     |
|   ├─IndexRangeScan_17(Build)       | 8748.62 | cop[tikv] | table:t, index:ib(b) | range:[-inf
    <-> ,9000], keep order:false |
|   └─TopN_19(Probe)                 | 1.00    | cop[tikv] |                       | test.t.a,
    <-> offset:0, count:1          |
```

```
|     └─TableRowIDScan_18          | 8748.62 | cop[tikv] | table:t               | keep order:
    <-> false                      |
+--
    <->                                  +          +           +                       +
    <->
```

4.1.1.281 tidb_opt_prefer_range_scan 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 将该变量值设为 ON 后，优化器总是偏好区间扫描而不是全表扫描。
- 在以下示例中，tidb_opt_prefer_range_scan 开启前，TiDB 优化器需要执行全表扫描。tidb_opt_prefer_range_scan <-> 开启后，优化器选择了索引区间扫描。

```
explain select * from t where age=5;
+-------------------------+------------+-----------+---------------+-------------------------------+
| id                      | estRows    | task      | access object | operator info                 |
+-------------------------+------------+-----------+---------------+-------------------------------+
| TableReader_7           | 1048576.00 | root      |               | data:Selection_6              |
| └─Selection_6           | 1048576.00 | cop[tikv] |               | eq(test.t.age, 5)             |
|   └─TableFullScan_5     | 1048576.00 | cop[tikv] | table:t       | keep order:false              |
+-------------------------+------------+-----------+---------------+-------------------------------+
3 rows in set (0.00 sec)

set session tidb_opt_prefer_range_scan = 1;

explain select * from t where age=5;
+--
    <->                              +           +           +                           +
    <->
| id                           | estRows    | task      | access object               | operator
    <-> info                  |
+--
    <->                              +           +           +                           +
    <->
| IndexLookUp_7                | 1048576.00 | root      |                             |
    <->                            |
| ├─IndexRangeScan_5(Build)    | 1048576.00 | cop[tikv] | table:t, index:idx_age(age) | range
    <-> :[5,5], keep order:false |
| └─TableRowIDScan_6(Probe)    | 1048576.00 | cop[tikv] | table:t                     | keep
    <-> order:false            |
```

```
+--
    <-->----------------------------+-----------+-------------------+--------------------------+-----------------------------------
    <-->
3 rows in set (0.00 sec)
```

4.1.1.282 tidb_opt_projection_push_down 从 v6.1.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 指定是否允许优化器将 Projection 算子下推到TiKV 或者TiFlash。

4.1.1.283 tidb_opt_range_max_size 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：67108864 (64 MiB)
- 取值范围：[0, 9223372036854775807]
- 单位：字节
- 该变量用于指定优化器构造扫描范围的内存用量上限。当该变量为 0 时，表示对扫描范围没有内存限制。如果构造精确的扫描范围会超出内存用量限制，优化器会使用更宽松的扫描范围（例如 [[NULL,+inf]]）。如果执行计划中未使用精确的扫描范围，可以调大该变量的值让优化器构造精确的扫描范围。

该变量的使用示例如下：

tidb_opt_range_max_size 使用示例

查看该变量的默认值，即优化器构造扫描范围最多使用 64 MiB 内存。

```
SELECT @@tidb_opt_range_max_size;
```

```
+-----------------------------+
| @@tidb_opt_range_max_size |
+-----------------------------+
| 67108864                    |
+-----------------------------+
1 row in set (0.01 sec)
```

```
EXPLAIN SELECT * FROM t use index (idx) WHERE a IN (10,20,30) AND b IN (40,50,60);
```

在 64 MiB 的内存最大限制约束下，优化器构造出精确的扫描范围[10 40,10 40], [10 50,10 50], [10 60,10 <--> 60], [20 40,20 40], [20 50,20 50], [20 60,20 60], [30 40,30 40], [30 50,30 50], [30 60,30 60]，见如下执行计划返回结果。

```
+--
    <->                                +         +           +                          +
    <->
| id                            | estRows | task      | access object             | operator info
    <->
    <-> |
+--
    <->-------------------------------+---------+-----------+---------------------------+
    <->
| IndexLookUp_7                 | 0.90    | root      |                           |
    <->
    <-> |
| ├─IndexRangeScan_5(Build)     | 0.90    | cop[tikv] | table:t, index:idx(a, b) | range:[10
    <-> 40,10 40], [10 50,10 50], [10 60,10 60], [20 40,20 40], [20 50,20 50], [20 60,20 60], [30
    <->  40,30 40], [30 50,30 50], [30 60,30 60], keep order:false, stats:pseudo |
| └─TableRowIDScan_6(Probe)     | 0.90    | cop[tikv] | table:t                   | keep order:
    <-> false, stats:pseudo
    <->
    <-> |
+--
    <->                                +         +           +                          +
    <->
3 rows in set (0.00 sec)
```

现将优化器构造扫描范围的内存用量上限设为 1500 字节。

```
SET @@tidb_opt_range_max_size = 1500;
```

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

```
EXPLAIN SELECT * FROM t USE INDEX (idx) WHERE a IN (10,20,30) AND b IN (40,50,60);
```

在 1500 字节内存的最大限制约束下，优化器构造出了更宽松的扫描范围 `[10,10]`, `[20,20]`, `[30,30]`，并用 warning 提示用户构造精确的扫描范围所需的内存用量超出了 `tidb_opt_range_max_size` 的限制。

```
+--
    <->                                +         +           +                          +
    <->
| id                            | estRows | task      | access object             | operator info
    <->                                                    |
+--
    <>-------------------------------+---------+-----------+---------------------------+
    <->
| IndexLookUp_8                 | 0.09    | root      |                           |
    <->                                                              |
```

```
| ├─Selection_7(Build)          | 0.09    | cop[tikv] |                          | in(test.t.b,
    <-> 40, 50, 60)                                          |
| | └─IndexRangeScan_5          | 30.00   | cop[tikv] | table:t, index:idx(a, b) | range
    <-> :[10,10], [20,20], [30,30], keep order:false, stats:pseudo |
| └─TableRowIDScan_6(Probe)     | 0.09    | cop[tikv] | table:t                  | keep order:
    <-> false, stats:pseudo                                    |
+--
    <->                                +         +           +                          +
    <->
4 rows in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+--
    <->--------+-------+-----------------------------------------------------------------------------------
    <->
| Level   | Code | Message
    <->
    <-> |
+--
    <->---------+------+-----------------------------------------------------------------------------------
    <->
| Warning | 1105 | Memory capacity of 1500 bytes for 'tidb_opt_range_max_size' exceeded when
    <-> building ranges. Less accurate ranges such as full range are chosen |
+--
    <->         +      +
    <->
1 row in set (0.00 sec)
```

再将优化器构造扫描范围的内存用量上限设为 100 字节。

```
set @@tidb_opt_range_max_size = 100;
```

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

```
EXPLAIN SELECT * FROM t USE INDEX (idx) WHERE a IN (10,20,30) AND b IN (40,50,60);
```

在 100 字节的内存最大限制约束下，优化器选择了 IndexFullScan，并用 warning 提示用户构造精确的扫描范围所需的内存超出了 tidb_opt_range_max_size 的限制。

```
+--
    <->                                +         +           +                          +
    <->
| id                            | estRows   | task      | access object             | operator info
    <->                                      |
```

```
+--
    <-->                                +           +           +                          +
    <-->
| IndexLookUp_8                  | 8000.00  | root      |                            |
    <->                                                  |
| ├─Selection_7(Build)           | 8000.00  | cop[tikv] |                            | in(test.t.a,
    <->  10, 20, 30), in(test.t.b, 40, 50, 60) |
| │ └─IndexFullScan_5            | 10000.00 | cop[tikv] | table:t, index:idx(a, b) | keep order:
    <-> false, stats:pseudo                      |
| └─TableRowIDScan_6(Probe)      | 8000.00  | cop[tikv] | table:t                    | keep order:
    <-> false, stats:pseudo                      |
+--
    <->                                +           +           +                          +
    <-->
4 rows in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+--
    <->-------+------+-----------------------------------------------------------------------------------
    <->
| Level   | Code | Message
    <->
    <-> |
+--
    <->-------+------+-----------------------------------------------------------------------------------
    <->
| Warning | 1105 | Memory capacity of 100 bytes for 'tidb_opt_range_max_size' exceeded when
    <-> building ranges. Less accurate ranges such as full range are chosen |
+--
    <->       +      +
    <->-------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

4.1.1.284 tidb_opt_scan_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：1.5
- 表示升序扫描时，TiKV 在磁盘上扫描一行数据的开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.285 tidb_opt_seek_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：20
- 表示TiDB 从TiKV 请求数据的初始开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.286 tidb_opt_skew_distinct_agg 从 v6.2.0 版本开始引入

> 注意：
> 开启该变量带来的查询性能优化仅对TiFlash 有效。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
  - 这个变量用来设置优化器是否将带有 DISTINCT 的聚合函数（例如 SELECT b, count(DISTINCT a)FROM t GROUP BY b）改写为两层聚合函数（例如 SELECT b, count(a)FROM (SELECT b, a FROM t GROUP BY b, a)t GROUP BY b）。当聚合列有严重的数据倾斜，且 DISTINCT 列有很多不同的值时，这种改写能够避免查询执行过程中的数据倾斜，从而提升查询性能。

4.1.1.287 tidb_opt_three_stage_distinct_agg 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量用于控制在 MPP 模式下是否将 COUNT(DISTINCT) 聚合改写为三阶段分布式执行的聚合。
- 该变量目前仅对只有一个 COUNT(DISTINCT) 的聚合生效。

4.1.1.288 tidb_opt_tiflash_concurrency_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：24.0
- 表示TiFlash 计算的并发数。该变量是代价模型内部使用的变量，不建议修改该变量的值。

4.1.1.289 tidb_opt_write_row_id

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 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。

4.1.1.290 tidb_optimizer_selectivity_level

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 控制优化器估算逻辑的更迭。更改该变量值后，优化器的估算逻辑会产生较大的改变。目前该变量的有效值只有 0，不建议设为其它值。

4.1.1.291 tidb_partition_prune_mode 从 v5.1 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：dynamic
- 可选值：static、dynamic、static-only、dynamic-only
- 这个变量用来设置是否开启分区表动态裁剪模式。默认值为 dynamic。但是注意，dynamic 模式仅在表级别汇总统计信息（即 GlobalStats）收集完成的情况下生效。如果选择了 dynamic 但 GlobalStats 未收集完成，TiDB 会仍采用 static 模式。关于GlobalStats 更多信息，请参考动态裁剪模式下的分区表统计信息。关于动态裁剪模式更多信息，请参考分区表动态裁剪模式。

4.1.1.292 tidb_persist_analyze_options 从 v5.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否开启ANALYZE 配置持久化特性。

4.1.1.293 tidb_pessimistic_txn_fair_locking 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON

- 是否对悲观锁启用加强的悲观锁唤醒模型。该模型可严格控制悲观锁单点冲突场景下事务的唤醒顺序，避免无效唤醒，大大降低原有唤醒机制中的随机性对事务延迟带来的不确定性。如果业务场景中遇到了单点悲观锁冲突频繁的情况（如高频更新同一行数据等），并进而引起语句重试频繁、尾延迟高，甚至偶尔发生 `pessimistic lock retry limit reached` 错误，可以尝试开启该变量来解决问题。
- 对于从 v7.0.0 以前的版本升级到 v7.0.0 或更新版本的TiDB 集群，该选项默认关闭。

> 注意：
>
> - 视具体业务场景的不同，启用该选项可能对存在频繁锁冲突的事务造成一定程度的吞吐下降（平均延迟上升）。
> - 该选项目前仅对需要上锁单个 key 的语句有效。如果一个语句需要对多行同时上锁，则该选项不会对此类语句生效。
> - 该功能从 v6.6.0 版本引入。在 v6.6.0 版本中，该功能由变量 `tidb_pessimistic_txn_aggressive_locking` 控制，默认关闭。

4.1.1.294 `tidb_placement_mode` 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：`STRICT`
- 可选值：`STRICT`，`IGNORE`
- 该变量用于控制 DDL 语句是否忽略Placement Rules in SQL 指定的放置规则。变量值为 `IGNORE` 时将忽略所有放置规则选项。
- 该变量可由逻辑转储或逻辑恢复工具使用，确保即使绑定了不合适的放置规则，也始终可以成功创建表。这类似于mysqldump 将 `SET FOREIGN_KEY_CHECKS=0;` 写入每个转储文件的开头部分。

4.1.1.295 `tidb_plan_cache_invalidation_on_fresh_stats` 从 v7.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：`ON`
- 该变量用于控制当某张表上的统计信息更新后，与该表相关的 Plan Cache 是否自动失效。
- 开启此变量有助于 Plan Cache 更有效地利用可用的统计信息生成执行计划，例如：
  - 有时 Plan Cache 会在统计信息尚不可用时生成执行计划。开启此变量后，Plan Cache 会在统计信息可用时重新生成执行计划。
  - 当表上数据分布发生变化时，之前的最优执行计划可能对于现在不再是最优的。开启此变量后，Plan Cache 会在重新收集统计信息后重新生成执行计划。
- 对于从 v7.1.0 以前的版本升级到 v7.1.0 及以上版本的TiDB 集群，该选项默认关闭(OFF)。

4.1.1.296 tidb_plan_cache_max_plan_size 从 v7.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：2097152（即 2 MB）
- 取值范围：[0, 9223372036854775807]，单位为 Byte。支持带单位的内存格式“KB|MB|GB|TB”。0 表示表示不设限制。
- 这个变量用来控制可以缓存的 Prepare 或非 Prepare 语句执行计划的最大大小。超过该值的执行计划将不会被缓存到 Plan Cache 中。详情请参考Prepare 语句执行计划缓存和非 Prepare 语句执行计划缓存。

4.1.1.297 tidb_pprof_sql_cpu 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：整数型
- 默认值：0
- 范围：[0, 1]
- 这个变量用来控制是否在profile 输出中标记出对应的 SQL 语句，用于定位和排查性能问题。

4.1.1.298 tidb_opt_prefix_index_single_scan 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制TiDB 优化器是否将某些过滤条件下推到前缀索引，尽量避免不必要的回表，从而提高查询性能。
- 将该变量设置为 ON 时，会将过滤条件下推到前缀索引。此时，假设一张表中 col 列是索引前缀列，查询语句中的 col is null 或者 col is not null 条件会被归为索引上的过滤条件，而不是回表时的过滤条件，从而避免不必要的回表。

该变量的使用示例

创建一张带前缀索引的表：

```
CREATE TABLE t (a INT, b VARCHAR(10), c INT, INDEX idx_a_b(a, b(5)));
```

此时关闭 tidb_opt_prefix_index_single_scan：

```
SET tidb_opt_prefix_index_single_scan = 'OFF';
```

对于以下查询，执行计划使用了前缀索引 idx_a_b 但需要回表（出现了 IndexLookUp 算子）。

```
EXPLAIN FORMAT='brief' SELECT COUNT(1) FROM t WHERE a = 1 AND b IS NOT NULL;
+--
    <-->                              +          +            +                                  +

    <-->
```

```
| id                            | estRows | task      | access object                | operator
    <-> info                                          |
+--
    <->                           +         +           +                              +
    <->
| HashAgg                       | 1.00    | root      |                              | funcs:
    <-> count(Column#8)->Column#5                       |
| └─IndexLookUp                 | 1.00    | root      |                               |
    <->                                                 |
|   ├─IndexRangeScan(Build)     | 99.90   | cop[tikv] | table:t, index:idx_a_b(a, b) | range:[1
    <-> -inf,1 +inf], keep order:false, stats:pseudo |
|   └─HashAgg(Probe)            | 1.00    | cop[tikv] |                               | funcs:
    <-> count(1)->Column#8                              |
|     └─Selection               | 99.90   | cop[tikv] |                               | not(
    <-> isnull(test.t.b))                               |
|       └─TableRowIDScan        | 99.90   | cop[tikv] | table:t                      | keep
    <-> order:false, stats:pseudo                       |
+--
    <->                             +         +           +                              +
    <->
6 rows in set (0.00 sec)
```

此时打开 tidb_opt_prefix_index_single_scan：

```
SET tidb_opt_prefix_index_single_scan = 'ON';
```

开启该变量后，对于以下查询，执行计划使用了前缀索引 idx_a_b 且不需要回表。

```
EXPLAIN FORMAT='brief' SELECT COUNT(1) FROM t WHERE a = 1 AND b IS NOT NULL;
+--
    <->                           +         +           +                              +
    <->
| id                          | estRows | task      | access object                | operator info
    <->                                   |
+--
    <->                           +         +           +                              +
    <->
| StreamAgg                   | 1.00    | root      |                               | funcs:count(
    <-> Column#7)->Column#5                             |
| └─IndexReader               | 1.00    | root      |                               | index:
    <-> StreamAgg                                       |
|   └─StreamAgg               | 1.00    | cop[tikv] |                               | funcs:count(1)
    <-> ->Column#7                                      |
|     └─IndexRangeScan        | 99.90   | cop[tikv] | table:t, index:idx_a_b(a, b) | range:[1 -inf
    <-> ,1 +inf], keep order:false, stats:pseudo |
+--
    <->                           +         +           +                              +
```

```
    <->
4 rows in set (0.00 sec)
```

4.1.1.299 tidb_prefer_broadcast_join_by_exchange_data_size 从 v7.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于设定TiDB 选择MPP Hash Join 算法时，是否使用最小网络交换的数据量策略。开启该变量后，TiDB 会估算 Broadcast Hash Join 和 Shuffled Hash Join 两种算法所需进行网络交换的数据量，并选择网络交换数据量较小的算法。
- 该功能开启后tidb_broadcast_join_threshold_count 和tidb_broadcast_join_threshold_size 将不再生效。

4.1.1.300 tidb_prepared_plan_cache_memory_guard_ratio 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0.1
- 范围：[0, 1]
- 这个变量用来控制 Prepared Plan Cache 触发内存保护机制的阈值，具体可见Prepared Plan Cache 的内存管理。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(prepared-plan-cache.memory-guard-ratio) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.301 tidb_prepared_plan_cache_size 从 v6.1.0 版本开始引入

> 警告：
>
> 从 v7.1.0 开始，该变量被废弃。请使用tidb_session_plan_cache_size 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：100
- 范围：[1, 100000]
- 这个变量用来控制单个 SESSION 的 Prepared Plan Cache 最多能够缓存的计划数量，具体可见Prepared Plan Cache 的内存管理。
- 在 v6.1.0 之前这个开关通过TiDB 配置文件(prepared-plan-cache.capacity) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

4.1.1.302 tidb_projection_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1, 256]
- 单位：线程
- 这个变量用来设置 Projection 算子的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.303 tidb_query_log_max_len

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4096 (4 KiB)
- 范围：[0, 1073741824]
- 单位：字节
- 该变量控制 SQL 语句输出的最大长度。当一条 SQL 语句的输出长度大于 tidb_query_log_max_len 时，输出将会被截断。
- 在 v6.1.0 之前这个开关也可以通过TiDB 配置文件(log.query-log-max-len) 进行配置，升级到 v6.1.0 后仅可通过系统变量配置。

4.1.1.304 tidb_rc_read_check_ts 从 v6.0.0 版本开始引入

> **警告：**
>
> - 该特性与replica-read 尚不兼容， 开启 tidb_rc_read_check_ts 的读请求无法使用replica-read，请勿同时开启两项特性。
> - 如果客户端使用游标操作，建议不开启 tidb_rc_read_check_ts 这一特性，避免前一批返回数据已经被客户端使用而语句最终会报错的情况。
> - 自 v7.0.0 版本开始，该变量对于使用prepared statement 协议下cursor fetch read 游标模式不再生效。

- 作用域：GLOBAL

- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：布尔型
- 默认值：OFF
- 该变量用于优化时间戳的获取，适用于悲观事务 READ-COMMITTED 隔离级别下读写冲突较少的场景，开启此变量可以避免获取全局timestamp 带来的延迟和开销，并优化事务内读语句延迟。
- 如果读写冲突较为严重，开启此功能会增加额外开销和延迟，造成性能回退。更详细的说明，请参考读已提交隔离级别(Read Committed) 文档。

4.1.1.305 tidb_rc_write_check_ts 从 v6.3.0 版本开始引入

> **警告：**
>
> 该特性与replica-read 尚不兼容。开启本变量后，客户端发送的所有请求都将无法使用 replica-read，因此请勿同时开启 tidb_rc_write_check_ts 和 replica-read。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于优化时间戳的获取，适用于悲观事务READ-COMMITTED 隔离级别下点写冲突较少的场景。开启此变量可以避免点写语句获取全局时间戳带来的延迟和开销。目前该变量适用的点写语句包括 UPDATE、DELETE、SELECT ..... FOR UPDATE 三种类型。点写语句是指将主键或者唯一键作为过滤条件且最终执行算子包含 POINT-GET 的写语句。
- 如果点写冲突较为严重，开启此变量会增加额外开销和延迟，造成性能回退。更详细的说明，请参考读已提交隔离级别(Read Committed) 文档。

4.1.1.306 tidb_read_consistency New in v5.4.0

- 作用域：SESSION
- 类型：字符串
- 默认值：strict
- 此变量用于控制自动提交的读语句的读一致性。
- 如果将变量值设置为 weak，则直接跳过读语句遇到的锁，读的执行可能会更快，这就是弱一致性读模式。但在该模式下，事务语义（例如原子性）和分布式一致性（线性一致性）并不能得到保证。
- 如果用户场景中需要快速返回自动提交的读语句，并且可接受弱一致性的读取结果，则可以使用弱一致性读取模式。

4.1.1.307 tidb_read_staleness 从 v5.4.0 版本开始引入

- 作用域：SESSION
- 类型：整数型
- 默认值：0

- 范围 [-2147483648, 0]
- 这个变量用于设置当前会话允许读取的历史数据范围。设置后，TiDB 会从参数允许的范围内选出一个尽可能新的时间戳，并影响后继的所有读操作。比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证TiKV 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

4.1.1.308 tidb_record_plan_in_slow_log

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否在 slow log 里包含慢查询的执行计划。

4.1.1.309 tidb_redact_log

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制在记录TiDB 日志和慢日志时，是否将 SQL 中的用户信息遮蔽。
- 将该变量设置为 1 即开启后，假设执行的 SQL 为 `insert into t values (1,2)`，在日志中记录的 SQL 会是 `insert into t values (?,?)`，即用户输入的信息被遮蔽。

4.1.1.310 tidb_regard_null_as_point 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制优化器是否可以将包含null 的等值条件作为前缀条件来访问索引。
- 该变量默认开启。开启后，该变量可以使优化器减少需要访问的索引数据量，从而提高查询的执行速度。例如，在有多列索引 `index(a, b)` 且查询条件为 `a<=>null and b=1` 的情况下，优化器可以同时使用查询条件中的 `a<=>null` 和 `b=1` 进行索引访问。如果关闭该变量，因为 `a<=>null and b=1` 包含null 的等值条件，优化器不会使用 `b=1` 进行索引访问。

4.1.1.311 tidb_remove_orderby_in_subquery 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 指定是否在子查询中移除 ORDER BY 子句。

4.1.1.312 tidb_replica_read 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：`leader`
- 可选值：`leader`、`follower`、`leader-and-follower`、`prefer-leader`、`closest-replicas`、`closest-adaptive` 和 `learner`。其中，`learner` 从 v6.6.0 开始引入。
- 这个变量用于控制TiDB 的 Follower Read 功能的行为。
- 关于使用方式与实现原理，见 Follower Read。

4.1.1.313 tidb_retry_limit

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：10
- 范围：`[-1, 9223372036854775807]`
- 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。

4.1.1.314 tidb_row_format_version

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：2
- 范围：`[1, 2]`
- 控制新保存数据的表数据格式版本。TiDB v4.0 中默认使用版本号为 2 的新表数据格式保存新数据。
- 但如果从 4.0.0 之前的版本升级到 4.0.0，不会改变表数据格式版本，TiDB 会继续使用版本为 1 的旧格式写入表中，即只有新创建的集群才会默认使用新表数据格式。
- 需要注意的是修改该变量不会对已保存的老数据产生影响，只会对修改变量后的新写入数据使用对应版本格式保存。

4.1.1.315 tidb_scatter_region

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型

- 默认值：OFF
- TiDB 默认会在建表时为新表分裂Region。开启该变量后，会在建表语句执行时，同步打散刚分裂出的 Region。适用于批量建表后紧接着批量写入数据，能让刚分裂出的 Region 先在TiKV 分散而不用等待 PD 进行调度。为了保证后续批量写入数据的稳定性，建表语句会等待打散 Region 完成后再返回建表成功，建表语句执行时间会是该变量关闭时的数倍。
- 如果建表时设置了 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS，建表成功后会均匀切分出指定数量的 Region。

4.1.1.316 tidb_server_memory_limit 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：80%
- 取值范围：
  - 你可以将该变量值设为百分比格式，表示内存用量占总内存的百分比，取值范围为 [1%, 99%]。
  - 你还可以将变量值设为内存大小，取值范围为 0 以及 [536870912, 9223372036854775807]，单位为 Byte。支持带单位的内存格式“KB|MB|GB|TB”。0 值表示不设内存限制。
  - 当设置的内存值小于 512 MB 且不为 0 时，TiDB 将会使用 512 MB 作为替代。
- 该变量指定TiDB 实例的内存限制。TiDB 会在内存用量达到该限制时，对当前内存用量最高的 SQL 语句进行取消(Cancel) 操作。在该 SQL 语句被成功 Cancel 掉后，TiDB 会尝试调用Golang GC 立刻回收内存，以最快速度缓解内存压力。
- 只有内存使用大于 tidb_server_memory_limit_sess_min_size 的 SQL 语句会被选定为最优先被 Cancel 的 SQL 语句。
- 目前TiDB 一次只能 Cancel 一条 SQL 语句。如果TiDB 完全 Cancel 掉一条 SQL 语句并回收资源后，内存使用仍然大于该变量所设限制，TiDB 会开始下一次 Cancel 操作。

4.1.1.317 tidb_server_memory_limit_gc_trigger 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：70%
- 取值范围：[50%, 99%]
- TiDB 尝试触发GC 的阈值。当iDB 的内存使用达到tidb_server_memory_limit 值* tidb_server_memory_limit_gc_trigger <—> 值时，则会主动触发一次Golang GC。在一分钟之内只会主动触发一次GC。

4.1.1.318 tidb_server_memory_limit_sess_min_size 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：134217728（即 128 MB）
- 取值范围：[128, 9223372036854775807]，单位为 Byte。支持带单位的内存格式“KB|MB|GB|TB”。
- 开启内存限制后，TiDB 会终止当前实例上内存用量最高的 SQL 语句。本变量指定此情况下 SQL 语句被终止的最小内存用量。如果TiDB 实例的内存超限是由许多内存使用量不明显的会话导致的，可以适当调小该变量值，使得更多会话成为 Cancel 的对象。

4.1.1.319 tidb_session_plan_cache_size 从 v7.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：100
- 范围：[1, 100000]
- 这个变量用来控制 Plan Cache 最多能够缓存的计划数量。其中，Prepare 语句执行计划缓存和非 Prepare 语句执行计划缓存共用一个缓存。
- 从旧版本升级到v7.1.0 及之后的版本，tidb_session_plan_cache_size 的值与tidb_prepared_plan_cache_size <→ 保持一致。

4.1.1.320 tidb_shard_allocate_step 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：9223372036854775807
- 范围：[1, 9223372036854775807]
- 该变量设置为AUTO_RANDOM 或SHARD_ROW_ID_BITS 属性列分配的最大连续ID 数。通常，AUTO_RANDOM ID 或带有 SHARD_ROW_ID_BITS 属性的行ID 在一个事务中是增量和连续的。你可以使用该变量来解决大事务场景下的热点问题。

4.1.1.321 tidb_simplified_metrics

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量开启后，TiDB 将不会收集或记录Grafana 面板未使用到的metrics。

4.1.1.322 tidb_skip_ascii_check 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否校验ASCII 字符的合法性。
- 校验ASCII 字符会损耗些许性能。当你确认输入的字符串为有效的ASCII 字符时，可以将其设置为 ON。

4.1.1.323 tidb_skip_isolation_level_check

- 作用域：SESSION | GLOBAL

- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 开启这个开关之后，如果对 tx_isolation 赋值一个TiDB 不支持的隔离级别，不会报错，有助于兼容其他设置了（但不依赖于）不同隔离级别的应用。

```
tidb> set tx_isolation='serializable';
ERROR 8048 (HY000): The isolation level 'serializable' is not supported. Set
    <—> tidb_skip_isolation_level_check=1 to skip this error
tidb> set tidb_skip_isolation_level_check=1;
Query OK, 0 rows affected (0.00 sec)

tidb> set tx_isolation='serializable';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

4.1.1.324 tidb_skip_utf8_check

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否校验 UTF-8 字符的合法性。
- 校验 UTF-8 字符会损耗些许性能。当你确认输入的字符串为有效的 UTF-8 字符时，可以将其设置为 ON。

> 注意：
>
> 跳过字符检查可能会使TiDB 检测不到应用写入的非法 UTF-8 字符，进一步导致执行 ANALYZE 时解码错误，以及引入其他未知的编码问题。如果应用不能保证写入字符串的合法性，不建议跳过该检查。

4.1.1.325 tidb_slow_log_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的TiDB 实例
- 默认值：300
- 类型：整数型
- 范围：[-1, 9223372036854775807]
- 单位：毫秒
- 输出慢日志的耗时阈值。当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。默认为 300 ms。

4.1.1.326 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。

4.1.1.327 tidb_snapshot

- 作用域：SESSION
- 默认值：“”
- 这个变量用来设置当前会话期待读取的历史数据所处时刻。比如当设置为 "2017-11-11 20:20:20" 时或者一个TSO 数字“400036290571534337”，当前会话将能读取到该时刻的数据。

4.1.1.328 tidb_source_id 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 15]
- 这个变量用来设置在双向复制系统内不同集群的ID。

4.1.1.329 tidb_stats_cache_mem_quota 从 v6.1.0 版本开始引入

> **警告：**
>
> 该变量为实验特性，不推荐在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 这个变量用于控制TiDB 内部统计信息缓存使用内存的上限。

4.1.1.330 tidb_stats_load_sync_wait 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：100
- 单位：毫秒
- 范围：[0, 2147483647]
- 这个变量用于控制是否开启统计信息的同步加载模式（为 0 代表不开启，即为异步加载模式），以及开启的情况下，SQL 执行同步加载完整统计信息等待多久后会超时。更多信息，请参考统计信息的加载。

4.1.1.331 tidb_stats_load_pseudo_timeout 从 v5.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制统计信息同步加载超时后，SQL 是执行失败（OFF），还是退回使用 pseudo 的统计信息（ON）。

4.1.1.332 tidb_stmt_summary_enable_persistent 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：GLOBAL
- 类型：布尔型
- 默认值：OFF
- 只读变量。表示是否开启statement summary tables 持久化。该变量的值与配置文件中tidb_stmt_summary_enable_persistent <—> 的取值相同。

4.1.1.333 tidb_stmt_summary_filename 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：GLOBAL
- 类型：字符串
- 默认值："tidb-statements.log"
- 只读变量。表示当开启statement summary tables 持久化后持久化数据所写入的文件。该变量的值与配置文件中tidb_stmt_summary_filename 的取值相同。

4.1.1.334 tidb_stmt_summary_file_max_backups 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：GLOBAL
- 类型：整数型
- 默认值：0
- 只读变量。表示当开启statement summary tables 持久化后持久化数据文件的最大数量限制。该变量的值与配置文件中tidb_stmt_summary_file_max_backups 的取值相同。

4.1.1.335 tidb_stmt_summary_file_max_days 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：GLOBAL
- 类型：整数型
- 默认值：3
- 单位：天
- 只读变量。表示当开启statement summary tables 持久化后持久化数据文件所保留的最大天数。该变量的值与配置文件中tidb_stmt_summary_file_max_days 的取值相同。

4.1.1.336 tidb_stmt_summary_file_max_size 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 作用域：GLOBAL
- 类型：整数型
- 默认值：64
- 单位：MiB
- 只读变量。表示当开启statement summary tables 持久化后持久化数据单个文件的大小限制。该变量的值与配置文件中tidb_stmt_summary_file_max_size 的取值相同。

4.1.1.337 tidb_stmt_summary_history_size 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：24
- 范围：[0, 255]
- 这个变量设置了 statement summary tables 的历史记录容量。

4.1.1.338 tidb_stmt_summary_internal_query 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否在 statement summary tables 中包含TiDB 内部 SQL 的信息。

4.1.1.339 tidb_stmt_summary_max_sql_length 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4096
- 范围：[0, 2147483647]
- 这个变量控制 statement summary tables 显示的 SQL 字符串长度。

4.1.1.340 tidb_stmt_summary_max_stmt_count 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：3000
- 范围：[1, 32767]
- 这个变量设置了 statement summary tables 在内存中保存的语句的最大数量。

4.1.1.341 tidb_stmt_summary_refresh_interval 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1800
- 范围：[1, 2147483647]
- 单位：秒
- 这个变量设置了 statement summary tables 的刷新时间。

4.1.1.342 tidb_store_batch_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4
- 范围：[0, 25000]
- 设置 IndexLookUp 算子回表时多个 Coprocessor Task 的 batch 大小。0 代表不使用batch。当 IndexLookUp 算子的回表 Task 数量特别多，出现极长的慢查询时，可以适当调大该参数以加速查询。

4.1.1.343 tidb_streamagg_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 设置 StreamAgg 算子执行查询时的并发度。
- 不推荐设置该变量，修改该变量值可能会造成数据正确性问题。

4.1.1.344 tidb_top_sql_max_meta_count 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：5000
- 范围：[1, 10000]
- 这个变量用于控制Top SQL 每分钟最多收集 SQL 语句类型的数量。

4.1.1.345 tidb_top_sql_max_time_series_count 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：100
- 范围：[1, 5000]
- 这个变量用于控制Top SQL 每分钟保留消耗负载最大的前多少条 SQL（即 Top N) 的数据。

> 注意：
>
> TiDB Dashboard 中的 Top SQL 页面目前只显示消耗负载最多的 5 类 SQL 查询，这与 tidb_top_sql_max_time_series_count 的配置无关。

4.1.1.346 tidb_store_limit 从 v3.0.4 和 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 9223372036854775807]
- 这个变量用于限制TiDB 同时向TiKV 发送的请求的最大数量，0 表示没有限制。

4.1.1.347 tidb_super_read_only 从 v5.3.1 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值: OFF。
- tidb_super_read_only 用于实现对 MySQL 变量 super_read_only 的替代。然而，由于TiDB 是一个分布式数据库，开启 tidb_super_read_only 后数据库各个TiDB 服务器进入只读模式的时刻不是强一致的，而是最终一致的。
- 拥有 SUPER 或 SYSTEM_VARIABLES_ADMIN 权限的用户可以修改该变量。
- 该变量可以控制整个集群的只读状态。开启后（即该值为 ON），整个集群中的 TiDB 服务器都将进入只读状态，只有 SELECT、USE、SHOW 等不会修改数据的语句才能被执行，其他如 INSERT、UPDATE 等语句会被拒绝执行。
- 该变量开启只读模式只保证整个集群最终进入只读模式，当变量修改状态还没被同步到其他TiDB 服务器时，尚未同步的TiDB 仍然停留在非只读模式。
- 在执行 SQL 语句之前，TiDB 会检查集群的只读标志。从 v6.2.0 起，在提交 SQL 语句之前，TiDB 也会检查该标志，从而防止在服务器被置于只读模式后某些长期运行的auto commit 语句可能修改数据的情况。
- 在变量开启时，对于尚未提交的事务：
  - 如果有尚未提交的只读事务，可正常提交该事务。
  - 如果尚未提交的事务为非只读事务，在事务内执行写入的 SQL 语句会被拒绝。
  - 如果尚未提交的事务已经有数据改动，其提交也会被拒绝。
- 当集群开启只读模式后，所有用户（包括 SUPER 用户）都无法执行可能写入数据的 SQL 语句，除非该用户被显式地授予了 RESTRICTED_REPLICA_WRITER_ADMIN 权限。
- 当系统变量tidb_restricted_read_only 为ON 时,tidb_super_read_only 的值会受到tidb_restricted_read_only <—> 的影响。详情请参见tidb_restricted_read_only 中的描述。

4.1.1.348 tidb_sysdate_is_now 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制 SYSDATE 函数能否替换为 NOW 函数，其效果与 MYSQL 中的 sysdate-is-now 一致。

4.1.1.349 tidb_sysproc_scan_concurrency 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 256]
- 这个变量用来设置TiDB 执行内部 SQL 语句（例如统计信息自动更新）时 scan 操作的并发度。

4.1.1.350 tidb_table_cache_lease 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：3
- 范围：[1, 10]
- 单位：秒
- 这个变量用来控制缓存表的 lease 时间，默认值是 3 秒。该变量值的大小会影响缓存表的修改。在缓存表上执行修改操作后，最长可能出现 tidb_table_cache_lease 变量值时长的等待。如果业务表为只读表，或者能接受很高的写入延迟，则可以将该变量值调大，从而增加缓存的有效时间，减少lease 续租的频率。

4.1.1.351 tidb_tmp_table_max_size 从 v5.3 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：67108864
- 范围：[1048576, 137438953472]
- 单位：字节
- 这个变量用于限制单个临时表的最大大小，临时表超出该大小后报错。

4.1.1.352 tidb_track_aggregate_memory_usage

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 本变量控制TiDB 是否跟踪聚合函数的内存使用情况。

> 警告：
>
> 如果禁用该变量，TiDB 可能无法准确跟踪内存使用情况，并且无法控制对应 SQL 语句的内存使用。

4.1.1.353 tidb_tso_client_batch_max_wait_time 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0
- 范围：[0, 10]
- 单位：毫秒
- 这个变量用来设置TiDB 向 PD 请求TSO 时进行一次攒批操作的最大等待时长。默认值为 0，即不进行额外的等待。
- 在向 PD 获取TSO 请求时，TiDB 使用的 PD Client 会一次尽可能多地收集同一时刻的TSO 请求，将其攒批合并成一个RPC 请求后再发送给 PD，从而减轻 PD 的压力。
- 将这个变量值设置为非 0 后，TiDB 会在每一次攒批结束前进行一个最大时长为其值的等待，目的是为了收集到更多的TSO 请求，从而提高攒批效果。
- 适合调高这个变量值的场景：
  - PD leader 因高压力的TSO 请求而达到 CPU 瓶颈，导致TSO RPC 请求的延迟较高。
  - 集群中TiDB 实例的数量不多，但每一台TiDB 实例上的并发量较高。
- 在实际使用中，推荐将该变量尽可能设置为一个较小的值。

> 注意：
>
> 如果 PD leader 的 TSO RPC 延迟升高，但其现象并非由 CPU 使用率达到瓶颈而导致（可能存在网络等问题），此时，调高 tidb_tso_client_batch_max_wait_time 可能会导致TiDB 的语句执行延迟上升，影响集群的 QPS 表现。

4.1.1.354 tidb_ttl_delete_rate_limit 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 9223372036854775807]
- 这个变量用来对每个TiDB 节点的 TTL 删除操作进行限流。其值代表了在 TTL 任务中单个节点每秒允许 DELETE 语句执行的最大次数。当此变量设置为 0 时，则表示不做限制。更多信息，请参考Time to Live。

4.1.1.355 tidb_ttl_delete_batch_size 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[1, 10240]
- 这个变量用于设置 TTL 任务中单个删除事务中允许删除的最大行数。更多信息，请参考Time to Live。

4.1.1.356 tidb_ttl_delete_worker_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 范围：[1, 256]
- 这个变量用于设置每个TiDB 节点上 TTL 删除任务的最大并发数。更多信息，请参考Time to Live。

4.1.1.357 tidb_ttl_job_enable 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：布尔型
- 这个变量用于控制是否启动 TTL 后台清理任务。如果设置为 OFF，所有具有 TTL 属性的表会自动停止对过期数据的清理。更多信息，请参考Time to Live。

4.1.1.358 tidb_ttl_scan_batch_size 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：500
- 范围：[1, 10240]
- 这个变量用于设置 TTL 任务中用来扫描过期数据的每个 SELECT 语句的 LIMIT 的值。更多信息，请参考 Time to Live。

4.1.1.359 tidb_ttl_scan_worker_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 范围：[1, 256]
- 这个变量用于设置每个TiDB 节点 TTL 扫描任务的最大并发数。更多信息，请参考Time to Live。

4.1.1.360 tidb_ttl_job_schedule_window_start_time 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 类型：时间
- 是否持久化到集群：是
- 默认值：00:00 +0000
- 这个变量用于控制 TTL 后台清理任务的调度窗口的起始时间。请谨慎调整此参数，过小的窗口有可能会造成过期数据的清理无法完成。更多信息，请参考Time to Live。

4.1.1.361 tidb_ttl_job_schedule_window_end_time 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 类型：时间
- 是否持久化到集群：是
- 默认值：23:59 +0000
- 这个变量用于控制 TTL 后台清理任务的调度窗口的结束时间。请谨慎调整此参数，过小的窗口有可能会造成过期数据的清理无法完成。更多信息，请参考Time to Live。

4.1.1.362 tidb_ttl_running_tasks 从 v7.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：-1 或 [1, 256]
- 这个变量用于限制整个集群内 TTL 任务的并发量。-1 表示与TiKV 节点的数量相同。更多信息，请参考 Time to Live。

4.1.1.363 tidb_txn_assertion_level 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：FAST
- 可选值：OFF，FAST，STRICT
- 这个变量用于设置assertion 级别。assertion 是一项在事务提交过程中进行的数据索引一致性校验，它对正在写入的 key 是否存在进行检查。如果不符则说明数据索引不一致，会导致事务 abort。详见数据索引一致性报错。
- 对于新创建的 v6.0.0 及以上的集群，默认值为 FAST。对于升级版本的集群，如果升级前是低于 v6.0.0 的版本，升级后默认值为 OFF。
  - OFF: 关闭该检查。
  - FAST: 开启大多数检查项，对性能几乎无影响。
  - STRICT: 开启全部检查项，当系统负载较高时，对悲观事务的性能有较小影响。

4.1.1.364 tidb_txn_commit_batch_size 从 v6.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：16384
- 范围：[1, 1073741824]
- 单位：字节
- 这个变量用于控制 TiDB 向 TiKV 发送的事务提交请求的批量大小。如果业务负载的大部分事务都有大量的写操作，适当调大该变量可以提高批处理的效果。但需要注意的是，设置过大将会超过TiKV的 raft-entry-max-size 限制，导致提交失败。

4.1.1.365 tidb_txn_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：pessimistic
- 可选值：pessimistic，optimistic
- 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式，即只有新创建的集群才会默认使用悲观事务模式。
- 将该变量设置为“optimistic”或“”时，将会使用乐观事务模式。

4.1.1.366 tidb_use_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制是否开启执行计划绑定功能，默认打开，可通过赋值 OFF 来关闭。关于执行计划绑定功能的使用可以参考执行计划绑定文档。

4.1.1.367 tidb_wait_split_region_finish

- 作用域：SESSION
- 类型：布尔型
- 默认值：ON
- 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 ON 代表等待打散完成后再返回结果
  - OFF 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。

4.1.1.368 tidb_wait_split_region_timeout

- 作用域：SESSION

- 类型：整数型
- 默认值：300
- 范围：[1, 2147483647]
- 单位：秒
- 这个变量用来设置 SPLIT REGION 语句的执行超时时间，默认值是 300 秒，如果超时还未完成，就返回一个超时错误。

4.1.1.369 tidb_window_concurrency 从 v4.0 版本开始引入

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用于设置 window 算子的并行度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

4.1.1.370 tiflash_fastscan 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 类型：布尔型
- 如果开启 FastScan 功能（设置为 ON 时），TiFlash 可以提供更高效的查询性能，但不保证查询结果的精度和数据一致性。

4.1.1.371 tiflash_fine_grained_shuffle_batch_size 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：8192
- 范围：[1, 18446744073709551615]
- 细粒度 shuffle 功能开启时，下推到TiFlash 的窗口函数可以并行执行。该变量控制发送端发送数据的攒批大小。
- 对性能影响：如果该值设置过小，例如极端值 1，会导致每个 Block 都进行一次网络传输。如果设置过大，例如极端值整个表的行数，会导致接收端大部分时间都在等待数据，无法流水线计算。可以观察 TiFlash 接收端收到的行数分布情况，如果大部分线程接收的行数很少，例如只有几百行，可以增加该值以达到减少网络开销的目的。

4.1.1.372 tiflash_fine_grained_shuffle_stream_count 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[-1, 1024]
- 当窗口函数下推到TiFlash 执行时，可以通过该变量控制窗口函数执行的并行度。不同取值含义：
  - -1: 表示不使用细粒度 shuffle 功能，下推到TiFlash 的窗口函数以单线程方式执行
  - 0: 表示使用细粒度shuffle 功能。如果tidb_max_tiflash_threads 有效（大于0），则 tiflash_fine_grained_shuffle_str <⟶ 会自动取值为tidb_max_tiflash_threads，否则为默认值 8。最终在TiFlash 上窗口函数的实际并发度为：min(tiflash_fine_grained_shuffle_stream_count，TiFlash 节点物理线程数)
  - 大于 0: 表示使用细粒度 shuffle 功能，下推到TiFlash 的窗口函数会以多线程方式执行，并发度为：min(tiflash_fine_grained_shuffle_stream_count, TiFlash 节点物理线程数)
- 理论上窗口函数的性能会随着该值的增加线性提升。但是如果设置的值超过实际的物理线程数，反而会导致性能下降。

4.1.1.373 time_zone

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：SYSTEM
- 数据库所使用的时区。这个变量值可以写成时区偏移的形式，如‘-8:00’，也可以写成一个命名时区，如‘America/Los_Angeles’。
- 默认值 SYSTEM 表示时区应当与系统主机的时区相同。系统的时区可通过system_time_zone 获取。

4.1.1.374 timestamp

- 作用域：SESSION
- 类型：浮点数
- 默认值：0
- 取值范围：[0, 2147483647]
- 一个 Unix 时间戳。变量值非空时，表示 CURRENT_TIMESTAMP()、NOW() 等函数的时间戳。该变量通常用于数据恢复或数据复制。

4.1.1.375 transaction_isolation

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型

- 默认值：REPEATABLE-READ
- 可选值：READ-UNCOMMITTED，READ-COMMITTED，REPEATABLE-READ，SERIALIZABLE
- 这个变量用于设置事务隔离级别。TiDB 为了兼容 MySQL，支持可重复读 (REPEATABLE-READ)，但实际的隔离级别是快照隔离。详情见事务隔离级别。

4.1.1.376 tx_isolation

这个变量是 transaction_isolation 的别名。

4.1.1.377 tx_isolation_one_shot

> 注意：
> 该变量仅用于TiDB 内部实现，不推荐设置该变量。

在 TiDB 内部实现中，TiDB 解释器会将SET TRANSACTION ISOLATION LEVEL [READ COMMITTED| REPEATABLE READ ⟷ | ...] 语句转化为 SET @@SESSION.TX_ISOLATION_ONE_SHOT = [READ COMMITTED| REPEATABLE READ | ...] ⟷。

4.1.1.378 tx_read_ts

- 作用域：SESSION
- 默认值：“”
- 在 Stale Read 场景下，该会话变量用于帮助记录 Stable Read TS 值。
- 该变量仅用于TiDB 内部实现，不推荐设置该变量。

4.1.1.379 txn_scope

- 作用域：SESSION
- 默认值：global
- 可选值：global 和 local
- 该变量用于设置当前会话下事务为全局事务（设为 global）还是局部事务（设为 local）。
- 该变量仅用于TiDB 内部实现，不推荐设置该变量。

4.1.1.380 validate_password.check_user_name 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：布尔型
- 该变量是密码复杂度策略检查中的一个检查项，用于进行密码与用户名匹配检查。只有validate_password.enable 开启时，该变量才生效。

- 当该变量生效且为 ON 时，如果设置账户密码，TiDB 会将密码与当前会话账户的用户名部分（不包含主机名部分）进行比较，如果匹配则拒绝该密码。
- 该变量独立于validate_password.policy，即不受密码复杂度检测强度的控制。

4.1.1.381 validate_password.dictionary 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 类型：字符串
- 该变量是密码复杂度策略检查中的一个检查项， 用于进行密码与字典字符串匹配检查。只有当 validate_password.enable 开启且validate_password.policy 设置为 2 (STRONG) 时，该变量才生效。
- 该变量是一个长字符串，长度不超过 1024，字符串内容可包含一个或多个在密码中不允许出现的单词，每个单词之间采用英文分号（;）分隔。
- 默认情况下，该变量为空值，不执行字典检查。要进行字典检查，该变量值必须包含待匹配的单词。配置了该变量后，在设置账户密码时，TiDB 会将长度为 4 到 100 的密码的每个子字符串与该变量中配置的单词进行比较。任何匹配都会导致密码被拒绝。比较不区分大小写。

4.1.1.382 validate_password.enable 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 类型：布尔型
- 该变量是密码复杂度策略检查的开关。该变量设置为 ON 后，当设置账户密码时，TiDB 才会进行密码复杂度的各项检查。

4.1.1.383 validate_password.length 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：8
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码的最小长度，默认最小长度为 8。只有validate_password.enable 开启时，该变量才生效。
- 设置该变量时有最小值要求，最小值由其他几个相关的系统变量控制，即该变量的值不能设置为小于此表达式的值：validate_password.number_count + validate_password.special_char_count + (2 * validate_password.mixed_case_count)。
- 当用户修改validate_password.number_count、validate_password.special_char_count、validate_password.mixed_case_count 后导致表达式的值大于 validate_password.length 时，validate_password.length 将自动被修改为满足表达式的最小值。

4.1.1.384 validate_password.mixed_case_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码中至少需要包含多少个大写字符和小写字符。只有当validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。
- 对于给定的 validate_password.mixed_case_count 值，密码中的小写字符数和大写字符数都不能少于该值。例如，值为 1 时，密码中至少需要 1 个小写字母，至少需要 1 个大写字母。

4.1.1.385 validate_password.number_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码中至少需要包含多少个数字字符。只有当validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。

4.1.1.386 validate_password.policy 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：1
- 可选值：[0, 1, 2]
- 该变量是密码复杂度策略检查的强度策略， 该变量影响其他密码复杂度系统变量（前缀为 validate_password） 在密码检查时是否生效， 但是 validate_password.check_user_name 除外。只有validate_password.enable 开启时，该变量才生效。
- 该变量可以使用数值 0、1、2 或相应的符号值 LOW、MEDIUM、STRONG，密码强度策略对应的检查项如下：
  - 0 或者 LOW：检查密码长度。
  - 1 或者 MEDIUM：检查密码长度，检查密码中数字、小写字符、大写字符、特殊字符数量。
  - 2 或者 STRONG：检查密码长度，检查密码中数字、小写字符、大写字符、特殊字符数量，检查密码字典匹配。

4.1.1.387 validate_password.special_char_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是

- 类型：整数型
- 默认值：1
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码中至少需要包含多少个特殊字符。只有当validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。

4.1.1.388 version

- 作用域：NONE
- 默认值：5.7.25-TiDB-(tidb version)
- 这个变量的值是 MySQL 的版本和TiDB 的版本，例如‘5.7.25-TiDB-v4.0.0-beta.2-716-g25e003253’。

4.1.1.389 version_comment

- 作用域：NONE
- 默认值：(string)
- 这个变量的值是TiDB 版本号的其他信息，例如‘TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7 compatible’。

4.1.1.390 version_compile_os

- 作用域：NONE
- 默认值：(string)
- 这个变量值是TiDB 所在操作系统的名称。

4.1.1.391 version_compile_machine

- 作用域：NONE
- 默认值：(string)
- 这个变量值是运行TiDB 的 CPU 架构的名称。

4.1.1.392 wait_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：28800
- 范围：[0, 31536000]
- 单位：秒
- 这个变量表示用户会话的空闲超时。0 代表没有时间限制。

4.1.1.393 warning_count

- 作用域：SESSION
- 默认值：0
- 这个只读变量表示之前执行语句中出现的警告数。

4.1.1.394 windowing_use_high_precision

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于控制计算窗口函数时是否采用高精度模式。

4.2 配置文件参数

4.2.1 TiDB 配置文件描述

TiDB 配置文件比命令行参数支持更多的选项。你可以在config/config.toml.example 找到默认值的配置文件，重命名为 config.toml 即可。本文档只介绍未包含在命令行参数中的参数。

> Tip:
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

4.2.1.0.1 split-table

- 为每个table 建立单独的Region。
- 默认值：true
- 如果需要创建大量的表（例如 10 万张以上），建议将此参数设置为 false。

4.2.1.0.2 tidb-max-reuse-chunk 从 v6.4.0 版本开始引入

- 用于控制每个连接最多缓存的 Chunk 对象数。配置过大会增加 OOM 的风险。
- 默认值：64
- 最小值：0
- 最大值：2147483647

4.2.1.0.3 tidb-max-reuse-column 从 v6.4.0 版本开始引入

- 用于控制每个连接最多缓存的column 对象数。配置过大会增加 OOM 的风险。
- 默认值：256
- 最小值：0
- 最大值：2147483647

4.2.1.0.4 token-limit

- 可以同时执行请求的 session 个数
- 类型：Integer
- 默认值：1000
- 最小值：1
- 最大值（64 位平台）：18446744073709551615
- 最大值（32 位平台）：4294967295

4.2.1.0.5 temp-dir 从 v6.3.0 版本开始引入

- TiDB 用于存放临时数据的路径。如果一个功能需要使用 TiDB 节点的本地存储，TiDB 将把对应数据临时存放在这个目录下。
- 在创建索引的过程中，如果开启了创建索引加速，那么新创建索引需要回填的数据会被先存放在TiDB本地临时存储路径，然后批量导入到TiKV，从而提升索引创建速度。
- 默认值：“/tmp/tidb”

4.2.1.0.6 oom-use-tmp-storage

> **警告：**
>
> 自 v6.3.0 起，该配置项被废弃，其功能由系统变量tidb_enable_tmp_storage_on_oom 代替。集群升级到 v6.3.0 及之后的版本后，会自动继承升级前的 oom-use-tmp-storage 设置，升级后再设置 oom-use-tmp-storage 将不生效。

- 设置是否在单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时为某些算子启用临时磁盘。
- 默认值：true

4.2.1.0.7 tmp-storage-path

- 单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时，某些算子的临时磁盘存储位置。
- 默 认 值：<操作系统临时文件夹>/<操作系统用户ID>_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/<⟷ tmp-storage。 其中 MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA= 是对 <host>:<port>/<statusHost>:<⟷ statusPort> 进行 Base64 编码的输出结果。
- 此配置仅在系统变量tidb_enable_tmp_storage_on_oom 的值为 ON 时有效。

4.2.1.0.8 tmp-storage-quota

- tmp-storage-path 存储使用的限额。
- 单位：Byte

- 当单条 SQL 语句使用临时磁盘，导致TiDB server 的总体临时磁盘总量超过 tmp-storage-quota 时，当前 SQL 操作会被取消，并返回 Out Of Global Storage Quota! 错误。
- 当 tmp-storage-quota 小于 0 时则没有上述检查与限制。
- 默认值：-1
- 当 tmp-storage-path 的剩余可用容量低于 tmp-storage-quota 所定义的值时，TiDB server 启动时将会报出错误并退出。

4.2.1.0.9 lease

- DDL 租约超时时间。
- 默认值：45s
- 单位：秒

4.2.1.0.10 compatible-kill-query

- 设置 KILL 语句的兼容性。
- 默认值：false
- compatible-kill-query 仅在enable-global-kill 为 false 时生效。
- 当enable-global-kill 为 false 时，compatible-kill-query 控制杀死一条查询时是否需要加上 TIDB 关键词。
  - compatible-kill-query 为 false 时，TiDB 中 KILL xxx 的行为和 MySQL 中的行为不同。为杀死一条查询，在TiDB 中需要加上 TIDB 关键词，即 KILL TIDB xxx。
  - compatible-kill-query 为 true 时，为杀死一条查询，在TiDB 中无需加上 TIDB 关键词。强烈不建议设置 compatible-kill-query 为 true，除非你确定客户端将始终连接到同一个 TiDB 节点。这是因为当你在默认的 MySQL 客户端按下 Control+C 时，客户端会开启一个新连接，并在这个新连接中执行 KILL 语句。此时，如果客户端和TiDB 之间存在代理，新连接可能会被路由到其他TiDB 节点，从而错误地终止其他会话。
- 当enable-global-kill 为 true 时，KILL xxx 和 KILL TIDB xxx 的作用相同，但是暂不支持 Control+C 终止查询。
- 关于 KILL 语句的更多信息，请参考KILL [TIDB]。

4.2.1.0.11 check-mb4-value-in-utf8

- 开启检查utf8mb4 字符的开关，如果开启此功能，字符集是utf8，且在utf8 插入 mb4 字符，系统将会报错。
- 默认值：true
- 自 v6.1.0 起，utf8mb4 字符检查改为通过 TiDB 配置项 instance.tidb_check_mb4_value_in_utf8 或系统变量 tidb_check_mb4_value_in_utf8 进行设置。check-mb4-value-in-utf8 仍可使用，但如果同时设置了 check-mb4-value-in-utf8 与 instance.tidb_check_mb4_value_in_utf8，TiDB 将采用 instance.<—> tidb_check_mb4_value_in_utf8 的值。

4.2.1.0.12 treat-old-version-utf8-as-utf8mb4

- 将旧表中的utf8 字符集当成utf8mb4 的开关。
- 默认值：true

4.2.1.0.13 alter-primary-key（已废弃）

- 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。

> 注意：
>
> 该配置项已被废弃，目前仅在 `@@tidb_enable_clustered_index` 取值为 `INT_ONLY` 时生效。如果需要增删主键，请在建表时使用 `NONCLUSTERED` 关键字代替。要了解关于 `CLUSTERED` 主键的详细信息，请参考聚簇索引。

4.2.1.0.14 server-version

- 用来修改TiDB 在以下情况下返回的版本号：
  - 当使用内置函数 `VERSION()` 时。
  - 当与客户端初始连接，TiDB 返回带有服务端版本号的初始握手包时。具体可以查看 MySQL 初始握手包的描述。
- 默认值：“”
- 默认情况下，TiDB 版本号格式为：`5.7.${mysql_latest_minor_version}-TiDB-${tidb_version}`。

4.2.1.0.15 repair-mode

- 用于开启非可信修复模式，启动该模式后，可以过滤 `repair-table-list` 名单中坏表的加载。
- 默认值：false
- 默认情况下，不支持修复语法，默认启动时会加载所有表信息。

4.2.1.0.16 repair-table-list

- 配合 `repair-mode` 为 true 时使用， 用于列出实例中需要修复的坏表的名单， 该名单的写法为 [“db.table1”, “db.table2”, ……]。
- 默认值：[]
- 默认情况下，该list 名单为空，表示没有所需修复的坏表信息。

4.2.1.0.17 new_collations_enabled_on_first_bootstrap

- 用于开启新的collation 支持
- 默认值：true
- 注意：该配置项只有在初次初始化集群时生效，初始化集群后，无法通过更改该配置项打开或关闭新的 collation 框架。

4.2.1.0.18 max-server-connections

- TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认值：0
- 默认情况下，TiDB 不限制客户端连接数。当本配置项的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。
- 自 v6.2.0 起，客户端连接数已改用配置项instance.max_connections 或系统变量max_connections 进行设置。max-server-connections 仍可使用，但如果同时设置了 max-server-connections 与 instance.<—> max_connections，TiDB 将采用 instance.max_connections 的值。

4.2.1.0.19 max-index-length

- 用于设置新建索引的长度限制。
- 默认值：3072
- 单位：Byte
- 目前的合法值范围 [3072，3072*4]。MySQL 和 TiDB v3.0.11 之前版本（不包含 v3.0.11）没有此配置项，不过都对新建索引的长度做了限制。MySQL 对此的长度限制为 3072，TiDB 在 v3.0.7 以及之前版本该值为 3072*4，在 v3.0.7 之后版本（包含 v3.0.8、v3.0.9 和 v3.0.10）的该值为 3072。为了与 MySQL 和 TiDB 之前版本的兼容，添加了此配置项。

4.2.1.0.20 table-column-count-limit 从 v5.0 版本开始引入

- 用于设置单个表中列的数量限制
- 默认值：1017
- 目前的合法值范围 [1017，4096]。

4.2.1.0.21 index-limit 从 v5.0 版本开始引入

- 用于设置单个表中索引的数量限制
- 默认值：64
- 目前的合法值范围 [64，512]。

4.2.1.0.22 enable-telemetry 从 v4.0.2 版本开始引入

- 是否开启TiDB 遥测功能。
- 默认值：false
- 如果在TiDB 实例上该配置项设为 true，该 TiDB 实例上将开启遥测功能，且tidb_enable_telemetry 系统变量生效。
- 如果所有TiDB 实例上该选项都设置为false，那么将完全禁用TiDB 遥测功能，且忽略tidb_enable_telemetry<—> 系统变量。参阅遥测了解该功能详情。

4.2.1.0.23 enable-tcp4-only 从 v5.0 版本开始引入

- 控制是否只监听TCP4。
- 默认值：false
- 当使用LVS 为 TiDB 做负载均衡时，可开启此配置项。这是因为LVS 的 TOA 模块可以通过TCP4 协议从TCP 头部信息中解析出客户端的真实IP。

4.2.1.0.24 enable-enum-length-limit 从 v5.0 版本开始引入

- 是否限制单个 ENUM 元素和单个 SET 元素的最大长度
- 默认值：true
- 当该配置项值为 `true` 时，ENUM 和 SET 单个元素的最大长度为 255 个字符，与 MySQL 8 兼容；当该配置项值为 `false` 时，不对单个元素的长度进行限制，与TiDB v5.0 之前的版本兼容。

4.2.1.0.25 graceful-wait-before-shutdown 从 v5.0 版本开始引入

- 指定关闭服务器时TiDB 等待的秒数，使得客户端有时间断开连接。
- 默认值：0
- 在TiDB 等待服务器关闭期间，HTTP 状态会显示失败，使得负载均衡器可以重新路由流量。

4.2.1.0.26 enable-global-kill 从 v6.1.0 版本开始引入

- 用于开启Global Kill（跨节点终止查询或连接）功能。
- 默认值：true
- 当该配置项值为 `true` 时，`KILL` 语句和 `KILL TIDB` 语句均能跨节点终止查询或连接，无需担心错误地终止其他查询或连接。当你使用客户端连接到任何一个TiDB 节点执行 `KILL` 语句或 `KILL TIDB` 语句时，该语句会被转发给对应的TiDB 节点。当客户端和TiDB 中间有代理时，`KILL` 语句或 `KILL TIDB` 语句也会被转发给对应的TiDB 节点执行。目前暂时不支持在 `enable-global-kill` 为 `true` 时用 MySQL 命令行ctrl+c 终止查询或连接。关于 `KILL` 语句的更多信息，请参考KILL [TIDB]。

4.2.1.0.27 initialize-sql-file 从 v6.6.0 版本开始引入

- 用于指定TiDB 集群初次启动时执行的 SQL 脚本。
- 默认值：“”
- 脚本中的所有 SQL 语句将以最高权限执行，不会进行权限检查。若指定的 SQL 脚本无法正确执行，可能导 致TiDB 集群启动失败。
- 通常用于修改系统变量的值、创建用户或分配权限等。

4.2.1.0.28 enable-forwarding 从 v5.0.0 版本开始引入

- 控制TiDB 中的 PD client 以及TiKV client 在疑似网络隔离的情况下是否通过 follower 将请求转发给leader。
- 默认值：false
- 如果确认环境存在网络隔离的可能，开启这个参数可以减少服务不可用的窗口期。
- 如果无法准确判断隔离、网络中断、宕机等情况，这个机制存在误判情况从而导致可用性、性能降低。如果网络中从未发生过网络故障，不推荐开启此选项。

4.2.1.0.29 enable-table-lock 从 v4.0.0 版本开始引入

> **警告：**
>
> 表级锁(Table Lock) 为实验特性，不建议在生产环境中使用。

- 控制是否开启表级锁特性。
- 默认值：false
- 表级锁用于协调多个 session 之间对同一张表的并发访问。目前已支持的锁种类包括 READ、WRITE 和 WRITE LOCAL。当该配置项为 false 时，执行 LOCK TABLES 和 UNLOCK TABLES 语句不会生效，并且会报“LOCK/UNLOCK TABLES is not supported”的警告。更多信息，请参考LOCK TABLES 和 UNLOCK TABLES。

4.2.1.0.30 labels

- 指定服务器标签，例如{ zone = "us-west-1", dc = "dc1", rack = "rack1", host = "tidb1" }。
- 默认值：{}

> **注意：**
>
> - 标签 zone 在 TiDB 中具有特殊用途，用于指定服务器所在的区域信息，当设置 zone 为非空值时，对应的值会被自动用于txn-score 和 Follower read 等功能。
> - 标签 group 在 TiDB Operator 中具有特殊用途。对于使用TiDB Operator 部署的集群，建议不要手动指定此标签。

4.2.1.1 log

日志相关的配置项。

4.2.1.1.1 level

- 指定日志的输出级别，可选项为 [debug, info, warn, error, fatal]
- 默认值：“info”

4.2.1.1.2 format

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

4.2.1.1.3 enable-timestamp

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

> 注意：
>
> - 考虑后向兼容性， 原来的配置项 disable-timestamp 仍然有效， 但如果和 enable-timestamp 配置的值在语义上冲突（例如在配置中把 enable-timestamp 和 disable-timestamp 同时设置为 true），则 TiDB 会忽略 disable-timestamp 的值。
> - 当前TiDB 默认使用 disable-timestamp 来决定是否在日志中输出时间戳，此时 enable-timestamp 的值为 null。
> - 在未来的版本中，disable-timestamp 配置项将被彻底移除，请废弃 disable-timestamp 的用法，使用语义上更易于理解的 enable-timestamp。

4.2.1.1.4 enable-slow-log

- 是否开启慢查询日志
- 默认值：true
- 可以设置成 true 或 false 来启用或禁用慢查询日志。
- 自 v6.1.0 起，已改用配置项 instance.tidb_enable_slow_log 或系统变量 tidb_enable_slow_log 来设置是否开启慢查询日志。enable-slow-log 仍可使用，但如果同时设置了 enable-slow-log 与 instance.tidb_enable_slow_log，TiDB 将采用 instance.tidb_enable_slow_log 的值。

4.2.1.1.5 slow-query-file

- 慢查询日志的文件名。
- 默认值：“tidb-slow.log”。注：由于TiDB V2.1.8 更新了慢日志格式，所以将慢日志单独输出到了慢日志文件。V2.1.8 之前的版本，该变量的默认值是“”。
- 设置后，慢查询日志会单独输出到该文件。

4.2.1.1.6 slow-threshold

- 输出慢日志的耗时阈值。
- 默认值：300
- 单位：毫秒
- 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。
- 自 v6.1.0 起，已改用配置项 instance.tidb_slow_log_threshold 或系统变量 tidb_slow_log_threshold 来设置输出慢日志的耗时阈值。slow-threshold 仍可使用，但如果同时设置了 slow-threshold 与 instance.tidb_slow_log_threshold，TiDB 将采用 instance.tidb_slow_log_threshold 的值。

4.2.1.1.7 record-plan-in-slow-log

- 在慢日志中记录执行计划
- 默认值：1
- 自v6.1.0 起，已改用配置项instance.tidb_record_plan_in_slow_log 或系统变量tidb_record_plan_in_slow_log <⟷ 来设置在慢日志中记录执行计划。record-plan-in-slow-log 仍可使用， 但如果同时设置了 record-plan-in-slow-log 与 instance.tidb_record_plan_in_slow_log，TiDB 将采用 instance. <⟷ tidb_record_plan_in_slow_log 的值。

4.2.1.1.8 expensive-threshold

- 输出 expensive 操作的行数阈值。
- 默认值：10000
- 当查询的行数（包括中间结果，基于统计信息）大于这个值，该操作会被认为是 expensive 查询，并输出一个前缀带有 [EXPENSIVE_QUERY] 的日志。

4.2.1.1.9 timeout 从 v7.1.0 版本开始引入

- 用于设置TiDB 写日志操作的超时时间。当磁盘故障导致日志无法写入时，该配置可以让TiDB 进程崩溃而不是卡死。
- 默认值：0，表示不设置超时
- 单位：秒
- 在某些用户场景中，TiDB 日志可能是保存在热插拔盘或网络挂载盘上，这些磁盘可能会永久丢失。在这种场景下，TiDB 无法自动恢复，写日志操作会永久阻塞。尽管TiDB 进程看起来仍在运行，但不会响应任何请求。该配置项用于处理这样的场景。

4.2.1.2 log.file

日志文件相关的配置项。

filename

- 一般日志文件名字。
- 默认值：“”
- 如果设置，会输出一般日志到这个文件。

max-size

- 日志文件的大小限制。
- 默认值：300
- 单位：MB
- 最大设置上限为 4096。

max-days

- 日志最大保留的天数。
- 默认值：0
- 默认不清理；如果设置了参数值，在 max-days 之后TiDB 会清理过期的日志文件。

max-backups

- 保留的日志的最大数量。
- 默认值：0
- 默认全部保存；如果设置为 7，会最多保留 7 个老的日志文件。

4.2.1.3 security

安全相关配置。

4.2.1.3.1 enable-sem

- 启用安全增强模式(SEM)。
- 默认值：false
- 可以通过系统变量tidb_enable_enhanced_security 获取安全增强模式的状态。

4.2.1.3.2 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。

4.2.1.3.3 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。

4.2.1.3.4 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。

4.2.1.3.5 cluster-ssl-ca

- CA 根证书，用于用tls 连接TiKV/PD
- 默认值：“”

4.2.1.3.6 cluster-ssl-cert

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

4.2.1.3.7 cluster-ssl-key

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

4.2.1.3.8 spilled-file-encryption-method

- 内存落盘文件的加密方式。
- 默认值：`"plaintext"`，表示不进行加密。
- 可选值：`"plaintext"`、`"aes128-ctr"`。

4.2.1.3.9 auto-tls

- 控制TiDB 启动时是否自动生成 TLS 证书。
- 默认值：`false`

4.2.1.3.10 tls-version

- 设置用于连接 MySQL 协议的最低 TLS 版本。
- 默认值： “” ，支持 TLSv1.1 及以上版本。
- 可选值：`"TLSv1.0"`、`"TLSv1.1"`、`"TLSv1.2"` 和 `"TLSv1.3"`

4.2.1.3.11 auth-token-jwks 从 v6.4.0 版本开始引入

> **警告：**
>
> `tidb_auth_token` 认证方式仅用于TiDB Cloud 内部实现，不要修改该配置。

- 设置 `tidb_auth_token` 认证方式的JSON Web Key Sets (JWKS) 的本地文件路径。
- 默认值： “”

4.2.1.3.12 auth-token-refresh-interval 从 v6.4.0 版本开始引入

> **警告：**
>
> tidb_auth_token 认证方式仅用于TiDB Cloud 内部实现，不要修改该配置。

- 设置 tidb_auth_token 认证方式的JWKS 刷新时间间隔。
- 默认值：1h

4.2.1.3.13 disconnect-on-expired-password 从 v6.5.0 版本开始引入

- 对于密码已过期的用户，通过 disconnect-on-expired-password 控制TiDB 服务端是否直接断开该用户的连接。
- 默认值：true
- 默认值为“true”表示TiDB 服务端将直接断开密码已过期用户的连接。设置为“false”时，TiDB 服务端将密码已过期用户的连接置于“沙盒模式”，允许该用户建立连接并执行密码重置操作。

4.2.1.3.14 session-token-signing-cert 从 v6.4.0 版本开始引入

> **警告：**
>
> 该配置与一个未发布的特性相关。请勿设置该配置。

- 默认值：“”

4.2.1.3.15 session-token-signing-key 从 v6.4.0 版本开始引入

> **警告：**
>
> 该配置与一个未发布的特性相关。请勿设置该配置。

- 默认值：“”

4.2.1.4 performance

性能相关配置。

4.2.1.4.1 max-procs

- TiDB 的 CPU 使用数量。
- 默认值：0
- 默认值为 0 表示使用机器上所有的 CPU；如果设置成 n，那么 TiDB 会使用 n 个 CPU 数量。

4.2.1.4.2 server-memory-quota 从 v4.0.9 版本开始引入

> 警告：
> 自 v6.5.0 起，该配置项被废弃。请使用 tidb_server_memory_limit 系统变量进行设置。

- 设置 tidb-server 实例的最大内存用量，单位为字节。
- 默认值：0
- 默认值为 0 表示无内存限制。

4.2.1.4.3 txn-entry-size-limit 从 v5.0 版本开始引入

- TiDB 单行数据的大小限制
- 默认值：6291456
- 单位：Byte
- 事务中单个 key-value 记录的大小限制。若超出该限制，TiDB 将会返回 entry too large 错误。该配置项的最大值不超过 125829120（表示 120MB）。
- 注意，TiKV 有类似的限制。若单个写入请求的数据量大小超出 raft-entry-max-size，默认为 8MB，TiKV 会拒绝处理该请求。当表的一行记录较大时，需要同时修改这两个配置。
- max_allowed_packet (MySQL 协议的最大数据包大小) 的默认值为 67108864（64 MiB）。如果一行记录的大小超过 max_allowed_packet，该行记录会被截断。
- txn-total-size-limit（TiDB 单个事务大小限制）的默认值为 100 MiB。如果将 txn-entry-size-limit 的值设置为 100 MiB 以上，需要相应地调大 txn-total-size-limit 的值。

4.2.1.4.4 txn-total-size-limit

- TiDB 单个事务大小限制
- 默认值：104857600
- 单位：Byte
- 单个事务中，所有 key-value 记录的总大小不能超过该限制。该配置项的最大值不超过 1099511627776（表示 1TB）。注意，如果使用了以 Kafka 为下游消费者的 binlog，如：arbiter 集群，该配置项的值不能超过 1073741824（表示 1GB），因为这是 Kafka 的处理单条消息的最大限制，超过该限制 Kafka 将会报错。
- 在 v6.5.0 及之后的版本中，不再推荐使用该配置项，事务的内存大小会被累计计入所在会话的内存使用量中，并由 tidb_mem_quota_query 变量在单个会话内存超阈值时采取控制行为。为了向前兼容，由低版本升级至 v6.5.0 及更高版本时，该配置项的行为如下所述:

  - 若该配置项未设置，或设置为默认值(104857600)，升级后事务内存大小将会计入所在会话的内存使用中，由 tidb_mem_quota_query 变量控制。
  - 若该配置项未设为默认值(104857600)，升级前后该配置项仍生效，对单个事务大小的限制行为不会发生变化，事务内存大小不由 tidb_mem_quota_query 控制。

4.2.1.4.5 max-txn-ttl

- 单个事务持锁的最长时间，超过该时间，该事务的锁可能会被其他事务清除，导致该事务无法成功提交。
- 默认值：3600000
- 单位：毫秒
- 超过此时间的事务只能执行提交或者回滚，提交不一定能够成功。

4.2.1.4.6 stmt-count-limit

- TiDB 单个事务允许的最大语句条数限制。
- 默认值：5000
- 在一个事务中，超过 stmt-count-limit 条语句后还没有rollback 或者 commit，TiDB 将会返回 statement count 5001 exceeds the transaction limitation, autocommit = false 错误。该限制只在可重试的乐观事务中生效，如果使用悲观事务或者关闭了事务重试，事务中的语句数将不受此限制。

4.2.1.4.7 tcp-keep-alive

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

4.2.1.4.8 tcp-no-delay

- 控制TiDB 是否在TCP 层开启TCP_NODELAY。开启后，TiDB 将禁用TCP/IP 协议中的 Nagle 算法，允许小数据包的发送，可以降低网络延时，适用于延时敏感型且数据传输量比较小的应用。
- 默认值：true

4.2.1.4.9 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

4.2.1.4.10 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

4.2.1.4.11 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：0
- 最大值：1

4.2.1.4.12 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： 默认值 NO_PRIORITY 表示不强制改变执行语句的优先级， 其它优先级从低到高可设置为 LOW_PRIORITY、 DELAYED 或 HIGH_PRIORITY。
- 自 v6.1.0 起，已改用配置项instance.tidb_force_priority 或系统变量tidb_force_priority 来将所有语句优先级设为 force-priority 的值。force-priority 仍可使用，但如果同时设置了 force-priority 与 instance.tidb_force_priority，TiDB 将采用 instance.tidb_force_priority 的值。

4.2.1.4.13 distinct-agg-push-down

- 设置优化器是否执行将带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。
- 默认值：false
- 该变量作为系统变量tidb_opt_distinct_agg_push_down 的初始值。

4.2.1.4.14 enforce-mpp

- 用于控制是否忽略优化器代价估算，强制使用TiFlash 的 MPP 模式执行查询。
- 默认值：false
- 该配置项可以控制系统变量 tidb_enforce_mpp 的初始值。 例如， 当设置该配置项为 true 时，tidb_enforce_mpp 的默认值为 ON。

4.2.1.4.15 stats-load-concurrency 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载功能目前为实验性特性，不建议在生产环境中使用。

- TiDB 统计信息同步加载功能可以并发处理的最大列数
- 默认值：5
- 目前的合法值范围：[1, 128]

4.2.1.4.16 stats-load-queue-size 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载功能目前为实验性特性，不建议在生产环境中使用。

- 用于设置TiDB 统计信息同步加载功能最多可以缓存多少列的请求
- 默认值：1000
- 目前的合法值范围：[1, 100000]

4.2.1.4.17 enable-stats-cache-mem-quota 从 v6.1.0 版本开始引入

> **警告：**
>
> 该变量为实验特性，不推荐在生产环境中使用。

- 用于控制TiDB 是否开启统计信息缓存的内存上限。
- 默认值：false

4.2.1.4.18 lite-init-stats 从 v7.1.0 版本开始引入

> **警告：**
>
> 该变量控制的功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 用于控制TiDB 启动时是否采用轻量级的统计信息初始化。
- 默认值：false
- 当 `lite-init-stats` 为 `true` 时，统计信息初始化时列和索引的直方图、TopN、Count-Min Sketch 均不会加载到内存中。当 `lite-init-stats` 为 `false` 时，统计信息初始化时索引和主键的直方图、TopN、Count-Min Sketch 会被加载到内存中，非主键列的直方图、TopN、Count-Min Sketch 不会加载到内存中。当优化器需要某一索引或者列的直方图、TopN、Count-Min Sketch 时，这些统计信息会被同步或异步加载到内存中（由`tidb_stats_load_sync_wait` 控制）。
- 将 `lite-init-stats` 设置为true，可以加速统计信息初始化，避免加载不必要的统计信息，从而降低TiDB 的内存使用。详情请参考统计信息的加载。

4.2.1.4.19 `force-init-stats` 从 v7.1.0 版本开始引入

- 用于控制TiDB 启动时是否在统计信息初始化完成后再对外提供服务。
- 默认值：false
- 当 `force-init-stats` 为 `true` 时，TiDB 启动时会等到统计信息初始化完成后再对外提供服务。在表和分区数量较多的情况下，将 `force-init-stats` 设置为 `true` 可能会导致TiDB 从启动到开始对外提供服务的时间变长。
- 当 `force-init-stats` 为 `false` 时，TiDB 在统计信息初始化未完成时即可对外提供服务，但由于统计信息初始化未完成，优化器会用pseudo 统计信息进行决策，可能会产生不合理的执行计划。

4.2.1.5 opentracing

opentracing 的相关的设置。

4.2.1.5.1 `enable`

- 开启 opentracing 跟踪TiDB 部分组件的调用开销。注意开启后会有一定的性能损失。
- 默认值：false

4.2.1.5.2 `rpc-metrics`

- 开启rpc metrics。
- 默认值：false

4.2.1.6 opentracing.sampler

opentracing.sampler 相关的设置。

4.2.1.6.1 `type`

- opentracing 采样器的类型。字符串取值大小写不敏感。
- 默认值：“const”
- 可选值：“const”，“probabilistic”，“ratelimiting”，remote”

4.2.1.6.2 param

- 采样器参数。
  - 对于const类型，可选值为 0 或 1，表示是否开启。
  - 对于probabilistic类型，参数为采样概率，可选值为 0 到 1 之间的浮点数。
  - 对于ratelimiting类型，参数为每秒采样 span 的个数。
  - 对于remote类型，参数为采样概率，可选值为 0 到 1 之间的浮点数。
- 默认值：1.0

4.2.1.6.3 sampling-server-url

- jaeger-agent 采样服务器的 HTTP URL 地址。
- 默认值：“”

4.2.1.6.4 max-operations

- 采样器可追踪的最大操作数。如果一个操作没有被追踪，会启用默认的probabilistic 采样器。
- 默认值：0

4.2.1.6.5 sampling-refresh-interval

- 控制远程轮询jaeger-agent 采样策略的频率。
- 默认值：0

4.2.1.7 opentracing.reporter

opentracing.reporter 相关的设置。

4.2.1.7.1 queue-size

- reporter 在内存中记录 spans 个数的队列容量。
- 默认值：0

4.2.1.7.2 buffer-flush-interval

- reporter 缓冲区的刷新频率。
- 默认值：0

4.2.1.7.3 log-spans

- 是否为所有提交的 span 打印日志。
- 默认值：false

4.2.1.7.4 local-agent-host-port

- reporter 向 jaeger-agent 发送 span 的地址。
- 默认值：“”

4.2.1.8 tikv-client

4.2.1.8.1 grpc-connection-count

- 跟每个TiKV 之间建立的最大连接数。
- 默认值：4

4.2.1.8.2 grpc-keepalive-time

- TiDB 与 TiKV 节点之间rpc 连接 keepalive 时间间隔，如果超过该值没有网络包，grpc client 会 ping 一下 TiKV 查看是否存活。
- 默认值：10
- 单位：秒

4.2.1.8.3 grpc-keepalive-timeout

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

4.2.1.8.4 grpc-compression-type

- 控制TiDB 向 TiKV 节点传输数据使用的压缩算法类型。默认值为“none”即不压缩。修改为“gzip”可以使用 gzip 算法压缩数据。
- 默认值：“none”
- 可选值：“none”，“gzip”

4.2.1.8.5 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。

4.2.1.8.6 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

4.2.1.8.7 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

4.2.1.8.8 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。

4.2.1.8.9 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

4.2.1.9 tikv-client.copr-cache 从 v4.0.0 版本开始引入

本部分介绍 Coprocessor Cache 相关的配置项。

4.2.1.9.1 capacity-mb

- 缓存的总数据量大小。当缓存空间满时，旧缓存条目将被逐出。值为 0.0 时表示关闭 Coprocessor Cache。
- 默认值：1000.0
- 单位：MB
- 类型：Float

4.2.1.10 binlog

TiDB Binlog 相关配置。

4.2.1.10.1 enable

- binlog 开关。
- 默认值：false

4.2.1.10.2 write-timeout

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

4.2.1.10.3 ignore-error

- 忽略写binlog 发生的错误时处理开关，推荐不修改该值。
- 默认值：false
- 如果设置为true，发生错误时，TiDB 会停止写入binlog，并且在监控项tidb_server_critical_error_total 上计数加 1；如果设置为 false，写入 binlog 失败，会停止整个TiDB 的服务。

4.2.1.10.4 binlog-socket

- binlog 输出网络地址。
- 默认值：“”

4.2.1.10.5 strategy

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

4.2.1.11 status

TiDB 服务状态相关配置。

4.2.1.11.1 report-status

- 开启 HTTP API 服务的开关。
- 默认值：true

4.2.1.11.2 record-db-qps

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

4.2.1.12 pessimistic-txn

悲观事务使用方法请参考TiDB 悲观事务模式。

4.2.1.12.1 max-retry-count

- 悲观事务中单个语句最大重试次数，重试次数超过该限制，语句执行将会报错。
- 默认值：256

4.2.1.12.2 deadlock-history-capacity

- 单个TiDB 节点的INFORMATION_SCHEMA.DEADLOCKS 表最多可记录的死锁事件个数。当表的容量已满时，如果再次发生死锁错误，最早的一次死锁错误的信息将从表中移除。
- 默认值：10
- 最小值：0
- 最大值：10000

4.2.1.12.3 deadlock-history-collect-retryable

- 控制INFORMATION_SCHEMA.DEADLOCKS 表中是否收集可重试的死锁错误信息。详见 DEADLOCKS 表文档的可重试的死锁错误小节。
- 默认值：false

4.2.1.12.4 pessimistic-auto-commit

- 用来控制开启全局悲观事务模式下(tidb_txn_mode='pessimistic') 时，自动提交的事务使用的事务模式。默认情况下，即使开启全局悲观事务模式，自动提交事务依然使用乐观事务模式来执行。当开启该配置项后（设置为 true），在全局悲观事务模式下，自动提交事务将也使用悲观事务模式执行。行为与其他显式提交的悲观事务相同。
- 对于存在冲突的场景，开启本开关可以将自动提交事务纳入全局等锁管理中，从而避免死锁，改善冲突造成死锁带来的时延尖刺。
- 对于不存在冲突的场景，如果有大量自动提交事务（例如自动提交事务数量占业务数量的比例超过一半甚至更多，需要根据实际情况分析）且单个事务操作数据量较大的情况下，开启该配置项会造成性能回退。例如，自动提交的 INSERT INTO SELECT 语句。
- 默认值：false

4.2.1.12.5 constraint-check-in-place-pessimistic 从 v6.4.0 版本开始引入

- 用来控制系统变量tidb_constraint_check_in_place_pessimistic 的默认值。
- 默认值：true

4.2.1.13 isolation-read

读取隔离相关的配置项。

4.2.1.13.1 engines

- 用于控制TiDB 节点允许从哪种类型的引擎读取数据。
- 默认值：[“tikv”，“tiflash”，“tidb”]，表示由优化器自动选择存储引擎。
- 可选值：“tikv”，“tiflash”，“tidb”的组合，如：[“tikv”，“tidb”]、[“tiflash”，“tidb”]。

4.2.1.14 instance

4.2.1.14.1 tidb_enable_collect_execution_info

- 用于控制是否同时将各个执行算子的执行信息记录入 slow query log 中。
- 默认值：true
- 在 v6.1.0 之前，该功能通过配置项 `enable-collect-execution-info` 进行设置。

4.2.1.14.2 tidb_enable_slow_log

- 是否开启慢查询日志。
- 默认值：true
- 可以设置成 `true` 或 `false` 来启用或禁用慢查询日志。
- 在 v6.1.0 之前，该功能通过配置项 `enable-slow-log` 进行设置。

4.2.1.14.3 tidb_slow_log_threshold

- 输出慢日志的耗时阈值。
- 默认值：300
- 范围：`[-1, 9223372036854775807]`
- 单位：毫秒
- 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。
- 在 v6.1.0 之前，该功能通过配置项 `slow-threshold` 进行设置。

4.2.1.14.4 tidb_record_plan_in_slow_log

- 在慢日志中记录执行计划。
- 默认值：1
- 0 表示关闭，1 表示开启，默认开启，该值作为系统变量tidb_record_plan_in_slow_log 的初始值。
- 在 v6.1.0 之前，该功能通过配置项 `record-plan-in-slow-log` 进行设置。

4.2.1.14.5 tidb_force_priority

- 把所有的语句优先级设置为系统变量 `tidb_force_priority` 的值。
- 默认值：NO_PRIORITY
- 默认值 NO_PRIORITY 表示不强制改变执行语句的优先级，其它优先级从低到高可设置为 LOW_PRIORITY、DELAYED 或 HIGH_PRIORITY。
- 在 v6.1.0 之前，该功能通过配置项 `force-priority` 进行设置。

4.2.1.14.6 max_connections

- TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认值：0
- 取值范围：[0, 100000]
- 默认情况下，TiDB 不限制客户端连接数。当本配置项的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。
- 该值作为系统变量max_connections 的初始值。
- 在 v6.2.0 之前，该功能通过配置项 max-server-connections 进行设置。

4.2.1.14.7 tidb_enable_ddl

- 用于表示该tidb-server 是否可以成为 DDL owner。
- 默认值：true
- 该值作为系统变量tidb_enable_ddl 的初始值。
- 在 v6.3.0 之前，该功能由配置项 run-ddl 进行设置。

4.2.1.14.8 tidb_stmt_summary_enable_persistent 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 用于控制是否开启 statements summary 持久化。
- 默认值：false
- 详情参考持久化 statements summary。

4.2.1.14.9 tidb_stmt_summary_filename 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于指定持久化数据所写入的文件。
- 默认值：“tidb-statements.log”

4.2.1.14.10 tidb_stmt_summary_file_max_days 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于指定持久化数据文件所保留的最大天数。
- 默认值：3
- 单位：天
- 可结合数据保留时长需求与磁盘空间占用适当调整。

4.2.1.14.11 tidb_stmt_summary_file_max_size 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于限制持久化数据单个文件的大小。
- 默认值：64
- 单位：MiB
- 可结合数据保留时长需求与磁盘空间占用适当调整。

4.2.1.14.12 tidb_stmt_summary_file_max_backups 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于限制持久化数据文件最大数量，0 表示不限制。
- 默认值：0
- 可结合数据保留时长需求与磁盘空间占用适当调整。

4.2.1.15 proxy-protocol

PROXY 协议相关的配置项。

4.2.1.15.1 networks

- 允许使用 PROXY 协议连接TiDB 的代理服务器地址列表。
- 默认值：“”
- 通常情况下，通过反向代理使用TiDB 时，TiDB 会将反向代理服务器的IP 地址视为客户端IP 地址。对于支持 PROXY 协议的反向代理（如 HAProxy），开启 PROXY 协议后能让反向代理透传客户端真实的IP 地址给TiDB。
- 配置该参数后，TiDB 将允许配置的源IP 地址使用 PROXY 协议连接到TiDB，且拒绝这些源IP 地址使用非 PROXY 协议连接。若该参数为空，则任何源IP 地址都不能使用 PROXY 协议连接到TiDB。地址可以使用IP 地址格式(192.168.1.50) 或者CIDR 格式(192.168.1.0/24)，并可用 , 分隔多个地址，或用* 代表所有IP 地址。

> **警告：**
>
> 需谨慎使用* 符号，因为* 允许来自任何IP 的客户端自行汇报其IP 地址，从而可能引入安全风险。另外，* 可能导致部分直接连接TiDB 的内部组件无法使用，例如TiDB Dashboard。

4.2.1.15.2 fallbackable 从 v6.5.1 版本开始引入

- 用于控制是否启用 PROXY 协议回退模式。如果设置为 true，TiDB 可以接受属于 proxy-protocol.networks 的客户端使用非 PROXY 协议规范或者没有发送 PROXY 协议头的客户端连接。默认情况下，TiDB 仅接受属于 proxy-protocol.networks 的客户端发送 PROXY 协议头的客户端连接。
- 默认：false

4.2.1.16 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。

4.2.1.16.1 allow-expression-index 从 v4.0.0 版本开始引入

- 用于控制是否能创建表达式索引。自 v5.2.0 版本起，如果表达式中的函数是安全的，你可以直接基于该函数创建表达式索引，不需要打开该配置项。如果要创建基于其他函数的表达式索引，可以打开该配置项，但可能存在正确性问题。通过查询 tidb_allow_function_for_expression_index 变量可得到能直接用于创建表达式的安全函数。
- 默认值：false

4.2.2 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。

Tip:

如果你需要调整配置项的值，请参考修改配置参数进行操作。

4.2.2.1 全局配置

4.2.2.1.1 abort-on-panic

- 设置TiKV panic 时是否调用 `abort()` 退出进程。此选项影响TiKV 是否允许系统生成core dump 文件。
  - 如果此配置项值为 false，当 TiKV panic 时，TiKV 调用 `exit()` 退出进程。
  - 如果此配置项值为true，当TiKV panic 时，TiKV 调用 `abort()` 退出进程。此时TiKV 允许系统在退出时生成 core dump 文件。要生成core dump 文件，你还需要进行core dump 相关的系统配置（比如打开 `ulimit -c` 和配置core dump 路径，不同操作系统配置方式不同）。建议将 core dump 生成路径设置在 TiKV 数据的不同磁盘分区，避免core dump 文件占用磁盘空间过大，造成TiKV 磁盘空间不足。
- 默认值：false

4.2.2.1.2 slow-log-file

- 存储慢日志的文件。
- 如果未设置本项但设置了 `log.file.filename`，慢日志将输出至 `log.file.filename` 指定的日志文件中。
- 如果本项和 `log.file.filename` 均未设置，所有日志默认输出到 `"stderr"`。
- 如果同时设置了两项，普通日志会输出至 `log.file.filename` 指定的日志文件中，而慢日志则会输出至本配置项指定的日志文件中。
- 默认值：“”

4.2.2.1.3 slow-log-threshold

- 输出慢日志的阈值。处理时间超过该阈值后会输出慢日志。
- 默认值：“1s”

4.2.2.1.4 memory-usage-limit

- TiKV 实例的内存使用限制。当TiKV 的内存使用量接近此阈值时，内部缓存会被清除以释放内存。
- 在大多数情况下，TiKV 实例被设置为占系统可用总内存的 75%，因此你不需要显式指定此配置项。剩余 25% 的内存用于操作系统的页缓存，详情参见storage.block-cache.capacity。
- 在单个物理机上部署多个 TiKV 节点时，你也不需要设置此配置项。在这种情况下，TiKV 实例使用 `5/3 * block-cache.capacity` 的内存。
- 不同系统内存容量的默认值如下：

- system=8G block-cache=3.6G memory-usage-limit=6G page-cache=2G
- system=16G block-cache=7.2G memory-usage-limit=12G page-cache=4G
- system=32G block-cache=14.4G memory-usage-limit=24G page-cache=8G

4.2.2.2 log 从 v5.4.0 版本开始引入

日志相关的配置项。

自v5.4.0 版本起，废弃原log 参数`log-rotation-timespan`，并将`log-level`、`log-format`、`log-file`、`log-rotation-size` 变更为下列参数，与TiDB 的log 参数保持一致。如果只设置了原参数、且把其值设为非默认值，原参数与新参数会保持兼容；如果同时设置了原参数和新参数，则会使用新参数。

4.2.2.2.1 `level` 从 v5.4.0 版本开始引入

- 日志等级。
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”
- 默认值：“info”

4.2.2.2.2 `format` 从 v5.4.0 版本开始引入

- 日志的格式。
- 可选值：“json”，“text”
- 默认值：“text”

4.2.2.2.3 `enable-timestamp` 从 v5.4.0 版本开始引入

- 是否开启日志中的时间戳。
- 可选值：“true”，“false”
- 默认值：“true”

4.2.2.3 log.file 从 v5.4.0 版本开始引入

日志文件相关的配置项。

4.2.2.3.1 `filename` 从 v5.4.0 版本开始引入

- log 文件。如果未设置该参数，日志会默认输出到 `"stderr"`；如果设置了该参数，log 会输出到对应的文件中。
- 默认值：“”

4.2.2.3.2 `max-size` 从 v5.4.0 版本开始引入

- 单个log 文件最大大小，超过设定的参数值后，系统自动切分成多个文件。
- 默认值：300
- 最大值：4096
- 单位：MiB

4.2.2.3.3 max-days 从 v5.4.0 版本开始引入

- 保留 log 文件的最长天数。
  - 如果未设置本参数或把此参数设置为默认值 0，TiKV 不清理log 文件。
  - 如果把此参数设置为非 0 的值，在 max-days 之后，TiKV 会清理过期的日志文件。
- 默认值：0

4.2.2.3.4 max-backups 从 v5.4.0 版本开始引入

- 可保留的log 文件的最大数量。
  - 如果未设置本参数或把此参数设置为默认值 0，TiKV 会保存所有的log 文件；
  - 如果把此参数设置为非 0 的值，TiKV 最多会保留 max-backups 中指定的数量的旧日志文件。比如，如果该值设置为 7，TiKV 最多会保留 7 个旧的日志文件。
- 默认值：0

4.2.2.3.5 pd.enable-forwarding 从 v5.0.0 版本开始引入

- 控制TiKV 中的 PD client 在疑似网络隔离的情况下是否通过 follower 将请求转发给leader。
- 默认值：false
- 如果确认环境存在网络隔离的可能，开启这个参数可以减少服务不可用的窗口期。
- 如果无法准确判断隔离、网络中断、宕机等情况，这个机制存在误判情况从而导致可用性、性能降低。如果网络中从未发生过网络故障，不推荐开启此选项。

4.2.2.4 server

服务器相关的配置项。

4.2.2.4.1 addr

- 服务器监听的IP 地址和端口号。
- 默认值："127.0.0.1:20160"

4.2.2.4.2 advertise-addr

- 用于客户端通信的对外访问地址。
- 如果没有设置该配置项，则使用 addr 的值。
- 默认值：""

4.2.2.4.3 status-addr

- 通过 HTTP 直接报告TiKV 状态的地址。

> **警告：**
> 如果该值暴露在公网，TiKV 服务器的状态可能会泄露。

- 要禁用 `status-addr`，请将该值设置为 `""`。
- 默认值：`"127.0.0.1:20180"`

4.2.2.4.4 status-thread-pool-size

- HTTP API 服务的工作线程数量。
- 默认值：1
- 最小值：1

4.2.2.4.5 grpc-compression-type

- gRPC 消息的压缩算法。
- 可选值：`"none"`、`"deflate"`、`"gzip"`
- 默认值：`"none"`

4.2.2.4.6 grpc-concurrency

- gRPC 工作线程的数量。调整 gRPC 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：5
- 最小值：1

4.2.2.4.7 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

4.2.2.4.8 grpc-memory-pool-quota

- gRPC 可使用的内存大小限制。
- 默认值：无限制
- 建议仅在出现内存不足(OOM) 的情况下限制内存使用。需要注意，限制内存使用可能会导致卡顿。

4.2.2.4.9 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

4.2.2.4.10 max-grpc-send-msg-len

- 设置可发送的最大 gRPC 消息长度。
- 默认值：10485760
- 单位：Bytes
- 最大值：2147483647

4.2.2.4.11 grpc-stream-initial-window-size

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

4.2.2.4.12 grpc-keepalive-time

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

4.2.2.4.13 grpc-keepalive-timeout

- 关闭 gRPC 链接的超时时长。
- 默认值：3s
- 最小值：1s

4.2.2.4.14 concurrent-send-snap-limit

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

4.2.2.4.15 concurrent-recv-snap-limit

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

4.2.2.4.16 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

4.2.2.4.17 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

4.2.2.4.18 snap-io-max-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

4.2.2.4.19 enable-request-batch

- 控制是否开启批处理请求。
- 默认值：true

4.2.2.4.20 labels

- 指定服务器属性，例如{ zone = "us-west-1", disk = "ssd" }。
- 默认值：{}

4.2.2.4.21 background-thread-count

- 后台线程池的工作线程数量，包括 endpoint 线程、 BR 线程、 split check 线程、 Region 线程以及其他延迟不敏感的任务线程。
- 默认值：当 CPU 核数小于 16 时，默认值为 2。否则，默认值为 3。

4.2.2.4.22 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

4.2.2.4.23 raft-client-queue-size

- 该配置项指定TiKV 中发送Raft 消息的缓冲区大小。如果存在消息发送不及时导致缓冲区满、消息被丢弃的情况，可以适当调大该配置项值以提升系统运行的稳定性。
- 默认值：8192

4.2.2.4.24 simplify-metrics 从 v6.2.0 版本开始引入

- 是否精简返回的监控指标 Metrics 数据。设置为 true 后，TiKV 可以通过过滤部分 Metrics 采样数据以减少每次请求返回的 Metrics 数据量。
- 默认值：false

4.2.2.4.25 forward-max-connections-per-address 从 v5.0.0 版本开始引入

- 设置服务与转发请求的连接池大小。设置过小会影响请求的延迟和负载均衡。
- 默认值：4

4.2.2.5 readpool.unified

统一处理读请求的线程池相关的配置项。该线程池自 4.0 版本起取代原有的 storage 和coprocessor 线程池。

4.2.2.5.1 min-thread-count

- 统一处理读请求的线程池最少的线程数量。
- 默认值：1

4.2.2.5.2 max-thread-count

- 统一处理读请求的线程池最多的线程数量，即 UnifyReadPool 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 可调整范围：[min-thread-count, MAX(4, CPU quota * 10)]。其中，MAX(4, CPU quota * 10) 表示：如果 CPU 配额乘 10 小于 4，取 4；如果 CPU 配额乘 10 大于 4，即 CPU 配额大于 0.4，则取 CPU 配额乘 10。
- 默认值：MAX(4, CPU quota * 0.8)

> 注意：
> 增加线程数量会导致上下文切换增多，可能会导致性能下降，因此不推荐修改此配置。

4.2.2.5.3 stack-size

- 统一处理读请求的线程池中线程的栈大小。
- 类型：整数+单位
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 ulimit -sH 命令后，输出的千字节数。

4.2.2.5.4 max-tasks-per-worker

- 统一处理读请求的线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.5.5 auto-adjust-pool-size 从 v6.3.0 版本开始引入

- 是否开启自动调整线程池的大小。开启此配置可以基于当前的 CPU 使用情况，自动调整统一处理读请求的线程池(UnifyReadPool) 的大小，优化TiKV 的读性能。目前线程池自动调整的范围为：[max-thread-<⟶ count, MAX(4, CPU)](上限与max-thread-count 可设置的最大值相同)。
- 默认值：false

4.2.2.6 readpool.storage

存储线程池相关的配置项。

4.2.2.6.1 use-unified-pool

- 是否使用统一的读取线程池（在readpool.unified 中配置）处理存储请求。该选项值为 false 时，使用单独的存储线程池。通过本节(readpool.storage) 中的其余配置项配置单独的线程池。
- 默认值：如果本节(readpool.storage) 中没有其他配置，默认为true。否则，为了升级兼容性，默认为 false，请根据需要更改readpool.unified 中的配置后再启用该选项。

4.2.2.6.2 high-concurrency

- 处理高优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

4.2.2.6.3 normal-concurrency

- 处理普通优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

4.2.2.6.4 low-concurrency

- 处理低优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

4.2.2.6.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.6.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.6.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.6.8 stack-size

- Storage 读线程池中线程的栈大小。
- 类型：整数+ 单位
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

4.2.2.7 readpool.coprocessor

协处理器线程池相关的配置项。

4.2.2.7.1 use-unified-pool

- 是否使用统一的读取线程池（在`readpool.unified` 中配置）处理协处理器请求。该选项值为 false 时，使用单独的协处理器线程池。通过本节(`readpool.coprocessor`) 中的其余配置项配置单独的线程池。
- 默认值：如果本节(`readpool.coprocessor`) 中没有其他配置，默认为true。否则，为了升级兼容性，默认为 false，请根据需要更改`readpool.unified` 中的配置后再启用该选项。

4.2.2.7.2 high-concurrency

- 处理高优先级 Coprocessor 请求（如点查）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

4.2.2.7.3 normal-concurrency

- 处理普通优先级 Coprocessor 请求的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

4.2.2.7.4 low-concurrency

- 处理低优先级 Coprocessor 请求（如扫表）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

4.2.2.7.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.7.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.7.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

4.2.2.7.8 stack-size

- Coprocessor 线程池中线程的栈大小。
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

4.2.2.8 storage

存储相关的配置项。

4.2.2.8.1 data-dir

- RocksDB 存储路径。
- 默认值：`"./"`

4.2.2.8.2 engine 从 v6.6.0 版本开始引入

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提 issue 反馈。

- 设置存储引擎类型。该配置只能在创建新集群时指定，且后续无法更改。
- 默认值：`"raft-kv"`
- 可选值：
  - `"raft-kv"`：TiDB v6.6.0 之前版本的默认存储引擎。
  - `"partitioned-raft-kv"`：TiDB v6.6.0 新引入的存储引擎。

4.2.2.8.3 scheduler-concurrency

- scheduler 内置一个内存锁机制，防止同时对一个 key 进行操作。每个 key hash 到不同的槽。
- 默认值：524288
- 最小值：1

4.2.2.8.4 scheduler-worker-pool-size

- Scheduler 线程池中线程的数量。Scheduler 线程主要负责写入之前的事务一致性检查工作。如果 CPU 核心数量大于等于 16，默认为 8；否则默认为 4。调整 scheduler 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：4
- 可调整范围：`[1, MAX(4, CPU)]`。其中，`MAX(4, CPU)` 表示：如果 CPU 核心数量小于 4，取 4；如果 CPU 核心数量大于 4，则取 CPU 核心数量。

4.2.2.8.5 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

4.2.2.8.6 enable-async-apply-prewrite

- 控制异步提交(Async Commit)事务在应用prewrite请求之前是否响应TiKV客户端。开启该配置项可以降低apply耗时较高时的延迟，或者减少apply耗时不稳定时的延迟抖动。
- 默认值：false

4.2.2.8.7 reserve-space

- TiKV 启动时会预留一块空间用于保护磁盘空间。当磁盘剩余空间小于该预留空间时，TiKV 会限制部分写操作。预留空间形式上分为两个部分：预留空间的 80% 用作磁盘空间不足时的运维操作所需要的额外磁盘空间，剩余的 20% 为磁盘临时文件。在回收空间的过程中，如果额外使用的磁盘空间过多，导致存储耗尽时，该临时文件会成为恢复服务的最后一道防御。
- 临时文件名为 space_placeholder_file，位于 storage.data-dir 目录下。当TiKV 因磁盘空间耗尽而下线时，重启TiKV 会自动删除该临时文件，并自动尝试回收空间。
- 当剩余空间不足时，TiKV 不会创建该临时文件。防御的有效性与预留空间的大小有关。预留空间大小的计算方式为磁盘容量的 5% 与该配置项之间的最大值。当该配置项的值为 0MB 时，TiKV 会关闭磁盘防护功能。
- 默认值：5GB
- 单位：MB|GB

4.2.2.8.8 enable-ttl

> 警告：
>
> - 你只能在部署新的TiKV 集群时将 enable-ttl 的值设置为 true 或 false，不能在已有的 TiKV 集群中修改该配置项的值。由于该配置项为 true 和 false 的TiKV 集群所存储的数据格式不相同，如果你在已有的TiKV 集群中修改该配置项的值，会造成不同格式的数据存储在同一个集群，导致重启对应的 TiKV 集群时TiKV 报“can’t enable ttl on a non-ttl instance”错误。
> - 你只能在TiKV 集群中使用 enable-ttl，不能在有 TiDB 节点的集群中使用该配置项（即在此类集群中把 enable-ttl 设置为 true），否则会导致数据损坏、TiDB 集群升级失败等严重后果。

- TTL 即 Time to live。数据超过 TTL 时间后会被自动删除。用户需在客户端写入请求中指定 TTL。不指定 TTL 即表明相应数据不会被自动删除。
- 默认值：false

4.2.2.8.9 ttl-check-poll-interval

- 回收数据物理空间的检查周期。如果数据超过了 TTL 时间，数据的物理空间会在检查时被强制回收。
- 默认值：12h
- 最小值：0s

4.2.2.8.10 background-error-recovery-window 从 v6.1.0 版本开始引入

- RocksDB 检测到可恢复的后台错误后，所允许的最长恢复时间。如果后台 SST 文件出现损坏，RocksDB 在检测到故障 SST 文件所属的 Peer 后，会通过心跳上报到 PD。PD 随后会进行调度操作移除该 Peer。最后故障 SST 文件将会被直接删除，随后TiKV 后台恢复正常。
- 在恢复操作完成之前，损坏的 SST 文件将一直存在。此时RocksDB 可以继续写入新的内容，但读到损坏的数据范围时会返回错误。
- 如果恢复操作未能在该时间窗口内完成，TiKV 会崩溃。
- 默认值：1h

4.2.2.8.11 api-version 从 v6.1.0 版本开始引入

- TiKV 作为RawKV 存储数据时使用的存储格式与接口版本。
- 可选值：
  - 1：使用API V1。不对客户端传入的数据进行编码，而是原样存储。在 v6.1.0 之前的版本，TiKV 都使用 API V1。
  - 2：使用API V2：
    * 数据采用多版本并发控制(MVCC) 方式存储，其中时间戳由tikv-server 从 PD 获取（即TSO）。
    * 数据根据使用方式划分范围，支持单一集群TiDB、事务 KV、RawKV 应用共存。
    * 需要同时设置 storage.enable-ttl = true。由于 API V2 支持 TTL 特性，因此强制要求打开 enable-ttl 以避免这个参数出现歧义。
    * 启用API V2 后需要在集群中额外部署至少一个tidb-server 以回收过期数据。该tidb-server 可同时提供数据库读写服务。可以部署多个tidb-server 以保证高可用。
    * 需要客户端的支持。请参考对应客户端的API V2 使用说明。
    * 从 v6.2.0 版本开始，你可以通过RawKV CDC 组件实现RawKV 的 Change Data Capture (CDC)。
- 默认值：1

> 警告：
>
> - 由于API V1 和 API V2 底层存储格式不同，因此仅当TiKV 中只有TiDB 数据时，可以平滑启用或关闭API V2。其他情况下，需要新建集群，并使用TiKV Backup & Restore 工具进行数据迁移。
> - 启用API V2 后，不能将TiKV 集群回退到 v6.1.0 之前的版本，否则可能导致数据损坏。

4.2.2.9 storage.block-cache

RocksDB 多个 CF 之间共享block cache 的配置选项。

4.2.2.9.1 capacity

- 共享 block cache 的大小。
- 默认值：系统总内存大小的 45%
- 单位：KB|MB|GB

4.2.2.10 storage.flow-control

在 scheduler 层进行流量控制代替RocksDB 的 write stall 机制，可以避免 write stall 机制卡住Raftstore 或 Apply 线程导致的次生问题。本节介绍TiKV 流量控制机制相关的配置项。

4.2.2.10.1 enable

- 是否开启流量控制机制。开启后，TiKV 会自动关闭 KvDB 的 write stall 机制，还会关闭RaftDB 中除memtable 以外的 write stall 机制。
- 默认值：true

4.2.2.10.2 memtables-threshold

- 当 KvDB 的 memtable 的个数达到该阈值时，流控机制开始工作。当 `enable` 的值为 `true` 时，会覆盖 `rocksdb.(defaultcf|writecf|lockcf).max-write-buffer-number` 的配置。
- 默认值：5

4.2.2.10.3 l0-files-threshold

- 当 KvDB 的 L0 文件个数达到该阈值时，流控机制开始工作。当 `enable` 的值为 `true` 时，会覆盖 `rocksdb.(defaultcf|writecf|lockcf).level0-slowdown-writes-trigger`的配置。
- 默认值：20

4.2.2.10.4 soft-pending-compaction-bytes-limit

- 当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制开始拒绝部分写入请求，报错`ServerIsBusy`。当 `enable` 的值为 `true` 时，会覆盖 `rocksdb.(defaultcf|writecf|lockcf).soft-pending-compaction-bytes-limit` 的配置。
- 默认值：“192GB”

4.2.2.10.5 hard-pending-compaction-bytes-limit

- 当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制拒绝所有写入请求，报错 `ServerIsBusy`。当 `enable` 的值为 `true` 时，会覆盖 `rocksdb.(defaultcf|writecf|lockcf).hard-pending-compaction-bytes-limit` 的配置。
- 默认值：“1024GB”

4.2.2.11 storage.io-rate-limit

I/O rate limiter 相关的配置项。

4.2.2.11.1 max-bytes-per-sec

- 限制服务器每秒从磁盘读取数据或写入数据的最大I/O 字节数，I/O 类型由下面的 mode 配置项决定。达到该限制后，TiKV 倾向于放缓后台操作为前台操作节流。该配置项值应设为磁盘的最佳I/O 带宽，例如云盘厂商指定的最大I/O 带宽。
- 默认值：“0MB”

4.2.2.11.2 mode

- 确定哪些类型的I/O 操作被计数并受 max-bytes-per-sec 阈值的限流。当前TiKV 只支持 write-only 只写模式。
- 可选值：`"read-only"`，`"write-only"`，`"all-io"`
- 默认值：`"write-only"`

4.2.2.12 pd

4.2.2.12.1 endpoints

- PD 的地址。当指定多个地址时，需要用逗号，分隔。
- 默认值：`["127.0.0.1:2379"]`

4.2.2.12.2 retry-interval

- 初始化 PD 连接时的重试间隔。
- 默认值：`"300ms"`

4.2.2.12.3 retry-log-every

- 指定 PD 客户端在观察到错误时跳过报错的频率。例如，当配置项值为 5 时，每次 PD 观察到错误时，将跳过 4 次报错，直到第 5 次错误时才报告。
- 要禁用此功能，请将值设置为 1。
- 默认值：10

4.2.2.12.4 retry-max-count

- 初始化 PD 连接的最大重试次数。
- 要禁用重试，请将该值设置为 0。要解除重试次数的限制，请将该值设置为 -1。
- 默认值：-1

4.2.2.13 raftstore

raftstore 相关的配置项。

4.2.2.13.1 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

4.2.2.13.2 capacity

- 存储容量，即允许的最大数据存储大小。如果没有设置，则使用当前磁盘容量。如果要将多个TiKV 实例部署在同一块物理磁盘上，需要在TiKV 配置中添加该参数，参见混合部署的关键参数介绍。
- 默认值：0
- 单位：KB|MB|GB

4.2.2.13.3 raftdb-path

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

4.2.2.13.4 raft-base-tick-interval

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 状态机tick 一次的间隔时间。
- 默认值：1s
- 最小值：大于 0

4.2.2.13.5 raft-heartbeat-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 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

4.2.2.13.6 raft-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 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

4.2.2.13.7 raft-min-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 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

4.2.2.13.8 raft-max-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 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

4.2.2.13.9 raft-max-size-per-msg

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 产生的单个消息包的大小限制，软限制。
- 默认值：1MB
- 最小值：大于 0
- 最大值: 3GB
- 单位：KB|MB|GB

4.2.2.13.10 raft-max-inflight-msgs

> **注意：**
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 待确认的日志个数，如果超过这个数量，Raft 状态机会减缓发送日志的速度。
- 默认值：256
- 最小值：大于 0
- 最大值: 16384

4.2.2.13.11 raft-entry-max-size

- 单个日志最大大小，硬限制。
- 默认值：8MB
- 最小值：0
- 单位：MB|GB

4.2.2.13.12 raft-log-compact-sync-interval 从 v5.3 版本开始引入

- 压缩非必要Raft 日志的时间间隔。
- 默认值：“2s”
- 最小值：“0s”

4.2.2.13.13 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：“3s”
- 最小值：“0s”

4.2.2.13.14 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

4.2.2.13.15 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：3/4 Region 大小所能容纳的日志个数，按照每个日志 1 MiB 计算
- 最小值：0

4.2.2.13.16 raft-log-gc-size-limit

- 允许残余的Raft 日志大小，这是一个硬限制。
- 默认值：Region 大小的 3/4
- 最小值：大于 0

4.2.2.13.17 raft-log-reserve-max-ticks 从 v5.3 版本开始引入

- 超过本配置项设置的的tick 数后，即使剩余Raft 日志的数量没有达到 raft-log-gc-threshold 设置的值，TiKV 也会进行GC 操作。
- 默认值：6
- 最小值：大于 0

4.2.2.13.18 raft-engine-purge-interval

- 清除旧的TiKV 日志文件的间隔时间，以尽快回收磁盘空间。Raft 引擎是可替换的组件，因此某些功能或优化的实现需要清除TiKV 日志文件。
- 默认值："10s"

4.2.2.13.19 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

4.2.2.13.20 hibernate-regions

- 打开或关闭静默Region。打开后，如果Region 长时间处于非活跃状态，即被自动设置为静默状态。静默状态的Region 可以降低 Leader 和 Follower 之间心跳信息的系统开销。可以通过 peer-stale-state-check-interval 调整 Leader 和 Follower 之间的心跳间隔。
- 默认值：v5.0.2 及以后版本默认值为true，v5.0.2 以前的版本默认值为 false

4.2.2.13.21 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

4.2.2.13.22 region-split-check-diff

- 允许Region 数据超过指定大小的最大值。
- 默认值：Region 大小的 1/16
- 最小值：0

4.2.2.13.23 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

4.2.2.13.24 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

4.2.2.13.25 region-compact-min-tombstones

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

4.2.2.13.26 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

4.2.2.13.27 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

4.2.2.13.28 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

4.2.2.13.29 snap-mgr-gc-tick-interval

- 触发回收过期 snapshot 文件的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

4.2.2.13.30 snap-gc-timeout

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

4.2.2.13.31 snap-generator-pool-size 从 v5.4.0 版本开始引入

- 用于配置 snap-generator 线程池的大小。
- 为了让TiKV 在恢复场景下加快Region 生成 Snapshot 的速度，需要调大对应Worker 的 snap-generator 线程数量。可通过本配置项调大对应线程的数量。
- 默认值：2
- 最小值：1

4.2.2.13.32 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

4.2.2.13.33 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

4.2.2.13.34 notify-capacity

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

4.2.2.13.35 messages-per-tick

- 每轮处理的消息最大个数。
- 默认值：4096
- 最小值：0

4.2.2.13.36 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：10m
- 最小值：当 Hibernate Region 功能启用时，为peer-stale-state-check-interval * 2；Hibernate Region 功能关闭时，为 0。

4.2.2.13.37 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

4.2.2.13.38 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

4.2.2.13.39 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

4.2.2.13.40 leader-transfer-max-log-lag

- 尝试转移领导权时被转移者允许的最大日志缺失个数。
- 默认值：128
- 最小值：10

4.2.2.13.41 max-snapshot-file-raw-size 从 v6.1.0 版本开始引入

- 当 snapshot 文件大于该配置项指定的大小时，snapshot 文件会被切割为多个文件。
- 默认值：100MiB
- 最小值：100MiB

4.2.2.13.42 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

4.2.2.13.43 consistency-check-interval

> **警告：**
> 开启一致性检查对集群性能有影响，并且和TiDB GC 操作不兼容，不建议在生产环境中使用。

- 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

4.2.2.13.44 raft-store-max-leader-lease

- Region 主可信任期的最长时间。
- 默认值：9s
- 最小值：0

4.2.2.13.45 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

4.2.2.13.46 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

4.2.2.13.47 merge-check-tick-interval

- 触发merge完成检查的时间间隔。
- 默认值：2s
- 最小值：大于 0

4.2.2.13.48 use-delete-range

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

4.2.2.13.49 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

4.2.2.13.50 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

4.2.2.13.51 apply-yield-write-size 从 v6.4.0 版本开始引入

- Apply 线程每一轮处理单个状态机写入的最大数据量，这是个软限制。
- 默认值：32KiB
- 最小值：大于 0
- 单位：KiB|MiB|GiB

4.2.2.13.52 apply-max-batch-size

- Raft 状态机由 BatchSystem 批量执行数据写入请求，该配置项指定每批可执行请求的最多Raft 状态机个数。
- 默认值：256
- 最小值：大于 0
- 最大值: 10240

4.2.2.13.53 apply-pool-size

- Apply 线程池负责把数据落盘至磁盘。该配置项为Apply 线程池中线程的数量，即Apply 线程池的大小。调整Apply 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：2
- 可调整范围：[1, CPU * 10]

4.2.2.13.54 store-max-batch-size

- Raft 状态机由 BatchSystem 批量执行把日志落盘至磁盘的请求，该配置项指定每批可执行请求的最多Raft 状态机个数。
- 如果开启 hibernate-regions，默认值为 256；如果关闭 hibernate-regions，默认值为 1024
- 最小值：大于 0
- 最大值: 10240

4.2.2.13.55 store-pool-size

- 表示处理Raft 的线程池中线程的数量，即Raftstore 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 默认值：2
- 可调整范围：[1, CPU * 10]

4.2.2.13.56 store-io-pool-size 从 v5.3.0 版本开始引入

- 表示处理Raft I/O 任务的线程池中线程的数量，即 StoreWriter 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 默认值：0
- 最小值：0

4.2.2.13.57 future-poll-size

- 驱动 future 的线程池中线程的数量。
- 默认值：1
- 最小值：大于 0

4.2.2.13.58 cmd-batch

- 对请求进行攒批的控制开关，开启后可显著提升写入性能。
- 默认值：true

4.2.2.13.59 inspect-interval

- TiKV 每隔一段时间会检测Raftstore 组件的延迟情况，该配置项设置检测的时间间隔。当检测的延迟超过该时间，该检测会被记为超时。
- 根据超时的检测延迟的比例计算判断TiKV 是否为慢节点。
- 默认值：500ms
- 最小值：1ms

4.2.2.13.60 raft-write-size-limit 从 v5.3.0 版本开始引入

- 触发Raft 数据写入的阈值。当数据大小超过该配置项值，数据会被写入磁盘。当 store-io-pool-size 的值为 0 时，该配置项不生效。
- 默认值：1MB
- 最小值：0

4.2.2.13.61 report-min-resolved-ts-interval

- 设置 PD leader 收到Resolved TS 的最小间隔时间。如果该值设置为 0，表示禁用该功能。
- 默认值：`"1s"`，即最小正值
- 最小值：0
- 单位：秒

4.2.2.14 coprocessor

Coprocessor 相关的配置项。

4.2.2.14.1 split-region-on-table

- 开启按table 分裂Region 的开关，建议仅在TiDB 模式下使用。
- 默认值：false

4.2.2.14.2 batch-split-limit

- 批量分裂Region 的阈值，调大该值可加速分裂Region。
- 默认值：10
- 最小值：1

4.2.2.14.3 region-max-size

- Region 容量空间最大值，超过时系统分裂成多个Region。
- 默认值：`region-split-size / 2 * 3`
- 单位：KiB|MiB|GiB

4.2.2.14.4 region-split-size

- 分裂后新Region 的大小，此值属于估算值。
- 默认值：96MiB
- 单位：KiB|MiB|GiB

4.2.2.14.5 region-max-keys

- Region 最多允许的 key 的个数，超过时系统分裂成多个Region。
- 默认值：`region-split-keys / 2 * 3`

4.2.2.14.6 region-split-keys

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

4.2.2.14.7 consistency-check-method

- 指定数据一致性检查的方法。
- 要对 MVCC 数据进行一致性检查，设置该值为 "mvcc"。要对原始数据进行一致性检查，设置该值为 "raw"。
- 默认值："mvcc"

4.2.2.15 coprocessor-v2

4.2.2.15.1 coprocessor-plugin-directory

- 已编译coprocessor 插件所在目录的路径。TiKV 会自动加载该目录下的插件。
- 如果未设置该配置项，则coprocessor 插件会被禁用。
- 默认值："./coprocessors"

4.2.2.15.2 enable-region-bucket 从 v6.1.0 版本开始引入

- 是否将Region 划分为更小的区间bucket，并且以bucket 作为并发查询单位，以提高扫描数据的并发度。bucket 的详细设计可见 Dynamic size Region。
- 默认值：false

> 警告：
>
> - enable-region-bucket 是TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。
> - 这个参数仅在 region-split-size 调到两倍 region-bucket-size 及以上时才有意义，否则不会真正生成bucket。
> - 将 region-split-size 调大可能会有潜在的性能回退、数据调度缓慢的风险。

4.2.2.15.3 region-bucket-size 从 v6.1.0 版本开始引入

- 设置 enable-region-bucket 启用时bucket 的预期大小。
- 默认值：96MiB

> 警告：
>
> region-bucket-size 是TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。

4.2.2.15.4 report-region-buckets-tick-interval 从 v6.1.0 版本开始引入

> **警告：**
>
> report-region-buckets-tick-interval 是 TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。

- 启用 enable-region-bucket 后，该配置项设置TiKV 向 PD 上报bucket 信息的间隔时间。
- 默认值：10s

4.2.2.16 rocksdb

RocksDB 相关的配置项。

4.2.2.16.1 max-background-jobs

- RocksDB 后台线程个数。 调整RocksDB 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：
  - CPU 核数为 10 时，默认值为 9
  - CPU 核数为 8 时，默认值为 7
  - CPU 核数为N 时，默认值为 max(2, min(N - 1, 9))
- 最小值：2

4.2.2.16.2 max-background-flushes

- RocksDB 用于刷写memtable 的最大后台线程数量。
- 默认值：
  - CPU 核数为 10 时，默认值为 3
  - CPU 核数为 8 时，默认值为 2
  - CPU 核数为N 时，默认值为 [(max-background-jobs + 3)/ 4]
- 最小值：1

4.2.2.16.3 max-sub-compactions

- RocksDB 进行 subcompaction 的并发个数。
- 默认值：3
- 最小值：1

4.2.2.16.4 max-open-files

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

4.2.2.16.5 max-manifest-file-size

- RocksDB Manifest 文件最大大小。
- 默认值：128MB
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.16.6 create-if-missing

- 自动创建 DB 开关。
- 默认值：true

4.2.2.16.7 wal-recovery-mode

- 预写式日志(WAL, Write Ahead Log) 的恢复模式。
- 可选值：
  - "tolerate-corrupted-tail-records"：容忍并丢弃位于日志尾部的不完整的数据 (trailing data)。
  - "absolute-consistency"：当发现待恢复的日志中有被损坏的日志时，放弃恢复所有日志。
  - "point-in-time"：按顺序恢复日志。遇到第一个损坏的日志时，停止恢复剩余的日志。
  - "skip-any-corrupted-records"：灾难后恢复。跳过日志中的损坏记录，尽可能多地恢复数据。
- 默认值："point-in-time"

4.2.2.16.8 wal-dir

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

4.2.2.16.9 wal-ttl-seconds

- 归档WAL 生存周期，超过该值时，系统会删除相关WAL。
- 默认值：0
- 最小值：0
- 单位：秒

4.2.2.16.10 wal-size-limit

- 归档WAL 大小限制，超过该值时，系统会删除相关WAL。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.16.11 max-total-wal-size

- RocksDB WAL 总大小限制，即 data-dir 目录下 *.log 文件的大小总和。
- 默认值："4GB"

4.2.2.16.12 stats-dump-period

- 将统计信息输出到日志中的间隔时间。
- 默认值：10m

4.2.2.16.13 compaction-readahead-size

- 开启RocksDB compaction 过程中的预读功能，该项指定预读数据的大小。如果使用的是机械磁盘，建议该值至少为 2MB。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.16.14 writable-file-max-buffer-size

- WritableFileWrite 所使用的最大的buffer 大小。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

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

- 决定后台 flush 或者compaction 的读写是否设置 O_DIRECT 的标志。该选项对性能的影响：开启 O_DIRECT 可以绕过并防止污染操作系统buffer cache，但后续文件读取需要把内容重新读到buffer cache。
- 默认值：false

4.2.2.16.16 rate-bytes-per-sec

- RocksDB compaction rate limiter 的限制速率。
- 默认值：10GB
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.16.17 rate-limiter-refill-period

- 控制I/O 令牌的刷新频率。较小的值可以减少I/O 尖刺，但会增加 CPU 开销。
- 默认值：`"100ms"`

4.2.2.16.18 rate-limiter-mode

- RocksDB 的 compaction rate limiter 模式。
- 可选值：“read-only”，“write-only”，“all-io”
- 默认值：“write-only”

4.2.2.16.19 rate-limiter-auto-tuned 从 v5.0 版本开始引入

- 控制是否依据最近的负载量自动优化RocksDB 的 compaction rate limiter 配置。此配置项开启后，compaction pending bytes 监控指标值会比一般情况下稍微高些。
- 默认值：true

4.2.2.16.20 enable-pipelined-write

- 控制是否开启 Pipelined Write。开启时会使用旧的 Pipelined Write，关闭时会使用新的 Pipelined Commit 机制。
- 默认值：false

4.2.2.16.21 bytes-per-sync

- 异步 Sync 限速速率。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.16.22 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

4.2.2.16.23 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

4.2.2.16.24 info-log-roll-time

- 日志截断间隔时间，如果为 0s 则不截断。
- 默认值：0s

4.2.2.16.25 info-log-keep-log-file-num

- 保留日志文件最大个数。
- 默认值：10
- 最小值：0

4.2.2.16.26 info-log-dir

- 日志存储目录。
- 默认值：“”

4.2.2.16.27 info-log-level

- RocksDB 的日志级别。
- 默认值："info"

4.2.2.16.28 write-buffer-flush-oldest-first 从 v6.6.0 版本开始引入

> 警告：
>
> 该功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提 issue 反馈。

- 设置当RocksDB 当前memtable 内存占用达到阈值之后的 Flush 策略。
- 默认值：false
- 可选值：
  - false：Flush 策略是优先选择数据量大的memtable 落盘到 SST。
  - true：Flush 策略是优先选择最早的memtable 落盘到 SST。该策略可以清除冷数据的 memtable，用于有明显冷热数据的场景。

4.2.2.16.29 write-buffer-limit 从 v6.6.0 版本开始引入

> 警告：
>
> 该功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在GitHub 上提 issue 反馈。

- 设置单个TiKV 中所有 RocksDB 实例使用的memtable 的总内存上限，默认值为本机内存的 25%，推荐配置不低于 5 GiB 的内存。该配置只对分区Raft KV (storage.engine="partitioned-raft-kv") 生效。
- 默认值：25%
- 单位：KiB|MiB|GiB

4.2.2.17 rocksdb.titan

Titan 相关的配置项。

4.2.2.17.1 enabled

- 开启Titan 开关。
- 默认值：false

4.2.2.17.2 dirname

- Titan Blob 文件存储目录。
- 默认值：titandb

4.2.2.17.3 disable-gc

- 关闭Titan 对 Blob 文件的GC 的开关。
- 默认值：false

4.2.2.17.4 max-background-gc

- Titan 后台GC 的线程个数。
- 默认值：4
- 最小值：1

4.2.2.18 rocksdb.defaultcf | rocksdb.writecf | rocksdb.lockcf

rocksdb defaultcf、 rocksdb writecf 和 rocksdb lockcf 相关的配置项。

4.2.2.18.1 block-size

- 一个RocksDB block 的默认大小。
- defaultcf 默认值：32KB
- writecf 默认值：32KB
- lockcf 默认值：16KB
- 最小值：1KB
- 单位：KB|MB|GB

4.2.2.18.2 block-cache-size

> **警告：**
>
> 从 v6.6.0 起，该配置项被废弃。

- 一个 RocksDB block 的默认缓存大小。
- defaultcf 默认值：机器总内存 * 25%
- writecf 默认值：机器总内存 * 15%
- lockcf 默认值：机器总内存 * 2%
- 最小值：0
- 单位：KB|MB|GB

4.2.2.18.3 disable-block-cache

- 开启 block cache 开关。
- 默认值：false

4.2.2.18.4 cache-index-and-filter-blocks

- 开启缓存 index 和 filter 的开关。
- 默认值：true

4.2.2.18.5 pin-l0-filter-and-index-blocks

- 控制第 0 层 SST 文件的 index block 和 filter block 是否常驻在内存中的开关。
- 默认值：true

4.2.2.18.6 use-bloom-filter

- 开启 bloom filter 的开关。
- 默认值：true

4.2.2.18.7 optimize-filters-for-hits

- 开启优化 filter 的命中率的开关。
- defaultcf 默认值：true
- writecf 默认值：false
- lockcf 默认值：false

4.2.2.18.8 optimize-filters-for-memory 从 v7.1.0 版本开始引入

- 控制是否生成能够最小化内存碎片的 Bloom/Ribbon filter。
- 只有当 format-version >= 5 时，该配置项才生效。
- 默认值：false

4.2.2.18.9 whole-key-filtering

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

4.2.2.18.10 bloom-filter-bits-per-key

- bloom filter 为每个 key 预留的长度。
- 默认值：10
- 单位：字节

4.2.2.18.11 block-based-bloom-filter

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

4.2.2.18.12 ribbon-filter-above-level 从 v7.1.0 版本开始引入

- 控制是否对于大于等于该值的 level 使用 Ribbon filter，对于小于该值的 level，使用非 block-based bloom filter。当该配置开启时，block-based-bloom-filter 将被忽略。
- 只有当 format-version >= 5 时，该配置项才生效。
- 默认值：false

4.2.2.18.13 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

4.2.2.18.14 compression-per-level

- 每一层默认压缩算法。
- defaultcf 的默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]
- writecf 的默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]
- lockcf 的默认值：[“no”，“no”，“no”，“no”，“no”，“no”，“no”]

4.2.2.18.15 bottommost-level-compression

- 设置最底层的压缩算法。该设置将覆盖 compression-per-level 的设置。
- 因为最底层并非从数据开始写入LSM-tree 起就直接采用 compression-per-level 数组中的最后一个压缩算法，使用 bottommost-level-compression 可以让最底层从一开始就使用压缩效果最好的压缩算法。
- 如果不想设置最底层的压缩算法，可以将该配置项的值设为 disable。
- 默认值：“zstd”

4.2.2.18.16 write-buffer-size

- memtable 大小。
- defaultcf 默认值："128MB"
- writecf 默认值："128MB"
- lockcf 默认值："32MB"
- 最小值：0
- 单位：KB|MB|GB

4.2.2.18.17 max-write-buffer-number

- 最大memtable 个数。当storage.flow-control.enable 的值为true 时，storage.flow-control.memtables <⟶ -threshold 会覆盖此配置。
- 默认值：5
- 最小值：0

4.2.2.18.18 min-write-buffer-number-to-merge

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

4.2.2.18.19 max-bytes-for-level-base

- base level (L1) 最大字节数，一般设置为 memtable 大小 4 倍。当 L1 的数据量大小达到 max-bytes-for-level <⟶ -base 限定的值的时候，会触发 L1 的 SST 文件和 L2 中有 overlap 的 SST 文件进行compaction。
- defaultcf 默认值："512MB"
- writecf 默认值："512MB"
- lockcf 默认值："128MB"
- 最小值：0
- 单位：KB|MB|GB
- 建议 max-bytes-for-level-base 的取值和 L0 的数据量大致相等，以减少不必要的compaction。假如压缩方式为“no:no:lz4:lz4:lz4:lz4:lz4”，那么 max-bytes-for-level-base 的值应该是 write-buffer-size * 4，因为 L0 和 L1 均没有压缩，且 L0 触发compaction 的条件是 SST 文件的个数到达 4（默认值）。当 L0 和 L1 都发生了compaction 时，需要分析RocksDB 的日志了解由一个memtable 压缩成的 SST 文件的大小。如果文件大小为 32MB，那么 max-bytes-for-level-base 的值建议设为 32MB * 4 = 128MB。

4.2.2.18.20 target-file-size-base

- base level 的目标文件大小。当 enable-compaction-guard 的值为 true 时，compaction-guard-max-output-file-size 会覆盖此配置。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

4.2.2.18.21 level0-file-num-compaction-trigger

- 触发 compaction 的 L0 文件最大个数。
- defaultcf 默认值：4
- writecf 默认值：4
- lockcf 默认值：1
- 最小值：0

4.2.2.18.22 level0-slowdown-writes-trigger

- 触发 write stall 的 L0 文件最大个数。当 storage.flow-control.enable 的值为 true 时，storage.flow-control.l0-files-threshold 会覆盖此配置。
- 默认值：20
- 最小值：0

4.2.2.18.23 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

4.2.2.18.24 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

4.2.2.18.25 compaction-pri

- 优先处理 compaction 的类型。
- 可选值：
  - "by-compensated-size"：根据大小顺序，优先对大文件进行 compaction。
  - "oldest-largest-seq-first"：根据时间顺序，优先对数据更新时间晚的文件进行 compaction。当你只在小范围内更新部分热点键(hot keys) 时，可以使用此配置。

  - "oldest-smallest-seq-first"：根据时间顺序，优先对长时间没有被 compact 到下一级的文件进行 compaction。如果你在大范围内随机更新了部分热点键，使用该配置可以轻微缓解写放大。
  - "min-overlapping-ratio"：根据重叠比例，优先对在不同层之间文件重叠比例高的文件进行 compaction，即一个文件在下一层的大小/本层的大小的值越小，compaction 的优先级越高。在诸多场景下，该配置可以有效缓解写放大。
- 默认值：
  - defaultcf 和 writecf 的默认值："min-overlapping-ratio"
  - lockcf 的默认值："by-compensated-size"

4.2.2.18.26 dynamic-level-bytes

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

4.2.2.18.27 num-levels

- RocksDB 文件最大层数。
- 默认值：7

4.2.2.18.28 max-bytes-for-level-multiplier

- 每一层的默认放大倍数。
- 默认值：10

4.2.2.18.29 compaction-style

- compaction 方法。
- 可选值：“level”，“universal”，“fifo”
- 默认值：“level”

4.2.2.18.30 disable-auto-compactions

- 是否关闭自动compaction。
- 默认值：false

4.2.2.18.31 soft-pending-compaction-bytes-limit

- pending compaction bytes 的软限制。当 storage.flow-control.enable 的值为 true 时，storage.flow-control.soft-pending-compaction-bytes-limit 会覆盖此配置。
- 默认值：192GB
- 单位：KB|MB|GB

4.2.2.18.32 hard-pending-compaction-bytes-limit

- pending compaction bytes 的硬限制。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.hard-pending-compaction-bytes-limit` 会覆盖此配置。
- 默认值：256GB
- 单位：KB|MB|GB

4.2.2.18.33 enable-compaction-guard

- 设置compaction guard 的启用状态。compaction guard 优化通过使用TiKV Region 边界分割 SST 文件，帮助降低 compaction I/O，让TiKV 能够输出较大的 SST 文件，并且在迁移Region 时及时清理过期数据。
- `defaultcf` 默认值：`true`
- `writecf` 默认值：`true`
- `lockcf` 默认值：`false`

4.2.2.18.34 compaction-guard-min-output-file-size

- 设置compaction guard 启用时 SST 文件大小的最小值，防止 SST 文件过小。
- 默认值：8MB
- 单位：KB|MB|GB

4.2.2.18.35 compaction-guard-max-output-file-size

- 设置compaction guard 启用时 SST 文件大小的最大值，防止 SST 文件过大。对于同一列族，此配置项的值会覆盖 `target-file-size-base`。
- 默认值：128MB
- 单位：KB|MB|GB

4.2.2.18.36 format-version 从 v6.2.0 版本开始引入

- 设置 SST 文件的格式版本。该配置项只影响新写入的表，对于已经存在的表，版本信息会从 footer 中读取。
- 可选值：
  - 0：适用于所有TiKV 版本。默认checksum 类型为 CRC32。该版本不支持修改 checksum 类型。
  - 1：适用于所有TiKV 版本。支持使用非默认的checksum 类型，例如xxHash。只有在checksum 类型不是 CRC32 时，RocksDB 才会写入数据。（0 版本会自动升级）
  - 2：适用于所有TiKV 版本。更改了压缩块的编码方式，使用 LZ4、BZip2 和 Zlib 压缩。
  - 3：适用于TiKV v2.1 及以上版本。更改了索引块中 key 的编码方式。
  - 4：适用于TiKV v3.0 及以上版本。更改了索引块中 value 的编码方式。
  - 5：适用于TiKV v6.1 及以上版本。全量和分区 filter 采用一种具有不同模式的、更快、更准确的 Bloom filter 实现。
- 默认值：2

4.2.2.19 rocksdb.defaultcf.titan

rocksdb defaultcf titan 相关的配置项。

4.2.2.19.1 min-blob-size

- 最小存储在 Blob 文件中 value 大小，低于该值的 value 还是存在LSM-Tree 中。
- 默认值：1KB
- 最小值：0
- 单位：KB|MB|GB

4.2.2.19.2 blob-file-compression

- Blob 文件所使用的压缩算法，可选值：no、 snappy、 zlib、 bz2、 lz4、 lz4hc、 zstd。
- 默认值：lz4

4.2.2.19.3 blob-cache-size

- Blob 文件的cache 大小。
- 默认值：0GB
- 最小值：0
- 单位：KB|MB|GB

4.2.2.19.4 min-gc-batch-size

- 做一次GC 所要求的最低 Blob 文件大小总和。
- 默认值：16MB
- 最小值：0
- 单位：KB|MB|GB

4.2.2.19.5 max-gc-batch-size

- 做一次GC 所要求的最高 Blob 文件大小总和。
- 默认值：64MB
- 最小值：0
- 单位：KB|MB|GB

4.2.2.19.6 discardable-ratio

- Blob 文件GC 的触发比例，如果某 Blob 文件中的失效 value 的比例高于该值才可能被GC 选中。
- 默认值：0.5
- 最小值：0
- 最大值：1

4.2.2.19.7 sample-ratio

- 进行GC 时，对 Blob 文件进行采样时读取数据占整个文件的比例。
- 默认值：0.1
- 最小值：0
- 最大值：1

4.2.2.19.8 merge-small-file-threshold

- Blob 文件的大小小于该值时，无视discardable-ratio 仍可能被GC 选中。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

4.2.2.19.9 blob-run-mode

- Titan 的运行模式选择。
- 可选值：
  - “normal”：value size 超过min-blob-size 的数据会写入到blob 文件。
  - “read_only”：不再写入新数据到blob，原有blob 内的数据仍然可以读取。
  - “fallback”：将blob 内的数据写回LSM。
- 默认值：“normal”

4.2.2.19.10 level-merge

- 是否通过开启level-merge 来提升读性能，副作用是写放大会比不开启更大。
- 默认值：false

4.2.2.20 raftdb

raftdb 相关配置项。

4.2.2.20.1 max-background-jobs

- RocksDB 后台线程个数。调整RocksDB 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：4
- 最小值：2

4.2.2.20.2 max-sub-compactions

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

4.2.2.20.3 max-open-files

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

4.2.2.20.4 max-manifest-file-size

- 单个RocksDB Manifest 文件的最大大小。
- 默认值："20MB"
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.20.5 create-if-missing

- 如果值为 true，当数据库不存在时将自动创建。
- 默认值：true

4.2.2.20.6 stats-dump-period

- 输出统计信息到日志的时间间隔。
- 默认值：10m

4.2.2.20.7 wal-dir

- 存储Raft RocksDB WAL 文件的目录，即WAL 的绝对路径。请勿将该配置项设置为与rocksdb.wal-dir 相同的值。
- 如果未设置该配置项，日志文件将存储在与数据相同的目录中。
- 如果机器上有两个磁盘，将RocksDB 数据和WAL 日志存储在不同磁盘上可以提高性能。
- 默认值：""

4.2.2.20.8 wal-ttl-seconds

- 归档的WAL 文件的保留时间。当超过该值时，系统将删除这些文件。
- 默认值：0
- 最小值：0
- 单位：秒

4.2.2.20.9 wal-size-limit

- 归档WAL 文件的大小限制。当超过该值时，系统将删除这些文件。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

4.2.2.20.10 max-total-wal-size

- RocksDB WAL 文件的最大总大小。
- 默认值：`"4GB"`

4.2.2.20.11 compaction-readahead-size

- 控制在RocksDB compaction 时是否开启预读取功能，并指定预读取数据的大小。
- 如果使用机械硬盘，建议将该值至少设置为 `2MB`。
- 默认值：`0`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

4.2.2.20.12 writable-file-max-buffer-size

- WriteableFileWrite 中使用的最大缓冲区大小。
- 默认值：`"1MB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

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

- 控制是否在后台刷新和compaction 时使用 `O_DIRECT` 进行读写。启用 `O_DIRECT` 的性能影响：它可以绕过和防止操作系统缓存污染，但是后续文件读取需要重新读取内容到缓存中。
- 默认值：`false`

4.2.2.20.14 enable-pipelined-write

- 控制是否开启 Pipelined Write。开启时会使用旧的 Pipelined Write，关闭时会使用新的 Pipelined Commit 机制。
- 默认值：`true`

4.2.2.20.15 allow-concurrent-memtable-write

- 控制是否开启并发memtable 写入。
- 默认值：`true`

4.2.2.20.16 bytes-per-sync

- 异步 Sync 限速速率。
- 默认值：`"1MB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

4.2.2.20.17 wal-bytes-per-sync

- WAL Sync 限速速率。
- 默认值：`"512KB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

4.2.2.20.18 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`

4.2.2.20.19 info-log-roll-time

- Info 日志截断间隔时间，如果为 `"0s"` 则不截断。
- 默认值：`"0s"`

4.2.2.20.20 info-log-keep-log-file-num

- RaftDB 中保存的Info 日志文件的最大数量。
- 默认值：`10`
- 最小值：`0`

4.2.2.20.21 info-log-dir

- Info 日志存储的目录。
- 默认值：`""`

4.2.2.20.22 info-log-level

- RaftDB 的日志级别。
- 默认值：`"info"`

4.2.2.21 raft-engine

Raft Engine 相关的配置项。

> **注意：**

- 第一次开启Raft Engine 时，TiKV 会将原有的RocksDB 数据转移至Raft Engine 中。因此，TiKV 的启动时间会比较长，你需要额外等待几十秒。
- 如果你要将TiDB 集群降级至 v5.4.0 以前的版本（不含 v5.4.0），你需要在降级之前先关闭 Raft Engine（即把 enable 配置项设置为 false，并重启 TiKV 使配置生效），否则会导致集群降级后无法正常开启。

4.2.2.21.1 enable

- 决定是否使用Raft Engine 来存储Raft 日志。开启该配置项后，raftdb 的配置不再生效。
- 默认值：true

4.2.2.21.2 dir

- 存储Raft 日志文件的目录。如果该目录不存在，则在启动TiKV 时创建该目录。
- 如果未设置此配置，则使用 {data-dir}/raft-engine。
- 如果你的机器上有多个磁盘，建议将Raft Engine 的数据存储在单独的磁盘上，以提高TiKV 性能。
- 默认值："

4.2.2.21.3 batch-compression-threshold

- 指定日志批处理的阈值大小。大于此配置的日志批次将被压缩。如果将此配置项设置为 0，则禁用压缩。
- 默认值："8KB"

4.2.2.21.4 bytes-per-sync

- 指定缓存写入的最大累积大小。当超过此配置值时，缓存的写入将被刷写到磁盘。
- 如果将此配置项设置为 0，则禁用增量同步。
- 默认值："4MB"

4.2.2.21.5 target-file-size

- 指定日志文件的最大大小。当日志文件大于此值时，将对其进行轮转。
- 默认值："128MB"

4.2.2.21.6 purge-threshold

- 指定主日志队列的阈值大小。当超过此配置值时，将对主日志队列执行垃圾回收。
- 此参数可用于调整Raft Engine 的空间占用大小。
- 默认值："10GB"

4.2.2.21.7 recovery-mode

- 确定在日志恢复过程中如何处理文件损坏。
- 可选值：`"absolute-consistency"`, `"tolerate-tail-corruption"`, `"tolerate-any-corruption"`
- 默认值：`"tolerate-tail-corruption"`

4.2.2.21.8 recovery-read-block-size

- 恢复期间读取日志文件的最小I/O 大小。
- 默认值：`"16KB"`
- 最小值：`"512B"`

4.2.2.21.9 recovery-threads

- 用于扫描和恢复日志文件的线程数。
- 默认值：4
- 最小值：1

4.2.2.21.10 memory-limit

- 指定 Raft Engine 使用内存的上限。
- 当该配置项未设置时，Raft Engine 默认使用系统总内存的 15%。
- 默认值：系统总内存 * 15%

4.2.2.21.11 format-version 从 v6.3.0 版本开始引入

> **注意：**
>
> format-version 的值设置为 2 后，如果你需要将TiKV 集群降级至 v6.3.0 以前的版本，你需要在降级之前执行如下操作：
>
> 1. 关闭Raft Engine。将enable 配置项设置为 false，并重启 TiKV 使配置生效。
> 2. 将 format-version 的值重新设置为 1。
> 3. 重新打开Raft Engine，即把 enable 配置项重设为 true，并重启 TiKV 使配置生效。

- 指定 Raft Engine 的日志文件格式版本。
- 可选值：
  - 1：v6.3.0 以前的默认日志文件格式。v6.1.0 及以后版本的TiKV 可以读取该格式。
  - 2：支持日志回收。v6.3.0 及以后版本的TiKV 可以读取该格式。
- 默认值：2

4.2.2.21.12 enable-log-recycle 从 v6.3.0 版本开始引入

> **注意：**
>
> 仅在format-version 的值大于等于 2 时，该配置项才生效。

- 控制Raft Engine 是否回收过期的日志文件。该配置项启用时，Raft Engine 将保留逻辑上被清除的日志文件，用于日志回收，减少写负载的长尾延迟。
- 默认值：true

4.2.2.21.13 prefill-for-recycle 从 v7.0.0 版本开始引入

> **注意：**
>
> 仅在enable-log-recycle 的值为 true 时，该配置项才生效。

- 控制Raft Engine 是否自动生成空的日志文件用于日志回收。该配置项启用时，Raft Engine 将在初始化时自动填充一批空日志文件用于日志回收，保证日志回收在初始化后立即生效。
- 默认值：false

4.2.2.22 security

安全相关配置项。

4.2.2.22.1 ca-path

- CA 文件路径。
- 默认值：“”

4.2.2.22.2 cert-path

- 包含X.509 证书的PEM 文件路径。
- 默认值：“”

4.2.2.22.3 key-path

- 包含X.509 key 的 PEM 文件路径。
- 默认值：“”

4.2.2.22.4 cert-allowed-cn

- 客户端提供的证书中，可接受的X.509通用名称列表。仅当提供的通用名称与列表中的条目之一完全匹配时，才会允许其请求。
- 默认值：`[]`。这意味着默认情况下禁用客户端证书 CN 检查。

4.2.2.22.5 redact-info-log 从 v4.0.8 版本开始引入

- 若开启该选项，日志中的用户数据会以? 代替。
- 默认值：`false`

4.2.2.23 security.encryption

静态加密(TDE)有关的配置项。

4.2.2.23.1 data-encryption-method

- 数据文件的加密方法。
- 可选值：`"plaintext"`，`"aes128-ctr"`，`"aes192-ctr"`，`"aes256-ctr"`，`"sm4-ctr"`（从 v6.3.0 开始支持）
- 选择 `"plaintext"` 以外的值则表示启用加密功能。此时必须指定主密钥。
- 默认值：`"plaintext"`

4.2.2.23.2 data-key-rotation-period

- 指定TiKV轮换数据密钥的频率。
- 默认值：`7d`

4.2.2.23.3 enable-file-dictionary-log

- 启用优化，以减少TiKV管理加密元数据时的I/O操作和互斥锁竞争。
- 此配置参数默认启用，为避免可能出现的兼容性问题，请参考静态加密- TiKV 版本间兼容性。
- 默认值：`true`

4.2.2.23.4 master-key

- 指定启用加密时的主密钥。若要了解如何配置主密钥，可以参考静态加密- 配置加密。

4.2.2.23.5 previous-master-key

- 指定轮换新主密钥时的旧主密钥。旧主密钥的配置格式与主密钥相同。若要了解如何配置主密钥，可以参考静态加密- 配置加密。

4.2.2.24 import

用于TiDB Lightning 导入及 BR 恢复相关的配置项。

4.2.2.24.1 num-threads

- 处理RPC 请求的线程数量。
- 默认值：8
- 最小值：1

4.2.2.24.2 stream-channel-window

- Stream channel 的窗口大小。当channel 满时，Stream 会被阻塞。
- 默认值：128

4.2.2.24.3 memory-use-ratio 从 v6.5.0 版本开始引入

- 从 v6.5.0 开始，PITR 支持直接将备份日志文件读取到缓存中，然后进行恢复。此配置项用来配置 PITR 恢复中可用内存与系统总内存的占比。
- 可调整范围：[0.0, 0.5]
- 默认值：0.3，表示系统 30% 的内存可用于 PITR 恢复；当为 0.0 时，表示通过下载日志文件到本地进行 PITR 恢复。

> 注意：
> 在小于 v6.5.0 的版本中，PITR 仅支持将备份文件下载到本地进行恢复。

4.2.2.25 gc

4.2.2.25.1 batch-keys

- 一次GC 操作中的 key 的数量。
- 默认值：512

4.2.2.25.2 max-write-bytes-per-sec

- GC 工作线程每秒可以写入RocksDB 的最大字节数。
- 如果设置为 0，则没有限制。
- 默认值："0"

4.2.2.25.3 enable-compaction-filter 从 v5.0 版本开始引入

- 是否开启GC in Compaction Filter 特性。
- 默认值：true

4.2.2.25.4 ratio-threshold

- 触发 GC 的垃圾比例阈值。
- 默认值：1.1

4.2.2.26 backup

用于 BR 备份相关的配置项。

4.2.2.26.1 num-threads

- 处理备份的工作线程数量。
- 默认值：CPU * 0.5，但最大为 8
- 可调整范围：[1, CPU]
- 最小值：1

4.2.2.26.2 batch-size

- 一次备份的数据范围数量。
- 默认值：8

4.2.2.26.3 sst-max-size

- 备份 SST 文件大小的阈值。如果TiKV Region 中备份文件的大小超过该阈值，则将该文件备份到Region 分割的多个Region 文件中，每个分割Region 中的文件大小均为 sst-max-size（或略大）。
- 例如，当Region [a,e) 中备份文件大小超过 sst-max-size 时，该文件会被备份到多个Region 范围中，分别为Region [a,b)、[b,c)、[c,d) 和 [d,e)，并且 [a,b)、[b,c) 和 [c,d) 的大小均为 sst-max-size（或略大）。
- 默认值："144MB"

4.2.2.26.4 enable-auto-tune 从 v5.4 版本开始引入

- 在集群资源占用率较高的情况下，是否允许 BR 自动限制备份使用的资源，减少对集群的影响。详情见自动调节。
- 默认值：true

4.2.2.26.5 s3-multi-part-size 从 v5.3.2 版本开始引入

> 注意：
>
> 引入该配置项是为了解决备份期间遇到的 S3 限流导致备份失败的问题。该问题已通过优化 BR 备份数据存储的目录结构得到解决。因此，该配置项自 v6.1.1 起开始废弃，不再推荐使用。

- 备份阶段 S3 分块上传的块大小。可通过调整该参数来控制备份时发往 S3 的请求数量。
- TiKV 备份数据到 S3 时，如果备份文件大于该配置项的值，会自动进行分块上传。根据压缩率的不同，96 MiB Region 产生的备份文件大约在 10 MiB~30 MiB 之间。
- 默认值：5MiB

4.2.2.27 backup.hadoop

4.2.2.27.1 home

- 指定 HDFS shell 命令的位置，并且允许TiKV 找到该 shell 命令。该配置项与环境变量 `$HADOOP_HOME` 有相同的效果。
- 默认值：""

4.2.2.27.2 `linux-user`

- 指定 TiKV 运行 HDFS shell 命令的 Linux 用户。
- 如果未设置该配置项，TiKV 会使用当前 Linux 用户。
- 默认值：""

4.2.2.28 log-backup

用于日志备份相关的配置项。

4.2.2.28.1 `enable` 从 v6.2.0 版本开始引入

- 用于开启日志备份功能。
- 默认值：true

4.2.2.28.2 `file-size-limit` 从 v6.2.0 版本开始引入

- 日志备份任务中，保存到存储的备份文件大小。
- 默认值：256MiB
- 注意：一般情况下，`file-size-limit` 的值会大于外部存储上显示的备份文件大小，这是因为备份文件在上传时会被压缩。

4.2.2.28.3 `initial-scan-pending-memory-quota` 从 v6.2.0 版本开始引入

- 日志备份任务在扫描增量数据时，用于存放扫描数据的缓存大小。
- 默认值：`min(机器总内存 * 10%, 512 MB)`

4.2.2.28.4 `initial-scan-rate-limit` 从 v6.2.0 版本开始引入

- 日志备份任务在扫描增量数据时的吞吐限流参数。
- 默认值：60，即默认限流 60 MB/s

4.2.2.28.5 `max-flush-interval` 从 v6.2.0 版本开始引入

- 日志备份任务将备份数据写入到外部存储的最大间隔时间。
- 默认值：3min

4.2.2.28.6 `num-threads` 从 v6.2.0 版本开始引入

- 日志备份功能占用的线程数目。
- 默认值：CPU * 0.5
- 可调整范围：[2, 12]

4.2.2.28.7 `temp-path` 从 v6.2.0 版本开始引入

- 日志文件存放的临时目录，日志文件预先写入临时目录，然后 flush 到外部存储中。
- 默认值：`${deploy-dir}/data/log-backup-temp`

4.2.2.29 cdc

用于TiCDC 捕捉变更数据相关的配置项。

4.2.2.29.1 `min-ts-interval`

- 定期推进Resolved TS 的时间间隔。
- 默认值：200ms

4.2.2.29.2 `old-value-cache-memory-quota`

- 缓存在内存中的TiCDC Old Value 的条目占用内存的上限。
- 默认值：512MB

4.2.2.29.3 `sink-memory-quota`

- 缓存在内存中的TiCDC 数据变更事件占用内存的上限。
- 默认值：512MB

4.2.2.29.4 incremental-scan-speed-limit

- 增量扫描历史数据的速度上限。
- 默认值：128MB，即 128MB 每秒。

4.2.2.29.5 incremental-scan-threads

- 增量扫描历史数据任务的线程个数。
- 默认值：4，即 4 个线程

4.2.2.29.6 incremental-scan-concurrency

- 增量扫描历史数据任务的最大并发执行个数。
- 默认值：6，即最多并发执行 6 个任务
- 注意：incremental-scan-concurrency 需要大于等于 incremental-scan-threads，否则 TiKV 启动会报错。

4.2.2.30 resolved-ts

用于维护Resolved TS 以服务 Stale Read 请求的相关配置项。

4.2.2.30.1 enable

- 是否为所有Region 维护 Resolved TS。
- 默认值：true

4.2.2.30.2 advance-ts-interval

- 定期推进Resolved TS 的时间间隔。
- 默认值：20s

4.2.2.30.3 scan-lock-pool-size

- 初始化Resolved TS 时 TiKV 扫描 MVCC（多版本并发控制）锁数据的线程个数。
- 默认值：2，即 2 个线程

4.2.2.31 pessimistic-txn

悲观事务使用方法请参考TiDB 悲观事务模式。

4.2.2.31.1 wait-for-lock-timeout

- 悲观事务在TiKV 中等待其他事务释放锁的最长时间。若超时则会返回错误给TiDB 并由 TiDB 重试加锁，语句最长等锁时间由 innodb_lock_wait_timeout 控制。
- 默认值：1s
- 最小值：1ms

4.2.2.31.2 wake-up-delay-duration

- 悲观事务释放锁时，只会唤醒等锁事务中 start_ts 最小的事务，其他事务将会延迟 wake-up-delay-duration 之后被唤醒。
- 默认值：20ms

4.2.2.31.3 pipelined

- 开启流水线式加悲观锁流程。开启该功能后，TiKV 在检测数据满足加锁要求后，立刻通知TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但有较低概率出现悲观锁异步写入失败的情况，可能会导致悲观事务提交失败。
- 默认值：true

4.2.2.31.4 in-memory 从 v6.0.0 版本开始引入

- 开启内存悲观锁功能。开启该功能后，悲观事务会尽可能在TiKV 内存中存储悲观锁，而不将悲观锁写入磁盘，也不将悲观锁同步给其他副本，从而提升悲观事务的性能。但有较低概率出现悲观锁丢失的情况，可能会导致悲观事务提交失败。
- 默认值：true
- 注意：in-memory 仅在 pipelined 为true 时生效。

4.2.2.32 quota

用于请求限流(Quota Limiter) 相关的配置项。

4.2.2.32.1 max-delay-duration 从 v6.0.0 版本开始引入

- 单次读写请求被强制等待的最大时间。
- 默认值：500ms
- 推荐设置：一般使用默认值即可。如果实例出现了内存溢出或者是剧烈的性能抖动，可以设置为 1S，使得请求被延迟调节的时间不超过 1 秒。

4.2.2.32.2 前台限流

用于前台限流相关的配置项。

当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果TiKV 前台处理的读写请求量过大，以至于占用TiKV 后台处理请求所需的 CPU 资源，最终影响TiKV 性能的稳定性。此时，你可以使用前台限流相关的quota 配置项以限制前台各类请求占用的 CPU 资源。触发该限制的请求会被强制等待一段时间以让出 CPU 资源。具体等待时间与新增请求量相关，最多不超过max-delay-duration 的值。

foreground-cpu-time 从 v6.0.0 版本开始引入

- 限制处理TiKV 前台读写请求所使用的 CPU 资源使用量，这是一个软限制。
- 默认值：0（即无限制）

- 单位：millicpu （当该参数值为 1500 时，前端请求会消耗 1.5v CPU ）。
- 推荐设置：对于 4 核以上的实例，使用默认值 0 即可；对 4 核实例，设置为 1000 到 1500 之间的值能取得比较均衡的效果；对 2 核实例，则不要超过 1200。

`foreground-write-bandwidth` 从 v6.0.0 版本开始引入

- 限制前台事务写入的带宽，这是一个软限制。
- 默认值：0KB（即无限制）
- 推荐设置：除非因为 `foreground-cpu-time` 设置不足以对写带宽做限制，一般情况下本配置项使用默认值 0 即可；否则，在 4 核及 4 核以下规格实例上，建议设置在 50MB 以下。

`foreground-read-bandwidth` 从 v6.0.0 版本开始引入

- 限制前台事务读取数据和 Coprocessor 读取数据的带宽，这是一个软限制。
- 默认值：0KB（即无限制）
- 推荐设置：除非因为 `foreground-cpu-time` 设置不足以对读带宽做限制，一般情况本配置项使用默认值 0 即可；否则，在 4 核及 4 核以下规格实例上，建议设置在 20MB 以内。

4.2.2.32.3 后台限流

用于后台限流相关的配置项。

当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果TiKV 后台处理的计算或者读写请求量过大，以至于占用TiKV 前台处理请求所需的 CPU 资源，最终影响TiKV 性能的稳定性。此时，你可以使用后台限流相关的 quota 配置项以限制后台各类请求占用的 CPU 资源。触发该限制的请求会被强制等待一段时间以让出 CPU 资源。具体等待时间与新增请求量相关，最多不超过`max-delay-duration` 的值。

> 警告：
>
> - 后台限流是TiDB 在 v6.2.0 中引入的实验特性，不建议在生产环境中使用。
> - 该功能仅适合在资源有限的环境中使用，以保证TiKV 在该环境下可以长期稳定地运行。如果在资源丰富的机型环境中开启该功能，可能会导致读写请求量达到峰值时TiKV 的性能下降的问题。

`background-cpu-time` 从 v6.2.0 版本开始引入

- 限制处理TiKV 后台读写请求所使用的 CPU 资源使用量，这是一个软限制。
- 默认值：0（即无限制）
- 单位：millicpu（当该参数值为 1500 时，后端请求会消耗 1.5v CPU ）。

`background-write-bandwidth` 从 v6.2.0 版本开始引入

> 注意：
> 该配置项可以通过 SHOW CONFIG 查询到，但暂未生效。设置该配置项的值不生效。

- 限制后台事务写入的带宽，这是一个软限制。
- 默认值：0KB（即无限制）

background-read-bandwidth 从 v6.2.0 版本开始引入

> 注意：
> 该配置项可以通过 SHOW CONFIG 查询到，但暂未生效。设置该配置项的值不生效。

- 限制后台事务读取数据和 Coprocessor 读取数据的带宽，这是一个软限制。
- 默认值：0KB（即无限制）

enable-auto-tune 从 v6.2.0 版本开始引入

- 是否支持quota 动态调整。如果打开该配置项，TiKV 会根据TiKV 实例的负载情况动态调整对后台请求的限制quota。
- 默认值：false（即关闭动态调整）

4.2.2.33 causal-ts 从 v6.1.0 版本开始引入

用于TiKV API V2（storage.api-version = 2）中时间戳获取相关的配置项。

为了降低写请求延迟，TiKV 会定期获取一批时间戳缓存在本地，避免频繁访问 PD，并容忍短时间的 TSO 服务故障。

4.2.2.33.1 alloc-ahead-buffer 从 v6.4.0 版本开始引入

- 预分配TSO 缓存大小（以时长计算）。
- 表示TiKV 将按照这个参数指定的时长，预分配TSO 缓存。TiKV 会根据前一周期的使用情况，预估并请求满足 alloc-ahead-buffer 时长所需要的TSO 数量，缓存在本地。
- 这个参数通常用于提高TiKV API V2 (storage.api-version = 2) 对 PD 故障的容忍度。
- 调大这个参数会增加 TSO 消耗，并增加 TiKV 的内存开销。为了获得足够的 TSO，建议同时调小 PD 的tso-update-physical-interval 参数。
- 根据测试，默认配置下，当 PD 主节点由于故障切换到其节点时，写请求会短暂出现延迟增大和 QPS 下降（幅度约 15%）。
- 如果希望业务不受影响，可以尝试采用以下配置：

- causal-ts.alloc-ahead-buffer = "6s"
- causal-ts.renew-batch-max-size = 65536
- causal-ts.renew-batch-min-size = 2048
- 在 PD 中配置 tso-update-physical-interval = "1ms"

- 默认值：3s

4.2.2.33.2 renew-interval

- 更新本地缓存时间戳的周期。
- TiKV 会每隔 renew-interval 发起一次时间戳更新，并根据前一周期的使用情况以及 alloc-ahead-buffer 参数，来调整时间戳的缓存数量。这个参数配置过大会导致不能及时反映最新的TiKV 负载变化。而配置过小则会增加 PD 的负载。如果写流量剧烈变化、频繁出现时间戳耗尽、写延迟增加，可以适当调小这个参数，但需要同时关注 PD 的负载情况。
- 默认值：100ms

4.2.2.33.3 renew-batch-min-size

- 单次时间戳请求的最小数量。
- TiKV 会根据前一周期的使用情况以及 alloc-ahead-buffer 参数设置，来调整时间戳的缓存数量。如果 TSO 需求量较低，TiKV 会降低单次TSO 请求量，直至等于 renew-batch-min-size。如果业务中经常出现突发的大流量写入，可以适当调大这个参数。注意这个参数是单个tikv-server 的缓存大小，如果配置过大、而同时集群中tikv-server 较多，会导致TSO 消耗过快。
- Grafana TiKV-Raw 面板下 Causal timestamp 中的TSO batch size 是根据业务负载动态调整后的本地缓存数量。可以参考该监控指标值调整这个参数的大小。
- 默认值：100

4.2.2.33.4 renew-batch-max-size 从 v6.4.0 版本开始引入

- 单次时间戳请求的最大数量。
- 在默认的一个TSO 物理时钟更新周期内(50ms)，PD 最多提供 262144 个 TSO，超过这个数量后 PD 会暂缓 TSO 请求的处理。这个配置用于避免 PD 的TSO 消耗殆尽、影响其他业务的使用。如果增大这个参数，建议同时减小 PD 的tso-update-physical-interval 参数，以获得足够的TSO。
- 默认值：8192

4.2.2.34 resource-control

资源控制(Resource Control) 在 TiKV 存储层相关的配置项。

4.2.2.34.1 enabled 从 v6.6.0 版本开始引入

- 是否支持对用户前台的读写请求按照对应的资源组配额做优先级调度。有关TiDB 资源组和资源管控的信息，请参考TiDB 资源管控

- 在TiDB 侧开启tidb_enable_resource_control 全局变量的情况下，开启这个配置项才有意义。此配置参数开启后，TiKV 会使用优先级队列对排队的用户前台读写请求做调度，调度的优先级和请求所在资源组已经消费的资源量反相关，和对应资源组的配额正相关。
- 默认值：true（即开启按照资源组配额调度）

4.2.2.35 split

Load Base Split 相关的配置项。

4.2.2.35.1 byte-threshold 从 v5.0 版本开始引入

- 控制某个Region 被识别为热点Region 的流量阈值。
- 默认值：
  - 当region-split-size 小于 4 GB 时，默认值为每秒 30MiB 流量。
  - 当region-split-size 大于或等于 4 GB 时，默认值为每秒 100MiB 流量。

4.2.2.35.2 qps-threshold

- 控制某个Region 被识别为热点Region 的 QPS 阈值。
- 默认值：
  - 当region-split-size 小于 4 GB 时，默认值为每秒 3000 QPS。
  - 当region-split-size 大于或等于 4 GB 时，默认值为每秒 7000 QPS。

4.2.2.35.3 region-cpu-overload-threshold-ratio 从 v6.2.0 版本开始引入

- 控制某个Region 被识别为热点Region 的 CPU 使用率阈值。
- 默认值：
  - 当region-split-size 小于 4 GB 时，默认值为 0.25。
  - 当region-split-size 大于或等于 4 GB 时，默认值为 0.75。

4.2.3 TiFlash 配置参数

本文介绍了与部署使用TiFlash 相关的配置参数。

4.2.3.1 PD 调度参数

可通过pd-ctl 调整参数。如果你使用 TiUP 部署，可以用 `tiup ctl:v<CLUSTER_VERSION> pd` 代替 `pd-ctl -u <pd_ip:pd_port>` 命令。

- `replica-schedule-limit`：用来控制 replica 相关 operator 的产生速度（涉及到下线、补副本的操作都与该参数有关）

> 注意：
>
> 不要超过 `region-schedule-limit`，否则会影响正常 TiKV 之间的Region 调度。

- `store-balance-rate`：用于限制每个 TiKV store 或 TiFlash store 的 Region 调度速度。注意这个参数只对新加入集群的 store 有效，如果想立刻生效请用下面的方式。

> 注意：
>
> 4.0.2 版本之后（包括 4.0.2 版本）废弃了 `store-balance-rate` 参数且 `store limit` 命令有部分变化。该命令变化的细节请参考store-limit 文档。

```
- 使用 `pd-ctl -u <pd_ip:pd_port> store limit <store_id> <value>` 命令单独设置某个 store 的
    <-> Region 调度速度。（`store_id` 可通过 `pd-ctl -u <pd_ip:pd_port> store` 命令获得）
    <-> 如果没有单独设置，则继承 `store-balance-rate` 的设置。你也可以使用 `pd-ctl -u <pd_ip:
    <-> pd_port> store limit` 命令查看当前设置值。
```

- `replication.location-labels`：用来表示 TiKV 实例的拓扑关系，其中 key 的顺序代表了不同标签的层次关系。在TiFlash 开启的情况下需要使用`pd-ctl config placement-rules` 来设置默认值，详细可参考geo-distributed-deployment-topology。

4.2.3.2 TiFlash 配置参数

> Tip:
>
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

4.2.3.2.1 配置文件tiflash.toml

```
#### TiFlash TCP/HTTP 等辅助服务的监听 host。建议配置成 0.0.0.0，即监听本机所有 IP 地址。
listen_host = "0.0.0.0"
#### TiFlash TCP 服务的端口
tcp_port = 9000
#### 数据块元信息的内存 cache 大小限制，通常不需要修改
mark_cache_size = 5368709120
#### 数据块 min-max 索引的内存 cache 大小限制，通常不需要修改
minmax_index_cache_size = 5368709120
#### DeltaIndex 内存 cache 大小限制，默认为 0，代表没有限制
delta_index_cache_size = 0

#### TiFlash 数据的存储路径。如果有多个目录，以英文逗号分隔。
#### 从 v4.0.9 版本开始，不推荐使用 path 及 path_realtime_mode 参数。推荐使用 [storage]
    <↪ 下的配置项代替，这样在多盘部署的场景下能更好地利用节点性能。
#### 从 v5.2.0 版本开始，如果要使用配置项 storage.io_rate_limit，需要同时将 TiFlash
    <↪ 的数据存储路径设置为 storage.main.dir。
#### 当 [storage] 配置项存在的情况下，path 和 path_realtime_mode 两个配置会被忽略。
### path = "/tidb-data/tiflash-9000"
#### 或
### path = "/ssd0/tidb-data/tiflash,/ssd1/tidb-data/tiflash,/ssd2/tidb-data/tiflash"
#### 默认为 false。如果设为 true，且 path 配置了多个目录，表示在第一个目录存放最新数据，
    <↪ 在其他目录存放较旧的数据。
### path_realtime_mode = false

#### TiFlash 临时文件的存放路径。默认使用 [`path` 或者 `storage.latest.dir` 的第一个目录] + "/tmp
    <↪ "
### tmp_path = "/tidb-data/tiflash-9000/tmp"

#### 存储路径相关配置，从 v4.0.9 开始生效
[storage]

    ## DTFile 储存文件格式
    ## * format_version = 2 v6.0.0 以前版本的默认文件格式
    ## * format_version = 3 v6.0.0 及 v6.1.x 版本的默认文件格式，具有更完善的检验功能
    ## * format_version = 4 v6.2.0 及以后版本的默认文件格式，优化了写放大问题，
        <↪ 同时减少了后台线程消耗
    # format_version = 4

    [storage.main]
    ## 用于存储主要的数据，该目录列表中的数据占总数据的 90% 以上。
    dir = [ "/tidb-data/tiflash-9000" ]
    ## 或
    # dir = [ "/ssd0/tidb-data/tiflash", "/ssd1/tidb-data/tiflash" ]
```

```
## storage.main.dir 存储目录列表中每个目录的最大可用容量。
## * 在未定义配置项，或者列表中全填 0 时，会使用目录所在的硬盘容量
## * 以 byte 为单位。目前不支持如 "10GB" 的设置
## * capacity 列表的长度应当与 dir 列表长度保持一致
## 例如：
# capacity = [ 10737418240, 10737418240 ]

[storage.latest]
## 用于存储最新的数据，大约占总数据量的 10% 以内，需要较高的 IOPS。
## 默认情况该项可留空。在未配置或者为空列表的情况下，会使用 storage.main.dir 的值。
# dir = [ ]
## storage.latest.dir 存储目录列表中，每个目录的最大可用容量。
# capacity = [ 10737418240, 10737418240 ]

## [storage.io_rate_limit] 相关配置从 v5.2.0 开始引入。
[storage.io_rate_limit]
## 该配置项是 I/O 限流功能的开关，默认关闭。TiFlash 的 I/O
    <↪ 限流功能适用于磁盘带宽较小且磁盘带宽大小明确的云盘场景。
## I/O 限流功能限制下的读写流量总带宽，单位为 Byte，默认值为 0，即默认关闭 I/O 限流功能。
# max_bytes_per_sec = 0
## max_read_bytes_per_sec 和 max_write_bytes_per_sec 的含义和 max_bytes_per_sec 类似，分别指
    <↪ I/O 限流功能限制下的读流量总带宽和写流量总带宽。
## 分别用两个配置项控制读写带宽限制，适用于一些读写带宽限制分开计算的云盘，例如 Google Cloud
    <↪ 上的 persistent disk。
## 当 max_bytes_per_sec 配置不为 0 时，优先使用 max_bytes_per_sec。
# max_read_bytes_per_sec = 0
# max_write_bytes_per_sec = 0

## 下面的参数用于控制不同 I/O 流量类型分配到的带宽权重，一般不需要调整。
## TiFlash 内部将 I/O 请求分成 4 种类型：前台写、后台写、前台读、后台读。
## I/O 限流初始化时，TiFlash 会根据下面的权重 (weight) 比例分配带宽。
## 以下默认配置表示每一种流量将获得 25 / (25 + 25 + 25 + 25) = 25% 的权重。
## 如果将 weight 配置为 0，则对应的 I/O 操作不会被限流。
# foreground_write_weight = 25
# background_write_weight = 25
# foreground_read_weight = 25
# background_read_weight = 25
## TiFlash 支持根据当前的 I/O 负载情况自动调整各种 I/O 类型的限流带宽，有可能会超过设置的权重
    <↪ 。
## auto_tune_sec 表示自动调整的执行间隔，单位为秒。设为 0 表示关闭自动调整。
# auto_tune_sec = 5

## 下面的配置只针对存算分离模式生效，详细请参考 TiFlash 存算分离架构与 S3 支持文档 https://
    <↪ docs.pingcap.com/zh/tidb/dev/tiflash-disaggregated-and-s3
# [storage.s3]
```

```
    # endpoint: http://s3.{region}.amazonaws.com # S3 的 endpoint 地址
    # bucket: mybucket                          # TiFlash 的所有数据存储在这个 bucket 中
    # root: /cluster1_data                      # S3 bucket 中存储数据的根目录
    # access_key_id: {ACCESS_KEY_ID}            # 访问 S3 的 ACCESS_KEY_ID
    # secret_access_key: {SECRET_ACCESS_KEY}    # 访问 S3 的 SECRET_ACCESS_KEY

    # [storage.remote.cache]
    # dir: /data1/tiflash/cache         # TiFlash Compute Node 的本地数据缓存目录
    # capacity: 858993459200            # 800 GiB

[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

    ## 下面的配置只针对存算分离模式生效，详情请参考 TiFlash 存算分离架构与 S3 支持文档 https://
        <↪  docs.pingcap.com/zh/tidb/dev/tiflash-disaggregated-and-s3
    # disaggregated_mode = tiflash_write # 可选值为 tiflash_write 或者 tiflash_compute

### 多个 TiFlash 节点会选一个 master 来负责往 PD 增删 placement rule，通过 flash.flash_cluster
    <↪ 中的参数控制。
[flash.flash_cluster]
    refresh_interval = master 定时刷新有效期
    update_rule_interval = master 定时向 tidb 获取 TiFlash 副本状态并与 pd 交互
    master_ttl = master 选出后的有效期
    cluster_manager_path = pd buddy 所在目录的绝对路径
    log = pd buddy log 路径

[flash.proxy]
    addr = proxy 监听地址，不填则默认是 127.0.0.1:20170
    advertise-addr = 外部访问 addr 的地址，不填则默认是 "addr"
    data-dir = proxy 数据存储路径
    config = proxy 配置文件路径
    log-file = proxy log 路径
    log-level = proxy log 级别，默认是 "info"
    status-addr = 拉取 proxy metrics | status 信息的监听地址，不填则默认是 127.0.0.1:20292
    advertise-status-addr = 外部访问 status-addr 的地址，不填则默认是 "status-addr"

[logger]
    ## log 级别（支持 "trace"、"debug"、"info"、"warn"、"error"），默认是 "debug"
    level = "debug"
    log = TiFlash log 路径
    errorlog = TiFlash error log 路径
    ## 单个日志文件的大小，默认是 "100M"
    size = "100M"
    ## 最多保留日志文件个数，默认是 10
```

```
    count = 10

[raft]
    pd_addr = pd 服务地址 # 多个地址以逗号隔开

[status]
    ## Prometheus 拉取 metrics 信息的端口，默认是 8234
    metrics_port = 8234

[profiles]

[profiles.default]
    ## 存储引擎的 segment 分裂是否使用逻辑分裂。使用逻辑分裂可以减小写放大，
        <-> 但是会造成一定程度的硬盘空间回收不及时。默认为 false。
    ## 在 v6.2.0 以及后续版本，强烈建议保留默认值 `false`，不要将其修改为 `true`。
        <-> 具体请参考已知问题 [#5576](https://github.com/pingcap/tiflash/issues/5576)。
    # dt_enable_logical_split = false

    ## 单次查询过程中，节点对中间数据的内存限制
    ## 设置为整数时，单位为 byte，比如 34359738368 表示 32 GiB 的内存限制，0 表示无限制
    ## 设置为 [0.0, 1.0) 之间的浮点数时，指节点总内存的比值，比如 0.8 表示总内存的 80%，0.0
        <-> 表示无限制
    ## 默认值为 0，表示不限制
    ## 当查询试图申请超过限制的内存时，查询终止执行并且报错
    max_memory_usage = 0

    ## 所有查询过程中，节点对中间数据的内存限制
    ## 设置为整数时，单位为 byte，比如 34359738368 表示 32 GiB 的内存限制，0 表示无限制
    ## 设置为 [0.0, 1.0) 之间的浮点数时，指节点总内存的比值，比如 0.8 表示总内存的 80%，0.0
        <-> 表示无限制
    ## 默认值为 0.8，表示总内存的 80%
    ## 当查询试图申请超过限制的内存时，查询终止执行并且报错
    max_memory_usage_for_all_queries = 0.8

    ## 从 v5.0 引入，表示 TiFlash Coprocessor 最多同时执行的 cop 请求数量。
        <-> 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
        <-> 则使用默认值，即物理核数的两倍。
    cop_pool_size = 0

    ## 从 v5.0 引入，表示 TiFlash Coprocessor 最多同时执行的 batch 请求数量。
        <-> 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
        <-> 则使用默认值，即物理核数的两倍。
    batch_cop_pool_size = 0

    ## 从 v6.1 引入，指定 TiFlash 执行来自 TiDB 的 ALTER TABLE ... COMPACT 请求时，
```

```
    <↔ 能同时并行处理的请求数量。
    ## 如果这个值没有设置或设为了 0，则会采用默认值（1）。
    manual_compact_pool_size = 1

    ## 从 v5.4.0 引入，表示是否启用弹性线程池，这项功能可以显著提高 TiFlash 在高并发场景的 CPU
    <↔ 利用率。默认为 true。
    # enable_elastic_threadpool = true

    ## TiFlash 存储引擎的压缩算法，支持 LZ4、zstd 和 LZ4HC，大小写不敏感。默认使用 LZ4 算法。
    dt_compression_method = "LZ4"

    ## TiFlash 存储引擎的压缩级别，默认为 1。
    ## 如果 dt_compression_method 设置为 LZ4，推荐将该值设为 1；
    ## 如果 dt_compression_method 设置为 zstd，推荐将该值设为 -1 或 1，设置为 -1 的压缩率更小，
    <↔ 但是读性能会更好；
    ## 如果 dt_compression_method 设置为 LZ4HC，推荐将该值设为 9。
    dt_compression_level = 1

    ## 从 v6.2.0 引入，表示 PageStorage 单个数据文件中有效数据的最低比例。
    <↔ 当某个数据文件的有效数据比例低于该值时，会触发 GC 对该文件的数据进行整理。默认为
    <↔ 0.5。
    dt_page_gc_threshold = 0.5

    ## 从 v7.0.0 引入，表示带 group by key 的 HashAggregation 算子在触发 spill 之前的最大可用内存
    <↔ ，超过该阈值之后 HashAggregation 会采用 spill to disk 的方式来减小内存使用。默认值为
    <↔ 0，表示内存使用无限制，即不会触发 spill。
    max_bytes_before_external_group_by = 0

    ## 从 v7.0.0 引入，表示 sort/topN 算子在触发 spill 之前的最大可用内存，超过该阈值之后 sort/
    <↔ TopN 会采用 spill to disk 的方式来减小内存使用。默认值为 0，表示内存使用无限制，
    <↔ 即不会触发 spill。
    max_bytes_before_external_sort = 0

    ## 从 v7.0.0 引入，表示带等值 join 条件的 HashJoin 算子在触发 spill 之前的最大可用内存，
    <↔ 超过该阈值之后 HashJoin 算子会采用 spill to disk 的方式来减小内存使用。默认值为 0，
    <↔ 表示内存使用无限制，即不会触发 spill。
    max_bytes_before_external_join = 0

#### 安全相关配置，从 v4.0.5 开始生效
[security]
    ## 从 v5.0 引入，控制是否开启日志脱敏
    ## 若开启该选项，日志中的用户数据会以 `?` 代替显示
    ## 注意，tiflash-learner 对应的安全配置选项为 `security.redact-info-log`，需要在 tiflash-
    <↔ learner.toml 中另外开启
    # redact_info_log = false
```

```
    ## 包含可信 SSL CA 列表的文件路径。如果你设置了该值，`cert_path` 和 `key_path`
        <↔ 中的路径也需要填写
    # ca_path = "/path/to/ca.pem"
    ## 包含 PEM 格式的 X509 certificate 文件路径
    # cert_path = "/path/to/tiflash-server.pem"
    ## 包含 PEM 格式的 X509 key 文件路径
    # key_path = "/path/to/tiflash-server-key.pem"
```

4.2.3.2.2 配置文件tiflash-learner.toml

```
[server]
    engine-addr = 外部访问 TiFlash coprocessor 服务的地址

[raftstore]
    ## 处理 Raft 数据落盘的线程池中线程的数量
    apply-pool-size = 4
    ## 处理 Raft 的线程池中线程的数量，即 Raftstore 线程池的大小。
    store-pool-size = 4
    ## 控制处理 snapshot 的线程数，默认为 2。设为 0 则关闭多线程优化
    snap-handle-pool-size = 2
    ## 控制 raft store 持久化 WAL 的最小间隔。通过适当增大延迟以减少 IOPS 占用，默认为 "4ms"，
        <↔ 设为 "0ms" 则关闭该优化。
    store-batch-retry-recv-timeout = "4ms"

[security]
    ## 从 v5.0 引入，控制是否开启日志脱敏
    ## 若开启该选项，日志中的用户数据会以 `?` 代替显示
    ## 默认值为 false
    redact-info-log = false

[security.encryption]
    ## 数据文件的加密方法。
    ## 可选值为 "aes128-ctr"、"aes192-ctr"、"aes256-ctr"、"sm4-ctr"（仅 v6.4.0 及之后版本）和 "
        <↔ plaintext"。
    ## 默认值为 "plaintext"，即默认不开启加密功能。选择 "plaintext" 以外的值则表示启用加密功能。
        <↔ 此时必须指定主密钥。
    data-encryption-method = "aes128-ctr"
    ## 轮换密钥的频率，默认值：`7d`。
    data-key-rotation-period = "168h" # 7 days

[security.encryption.master-key]
    ## 指定启用加密时的主密钥。若要了解如何配置主密钥，可以参考《静态加密 - 配置加密》：https://
        <↔ docs.pingcap.com/zh/tidb/dev/encryption-at-rest#配置加密
```

```
[security.encryption.previous-master-key]
    ## 指定轮换新主密钥时的旧主密钥。旧主密钥的配置格式与主密钥相同。若要了解如何配置主密钥，
        <↪ 可以参考《静态加密 - 配置加密》：https://docs.pingcap.com/zh/tidb/dev/encryption-at-
        <↪ rest#配置加密
```

除以上几项外，其余功能参数和TiKV的配置相同。需要注意的是：key 为 engine 的 label 是保留项，不可手动配置。

4.2.3.2.3 通过拓扑label进行副本调度

TiFlash 设置可用区

4.2.3.2.4 多盘部署

TiFlash 支持单节点多盘部署。如果你的部署节点上有多块硬盘，可以通过以下的方式配置参数，提高节点的硬盘I/O利用率。TiUP 中参数配置格式参照详细 TiFlash 配置模版。

TiDB 集群版本低于 v4.0.9

TiDB v4.0.9 之前的版本中，TiFlash 只支持将存储引擎中的主要数据分布在多盘上。通过path（TiUP 中为data_dir）和 path_realtime_mode 这两个参数配置多盘部署。

多个数据存储目录在 path 中以英文逗号分隔，比如 /nvme_ssd_a/data/tiflash,/sata_ssd_b/data/tiflash,/<↪ sata_ssd_c/data/tiflash。如果你的节点上有多块硬盘，推荐把性能最好的硬盘目录放在最前面，以更好地利用节点性能。

如果节点上有多块相同规格的硬盘，可以把 path_realtime_mode 参数留空（或者把该值明确地设为 false）。这表示数据会在所有的存储目录之间进行均衡。但由于最新的数据仍然只会被写入到第一个目录，因此该目录所在的硬盘会较其他硬盘繁忙。

如果节点上有多块规格不一致的硬盘，推荐把 path_relatime_mode 参数设置为 true，并且把性能最好的硬盘目录放在 path 参数内的最前面。这表示第一个目录只会存放最新数据，较旧的数据会在其他目录之间进行均衡。注意此情况下，第一个目录规划的容量大小需要占总容量的约 10%。

TiDB 集群版本为 v4.0.9 及以上

TiDB v4.0.9 及之后的版本中，TiFlash 支持将存储引擎的主要数据和新数据都分布在多盘上。多盘部署时，推荐使用 [storage] 中的参数，以更好地利用节点的I/O性能。但TiFlash 仍然支持TiDB 集群版本低于 v4.0.9 中的参数。

如果节点上有多块相同规格的硬盘，推荐把硬盘目录填到 storage.main.dir 列表中，storage.latest.dir 列表留空。TiFlash 会在所有存储目录之间分摊I/O压力以及进行数据均衡。

如果节点上有多块规格不同的硬盘，推荐把I/O 性能较好的硬盘目录配置在 storage.latest.dir 中，把I/O性能较一般的硬盘目录配置在 storage.main.dir 中。例如节点上有一块 NVMe-SSD 硬盘加上两块 SATA-SSD 硬盘，你可以把 storage.latest.dir 设为 ["/nvme_ssd_a/data/tiflash"] 以及把 storage.main.dir 设为 ["/<↪ sata_ssd_b/data/tiflash", "/sata_ssd_c/data/tiflash"]。TiFlash 会根据两个目录列表分别进行I/O 压力分摊及数据均衡。需要注意此情况下，storage.latest.dir 中规划的容量大小需要占总规划容量的约 10%。

> **警告：**
>
> `[storage]` 参数从TiUP v1.2.5 版本开始支持。如果你的TiDB 版本为 v4.0.9 及以上，请确保你的 TiUP 版本不低于 v1.2.5，否则 `[storage]` 中定义的数据目录不会被TiUP 纳入管理。

4.2.4 PD 配置文件描述

PD 配置文件比命令行参数支持更多的选项。你可以在conf/config.toml 找到默认的配置文件。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见PD 配置参数。

> Tip:
>
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

4.2.4.0.1 name

- PD 节点名称。
- 默认值：`"pd"`
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字。

4.2.4.0.2 data-dir

- PD 存储数据路径。
- 默认值：`"default.${name}"`

4.2.4.0.3 client-urls

- PD 监听的客户端 URL 列表。
- 默认值：`"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"`。

4.2.4.0.4 advertise-client-urls

- 用于外部访问 PD 的 URL 列表。
- 默认值：`"${client-urls}"`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。

- 例如，Docker 内部IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 2379:2379，那么可以设置为 advertise-client-urls="http://192.168.100.113:2379"，客户端可以通过 http://192.168.100.113:2379 来找到这个服务。

4.2.4.0.5 peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认："http://127.0.0.1:2380"
- 如果部署一个集群，peer URLs 必须指定当前主机的IP 地址，例如 "http://192.168.100.113:2380"，如果是运行在 Docker 则需要指定为 "http://0.0.0.0:2380"。

4.2.4.0.6 advertise-peer-urls

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认值："${peer-urls}"
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问
- 例如，Docker 内部IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 2380:2380，那么可以设置为 advertise-peer-urls="http://192.168.100.113:2380"，其他 PD 节点可以通过 http://192.168.100.113:2380 来找到这个服务。

4.2.4.0.7 initial-cluster

- 初始化 PD 集群配置。
- 默认值："{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。

4.2.4.0.8 initial-cluster-state

- 集群初始状态
- 默认值：“new”

4.2.4.0.9 initial-cluster-token

- 用于在集群初始化阶段标识不同的集群。
- 默认值：“pd-cluster”
- 如果先后部署多个集群，且多个集群有相同配置的节点，应指定不同的token 来隔离不同的集群。

4.2.4.0.10 lease

- PD Leader Key 租约超时时间，超时系统重新选举 Leader。
- 默认值：3
- 单位：秒

4.2.4.0.11 quota-backend-bytes

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

4.2.4.0.12 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）。
- 默认值：periodic

4.2.4.0.13 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。
- 默认值：1h

4.2.4.0.14 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。
- 默认值：false

4.2.4.0.15 tso-update-physical-interval

- TSO 物理时钟更新周期。
- 在默认的一个TSO 物理时钟更新周期内(50ms)，PD 最多提供 262144 个 TSO。如果需要更多的 TSO，可以将这个参数调小。最小值为 1ms。
- 缩短这个参数会增加 PD 的 CPU 消耗。根据测试，相比 50ms 更新周期，更新周期为 1ms 时，PD 的 CPU 占用率(CPU usage) 将增加约 10%。
- 默认值：50ms
- 最小值：1ms

4.2.4.1 pd-server

pd-server 相关配置项。

4.2.4.1.1 server-memory-limit 从 v6.6.0 版本开始引入

> **警告：**
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- PD 实例的内存限制比例。0 值表示不设内存限制。
- 默认值：0
- 最小值：0
- 最大值：0.99

4.2.4.1.2 server-memory-limit-gc-trigger 从 v6.6.0 版本开始引入

> **警告：**
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- PD 尝试触发GC 的阈值比例。当 PD 的内存使用达到 server-memory-limit 值* server-memory-limit-gc-trigger 值时，则会主动触发一次Golang GC。在一分钟之内只会主动触发一次GC。
- 默认值：0.7
- 最小值：0.5
- 最大值：0.99

4.2.4.1.3 enable-gogc-tuner 从 v6.6.0 版本开始引入

> **警告：**
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- 是否开启GOGC Tuner。
- 默认值：false

4.2.4.1.4 gc-tuner-threshold 从 v6.6.0 版本开始引入

> **警告：**
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- GOGC Tuner 自动调节的最大内存阈值比例， 即 server-memory-limit 值 * server-memory-limit-gc-trigger 值，超过阈值后GOGC Tuner 会停止工作。
- 默认值：0.6
- 最小值：0
- 最大值：0.9

4.2.4.1.5 flow-round-by-digit 从 v5.1 版本开始引入

- 默认值：3
- PD 会对流量信息的末尾数字进行四舍五入处理，减少Region 流量信息变化引起的统计信息更新。该配置项用于指定对Region 流量信息的末尾进行四舍五入的位数。例如流量 100512 会归约到 101000。默认值为 3。该配置替换了 trace-region-flow。

> **注意：**
>
> 如果是从 v4.0 升级至当前版本，升级后的 flow-round-by-digit 行为和升级前的 trace-region-flow 行为默认保持一致：如果升级前 trace-region-flow 为 false，则升级后 flow-round-by-digit 为 127；如果升级前 trace-region-flow 为true，则升级后 flow-round-by-digit 为 3。

4.2.4.2 security

安全相关配置项。

4.2.4.2.1 cacert-path

- CA 文件路径
- 默认值： “”

4.2.4.2.2 cert-path

- 包含X509 证书的PEM 文件路径
- 默认值： “”

4.2.4.2.3 key-path

- 包含X509 key 的 PEM 文件路径
- 默认值： “”

4.2.4.2.4 `redact-info-log` 从 v5.0 版本开始引入

- 控制 PD 日志脱敏的开关
- 该配置项值设为true 时将对 PD 日志脱敏，遮蔽日志中的用户信息。
- 默认值：false

4.2.4.3 log

日志相关的配置项。

4.2.4.3.1 `level`

- 指定日志的输出级别。
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”
- 默认值：“info”

4.2.4.3.2 `format`

- 日志格式。
- 可选值：“text”，“json”
- 默认值：“text”

4.2.4.3.3 `disable-timestamp`

- 是否禁用日志中自动生成的时间戳。
- 默认值：false

4.2.4.4 log.file

日志文件相关的配置项。

4.2.4.4.1 `max-size`

- 单个日志文件最大大小，超过该值系统自动切分成多个文件。
- 默认值：300
- 单位：MiB
- 最小值为 1

4.2.4.4.2 `max-days`

- 日志保留的最长天数。
- 如果未设置本参数或把本参数设置为默认值 0，PD 不清理日志文件。
- 默认：0

4.2.4.4.3 max-backups

- 日志文件保留的最大个数。
- 如果未设置本参数或把本参数设置为默认值 0，PD 会保留所有的日志文件。
- 默认：0

4.2.4.5 metric

监控相关的配置项。

4.2.4.5.1 interval

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

4.2.4.6 schedule

调度相关的配置项。

4.2.4.6.1 max-merge-region-size

- 控制Region Merge 的 size 上限，当Region Size 大于指定值时 PD 不会将其与相邻的Region 合并。
- 默认：20
- 单位：MiB

4.2.4.6.2 max-merge-region-keys

- 控制Region Merge 的 key 上限，当Region key 大于指定值时 PD 不会将其与相邻的Region 合并。
- 默认：200000

4.2.4.6.3 patrol-region-interval

- 控制replicaChecker 检查Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整
- 默认：10ms

4.2.4.6.4 split-merge-interval

- 控制对同一个Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被merge。
- 默认：1h

4.2.4.6.5 max-snapshot-count

- 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。
- 默认：64

4.2.4.6.6 max-pending-peer-count

- 控制单个 store 的pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的Region。
- 默认值：64

4.2.4.6.7 max-store-down-time

- PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。
- 默认值：30m

4.2.4.6.8 max-store-preparing-time 从 v6.1.0 版本开始引入

- 控制 store 上线阶段的最长等待时间。在 store 的上线阶段，PD 可以查询该 store 的上线进度。当超过该配置项指定的时间后，PD 会认为该 store 已完成上线，无法再次查询这个 store 的上线进度，但是不影响 Region 向这个新上线 store 的迁移。通常用户无需修改该配置项。
- 默认值：48h

4.2.4.6.9 leader-schedule-limit

- 同时进行leader 调度的任务个数。
- 默认值：4

4.2.4.6.10 region-schedule-limit

- 同时进行Region 调度的任务个数
- 默认值：2048

4.2.4.6.11 hot-region-schedule-limit

- 控制同时进行的hot Region 任务。该配置项独立于Region 调度。
- 默认值：4

4.2.4.6.12 hot-region-cache-hits-threshold

- 设置识别热点Region 所需的分钟数。只有当Region 处于热点状态持续时间超过此分钟数时，PD 才会参与热点调度。
- 默认值：3

4.2.4.6.13 replica-schedule-limit

- 同时进行replica 调度的任务个数。
- 默认值：64

4.2.4.6.14 merge-schedule-limit

- 同时进行的Region Merge 调度的任务，设置为 0 则关闭Region Merge。
- 默认值：8

4.2.4.6.15 high-space-ratio

- 设置 store 空间充裕的阈值。当节点的空间占用比例小于该阈值时，PD 调度时会忽略节点的剩余空间，主要根据实际数据量进行均衡。此配置仅在 region-score-formula-version = v1 时生效。
- 默认值：0.7
- 最小值：大于 0
- 最大值：小于 1

4.2.4.6.16 low-space-ratio

- 设置 store 空间不足的阈值。当某个节点的空间占用比例超过该阈值时，PD 会尽可能避免往该节点迁移数据，同时主要根据节点剩余空间大小进行调度，避免对应节点的磁盘空间被耗尽。
- 默认值：0.8
- 最小值：大于 0
- 最大值：小于 1

4.2.4.6.17 tolerant-size-ratio

- 控制balance 缓冲区大小。
- 默认值：0（为 0 为自动调整缓冲区大小）
- 最小值：0

4.2.4.6.18 enable-cross-table-merge

- 设置是否开启跨表merge。
- 默认值：true

4.2.4.6.19 region-score-formula-version 从 v5.0 版本开始引入

- 设置Region 算分公式版本。
- 默认值：v2
- 可选值：v1，v2。v2 相比于 v1，变化会更平滑，空间回收引起的调度抖动情况会得到改善。

> 注意：
>
> 如果是从 v4.0 升级至当前版本，默认不自动开启该算分公式新版本，以保证升级前后 PD 行为一致。若想切换算分公式的版本，使用需要手动通过 pd-ctl 设置切换，详见PD Control 文档。

4.2.4.6.20 enable-joint-consensus 从 v5.0 版本开始引入

- 是否使用Joint Consensus 进行副本调度。关闭该特性时，PD 将采用一次调度一个副本的方式进行调度。
- 默认值：true

4.2.4.6.21 enable-diagnostic 从 v6.3.0 版本开始引入

- 是否开启诊断功能。开启特性时，PD 将会记录调度中的一些状态来帮助诊断。开启时会略微影响调度速度，在 Store 数量较多时会消耗较大内存。
- 默认值：true

4.2.4.6.22 hot-regions-write-interval 从 v5.4.0 版本开始引入

- 设置 PD 存储 Hot Region 信息时间间隔。
- 默认值：10m

> **注意：**
>
> Hot Region 的信息一般 3 分钟更新一次。如果设置时间间隔小于 3 分钟，中间部分的更新可能没有意义。

4.2.4.6.23 hot-regions-reserved-days 从 v5.4.0 版本开始引入

- 设置 PD 保留的 Hot Region 信息的最长时间。单位为天。
- 默认值: 7

4.2.4.7 replication

副本相关的配置项。

4.2.4.7.1 max-replicas

- 所有副本数量，即leader 与 follower 数量之和。默认为 3，即 1 个 leader 和 2 个 follower。当此配置被在线修改后，PD 会在后台通过调度使得Region 的副本数量符合配置。
- 默认值：3

4.2.4.7.2 location-labels

- TiKV 集群的拓扑信息。
- 默认值：[]
- 配置集群拓扑

4.2.4.7.3 isolation-level

- TiKV 集群的最小强制拓扑隔离级别。
- 默认值：“”
- 配置集群拓扑

4.2.4.7.4 strictly-match-label

- 打开强制TiKV Label 和 PD 的 location-labels 是否匹配的检查
- 默认值：false

4.2.4.7.5 enable-placement-rules

- 打开 placement-rules
- 默认值：true
- 参考Placement Rules 使用文档

4.2.4.7.6 store-limit-version 从 v7.1.0 版本开始引入

> 警告：
> 在当前版本中，将该配置项设置为 "v2" 为实验特性，不建议在生产环境中使用。

- 设置 store limit 工作模式
- 默认值：v1
- 可选值：
  - v1：在 v1 模式下，你可以手动修改 store limit 以限制单个TiKV 调度速度。
  - v2：（ 实验特性 ）在 v2 模式下，你无需关注 store limit 值，PD 将根据TiKV Snapshot 执行情况动态调整 TiKV 调度速度。详情请参考Store Limit v2 原理。

4.2.4.8 label-property

标签相关的配置项。

4.2.4.8.1 key

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

4.2.4.8.2 value

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

4.2.4.9 dashboard

PD 中内置的TiDB Dashboard 相关配置项。

4.2.4.9.1 tidb-cacert-path

- CA 根证书文件路径。可配置该路径来使用 TLS 连接TiDB 的 SQL 服务。
- 默认值：“”

4.2.4.9.2 tidb-cert-path

- SSL 证书文件路径。可配置该路径来使用 TLS 连接TiDB 的 SQL 服务。
- 默认值：“”

4.2.4.9.3 tidb-key-path

- SSL 私钥文件路径。可配置该路径来使用 TLS 连接TiDB 的 SQL 服务。
- 默认值：“”

4.2.4.9.4 public-path-prefix

- 通过反向代理访问TiDB Dashboard 时，配置反向代理提供服务的路径前缀。
- 默认值：“/dashboard”
- 若不通过反向代理访问TiDB Dashboard，请勿配置该项，否则可能导致 TiDB Dashboard 无法正常访问。关于该配置的详细使用场景，参见通过反向代理使用TiDB Dashboard。

4.2.4.9.5 enable-telemetry

- 是否启用TiDB Dashboard 遥测功能。
- 默认值：false
- 参阅遥测了解该功能详情。

4.2.4.10 replication-mode

Region 同步模式相关的配置项。更多详情，请参阅启用自适应同步模式。

4.2.4.11 Controllor

PD 中内置的Resource Control 相关的配置项。

4.2.4.11.1 degraded-mode-wait-duration

- 触发降级模式需要等待的时间。降级模式是指在 Local Token Bucket (LTB) 和 Global Token Bucket (GTB) 失联的情况下，LTB 将回退到默认的资源组配置，不再有GTB 授权token，从而保证在网络隔离或者异常情况下，服务不受影响。
- 默认值: 0s
- 默认为不开启降级模式

4.2.4.11.2 request-unit

下面是Request Unit (RU) 相关的配置项。

`read-base-cost`

- 每次读请求转换成RU 的基准系数
- 默认值: 0.25

`write-base-cost`

- 每次写请求转换成RU 的基准系数
- 默认值: 1

`read-cost-per-byte`

- 读流量转换成RU 的基准系数
- 默认值: 1/(64 * 1024)
- 1 RU = 64 KiB 读取字节

`write-cost-per-byte`

- 写流量转换成RU 的基准系数
- 默认值: 1/1024
- 1 RU = 1 KiB 写入字节

`read-cpu-ms-cost`

- CPU 转换成RU 的基准系数
- 默认值: 1/3
- 1 RU = 3 毫秒 CPU 时间

4.3 CLI

4.3.1 TiKV Control 使用说明

TiKV Control（以下简称 tikv-ctl）是TiKV 的命令行工具，用于管理TiKV 集群。它的安装目录如下：

- 如果是使用TiUP 部署的集群，在 `~/.tiup/components/ctl/{VERSION}/` 目录下。

4.3.1.1 通过TiUP 使用TiKV Control

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

tikv-ctl 也集成在了 tiup 命令中。执行以下命令，即可调用 tikv-ctl 工具：

```
tiup ctl:v<CLUSTER_VERSION> tikv
```

```
Starting component `ctl`: /home/tidb/.tiup/components/ctl/v4.0.8/ctl tikv
TiKV Control (tikv-ctl)
Release Version:   4.0.8
Edition:           Community
Git Commit Hash:   83091173e960e5a0f5f417e921a0801d2f6635ae
Git Commit Branch: heads/refs/tags/v4.0.8
UTC Build Time:    2020-10-30 08:40:33
Rust Version:      rustc 1.42.0-nightly (0de96d37f 2019-12-19)
Enable Features:   jemalloc mem-profiling portable sse protobuf-codec
Profile:           dist_release

A tool for interacting with TiKV deployments.

USAGE:
    TiKV Control (tikv-ctl) [FLAGS] [OPTIONS] [SUBCOMMAND]

FLAGS:
    -h, --help                    Prints help information
        --skip-paranoid-checks    Skip paranoid checks when open rocksdb
    -V, --version                 Prints version information


OPTIONS:
        --ca-path <ca-path>                    Set the CA certificate path
        --cert-path <cert-path>                Set the certificate path
        --config <config>                      TiKV config path, by default it's <deploy-dir>/conf/tikv
            <⟷ .toml
        --data-dir <data-dir>                  TiKV data directory path, check <deploy-dir>/scripts/run
            <⟷ .sh to get it
        --decode <decode>                      Decode a key in escaped format
        --encode <encode>                      Encode a key in escaped format
        --to-hex <escaped-to-hex>              Convert an escaped key to hex key
        --to-escaped <hex-to-escaped>          Convert a hex key to escaped key
        --host <host>                          Set the remote host
        --key-path <key-path>                  Set the private key path
```

```
        --log-level <log-level>          Set the log level [default:warn]
        --pd <pd>                        Set the address of pd

SUBCOMMANDS:
    bad-regions            Get all regions with corrupt raft
    cluster                Print the cluster id
    compact                Compact a column family in a specified range
    compact-cluster        Compact the whole cluster in a specified range in one or more column
        <⟷ families
    consistency-check      Force a consistency-check for a specified region
    decrypt-file           Decrypt an encrypted file
    diff                   Calculate difference of region keys from different dbs
    dump-snap-meta         Dump snapshot meta file
    encryption-meta        Dump encryption metadata
    fail                   Inject failures to TiKV and recovery
    help                   Prints this message or the help of the given subcommand(s)
    metrics                Print the metrics
    modify-tikv-config     Modify tikv config, eg. tikv-ctl --host ip:port modify-tikv-config -n
                           rocksdb.defaultcf.disable-auto-compactions -v true
                           Print the mvcc value
    mvcc                   Print the raw value
    print                  Print a raft log entry
    raft                   Print all raw keys in the range
    raw-scan               Recover mvcc data on one node by deleting corrupted keys
    recover-mvcc           Recreate a region with given metadata, but alloc new id for it
    recreate-region        Show region properties
    region-properties      Print the range db range
    scan                   Print region size
    size                   Split the region
    split-region           Print the store id
    store                  Set some regions on the node to tombstone by manual
    tombstone              Unsafely recover the cluster when the majority replicas are failed
    unsafe-recover
```

你可以在 `tiup ctl:v<CLUSTER_VERSION> tikv` 后面再接上相应的参数与子命令。

4.3.1.2 通用参数

tikv-ctl 提供以下两种运行模式：

- 远程模式。通过 `--host` 选项接受TiKV 的服务地址作为参数。在此模式下，如果 TiKV 启用了 SSL，则 tikv-ctl 也需要指定相关的证书文件，例如：

```
tikv-ctl --ca-path ca.pem --cert-path client.pem --key-path client-key.pem --host
    <⟷ 127.0.0.1:20160 <subcommands>
```

  某些情况下，tikv-ctl 与 PD 进行通信，而不与TiKV 通信。此时你需要使用 `--pd` 选项而非 `--host` 选项，例如：

```
tikv-ctl --pd 127.0.0.1:2379 compact-cluster
```

```
store:"127.0.0.1:20160" compact db:KV cf:default range:([], []) success!
```

- 本地模式：
  - 通过 `--data-dir` 选项来指定本地TiKV 数据的目录路径。
  - 通过 `--config` 选项来指定本地TiKV 配置文件到路径。

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。

4.3.1.3 各项子命令及部分参数、选项

下面逐一对tikv-ctl 支持的子命令进行举例说明。有的子命令支持很多可选参数，要查看全部细节，可运行 `tikv-ctl --help <subcommand>`。

4.3.1.3.1 查看Raft 状态机的信息

`raft` 子命令可以查看Raft 状态机在某一时刻的状态。状态信息包括RegionLocalState、 RaftLocalState 和 RegionApplyState 三个结构体，及某一条log 对应的Entries。

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。

对于 `region` 子命令：

- 要查看指定的Region，可在命令中使用 `-r` 参数，多个Region 以 , 分隔。也可以使用 `--all-regions` 参数来返回所有Region（`-r` 与 `--all-regions` 不能同时使用）

- 要限制输出的Region 的数量，可在命令中使用 `--limit` 参数（默认为 16）
- 要查询某个 key 范围中包含哪些Region，可在命令中使用 `--start` 和 `--end` 参数（默认不限范围，采用 Hex 格式）

需要输出ID 为 1239 的 Region 时，用法及输出内容如下所示：

```
tikv-ctl --host 127.0.0.1:20160 raft region -r 1239
```

```
"region id": 1239
 "region state":
   { id: 1239,
    start_key: 7480000000000000FF4E5F728000000000FF1443770000000000FA,
    end_key: 7480000000000000FF4E5F728000000000FF21C4420000000000FA,
    region_epoch: {conf_ver: 1 version: 43},
    peers: [ {id: 1240 store_id: 1 role: Voter} ]
}
"raft state": {
    hard_state {term: 8 vote: 5 commit: 7}
    last_index: 8)
}
"apply state": {
    applied_index: 8 commit_index: 8 commit_term: 8
    truncated_state {index: 5 term: 5}
}
```

需要查询某个 key 范围中包含哪些Region 时，用法及输出内容如下所示：

- 当 key 范围包含在某个Region 中时，将会输出该Region 信息。
- 当 key 范围精准到某个Region 的范围时，以上述Region 1239 为例：当给定的 key 范围为Region 1239 的范围时，由于Region 范围为左闭右开区间，并且Region 1009 以 Region 1239 的 end_key 作为 `start_key`，因此会同时输出Region 1009 和 Region 1239 的信息。

```
 tikv-ctl --host 127.0.0.1:20160 raft region --start 7480000000000000
     <⟷ FF4E5F728000000000FF1443770000000000FA --end 7480000000000000
    <⟷ FF4E5F728000000000FF21C4420000000000FA
```

```
"region state":
    { id: 1009
    start_key: 7480000000000000FF4E5F728000000000FF21C4420000000000FA,
    end_key: 7480000000000000FF5000000000000000F8,
    ...
}
"region state":
    { id: 1239
    start_key: 7480000000000000FF4E5F728000000000FF06C6D60000000000FA,
```

```
    end_key: 7480000000000000FF4E5F728000000000FF1443770000000000FA,
    ...
}
```

4.3.1.3.2 查看 Region 的大小

使用 `size` 命令可以查看Region 的大小：

```
tikv-ctl --data-dir /path/to/tikv 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
```

4.3.1.3.3 扫描查看给定范围的 MVCC

`scan` 命令的 `--from` 和 `--to` 参数接受两个 escaped 形式的raw key，并用 `--show-cf` 参数指定只需要查看哪些列族。

```
tikv-ctl --data-dir /path/to/tikv 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"
```

4.3.1.3.4 查看给定 key 的 MVCC

与 `scan` 命令类似，`mvcc` 命令可以查看给定 key 的 MVCC：

```
tikv-ctl --data-dir /path/to/tikv 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。

4.3.1.3.5 扫描raw key

使用 `raw-scan` 命令，TiKV 可直接在RocksDB 中扫描raw key。

> **注意：**
>
> 如果要扫描数据 key，需要在 key 前添加 `'z'` 前缀。

- 要指定扫描范围，可在 `raw-scan` 命令中使用 `--from` 和 `--to` 参数（默认不限范围）
- 要限制能够打印出的 key 的数量（默认为 `30`），可在命令中使用 `--limit` 参数
- 要指定扫描的 CF，可在命令中使用 `--cf` 参数（可选值为 `default`，`write`，`lock`）

```
tikv-ctl --data-dir /var/lib/tikv raw-scan --from 'zt' --limit 2 --cf default
```

```
key: "zt\200\000\000\000\000\000\000\377\005_r
    <⟷ \200\000\000\000\000\377\000\000\001\000\000\000\000\000\372\372b2,^\033\377\364", value:
    <⟷ "\010\002\002\002%\010\004\002\010root\010\006\002\000\010\010\t\002\010\n\t
    <⟷ \002\010\014\t\002\010\016\t\002\010\020\t\002\010\022\t\002\010\024\t\002\010\026\t
    <⟷ \002\010\030\t\002\010\032\t\002\010\034\t\002\010\036\t\002\010 \t\002\010\"\t\002\010s\
    <⟷ t\002\010&\t\002\010(\t\002\010*\t\002\010,\t\002\010.\t\002\0100\t\002\0102\t\002\0104\t
    <⟷ \002"
key: "zt\200\000\000\000\000\000\000\377\025_r
    <⟷ \200\000\000\000\000\377\000\000\023\000\000\000\000\000\372\372b2,^\033\377\364", value:
    <⟷  "\010\002\002&slow_query_log_file\010\004\002P/usr/local/mysql/data/localhost-slow.log"

Total scanned keys: 2
```

4.3.1.3.6 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。

4.3.1.3.7 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。

4.3.1.3.8 手动compact单个TiKV的数据

`compact` 命令可以对单个TiKV进行手动compact。

- `--from` 和 `--to` 选项以 escaped raw key 形式指定compact的范围。如果没有设置，表示compact整个TiKV。
- `--region` 选项指定compact Region 的范围。如果设置，则 `--from` 和 `--to` 选项会被忽略。
- `--db` 选项可以指定要compact的 RocksDB，可选值为 `kv` 和 `raft`。
- `--threads` 选项可以指定compact的并发数，默认值是 8。一般来说，并发数越大，compact的速度越快，但是也会对服务造成影响，所以需要根据情况选择合适的并发数。
- `--bottommost` 选项可以指定compact是否包括最下层的文件。可选值为 `default`、`skip` 和 `force`，默认为 `default`。
  - `default` 表示只有开启了 Compaction Filter 时 compact 才会包括最下层文件。
  - `skip` 表示compact不包括最下层文件。
  - `force` 表示compact总是包括最下层文件。
- 在本地模式compact data，执行如下命令：

```
tikv-ctl --data-dir /path/to/tikv compact --db kv
```

- 在远程模式compact data，执行如下命令：

```
tikv-ctl --host ip:port compact --db kv
```

4.3.1.3.9 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。

4.3.1.3.10 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 --data-dir /path/to/tikv 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 --data-dir /path/to/tikv 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。

4.3.1.3.11 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：

- 目前consistency-check 与 TiDB GC 操作不兼容，存在误报错误的可能，因此不建议使用该命令。
- 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。

4.3.1.3.12 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。

4.3.1.3.13 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 --data-dir /path/to/tikv 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。

4.3.1.3.14 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 --data-dir /path/to/tikv/data 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
  ```

4.3.1.3.15 动态修改TiKV 的配置

使用 `modify-tikv-config` 命令可以动态修改配置参数。目前可动态修改的TiKV 配置与具体的修改行为与 SQL 动态修改配置功能相同，可参考在线修改TiKV 配置。

- `-n` 用于指定完整的配置名。支持动态修改的配置名可以参考在线修改TiKV 配置中支持的配置项列表。
- `-v` 用于指定配置值。

设置 shared block cache 的大小：

```
tikv-ctl --host ip:port modify-tikv-config -n storage.block-cache.capacity -v 10GB
```

```
success
```

当禁用 shared block cache 时，为 write CF 设置 block cache size：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.writecf.block-cache-size -v 256MB
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftdb.defaultcf.disable-auto-compactions -v true
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftstore.sync-log -v false
```

```
success
```

如果compaction 的流量控制导致待compact 数据量(compaction pending bytes) 堆积，可以禁用 rate-limiter-auto <—> -tuned 配置项或调高compaction 相关的流量阈值。示例如下：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.rate-limiter-auto-tuned -v false
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.rate-bytes-per-sec -v "1GB"
```

```
success
```

4.3.1.3.16 强制Region 从多副本失败状态恢复服务（弃用）

> 警告：
>
> 不推荐使用该功能，恢复需求可通过 pd-ctl 的 Online Unsafe Recovery 功能实现。它提供了一键式自动恢复的能力，无需停止服务等额外操作，具体使用方式请参考 Online Unsafe Recovery 使用文档。

unsafe-recover remove-fail-stores 命令可以将故障机器从指定Region 的 peer 列表中移除。运行命令之前，需要目标TiKV 先停掉服务以便释放文件锁。

-s 选项接受多个以逗号分隔的 store_id，并使用 -r 参数来指定包含的Region。如果要对某一个 store 上的全部 Region 都执行这个操作，可简单指定 --all-regions。

警告：

- 此功能使用不当可能导致集群难以恢复，存在风险。请悉知潜在的风险，尽量避免在生产环境中使用。
- 如果使用 `--all-regions`，必须在剩余所有连入集群的 store 上执行此命令。需要保证这些健康的 store 都停掉服务后再进行恢复，否则期间Region 副本之间的peer 列表不一致会导致执行 `split-region` 或者 `remove-peer` 时报错进而引起其他元数据的不一致，最终引发 Region 不可用。
- 一旦执行了 `remove-fail-stores`，不可再重新启动被移除的节点并将其加入集群，否则会导致元数据的不一致，最终引发Region 不可用。

```
tikv-ctl --data-dir /path/to/tikv unsafe-recover remove-fail-stores -s 3 -r 1001,1002
```

```
success!
```

```
tikv-ctl --data-dir /path/to/tikv unsafe-recover remove-fail-stores -s 4,5 --all-regions
```

之后启动TiKV，这些Region 便可以使用剩下的健康副本继续提供服务了。此命令常用于多个TiKV store 损坏或被删除的情况。

注意：

- 一般来说，您需要为指定Region 的 peers 所在的每个 store 运行此命令。
- 该命令只支持本地模式。在运行成功后，会打印 `success!`。

4.3.1.3.17 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 --data-dir /path/to/tikv 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。

4.3.1.3.18 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。

4.3.1.3.19 打印加密元数据

`encryption-meta` 命令用于打印加密元数据。该子命令可以打印两种加密元数据：数据文件的加密信息，以及所有的数据加密密钥。

使用 `encryption-meta dump-file` 子命令打印数据文件的加密信息。你需要创建一个TiKV 配置文件用以指定 TiKV 的数据目录：

```
### conf.toml
[storage]
data-dir = "/path/to/tikv/data"
```

`--path` 选项可以指定数据文件的绝对或者相对路径。如果指定的文件是明文存储的，本命令有可能没有输出。如果不指定 `--path` 选项，本命令打印所有数据文件的加密信息。

```
tikv-ctl --config=./conf.toml encryption-meta dump-file --path=/path/to/tikv/data/db/CURRENT
/path/to/tikv/data/db/CURRENT: key_id: 9291156302549018620 iv: E3C2FDBF63FC03BFC28F265D7E78283F
    <-> method: Aes128Ctr
```

使用 `encryption-meta dump-key` 打印数据加密密钥。使用本命令的时候，除了在TiKV 配置文件中指定TiKV 的数据目录以外，还需要指定当前的主加密密钥。请参阅静态加密文档关于配置TiKV 主加密密钥的说明。使用

本命令时 security.encryption.previous-master-key 配置项不生效，即使配置文件中使用了该配置，本命令也不会触发更换主加密密钥。

```
### conf.toml
[storage]
data-dir = "/path/to/tikv/data"

[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

注意如果使用了AWS KMS 作为主加密密钥，使用本命令时 tikv-ctl 需要该 KMS 密钥的访问权限。KMS 访问权限可以通过环境变量、AWS 默认配置文件或IAM 的方式传递给 tikv-ctl。详情请参阅相关 AWS 文档。

--ids 选项可以指定以逗号分隔的数据加密密钥id 列表。如果不指定 --ids 选项，本命令打印所有的数据加密密钥，以及最新的数据加密密钥的id。

本命令会输出一个警告，提示本命令会泄漏敏感数据。根据提示输入“I consent” 即可。

```
tikv-ctl --config=./conf.toml encryption-meta dump-key
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
current key id: 9291156302549018620
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    <↪ 1592938357
```

```
tikv-ctl --config=./conf.toml encryption-meta dump-key --ids=9291156302549018620
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    <↪ 1592938357
```

> **注意：**
>
> 本命令会以明文方式打印数据加密密钥。在生产环境中，请勿将本命令的输出重定向到磁盘文件中。即使使用以后删除该文件也不能保证文件内容从磁盘中干净清除。

4.3.1.3.20 打印损坏的 SST 文件信息

TiKV 中损坏的 SST 文件会导致TiKV 进程崩溃。在TiDB v6.1.0 之前，损坏的 SST 文件会导致TiKV 进程立即崩溃。从 TiDB v6.1.0 起，TiKV 进程会在 SST 文件损坏 1 小时后崩溃。

为了方便清理掉这些 SST 文件，你可以先使用 bad-ssts 命令打印出损坏的 SST 文件信息。

> **注意：**
>
> 执行此命令前，请保证关闭当前运行的TiKV 实例。

```
tikv-ctl --data-dir </path/to/tikv> bad-ssts --pd <endpoint>
```

```
--------------------------------------------------------
corruption info:
data/tikv-21107/db/000014.sst: Corruption: Bad table magic number: expected 9863518390377041911,
    <⟶ found 759105309091689679 in data/tikv-21107/db/000014.sst
sst meta:
14:552997[1 .. 5520]['0101' seq:1, type:1 .. '7
    <⟶ A7480000000000000FF0F5F728000000000FF0002160000000000FAFA13AB33020BFFFA' seq:2032, type
    <⟶ :1] at level 0 for Column family "default"  (ID 0)
it isn't easy to handle local data, start key:0101
overlap region:
RegionInfo { region: id: 4 end_key: 7480000000000000FF0500000000000000F8 region_epoch { conf_ver:
    <⟶  1 version: 2 } peers { id: 5 store_id: 1 }, leader: Some(id: 5 store_id: 1) }
suggested operations:
tikv-ctl ldb --db=data/tikv-21107/db unsafe_remove_sst_file "data/tikv-21107/db/000014.sst"
tikv-ctl --db=data/tikv-21107/db tombstone -r 4 --pd <endpoint>

corruption analysis has completed------------------------


```

通过上面的输出，你可以看到损坏的 SST 文件和损坏原因等信息先被打印出，然后是相关的元信息。

- 在 `sst meta` 输出部分，14 表示 SST 文件号，552997 表示文件大小，紧随其后的是最小和最大的序列号 (seq) 等其它元信息。
- `overlap region` 部分为损坏 SST 文件所在Region 的信息。该信息是从 PD 组件获取的。
- `suggested operations` 部分为你清理损坏的 SST 文件提供建议操作。你可以参考这些建议的命令，清理文件，并重新启动该TiKV 实例。

4.3.2 PD Control 使用说明

PD Control 是 PD 的命令行工具，用于获取集群状态信息和调整集群。

4.3.2.1 安装方式

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

4.3.2.1.1 使用 TiUP

可直接通过 `tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> [-i]` 使用。

4.3.2.1.2 下载安装包

如需下载最新版本的 `pd-ctl`，直接下载 TiDB 安装包即可。`pd-ctl` 位于 TiDB 安装包的 `ctl-{version}-linux-{arch}.tar.gz` 包中。

安装包	操作系统	架构	SHA256 校验和
`https://download.pingcap.org/tidb-community-server-{version}-linux-amd64.tar.gz` (pd-ctl)	Linux	amd64	`https://download.pingcap.org/tidb-community-server-{version}-linux-amd64.sha256`
`https://download.pingcap.org/tidb-community-server-{version}-linux-arm64.tar.gz` (pd-ctl)	Linux	arm64	`https://download.pingcap.org/tidb-community-server-{version}-linux-arm64.sha256`

> **注意：**
>
> 下载链接中的 `{version}` 为 TiDB 的版本号。例如，amd64 架构的 `v7.0.0` 版本的下载链接为 `https://download.pingcap.org/tidb-community-server-v7.0.0-linux-amd64.tar.gz`。

4.3.2.1.3 源码编译

1. Go 1.20 或以上版本
2. 在 PD 项目根目录使用 `make` 或者 `make pd-ctl` 命令进行编译，生成 bin/pd-ctl

4.3.2.2 简单例子

单命令模式：

```
tiup ctl:v<CLUSTER_VERSION> pd store -u http://127.0.0.1:2379
```

交互模式：

```
tiup ctl:v<CLUSTER_VERSION> pd -i -u http://127.0.0.1:2379
```

使用环境变量：

```
export PD_ADDR=http://127.0.0.1:2379 &&
tiup ctl:v<CLUSTER_VERSION> pd
```

使用 TLS 加密：

```
tiup ctl:v<CLUSTER_VERSION> pd -u https://127.0.0.1:2379 --cacert="path/to/ca" --cert="path/to/
    <↩ cert" --key="path/to/key"
```

4.3.2.3 命令行参数(flags)

4.3.2.3.1 --cacert

- 指定PEM 格式的受信任 CA 证书的文件路径
- 默认值：“”

4.3.2.3.2 --cert

- 指定PEM 格式的 SSL 证书的文件路径
- 默认值：“”

4.3.2.3.3 --detach / -d

- 使用单命令行模式（不进入readline）
- 默认值：true

4.3.2.3.4 --help / -h

- 输出帮助信息
- 默认值：false

4.3.2.3.5 --interact/-i

- 使用交互模式（进入readline）
- 默认值：false

4.3.2.3.6 --key

- 指定PEM 格式的 SSL 证书密钥文件路径，即 --cert 所指定的证书的私钥
- 默认值：“”

4.3.2.3.7 --pd/-u

- 指定 PD 的地址
- 默认地址：http://127.0.0.1:2379
- 环境变量：PD_ADDR

4.3.2.3.8 --version/-V

- 打印版本信息并退出
- 默认值：false

4.3.2.4 命令(command)

4.3.2.4.1 cluster

用于显示集群基本信息。

示例：

```
cluster
```

```
{
  "id": 6493707687106161130,
  "max_peer_count": 3
}
```

4.3.2.4.2 config [show | set <option> <value> | placement-rules]

用于显示或调整配置信息。示例如下。

显示 scheduling 的相关config 信息：

```
config show
```

```
{
  "replication": {
    "enable-placement-rules": "true",
    "isolation-level": "",
    "location-labels": "",
    "max-replicas": 3,
    "strictly-match-label": "false"
  },
  "schedule": {
    "enable-cross-table-merge": "true",
    "high-space-ratio": 0.7,
    "hot-region-cache-hits-threshold": 3,
    "hot-region-schedule-limit": 4,
    "leader-schedule-limit": 4,
    "leader-schedule-policy": "count",
    "low-space-ratio": 0.8,
    "max-merge-region-keys": 200000,
    "max-merge-region-size": 20,
    "max-pending-peer-count": 64,
```

```
    "max-snapshot-count": 64,
    "max-store-down-time": "30m0s",
    "merge-schedule-limit": 8,
    "patrol-region-interval": "10ms",
    "region-schedule-limit": 2048,
    "region-score-formula-version": "v2",
    "replica-schedule-limit": 64,
    "scheduler-max-waiting-operator": 5,
    "split-merge-interval": "1h0m0s",
    "tolerant-size-ratio": 0
  }
}
```

显示所有的config 信息：

```
config show all
```

显示replication 的相关config 信息：

```
config show replication
```

```
{
  "max-replicas": 3,
  "isolation-level": "",
  "location-labels": "",
  "strictly-match-label": "false",
  "enable-placement-rules": "true"
}
```

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
```

```
"5.2.2"
```

- 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 64：

  ```
  config set max-snapshot-count 64
  ```

- 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 MiB）。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 MiB：

```
config set max-merge-region-size 16
```

- `max-merge-region-keys` 控制Region Merge 的 keyCount 上限。当Region KeyCount 大于指定值时 PD 不会将其与相邻的Region 合并。

  设置 Region Merge 的 keyCount 上限为 50000：

```
config set max-merge-region-keys 50000
```

- `split-merge-interval` 控制对同一个Region 做 `split` 和 `merge` 操作的间隔，即对于新 `split` 的 Region 一段时间内不会被 `merge`。

  设置 `split` 和 `merge` 的间隔为 1 天：

```
config set split-merge-interval 24h
```

- `enable-one-way-merge` 用于控制是否只允许和相邻的后一个Region 进行合并。当设置为 `false` 时，PD 允许与相邻的前后Region 进行合并。

  设置只允许和相邻的后一个Region 合并：

```
config set enable-one-way-merge true
```

- `enable-cross-table-merge` 用于开启跨表 Region 的合并。当设置为 `false` 时，PD 不会合并不同表的 Region。该选项只在键类型为“table”时生效。

  设置允许跨表合并：

```
config set enable-cross-table-merge true
```

- `key-type` 用于指定集群的键编码类型。支持的类型有 `["table", "raw", "txn"]`，默认值为 “table”。
  - 如果集群中不存在TiDB 实例，`key-type` 的值为“raw”或“txn”。此时，无论 `enable-cross-table-merge` 设置为何，PD 均可以跨表合并Region。
  - 如果集群中存在TiDB 实例，`key-type` 的值应当为“table”。此时，`enable-cross-table-merge` 的设置决定了 PD 是否能跨表合并Region。如果 `key-type` 的值为“raw”，placement rules 不生效。

  启用跨表合并：

```
config set key-type raw
```

- `region-score-formula-version` 用于设置Region 算分公式的版本，支持的值有 `["v1", "v2"]`。v2 版本公式有助于减少上下线等场景下冗余的balance Region 调度。

  开启 v2 版本Region 算分公式：

```
config set region-score-formula-version v2
```

- 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 10 毫秒：

```
config set patrol-region-interval 10ms
```

- 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
```

- max-store-preparing-time 控制 store 上线阶段的最长等待时间。在 store 的上线阶段，PD 可以查询该 store 的上线进度。当超过该配置项指定的时间后，PD 会认为该 store 已完成上线，无法再次查询这个 store 的上线进度，但是不影响Region 向这个新上线 store 的迁移。通常用户无需修改该配置项。

  设置 store 上线阶段最多等待 4 小时：

```
config set max-store-preparing-time 4h
```

- 通过调整 leader-schedule-limit 可以控制同时进行 leader 调度的任务个数。这个值主要影响 leader balance 的速度，值越大调度得越快，设置为 0 则关闭调度。Leader 调度的开销较小，需要的时候可以适当调大。

  最多同时进行 4 个 leader 调度：

```
config set leader-schedule-limit 4
```

- 通过调整 region-schedule-limit 可以控制同时进行Region 调度的任务个数。这个值可以避免创建过多的Region balance operator。默认值为 2048，对所有大小的集群都足够。设置为 0 则关闭调度。Region 调度的速度通常受到 store-limit 的限制，但除非你熟悉该设置，否则不推荐自定义该参数。

  最多同时进行 2 个 Region 调度：

```
config set region-schedule-limit 2
```

- 通过调整 replica-schedule-limit 可以控制同时进行replica 调度的任务个数。这个值主要影响节点挂掉或者下线的时候进行调度的速度，值越大调度得越快，设置为 0 则关闭调度。Replica 调度的开销较大，所以这个值不宜调得太大。注意：该参数通常保持为默认值。如需调整，需要根据实际情况反复尝试设置该值大小。

  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。注意：该参数通常保持为默认值。如需调整，需要根据实际情况反复尝试设置该值大小。

  最多同时进行 16 个 merge 调度：

```
config set merge-schedule-limit 16
```

- `hot-region-schedule-limit` 控制同时进行的 Hot Region 调度的任务，设置为 0 则关闭调度。这个值不宜调得过大，否则可能对系统性能造成影响。注意：该参数通常保持为默认值。如需调整，需要根据实际情况反复尝试设置该值大小。

  最多同时进行 4 个 Hot Region 调度：

```
config set hot-region-schedule-limit 4
```

- `hot-region-cache-hits-threshold` 用于设置识别热点Region 所需的分钟数，只有Region 处于热点状态持续时间超过该分钟数后，才能参与热点调度。

- `tolerant-size-ratio` 控制balance 缓冲区大小。当两个 store 的 leader 或 Region 的得分差距小于指定倍数的 Region size 时，PD 会认为此时balance 达到均衡状态。

  设置缓冲区为约 20 倍平均RegionSize：

```
config set tolerant-size-ratio 20
```

- `low-space-ratio` 用于设置 store 空间不足的阈值。当节点的空间占用比例超过指定值时，PD 会尽可能避免往对应节点迁移数据，同时主要针对剩余空间大小进行调度，避免对应节点磁盘空间被耗尽。

  设置空间不足阈值为 0.9：

```
config set low-space-ratio 0.9
```

- `high-space-ratio` 用于设置 store 空间充裕的阈值，此配置仅的在 `region-score-formula-version = v1` 时生效。当节点的空间占用比例小于指定值时，PD 调度时会忽略剩余空间这个指标，主要针对实际数据量进行均衡。

  设置空间充裕阈值为 0.5：

```
config set high-space-ratio 0.5
```

- `cluster-version` 集群的版本，用于控制某些 Feature 是否开启，处理兼容性问题。通常是集群正常运行的所有TiKV 节点中的最低版本，需要回滚到更低的版本时才进行手动设置。

  设置cluster version 为 1.0.8：

```
config set cluster-version 1.0.8
```

- `leader-schedule-policy` 用于选择 Leader 的调度策略，可以选择按照 `size` 或者 `count` 来进行调度。

- `scheduler-max-waiting-operator` 用于控制每个调度器同时存在的 operator 的个数。

- `enable-remove-down-replica` 用于开启自动删除 DownReplica 的特性。当设置为 false 时，PD 不会自动清理宕机状态的副本。

- enable-replace-offline-replica 用于开启迁移 OfflineReplica 的特性。当设置为 false 时，PD 不会迁移下线状态的副本。
- enable-make-up-replica 用于开启补充副本的特性。当设置为 false 时，PD 不会为副本数不足的Region 补充副本。
- enable-remove-extra-replica 用于开启删除多余副本的特性。当设置为 false 时，PD 不会为副本数过多的 Region 删除多余副本。
- enable-location-replacement 用于开启隔离级别检查。当设置为false 时，PD 不会通过调度来提升Region 副本的隔离级别。
- enable-debug-metrics 用于开启debug 的metrics。当设置为 true 时，PD 会开启一些metrics，比如 balance <-> -tolerant-size 等。
- enable-placement-rules 用于开启placement rules，在 v5.0 及以上的版本默认开启。
- store-limit-mode 用于控制 store 限速机制的模式。主要有两种模式：auto 和 manual。auto 模式下会根据 load 自动进行平衡调整（弃用）。
- store-limit-version 用于设置 store limit 限制模式，目前提供两种方式：v1 和 v2。默认值为 v1。在 v1 模式下，你可以手动修改 store limit 以限制单个TiKV 调度速度。v2 模式为实验特性，在 v2 模式下，你无需关注 store limit 值，PD 将根据TiKV Snapshot 执行情况动态调整TiKV 调度速度。详情请参考Store Limit v2 原理。

```
config set store-limit-version v2        // 使用 Store Limit v2
```

- PD 会对流量信息的末尾数字进行四舍五入处理，减少Region 流量信息变化引起的统计信息更新。该配置项用于指定对Region 流量信息的末尾进行四舍五入的位数。例如流量 100512 会归约到 101000。默认值为 3。该配置替换了 trace-region-flow。

  示例：将 flow-round-by-digit 的值设为 4：

```
config set flow-round-by-digit 4
```

4.3.2.4.3 config placement-rules [disable | enable | load | save | show | rule-group]

关于 config placement-rules 的具体用法，参考Placement Rules 使用文档。

4.3.2.4.4 health

用于显示集群健康信息。示例如下。

显示健康信息：

```
health
```

```
[
  {
    "name": "pd",
    "member_id": 13195394291058371180,
```

```
    "client_urls":
      [ "http://127.0.0.1:2379
      "
      ......
    ],
    "health": true
  }
  ......
]
```

4.3.2.4.5 hot [read | write | store| history <start_time> <end_time> [<key> <value>]]

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
```

显示历史读写热点信息:

```
hot history startTime endTime [ <name> <value> ]
```

例如查询时间 1629294000000 到 1631980800000 （毫秒）之间的历史热点Region 信息:

```
hot history 1629294000000 1631980800000
```

```
{
  "history_hot_region": [
    {
      "update_time": 1630864801948,
      "region_id": 103,
      "peer_id": 1369002,
      "store_id": 3,
      "is_leader": true,
      "is_learner": false,
      "hot_region_type": "read",
      "hot_degree": 152,
      "flow_bytes": 0,
      "key_rate": 0,
      "query_rate": 305,
      "start_key": "7480000000000000FF5300000000000000F8",
```

```
      "end_key": "7480000000000000FF5600000000000000F8"
    },
    ...
  ]
}
```

对于参数的值为数组的请用 `x, y, ...` 的形式进行参数值的设置，所有支持的参数如下所示:

```
hot history 1629294000000 1631980800000 hot_region_type read region_id 1,2,3 store_id 1,2,3
    <-> peer_id 1,2,3 is_leader true is_learner true
```

```
{
  "history_hot_region": [
    {
      "update_time": 1630864801948,
      "region_id": 103,
      "peer_id": 1369002,
      "store_id": 3,
      "is_leader": true,
      "is_learner": false,
      "hot_region_type": "read",
      "hot_degree": 152,
      "flow_bytes": 0,
      "key_rate": 0,
      "query_rate": 305,
      "start_key": "7480000000000000FF5300000000000000F8",
      "end_key": "7480000000000000FF5600000000000000F8"
    },
    ...
  ]
}
```

4.3.2.4.6 label [store <name> <value>]

用于显示集群标签信息。示例如下。

显示所有label：

```
label
```

显示所有包含label 为 “zone”：“cn” 的 store：

```
label store zone cn
```

4.3.2.4.7 member [delete | leader_priority | leader [show | resign | transfer <member_name>]]

用于显示 PD 成员信息，删除指定成员，设置成员的leader 优先级。示例如下。

显示所有成员的信息：

```
member
```

```
{
  "header": {.....},
  "members": [.....],
  "leader": {.....},
  "etcd_leader": {.....},
}
```

下线“pd2”：

```
member delete name pd2
```

```
Success!
```

使用id 下线节点：

```
member delete id 1319539429105371180
```

```
Success!
```

显示leader 的信息：

```
member leader show
```

```
{
  "name": "pd",
  "member_id": 13155432540099656863,
  "peer_urls": [.....],
  "client_urls": [.....]
}
```

将 leader 从当前成员移走：

```
member leader resign
```

```
......
```

将 leader 迁移至指定成员：

```
member leader transfer pd3
```

```
......
```

4.3.2.4.8 operator [check | show | add | remove]

用于显示和控制调度操作。

示例：

```
>> operator show                                    // 显示所有的 operators
>> operator show admin                              // 显示所有的 admin operators
>> operator show leader                             // 显示所有的 leader operators
>> operator show region                             // 显示所有的 Region operators
>> operator add add-peer 1 2                        // 在 store 2 上新增 Region 1 的一个副本
>> operator add add-learner 1 2                     // 在 store 2 上新增 Region 1 的一个
    <-> learner 副本
>> operator add remove-peer 1 2                     // 移除 store 2 上的 Region 1 的一个副本
>> operator add transfer-leader 1 2                 // 把 Region 1 的 leader 调度到 store 2
>> operator add transfer-region 1 2 3 4             // 把 Region 1 调度到 store 2,3,4
>> operator add transfer-peer 1 2 3                 // 把 Region 1 在 store 2 上的副本调度到
    <-> store 3
>> operator add merge-region 1 2                    // 将 Region 1 与 Region 2 合并
>> operator add split-region 1 --policy=approximate // 将 Region 1 对半拆分成两个 Region，
    <-> 基于粗略估计值
>> operator add split-region 1 --policy=scan        // 将 Region 1 对半拆分成两个 Region，
    <-> 基于精确扫描值
>> operator remove 1                                // 把 Region 1 的调度操作删掉
>> operator check 1                                 // 查看 Region 1 相关 operator 的状态
```

其中，Region 的分裂都是尽可能地从靠近中间的位置开始。对这个位置的选择支持两种策略，即 scan 和 approximate。它们之间的区别是，前者通过扫描这个Region 的方式来确定中间的 key，而后者是通过查看 SST 文件中记录的统计信息，来得到近似的位置。一般来说，前者更加精确，而后者消耗更少的I/O，可以更快地完成。

4.3.2.4.9 ping

用于显示ping PD 所需要花费的时间

示例：

```
ping
```

```
time: 43.12698ms
```

4.3.2.4.10 region <region_id> [--jq="<query string>"]

用于显示Region 信息。使用jq 格式化输出请参考jq 格式化json 输出示例。示例如下。

显示所有Region 信息：

```
region
```

```

```

```
{
  "count": 1,
  "regions": [.....]
}
```

显示 Region id 为 2 的信息：

```
region 2
```

```
{
  "id": 2,
  "start_key": "7480000000000000FF1D00000000000000F8",
  "end_key": "7480000000000000FF1F00000000000000F8",
  "epoch": {
    "conf_ver": 1,
    "version": 15
  },
  "peers": [
    {
      "id": 40,
      "store_id": 3
    }
  ],
  "leader": {
    "id": 40,
    "store_id": 3
  },
  "written_bytes": 0,
  "read_bytes": 0,
  "written_keys": 0,
  "read_keys": 0,
  "approximate_size": 1,
  "approximate_keys": 0
}
```

4.3.2.4.11 region key [--format=raw|encode|hex] <key>

用于查询某个 key 位于哪一个Region 上，支持raw、 encoding 和 hex 格式。使用 encoding 格式时，key 需要使用单引号。

Hex 格式（默认）示例：

```
region key 7480000000000000FF1300000000000000F8
{
  "region": {
    "id": 2,
```

```
    ......
  }
}
```

Raw 格式示例：

```
region key --format=raw abc
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

Encoding 格式示例：

```
region key --format=encode 't\200\000\000\000\000\000\000\377\035_r\200\000\000\000\000\377\017U
    <-> \320\000\000\000\000\000\372'
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

4.3.2.4.12 region scan

用于获取所有Region。

示例：

```
region scan
```

```
{
  "count": 20,
  "regions": [.....],
}
```

4.3.2.4.13 region sibling <region_id>

用于查询某个Region 相邻的Region。

示例：

```
region sibling 2
```

```
{
  "count": 2,
  "regions": [.....],
}
```

4.3.2.4.14 region keys [--format=raw|encode|hex] <start_key> <end_key> <limit>

用于查询某个 key 范围内的所有Region。支持不带 endKey 的范围。`limit` 的默认值是 16，设为 -1 则表示无数量限制。示例如下：

显示从a开始的所有Region 信息，数量上限为 16：

```
region keys --format=raw a
```

```
{
  "count": 16,
  "regions": [..... ],
}
```

显示 [a, z) 范围内的所有Region 信息，数量上限为 16：

```
region keys --format=raw a z
```

```
{
  "count": 16,
  "regions": [..... ],
}
```

显示 [a, z) 范围内的所有Region 信息，无数量上限：

```
region keys --format=raw a z -1
```

```
{
  "count": ...,
  "regions": [..... ],
}
```

显示从a开始的所有Region 信息，数量上限为 20：

```
region keys --format=raw a "" 20
```

```
{
  "count": 20,
  "regions": [..... ],
}
```

4.3.2.4.15 region store <store_id>

用于查询某个 store 上面所有的Region。

示例：

```
region store 2
```

```
{
  "count": 10,
  "regions": [.....],
}
```

4.3.2.4.16 region topread [limit]

用于查询读流量最大的Region。 limit 的默认值是 16。

示例：

```
region topread
```

```
{
  "count": 16,
  "regions": [.....],
}
```

4.3.2.4.17 region topwrite [limit]

用于查询写流量最大的Region。 limit 的默认值是 16。

示例：

```
region topwrite
```

```
{
  "count": 16,
  "regions": [.....],
}
```

4.3.2.4.18 region topconfver [limit]

用于查询conf version 最大的Region。 limit 的默认值是 16。

示例：

```
region topconfver
```

```
{
  "count": 16,
  "regions": [.....],
}
```

4.3.2.4.19 region topversion [limit]

用于查询 version 最大的Region。 limit 的默认值是 16。

示例：

```
region topversion
```

```
{
  "count": 16,
  "regions": [..... ],
}
```

4.3.2.4.20 region topsize [limit]

用于查询approximate size 最大的Region。 limit 的默认值是 16。

示例：

```
region topsize
```

```
{
  "count": 16,
  "regions": [..... ],
}
```

4.3.2.4.21 region check [miss-peer | extra-peer | down-peer | pending-peer | offline-peer | empty-region | hist-size | hist-keys] [--jq="<query string>"]

用于查询处于异常状态的Region，使用jq 格式化输出请参考jq 格式化JSON 输出示例。

各类型的意义如下：

- miss-peer：缺副本的Region
- extra-peer：多副本的 Region
- down-peer：有副本状态为 Down 的 Region
- pending-peer：有副本状态为 Pending 的 Region

示例：

```
region check miss-peer
```

```
{
  "count": 2,
  "regions": [.....],
}
```

4.3.2.4.22 scheduler [show | add | remove | pause | resume | config | describe]

用于显示和控制调度策略。

示例：

```
>> scheduler show                                          // 显示所有已经创建的 schedulers
>> scheduler add grant-leader-scheduler 1                  // 把 store 1 上的所有 Region 的 leader
    <-> 调度到 store 1
>> scheduler add evict-leader-scheduler 1                  // 把 store 1 上的所有 Region 的 leader
    <-> 从 store 1 调度出去
>> scheduler config evict-leader-scheduler                 // v4.0.0 起，展示该调度器具体在哪些
    <-> store 上
>> scheduler add shuffle-leader-scheduler                  // 随机交换不同 store 上的 leader
>> scheduler add shuffle-region-scheduler                  // 随机调度不同 store 上的 Region
>> scheduler add evict-slow-store-scheduler                // 当有且仅有一个 slow store 时将该
    <-> store 上的所有 Region 的 leader 驱逐出去
>> scheduler remove grant-leader-scheduler-1               // 把对应的调度器删掉，`-1` 对应 store
    <-> ID
>> scheduler pause balance-region-scheduler 10             // 暂停运行 balance-region 调度器 10 秒
>> scheduler pause all 10                                  // 暂停运行所有的调度器 10 秒
>> scheduler resume balance-region-scheduler               // 继续运行 balance-region 调度器
>> scheduler resume all                                    // 继续运行所有的调度器
>> scheduler config balance-hot-region-scheduler           // 显示 balance-hot-region 调度器的配置
>> scheduler describe balance-region-scheduler             // 显示 balance-region
    <-> 的运行状态和相应的诊断信息
```

4.3.2.4.23 scheduler describe balance-region-scheduler

用于查看 balance-region-scheduler 的运行状态和相应的诊断信息。

从 TiDB v6.3.0 起，PD 为 balance-region-scheduler 和 balance-leader-scheduler 提供了运行状态和简要诊断信息的功能，其余 scheduler 和checker 暂未支持。你可以通过 pd-ctl 修改enable-diagnostic 配置项开启该功能。

调度器运行状态有以下几种类型：

- disabled：表示当前调度器不可用或被移除。
- paused：表示当前调度器暂停工作。
- scheduling：表示当前调度器正在生成调度。

- pending：表示当前调度器无法产生调度。pending 状态的调度器，会返回一个概览信息，来帮助用户诊断。概览信息包含了 store 的一些状态信息，解释了它们为什么不能被选中进行调度。
- normal：表示当前调度器无需进行调度。

4.3.2.4.24 scheduler config balance-leader-scheduler

用于查看和控制 balance-leader-scheduler 策略。

从TiDB v6.0.0 起，PD 为 balance-leader-scheduler 引入了 Batch 参数，用于控制 balance-leader 执行任务的速度。你可以通过pd-ctl 修改 balance-leader batch 配置项设置该功能。

在 v6.0.0 前，PD 不带有该配置（即 balance-leader batch=1）。在 v6.0.0 或更高版本中，balance-leader batch 的默认值为 4。如果你想为该配置项设置大于 4 的值，你需要同时调大scheduler-max-waiting-operator（默认值 5）。同时调大两个配置项后，你才能体验预期的加速效果。

```
scheduler config balance-leader-scheduler set batch 3  // 将 balance-leader
    <-> 调度器可以批量执行的算子大小设置为 3
```

4.3.2.4.25 scheduler config balance-hot-region-scheduler

用于查看和控制 balance-hot-region-scheduler 策略。

示例：

```
scheduler config balance-hot-region-scheduler // 显示 balance-hot-region 调度器的所有配置
{
  "min-hot-byte-rate": 100,
  "min-hot-key-rate": 10,
  "min-hot-query-rate": 10,
  "max-zombie-rounds": 3,
  "max-peer-number": 1000,
  "byte-rate-rank-step-ratio": 0.05,
  "key-rate-rank-step-ratio": 0.05,
  "query-rate-rank-step-ratio": 0.05,
  "count-rank-step-ratio": 0.01,
  "great-dec-ratio": 0.95,
  "minor-dec-ratio": 0.99,
  "src-tolerance-ratio": 1.05,
  "dst-tolerance-ratio": 1.05,
  "read-priorities": [
    "query",
    "byte"
  ],
  "write-leader-priorities":
    [ "key",
    "byte"
  ],
  "write-peer-priorities": [
```

```
    "byte",
    "key"
  ],
  "strict-picking-store": "true",
  "enable-for-tiflash": "true",
  "rank-formula-version": "v2"
}
```

- `min-hot-byte-rate` 指计数的最小字节数，通常为 100。

```
scheduler config balance-hot-region-scheduler set min-hot-byte-rate 100
```

- `min-hot-key-rate` 指计数的最小 key 数，通常为 10。

```
scheduler config balance-hot-region-scheduler set min-hot-key-rate 10
```

- `min-hot-query-rate` 指计数的最小query 数，通常为 10。

```
scheduler config balance-hot-region-scheduler set min-hot-query-rate 10
```

- `max-zombie-rounds` 指一个 operator 可被纳入pending influence 所允许的最大心跳次数。如果将它设置为更大的值，更多的 operator 可能会被纳入pending influence。通常用户不需要修改这个值。pending influence 指的是在调度中产生的、但仍生效的影响。

```
scheduler config balance-hot-region-scheduler set max-zombie-rounds 3
```

- `max-peer-number` 指最多要被解决的peer 数量。这个配置可避免调度器处理速度过慢。

```
scheduler config balance-hot-region-scheduler set max-peer-number 1000
```

- `byte-rate-rank-step-ratio`、`key-rate-rank-step-ratio`、`query-rate-rank-step-ratio` 和 `count-rank-step-ratio` 分别控制byte、 key、 query 和 count 的 step ranks。rank-step-ratio 决定了计算rank 时的 step 值。`great-dec-ratio` 和 `minor-dec-ratio` 控制 dec 的rank。通常用户不需要修改这些配置项。

```
scheduler config balance-hot-region-scheduler set byte-rate-rank-step-ratio 0.05
```

- `src-tolerance-ratio` 和 `dst-tolerance-ratio` 是期望调度器的配置项。`tolerance-ratio` 的值越小，调度就越容易。当出现冗余调度时，你可以适当调大这个值。

```
scheduler config balance-hot-region-scheduler set src-tolerance-ratio 1.1
```

- `read-priorities`、`write-leader-priorities` 和 `write-peer-priorities` 用于控制调度器优先从哪些维度进行热点均衡，支持配置两个维度。
  - `read-priorities` 和 `write-leader-priorities` 用于控制调度器在处理read 和 write-leader 类型的热点时优先均衡的维度，可选的维度有 `query`、`byte` 和 `key`。
  - `write-peer-priorities` 用于控制调度器在处理 write-peer 类型的热点时优先均衡的维度，支持配置 `byte` 和 `key` 维度。

> **注意：**
> 若集群的所有组件未全部升级到 v5.2 及以上版本，query 维度的配置不生效，部分组件升级完成后调度器仍默认优先从 byte 和 key 维度进行热点均衡，集群的所有组件全部升级完成后，也会继续保持这样的兼容配置，可通过 pd-ctl 查看实时配置。通常用户不需要修改这些配置项。

```
scheduler config balance-hot-region-scheduler set read-priorities query,byte
```

- strict-picking-store 是控制热点调度搜索空间的开关，通常为打开。该配置项仅影响 rank-formula-version 为 v1 时的行为。当打开时，热点调度的目标是保证所配置的两个维度的热点均衡。当关闭后，热点调度只保证处于第一优先级的维度的热点均衡表现更好，但可能会导致其他维度的热点不再那么均衡。通常用户不需要修改这个配置项。

```
scheduler config balance-hot-region-scheduler set strict-picking-store true
```

- rank-formula-version 适用于热点调度，其用来确定调度策略的算法版本，支持的值有 ["v1", "v2"]。目前该配置的默认值为 v2。
  - v1 版本为 v6.3.0 之前的策略，主要关注调度是否降低了不同 Store 之间的负载差值，以及是否在另一维度引入副作用。
  - v2 版本是 v6.3.0 引入的实验特性算法，在 v6.4.0 正式发布，主要关注 Store 之间均衡度的提升率，同时降低了对副作用的关注度。对比 strict-picking-store 为 true 的 v1 算法，v2 版本更注重优先均衡第一维度。对比 strict-picking-store 为 false 的 v1 算法，v2 版本兼顾了第二维度的均衡。
  - strict-picking-store 为true 的v1 版本算法较为保守，只有当存在两个维度的负载都偏高的Store 时才能产生调度。在特定场景下有可能因为维度冲突导致无法继续均衡，需要将 strict-picking-store 改为 false 才能在第一维度取得更好的均衡效果。v2 版本算法则可以在两个维度都取得更好的均衡效果，并减少无效调度。

```
scheduler config balance-hot-region-scheduler set rank-formula-version v2
```

- enable-for-tiflash 是控制热点调度是否对TiFlash 生效的开关。通常为打开，关闭后将不会产生TiFlash 实例之间的热点调度。

```
scheduler config balance-hot-region-scheduler set enable-for-tiflash true
```

4.3.2.4.26 service-gc-safepoint

用于查询当前的GC safepoint 与 service GC safepoint，输出结果示例如下：

```
{
  "service_gc_safe_points": [
    {
      "service_id": "gc_worker",
      "expired_at": 9223372036854775807,
```

```
      "safe_point": 439923410637160448
    }
  ],
  "gc_safe_point": 0
}
```

4.3.2.4.27 store [delete | cancel-delete | label | weight | remove-tombstone | limit] <store_id> [--jq="<query string>"]

使用jq格式化输出请参考jq格式化json输出示例。

查询store

显示所有store信息：

```
store
```

```
{
  "count": 3,
  "stores": [...]
}
```

获取id为1的store：

```
store 1
```

```
......
```

下线store

下线id为1的store：

```
store delete 1
```

执行 `store cancel-delete` 命令，你可以撤销已使用 `store delete` 下线并处于 `Offline` 状态的store。撤销后，该store会从 `Offline` 状态变为 `Up` 状态。注意，`store cancel-delete` 命令无法使 `Tombstone` 状态的store变回 `Up` 状态。

撤销通过 `store delete` 下线id为1的store：

```
store cancel-delete 1
```

删除所有Tombstone状态的store：

```
store remove-tombstone
```

> 注意：
> 若下线过程中切换了PD leader，需要使用 `store limit` 命令修改store调度限速。

管理 store label

`store label` 命令用于管理 store label。

- 为 id 为 1 的 store 设置键为 `"zone"`、值为 `"cn"` 的 label：

```
store label 1 zone=cn
```

- 更新 id 为 1 的 store 的 label：

```
store label 1 zone=us
```

- 通过 `--rewrite` 选项重写 id 为 1 的 store 的所有 label，之前的 label 会被覆盖：

```
store label 1 region=us-est-1 disk=ssd --rewrite
```

- 删除 id 为 1 的 store 的键为 `"disk"` 的 label：

```
store label 1 disk --delete
```

> 注意：
>
> - store 的 label 更新方法使用的是合并策略。如果修改了 TiKV 配置文件中的 store label，进程重启之后，PD 会将自身存储的 store label 与其进行合并更新，并持久化合并后的结果。
> - 如果希望使用 TiUP 统一管理 store label，你可以在集群重启前，使用 PD Control 的 `store label <id> --force` 命令将 PD 存储的 store label 清空。

设置 store weight

将 id 为 1 的 store 的 leader weight 设为 5，Region weight 设为 10：

```
store weight 1 5 10
```

设置 store 调度限速

通过 `store-limit`，你可以设置 store 的调度速度。关于 `store limit` 的原理和使用方法，请参考store limit。

```
>> store limit                          // 显示所有 store 添加和删除 peer 的速度上限
>> store limit add-peer                 // 显示所有 store 添加 peer 的速度上限
>> store limit remove-peer              // 显示所有 store 删除 peer 的速度上限
>> store limit all 5                    // 设置所有 store 添加和删除 peer 的速度上限为每分钟 5 个
>> store limit 1 5                      // 设置 store 1 添加和删除 peer 的速度上限为每分钟 5 个
>> store limit all 5 add-peer           // 设置所有 store 添加 peer 的速度上限为每分钟 5 个
>> store limit 1 5 add-peer             // 设置 store 1 添加 peer 的速度上限为每分钟 5 个
>> store limit 1 5 remove-peer          // 设置 store 1 删除 peer 的速度上限为每分钟 5 个
>> store limit all 5 remove-peer        // 设置所有 store 删除 peer 的速度上限为每分钟 5 个
```

> **注意：**
>
> 使用 pd-ctl 可以查看TiKV 节点的状态信息，即 Up，Disconnect，Offline，Down，或 Tombstone。如需查看各个状态之间的关系，请参考TiKV Store 状态之间的关系。

4.3.2.4.28 log [fatal | error | warn | info | debug]

用于设置 PD leader 的日志级别。

```
log warn
```

4.3.2.4.29 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
```

4.3.2.4.30 unsafe remove-failed-stores [store-ids | show]

> **警告：**
>
> - 此功能为有损恢复，无法保证数据和数据索引完整性。
> - 建议在TiDB 团队支持下进行相关操作，操作不当可能导致集群难以恢复。

用于在多数副本永久损坏造成数据不可用时进行有损恢复。示例如下。详见 Online Unsafe Recovery。

执行 Online Unsafe Recovery，移除永久损坏的节点(Store):

```
unsafe remove-failed-stores 101,102,103
```

```
Success!
```

显示正在运行的 Online Unsafe Recovery 的当前状态或历史状态。

```
unsafe remove-failed-stores show
```

```
[
  "Collecting cluster info from all alive stores, 10/12.",
  "Stores that have reports to PD: 1, 2, 3, ...",
  "Stores that have not reported to PD: 11, 12",
]
```

4.3.2.5 jq 格式化JSON 输出示例

4.3.2.5.1 简化 store 的输出

```
store --jq=".stores[].store | { id, address, state_name}"
```

```
{"id":1,"address":"127.0.0.1:20161","state_name":"Up"}
{"id":30,"address":"127.0.0.1:20162","state_name":"Up"}
...
```

4.3.2.5.2 查询节点剩余空间

```
store --jq=".stores[] | {id: .store.id, available: .status.available}"
```

```
{"id":1,"available":"10 GiB"}
{"id":30,"available":"10 GiB"}
...
```

4.3.2.5.3 查询状态不为 Up 的所有节点

```
store --jq='.stores[].store | select(.state_name!="Up") | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Offline"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Offline"}
...
```

4.3.2.5.4 查询所有的TiFlash 节点

```
store --jq='.stores[].store | select(.labels | length>0 and contains([{"key":"engine","value":"
    ⟷ tiflash"}])) | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Up"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Up"}
...
```

4.3.2.5.5 查询Region副本的分布情况

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id]}"
```

```
{"id":2,"peer_stores":[1,30,31]}
{"id":4,"peer_stores":[1,31,34]}
...
```

4.3.2.5.6 根据副本数过滤Region

例如副本数不为 3 的所有Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length != 3)}"
```

```
{"id":12,"peer_stores":[30,32]}
{"id":2,"peer_stores":[1,30,31,32]}
```

4.3.2.5.7 根据副本 store ID 过滤Region

例如在 store30 上有副本的所有Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==30))}"
```

```
{"id":6,"peer_stores":[1,30,31]}
{"id":22,"peer_stores":[1,30,32]}
...
```

还可以像这样找出在 store30 或 store31 上有副本的所有Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==(30,31)))}"
```

```
{"id":16,"peer_stores":[1,30,34]}
{"id":28,"peer_stores":[1,30,32]}
{"id":12,"peer_stores":[30,32]}
...
```

4.3.2.5.8 恢复数据时寻找相关Region

例如当 [store1, store30, store31] 宕机时不可用时，我们可以通过查找所有 Down 副本数量大于正常副本数量的所有 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length as $total |
    <⟶ map(if .==(1,30,31) then . else empty end) | length>=$total-length) }"
```

```
{"id":2,"peer_stores":[1,30,31,32]}
{"id":12,"peer_stores":[30,32]}
{"id":14,"peer_stores":[1,30,32]}
...
```

或者在 [store1, store30, store31] 无法启动时，找出 store1 上可以安全手动移除数据的Region。我们可以这样过滤出所有在 store1 上有副本并且没有其他 DownPeer 的 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length>1 and any
    <-> (.==1) and all(.!=(30,31)))}"
```

```
{"id":24,"peer_stores":[1,32,33]}
```

在 [store30, store31] 宕机时，找出能安全地通过创建 `remove-peer` Operator 进行处理的所有Region，即有且仅有一个 DownPeer 的 Region：

```
region --jq=".regions[] | {id: .id, remove_peer: [.peers[].store_id] | select(length>1) | map(if
    <-> .==(30,31) then . else empty end) | select(length==1)}"
```

```
{"id":12,"remove_peer":[30]}
{"id":4,"remove_peer":[31]}
{"id":22,"remove_peer":[30]}
...
```

4.3.3 TiDB Control 使用说明

TiDB Control 是 TiDB 的命令行工具，用于获取TiDB 状态信息，多用于调试。本文介绍了TiDB Control 的主要功能和各个功能的使用方法。

4.3.3.1 获取TiDB Control

本节提供了两种方式获取TiDB Control 工具。

> 注意：
> 建议使用的 Control 工具版本与集群版本保持一致。

4.3.3.1.1 通过TiUP 安装

在安装TiUP 之后，可以使用 `tiup ctl:v<CLUSTER_VERSION> tidb` 命令来获取TiDB Control 的二进制程序以及运行 TiDB Control。

4.3.3.1.2 从源代码编译安装

编译环境要求：Go 1.20 或以上版本

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。

4.3.3.2 使用介绍

`tidb-ctl` 的使用由命令（包括子命令）、选项和参数组成。命令即不带 - 或者 -- 的字符，选项即带有- 或者 -- 的字符，参数即命令或选项字符后紧跟的传递给命令和选项的字符。

如：`tidb-ctl schema in mysql -n db`

- schema: 命令
- in: schema 的子命令
- mysql: in 的参数
- -n: 选项
- db: -n 的参数

目前，TiDB Control 包含以下子命令。

- `tidb-ctl base64decode` 用于 BASE64 解码
- `tidb-ctl decoder` 用于KEY 解码
- `tidb-ctl etcd` 用于操作 etcd
- `tidb-ctl log` 用于格式化日志文件，将单行的堆栈信息展开
- `tidb-ctl mvcc` 用于获取 MVCC 信息
- `tidb-ctl region` 用于获取Region 信息
- `tidb-ctl schema` 用于获取 Schema 信息
- `tidb-ctl table` 用于获取 Table 信息

4.3.3.2.1 获取帮助

`tidb-ctl -h/--help` 用于获取帮助信息。tidb-ctl 由多层命令组成，tidb-ctl 及其所有子命令都可以通过`-h/--help` 来获取使用帮助。

以获取 Schema 信息为例：

通过 `tidb-ctl schema -h` 可以获取这个子命令的使用帮助。schema 有两个子命令——in 和 tid。in 用来通过数据库名获取数据库中所有表的表结构，tid 用来通过全数据库唯一的table_id 获取表的表结构。

4.3.3.2.2 全局参数

`tidb-ctl` 有 4 个与连接相关的全局参数，分别为：

- `--host` TiDB 服务地址
- `--port` TiDB status 端口
- `--pdhost` PD 服务地址

- --pdport PD 服务端口
- --ca 连接使用的 TLS CA 文件路径
- --ssl-key 连接使用的 TLS 密钥文件路径
- --ssl-cert 连接使用的 TLS 证书文件路径

其中 --pdhost 和 --pdport 主要是用于 etcd 子命令，例如：tidb-ctl etcd ddlinfo。如不添加地址和端口将使用默认值，TiDB/PD 服务默认的地址是 127.0.0.1（服务地址只能使用IP 地址），TiDB 服务端口默认的端口是 10080，PD 服务端口默认的端口是 2379 连接选项是全局选项，适用于以下所有命令。

4.3.3.2.3 schema 命令

in 子命令

in 子命令用来通过数据库名获取数据库中所有表的表结构。

tidb-ctl schema in {数据库名}

如：tidb-ctl schema in mysql 将得到以下结果

```
[
    {
        "id": 13,
        "name": {
            "O": "columns_priv",
            "L": "columns_priv"
        },
              ...
        "update_timestamp": 399494726837600268,
        "ShardRowIDBits": 0,
        "Partition": null
    }
]
```

结果将以json 形式展示，内容较长，这里做了截断。

如希望指定表名，可以使用 tidb-ctl schema in {数据库名} -n {表名} 进行过滤。

如：tidb-ctl schema in mysql -n db 将得到mysql 库中db 表的表结构。结果如下：

```
{
    "id": 9,
    "name": {
        "O": "db",
        "L": "db"
    },
    ...
    "Partition": null
}
```

这里同样做了截断。

如使用的TiDB 地址不为默认地址和端口，可以使用命令行参数 --host, --port 选项，如：tidb-ctl --host <⟶ 172.16.55.88 --port 8898 schema in mysql -n db。

tid 子命令

tid 子命令用来通过表的id 获取数据库中表的表结构。

通过使用in 子命令查询到数据库中表的id，之后可以通过tid 子命令查看表的详细信息。

例如，查询到 mysql.stat_meta 表的id 是 21，可以通过 tidb-ctl schema tid -i 21 查看表的详细信息。

```
{
 "id": 21,
 "name": {
  "O": "stats_meta",
  "L": "stats_meta"
 },
 "charset": "utf8mb4",
 "collate": "utf8mb4_bin",
  ...
}
```

同 in 子命令一样，如果使用的TiDB 地址不是默认的地址和端口，需要通过 --host 和 --port 参数指定TiDB 的地址和 status 端口。

4.3.3.2.4 base64decode 命令

base64decode 用来解码base64 数据。

基本用法：

```
tidb-ctl base64decode [base64_data]
tidb-ctl base64decode [db_name.table_name] [base64_data]
tidb-ctl base64decode [table_id] [base64_data]
```

1. 准备环境，执行以下 SQL

```
use test;
create table t (a int, b varchar(20),c datetime default current_timestamp , d timestamp
    <⟶ default current_timestamp, unique index(a));
insert into t (a,b,c) values(1,"哈哈 hello",NULL);
alter table t add column e varchar(20);
```

2. 用 HTTP API 接口获取 MVCC 数据

```
curl "http://$IP:10080/mvcc/index/test/t/a/1?a=1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306449994645510,
    "commit_ts": 407306449994645513,
    "short_value": "AAAAAAAAAAE="     # unique index a 存的值是对应行的 handle id.
   }
  ]
 }
}%
```

```
curl "http://$IP:10080/mvcc/key/test/t/1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306588892692486,
    "commit_ts": 407306588892692489,
    "short_value": "CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuu0Rk=" # 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 CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuu0Rk=
```

```
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
```

4.3.3.2.5 decoder 命令

- 以下示例解码row key，index key 类似。

```
./tidb-ctl decoder "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\x00\x00\xfa"
```

```
format: table_row
table_id: -9223372036854775780
row_id: -9223372036854775558
```

- 以下示例解码 value

```
./tidb-ctl decoder AhZoZWxsbyB3b3JsZAiAEA==
```

```
format: index_value
index_value[0]: {type: bytes, value: hello world}
index_value[1]: {type: bigint, value: 1024}
```

4.3.3.2.6 etcd 命令

- `tidb-ctl etcd ddlinfo` 获取 DDL 信息。
- `tidb-ctl etcd putkey KEY VALUE` 添加KEY VALUE 到 etcd（所有的KEY 会添加到`/tidb/ddl/all_schema_versions <-> /` 之下）。

```
tidb-ctl etcd putkey "foo" "bar"
```

  实际是添加KEY 为 `/tidb/ddl/all_schema_versions/foo`，VALUE 为 bar 的键值对到 etcd 中。

- `tidb-ctl etcd delkey` 删除etcd 中的KEY，只有前缀以`/tidb/ddl/fg/owner/` 和`/tidb/ddl/all_schema_versions <-> /` 开头才允许被删除。

```
tidb-ctl etcd delkey "/tidb/ddl/fg/owner/foo" &&
tidb-ctl etcd delkey "/tidb/ddl/all_schema_versions/bar"
```

4.3.3.2.7 log 命令

TiDB 错误日志的堆栈信息是一行的格式，可以使用 `tidb-ctl log` 将堆栈信息格式化成多行形式。

4.3.3.2.8 keyrange 命令

keyrange 子命令用于查询全局或表相关的关键 key range 信息，以十六进制形式输出。

- 使用 `tidb-ctl keyrange` 命令查看全局的关键 key range。

```
tidb-ctl keyrange
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
```

- 添加 `--encode` 选项可以显示 encode 过的 key（与 TiKV 及 PD 中的格式相同）。

```
tidb-ctl keyrange --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
```

- 使用 `tidb-ctl keyrange --database={db} --table={tbl}` 命令查看全局和表相关的关键 key range。

```
tidb-ctl keyrange --database test --table ttt
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (748000000000000002f, 748000000000000030)
  table indexes: (74800000000000002f5f69, 74800000000000002f5f72)
    index c2: (74800000000000002f5f698000000000000001, 748000000000000002
        <-> f5f6980000000000000002)
    index c3: (74800000000000002f5f698000000000000002, 748000000000000002
        <-> f5f6980000000000000003)
    index c4: (74800000000000002f5f698000000000000003, 748000000000000002
        <-> f5f6980000000000000004)
  table rows: (74800000000000002f5f72, 748000000000000030)
```

4.3.4 PD Recover 使用文档

PD Recover 是对 PD 进行灾难性恢复的工具，用于恢复无法正常启动或服务的 PD 集群。

4.3.4.1 安装 PD Recover

要使用 PD Recover，你可以从源代码编译，也可以直接下载TiDB 工具包。

4.3.4.1.1 从源代码编译

- Go：PD Recover 使用了 Go 模块，请安装 Go 1.20 或以上版本。
- 在 PD 根目录下，运行 `make pd-recover` 命令来编译源代码并生成 `bin/pd-recover`。

> **注意：**
>
> 一般来说，用户不需要编译源代码，因为发布的二进制文件或 Docker 中已包含 PD Recover 工具。开发者可以参考以上步骤来编译源代码。

4.3.4.1.2 下载 TiDB 工具包

PD Recover 的安装包位于 TiDB 离线工具包中。下载方式，请参考 TiDB 工具下载。

下面介绍两种重建集群的方式：从存活的 PD 节点重建和完全重建。

4.3.4.2 方式一：从存活的 PD 节点重建集群

当 PD 集群的大多数节点发生灾难性故障时，集群将无法提供服务。当还有 PD 节点存活时，可以选择一个存活的 PD 节点，通过强制修改 Raft Group 的成员，使该节点重新恢复服务。具体操作步骤如下：

4.3.4.2.1 第 1 步：停止所有节点

停止集群中的 TiDB、TiKV 和 TiFlash 服务进程，以防止在恢复过程中与 PD 参数交互，造成数据错乱或其他无法挽救的异常状况。

4.3.4.2.2 第 2 步：启动存活的 PD 节点

使用启动参数 `--force-new-cluster` 拉起该存活的 PD 节点，如：

```
./bin/pd-server --force-new-cluster --name=pd-127.0.0.10-2379 --client-urls=http://0.0.0.0:2379
    <↪ --advertise-client-urls=http://127.0.0.1:2379 --peer-urls=http://0.0.0.0:2380 --advertise
    <↪ -peer-urls=http://127.0.0.1:2380 --config=conf/pd.toml
```

4.3.4.2.3 第 3 步：使用 pd-recover 修复元数据

该方法是利用少数派 PD 节点恢复服务，但由于该节点可能存在数据落后的情况，因此对于 `alloc_id` 和 `tso` 等数据，一旦发生回退，可能导致集群数据错乱或不可用。为确保该节点能提供正确的分配 ID 和 TSO 等服务，需要使用 `pd-recover` 修改元数据。具体命令参考：

```
./bin/pd-recover --from-old-member --endpoints=http://127.0.0.1:2379 # 指定对应的 PD 地址
```

> **注意：**
>
> 该步骤会自动将存储中的 `alloc_id` 增加一个安全值 100000000。这将导致后续集群中分配的 ID 偏大。
>
> 此外，`pd-recover` 不会修改TSO。因此，在执行此步骤之前，请确保本地时间晚于故障发生时间，并且确认故障前 PD 组件之间已开启 NTP 时钟同步服务。如果未开启，则需要将本地时钟调整到一个未来的时间，以确保TSO 不会回退。

4.3.4.2.4 第 4 步：重启这个 PD

当上一步出现 `recovery is successful` 的提示信息后，重启 PD。

4.3.4.2.5 第 5 步：扩容 PD 并启动集群

通过部署工具扩容 PD，并启动集群中的其他组件。至此服务恢复。

4.3.4.3 方式二：完全重建 PD 集群

该方式适用于所有 PD 的数据都丢失，但TiDB、 TiKV 和TiFlash 等其他组件数据都还存在的情况。

4.3.4.3.1 第 1 步：获取 Cluster ID

一般在 PD、 TiKV 或 TiDB 的日志中都可以获取 Cluster ID。你可以直接在服务器上查看日志以获取 Cluster ID。

从 PD 日志获取 Cluster ID（推荐）

使用以下命令，从 PD 日志中获取 Cluster ID：

```
cat {{/path/to}}/pd.log | grep "init cluster id"
```

```
[2019/10/14 10:35:38.880 +00:00] [INFO] [server.go:212] ["init cluster id"] [cluster-id
    ↪ =6747551640615446306]
...
```

或者也可以从TiDB 或 TiKV 的日志中获取。

从 TiDB 日志获取 Cluster ID

使用以下命令，从TiDB 日志中获取 Cluster ID：

```
cat {{/path/to}}/tidb.log | grep "init cluster id"
```

```
2019/10/14 19:23:04.688 client.go:161: [info] [pd] init cluster id 6747551640615446306
...
```

从 TiKV 日志获取 Cluster ID

使用以下命令，从TiKV 日志中获取 Cluster ID：

```
cat {{/path/to}}/tikv.log | grep "connect to PD cluster"
```

```
[2019/10/14 07:06:35.278 +00:00] [INFO] [tikv-server.rs:464] ["connect to PD cluster
    <-> 6747551640615446306"]
...
```

4.3.4.3.2 第 2 步：获取已分配ID

在指定已分配ID 时，需指定一个比当前最大的已分配ID 更大的值。可以从监控中获取已分配ID，也可以直接在服务器上查看日志。

从监控中获取已分配ID（推荐）

要从监控中获取已分配的ID，需要确保你所查看的监控指标是上一任 PD Leader 的指标。可从 PD Dashboard 中 Current ID allocation 面板获取最大的已分配ID。

从 PD 日志获取已分配ID

要从 PD 日志中获取分配的ID，需要确保你所查看的日志是上一任 PD Leader 的日志。运行以下命令获取最大的已分配ID：

```
cat {{/path/to}}/pd*.log | grep "idAllocator allocates a new id" |  awk -F'=' '{print $2}' | awk
    <-> -F']' '{print $1}' | sort -r -n | head -n 1
```

```
4000
...
```

你也可以在所有 PD server 中运行上述命令，找到最大的值。

4.3.4.3.3 第 3 步：部署一套新的 PD 集群

部署新的 PD 集群之前，需要停止当前的 PD 集群，然后删除旧的数据目录（或者用 `--data-dir` 指定新的数据目录）。

4.3.4.3.4 第 4 步：使用pd-recover

只需在一个 PD 节点上执行 `pd-recover` 即可。

```
./pd-recover -endpoints http://10.0.1.13:2379 -cluster-id 6747551640615446306 -alloc-id 10000
```

4.3.4.3.5 第 5 步：重启整个集群

当出现 `recovery is successful` 的提示信息时，重启整个集群。

4.3.4.4 常见问题

4.3.4.4.1 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。

4.3.4.4.2 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。

4.3.5 binlogctl 工具

Binlog Control（以下简称 binlogctl）是 TiDB Binlog 的命令行工具，用于管理 TiDB Binlog 集群。

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

使用 binlogctl 的场景：

- 同步出现故障/检查运行情况，需要查看 Pump/Drainer 的状态
- 维护集群，需要暂停/下线 Pump/Drainer
- Pump/Drainer 异常退出，状态没有更新，或者状态不符合预期，对业务造成影响

4.3.5.1 binlogctl 下载

> 注意：
> 建议使用的 Control 工具版本与集群版本保持一致。

binlogctl 的安装包位于 TiDB 离线工具包中。下载方式，请参考 TiDB 工具下载。

4.3.5.2 binlogctl 使用说明

命令行参数：

```
Usage of binlogctl:
-V
输出 binlogctl 的版本信息
-cmd string
    命令模式，包括 "generate_meta"（已废弃）, "pumps", "drainers", "update-pump" ,"update-drainer
        <-> ", "pause-pump", "pause-drainer", "offline-pump", "offline-drainer"
-data-dir string
    保存 Drainer 的 checkpoint 的文件的路径（默认 "binlog_position"）（已废弃）
-node-id string
```

```
    Pump/Drainer 的 ID
-pd-urls string
    PD 的地址，如果有多个，则用"," 连接 (默认 "http://127.0.0.1:2379")
-ssl-ca string
    SSL CAs 文件的路径
-ssl-cert string
        PEM 格式的 X509 认证文件的路径
-ssl-key string
        PEM 格式的 X509 key 文件的路径
-time-zone string
    如果设置时区，在 "generate_meta" 模式下会打印出获取到的 tso 对应的时间。例如 "Asia/Shanghai"
        <-> 为 CST 时区，"Local" 为本地时区
-show-offline-nodes
    在用 `-cmd pumps` 或 `-cmd drainers` 命令时使用，这两个命令默认不显示 offline 的节点，
        <-> 仅当明确指定 `-show-offline-nodes` 时会显示
```

命令示例：

- 查询所有的 Pump/Drainer 的状态：

  设置 cmd 为 pumps 或者 drainers 来查看所有 Pump 或者 Drainer 的状态。例如：

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pumps
```

```
[2019/04/28 09:29:59.016 +00:00] [INFO] [nodes.go:48] ["query node"] [type=pump] [node="{
    <-> NodeID: 1.1.1.1:8250, Addr: pump:8250, State: online, MaxCommitTS:
    <-> 408012403141509121, UpdateTime: 2019-04-28 09:29:57 +0000 UTC}"]
```

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd drainers
```

```
[2019/04/28 09:29:59.016 +00:00] [INFO] [nodes.go:48] ["query node"] [type=drainer] [node="{
    <-> NodeID: 1.1.1.1:8249, Addr: 1.1.1.1:8249, State: online, MaxCommitTS:
    <-> 408012403141509121, UpdateTime: 2019-04-28 09:29:57 +0000 UTC}"]
```

- 暂停/下线 Pump/Drainer

  binlogctl 提供以下命令暂停/下线服务：

cmd	说明	示例
pause-pump	暂停 Pump	bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pause-pump -node-<-> id ip-127-0-0-1:8250
pause-drainer	暂停 Drainer	bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pause-drainer -<-> node-id ip-127-0-0-1:8249
offline-pump	下线 Pump	bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd offline-pump -node<-> -id ip-127-0-0-1:8250
offline-drainer	下线 Drainer	bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd offline-drainer -<-> node-id ip-127-0-0-1:8249

binlogctl 会发送 HTTP 请求给 Pump/Drainer，Pump/Drainer 收到命令后会主动执行对应的退出流程。

- 异常情况下修改 Pump/Drainer 的状态

  在服务正常运行以及符合流程的暂停、下线过程中，Pump/Drainer 的状态都是可以正确的。但是在一些异常情况下 Pump/Drainer 无法正确维护自己的状态，可能会影响数据同步任务，在这种情况下需要使用 binlogctl 修复状态信息。

  设置 `cmd` 为 `update-pump` 或者 `update-drainer` 来更新 Pump 或者 Drainer 的状态。Pump 和 Drainer 的状态可以为paused 或者 offline。例如：

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd update-pump -node-id ip-127-0-0-1:8250 -
    <-> state paused
```

> 注意：
> Pump/Drainer 在正常运行过程中会定期在 PD 中更新自己的状态，而这条命令是直接去修改 Pump/Drainer 保存在 PD 中的状态，所以在 Pump/Drainer 服务正常的情况下使用这些命令是没有意义的。仅在 Pump/Drainer 服务异常的情况下使用，具体哪些场景下使用这条命令可以参考FAQ。

4.4 命令行参数

4.4.1 TiDB 配置参数

在启动TiDB 时，你可以使用命令行参数或环境变量来配置TiDB。

要快速了解TiDB 的参数体系与参数作用域，建议先观看下面的培训视频（时长 17 分钟）。

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）。

4.4.1.1 `--advertise-address`

- 登录TiDB 的 IP 地址
- 默认：“”
- 必须确保用户和集群中的其他机器都能够访问到该IP 地址

4.4.1.2 `--config`

- 配置文件
- 默认：“”
- 如果你指定了配置文件，TiDB 会首先读取配置文件的配置。如果对应的配置在命令行参数里面也存在，TiDB 就会使用命令行参数的配置来覆盖配置文件中的配置。详细的配置项请参阅TiDB 配置文件描述。

4.4.1.3 `--config-check`

- 检查配置文件的有效性并退出
- 默认：false

4.4.1.4 --config-strict

- 增强配置文件的有效性
- 默认：false

4.4.1.5 --cors

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

4.4.1.6 --enable-binlog

- 开启或关闭TiDB 中 binlog 的生成
- 默认：false

4.4.1.7 --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

4.4.1.8 --initialize-insecure

- 在不安全模式下启动tidb-server
- 默认：true

4.4.1.9 --initialize-secure

- 在安全模式下启动tidb-server
- 默认：false

4.4.1.10 --initialize-sql-file

- 用于指定TiDB 集群初次启动时执行的 SQL 脚本。参考配置项 initialize-sql-file
- 默认：“”

4.4.1.11 -L

- Log 级别
- 默认：“info”
- 可选：“debug”，“info”，“warn”，“error”，“fatal”

4.4.1.12 --lease

- Schema lease 的持续时间。除非你知道更改该值带来的后果，否则你的更改操作是危险的。
- 默认：45s

4.4.1.13 --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。

4.4.1.14 --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

4.4.1.15 --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

4.4.1.16 --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

4.4.1.17 -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

4.4.1.18 --path

- 对于本地存储引擎“unistore”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 = unistore 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P D，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=unistore --path="" 来启动一个纯内存引擎的TiDB

4.4.1.19 --proxy-protocol-networks

- 允许使用 PROXY 协议连接TiDB 的代理服务器地址列表。
- 默认：“”
- 通常情况下，通过反向代理使用TiDB 时，TiDB 会将反向代理服务器的IP 地址视为客户端IP 地址。对于支持 PROXY 协议的反向代理（如 HAProxy），开启 PROXY 协议后能让反向代理透传客户端真实的IP 地址给TiDB。
- 配置该参数后，TiDB 将允许配置的源IP 地址使用 PROXY 协议连接到TiDB，且拒绝这些源IP 地址使用非PROXY 协议连接。其他地址可以使用非 PROXY 协议连接到TiDB。若该参数为空，则任何源IP 地址都不能使用 PROXY 协议连接到TiDB。地址可以使用IP 地址格式(192.168.1.50) 或者CIDR 格式(192.168.1.0/24)，并可用 , 分隔多个地址，或用* 代表所有IP 地址。

> **警告：**
>
> 需谨慎使用* 符号，因为它可能引入安全风险，允许来自任何IP 的客户端自行汇报其IP 地址。另外，当--proxy-protocol-fallbackable 设置为 true 以外的值时，使用* 可能会导致部分直接连接TiDB 的内部组件无法使用，例如TiDB Dashboard。

> **注意：**
>
> 如果使用AWS 的 Network Load Balancer (NLB) 并开启 PROXY 协议，需要设置 NLB 的 target group 属性：将proxy_protocol_v2.client_to_server.header_place 设为on_first_ack。同时向AWS 的 Support 提工单开通此功能的支持。注意，AWS NLB 在开启 PROXY 协议后，客户端将无法获取服务器端的握手报文，因此报文会一直阻塞到客户端超时。这是因为，NLB 默认只在客户端发送数据之后才会发送 PROXY 的报文，而在客户端发送数据包之前，服务器端发送的任何数据包都会在内网被丢弃。

4.4.1.20 --proxy-protocol-fallbackable

- 用于控制是否启用 PROXY 协议回退模式。如果设置为 true，TiDB 可以接受属于 --proxy-protocol-networks 的客户端使用非 PROXY 协议规范或者没有发送 PROXY 协议头的客户端连接。默认情况下，TiDB 仅接受属于 --proxy-protocol-networks 的客户端发送 PROXY 协议头的客户端连接。
- 默认：false

4.4.1.21 --proxy-protocol-header-timeout

- PROXY 协议请求头读取超时时间
- 默认：5
- 单位：秒

> **警告：**
>
> 自 v6.3.0 起，该参数被废弃。因为自 v6.3.0 起，读取 PROXY 协议报头的操作会在第一次读取网络数据时进行，废弃该参数可避免影响首次读取网络数据时设置的超时时间。

> **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。

4.4.1.22 `--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

4.4.1.23 `--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

4.4.1.24 `--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

4.4.1.25 `--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

4.4.1.26 `--status-host`

- TiDB 服务状态监听host
- 默认：“0.0.0.0”

4.4.1.27 --store

- 用来指定TiDB 底层使用的存储引擎
- 默认：“unistore”
- 可以选择“unistore”（本地存储引擎）或者“tikv”（分布式存储引擎）

4.4.1.28 --temp-dir

- TiDB 用于存放临时文件的目录
- 默认：“/tmp/tidb”

4.4.1.29 --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

4.4.1.30 -V

- 输出TiDB 的版本
- 默认：“”

4.4.1.31 --plugin-dir

- plugin 存放目录
- 默认：“/data/deploy/plugin”

4.4.1.32 --plugin-load

- 需要加载的plugin 名称，多个plugin 以“,”逗号分隔
- 默认：“”

4.4.1.33 --affinity-cpus

- 设置TiDB server CPU 亲和性，以“,”逗号分隔，例如“1,2,3”
- 默认：“”

4.4.1.34 --repair-mode

- 是否开启修复模式，仅用于数据修复场景
- 默认：false

4.4.1.35 --repair-list

- 修复模式下需要修复的表名
- 默认：“”

4.4.2 TiKV 配置参数

TiKV 的命令行参数支持一些可读性好的单位转换。

- 文件大小（以bytes 为单位）：KB，MB, GB, TB, PB（也可以全小写）
- 时间（以毫秒为单位）：ms, s, m, h

4.4.2.1 -A，--addr

- TiKV 监听地址。
- 默认：“127.0.0.1:20160”
- 如果部署一个集群，--addr 必须指定当前主机的IP 地址，例如“192.168.100.113:20160”，如果是运行在 docker 则需要指定为“0.0.0.0:20160”。

4.4.2.2 --advertise-addr

- TiKV 对外访问地址。
- 默认：${addr}
- 在某些情况下，比如 Docker 或者 NAT 网络环境，客户端并不能通过 --addr 的地址来访问到TiKV。这时候，你可以设置 --advertise-addr 来让客户端访问TiKV。
- 例如，docker 内部 IP 地址为 172.17.0.1， 而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20160:20160， 那么可以设置为 --advertise-addr=“192.168.100.113:20160”， 客户端可以通过 192.168.100.113:20160 来找到这个服务。

4.4.2.3 --status-addr

- TiKV 服务状态监听端口。
- 默认：“20180”
- Prometheus 统计可以通过 http://host:status_port/metrics 访问。
- Profile 数据可以通过 http://host:status_port/debug/pprof/profile 访问。

4.4.2.4 --advertise-status-addr

- TiKV 对外访问服务状态地址。
- 默认：使用 --status-addr
- 在某些情况下，例如docker 或者 NAT 网络环境，客户端并不能通过 --status-addr 的地址来访问到TiKV。此时，你可以设置 --advertise-status-addr 来让客户端访问TiKV。
- 例如，Docker 内部IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20180:20180，那么可以设置 --advertise-status-addr="192.168.100.113:20180"，客户端可以通过 192.168.100.113:20180 来找到这个服务。

4.4.2.5 -C, --config

- 配置文件。
- 默认：“”
- 如果你指定了配置文件，TiKV 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，TiKV 就会使用命令行参数的配置来覆盖配置文件里面的。

4.4.2.6 --capacity

- TiKV 存储数据的容量。
- 默认：0（无限）
- PD 需要使用这个值来对整个集群做balance 操作。（提示：你可以使用 10GB 来替代 10737418240，从而简化参数的传递）。

4.4.2.7 --config-info <FORMAT>

- 按照指定的 FORMAT 输出各个配置项的取值信息并退出。
- FORMAT 可选值：json。
- 目前仅支持以JSON 格式输出每个配置项的名字(Name)、默认值(DefaultValue) 和当前配置值(ValueInFile)。当执行此命令时，若同时指定了 -C 或 --config 参数，则对应的配置文件包含的配置项会同时输出当前配置值和默认值，其他未指定的配置项仅输出默认值，示例如下：

```
json {    "Component": "TiKV Server", "Version": "6.2.0", "Parameters": [ { "Name": "log-level",
<⟶ "DefaultValue": "info", "ValueInFile": "warn" },    {       "Name": "log-file",  "DefaultValue
<⟶ ": "" },     ...    ]  }
```

4.4.2.8 --data-dir

- TiKV 数据存储路径。
- 默认：“/tmp/tikv/store”

4.4.2.9 -L

- Log 级别。
- 默认：“info”
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”

4.4.2.10 --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。

4.4.2.11 --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。

4.4.3 TiFlash 命令行参数

本文介绍了TiFlash 的命令行启动参数。

4.4.3.1 server --config-file

- 指定TiFlash 的配置文件路径
- 默认：“”
- 必须指定配置文件，详细的配置项请参阅TiFlash 配置参数

4.4.3.2 dttool migrate

- 迁移 DTFile 的文件格式（用于测试和原地降级）。数据迁移的单位为单个 DTFile。如果想进行整表迁移，通常需要定位到所有形如 `<data dir>/t_<table id>/stable/dmf_<file id>` 的路径，逐一进行迁移。可以结合脚本来自动进行这一操作。
- 使用场景：
  - 当需要从开启了数据校验功能(`storage.format_version` >= 3) 的 TiFlash v5.4 及以上版本降级回以前的版本时，可以使用此工具完成数据格式降级。
  - 当升级到TiFlash v5.4 及以上，并希望对存量数据也加上数据校验功能以加固数据检验时，可以使用此工具完成数据格式升级。
  - 测试不同配置的 DTFile 空间占用和读取速度。
- 参数：
  - `--imitative`：当不使用 DTFile 的加密功能时，可以使用本选项避免使用配置文件和连接 PD。
  - `--version`：DTFile 的版本，可选值为 1 和 2，默认为 2。1 为传统格式，2 为新版checksum 对应的 DTFile 格式。
  - `--algorithm`：检验哈希算法，可选值为xxh3，city128，crc32，crc64，none，默认为xxh3，仅在version=2 时有用。
  - `--frame`：校验帧大小，默认为 1048576，仅在 version=2 时有用。
  - `--compression`：目标压缩算法，可选值为 LZ4（默认）、LZ4HC、zstd 和none。
  - `--level`：目标压缩等级，不指定则根据压缩算法默认使用推荐的压缩级别。如果 `compression` 设置为 `LZ4` 或 `zstd`，则默认设置为 1；如果 `compression` 设置为 `LZ4HC`，则默认设置为 9。
  - `--config-file`：dttool migrate 的配置文件应当与 server 模式下的配置文件保持一致。见 `--imitative` 选项。
  - `--file-id`：对应 DTFile 的 ID，如 `dmf_123` 对应的ID 是 123。

- `--workdir`：指向 `dmf_xxx` 的父级目录。
- `--dry`：空跑模式，只输出迁移过程。
- `--nokeep`：不保留原数据。不开启该选项时，会产生 `dmf_xxx.old` 文件。

> **警告：**
>
> 虽然TiFlash 可以读取自定义压缩算法和压缩等级的 DTFile，但目前正式支持的只有默认压缩等级的 LZ4 算法。自定义压缩参数并未经过大量测试，仅作实验。

> **注意：**
>
> 为保证安全 DTTool 在迁移模式下会尝试对工作目录进行加锁，因此同一工作目录下同一时间只能有一个 DTTool 执行迁移工作。如果您在中途强制停止 DTTool，可能会因锁未释放导致后面在运行 DTTool 时工具拒绝进行迁移工作。如果您遇到这种情况，在保证安全的前提下，可以手动删除工作目录下的 LOCK 文件来释放锁。

4.4.3.3 `dttool bench`

- 提供 DTFile 的简单IO 速度测试。
- 参数：
  - `--version`：DTFile 的版本，见dttool migrate 对应参数。
  - `--algorithm`：检验哈希算法，见dttool migrate 对应参数。
  - `--frame`：校验帧大小，见dttool migrate 对应参数。
  - `--column`：测试表宽度，默认为 100。
  - `--size`：测试表长度，默认为 1000。
  - `--field`：测试表字段长度上限，默认为 1024。
  - `--random`：随机数种子。如未提供，该值从系统熵池抽取。
  - `--encryption`：启用加密功能。
  - `--repeat`：性能测试采样次数，默认为 5。
  - `--workdir`：临时数据文件夹，应指向需要测试的文件系统下的路径，默认为 /tmp/test。

4.4.3.4 `dttool inspect`

- 检查 DTFile 的完整性。数据校验的单位为单个 DTFile。如果想进行整表校验，通常需要定位到所有形如 `<data dir>/t_<table id>/stable/dmf_<file id>` 的路径，逐一进行校验。可以结合脚本来自动进行这一操作。
- 使用场景：
  - 完成格式升降级后进行完整性检测。

- 将原有数据文件搬迁至新环境后进行完整性检测。

- 参数：

  - `--config-file`：dttool bench 的配置文件，见dttool migrate 对应参数。
  - `--check`：进行哈希校验。
  - `--file-id`：对应 DTFile 的 ID，见dttool migrate 对应参数。
  - `--imitative`：模拟数据库上下文，见dttool migrate 对应参数。
  - `--workdir`：数据文件夹，见dttool migrate 对应参数。

4.4.4 PD 配置参数

PD 可以通过命令行参数或环境变量配置。

4.4.4.1 `--advertise-client-urls`

- 用于外部访问 PD 的 URL 列表。
- 默认：`${client-urls}`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2379:2379`，那么可以设置为 `--advertise-client-urls="http://192.168.100.113:2379"`，客户端可以通过 `http://192.168.100.113:2379` 来找到这个服务。

4.4.4.2 `--advertise-peer-urls`

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认：`${peer-urls}`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2380:2380`，那么可以设置为 `advertise-peer-urls="http://192.168.100.113:2380"`，其他 PD 节点可以通过 `http://192.168.100.113:2380` 来找到这个服务。

4.4.4.3 `--client-urls`

- PD 监听的客户端 URL 列表。
- 默认：`http://127.0.0.1:2379`
- 如果部署一个集群，`--client-urls` 必须指定当前主机的IP 地址，例如 `http://192.168.100.113:2379"`，如果是运行在 Docker 则需要指定为 `http://0.0.0.0:2379`。

4.4.4.4 --peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认：http://127.0.0.1:2380
- 如果部署一个集群，--peer-urls 必须指定当前主机的IP 地址，例如 http://192.168.100.113:2380，如果是运行在 Docker 则需要指定为 http://0.0.0.0:2380。

4.4.4.5 --config

- 配置文件。
- 默认：“”
- 如果你指定了配置文件，PD 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，PD 就会使用命令行参数的配置来覆盖配置文件里面的。

4.4.4.6 --data-dir

- PD 存储数据路径。
- 默认：default.${name}

4.4.4.7 --initial-cluster

- 初始化 PD 集群配置。
- 默认："{name}=http://{advertise-peer-url}"
- 例如，如果name 是“pd”，并且 advertise-peer-urls 是 http://192.168.100.113:2380，那么 initial-cluster 就是 pd=http://192.168.100.113:2380。
- 如果你需要启动三台PD，那么initial-cluster 可能就是pd1=http://192.168.100.113:2380, pd2=http://192.168.100.114:2380, pd3=192.168.100.115:2380。

4.4.4.8 --join

- 动态加入 PD 集群。
- 默认：“”
- 如果你想动态将一台PD 加入集群，你可以使用--join="${advertise-client-urls}"，advertise-client-url 是当前集群里面任意 PD 的 advertise-client-url，你也可以使用多个 PD 的，需要用逗号分隔。

4.4.4.9 -L

- Log 级别。
- 默认：“info”
- 可选：“debug”，“info”，“warn”，“error”，“fatal”

4.4.4.10 --log-file

- Log 文件。
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面。

4.4.4.11 --log-rotate

- 是否开启日志切割。
- 默认：true
- 当值为true时，按照 PD 配置文件中 [log.file] 信息执行。

4.4.4.12 --name

- 当前 PD 的名字。
- 默认：“pd”
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字

4.4.4.13 --cacert

- CA 文件路径，用于开启 TLS。
- 默认：“”

4.4.4.14 --cert

- 包含X509证书的PEM 文件路径，用户开启 TLS。
- 默认：“”

4.4.4.15 --key

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

4.4.4.16 --metrics-addr

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

4.4.4.17 --force-new-cluster

- 强制使用当前节点创建新的集群。
- 默认：false
- 仅用于在 PD 丢失多数副本的情况下恢复服务，可能会产生部分数据丢失。

4.4.4.18 -V, --version

- 输出版本信息并退出。

4.5 监控指标

4.5.1 Overview 面板重要监控指标详解

使用TiUP 部署TiDB 集群时，一键部署监控系统(Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_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：

4.5.1.1 Services Port Status

- Services Up：各服务在线节点数量

4.5.1.2 PD

- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小，TiKV 多副本的空间占用也会包含在内
- Normal stores：处于正常状态的节点数目
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Number of Regions：当前集群的Region 总量，请注意Region 数量与副本数无关
- 99% completed_cmds_duration_seconds：单位时间内，99% 的 pd-server 请求执行时间小于监控曲线的值，一般 <= 5ms
- Handle_requests_duration_seconds：PD 发送请求的网络耗时
- Region health：每个 Region 的状态，通常情况下，pending 的peer 应该少于 100，miss 的peer 不能一直大于 0
- Hot write Region’s leader distribution：每个TiKV 实例上是写入热点的leader 的数量
- Hot read Region’s leader distribution：每个TiKV 实例上是读取热点的leader 的数量
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

4.5.1.3 TiDB

- Statement OPS：不同类型 SQL 语句每秒执行的数量。按 SELECT、 INSERT、 UPDATE 等来统计
- Duration：执行的时间
  - 客户端网络请求发送到TiDB，到TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行的时间

  - 由于TiDB 支持 Multi-Query，因此，可以接受客户端一次性发送的多条 SQL 语句，如：`select 1;` <—> `select 1; select 1;`。此时，统计的执行时间是所有 SQL 执行完之后的总时间

- CPS By Instance：每个TiDB 实例上的命令统计。按照命令和执行结果成功或失败来统计
- Failed Query OPM：每个 TiDB 实例上，每秒钟执行 SQL 语句发生错误按照错误类型的统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码
- Connection count：每个TiDB 的连接数
- Memory Usage：每个 TiDB 实例的内存使用统计，分为进程占用内存和Golang 在堆上申请的内存
- Transaction OPS：每秒事务执行数量统计
- Transaction Duration：事务执行的时间
- KV Cmd OPS：KV 命令执行数量统计
- KV Cmd Duration 99：KV 命令执行的时间
- PD TSO OPS：TiDB 每秒从 PD 获取TSO 的数量
- PD TSO Wait Duration：TiDB 等待从 PD 获取TS 的时间
- TiClient Region Error OPS：TiKV 返回Region 相关错误信息的数量
- Lock Resolve OPS：TiDB 清理锁操作的数量。当TiDB 的读写请求遇到锁时，会尝试进行锁清理
- Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
- KV Backoff OPS：TiKV 返回错误信息的数量

4.5.1.4 TiKV

- leader：各个TiKV 节点上 Leader 的数量分布
- region：各个TiKV 节点上Region 的数量分布
- CPU：各个 TiKV 节点的 CPU 使用率
- Memory：各个 TiKV 节点的内存使用量
- store size：每个 TiKV 实例的使用的存储空间的大小
- cf size：每个列族的大小
- channel full：每个 TiKV 实例上channel full 错误的数量，正常情况下应当为 0
- server report failures：每个 TiKV 实例上报错的消息个数，正常情况下应当为 0
- scheduler pending commands：每个TiKV 实例上pending 命令的个数
- coprocessor executor count：TiKV 每秒收到的coprocessor 操作数量，按照coprocessor 类型统计
- coprocessor request duration：处理coprocessor 读请求所花费的时间
- raft store CPU：raftstore 线程的 CPU 使用率，线程数量默认为 2（通过 `raftstore.store-pool-size` 配置）。如果单个线程使用率超过 80%，说明使用率很高
- Coprocessor CPU：coprocessor 线程的 CPU 使用率

4.5.1.5 System Info

- Vcores：CPU 核心数量
- Memory：内存总大小
- CPU Usage：CPU 使用率，最大为 100%
- Load [1m]：1 分钟的负载情况
- Memory Available：剩余内存大小
- Network Traffic：网卡流量统计
- TCP Retrans：TCP 重传数量统计
- IO Util：磁盘使用率，最高为 100%，一般到 80% - 90% 就需要考虑加节点

4.5.1.6 图例



图 51: overview

4.5.2 Performance Overview 面板重要监控指标详解

使用TiUP 部署TiDB 集群时，你可以一键部署监控系统 (Prometheus & Grafana)。监控架构参见 TiDB 监控框架概述。

目前Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

Performance Overview Dashboard 按总分结构对TiDB、 TiKV、 PD 的性能指标进行编排组织，包含了以下三部分内容：

- 总的概览：数据库时间和 SQL 执行时间概览。通过颜色优化法，你可以快速识别数据库负载特征和性能瓶颈。
- 资源负载：关键指标和资源利用率，包含数据库 QPS、应用和数据库的连接信息和请求命令类型、数据库内部TSO 和 KV 请求 OPS、 TiDB 和 TiKV 的资源使用概况。
- 自上而下的延迟分解：Query 延迟和连接空闲时间对比、 Query 延迟分解、 execute 阶段TSO 请求和 KV 请求的延迟、 TiKV 内部写延迟的分解等。

借助 Performance Overview Dashboard，你可以高效地进行性能分析，确认用户响应时间的瓶颈是否在数据库中。如果数据库是整个系统的瓶颈，通过数据库时间概览和 SQL 延迟的分解，定位数据库内部的瓶颈点，并进行针对性的优化。详情请参考TiDB 性能分析和优化方法。

以下为 Performance Overview Dashboard 监控说明：

4.5.2.1 Performance Overview

4.5.2.1.1 Database Time by SQL Type

- database time: 每秒的总数据库时间
- sql_type: 每种 SQL 语句每秒消耗的数据库时间

4.5.2.1.2 Database Time by SQL Phase

- database time: 每秒的总数据库时间
- get token/parse/compile/execute: 4 个 SQL 处理阶段每秒消耗的数据库时间

execute 执行阶段为绿色，其他三个阶段偏红色系，如果非绿色的颜色占比明显，意味着在执行阶段之外数据库消耗了过多时间，需要进一步分析根源。

4.5.2.1.3 SQL Execute Time Overview

- execute time: execute 阶段每秒消耗的数据库时间
- tso_wait: execute 阶段每秒同步等待TSO 的时间
- kv request type: execute 阶段每秒等待每种 KV 请求类型的时间，总的 KV request 等待时间可能超过 execute time，因为 KV request 是并发的。
- tiflash_mpp: execute 阶段每秒TiFlash 请求处理时间。

绿色系标识代表常规的写 KV 请求（例如 Prewrite 和 Commit），蓝色系标识代表常规的读 KV 请求，紫色系标识代表TiFlash MPP 请求，其他色系标识需要注意的问题。例如，悲观锁加锁请求为红色，TSO 等待为深褐色。如果非蓝色系或者非绿色系占比明显，意味着执行阶段存在异常的瓶颈。例如，当发生严重锁冲突时，红色的悲观锁时间会占比明显；当负载中TSO 等待的消耗时间过长时，深褐色会占比明显。

4.5.2.1.4 QPS

QPS：按 SELECT、 INSERT、 UPDATE 等类型统计所有TiDB 实例上每秒执行的 SQL 语句数量

4.5.2.1.5 CPS By Type

CPS By Type：按照类型统计所有TiDB 实例每秒处理的命令数（Command Per Second）

4.5.2.1.6 Queries Using Plan Cache OPS

- avg-hit：所有TiDB 实例每秒执行计划缓存的命中次数
- avg-miss：所有TiDB 实例每秒执行计划缓存的未命中次数

`avg-hit + avg-miss` 等于 StmtExecute 每秒执行次数。

4.5.2.1.7 KV/TSO Request OPS

- kv request total: 所有TiDB 实例每秒总的 KV 请求数量
- kv request by type: 按 `Get`、`Prewrite`、`Commit` 等类型统计在所有TiDB 实例每秒的请求数据
- tso - cmd：在所有TiDB 实例每秒tso cmd 的请求数量
- tso - request：在所有TiDB 实例每秒tso request 的请求数量

通常tso - cmd 除以tso - request 等于平均请求的batch 大小。

4.5.2.1.8 KV Request Time By Source

- kv request total time: 所有TiDB 实例每秒总的 KV 和 TiFlash 请求处理时间。
- 每种 KV 请求和请求来源组成柱状堆叠图，`external` 标识正常业务的请求，`internal` 标识内部活动的请求（比如 DDL、auto analyze 等请求）。

4.5.2.1.9 TiDB CPU

- avg：所有TiDB 实例平均 CPU 利用率
- delta：所有TiDB 实例中最大 CPU 利用率减去所有TiDB 实例中最小 CPU 利用率
- max：所有TiDB 实例中最大 CPU 利用率

4.5.2.1.10 TiKV CPU/IO MBps

- CPU-Avg：所有 TiKV 实例平均 CPU 利用率
- CPU-Delta：所有 TiKV 实例中最大 CPU 利用率减去所有TiKV 实例中最小 CPU 利用率
- CPU-MAX：所有 TiKV 实例中最大 CPU 利用率
- IO-Avg：所有TiKV 实例平均 MBps
- IO-Delta：所有TiKV 实例中最大 MBps 减去所有TiKV 实例中最小 MBps
- IO-MAX：所有TiKV 实例中最大 MBps

4.5.2.1.11 Duration

- Duration：执行时间解释
  - 从客户端网络请求发送到TiDB，到TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以SQL 语句的形式发送，但也可以包含COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行时间。

  - 由于TiDB 支持 Multi-Query，因此，客户端可以一次性发送多条 SQL 语句，如 `select 1; select 1;` <—> `select 1;`。此时的执行时间是所有 SQL 语句执行完成的总时间。

- avg：所有请求命令的平均执行时间
- 99：所有请求命令的 P99 执行时间
- avg by type：按 SELECT、 INSERT、 UPDATE 类型统计所有TiDB 实例上所有请求命令的平均执行时间

4.5.2.1.12 Connection Idle Duration

Connection Idle Duration 指空闲连接的持续时间。

- avg-in-txn：处于事务中，空闲连接的平均持续时间
- avg-not-in-txn：没有处于事务中，空闲连接的平均持续时间
- 99-in-txn：处于事务中，空闲连接的 P99 持续时间

4.5.2.1.13 Connection Count

- total：所有TiDB 节点的总连接数
- active connections：所有TiDB 节点的总活跃连接数
- tidb-{node-number}-peer：各个TiDB 节点的连接数
- disconnection/s：集群每秒断开连接的数量
- 99-not-in-txn：没有处于事务中，空闲连接的 P99 持续时间

4.5.2.1.14 Parse Duration、 Compile Duration 和 Execute Duration

- Parse Duration：SQL 语句解析耗时统计
- Compile Duration：将解析后的 SQL AST 编译成执行计划的耗时
- Execution Duration：执行 SQL 语句执行计划耗时

这三个时间指标均包含均所有TiDB 实例的平均值和 P99 值。

4.5.2.1.15 Avg TiDB KV Request Duration

按 `Get`、 `Prewrite`、 `Commit` 等类型统计在所有TiDB 实例 KV 请求的平均执行时间。

4.5.2.1.16 Avg TiKV GRPC Duration

按 `get`、 `kv_prewrite`、 `kv_commit` 等类型统计所有TiKV 实例对 gRPC 请求的平均执行时间。

4.5.2.1.17 PD TSO Wait/RPC Duration

- wait - avg：所有TiDB 实例等待从 PD 返回TSO 的平均时间
- rpc - avg：所有TiDB 实例从向 PD 发送获取TSO 的请求到接收到TSO 的平均耗时
- wait - 99：所有TiDB 实例等待从 PD 返回TSO 的 P99 时间
- rpc - 99：所有TiDB 实例从向 PD 发送获取TSO 的请求到接收到TSO 的 P99 耗时

4.5.2.1.18 Storage Async Write Duration、Store Duration 和 Apply Duration

- Storage Async Write Duration：异步写所花费的时间
- Store Duration：异步写 Store 步骤所花费的时间
- Apply Duration：异步写 Apply 步骤所花费的时间

这三个时间指标都包含所有TiKV 实例的平均值和 P99 值

平均 Storage async write duration = 平均 Store Duration + 平均 Apply Duration

4.5.2.1.19 Append Log Duration、Commit Log Duration 和 Apply Log Duration

- Append Log Duration：Raft append 日志所花费的时间
- Commit Log Duration：Raft commit 日志所花费的时间
- Apply Log Duration：Raft apply 日志所花费的时间

这三个时间指标均包含所有TiKV 实例的平均值和 P99 值。

4.5.2.1.20 图例



图 52: performance overview

4.5.2.2 TiFlash

- CPU：每个 TiFlash 实例 CPU 的使用率
- Memory：每个 TiFlash 实例内存的使用情况
- IO utilization：每个TiFlash 实例的IO 使用率

- MPP Query count：每个TiFlash 实例每秒 MPP 查询数量
- Request QPS：所有 TiFlash 实例收到的coprocessor 请求数量。
  - batch：batch 请求数量
  - batch_cop：batch 请求中的coprocessor 请求数量
  - cop：直接通过 coprocessor 接口发送的coprocessor 请求数量
  - cop_dag：所有 coprocessor 请求中dag 请求数量
  - super_batch：开启 super batch 特性的请求数量
- Executor QPS：所有 TiFlash 实例收到的请求中，每种 dag 算子的数量，其中 table_scan 是扫表算子，selection 是过滤算子，aggregation 是聚合算子，top_n 是TopN 算子，limit 是limit 算子
- Request Duration Overview：每秒所有 TiFlash 实例所有请求类型总处理时间的堆叠图
- Request Duration：所有 TiFlash 实例每种 MPP 和 coprocessor 请求类型的总处理时间，此时间为接收到该 coprocessor 请求至请求应答完毕的时间，包含平均和 P99 处理延迟
- Request Handle Duration：所有 TiFlash 实例每种 MPP 和 coprocessor 请求的处理时间，此时间为该coprocessor 请求从开始执行到结束的时间，包含平均和 P99 延迟
- Raft Wait Index Duration：所有 TiFlash 实例在进行 wait_index 消耗的时间，即拿到read_index 请求后，等待本地的 Region index >= read_index 所花费的时间
- Raft Batch Read Index Duration：所有 TiFlash 实例在进行read_index 消耗的时间，主要消耗在于和Region leader 的交互和重试时间
- Write Throughput By Instance：每个实例写入数据的吞吐量，包括apply Raft 数据日志以及Raft 快照的写入吞吐量
- Write flow：所有 TiFlash 实例磁盘写操作的流量
- Read flow：所有 TiFlash 实例磁盘读操作的流量

4.5.2.3 CDC

- CPU usage：TiCDC 节点的 CPU 使用情况
- Memory usage：TiCDC 节点的内存使用情况
- Goroutine count：TiCDC 节点Goroutine 的个数
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间单位秒计算）
- Changefeed resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间单位秒计算）
- The status of changefeeds：changefeed 的状态
  - 0：Normal
  - 1：Error
  - 2：Failed
  - 3：Stopped

  - 4：Finished
  - -1：Unknown

- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更行数
- Sorter output events/s：TiCDC 节点中 Sorter 模块每秒输出到 Mounter 模块的行数
- Mounter output events/s：TiCDC 节点中 Mounter 模块每秒输出到 Sink 模块的行数
- Table sink output events/s：TiCDC 节点中 Table Sorter 模块每秒输出到 Sink 模块的行数
- SinkV2 - Sink flush rows/s：TiCDC 节点中 Sink 模块每秒输出到下游的行数
- Transaction Sink Full Flush Duration：TiCDC 节点中 MySQL Sink 写下游事务的平均延迟和p999 延迟
- MQ Worker Send Message Duration Percentile：下游为 Kafka 时 MQ worker 发送消息的延迟
- Kafka Outgoing Bytes：MQ Workload 写下游事务的流量

4.5.3 TiDB 重要监控指标详解

使用TiUP 部署TiDB 集群时，你可以一键部署监控系统 (Prometheus & Grafana)，参考监控架构 TiDB 监控框架概述。

目前Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview、Performance_overview 等。TiDB 分为TiDB 和TiDB Summary 面板，两个面板的区别如下：

- TiDB 面板：提供尽可能全面的信息，供排查集群异常。
- TiDB Summary 面板：将TiDB 面板中用户最为关心的部分抽取出来，并做了些许修改。主要用于提供数据库日常运行中用户关心的数据，如 QPS、TPS、响应延迟等，以便作为外部展示、汇报用的监控信息。

以下为TiDB Dashboard 关键监控指标的说明：

4.5.3.1 关键指标说明

4.5.3.1.1 Query Summary

- Duration：执行时间
  - 客户端网络请求发送到TiDB，到TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行时间。
  - 由于TiDB 支持 Multi-Query，因此，客户端可以一次性发送多条 SQL 语句，如 `select 1; select 1;` <⟶ `select 1;`。此时的执行时间是所有 SQL 语句执行完之后的总时间。
- Command Per Second：TiDB 按照执行结果成功或失败来统计每秒处理的命令数。
- QPS：按 SELECT、INSERT、UPDATE 类型统计所有TiDB 实例上每秒执行的 SQL 语句数量。
- CPS By Instance：按照命令和执行结果成功或失败来统计每个TiDB 实例上的命令。
- Failed Query OPM：每个 TiDB 实例上，对每分钟执行 SQL 语句发生的错误按照错误类型进行统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码。

- Slow query：慢查询的处理时间（整个慢查询耗时、Coprocessor 耗时、Coprocessor 调度等待时间），慢查询分为 internal 和 general SQL 语句。
- Connection Idle Duration：空闲连接的持续时间。
- 999/99/95/80 Duration：不同类型的 SQL 语句执行耗时（不同百分位）。

4.5.3.1.2 Query Detail

- Duration 80/95/99/999 By Instance：每个TiDB 实例执行 SQL 语句的耗时（不同百分位）。
- Failed Query OPM Detail：每个 TiDB 实例上，对每分钟执行 SQL 语句发生的错误按照错误类型进行统计（例如语法错误、主键冲突等）。
- Internal SQL OPS：整个 TiDB 集群内部 SQL 语句执行的 QPS。内部 SQL 语句是指TiDB 内部自动执行的 SQL 语句，一般由用户 SQL 语句来触发或者内部定时任务触发。

4.5.3.1.3 Server

- Uptime：每个 TiDB 实例的运行时间。
- Memory Usage：每个 TiDB 实例的内存使用，分为进程占用内存和Golang 在堆上申请的内存。
- CPU Usage：每个 TiDB 实例的 CPU 使用。
- Connection Count：每个 TiDB 的连接数。
- Open FD Count：每个 TiDB 实例的打开的文件描述符数量。
- Disconnection Count：每个 TiDB 实例断开连接的数量。
- Event OPM：每个 TiDB 实例关键事件，例如 start，close，graceful-shutdown，kill，hang 等。
- Goroutine Count：每个 TiDB 实例的Goroutine 数量。
- Prepare Statement Count：每个 TiDB 实例现存的 `Prepare` 语句数以及总数。
- Keep Alive OPM：每个 TiDB 实例每分钟刷新监控的次数，通常不需要关注。
- Panic And Critical Error：TiDB 中出现的 Panic、Critical Error 数量。
- Time Jump Back OPS：每个 TiDB 实例上每秒操作系统时间回跳的次数。
- Get Token Duration：每个连接获取 Token 的耗时。
- Skip Binlog Count：TiDB 写入 Binlog 失败的数量。
- Client Data Traffic：TiDB 和客户端的数据流量。

4.5.3.1.4 Transaction

- Transaction OPS：每秒事务的执行数量
- Duration：事务执行时间
- Transaction Statement Num：事务中的 SQL 语句数量
- Transaction Retry Num：事务重试次数
- Session Retry Error OPS：事务重试时每秒遇到的错误数量，分为重试失败和超过最大重试次数两种类型
- Commit Token Wait Duration：事务提交时的流控队列等待时间。当出现较长等待时，代表提交事务过大，正在限流。如果系统还有资源可以使用，可以通过增大系统变量 `tidb_committer_concurrency` 的值来加速提交
- KV Transaction OPS：每个 TiDB 内部每秒执行的事务数量
  - 一个用户的事务，在TiDB 内部可能会触发多次事务执行，其中包含，内部元数据的读取，用户事务原子性地多次重试执行等

  - TiDB 内部的定时任务也会通过事务来操作数据库，这部分也包含在这个面板里

- KV Transaction Duration：每个 TiDB 内部执行事务的耗时
- Transaction Regions Num：事务操作的 Region 数量
- Transaction Write KV Num Rate and Sum：事务写入 KV 的速率总和
- Transaction Write KV Num：事务操作的 KV 数量
- Statement Lock Keys：单个语句的加锁个数
- Send HeartBeat Duration：事务发送心跳的时间间隔
- Transaction Write Size Bytes Rate and sum：事务写入字节数的速率总和
- Transaction Write Size Bytes：事务写入的数据大小
- Acquire Pessimistic Locks Duration：加锁所消耗的时间
- TTL Lifetime Reach Counter：事务的 TTL 寿命上限。TTL 上限默认值 1 小时，它的含义是从悲观事务第一次加锁，或者乐观事务的第一个prewrite 开始，超过了 1 小时。可以通过修改TiDB 配置文件中 `max-txn-ttl` 来改变 TTL 寿命上限
- Load Safepoint OPS：加载 Safepoint 的次数。Safepoint 作用是在事务读数据时，保证不读到 Safepoint 之前的数据，保证数据安全。因为，Safepoint 之前的数据有可能被GC 清理掉
- Pessimistic Statement Retry OPS：悲观语句重试次数。当语句尝试加锁时，可能遇到写入冲突，此时，语句会重新获取新的 snapshot 并再次加锁
- Transaction Types Per Seconds：每秒采用两阶段提交 (2PC)、异步提交（Async Commit) 和一阶段提交(1PC) 机制的事务数量，提供成功和失败两种数量

4.5.3.1.5 Executor

- Parse Duration：SQL 语句解析耗时统计。
- Compile Duration：将解析后的 SQL AST 编译成执行计划的耗时。
- Execution Duration：执行 SQL 语句执行计划耗时。
- Expensive Executor OPS：每秒消耗系统资源比较多的算子。包括 Merge Join、 Hash Join、 Index Look Up Join、 Hash Agg、 Stream Agg、 Sort、 TopN 等。
- Queries Using Plan Cache OPS：每秒使用 Plan Cache 的查询数量。
- Plan Cache Miss OPS：每秒出现 Plan Cache Miss 的数量。
- Plan Cache Memory Usage：每个 TiDB 实例上所有 Plan Cache 缓存的执行计划占用的总内存。
- Plan Cache Plan Num：每个 TiDB 实例上所有 Plan Cache 缓存的执行计划总数。

4.5.3.1.6 Distsql

- Distsql Duration：Distsql 处理的时长
- Distsql QPS：每秒 Distsql 的数量
- 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

4.5.3.1.7 KV Errors

- KV Backoff Duration：KV 每个请求重试的总时间。TiDB 向 TiKV 的请求都有重试机制，这里统计的是向TiKV 发送请求时遇到错误重试的总时间
- TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
- KV Backoff OPS：TiKV 返回错误信息的数量
- Lock Resolve OPS：TiDB 清理锁操作的数量。当TiDB 的读写请求遇到锁时，会尝试进行锁清理
- Other Errors OPS：其他类型的错误数量，包括清锁和更新 SafePoint

4.5.3.1.8 KV Request

- KV Request OPS：KV Request 根据TiKV 显示执行次数
- KV Request Duration 99 by store：根据 TiKV 显示 KV Request 执行时间
- KV Request Duration 99 by type：根据类型显示 KV Request 的执行时间

4.5.3.1.9 PD Client

- PD Client CMD OPS：PD Client 每秒执行命令的数量
- PD Client CMD Duration：PD Client 执行命令耗时
- PD Client CMD Fail OPS：PD Client 每秒执行命令失败的数量
- PD TSO OPS：TiDB 每秒从 PD 获取TSO 的数量
- PD TSO Wait Duration：TiDB 等待从 PD 返回TSO 的时间
- PD TSO RPC Duration：TiDB 从向 PD 发送获取TSO 的请求到接收到TSO 的耗时
- Start TSO Wait Duration：TiDB 从向 PD 发送获取 start tso 请求开始到开始等待tso 返回的时间

4.5.3.1.10 Schema Load

- Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
- Load Schema OPS：TiDB 从 TiKV 每秒获取 Schema 的数量
- Schema Lease Error OPM：Schema Lease 出错统计，包括change 和 outdate 两种，change 代表 schema 发生了变化，outdate 代表无法更新 schema，属于较严重错误，出现 outdate 错误时会报警
- Load Privilege OPS：TiDB 从 TiKV 每秒获取权限信息的数量

4.5.3.1.11 DDL

- DDL Duration 95：DDL 语句处理时间的 95% 分位
- Batch Add Index Duration 100：创建索引时每个 Batch 所花费的最大时间
- DDL Waiting Jobs Count：等待的 DDL 任务数量
- DDL META OPM：DDL 每分钟获取 META 的次数
- DDL Worker Duration 99：每个 DDL worker 执行时间的 99% 分位
- Deploy Syncer Duration：Schema Version Syncer 初始化，重启，清空等操作耗时
- Owner Handle Syncer Duration：DDL Owner 在执行更新，获取以及检查 Schema Version 的耗时
- Update Self Version Duration：Schema Version Syncer 更新版本信息耗时
- DDL OPM：DDL 语句的每秒执行次数
- DDL backfill progress in percentage：backfill DDL 任务的进度展示

4.5.3.1.12 Statistics

- Auto Analyze Duration 95：自动ANALYZE 耗时
- Auto Analyze QPS：自动 ANALYZE 数量
- Stats Inaccuracy Rate：统计信息不准确度
- Pseudo Estimation OPS：使用假的统计信息优化 SQL 的数量
- Dump Feedback OPS：存储统计信息 Feedback 的数量
- Store Query Feedback QPS：存储合并查询的 Feedback 信息的每秒操作数量，该操作在TiDB 内存中进行
- Significant Feedback：重要的 Feedback 更新统计信息的数量
- Update Stats OPS：利用 Feedback 更新统计信息的数量
- Fast Analyze Status 100：快速收集统计信息的状态

4.5.3.1.13 Owner

- New ETCD Session Duration 95：创建一个新的 etcd 会话花费的时间。TiDB 通过 etcd client 连接 PD 中的 etcd 保存/读取部分元数据信息。这里记录了创建会话花费的时间
- Owner Watcher OPS：DDL owner watch PD 的 etcd 的元数据的 goroutine 的每秒操作次数

4.5.3.1.14 Meta

- AutoID QPS：AutoID 相关操作的数量统计，包括全局 ID 分配、单个 Table AutoID 分配、单个 Table AutoID Rebase 三种操作
- AutoID Duration：AutoID 相关操作的耗时
- Region Cache Error OPS：TiDB 缓存的region 信息每秒遇到的错误次数
- Meta Operations Duration 99：元数据操作延迟

4.5.3.1.15 GC

- Worker Action OPM：GC 相关操作的数量，包括run_job，resolve_lock，delete_range 等操作
- Duration 99：GC 相关操作的耗时
- Config：GC 的数据保存时长(life time) 和 GC 运行间隔(run interval) 配置
- GC Failure OPM：GC 相关操作失败的数量
- Delete Range Failure OPM：Delete range 失败的次数
- Too Many Locks Error OPM：GC 清锁过多错误的数量
- Action Result OPM：GC 相关操作结果数量
- Delete Range Task Status：Delete range 的任务状态，包含完成和失败状态
- Push Task Duration 95：将GC 子任务推送给GC worker 的耗时

4.5.3.1.16 Batch Client

- Pending Request Count by TiKV：TiKV 批量消息处理的等待数量
- Wait Duration 95: 批量消息处理的等待时间。
- Batch Client Unavailable Duration 95：批处理客户端的不可用时长。
- No Available Connection Counter：批处理客户端不可用的连接数。

4.5.3.1.17 TTL

- TiDB CPU Usage: 每个 TiDB 实例的 CPU 使用。
- TiKV IO MBps: 每个 TiKV 实例的 I/O 吞吐量。
- TiKV CPU: 每个 TiKV 实例的 CPU 使用。
- TTL QPS By Type：TTL 任务产生的不同类型语句的 QPS 信息。
- TTL Insert Rows Per Second: 每秒钟向 TTL 表插入的数据行数。
- TTL Processed Rows Per Second：TTL 任务每秒处理的过期数据的行数。
- TTL Insert Rows Per Hour: 每小时总共向 TTL 表插入的行数。
- TTL Delete Rows Per Hour: 每小时总共删除的过期行数。
- TTL Scan/Delete Query Duration：TTL 的扫描/删除语句的执行时间。
- TTL Scan/Delete Worker Time By Phase：TTL 内部工作线程的不同阶段所占用的时间。
- TTL Job Count By Status：当前正在执行的 TTL 任务的数量。
- TTL Task Count By Status：当前正在执行的 TTL 子任务的数量。

4.5.4 PD 重要监控指标详解

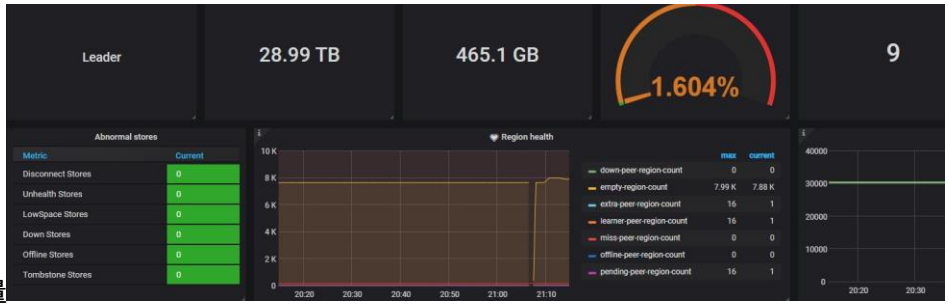
使用 TiUP 部署 TiDB 集群时，一键部署监控系统(Prometheus & Grafana)，监控架构参见 TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview、Performance_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。

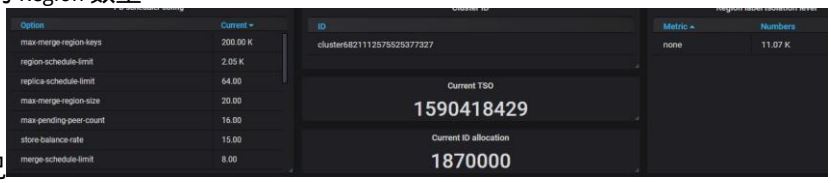
以下为 PD Dashboard 监控说明：

- 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



4.5.4.1 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

4.5.4.2 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 53: PD Dashboard - Operator metrics

4.5.4.3 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 54: PD Dashboard - Balance metrics

4.5.4.4 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 55: PD Dashboard - Hot write metrics

4.5.4.5 Statistics - hot read

- Hot Region’s peer distribution：每个TiKV 实例上成为读取热点的peer 的数量
- Total read bytes on hot peer Regions：每个TiKV 实例上所有成为读取热点的peer 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 56: PD Dashboard - Hot read metrics

4.5.4.6 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

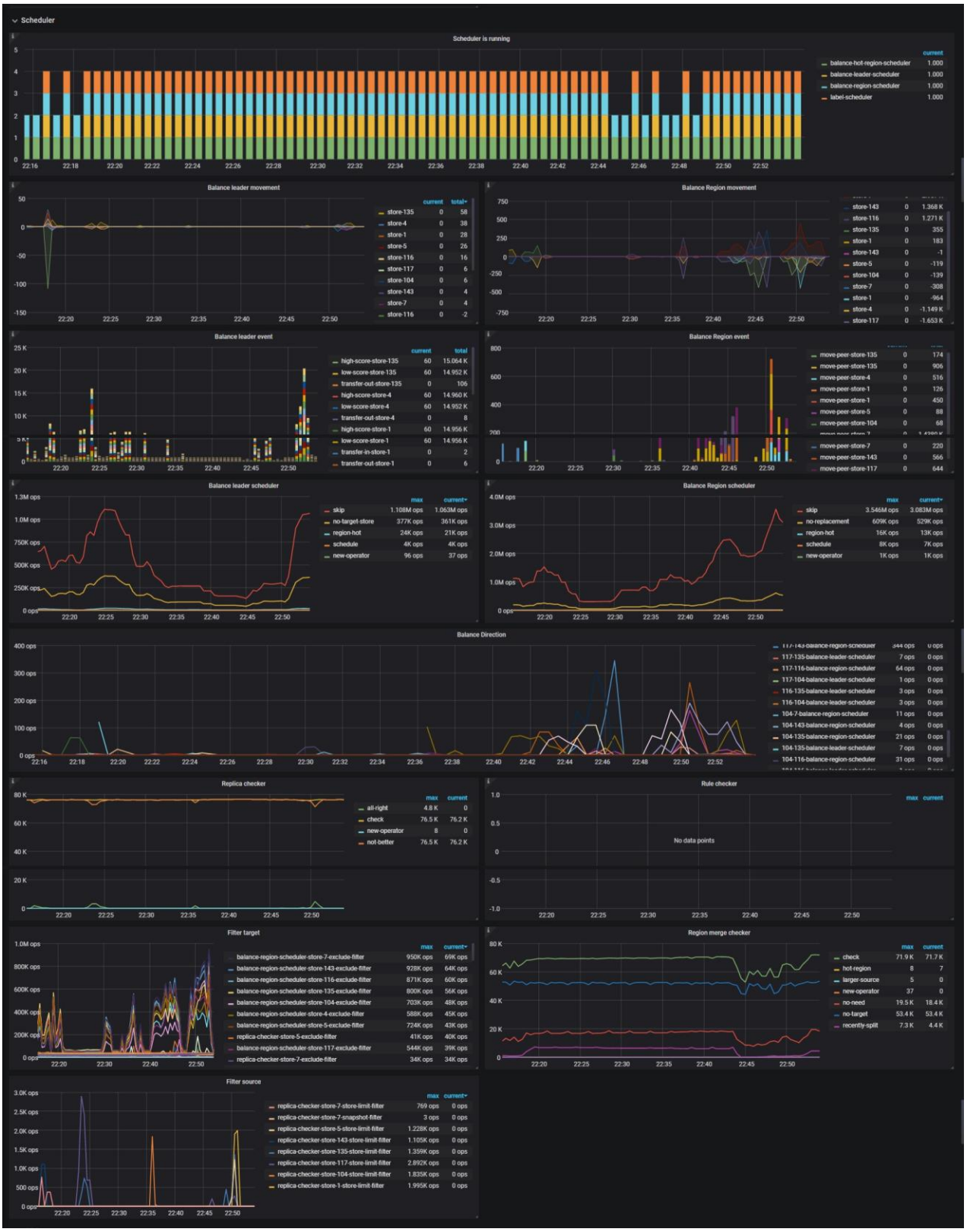


图 57: PD Dashboard - Scheduler metrics

4.5.4.7 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 58: PD Dashboard - gRPC metrics

4.5.4.8 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 59: PD Dashboard - etcd metrics

4.5.4.9 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

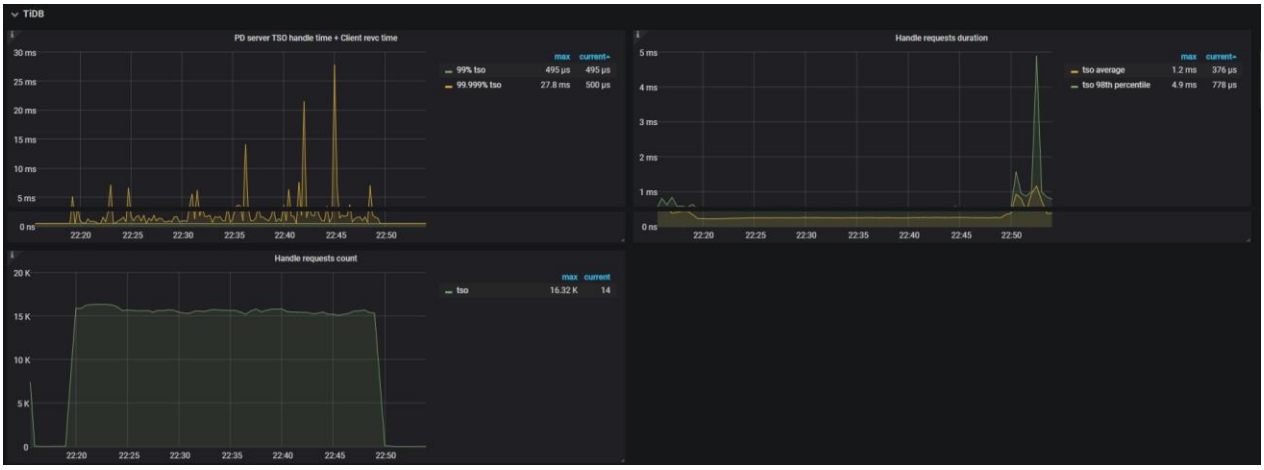


图 60: PD Dashboard - TiDB metrics

4.5.4.10 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 61: PD Dashboard - Heartbeat metrics

4.5.4.11 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 62: PD Dashboard - Region storage

4.5.5 TiKV 监控指标详解

使用TiUP 部署TiDB 集群时，一键部署监控系统(Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

4.5.5.1 TiKV-Details 面板

对于日常运维，通过观察TiKV-Details 面板上的指标，可以了解TiKV 当前的状态。根据性能地图可以检查集群的状态是否符合预期。

以下为TiKV-Details 默认的监控信息：

4.5.5.1.1 Cluster

- Store size：每个 TiKV 实例的使用的存储空间的大小
- Available size：每个 TiKV 实例的可用的存储空间的大小
- Capacity size：每个 TiKV 实例的存储容量的大小
- CPU：每个 TiKV 实例 CPU 的使用率
- Memory：每个 TiKV 实例内存的使用情况
- IO utilization：每个TiKV 实例IO 的使用率
- MBps：每个 TiKV 实例写入和读取的数据量大小
- QPS：每个 TiKV 实例上各种命令的 QPS
- Errps：每个TiKV 实例上 gRPC 消息失败的速率
- leader：每个TiKV 实例leader 的个数
- Region：每个TiKV 实例Region 的个数
- Uptime：自上次重启以来 TiKV 正常运行的时间

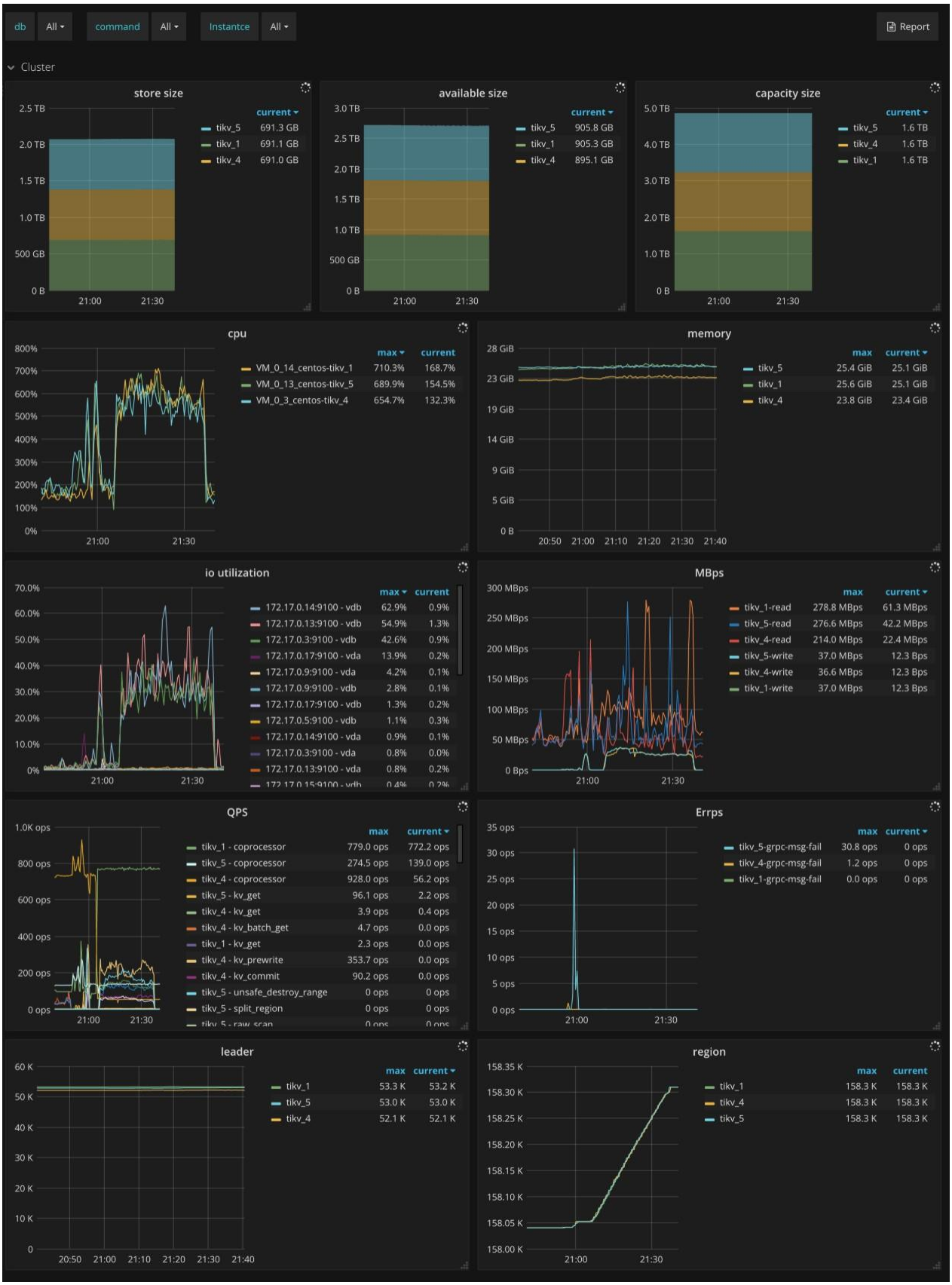


图 63: TiKV Dashboard - Cluster metrics

4.5.5.1.2 Errors

- Critical error：严重错误的数量
- Server is busy：各种会导致TiKV 实例暂时不可用的事件个数，如 write stall，channel full 等，正常情况下应当为 0
- Server report failures：server 报错的消息个数，正常情况下应当为 0
- Raftstore error：每个 TiKV 实例上raftstore 发生错误的个数
- Scheduler error：每个 TiKV 实例上 scheduler 发生错误的个数
- Coprocessor error：每个 TiKV 实例上coprocessor 发生错误的个数
- gRPC message error：每个 TiKV 实例上 gRPC 消息发生错误的个数
- Leader drop：每个TiKV 实例上drop leader 的个数
- Leader missing：每个TiKV 实例上missing leader 的个数
- Log Replication Rejected：每个TiKV 实例上由于内存不足而拒绝logappend 消息的个数



图 64: TiKV Dashboard - Errors metrics

4.5.5.1.3 Server

- CF size：每个列族的大小
- Store size：每个 TiKV 实例的使用的存储空间的大小
- Channel full：每个 TiKV 实例上channel full 错误的数量，正常情况下应当为 0
- Active written leaders：各个TiKV 实例中正在被写入的 Leader 的数量
- Approximate Region size：每个 Region 近似的大小
- Approximate Region size Histogram：每个 Region 近似大小的直方图
- Region average written keys：每个 TiKV 实例上所有Region 的平均 key 写入个数
- Region average written bytes：每个TiKV 实例上所有Region 的平均写入大小

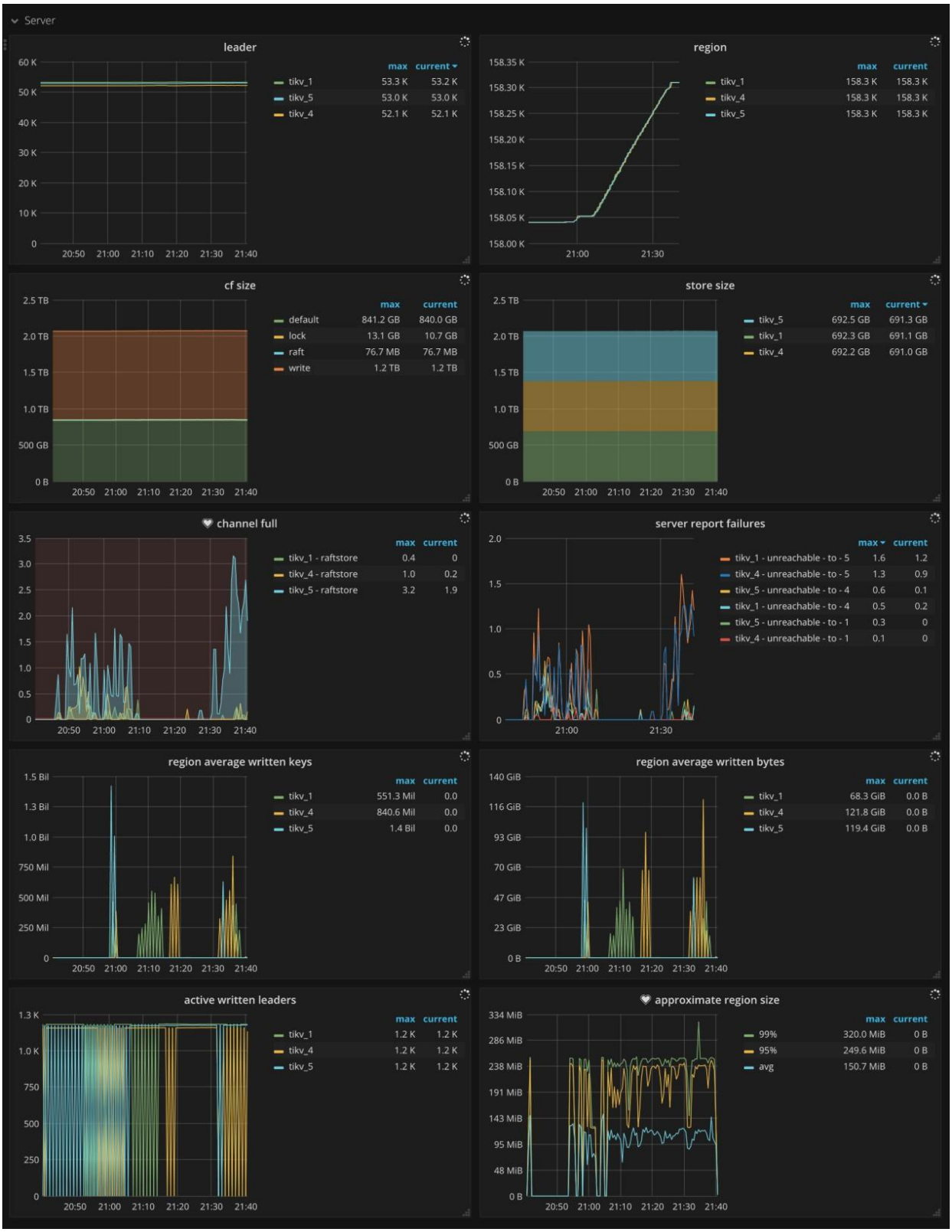


图 65: TiKV Dashboard - Server metrics

4.5.5.1.4 gRPC

- gRPC message count：每种 gRPC 请求的速度
- gRPC message failed：失败的 gRPC 请求的速度
- 99% gRPC message duration：99% gRPC 请求的执行时间小于该值
- Average gRPC message duration：gRPC 请求平均的执行时间
- gRPC batch size：TiDB 与 TiKV 之间 grpc 请求的batch 大小
- raft message batch size：TiKV 与 TiKV 之间raft 消息的batch 大小
- gRPC request sources QPS: 不同 gRPC 请求来源的速度
- gRPC request sources duration: 不同 gRPC 请求来源的执行总时间
- gRPC resource group QPS: 不同resource group 的 gRPC 请求速度

4.5.5.1.5 Thread CPU

- Raft store CPU：raftstore 线程的 CPU 使用率，通常应低于 80% * `raftstore.store-pool-size`
- Async apply CPU：async apply 线程的 CPU 使用率，通常应低于 90% * `raftstore.apply-pool-size`
- Scheduler worker CPU：scheduler worker 线程的 CPU 使用率，通常应低于 90% * `storage.scheduler-worker-pool-size`
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80% * `server.grpc-concurrency`
- Unified read pool CPU：unified read pool 线程的 CPU 使用率
- Storage ReadPool CPU：storage read pool 线程的 CPU 使用率
- Coprocessor CPU：coprocessor 线程的 CPU 使用率
- RocksDB CPU：RocksDB 线程的 CPU 使用率
- GC worker CPU：GC worker 线程的 CPU 使用率
- BackGround worker CPU：background worker 线程的 CPU 使用率

4.5.5.1.6 PD

- PD requests：TiKV 发送给 PD 的请求速度
- PD request duration (average)：TiKV 发送给 PD 的请求处理的平均时间
- PD heartbeats：发送给 PD 的心跳的速度
- PD validate peers：TiKV 发送给 PD 用于验证TiKV 的 peer 有效的消息的速度

4.5.5.1.7 Raft IO

- Apply log duration：Raft apply 日志所花费的时间
- Apply log duration per server：每个 TiKV 实例上Raft apply 日志所花费的时间
- Append log duration：Raft append 日志所花费的时间
- Append log duration per server：每个 TiKV 实例上Raft append 日志所花费的时间
- Commit log duration：Raft commit 日志所花费的时间
- Commit log duration per server：每个 TiKV 实例上Raft commit 日志所花费的时间



图 66: TiKV Dashboard - Raft IO metrics

4.5.5.1.8 Raft process

- Ready handled：Raft 中不同ready 类型的 ops
  - count: 批量处理ready 的 ops
  - has_ready_region: 获得ready 的 Region 的 ops
  - pending_region: 被检查是否获得ready 的 Region 的 ops，v3.0.0 后废弃
  - message: ready 内待发送message 的 ops
  - append: ready 内 Raft log entry 的 ops
  - commit: ready 内 committed Raft log entry 的 ops
  - snapshot: 携带 snapshot 的 ready 的 ops
- 0.99 Duration of Raft store events：99% 的 raftstore 事件所花费的时间
- Process ready duration：处理ready 所花费的时间
- Process ready duration per server：每个 TiKV 实例处理ready 所花费的时间，99.99% 的情况下，应该小于 2s

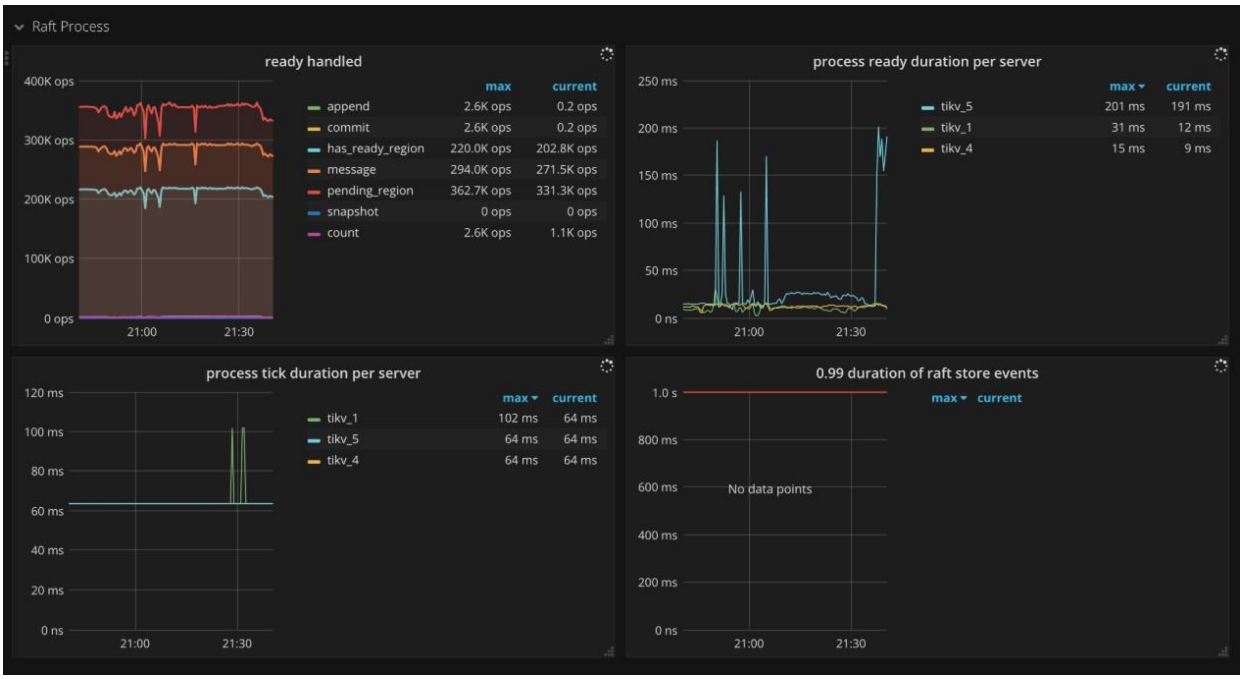


图 67: TiKV Dashboard - Raft process metrics

4.5.5.1.9 Raft message

- Sent messages per server：每个 TiKV 实例发送Raft 消息的 ops
- Flush messages per server：每个 TiKV 实例中raft client 往外 flush Raft 消息的 ops
- Receive messages per server：每个 TiKV 实例接受Raft 消息的 ops
- Messages：发送不同类型的 Raft 消息的 ops
- Vote：Raft 投票消息发送的 ops
- Raft dropped messages：每秒钟丢弃不同类型的Raft 消息的个数

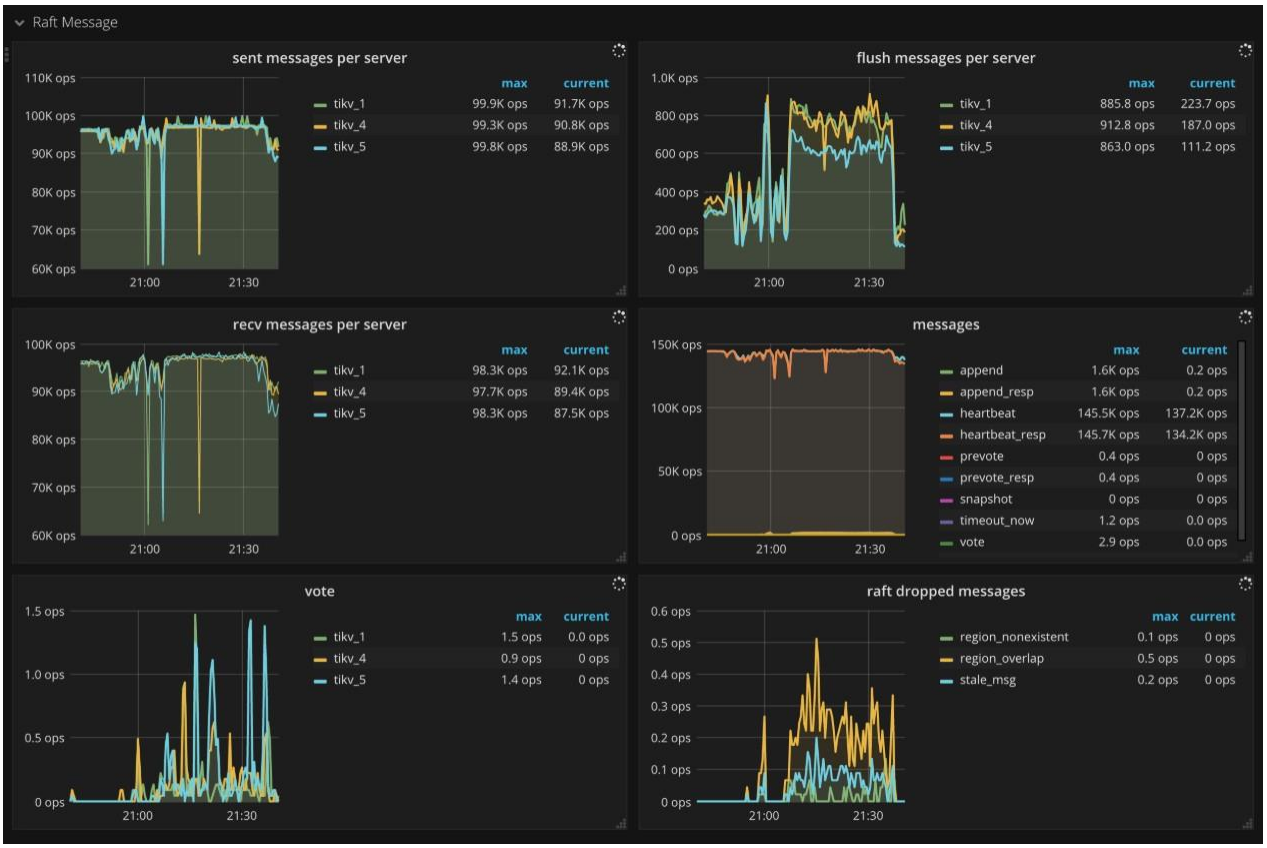


图 68: TiKV Dashboard - Raft message metrics

4.5.5.1.10 Raft propose

- Raft apply proposals per ready：在一个batch 内，apply proposal 时每个ready 中包含proposal 的个数的直方图
- Raft read/write proposals：不同类型的proposal 的 ops
- Raft read proposals per server：每个 TiKV 实例发起读proposal 的 ops
- Raft write proposals per server：每个 TiKV 实例发起写proposal 的 ops
- Propose wait duration：proposal 的等待时间的直方图
- Propose wait duration per server：每个 TiKV 实例上每个proposal 的等待时间的直方图
- Apply wait duration：apply 的等待时间的直方图
- Apply wait duration per server：每个 TiKV 实例上每个apply 的等待时间的直方图
- Raft log speed：peer propose 日志的平均速度

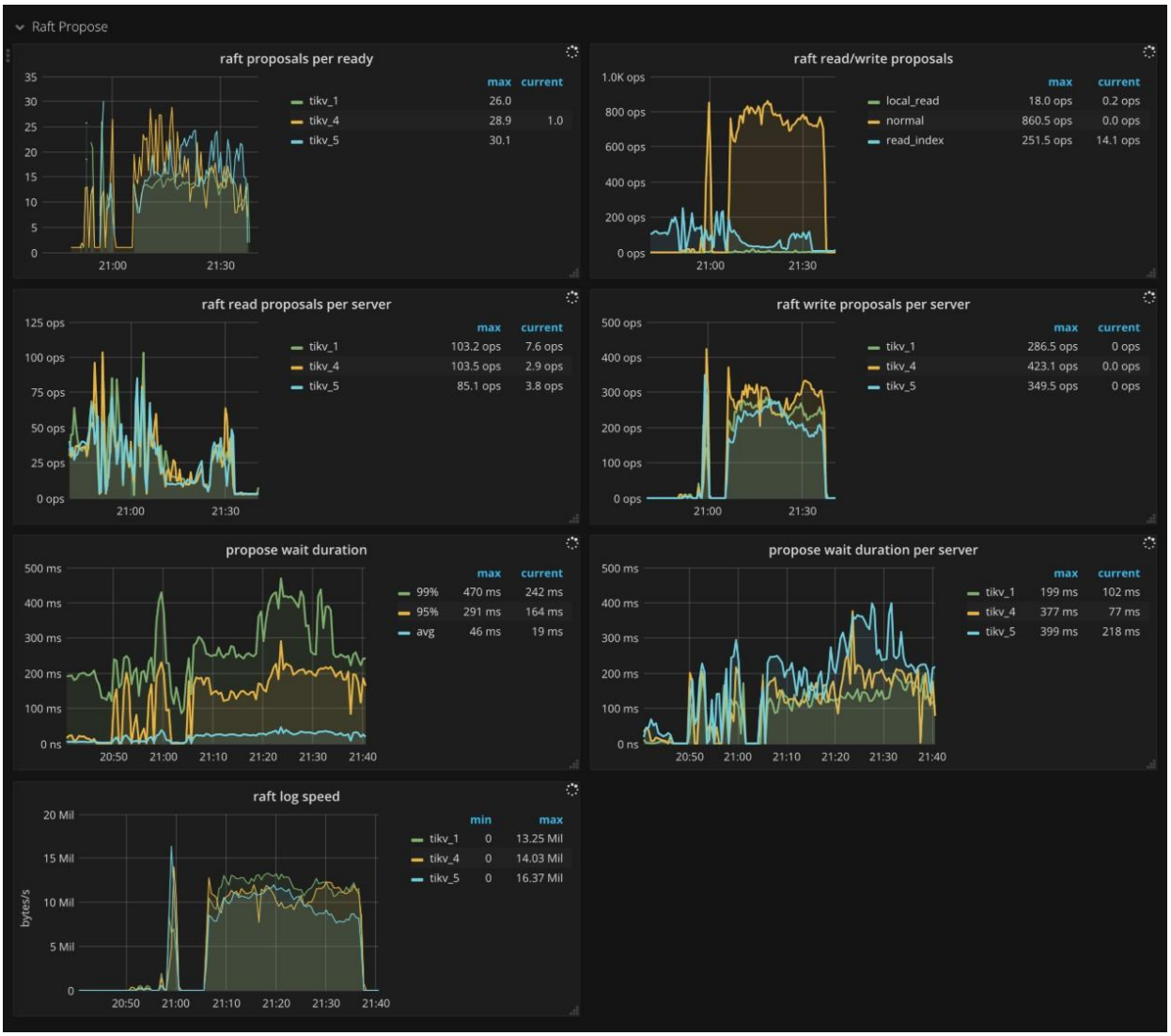


图 69: TiKV Dashboard - Raft propose metrics

4.5.5.1.11 Raft admin

- Admin proposals：admin proposal 的 ops
- Admin apply：apply 命令的 ops
- Check split：split check 命令的 ops
- 99.99% Check split duration：99.99% 的情况下，split check 所需花费的时间

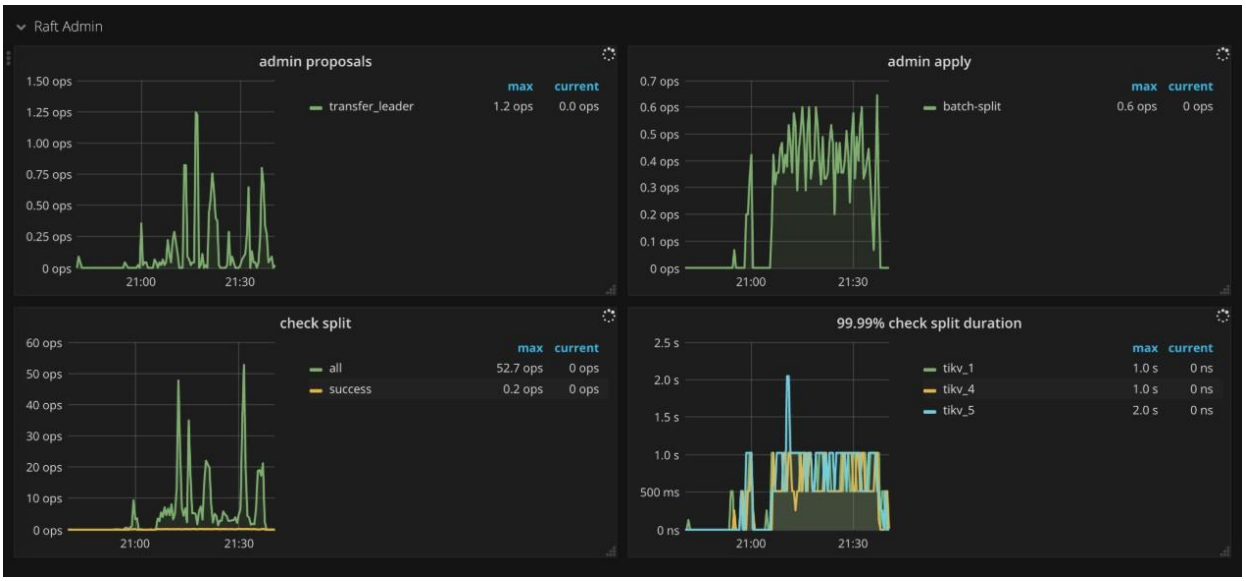


图 70: TiKV Dashboard - Raft admin metrics

4.5.5.1.12 Local reader

- Local reader requests：所有请求的总数以及local read 线程拒绝的请求数量

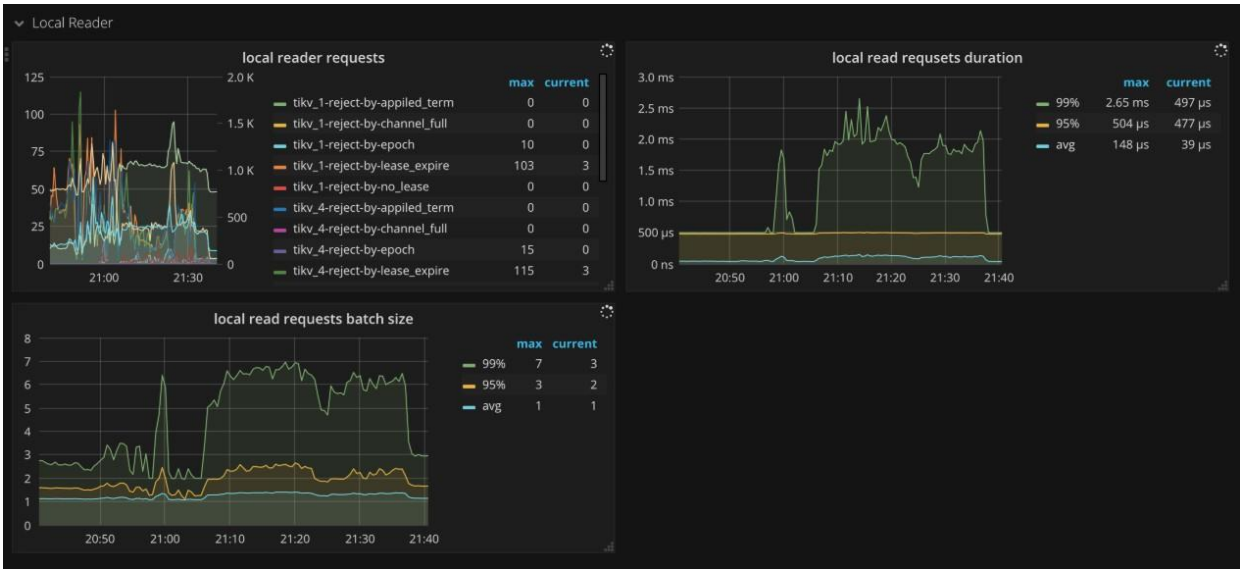


图 71: TiKV Dashboard - Local reader metrics

4.5.5.1.13 Unified Read Pool

- Time used by level： 在 unified read pool 中每个级别使用的时间，级别 0 指小查询

- Level 0 chance：在 unified read pool 中调度的level 0 任务的比例
- Running tasks：在 unified read pool 中并发运行的任务数量

4.5.5.1.14 Storage

- Storage command total：收到不同命令的 ops
- Storage async request error：异步请求出错的 ops
- Storage async snapshot duration：异步处理 snapshot 所花费的时间，99% 的情况下，应该小于 1s
- Storage async write duration：异步写所花费的时间，99% 的情况下，应该小于 1s



图 72: TiKV Dashboard - Storage metrics

4.5.5.1.15 Scheduler

- Scheduler stage total：每种命令不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler writing bytes：每个TiKV 实例正在处理的命令的写入字节数量
- Scheduler priority commands：不同优先级命令的 ops

- Scheduler pending commands：每个TiKV 实例上pending 命令的 ops



图 73: TiKV Dashboard - Scheduler metrics

4.5.5.1.16 Scheduler - commit

- Scheduler stage total：commit 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行commit 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：commit 命令读取 key 的个数
- Scheduler keys written：commit 命令写入 key 的个数
- Scheduler scan details：执行commit 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 commit 命令时，扫描每个lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 commit 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 commit 命令时，扫描每个default CF 中 key 的详细情况



图 74: TiKV Dashboard - Scheduler commit metrics

4.5.5.1.17 Scheduler - pessimistic_rollback

- Scheduler stage total：pessimistic_rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行pessimistic_rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：pessimistic_rollback 命令读取 key 的个数
- Scheduler keys written：pessimistic_rollback 命令写入 key 的个数

- Scheduler scan details：执行pessimistic_rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 pessimistic_rollback 命令时，扫描每个lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 pessimistic_rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 pessimistic_rollback 命令时，扫描每个default CF 中 key 的详细情况

4.5.5.1.18 Scheduler - prewrite

- Scheduler stage total：prewrite 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行prewrite 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：prewrite 命令读取 key 的个数
- Scheduler keys written：prewrite 命令写入 key 的个数
- Scheduler scan details：执行prewrite 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 prewrite 命令时，扫描每个lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 prewrite 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 prewrite 命令时，扫描每个default CF 中 key 的详细情况

4.5.5.1.19 Scheduler - rollback

- Scheduler stage total：rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：rollback 命令读取 key 的个数
- Scheduler keys written：rollback 命令写入 key 的个数
- Scheduler scan details：执行rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 rollback 命令时，扫描每个lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 rollback 命令时，扫描每个default CF 中 key 的详细情况

4.5.5.1.20 GC

- 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
- TiDB GC seconds：TiDB 执行GC 花费的时间
- TiDB GC worker actions：TiDB GC worker 的不同action 的个数
- TiKV AutoGC Working：Auto GC 管理器的工作状态
- ResolveLocks Progress：GC 第一阶段(ResolveLocks) 的进度
- TiKV Auto GC Progress：GC 第二阶段的进度
- GC speed：GC 每秒删除的 key 的数量
- TiKV Auto GC SafePoint：TiKV GC 的 safe point 的数值，safe point 为当前GC 的时间戳
- GC lifetime：TiDB 设置的GC lifetime
- GC interval：TiDB 设置的GC 间隔
- GC in Compaction Filter：write CF 的 Compaction Filter 中已过滤版本的数量

4.5.5.1.21 Snapshot

- Rate snapshot message：发送Raft snapshot 消息的速率
- 99% Handle snapshot duration：99% 的情况下，处理 snapshot 所需花费的时间
- Snapshot state count：不同状态的 snapshot 的个数
- 99.99% Snapshot size：99.99% 的 snapshot 的大小
- 99.99% Snapshot KV count：99.99% 的 snapshot 包含的 key 的个数

4.5.5.1.22 Task

- Worker handled tasks：worker 每秒钟处理的任务的数量
- Worker pending tasks：当前 worker 中，每秒钟 pending 和running 的任务的数量，正常情况下，应该小于 1000
- FuturePool handled tasks：future pool 每秒钟处理的任务的数量
- FuturePool pending tasks：当前 future pool 中，每秒钟pending 和 running 的任务的数量

4.5.5.1.23 Coprocessor Overview

- Request duration：从收到coprocessor 请求到处理结束所消耗的总时间
- Total Requests：每种类型的总请求的 ops
- Handle duration：每分钟实际处理coprocessor 请求所消耗的时间的直方图
- Total Request Errors：Coprocessor 每秒请求错误的数量，正常情况下，短时间内不应该有大量的错误
- Total KV Cursor Operations： 各种类型的 KV cursor 操作的总数量的 ops， 例如 select、 index、 analyze_table、 analyze_index、 checksum_table、 checksum_index 等
- KV Cursor Operations：每秒各种类型的 KV cursor 操作的数量，以直方图形式显示
- Total RocksDB Perf Statistics：RocksDB 性能统计数据
- Total Response Size：coprocessor 回应的数据大小

4.5.5.1.24 Coprocessor Detail

- Handle duration：每秒钟实际处理coprocessor 请求所消耗的时间的直方图
- 95% Handle duration by store：每秒钟中 95% 的情况下，每个TiKV 实例处理coprocessor 请求所花费的时间
- Wait duration：coprocessor 每秒钟内请求的等待时间，99.99% 的情况下，应该小于 10s
- 95% Wait duration by store：每秒钟 95% 的情况下，每个TiKV 实例上coprocessor 请求的等待时间
- Total DAG Requests：DAG 请求的总数量的 ops
- Total DAG Executors：DAG executor 的总数量的 ops
- Total Ops Details (Table Scan)：coprocessor 中请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details (Index Scan)：coprocessor 中请求为index 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Table Scan)：coprocessor 中对于每个 CF 请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Index Scan)：coprocessor 中对于每个 CF 请求为index 的 scan 过程中每秒钟各种事件发生的次数

4.5.5.1.25 Threads

- Threads state：TiKV 线程的状态
- Threads IO：TiKV 各个线程的I/O 流量
- Thread Voluntary Context Switches：TiKV 线程自主切换的次数
- Thread Nonvoluntary Context Switches：TiKV 线程被动切换的次数

4.5.5.1.26 RocksDB - kv/raft

- Get operations：get 操作的 ops
- Get duration：get 操作的耗时
- Seek operations：seek 操作的 ops
- Seek duration：seek 操作的耗时
- Write operations：write 操作的 ops
- Write duration：write 操作的耗时
- WAL sync operations：sync WAL 操作的 ops
- Write WAL duration：write 操作中写WAL 的耗时
- WAL sync duration：sync WAL 操作的耗时
- Compaction operations：compaction 和 flush 操作的 ops
- Compaction duration：compaction 和 flush 操作的耗时
- SST read duration：读取 SST 所需的时间
- Write stall duration：由于 write stall 造成的时间开销，正常情况下应为 0
- Memtable size：每个 CF 的 memtable 的大小
- Memtable hit：memtable 的命中率
- Block cache size：block cache 的大小。如果将 `shared block cache` 禁用，即为每个 CF 的 block cache 的大小
- Block cache hit：block cache 的命中率
- Block cache flow：不同 block cache 操作的流量
- Block cache operations 不同block cache 操作的个数
- Keys flow：不同操作造成的 key 的流量
- Total keys：每个 CF 中 key 的个数
- Read flow：不同读操作的流量
- Bytes/Read：每次读的大小
- Write flow：不同写操作的流量
- Bytes/Write：每次写的大小
- Compaction flow：compaction 相关的流量
- Compaction pending bytes：等待compaction 的大小
- Read amplification：每个TiKV 实例的读放大
- Compression ratio：每一层的压缩比
- Number of snapshots：每个 TiKV 的 snapshot 的数量
- Oldest snapshots duration：最旧的 snapshot 保留的时间
- Number files at each level：每一层的文件个数
- Ingest SST duration seconds：ingest SST 所花费的时间
- Stall conditions changed of each CF：每个 CF stall 的原因

4.5.5.1.27 Titan - All

- Blob file count：Titan blob 文件的数量
- Blob file size：Titan blob 文件总大小
- Live blob size：有效 blob record 的总大小
- Blob cache hit：Titan 的 blob cache 命中率
- Iter touched blob file count：单个Iterator 所涉及到blob 文件的数量
- Blob file discardable ratio distribution：blob 文件的失效blob record 比例的分布情况
- Blob key size：Titan 中 blob key 的大小
- Blob value size：Titan 中 blob value 的大小
- Blob get operations：blob 的 get 操作的数量
- Blob get duration：blob 的 get 操作的耗时
- Blob iter operations：blob 的 iter 操作的耗时
- Blob seek duration：blob 的 seek 操作的耗时
- Blob next duration：blob 的 next 操作的耗时
- Blob prev duration：blob 的 prev 操作的耗时
- Blob keys flow：Titan blob 读写的 key 数量
- Blob bytes flow：Titan blob 读写的bytes 数量
- Blob file read duration：blob 文件的读取耗时
- Blob file write duration：blob 文件的写入耗时
- Blob file sync operations：blob 文件 sync 次数
- Blob file sync duration：blob 文件 sync 耗时
- Blob GC action：Titan GC 细分动作的次数
- Blob GC duration：Titan GC 的耗时
- Blob GC keys flow：Titan GC 读写的 key 数量
- Blob GC bytes flow：Titan GC 读写的bytes 数量
- Blob GC input file size：Titan GC 输入文件的大小
- Blob GC output file size：Titan GC 输出文件的大小
- Blob GC file count：Titan GC 涉及的blob 文件数量

4.5.5.1.28 Pessimistic Locking

- Lock Manager Thread CPU：lock manager 的线程 CPU 使用率
- Lock Manager Handled tasks：lock manager 处理的任务数量
- Waiter lifetime duration：事务等待锁释放的时间
- Wait table：wait table 的状态信息，包括锁的数量和等锁事务的数量
- Deadlock detect duration：处理死锁检测请求的耗时
- Detect error：死锁检测遇到的错误数量，包含死锁的数量
- Deadlock detector leader：死锁检测器leader 所在节点的信息
- Total pessimistic locks memory size：内存悲观锁占用内存的总大小
- In-memory pessimistic locking result：将悲观锁仅保存到内存的结果，其中 full 表示因为超过内存限制而无法将悲观锁保存至内存的次数

4.5.5.1.29 Memory

- Allocator Stats：内存分配器的统计信息

4.5.5.1.30 Backup

- Backup CPU：backup 的线程 CPU 使用率
- Range Size：backup range 的大小直方图
- Backup Duration：backup 的耗时
- Backup Flow：backup 总的字节大小
- Disk Throughput：实例磁盘的吞吐量
- Backup Range Duration：backup range 的耗时
- Backup Errors：backup 中发生的错误数量

4.5.5.1.31 Encryption

- Encryption data keys：正在使用的加密 data key 的总数量
- Encrypted files：被加密的文件数量
- Encryption initialized：显示加密是否被启用，1 代表已经启用
- Encryption meta files size：加密相关的元数据文件的大小
- Encrypt/decrypt data nanos：每次加密/解密数据的耗时的直方图
- Read/write encryption meta duration：每秒钟读写加密文件所耗费的时间

4.5.5.1.32 面板常见参数的解释

gRPC 消息类型

1. 使用事务型接口的命令：
    - kv_get：事务型的 get 命令，获取指定ts 能读到的最新版本数据
    - kv_scan：扫描连续的一段数据
    - kv_prewrite：2PC 的第一阶段，预写入事务要提交的数据
    - kv_pessimistic_lock：对 key 加悲观锁，防止其他事务修改
    - kv_pessimistic_rollback：删除 key 上的悲观锁
    - kv_txn_heart_beat：更新悲观事务或大事务的 `lock_ttl` 以防止其被回滚
    - kv_check_txn_status：检查事务的状态
    - kv_commit：2PC 的第二阶段，提交prewrite 阶段写入的数据
    - kv_cleanup：回滚一个事务（此命令将会在 4.0 中废除）
    - kv_batch_get：与 `kv_get` 类似，一次性获取批量 key 的 value
    - kv_batch_rollback：批量回滚多个预写的事务
    - kv_scan_lock：扫描所有版本号在 `max_version` 之前的锁，用于清理过期的事务
    - kv_resolve_lock：根据事务状态，提交或回滚事务的锁
    - kv_gc：触发垃圾回收
    - kv_delete_range：从 TiKV 中删除连续的一段数据
2. 非事务型的裸命令：
    - raw_get：获取 key 所对应的 value
    - raw_batch_get：获取一批 key 所对应的 value
    - raw_scan：扫描一段连续的数据

- raw_batch_scan：扫描多段连续的数据
- raw_put：写入一个 key/value 对
- raw_batch_put：直接写入一批 key/value 对
- raw_delete：删除一个 key/value 对
- raw_batch_delete：删除一批 key/value 对
- raw_delete_range：删除连续的一段区间

4.5.5.2 TiKV-FastTune 面板

当TiKV 出现 QPS 抖动、延迟抖动、延迟增加趋势等性能问题时，你可以查看TiKV-FastTune 面板。TiKV-FastTune 包括多组子面板，可帮助你诊断性能问题，尤其适用于集群中写入负载较大的场景。

当出现写入相关的性能问题时，可以先在 Grafana 中查看 TiDB 相关的面板。如果问题出在存储端，打开 TiKV-FastTune 面板，浏览并检查上面的每个指标。

在 TiKV-FastTune 的面板中，指标标题描述了性能问题的可能成因。要验证成因是否正确，你需要检查具体的图表曲线。

左边 Y 轴表示存储端的 write-RPC QPS，右边 Y 轴上的一组图是倒置绘制的。如果左边Y 轴的曲线形状与右边的形状匹配，则指标标题描述的问题成因是正确的。

有关该面板的具体监控项以及解释，参考TiKV-FastTune 用户手册（英文）。

4.5.6 TiFlash 集群监控

使用TiUP 部署TiDB 集群时，一键部署监控系统(Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

TiFlash 面板一共包括TiFlash-Summary、 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details。通过面板上的指标，可以了解 TiFlash 当前的状态。其中TiFlash-Proxy-Summary、 TiFlash-Proxy-Details 主要为TiFlash 的 Raft 层信息，其监控指标信息可参考TiKV 监控指标详解。

> 注意：
> 低版本的TiFlash 监控信息较不完善，如有需要推荐使用 v4.0.5 或更高版本的TiDB 集群。

以下为TiFlash-Summary 默认的监控信息：

4.5.6.1 Server

- Store size：每个 TiFlash 实例的使用的存储空间的大小。
- Available size：每个 TiFlash 实例的可用的存储空间的大小。
- Capacity size：每个 TiFlash 实例的存储容量的大小。
- Uptime：自上次重启以来 TiFlash 正常运行的时间。
- Memory：每个 TiFlash 实例内存的使用情况。

- CPU Usage：每个 TiFlash 实例 CPU 的使用率。
- FSync OPS：每个 TiFlash 实例每秒进行 fsync 操作的次数。
- File Open OPS：每个 TiFlash 实例每秒进行 open 操作的次数。
- Opened File Count：当前每个 TiFlash 实例打开的文件句柄数。

> **注意：**
>
> Store size、 FSync OPS、 File Open OPS、 Opened File Count 目前仅包含了TiFlash 存储层的统计指标，未包括TiFlash-Proxy 内的信息。

4.5.6.2 Coprocessor

- Request QPS：所有 TiFlash 实例收到的coprocessor 请求数量。其中 batch 是batch 请求数量，batch_cop 是batch 请求中的coprocessor 请求数量，cop 是直接通过coprocessor 接口发送的coprocessor 请求数量，cop_dag 是所有coprocessor 请求中dag 请求数量，super_batch 是开启 super batch 特性的请求数量。
- Executor QPS：所有 TiFlash 实例收到的请求中，每种dag 算子的数量，其中table_scan 是扫表算子，selection 是过滤算子，aggregation 是聚合算子，top_n 是 TopN 算子，limit 是 limit 算子。
- Request Duration：所有 TiFlash 实例处理coprocessor request 总时间，总时间为接收到该 coprocessor 请求至请求应答完毕的时间。
- Error QPS： 所有 TiFlash 实例处理 coprocessor 请求的错误数量。其中 meet_lock 为读取的数据有锁，region_not_found 为 Region 不存在，epoch_not_match 为读取的Region epoch 与本地不一致，kv_client_error 为与 TiKV 通信产生的错误，internal_error 为 TiFlash 内部系统错误，other 为其他错误。
- Request Handle Duration：所有 TiFlash 实例处理coprocessor 请求处理时间，处理时间为该coprocessor 请求开始执行到执行结束的时间。
- Response Bytes/Seconds：所有 TiFlash 实例应答总字节数。
- Cop task memory usage：所有TiFlash 实例处理coprocessor 请求占用的总内存。
- Handling Request Number：所有 TiFlash 实例正在处理的coprocessor 请求数量之和。请求的分类与 Request QPS 中的分类相同。
- Threads of RPC：每个TiFlash 实例使用的实时RPC 线程数。
- Max Threads of RPC：最近一段时间每个TiFlash 实例使用的RPC 线程数峰值。
- Threads：每个 TiFlash 实例使用的实时线程数。
- Max Threads：最近一段时间每个 TiFlash 实例使用的线程数峰值。

4.5.6.3 Task Scheduler

- Min TSO：每个 TiFlash 实例上正在运行的查询语句中的最小TSO，该值确保具有最小 TSO 的查询可以被调度。如果当前没有正在运行的查询，则该值为 `uint64` 整数型最大值。
- Estimated Thread Usage and Limit：每个 TiFlash 实例上正在运行的所有任务占用的线程估值，以及该实例上任务调度器设置的估算线程用量的软限制和硬限制。
- Active and Waiting Queries Count：每个 TiFlash 实例上正在运行的查询数量和正在等待的查询数量。
- Active and Waiting Tasks Count：每个 TiFlash 实例上正在运行的任务数量和正在等待的任务数量。
- Hard Limit Exceeded Count：每个 TiFlash 实例上运行中任务的估算线程用量超过了设置的硬限制的次数。
- Task Waiting Duration：每个 TiFlash 实例上任务从初始化到被调度的等待时长。

4.5.6.4 DDL

- Schema Version：每个 TiFlash 实例目前缓存的 schema 版本。
- Schema Apply OPM：所有 TiFlash 实例每分钟apply 同步 TiDB schema diff 的次数。diff apply 是正常的单次apply 过程，如果diff apply 失败，则 failed apply +1，并回退到 full apply，拉取最新的 schema 信息以更新TiFlash 的 schema 版本。
- Schema Internal DDL OPM：所有 TiFlash 实例每分钟执行的内部 DDL 次数。
- Schema Apply Duration：所有 TiFlash 实例apply schema 消耗的时间。

4.5.6.5 Storage

- Write Command OPS：所有 TiFlash 实例存储层每秒收到的写请求数量。
- Write Amplification：每个TiFlash 实例写放大倍数（实际磁盘写入量/逻辑数据写入量）。total 为自此次启动以来的写放大倍数，5min 为最近 5 分钟内的写放大倍数。
- Read Tasks OPS：每个 TiFlash 实例每秒存储层内部读取任务的数量。
- Rough Set Filter Rate：每个TiFlash 实例最近 1 分钟内读取的packet 数被存储层粗糙索引过滤的比例。
- Internal Tasks OPS：所有 TiFlash 实例每秒进行内部数据整理任务的次数。
- Internal Tasks Duration：所有 TiFlash 实例进行内部数据整理任务消耗的时间。
- Page GC Tasks OPM：所有 TiFlash 实例每分钟进行 Delta 部分数据整理任务的次数。
- Page GC Tasks Duration：所有 TiFlash 实例进行 Delta 部分数据整理任务消耗的时间分布。
- Disk Write OPS：所有 TiFlash 实例每秒进行磁盘写入的次数。
- Disk Read OPS：所有 TiFlash 实例每秒进行磁盘读取的次数。
- Write flow：所有 TiFlash 实例磁盘写操作的流量。
- Read flow：所有 TiFlash 实例磁盘读操作的流量。

> 注意：
>
> 目前这部分监控指标仅包含了TiFlash 存储层的统计指标，未包括TiFlash-Proxy 内的信息。

4.5.6.6 Storage Write Stall

- Write & Delta Management Throughput：所有实例写入及数据整理的吞吐量。
  - `throughput_write` 表示通过Raft 进行数据同步的吞吐量。
  - `throughput_delta-management` 表示数据整理的吞吐量。
  - `total_write` 表示自上次启动以来的总写入字节数。
  - `total_delta-management` 表示自上次启动以来数据整理的总字节数。
- Write Stall Duration：每个实例写入和移除 Region 数据产生的卡顿时长。
- Write Throughput By Instance：每个实例写入数据的吞吐量，包括apply Raft 数据日志以及Raft 快照的写入吞吐量。
- Write Command OPS By Instance：每个实例收到各种命令的总计数。
  - `write block` 表示通过Raft 同步数据日志。

- delete_range 表示从该实例中删除一些Region 或移动一些Region 到该实例中。
- ingest 表示这些Region 的快照被应用到这个实例中。

4.5.6.7 Raft

- Read Index OPS：每个 TiFlash 实例每秒触发read_index 请求的次数，等于请求触发的Region 总数。
- Read Index Duration：所有 TiFlash 实例在进行read_index 消耗的时间，主要消耗在于和Region leader 的交互和重试时间。
- Wait Index Duration：所有 TiFlash 实例在进行 wait_index 消耗的时间，即拿到read_index 请求后，等待本地的 Region index >= read_index 所花费的时间。

4.5.7 TiCDC 详细监控指标

使用TiUP 部署TiDB 集群时，一键部署的监控系统面板包含TiCDC 面板。本文档对TiCDC 监控面板上的各项指标进行详细说明。在日常运维中，运维人员可通过观察TiCDC 面板上的指标了解TiCDC 当前的状态。

本文档的对指标的介绍基于以下同步任务，即使用默认配置同步数据到 MySQL。

```
cdc cli changefeed create --server=http://10.0.10.25:8300 --sink-uri="mysql://root:123456@127
    <⟶ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

下图显示了TiCDC Dashboard 各监控面板：

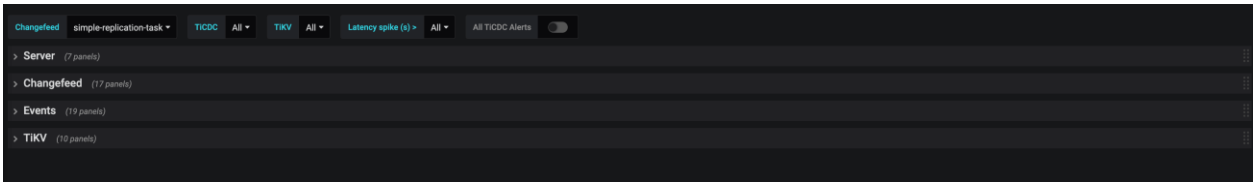


图 75: TiCDC Dashboard - Overview

各监控面板说明如下：

- Server：TiDB 集群中TiKV 节点和TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务的详细信息
- Events：TiCDC 内部数据流转的详细信息
- TiKV：TiKV 中和TiCDC 相关的详细信息

4.5.7.1 Server 面板

Server 面板示例如下：



图 76: TiCDC Dashboard - Server metrics

Server 面板的各指标说明如下：

- Uptime：TiKV 节点和TiCDC 节点已经运行的时间
- Goroutine count：TiCDC 节点 Goroutine 的个数
- Open FD count：TiCDC 节点打开的文件句柄个数
- Ownership：TiCDC 集群中节点的当前状态
- Ownership history：TiCDC 集群中 Owner 节点的历史记录
- CPU usage：TiCDC 节点使用的 CPU
- Memory usage：TiCDC 节点使用的内存

4.5.7.2 Changefeed 面板

Changefeed 面板示例如下：



图 77: TiCDC Dashboard - Changefeed metrics 1

- Changefeed table count：一个同步任务中分配到各个TiCDC 节点同步的数据表个数
- Processor resolved ts：TiCDC 节点内部状态中已同步的时间点
- Table resolved ts：同步任务中各数据表的同步进度
- Changefeed checkpoint：同步任务同步到下游的进度，正常情况下绿柱应和黄线相接
- PD etcd requests/s：TiCDC 节点每秒向 PD 读写数据的次数
- Exit error count/m：每分钟内导致同步中断的错误发生次数
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间计算）
- Processor resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间计算）



图 78: TiCDC Dashboard - Changefeed metrics 2

- Sink write duration：TiCDC 将一个事务的更改写到下游的耗时直方图
- Sink write duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 将一个事务的更改写到下游所花费的时间
- Flush sink duration：TiCDC 异步刷写数据入下游的耗时直方图
- Flush sink duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 异步刷写数据入下游所花费的时间

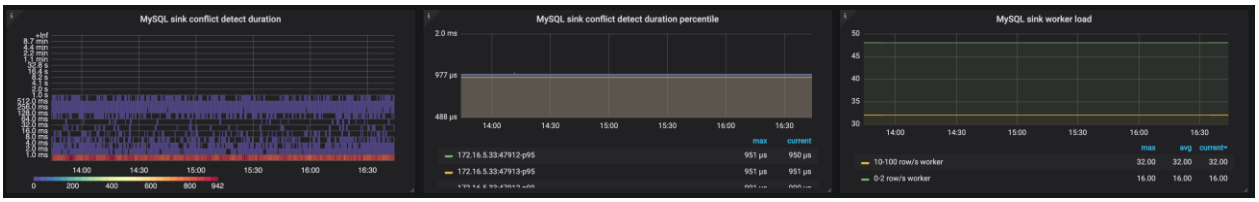


图 79: TiCDC Dashboard - Changefeed metrics 3

- MySQL sink conflict detect duration：MySQL 写入冲突检测耗时直方图

- MySQL sink conflict detect duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，MySQL 写入冲突检测耗时
- MySQL sink worker load：TiCDC 节点中写 MySQL 线程的负载情况

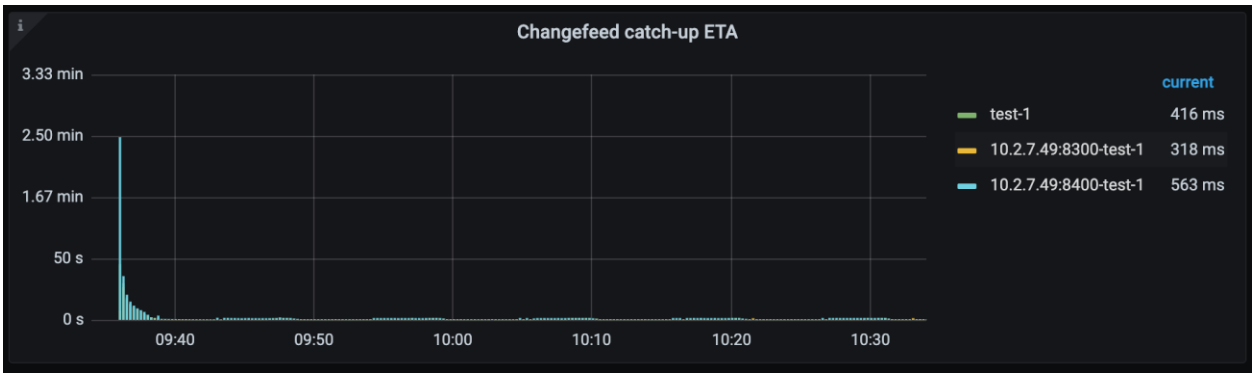


图 80: TiCDC Dashboard - Changefeed metrics 4

- Changefeed catch-up ETA：同步完上游写入的数据所需时间的估计值。当上游的写入速度大于 TiCDC 同步速度时，该值可能会异常的大。（由于 TiCDC 的同步速度受到较多因素制约，因此该值仅供参考，不能完全代表实际所需的同步时间。）

4.5.7.3 Events 面板

Events 面板示例如下：





Events 面板的各指标说明如下：

- Eventfeed count：TiCDC 节点中Eventfeed RPC 的个数
- Event size percentile：每秒钟中 95% 和 99.9% 的情况下，TiCDC 收到的来自TiKV 的数据变更消息大小
- Eventfeed error/m：TiCDC 节点中每分钟Eventfeed RPC 遇到的错误个数
- KV client receive events/s：TiCDC 节点中 KV client 模块每秒收到来自TiKV 的数据变更个数
- Puller receive events/s：TiCDC 节点中 Puller 模块每秒收到来自 KV client 模块的数据变更个数
- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更个数
- Sink flush rows/s：TiCDC 节点每秒写到下游的数据变更的个数
- Puller buffer size：TiCDC 节点中缓存在 Puller 模块中的数据变更个数
- Entry sorter buffer size：TiCDC 节点中缓存在 Sorter 模块中的数据变更个数
- Processor/Mounter buffer size：TiCDC 节点中缓存在 Processor 模块和 Mounter 模块中的数据变更个数
- Sink row buffer size：TiCDC 节点中缓存在 Sink 模块中的数据变更个数
- Entry sorter sort duration：TiCDC 节点对数据变更进行排序的耗时直方图
- Entry sorter sort duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 排序数据变更所花费的时间
- Entry sorter merge duration：TiCDC 节点合并排序后的数据变更的耗时直方图
- Entry sorter merge duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 合并排序后的数据变更所花费的时间
- Mounter unmarshal duration：TiCDC 节点解码数据变更的耗时直方图
- Mounter unmarshal duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 解码数据变更所花费的时间
- KV client dispatch events/s：TiCDC 节点内部 KV client 模块每秒分发数据变更的个数
- KV client batch resolved size：TiKV 批量发给TiCDC 的 resolved ts 消息的大小

4.5.7.4 TiKV 面板

TiKV 面板示例如下：



TiKV 面板的各指标说明如下：

- CDC endpoint CPU：TiKV 节点上 CDC endpoint 线程使用的 CPU
- CDC worker CPU：TiKV 节点上 CDC worker 线程使用的 CPU
- Min resolved ts：TiKV 节点上最小的resolved ts
- Min resolved region：TiKV 节点上最小的resolved ts 的 Region ID
- Resolved ts lag duration percentile：TiKV 节点上最小的resolved ts 与当前时间的差距
- Initial scan duration：TiKV 节点与TiCDC 建立链接时增量扫的耗时直方图
- Initial scan duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiKV 节点增量扫的耗时
- Memory without block cache：TiKV 节点在减去RocksDB block cache 后使用的内存
- CDC pending bytes in memory：TiKV 节点中 CDC 模块使用的内存
- Captured region count：TiKV 节点上捕获数据变更的Region 个数

4.5.8 资源管控(Resource Control) 监控指标详解

使用TiUP 部署TiDB 集群时，可以一键部署监控系统 (Prometheus & Grafana)。监控架构请参见 TiDB 监控框架概述。

目前Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

如果你的集群配置了Resource Control ，通过观察Resource Control 面板上的 Metrics，你可以了解当前集群整体的资源消耗状态。

TiDB 使用令牌桶算法做流控，正如资源管控实现机制(RFC: Global Resource Control in TiDB) 中所描述：一个TiDB 节点可能存在多个Resource Group（资源组），将在 PD 端的GAC（Global Admission Control）进行流控。每个 TiDB 节点中的本地令牌桶（Local Token Buckets）将定期（默认 5 秒）与 PD 端的GAC 进行通信，以重新配置本地令牌。其中的本地令牌桶（Local Token Buckets）具体实现为 Resource Controller Client。

以下为Resource Control 关键监控指标的说明。

4.5.8.1 Request Unit 相关指标

- RU：以 Resource Group（资源组）为单位进行实时统计的Request Unit (RU) 消耗信息。`total` 为当前所有Resource Group 消耗的Request Unit 之和。每个 Resource Group 的 Request Unit 消耗等于其读消耗(Read Request Unit) 和写消耗(Write Request Unit) 之和。
- RU Per Query：平均每个 SQL 语句消耗的Request Unit 数量。计算方法是将前述 Request Unit 监控指标除以当前每秒执行的 SQL 语句数量。
- RRU：以 Resource Group 为单位进行实时统计的读请求 Read Request Unit 消耗信息。`total` 为当前所有Resource Group 消耗的Read Request Unit 之和。
- RRU Per Query：平均每个 SQL 语句消耗的Read Request Unit 数量。计算方法是将前述Read Request Unit 监控指标除以当前每秒执行的 SQL 语句数量。
- WRU：以 Resource Group 为单位进行实时统计的写请求 Write Request Unit 消耗信息。`total` 为当前所有Resource Group 消耗的Write Request Unit 之和。
- WRU Per Query：平均每个 SQL 语句消耗的Write Request Unit 数量。计算方法是将前述 Write Request Unit 监控指标除以当前每秒执行的 SQL 语句数量。

4.5.8.2 Resource 相关指标

- KV Request Count：以 Resource Group（资源组）为单位进行实时统计的 KV 请求数量，区分了读和写两种类型。`total` 为当前所有Resource Group 涉及的 KV 请求数量之和。
- KV Request Count Per Query：平均每个 SQL 语句涉及的读写 KV 请求数量。计算方法是将前述 KV Request Count 监控指标除以当前每秒执行的 SQL 语句数量。
- Bytes Read：以Resource Group 为单位进行实时统计的读取数据量。`total` 为当前所有Resource Group 读取数据量之和。
- Bytes Read Per Query：平均每个 SQL 语句的读取数据量。将前述 Bytes Read 监控指标除以当前每秒执行的 SQL 语句数量。
- Bytes Written：以Resource Group 为单位进行实时统计的写入数据量。`total` 为当前所有Resource Group 写入数据量之和。
- Bytes Written Per Query：平均每个 SQL 语句的写入数据量。计算方法是将前述 Bytes Written 监控指标除以当前每秒执行的 SQL 语句数量。
- KV CPU Time：以 Resource Group 为单位进行实时统计的 KV 层 CPU 时间消耗。`total` 为当前所有 Resource Group 消耗 KV 层 CPU 时间之和。
- SQL CPU Time：以Resource Group 为单位进行实时统计的 SQL 层 CPU 时间消耗。`total` 为当前所有Resource Group 消耗 SQL 层 CPU 时间之和。

4.5.8.3 Resource Controller Client 相关指标

- Active Resource Groups：实时统计各个Resource Controller Client 的 Resource Groups 数量。
- Total KV Request Count：以 Resource Group 为单位，实时统计各个 Resource Controller Client 的 KV 请求数量。`total` 为 Resource Controller Client 下 KV 请求数量之和。
- Failed KV Request Count：以 Resource Group 为单位，实时统计各个Resource Controller Client 的 KV 失败请求数量。`total` 为 Resource Controller Client 下 KV 失败请求数量之和。
- Successful KV Request Count：以 Resource Group 为单位，实时统计各个Resource Controller Client 的 KV 成功请求数量。`total` 为 Resource Controller Client 下 KV 成功请求数量之和。
- Successful KV Request Wait Duration (99/90)：以Resource Group 为单位，实时统计各个Resource Controller Client 成功 KV 请求的等待时间（不同百分位）。
- Token Request Handle Duration (999/99)：以Resource Group 为单位，实时统计各个Resource Controller Client 向 Server 端申请Token 等待的响应时间（不同百分位）。
- Token Request Count：以 Resource Group 为单位，实时统计各个 Resource Controller Client 向 Server 端申请Token 的次数。`successful` 和 `failed` 分别为Resource Controller Client 下申请Token 成功和失败数量之和。

4.6 安全加固

4.6.1 为TiDB 客户端服务端间通信开启加密传输

TiDB 服务端与客户端之间默认采用非加密连接，因而具备监视信道流量能力的第三方可以知悉TiDB 服务端与客户端之间发送和接受的数据，包括但不限于查询语句内容、查询结果等。若信道是不可信的，例如客户端是通过公网连接到TiDB 服务端的，则非加密连接容易造成信息泄露，建议使用加密连接确保安全性。

TiDB 服务端支持启用基于 TLS（传输层安全）协议的加密连接，协议与 MySQL 加密连接一致，现有 MySQL Client 如 MySQL Shell 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 TLSv1.0、TLSv1.1、TLSv1.2 和 TLSv1.3。

使用加密连接后，连接将具有以下安全性质：

- 保密性：流量明文被加密，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。

要为TiDB 客户端与服务端间的通信开启 TLS 加密传输，首先需要在TiDB 服务端通过配置开启 TLS 加密连接的支持，然后通过配置客户端应用程序使用 TLS 加密连接。一般情况下，如果服务端正确配置了 TLS 加密连接支持，客户端库都会自动启用 TLS 加密传输。

另外，与 MySQL 相同，TiDB 也支持在同一TCP 端口上开启 TLS 连接或非 TLS 连接。对于开启了 TLS 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。如需使用加密连接，你可以通过以下方式进行配置：

- 通过配置系统变量 `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`)，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'@'%' IDENTIFIED BY 'my_random_password' 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。

4.6.1.1 配置TiDB 服务端启用安全连接

要启用安全连接，请参考以下相关参数说明：

- auto-tls：启用证书自动生成功能（从 v5.2.0 开始）
- 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
- tls-version：可选，指定最低 TLS 版本，例如 TLSv1.2

auto tls 支持安全连接，但不提供客户端证书验证。有关证书验证和控制证书生成方式的说明，请参考下面配置 ssl-cert，ssl-key 和 ssl-ca 变量的建议：

- 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，才能在TiDB 服务端开启安全连接。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）。
- 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。
- 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。

> 注意：
>
> v5.2.0 版本之前，你可以使用 mysql_ssl_rsa_setup --datadir=./certs 生成证书。mysql_ssl_rsa_setup 工具是 MySQL Server 的一部分。

4.6.1.2 配置 MySQL Client 使用安全连接

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)

除此参数外，MySQL 8.0 客户端有两种 SSL 模式：

- --ssl-mode=VERIFY_CA：根据 --ssl-ca 签发的 CA 验证来自服务器的证书。
- --ssl-mode=VERIFY_IDENTITY：与 VERIFY_CA 相同，但也验证所连接的主机名是否与证书匹配。

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。

4.6.1.3 配置启用身份验证

若在 TiDB 服务端或 MySQL Client 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 Client 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 Client 身份，TiDB 服务端需配置 ssl-cert、ssl-key、ssl-ca 参数，客户端需至少指定 --ssl-cert、--ssl-key 参数。必须确保服务端配置的证书和客户端配置的证书都是由服务端配置指定的 ssl-ca 签发的。
- 若要进行双向身份验证，请同时满足上述要求。

默认情况，服务端对客户端的身份验证是可选的。若客户端在 TLS 握手时未出示自己的身份证书，也能正常建立 TLS 连接。但也可以通过在创建用户(create user)，赋予权限 (grant) 或修改已有用户(alter user) 时指定 require x509 要求客户端需进行身份验证，以创建用户为例：

```
create user 'u1'@'%'  require x509;
```

> 注意：
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户需进行身份验证。

4.6.1.4 检查当前连接是否是加密连接

可以通过 SHOW STATUS LIKE "%Ssl%"; 了解当前连接的详细情况，包括是否使用了安全连接、安全连接采用的加密协议、TLS 版本号等。

以下是一个安全连接中执行该语句的结果。由于客户端支持的 TLS 版本号和加密协议会有所不同，执行结果相应地也会有所变化。

```
SHOW STATUS LIKE "%Ssl%";
```

```
......
| Ssl_verify_mode  | 5                           |
| Ssl_version      | TLSv1.2                     |
| Ssl_cipher       | ECDHE-RSA-AES128-GCM-SHA256 |
......
```

除此以外，对于 MySQL 自带客户端，还可以使用 STATUS 或 `\s` 语句查看连接情况：

```
\s
```

```
...
SSL: Cipher in use is ECDHE-RSA-AES128-GCM-SHA256
...
```

4.6.1.5 支持的 TLS 版本及密钥交换协议和加密算法

TiDB 支持的 TLS 版本及密钥交换协议和加密算法由Golang 官方库决定。

你使用的客户端库和操作系统加密策略也可能会影响到支持的协议版本和密码套件。

4.6.1.5.1 支持的 TLS 版本

- TLSv1.0 （默认禁用）
- TLSv1.1
- TLSv1.2
- TLSv1.3

可以使用配置项 `tls-version` 来限制 TLS 版本。

实际可用的 TLS 版本取决于操作系统的加密策略、 MySQL 客户端版本和客户端的 SSL/TLS 库。

4.6.1.5.2 支持的密钥交换协议及加密算法

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

4.6.1.6 重加载证书、密钥和 CA

在需要替换证书、密钥或 CA 时，可以在完成对应文件替换后，对运行中的TiDB 实例执行`ALTER INSTANCE <-> RELOAD TLS` 语句从原配置的证书(`ssl-cert`)、密钥 (`ssl-key`) 和 CA (`ssl-ca`) 的路径重新加证书、密钥和 CA，而无需重启TiDB 实例。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

4.6.1.7 监控

自 TiDB v5.2.0 版本起，你可以使用 `Ssl_server_not_after` 和 `Ssl_server_not_before` 状态变量监控证书有效期的起止时间。

```
SHOW GLOBAL STATUS LIKE 'Ssl\_server\_not\_%';
```

```
+-----------------------+--------------------------------------------+
| Variable_name         | Value                                      |
+-----------------------+--------------------------------------------+
| Ssl_server_not_after  | Nov 28 06:42:32 2021 UTC |
| Ssl_server_not_before | Aug 30 06:42:32 2021 UTC |
+-----------------------+--------------------------------------------+
2 rows in set (0.0076 sec)
```

4.6.1.8 另请参阅

- 为TiDB 组件间通信开启加密传输

4.6.2 为TiDB 组件间通信开启加密传输

本部分介绍如何为TiDB 集群内各组件间开启加密传输。一旦开启，以下组件间均将使用加密传输：

- TiDB、 TiKV、 PD、 TiFlash 之间的通讯
- TiDB Control 与 TiDB，TiKV Control 与 TiKV，PD Control 与 PD
- TiKV、 PD、 TiDB、 TiFlash 各自集群内内部通讯

目前暂不支持只开启其中部分组件的加密传输。

4.6.2.1 配置开启加密传输

1. 准备证书。

   推荐为TiDB、 TiKV、 PD 分别准备一个 Server 证书，并保证可以相互验证，而它们的 Control 工具则可选择共用 Client 证书。

   有多种工具可以生成自签名证书，如 openssl，easy-rsa，cfssl。

   这里提供一个使用 openssl 生成证书的示例：生成自签名证书。

2. 配置证书。

   - TiDB

     在 config 文件或命令行参数中设置：

     ```
     [security]
     # Path of file that contains list of trusted SSL CAs for connection with cluster
         <↔ components.
     cluster-ssl-ca = "/path/to/ca.pem"
     # Path of file that contains X509 certificate in PEM format for connection with cluster
         <↔ components.
     cluster-ssl-cert = "/path/to/tidb-server.pem"
     # Path of file that contains X509 key in PEM format for connection with cluster
         <↔ components.
     cluster-ssl-key = "/path/to/tidb-server-key.pem"
     ```

   - TiKV

     在 config 文件或命令行参数中设置，并设置相应的 URL 为https：

     ```
     [security]
     # set the path for certificates. Empty string means disabling secure connectoins.
     ca-path = "/path/to/ca.pem"
     cert-path = "/path/to/tikv-server.pem"
     key-path = "/path/to/tikv-server-key.pem"
     ```

   - PD

     在 config 文件或命令行参数中设置，并设置相应的 URL 为https：

     ```
     [security]
     # Path of file that contains list of trusted SSL CAs. if set, following four settings
         <↔ shouldn't be empty
     cacert-path = "/path/to/ca.pem"
     # Path of file that contains X509 certificate in PEM format.
     cert-path = "/path/to/pd-server.pem"
     # Path of file that contains X509 key in PEM format.
     key-path = "/path/to/pd-server-key.pem"
     ```

   - TiFlash（从 v4.0.5 版本开始引入）

     在 tiflash.toml 文件中设置，将 http_port 项改为 https_port:

```
[security]
# Path of file that contains list of trusted SSL CAs. if set, following four settings
    <-> shouldn't be empty
ca_path = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format.
cert_path = "/path/to/tiflash-server.pem"
# Path of file that contains X509 key in PEM format.
key_path = "/path/to/tiflash-server-key.pem"
```

在 tiflash-learner.toml 文件中设置，

```
[security]
# Sets the path for certificates. The empty string means that secure connections are
    <-> disabled.
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/tiflash-server.pem"
key-path = "/path/to/tiflash-server-key.pem"
```

- TiCDC

  在 config 文件中设置

```
[security]
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/cdc-server.pem"
key-path = "/path/to/cdc-server-key.pem"
```

  或者在启动命令行中设置，并设置相应的 URL 为 https：

```
cdc server --pd=https://127.0.0.1:2379 --log-file=ticdc.log --addr=0.0.0.0:8301 --
    <-> advertise-addr=127.0.0.1:8301 --ca=/path/to/ca.pem --cert=/path/to/ticdc-cert.
    <-> pem --key=/path/to/ticdc-key.pem
```

此时TiDB 集群各个组件间已开启加密传输。

> 注意：
> 若 TiDB 集群各个组件间开启加密传输后，在使用tidb-ctl、 tikv-ctl 或 pd-ctl 工具连接集群时，需要指定client 证书，示例：

```
./tidb-ctl -u https://127.0.0.1:10080 --ca /path/to/ca.pem --ssl-cert /path/to/client.pem --
    <-> ssl-key /path/to/client-key.pem
```

```
tiup ctl:v<CLUSTER_VERSION> pd -u https://127.0.0.1:2379 --cacert /path/to/ca.pem --cert /
    <-> path/to/client.pem --key /path/to/client-key.pem
```

```
./tikv-ctl --host="127.0.0.1:20160" --ca-path="/path/to/ca.pem" --cert-path="/path/to/client
    <-> .pem" --key-path="/path/to/clinet-key.pem"
```

4.6.2.2 认证组件调用者身份

通常被调用者除了校验调用者提供的密钥、证书和 CA 有效性外，还需要校验调用方身份以防止拥有有效证书的非法访问者进行访问（例如：TiKV 只能被TiDB 访问，需阻止拥有合法证书但非 TiDB 的其他访问者访问TiKV）。

如希望进行组件调用者身份认证，需要在生证书时通过 Common Name 标识证书使用者身份，并在被调用者配置检查证书 Common Name 列表来检查调用者身份。

- TiDB

  在 config 文件或命令行参数中设置：

```
[security]
cluster-verify-cn =
  [ "TiDB-Server",
  "TiKV-Control",
]
```

- TiKV

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = [
    "TiDB-Server", "PD-Server", "TiKV-Control", "RawKvClient1",
]
```

- PD

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = ["TiKV-Server", "TiDB-Server", "PD-Control"]
```

- TiFlash（从 v4.0.5 版本开始引入）

  在 tiflash.toml 文件中设置：

```
[security]
cert_allowed_cn = ["TiKV-Server", "TiDB-Server"]
```

  在 tiflash-learner.toml 文件中设置：

```
[security]
cert-allowed-cn = ["PD-Server", "TiKV-Server", "TiFlash-Server"]
```

4.6.2.3 证书重新加载

- 如果TiDB 集群部署在本地的数据中心，TiDB、 PD、 TiKV、 TiFlash、 TiCDC 和各种client 在每次新建相互通讯的连接时都会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新加载，无需重启TiDB 集群。
- 如果TiDB 集群部署在自己管理的 Cloud，TLS 证书的签发需要与云服务商的证书管理服务集成，TiDB、 PD、 TiKV、 TiFlash、 TiCDC 组件的 TLS 证书支持自动轮换，无需重启TiDB 集群。

4.6.2.4 证书有效期

你可以自定义TiDB 集群中各组件 TLS 证书的有效期。例如，使用 OpenSSL 签发生成 TLS 证书时，可以通过days参数设置有效期，详见生成自签名证书。

4.6.2.5 另请参阅

- 为TiDB 客户端服务端间通信开启加密传输

4.6.3 生成自签名证书

> **注意：**
>
> 要在TiDB 客户端与服务端间通信开启加密传输，你只需配置 `auto-tls` 参数。

本文档提供使用 `openssl` 生成自签名证书的一个示例，用户也可以根据自己的需求生成符合要求的证书和密钥。

假设实例集群拓扑如下：

Name	Host IP	Services
node1	172.16.10.11	PD1, TiDB1
node2	172.16.10.12	PD2
node3	172.16.10.13	PD3
node4	172.16.10.14	TiKV1
node5	172.16.10.15	TiKV2
node6	172.16.10.16	TiKV3

4.6.3.1 安装 OpenSSL

对于 Debian 或 Ubuntu 操作系统：

```
apt install openssl
```

对于 RedHat 或 CentOS 操作系统：

```
yum install openssl
```

也可以参考 OpenSSL 官方的下载文档进行安装。

4.6.3.2 生成 CA 证书

CA 的作用是签发证书。实际情况中，请联系你的管理员签发证书或者使用信任的 CA 机构。CA 会管理多个证书对，这里只需生成原始的一对证书，步骤如下：

1. 生成 root 密钥：

```
openssl genrsa -out root.key 4096
```

2. 生成 root 证书：

```
openssl req -new -x509 -days 1000 -key root.key -out root.crt
```

3. 验证 root 证书：

```
openssl x509 -text -in root.crt -noout
```

4.6.3.3 签发各个组件的证书

4.6.3.3.1 集群中可能使用到的证书

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

4.6.3.3.2 给 TiKV 实例签发证书

给TiKV 实例签发证书的步骤如下：

1. 生成该证书对应的私钥：

```
openssl genrsa -out tikv.key 2048
```

2. 拷贝一份 OpenSSL 的配置模板文件。

   模板文件可能存在多个位置，请以实际位置为准：

```
cp /usr/lib/ssl/openssl.cnf .
```

   如果不知道实际位置，请在根目录下查找：

```
find / -name openssl.cnf
```

3. 编辑 `openssl.cnf`，在 `[ req ]` 字段下加入 `req_extensions = v3_req`，然后在 `[ v3_req ]` 字段下加入 `subjectAltName = @alt_names`。最后新建一个字段，并编辑 SAN 的信息：

```
[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 172.16.10.14
IP.3 = 172.16.10.15
IP.4 = 172.16.10.16
```

4. 保存 openssl.cnf 文件后，生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，各个组件默认不会开启验证，需要在配置文件中启用该功能才生效）：

```
openssl req -new -key tikv.key -out tikv.csr -config openssl.cnf
```

5. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in tikv.csr -out
    <-> tikv.crt -extensions v3_req -extfile openssl.cnf
```

6. 验证证书携带 SAN 字段信息（可选）：

```
openssl x509 -text -in tikv.crt -noout
```

7. 确认在当前目录下得到如下文件：

```
root.crt
tikv.crt
tikv.key
```

为其它TiDB 组件签发证书的过程类似，此文档不再赘述。

4.6.3.3.3 为客户端签发证书

为客户端签发证书的步骤如下。

1. 生成该证书对应的私钥：

```
openssl genrsa -out client.key 2048
```

2. 生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，默认不会开启对各个组件的验证，需要在配置文件中启用该功能才生效）

```
openssl req -new -key client.key -out client.csr
```

3. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in client.csr -out
    <-> client.crt
```

4.6.4 静态加密

> **注意：**
>
> 如果集群部署在AWS 上并在使用EBS (Elastic Block Store) 存储数据，建议使用EBS 加密，详细信息请参考AWS 文档- EBS 加密。如果集群部署在AWS 上，但未使用EBS 存储（例如使用本地NVMe 存储），则建议使用本文中介绍的静态加密方式。

静态加密(encryption at rest) 即在存储数据时进行数据加密。对于数据库，静态加密功能也叫透明数据加密 (TDE)，区别于传输数据加密(TLS) 或使用数据加密（很少使用）。SSD 驱动器、文件系统、云供应商等都可进行静态加密。但区别于这些加密方式，若TiKV 在存储数据前就进行数据加密，攻击者则必须通过数据库的身份验证才能访问数据。例如，即使攻击者获得物理机的访问权限时，也无法通过复制磁盘上的文件来访问数据。

4.6.4.1 各 TiDB 组件支持的加密方式

在一个TiDB 集群中，不同的组件使用不同的加密方式。本节介绍TiKV、 TiFlash、 PD 和 Backup & Restore (BR) 等不同 TiDB 组件支持的加密方式。

部署好一个TiDB 集群后，大部分用户数据会存储在TiKV 和TiFlash 节点上。此外，一些元数据会存储在 PD 节点上，例如用作为TiKV Region 边界的二级索引密钥。为了能够充分发挥静态加密的优势，所有组件都需要启用加密功能。此外，进行加密时，也需要对备份、日志文件和通过网络传输的数据进行加密。

4.6.4.1.1 TiKV

TiKV 支持静态加密，即在 CTR 模式下使用AES 或 SM4 对数据文件进行透明加密。要启用静态加密，用户需提供一个加密密钥，即主密钥。TiKV 自动轮换(rotate) 用于加密实际数据文件的密钥，主密钥则可以由用户手动轮换。请注意，静态加密仅加密静态数据（即磁盘上的数据），而不加密网络传输中的数据。建议 TLS 与静态加密一起使用。

可以选择将AWS KMS (Key Management Service) 用于云上部署或本地部署，也可以指定将密钥以明文形式存储在文件中。

TiKV 当前不从核心转储(core dumps) 中排除加密密钥和用户数据。建议在使用静态加密时禁用TiKV 进程的核心转储，该功能目前无法由TiKV 独立处理。

TiKV 使用文件的绝对路径来跟踪已加密的数据文件。一旦TiKV 节点开启了加密功能，用户就不应更改数据文件的路径配置，例如 `storage.data-dir`，`raftstore.raftdb-path`，`rocksdb.wal-dir` 和 `raftdb.wal-dir`。

SM4 加密只在 v6.3.0 及之后版本的TiKV 上支持。v6.3.0 之前的TiKV 仅支持AES 加密。SM4 加密会对吞吐造成 50% 到 80% 的回退。

4.6.4.1.2 TiFlash

TiFlash 支持静态加密。数据密钥由 TiFlash 生成。TiFlash（包括 TiFlash Proxy）写入的所有文件，包括数据文件、Schema 文件、临时文件等，均由当前数据密钥加密。TiFlash 支持的加密算法、加密配置方法（配置项在`tiflash-learner.toml` 中）和监控项含义等均与TiKV 一致。

如果TiFlash 中部署了Grafana 组件，可以查看TiFlash-Proxy-Details -> Encryption。

SM4 加密只在 v6.4.0 及之后版本的TiFlash 上支持。v6.4.0 之前的TiFlash 仅支持AES 加密。

4.6.4.1.3 PD

PD 的静态加密为实验特性，其配置方式与TiKV 相同。

4.6.4.1.4 BR 备份

BR 支持对备份到 S3 的数据进行 S3 服务端加密(SSE)。BR S3 服务端加密也支持使用用户自行创建的AWS KMS 密钥进行加密，详细信息请参考BR S3 服务端加密。

4.6.4.1.5 日志

TiKV，TiDB 和 PD 信息日志中可能包含用于调试的用户数据。信息日志不会被加密，建议开启日志脱敏功能。

4.6.4.2 TiKV 静态加密

TiKV 当前支持的加密算法包括AES128-CTR、AES192-CTR、AES256-CTR 和 SM4-CTR (仅 v6.3.0 及之后版本)。TiKV 使用信封加密(envelop encryption)，所以启用加密后，TiKV 使用以下两种类型的密钥：

- 主密钥(master key)：主密钥由用户提供，用于加密 TiKV 生成的数据密钥。用户在TiKV 外部进行主密钥的管理。
- 数据密钥(data key)：数据密钥由 TiKV 生成，是实际用于加密的密钥。

多个TiKV 实例可共用一个主密钥。在生产环境中，推荐通过 AWS KMS 提供主密钥。首先通过 AWS KMS 创建用户主密钥(CMK)，然后在配置文件中将 CMK 密钥的ID 提供给TiKV。TiKV 进程在运行时可以通过IAM 角色访问 KMS CMK。如果 TiKV 无法访问 KMS CMK，TiKV 就无法启动或重新启动。详情参阅AWS 文档中的 KMS and IAM。

用户也可以通过文件形式提供主密钥。该文件须包含一个用十六进制字符串编码的 256 位（32 字节）密钥，并以换行符结尾（即 `\n`），且不包含其他任何内容。将密钥存储在磁盘上会泄漏密钥，因此密钥文件仅适合存储在RAM 内存的 `tempfs` 中。

数据密钥由TiKV 传递给底层存储引擎（即 RocksDB）。RocksDB 写入的所有文件，包括 SST 文件，WAL 文件和 MANIFEST 文件，均由当前数据密钥加密。TiKV 使用的其他临时文件（可能包括用户数据）也由相同的数据密钥加密。默认情况下，TiKV 每周自动轮换数据密钥，但是该时间段是可配置的。密钥轮换时，TiKV 不会重写全部现有文件来替换密钥，但如果集群的写入量恒定，则RocksDB compaction 会将使用最新的数据密钥对数据重新加密。TiKV 跟踪密钥和加密方法，并使用密钥信息对读取的内容进行解密。

无论用户配置了哪种数据加密方法，数据密钥都使用AES256-GCM 算法进行加密，以方便对主密钥进行验证。所以当使用文件而不是 KMS 方式指定主密钥时，主密钥必须为 256 位（32 字节）。

4.6.4.2.1 创建密钥

如需在AWS 上创建一个密钥，请执行以下步骤：

1. 进入AWS 控制台的AWS KMS。
2. 确保在控制台的右上角选择正确的区域。
3. 点击创建密钥，选择对称(Symmetric)作为密钥类型。
4. 为钥匙设置一个别名。

你也可以使用AWS CLI执行该操作：

```
aws --region us-west-2 kms create-key
aws --region us-west-2 kms create-alias --alias-name "alias/tidb-tde" --target-key-id 0987dcba-09
    <⟶ fe-87dc-65ba-ab0987654321
```

需要在第二条命令中输入的 `--target-key-id` 是第一条命令的结果。

4.6.4.2.2 配置加密

要启用加密，你可以在TiKV和PD的配置文件中添加加密部分：

```
[security.encryption]
data-encryption-method = "aes128-ctr"
data-key-rotation-period = "168h" # 7 days
```

`data-encryption-method` 的可选值为 `"aes128-ctr"`、`"aes192-ctr"`、`"aes256-ctr"`、`"sm4-ctr"` (仅 v6.3.0 及之后版本) 和 `"plaintext"`。默认值为 `"plaintext"`，即默认不开启加密功能。`data-key-rotation-period` 指定TiKV轮换密钥的频率。可以为新TiKV集群或现有TiKV集群开启加密，但只有启用后写入的数据才保证被加密。要禁用加密，请在配置文件中删除 `data-encryption-method`，或将该参数值为 `"plaintext"`，然后重启 TiKV。若要替换加密算法，则将 `data-encryption-method` 替换成已支持的加密算法，然后重启TiKV。替换加密算法后，旧加密算法生成的加密文件会随着新数据的写入逐渐被重写成新加密算法所生成的加密文件。

如果启用了加密（即 `data-encryption-method` 的值不是 `"plaintext"`），则必须指定主密钥。要使用 AWS KMS 方式指定为主密钥，请在 `[security.encryption]` 部分之后添加 `[security.encryption.master-key]` 部分：

```
[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
endpoint = "https://kms.us-west-2.amazonaws.com"
```

`key-id` 指定 KMS CMK 的密钥ID。`region` 为 KMS CMK 的AWS 区域名。`endpoint` 通常无需指定，除非你在使用非AWS 提供的AWS KMS 兼容服务或需要使用 KMS VPC endpoint。

你也可以使用AWS 多区域键。为此，你需要在一个特定的区域设置一个主键，并在需要的区域中添加副本密钥。

若要使用文件方式指定主密钥，主密钥配置应如下所示：

```
[security.encryption.master-key]
type = "file"
path = "/path/to/key/file"
```

以上示例中，`path` 为密钥文件的路径。该文件须包含一个 256 位（32 字节）的十六进制字符串，并以换行符结尾（即 `\n`），且不包含其他任何内容。密钥文件示例如下：

```
3b5896b5be691006e0f71c3040a29495ddcad20b14aff61806940ebd780d3c62
```

4.6.4.2.3 轮换主密钥

要轮换主密钥，你必须在配置中同时指定新的主密钥和旧的主密钥，然后重启TiKV。用 `security.encryption.master-key` 指定新的主密钥，用 `security.encryption.previous-master-key` 指定旧的主密钥。`security.encryption.previous-master-key` 配置的格式与 `encryption.master-key` 相同。重启时，TiKV 必须能同时访问旧主密钥和新主密钥，但一旦TiKV 启动并运行，就只需访问新密钥。此后，可以将 `encryption.previous-master-key` 项保留在配置文件中。即使重启时，TiKV 也只会在无法使用新的主密钥解密现有数据时尝试使用旧密钥。

TiKV 当前不支持在线轮换主密钥，因此你需要重启TiKV 进行主密钥轮换。建议对运行中的、提供在线查询的 TiKV 集群进行滚动重启。

轮换 KMS CMK 的配置示例如下：

```
[security.encryption.master-key]
type = "kms"
key-id = "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
region = "us-west-2"

[security.encryption.previous-master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

4.6.4.2.4 监控和调试

要监控静态加密（如果TiKV 中部署了Grafana 组件），可以查看 TiKV-Details -> Encryption 面板中的监控项：

- Encryption initialized：如果在TiKV 启动期间初始化了加密，则为 1，否则为 0。进行主密钥轮换时可通过该监控项确认主密钥轮换是否已完成。
- Encryption data keys：现有数据密钥的数量。每次轮换数据密钥后，该数字都会增加 1。通过此监控指标可以监测数据密钥是否按预期轮换。
- Encrypted files：当前的加密数据文件数量。为先前未加密的集群启用加密时，将此数量与数据目录中的当前数据文件进行比较，可通过此监控指标估计已经被加密的数据量。
- Encryption meta file size：加密元数据文件的大小。
- Read/Write encryption meta duration：对用于加密的元数据进行操作带来的额外开销。

在调试方面，可使用 `tikv-ctl` 命令查看加密元数据（例如使用的加密方法和数据密钥列表）。该操作可能会暴露密钥，因此不推荐在生产环境中使用。详情参阅TiKV Control。

4.6.4.2.5 TiKV 版本间兼容性

为了减少TiKV 管理加密元数据造成的I/O 操作和互斥锁竞争引发的开销，TiKV v4.0.9 对此进行了优化。此优化可以通过TiKV 配置文件中的 `security.encryption.enable-file-dictionary-log` 参数启用或关闭。此配置参数仅在TiKV v4.0.9 或更高版本中生效。

该配置项默认开启，此时TiKV v4.0.8 或更早期的版本无法识别加密元数据的数据格式。例如，假设你正在使用 TiKV v4.0.9 或更高版本，开启了静态加密和默认开启了 `enable-file-dictionary-log` 配置。如果将集群降级到 TiKV v4.0.8 或更早版本，TiKV 将无法启动，并且信息日志中会有类似如下的报错：

```
[2020/12/07 07:26:31.106 +08:00] [ERROR] [mod.rs:110] ["encryption: failed to load file
    <→ dictionary."]
[2020/12/07 07:26:33.598 +08:00] [FATAL] [lib.rs:483] ["called `Result::unwrap()` on an `Err`
    <→ value: Other(\"[components/encryption/src/encrypted_file/header.rs:18]: unknown version
    <→ 2\")"]
```

为了避免上面所示错误，你可以首先将 `security.encryption.enable-file-dictionary-log` 设置为 `false`，然后启动TiKV v4.0.9 或更高版本。TiKV 成功启动后，加密元数据的数据格式将降级为TiKV 早期版本可以识别的格式。此时，你可再将TiKV 集群降级到较早的版本。

4.6.4.3 TiFlash 静态加密

TiFlash 当前支持的加密算法与TiKV 一致，包括AES128-CTR、AES192-CTR、AES256-CTR 和 SM4-CTR（仅 v6.4.0 及之后版本）。TiFlash 同样使用信封加密(envelop encryption)，所以启用加密后，TiFlash 使用以下两种类型的密钥：

- 主密钥(master key)：主密钥由用户提供，用于加密 TiFlash 生成的数据密钥。用户在TiFlash 外部进行主密钥的管理。
- 数据密钥(data key)：数据密钥由 TiFlash 生成，是实际用于加密的密钥。

多个TiFlash 实例可共用一个主密钥，并且也可以和TiKV 共用一个主密钥。在生产环境中，推荐通过AWS KMS 提供主密钥。另外，你也可以通过文件形式提供主密钥。具体的主密钥生成方式和格式均与TiKV 相同。

TiFlash 使用数据密钥加密所有落盘的数据文件，包括数据文件、Schmea 文件和计算过程中产生的临时数据文件等。默认情况下，TiFlash 每周自动轮换数据密钥，该轮换周期也可根据需要自定义配置。密钥轮换时，TiFlash 不会重写全部现有文件来替换密钥，但如果集群的写入量恒定，则后台compaction 任务将会用最新的数据密钥对数据重新加密。TiFlash 跟踪密钥和加密方法，并使用密钥信息对读取的内容进行解密。

4.6.4.3.1 创建密钥

如需在AWS 上创建一个密钥，请参考TiKV 密钥的创建方式。

4.6.4.3.2 配置加密

启用加密的配置，可以在 `tiflash-learner.toml` 文件中添加以下内容：

```
[security.encryption]
data-encryption-method = "aes128-ctr"
data-key-rotation-period = "168h" # 7 days
```

或者，在TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.data-encryption-method: "aes128-ctr"
    security.encryption.data-key-rotation-period: "168h" # 7 days
```

data-encryption-method 的可选值为 "aes128-ctr"、"aes192-ctr"、"aes256-ctr"、"sm4-ctr" (仅 v6.4.0 及之后版本) 和 "plaintext"。默认值为 "plaintext"，即默认不开启加密功能。data-key-rotation-period 指定 TiFlash 轮换密钥的频率。可以为新 TiFlash 集群或现有TiFlash 集群开启加密，但只有启用后写入的数据才保证被加密。要禁用加密，请在配置文件中删除 data-encryption-method，或将该参数值为 "plaintext"，然后重启 TiFlash。若要替换加密算法，则将 data-encryption-method 替换成已支持的加密算法，然后重启TiFlash。替换加密算法后，旧加密算法生成的加密文件会随着新数据的写入逐渐被重写成新加密算法所生成的加密文件。

如果启用了加密 (即 data-encryption-method 的值不是 "plaintext")，则必须指定主密钥。要使用 AWS KMS 方式指定为主密钥，配置方式如下：

在 tiflash-learner.toml 配置文件的 [security.encryption] 部分之后添加 [security.encryption.master-key]：

```
[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
endpoint = "https://kms.us-west-2.amazonaws.com"
```

或者，在TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.master-key.type: "kms"
    security.encryption.master-key.key-id: "0987dcba-09fe-87dc-65ba-ab0987654321"
    security.encryption.master-key.region: "us-west-2"
    security.encryption.master-key.endpoint: "https://kms.us-west-2.amazonaws.com"
```

上述配置项的含义与TiKV 均相同。

若要使用文件方式指定主密钥，可以在 tiflash-learner.toml 配置文件中添加以下内容：

```
[security.encryption.master-key]
type = "file"
path = "/path/to/key/file"
```

或者，在TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.master-key.type: "file"
    security.encryption.master-key.path: "/path/to/key/file"
```

上述配置项的含义以及密钥文件的内容格式与TiKV 均相同。

4.6.4.3.3 轮换主密钥

TiFlash 轮换主秘钥的方法与TiKV 相同。TiFlash 当前也不支持在线轮换主密钥，因此你需要重启TiFlash 进行主密钥轮换。建议对运行中的、提供在线查询的TiFlash 集群进行滚动重启。

要轮换 KMS CMK，可以在 tiflash-learner.toml 配置文件中添加以下内容：

```
[security.encryption.master-key]
type = "kms"
key-id = "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
region = "us-west-2"

[security.encryption.previous-master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

或者，在TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.master-key.type: "kms"
    security.encryption.master-key.key-id: "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
    security.encryption.master-key.region: "us-west-2"
    security.encryption.previous-master-key.type: "kms"
    security.encryption.previous-master-key.key-id: "0987dcba-09fe-87dc-65ba-ab0987654321"
    security.encryption.previous-master-key.region: "us-west-2"
```

4.6.4.3.4 监控和调试

要监控静态加密（如果TiFlash 中部署了Grafana 组件），可以查看 TiFlash-Proxy-Details -> Encryption 面板中的监控项，其中监控项的含义与TiKV 相同。

在调试方面，由于TiFlash 复用了TiKV 管理加密元数据的逻辑，因此也可使用 `tikv-ctl` 命令查看加密元数据（例如使用的加密方法和数据密钥列表）。该操作可能会暴露密钥，因此不推荐在生产环境中使用。详情参阅TiKV Control。

4.6.4.3.5 TiFlash 版本间兼容性

TiFlash 在 v4.0.9 同样对加密元数据操作进行了优化，其兼容性要求与TiKV 相同，详情参考TiKV 版本间兼容性。

4.6.4.4 BR S3 服务端加密

使用 BR 备份数据到 S3 时，若要启用 S3 服务端加密，需要传递 `--s3.sse` 参数并将参数值设置为 `aws:kms`。S3 将使用自己的 KMS 密钥进行加密。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.sse aws:kms
```

若要使用用户创建和拥有的自定义AWS KMS CMK，需另外传递 `--s3.sse-kms-key-id` 参数。此时，BR 进程和集群中的所有TiKV 节点都需访问该 KMS CMK（例如，通过 AWS IAM），并且该 KMS CMK 必须与存储备份的 S3 bucket 位于同一AWS 区域。建议通过AWS IAM 向 BR 进程和TiKV 节点授予对 KMS CMK 的访问权限。参见AWS 文档中的IAM。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.sse aws:kms --s3.sse-
    <-> kms-key-id 0987dcba-09fe-87dc-65ba-ab0987654321
```

恢复备份时，不需要也不可指定 `--s3.sse` 和 `--s3.sse-kms-key-id` 参数。S3 将自动相应进行解密。用于恢复备份数据的 BR 进程和集群中的TiKV 节点也需要访问 KMS CMK，否则恢复将失败。示例如下：

```
./br restore full --pd <pd-address> --storage "s3://<bucket>/<prefix>"
```

4.6.5 为TiDB 落盘文件开启加密

当系统变量tidb_enable_tmp_storage_on_oom 为 ON 时， 如果单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 的限制，某些算子可以将执行时的中间结果作为临时文件落盘保存，直到查询执行完成之后将它们删除。

用户可以开启落盘文件加密功能，防止攻击者通过读取临时文件来访问数据。

4.6.5.1 配置方法

要启用落盘文件加密功能，可以在TiDB 配置文件中的[security] 部分，配置spilled-file-encryption-method 选项：

```
[security]
spilled-file-encryption-method = "aes128-ctr"
```

spilled-file-encryption-method 的可选值为 aes128-ctr 和 plaintext。默认值为 plaintext，表示不启用加密。

4.6.6 日志脱敏

TiDB 在提供详细的日志信息时，可能会把数据库敏感的数据（例如用户数据）打印出来，造成数据安全方面的风险。因此 TiDB、TiKV、PD 等组件各提供了一个配置项开关，开关打开后，会隐藏日志中包含的用户数据值。

4.6.6.1 TiDB 组件日志脱敏

TiDB 侧的日志脱敏需要将global.tidb_redact_log 的值设为 1。该变量值默认为 0，即关闭脱敏。

可以通过 set 语法，设置全局系统变量 tidb_redact_log，示例如下：

```
set @@global.tidb_redact_log=1;
```

设置后，所有新 session 产生的日志都会脱敏：

```
create table t (a int, unique key (a));
Query OK, 0 rows affected (0.00 sec)

insert into t values (1),(1);
ERROR 1062 (23000): Duplicate entry '1' for key 't.a'
```

打印出的错误日志如下：

```
[2020/10/20 11:45:49.539 +08:00] [INFO] [conn.go:800] ["command dispatched failed"] [conn=5] [
    <-> connInfo="id:5, addr:127.0.0.1:57222 status:10, collation:utf8_general_ci,  user:root"] [
    <-> command=Query] [status="inTxn:0, autocommit:1"] [sql="insert into t values ( ? ) , ( ? )
    <-> "] [txn_mode=OPTIMISTIC] [err="[kv:1062]Duplicate entry '?' for key 't.a'"]
```

从以上报错日志可以看到，开启 tidb_redact_log 后，报错信息里的敏感内容被隐藏掉了（目前是用问号替代）。TiDB 日志中会把敏感信息隐藏掉，以此规避数据安全风险。

4.6.6.2 TiKV 组件日志脱敏

TiKV 侧的日志脱敏需要将security.redact-info-log 的值设为 true。该配置项值默认为 false，即关闭脱敏。

4.6.6.3 PD 组件日志脱敏

PD 侧的日志脱敏需要将security.redact-info-log 的值设为 true。该配置项值默认为 false，即关闭脱敏。

4.6.6.4 TiFlash 组件日志脱敏

TiFlash 侧的日志脱敏需要将tiflash-server 中security.redact_info_log 配置项的值以及tiflash-learner 中security <-> .redact-info-log 配置项的值均设为 true。两配置项默认值均为 false，即关闭脱敏。

4.7 权限

4.7.1 与 MySQL 安全特性差异

TiDB 支持与 MySQL 5.7 类似的安全特性，同时TiDB 还支持 MySQL 8.0 的部分安全特性。TiDB 的安全特性在实现上与 MySQL 存在差异。

4.7.1.1 不支持的安全功能特性

- 不支持列级别权限设置。
- 不支持权限属性 max_questions，max_updated 以及 max_user_connections。
- 不支持密码修改验证策略，修改密码时需要验证当前密码。
- 不支持双密码策略。
- 不支持随机密码生成策略。
- 不支持多因素身份验证。

4.7.1.2 与 MySQL 有差异的安全特性详细说明

4.7.1.2.1 密码过期策略

针对密码过期策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 和 8.0 支持密码过期策略管理功能。
- TiDB 从 v6.5.0 起支持密码过期策略管理功能。

TiDB 的密码过期策略功能与 MySQL 保持一致，但是在密码过期处理机制上存在以下差异：

- MySQL 5.7 和 8.0 在密码过期后是否将客户端的连接限制为“沙盒模式”，由客户端和服务端设置的组合确定。
- TiDB 在密码过期后是否将客户端的连接限制为“沙盒模式”，仅由TiDB 配置文件中的 `[security]` 部分的`disconnect-on-expired-password` 选项确定。

4.7.1.2.2 密码复杂度策略

针对密码复杂度策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 以 validate_password 插件的形式实现了密码复杂度策略管理功能。
- MySQL 8.0 重新以 validate_password 组件的形式实现了密码复杂度策略管理功能。
- TiDB 从 v6.5.0 起内置实现了密码复杂度策略管理功能。

因此，功能实现上存在以下差异：

- 密码复杂度策略功能如何启用：
  - MySQL 5.7 以 validate_password 插件的形式实现，需要进行插件的安装以启用密码复杂度策略管理。
  - MySQL 8.0 以 validate_password 组件的形式实现，需要进行组件的安装以启用密码复杂度策略管理。
  - TiDB 内置实现了密码复杂度策略管理，支持通过系统变量`validate_password.enable` 启用密码复杂度策略管理。
- 密码字典功能：
  - MySQL 5.7 通过变量 `validate_password_dictionary_file` 指定一个文件路径，在文件中写入密码中不允许包含的单词。
  - MySQL 8.0 通过变量 `validate_password.dictionary_file` 指定一个文件路径，在文件中写入密码中不允许包含的单词。
  - TiDB 通过变量`validate_password.dictionary` 指定一个字符串，在该字符串中写入密码中不允许包含的单词。

4.7.1.2.3 密码连续错误限制登录策略

针对密码连续错误限制登录策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 不支持密码连续错误限制登录策略管理功能。
- MySQL 8.0 支持密码连续错误限制登录策略管理功能。
- TiDB 从 v6.5.0 起支持密码连续错误限制登录策略管理功能。

因为用户的失败尝试次数和锁定状态需要做到全局一致，而TiDB 是分布式数据库，不能像 MySQL 在服务端的内存中记录失败尝试次数和锁定状态，所以实现机制存在以下差异：

- 用户未被自动锁定，失败尝试次数的计数重置场景：
  - MySQL 8.0：
    * 服务器重启时，所有用户失败尝试次数的计数都会被重置。
    * 执行 FLUSH PRIVILEGES 时，所有用户失败尝试次数的计数都会被重置。
    * 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。
    * 该用户登录成功时。
  - TiDB：
    * 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。
    * 该用户登录成功时。
- 账户被自动锁定后的解锁场景：
  - MySQL 8.0：
    * 服务器重启时，所有用户的自动锁定标识都会被重置。
    * 执行 FLUSH PRIVILEGES 时，所有用户的自动锁定标识都会被重置。
    * 该用户锁定时间结束，这种情况下，用户的自动锁定标识将在下次登录尝试时重置。
    * 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。
  - TiDB：
    * 该用户锁定时间结束，这种情况下，用户的自动锁定标识将在下次登录尝试时重置。
    * 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。

4.7.1.2.4 密码重用策略

针对密码重用策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 不支持密码重用策略管理功能。
- MySQL 8.0 支持密码重用策略管理功能。
- TiDB 从 v6.5.0 起支持密码重用策略管理功能。

TiDB 的密码重用策略功能与 MySQL 一致，在实现密码重用策略时都增加了系统表 mysql.password_history，但 TiDB 与 MySQL 在删除 mysql.user 系统表中不存在的用户时存在以下差异：

- 场景：没有正确创建用户（例如：user01 ），而通过INSERT INTO mysql.password_history VALUES (...) 命令直接向 mysql.password_history 系统表中添加一条 user01 的记录，此时系统表 mysql.user 中没有 user01 的记录。对该用户执行 DROP USER 操作时，TiDB 和 MySQL 状态不一致。
- 差异点：
  - MySQL：执行DROP USER user01 时，在mysql.user 和mysql.password_history 系统表中匹配user01，若在两个系统表或其中一个系统表中匹配成功，则 DROP USER 操作可以正常执行，不会报错。
  - TiDB：执行 DROP USER user01 时，只在 mysql.user 系统表中匹配 user01，若没有匹配成功，则 DROP USER 操作执行失败， 返回报错。此时如果需要成功执行 DROP USER user01 操作， 删除 mysql.password_history 中 user01 的记录，请使用 DROP USER IF EXISTS user01。

4.7.1.3 可用的身份验证插件

TiDB 支持多种身份验证方式。通过使用CREATE USER 语句和ALTER USER 语句，即可创建新用户或更改TiDB 权限系统内的已有用户。TiDB 身份验证方式与 MySQL 兼容，其名称与 MySQL 保持一致。

TiDB 目前支持的身份验证方式可在以下的表格中查找到。服务器和客户端建立连接时，如要指定服务器对外通告的默认验证方式，可通过default_authentication_plugin 变量进行设置。tidb_sm3_password 为仅在 TiDB 支持的 SM3 身份验证方式，使用该方式登录的用户需要使用TiDB-JDBC。tidb_auth_token 为仅用于TiDB Cloud 内部的基于JSON Web Token (JWT) 的认证方式。

针对 TLS 身份验证，TiDB 目前采用不同的配置方案。具体情况请参见为 TiDB 客户端服务端间通信开启加密传输。

身份验证方式	支持
mysql_native_password	是
sha256_password	否
caching_sha2_password	是（5.2.0 版本起）
auth_socket	是（5.3.0 版本起）
tidb_sm3_password	是（6.3.0 版本起）
tidb_auth_token	是（6.4.0 版本起）
authentication_ldap_sasl	是（7.1.0 版本起）
authentication_ldap_simple	是（7.1.0 版本起）
TLS 证书	是
LDAP	是（7.1.0 版本起）
PAM	否
ed25519 (MariaDB)	否
GSSAPI (MariaDB)	否
FIDO	否

4.7.2 权限管理

TiDB 支持 MySQL 5.7 的权限管理系统，包括 MySQL 的语法和权限类型。同时TiDB 还支持 MySQL 8.0 的以下特性：

- 从 TiDB 3.0 开始，支持 SQL 角色。
- 从 TiDB 5.1 开始，支持动态权限。

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。

4.7.2.1 权限相关操作

4.7.2.1.1 授予权限

授予 xxx 用户对数据库 test 的读权限：

```
GRANT SELECT ON test.* TO 'xxx'@'%';
```

为 xxx 用户授予所有数据库，全部权限：

```
GRANT ALL PRIVILEGES ON *.* TO 'xxx'@'%';
```

默认情况下，如果指定的用户不存在，GRANT 语句将报错。该行为受 SQL Mode 中的 NO_AUTO_CREATE_USER 控制。

```
SET sql_mode=DEFAULT;
Query OK, 0 rows affected (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)

SELECT * FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)

GRANT ALL PRIVILEGES ON test.* TO 'idontexist';
ERROR 1105 (HY000): You are not allowed to create a user with GRANT

SELECT user,host,authentication_string FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)
```

在下面的例子中，由于没有将 SQL Mode 设置为 NO_AUTO_CREATE_USER，用户 idontexist 会被自动创建且密码为空。不推荐使用这种方式，因为会带来安全风险：如果用户名拼写错误，会导致新用户被创建且密码为空。

```
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 @@sql_mode;
+--
    <->
    <-> ----------------------------------------------------------------------------------------------------
    <->
| @@sql_mode
    <->
```

```
    <-->|
+--
    <-->
    <-->
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
    <--> NO_ENGINE_SUBSTITUTION |
+--
    <-->
    <-->
1 row in set (0.00 sec)

SELECT * FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)

GRANT ALL PRIVILEGES ON test.* TO 'idontexist';
Query OK, 1 row affected (0.05 sec)

SELECT user,host,authentication_string FROM mysql.user WHERE user='idontexist';
+------------+------+-----------------------------------------+
| user       | host | authentication_string |
+------------+------+-----------------------------------------+
| idontexist | %    |                       |
+------------+------+-----------------------------------------+
1 row in set (0.01 sec)
```

GRANT 还可以模糊匹配地授予用户数据库的权限：

```
GRANT ALL PRIVILEGES ON `te%`.* TO genius;
```

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

```
SELECT user,host,db FROM mysql.db WHERE user='genius';
```

```
+--------|------|---------+
| user   | host | db  |
+--------|------|---------+
| genius | %    | te% |
+--------|------|---------+
1 row in set (0.00 sec)
```

这个例子中通过% 模糊匹配，所有 te 开头的数据库，都被授予了权限。

4.7.2.1.2 收回权限

REVOKE 语句与 GRANT 对应：

```
REVOKE ALL PRIVILEGES ON `test`.* FROM 'genius'@'localhost';
```

> 注意：
>
> REVOKE 收回权限时只做精确匹配，若找不到记录则报错。而 GRANT 授予权限时可以使用模糊匹配。

```
REVOKE ALL PRIVILEGES ON `te%`.* FROM 'genius'@'%';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'genius' on host '%'
```

关于模糊匹配和转义，字符串和identifier：

```
GRANT ALL PRIVILEGES ON `te\%`.* TO 'genius'@'localhost';
```

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

上述例子是精确匹配名为 te% 的数据库，注意使用\ 转义字符。

以单引号包含的部分，是一个字符串。以反引号包含的部分，是一个identifier。注意下面的区别：

```
GRANT ALL PRIVILEGES ON 'test'.* TO 'genius'@'localhost';
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near ''test'.* to 'genius'@'localhost'' at line 1
```

```
GRANT ALL PRIVILEGES ON `test`.* TO 'genius'@'localhost';
```

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

如果想将一些特殊的关键字做为表名，可以用反引号包含起来。比如：

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

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

4.7.2.1.3 查看为用户分配的权限

SHOW GRANTS 语句可以查看为用户分配了哪些权限。例如：

查看当前用户的权限：

```
SHOW GRANTS;
```

```
+------------------------------------------------------------+
| Grants for User                                            |
+------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION |
+------------------------------------------------------------+
```

或者：

```
SHOW GRANTS FOR CURRENT_USER();
```

查看某个特定用户的权限：

```
SHOW GRANTS FOR 'user'@'host';
```

例如，创建一个用户 `rw_user@192.168.%` 并为其授予 `test.write_table` 表的写权限，和全局读权限。

```
CREATE USER `rw_user`@`192.168.%`;
GRANT SELECT ON *.* TO `rw_user`@`192.168.%`;
GRANT INSERT, UPDATE ON `test`.`write_table` TO `rw_user`@`192.168.%`;
```

查看用户 `rw_user@192.168.%` 的权限。

```
SHOW GRANTS FOR `rw_user`@`192.168.%`;
```

```
+----------------------------------------------------------------------+
| Grants for rw_user@192.168.%                                         |
+----------------------------------------------------------------------+
| GRANT Select ON *.* TO 'rw_user'@'192.168.%'                         |
| GRANT Insert,Update ON test.write_table TO 'rw_user'@'192.168.%' |
+----------------------------------------------------------------------+
```

4.7.2.1.4 动态权限

从 v5.1 开始，TiDB 支持 MySQL 8.0 中的动态权限特性。动态权限用于限制 SUPER 权限，实现对某些操作更细粒度的访问。例如，系统管理员可以使用动态权限来创建一个只能执行 BACKUP 和 RESTORE 操作的用户帐户。

动态权限包括：

- BACKUP_ADMIN
- RESTORE_ADMIN
- SYSTEM_USER
- SYSTEM_VARIABLES_ADMIN
- ROLE_ADMIN
- CONNECTION_ADMIN
- PLACEMENT_ADMIN 允许创建、删除和修改放置策略(placement policy)。
- DASHBOARD_CLIENT 允许登录TiDB Dashboard。

- RESTRICTED_TABLES_ADMIN 允许在 SEM 打开的情况下查看系统表。
- RESTRICTED_STATUS_ADMIN 允许在 SEM 打开的情况下查看SHOW [GLOBAL|SESSION] STATUS 中的状态变量。
- RESTRICTED_VARIABLES_ADMIN 允许在 SEM 打开的情况下查看所有系统变量。
- RESTRICTED_USER_ADMIN 不允许在 SEM 打开的情况下使用 SUPER 用户撤销访问权限。
- RESTRICTED_CONNECTION_ADMIN 允许 KILL 属于 RESTRICTED_USER_ADMIN 用户的连接。该权限对 KILL 和 KILL TIDB 语句生效。
- RESTRICTED_REPLICA_WRITER_ADMIN 允许权限拥有者在TiDB 集群开启了只读模式的情况下不受影响地执行写入或更新操作，详见tidb_restricted_read_only 配置项。

若要查看全部的动态权限，请执行 SHOW PRIVILEGES 语句。由于用户可使用插件来添加新的权限，因此可分配的权限列表可能因用户的TiDB 安装情况而异。

4.7.2.2 SUPER 权限

- 拥有 SUPER 权限的用户能完成几乎所有的操作，默认情况下只有 root 用户拥有该权限。请谨慎向其它用户授予 SUPER 权限。
- SUPER 权限在 MySQL 8.0 中被认为是过时的，可以通过动态权限替代 SUPER 权限进行更细粒度的权限控制。

4.7.2.3 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。例如：

```
SELECT * FROM INFORMATION_SCHEMA.USER_PRIVILEGES WHERE grantee = "'root'@'%'";
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | Select                  | YES          |
| 'root'@'%' | def           | Insert                  | YES          |
| 'root'@'%' | def           | Update                  | YES          |
| 'root'@'%' | def           | Delete                  | YES          |
| 'root'@'%' | def           | Create                  | YES          |
| 'root'@'%' | def           | Drop                    | YES          |
| 'root'@'%' | def           | Process                 | YES          |
| 'root'@'%' | def           | References              | YES          |
| 'root'@'%' | def           | Alter                   | YES          |
| 'root'@'%' | def           | Show Databases          | YES          |
| 'root'@'%' | def           | Super                   | YES          |
| 'root'@'%' | def           | Execute                 | YES          |
| 'root'@'%' | def           | Index                   | YES          |
| 'root'@'%' | def           | Create User             | YES          |
| 'root'@'%' | def           | Create Tablespace       | YES          |
| 'root'@'%' | def           | Trigger                 | YES          |
| 'root'@'%' | def           | Create View             | YES          |
| 'root'@'%' | def           | Show View               | YES          |
| 'root'@'%' | def           | Create Role             | YES          |
```

```
| 'root'@'%'  | def           | Drop Role                | YES          |
| 'root'@'%'  | def           | CREATE TEMPORARY TABLES  | YES          |
| 'root'@'%'  | def           | LOCK TABLES              | YES          |
| 'root'@'%'  | def           | CREATE ROUTINE           | YES          |
| 'root'@'%'  | def           | ALTER ROUTINE            | YES          |
| 'root'@'%'  | def           | EVENT                    | YES          |
| 'root'@'%'  | def           | SHUTDOWN                 | YES          |
| 'root'@'%'  | def           | RELOAD                   | YES          |
| 'root'@'%'  | def           | FILE                     | YES          |
| 'root'@'%'  | def           | CONFIG                   | YES          |
| 'root'@'%'  | def           | REPLICATION CLIENT       | YES          |
| 'root'@'%'  | def           | REPLICATION SLAVE        | YES          |
+-------------+---------------+--------------------------+------------------------------+
31 rows in set (0.00 sec)
```

4.7.2.3.1 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。

4.7.2.3.2 BACKUP

需要拥有 SUPER 或者 BACKUP_ADMIN 权限。

4.7.2.3.3 CREATE DATABASE

需要拥有全局 CREATE 权限。

4.7.2.3.4 CREATE INDEX

需要对所操作的表拥有 INDEX 权限。

4.7.2.3.5 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。

4.7.2.3.6 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。

4.7.2.3.7 DROP DATABASE

需要对数据库拥有 `DROP` 权限。

4.7.2.3.8 DROP INDEX

需要对所操作的表拥有 `INDEX` 权限。

4.7.2.3.9 DROP TABLES

需要对所操作的表拥有 `DROP` 权限。

4.7.2.3.10 LOAD DATA

`LOAD DATA` 需要对所操作的表拥有 `INSERT` 权限。执行 `REPLACE INTO` 语句还需要对所操作的表拥有 `DELETE` 权限。

4.7.2.3.11 TRUNCATE TABLE

需要对所操作的表拥有 `DROP` 权限。

4.7.2.3.12 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。

4.7.2.3.13 ANALYZE TABLE

需要对所操作的表拥有 `INSERT` 和 `SELECT` 权限。

4.7.2.3.14 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。

SHOW PROCESSLIST 需要 SUPER 权限来显示属于其他用户的连接。

4.7.2.3.15 CREATE ROLE/USER

CREATE ROLE 需要 CREATE ROLE 权限。

CREATE USER 需要 CREATE USER 权限

4.7.2.3.16 DROP ROLE/USER

DROP ROLE 需要 DROP ROLE 权限。

DROP USER 需要 CREATE USER 权限

4.7.2.3.17 ALTER USER

ALTER USER 需要 CREATE USER 权限。

4.7.2.3.18 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 或者 ROLE_ADMIN 权限。

4.7.2.3.19 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 或者 ROLE_ADMIN 权限。

4.7.2.3.20 SET GLOBAL

使用 SET GLOBAL 设置全局变量需要拥有 SUPER 或者 SYSTEM_VARIABLES_ADMIN 权限。

4.7.2.3.21 ADMIN

需要拥有 SUPER 权限。

4.7.2.3.22 SET DEFAULT ROLE

需要拥有 SUPER 权限。

4.7.2.3.23 KILL

使用 KILL 终止其他用户的会话需要拥有 SUPER 或者 CONNECTION_ADMIN 权限。

4.7.2.3.24 CREATE RESOURCE GROUP

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

4.7.2.3.25 ALTER RESOURCE GROUP

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

4.7.2.3.26 DROP RESOURCE GROUP

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

4.7.2.3.27 CALIBRATE RESOURCE

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

4.7.2.4 权限系统的实现

4.7.2.4.1 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';
```

4.7.2.4.2 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 authentication_string。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。

4.7.2.4.3 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。

4.7.2.4.4 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;
```

4.7.3 TiDB 用户账户管理

本文档主要介绍如何管理TiDB 用户账户。

要快速了解TiDB 如何进行认证与赋权并创建与管理用户账户，建议先观看下面的培训视频（时长 22 分钟）。注意本视频只作为学习参考，如需了解具体的用户账户管理方法，请参考本文档的内容。

4.7.3.1 用户名和密码

TiDB 将用户账户存储在 mysql.user 系统表里面。每个账户由用户名和host 作为标识。每个账户可以设置一个密码。每个用户名最长为 32 个字符。

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
```

4.7.3.2 添加用户

添加用户有两种方式：

- 通过标准的用户管理的 SQL 语句创建用户以及授予权限，比如 CREATE USER 和 GRANT。
- 直接通过 INSERT、UPDATE 和 DELETE 操作授权表。不推荐使用这种方式添加用户，因为容易导致修改不完整。

除以上两种方法外，你还可以使用第三方图形化界面工具来添加用户。

```
CREATE USER [IF NOT EXISTS] user [IDENTIFIED BY 'auth_string'];
```

设置登录密码后，auth_string 会被TiDB 经过加密存储在 mysql.user 表中。

```
CREATE USER 'test'@'127.0.0.1' IDENTIFIED BY 'xxx';
```

TiDB 的用户账户名由一个用户名和一个主机名组成。账户名的语法为 'user_name'@'host_name'。

- user_name 大小写敏感。
- host_name 可以是一个主机名或IP 地址。主机名或IP 地址中允许使用通配符% 和 _。例如，名为 '%' 的主机名可以匹配所有主机，'192.168.1.%' 可以匹配子网中的所有主机。

host 支持模糊匹配，比如：

```
CREATE USER 'test'@'192.168.10.%';
```

允许 test 用户从 192.168.10 子网的任何一个主机登录。

如果没有指定host，则默认是所有IP 均可登录。如果没有指定密码，默认为空：

```
CREATE USER 'test';
```

等价于：

```
CREATE USER 'test'@'%' IDENTIFIED BY '';
```

为一个不存在的用户授权时， 是否会自动创建用户的行为受 sql_mode 影响。如果 sql_mode 中包含 NO_AUTO_CREATE_USER，则 GRANT 不会自动创建用户并报错。

假设 sql_mode 不包含 NO_AUTO_CREATE_USER，下面的例子用 CREATE USER 和 GRANT 语句创建了四个账户：

```
CREATE USER 'finley'@'localhost' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'localhost' WITH GRANT OPTION;
```

```
CREATE USER 'finley'@'%' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'%' WITH GRANT OPTION;
```

```
CREATE USER 'admin'@'localhost' IDENTIFIED BY 'admin_pass';
```

```
GRANT RELOAD,PROCESS ON *.* TO 'admin'@'localhost';
```

```
CREATE USER 'dummy'@'localhost';
```

使用 SHOW GRANTS 可以看到为一个用户授予的权限：

```
SHOW GRANTS FOR 'admin'@'localhost';
```

```
+-------------------------------------------------------+
| Grants for admin@localhost                            |
+-------------------------------------------------------+
| GRANT RELOAD, PROCESS ON *.* TO 'admin'@'localhost' |
+-------------------------------------------------------+
```

4.7.3.3 删除用户

使用 DROP USER 语句可以删除用户，例如：

```
DROP USER 'test'@'localhost';
```

这个操作会清除用户在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录。

4.7.3.4 保留用户账户

TiDB 在数据库初始化时会生成一个 'root'@'%' 的默认账户。

4.7.3.5 设置资源限制

暂不支持。

4.7.3.6 设置密码

TiDB 将密码存在 mysql.user 系统数据库里面。只有拥有 CREATE USER 权限，或者拥有 mysql 数据库权限（INSERT 权限用于创建，UPDATE 权限用于更新）的用户才能够设置或修改密码。

- 在 CREATE USER 创建用户时通过 IDENTIFIED BY 指定密码：

```
CREATE USER 'test'@'localhost' IDENTIFIED BY 'mypass';
```

- 为一个已存在的账户修改密码，可以通过 SET PASSWORD FOR 或者 ALTER USER 语句完成：

```
SET PASSWORD FOR 'root'@'%' = 'xxx';
```

或者：

```
ALTER USER 'test'@'localhost' IDENTIFIED BY 'mypass';
```

4.7.3.7 忘记 root 密码

1. 修改TiDB 配置文件：
    1. 登录其中一台tidb-server 实例所在的机器。
    2. 进入TiDB 节点的部署目录下的 conf 目录，找到 tidb.toml 配置文件。
    3. 在配置文件的 security 部分添加配置项 skip-grant-table。如无 security 部分，则将以下两行内容添加至tidb.toml 配置文件尾部：

    ```
    [security]
    skip-grant-table = true
    ```

2. 终止该tidb-server 的进程：
    1. 查看tidb-server 的进程：

    ```
    ps aux | grep tidb-server
    ```

    2. 找到tidb-server 对应的进程ID (PID) 并使用 kill 命令停掉该进程：

    ```
    kill -9 <pid>
    ```

3. 使用修改之后的配置启动TiDB：

> 注意：
> 设置 skip-grant-table 之后，启动TiDB 进程会增加操作系统用户检查，只有操作系统的 root 用户才能启动TiDB 进程。

    1. 进入TiDB 节点部署目录下的 scripts 目录。

2. 切换到操作系统 root 账号。
3. 在前台执行目录中的 run_tidb.sh 脚本。
4. 在新的终端窗口中使用 root 登录后修改密码：

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

4. 停止运行 run_tidb.sh 脚本，并去掉第 1 步中在TiDB 配置文件中添加的内容，等待tidb-server 自启动。

4.7.3.8 FLUSH PRIVILEGES

用户以及权限相关的信息都存储在TiKV 服务器中，TiDB 在进程内部会缓存这些信息。一般通过 CREATE USER，GRANT 等语句来修改相关信息时，可在整个集群迅速生效。如果遇到网络或者其它因素影响，由于TiDB 会周期性地更新缓存信息，正常情况下，最多 15 分钟左右生效。

如果授权表已被直接修改，则不会通知TiDB 节点更新缓存，运行如下命令可使改动立即生效：

```
FLUSH PRIVILEGES;
```

详情参见权限管理。

4.7.4 TiDB 密码管理

为了保护用户密码的安全，从TiDB v6.5.0 开始支持密码管理能力：

- 密码复杂度策略：要求用户设置强密码，以防止出现空密码、弱密码。
- 密码过期策略：要求用户定期修改密码。
- 密码重用策略：限制用户重复使用旧密码。
- 密码连续错误限制登录策略：连续多次密码错误导致登录失败后，临时锁定用户，限制该用户继续尝试登录。

4.7.4.1 TiDB 身份验证凭据存储

密码作为一种用户身份凭据，在用户登录到服务端时用于身份验证，确保用户身份的合法性。本文中描述的密码是指由TiDB 生成、存储、验证的内部凭据，TiDB 将用户密码被存储到 mysql.user 系统表中，以下身份验证插件涉及本文的密码管理功能：

- mysql_native_password
- caching_sha2_password
- tidb_sm3_password

有关TiDB 支持身份验证插件的更多信息，请查看可用的身份验证插件。

4.7.4.2 密码复杂度策略

在TiDB 中，密码复杂度检查默认未开启。通过配置密码复杂度相关的系统变量，你可以开启密码复杂度检查，并确保为账户设置的密码符合密码复杂度策略。

密码复杂度策略支持以下功能：

- 对采用明文方式设置用户密码的 SQL 语句（包括 CREATE USER、ALTER USER、SET PASSWORD），系统会根据密码复杂度策略检查该密码，如果该密码不符合要求，则拒绝该密码。
- 可以使用 SQL 函数 VALIDATE_PASSWORD_STRENGTH() 评估给定密码的强度。

> 注意：
>
> - 对于 CREATE USER 语句，即使该账户最初被锁定，也必须提供满足密码复杂度策略的密码，否则将账户解锁后，该账户可以使用不符合密码复杂度策略的密码访问TiDB。
> - 对密码复杂度策略的变更不影响已存在的密码，只会对新设置的密码产生影响。

通过以下 SQL 语句，你可以查看所有密码复杂度策略相关的系统变量：

```
mysql> SHOW VARIABLES LIKE 'validate_password.%';
+--------------------------------------+--------+
| Variable_name                        | Value  |
+--------------------------------------+--------+
| validate_password.check_user_name    | ON     |
| validate_password.dictionary         |        |
| validate_password.enable             | OFF    |
| validate_password.length             | 8      |
| validate_password.mixed_case_count   | 1      |
| validate_password.number_count       | 1      |
| validate_password.policy             | MEDIUM |
| validate_password.special_char_count | 1      |
+--------------------------------------+--------+
8 rows in set (0.00 sec)
```

关于这些变量的详细解释，请查阅系统变量文档。

4.7.4.2.1 配置密码复杂度策略

密码复杂度策略相关的系统变量的配置方式如下：

开启密码复杂度策略检查：

```
SET GLOBAL validate_password.enable = ON;
```

设置不允许密码与当前用户名相同：

```
SET GLOBAL validate_password.check_user_name = ON;
```

设置密码复杂度的检查等级为 LOW：

```
SET GLOBAL validate_password.policy = LOW;
```

设置密码最小长度为 10：

```
SET GLOBAL validate_password.length = 10;
```

设置密码中至少含有 2 个数字，至少含有 1 个大写和小写字符，至少含有 1 个特殊字符：

```
SET  GLOBAL  validate_password.number_count  =  2;
SET GLOBAL validate_password.mixed_case_count = 1;
SET GLOBAL validate_password.special_char_count = 1;
```

设置密码字典功能，要求密码中不允许包含 mysql 或 abcd：

```
SET GLOBAL validate_password.dictionary = 'mysql;abcd';
```

> 注意：
>
> - validate_password.dictionary 是一个长字符串，长度不超过 1024，字符串内容可包含一个或多个在密码中不允许出现的单词，每个单词之间采用英文分号（;）分隔。
> - 密码字典功能进行单词比较时，不区字符分大小写。

4.7.4.2.2 密码复杂度检查示例

配置系统变量 validate_password.enable = ON 后，TiDB 将开启密码复杂度检查。以下为一些典型的检查示例：

按照默认密码复杂度策略，检测用户明文密码，若设置的密码不符合复杂度策略要求，则设置失败。

```
ALTER USER 'test'@'localhost' IDENTIFIED BY 'abc';
ERROR 1819 (HY000): Require Password Length: 8
```

TiDB 进行密码复杂度检查时，不检查散列后的密码。

```
ALTER USER 'test'@'localhost' IDENTIFIED WITH mysql_native_password AS '*0
    <-> D3CED9BEC10A777AEC23CCC353A8C08A633045E';
Query OK, 0 rows affected (0.01 sec)
```

创建一个最初被锁定的账户时，也必须设置符合密码复杂度策略的密码，否则创建失败。

```
CREATE USER 'user02'@'localhost' ACCOUNT LOCK;
ERROR 1819 (HY000): Require Password Length: 8
```

4.7.4.2.3 密码强度评估函数

使用 VALIDATE_PASSWORD_STRENGTH() 函数评估给定密码的强度，该函数接受一个密码参数，并返回一个从 0（弱）到 100（强）的整数。

> **注意：**
>
> 密码强度是基于当前已配置的密码复杂度策略进行评估的，密码复杂度配置改变后，同一个密码的评估结果可能与之前不同。

VALIDATE_PASSWORD_STRENGTH() 函数使用示例如下：

```
SELECT VALIDATE_PASSWORD_STRENGTH('weak');
+--------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('weak') |
+--------------------------------------+
|                                   25 |
+--------------------------------------+
1 row in set (0.01 sec)

SELECT VALIDATE_PASSWORD_STRENGTH('lessweak$_@123');
+-----------------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('lessweak$_@123') |
+-----------------------------------------------+
|                                            50 |
+-----------------------------------------------+
1 row in set (0.01 sec)

SELECT VALIDATE_PASSWORD_STRENGTH('NOTweak$_@123!');
+-----------------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('NOTweak$_@123!') |
+-----------------------------------------------+
|                                           100 |
+-----------------------------------------------+
1 row in set (0.01 sec)
```

4.7.4.3 密码过期策略

TiDB 支持通过设置密码过期策略，要求用户定期修改密码，从而提高密码的安全性。可以手动将指定账户的密码设置为过期，也可以建立密码自动过期策略。自动过期策略分为全局级别和账户级别，管理员可以在全局级别建立密码过期策略，也可以使用账户级别密码过期策略覆盖全局级别策略。设置密码过期策略的权限要求如下：

- 具有 SUPER 或者 CREATE USER 权限的数据库管理员可以手动设置密码过期。

- 具有 SUPER 或者 CREATE USER 权限的数据库管理员可以设置账户级别自动密码过期策略。
- 具有 SUPER 或者 SYSTEM_VARIABLES_ADMIN 权限的数据库管理员可以设置全局级别自动密码过期策略。

4.7.4.3.1 手动密码过期

要手动设置账户密码过期，请使用 CREATE USER 或 ALTER USER 语句。

```
ALTER USER 'test'@'localhost' PASSWORD EXPIRE;
```

当账户密码被管理员手动设置过期后，必须修改该账户密码才能解除密码过期，不支持取消手动过期。

对于通过 CREATE ROLE 语句创建的角色，由于角色不需要设置密码，所以该角色对应的密码字段为空，此时对应的 password_expired 属性为 'Y'，即该角色的密码处于手动过期状态。如此设计的目的是防止出现角色锁定状态被解除后，该角色以空密码登录到TiDB，当该角色被 ALTER USER ... ACCOUNT UNLOCK 命令解锁后，此时该角色处于可登录的状态，但是密码为空；因此TiDB 通过 password_expired 属性，使得角色的密码处于手动过期状态，从而强制要求为该角色设置一个有效的密码。

```
CREATE ROLE testrole;
Query OK, 0 rows affected (0.01 sec)

SELECT user,password_expired,Account_locked FROM mysql.user WHERE user = 'testrole';
+----------+------------------+----------------+
| user     | password_expired | Account_locked |
+----------+------------------+----------------+
| testrole | Y                | Y              |
+----------+------------------+----------------+
1 row in set (0.02 sec)
```

4.7.4.3.2 自动密码过期

自动密码过期是基于密码使用期限和密码被允许的生存期来判断的。

- 密码使用期限：从最近一次密码更改日期到当前日期的时间间隔。系统表 mysql.user 中会记录最近一次修改密码的时间。
- 密码被允许的生存期：一个密码被设置后，其可以正常用于登录TiDB 的天数。

如果密码使用期限大于其被允许的生存期，服务器会自动将密码视为已过期。

TiDB 支持在全局级别和账户级别设置自动密码过期：

- 全局级别自动密码过期

  你可以设置系统变量default_password_lifetime 来控制密码生存期。该变量默认值为 0，表示禁用自动密码过期。如果设置该变量的值为正整数 N，则表示允许的密码生存期为N 天，即必须在N 天之内更改密码。

  全局自动密码过期策略适用于所有未设置账户级别覆盖的账户。

  以下示例建立全局自动密码过期策略，密码有效期为 180 天：

```
SET GLOBAL default_password_lifetime = 180;
```

- 账户级别自动密码过期

  要为个人账户建立自动密码过期策略，请使用 CREATE USER 或 ALTER USER 语句的 PASSWORD EXPIRE 选项。

  以下示例要求用户密码每 90 天更改一次：

```
CREATE USER 'test'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;
ALTER USER 'test'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;
```

  在账户级别禁用自动密码过期策略：

```
CREATE USER 'test'@'localhost' PASSWORD EXPIRE NEVER;
ALTER USER 'test'@'localhost' PASSWORD EXPIRE NEVER;
```

  移除指定账户的账户级别自动密码过期策略，使其遵循于全局自动密码过期策略：

```
CREATE USER 'test'@'localhost' PASSWORD EXPIRE DEFAULT;
ALTER USER 'test'@'localhost' PASSWORD EXPIRE DEFAULT;
```

4.7.4.3.3 密码过期策略检查机制

当客户端连接成功后，服务端将按顺序进行以下检查，判断账号密码是否过期：

1. 服务器检查密码是否已手动过期。
2. 若密码没有手动过期，服务器根据自动密码过期策略检查密码使用期限是否大于其允许的生存期。如果是，服务器认为密码已过期。

4.7.4.3.4 密码过期处理机制

TiDB 支持密码过期策略控制。当密码过期后，服务器要么断开客户端的连接，要么将客户端限制为“沙盒模式”。“沙盒模式” 下，TiDB 服务端接受密码过期账户的连接，但是连接成功后只允许该用户执行重置密码的操作。

TiDB 服务端可以控制是否将密码已过期用户的连接限制为 “沙盒模式”。你可以在 TiDB 配置文件中的 [security] 部分，配置disconnect-on-expired-password 选项：

```
[security]
disconnect-on-expired-password = true
```

- 默认情况下，disconnect-on-expired-password 为 true，表示当密码过期后，服务器将直接断开客户端的连接。
- 如果配置 disconnect-on-expired-password 为 false，则服务端处于沙盒模式，服务端允许用户建立连接，但只能执行密码重置操作，密码重置后将允许用户正常执行各类 SQL 语句。

当 disconnect-on-expired-password 为 true 时，TiDB 将拒绝密码已过期用户的连接，此时可以通过如下方法修改密码：

- 普通用户密码过期，可以由管理员用户通过 SQL 语句修改该用户的密码。
- 管理员密码过期，可以由其他管理员用户通过 SQL 语句修改该用户的密码。
- 如果管理员密码过期，且无法寻求其他管理员帮助修改该用户的密码，此时可以采用skip-grant-table 机制修改该用户密码，具体可参看忘记密码流程。

4.7.4.4 密码重用策略

TiDB 支持限制重复使用以前的密码。密码重用策略可以基于密码更改的次数或经过的时间，也可以同时基于两者。密码重用策略分为全局级别和账户级别。你可以在全局级别建立密码重用策略，也可以使用账户级别密码重用策略覆盖全局策略。

TiDB 会记录账户的历史密码，并限制从该历史记录中选择新密码：

- 如果密码重用策略基于密码更改次数，则新密码不得与指定数量的历史密码相同。例如，如果密码的最小更改次数设置为 3，则新密码不能与最近 3 个密码中的任何一个相同。
- 如果密码重用策略基于经过的时间，则新密码不得与历史记录中指定天数内使用过的密码相同。例如，如果密码重用间隔设置为 60，则新密码不能与最近 60 天内使用过的密码相同。

> 注意：
> 空密码不计入密码历史记录，可以随时重复使用。

4.7.4.4.1 全局级别密码重用策略

要在全局范围内建立密码重用策略，请使用password_history 和password_reuse_interval 系统变量。

例如，建立全局密码重用策略，禁止重复使用最近 6 个密码或最近 365 天的密码：

```
SET GLOBAL password_history = 6;
SET GLOBAL password_reuse_interval = 365;
```

全局密码重用策略适用于所有未在账户级别设置过密码重用策略的账户。

4.7.4.4.2 账户级别密码重用策略

要建立账户级别密码重用策略，请使用 CREATE USER 或 ALTER USER 语句的 PASSWORD HISTORY 和 PASSWORD <—> REUSE INTERVAL 选项。

示例：

禁止重复使用最近 5 次使用过的密码：

```
CREATE USER 'test'@'localhost' PASSWORD HISTORY 5;
ALTER USER 'test'@'localhost' PASSWORD HISTORY 5;
```

禁止重复使用最近 365 天内使用过的密码：

```
CREATE USER 'test'@'localhost' PASSWORD REUSE INTERVAL 365 DAY;
ALTER USER 'test'@'localhost' PASSWORD REUSE INTERVAL 365 DAY;
```

如需组合两种类型的重用策略，请一起使用 PASSWORD HISTORY 和 PASSWORD REUSE INTERVAL：

```
CREATE USER 'test'@'localhost'
  PASSWORD HISTORY 5
  PASSWORD REUSE INTERVAL 365 DAY;
ALTER USER 'test'@'localhost'
  PASSWORD HISTORY 5
  PASSWORD REUSE INTERVAL 365 DAY;
```

移除指定账户的账户级别密码重用策略，使其遵循于全局密码重用策略：

```
CREATE USER 'test'@'localhost'
  PASSWORD HISTORY DEFAULT
  PASSWORD REUSE INTERVAL DEFAULT;
ALTER USER 'test'@'localhost'
  PASSWORD HISTORY DEFAULT
  PASSWORD REUSE INTERVAL DEFAULT;
```

> 注意：
>
> - 如果多次设置密码重用策略，则最后一次设置的值生效。
> - PASSWORD HISTORY 和 PASSWORD REUSE INTERVAL 选项的默认值为 0，表示禁用该项重用策略。
> - 在修改用户名时，TiDB 会将 mysql.password_history 系统表中原用户名的历史密码记录迁移到新用户名的记录中。

4.7.4.5 密码连续错误限制登录策略

TiDB 支持限制账户持续尝试登录，防止用户密码被暴力破解。当账户连续登录失败次数过多时，账户将被临时锁定。

> 注意：
>
> - 只支持账户级别的密码连续错误限制登录策略，不支持全局级别的策略。
> - 登录失败是指客户端在连接尝试期间未能提供正确的密码，不包括由于未知用户或网络问题等原因而导致的连接失败。
> - 对用户启用密码连续错误限制登录策略后，将增加该用户登录时的检查步骤，此时会影响该用户登录相关操作的性能，尤其是高并发登录场景。

4.7.4.5.1 配置密码连续错误限制登录策略

每个账户的登录失败次数和锁定时间是可配置的，你可以使用 CREATE USER、ALTER USER 语句的 FAILED_LOGIN_ATTEMPTS 和 PASSWORD_LOCK_TIME 选项。FAILED_LOGIN_ATTEMPTS 和 PASSWORD_LOCK_TIME 选项的可设置值如下：

- FAILED_LOGIN_ATTEMPTS：N。表示连续登录失败 N 次后，账户将被临时锁定。N 取值范围为 0 到 32767。
- PASSWORD_LOCK_TIME：N | UNBOUNDED。N 表示登录失败后，账户将被临时锁定N 天。UNBOUNDED 表明锁定时间无限期，账户必须被手动解锁。N 取值范围为 0 到 32767。

> 注意：
>
> - 允许单条 SQL 语句只设置 FAILED_LOGIN_ATTEMPTS 或 PASSWORD_LOCK_TIME 中的一个选项，这时密码连续错误限制登录策略不会实质生效。
> - 只有当账户的 FAILED_LOGIN_ATTEMPTS 和 PASSWORD_LOCK_TIME 都不为 0 时，系统才会跟踪该账户的失败登录次数并执行临时锁定。

配置密码连续错误限制登录策略的示例如下：

新建一个用户，并配置密码连续错误限制登录策略，当该用户密码连续错误 3 次时，临时锁定 3 天：

```
CREATE USER 'test1'@'localhost' IDENTIFIED BY 'password' FAILED_LOGIN_ATTEMPTS 3
    <-> PASSWORD_LOCK_TIME 3;
```

修改用户的密码连续错误限制登录策略，当该用户密码连续错误 4 次时，无限期锁定，直到账户被手动解锁：

```
ALTER USER 'test2'@'localhost' FAILED_LOGIN_ATTEMPTS 4 PASSWORD_LOCK_TIME UNBOUNDED;
```

关闭用户的密码连续错误限制登录策略：

```
ALTER USER 'test3'@'localhost' FAILED_LOGIN_ATTEMPTS 0 PASSWORD_LOCK_TIME 0;
```

4.7.4.5.2 锁定账户解锁

在以下场景中，TiDB 将重置用户密码连续错误次数的计数：

- 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。
- 该用户登录成功时。

当用户因密码连续多次错误触发账户锁定后，以下情况下可以解锁账户：

- 该用户锁定时间结束，这种情况下，用户的自动锁定标识将在下次登录尝试时重置。
- 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。

注意：

当用户因密码连续失败次数达到设定值而被自动锁定时，如果修改该账户的密码连续错误限制登录策略，需要注意：

- 修改该用户的登录失败次数 FAILED_LOGIN_ATTEMPTS，用户的自动锁定状态不会改变。修改后的连续登录失败次数，将在用户解锁后再次尝试登录时生效。
- 修改该用户的锁定时间 PASSWORD_LOCK_TIME，用户的自动锁定状态不会改变。修改后的用户锁定时间，将在账户再次登录尝试时检查是否满足此时的锁定时间要求，如果锁定时间结束就会解锁该用户。

4.7.5 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。

4.7.5.1 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。

4.7.5.1.1 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。

4.7.5.1.2 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';
```

4.7.5.1.3 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。

4.7.5.1.4 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。

4.7.5.1.5 设置默认启用角色

角色在授予给用户之后，并不会生效；只有在用户启用了某些角色之后，才可以使用角色拥有的权限。

可以对用户设置默认启用的角色；用户在登录时，默认启用的角色会被自动启用。

```
SET DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}
    TO user [, user ]
```

比如将 app_read 和 app_wirte 设置为 rw_user1@localhost 的默认启用角色：

```
SET DEFAULT ROLE app_read, app_write TO 'rw_user1'@'localhost';
```

将 dev1@localhost 的所有角色，设为其默认启用角色：

```
SET DEFAULT ROLE ALL TO 'dev1'@'localhost';
```

关闭 dev1@localhost 的所有默认启用角色：

```
SET DEFAULT ROLE NONE TO 'dev1'@'localhost';
```

需要注意的是，设置为默认启用角色的角色必须已经授予给那个用户。

4.7.5.1.6 在当前 session 启用角色

除了使用 SET DEFAULT ROLE 启用角色外，TiDB 还提供让用户在当前 session 启用某些角色的功能。

```
SET ROLE {
    DEFAULT
  | NONE
  | ALL
  | ALL EXCEPT role [, role ] ...
  | role [, role ] ...
}
```

例如，登录 rw_user1 后，为当前用户启用角色 app_read 和 app_write，仅在当前 session 有效：

```
SET ROLE 'app_read', 'app_write';
```

启用当前用户的默认角色：

```
SET ROLE DEFAULT
```

启用授予给当前用户的所有角色：

```
SET ROLE ALL
```

不启用任何角色：

```
SET ROLE NONE
```

启用除 app_read 外的角色：

```
SET ROLE ALL EXCEPT 'app_read'
```

> **注意：**
>
> 使用 SET ROLE 启用的角色只有在当前 session 才会有效。

4.7.5.1.7 查看当前启用角色

当前用户可以通过 CURRENT_ROLE() 函数查看当前用户启用了哪些角色。

例如，先对 rw_user1'@'localhost 设置默认角色：

```
SET DEFAULT ROLE ALL TO 'rw_user1'@'localhost';
```

用 rw_user1@localhost 登录后：

```
SELECT CURRENT_ROLE();
```

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

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

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

4.7.5.1.8 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。

4.7.5.1.9 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。

4.7.5.1.10 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。

4.7.5.1.11 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。

4.7.5.1.12 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

4.7.6 TiDB 证书鉴权使用指南

TiDB 支持基于证书鉴权的登录方式。采用这种方式，TiDB 对不同用户签发证书，使用加密连接来传输数据，并在用户登录时验证证书。相比 MySQL 用户常用的用户名密码验证方式，与 MySQL 相兼容的证书鉴权方式更安全，因此越来越多的用户使用证书鉴权来代替用户名密码验证。

在TiDB 上使用证书鉴权的登录方法，可能需要进行以下操作：

- 创建安全密钥和证书
- 配置TiDB 和客户端使用的证书
- 配置登录时需要校验的用户证书信息
- 更新和替换证书

本文介绍了如何进行证书鉴权的上述几个操作。

4.7.6.1 创建安全密钥和证书

目前推荐使用 OpenSSL 来生成密钥和证书，生成证书的过程和为 TiDB 客户端服务端间通信开启加密传输过程类似，下面更多演示如何在证书中配置更多需校验的属性字段。

4.7.6.1.1 生成 CA 密钥和证书

1. 执行以下命令生成 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

命令执行后输出以下结果：

```
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
.....................................................................................+++++
e is 65537 (0x010001)
```

2. 执行以下命令生成该密钥对应的证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.pem
```

3. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:TiDB admin
Email Address []:s@pingcap.com
```

> **注意：**
> 以上信息中，: 后的文字为用户输入的信息。

4.7.6.1.2 生成服务端密钥和证书

1. 执行以下命令生成服务端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.pem -out server-req
    <↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiKV
Common Name (e.g. server FQDN or YOUR name) []:TiKV Test Server
Email Address []:k@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成服务端的RSA 密钥：

```
sudo openssl rsa -in server-key.pem -out server-key.pem
```

输出结果如下：

```
writing RSA key
```

4. 使用 CA 证书签名来生成服务端的证书：

```
sudo openssl x509 -req -in server-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    <↪ set_serial 01 -out server-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiKV, CN = TiKV
    <↪ Test Server, emailAddress = k@pingcap.com
Getting CA Private Key
```

> 注意：
> 以上结果中，用户登录时TiDB 将强制检查 `subject` 部分的信息是否一致。

4.7.6.1.3 生成客户端密钥和证书

生成服务端密钥和证书后，需要生成客户端使用的密钥和证书。通常需要为不同的用户生成不同的密钥和证书。

1. 执行以下命令生成客户端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.pem -out client-req
    <↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:tpch-user1
Email Address []:zz@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成客户端RSA 证书：

```
sudo openssl rsa -in client-key.pem -out client-key.pem
```

以上命令的输出结果如下：

```
writing RSA key
```

4. 执行以下命令，使用 CA 证书签名来生成客户端证书：

```
sudo openssl x509 -req -in client-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    <⟶ set_serial 01 -out client-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiDB, CN = tpch-
    <⟶ user1, emailAddress = zz@pingcap.com
Getting CA Private Key
```

> 注意：
> 以上结果中，`subject` 部分后的信息会被用来在 `require` 中配置和要求验证。

4.7.6.1.4 验证证书

执行以下命令验证证书：

```
openssl verify -CAfile ca-cert.pem server-cert.pem client-cert.pem
```

如果验证通过，会显示以下信息：

```
server-cert.pem: OK
client-cert.pem: OK
```

4.7.6.2 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。

4.7.6.2.1 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]
```

4.7.6.2.2 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。

4.7.6.3 配置登录时需要校验的用户证书信息

使用客户端连接TiDB 进行授权配置。先获取需要验证的用户证书信息，再对这些信息进行配置。

4.7.6.3.1 获取用户证书信息

用户证书信息可由`require subject`、`require issuer`、`require san` 和 `require cipher` 来指定，用于检查X.509 certificate attributes。

- `require subject`：指定用户在连接时需要提供客户端证书的 `subject` 内容。指定该选项后，不需要再配置 `require ssl` 或 x509。配置内容对应生成客户端密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

  ```
  openssl x509 -noout -subject -in client-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed
      <↪  's/ = /=/g' | sed 's/^/\//'
  ```

- `require issuer`：指定签发用户证书的 CA 证书的 `subject` 内容。配置内容对应生成 CA 密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

  ```
  openssl x509 -noout -subject -in ca-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed 's/
      <↪  = /=/g' | sed 's/^/\//'
  ```

- `require san`：指定签发用户证书的 CA 证书的 `Subject Alternative Name` 内容。配置内容对应生成客户端证书使用的openssl.cnf 配置文件的 `alt_names` 信息。

  - 可以执行以下命令来获取已生成证书中的 `require san` 项的信息：

    ```
    openssl x509 -noout -extensions subjectAltName -in client.crt
    ```

  - `require san` 目前支持以下 `Subject Alternative Name` 检查项：

    * URI
    * IP
    * DNS
  - 多个检查项可通过逗号连接后进行配置。例如，对用户 u1 进行以下配置：

```
create user 'u1'@'%' require san 'DNS:d1,URI:spiffe://example.org/myservice1,URI:spiffe
    <-> ://example.org/myservice2'
```

    以上配置只允许用户 u1 使用 URI 项为 spiffe://example.org/myservice1 或 spiffe://example.org <-> /myservice2、DNS 项为 d1 的证书登录TiDB。

- require cipher：配置该项检查客户端支持的 cipher method。可以使用以下语句来查看支持的列表：

```
SHOW SESSION STATUS LIKE 'Ssl_cipher_list';
```

4.7.6.3.2 配置用户证书信息

获取用户证书信息（require subject, require issuer、require san 和 require cipher）后，可在创建用户、赋予权限或更改用户时配置用户证书信息。将以下命令中的 <replaceable> 替换为对应的信息。可以选择配置其中一项或多项，使用空格或 and 分隔。

- 可以在创建用户(create user)时配置登录时需要校验的证书信息：

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

- 可以在赋予权限(grant)时配置登录时需要校验的证书信息：

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

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

```
alter user 'u1'@'%' require issuer '<replaceable>' subject '<replaceable>' san '<replaceable
    <-> >' cipher '<replaceable>';
```

配置完成后，用户在登录时TiDB 会验证以下内容：

- 使用 SSL 登录，且证书为服务器配置的 CA 证书所签发
- 证书的 Issuer 信息和权限配置里的信息相匹配
- 证书的 Subject 信息和权限配置里的信息相匹配
- 证书的 Subject Alternative Name 信息和权限配置里的信息相匹配

全部验证通过后用户才能登录，否则会报 ERROR 1045 (28000): Access denied 错误。登录后，可以通过以下命令来查看当前链接是否使用证书登录、TLS 版本和 Cipher 算法。

连接 MySQL 客户端并执行：

```
\s
```

返回结果如下：

```
--------------
mysql  Ver 15.1 Distrib 10.4.10-MariaDB, for Linux (x86_64) using readline 5.1

Connection id:       1
Current database:    test
Current user:        root@127.0.0.1
SSL:                 Cipher in use is TLS_AES_256_GCM_SHA384
```

然后执行：

```
show variables like '%ssl%';
```

返回结果如下：

```
+---------------+-----------------------------------------------------------+
| Variable_name | Value                                                     |
+---------------+-----------------------------------------------------------+
| ssl_cert      | /path/to/server-cert.pem                                  |
| ssl_ca        | /path/to/ca-cert.pem                                      |
| have_ssl      | YES                                                       |
| have_openssl  | YES                                                       |
| ssl_key       | /path/to/server-key.pem                                   |
+---------------+-----------------------------------------------------------+
6 rows in set (0.067 sec)
```

4.7.6.4 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。

4.7.6.4.1 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。

4.7.6.4.2 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。

4.7.6.4.3 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。

4.8 SQL

4.8.1 SQL 语言结构和语法

4.8.1.1 属性

4.8.1.1.1 AUTO_INCREMENT

本文介绍列属性 AUTO_INCREMENT 的基本概念、实现原理、自增相关的特性，以及使用限制。

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

基本概念

AUTO_INCREMENT 是用于自动填充缺省列值的列属性。当 INSERT 语句没有指定 AUTO_INCREMENT 列的具体值时，系统会自动地为该列分配一个值。

出于性能原因，自增编号是系统批量分配给每台TiDB 服务器的值（默认 3 万个值），因此自增编号能保证唯一性，但分配给 INSERT 语句的值仅在单台TiDB 服务器上具有单调性。

> 注意：
>
> 如果要求自增编号在所有TiDB 实例上具有单调性，并且你的TiDB 版本在 v6.5.0 及以上，推荐使用MySQL 兼容模式。

```
CREATE TABLE t(id int PRIMARY KEY AUTO_INCREMENT, c int);
```

```
INSERT INTO t(c) VALUES (1);
INSERT INTO t(c) VALUES (2);
INSERT INTO t(c) VALUES (3), (4), (5);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
+----+---+
5 rows in set (0.01 sec)
```

此外，AUTO_INCREMENT 还支持显式指定列值的插入语句，此时TiDB 会保存显式指定的值：

```
INSERT INTO t(id, c) VALUES (6, 6);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
| 6  | 6 |
```

```
+----+---+
6 rows in set (0.01 sec)
```

以上用法和 MySQL 的 AUTO_INCREMENT 用法一致。但在隐式分配的具体值方面，TiDB 和 MySQL 之间具有较为显著的差异。

实现原理

TiDB 实现 AUTO_INCREMENT 隐式分配的原理是，对于每一个自增列，都使用一个全局可见的键值对用于记录当前已分配的最大ID。由于分布式环境下的节点通信存在一定开销，为了避免写请求放大的问题，每个TiDB 节点在分配ID 时，都申请一段ID 作为缓存，用完之后再去取下一段，而不是每次分配都向存储节点申请。例如，对于以下新建的表：

```
CREATE TABLE t(id int UNIQUE KEY AUTO_INCREMENT, c int);
```

假设集群中有两个TiDB 实例A 和 B，如果向A 和 B 分别对 t 执行一条插入语句：

```
INSERT INTO t (c) VALUES (1)
```

实例A 可能会缓存 [1,30000] 的自增ID，而实例B 则可能缓存 [30001,60000] 的自增ID。各自实例缓存的ID 将随着执行将来的插入语句被作为缺省值，顺序地填充到 AUTO_INCREMENT 列中。

基本特性

唯一性保证

> 警告：
>
> 在集群中有多个TiDB 实例时，如果表结构中有自增ID，建议不要混用显式插入和隐式分配 (即自增列的缺省值和自定义值)，否则可能会破坏隐式分配值的唯一性。

例如在上述示例中，依次执行如下操作：

1. 客户端向实例B 插入一条将 id 设置为 2 的语句 INSERT INTO t VALUES (2, 1)，并执行成功。
2. 客户端向实例A 发送 INSERT 语句 INSERT INTO t (c)(1)，这条语句中没有指定 id 的值，所以会由A 分配。当前A 缓存了 [1, 30000] 这段ID，可能会分配 2 为自增ID 的值，并把本地计数器加 1。而此时数据库中已经存在 id 为 2 的数据，最终返回 Duplicated Error 错误。

单调性保证

TiDB 保证 AUTO_INCREMENT 自增值在单台服务器上单调递增。以下示例在一台服务器上生成连续的 AUTO_INCREMENT 自增值 1-3：

```
CREATE TABLE t (a int PRIMARY KEY AUTO_INCREMENT, b timestamp NOT NULL DEFAULT NOW());
INSERT INTO t (a) VALUES (NULL), (NULL), (NULL);
SELECT * FROM t;
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
+---+---------------------+
| a | b                   |
+---+---------------------+
| 1 | 2020-09-09 20:38:22 |
| 2 | 2020-09-09 20:38:22 |
| 3 | 2020-09-09 20:38:22 |
+---+---------------------+
3 rows in set (0.00 sec)
```

TiDB 能保证自增值的单调性，但并不能保证其连续性。参考以下示例：

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, a VARCHAR(10), cnt INT NOT NULL
    <-> DEFAULT 1, UNIQUE KEY (a));
INSERT INTO t (a) VALUES ('A'), ('B');
SELECT * FROM t;
INSERT INTO t (a) VALUES ('A'), ('C') ON DUPLICATE KEY UPDATE cnt = cnt + 1;
SELECT * FROM t;
```

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

Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

+----+------+---------+
| id | a    | cnt |
+----+------+---------+
|  1 | A    |   1 |
|  2 | B    |   1 |
+----+------+---------+
2 rows in set (0.00 sec)

Query OK, 3 rows affected (0.00 sec)
Records: 2  Duplicates: 1  Warnings: 0

+----+------+---------+
| id | a    | cnt |
+----+------+---------+
|  1 | A    |   2 |
|  2 | B    |   1 |
|  4 | C    |   1 |


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

在以上示例 INSERT INTO t (a)VALUES ('A'), ('C')ON DUPLICATE KEY UPDATE cnt = cnt + 1; 语句中，自增值 3 被分配为A 键对应的 id 值，但实际上 3 并未作为 id 值插入进表中。这是因为该 INSERT 语句包含一个重复键A，使得自增序列不连续，出现了间隙。该行为尽管与 MySQL 不同，但仍是合法的。MySQL 在其他情况下也会出现自增序列不连续的情况，例如事务被中止和回滚时。

AUTO_ID_CACHE

如果在另一台服务器上执行插入操作，那么 AUTO_INCREMENT 值的顺序可能会剧烈跳跃，这是由于每台服务器都有各自缓存的 AUTO_INCREMENT 自增值。

```
CREATE TABLE t (a INT PRIMARY KEY AUTO_INCREMENT, b TIMESTAMP NOT NULL DEFAULT NOW());
INSERT INTO t (a) VALUES (NULL), (NULL), (NULL);
INSERT INTO t (a) VALUES (NULL);
SELECT * FROM t;
```

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


+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
+---------+---------------------+
4 rows in set (0.00 sec)
```

以下示例在最先的一台服务器上执行一个插入 INSERT 操作，生成 AUTO_INCREMENT 值 4。因为这台服务器上仍有剩余的 AUTO_INCREMENT 缓存值可用于分配。在该示例中，值的顺序不具有全局单调性：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
+---------+---------------------+
5 rows in set (0.00 sec)
```

AUTO_INCREMENT 缓存不会持久化，重启会导致缓存值失效。以下示例中，最先的一台服务器重启后，向该服务器执行一条插入操作：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
| 2030001 | 2020-09-09 20:54:11 |
+---------+---------------------+
6 rows in set (0.00 sec)
```

TiDB 服务器频繁重启可能导致 AUTO_INCREMENT 缓存值被快速消耗。在以上示例中，最先的一台服务器本来有可用的缓存值 [5-3000]。但重启后，这些值便丢失了，无法进行重新分配。

用户不应指望 AUTO_INCREMENT 值保持连续。在以下示例中，一台TiDB 服务器的缓存值为 [2000001-2030000]。当手动插入值 2029998 时，TiDB 取用了一个新缓存区间的值：

```
INSERT INTO t (a) VALUES (2029998);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.00 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.02 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
```

```
| 2030001 | 2020-09-09 20:54:11 |
| 2029998 | 2020-09-09 21:08:11 |
| 2029999 | 2020-09-09 21:08:11 |
| 2030000 | 2020-09-09 21:08:11 |
| 2060001 | 2020-09-09 21:08:11 |
| 2060002 | 2020-09-09 21:08:11 |
+---------+---------------------+
11 rows in set (0.00 sec)
```

以上示例插入 2030000 后，下一个值为 2060001，即顺序出现跳跃。这是因为另一台 TiDB 服务器获取了中间缓存区间 [2030001-2060000]。当部署有多台 TiDB 服务器时，AUTO_INCREMENT 值的顺序会出现跳跃，因为对缓存值的请求是交叉出现的。

缓存大小控制

TiDB 自增 ID 的缓存大小在早期版本中是对用户透明的。从 v3.1.2、v3.0.14 和 v4.0.rc-2 版本开始，TiDB 引入了 AUTO_ID_CACHE 表选项来允许用户自主设置自增ID 分配缓存的大小。例如：

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 100;
Query OK, 0 rows affected (0.02 sec)

INSERT INTO t VALUES();
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

SELECT * FROM t;
+---+
| a |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
```

此时如果将该列的自增缓存无效化，重新进行隐式分配：

```
DELETE FROM t;
Query OK, 1 row affected (0.01 sec)

RENAME TABLE t to t1;
Query OK, 0 rows affected (0.01 sec)

INSERT INTO t1 VALUES()
Query OK, 1 row affected (0.00 sec)

SELECT * FROM t;
+-----+
| a   |
```

```
+-----+
| 101 |
+-----+
1 row in set (0.00 sec)
```

可以看到再一次分配的值为 101，说明该表的自增 ID 分配缓存的大小为 100。

此外如果在批量插入的 INSERT 语句中所需连续ID 长度超过 AUTO_ID_CACHE 的长度时，TiDB 会适当调大缓存以便能够保证该语句的正常插入。

自增步长和偏移量设置

从v3.0.9 和v4.0.rc-1 开始，和MySQL 的行为类似，自增列隐式分配的值遵循session 变量@@auto_increment_increment <⟶ 和@@auto_increment_offset 的控制，其中自增列隐式分配的值(ID) 将满足式子(ID - auto_increment_offset <⟶)% auto_increment_increment == 0。

MySQL 兼容模式

从 v6.4.0 开始，TiDB 实现了中心化分配自增ID 的服务，可以支持TiDB 实例不缓存数据，而是每次请求都访问中心化服务获取ID。

当前中心化分配服务内置在TiDB 进程，类似于 DDL Owner 的工作模式。有一个TiDB 实例将充当“主”的角色提供 ID 分配服务，而其它的TiDB 实例将充当“备” 角色。当“主” 节点发生故障时，会自动进行“主备切换”，从而保证中心化服务的高可用。

MySQL 兼容模式的使用方式是，建表时将 AUTO_ID_CACHE 设置为 1：

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 1;
```

> 注意：
>
> 在TiDB 各个版本中，AUTO_ID_CACHE 设置为 1 都表明TiDB 不再缓存ID，但是不同版本的实现方式不一样：
>
> - 对于 TiDB v6.4.0 之前的版本，由于每次分配 ID 都需要通过一个 TiKV 事务完成 AUTO_INCREMENT 值的持久化修改，因此设置 AUTO_ID_CACHE 为 1 会出现性能下降。
> - 对于 v6.4.0 及以上版本，由于引入了中心化的分配服务，AUTO_INCREMENT 值的修改只是在 TiDB 服务进程中的一个内存操作，相较于之前版本更快。
> - 将 AUTO_ID_CACHE 设置为 1 表示TiDB 使用默认的缓存大小 30000。

使用 MySQL 兼容模式后，能保证ID 唯一、单调递增，行为几乎跟 MySQL 完全一致。即使跨TiDB 实例访问，ID 也不会出现回退。只有当中心化服务的“主” TiDB 实例异常崩溃时，才有可能造成少量ID 不连续。这是因为主备切换时，“备” 节点需要丢弃一部分之前的“主” 节点可能已经分配的ID，以保证ID 不出现重复。

使用限制

目前在TiDB 中使用 AUTO_INCREMENT 有以下限制：

- 对于 v6.6.0 及更早的TiDB 版本，定义的列必须为主键或者索引前缀。

- 只能定义在类型为整数、FLOAT 或 DOUBLE 的列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- 不支持使用 ALTER TABLE 来添加 AUTO_INCREMENT 属性。
- 支持使用 ALTER TABLE 来移除 AUTO_INCREMENT 属性。但从TiDB 2.1.18 和 3.0.4 版本开始，TiDB 通过 session 变量 @@tidb_allow_remove_auto_inc 控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除列的 AUTO_INCREMENT 属性，默认是不允许移除。
- ALTER TABLE 需要 FORCE 选项来将 AUTO_INCREMENT 设置为较小的值。
- 将 AUTO_INCREMENT 设置为小于 MAX(<auto_increment_column>) 的值会导致重复键，因为预先存在的值不会被跳过。

4.8.1.1.2 AUTO_RANDOM 从 v3.1.0 版本开始引入

使用场景

由于 AUTO_RANDOM 的值具有随机性和唯一性，因此 AUTO_RANDOM 通常用于代替AUTO_INCREMENT，以避免 TiDB 分配连续的ID 值造成单个存储节点的写热点问题。如果当前表的 AUTO_INCREMENT 列是主键列，且列类型为 BIGINT，可以通过 ALTER TABLE t MODIFY COLUMN id BIGINT AUTO_RANDOM(5); 从 AUTO_INCREMENT 切换成 AUTO_RANDOM。

关于如何在高并发写入场景下调优TiDB，请参阅TiDB 高并发写入场景最佳实践。

基本概念

AUTO_RANDOM 是应用在 BIGINT 类型列的属性，用于列值的自动分配。其自动分配的值满足随机性和唯一性。

以下语句均可创建包含 AUTO_RANDOM 列的表，其中 AUTO_RANDOM 列必须被包含在主键中，并且是主键的第一列。

```
CREATE TABLE t (a BIGINT AUTO_RANDOM, b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b VARCHAR(255));
CREATE TABLE t (a BIGINT AUTO_RANDOM(6), b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT AUTO_RANDOM(5, 54), b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT AUTO_RANDOM(5, 54), b VARCHAR(255), PRIMARY KEY (a, b));
```

AUTO_RANDOM 关键字可以被包裹在TiDB 可执行注释中，注释语法请参考TiDB 可执行注释。

```
CREATE TABLE t (a bigint /*T![auto_rand] AUTO_RANDOM */, b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a bigint PRIMARY KEY /*T![auto_rand] AUTO_RANDOM */, b VARCHAR(255));
CREATE TABLE t (a BIGINT /*T![auto_rand] AUTO_RANDOM(6) */, b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT  /*T![auto_rand] AUTO_RANDOM(5, 54) */, b VARCHAR(255), PRIMARY KEY (a))
    <-> ;
```

在用户执行 INSERT 语句时：

- 如果语句中显式指定了 AUTO_RANDOM 列的值，则该值会被正常插入到表中。
- 如果语句中没有显式指定 AUTO_RANDOM 列的值，TiDB 会自动生成一个随机的值插入到表中。

```
tidb> CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b VARCHAR(255));
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

```
tidb> INSERT INTO t(a, b) VALUES (1, 'string');
Query OK, 1 row affected (0.00 sec)

tidb> SELECT * FROM t;
+---+--------+
| a | b      |
+---+--------+
| 1 | string |
+---+--------+
1 row in set (0.01 sec)

tidb> INSERT INTO t(b) VALUES ('string2');
Query OK, 1 row affected (0.00 sec)

tidb> INSERT INTO t(b) VALUES ('string3');
Query OK, 1 row affected (0.00 sec)

tidb> SELECT * FROM t;
+---------------------+---------+
| a                   | b       |
+                     +         +
|                   1 | string  |
| 1152921504606846978 | string2 |
| 4899916394579099651 | string3 |
+                     +         +
3 rows in set (0.00 sec)
 --------------------- ----------
```

TiDB 自动分配的 AUTO_RANDOM(S, R) 列值共有 64 位：

- S 表示分片位的数量，取值范围是 1 到 15。默认为 5。
- R 表示自动分配值域的总长度，取值范围是 32 到 64。默认为 64。

AUTO_RANDOM 列值的具体结构如下：

总位数	符号位	保留位	分片位	自增位
64 bits	0/1 bit	(64-R) bits	S bits	(R-1-S) bits

- 符号位的长度由该列是否存在 UNSIGNED 属性决定：存在则为 0，否则为 1。
- 保留位的长度为 64-R，保留位的内容始终为 0。
- 分片位的内容通过计算当前事务的开始时间的哈希值而得。要使用不同的分片位数量（例如 10），可以在建表时指定 AUTO_RANDOM(10)。
- 自增位的值保存在存储引擎中，按顺序分配，每次分配完值后会自增 1。自增位保证了 AUTO_RANDOM 列值全局唯一。当自增位耗尽后，再次自动分配时会报 Failed to read auto-increment value from

注意：

分片位长度(S)的选取：

- 由于总位数固定为 64 位，分片位的数量会影响到自增位的数量：当分片位数增加时，自增位数会减少，反之亦然。因此，你需要权衡“自动分配值的随机性” 以及“可用空间”。
- 最佳实践是将分片位设置为 `log(2, x)`，其中 `x` 为当前集群存储引擎的数量。例如，一个 TiDB 集群中存在 16 个 TiKV，分片位可以设置为 `log(2, 16)`，即 4。在所有Region 被均匀调度到各个TiKV 上以后，此时大批量写入的负载可被均匀分布到不同TiKV 节点，以实现资源最大化利用。

值域长度(R)的选取：

- 通常，在应用程序的数值类型无法表示完整的 64 位整数时需要设置R 参数。
- 例如：JSON number 类型的取值范围为 `[-(2^53)+1, (2^53)-1]`， 而 TiDB 很容易会为 `AUTO_RANDOM(5)` 的列分配超出该范围的整数，导致应用程序读取该列的值时出现异常。此时，你可以用 `AUTO_RANDOM(5, 54)` 代替 `AUTO_RANDOM(5)`，使得 TiDB 不会分配出大于 9007199254740991 (2^53-1) 的整数。

AUTO RANDOM 列隐式分配的值会影响 `last_insert_id()`。可以使用 `SELECT last_insert_id()` 获取上一次TiDB 隐式分配的ID。

要查看某张含有 AUTO_RANDOM 属性的表的分片位数量， 除了 SHOW CREATE TABLE 以外， 还可以在系统表 INFORMATION_SCHEMA.TABLES 中 TIDB_ROW_ID_SHARDING_INFO 一列中查到模式为 PK_AUTO_RANDOM_BITS=x 的值，其中x 为分片位的数量。

创建完一张含有 AUTO_RANDOM 属性的表后，可以使用 SHOW WARNINGS 查看当前表可支持的最大隐式分配的次数：

```
CREATE TABLE t (a BIGINT AUTO_RANDOM, b VARCHAR(255), PRIMARY KEY (a));
SHOW WARNINGS;
```

输出结果如下：

```
+-------+------+-----------------------------------------------------------+
| Level | Code | Message                                                   |
+-------+------+-----------------------------------------------------------+
| Note  | 1105 | Available implicit allocation times: 288230376151711743 |
+-------+------+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

使用限制

目前在TiDB 中使用 AUTO_RANDOM 有以下限制：

- 要使用显式插入的功能，需要将系统变量 @@allow_auto_random_explicit_insert 的值设置为 1（默认值为 0）。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。
- 该属性必须指定在 BIGINT 类型的主键列上，否则会报错。此外，当主键属性为 NONCLUSTERED 时，即使是整型主键列，也不支持使用 AUTO_RANDOM。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 不支持使用 ALTER TABLE 来修改 AUTO_RANDOM 属性，包括添加或移除该属性。
- 支持将AUTO_INCREMENT 属性改为AUTO_RANDOM 属性。但在AUTO_INCREMENT 的列数据最大值已接近BIGINT 类型最大值的情况下，修改可能会失败。
- 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。
- AUTO_RANDOM 列的数据很难迁移到 AUTO_INCREMENT 列上，因为 AUTO_RANDOM 列自动分配的值通常都很大。

4.8.1.1.3 SHARD_ROW_ID_BITS

本文介绍表属性 SHARD_ROW_ID_BITS，它用来设置隐式 _tidb_rowid 分片数量的bit 位数。

基本概念

对于非聚簇索引主键或没有主键的表，TiDB 会使用一个隐式的自增rowid。大量执行 INSERT 插入语句时会把数据集中写入单个Region，造成写入热点。

通过设置 SHARD_ROW_ID_BITS，可以把 rowid 打散写入多个不同的Region，缓解写入热点问题。

- 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

关于 SHARD_ROW_ID_BITS 的更多使用信息，可参考使用 SHARD_ROW_ID_BITS 处理热点表。

语句示例

```
CREATE TABLE t (
    id INT PRIMARY KEY NONCLUSTERED
) SHARD_ROW_ID_BITS = 4;
```

```
ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

4.8.1.2 字面值

TiDB 字面值包括字符字面值、数值字面值、时间日期字面值、十六进制、二进制字面值和 NULL 字面值。以下分别对这些字面值进行一一介绍。

4.8.1.2.1 String Literals

String Literals 是一个bytes 或者characters 的序列，两端被单引号 ' 或者双引号 " 包围，例如：

```
'example string'
"example string"
```

如果字符串是连续的，会被合并为一个独立的 string。以下表示是一样的：

```
'a string'
'a' ' ' 'string'
"a" ' ' "string"
```

如果开启了 ANSI_QUOTES SQL MODE，那么只有单引号内的会被认为是 String Literals，对于双引号内的字符串，会被认为是一个identifier。

字符串分为以下两种：

- 二进制字符串(binary string)：由字节序列构成，它的 charset 和 collation 都是 binary，在互相比较时利用字节作为单位。
- 非二进制字符串：由字符序列构成，有除 binary 以外的多种charset 和 collation，在相互比较时用字符（一个字符可能包含多个字节，取决于charset 的选择）作为单位。

一个 String Literal 可以拥有一个可选的 character set introducer 和 COLLATE clause，可以用来指定特定的 charset 和 collation。

```
[_charset_name]'string' [COLLATE collation_name]
```

例如：

```
SELECT _latin1'string';
SELECT _binary'string';
SELECT _utf8'string' COLLATE utf8_bin;
```

你可以使用 N'literal' 或者 n'literal' 来创建使用national character set 的字符串，下列语句是一样的：

```
SELECT N'some text';
SELECT n'some text';
SELECT _utf8'some text';
```

要在字符串中表示某些特殊字符，可以利用转义字符进行转义：

转义字符	含义
\0	ASCII NUL (X’00’) 字符
\’	单引号
\“	双引号
\b	退格符号
\n	换行符
\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。

4.8.1.2.2 Numeric Literals

数值字面值包括integer 跟 Decimal 类型跟浮点数字面值。

integer 可以包括 . 作为小数点分隔，数字前可以有 - 或者 + 来表示正数或者负数。

精确数值字面值可以表示为如下格式：`1, .2, 3.4, -5, -6.78, +9.10.`

科学记数法也是被允许的，表示为如下格式：`1.2E3, 1.2E-3, -1.2E3, -1.2E-3`。

更多细节见 MySQL 官方文档。

4.8.1.2.3 Date and Time Literals

Date 跟 Time 字面值有几种格式，例如用字符串表示，或者直接用数字表示。在TiDB 里面，当TiDB 期望一个 Date 的时候，它会把 `'2017-08-24'`，`'20170824'`，20170824 当做是 Date。

TiDB 的 Date 值有以下几种格式：

- `'YYYY-MM-DD'` 或者 `'YY-MM-DD'`， 这里的 - 分隔符并不是严格的， 可以是任意的标点符号。比如 `'2017-08-24'`，`'2017&08&24'`，`'2012@12^31'` 都是一样的。唯一需要特别对待的是 '.' 号，它被当做是小数点，用于分隔整数和小数部分。

  Date 和Time 部分可以被 'T' 分隔，它的作用跟空格符是一样的，例如 `2017-8-24 10:42:00` 跟 `2017-8-24 T10:42:00` 是一样的。

- `'YYYYMMDDHHMMSS'` 或者 `'YYMMDDHHMMSS'`， 例如 `'20170824104520'` 和 `'170824104520'` 被当做是 `'2017-08-24 10:45:20'`， 但是如果你提供了一个超过范围的值， 例如`'170824304520'`， 那这就不是一个有效的 Date 字面值。需要注意 `YYYYMMDD HHMMSS`、`YYYYMMDD HH:MM:DD` 和 `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 `'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。

4.8.1.2.4 Boolean Literals

常量 TRUE 和 FALSE 等于 1 和 0，它是大小写不敏感的。

```
SELECT TRUE, true, tRuE, FALSE, FaLsE, false;
```

```
+------+------+------+-------+-------+-------+
| TRUE | true | tRuE | FALSE | FaLsE | false |
+------+------+------+-------+-------+-------+
|    1 |    1 |    1 |     0 |     0 |     0 |
+------+------+------+-------+-------+-------+
1 row in set (0.00 sec)
```

4.8.1.2.5 Hexadecimal Literals

十六进制字面值是有X 和 0x 前缀的字符串，后接表示十六进制的数字。注意 0x 是大小写敏感的，不能表示为 0X。

例如：

```
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)
```

4.8.1.2.6 Bit-Value Literals

位值字面值用b 或者 0b 做前缀，后接以 0 和 1 组成的二进制数字。其中 0b 是区分大小写的，0B 则会报错。

合法的 Bit-value：

- b’01’

- B’01’
- 0b01

非法的 Bit-value：

- b'2'（2 不是二进制数值，必须为 0 或 1）
- 0B01（0B 必须是小写 0b）

默认情况，位值字面值是一个二进制字符串。

Bit-value 是作为二进制返回的，所以输出到 MySQL Client 可能会无法显示，如果要转换为可打印的字符，可以使用内建函数 BIN() 或者 HEX()：

```
CREATE TABLE t (b BIT(8));
INSERT INTO t SET b = b'00010011';
INSERT INTO t SET b = b'1110';
INSERT INTO t SET b = b'100101';
```

```
SELECT b+0, BIN(b), HEX(b) FROM t;
```

```
+------+--------+--------------+
| b+0  | BIN(b) | HEX(b) |
+------+--------+--------------+
|   19 | 10011  | 13     |
|   14 | 1110   | E      |
|   37 | 100101 | 25     |
+------+--------+--------------+
3 rows in set (0.00 sec)
```

4.8.1.2.7 NULL Values

NULL 代表数据为空，它是大小写不敏感的，与 \N（大小写敏感）同义。

> 注意：
>
> NULL 跟 0 并不一样，跟空字符串 '' 也不一样。

4.8.1.3 Schema 对象名

本文介绍TiDB SQL 语句中的模式对象名。

模式对象名用于命名TiDB 中所有的模式对象，包括database、 table、 index、 column、 alias 等等。在 SQL 语句中，可以通过标识符(identifier) 来引用这些对象。

标识符可以被反引号包裹，即 SELECT * FROM t 也可以写成 SELECT * FROM `t`。但如果标识符中存在至少一个特殊符号，或者它是一个保留关键字，那就必须使用反引号包裹来引用它所代表的模式对象。

```
SELECT * FROM `table` WHERE `table`.id = 20;
```

如果 SQL MODE 中设置了 ANSI_QUOTES，那么 TiDB 会将被双引号" 包裹的字符串识别为identifier。

```
MySQL [test]> CREATE TABLE "test" (a varchar(10));
 ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
     <-> your TiDB version for the right syntax to use line 1 column 19 near ""test" (a varchar
    <-> (10))"

MySQL [test]> SET SESSION sql_mode='ANSI_QUOTES';
Query OK, 0 rows affected (0.000 sec)

MySQL [test]> CREATE TABLE "test" (a varchar(10));
Query OK, 0 rows affected (0.012 sec)
```

如果要在被引用的标识符中使用反引号这个字符，则需要重复两次反引号，例如创建一个表a `b：

```
CREATE TABLE `a``b` (a int);
```

在 select 语句中，alias 部分可以用标识符或者字符串：

```
SELECT 1 AS `identifier`, 2 AS 'string';
```

```
+------------+--------------+
| identifier | string |
+------------+--------------+
|          1 |      2 |
+------------+--------------+
1 row in set (0.00 sec)
```

更多细节，请参考 MySQL 文档。

4.8.1.3.1 Identifier Qualifiers

Object Names（对象名字）有时可以被限定或者省略。例如在创建表的时候可以省略数据库限定名：

```
CREATE TABLE t (i int);
```

如果之前没有使用 USE 或者连接参数来设定数据库，会报 ERROR 1046 (3D000): No database selected 错误。此时可以指定数据库限定名：

```
CREATE TABLE test.t (i int);
```

. 的左右两端可以出现空格，table_name.col_name 等于 table_name . col_name。

如果要引用这个模式对象，那么请使用：

```
`table_name`.`col_name`
```

而不是：

```
`table_name.col_name`
```

更多细节，请参考 MySQL 文档。

4.8.1.4 关键字

本文介绍TiDB 的关键字，对保留字和非保留字作出区分，并汇总所有的关键字以供查询使用。

关键字是 SQL 语句中具有特殊含义的单词，例如 SELECT，UPDATE，DELETE 等等。它们之中有的能够直接作为标识符，被称为非保留关键字（简称非保留字），但有需要经过特殊处理才能作为标识符的字，被称为保留关键字（简称保留字）。但是，也存在一些特殊的非保留关键字，有时候可能也需要进行特殊处理，推荐你将它们当作保留关键字处理。

对于保留字，必须使用反引号包裹，才能作为标识符被使用。例如：

```
CREATE TABLE select (a INT);
```

```
ERROR 1105 (HY000): line 0 column 19 near " (a INT)" (total length 27)
```

```
CREATE TABLE `select` (a INT);
```

```
Query OK, 0 rows affected (0.09 sec)
```

而非保留字则不需要反引号也能直接作为标识符。例如 BEGIN 和 END 是非保留字，以下语句能够正常执行：

```
CREATE TABLE `select` (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.09 sec)
```

有一种特殊情况，如果使用了限定符.，那么也不需要用反引号：

```
CREATE TABLE test.select (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.08 sec)
```

4.8.1.4.1 关键字列表

下表列出了TiDB 中所有的关键字。其中保留字用 (R) 来标识。窗口函数的保留字用 (R-Window) 来标识。需要用反引号` 包裹的特殊非保留字用 (S) 来标识。

A

- ACCOUNT
- ACTION
- ADD (R)
- ADMIN (R)

- ADVISE
- AFTER
- AGAINST
- AGO
- ALGORITHM
- ALL (R)
- ALTER (R)
- ALWAYS
- ANALYZE (R)
- AND (R)
- ANY
- ARRAY (R)
- AS (R)
- ASC (R)
- ASCII
- ATTRIBUTE
- ATTRIBUTES
- AUTO_ID_CACHE
- AUTO_INCREMENT
- AUTO_RANDOM
- AUTO_RANDOM_BASE
- AVG
- AVG_ROW_LENGTH

B

- BACKEND
- BACKUP
- BACKUPS
- BEGIN
- BERNOULLI
- BETWEEN (R)
- BIGINT (R)
- BINARY (R)
- BINDING
- BINDINGS
- BINDING_CACHE
- BINLOG
- BIT
- BLOB (R)
- BLOCK
- BOOL
- BOOLEAN
- BOTH (R)
- BTREE

- BUCKETS (R)
- BUILTINS (R)
- BY (R)
- BYTE

C

- CACHE
- CALIBRATE
- CALL (R)
- CANCEL (R)
- CAPTURE
- CASCADE (R)
- CASCADED
- CASE (R)
- CAUSAL
- CHAIN
- CHANGE (R)
- CHAR (R)
- CHARACTER (R)
- CHARSET
- CHECK (R)
- CHECKPOINT
- CHECKSUM
- CIPHER
- CLEANUP
- CLIENT
- CLIENT_ERRORS_SUMMARY
- CLOSE
- CLUSTER
- CLUSTERED
- CMSKETCH (R)
- COALESCE
- COLLATE (R)
- COLLATION
- COLUMN (R)
- COLUMN_FORMAT
- COLUMNS
- COMMENT
- COMMIT
- COMMITTED
- COMPACT
- COMPRESSED
- COMPRESSION
- CONCURRENCY

- CONFIG
- CONNECTION
- CONSISTENCY
- CONSISTENT
- CONSTRAINT (R)
- CONTEXT
- CONTINUE (R)
- CONVERT (R)
- CPU
- CREATE (R)
- CROSS (R)
- CSV_BACKSLASH_ESCAPE
- CSV_DELIMITER
- CSV_HEADER
- CSV_NOT_NULL
- CSV_NULL
- CSV_SEPARATOR
- CSV_TRIM_LAST_SEPARATORS
- CUME_DIST (R-Window)
- CURRENT
- CURRENT_DATE (R)
- CURRENT_ROLE (R)
- CURRENT_TIME (R)
- CURRENT_TIMESTAMP (R)
- CURRENT_USER (R)
- CURSOR (R)
- CYCLE

D

- DATA
- DATABASE (R)
- DATABASES (R)
- DATE
- DATETIME
- DAY
- DAY_HOUR (R)
- DAY_MICROSECOND (R)
- DAY_MINUTE (R)
- DAY_SECOND (R)
- DDL (R)
- DEALLOCATE
- DECIMAL (R)
- DECLARE
- DEFAULT (R)

- DEFINER
- DELAY_KEY_WRITE
- DELAYED (R)
- DELETE (R)
- DENSE_RANK (R-Window)
- DEPTH (R)
- DESC (R)
- DESCRIBE (R)
- DIGEST
- DIRECTORY
- DISABLE
- DISABLED
- DISCARD
- DISK
- DISTINCT (R)
- DISTINCTROW (R)
- DIV (R)
- DO
- DOUBLE (R)
- DRAINER (R)
- DROP (R)
- DUAL (R)
- DUPLICATE
- DYNAMIC

E

- ELSE (R)
- ELSEIF (R)
- ENABLE
- ENABLED
- ENCLOSED (R)
- ENCRYPTION
- END
- ENFORCED
- ENGINE
- ENGINES
- ENUM
- ERROR
- ERRORS
- ESCAPE
- ESCAPED (R)
- EVENT
- EVENTS
- EVOLVE

- EXCEPT (R)
- EXCHANGE
- EXCLUSIVE
- EXECUTE
- EXISTS (R)
- EXIT (R)
- EXPANSION
- EXPIRE
- EXPLAIN (R)
- EXTENDED

F

- FAILED_LOGIN_ATTEMPTS
- FALSE (R)
- FAULTS
- FETCH (R)
- FIELDS
- FILE
- FIRST
- FIRST_VALUE (R-Window)
- FIXED
- FLOAT (R)
- FLUSH
- FOLLOWING
- FOR (R)
- FORCE (R)
- FOREIGN (R)
- FORMAT
- FOUND
- FROM (R)
- FULL
- FULLTEXT (R)
- FUNCTION

G

- GENERAL
- GENERATED (R)
- GLOBAL
- GRANT (R)
- GRANTS
- GROUP (R)
- GROUPS (R-Window)

H

- HANDLER
- HASH
- HAVING (R)
- HELP
- HIGH_PRIORITY (R)
- HISTOGRAM
- HISTORY
- HOSTS
- HOUR
- HOUR_MICROSECOND (R)
- HOUR_MINUTE (R)
- HOUR_SECOND (R)

I

- IDENTIFIED
- IF (R)
- IGNORE (R)
- ILIKE (R)
- IMPORT
- IMPORTS
- IN (R)
- INCREMENT
- INCREMENTAL
- INDEX (R)
- INDEXES
- INFILE (R)
- INNER (R)
- INOUT (R)
- INSERT (R)
- INSERT_METHOD
- INSTANCE
- INT (R)
- INT1 (R)
- INT2 (R)
- INT3 (R)
- INT4 (R)
- INT8 (R)
- INTEGER (R)
- INTERSECT (R)
- INTERVAL (R)
- INTO (R)
- INVISIBLE
- INVOKER
- IO

- IPC
- IS (R)
- ISOLATION
- ISSUER
- ITERATE (R)

J

- JOB (R)
- JOBS (R)
- JOIN (R)
- JSON

K

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

L

- LABELS
- LAG (R-Window)
- LANGUAGE
- LAST
- LAST_BACKUP
- LAST_VALUE (R-Window)
- LASTVAL
- LEAD (R-Window)
- LEADING (R)
- LEAVE (R)
- LEFT (R)
- LESS
- LEVEL
- LIKE (R)
- LIMIT (R)
- LINEAR (R)
- LINES (R)
- LIST
- LOAD (R)
- LOCAL
- LOCALTIME (R)
- LOCALTIMESTAMP (R)
- LOCATION

- LOCK (R)
- LOCKED
- LOGS
- LONG (R)
- LONGBLOB (R)
- LONGTEXT (R)
- LOW_PRIORITY (R)

M

- MASTER
- MATCH (R)
- MAXVALUE (R)
- MAX_CONNECTIONS_PER_HOUR
- MAX_IDXNUM
- MAX_MINUTES
- MAX_QUERIES_PER_HOUR
- MAX_ROWS
- MAX_UPDATES_PER_HOUR
- MAX_USER_CONNECTIONS
- MB
- MEDIUMBLOB (R)
- MEDIUMINT (R)
- MEDIUMTEXT (R)
- MEMBER
- MEMORY
- MERGE
- MICROSECOND
- MINUTE
- MINUTE_MICROSECOND (R)
- MINUTE_SECOND (R)
- MINVALUE
- MIN_ROWS
- MOD (R)
- MODE
- MODIFY
- MONTH

N

- NAMES
- NATIONAL
- NATURAL (R)
- NCHAR
- NEVER

- NEXT
- NEXTVAL
- NO
- NOCACHE
- NOCYCLE
- NODEGROUP
- NODE_ID (R)
- NODE_STATE (R)
- NOMAXVALUE
- NOMINVALUE
- NONCLUSTERED
- NONE
- NOT (R)
- NOWAIT
- NO_WRITE_TO_BINLOG (R)
- NTH_VALUE (R-Window)
- NTILE (R-Window)
- NULL (R)
- NULLS
- NUMERIC (R)
- NVARCHAR

O

- OF (R)
- OFF
- OFFSET
- OLTP_READ_ONLY
- OLTP_READ_WRITE
- OLTP_WRITE_ONLY
- ON (R)
- ON_DUPLICATE
- ONLINE
- ONLY
- OPEN
- OPTIMISTIC (R)
- OPTIMIZE (R)
- OPTION (R)
- OPTIONAL
- OPTIONALLY (R)
- OR (R)
- ORDER (R)
- OUT (R)
- OUTER (R)
- OUTFILE (R)

- OVER (R-Window)

P

- PACK_KEYS
- PAGE
- PARSER
- PARTIAL
- PARTITION (R)
- PARTITIONING
- PARTITIONS
- PASSWORD
- PASSWORD_LOCK_TIME
- PAUSE
- PERCENT
- PERCENT_RANK (R-Window)
- PER_DB
- PER_TABLE
- PESSIMISTIC (R)
- PLACEMENT (S)
- PLUGINS
- POINT
- POLICY
- PRECEDING
- PRECISION (R)
- PREPARE
- PRESERVE
- PRE_SPLIT_REGIONS
- PRIMARY (R)
- PRIVILEGES
- PROCEDURE (R)
- PROCESS
- PROCESSLIST
- PROFILE
- PROFILES
- PROXY
- PUMP (R)
- PURGE

Q

- QUARTER
- QUERIES
- QUERY
- QUICK

R

- RANGE (R)
- RANK (R-Window)
- RATE_LIMIT
- READ (R)
- REAL (R)
- REBUILD
- RECOVER
- RECURSIVE (R)
- REDUNDANT
- REFERENCES (R)
- REGEXP (R)
- REGION (R)
- REGIONS (R)
- RELEASE (R)
- RELOAD
- REMOVE
- RENAME (R)
- REORGANIZE
- REPAIR
- REPEAT (R)
- REPEATABLE
- REPLACE (R)
- REPLICA
- REPLICAS
- REPLICATION
- REQUIRE (R)
- REQUIRED
- RESOURCE
- RESPECT
- RESTART
- RESTORE
- RESTORES
- RESTRICT (R)
- RESUME
- REUSE
- REVERSE
- REVOKE (R)
- RIGHT (R)
- RLIKE (R)
- ROLE
- ROLLBACK
- ROUTINE
- ROW (R)

- ROW_COUNT
- ROW_FORMAT
- ROW_NUMBER (R-Window)
- ROWS (R-Window)
- RTREE

S

- SAMPLES (R)
- SAN
- SAVEPOINT
- SECOND
- SECOND_MICROSECOND (R)
- SECONDARY_ENGINE
- SECONDARY_LOAD
- SECONDARY_UNLOAD
- SECURITY
- SELECT (R)
- SEND_CREDENTIALS_TO_TIKV
- SEPARATOR
- SEQUENCE
- SERIAL
- SERIALIZABLE
- SESSION
- SET (R)
- SETVAL
- SHARD_ROW_ID_BITS
- SHARE
- SHARED
- SHOW (R)
- SHUTDOWN
- SIGNED
- SIMPLE
- SKIP
- SKIP_SCHEMA_FILES
- SLAVE
- SLOW
- SMALLINT (R)
- SNAPSHOT
- SOME
- SOURCE
- SPATIAL (R)
- SPLIT (R)
- SQL (R)
- SQL_BIG_RESULT (R)

- SQL_BUFFER_RESULT
- SQL_CACHE
- SQL_CALC_FOUND_ROWS (R)
- SQL_NO_CACHE
- SQL_SMALL_RESULT (R)
- SQL_TSI_DAY
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR
- SQLEXCEPTION (R)
- SQLSTATE (R)
- SQLWARNING (R)
- SSL (R)
- START
- STARTING (R)
- STATS (R)
- STATS_AUTO_RECALC
- STATS_BUCKETS (R)
- STATS_COL_CHOICE
- STATS_COL_LIST
- STATS_EXTENDED (R)
- STATS_HEALTHY (R)
- STATS_HISTOGRAMS (R)
- STATS_META (R)
- STATS_OPTIONS
- STATS_PERSISTENT
- STATS_SAMPLE_PAGES
- STATS_SAMPLE_RATE
- STATUS
- STORAGE
- STORED (R)
- STRAIGHT_JOIN (R)
- STRICT_FORMAT
- SUBJECT
- SUBPARTITION
- SUBPARTITIONS
- SUPER
- SWAPS
- SWITCHES
- SYSTEM
- SYSTEM_TIME

T

- TABLE (R)
- TABLES
- TABLESAMPLE (R)
- TABLESPACE
- TABLE_CHECKSUM
- TEMPORARY
- TEMPTABLE
- TERMINATED (R)
- TEXT
- THAN
- THEN (R)
- TIDB (R)
- TIFLASH (R)
- TIKV_IMPORTER
- TIME
- TIMESTAMP
- TINYBLOB (R)
- TINYINT (R)
- TINYTEXT (R)
- TO (R)
- TOKEN_ISSUER
- TOPN (R)
- TPCC
- TRACE
- TRADITIONAL
- TRAILING (R)
- TRANSACTION
- TRIGGER (R)
- TRIGGERS
- TRUE (R)
- TRUNCATE
- TTL
- TTL_ENABLE
- TTL_JOB_INTERVAL
- TYPE
- TiDB_CURRENT_TSO (R)

U

- UNBOUNDED
- UNCOMMITTED
- UNDEFINED
- UNICODE

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

V

- VALIDATION
- VALUE
- VALUES (R)
- VARBINARY (R)
- VARCHAR (R)
- VARCHARACTER (R)
- VARIABLES
- VARYING (R)
- VIEW
- VIRTUAL (R)
- VISIBLE

W

- WAIT
- WARNINGS
- WEEK
- WEIGHT_STRING
- WHEN (R)
- WHERE (R)
- WHILE (R)
- WIDTH (R)
- WINDOW (R-Window)
- WITH (R)
- WITHOUT
- WORKLOAD
- WRITE (R)

X

- X509
- XOR (R)

Y

- YEAR
- YEAR_MONTH (R)

Z

- ZEROFILL (R)

4.8.1.5 用户自定义变量

> **警告：**
> 当前该功能为实验特性，不建议在生产环境中使用。

本文介绍TiDB 的用户自定义变量的概念，以及设置和读取用户自定义变量的方法。

用户自定义变量格式为 `@var_name`。组成 `var_name` 的字符可以是任何能够组成标识符(identifier) 的字符，包括数字 `0-9`、字母 `a-zA-Z`、下划线 `_`、美元符号`$` 以及 UTF-8 字符。此外，还包括英文句号 `.`。用户自定义变量是大小写不敏感的。

用户自定义变量跟 session 绑定，当前设置的用户变量只在当前连接中可见，其他客户端连接无法查看。

4.8.1.5.1 设置用户自定义变量

用 SET 语句可以设置用户自定义变量，语法为 `SET @var_name = expr [, @var_name = expr] ...;`。例如：

```
SET @favorite_db = 'TiDB';
```

```
SET @a = 'a', @b = 'b', @c = 'c';
```

其中赋值符号还可以使用 `:=`。例如：

```
SET @favorite_db := 'TiDB';
```

赋值符号右边的内容可以是任意合法的表达式。例如：

```
SET @c = @a + @b;
```

```
SET @c = b'1000001' + b'1000001';
```

4.8.1.5.2 读取用户自定义变量

要读取一个用户自定义变量，可以使用 SELECT 语句查询：

```
SELECT @a1, @a2, @a3
```

```
+------+------+------+
| @a1  | @a2  | @a3  |
+------+------+------+
|    1 |    2 |    4 |
+------+------+------+
```

还可以在 SELECT 语句中赋值：

```
SELECT @a1, @a2, @a3, @a4 := @a1+@a2+@a3;
```

```
+------+------+------+----------------------------------+
| @a1  | @a2  | @a3  | @a4 := @a1+@a2+@a3 |
+------+------+------+----------------------------------+
|    1 |    2 |    4 |                                7 |
+------+------+------+----------------------------------+
```

其中变量 @a4 在被修改或关闭连接之前，值始终为 7。

如果设置用户变量时用了十六进制字面量或者二进制字面量，TiDB 会把它当成二进制字符串。如果要将其设置成数字，那么可以手动加上 CAST 转换，或者在表达式中使用数字的运算符：

```
SET @v1 = b'1000001';
SET @v2 = b'1000001'+0;
SET @v3 = CAST(b'1000001' AS UNSIGNED);
```

```
SELECT @v1, @v2, @v3;
```

```
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| A    | 65   | 65   |
+------+------+------+
```

如果获取一个没有设置过的变量，会返回一个 NULL：

```
SELECT @not_exist;
```

```
+------------+
| @not_exist |
+------------+
| NULL       |
+------------+
```

除了 SELECT 读取用户自定义变量以外，常见的用法还有 PREPARE 语句，例如：

```
SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
PREPARE stmt FROM @s;
SET @a = 6;
SET @b = 8;
EXECUTE stmt USING @a, @b;
```

```
+------------------- +
| hypotenuse |
+------------------- +
|          10 |
+------------------- +
```

用户自定义变量的内容不会在 SQL 语句中被当成标识符，例如：

```
SELECT * FROM t;
```

```
+---+
| a |
+---+
| 1 |
+---+
```

```
SET @col = "`a`";
SELECT @col FROM t;
```

```
+----------+
| @col |
+----------+
| `a`  |
+----------+
```

4.8.1.5.3 MySQL 兼容性

除 SELECT ... INTO <variable> 外，MySQL 和 TiDB 支持的语法相同。

更多细节，请参考 MySQL 文档。

4.8.1.6 表达式语法(Expression Syntax)

表达式是一个或多个值、操作符或函数的组合。在TiDB 中，表达式主要使用在 SELECT 语句的各个子句中，包括 Group by 子句、Where 子句、Having 子句、Join 条件以及窗口函数等。此外，部分 DDL 语句也会使用到表达式，例如建表时默认值的设置、生成列的设置，分区规则等。

表达式包含几种类型：

- 标识符，可参考模式对象名。
- 谓词、数值、字符串、日期表达式等，这些类型的字面值也是表达式。
- 函数调用，窗口函数等。可参考函数和操作符概述和窗口函数。
- 其他，包括paramMarker（即?）、系统变量和用户变量、CASE 表达式等。

以下规则是表达式的语法，该语法基于TiDB parser 的 parser.y 文件中所定义的规则。此外，下列语法图的可导航版本请参考TiDB SQL 语法图。

```
Expression ::=
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        <⟷ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    <⟷ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?

PredicateExpr ::=
     BitExpr ( BetweenOrNotOp BitExpr 'AND' BitExpr )* ( InOrNotOp ( '(' ExpressionList ')' |
         <⟷ SubSelect ) | LikeOrNotOp SimpleExpr LikeEscapeOpt | RegexpOrNotOp SimpleExpr )?

BitExpr ::=
    BitExpr ( ( '|' | '&' | '<<' | '>>' | '*' | '/' | '%' | 'DIV' | 'MOD' | '^' ) BitExpr | ( '+'
        <⟷  | '-' ) ( BitExpr | "INTERVAL" Expression TimeUnit ) )
|   SimpleExpr

SimpleExpr ::=
    SimpleIdent ( ( '->' | '->>' ) stringLit )?
|   FunctionCallKeyword
|   FunctionCallNonKeyword
|   FunctionCallGeneric
|   SimpleExpr ( 'COLLATE' CollationName | pipes SimpleExpr )
|   WindowFuncCall
|   Literal
|   paramMarker
|   Variable
|   SumExpr
|   ( '!' | '~' | '-' | '+' | 'NOT' | 'BINARY' ) SimpleExpr
|   'EXISTS'? SubSelect
|   ( ( '(' ( ExpressionList ',' )? | 'ROW' '(' ExpressionList ',' ) Expression | builtinCast '('
    <⟷  Expression 'AS' CastType | ( 'DEFAULT' | 'VALUES' ) '(' SimpleIdent | 'CONVERT' '('
    <⟷ Expression ( ',' CastType | 'USING' CharsetName ) ) ')'
|   'CASE' ExpressionOpt WhenClause+ ElseOpt 'END'
```

4.8.1.7 注释语法

本文档介绍TiDB 支持的注释语法。

TiDB 支持三种注释风格：

- 用 # 注释一行：

```
SELECT 1+1;     # 注释文字
```

```
+---------+
| 1+1   |
+---------+
|     2 |
+---------+
1 row in set (0.00 sec)
```

- 用 -- 注释一行：

```
SELECT 1+1;     -- 注释文字
```

```
+---------+
| 1+1   |
+---------+
|     2 |
+---------+
1 row in set (0.00 sec)
```

用 -- 注释时，必须要在其之后留出至少一个空格，否则注释不生效：

```
SELECT 1+1--1;
```

```
+-------------+
| 1+1--1 |
+-------------+
|       3 |
+-------------+
1 row in set (0.01 sec)
```

- 用 /* */ 注释一块，可以注释多行：

```
SELECT 1 /* 这是行内注释文字 */ + 1;
```

```
+-------------+
| 1  + 1 |
+-------------+
|       2 |
+-------------+
1 row in set (0.01 sec)
```

```
SELECT 1+
/*
/*> 这是一条
/*> 多行注释
/*> */
    1;
```

```
+--------------------+
| 1+


        1 |
+--------------------+
|                  2 |
+--------------------+
1 row in set (0.001 sec)
```

4.8.1.7.1 MySQL 兼容的注释语法

TiDB 也跟 MySQL 保持一致，支持一种C 风格注释的变体：

```
/*! Specific code */
```

或者

```
/*!50110 Specific code */
```

和 MySQL 一样，TiDB 会执行注释中的语句。

例如：`SELECT /*! STRAIGHT_JOIN */ col1 FROM table1,table2 WHERE ...`

在 TiDB 中，这种写法等价于 `SELECT STRAIGHT_JOIN col1 FROM table1,table2 WHERE ...`

如果注释中指定了 Server 版本号，例如 `/*!50110 KEY_BLOCK_SIZE=1024 */`，在 MySQL 中表示只有 MySQL 的版本大于等于 5.1.10 才会处理这个comment 中的内容。但是在TiDB 中，这个 MySQL 版本号不会起作用，所有的 comment 都被会处理。

4.8.1.7.2 TiDB 可执行的注释语法

TiDB 也有独立的注释语法，称为TiDB 可执行注释语法。主要分为两种：

- `/*T! Specific code */`：该语法只能被 TiDB 解析执行，而在其他数据库中会被忽略。
- `/*T![feature_id] Specific code */`：该语法用于保证 TiDB 不同版本之间的兼容性。只有在当前版本中实现了 `feature_id` 对应的功能特性的TiDB，才会试图解析该注释里的 SQL 片段。例如 v3.1.1 中引入了 AUTO_RANDOM 特性，该版本能够将 `/*T![auto_rand] auto_random */` 解析为 `auto_random`；而 v3.0.0 中没有实现 AUTO_RANDOM 特性，则上述 SQL 语句片段会被忽略。注意前几个字符 `/*T![` 中，各字符之间没有任何空格。

4.8.1.7.3 优化器注释语法

还有一种注释会被当做是优化器 Hint 特殊对待：

```
SELECT /*+ hint */ FROM ...;
```

TiDB 支持的相关优化器hint 详见 Optimizer Hints。

> **注意：**
>
> 在 MySQL 客户端中，TiDB 可执行注释语法会被默认当成注释被清除掉。在 MySQL 客户端 5.7.7 之前的版本中，Hint 也会被默认当成注释被清除掉。推荐在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

更多细节，请参考 MySQL 文档。

4.8.2 SQL 语句

4.8.2.1 ADD COLUMN

`ALTER TABLE.. ADD COLUMN` 语句用于在已有表中添加列。在TiDB 中，`ADD COLUMN` 为在线操作，不会阻塞表中的数据读写。

4.8.2.1.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AddColumnSpec ( ',' AddColumnSpec )*

TableName ::=
    Identifier ('.' Identifier)?

AddColumnSpec
         ::= 'ADD' 'COLUMN' 'IF NOT EXISTS'? ColumnName ColumnType ColumnOption+ ( 'FIRST' | '
             <⟷ AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
```

```
            | 'UNIQUE' 'KEY'?
            | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
            | 'SERIAL' 'DEFAULT' 'VALUE'
            | 'ON' 'UPDATE' NowSymOptionFraction
            | 'COMMENT' stringLit
            | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
                <-> NULL' ) )?
            | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
            | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
                <-> OnDeleteUpdateOpt
            | 'COLLATE' CollationName
            | 'COLUMN_FORMAT' ColumnFormat
            | 'STORAGE' StorageMedia
            | 'AUTO_RANDOM' ( '(' LengthNum ')' )?
```

4.8.2.1.2 示例

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

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

```
INSERT INTO t1 VALUES (NULL);
```

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

```
SELECT * FROM t1;
```

```
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD COLUMN c1 INT NOT NULL;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+--------+
| id | c1 |
+----+--------+
|  1 |  0 |
```

```
+----+-------+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD c2 INT NOT NULL AFTER c1;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+-------+
| id | c1 | c2 |
+----+----+-------+
|  1 |  0 |  0 |
+----+----+-------+
1 row in set (0.00 sec)
```

4.8.2.1.3 MySQL 兼容性

- 不支持将新添加的列设为 PRIMARY KEY。
- 不支持将新添加的列设为 AUTO_INCREMENT。
- 对添加生成列有局限性，具体可参考：生成列局限性。

4.8.2.1.4 另请参阅

- ADD INDEX
- CREATE TABLE

4.8.2.2 ADD INDEX

ALTER TABLE.. ADD INDEX 语句用于在已有表中添加一个索引。在TiDB 中，ADD INDEX 为在线操作，不会阻塞表中的数据读写。

> **警告：**
>
> - 在升级TiDB 集群的过程中，请勿执行 DDL 语句，否则可能会出现行为未定义的问题。
> - 集群中有 DDL 语句正在被执行时（通常为 ADD INDEX 和列类型变更等耗时较久的 DDL 语句），请勿进行升级操作。在升级前，建议使用ADMIN SHOW DDL 命令查看集群中是否有正在进行的 DDL Job。如需升级，请等待 DDL 执行完成或使用ADMIN CANCEL DDL 命令取消该 DDL Job 后再进行升级。

4.8.2.2.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AddIndexSpec ( ',' AddIndexSpec )*

AddIndexSpec
         ::= 'ADD' ( ( 'PRIMARY' 'KEY' | ( 'KEY' | 'INDEX' ) 'IF NOT EXISTS'? | 'UNIQUE' ( 'KEY'
             <—> | 'INDEX' )? ) ( ( Identifier? 'USING' | Identifier 'TYPE' ) IndexType )? | '
             <—> FULLTEXT' ( 'KEY' | 'INDEX' )? IndexName ) '(' IndexPartSpecification ( ','
             <—> IndexPartSpecification )* ')' IndexOption*

IndexPartSpecification
         ::= ( ColumnName ( '(' LengthNum ')' )? | '(' Expression ')' ) ( 'ASC' | 'DESC' )

IndexOption
         ::= 'KEY_BLOCK_SIZE' '='? LengthNum
           | IndexType
           | 'WITH' 'PARSER' Identifier
           | 'COMMENT' stringLit
           | 'VISIBLE'
           | 'INVISIBLE'

IndexType
         ::= 'BTREE'
           | 'HASH'
           | 'RTREE'
```

4.8.2.2.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------------------------------------+
    <—>
| id                      | estRows  | task      | access object | operator info
    <—>                 |
```

```
+-------------------------+----------+-----------+---------------+--------------------------------------------------------------+
    <->
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    <->              |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    <->             |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    <-> pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------------------------------------+
    <->
3 rows in set (0.00 sec)
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+                         +         +           +                         +-----------------------------------------------------
    <->
| id                      | estRows | task      | access object           | operator info
    <->                               |
+-------------------------+---------+-----------+-------------------------+                                                     -----
    <->
| IndexReader_6           | 0.01    | root      |                         | index:IndexRangeScan_5
    <->                        |
| └─IndexRangeScan_5      | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    <-> order:false, stats:pseudo |
+                         +         +           +                         +-----------------------------------------------------
    <->
2 rows in set (0.00 sec)
```

4.8.2.2.3 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）。
- 无法向表中添加 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

4.8.2.2.4 另请参阅

- 索引的选择
- 错误索引的解决方案

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

4.8.2.3 ADMIN

ADMIN 语句是TiDB 扩展语法，用于查看TiDB 自身的状态，并对TiDB 中的表数据进行校验。本文介绍了下列与 ADMIN 相关的扩展语句：

- ADMIN RELOAD
- ADMIN PLUGIN
- ADMIN ... BINDINGS
- ADMIN REPAIR TABLE
- ADMIN SHOW SLOW

4.8.2.3.1 ADMIN 与 DDL 相关的扩展语句

语句	功能描述
ADMIN CANCEL DDL JOBS	取消当前正在运行的 DDL 作业
ADMIN CHECKSUM TABLE	计算表中所有行和索引的 CRC64 校验和
[ADMIN CHECK [TABLE|INDEX]](#admin-check-[table	index])
[ADMIN SHOW DDL [JOBS|QUERIES]](#admin-show-ddl-[jobs	job-queries])
ADMIN SHOW TELEMETRY	显示通过遥测功能收集到并分享给 PingCAP 的使用信息。

4.8.2.3.2 ADMIN RELOAD 语句

```
ADMIN RELOAD expr_pushdown_blacklist;
```

以上语句用于重新加载表达式下推的黑名单。

```
ADMIN RELOAD opt_rule_blacklist;
```

以上语句用于重新加载逻辑优化规则的黑名单。

4.8.2.3.3 ADMIN PLUGIN 语句

```
ADMIN PLUGINS ENABLE plugin_name [, plugin_name] ...;
```

以上语句用于启用 plugin_name 插件。

```
ADMIN PLUGINS DISABLE plugin_name [, plugin_name] ...;
```

以上语句用于禁用 plugin_name 插件。

4.8.2.3.4 ADMIN ... BINDINGS 语句

```
ADMIN FLUSH bindings;
```

以上语句用于持久化 SQL Plan 绑定的信息。

```
ADMIN CAPTURE bindings;
```

以上语句可以将出现超过一次的 selectexecution-plan 语句生成 SQL Plan 的绑定。

```
ADMIN EVOLVE bindings;
```

开启自动绑定功能后，每隔 bind-info-lease（默认值为 3s）触发一次 SQL Plan 绑定信息的演进。以上语句用于主动触发此演进，SQL Plan 绑定详情可参考：执行计划管理。

```
ADMIN RELOAD bindings;
```

以上语句用于重新加载 SQL Plan 绑定的信息。

4.8.2.3.5 ADMIN REPAIR TABLE 语句

```
ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT;
```

ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT 用于在极端情况下，对存储层中的表的元信息进行非可信的覆盖。“非可信”是指需要人为保证原表的元信息可以完全由 CREATE TABLE STATEMENT 提供。该语句需要打开配置文件项中的repair-mode 开关，并且需要确保所修复的表名在repair-table-list 名单中。

4.8.2.3.6 ADMIN SHOW SLOW 语句

```
ADMIN SHOW SLOW RECENT N;
```

```
ADMIN SHOW SLOW TOP [INTERNAL | ALL] N;
```

这两种语句的具体操作详情可参考：admin show slow 语句。

4.8.2.3.7 语句概览

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        <⟷> | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
       <⟷> | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
       <⟷> INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
       <⟷> LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
       <⟷> NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
       <⟷> | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
       <⟷> CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

4.8.2.3.8 使用示例

执行以下命令，可查看正在执行的 DDL 任务中最近 10 条已经完成的 DDL 任务。未指定 NUM 时，默认只显示最近 10 条已经执行完的 DDL 任务。

```
ADMIN SHOW DDL jobs;
```

```
+--------+---------+------------+--------------+----------------------+-----------+----------+-----------+----
    <->
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE       | SCHEMA_STATE          | SCHEMA_ID | TABLE_ID |
    <-> ROW_COUNT | START_TIME                         | END_TIME                              | STATE
    <->          |
+--------+---------+------------+--------------+----------------------+-----------+----------+-----------+----
    <->
| 45     | test    | t1         | add index      | write reorganization | 32        | 37       | 0
    <->          | 2019-01-10 12:38:36.501 +0800 CST |                                       |
    <-> running        |
| 44     | test    | t1         | add index      | none                 | 32        | 37       | 0
    <->          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    <-> rollback done |
| 43     | test    | t1         | add index      | public               | 32        | 37       | 6
     <->         | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    <->         |
| 42     | test    | t1         | drop index     | none                 | 32        | 37       | 0
     <->         | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    <->         |
| 41     | test    | t1         | add index      | public               | 32        | 37       | 0
     <->         | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    <->         |
| 40     | test    | t1         | drop column    | none                 | 32        | 37       | 0
     <->         | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    <->         |
| 39     | test    | t1         | add column     | public               | 32        | 37       | 0
     <->         | 2019-01-10 12:32:55.42 +0800 CST  | 2019-01-10 12:32:56.24 +0800 CST  | synced
    <->         |
| 38     | test    | t1         | create table   | public               | 32        | 37       | 0
     <->         | 2019-01-10 12:32:41.956 +0800 CST | 2019-01-10 12:32:43.956 +0800 CST | synced
    <->         |
| 36     | test    |            | drop table     | none                 | 32        | 34       | 0
     <->         | 2019-01-10 11:29:59.982 +0800 CST | 2019-01-10 11:30:00.45 +0800  CST | synced
    <->         |
| 35     | test    |            | create table   | public               | 32        | 34       | 0
     <->         | 2019-01-10 11:29:40.741 +0800 CST | 2019-01-10 11:29:41.682 +0800 CST | synced
    <->         |
| 33     | test    |            | create schema  | public               | 32        | 0        | 0
     <->         | 2019-01-10 11:29:22.813 +0800 CST | 2019-01-10 11:29:23.954 +0800 CST | synced
```

```
    <->        |
+--------+---------+------------+--------------------+---------------+-----------+----------+------------+----
    <->
```

执行以下命令，可查看正在执行的 DDL 任务中最近 5 条已经执行完的 DDL 任务：

```
ADMIN SHOW DDL JOBS 5;
```

```
+--------+---------+------------+--------------+---------------------+-----------+----------+------------+----
    <->
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE       | SCHEMA_STATE          | SCHEMA_ID | TABLE_ID |
    <-> ROW_COUNT | START_TIME                         | END_TIME                           | STATE
    <->          |
+--------+---------+------------+--------------+---------------------+-----------+----------+------------+----
    <->
| 45     | test    | t1         | add index      | write reorganization | 32        | 37       | 0
    <->          | 2019-01-10 12:38:36.501 +0800 CST |                                    |
    <-> running        |
| 44     | test    | t1         | add index      | none                 | 32        | 37       | 0
    <->          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    <-> rollback done |
| 43     | test    | t1         | add index      | public               | 32        | 37       | 6
     <->          | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    <->         |
| 42     | test    | t1         | drop index     | none                 | 32        | 37       | 0
     <->          | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    <->         |
| 41     | test    | t1         | add index      | public               | 32        | 37       | 0
     <->          | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    <->         |
| 40     | test    | t1         | drop column    | none                 | 32        | 37       | 0
     <->          | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    <->         |
+--------+---------+------------+--------------+---------------------+-----------+----------+------------+----
    <->
```

执行以下命令，可查看test 数据库中未执行完成的 DDL 任务，包括正在执行中以及最近 5 条已经执行完但是执行失败的 DDL 任务。

```
ADMIN SHOW DDL JOBS 5 WHERE state != 'synced' AND db_name = 'test';
```

```
+--------+---------+------------+--------------+---------------------+-----------+----------+------------+----
    <->
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE       | SCHEMA_STATE          | SCHEMA_ID | TABLE_ID |
    <-> ROW_COUNT | START_TIME                         | END_TIME                           | STATE
    <->          |
```

```
+--------+---------+------------+--------------+---------------------+----------+----------+-----------+----
    <->
| 45     | test    | t1         | add index    | write reorganization | 32       | 37       | 0
    <->         | 2019-01-10 12:38:36.501 +0800 CST |                                        |
    <-> running       |
| 44     | test    | t1         | add index    | none                 | 32       | 37       | 0
    <->         | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    <-> rollback done |
+--------+---------+------------+--------------+---------------------+----------+----------+-----------+----
    <->
```

- JOB_ID：每个 DDL 操作对应一个 DDL 作业，JOB_ID 全局唯一。
- DB_NAME：执行 DDL 操作的数据库的名称。
- TABLE_NAME：执行 DDL 操作的表的名称。
- JOB_TYPE：DDL 操作的类型。
- SCHEMA_STATE：schema 的当前状态。如果 JOB_TYPE 是 add index，则为 index 的状态；如果是 add column，则为 column 的状态，如果是 create table，则为 table 的状态。常见的状态有以下几种：
  - none：表示不存在。一般 drop 操作或者 create 操作失败回滚后，会变为 none 状态。
  - delete only、write only、delete reorganization、write reorganization：这四种状态是中间状态。由于中间状态转换很快，一般操作中看不到这几种状态，只有执行 add index 操作时能看到处于 write reorganization 状态，表示正在添加索引数据。
  - public：表示存在且可用。一般 create table 和 add index/column 等操作完成后，会变为 public 状态，表示新建的table/column/index 可以正常读写了。
- SCHEMA_ID：执行 DDL 操作的数据库的ID。
- TABLE_ID：执行 DDL 操作的表的ID。
- ROW_COUNT：执行 add index 操作时，当前已经添加完成的数据行数。
- START_TIME：DDL 操作的开始时间。
- END_TIME：DDL 操作的结束时间。
- STATE：DDL 操作的状态。常见的状态有以下几种：
  - none：表示该操作任务已经进入 DDL 作业队列中，但尚未执行，因为还在排队等待前面的 DDL 作业完成。另一种原因可能是执行 drop 操作后，会变为 none 状态，但是很快会更新为 synced 状态，表示所有TiDB 实例都已经同步到该状态。
  - running：表示该操作正在执行。
  - synced：表示该操作已经执行成功，且所有 TiDB 实例都已经同步该状态。
  - rollback done：表示该操作执行失败，回滚完成。
  - rollingback：表示该操作执行失败，正在回滚。
  - cancelling：表示正在取消该操作。这个状态只有在用 ADMIN CANCEL DDL JOBS 命令取消 DDL 作业时才会出现。

4.8.2.3.9 MySQL 兼容性

ADMIN 语句是TiDB 对于 MySQL 语法的扩展。

4.8.2.4 ADMIN CANCEL DDL

ADMIN CANCEL DDL 语句用于取消当前正在运行的 DDL 作业。可以通过ADMIN SHOW DDL JOBS 语句获取 DDL 作业的 job_id。

用于取消已经提交但未执行完成的DDL 任务。取消完成后，执行DDL 任务的SQL 语句会返回ERROR 8214 (HY000) <-> : Cancelled DDL job 的错误。取消一个已经执行完成的DDL 任务会在RESULT 列看到DDL Job:90 not found 的错误，表示该任务已从 DDL 等待队列中被移除。

4.8.2.4.1 语法图

```
AdminStmt ::=
     'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
         <-> | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        <-> | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        <-> INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        <-> LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        <-> NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        <-> | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        <-> CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

NumList ::=
    Int64Num ( ',' Int64Num )*
```

4.8.2.4.2 示例

可以通过 ADMIN CANCEL DDL JOBS 语句取消当前正在运行的 DDL 作业，并返回对应作业是否取消成功：

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

如果取消失败，会显示失败的具体原因。

> 注意：
>
> - 只有该操作可以取消 DDL 作业，其他所有的操作和环境变更（例如机器重启、集群重启）都不会取消 DDL 作业。
> - 该操作可以同时取消多个 DDL 作业，可以通过ADMIN SHOW DDL JOBS 语句来获取 DDL 作业的 job_id。
> - 如果希望取消的作业已经执行完毕，取消操作将失败。

4.8.2.4.3 MySQL 兼容性

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

4.8.2.4.4 另请参阅

- ADMIN SHOW DDL [JOBS|QUERIES]

4.8.2.5 ADMIN CHECKSUM TABLE

ADMIN CHECKSUM TABLE 语句用于计算表中所有行和索引的 CRC64 校验和。在TiDB Lightning 等程序中，可通过此语句来确保导入操作成功。

4.8.2.5.1 语法图

```
AdminStmt ::=
     'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
         <⟷ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        <⟷ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        <⟷ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        <⟷ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        <⟷ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        <⟷ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        <⟷ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

TableNameList ::=
    TableName ( ',' TableName )*
```

4.8.2.5.2 示例

创建表 t1：

```
CREATE TABLE t1(id INT PRIMARY KEY);
```

插入一些数据：

```
INSERT INTO t1 VALUES (1),(2),(3);
```

计算表 t1 的校验和：

```
ADMIN CHECKSUM TABLE t1;
```

输出结果示例如下：

```
+---------+------------+---------------------+-----------+-------------------------+
| Db_name | Table_name | Checksum_crc64_xor  | Total_kvs | Total_bytes |
+---------+------------+---------------------+-----------+-------------------------+
| test    | t1         | 10909174369497628533 |         3 |          75 |
+---------+------------+---------------------+-----------+-------------------------+
1 row in set (0.00 sec)
```

4.8.2.5.3 MySQL 兼容性

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

4.8.2.6 ADMIN CHECK [TABLE|INDEX]

`ADMIN CHECK [TABLE|INDEX]` 语句用于校验表中数据和对应索引的一致性，不支持校验外键约束。

4.8.2.6.1 语法图

```
AdminStmt ::=
     'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
         <⟷ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        <⟷ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        <⟷ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        <⟷ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        <⟷ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        <⟷ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        <⟷ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

TableNameList ::=
    TableName ( ',' TableName )*
```

4.8.2.6.2 示例

可以通过 `ADMIN CHECK TABLE` 语句校验 `tbl_name` 表中所有数据和对应索引的一致性：

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

若通过一致性校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name;
```

以上语句用于对 `tbl_name` 表中 `idx_name` 索引对应列数据和索引数据进行一致性校验。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name (lower_val, upper_val) [, (lower_val, upper_val)] ...;
```

以上语句用于对 `tbl_name` 表中 `idx_name` 索引对应列数据和索引数据进行一致性校验，并且指定了需要检查的数据范围。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

4.8.2.6.3 MySQL 兼容性

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

4.8.2.6.4 另请参阅

- ADMIN REPAIR

4.8.2.7 ADMIN CLEANUP INDEX

`ADMIN CLEANUP INDEX` 语句用于在表发生行数据和索引的一致性故障时，删除表中多余的索引，使表的行数据和索引恢复一致状态。注意，该语法尚不支持外键约束。

4.8.2.7.1 语法图

```
AdminCleanupStmt ::=
    'ADMIN' 'CLEANUP' ( 'INDEX' TableName IndexName | 'TABLE' 'LOCK' TableNameList )

TableNameList ::=
    TableName ( ',' TableName )*
```

4.8.2.7.2 示例

假设由于一些原因（例如灾难恢复场景，集群中丢失了部分行数据），数据库中的 `tbl` 表出现数据和索引不一致现象：

```
SELECT * FROM tbl;
ERROR 1105 (HY000): inconsistent index idx handle count 3 isn't equal to value count 2

ADMIN CHECK INDEX tbl idx ;
ERROR 1105 (HY000): handle &kv.CommonHandle{encoded:[]uint8{0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0
    <-> x0, 0x0, 0xf8}, colEndOffsets:[]uint16{0xa}}, index:types.Datum{k:0x5, decimal:0x0,
    <-> length:0x0, i:0, collation:"utf8mb4_bin", b:[]uint8{0x0}, x:interface {}(nil)} != record
    <-> :<nil>
```

从 `SELECT` 查询的错误信息可以看到，`tbl` 表中包含 2 条行数据和 3 条索引数据，这意味着行数据与索引数据出现了不一致故障，同时至少有 1 条索引处于悬空状态。此时可以使用 `ADMIN CLEANUP INDEX` 语句删除悬空的索引：

```
ADMIN CLEANUP INDEX tbl idx;
```

执行结果示例如下：

```
ADMIN CLEANUP INDEX tbl idx;
+-------------------------+
| REMOVED_COUNT |
+-------------------------+
|              1 |
+-------------------------+
```

此时，可以重新使用 `ADMIN CHECK INDEX` 语句检查数据索引的一致性，验证数据是否恢复到正常状态：

```
ADMIN CHECK INDEX tbl idx;
Query OK, 0 rows affected (0.01 sec)
```

注意：

当由于副本丢失导致数据索引出现不一致时：

- 通常行数据与索引数据都有丢失，此时需要同时使用 ADMIN CLEANUP INDEX 与ADMIN <—> RECOVER INDEX 语句保证行数据与索引数据的一致性。
- ADMIN CLEANUP INDEX 总是单并发执行。在表数据量大时，建议通过重建索引的方式进行索引数据的恢复。
- ADMIN CLEANUP INDEX 在执行期间，不会为对应的表或索引记录加锁，TiDB 允许其他的会话同时对表记录进行更新或修改。然而，在此情况下，ADMIN CLEANUP INDEX 可能无法正确处理所有表记录。因此，在执行 ADMIN CLEANUP INDEX 时，请避免同时修改表数据。
- 若你使用TiDB 企业版，此时建议提交工单联系 PingCAP 支持工程师进行处理。

ADMIN CLEANUP INDEX 命令不具有原子性：若该命令在执行期间中断，建议重新执行直到成功。

4.8.2.7.3 MySQL 兼容性

ADMIN CLEANUP INDEX 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.7.4 另请参阅

- ADMIN CHECK TABLE/INDEX
- ADMIN RECOVER INDEX

4.8.2.8 ADMIN RECOVER INDEX

ADMIN RECOVER INDEX 语句用于在表发生行数据和索引的一致性故障时，根据表中多余的索引，使表的行数据和索引重新恢复到一致状态。注意，该语法尚不支持外键约束。

4.8.2.8.1 语法图

```
AdminCleanupStmt ::=
    'ADMIN' 'RECOVER' 'INDEX' TableName IndexName
```

4.8.2.8.2 示例

假设由于一些原因（例如灾难恢复场景，集群中丢失了部分索引数据），数据库中的 tbl 表出现行数据和索引不一致现象：

```
SELECT * FROM tbl;
ERROR 1105 (HY000): inconsistent index idx handle count 2 isn't equal to value count 3

ADMIN CHECK INDEX tbl idx ;
```

```
ERROR 1105 (HY000): handle &kv.CommonHandle{encoded:[]uint8{0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0
    <-> x0, 0x0, 0xf8}, colEndOffsets:[]uint16{0xa}}, index:types.Datum{k:0x5, decimal:0x0,
    <-> length:0x0, i:0, collation:"utf8mb4_bin", b:[]uint8{0x0}, x:interface {}(nil)} != record
    <-> :<nil>
```

从 SELECT 查询的错误信息可以看到，tbl 表中包含 3 条行数据和 2 条索引数据，这意味着行数据与索引数据出现了不一致故障，同时至少有 1 条行数据缺少了对应的索引。此时可以使用 ADMIN RECOVER INDEX 语句补充缺少的索引：

```
ADMIN RECOVER INDEX tbl idx;
```

执行结果示例如下：

```
ADMIN RECOVER INDEX tbl idx;
+-------------+---------------------+
| ADDED_COUNT | SCAN_COUNT |
+-------------+---------------------+
|           1 |           3 |
+-------------+---------------------+
1 row in set (0.00 sec)
```

此时，可以重新使用 ADMIN CHECK INDEX 语句对数据索引的一致性进行检查，验证数据恢复到正常状态：

```
ADMIN CHECK INDEX tbl idx;
Query OK, 0 rows affected (0.01 sec)
```

注意：

当由于副本丢失导致数据索引出现不一致时：

- 通常行数据与索引数据都有丢失，此时需要同时使用ADMIN CLEANUP INDEX 与 ADMIN <-> RECOVER INDEX 语句保证行数据与索引数据的一致性。
- ADMIN RECOVER INDEX 总是单并发执行。在表数据量大时，建议通过重建索引的方式进行索引数据的恢复。
- ADMIN RECOVER INDEX 在执行期间，不会为对应的表或索引记录加锁，这意味着TiDB 允许同时有其他的会话对表记录进行更新或修改。然而，在此情况下 ADMIN RECOVER INDEX 可能无法正确处理所有数据。因此，在执行 ADMIN RECOVER INDEX 时，请避免同时修改表数据。
- 若你使用TiDB 企业版，此时建议提交工单联系 PingCAP 支持工程师进行处理。

ADMIN RECOVER INDEX 命令不具有原子性：若该命令在执行期间中断，建议重新执行直到成功。

4.8.2.8.3 MySQL 兼容性

ADMIN RECOVER INDEX 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.8.4 另请参阅

- ADMIN CHECK TABLE/INDEX
- ADMIN CLEANUP INDEX

4.8.2.9 ADMIN SHOW DDL [JOBS|JOB QUERIES]

ADMIN SHOW DDL [JOBS|JOB QUERIES] 语句显示了正在运行和最近完成的 DDL 作业的信息。

4.8.2.9.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList | '
        <⟷ JOB' 'QUERIES' 'LIMIT' m 'OFFSET' n )? | 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?
```

4.8.2.9.2 示例

ADMIN SHOW DDL

可以通过 ADMIN SHOW DDL 语句查看当前正在运行的 DDL 作业状态，包括当前 schema 版本号、Owner 的 DDL ID 和地址、正在执行的 DDL 任务和 SQL、当前 TiDB 实例的 DDL ID。

```
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。该语句的返回结果字段描述如下：

- JOB_ID：每个 DDL 操作对应一个 DDL 任务，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：DDL 所操作的 schema 对象的当前状态。如果 JOB_TYPE 是 ADD INDEX，则为索引的状态；如果是 ADD COLUMN，则为列的状态；如果是 CREATE TABLE，则为表的状态。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，具体含义请参考TiDB 中在线 DDL 异步变更的原理。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：表示存在且对用户可用。一般 CREATE TABLE 和 ADD INDEX（或 ADD COLUMN）等操作完成后，会变为 public 状态，表示新建的表、列、索引可以正常读写了。
- 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：
  - queueing：表示该操作任务已经进入 DDL 任务队列中，但尚未执行，因为还在排队等待前面的 DDL 任务完成。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 任务时才会出现。

示例如下：

```
ADMIN SHOW DDL JOBS;
```

```
+--
    <--> ------+---------+-------------------+-------------+---------------------+-----------+----------+------
    <-->
| JOB_ID | DB_NAME | TABLE_NAME          | JOB_TYPE      | SCHEMA_STATE          | SCHEMA_ID |
    <--> TABLE_ID | ROW_COUNT | CREATE_TIME          | START_TIME          | END_TIME            |
    <--> STATE    |
+--
    <--> ------+---------+-------------------+-------------+---------------------+-----------+----------+------
    <-->
|     59 | test    | t1                  | add index     | write reorganization |         1 |
    <-->      55 |     88576 | 2020-08-17 07:51:58 | 2020-08-17 07:51:58 | NULL                |
    <--> running |
|     60 | test    | t2                  | add index     | none                 |         1 |
    <-->      57 |         0 | 2020-08-17 07:51:59 | 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 | 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 | 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 | 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 | 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 07:34:43 | 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 | 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 | 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 | 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 | 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 | 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。

ADMIN SHOW DDL JOB QUERIES LIMIT m OFFSET n

ADMIN SHOW DDL JOB QUERIES LIMIT m OFFSET n 语句用于查看指定范围 [n+1, n+m] 的 job_id 对应的 DDL 任务的原始 SQL 语句：

```
ADMIN SHOW DDL JOB QUERIES LIMIT m;           # -- 取出前 m 行
ADMIN SHOW DDL JOB QUERIES LIMIT n, m;        # -- 取出第 n+1 到 n+m 行
ADMIN SHOW DDL JOB QUERIES LIMIT m OFFSET n;  # -- 取出第 n+1 到 n+m 行
```

以上语法中n和m都是非负整数。语法的具体示例如下：

```
ADMIN SHOW DDL JOB QUERIES LIMIT 3;  # Retrieve first 3 rows
+--------+-----------------------------------------------------------------+
| JOB_ID | QUERY                                                           |
+--------+-----------------------------------------------------------------+
|     59 | ALTER TABLE t1 ADD INDEX index2 (col2)                          |
|     60 | ALTER TABLE t2 ADD INDEX index1 (col1)                          |
|     58 | CREATE TABLE t2 (id INT NOT NULL PRIMARY KEY auto_increment) |
+--------+-----------------------------------------------------------------+
3 rows in set (0.00 sec)
```

```
ADMIN SHOW DDL JOB QUERIES LIMIT 6, 2;  # Retrieve rows 7-8
+--------+-------------------------------------------------------------------------------------+
| JOB_ID | QUERY                                                                               |
+--------+-------------------------------------------------------------------------------------+
|     52 | ALTER TABLE t1 ADD INDEX index1 (col1)                                              |
|     51 | CREATE TABLE IF NOT EXISTS t1 (id INT NOT NULL PRIMARY KEY auto_increment) |
+--------+-------------------------------------------------------------------------------------+
3 rows in set (0.00 sec)
```

```
ADMIN SHOW DDL JOB QUERIES LIMIT 3 OFFSET 4;  # Retrieve rows 5-7
+--------+-----------------------------------------+
| JOB_ID | QUERY                                   |
+--------+-----------------------------------------+
|     54 | DROP TABLE IF EXISTS t3                 |
|     53 | ALTER TABLE t1 DROP INDEX index1        |
|     52 | ALTER TABLE t1 ADD INDEX index1 (col1) |
+--------+-----------------------------------------+
3 rows in set (0.00 sec)
```

该语句可以在 DDL 历史作业队列任意指定范围中搜索与 job_id 对应的正在运行中的 DDL 作业， 没有 ADMIN SHOW DDL JOB QUERIES 语句的最近 10 条结果的限制。

4.8.2.9.3 MySQL 兼容性

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

4.8.2.9.4 另请参阅

- ADMIN CANCEL DDL

4.8.2.10 ADMIN SHOW TELEMETRY

`ADMIN SHOW TELEMETRY` 语句用于查看通过遥测功能收集到并分享给 PingCAP 的使用信息。

4.8.2.10.1 语法图

```
AdminStmt ::=
     'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
         <⟶ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow | 'TELEMETRY' ) | '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' )
```

4.8.2.10.2 示例

```
ADMIN SHOW TELEMETRY\G
```

```
*************************** 1. row ***************************
 TRACKING_ID: a1ba1d97-b940-4d5b-a9d5-ddb0f2ac29e7
 LAST_STATUS: {
  "check_at": "2021-08-11T08:23:38+02:00",
  "is_error": false,
  "error_msg": "",
  "is_request_sent": true
}
DATA_PREVIEW: {
  "hardware": [
    {
      "instanceType": "tidb",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "4000",
      "cpu": {
        "cache": "8192",
        "cpuFrequency": "2301.00MHz",
        "cpuLogicalCores": "8",
        "cpuPhysicalCores": "4"
      },
      "memory": {
        "capacity": "16410021888"
      },
      "disk":
        { "ebbca862689fa9fef7c55c3112e375c4ce575fe4":
        {
```

```
        "deviceName": "ebbca862689fa9fef7c55c3112e375c4ce575fe4",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "opts": "bind,rw,relatime",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "397115568128",
        "usedPercent": "0.39"
      },
      "nvme0n1p1":
        { "deviceName":
        "nvme0n1p1", "free":
        "582250496",
        "freePercent": "0.93",
        "fstype": "vfat",
        "opts": "rw,relatime",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2":
        { "deviceName":
        "nvme0n1p2", "free":
        "701976576",
        "freePercent": "0.74",
        "fstype": "ext4",
        "opts": "rw,relatime",
        "path": "/boot",
        "total": "1023303680",
        "used": "250863616",
        "usedPercent": "0.26"
      }
    }
  },
  {
    "instanceType": "pd",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "2379",
    "cpu": {
      "cache": "8192",
      "cpuFrequency": "2301.00MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4"
    },
    "memory": {
```

```
      "capacity": "16410021888"
    },
    "disk":
      { "ebbca862689fa9fef7c55c3112e375c4ce575fe4":
      {
        "deviceName": "ebbca862689fa9fef7c55c3112e375c4ce575fe4",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "opts": "bind,rw,relatime",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "397115568128",
        "usedPercent": "0.39"
      },
      "nvme0n1p1":
        { "deviceName":
        "nvme0n1p1", "free":
        "582250496",
        "freePercent": "0.93",
        "fstype": "vfat",
        "opts": "rw,relatime",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2":
        { "deviceName":
        "nvme0n1p2", "free":
        "701976576",
        "freePercent": "0.74",
        "fstype": "ext4",
        "opts": "rw,relatime",
        "path": "/boot",
        "total": "1023303680",
        "used": "250863616",
        "usedPercent": "0.26"
      }
    }
  },
  {
    "instanceType": "tikv",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "20160",
    "cpu": {
      "cpuFrequency": "3730MHz",
      "cpuLogicalCores": "8",
```

```
      "cpuPhysicalCores": "4",
      "cpuVendorId": "GenuineIntel",
      "l1CacheLineSize": "64",
      "l1CacheSize": "32768",
      "l2CacheLineSize": "64",
      "l2CacheSize": "262144",
      "l3CacheLineSize": "64",
      "l3CacheSize": "8388608"
    },
    "memory": {
      "capacity": "16803861504"
    },
    "disk":
      { "36e7dfacbb83843f83075d78aeb4cf850a4882a1":
      {
        "deviceName": "36e7dfacbb83843f83075d78aeb4cf850a4882a1",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "398049751040",
        "usedPercent": "0.39"
      },
      "nvme0n1p1":
        { "deviceName":
        "nvme0n1p1", "free":
        "582250496",
        "freePercent": "0.93",
        "fstype": "vfat",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2":
        { "deviceName":
        "nvme0n1p2", "free":
        "701976576",
        "freePercent": "0.69",
        "fstype": "ext4",
        "path": "/boot",
        "total": "1023303680",
        "used": "321327104",
        "usedPercent": "0.31"
      }
    }
  },
```

```
    {
      "instanceType": "tiflash",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "3930",
      "cpu": {
        "cpuFrequency": "3400MHz",
        "cpuLogicalCores": "8",
        "cpuPhysicalCores": "4",
        "l1CacheLineSize": "64",
        "l1CacheSize": "32768",
        "l2CacheLineSize": "64",
        "l2CacheSize": "262144",
        "l3CacheLineSize": "64",
        "l3CacheSize": "8388608"
      },
      "memory": {
        "capacity": "16410021888"
      },
      "disk":
        { "36e7dfacbb83843f83075d78aeb4cf850a4882a1":
        {
          "deviceName": "36e7dfacbb83843f83075d78aeb4cf850a4882a1",
          "free": "624438726656",
          "freePercent": "0.61",
          "fstype": "btrfs",
          "path": "fb365c1216b59e1cfc86950425867007a60f4435",
          "total": "1022488477696",
          "used": "398049751040",
          "usedPercent": "0.39"
        }
      }
    }
  ],
  "instances": [
    {
      "instanceType": "tidb",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "4000",
      "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "statusPort": "10080",
      "version": "5.1.1",
      "gitHash": "797bddd25310ed42f0791c8eccb78be8cce2f502",
      "startTime": "2021-08-11T08:23:38+02:00",
      "upTime": "22.210217487s"
    },
    {
```

```
      "instanceType": "pd",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "2379",
      "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "statusPort": "2379",
      "version": "5.1.1",
      "gitHash": "7cba1912b317a533e18b16ea2ba9a14ed2891129",
      "startTime": "2021-08-11T08:23:32+02:00",
      "upTime": "28.210220368s"
    },
    {
      "instanceType": "tikv",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "20160",
      "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "statusPort": "20180",
      "version": "5.1.1",
      "gitHash": "4705d7c6e9c42d129d3309e05911ec6b08a25a38",
      "startTime": "2021-08-11T08:23:33+02:00",
      "upTime": "27.210221447s"
    },
    {
      "instanceType": "tiflash",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "3930",
      "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "statusPort": "20292",
      "version": "v5.1.1",
      "gitHash": "c8fabfb50fe28db17cc5118133a69be255c40efd",
      "startTime": "2021-08-11T08:23:40+02:00",
      "upTime": "20.210222452s"
    }
  ],
  "hostExtra": {
    "cpuFlags": [
      "fpu",
      "vme",
      "de",
      "pse",
      "tsc",
      "msr",
      "pae",
      "mce",
      "cx8",
      "apic",
```

```
        "sep",
        "mtrr",
        "pge",
        "mca",
        "cmov",
        "pat",
        "pse36",
        "clflush",
        "dts",
        "acpi",
        "mmx",
        "fxsr",
        "sse",
        "sse2",
        "ss",
        "ht",
        "tm",
        "pbe",
        "syscall",
        "nx",
        "pdpe1gb",
        "rdtscp",
        "lm",
        "constant_tsc",
        "art",
        "arch_perfmon",
        "pebs",
        "bts",
        "rep_good",
        "nopl",
        "xtopology",
        "nonstop_tsc",
        "cpuid",
        "aperfmperf",
        "pni",
        "pclmulqdq",
        "dtes64",
        "monitor",
        "ds_cpl",
        "vmx",
        "est",
        "tm2",
        "ssse3",
        "sdbg",
        "fma",
```

```
        "cx16",
        "xtpr",
        "pdcm",
        "pcid",
        "sse4_1",
        "sse4_2",
        "x2apic",
        "movbe",
        "popcnt",
        "tsc_deadline_timer",
        "aes",
        "xsave",
        "avx",
        "f16c",
        "rdrand",
        "lahf_lm",
        "abm",
        "3dnowprefetch",
        "cpuid_fault",
        "epb",
        "invpcid_single",
        "ssbd",
        "ibrs",
        "ibpb",
        "stibp",
        "ibrs_enhanced",
        "tpr_shadow",
        "vnmi",
        "flexpriority",
        "ept",
        "vpid",
        "ept_ad",
        "fsgsbase",
        "tsc_adjust",
        "sgx",
        "bmi1",
        "avx2",
        "smep",
        "bmi2",
        "erms",
        "invpcid",
        "mpx",
        "rdseed",
        "adx",
        "smap",
```

```
      "clflushopt",
      "intel_pt",
      "xsaveopt",
      "xsavec",
      "xgetbv1",
      "xsaves",
      "dtherm",
      "ida",
      "arat",
      "pln",
      "pts",
      "hwp",
      "hwp_notify",
      "hwp_act_window",
      "hwp_epp",
      "md_clear",
      "flush_l1d",
      "arch_capabilities"
    ],
    "cpuModelName": "Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz",
    "os": "linux",
    "platform": "fedora",
    "platformFamily": "fedora",
    "platformVersion": "34",
    "kernelVersion": "5.13.5-200.fc34.x86_64",
    "kernelArch": "x86_64",
    "virtualizationSystem": "kvm",
    "virtualizationRole": "host"
  },
  "reportTimestamp": 1628663040,
  "trackingId": "a1ba1d97-b940-4d5b-a9d5-ddb0f2ac29e7",
  "featureUsage": {
    "txn":
      { "asyncCommitUsed":
      true, "onePCUsed": true,
      "txnCommitCounter": {
        "twoPC": 9,
        "asyncCommit": 0,
        "onePC": 0
      }
    },
    "clusterIndex": {},
    "temporaryTable": false,
    "cte": {
      "nonRecursiveCTEUsed": 0,
```

```
      "recursiveUsed": 0,
      "nonCTEUsed": 13
    }
  },
  "windowedStats": [],
  "slowQueryStats": {
    "slowQueryBucket": {}
  }
}
1 row in set (0.0259 sec)
```

4.8.2.10.3 MySQL 兼容性

ADMIN 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.10.4 另请参阅

- 遥测
- 系统变量 tidb_enable_telemetry

4.8.2.11 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。

4.8.2.11.1 语法图

```
AlterDatabaseStmt ::=
    'ALTER' 'DATABASE' DBName? DatabaseOptionList

DatabaseOption ::=
    DefaultKwdOpt ( CharsetKw '='? CharsetName | 'COLLATE' '='? CollationName | 'ENCRYPTION' '='?
        <⟶ EncryptionOpt )
```

4.8.2.11.2 示例

修改数据库 test 的字符集为 utf8mb4：

```
ALTER DATABASE test DEFAULT CHARACTER SET = utf8mb4;
```

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

目前TiDB 只支持部分的字符集和排序规则，详情参阅字符集支持。

4.8.2.11.3 MySQL 兼容性

`ALTER DATABASE` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在GitHub 上提交issue。

4.8.2.11.4 另请参阅

- CREATE DATABASE
- SHOW DATABASES

4.8.2.12 ALTER INDEX

`ALTER INDEX` 语句用于修改索引的可见性，可以将索引设置为 `Visible` 或者 `Invisible`。设置为 `Invisible` 的索引即不可见索引(Invisible Index) 由 DML 语句维护，不会被查询优化器使用。

4.8.2.12.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AlterIndexSpec ( ',' AlterIndexSpec )*

AlterIndexSpec
         ::= 'ALTER' 'INDEX' Identifier ( 'VISIBLE' | 'INVISIBLE' )
```

4.8.2.12.2 示例

可以通过 `ALTER TABLE ... ALTER INDEX ...` 语句，修改索引的可见性：

```
CREATE TABLE t1 (c1 INT, UNIQUE(c1));
ALTER TABLE t1 ALTER INDEX c1 INVISIBLE;
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+--
    <->-----+----------------------------------------------------------------------------------------------------------
    <->
| Table | Create Table
                                      |
+--
    <->     +----------------------------------------------------------------------------------------------------------
    <->
| t1    | CREATE TABLE `t1` (
  `c1` int(11) DEFAULT NULL,
  UNIQUE KEY `c1` (`c1`) /*!80000 INVISIBLE */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
```

```
+--
    <-->------+-------------------------------------------------------------------------------------------------------------
    <-->
1 row in set (0.00 sec)
```

优化器将无法使用 c1 这个不可见的索引：

```
EXPLAIN SELECT c1 FROM t1 ORDER BY c1;
```

```
+--
    <-->  ----------------------+----------+-----------+---------------+-------------------------------------------------+
    <-->
| id                        | estRows  | task      | access object | operator info
    <-->                 |
+--
    <-->  ----------------------+----------+-----------+---------------+-------------------------------------------------+
    <-->
| Sort_4                    | 10000.00 | root      |               | test.t1.c1:asc
    <-->                 |
| └─TableReader_8           | 10000.00 | root      |               | data:TableFullScan_7
    <-->           |
|   └─TableFullScan_7       | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    <--> pseudo |
+--
    <-->  ----------------------+----------+-----------+---------------+-------------------------------------------------+
    <-->
3 rows in set (0.00 sec)
```

作为对比，c2 是可见的索引，优化器将可以使用索引：

```
EXPLAIN SELECT c2 FROM t1 ORDER BY c2;
```

```
+--
    <>----------------------+----------+-----------+-------------------------+                                    +
    <-->
| id                        | estRows  | task      | access object           | operator info
    <-->                 |
+--
    <>----------------------+----------+-----------+-------------------------+                                    +
    <-->
| IndexReader_13            | 10000.00 | root      |                         | index:IndexFullScan_12
    <-->          |
| └─IndexFullScan_12        | 10000.00 | cop[tikv] | table:t1, index:c2(c2) | keep order:true,
    <--> stats:pseudo |
+--    ----------------------+----------+-----------+-------------------------+                                    +
    <>                       +          +           +                         +
    <-->
```

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

即使用 SQL Hint USE INDEX 强制使用索引，优化器也无法使用不可见索引，否则 SQL 语句会报错：

```
SELECT * FROM t1 USE INDEX(c1);
```

```
ERROR 1176 (42000): Key 'c1' doesn't exist in table 't1'
```

> 注意：
> “不可见”是仅仅对优化器而言的，不可见索引仍然可以被修改或删除。

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.02 sec)
```

4.8.2.12.3 MySQL 兼容性

- TiDB 中的不可见索引是基于 MySQL 8.0 中的同等特性构建的。
- 与 MySQL 类似，TiDB 不允许将主键索引设为不可见。
- MySQL 中提供的优化器开关 use_invisible_indexes=on 可将所有的不可见索引重新设为可见。该功能在 TiDB 中不可用。

4.8.2.12.4 另请参阅

- CREATE TABLE
- CREATE INDEX
- ADD INDEX
- DROP INDEX
- RENAME INDEX

4.8.2.13 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。

4.8.2.13.1 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。

4.8.2.13.2 语法图

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

InstanceOption ::=
    'RELOAD' 'TLS' ('NO' 'ROLLBACK' 'ON' 'ERROR')?
```

4.8.2.13.3 示例

```
ALTER INSTANCE RELOAD TLS;
```

4.8.2.13.4 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。

4.8.2.13.5 另请参阅

为TiDB 客户端服务端间通信开启加密传输

4.8.2.14 ALTER PLACEMENT POLICY

`ALTER PLACEMENT POLICY` 用于修改已创建的放置策略。此修改会自动更新至所有使用这些放置策略的表和分区。

`ALTER PLACEMENT POLICY` 会完全替换之前定义的规则，而不会和之前的规则合并，比如在下面的例子中，`FOLLOWERS=4` 就被 `ALTER PLACEMENT POLICY` 语句覆盖了：

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=4;
ALTER PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1";
```

4.8.2.14.1 语法图

```
AlterPolicyStmt ::=
    "ALTER" "PLACEMENT" "POLICY" IfExists PolicyName PlacementOptionList

PolicyName ::=
    Identifier

PlacementOptionList ::=
    PlacementOption
|   PlacementOptionList PlacementOption
|   PlacementOptionList ',' PlacementOption

PlacementOption ::=
```

```
    CommonPlacementOption
|   SugarPlacementOption
|   AdvancedPlacementOption

CommonPlacementOption ::=
    "FOLLOWERS" EqOpt LengthNum

SugarPlacementOption ::=
    "PRIMARY_REGION" EqOpt stringLit
|   "REGIONS" EqOpt stringLit
|   "SCHEDULE" EqOpt stringLit

AdvancedPlacementOption ::=
    "LEARNERS" EqOpt LengthNum
|   "CONSTRAINTS" EqOpt stringLit
|   "LEADER_CONSTRAINTS" EqOpt stringLit
|   "FOLLOWER_CONSTRAINTS" EqOpt stringLit
|   "LEARNER_CONSTRAINTS" EqOpt stringLit
```

4.8.2.14.2 示例

> **注意：**
>
> 如要查看所在集群中可用的区域，见SHOW PLACEMENT LABELS。如果未看到任何可用的区域，此 TiKV 集群在部署时可能未正确设置标签(label)。

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1";
CREATE TABLE t1 (i INT) PLACEMENT POLICY=p1; -- 绑定放置策略 p1 到表 t1。
ALTER PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1,us-west-2"
    <-> FOLLOWERS=4; -- t1 上的放置规则会自动更新。
SHOW CREATE PLACEMENT POLICY p1\G;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

***************************[ 1. row ]***************************
Policy        | p1
Create Policy | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-
    <-> west-1,us-west-2" FOLLOWERS=4
1 row in set (0.00 sec)
```

4.8.2.14.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.14.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- DROP PLACEMENT POLICY

4.8.2.15 ALTER RESOURCE GROUP

`ALTER RESOURCE GROUP` 语句用于在当前所选数据库中修改资源组。

4.8.2.15.1 语法图

```
AlterResourceGroupStmt ::=
   "ALTER" "RESOURCE" "GROUP" IfExists ResourceGroupName ResourceGroupOptionList

IfExists ::=
    ('IF' 'EXISTS')?

ResourceGroupName ::=
   Identifier

ResourceGroupOptionList ::=
    DirectResourceGroupOption
|   ResourceGroupOptionList DirectResourceGroupOption
|   ResourceGroupOptionList ',' DirectResourceGroupOption

DirectResourceGroupOption ::=
    "RU_PER_SEC" EqOpt stringLit
|   "PRIORITY" EqOpt ResourceGroupPriorityOption
|   "BURSTABLE"

ResourceGroupPriorityOption ::=
    LOW
|   MEDIUM
|   HIGH
```

TiDB 支持以下 `DirectResourceGroupOption`, 其中Request Unit (RU) 是TiDB 对 CPU、 IO 等系统资源统一抽象的单位。

参数	含义	举例
`RU_PER_SEC`	每秒RU 填充的速度	`RU_PER_SEC = 500` 表示此资源组每秒回填 500 个 RU。

参数	含义	举例
PRIORITY	任务在 TiKV 上处理的绝对优先级	PRIORITY = HIGH 表示优先级高。若未指定则默认为 MEDI
BURSTABLE	允许对应的资源组超出配额后使用空余的系统资源。	

> **注意：**
>
> ALTER RESOURCE GROUP 语句只能在全局变量tidb_enable_resource_control 参数设置为 ON 时才能执行。

4.8.2.15.2 示例

创建一个名为 rg1 的资源组，并修改它的属性。

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1
  RU_PER_SEC = 100
  BURSTABLE;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |        100 | MEDIUM   | YES       |
+------+------------+----------+-----------+
1 rows in set (1.30 sec)
```

```
ALTER RESOURCE GROUP rg1
  RU_PER_SEC = 200
  PRIORITY = LOW;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
+------+------------+----------+-------------------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-------------------+
| rg1  |        200 | LOW      | NO        |
+------+------------+----------+-------------------+
1 rows in set (1.30 sec)
```

4.8.2.15.3 MySQL 兼容性

MySQL 也支持ALTER RESOURCE GROUP，但是接受的参数和TiDB 不同，两者并不兼容。

4.8.2.15.4 另请参阅

- DROP RESOURCE GROUP
- CREATE RESOURCE GROUP
- RU

4.8.2.16 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
- COMPACT 表数据

4.8.2.16.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName (
        AlterTableSpecListOpt AlterTablePartitionOpt |
        'ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )? AnalyzeOptionListOpt |
        'COMPACT' ( 'PARTITION' PartitionNameList )? 'TIFLASH' 'REPLICA'
    )

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'
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'SHARD_ROW_ID_BITS' ) EqOpt
    <⟷ LengthNum
|   ( 'CACHE' | 'NOCACHE' )
|   (
        'TTL' EqOpt TimeColumnName '+' 'INTERVAL' Expression TimeUnit (TTLEnable EqOpt ( 'ON' | '
            <⟷ OFF' ))?
        | 'REMOVE' 'TTL'
        | TTLEnable EqOpt ( 'ON' | 'OFF' )
        | TTLJobInterval EqOpt stringLit
    )
|   PlacementPolicyOption

PlacementPolicyOption ::=
    "PLACEMENT" "POLICY" EqOpt PolicyName
|   "PLACEMENT" "POLICY" (EqOpt | "SET") "DEFAULT"
```

4.8.2.16.2 示例

创建一张表，并插入初始数据：

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

执行以下查询需要扫描全表，因为 c1 列未被索引：

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+--
    <-> ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    <->
| id                        | estRows  | task      | access object | operator info
    <->                  |
+--
    <> ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    <->
| TableReader_7             | 10.00    | root      |               | data:Selection_6
    <->               |
| └─Selection_6             | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    <->              |
|   └─TableFullScan_5       | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    <> pseudo |
+--
    <> ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    <->
3 rows in set (0.00 sec)
```

你可以使用ALTER TABLE .. ADD INDEX 语句在 t1 表上添加索引。添加后，EXPLAIN 的分析结果显示 SELECT * <-> FROM t1 WHERE c1 = 3; 查询已使用效率更高的索引范围扫描：

```
ALTER TABLE t1 ADD INDEX (c1);
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
Query OK, 0 rows affected (0.30 sec)
+--
    <->                      +          +           +                          +
    <->
| id                        | estRows | task      | access object           | operator info
    <->                                |
+--
    <->                      +          +           +                          +
    <>----------------------  ----------  -----------  ------------------------
| IndexReader_6             | 10.00    | root      |                         | index:IndexRangeScan_5
    <->                       |
```

```
| └─IndexRangeScan_5      | 10.00    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    <-> order:false, stats:pseudo |
+--
    <->                          +          +            +                          +
    <->
2 rows in set (0.00 sec)
```

TiDB 允许用户为 DDL 操作指定使用某一种 ALTER 算法。这仅为一种指定，并不改变实际的用于更改表的算法。如果你只想在群集的高峰时段允许即时 DDL 更改，则 ALTER 算法会很有用。示例如下：

```
ALTER TABLE t1 DROP INDEX c1, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.24 sec)
```

如果某一 DDL 操作要求使用 INPLACE 算法，而用户指定 ALGORITHM=INSTANT，会导致报错：

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=INSTANT;
```

```
ERROR 1846 (0A000): ALGORITHM=INSTANT is not supported. Reason: Cannot alter table by INSTANT.
    <-> Try ALGORITHM=INPLACE.
```

但如果为 INPLACE 操作指定 ALGORITHM=COPY，会产生警告而非错误，这是因为 TiDB 将该指定解读为该算法或更好的算法。由于TiDB 使用的算法可能不同于 MySQL，所以这一行为可用于 MySQL 兼容性。

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=COPY;
SHOW WARNINGS;
```

```
Query OK, 0 rows affected, 1 warning (0.25 sec)
+--
    <->-------+-------+
    <->
| Level | Code | Message
    <->                                                                                |
+--
    <->       +       +
    <->
| Error | 1846 | ALGORITHM=COPY is not supported. Reason: Cannot alter table by COPY. Try
    <-> ALGORITHM=INPLACE. |
+--
    <->-------+-------+
    <->
1 row in set (0.00 sec)
```

4.8.2.16.3 MySQL 兼容性

TiDB 中的 ALTER TABLE 语法主要存在以下限制：

- 使用 ALTER TABLE 语句修改一个表的多个模式对象（如列、索引）时：
  - 不允许在多个更改中指定同一个模式对象。
  - TiDB 根据执行前的表结构检查合法性。例如 ALTER TABLE t ADD COLUMN c1 INT, ADD COLUMN c2 <⟶ INT AFTER c1; 会报错，因为表结构中不存在名字为 c1 的表。
  - TiDB 的执行顺序是从左往右逐个执行更改，该行为在个别场景下和 MySQL 不兼容。
- 不支持主键列上Reorg-Data 类型的变更。
- 不支持分区表上的列类型变更。
- 不支持生成列上的列类型变更。
- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为TiDB 中的 CAST 函数与 MySQL 的行为存在兼容性问题。
- 不支持空间数据类型。
- ALTER TABLE t CACHE | NOCACHE 不是 MySQL 标准语法，而是TiDB 扩展功能，参见缓存表。

其它限制可参考：TiDB 中 DDL 语句与 MySQL 的兼容性情况。

4.8.2.16.4 另请参阅

- 与 MySQL 兼容性对比
- ALTER TABLE ··· COMPACT
- ADD COLUMN
- DROP COLUMN
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

4.8.2.17 ALTER TABLE ··· COMPACT

TiDB 存储节点在后台会自动发起数据整理（Compaction）。数据整理时， 表中的物理数据会被重写， 如清理已删除的数据、合并多版本数据等，从而可以获得更高的访问性能，并减少磁盘空间占用。使用 ALTER TABLE ... COMPACT 语句可以立即对指定的表进行数据整理，而无需等待后台触发。

该语句执行时不会阻塞现有 SQL 语句的执行或TiDB 功能的使用，包括事务、DDL、GC 等，也不会改变通过 SQL 语句访问获得的数据内容。该语句执行时会消耗一定量的IO 及 CPU 资源，请注意选择合适的时机执行，如资源空闲时段，避免对业务造成负面影响。

该语句会等待表中所有副本都数据整理完毕后才结束运行并返回。在执行过程中，你可以通过KILL 语句安全地中断本张表的数据整理过程。中断不会破坏数据一致性或丢失数据，也不会影响后续重新发起或自动触发后台数据整理。

目前该语句仅支持对TiFlash 进行数据整理，不支持对TiKV 进行数据整理。

4.8.2.17.1 语法图

```
AlterTableCompactStmt ::=
    'ALTER' 'TABLE' TableName 'COMPACT' ( 'PARTITION' PartitionNameList )? ( 'TIFLASH' 'REPLICA'
        <↪ )?
```

自 v6.2.0 起，语法中 TIFLASH REPLICA 部分可以被省略。省略后语句含义不变，同样只对TiFlash 列存有效。

4.8.2.17.2 示例

对表中TiFlash 副本进行数据整理

假设目前有一张 employees 表具有 4 个分区，且具有 2 个TiFlash 副本：

```
CREATE TABLE employees
    ( id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);

ALTER TABLE employees SET TIFLASH REPLICA 2;
```

执行以下语句可对 employees 表上所有分区的 2 个TiFlash 副本立即进行数据整理：

```
ALTER TABLE employees COMPACT TIFLASH REPLICA;
```

对分区表中指定分区的TiFlash 副本进行数据整理

假设目前有一张 employees 表具有 4 个分区，且具有 2 个TiFlash 副本：

```
CREATE TABLE employees
    ( id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);

ALTER TABLE employees SET TIFLASH REPLICA 2;
```

执行以下语句可对 employees 表上 pNorth、pEast 这两个分区的 2 个TiFlash 副本立即进行数据整理：

```
ALTER TABLE employees COMPACT PARTITION pNorth, pEast TIFLASH REPLICA;
```

4.8.2.17.3 并发度

ALTER TABLE ... COMPACT 语句会同时对表中所有副本发起数据整理操作。

为了避免数据整理操作对在线业务造成显著影响，在同一时间，每个TiFlash 实例默认只会处理一张表的数据整理操作（后台自动触发的数据整理除外）。这意味着，若你同时对多张表执行 ALTER TABLE ... COMPACT 语句，则它们在同一个TiFlash 实例上会排队依次执行，而非同时执行。

你可以修改TiFlash 配置文件参数manual_compact_pool_size，以更大资源占用为代价、获得更大的表级别并发度。例如，参数指定为 2 时，可以同时对 2 张表进行数据整理。

4.8.2.17.4 观测数据整理进度

你可以通过 INFORMATION_SCHEMA.TIFLASH_TABLES 表中 TOTAL_DELTA_ROWS 列来观测TiFlash 存储引擎上数据整理的进度，或判断是否有必要对某张表发起数据整理。TOTAL_DELTA_ROWS 的值越大，说明还能被整理的数据越多；若 TOTAL_DELTA_ROWS 为 0，说明表中所有数据都处于最佳状态，无需整理。

示例：查询普通表的数据整理状态

```
USE test;

CREATE TABLE foo(id INT);

ALTER TABLE foo SET TIFLASH REPLICA 1;

SELECT TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS FROM INFORMATION_SCHEMA.TIFLASH_TABLES
    WHERE IS_TOMBSTONE = 0 AND
    `TIDB_DATABASE` = "test" AND `TIDB_TABLE` = "foo";
+------------------+--------------------+
| TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+------------------+--------------------+
|                0 |                  0 |
+------------------+--------------------+

INSERT INTO foo VALUES (1), (3), (7);

SELECT TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS FROM INFORMATION_SCHEMA.TIFLASH_TABLES
    WHERE IS_TOMBSTONE = 0 AND
    `TIDB_DATABASE` = "test" AND `TIDB_TABLE` = "foo";
+------------------+--------------------+
| TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+------------------+--------------------+
|                3 |                  0 |
+------------------+--------------------+
```

```
-- 新写入的数据可被整理

ALTER TABLE foo COMPACT TIFLASH REPLICA;

SELECT TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS FROM INFORMATION_SCHEMA.TIFLASH_TABLES
    WHERE IS_TOMBSTONE = 0 AND
    `TIDB_DATABASE` = "test" AND `TIDB_TABLE` = "foo";
+------------------+-------------------+
| TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+------------------+-------------------+
|                0 |                 3 |
+------------------+-------------------+
-- 所有数据都处于最佳整理状态
```

示例：查询分区表的数据整理状态

```
USE test;

CREATE TABLE employees
    (id INT NOT NULL, store_id INT)
    PARTITION BY LIST (store_id) (
        PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
        PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
        PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
        PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
    );

ALTER TABLE employees SET TIFLASH REPLICA 1;

INSERT INTO employees VALUES (1, 1), (6, 6), (10, 10);

SELECT PARTITION_NAME, TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS
    FROM INFORMATION_SCHEMA.TIFLASH_TABLES t, INFORMATION_SCHEMA.PARTITIONS p
    WHERE t.IS_TOMBSTONE = 0 AND t.TABLE_ID = p.TIDB_PARTITION_ID AND
    p.TABLE_SCHEMA = "test" AND p.TABLE_NAME = "employees";
+----------------+------------------+-------------------+
| PARTITION_NAME | TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+----------------+------------------+-------------------+
| pNorth         |                1 |                 0 |
| pEast          |                2 |                 0 |
| pWest          |                0 |                 0 |
| pCentral       |                0 |                 0 |


+----------------+------------------+-------------------+
-- 一些分区上有数据可被整理

ALTER TABLE employees COMPACT TIFLASH REPLICA;
```

```
SELECT PARTITION_NAME, TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS
    FROM INFORMATION_SCHEMA.TIFLASH_TABLES t, INFORMATION_SCHEMA.PARTITIONS p
    WHERE t.IS_TOMBSTONE = 0 AND t.TABLE_ID = p.TIDB_PARTITION_ID AND
    p.TABLE_SCHEMA = "test" AND p.TABLE_NAME = "employees";
+----------------+------------------+--------------------------------+
| PARTITION_NAME | TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+----------------+------------------+--------------------------------+
| pNorth         |                0 |                              1 |
| pEast          |                0 |                              2 |
| pWest          |                0 |                              0 |
| pCentral       |                0 |                              0 |
+----------------+------------------+--------------------------------+
-- 所有分区上的数据都处于最佳整理状态
```

注意：

- 若数据整理的过程中发生了数据更新，你可能观察到数据整理完毕后 `TOTAL_DELTA_ROWS` 仍为非零值。这是正常现象，表明这些更新部分没有被整理到。若你想对这部分更新数据也进行整理，可再次执行 `ALTER TABLE ... COMPACT` 语句。
- `TOTAL_DELTA_ROWS` 的单位是数据版本数，而非数据行数。例如，插入一行记录再删除该行后，`TOTAL_DELTA_ROWS` 会增加 2。

4.8.2.17.5 兼容性

MySQL 兼容性

`ALTER TABLE ... COMPACT` 语法是TiDB 引入的对标准 SQL 语法的扩展。尽管没有对应的 MySQL 语法，但你仍然可通过 MySQL 各版本客户端，或各个遵循 MySQL 协议的数据库驱动执行该语句。

TiDB Binlog 及 TiCDC 兼容性

`ALTER TABLE ... COMPACT` 语句不会导致逻辑数据变化，因而不会被TiDB Binlog 及 TiCDC 同步到下游。

4.8.2.17.6 另请参阅

- ALTER TABLE
- KILL

4.8.2.18 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。

4.8.2.18.1 语法图

```
AlterUserStmt ::=
    'ALTER' 'USER' IfExists (UserSpecList RequireClauseOpt ConnectionOptions PasswordOption
        <⟷ LockOption AttributeOption | 'USER' '(' ')' 'IDENTIFIED' 'BY' AuthString)
        <⟷ ResourceGroupNameOption

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

UserSpec ::=
    Username AuthOption

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces

AuthOption ::=
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        <⟷ AuthString | 'AS' HashString )? ) )?

PasswordOption ::= ( 'PASSWORD' 'EXPIRE' ( 'DEFAULT' | 'NEVER' | 'INTERVAL' N 'DAY' )? | '
    <⟷ PASSWORD' 'HISTORY' ( 'DEFAULT' | N ) | 'PASSWORD' 'REUSE' 'INTERVAL' ( 'DEFAULT' | N '
    <⟷ DAY' ) | 'FAILED_LOGIN_ATTEMPTS' N | 'PASSWORD_LOCK_TIME' ( N | 'UNBOUNDED' ) )*

LockOption ::= ( 'ACCOUNT' 'LOCK' | 'ACCOUNT' 'UNLOCK' )?

AttributeOption ::= ( 'COMMENT' CommentString | 'ATTRIBUTE' AttributeString )?

ResourceGroupNameOption::= ( 'RESOURCE' 'GROUP' Identifier)?
```

4.8.2.18.2 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)
```

修改用户基本信息

修改用户 newuser 的密码：

```
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)
```

锁定用户 newuser：

```
ALTER USER 'newuser' ACCOUNT LOCK;
```

```
Query OK, 0 rows affected (0.02 sec)
```

修改 newuser 的属性：

```
ALTER USER 'newuser' ATTRIBUTE '{"newAttr": "value", "deprecatedAttr": null}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+---------------------------+
| USER      | HOST | ATTRIBUTE                 |
+-----------+------+---------------------------+
| newuser   | %    | {"newAttr": "value"}      |
+-----------+------+---------------------------+
1 rows in set (0.00 sec)
```

通过 ALTER USER ... COMMENT 修改用户 newuser 的注释：

```
ALTER USER 'newuser' COMMENT 'Here is the comment';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                                 |
+-----------+------+-----------------------------------------------------------+
| newuser   | %    | {"comment": "Here is the comment", "newAttr": "value"} |
+-----------+------+-----------------------------------------------------------+
1 rows in set (0.00 sec)
```

通过 ALTER USER ... ATTRIBUTE 删除用户 newuser 的注释：

```
ALTER USER 'newuser' ATTRIBUTE '{"comment": null}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+---------------------------+
| USER      | HOST | ATTRIBUTE                 |
+-----------+------+---------------------------+
| newuser   | %    | {"newAttr": "value"}      |
+-----------+------+---------------------------+
1 rows in set (0.00 sec)
```

通过 ALTER USER ... PASSWORD EXPIRE NEVER 修改用户 newuser 的自动密码过期策略为永不过期：

```
ALTER USER 'newuser' PASSWORD EXPIRE NEVER;
```

```
Query OK, 0 rows affected (0.02 sec)
```

通过 ALTER USER ... PASSWORD REUSE INTERVAL ... DAY 修改用户 newuser 的密码重用策略为不允许重复使用最近 90 天内使用过的密码：

```
ALTER USER 'newuser' PASSWORD REUSE INTERVAL 90 DAY;
```

```
Query OK, 0 rows affected (0.02 sec)
```

修改用户绑定的资源组

通过 ALTER USER ... RESOURCE GROUP 修改用户 newuser 的资源组到 rg1：

```
ALTER USER 'newuser' RESOURCE GROUP rg1;
```

```
Query OK, 0 rows affected (0.02 sec)
```

查看当前用户绑定的资源组：

```
SELECT USER, JSON_EXTRACT(User_attributes, "$.resource_group") FROM mysql.user WHERE user = "
    <-> newuser";
```

```
+----------+-----------------------------------------------------+
| USER     | JSON_EXTRACT(User_attributes, "$.resource_group") |
+----------+-----------------------------------------------------+
| newuser | "rg1"                                                |
+----------+-----------------------------------------------------+
1 row in set (0.02 sec)
```

取消用户绑定的资源组，即将用户绑定的资源组设置为空。取消后，用户会绑定到 default 资源组。

```
ALTER USER 'newuser' RESOURCE GROUP ``;
SELECT USER, JSON_EXTRACT(User_attributes, "$.resource_group") FROM mysql.user WHERE user = "
    <-> newuser";
```

```
+----------+-----------------------------------------------------+
| USER     | JSON_EXTRACT(User_attributes, "$.resource_group") |
+----------+-----------------------------------------------------+
| newuser | ""                                                   |
+----------+-----------------------------------------------------+
1 row in set (0.02 sec)
```

4.8.2.18.3 另请参阅

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

4.8.2.19 ANALYZE

ANALYZE 语句用于更新TiDB在表和索引上留下的统计信息。执行大批量更新或导入记录后，或查询执行计划不是最佳时，建议运行 ANALYZE。

当TiDB 逐渐发现这些统计数据与预估不一致时，也会自动更新其统计数据。

目前TiDB 收集统计信息分为全量收集和增量收集两种方式，分别通过 ANALYZE TABLE 和 ANALYZE INCREMENTAL <-> TABLE 语句来实现。关于这两种语句的详细使用方式，可参考统计信息简介。

4.8.2.19.1 语法图

```
AnalyzeTableStmt ::=
    'ANALYZE' ( 'TABLE' ( TableNameList ( 'ALL COLUMNS' | 'PREDICATE COLUMNS' ) | TableName ( '
        <-> INDEX' IndexNameList? | AnalyzeColumnOption | 'PARTITION' PartitionNameList ( 'INDEX'
        <->  IndexNameList? | AnalyzeColumnOption )? )? ) | 'INCREMENTAL' 'TABLE' TableName ( '
        <-> PARTITION' PartitionNameList )? 'INDEX' IndexNameList? ) AnalyzeOptionListOpt
```

```
AnalyzeOptionListOpt ::=
( WITH AnalyzeOptionList )?

AnalyzeOptionList ::=
AnalyzeOption ( ',' AnlyzeOption )*

AnalyzeOption ::=
( NUM ( 'BUCKETS' | 'TOPN' | ( 'CMSKETCH' ( 'DEPTH' | 'WIDTH' ) ) | 'SAMPLES' ) ) | ( FLOATNUM '
    <⟶ SAMPLERATE' )

AnalyzeColumnOption ::=
( 'ALL COLUMNS' | 'PREDICATE COLUMNS' | 'COLUMNS' ColumnNameList )

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?

ColumnNameList ::=
    Identifier ( ',' Identifier )*

IndexNameList ::=
    Identifier ( ',' Identifier )*

PartitionNameList ::=
    Identifier ( ',' Identifier )*
```

4.8.2.19.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+                         +          +            +                         +----------------------------------------------------------------
    <->
| id                      | estRows | task       | access object           | operator info
    <->                               |
+-------------------------+----------+------------+-------------------------+                                                                ----------
    <->
| IndexReader_6           | 10.00    | root       |                         | index:IndexRangeScan_5
    <->                       |
| └─IndexRangeScan_5      | 10.00    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    <-> order:false, stats:pseudo |
+                         +          +            +                         +----------------------------------------------------------------
    <->
2 rows in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+---------+-----------+-----------------------+ ------------------------------------------------------------------+
    <->
| id                      | estRows | task      | access object         | operator info
    <->                |
+-------------------------+---------+-----------+-----------------------+ ------------------------------------------------------------------+
    <->
| IndexReader_6           | 1.00    | root      |                       | index:IndexRangeScan_5
    <->       |
| └─IndexRangeScan_5      | 1.00    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    <-> order:false |
+-------------------------+---------+-----------+-----------------------+ ------------------------------------------------------------------+
    <->
2 rows in set (0.00 sec)
```

4.8.2.19.3 MySQL 兼容性

- ANALYZE TABLE 在语法上与 MySQL 类似。但 ANALYZE TABLE 在 TiDB 上的执行时间可能长得多，因为它的内部运行方式不同。
- 在 MySQL 上不支持 ANALYZE INCREMENTAL TABLE 语句，它的使用可参考增量收集文档。

TiDB 与 MySQL 在以下方面存在区别：所收集的统计信息，以及查询执行过程中统计信息是如何被使用的。虽然 TiDB 中的 ANALYZE 语句在语法上与 MySQL 类似，但存在以下差异：

- 执行ANALYZE TABLE时，TiDB可能不包含最近提交的更改。若对行进行了批量更改，在执行ANALYZE TABLE之前，你可能需要先执行 `sleep(1)`，这样统计信息更新才能反映这些更改。参见 #16570。
- ANALYZE TABLE在TiDB中的执行时间比在MySQL中的执行时间要长得多。但你可以通过执行 `SET GLOBAL <-> tidb_enable_fast_analyze=1` 来启用快速分析，这样能部分抵消这种执行上的性能差异。快速分析利用了采样，会导致统计信息的准确性降低。因此快速分析仍是一项实验特性。

4.8.2.19.4 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE

4.8.2.20 BACKUP

BACKUP语句用于对TiDB集群执行分布式备份操作。

BACKUP语句使用的引擎与 BR 相同，但备份过程是由TiDB本身驱动，而非单独的 BR 工具。BR工具的优势和警告也适用于 BACKUP 语句。

执行 BACKUP 需要 BACKUP_ADMIN 或 SUPER 权限。此外，执行备份的 TiDB 节点和集群中的所有TiKV节点都必须有对目标存储的读或写权限。

BACKUP语句开始执行后将会被阻塞，直到整个备份任务完成、失败或取消。因此，执行 BACKUP 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和RESTORE 任务。如果TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 BACKUP 将等待前面所有的任务完成后再执行。

4.8.2.20.1 语法图

```
BackupStmt ::=
    "BACKUP" BRIETables "TO" stringLit BackupOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

BackupOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean
|   "LAST_BACKUP" '='? BackupTSO
|   "SNAPSHOT" '='? ( BackupTSO | LengthNum TimestampUnit "AGO" )

Boolean ::=
    NUM | "TRUE" | "FALSE"

BackupTSO ::=
```

```
    LengthNum | stringLit
```

4.8.2.20.2 示例

备份数据库

```
BACKUP DATABASE `test` TO 'local:///mnt/backup/2020/04/';
```

```
+--
    <-->----------------------------+-----------+-----------------+---------------------+                +
    <-->
| Destination                    | Size      | BackupTS        | Queue Time          | Execution
    <--> Time       |
+--
    <-->----------------------------+-----------+-----------------+---------------------+                +
    <-->
| local:///mnt/backup/2020/04/ | 248665063 | 416099531454472 | 2020-04-12 23:09:48 | 2020-04-12
    <--> 23:09:48 |
+--
    <>-----------------------------+-----------+-----------------+---------------------+                +
    <-->
1 row in set (58.453 sec)
```

上述示例中，test 数据库被备份到本地， 数据以 SST 文件的形式存储在分布于所有 TiDB 和 TiKV 节点的 /mnt/backup/2020/04/ 目录中。

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL
Size	备份文件的总大小，单位为字节
BackupTS	创建备份时的快照 TSO（用于增量备份）
Queue Time	BACKUP 任务开始排队的时间戳（当前时区）
Execution Time	BACKUP 任务开始执行的时间戳（当前时区）

备份表

```
BACKUP TABLE `test`.`sbtest01` TO 'local:///mnt/backup/sbtest01/';
```

```
BACKUP TABLE sbtest02, sbtest03, sbtest04 TO 'local:///mnt/backup/sbtest/';
```

备份集群

```
BACKUP DATABASE * TO 'local:///mnt/backup/full/';
```

注意，备份中不包含系统表（mysql.*、INFORMATION_SCHEMA.*、PERFORMANCE_SCHEMA.* 等）。

外部存储

BR 支持备份数据到Amazon S3 或 Google Cloud Storage (GCS)：

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-05/?access-key={YOUR_ACCESS_KEY}&
    ↪ secret-access-key={YOUR_SECRET_KEY}';
```

有关详细的 URL 语法，见外部存储 URI 格式。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-05/'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个TiKV 节点的平均上传速度。

默认情况下，每个TiKV 节点上运行 4 个备份线程。可以通过 CONCURRENCY 选项来调整这个值。

在备份完成之前，BACKUP 将对集群上的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 8
    CHECKSUM = FALSE;
```

快照

可以指定一个时间戳、TSO 或相对时间，来备份历史数据。

```
-- 相对时间
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist01'
    SNAPSHOT = 36 HOUR AGO;
-- 时间戳（当前时区）
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist02'
    SNAPSHOT = '2020-04-01 12:00:00';
-- TSO
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist03'
    SNAPSHOT = 415685305958400;
```

对于相对时间，支持以下时间单位：

- MICROSECOND（微秒）
- SECOND（秒）
- MINUTE（分钟）
- HOUR（小时）
- DAY（天）
- WEEK（周）

注意，相对时间的单位遵循 SQL 标准，永远使用单数。

增量备份

提供 LAST_BACKUP 选项，只备份从上一次备份到当前快照之间的增量数据。

```
-- 时间戳（当前时区）
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist02'
    LAST_BACKUP = '2020-04-01 12:00:00';

-- TSO
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist03'
    LAST_BACKUP = 415685305958400;
```

4.8.2.20.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.20.4 另请参阅

- RESTORE
- SHOW BACKUPS

4.8.2.21 BATCH

BATCH 语句将一个 DML 语句拆成多个语句在内部执行，因此不保证事务的原子性和隔离性，是一个“非事务”语句。

目前 BATCH 语句支持 INSERT、 REPLACE、 UPDATE、 DELETE。

BATCH 语句在某一列将 DML 语句涉及的范围划分为多个区间，在每个区间执行一条 SQL。

详细的说明和使用限制见非事务语句。

在涉及多表join 时，BATCH 语法中指定拆分列时需要指明完整的路径以避免歧义，如：

```
BATCH ON test.t2.id LIMIT 1 INSERT INTO t SELECT t2.id, t2.v, t3.v FROM t2 JOIN t3 ON t2.k = t3.k
    ↩;
```

上面这条语句的拆分列用 test.t2.id 指明，不具有歧义。如果写成如下 id 的形式，则会报错：

```
BATCH ON id LIMIT 1 INSERT INTO t SELECT t2.id, t2.v, t3.v FROM t2 JOIN t3 ON t2.k = t3.k;

Non-transactional DML, shard column must be fully specified
```

4.8.2.21.1 语法图

```
NonTransactionalDMLStmt ::=
    'BATCH' ( 'ON' ColumnName )? 'LIMIT' NUM DryRunOptions? ShardableStmt

DryRunOptions ::=
    'DRY' 'RUN' 'QUERY'?

ShardableStmt ::=
    DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
```

4.8.2.21.2 MySQL 兼容性

BATCH 语句是TiDB 独有的语句，与 MySQL 不兼容。

4.8.2.21.3 另请参阅

- 非事务语句

4.8.2.22 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。

4.8.2.22.1 语法图

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( 'WITH' 'TIMESTAMP' 'BOUND' TimestampBound
    <⟶  )? ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' )?
```

4.8.2.22.2 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)
```

4.8.2.22.3 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。

4.8.2.22.4 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION
- TiDB 乐观事务模型
- TiDB 悲观事务模式

4.8.2.23 CALIBRATE RESOURCE

`CALIBRATE RESOURCE` 语句用于预估并输出当前集群的Request Unit (RU) 的容量。

4.8.2.23.1 语法图

```
CalibrateResourceStmt ::= 'CALIBRATE' 'RESOURCE' CalibrateOption

CalibrateOption ::=
('START_TIME' 'TIMESTAMP' ('DURATION' stringLit | 'END_TIME' 'TIMESTAMP')?)
| ('WORKLOAD' ('TPCC' | 'OLTP_READ_WRITE' | 'OLTP_READ_ONLY' | 'OLTP_WRITE_ONLY'))?
```

4.8.2.23.2 预估方式

TiDB 提供两种预估方式：

根据实际负载估算容量

如果应用已经在线上运行，或者你能够运行实际业务测试，建议利用一段时间的实际负载来预估总容量。为了提高预估准确性，需要遵守以下约束条件：

- 使用 `START_TIME` 参数指定预估开始的时间点，格式为 `2006-01-02 15:04:05`，默认预估结束时间为当前时间。
- 指定完成 `START_TIME` 参数后，可以使用 `END_TIME` 参数指定预估结束时间，或者使用 `DURATION` 参数指定距离 `START_TIME` 的预估时间窗口。

- 时间窗口范围为 10 分钟至 24 小时。
- 在预估的时间窗口内，TiDB 与TiKV 的 CPU 利用率不能过低，否则无法进行容量估算。

基于硬件部署估算容量

这种方式主要根据当前的集群配置，结合对不同负载观测的经验值进行预估。由于不同类型的负载对硬件的配比要求不同，相同配置的硬件所输出的容量也会有所不同。这里的 WORKLOAD 参数提供了以下不同的负载类型供选择，默认为 TPCC：

- TPCC：数据写入较重的负载，根据类似 TPC-C 的负载模型预测。
- OLTP_WRITE_ONLY：数据写入较重的负载，根据类似 sysbench oltp_write_only 的负载模型预测。
- OLTP_READ_WRITE：数据读写平衡的负载，根据类似 sysbench oltp_read_write 的负载模型预测。
- OLTP_READ_ONLY：数据读取较重的负载，根据类似 sysbench oltp_read_only 的负载模型预测。

> 注意：
>
> 集群RU 的容量会随集群的拓扑结构和各个组件软硬件配置的变化而变化，每个集群实际能消耗的RU 还与实际的负载相关。基于硬件部署估算容量的预估值仅供参考，可能会与实际的最大值存在偏差。建议根据实际负载估算容量。

4.8.2.23.3 权限

执行此命令依赖如下配置和权限：

- tidb_enable_resource_control 需要处于开启状态
- 需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限
- 需要拥有 METRICS_SCHEMA 库下所有表的 SELECT 权限

4.8.2.23.4 示例

指定初始时间 START_TIME 和时间窗口 DURATION 大小，根据实际负载查看RU 容量。

```
CALIBRATE RESOURCE START_TIME '2023-04-18 08:00:00' DURATION '20m';
+----------+
| QUOTA |
+----------+
| 27969 |
+----------+
1 row in set (0.01 sec)
```

指定初始时间 START_TIME 和结束时间 END_TIME，根据实际负载查看 RU 容量。

```
CALIBRATE RESOURCE START_TIME '2023-04-18 08:00:00' END_TIME '2023-04-18 08:20:00';
+----------+
| QUOTA |
```

```
+------------+
| 27969 |
+------------+
1 row in set (0.01 sec)
```

当时间窗口范围 DURATION 不满足 10 分钟至 24 小时的条件，会导致报错提醒。

```
CALIBRATE RESOURCE START_TIME '2023-04-18 08:00:00' DURATION '25h';
ERROR 1105 (HY000): the duration of calibration is too long, which could lead to inaccurate
    <-> output. Please make the duration between 10m0s and 24h0m0s
CALIBRATE RESOURCE START_TIME '2023-04-18 08:00:00' DURATION '9m';
ERROR 1105 (HY000): the duration of calibration is too short, which could lead to inaccurate
    <-> output. Please make the duration between 10m0s and 24h0m0s
```

当时间窗口范围内的负载过低，会导致报错提醒。

```
CALIBRATE RESOURCE START_TIME '2023-04-18 08:00:00' DURATION '60m';
ERROR 1105 (HY000): The workload in selected time window is too low, with which TiDB is unable to
    <->  reach a capacity estimation; please select another time window with higher workload, or
    <-> calibrate resource by hardware instead
```

指定 WORKLOAD 查看RU 容量，默认为 TPCC。

```
CALIBRATE RESOURCE;
+------------ +
| QUOTA |
+------------ +
| 190470 |
+------------ +
1 row in set (0.01 sec)

CALIBRATE RESOURCE WORKLOAD OLTP_WRITE_ONLY;
+------------ +
| QUOTA |
+------------ +
| 27444 |
+------------ +
1 row in set (0.01 sec)
```

4.8.2.24 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。从 v5.1.0 版本起，TiDB 开始支持Reorg 数据的类型变更，包括但不限于：

- 从 varchar 转换为bigint
- decimal 精度修改
- 从 varchar(10) 到 varchar(5) 的长度压缩

4.8.2.24.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName ChangeColumnSpec ( ',' ChangeColumnSpec )*

ChangeColumnSpec
         ::= 'CHANGE' ColumnKeywordOpt 'IF EXISTS' ColumnName ColumnName ColumnType ColumnOption*
             <⟶ ( 'FIRST' | 'AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               <⟶ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               <⟶ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
           | 'STORAGE' StorageMedia
           | 'AUTO_RANDOM' ( '(' LengthNum ')' )?

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

4.8.2.24.2 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 col4 BIGINT, CHANGE id id2 INT NOT NULL;
```

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

```
CREATE TABLE t (a int primary key);
ALTER TABLE t CHANGE COLUMN a a VARCHAR(10);
```

```
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (c1 INT, c2 INT, c3 INT) partition by range columns(c1) ( partition p0 values less
    <⟶  than (10), partition p1 values less than (maxvalue));
ALTER TABLE t CHANGE COLUMN c1 c1 DATETIME;
```

```
ERROR 8200 (HY000): Unsupported modify column: table is partition table
```

```
CREATE TABLE t (a INT, b INT as (a+1));
ALTER TABLE t CHANGE COLUMN b b VARCHAR(10);
```

```
ERROR 8200 (HY000): Unsupported modify column: column is generated
```

```
CREATE TABLE t (a DECIMAL(13, 7));
ALTER TABLE t CHANGE COLUMN a a DATETIME;
```

```
ERROR 8200 (HY000): Unsupported modify column: change from original type decimal(13,7) to
    <⟶ datetime is currently unsupported yet
```

4.8.2.24.3 MySQL 兼容性

- 不支持主键列上Reorg-Data 类型的变更。
- 不支持分区表上的列类型变更。
- 不支持生成列上的列类型变更。
- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为TiDB 中 CAST 函数与 MySQL 的行为存在兼容性问题。

4.8.2.24.4 另请参阅

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

4.8.2.25 CHANGE DRAINER

`CHANGE DRAINER` 语句用于修改集群中 Drainer 的状态信息。

> **注意：**
>
> Drainer 在正常运行时会自动上报状态到 PD，仅在 Drainer 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Drainer 状态信息。

4.8.2.25.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------------------------- |
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time       |
+----------|----------------|--------|--------------------|---------------------------------------- |
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-04-30 00:00:03 |
+----------|----------------|--------|--------------------|---------------------------------------- |
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------------------------- |
2 rows in set (0.00 sec)
```

可以看出 `drainer1` 已经超过一天没有更新状态，该 Drainer 处于异常状态，但是 State 仍然为 `Online`，使用 `CHANGE DRAINER` 将该 Drainer 状态修改为 `paused`：

```
CHANGE DRAINER TO NODE_STATE ='paused' FOR NODE_ID 'drainer1';
```

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

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------------------------- |
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time       |
+----------|----------------|--------|--------------------|---------------------------------------- |
| drainer1 | 127.0.0.3:8249 | Paused | 408553768673342532 | 2019-04-30 00:00:03 |
```

```
+----------|----------------|--------|--------------------|--------------------------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|--------------------------------------|
2 rows in set (0.00 sec)
```

4.8.2.25.2 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.25.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS

4.8.2.26 CHANGE PUMP

CHANGE PUMP 语句用于修改集群中 Pump 的状态信息。

> **注意：**
>
> Pump 在正常运行时会自动上报状态到 PD，仅在 Pump 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Pump 状态信息。

4.8.2.26.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|--------------------------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time       |
+--------|----------------|--------|--------------------|--------------------------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|--------------------------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|--------------------------------------|
2 rows in set (0.00 sec)
```

可以看出 pump1 已经超过一天没有更新状态， 该 Pump 处于异常状态， 但是 State 仍然为 Online， 使用 CHANGE PUMP 将该 Pump 状态修改为 paused：

```
CHANGE PUMP TO NODE_STATE ='paused' FOR NODE_ID 'pump1';
```

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

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------------------|
| NodeID |     Address    | State  |    Max_Commit_Ts   |    Update_Time                  |
+--------|----------------|--------|--------------------|---------------------------------|
| pump1  | 127.0.0.1:8250 | Paused | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------------------|
2 rows in set (0.00 sec)
```

4.8.2.26.2 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.26.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE DRAINER STATUS

4.8.2.27 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。

4.8.2.27.1 语法图

```
CommitStmt ::=
    'COMMIT' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

4.8.2.27.2 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)
```

4.8.2.27.3 MySQL 兼容性

- 目前，默认情况下TiDB 使用元数据锁(MDL) 来防止 DDL 语句修改事务使用的表，这与 MySQL 中的元数据锁在行为上有一些差别，具体可以参考元数据锁。
- TiDB 3.0.8 及更新版本默认使用悲观事务模式。在乐观事务模型下，需要考虑到修改的行已被另一个事务修改，导致 COMMIT 语句可能执行失败的情况。
- 启用乐观事务模型后，UNIQUE 和 PRIMARY KEY 约束检查将延迟直至语句提交。当 COMMIT 语句失败时，这可能导致其他问题。可通过设置 tidb_constraint_check_in_place=ON 来改变该行为。
- TiDB 解析但忽略 ROLLBACK AND [NO] RELEASE 语法。在 MySQL 中，使用该语法可在提交事务后立即断开客户端会话。在TiDB 中，建议使用客户端程序的 mysql_close() 来实现该功能。
- TiDB 解析但忽略 ROLLBACK AND [NO] CHAIN 语法。在 MySQL 中，使用该语法可在提交当前事务时立即以相同的隔离级别开启新事务。在TiDB 中，推荐直接开启新事务。

4.8.2.27.4 另请参阅

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

4.8.2.28 CREATE [GLOBAL|SESSION] BINDING

CREATE [GLOBAL|SESSION] BINDING 语句用于在TiDB 中创建新的执行计划绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内创建执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

被绑定的 SQL 语句会被参数化后存储到系统表中。在处理 SQL 查询时，只要参数化后的 SQL 语句和系统表中某个被绑定的 SQL 语句一致，并且系统变量 tidb_use_plan_baselines 的值为 ON（其默认值为 ON），即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。更多信息，请参考创建绑定。

4.8.2.28.1 语法图

```
CreateBindingStmt ::=
    'CREATE' GlobalScope 'BINDING' ( 'FOR' BindableStmt 'USING' BindableStmt
|   'FROM' 'HISTORY' 'USING' 'PLAN' 'DIGEST' PlanDigest )

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

4.8.2.28.2 示例

你可以根据 SQL 或历史执行计划创建绑定。

下面的示例演示如何根据 SQL 创建绑定。

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0
```

```
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    -> LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    -> LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    -> LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+-----------+
| SLEEP(1) |
+-----------+
|         0 |
+-----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
     <-> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    <->
| id                           | estRows | actRows | task      | access object        |
    <-> execution info                                                          | operator info
    <->                      | memory         | disk |
+--
     <-> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    <->
| IndexLookUp_10               | 583.00  | 297     | root      |                      | time
    <-> :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    <->                                  | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)    | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    <-> , loops:4                                                               | range:[123,123],
    <-> keep order:false | N/A           | N/A  |
| └─TableRowIDScan_9(Probe)    | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    <-> ms, loops:4                                                             | keep order:false
    <->                  | N/A           | N/A  |
```

```
+--
    <⟷ ---------------------------+---------+---------+-----------+----------------------+------ ----------
    <⟷
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    <⟷ ----------------------+-----------+---------+-----------+--------------+----------------- ----------
    <⟷
| id                       | estRows    | actRows | task      | access object | execution info
    <⟷                                                                  | operator info       |
    <⟷ memory         | disk |
+--
    <⟷ ----------------------+-----------+---------+-----------+--------------+----------------- ----------
    <⟷
| TableReader_7            | 583.00     | 297     | root      |               | time:222.32506ms,
    <⟷ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6    |
    <⟷ 88.6640625 KB | N/A  |
| └─Selection_6            | 583.00     | 297     | cop[tikv] |               | time:224ms, loops
    <⟷ :298                                                             | eq(test.t1.b, 123) | N/A
    <⟷           | N/A  |
|   └─TableFullScan_5      | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    <⟷ :298                                                             | keep order:false    | N/A
    <⟷           | N/A  |
+--                                                                                              ----------
    <⟷ ----------------------+-----------+---------+-----------+--------------+-----------------
    <⟷
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
```

```
1 row in set (0.00 sec)

DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    <--> ----------------------------+---------+---------+-----------+----------------------+------------------
    <-->
| id                            | estRows | actRows | task      | access object        |
    <--> execution info                                                                  | operator info
    <-->                    | memory         | disk |
+--
    <--> ----------------------------+---------+---------+-----------+----------------------+------------------
    <-->
| IndexLookUp_10                | 583.00  | 297     | root      |                      | time
    <--> :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
    <-->                                    | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    <--> , loops:4                                                                       | range:[123,123], keep
    <--> order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    <--> , loops:4                                                                       | keep order:false
    <-->                  | N/A            | N/A  |
+--
    <--> ----------------------------+---------+---------+-----------+----------------------+------------------
    <-->
3 rows in set (0.01 sec)
```

下面的示例演示如何根据历史执行计划创建绑定。

```
mysql> CREATE TABLE t(id INT PRIMARY KEY , a INT, KEY(a));
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT plan_digest FROM INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE QUERY_SAMPLE_TEXT = '
    <--> SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1';
+                                                                  +
| plan_digest                                                      |
+                                                                  +
| 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb |
+                                                                  +
1 row in set (0.01 sec)
```

```
mysql> CREATE BINDING FROM HISTORY USING PLAN DIGEST '4
    <-> e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb';
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT @@LAST_PLAN_FROM_BINDING;
+                          +
| ------------------------ |
| @@LAST_PLAN_FROM_BINDING |
+                          +
|--------------------------1 |
+                          +
1 row in set (0.01 sec)
```

4.8.2.28.3 MySQL 兼容性

CREATE [GLOBAL|SESSION] BINDING 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.28.4 另请参阅

- DROP [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理(SPM)

4.8.2.29 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”。

4.8.2.29.1 语法图

```
CreateDatabaseStmt ::=
    'CREATE' 'DATABASE' IfNotExists DBName DatabaseOptionListOpt

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

DBName ::=
    Identifier

DatabaseOptionListOpt ::=
    DatabaseOptionList?
```

```
DatabaseOptionList ::=
    DatabaseOption ( ','? DatabaseOption )*

DatabaseOption ::=
    DefaultKwdOpt ( CharsetKw '='? CharsetName | 'COLLATE' '='? CollationName | 'ENCRYPTION' '='?
        <—> EncryptionOpt )
|   DefaultKwdOpt PlacementPolicyOption

PlacementPolicyOption ::=
    "PLACEMENT" "POLICY" EqOpt PolicyName
|   "PLACEMENT" "POLICY" (EqOpt | "SET") "DEFAULT"
```

4.8.2.29.2 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。

4.8.2.29.3 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)
```

4.8.2.29.4 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。

4.8.2.29.5 另请参阅

- USE
- ALTER DATABASE
- DROP DATABASE
- SHOW DATABASES

4.8.2.30 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。

4.8.2.30.1 语法图

```
CreateIndexStmt ::=
    'CREATE' IndexKeyTypeOpt 'INDEX' IfNotExists Identifier IndexTypeOpt 'ON' TableName '('
        <~> IndexPartSpecificationList ')' IndexOptionList IndexLockAndAlgorithmOpt

IndexKeyTypeOpt ::=
    ( 'UNIQUE' | 'SPATIAL' | 'FULLTEXT' )?

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

IndexTypeOpt ::=
    IndexType?

IndexPartSpecificationList ::=
    IndexPartSpecification ( ',' IndexPartSpecification )*

IndexOptionList ::=
    IndexOption*
```

```
IndexLockAndAlgorithmOpt ::=
    ( LockClause AlgorithmClause? | AlgorithmClause LockClause? )?

IndexType ::=
    ( 'USING' | 'TYPE' ) IndexTypeName

IndexPartSpecification ::=
    ( ColumnName OptFieldLen | '(' Expression ')' ) Order

IndexOption ::=
    'KEY_BLOCK_SIZE' '='? LengthNum
|   IndexType
|   'WITH' 'PARSER' Identifier
|   'COMMENT' stringLit
|   IndexInvisible

IndexTypeName ::=
    'BTREE'
|   'HASH'
|   'RTREE'

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?

OptFieldLen ::=
    FieldLen?

IndexNameList ::=
    ( Identifier | 'PRIMARY' )? ( ',' ( Identifier | 'PRIMARY' ) )*

KeyOrIndex ::=
    'Key' | 'Index'
```

4.8.2.30.2 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                      | estRows  | task      | access object | operator info                  |
+-------------------------+----------+-----------+---------------+--------------------------------+
| TableReader_7           | 10.00    | root      |               | data:Selection_6               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+---------+-----------+-------------------------+--------------------------------+
| id                      | estRows | task      | access object           | operator info                  |
+-------------------------+---------+-----------+-------------------------+--------------------------------+
| IndexReader_6           | 10.00   | root      |                         | index:IndexRangeScan_5         |
| └─IndexRangeScan_5      | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep order:false, stats:pseudo |
+-------------------------+---------+-----------+-------------------------+--------------------------------+
2 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
CREATE UNIQUE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.31 sec)
```

4.8.2.30.3 表达式索引

在一些场景中，查询的条件往往是基于某个表达式进行过滤。在这些场景中，一般的索引不能生效，执行查询只能遍历整个表，导致查询性能较差。表达式索引是一种特殊的索引，能将索引建立于表达式上。在创建了表达式索引后，基于表达式的查询便可以使用上索引，极大提升查询的性能。

假设要基于 `lower(col1)` 这个表达式建立索引，示例的 SQL 语句如下：

```
CREATE INDEX idx1 ON t1 ((lower(col1)));
```

或者等价的语句：

```
ALTER TABLE t1 ADD INDEX idx1((lower(col1)));
```

还可以在建表的同时指定表达式索引：

```
CREATE TABLE t1(col1 char(10), col2 char(10), index((lower(col1))));
```

> 注意：
>
> 表达式索引中的表达式需要用 `(` 和 `)` 包围起来，否则会报语法错误。

删除表达式索引与删除普通索引的方法一致：

```
DROP INDEX idx1 ON t1;
```

> 注意：
>
> 表达式索引涉及众多表达式。为了确保正确性，当前仅允许经充分测试的一部分函数用于创建表达式索引，即生产环境中仅允许表达式中包含这些函数。这些函数可以通过查询变量 `tidb_allow_function_for_expression_index` 得到。在后续版本中，这些函数会持续增加。目前允许的函数如下:
>
> ```
> json_array, json_array_append, json_array_insert, json_contains, json_contains_path
>     <-> , json_depth, json_extract, json_insert, json_keys, json_length,
>     <-> json_merge_patch, json_merge_preserve, json_object, json_pretty, json_quote
>     <-> , json_remove, json_replace, json_search, json_set, json_storage_size,
>     <-> json_type, json_unquote, json_valid, lower, md5, reverse, tidb_shard, upper
>     <-> , vitess_hash
> ```
>
> 对于以上列表之外的函数，由于未完成充分测试，当前仍为实验特性，不建议在生产环境中使用。其他的表达式例如运算符、`cast` 和 `case when` 也同样为实验特性，不建议在生产环境中使用。如果仍然希望使用，可以在TiDB 配置文件中进行以下设置：
>
> ```
> allow-expression-index = true
> ```

-
-
-
-

-
-
-

←→

当查询语句中的表达式与表达式索引中的表达式一致时，优化器可以为该查询选择使用表达式索引。依赖于统计信息，某些情况下优化器不一定选择表达式索引。这时可以通过hint 指定强制使用表达式索引。

在以下示例中，假设建立在 `lower(col1)` 表达式上的索引为 `idx`。

当读取的结果为相同的表达式时，可以使用表达式索引。例如：

```
SELECT lower(col1) FROM t;
```

当过滤的条件中有相同的表达式时，可以使用表达式索引。例如：

```
SELECT * FROM t WHERE lower(col1) = "a";
SELECT * FROM t WHERE lower(col1) > "a";
SELECT * FROM t WHERE lower(col1) BETWEEN "a" AND "b";
SELECT * FROM t WHERE lower(col1) in ("a", "b");
SELECT * FROM t WHERE lower(col1) > "a" AND lower(col1) < "b";
SELECT * FROM t WHERE lower(col1) > "b" OR lower(col1) < "a";
```

当查询按照相同的表达式进行排序时，可以使用表达式索引。例如：

```
SELECT * FROM t ORDER BY lower(col1);
```

当聚合函数或者 `GROUP BY` 中包含相同的表达式时，可以使用表达式索引。例如：

```
SELECT max(lower(col1)) FROM t;
SELECT min(col1) FROM t GROUP BY lower(col1);
```

要查看表达式索引对应的表达式，可执行 `show index` 或查看系统表 `information_schema.tidb_indexes` 以及 `information_schema.STATISTICS` 表，输出中 `Expression` 这一列显示对应的表达式。对于非表达式索引，该列的值为 `NULL`。

维护表达式索引的代价比一般的索引更高，因为在插入或者更新每一行时都需要计算出表达式的值。因为表达式的值已经存储在索引中，所以当优化器选择表达式索引时，表达式的值就不需要再计算。因此，当查询速度比插入速度和更新速度更重要时，可以考虑建立表达式索引。

表达式索引的语法和限制与 MySQL 相同，是通过将索引建立在隐藏的虚拟生成列(generated virtual column) 上来实现的。因此所支持的表达式继承了虚拟生成列的所有限制。

4.8.2.30.4 多值索引

多值索引是一种定义在数组列上的二级索引。在普通索引中，一条索引记录对应一条数据记录(1:1)。而在多值索引中，存在多条索引记录对应一条数据记录(N:1)。多值索引用于索引JSON 数组。例如，一个定义在 `zipcode` 字段上的多值索引会对每一个 `zipcode` 中的记录产生一条索引记录。

```
{
    "user":"Bob",
    "user_id":31,
    "zipcode":[94477,94536]
}
```

创建多值索引

创建多值索引与创建表达式索引的方法一致。在索引定义中使用 `CAST(... AS ... ARRAY)` 表达式来创建一个多值索引。

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
);
```

多值索引可以被定义为唯一索引：

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    UNIQUE INDEX zips( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
);
```

当被定义为唯一索引时，试图插入重复数据将会报错：

```
mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [1,2]}');
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [2,3]}');
ERROR 1062 (23000): Duplicate entry '2' for key 'customers.zips'
```

允许同一条记录存在重复的值，不同记录出现重复值时将报错：

```
-- 插入成功
mysql> INSERT INTO t1 VALUES('[1,1,2]');
mysql> INSERT INTO t1 VALUES('[3,3,3,4,4,4]');

-- 插入失败
mysql> INSERT INTO t1 VALUES('[1,2]');
mysql> INSERT INTO t1 VALUES('[2,3]');
```

多值索引也可以被定义为复合索引：

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    INDEX zips(name, (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
);
```

当被定义为复合索引时，多值部分可以出现在任意位置，但是只能出现一次。

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    INDEX zips(name, (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)), (CAST(custinfo->'$.zipcode'
        <-> AS UNSIGNED ARRAY)))
);
ERROR 1235 (42000): This version of TiDB doesn't yet support 'more than one multi-valued key part
    <-> per index'.
```

写入的数据必须与多值索引的定义类型完全匹配，否则数据写入失败：

```
-- zipcode 字段中的所有元素必须为 UNSIGNED 类型
mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [-1]}');
ERROR 3752 (HY000): Value is out of range for expression index 'zips' at row 1

mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": ["1"]}'); -- 与 MySQL 不兼容
ERROR 3903 (HY000): Invalid JSON value for CAST for expression index 'zips'

mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [1]}');
Query OK, 1 row affected (0.00 sec)
```

使用多值索引

请参考索引的选择。

特性与限制

- 如果是空JSON 数组，则不会有对应的索引记录。
- CAST(... AS ... ARRAY) 中的目标类型不能是 BINARY、 JSON、 YEAR、 FLOAT、 DECIMAL。其中源类型必须是JSON。
- 无法使用多值索引进行排序。
- 只允许在JSON 数组上建立多值索引。
- 多值索引不可以作为主键或外键。
- 多值索引使用额外的存储空间为：平均每行数组元素个数* 普通二级索引使用空间。
- 相比于普通索引，DML 会对多值索引产生更多的索引记录的修改，因此多值索引会带来比普通索引更大的性能影响。
- 由于多值索引是一种特殊的表达式索引，因此具有表达式索引的限制。
- 使用备份恢复工具(BR)、同步工具(TiCDC)、导入工具(TiDB Lightning) 无法将定义了多值索引的表备份、同步、导入到低于 v6.6.0 版本的TiDB。

4.8.2.30.5 不可见索引

不可见索引(Invisible Indexes) 不会被查询优化器使用：

```
CREATE TABLE t1 (c1 INT, c2 INT, UNIQUE(c2));
CREATE UNIQUE INDEX c1 ON t1 (c1) INVISIBLE;
```

具体可以参考ALTER INDEX。

4.8.2.30.6 相关系统变量

和 CREATE INDEX 语句相关的系统变量有 tidb_ddl_enable_fast_reorg、tidb_ddl_reorg_worker_cnt 、tidb_ddl_reorg_batch_size 、tidb_ddl_reorg_priority 和 tidb_enable_auto_increment_in_generated，具体可以参考系统变量。

4.8.2.30.7 MySQL 兼容性

- TiDB 支持解析 FULLTEXT 和 SPATIAL 语法，但尚不支持使用 FULLTEXT，HASH 和 SPATIAL 索引。
- 不支持降序索引（类似于 MySQL 5.7）。
- 无法向表中添加 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 表达式索引与视图存在兼容性问题。通过视图进行查询时，无法使用上表达式索引。
- 表达式索引与 Binding 存在兼容性问题。当表达式索引中的表达式存在常量时，对应查询所建的 Binding 会扩大范围。假设表达式索引中的表达式为 a+1，对应的查询条件为 a+1 > 2。则建立的 Binding 为 a+? > ?，这会导致像 a+2 > 2 这样的查询也会强制使用表达式索引，得到一个较差的执行计划。这同样影响 SQL Plan Management (SPM) 中的捕获和演进功能。
- 多值索引写入的数据必须与定义类型完全匹配，否则数据写入失败。详见创建多值索引。

4.8.2.30.8 另请参阅

- 索引的选择
- 错误索引的解决方案
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

4.8.2.31 CREATE PLACEMENT POLICY

`CREATE PLACEMENT POLICY` 用于创建命名的放置策略，随后可以将该策略分配给表、分区或数据库。

4.8.2.31.1 语法图

```
CreatePolicyStmt ::=
    "CREATE" "PLACEMENT" "POLICY" IfNotExists PolicyName PlacementOptionList

PolicyName ::=
    Identifier

PlacementOptionList ::=
    PlacementOption
|   PlacementOptionList PlacementOption
|   PlacementOptionList ',' PlacementOption

PlacementOption ::=
    CommonPlacementOption
|   SugarPlacementOption
|   AdvancedPlacementOption

CommonPlacementOption ::=
    "FOLLOWERS" EqOpt LengthNum

SugarPlacementOption ::=
    "PRIMARY_REGION" EqOpt stringLit
|   "REGIONS" EqOpt stringLit
|   "SCHEDULE" EqOpt stringLit

AdvancedPlacementOption ::=
    "LEARNERS" EqOpt LengthNum
|   "CONSTRAINTS" EqOpt stringLit
|   "LEADER_CONSTRAINTS" EqOpt stringLit
```

```
|   "FOLLOWER_CONSTRAINTS" EqOpt stringLit
|   "LEARNER_CONSTRAINTS" EqOpt stringLit
```

4.8.2.31.2 示例

> **注意：**
>
> 如要查看所在集群中可用的区域，见SHOW PLACEMENT LABELS。如果未看到任何可用的区域，此 TiKV 集群在部署时可能未正确设置标签(label)。

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW CREATE PLACEMENT POLICY p1;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

+---------+                                                
```

4.8.2.32 CREATE RESOURCE GROUP

CREATE RESOURCE GROUP 语句用于在当前所选数据库中创建资源组。

4.8.2.32.1 语法图

```
CreateResourceGroupStmt ::=
   "CREATE" "RESOURCE" "GROUP" IfNotExists ResourceGroupName ResourceGroupOptionList

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

ResourceGroupName ::=
   Identifier

ResourceGroupOptionList ::=
    DirectResourceGroupOption
|   ResourceGroupOptionList DirectResourceGroupOption
|   ResourceGroupOptionList ',' DirectResourceGroupOption

DirectResourceGroupOption ::=
    "RU_PER_SEC" EqOpt stringLit
|   "PRIORITY" EqOpt ResourceGroupPriorityOption
|   "BURSTABLE"

ResourceGroupPriorityOption ::=
    LOW
|   MEDIUM
|   HIGH
```

资源组的 ResourceGroupName 是全局唯一的，不允许重复。

TiDB 支持以下 DirectResourceGroupOption, 其中Request Unit (RU) 是TiDB 对 CPU、IO 等系统资源统一抽象的单位。

参数	含义	举例
RU_PER_SEC	每秒 RU 填充的速度	RU_PER_SEC = 500 表示此资源组每秒回填 500 个 RU。
PRIORITY	任务在 TiKV 上处理的绝对优先级	PRIORITY = HIGH 表示优先级高。若未指定，则默认为 ME
BURSTABLE	允许对应的资源组超出配额后使用空余的系统资源。	

> 注意：
>
> - CREATE RESOURCE GROUP 语句只能在全局变量tidb_enable_resource_control 设置为 ON 时才能执行。

- TiDB 集群在初始化时会自动创建 default 资源组，其 RU_PER_SEC 的默认值为 UNLIMITED (等同于 INT 类型最大值，即 2147483647)，且为 BURSTABLE 模式。所有未绑定资源组的请求都将自动绑定至此资源组。在新建配置其他资源组时，建议根据实际情况修改default 资源组的配置。

4.8.2.32.2 示例

创建 rg1 和 rg2 两个资源组。

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1
  RU_PER_SEC = 100
  PRIORITY = HIGH
  BURSTABLE;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg2
  RU_PER_SEC = 200;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1' or NAME = 'rg2';
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |        100 | HIGH     | YES       |
| rg2  |        200 | MEDIUM   | NO        |
+------+------------+----------+-----------+
2 rows in set (1.30 sec)
```

4.8.2.32.3 MySQL 兼容性

MySQL 也支持 CREATE RESOURCE GROUP，但是接受的参数和TiDB 不同，两者并不兼容。

4.8.2.32.4 另请参阅

- DROP RESOURCE GROUP
- ALTER RESOURCE GROUP
- ALTER USER RESOURCE GROUP
- RU

4.8.2.33 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。

4.8.2.33.1 语法图

```
CreateRoleStmt ::=
    'CREATE' 'ROLE' IfNotExists RoleSpec (',' RoleSpec)*

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

RoleSpec ::=
    Rolename
```

4.8.2.33.2 示例

以 `root` 用户连接TiDB：

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

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
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)
```

以 `jennifer` 用户连接TiDB：

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

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`analyticsteam` 角色相关联的权限：

```
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)
```

以 root 用户连接TiDB：

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

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接TiDB：

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

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
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)
```

4.8.2.33.3 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。

4.8.2.33.4 另请参阅

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

4.8.2.34 CREATE SEQUENCE

CREATE SEQUENCE 语句用于在TiDB 中创建序列对象。序列是一种与表、视图对象平级的数据库对象，用于生成自定义的序列化ID。

4.8.2.34.1 语法图

```
CreateSequenceStmt ::=
    'CREATE' 'SEQUENCE' IfNotExists TableName CreateSequenceOptionListOpt
        <-> CreateTableOptionListOpt

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

CreateSequenceOptionListOpt ::=
    SequenceOption*
```

```
SequenceOptionList ::=
    SequenceOption

SequenceOption ::=
    ( 'INCREMENT' ( '='? | 'BY' ) | 'START' ( '='? | 'WITH' ) | ( 'MINVALUE' | 'MAXVALUE' | '
        <↔ CACHE' ) '='? ) SignedNum
|   'NOMINVALUE'
|   'NO' ( 'MINVALUE' | 'MAXVALUE' | 'CACHE' | 'CYCLE' )
|   'NOMAXVALUE'
|   'NOCACHE'
|   'CYCLE'
|   'NOCYCLE'
```

4.8.2.34.2 语法说明

```
CREATE [TEMPORARY] SEQUENCE [IF NOT EXISTS] sequence_name
    [ INCREMENT [ BY | = ] increment ]
    [ MINVALUE [=] minvalue | NO MINVALUE | NOMINVALUE ]
    [ MAXVALUE [=] maxvalue | NO MAXVALUE | NOMAXVALUE ]
    [ START [ WITH | = ] start ]
    [ CACHE [=] cache | NOCACHE | NO CACHE]
    [ CYCLE | NOCYCLE | NO CYCLE]
    [table_options]
```

4.8.2.34.3 参数说明

参数	默认值	描述
TEMPORARY <↔	fYalse <↔	TiDB 暂时不支持 TEMPORARY <↔ 选项，仅在语法上做兼容。

参数	默认值	描述
INCREMEN1T <⟷		指定序列的步长。其正负值可以控制序列的增长方向。
MINVALUE1 或 <⟷	-9223372036854775807 <⟷	指定序列的最小值。当 INCREMENT <⟷ > 0 时，默认值为 1；当 INCREMENT <⟷ < 0 时，默认值为 -9223372036854775807 <⟷。

参数	默认值	描述
MAXVALUE	9223372036854775806 或 -1	指定序列的最大值。当 INCREMENT <⟷ > 0 时，默认值为 9223372036854775806 <⟷；当 INCREMENT <⟷ < 0 时，默认值为 -1。
START <⟷	MINVALUE <⟷ 或 MAXVALUE <⟷	指定序列的初始值。当 INCREMENT <⟷ > 0 时，默认值为 MINVALUE <⟷；当 INCREMENT <⟷ < 0 时，默认值为 MAXVALUE <⟷。

参数	默认值	描述
CACHE ⟷	1000	指定每个 TiDB 本地缓存序列的大小。

参数	默认值	描述
CYCLE ⟷	NO ⟷ CYCLE ⟷	指定序列用完之后是否要循环使用。在 CYCLE ⟷ 的情况下，当 INCREMENT ⟷ > 0 时，序列用完后的后续起始值为 MINVALUE ⟷；当 INCREMENT ⟷ < 0 时，序列用完后的后续起始值为 MAXVALUE ⟷。

4.8.2.34.4 SEQUENCE 函数

主要通过表达式函数来操纵序列的使用。

- NEXTVAL 或 NEXT VALUE FOR

本质上都是 nextval() 函数，获取序列对象的下一个有效值，其参数为序列的 identifier。

- LASTVAL

  lastval() 函数， 用于获取本会话上一个使用过的值。如果没有值， 则为 NULL， 其参数为序列的 identifier。

- SETVAL

  setval() 函数，用于设置序列的增长。其第一参数为序列的 identifier，第二个参数为 num。

> **注意：**
>
> 在TiDB 序列的实现中，SETVAL 函数并不能改变序列增长的初始步调或循环步调。在 SETVAL 之后只会返回符合步调规律的下一个有效的序列值。

4.8.2.34.5 示例

- 创建一个默认参数的序列对象。

```
CREATE SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.06 sec)
```

- 使用 nextval() 函数获取序列对象的下一个值。

```
SELECT nextval(seq);
```

```
+---------------------+
| nextval(seq) |
+---------------------+
|             1 |
+---------------------+
1 row in set (0.02 sec)
```

- 使用 lastval() 函数获取本会话上一次调用序列对象所产生的值。

```
SELECT lastval(seq);
```

```
+---------------------+
| lastval(seq) |
+---------------------+
|             1 |
+---------------------+
1 row in set (0.02 sec)
```

- 使用 `setval()` 函数设置序列对象当前值的位置。

```
SELECT setval(seq, 10);
```

```
+------------------+
| setval(seq, 10) |
+------------------+
|               10 |
+------------------+
1 row in set (0.01 sec)
```

- 也可使用 `next value for` 语法获取序列的下一个值。

```
SELECT next value for seq;
```

```
+---------------------+
| next value for seq |
+---------------------+
|                  11 |
+---------------------+
1 row in set (0.00 sec)
```

- 创建一个默认自定义参数的序列对象。

```
CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;
```

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

- 当本会话还未使用过序列对象时，`lastval()` 函数返回 NULL 值。

```
SELECT lastval(seq2);
```

```
+-----------------------+
| lastval(seq2) |
+-----------------------+
|           NULL |
+-----------------------+
1 row in set (0.01 sec)
```

- 序列对象 `nextval()` 的第一个有效值为 `start` 值。

```
SELECT nextval(seq2);
```

```
+-----------------------+
| nextval(seq2) |
+-----------------------+
|              3 |
```

```
+------------------------+
1 row in set (0.00 sec)
```

- 使用 `setval()` 虽然可以改变序列对象当前值的位置，但是无法改变下一个值的等差规律。

```
SELECT setval(seq2, 6);
```

```
+------------------+
| setval(seq2, 6) |
+------------------+
|                6 |
+------------------+
1 row in set (0.00 sec)
```

- 使用 `nextval()` 下一个值获取时，会遵循序列定义的等差规律。

```
SELECT next value for seq2;
```

```
+-----------------------------------+
| next value for seq2 |
+-----------------------------------+
|                    7 |
+-----------------------------------+
1 row in set (0.00 sec)
```

- 可以将序列的下一个值作为列的默认值来使用。

```
CREATE TABLE t(a int default next value for seq2);
```

```
Query OK, 0 rows affected (0.02 sec)
```

- 下列示例中，因为没有指定值，会直接获取 `seq2` 的默认值来使用。

```
INSERT into t values();
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * from t;
```

```
+----------+
| a     |
+----------+
|     9 |
+----------+
1 row in set (0.00 sec)
```

- 下列示例中，因为没有指定值，会直接获取 seq2 的默认值来使用。由于 seq2 的下一个值超过了上述示例 (CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;) 的定义范围，所以会显示报错。

```
INSERT into t values();
```

```
ERROR 4135 (HY000): Sequence 'test.seq2' has run out
```

4.8.2.34.6 MySQL 兼容性

该语句是TiDB 的扩展，序列的实现借鉴自 MariaDB。

除了 SETVAL 函数外，其他函数的 “步调(progressions)” 与 MariaDB 一致。这里的步调是指，序列中的数在定义之后会产生一定的等差关系。SETVAL 虽然可以将序列的当前值进行移动设置，但是后续出现的值仍会遵循原有的等差关系。

示例如下：

```
1, 3, 5, ...            // 序列遵循起始为 1、步长为 2 的等差关系。
select setval(seq, 6)   // 设置序列的当前值为 6。
7, 9, 11, ...           // 后续产生值仍会遵循这个等差关系。
```

在 CYCLE 模式下，序列的起始值第一轮为 start，后续轮次将会是 MinValue (increment > 0) 或 MaxValue (increment < 0)。

4.8.2.34.7 另请参阅

- DROP SEQUENCE
- SHOW CREATE SEQUENCE

4.8.2.35 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。

4.8.2.35.1 语法图

```
CreateTableLikeStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName LikeTableWithOrWithoutParen OnCommitOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

LikeTableWithOrWithoutParen ::=
    'LIKE' TableName
|   '(' 'LIKE' TableName ')'

OnCommitOpt ::=
    ('ON' 'COMMIT' 'DELETE' 'ROWS')?
```

4.8.2.35.2 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)
```

4.8.2.35.3 Region 的预切分

如果被复制的表定义了 PRE_SPLIT_REGIONS 属性，则通过 CREATE TABLE LIKE 语句复制的表，会继承该属性并在建表时预切分Region。关于 PRE_SPLIT_REGIONS 属性的说明，参见CREATE TABLE 语句。

4.8.2.35.4 MySQL 兼容性

CREATE TABLE LIKE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在GitHub 上提交issue。

4.8.2.35.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE

4.8.2.36 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。

4.8.2.36.1 语法图

```
CreateTableStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName ( TableElementListOpt
        <⟶ CreateTableOptionListOpt PartitionOpt DuplicateOpt AsOpt CreateTableSelectOpt |
        <⟶ LikeTableWithOrWithoutParen ) OnCommitOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

TableElementListOpt ::=
    ( '(' TableElementList ')' )?

TableElementList ::=
    TableElement ( ',' TableElement )*

TableElement ::=
    ColumnDef
|   Constraint

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

ColumnOptionListOpt ::=
    ColumnOption*

ColumnOptionList ::=
    ColumnOption*

ColumnOption ::=
    'NOT'? 'NULL'
|   'AUTO_INCREMENT'
|   PrimaryOpt 'KEY'
|   'UNIQUE' 'KEY'?
|   'DEFAULT' DefaultValueExpr
```

```
|   'SERIAL' 'DEFAULT' 'VALUE'
|   'ON' 'UPDATE' NowSymOptionFraction
|   'COMMENT' stringLit
|   ConstraintKeywordOpt 'CHECK' '(' Expression ')' EnforcedOrNotOrNotNullOpt
|   GeneratedAlways 'AS' '(' Expression ')' VirtualOrStored
|   ReferDef
|   'COLLATE' CollationName
|   'COLUMN_FORMAT' ColumnFormat
|   'STORAGE' StorageMedia
|   'AUTO_RANDOM' OptFieldLen

Constraint ::=
    IndexDef
|   ForeignKeyDef

IndexDef ::=
    ( 'INDEX' | 'KEY' ) IndexName? '(' KeyPartList ')' IndexOption?

KeyPartList ::=
    KeyPart ( ',' KeyPart )*

KeyPart ::=
    ColumnName ( '(' Length ')')? ( 'ASC' | 'DESC' )?
|   '(' Expression ')' ( 'ASC' | 'DESC' )?

IndexOption ::=
    'COMMENT' String
|   ( 'VISIBLE' | 'INVISIBLE' )

ForeignKeyDef
         ::= ( 'CONSTRAINT' Identifier )? 'FOREIGN' 'KEY'
             Identifier? '(' ColumnName ( ',' ColumnName )* ')'
             'REFERENCES' TableName '(' ColumnName ( ',' ColumnName )* ')'
             ( 'ON' 'DELETE' ReferenceOption )?
             ( 'ON' 'UPDATE' ReferenceOption )?

ReferenceOption
         ::= 'RESTRICT'
           | 'CASCADE'
           | 'SET' 'NULL'
           | 'SET' 'DEFAULT'
           | 'NO' 'ACTION'

CreateTableOptionListOpt ::=
    TableOptionList?
```

```
PartitionOpt ::=
    ( 'PARTITION' 'BY' PartitionMethod PartitionNumOpt SubPartitionOpt PartitionDefinitionListOpt
        <⟷ )?

DuplicateOpt ::=
    ( 'IGNORE' | 'REPLACE' )?

TableOptionList ::=
    TableOption ( ','? TableOption )*

TableOption ::=
    PartDefOption
|   DefaultKwdOpt ( CharsetKw EqOpt CharsetName | 'COLLATE' EqOpt CollationName )
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'AVG_ROW_LENGTH' | 'CHECKSUM' | '
    <⟷ TABLE_CHECKSUM' | 'KEY_BLOCK_SIZE' | 'DELAY_KEY_WRITE' | 'SHARD_ROW_ID_BITS' | '
    <⟷ PRE_SPLIT_REGIONS' ) EqOpt LengthNum
|   ( 'CONNECTION' | 'PASSWORD' | 'COMPRESSION' ) EqOpt stringLit
|   RowFormat
|   ( 'STATS_PERSISTENT' | 'PACK_KEYS' ) EqOpt StatsPersistentVal
|   ( 'STATS_AUTO_RECALC' | 'STATS_SAMPLE_PAGES' ) EqOpt ( LengthNum | 'DEFAULT' )
|   'STORAGE' ( 'MEMORY' | 'DISK' )
|   'SECONDARY_ENGINE' EqOpt ( 'NULL' | StringName )
|   'UNION' EqOpt '(' TableNameListOpt ')'
|   'ENCRYPTION' EqOpt EncryptionOpt
|    'TTL' EqOpt TimeColumnName '+' 'INTERVAL' Expression TimeUnit (TTLEnable EqOpt ( 'ON' | 'OFF
    <⟷ ' ))? (TTLJobInterval EqOpt stringLit)?
|   PlacementPolicyOption

OnCommitOpt ::=
    ('ON' 'COMMIT' 'DELETE' 'ROWS')?

PlacementPolicyOption ::=
    "PLACEMENT" "POLICY" EqOpt PolicyName
|   "PLACEMENT" "POLICY" (EqOpt | "SET") "DEFAULT"
```

TiDB 支持以下 table_option。TiDB 会解析并忽略其他 table_option 参数，例如 AVG_ROW_LENGTH、 CHECKSUM、COMPRESSION、 CONNECTION、 DELAY_KEY_WRITE、 ENGINE、 KEY_BLOCK_SIZE、 MAX_ROWS、 MIN_ROWS、 ROW_FORMAT 和 STATS_PERSISTENT。

参数	含义	举例
AUTO_INCREMENT	自增字段初始值	AUTO_INCREMENT = 5
SHARD_ROW_ID_BITS	用来设置隐式 _tidb_rowid 的分片数量的bit 位数	SHARD_ROW_ID_BITS = 4

参数	含义	举例
PRE_SPLIT_REGIONS	用来在建表时预先均匀切分 2^(PRE_SPLIT_REGIONS) 个Region	PRE_SPLIT_REGIONS = 4
AUTO_ID_CACHE	用来指定Auto ID 在 TiDB 实例中 Cache 的大小，默认情况下TiDB 会根据Auto ID 分配速度自动调整	AUTO_ID_CACHE = 200
AUTO_RANDOM_BASE	用来指定AutoRandom 自增部分的初始值，该参数可以被认为属于内部接口的一部分，对于用户而言请忽略	AUTO_RANDOM_BASE = 0
CHARACTER SET	指定该表所使用的字符集	CHARACTER SET = ‘utf8mb4’
COLLATE	指定该表所使用的字符集排序规则	COLLATE = ‘utf8mb4_bin’
COMMENT	注释信息	COMMENT = ‘comment info’

> **注意：**
>
> 在TiDB 配置文件中，split-table 默认开启。当该配置项开启时，建表操作会为每个表建立单独的Region，详情参见TiDB 配置文件描述。

4.8.2.36.2 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)
```

4.8.2.36.3 MySQL 兼容性

- 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 属性不支持 WITH PARSER 选项。
- TiDB 在单个表中默认支持 1017 列，最大可支持 4096 列。InnoDB 中相应的数量限制为 1017 列，MySQL 中的硬限制为 4096 列。详情参阅TiDB 使用限制。
- 当前仅支持Range、 Hash 和Range Columns（单列）类型的分区表，详情参阅分区表。
- TiDB 会解析并忽略 CHECK 约束，与 MySQL 5.7 相兼容。详情参阅CHECK 约束。

4.8.2.36.4 另请参阅

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

4.8.2.37 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。

4.8.2.37.1 语法图

```
CreateUserStmt ::=
    'CREATE' 'USER' IfNotExists UserSpecList RequireClauseOpt ConnectionOptions PasswordOption
        <⟷ LockOption AttributeOption ResourceGroupNameOption

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

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

UserSpec ::=
    Username AuthOption

AuthOption ::=
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        <⟷ AuthString | 'AS' HashString )? ) )?

StringName ::=
    stringLit
```

```
|   Identifier

PasswordOption ::= ( 'PASSWORD' 'EXPIRE' ( 'DEFAULT' | 'NEVER' | 'INTERVAL' N 'DAY' )? | '
    <⟶ PASSWORD' 'HISTORY' ( 'DEFAULT' | N ) | 'PASSWORD' 'REUSE' 'INTERVAL' ( 'DEFAULT' | N '
    <⟶ DAY' ) | 'FAILED_LOGIN_ATTEMPTS' N | 'PASSWORD_LOCK_TIME' ( N | 'UNBOUNDED' ) )*

LockOption ::= ( 'ACCOUNT' 'LOCK' | 'ACCOUNT' 'UNLOCK' )?

AttributeOption ::= ( 'COMMENT' CommentString | 'ATTRIBUTE' AttributeString )?

ResourceGroupNameOption::= ( 'RESOURCE' 'GROUP' Identifier)?
```

4.8.2.37.2 示例

创建一个密码为 newuserpassword 的用户。

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

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

创建一个只能在 192.168.1.1 登录的用户，密码为 newuserpassword。

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

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

创建一个要求在登录时使用 TLS 连接的用户。

```
CREATE USER 'newuser3'@'%' IDENTIFIED BY 'newuserpassword' REQUIRE SSL;
```

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

创建一个要求在登录时提供指定客户端证书的用户。

```
CREATE USER 'newuser4'@'%' IDENTIFIED BY 'newuserpassword' REQUIRE ISSUER '/C=US/ST=California/L=
    <⟶ San Francisco/O=PingCAP';
```

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

创建一个初始状态下被锁住的用户。

```
CREATE USER 'newuser5'@'%' ACCOUNT LOCK;
```

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

创建一个带注释的用户。

```
CREATE USER 'newuser6'@'%' COMMENT 'This user is created only for test';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| newuser6  | %    | {"comment": "This user is created only for test"} |
+-----------+------+-----------------------------------------------------+
1 rows in set (0.00 sec)
```

创建一个具有邮箱(`email`)属性的用户。

```
CREATE USER 'newuser7'@'%' ATTRIBUTE '{"email": "user@pingcap.com"}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| newuser7  | %    | {"email": "user@pingcap.com"} |
+-----------+------+-----------------------------------------------------+
1 rows in set (0.00 sec)
```

创建一个禁止重复使用最近 5 次密码的用户。

```
CREATE USER 'newuser8'@'%' PASSWORD HISTORY 5;
```

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

创建一个密码已经手动过期的用户。

```
CREATE USER 'newuser9'@'%' PASSWORD EXPIRE;
```

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

创建一个使用资源组 `rg1` 的用户：

```
CREATE USER 'newuser7'@'%' RESOURCE GROUP rg1;
SELECT USER, HOST, USER_ATTRIBUTES FROM MYSQL.USER WHERE USER='newuser7';
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | USER_ATTRIBUTES                                     |
+-----------+------+-----------------------------------------------------+
| newuser7  | %    | {"resource_group": "rg1"} |
+-----------+------+-----------------------------------------------------+
1 rows in set (0.00 sec)
```

4.8.2.37.3 MySQL 兼容性

- TiDB 不支持 `WITH MAX_QUERIES_PER_HOUR`、`WITH MAX_UPDATES_PER_HOUR`、`WITH MAX_USER_CONNECTIONS` 等 `CREATE` 选项。
- TiDB 不支持 `DEFAULT ROLE` 选项。
- TiDB 不支持 `PASSWORD EXPIRE`、`PASSWORD HISTORY` 等有关密码限制的 `CREATE` 选项。
- 对于TiDB 尚不支持的 `CREATE` 选项。这些选项可被解析，但会被忽略。

4.8.2.37.4 另请参阅

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

4.8.2.38 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。

4.8.2.38.1 语法图

```
CreateViewStmt ::=
    'CREATE' OrReplace ViewAlgorithm ViewDefiner ViewSQLSecurity 'VIEW' ViewName ViewFieldList '
        <⟷ AS' CreateViewSelectOpt ViewCheckOption

OrReplace ::=
    ( 'OR' 'REPLACE' )?

ViewAlgorithm ::=
    ( 'ALGORITHM' '=' ( 'UNDEFINED' | 'MERGE' | 'TEMPTABLE' ) )?

ViewDefiner ::=
    ( 'DEFINER' '=' Username )?

ViewSQLSecurity ::=
    ( 'SQL' 'SECURITY' ( 'DEFINER' | 'INVOKER' ) )?

ViewName ::= TableName

ViewFieldList ::=
    ( '(' Identifier ( ',' Identifier )* ')' )?

ViewCheckOption ::=
    ( 'WITH' ( 'CASCADED' | 'LOCAL' ) 'CHECK' 'OPTION' )?
```

4.8.2.38.2 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.
```

4.8.2.38.3 MySQL 兼容性

- 目前TiDB 中的任何视图都不可被插入，也不可被更新（即不支持INSERT VIEW，也不支持 UPDATE VIEW）。WITH CHECK OPTION 只做了语法兼容但不生效。
- 目前TiDB 中的视图不支持 ALTER VIEW，但可以使用 CREATE OR REPLACE 替代。
- 目前 ALGORITHM 字段在TiDB 中只做了语法兼容但不生效，TiDB 目前只支持 MERGE 算法。

4.8.2.38.4 另请参阅

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

4.8.2.39 DEALLOCATE

DEALLOCATE 语句用于为服务器端预处理语句提供 SQL 接口。

4.8.2.39.1 语法图

```
DeallocateStmt ::=
    DeallocateSym 'PREPARE' Identifier

DeallocateSym ::=
    'DEALLOCATE'
|   'DROP'

Identifier ::=
    identifier
```

```
|    UnReservedKeyword
|    NotKeywordToken
|    TiDBKeyword
```

4.8.2.39.2 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)
```

4.8.2.39.3 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。

4.8.2.39.4 另请参阅

- PREPARE
- EXECUTE

4.8.2.40 DELETE

DELETE 语句用于从指定的表中删除行。

4.8.2.40.1 语法图

```
DeleteFromStmt ::=
    'DELETE' TableOptimizerHints PriorityOpt QuickOptional IgnoreOptional ( 'FROM' ( TableName
        <⟷ TableAsNameOpt IndexHintListOpt WhereClauseOptional OrderByOptional LimitClause |
        <⟷ TableAliasRefList 'USING' TableRefs WhereClauseOptional ) | TableAliasRefList 'FROM'
        <⟷ TableRefs WhereClauseOptional )
```

4.8.2.40.2 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)
```

4.8.2.40.3 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。

4.8.2.40.4 另请参阅

- INSERT
- SELECT
- UPDATE
- REPLACE

4.8.2.41 DESC

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

4.8.2.42 DESCRIBE

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

4.8.2.43 DO

DO 语句用于执行表达式，但不返回任何结果。大部分情况下，DO 相当于不返回结果的 SELECT expr, ...,。

> **注意：**
>
> DO 只能执行表达式，所以不是所有能够用 SELECT 的地方都能用 DO 替换。例如 DO id FROM t1 就是不是合法的 SQL 语句，因为它引用了一张表。

DO 在 MySQL 中的一个主要应用场景是存储过程或者触发器。因为 TiDB 当前不支持存储过程和触发器，所以 DO 的实际使用场景较少。

4.8.2.43.1 语法图

```
DoStmt   ::= 'DO' ExpressionList

ExpressionList ::=
    Expression ( ',' Expression )*

Expression ::=
```

```
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        <-> Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    <-> PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?
```

4.8.2.43.2 示例

这条 SELECT 语句会暂停执行，但同时也会返回一个结果集。

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

```
+-----------------+
| SLEEP(5) |
+-----------------+
|         0 |
+-----------------+
1 row in set (5.00 sec)
```

如果使用 DO 的话，语句同样会暂停，但不会返回结果集。

```
DO SLEEP(5);
```

```
Query OK, 0 rows affected (5.00 sec)
```

```
DO SLEEP(1), SLEEP(1.5);
```

```
Query OK, 0 rows affected (2.50 sec)
```

4.8.2.43.3 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。

4.8.2.43.4 另请参阅

- SELECT

4.8.2.44 DROP [GLOBAL|SESSION] BINDING

DROP BINDING 语句用于删除指定的 SQL 绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内删除执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

4.8.2.44.1 语法图

```
DropBindingStmt ::=
    'DROP' GlobalScope 'BINDING' 'FOR' ( BindableStmt ( 'USING' BindableStmt )?
|   'SQL' 'DIGEST' SqlDigest)

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

4.8.2.44.2 示例

你可以根据 SQL 语句或 `sql_digest` 删除绑定。

下面的示例演示如何根据 SQL 语句删除绑定。

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
```

```
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+-----------+
| SLEEP(1) |
+-----------+
|         0 |
+-----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
     <-> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    <->
| id                           | estRows | actRows | task      | access object        |
    <-> execution info                                                                | operator info
    <->                      | memory         | disk |
+--
     <-> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    <->
| IndexLookUp_10               | 583.00  | 297     | root      |                      | time
    <-> :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    <->                                  | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)    | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    <-> , loops:4                                                                | range:[123,123],
    <-> keep order:false | N/A           | N/A  |
| └─TableRowIDScan_9(Probe)    | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    <-> ms, loops:4                                                              | keep order:false
    <->                  | N/A           | N/A  |
+--
     <-> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
```

```
    <->
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    <-> ----------------------+----------+---------+----------+--------------+----------------- ----------
    <->
| id                        | estRows    | actRows | task      | access object | execution info
    <->                                                                        | operator info       |
    <-> memory         | disk |
+--
    <-> ----------------------+----------+---------+----------+--------------+----------------- ----------
    <->
| TableReader_7             | 583.00     | 297     | root      |               | time:222.32506ms,
    <-> loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6     |
    <-> 88.6640625 KB | N/A  |
| └─Selection_6             | 583.00     | 297     | cop[tikv] |               | time:224ms, loops
    <-> :298                                                                   | eq(test.t1.b, 123) | N/A
    <->           | N/A  |
|   └─TableFullScan_5       | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    <-> :298                                                                   | keep order:false    | N/A
    <->           | N/A  |
+--                                                                                                  ----------
    <-> ----------------------+----------+---------+----------+--------------+-----------------
    <->
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

```
DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    <--> ----------------------------+---------+---------+-----------+----------------------+------------------
    <-->
| id                            | estRows | actRows | task      | access object        |
    <--> execution info                                                          | operator info
    <-->                     | memory          | disk |
+--
    <--> ----------------------------+---------+---------+-----------+----------------------+------------------
    <-->
| IndexLookUp_10                | 583.00  | 297     | root      |                      | time
    <--> :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
    <-->                                  | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    <--> , loops:4                                                               | range:[123,123], keep
    <--> order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    <--> , loops:4                                                               | keep order:false
    <-->                   | N/A            | N/A  |
+--
    <--> ----------------------------+---------+---------+-----------+----------------------+------------------
    <-->
3 rows in set (0.01 sec)

SHOW SESSION BINDINGS\G
Empty set (0.00 sec)
```

下面的示例演示如何根据 sql_digest 删除绑定。

```
mysql> CREATE TABLE t(id INT PRIMARY KEY , a INT, KEY(a));
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT plan_digest FROM INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE QUERY_SAMPLE_TEXT = '
    <--> SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1';
+                                                                  +
| plan_digest                                                      |
+                                                                  +
| 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb |
+                                                                  +
1 row in set (0.01 sec)
```

```
mysql> CREATE BINDING FROM HISTORY USING PLAN DIGEST '4
    -> e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb';
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT @@LAST_PLAN_FROM_BINDING;
+---------------------------+
| @@LAST_PLAN_FROM_BINDING |
+---------------------------+
|                         1 |
+---------------------------+
1 row in set (0.01 sec)

mysql> SHOW BINDINGS\G;
*************************** 1. row ***************************
Original_sql: select * from `test` . `t` where `a` = ?
    Bind_sql: SELECT /*+ use_index(@`sel_1` `test`.`t` ) ignore_index(`t` `a`)*/ * FROM `test`.`t
        -> ` WHERE `a` = 1
  Default_db: test
      Status: enabled
 Create_time: 2022-12-14 15:26:22.277
 Update_time: 2022-12-14 15:26:22.277
     Charset: utf8mb4
   Collation: utf8mb4_general_ci
      Source: history
  Sql_digest: 6909a1bbce5f64ade0a532d7058dd77b6ad5d5068aee22a531304280de48349f
 Plan_digest: 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb
1 row in set (0.02 sec)

ERROR:
No query specified

mysql> DROP BINDING FOR SQL DIGEST '6909
    -> a1bbce5f64ade0a532d7058dd77b6ad5d5068aee22a531304280de48349f';
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW BINDINGS\G;
Empty set (0.01 sec)

ERROR:
No query specified
```

4.8.2.44.3 MySQL 兼容性

`DROP [GLOBAL|SESSION] BINDING` 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.44.4 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理(SPM)

4.8.2.45 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。

4.8.2.45.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName DropColumnSpec ( ',' DropColumnSpec )*

DropColumnSpec
         ::= 'DROP' 'COLUMN'? 'IF EXISTS'? ColumnName ( 'RESTRICT' | 'CASCADE' )?

ColumnName
         ::= Identifier ( '.' Identifier ( '.' Identifier )? )?
```

4.8.2.45.2 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 s c)
```

```
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 s c)
```

```
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)
```

4.8.2.45.3 MySQL 兼容性

- 目前不支持删除主键列或组合索引相关列。

4.8.2.45.4 另请参阅

- ADD COLUMN
- SHOW CREATE TABLE
- CREATE TABLE

4.8.2.46 DROP DATABASE

DROP DATABASE 语句用于永久删除指定的数据库，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。

4.8.2.46.1 语法图

```
DropDatabaseStmt ::=
    'DROP' 'DATABASE' IfExists DBName

IfExists ::= ( 'IF' 'EXISTS' )?
```

4.8.2.46.2 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)
```

4.8.2.46.3 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。

4.8.2.46.4 另请参阅

- CREATE DATABASE
- ALTER DATABASE

4.8.2.47 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。

4.8.2.47.1 语法图

```
DropIndexStmt ::=
    "DROP" "INDEX" IfExists Identifier "ON" TableName IndexLockAndAlgorithmOpt

IfExists ::=
    ( 'IF' 'EXISTS' )?

IndexLockAndAlgorithmOpt ::=
    ( LockClause AlgorithmClause? | AlgorithmClause LockClause? )?
```

4.8.2.47.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

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

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------------------------------------+
    <↩>
| id                      | estRows  | task      | access object | operator info
    <↩>                 |
+-------------------------+----------+-----------+---------------+--------------------------------------------------------------+
    <↩>
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    <↩>               |
```

```
| └─Selection_6            | 10.00     | cop[tikv] |               | eq(test.t1.c1, 3)
    <->              |
|   └─TableFullScan_5      | 10000.00 | cop[tikv] | table:t1       | keep order:false, stats:
    <-> pseudo |
+--------------------------+----------+-----------+---------------+-------------------------------------------------------------+
    <->
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+                          +          +            +                          +-------------------------------------------------------------
    <->
| id                       | estRows | task       | access object           | operator info
    <->                                |
+--------------------------+----------+------------+--------------------------+                                                             ----------
    <->
| IndexReader_6            | 0.01     | root       |                          | index:IndexRangeScan_5
    <->                        |
| └─IndexRangeScan_5       | 0.01     | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    <-> order:false, stats:pseudo |
+                          +          +            +                          +-------------------------------------------------------------
    <->
2 rows in set (0.00 sec)
```

```
DROP INDEX c1 ON t1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

4.8.2.47.3 MySQL 兼容性

- 不支持删除 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

4.8.2.47.4 另请参阅

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

4.8.2.48 DROP PLACEMENT POLICY

`DROP PLACEMENT POLICY` 用于删除已创建的放置策略。

4.8.2.48.1 语法图

```
DropPolicyStmt ::=
    "DROP" "PLACEMENT" "POLICY" IfExists PolicyName

PolicyName ::=
    Identifier
```

4.8.2.48.2 示例

删除放置规则时，确保该策略未被任何表或分区引用，否则会删除失败。

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=4;
CREATE TABLE t1 (a INT PRIMARY KEY) PLACEMENT POLICY=p1;
DROP PLACEMENT POLICY p1;  -- 该语句执行失败，因为放置规则 p1 被引用。

-- 查看引用放置规则的表和分区。
SELECT table_schema, table_name FROM information_schema.tables WHERE tidb_placement_policy_name='
    <-> p1';
SELECT table_schema, table_name FROM information_schema.partitions WHERE
    <-> tidb_placement_policy_name='p1';

ALTER TABLE t1 PLACEMENT POLICY=default;  -- 移除表 t1 上的默认放置规则。
DROP PLACEMENT POLICY p1;  -- 执行成功。
```

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

Query OK, 0 rows affected (0.11 sec)

ERROR 8241 (HY000): Placement policy 'p1' is still in use

+--------------+------------+
| table_schema | table_name |
+--------------+------------+
| test         | t1         |
+--------------+------------+
1 row in set (0.00 sec)

Empty set (0.01 sec)

Query OK, 0 rows affected (0.08 sec)
```

```
Query OK, 0 rows affected (0.21 sec)
```

4.8.2.48.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.48.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- ALTER PLACEMENT POLICY

4.8.2.49 DROP RESOURCE GROUP

`DROP RESOURCE GROUP` 语句用于在当前所选数据库中删除资源组。

4.8.2.49.1 语法图

```
DropResourceGroupStmt ::=
    "DROP" "RESOURCE" "GROUP" IfExists ResourceGroupName

IfExists ::=
    ('IF' 'EXISTS')?

ResourceGroupName ::=
    Identifier
```

> **注意：**
>
> - `DROP RESOURCE GROUP` 语句只能在全局变量`tidb_enable_resource_control` 设置为 `ON` 时才能执行。
> - `default` 资源组为系统保留的资源组，不支持删除。

4.8.2.49.2 示例

删除名为 `rg1` 的资源组：

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1 RU_PER_SEC = 500 BURSTABLE;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
+------+------------+----------+--------------------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+--------------------+
| rg1  |        500 | MEDIUM   | YES       |
+------+------------+----------+--------------------+
1 row in set (0.01 sec)
```

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 1 rows affected (0.09 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

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

4.8.2.49.3 MySQL 兼容性

MySQL 也支持 DROP RESOURCE GROUP，但 TiDB 不支持 FORCE 参数。

4.8.2.49.4 另请参阅

- ALTER RESOURCE GROUP
- CREATE RESOURCE GROUP
- RU

4.8.2.50 DROP ROLE

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

4.8.2.50.1 语法图

```
DropRoleStmt ::=
    'DROP' 'ROLE' ( 'IF' 'EXISTS' )? RolenameList

RolenameList ::=
    Rolename ( ',' Rolename )*
```

4.8.2.50.2 示例

以 root 用户连接TiDB：

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

创建新角色 analyticsteam 和新用户 jennifer：

```
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)
```

以 jennifer 用户连接TiDB：

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

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：

```
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)
```

以 root 用户连接TiDB：

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

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接TiDB：

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

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
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)
```

以 root 用户连接TiDB：

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

删除角色 analyticsteam：

```
DROP ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
```

jennifer 用户不再具有与 analyticsteam 关联的默认角色，或不能再将 analyticsteam 设为启用角色：

以 jennifer 用户连接TiDB：

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

查看 jennifer 用户的权限：

```
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@%
```

4.8.2.50.3 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。

4.8.2.50.4 另请参阅

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

4.8.2.51 DROP SEQUENCE

DROP SEQUENCE 语句用于删除序列对象。

4.8.2.51.1 语法图

```
DropSequenceStmt ::=
    'DROP' 'SEQUENCE' IfExists TableNameList

IfExists ::= ( 'IF' 'EXISTS' )?
```

```
TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

4.8.2.51.2 示例

```
DROP SEQUENCE seq;
```

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

```
DROP SEQUENCE seq, seq2;
```

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

4.8.2.51.3 MySQL 兼容性

该语句是TiDB 的扩展，序列的实现借鉴自 MariaDB。

4.8.2.51.4 另请参阅

- CREATE SEQUENCE
- SHOW CREATE SEQUENCE

4.8.2.52 DROP STATS

DROP STATS 语句用于从当前所选定的数据库中删除选定表的统计信息。

4.8.2.52.1 语法图

```
DropStatsStmt ::=
    'DROP' 'STATS' TableNameList

TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

4.8.2.52.2 示例

```
CREATE TABLE t(a INT);
```

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

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

```
+---------+------------+----------------+---------------------+--------------+-------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-------------+
| test    | t          |                | 2020-05-25 20:34:33 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-------------+
1 row in set (0.00 sec)
```

```
DROP STATS t;
```

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

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

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

4.8.2.52.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.52.4 另请参阅

- 统计信息简介

4.8.2.53 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。

4.8.2.53.1 语法图

```
DropTableStmt ::=
    'DROP' OptTemporary TableOrTables IfExists TableNameList RestrictOrCascadeOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

TableOrTables ::=
    'TABLE'
```

```
|   'TABLES'

TableNameList ::=
    TableName ( ',' TableName )*
```

4.8.2.53.2 删除临时表

删除普通表和临时表的用法如下：

- DROP TEMPORARY TABLE 只能删除本地临时表
- DROP GLOBAL TEMPORARY TABLE 只能删除全局临时表
- DROP TABLE 可以删除普通表或临时表

4.8.2.53.3 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, 1 warning (0.01 sec)
```

```
SHOW WARNINGS;
```

```
+--------+------+----------------------------------------+
| Level | Code | Message                                  |
+--------+------+----------------------------------------+
| Note  | 1051 | Unknown table 'test.table_not_exists' |
+--------+------+----------------------------------------+
1 row in set (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)
```

4.8.2.53.4 MySQL 兼容性

- 目前 RESTRICT 和 CASCADE 仅在语法上支持。

4.8.2.53.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- SHOW TABLES

4.8.2.54 DROP USER

DROP USER 语句用于从TiDB 系统数据库中删除用户。如果用户不存在，使用关键词 IF EXISTS 可避免出现警告。执行 DROP USER 语句需要拥有 CREATE USER 权限。

4.8.2.54.1 语法图

```
DropUserStmt ::=
    'DROP' 'USER' ( 'IF' 'EXISTS' )? UsernameList

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces
```

4.8.2.54.2 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 '%'
```

4.8.2.54.3 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

4.8.2.54.4 另请参阅

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

4.8.2.55 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。

4.8.2.55.1 语法图

```
DropViewStmt ::=
    'DROP' 'VIEW' ( 'IF' 'EXISTS' )? TableNameList RestrictOrCascadeOpt

TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

4.8.2.55.2 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)
```

4.8.2.55.3 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。

4.8.2.55.4 See also

- CREATE VIEW
- DROP TABLE

4.8.2.56 EXECUTE

EXECUTE 语句为服务器端预处理语句提供 SQL 接口。

4.8.2.56.1 语法图

```
ExecuteStmt ::=
    'EXECUTE' Identifier ( 'USING' UserVariable ( ',' UserVariable )* )?
```

4.8.2.56.2 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)
```

4.8.2.56.3 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。

4.8.2.56.4 另请参阅

- PREPARE
- DEALLOCATE

4.8.2.57 EXPLAIN ANALYZE

EXPLAIN ANALYZE 语句的工作方式类似于 EXPLAIN，主要区别在于前者实际上会执行语句。这样可以将查询计划中的估计值与执行时所遇到的实际值进行比较。如果估计值与实际值显著不同，那么应考虑在受影响的表上运行 ANALYZE TABLE。

> **注意：**
>
> 在使用 EXPLAIN ANALYZE 执行 DML 语句时，数据的修改操作会被正常执行。但目前 DML 语句还无法展示执行计划。

4.8.2.57.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|    'DESC'

ExplainStmt ::=
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        <→ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        <→ ExplainableStmt ) )

ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

4.8.2.57.2 EXPLAIN ANALYZE 输出格式

和 EXPLAIN 不同，EXPLAIN ANALYZE 会执行对应的 SQL 语句，记录其运行时信息，和执行计划一并返回出来。因此，可以将 EXPLAIN ANALYZE 视为 EXPLAIN 语句的扩展。EXPLAIN ANALYZE 语句的返回结果相比 EXPLAIN，增加了 actRows，execution info，memory，disk 这几列信息：

属性名	含义
actRows	算子实际输出的数据条数。
execution info	算子的实际执行信息。time 表示从进入算子到离开算子的全部 wall time，包括所有子算子操作的全部执行时间。如果该算子被父算子多次调用(loops)，这个时间就是累积的时间。loops 是当前算子被父算子调用的次数。
memory	算子占用内存空间的大小。
disk	算子占用磁盘空间的大小。

4.8.2.57.3 示例

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

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE id = 1;
```

```
+             +         +         +      +                +
    <->
| id          | estRows | actRows | task | access object | execution info
    <->                                                  | operator info | memory | disk |
+-------------+---------+---------+------+---------------+
    <->
| Point_Get_1 | 1.00    | 1       | root | table:t1      | time:757.205µs, loops:2, Get:{num_rpc
    <-> :1, total_time:697.051µs} | handle:1      | N/A    | N/A  |
+             +         +         +      +                +
    <->
1 row in set (0.01 sec)
```

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

```
+------------------+----------+---------+-----------+--------------+--------------------------------------
    <->
| id                | estRows  | actRows | task      | access object | execution info
    <->
    <-> | operator info                  | memory    | disk |
+------------------+----------+---------+-----------+--------------+--------------------------------------
    <->
| TableReader_5     | 10000.00 | 3       | root      |               | time:278.2µs, loops:2,
    <-> cop_task: {num: 1, max: 437.6µs, proc_keys: 3, rpc_num: 1, rpc_time: 423.9µs,
    <-> copr_cache_hit_ratio: 0.00}
    <->
    <-> | data:TableFullScan_4               | 251 Bytes | N/A  |
| └─TableFullScan_4 | 10000.00 | 3       | cop[tikv] | table:t1      | tikv_task:{time:0s, loops
    <-> :1}, scan_detail: {total_process_keys: 3, total_process_keys_size: 111, total_keys: 4,
    <-> rocksdb: {delete_skipped_count: 0, key_skipped_count: 3, block: {cache_hit_count: 0,
    <-> read_count: 0, read_byte: 0 Bytes}}} | keep order:false, stats:pseudo | N/A        | N/A
    <-> |
+------------------+----------+---------+-----------+--------------+--------------------------------------
    <->
2 rows in set (0.00 sec)
```

4.8.2.57.4 算子执行信息介绍

`execution info` 信息除了基本的 `time` 和 `loop` 信息外，还包含算子特有的执行信息，主要包含了该算子发送 RPC 请求的耗时信息以及其他步骤的耗时。

Point_Get

Point_Get 算子可能包含以下执行信息：

- Get:{num_rpc:1, total_time:697.051µs}：向 TiKV 发送 Get 类型的RPC 请求的数量(num_rpc) 和所有RPC 请求的总耗时(total_time)。
- ResolveLock:{num_rpc:1, total_time:12.117495ms}：读数据遇到锁后，进行 resolve lock 的时间。一般在读写冲突的场景下会出现。
- regionMiss_backoff:{num:11, total_time:2010 ms},tikvRPC_backoff:{num:11, total_time:10691 ms <⟷ }：RPC 请求失败后，会在等待backoff 的时间后重试，包括了backoff 的类型（如 regionMiss，tikvRPC），backoff 等待的总时间(total_time) 和 backoff 的总次数(num)。

Batch_Point_Get

Batch_Point_get 算子的执行信息和 Point_Get 算子类似，不过 Batch_Point_Get 一般向TiKV 发送 BatchGet 类型的RPC 请求来读取数据。

BatchGet:{num_rpc:2, total_time:83.13µs}：向 TiKV 发送 BatchGet 类型的RPC 请求的数量(num_rpc) 和所有RPC 请求的总耗时(total_time)。

TableReader

TableReader 算子可能包含以下执行信息：

```
cop_task: {num: 6, max: 1.07587ms, min: 844.312µs, avg: 919.601µs, p95: 1.07587ms, max_proc_keys:
    <⟷  16, p95_proc_keys: 16, tot_proc: 1ms, tot_wait: 1ms, rpc_num: 6, rpc_time: 5.313996ms,
    <⟷ copr_cache_hit_ratio: 0.00}
```

- cop_task：包含 cop task 的相关信息，如：
  - num：cop task 的数量
  - max,min,avg,p95：所有 cop task 中执行时间的最大值，最小值，平均值和 P95 值。
  - max_proc_keys, p95_proc_keys：所有 cop task 中 tikv 扫描 kv 数据的最大值，P95 值，如果max 和 p95 的值差距很大，说明数据分布不太均匀。
  - rpc_num, rpc_time：向 TiKV 发送 Cop 类型的RPC 请求总数量和总时间。
  - copr_cache_hit_ratio：cop task 请求的 Coprocessor Cache 缓存命中率。Coprocessor Cache 配置。
- backoff：包含不同类型的 backoff 以及等待总耗时。

Insert

Insert 算子可能包含以下执行信息：

```
prepare:109.616µs, check_insert:{total_time:1.431678ms, mem_insert_time:667.878µs, prefetch:763.8
    <⟷ µs, rpc:{BatchGet:{num_rpc:1, total_time:699.166µs},Get:{num_rpc:1, total_time:378.276µs
    <⟷ }}}
```

- prepare：准备写入前的耗时，包括表达式，默认值相关的计算等。
- check_insert：这个信息一般出现在 insert ignore 和 insert on duplicate 语句中，包含冲突检查和写入TiDB 事务缓存的耗时。注意，这个耗时不包含事务提交的耗时。具体包含以下信息：

  - `total_time`：`check_insert` 步骤的总耗时。
  - `mem_insert_time`：将数据写入 TiDB 事务缓存的耗时。
  - `prefetch`：从 TiKV 中获取需要检查冲突的数据的耗时，该步骤主要是向TiKV 发送 `BatchGet` 类型的 RPC 请求的获取数据。
  - `rpc`：向 TiKV 发送RPC 请求的总耗时，一般包含 `BatchGet` 和 `Get` 两种类型的RPC 耗时，其中：
    * `BatchGet` 请求是 `prefetch` 步骤发送的RPC 请求。
    * `Get` 请求是 `insert on duplicate` 语句在执行 `duplicate update` 时发送的RPC 请求。
- `backoff`：包含不同类型的 backoff 以及等待总耗时。

IndexJoin

`IndexJoin` 算子有 1 个 outer worker 和 N 个 inner worker 并行执行，其join 结果的顺序和 outer table 的顺序一致，具体执行流程如下：

1. Outer worker 读取N 行 outer table 的数据，然后包装成一个task 发送给result channel 和 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收task，然后根据task 生成需要读取inner table 的 key ranges 范围，然后读取相应范围的inner table 行数据，并生成一个inner table row 的 hash table。
3. `IndexJoin` 的主线程从result channel 中接收task，然后等待inner worker 执行完这个task。
4. `IndexJoin` 的主线程用 outer table rows 和 inner table rows 的 hash table 做 join。

`IndexJoin` 算子包含以下执行信息：

```
inner:{total:4.297515932s, concurrency:5, task:17, construct:97.96291ms, fetch:4.164310088s,
    <--> build:35.219574ms}, probe:53.574945ms
```

- `inner`：inner worker 的执行信息，具体如下：
  - `total`：inner worker 的总耗时。
  - `concurrency`：inner worker 的数量。
  - `task`：inner worker 处理task 的总数量。
  - `construct`：inner worker 读取task 对应的inner table rows 之前的准备时间。
  - `fetch`：inner worker 读取inner table rows 的总耗时。
  - `build`: inner worker 构造inner table rows 对应的hash table 的总耗时。
- `probe`：`IndexJoin` 主线程用 outer table rows 和 inner table rows 的 hash table 做 join 的总耗时。

IndexHashJoin

`IndexHashJoin` 算子和 `IndexJoin` 算子执行流程类似，也有 1 个 outer worker 和 N 个 inner worker 并行执行，但是其 join 结果的顺序是不和 outer table 一致。具体执行流程如下：

1. Outer worker 读取N 行 out table 的数据，然后包装成一个task 发送给inner worker channel。
2. Inner worker 从 inner worker channel 里面接收task，然后做以下三件事情，其中步骤a 和 b 是并行执行。
   a. 用 outer table rows 生成一个hash table。
   b. 根据task 生成 key 的范围后，读取inner table 相应范围的行数据。
   c. 用 inner table rows 和 outer table rows 的 hash table 做 join，然后把join 结果发送给result channel。

3. IndexHashJoin 的主线程从result channel 中接收join 结果。

IndexHashJoin 算子包含以下执行信息：

```
inner:{total:4.429220003s, concurrency:5, task:17, construct:96.207725ms, fetch:4.239324006s,
    <--> build:24.567801ms, join:93.607362ms}
```

- inner：inner worker 的执行信息，具体如下：
  - total：inner worker 的总耗时。
  - concurrency：inner worker 的数量。
  - task：inner worker 处理task 的总数量。
  - construct：inner worker 读取task 对应的inner table rows 之前的准备时间。
  - fetch：inner worker 读取inner table rows 的总耗时。
  - build: inner worker 构造 outer table rows 对应的hash table 的总耗时。
  - join: inner worker 用 inner table rows 和 outer table rows 的 hash table 做 join 的总耗时。

HashJoin

HashJoin 算子有一个inner worker，一个 outer worker 和 N 个 join worker，其具体执行逻辑如下：

1. inner worker 读取inner table rows 并构造hash table。
2. outer worker 读取 outer table rows，然后包装成task 发送给join worker。
3. 等待第 1 步的hash table 构造完成。
4. join worker 用 task 里面的 outer table rows 和 hash table 做 join，然后把join 结果发送给result channel。
5. HashJoin 的主线程从result channel 中接收join 结果。

HashJoin 算子包含以下执行信息：

```
build_hash_table:{total:146.071334ms, fetch:110.338509ms, build:35.732825ms}, probe:{concurrency
    <--> :5, total:857.162518ms, max:171.48271ms, probe:125.341665ms, fetch:731.820853ms}
```

- build_hash_table: 读取inner table 的数据并构造hash table 的执行信息：
  - total：总耗时。
  - fetch：读取 inner table 数据的总耗时。
  - build：构造 hash table 的总耗时。
- probe: join worker 的执行信息：
  - concurrency：join worker 的数量。
  - total：所有 join worker 执行的总耗时。
  - max：单个 join worker 执行的最大耗时。
  - probe: 用 outer table rows 和 hash table 做 join 的总耗时。
  - fetch：join worker 等待读取 outer table rows 数据的总耗时。

TableFullScan (TiFlash)

在TiFlash 节点上执行的 TableFullScan 算子包含以下执行信息：

```
tiflash_scan:
  { dtfile: {
    total_scanned_packs: 2,
    total_skipped_packs: 1,
    total_scanned_rows: 16000,
    total_skipped_rows: 8192,
    total_rough_set_index_load_time: 2ms,
    total_read_time: 20ms
  },
  total_create_snapshot_time: 1ms
}
```

- `dtfile`：扫表过程中与 DTFile （即 DeltaTree File ）相关的信息；这基本反映了 TiFlash 在 Stable 层数据的读取情况。
  - `total_scanned_packs`：DTFile 内累计读取的 Pack 的数量；Pack 是 TiFlash DTFile 读取的最小粒度，默认情况下每 8192 行构成一个 Pack。
  - `total_skipped_packs`：DTFile 内累计跳过的 Pack 的数量；Pack 会由于 WHERE 条件命中粗糙索引或主键范围过滤而被跳过。
  - `total_scanned_rows`：DTFile 内累计读取的行数；若存在 MVCC 带来的多版本更新或删除，则每个版本独立计数。
  - `total_skipped_rows`：DTFile 内累计跳过的行数。
  - `total_rs_index_load_time`：读取 DTFile 粗糙索引的累计耗时。
  - `total_read_time`：读取 DTFile 数据的累计耗时。
- `total_create_snapshot_time`：扫表过程中创建快照的总耗时。

lock_keys 执行信息

在悲观事务中执行 DML 语句时，算子的执行信息还可能包含 `lock_keys` 的执行信息，示例如下：

```
lock_keys: {time:94.096168ms, region:6, keys:8, lock_rpc:274.503214ms, rpc_count:6}
```

- `time`：执行 `lock_keys` 操作的总耗时。
- `region`：执行 `lock_keys` 操作涉及的Region 数量。
- `keys`：需要 Lock 的 Key 的数量。
- `lock_rpc`：向 TiKV 发送 Lock 类型的RPC 总耗时。因为可以并行发送多个RPC 请求，所以总RPC 耗时可能比 `lock_keys` 操作总耗时大。
- `rpc_count`：向 TiKV 发送 Lock 类型的RPC 总数量。

commit_txn 执行信息

在 `autocommit=1` 的事务中执行写入类型的 DML 语句时，算子的执行信息还会包括事务提交的耗时信息，示例如下：

```
commit_txn: {prewrite:48.564544ms, wait_prewrite_binlog:47.821579, get_commit_ts:4.277455ms,
    ⟷ commit:50.431774ms, region_num:7, write_keys:16, write_byte:536}
```

- prewrite：事务 2PC 提交阶段中 prewrite 阶段的耗时。
- wait_prewrite_binlog:：等待写 prewrite Binlog 的耗时。
- get_commit_ts：获取事务提交时间戳的耗时。
- commit：事务 2PC 提交阶段中，commit 阶段的耗时。
- write_keys：事务中写入 key 的数量。
- write_byte：事务中写入 key-value 的总字节数量，单位是byte。

其它常见执行信息

Coprocessor 算子通常包含 cop_task 和 tikv_task 两部分执行时间信息。前者是TiDB 端记录的时间，从发出请求到接收回复；后者是TiKV Coprocessor 算子自己记录的时间。两者相差较大可能说明在等待、gRPC 或网络上耗时较长。

4.8.2.57.5 MySQL 兼容性

EXPLAIN ANALYZE 是 MySQL 8.0 的功能，但该语句在 MySQL 中的输出格式和可能的执行计划都与TiDB 有较大差异。

4.8.2.57.6 另请参阅

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

4.8.2.58 EXPLAIN

EXPLAIN 语句仅用于显示查询的执行计划，而不执行查询。EXPLAIN ANALYZE 可执行查询，补充 EXPLAIN 语句。如果 EXPLAIN 的输出与预期结果不匹配，可考虑在查询的每个表上执行 ANALYZE TABLE。

语句 DESC 和 DESCRIBE 是 EXPLAIN 的别名。EXPLAIN <tableName> 的替代用法记录在SHOW [FULL] COLUMNS FROM 下。

TiDB 支持 EXPLAIN [options] FOR CONNECTION connection_id，但与 MySQL 的 EXPLAIN FOR 有一些区别，请参见EXPLAIN FOR CONNECTION。

4.8.2.58.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|   'DESC'

ExplainStmt ::=
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        <⟷ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        <⟷ ExplainableStmt ) )
```

```
ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

4.8.2.58.2 EXPLAIN 输出格式

> **注意：**
>
> 使用 MySQL 客户端连接到TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 `EXPLAIN` 的输出结果不再换行，可按右箭头→ 键水平滚动阅读输出结果。

> **注意：**
>
> 在执行计划返回结果中，自 v6.4.0 版本起，特定算子（即 `IndexJoin` 和 `Apply` 算子的 Probe 端所有子节点）的 `estRows` 字段意义与 v6.4.0 之前的有所不同。细节请参考TiDB 执行计划概览。

目前TiDB 的 `EXPLAIN` 会输出 5 列，分别是：`id`，`estRows`，`task`，`access object`，`operator info`。执行计划中每个算子都由这 5 列属性来描述，`EXPLAIN` 结果中每一行描述一个算子。每个属性的具体含义如下：

属性名	含义
id	算子的ID，是算子在整个执行计划中唯一的标识。在TiDB 2.1 中，ID 会格式化地显示算子的树状结构。数据从孩子结点流向父亲结点，每个算子的父亲结点有且仅有一个。
estRows	算子预计将会输出的数据条数，基于统计信息以及算子的执行逻辑估算而来。在 4.0 之前叫count。
task	算子属于的task 种类。目前的执行计划分成为两种task，一种叫root task，在tidb-server 上执行，一种叫cop task，在 TiKV 或者 TiFlash 上并行执行。当前的执行计划在task 级别的拓扑关系是一个root task 后面可以跟许多cop task，root task 使用cop task 的输出结果作为输入。cop task 中执行的也即是TiDB 下推到TiKV 或者 TiFlash 上的任务，每个cop task 分散在 TiKV 或者 TiFlash 集群中，由多个进程共同执行。
access object	算子所访问的数据项信息。包括表 `table`，表分区 `partition` 以及使用的索引 `index`（如果有）。只有直接访问数据的算子才拥有这些信息。
operator info	算子的其它信息。各个算子的 operator info 各有不同，可参考下面的示例解读。

4.8.2.58.3 示例

```
EXPLAIN SELECT 1;
```

```
+-------------------+---------+------+---------------+--------------------------------+
| id                | estRows | task | access object | operator info |
+-------------------+---------+------+---------------+--------------------------------+
| Projection_3      | 1.00    | root |               | 1->Column#1    |
| └─TableDual_4     | 1.00    | root |               | rows:1         |
+-------------------+---------+------+---------------+--------------------------------+
2 rows in set (0.00 sec)
```

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

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

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+--------------------------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+--------------------------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+--------------------------------+
1 row in set (0.00 sec)
```

```
DESC SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+--------------------------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+--------------------------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+--------------------------------+
1 row in set (0.00 sec)
```

```
DESCRIBE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+--------------------------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+--------------------------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
```

```
+-------------+---------+------+---------------+-------------------------------+
1 row in set (0.00 sec)
```

```
EXPLAIN INSERT INTO t1 (c1) VALUES (4);
```

```
+----------+---------+------+---------------+-------------------------------+
| id       | estRows | task | access object | operator info |
+----------+---------+------+---------------+-------------------------------+
| Insert_1 | N/A     | root |               | N/A           |
+----------+---------+------+---------------+-------------------------------+
1 row in set (0.00 sec)
```

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

```
+--------------------------+---------+-----------+---------------+-------------------------------------------------------------------------------+
    <-->
| id                       | estRows | task      | access object | operator info
    <-->                 |
+--------------------------+---------+-----------+---------------+-------------------------------------------------------------------------------+
    <-->
| Update_4                 | N/A     | root      |               | N/A
    <-->                           |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    <-->               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    <-->               |

|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    <--> pseudo |
+--------------------------+---------+-----------+---------------+-------------------------------------------------------------------------------+
    <-->
4 rows in set (0.00 sec)
```

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

```
+--------------------------+---------+-----------+---------------+-------------------------------------------------------------------------------+
    <-->
| id                       | estRows | task      | access object | operator info
    <-->                 |
+--------------------------+---------+-----------+---------------+-------------------------------------------------------------------------------+
    <-->
| Delete_4                 | N/A     | root      |               | N/A
    <-->                           |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    <-->               |
```

```
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    <-->              |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    <--> pseudo |
+---------------------------+---------+-----------+---------------+-------------------------------------------------------------------+
    <-->
4 rows in set (0.01 sec)
```

在 EXPLAIN 语句中，你可以使用 FORMAT = "xxx" 语法指定输出内容的格式。目前支持的输出格式如下：

FORMAT	作用
未指定	未指定 FORMAT 时，默认输出格式为 row
row	EXPLAIN 语句将以表格格式输出结果。更多信息，可参阅TiDB 执行计划概览
brief	EXPLAIN 语句输出结果中的算子ID 将被简化，较之未指定 FORMAT 时输出结果的算子ID 更为简化
dot	EXPLAIN 语句将输出 DOT 格式的执行计划，可以通过 dot 程序（在 graphviz 包中）生成 PNG 文件
tidb_json	EXPLAIN 语句将输出JSON 格式的执行计划，算子信息存放在一个JSON 数组中

在 EXPLAIN 中指定 FORMAT = "brief" 时，示例如下:

```
EXPLAIN FORMAT = "brief" DELETE FROM t1 WHERE c1=3;
```

```
+-------------------------+---------+-----------+---------------+-------------------------------------------------------------------+
    <-->
| id                      | estRows | task      | access object | operator info
    <--> |
+-------------------------+---------+-----------+---------------+-------------------------------------------------------------------+
    <-->
| Delete                  | N/A     | root      |               | N/A
    <--> |
| └─TableReader            | 0.00    | root      |               | data:Selection
    <-->                |
|   └─Selection            | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    <-->              |
|     └─TableFullScan      | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:pseudo
    <--> |
+-------------------------+---------+-----------+---------------+-------------------------------------------------------------------+
    <-->
4 rows in set (0.001 sec)
```

除 MySQL 标准结果格式外，TiDB 还支持 DotGraph，需要在 EXPLAIN 中指定 FORMAT = "dot"，示例如下：

```
create table t(a bigint, b bigint);
explain format = "dot" select A.a, B.b from t A join t B on A.a > B.b where A.a < 10;
```

```
+
    <->
| dot contents
    <->
    <-> |
+
    <->
|
digraph Projection_8
{ subgraph cluster8{
node [style=filled, color=lightgrey]
color=black
label = "root"
"Projection_8" -> "HashJoin_9"
"HashJoin_9" -> "TableReader_13"
"HashJoin_9" -> "Selection_14"
"Selection_14" -> "TableReader_17"
}
subgraph cluster12{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_12" -> "TableFullScan_11"
}
subgraph cluster16{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_16" -> "TableFullScan_15"
}
"TableReader_13" -> "Selection_12"
"TableReader_17" -> "Selection_16"
}
  |
+
```

如果你的计算机上安装了 dot 程序，可使用以下方法生成 PNG 文件：

```
1 row in set (0.00 sec)
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：

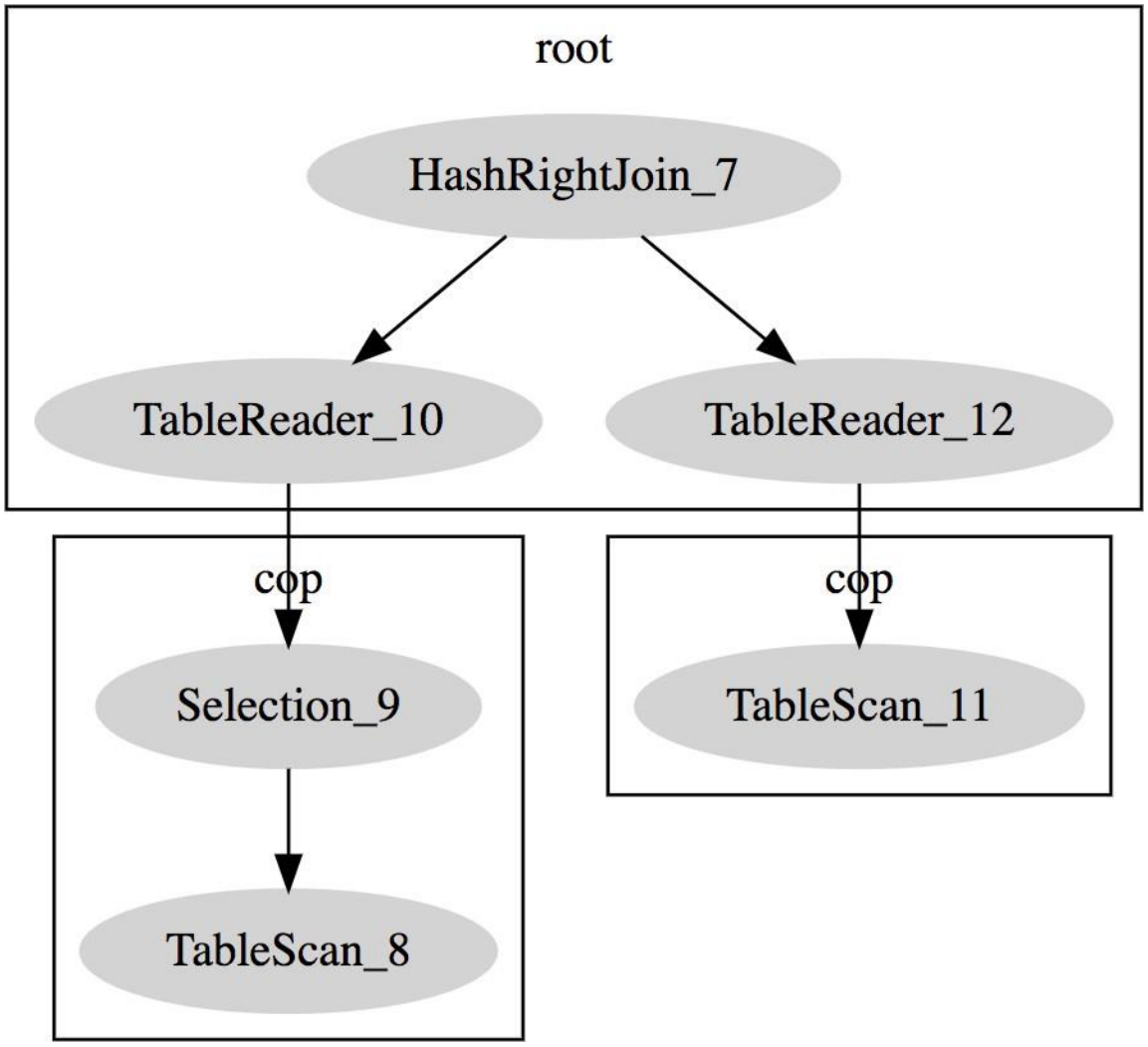


图 81: Explain Dot

在 EXPLAIN 中指定 FORMAT = "tidb_json" 时，示例如下：

```
CREATE TABLE t(id int primary key, a int, b int, key(a));
EXPLAIN FORMAT = "tidb_json" SELECT id FROM t WHERE a = 1;
```

```
+
    ↪
| TiDB_JSON
    ↪
    ↪ |
+
    ↪
```

```
| [
    {
        "id": "Projection_4",
        "estRows": "10.00",
        "taskType": "root",
        "operatorInfo": "test.t.id",
        "subOperators": [
            {
                "id": "IndexReader_6",
                "estRows": "10.00",
                "taskType": "root",
                "operatorInfo": "index:IndexRangeScan_5",
                "subOperators": [
                    {
                        "id": "IndexRangeScan_5",
                        "estRows": "10.00",
                        "taskType": "cop[tikv]",
                        "accessObject": "table:t, index:a(a)",
                        "operatorInfo": "range:[1,1], keep order:false, stats:pseudo"
                    }
                ]
            }
        ]
    }
]
 |
+
```

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

在输出结果中，id、estRows、taskType、accessObject、operatorInfo 与默认格式输出的列含义相同。subOperators 是存放子节点的数组，子节点的字段及含义与父节点均相同。若某字段缺失，则说明该字段为空。

4.8.2.58.4 MySQL 兼容性

- EXPLAIN 的格式和TiDB 中潜在的执行计划都与 MySQL 有很大不同。
- TiDB 不支持 FORMAT=JSON 或 FORMAT=TREE 选项。
- TiDB 支持的 FORMAT=tidb_json 是对当前默认 EXPLAIN 格式的JSON 编码，与 MySQL 的 FORMAT=JSON 结果的格式、字段信息都不同。

4.8.2.58.5 EXPLAIN FOR CONNECTION

EXPLAIN FOR CONNECTION 用于获得一个连接中当前正在执行 SQL 的执行计划或者是最后执行 SQL 的执行计划，其输出格式与 EXPLAIN 完全一致。但TiDB 中的实现与 MySQL 不同，除了输出格式之外，还有以下区别：

- 如果连接处于睡眠状态，MySQL 返回为空，而TiDB 返回的是最后执行的查询计划。
- 如果获取当前会话连接的执行计划，MySQL 会报错，而TiDB 会正常返回。
- MySQL 的文档中指出，MySQL 要求登录用户与被查询的连接相同，或者拥有 PROCESS 权限，而TiDB 则要求登录用户与被查询的连接相同，或者拥有 SUPER 权限。

4.8.2.58.6 另请参阅

- 理解TiDB 执行计划
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE

4.8.2.59 FLASHBACK CLUSTER TO TIMESTAMP

TiDB v6.4.0 引入了 `FLASHBACK CLUSTER TO TIMESTAMP` 语法，其功能是将集群的数据恢复到特定的时间点。

> **注意：**
>
> `FLASHBACK CLUSTER TO TIMESTAMP` 是用最新的时间戳写入特定时间点的旧数据，但不会删除当前数据，所以在使用前请确保集群有足够的存储空间来同时容纳旧数据和当前数据。

4.8.2.59.1 语法

```
FLASHBACK CLUSTER TO TIMESTAMP '2022-09-21 16:02:50';
```

语法图

```
FlashbackToTimestampStmt ::=
    "FLASHBACK" "CLUSTER" "TO" "TIMESTAMP" stringLit
```

4.8.2.59.2 注意事项

- FLASHBACK 指定的时间点需要在Garbage Collection (GC) life time 时间内。你可以使用系统变量tidb_gc_life_time <—> 配置数据的历史版本的保留时间（默认值是 10m0s）。可以使用以下 SQL 语句查询当前的 safePoint，即 GC 已经清理到的时间点：

  ```
  SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
  ```

- 执行 FLASHBACK CLUSTER SQL 语句的用户需要有 SUPER 权限。
- FLASHBACK CLUSTER 不支持回退修改 PD 相关信息的 DDL，如 ALTER TABLE ATTRIBUTE、ALTER TABLE <—> REPLICA、CREATE PLACEMENT POLICY 等。
- FLASHBACK CLUSTER 指定的时间点不能存在未执行完成的 DDL 记录。若存在，TiDB 会拒绝该 DDL 操作。

- 在执行 FLASHBACK CLUSTER TO TIMESTAMP 前，TiDB 会主动断开所有相关表上的连接，并禁止对这些表进行读写操作，直到 FLASHBACK CLUSTER 完成。
- FLASHBACK CLUSTER TO TIMESTAMP 命令不能取消，一旦开始执行TiDB 会一直重试，直到成功。
- 在 FLASHBACK CLUSTER 执行期间，若有数据备份需求，只支持使用 BR 命令行工具进行快照备份，并需要指定早于 FLASHBACK CLUSTER 开始时间的 BackupTS。同时，在执行 FLASHBACK CLUSTER 期间，开启日志备份的操作会失败，请等待 FLASHBACK CLUSTER 结束后再尝试开启日志备份。
- 若 FLASHBACK CLUSTER 导致了元信息（表结构、库结构）的回滚，则相关的修改不会被TiCDC 同步。因此，用户需主动暂停任务，待 FLASHBACK CLUSTER 完成后将上下游的 schema 定义手动同步一致，然后重新创建TiCDC changefeed。

4.8.2.59.3 示例

恢复新插入的数据：

```
mysql> CREATE TABLE t(a INT);
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT * FROM t;
Empty set (0.01 sec)

mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2022-09-28 17:24:16 |
+---------------------+
1 row in set (0.02 sec)

mysql> INSERT INTO t VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> SELECT * FROM t;
+----------+
| a    |
+----------+
|    1 |
+----------+
1 row in set (0.01 sec)

mysql> FLASHBACK CLUSTER TO TIMESTAMP '2022-09-28 17:24:16';
Query OK, 0 rows affected (0.20 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)
```

如果 FLASHBACK CLUSTER 指定的时间点有未完成的 DDL 记录，那么 FLASHBACK CLUSTER 将执行失败：

```
mysql> ALTER TABLE t ADD INDEX k(a);
Query OK, 0 rows affected (0.56 sec)

mysql> ADMIN SHOW DDL JOBS 1;
+--
    <-> ------+---------+---------------------+-----------------------+-------------+-----------++----------------------+-
    <->
 | JOB_ID | DB_NAME | TABLE_NAME             | JOB_TYPE                | SCHEMA_STATE | SCHEMA_ID |
    <-> TABLE_ID | ROW_COUNT | CREATE_TIME          | START_TIME           | END_TIME             |
    <-> STATE  |
+--
    <-> ------+---------+---------------------+-----------------------+-------------+-----------++----------------------+-
    <->
|     84 | test    | t                     | add index /* ingest */ | public       |          2 |
     <->       82 |          0 | 2023-01-29 14:33:11 | 2023-01-29 14:33:11 | 2023-01-29 14:33:12 |
    <-> synced |
+--
    <-> ------+---------+---------------------+-----------------------+-------------+-----------++----------------------+-
    <->
1 rows in set (0.01 sec)

mysql> FLASHBACK CLUSTER TO TIMESTAMP '2023-01-29 14:33:12';
ERROR 1105 (HY000): Detected another DDL job at 2023-01-29 14:33:12 +0800 CST, can't do flashback
```

可以通过日志查看 FLASHBACK 执行进度，具体的日志如下所示：

```
[2022/10/09 17:25:59.316 +08:00] [INFO] [cluster.go:463] ["flashback cluster stats"] ["complete
    <-> regions"=9] ["total regions"=10] []
```

4.8.2.59.4 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.60 FLASHBACK DATABASE

TiDB v6.4.0 引入了FLASHBACK DATABASE 语法，其功能是在Garbage Collection (GC) life time 时间内，可以用FLASHBACK <-> DATABASE 语句来恢复被 DROP 删除的数据库以及数据。

可以使用系统变量tidb_gc_life_time 配置数据的历史版本的保留时间（默认值是 10m0s）。可以使用以下 SQL 语句查询当前的 safePoint，即 GC 已经清理到的时间点：

```
SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
```

只要一个数据库是在 tikv_gc_safe_point 时间之后被 DROP，即可以用 FLASHBACK DATABASE 语法来恢复。

4.8.2.60.1 语法

```
FLASHBACK DATABASE DBName [TO newDBName]
```

语法图

```
FlashbackDatabaseStmt ::=
    'FLASHBACK' DatabaseSym DBName FlashbackToNewName

FlashbackToNewName  ::=
    ( 'TO' Identifier )?
```

4.8.2.60.2 注意事项

- 如果数据库被删除的时间超过了GC life time (`tikv_gc_safe_point`)，就无法使用 `FLASHBACK DATABASE` 语句来恢复被删除的数据了，否则会返回错误。错误类似于 `ERROR 1105 (HY000): Can't find dropped database 'test' in GC safe point 2022-11-06 16:10:10 +0800 CST`。
- 不能用 `FLASHBACK DATABASE` 多次恢复同一个被删除的数据库，因为 `FLASHBACK DATABASE` 所恢复数据库的 schema ID 和原被删除数据库的 schema ID 一致，多次恢复同一数据库会导致重复的 schema ID。 在TiDB中，所有数据库的 schema ID 必须全局唯一。
- 在开启TiDB Binlog时，使用 `FLASHBACK DATABASE` 需要注意以下情况：
  - 下游从集群也需要支持 `FLASHBACK DATABASE`。
  - 从集群的GC life time 一定要长于主集群的GC life time。否则上下游同步存在的延迟可能也会造成下游恢复数据失败。
  - 如果TiDB Binlog同步出错，则需要在TiDB Binlog中过滤掉该数据库，同时手动全量重新导入该数据库的数据。

4.8.2.60.3 示例

- 恢复被 `DROP` 删除的 `test` 数据库：

```
DROP DATABASE test;
```

```
FLASHBACK DATABASE test;
```

- 恢复被 `DROP` 删除的 `test` 数据库并重命名为 `test1`：

```
DROP DATABASE test;
```

```
FLASHBACK DATABASE test TO test1;
```

4.8.2.60.4 MySQL兼容性

该语句是TiDB对MySQL语法的扩展。

4.8.2.61 FLASHBACK TABLE

在 TiDB 4.0 中，引入了 FLASHBACK TABLE 语法，其功能是在 Garbage Collection (GC) life time 时间内，可以用 FLASHBACK TABLE 语句来恢复被 DROP 或 TRUNCATE 删除的表以及数据。

可以使用系统变量 tidb_gc_life_time 配置数据的历史版本的保留时间（默认值是 10m0s）。可以使用以下 SQL 语句查询当前的 safePoint，即 GC 已经清理到的时间点：

```
sql SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
```

只要被 DROP 或 TRUNCATE 删除的表是在 tikv_gc_safe_point 时间之后，都能用 FLASHBACK TABLE 语法来恢复。

4.8.2.61.1 语法

```
FLASHBACK TABLE table_name [TO other_table_name]
```

语法图

```
FlashbackTableStmt ::=
    'FLASHBACK' 'TABLE' TableName FlashbackToNewName

TableName ::=
    Identifier ( '.' Identifier )?

FlashbackToNewName ::=
    ( 'TO' Identifier )?
```

4.8.2.61.2 注意事项

如果删除了一张表并过了 GC lifetime，就不能再用 FLASHBACK TABLE 语句来恢复被删除的数据了，否则会返回错误，错误类似于 Can't find dropped/truncated table 't' in GC safe point 2020-03-16 16:34:52 +0800 CST。

在开启 TiDB Binlog 时使用 FLASHBACK TABLE 需要注意以下情况：

- 下游从集群也支持 FLASHBACK TABLE
- 从集群的 GC lifetime 一定要长于主集群的 GC lifetime。上下游同步存在的延迟可能也会造成下游恢复数据失败。

如果 Binlog 同步出错，则需要在 Binlog 过滤掉该表，同时手动全量重新导入该表的数据。

4.8.2.61.3 示例

- 恢复被 DROP 删除的表数据：

```
DROP TABLE t;
```

```
FLASHBACK TABLE t;
```

- 恢复被 TRUNCATE 的表数据，由于被 TRUNCATE 的表还存在，所以需要重命名被恢复的表，否则会报错表 t 已存在。

```
TRUNCATE TABLE t;
```

```
FLASHBACK TABLE t TO t1;
```

4.8.2.61.4 工作原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 mysql.gc_delete_range 表。TiDB 后台的GC Worker 会定期从 mysql.gc_delete_range 表中取出超过GC lifetime 相关范围的 key 进行删除。

所以，FLASHBACK TABLE 只需要在GC Worker 还没删除表数据前，恢复表的元信息并删除mysql.gc_delete_range 表中相应的行记录即可。恢复表的元信息可以用TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。下面是 FLASHBACK TABLE t TO t1 的工作流程：

1. 从 DDL History job 中查找 drop table 或者 truncate table 类型的操作，且操作的表名是t 的第一个 DDL job，若没找到，则返回错误。
2. 检查 DDL job 的开始时间，是否在 tikv_gc_safe_point 之前。如果是tikv_gc_safe_point 之前，说明被 DROP 或 TRUNCATE 删除的表已经被GC 清理掉，返回错误。
3. 用 DDL job 的开始时间作为 snapshot 读取历史数据，读取表的元信息。
4. 删除 mysql.gc_delete_range 中和表t 相关的GC 任务。
5. 将表的元信息中的 name 修改成 t1，并用该元信息新建一个表。注意：这里只是修改了表名，但是 table ID 不变，依旧是之前被删除的表t 的 table ID。

可以发现，从表t 被删除，到表t 被 FLASHBACK 恢复到 t1，一直都是对表的元信息进行操作，而表的用户数据一直未被修改过。被恢复的表 t1 和之前被删除的表t 的 table ID 相同，所以表 t1 才能读取表t 的用户数据。

> 注意：
>
> 不能用 FLASHBACK 多次恢复同一个被删除的表，因为 FLASHBACK 所恢复表的table ID 还是被删除表的table ID，而TiDB 要求所有还存在的表table ID 必须全局唯一。

FLASHBACK TABLE 是通过快照读获取表的元信息后，再走一次类似于CREATE TABLE 的建表流程，所以FLASHBACK <⟶ TABLE 实际上也是一种 DDL 操作。

4.8.2.61.5 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.62 FLUSH PRIVILEGES

FLUSH PRIVILEGES 语句可触发 TiDB 从权限表中重新加载权限的内存副本。在对如 mysql.user 一类的表进行手动编辑后，应当执行 FLUSH PRIVILEGES。使用如 GRANT 或 REVOKE 一类的权限语句后，不需要执行 FLUSH PRIVILEGES 语句。执行 FLUSH PRIVILEGES 语句的用户需要拥有 RELOAD 权限。

4.8.2.62.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

4.8.2.62.2 示例

```
FLUSH PRIVILEGES;
```

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

4.8.2.62.3 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。

4.8.2.62.4 另请参阅

- GRAT <privileges>
- REVOKE <privileges>
- Privilege Management

4.8.2.63 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。

4.8.2.63.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?
```

```
FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

4.8.2.63.2 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)
```

4.8.2.63.3 MySQL 兼容性

- FLUSH STATUS 语句仅用于提供 MySQL 兼容性。

4.8.2.63.4 另请参阅

- SHOW [GLOBAL | SESSION] STATUS

4.8.2.64 FLUSH TABLES

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

4.8.2.64.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|    LogTypeOpt 'LOGS'
|    TableOrTables TableNameListOpt WithReadLockOpt

LogTypeOpt ::=
    ( 'BINARY' | 'ENGINE' | 'ERROR' | 'GENERAL' | 'SLOW' )?
```

```
TableOrTables ::=
    'TABLE'
|   'TABLES'

TableNameListOpt ::=
    TableNameList?

WithReadLockOpt ::=
    ( 'WITH' 'READ' 'LOCK' )?
```

4.8.2.64.2 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
```

4.8.2.64.3 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。

4.8.2.64.4 另请参阅

- Read historical data

4.8.2.65 GRANT <privileges>

`GRANT <privileges>` 语句用于为TiDB 中已存在的用户分配权限。TiDB 中的权限系统同 MySQL 一样，都基于数据库/表模式来分配凭据。执行 `GRANT <privileges>` 语句需要拥有分配的权限，并且拥有 `GRANT OPTION` 权限。

4.8.2.65.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        <-> WithGrantOptionOpt

PrivElemList ::=
```

```
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
     'ALL' 'PRIVILEGES'?
|     'ALTER' 'ROUTINE'?
|    'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|    'TRIGGER'
|    'DELETE'
|    'DROP' 'ROLE'?
|     'PROCESS'
|    'EXECUTE'
|    'INDEX'
|    'INSERT'
|    'SELECT'
|    'SUPER'
|    'SHOW' ( 'DATABASES' | 'VIEW' )
|    'UPDATE'
|     'GRANT' 'OPTION'
|     'REFERENCES'
|     'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|     'USAGE'
|    'RELOAD'
|    'FILE'
|    'CONFIG'
|    'LOCK' 'TABLES'
|    'EVENT'
|    'SHUTDOWN'

ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|     Identifier ( '.' ( '*' | Identifier ) )?

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

4.8.2.65.2 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)
```

4.8.2.65.3 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。
- GRANT <privileges> 语句执行成功后，在TiDB 中语句执行的结果会在当前连接立即生效，而 MySQL 中部分权限的结果需要等到之后的连接才生效。见 TiDB #39356。

4.8.2.65.4 另请参阅

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

4.8.2.66 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。

4.8.2.66.1 语法图

```
GrantRoleStmt ::=
    'GRANT' RolenameList 'TO' UsernameList

RolenameList ::=
```

```
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

4.8.2.66.2 示例

以 root 用户连接TiDB：

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

创建新角色 analyticsteam 和新用户 jennifer：

```
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)
```

以 jennifer 用户连接TiDB：

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

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analyticsteam 角色相关联的权限：

```
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)
```

以 root 用户连接TiDB：

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

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接TiDB：

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

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
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)
```

4.8.2.66.3 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。

4.8.2.66.4 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

4.8.2.67 INSERT

使用 `INSERT` 语句在表中插入新行。

4.8.2.67.1 语法图

```
InsertIntoStmt ::=
    'INSERT' TableOptimizerHints PriorityOpt IgnoreOptional IntoOpt TableName
        <⟷ PartitionNameListOpt InsertValues OnDuplicateKeyUpdate

TableOptimizerHints ::=
    hintComment?

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IgnoreOptional ::=
    'IGNORE'?

IntoOpt  ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?

InsertValues ::=
```

```
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        <→ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*

OnDuplicateKeyUpdate ::=
    ( 'ON' 'DUPLICATE' 'KEY' 'UPDATE' AssignmentList )?
```

4.8.2.67.2 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)
```

4.8.2.67.3 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。

4.8.2.67.4 另请参阅

- DELETE
- SELECT
- UPDATE
- REPLACE

4.8.2.68 KILL

KILL 语句可以终止当前TiDB 集群中任意一个TiDB 实例中的某个连接。

4.8.2.68.1 语法图

```
KillStmt ::= 'KILL' 'TIDB'? ( 'CONNECTION' | 'QUERY' )? CONNECTION_ID
```

4.8.2.68.2 示例

查询当前集群中所有活跃查询，并终止其中某一个连接：

```
SELECT ID, USER, INSTANCE, INFO FROM INFORMATION_SCHEMA.CLUSTER_PROCESSLIST;
```

```
+---------------------+------+-----------------+-------------------------------------------------------------------
    <->
| ID                  | USER | INSTANCE        | INFO
    <->                                                                          |
+---------------------+------+-----------------+-------------------------------------------------------------------
    <->
| 8306449708033769879 | root | 127.0.0.1:10082 | select sleep(30), 'foo'
    <->                                                        |
| 5857102839209263511 | root | 127.0.0.1:10080 | select sleep(50)
    <->                                                              |
| 5857102839209263513 | root | 127.0.0.1:10080 | SELECT ID, USER, INSTANCE, INFO FROM
    <-> INFORMATION_SCHEMA.CLUSTER_PROCESSLIST |
+                     +      +                 +
 ---------------------+------+-----------------+-------------------------------------------------------------------
    <->
```

```
KILL 5857102839209263511;
```

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

4.8.2.68.3 MySQL 兼容性

- MySQL 的 KILL 语句仅能终止当前连接的 MySQL 实例上的连接，TiDB 的 KILL 语句能终止整个集群中任意一个TiDB 实例上的连接。
- 暂时不支持使用 MySQL 命令行ctrl+c 终止查询或连接。

4.8.2.68.4 行为变更说明

TiDB 从 v6.1.0 起新增Global Kill 功能（由enable-global-kill 配置项控制，默认启用）。启用 Global Kill 功能时，KILL 语句和 KILL TIDB 语句均能跨节点终止查询或连接，且无需担心错误地终止其他查询或连接。当你使用客户端连接到任何一个TiDB 节点执行 KILL 语句或 KILL TIDB 语句时，该语句会被转发给对应的TiDB 节点。当客户端和TiDB 中间有代理时，KILL 及 KILL TIDB 语句也会被转发给对应的TiDB 节点执行。

对于TiDB v6.1.0 之前的版本，或未启用Global Kill 功能时：

- KILL 语句与 MySQL 不兼容，负载均衡器后面通常放有多个TiDB 服务器，这种不兼容有助于防止在错误的 TiDB 服务器上终止连接。你需要显式地增加 TIDB 后缀，通过执行 KILL TIDB 语句来终止当前连接的 TiDB 实例上的其他连接。
- 强烈不建议在配置文件里设置compatible-kill-query = true，除非你确定客户端将始终连接到同一个 TiDB 节点。这是因为当你在默认的 MySQL 客户端按下ctrl+c 时，客户端会开启一个新连接，并在这个新连接中执行 KILL 语句。此时，如果客户端和TiDB 中间有代理，新连接可能会被路由到其他的TiDB 节点，从而错误地终止其他会话。
- KILL TIDB 语句是TiDB 的扩展语法，其功能与MySQL 命令KILL [CONNECTION|QUERY] 和 MySQL 命令行ctrl+c 相同。在同一个TiDB 节点上，你可以安全地使用 KILL TIDB 语句。

4.8.2.68.5 另请参阅

- SHOW [FULL] PROCESSLIST
- CLUSTER_PROCESSLIST

4.8.2.69 LOAD DATA

LOAD DATA 语句用于将数据批量加载到TiDB 表中。

在 v7.0.0 版本支持以下功能：

- 支持从 S3、GCS 导入数据
- 新增参数 FIELDS DEFINED NULL BY

> **警告：**
>
> v7.0.0 新增支持从 S3、GCS 导入数据和 FIELDS DEFINED NULL BY 参数为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现bug，请在 GitHub 上提 issue 反馈。

4.8.2.69.1 语法图

```
LoadDataStmt ::=
    'LOAD' 'DATA' LocalOpt 'INFILE' stringLit DuplicateOpt 'INTO' 'TABLE' TableName CharsetOpt
        <⟶ Fields Lines IgnoreLines ColumnNameOrUserVarListOptWithBrackets LoadDataSetSpecOpt

LocalOpt ::= ('LOCAL')?

Fields ::=
    ('TERMINATED' 'BY' stringLit
    | ('OPTIONALLY')? 'ENCLOSED' 'BY' stringLit
    | 'ESCAPED' 'BY' stringLit
    | 'DEFINED' 'NULL' 'BY' stringLit ('OPTIONALLY' 'ENCLOSED')?)?
```

4.8.2.69.2 参数说明

LOCAL

你可以使用 LOCAL 来指定导入位于客户端的数据文件，此时传入文件参数必须为客户端文件系统路径。

S3/GCS 路径

如果你不指定 LOCAL，则文件参数必须是有效的 S3/GCS URI 路径，详见外部存储。

当数据文件存储在 S3/GCS 上时，你可以导入单个文件，也可使用通配符* 来匹配需要导入的多个文件。注意通配符不会递归处理子目录下相关的文件。以数据存储在 S3 为例，示例如下:

- 导入单个文件：s3://<bucket-name>/path/to/data/foo.csv
- 导入指定路径下的所有文件：s3://<bucket-name>/path/to/data/*
- 导入指定路径下的所有以 .csv 结尾的文件：s3://<bucket-name>/path/to/data/*.csv
- 导入指定路径下所有以 foo 为前缀的文件：s3://<bucket-name>/path/to/data/foo*
- 导入指定路径下以 foo 为前缀、以 .csv 结尾的文件：s3://<bucket-name>/path/to/data/foo*.csv

Fields、Lines、Ignore Lines

你可以使用 Fields 和 Lines 参数来指定如何处理数据格式：

- 使用 FIELDS TERMINATED BY 来指定数据的分隔符号。
- 使用 FIELDS ENCLOSED BY 来指定数据的包围符号。
- 如果你希望以某个字符为结尾来切分行数据，可以使用 LINES TERMINATED BY 来指定行的终止符。

可以使用 DEFINED NULL BY 来指定数据文件中如何表示 NULL 值。

- 与 MySQL 行为一致，如果 ESCAPED BY 不为空时，例如是默认值\，那么 \N 会被认为是 NULL 值。
- 如果使用 DEFINED NULL BY，例如 DEFINED NULL BY 'my-null'，my-null 会被认为是 NULL 值。
- 如果使用 DEFINED NULL BY ... OPTIONALLY ENCLOSED，例如 DEFINED NULL BY 'my-null' OPTIONALLY <⟶ ENCLOSED，my-null 和 "my-null"（假设 ENCLOSED BY '"'）会被认为是 NULL 值。
- 如果没有使用 DEFINED NULL BY 或者 DEFINED NULL BY ... OPTIONALLY ENCLOSED，但使用了 ENCLOSED <⟶ BY，例如 ENCLOSED BY '"'，那么 NULL 会被认为是 NULL 值。这个行为与 MySQL 一致。
- 其他情况不会被认为是 NULL 值。

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'
```

当不指定处理数据的参数时，将按以下参数处理：

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES TERMINATED BY '\n' STARTING BY ''
```

你可以通过 IGNORE <number> LINES 参数来忽略文件开始的 <number> 行。例如，可以使用 IGNORE 1 LINES 来忽略文件的第一行。

4.8.2.69.3 示例

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。可以参考文档解决：ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    <—> ' INTO TABLE trips FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
    <—> IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station,
    <—> end_station_number, end_station, bike_number, member_type);
```

```
Query OK, 815264 rows affected (39.63 sec)
Records: 815264  Deleted: 0  Skipped: 0  Warnings: 0
```

LOAD DATA 也支持使用十六进制 ASCII 字符表达式或二进制 ASCII 字符表达式作为 FIELDS ENCLOSED BY 和 FIELDS TERMINATED BY 的参数。示例如下：

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    <—> ' INTO TABLE trips FIELDS TERMINATED BY x'2c' ENCLOSED BY b'100010' LINES TERMINATED BY '
    <—> \r\n' IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station
    <—> , end_station_number, end_station, bike_number, member_type);
```

以上示例中 x'2c' 是字符，的十六进制表示，b'100010' 是字符" 的二进制表示。

4.8.2.69.4 MySQL 兼容性

TiDB 中的 LOAD DATA 语句语法上兼容 MySQL（除字符集选项被解析但会被忽略以外）。若发现任何语法兼容性差异，请在GitHub 上提交issue。

> 注意：
>
> - 在 TiDB v4.0.0 之前的版本中，LOAD DATA 语句每 20000 行进行一次提交。
> - 从 TiDB v4.0.0 开始一直到TiDB v6.6.0 的版本，TiDB 默认在一个事务中提交所有行。
> - 从 TiDB v4.0.0 及以前版本升级后，可能出现 ERROR 8004 (HY000)at line 1: Transaction <—> is too large, size: 100000058 错误。要解决该问题，建议调大 tidb.toml 文件中的txn-total-size-limit 值。如果无法增加此限制，还可以将tidb_dml_batch_size 的值设置为 20000 来恢复升级前的行为。注意自 v7.0.0 起，tidb_dml_batch_size 对 LOAD DATA 语句不再生效。
> - 无论以多少行为一个事务提交，LOAD DATA 都不会被显式事务中的ROLLBACK 语句回滚。
> - LOAD DATA 语句始终以乐观事务模式执行，不受TiDB 事务模式设置的影响。

4.8.2.69.5 另请参阅

- INSERT

- 乐观事务模型
- 悲观事务模式

4.8.2.70 LOAD STATS

`LOAD STATS` 语句用于将统计信息加载到TiDB 中。

4.8.2.70.1 语法图

```
LoadStatsStmt ::=
    'LOAD' 'STATS' stringLit
```

4.8.2.70.2 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)
```

4.8.2.70.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.70.4 另请参阅

- 统计信息

4.8.2.71 LOCK STATS

`LOCK STATS` 语句用于锁定统计信息。

> **警告：**
>
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

4.8.2.71.1 语法图

```
LockStatsStmt ::=
    'LOCK' 'STATS' TableNameList

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?
```

4.8.2.71.2 示例

创建表t，插入一些数据，在未锁定表t 的统计信息的情况下成功执行 ANALYZE 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
+       +      +
| Level | Code | Message                                                
```

```
mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> show warnings;
+---------+-------+-----------------------------------------------------------------------+
| Level   | Code | Message                                                                |
+---------+-------+-----------------------------------------------------------------------+
| Note    | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
| Warning | 1105 | skip analyze locked table: t                                           |
+---------+-------+-----------------------------------------------------------------------+
2 rows in set (0.00 sec)
```

解锁表t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+       +      +
 -------+------+-----------------------------------------------------------------------+
| Level | Code | Message                                                                |
                                                                                       |
+-------+------+-----------------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+-----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

4.8.2.71.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.71.4 另请参阅

- 统计信息
- UNLOCK STATS
- SHOW STATS_LOCKED

4.8.2.72 LOCK TABLES 和 UNLOCK TABLES

> **警告：**
>
> LOCK TABLES 和 UNLOCK TABLES 目前为实验特性，不建议在生产环境中使用。

客户端会话可以使用 LOCK TABLES 语句获取表锁，以便和其他会话合作访问表，或者防止其他会话修改表。会话只能为自己获取或释放锁。一个会话无法为另一个会话获取表锁或释放另一个会话持有的表锁。

LOCK TABLES 可以为当前客户端会话获取表锁。你可以获取普通表的表锁， 但你必须拥有锁定对象的 LOCK TABLES 和 SELECT 权限。

UNLOCK TABLES 显式释放当前会话持有的所有表锁。LOCK TABLES 在获取新锁之前会隐式释放当前会话持有的所有表锁。

表锁可以防止其他会话的读取或写入。持有 WRITE 锁的会话可以执行表级操作，例如 DROP TABLE 或 TRUNCATE <—> TABLE。

> **警告：**
> 开启表锁功能需要在所有TiDB 实例的配置文件中设置enable-table-lock 为 true。

4.8.2.72.1 语法图

```
LockTablesDef
         ::= 'LOCK' ( 'TABLES' | 'TABLE' ) TableName LockType ( ',' TableName LockType)*


UnlockTablesDef
         ::= 'UNLOCK' 'TABLES'

LockType
          ::= 'READ' ('LOCAL')?
            | 'WRITE' ('LOCAL')?
```

4.8.2.72.2 获取表锁

你可以在会话中使用 LOCK TABLES 语句获取表锁。表锁有以下类型：

READ 锁：

- 持有 READ 锁的会话可以读表，但不能写入。
- 多个会话可以同时获取同一个表的 READ 锁。
- 其他会话可以在不显式获取 READ 锁的情况下读表。

READ LOCAL 锁只是语法兼容 MySQL，实际上并不支持。

WRITE 锁：

- 持有 WRITE 锁的会话可以读取和写入表。
- 只有持有 WRITE 锁的会话才能访问该表。在释放锁之前，其他会话不能读取或写入该表。

WRITE LOCAL 锁：

- 持有 WRITE LOCAL 锁的会话可以读取和写入表。
- 只有持有 WRITE LOCAL 锁的会话才能写入该表，但其他会话依然可以读取该表的数据。

如果 LOCK TABLES 语句想要获取的表锁被其他会话持有且必须等待锁释放时，则 LOCK TABLES 语句会执行报错，例如：

```
> LOCK TABLES t1 read;
ERROR 8020 (HY000): Table 't1' was locked in WRITE by server: f4799bcb-cad7-4285-8a6d-23
    <--> d3555173f1_session: 2199023255959
```

以上错误信息表明，在 TiDB f4799bcb-cad7-4285-8a6d-23d3555173f1 中，ID 为 2199023255959 的会话已经持有表 t1 的 WRITE 锁，所以，当前会话无法获取表 t1 的 READ 锁。

不能在单个 LOCK TABLES 语句中多次获取同一个表的锁。

```
> LOCK TABLES t WRITE, t READ;
ERROR 1066 (42000): Not unique table/alias: 't'
```

4.8.2.72.3 释放表锁

释放会话持有的表锁时，将同时释放该会话所有持有的表锁。会话可以显式或隐式释放表锁：

- 会话可以使用 UNLOCK TABLES 语句显式释放其持有的所有表锁。
- 当会话使用 LOCK TABLES 语句来获取表锁，而且同时已经持有其他表锁时，则在获取新表锁之前，将隐式释放其已经持有的所有表锁。

当客户端会话的连接终止（无论是正常还是异常），TiDB 都会隐式释放会话持有的所有表锁。如果客户端重新连接，锁将不再有效。因此，通常不建议在客户端使用自动重新连接，因为在开启自动重新连接时，如果发生重新连接，任何表锁或当前事务都将丢失，而且不会通知客户端。禁用自动重新连接后，如果连接断开，则客户端会话发出的下一条语句将会收到报错信息。客户端可以检测到错误并采取适当的操作，例如重新获取锁或重做事务。

4.8.2.72.4 表锁的使用限制和条件

你可以安全地使用 KILL 语句终止已经持有表锁的会话。

不支持获取以下数据库中表的表锁：

- INFORMATION_SCHEMA
- PERFORMANCE_SCHEMA
- METRICS_SCHEMA
- mysql

4.8.2.72.5 和 MySQL 的兼容性

获取表锁

- 在 TiDB 中，如果会话A 已经持有了一个表的表锁，另一个会话B 对该表写入时会报错；但 MySQL 会阻塞会话B 对该表的写入，直到会话A 释放该表锁。其他会话对该表的锁请求会被阻塞直到当前会话释放 WRITE 锁。
- 在 TiDB 中，如果 LOCK TABLES 语句想要获取的表锁被其他会话持有且必须等待锁释放时，LOCK TABLES 语句会执行报错；但 MySQL 会阻塞 LOCK TABLES 语句的执行，直到成功获取想要的表锁。
- 在 TiDB 中，使用 LOCK TABLES 语句获取表锁的作用域是整个集群；但 MySQL 中表锁的作用域是单个 MySQL 服务器，与 NDB 群集不兼容。

释放表锁

在TiDB 的会话中显示开启一个事务时（例如使用 BEGIN 语句），TiDB 不会隐式释放当前会话已经持有的表锁；但 MySQL 会隐式释放当前会话已经持有的表锁。

4.8.2.73 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。

从 v5.1.0 版本起，TiDB 开始支持Reorg 类型变更，包括但不限于：

- 从 VARCHAR 转换为 BIGINT
- DECIMAL 精度修改
- 从 VARCHAR(10) 到 VARCHAR(5) 的长度压缩

4.8.2.73.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName ModifyColumnSpec ( ',' ModifyColumnSpec )*

ModifyColumnSpec
         ::= 'MODIFY' ColumnKeywordOpt 'IF EXISTS' ColumnName ColumnType ColumnOption* ( 'FIRST'
             <⟶ | 'AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
```

```
            | 'UNIQUE' 'KEY'?
            | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
            | 'SERIAL' 'DEFAULT' 'VALUE'
            | 'ON' 'UPDATE' NowSymOptionFraction
            | 'COMMENT' stringLit
            | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
                <-> NULL' ) )?
            | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
            | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
                <-> OnDeleteUpdateOpt
            | 'COLLATE' CollationName
            | 'COLUMN_FORMAT' ColumnFormat
            | 'STORAGE' StorageMedia
            | 'AUTO_RANDOM' ( '(' LengthNum ')' )?

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

4.8.2.73.2 示例

Meta-Only Change

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

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

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

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

```
ALTER TABLE t1 MODIFY col1 BIGINT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

Reorg-Data Change

```
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 (12345),(67890);
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 MODIFY col1 VARCHAR(5);
```

```
Query OK, 0 rows affected (2.52 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` varchar(5) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

注意：

- 当所变更的类型与已经存在的数据行产生冲突时，TiDB 会进行报错处理。在上述例子中，TiDB 将进行如下报错：

```
alter table t1 modify column col1 varchar(4); ERROR 1406 (22001): Data Too Long,
<-> field len 4, data len 5
```

- 由于和Async Commit 功能兼容，DDL 在开始进入到Reorg Data 前会有一定时间（约 2.5s）的等待处理：

```
Query OK, 0 rows affected (2.52 sec)
```

4.8.2.73.3 MySQL 兼容性

- 不支持修改主键列上需要Reorg-Data 的类型，但是支持修改 Meta-Only 的类型。例如：

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a VARCHAR(10);
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a INT(10) UNSIGNED;
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a bigint;
Query OK, 0 rows affected (0.01 sec)
```

- 不支持修改生成列的类型。例如：

```
CREATE TABLE t (a INT, b INT as (a+1));
ALTER TABLE t MODIFY COLUMN b VARCHAR(10);
ERROR 8200 (HY000): Unsupported modify column: column is generated
```

- 不支持修改分区表上的列类型。例如：

```
CREATE TABLE t (c1 INT, c2 INT, c3 INT) partition by range columns(c1) ( partition p0 values
    <-> less than (10), partition p1 values less than (maxvalue));
ALTER TABLE t MODIFY COLUMN c1 DATETIME;
ERROR 8200 (HY000): Unsupported modify column: table is partition table
```

- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为TiDB cast 函数与 MySQL 的行为有一些兼容性问题。例如：

```
CREATE TABLE t (a DECIMAL(13, 7));
ALTER TABLE t MODIFY COLUMN a DATETIME;
ERROR 8200 (HY000): Unsupported modify column: change from original type decimal(13,7) to
    <-> datetime is currently unsupported yet
```

4.8.2.73.4 另请参阅

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

4.8.2.74 PREPARE

PREPARE 语句为服务器端预处理语句提供 SQL 接口。

4.8.2.74.1 语法图

```
PreparedStmt ::=
    'PREPARE' Identifier 'FROM' PrepareSQL

PrepareSQL ::=
    stringLit
|   UserVariable
```

4.8.2.74.2 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)
```

4.8.2.74.3 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。

4.8.2.74.4 另请参阅

- EXECUTE
- DEALLOCATE

4.8.2.75 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。

4.8.2.75.1 语法

```
RECOVER TABLE table_name
```

```
RECOVER TABLE BY JOB JOB_ID
```

语法图

```
RecoverTableStmt ::=
    'RECOVER' 'TABLE' ( 'BY' 'JOB' Int64Num | TableName Int64Num? )

TableName ::=
    Identifier ( '.' Identifier )?

Int64Num ::= NUM

NUM ::= intLit
```

4.8.2.75.2 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。

4.8.2.75.3 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。

4.8.2.75.4 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。

4.8.2.75.5 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.76 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。

4.8.2.76.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName RenameIndexSpec ( ',' RenameIndexSpec )*

RenameIndexSpec
         ::= 'RENAME' ( 'KEY' | 'INDEX' ) Identifier 'TO' Identifier
```

4.8.2.76.2 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)
```

4.8.2.76.3 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。

4.8.2.76.4 另请参阅

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

4.8.2.77 RENAME TABLE

`RENAME TABLE` 语句用于对已有表进行重命名。

4.8.2.77.1 语法图

```
RenameTableStmt ::=
    'RENAME' 'TABLE' TableToTable ( ',' TableToTable )*

TableToTable ::=
    TableName 'TO' TableName
```

4.8.2.77.2 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)
```

4.8.2.77.3 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。

4.8.2.77.4 另请参阅

- CREATE TABLE
- SHOW TABLES
- ALTER TABLE

4.8.2.78 RENAME USER

RENAME USER 语句用于对已有用户进行重命名。

4.8.2.78.1 语法图

```
RenameUserStmt ::=
    'RENAME' 'USER' UserToUser ( ',' UserToUser )*

UserToUser ::=
    Username 'TO' Username

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces
```

4.8.2.78.2 示例

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

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

```
SHOW GRANTS FOR 'newuser';
```

```
+--------------------------------------+
| Grants for newuser@%                 |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)
```

```
RENAME USER 'newuser' TO 'testuser';
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SHOW GRANTS FOR 'testuser';
```

```
+--------------------------------------+
| Grants for testuser@%                |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'testuser'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

4.8.2.78.3 MySQL 兼容性

RENAME USER 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。

4.8.2.78.4 另请参阅

- CREATE USER
- SHOW GRANTS
- DROP USER

4.8.2.79 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。

4.8.2.79.1 语法图

```
ReplaceIntoStmt ::=
    'REPLACE' PriorityOpt IntoOpt TableName PartitionNameListOpt InsertValues

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IntoOpt ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?

InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        <⟶ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*
```

4.8.2.79.2 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)
```

4.8.2.79.3 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。

4.8.2.79.4 另请参阅

- DELETE
- INSERT
- SELECT
- UPDATE

4.8.2.80 RESTORE

RESTORE 语句用于执行分布式恢复，把BACKUP 语句生成的备份文件恢复到TiDB 集群中。

RESTORE 语句使用的引擎与 BR 相同，但恢复过程是由TiDB 本身驱动，而非单独的 BR 工具。BR 工具的优势和警告也适用于 RESTORE 语句。需要注意的是，RESTORE 语句目前不遵循ACID 原则。

执行 RESTORE 语句前，确保集群已满足以下要求：

- 集群处于“下线”状态，当前的TiDB 会话是唯一在访问待恢复表的活跃 SQL 连接。
- 执行全量恢复时，确保即将恢复的表不存在于集群中，因为现有的数据可能被覆盖，从而导致数据与索引不一致。
- 执行增量恢复时，表的状态应该与创建备份时 LAST_BACKUP 时间戳的状态完全一致。

执行 RESTORE 需要 RESTORE_ADMIN 或 SUPER 权限。此外，执行恢复操作的 TiDB 节点和集群中的所有TiKV 节点都必须有对目标存储的读权限。

RESTORE 语句开始执行后将会被阻塞，直到整个恢复任务完成、失败或取消。因此，执行 RESTORE 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和 RESTORE 任务。如果TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 RESTORE 将等待前面所有的任务完成后再执行。

RESTORE 只能在“tikv”存储引擎上使用，如果使用“unistore”存储引擎，RESTORE 操作会失败。

4.8.2.80.1 语法图

```
RestoreStmt ::=
    "RESTORE" BRIETables "FROM" stringLit RestoreOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

RestoreOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean

Boolean ::=
    NUM | "TRUE" | "FALSE"
```

4.8.2.80.2 示例

从备份文件恢复

```
RESTORE DATABASE * FROM 'local:///mnt/backup/2020/04/';
```

```
+--
    <--> ---------------------------+-----------+----------+--------------------+---------------------------------- +
    <-->
| Destination                  | Size       | BackupTS | Queue Time           | Execution Time
    <-->     |
+--
    <--> ---------------------------+-----------+----------+--------------------+---------------------------------- +
    <-->
| local:///mnt/backup/2020/04/ | 248665063 | 0         | 2020-04-21 17:16:55 | 2020-04-21 17:16:55
    <--> |
+--
    <--> ---------------------------+-----------+----------+--------------------+---------------------------------- +
    <-->
1 row in set (28.961 sec)
```

上述示例中，所有数据均从本地的备份文件中恢复到集群中。RESTORE 从 SST 文件里读取数据，SST 文件存储在所有TiDB 和 TiKV 节点的 /mnt/backup/2020/04/ 目录下。

输出结果的第一行描述如下：

列名	描述
Destination	读取的目标存储 URL
Size	备份文件的总大小，单位为字节
BackupTS	不适用
Queue Time	RESTORE 任务开始排队的时间戳（当前时区）
Execution Time	RESTORE 任务开始执行的时间戳（当前时区）

部分恢复

你可以指定恢复部分数据库或部分表数据。如果备份文件中缺失了某些数据库或表，缺失的部分将被忽略。此时，RESTORE 语句不进行任何操作即完成执行。

```
RESTORE DATABASE `test` FROM 'local:///mnt/backup/2020/04/';
```

```
RESTORE TABLE `test`.`sbtest01`, `test`.`sbtest02` FROM 'local:///mnt/backup/2020/04/';
```

外部存储

BR 支持从 Amazon S3 或 Google Cloud Storage (GCS) 恢复数据：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/';
```

有关详细的 URL 语法，见外部存储 URI 格式。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个TiKV 节点的平均下载速度。

默认情况下，每个TiKV 节点上运行 128 个恢复线程。可以通过 CONCURRENCY 选项来调整这个值。

在恢复完成之前，RESTORE 将对备份文件中的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 64
    CHECKSUM = FALSE;
```

增量恢复

增量恢复没有特殊的语法。TiDB 将识别备份文件属于全量备份或增量备份，然后执行对应的恢复操作，用户只需按照正确顺序进行增量恢复。

假设按照如下方式创建一个备份任务：

```
BACKUP DATABASE `test` TO 's3://example-bucket/full-backup'  SNAPSHOT = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-1' SNAPSHOT = 414971854848000
    <→ LAST_BACKUP = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-2' SNAPSHOT = 416353458585600
    <→ LAST_BACKUP = 414971854848000;
```

在恢复备份时，需要采取同样的顺序：

```
RESTORE DATABASE * FROM 's3://example-bucket/full-backup';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-1';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-2';
```

4.8.2.80.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.80.4 另请参阅

- BACKUP
- SHOW RESTORES

4.8.2.81 REVOKE <privileges>

`REVOKE <privileges>` 语句用于删除已有用户的权限。执行 `REVOKE <privileges>` 语句需要拥有分配的权限，并且拥有 `GRANT OPTION` 权限。

4.8.2.81.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        <→ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
     'ALL' 'PRIVILEGES'?
|    'ALTER' 'ROUTINE'?
|    'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|     'TRIGGER'
|    'DELETE'
|     'DROP' 'ROLE'?
```

```
|    'PROCESS'
|    'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|    'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|   'GRANT' 'OPTION'
|   'REFERENCES'
|   'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|   'USAGE'
|    'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|    'EVENT'
|   'SHUTDOWN'

ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

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

4.8.2.81.2 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 '%'
```

4.8.2.81.3 MySQL 兼容性

`REVOKE <privileges>` 语句执行成功后，在TiDB 中语句执行的结果会在当前连接立即生效，而 MySQL 中部分权限的结果需要等到之后的连接才生效。见 TiDB #39356。

4.8.2.81.4 另请参阅

- GRANT <privileges>
- SHOW GRANTS
- Privilege Management

4.8.2.82 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。

4.8.2.82.1 语法图

```
RevokeRoleStmt ::=
    'REVOKE' RolenameList 'FROM' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

4.8.2.82.2 示例

以 root 用户连接TiDB：

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

创建新角色 analyticsteam 和新用户 jennifer：

```
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)
```

以 jennifer 用户连接TiDB：

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

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：

```
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)
```

以 root 用户连接TiDB：

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

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接TiDB：

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

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
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)
```

以 root 用户连接TiDB：

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

收回角色 analyticsteam：

```
REVOKE analyticsteam FROM jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 jennifer 用户连接TiDB：

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

查看 jennifer 用户的权限：

```
SHOW GRANTS;
+---------------------------------------+
| Grants for User                       |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+---------------------------------------+
1 row in set (0.00 sec)
```

4.8.2.82.3 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。

4.8.2.82.4 另请参阅

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

4.8.2.83 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。

4.8.2.83.1 语法图

```
RollbackStmt ::=
    'ROLLBACK' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

4.8.2.83.2 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)
```

4.8.2.83.3 MySQL 兼容性

- TiDB 解析但忽略 `ROLLBACK AND [NO] RELEASE` 语法。在 MySQL 中，使用该语法可在回滚事务后立即断开客户端会话。在TiDB 中，建议使用客户端程序的 `mysql_close()` 来实现该功能。
- TiDB 解析但忽略 `ROLLBACK AND [NO] CHAIN` 语法。在 MySQL 中，使用该语法可在回滚当前事务时立即以相同的隔离级别开启新事务。在TiDB 中，推荐直接开启新事务。

4.8.2.83.4 另请参阅

- SAVEPOINT
- COMMIT
- BEGIN
- START TRANSACTION

4.8.2.84 SAVEPOINT

SAVEPOINT 是TiDB 从 v6.2.0 开始支持的特性，语法如下：

```
SAVEPOINT identifier
ROLLBACK TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

> **警告：**
>
> SAVEPOINT 特性不支持与 TiDB Binlog 一起使用，也不支持在关闭tidb_constraint_check_in_place_pessimistic 的悲观事务中使用。

- SAVEPOINT 语句用于在当前事务中，设置一个指定名字保存点。如果已经存在相同名字的保存点，就删除已有的保存点并设置新的保存点。
- ROLLBACK TO SAVEPOINT 语句将事务回滚到指定名称的事务保存点，而不终止该事务。当前事务在设置保存点后，对表数据所做的修改将在回滚中撤销，且删除事务保存点之后的所有保存点。在悲观事务中，对于已经持有的悲观锁不会回滚，而是在事务结束时才释放。

  如果 ROLLBACK TO SAVEPOINT 语句中指定名称的保存点不存在，则会返回以下错误信息：

  ```
  ERROR 1305 (42000): SAVEPOINT identifier does not exist
  ```

- RELEASE SAVEPOINT 语句将从当前事务中删除指定名称及之后的所有保存点，而不会提交或回滚当前事务。如果指定名称的保存点不存在，则会返回以下错误信息：

  ```
  ERROR 1305 (42000): SAVEPOINT identifier does not exist
  ```

  当事务提交或者回滚后，事务中所有保存点都会被删除。

4.8.2.84.1 示例

创建表 t1：

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

开始当前事务：

```
BEGIN;
```

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

向表中插入数据并设置保存点 sp1：

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SAVEPOINT sp1;
```

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

向表中再次插入数据并设置保存点 sp2：

```
INSERT INTO t1 VALUES (2);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SAVEPOINT sp2;
```

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

释放保存点 sp2：

```
RELEASE SAVEPOINT sp2;
```

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

回滚至保存点 sp1：

```
ROLLBACK TO SAVEPOINT sp1;
```

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

提交事务并查询表格，发现表中仅有 sp1 前插入的数据：

```
COMMIT;
```

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

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
+---+
1 row in set
```

4.8.2.84.2 MySQL 兼容性

使用 ROLLBACK TO SAVEPOINT 语句将事物回滚到指定保存点时，MySQL 会释放该保存点之后才持有的锁，但在 TiDB 悲观事务中，不会立即释放该保存点之后才持有的锁，而是等到事务提交或者回滚时，才释放全部持有的锁。

4.8.2.84.3 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION
- TiDB 乐观事务模型
- TiDB 悲观事务模式

4.8.2.85 SELECT

SELECT 语句用于从TiDB 读取数据。

4.8.2.85.1 语法图

SelectStmt:

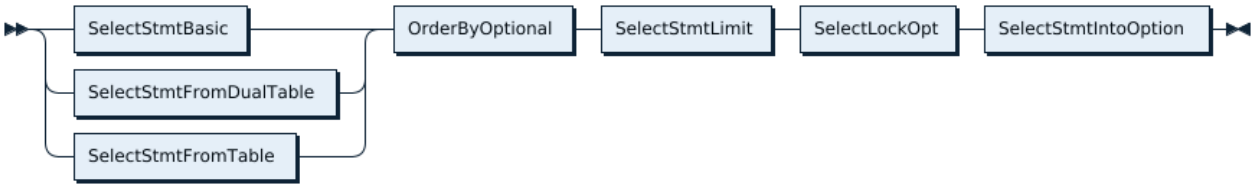


图 82: SelectStmt

FromDual:

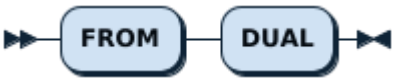


图 83: FromDual

WhereClauseOptional:

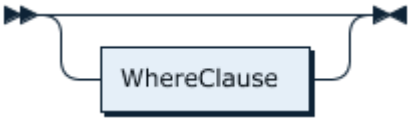


图 84: WhereClauseOptional

SelectStmtOpts:

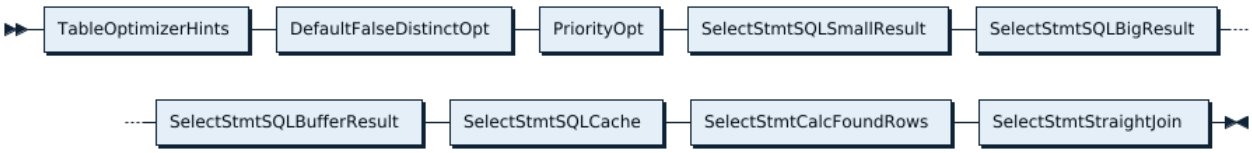


图 85: SelectStmtOpts

SelectStmtFieldList:

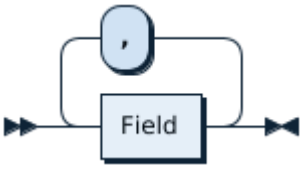


图 86: SelectStmtFieldList

TableRefsClause:

```
TableRefsClause ::=
    TableRef AsOfClause? ( ',' TableRef AsOfClause? )*

AsOfClause ::=
    'AS' 'OF' 'TIMESTAMP' Expression
```

WhereClauseOptional:

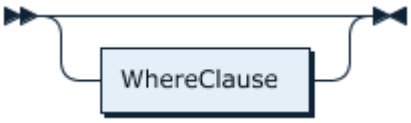


图 87: WhereClauseOptional

SelectStmtGroup:



图 88: SelectStmtGroup

HavingClause:

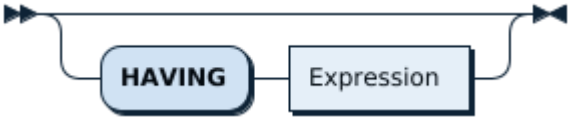


图 89: HavingClause

OrderByOptional:



图 90: OrderByOptional

SelectStmtLimit:

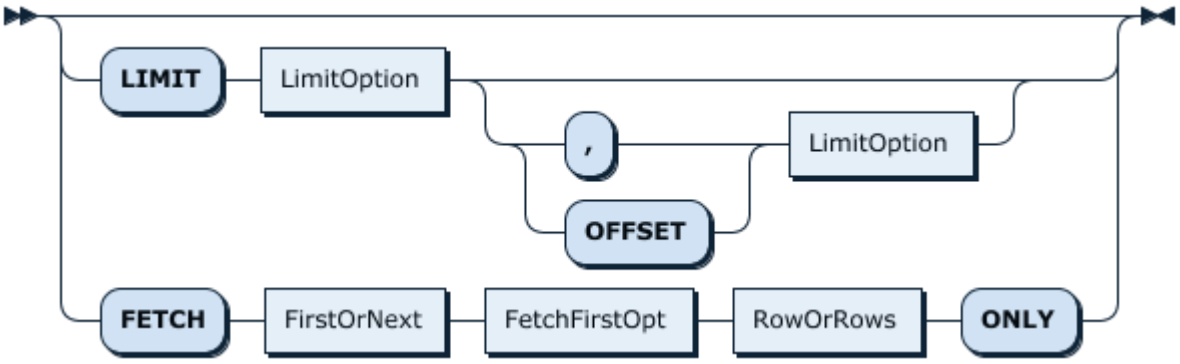


图 91: SelectStmtLimit

FirstOrNext:

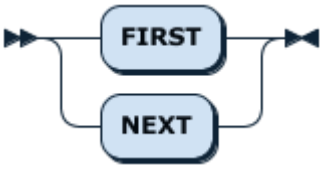


图 92: FirstOrNext

FetchFirstOpt:



图 93: FetchFirstOpt

RowOrRows:

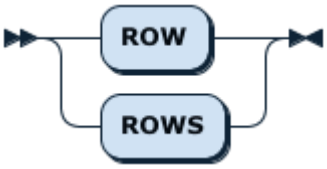


图 94: RowOrRows

SelectLockOpt:

```
SelectLockOpt ::=
    ( ( 'FOR' 'UPDATE' ( 'OF' TableList )? 'NOWAIT'? )
|   ( 'LOCK' 'IN' 'SHARE' 'MODE' ) )?

TableList ::=
    TableName ( ',' TableName )*
```

WindowClauseOptional

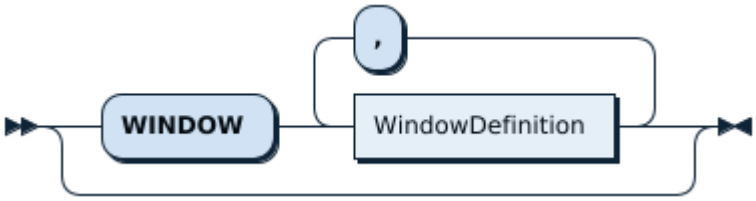


图 95: WindowClauseOptional

4.8.2.85.2 语法元素说明

语法元素	说明
TableOptimizerHints	用于控制优化器行为的 Hint，具体可参见 Optimizer Hints
ALL、DISTINCT、DISTINCTROW	查询结果集中可能会包含重复值。指定 DISTINCT/DISTINCTROW 则在查询结果中过滤掉重复的行；指定ALL 则列出所有的行。默认为 ALL。
HIGH_PRIORITY	该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
SQL_CALC_FOUND_ROWS	TiDB 不支持该语法，并报错（若tidb_enable_noop_functions 值设为 1 则不会报错）
SQL_CACHE、SQL_NO_CACHE	是否把请求结果缓存到TiKV (RocksDB) 的 BlockCache 中。对于一次性的大数据量的查询，比如 count(*) 查询，为了避免冲掉 BlockCache 中用户的热点数据，建议填上 SQL_NO_CACHE
STRAIGHT_JOIN	STRAIGHT_JOIN 会强制优化器按照 FROM 子句中所使用的表的顺序做联合查询。当优化器选择的Join 顺序并不优秀时，你可以使用这个语法来加速查询的执行
select_expr	投影操作列表，一般包括列名、表达式，或者是用 ‘*’ 表示全部列
FROM table_references	表示数据来源，数据来源可以是一个表(select * from t;)或者是多个表(select * from t1 join t2;)或者是 0 个表（select 1+1 from dual;，等价于 select 1+1;）
WHERE where_condition	Where 子句用于设置过滤条件，查询结果中只会包含满足条件的数据
GROUP BY	GroupBy 子句用于对查询结果集进行分组
HAVING where_condition	Having 子句与Where 子句作用类似，Having 子句可以让过滤 GroupBy 后的各种数据，Where 子句用于在聚合前过滤记录。

语法元素	说明
ORDER BY	OrderBy 子句用于指定结果排序顺序，可以按照列、表达式或者是 select_expr 列表中某个位置的字段进行排序。
LIMIT	Limit 子句用于限制结果条数。Limit 接受一个或两个数字参数，如果只有一个参数，那么表示返回数据的最大行数；如果是两个参数，那么第一个参数表示返回数据的第一行的偏移量（第一行数据的偏移量是 0），第二个参数指定返回数据的最大条目数。另支持 FETCH FIRST/NEXT n ROW/ROWS ONLY 语法，与 LIMIT n 效果相同，其中n 可省略，省略时与 LIMIT 1 效果相同。
Window window_definition	窗口函数的相关语法，用来进行一些分析型计算的操作，详情可见窗口函数
FOR UPDATE	对查询结果集所有行上锁（对于在查询条件内，但是不在结果集的行，将不会加锁，如事务启动后由其他事务写入的行），以监测其他事务对这些的并发修改。使用乐观事务模型时，语句执行期间不会检测锁，因此，不会像 PostgreSQL 之类的数据库一样，在当前事务结束前阻止其他事务执行 UPDATE、DELETE 和 SELECT FOR UPDATE。在事务的提交阶段 SELECT FOR UPDATE 读到的行，也会进行两阶段提交，因此，它们也可以参与事务冲突检测。如发生写入冲突，那么包含 SELECT FOR UPDATE 语句的事务会提交失败。如果没有冲突，事务将成功提交，当提交结束时，这些被加锁的行，会产生一个新版本，可以让其他尚未提交的事务，在将来提交时发现写入冲突。若使用悲观事务，则行为与其他数据库基本相同，不一致之处参考和 MySQL InnoDB 的差异。TiDB 支持 FOR UPDATE NOWAIT 语法，详情可见TiDB 中悲观事务模式的行为。
LOCK IN SHARE MODE	TiDB 出于兼容性解析这个语法，但是不做任何处理

4.8.2.85.3 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)
```

4.8.2.85.4 MySQL 兼容性

- 不支持 `SELECT ... INTO @variable` 语法。
- 不支持 `SELECT ... GROUP BY ... WITH ROLLUP` 语法。
- 不支持 MySQL 5.7 中支持的 `SELECT .. GROUP BY expr` 语法，而是匹配 MySQL 8.0 的行为，不按照默认的顺序进行排序。

4.8.2.85.5 另请参阅

- INSERT
- DELETE
- UPDATE
- REPLACE

4.8.2.86 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。

4.8.2.86.1 语法图

SetDefaultRoleStmt:



图 96: SetDefaultRoleStmt

SetDefaultRoleOpt:

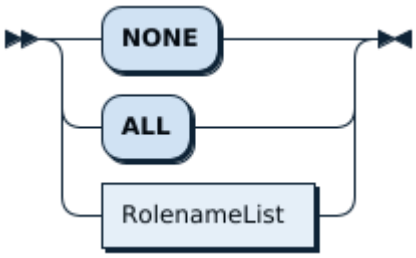


图 97: SetDefaultRoleOpt

RolenameList:

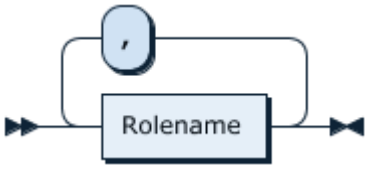


图 98: RolenameList

UsernameList:

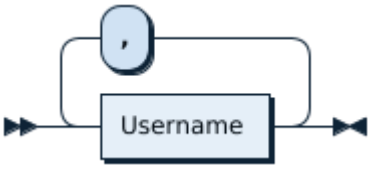


图 99: UsernameList

4.8.2.86.2 示例

以 root 用户连接TiDB：

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

创建新角色 analyticsteam 和新用户 jennifer：

```
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)
```

以 jennifer 用户连接TiDB：

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

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analyticsteam 角色相关联的权限：

```
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)
```

以 root 用户连接TiDB：

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

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接TiDB：

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

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
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@%%
```

4.8.2.86.3 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。

4.8.2.86.4 另请参阅

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

4.8.2.87 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。

4.8.2.87.1 语法图

SetNamesStmt:



图 100: SetNamesStmt

VariableAssignmentList:

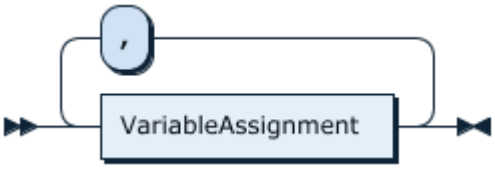


图 101: VariableAssignmentList

VariableAssignment:

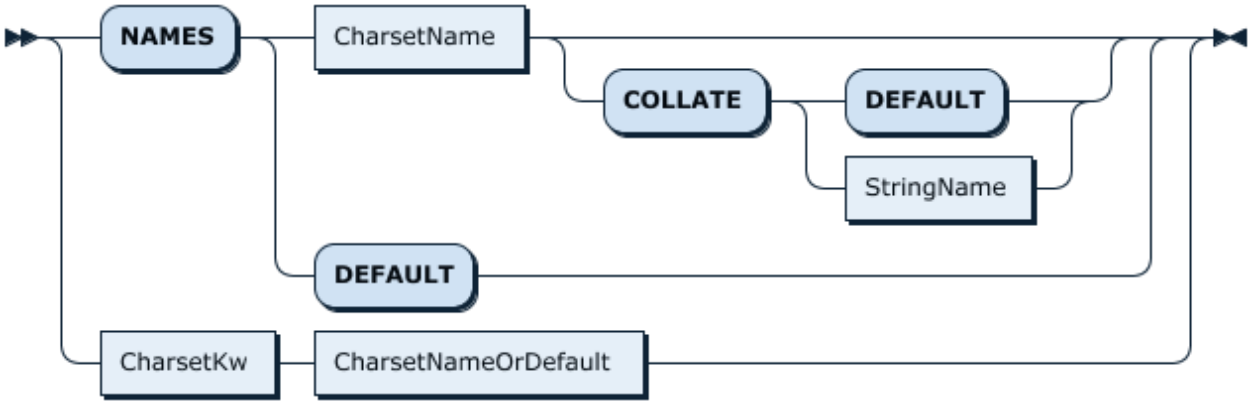


图 102: VariableAssignment

CharsetName:



图 103: CharsetName

StringName:



图 104: StringName

CharsetKw:

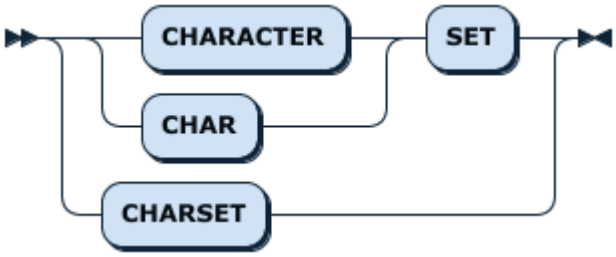


图 105: CharsetKw

CharsetNameOrDefault:

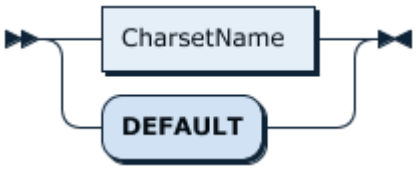


图 106: CharsetNameOrDefault

4.8.2.87.2 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)
```

4.8.2.87.3 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。

4.8.2.87.4 另请参阅

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

4.8.2.88 SET PASSWORD

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

4.8.2.88.1 语法图

SetStmt:

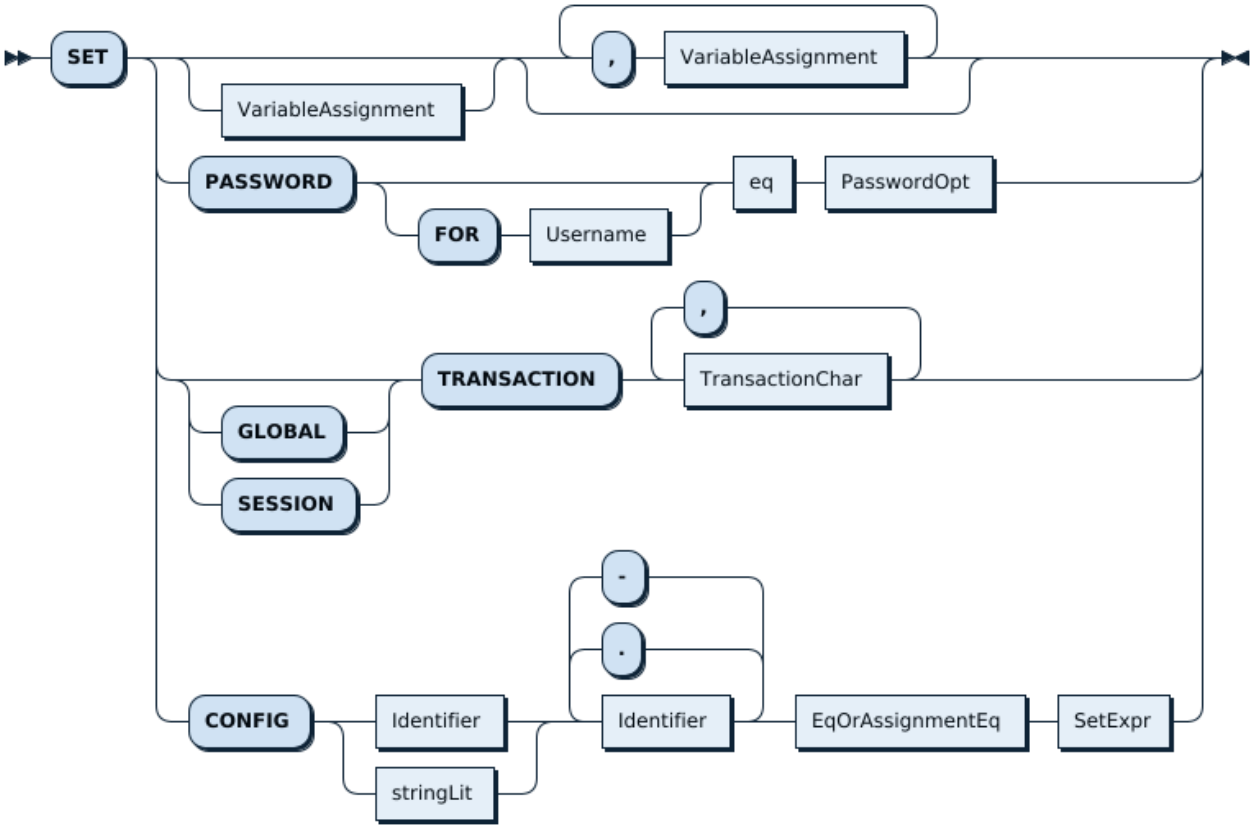


图 107: SetStmt

4.8.2.88.2 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)
```

4.8.2.88.3 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。

4.8.2.88.4 另请参阅

- CREATE USER
- Privilege Management

4.8.2.89 SET RESOURCE GROUP

`SET RESOURCE GROUP` 用于设置当前会话使用的资源组。

4.8.2.89.1 语法图

SetResourceGroupStmt:

```
SetResourceGroupStmt ::=
   "SET" "RESOURCE" "GROUP" ResourceGroupName

ResourceGroupName ::=
   Identifier
```

4.8.2.89.2 示例

创建一个用户 `user1`，创建两个资源组 `rg1` 和 `rg2`，并将用户 `user1` 绑定资源组 `rg1`。

```
CREATE USER 'user1';
CREATE RESOURCE GROUP 'rg1' RU_PER_SEC = 1000;
ALTER USER 'user1' RESOURCE GROUP `rg1`;
```

使用 `user1` 登录，查看当前用户绑定的资源组。

```
SELECT CURRENT_RESOURCE_GROUP();
```

```
+---------------------------+
| CURRENT_RESOURCE_GROUP() |
+---------------------------+
| rg1                       |
```

```
+--------------------------+
1 row in set (0.00 sec)
```

执行 SET RESOURCE GROUP 将当前会话的资源组设置为 rg2。

```
SET  RESOURCE  GROUP  `rg2`;
SELECT CURRENT_RESOURCE_GROUP();
```

```
+--------------------------+
| CURRENT_RESOURCE_GROUP() |
+--------------------------+
| rg2                      |
+--------------------------+
1 row in set (0.00 sec)
```

执行 SET RESOURCE GROUP 设置当前会话使用默认资源组。

```
SET RESOURCE GROUP ``;
SELECT CURRENT_RESOURCE_GROUP();
```

```
+--------------------------+
| CURRENT_RESOURCE_GROUP() |
+--------------------------+
| default                  |
+--------------------------+
1 row in set (0.00 sec)
```

4.8.2.89.3 MySQL 兼容性

MySQL 也支持 SET RESOURCE GROUP，但是接受的参数和TiDB 不同，两者并不兼容。

4.8.2.89.4 另请参阅

- CREATE RESOURCE GROUP
- DROP RESOURCE GROUP
- ALTER RESOURCE GROUP
- 使用资源管控(Resource Control) 实现资源隔离

4.8.2.90 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。

4.8.2.90.1 语法图

SetRoleStmt:



图 108: SetRoleStmt

SetRoleOpt:

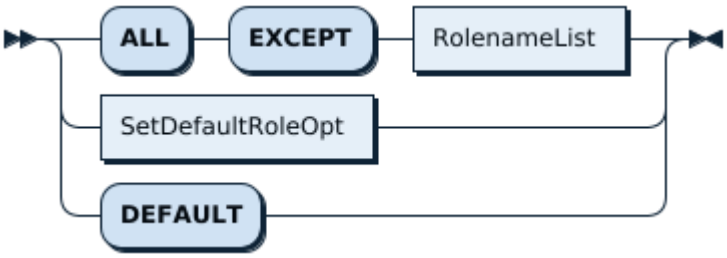


图 109: SetRoleOpt

SetDefaultRoleOpt:

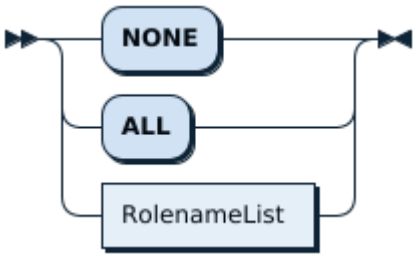


图 110: SetDefaultRoleOpt

4.8.2.90.2 示例

创建一个用户 `'u1'@'%'`，创建三个角色 `'r1'@'%'`、`'r2'@'%'` 和 `'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)
```

4.8.2.90.3 MySQL 兼容性

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

4.8.2.90.4 另请参阅

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

4.8.2.91 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。

4.8.2.91.1 语法图

```
SetStmt ::=
    'SET' ( VariableAssignmentList |
    'PASSWORD' ('FOR' Username)? '=' PasswordOpt |
    ( 'GLOBAL'| 'SESSION' )? 'TRANSACTION' TransactionChars |
    'CONFIG' ( Identifier | stringLit) ConfigItemName EqOrAssignmentEq SetExpr )

TransactionChars ::=
    ( 'ISOLATION' 'LEVEL' IsolationLevel | 'READ' 'WRITE' | 'READ' 'ONLY' AsOfClause? )

IsolationLevel ::=
    ( 'REPEATABLE' 'READ' | 'READ' ( 'COMMITTED' | 'UNCOMMITTED' ) | 'SERIALIZABLE' )

AsOfClause ::=
    ( 'AS' 'OF' 'TIMESTAMP' Expression)
```

4.8.2.91.2 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)
```

4.8.2.91.3 MySQL 兼容性

- TiDB 支持仅在语法中将事务设置为只读的功能。
- 不支持隔离级别 READ-UNCOMMITTED 和 SERIALIZABLE。
- 通过快照隔离(Snapshot Isolation)技术，实现乐观事务的 REPEATABLE-READ 隔离级别，和 MySQL 兼容。
- 在悲观事务中，TiDB 支持与 MySQL 兼容的 REPEATABLE-READ 和 READ-COMMITTED 两种隔离级别。具体描述详见Isolation Levels。

4.8.2.91.4 另请参阅

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

4.8.2.92 SET [GLOBAL|SESSION] <variable>

SET [GLOBAL|SESSION] 语句用于在 SESSION 或 GLOBAL 的范围内，对某个TiDB 的内置变量进行更改。

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

4.8.2.92.1 语法图

SetStmt:

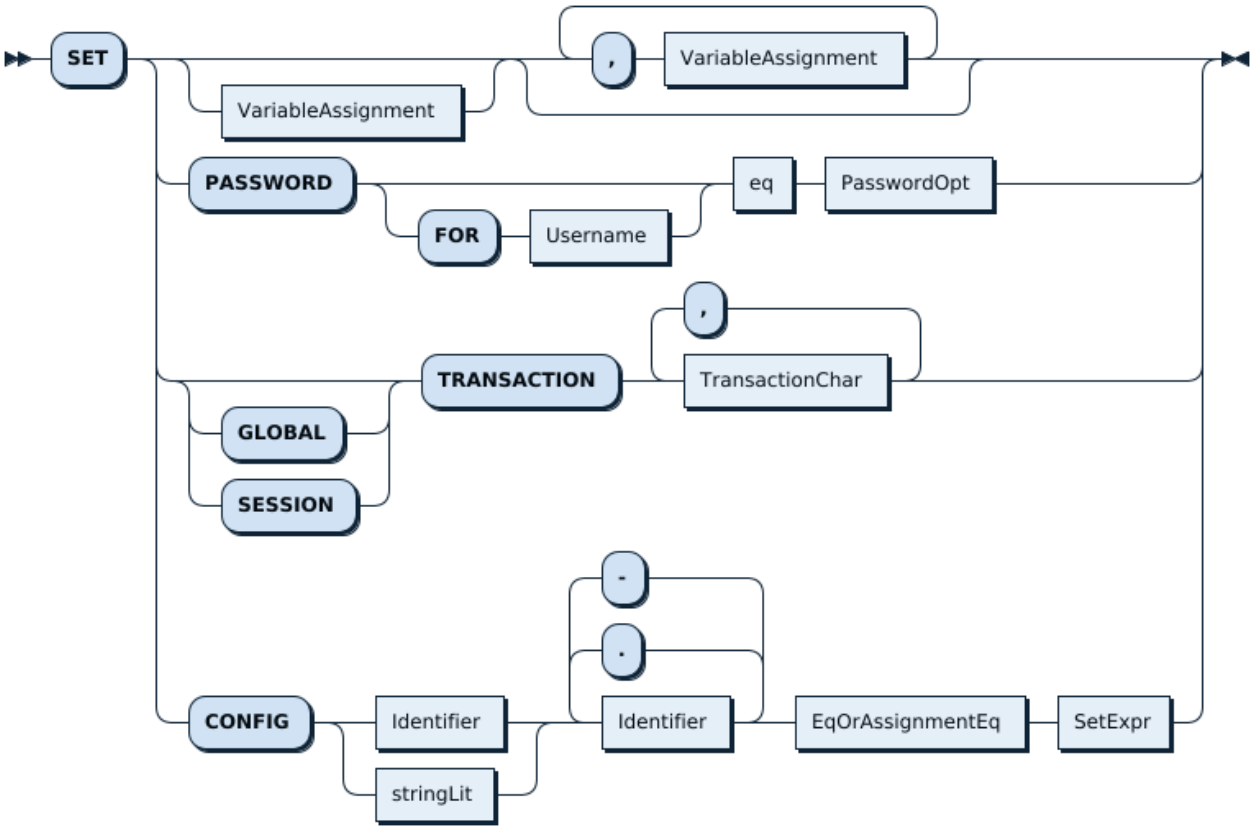


图 111: SetStmt

VariableAssignment:

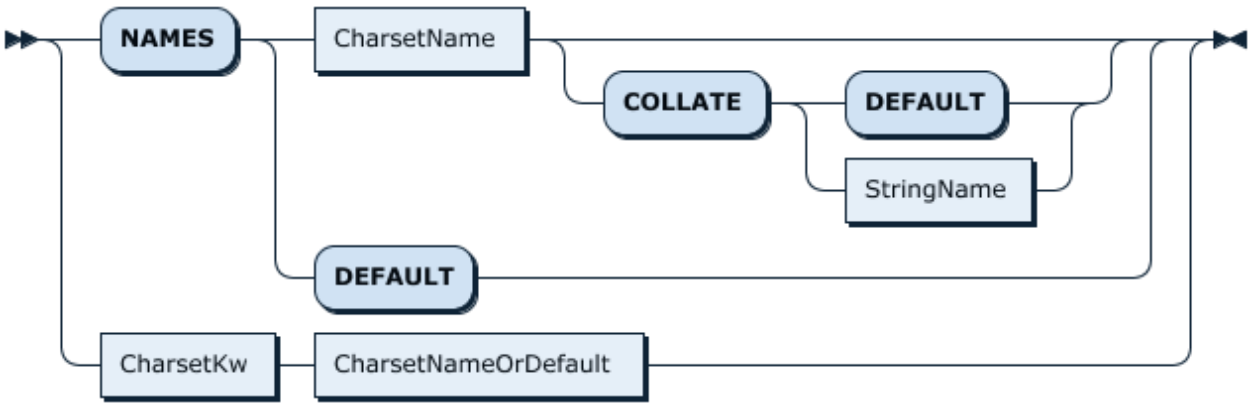


图 112: VariableAssignment

4.8.2.92.2 示例

获取 sql_mode 的值：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------
```

```
1 row in set (0.00 sec)
```

更新全局的 sql_mode：

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

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

检查更新之后的 sql_mode 的取值，可以看到 SESSION 级别的值没有更新：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+-----------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-------------------------------------
```

4.8.2.92.3 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。

4.8.2.92.4 另请参阅

- SHOW [GLOBAL|SESSION] VARIABLES

4.8.2.93 SHOW [BACKUPS|RESTORES]

SHOW [BACKUPS|RESTORES] 语句会列出所有队列中或正在执行的BACKUP 和RESTORE 任务。

查询 BACKUP 任务时，使用 SHOW BACKUPS 语句。查询 RESTORE 任务时，使用 SHOW RESTORES 语句。执行两个语句均需要 SUPER 权限。

4.8.2.93.1 语法图

```
ShowBRIEStmt ::=
    "SHOW" ("BACKUPS" | "RESTORES") ShowLikeOrWhere?

ShowLikeOrWhere ::=
    "LIKE" SimpleExpr
|   "WHERE" Expression
```

4.8.2.93.2 示例

在一个连接中，执行以下命令备份数据库：

```
BACKUP DATABASE `test` TO 's3://example-bucket/backup-01/';
```

在备份完成之前，在新的连接中执行 SHOW BACKUPS：

```
SHOW BACKUPS;
```

```
+--
    <-> ----------------------------+---------+----------+--------------------+--------------------+---------
    <->
| Destination                    | State    | Progress | Queue_Time          | Execution_Time
    <->     | Finish_Time | Connection |
```

```
+--
    <-> ----------------------------+---------+----------+--------------------+--------------------+---------
    <->
| s3://example-bucket/backup-01/ | Backup  | 98.38    | 2020-04-12 23:09:03 | 2020-04-12 23:09:25
    <-> |        NULL |          4 |
+--
    <-> ----------------------------+---------+----------+--------------------+--------------------+---------
    <->
1 row in set (0.00 sec)
```

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL（为避免泄露密钥，所有参数均不显示）
State	任务状态
Progress	当前状态的进度（百分比）
Queue Time	任务开始排队的时间
Execution Time	任务开始执行的时间；对于队列中任务，该值为 0000-00-00 00:00:00
Finish_Time	（暂不适用）
Connection	运行任务的连接ID

连接ID 可用于在KILL TIDB QUERY 语句中取消备份/恢复任务：

```
KILL TIDB QUERY 4;
```

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

过滤

在 LIKE 子句中使用通配符，可以按目标存储 URL 筛选任务：

```
SHOW BACKUPS LIKE 's3://%';
```

使用 WHERE 子句，可以按列筛选任务：

```
SHOW BACKUPS WHERE `Progress` < 25.0;
```

4.8.2.93.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.93.4 另请参阅

- BACKUP
- RESTORE

4.8.2.94 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。

从 TiDB v6.1.0 起，执行 SHOW ANALYZE STATUS 语句将显示集群级别的任务，且TiDB 重启后仍能看到重启之前的任务记录。在TiDB v6.1.0 之前，执行 SHOW ANALYZE STATUS 语句仅显示实例级别的任务，且TiDB 重启后任务记录会被清空。

从TiDB v6.1.0 起，你可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史记录。

4.8.2.94.1 语法图

```
ShowAnalyzeStatusStmt ::= 'SHOW' 'ANALYZE' 'STATUS' ShowLikeOrWhereOpt

ShowLikeOrWhereOpt ::= 'LIKE' SimpleExpr | 'WHERE' Expression
```

4.8.2.94.2 示例

```
mysql> create table t(x int, index idx(x)) partition by hash(x) partitions 2;
Query OK, 0 rows affected (0.69 sec)

mysql> set @@tidb_analyze_version = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> analyze table t;
Query OK, 0 rows affected (0.20 sec)

mysql> show analyze status;
+--
    <--> ------------+------------+----------------+-------------------+---------------+---------------------+---
    <-->
| Table_schema | Table_name | Partition_name | Job_info           | Processed_rows | Start_time
    <-->          | End_time             | State     | Fail_reason | Instance        | Process_ID |
+--
    <--> ------------+------------+----------------+-------------------+---------------+---------------------+---
    <-->
| test         | t          | p1             | analyze index idx |              0 | 2022-05-27
    <--> 11:29:46 | 2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx |              0 | 2022-05-27
    <--> 11:29:46 | 2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns   |              0 | 2022-05-27
    <--> 11:29:46 | 2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns   |              0 | 2022-05-27
    <--> 11:29:46 | 2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
+--
    <--> ------------+------------+----------------+-------------------+---------------+---------------------+---
    <-->
```

```
4 rows in set (0.01 sec)

mysql> set @@tidb_analyze_version = 2;
Query OK, 0 rows affected (0.00 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.03 sec)

mysql> show analyze status;
+--------------+------------+--------------+--------------------------------------------------------------------------------------------------------------------------------+----------------+---------------------+---------------------+----------+-------------+-----------------+------------+
| Table_schema | Table_name | Partition_name | Job_info                                                                                                                       | Processed_rows | Start_time          | End_time            | State    | Fail_reason | Instance        | Process_ID |
+--------------+------------+--------------+--------------------------------------------------------------------------------------------------------------------------------+----------------+---------------------+---------------------+----------+-------------+-----------------+------------+
| test         | t          | p1           | analyze table all columns with 256 buckets, 500 topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0           | analyze table all columns with 256 buckets, 500 topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1           | analyze index idx                                                                                                              |              0 | 2022-05-27 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0           | analyze index idx                                                                                                              |              0 | 2022-05-27 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1           | analyze columns                                                                                                                |              0 | 2022-05-27 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0           | analyze columns                                                                                                                |              0 | 2022-05-27 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--------------+------------+--------------+--------------------------------------------------------------------------------------------------------------------------------+----------------+---------------------+---------------------+----------+-------------+-----------------+------------+
6 rows in set (0.00 sec)
```

4.8.2.94.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.94.4 另请参阅

- ANALYZE_STATUS 表

4.8.2.95 SHOW [GLOBAL|SESSION] BINDINGS

`SHOW BINDINGS` 语句用于显示创建过的 `SQL` 绑定的相关信息。`BINDING` 语句可以在 `GLOBAL` 或者 `SESSION` 作用域内显示执行计划绑定。在不指定作用域时，默认的作用域为 `SESSION`。

4.8.2.95.1 语法图

ShowStmt:

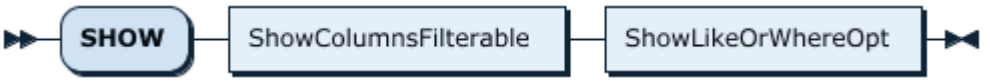


图 113: ShowStmt

ShowTargetFilterable:

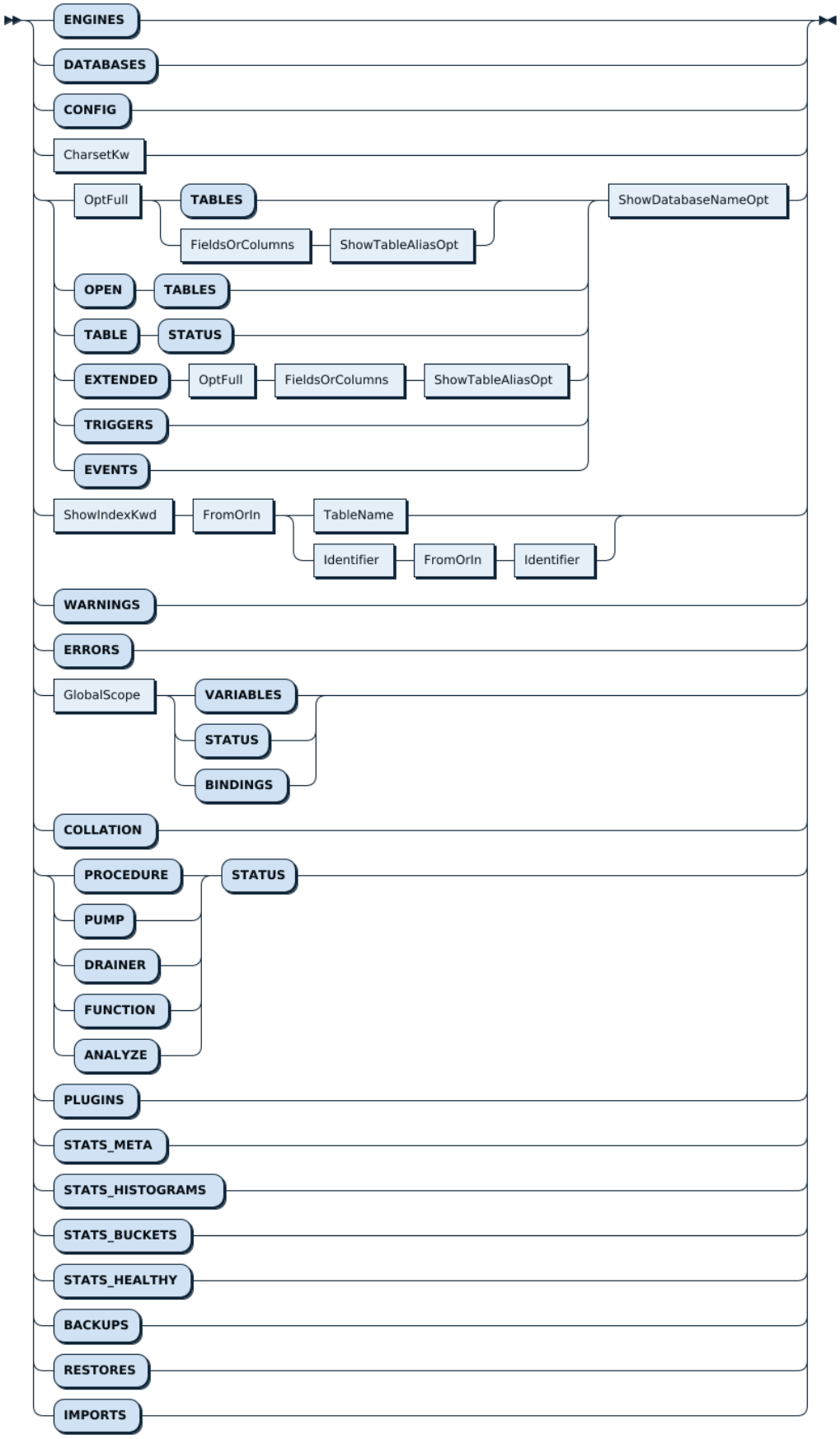


图 114: ShowTargetFilterable

GlobalScope:



图 115: GlobalScope

ShowLikeOrWhereOpt

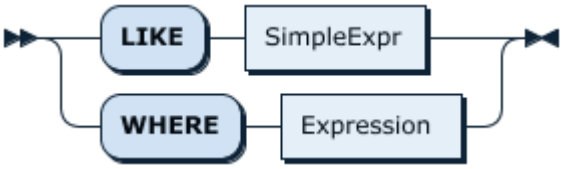


图 116: ShowLikeOrWhereOpt

4.8.2.95.2 语法说明

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhereOpt];
```

该语句会输出GLOBAL 或者 SESSION 作用域内的执行计划绑定，在不指定作用域时默认作用域为 SESSION。目前 SHOW BINDINGS 会输出 8 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL
default_db	默认数据库名
status	状态，包括using（正在使用）、deleted（已删除）、invalid（无效）、rejected（演进时被拒绝）和 pending verify（等待演进验证）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则

列名	说明
source	创建方式，包括 manual（由 create [global] <-> binding 生成）、capture（由 tidb 自动创建生成）和 evolve（由 tidb 自动演进生成）

4.8.2.95.3 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

```
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    -> LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    -> LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    -> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    ->
| id                            | estRows | actRows | task      | access object        |
    -> execution info                                                                | operator info
    ->                       | memory         | disk |
+--
    -> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    ->
| IndexLookUp_10                | 583.00  | 297     | root      |                      | time
    -> :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ->                                    | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    -> , loops:4                                                                     | range:[123,123],
    -> keep order:false | N/A             | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    -> ms, loops:4                                                                   | keep order:false
    ->                  | N/A             | N/A  |
+--
    -> ----------------------------+---------+---------+-----------+----------------------+-----+--------------
    ->
3 rows in set (0.02 sec)
```

```
CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    <--> ----------------------+----------+---------+----------+--------------+------------------------------
    <-->
| id                        | estRows    | actRows | task      | access object | execution info
    <-->                                                                    | operator info       |
    <--> memory         | disk |
+--
    <--> ----------------------+----------+---------+----------+--------------+------------------------------
    <-->
| TableReader_7             | 583.00     | 297     | root      |               | time:222.32506ms,
    <--> loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6    |
    <--> 88.6640625 KB | N/A  |
| └─Selection_6             | 583.00     | 297     | cop[tikv] |               | time:224ms, loops
    <--> :298                                                          | eq(test.t1.b, 123) | N/A
    <-->           | N/A  |
|   └─TableFullScan_5       | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    <--> :298                                                          | keep order:false    | N/A
    <-->           | N/A  |
+--
    <--> ----------------------+----------+---------+----------+--------------+------------------------------
    <-->
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

4.8.2.95.4 MySQL 兼容性

SHOW [GLOBAL|SESSION] BINDINGS 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.95.5 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- DROP [GLOBAL|SESSION] BINDING
- ANALYZE
- Optimizer Hints
- 执行计划管理(SPM)

4.8.2.96 SHOW BUILTINS

SHOW BUILTINS 语句用于列出TiDB 中所有的内置函数。

4.8.2.96.1 语法图

ShowBuiltinsStmt:

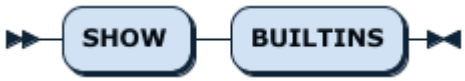


图 117: ShowBuiltinsStmt

4.8.2.96.2 示例

```
SHOW BUILTINS;
```

```
+----------------------------------------------- +
| Supported_builtin_functions |
+----------------------------------------------- +
| abs                          |
| acos                         |
| adddate                      |
| addtime                      |
| aes_decrypt                  |
| aes_encrypt                  |
| and                          |
| any_value                    |
| ascii                        |
| asin                         |
| atan                         |
| atan2                        |
| benchmark                    |
| bin                          |
| bit_count                    |
| bit_length                   |
| bitand                       |
| bitneg                       |
```

```
| bitor                      |
| bitxor                     |
| case                       |
| ceil                       |
| ceiling                    |
| char_func                  |
| char_length                |
| character_length           |
| charset                    |
| coalesce                   |
| coercibility               |
| collation                  |
| compress                   |
| concat                     |
| concat_ws                  |
| connection_id              |
| conv                       |
| convert                    |
| convert_tz                 |
| cos                        |
| cot                        |
| crc32                      |
| curdate                    |
| current_date               |
| current_role               |
| current_time               |
| current_timestamp          |
| current_user               |
| curtime                    |
| database                   |
| date                       |
| date_add                   |
| date_format                |
| date_sub                   |
| datediff                   |
| day                        |
| dayname                    |
| dayofmonth                 |
| dayofweek                  |
| dayofyear                  |
| decode                     |
| default_func               |
| degrees                    |
| des_decrypt                |
| des_encrypt                |
```

```
| div                        |
| elt                        |
| encode                     |
| encrypt                    |
| eq                         |
| exp                        |
| export_set                 |
| extract                    |
| field                      |
| find_in_set                |
| floor                      |
| format                     |
| format_bytes               |
| format_nano_time           |
| found_rows                 |
| from_base64                |
| from_days                  |
| from_unixtime              |
| ge                         |
| get_format                 |
| get_lock                   |
| getparam                   |
| getvar                     |
| greatest                   |
| gt                         |
| hex                        |
| hour                       |
| if                         |
| ifnull                     |
| in                         |
| inet6_aton                 |
| inet6_ntoa                 |
| inet_aton                  |
| inet_ntoa                  |
| insert_func                |
| instr                      |
| intdiv                     |
| interval                   |
| is_free_lock               |
| is_ipv4                    |
| is_ipv4_compat             |
| is_ipv4_mapped             |
| is_ipv6                    |
| is_used_lock               |
| isfalse                    |
```

```
| isnull                    |
| istrue                    |
| json_array                |
| json_array_append         |
| json_array_insert         |
| json_contains             |
| json_contains_path        |
| json_depth                |
| json_extract              |
| json_insert               |
| json_keys                 |
| json_length               |
| json_merge                |
| json_merge_patch          |
| json_merge_preserve       |
| json_object               |
| json_pretty               |
| json_quote                |
| json_remove               |
| json_replace              |
| json_search               |
| json_set                  |
| json_storage_size         |
| json_type                 |
| json_unquote              |
| json_valid                |
| last_day                  |
| last_insert_id            |
| lastval                   |
| lcase                     |
| le                        |
| least                     |
| left                      |
| leftshift                 |
| length                    |
| like                      |
| ln                        |
| load_file                 |
| localtime                 |
| localtimestamp            |
| locate                    |
| log                       |
| log10                     |
| log2                      |
| lower                     |
```

```
| lpad                        |
| lt                          |
| ltrim                       |
| make_set                    |
| makedate                    |
| maketime                    |
| master_pos_wait             |
| md5                         |
| microsecond                 |
| mid                         |
| minus                       |
| minute                      |
| mod                         |
| month                       |
| monthname                   |
| mul                         |
| name_const                  |
| ne                          |
| nextval                     |
| not                         |
| now                         |
| nulleq                      |
| oct                         |
| octet_length                |
| old_password                |
| or                          |
| ord                         |
| password_func               |
| period_add                  |
| period_diff                 |
| pi                          |
| plus                        |
| position                    |
| pow                         |
| power                       |
| quarter                     |
| quote                       |
| radians                     |
| rand                        |
| random_bytes                |
| regexp                      |
| release_all_locks           |
| release_lock                |
| repeat                      |
| replace                     |
```

```
| reverse                 |
| right                   |
| rightshift              |
| round                   |
| row_count               |
| rpad                    |
| rtrim                   |
| schema                  |
| sec_to_time             |
| second                  |
| session_user            |
| setval                  |
| setvar                  |
| sha                     |
| sha1                    |
| sha2                    |
| sign                    |
| sin                     |
| sleep                   |
| sm3                     |
| space                   |
| sqrt                    |
| str_to_date             |
| strcmp                  |
| subdate                 |
| substr                  |
| substring               |
| substring_index         |
| subtime                 |
| sysdate                 |
| system_user             |
| tan                     |
| tidb_decode_key         |
| tidb_decode_plan        |
| tidb_is_ddl_owner       |
| tidb_parse_tso          |
| tidb_version            |
| time                    |
| time_format             |
| time_to_sec             |
| timediff                |
| timestamp               |
| timestampadd            |
| timestampdiff           |
| to_base64               |
```

```
| to_days                          |
| to_seconds                       |
| trim                             |
| truncate                         |
| ucase                            |
| unaryminus                       |
| uncompress                       |
| uncompressed_length              |
| unhex                            |
| unix_timestamp                   |
| upper                            |
| user                             |
| utc_date                         |
| utc_time                         |
| utc_timestamp                    |
| uuid                             |
| uuid_short                       |
| validate_password_strength       |
| version                          |
| week                             |
| weekday                          |
| weekofyear                       |
| weight_string                    |
| xor                              |
| year                             |
| yearweek                         |
+---------------------------------------------- +
268 rows in set (0.00 sec)
```

4.8.2.96.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.97 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。

4.8.2.97.1 语法图

ShowCharsetStmt:



图 118: ShowCharsetStmt

CharsetKw:

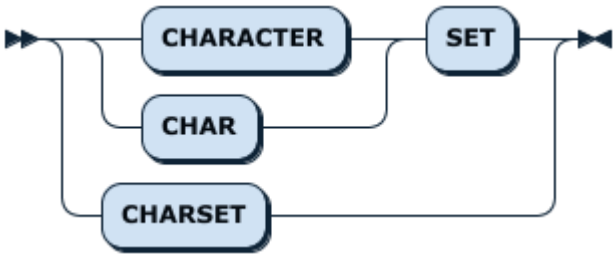


图 119: CharsetKw

4.8.2.97.2 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)
```

4.8.2.97.3 MySQL 兼容性

SHOW CHARACTER SET 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可以参考与 MySQL 兼容性对比。如发现任何兼容性差异，请在GitHub 上提交issue。

4.8.2.97.4 另请参阅

- SHOW COLLATION
- 字符集和排序规则

4.8.2.98 SHOW COLLATION

SHOW COLLATION 语句用于提供一个静态的排序规则列表，确保与 MySQL 客户端库的兼容性。

> **注意：**
>
> SHOW COLLATION 所展示的排序规则列表与TiDB 集群是否开启新排序规则框架有关，详情请见TiDB 字符集和排序规则。

4.8.2.98.1 语法图

ShowCollationStmt:

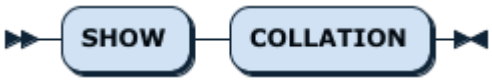


图 120: ShowCollationStmt

4.8.2.98.2 示例

若未开启新排序规则框架，仅展示二进制排序规则：

```
SHOW COLLATION;
```

```
+-------------+---------+------+---------+----------+-----------------+
| Collation   | Charset | Id   | Default | Compiled | Sortlen |
+-------------+---------+------+---------+----------+-----------------+
| utf8mb4_bin | utf8mb4 |   46 | Yes     | Yes      |       1 |
| latin1_bin  | latin1  |   47 | Yes     | Yes      |       1 |
| binary      | binary  |   63 | Yes     | Yes      |       1 |
| ascii_bin   | ascii   |   65 | Yes     | Yes      |       1 |
| utf8_bin    | utf8    |   83 | Yes     | Yes      |       1 |
+-------------+---------+------+---------+----------+-----------------+
5 rows in set (0.02 sec)
```

若开启了新排序规则框架，则在二进制排序规则之外，额外支持 `utf8_general_ci` 和 `utf8mb4_general_ci` 两种大小写和口音不敏感的排序规则：

```
SHOW COLLATION;
```

```
+--------------------+---------+------+---------+----------+-----------------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+-----------------+
| ascii_bin          | ascii   |   65 | Yes     | Yes      |       1 |
| binary             | binary  |   63 | Yes     | Yes      |       1 |
| gbk_bin            | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci     | gbk     |   28 | Yes     | Yes      |       1 |
| latin1_bin         | latin1  |   47 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |   83 | Yes     | Yes      |       1 |
| utf8_general_ci    | utf8    |   33 |         | Yes      |       1 |
| utf8_unicode_ci    | utf8    |  192 |         | Yes      |       1 |
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci | utf8mb4 |  224 |         | Yes      |       1 |
```

```
+--------------------+---------+------+---------+-----------+-----------------+
11 rows in set (0.001 sec)
```

4.8.2.98.3 MySQL 兼容性

`SHOW COLLATION` 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可参考与 MySQL 兼容性对比。如发现任何其他兼容性差异，请在GitHub 上提交issue。

4.8.2.98.4 另请参阅

- SHOW CHARACTER SET
- 字符集和排序规则

4.8.2.99 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。

4.8.2.99.1 语法图

ShowStmt:

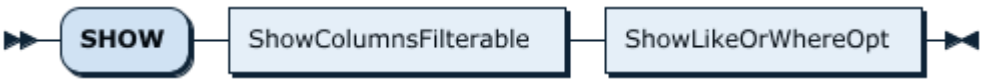


图 121: ShowStmt

ShowColumnsFilterable:



图 122: ShowColumnsFilterable

OptFull:



图 123: OptFull

FieldsOrColumns:



图 124: FieldsOrColumns

ShowTableAliasOpt:



图 125: ShowTableAliasOpt

FromOrIn:

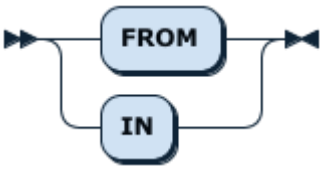


图 126: FromOrIn

TableName:

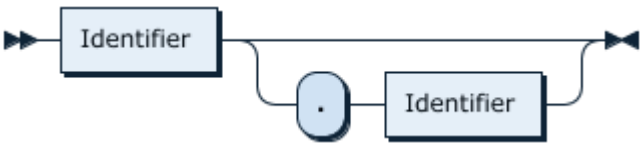


图 127: TableName

ShowDatabaseNameOpt:



图 128: ShowDatabaseNameOpt

DBName:



图 129: DBName

ShowLikeOrWhereOpt:

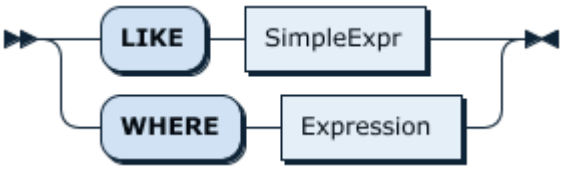


图 130: ShowLikeOrWhereOpt

4.8.2.99.2 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(255)     | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    <-> insert,update,references |          |
| User                  | char(32)      | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    <-> insert,update,references |          |
| authentication_string | text          | utf8mb4_bin | YES  |      | NULL    |       | select,
    <-> insert,update,references |          |
| plugin                | char(64)      | 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 |          |
| Repl_slave_priv         | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Repl_client_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Trigger_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Create_role_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Drop_role_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Account_locked          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Shutdown_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Reload_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| FILE_priv               | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Config_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| Create_Tablespace_Priv | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    <-> insert,update,references |          |
| User_attributes         | json          | NULL        | YES  |      | NULL    |       | select,
    <-> insert,update,references |          |
+-------------------------+---------------+-------------+------+------+---------+-------+-------------------------
    <->
38 rows in set (0.00 sec)
```

4.8.2.99.3 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。

4.8.2.99.4 另请参阅

- SHOW CREATE TABLE

4.8.2.100 SHOW CONFIG

`SHOW CONFIG` 语句用于展示TiDB 各个组件当前正在应用的配置，请注意，配置与系统变量作用于不同维度，请不要混淆，如果希望获取系统变量信息，请使用SHOW VARIABLES 语法。

4.8.2.100.1 语法图

ShowStmt:

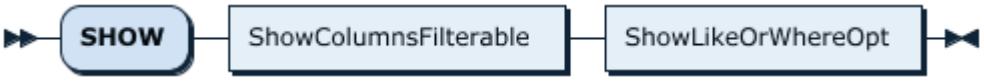


图 131: ShowStmt

ShowTargetFilterable:

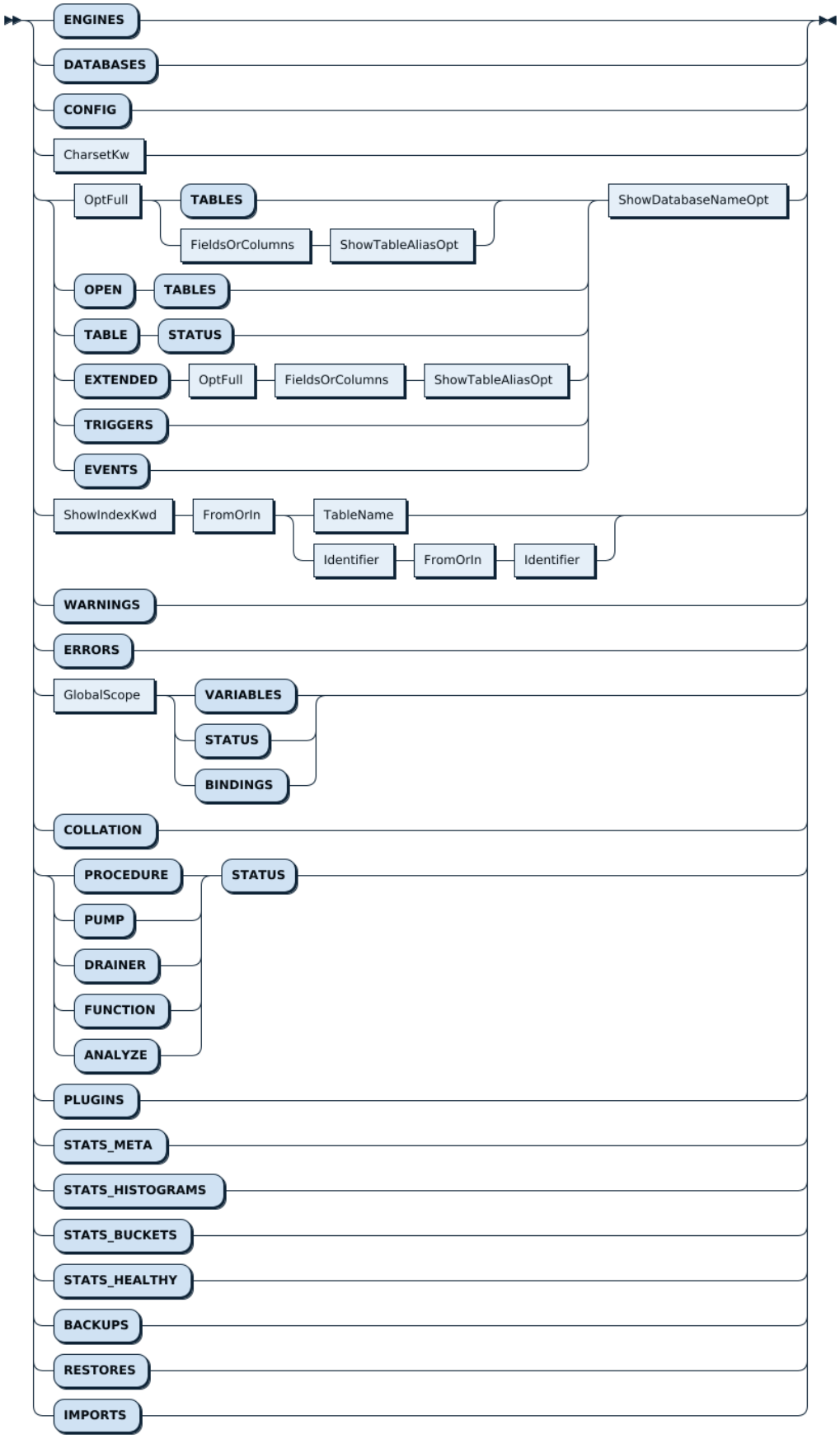


图 132: ShowTargetFilterable

4.8.2.100.2 示例

显示所有配置：

```
SHOW CONFIG;
```

```
+      +                +                                                  +
    <->
| Type | Instance       | Name                                               | Value
    <->                                                                  |
+------+----------------+----------------------------------------------------+
    <->
| tidb | 127.0.0.1:4000 | advertise-address                                  | 127.0.0.1
    <->                                                                |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                               |
    <->                                                                  |
| tidb | 127.0.0.1:4000 | binlog.enable                                      | false
    <->                                                                |
...
120 rows in set (0.01 sec)
```

显示 type 是 tidb 的配置：

```
SHOW CONFIG WHERE type = 'tidb' AND name = 'advertise-address';
```

```
+------+----------------+-------------------+-----------+
| Type | Instance       | Name              | Value     |
+------+----------------+-------------------+-----------+
| tidb | 127.0.0.1:4000 | advertise-address | 127.0.0.1 |
+------+----------------+-------------------+-----------+
1 row in set (0.05 sec)
```

也可以用 LIKE 子句来显示 type 是 tidb 的配置：

```
SHOW CONFIG LIKE 'tidb';
```

```
+      +                +                                                  +
   <->
| Type | Instance       | Name                                               | Value
   <->                                                                  |
+------+----------------+----------------------------------------------------+
   <->
| tidb | 127.0.0.1:4000 | advertise-address                                  | 127.0.0.1
   <->                                                                |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                               |
   <->                                                                  |
| tidb | 127.0.0.1:4000 | binlog.enable                                      | false
   <->                                                                |
```

```
...
40 rows in set (0.01 sec)
```

4.8.2.100.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.100.4 另请参阅

- SHOW VARIABLES

4.8.2.101 SHOW CREATE PLACEMENT POLICY

SHOW CREATE PLACEMENT POLICY 语句可用于查看放置策略当前的定义，并在另一个TiDB 集群中重新创建该策略。

4.8.2.101.1 语法图

```
ShowCreatePlacementPolicyStmt ::=
    "SHOW" "CREATE" "PLACEMENT" "POLICY" PolicyName

PolicyName ::=
    Identifier
```

4.8.2.101.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW CREATE PLACEMENT POLICY p1\G;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

***************************[ 1. row ]***************************
Policy        | p1
Create Policy | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-
    <-> west-1" FOLLOWERS=4
1 row in set (0.00 sec)
```

4.8.2.101.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.101.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- ALTER PLACEMENT POLICY
- DROP PLACEMENT POLICY

4.8.2.102 SHOW CREATE RESOURCE GROUP

`SHOW CREATE RESOURCE GROUP` 语句可用于查看资源组当前的定义。

4.8.2.102.1 语法图

```
ShowCreateResourceGroupStmt ::=
    "SHOW" "CREATE" "RESOURCE" "GROUP" ResourceGroupName

ResourceGroupName ::=
    Identifier
```

4.8.2.102.2 示例

查看资源组 `rg1` 当前的定义：

```
CREATE RESOURCE GROUP rg1 RU_PER_SEC=100;
Query OK, 0 rows affected (0.10 sec)
```

```
SHOW CREATE RESOURCE GROUP rg1;
***************************[ 1. row ]***************************
+----------------+-----------------------------------------------------------------+
| Resource_Group | Create Resource Group                                           |
+----------------+-----------------------------------------------------------------+
| rg1            | CREATE RESOURCE GROUP `rg1` RU_PER_SEC=100 PRIORITY=MEDIUM |
+----------------+-----------------------------------------------------------------+
1 row in set (0.01 sec)
```

4.8.2.102.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.102.4 另请参阅

- TiDB RESOURCE CONTROL
- CREATE RESOURCE GROUP
- ALTER RESOURCE GROUP
- DROP RESOURCE GROUP

4.8.2.103 SHOW CREATE SEQUENCE

`SHOW CREATE SEQUENCE` 语句用于查看一个序列的详细信息，类似于 `SHOW CREATE TABLE` 语句。

4.8.2.103.1 语法图

ShowCreateSequenceStmt:



图 133: ShowCreateSequenceStmt

TableName:

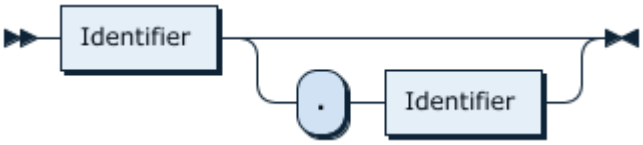


图 134: TableName

4.8.2.103.2 示例

```
CREATE SEQUENCE seq;
```

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

```
SHOW CREATE SEQUENCE seq;
```

```
+       +----------------------
```

4.8.2.103.4 另请参阅

- CREATE SEQUENCE
- DROP SEQUENCE

4.8.2.104 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。

4.8.2.104.1 语法图

ShowCreateTableStmt:

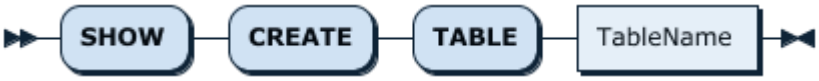


图 135: ShowCreateTableStmt

TableName:

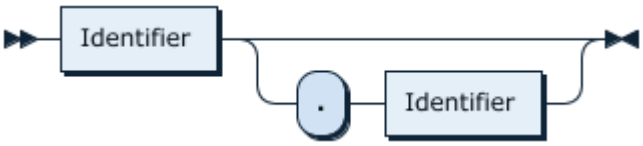


图 136: TableName

4.8.2.104.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+       +----------------------
```

4.8.2.104.3 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。

4.8.2.104.4 另请参阅

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

4.8.2.105 SHOW CREATE DATABASE

SHOW CREATE DATABASE 语句用于显示用 SQL 重新创建已有库的确切语句。SHOW CREATE SCHEMA 与其同义。

4.8.2.105.1 语法图

ShowCreateDatabaseStmt:

```
ShowCreateDatabaseStmt ::=
    "SHOW" "CREATE" "DATABASE" | "SCHEMA" ("IF" "NOT" "EXISTS")? DBName
```

4.8.2.105.2 示例

```
CREATE DATABASE test;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW CREATE DATABASE test;
```

```
+----------+-----------------------------------------------------------------+
| Database | Create Database                                                 |
+----------+-----------------------------------------------------------------+
| test     | CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4 */ |
+----------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW CREATE SCHEMA IF NOT EXISTS test;
```

```
+----------+-----------------------------------------------------------------------------------------------------+
    ↪
| Database | Create Database
    ↪                                                                                                  |
+----------+-----------------------------------------------------------------------------------------------------+
    ↪
```

```
| test     | CREATE DATABASE /*!32312 IF NOT EXISTS*/ `test` /*!40100 DEFAULT CHARACTER SET
    <⟷ utf8mb4 */ |
+----------+-----------------------------------------------------------------------------------------------------+
    <⟷
1 row in set (0.00 sec)
```

4.8.2.105.3 MySQL 兼容性

SHOW 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。

4.8.2.105.4 另请参阅

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

4.8.2.106 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。

4.8.2.106.1 语法图

ShowCreateUserStmt:

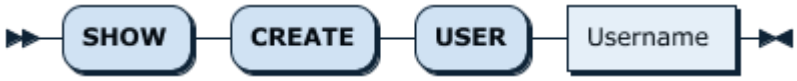


图 137: ShowCreateUserStmt

Username:

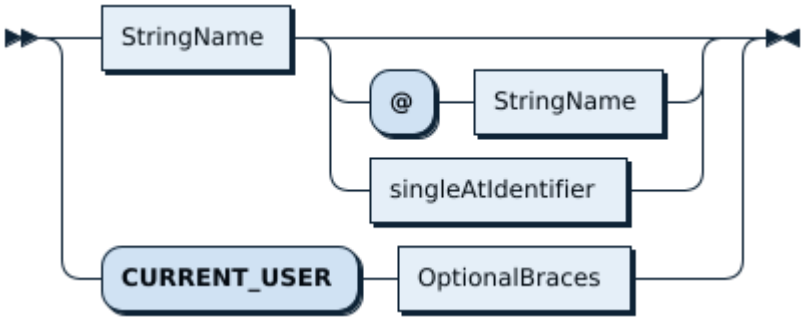


图 138: Username

4.8.2.106.2 示例

```
SHOW CREATE USER 'root';
```

```
+
----------------------------------------------------------------------------------------------------------------------------------
    <->
| CREATE USER for root@%
    <->
    <-> |
+
----------------------------------------------------------------------------------------------------------------------------------
    <->
| CREATE USER 'root'@'%' IDENTIFIED WITH 'mysql_native_password' AS '' REQUIRE NONE PASSWORD
    <-> EXPIRE DEFAULT ACCOUNT UNLOCK |
+
----------------------------------------------------------------------------------------------------------------------------------
    <->
1 row in set (0.00 sec)

mysql> SHOW GRANTS FOR 'root';
+--------------------------------------------+
| Grants for root@%                          |
+--------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+--------------------------------------------+
1 row in set (0.00 sec)
```

4.8.2.106.3 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]。

4.8.2.106.4 另请参阅

- CREATE USER
- SHOW GRANTS
- DROP USER

4.8.2.107 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。

4.8.2.107.1 语法图

ShowDatabasesStmt:



图 139: ShowDatabasesStmt

ShowLikeOrWhereOpt:

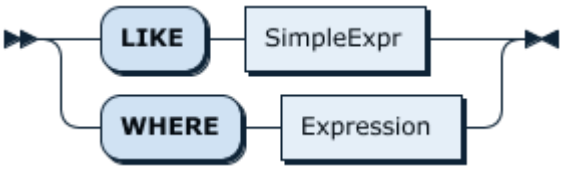


图 140: ShowLikeOrWhereOpt

4.8.2.107.2 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)
```

4.8.2.107.3 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。

4.8.2.107.4 另请参阅

- SHOW SCHEMAS
- DROP DATABASE
- CREATE DATABASE

4.8.2.108 SHOW DRAINER STATUS

SHOW DRAINER STATUS 语句用于显示集群中所有 Drainer 的状态信息。

4.8.2.108.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------------------------- |
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time       |
+----------|----------------|--------|--------------------|---------------------------------------- |
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------------------------- |
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------------------------- |
2 rows in set (0.00 sec)
```

4.8.2.108.2 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.108.3 另请参阅

- SHOW PUMP STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

4.8.2.109 SHOW ENGINES

SHOW ENGINES 语句用于列出所有支持的存储引擎，该语法仅提供 MySQL 兼容性。

4.8.2.109.1 语法图

ShowEnginesStmt:

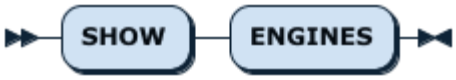


图 141: ShowEnginesStmt

```
SHOW ENGINES;
```

4.8.2.109.2 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)
```

4.8.2.109.3 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。

4.8.2.110 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。

4.8.2.110.1 语法图

ShowErrorsStmt:

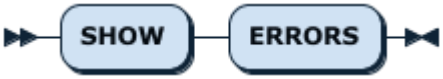


图 142: ShowErrorsStmt

4.8.2.110.2 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)
```

4.8.2.110.3 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。

4.8.2.110.4 另请参阅

- SHOW WARNINGS

4.8.2.111 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。

4.8.2.112 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。

4.8.2.112.1 语法图

ShowGrantsStmt:

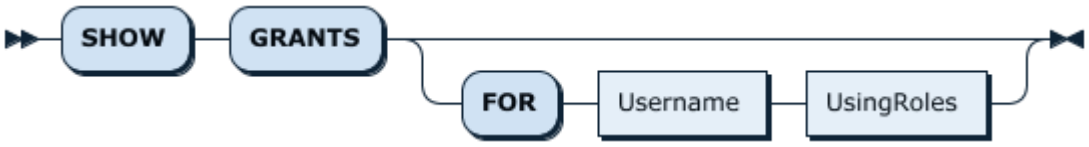


图 143: ShowGrantsStmt

Username:

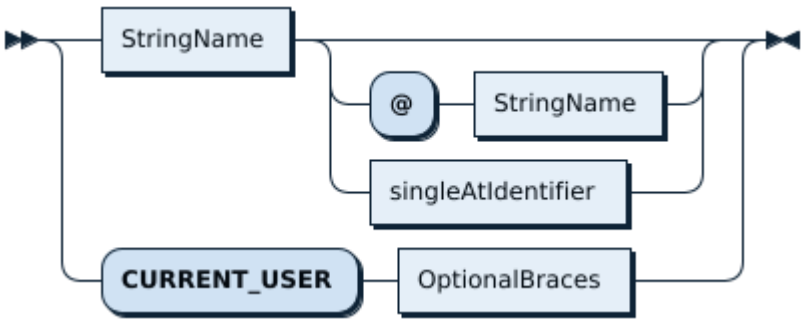


图 144: Username

UsingRoles:



图 145: UsingRoles

RolenameList:

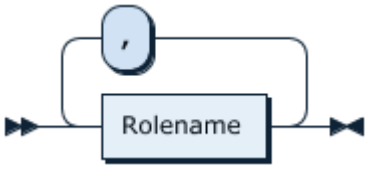


图 146: RolenameList

Rolename:

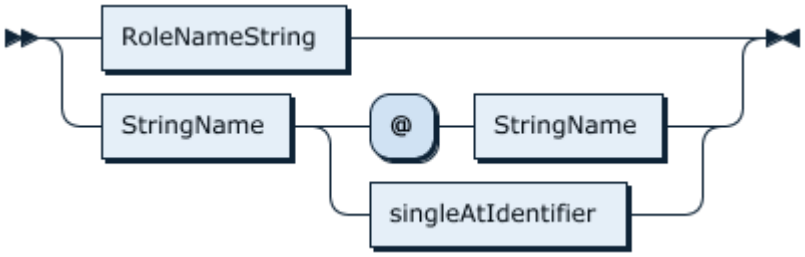


图 147: Rolename

4.8.2.112.2 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)
```

4.8.2.112.3 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。

4.8.2.112.4 另请参阅

- SHOW CREATE USER
- GRANT

4.8.2.113 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。

4.8.2.114 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。

4.8.2.114.1 语法图

ShowIndexStmt:

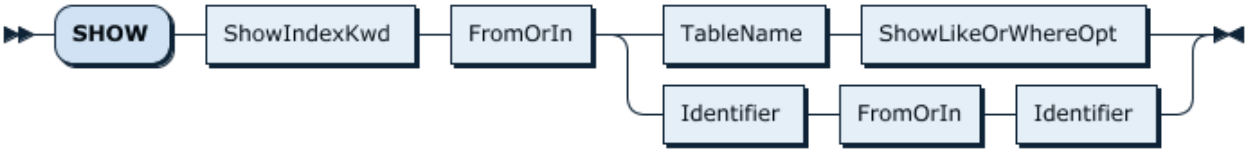


图 148: ShowIndexStmt

ShowIndexKwd:

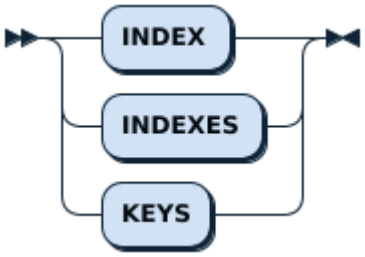


图 149: ShowIndexKwd

FromOrIn:

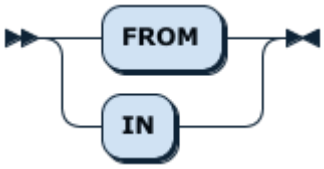


图 150: FromOrIn

TableName:

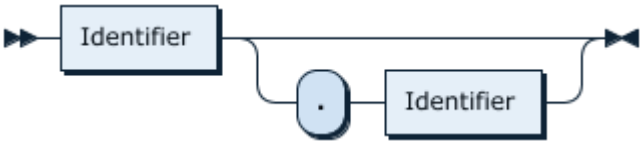


图 151: TableName

ShowLikeOrWhereOpt:

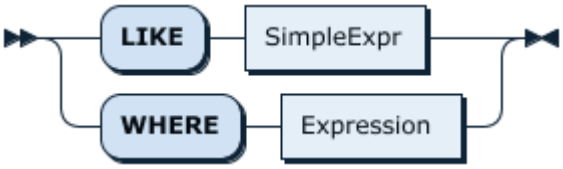


图 152: ShowLikeOrWhereOpt

4.8.2.114.2 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 | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+-------+----
    <->
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    <-> | NULL    |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    <-> | NULL    | YES  | BTREE      |         |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+-------+----
    <->
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 | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+-------+----
    <->
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    <-> | NULL    |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    <-> | NULL    | YES  | BTREE      |         |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+-------+----
    <->
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 | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+-------+----
    <->
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    <-> | NULL    |      | BTREE      |         |               | YES
```

```
    | NULL       |
| t1    |           1 | col1     |            1 | col1        | A         |           0 |     NULL
    <-> | NULL    | YES  | BTREE      |          |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+-------+----
    <->
2 rows in set (0.00 sec)
```

4.8.2.114.3 MySQL 兼容性

MySQL 中的 `Cardinality` 列返回该索引上不同值的个数，而TiDB 中的 `Cardinality` 列始终返回 `0`。

4.8.2.114.4 另请参阅

- SHOW CREATE TABLE
- DROP INDEX
- CREATE INDEX

4.8.2.115 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。

4.8.2.116 SHOW MASTER STATUS

`SHOW MASTER STATUS` 语句用于显示集群当前最新的TSO 信息。

4.8.2.116.1 示例

```
SHOW MASTER STATUS;
```

```
+-------------+--------------------+--------------+------------------+--------------------------------------+
| File        | Position           | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-------------+--------------------+--------------+------------------+--------------------------------------+
| tidb-binlog | 416916363252072450 |              |                  |                   |
+-------------+--------------------+--------------+------------------+--------------------------------------+
1 row in set (0.00 sec)
```

4.8.2.116.2 MySQL 兼容性

`SHOW MASTER STATUS` 语句与 MySQL 兼容，但是执行结果有差异，在 MySQL 中执行结果为binlog 的位置信息，而在 TiDB 中为最新的TSO 信息。

4.8.2.116.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

4.8.2.117 SHOW PLACEMENT

SHOW PLACEMENT 汇总了所有放置策略(placement policy)，并用统一的形式呈现相关信息。

本语句返回结果中的 Scheduling_State 列标识了 Placement Driver (PD) 在当前对象上的调度进度，有以下可能结果：

- PENDING: PD 没有进行调度。可能的原因之一是放置规则虽然语法上正确，但集群拓扑并不满足。比如指定 FOLLOWERS=4 但只有 3 个可用作 follower 的 TiKV 实例。
- INPROGRESS: PD 正在进行调度。
- SCHEDULED: PD 调度完成。

4.8.2.117.1 语法图

```
ShowStmt ::=
    "PLACEMENT"
```

4.8.2.117.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW PLACEMENT;
```

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

Query OK, 0 rows affected (0.00 sec)

+--
    <-->--------------+-------------------------------------------------------------------------------+-------------------+
    <-->
| Target         | Placement                                                                     |
    <--> Scheduling_State |
+--
    <--> --------------+-------------------------------------------------------------------------------+-------------------+
    <-->
| POLICY p1      | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 | NULL
    <-->              |
| DATABASE test | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    <--> INPROGRESS        |
```

```
| TABLE test.t1 | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    <-> INPROGRESS        |
+--
    <->-------------+-------------------------------------------------------------------------------+-------------------+
    <->
4 rows in set (0.00 sec)
```

4.8.2.117.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.117.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT FOR
- CREATE PLACEMENT POLICY

4.8.2.118 SHOW PLACEMENT FOR

SHOW PLACEMENT FOR 用于汇总所有放置策略(placement policy)，并用统一的形式呈现特定表、数据库或分区的信息。

本语句返回结果中的 Scheduling_State 列标识了 Placement Driver (PD) 在当前对象上的调度进度，有以下可能的结果：

- PENDING: PD 没有进行调度。可能的原因之一是放置规则虽然语法上正确，但集群拓扑并不满足。比如指定 FOLLOWERS=4 但只有 3 个可用作 follower 的 TiKV 实例。
- INPROGRESS: PD 正在进行调度。
- SCHEDULED: PD 调度完成。

4.8.2.118.1 语法图

```
ShowStmt ::=
    "PLACEMENT" "FOR" ShowPlacementTarget

ShowPlacementTarget ::=
    DatabaseSym DBName
|   "TABLE" TableName
|   "TABLE" TableName "PARTITION" Identifier
```

4.8.2.118.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
use test;
ALTER DATABASE test PLACEMENT POLICY=p1;
```

```
CREATE TABLE t1 (a INT);
SHOW PLACEMENT FOR DATABASE test;
SHOW  PLACEMENT  FOR  TABLE  t1;
SHOW CREATE TABLE t1\G;
CREATE TABLE t3 (a INT) PARTITION BY RANGE (a) (PARTITION p1 VALUES LESS THAN (10), PARTITION p2
    <-> VALUES LESS THAN (20));
SHOW PLACEMENT FOR TABLE t3 PARTITION p1\G;
```

```
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

+--
    <->             +                                                                                +
    <->
| Target        | Placement                                                                          |
    <-> Scheduling_State |
+--
    <->             +                                                                                +
    <->
| DATABASE test | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    <-> INPROGRESS       |
+--
    <->             +                                                                                +
    <>
1 row in set (0.00 sec)

+---------------+-------------+-----------------------------+
| Target        | Placement   | Scheduling_State |
+---------------+-------------+-----------------------------+
| TABLE test.t1 | FOLLOWERS=4 | INPROGRESS       |
+---------------+-------------+-----------------------------+
1 row in set (0.00 sec)

***************************[ 1. row ]***************************
Table        | t1
Create Table | CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /*T![placement] PLACEMENT POLICY=`p1`
    <-> */
1 row in set (0.00 sec)

***************************[ 1. row ]***************************
```

```
Target           | TABLE test.t3 PARTITION p1
Placement        | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4
Scheduling_State | PENDING
1 row in set (0.00 sec)
```

4.8.2.118.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.118.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY

4.8.2.119 SHOW PLACEMENT LABELS

SHOW PLACEMENT LABELS 汇总可用于放置规则(Placement Rules) 的标签(label) 和值。

4.8.2.119.1 语法图

```
ShowStmt ::=
    "PLACEMENT" "LABELS"
```

4.8.2.119.2 示例

```
SHOW PLACEMENT LABELS;
```

```
+--------+----------------------------+
| Key    | Values                     |
+--------+----------------------------+
| region | ["us-east-1"]              |
| zone   | ["us-east-1a"]             |
+--------+----------------------------+
2 rows in set (0.00 sec)
```

4.8.2.119.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.119.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY

4.8.2.120 SHOW PLUGINS

`SHOW PLUGINS` 用于查看TiDB 安装的插件，各个插件运行的状态以及插件版本信息。

4.8.2.120.1 语法图

ShowStmt:

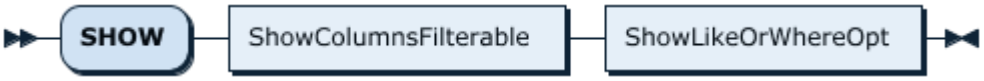


图 153: ShowStmt

ShowTargetFilterable:

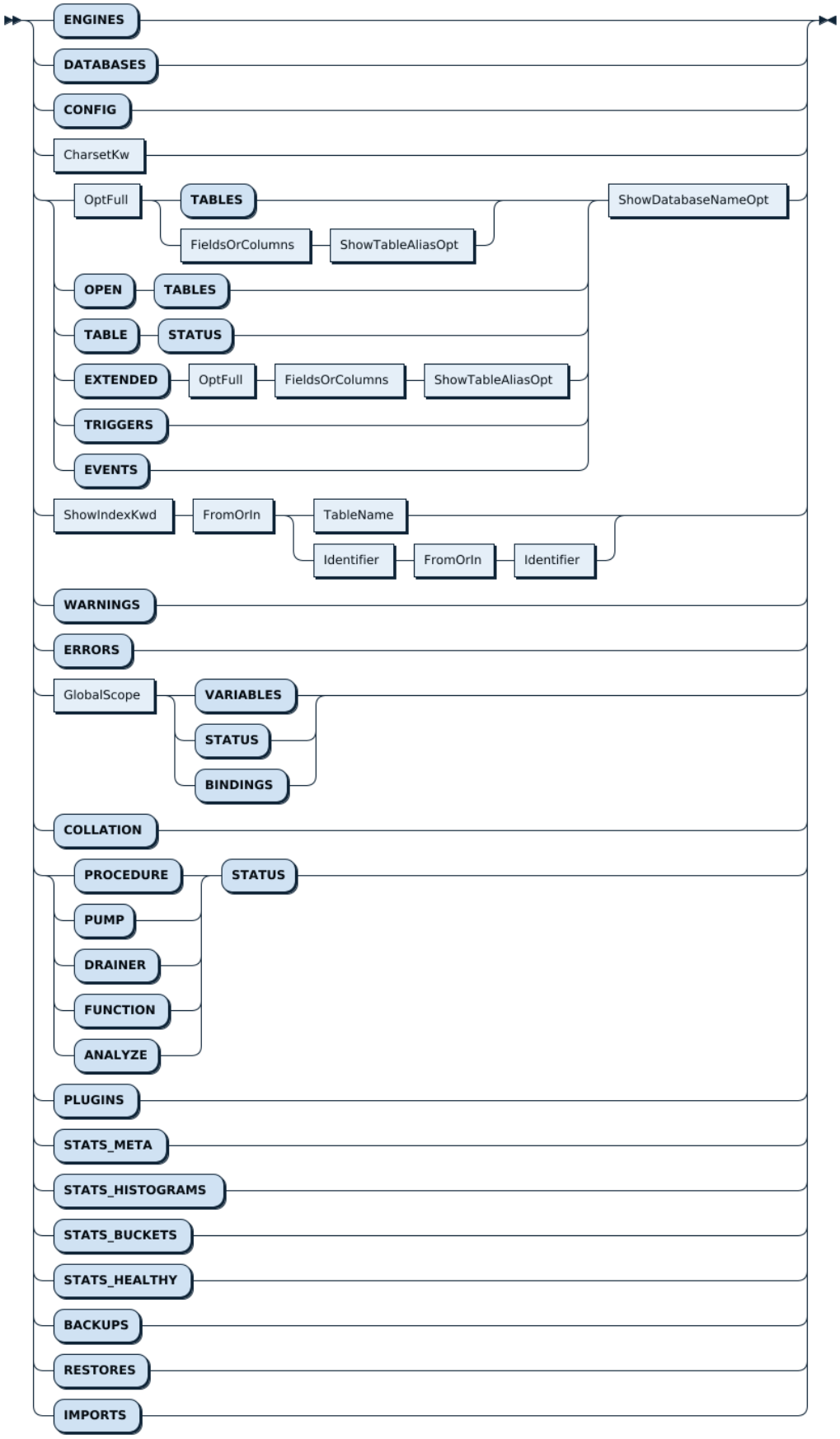


图 154: ShowTargetFilterable

4.8.2.120.2 示例

```
SHOW PLUGINS;
```

```
+-------+--------------+-------+----------------------------+---------+-------------------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+-------------------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |          | 1       |
+-------+--------------+-------+----------------------------+---------+-------------------+
1 row in set (0.000 sec)
```

SHOW PLUGINS 也支持模糊匹配：

```
SHOW PLUGINS LIKE 'a%';
```

```
+-------+--------------+-------+----------------------------+---------+-------------------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+-------------------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |          | 1       |
+-------+--------------+-------+----------------------------+---------+-------------------+
1 row in set (0.000 sec)
```

4.8.2.120.3 MySQL 兼容性

SHOW PLUGINS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在GitHub 上提交issue。

4.8.2.121 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。

4.8.2.121.1 语法图

ShowStmt:

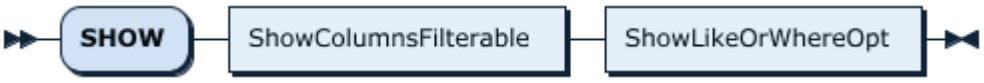


图 155: ShowStmt

4.8.2.121.2 示例

```
show privileges;
```

```
mysql> show privileges;
+--
    <-->-------------------------------+-------------------------------------------------------------+-------------------------------------------------------------
    <-->
```

Privilege	Context	Comment
Alter	Tables	To alter the table
Alter routine	Functions,Procedures	To alter or drop stored functions/procedures
Config	Server Admin	To use SHOW CONFIG and SET CONFIG statements
Create	Databases,Tables,Indexes	To create new databases and tables
Create routine	Databases	To use CREATE FUNCTION/PROCEDURE
Create role	Server Admin	To create new roles
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
Drop role	Server Admin	To drop roles
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                     |
| BACKUP_ADMIN                   | Server Admin                    |
    <-->                                                           |
| RESTORE_ADMIN                  | Server Admin                    |
    <-->                                                           |
| SYSTEM_USER                    | Server Admin                    |
    <-->                                                           |
| SYSTEM_VARIABLES_ADMIN         | Server Admin                    |
    <-->                                                           |
| ROLE_ADMIN                     | Server Admin                    |
    <-->                                                           |
| CONNECTION_ADMIN               | Server Admin                    |
    <-->                                                           |
| PLACEMENT_ADMIN                | Server Admin                    |
    <-->                                                           |
| DASHBOARD_CLIENT               | Server Admin                    |
    <-->                                                           |
```

```
| RESTRICTED_TABLES_ADMIN          | Server Admin                          |
    <->                                                                    |
| RESTRICTED_STATUS_ADMIN          | Server Admin                          |
    <->                                                                    |
| RESTRICTED_VARIABLES_ADMIN       | Server Admin                          |
    <->                                                                    |
| RESTRICTED_USER_ADMIN            | Server Admin                          |
    <->                                                                    |
| RESTRICTED_CONNECTION_ADMIN      | Server Admin                          |
    <->                                                                    |
| RESTRICTED_REPLICA_WRITER_ADMIN  | Server Admin                          |
    <->                                                                    |
| RESOURCE_GROUP_ADMIN             | Server Admin                          |
    <->                                                                    |
+--
    <>-------------------------------+---------------------------------------+-----------------------------
    <>
49 rows in set (0.00 sec)
```

4.8.2.121.3 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。

4.8.2.121.4 另请参阅

- SHOW GRANTS
- GRAN T <privileges>

4.8.2.122 SHOW [FULL] PROCESSLIST

`SHOW [FULL] PROCESSLIST` 语句列出连接到相同TiDB 服务器的当前会话。

4.8.2.122.1 语法图

ShowProcesslistStmt:



图 156: ShowProcesslistStmt

OptFull:



图 157: OptFull

4.8.2.122.2 示例

```
SHOW PROCESSLIST;
```

```
+------+------+-----------------+------+---------+------+------------+-------------------------------+
| Id   | User | Host            | db   | Command | Time | State      | Info                          |
+------+------+-----------------+------+---------+------+------------+-------------------------------+
|    5 | root | 127.0.0.1:45970 | test | Query   |    0 | autocommit | SHOW PROCESSLIST |
+------+------+-----------------+------+---------+------+------------+-------------------------------+
1 rows in set (0.00 sec)
```

以上返回结果中的主要字段描述如下：

- Command：SQL 语句的类型，通常值为 Query。
- Time：SQL 语句开始执行的时间。
- State：SQL 语句的状态。常见的值是 autocommit，表示该 SQL 语句是自动提交的。in transaction 表示该 SQL 语句处于事务中。
- Info：表示具体的 SQL 文本。除非指定可选关键字 FULL，否则文本会被截断。

4.8.2.122.3 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。

4.8.2.122.4 另请参阅

- KILL [TIDB]

4.8.2.123 SHOW PROFILES

SHOW PROFILES 语句目前只会返回空结果。

4.8.2.123.1 语法图

ShowStmt:

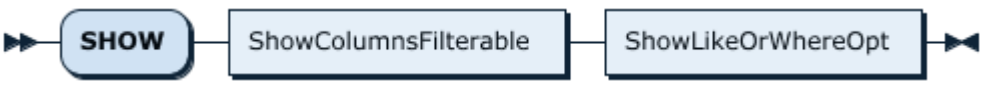


图 158: ShowStmt

4.8.2.123.2 示例

```
SHOW PROFILES
```

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

4.8.2.123.3 MySQL 兼容性

该语句仅与 MySQL 兼容，无其他作用。执行 `SHOW PROFILES` 始终返回空结果。

4.8.2.124 SHOW PUMP STATUS

`SHOW PUMP STATUS` 语句用于显示集群中所有 Pump 的状态信息。

4.8.2.124.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |      Address     | State  |    Max_Commit_Ts     |    Update_Time       |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

4.8.2.124.2 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.124.3 另请参阅

- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

4.8.2.125 SHOW SCHEMAS

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

4.8.2.126 SHOW STATS_HEALTHY

`SHOW STATS_HEALTHY` 语句可以预估统计信息的准确度，也就是健康度。健康度低的表可能会生成次优查询执行计划。

可以通过执行 `ANALYZE` 表命令来改善表的健康度。当表的健康度下降到低于`tidb_auto_analyze_ratio` 时，则会自动执行 `ANALYZE` 命令。

4.8.2.126.1 语法图

ShowStmt

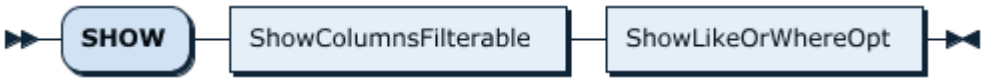


图 159: ShowStmt

ShowTargetFilterable

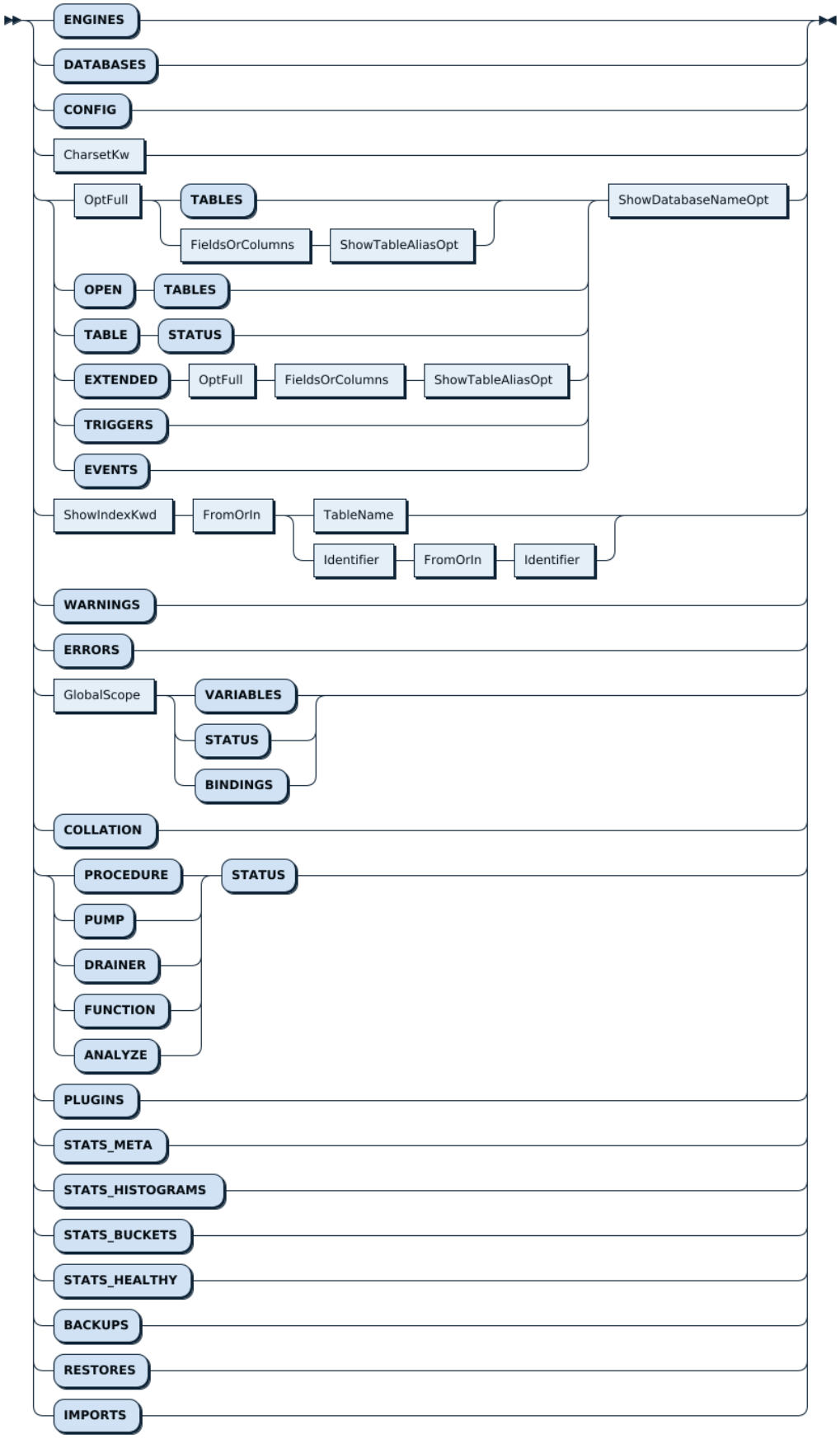


图 160: ShowTargetFilterable

ShowLikeOrWhereOpt

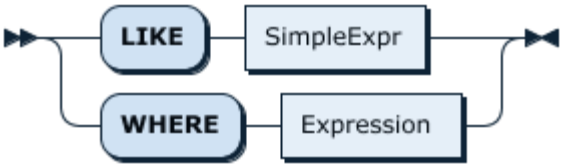


图 161: ShowLikeOrWhereOpt

4.8.2.126.2 示例

加载示例数据并运行 ANALYZE 命令：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad VARBINARY(255),
 INDEX(b)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    <-> LIMIT 100000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
SHOW STATS_HEALTHY; # should be 100% healthy
```

```
SHOW STATS_HEALTHY;
```

```
+---------+------------+----------------+---------------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------------+
| test    | t1         |                |     100 |
+---------+------------+----------------+---------------+
1 row in set (0.00 sec)
```

执行批量更新来删除大约 30% 的记录，然后检查统计信息的健康度：

```
DELETE FROM t1 WHERE id BETWEEN 101010 AND 201010; # delete about 30% of records
SHOW STATS_HEALTHY;
```

```
SHOW STATS_HEALTHY;
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |      50 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

4.8.2.126.3 MySQL 兼容性

`SHOW STATS_HEALTHY` 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.126.4 另请参阅

- ANALYZE
- 统计信息简介

4.8.2.127 SHOW STATS_HISTOGRAMS

你可以使用 `SHOW STATS_HISTOGRAMS` 语句查看统计信息中直方图的相关信息。

4.8.2.127.1 语法图

ShowStmt

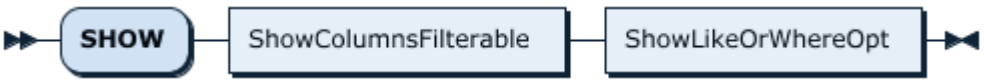


图 162: ShowStmt

ShowTargetFiltertable

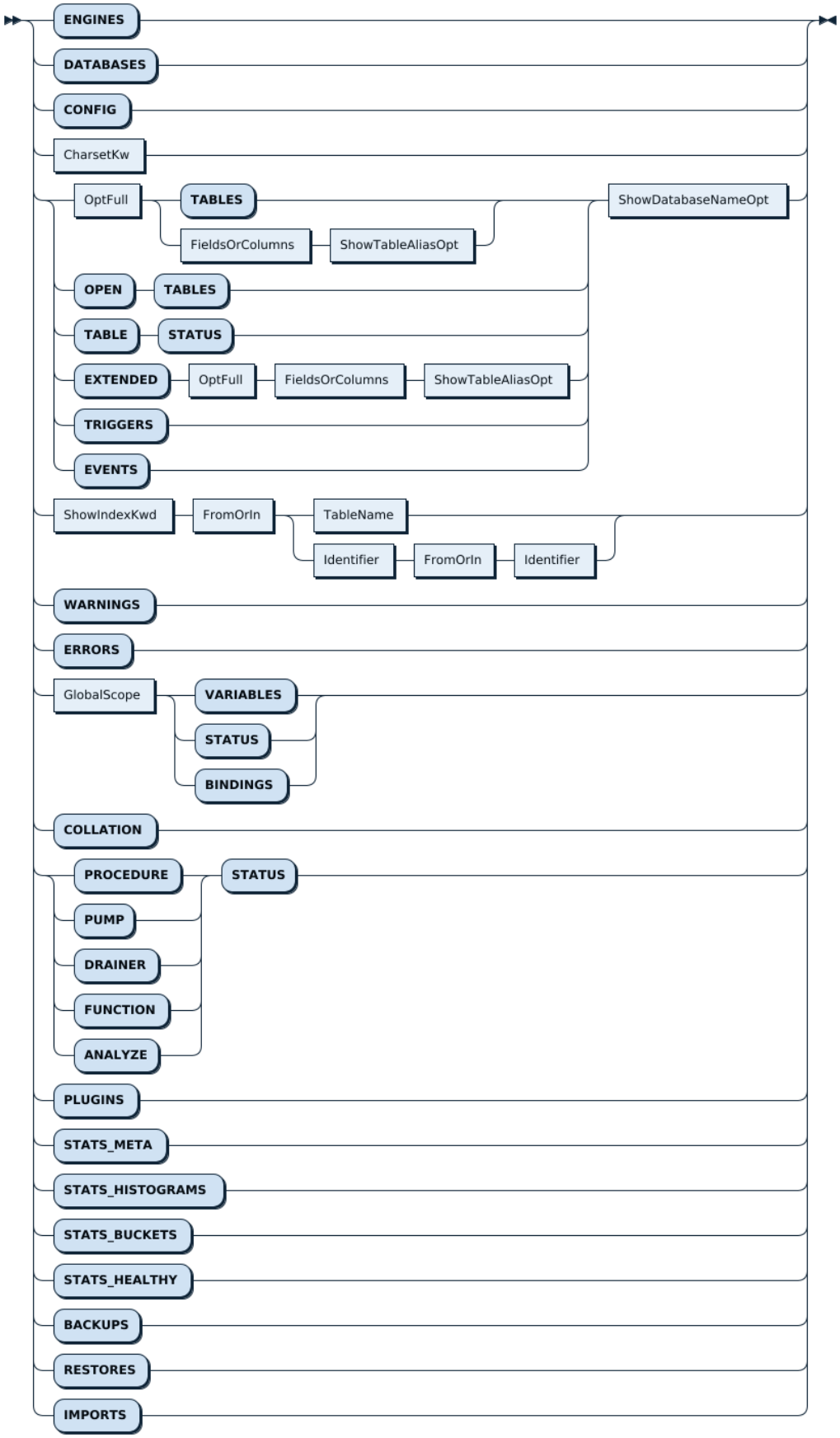


图 163: ShowTargetFilterable

ShowLikeOrWhereOpt

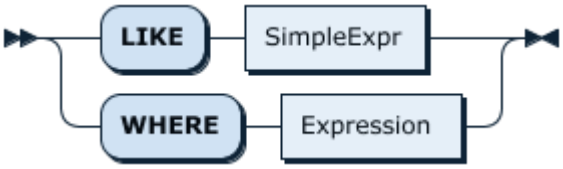


图 164: ShowLikeOrWhereOpt

4.8.2.127.2 示例

```
show stats_histograms;
```

```
+--
    <-> -------+------------+----------------+-------------+----------+---------------------+-----------------+---
    <->
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time          |
    <-> Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    <-> -------+------------+----------------+-------------+----------+---------------------+-----------------+---
    <->
| test    | t          |                | a           |        0 | 2020-05-25 19:20:00 |
    <->              7 |          0 |              1 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    <->              6 |          0 |              8 |           0 |
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    <->              6 |          0 |           1.67 |           1 |
+--
    <-> -------+------------+----------------+-------------+----------+---------------------+-----------------+---
    <->
3 rows in set (0.00 sec)
```

```
show stats_histograms where table_name = 't2';
```

```
+--
    <-> -------+------------+----------------+-------------+----------+---------------------+-----------------+---
    <->
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time          |
    <-> Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    <-> -------+------------+----------------+-------------+----------+---------------------+-----------------+---
    <->
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    <->              6 |          0 |           1.67 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    <->              6 |          0 |              8 |           0 |
```

```
+--
    <-> -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    <->
2 rows in set (0.00 sec)
```

4.8.2.127.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.127.4 另请参阅

- ANALYZE
- 统计信息介绍

4.8.2.128 SHOW STATS_LOCKED

SHOW STATS_LOCKED 语句显示统计信息被锁定的表。

> **警告：**
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

4.8.2.128.1 语法图

```
ShowStatsLockedStmt ::= 'SHOW' 'STATS_LOCKED' ShowLikeOrWhereOpt

ShowLikeOrWhereOpt ::= 'LIKE' SimpleExpr | 'WHERE' Expression
```

4.8.2.128.2 示例

创建表t，插入一些数据，在未锁定表t 的统计信息的情况下成功执行 ANALYZE 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
```

```
+-------+------+-----------------------------------------------------------------------+
| Level | Code | Message                                                               |
+-------+------+-----------------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+-----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

锁定表t 的统计信息，执行 ANALYZE 语句，使用 SHOW STATS_LOCKED 可查看表t 的统计信息已被锁定。warning 提示跳过对表t 的 ANALYZE。

```
mysql> lock stats t;
Query OK, 0 rows affected (0.00 sec)

mysql> show stats_locked;
+---------+------------+----------------+--------+
| Db_name | Table_name | Partition_name | Status |
+---------+------------+----------------+--------+
| test    | t          |                | locked |
+---------+------------+----------------+--------+
1 row in set (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> show warnings;

+----------+------+-----------------------------------------------------------------------+
| Level    | Code | Message                                                               |
+----------+------+-----------------------------------------------------------------------+
| Note     | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
| Warning | 1105 | skip analyze locked table: t                                          |
+----------+------+-----------------------------------------------------------------------+
2 rows in set (0.00 sec)
```

解锁表t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+-------+------+-----------------------------------------------------------------------+
| Level | Code | Message                                                               |
+-------+------+-----------------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
```

```
+-------+------+-------------------------------------------------------------------+
1 row in set (0.00 sec)
```

4.8.2.128.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.128.4 另请参阅

- 统计信息
- LOCK STATS
- UNLOCK STATS

4.8.2.129 SHOW STATS_META

你可以通过 `SHOW STATS_META` 来查看表的总行数以及修改的行数等信息，可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_META` 会输出 6 列，具体如下：

列名	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。

4.8.2.129.1 语法图

ShowStmt

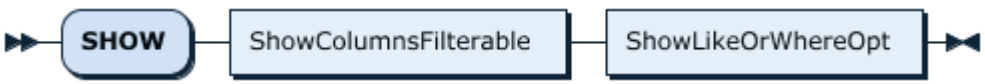


图 165: ShowStmt

ShowTargetFiltertable

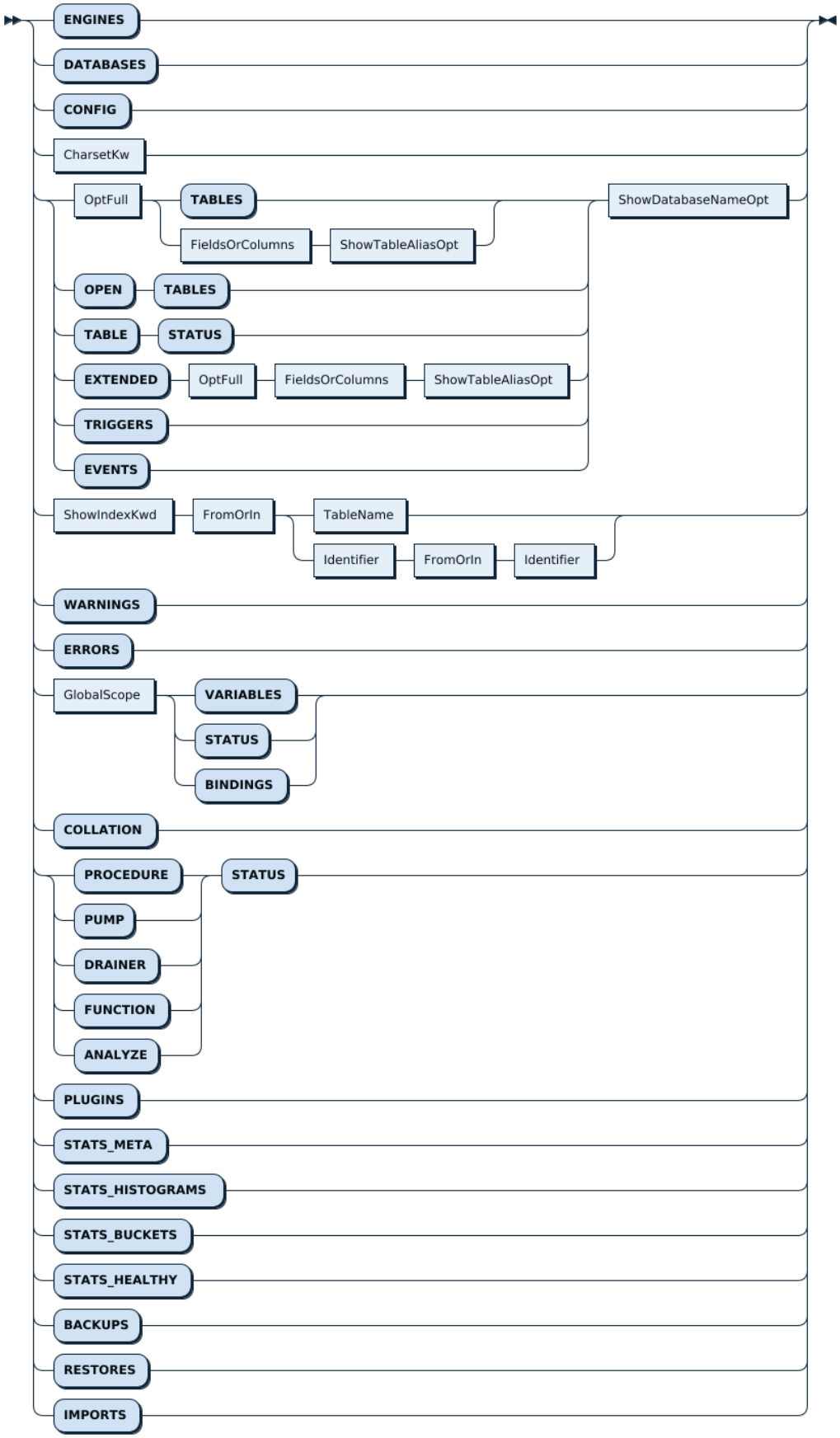


图 166: ShowTargetFilterable

ShowLikeOrWhereOpt

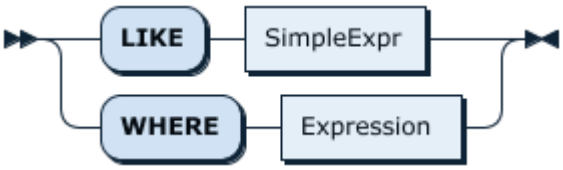


图 167: ShowLikeOrWhereOpt

4.8.2.129.2 示例

```
show stats_meta;
```

```
+---------+------------+----------------+---------------------+--------------+------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+------------+
| test    | t0         |                | 2020-05-15 16:58:00 |            0 |          0 |
| test    | t1         |                | 2020-05-15 16:58:04 |            0 |          0 |
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |          0 |
| test    | s          |                | 2020-05-22 19:46:43 |            0 |          0 |
| test    | t          |                | 2020-05-25 12:04:21 |            0 |          0 |
+---------+------------+----------------+---------------------+--------------+------------+
5 rows in set (0.00 sec)
```

```
show stats_meta where table_name = 't2';
```

```
+---------+------------+----------------+---------------------+--------------+------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+------------+
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |          0 |
+---------+------------+----------------+---------------------+--------------+------------+
1 row in set (0.00 sec)
```

4.8.2.129.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.129.4 另请参阅

- ANALYZE
- 统计信息介绍

4.8.2.130 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。

4.8.2.130.1 语法图

ShowStmt:

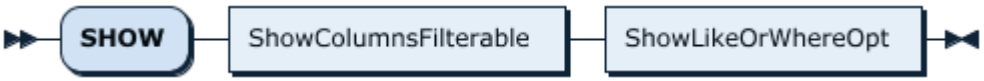


图 168: ShowStmt

ShowTargetFilterable:

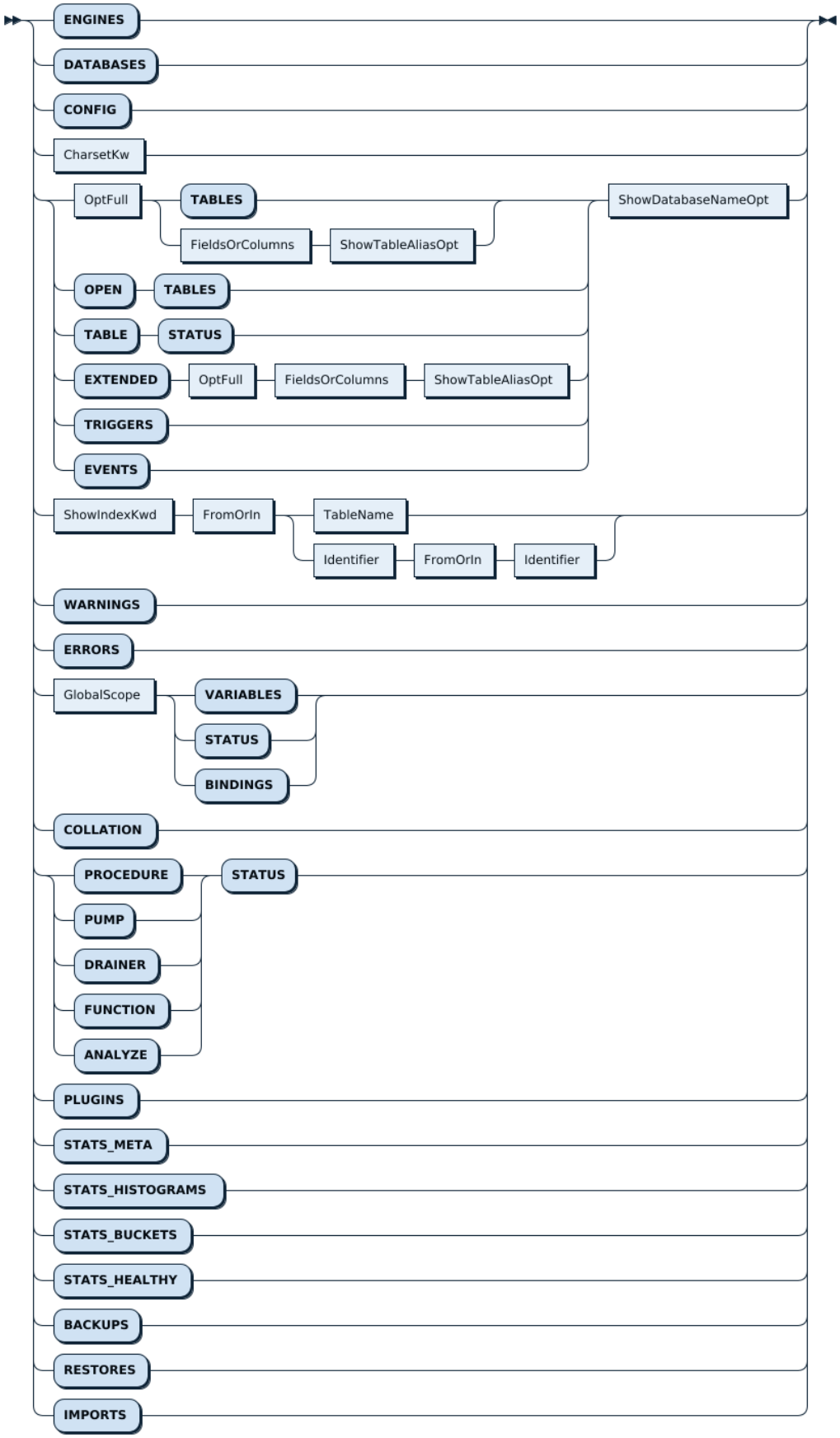


图 169: ShowTargetFilterable

GlobalScope:



图 170: GlobalScope

4.8.2.130.2 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)
```

4.8.2.130.3 MySQL 兼容性

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

4.8.2.130.4 另请参阅

- FLUSH STATUS

4.8.2.131 SHOW TABLE NEXT_ROW_ID

SHOW TABLE NEXT_ROW_ID 语句用于显示用表中某些特殊列的详情，主要包含以下几种类型：

- TiDB 创建的 AUTO_INCREMENT 类型列，即 _tidb_rowid 列
- 用户创建的 AUTO_INCREMENT 类型列
- 用户创建的AUTO_RANDOM 类型列
- 用户创建的SEQUENCE 对象信息

4.8.2.131.1 语法图

ShowTableNextRowIDStmt:



图 171: ShowTableNextRowIDStmt

TableName:

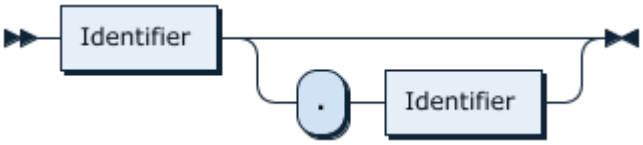


图 172: TableName

4.8.2.131.2 示例

对于新建的表，由于没有任何的Row ID 分配，NEXT_GLOBAL_ROW_ID 值为 1

```
create table t(a int);
Query OK, 0 rows affected (0.06 sec)
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------------------+
| test    | t          | _tidb_rowid |                              1 |
+---------+------------+-------------+--------------------------------+
1 row in set (0.00 sec)
```

表中写入了数据，负责写入的TiDB Server 一次性向存储层请求了 30000 个 ID 缓存起来，NEXT_GLOBAL_ROW_ID 值为 30001

```
insert into t values (), (), ();
Query OK, 3 rows affected (0.02 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------------------+
| test    | t          | _tidb_rowid |              30001 |
+---------+------------+-------------+--------------------------------+
1 row in set (0.00 sec)
```

4.8.2.131.3 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.131.4 另请参阅

- CREATE TABLE
- AUTO_RANDOM
- CREATE_SEQUENCE

4.8.2.132 SHOW TABLE REGIONS

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

4.8.2.132.1 语法

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

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

语法图

ShowTableRegionStmt:

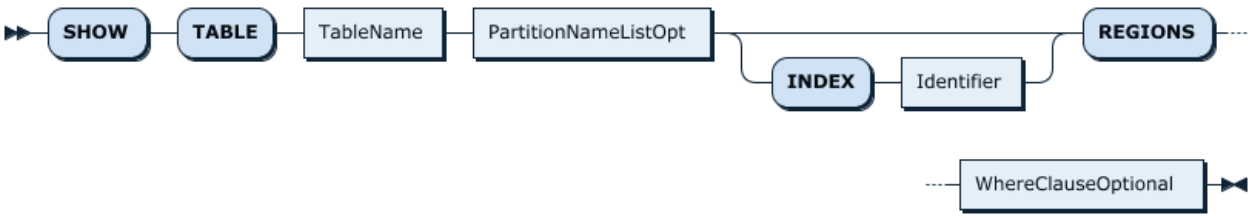


图 173: ShowTableRegionStmt

TableName:

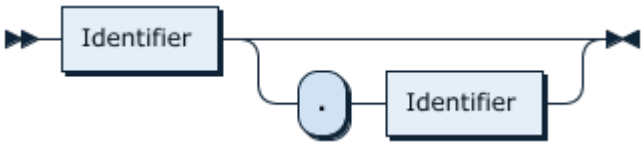


图 174: TableName

PartitionNameListOpt:

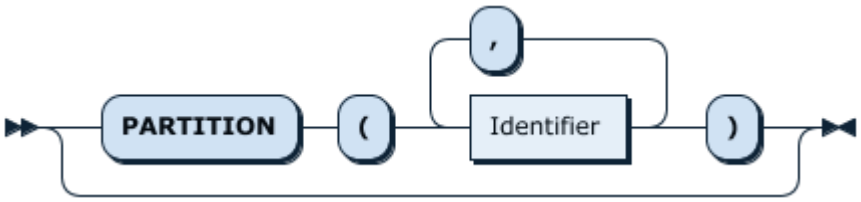


图 175: PartitionNameListOpt

WhereClauseOptional:

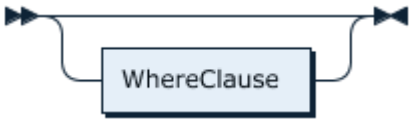


图 176: WhereClauseOptional

WhereClause:



图 177: WhereClause

`SHOW TABLE REGIONS` 会返回如下列：

- `REGION_ID`：Region 的 ID。
- `START_KEY`：Region 的 Start key。
- `END_KEY`：Region 的 End key。
- `LEADER_ID`：Region 的 Leader ID。
- `LEADER_STORE_ID`：Region leader 所在的 store (TiKV) ID。
- `PEERS`：Region 所有副本的ID。
- `SCATTERING`：Region 是否正在调度中。1 表示正在调度。
- `WRITTEN_BYTES`：估算的 Region 在 1 个心跳周期内的写入数据量大小，单位是byte。
- `READ_BYTES`：估算的 Region 在 1 个心跳周期内的读数据量大小，单位是byte。
- `APPROXIMATE_SIZE(MB)`：估算的 Region 的数据量大小，单位是 MB。

- APPROXIMATE_KEYS：估算的 Region 内 Key 的个数。
- SCHEDULING_CONSTRAINTS：Region 所在的表或者分区所关联的放置策略的规则。
- SCHEDULING_STATE：关联了放置策略的 Region 的当前调度状态。

> **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。

4.8.2.132.2 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 |
    <--> SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+-----------+--------------+--------------+-----------+-----------------+-------+------------+-----------------+--
    <-->
|        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 |
    <--> SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+--
    <--> ---------+--------------+--------------+-----------+----------------+-------+------------+--------------
    <-->
|        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 |
    <--> SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+--
    <--> ---------+--------------+--------------+-----------+----------------+---------------+------------+------
    <-->
```

```
| 102        | t_43_r        | t_43_r_20000 | 118        | 7                | 105, 118, 119 | 0
    <-->          | 0             | 0          | 1                    | 0                |
    <-->                        |                  |
| 106        | t_43_r_20000 | t_43_r_40000 | 120        | 7                | 107, 108, 120 | 0
    <-->          | 23            | 0          | 1                    | 0                |
    <-->                        |                  |
| 110        | t_43_r_40000 | t_43_r_60000 | 112        | 9                | 112, 113, 121 | 0
    <-->          | 0             | 0          | 1                    | 0                |
    <-->                        |                  |
| 114        | t_43_r_60000 | t_43_r_80000 | 122        | 7                | 115, 122, 123 | 0
    <-->          | 35            | 0          | 1                    | 0                |
    <-->                        |                  |
| 3          | t_43_r_80000 |              | 93         | 8                | 5, 73, 93     | 0
    <-->          | 0             | 0          | 1                    | 0                |
    <-->                        |                  |
| 98         | t_43_         | t_43_r       | 99         | 1                | 99, 100, 101  | 0
    <-->          | 0             | 0          | 1                    | 0                |
    <-->                        |                  |
+--
    <--> ---------+--------------+--------------+-----------+----------------+---------------+-------------+------
    <-->
6 rows in set
```

解释：

- Region 102 的 START_KEY 和 END_KEY 中，t_43 是表数据前缀和table ID，_r 是表t record 数据的前缀，索引数据的前缀是 _i，所以 Region 102 的 START_KEY 和 END_KEY 表示用来存储 [-inf, 20000) 之前的record 数据。其他 Region (106, 110, 114, 3) 的存储范围依次类推。
- Region 98 用来存储索引数据存储。表t 索引数据的起始 key 是 t_43_i，处于Region 98 的存储范围内。

查看表t 在 store 1 上的region，用where 条件过滤。

```
show table t regions where leader_store_id =1;
```

```
+-----------+-----------+---------+-----------+----------------+-------------+------------+----------------+---
    <-->
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS         | SCATTERING |
    <--> WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
    <--> SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+-----------+-----------+---------+-----------+----------------+-------------+------------+----------------+---
    <-->
| 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 | SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+-----------+-----------------------------+-----------------------------+-----------+-----------------+---------
    <->
| 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
```

4.8.2.132.3 MySQL 兼容性

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

4.8.2.132.4 另请参阅

- SPLIT REGION
- CREATE TABLE

4.8.2.133 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。

4.8.2.133.1 语法图

ShowTableStatusStmt:

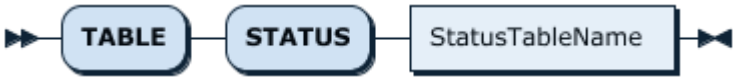


图 178: ShowTableStatusStmt

FromOrIn:

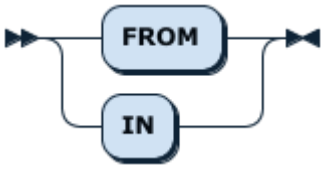


图 179: FromOrIn

StatusTableName:



图 180: StatusTableName

4.8.2.133.2 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)
```

4.8.2.133.3 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。

4.8.2.133.4 另请参阅

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

4.8.2.134 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。

4.8.2.134.1 语法图

ShowTablesStmt:



图 181: ShowTablesStmt

OptFull:



图 182: OptFull

ShowDatabaseNameOpt:



图 183: ShowDatabaseNameOpt

ShowLikeOrWhereOpt:

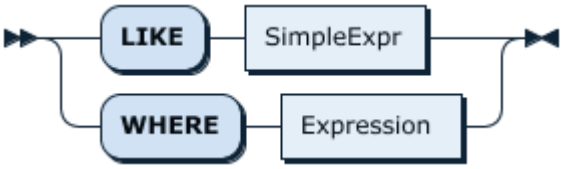


图 184: ShowLikeOrWhereOpt

4.8.2.134.2 示例

```
mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.12 sec)

mysql> CREATE VIEW v1 AS SELECT 1;
Query OK, 0 rows affected (0.10 sec)

mysql> SHOW TABLES;
+----------------------------+
| Tables_in_test |
+----------------------------+
| t1             |
| v1             |
+----------------------------+
2 rows in set (0.00 sec)

mysql> SHOW FULL TABLES;
+-----------------+--------------------+
| Tables_in_test | Table_type |
+-----------------+--------------------+

| t1              | BASE TABLE |
| v1              | VIEW       |
+-----------------+--------------------+
2 rows in set (0.00 sec)

mysql> SHOW TABLES IN mysql;
+---------------------------+
| Tables_in_mysql           |
+---------------------------+
```

```
| GLOBAL_VARIABLES         |
| bind_info                |
| columns_priv             |
| db                       |
| default_roles            |
| expr_pushdown_blacklist |
| gc_delete_range          |
| gc_delete_range_done     |
| global_priv              |
| help_topic               |
| opt_rule_blacklist       |
| role_edges               |
| stats_buckets            |
| stats_feedback           |
| stats_histograms         |
| stats_meta               |
| stats_top_n              |
| tables_priv              |
| tidb                     |
| user                     |
+                          +
20 rows in set (0.00 sec)
```

4.8.2.134.3 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。

4.8.2.134.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

4.8.2.135 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`。

4.8.2.135.1 语法图

ShowStmt:

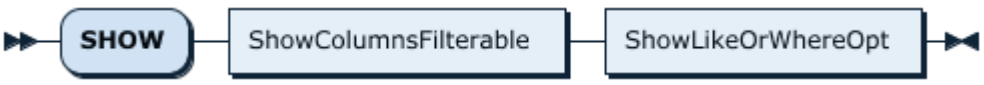


图 185: ShowStmt

ShowTargetFilterable:

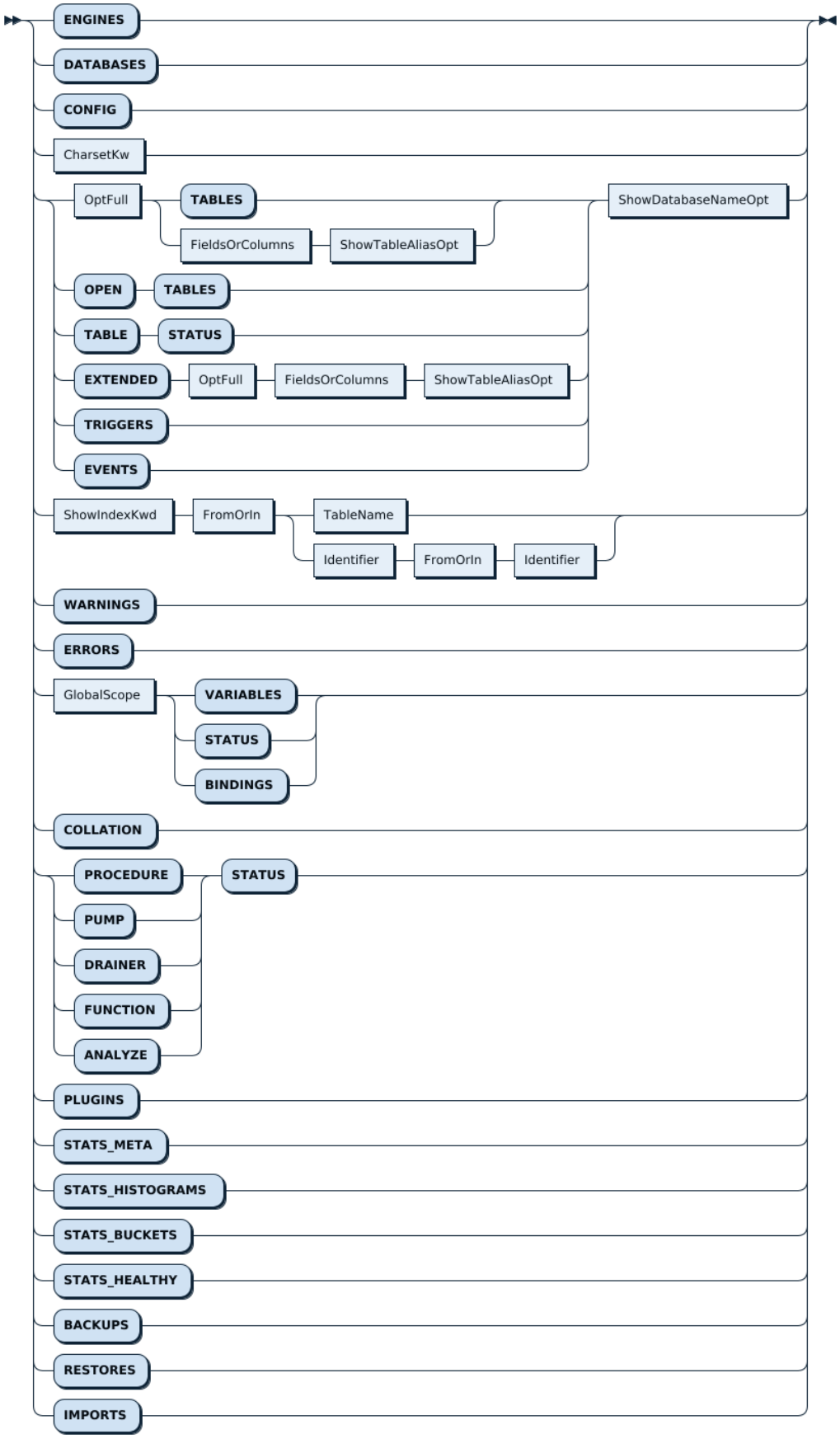


图 186: ShowTargetFilterable

GlobalScope:



图 187: GlobalScope

4.8.2.135.2 示例

查看TiDB 定义的专用系统变量，关于这些变量的含义参见系统变量和语法。

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

```
+----------------------------------------+---------------------+
| Variable_name                          | Value               |
+----------------------------------------+---------------------+
| tidb_allow_batch_cop                   | 0                   |
| tidb_allow_remove_auto_inc             | 0                   |
| tidb_auto_analyze_end_time             | 23:59 +0000         |
| tidb_auto_analyze_ratio                | 0.5                 |
| tidb_auto_analyze_start_time           | 00:00 +0000         |
| tidb_backoff_lock_fast                 | 100                 |
| tidb_backoff_weight                    | 2                   |
| tidb_batch_commit                      | 0                   |
| tidb_batch_delete                      | 0                   |
| tidb_build_stats_concurrency           | 4                   |
| tidb_capture_plan_baselines            | off                 |
| tidb_check_mb4_value_in_utf8           | 1                   |
| tidb_checksum_table_concurrency        | 4                   |
| tidb_config                            |                     |
| tidb_constraint_check_in_place         | 0                   |
| tidb_current_ts                        | 0                   |
| tidb_ddl_error_count_limit             | 512                 |
| tidb_ddl_reorg_batch_size              | 256                 |
| tidb_ddl_reorg_priority                | PRIORITY_LOW        |
| tidb_ddl_reorg_worker_cnt              | 4                   |
| tidb_disable_txn_auto_retry            | 1                   |
| tidb_distsql_scan_concurrency          | 15                  |
| tidb_dml_batch_size                    | 20000               |
| tidb_enable_cascades_planner           | 0                   |
| tidb_enable_chunk_rpc                  | 1                   |
| tidb_enable_collect_execution_info     | 1                   |
| tidb_enable_fast_analyze               | 0                   |
```

```
| tidb_enable_index_merge              | 0                    |
| tidb_enable_noop_functions           | 0                    |
| tidb_enable_radix_join               | 0                    |
| tidb_enable_slow_log                 | 1                    |
| tidb_enable_stmt_summary             | 1                    |
| tidb_enable_table_partition           | on                   |
| tidb_enable_vectorized_expression    | 1                    |
| tidb_enable_window_function          | 1                    |
| tidb_evolve_plan_baselines           | off                  |
| tidb_evolve_plan_task_end_time       | 23:59 +0000          |
| tidb_evolve_plan_task_max_time       | 600                  |
| tidb_evolve_plan_task_start_time     | 00:00 +0000          |
| tidb_expensive_query_time_threshold | 60                   |
| tidb_force_priority                  | NO_PRIORITY          |
| tidb_general_log                     | 0                    |
| tidb_hash_join_concurrency           | 5                    |
| tidb_hashagg_final_concurrency       | 4                    |
| tidb_hashagg_partial_concurrency     | 4                    |
| tidb_index_join_batch_size           | 25000                |
| tidb_index_lookup_concurrency        | 4                    |
| tidb_index_lookup_join_concurrency  | 4                    |
| tidb_index_lookup_size               | 20000                |
| tidb_index_serial_scan_concurrency  | 1                    |
| tidb_init_chunk_size                 | 32                   |
| tidb_isolation_read_engines          | tikv, tiflash, tidb |
| tidb_low_resolution_tso              | 0                    |
| tidb_max_chunk_size                  | 1024                 |
| tidb_max_delta_schema_count          | 1024                 |
| tidb_mem_quota_hashjoin              | 34359738368          |
| tidb_mem_quota_indexlookupjoin       | 34359738368          |
| tidb_mem_quota_indexlookupreader     | 34359738368          |
| tidb_mem_quota_mergejoin             | 34359738368          |
| tidb_mem_quota_nestedloopapply       | 34359738368          |
| tidb_mem_quota_query                 | 1073741824           |
| tidb_mem_quota_sort                  | 34359738368          |
| tidb_mem_quota_topn                  | 34359738368          |
| tidb_metric_query_range_duration     | 60                   |
| tidb_metric_query_step               | 60                   |
| tidb_opt_agg_push_down               | 0                    |
| tidb_opt_concurrency_factor          | 3                    |
| tidb_opt_copcpu_factor               | 3                    |
| tidb_opt_correlation_exp_factor      | 1                    |
| tidb_opt_correlation_threshold       | 0.9                  |
| tidb_opt_cpu_factor                  | 3                    |
| tidb_opt_desc_factor                 | 3                    |
```

```
| tidb_opt_disk_factor                | 1.5                 |
| tidb_opt_distinct_agg_push_down     | 0                   |
| tidb_opt_insubq_to_join_and_agg     | 1                   |
| tidb_opt_join_reorder_threshold     | 0                   |
| tidb_opt_memory_factor              | 0.001               |
| tidb_opt_network_factor             | 1                   |
| tidb_opt_scan_factor                | 1.5                 |
| tidb_opt_seek_factor                | 20                  |
| tidb_opt_write_row_id               | 0                   |
| tidb_optimizer_selectivity_level    | 0                   |
| tidb_pprof_sql_cpu                  | 0                   |
| tidb_projection_concurrency         | 4                   |
| tidb_query_log_max_len              | 4096                |
| tidb_record_plan_in_slow_log        | 1                   |
| tidb_replica_read                   | leader              |
| tidb_retry_limit                    | 10                  |
| tidb_row_format_version             | 2                   |
| tidb_scatter_region                 | 0                   |
| tidb_skip_isolation_level_check     | 0                   |
| tidb_skip_utf8_check                | 0                   |
| tidb_slow_log_threshold             | 300                 |
| tidb_slow_query_file                | tidb-slow.log       |
| tidb_snapshot                       |                     |
| tidb_stmt_summary_history_size      | 24                  |
| tidb_stmt_summary_internal_query    | 0                   |
| tidb_stmt_summary_max_sql_length    | 4096                |
| tidb_stmt_summary_max_stmt_count    | 3000                |
| tidb_stmt_summary_refresh_interval  | 1800                |
| tidb_store_limit                    | 0                   |
| tidb_txn_mode                       |                     |
| tidb_use_plan_baselines             | on                  |
| tidb_wait_split_region_finish       | 1                   |
| tidb_wait_split_region_timeout      | 300                 |
| tidb_window_concurrency             | 4                   |
+                                     +                     +
108 rows in set (0.01 sec)

 ------------------------------------- ----------------------
```

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

```
+---------------+--------------+
| Variable_name | Value  |
+---------------+--------------+
| time_zone     | SYSTEM |
+---------------+--------------+
1 row in set (0.00 sec)
```

4.8.2.135.3 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。

4.8.2.135.4 另请参阅

- `SET [GLOBAL|SESSION]`

4.8.2.136 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。

4.8.2.136.1 语法图

ShowWarningsStmt:



图 188: ShowWarningsStmt

4.8.2.136.2 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)
```

4.8.2.136.3 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。

4.8.2.136.4 另请参阅

- SHOW ERRORS

4.8.2.137 SHUTDOWN

SHUTDOWN 语句用于在TiDB 中执行停机操作。执行 SHUTDOWN 语句需要用户拥有 SHUTDOWN privilege。

4.8.2.137.1 语法图

Statement:



图 189: Statement

4.8.2.137.2 示例

```
SHUTDOWN;
```

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

4.8.2.137.3 MySQL 兼容性

> **注意：**
>
> 由于TiDB 是分布式数据库，因此TiDB 中的停机操作停止的是客户端连接的TiDB 实例，而不是整个TiDB 集群。

SHUTDOWN 语句与 MySQL 不完全兼容。如发现任何其他兼容性差异，请在GitHub 上提交issue。

4.8.2.138 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。

4.8.2.138.1 语法图

SplitRegionStmt:



图 190: SplitRegionStmt

SplitSyntaxOption:



图 191: SplitSyntaxOption

TableName:

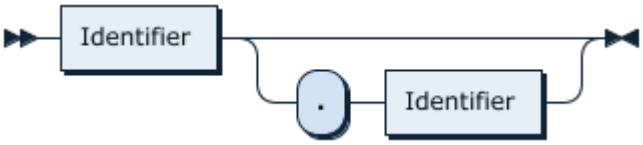


图 192: TableName

PartitionNameListOpt:

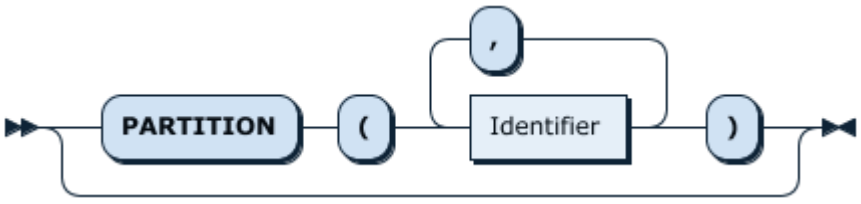


图 193: PartitionNameListOpt

SplitOption:

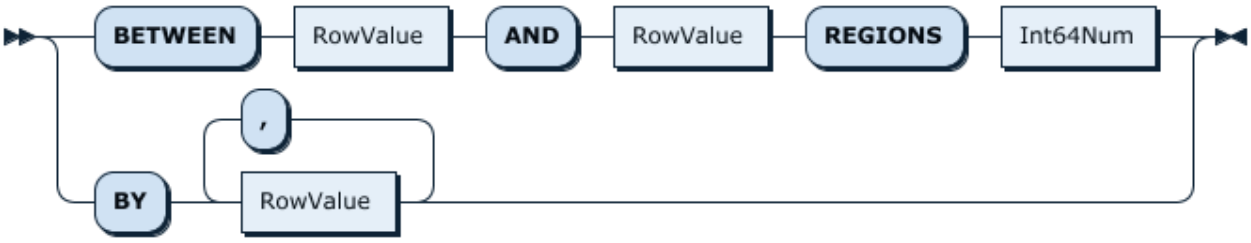


图 194: SplitOption

RowValue:



图 195: RowValue

Int64Num:

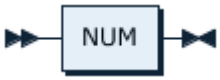


图 196: Int64Num

4.8.2.138.2 Split Region 的使用

Split Region 有 2 种不同的语法，具体如下：

- 均匀切分的语法：

```
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` 语法是通过指定数据的上、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。

- 不均匀切分的语法：

```
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)/region_num)，然后在 lower_value 和 upper_value 之间每隔 step 区间切一次，最终切出 region_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。

如果表的主键为非聚簇索引NONCLUSTERED，切分 Region 时需要用反引号` 来转义 PRIMARY 关键字。例如：

```
SPLIT TABLE t INDEX `PRIMARY` BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS
    ⟷ 16;
```

不均匀切分

索引数据也可以根据用户指定的索引值来做切分。

假如有idx4 (a,b)，其中a 列是 varchar 类型，b 列是timestamp 类型。

```
SPLIT TABLE t1 INDEX idx4 BY ("a", "2000-01-01 00:00:01"), ("b", "2019-04-17 14:26:19"), ("c", ""
    ⟷ );
```

该语句指定了 3 个值，会切分出 4 个 Region，每个 Region 的范围如下。

```
region1  [ minIndexValue                , ("a", "2000-01-01 00:00:01"))
region2  [("a", "2000-01-01 00:00:01") , ("b", "2019-04-17 14:26:19"))
region3  [("b", "2019-04-17 14:26:19") , ("c", "")                    )
region4  [("c", "")                    , maxIndexValue                )
```

Split 分区表的Region

预切分分区表的Region 在使用上和普通表一样，差别是会为每一个partition 都做相同的切分。

- 均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE t [PARTITION] [(partition_name_list...)] [INDEX index_name] BETWEEN
    <⟷ (lower_value) AND (upper_value) REGIONS region_num
```

- 不均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE table_name [PARTITION (partition_name_list...)] [INDEX index_name]
    <⟷ BY (value_list) [, (value_list)] ...
```

Split 分区表的Region 示例

1. 首先创建一个分区表。如果你要建一个 Hash 分区表，分成 2 个partition，示例语句如下：

```
create table t (a int,b int,index idx(a)) partition by hash(a) partitions 2;
```

此时建完表后会为每个partition 都单独 split 一个Region，用 SHOW TABLE REGIONS 语法查看该表的Region 如下：

```
show table t regions;
```

```
+--
    <⟷ ---------+-----------+---------+-----------+----------------+------------------+-------------+------
    <⟷
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS              |
    <⟷ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    <⟷ ---------+-----------+---------+-----------+----------------+------------------+-------------+------
    <⟷
| 1978       | t_1400_    | t_1401_ | 1979       | 4                | 1979, 1980, 1981 | 0
    <⟷           | 0              | 0           | 1                    | 0                |
| 6          | t_1401_    |         | 17         | 4                | 17, 18, 21       | 0
    <⟷           | 223            | 0           | 1                    | 0                |
+--
    <⟷ ---------+-----------+---------+-----------+----------------+------------------+-------------+------
    <⟷
```

2. 用 SPLIT 语法为每个partition 切分 Region。如果你要将各个partition 的 [0,10000] 范围内的数据切分成 4 个 Region，示例语句如下：

```
split partition table t between (0) and (10000) regions 4;
```

其中，0 和 10000 分别代表你想要打散的热点数据对应的上、下边界的 row_id。

> 注意：
> 此示例仅适用于数据热点均匀分布的场景。如果热点数据在你指定的数据范围内是不均匀分布的，请参考Split 分区表的Region 中不均匀切分的语法。

3. 用 SHOW TABLE REGIONS 语法查看该表的Region。如下会发现该表现在一共有 10 个 Region，每个partition 分别有 5 个 Region，其中 4 个 Region 是表的行数据，1 个 Region 是表的索引数据。

```
show table t regions;
```

```
+--
    <-> ---------+--------------+--------------+-----------+----------------+-----------------+---------
    <->
| REGION_ID | START_KEY      | END_KEY        | LEADER_ID | LEADER_STORE_ID | PEERS
    <->            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    <-> APPROXIMATE_KEYS |
+--
    <-> ---------+--------------+--------------+-----------+----------------+-----------------+---------
    <->
| 1998       | t_1400_r       | t_1400_r_2500  | 2001       | 5                | 2000, 2001, 2015
    <-> | 0          | 132            | 0          | 1                      | 0
    <->                |
| 2006       | t_1400_r_2500  | t_1400_r_5000  | 2016       | 1                | 2007, 2016, 2017
    <-> | 0          | 35             | 0          | 1                      | 0
    <->                |
| 2010       | t_1400_r_5000  | t_1400_r_7500  | 2012       | 2                | 2011, 2012, 2013
    <-> | 0          | 35             | 0          | 1                      | 0
    <->                |
| 1978       | t_1400_r_7500  | t_1401_        | 1979       | 4                | 1979, 1980, 1981
    <-> | 0          | 621            | 0          | 1                      | 0
    <->                |
| 1982       | t_1400_        | t_1400_r       | 2014       | 3                | 1983, 1984, 2014
    <-> | 0          | 35             | 0          | 1                      | 0
    <->                |
| 1990       | t_1401_r       | t_1401_r_2500  | 1992       | 2                | 1991, 1992, 2020
    <-> | 0          | 120            | 0          | 1                      | 0
    <->                |
| 1994       | t_1401_r_2500  | t_1401_r_5000  | 1997       | 5                | 1996, 1997, 2021
    <-> | 0          | 129            | 0          | 1                      | 0
    <->                |
| 2002       | t_1401_r_5000  | t_1401_r_7500  | 2003       | 4                | 2003, 2023, 2022
    <-> | 0          | 141            | 0          | 1                      | 0
    <->                |
| 6          | t_1401_r_7500  |                | 17         | 4                | 17, 18, 21
    <->       | 0           | 601            | 0          | 1                       | 0
    <->                |
| 1986       | t_1401_        | t_1401_r       | 1989       | 5                | 1989, 2018, 2019
    <-> | 0          | 123            | 0          | 1                      | 0
    <->                |
+--
    <-> ---------+--------------+--------------+-----------+----------------+-----------------+---------
```

```
    <->
```

4. 如果你要给每个分区的索引切分Region，如将索引 idx 的 [1000,10000] 范围切分成 2 个 Region，示例语句如下：

```
split partition table t index idx between (1000) and (10000) regions 2;
```

Split 单个分区的Region 示例

可以单独指定要切分的partition，示例如下：

1. 首先创建一个分区表。如果你要建一个Range 分区表，分成 3 个partition，示例语句如下：

```
create table t ( a int, b int, index idx(b)) partition by range( a )
    ( partition p1 values less than (10000),
    partition p2 values less than (20000),
    partition p3 values less than (MAXVALUE) );
```

2. 如果你要将 p1 分区的 [0,10000] 范围内的数据预切分 2 个Region，示例语句如下：

```
split partition table t partition (p1) between (0) and (10000) regions 2;
```

3. 如果你要将 p2 分区的 [10000,20000] 范围内的数据预切分 2 个Region，示例语句如下：

```
split partition table t partition (p2) between (10000) and (20000) regions 2;
```

4. 用 SHOW TABLE REGIONS 语法查看该表的Region 如下：

```
show table t regions;
```

```
+--
    <-> ---------+---------------+---------------+-----------+-----------------+------------------+-------
    <->
| REGION_ID | START_KEY       | END_KEY         | LEADER_ID | LEADER_STORE_ID | PEERS
    <->           | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    <-> APPROXIMATE_KEYS |
+--
    <-> ---------+---------------+---------------+-----------+-----------------+------------------+-------
    <->
| 2040      | t_1406_         | t_1406_r_5000  | 2045      | 3               | 2043, 2045,
    <-> 2044 | 0           | 0              | 0          | 1                 | 0
    <->                |
| 2032      | t_1406_r_5000  | t_1407_         | 2033      | 4               | 2033, 2034,
    <-> 2035 | 0           | 0              | 0          | 1                 | 0
    <->                |
| 2046      | t_1407_         | t_1407_r_15000 | 2048      | 2               | 2047, 2048,
    <-> 2050 | 0           | 35             | 0          | 1                 | 0
    <->                |
```

```
| 2036       | t_1407_r_15000 | t_1408_          | 2037       | 4                 | 2037, 2038,
    <-> 2039 | 0              | 0                | 0          | 1                   | 0
    <->                  |
| 6          | t_1408_          |                  | 17         | 4                 | 17, 18, 21
    <->        | 0            | 214              | 0          | 1                   | 0
    <->                  |
+--
    <> ---------+----------------+----------------+------------+-----------------+-------------------+-------
    <>
```

5. 如果你要将 p1 和 p2 分区的索引 idx 的 [0,20000] 范围预切分 2 个Region，示例语句如下：

```
split partition table t partition (p1,p2) index idx between (0) and (20000) regions 2;
```

4.8.2.138.3 pre_split_regions

使用带有 SHARD_ROW_ID_BITS 的表时，如果希望建表时就均匀切分Region，可以考虑配合 PRE_SPLIT_REGIONS 一起使用，用来在建表成功后就开始预均匀切分 2^(PRE_SPLIT_REGIONS) 个Region。

> **注意：**
>
> PRE_SPLIT_REGIONS 必须小于等于 SHARD_ROW_ID_BITS。

以下全局变量会影响 PRE_SPLIT_REGIONS 的行为，需要特别注意：

- tidb_scatter_region：该变量用于控制建表完成后是否等待预切分和打散 Region 完成后再返回结果。如果建表后有大批量写入，需要设置该变量值为 1，表示等待所有Region 都切分和打散完成后再返回结果给客户端。否则未打散完成就进行写入会对写入性能影响有较大的影响。

pre_split_regions 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  )
```

4.8.2.138.4 注意事项

Split Region 语句切分的Region 会受到 PD 中Region merge 调度的控制，需要动态修改Region merge 相关的配置项，避免新切分的Region 不久后又被 PD 重新合并的情况。

4.8.2.138.5 MySQL 兼容性

该语句是TiDB 对 MySQL 语法的扩展。

4.8.2.138.6 另请参阅

- SHOW TABLE REGIONS
- Session 变量:tidb_scatter_region,tidb_wait_split_region_finish 和tidb_wait_split_region_timeout <↩>.

4.8.2.139 START TRANSACTION

START TRANSACTION 语句用于在TiDB 内部启动新事务。它类似于语句 BEGIN。

在没有 START TRANSACTION 语句的情况下，每个语句都会在各自的事务中自动提交，这样可确保 MySQL 兼容性。

4.8.2.139.1 语法图

BeginTransactionStmt:

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( ( 'WITH' 'TIMESTAMP' 'BOUND'
    <⟶ TimestampBound )? | AsOfClause ) ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' | 'WITH' 'CAUSAL' '
    <⟶ CONSISTENCY' 'ONLY' )?

AsOfClause ::=
    ( 'AS' 'OF' 'TIMESTAMP' Expression)
```

4.8.2.139.2 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)
```

4.8.2.139.3 MySQL 兼容性

- 执行 START TRANSACTION 在 TiDB 中开启事务并立即生成快照。而在 MySQL 中，执行 START TRANSACTION 会开启事务但不会立即生成快照。TiDB 中的 START TRANSACTION 等同于 MySQL 中的 START TRANSACTION <—> WITH CONSISTENT SNAPSHOT。
- 为与 MySQL 兼容，TiDB 会解析 START TRANSACTION READ ONLY 语句，但解析后TiDB 仍允许写入操作。

4.8.2.139.4 另请参阅

- COMMIT
- ROLLBACK
- BEGIN
- START TRANSACTION WITH CAUSAL CONSISTENCY ONLY

4.8.2.140 TABLE

当不需要聚合或复杂的过滤操作时，可以用 TABLE 语句代替 SELECT * FROM。

4.8.2.140.1 语法图

```
TableStmt ::=
    "TABLE" Table ( "ORDER BY" Column )? ( "LIMIT" NUM )?
```

4.8.2.140.2 示例

创建表 t1：

```
CREATE TABLE t1(id INT PRIMARY KEY);
```

插入一些数据：

```
INSERT INTO t1 VALUES (1),(2),(3);
```

查看表 t1 的数据：

```
TABLE t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
+----+
3 rows in set (0.01 sec)
```

查询 t1 表，并按 id 字段倒序排列结果：

```
TABLE t1 ORDER BY id DESC;
```

```
+----+
| id |
+----+
|  3 |
|  2 |
|  1 |
+----+
3 rows in set (0.01 sec)
```

查询表 t1 中的前 1 条记录：

```
TABLE t1 LIMIT 1;
```

```
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

4.8.2.140.3 MySQL 兼容性

TABLE 语句是从 MySQL 8.0.19 开始引入的。

4.8.2.140.4 另请参阅

- SELECT
- TABLE statements in MySQL

4.8.2.141 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。

4.8.2.141.1 语法图

TraceStmt:

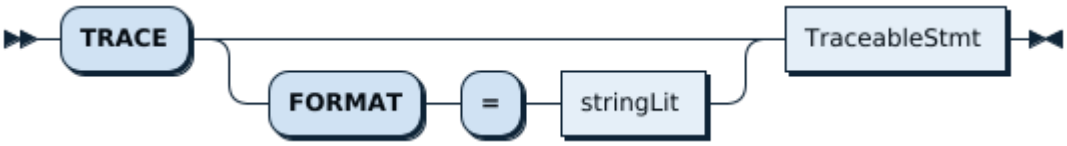


图 197: TraceStmt

TraceableStmt:

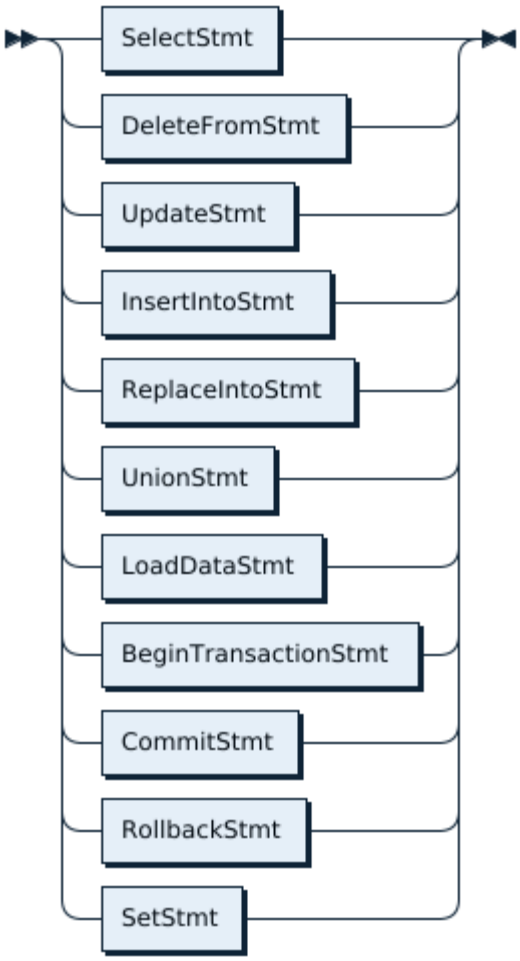


图 198: TraceableStmt

4.8.2.141.2 示例

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

```
+-----------------------------------------+-----------------+----------------------+
| operation                               | startTS         | duration    |
```

```
+-----------------------------------------+-----------------+-----------------------+
| trace                                   | 17:03:31.938237 | 886.086µs             |
|   ├─session.Execute                     | 17:03:31.938247 | 507.812µs             |
|   │ ├─session.ParseSQL                  | 17:03:31.938254 | 22.504µs              |
|   │ ├─executor.Compile                  | 17:03:31.938321 | 278.931µs             |
|   │ │ └─session.getTxnFuture            | 17:03:31.938337 | 1.515µs               |
|   │ └─session.runStmt                   | 17:03:31.938613 | 109.578µs             |
|   │   ├─TableReaderExecutor.Open        | 17:03:31.938645 | 50.657µs              |
|   │   │ └─distsql.Select                | 17:03:31.938666 | 21.066µs              |
|   │   │   └─RPCClient.SendRequest       | 17:03:31.938799 | 158.411µs             |
|   │   └─session.CommitTxn               | 17:03:31.938705 | 12.06µs               |
|   │     └─session.doCommitWitRetry      | 17:03:31.938709 | 2.437µs               |
|   ├─*executor.TableReaderExecutor.Next  | 17:03:31.938781 | 224.327µs             |
|   └─*executor.TableReaderExecutor.Next  | 17:03:31.939019 | 6.266µs               |
+-----------------------------------------+-----------------+-----------------------+
13 rows in set (0.00 sec)
```

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

可将JSON 格式的跟踪文件粘贴到跟踪查看器中。查看器可通过TiDB 状态端口访问：

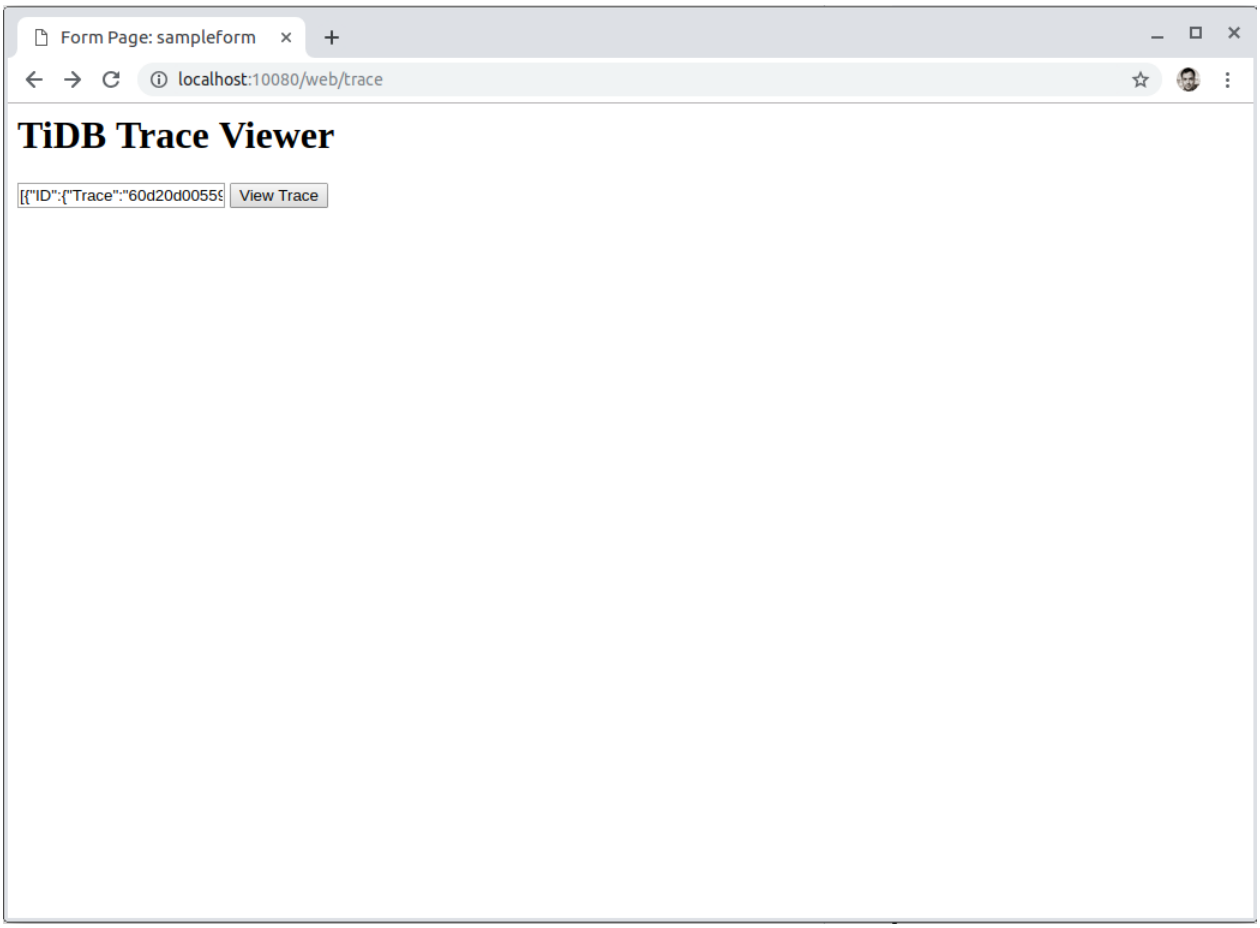


图 199: TiDB Trace Viewer-1

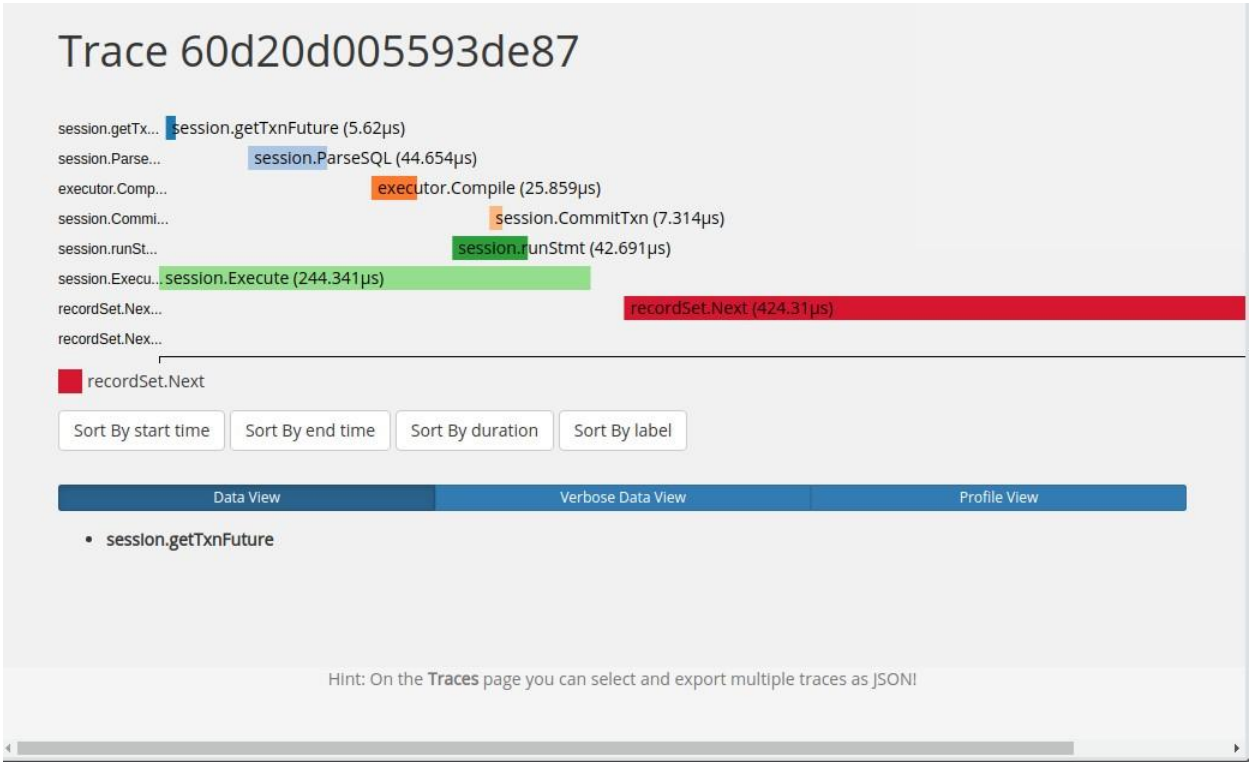


图 200: TiDB Trace Viewer-2

4.8.2.141.3 MySQL 兼容性

TRACE 语句是TiDB 对 MySQL 语法的扩展。

4.8.2.141.4 另请参阅

- EXPLAIN ANALYZE

4.8.2.142 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。

4.8.2.142.1 语法图

TruncateTableStmt:



图 201: TruncateTableStmt

OptTable:



图 202: OptTable

TableName:

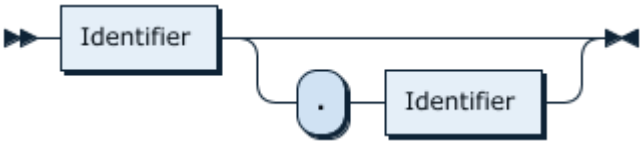


图 203: TableName

4.8.2.142.2 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)
```

4.8.2.142.3 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。

4.8.2.142.4 另请参阅

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

4.8.2.143 UNLOCK STATS

UNLOCK STATS 语句用于解锁统计信息被锁定的表。

> **警告：**
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

4.8.2.143.1 语法图

```
UnlockStatsStmt ::=
    'UNLOCK' 'STATS' TableNameList

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?
```

4.8.2.143.2 示例

创建表t，插入一些数据，在未锁定表t 的统计信息的情况下成功执行 ANALYZE 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
+       +      +
| Level | Code | Message                                                
```

解锁表t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+       +      +
 --------+-------+-----------------------------------------------------------------------+
| Level | Code | Message                                                
```

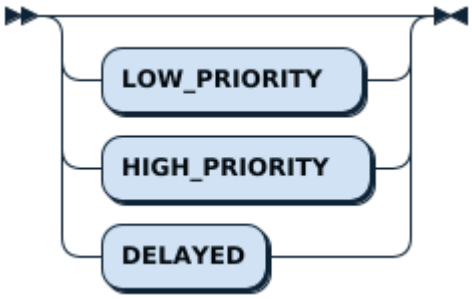


图 205: PriorityOpt

TableRef:

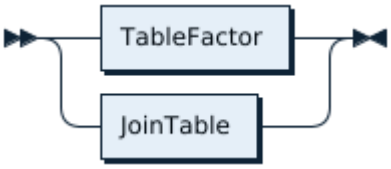


图 206: TableRef

TableRefs:



图 207: TableRefs

AssignmentList:

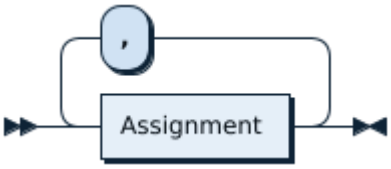


图 208: AssignmentList

WhereClauseOptional:

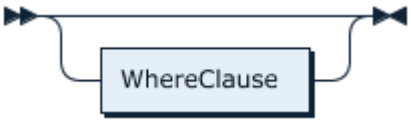


图 209: WhereClauseOptional

4.8.2.144.2 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)
```

4.8.2.144.3 MySQL 兼容性

在计算表达式中的列时，TiDB 总使用原始的值。例如：

```
CREATE TABLE t (a int, b int);
INSERT INTO t VALUES (1,2);
UPDATE t SET a = a+1,b=a;
```

在 MySQL 中，b 列的值会被更新成 2，因为b 列被设置为与a 列相同，而a（最初是 1）在同一条语句中被更新成了 2。

TiDB 遵守标准的 SQL 行为，这里将b 列值更新成 1。

4.8.2.144.4 另请参阅

- INSERT
- SELECT
- DELETE
- REPLACE

4.8.2.145 USE

USE 语句可为用户会话选择当前数据库。

4.8.2.145.1 语法图

UseStmt:



图 210: UseStmt

DBName:



图 211: DBName

4.8.2.145.2 示例

```
USE mysql;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
SHOW TABLES;
```

```
+----------------------------------------+
| Tables_in_mysql                        |
+----------------------------------------+
| GLOBAL_VARIABLES                       |
```

```
| bind_info                |
| columns_priv             |
| db                       |
| default_roles            |
| expr_pushdown_blacklist |
| gc_delete_range          |
| gc_delete_range_done     |
| global_priv              |
| help_topic               |
| opt_rule_blacklist       |
| role_edges               |
| stats_buckets            |
| stats_feedback           |
| stats_histograms         |
| stats_meta               |
| stats_top_n              |
| tables_priv              |
| tidb                     |
| user                     |
+------------------------------------------+
20 rows in set (0.01 sec)
```

```
CREATE DATABASE newtest;
```

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

```
USE newtest;
```

```
Database changed
```

```
SHOW TABLES;
```

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

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

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

```
SHOW TABLES;
```

```
+-------------------------------+
| Tables_in_newtest |
+-------------------------------+
| t1                |
+-------------------------------+
1 row in set (0.00 sec)
```

4.8.2.145.3 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。

4.8.2.145.4 另请参阅

- CREATE DATABASE
- SHOW TABLES

4.8.2.146 WITH

公共表表达式(CTE)是一个临时的中间结果集，能够在 SQL 语句中引用多次，提高 SQL 语句的可读性与执行效率。在TiDB 中可以通过 `WITH` 语句使用公共表表达式。

4.8.2.146.1 语法图

WithClause:

```
WithClause ::=
        "WITH" WithList
|       "WITH" recursive WithList
```

WithList:

```
WithList ::=
        WithList ',' CommonTableExpr
|       CommonTableExpr
```

CommonTableExpr:

```
CommonTableExpr ::=
        Identifier IdentListWithParenOpt "AS" SubSelect
```

IdentListWithParenOpt:

```
IdentListWithParenOpt ::=
( '(' IdentList ')' )?
```

4.8.2.146.2 示例

非递归的 CTE：

```
WITH CTE AS (SELECT 1, 2) SELECT * FROM cte t1, cte t2;
```

```
+---+---+---+---+
| 1 | 2 | 1 | 2 |
+---+---+---+---+
| 1 | 2 | 1 | 2 |
```

```
+---+---+---+---+
1 row in set (0.00 sec)
```

递归的 CTE：

```
WITH RECURSIVE cte(a) AS (SELECT 1 UNION SELECT a+1 FROM cte WHERE a < 5) SELECT * FROM cte;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

4.8.2.146.3 MySQL 兼容性

- 在严格模式下，当递归部分算出的数据长度超过初始部分的数据长度时，TiDB 会返回警告，而 MySQL 会返回错误。在非严格模式下，TiDB 与 MySQL 行为一致。
- 递归 CTE 所使用的数据类型由初始部分决定。初始部分的数据类型在某些情况（例如函数）下与 MySQL 并不完全一致。
- 多个 UNION / UNION ALL 情况下，MySQL 不允许 UNION 后面加 UNION ALL，TiDB 允许。
- 如果 CTE 的定义存在问题，TiDB 会报错，而 MySQL 在未引用的情况下不报错。

4.8.2.146.4 另请参阅

- SELECT
- INSERT
- DELETE
- UPDATE
- REPLACE

4.8.3 数据类型

4.8.3.1 数据类型概述

TiDB 支持除空间类型(SPATIAL)之外的所有 MySQL 数据类型，包括数值型类型、字符串类型、时间和日期类型、JSON 类型。

数据类型定义一般为 `T(M[, D])`，其中：

- T 表示具体的类型。

- M 在整数类型中表示最大显示长度；在浮点数或者定点数中表示精度；在字符类型中表示最大长度。M 的最大值取决于具体的类型。
- D 表示浮点数、定点数的小数位长度。
- fsp 在时间和日期类型里的 TIME、DATETIME 以及 TIMESTAMP 中表示秒的精度，其取值范围是 0 到 6。值为 0 表示没有小数部分。如果省略，则默认精度为 0。

4.8.3.2 数据类型的默认值

在一个数据类型描述中的DEFAULT value 段描述了一个列的默认值。这个默认值必须是常量，不可以是一个函数或者是表达式。但是对于时间类型，可以例外的使用 NOW、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMP 等函数作为 DATETIME 或者 TIMESTAMP 的默认值。

BLOB、TEXT 以及 JSON 不可以设置默认值。

如果一个列的定义中没有 DEFAULT 的设置。TiDB 按照如下的规则决定：

- 如果该类型可以使用 NULL 作为值，那么这个列会在定义时添加隐式的默认值设置 DEFAULT NULL。
- 如果该类型无法使用 NULL 作为值，那么这个列在定义时不会添加隐式的默认值设置。

对于一个设置了 NOT NULL 但是没有显式设置 DEFAULT 的列，当 INSERT、REPLACE 没有涉及到该列的值时，TiDB 根据当时的 SQL_MODE 进行不同的行为：

- 如果此时是 strict sql mode，在事务中的语句会导致事务失败并回滚，非事务中的语句会直接报错。
- 如果此时不是 strict sql mode，TiDB 会为这列赋值为列数据类型的隐式默认值。

此时隐式默认值的设置按照如下规则：

- 对于数值类型，它们的默认值是 0。当有 AUTO_INCREMENT 参数时，默认值会按照增量情况赋予正确的值。
- 对于除了时间戳外的日期时间类型，默认值会是该类型的“零值”。时间戳类型的默认值会是当前的时间。
- 对于除枚举以外的字符串类型，默认值会是空字符串。对于枚举类型，默认值是枚举中的第一个值。

4.8.3.3 数值类型

TiDB 支持 MySQL 所有的数值类型，按照精度可以分为：

- 整数类型（精确值）
- 浮点类型（近似值）
- 定点类型（精确值）

4.8.3.3.1 整数类型

TiDB 支持 MySQL 所有的整数类型，包括 INTEGER/INT、TINYINT、SMALLINT、MEDIUMINT 以及 BIGINT，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	类型显示宽度，可选
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识，但是没有做补零的操作

类型定义

BIT 类型

比特值类型。M 表示比特位的长度，取值范围从 1 到 64，其默认值是 1。

```
BIT[(M)]
```

BOOLEAN 类型

布尔类型，别名为 BOOL，和 TINYINT(1) 等价。零值被认为是 False，非零值认为是 True。在 TiDB 内部，True 存储为 1，False 存储为 0。

```
BOOLEAN
```

TINYINT 类型

TINYINT 类型。有符号数的范围是 [-128, 127]。无符号数的范围是 [0, 255]。

```
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
```

SMALLINT 类型

SMALLINT 类型。有符号数的范围是 [-32768, 32767]。无符号数的范围是 [0, 65535]。

```
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
```

MEDIUMINT 类型

MEDIUMINT 类型。有符号数的范围是 [-8388608, 8388607]。无符号数的范围是 [0, 16777215]。

```
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
```

INTEGER 类型

INTEGER 类型，别名INT。有符号数的范围是[-2147483648, 2147483647]。无符号数的范围是[0, 4294967295]。

```
INT[(M)] [UNSIGNED] [ZEROFILL]
```

或者：

```
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

BIGINT 类型

BIGINT 类型。有符号数的范围是 [-9223372036854775808, 9223372036854775807]。无符号数的范围是 [0, <⟷ 18446744073709551615]。

```
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
```

存储空间以及取值范围

每种类型对存储空间的需求以及最大/最小值如下表所示：

类型	存储空间	最小值(有符号/无符号)	最大值(有符号/无符号)
TINYINT	1	-128 / 0	127 / 255
SMALLINT	2	-32768 / 0	32767 / 65535
MEDIUMINT	3	-8388608 / 0	8388607 / 16777215
INT	4	-2147483648 / 0	2147483647 / 4294967295
BIGINT	8	-9223372036854775808 / 0	9223372036854775807 / 18446744073709551615

4.8.3.3.2 浮点类型

TiDB 支持 MySQL 所有的浮点类型，包括 FLOAT、 DOUBLE，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

类型定义

FLOAT 类型

单精度浮点数。允许的值范围为-2^128~+2^128，也即-3.402823466E+38 到 -1.175494351E-38、 0 和 1.175494351E-38 到 3.402823466E+38。这些是基于IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

FLOAT(p) 类型中，p 表示精度（以位数表示）。只使用该值来确定是否结果列的数据类型为 FLOAT 或 DOUBLE。如果p 为从 0 到 24，数据类型变为没有M 或 D 值的 FLOAT。如果 p 为从 25 到 53，数据类型变为没有M 或 D 值的 DOUBLE。结果列范围与本节前面描述的单精度 FLOAT 或双精度 DOUBLE 数据类型相同。

```
FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]
FLOAT(p) [UNSIGNED] [ZEROFILL]
```

注意：

- 与在 MySQL 中一样，FLOAT 数据类型用于存储近似值。对于类似货币的精确值，建议使用 DECIMAL 类型。

- 在 TiDB 中，FLOAT 数据类型默认的精度是 8 位，这与 MySQL 不同。在 MySQL 中，FLOAT 数据类型默认的精度是 6 位。例如，同时往 MySQL 和 TiDB 中类型为 FLOAT 的列中插入数据 123456789 和 1.23456789。在 MySQL 中查询对应的值，将会得到结果 123457000 和 1.23457。而在 TiDB 中查询对应的值，将会得到 123456790 和 1.2345679。

DOUBLE 类型

双精度浮点数，别名为DOUBLE PRECISION。允许的值范围为：-2^1024 ~ +2^1024，也即是-1.7976931348623157E+308 到 -2.2250738585072014E-308、0 和 2.2250738585072014E-308 到 1.7976931348623157E+308。这些是基于IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

```
DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]
DOUBLE PRECISION [(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL]
```

警告：

与在 MySQL 中一样，DOUBLE 数据类型存储近似值。对于货币之类的精确值，建议使用 DECIMAL 类型。

注意：

当TiDB 将用科学计数法表示的双精度浮点数转换到 CHAR 类型时，其结果在显示上与 MySQL 不一致，详情参见Cast 函数和操作符。

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

4.8.3.3.3 定点类型

TiDB 支持 MySQL 所有的定点类型，包括 DECIMAL、NUMERIC，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

类型定义

DECIMAL 类型

定点数，别名为 NUMERIC。M 是小数位数（精度）的总数，D 是小数点（标度）后面的位数。小数点和-（负数）符号不包括在M 中。如果D 是 0，则值没有小数点或分数部分。如果D 被省略，默认是 0。如果M 被省略，默认是 10。

```
DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL]
```

4.8.3.4 日期和时间类型

TiDB 支持 MySQL 所有的日期和时间类型，包括DATE、TIME、DATETIME、TIMESTAMP 以及YEAR。完整信息可以参考 MySQL 中的时间和日期类型。

每种类型都有有效值范围，值为 0 表示无效值。此外，TIMESTAMP 和 DATETIME 类型能自动生成新的时间值。

关于日期和时间值类型，需要注意：

- 日期部分必须是“年-月-日” 的格式（例如 1998-09-04），而不是 “月-日-年” 或“日-月-年” 的格式。
- 如果日期的年份部分是 2 位数，TiDB 会根据年份为两位数的具体规则进行转换。
- 如果格式必须是数值类型，TiDB 会自动将日期或时间值转换为数值类型。例如：

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------+----------------+---------------------------------+
| NOW()               | NOW()+0        | NOW(3)+0                        |
+---------------------+----------------+---------------------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------+----------------+---------------------------------+
```

- TiDB 可以自动将无效值转换同一类型的零值。是否进行转换取决于 SQL 模式的设置。比如：

```
show create table t1;
```

```
+--
    <-->-----+----------------------------------------------------------------------------------------------------------------------------------
    <-->
```

```
| Table | Create Table
    <->
    <-> |
+--
    <->-----+----------------------
```

```
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）。

4.8.3.4.1 类型定义

DATE 类型

DATE 类型只包含日期部分，不包含时间部分。DATE 类型的格式为 YYYY-MM-DD，支持的范围是 0000-01-01 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 0000-01-01 <⟶ 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）。

4.8.3.4.2 自动初始化和更新 TIMESTAMP 或 DATETIME

带有 TIMESTAMP 或 DATETIME 数据类型的列可以自动初始化为或更新为当前时间。

对于表中任何带有 TIMESTAMP 或 DATETIME 数据类型的列，你可以设置默认值，或自动更新为当前时间戳。

在定义列的时候，TIMESTAMP 和DATETIME 可以通过DEFAULT CURRENT_TIMESTAMP 和ON UPDATE CURRENT_TIMESTAMP <⟶ 来设置。DEFAULT 也可以设置为一个特定的值，例如 DEFAULT 0 或 DEFAULT '2000-01-01 00:00:00'。

```
CREATE TABLE t1 (
    ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    dt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

除非指定 DATETIME 的值为 NOT NULL，否则默认 DATETIME 的值为 NULL。指定 DATETIME 的值为 NOT NULL 时，如果没有设置默认值，则默认值为 0。

```
CREATE TABLE t1 (
    dt1 DATETIME ON UPDATE CURRENT_TIMESTAMP,          -- default NULL
    dt2 DATETIME NOT NULL ON UPDATE CURRENT_TIMESTAMP -- default 0
);
```

4.8.3.4.3 时间值的小数部分

DATETIME 和 TIMESTAMP 值最多可以有 6 位小数，精确到微秒。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)
```

4.8.3.4.4 日期和时间类型的转换

在日期和时间类型之间进行转换时，有些转换可能会导致信息丢失。例如，DATE、DATETIME 和 TIMESTAMP 都有各自的有效值范围。TIMESTAMP 不能早于 UTC 时间的 1970 年，也不能晚于 UTC 时间的 2038-01-19 03:14:07。根据这个规则，1968-01-01 对于 DATE 或 DATETIME 是有效的，但当 1968-01-01 转换为 TIMESTAMP 时，就会变成 0。

DATE 的转换：

- 当 DATE 转换为 DATETIME 或 TIMESTAMP 时，会添加时间部分 00:00:00，因为 DATE 不包含任何时间信息。
- 当 DATE 转换为 TIME 时，结果是 00:00:00。

DATETIME 或 TIMESTAMP 的转换：

- 当 DATETIME 或 TIMESTAMP 转换为 DATE 时，时间和小数部分将被舍弃。例如，1999-12-31 23:59:59.499 被转换为 1999-12-31。
- 当 DATETIME 或 TIMESTAMP 转换为 TIME 时，日期部分被舍弃，因为 TIME 不包含任何日期信息。

如果要将 TIME 转换为其他时间和日期格式，日期部分会自动指定为 CURRENT_DATE()。最终的转换结果是由 TIME 和 CURRENT_DATE() 组成的日期。也就是说，如果 TIME 的值超出了 00:00:00 到 23:59:59 的范围，那么转换后的日期部分并不表示当前的日期。

当 TIME 转换为 DATE 时，转换过程类似，时间部分被舍弃。

使用 CAST() 函数可以显式地将值转换为 DATE 类型。例如：

```
date_col = CAST(datetime_col AS DATE)
```

将 TIME 和 DATETIME 转换为数字格式。例如：

```
SELECT CURTIME(), CURTIME()+0, CURTIME(3)+0;
```

```
+-----------|-------------|-------------------------+
| CURTIME() | CURTIME()+0 | CURTIME(3)+0 |
+-----------|-------------|-------------------------+
| 09:28:00  |       92800 |     92800.887 |
+-----------|-------------|-------------------------+
```

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------|----------------|---------------------------------+
| NOW()               | NOW()+0        | NOW(3)+0                        |
+---------------------|----------------|---------------------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------|----------------|---------------------------------+
```

4.8.3.4.5 年份为两位数

如果日期中包含年份为两位数，这个年份是有歧义的，并不显式地表示实际年份。

对于 DATETIME、DATE 和 TIMESTAMP 类型，TiDB 使用如下规则来消除歧义。

- 将 01 至 69 之间的值转换为 2001 至 2069 之间的值。
- 将 70 至 99 之间的值转化为 1970 至 1999 之间的值。

上述规则也适用于 YEAR 类型，但有一个例外。将数字 00 插入到 YEAR(4) 中时，结果是 0000 而不是 2000。

如果想让结果是 2000，需要指定值为 2000。

对于 MIN() 和 MAX() 等函数，年份为两位数时可能会得到错误的计算结果。建议年份为四位数时使用这类函数。

4.8.3.5 字符串类型

TiDB 支持 MySQL 所有的字符串类型，包括 CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM 以及 SET，完整信息参考这篇文档。

4.8.3.5.1 类型定义

CHAR 类型

定长字符串。CHAR 列的长度固定为创建表时声明的长度。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。和 VARCHAR 类型不同，CHAR 列在写入时会对数据末尾的空格进行截断。

```
[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]

字符集	单个字符字节数	VARCHAR 最大列长度的取值范围
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。最大列长为 65,535 字节。可选的M 参数以字符为单位，用于自动选择 TEXT 列的最合适类型。例如 TEXT(60) 会产生一个 TINYTEXT 数据类型，最多可容纳 255 字节，即容纳一个 60 字符的 UTF-8 字符串，每个字符最多包含 4 字节（即 4×60=240）。不推荐使用 M 参数。

```
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。但由于TiDB 单列的限制，TiDB 中默认单列存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
MEDIUMTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

LONGTEXT 类型

类似于TEXT，区别在于最大列长度为 4,294,967,295。但由于TiDB 单列的限制，TiDB 中默认单列存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
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 16,777,215。但由于TiDB 单列的限制，TiDB 中默认单列存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
MEDIUMBLOB
```

LONGBLOB 类型

类似于BLOB，区别在于最大列长度为 4,294,967,295。但由于TiDB 单列的限制，TiDB 中默认单列存储最大不超过 6 MB，可通过配置项将该限制调整至 120 MiB。

```
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。

4.8.3.6 JSON 类型

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，但是可以对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;
```

4.8.3.6.1 使用限制

- 暂不支持将JSON 函数下推至TiFlash。
- TiDB Backup & Restore（BR）在 v6.3.0 版本对JSON 列的数据的编码进行了修改。因此不建议使用 BR 恢复包含 JSON 列的数据到 v6.3.0 之前的TiDB 集群。
- 请勿使用任何同步工具同步非标准JSON 类型（例如 DATE、DATETIME、TIME 等）的数据。

4.8.3.6.2 MySQL 兼容性

- 当使用二进制类型数据创建JSON 时，目前 MySQL 会将其误标记为 STRING 类型，而TiDB 会保持正确的二进制类型。

```
CREATE TABLE test(a json);
INSERT INTO test SELECT json_objectagg('a', b'01010101');

-- 在 TiDB 中，执行以下 SQL 语句返回结果如下所示。在 MySQL 中，执行以下 SQL 语句的结果为 `0,
    <-> 1`。
mysql> SELECT JSON_EXTRACT(JSON_OBJECT('a', b'01010101'), '$.a') = "base64:type15:VQ==" AS
    <-> r1, JSON_EXTRACT(a, '$.a') = "base64:type15:VQ==" AS r2 FROM test;
+------+-----------+
| r1   | r2    |
+------+-----------+
|    0 |    0 |
+------+-----------+
1 row in set (0.01 sec)
```

详情可见此issue。

- 当将ENUM 或 SET 数据类型转换为JSON 时，TiDB 会检查其格式正确性。例如，当执行下面的 SQL 语句时，TiDB 中会报错：

```
CREATE TABLE t(e ENUM('a'));
INSERT INTO t VALUES ('a');
mysql> SELECT CAST(e AS JSON) FROM t;
ERROR 3140 (22032): Invalid JSON text: The document root must not be followed by other
    <-> values.
```

详情可见此issue。

- TiDB 支持使用 ORDER BY 对 JSON Array 或 JSON Object 进行排序。

  当使用 ORDER BY 对 JSON Array 或 JSON Object 进行排序时，MySQL 会返回一个警告，且排序结果与比较运算结果不一致：

```
CREATE TABLE t(j JSON);
INSERT INTO t VALUES ('[1,2,3,4]');
INSERT INTO t VALUES ('[5]');
```

```
mysql> SELECT j FROM t WHERE j < JSON_ARRAY(5);
+-----------------------+
| j                     |
+-----------------------+
| [1, 2, 3, 4] |
+-----------------------+
1 row in set (0.00 sec)

-- 在 TiDB 中，执行以下 SQL 语句返回结果如下所示。在 MySQL 中，执行以下 SQL 语句会返回警告
    <-> “This version of MySQL doesn't yet support 'sorting of non-scalar JSON values'.
    <->”，且排序结果与 `<` 比较结果不一致。
mysql> SELECT j FROM t ORDER BY j;
+-----------------------+
| j                     |
+-----------------------+
| [1, 2, 3, 4] |
| [5]           |
+-----------------------+
2 rows in set (0.00 sec)
```

详情可见此issue。

- 在 INSERT JSON 列时，TiDB 会将值隐式转换为JSON：

```
CREATE TABLE t(col JSON);

-- 在 TiDB 中，执行以下 INSERT 语句成功。在 MySQL 中，执行以下 INSERT 语句将返回 Invalid
    <-> JSON text 错误。
INSERT INTO t VALUES (3);
```

有关JSON 的更多信息，可以参考JSON 函数和生成列。

4.8.4 函数与操作符

4.8.4.1 函数和操作符概述

TiDB 中函数和操作符使用方法与 MySQL 基本一致，详情参见：Functions and Operators。

在SQL 语句中，表达式可用于诸如SELECT 语句的ORDER BY 或HAVING 子句，SELECT/DELETE/UPDATE 语句的WHERE 子句，或 SET 语句之类的地方。

可使用字面值，列名，NULL，内置函数，操作符等来书写表达式。其中有些表达式下推到 TiKV 上执行，详见下推到 TiKV 的表达式列表。

4.8.4.2 表达式求值的类型转换

TiDB 中表达式求值的类型转换与 MySQL 基本一致，详情参见 MySQL 表达式求值的类型转换。

4.8.4.3 操作符

操作符名	功能描述
AND, &&	逻辑与
=	赋值（可用于 SET 语句中，或用于 UPDATE 语句的 SET 中）
:=	赋值
BETWEEN ... AND ...	判断值满足范围
BINARY	将一个字符串转换为一个二进制字符串
&	位与
~	位非
|	位或
^	按位异或
CASE	case 操作符
DIV	整数除
/	除法
=	相等比较
<=>	空值安全型相等比较
>	大于
>=	大于或等于
IS	判断一个值是否等于一个布尔值
IS NOT	判断一个值是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
<<	左移
<	小于
<=	小于或等于
LIKE	简单模式匹配
ILIKE	大小写不敏感的简单模式匹配（TiDB 支持但 MySQL 不支持）
-	减
lstinlineMOD!	求余
NOT, !	取反
NOT BETWEEN ... AND ...	判断值是否不在范围内
!=, <>	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

4.8.4.3.1 不支持的操作符

- SOUNDS LIKE

4.8.4.3.2 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。

4.8.4.3.3 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
>	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

操作符名	功能描述
LEAST()	返回最小值
<	小于
<=	小于或等于
LIKE	简单模式匹配
ILIKE	大小写不敏感的简单模式匹配（TiDB 支持但 MySQL 不支持）
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。

4.8.4.3.4 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。

4.8.4.3.5 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。

4.8.4.3.6 MySQL 兼容性

- MySQL 不支持 ILIKE 操作符。

4.8.4.4 控制流程函数

TiDB 支持使用 MySQL 5.7 中提供的所有控制流程函数。

函数名	功能描述
CASE	Case 操作符
IF()	构建if/else
IFNULL()	构建 Null if/else
NULLIF()	如果 expr1 = expr2，返回 NULL

4.8.4.5 字符串函数

TiDB 支持使用大部分 MySQL 5.7 中提供的字符串函数、一部分 MySQL 8.0 中提供的字符串函数和一部分 Oracle 21 所提供的函数。

关于 Oracle 函数和TiDB 函数的对照关系，请参考Oracle 与 TiDB 函数和语法差异对照。

4.8.4.5.1 支持的函数

函数名	功能描述
ASCII()	返回最左字符的数值
BIN()	返回一个数的二进制值的字符串表示
BIT_LENGTH()	返回字符串的位长度
CHAR()	返回由整数的代码值所给出的字符组成的字符串
CHAR_LENGTH()	返回字符串的字符长度
CHARACTER_LENGTH()	与 CHAR_LENGTH() 功能相同
CONCAT()	返回连接的字符串
CONCAT_WS()	返回由分隔符连接的字符串
ELT()	返回指定位置的字符串
EXPORT_SET()	返回一个字符串，其中值位中设置的每个位，可以得到一个 on 字符串，而每个未设置的位，可以得到
FIELD()	返回参数在后续参数中出现的第一个位置
FIND_IN_SET()	返回第一个参数在第二个参数中出现的位置
FORMAT()	返回指定小数位数格式的数字
FROM_BASE64()	解码 base-64 表示的字符串，并返回结果
HEX()	返回一个十进制数或字符串值的 16 进制表示
INSERT()	在指定位置插入一个子字符串，最多不超过指定字符数
INSTR()	返回第一次出现的子字符串的索引
LCASE()	与 LOWER() 功能相同
LEFT()	返回最左侧指定长度的字符
LENGTH()	返回字符串长度，单位为字节
LIKE	进行简单模式匹配
LOCATE()	返回第一次出现的子字符串的位置
LOWER()	返回全小写的参数
LPAD()	返回字符串参数，左侧添加指定字符串
LTRIM()	去掉前缀空格
MAKE_SET()	返回一组用逗号分隔的字符串，这些字符串的位数与给定的bits 参数对应
MID()	返回一个以指定位置开始的子字符串
NOT LIKE	否定简单模式匹配

函数名	功能描述
NOT REGEXP	REGEXP 的否定形式
OCT()	返回一个数值的八进制表示，形式为字符串
OCTET_LENGTH()	与 LENGTH() 功能相同
ORD()	返回该参数最左侧字符的字符编码
POSITION()	与 LOCATE() 功能相同
QUOTE()	使参数逃逸，为了在 SQL 语句中使用
REGEXP	使用正则表达式匹配模式
REGEXP_INSTR()	返回满足正则的子字符串的第一个索引位置（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的
REGEXP_LIKE()	判断字符串是否满足正则表达式（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼容性）
REGEXP_REPLACE()	替换满足正则表达式的子字符串（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼容性）
REGEXP_SUBSTR()	返回满足正则表达式的子字符串（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼容性）
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
TRANSLATE()	将字符串中出现的所有指定字符替换为其它字符。这个函数不会像 Oracle 一样将空字符串视为NULL
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
WEIGHT_STRING()	返回字符串的权重

4.8.4.5.2 不支持的函数

- LOAD_FILE()
- MATCH()
- SOUNDEX()

4.8.4.5.3 正则函数与 MySQL 的兼容性

本节介绍TiDB 中正则函数 REGEXP_INSTR()、REGEXP_LIKE()、REGEXP_REPLACE()、REGEXP_SUBSTR() 与 MySQL 的兼容情况。

语法兼容性

MySQL 的实现使用的是ICU (International Components for Unicode) 库，TiDB 的实现使用的是RE2 库，两个库之间的语法差异可以查阅ICU 文档和 RE2 文档。

匹配模式 `match_type` 兼容性

TiDB 与 MySQL 在 `match_type` 上的差异：

- TiDB 中 `match_type` 可选值为：`"c"`、`"i"`、`"m"`、`"s"`。MySQL 中 `match_type` 可选值为：`"c"`、`"i"`、`"m"`、`"n"`、`"u"`。
- TiDB 中 `"s"` 对应 MySQL 中的 `"n"`，即 `.` 字符匹配行结束符。

  例如：MySQL 中 `SELECT REGEXP_LIKE(a, b, "n")FROM t1;` 在 TiDB 中需要修改为 `SELECT REGEXP_LIKE(a, b, "s")FROM t1;`。
- TiDB 不支持 `match_type` 为 `"u"`。

数据类型兼容性

TiDB 与 MySQL 在二进制字符串(binary string) 数据类型上的差异：

- MySQL 8.0.22 及以上版本中正则函数不支持二进制字符串，具体信息可查看 MySQL 文档。但在实际使用过程中，如果所有参数或者返回值的数据类型都是二进制字符串，则正则函数可以正常使用，否则报错。
- TiDB 目前完全禁止使用二进制字符串，无论什么情况都会报错。

其它兼容性

TiDB 与 MySQL 在替换空字符串上的差异：

下面以 `REGEXP_REPLACE("", "^$", "123")` 为例：

- MySQL 不会对空串进行替换，其结果为 `""`。
- TiDB 会对空串进行替换，其结果为 `"123"`。

4.8.4.6 数值函数与操作符

TiDB 支持使用 MySQL 5.7 中提供的所有数值函数与操作符。

4.8.4.6.1 算术操作符

操作符名	功能描述
+	加号
-	减号
*	乘号
/	除号
DIV	整数除法
lstinlineMOD!	模运算，取余
-	更改参数符号

4.8.4.6.2 数学函数

函数名	功能描述
POW()	返回参数的指定乘方的结果值
POWER()	返回参数的指定乘方的结果值
EXP()	返回e（自然对数的底）的指定乘方后的值
SQRT()	返回非负数的二次方根
LN()	返回参数的自然对数
LOG()	返回第一个参数的自然对数
LOG2()	返回参数以 2 为底的对数
LOG10()	返回参数以 10 为底的对数
PI()	返回pi的值
TAN()	返回参数的正切值
COT()	返回参数的余切值
SIN()	返回参数的正弦值
COS()	返回参数的余弦值
ATAN()	返回参数的反正切值
ATAN2(), ATAN()	返回两个参数的反正切值
ASIN()	返回参数的反正弦值
ACOS()	返回参数的反余弦值
RADIANS()	返回由度转化为弧度的参数
DEGREES()	返回由弧度转化为度的参数
MOD()	返回余数
ABS()	返回参数的绝对值
CEIL()	返回不小于参数的最小整数值
CEILING()	返回不小于参数的最小整数值
FLOOR()	返回不大于参数的最大整数值
ROUND()	返回参数最近似的整数或指定小数位数的数值
RAND()	返回一个随机浮点值
SIGN()	返回参数的符号
CONV()	不同数基间转换数字，返回数字的字符串表示
TRUNCATE()	返回被舍位至指定小数位数的数字
CRC32()	计算循环冗余码校验值并返回一个 32 位无符号值

4.8.4.7 日期和时间函数

TiDB 支持使用 MySQL 5.7 中提供的所有日期和时间函数。

4.8.4.7.1 日期时间函数表

函数名	功能描述
ADDDATE()	将时间间隔添加到日期上
ADDTIME()	时间数值相加
CONVERT_TZ()	转换时区

函数名	功能描述
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
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

函数名	功能描述
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
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

4.8.4.7.2 相关系统变量

`default_week_format` 变量影响 `WEEK()` 函数。

4.8.4.8 位函数和操作符

TiDB 支持使用 MySQL 5.7 中提供的所有位函数和操作符。

位函数和操作符表

函数和操作符名	功能描述
BIT_COUNT()	返回参数二进制表示中为 1 的个数
&	位与
~	按位取反
|	位或
[^](https://dev.mysql.com/doc/refman/5.7/en/bit-functions.html#operator_bitwise-xor)	位亦或
<<	左移
>>	右移

4.8.4.9 Cast 函数和操作符

Cast 函数和操作符用于将某种数据类型的值转换为另一种数据类型。TiDB 支持使用 MySQL 5.7 中提供的所有 Cast 函数和操作符。

4.8.4.9.1 Cast 函数和操作符表

函数和操作符名	功能描述
BINARY	将一个字符串转换成一个二进制字符串
CAST()	将一个值转换成一个确定类型
CONVERT()	将一个值转换成一个确定类型

> **注意：**
>
> TiDB 和 MySQL 对于 `SELECT CAST(MeN AS CHAR)`（或者等价的 `SELECT CONVERT(MeM, CHAR)`）的结果显示不一致，其中 `MeN` 是用科学计数法表示的双精度浮点数。MySQL 在 `-15 <= N <= 14` 时显示完整数值，在`N < -15` 或 `N > 14` 时显示科学计数法。而TiDB 始终显示完整数值。例如，MySQL 对于 `SELECT CAST(3.1415e15 AS CHAR)` 的显示结果为 `3.1415e15`，而 TiDB 的显示结果为 `3141500000000000`。

4.8.4.10 加密和压缩函数

TiDB 支持使用 MySQL 5.7 中提供的大部分加密和压缩函数。

4.8.4.10.1 支持的函数

函数名	功能描述
MD5()	计算字符串的 MD5 校验和
PASSWORD()	计算并返回密码字符串
RANDOM_BYTES()	返回随机字节向量
SHA1(), SHA()	计算 SHA-1 160 位校验和
SHA2()	计算 SHA-2 校验和
SM3()	计算 SM3 校验和（MySQL 中暂不支持该函数）
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
VALIDATE_PASSWORD_STRENGTH()	确定密码强度

4.8.4.10.2 相关系统变量

`block_encryption_mode` 变量设置 `AES_ENCRYPT()` 和 `AES_DECRYPT()` 所使用的加密模式。

4.8.4.10.3 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。
- 只在 MySQL 企业版中支持的函数。见Issue #2632。

4.8.4.11 锁函数

TiDB 支持 MySQL 5.7 中的大部分用户级锁函数。

4.8.4.11.1 支持的函数

函数名	功能描述
`GET_LOCK(lockName, timeout)`	获取咨询锁。`lockName` 参数不得超过 64 个字符。在超时前，TiDB 最长等待 `timeout` 秒并返回失败。
`RELEASE_LOCK(lockName)`	释放先前获取的锁。`lockName` 参数不得超过 64 个字符。
`RELEASE_ALL_LOCKS()`	释放当前会话持有的所有锁。

4.8.4.11.2 MySQL 兼容性

- TiDB 允许的最短超时时间为 1 秒，最长超时时间为 1 小时（即 3600 秒）。而 MySQL 允许最短 0 秒和最长无限超时（`timeout=-1`）。TiDB 会自动将超出范围的值转换为最接近的允许值，`timeout=-1` 会被转换为 3600 秒。
- TiDB 不会自动检测用户级锁导致的死锁。死锁会话将在 1 小时内超时，但你也可以在任一受影响的会话上使用 `KILL` 语句手动终止死锁。你还可以通过始终用相同顺序获取用户级锁的方法来防止死锁。
- 在 TiDB 中，锁对集群中所有TiDB 服务器生效。而在 MySQL Cluster 和 Group Replication 中，锁只对本地单个服务器生效。

4.8.4.11.3 不支持的函数

- `IS_FREE_LOCK()`
- `IS_USED_LOCK()`

4.8.4.12 信息函数

TiDB 支持使用 MySQL 5.7 中提供的大部分信息函数。

4.8.4.12.1 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_RESOURCE_GROUP()	返回当前连接的资源组名
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

4.8.4.12.2 不支持的函数

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

4.8.4.13 JSON 函数

TiDB 支持 MySQL 5.7 GA 版本发布的大多数JSON 函数。

4.8.4.13.1 创建JSON 值的函数

函数	功能描述
JSON_ARRAY([val[, val] ···])	根据一系列元素创建一个 JSON 文档
JSON_OBJECT(key, val[, key, val] ···)	根据一系列 K/V 对创建一个 JSON 文档
JSON_QUOTE(string)	返回一个字符串，该字符串为带引号的JSON 值

4.8.4.13.2 搜索JSON 值的函数

函数	功能描述
JSON_CONTAINS(target, candidate[, path])	通过返回 1 或 0 来表示目标JSON 文档中是否包含给定的candidate JSON 文档
JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ···)	通过返回 0 或 1 来表示一个JSON 文档在给定路径是否包含数据
JSON_EXTRACT(json_doc, path[, path] ···)	从JSON 文档中解出某一路径对应的子文档
->	返回执行路径后面的JSON 列的值；JSON_EXTRACT(doc, path_literal) 的别名

函数	功能描述
->>	返回执行路径后面的JSON 列的值和转义后的结果；`JSON_UNQUOTE(JSON_EXTRACT(doc, path_literal))` 的别名
JSON_KEYS(json_doc[, path])	返回从JSON 对象的顶级值作为JSON array 的键，如果给定了路径参数，则从选定路径中获取顶级键
JSON_SEARCH(json_doc, one_or_all, search_str[, escape_char[, path] ⋯])	返回指定字符在JSON 文档中的路径
value MEMBER OF(json_array)	如果传入值是JSON array 中的一个元素，返回 1，否则返回 0
JSON_OVERLAPS(json_doc1, json_doc2)	表示两个JSON 文档中是否包含公共部分。返回 1 表示两个JSON 文档中包含公共部分，否则返回 0

4.8.4.13.3 修改JSON 值的函数

函数	功能描述
JSON_APPEND(json_doc, path, value)	JSON_ARRAY_APPEND <⟷ 的别名
JSON_ARRAY_APPEND(json_doc, path, value)	将值追加到指定路径的JSON 数组的末尾
JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ⋯)	将数组插入JSON 文档，并返回修改后的文档
JSON_INSERT(json_doc, path, val[, path, val] ⋯)	在 JSON 文档中在某一路径下插入子文档
JSON_MERGE(json_doc, json_doc[, json_doc] ⋯)	已废弃的 JSON_MERGE_PRESERVE <⟷ 别名
JSON_MERGE_PATCH(json_doc, json_doc[, json_doc] ⋯)	合并JSON 文档
JSON_MERGE_PRESERVE(json_doc, json_doc[, json_doc] ⋯)	将两个或多个 JSON 文档合并成一个文档，并返回合并结果
JSON_REMOVE(json_doc, path[, path] ⋯)	移除JSON 文档中某一路径下的子文档

函数	功能描述
JSON_REPLACE(json_doc, path, val[, path, val] ···)	替换JSON 文档中的某一路径下的子文档
JSON_SET(json_doc, path, val[, path, val] ···)	在 JSON 文档中为某一路径设置子文档
JSON_UNQUOTE(json_val)	去掉JSON 值外面的引号，返回结果为字符串
JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ···)	将值添加到 JSON 文档指定数组的末尾，并返回添加结果
JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ···)	将值插入到 JSON 文档的指定位置，并返回插入结果

4.8.4.13.4 返回JSON 值属性的函数

函数	功能描述
JSON_DEPTH(json_doc)	返回JSON 文档的最大深度
JSON_LENGTH(json_doc[, path])	返回JSON 文档的长度；如果路径参数已定，则返回该路径下值的长度
JSON_TYPE(json_val)	检查某JSON 文档内部内容的类型
JSON_VALID(json_doc)	检查JSON 文档内容是否有效；用于将列转换为JSON 类型之前对该列进行检查

4.8.4.13.5 效用函数

函数	功能描述
JSON_PRETTY(json_doc)	格式化JSON 文档

函数	功能描述
JSON_STORAGE_FREE(json_doc)	返回该JSON 对象的存储空间中空闲的字节数。由于TiDB 采用与 MySQL 完全不同的存储结构，本函数对合法的 JSON 值总是返回 0，主要用于兼容 MySQL 8.0
JSON_STORAGE_SIZE(json_doc)	返回存储JSON 值所需的大致字节大小，由于不考虑TiKV 压缩的字节大小，因此函数的输出与 MySQL 不严格兼容

4.8.4.13.6 聚合函数

函数	功能描述
JSON_ARRAYAGG(key)	提供指定列 key 的聚合
JSON_OBJECTAGG(key, value)	提供给定两列键值对的聚合

4.8.4.13.7 另请参阅

- JSON Function Reference
- JSON Data Type

4.8.4.14 GROUP BY 聚合函数

本文将详细介绍TiDB 支持的聚合函数。

4.8.4.14.1 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

函数名	功能描述
COUNT(DISTINCT)	返回不同值的数目
SUM()	返回和
AVG()	返回平均值
MAX()	返回最大值
MIN()	返回最小值
GROUP_CONCAT()	返回连接的字符串
VARIANCE(), VAR_POP()	返回总体标准方差
STD(), STDDEV(), STDDEV_POP	返回总体标准差
VAR_SAMP()	返回采样方差
STDDEV_SAMP()	返回采样标准方差
JSON_OBJECTAGG(key, value)	将结果集返回为单个含(key, value)键值对的JSON object

注意：

- 除非另有说明，否则聚合函数默认忽略 NULL 值。
- 如果在不包含 GROUP BY 子句的语句中使用聚合函数，则相当于对所有行进行分组。

另外，TiDB 还支持以下聚合函数：

- APPROX_PERCENTILE(expr, constant_integer_expr)

  该函数用于计算 expr 值的百分位数。参数 constant_integer_expr 是一个取值为区间 [1,100] 内整数的常量表达式，表示百分数。一个百分位数 Pk（k为百分数）表示数据集中至少有 k% 的数据小于等于 Pk。

  该函数中，表达式的返回结果必须为数值类型或日期与时间类型。函数不支持计算其他类型的返回结果，并直接返回 NULL。

  以下是一个计算第 50 百分位数的例子：

```
drop table if exists t;
create table t(a int);
insert into t values(1), (2), (3);
```

```
select approx_percentile(a, 50) from t;
```

```
+----------------------------+
| approx_percentile(a, 50) |
+----------------------------+
|                          2 |
+----------------------------+
1 row in set (0.00 sec)
```

上述聚合函数除 GROUP_CONCAT() 和 APPROX_PERCENTILE() 以外，均可作为窗口函数使用。

4.8.4.14.2 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。

4.8.4.14.3 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;
```

4.8.4.14.4 相关系统变量

group_concat_max_len 变量设置 GROUP_CONCAT() 函数中的最大项目数。

4.8.4.15 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。

本页列出的窗口函数可以下推到TiFlash。

TiDB 支持除 GROUP_CONCAT() 和 APPROX_PERCENTILE() 以外的所有GROUP BY 聚合函数。此外，TiDB 支持的其他窗口函数如下：

函数名	功能描述
CUME_DIST()	返回一组值中的累积分布
DENSE_RANK()	返回分区中当前行的排名，并且排名是连续的
FIRST_VALUE()	当前窗口中第一行的表达式值
LAG()	分区中当前行前面第 N 行的表达式值
LAST_VALUE()	当前窗口中最后一行的表达式值
LEAD()	分区中当前行后面第 N 行的表达式值
NTH_VALUE()	当前窗口中第 N 行的表达式值
NTILE()	将分区划分为N 桶，为分区中的每一行分配桶号
PERCENT_RANK()	返回分区中小于当前行的百分比
RANK()	返回分区中当前行的排名，排名可能不连续
ROW_NUMBER()	返回分区中当前行的编号

4.8.4.16 其他函数

TiDB 支持使用 MySQL 5.7 中提供的大部分其他函数。

4.8.4.16.1 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
BIN_TO_UUID()	将通用唯一识别码 (UUID) 从二进制格式转换为文本格式
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)
UUID_TO_BIN()	将 UUID 从文本格式转换为二进制格式
VALUES()	定义 INSERT 语句使用的值

4.8.4.16.2 不支持的函数

函数名	功能描述
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

4.8.4.17 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

4.8.4.17.1 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。

4.8.4.17.2 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。

4.8.4.17.3 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

4.8.4.17.4 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)
```

4.8.4.18 集合运算

TiDB 支持三种集合运算：并集(UNION)，差集(EXCEPT) 和交集(INTERSECT)。最小的集合单位是一个SELECT 语句。

4.8.4.18.1 并集(UNION)

数学上，两个集合A 和 B 的并集是含有所有属于A 或属于B 的元素。下面是一个 UNION 的例子：

```
SELECT 1 UNION SELECT 2;
+---+
| 1 |
+---+
| 2 |
| 1 |
+---+
2 rows in set (0.00 sec)
```

TiDB 支持 UNION ALL 和 UNION DISTINCT 并集，两者区别在于 UNION DISTINCT 会对并集结果去重复，而 UNION <—> ALL 不会。TiDB 中默认使用 UNION DISTINCT。

```
CREATE TABLE t1 (a int);
CREATE TABLE t2 (a int);
INSERT INTO t1 VALUES (1),(2);
INSERT INTO t2 VALUES (1),(3);
```

UNION DISTINCT与 UNION ALL 的结果分别如下：

```
SELECT * FROM t1 UNION DISTINCT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows in set (0.00 sec)

SELECT * FROM t1 UNION ALL SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 1 |
| 3 |
+---+
4 rows in set (0.00 sec)
```

4.8.4.18.2 差集(EXCEPT)

若A和B是集合，则A与B的差集是由所有属于A但不属于B的元素组成的集合。

```
SELECT * FROM t1 EXCEPT SELECT * FROM t2;
+---+
| a |
+---+
| 2 |
+---+
1 rows in set (0.00 sec)
```

差集(EXCEPT)暂时不支持 EXCEPT ALL。

4.8.4.18.3 交集(INTERSECT)

数学上，两个集合A和B的交集是含有所有既属于A又属于B的元素，而且没有其他元素的集合。

```
SELECT * FROM t1 INTERSECT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
+---+
```

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

交集(INTERSECT)暂时不支持 INTERSECT ALL。交集 (INTERSECT) 的计算优先级大于差集(EXCEPT) 和并集(UNION)。

```
SELECT * FROM t1 UNION ALL SELECT * FROM t1 INTERSECT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
| 1 |
| 2 |
+---+
3 rows in set (0.00 sec)
```

4.8.4.18.4 括号优先

TiDB 支持使用括号修改集合运算的优先级，如同四则运算中先计算括号部分，集合运算也先计算括号内的部分。

```
(SELECT * FROM t1 UNION ALL SELECT * FROM t1) INTERSECT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
+---+
1 rows in set (0.00 sec)
```

4.8.4.18.5 与 ORDER BY 和 Limit 结合

TiDB 支持单独为整个集合运算进行 ORDER BY 或者 LIMIT。

```
(SELECT * FROM t1 UNION ALL SELECT * FROM t1 INTERSECT SELECT * FROM t2) ORDER BY a LIMIT 2;
+---+
| a |
+---+
| 1 |
| 1 |
+---+
2 rows in set (0.00 sec)
```

4.8.4.19 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。

TiFlash 也支持本页列出的函数和算子下推。

4.8.4.19.1 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 (!), XOR
位运算	&, ~, |, ^, <<, >>
比较运算	<, <=, =, !=（<>）, >, >=, <=>, BETWEEN ··· AND ···, COALESCE(), IN(), INTERVAL(), IS NOT NULL, IS NOT, IS NULL, IS, ISNULL(), LIKE, NOT BETWEEN ··· AND ···, NOT IN(), NOT LIKE, STRCMP()
数值运算	+, -, *, /, DIV, % (MOD), -, ABS(), ACOS(), ASIN(), ATAN(), ATAN2(), ATAN(), CEIL(), CEILING(), CONV(), COS(), COT(), CRC32(), DEGREES(), EXP(), FLOOR(), LN(), LOG(), LOG10(), LOG2(), MOD(), PI(), POW(), POWER(), RADIANS(), RAND(), ROUND(), SIGN(), SIN(), SQRT()
控制流运算	CASE, IF(), IFNULL()
JSON 运算	JSON_ARRAY([val[, val] ···]), JSON_CONTAINS(target, candidate[, path]), JSON_EXTRACT(json_doc, path[, path] ···), JSON_INSERT(json_doc, path, val[, path, val] ···), JSON_LENGTH(json_doc[, path]), JSON_MERGE(json_doc, json_doc[, json_doc] ···), JSON_OBJECT([key, val[, key, val] ···]), JSON_REMOVE(json_doc, path[, path] ···), JSON_REPLACE(json_doc, path, val[, path, val] ···), JSON_SET(json_doc, path, val[, path, val] ···), JSON_TYPE(json_val), JSON_UNQUOTE(json_val), JSON_VALID(val)
日期运算	DATE(), DATE_FORMAT(), DATEDIFF(), DAYOFMONTH(), DAYOFWEEK(), DAYOFYEAR(), FROM_DAYS(), HOUR(), MAKEDATE(), MAKETIME(), MICROSECOND(), MINUTE(), MONTH(), MONTHNAME(), PERIOD_ADD(), PERIOD_DIFF(), SEC_TO_TIME(), SECOND(), SYSDATE(), TIME_TO_SEC(), TIMEDIFF(), WEEK(), WEEKOFYEAR(), YEAR()
字符串函数	ASCII(), BIT_LENGTH(), CHAR(), CHAR_LENGTH(), CONCAT(), CONCAT_WS(), ELT(), FIELD(), HEX(), LENGTH(), LIKE, LTRIM(), MID(), NOT LIKE, NOT REGEXP, REGEXP, REPLACE(), REVERSE(), RIGHT(), RTRIM(), SPACE(), STRCMP(), SUBSTR(), SUBSTRING()
聚合函数	COUNT(), COUNT(DISTINCT), SUM(), AVG(), MAX(), MIN(), VARIANCE(), VAR_POP(), STD(), STDDEV(), STDDEV_POP, VAR_SAMP(), STDDEV_SAMP(), JSON_ARRAYAGG(key), JSON_OBJECTAGG(key, value)
加密和压缩函数	MD5(), SHA1(), SHA(), UNCOMPRESSED_LENGTH()
Cast 函数	CAST(), CONVERT()
其他函数	UUID()

4.8.4.19.2 禁止特定表达式下推

当已支持下推的表达式列表中的函数和运算符，或特定的数据类型（仅限ENUM 类型和BIT 类型）的计算过程因下推而出现异常时，你可以使用黑名单功能禁止其下推，从而快速恢复TiDB 业务。具体而言，你可以将函数名、运算符名，或数据列类型加入黑名单 mysql.expr_pushdown_blacklist 中，以禁止特定表达式下推。具体方法，请参阅表达式下推黑名单。

4.8.4.20 TiDB 特有的函数

本文档介绍TiDB 特有的函数。

4.8.4.20.1 TIDB_BOUNDED_STALENESS

TIDB_BOUNDED_STALENESS 是TiDB 的内部函数，用于指定一个时间范围。用法为 TIDB_BOUNDED_STALENESS(t1, t2)，其中 t1 和t2 为时间范围的两端，支持使用日期时间和时间函数。

使用该函数，TiDB 会在指定的时间范围内选择一个合适的时间戳，该时间戳能保证所访问的副本上不存在开始于这个时间戳之前且还没有提交的相关事务，即能保证所访问的可用副本上执行读取操作而且不会被阻塞。

4.8.4.20.2 TIDB_DECODE_KEY

TIDB_DECODE_KEY 函数用于将TiDB 编码的键输入解码为包含 _tidb_rowid 和 table_id 的JSON 结构。你可以在一些系统表和日志输出中找到TiDB 的编码键。

语法图

```
TableStmt ::=
    "TIDB_DECODE_KEY(" STR ")"
```

示例

以下示例中，表 t1 有一个隐藏的 rowid，该 rowid 由 TiDB 生成。语句中使用了 TIDB_DECODE_KEY 函数。结果显示，隐藏的 rowid 被解码后并输出，这是典型的非聚簇主键结果。

```
SELECT START_KEY, TIDB_DECODE_KEY(START_KEY) FROM information_schema.tikv_region_status WHERE
    <-> table_name='t1' AND REGION_ID=2\G
```

```
*************************** 1. row ***************************
                 START_KEY: 7480000000000000FF3B5F728000000000FF1DE3F10000000000FA
TIDB_DECODE_KEY(START_KEY): {"_tidb_rowid":1958897,"table_id":"59"}
1 row in set (0.00 sec)
```

以下示例中，表 t2 有一个复合聚簇主键。由JSON 输出可知，输出结果的 handle 项中包含了主键部分两列的信息，即两列的名称和对应的值。

```
show create table t2\G
```

```
*************************** 1. row ***************************
       Table: t2
Create Table: CREATE TABLE `t2` (
  `id` binary(36) NOT NULL,
  `a` tinyint(3) unsigned NOT NULL,
  `v` varchar(512) DEFAULT NULL,
  PRIMARY KEY (`a`,`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.001 sec)
```

```
select * from information_schema.tikv_region_status where table_name='t2' limit 1\G
```

```
*************************** 1. row ***************************
                REGION_ID: 48
                START_KEY: 7480000000000000
                    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DF
                    ↪
                  END_KEY:
                 TABLE_ID: 62
                  DB_NAME: test
               TABLE_NAME: t2
                 IS_INDEX: 0
                 INDEX_ID: NULL
               INDEX_NAME: NULL
           EPOCH_CONF_VER: 1
            EPOCH_VERSION: 38
            WRITTEN_BYTES: 0
               READ_BYTES: 0
         APPROXIMATE_SIZE: 136
         APPROXIMATE_KEYS: 479905
  REPLICATIONSTATUS_STATE: NULL
REPLICATIONSTATUS_STATEID: NULL
1 row in set (0.005 sec)
```

```
select tidb_decode_key('7480000000000000
    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DFFFF3830653635303
    ↪ ');
```

```
+--
    ↪ ------------------------------------------------------------------------------------------------------------
    ↪
| tidb_decode_key('7480000000000000
    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DFFF3830653635303
    ↪ ') |
```

```
+--
    <->
    <->
|  {"handle":{"a":"6","id":"c4038db2-d51c-11eb-8c75-80e65018a9be"},"table_id":62}
    <->
    <-> |
+--
    <->
    <->
1 row in set (0.001 sec)
```

MySQL 兼容性

TIDB_DECODE_KEY 是TiDB 特有的函数，和 MySQL 不兼容。

4.8.4.20.3 TIDB_DECODE_PLAN

TIDB_DECODE_PLAN 函数用于解码TiDB 执行计划。你可以在慢查询日志中找到TiDB 执行计划。

语法图

```
TableStmt ::=
    "TIDB_DECODE_PLAN(" STR ")"
```

示例

```
SELECT tidb_decode_plan('8
    <-> QIYMAkzMV83CQEH8E85LjA0CWRhdGE6U2VsZWN0aW9uXzYJOTYwCXRpbWU6NzEzLjHCtXMsIGxvb3BzOjIsIGNvcF90YXNrOiB7bnVtOi
    <-> ')\G
```

```
*************************** 1. row ***************************
  tidb_decode_plan('8
      <-> QIYMAkzMV83CQEH8E85LjA0CWRhdGE6U2VsZWN0aW9uXzYJOTYwCXRpbWU6NzEzLjHCtXMsIGxvb3BzOjIsIGNvcF90YXNrOiB7bnVt
      <-> :     id                    task          estRows     operator info
      <->                             actRows     execution info
      <->
      <-> memory      disk
    TableReader_7           root          319.04     data:Selection_6                                960
        <->         time:713.1µs, loops:2, cop_task: {num: 1, max: 568.5µs, proc_keys: 0, rpc_num
        <-> : 1, rpc_time: 549.1µs, copr_cache_hit_ratio: 0.00}     3.99 KB     N/A
    └─Selection_6           cop[tikv]     319.04     lt(test.t.a, 10000)
        <-> 960         tikv_task:{time:313.8µs, loops:960}
        <->
        <-> N/A         N/A
      └─TableFullScan_5     cop[tikv]     960         table:t, keep order:false, stats:pseudo
          <-> 960         tikv_task:{time:153µs, loops:960}
          <->
          <-> N/A         N/A
```

MySQL 兼容性

TIDB_DECODE_PLAN 是TiDB 特有的函数，和 MySQL 不兼容。

4.8.4.20.4 TIDB_IS_DDL_OWNER

TIDB_IS_DDL_OWNER 函数用于检查你连接的TiDB 实例是否是 DDL Owner。 DDL Owner 代表集群中所有其他节点执行 DDL 语句的TiDB 实例。

语法图

```
TableStmt ::=
    "TIDB_IS_DDL_OWNER())"
```

示例

```
SELECT tidb_is_ddl_owner();
```

```
+----------------------+
| tidb_is_ddl_owner() |
+----------------------+
|                    1 |
+----------------------+
1 row in set (0.00 sec)
```

MySQL 兼容性

TIDB_IS_DDL_OWNER 是 TiDB 特有的函数，和 MySQL 不兼容。

另请参阅

- ADMIN SHOW DDL
- ADMIN CANCEL DDL

4.8.4.20.5 TIDB_PARSE_TSO

TIDB_PARSE_TSO 函数用于从TiDB TSO 时间戳中提取物理时间戳。

TSO 指 Time Stamp Oracle，是 PD (Placement Driver) 为每个事务提供的单调递增的时间戳。

TSO 是一串数字，包含以下两部分：

- 一个物理时间戳
- 一个逻辑计数器

语法图

```
TableStmt ::=
    "TIDB_PARSE_TSO(" NUM ")"
```

示例

```
BEGIN;
SELECT TIDB_PARSE_TSO(@@tidb_current_ts);
ROLLBACK;
```

```
+-----------------------------------+
| TIDB_PARSE_TSO(@@tidb_current_ts) |
+-----------------------------------+
| 2021-05-26 11:33:37.776000        |
+-----------------------------------+
1 row in set (0.0012 sec)
```

以上示例使用 TIDB_PARSE_TSO 函数从 tidb_current_ts 会话变量提供的可用时间戳编号中提取物理时间戳。因为每个事务都会分配到时间戳，所以此函数在事务中运行。

MySQL 兼容性

TIDB_PARSE_TSO 是 TiDB 特有的函数，和 MySQL 不兼容。

另请参阅

- tidb_current_ts

4.8.4.20.6 TIDB_VERSION

TIDB_VERSION 函数用于获取当前连接的TiDB 服务器版本和构建详细信息。向 GitHub 上提交issue 时，你可使用此函数获取相关信息。

语法图

```
TableStmt ::=
    "TIDB_VERSION()"
```

示例

```
SELECT TIDB_VERSION()\G
```

```
*************************** 1. row ***************************
TIDB_VERSION(): Release Version: v5.1.0-alpha-13-gd5e0ed0aa-dirty
Edition: Community
Git Commit Hash: d5e0ed0aaed72d2f2dfe24e9deec31cb6cb5fdf0
Git Branch: master
UTC Build Time: 2021-05-24 14:39:20
GoVersion: go1.13
Race Enabled: false
TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306
Check Table Before Drop: false
1 row in set (0.00 sec)
```

MySQL 兼容性

TIDB_VERSION 是 TiDB 特有的函数，和 MySQL 不兼容。如果要求兼容 MySQL，可以使用 VERSION 获取版本信息，但结果不包含详细的构建信息。

4.8.4.20.7 TIDB_DECODE_SQL_DIGESTS

TIDB_DECODE_SQL_DIGESTS 函数用于在集群中查询一组 SQL Digest 所对应的 SQL 语句的归一化形式（即去除格式和参数后的形式）。函数接受 1 个或 2 个参数：

- digests：字符串类型，该参数应符合 JSON 字符串数组的格式，数组中的每个字符串应为一个 SQL Digest。
- stmtTruncateLength：可选参数，整数类型，用来限制返回结果中每条 SQL 语句的长度，超过指定的长度会被截断。0 表示不限制长度。

返回一个字符串，符合JSON 字符串数组的格式，数组中的第i 项为参数 digests 中的第i 个元素所对应的语句。如果参数 digests 中的某一项不是一个有效的 SQL Digest 或系统无法查询到其对应的 SQL 语句，则返回结果中对应项为 null。如果指定了截断长度（stmtTruncateLength > 0），则返回结果中每条超过该长度的语句，保留前 stmtTruncateLength 个字符，并在尾部增加 "..." 后缀表示发生了截断。如果参数 digests 为 NULL，则函数的返回值为 NULL。

> 注意：
>
> - 仅持有 PROCESS 权限的用户可以使用该函数。
> - TIDB_DECODE_SQL_DIGESTS 执行时，TiDB 内部从 Statement Summary 一系列表中查询每个 SQL Digest 所对应的语句，因而并不能保证对任意 SQL Digest 都总是能查询到对应的语句，只有在集群中执行过的语句才有可能被查询到，且是否能查询到受 Statement Summary 表相关配置的影响。有关 Statement Summary 表的详细说明，参见 Statement Summary Tables。
> - 该函数开销较大，在行数很多的查询中（比如在规模较大、比较繁忙的集群上查询 information_schema.cluster_tidb_trx 全表时）直接使用该函数可能导致查询运行时间较长。请谨慎使用。
>   - 该函数开销大的原因是，其每次被调用时，都会在内部发起对 STATEMENTS_SUMMARY、STATEMENTS_SUMMARY_HISTORY、CLUSTER_STATEMENTS_SUMMARY 和 CLUSTER_STATEMENTS_SUMMARY_HISTORY 这几张表的查询，且其中涉及 UNION 操作。且该函数目前不支持向量化，即对于多行数据调用该函数时，对每行都会独立进行一次上述的查询。

语法图

```
DecodeSQLDigestsExpr ::=
    "TIDB_DECODE_SQL_DIGESTS" "(" digests ( "," stmtTruncateLength )? ")"
```

示例

```
set @digests = '["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
    <-> b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821","
    <-> e5796985ccafe2f71126ed6c0ac939ffa015a8c0744a24b7aee6d587103fd2f7"]';

select tidb_decode_sql_digests(@digests);
```

```
+--------------------------------------+
| tidb_decode_sql_digests(@digests)  |
+--------------------------------------+
| ["begin",null,"select * from `t`"] |
+--------------------------------------+
1 row in set (0.00 sec)
```

上面的例子中，参数是一个包含 3 个 SQL Digest 的 JSON 数组，其对应的 SQL 语句分别为查询结果中给出的三项。但是其中第二条 SQL Digest 所对应的 SQL 语句未能从集群中找到，因而结果中的第二项为 `null`。

```
select tidb_decode_sql_digests(@digests, 10);
```

```
+----------------------------------------+
 | tidb_decode_sql_digests(@digests, 10) |
+----------------------------------------+
| ["begin",null,"select * f..."]         |
+----------------------------------------+
1 row in set (0.01 sec)
```

上述调用指定了第二个参数（即截断长度）为 10，而查询结果中的第三条语句的长度大于 10，因而仅保留了前 10 个字符，并在尾部添加了 `"..."` 表示发生了截断。

MySQL 兼容性

`TIDB_DECODE_SQL_DIGESTS` 是 TiDB 特有的函数，和 MySQL 不兼容。

另请参阅

- Statement Summary Tables
- INFORMATION_SCHEMA.TIDB_TRX

4.8.4.20.8 TIDB_SHARD

`TIDB_SHARD` 函数用于创建一个 SHARD INDEX 来打散热点索引。SHARD INDEX 是一种以 `TIDB_SHARD` 函数为前缀的表达式索引。

SHARD INDEX

- 创建方式：

  使用 `uk((tidb_shard(a)), a))` 为字段a 创建一个 SHARD INDEX。当二级唯一索引 `uk((tidb_shard(a)), a))` 的索引字段a 上存在因单调递增或递减而产生的热点时，索引的前缀 `tidb_shard(a)` 会打散热点，从而提升集群可扩展性。

- 适用场景：
  - 二级唯一索引上 key 值存在单调递增或递减导致的写入热点，且该索引包含的列是整型。
  - 业务中 SQL 语句根据该二级索引的全部字段做等值查询，查询可以是单独的 SELECT，也可以是 UPDATE，DELETE 等产生的内部查询，等值查询包括 a = 1 或 a IN (1, 2, ...) 两种方式。
- 使用限制：
  - 非等值查询无法使用索引。
  - 查询条件中 AND 和 OR 混合且最外层是 AND 算子时无法使用 SHARD INDEX。
  - GROUP BY 无法使用 SHARD INDEX。
  - ORDER BY 无法使用 SHARD INDEX。
  - ON 子句无法使用 SHARD INDEX。
  - WHERE 子查询无法使用 SHARD INDEX。
  - SHARD INDEX 只能打散整型字段的唯一索引。
  - SHARD INDEX 联合索引可能失效。
  - SHARD INDEX 无法走 FastPlan 流程，影响优化器性能。
  - SHARD INDEX 无法使用执行计划缓存。

语法图

```
TIDBShardExpr ::=
    "TIDB_SHARD" "(" expr ")"
```

示例

- 使用 TIDB_SHARD 函数计算 SHARD 值

  以下示例说明如何使用 TIDB_SHARD 函数计算 12373743746 的 SHARD 值。

```
SELECT TIDB_SHARD(12373743746);
```

- 计算得出 SHARD 值为：

```
+--------------------------+
| TIDB_SHARD(12373743746) |
+--------------------------+
|                      184 |
+--------------------------+
1 row in set (0.00 sec)
```

- 使用 TIDB_SHARD 函数创建 SHARD INDEX

```
CREATE TABLE test(id INT PRIMARY KEY CLUSTERED, a INT, b INT, UNIQUE KEY uk((tidb_shard(a)),
    <-> a));
```

MySQL 兼容性

TIDB_SHARD 是TiDB 特有的函数，和 MySQL 不兼容。

4.8.4.20.9 TIDB_ROW_CHECKSUM

TIDB_ROW_CHECKSUM 函数用于查询行数据的 Checksum 值。该函数只能用于 FastPlan 流程的 SELECT 语句，即你可通过形如 SELECT TIDB_ROW_CHECKSUM()FROM t WHERE id = ? 或 SELECT TIDB_ROW_CHECKSUM()FROM t WHERE id IN (?, ?, ...) 的语句进行查询。

语法图

```
TableStmt ::=
    "TIDB_ROW_CHECKSUM()"
```

示例

在 TiDB 中开启行数据 Checksum 功能tidb_enable_row_level_checksum：

```
SET GLOBAL tidb_enable_row_level_checksum = ON;
```

创建表t 并插入数据：

```
USE test;
CREATE TABLE t (id INT PRIMARY KEY, k INT, c int);
INSERT INTO TABLE t values (1, 10, a);
```

查询表t 中 id = 1 的行数据的 Checksum 值：

```
SELECT *, TIDB_ROW_CHECKSUM() FROM t WHERE id = 1;
```

```
+----+------+------+--------------------------------- +
| id | k    | c    | TIDB_ROW_CHECKSUM() |
+----+------+------+--------------------------------- +
|  1 |   10 | a    | 3813955661           |
+----+------+------+--------------------------------- +
1 row in set (0.000 sec)
```

MySQL 兼容性

TIDB_ROW_CHECKSUM 是 TiDB 特有的函数，和 MySQL 不兼容。

另请参阅

- 数据正确性校验

4.8.4.21 Oracle 与 TiDB 函数和语法差异对照

本文档提供了 Oracle 与TiDB 的函数和语法差异对照，方便你根据 Oracle 函数查找对应的TiDB 函数，了解 Oracle 与 TiDB 语法差异。

> 注意：
> 本文的内容是基于 Oracle 12.2.0.1.0 和TiDB v5.4.0，其他版本可能存在差异。

4.8.4.21.1 函数对照表

下表列出了 Oracle 与 TiDB 部分函数的对照表。

函数	Oracle 语法	TiDB 语法	说明
转换数据类型			
	TO_NUMBER(key)		
	TO_CHAR(key)		
	CONVERT(key,dataType)	TiDB 支持转换为下面类型：BINARY、CHAR、DATE、DATETIME、TIME、SIGNED INTEGER、UNSIGNED INTEGER 和 DECIMAL。	
日期类型转换为字符串类型			
	TO_CHAR(SYSDATE,'yyyy-MM-dd hh24:mi:ss')		
	TO_CHAR(SYSDATE,'yyyy-MM-dd')		
	DATE_FORMAT(NOW(),'%Y-%m-%d %H:%i:%s')		
	DATE_FORMAT(NOW(),'%Y-%m-%d')		
		TiDB 的格式化字符串大小写敏感。	
字符串类型转换为日期类型			
	TO_DATE('2021-05-28 17:31:37','yyyy-MM-dd hh24:mi:ss')		
	TO_DATE('2021-05-28','yyyy-MM-dd hh24:mi:ss')		
	STR_TO_DATE('2021-05-28 17:31:37','%Y-%m-%d %H:%i:%s')		
	STR_TO_DATE('2021-05-28','%Y-%m-%d%T')		
		TiDB 的格式化字符串大小写敏感。	
获取系统当前时间（精确到秒）	SYSDATE	NOW()	
获取当前时间（精确到微秒）	SYSTIMESTAMP	CURRENT_TIMESTAMP(6)	
获取两个日期相差的天数	date1 - date2	DATEDIFF(date1, date2)	
获取两个日期间隔月份	MONTHS_BETWEEN(ENDDATE,SYSDATE)	TIMESTAMPDIFF(<⟷ MONTH,SYSDATE,ENDDATE)	Oracle 中 MONTHS_BETWEEN() 函数与TiDB 中 TIMESTAMPDIFF() 函数的结果会有误差。TIMESTAMPDIFF() 只保留整数月。使用时需要注意，两个函数的参数位置相反。
日期增加/减少 n 天	DATEVAL + n	DATE_ADD(dateVal,INTERVAL n DAY)	n 可为负数。
日期增加/减少 n 月	ADD_MONTHS(dateVal<⟷ ,n)	DATE_ADD(dateVal,INTERVAL n MONTH)	n 可为负数。
获取日期到日	TRUNC(SYSDATE)		
	CAST(NOW()AS DATE)		
	DATE_FORMAT(NOW(),'%Y-%m-%d')		
		TiDB 中 CAST 与 DATE_FORMAT 结果一致。	
获取日期当月第一天	TRUNC(SYSDATE,'mm')	DATE_ADD(CURDATE()<⟷ ,interval - day(CURDATE())+ 1 day)	
截取数据	TRUNC(2.136)= 2 TRUNC(2.136,2)= 2.13	TRUNCATE<⟷ (2.136,0)= 2 TRUNCATE(2.136,2)= 2.13	数据精度保留，直接截取相应小数位，不涉及四舍五入。
获取序列下一个值	sequence_name.NEXTVAL	NEXTVAL(sequence_name)	
获取随机序列值	SYS_GUID()	UUID()	TiDB 返回一个通用唯一识别码 (UUID)。
左/右外连接	SELECT * FROM a, b WHERE a.id = b.id<⟷ (+);SELECT * FROM a, b WHERE a.id(+)= b.id;	SELECT * FROM a LEFT JOIN b ON a.id = b.id;SELECT<⟷ * FROM a RIGHT JOIN b ON a.id = b.id;	关联查询时，TiDB 不支持使用 (+) 实现左/右关联， 只能通

过 LEFT JOIN 或 RIGHT JOIN 实现。| | NVL() | NVL(key,val) | IFNULL(key,val) | 如果该字段值为 NULL，则返回 val 值，否则返回该字段的值。| | NVL2() | NVL2(key, val1, val2) | IF(key is NULL, val1, val2) | 如果该字段值非 NULL，则返回 val1 值，否则返回 val2 值。| | DECODE() |

DECODE(key,val1,val2,val3)

DECODE(value,if1,val1,if2,val2,...,ifn,valn,val)

|

IF(key=val1,val2,val3)

CASE WHEN value=if1 THEN val1 WHEN value=if2 THEN val2,...,WHEN value=ifn THEN valn ELSE val END

|

如果该字段值等于 val1，则返回 val2，否则返回 val3。

当该字段值满足条件 1 (if1) 时，返回 val1，满足条件 2 (if2) 时，返回 val2，满足条件 3 (if3) 时，返回 val3。

| | 拼接字符串 a 和 b | 'a' || 'b' | CONCAT('a','b') | | | 获取字符串长度 | LENGTH(str) | CHAR_LENGTH(str) | | | 获取子串 | SUBSTR('abcdefg',0,2)= 'ab' SUBSTR('abcdefg',1,2)= 'ab' | SUBSTRING('abcdefg',0,2)= <⟷ ''SUBSTRING('abcdefg',1,2)= 'ab' |

Oracle 中起始位置 0 与 1 作用一样。

TiDB 中 0 开始获取的子串为空，若需从字符串的起始位置开始，则应从 1 开始。

| | 字符串在源字符串中的位置 | INSTR('abcdefg','b',1,1) | INSTR('abcdefg','b') | 从字符串 'abcdefg' 第一个字符开始查询，返回 'b' 字符串第一次出现的位置。| | 字符串在源字符串中的位置 | INSTR('stst','s <⟷ ',1,2) | LENGTH(SUBSTRING_INDEX('stst','s',2))+ 1 | 从字符串'stst' 第一个字符开始查找，返回's' 字符第二次出现的位置。| | 字符串在源字符串中的位置 | INSTR('abcabc','b',2,1) | LOCATE('b','abcabc',2) | 从字符串 'abcabc' 第二个字符开始查询，返回 'b' 字符第一次出现的位置。| | 列合并为行 | LISTAGG(CONCAT(<⟷ E.dimensionid,'---',E.DIMENSIONNAME),'***')within GROUP(ORDER BY DIMENSIONNAME) | GROUP_CONCAT(<⟷ CONCAT(E.dimensionid,'---',E.DIMENSIONNAME)ORDER BY DIMENSIONNAME SEPARATOR '***') | 将一列字段合并为一行并根据 *** 符号进行分割。| | ASCII 值转化为对应字符 | CHR(n) | CHAR(n) | Oracle 中制表符(CHR(9))、换行符(CHR(10))、回车符 (CHR(13)) 对应 TiDB 中的 CHAR(9)、CHAR(10)、CHAR(13)。 |

4.8.4.21.2 语法差异

本节介绍 Oracle 部分语法与 TiDB 的差异。

字符串语法

Oracle 中字符串只能使用单引号(')。例如 'a'。

TiDB 中字符串可以使用单引号(') 或双引号(" ")。例如 'a' 或 "a"。

NULL 与空字符串的区分

Oracle 中不区分 NULL 和空字符串 ''，即 NULL 与 '' 是等价的。

TiDB 中区分 NULL 和空字符串 ''。

INSERT 语句中读写同一张表

Oracle 支持 INSERT 语句中读写同一张表。例如：

```
INSERT INTO table1 VALUES (feild1,(SELECT feild2 FROM table1 WHERE...))
```

TiDB 不支持 INSERT 语句中读写同一张表。例如：

```
INSERT INTO table1 VALUES (feild1,(SELECT T.fields2 FROM table1 T WHERE...))
```

获取前 n 行数据

Oracle 通过 ROWNUM <= n 获取前 n 行数据。例如，ROWNUM <= 10。

TiDB 通过 LIMIT n 获取前 n 行数据。例如，LIMIT 10。Hibernate Query Language (HQL) 方式运行带 LIMIT 的 SQL 语句会出现错误，需要将 Hibernate 的运行方式改为 SQL 方式运行。

UPDATE 语句多表更新

Oracle 多表更新时不需要列出具体的字段更新关系。例如：

```
UPDATE test1 SET(test1.name,test1.age) = (SELECT test2.name,test2.age FROM test2 WHERE test2.id=
    ↔ test1.id)
```

TiDB 多表更新时需要在 SET 时把具体的字段更新关系都列出来。例如：

```
UPDATE test1,test2 SET test1.name=test2.name,test1.age=test2.age WHERE test1.id=test2.id
```

派生表别名

Oracle 多表查询时，派生表可以不起别名。例如：

```
SELECT * FROM (SELECT * FROM test)
```

TiDB 多表查询时，每一个派生出来的表都必须有一个自己的别名。例如：

```
SELECT * FROM (SELECT * FROM test) t
```

差集运算

Oracle 使用 MINUS 进行差集运算。例如：

```
SELECT * FROM t1 MINUS SELECT * FROM t2
```

TiDB 不支持 MINUS，需要改写为 EXCEPT 进行差集运算。例如：

```
SELECT * FROM t1 EXCEPT SELECT * FROM t2
```

注释语法

Oracle 中注释语法为 --注释，其中 -- 后面不需要空格。

TiDB 中注释语法为 -- 注释，其中 -- 后面需要有一个空格。

分页查询

Oracle 分页查询时 OFFSET m 表示跳过 m 行数据，FETCH NEXT n ROWS ONLY 表示取 n 条数据。例如：

```
SELECT * FROM tables OFFSET 0 ROWS FETCH NEXT 2000 ROWS ONLY
```

TiDB 使用 `LIMIT n OFFSET m` 等价改写 `OFFSET m ROWS FETCH NEXT n ROWS ONLY`。例如：

```
SELECT * FROM tables LIMIT 2000 OFFSET 0
```

`ORDER BY` 语句对 `NULL` 的排序规则

Oracle 中 `ORDER BY` 语句对 `NULL` 的排序规则：

- `ORDER BY COLUMN ASC` 时，`NULL` 默认被放在最后。
- `ORDER BY COLUMN DESC` 时，`NULL` 默认被放在最前。
- `ORDER BY COLUMN [ASC|DESC] NULLS FIRST` 时，强制 `NULL` 放在最前，非 `NULL` 的值仍然按声明顺序 `ASC|DESC` 进行排序。
- `ORDER BY COLUMN [ASC|DESC] NULLS LAST` 时，强制 `NULL` 放在最后，非 `NULL` 的值仍然按声明顺序 `ASC|DESC` 进行排序。

TiDB 中 `ORDER BY` 语句对 `NULL` 的排序规则：

- `ORDER BY COLUMN ASC` 时，`NULL` 默认被放在最前。
- `ORDER BY COLUMN DESC` 时，`NULL` 默认被放在最后。

下表是 Oracle 与 TiDB 中等价 `ORDER BY` 语句示例：

Oracle 中的 `ORDER BY`	TiDB 中的 `ORDER BY`
`SELECT * FROM t1 ORDER BY name NULLS FIRST;`	`SELECT * FROM t1 ORDER BY name;`
`SELECT * FROM t1 ORDER BY name DESC NULLS LAST;`	`SELECT * FROM t1 ORDER BY name DESC ;`
`SELECT * FROM t1 ORDER BY name DESC NULLS FIRST;`	`SELECT * FROM t1 ORDER BY ISNULL( name)DESC, name DESC;`
`SELECT * FROM t1 ORDER BY name ASC NULLS LAST;`	`SELECT * FROM t1 ORDER BY ISNULL( name), name;`

4.8.5 聚簇索引

聚簇索引(clustered index) 是 TiDB 从 v5.0 开始支持的特性，用于控制含有主键的表数据的存储方式。通过使用聚簇索引，TiDB 可以更好地组织数据表，从而提高某些查询的性能。有些数据库管理系统也将聚簇索引称为

“索引组织表”(index-organized tables)。

目前TiDB 中含有主键的表分为以下两类：

- NONCLUSTERED，表示该表的主键为非聚簇索引。在非聚簇索引表中，行数据的键由 TiDB 内部隐式分配的 _tidb_rowid 构成，而主键本质上是唯一索引，因此非聚簇索引表存储一行至少需要两个键值对，分别为
  - _tidb_rowid（键）- 行数据（值）
  - 主键列数据（键）- _tidb_rowid（值）
- CLUSTERED，表示该表的主键为聚簇索引。在聚簇索引表中，行数据的键由用户给定的主键列数据构成，因此聚簇索引表存储一行至少只要一个键值对，即
  - 主键列数据（键）- 行数据（值）

> 注意：
>
> TiDB 仅支持根据表的主键来进行聚簇操作。聚簇索引启用时，“主键” 和 “聚簇索引” 两个术语在一些情况下可互换使用。主键指的是约束（一种逻辑属性），而聚簇索引描述的是数据存储的物理实现。

4.8.5.1 使用场景

相较于非聚簇索引表，聚簇索引表在以下几个场景中，性能和吞吐量都有较大优势：

- 插入数据时会减少一次从网络写入索引数据。
- 等值条件查询仅涉及主键时会减少一次从网络读取数据。
- 范围条件查询仅涉及主键时会减少多次从网络读取数据。
- 等值或范围条件查询仅涉及主键的前缀时会减少多次从网络读取数据。

另一方面，聚簇索引表也存在一定的劣势：

- 批量插入大量取值相邻的主键时，可能会产生较大的写热点问题。
- 当使用大于 64 位的数据类型作为主键时，可能导致表数据需要占用更多的存储空间。该现象在存在多个二级索引时尤为明显。

4.8.5.2 使用方法

4.8.5.2.1 创建聚簇索引表

从 TiDB 版本 5.0 开始，要指定一个表的主键是否使用聚簇索引，可以在 CREATE TABLE 语句中将 CLUSTERED 或者 NONCLUSTERED 非保留关键字标注在 PRIMARY KEY 后面，例如：

```
CREATE TABLE t (a BIGINT PRIMARY KEY CLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT PRIMARY KEY NONCLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT KEY CLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT KEY NONCLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) CLUSTERED);
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) NONCLUSTERED);
```

注意，列定义中的 KEY 和 PRIMARY KEY 含义相同。

此外，TiDB 支持使用可执行的注释语法指定聚簇索引属性：

```
CREATE TABLE t (a BIGINT PRIMARY KEY /*T![clustered_index] CLUSTERED */, b VARCHAR(255));
CREATE TABLE t (a BIGINT PRIMARY KEY /*T![clustered_index] NONCLUSTERED */, b VARCHAR(255));
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /*T![clustered_index] CLUSTERED */);
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /*T![clustered_index] NONCLUSTERED */
    ↪ );
```

对于未显式指定该关键字的语句，默认行为受系统变量@@global.tidb_enable_clustered_index 影响。该变量有三个取值：

- OFF 表示所有主键默认使用非聚簇索引。
- ON 表示所有主键默认使用聚簇索引。
- INT_ONLY 此时的行为受配置项 alter-primary-key 控制。如果该配置项取值为 true，则所有主键默认使用非聚簇索引；如果该配置项取值为 false，则由单个整数类型的列构成的主键默认使用聚簇索引，其他类型的主键默认使用非聚簇索引。

系统变量 @@global.tidb_enable_clustered_index 本身的默认值为 ON。

4.8.5.2.2 添加、删除聚簇索引

目前TiDB 不支持在建表之后添加或删除聚簇索引，也不支持聚簇索引和非聚簇索引的互相转换。例如：

```
ALTER TABLE t ADD PRIMARY KEY(b, a) CLUSTERED; -- 暂不支持
ALTER TABLE t DROP PRIMARY KEY;      -- 如果主键为聚簇索引，则不支持
ALTER TABLE t DROP INDEX `PRIMARY`; -- 如果主键为聚簇索引，则不支持
```

4.8.5.2.3 添加、删除非聚簇索引

TiDB 支持在建表之后添加或删除非聚簇索引。此时可以选择显式指定 NONCLUSTERED 关键字或省略关键字：

```
ALTER TABLE t ADD PRIMARY KEY(b, a) NONCLUSTERED;
ALTER TABLE t ADD PRIMARY KEY(b, a); -- 不指定关键字，则为非聚簇索引
ALTER TABLE t DROP PRIMARY KEY;
ALTER TABLE t DROP INDEX `PRIMARY`;
```

4.8.5.2.4 查询主键是否为聚簇索引

可通过以下方式来确定一张表的主键是否使用了聚簇索引：

- 执行语句 SHOW CREATE TABLE。
- 执行语句 SHOW INDEX FROM。
- 查询系统表 information_schema.tables 中的 TIDB_PK_TYPE 列。

通过 SHOW CREATE TABLE 查看，PRIMARY KEY 的属性可能为 CLUSTERED 或 NONCLUSTERED：

```
mysql> SHOW CREATE TABLE t;
+--
    <-->------+
    <-->
| Table | Create Table
    <-->
    <--> |
+--
    <-->      +
    <-->
| t     | CREATE TABLE `t` (
  `a` bigint(20) NOT NULL,
  `b` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    <-->      +
    <>------
1 row in set (0.01 sec)
```

通过 SHOW INDEX FROM 查看，Clustered 一列可能的结果为 Yes 或 No：

```
mysql> SHOW INDEX FROM t;
+--
    <> -----+------------+----------+--------------+-------------+-----------+-------------+----------+ ----------------+
    <>
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    <>  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
    <> Clustered |
+--
    <> -----+------------+----------+--------------+-------------+-----------+-------------+----------+ ----------------+
    <>
| t     |          0 | PRIMARY  |            1 | a           | A         |           0 |     NULL
    <> | NULL   |      | BTREE      |         |               | YES     | NULL       | YES
    <>      |
+--
    <> -----+------------+----------+--------------+-------------+-----------+-------------+----------+ ----------------+
    <>
```

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

查询 information_schema.tables 系统表中的 TIDB_PK_TYPE 列，可能的结果为 CLUSTERED 或 NONCLUSTERED：

```
mysql> SELECT TIDB_PK_TYPE FROM information_schema.tables WHERE table_schema = 'test' AND
    <-> table_name = 't';
+---------------+
| TIDB_PK_TYPE |
+---------------+
| CLUSTERED    |
+---------------+
1 row in set (0.03 sec)
```

4.8.5.3 限制

目前TiDB 的聚簇索引具有以下几类限制：

- 明确不支持且没有支持计划的使用限制：
  - 不支持与SHARD_ROW_ID_BITS 一起使用；PRE_SPLIT_REGIONS 在聚簇索引表上不生效。
  - 不支持对聚簇索引表进行降级。如需降级，请使用逻辑备份工具迁移数据。
- 尚未支持，但未来有计划支持的使用限制：
  - 尚未支持通过 ALTER TABLE 语句增加、删除、修改聚簇索引。
- 特定版本的限制：
  - 在 v5.0 版本中，聚簇索引不支持与 TiDB Binlog 一起使用。开启 TiDB Binlog 后，TiDB 只允许创建单个整数列作为主键的聚簇索引；已创建的聚簇索引表的数据插入、删除和更新动作不会通过TiDB Binlog 同步到下游。如需同步聚簇索引表，请升级至 v5.1 版本或使用TiCDC。

开启TiDB Binlog 之后，要创建的聚簇索引如果不是由单个整数列构成，会报以下错误：

```
mysql> CREATE TABLE t (a VARCHAR(255) PRIMARY KEY CLUSTERED);
ERROR 8200 (HY000): Cannot create clustered index table when the binlog is ON
```

与 SHARD_ROW_ID_BITS 一起使用时会报以下错误：

```
mysql> CREATE TABLE t (a VARCHAR(255) PRIMARY KEY CLUSTERED) SHARD_ROW_ID_BITS = 3;
ERROR 8200 (HY000): Unsupported shard_row_id_bits for table with primary key as row id
```

4.8.5.4 兼容性

4.8.5.4.1 升降级兼容性

TiDB 支持对聚簇索引表的升级兼容，但不支持降级兼容，即高版本TiDB 聚簇索引表的数据在低版本TiDB 上不可用。

聚簇索引在TiDB v3.0 和 v4.0 中已完成部分支持，当表中存在单个整数列作为主键时默认启用，即：

- 表设置了主键
- 主键只有一列
- 主键的数据类型为整数类型

TiDB v5.0 完成了所有类型主键的支持，但默认行为与TiDB v3.0 和 v4.0 保持一致。要修改默认行为，请设置系统变量 `@@tidb_enable_clustered_index` 为 `ON` 或 `OFF`。

4.8.5.4.2 MySQL 兼容性

TiDB 支持使用可执行注释的语法来包裹 `CLUSTERED` 或 `NONCLUSTERED` 关键字，且 `SHOW CREATE TABLE` 的结果均包含TiDB 特有的可执行注释，这些注释在 MySQL 或低版本的TiDB 中会被忽略。

4.8.5.4.3 TiDB 数据迁移工具兼容性

聚簇索引仅与 v5.0 及以后版本的以下数据迁移工具兼容：

- 备份与恢复工具 BR、 Dumpling、 TiDB Lightning。
- 数据迁移和同步工具 DM、 TiCDC。

v5.0 的 BR 不能通过备份恢复将非聚簇索引表转换成聚簇索引表，反之亦然。

4.8.5.4.4 与TiDB 其他特性的兼容性

在非单整数列作为主键的表中，从非聚簇索引变为聚簇索引之后，在 v5.0 之前版本的 TiDB 能够执行的 `SPLIT TABLE BY/BETWEEN` 语句在 v5.0 及以后版本的TiDB 上不再可用，原因是行数据键的构成发生了变化。在聚簇索引表上执行 `SPLIT TABLE BY/BETWEEN` 时需要依据主键列指定值，而不是指定一个整数值。例如：

```
mysql> create table t (a int, b varchar(255), primary key(a, b) clustered);
Query OK, 0 rows affected (0.01 sec)

mysql> split table t between (0) and (1000000) regions 5;
ERROR 1105 (HY000): Split table region lower value count should be 2

mysql> split table t by (0), (50000), (100000);
ERROR 1136 (21S01): Column count doesn't match value count at row 0

mysql> split table t between (0, 'aaa') and (1000000, 'zzz') regions 5;
+--------------------+------------------------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+------------------------------------+
```

```
|                   4 |                     1 |
+---------------------+-----------------------+
1 row in set (0.00 sec)

mysql> split table t by (0, ''), (50000, ''), (100000, '');
+---------------------+-----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+---------------------+-----------------------+
|                   3 |                     1 |
+---------------------+-----------------------+
1 row in set (0.01 sec)
```

AUTO_RANDOM 属性只能在聚簇索引表上使用。在非聚簇索引上使用 AUTO_RANDOM 会报以下错误：

```
mysql> create table t (a bigint primary key nonclustered auto_random);
ERROR 8216 (HY000): Invalid auto random: column a is not the integer primary key, or the primary
    <-> key is nonclustered
```

4.8.6 约束

TiDB 支持的约束与 MySQL 的基本相同。

4.8.6.1 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。

4.8.6.2 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;
```

4.8.6.3 唯一约束

唯一约束是指唯一索引和主键列中所有的非空值都是唯一的。

4.8.6.3.1 乐观事务

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');
```

乐观事务模式下且 tidb_constraint_check_in_place=OFF：

```
BEGIN OPTIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
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 'users.username'
```

在以上乐观事务的示例中，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 ON 停用此行为（该变量仅适用于乐观事务，悲观事务需通过 tidb_constraint_check_in_place_pessimistic 设置）。当 tidb_constraint_check_in_place 设置为 ON 时，TiDB 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 = ON;
```

```
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 'users.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。

4.8.6.3.2 悲观事务

在TiDB 的悲观事务中，默认在执行任何一条需要插入或更新唯一索引的 SQL 语句时都会进行唯一约束检查：

```
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');

BEGIN PESSIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'users.username'
```

对于悲观事务，你可以设置变量tidb_constraint_check_in_place_pessimistic 为 OFF 来推迟唯一约束检查，到下一次对该唯一索引项加锁时或事务提交时再进行检查，同时也跳过对该悲观锁加锁，以获得更好的性能。此时需要注意：

- 由于推迟了唯一约束检查，TiDB 可能会读取到不满足唯一约束的结果，执行 COMMIT 语句时可能返回 Duplicate entry 错误。返回该错误时，TiDB 会回滚当前事务。

  下面这个例子跳过了对 bill 的加锁，因此TiDB 可能读到不满足唯一性约束的结果：

  ```
  SET tidb_constraint_check_in_place_pessimistic = OFF;
  BEGIN PESSIMISTIC;
  INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill'); -- Query OK, 3 rows
      <-> affected
  SELECT * FROM users FOR UPDATE;
  ```

  TiDB 读到了不满足唯一性约束的结果：有两个 bill。

  ```
  +----+----------+
  | id | username |
  +----+----------+
  | 1  | dave     |
  | 2  | sarah    |
  | 3  | bill     |
  | 7  | jane     |
  | 8  | chris    |
  | 9  | bill     |
  +----+----------+
  ```

  此时，如果提交事务，TiDB 将进行唯一约束检查，报出 Duplicate entry 错误并回滚事务。

  ```
  COMMIT;
  ```

  ```
  ERROR 1062 (23000): Duplicate entry 'bill' for key 'users.username'
  ```

- 关闭该变量时，如果在事务中写入数据，执行 COMMIT 语句可能会返回 Write conflict 错误。返回该错误时，TiDB 会回滚当前事务。

  在下面这个例子中，当有并发事务写入时，跳过悲观锁导致事务提交时报出 Write conflict 错误并回滚。

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
username VARCHAR(60) NOT NULL,
UNIQUE KEY (username)
);

SET tidb_constraint_check_in_place_pessimistic = OFF;
BEGIN PESSIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill'); -- Query OK, 3 rows
    <-> affected
```

然后另一个会话中写入了 bill：

```
INSERT INTO users (username) VALUES ('bill'); -- Query OK, 1 row affected
```

在第一个会话中提交时，TiDB 会报出 Write conflict 错误。

```
COMMIT;
```

```
ERROR 9007 (HY000): Write conflict, txnStartTS=435688780611190794, conflictStartTS
    <-> =435688783311536129, conflictCommitTS=435688783311536130, key={tableID=74, indexID
    <-> =1, indexValues={bill, }} primary={tableID=74, indexID=1, indexValues={bill, }},
    <-> reason=LazyUniquenessCheck [try again later]
```

- 关闭该变量时，如果多个悲观事务之间存在写冲突，悲观锁可能会在其它悲观事务提交时被强制回滚，因此产生 PessimisticLockNotFound 错误。发生该错误时，说明该业务不适合推迟悲观事务的唯一约束检查，应考虑调整业务避免冲突，或在发生错误后重试事务。
- 关闭该变量会导致悲观事务中可能报出错误 8147: LazyUniquenessCheckFailure。

> 注意：
> 返回 8147 错误时当前事务回滚。

下面的例子在INSERT 语句执行时跳过了一次加锁后，在 DELETE 语句执行时对该唯一索引加锁并检查，即会在该语句报错：

```
SET tidb_constraint_check_in_place_pessimistic = OFF;
BEGIN PESSIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill'); -- Query OK, 3 rows
    <-> affected
DELETE FROM users where username = 'bill';
```

```
ERROR 8147 (23000): transaction aborted because lazy uniqueness check is enabled and an
    <-> error occurred: [kv:1062]Duplicate entry 'bill' for key 'users.username'
```

- 关闭该变量时，1062 Duplicate entry 报错不一定是当前执行的 SQL 语句所发生的错误。因此，在一个事务操作多个表，且这些表有同名索引时，请注意 1062 报错信息中提示的是哪个表的哪个索引发生了错误。

4.8.6.4 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，TiDB 目前仅支持对 NONCLUSTERED 的主键进行添加和删除操作。例如：

```
CREATE TABLE t5 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b) CLUSTERED);
ALTER TABLE t5 DROP PRIMARY KEY;
```

```
ERROR 8200 (HY000): Unsupported drop primary key when the table is using clustered index
```

```
CREATE TABLE t5 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b) NONCLUSTERED);
ALTER TABLE t5 DROP PRIMARY KEY;
```

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

要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

4.8.6.5 外键约束

> **注意：**
>
> TiDB 从 v6.6.0 开始支持外键约束。在 v6.6.0 之前，TiDB 支持创建和删除外键约束，但外键约束并不生效。升级到 v6.6.0 后，可以先删除不生效的外键后再创建外键使外键约束生效。

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);
```

4.8.7 生成列

本文介绍生成列的概念以及用法。

4.8.7.1 生成列的基本概念

与一般的列不同，生成列的值由列定义中表达式计算得到。对生成列进行插入或更新操作时，并不能对之赋值，只能使用 DEFAULT。

生成列包括存储生成列和虚拟生成列。存储生成列会将计算得到的值存储起来，在读取时不需要重新计算。虚拟生成列不会存储其值，在读取时会重新计算。存储生成列和虚拟生成列相比，前者在读取时性能更好，但是要占用更多的磁盘空间。

无论是存储生成列还是虚拟列，都可以在其上面建立索引。

4.8.7.2 生成列的应用

生成列的主要的作用之一：从JSON 数据类型中解出数据，并为该数据建立索引。

MySQL 5.7 及 TiDB 都不能直接为JSON 类型的列添加索引，即不支持在如下表结构中的 address_info 上建立索引：

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

如果要为JSON 列某个字段添加索引，可以抽取该字段为生成列。

以 city 这一 address_info 中的字段为例，可以为其建立一个虚拟生成列并添加索引：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))), -- 虚拟生成列
    -- city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) VIRTUAL, --
        <-> 虚拟生成列
    -- city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) STORED, --
        <-> 存储生成列
    KEY (city)
);
```

该表中，city 列是一个虚拟生成列。并且在该列上建立了索引。以下语句能够利用索引加速语句的执行速度：

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

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

```
+                                  +         +           +                                  +
    <->
```

```
| id                              | estRows | task      | access object                  |
    <-> operator info                                                  |
+                                 +         +           +                                +----------------------------------
    <->
| Projection_4                    | 10.00   | root      |                                | test.
    <-> person.name, test.person.id                                |
| └─IndexLookUp_10                | 10.00   | root      |                                 |
    <->                                                          |
|   ├─IndexRangeScan_8(Build)     | 10.00   | cop[tikv] | table:person, index:city(city) | range
    <-> :["Beijing","Beijing"], keep order:false, stats:pseudo |
|   └─TableRowIDScan_9(Probe)     | 10.00   | cop[tikv] | table:person                    | keep
    <-> order:false, stats:pseudo                                  |
+                                 +         +           +                                +----------------------------------
    <->
```

从执行计划中，可以看出使用了 city 这个索引来读取满足 city = 'Beijing' 这个条件的行的 HANDLE，再用这个 HANDLE 来读取该行的数据。

如果 $.city 路径中无数据，则 JSON_EXTRACT 返回 NULL。如果想增加约束，city 列必须是 NOT NULL，则可按照以下方式定义虚拟生成列：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) NOT NULL,
    KEY (city)
);
```

4.8.7.3 生成列在INSERT和UPDATE语句中的行为

INSERT 和 UPDATE 语句都会检查生成列计算得到的值是否满足生成列的定义。未通过有效性检测的行会返回错误：

```
INSERT INTO person (name, address_info) VALUES ('Morgan', JSON_OBJECT('Country', 'Canada'));
```

```
ERROR 1048 (23000): Column 'city' cannot be null
```

4.8.7.4 索引生成列替换

当查询中出现的某个表达式与一个含索引的生成列严格同等时，TiDB 会将这个表达式替换为对应的生成列，这样就可以在生成查询计划时考虑使用这个索引。

下面的例子为 a+1 这个表达式创建生成列并添加索引，从而加速查询。其中，a 的列类型是int，而 a+1 的列类型是bigint。如果将生成列的类型改为int，就不会发生替换。关于类型转换的规则，可以参见表达式求值的类型转换。

```
create table t(a int);
desc select a+1 from t where a+1=3;
```

```
+--
    <--> ------------------------+----------+-----------+---------------+--------------------------------------------------+
    <-->
| id                          | estRows  | task      | access object | operator info
    <-->                 |
+--
    <--> ------------------------+----------+-----------+---------------+--------------------------------------------------+
    <-->
| Projection_4                | 8000.00  | root      |               | plus(test.t.a, 1)->Column#3
    <-->    |
| └─TableReader_7             | 8000.00  | root      |               | data:Selection_6
    <-->                |
|   └─Selection_6             | 8000.00  | cop[tikv] |               | eq(plus(test.t.a, 1), 3)
    <-->       |
|     └─TableFullScan_5       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    <--> pseudo |
+--
    <--> ------------------------+----------+-----------+---------------+--------------------------------------------------+
    <-->
4 rows in set (0.00 sec)
```

```
alter table t add column b bigint as (a+1) virtual;
alter table t add index idx_b(b);
desc select a+1 from t where a+1=3;
```

```
+--
    <-->                        +          +           +                             +
    <-->
| id                          | estRows | task      | access object            | operator info
    <-->                                |
+--
    <-->                        +          +           +                             +
    <--> ----------------------- ---------- ----------- --------------------------
| IndexReader_6               | 10.00    | root      |                          | index:IndexRangeScan_5
    <-->                        |
| └─IndexRangeScan_5          | 10.00    | cop[tikv] | table:t, index:idx_b(b) | range:[3,3], keep
    <--> order:false, stats:pseudo |
+--
    <-->                        +          +           +                             +
    <-->
2 rows in set (0.01 sec)
```

> 注意：
>
> 若待替换的表达式类型和生成列类型都是字符类型，但两种类型长度不同时，仍可通过将系统变量tidb_enable_unsafe_substitute 设置为 ON 来允许其替换。配置该系统变量时，需要保证生成列计算得到的值严格满足生成列的定义，否则，可能因为长度不同，导致数据截断得到错误的结果。详情见GitHub issue #35490。

4.8.7.5 生成列的局限性

目前生成列有以下局限性：

- 不能通过 ALTER TABLE 增加存储生成列；
- 不能通过 ALTER TABLE 将存储生成列转换为普通列，也不能将普通列转换成存储生成列；
- 不能通过 ALTER TABLE 修改存储生成列的生成列表达式；
- 并未支持所有的JSON 函数；
- 目前仅当生成列是虚拟生成列时索引生成列替换规则有效，暂不支持将表达式替换为存储生成列，但仍然可以通过直接使用该生成列本身来使用索引。

4.8.8 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。

4.8.8.1 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，无法被插入到表中，并且返回错误。
- TRADITIONAL：采用此模式使 TiDB 的行为像“传统”SQL 数据库系统，当在列中插入不正确的值时“给出错误而不是警告”，一旦发现错误立即放弃 INSERT 或 UPDATE。

4.8.8.2 SQL mode 列表，如下

名称	含义
PIPES_AS_CONCAT	将 “||” 视为字符串连接操作符 (+) (同 CONCAT()), 而不视为 OR (支持)

名称	含义
ANSI_QUOTES	将"视为识别符，如果启用ANSI_QUOTES，只单引号内的会被认为是String Liter-als，双引号被解释为识别符，因此不能用双引号来引用字符串（支持）

名称	含义
IGNORE_SPACE	若开启该模式，系统忽略空格。例如：“user”和“user “是相同的（支持）

名称	含义
ONLY_FULL_GROUP_BY	如果 GROUP BY 出现的列并没有在 SELECT，HAVING，ORDER BY 中出现，此 SQL 不合法，因为不在 GROUP BY 中的列被查询展示出来不符合正常现象(支持)

名称	含义
NO_UNSIGNED_SUBTRACTION	在减运算中，如果某个操作数没有符号，不要将结果标记为 UNSIGNED（支持）
NO_DIR_IN_CREATE	创建表时，忽视所有 INDEX DIRECTORY 和 DATA DIRECTORY 指令，该选项仅对从复制服务器有用（仅语法支持）

名称	含义
NO_KEY_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_FIELD_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_TABLE_OPTIONS	使用 SHOW 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_AUTO_VALUE_ON_ZERO	若启用该模式，在 AUTO_INCREMENT 列的处理传入的值是 0 或者具体数值时系统直接将该值写入此列，传入 NULL 时系统自动生成下一个序列号（支持）
NO_BACKSLASH_ESCAPES	若启用该模式，\ 反斜杠符号仅代表它自己（支持）

名称	含义
STRICT_TRANS_TABLES	对于事务存储引擎启用严格模式，insert非法值之后，回滚整条语句（支持）
STRICT_ALL_TABLES	对于事务型表，写入非法值之后，回滚整个事务语句（支持）

名称	含义
NO_ZERO_DATE	在严格模式，不接受月或日部分为 0 的日期。如果使用 IGNORE 选项，我们为类似的日期插入’0000-00-00’。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
NO_ZERO_DATE	在严格模式，不要将’0000-00-00’做为合法日期。你仍然可以用 IGNORE 选项插入零日期。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
ALLOW_INVALID_DATES	不检查全部日期的合法性，仅检查月份值在 1 到 12 及日期值在 1 到 31 之间，仅适用于 DATE 和 DATA-TIME 列，TIMES-TAMP 列需要全部检查其合法性（支持）

名称	含义
ERROR_FOR_DIVISION_BY_ZERO	若启用该模式，在 INSERT 或 UPDATE 过程中，被除数为 0 值时，系统产生错误若未启用该模式，被除数为 0 值时，系统产生警告，并用 NULL 代替（支持）

名称	含义
NO_AUTO_CREATE_USER	防止 GRANT 自动创建新用户，但指定密码除外（支持）

名称	含义
HIGH_NOT_PRECEDENCE	NOT 操作符的优先级是表达式。例如：NOT a BETWEEN b AND c 被解释为 NOT (a BETWEEN b AND c)。在部分旧版本 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）
MySQL323	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）

4.8.9 表属性

表属性是TiDB 从 5.3.0 版本开始引入的新特性，用于为表或分区添加特定的属性，以对表或分区执行相应属性对应的操作，例如可以利用表属性控制Region 的合并。

注意：

- 目前TiDB 仅支持为表或分区添加 merge_option 属性，用于控制Region 合并。该属性仅能处理部分热点问题。如需了解更多的热点问题处理相关内容，请参阅 TiDB 热点问题处理。
- 当使用TiDB Binlog 或 TiCDC 进行同步或者使用 BR 进行增量备份时，同步和备份会跳过设置表属性的 DDL 语句。如需在下游或者备份集群使用表属性，需要在下游或者备份集群手动执行该 DDL 语句以设置表属性。

4.8.9.1 使用方法

表属性为 key=value 的形式，多个属性需要用逗号分隔。具体示例如下，其中t 为所要修改的表名，p 为所要修改的分区名，[]内部为可选项。

- 设置表或分区属性

```
ALTER TABLE t [PARTITION p] ATTRIBUTES [=] 'key=value[, key1=value1...]';
```

- 重置表或分区属性

```
ALTER TABLE t [PARTITION p] ATTRIBUTES [=] DEFAULT;
```

- 查看全部表及分区属性

```
SELECT * FROM information_schema.attributes;
```

- 查看某一张表或分区配置的属性

```
SELECT * FROM information_schema.attributes WHERE id='schema/t[/p]';
```

- 查看拥有某属性的所有表及分区

```
SELECT * FROM information_schema.attributes WHERE attributes LIKE '%key%';
```

4.8.9.2 覆盖关系

为分区表配置的属性会对表的所有分区生效。一种例外情况是，如果分区表和分区都配置了相同属性但属性值不同，分区属性将覆盖分区表属性。例如，当分区表t 配置属性 key=value，同时分区 p 配置属性 key=value1 时：

```
ALTER TABLE t ATTRIBUTES[=]'key=value';
ALTER TABLE t PARTITION p ATTRIBUTES[=]'key=value1';
```

分区p 实际生效的属性为 key=value1。

4.8.9.3 使用表属性控制Region 合并

4.8.9.3.1 使用场景

当写入或读取数据存在热点时，可以使用表属性控制Region 合并，通过为表或分区添加 `merge_option` 属性，将其设置为 `deny` 来解决。以下介绍了两种使用场景。

新建表或分区的写入热点问题

在对某张新建表或某个新建分区写入数据存在热点问题时，通常需要使用分裂打散Region 的操作避免写入热点，但由于新建表或分区的分裂操作实际产生的是空Region，如果分裂打散操作距离写入存在一定时间间隔，则 Region 可能会被合并，从而导致无法真正规避写入热点问题。此时可以为表或分区添加 `merge_option` 属性，设置为 `deny` 来解决问题。

只读场景下周期性读热点问题

在只读场景下，如果是通过手动分裂Region 缓解某张表或分区的周期性读热点问题，且不希望热点消失后手动分裂的Region 被合并。此时可以为表或分区添加 `merge_option` 属性，设置为 `deny` 来解决问题。

4.8.9.3.2 使用方法

使用方法如下，其中`t` 为所要修改的表名，`p` 为所要修改的分区名。

- 禁止属于某个表的Region 被合并

```
ALTER TABLE t ATTRIBUTES 'merge_option=deny';
```

- 允许属于某个表的Region 被合并

```
ALTER TABLE t ATTRIBUTES 'merge_option=allow';
```

- 重置某个表的属性

```
ALTER TABLE t ATTRIBUTES DEFAULT;
```

- 禁止属于某个分区的Region 被合并

```
ALTER TABLE t PARTITION p ATTRIBUTES 'merge_option=deny';
```

- 允许属于某个分区的Region 被合并

```
ALTER TABLE t PARTITION p attributes 'merge_option=allow';
```

- 查看所有配置了 `merge_option` 属性的表或分区

```
SELECT * FROM information_schema.attributes WHERE attributes LIKE '%merge_option%';
```

4.8.9.3.3 覆盖关系

```
ALTER TABLE t ATTRIBUTES 'merge_option=deny';
ALTER TABLE t PARTITION p ATTRIBUTES 'merge_option=allow';
```

同时配置上述两个属性时，实际分区p 的 Region 可以被合并。当分区的属性被重置时，分区p 则会继承表t 的属性，Region 无法被合并。

> 注意：
>
> - 如果目前只存在分区表的属性，即使配置 merge_option=allow，分区也会默认按照实际分区数量切分成多个Region。如需合并所有Region，则需要重置该分区表的属性。
> - 使用该属性需要注意 PD 的参数 split-merge-interval 的配置。如果没有配置 merge_option，Region 在超过 split-merge-interval 指定的时间后满足条件即可合并。如果配置了 merge_option，则超过指定时间后会根据 merge_option 的配置情况再决定是否可以合并。

4.8.10 事务

4.8.10.1 TiDB 事务概览

TiDB 支持分布式事务，提供乐观事务与悲观事务两种事务模式。TiDB 3.0.8 及以后版本，TiDB 默认采用悲观事务模式。

本文主要介绍涉及事务的常用语句、显式/隐式事务、事务的隔离级别和惰性检查，以及事务大小的限制。

常用的变量包括autocommit、tidb_disable_txn_auto_retry、tidb_retry_limit 以及tidb_txn_mode。

> 注意：
>
> 变量tidb_disable_txn_auto_retry 和tidb_retry_limit 仅适用于乐观事务，不适用于悲观事务。

4.8.10.1.1 常用事务语句

开启事务

要显式地开启一个新事务，既可以使用BEGIN 语句，也可以使用START TRANSACTION 语句，两者效果相同。

语法：

```
BEGIN;
```

```
START TRANSACTION;
```

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

```
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY;
```

如果执行以上语句时，当前 Session 正处于一个事务的中间过程，那么系统会先自动提交当前事务，再开启一个新的事务。

> 注意：
>
> 与 MySQL 不同的是，TiDB 在执行完上述语句后即会获取当前数据库快照，而 MySQL 的 BEGIN 和 START TRANSACTION 是在开启事务后的第一个从InnoDB 读数据的 SELECT 语句（非 SELECT <—> FOR UPDATE）后获取快照，START TRANSACTION WITH CONSISTENT SNAPSHOT 是语句执行时获取快照。因此，TiDB 中的 BEGIN、START TRANSACTION 和 START TRANSACTION WITH CONSISTENT <—> SNAPSHOT 都等效为 MySQL 中的 START TRANSACTION WITH CONSISTENT SNAPSHOT。

提交事务

COMMIT 语句用于提交TiDB 在当前事务中进行的所有修改。

语法：

```
COMMIT;
```

> 建议：
>
> 启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

回滚事务

ROLLBACK 语句用于回滚并撤销当前事务的所有修改。

语法：

```
ROLLBACK;
```

如果客户端连接中止或关闭，也会自动回滚该事务。

4.8.10.1.2 自动提交

为满足 MySQL 兼容性的要求，在默认情况下，TiDB 将在执行语句后立即进行autocommit（自动提交）。

举例：

```
mysql> CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  pad1 VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            1 |
+--------------+
1 row in set (0.00 sec)

mysql> INSERT INTO t1 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM t1;
+----+-------+
| id | pad1 |
+----+-------+
|  1 | test |
+----+-------+
1 row in set (0.00 sec)
```

以上示例中，ROLLBACK 语句没产生任何效果。由于 INSERT 语句是在自动提交的情况下执行的，等同于以下单语句事务：

```
START TRANSACTION;
INSERT INTO t1 VALUES (1, 'test');
COMMIT;
```

如果已显式地启动事务，则不适用自动提交。以下示例，ROLLBACK 语句成功撤回了 INSERT 语句：

```
mysql> CREATE TABLE t2 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  pad1 VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.10 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
```

```
+---------------+
| 1             |
+---------------+
1 row in set (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t2 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

autocommit 是一个系统变量，可以基于 Session 或 Global 进行修改。

举例：

```
SET autocommit = 0;
```

```
SET GLOBAL autocommit = 0;
```

4.8.10.1.3 显式事务和隐式事务

> 注意：
>
> 有些语句是隐式提交的。例如，执行 [BEGIN|START TRANCATION] 语句时，TiDB 会隐式提交上一个事务，并开启一个新的事务以满足 MySQL 兼容性的需求。详情参见implicit commit。

TiDB 可以显式地使用事务（通过 [BEGIN|START TRANSACTION]/COMMIT 语句定义事务的开始和结束）或者隐式地使用事务(SET autocommit = 1)。

在自动提交状态下，使用 [BEGIN|START TRANSACTION] 语句会显式地开启一个事务，同时也会禁用自动提交，使隐式事务变成显式事务。直到执行 COMMIT 或 ROLLBACK 语句时才会恢复到此前默认的自动提交状态。

对于 DDL 语句，会自动提交并且不能回滚。如果运行 DDL 的时候，正在一个事务的中间过程中，会先自动提交当前事务，再执行 DDL。

4.8.10.1.4 惰性检查

执行 DML 语句时，乐观事务默认不会检查主键约束或唯一约束，而是在 COMMIT 事务时进行这些检查。

举例：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
INSERT INTO t1 VALUES (1);
BEGIN OPTIMISTIC;
INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
INSERT INTO t1 VALUES (2);
COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
```

```
mysql> CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.10 sec)

mysql> INSERT INTO t1 VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> BEGIN OPTIMISTIC;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (2);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
ERROR 1062 (23000): Duplicate entry '1' for key 't1.PRIMARY'
mysql> SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

惰性检查优化通过批处理约束检查并减少网络通信来提升性能。可以通过设置`tidb_constraint_check_in_place <-> = ON`禁用该行为。

> 注意：
>
> - 本优化仅适用于乐观事务。
> - 本优化仅对普通的 `INSERT` 语句生效，对 `INSERT IGNORE` 和 `INSERT ON DUPLICATE KEY <-> UPDATE` 不会生效。

4.8.10.1.5 语句回滚

TiDB 支持语句执行失败后的原子性回滚。如果语句报错，则所做的修改将不会生效。该事务将保持打开状态，并且在发出 COMMIT 或 ROLLBACK 语句之前可以进行其他修改。

```
CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
BEGIN;
INSERT INTO test VALUES (1);
INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
INSERT INTO test VALUES (3);
COMMIT;
SELECT * FROM test;
```

```
mysql> CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.09 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO test VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
ERROR 1146 (42S02): Table 'test.tset' doesn't exist
mysql> INSERT INTO test VALUES (1),(2); -- 违反 PRIMARY KEY 约束，语句不生效。
ERROR 1062 (23000): Duplicate entry '1' for key 'test.PRIMARY'
mysql> INSERT INTO test VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM test;
+----+
| id |
+----+
|  1 |
|  3 |
+----+
2 rows in set (0.00 sec)
```

以上例子中，INSERT 语句执行失败之后，事务保持打开状态。最后的 INSERT 语句执行成功，并且提交了修改。

4.8.10.1.6 事务限制

由于底层存储引擎的限制，TiDB 要求单行不超过 6 MB。可以将一行的所有列根据类型转换为字节数并加和来估算单行大小。

TiDB 同时支持乐观事务与悲观事务，其中乐观事务是悲观事务的基础。由于乐观事务是先将修改缓存在私有内存中，因此，TiDB 对于单个事务的容量做了限制。

TiDB 中，单个事务的总大小默认不超过 100 MB，这个默认值可以通过配置文件中的配置项 `txn-total-size-limit` 进行修改，最大支持 1 TB。单个事务的实际大小限制还取决于服务器剩余可用内存的大小，执行事务时 TiDB 进程的内存消耗相对于事务大小会存在一定程度的放大，最大可能达到提交事务大小的 2 到 3 倍以上。

在 4.0 以前的版本，TiDB 限制了单个事务的键值对的总数量不超过 30 万条，从 4.0 版本起 TiDB 取消了这项限制。

> 注意：
>
> 通常，用户会开启 TiDB Binlog 将数据向下游进行同步。某些场景下，用户会使用消息中间件来消费同步到下游的 binlog，例如 Kafka。
>
> 以 Kafka 为例，Kafka 的单条消息处理能力的上限是 1 GB。因此，当把 `txn-total-size-limit` 设置为 1 GB 以上时，可能出现事务在 TiDB 中执行成功，但下游 Kafka 报错的情况。为避免这种情况出现，请用户根据最终消费者的限制来决定 `txn-total-size-limit` 的实际大小。例如：下游使用了 Kafka，则 `txn-total-size-limit` 不应超过 1 GB。

4.8.10.1.7 因果一致性事务

> 注意：
>
> 因果一致性事务只在启用 Async Commit 特性和一阶段提交特性时生效。关于这两个特性的启用情况，请参见 `tidb_enable_async_commit` 系统变量介绍和 `tidb_enable_1pc` 系统变量介绍。

TiDB 支持开启因果一致性的事务。因果一致性的事务在提交时无需向 PD 获取时间戳，所以提交延迟更低。开启因果一致性事务的语法为：

```
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY;
```

默认情况下，TiDB 保证线性一致性。在线性一致性的情况下，如果事务 2 在事务 1 提交完成后提交，逻辑上事务 2 就应该在事务 1 后发生。

因果一致性弱于线性一致性。在因果一致性的情况下，只有事务 1 和事务 2 加锁或写入的数据有交集时（即事务 1 和事务 2 存在数据库可知的因果关系时），才能保证事务的提交顺序与事务的发生顺序保持一致。目前暂不支持传入数据库外部的因果关系。

采用因果一致性的两个事务有以下特性：

- 有潜在因果关系的事务之间的逻辑顺序与物理提交顺序一致

- 无因果关系的事务之间的逻辑顺序与物理提交顺序不保证一致
- 不加锁的读取不产生因果关系

有潜在因果关系的事务之间的逻辑顺序与物理提交顺序一致

假设事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY	START TRANSACTION WITH CAUSAL CONSISTENCY ONLY
x = SELECT v FROM t WHERE id = 1 FOR UPDATE	
UPDATE t set v = $(x + 1) WHERE id = 2	
COMMIT	
	UPDATE t SET v = 2 WHERE id = 1
	COMMIT

上面的例子中，事务 1 对 `id = 1` 的记录加了锁，事务 2 的事务对 `id = 1` 的记录进行了修改，所以事务 1 和事务 2 有潜在的因果关系。所以即使用因果一致性开启事务，只要事务 2 在事务 1 提交成功后才提交，逻辑上事务 2 就必定比事务 1 晚发生。因此，不存在某个事务读到了事务 2 对 `id = 1` 记录的修改，但却没有读到事务 1 对 `id = 2` 记录的修改的情况。

无因果关系的事务之间的逻辑顺序与物理提交顺序不保证一致

假设 `id = 1` 和 `id = 2` 的记录最初值都为 0，事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2	事务 3
START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY	START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY	
UPDATE t set v = 3 WHERE id = 2		
	UPDATE t SET v = 2 WHERE id = 1	
		BEGIN
COMMIT		
	COMMIT	
		SELECT v FROM t WHERE id IN (1, 2)

在本例中，事务 1 不读取 `id = 1` 的记录。此时事务 1 和事务 2 没有数据库可知的因果关系。如果使用因果一致性开启事务，即使物理时间上事务 2 在事务 1 提交完成后才开始提交，TiDB 也不保证逻辑上事务 2 比事务 1 晚发生。

此时如果有一个事务 3 在事务 1 提交前开启，并在事务 2 提交后读取 `id = 1` 和 `id = 2` 的记录，事务 3 可能读到 `id = 1` 的值为 2 但是 `id = 2` 的值为 0。

不加锁的读取不产生因果关系

假设事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2
START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY	START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY

事务 1	事务 2
	UPDATE t SET v = 2 WHERE id = 1
SELECT v FROM t WHERE id = 1	
UPDATE t set v = 3 WHERE id = 2	
	COMMIT
COMMIT	

如本例所示，不加锁的读取不产生因果关系。事务 1 和事务 2 产生了写偏斜的异常，如果他们有业务上的因果关系，则是不合理的。所以本例中，使用因果一致性的事务 1 和事务 2 没有确定的逻辑顺序。

4.8.10.2 TiDB 事务隔离级别

事务隔离级别是数据库事务处理的基础，ACID 中的“I”，即Isolation，指的就是事务的隔离性。

SQL-92 标准定义了 4 种隔离级别：读未提交 (READ UNCOMMITTED)、读已提交 (READ COMMITTED)、可重复读 (REPEATABLE READ)、串行化(SERIALIZABLE)。详见下表：

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not possible	Possible	Possible
REPEATABLE READ	Not Possible	Not possible	Not possible	Possible
SERIALIZABLE	Not Possible	Not possible	Not possible	Not possible

TiDB 实现了快照隔离(Snapshot Isolation, SI) 级别的一致性。为与 MySQL 保持一致，又称其为“可重复读”。该隔离级别不同于ANSI 可重复读隔离级别和MySQL 可重复读隔离级别。

> 注意：
>
> 在 TiDB v3.0 中，事务的自动重试功能默认为禁用状态。不建议开启自动重试功能，因为可能导致事务隔离级别遭到破坏。更多关于事务自动重试的文档说明，请参考事务重试。
>
> 从 TiDB v3.0.8 版本开始，新创建的TiDB 集群会默认使用悲观事务模式，悲观事务中的当前读（for update 读）为不可重复读，关于悲观事务使用注意事项，请参考悲观事务模式

4.8.10.2.1 可重复读隔离级别(Repeatable Read)

当事务隔离级别为可重复读时，只能读到该事务启动时已经提交的其他事务修改的数据，未提交的数据或在事务启动后其他事务提交的数据是不可见的。对于本事务而言，事务语句可以看到之前的语句做出的修改。

对于运行于不同节点的事务而言，不同事务启动和提交的顺序取决于从 PD 获取时间戳的顺序。

处于可重复读隔离级别的事务不能并发的更新同一行，当事务提交时发现该行在该事务启动后，已经被另一个已提交的事务更新过，那么该事务会回滚。示例如下：

```
create table t1(id int);
insert into t1 values(0);


start transaction;              |               start transaction;
select * from t1;               |               select * from t1;
update t1 set id=id+1;          |               update t1 set id=id+1; -- 如果使用悲观事务，
    <-> 则后一个执行的 update 语句会等锁，直到持有锁的事务提交或者回滚释放行锁。
commit;                         |
                                |               commit; -- 事务提交失败，回滚。如果使用悲观事务，
                                    <-> 可以提交成功。
```

与 ANSI 可重复读隔离级别的区别

尽管名称是可重复读隔离级别，但是 TiDB 中可重复读隔离级别和 ANSI 可重复隔离级别是不同的。按照 A Critique of ANSI SQL Isolation Levels 论文中的标准，TiDB 实现的是论文中的快照隔离级别。该隔离级别不会出现狭义上的幻读(A3)，但不会阻止广义上的幻读(P3)，同时，SI 还会出现写偏斜，而ANSI 可重复读隔离级别不会出现写偏斜，会出现幻读。

与 MySQL 可重复读隔离级别的区别

MySQL 可重复读隔离级别在更新时并不检验当前版本是否可见，也就是说，即使该行在事务启动后被更新过，同样可以继续更新。这种情况在TiDB 使用乐观事务时会导致事务回滚，导致事务最终失败，而TiDB 默认的悲观事务和 MySQL 是可以更新成功的。

4.8.10.2.2 读已提交隔离级别(Read Committed)

从 TiDB v4.0.0-beta 版本开始，TiDB 支持使用Read Committed 隔离级别。由于历史原因，当前主流数据库的Read Committed 隔离级别本质上都是 Oracle 定义的一致性读隔离级别。TiDB 为了适应这一历史原因，悲观事务中的 Read Committed 隔离级别的实质行为也是一致性读。

> 注意：
>
> Read Committed 隔离级别仅在悲观事务模式下生效。在乐观事务模式下设置事务隔离级别为 Read Committed 将不会生效，事务将仍旧使用可重复读隔离级别。

从 v6.0.0 版本开始，TiDB 支持使用系统变量`tidb_rc_read_check_ts` 对读写冲突较少情况下优化时间戳的获取。开启此变量后，`SELECT` 语句会尝试使用前一个有效的时间戳进行数据读取，初始值为事务的 `start_ts`。

- 如果整个读取过程没有遇到更新的数据版本，则返回结果给客户端且 SELECT 语句执行成功。
- 如果读取过程中遇到更新的数据版本：
  - 如果当前TiDB 尚未向客户端回复数据，则尝试重新获取一个新的时间戳重试此语句。
  - 如果 TiDB 已经向客户端返回部分数据，则TiDB 会向客户端报错。每次向客户端回复的数据量受 tidb_init_chunk_size 和 tidb_max_chunk_size 控制。

在使用 READ-COMMITTED 隔离级别且单个事务中 SELECT 语句较多、读写冲突较少的场景，可通过开启此变量来避免获取全局timestamp 带来的延迟和开销。

从 v6.3.0 版本开始，TiDB 支持通过开启系统变量tidb_rc_write_check_ts 对点写冲突较少情况下优化时间戳 的获取。开启此变量后，点写语句会尝试使用当前事务有效的时间戳进行数据读取和加锁操作，且在读取数据时按照开启tidb_rc_read_check_ts 的方式读取数据。目前该变量适用的点写语句包括 UPDATE、DELETE、SELECT ..... FOR UPDATE 三种类型。点写语句是指将主键或者唯一键作为过滤条件且最终执行算子包含 POINT-GET 的写语句。目前这三种点写语句的共同点是会先根据 key 值做点查，如果 key 存在再加锁，如果不存在则直接返回空集。

- 如果点写语句的整个读取过程中没有遇到更新的数据版本，则继续使用当前事务的时间戳进行加锁。
  - 如果加锁过程中遇到因时间戳旧而导致写冲突，则重新获取最新的全局时间戳进行加锁。
  - 如果加锁过程中没有遇到写冲突或其他错误，则加锁成功。
- 如果读取过程中遇到更新的数据版本，则尝试重新获取一个新的时间戳重试此语句。

在使用 READ-COMMITTED 隔离级别且单个事务中点写语句较多、点写冲突较少的场景，可通过开启此变量来避免获取全局时间戳带来的延迟和开销。

与 MySQL Read Committed 隔离级别的区别

MySQL 的 Read Committed 隔离级别大部分符合一致性读特性，但其中存在某些特例，如半一致性读 (semi-consistent read)，TiDB 没有兼容这个特殊行为。

4.8.10.2.3 更多阅读

- TiDB 的乐观事务模型
- TiDB 新特性漫谈-悲观事务
- TiDB 新特性-白话悲观锁
- TiKV 的 MVCC (Multi-Version Concurrency Control) 机制

4.8.10.3 TiDB 乐观事务模型

乐观事务模型下，将修改冲突视为事务提交的一部分。因此并发事务不常修改同一行时，可以跳过获取行锁的过程进而提升性能。但是并发事务频繁修改同一行（冲突）时，乐观事务的性能可能低于悲观事务。

启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

注意：

自 v3.0.8 开始，TiDB 集群默认使用悲观事务模式。但如果从 3.0.7 及之前版本创建的集群升级到 3.0.8 及之后的版本，不会改变默认事务模式，即只有新创建的集群才会默认使用悲观事务模式。

4.8.10.3.1 乐观事务原理

为支持分布式事务，TiDB 中乐观事务使用两阶段提交协议，流程如下：

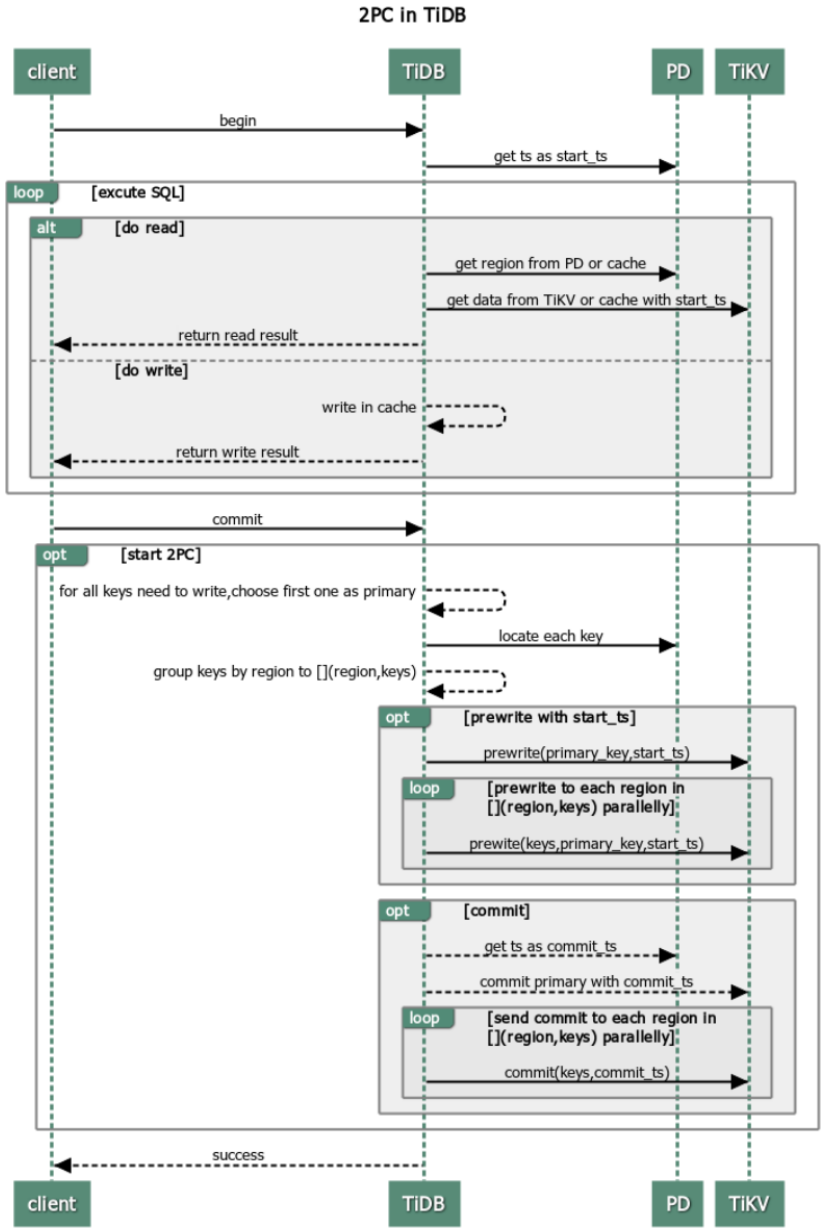


图 212: TiDB 中的两阶段提交

1. 客户端开始一个事务。

   TiDB 从 PD 获取一个全局唯一递增的时间戳作为当前事务的唯一事务ID，这里称为该事务的 `start_ts`。TiDB 实现了多版本并发控制(MVCC)，因此 `start_ts` 同时也作为该事务获取的数据库快照版本。该事务只能读到此 `start_ts` 版本可以读到的数据。

2. 客户端发起读请求。

   1. TiDB 从 PD 获取数据路由信息，即数据具体存在哪个TiKV 节点上。
   2. TiDB 从TiKV 获取 `start_ts` 版本下对应的数据。

3. 客户端发起写请求。

   TiDB 校验写入数据是否符合约束（如数据类型是否正确、是否符合非空约束等）。校验通过的数据将存放在 TiDB 中该事务的私有内存里。

4. 客户端发起commit。

5. TiDB 开始两阶段提交，在保证事务原子性的前提下，进行数据持久化。

   1. TiDB 从当前要写入的数据中选择一个 Key 作为当前事务的 Primary Key。
   2. TiDB 从 PD 获取所有数据的写入路由信息，并将所有的 Key 按照所有的路由进行分类。
   3. TiDB 并发地向所有涉及的TiKV 发起prewrite 请求。 TiKV 收到prewrite 数据后，检查数据版本信息是否存在冲突或已过期。符合条件的数据会被加锁。
   4. TiDB 收到所有prewrite 响应且所有prewrite 都成功。
   5. TiDB 向 PD 获取第二个全局唯一递增版本号，定义为本次事务的 `commit_ts`。
   6. TiDB 向 Primary Key 所在TiKV 发起第二阶段提交。 TiKV 收到commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁。
   7. TiDB 收到两阶段提交成功的信息。

6. TiDB 向客户端返回事务提交成功的信息。

7. TiDB 异步清理本次事务遗留的锁信息。

4.8.10.3.2 优缺点分析

通过分析TiDB 中事务的处理流程，可以发现TiDB 事务有如下优点：

- 实现原理简单，易于理解。
- 基于单实例事务实现了跨节点事务。
- 锁管理实现了去中心化。

但TiDB 事务也存在以下缺点：

- 两阶段提交使网络交互增多。
- 需要一个中心化的分配时间戳服务。
- 事务数据量过大时易导致内存暴涨。

4.8.10.3.3 事务的重试

使用乐观事务模型时，在高冲突率的场景中，事务容易发生写写冲突而导致提交失败。MySQL 使用悲观事务模型，在执行写入类型的 SQL 语句的过程中进行加锁并且在Repeatable Read 隔离级别下使用了当前读的机制，能够读取到最新的数据，所以提交时一般不会出现异常。为了降低应用改造难度，TiDB 提供了数据库内部自动重试机制。

重试机制

当事务提交时， 如果发现写写冲突，TiDB 内部重新执行包含写操作的 SQL 语句。你可以通过设置 `tidb_disable_txn_auto_retry = OFF` 开启自动重试，并通过 `tidb_retry_limit` 设置重试次数：

```
#### 设置是否禁用自动重试，默认为 “on”，即不重试。
tidb_disable_txn_auto_retry = OFF
#### 控制重试次数，默认为 “10”。只有自动重试启用时该参数才会生效。
#### 当 “tidb_retry_limit= 0” 时，也会禁用自动重试。
tidb_retry_limit = 10
```

你也可以修改当前 Session 或 Global 的值：

- Session 级别设置：

```
SET tidb_disable_txn_auto_retry = OFF;
```

```
SET tidb_retry_limit = 10;
```

- Global 级别设置：

```
SET GLOBAL tidb_disable_txn_auto_retry = OFF;
```

```
SET GLOBAL tidb_retry_limit = 10;
```

> 注意：
>
> `tidb_retry_limit` 变量决定了事务重试的最大次数。当它被设置为 0 时，所有事务都不会自动重试，包括自动提交的单语句隐式事务。这是彻底禁用TiDB 中自动重试机制的方法。禁用自动重试后，所有冲突的事务都会以最快的方式上报失败信息（包含 `try again later`）给应用层。

重试的局限性

TiDB 默认不进行事务重试，因为重试事务可能会导致更新丢失，从而破坏可重复读的隔离级别。

事务重试的局限性与其原理有关。事务重试可概括为以下三个步骤：

1. 重新获取 `start_ts`。
2. 重新执行包含写操作的 SQL 语句。
3. 再次进行两阶段提交。

第二步中，重试时仅重新执行包含写操作的 SQL 语句，并不涉及读操作的 SQL 语句。但是当前事务中读到数据的时间与事务真正开始的时间发生了变化，写入的版本变成了重试时获取的 `start_ts` 而非事务一开始时获取的 `start_ts`。因此，当事务中存在依赖查询结果来更新的语句时，重试将无法保证事务原本可重复读的隔离级别，最终可能导致结果与预期出现不一致。

如果业务可以容忍事务重试导致的异常，或并不关注事务是否以可重复读的隔离级别来执行，则可以开启自动重试。

4.8.10.3.4 冲突检测

作为一个分布式系统，TiDB 在内存中的冲突检测是在TiKV 中进行，主要发生在prewrite 阶段。因为TiDB 集群是一个分布式系统，TiDB 实例本身无状态，实例之间无法感知到彼此的存在，也就无法确认自己的写入与别的TiDB 实例是否存在冲突，所以会在TiKV 这一层检测具体的数据是否有冲突。

具体配置如下：

```
#### scheduler 内置一个内存锁机制，防止同时对一个 Key 进行操作。
#### 每个 Key hash 到不同的 slot。（默认为 2048000）
scheduler-concurrency = 2048000
```

此外，TiKV 支持监控等待latch 的时间：

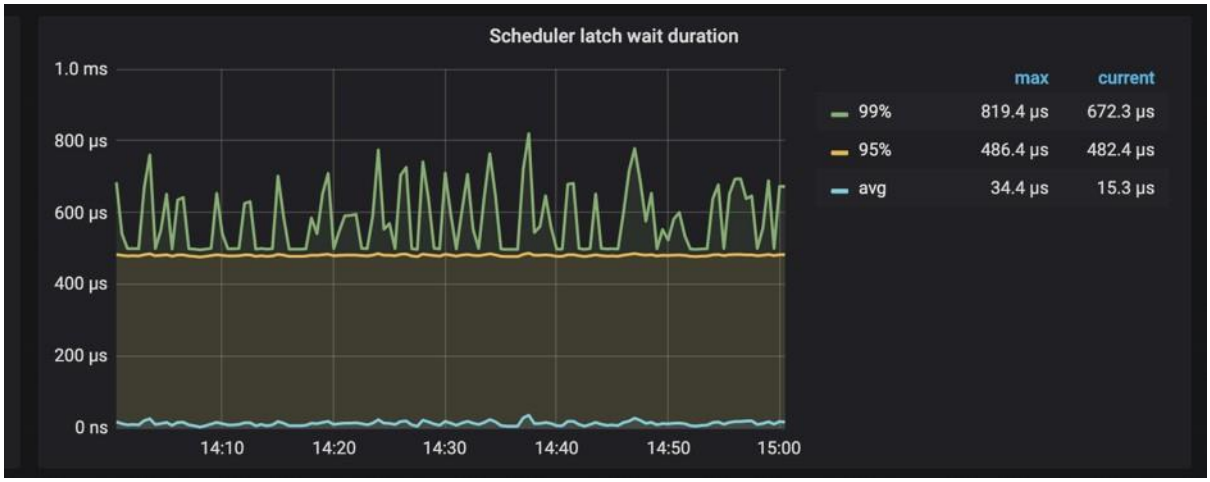


图 213: Scheduler latch wait duration

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。

4.8.10.3.5 更多阅读

- Percolator 和 TiDB 事务算法

4.8.10.4 TiDB 悲观事务模式

为了使TiDB 的使用方式更加贴近传统数据库，降低用户迁移的成本，TiDB 自 v3.0 版本开始在乐观事务模型的基础上支持了悲观事务模式。本文将介绍TiDB 悲观事务的相关特性。

> 注意：
>
> 自 v3.0.8 开始，新创建的TiDB 集群默认使用悲观事务模式。但如果从 v3.0.7 版本及之前创建的集群升级到>= v3.0.8 的版本，则不会改变默认的事务模式，即只有新创建的集群才会默认使用悲观事务模式。

4.8.10.4.1 事务模式的修改方法

你可以使用tidb_txn_mode 系统变量设置事务模式。执行以下命令，即可使整个集群中所有新创建 session 执行的所有显示事务（即非autocommit 的事务）进入悲观事务模式：

```
SET GLOBAL tidb_txn_mode = 'pessimistic';
```

除此之外，还可以执行以下 SQL 语句显式地开启悲观事务：

```
BEGIN PESSIMISTIC;
```

```
BEGIN /*T! PESSIMISTIC */;
```

BEGIN PESSIMISTIC; 和 BEGIN OPTIMISTIC; 等语句的优先级高于 tidb_txn_mode 系统变量。使用这两个语句开启的事务，会忽略系统变量，从而支持悲观、乐观事务混合使用。

4.8.10.4.2 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）：

- 悲观事务中引入快照读和当前读的概念：
  - 快照读是一种不加锁读，读的是该事务开始时刻前已提交的版本。SELECT 语句中的读是快照读。
  - 当前读是一种加锁读，读取的是最新已提交的版本，UPDATE、DELETE、INSERT、SELECT FOR UPDATE 语句中的读是当前读。

  以下示例是对快照读和当前读的详细说明：

session 1	session 2
CREATE TABLE t (a INT); INSERT INTO T VALUES(1); BEGIN PESSIMISTIC; UPDATE t SET a = a + 1;	
	BEGIN PESSIMISTIC; SELECT * FROM t; –
COMMIT; – 释放锁，session 3 的 SELECT FOR UPDATE 操作获得锁，使用当前读，读到最新已提交的版本(a=2)	
	SELECT * FROM t; –

- 悲观锁会在事务提交或回滚时释放。其他尝试修改这一行的写事务会被阻塞，等待悲观锁的释放。其他尝试读取这一行的事务不会被阻塞，因为TiDB 采用多版本并发控制机制(MVCC)。
- 需要检查唯一性约束的悲观锁可以通过设置系统变量tidb_constraint_check_in_place_pessimistic 控制是否跳过，详见约束。
- 如果多个事务尝试获取各自的锁，会出现死锁，并被检测器自动检测到。其中一个事务会被随机终止掉并返回兼容 MySQL 的错误码 1213。

- 通过 innodb_lock_wait_timeout 变量，设置事务等锁的超时时间（默认值为 50，单位为秒）。等锁超时后返回兼容 MySQL 的错误码 1205。如果多个事务同时等待同一个锁释放，会大致按照事务 start ts 顺序获取锁。
- 乐观事务和悲观事务可以共存，事务可以任意指定使用乐观模式或悲观模式来执行。
- 支持 FOR UPDATE NOWAIT 语法，遇到锁时不会阻塞等锁，而是返回兼容 MySQL 的错误码 3572。
- 如果 Point Get 和 Batch Point Get 算子没有读到数据，依然会对给定的主键或者唯一键加锁，阻塞其他事务对相同主键唯一键加锁或者进行写入操作。
- 支持 FOR UPDATE OF TABLES 语法，对于存在多表join 的语句，只对 OF TABLES 中包含的表关联的行进行悲观锁加锁操作。

4.8.10.4.3 和 MySQL InnoDB 的差异

1. 有些 WHERE 子句中使用了range，TiDB 在执行这类 DML 语句和 SELECT FOR UPDATE 语句时，不会阻塞 range 内并发的 DML 语句的执行。

   举例：

   ```
   CREATE TABLE t1 (
    id INT NOT NULL PRIMARY KEY,
    pad1 VARCHAR(100)
   );
   INSERT INTO t1 (id) VALUES (1),(5),(10);
   ```

   ```
   BEGIN /*T! PESSIMISTIC */;
   SELECT * FROM t1 WHERE id BETWEEN 1 AND 10 FOR UPDATE;
   ```

   ```
   BEGIN /*T! PESSIMISTIC */;
   INSERT INTO t1 (id) VALUES (6); -- 仅 MySQL 中出现阻塞。
   UPDATE t1 SET pad1='new value' WHERE id = 5; -- MySQL 和 TiDB 处于等待阻塞状态。
   ```

   产生这一行为是因为TiDB 当前不支持 gap locking（间隙锁）。

2. TiDB 不支持 SELECT LOCK IN SHARE MODE。

   使用这个语句执行的时候，效果和没有加锁是一样的，不会阻塞其他事务的读写。

3. DDL 可能会导致悲观事务提交失败。

   MySQL 在执行 DDL 语句时，会被正在执行的事务阻塞住，而在TiDB 中 DDL 操作会成功，造成悲观事务提交失败：ERROR 1105 (HY000): Information schema is changed. [try again later]。TiDB 事务执行过程中并发执行 TRUNCATE TABLE 语句，可能会导致事务报错 table doesn't exist。

4. START TRANSACTION WITH CONSISTENT SNAPSHOT 之后，MySQL 仍然可以读取到之后在其他事务创建的表，而 TiDB 不能。

5. autocommit 事务优先采用乐观事务提交。

   使用悲观事务模式时，autocommit 事务首先尝试使用开销更小的乐观事务模式提交。如果发生了写冲突，重试时才会使用悲观事务提交。所以 `tidb_retry_limit = 0` 时，autocommit 事务遇到写冲突仍会报 `Write Conflict` 错误。

   自动提交的 `SELECT FOR UPDATE` 语句不会等锁。

6. 对语句中 `EMBEDDED SELECT` 读到的相关数据不会加锁。

7. 垃圾回收(GC) 不会影响到正在执行的事务，但悲观事务的执行时间仍有上限，默认为 1 小时，可通过 TiDB 配置文件 `[performance]` 类别下的 `max-txn-ttl` 修改。

4.8.10.4.4 隔离级别

TiDB 在悲观事务模式下支持了 2 种隔离级别：

1. 默认使用与 MySQL 行为相同的可重复读隔离级别(Repeatable Read)。

> 注意：
> 在这种隔离级别下，DML 操作会基于已提交的最新数据来执行，行为与 MySQL 相同，但与 TiDB 乐观事务不同，请参考与 MySQL 可重复读隔离级别的区别。

2. 使用`SET TRANSACTION` 语句可将隔离级别设置为读已提交隔离级别(Read Committed)。

4.8.10.4.5 悲观事务提交流程

TiDB 悲观锁复用了乐观锁的两阶段提交逻辑，重点在 DML 执行时做了改造。

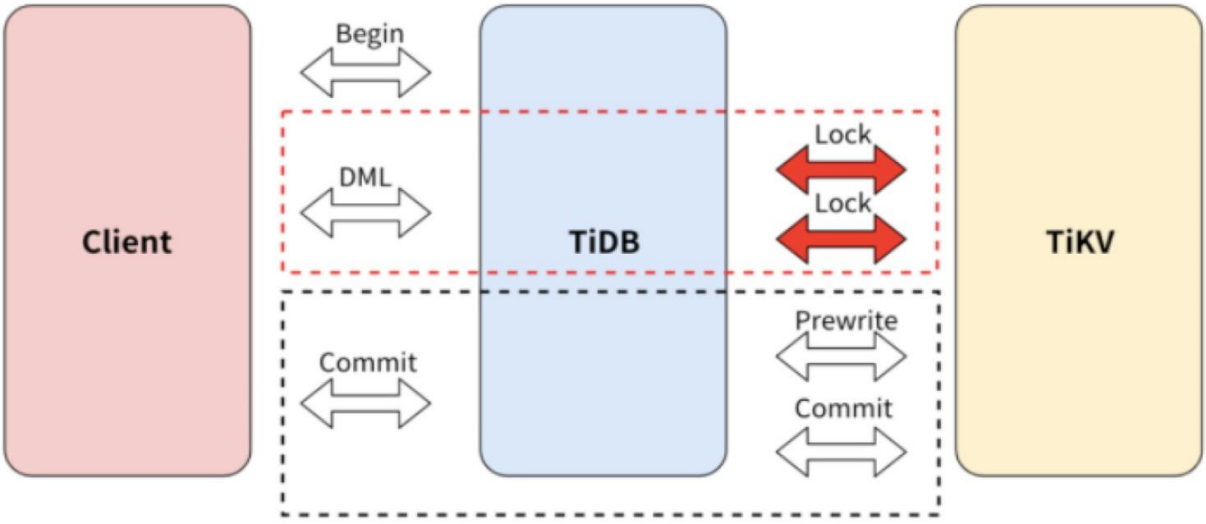


图 214: TiDB 悲观事务的提交流程

在两阶段提交之前增加了Acquire Pessimistic Lock 阶段，简要步骤如下。

1. （同乐观锁）TiDB 收到来自客户端的begin 请求，获取当前时间戳作为本事务的 StartTS。
2. TiDB 收到来自客户端的更新数据的请求：TiDB 向TiKV 发起加悲观锁请求，该锁持久化到TiKV。
3. （同乐观锁）客户端发起commit，TiDB 开始执行与乐观锁一样的两阶段提交。

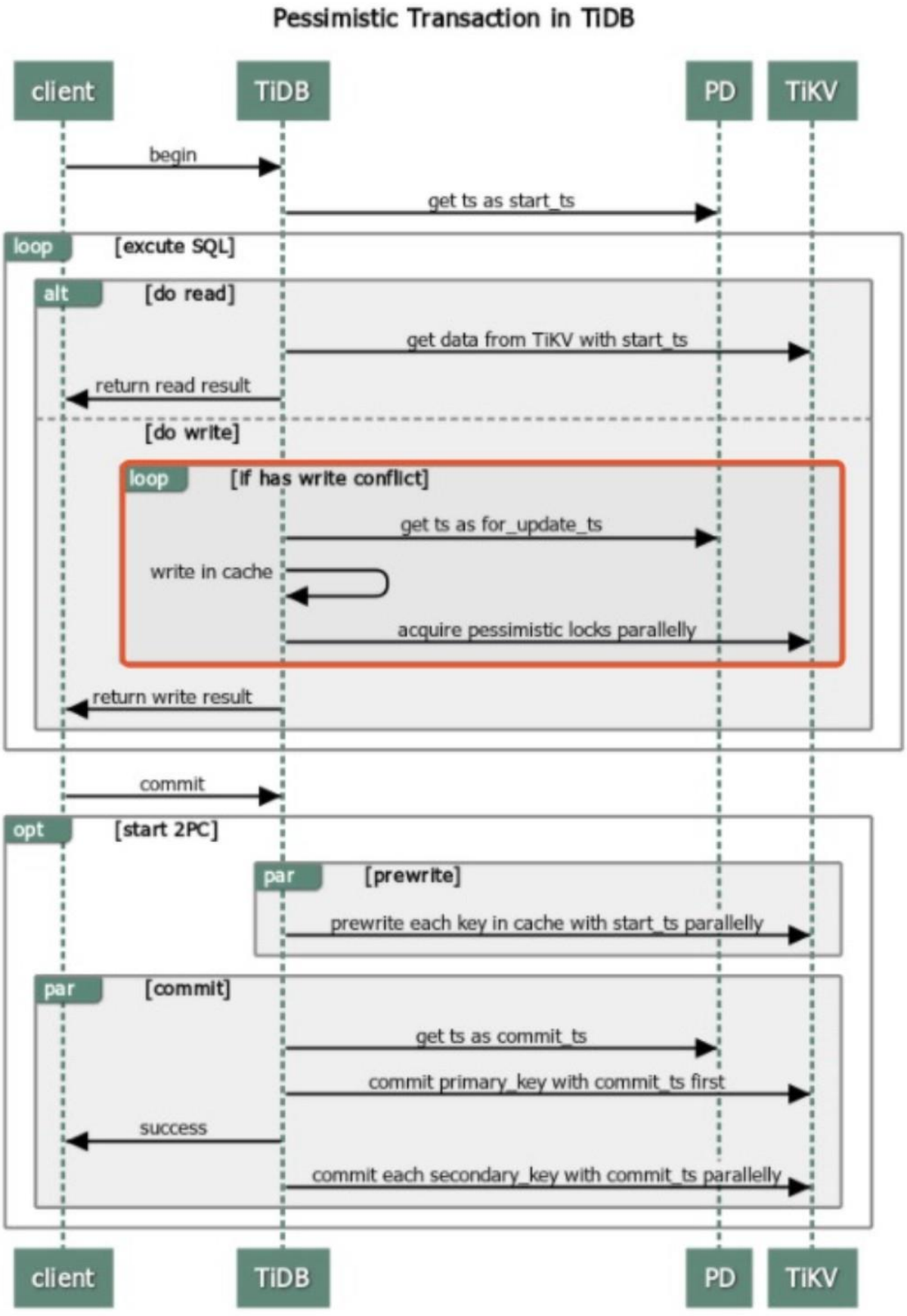


图 215: TiDB 中的悲观事务

相关细节本节不再赘述，详情可阅读TiDB 悲观锁实现原理。

4.8.10.4.6 Pipelined 加锁流程

加悲观锁需要向TiKV 写入数据，要经过Raft 提交并apply 后才能返回，相比于乐观事务，不可避免的会增加部分延迟。为了降低加锁的开销，TiKV 实现了pipelined 加锁流程：当数据满足加锁要求时，TiKV 立刻通知TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但当TiKV 出现网络隔离或者节点宕机时，悲观锁异步写入有可能失败，从而产生以下影响：

- 无法阻塞修改相同数据的其他事务。如果业务逻辑依赖加锁或等锁机制，业务逻辑的正确性将受到影响。
- 有较低概率导致事务提交失败，但不会影响事务正确性。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭pipelined 加锁功能。

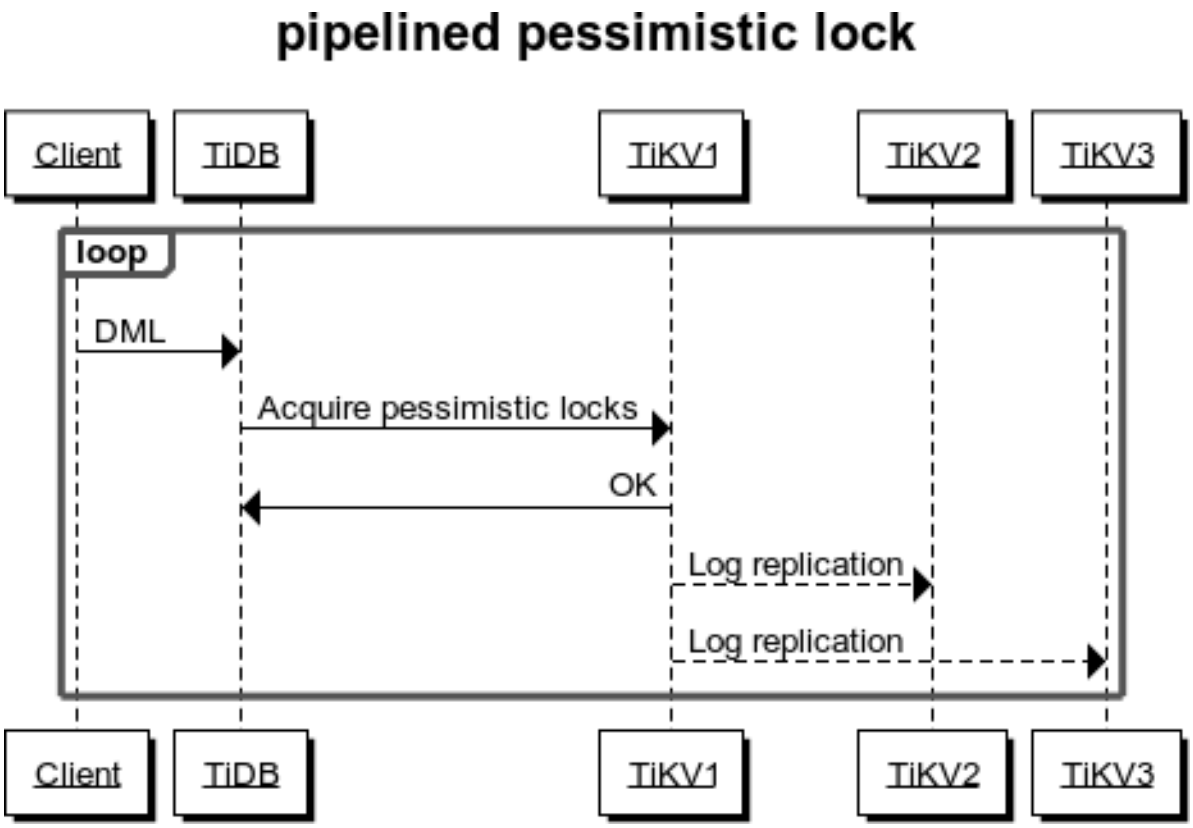


图 216: Pipelined pessimistic lock

该功能默认开启，可修改TiKV 配置关闭：

```
[pessimistic-txn]
pipelined = false
```

若集群是 v4.0.9 及以上版本，也可通过在线修改TiKV 配置功能动态关闭该功能：

```
set config tikv pessimistic-txn.pipelined='false';
```

4.8.10.4.7 内存悲观锁

TiKV 在 v6.0.0 中引入了内存悲观锁功能。开启内存悲观锁功能后，悲观锁通常只会被存储在Region leader 的内存中，而不会将锁持久化到磁盘，也不会通过Raft 协议将锁同步到其他副本，因此可以大大降低悲观事务加锁的开销，提升悲观事务的吞吐并降低延迟。

当内存悲观锁占用的内存达到Region 或节点的阈值时，加悲观锁会回退为使用pipelined 加锁流程。 当Region 发生合并或leader 迁移时，为避免悲观锁丢失，TiKV 会将内存悲观锁写入磁盘并同步到其他副本。

内存悲观锁实现了和pipelined 加锁类似的表现，即集群无异常时不影响加锁表现，但当TiKV 出现网络隔离或者节点宕机时，事务加的悲观锁可能丢失。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭内存悲观锁功能。

该功能默认开启。如要关闭，可修改TiKV 配置：

```
[pessimistic-txn]
in-memory = false
```

也可通过在线修改TiKV 配置功能动态关闭该功能：

```
set config tikv pessimistic-txn.in-memory='false';
```

4.8.10.5 非事务 DML 语句

本文档介绍非事务 DML 语句的使用场景、使用方法、使用限制和使用该功能的常见问题。

非事务 DML 语句是将一个普通 DML 语句拆成多个 SQL 语句（即多个batch）执行，以牺牲事务的原子性和隔离性为代价，增强批量数据处理场景下的性能和易用性。

通常，对于消耗内存过多的大事务，你需要在应用中拆分 SQL 语句以绕过事务大小限制。非事务 DML 语句将这一过程集成到TiDB 内核中，实现等价的效果。非事务 DML 语句的执行效果可以通过拆分 SQL 语句的结果来理解，DRY RUN 语法提供了预览拆分后语句的功能。

非事务 DML 语句包括：

- INSERT INTO ... SELECT
- REPLACE INTO .. SELECT
- UPDATE
- DELETE

详细的语法介绍见BATCH。

> 注意：
>
> - 非事务 DML 语句不保证该语句的原子性和隔离性，不能认为它和原始 DML 语句等价。
> - 在任意 DML 语句改写为非事务 DML 语句后，不应假设其行为与原来一致。
> - 使用非事务 DML 前需要分析其拆分后的语句是否会互相影响。

4.8.10.5.1 使用场景

在大批量的数据处理场景，用户经常需要对一大批数据执行相同的操作。如果直接使用一个 SQL 语句执行操作，很可能导致事务大小超过限制，而大事务会明显影响执行性能。

批量数据处理操作往往和在线业务操作不具有时间或数据上的交集。没有并发操作时，隔离性是不必要的。如果将批量数据操作设计成幂等的，或者易于重试的，那么原子性也是不必要的。如果你的业务满足这两个条件，那么可以考虑使用非事务 DML 语句。

非事务 DML 语句用于在特定场景下绕过大事务的事务大小限制，用一条语句完成原本需要拆分成多个事务完成的任务，且执行效率更高，占用资源更少。

例如，对于一个删除过期数据的需求，在确保没有任何业务会访问过期数据时，适合使用非事务 DML 语句来提升删除性能。

4.8.10.5.2 前提条件

要使用非事务 DML 语句，必须满足以下条件：

- 确保该语句不需要原子性，即允许执行结果中，一部分行被修改，而一部分行没有被修改。
- 确保该语句具有幂等性，或是做好准备根据错误信息对部分数据重试。如果系统变量 `tidb_redact_log` <⟷> `= 1` 且 `tidb_nontransactional_ignore_error = 1`，则该语句必须是幂等的。否则语句部分失败时，无法准确定位失败的部分。
- 确保该语句将要操作的数据没有其它并发的写入，即不被其它语句同时更新。否则可能出现漏写、多写、重复修改同一行等非预期的现象。
- 确保该语句不会修改语句自身会读取的内容，否则后续的batch 读到之前batch 写入的内容，容易引起非预期的情况。
  - 在使用非事务 `INSERT INTO ... SELECT` 处理同一张表时，尽量不要在插入时修改拆分列，否则可能因为多个batch 读取到同一行，导致重复插入：
    * 不推荐使用 `BATCH ON test.t.id LIMIT 10000 INSERT INTO t SELECT id+1, value FROM t;`
    * 推荐使用 `BATCH ON test.t.id LIMIT 10000 INSERT INTO t SELECT id, value FROM t;`
    * 当 `id` 列具有AUTO_INCREMENT 属性时，推荐使用`BATCH ON test.t.id LIMIT 10000 INSERT INTO` <⟷> `t(value)SELECT value FROM t;`
  - 在使用非事务 `UPDATE`、`INSERT ... ON DUPLICATE KEY UPDATE`、`REPLACE INTO` 时，拆分列不应该在语句中更新：

    * 例如，对于一条非事务 UPDATE 语句，拆分后的 SQL 依次执行，前一batch 的修改提交后被后一 batch 读到，导致同一行数据被多次修改。
    * 这类语句不支持 `BATCH ON test.t.id LIMIT 10000 UPDATE t SET test.t.id = test.t.id-1;`
    * 不推荐使用 `BATCH ON test.t.id LIMIT 1 INSERT INTO t SELECT id+1, value FROM t ON <⟶ DUPLICATE KEY UPDATE id = id + 1;`
  - 拆分列也不应该用于Join key。例如，下面示例将拆分列 `test.t.id` 作为Join key，导致一个非事务 UPDATE 语句多次更新同一行：

```
CREATE TABLE t(id int, v int, key(id));
CREATE TABLE t2(id int, v int, key(id));
INSERT INTO t VALUES (1, 1), (2, 2), (3, 3);
INSERT INTO t2 VALUES (1, 1), (2, 2), (4, 4);
BATCH ON test.t.id LIMIT 1 UPDATE t JOIN t2 ON t.id = t2.id SET t2.id = t2.id+1;
SELECT * FROM t2; -- (4, 1) (4, 2) (4, 4)
```

- 确认该语句满足使用限制。
- 不建议在该 DML 语句将要读写的表上同时进行并发的 DDL 操作。

> **警告：**
>
> 如果同时开启了 `tidb_redact_log` 和 `tidb_nontransactional_ignore_error`，你可能无法完整得知每个batch 的错误信息，无法只对失败的batch 进行重试。因此，如果同时开启了这两个系统变量，该非事务 DML 语句必须是幂等的。

4.8.10.5.3 使用示例

使用非事务 DML 语句

以下部分通过示例说明非事务 DML 语句的使用方法：

创建一张表`t`，表结构如下：

```
CREATE TABLE t(id int, v int, key(id));
```

```
Query OK, 0 rows affected
```

向表`t` 中插入一些数据。

```
INSERT INTO t VALUES (1,2),(2,3),(3,4),(4,5),(5,6);
```

```
Query OK, 5 rows affected
```

以下操作使用非事务 DML 语句，删除表`t` 的 `v` 列上小于整数 6 的行。该语句将以 2 为 batch size，以 `id` 列为划分列，拆分为两个 SQL 语句执行。

```
BATCH ON id LIMIT 2 DELETE FROM t where v < 6;
```

```
+-----------------+----------------+
| number of jobs | job status     |
+-----------------+----------------+
| 2               | all succeeded |
+-----------------+----------------+
1 row in set
```

查看非事务 DML 语句的删除结果。

```
SELECT * FROM t;
```

```
+----+---+
| id | v |
+----+---+
| 5  | 6 |
+----+---+
1 row in set
```

以下示例说明多表join 的使用方法。首先创建表 t2 并插入数据。

```
CREATE TABLE t2(id int, v int, key(id));
INSERT INTO t2 VALUES (1,1), (3,3), (5,5);
```

然后进行涉及多表join 的更新（表 t 和 t2）。需要注意的是，指定拆分列时需要完整的数据库名、表名和列名（test.t._tidb_rowid)。

```
BATCH ON test.t._tidb_rowid LIMIT 1 UPDATE t JOIN t2 ON t.id = t2.id SET t2.id = t2.id+1;
```

查看更新后表的数据：

```
SELECT * FROM t2;
```

```
+----+---+
| id | v |
+----+---+
| 1  | 1 |
| 3  | 3 |
| 6  | 5 |
+----+---+
```

查看非事务 DML 语句的执行进度

非事务 DML 语句执行过程中，可以通过 SHOW PROCESSLIST 查看执行进度，返回结果中的 Time 表示当前batch 执行的耗时。日志、慢日志等也会记录每个拆分后的语句在整个非事务 DML 语句中的进度。例如：

```
show processlist;
```

```
+--
    <-> ----+------+--------------------+--------+---------+------+------------+-----------------------------
    <->
| Id    | User | Host                | db     | Command | Time | State      | Info
    <->
    <-> |
+--
    <-> ----+------+--------------------+--------+---------+------+------------+-----------------------------
    <->
| 1203 | root | 100.64.10.62:52711 | test    | Query    | 0    | autocommit | /* job 506/500000 */
    <-> DELETE FROM `test`.`t1` WHERE `test`.`t1`.`_tidb_rowid` BETWEEN 2271 AND 2273 |
| 1209 | root | 100.64.10.62:52735 | <null> | Query    | 0    | autocommit | show full processlist
    <->                                                                                |
+--
    <-> ----+------+--------------------+--------+---------+------+------------+-----------------------------
    <->
```

终止一个非事务 DML 语句

通过 KILL TIDB <processlist_id> 终止一个非事务语句时，TiDB 会取消当前正在执行的 batch 之后的所有 batch。执行结果信息需要从日志里获得。

关于 KILL TiDB 的更多信息，参见KILL。

查询非事务 DML 语句中划分batch 的语句

要查询非事务 DML 语句中用于划分batch 的语句，你可在非事务 DML 语句中添加 DRY RUN QUERY。添加后，TiDB 不实际执行这个查询和后续的 DML 操作。

下面这条语句查询 BATCH ON id LIMIT 2 DELETE FROM t WHERE v < 6 这条非事务 DML 语句内用于划分batch 的查询语句。

```
BATCH ON id LIMIT 2 DRY RUN QUERY DELETE FROM t WHERE v < 6;
```

```
+-----------------------------------------------------------------------------------+
| query statement                                                                   |
+-----------------------------------------------------------------------------------+
| SELECT `id` FROM `test`.`t` WHERE (`v` < 6) ORDER BY IF(ISNULL(`id`),0,1),`id` |
+-----------------------------------------------------------------------------------+
1 row in set
```

查询非事务 DML 语句中首末batch 对应的语句

要查询非事务 DML 语句中第一个和最后一个batch 对应的实际 DML 语句，你可在语句中添加 DRY RUN。添加后，TiDB 只划分batch，不执行这些 SQL 语句。因为batch 数量可能很多，不显示全部batch，只显示第一个和最后一个batch。

```
BATCH ON id LIMIT 2 DRY RUN DELETE FROM t where v < 6;
```

```
+-----------------------------------------------------------------------+
| split statement examples                                              |
+-----------------------------------------------------------------------+
| DELETE FROM `test`.`t` WHERE (`id` BETWEEN 1 AND 2 AND (`v` < 6)) |
| DELETE FROM `test`.`t` WHERE (`id` BETWEEN 3 AND 4 AND (`v` < 6)) |
+-----------------------------------------------------------------------+
2 rows in set
```

在非事务 DML 语句中使用 Optimizer Hint

对于 DELETE 语句原本支持的 Optimizer Hint，非事务 DELETE 语句也同样支持，hint 位置与普通 DELETE 语句中的位置相同：

```
BATCH ON id LIMIT 2 DELETE /*+ USE_INDEX(t)*/ FROM t where v < 6;
```

4.8.10.5.4 最佳实践

建议按照以下步骤执行非事务 DML 语句：

1. 选择合适的划分列。建议使用整数或字符串类型。
2. 在非事务 DML 语句中添加 DRY RUN QUERY，手动执行查询，确认 DML 语句影响的数据范围是否大体正确。
3. 在非事务 DML 语句中添加 DRY RUN，手动执行查询，检查拆分后的语句和执行计划。需要关注：
    - 一条拆分后的语句是否有可能读到之前的语句执行写入的结果，否则容易造成异常现象。
    - 索引选择效率。
    - 由TiDB 自动选择的拆分列是否可能会被修改。
4. 执行非事务 DML 语句。
5. 如果报错，从报错信息或日志中获取具体失败的数据范围，进行重试或手动处理。

4.8.10.5.5 参数说明

参数	说明	默认值	是否必填	建议值

参数	说明	默认值	是否必填	建议值
划分列	用于划分 batch 的列，例如以上非事务 DML 语句 `BATCH ON id LIMIT 2 DELETE FROM t WHERE v < 1077 LIMIT 6`	TiDB 尝试自动选择（不建议）	否	选择可以最高效地满足 WHERE 条件的列

参数	说明	默认值	是否必填	建议值
Batch size	用于控制每个 batch 的大小，batch 即 DML 操作拆分成的 SQL 语句个数，例如以上非事务 DML 语句 BATCH <⟷ <⟷ ON <⟷ <⟷ id <⟷ <⟷ LIMIT <⟷ <⟷ 2 1078 <⟷ <⟷ DELETE <⟷	N/A	是	1000 ~ 1000000，过小和过大都会导致性能下降

参数	说明	默认值	是否必填	建议值

划分列的选择

非事务 DML 语句需要用一个列作为数据分批的标准，该列即为划分列。为获得更高的执行效率，划分列必须能够利用索引。不同的索引和划分列所导致的执行效率可能有数十倍的差别。选择划分列时，可以考虑以下建议：

- 当你对业务数据分布有一定了解时，根据 WHERE 条件，选择划分batch 后，划分范围较小的列。
  - 在理想情况下，WHERE 条件可以利用划分列的索引，降低每个batch 需要扫描的数据量。例如有一个交易表，记录了每一笔交易的开始和结束时间，你希望删除结束时间在一个月之前的所有交易记录。如果在交易的开始时间有索引，且已知交易的开始和结束时间通常相差不大，那么可以选择开始时间作为划分列。
  - 在不太理想的情况下：划分列的数据分布与 WHERE 条件完全无关，无法利用划分列的索引来减少数据扫描的范围。
- 有聚簇索引时，建议用主键作为划分列，这样语句执行效率更高（包括 INT 主键和 _tidb_rowid）。
- 选择重复值较少的列。

你可以不指定划分列，TiDB 默认会使用handle 的第一列作为划分列。但如果聚簇索引主键的第一列是非事务 DML 语句不支持的数据类型（即 ENUM，BIT，SET，JSON），TiDB 会报错。你可根据业务需要选择合适的划分列。

Batch size 的选择

非事务 DML 语句中，batch size 越大，拆分出来的 SQL 语句越少，每个 SQL 语句执行起来越慢。最优的batch size 依据 workload 而定。根据经验值，推荐从 50000 开始尝试。过小和过大的batch size 都会导致执行效率下降。

每个batch 的信息存储在内存里，因此batch 数量过多会显著增加内存消耗。这也是batch size 不能过小的原因之一。非事务语句用于存储 batch 信息消耗的内存上限与tidb_mem_quota_query 相同，超出这个限制时触发的操作由系统变量tidb_mem_oom_action 控制。

4.8.10.5.6 使用限制

非事务 DML 语句的硬性限制，不满足这些条件时TiDB 会报错。

- DML 语句不能包含 ORDER BY 或 LIMIT 字句。
- 不支持子查询或集合操作。
- 用于拆分的列必须被索引。该索引可以是单列的索引，或是一个联合索引的第一列。
- 必须在autocommit 模式中使用。
- 不能在开启了batch-dml 时使用。
- 不能在设置了tidb_snapshot 时使用。
- 不能与 prepare 语句一起使用。
- 划分列不支持 ENUM，BIT，SET，JSON 类型。
- 不支持用在临时表上。
- 不支持公共表表达式。

4.8.10.5.7 控制batch 执行失败

非事务DML 语句不满足原子性，可能存在一些batch 成功，一些batch 失败的情况。系统变量tidb_nontransactional_ignore_error <—> 控制非事务 DML 语句处理错误的行为。

一个例外是，如果第一个batch 就执行失败，有很大概率是语句本身有错，此时整个非事务语句会直接返回一个错误。

4.8.10.5.8 实现原理

非事务 DML 语句的实现原理，是将原本需要在用户侧手动执行的 SQL 语句拆分工作内置为TiDB 的一个功能，简化用户操作。要理解非事务 DML 语句的行为，可以将其想象成一个用户脚本进行了如下操作：

对于非事务 DML `BATCH ON $C$ LIMIT $N$ DELETE FROM ... WHERE $P$`，其中 C 为用于拆分的列，N 为 batch size，P 为筛选条件。

1. TiDB 根据原始语句的筛选条件 P，和指定的用于拆分的列 C，查询出所有满足 P 的 C。对这些 C 排序后按 N 分成多个分组 $B_1 \ldots B_k$。对所有 B_i，保留它的第一个和最后一个 C，记为 S_i 和 E_i。这一步所执行的查询语句，可以通过DRY RUN QUERY 查看。
2. B_i 所涉及的数据就是满足 P_i: C BETWEEN S_i AND E_i 的一个子集。可以通过 P_i 来缩小每个batch 需要处理的数据范围。
3. 对 B_i，将上面这个条件嵌入原始语句的WHERE 条件，使其变为WHERE (P_i) AND (P)。这一步的执行结果，可以通过DRY RUN 查看。
4. 对所有batch，依次执行新的语句。收集每个分组的错误并合并，在所有分组结束后作为整个非事务 DML 语句的结果返回。

4.8.10.5.9 与 batch-dml 的异同

batch-dml 是一种在 DML 语句执行期间将一个事务拆成多个事务提交的机制。

> 注意：
>
> batch-dml 功能使用不当时，存在数据索引不一致风险。该功能已被废弃，因此不建议使用。

非事务 DML 语句尚不能替代所有的batch-dml 使用场景。它们的主要区别有：

- 性能：在划分效率较高的情况下，非事务 DML 语句和batch-dml 性能接近。在划分效率较低的情况下，非事务 DML 语句可能会明显慢于batch-dml。
- 稳定性：batch-dml 极易因为使用不当导致数据索引不一致问题。而非事务 DML 语句不会导致数据索引不一致问题。但使用不当时，非事务 DML 语句与原始语句不等价，应用可能观察到和预期不符的现象。详见常见问题。

4.8.10.5.10 常见问题

实际的batch 大小和指定的batch size 不一样

在非事务 DML 语句的执行过程中，最后一个batch 处理的数据量可能会小于batch size。

在划分列有重复值时，每个batch 会将当前batch 中划分列的最后一个元素的所有重复值全部加入当前batch 中，因此这个batch 的行数可能会多于batch size。

另外，在有其它并发的数据写入时，也可能导致每个batch 实际处理的行数和batch size 不一致。

执行时出现报错 `Failed to restore the delete statement, probably because of unsupported type of the shard column`

划分列的类型暂时不支持 ENUM、BIT、SET、JSON 类型，请尝试重新指定一个划分列。推荐使用整数或字符串类型的列。如果划分列不是这些类型，请从 PingCAP 官方或TiDB 社区获取支持。

非事务 DELETE 出现和普通的 DELETE 不等价的“异常”行为

非事务 DML 语句和这个 DML 语句的原始形式并不等价，这可能是由以下原因导致的：

- 有并发的其它写入。
- 非事务 DML 语句修改了语句自身会读取的值。
- 在每个batch 上实际执行的 SQL 语句由于改变了 WHERE 条件，可能会导致执行计划以及表达式计算顺序不同，由此导致了执行结果不一样。
- DML 语句中含有非确定性的操作。

4.8.10.5.11 兼容信息

非事务语句是TiDB 独有的功能，与 MySQL 不兼容。

4.8.10.5.12 探索更多

- BATCH 语法
- tidb_nontransactional_ignore_error

4.8.11 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。

4.8.11.1 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。

4.8.11.2 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。

4.8.11.2.1 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)
```

4.8.11.2.2 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       |
```

```
+--
     <-->-------------+--------------+-------------+----------------------
```

```
   "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
}
```


4.8.11.3 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)
```

4.8.11.4 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]。

4.8.11.5 扩展阅读

- 创建视图
- 删除视图

4.8.12 分区表

本文介绍TiDB 的分区表。

4.8.12.1 分区类型

本节介绍TiDB 中的分区类型。当前支持的类型包括Range 分区、 Range COLUMNS 分区、 Range INTERVAL 分区、 List 分区、 List COLUMNS 分区、 Hash 分区和Key 分区。

- Range 分区、 Range COLUMNS 分区、 List 分区和 List COLUMNS 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 分区和 Key 分区可以用于大量写入场景下的数据打散。与 Hash 分区相比，Key 分区支持多列打散和非整数类型字段的打散。

4.8.12.1.1 Range 分区

一个表按Range 分区是指，对于表的每个分区中包含的所有行，按分区表达式计算的值都落在给定的范围内。Range 必须是连续的，并且不能有重叠，通过使用 VALUES LESS THAN 进行定义。

下列场景中，假设你要创建一个人事记录的表：

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

你可以根据需求按各种方式进行Range 分区。其中一种方式是按 store_id 列进行分区：

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

```
PARTITION BY RANGE (store_id)
    ( PARTITION p0 VALUES LESS THAN
    (6), PARTITION p1 VALUES LESS THAN
    (11), PARTITION p2 VALUES LESS THAN
    (16), PARTITION p3 VALUES LESS THAN
    (21)
);
```

在这个分区模式中，所有 store_id 为 1 到 5 的员工，都存储在分区 p0 里面，store_id 为 6 到 10 的员工则存储在分区 p1 里面。Range 分区要求，分区的定义必须是有序的，按从小到大递增。

新插入一行数据 (72, 'Tom', 'John', '2015-06-25', NULL, NULL, 15) 将会落到分区 p2 里面。但如果你插入一条 store_id 大于 20 的记录，则会报错，因为TiDB 无法知晓应该将它插入到哪个分区。这种情况下，可以在建表时使用最大值：

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

PARTITION BY RANGE (store_id)
    ( PARTITION p0 VALUES LESS THAN
    (6), PARTITION p1 VALUES LESS THAN
    (11), PARTITION p2 VALUES LESS THAN
    (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

MAXVALUE 表示一个比所有整数都大的整数。现在，所有 store_id 列大于等于 16 的记录都会存储在 p3 分区中。

你也可以按员工的职位编号进行分区，也就是使用 job_code 列的值进行分区。假设两位数字编号是用于普通员工，三位数字编号是用于办公室以及客户支持，四位数字编号是管理层职位，那么你可以这样建表：

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

```
PARTITION BY RANGE (job_code)
    ( PARTITION p0 VALUES LESS THAN
    (100),
    PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (10000)
);
```

在这个例子中，所有普通员工存储在 p0 分区，办公室以及支持人员在 p1 分区，管理者在 p2 分区。

除了可以按 store_id 切分，你还可以按日期切分。例如，假设按员工离职的年份进行分区：

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

PARTITION BY RANGE ( YEAR(separated) )
    ( PARTITION p0 VALUES LESS THAN
    (1991), PARTITION p1 VALUES LESS THAN
    (1996), PARTITION p2 VALUES LESS THAN
    (2001), PARTITION p3 VALUES LESS THAN
    MAXVALUE
);
```

在Range 分区中，可以基于 timestamp 列的值分区，并使用 unix_timestamp() 函数，例如：

```
CREATE TABLE quarterly_report_status
    ( report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)

PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

对于timestamp 列，使用其它的分区表达式是不允许的。

Range 分区在下列条件之一或者多个都满足时，尤其有效：

- 删除旧数据。如果你使用之前的 employees 表的例子，你可以简单使用 ALTER TABLE employees DROP PARTITION p0; 删除所有在 1991 年以前停止继续在这家公司工作的员工记录。这会比使用 DELETE FROM employees WHERE YEAR(separated)<= 1990; 执行快得多。
- 使用包含时间或者日期的列，或者是其它按序生成的数据。
- 频繁查询分区使用的列。例如执行这样的查询 EXPLAIN SELECT COUNT(*)FROM employees WHERE separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id; 时，TiDB 可以迅速确定，只需要扫描 p2 分区的数据，因为其它的分区不满足 where 条件。

4.8.12.1.2 Range COLUMNS 分区

Range COLUMNS 分区是Range 分区的一种变体。你可以使用一个或者多个列作为分区键，分区列的数据类型可以是整数(integer)、字符串（CHAR/VARCHAR），DATE 和 DATETIME。不支持使用任何表达式。

假设你想要按名字进行分区，并且能够轻松地删除旧的无效数据，那么你可以创建一个表格，如下所示：

```
CREATE TABLE t (
  valid_until datetime,
  name varchar(255) CHARACTER SET ascii,
  notes text
)
PARTITION BY RANGE COLUMNS(name, valid_until)
(PARTITION `p2022-g` VALUES LESS THAN ('G','2023-01-01 00:00:00'),
 PARTITION `p2023-g` VALUES LESS THAN ('G','2024-01-01 00:00:00'),
 PARTITION `p2022-m` VALUES LESS THAN ('M','2023-01-01 00:00:00'),
 PARTITION `p2023-m` VALUES LESS THAN ('M','2024-01-01 00:00:00'),
 PARTITION `p2022-s` VALUES LESS THAN ('S','2023-01-01 00:00:00'),
 PARTITION `p2023-s` VALUES LESS THAN ('S','2024-01-01 00:00:00'))
```

该语句将按名字和年份的范围 [('', ''), ('G', '2023-01-01 00:00:00')), [('G', '2023-01-01 00:00:00'), ('G', '2024-01-01 00:00:00')), [('G', '2024-01-01 00:00:00'), ('M', '2023-01-01 00:00:00')), [('M', '2023-01-01 00:00:00'), ('M', '2024-01-01 00:00:00')), [('M', '2024-01-01 00:00:00'), ('S', '2023-01-01 00:00:00')), [('S', '2023-01-01 00:00:00'), ('S', '2024-01-01 00:00:00')) 进行分区，删除无效数据，同时仍然可以在name 和 valid_until 列上进行分区裁剪。其中，[,) 是一个左闭右开区间，比如[('G', '2023-01-01 00:00:00'), ('G', '2024-01-01 00:00:00'))，表示 name 为 'G'，年份包含 2023-01-01 00:00:00 并大于 2023-01-01 00:00:00 但小于 2024-01-01 00:00:00 的数据，其中不包含 (G, 2024-01-01 00:00:00)。

4.8.12.1.3 Range INTERVAL 分区

TiDB v6.3.0 新增了Range INTERVAL 分区特性，作为语法糖（syntactic sugar）引入。Range INTERVAL 分区是对Range 分区的扩展。你可以使用特定的间隔（interval）轻松创建分区。

其语法如下：

```
PARTITION BY RANGE [COLUMNS] (<partitioning expression>)
INTERVAL (<interval expression>)
FIRST PARTITION LESS THAN (<expression>)
LAST PARTITION LESS THAN (<expression>)
[NULL PARTITION]
[MAXVALUE PARTITION]
```

示例：

```
CREATE TABLE employees (
    id int unsigned NOT NULL,
    fname varchar(30),
    lname varchar(30),
    hired date NOT NULL DEFAULT '1970-01-01',
    separated date DEFAULT '9999-12-31',
    job_code int,
    store_id int NOT NULL
) PARTITION BY RANGE (id)
INTERVAL (100) FIRST PARTITION LESS THAN (100) LAST PARTITION LESS THAN (10000) MAXVALUE
    <↔ PARTITION
```

该示例创建的表与如下 SQL 语句相同：

```
CREATE TABLE `employees` (
  `id` int unsigned NOT NULL,
  `fname` varchar(30) DEFAULT NULL,
  `lname` varchar(30) DEFAULT NULL,
  `hired` date NOT NULL DEFAULT '1970-01-01',
  `separated` date DEFAULT '9999-12-31',
  `job_code` int DEFAULT NULL,
  `store_id` int NOT NULL
)
PARTITION BY RANGE (`id`)
(PARTITION `P_LT_100` VALUES LESS THAN (100),
 PARTITION `P_LT_200` VALUES LESS THAN (200),
...
 PARTITION `P_LT_9900` VALUES LESS THAN (9900),
 PARTITION `P_LT_10000` VALUES LESS THAN (10000),
 PARTITION `P_MAXVALUE` VALUES LESS THAN (MAXVALUE))
```

Range INTERVAL 还可以配合Range COLUMNS 分区一起使用。如下面的示例：

```
CREATE TABLE monthly_report_status
    ( report_id int NOT NULL,
    report_status varchar(20) NOT NULL,
    report_date date NOT NULL
```

```
)
PARTITION BY RANGE COLUMNS (report_date)
INTERVAL (1 MONTH) FIRST PARTITION LESS THAN ('2000-01-01') LAST PARTITION LESS THAN ('2025-01-01
    <↪ ')
```

该示例创建的表与如下 SQL 语句相同：

```
CREATE TABLE `monthly_report_status` (
  `report_id` int(11) NOT NULL,
  `report_status` varchar(20) NOT NULL,
  `report_date` date NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
PARTITION BY RANGE COLUMNS(`report_date`)
(PARTITION `P_LT_2000-01-01` VALUES LESS THAN ('2000-01-01'),
 PARTITION `P_LT_2000-02-01` VALUES LESS THAN ('2000-02-01'),
...
 PARTITION `P_LT_2024-11-01` VALUES LESS THAN ('2024-11-01'),
 PARTITION `P_LT_2024-12-01` VALUES LESS THAN ('2024-12-01'),
 PARTITION `P_LT_2025-01-01` VALUES LESS THAN ('2025-01-01'))
```

可选参数 NULL PARTITION 会创建一个分区，其中分区表达式推导出的值为 NULL 的数据会放到该分区。在分区表达式中，NULL 会被认为是小于任何其他值。参见分区对 NULL 值的处理。

可选参数 MAXVALUE PARTITION 会创建一个最后的分区，其值为 PARTITION P_MAXVALUE VALUES LESS THAN (<↪ MAXVALUE)。

ALTER INTERVAL 分区

INTERVAL 分区还增加了添加和删除分区的更加简单易用的语法。

下面的语句会变更第一个分区，该语句会删除所有小于给定表达式的分区，使匹配的分区成为新的第一个分区。它不会影响 NULL PARTITION。

```
ALTER TABLE table_name FIRST PARTITION LESS THAN (<expression>)
```

下面的语句会变更最后一个分区，该语句会添加新的分区，分区范围扩大到给定的表达式的值。如果存在 MAXVALUE PARTITION，则该语句不会生效，因为它需要数据重组。

```
ALTER TABLE table_name LAST PARTITION LESS THAN (<expression>)
```

INTERVAL 分区相关细节和限制

- INTERVAL 分区特性仅涉及 CREATE/ALTER TABLE 语法。元数据保持不变，因此使用该新语法创建或变更的表仍然兼容 MySQL。
- 为保持兼容 MySQL，SHOW CREATE TABLE 的输出格式保持不变。
- 遵循INTERVAL 的存量表可以使用新的 ALTER 语法。不需要使用 INTERVAL 语法重新创建这些表。
- 对于 RANGE COLUMNS，仅支持整数 (INTEGER) 类型、 日期(DATE) 和日期时间(DATETIME) 列类型。

4.8.12.1.4 List 分区

在创建 List 分区表之前，需要先将 session 变量 `tidb_enable_list_partition` 的值设置为 ON。

```
set @@session.tidb_enable_list_partition = ON
```

此外，还需保证 `tidb_enable_table_partition` 变量已开启（默认开启）。

List 分区和Range 分区有很多相似的地方。不同之处主要在于 List 分区中，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属于给定的数据集合。每个分区定义的数据集合有任意个值，但不能有重复的值，可通过 `PARTITION ... VALUES IN (...)` 子句对值进行定义。

假设你要创建一张人事记录表，示例如下：

```
CREATE TABLE employees
    ( id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
);
```

假如一共有 20 个商店分布在 4 个地区，如下表所示：

```
| Region   | Store ID Numbers     |
| -------- | -------------------- |
| North    | 1, 2, 3, 4, 5        |
| East     | 6, 7, 8, 9, 10       |
| West     | 11, 12, 13, 14, 15   |
| Central | 16, 17, 18, 19, 20    |
```

如果想把同一个地区商店员工的人事数据都存储在同一个分区中，你可以根据 `store_id` 来创建 List 分区：

```
CREATE TABLE employees
    ( id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);
```

这样就能方便地在表中添加或删除与特定区域相关的记录。例如，假设东部地区 (East) 所有的商店都卖给了另一家公司，所有该地区商店员工相关的行数据都可以通过`ALTER TABLE employees TRUNCATE PARTITION pEast`删除，这比等效的 `DELETE` 语句 `DELETE FROM employees WHERE store_id IN (6, 7, 8, 9, 10)` 执行起来更加高效。

使用 `ALTER TABLE employees DROP PARTITION pEast` 也能删除所有这些行，但同时也会从表的定义中删除分区 `pEast`。那样你还需要使用 `ALTER TABLE ... ADD PARTITION` 语句来还原表的原始分区方案。

与 Range 分区的情况不同，List 分区没有类似的 MAXVALUE 分区来存储所有不属于其他partition 的值。分区表达式的所有期望值都应包含在 PARTITION ... VALUES IN (...) 子句中。如果 INSERT 语句要插入的值不匹配分区的列值，该语句将执行失败并报错，如下例所示：

```
test> CREATE TABLE t (
    ->   a INT,
    ->   b INT
    -> )
    -> PARTITION BY LIST (a) (
    ->   PARTITION p0 VALUES IN (1, 2, 3),
    ->   PARTITION p1 VALUES IN (4, 5, 6)
    -> );
Query OK, 0 rows affected (0.11 sec)

test> INSERT INTO t VALUES (7, 7);
ERROR 1525 (HY000): Table has no partition for value 7
```

要忽略以上类型的错误，可以通过使用 IGNORE 关键字。使用该关键字后，就不会插入包含不匹配分区列值的行，但是会插入任何具有匹配值的行，并且不会报错：

```
test> TRUNCATE t;
Query OK, 1 row affected (0.00 sec)

test> INSERT IGNORE INTO t VALUES (1, 1), (7, 7), (8, 8), (3, 3), (5, 5);
Query OK, 3 rows affected, 2 warnings (0.01 sec)
Records: 5  Duplicates: 2  Warnings: 2

test> select * from t;
+------+------+
| a    | b    |
+------+------+
|    5 |    5 |
|    1 |    1 |
|    3 |    3 |
+------+------+
3 rows in set (0.01 sec)
```

4.8.12.1.5 List COLUMNS 分区

List COLUMNS 分区是 List 分区的一种变体，可以将多个列用作分区键，并且可以将整数类型以外的数据类型的列用作分区列。你还可以使用字符串类型、DATE 和 DATETIME 类型的列。

假设商店员工分别来自以下 12 个城市，想要根据相关规定分成 4 个区域，如下表所示：

```
| Region | Cities                          |
| :------ | ------------------------------- |
| 1       | LosAngeles,Seattle, Houston     |
```

```
| 2       | Chicago, Columbus, Boston       |
| 3       | NewYork, LongIsland, Baltimore |
| 4       | Atlanta, Raleigh, Cincinnati    |
```

使用列表列分区，你可以为员工数据创建一张表，将每行数据存储在员工所在城市对应的分区中，如下所示：

```
CREATE TABLE employees_1
    ( id  INT  NOT  NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT,
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(city) (
    PARTITION pRegion_1 VALUES IN('LosAngeles', 'Seattle', 'Houston'),
    PARTITION pRegion_2 VALUES IN('Chicago', 'Columbus', 'Boston'),
    PARTITION pRegion_3 VALUES IN('NewYork', 'LongIsland', 'Baltimore'),
    PARTITION pRegion_4 VALUES IN('Atlanta', 'Raleigh', 'Cincinnati')
);
```

与 List 分区不同的是，你不需要在 COLUMNS() 子句中使用表达式来将列值转换为整数。

List COLUMNS 分区也可以使用 DATE 和 DATETIME 类型的列进行分区，如以下示例中所示，该示例使用与先前的 employees_1 表相同的名称和列，但根据 hired 列采用 List COLUMNS 分区：

```
CREATE TABLE employees_2
    ( id  INT  NOT  NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT,
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(hired) (
    PARTITION pWeek_1 VALUES IN('2020-02-01', '2020-02-02', '2020-02-03',
        '2020-02-04', '2020-02-05', '2020-02-06', '2020-02-07'),
    PARTITION pWeek_2 VALUES IN('2020-02-08', '2020-02-09', '2020-02-10',
        '2020-02-11', '2020-02-12', '2020-02-13', '2020-02-14'),
    PARTITION pWeek_3 VALUES IN('2020-02-15', '2020-02-16', '2020-02-17',
        '2020-02-18', '2020-02-19', '2020-02-20', '2020-02-21'),
    PARTITION pWeek_4 VALUES IN('2020-02-22', '2020-02-23', '2020-02-24',
        '2020-02-25', '2020-02-26', '2020-02-27', '2020-02-28')
```

```
);
```

另外，你也可以在 COLUMNS() 子句中添加多个列，例如：

```
CREATE TABLE t (
    id int,
    name varchar(10)
)
PARTITION BY LIST COLUMNS(id,name) (
     partition p0 values IN ((1,'a'),(2,'b')),
     partition p1 values IN ((3,'c'),(4,'d')),
     partition p3 values IN ((5,'e'),(null,null))
);
```

4.8.12.1.6 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。

创建 Hash 分区表时，需要在 CREATE TABLE 后面添加 PARTITION BY HASH (expr)，其中 expr 是一个返回整数的表达式。当这一列的类型是整数类型时，它可以是一个列名。此外，你很可能还需要加上 PARTITIONS num，其中 num 是一个正整数，表示将表划分多少分区。

下面的语句将创建一个 Hash 分区表，按 store_id 分成 4 个分区：

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

PARTITION BY HASH(store_id)
PARTITIONS 4;
```

如果不指定 PARTITIONS num，默认的分区数量为 1。

你也可以使用一个返回整数的 SQL 表达式。例如，你可以按入职年份分区：

```
CREATE TABLE employees
    ( id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
```

```
    job_code INT,
    store_id INT
)

PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

最高效的 Hash 函数是作用在单列上，并且函数的单调性是跟列的值是一样递增或者递减的。

例如，date_col 是类型为 DATE 的列，表达式 TO_DAYS(date_col) 的值是直接随 date_col 的值变化的。YEAR(<⟶ date_col) 跟 TO_DAYS(date_col) 就不太一样，因为不是每次 date_col 变化时 YEAR(date_col) 都会得到不同的值。

作为对比，假设我们有一个类型是INT 的 int_col 的列。考虑一下表达式 POW(5-int_col,3)+ 6，这并不是一个比较好的 Hash 函数，因为随着 int_col 的值的变化，表达式的结果不会成比例地变化。改变 int_col 的值会使表达式的结果的值变化巨大。例如，int_col 从 5 变到 6 表达式的结果变化是-1，但是从 6 变到 7 的时候表达式的值的变化是-7。

总而言之，表达式越接近y = cx 的形式，它越是适合作为 Hash 函数。因为表达式越是非线性的，在各个分区上面的数据的分布越是倾向于不均匀。

理论上，Hash 分区也是可以做分区裁剪的。而实际上对于多列的情况，实现很难并且计算很耗时。因此，不推荐 Hash 分区在表达式中涉及多列。

使用 PARTITIION BY HASH 的时候，TiDB 通过表达式的结果做“取余”运算，决定数据落在哪个分区。换句话说，如果分区表达式是 expr，分区数是 num，则由 MOD(expr, num) 决定存储的分区。假设 t1 定义如下：

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY HASH( YEAR(col3) )
    PARTITIONS 4;
```

向 t1 插入一行数据，其中 col3 列的值是‘2005-09-15’，这条数据会被插入到分区 1 中：

```
MOD(YEAR('2005-09-01'),4)
=  MOD(2005,4)
=  1
```

4.8.12.1.7 Key 分区

TiDB 从 v7.0.0 开始支持 Key 分区。在 v7.0.0 之前的版本中，创建 Key 分区表时，TiDB 会将其创建为非分区表并给出告警。

Key 分区与 Hash 分区都可以保证将数据均匀地分散到一定数量的分区里面，区别是 Hash 分区只能根据一个指定的整数表达式或字段进行分区，而 Key 分区可以根据字段列表进行分区，且 Key 分区的分区字段不局限于整数类型。TiDB Key 分区表的 Hash 算法与 MySQL 不一样，因此表的数据分布也不一样。

创建 Key 分区表时，你需要在 CREATE TABLE 后面添加 PARTITION BY KEY (columList)，其中 columnList 是字段列表，可以包含一个或多个字段。每个字段的类型可以是除 BLOB、JSON、GEOMETRY 之外的任意类型（请注意 TiDB 不支持 GEOMETRY 类型）。此外，你很可能还需要加上 PARTITIONS num，其中 num 是一个正整数，表示

将表划分多少个分区；或者加上分区名的定义，例如，加上 (PARTITION p0, PARTITION p1) 代表将表划分为两个分区，分区名为 p0 和 p1。

下面的语句将创建一个 Key 分区表，按 store_id 分成 4 个分区：

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

PARTITION BY KEY(store_id)
PARTITIONS 4;
```

如果不指定 PARTITIONS num，默认的分区数量为 1。

你也可以根据VARCHAR 等非整数字段创建 Key 分区表。下面的语句按 fname 将表分成 4 个分区：

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

PARTITION BY KEY(fname)
PARTITIONS 4;
```

你还可以根据多列字段创建 Key 分区表。下面的语句按 fname、 store_id 将表分成 4 个分区：

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

PARTITION BY KEY(fname, store_id)
```

```
PARTITIONS 4;
```

目前，TiDB 不支持分区字段列表 PARTITION BY KEY 为空的 Key 分区表。下面的语句将创建一个非分区表，并向客户端返回 Unsupported partition type KEY, treat as normal table 警告。

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

PARTITION BY KEY()
PARTITIONS 4;
```

4.8.12.1.8 TiDB 对 Linear Hash 分区的处理

在 v6.4.0 之前，如果在TiDB 上执行 MySQL Linear Hash 分区的 DDL 语句，TiDB 只能创建非分区表。在这种情况下，如果你仍然想要在TiDB 中创建分区表，你需要修改这些 DDL 语句。

从 v6.4.0 起，TiDB 支持解析 MySQL 的 PARTITION BY LINEAR HASH 语法，但会忽略其中的 LINEAR 关键字。你可以直接在TiDB 中执行现有的 MySQL Linear Hash 分区的 SQL 语句，而无需修改。

- 对于 MySQL Linear Hash 分区的 CREATE 语句，TiDB 将创建一个常规的非线性 Hash 分区表（注意TiDB 内部实际不存在 Linear Hash 分区表）。如果分区数是 2 的幂，该分区表中行的分布情况与 MySQL 相同。如果分区数不是 2 的幂，该分区表中行的分布情况与 MySQL 会有所差异。这是因为TiDB 中非线性分区表使用简单的“分区模数”，而线性分区表使用“模数的下一个 2 次方并会折叠分区数和下一个 2 次方之间的值”。详情请见#38450。
- 对于 MySQL Linear Hash 分区的其他 SQL 语句，TiDB 将正常返回对应的 Hash 分区的查询结果。但当分区数不是 2 的幂（意味着分区表中行的分布情况与 MySQL 不同）时，分区选择、TRUNCATE PARTITION、EXCHANGE PARTITION 返回的结果将和 MySQL 有所差异。

4.8.12.1.9 TiDB 对 Linear Key 分区的处理

TiDB 从 v7.0.0 开始支持 Key 分区，并支持解析 MySQL 的 PARTITION BY LINEAR KEY 语法，但会忽略其中的 LINEAR 关键字，只采用非线性 Hash 算法。

在 v7.0.0 之前的版本中，创建 Key 分区表时，TiDB 会将其创建为非分区表并给出告警。

4.8.12.1.10 分区对 NULL 值的处理

TiDB 允许计算结果为 NULL 的分区表达式。注意，NULL 不是一个整数类型，NULL 小于所有的整数类型值，正如 ORDER BY 的规则一样。

Range 分区对 NULL 的处理

如果插入一行到Range 分区表，它的分区列的计算结果是 NULL，那么这一行会被插入到最小的那个分区。

```
CREATE TABLE t1
    ( c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY RANGE(c1) (
    PARTITION p0 VALUES LESS THAN (0),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
select * from t1 partition(p0);
```

```
+------|-------------+
| c1   | c2      |
+------|-------------+
| NULL | mothra |
+------|-------------+
1 row in set (0.00 sec)
```

```
select * from t1 partition(p1);
```

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

```
select * from t1 partition(p2);
```

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

删除 p0 后验证：

```
alter table t1 drop partition p0;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
select * from t1;
```

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

Hash 分区对 NULL 的处理

在 Hash 分区中 NULL 值的处理有所不同，如果分区表达式的计算结果为 NULL，它会被当作 0 值处理。

```
CREATE TABLE th
    ( c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY HASH(c1)
PARTITIONS 2;
```

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

```
INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
```

```
Query OK, 2 rows affected (0.04 sec)
```

```
select * from th partition (p0);
```

```
+------|-------------+
| c1   | c2      |
+------|-------------+
| NULL | mothra |
|    0 | gigan  |
+------|-------------+
2 rows in set (0.00 sec)
```

```
select * from th partition (p1);
```

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

可以看到，插入的记录 `(NULL, 'mothra')` 跟 `(0, 'gigan')` 落在了同一个分区。

> **注意：**
>
> 这里 Hash 分区对 NULL 的处理跟 MySQL 的文档描述一致，但是跟 MySQL 的实际行为并不一致。也就是说，MySQL 的文档跟它的实现并不一致。
>
> TiDB 的最终行为以本文档描述为准。

Key 分区对 NULL 的处理

在 Key 分区中 NULL 值的处理与 Hash 分区一致：如果分区字段的值为 NULL，它会被当作 0 值处理。

4.8.12.2 分区管理

对于 RANGE、 RANGE COLUMNS、 LIST、 LIST COLUMNS 分区表，你可以进行以下分区管理操作：

- 使用 ALTER TABLE <表名> ADD PARTITION (<分区说明>) 语句添加分区。
- 使用 ALTER TABLE <表名> DROP PARTITION <分区列表> 删除分区。
- 使用 ALTER TABLE <表名> TRUNCATE PARTITION <分区列表> 语句清空分区里的数据。TRUNCATE PARTITION 的逻辑与TRUNCATE TABLE 相似，但它的操作对象为分区。
- 使用 ALTER TABLE <表名> REORGANIZE PARTITION <分区列表> INTO (<新的分区说明>)语句对分区进行合并、拆分、或者其他修改。

对于 HASH 和 KEY 分区表，你可以进行以下分区管理操作：

- 使用 ALTER TABLE <table name> COALESCE PARTITION <要减少的分区数量> 语句减少分区数量。此操作会重组分区，将所有数据按照新的分区个数复制到对应的分区。
- 使用ALTER TABLE <table name> ADD PARTITION <要增加的分区数量 | (新的分区说明)> 语句增加分区的数量。此操作会重组分区，将所有数据按照新的分区个数复制到对应的分区。
- 使用 ALTER TABLE <table name> TRUNCATE PARTITION <分区列表> 语句清空分区里的数据。TRUNCATE PARTITION 的逻辑与TRUNCATE TABLE 相似，但它的操作对象为分区。

EXCHANGE PARTITION 语句用来交换分区和非分区表，类似于重命名表如 RENAME TABLE t1 TO t1_tmp, t2 TO t1, t1_tmp TO t2 的操作。

例如，ALTER TABLE partitioned_table EXCHANGE PARTITION p1 WITH TABLE non_partitioned_table 交换的是 p1 分区的 partitioned_table 表和 non_partitioned_table 表。

确保要交换入分区中的所有行与分区定义匹配；否则，交换将失败。

请注意对于以下TiDB 专有的特性，当表结构中包含这些特性时，在TiDB 中使用 EXCHANGE PARTITION 功能不仅需要满足 MySQL 的 EXCHANGE PARTITION 条件，还要保证这些专有特性对于分区表和非分区表的定义相同。

- Placement Rules in SQL：Placement Policy 定义相同。
- TiFlash：TiFlash Replica 数量相同。
- 聚簇索引：分区表和非分区表要么都是聚簇索引(CLUSTERED)，要么都不是聚簇索引(NONCLUSTERED)。

此外，EXCHANGE PARTITION 和其他组件兼容性上存在一些限制，需要保证分区表和非分区表的一致性：

- TiFlash：TiFlash Replica 定义不同时，无法执行 EXCHANGE PARTITION 操作。
- TiCDC：分区表和非分区表都有主键或者唯一键时，TiCDC 同步 EXCHANGE PARTITION 操作；反之TiCDC 将不会同步。
- TiDB Lightning 和 BR：使用 TiDB Lightning 导入或使用 BR 恢复的过程中，不要执行 EXCHANGE PARTITION 操作。

4.8.12.2.1 管理 List 分区、 List COLUMNS 分区、 Range 分区、 Range COLUMNS 分区

本小节将以如下 SQL 语句创建的分区表为例，介绍如何管理Range 分区和 List 分区。

```
CREATE  TABLE  members
    ( id int,
    fname varchar(255),
    lname varchar(255),
    dob date,
    data json
)
PARTITION BY RANGE (YEAR(dob)) (
 PARTITION pBefore1950 VALUES LESS THAN (1950),
 PARTITION p1950 VALUES LESS THAN (1960),
 PARTITION p1960 VALUES LESS THAN (1970),
 PARTITION p1970 VALUES LESS THAN (1980),
 PARTITION p1980 VALUES LESS THAN (1990),
 PARTITION p1990 VALUES LESS THAN (2000));

CREATE TABLE member_level
 ( id int,
 level int,
 achievements json
)
PARTITION BY LIST (level)
 ( PARTITION l1 VALUES IN
 (1), PARTITION l2 VALUES IN
 (2), PARTITION l3 VALUES IN
 (3), PARTITION l4 VALUES IN
 (4), PARTITION l5 VALUES IN
 (5));
```

删除分区

```
ALTER TABLE members DROP PARTITION p1990;

ALTER TABLE member_level DROP PARTITION l5;
```

清空分区

```
ALTER TABLE members TRUNCATE PARTITION p1980;

ALTER TABLE member_level TRUNCATE PARTITION l4;
```

添加分区

```
ALTER TABLE members ADD PARTITION (PARTITION `p1990to2010` VALUES LESS THAN (2010));

ALTER TABLE member_level ADD PARTITION (PARTITION l5_6 VALUES IN (5,6));
```

对于Range 分区表，ADD PARTITION 只能在分区列表的最后添加新的分区。与分区列表中已有的分区相比，你需要将新分区的 VALUES LESS THAN 定义为更大的值。否则，执行该语句时将会报错。

```
ALTER TABLE members ADD PARTITION (PARTITION p1990 VALUES LESS THAN (2000));
```

```
ERROR 1493 (HY000): VALUES LESS THAN value must be strictly increasing for each partition
```

重组分区

拆分分区：

```
ALTER TABLE members REORGANIZE PARTITION `p1990to2010` INTO
(PARTITION p1990 VALUES LESS THAN (2000),
 PARTITION p2000 VALUES LESS THAN (2010),
 PARTITION p2010 VALUES LESS THAN (2020),
 PARTITION p2020 VALUES LESS THAN (2030),
 PARTITION pMax VALUES LESS THAN (MAXVALUE));

ALTER TABLE member_level REORGANIZE PARTITION l5_6 INTO
(PARTITION l5 VALUES IN (5),
 PARTITION l6 VALUES IN (6));
```

合并分区：

```
ALTER TABLE members REORGANIZE PARTITION pBefore1950,p1950 INTO (PARTITION pBefore1960 VALUES
    <↪ LESS THAN (1960));

ALTER TABLE member_level REORGANIZE PARTITION l1,l2 INTO (PARTITION l1_2 VALUES IN (1,2));
```

修改分区表定义：

```
ALTER TABLE members REORGANIZE PARTITION pBefore1960,p1960,p1970,p1980,p1990,p2000,p2010,p2020,
    <↪ pMax INTO
(PARTITION p1800 VALUES LESS THAN (1900),
 PARTITION p1900 VALUES LESS THAN (2000),
 PARTITION p2000 VALUES LESS THAN (2100));

ALTER TABLE member_level REORGANIZE PARTITION l1_2,l3,l4,l5,l6 INTO
(PARTITION lOdd VALUES IN (1,3,5),
 PARTITION lEven VALUES IN (2,4,6));
```

在重组分区时，需要注意以下关键点：

- 重组分区（包括合并或拆分分区）只能修改分区定义，无法修改分区表类型。例如，无法将 List 类型修改为 Range 类型，或将Range COLUMNS 类型修改为Range 类型。
- 对于Range 分区表，你只能对表中相邻的分区进行重组：

  ```
  ALTER TABLE members REORGANIZE PARTITION p1800,p2000 INTO (PARTITION p2000 VALUES LESS THAN
      <↪ (2100));
  ```

```
ERROR 8200 (HY000): Unsupported REORGANIZE PARTITION of RANGE; not adjacent partitions
```

- 对于 Range 分区表，如需修改 Range 定义中的最大值，必须保证 VALUES LESS THAN 中新定义的值大于现有分区中的所有值。否则，TiDB 将报错，提示现有的行值对应不到分区。

```
INSERT INTO members VALUES (313, "John", "Doe", "2022-11-22", NULL);
ALTER TABLE members REORGANIZE PARTITION p2000 INTO (PARTITION p2000 VALUES LESS THAN (2050)
    <⟷ ); -- 执行成功，因为 2050 包含了现有的所有行
ALTER TABLE members REORGANIZE PARTITION p2000 INTO (PARTITION p2000 VALUES LESS THAN (2020)
    <⟷ ); -- 执行失败，因为 2022 将对应不到分区
```

```
ERROR 1526 (HY000): Table has no partition for value 2022
```

- 对于 List 分区表，如需修改分区定义中的数据集合，必须保证新的数据集合能覆盖到该分区中现有的所有值，否则 TiDB 将报错。

```
INSERT INTO member_level (id, level) values (313, 6);
ALTER TABLE member_level REORGANIZE PARTITION lEven INTO (PARTITION lEven VALUES IN (2,4));
```

```
ERROR 1526 (HY000): Table has no partition for value 6
```

- 分区重组后，相应分区的统计信息将会过期，并返回以下警告。此时，你可以通过 ANALYZE TABLE（/sql-statements/sql-statement-analyze-table.md）语句更新统计信息。

```
+--
    <⟷--------+-------+-----------------------------------------------------------------------------------------------------------------
    <⟷
| Level   | Code | Message
    <⟷
    <⟷ |
+--
    <⟷--------+-------+-----------------------------------------------------------------------------------------------------------------
    <⟷
 | Warning | 1105 | The statistics of related partitions will be outdated after reorganizing
     <⟷ partitions. Please use 'ANALYZE TABLE' statement if you want to update it now |
+--
    <⟷--------+-------+-----------------------------------------------------------------------------------------------------------------
    <⟷
1 row in set (0.00 sec)
```

4.8.12.2.2 管理 Hash 分区和 Key 分区

本小节将以如下 SQL 语句创建的分区表为例，介绍如何管理 Hash 分区。对于 Key 分区，你也可以使用与 Hash 分区相同的分区管理语句。

```
CREATE TABLE example
  ( id INT PRIMARY
  KEY, data
  VARCHAR(1024)
)
PARTITION BY HASH(id)
PARTITIONS 2;
```

增加分区数量

将 example 表的分区个数增加 1 个 (从 2 增加到 3):

```
ALTER TABLE example ADD PARTITION PARTITIONS 1;
```

你也可以通过添加分区定义来指定分区选项。例如，你可以通过以下语句将分区数量从 3 增加到 5，并指定新增的分区名为 pExample4 和 pExample5:

```
ALTER TABLE example ADD PARTITION
(PARTITION pExample4 COMMENT = 'not p3, but pExample4 instead',
 PARTITION pExample5 COMMENT = 'not p4, but pExample5 instead');
```

减少分区数量

与 Range 和 List 分区不同，Hash 和 Key 分区不支持 DROP PARTITION，但可以使用 COALESCE PARTITION 来减少分区数量，或使用 TRUNCATE PARTITION 清空指定分区的所有数据。

将 example 表的分区个数减少 1 个 (从 5 减少到 4):

```
ALTER TABLE example COALESCE PARTITION 1;
```

注意:

更改 Hash 和 Key 分区表的分区个数的过程会重组分区，将所有数据按照新的分区个数复制到对应的分区。因此，更改 Hash 和 Key 分区表的分区个数后，会遇到以下关于过时统计信息的警告。此时，你可以通过ANALYZE TABLE 语句更新统计信息。

```
+--
    <->        +       +
    <->
| Level    | Code | Message
    <->
    <-> |
+--
    <->        +       +
    <->
| Warning | 1105 | The statistics of related partitions will be outdated after
    <-> reorganizing partitions. Please use 'ANALYZE TABLE' statement if you want
    <-> to update it now |
```

```
+--
    <-->-------+------+-----------------------------------------------------------------------------------------------------------------------------------
    <-->
1 row in set (0.00 sec)
```

为了更好地理解 example 表重组后的结构，你可以查看重新创建 example 表所使用的 SQL 语句，如下所示：

```
SHOW CREATE TABLE\G
```

```
*************************** 1. row ***************************
       Table: example
Create Table: CREATE TABLE `example` (
  `id` int(11) NOT NULL,
  `data` varchar(1024) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
PARTITION BY HASH (`id`)
(PARTITION `p0`,
 PARTITION `p1`,
 PARTITION `p2`,
 PARTITION `pExample4` COMMENT 'not p3, but pExample4 instead')
1 row in set (0.01 sec)
```

清空分区

清空指定分区的所有数据：

```
ALTER TABLE example TRUNCATE PARTITION p0;
```

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

4.8.12.3 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)

分区裁剪暂不支持 LIKE 语句。

4.8.12.3.1 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 和 Key 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 id > 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 dt > '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 ts > '2020-04-18 02:00:42.123';
```

4.8.12.4 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。

```
SET @@sql_mode = '';

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。

4.8.12.5 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。

4.8.12.5.1 分区键，主键和唯一键

本节讨论分区键，主键和唯一键之间的关系。一句话总结它们之间的关系要满足的规则：分区表的每个唯一键，必须包含分区表达式中用到的所有列。

> every unique key on the table must use every column in the table’s partitioning expression.

这里所指的唯一也包含了主键，因为根据主键的定义，主键必须是唯一的。例如，下面这些建表语句就是无效的：

```
CREATE TABLE t1 (
    col1  INT  NOT  NULL,
    col2 DATE NOT NULL,
    col3  INT  NOT  NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
```

```
PARTITIONS 4;
```

它们都是有唯一键但没有包含所有分区键的。

下面是一些合法的语句的例子：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1  INT  NOT  NULL,
    col2 DATE NOT NULL,
    col3  INT  NOT  NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

下例中会产生一个报错：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

```
ERROR 1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

原因是 `col1` 和 `col3` 出现在分区键中，但是几个唯一键定义并没有完全包含它们，做如下修改后语句即为合法：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3),
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

下面这个表就没法做分区了，因为无论如何都不可能找到满足条件的分区键：

```
CREATE TABLE t4 (
    col1 INT NOT NULL,
    col2 INT NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3),
    UNIQUE KEY (col2, col4)
);
```

根据定义，主键也是唯一键，下面两个建表语句是无效的：

```
CREATE TABLE t5 (
    col1  INT  NOT  NULL,
    col2 DATE NOT NULL,
    col3  INT  NOT  NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
```

```
PARTITIONS 4;
```

以上两个例子中，主键都没有包含分区表达式中的全部的列，在主键中补充缺失列后语句即为合法：

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

如果既没有主键，也没有唯一键，则不存在这个限制。

DDL 变更时，添加唯一索引也需要考虑到这个限制。比如创建了这样一个表：

```
CREATE TABLE t_no_pk (c1 INT, c2 INT)
    PARTITION BY RANGE(c1) (
        PARTITION p0 VALUES LESS THAN (10),
        PARTITION p1 VALUES LESS THAN (20),
        PARTITION p2 VALUES LESS THAN (30),
        PARTITION p3 VALUES LESS THAN (40)
    );
```

```
Query OK, 0 rows affected (0.12 sec)
```

通过 `ALTER TABLE` 添加非唯一索引是可以的。但是添加唯一索引时，唯一索引里面必须包含 `c1` 列。

使用分区表时，前缀索引是不能指定为唯一属性的：

```
CREATE TABLE t (a varchar(20), b blob,
    UNIQUE INDEX (a(5)))
    PARTITION by range columns (a) (
```

```
    PARTITION p0 values less than ('aaaaa'),
    PARTITION p1 values less than ('bbbbb'),
    PARTITION p2 values less than ('ccccc'));
```

```
ERROR 1503 (HY000): A UNIQUE INDEX must include all columns in the table's partitioning function
```

4.8.12.5.2 关于函数的分区限制

只有以下函数可以用于分区表达式：

```
ABS()
CEILING()
DATEDIFF()
DAY()
DAYOFMONTH()
DAYOFWEEK()
DAYOFYEAR()
EXTRACT() (see EXTRACT() function with WEEK specifier)
FLOOR()
HOUR()
MICROSECOND()
MINUTE()
MOD()
MONTH()
QUARTER()
SECOND()
TIME_TO_SEC()
TO_DAYS()
TO_SECONDS()
UNIX_TIMESTAMP() (with TIMESTAMP columns)
WEEKDAY()
YEAR()
YEARWEEK()
```

4.8.12.5.3 兼容性

目前TiDB 支持Range 分区、 Range Columns 分区、 List 分区、 List COLUMNS 分区、 Hash 分区和 Key 分区，其它的MySQL 分区类型尚不支持。

对于 Key 分区，目前不支持分区字段为空的场景。

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，合并分区等。

对于暂不支持的分区类型，在TiDB 中建表时会忽略分区信息，以普通表的形式创建，并且会报Warning。

Load Data 暂时不支持分区选择。

```
create table t (id int, val int) partition by hash(id) partitions 4;
```

普通的 Load Data 操作在TiDB 中是支持的，如下：

```
load local data infile "xxx" into t ...
```

但 Load Data 不支持分区选择操作：

```
load local data infile "xxx" into t partition (p1)...
```

对于分区表，`select * from t` 的返回结果是分区之间无序的。这跟 MySQL 不同，MySQL 的返回结果是分区之间有序，分区内部无序。

```
create table t (id int, val int) partition by range (id)
    ( partition p0 values less than (3),
    partition p1 values less than (7),
    partition p2 values less than (11));
```

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

```
insert into t values (1, 2), (3, 4),(5, 6),(7,8),(9,10);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

TiDB 每次返回结果会不同，例如：

```
select * from t;
```

```
+------|----------+
| id   | val  |
+------|----------+
|    7 |    8 |
|    9 |   10 |
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
+------|----------+
5 rows in set (0.00 sec)
```

MySQL 的返回结果：

```
select * from t;
```

```
+------+----------+
| id   | val  |
+------+----------+
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
|    7 |    8 |
|    9 |   10 |
+------+----------+
5 rows in set (0.00 sec)
```

环境变量 tidb_enable_list_partition 可以控制是否启用分区表功能。如果该变量设置为 OFF，则建表时会忽略分区信息，以普通表的方式建表。

该变量仅作用于建表，已经建表之后再修改该变量无效。详见系统变量和语法。

4.8.12.5.4 动态裁剪模式

TiDB 访问分区表有两种模式，dynamic 和 static。从 v6.3.0 开始，默认使用 dynamic 模式。但是注意，dynamic 模式仅在表级别汇总统计信息（即 GlobalStats）收集完成的情况下生效。如果选择了 dynamic 但 GlobalStats 未收集完成，TiDB 会仍采用 static 模式。关于GlobalStats 更多信息，请参考动态裁剪模式下的分区表统计信息。

```
set @@session.tidb_partition_prune_mode = 'dynamic'
```

普通查询和手动analyze 使用的是 session 级别的 tidb_partition_prune_mode 设置，后台的auto-analyze 使用的是 global 级别的 tidb_partition_prune_mode 设置。

静态裁剪模式下，分区表使用的是分区级别的统计信息，而动态裁剪模式下，分区表用的是表级别的汇总统计信息。

从 static 静态裁剪模式切到 dynamic 动态裁剪模式时，需要手动检查和收集统计信息。在刚切换到 dynamic 时，分区表上仍然只有分区的统计信息，需要等到全局 dynamic 动态裁剪模式开启后的下一次 auto-analyze 周期，才会更新生成汇总统计信息。

```
set session tidb_partition_prune_mode = 'dynamic';
show stats_meta where table_name like "t";
```

```
+---------+------------+----------------+---------------------+--------------+----------------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+----------------------+
| test    | t          | p0             | 2022-05-27 20:23:34 |            1 |          2 |
| test    | t          | p1             | 2022-05-27 20:23:34 |            2 |          4 |
| test    | t          | p2             | 2022-05-27 20:23:34 |            2 |          4 |
+---------+------------+----------------+---------------------+--------------+----------------------+
3 rows in set (0.01 sec)
```

为保证开启全局 dynamic 动态裁剪模式时，SQL 可以用上正确的统计信息，此时需要手动触发一次 analyze 来更新汇总统计信息，可以通过 analyze 表或者单个分区来更新。

```
analyze table t partition p1;
show stats_meta where table_name like "t";
```

```
+---------+------------+----------------+---------------------+--------------+-----------------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------------------+
| test    | t          | global         | 2022-05-27 20:50:53 |            0 |         5 |
| test    | t          | p0             | 2022-05-27 20:23:34 |            1 |         2 |
| test    | t          | p1             | 2022-05-27 20:50:52 |            0 |         2 |
| test    | t          | p2             | 2022-05-27 20:50:08 |            0 |         2 |
+---------+------------+----------------+---------------------+--------------+-----------------------+
4 rows in set (0.00 sec)
```

若 analyze 过程中提示如下 warning，说明分区的统计信息之间存在不一致，需要重新收集分区或整个表统计信息。

```
 | Warning | 8244 | Build table: `t` column: `a` global-level stats failed due to missing
     <-> partition-level column stats, please run analyze table to refresh columns of all
    <-> partitions
```

也可以使用脚本来统一更新所有的分区表统计信息，详见为动态裁剪模式更新所有分区表的统计信息。

表级别统计信息准备好后，即可开启全局的动态裁剪模式。全局动态裁剪模式，对全局所有的 SQL 和对后台的统计信息自动收集（即 auto analyze）起作用。

```
set global tidb_partition_prune_mode = dynamic
```

在 static 模式下，TiDB 用多个算子单独访问每个分区，然后通过 Union 将结果合并起来。下面例子进行了一个简单的读取操作，可以发现TiDB 用 Union 合并了对应两个分区的结果：

```
mysql> create table t1(id int, age int, key(id)) partition by range(id) (
    ->     partition p0 values less than (100),
    ->     partition p1 values less than (200),
    ->     partition p2 values less than (300),
    ->     partition p3 values less than (400));
Query OK, 0 rows affected (0.01 sec)

mysql> explain select * from t1 where id < 150;
```

```
+                               +          +           +                         +-----------------------------------------------------------------
    <->
| id                            | estRows  | task      | access object           | operator info
    <->                 |
+                               +          +           +                         +                         ------------------
 -------------------------------  ---------  ----------  -------------------------
    <->
| PartitionUnion_9              | 6646.67  | root      |                         |
    <->                                |
```

```
| ├─TableReader_12                | 3323.33  | root      |                         | data:
    ↔ Selection_11                  |
| | └─Selection_11                | 3323.33  | cop[tikv] |                         | lt(test.t1.id,
    ↔ 150)            |
| |   └─TableFullScan_10          | 10000.00 | cop[tikv] | table:t1, partition:p0  | keep order:
    ↔ false, stats:pseudo |
| └─TableReader_18                | 3323.33  | root      |                         | data:
    ↔ Selection_17                  |
|   └─Selection_17                | 3323.33  | cop[tikv] |                         | lt(test.t1.id,
    ↔ 150)            |
|     └─TableFullScan_16          | 10000.00 | cop[tikv] | table:t1, partition:p1  | keep order:
    ↔ false, stats:pseudo |   +          +           +                         +
 ------------------------------- ---------- ---------- ------------------------- ---------------------------------------------------------------------------
    ↔
7 rows in set (0.00 sec)
```

在 dynamic 模式下，每个算子都支持直接访问多个分区，所以TiDB 不再使用 Union。

```
mysql> set @@session.tidb_partition_prune_mode = 'dynamic';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select * from t1 where id < 150;
+--
    ↔ ----------------------+----------+-----------+-----------------+-----------------------------------------------------------+
    ↔
| id                        | estRows  | task      | access object   | operator info
    ↔                 |
+--
    ↔ ----------------------+----------+-----------+-----------------+-----------------------------------------------------------+
    ↔
| TableReader_7             | 3323.33  | root      | partition:p0,p1 | data:Selection_6
    ↔               |
| └─Selection_6             | 3323.33  | cop[tikv] |                 | lt(test.t1.id, 150)
    ↔            |
|   └─TableFullScan_5       | 10000.00 | cop[tikv] | table:t1        | keep order:false, stats:
    ↔ pseudo |
+--
    ↔ ----------------------+----------+-----------+-----------------+-----------------------------------------------------------+
    ↔
3 rows in set (0.00 sec)
```

从以上查询结果可知，执行计划中的 Union 消失了，分区裁剪依然生效，且执行计划只访问了 p0 和 p1 两个分区。

dynamic 模式让执行计划更简单清晰，省略 Union 操作可提高执行效率，还可避免 Union 并发管理的问题。此外 dynamic 模式下，执行计划可以使用IndexJoin 的方式，这在 static 模式下是无法实现的。请看下面的例子：

示例一：以下示例在 static 模式下执行计划带IndexJoin 的查询。

```
mysql> create table t1 (id int, age int, key(id)) partition by range(id)
    -> (partition p0 values less than (100),
    ->  partition p1 values less than (200),
    ->  partition p2 values less than (300),
    ->  partition p3 values less than (400));
Query OK, 0 rows affected (0,08 sec)
mysql> create table t2 (id int, code int);

Query OK, 0 rows affected (0.01 sec)

mysql> set @@tidb_partition_prune_mode = 'static';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select /*+ TIDB_INLJ(t1, t2) */ t1.* from t1, t2 where t2.code = 0 and t2.id = t1.
    <-> id;
+--
    <->                               +          +           +                         +
    <-> ------------------------------ ---------- ----------- -------------------------
| id                              | estRows  | task      | access object           | operator info
    <->                                    |
+--
    <->                               +          +           +                         +
    <->
| HashJoin_13                     | 12.49    | root      |                         | inner join,
    <-> equal:[eq(test.t1.id, test.t2.id)] |
| ├─TableReader_42(Build)         | 9.99     | root      |                         | data:
    <-> Selection_41                          |
| │ └─Selection_41                | 9.99     | cop[tikv] |                         | eq(test.t2.
    <-> code, 0), not(isnull(test.t2.id))    |
| │   └─TableFullScan_40          | 10000.00 | cop[tikv] | table:t2                | keep order:
    <-> false, stats:pseudo                  |
| └─PartitionUnion_15(Probe)      | 39960.00 | root      |                         |
    <->                                               |
|   ├─TableReader_18              | 9990.00  | root      |                         | data:
    <-> Selection_17                          |
|   │ └─Selection_17              | 9990.00  | cop[tikv] |                         | not(isnull(
    <-> test.t1.id))                         |
|   │   └─TableFullScan_16        | 10000.00 | cop[tikv] | table:t1, partition:p0 | keep order:
    <-> false, stats:pseudo                  |
|   ├─TableReader_24              | 9990.00  | root      |                         | data:
    <-> Selection_23                          |
|   │ └─Selection_23              | 9990.00  | cop[tikv] |                         | not(isnull(
    <-> test.t1.id))                         |
|   │   └─TableFullScan_22        | 10000.00 | cop[tikv] | table:t1, partition:p1 | keep order:
```

```
    <-> false, stats:pseudo                  |
|   ├─TableReader_30              | 9990.00  | root      |                          | data:
    <-> Selection_29                           |
|   │ └─Selection_29              | 9990.00  | cop[tikv] |                          | not(isnull(
    <-> test.t1.id))                         |
|   │   └─TableFullScan_28        | 10000.00 | cop[tikv] | table:t1, partition:p2 | keep order:
    <-> false, stats:pseudo                  |
|   └─TableReader_36              | 9990.00  | root      |                          | data:
    <-> Selection_35                           |
|     └─Selection_35              | 9990.00  | cop[tikv] |                          | not(isnull(
    <-> test.t1.id))                         |
|       └─TableFullScan_34        | 10000.00 | cop[tikv] | table:t1, partition:p3 | keep order:
    <-> false, stats:pseudo                  |
+--
    <->--------------------------------+----------+-----------+------------------------+-------------------------
    <->
17 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+--
    <->-------+------+                                                                                    +
    <->                 ---------------------------------------------------------------------------------
    <->
| Level   | Code | Message
    <->                                                                                                  |
+--
    <->-------+------+                                                                                    +
    <->                 ---------------------------------------------------------------------------------
    <->
| Warning | 1815 | Optimizer Hint /*+ INL_JOIN(t1, t2) */ or /*+ TIDB_INLJ(t1, t2) */ is
    <-> inapplicable |
+--
    <->-------+------+                                                                                    +
    <->                 ---------------------------------------------------------------------------------
    <->
1 row in set (0,00 sec)
```

从以上示例一结果可知，即使使用了 `TIDB_INLJ` 的hint，也无法使得带分区表的查询选上带IndexJoin 的执行计划。

示例二：以下示例在 `dynamic` 模式下尝试执行计划带IndexJoin 的查询。

```
        ------------------------------ ------------------ ------------------ ------------------------------------ ------------------------
```

```
    <->
| id                              | estRows  | task      | access object          | operator info
    <->
    <->
    <-> |
+--
    <->-------------------------------+----------+-----------+------------------------+
    <->
| IndexJoin_11                    | 12.49    | root      |                        | inner join,
     <-> inner:IndexLookUp_10, outer key:test.t2.id, inner key:test.t1.id, equal cond:eq(test.t2.
    <-> id, test.t1.id) |
| ├─TableReader_16(Build)         | 9.99     | root      |                        | data:
    <-> Selection_15
    <->
    <-> |
| │ └─Selection_15                | 9.99     | cop[tikv] |                        | eq(test.t2.
    <-> code, 0), not(isnull(test.t2.id))
    <->                                                                                 |
| │   └─TableFullScan_14          | 10000.00 | cop[tikv] | table:t2               | keep order:
    <-> false, stats:pseudo
    <->                                                                                 |
| └─IndexLookUp_10(Probe)         | 12.49    | root      | partition:all          |
    <->
    <-> |
|   ├─Selection_9(Build)          | 12.49    | cop[tikv] |                        | not(isnull(
    <-> test.t1.id))
    <->
    <-> |
|   │ └─IndexRangeScan_7          | 12.50    | cop[tikv] | table:t1, index:id(id) | range:
    <-> decided by [eq(test.t1.id, test.t2.id)], keep order:false, stats:pseudo
    <->                                        |
|   └─TableRowIDScan_8(Probe)     | 12.49    | cop[tikv] | table:t1               | keep order:
    <-> false, stats:pseudo
    <->                                                                                 |
+--
    <->                               +          +           +                          +
                                                                                        ----------

    <->
8 rows in set (0.00 sec)
```

从示例二结果可知，开启 dynamic 模式后，带IndexJoin 的计划在执行查询时被选上。

目前，静态和动态裁剪模式都不支持执行计划缓存。

为动态裁剪模式更新所有分区表的统计信息

1. 找到所有的分区表：

```
SELECT DISTINCT CONCAT(TABLE_SCHEMA,'.', TABLE_NAME)
    FROM information_schema.PARTITIONS
    WHERE TIDB_PARTITION_ID IS NOT NULL
    AND TABLE_SCHEMA NOT IN ('INFORMATION_SCHEMA', 'mysql', 'sys', 'PERFORMANCE_SCHEMA', '
        <-> METRICS_SCHEMA');
```

```
+--------------------------------------+
| concat(TABLE_SCHEMA,'.',TABLE_NAME) |
+--------------------------------------+
| test.t                               |
+--------------------------------------+
1 row in set (0.02 sec)
```

2. 生成所有分区表的更新统计信息的语句：

```
select distinct concat('ANALYZE TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;')
    from information_schema.PARTITIONS
    where TABLE_SCHEMA not in ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        <-> METRICS_SCHEMA');
```

```
+-----------------------------------------------------------------------+
| concat('ANALYZE TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;') |
+-----------------------------------------------------------------------+
| ANALYZE TABLE test.t ALL COLUMNS;                                     |
+-----------------------------------------------------------------------+
1 row in set (0.01 sec)
```

可以按需将 ALL COLUMNS 改为实际需要的列。

3. 将批量更新语句导出到文件：

```
mysql --host xxxx --port xxxx -u root -p -e "select distinct concat('ANALYZE TABLE ',
    <-> TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;') \
    from information_schema.PARTITIONS \
    where TABLE_SCHEMA not in ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        <-> METRICS_SCHEMA');" | tee gatherGlobalStats.sql
```

4. 执行批量更新：

在运行 source 命令之前处理 SQL 文件：

```
sed -i "" '1d' gatherGlobalStats.sql --- mac
sed -i '1d' gatherGlobalStats.sql --- linux
```

```
SET session tidb_partition_prune_mode = dynamic;
source gatherGlobalStats.sql
```

4.8.13 临时表

TiDB 在 v5.3.0 版本中引入了临时表功能。该功能针对业务中间计算结果的临时存储问题，让用户免于频繁地建表和删表等操作。用户可将业务上的中间计算数据存入临时表，用完数据后TiDB 自动清理回收临时表。这避免了用户业务过于复杂，减少了表管理开销，并提升了性能。

本文介绍了TiDB 临时表的使用场景、临时表类型、使用示例、限制临时表内存占用的方法、与其他TiDB 功能的兼容性限制。

4.8.13.1 使用场景

TiDB 临时表主要应用于以下业务场景：

- 缓存业务的中间临时数据，计算完成后将数据转储至普通表，临时表会自动释放。
- 短期内对同一数据进行多次 DML 操作。例如在电商购物车应用中，添加、修改、删除商品及完成结算，并移除购物车信息。
- 快速批量导入中间临时数据，提升导入临时数据的性能。
- 批量更新数据。将数据批量导入到数据库的临时表，修改完成后再导出到文件。

4.8.13.2 临时表类型

TiDB 的临时表分为本地临时表和全局临时表：

- 本地临时表的表定义和表内数据只对当前会话可见，适用于暂存会话内的中间数据。
- 全局临时表的表定义对整个TiDB 集群可见，表内数据只对当前事务可见，适用于暂存事务内的中间数据。

4.8.13.3 本地临时表

本地临时表的语义与 MySQL 临时表一致，它有以下特性：

- 本地临时表的表定义不持久化，只在创建该表的会话内可见，其他会话无法访问该本地临时表
- 不同会话可以创建同名的本地临时表，各会话只会读写该会话内创建的本地临时表
- 本地临时表的数据对会话内的所有事务可见
- 在会话结束后，该会话创建的本地临时表会被自动删除
- 本地临时表可以与普通表同名，此时在 DDL 和 DML 语句中，普通表被隐藏，直到本地临时表被删除

用户可通过 `CREATE TEMPORARY TABLE` 语句创建本地临时表，通过 `DROP TABLE` 或 `DROP TEMPORARY TABLE` 语句删除本地临时表。

不同于 MySQL，TiDB 本地临时表都是外部表，SQL 语句不会创建内部临时表。

4.8.13.3.1 本地临时表使用示例

> 注意：

- 使用TiDB 中的临时表前，注意临时表与其他TiDB 功能的兼容性限制以及与 MySQL 临时表的兼容性。
- 如果在 v5.3.0 升级前创建了本地临时表，这些临时表实际为普通表，在升级后也会被 TiDB 当成普通表处理。

假设已存在普通表 users:

```
CREATE TABLE users
    ( id BIGINT,
    name VARCHAR(100),
    PRIMARY KEY(id)
);
```

在会话A 中创建本地临时表 users，不会有名字冲突。会话 A 访问 users 时，访问的是本地临时表 users。

```
CREATE TEMPORARY TABLE users
    ( id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
);
```

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

此时将数据插入 users，插入到的是会话 A 中的本地临时表 users。

```
INSERT INTO users(id, name, city) VALUES(1001, 'Davis', 'LosAngeles');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+----------------------+
| id   | name  | city                 |
+------+-------+----------------------+
| 1001 | Davis | LosAngeles |
+------+-------+----------------------+
1 row in set (0.00 sec)
```

在会话B 中创建本地临时表 users，不会与普通表 users 冲突，也不会与会话A 中的本地临时表 users 冲突。会话B 内访问 users 时，访问的是会话B 内创建的本地临时表 users 数据。

```
CREATE TEMPORARY TABLE users
    ( id BIGINT,
    name VARCHAR(100),
```

```
    city VARCHAR(50),
    PRIMARY KEY(id)
);
```

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

此时将数据插入 users，插入到的是会话 B 中的本地临时表 users。

```
INSERT INTO users(id, name, city) VALUES(1001, 'James', 'NewYork');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+---------- +
| id   | name  | city     |
+------+-------+---------- +
| 1001 | James | NewYork |
+------+-------+---------- +
1 row in set (0.00 sec)
```

4.8.13.3.2 与 MySQL 临时表的兼容性

TiDB 本地临时表的以下特性与限制与 MySQL 一致：

- 创建、删除本地临时表时，不会自动提交当前事务
- 删除本地临时表所在的 schema 后，临时表不会被删除，仍然可以读写
- 创建本地临时表需要 CREATE TEMPORARY TABLES 权限，随后对该表的所有操作不需要权限
- 本地临时表不支持外键和分区表
- 不支持基于本地临时表创建视图
- SHOW [FULL] TABLES 不显示本地临时表

TiDB 本地临时表与 MySQL 临时表有以下方面不兼容：

- TiDB 本地临时表不支持 ALTER TABLE
- TiDB 本地临时表忽略 ENGINE 表选项，始终在TiDB 内存中暂存临时表数据，并且有内存限制
- 当声明存储引擎为 MEMORY 时，TiDB 本地临时表没有 MEMORY 存储引擎的限制
- 当声明存储引擎为 INNODB 或 MYISAM 时，TiDB 本地临时表忽略InnoDB 临时表特有的系统变量
- MySQL 不允许在同一条 SQL 中多次引用同一张临时表，而TiDB 本地临时表没有该限制
- MySQL 中用于显示临时表的 information_schema.INNODB_TEMP_TABLE_INFO 表在TiDB 中不存在。TiDB 暂无用于显示本地临时表的系统表。
- TiDB 没有内部临时表，MySQL 针对内部临时表的系统变量对TiDB 不生效

4.8.13.4 全局临时表

全局临时表是TiDB 的扩展功能，它有以下特性：

- 全局临时表的表定义会持久化，对所有会话可见
- 全局临时表的数据只对当前的事务内可见，事务结束后数据自动清空
- 全局临时表不能与普通表同名

用户可通过 `CREATE GLOBAL TEMPORARY TABLE` 语句创建全局临时表，语句末尾要加上 `ON COMMIT DELETE ROWS`。可通过 `DROP TABLE` 或 `DROP GLOBAL TEMPORARY TABLE` 语句删除全局临时表。

4.8.13.4.1 全局临时表使用示例

> 注意：
>
> - 使用TiDB 中的临时表前，注意临时表与其他TiDB 功能的兼容性限制。
> - 如果在 v5.3.0 或以上版本中创建了全局临时表，这些临时表在降级后会被当作普通表处理，导致数据错误。

在会话A 中创建全局临时表 `users`：

```
CREATE GLOBAL TEMPORARY TABLE users
    ( id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
) ON COMMIT DELETE ROWS;
```

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

写入 `users` 的数据对当前事务可见：

```
BEGIN;
```

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

```
INSERT INTO users(id, name, city) VALUES(1001, 'Davis', 'LosAngeles');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+--------------------+
| id   | name  | city               |
+------+-------+--------------------+
| 1001 | Davis | LosAngeles |
+------+-------+--------------------+
1 row in set (0.00 sec)
```

事务结束后数据自动被清空：

```
COMMIT;
```

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

```
SELECT * FROM users;
```

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

会话A 创建了 users 后，会话B 也可以读写该表：

```
SELECT * FROM users;
```

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

> 注意：
>
> 如果事务是自动提交的，插入的数据在 SQL 语句执行结束后会被自动清空，导致后续 SQL 执行查找不到结果。因此应该使用非自动提交的事务读写全局临时表。

4.8.13.5 限制临时表的内存占用

无论定义表时声明的 ENGINE 是哪种存储引擎，本地临时表和全局临时表的数据都只暂存在TiDB 实例的内存中，不持久化。

为了避免内存溢出，用户可通过系统变量tidb_tmp_table_max_size 限制每张临时表的大小。当临时表大小超过限制后TiDB 会报错。tidb_tmp_table_max_size 的默认值是 64MB。

例如，将每张临时表的大小限制为 256MB：

```
SET GLOBAL tidb_tmp_table_max_size=268435456;
```

4.8.13.6 与其他TiDB 功能的兼容性限制

以下是本地临时表和全局临时表都不支持的功能：

- 不支持 AUTO_RANDOM 列
- 不支持 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 表选项
- 不支持分区表
- 不支持 SPLIT REGION 语句
- 不支持 ADMIN CHECK TABLE 和 ADMIN CHECKSUM TABLE 语句
- 不支持 FLASHBACK TABLE 和 RECOVER TABLE 语句
- 不支持以临时表为源表执行 CREATE TABLE LIKE 语句
- 不支持 Stale Read
- 不支持外键
- 不支持 SQL binding
- 不支持添加TiFlash 副本
- 不支持在临时表上创建视图
- 不支持 Placement Rules
- 包含临时表的执行计划不会被prepared plan cache 缓存

以下是只有本地临时表不支持的功能：

- 不支持通过系统变量 tidb_snapshot 读取历史数据

4.8.13.7 TiDB 数据迁移工具支持

本地临时表只对当前会话可见，因此本地临时表不会被TiDB 数据迁移工具导出、备份、同步。

全局临时表的表定义全局可见，因此全局临时表的表定义会被TiDB 数据迁移工具导出、备份、同步，但不导出数据。

> 注意：
>
> - TiCDC 必须使用 v5.3.0 及以上版本同步，否则下游集群的表定义错误
> - BR 必须使用 v5.3.0 及以上版本备份，否则备份后的表定义错误
> - 导入的集群、恢复后的集群、同步的下游集群需要支持全局临时表，否则报错

4.8.13.8 另请参阅

- CREATE TABLE
- CREATE TABLE LIKE
- DROP TABLE

4.8.14 缓存表

TiDB 在 v6.0.0 版本中引入了缓存表功能。该功能适用于频繁被访问且很少被修改的热点小表，即把整张表的数据加载到TiDB 服务器的内存中，直接从内存中获取表数据，避免从TiKV 获取表数据，从而提升读性能。

本文介绍了TiDB 缓存表的使用场景、使用示例、与其他TiDB 功能的兼容性限制。

4.8.14.1 使用场景

TiDB 缓存表功能适用于以下特点的表：

- 表的数据量不大
- 只读表，或者几乎很少修改
- 表的访问很频繁，期望有更好的读性能

当表的数据量不大，访问又特别频繁的情况下，数据会集中在TiKV 一个Region 上，形成热点，从而影响性能。因此，TiDB 缓存表的典型使用场景如下：

- 配置表，业务通过该表读取配置信息
- 金融场景中的存储汇率的表，该表不会实时更新，每天只更新一次
- 银行分行或者网点信息表，该表很少新增记录项

以配置表为例，当业务重启的瞬间，全部连接一起加载配置，会造成较高的数据库读延迟。如果使用了缓存表，则可以解决这样的问题。

4.8.14.2 使用示例

本节通过示例介绍缓存表的使用方法。

4.8.14.2.1 将普通表设为缓存表

假设已存在普通表 `users`:

```
CREATE TABLE users
    ( id BIGINT,
    name VARCHAR(100),
    PRIMARY KEY(id)
);
```

通过 `ALTER TABLE` 语句，可以将这张表设置成缓存表：

```
ALTER TABLE users CACHE;
```

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

4.8.14.2.2 验证是否为缓存表

要验证一张表是否为缓存表，使用 SHOW CREATE TABLE 语句。如果为缓存表，返回结果中会带有 CACHED ON 属性：

```
SHOW CREATE TABLE users;
```

```
+       +
    <->
| Table | Create Table
    <->
    <-> |
+       +
 --------
    <->
| users | CREATE TABLE `users` (
  `id` bigint(20) NOT NULL,
  `name` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /* CACHED ON */ |
+       +
    <->
1 row in set (0.00 sec)
```

从缓存表读取数据后，TiDB 会将数据加载到内存中。你可使用 trace 语句查看TiDB 是否已将数据加载到内存中。当缓存还未加载时，语句的返回结果会出现 regionRequest.SendReqCtx，表示 TiDB 从 TiKV 读取了数据。

```
TRACE SELECT * FROM users;
```

```
+-------------------------------------------------+-----------------+-------------+
| operation                                       | startTS         | duration    |
+-------------------------------------------------+-----------------+-------------+
| trace                                           | 17:47:39.969980 | 827.73µs    |
|   ├─session.ExecuteStmt                         | 17:47:39.969986 | 413.31µs    |
|   │ ├─executor.Compile                          | 17:47:39.969993 | 198.29µs    |
|   │ └─session.runStmt                           | 17:47:39.970221 | 157.252µs   |
|   │   └─TableReaderExecutor.Open                | 17:47:39.970294 | 47.068µs    |
|   │     └─distsql.Select                        | 17:47:39.970312 | 24.729µs    |
|   │       └─regionRequest.SendReqCtx            | 17:47:39.970454 | 189.601µs   |
|   ├─*executor.UnionScanExec.Next                | 17:47:39.970407 | 353.073µs   |
|   │ ├─*executor.TableReaderExecutor.Next        | 17:47:39.970411 | 301.106µs   |
|   │ └─*executor.TableReaderExecutor.Next        | 17:47:39.970746 | 6.57µs      |
|   └─*executor.UnionScanExec.Next                | 17:47:39.970772 | 17.589µs    |
|     └─*executor.TableReaderExecutor.Next        | 17:47:39.970776 | 6.59µs      |
+-------------------------------------------------+-----------------+-------------+
12 rows in set (0.01 sec)
```

而再次执行 trace，返回结果中不再有 regionRequest.SendReqCtx，表示 TiDB 已经不再从TiKV 读取数据，而是直接从内存中读取：

```
+----------------------------------------+----------------+------------------------+
| operation                              | startTS        | duration               |
+----------------------------------------+----------------+------------------------+
| trace                                  | 17:47:40.533888 | 453.547µs  |
|   ├─session.ExecuteStmt                 | 17:47:40.533894 | 402.341µs   |
|   │ ├─executor.Compile                  | 17:47:40.533903 | 205.54µs    |
|   │ └─session.runStmt                   | 17:47:40.534141 | 132.084µs   |
|   │   └─TableReaderExecutor.Open        | 17:47:40.534202 | 14.749µs    |
|   ├─*executor.UnionScanExec.Next        | 17:47:40.534306 | 3.21µs      |
|   └─*executor.UnionScanExec.Next        | 17:47:40.534316 | 1.219µs     |
+----------------------------------------+----------------+------------------------+
7 rows in set (0.00 sec)
```

注意，读取缓存表会使用 UnionScan 算子，所以通过 explain 查看缓存表的执行计划时，可能会在结果中看到 UnionScan：

```
+--
    <->  ----------------------+---------+-----------+---------------+ ------------------------------------------------------ +
    <->
| id                       | estRows | task      | access object | operator info
    <-> |
+--
    <->  ----------------------+---------+-----------+---------------+ ------------------------------------------------------ +
    <->
| UnionScan_5              | 1.00    | root      |               |
    <-> |
| └─TableReader_7          | 1.00    | root      |               | data:TableFullScan_6
    <->          |
|   └─TableFullScan_6      | 1.00    | cop[tikv] | table:users   | keep order:false, stats:pseudo
    <> |
+--
    <->  ----------------------+---------+-----------+---------------+ ------------------------------------------------------ +
    <->
3 rows in set (0.00 sec)
```

4.8.14.2.3 往缓存表写入数据

缓存表支持写入数据。例如，往 users 表中插入一条记录：

```
INSERT INTO users(id, name) VALUES(1001, 'Davis');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-----------+
| id   | name  |
+------+-----------+
| 1001 | Davis |
+------+-----------+
1 row in set (0.00 sec)
```

> 注意：
>
> 往缓存表写入数据时， 有可能出现秒级别的写入延迟。延迟的时长由全局环境变量 tidb_table_cache_lease 控制。你可根据实际业务能否承受此限制带来的延迟， 决定是否适合使用缓存表功能。例如，对于完全只读的场景，可以将 tidb_table_cache_lease 调大：
>
> ```
> set @@global.tidb_table_cache_lease = 10;
> ```
>
> 缓存表的写入延时高是受到实现的限制。存在多个TiDB 实例时，一个TiDB 实例并不知道其它的 TiDB 实例是否缓存了数据，如果该实例直接修改了表数据，而其它TiDB 实例依然读取旧的缓存数据，就会读到错误的结果。为了保证数据正确性，缓存表的实现使用了一套基于lease的复杂机制：读操作在缓存数据同时，还会对于缓存设置一个有效期，也就是lease。在lease过期之前，无法对数据执行修改操作。因为修改操作必须等待lease 过期，所以会出现写入延迟。

缓存表相关的元信息存储在 mysql.table_cache_meta 表中。这张表记录了所有缓存表的ID、当前的锁状态 lock_type，以及锁租约 lease 相关的信息。这张表仅供TiDB 内部使用，不建议用户修改该表，否则可能导致不可预期的错误。

```
SHOW CREATE TABLE mysql.table_cache_meta\G
*************************** 1. row ***************************
       Table: table_cache_meta
Create Table: CREATE TABLE `table_cache_meta` (
  `tid` bigint(11) NOT NULL DEFAULT '0',
  `lock_type` enum('NONE','READ','INTEND','WRITE') NOT NULL DEFAULT 'NONE',
  `lease` bigint(20) NOT NULL DEFAULT '0',
  `oldReadLease` bigint(20) NOT NULL DEFAULT '0',
  PRIMARY KEY (`tid`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

4.8.14.2.4 将缓存表恢复为普通表

> **注意：**
>
> 对缓存表执行 DDL 语句会失败。若要对缓存表执行 DDL 语句，需要先去掉缓存属性，将缓存表设回普通表后，才能对其执行 DDL 语句。

```
TRUNCATE TABLE users;
```

```
ERROR 8242 (HY000): 'Truncate Table' is unsupported on cache tables.
```

```
mysql> ALTER TABLE users ADD INDEX k_id(id);
```

```
ERROR 8242 (HY000): 'Alter Table' is unsupported on cache tables.
```

使用 `ALTER TABLE t NOCACHE` 语句可以将缓存表恢复成普通表：

```
ALTER TABLE users NOCACHE
```

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

4.8.14.3 缓存表大小限制

由于 TiDB 将整张缓存表的数据加载到TiDB 进程的内存中，并且执行修改操作后缓存会失效，需要重新加载，所以 TiDB 缓存表只适用于表比较小的场景。

目前 TiDB 对于每张缓存表的大小限制为 64 MB。如果表的数据超过了 64 MB，执行 `ALTER TABLE t CACHE` 会失败。

4.8.14.4 与其他 TiDB 功能的兼容性限制

以下是缓存表不支持的功能：

- 不支持对分区表执行 `ALTER TABLE t CACHE` 操作
- 不支持对临时表执行 `ALTER TABLE t CACHE` 操作
- 不支持对视图执行 `ALTER TABLE t CACHE` 操作
- 不支持 Stale Read 功能
- 不支持对缓存表直接做 DDL 操作，需要先通过 `ALTER TABLE t NOCACHE` 将缓存表改回普通表后再进行 DDL 操作。

以下是缓存表无法使用缓存的场景：

- 设置系统变量 `tidb_snapshot` 读取历史数据
- 执行修改操作期间，已有缓存会失效，直到数据被再次加载

4.8.14.5 TiDB 数据迁移工具兼容性

缓存表并不是标准的 MySQL 功能，而是TiDB 扩展。只有 TiDB 能识别 `ALTER TABLE ... CACHE` 语句。所有的 TiDB 数据迁移工具均不支持缓存表功能，包括 Backup & Restore (BR)、TiCDC、Dumpling 等组件，它们会将缓存表当作普通表处理。

这意味着，备份恢复一张缓存表时，它会变成一张普通表。如果下游集群是另一套TiDB 集群并且你希望继续使用缓存表功能，可以对下游集群中的表执行 `ALTER TABLE ... CACHE` 手动开启缓存表功能。

4.8.14.6 另请参阅

- ALTER TABLE
- System Variables

4.8.15 外键约束

从 v6.6.0 开始，TiDB 支持外键以及外键约束功能，外键允许跨表交叉引用相关数据，外键约束则可以保证相关数据的一致性。

> **注意：**
>
> 外键功能通常适用于为中小规模的数据提供完整性和一致性约束校验，但是在大数据量和分布式数据库系统下，使用外键可能会导致严重的性能问题，并对系统产生不可预知的影响。如果计划使用外键，请进行充分验证后谨慎使用。

外键是在子表中定义的，语法如下：

```
ForeignKeyDef
         ::= ( 'CONSTRAINT' Identifier )? 'FOREIGN' 'KEY'
             Identifier? '(' ColumnName ( ',' ColumnName )* ')'
             'REFERENCES' TableName '(' ColumnName ( ',' ColumnName )* ')'
             ( 'ON' 'DELETE' ReferenceOption )?
             ( 'ON' 'UPDATE' ReferenceOption )?

ReferenceOption
         ::= 'RESTRICT'
           | 'CASCADE'
           | 'SET' 'NULL'
           | 'SET' 'DEFAULT'
           | 'NO' 'ACTION'
```

4.8.15.1 命名

外键的命名遵循以下规则：

- 如果在 CONSTRAINT identifier 语句中指定了名称，则使用该名称。
- 如果 CONSTRAINT identifier 语句未指定名称，但在 FOREIGN KEY identifier 语句中指定了名称，则使用 FOREIGN KEY identifier 定义的名称。
- 如果 CONSTRAINT identifier 和 FOREIGN KEY identifier 语句都没有指定名称，则会自动生成一个名称，例如 fk_1、fk_2、fk_3 等。
- 外键名称必须在当前表中唯一，否则创建时会报错 ERROR 1826: Duplicate foreign key constraint <⟶ name 'fk'。

4.8.15.2 限制

创建外键时需要满足以下条件：

- 父表和子表都不能是临时表。
- 用户需要对父表有 REFERENCES 权限。
- 外键中的列和引用的父表中的列必须是相同的数据类型，并具有相同的大小、精度、长度、字符集(charset)和排序规则(collation)。
- 外键中的列不能引用自身。
- 外键中的列和引用的父表中的列必须有相同的索引，并且索引中的列顺序必须与外键的列顺序一样，这样才能在执行外键约束检查时使用索引来避免全表扫描。
  - 如果父表中没有对应的外键索引， 则会报错 ERROR 1822: Failed to add the foreign key <⟶ constraint. Missing index for constraint 'fk' in the referenced table 't'。
  - 如果子表中没有对应的外键索引，则会自动创建一个索引，索引名和外键名一样。
- 不支持在 BLOB 和 TEXT 类型的列上创建外键。
- 不支持在分区表上创建外键。
- 不支持在虚拟生成列(VIRTUAL GENERATED COLUMNS)上创建外键。

4.8.15.3 引用操作

当 UPDATE 或 DELETE 操作影响父表中的外键值时，其在子表中相匹配的外键值取决于外键定义中 ON UPDATE 和 ON DELETE 定义的引用操作，引用操作包括：

- CASCADE：当 UPDATE 或 DELETE 父表中的行数据时，自动级联更新或删除子表中的匹配行数据。级联操作会用深度优先方式执行。
- SET NULL：当 UPDATE 或 DELETE 父表中的行数据时，自动将子表中匹配的外键列数据设置为 NULL。
- RESTRICT：如果子表中存在外键匹配的行数据，则拒绝 UPDATE 或 DELETE 父表的操作。
- NO ACTION：行为和 RESTRICT 一样。
- SET DEFAULT：行为和 RESTRICT 一样。

如果父表中没有匹配的外键值，则拒绝 INSERT 或 UPDATE 子表的操作。

如果外键定义中没有指定 ON DELETE 或者 ON UPDATE，则默认的行为是 NO ACTION。

如果外键是定义在 STORED GENERATED COLUMN 上的，则不支持使用 CASCADE、SET NULL 和 SET DEFAULT 引用操作。

4.8.15.4 外键使用示例

下面的示例通过单列外键关联父表和子表：

```
CREATE TABLE parent
    ( id INT KEY
);

CREATE TABLE child (
    id INT,
    pid INT,
    INDEX idx_pid (pid),
    FOREIGN KEY (pid) REFERENCES parent(id) ON DELETE CASCADE
);
```

下面是一个更复杂的示例，其中 `product_order` 表有两个外键分别引用其他两个表。一个外键引用 `product` 表中的两列索引。另一个引用 `customer` 表中的单列索引：

```
CREATE TABLE product
    ( category INT NOT
    NULL, id INT NOT NULL,
    price DECIMAL(20,10),
    PRIMARY KEY(category, id)
);

CREATE TABLE customer
    ( id INT KEY
);

CREATE TABLE product_order (
    id INT NOT NULL AUTO_INCREMENT,
    product_category INT NOT NULL,
    product_id INT NOT NULL,
    customer_id INT NOT NULL,

    PRIMARY KEY(id),
    INDEX (product_category, product_id),
    INDEX (customer_id),

    FOREIGN KEY (product_category, product_id)
      REFERENCES product(category, id)
      ON UPDATE CASCADE ON DELETE RESTRICT,

    FOREIGN KEY (customer_id)
      REFERENCES customer(id)
);
```

4.8.15.5 新增外键约束

可以使用下面 ALTER TABLE 语句来新增一个外键约束：

```
ALTER TABLE table_name
    ADD [CONSTRAINT [identifier]] FOREIGN KEY
    [identifier] (col_name, ...)
    REFERENCES tbl_name (col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

外键可以是自引用的，即引用同一个表。使用 ALTER TABLE 向表添加外键约束时，请先在外键引用父表的列上创建索引。

4.8.15.6 删除外键约束

可以使用下面 ALTER TABLE 语句来删除一个外键约束：

```
ALTER TABLE table_name DROP FOREIGN KEY fk_identifier;
```

如果外键约束在创建时定义了名称，则可以引用该名称来删除外键约束。否则，只能引用自动生成的约束名称进行删除。你可以使用 SHOW CREATE TABLE 查看外键名称：

```
mysql> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
       Table: child
Create Table: CREATE TABLE `child` (
  `id` int(11) DEFAULT NULL,
  `pid` int(11) DEFAULT NULL,
  KEY `idx_pid` (`pid`),
  CONSTRAINT `fk_1` FOREIGN KEY (`pid`) REFERENCES `test`.`parent` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

4.8.15.7 外键约束检查

TiDB 支持是否开启外键约束检查，由系统变量foreign_key_checks 控制，其默认值是 ON，即开启外键约束检查，它有 GLOBAL 和 SESSION 两种作用域。在一般的操作中保持该变量开启可以保证外键引用关系的完整性。

关闭外键约束检查的作用如下：

- 当删除一个被外键引用的父表时，只有关闭外键约束检查时才能删除成功。
- 当给数据库导入数据时，创建表的顺序可能和外键依赖顺序不一样而导致创建表报错，只有关闭外键约束检查时才能创建表成功，另外，导入数据时关闭外键约束检查也能加快导数据的速度。
- 当给数据库导入数据时，先导入子表的数据会报错，只有关闭外键约束检查，才能确保顺利导入子表数据。
- 执行有关外键的 ALTER TABLE 操作时，关闭外键约束检查才能执行成功。

当关闭关键约束检查时，不会执行外键约束检查以及引用操作，但以下场景除外：

- 如果执行 ALTER TABLE 会导致外键定义不正确，则依然会执行报错。
- 删除外键所需的索引时，需要先删除外键，否则删除外键会执行报错。
- 创建外键时，如果不符合外键的条件或限制，则依然会执行报错。

4.8.15.8 锁

在 INSERT 或者 UPDATE 子表时，外键约束会检查父表中是否存在对应的外键值，并对父表中的该行数据上锁，避免该外键值被其他操作删除，导致破坏外键约束。这里的上锁行为等同于对父表中外键值所在行做 SELECT FOR UPDATE 操作。

因为TiDB 目前暂不支持 LOCK IN SHARE MODE，所以，在并发写入子表场景，如果引用的外键值大部分都一样，可能会有比较严重的锁冲突。建议在大批量写入子表数据时，关闭foreign_key_checks。

4.8.15.9 外键的定义和元信息

你可以使用SHOW CREATE TABLE 语句查看外键的定义：

```
mysql> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
       Table: child
Create Table: CREATE TABLE `child` (
  `id` int(11) DEFAULT NULL,
  `pid` int(11) DEFAULT NULL,
  KEY `idx_pid` (`pid`),
  CONSTRAINT `fk_1` FOREIGN KEY (`pid`) REFERENCES `test`.`parent` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

你可以使用以下任一系统表获取有关外键的信息：

- INFORMATION_SCHEMA.KEY_COLUMN_USAGE
- INFORMATION_SCHEMA.TABLE_CONSTRAINTS
- INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS

下面提供了查询示例：

从 INFORMATION_SCHEMA.KEY_COLUMN_USAGE 系统表中获取有关的外键信息：

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME F OM INFORMATION_SCHEMA.
    <-> KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_SCHEMA IS NOT NULL;
+--------------+---------------+------------------+-----------------------------+
| TABLE_SCHEMA | TABLE_NAME    | COLUMN_NAME      | CONSTRAINT_NAME |
+--------------+---------------+------------------+-----------------------------+
| test         | child         | pid              | fk_1            |
| test         | product_order | product_category | fk_1            |
| test         | product_order | product_id       | fk_1            |
| test         | product_order | customer_id      | fk_2            |
+--------------+---------------+------------------+-----------------------------+
```

从 INFORMATION_SCHEMA.TABLE_CONSTRAINTS 系统表中获取有关的外键信息：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS WHERE CONSTRAINT_TYPE='FOREIGN KEY'\G
***************************[ 1. row ]***************************
CONSTRAINT_CATALOG | def
CONSTRAINT_SCHEMA  | test
CONSTRAINT_NAME    | fk_1
TABLE_SCHEMA       | test
TABLE_NAME         | child
CONSTRAINT_TYPE    | FOREIGN KEY
```

从 INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS 系统表中获取有关的外键信息：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS\G
***************************[ 1. row ]***************************
CONSTRAINT_CATALOG         | def
CONSTRAINT_SCHEMA          | test
CONSTRAINT_NAME            | fk_1
UNIQUE_CONSTRAINT_CATALOG  | def
UNIQUE_CONSTRAINT_SCHEMA   | test
UNIQUE_CONSTRAINT_NAME     | PRIMARY
MATCH_OPTION               | NONE
UPDATE_RULE                | NO ACTION
DELETE_RULE                | CASCADE
TABLE_NAME                 | child
REFERENCED_TABLE_NAME      | parent
```

4.8.15.10 查看带有外键的执行计划

你可以使用 EXPLAIN 语句查看执行计划。Foreign_Key_Check 算子是执行 DML 语句时，执行外键约束检查的算子。

```
mysql> explain insert into child values (1,1);
+------------------------+----------+------+---------------+--------------------------------+
| id                     | estRows | task | access object | operator info                  |
+------------------------+----------+------+---------------+--------------------------------+
| Insert_1               | N/A      | root |               | N/A                            |
| └─Foreign_Key_Check_3 | 0.00     | root | table:parent  | foreign_key:fk_1, check_exist |
+------------------------+----------+------+---------------+--------------------------------+
```

你可以使用 EXPLAIN ANALYZE 语句查看外键引用行为的执行。Foreign_Key_Cascade 算子是执行 DML 语句时，执行外键引用行为的算子。

```
mysql> explain analyze delete from parent where id = 1;
+--
    <-> -------------------------------+---------+---------+-----------+-------------------------------+------
    <->
```

```
| id                              | estRows | actRows | task      | access object
    <-->                   | execution info
    <-->
    <--> | operator info                                | memory    | disk |
+--
    <--> --------------------------------+---------+---------+-----------+------------------------------+------
    <-->
| Delete_2                        | N/A     | 0       | root      |
    <-->                               | time:117.3µs, loops:1
    <-->
    <--> | N/A                                           | 380 Bytes | N/A  |
| ├─Point_Get_1                   | 1.00    | 1       | root      | table:parent
    <-->                    | time:63.6µs, loops:2, Get:{num_rpc:1, total_time:29.9µs}
    <-->
    <--> | handle:1                                      | N/A       | N/A  |
| └─Foreign_Key_Cascade_3         | 0.00    | 0       | root      | table:child, index:idx_pid
    <-->     | total:1.28ms, foreign_keys:1
    <-->
    <--> | foreign_key:fk_1, on_delete:CASCADE           | N/A       | N/A  |
|   └─Delete_7                    | N/A     | 0       | root      |
    <-->                               | time:904.8µs, loops:1
    <-->
    <--> | N/A                                           | 1.11 KB   | N/A  |
|     └─IndexLookUp_11            | 10.00   | 1       | root      |
    <-->                               | time:869.5µs, loops:2, index_task: {total_time: 371.1µs
    <--> , fetch_handle: 357.3µs, build: 1.25µs, wait: 12.5µs}, table_task: {total_time: 382.6µs,
    <--> num: 1, concurrency: 5}                      |
    <-->                                            | 9.13 KB   | N/A  |
|       ├─IndexRangeScan_9(Build) | 10.00   | 1       | cop[tikv] | table:child, index:idx_pid(
    <--> pid) | time:351.2µs, loops:3, cop_task: {num: 1, max: 282.3µs, proc_keys: 0, rpc_num: 1,
    <--> rpc_time: 263µs, copr_cache_hit_ratio: 0.00, distsql_concurrency: 15}, tikv_task:{time
    <--> :220.2µs, loops:0}    | range:[1,1], keep order:false, stats:pseudo | N/A       | N/A  |
|       └─TableRowIDScan_10(Probe) | 10.00   | 1       | cop[tikv] | table:child
    <-->                    | time:223.9µs, loops:2, cop_task: {num: 1, max: 168.8µs, proc_keys:
    <--> 0, rpc_num: 1, rpc_time: 154.5µs, copr_cache_hit_ratio: 0.00, distsql_concurrency: 15},
    <--> tikv_task:{time:145.6µs, loops:0} | keep order:false, stats:pseudo              | N/A
    <-->      | N/A  |
+--
    <--> --------------------------------+---------+---------+-----------+------------------------------+------
    <-->
```

4.8.15.11 兼容性

4.8.15.11.1 TiDB 版本间兼容性

TiDB 在 v6.6.0 之前已经支持创建外键的语法，但创建的外键并不生效。如果将之前创建的TiDB 集群升级到 v6.6.0 及之后的版本，之前创建的外键依然是不生效的，可以先删除不生效的外键后再创建外键使外键约束生效。只有在 v6.6.0 及之后版本中新创建的外键才生效。你可以使用 SHOW CREATE TABLE 语句查看外键是否生效，不生效的外键会有一条 /* FOREIGN KEY INVALID */ 注释。

```
mysql> SHOW CREATE TABLE child\G
***************************[ 1. row ]***************************
Table        | child
Create Table | CREATE TABLE `child` (
  `id` int(11) DEFAULT NULL,
  `pid` int(11) DEFAULT NULL,
  KEY `idx_pid` (`pid`),
  CONSTRAINT `fk_1` FOREIGN KEY (`pid`) REFERENCES `test`.`parent` (`id`) ON DELETE CASCADE /*
      <⟶ FOREIGN KEY INVALID */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

4.8.15.11.2 与 TiDB 工具的兼容性

- TiDB Binlog 不支持外键功能。
- DM 不兼容外键功能。DM v6.6.0 在同步数据到下游TiDB 时，会显式关闭下游TiDB 的foreign_key_checks，所以由外键产生的级联操作不会从上游同步到下游，进而导致上下游数据不一致。这与 v6.6.0 之前版本 DM 的行为一致。
- TiCDC v6.6.0 兼容外键功能。旧版本的TiCDC 在同步带外键的表时，可能会报错，建议使用 v6.6.0 之前版本 TiCDC 时先关闭下游TiDB 集群的 foreign_key_checks。
- BR v6.6.0 兼容外键功能。之前版本的 BR 在恢复带外键的表到 v6.6.0 及之后版本的集群时，可能会报错，建议先关闭下游TiDB 集群的 foreign_key_checks 后再恢复集群。
- TiDB Lightning 导入数据到TiDB 前，建议先关闭TiDB 集群的 foreign_key_checks。
- Dumpling 兼容外键功能。
- sync-diff-inspector 在对比上下游数据时，如果上下游数据库的版本不一样，且下游TiDB 中存在不生效的外键，则 sync-diff-inspector 可能会报上下游表结构不一致的错误。因为TiDB v6.6.0 会对表结构中不生效的外键添加一条 /* FOREIGN KEY INVALID */ 注释。

4.8.15.11.3 与 MySQL 的兼容性

创建外键未指定名称时，TiDB 自动生成的外键名称和 MySQL 不一样。例如TiDB 生成的外键名称为 fk_1、fk_2、fk_3 等，MySQL 生成的外键名称为 table_name_ibfk_1、table_name_ibfk_2、table_name_ibfk_3 等。

MySQL 和 TiDB 均能解析但会忽略以内联 REFERENCES 的方式定义的外键。只有当 REFERENCES 作为 FOREIGN KEY 定义的一部分时，才会进行检查和执行。下面的示例在定义外键约束时只使用了 REFERENCES：

```
CREATE TABLE parent
    ( id INT KEY
);

CREATE TABLE child (
    id INT,
```

```
    pid INT REFERENCES parent(id)
);

SHOW CREATE TABLE child;
```

输出结果显示 child 表不包含任何外键：

```
+-------+-----------------------------------------------------------------+
| Table | Create Table                                                    |
+-------+-----------------------------------------------------------------+
| child | CREATE TABLE `child` (                                          |
|       |   `id` int(11) DEFAULT NULL,                                    |
|       |   `pid` int(11) DEFAULT NULL                                    |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-------+-----------------------------------------------------------------+
```

4.8.16 字符集和排序

4.8.16.1 字符集和排序规则

本文介绍了TiDB 中支持的字符集和排序规则。

4.8.16.1.1 字符集和排序规则的概念

字符集(character set) 是符号与编码的集合。TiDB 中的默认字符集是utf8mb4，与 MySQL 8.0 及更高版本中的默认字符集匹配。

排序规则(collation) 是在字符集中比较字符以及字符排序顺序的规则。例如，在二进制排序规则中，比较A 和 a 的结果是不一样的：

```
SET NAMES utf8mb4 COLLATE utf8mb4_bin;
SELECT 'A' = 'a';
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
SELECT 'A' = 'a';
```

```
SELECT 'A' = 'a';
```

```
+------------+
| 'A' = 'a' |
+------------+
|          0 |
+------------+
1 row in set (0.00 sec)
```

```
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
```

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

```
SELECT 'A' = 'a';
```

```
+-----------+
| 'A' = 'a' |
+-----------+
|         1 |
+-----------+
1 row in set (0.00 sec)
```

TiDB 默认使用二进制排序规则。这一点与 MySQL 不同，MySQL 默认使用不区分大小写的排序规则。

4.8.16.1.2 支持的字符集和排序规则

目前TiDB 支持以下字符集：

```
SHOW CHARACTER SET;
```

```
+---------+---------------------------------+--------------------+--------+
| Charset | Description                     | Default collation  | Maxlen |
+---------+---------------------------------+--------------------+--------+
| ascii   | US ASCII                        | ascii_bin          |      1 |
| binary  | binary                          | binary             |      1 |
| gbk     | Chinese Internal Code Specification | gbk_bin        |      2 |
| latin1  | Latin1                          | latin1_bin         |      1 |
| utf8    | UTF-8 Unicode                   | utf8_bin           |      3 |
| utf8mb4 | UTF-8 Unicode                   | utf8mb4_bin        |      4 |
+---------+---------------------------------+--------------------+--------+
6 rows in set (0.00 sec)
```

TiDB 支持以下排序规则：

```
SHOW COLLATION;
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| ascii_bin          | ascii   |   65 | Yes     | Yes      |       1 |
| binary             | binary  |   63 | Yes     | Yes      |       1 |
| gbk_bin            | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci     | gbk     |   28 | Yes     | Yes      |       1 |
| latin1_bin         | latin1  |   47 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |   83 | Yes     | Yes      |       1 |
| utf8_general_ci    | utf8    |   33 |         | Yes      |       1 |
```

```
| utf8_unicode_ci     | utf8    |  192 |         | Yes      |       1 |
| utf8mb4_bin         | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci  | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci  | utf8mb4 |  224 |         | Yes      |       1 |
+---------------------+---------+------+---------+----------+---------+
11 rows in set (0.00 sec)
```

> **警告：**
>
> TiDB 会错误地将 `latin1` 视为 `utf8` 的子集。当用户存储不同于 `latin1` 和 `utf8` 编码的字符时，可能会导致意外情况出现。因此强烈建议使用 `utf8mb4` 字符集。详情参阅TiDB #18955。

> **注意：**
>
> TiDB 中的默认排序规则（后缀为 `_bin` 的二进制排序规则）与 MySQL 中的默认排序规则不同，后者通常是一般排序规则，后缀为 `_general_ci`。当用户指定了显式字符集，但依赖于待选的隐式默认排序规则时，这个差异可能导致兼容性问题。

利用以下的语句可以查看字符集对应的排序规则（以下是新的排序规则框架）下的结果：

```
SHOW COLLATION WHERE Charset = 'utf8mb4';
```

```
+---------------------+---------+------+---------+----------+---------+
| Collation           | Charset | Id   | Default | Compiled | Sortlen |
+---------------------+---------+------+---------+----------+---------+
| utf8mb4_bin         | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci  | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci  | utf8mb4 |  224 |         | Yes      |       1 |
+---------------------+---------+------+---------+----------+---------+
3 rows in set (0.00 sec)
```

TiDB 对 GBK 字符集的支持详情见GBK。

4.8.16.1.3 TiDB 中的 utf8 和 utf8mb4

MySQL 限制字符集 `utf8` 为最多 3 个字节。这足以存储在基本多语言平面(BMP) 中的字符，但不足以存储表情符号(emoji) 等字符。因此，建议改用字符集`utf8mb4`。

默认情况下，TiDB 同样限制字符集 `utf8` 为最多 3 个字节，以确保 TiDB 中创建的数据可以在 MySQL 中顺利恢复。你可以禁用此功能，方法是将系统变量`tidb_check_mb4_value_in_utf8` 的值更改为 `OFF`。

以下示例演示了在表中插入 4 字节的表情符号字符（emoji 字符）时的默认行为。`utf8` 字符集下 `INSERT` 语句不能执行，`utf8mb4` 字符集下可以执行 `INSERT` 语句：

```
CREATE TABLE utf8_test (
    ->  c char(1) NOT NULL
    -> ) CHARACTER SET utf8;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
CREATE TABLE utf8m4_test (
    ->  c char(1) NOT NULL
    -> ) CHARACTER SET utf8mb4;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
INSERT INTO utf8_test VALUES ('D');
```

```
ERROR 1366 (HY000): incorrect utf8 value f09f9889(D) for column c
```

```
INSERT INTO utf8m4_test VALUES ('D');
```

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

```
SELECT char_length(c), length(c), c FROM utf8_test;
```

```
Empty set (0.01 sec)
```

```
SELECT char_length(c), length(c), c FROM utf8m4_test;
```

```
+----------------+-----------+----------+
| char_length(c) | length(c) | c        |
+----------------+-----------+----------+
|              1 |         4 | D        |
+----------------+-----------+----------+
1 row in set (0.00 sec)
```

4.8.16.1.4 不同范围的字符集和排序规则

字符集和排序规则可以在设置在不同的层次。

数据库的字符集和排序规则

每个数据库都有相应的字符集和排序规则。数据库的字符集和排序规则可以通过以下语句来设置：

```
CREATE DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

```
ALTER DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

在这里 DATABASE 可以跟 SCHEMA 互换使用。

不同的数据库之间可以使用不一样的字符集和排序规则。

通过系统变量 character_set_database 和 collation_database 可以查看到当前数据库的字符集以及排序规则：

```
CREATE SCHEMA test1 CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test1;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| utf8mb4                  | utf8mb4_general_ci   |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

```
CREATE SCHEMA test2 CHARACTER SET latin1 COLLATE latin1_bin;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test2;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| latin1                   | latin1_bin           |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

在INFORMATION_SCHEMA 中也可以查看到这两个值：

```
SELECT DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME
FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'db_name';
```

表的字符集和排序规则

表的字符集和排序规则可以通过以下语句来设置：

```
CREATE TABLE tbl_name (column_list)
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]]

ALTER TABLE tbl_name
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]
```

例如：

```
CREATE TABLE t1(a int) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.08 sec)
```

如果表的字符集和排序规则没有设置，那么数据库的字符集和排序规则就作为其默认值。

列的字符集和排序规则

列的字符集和排序规则的语法如下：

```
col_name {CHAR | VARCHAR | TEXT} (col_length)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]

col_name {ENUM | SET} (val_list)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]
```

如果列的字符集和排序规则没有设置，那么表的字符集和排序规则就作为其默认值。

字符串的字符集和排序规则

每一个字符串都对应一个字符集和一个排序规则，在使用字符串时指此选项可选，如下：

```
[_charset_name]'string' [COLLATE collation_name]
```

示例如下：

```
SELECT 'string';
SELECT _utf8mb4'string';
SELECT _utf8mb4'string' COLLATE utf8mb4_general_ci;
```

规则如下：

- 规则1：如果指定了CHARACTER SET charset_name和COLLATE collation_name，则直接使用charset_name字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则3：如果CHARACTER SET charset_name和COLLATE collation_name都未指定，则使用character_set_connection <↪ 和 collation_connection 系统变量给出的字符集和排序规则。

客户端连接的字符集和排序规则

- 服务器的字符集和排序规则可以通过系统变量 character_set_server 和 collation_server 获取。
- 数据库的字符集和排序规则可以通过环境变量 character_set_database 和 collation_database 获取。

对于每一个客户端的连接， 也有相应的变量表示字符集和排序规则：character_set_connection 和 collation_connection。

character_set_client 代表客户端的字符集。在返回结果前，服务端会把结果根据 character_set_results 转换成对应的字符集，包括结果的元信息等。

可以用以下的语句来影响这些跟客户端相关的字符集变量：

- SET NAMES 'charset_name' [COLLATE 'collation_name']

  SET NAMES 用来设定客户端会在之后的请求中使用的字符集。SET NAMES utf8mb4 表示客户端会在接下来的请求中，都使用 utf8mb4 字符集。服务端也会在之后返回结果的时候使用 utf8mb4 字符集。SET NAMES 'charset_name' 语句其实等于下面语句的组合：

  ```
  SET character_set_client = charset_name;
  SET character_set_results = charset_name;
  SET character_set_connection = charset_name;
  ```

  COLLATE 是可选的，如果没有提供，将会用charset_name 对应的默认排序规则设置collation_connection <↪。

- SET CHARACTER SET 'charset_name'

  跟 SET NAMES 类似，等价于下面语句的组合：

  ```
  SET character_set_client = charset_name;
  SET character_set_results = charset_name;
  SET charset_connection = @@charset_database;
  SET collation_connection = @@collation_database;
  ```

4.8.16.1.5 服务器、数据库、表、列、字符串的字符集和排序规则的优先级

优先级从高到低排列顺序为：

字符串> 列 > 表 > 数据库> 服务器

4.8.16.1.6 字符集和排序规则的通用选择规则

- 规则1：如果指定了CHARACTER SET charset_name和COLLATE collation_name，则直接使用charset_name字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用更高优先级的字符集和排序规则。

4.8.16.1.7 字符合法性检查

当指定的字符集为utf8 或utf8mb4 时，TiDB 仅支持合法的utf8 字符。对于不合法的字符，会报错：incorrect <⟶ utf8 value。该字符合法性检查与 MySQL 8.0 兼容，与 MySQL 5.7 及以下版本不兼容。

如果不希望报错，可以通过 set @@tidb_skip_utf8_check=1; 跳过字符检查。

> 注意：
>
> 跳过字符检查可能会使TiDB 检测不到应用写入的非法 UTF-8 字符，进一步导致执行 ANALYZE 时解码错误，以及引入其他未知的编码问题。如果应用不能保证写入字符串的合法性，不建议跳过该检查。

4.8.16.1.8 排序规则支持

排序规则的语法支持和语义支持受到配置项new_collations_enabled_on_first_bootstrap 的影响。这里语法支持和语义支持有所区别。语法支持是指TiDB 能够解析和设置排序规则；而语义支持是指TiDB 能够在比较字符串时正确地使用排序规则。

在 4.0 版本之前，TiDB 只提供了旧的排序规则框架，能够在语法上支持的绝大部分 MySQL 排序规则，但语义上所有的排序规则都当成二进制排序规则。

4.0 版本中，TiDB 增加了新的排序规则框架用于在语义上支持不同的排序规则，保证字符串比较时严格遵循对应的排序规则，详情请见下文。

旧框架下的排序规则支持

在 4.0 版本之前，TiDB 中可以指定大部分 MySQL 中的排序规则，并把这些排序规则按照默认排序规则处理，即以编码字节序为字符定序。和 MySQL 不同的是，TiDB 不会处理字符末尾的空格，因此会造成以下的行为区别：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
```

```
Query OK, 0 rows affected
```

```
INSERT INTO t VALUES ('A');
```

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

```
INSERT INTO t VALUES ('a');
```

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

以上语句，在TiDB 会执行成功，而在 MySQL 中，由于 utf8mb4_general_ci 大小写不敏感，报错 Duplicate <⟶ entry 'a'。

```
INSERT INTO t VALUES ('a ');
```

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

以上语句，在TiDB 会执行成功，而在 MySQL 中，由于补齐空格比较，报错 Duplicate entry 'a '。

新框架下的排序规则支持

TiDB 4.0 新增了完整的排序规则支持框架，从语义上支持了排序规则，并新增了配置开关new_collations_enabled_on_first_boo <⟶ ， 在集群初次初始化时决定是否启用新排序规则框架。 如需启用新排序规则框架， 可将 new_collations_enabled_on_first_bootstrap 的值设为true，详情参见new_collations_enabled_on_first_bootstrap <⟶。要在该配置开关打开之后初始化集群，可以通过 mysql.tidb 表中的 new_collation_enabled 变量确认是否启用了新排序规则框架：

```
SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME='new_collation_enabled';
```

```
+----------------------------+
| VARIABLE_VALUE |
+----------------------------+
| True           |
+----------------------------+
1 row in set (0.00 sec)
```

在新的排序规则框架下，TiDB 能够支持 utf8_general_ci、utf8mb4_general_ci、utf8_unicode_ci、utf8mb4_unicode_ci、gbk_chinese_ci 和 gbk_bin 这几种排序规则，与 MySQL 兼容。

使用 utf8_general_ci、utf8mb4_general_ci、utf8_unicode_ci 、utf8mb4_unicode_ci 和 gbk_chinese_ci 中任一种时，字符串之间的比较是大小写不敏感(case-insensitive) 和口音不敏感(accent-insensitive) 的。同时，TiDB 还修正了排序规则的 PADDING 行为：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
```

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

```
INSERT INTO t VALUES ('A');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
INSERT INTO t VALUES ('a');
```

```
ERROR 1062 (23000): Duplicate entry 'a' for key 't.PRIMARY'
```

TiDB 兼容了 MySQL 的 case insensitive collation。

```
INSERT INTO t VALUES ('a ');
```

```
ERROR 1062 (23000): Duplicate entry 'a ' for key 't.PRIMARY'
```

TiDB 修正了 PADDING 行为，与 MySQL 兼容。

> **注意：**
>
> TiDB 中 padding 的实现方式与 MySQL 的不同。在 MySQL 中，padding 是通过补齐空格实现的。而在 TiDB 中 padding 是通过裁剪掉末尾的空格来实现的。两种做法在绝大多数情况下是一致的，唯一的例外是字符串尾部包含小于空格(0x20) 的字符时，例如 `'a' < 'a\t'` 在 TiDB 中的结果为 1，而在 MySQL 中，其等价于 `'a ' < 'a\t'`，结果为 0。

4.8.16.1.9 表达式中排序规则的 Coercibility 值

如果一个表达式涉及多个不同排序规则的子表达式时，需要对计算时用的排序规则进行推断，规则如下：

- 显式 COLLATE 子句的coercibility 值为 0。
- 如果两个字符串的排序规则不兼容，这两个字符串 concat 结果的coercibility 值为 1。
- 列或者 CAST()、CONVERT() 和 BINARY() 的排序规则的coercibility 值为 2。
- 系统常量（USER() 或者 VERSION() 返回的字符串）的coercibility 值为 3。
- 常量的coercibility 值为 4。
- 数字或者中间变量的coercibility 值为 5。
- NULL 或者由 NULL 派生出的表达式的coercibility 值为 6。

在推断排序规则时，TiDB 优先使用coercibility 值较低的表达式的排序规则。如果coercibility 值相同，则按以下优先级确定排序规则：

binary > utf8mb4_bin > (utf8mb4_general_ci = utf8mb4_unicode_ci) > utf8_bin > (utf8_general_ci = utf8_unicode_ci) > latin1_bin > ascii_bin

以下情况TiDB 无法推断排序规则并报错：

- 如果两个子表达式的排序规则不相同，而且表达式的coercibility 值都为 0。
- 如果两个子表达式的排序规则不兼容，而且表达式的返回类型为 String 类。

4.8.16.1.10 COLLATE 子句

TiDB 支持使用 COLLATE 子句来指定一个表达式的排序规则，该表达式的coercibility 值为 0，具有最高的优先级。示例如下：

```
SELECT 'a' = _utf8mb4 'A' collate utf8mb4_general_ci;
```

```
+-----------------------------------------------+
| 'a' = _utf8mb4 'A' collate utf8mb4_general_ci |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

更多细节，参考 Connection Character Sets and Collations。

4.8.16.2 GBK

TiDB 从 v5.4.0 开始支持GBK 字符集。本文档介绍TiDB 对 GBK 字符集的支持和兼容情况。

```
SHOW CHARACTER SET WHERE CHARSET = 'gbk';
+---------+-----------------------------------+--------------------+--------+
| Charset | Description                       | Default collation | Maxlen |
+---------+-----------------------------------+--------------------+--------+
| gbk     | Chinese Internal Code Specification | gbk_bin            |      2 |
+---------+-----------------------------------+--------------------+--------+
1 row in set (0.00 sec)

SHOW COLLATION WHERE CHARSET = 'gbk';
+----------------+---------+------+---------+----------+---------+
| Collation      | Charset | Id   | Default | Compiled | Sortlen |
+----------------+---------+------+---------+----------+---------+
| gbk_bin        | gbk     |   87 |         | Yes      |       1 |
+----------------+---------+------+---------+----------+---------+
1 rows in set (0.00 sec)
```

4.8.16.2.1 与 MySQL 的兼容性

本节介绍TiDB 中 GBK 字符集与 MySQL 的兼容情况。

排序规则兼容性

MySQL 的字符集默认排序规则是 gbk_chinese_ci。与 MySQL 不同，TiDB GBK 字符集的默认排序规则为 gbk_bin。另外，TiDB 支持的 gbk_bin 与 MySQL 支持的 gbk_bin 排序规则也不一致，TiDB 是将GBK 转换成 UTF8MB4 然后做二进制排序。

如果要使 TiDB 兼容 MySQL 的 GBK 字符集排序规则， 你需要在初次初始化 TiDB 集群时设置 TiDB 配置项new_collations_enabled_on_first_bootstrap 为 true 来开启新的排序规则框架。

开启新的排序规则框架后，如果查看GBK 字符集对应的排序规则，你可以看到TiDB GBK 默认排序规则已经切换为 gbk_chinese_ci。

```
SHOW CHARACTER SET WHERE CHARSET = 'gbk';
+---------+-----------------------------------+--------------------+--------+
| Charset | Description                       | Default collation | Maxlen |
+---------+-----------------------------------+--------------------+--------+
| gbk     | Chinese Internal Code Specification | gbk_chinese_ci     |      2 |
+---------+-----------------------------------+--------------------+--------+
1 row in set (0.00 sec)

SHOW COLLATION WHERE CHARSET = 'gbk';
+----------------+---------+------+---------+----------+--------+
| Collation      | Charset | Id   | Default | Compiled | Sortlen |
+----------------+---------+------+---------+----------+--------+
| gbk_bin        | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci | gbk     |   28 | Yes     | Yes      |       1 |
+----------------+---------+------+---------+----------+--------+
2 rows in set (0.00 sec)
```

非法字符兼容性

- 在系统变量character_set_client 和character_set_connection 不同时设置为 gbk 的情况下，TiDB 处理非法字符的方式与 MySQL 一致。
- 在 character_set_client 和 character_set_connection 同时为 gbk 的情况下，TiDB 处理非法字符的方式与 MySQL 有所区别。
  - MySQL 处理非法GBK 字符集时，对读和写操作的处理方式不同。
  - TiDB 处理非法GBK 字符集时，对读和写操作的处理方式相同。TiDB 在严格模式下读写非法GBK 字符都会报错，在非严格模式下，读写非法GBK 字符都会用? 替换。

例如， 当 SET NAMES gbk 时， 如果分别在 MySQL 和 TiDB 上通过 CREATE TABLE gbk_table(a VARCHAR(32) <⟶ CHARACTER SET gbk) 语句建表，然后按照下表中的 SQL 语句进行操作，就能看到具体的区别。

数据库	如果设置的 SQL 模式包含 STRICT_ALL_TABLES 或 STRICT_TRANS_TABLES	如果设置的 SQL 模式不包含 STRICT_ALL_TABLES 和 STRICT_TRANS_TABLES
MySQL	SELECT HEX('一a'); e4b88061INSERT INTO <⟶ gbk_table values('一a'); Incorrect Error	SELECT HEX('一a'); e4b88061INSERT INTO gbk_table VALUES('一a'); <⟶ SELECT HEX(a)FROM gbk_table; e4b8
TiDB	SELECT HEX('一a'); Incorrect ErrorINSERT <⟶ INTO gbk_table VALUES('一a'); Incorrect Error	SELECT HEX('一a'); e4b83fINSERT INTO gbk_table VALUES('一a'); <⟶ SELECT HEX(a)FROM gbk_table; e4b83f

说明：该表中 SELECT HEX('一a'); 在 utf8mb4 字节集下的结果为 e4b88061。

其它

- 目前TiDB 不支持通过 ALTER TABLE 语句将其它字符集类型改成 gbk 或者从 gbk 转成其它字符集类型。
- TiDB 不支持使用 _gbk，比如：

```
sql CREATE TABLE t(a CHAR(10)CHARSET BINARY); Query OK, 0 rows affected (0.00 sec)INSERT INTO t
<-> VALUES (_gbk'啊'); ERROR 1115 (42000): Unsupported character introducer: 'gbk'
```

- 对于 ENUM 和 SET 类型中的二进制字符，TiDB 目前都会将其作为 utf8mb4 字符集处理。

4.8.16.2.2 组件兼容性

- TiFlash 目前不支持GBK 字符集。
- TiDB Data Migration (DM) 在 v5.4.0 之前不支持将 charset=GBK 的表迁移到TiDB。
- TiDB Lightning 在 v5.4.0 之前不支持导入 charset=GBK 的表。
- TiCDC 在 v6.1.0 之前不支持同步 charset=GBK 的表。另外，任何版本的 TiCDC 都不支持同步 charset=GBK 的表到 6.1.0 之前的TiDB 集群。
- TiDB Backup & Restore（BR）在 v5.4.0 之前不支持恢复 charset=GBK 的表。另外，任何版本的 BR 都不支持恢复 charset=GBK 的表到 5.4.0 之前的TiDB 集群。

4.8.17 Placement Rules in SQL

Placement Rules in SQL 特性用于通过 SQL 接口配置数据在TiKV 集群中的放置位置。通过该功能，用户可以将表和分区指定部署至不同的地域、机房、机柜、主机。适用场景包括低成本优化数据高可用策略、保证本地的数据副本可用于本地 Stale Read 读取、遵守数据本地要求等。

> 注意：
>
> Placement Rules in SQL 底层的实现依赖 PD 提供的放置规则(placement rules) 功能，参考Placement Rules 使用文档。在 Placement Rules in SQL 语境下，放置规则既可以代指绑定对象的放置策略(placement policy)，也可以代指TiDB 发给 PD 的放置规则。

该功能可以实现以下业务场景：

- 合并多个不同业务的数据库，大幅减少数据库常规运维管理的成本
- 增加重要数据的副本数，提高业务可用性和数据可靠性
- 将最新数据存入 NVMe，历史数据存入 SSD，降低归档数据存储成本
- 把热点数据的leader 放到高性能的TiKV 实例上
- 将冷数据分离到不同的存储中以提高可用性
- 支持物理隔离不同用户之间的计算资源，满足实例内部不同用户的隔离需求，以及不同混合负载 CPU、I/O、内存等资源隔离的需求

4.8.17.1 指定放置规则

指定放置规则，首先需要通过CREATE PLACEMENT POLICY 语句创建放置策略(placement policy)。

```
CREATE PLACEMENT POLICY myplacementpolicy PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1
    <-> ";
```

然后可以使用 CREATE TABLE 或者 ALTER TABLE 将规则绑定至表或分区表，这样就在表或分区上指定了放置规则：

```
CREATE TABLE t1 (a INT) PLACEMENT POLICY=myplacementpolicy;
CREATE TABLE t2 (a INT);
ALTER TABLE t2 PLACEMENT POLICY=myplacementpolicy;
```

PLACEMENT POLICY 为全局作用域，不与任何数据库表结构相关联。因此，通过 CREATE TABLE 指定放置规则时，无需任何额外的权限。

要修改放置策略，你可以使用ALTER PLACEMENT POLICY 语句。修改将传播到所有绑定此放置策略的对象。

```
ALTER PLACEMENT POLICY myplacementpolicy FOLLOWERS=5;
```

要删除没有绑定任何分区或表的放置策略，你可以使用DROP PLACEMENT POLICY：

```
DROP PLACEMENT POLICY myplacementpolicy;
```

4.8.17.2 查看放置规则

如果一张表绑定了放置规则，你可以用SHOW CREATE TABLE 来查看。还可以用SHOW CREATE PLACEMENT POLICY 来查看已经创建的放置策略。

```
tidb> 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 /*T![placement] PLACEMENT POLICY=`
    <-> myplacementpolicy` */
1 row in set (0.00 sec)
tidb> SHOW CREATE PLACEMENT POLICY myplacementpolicy\G
*************************** 1. row ***************************
       Policy: myplacementpolicy
Create Policy: CREATE PLACEMENT POLICY myplacementpolicy PRIMARY_REGION="us-east-1" REGIONS="us-
    <-> east-1,us-west-1"
1 row in set (0.00 sec)
```

你也可以用INFORMATION_SCHEMA.PLACEMENT_POLICIES 系统表查看所有放置策略的定义。

```
tidb> select * from information_schema.placement_policies\G
***************************[ 1. row ]***************************
```

```
POLICY_ID             | 1
CATALOG_NAME          | def
POLICY_NAME           | p1
PRIMARY_REGION        | us-east-1
REGIONS               | us-east-1,us-west-1
CONSTRAINTS           |
LEADER_CONSTRAINTS    |
FOLLOWER_CONSTRAINTS  |
LEARNER_CONSTRAINTS   |
SCHEDULE              |
FOLLOWERS             | 4
LEARNERS              | 0
1 row in set
```

`information_schema.tables` 表和 `information_schema.partitions` 表也有一列 `tidb_placement_policy_name`，用于展示所有绑定了放置规则的对象：

```
SELECT * FROM information_schema.tables WHERE tidb_placement_policy_name IS NOT NULL;
SELECT * FROM information_schema.partitions WHERE tidb_placement_policy_name IS NOT NULL;
```

所有绑定规则的对象都是异步调度的。可以用`SHOW PLACEMENT` 来查看放置规则的调度进度。

4.8.17.3 放置选项参考

注意：

- 放置选项依赖于正确地指定在每个TiKV 节点配置中的标签(label)。例如，`PRIMARY_REGION` <⟶> 选项依赖TiKV 中的 `region` 标签。若要查看当前TiKV 集群中所有可用的标签，可执行`SHOW PLACEMENT LABELS` 语句。

```
mysql> show placement labels;
+--------+----------------------+
| Key    | Values               |
+--------+----------------------+
| disk   | ["ssd"]              |
| region | ["us-east-1"]        |
| zone   | ["us-east-1a"]       |
+--------+----------------------+
3 rows in set (0.00 sec)
```

- 使用 `CREATE PLACEMENT POLICY` 创建放置规则时，TiDB 不会检查标签是否存在，而是在绑定表的时候进行检查。

选项名	描述
PRIMARY_REGION	Raft leader 被放置在有 region 标签的节点上，且这些 region 标签匹配本选项的值。
REGIONS	Raft followers 被放置在有 region 标签的节点上，且这些 region 标签匹配本选项的值。
SCHEDULE	用于调度 follower 放置位置的策略。可选值为 EVEN（默认值）或 MAJORITY_IN_PRIMARY。
FOLLOWERS	Follower 的数量。例如 FOLLOWERS=2 表示数据有 3 个副本（2 个 follower 和 1 个 leader）。

除以上配置选项外，你还可以使用高级配置，详细介绍见高级放置选项。

选项名	描述
CONSTRAINTS	适用于所有角色(role) 的约束列表。例如，CONSTRAINTS="[+disk=ssd]"。
LEADER_CONSTRAINTS	仅适用于 leader 的约束列表。
FOLLOWER_CONSTRAINTS	仅适用于 follower 的约束列表。
LEARNER_CONSTRAINTS	仅适用于 learner 的约束列表。
LEARNERS	指定 learner 的数量。
SURVIVAL_PREFERENCE	指定按label 容灾等级的优先级放置副本。例如 SURVIVAL_PREFERENCE="[region, zone, host]"。

4.8.17.4 示例

4.8.17.4.1 增加副本数

max-replicas 配置项的默认值为 3。如要为特定的表调大该值，可使用配置策略，示例如下：

```
CREATE PLACEMENT POLICY fivereplicas FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=fivereplicas;
```

注意，PD 配置中包含了leader 数和 follower 数。因此，5 个副本为 4 个 follower + 1 个 leader。

对于以上示例，你还可使用 PRIMARY_REGION 和 REGIONS 选项来描述 follower 的放置规则：

```
CREATE PLACEMENT POLICY eastandwest PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-east-2,us-
    ⟷ west-1" SCHEDULE="MAJORITY_IN_PRIMARY" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=eastandwest;
```

SCHEDULE 选项指示TiDB 如何平衡 follower。该选项默认的 EVEN 调度规则确保 follower 在所有区域内分布平衡。

如要保证在主区域内 (us-east-1) 放置足够多的 follower 副本，你可以使用 MAJORITY_IN_PRIMARY 调度规则来使该区域的 follower 达到指定数量。该调度牺牲一些可用性来换取更低的事务延迟。如果主区域宕机，MAJORITY_IN_PRIMARY 无法提供自动故障转移。

4.8.17.4.2 为分区表指定放置规则

除了给表绑定放置策略之外，你还可以给表分区绑定放置策略。示例如下：

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=5;
CREATE PLACEMENT POLICY europe PRIMARY_REGION="eu-central-1" REGIONS="eu-central-1,eu-west-1";
```

```
CREATE PLACEMENT POLICY northamerica PRIMARY_REGION="us-east-1" REGIONS="us-east-1";

SET tidb_enable_list_partition = 1;
CREATE TABLE t1 (
  country VARCHAR(10) NOT NULL,
  userdata VARCHAR(100) NOT NULL
) PLACEMENT POLICY=p1 PARTITION BY LIST COLUMNS (country) (
  PARTITION pEurope VALUES IN ('DE', 'FR', 'GB') PLACEMENT POLICY=europe,
  PARTITION pNorthAmerica VALUES IN ('US', 'CA', 'MX') PLACEMENT POLICY=northamerica,
  PARTITION pAsia VALUES IN ('CN', 'KR', 'JP')
);
```

如果分区没有绑定任何放置策略，分区将尝试继承表上可能存在的策略。比如，pEurope 分区将会应用 europe 策略，而 pAsia 分区将会应用表 t1 的放置策略 p1。如果 t1 没有绑定任何策略，pAsia 就不会应用任何策略。

给分区绑定放置策略后，你可以更改指定分区的放置策略。示例如下：

```
ALTER TABLE t1 PARTITION pEurope PLACEMENT POLICY=p1;
```

4.8.17.4.3 为数据库配置默认的放置规则

你可以为某个数据库指定默认的放置策略，类似于为数据库设置默认字符集或排序规则。如果没有指定其他选项，就会使用数据库上指定的配置。示例如下：

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-east-2";  --
    <-> 创建放置策略

CREATE PLACEMENT POLICY p2 FOLLOWERS=4;

CREATE PLACEMENT POLICY p3 FOLLOWERS=2;

CREATE TABLE t1 (a INT);  -- 创建表 t1，且未指定放置规则。

ALTER DATABASE test PLACEMENT POLICY=p2;  -- 更改默认的放置规则，但更改不影响已有的表 t1。

CREATE TABLE t2 (a INT);  -- 创建表 t2，默认的放置策略 p2 在 t2 上生效。

CREATE TABLE t3 (a INT) PLACEMENT POLICY=p1;  -- 创建表 t3。因为语句中已经指定了其他放置规则，
    <-> 默认的 p2 策略在 t3 上不生效。

ALTER DATABASE test PLACEMENT POLICY=p3;  -- 再次更改默认的放置规则，此更改不影响已有的表。

CREATE TABLE t4 (a INT);  -- 创建表 t4，默认的放置策略 p3 生效。

ALTER PLACEMENT POLICY p3 FOLLOWERS=3; -- 绑定策略 p3 的表，也就是 t4，会采用 FOLLOWERS=3。
```

注意分区与表之间的继承和这里的继承不同。改变表的放置策略，也会让分区应用新的策略。但是只有建表时没有指定放置策略的时候，表才会从数据库继承放置策略，且之后再改变数据库也不影响已经继承的表。

4.8.17.4.4 高级放置选项

放置选项 PRIMARY_REGION、REGIONS 和 SCHEDULE 可满足数据放置的基本需求，但会缺乏一些灵活性。在较复杂的场景下，若需要更灵活地放置数据，可以使用高级放置选项 CONSTRAINTS 和 FOLLOWER_CONSTRAINTS。PRIMARY_REGION、REGIONS 和 SCHEDULE 选项不可与 CONSTRAINTS 选项同时指定，否则会报错。

以下示例设置了一个约束，要求数据必须位于某个TiKV 节点，且该节点的 disk 标签必须匹配特定的值：

```
CREATE PLACEMENT POLICY storageonnvme CONSTRAINTS="[+disk=nvme]";
CREATE PLACEMENT POLICY storageonssd CONSTRAINTS="[+disk=ssd]";
CREATE PLACEMENT POLICY companystandardpolicy CONSTRAINTS="";

CREATE TABLE t1 (id INT, name VARCHAR(50), purchased DATE)
PLACEMENT POLICY=companystandardpolicy
PARTITION BY RANGE( YEAR(purchased) ) (
  PARTITION p0 VALUES LESS THAN (2000) PLACEMENT POLICY=storageonssd,
  PARTITION p1 VALUES LESS THAN (2005),
  PARTITION p2 VALUES LESS THAN (2010),
  PARTITION p3 VALUES LESS THAN (2015),
  PARTITION p4 VALUES LESS THAN MAXVALUE PLACEMENT POLICY=storageonnvme
);
```

该约束可通过列表格式([+disk=ssd])或字典格式({+disk=ssd: 1,+disk=nvme: 2})指定。

在列表格式中，约束以键值对列表格式。键以+ 或- 开头。+disk=nvme 表示disk 标签必须设为nvme，-disk=nvme 表示 disk 标签值不能为 nvme。

在字典格式中，约束还指定了适用于该规则的多个实例。例如，FOLLOWER_CONSTRAINTS="{+region=us-east-1: 1,+region=us-east-2: 1,+region=us-west-1: 1}"；表示1 个follower 位于us-east-1，1 个follower 位于us-east-2，1 个follower 位于us-west-1。再例如，FOLLOWER_CONSTRAINTS='{"+region=us-east-1,+disk=nvme": 1,"+region=us-west-1": 1}'；表示 1 个 follower 位于 us-east-1 区域中有 nvme 硬盘的机器上，1 个 follower 位于 us-west-1。

> **注意：**
>
> 字典和列表格式都基于 YAML 解析，但 YAML 语法有些时候不能被正常解析。例如 YAML 会把 "{+disk=ssd:1,+disk=nvme:2}"（: 后无空格）错误地解析成 '{"+disk=ssd:1": null, "+disk=nvme:2": null}'，不符合预期。但 "{+disk=ssd: 1,+disk=nvme: 2}"（: 后有空格）能被正确解析成 '{"+disk=ssd": 1, "+disk=nvme": 2}'。

4.8.17.4.5 生存偏好

在创建或修改放置策略时，你可以使用 SURVIVAL_PREFERENCES 选项设置数据的生存能力偏好。

例如，假设你的TiDB 集群分布在 3 个 zone（即可用区），且每个可用区的 host（即节点）上混合部署了多个 TiKV 实例。在为该集群创建放置策略时，假设 SURVIVAL_PREFERENCES 的设置如下：

```
CREATE PLACEMENT POLICY multiaz SURVIVAL_PREFERENCES="[zone, host]";
CREATE PLACEMENT POLICY singleaz CONSTRAINTS="[+zone=zone1]" SURVIVAL_PREFERENCES="[host]";
```

创建好放置策略后，你可以按需将放置策略绑定到对应的表上：

- 对于绑定了 multiaz 放置策略的表，数据将以 3 副本的形式放置在不同的可用区里，优先满足跨 zone 级别数据隔离的生存目标，再满足跨 host 级别的数据隔离的生存目标。
- 对于绑定了 singleaz 放置策略的表，数据会优先以 3 副本的形式全部放置在 zone1 这个可用区里，再满足跨 host 级别的数据隔离的生存目标。

> 注意：
>
> SURVIVAL_PREFERENCES 和 PD 中的 location-labels 是等价的，更多信息可以参考通过拓扑 label 进行副本调度。

4.8.17.5 工具兼容性

工具名称	最低兼容版本	说明
Backup & Restore (BR)	6.0	BR 在 v6.0 之前不支持放置规则的备份与恢复，见恢复 Placement Rule 到集群时为什么会报
TiDB Lightning	暂时不兼容	导入包含放置策略的数据时会报错
TiCDC	6.0	忽略放置规则，不同步规则到下游集群
TiDB Binlog	6.0	忽略放置规则，不同步规则到下游集群

4.8.17.6 使用限制

目前已知 Placement Rules in SQL 特性存在以下限制：

- 临时表不支持放置规则。
- 设置 PRIMARY_REGION 和 REGIONS 时允许存在语法糖。但在未来版本中， 我们计划为 PRIMARY_RACK、PRIMARY_ZONE 和 PRIMARY_HOST 添加变体支持，见issue #18030。
- 放置规则仅保证静态数据被放置在正确的TiKV 节点上。该规则不保证传输中的数据（通过用户查询或内部操作）只出现在特定区域内。

4.8.18 系统表

4.8.18.1 mysql Schema

mysql 库里存储的是TiDB 系统表。该设计类似于 MySQL 中的 mysql 库，其中 mysql.user 之类的表可以直接编辑。该库还包含许多 MySQL 的扩展表。

4.8.18.1.1 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
- `password_history` 记录密码更改历史
- `default_roles` 默认启用的角色
- `global_grants` 动态权限
- `global_priv` 基于证书的认证信息
- `role_edges` 角色之间的关系信息

4.8.18.1.2 服务端帮助信息系统表

- `help_topic` 目前为空

4.8.18.1.3 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_top_n` 统计信息的TopN
- `stats_meta` 表的元信息，比如总行数和修改数
- `stats_extended` 扩展统计信息，比如列之间的顺序相关性
- `stats_feedback` 统计信息的查询反馈
- `stats_fm_sketch` 统计信息列的直方图 FMSketch 分布
- `analyze_options` 各个表默认的 `analyze` 参数
- `column_stats_usage` 列统计信息的使用情况
- `schema_index_usage` 索引的使用情况
- `analyze_jobs` 正在执行的统计信息收集任务以及过去 7 天内的历史任务记录

4.8.18.1.4 执行计划相关系统表

- `bind_info` 执行计划的绑定信息
- `capture_plan_baselines_blacklist` 关于自动绑定执行计划对象的黑名单

4.8.18.1.5 GC Worker 相关系统表

- `gc_delete_range` 需要被GC worker 定期删除的 KV 范围段
- `gc_delete_range_done` 已经被删除的 KV 范围段

4.8.18.1.6 缓存表使用的系统表

- `table_cache_meta` 存储了缓存表的元信息

4.8.18.1.7 TTL 相关系统表

- `mysql.tidb_ttl_table_status` 所有 TTL 表的上一次执行与正在执行的 TTL 任务
- `mysql.tidb_ttl_task` 正在执行的 TTL 子任务
- `mysql.tidb_ttl_job_history` 过去 90 天内 TTL 任务的执行历史

4.8.18.1.8 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
- `expr_pushdown_blacklist` 表达式下推的黑名单
- `opt_rule_blacklist` 逻辑优化规则的黑名单
- `table_cache_meta` 缓存表的信息

4.8.18.2 INFORMATION_SCHEMA

4.8.18.2.1 Information Schema

Information Schema 提供了一种查看系统元数据的ANSI 标准方法。除了包含与 MySQL 兼容的表外，TiDB 还提供了许多自定义的 `INFORMATION_SCHEMA` 表。

许多 `INFORMATION_SCHEMA` 表都有相应的 `SHOW` 命令。查询 `INFORMATION_SCHEMA` 的好处是可以在表之间进行 `join` 操作。

与 MySQL 兼容的表

表名	描述
CHARACTER_SETS	提供TiDB 支持的字符集列表。
COLLATIONS	提供 TiDB 支持的排序规则列表。
COLLATION_CHARACTER_SET_APPLICABILITY	说明哪些排序规则适用于哪些字符集。
COLUMNS	提供所有表中列的列表。
COLUMN_PRIVILEGES	TiDB 未实现，返回零行。
COLUMN_STATISTICS	TiDB 未实现，返回零行。
ENGINES	提供支持的存储引擎列表。
EVENTS	TiDB 未实现，返回零行。
FILES	TiDB 未实现，返回零行。
GLOBAL_STATUS	TiDB 未实现，返回零行。
GLOBAL_VARIABLES	TiDB 未实现，返回零行。
KEY_COLUMN_USAGE	描述列的键约束，例如主键约束。
OPTIMIZER_TRACE	TiDB 未实现，返回零行。
PARAMETERS	TiDB 未实现，返回零行。
PARTITIONS	提供表分区的列表。
PLUGINS	TiDB 未实现，返回零行。
PROCESSLIST	提供与 `SHOW PROCESSLIST` 命令类似的信息。
PROFILING	TiDB 未实现，返回零行。
REFERENTIAL_CONSTRAINTS	提供有关 `FOREIGN KEY` 约束的信息。

表名	描述
ROUTINES	TiDB 未实现，返回零行。
SCHEMATA	提供与 SHOW DATABASES 命令类似的信息。
SCHEMA_PRIVILEGES	TiDB 未实现，返回零行。
SESSION_STATUS	TiDB 未实现，返回零行。
SESSION_VARIABLES	提供与 SHOW SESSION VARIABLES 命令类似的功能。
STATISTICS	提供有关表索引的信息。
TABLES	提供当前用户可见的表的列表。类似于 SHOW TABLES。
TABLESPACES	TiDB 未实现，返回零行。
TABLE_CONSTRAINTS	提供有关主键、唯一索引和外键的信息。
TABLE_PRIVILEGES	TiDB 未实现，返回零行。
TRIGGERS	TiDB 未实现，返回零行。
USER_ATTRIBUTES	汇总用户的注释和属性信息。
USER_PRIVILEGES	汇总与当前用户相关的权限。
VARIABLES_INFO	提供TiDB 系统变量的信息。
VIEWS	提供当前用户可见的视图列表。类似于 SHOW FULL TABLES WHERE table_type = '

TiDB 中的扩展表

表名	描述
ANALYZE_STATUS	提供有关收集统计信息的任务的信息。
CLIENT_ERRORS_SUMMARY_BY_HOST	汇总由客户端请求生成并返回给客户端的错误和警告。
CLIENT_ERRORS_SUMMARY_BY_USER	汇总由客户端产生的错误和警告。
CLIENT_ERRORS_SUMMARY_GLOBAL	汇总由客户端产生的错误和警告。
CLUSTER_CONFIG	提供有关整个TiDB 集群的配置设置的详细信息。
CLUSTER_DEADLOCKS	提供 DEADLOCKS 表的集群级别的视图。
CLUSTER_HARDWARE	提供在每个TiDB 组件上发现的底层物理硬件的详细信息。
CLUSTER_INFO	提供当前集群拓扑的详细信息。
CLUSTER_LOAD	提供集群中TiDB 服务器的当前负载信息。
CLUSTER_LOG	提供整个TiDB 集群的日志。
CLUSTER_MEMORY_USAGE	提供 MEMORY_USAGE 表的集群级别的视图。
CLUSTER_MEMORY_USAGE_OPS_HISTORY	提供 MEMORY_USAGE_OPS_HISTORY 表的集群级别的视图。
CLUSTER_PROCESSLIST	提供 PROCESSLIST 表的集群级别的视图。
CLUSTER_SLOW_QUERY	提供 SLOW_QUERY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY	提供 STATEMENTS_SUMMARY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY_HISTORY	提供 STATEMENTS_SUMMARY_HISTORY 表的集群级别的视图。
CLUSTER_TIDB_TRX	提供 TIDB_TRX 表的集群级别的视图。
CLUSTER_SYSTEMINFO	提供集群中服务器的内核参数配置的详细信息。
DATA_LOCK_WAITS	提供TiKV 服务器上的等锁信息。
DDL_JOBS	提供与 ADMIN SHOW DDL JOBS 类似的输出。
DEADLOCKS	提供TiDB 节点上最近发生的数次死锁错误的信息。
INSPECTION_RESULT	触发内部诊断检查。
INSPECTION_RULES	进行的内部诊断检查的列表。
INSPECTION_SUMMARY	重要监视指标的摘要报告。

表名	描述
MEMORY_USAGE	提供当前 TiDB 实例的内存使用情况。
MEMORY_USAGE_OPS_HISTORY	提供当前 TiDB 实例内存相关的历史操作和执行依据。
METRICS_SUMMARY	从 Prometheus 获取的指标的摘要。
METRICS_SUMMARY_BY_LABEL	参见 METRICS_SUMMARY 表。
METRICS_TABLES	为 METRICS_SCHEMA 中的表提供 PromQL 定义。
PLACEMENT_POLICIES	提供所有放置策略的定义信息。
SEQUENCES	描述了基于 MariaDB 实现的TiDB 序列。
SLOW_QUERY	提供当前TiDB 服务器上慢查询的信息。
STATEMENTS_SUMMARY	类似于 MySQL 中的 performance_schema 语句摘要。
STATEMENTS_SUMMARY_HISTORY	类似于 MySQL 中的 performance_schema 语句摘要历史。
TABLE_STORAGE_STATS	提供存储的表的大小的详细信息。
TIDB_HOT_REGIONS	提供有关哪些 Region 访问次数最多的统计信息。
TIDB_HOT_REGIONS_HISTORY	提供有关哪些 Region 访问次数最多的历史统计信息。
TIDB_INDEXES	提供有关 TiDB 表的索引信息。
TIDB_SERVERS_INFO	提供TiDB 服务器的列表
TIDB_TRX	提供TiDB 节点上正在执行的事务的信息。
TIFLASH_REPLICA	提供有关TiFlash 副本的详细信息。
TIKV_REGION_PEERS	提供Region 存储位置的详细信息。
TIKV_REGION_STATUS	提供Region 的统计信息。
TIKV_STORE_STATUS	提供TiKV 服务器的基本信息。

4.8.18.2.2 ANALYZE_STATUS

ANALYZE_STATUS 表提供正在执行的收集统计信息的任务以及有限条历史任务记录。

从 TiDB v6.1.0 起，ANALYZE_STATUS 表显示集群级别的任务，且TiDB 重启后仍能看到重启之前的任务记录。在 TiDB v6.1.0 之前，ANALYZE_STATUS 仅显示实例级别的任务，且TiDB 重启后任务记录会被清空。

从TiDB v6.1.0 起，可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史任务记录。

```
USE information_schema;
DESC analyze_status;
```

```
+----------------+---------------------+------+------+---------+----------------+
| Field          | Type                | Null | Key  | Default | Extra          |
+----------------+---------------------+------+------+---------+----------------+
| TABLE_SCHEMA   | varchar(64)         | YES  |      | NULL    |                |
| TABLE_NAME     | varchar(64)         | YES  |      | NULL    |                |
| PARTITION_NAME | varchar(64)         | YES  |      | NULL    |                |
| JOB_INFO       | longtext            | YES  |      | NULL    |                |
| PROCESSED_ROWS | bigint(64) unsigned | YES  |      | NULL    |                |
| START_TIME     | datetime            | YES  |      | NULL    |                |
| END_TIME       | datetime            | YES  |      | NULL    |                |
| STATE          | varchar(64)         | YES  |      | NULL    |                |
| FAIL_REASON    | longtext            | YES  |      | NULL    |                |
```

```
| INSTANCE        | varchar(512)         | YES  |      | NULL    |        |
| PROCESS_ID      | bigint(64) unsigned  | YES  |      | NULL    |        |
+-----------------+----------------------+------+------+---------+--------------+
11 rows in set (0.00 sec)
```

```
SELECT * FROM information_schema.analyze_status;
```

```
+--
    <->             +             +                 +-------------------------------------------------------------------------------
    <->
| TABLE_SCHEMA | TABLE_NAME | PARTITION_NAME | JOB_INFO
    <->                                                                  | PROCESSED_ROWS | START_TIME
    <->           | END_TIME             | STATE     | FAIL_REASON | INSTANCE        | PROCESS_ID |
+--
    <> -------------+-------------+-----------------+                                                 ---------------
    <->
| test          | t           | p1              | analyze table all columns with 256 buckets, 500
    <-> topn, 1 samplerate |               0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    <-> finished | NULL         | 127.0.0.1:4000 |       NULL |
| test          | t           | p0              | analyze table all columns with 256 buckets, 500
    <-> topn, 1 samplerate |               0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    <-> finished | NULL         | 127.0.0.1:4000 |       NULL |
| test          | t           | p1              | analyze index idx
    <->                                                  |               0 | 2022-05-27 11:29:46 |
    <->  2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
| test          | t           | p0              | analyze index idx
    <->                                                  |               0 | 2022-05-27 11:29:46 |
    <->  2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
| test          | t           | p1              | analyze columns
    <->                                                    |               0 | 2022-05-27 11:29:46
    <->  | 2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
| test          | t           | p0              | analyze columns
    <->                                                    |               0 | 2022-05-27 11:29:46
    <->  | 2022-05-27 11:29:46 | finished | NULL         | 127.0.0.1:4000 |       NULL |
+--
    <->             +             +                 +-------------------------------------------------------------------------------
    <->
6 rows in set (0.00 sec)
```

ANALYZE_STATUS 表中列的含义如下：

- TABLE_SCHEMA：表所属的数据库的名称。
- TABLE_NAME：表的名称。
- PARTITION_NAME：分区表的名称。

- JOB_INFO：ANALYZE 任务的信息。如果分析索引，该信息会包含索引名。当 tidb_analyze_version =2 时，该信息会包含采样率等配置项。
- PROCESSED_ROWS：已经处理的行数。
- START_TIME：ANALYZE 任务的开始时间。
- END_TIME：ANALYZE 任务的结束时间。
- STATE：ANALYZE 任务的执行状态。其值可以是 pending、running、finished 或者 failed。
- FAIL_REASON：任务失败的原因。如果执行成功则为 NULL。
- INSTANCE：执行任务的 TiDB 实例。
- PROCESS_ID：执行任务的 process ID。

4.8.18.2.3 CLIENT_ERRORS_SUMMARY_BY_HOST

CLIENT_ERRORS_SUMMARY_BY_HOST 表汇总了已返回给连接到TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。INFORMATION_SCHEMA <—> .CLIENT_ERRORS_SUMMARY_BY_HOST 表提供了一种有效方法，能够在应用程序没有正确处理（或记录）TiDB 服务器返回的错误的情况下检查错误。

由于 CLIENT_ERRORS_SUMMARY_BY_HOST 会基于每个远程主机汇总错误，在诊断其中一台应用程序服务器比其他服务器生成更多错误的情况时很有用。可能的情况包括：

- 过时的 MySQL 客户端库。
- 过时的应用程序（可能是在推出新部署时遗漏了此服务器）。
- 用户权限中“主机”部分的使用不正确。
- 网络连接不可靠，导致更多超时或连接断开。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个TiDB 服务器的本地数据，并且只保留在内存中。如果TiDB 服务器重新启动，会丢失汇总信息。

```
USE INFORMATION_SCHEMA;
DESC CLIENT_ERRORS_SUMMARY_BY_HOST;
```

输出结果如下：

```
+---------------+---------------+------+------+---------+-------------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------------+
| HOST          | varchar(255)  | NO   |      | NULL    |       |
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
```

```
| ERROR_COUNT    | bigint(64)    | NO   |       | NULL    |       |
| WARNING_COUNT  | bigint(64)    | NO   |       | NULL    |       |
| FIRST_SEEN     | timestamp     | YES  |       | NULL    |       |
| LAST_SEEN      | timestamp     | YES  |       | NULL    |       |
+---------------+---------------+------+------+---------+-------------+
7 rows in set (0.00 sec)
```

字段说明如下：

- HOST：客户端的远程主机。
- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：此错误返回到客户端主机的次数。
- WARNING_COUNT：此警告返回到客户端主机的次数。
- FIRST_SEEN：首次从客户端主机看到此错误（或警告）的时间。
- LAST_SEEN：最近一次从客户端主机看到此错误（或警告）的时间。

以下示例显示了客户端连接到本地TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_HOST;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_HOST;
```

输出结果如下：

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    <-> ---------+--------------+---------------+-------------+---------------+---------------------+------------
    <->
| HOST      | ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          |
    <-> LAST_SEEN            |
+--
    <-> ---------+--------------+---------------+-------------+---------------+---------------------+------------
    <->
| 127.0.0.1 |         1365 | Division by 0 |           0 |             1 | 2021-03-18 12:51:54 |
    <-> 2021-03-18 12:51:54 |
+--
    <-> ---------+--------------+---------------+-------------+---------------+---------------------+------------
    <->
```

```
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

4.8.18.2.4 CLIENT_ERRORS_SUMMARY_BY_USER

CLIENT_ERRORS_SUMMARY_BY_USER 表汇总了已返回给连接到TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。INFORMATION_SCHEMA <—> .CLIENT_ERRORS_SUMMARY_BY_USER 表提供了一种有效方法，能够在应用程序没有正确处理（或记录）TiDB 服务器返回的错误的情况下检查错误。

因为 CLIENT_ERRORS_SUMMARY_BY_USER 会基于每个用户汇总错误，所以在诊断一个用户服务器比其他服务器产生更多错误的方案时很有用。可能的情况包括：

- 权限错误。
- 表或关系对象丢失。
- SQL 语法不正确，或应用程序和TiDB 版本之间不兼容。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个TiDB 服务器的本地数据，并且只保留在内存中。如果TiDB 服务器重新启动，会丢失汇总信息。

```
USE INFORMATION_SCHEMA;
DESC CLIENT_ERRORS_SUMMARY_BY_USER;
```

```
+---------------+---------------+------+------+---------+-------------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------------+
| USER          | varchar(64)   | NO   |      | NULL    |       |
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------------+
7 rows in set (0.00 sec)
```

字段说明如下：

- USER：已认证的用户。
- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：返回此错误给用户的次数。
- WARNING_COUNT：返回此警告给用户的次数。
- FIRST_SEEN：首次向用户发送此错误（或警告）的时间。
- LAST_SEEN：最近一次向用户发送此错误（或警告）的时间。

以下示例显示了客户端连接到本地TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_USER;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_USER;
```

输出结果如下：

```
+------+
| 0/0 |
+------+
| NULL |
+------+
1 row in set, 1 warning (0.00 sec)

+--
    <-> ----+--------------+---------------+-------------+---------------+---------------------+-----------------
    <->
| USER | ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN           |
    <-> LAST_SEEN            |
+--
    <-> ----+--------------+---------------+-------------+---------------+---------------------+-----------------
    <->
| root |         1365 | Division by 0 |           0 |             1 | 2021-03-18 13:05:36 |
    <-> 2021-03-18 13:05:36 |
+--
    <-> ----+--------------+---------------+-------------+---------------+---------------------+-----------------
    <->
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

4.8.18.2.5 CLIENT_ERRORS_SUMMARY_GLOBAL

CLIENT_ERRORS_SUMMARY_GLOBAL 表汇总了已返回给连接到TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。information_schema <⟶ .CLIENT_ERRORS_SUMMARY_BY_GLOBAL 表提供了高级概述，在应用程序无法正确处理（或记录）TiDB 服务器返回的错误的情况下很有用。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个TiDB 服务器的本地数据，并且只保留在内存中。如果TiDB 服务器重新启动，会丢失汇总信息。

```
USE information_schema;
DESC CLIENT_ERRORS_SUMMARY_GLOBAL;
```

输出结果如下：

```
+---------------+---------------+------+------+---------+-------------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------------+
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------------+
6 rows in set (0.00 sec)
```

字段说明如下：

- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：返回此错误的次数。
- WARNING_COUNT：返回此警告的次数。
- FIRST_SEEN：首次返回此错误（或警告）的时间。
- LAST_SEEN：最近一次发送此错误（或警告）的时间。

以下示例显示了客户端连接到本地TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_GLOBAL;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_GLOBAL;
```

输出结果如下：

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    <-> ------------+---------------+-------------+---------------+---------------------+----------------------------------+
    <->
| ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN           | LAST_SEEN
    <->           |
+--
    <-> ------------+---------------+-------------+---------------+---------------------+----------------------------------+
    <->
|         1365 | Division by 0 |           0 |             1 | 2021-03-18 13:10:51 | 2021-03-18
    <-> 13:10:51 |
+--
    <-> ------------+---------------+-------------+---------------+---------------------+----------------------------------+
    <->
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

4.8.18.2.6 CHARACTER_SETS

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

```
USE INFORMATION_SCHEMA;
DESC CHARACTER_SETS;
```

输出结果如下：

```
+----------------------+-------------+------+------+---------+-------------+
| Field                | Type        | Null | Key  | Default | Extra |
+----------------------+-------------+------+------+---------+-------------+
| CHARACTER_SET_NAME   | varchar(32) | YES  |      | NULL    |       |
```

```
| DEFAULT_COLLATE_NAME | varchar(32) | YES   |       | NULL    |        |
| DESCRIPTION          | varchar(60) | YES   |       | NULL    |        |
| MAXLEN               | bigint(3)   | YES   |       | NULL    |        |
+----------------------+-------------+-------+-------+---------+--------+
4 rows in set (0.00 sec)
```

查看 CHARACTER_SETS 表的信息：

```
SELECT * FROM `CHARACTER_SETS`;
```

输出结果如下：

```
+--------------------+----------------------+---------------------------------------+--------+
| CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME | DESCRIPTION                           | MAXLEN |
+--------------------+----------------------+---------------------------------------+--------+
| ascii              | ascii_bin            | US ASCII                              |      1 |
| binary             | binary               | binary                                |      1 |
| gbk                | gbk_chinese_ci       | Chinese Internal Code Specification |      2 |
| latin1             | latin1_bin           | Latin1                                |      1 |
| utf8               | utf8_bin             | UTF-8 Unicode                         |      3 |
| utf8mb4            | utf8mb4_bin          | UTF-8 Unicode                         |      4 |
+--------------------+----------------------+---------------------------------------+--------+
6 rows in set (0.00 sec)
```

CHARACTER_SETS 表中列的含义如下：

- CHARACTER_SET_NAME：字符集名称
- DEFAULT_COLLATE_NAME：字符集的默认排序规则名称
- DESCRIPTION：字符集的描述信息
- MAXLEN：该字符集存储一个字符所需要的最大字节数

4.8.18.2.7 CLUSTER_CONFIG

CLUSTER_CONFIG 表用于获取集群当前所有组件实例的配置。在TiDB 早期的版本，用户需要逐个访问各个实例的 HTTP API 才能收集到所有组件配置。TiDB v4.0 后，该表的引入提高了易用性。

```
USE information_schema;
DESC cluster_config;
```

```
+----------+--------------+------+------+---------+-------+
| Field    | Type         | Null | Key  | Default | Extra |
+----------+--------------+------+------+---------+-------+
| TYPE     | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE | varchar(64)  | YES  |      | NULL    |       |
| KEY      | varchar(256) | YES  |      | NULL    |       |
| VALUE    | varchar(128) | YES  |      | NULL    |       |
+----------+--------------+------+------+---------+-------+
```

字段解释：

- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- KEY：配置项名。
- VALUE：配置项值。

以下示例查询TiKV 实例的 coprocessor 相关配置：

```
SELECT * FROM cluster_config WHERE type='tikv' AND `key` LIKE 'coprocessor%';
```

```
+------+-----------------+-----------------------------------+-----------------+
| TYPE | INSTANCE        | KEY                               | VALUE           |
+------+-----------------+-----------------------------------+-----------------+
| tikv | 127.0.0.1:20165 | coprocessor.batch-split-limit     | 10              |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-keys       | 1440000         |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-size       | 144MiB          |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-keys     | 960000          |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-size     | 96MiB           |
| tikv | 127.0.0.1:20165 | coprocessor.split-region-on-table | false           |
+------+-----------------+-----------------------------------+-----------------+
6 rows in set (0.00 sec)
```

4.8.18.2.8 CLUSTER_HARDWARE

集群硬件表 CLUSTER_HARDWARE 提供了集群各实例所在服务器的硬件信息。

```
USE information_schema;
DESC cluster_hardware;
```

```
+-------------+--------------+------+------+---------+-------------+
| Field       | Type         | Null | Key  | Default | Extra       |
+-------------+--------------+------+------+---------+-------------+
| TYPE        | varchar(64)  | YES  |      | NULL    |             |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |             |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |             |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |             |
| NAME        | varchar(256) | YES  |      | NULL    |             |
| VALUE       | varchar(128) | YES  |      | NULL    |             |
+-------------+--------------+------+------+---------+-------------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应集群信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。

- INSTANCE：对应于集群信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型。目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 的取值不同。
  - cpu：硬件名为 cpu。
  - memory：硬件名为 memory。
  - disk：磁盘名。
  - net：网卡名。
- NAME：硬件不同的信息名，比如 cpu 有 cpu-logical-cores 和 cpu-physical-cores 两个信息名，表示逻辑核心数量和物理核心数量。
- VALUE：对应硬件信息的值。例如磁盘容量和 CPU 核数。

查询集群 CPU 信息的示例如下：

```
SELECT * FROM cluster_hardware WHERE device_type='cpu' AND device_name='cpu' AND name LIKE '%
    <-> cores';
```

```
+------+-----------------+-------------+-------------+--------------------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME               | VALUE |
+------+-----------------+-------------+-------------+--------------------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-logical-cores  | 16    |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-physical-cores | 8     |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-logical-cores  | 16    |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-physical-cores | 8     |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-logical-cores  | 16    |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-physical-cores | 8     |
+------+-----------------+-------------+-------------+--------------------+-------+
6 rows in set (0.03 sec)
```

4.8.18.2.9 CLUSTER_INFO

集群拓扑表 CLUSTER_INFO 提供集群当前的拓扑信息，以及各个节点的版本信息、版本对应的Git Hash、各节点的启动时间、各实例的运行时间。

```
USE information_schema;
desc cluster_info;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| INSTANCE       | varchar(64) | YES  |      | NULL    |       |
| STATUS_ADDRESS | varchar(64) | YES  |      | NULL    |       |
| VERSION        | varchar(64) | YES  |      | NULL    |       |
| GIT_HASH       | varchar(64) | YES  |      | NULL    |       |
| START_TIME     | varchar(32) | YES  |      | NULL    |       |
```

```
| UPTIME          | varchar(32) | YES  |      | NULL    |       |
| SERVER_ID       | bigint(21)  | YES  |      | NULL    |       |
+-----------------+-------------+------+------+---------+-------+
8 rows in set (0.01 sec)
```

字段解释：

- TYPE：节点类型，目前节点的可取值为 tidb，pd 和 tikv。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。部分 tikv-ctl、pd-ctl 或 tidb-ctl 命令会使用到 HTTP API 和该地址。用户也可以通过该地址获取一些额外的集群信息，详情可参考 HTTP API 文档。
- VERSION：对应节点的语义版本号。TiDB 版本为了兼容 MySQL 的版本号，以 ${mysql-version}-${tidb-version} 的格式展示版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。
- SERVER_ID：对应节点的服务器 ID。

```
SELECT * FROM cluster_info;
```

```
+--
    <-> ----+-----------------+-----------------+-------------+-----------------------------------------+------
    <->
| TYPE | INSTANCE         | STATUS_ADDRESS   | VERSION       | GIT_HASH
    <->                                  | START_TIME                 | UPTIME               |
+--
    <-> ----+-----------------+-----------------+-------------+-----------------------------------------+------
    <->
| tidb | 0.0.0.0:4000     | 0.0.0.0:10080    | 4.0.0-beta.2 | 0
    <-> df3b74f55f8f8fbde39bbd5d471783f49dc10f7 | 2020-07-05T09:25:53-06:00 | 26h39m4.352862693s
    <-> |
| pd   | 127.0.0.1:2379   | 127.0.0.1:2379   | 4.1.0-alpha   | 1
    <-> ad59bcbf36d87082c79a1fffa3b0895234ac862 | 2020-07-05T09:25:47-06:00 | 26h39m10.352868103s
    <-> |
| tikv | 127.0.0.1:20165 | 127.0.0.1:20180 | 4.1.0-alpha   |
    <-> b45e052df8fb5d66aa8b3a77b5c992ddbfbb79df | 2020-07-05T09:25:50-06:00 | 26h39m7.352869963s
    <>  |
+--
    <-> ----+-----------------+-----------------+-------------+-----------------------------------------+------
    <->
3 rows in set (0.00 sec)
```

4.8.18.2.10 CLUSTER_LOAD

集群负载表 CLUSTER_LOAD 提供集群各个实例所在服务器的当前负载信息。

```
USE information_schema;
DESC cluster_load;
```

```
+-------------+--------------+------+------+---------+-------------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型，目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 取值不同。
  - cpu：硬件名为 cpu。
  - disk：磁盘名。
  - net：网卡名。
  - memory：硬件名为 memory。
- NAME：不同的负载类型。例如 cpu 有 load1/load5/load15 三个负载类型，分别表示 cpu 在 1min/5min/15min 内的平均负载。
- VALUE：硬件负载的值，例如 cpu 在 1min/5min/15min 内的平均负载。

查询集群当前的 CPU 负载信息示例如下：

```
SELECT * FROM cluster_load WHERE device_type='cpu' AND device_name='cpu';
```

```
+------+-----------------+-------------+-------------+---------+-------------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME    | VALUE |
+------+-----------------+-------------+-------------+---------+-------------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load1   | 0.13  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load5   | 0.25  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load15  | 0.31  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load1   | 0.13  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load5   | 0.25  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load15  | 0.31  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load1   | 0.13  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load5   | 0.25  |
```

```
| tikv | 127.0.0.1:20165 | cpu          | cpu          | load15 | 0.31   |
+------+-----------------+--------------+--------------+--------+--------------+
9 rows in set (1.50 sec)
```

4.8.18.2.11 CLUSTER_LOG

集群日志表 CLUSTER_LOG 表用于查询集群当前所有TiDB/PD/TiKV 节点日志。它通过将查询条件下推到各个节点，降低了日志查询对集群的影响。该表的查询性能优于 grep 命令。

TiDB 4.0 版本之前，要获取集群的日志，用户需要逐个登录各个节点汇总日志。TiDB 4.0 的集群日志表提供了一个全局且时间有序的日志搜索结果，为跟踪全链路事件提供了便利的手段。例如按照某一个 region id 搜索日志，可以查询该Region 生命周期内的所有日志；类似地，通过慢日志的 txn id 搜索全链路日志，可以查询该事务在各个节点扫描的 key 数量以及流量等信息。

```
USE information_schema;
DESC cluster_log;
```

```
+----------+------------------+------+------+---------+--------------+
| Field    | Type             | Null | Key  | Default | Extra        |
+----------+------------------+------+------+---------+--------------+
| TIME     | varchar(32)      | YES  |      | NULL    |              |
| TYPE     | varchar(64)      | YES  |      | NULL    |              |
| INSTANCE | varchar(64)      | YES  |      | NULL    |              |
| LEVEL    | varchar(8)       | YES  |      | NULL    |              |
| MESSAGE  | var_string(1024) | YES  |      | NULL    |              |
+----------+------------------+------+------+---------+--------------+
5 rows in set (0.00 sec)
```

字段解释：

- TIME：日志打印时间。
- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- LEVEL：日志级别。
- MESSAGE：日志内容。

> 注意：
>
> - 日志表的所有字段都会下推到对应节点执行， 所以为了降低使用集群日志表的开销， 必须指定搜索关键字以及时间范围， 然后尽可能地指定更多的条件。例如 select * from cluster_log where message like '%ddl%' and time > '2020-05-18 <⟶ 20:40:00' and time<'2020-05-18 21:40:00' and type='tidb'。

- message 字段支持 like 和 regexp 正则表达式，对应的 pattern 会编译为 regexp。同时指定多个 message 条件， 相当于 grep 命令的 pipeline 形式， 例如：select * from cluster_log where message like 'coprocessor%' and message regexp '.*slow.*' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00' 相当于在集群所有节点执行 grep 'coprocessor' xxx.log | grep -E '.*slow.*'。

查询某个 DDL 的执行过程示例如下：

```
SELECT time,instance,left(message,150) FROM cluster_log WHERE message LIKE '%ddl%job%ID.80%' AND type='tidb' AND time > '2020-05-18 20:40:00' AND time < '2020-05-18 21:40:00'
```

```
+----------------------------+-----------------+-----------------------------
```

- 用户将 DDL JOB ID 为 80 的请求发给 127.0.0.1:4002 TiDB 节点。
- 127.0.0.1:4000 TiDB 节点处理这个 DDL 请求，说明此时 127.0.0.1:4000 节点是 DDL owner。
- DDL JOB ID 为 80 的请求处理完成。

4.8.18.2.12 CLUSTER_SYSTEMINFO

内核参数表 CLUSTER_SYSTEMINFO 用于查询集群所有实例所在服务器的内核配置信息。目前支持查询 sysctl 的信息。

```
USE information_schema;
DESC cluster_systeminfo;
```

```
+-------------+--------------+------+------+---------+-------------+
| Field       | Type         | Null | Key  | Default | Extra       |
+-------------+--------------+------+------+---------+-------------+
| TYPE        | varchar(64)  | YES  |      | NULL    |             |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |             |
| SYSTEM_TYPE | varchar(64)  | YES  |      | NULL    |             |
| SYSTEM_NAME | varchar(64)  | YES  |      | NULL    |             |
| NAME        | varchar(256) | YES  |      | NULL    |             |
| VALUE       | varchar(128) | YES  |      | NULL    |             |
+-------------+--------------+------+------+---------+-------------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- SYSTEM_TYPE：系统类型，目前可以查询的系统类型有 system。
- SYSTEM_NAME：目前可以查询的 SYSTEM_NAME 为 sysctl。
- NAME：sysctl 对应的配置名。
- VALUE：sysctl 对应配置项的值。

查询集群所有服务器的内核版本示例如下：

```
SELECT * FROM cluster_systeminfo WHERE name LIKE '%kernel.osrelease%'
```

```
+--
    <-> ----+------------------+-------------+-------------+-----------------+ ---------------------------------------- +
    <->
| TYPE | INSTANCE           | SYSTEM_TYPE | SYSTEM_NAME | NAME              | VALUE
    <->                       |
+--
    <-> ----+------------------+-------------+-------------+-----------------+ ---------------------------------------- +
    <->
| tidb | 172.16.5.40:4008   | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    <-> x86_64 |
```

```
| pd   | 172.16.5.40:20379 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    <-> x86_64 |
| tikv | 172.16.5.40:21150 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    <-> x86_64 |
+--
    <-> ----+-------------------+-------------+-------------+------------------+ ---------------------------------------------------- +
    <->
```

4.8.18.2.13 COLLATIONS

COLLATIONS 表提供了 CHARACTER_SETS 表中字符集对应的排序规则列表。目前TiDB 包含该表仅为兼容 MySQL。

```
USE information_schema;
DESC collations;
```

```
+--------------------+-------------+------+------+---------+----------------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+----------------+
| COLLATION_NAME     | varchar(32) | YES  |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | YES  |      | NULL    |       |
| ID                 | bigint(11)  | YES  |      | NULL    |       |
| IS_DEFAULT         | varchar(3)  | YES  |      | NULL    |       |
| IS_COMPILED        | varchar(3)  | YES  |      | NULL    |       |
| SORTLEN            | bigint(3)   | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+----------------+
6 rows in set (0.00 sec)
```

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

```
+--------------------+--------------------+------+------------+-------------+ ---------------- +
| COLLATION_NAME     | CHARACTER_SET_NAME | ID   | IS_DEFAULT | IS_COMPILED | SORTLEN |
+--------------------+--------------------+------+------------+-------------+ ---------------- +
| utf8mb4_bin        | utf8mb4            |   46 | Yes        | Yes         |       1 |
| utf8mb4_general_ci | utf8mb4            |   45 |            | Yes         |       1 |
| utf8mb4_unicode_ci | utf8mb4            |  224 |            | Yes         |       1 |
+--------------------+--------------------+------+------------+-------------+ ---------------- +
3 rows in set (0.001 sec)
```

COLLATIONS 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称
- ID：排序规则的 ID
- IS_DEFAULT：该排序规则是否是所属字符集的默认排序规则
- IS_COMPILED：字符集是否编译到服务器中
- SORTLEN：排序规则在对字符进行排序时，所分配内存的最小长度

4.8.18.2.14 COLLATION_CHARACTER_SET_APPLICABILITY

COLLATION_CHARACTER_SET_APPLICABILITY 表将排序规则映射至适用的字符集名称。和 COLLATIONS 表一样，包含此表只是为了兼容 MySQL。

```
USE INFORMATION_SCHEMA;
DESC COLLATION_CHARACTER_SET_APPLICABILITY;
```

输出结果如下：

```
+--------------------+-------------+------+------+---------+----------------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+----------------+
| COLLATION_NAME     | varchar(32) | NO   |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | NO   |      | NULL    |       |
+--------------------+-------------+------+------+---------+----------------+
2 rows in set (0.00 sec)
```

查看 COLLATION_CHARACTER_SET_APPLICABILITY 表中 utf8mb4 字符集的排序规则映射：

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

输出结果如下：

```
+--------------------+--------------------+
| COLLATION_NAME     | CHARACTER_SET_NAME |
+--------------------+--------------------+
| utf8mb4_bin        | utf8mb4            |
| utf8mb4_general_ci | utf8mb4            |
| utf8mb4_unicode_ci | utf8mb4            |
+--------------------+--------------------+
3 rows in set (0.00 sec)
```

COLLATION_CHARACTER_SET_APPLICABILITY 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称

4.8.18.2.15 COLUMNS

COLUMNS 表提供了表的所有列的信息。

```
USE INFORMATION_SCHEMA;
DESC COLUMNS;
```

输出结果如下：

```
+--------------------------+---------------+------+------+---------+-------------+
| Field                    | Type          | Null | Key  | Default | Extra       |
+--------------------------+---------------+------+------+---------+-------------+
| TABLE_CATALOG            | varchar(512)  | YES  |      | NULL    |             |
| TABLE_SCHEMA             | varchar(64)   | YES  |      | NULL    |             |
| TABLE_NAME               | varchar(64)   | YES  |      | NULL    |             |
| COLUMN_NAME              | varchar(64)   | YES  |      | NULL    |             |
| ORDINAL_POSITION         | bigint(64)    | YES  |      | NULL    |             |
| COLUMN_DEFAULT           | text          | YES  |      | NULL    |             |
| IS_NULLABLE              | varchar(3)    | YES  |      | NULL    |             |
| DATA_TYPE                | varchar(64)   | YES  |      | NULL    |             |
| CHARACTER_MAXIMUM_LENGTH | bigint(21)    | YES  |      | NULL    |             |
| CHARACTER_OCTET_LENGTH   | bigint(21)    | YES  |      | NULL    |             |
| NUMERIC_PRECISION        | bigint(21)    | YES  |      | NULL    |             |
| NUMERIC_SCALE            | bigint(21)    | YES  |      | NULL    |             |
| DATETIME_PRECISION       | bigint(21)    | YES  |      | NULL    |             |
| CHARACTER_SET_NAME       | varchar(32)   | YES  |      | NULL    |             |
| COLLATION_NAME           | varchar(32)   | YES  |      | NULL    |             |
| COLUMN_TYPE              | text          | YES  |      | NULL    |             |
| COLUMN_KEY               | varchar(3)    | YES  |      | NULL    |             |
| EXTRA                    | varchar(30)   | YES  |      | NULL    |             |
| PRIVILEGES               | varchar(80)   | YES  |      | NULL    |             |
| COLUMN_COMMENT           | varchar(1024) | YES  |      | NULL    |             |
| GENERATION_EXPRESSION    | text          | NO   |      | NULL    |             |
+--------------------------+---------------+------+------+---------+-------------+
21 rows in set (0.00 sec)
```

创建表 `test.t1`，并查询 `COLUMNS` 表的信息：

```
CREATE TABLE test.t1 (a int);
SELECT * FROM COLUMNS WHERE table_schema='test' AND TABLE_NAME='t1'\G
```

输出结果如下：

```
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: test
              TABLE_NAME: t1
             COLUMN_NAME: a
        ORDINAL_POSITION: 1
          COLUMN_DEFAULT: NULL
             IS_NULLABLE: YES
               DATA_TYPE: int
CHARACTER_MAXIMUM_LENGTH: NULL
  CHARACTER_OCTET_LENGTH: NULL
       NUMERIC_PRECISION: 11
```

```
           NUMERIC_SCALE: 0
       DATETIME_PRECISION: NULL
       CHARACTER_SET_NAME: NULL
           COLLATION_NAME: NULL
              COLUMN_TYPE: int(11)
              COLUMN_KEY:
                    EXTRA:
               PRIVILEGES: select,insert,update,references
           COLUMN_COMMENT:
    GENERATION_EXPRESSION:
1 row in set (0.02 sec)
```

COLUMNS 表中列的含义如下：

- TABLE_CATALOG：包含列的表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：包含列的表所属的数据库的名称。
- TABLE_NAME：包含列的表的名称。
- COLUMN_NAME：列的名称。
- ORDINAL_POSITION：表中列的位置。
- COLUMN_DEFAULT：列的默认值。如果列的显式默认值为 NULL，或者列定义中不包含 default 子句，则此值为 NULL。
- IS_NULLABLE：列的可空性。如果列中可以存储空值，则该值为 YES，否则为 NO。
- DATA_TYPE：列的数据类型。
- CHARACTER_MAXIMUM_LENGTH：对于字符串列，以字符为单位的最大长度。
- CHARACTER_OCTET_LENGTH：对于字符串列，以字节为单位的最大长度。
- NUMERIC_PRECISION：对于数字列，为数字精度。
- NUMERIC_SCALE：对于数字列，为数字刻度。
- DATETIME_PRECISION：对于时间列，小数秒精度。
- CHARACTER_SET_NAME：对于字符串列，字符集名称。
- COLLATION_NAME：对于字符串列，排序规则名称。
- COLUMN_TYPE：列类型。
- COLUMN_KEY：该列是否被索引。具体显示如下：
  - 如果此值为空，则该列要么未被索引，要么被索引且是多列非唯一索引中的第二列。
  - 如果此值是 PRI，则该列是主键，或者是多列主键中的一列。
  - 如果此值是 UNI，则该列是唯一索引的第一列。
  - 如果此值是 MUL，则该列是非唯一索引的第一列，在该列中允许给定值的多次出现。
- EXTRA：关于给定列的任何附加信息。
- PRIVILEGES：当前用户对该列拥有的权限。目前在 TiDB 中，此值为定值，一直为 select,insert,update <-> ,references。
- COLUMN_COMMENT：列定义中包含的注释。
- GENERATION_EXPRESSION：对于生成的列，显示用于计算列值的表达式。对于未生成的列为空。

对应的 SHOW 语句如下：

```
SHOW COLUMNS FROM t1 FROM test;
```

输出结果如下：

```
+-------+---------+------+------+---------+--------------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+--------------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+--------------+
1 row in set (0.00 sec)
```

4.8.18.2.16 DATA_LOCK_WAITS

DATA_LOCK_WAITS 表展示了集群中所有TiKV 节点上当前正在发生的等锁情况，包括悲观锁的等锁情况和乐观事务被阻塞的信息。

```
USE information_schema;
DESC data_lock_waits;
```

```
+-----------------------+---------------------+------+------+---------+--------------+
| Field                 | Type                | Null | Key  | Default | Extra |
+-----------------------+---------------------+------+------+---------+--------------+
| KEY                   | text                | NO   |      | NULL    |       |
| KEY_INFO              | text                | YES  |      | NULL    |       |
| TRX_ID                | bigint(21) unsigned | NO   |      | NULL    |       |
| CURRENT_HOLDING_TRX_ID | bigint(21) unsigned | NO   |      | NULL    |       |
| SQL_DIGEST            | varchar(64)         | YES  |      | NULL    |       |
| SQL_DIGEST_TEXT       | text                | YES  |      | NULL    |       |
+-----------------------+---------------------+------+------+---------+--------------+
```

DATA_LOCK_WAITS 表中各列的字段含义如下：

- KEY：正在发生等锁的 key，以十六进制编码的形式显示。
- KEY_INFO：对 KEY 进行解读得出的一些详细信息，见KEY_INFO。
- TRX_ID：正在等锁的事务 ID，即 start_ts。
- CURRENT_HOLDING_TRX_ID：当前持有锁的事务 ID，即 start_ts。
- SQL_DIGEST：当前正在等锁的事务中被阻塞的 SQL 语句的 Digest。
- SQL_DIGEST_TEXT：当前正在等锁的事务中被阻塞的 SQL 语句的归一化形式，即去除了参数和格式的 SQL 语句。与 SQL_DIGEST 对应。

> 警告：
>
> - 仅拥有 PROCESS 权限的用户可以查询该表。
> - 由于实现限制，目前对于乐观事务被阻塞情况的 SQL_DIGEST 和 SQL_DIGEST_TEXT 字段为 null。如果需要知道导致阻塞的 SQL 语句，可以将此表与CLUSTER_TIDB_TRX 进行 JOIN 来获得对应事务的所有 SQL 语句。

- DATA_LOCK_WAITS 表中的信息是在查询时，从所有TiKV 节点实时获取的。目前，即使加上了 WHERE 查询条件，也无法避免对所有TiKV 节点都进行信息收集。如果集群规模很大、负载很高，查询该表有造成性能抖动的潜在风险，因此请根据实际情况使用。
- 来自不同TiKV 节点的信息不能保证是同一时间点的快照。
- SQL_DIGEST 列中的信息（SQL Digest）为 SQL 语句进行归一化后计算得到的哈希值。SQL_DIGEST_TEXT 列中的信息为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅Statement Summary Tables。

KEY_INFO

KEY_INFO 列中展示了对 KEY 列中所给出的 key 的详细信息，以JSON 格式给出。其包含的信息如下：

- "db_id"：该 key 所属的数据库（schema）的 ID。
- "db_name"：该 key 所属的数据库（schema）的名称。
- "table_id"：该 key 所属的表的ID。
- "table_name"：该 key 所属的表的名称。
- "partition_id"：该 key 所在的分区（partition）的ID。
- "partition_name"：该 key 所在的分区（partition）的名称。
- "handle_type"：该 row key （即储存一行数据的 key）的 handle 类型，其可能的值有：
  - "int"：handle 为 int 类型，即handle 为 row ID
  - "common"：非 int64 类型的handle，在启用clustered index 时非int 类型的主键会显示为此类型
  - "unknown"：当前暂不支持的 handle 类型
- "handle_value"：handle 的值。
- "index_id"：该 index key （即储存索引的 key）所属的 index ID。
- "index_name"：该 index key 所属的index 名称。
- "index_values"：该 index key 中的index value。

其中，不适用或当前无法查询到的信息会被省略。比如，row key 的信息中不会包含 index_id、index_name 和 index_values；index key 不会包含 handle_type 和 handle_value；非分区表不会显示 partition_id 和 partition_name；已经被删除掉的表中的 key 的信息无法获取 table_name、db_id、db_name、index_name 等 schema 信息，且无法区分是否为分区表。

注意：

如果一个 key 来自一张启用了分区的表，而在查询时，由于某些原因（例如，其所属的表已经被删除）导致无法查询其所属的 schema 信息，则其所属的分区的ID 可能会出现在 table_id 字段中。这是因为，TiDB 对不同分区的 key 的编码方式与对几张独立的表的 key 的编码方式一致，因而在缺失 schema 信息时无法确认该 key 属于一张未分区的表还是某张表的一个分区。

示例

```
select * from information_schema.data_lock_waits\G
```

```
*************************** 1. row ***************************
                  KEY: 748000000000000355F728000000000000001
             KEY_INFO: {"db_id":1,"db_name":"test","table_id":53,"table_name":"t","handle_type":
                 <-> "int","handle_value":"1"}
               TRX_ID: 426790594290122753
CURRENT_HOLDING_TRX_ID: 426790590082449409
            SQL_DIGEST: 38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821
       SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ?
1 row in set (0.01 sec)
```

以上查询结果显示，ID 为 426790594290122753 的事务在执行Digest 为 "38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e7<-> "、形如 update `t` set `v` = `v` + ? where `id` = ? 的语句的过程中，试图在 "7480000000000000355<-> F728000000000000001" 这个 key 上获取悲观锁，但是该 key 上的锁目前被ID 为 426790590082449409 的事务持有。

4.8.18.2.17 DDL_JOBS

DDL_JOBS 表为ADMIN SHOW DDL JOBS 命令提供了一个 INFORMATION_SCHEMA 接口。它提供了TiDB 集群中 DDL 操作的当前状态和简短历史记录。

```
USE information_schema;
DESC ddl_jobs;
```

```
+--------------+-------------+------+------+---------+--------------+
| Field        | Type        | Null | Key  | Default | Extra        |
+--------------+-------------+------+------+---------+--------------+
| JOB_ID       | bigint(21)  | YES  |      | NULL    |              |
| DB_NAME      | varchar(64) | YES  |      | NULL    |              |
| TABLE_NAME   | varchar(64) | YES  |      | NULL    |              |
| JOB_TYPE     | varchar(64) | YES  |      | NULL    |              |
| SCHEMA_STATE | varchar(64) | YES  |      | NULL    |              |
| SCHEMA_ID    | bigint(21)  | YES  |      | NULL    |              |
| TABLE_ID     | bigint(21)  | YES  |      | NULL    |              |
| ROW_COUNT    | bigint(21)  | YES  |      | NULL    |              |
| START_TIME   | datetime    | YES  |      | NULL    |              |
| END_TIME     | datetime    | YES  |      | NULL    |              |
| STATE        | varchar(64) | YES  |      | NULL    |              |
| QUERY        | varchar(64) | YES  |      | NULL    |              |
+--------------+-------------+------+------+---------+--------------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM ddl_jobs LIMIT 3\G
```

```
*************************** 1. row ***************************
      JOB_ID: 44
     DB_NAME: mysql
  TABLE_NAME: opt_rule_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 43
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.opt_rule_blacklist
        ( name char(100) NOT NULL
    );
*************************** 2. row ***************************
      JOB_ID: 42
     DB_NAME: mysql
  TABLE_NAME: expr_pushdown_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 41
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.expr_pushdown_blacklist
        ( name char(100) NOT NULL,
        store_type char(100) NOT NULL DEFAULT 'tikv,tiflash,tidb',
        reason varchar(200)
    );
*************************** 3. row ***************************
      JOB_ID: 40
     DB_NAME: mysql
  TABLE_NAME: stats_top_n
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 39
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:26
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
```

```
       QUERY: CREATE TABLE if not exists mysql.stats_top_n
        ( table_id bigint(64) NOT NULL,
        is_index tinyint(2) NOT NULL,
        hist_id bigint(64) NOT NULL,
        value longblob,
        count bigint(64) UNSIGNED NOT NULL,
        index tbl(table_id, is_index, hist_id)
    );
3 rows in set (0.01 sec)
```

4.8.18.2.18 DEADLOCKS

DEADLOCKS 表提供当前TiDB 节点上最近发生的若干次死锁错误的信息。

```
USE INFORMATION_SCHEMA;
DESC DEADLOCKS;
```

输出结果如下：

```
+-------------------------+---------------------+------+------+---------+----------------+
| Field                   | Type                | Null | Key  | Default | Extra          |
+-------------------------+---------------------+------+------+---------+----------------+
| DEADLOCK_ID             | bigint(21)          | NO   |      | NULL    |                |
| OCCUR_TIME              | timestamp(6)        | YES  |      | NULL    |                |
| RETRYABLE               | tinyint(1)          | NO   |      | NULL    |                |
| TRY_LOCK_TRX_ID         | bigint(21) unsigned | NO   |      | NULL    |                |
| CURRENT_SQL_DIGEST      | varchar(64)         | YES  |      | NULL    |                |
| CURRENT_SQL_DIGEST_TEXT | text                | YES  |      | NULL    |                |
| KEY                     | text                | YES  |      | NULL    |                |
| KEY_INFO                | text                | YES  |      | NULL    |                |
| TRX_HOLDING_LOCK        | bigint(21) unsigned | NO   |      | NULL    |                |
+-------------------------+---------------------+------+------+---------+----------------+
```

DEADLOCKS 表中需要用多行来表示同一个死锁事件，每行显示参与死锁的其中一个事务的信息。当该TiDB 节点记录了多次死锁错误时，需要按照 DEADLOCK_ID 列来区分，相同的 DEADLOCK_ID 表示同一个死锁事件。需要注意，DEADLOCK_ID 并不保证全局唯一，也不会持久化，因而其只能在同一个结果集里表示同一个死锁事件。

DEADLOCKS 表中各列的字段含义如下：

- DEADLOCK_ID：死锁事件的 ID。当表内存在多次死锁错误的信息时，需要使用该列来区分属于不同死锁错误的行。
- OCCUR_TIME：发生该次死锁错误的时间。
- RETRYABLE：该次死锁错误是否可重试。关于可重试的死锁错误的说明，参见可重试的死锁错误小节。
- TRY_LOCK_TRX_ID：试图上锁的事务 ID，即事务的 start_ts。
- CURRENT_SQL_DIGEST：试图上锁的事务中当前正在执行的 SQL 语句的 Digest。
- CURRENT_SQL_DIGEST_TEXT：试图上锁的事务中当前正在执行的 SQL 语句的归一化形式。

- KEY：该事务试图上锁、但是被阻塞的 key，以十六进制编码的形式显示。
- KEY_INFO：对 KEY 进行解读得出的一些详细信息，详见KEY_INFO。
- TRX_HOLDING_LOCK：该 key 上当前持锁并导致阻塞的事务ID，即事务的 start_ts。

要调整 DEADLOCKS 表中可以容纳的死锁事件数量，可通过 TiDB 配置文件中的pessimistic-txn.deadlock-history-capacity 配置项进行调整，默认容纳最近 10 次死锁错误的信息。

> 注意：
>
> - 仅拥有 PROCESS 权限的用户可以查询该表。
> - CURRENT_SQL_DIGEST 列中的信息(SQL Digest) 为 SQL 语句进行归一化后计算得到的哈希值。CURRENT_SQL_DIGEST_TEXT 列中的信息为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅Statement Summary Tables。

KEY_INFO

KEY_INFO 列中展示了对 KEY 列中所给出的 key 的详细信息，以JSON 格式给出。其包含的信息如下：

- "db_id"：该 key 所属的数据库(schema) 的 ID。
- "db_name"：该 key 所属的数据库(schema) 的名称。
- "table_id"：该 key 所属的表的ID。
- "table_name"：该 key 所属的表的名称。
- "partition_id"：该 key 所在的分区(partition) 的 ID。
- "partition_name"：该 key 所在的分区(partition) 的名称。
- "handle_type"：该 row key（即储存一行数据的 key）的 handle 类型，其可能的值有：
  - "int"：handle 为 int 类型，即handle 为 row ID
  - "common"：非 int64 类型的handle，在启用clustered index 时非int 类型的主键会显示为此类型
  - "unknown"：当前暂不支持的 handle 类型
- "handle_value"：handle 的值。
- "index_id"：该 index key（即储存索引的 key）所属的 index ID。
- "index_name"：该 index key 所属的index 名称。
- "index_values"：该 index key 中的index value。

其中，不适用或当前无法查询到的信息会被省略。比如，row key 的信息中不会包含 index_id、index_name 和 index_values；index key 不会包含 handle_type 和 handle_value；非分区表不会显示 partition_id 和 partition_name；已经被删除掉的表中的 key 的信息无法获取 table_name、db_id、db_name、index_name 等 schema 信息，且无法区分是否为分区表。

> 注意：

如果一个 key 来自一张启用了分区的表，而在查询时，由于某些原因（例如，其所属的表已经被删除）导致无法查询其所属的 schema 信息，则其所属的分区的ID 可能会出现在 `table_id` 字段中。这是因为，TiDB 对不同分区的 key 的编码方式与对几张独立的表的 key 的编码方式一致，因而在缺失 schema 信息时无法确认该 key 属于一张未分区的表还是某张表的一个分区。

可重试的死锁错误

注意：

DEADLOCKS 表中默认不收集可重试的死锁错误的信息。如果需要收集，可通过TiDB 配置文件中的`pessimistic-txn.deadlock-history-collect-retryable` 配置项进行调整。

当事务A 被另一个事务B 已经持有的锁阻塞，而事务B 直接或间接地被当前事务A 持有的锁阻塞，将会引发一个死锁错误。这里：

- 情况一：事务B 可能（直接或间接地）被事务A 开始后到被阻塞前这段时间内已经执行完成的语句产生的锁阻塞
- 情况二：事务B 也可能被事务A 目前正在执行的语句阻塞

对于情况一，TiDB 将会向事务A 的客户端报告死锁错误，并终止该事务；而对于情况二，事务A 当前正在执行的语句将在TiDB 内部被自动重试。例如，假设事务A 执行了如下语句：

```
UPDATE t SET v = v + 1 WHERE id = 1 OR id = 2;
```

事务B 则先后执行如下两条语句：

```
UPDATE t SET v = 4 WHERE id = 2;
UPDATE t SET v = 2 WHERE id = 1;
```

那么如果事务A 先后对 `id = 1` 和 `id = 2` 的两行分别上锁，且两个事务以如下时序运行：

1. 事务A 对 `id = 1` 的行上锁
2. 事务B 执行第一条语句并对 `id = 2` 的行上锁
3. 事务B 执行第二条语句试图对 `id = 1` 的行上锁，被事务A 阻塞
4. 事务A 试图对 `id = 2` 的行上锁，被B 阻塞，形成死锁

对于情况二，由于事务A 阻塞其它事务的语句也是当前正在执行的语句，因而可以解除当前语句所上的悲观锁（使得事务B 可以继续运行），并重试当前语句。TiDB 内部使用 key 的哈希值来判断是否属于这种情况。

当可重试的死锁发生时，内部自动重试并不会引起事务报错，因而对客户端透明，但是这种情况的频繁发生可能影响性能。当这种情况发生时，在 TiDB 的日志中可以观察到 `single statement deadlock, retry statement`字样的日志。

示例 1

假设有如下表定义和初始数据：

```
CREATE TABLE t (id int primary key, v int);
INSERT INTO t VALUES (1, 10), (2, 20);
```

使两个事务按如下顺序执行：

事务 1	事务 2	说明
UPDATE t SET v = 11 WHERE id = 1;		
	UPDATE t SET v = 21 WHERE id = 2;	
UPDATE t SET v = 12 WHERE id = 2;		事务 1 阻塞
	UPDATE t SET v = 22 WHERE id = 1;	事务 2 报出死锁错误

接下来，事务 2 将报出死锁错误。此时，查询 DEADLOCKS 表：

```
SELECT * FROM INFORMATION_SCHEMA.DEADLOCKS;
```

输出结果如下：

```
+--
    <->           +                          +           +                    +
    <->
| DEADLOCK_ID | OCCUR_TIME                  | RETRYABLE | TRY_LOCK_TRX_ID     | CURRENT_SQL_DIGEST
    <->                                              | CURRENT_SQL_DIGEST_TEXT                  |
    <-> KEY                                  | KEY_INFO
    <->
    <-> | TRX_HOLDING_LOCK       |
+--
    <->-----------+------------------------------+-----------+--------------------+
    <->
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <>  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <>  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
+--
    <->           +                          +           +                    +
    <->
```

该表中产生了两行数据， 两行的 DEADLOCK_ID 字段皆为 1， 表示这两行数据包含同一次死锁错误的信息。第一行显示ID 为 426812829645406216 的事务，在 "74800000000000000355F728000000000000002" 这个 key 上， 被 ID 为 "426812829645406217" 的事务阻塞了； 第二行则显示 ID 为 "426812829645406217" 的事务在 "74800000000000000355F728000000000000001" 这个 key 上被ID 为 426812829645406216 的事务阻塞了，构成了相互阻塞的状态，形成了死锁。

示例 2

假设查询 DEADLOCKS 表得到了如下结果集：

```
+--
    <->           +                              +           +                    +
    <->
| DEADLOCK_ID | OCCUR_TIME                  | RETRYABLE | TRY_LOCK_TRX_ID     | CURRENT_SQL_DIGEST
    <->                                              | CURRENT_SQL_DIGEST_TEXT                  |
    <-> KEY                                  | KEY_INFO
    <->
    <-> | TRX_HOLDING_LOCK       |
+--
    <->------------+------------------------------+-----------+--------------------+
    <->
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <->  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <->  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809412 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <->  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812832017809413 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809413 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <->  = ? ; | 74800000000000000355F728000000000000003 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"3"} | 426812832017809414 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809414 | 22230766411
    <-> edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    <->  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    <-> :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812832017809412 |
+--
    <->           +                              +           +                    +
    <->
```

以上查询结果中的 DEADLOCK_ID 列表明，前两行共同表示一次死锁错误的信息，两条事务相互等待构成了死锁；而后三行共同表示另一次死锁信息，三个事务循环等待构成了死锁。

CLUSTER_DEADLOCKS

CLUSTER_DEADLOCKS 表返回整个集群上每个TiDB 节点中最近发生的数次死锁错误的信息，即将每个节点上的 DEADLOCKS 表内的信息合并在一起。CLUSTER_DEADLOCKS 还包含额外的 INSTANCE 列展示所属节点的IP 地址和端口，用以区分不同的TiDB 节点。

需要注意的是，由于 DEADLOCK_ID 并不保证全局唯一，所以在 CLUSTER_DEADLOCKS 表的查询结果中，需要 INSTANCE 和 DEADLOCK_ID 两个字段共同区分结果集中的不同死锁错误的信息。

```
USE INFORMATION_SCHEMA;
DESC CLUSTER_DEADLOCKS;
```

```
+-------------------------+---------------------+------+------+---------+----------------+
| Field                   | Type                | Null | Key  | Default | Extra          |
+-------------------------+---------------------+------+------+---------+----------------+
| INSTANCE                | varchar(64)         | YES  |      | NULL    |                |
| DEADLOCK_ID             | bigint(21)          | NO   |      | NULL    |                |
| OCCUR_TIME              | timestamp(6)        | YES  |      | NULL    |                |
| RETRYABLE               | tinyint(1)          | NO   |      | NULL    |                |
| TRY_LOCK_TRX_ID         | bigint(21) unsigned | NO   |      | NULL    |                |
| CURRENT_SQL_DIGEST      | varchar(64)         | YES  |      | NULL    |                |
| CURRENT_SQL_DIGEST_TEXT | text                | YES  |      | NULL    |                |
| KEY                     | text                | YES  |      | NULL    |                |
| KEY_INFO                | text                | YES  |      | NULL    |                |
| TRX_HOLDING_LOCK        | bigint(21) unsigned | NO   |      | NULL    |                |
+-------------------------+---------------------+------+------+---------+----------------+
```

4.8.18.2.19 ENGINES

ENGINES 表提供了关于存储引擎的信息。从和 MySQL 兼容性上考虑，TiDB 会一直将InnoDB 描述为唯一支持的引擎。此外，ENGINES 表中其它列值也都是定值。

```
USE information_schema;
DESC engines;
```

```
+--------------+-------------+------+------+---------+----------------+
| Field        | Type        | Null | Key  | Default | Extra          |
+--------------+-------------+------+------+---------+----------------+
| ENGINE       | varchar(64) | YES  |      | NULL    |                |
| SUPPORT      | varchar(8)  | YES  |      | NULL    |                |
| COMMENT      | varchar(80) | YES  |      | NULL    |                |
| TRANSACTIONS | varchar(3)  | YES  |      | NULL    |                |
| XA           | varchar(3)  | YES  |      | NULL    |                |
| SAVEPOINTS   | varchar(3)  | YES  |      | NULL    |                |
+--------------+-------------+------+------+---------+----------------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM engines;
```

```
+--
    <-> ------+---------+-------------------------------------------------------------+--------------+------+-----
    <->
```

```
| ENGINE | SUPPORT | COMMENT                                                
```

字段解释：

- RULE：诊断规则名称，目前实现了以下规则：
  - config：配置一致性以及合理性检测。如果同一个配置在不同实例不一致，会生成 warning 诊断结果。
  - version：版本一致性检测。如果同一类型的实例版本不同，会生成 critical 诊断结果。
  - node-load：服务器负载检测。如果当前系统负载太高，会生成对应的 warning 诊断结果。
  - critical-error：系统各个模块定义了严重的错误，如果某一个严重错误在对应时间段内超过阈值，会生成 warning 诊断结果。
  - threshold-check：诊断系统会对一些关键指标进行阈值判断，如果超过阈值会生成对应的诊断信息。
- ITEM：每一个规则会对不同的项进行诊断，该字段表示对应规则下面的具体诊断项。
- TYPE：诊断的实例类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：诊断的具体实例地址。
- STATUS_ADDRESS：实例的 HTTP API 服务地址。
- VALUE：针对这个诊断项得到的值。
- REFERENCE：针对这个诊断项的参考值（阈值）。如果 VALUE 超过阈值，就会产生对应的诊断信息。
- SEVERITY：严重程度，取值为 warning 或 critical。
- DETAILS：诊断的详细信息，可能包含进一步调查的 SQL 或文档链接。

诊断示例

对当前时间的集群进行诊断。

```
SELECT * FROM information_schema.inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE       | config
ITEM       | log.slow-threshold
TYPE       | tidb
INSTANCE   | 172.16.5.40:4000
VALUE      | 0
REFERENCE | not 0
SEVERITY   | warning
DETAILS    | slow-threshold = 0 will record every query to slow log, it may affect performance
***************************[ 2. row ]***************************
RULE       | version
ITEM       | git_hash
TYPE       | tidb
INSTANCE  |
VALUE      | inconsistent
REFERENCE | consistent
SEVERITY  | critical
DETAILS    | the cluster has 2 different tidb version, execute the sql to see more detail: select
    <—> * from information_schema.cluster_info where type='tidb'
```

```
***************************[ 3. row ]***************************
RULE       | threshold-check
ITEM       | storage-write-duration
TYPE       | tikv
INSTANCE   | 172.16.5.40:23151
VALUE      | 130.417
REFERENCE  | < 0.100
SEVERITY   | warning
DETAILS    | max duration of 172.16.5.40:23151 tikv storage-write-duration was too slow
***************************[ 4. row ]***************************
RULE       | threshold-check
ITEM       | rocksdb-write-duration
TYPE       | tikv
INSTANCE   | 172.16.5.40:20151
VALUE      | 108.105
REFERENCE  | < 0.100
SEVERITY   | warning
DETAILS    | max duration of 172.16.5.40:20151 tikv rocksdb-write-duration was too slow
```

上述诊断结果发现了以下几个问题：

- 第一行表示TiDB 的 `log.slow-threshold` 配置值为 0，可能会影响性能。
- 第二行表示集群中有 2 个不同的TiDB 版本
- 第三、四行表示TiKV 的写入延迟太大，期望时间是不超过 0.1s，但实际值远超预期。

诊断集群在时间段“2020-03-26 00:03:00”，“2020-03-26 00:08:00”的问题。指定时间范围需要使用`/*+ time_range <-> ()*/` 的 SQL Hint，参考下面的查询示例：

```
select /*+ time_range("2020-03-26 00:03:00", "2020-03-26 00:08:00") */ * from information_schema.
    <-> inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE       | critical-error
ITEM       | server-down
TYPE       | tidb
INSTANCE   | 172.16.5.40:4009
VALUE      |
REFERENCE  |
SEVERITY   | critical
DETAILS    | tidb 172.16.5.40:4009 restarted at time '2020/03/26 00:05:45.670'
***************************[ 2. row ]***************************
RULE       | threshold-check
ITEM       | get-token-duration
TYPE       | tidb
INSTANCE   | 172.16.5.40:10089
VALUE      | 0.234
```

```
REFERENCE | < 0.001
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:10089 tidb get-token-duration is too slow
```

上面的诊断结果发现了以下问题：

- 第一行表示 172.16.5.40:4009 TiDB 实例在 2020/03/26 00:05:45.670 发生了重启。
- 第二行表示 172.16.5.40:10089 TiDB 实例的最大的 get-token-duration 时间为 0.234s，期望时间是小于 0.001s。

也可以指定条件，比如只查询 critical 严重级别的诊断结果：

```
select * from information_schema.inspection_result where severity='critical';
```

只查询 critical-error 规则的诊断结果:

```
select * from information_schema.inspection_result where rule='critical-error';
```

诊断规则介绍

诊断模块内部包含一系列的规则，这些规则会通过查询已有的监控表和集群信息表，对结果和阈值进行对比。如果结果超过阈值将生成 warning 或 critical 的结果，并在 details 列中提供相应信息。

可以通过查询 inspection_rules 系统表查询已有的诊断规则:

```
select * from information_schema.inspection_rules where type='inspection';
```

```
+-----------------+------------+---------------+
| NAME            | TYPE       | COMMENT       |
+-----------------+------------+---------------+
| config          | inspection |               |
| version         | inspection |               |
| node-load       | inspection |               |
| critical-error  | inspection |               |
| threshold-check | inspection |               |
+-----------------+------------+---------------+
```

config 诊断规则

config 诊断规则通过查询 CLUSTER_CONFIG 系统表，执行以下两个诊断规则：

- 检测相同组件的配置值是否一致，并非所有配置项都会有一致性检查，下面是一致性检查的白名单：

```
// TiDB 配置一致性检查白名单
port
status.status-port
host
path
advertise-address
```

```
status.status-port
log.file.filename
log.slow-query-file
tmp-storage-path

// PD 配置一致性检查白名单
advertise-client-urls
advertise-peer-urls
client-urls
data-dir
log-file
log.file.filename
metric.job
name
peer-urls

// TiKV 配置一致性检查白名单
server.addr
server.advertise-addr
server.status-addr
log-file
raftstore.raftdb-path
storage.data-dir
storage.block-cache.capacity
```

- 检测以下配置项的值是否符合预期。

组件	配置项	预期值
TiDB	log.slow-threshold	大于 0
TiKV	raftstore.sync-log	true

version 诊断规则

version 诊断规则通过查询 CLUSTER_INFO 系统表，检测相同组件的版本hash 是否一致。示例如下：

```
SELECT * FROM information_schema.inspection_result WHERE rule='version'\G
```

```
***************************[ 1. row ]***************************
RULE       | version
ITEM       | git_hash
TYPE       | tidb
INSTANCE   |
VALUE      | inconsistent
REFERENCE  | consistent
SEVERITY   | critical
```

```
DETAILS   | the cluster has 2 different tidb versions, execute the sql to see more detail: SELECT
    <-> * FROM information_schema.cluster_info WHERE type='tidb'
```

`critical-error` 诊断规则

`critical-error` 诊断规则执行以下两个诊断规则：

- 通过查询metrics schema 数据库中相关的监控系统表，检测集群是否有出现以下比较严重的错误：

组件	错误名字	相关监控表	错误说明
TiDB	panic-count	tidb_panic_TcioDuBnt_total_count	出现 panic 错误
TiDB	binlog-error	tidb_binlog_TeiDrrBor_total_count	写 binlog 时出现的错误
TiKV	critical-error	tikv_critical_TeiKrrVor_total_coun	的 critical error
TiKV	scheduler-is-busy	tikv_scheduTlieKrV_is_busy_total_count	的 sched-uler 太忙，会导致 TiKV 临时不可用
TiKV	coprocessor-is-busy	tikv_coprocTeisKsVor_busy_total_count	的 copro-cessor 太忙
TiKV	channel-is-full	tikv_channeTli_KfuVll_total_count	出现 chan-nel full 的错误

组件	错误名字	相关监控表	错误说明
TiKV	tikv_engine_write_stall	tikv_engine_write_stall	TiKV 出现写入 stall 的错误

- 通过查询 `metrics_schema.up` 监控表和 `CLUSTER_LOG` 系统表，检查是否有组件发生重启。

`threshold-check` 诊断规则

`threshold-check` 诊断规则通过查询metrics schema 数据库中相关的监控系统表，检测集群中以下指标是否超出阈值：

组件	监控指标	相关监控表	预期值	说明
TiDB	tso-duration	pd_tso_wait_duration	小于 50 ms	获取事务 TSO 时间戳的等待耗时
TiDB	get-token-duration	tidb_get_token_duration	小于 1 ms	查询获取 token 的耗时，相关的 TiDB 配置参数是 token-limit
TiDB	load-schema-duration	tidb_load_schema_duration	小于 1 s	TiDB 更新表元信息的耗时

组件	监控指标	相关监控表	预期值	说明
TiKV	scheduler-cmd-duration	tikv_scheduler_command_duration	小于 0.1 s	执行 KV cmd 请求的耗时
TiKV	handle-snapshot-duration	tikv_handle_snapshot_duration	小于 30 s	处理 snap-shot 的耗时
TiKV	storage-write-duration	tikv_storage_async_request_duration	小于 0.1 s	写入的延迟
TiKV	storage-snapshot-duration	tikv_storage_async_request_duration	小于 50 ms	获取 snap-shot 的耗时
TiKV	rocksdb-write-duration	tikv_engine_write_duration	小于 100 ms	RocksDB 的写入延迟
TiKV	rocksdb-get-duration	tikv_engine_max_get_duration	小于 50 ms	RocksDB 的读取延迟
TiKV	rocksdb-seek-duration	tikv_engine_max_seek_duration	小于 50 ms	RocksDB 执行 seek 的延迟
TiKV	scheduler-pending-cmd-coun	tikv_scheduler_pending_commands	小于 1000	KV 中被阻塞的命令数量

组件	监控指标	相关监控表	预期值	说明
TiKV	index-block-cache-hit	tikv_block_index_cache_hit	大于 0.95	中 index block 缓存的命中率
TiKV	filter-block-cache-hit	tikv_block_filter_cache_hit	大于 0.95	中 filter block 缓存的命中率
TiKV	data-block-cache-hit	tikv_block_data_cache_hit	大于 0.80	中 data block 缓存的命中率
TiKV	leader-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 leader score 是否均衡，期望实例间的差异小于 5%

组件	监控指标	相关监控表	预期值	说明
TiKV	region-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 Region score 是否均衡，期望实例间的差异小于 5%
TiKV	store-available-balance	pd_scheduler_store_status	小于 0.2	检测各个 TiKV 实例的存储可用空间大小是否均衡，期望实例间的差异小于 20%

组件	监控指标	相关监控表	预期值	说明
TiKV	region-count	pd_scheduler_store_status	小于 20000	检测各个 TiKV 实例的 Region 数量，期望单个实例的 Region 数量小于 20000
PD	region-health	pd_region_health	小于 100	检测集群中处于调度中间状态的 Region 数量，期望总数小于 100

另外还会检测TiKV 实例的以下thread cpu usage 是否过高:

- scheduler-worker-cpu
- coprocessor-normal-cpu
- coprocessor-high-cpu
- coprocessor-low-cpu
- grpc-cpu
- raftstore-cpu
- apply-cpu
- storage-readpool-normal-cpu
- storage-readpool-high-cpu
- storage-readpool-low-cpu
- split-check-cpu

TiDB 内置的诊断规则还在不断的完善改进中，如果你也想到了一些诊断规则，非常欢迎在tidb repository 下提 PR 或 Issue。

4.8.18.2.21 INSPECTION_RULES

INSPECTION_RULES 表提供在检查结果中运行哪些诊断测试的信息，示例用法参见inspection-result 表。

```
USE information_schema;
DESC inspection_rules;
```

```
+---------+--------------+------+------+---------+-------------+
| Field   | Type         | Null | Key  | Default | Extra       |
+---------+--------------+------+------+---------+-------------+
| NAME    | varchar(64)  | YES  |      | NULL    |             |
| TYPE    | varchar(64)  | YES  |      | NULL    |             |
| COMMENT | varchar(256) | YES  |      | NULL    |             |
+---------+--------------+------+------+---------+-------------+
3 rows in set (0.00 sec)
```

```
SELECT * FROM inspection_rules;
```

```
+-----------------+------------+----------------+
| NAME            | TYPE       | COMMENT        |
+-----------------+------------+----------------+
| config          | inspection |                |
| version         | inspection |                |
| node-load       | inspection |                |
| critical-error  | inspection |                |
| threshold-check | inspection |                |
| ddl             | summary    |                |
| gc              | summary    |                |
| pd              | summary    |                |
| query-summary   | summary    |                |
| raftstore       | summary    |                |
| read-link       | summary    |                |
| rocksdb         | summary    |                |
| stats           | summary    |                |
| wait-events     | summary    |                |
| write-link      | summary    |                |
+-----------------+------------+----------------+
15 rows in set (0.00 sec)
```

4.8.18.2.22 INSPECTION_SUMMARY

在部分场景下，用户只需要关注特定链路或模块的监控汇总。例如当前 Coprocessor 配置的线程池为 8，如果 Coprocessor 的 CPU 使用率达到了 750%，就可以确定存在风险，或者可能提前成为瓶颈。但是部分监控会因为

用户的 workload 不同而差异较大，所以难以定义确定的阈值。排查这部分场景的问题也非常重要，所以TiDB 提供了 inspection_summary 来进行链路汇总。

诊断汇总表 information_schema.inspection_summary 的表结构如下：

```
USE information_schema;
DESC inspection_summary;
```

```
+--------------+--------------+------+------+---------+----------------+
| Field        | Type         | Null | Key  | Default | Extra          |
+--------------+--------------+------+------+---------+----------------+
| RULE         | varchar(64)  | YES  |      | NULL    |                |
| INSTANCE     | varchar(64)  | YES  |      | NULL    |                |
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |                |
| LABEL        | varchar(64)  | YES  |      | NULL    |                |
| QUANTILE     | double       | YES  |      | NULL    |                |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |                |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |                |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |                |
| COMMENT      | varchar(256) | YES  |      | NULL    |                |
+--------------+--------------+------+------+---------+----------------+
9 rows in set (0.00 sec)
```

字段解释：

- RULE：汇总规则。由于规则在持续添加，最新的规则列表可以通过 select * from inspection_rules <⟶ where type='summary' 查询。
- INSTANCE：监控的具体实例。
- METRICS_NAME：监控表的名字。
- QUANTILE：对于包含 QUANTILE 的监控表有效，可以通过谓词下推指定多个百分位，例如 select * <⟶ from inspection_summary where rule='ddl' and quantile in (0.80, 0.90, 0.99, 0.999) 来汇总 DDL 相关监控，查询百分位为 80/90/99/999 的结果。AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示聚合的平均值、最小值、最大值。
- COMMENT：对应监控的解释。

> 注意：
>
> 由于汇总所有结果有一定开销，建议在 SQL 的谓词中显示指定的 rule 以减小开销。例如 select * from inspection_summary where rule in ('read-link', 'ddl') 会汇总读链路和 DDL 相关的监控。

使用示例:

诊断结果表和诊断监控汇总表都可以通过 hint 的方式指定诊断的时间范围，例如 select /*+ time_range <⟶ ('2020-03-07 12:00:00','2020-03-07 13:00:00')*/* from inspection_summary 是对 2020-03-07 12:00:00 -

2020-03-07 13:00:00 时间段的监控汇总。和监控汇总表一样，inspection_summary 系统表也可以通过对比两个不同时间段的数据，快速发现差异较大的监控项。

以下为一个例子，对比以下两个时间段，读系统链路的监控项:

- (2020-01-16 16:00:54.933, 2020-01-16 16:10:54.933)
- (2020-01-16 16:10:54.933, 2020-01-16 16:20:54.933)

```
SELECT
  t1.avg_value / t2.avg_value AS ratio,
  t1.*,
  t2.*
FROM
  (
    SELECT
      /*+ time_range("2020-01-16 16:00:54.933", "2020-01-16 16:10:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t1
  JOIN
  (
    SELECT
      /*+ time_range("2020-01-16 16:10:54.933", "2020-01-16 16:20:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t2
  ON t1.metrics_name = t2.metrics_name
  and t1.instance = t2.instance
  and t1.label = t2.label
ORDER BY
  ratio DESC;
```

4.8.18.2.23 KEY_COLUMN_USAGE

KEY_COLUMN_USAGE 表描述了列的键约束，比如主键约束。

```
USE information_schema;
DESC key_column_usage;
```

```
+------------------------------+--------------+------+------+---------+----------------+
| Field                        | Type         | Null | Key  | Default | Extra          |
+------------------------------+--------------+------+------+---------+----------------+
| CONSTRAINT_CATALOG           | varchar(512) | NO   |      | NULL    |                |
| CONSTRAINT_SCHEMA            | varchar(64)  | NO   |      | NULL    |                |
| CONSTRAINT_NAME              | varchar(64)  | NO   |      | NULL    |                |
| TABLE_CATALOG                | varchar(512) | NO   |      | NULL    |                |
| TABLE_SCHEMA                 | varchar(64)  | NO   |      | NULL    |                |
| TABLE_NAME                   | varchar(64)  | NO   |      | NULL    |                |
```

```
| COLUMN_NAME                    | varchar(64)   | NO   |       | NULL    |       |
| ORDINAL_POSITION               | bigint(10)    | NO   |       | NULL    |       |
| POSITION_IN_UNIQUE_CONSTRAINT  | bigint(10)    | YES  |       | NULL    |       |
| REFERENCED_TABLE_SCHEMA        | varchar(64)   | YES  |       | NULL    |       |
| REFERENCED_TABLE_NAME          | varchar(64)   | YES  |       | NULL    |       |
| REFERENCED_COLUMN_NAME         | varchar(64)   | YES  |       | NULL    |       |
+--------------------------------+---------------+------+-------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM key_column_usage WHERE table_schema='mysql' and table_name='user';
```

```
*************************** 1. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: Host
             ORDINAL_POSITION:    1
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
       REFERENCED_TABLE_NAME:  NULL
       REFERENCED_COLUMN_NAME: NULL
*************************** 2. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: User
             ORDINAL_POSITION:    2
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
       REFERENCED_TABLE_NAME:  NULL
       REFERENCED_COLUMN_NAME: NULL
2 rows in set (0.00 sec)
```

KEY_COLUMN_USAGE 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。该值始终为 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束名称。
- TABLE_CATALOG：表所属目录的名称。该值始终为 def。

- TABLE_SCHEMA：表所属的架构数据库的名称。
- TABLE_NAME：具有约束的表的名称。
- COLUMN_NAME：具有约束的列的名称。
- ORDINAL_POSITION：列在约束中的位置，而不是列在表中的位置。列位置从 1 开始编号。
- POSITION_IN_UNIQUE_CONSTRAINT：唯一约束和主键约束为空。对于外键约束，此列是被引用的表的键的序号位置。
- REFERENCED_TABLE_SCHEMA：约束引用的数据库的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_TABLE_NAME：约束引用的表的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_COLUMN_NAME：约束引用的列的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。

4.8.18.2.24 MEMORY_USAGE

MEMORY_USAGE 表描述了TiDB 实例当前的内存使用情况。

```
USE information_schema;
DESC memory_usage;
```

```
+--------------------+-------------+------+------+---------+----------------+
| Field              | Type        | Null | Key  | Default | Extra          |
+--------------------+-------------+------+------+---------+----------------+
| MEMORY_TOTAL       | bigint(21)  | NO   |      | NULL    |                |
| MEMORY_LIMIT       | bigint(21)  | NO   |      | NULL    |                |
| MEMORY_CURRENT     | bigint(21)  | NO   |      | NULL    |                |
| MEMORY_MAX_USED    | bigint(21)  | NO   |      | NULL    |                |
| CURRENT_OPS        | varchar(50) | YES  |      | NULL    |                |
| SESSION_KILL_LAST  | datetime    | YES  |      | NULL    |                |
| SESSION_KILL_TOTAL | bigint(21)  | NO   |      | NULL    |                |
| GC_LAST            | datetime    | YES  |      | NULL    |                |
| GC_TOTAL           | bigint(21)  | NO   |      | NULL    |                |
| DISK_USAGE         | bigint(21)  | NO   |      | NULL    |                |
| QUERY_FORCE_DISK   | bigint(21)  | NO   |      | NULL    |                |
+--------------------+-------------+------+------+---------+----------------+
11 rows in set (0.000 sec)
```

```
SELECT * FROM information_schema.memory_usage;
```

```
+--
    <-> ------------+-------------+----------------+-----------------+-------------+----------------------+------
    <->
| MEMORY_TOTAL | MEMORY_LIMIT | MEMORY_CURRENT | MEMORY_MAX_USED | CURRENT_OPS |
    <-> SESSION_KILL_LAST    | SESSION_KILL_TOTAL | GC_LAST             | GC_TOTAL | DISK_USAGE |
    <-> QUERY_FORCE_DISK |
```

```
+--
    <-> ------------+--------------+----------------+-----------------+-------------+---------------------+------
    <->
|  33674170368 |  10737418240 |      5097644032 |      10826604544 | NULL         | 2022-10-17
    <-> 22:47:47 |                    1 | 2022-10-17 22:47:47 |        20 |           0 |
    <->                0 |
+--
    <-> ------------+--------------+----------------+-----------------+-------------+---------------------+------
    <->
2 rows in set (0.002 sec)
```

MEMORY_USAGE 表中列的含义如下：

- MEMORY_TOTAL：TiDB 的可用内存总量，单位为byte。
- MEMORY_LIMIT：TiDB 的内存使用限制，单位为byte。其值与系统变量tidb_server_memory_limit 的值相同。
- MEMORY_CURRENT：TiDB 当前的内存使用量，单位为byte。
- MEMORY_MAX_USED：从 TiDB 启动到当前的最大内存使用量，单位为byte。
- CURRENT_OPS：“shrinking” | null。“shrinking”表示TiDB 正在执行收缩内存用量的操作。
- SESSION_KILL_LAST：上一次终止会话的时间戳。
- SESSION_KILL_TOTAL：从 TiDB 启动到当前累计终止会话的次数。
- GC_LAST：上一次由内存使用触发 Golang GC 的时间戳。
- GC_TOTAL：从TiDB 启动到当前累计由内存使用触发Golang GC 的次数。
- DISK_USAGE：当前数据落盘的硬盘使用量，单位为 byte。
- QUERY_FORCE_DISK：从 TiDB 启动到当前累计的落盘次数。

4.8.18.2.25 MEMORY_USAGE_OPS_HISTORY

MEMORY_USAGE_OPS_HISTORY 表描述了TiDB 实例内存相关的历史操作和执行依据。

```
USE information_schema;
DESC memory_usage_ops_history;
```

```
+----------------+---------------------+------+------+---------+-------------+
| Field          | Type                | Null | Key  | Default | Extra       |
+----------------+---------------------+------+------+---------+-------------+
| TIME           | datetime            | NO   |      | NULL    |             |
| OPS            | varchar(20)         | NO   |      | NULL    |             |
| MEMORY_LIMIT   | bigint(21)          | NO   |      | NULL    |             |
| MEMORY_CURRENT | bigint(21)          | NO   |      | NULL    |             |
| PROCESSID      | bigint(21) unsigned | YES  |      | NULL    |             |
| MEM            | bigint(21) unsigned | YES  |      | NULL    |             |
| DISK           | bigint(21) unsigned | YES  |      | NULL    |             |
| CLIENT         | varchar(64)         | YES  |      | NULL    |             |
| DB             | varchar(64)         | YES  |      | NULL    |             |
| USER           | varchar(16)         | YES  |      | NULL    |             |
```

```
| SQL_DIGEST      | varchar(64)          | YES  |      | NULL    |       |
| SQL_TEXT        | varchar(256)         | YES  |      | NULL    |       |
+-----------------+----------------------+------+------+---------+-------+
12 rows in set (0.000 sec)
```

```
SELECT * FROM information_schema.memory_usage_ops_history;
```

```
+--
    <-> ------------------+-------------+--------------+----------------+---------------------+------------+----
   <->
| TIME                | OPS          | MEMORY_LIMIT | MEMORY_CURRENT | PROCESSID           | MEM
   <->        | DISK | CLIENT           | DB    | USER | SQL_DIGEST
   <->                                                        | SQL_TEXT
   <->                                                             |
+--
    <-> ------------------+-------------+--------------+----------------+---------------------+------------+----
    <->
| 2022-10-17 22:46:25 | SessionKill |  10737418240 |     10880237568 | 6718275530455515543 |
    <-> 7905028235 |    0 | 127.0.0.1:34394 | test | root | 146
    <-> b3d812852663a20635fbcf02be01688f52c8d433dafec0d496a14f0b59df6 | desc analyze select *
    <-> from t t1 join t t2 on t1.a=t2.a order by t1.a |
+--
    <-> ------------------+-------------+--------------+----------------+---------------------+------------+----
    <->
2 rows in set (0.002 sec)
```

MEMORY_USAGE_OPS_HISTORY 表中列的含义如下：

- TIME：终止会话的时间戳。
- OPS：“SessionKill”
- MEMORY_LIMIT：TiDB 终止会话时的内存使用限制，单位为byte。其值和系统变量tidb_server_memory_limit 相同。
- MEMORY_CURRENT：TiDB 当前的内存使用量，单位为byte。
- PROCESSID：被终止会话的客户连接 ID。
- MEM：被终止的会话已使用的内存使用量，单位是 byte。
- DISK：被终止的会话已使用的硬盘使用量，单位是 byte。
- CLIENT：被终止的会话的客户连接的地址。
- DB：被终止的会话所连接的数据库名。
- USER：被终止的会话的用户名。
- SQL_DIGEST：被终止的会话正在执行 SQL 语句的digest。
- SQL_TEXT: 被终止的会话正在执行的 SQL 语句。

4.8.18.2.26 METRICS_SUMMARY

由于TiDB 集群的监控指标数量较多，为了方便用户从众多监控中找出异常的监控项，TiDB 4.0 提供了以下监控汇总表：

- information_schema.metrics_summary
- information_schema.metrics_summary_by_label

这两张表用于汇总所有监控数据， 用户排查各个监控指标会更有效率。其中 information_schema.metrics_summary_by_label 会对不同的label 进行区分统计。

```
USE information_schema;
DESC metrics_summary;
```

```
+--------------+--------------+------+------+---------+--------------+
| Field        | Type         | Null | Key  | Default | Extra        |
+--------------+--------------+------+------+---------+--------------+
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |              |
| QUANTILE     | double       | YES  |      | NULL    |              |
| SUM_VALUE    | double(22,6) | YES  |      | NULL    |              |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |              |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |              |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |              |
| COMMENT      | varchar(256) | YES  |      | NULL    |              |
+--------------+--------------+------+------+---------+--------------+
7 rows in set (0.00 sec)
```

字段解释：

- METRICS_NAME：监控表名。
- QUANTILE：百分位。可以通过 SQL 语句指定 QUANTILE，例如：
  - select * from metrics_summary where quantile=0.99 指定查看百分位为 0.99 的数据。
  - select * from metrics_summary where quantile in (0.80, 0.90, 0.99, 0.999) 同时查看百分位为 0.80, 0.90, 0.99, 0.999 的数据。
- SUM_VALUE、AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示总和、平均值、最小值、最大值。
- COMMENT：对应监控的解释。

具体查询示例：

查询 '2020-03-08 13:23:00', '2020-03-08 13:33:00' 时间范围内TiDB 集群中平均耗时最高的三组监控项。可直接查询 information_schema.metrics_summary 表，并通过 /*+ time_range()*/ 这个hint 来指定时间范围，构造的 SQL 语句如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary
WHERE metrics_name LIKE 'tidb%duration'
AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 3\G
```

```
***************************[ 1. row ]***************************
METRICS_NAME | tidb_get_token_duration
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    <-> concurrency limit is reached(second)
***************************[ 2. row ]***************************
METRICS_NAME | tidb_query_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.269079
AVG_VALUE    | 0.007272
MIN_VALUE    | 0.000667
MAX_VALUE    | 0.01554
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
METRICS_NAME | tidb_kv_request_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.170232
AVG_VALUE    | 0.004601
MIN_VALUE    | 0.000975
MAX_VALUE    | 0.013
COMMENT      | The quantile of kv requests durations by store
```

类似的，查询 `metrics_summary_by_label` 监控汇总表示例如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary_by_label
WHERE metrics_name LIKE 'tidb%duration'
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 10\G
```

```
***************************[ 1. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_get_token_duration
LABEL        |
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
```

```
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    <-> concurrency limit is reached(second)
***************************[ 2. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Select
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Rollback
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
```

前文提到 `metrics_summary_by_label` 表结构相对于 `metrics_summary` 多了一列 `LABEL`。以上面查询结果的第 2、3 行分别表示 `tidb_query_duration` 的 `Select` 和 `Rollback` 类型的语句平均耗时非常高。

除以上示例之外，监控汇总表可以通过对比两个时间段的全链路监控，迅速找出监控数据中变化最大的模块，快速定位瓶颈。以下示例对比两个时间段的所有监控（其中t1 为 baseline），并按照差别最大的监控排序：

- 时间段t1：("2020-03-03 17:08:00", "2020-03-03 17:11:00")
- 时间段t2：("2020-03-03 17:18:00", "2020-03-03 17:21:00")

对两个时间段的监控按照 `METRICS_NAME` 进行join，并按照差异值大小排序。其中 `TIME_RANGE` 是用于指定查询时间的hint。

```
SELECT GREATEST(t1.avg_value,t2.avg_value)/LEAST(t1.avg_value,
         t2.avg_value) AS ratio,
         t1.metrics_name,
         t1.avg_value as t1_avg_value,
         t2.avg_value as t2_avg_value,
         t2.comment
FROM
    (SELECT /*+ time_range("2020-03-03 17:08:00", "2020-03-03 17:11:00")*/ *
    FROM information_schema.metrics_summary ) t1
JOIN
    (SELECT /*+ time_range("2020-03-03 17:18:00", "2020-03-03 17:21:00")*/ *
```

```
    FROM information_schema.metrics_summary ) t2
    ON t1.metrics_name = t2.metrics_name
ORDER BY ratio DESC LIMIT 10;
```

```
+--
    <-->--------------+-----------------------------------------+---------------+-----------------+----------
    <-->
| ratio          | metrics_name                              | t1_avg_value    | t2_avg_value      |
    <--> comment
    <-->                                                                                          |
+--
    <-->--------------+-----------------------------------------+---------------+-----------------+----------
    <-->
| 5865.59537065  | tidb_slow_query_cop_process_total_time    |       0.016333 |        95.804724 |
    <-->  The total time of TiDB slow query statistics with slow query total cop process time(
    <--> second) |
| 3648.74109023  | tidb_distsql_partial_scan_key_total_num  |    10865.666667 |  39646004.4394    |
    <-->  The total num of distsql partial scan key numbers
    <-->                                          |
|  267.002351165 | tidb_slow_query_cop_wait_total_time       |       0.003333 |         0.890008 |
    <-->  The total time of TiDB slow query statistics with slow query total cop wait time(second)
|  192.43267836  | tikv_cop_total_response_total_size        | 2515333.66667  | 484032394.445     |
    <-->
    <-->
    <--> |
|  192.43267836  | tikv_cop_total_response_size_per_seconds |    41922.227778 |   8067206.57408  |
    <-->
    <-->
    <--> |
|  152.780296296 | tidb_distsql_scan_key_total_num           |    5304.333333 |     810397.618317 |
    <--> The total  num of distsql scan numbers
    <-->                                                    |
|  126.042290167 | tidb_distsql_execution_total_time         |       0.421622 |         53.142143 |
    <--> The total  time of distsql execution(second)
    <-->                                                 |
|  105.164020657 | tikv_cop_scan_details                     |     134.450733 |      14139.379665 |
    <-->
    <-->
    <--> |
|  105.164020657 | tikv_cop_scan_details_total               |    8067.043981 |     848362.77991  |
    <-->
    <-->
    <--> |
|  101.635495394 | tikv_cop_scan_keys_num                    |    1070.875    |     108838.91113  |
```

```
    <->
    <->
    <-> |
+--
    <->--------------+-------------------------------------------+-----------------+                  +-----------
    <->
```

上面查询结果表示：

- t2 时间段内的 tidb_slow_query_cop_process_total_time（TiDB 慢查询中的 cop process 耗时）比 t1 时间段高了 5865 倍。
- t2 时间段内的 tidb_distsql_partial_scan_key_total_num（TiDB 的 distsql 请求扫描 key 的数量）比 t1 时间段高了 3648 倍。
- t2 时间段内的 tidb_slow_query_cop_wait_total_time（TiDB 慢查询中的cop 请求排队等待的耗时）比t1 时间段高了 267 倍。
- t2 时间段内的 tikv_cop_total_response_size（TiKV 的cop 请求结果的大小）比 t1 时间段高了 192 倍。
- t2 时间段内的 tikv_cop_scan_details（TiKV 的cop 请求的 scan）比 t1 时间段高了 105 倍。

综上，可以马上知道 t2 时间段的cop 请求要比t2 时间段高很多，导致 TiKV 的 Coprocessor 过载，出现了 cop task 等待，可以猜测可能是t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在t1 ~ t2 整个时间段内都在跑 go-ycsb 的压测，然后在t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了出现很多的cop 请求。

4.8.18.2.27 METRICS_TABLES

METRICS_TABLES 表为METRICS_SCHEMA 数据库中的每个视图提供 PromQL（Prometheus 查询语言）定义。

```
USE INFORMATION_SCHEMA;
DESC METRICS_TABLES;
```

输出结果如下：

```
+------------+--------------+------+------+---------+--------------+
| Field      | Type         | Null | Key  | Default | Extra        |
+------------+--------------+------+------+---------+--------------+
| TABLE_NAME | varchar(64)  | YES  |      | NULL    |              |
| PROMQL     | varchar(64)  | YES  |      | NULL    |              |
| LABELS     | varchar(64)  | YES  |      | NULL    |              |
| QUANTILE   | double       | YES  |      | NULL    |              |
| COMMENT    | varchar(256) | YES  |      | NULL    |              |
+------------+--------------+------+------+---------+--------------+
```

表 METRICS_TABLES 的字段解释：

- TABLE_NAME：对应于 METRICS_SCHEMA 中的表名。

- PROMQL：监控表的主要原理是将 SQL 映射成 PromQL，并将 Prometheus 结果转换成 SQL 查询结果。这个字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控定义的 label，每一个label 对应监控表中的一列。SQL 中如果包含对应列的过滤，对应的 PromQL 也会改变。
- QUANTILE：百分位。对于直方图类型的监控数据，指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。
- COMMENT：对这个监控表的注释。

```
SELECT * FROM METRICS_TABLES LIMIT 5\G
```

输出结果如下：

```
*************************** 1. row ***************************
TABLE_NAME: abnormal_stores
    PROMQL: sum(pd_cluster_status{ type=~"store_disconnected_count|store_unhealth_count|
        <-> store_low_space_count|store_down_count|store_offline_count|store_tombstone_count"})
    LABELS: instance,type
  QUANTILE: 0
   COMMENT:
*************************** 2. row ***************************
TABLE_NAME: etcd_disk_wal_fsync_rate
    PROMQL: delta(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[$RANGE_DURATION])
    LABELS: instance
  QUANTILE: 0
   COMMENT: The rate of writing WAL into the persistent storage
*************************** 3. row ***************************
TABLE_NAME: etcd_wal_fsync_duration
    PROMQL: histogram_quantile($QUANTILE, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket{
        <-> $LABEL_CONDITIONS}[$RANGE_DURATION])) by (le,instance))
    LABELS: instance
  QUANTILE: 0.99
   COMMENT: The quantile time consumed of writing WAL into the persistent storage
*************************** 4. row ***************************
TABLE_NAME: etcd_wal_fsync_total_count
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[
        <-> $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
   COMMENT: The total count of writing WAL into the persistent storage
*************************** 5. row ***************************
TABLE_NAME: etcd_wal_fsync_total_time
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_sum{$LABEL_CONDITIONS}[
        <-> $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
```

```
   COMMENT: The total time of writing WAL into the persistent storage
5 rows in set (0.00 sec)
```

4.8.18.2.28 PARTITIONS

PARTITIONS 表提供有关分区表的信息。

```
USE INFORMATION_SCHEMA;
DESC PARTITIONS;
```

输出结果如下：

```
+-------------------------------+--------------+------+------+---------+----------------+
| Field                         | Type         | Null | Key  | Default | Extra          |
+-------------------------------+--------------+------+------+---------+----------------+
| TABLE_CATALOG                 | varchar(512) | YES  |      | NULL    |                |
| TABLE_SCHEMA                  | varchar(64)  | YES  |      | NULL    |                |
| TABLE_NAME                    | varchar(64)  | YES  |      | NULL    |                |
| PARTITION_NAME                | varchar(64)  | YES  |      | NULL    |                |
| SUBPARTITION_NAME             | varchar(64)  | YES  |      | NULL    |                |
| PARTITION_ORDINAL_POSITION    | bigint(21)   | YES  |      | NULL    |                |
| SUBPARTITION_ORDINAL_POSITION | bigint(21)   | YES  |      | NULL    |                |
| PARTITION_METHOD              | varchar(18)  | YES  |      | NULL    |                |
| SUBPARTITION_METHOD           | varchar(12)  | YES  |      | NULL    |                |
| PARTITION_EXPRESSION          | longtext     | YES  |      | NULL    |                |
| SUBPARTITION_EXPRESSION       | longtext     | YES  |      | NULL    |                |
| PARTITION_DESCRIPTION         | longtext     | YES  |      | NULL    |                |
| TABLE_ROWS                    | bigint(21)   | YES  |      | NULL    |                |
| AVG_ROW_LENGTH                | bigint(21)   | YES  |      | NULL    |                |
| DATA_LENGTH                   | bigint(21)   | YES  |      | NULL    |                |
| MAX_DATA_LENGTH               | bigint(21)   | YES  |      | NULL    |                |
| INDEX_LENGTH                  | bigint(21)   | YES  |      | NULL    |                |
| DATA_FREE                     | bigint(21)   | YES  |      | NULL    |                |
| CREATE_TIME                   | datetime     | YES  |      | NULL    |                |
| UPDATE_TIME                   | datetime     | YES  |      | NULL    |                |
| CHECK_TIME                    | datetime     | YES  |      | NULL    |                |
| CHECKSUM                      | bigint(21)   | YES  |      | NULL    |                |
| PARTITION_COMMENT             | varchar(80)  | YES  |      | NULL    |                |
| NODEGROUP                     | varchar(12)  | YES  |      | NULL    |                |
| TABLESPACE_NAME               | varchar(64)  | YES  |      | NULL    |                |
| TIDB_PARTITION_ID             | bigint(21)   | YES  |      | NULL    |                |
| TIDB_PLACEMENT_POLICY_NAME    | varchar(64)  | YES  |      | NULL    |                |
+-------------------------------+--------------+------+------+---------+----------------+
27 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT NOT NULL PRIMARY KEY) PARTITION BY HASH (id) PARTITIONS 2;
SELECT * FROM PARTITIONS WHERE table_schema='test' AND table_name='t1'\G
```

输出结果如下：

```
*************************** 1. row ***************************
               TABLE_CATALOG: def
                TABLE_SCHEMA: test
                  TABLE_NAME: t1
              PARTITION_NAME: p0
           SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 1
SUBPARTITION_ORDINAL_POSITION: NULL
             PARTITION_METHOD: HASH
          SUBPARTITION_METHOD: NULL
         PARTITION_EXPRESSION: `id`
      SUBPARTITION_EXPRESSION: NULL
        PARTITION_DESCRIPTION:
                   TABLE_ROWS: 0
               AVG_ROW_LENGTH: 0
                  DATA_LENGTH: 0
              MAX_DATA_LENGTH: 0
                 INDEX_LENGTH: 0
                    DATA_FREE: 0
                  CREATE_TIME: 2022-12-14 06:09:33
                  UPDATE_TIME: NULL
                   CHECK_TIME: NULL
                     CHECKSUM: NULL
            PARTITION_COMMENT:
                    NODEGROUP: NULL
              TABLESPACE_NAME: NULL
            TIDB_PARTITION_ID: 89
   TIDB_PLACEMENT_POLICY_NAME: NULL
*************************** 2. row ***************************
               TABLE_CATALOG: def
                TABLE_SCHEMA: test
                  TABLE_NAME: t1
              PARTITION_NAME: p1
           SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 2
SUBPARTITION_ORDINAL_POSITION: NULL
             PARTITION_METHOD: HASH
          SUBPARTITION_METHOD: NULL
         PARTITION_EXPRESSION: `id`
      SUBPARTITION_EXPRESSION: NULL
```

```
        PARTITION_DESCRIPTION:
                   TABLE_ROWS: 0
               AVG_ROW_LENGTH: 0
                  DATA_LENGTH: 0
              MAX_DATA_LENGTH: 0
                 INDEX_LENGTH: 0
                    DATA_FREE: 0
                  CREATE_TIME: 2022-12-14 06:09:33
                  UPDATE_TIME: NULL
                   CHECK_TIME: NULL
                     CHECKSUM: NULL
            PARTITION_COMMENT:
                     NODEGROUP: NULL
              TABLESPACE_NAME: NULL
            TIDB_PARTITION_ID: 90
   TIDB_PLACEMENT_POLICY_NAME: NULL
2 rows in set (0.00 sec)
```

4.8.18.2.29 PLACEMENT_POLICIES

PLACEMENT_POLICIES 表展示所有放置策略(placement policy) 的信息，见Placement Rules in SQL。

```
USE information_schema;
DESC placement_policies;
```

```
+----------------------+---------------+------+-----+---------+-------+
| Field                | Type          | Null | Key | Default | Extra |
+----------------------+---------------+------+-----+---------+-------+
| POLICY_ID            | bigint(64)    | NO   |     | <null>  |       |
| CATALOG_NAME         | varchar(512)  | NO   |     | <null>  |       |
| POLICY_NAME          | varchar(64)   | NO   |     | <null>  |       |
| PRIMARY_REGION       | varchar(1024) | YES  |     | <null>  |       |
| REGIONS              | varchar(1024) | YES  |     | <null>  |       |
| CONSTRAINTS          | varchar(1024) | YES  |     | <null>  |       |
| LEADER_CONSTRAINTS   | varchar(1024) | YES  |     | <null>  |       |
| FOLLOWER_CONSTRAINTS | varchar(1024) | YES  |     | <null>  |       |
| LEARNER_CONSTRAINTS  | varchar(1024) | YES  |     | <null>  |       |
| SCHEDULE             | varchar(20)   | YES  |     | <null>  |       |
| FOLLOWERS            | bigint(64)    | YES  |     | <null>  |       |
| LEARNERS             | bigint(64)    | YES  |     | <null>  |       |
+----------------------+---------------+------+-----+---------+-------+
12 rows in set (0.00 sec)
```

示例

PLACEMENT_POLICIES 表只展示放置策略(placement policy) 的信息。如需查看所有信息的规范版本（放置策略，以及绑定放置策略的对象），请改用 SHOW PLACEMENT 语句：

```
CREATE TABLE t1 (a INT);
CREATE PLACEMENT POLICY p1 primary_region="us-east-1" regions="us-east-1";
CREATE TABLE t3 (a INT) PLACEMENT POLICY=p1;
SHOW PLACEMENT; -- 显示所有信息，包含 t3。
SELECT * FROM information_schema.placement_policies; -- 只显示放置策略，不包含 t3。
```

```
Query OK, 0 rows affected (0.09 sec)

Query OK, 0 rows affected (0.11 sec)

Query OK, 0 rows affected (0.08 sec)

+---------------+-----------------------------------------------------+-------------------+
| Target        | Placement                                           | Scheduling_State |
+---------------+-----------------------------------------------------+-------------------+
| POLICY p1     | PRIMARY_REGION="us-east-1" REGIONS="us-east-1" | NULL              |
| TABLE test.t3 | PRIMARY_REGION="us-east-1" REGIONS="us-east-1" | SCHEDULED         |
+---------------+-----------------------------------------------------+-------------------+
2 rows in set (0.00 sec)

+--
    <--> ---------+--------------+-------------+----------------+-----------+-------------+ ------------------------------+--
    <-->
| POLICY_ID | CATALOG_NAME | POLICY_NAME | PRIMARY_REGION | REGIONS    | CONSTRAINTS |
    <--> LEADER_CONSTRAINTS | FOLLOWER_CONSTRAINTS | LEARNER_CONSTRAINTS | SCHEDULE | FOLLOWERS |
    <--> LEARNERS |
+--
    <--> ---------+--------------+-------------+----------------+-----------+-------------+ ------------------------------+--
    <-->
| 1         | def          | p1          | us-east-1      | us-east-1 |             |
    <-->                   |                       |                     |           | 2         |
    <--> 0        |
+--
    <--> ---------+--------------+-------------+----------------+-----------+-------------+ ------------------------------+--
    <-->
1 rows in set (0.00 sec)
```

4.8.18.2.30 PROCESSLIST

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

PROCESSLIST 表比 SHOW PROCESSLIST 的结果多出下面几列：

- DIGEST 列：显示 SQL 语句的digest。
- MEM 列：显示正在处理的请求已使用的内存，单位是byte。
- DISK 列：显示磁盘空间使用情况，单位是byte。
- TxnStart列：显示事务的开始时间。
- RESOURCE_GROUP列：显示对应的资源组名称。

```
USE information_schema;
DESC processlist;
```

```
+--------------------+---------------------+------+------+---------+--------------+
| Field              | Type                | Null | Key  | Default | Extra        |
+--------------------+---------------------+------+------+---------+--------------+
| ID                 | bigint(21) unsigned | NO   |      | 0       |              |
| USER               | varchar(16)         | NO   |      |         |              |
| HOST               | varchar(64)         | NO   |      |         |              |
| DB                 | varchar(64)         | YES  |      | NULL    |              |
| COMMAND            | varchar(16)         | NO   |      |         |              |
| TIME               | int(7)              | NO   |      | 0       |              |
| STATE              | varchar(7)          | YES  |      | NULL    |              |
| INFO               | longtext            | YES  |      | NULL    |              |
| DIGEST             | varchar(64)         | YES  |      |         |              |
| MEM                | bigint(21) unsigned | YES  |      | NULL    |              |
| DISK               | bigint(21) unsigned | YES  |      | NULL    |              |
| TxnStart           | varchar(64)         | NO   |      |         |              |
| RESOURCE_GROUP     | varchar(32)         | NO   |      |         |              |
+--------------------+---------------------+------+------+---------+--------------+
13 rows in set (0.00 sec)
```

```
SELECT * FROM processlist\G
```

```
*************************** 1. row ***************************
                ID: 2300033189772525975
              USER: root
              HOST: 127.0.0.1:51289
                DB: NULL
           COMMAND: Query
              TIME: 0
             STATE: autocommit
              INFO: SELECT * FROM processlist
            DIGEST: dbfaa16980ec628011029f0aaf0d160f4b040885240dfc567bf760d96d374f7e
               MEM: 0
              DISK: 0
          TxnStart:
    RESOURCE_GROUP: rg1
1 row in set (0.00 sec)
```

PROCESSLIST 表各列的含义如下：

- ID 列：客户连接ID。
- USER 列：执行当前 PROCESS 的用户名。
- HOST 列：客户连接的地址。
- DB 列：当前连接的默认数据库名。
- COMMAND 列：当前 PROCESS 执行的命令类型。
- TIME 列：当前 PROCESS 的已经执行的时间，单位是秒。
- STATE 列：当前连接的状态。
- INFO 列：正在处理的请求语句。
- DIGEST 列：SQL 语句的digest。
- MEM 列：正在处理的请求已使用的内存，单位是byte。
- DISK 列：磁盘空间使用情况，单位是byte。
- TxnStart列：显示事务的开始时间。
- RESOURCE_GROUP列：显示对应的资源组名称。

CLUSTER_PROCESSLIST

CLUSTER_PROCESSLIST 是 PROCESSLIST 对应的集群系统表，用于查询集群中所有TiDB 节点的 PROCESSLIST 信息。CLUSTER_PROCESSLIST 表结构上比 PROCESSLIST 多一列 INSTANCE，表示该行数据来自的 TiDB 节点地址。

```
SELECT * FROM cluster_processlist;
```

```
+--
    <-> ---------------+-----+------+----------+------+---------+------+------------+---------------------------
    <->
| INSTANCE         | ID   | USER | HOST      | DB    | COMMAND | TIME | STATE       | INFO
    <->                                                | MEM | TxnStart
    <->                                | RESOURCE_GROUP |
+--
    <-> ---------------+-----+------+----------+------+---------+------+------------+---------------------------
    <->

| 10.0.1.22:10080 | 150 | u1    | 10.0.1.1 | test | Query    | 0     | autocommit | select count(*)
    <-> from usertable                          | 372 | 05-28 03:54:21.230(416976223923077223) |
    <-> default         |
| 10.0.1.22:10080 | 138 | root | 10.0.1.1 | test | Query    | 0     | autocommit | SELECT * FROM
    <-> information_schema.cluster_processlist | 0    | 05-28 03:54:21.230(416976223923077220) |
    <-> rg1             |
| 10.0.1.22:10080 | 151 | u1    | 10.0.1.1 | test | Query    | 0     | autocommit | select count(*)
    <-> from usertable                          | 372 | 05-28 03:54:21.230(416976223923077224) | rg2
    <->             |
| 10.0.1.21:10080 | 15   | u2    | 10.0.1.1 | test | Query    | 0     | autocommit | select max(
    <-> field0) from usertable                      | 496 | 05-28 03:54:21.230(416976223923077222)
    <-> | default          |
```

```
| 10.0.1.21:10080 | 14   | u2    | 10.0.1.1 | test | Query    | 0     | autocommit | select max(
    <-> field0) from usertable                    | 496 | 05-28 03:54:21.230(416976223923077225)
    <-> | default          |
+--
    <-> ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    <->
```

4.8.18.2.31 REFERENTIAL_CONSTRAINTS

REFERENTIAL_CONSTRAINTS 表提供TiDB 表之间 FOREIGN KEY 关系的信息。

```
USE INFORMATION_SCHEMA;
DESC REFERENTIAL_CONSTRAINTS;
```

输出结果如下：

```
+--------------------------+--------------+------+------+---------+-------------- +
| Field                    | Type         | Null | Key  | Default | Extra |
+--------------------------+--------------+------+------+---------+-------------- +
| CONSTRAINT_CATALOG       | varchar(512) | NO   |      | NULL    |       |
| CONSTRAINT_SCHEMA        | varchar(64)  | NO   |      | NULL    |       |
| CONSTRAINT_NAME          | varchar(64)  | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_CATALOG | varchar(512) | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_SCHEMA | varchar(64)  | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_NAME   | varchar(64)  | YES  |      | NULL    |       |
| MATCH_OPTION             | varchar(64)  | NO   |      | NULL    |       |
| UPDATE_RULE              | varchar(64)  | NO   |      | NULL    |       |
| DELETE_RULE              | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME               | varchar(64)  | NO   |      | NULL    |       |
| REFERENCED_TABLE_NAME    | varchar(64)  | NO   |      | NULL    |       |
+--------------------------+--------------+------+------+---------+-------------- +
11 rows in set (0.00 sec)
```

```
CREATE TABLE test.parent (
  id INT NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (id)
);

CREATE TABLE test.child (
  id INT NOT NULL AUTO_INCREMENT,
  name varchar(255) NOT NULL,
  parent_id INT DEFAULT NULL,
  PRIMARY KEY (id),
  CONSTRAINT fk_parent FOREIGN KEY (parent_id) REFERENCES parent (id) ON UPDATE CASCADE ON DELETE
      <-> RESTRICT
);
```

```
SELECT * FROM REFERENTIAL_CONSTRAINTS\G
```

输出结果如下：

```
*************************** 1. row ***************************
       CONSTRAINT_CATALOG: def
        CONSTRAINT_SCHEMA: test
          CONSTRAINT_NAME: fk_parent
UNIQUE_CONSTRAINT_CATALOG: def
UNIQUE_CONSTRAINT_SCHEMA: test
   UNIQUE_CONSTRAINT_NAME: PRIMARY
             MATCH_OPTION: NONE
              UPDATE_RULE: CASCADE
              DELETE_RULE: RESTRICT
               TABLE_NAME: child
    REFERENCED_TABLE_NAME: parent
1 row in set (0.00 sec)
```

4.8.18.2.32 RESOURCE_GROUPS

RESOURCE_GROUPS 表展示所有资源组(resource group) 的信息，见使用资源管控(Resource Control) 实现资源隔离。

```
USE information_schema;
DESC resource_groups;
```

```
+------------+-------------+------+------+---------+-------------+
| Field      | Type        | Null | Key  | Default | Extra       |
+------------+-------------+------+------+---------+-------------+
| NAME       | varchar(32) | NO   |      | NULL    |             |
| RU_PER_SEC | bigint(21)  | YES  |      | NULL    |             |
| PRIORITY   | varchar(6)  | YES  |      | NULL    |             |
| BURSTABLE  | varchar(3)  | YES  |      | NULL    |             |
+------------+-------------+------+------+---------+-------------+
3 rows in set (0.00 sec)
```

示例

```
SELECT * FROM information_schema.resource_groups; -- 查看资源组，TiDB 默认预置 `default` 资源组
```

```
+---------+------------+----------+-------------------+
| NAME    | RU_PER_SEC | PRIORITY | BURSTABLE         |
+---------+------------+----------+-------------------+
| default | UNLIMITED  | MEDIUM   | YES               |
+---------+------------+----------+-------------------+
```

```
CREATE RESOURCE GROUP rg1 RU_PER_SEC=1000; -- 创建资源组 `rg1`
```

```
Query OK, 0 rows affected (0.34 sec)
```

```
SHOW CREATE RESOURCE GROUP rg1; -- 显示 `rg1` 资源组的定义
```

```
+----------------+-----------------------------------------------------------------+
| Resource_Group | Create Resource Group                                           |
+----------------+-----------------------------------------------------------------+
| rg1            | CREATE RESOURCE GROUP `rg1` RU_PER_SEC=1000 PRIORITY="MEDIUM" |
+----------------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME = 'rg1'; -- 查看资源组 `rg1`
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |       1000 | MEDIUM   | NO        |
+------+------------+----------+-----------+
1 row in set (0.00 sec)
```

RESOURCE_GROUPS 表中列的含义如下：

- NAME：资源组名称。
- RU_PER_SEC：资源组的回填速度，单位为每秒回填的Request Unit (RU) 数量。
- PRIORITY：任务在 TiKV 上处理的绝对优先级。不同的资源按照 PRIORITY 的设置进行调度，PRIORITY 高的任务会被优先调度。如果资源组的 PRIORITY 相同，则会根据 RU_PER_SEC 的配置按比例调度。如果不指定 PRIORITY，资源组的默认优先级为 MEDIUM。
- BURSTABLE：是否允许此资源组超额使用剩余的系统资源。

> **注意：**
>
> TiDB 集群在初始化时会自动创建 default 资源组，其 RU_PER_SEC 的默认值为 UNLIMITED (等同于 INT 类型最大值，即 2147483647)，且为 BURSTABLE 模式。所有未绑定资源组的请求都将自动绑定至此资源组。在新建配置其他资源组时，建议根据实际情况修改 default 资源组的配置。

4.8.18.2.33 SCHEMATA

SCHEMATA 表提供了关于数据库的信息。表中的数据与 SHOW DATABASES 语句的执行结果等价。

```
USE information_schema;
desc SCHEMATA;
```

```
+---------------------------+--------------+------+------+---------+-------------+
| Field                     | Type         | Null | Key  | Default | Extra       |
+---------------------------+--------------+------+------+---------+-------------+
| CATALOG_NAME              | varchar(512) | YES  |      | NULL    |             |
| SCHEMA_NAME               | varchar(64)  | YES  |      | NULL    |             |
| DEFAULT_CHARACTER_SET_NAME | varchar(64)  | YES  |      | NULL    |             |
| DEFAULT_COLLATION_NAME    | varchar(32)  | YES  |      | NULL    |             |
| SQL_PATH                  | varchar(512) | YES  |      | NULL    |             |
+---------------------------+--------------+------+------+---------+-------------+
5 rows in set (0.00 sec)
```

```
SELECT * FROM SCHEMATA;
```

```
+--
    <-> ------------+--------------------+---------------------------+------------------------+-------------------+
    <->
| CATALOG_NAME | SCHEMA_NAME         | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME |
    <-> SQL_PATH |
+--
    <-> ------------+--------------------+---------------------------+------------------------+-------------------+
    <->
| def          | INFORMATION_SCHEMA  | utf8mb4                    | utf8mb4_bin            | NULL
   <->      |
| def          | METRICS_SCHEMA      | utf8mb4                    | utf8mb4_bin            | NULL
   <->      |
| def          | mysql               | utf8mb4                    | utf8mb4_bin            | NULL
   <->      |
| def          | PERFORMANCE_SCHEMA  | utf8mb4                    | utf8mb4_bin            | NULL
   <->      |
| def          | test                | utf8mb4                    | utf8mb4_bin            | NULL
   <->      |
+--
    <-> ------------+--------------------+---------------------------+------------------------+-------------------+
    <->
5 rows in set (0.00 sec)
```

SCHEMATA 表各列字段含义如下：

- CATALOG_NAME：数据库归属的目录名，该列值永远为 def。
- SCHEMA_NAME：数据库的名字。
- DEFAULT_CHARACTER_SET_NAME：数据库的默认字符集。
- DEFAULT_COLLATION_NAME：数据库的默认 collation。
- SQL_PATH：该项值永远为 NULL。

4.8.18.2.34 SEQUENCES

SEQUENCES 表提供了有关序列的信息。TiDB 中序列的功能是参照 MariaDB 中的类似功能来实现的。

```
USE INFORMATION_SCHEMA;
DESC SEQUENCES;
```

输出结果如下：

```
+-----------------+--------------+------+------+---------+--------------+
| Field           | Type         | Null | Key  | Default | Extra        |
+-----------------+--------------+------+------+---------+--------------+
| TABLE_CATALOG   | varchar(512) | NO   |      | NULL    |              |
| SEQUENCE_SCHEMA | varchar(64)  | NO   |      | NULL    |              |
| SEQUENCE_NAME   | varchar(64)  | NO   |      | NULL    |              |
| CACHE           | tinyint(0)   | NO   |      | NULL    |              |
| CACHE_VALUE     | bigint(21)   | YES  |      | NULL    |              |
| CYCLE           | tinyint(0)   | NO   |      | NULL    |              |
| INCREMENT       | bigint(21)   | NO   |      | NULL    |              |
| MAX_VALUE       | bigint(21)   | YES  |      | NULL    |              |
| MIN_VALUE       | bigint(21)   | YES  |      | NULL    |              |
| START           | bigint(21)   | YES  |      | NULL    |              |
| COMMENT         | varchar(64)  | YES  |      | NULL    |              |
+-----------------+--------------+------+------+---------+--------------+
11 rows in set (0.00 sec)
```

创建一个名为 test.seq 的序列，并查询这个序列的下一个值：

```
CREATE SEQUENCE test.seq;
SELECT nextval(test.seq);
```

输出结果如下：

```
+--------------------+
| nextval(test.seq) |
+--------------------+
|                  1 |
+--------------------+
1 row in set (0.01 sec)
```

查询数据库中的所有序列：

```
SELECT * FROM SEQUENCES\G
```

输出结果如下：

```
*************************** 1. row ***************************
  TABLE_CATALOG: def
SEQUENCE_SCHEMA: test
```

```
  SEQUENCE_NAME: seq
          CACHE: 1
    CACHE_VALUE: 1000
          CYCLE: 0
      INCREMENT: 1
      MAX_VALUE: 9223372036854775806
      MIN_VALUE: 1
          START: 1
        COMMENT:
1 row in set (0.00 sec)
```

4.8.18.2.35 SESSION_VARIABLES

SESSION_VARIABLES 表提供了关于 session 变量的信息。表中的数据跟 SHOW SESSION VARIABLES 语句执行结果类似。

```
USE INFORMATION_SCHEMA;
DESC SESSION_VARIABLES;
```

输出结果如下：

```
+----------------+---------------+------+------+---------+-------------+
| Field          | Type          | Null | Key  | Default | Extra |
+----------------+---------------+------+------+---------+-------------+
| VARIABLE_NAME  | varchar(64)   | YES  |      | NULL    |       |
| VARIABLE_VALUE | varchar(1024) | YES  |      | NULL    |       |
+----------------+---------------+------+------+---------+-------------+
2 rows in set (0.00 sec)
```

查看 SESSION_VARIABLES 表的前 10 条信息：

```
SELECT * FROM SESSION_VARIABLES ORDER BY variable_name LIMIT 10;
```

输出结果如下：

```
+-----------------------------------+-------------------+
| VARIABLE_NAME                     | VARIABLE_VALUE    |
+-----------------------------------+-------------------+
| allow_auto_random_explicit_insert | OFF               |
| auto_increment_increment          | 1                 |
| auto_increment_offset             | 1                 |
| autocommit                        | ON                |
| automatic_sp_privileges           | 1                 |
| avoid_temporal_upgrade            | OFF               |
| back_log                          | 80                |
| basedir                           | /usr/local/mysql |
| big_tables                        | OFF               |
```

```
| bind_address                         | *                 |
+--------------------------------------+-------------------+
10 rows in set (0.00 sec)
```

SESSION_VARIABLES 表各列字段含义如下：

- VARIABLE_NAME：数据库中 session 级变量的名称。
- VARIABLE_VALUE：数据库中对应该 session 变量名的具体值。

4.8.18.2.36 SLOW_QUERY

SLOW_QUERY 表中提供了当前节点的慢查询相关的信息，其内容通过解析当前节点的TiDB 慢查询日志而来，列名和慢日志中的字段名是一一对应。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
USE INFORMATION_SCHEMA;
DESC slow_query;
```

输出结果示例如下：

```
+------------------------------+---------------------+------+------+---------+---------------+
| Field                        | Type                | Null | Key  | Default | Extra         |
+------------------------------+---------------------+------+------+---------+---------------+
| Time                         | timestamp(6)        | NO   | PRI  | 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    |               |
| Exec_retry_count             | bigint(20) unsigned | YES  |      | NULL    |               |
| Exec_retry_time              | double              | YES  |      | NULL    |               |
| Query_time                   | double              | YES  |      | NULL    |               |
| Parse_time                   | double              | YES  |      | NULL    |               |
| Compile_time                 | double              | YES  |      | NULL    |               |
| Rewrite_time                 | double              | YES  |      | NULL    |               |
| Preproc_subqueries           | bigint(20) unsigned | YES  |      | NULL    |               |
| Preproc_subqueries_time      | double              | YES  |      | NULL    |               |
| Optimize_time                | double              | YES  |      | NULL    |               |
| Wait_TS                      | double              | YES  |      | NULL    |               |
| Prewrite_time                | double              | YES  |      | NULL    |               |
| Wait_prewrite_binlog_time    | double              | YES  |      | NULL    |               |
| Commit_time                  | double              | YES  |      | NULL    |               |
| Get_commit_ts_time           | double              | YES  |      | NULL    |               |
| Commit_backoff_time          | double              | YES  |      | NULL    |               |
| Backoff_types                | varchar(64)         | YES  |      | NULL    |               |
| Resolve_lock_time            | double              | YES  |      | NULL    |               |
| Local_latch_wait_time        | double              | YES  |      | NULL    |               |
| Write_keys                   | bigint(22)          | YES  |      | NULL    |               |
```

```
| Write_size                    | bigint(22)           | YES  |      | NULL    |       |
| Prewrite_region               | bigint(22)           | YES  |      | NULL    |       |
| Txn_retry                     | bigint(22)           | YES  |      | NULL    |       |
| Cop_time                      | double               | YES  |      | NULL    |       |
| Process_time                  | double               | YES  |      | NULL    |       |
| Wait_time                     | double               | YES  |      | NULL    |       |
| Backoff_time                  | double               | YES  |      | NULL    |       |
| LockKeys_time                 | double               | YES  |      | NULL    |       |
| Request_count                 | bigint(20) unsigned | YES  |      | NULL    |       |
| Total_keys                    | bigint(20) unsigned | YES  |      | NULL    |       |
| Process_keys                  | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_delete_skipped_count  | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_key_skipped_count     | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_block_cache_hit_count | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_block_read_count      | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_block_read_byte       | bigint(20) unsigned | YES  |      | NULL    |       |
| DB                            | varchar(64)          | YES  |      | NULL    |       |
| Index_names                   | varchar(100)         | YES  |      | NULL    |       |
| Is_internal                   | tinyint(1)           | YES  |      | NULL    |       |
| Digest                        | varchar(64)          | YES  |      | NULL    |       |
| Stats                         | varchar(512)         | YES  |      | NULL    |       |
| Cop_proc_avg                  | double               | YES  |      | NULL    |       |
| Cop_proc_p90                  | double               | YES  |      | NULL    |       |
| Cop_proc_max                  | double               | YES  |      | NULL    |       |
| Cop_proc_addr                 | varchar(64)          | YES  |      | NULL    |       |
| Cop_wait_avg                  | double               | YES  |      | NULL    |       |
| Cop_wait_p90                  | double               | YES  |      | NULL    |       |
| Cop_wait_max                  | double               | YES  |      | NULL    |       |
| Cop_wait_addr                 | varchar(64)          | YES  |      | NULL    |       |
| Mem_max                       | bigint(20)           | YES  |      | NULL    |       |
| Disk_max                      | bigint(20)           | YES  |      | NULL    |       |
| KV_total                      | double               | YES  |      | NULL    |       |
| PD_total                      | double               | YES  |      | NULL    |       |
| Backoff_total                 | double               | YES  |      | NULL    |       |
| Write_sql_response_total      | double               | YES  |      | NULL    |       |
| Result_rows                   | bigint(22)           | YES  |      | NULL    |       |
| Warnings                      | longtext             | YES  |      | NULL    |       |
| Backoff_Detail                | varchar(4096)        | YES  |      | NULL    |       |
| Prepared                      | tinyint(1)           | YES  |      | NULL    |       |
| Succ                          | tinyint(1)           | YES  |      | NULL    |       |
| IsExplicitTxn                 | tinyint(1)           | YES  |      | NULL    |       |
| IsWriteCacheTable             | tinyint(1)           | YES  |      | NULL    |       |
| Plan_from_cache               | tinyint(1)           | YES  |      | NULL    |       |
| Plan_from_binding             | tinyint(1)           | YES  |      | NULL    |       |
| Has_more_results              | tinyint(1)           | YES  |      | NULL    |       |
```

```
| Plan                          | longtext            | YES  |      | NULL    |       |
| Plan_digest                   | varchar(128)        | YES  |      | NULL    |       |
| Binary_plan                   | longtext            | YES  |      | NULL    |       |
| Prev_stmt                     | longtext            | YES  |      | NULL    |       |
| Query                         | longtext            | YES  |      | NULL    |       |
+-------------------------------+---------------------+------+------+---------+--------------+
74 rows in set (0.001 sec)
```

CLUSTER_SLOW_QUERY table

CLUSTER_SLOW_QUERY 表中提供了集群所有节点的慢查询相关的信息，其内容通过解析TiDB 慢查询日志而来，该表使用上和 SLOW_QUERY 表一样。CLUSTER_SLOW_QUERY 表结构上比 SLOW_QUERY 多一列 INSTANCE，表示该行慢查询信息来自的TiDB 节点地址。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
DESC CLUSTER_SLOW_QUERY;
```

输出结果示例如下：

```
+-------------------------------+---------------------+------+------+---------+--------------+
| Field                         | Type                | Null | Key  | Default | Extra        |
+-------------------------------+---------------------+------+------+---------+--------------+
| INSTANCE                      | varchar(64)         | YES  |      | NULL    |              |
| Time                          | timestamp(6)        | NO   | PRI  | 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    |              |
| Exec_retry_count              | bigint(20) unsigned | YES  |      | NULL    |              |
| Exec_retry_time               | double              | YES  |      | NULL    |              |
| Query_time                    | double              | YES  |      | NULL    |              |
| Parse_time                    | double              | YES  |      | NULL    |              |
| Compile_time                  | double              | YES  |      | NULL    |              |
| Rewrite_time                  | double              | YES  |      | NULL    |              |
| Preproc_subqueries            | bigint(20) unsigned | YES  |      | NULL    |              |
| Preproc_subqueries_time       | double              | YES  |      | NULL    |              |
| Optimize_time                 | double              | YES  |      | NULL    |              |
| Wait_TS                       | double              | YES  |      | NULL    |              |
| Prewrite_time                 | double              | YES  |      | NULL    |              |
| Wait_prewrite_binlog_time     | double              | YES  |      | NULL    |              |
| Commit_time                   | double              | YES  |      | NULL    |              |
| Get_commit_ts_time            | double              | YES  |      | NULL    |              |
| Commit_backoff_time           | double              | YES  |      | NULL    |              |
| Backoff_types                 | varchar(64)         | YES  |      | NULL    |              |
| Resolve_lock_time             | double              | YES  |      | NULL    |              |
| Local_latch_wait_time         | double              | YES  |      | NULL    |              |
| Write_keys                    | bigint(22)          | YES  |      | NULL    |              |
| Write_size                    | bigint(22)          | YES  |      | NULL    |              |
```

```
| Prewrite_region               | bigint(22)          | YES  |      | NULL    |       |
| Txn_retry                     | bigint(22)          | YES  |      | NULL    |       |
| Cop_time                      | double              | YES  |      | NULL    |       |
| Process_time                  | double              | YES  |      | NULL    |       |
| Wait_time                     | double              | YES  |      | NULL    |       |
| Backoff_time                  | double              | YES  |      | NULL    |       |
| LockKeys_time                 | double              | YES  |      | NULL    |       |
| Request_count                 | bigint(20) unsigned | YES  |      | NULL    |       |
| Total_keys                    | bigint(20) unsigned | YES  |      | NULL    |       |
| Process_keys                  | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_delete_skipped_count  | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_key_skipped_count     | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_block_cache_hit_count | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_block_read_count      | bigint(20) unsigned | YES  |      | NULL    |       |
| Rocksdb_block_read_byte       | bigint(20) unsigned | YES  |      | NULL    |       |
| DB                            | varchar(64)         | YES  |      | NULL    |       |
| Index_names                   | varchar(100)        | YES  |      | NULL    |       |
| Is_internal                   | tinyint(1)          | YES  |      | NULL    |       |
| Digest                        | varchar(64)         | YES  |      | NULL    |       |
| Stats                         | varchar(512)        | YES  |      | NULL    |       |
| Cop_proc_avg                  | double              | YES  |      | NULL    |       |
| Cop_proc_p90                  | double              | YES  |      | NULL    |       |
| Cop_proc_max                  | double              | YES  |      | NULL    |       |
| Cop_proc_addr                 | varchar(64)         | YES  |      | NULL    |       |
| Cop_wait_avg                  | double              | YES  |      | NULL    |       |
| Cop_wait_p90                  | double              | YES  |      | NULL    |       |
| Cop_wait_max                  | double              | YES  |      | NULL    |       |
| Cop_wait_addr                 | varchar(64)         | YES  |      | NULL    |       |
| Mem_max                       | bigint(20)          | YES  |      | NULL    |       |
| Disk_max                      | bigint(20)          | YES  |      | NULL    |       |
| KV_total                      | double              | YES  |      | NULL    |       |
| PD_total                      | double              | YES  |      | NULL    |       |
| Backoff_total                 | double              | YES  |      | NULL    |       |
| Write_sql_response_total      | double              | YES  |      | NULL    |       |
| Result_rows                   | bigint(22)          | YES  |      | NULL    |       |
| Warnings                      | longtext            | YES  |      | NULL    |       |
| Backoff_Detail                | varchar(4096)       | YES  |      | NULL    |       |
| Prepared                      | tinyint(1)          | YES  |      | NULL    |       |
| Succ                          | tinyint(1)          | YES  |      | NULL    |       |
| IsExplicitTxn                 | tinyint(1)          | YES  |      | NULL    |       |
| IsWriteCacheTable             | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache               | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_binding             | tinyint(1)          | YES  |      | NULL    |       |
| Has_more_results              | tinyint(1)          | YES  |      | NULL    |       |
| Plan                          | longtext            | YES  |      | NULL    |       |
```

```
| Plan_digest                   | varchar(128)        | YES  |      | NULL    |       |
| Binary_plan                   | longtext            | YES  |      | NULL    |       |
| Prev_stmt                     | longtext            | YES  |      | NULL    |       |
| Query                         | longtext            | YES  |      | NULL    |       |
+-------------------------------+---------------------+------+------+---------+-------+
75 rows in set (0.001 sec)
```

查询集群系统表时，TiDB 也会将相关计算下推给其他节点执行，而不是把所有节点的数据都取回来，可以查看执行计划，如下：

```
DESC SELECT COUNT(*) FROM CLUSTER_SLOW_QUERY WHERE user = 'u1';
```

输出结果示例如下：

```
+--
    <-->                              +          +          +                          +
    <-->
| id                          | estRows   | task       | access object             | operator info
    <-->                                    |
+--
    <>----------------------------+----------+-----------+---------------------------+
    <-->
| StreamAgg_7                 | 1.00      | root       |                           | funcs:count(1)->
    <-> Column#75                           |
| └─TableReader_13        | 10.00     | root       |                           | data:Selection_12
    <-->                                    |
|    └─Selection_12       | 10.00     | cop[tidb] |                           | eq(
    <-> INFORMATION_SCHEMA.cluster_slow_query.user, "u1") |
|      └─TableFullScan_11 | 10000.00 | cop[tidb] | table:CLUSTER_SLOW_QUERY | keep order:false,
    <-> stats:pseudo                        |
+--
    <-->                              +          +          +                          +
    <-->
4 rows in set (0.00 sec)
```

上面执行计划表示，会将 user = u1 条件下推给其他的(cop) TiDB 节点执行，也会把聚合算子（即上面输出结果中的 StreamAgg 算子）下推。

目前由于没有对系统表收集统计信息，所以有时会导致某些聚合算子不能下推，导致执行较慢，用户可以通过手动指定聚合下推的 SQL HINT 来将聚合算子下推，示例如下：

```
SELECT /*+ AGG_TO_COP() */ COUNT(*) FROM CLUSTER_SLOW_QUERY GROUP BY user;
```

4.8.18.2.37 STATISTICS

STATISTICS 表提供了关于表索引的信息。

```
USE information_schema;
DESC statistics;
```

```
+---------------+---------------+------+------+---------+-------------+
| Field         | Type          | Null | Key  | Default | Extra       |
+---------------+---------------+------+------+---------+-------------+
| TABLE_CATALOG | varchar(512)  | YES  |      | NULL    |             |
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |             |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |             |
| NON_UNIQUE    | varchar(1)    | YES  |      | NULL    |             |
| INDEX_SCHEMA  | varchar(64)   | YES  |      | NULL    |             |
| INDEX_NAME    | varchar(64)   | YES  |      | NULL    |             |
| SEQ_IN_INDEX  | bigint(2)     | YES  |      | NULL    |             |
| COLUMN_NAME   | varchar(21)   | YES  |      | NULL    |             |
| COLLATION     | varchar(1)    | YES  |      | NULL    |             |
| CARDINALITY   | bigint(21)    | YES  |      | NULL    |             |
| SUB_PART      | bigint(3)     | YES  |      | NULL    |             |
| PACKED        | varchar(10)   | YES  |      | NULL    |             |
| NULLABLE      | varchar(3)    | YES  |      | NULL    |             |
| INDEX_TYPE    | varchar(16)   | YES  |      | NULL    |             |
| COMMENT       | varchar(16)   | YES  |      | NULL    |             |
| INDEX_COMMENT | varchar(1024) | YES  |      | NULL    |             |
| IS_VISIBLE    | varchar(3)    | YES  |      | NULL    |             |
| Expression    | varchar(64)   | YES  |      | NULL    |             |
+---------------+---------------+------+------+---------+-------------+
18 rows in set (0.00 sec)
```

STATISTICS 表中列的含义如下：

- TABLE_CATALOG：包含索引的表所属的目录的名称。这个值总是 def。
- TABLE_SCHEMA：包含索引的表所属的数据库的名称。
- TABLE_NAME：包含索引的表的名称。
- NON_UNIQUE：如果索引不能包含重复项，则为 0；如果可以，则为 1。
- INDEX_SCHEMA：索引所属的数据库的名称。
- INDEX_NAME：索引的名称。如果索引是主键，那么名称总是 PRIMARY。
- SEQ_IN_INDEX：索引中的列序号，从 1 开始。
- COLUMN_NAME：列名。请参见表达式列的说明。
- COLLATION：列在索引中的排序方式。取值可以是 A（升序）、D（降序）或 NULL（未排序）。
- CARDINALITY：TiDB 未使用该字段。该字段的值总是 0。
- SUB_PART：索引的前缀。如果只对列的部分前缀进行索引，则为索引字符的数量；如果对整个列进行索引，则为 NULL。
- PACKED：TiDB 未使用该字段。这个值总是 NULL。
- NULLABLE：如果列可能包含 NULL 值，则值为 YES；如果不包含，则值为 ''。
- INDEX_TYPE：索引的类型。

- COMMENT：其他与索引有关的信息。
- INDEX_COMMENT：在创建索引时为索引提供的带有注释属性的任何注释。
- IS_VISIBLE：优化器能否使用该索引。
- Expression 对于非表达式部分的索引键，这个值为 NULL；对于表达式部分的索引键，这个值为表达式本身。可参考表达式索引

下列语句是等价的：

```
SELECT * FROM INFORMATION_SCHEMA.STATISTICS
  WHERE table_name = 'tbl_name'
  AND table_schema = 'db_name'

SHOW INDEX
  FROM tbl_name
  FROM db_name
```

4.8.18.2.38 TABLES

TABLES 表提供了数据库里关于表的信息。

```
USE information_schema;
DESC tables;
```

```
+---------------------------+---------------+------+------+----------+----------------+
| Field                     | Type          | Null | Key  | Default  | Extra          |
+---------------------------+---------------+------+------+----------+----------------+
| TABLE_CATALOG             | varchar(512)  | YES  |      | NULL     |                |
| TABLE_SCHEMA              | varchar(64)   | YES  |      | NULL     |                |
| TABLE_NAME                | varchar(64)   | YES  |      | NULL     |                |
| TABLE_TYPE                | varchar(64)   | YES  |      | NULL     |                |
| ENGINE                    | varchar(64)   | YES  |      | NULL     |                |
| VERSION                   | bigint(21)    | YES  |      | NULL     |                |
| ROW_FORMAT                | varchar(10)   | YES  |      | NULL     |                |
| TABLE_ROWS                | bigint(21)    | YES  |      | NULL     |                |
| AVG_ROW_LENGTH            | bigint(21)    | YES  |      | NULL     |                |
| DATA_LENGTH               | bigint(21)    | YES  |      | NULL     |                |
| MAX_DATA_LENGTH           | bigint(21)    | YES  |      | NULL     |                |
| INDEX_LENGTH              | bigint(21)    | YES  |      | NULL     |                |
| DATA_FREE                 | bigint(21)    | YES  |      | NULL     |                |
| AUTO_INCREMENT            | bigint(21)    | YES  |      | NULL     |                |
| CREATE_TIME               | datetime      | YES  |      | NULL     |                |
| UPDATE_TIME               | datetime      | YES  |      | NULL     |                |
| CHECK_TIME                | datetime      | YES  |      | NULL     |                |
| TABLE_COLLATION           | varchar(32)   | NO   |      | utf8_bin |                |
| CHECKSUM                  | bigint(21)    | YES  |      | NULL     |                |
| CREATE_OPTIONS            | varchar(255)  | YES  |      | NULL     |                |
```

```
| TABLE_COMMENT              | varchar(2048) | YES  |      | NULL      |        |
| TIDB_TABLE_ID              | bigint(21)    | YES  |      | NULL      |        |
| TIDB_ROW_ID_SHARDING_INFO  | varchar(255)  | YES  |      | NULL      |        |
+----------------------------+---------------+------+------+-----------+--------------+
23 rows in set (0.00 sec)
```

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

```
*************************** 1. row ***************************
            TABLE_CATALOG: def
             TABLE_SCHEMA: mysql
               TABLE_NAME: user
               TABLE_TYPE: BASE TABLE
                   ENGINE: InnoDB
                  VERSION: 10
               ROW_FORMAT: Compact
               TABLE_ROWS: 0
           AVG_ROW_LENGTH: 0
              DATA_LENGTH: 0
          MAX_DATA_LENGTH: 0
             INDEX_LENGTH: 0
                DATA_FREE: 0
           AUTO_INCREMENT: NULL
              CREATE_TIME: 2020-07-05 09:25:51
              UPDATE_TIME: NULL
               CHECK_TIME: NULL
          TABLE_COLLATION: utf8mb4_bin
                 CHECKSUM: NULL
           CREATE_OPTIONS:
            TABLE_COMMENT:
            TIDB_TABLE_ID: 5
TIDB_ROW_ID_SHARDING_INFO: NULL
1 row in set (0.00 sec)
```

下列语句是等价的：

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

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

TABLES 表各列字段含义如下：

- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：表所属数据库的名称。
- TABLE_NAME：表的名称。
- TABLE_TYPE：表的类型。
- ENGINE：存储引擎类型。该值暂为 ‘InnoDB’。
- VERSION：版本，默认值为 10。
- ROW_FORMAT：行格式。该值暂为 ‘Compact’。
- TABLE_ROWS：统计信息中该表所存的行数。
- AVG_ROW_LENGTH：该表中所存数据的平均行长度。平均行长度 = DATA_LENGTH / 统计信息中的行数。
- DATA_LENGTH：数据长度。数据长度 = 统计信息中的行数× 元组各列存储长度和，这里尚未考虑TiKV 的副本数。
- MAX_DATA_LENGTH：最大数据长度。该值暂为 0，表示没有最大数据长度的限制。
- INDEX_LENGTH：索引长度。索引长度 = 统计信息中的行数× 索引元组各列长度和，这里尚未考虑TiKV 的副本数。
- DATA_FREE：空间碎片。该值暂为 0。
- AUTO_INCREMENT：该表中自增主键自动增量的当前值。
- CREATE_TIME：该表的创建时间。
- UPDATE_TIME：该表的更新时间。
- CHECK_TIME：该表的检查时间。
- TABLE_COLLATION：该表的字符校验编码集。
- CHECKSUM：校验和。
- CREATE_OPTIONS：创建选项。
- TABLE_COMMENT：表的注释、备注。

表中的信息大部分定义自 MySQL，只有两列是 TiDB 新增的：

- TIDB_TABLE_ID：标识表的内部 ID，该ID 在一个TiDB 集群内部唯一。
- TIDB_ROW_ID_SHARDING_INFO：标识表的 Sharding 类型，可能的值为：
  - "NOT_SHARDED"：表未被 Shard。
  - "NOT_SHARDED(PK_IS_HANDLE)"：一个定义了整型主键的表未被 Shard。
  - "PK_AUTO_RANDOM_BITS={bit_number}"：一个定义了整型主键的表由于定义了 AUTO_RANDOM 而被 Shard。
  - "SHARD_BITS={bit_number}"：表使用 SHARD_ROW_ID_BITS={bit_number} 进行了 Shard。
  - NULL：表属于系统表或 View，无法被 Shard。

4.8.18.2.39 TABLE_CONSTRAINTS

TABLE_CONSTRAINTS 表记录了表的约束信息。

```
USE information_schema;
DESC table_constraints;
```

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

```
| CONSTRAINT_CATALOG | varchar(512) | YES  |       | NULL    |       |
| CONSTRAINT_SCHEMA  | varchar(64)  | YES  |       | NULL    |       |
| CONSTRAINT_NAME    | varchar(64)  | YES  |       | NULL    |       |
| TABLE_SCHEMA       | varchar(64)  | YES  |       | NULL    |       |
| TABLE_NAME         | varchar(64)  | YES  |       | NULL    |       |
| CONSTRAINT_TYPE    | varchar(64)  | YES  |       | NULL    |       |
+--------------------+--------------+------+-------+---------+-------+
6 rows in set (0.00 sec)
```

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

```
+--
    <-->-------------------+--------------------+--------------------------+--------------------+-----------------------------
    <-->
| CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA  | CONSTRAINT_NAME          | TABLE_SCHEMA       |
    <-> TABLE_NAME                          | CONSTRAINT_TYPE |
+--
    <-->-------------------+--------------------+--------------------------+--------------------+-----------------------------
    <-->
| def                | mysql              | name                     | mysql              |
    <-> help_topic                          | UNIQUE          |
| def                | mysql              | tbl                      | mysql              |
    <-> stats_meta                          | UNIQUE          |
| def                | mysql              | tbl                      | mysql              |
    <-> stats_histograms                    | UNIQUE          |
| def                | mysql              | tbl                      | mysql              |
    <-> stats_buckets                       | UNIQUE          |
| def                | mysql              | delete_range_index       | mysql              |
    <-> gc_delete_range                     | UNIQUE          |
| def                | mysql              | delete_range_done_index | mysql              |
    <-> gc_delete_range_done                | UNIQUE          |
| def                | PERFORMANCE_SCHEMA | SCHEMA_NAME              | PERFORMANCE_SCHEMA |
    <-> events_statements_summary_by_digest | UNIQUE          |
+--
    <-->                  +                    +                          +                    +
    <>------------------- --------------------- -------------------------- --------------------- -----------------------------
7 rows in set (0.01 sec)
```

~~TABLE_CONSTRAINTS~~ 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。这个值总是 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束的名称。
- TABLE_NAME：表的名称。
- CONSTRAINT_TYPE：约束的类型。取值可以是UNIQUE、PRIMARY KEY 或者FOREIGN KEY。UNIQUE 和PRIMARY KEY 信息与 SHOW INDEX 语句的执行结果类似。

4.8.18.2.40 TABLE_STORAGE_STATS

TABLE_STORAGE_STATS 表提供有关由存储引擎(TiKV) 存储的表大小的信息。

```
USE INFORMATION_SCHEMA;
DESC TABLE_STORAGE_STATS;
```

输出结果如下：

```
+--------------------+-------------+------+------+---------+----------------+
| Field              | Type        | Null | Key  | Default | Extra          |
+--------------------+-------------+------+------+---------+----------------+
| TABLE_SCHEMA       | varchar(64) | YES  |      | NULL    |                |
| TABLE_NAME         | varchar(64) | YES  |      | NULL    |                |
| TABLE_ID           | bigint(21)  | YES  |      | NULL    |                |
| PEER_COUNT         | bigint(21)  | YES  |      | NULL    |                |
| REGION_COUNT       | bigint(21)  | YES  |      | NULL    |                |
| EMPTY_REGION_COUNT | bigint(21)  | YES  |      | NULL    |                |
| TABLE_SIZE         | bigint(64)  | YES  |      | NULL    |                |
| TABLE_KEYS         | bigint(64)  | YES  |      | NULL    |                |
+--------------------+-------------+------+------+---------+----------------+
8 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT);
INSERT INTO test.t1 VALUES (1);
SELECT * FROM TABLE_STORAGE_STATS WHERE table_schema = 'test' AND table_name = 't1'\G
```

输出结果如下：

```
*************************** 1. row ***************************
      TABLE_SCHEMA: test
        TABLE_NAME: t1
          TABLE_ID: 56
        PEER_COUNT: 1
      REGION_COUNT: 1
EMPTY_REGION_COUNT: 1
        TABLE_SIZE: 1
        TABLE_KEYS: 0
1 row in set (0.00 sec)
```

TABLE_STORAGE_STATS 表各列字段含义如下：

- TABLE_SCHEMA：表所在 schema 名字
- TABLE_NAME：表名
- TABLE_ID：表 ID
- PEER_COUNT：副本个数
- REGION_COUNT：表所在的 Region 个数

- EMPTY_REGION_COUNT：没有该表数据的 Region 个数
- TABLE_SIZE：数据量大小，单位 MiB
- TABLE_KEYS：表记录个数

4.8.18.2.41 TIDB_HOT_REGIONS

TIDB_HOT_REGIONS 表提供了关于当前热点Region 的相关信息。历史热点信息可以在TIDB_HOT_REGIONS_HISTORY 表查看。

```
USE information_schema;
DESC tidb_hot_regions;
```

```
+----------------+-------------+------+------+---------+----------------+
| Field          | Type        | Null | Key  | Default | Extra          |
+----------------+-------------+------+------+---------+----------------+
| TABLE_ID       | bigint(21)  | YES  |      | NULL    |                |
| INDEX_ID       | bigint(21)  | YES  |      | NULL    |                |
| DB_NAME        | varchar(64) | YES  |      | NULL    |                |
| TABLE_NAME     | varchar(64) | YES  |      | NULL    |                |
| INDEX_NAME     | varchar(64) | YES  |      | NULL    |                |
| REGION_ID      | bigint(21)  | YES  |      | NULL    |                |
| TYPE           | varchar(64) | YES  |      | NULL    |                |
| MAX_HOT_DEGREE | bigint(21)  | YES  |      | NULL    |                |
| REGION_COUNT   | bigint(21)  | YES  |      | NULL    |                |
| FLOW_BYTES     | bigint(21)  | YES  |      | NULL    |                |
+----------------+-------------+------+------+---------+----------------+
10 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS 表各列字段含义如下：

- TABLE_ID：热点Region 所在表的ID。
- INDEX_ID：热点Region 所在索引的ID。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_NAME：热点Region 所在表的名称。
- INDEX_NAME：热点Region 所在索引的名称。
- REGION_ID：热点Region 的 ID。
- TYPE：热点 Region 的类型。
- MAX_HOT_DEGREE：该 Region 的最大热度。
- REGION_COUNT：所在实例的热点Region 数量。
- FLOW_BYTES：该 Region 内读写的字节数量。

4.8.18.2.42 TIDB_HOT_REGIONS_HISTORY

TIDB_HOT_REGIONS_HISTORY 表提供了关于历史热点Region 的相关信息，这些信息由 PD 定期存储在本地，存储间隔为hot-regions-write-interval 的值（默认值为10m），历史热点信息保持的期限为hot-regions-reserved-days 的值（默认值为 7），详情参见PD 配置文件描述。

```
USE information_schema;
DESC tidb_hot_regions_history;
```

```
+-------------+--------------+------+------+---------+-------------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------------+
| UPDATE_TIME | timestamp(6) | YES  |      | NULL    |       |
| DB_NAME     | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME  | varchar(64)  | YES  |      | NULL    |       |
| TABLE_ID    | bigint(21)   | YES  |      | NULL    |       |
| INDEX_NAME  | varchar(64)  | YES  |      | NULL    |       |
| INDEX_ID    | bigint(21)   | YES  |      | NULL    |       |
| REGION_ID   | bigint(21)   | YES  |      | NULL    |       |
| STORE_ID    | bigint(21)   | YES  |      | NULL    |       |
| PEER_ID     | bigint(21)   | YES  |      | NULL    |       |
| IS_LEARNER  | tinyint(1)   | NO   |      | 0       |       |
| IS_LEADER   | tinyint(1)   | NO   |      | 0       |       |
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| HOT_DEGREE  | bigint(21)   | YES  |      | NULL    |       |
| FLOW_BYTES  | double       | YES  |      | NULL    |       |
| KEY_RATE    | double       | YES  |      | NULL    |       |
| QUERY_RATE  | double       | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------------+
16 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS_HISTORY 表各列字段含义如下：

- UPDATE_TIME：热点 Region 更新时间。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_ID：热点Region 所在表的ID。
- TABLE_NAME：热点Region 所在表的名称。
- INDEX_NAME：热点Region 所在索引的名称。
- INDEX_ID：热点Region 所在索引的ID。
- REGION_ID：热点Region 的 ID。
- STORE_ID：热点 Region 所在 TiKV Store 的 ID。
- PEER_ID：热点Region 对应的副本 Peer 的 ID。
- IS_LEARNER：PEER 是否是LEARNER。
- IS_LEADER：PEER 是否是LEADER。
- TYPE：热点 Region 的类型。
- HOT_DEGREE：该热点 Region 的热度。
- FLOW_BYTES：该 Region 内读写的字节数量。
- KEY_RATE：该Region 内读写的 Key 数量。
- QUERY_RATE：该 Region 内读写的 Query 数量。

> **注意：**
>
> TIDB_HOT_REGIONS_HISTORY 表的 UPDATE_TIME、REGION_ID、STORE_ID、PEER_ID、IS_LEARNER、IS_LEADER、TYPE 字段会下推到 PD 服务器执行。所以为了降低使用该表的开销，必须指定搜索时间范围，然后尽可能地指定更多的条件。例如 select * from tidb_hot_regions_history where store_id = 11 and update_time > '2020-05-18 20:40:00' and update_time < '2020-05-18 21:40:00' and type='write'。

常见使用场景

- 查询一段时间内的所有热点Regions。将以下语句中的 update_time 替换为实际所需的时间即可。

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18
    <-> 21:40:00' and update_time <'2021-09-19 00:00:00';
```

> **注意：**
>
> UPDATE_TIME 同样支持 Unix 时间戳。例如：update_time >TIMESTAMP('2012-08-18 21:40:00') 或 update_time > FROM_UNIXTIME(1629294000.000)。

- 查询某张表指定时间内的热点Regions，替换 update_time、table_name 即可。

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18
    <-> 21:40:00' and update_time <'2021-09-19 00:00:00' and TABLE_NAME = 'table_name';
```

- 查询某张表指定时间内热点Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, store_id FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE
    <->  update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and
    <-> table_name = 'table_name' GROUP BY STORE_ID ORDER BY cnt DESC;
```

- 查询某张表指定时间内热点 Leader Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, store_id FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE
    <->  update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and
    <-> table_name = 'table_name' and is_leader=1 GROUP BY STORE_ID ORDER BY cnt DESC;
```

- 查询某张表指定时间内热点Index Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, index_name, store_id FROM INFORMATION_SCHEMA.
    <-> TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'
    <-> 2021-09-19 00:00:00' and table_name = 'table_name' group by index_name, store_id
    <-> order by index_name,cnt desc;
```

- 查询某张表指定时间内热点Index Leader Regions 的分布，替换 update_time、 table_name 即可。

```
SELECT count(region_id) cnt, index_name, store_id FROM INFORMATION_SCHEMA.
    <—> TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'
    <—> 2022-09-19 00:00:00' and table_name = 'table_name' and is_leader=1 group by
    <—> index_name, store_id order by index_name,cnt desc;
```

4.8.18.2.43 TIDB_INDEXES

TIDB_INDEXES 记录了所有表中的INDEX 信息。

```
USE information_schema;
DESC tidb_indexes;
```

```
+---------------+---------------+------+------+---------+---------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+---------+
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |       |
| NON_UNIQUE    | bigint(21)    | YES  |      | NULL    |       |
| KEY_NAME      | varchar(64)   | YES  |      | NULL    |       |
| SEQ_IN_INDEX  | bigint(21)    | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(64)   | YES  |      | NULL    |       |
| SUB_PART      | bigint(21)    | YES  |      | NULL    |       |
| INDEX_COMMENT | varchar(2048) | YES  |      | NULL    |       |
| Expression    | varchar(64)   | YES  |      | NULL    |       |
| INDEX_ID      | bigint(21)    | YES  |      | NULL    |       |
| IS_VISIBLE    | varchar(64)   | YES  |      | NULL    |       |
| CLUSTERED     | varchar(64)   | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+---------+
12 rows in set (0.00 sec)
```

INDEX_ID 是TiDB 为每个索引分配的唯一ID。它可以与从另一个表或API 获得的 INDEX_ID 一起执行 join 操作。

例如，你可以在SLOW_QUERY 表 中获取某些慢查询所涉及的 TABLE_ID 和 INDEX_ID，然后使用以下 SQL 语句获取特定索引信息：

```
SELECT
 tidb_indexes.*
FROM
 tidb_indexes,
 tables
WHERE
  tidb_indexes.table_schema = tables.table_schema
 AND tidb_indexes.table_name = tidb_indexes.table_name
 AND tables.tidb_table_id = ?
 AND index_id = ?
```

TIDB_INDEXES 表中列的含义如下：

- TABLE_SCHEMA：索引所在表的所属数据库的名称。
- TABLE_NAME：索引所在表的名称。
- NON_UNIQUE：如果索引是唯一的，则为 0，否则为 1。
- KEY_NAME：索引的名称。如果索引是主键，则名称为 PRIMARY。
- SEQ_IN_INDEX：索引中列的顺序编号，从 1 开始。
- COLUMN_NAME：索引所在的列名。
- SUB_PART：索引前缀长度。如果列是部分被索引，则该值为被索引的字符数量，否则为 NULL。
- INDEX_COMMENT：创建索引时以 COMMENT 标注的注释。
- INDEX_ID：索引的 ID。
- IS_VISIBLE：索引是否可见。
- CLUSTERED：是否为聚簇索引。

4.8.18.2.44 TIDB_SERVERS_INFO

TIDB_SERVERS_INFO 表提供了TiDB 集群中TiDB 服务器的信息（即 tidb-server 进程）。

```
USE INFORMATION_SCHEMA;
DESC TIDB_SERVERS_INFO;
```

输出结果如下：

```
+---------------+--------------+------+------+---------+----------------+
| Field         | Type         | Null | Key  | Default | Extra          |
+---------------+--------------+------+------+---------+----------------+
| DDL_ID        | varchar(64)  | YES  |      | NULL    |                |
| IP            | varchar(64)  | YES  |      | NULL    |                |
| PORT          | bigint(21)   | YES  |      | NULL    |                |
| STATUS_PORT   | bigint(21)   | YES  |      | NULL    |                |
| LEASE         | varchar(64)  | YES  |      | NULL    |                |
| VERSION       | varchar(64)  | YES  |      | NULL    |                |
| GIT_HASH      | varchar(64)  | YES  |      | NULL    |                |
| BINLOG_STATUS | varchar(64)  | YES  |      | NULL    |                |
| LABELS        | varchar(128) | YES  |      | NULL    |                |
+---------------+--------------+------+------+---------+----------------+
9 rows in set (0.00 sec)
```

查看 TIDB_SERVERS_INFO 表的信息：

```
SELECT * FROM TIDB_SERVERS_INFO\G
```

输出结果如下：

```
*************************** 1. row ***************************
       DDL_ID: 771c169d-0a3a-48ea-a93c-a4d6751d3674
           IP: 0.0.0.0
```

```
         PORT: 4000
  STATUS_PORT: 10080
        LEASE: 45s
      VERSION: 5.7.25-TiDB-v6.5.0
     GIT_HASH: 827d8ff2d22ac4c93ae1b841b79d468211e1d393
BINLOG_STATUS: Off
       LABELS:
1 row in set (0.006 sec)
```

4.8.18.2.45 TIDB_TRX

TIDB_TRX 表提供了当前TiDB 节点上正在执行的事务的信息。

```
USE INFORMATION_SCHEMA;
DESC TIDB_TRX;
```

输出结果如下：

```
+--
    <-->-----------------------+-------------------------------------------------------------------------+------+-----------+-
    <-->
| Field                  | Type                                                                    |
    <--> Null | Key   | Default | Extra |
+--
    <-->-----------------------+-------------------------------------------------------------------------+------+-----------+-
    <-->
| ID                     | bigint(21) unsigned                                                     | NO
    <-->  | PRI  | NULL    |       |
| START_TIME             | timestamp(6)                                                            | YES
    <-->  |      | NULL    |       |
| CURRENT_SQL_DIGEST     | varchar(64)                                                             | YES
    <-->  |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                                                                   | YES
    <-->  |      | NULL    |       |
| STATE                  | enum('Idle','Running','LockWaiting','Committing','RollingBack') | YES
    <-->  |      | NULL    |       |
| WAITING_START_TIME     | timestamp(6)                                                            | YES
    <-->  |      | NULL    |       |
| MEM_BUFFER_KEYS        | bigint(64)                                                              | YES
    <-->  |      | NULL    |       |
| MEM_BUFFER_BYTES       | bigint(64)                                                              | YES
    <-->  |      | NULL    |       |
| SESSION_ID             | bigint(21) unsigned                                                     | YES
    <-->  |      | NULL    |       |
| USER                   | varchar(16)                                                             | YES
    <-->  |      | NULL    |       |
```

```
| DB                    | varchar(64)                                                
```

得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅Statement Summary Tables。

- TIDB_DECODE_SQL_DIGESTS 函数调用开销较大，如果对大量事务的信息调用该函数查询历史 SQL，可能查询耗时较长。如果集群规模较大、同一时刻并发运行的事务较多，请避免直接在查询 TIDB_TRX 全表的同时直接将该函数用于 ALL_SQL_DIGEST 列（即尽量避免 SELECT *, tidb_decode_sql_digests(all_sql_digests)FROM TIDB_TRX 这样的用法）。
- 目前 TIDB_TRX 表暂不支持显示TiDB 内部事务相关的信息。

示例

查看 TIDB_TRX 表中的信息：

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_TRX\G
```

输出结果如下：

```
*************************** 1. row ***************************
                   ID: 426789913200689153
           START_TIME: 2021-08-04 10:51:54.883000
    CURRENT_SQL_DIGEST: NULL
CURRENT_SQL_DIGEST_TEXT: NULL
                STATE: Idle
    WAITING_START_TIME: NULL
       MEM_BUFFER_KEYS: 1
      MEM_BUFFER_BYTES: 29
            SESSION_ID: 7
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","04
           <-> fa858fa491c62d194faec2ab427261cc7998b3f1ccf8f6844febca504cb5e9","
           <-> b83710fa8ab7df8504920e8569e48654f621cf828afbe7527fd003b79f48da9e"]
*************************** 2. row ***************************
                   ID: 426789921471332353
           START_TIME: 2021-08-04 10:52:26.433000
    CURRENT_SQL_DIGEST: 38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ?
                STATE: LockWaiting
    WAITING_START_TIME: 2021-08-04 10:52:35.106568
       MEM_BUFFER_KEYS: 0
      MEM_BUFFER_BYTES: 0
            SESSION_ID: 9
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
           <-> b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821"]
```

```
2 rows in set (0.01 sec)
```

此示例的查询结果表示： 当前节点有两个运行中的事务， 第一个事务正在闲置状态（STATE 为 Idle <-->，CURRENT_SQL_DIGEST 为 NULL），该事务已经执行过 3 条语句（ALL_SQL_DIGESTS 列表中有三条记录，分别为执行过的 3 条语句的 Digest）；第二个事务正在执行一条语句并正在等锁（STATE 为 LockWaiting，WAITING_START_TIME 显示了等锁开始的时间），该事务已经执行过两条语句，当前正在执行的语句形如 "update `t` set `v` = `v` + ? where `id` = ?"。

```
SELECT id, all_sql_digests, tidb_decode_sql_digests(all_sql_digests) AS all_sqls FROM
    <-> INFORMATION_SCHEMA.TIDB_TRX\G
```

输出结果如下：

```
*************************** 1. row ***************************
             id: 426789913200689153
all_sql_digests: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","04
    <-> fa858fa491c62d194faec2ab427261cc7998b3f1ccf8f6844febca504cb5e9","
    <-> b83710fa8ab7df8504920e8569e48654f621cf828afbe7527fd003b79f48da9e"]
       all_sqls: ["begin","insert into `t` values ( ... )","update `t` set `v` = `v` + ?"]
*************************** 2. row ***************************
             id: 426789921471332353
all_sql_digests: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
    <-> b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821"]
       all_sqls: ["begin","update `t` set `v` = `v` + ? where `id` = ?"]
```

此查询对 TIDB_TRX 表的 ALL_SQL_DIGESTS 列调用了TIDB_DECODE_SQL_DIGESTS 函数，将 SQL Digest 的数组通过系统内部的查询转换成归一化 SQL 语句的数组， 以便于直观地获取事务历史执行过的语句的信息。但是需要注意上述查询扫描了 TIDB_TRX 全表，并对每一行都调用了 TIDB_DECODE_SQL_DIGESTS 函数；而 TIDB_DECODE_SQL_DIGESTS 函数调用的开销很大，所以如果集群中并发事务数量较多，请尽量避免这种查询。

CLUSTER_TIDB_TRX

TIDB_TRX 表仅提供单个TiDB 节点中正在执行的事务信息。如果要查看整个集群上所有 TiDB 节点中正在执行的事务信息，需要查询 CLUSTER_TIDB_TRX 表。与 TIDB_TRX 表的查询结果相比，CLUSTER_TIDB_TRX 表的查询结果额外包含了 INSTANCE 字段。INSTANCE 字段展示了集群中各节点的IP 地址和端口，用于区分事务所在的TiDB 节点。

```
USE INFORMATION_SCHEMA;
DESC CLUSTER_TIDB_TRX;
```

输出结果如下：

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

```
+--
    <-->----------------------+-------------------------------------------------------------------------+-------+----------+-
    <-->
| INSTANCE                | varchar(64)                                                                | YES
    <-->  |       | NULL    |       |
| ID                      | bigint(21) unsigned                                                        | NO
    <-->  | PRI   | NULL    |       |
| START_TIME              | timestamp(6)                                                               | YES
    <-->  |       | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)                                                                | YES
    <-->  |       | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                                                                       | YES
    <-->  |       | NULL    |       |
| STATE                   | enum('Idle','Running','LockWaiting','Committing','RollingBack') | YES
    <-->  |       | NULL    |       |
| WAITING_START_TIME      | timestamp(6)                                                               | YES
    <-->  |       | NULL    |       |
| MEM_BUFFER_KEYS         | bigint(64)                                                                 | YES
    <-->  |       | NULL    |       |
| MEM_BUFFER_BYTES        | bigint(64)                                                                 | YES
    <-->  |       | NULL    |       |
| SESSION_ID              | bigint(21) unsigned                                                        | YES
    <-->  |       | NULL    |       |
| USER                    | varchar(16)                                                                | YES
    <-->  |       | NULL    |       |
| DB                      | varchar(64)                                                                | YES
    <-->  |       | NULL    |       |
| ALL_SQL_DIGESTS         | text                                                                       | YES
    <-->  |       | NULL    |       |
| RELATED_TABLE_IDS       | text                                                                       | YES
    <-->  |       | NULL    |       |
+--
    <-->----------------------+-------------------------------------------------------------------------+       +----------+-
    <-->                                                                                                                  -------
14 rows in set (0.00 sec)
```

4.8.18.2.46 TIFLASH_REPLICA

TIFLASH_REPLICA 表提供了有关可用的TiFlash 副本的信息。

```
USE INFORMATION_SCHEMA;
DESC TIFLASH_REPLICA;
```

输出结果如下：

```
+-----------------+-------------+------+------+---------+--------------+
```

```
| Field           | Type        | Null | Key  | Default | Extra |
+-----------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA    | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID        | bigint(21)  | YES  |      | NULL    |       |
| REPLICA_COUNT   | bigint(64)  | YES  |      | NULL    |       |
| LOCATION_LABELS | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE       | tinyint(1)  | YES  |      | NULL    |       |
| PROGRESS        | double      | YES  |      | NULL    |       |
+-----------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

TIFLASH_REPLICA 表中各列的字段含义如下：

- TABLE_SCHEMA：表所属的数据库的名称。
- TABLE_NAME：表的名称。
- TABLE_ID：表的内部 ID，该ID 在一个TiDB 集群内部唯一。
- REPLICA_COUNT：TiFlash 副本数。
- LOCATION_LABELS：设置 TiFlash 副本时设置的 LocationLabelList。
- AVAILABLE：表的 TiFlash 副本是否可用。1 代表可用，TiDB 优化器将依据查询代价智能选择下推查询到TiKV 或TiFlash；0 代表不可用，TiDB 将不会下推查询到TiFlash。副本状态变为可用之后就不再改变。
- PROGRESS：TiFlash 副本同步进度，范围是 [0, 1]，精确到小数点后两位，刷新的精度为分钟级。当 AVAILABLE 字段为 1 时，如果 PROGRESS 小于 1，表示TiFlash 副本落后TiKV 较多，下推到TiFlash 的查询很可能会因为等待数据同步超时而失败。

4.8.18.2.47 TIFLASH_SEGMENTS

> **警告：**
>
> 请不要在生产环境中使用该系统表，因为该表字段信息尚未固定下来，可能会在TiDB 未来版本中发生变化。

TIFLASH_SEGMENTS 表提供了TiFlash 内部数据表的统计信息。

```
USE information_schema;
DESC tiflash_segments;
```

```
+------------------------------+-------------+------+------+---------+-------+
| Field                        | Type        | Null | Key  | Default | Extra |
+------------------------------+-------------+------+------+---------+-------+
| DATABASE                     | varchar(64) | YES  |      | NULL    |       |
| TABLE                        | varchar(64) | YES  |      | NULL    |       |
| TIDB_DATABASE                | varchar(64) | YES  |      | NULL    |       |
```

```
| TIDB_TABLE                    | varchar(64) | YES  |     | NULL    |       |
| TABLE_ID                      | bigint(64)  | YES  |     | NULL    |       |
| IS_TOMBSTONE                  | bigint(64)  | YES  |     | NULL    |       |
| SEGMENT_ID                    | bigint(64)  | YES  |     | NULL    |       |
| RANGE                         | varchar(64) | YES  |     | NULL    |       |
| EPOCH                         | bigint(64)  | YES  |     | NULL    |       |
| ROWS                          | bigint(64)  | YES  |     | NULL    |       |
| SIZE                          | bigint(64)  | YES  |     | NULL    |       |
| DELTA_RATE                    | double      | YES  |     | NULL    |       |
| DELTA_MEMTABLE_ROWS           | bigint(64)  | YES  |     | NULL    |       |
| DELTA_MEMTABLE_SIZE           | bigint(64)  | YES  |     | NULL    |       |
| DELTA_MEMTABLE_COLUMN_FILES   | bigint(64)  | YES  |     | NULL    |       |
| DELTA_MEMTABLE_DELETE_RANGES  | bigint(64)  | YES  |     | NULL    |       |
| DELTA_PERSISTED_PAGE_ID       | bigint(64)  | YES  |     | NULL    |       |
| DELTA_PERSISTED_ROWS          | bigint(64)  | YES  |     | NULL    |       |
| DELTA_PERSISTED_SIZE          | bigint(64)  | YES  |     | NULL    |       |
| DELTA_PERSISTED_COLUMN_FILES  | bigint(64)  | YES  |     | NULL    |       |
| DELTA_PERSISTED_DELETE_RANGES | bigint(64)  | YES  |     | NULL    |       |
| DELTA_CACHE_SIZE              | bigint(64)  | YES  |     | NULL    |       |
| DELTA_INDEX_SIZE              | bigint(64)  | YES  |     | NULL    |       |
| STABLE_PAGE_ID                | bigint(64)  | YES  |     | NULL    |       |
| STABLE_ROWS                   | bigint(64)  | YES  |     | NULL    |       |
| STABLE_SIZE                   | bigint(64)  | YES  |     | NULL    |       |
| STABLE_DMFILES                | bigint(64)  | YES  |     | NULL    |       |
| STABLE_DMFILES_ID_0           | bigint(64)  | YES  |     | NULL    |       |
| STABLE_DMFILES_ROWS           | bigint(64)  | YES  |     | NULL    |       |
| STABLE_DMFILES_SIZE           | bigint(64)  | YES  |     | NULL    |       |
| STABLE_DMFILES_SIZE_ON_DISK   | bigint(64)  | YES  |     | NULL    |       |
| STABLE_DMFILES_PACKS          | bigint(64)  | YES  |     | NULL    |       |
| TIFLASH_INSTANCE              | varchar(64) | YES  |     | NULL    |       |
+-------------------------------+-------------+------+-----+---------+---------------+
33 rows in set (0.00 sec)
```

TIFLASH_SEGMENTS 表中各列的字段含义如下：

- DATABASE：Segment 所属表在TiFlash 内部所属的数据库的名称。
- TABLE：Segment 所属表在TiFlash 内部的名称。
- TIDB_DATABASE：Segment 所属表所属的数据库的名称。
- TIDB_TABLE：Segment 所属表的名称。
- TABLE_ID：Segment 所属表的内部ID，该ID 在一个TiDB 集群内部唯一。
- IS_TOMBSTONE：表示该 Segment 所属表是否处于待回收状态，为 1 表示该表处于待回收状态，为 0 则表示处于正常状态。
- SEGMENT_ID：Segment ID，该ID 在每张表内部唯一。
- RANGE：Segment 包含数据的范围。
- EPOCH：Segment 的更新版本号，每个 Segment 的版本号单调递增。

- ROWS：Segment 的数据总行数。
- SIZE：Segment 的数据总大小（单位：字节）。
- DELTA_RATE：Delta 层的数据总行数占 Segment 数据总行数的比例。
- DELTA_MEMTABLE_ROWS：Delta 层缓存中的数据总行数。
- DELTA_MEMTABLE_SIZE：Delta 层缓存中的数据总大小（单位：字节）。
- DELTA_MEMTABLE_COLUMN_FILES：Delta 层缓存中 Column File 数量。
- DELTA_MEMTABLE_DELETE_RANGES：Delta 层缓存中 Delete Range 数量。
- DELTA_PERSISTED_PAGE_ID：Delta 层数据在磁盘上存储的ID。
- DELTA_PERSISTED_ROWS：Delta 层已落盘的数据总行数。
- DELTA_PERSISTED_SIZE：Delta 层已落盘的数据总大小（单位：字节）。
- DELTA_PERSISTED_COLUMN_FILES：Delta 层已落盘 Column File 数量。
- DELTA_PERSISTED_DELETE_RANGES：Delta 层已落盘 Delete Range 数量。
- DELTA_CACHE_SIZE：Delta 层缓存大小（单位：字节）。
- DELTA_INDEX_SIZE：Delta 层索引大小（单位：字节）。
- STABLE_PAGE_ID：Stable 层数据在磁盘上存储的标识。
- STABLE_ROWS：Stable 层的数据总行数。
- STABLE_SIZE：Stable 层的数据总大小（单位：字节）。
- STABLE_DMFILES：Stable 层 DMFile 数量。
- STABLE_DMFILES_ID_0：Stable 层第一个 DMFile 在磁盘上存储的ID。
- STABLE_DMFILES_ROWS：Stable 层 DMFile 中的数据总行数。
- STABLE_DMFILES_SIZE：Stable 层 DMFile 中的数据总大小（单位：字节）。
- STABLE_DMFILES_SIZE_ON_DISK：Stable 层 DMFile 中的数据在磁盘占据的空间大小（单位：字节）。
- STABLE_DMFILES_PACKS：Stable 层 DMFile 中包含的 Pack 数量。
- TIFLASH_INSTANCE：TiFlash 实例地址。

4.8.18.2.48 TIFLASH_TABLES

> **警告：**
>
> 请不要在生产环境中使用该系统表，因为该表字段信息尚未固定下来，可能会在TiDB 未来版本中发生变化。

TIFLASH_TABLES 表提供了TiFlash 内部数据表的统计信息。

```
USE information_schema;
DESC tiflash_tables;
```

```
+-----------------------------------------+--------------+------+------+---------+----------------+
| Field                                   | Type         | Null | Key  | Default | Extra          |
+-----------------------------------------+--------------+------+------+---------+----------------+
| DATABASE                                | varchar(64)  | YES  |      | NULL    |                |
| TABLE                                   | varchar(64)  | YES  |      | NULL    |                |
| TIDB_DATABASE                           | varchar(64)  | YES  |      | NULL    |                |
```

```
| TIDB_TABLE                                | varchar(64)  | YES  |      | NULL    |       |
| TABLE_ID                                  | bigint(64)   | YES  |      | NULL    |       |
| IS_TOMBSTONE                              | bigint(64)   | YES  |      | NULL    |       |
| SEGMENT_COUNT                             | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_ROWS                                | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_SIZE                                | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_DELETE_RANGES                       | bigint(64)   | YES  |      | NULL    |       |
| DELTA_RATE_ROWS                           | double       | YES  |      | NULL    |       |
| DELTA_RATE_SEGMENTS                       | double       | YES  |      | NULL    |       |
| DELTA_PLACED_RATE                         | double       | YES  |      | NULL    |       |
| DELTA_CACHE_SIZE                          | bigint(64)   | YES  |      | NULL    |       |
| DELTA_CACHE_RATE                          | double       | YES  |      | NULL    |       |
| DELTA_CACHE_WASTED_RATE                   | double       | YES  |      | NULL    |       |
| DELTA_INDEX_SIZE                          | bigint(64)   | YES  |      | NULL    |       |
| AVG_SEGMENT_ROWS                          | double       | YES  |      | NULL    |       |
| AVG_SEGMENT_SIZE                          | double       | YES  |      | NULL    |       |
| DELTA_COUNT                               | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_DELTA_ROWS                          | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_DELTA_SIZE                          | bigint(64)   | YES  |      | NULL    |       |
| AVG_DELTA_ROWS                            | double       | YES  |      | NULL    |       |
| AVG_DELTA_SIZE                            | double       | YES  |      | NULL    |       |
| AVG_DELTA_DELETE_RANGES                   | double       | YES  |      | NULL    |       |
| STABLE_COUNT                              | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_STABLE_ROWS                         | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_STABLE_SIZE                         | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_STABLE_SIZE_ON_DISK                 | bigint(64)   | YES  |      | NULL    |       |
| AVG_STABLE_ROWS                           | double       | YES  |      | NULL    |       |
| AVG_STABLE_SIZE                           | double       | YES  |      | NULL    |       |
| TOTAL_PACK_COUNT_IN_DELTA                 | bigint(64)   | YES  |      | NULL    |       |
| MAX_PACK_COUNT_IN_DELTA                   | bigint(64)   | YES  |      | NULL    |       |
| AVG_PACK_COUNT_IN_DELTA                   | double       | YES  |      | NULL    |       |
| AVG_PACK_ROWS_IN_DELTA                    | double       | YES  |      | NULL    |       |
| AVG_PACK_SIZE_IN_DELTA                    | double       | YES  |      | NULL    |       |
| TOTAL_PACK_COUNT_IN_STABLE                | bigint(64)   | YES  |      | NULL    |       |
| AVG_PACK_COUNT_IN_STABLE                  | double       | YES  |      | NULL    |       |
| AVG_PACK_ROWS_IN_STABLE                   | double       | YES  |      | NULL    |       |
| AVG_PACK_SIZE_IN_STABLE                   | double       | YES  |      | NULL    |       |
| STORAGE_STABLE_NUM_SNAPSHOTS              | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_STABLE_OLDEST_SNAPSHOT_LIFETIME   | double       | YES  |      | NULL    |       |
| STORAGE_STABLE_OLDEST_SNAPSHOT_THREAD_ID  | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_STABLE_OLDEST_SNAPSHOT_TRACING_ID | varchar(128) | YES  |      | NULL    |       |
| STORAGE_DELTA_NUM_SNAPSHOTS               | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_DELTA_OLDEST_SNAPSHOT_LIFETIME    | double       | YES  |      | NULL    |       |
| STORAGE_DELTA_OLDEST_SNAPSHOT_THREAD_ID   | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_DELTA_OLDEST_SNAPSHOT_TRACING_ID  | varchar(128) | YES  |      | NULL    |       |
```

```
| STORAGE_META_NUM_SNAPSHOTS                | bigint(64)    | YES  |      | NULL    |       |
| STORAGE_META_OLDEST_SNAPSHOT_LIFETIME     | double        | YES  |      | NULL    |       |
| STORAGE_META_OLDEST_SNAPSHOT_THREAD_ID    | bigint(64)    | YES  |      | NULL    |       |
| STORAGE_META_OLDEST_SNAPSHOT_TRACING_ID   | varchar(128)  | YES  |      | NULL    |       |
| BACKGROUND_TASKS_LENGTH                   | bigint(64)    | YES  |      | NULL    |       |
| TIFLASH_INSTANCE                          | varchar(64)   | YES  |      | NULL    |       |
+-------------------------------------------+---------------+------+------+---------+---------------- +
54 rows in set (0.00 sec)
```

TIFLASH_TABLES 表中各列的字段含义如下：

- DATABASE：表在 TiFlash 内部所属的数据库的名称。
- TABLE：表在 TiFlash 内部的名称。
- TIDB_DATABASE：表所属的数据库的名称。
- TIDB_TABLE：表的名称。
- TABLE_ID：表的内部 ID，该ID 在一个TiDB 集群内部唯一。
- IS_TOMBSTONE：表示该表是否处于待回收状态，为 1 表示该表处于待回收状态，为 0 则表示处于正常状态。
- SEGMENT_COUNT：表的 Segment 数量，其中 Segment 为TiFlash 内部管理数据的单元。
- TOTAL_ROWS：表的总行数。
- TOTAL_SIZE：表的总大小（单位：字节）。
- TOTAL_DELETE_RANGES：表当前 Delete Range 的数量。
- DELTA_RATE_ROWS：表 Delta 层数据总行数占该表总行数的比例。
- DELTA_RATE_SEGMENTS：表所有 Segment 中包含非空 Delta 层的 Segment 的比例。
- DELTA_PLACED_RATE：表 Delta 层数据行中已经完成Index 构建的比例。
- DELTA_CACHE_SIZE：表 Delta 层缓存大小（单位：字节）。
- DELTA_CACHE_RATE：表 Delta 层数据的缓存比例。
- DELTA_CACHE_WASTED_RATE：表 Delta 层缓存中无效数据的比例。
- DELTA_INDEX_SIZE：Delta Index 占用的内存大小（单位：字节）。
- AVG_SEGMENT_ROWS：表所有 Segment 的平均行数。
- AVG_SEGMENT_SIZE：表所有 Segment 的平均大小（单位：字节）。
- DELTA_COUNT：表中包含 Delta 层的 Segment 的个数。
- TOTAL_DELTA_ROWS：所有 Delta 层的数据总行数。
- TOTAL_DELTA_SIZE：所有 Delta 层的数据总大小（单位：字节）。
- AVG_DELTA_ROWS：所有 Delta 层的平均数据行数。
- AVG_DELTA_SIZE：所有 Delta 层的平均数据大小（单位：字节）。
- AVG_DELTA_DELETE_RANGES：所有 Delta 层的平均 Delete Range 操作个数。
- STABLE_COUNT：表中包含非空 Stable 层的 Segment 的个数。
- TOTAL_STABLE_ROWS：所有 Stable 层的数据总行数。
- TOTAL_STABLE_SIZE：所有 Stable 层的数据总大小（单位：字节）。
- TOTAL_STABLE_SIZE_ON_DISK：所有 Stable 层的数据在磁盘占据的空间大小（单位：字节）。
- AVG_STABLE_ROWS：所有 Stable 层的平均数据行数。
- AVG_STABLE_SIZE：所有 Stable 层的平均数据大小（单位：字节）。
- TOTAL_PACK_COUNT_IN_DELTA：所有 Delta 层的 Column File 总数量。
- MAX_PACK_COUNT_IN_DELTA：单个 Delta 层包含 Column File 数量的最大值。

- AVG_PACK_COUNT_IN_DELTA：所有 Delta 层的平均 Column File 数量。
- AVG_PACK_ROWS_IN_DELTA：Delta 层所有 Column File 的平均数据行数。
- AVG_PACK_SIZE_IN_DELTA：Delta 层所有 Column File 的平均数据大小（单位：字节）。
- TOTAL_PACK_COUNT_IN_STABLE：所有 Stable 层的 Pack 总数量。
- AVG_PACK_COUNT_IN_STABLE：所有 Stable 层的平均 Pack 数量。
- AVG_PACK_ROWS_IN_STABLE：Stable 层所有 Pack 的平均数据行数。
- AVG_PACK_SIZE_IN_STABLE：Stable 层所有 Pack 的平均数据大小（单位：字节）。
- STORAGE_STABLE_NUM_SNAPSHOTS：Stable 层 Snapshot 数量。
- STORAGE_STABLE_OLDEST_SNAPSHOT_LIFETIME：Stable 层最早 Snapshot 的持续时间（单位：秒）。
- STORAGE_STABLE_OLDEST_SNAPSHOT_THREAD_ID：Stable 层最早 Snapshot 对应的线程ID。
- STORAGE_STABLE_OLDEST_SNAPSHOT_TRACING_ID：Stable 层最早 Snapshot 对应的跟踪ID。
- STORAGE_DELTA_NUM_SNAPSHOTS：Delta 层 Snapshot 数量。
- STORAGE_DELTA_OLDEST_SNAPSHOT_LIFETIME：Delta 层最早 Snapshot 的持续时间（单位：秒）。
- STORAGE_DELTA_OLDEST_SNAPSHOT_THREAD_ID：Delta 层最早 Snapshot 对应的线程ID。
- STORAGE_DELTA_OLDEST_SNAPSHOT_TRACING_ID：Delta 层最早 Snapshot 对应的跟踪ID。
- STORAGE_META_NUM_SNAPSHOTS：Meta 元信息 Snapshot 数量。
- STORAGE_META_OLDEST_SNAPSHOT_LIFETIME：Meta 元信息最早 Snapshot 的持续时间。（单位：秒）
- STORAGE_META_OLDEST_SNAPSHOT_THREAD_ID：Meta 元信息最早 Snapshot 对应的线程ID。
- STORAGE_META_OLDEST_SNAPSHOT_TRACING_ID：Meta 元信息最早 Snapshot 对应的跟踪ID。
- BACKGROUND_TASKS_LENGTH：后台任务队列长度。
- TIFLASH_INSTANCE：TiFlash 实例地址。

4.8.18.2.49 TIKV_REGION_PEERS

TIKV_REGION_PEERS 表提供了TiKV 中单个Region 节点的详细信息，比如它是一个learner 还是一个leader。

```
USE INFORMATION_SCHEMA;
DESC TIKV_REGION_PEERS;
```

输出结果如下：

```
+--------------+-------------+------+------+---------+--------------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+--------------+
| REGION_ID    | bigint(21)  | YES  |      | NULL    |       |
| PEER_ID      | bigint(21)  | YES  |      | NULL    |       |
| STORE_ID     | bigint(21)  | YES  |      | NULL    |       |
| IS_LEARNER   | tinyint(1)  | NO   |      | 0       |       |
| IS_LEADER    | tinyint(1)  | NO   |      | 0       |       |
| STATUS       | varchar(10) | YES  |      | 0       |       |
| DOWN_SECONDS | bigint(21)  | YES  |      | 0       |       |
+--------------+-------------+------+------+---------+--------------+
7 rows in set (0.01 sec)
```

例如，使用以下 SQL 语句，你可以查询 WRITTEN_BYTES 最大的前 3 个 Region 所在的TiKV 地址：

```
SELECT
```

```
  address,
  tikv.address,
  region.region_id
FROM
  TIKV_STORE_STATUS tikv,
  TIKV_REGION_PEERS peer,
  (SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3) region
WHERE
  region.region_id = peer.region_id
  AND peer.is_leader = 1
  AND peer.store_id = tikv.store_id;
```

TIKV_REGION_PEERS 表各列含义如下：

- REGION_ID：REGION 的 ID。
- PEER_ID：REGION 中对应的副本 Peer 的 ID。
- STORE_ID：REGION 所在 TiKV Store 的 ID。
- IS_LEARNER：Peer 是否是LEARNER。
- IS_LEADER：Peer 是否是LEADER。
- STATUS：Peer 的状态，一共有 3 种状态：
  - PENDING：暂时不可用状态。
  - DOWN：下线转态，该 Peer 不再提供服务。
  - NORMAL：正常状态。
- DOWN_SECONDS：处于下线状态的时间，单位是秒。

4.8.18.2.50 TIKV_REGION_STATUS

TIKV_REGION_STATUS 表通过 PD 的 API 展示TiKV Region 的基本信息，比如Region ID、开始和结束键值以及读写流量。

```
USE information_schema;
DESC tikv_region_status;
```

```
+--------------------------+-------------+------+------+---------+--------------+
| Field                    | Type        | Null | Key  | Default | Extra |
+--------------------------+-------------+------+------+---------+--------------+
| REGION_ID                | bigint(21)  | YES  |      | NULL    |       |
| START_KEY                | text        | YES  |      | NULL    |       |
| END_KEY                  | text        | YES  |      | NULL    |       |
| TABLE_ID                 | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME                  | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME               | varchar(64) | YES  |      | NULL    |       |
| IS_INDEX                 | tinyint(1)  | NO   |      | 0       |       |
| INDEX_ID                 | bigint(21)  | YES  |      | NULL    |       |
| INDEX_NAME               | varchar(64) | YES  |      | NULL    |       |
```

```
| EPOCH_CONF_VER             | bigint(21)   | YES  |     | NULL    |       |
| EPOCH_VERSION              | bigint(21)   | YES  |     | NULL    |       |
| WRITTEN_BYTES              | bigint(21)   | YES  |     | NULL    |       |
| READ_BYTES                 | bigint(21)   | YES  |     | NULL    |       |
| APPROXIMATE_SIZE           | bigint(21)   | YES  |     | NULL    |       |
| APPROXIMATE_KEYS           | bigint(21)   | YES  |     | NULL    |       |
| REPLICATIONSTATUS_STATE    | varchar(64)  | YES  |     | NULL    |       |
| REPLICATIONSTATUS_STATEID  | bigint(21)   | YES  |     | NULL    |       |
+----------------------------+--------------+------+-----+---------+-------+
17 rows in set (0.00 sec)
```

TIKV_REGION_STATUS 表中列的含义如下：

- REGION_ID：Region 的 ID。
- START_KEY：Region 的起始 key 的值。
- END_KEY：Region 的末尾 key 的值。
- TABLE_ID：Region 所属的表的ID。
- DB_NAME：TABLE_ID 所属的数据库的名称。
- TABLE_NAME：Region 所属的表的名称。
- IS_INDEX：Region 数据是否是索引，0 代表不是索引，1 代表是索引。如果当前Region 同时包含表数据和索引数据，会有多行记录，IS_INDEX 分别是 0 和 1。
- INDEX_ID：Region 所属的索引的ID。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- INDEX_NAME：Region 所属的索引的名称。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- EPOCH_CONF_VER：Region 的配置的版本号，在增加或减少peer 时版本号会递增。
- EPOCH_VERSION：Region 的当前版本号，在分裂或合并时版本号会递增。
- WRITTEN_BYTES：已经往 Region 写入的数据量(bytes)。
- READ_BYTES：已经从 Region 读取的数据量(bytes)。
- APPROXIMATE_SIZE：Region 的近似数据量(MB)。
- APPROXIMATE_KEYS：Region 中 key 的近似数量。
- REPLICATIONSTATUS_STATE：Region 当前的同步状态，可能为UNKNOWN/SIMPLE_MAJORITY/INTEGRITY_OVER_LABEL <--> 其中一种状态。
- REPLICATIONSTATUS_STATEID：REPLICATIONSTATUS_STATE 对应的标识符。

通过在 EPOCH_CONF_VER、WRITTEN_BYTES 和 READ_BYTES 列上执行 ORDER BY X LIMIT Y 操作，你可以在pd-ctl 中实现 top confver、top read 和 top write 操作。

你可以使用以下 SQL 语句查询写入数据最多的前 3 个 Region：

```
SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3;
```

4.8.18.2.51 TIKV_STORE_STATUS

TIKV_STORE_STATUS 表通过 PD 的 API 显示了TiKV 节点的一些基本信息，例如在集群中分配的ID、地址和端口、状态、容量以及当前节点的Region leader 的数量。

```
USE INFORMATION_SCHEMA;
DESC TIKV_STORE_STATUS;
```

输出结果如下：

```
+-------------------+-------------+------+------+---------+---------------+
| Field             | Type        | Null | Key  | Default | Extra |
+-------------------+-------------+------+------+---------+---------------+
| STORE_ID          | bigint(21)  | YES  |      | NULL    |       |
| ADDRESS           | varchar(64) | YES  |      | NULL    |       |
| STORE_STATE       | bigint(21)  | YES  |      | NULL    |       |
| STORE_STATE_NAME  | varchar(64) | YES  |      | NULL    |       |
| LABEL             | json        | YES  |      | NULL    |       |
| VERSION           | varchar(64) | YES  |      | NULL    |       |
| CAPACITY          | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE         | varchar(64) | YES  |      | NULL    |       |
| LEADER_COUNT      | bigint(21)  | YES  |      | NULL    |       |
| LEADER_WEIGHT     | double      | YES  |      | NULL    |       |
| LEADER_SCORE      | double      | YES  |      | NULL    |       |
| LEADER_SIZE       | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT      | bigint(21)  | YES  |      | NULL    |       |
| REGION_WEIGHT     | double      | YES  |      | NULL    |       |
| REGION_SCORE      | double      | YES  |      | NULL    |       |
| REGION_SIZE       | bigint(21)  | YES  |      | NULL    |       |
| START_TS          | datetime    | YES  |      | NULL    |       |
| LAST_HEARTBEAT_TS | datetime    | YES  |      | NULL    |       |
| UPTIME            | varchar(64) | YES  |      | NULL    |       |
+-------------------+-------------+------+------+---------+---------------+
19 rows in set (0.00 sec)
```

TIKV_STORE_STATUS 表中列的含义如下：

- STORE_ID：Store 的 ID。
- ADDRESS：Store 的地址。
- STORE_STATE：Store 状态的标识符，与 STORE_STATE_NAME 相对应。
- STORE_STATE_NAME：Store 状态的名字，为 Up、Offline、Tombstone 中的一种。
- LABEL：给 Store 设置的标签。
- VERSION：Store 的版本号。
- CAPACITY：Store 的存储容量。
- AVAILABLE：Store 的剩余存储空间。
- LEADER_COUNT：Store 上的 leader 的数量。
- LEADER_WEIGHT：Store 的 leader 权重。
- LEADER_SCORE：Store 的 leader 评分。
- LEADER_SIZE：Store 上的所有 leader 的近似总数据量(MB)。
- REGION_COUNT：Store 上的 Region 总数。
- REGION_WEIGHT：Store 的 Region 权重。
- REGION_SCORE：Store 的 Region 评分。
- REGION_SIZE：Store 上的所有 Region 的近似总数据量(MB)。
- START_TS：Store 启动时的时间戳。

- LAST_HEARTBEAT_TS：Store 上次发出心跳的时间戳。
- UPTIME：Store 启动以来的总时间。

4.8.18.2.52 USER_ATTRIBUTES

USER_ATTRIBUTES 表提供了用户的注释和属性。该表的数据根据 mysql.user 系统表生成。

```
USE information_schema;
DESC user_attributes;
```

```
+-----------+--------------+------+------+---------+-------------+
| Field     | Type         | Null | Key  | Default | Extra       |
+-----------+--------------+------+------+---------+-------------+
| USER      | varchar(32)  | NO   |      | NULL    |             |
| HOST      | varchar(255) | NO   |      | NULL    |             |
| ATTRIBUTE | longtext     | YES  |      | NULL    |             |
+-----------+--------------+------+------+---------+-------------+
3 rows in set (0.00 sec)
```

USER_ATTRIBUTES 表中列的含义如下：

- USER：用户名。
- HOST：用户可用于连接的主机。百分号（%）表示主机名不受限制。
- ATTRIBUTE：通过CREATE USER 或ALTER USER 语句设置的用户相关的注释和属性。

示例：

```
CREATE USER testuser1 COMMENT 'This user is created only for test';
CREATE USER testuser2 ATTRIBUTE '{"email": "user@pingcap.com"}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| root      | %    | NULL                                                |
| testuser1 | %    | {"comment": "This user is created only for test"} |
| testuser2 | %    | {"email": "user@pingcap.com"}                       |
+-----------+------+-----------------------------------------------------+
3 rows in set (0.00 sec)
```

4.8.18.2.53 USER_PRIVILEGES

USER_PRIVILEGES 表提供了关于全局权限的信息。该表的数据根据 mysql.user 系统表生成。

```
USE INFORMATION_SCHEMA;
DESC USER_PRIVILEGES;
```

输出结果如下：

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

查看 USER_PRIVILEGES 表的信息：

```
SELECT * FROM USER_PRIVILEGES;
```

输出结果如下：

```
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | SELECT                  | YES          |
| 'root'@'%' | def           | INSERT                  | YES          |
| 'root'@'%' | def           | UPDATE                  | YES          |
| 'root'@'%' | def           | DELETE                  | YES          |
| 'root'@'%' | def           | CREATE                  | YES          |
| 'root'@'%' | def           | DROP                    | YES          |
| 'root'@'%' | def           | PROCESS                 | YES          |
| 'root'@'%' | def           | REFERENCES              | YES          |
| 'root'@'%' | def           | ALTER                   | YES          |
| 'root'@'%' | def           | SHOW DATABASES          | YES          |
| 'root'@'%' | def           | SUPER                   | YES          |
| 'root'@'%' | def           | EXECUTE                 | YES          |
| 'root'@'%' | def           | INDEX                   | YES          |
| 'root'@'%' | def           | CREATE USER             | YES          |
| 'root'@'%' | def           | CREATE TABLESPACE       | YES          |
| 'root'@'%' | def           | TRIGGER                 | YES          |
| 'root'@'%' | def           | CREATE VIEW             | YES          |
| 'root'@'%' | def           | SHOW VIEW               | YES          |
| 'root'@'%' | def           | CREATE ROLE             | YES          |
| 'root'@'%' | def           | DROP ROLE               | YES          |
| 'root'@'%' | def           | CREATE TEMPORARY TABLES | YES          |
| 'root'@'%' | def           | LOCK TABLES             | YES          |
| 'root'@'%' | def           | CREATE ROUTINE          | YES          |
| 'root'@'%' | def           | ALTER ROUTINE           | YES          |
| 'root'@'%' | def           | EVENT                   | YES          |
| 'root'@'%' | def           | SHUTDOWN                | YES          |
```

```
| 'root'@'%' | def           | RELOAD                  | YES          |
| 'root'@'%' | def           | FILE                    | YES          |
| 'root'@'%' | def           | CONFIG                  | YES          |
| 'root'@'%' | def           | REPLICATION CLIENT      | YES          |
| 'root'@'%' | def           | REPLICATION SLAVE       | YES          |
+------------+---------------+-------------------------+--------------+
31 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。

4.8.18.2.54 VARIABLES_INFO

VARIABLES_INFO 可用于查看当前TiDB 集群或实例的系统变量默认值、当前值以及作用域等信息。

```
USE information_schema;
DESC variables_info;
```

```
+-----------------+---------------------+------+------+---------+-------------+
| Field           | Type                | Null | Key  | Default | Extra       |
+-----------------+---------------------+------+------+---------+-------------+
| VARIABLE_NAME   | varchar(64)         | NO   |      | NULL    |             |
| VARIABLE_SCOPE  | varchar(64)         | NO   |      | NULL    |             |
| DEFAULT_VALUE   | varchar(64)         | NO   |      | NULL    |             |
| CURRENT_VALUE   | varchar(64)         | NO   |      | NULL    |             |
| MIN_VALUE       | bigint(64)          | YES  |      | NULL    |             |
| MAX_VALUE       | bigint(64) unsigned | YES  |      | NULL    |             |
| POSSIBLE_VALUES | varchar(256)        | YES  |      | NULL    |             |
| IS_NOOP         | varchar(64)         | NO   |      | NULL    |             |
+-----------------+---------------------+------+------+---------+-------------+
8 rows in set (0.00 sec)
```

```
SELECT * FROM variables_info ORDER BY variable_name LIMIT 3;
```

```
+--
    <-> --------------------------------+---------------+--------------+--------------+-----------+----------
    <->
| VARIABLE_NAME                       | VARIABLE_SCOPE | DEFAULT_VALUE | CURRENT_VALUE | MIN_VALUE
    <-> | MAX_VALUE | POSSIBLE_VALUES | IS_NOOP |
+--
    <-> --------------------------------+---------------+--------------+--------------+-----------+----------
    <->
```

```
| allow_auto_random_explicit_insert | SESSION,GLOBAL | OFF            | OFF            |       NULL
    <-> |       NULL | NULL              | NO       |
| auto_increment_increment          | SESSION,GLOBAL | 1              | 1              |          1
    <-> |      65535 | NULL              | NO       |
| auto_increment_offset             | SESSION,GLOBAL | 1              | 1              |          1
    <-> |      65535 | NULL              | NO       |
+--
    <-> --------------------------------+----------------+---------------+---------------+-----------+----------
    <->
3 rows in set (0.01 sec)
```

VARIABLES_INFO 表中列的含义如下：

- VARIABLE_NAME：系统变量名称。
- VARIABLE_SCOPE：系统变量的作用域。SESSION 表示当前 session 可见；INSTANCE 表示当前TiDB 实例可见；GLOBAL 表示集群内可见。
- DEFAULT_VALUE：系统变量的默认值。
- CURRENT_VALUE：系统变量的当前值。如果应用范围中包含 SESSION，则显示当前 session 的值。
- MIN_VALUE：数值类型的系统变量允许的最小值。如果变量值为非数值类型，则为 NULL。
- MAX_VALUE：数值类型的系统变量允许的最大值。如果变量值为非数值类型，则为 NULL。
- POSSIBLE_VALUES：系统变量所有可能的值。如果变量值不可枚举，则为 NULL。
- IS_NOOP：是否为 noop 的系统变量。

4.8.18.2.55 VIEWS

VIEWS 表提供了关于 SQL 视图的信息。

```
USE INFORMATION_SCHEMA;
DESC VIEWS;
```

输出结果如下：

```
+----------------------+--------------+------+------+---------+----------------+
| Field                | Type         | Null | Key  | Default | Extra          |
+----------------------+--------------+------+------+---------+----------------+
| TABLE_CATALOG        | varchar(512) | NO   |      | NULL    |                |
| TABLE_SCHEMA         | varchar(64)  | NO   |      | NULL    |                |
| TABLE_NAME           | varchar(64)  | NO   |      | NULL    |                |
| VIEW_DEFINITION      | longtext     | NO   |      | NULL    |                |
| CHECK_OPTION         | varchar(8)   | NO   |      | NULL    |                |
| IS_UPDATABLE         | varchar(3)   | NO   |      | NULL    |                |
| DEFINER              | varchar(77)  | NO   |      | NULL    |                |
| SECURITY_TYPE        | varchar(7)   | NO   |      | NULL    |                |
| CHARACTER_SET_CLIENT | varchar(32)  | NO   |      | NULL    |                |
| COLLATION_CONNECTION | varchar(32)  | NO   |      | NULL    |                |
+----------------------+--------------+------+------+---------+----------------+
10 rows in set (0.00 sec)
```

创建视图并查询 VIEWS 表信息：

```
CREATE VIEW test.v1 AS SELECT 1;
SELECT * FROM VIEWS\G
```

输出结果如下：

```
*************************** 1. row ***************************
       TABLE_CATALOG: def
        TABLE_SCHEMA: test
          TABLE_NAME: v1
     VIEW_DEFINITION: SELECT 1
        CHECK_OPTION: CASCADED
        IS_UPDATABLE: NO
             DEFINER: root@127.0.0.1
       SECURITY_TYPE: DEFINER
CHARACTER_SET_CLIENT: utf8mb4
COLLATION_CONNECTION: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

VIEWS 表中列的含义如下：

- TABLE_CATALOG：视图所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：视图所属的数据库的名称。
- TABLE_NAME：视图名称。
- VIEW_DEFINITION：视图的定义，由创建视图时 SELECT 部分的语句组成。
- CHECK_OPTION：CHECK_OPTION 的值。取值为 NONE、CASCADE 或 LOCAL。
- IS_UPDATABLE：UPDATE/INSERT/DELETE 是否对该视图可用。在TiDB，始终为 NO。
- DEFINER：视图的创建者用户名称，格式为 'user_name'@'host_name'。
- SECURITY_TYPE：SQL SECURITY 的值，取值为 DEFINER 或 INVOKER。
- CHARACTER_SET_CLIENT：在视图创建时 session 变量 character_set_client 的值。
- COLLATION_CONNECTION：在视图创建时 session 变量 collation_connection 的值。

4.8.18.3 Metrics Schema

METRICS_SCHEMA 是基于 Prometheus 中 TiDB 监控指标的一组视图。每个表的 PromQL（Prometheus 查询语言）的源均可在INFORMATION_SCHEMA.METRICS_TABLES 表中找到。

```
use metrics_schema;
SELECT * FROM uptime;
SELECT * FROM information_schema.metrics_tables WHERE table_name='uptime'\G
```

```
+----------------------------+-----------------+------------+----------------------------------+
| time                       | instance        | job        | value                            |
+----------------------------+-----------------+------------+----------------------------------+
| 2020-07-06 15:26:26.203000 | 127.0.0.1:10080 | tidb       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:10080 | tidb       | 183.60300016403198 |
```

```
| 2020-07-06 15:26:26.203000 | 127.0.0.1:20180 | tikv       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:20180 | tikv       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:2379  | pd         | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:2379  | pd         | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:9090  | prometheus | 123.72300004959106 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:9090  | prometheus | 183.72300004959106 |
+----------------------------+-----------------+------------+--------------------+
8 rows in set (0.00 sec)
*************************** 1. row ***************************
TABLE_NAME: uptime
    PROMQL: (time() - process_start_time_seconds{$LABEL_CONDITIONS})
    LABELS: instance,job
  QUANTILE: 0
   COMMENT: TiDB uptime since last restart(second)
1 row in set (0.00 sec)
```

```
show tables;
```

```
+-----------------------------------------------------+
| Tables_in_metrics_schema                            |
+-----------------------------------------------------+
| abnormal_stores                                     |
| etcd_disk_wal_fsync_rate                            |
| etcd_wal_fsync_duration                             |
| etcd_wal_fsync_total_count                          |
| etcd_wal_fsync_total_time                           |
| go_gc_count                                         |
| go_gc_cpu_usage                                     |
| go_gc_duration                                      |
| go_heap_mem_usage                                   |
| go_threads                                          |
| goroutines_count                                    |
| node_cpu_usage                                      |
| node_disk_available_size                            |
| node_disk_io_util                                   |
| node_disk_iops                                      |
| node_disk_read_latency                              |
| node_disk_size                                      |
..
| tikv_storage_async_request_total_time               |
| tikv_storage_async_requests                         |
| tikv_storage_async_requests_total_count             |
| tikv_storage_command_ops                            |
| tikv_store_size                                     |
| tikv_thread_cpu                                     |
```

```
| tikv_thread_nonvoluntary_context_switches           |
| tikv_thread_voluntary_context_switches              |
| tikv_threads_io                                     |
| tikv_threads_state                                  |
| tikv_total_keys                                     |
| tikv_wal_sync_duration                              |
| tikv_wal_sync_max_duration                          |
| tikv_worker_handled_tasks                           |
| tikv_worker_handled_tasks_total_num                 |
| tikv_worker_pending_tasks                           |
| tikv_worker_pending_tasks_total_num                 |
| tikv_write_stall_avg_duration                       |
| tikv_write_stall_max_duration                       |
| tikv_write_stall_reason                             |
| up                                                  |
| uptime                                              |
+                                                     +
626 rows in set (0.00 sec)
```

METRICS_SCHEMA 是监控相关的 summary 表的数据源， 例如metrics_summary、metrics_summary_by_label 和inspection_summary。

4.8.18.3.1 更多例子

下面以 metrics_schema 中的 tidb_query_duration 监控表为例，介绍监控表相关的使用和原理，其他的监控表原理均类似。

查询 information_schema.metrics_tables 中关于 tidb_query_duration 表相关的信息如下：

```
select * from information_schema.metrics_tables where table_name='tidb_query_duration';
```

```
+--
    <-->--------------------+-----------------------------------------------------------------------------------------------
    <-->
| TABLE_NAME           | PROMQL
    <-->
    <--> | LABELS              | QUANTILE | COMMENT                                                |
+--
    <-->--------------------+-----------------------------------------------------------------------------------------------
    <-->
| tidb_query_duration | histogram_quantile($QUANTILE, sum(rate(
    <--> tidb_server_handle_query_duration_seconds_bucket{$LABEL_CONDITIONS}[$RANGE_DURATION])) by
    <--> (le,sql_type,instance)) | instance,sql_type | 0.9       | The quantile of TiDB query
    <--> durations(second) |
+--
    <-->                    +
    <>--------------------  ----------------------------------------------------------------------------------------------
    <-->
```

- TABLE_NAME：对应于 metrics_schema 中的表名，这里表名是 tidb_query_duration。
- PROMQL：监控表的原理是将 SQL 映射成 PromQL 后向 Prometheus 请求数据，并将 Prometheus 返回的结果转换成 SQL 查询结果。该字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控项定义的 label，tidb_query_duration 有两个label，分别是 instance 和 sql_type。
- QUANTILE：百分位。直方图类型的监控数据会指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。tidb_query_duration 默认查询 0.9，也就是 P90 的监控值。
- COMMENT：对这个监控表的解释。可以看出 tidb_query_duration 表是用来查询TiDB query 执行的百分位时间，如 P999/P99/P90 的查询耗时，单位是秒。

再来看 tidb_query_duration 的表结构：

```
show create table metrics_schema.tidb_query_duration;
```

```
+--
    <⟶--------------------+----------------------
```

- `value`：监控项的值。

下面是查询时间[2020-03-25 23:40:00, 2020-03-25 23:42:00] 范围内的 P99 的 TiDB Query 耗时：

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    <-> 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+------------------+----------+----------+-----------------------------+
| time                | instance         | sql_type | quantile | value                       |
+---------------------+------------------+----------+----------+-----------------------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert    | 0.99      | 0.509929485256 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert    | 0.99      | 0.494690793986 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert    | 0.99      | 0.493460506934 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select    | 0.99      | 0.152058493415 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select    | 0.99      | 0.152193879678 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select    | 0.99      | 0.140498483232 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal  | 0.99      | 0.47104        |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal  | 0.99      | 0.11776        |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal  | 0.99      | 0.11776        |
+---------------------+------------------+----------+----------+-----------------------------+
```

以上查询结果的第一行意思是，在 2020-03-25 23:40:00 时，在TiDB 实例 172.16.5.40:10089 上，Insert 类型的语句的 P99 执行时间是 0.509929485256 秒。其他各行的含义类似，`sql_type` 列的其他值含义如下：

- `Select`：表示执行的 `select` 类型的语句。
- `internal`：表示 TiDB 的内部 SQL 语句，一般是统计信息更新，获取全局变量相关的内部语句。

进一步再查看上面语句的执行计划如下：

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    <-> 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    <->                +          +      +                            +
    <->
| id                | estRows   | task | access object              | operator info
    <->
    <-> |
+--
    <->                +          +      +                            +
    <->----------------- ----------- -------. ----------------------------.
| Selection_5       | 8000.00   | root |                            | not(isnull(Column#5))
    <->
    <-> |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    <-> (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[60s])) by (le,sql_type
    <-> ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:1m0s |
```

```
+--
    <-->----------------+----------+------+--------------------------+-----------------------------
```

```
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99      | 0.0624           |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | internal | 0.99      | 0.12416          |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99      | 0.0304           |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | internal | 0.99      | 0.06272          |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99      | 0.0629333333333  |
+---------------------+-------------------+----------+-----------+------------------------------+
```

最后查看执行计划，也会发现执行计划中的 PromQL 以及 step 的值都已经变成了 30 秒。

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    <⟶ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    <⟶                 +           +      +                              +
    <⟶
| id                | estRows   | task | access object              | operator info
    <⟶
    <⟶ |
+--
    <⟶                 +           +      +                              +
    <⟶
| Selection_5       | 8000.00   | root |                            | not(isnull(Column#5))
    <⟶
    <⟶ |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    <⟶ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[30s])) by (le,sql_type
    <⟶ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:30s |
+--
    <⟶                 +           +      +                              +
    <⟶
```

4.8.19 元数据锁

本文介绍了TiDB 中的元数据锁。

4.8.19.1 元数据锁的概念

在 TiDB 中，对元数据对象的更改采用的是在线异步变更算法。事务在执行时会获取开始时对应的元数据快照。如果事务执行过程中相关表上发生了元数据的更改，为了保证数据的一致性，TiDB 会返回 `Information <⟶ schema is changed` 的错误，导致用户事务提交失败。

为了解决这个问题，在TiDB v6.3.0 中，online DDL 算法中引入了元数据锁特性。通过协调表元数据变更过程中 DML 语句和 DDL 语句的优先级，让执行中的 DDL 语句等待持有旧版本元数据的 DML 语句提交，尽可能避免 DML 语句报错。

4.8.19.2 适用场景

元数据锁适用于所有的 DDL 语句，包括但不限于：

- ADD INDEX
- ADD COLUMN
- DROP COLUMN
- DROP INDEX
- DROP PARTITION
- TRUNCATE TABLE
- EXCHANGE PARTITION
- CHANGE COLUMN
- MODIFY COLUMN

使用元数据锁机制会给TiDB DDL 任务的执行带来一定的性能影响。为了降低元数据锁对 DDL 任务的影响，下列场景不需要加元数据锁：

- 开启了auto-commit 的查询语句
- 开启了 Stale Read 功能
- 访问临时表

4.8.19.3 使用元数据锁

在 v6.5.0 及之后的版本中，TiDB 默认开启元数据锁特性。当集群从 v6.5.0 之前的版本升级到 v6.5.0 及之后的版本时，TiDB 会自动开启元数据锁功能。如果需要关闭元数据锁，你可以将系统变量tidb_enable_metadata_lock设置为 OFF。

4.8.19.4 元数据锁的影响

- 对于 DML 语句来说，元数据锁不会导致 DML 语句被阻塞，因此也不会存在死锁的问题。
- 开启元数据锁后，事务中某个元数据对象的元数据信息在第一次访问时确定，之后不再变化。
- 对于 DDL 语句来说，在进行元数据状态变更时，会被涉及相关元数据的旧事务所阻塞。例如以下的执行流程：

Session 1	Session 2
CREATE TABLE t (a INT);	
INSERT INTO t VALUES(1);	
BEGIN;	
	ALTER TABLE t ADD COLUMN b INT
SELECT * FROM t;（采用 t 表当前的元数据版本，返回 (a=1，b=NULL)，同时给表 t 加锁）	
	ALTER TABLE t ADD COLUMN c INT

在可重复读隔离级别下，如果从事务开始到确定一个表的元数据过程中，执行了加索引或者变更列类

型等需要更改数据的 DDL，则有以下表现：

Session 1	Session 2
CREATE TABLE t (a INT); INSERT INTO t VALUES(1); BEGIN;	
	ALTER TABLE t ADD INDEX idx(a);
SELECT * FROM t; (索引 idx 不可用) COMMIT; BEGIN;	
	ALTER TABLE t MODIFY COLUMN a CHAR(10);
SELECT * FROM t; (报错 Error 8028: Information schema is changed)	

4.8.19.5 元数据锁的可观测性

TiDB v6.3.0 引入了 mysql.tidb_mdl_view 视图，可以用于查看当前阻塞的 DDL 的相关信息。

> 注意：
> 查询 mysql.tidb_mdl_view 视图需要有 PROCESS 权限。

下面以给表t 添加索引为例，假设有 DDL 语句 ALTER TABLE t ADD INDEX idx(a)：

```
SELECT * FROM mysql.tidb_mdl_view\G
*************************** 1. row ***************************
    JOB_ID: 141
   DB_NAME: test
TABLE_NAME: t
     QUERY: ALTER TABLE t ADD INDEX idx(a)
SESSION ID: 2199023255957
  TxnStart: 08-30 16:35:41.313(435643624013955072)
SQL_DIGESTS: ["begin","select * from `t`"]
1 row in set (0.02 sec)
```

可以从上面的输出结果中了解到，有一个 SESSION ID 为 2199023255957 的事务阻塞了该添加索引 DDL 的执行。该事务执行的 SQL 语句如 SQL_DIGESTS 中所示，即 ["begin","select * from `t`"]。如果想要使被阻塞的 DDL 能够继续执行，可以通过如下Global KILL 命令中止 SESSION ID 为 2199023255957 的事务：

```
mysql> KILL 2199023255957;
Query OK, 0 rows affected (0.00 sec)
```

中止该事务后，再次查询 mysql.tidb_mdl_view 视图。此时，查询结果不再显示上面的事务信息，说明 DDL 不再被阻塞：

```
SELECT * FROM mysql.tidb_mdl_view\G
Empty set (0.01 sec)
```

4.8.19.6 元数据锁的原理

4.8.19.6.1 问题描述

TiDB 中 DDL 操作使用的是 online DDL 模式。一个 DDL 语句在执行过程中，需要修改定义的对象元数据版本可能会进行多次小版本变更，而元数据在线异步变更的算法只论证了相邻的两个小版本之间是兼容的，即在相邻的两个元数据版本间操作，不会破坏 DDL 变更对象所存储的数据一致性。

以添加索引为例，DDL 语句的状态会经历 None -> Delete Only，Delete Only -> Write Only，Write Only -> Write Reorg，Write Reorg -> Public 这四个变化。

以下的提交流程将违反“相邻的两个小版本之间是兼容的” 约束：

事务	所用的版本	集群最新版本	版本差
txn1	None	None	0
txn2	DeleteOnly	DeleteOnly	0
txn3	WriteOnly	WriteOnly	0
txn4	None	WriteOnly	2
txn5	WriteReorg	WriteReorg	0
txn6	WriteOnly	WriteReorg	1
txn7	Public	Public	0

其中 txn4 提交时采用的元数据版本与集群最新的元数据版本相差了两个版本，会影响数据正确性。

4.8.19.6.2 实现

引入元数据锁会保证整个TiDB 集群中的所有事务所用的元数据版本最多相差一个版本。为此：

- 执行 DML 语句时，TiDB 会在事务上下文中记录该 DML 语句访问的元数据对象，例如表、视图，以及对应的元数据版本。事务提交时会清空这些记录。
- DDL 语句进行状态变更时，会向所有的TiDB 节点推送最新版本的元数据。如果一个TiDB 节点上所有与这次状态变更相关的事务使用的元数据版本与当前元数据版本之差小于 2，则称这个TiDB 节点获得了该元数据对象的元数据锁。当集群中的所有TiDB 节点都获得了该元数据对象的元数据锁后，才能进行下一次状态变更。

4.9 遥测

开启遥测后，TiDB、TiUP 及 TiDB Dashboard 会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品，例如，通过这些使用信息，PingCAP 可以了解常见的TiDB 集群操作，从而确定新功能优先级。

注意：

- 自 2023 年 2 月 20 日起，新发布的TiDB 和 TiDB Dashboard 版本（含 v6.6.0），默认关闭遥测功能，即默认不再收集使用情况信息分享给 PingCAP。如果集群在升级至这些版本前使用默认的遥测配置，则升级后遥测功能处于关闭状态。具体的版本可参考TiDB 版本发布时间线。
- 从 v1.11.3 起，新部署的TiUP 默认关闭遥测功能，即默认不再收集使用情况信息。如果从 v1.11.3 之前的TiUP 版本升级至 v1.11.3 或更高TiUP 版本，遥测保持升级前的开启或关闭状态。

4.9.1 哪些使用情况信息会被收集？

以下章节具体描述了各个组件收集并分享的使用情况信息。若收集的使用情况信息有变化，将在版本更新说明中告知。

注意：

在任何情况下，集群中用户存储的数据都不会被收集。另请参阅 PingCAP 隐私声明。

4.9.1.1 TiDB

当 TiDB 遥测功能开启时，TiDB 集群将会以 6 小时为周期收集使用情况信息并分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- 集群的部署情况，包括各个组件所在的硬件信息（CPU、内存、磁盘）、组件版本号、操作系统版本号等
- 系统的查询请求状态，例如查询请求次数、持续时长等
- 系统组件的使用情况，例如Async Commit 功能是否有被使用
- 去识别化处理后的产品遥测数据发送端的IP 地址

可以通过执行以下 SQL 语句查看TiDB 收集的使用情况信息内容：

```
ADMIN SHOW TELEMETRY;
```

4.9.1.2 TiDB Dashboard

当 TiDB Dashboard 遥测功能开启时，用户在TiDB Dashboard 网页界面上进行操作时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符

- 界面访问情况，如访问的TiDB Dashboard 功能页面名称
- 用户浏览器及操作系统信息，如浏览器名称和版本号、操作系统名称、屏幕分辨率等

可以使用 Chrome 开发者工具的网络功能或 Firefox 开发者工具的网络监视器功能查看 TiDB Dashboard 发送的使用情况信息内容。

4.9.1.3 TiUP

当TiUP 遥测功能开启时，执行TiUP 命令时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- TiUP 命令的执行情况，如命令执行是否成功、命令执行耗时等
- 使用TiUP 进行部署的情况，如部署的目标机器硬件信息、组件版本号、修改过的部署配置名称等

使用TiUP 时，可通过设置 `TIUP_CLUSTER_DEBUG=enable` 环境变量输出执行命令时收集的使用情况信息，例如：

```
TIUP_CLUSTER_DEBUG=enable tiup cluster list
```

4.9.1.4 TiSpark

> **注意：**
>
> 自 TiSpark v3.3 起，默认关闭遥测功能，即TiSpark 默认不收集使用情况信息，不将这些信息分享给 PingCAP 用于改善产品。

当TiSpark 遥测功能开启时，Spark 在使用TiSpark 时会发送会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- TiSpark 的部分配置信息，如读取引擎、是否开启流式读取等
- 用户集群部署情况，包括TiSpark 所在节点的机器硬件信息、操作系统信息和组件版本号等

使用TiSpark 时，可以通过查看 Spark 日志来了解TiSpark 收集的使用情况，可将 Spark 日志级别调至INFO 或更低，例如：

```
cat {spark.log} | grep Telemetry report | tail -n 1
```

4.9.2 开启遥测功能

4.9.2.1 开启TiDB 遥测

4.9.2.1.1 第 1 步：修改配置文件中的遥测配置

在已有TiDB集群上开启遥测功能，需要为集群设置`enable-telemetry = true`，但需要重启集群后才能生效。

以下是在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = true
```

启动TiDB时指定命令行参数 `--config=tidb_config.toml` 使得该配置生效。

详情参见TiDB 配置参数、TiDB 配置文件描述。

通过TiUP Playground 试用

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = true
```

启动TiUP Playground 时，指定命令行参数 `--db.config tidb_config.toml` 使得该配置生效，如：

```
tiup playground --db.config tidb_config.toml
```

详情参见TiUP - 本地快速部署TiDB 集群。

通过TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  tidb:
    enable-telemetry: true
```

通过TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者TidbCluster Custom Resource 中配置 `spec.tidb.config.enable-telemetry: true`。

详情参见在标准 Kubernetes 上部署TiDB 集群。

> 注意：
>
> 该配置需使用TiDB Operator v1.1.3 或更高版本才能生效。

4.9.2.1.2 第 2 步：修改系统全局变量的遥测配置

在步骤 1 完成后，还需要修改系统全局变量`tidb_enable_telemetry`，才能打开 TiDB 遥测功能：

```
SET GLOBAL tidb_enable_telemetry = 1;
```

配置文件和全局变量都开启后，遥测功能才能处于开启状态。

4.9.2.2 开启TiDB Dashboard遥测

可以修改PD配置中dashboard.enable-telemetry = true启动TiDB Dashboard遥测功能。对于已启动的集群，该配置需要重启后才能生效。

以下列出在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件pd_config.toml包含如下内容：

```
[dashboard]
enable-telemetry = true
```

启动PD时指定命令行参数--config=pd_config.toml使得该配置生效。

详情参见PD配置参数、PD配置文件描述。

通过TiUP Playground试用

创建配置文件pd_config.toml包含如下内容：

```
[dashboard]
enable-telemetry = true
```

启动TiUP Playground时，指定命令行参数--pd.config pd_config.toml使得该配置生效，如：

```
tiup playground --pd.config pd_config.toml
```

详情参见TiUP - 本地快速部署TiDB集群。

通过TiUP Cluster部署

修改部署拓扑文件topology.yaml，新增（或在现有项中添加）以下内容：

```
server_configs:
  pd:
    dashboard.enable-telemetry: true
```

通过TiDB Operator在Kubernetes上部署

在tidb-cluster.yaml中或者TidbCluster Custom Resource中配置spec.pd.config.dashboard.enable-telemetry: <⟷ true。

详情参见在标准Kubernetes上部署TiDB集群。

> 注意：
>
> 该配置需使用TiDB Operator v1.1.3或更高版本才能生效。

4.9.2.3 开启TiUP 遥测

可通过执行以下命令开启TiUP 遥测功能：

```
tiup telemetry enable
```

4.9.2.4 开启TiSpark 遥测

可以通过在 Spark 配置文件设置 `spark.tispark.telemetry.enable = true` 来开启TiSpark 的遥测功能。

4.9.3 禁用遥测功能

4.9.3.1 部署TiDB 时禁用TiDB 遥测

在已有TiDB 集群上开启遥测后，可以为每个集群设置`enable-telemetry = false` 以禁用TiDB 遥测功能，但需要重启集群后才能生效。

以下是在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = false
```

启动TiDB 时指定命令行参数 `--config=tidb_config.toml` 使得该配置生效。

详情参见TiDB 配置参数、TiDB 配置文件描述。

通过TiUP Playground 试用

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = false
```

启动TiUP Playground 时，指定命令行参数 `--db.config tidb_config.toml` 使得该配置生效，如：

```
tiup playground --db.config tidb_config.toml
```

详情参见TiUP - 本地快速部署TiDB 集群。

通过TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  tidb:
    enable-telemetry: false
```

通过TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者TidbCluster Custom Resource 中配置 `spec.tidb.config.enable-telemetry: false`。

详情参见在标准 Kubernetes 上部署TiDB 集群。

> **注意：**
>
> 该配置需使用TiDB Operator v1.1.3 或更高版本才能生效。

4.9.3.2 动态禁用TiDB 遥测

对于已部署的TiDB 集群，还可以修改系统全局变量`tidb_enable_telemetry` 动态禁用TiDB 遥测功能：

```
SET GLOBAL tidb_enable_telemetry = 0;
```

配置文件的禁用优先级高于全局变量。若通过配置文件禁用了遥测功能，则全局变量的配置将不起作用，遥测功能总是处于关闭状态。

4.9.3.3 禁用TiDB Dashboard 遥测

可以修改 PD 配置中`dashboard.enable-telemetry = false` 禁用TiDB Dashboard 遥测功能。对于已启动的集群，该配置需要重启后才能生效。

以下列出在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = false
```

启动 PD 时指定命令行参数 `--config=pd_config.toml` 使得该配置生效。

详情参见PD 配置参数、PD 配置文件描述。

通过TiUP Playground 试用

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = false
```

启动TiUP Playground 时，指定命令行参数 `--pd.config pd_config.toml` 使得该配置生效，如：

```
tiup playground --pd.config pd_config.toml
```

详情参见TiUP - 本地快速部署TiDB 集群。

通过TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  pd:
    dashboard.enable-telemetry: false
```

通过TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者TidbCluster Custom Resource 中配置 `spec.pd.config.dashboard.enable-telemetry: false`。

详情参见在标准 Kubernetes 上部署TiDB 集群。

> **注意：**
>
> 该配置需使用TiDB Operator v1.1.3 或更高版本才能生效。

4.9.3.4 禁用TiUP 遥测

可通过执行以下命令禁用TiUP 遥测功能：

```
tiup telemetry disable
```

4.9.3.5 禁用TiSpark 遥测

可以通过在 Spark 配置文件设置 `spark.tispark.telemetry.enable = false` 来禁用TiSpark 的遥测功能。

4.9.4 查看遥测启用状态

对于TiDB 遥测，可通过执行以下 SQL 语句查看遥测状态：

```
ADMIN SHOW TELEMETRY;
```

若 `DATA_PREVIEW` 列为空，说明遥测没有开启，否则说明遥测已开启。还可以从 `LAST_STATUS` 列了解上次分享使用情况信息的时间、是否成功等。

对于TiUP 遥测，可通过执行以下命令查看遥测状态：

```
tiup telemetry status
```

4.9.5 使用情况信息合规性

为了满足不同国家或地区对于此类信息的合规性要求，使用情况信息会按照不同的操作者IP 地址发送到位于不同国家的服务器，具体如下：

- 若为中国大陆IP 地址，使用情况信息将会发送并存储于中国大陆境内的公有云服务器。
- 若为中国大陆以外IP 地址，使用情况信息将会发送并存储于美国的公有云服务器。

可参阅 PingCAP 隐私声明了解详情。

4.10 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。

4.10.1 错误码

TiDB 兼容 MySQL 的错误码，在大多数情况下，返回和 MySQL 一样的错误码。关于 MySQL 的错误码列表，详见[MySQL 5.7 Error Message Reference。另外还有一些TiDB 特有的错误码：

> 注意：
>
> 有一部分错误码属于内部错误，正常情况下TiDB 会自行处理不会直接返回给用户，故没有在此列出。
>
> 如果您遇到了这里没有列出的错误码，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8001

  请求使用的内存超过 TiDB 内存使用的阈值限制。出现这种错误，可以通过调整系统变量 `tidb_mem_quota_query` 来增大单个 SQL 使用的内存上限。

- Error Number: 8002

  带有 `SELECT FOR UPDATE` 语句的事务，在遇到写入冲突时，为保证一致性无法进行重试，事务将进行回滚并返回该错误。出现这种错误，应用程序可以安全地重新执行整个事务。

- Error Number: 8003

  `ADMIN CHECK TABLE` 命令在遇到行数据跟索引不一致的时候返回该错误，在检查表中数据是否有损坏时常出现。出现该错误时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8004

  单个事务过大，原因及解决方法请参考这里

- Error Number: 8005

  完整的报错信息为 `ERROR 8005 (HY000): Write Conflict, txnStartTS is stale`。

  事务在TiDB 中遇到了写入冲突。请检查业务逻辑，重试写入操作。

- Error Number: 8018

  当执行重新载入插件时，如果之前插件没有载入过，则会出现该错误。出现该错误，进行插件首次载入即可。

- Error Number: 8019

  重新载入的插件版本与之前的插件版本不一致，无法重新载入插件并报告该错误。可重新载入插件，确保插件的版本与之前载入的插件版本一致。

- Error Number: 8020

  当表被加锁时，如果对该表执行写入操作，将出现该错误。请将表解锁后，再进行尝试写入。

- Error Number: 8021

  当向TiKV 读取的 key 不存在时将出现该错误，该错误用于内部使用，对外表现为读到的结果为空。

- Error Number: 8022

  事务提交失败，已经回滚，应用程序可以安全的重新执行整个事务。

- Error Number: 8023

  在事务内，写入事务缓存时，设置了空值，将返回该错误。这是一个内部使用的错误，将由内部进行处理，不会返回给应用程序。

- Error Number: 8024

  非法的事务。当事务执行时，发现没有获取事务的ID (Start Timestamp)，代表正在执行的事务是一个非法的事务，将返回该错误。通常情况下不会出现该问题，当发生时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8025

  写入的单条键值对过大。TiDB 默认支持最大 6MB 的单个键值对，超过该限制可适当调整`txn-entry-size-limit`配置项以放宽限制。

- Error Number: 8026

  使用了没有实现的接口函数。该错误仅用于数据库内部，应用程序不会收到这个错误。

- Error Number: 8027

  表结构版本过期。TiDB 采用在线变更表结构的方法。当TiDB server 表结构版本落后于整个系统的时，执行 SQL 将遇到该错误。遇到该错误，请检查该TiDB server 与 PD leader 之间的网络。

- Error Number: 8028

  TiDB v6.3.0 引入了元数据锁特性。在关闭元数据锁的情况下，当事务执行时，事务无法感知到TiDB 的表结构发生了变化。因此，TiDB 在事务提交时，会对事务涉及表的结构进行检查。如果事务执行中表结构发生了变化，则事务将提交失败，并返回该错误。遇到该错误，应用程序可以安全地重新执行整个事务。

  在打开元数据锁的情况下，非RC 隔离级别中，如果从事务开始到初次访问一个表之间，该表进行了有损的列类型变更操作（例如 `INT` 类型变成 `CHAR` 类型是有损的，`TINYINT` 类型变成 `INT` 类型这种不需要重写数据的则是无损的），则访问该表的语句报错，事务不会自动回滚。用户可以继续执行其他语句，并决定是否回滚或者提交事务。

- Error Number: 8029

  当数据库内部进行数值转换发生错误时，将会出现该错误，该错误仅在内部使用，对外部应用将转换为具体类型的错误。

- Error Number: 8030

  将值转变为带符号正整数时发生了越界，将结果显示为负数。多在告警信息里出现。

- Error Number: 8031

  将负数转变为无符号数时，将负数转变为了正数。多在告警信息里出现。

- Error Number: 8032

  使用了非法的year 格式。 year 只允许 1 位、 2 位和 4 位数。

- Error Number: 8033

  使用了非法的year 值。 year 的合法范围是(1901, 2155)。

- Error Number: 8037

  week 函数中使用了非法的mode 格式。 mode 必须是一位数字，范围 [0, 7]。

- Error Number: 8038

  字段无法获取到默认值。一般作为内部错误使用，转换成其他具体错误类型后，返回给应用程序。

- Error Number: 8040

  尝试进行不支持的操作，比如在View 和 Sequence 上进行lock table。

- Error Number: 8047

  设置了不支持的系统变量值，通常在用户设置了数据库不支持的变量值后的告警信息里出现。

- Error Number: 8048

  设置了不支持的隔离级别，如果是使用第三方工具或框架等无法修改代码进行适配的情况，可以考虑通过`tidb_skip_isolation_level_check` 来绕过这一检查。

  ```
  set @@tidb_skip_isolation_level_check = 1;
  ```

- Error Number: 8050

  设置了不支持的权限类型，遇到该错误请参考TiDB 权限说明进行调整。

- Error Number: 8051

  TiDB 在解析客户端发送的Exec 参数列表时遇到了未知的数据类型。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8052

  来自客户端的数据包的序列号错误。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8055

  当前快照过旧，数据可能已经被GC。可以调大`tidb_gc_life_time` 的值来避免该问题。从TiDB v4.0.8 版本起，TiDB 会自动为长时间运行的事务保留数据，一般不会遇到该错误。

  有关GC 的介绍和配置可以参考GC 机制简介和 GC 配置文档。

- Error Number: 8059

  自动随机量可用次数用尽无法进行分配。当前没有恢复这类错误的方法。建议在使用auto random 功能时使用bigint 以获取最大的可分配次数，并尽量避免手动给auto random 列赋值。相关的介绍和使用建议可以参考auto random 功能文档。

- Error Number: 8060

  非法的自增列偏移量。请检查 `auto_increment_increment` 和 `auto_increment_offset` 的取值是否符合要求。

- Error Number: 8061

  不支持的 SQL Hint。请参考 Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8062

  SQL Hint 中使用了非法的token，与 Hint 的保留字冲突。请参考 Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8063

  SQL Hint 中限制内存使用量超过系统设置的上限，设置被忽略。请参考 Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8064

  解析 SQL Hint 失败。请参考 Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8065

  SQL Hint 中使用了非法的整数。请参考 Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8066

  JSON_OBJECTAGG 函数的第二个参数是非法参数。

- Error Number: 8101

  插件 ID 格式错误，正确的格式是 `[name]-[version]` 并且 name 和 version 中不能带有 '-'。

- Error Number: 8102

  无法读取插件定义信息。请检查插件相关的配置。

- Error Number: 8103

  插件名称错误，请检查插件的配置。

- Error Number: 8104

  插件版本不匹配，请检查插件的配置。

- Error Number: 8105

  插件被重复载入。

- Error Number: 8106

  插件定义的系统变量名称没有以插件名作为开头，请联系插件的开发者进行修复。

- Error Number: 8107

  载入的插件未指定版本或指定的版本过低，请检查插件的配置。

- Error Number: 8108

  不支持的执行计划类型。该错误为内部处理的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8109

  analyze 索引时找不到指定的索引。

- Error Number: 8110

  不能进行笛卡尔积运算，需要将配置文件里的 `cross-join` 设置为 `true`。

- Error Number: 8111

  execute 语句执行时找不到对应的prepare 语句。

- Error Number: 8112

  execute 语句的参数个数与prepare 语句不符合。

- Error Number: 8113

  execute 语句涉及的表结构在prepare 语句执行后发生了变化。

- Error Number: 8115

  不支持prepare 多行语句。

- Error Number: 8116

  不支持prepare DDL 语句。

- Error Number: 8120

  获取不到事务的 start tso，请检查 PD Server 状态/监控/日志以及TiDB Server 与 PD Server 之间的网络。

- Error Number: 8121

  权限检查失败，请检查数据库的权限配置。

- Error Number: 8122

  指定了通配符，但是找不到对应的表名。

- Error Number: 8123

  带聚合函数的 SQL 中返回非聚合的列，违反了 `only_full_group_by` 模式。请修改 SQL 或者考虑关闭 `only_full_group_by` 模式。

- Error Number: 8129

  TiDB 尚不支持键长度>= 65536 的 JSON 对象。

- Error Number: 8130

  完整的报错信息为 `ERROR 8130 (HY000): client has multi-statement capability disabled`。

  从早期版本的TiDB 升级后，可能会出现该问题。为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 `COM_QUERY` 调用中执行多个查询。

  可通过系统变量`tidb_multi_statement_mode` 控制是否在同一 `COM_QUERY` 调用中执行多个查询。

- Error Number: 8138

  事务试图写入的行值有误，请参考数据索引不一致报错。

- Error Number: 8139

  事务试图写入的行和索引的handle 值不一致，请参考数据索引不一致报错。

- Error Number: 8140

  事务试图写入的行和索引的值不一致，请参考数据索引不一致报错。

- Error Number: 8141

  事务写入时，对 key 的存在性断言报错，请参考数据索引不一致报错。

- Error Number: 8143

  非事务 DML 语句的一个batch 报错，语句中止，请参考非事务 DML 语句

- Error Number: 8147

当tidb_constraint_check_in_place_pessimistic 设置为 OFF 时，为保证事务的正确性，SQL 语句执行时产生的任何错误都可能导致TiDB 返回 8147 报错并中止当前事务。具体的错误原因，请参考对应的报错信息。详见约束。

- Error Number: 8154

  目前 LOAD DATA 不支持从TiDB 服务器本地导入数据，可以指定 LOCAL 从客户端导入，或者将数据上传到 S3/GCS 再进行导入。请参考LOAD DATA。

- Error Number: 8156

  LOAD DATA 语句的文件路径不能为空。需要设置正确的路径再进行导入。请参考LOAD DATA。

- Error Number: 8158

  传入的 S3/GCS 路径无效。请参考外部存储设置有效的路径。

- Error Number: 8159

  TiDB 无法访问 LOAD DATA 语句中传入的 S3/GCS 路径。请确保填入的 S3/GCS bucket 存在，且你输入了正确的 access key 和 secret access key 以让TiDB 服务器有权限访问 S3/GCS 对应的bucket。

- Error Number: 8160

  LOAD DATA 读取数据文件失败。请根据具体的错误提示进行处理。

- Error Number: 8162

  LOAD DATA 语句存在错误。请参考LOAD DATA 查看已支持的功能。

- Error Number: 8200

  尚不支持的 DDL 语法。请参考与 MySQL DDL 的兼容性。

- Error Number: 8214

  DDL 操作被admin cancel 操作终止。

- Error Number: 8215

  Admin Repair 表失败，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8216

  自动随机列使用的方法不正确，请参考auto random 功能文档进行修改。

- Error Number: 8223

  检测出数据与索引不一致的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8224

  找不到 DDL job，请检查restore 操作指定的job id 是否存在。

- Error Number: 8225

  DDL 已经完成，无法被取消。

- Error Number: 8226

  DDL 几乎要完成了，无法被取消。

- Error Number: 8227

  创建 Sequence 时使用了不支持的选项，支持的选项的列表可以参考Sequence 使用文档。

- Error Number: 8228

  在 Sequence 上使用 `setval` 时指定了不支持的类型，该函数的示例可以在Sequence 使用文档中找到。

- Error Number: 8229

  事务超过存活时间，遇到该问题可以提交或者回滚当前事务，开启一个新事务。

- Error Number: 8230

  TiDB 目前不支持在新添加的列上使用 Sequence 作为默认值，如果尝试进行这类操作会返回该错误。

- Error Number: 8248

  资源组已存在。在重复创建资源组时返回该错误。

- Error Number: 8249

  资源组不存在。在修改或绑定不存在的资源组时返回该错误。请参考创建资源组。

- Error Number: 8250

  完整的报错信息如下：

  ```
  ERROR 8250 (HY000): Resource control feature is disabled. Run "SET GLOBAL tidb_enable_resource_control
  ⟷ ='on'" to enable the feature
  ```

  资源控制的功能没有打开时，使用资源管控(Resource Control) 相关功能会返回该错误。你可以开启全局变量`tidb_enable_resource_control` 启用资源管控。

- Error Number: 8251

  `Resource Control` 组件在TiDB 启动时进行初始化，相关配置会从 `Resource Control` 的服务端 `Resource ⟷ Manager` 上获取，如果此过程中出错，则会返回此错误。

- Error Number: 8252

  完整的报错信息如下：

  ```
  ERROR 8252 (HY000): Exceeded resource group quota limitation
  ```

  在尝试消耗超过资源组的限制时返回该错误。一般出现该错误，是由于单次事务太大或者并发太多导致，需调整事务大小或减少客户端并发数。

- Error Number: 9001

  完整的报错信息为 `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。

- Error Number: 9002

  完整的报错信息为 `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。

- Error Number: 9003

  完整的报错信息为 `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。

- Error Number: 9004

  完整的报错信息为 `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。

- Error Number: 9005

  完整的报错信息为 `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。

- Error Number: 9006

  完整的报错信息为 `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。 你可以使用如下命令修改 `tidb_gc_life_time` 的值：

  ```
  SET GLOBAL tidb_gc_life_time = '30m';
  ```

  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。

- Error Number: 9007

  报错信息以 `ERROR 9007 (HY000): Write conflict` 开头。

  如果报错信息中含有“reason=LazyUniquenessCheck”，说明是悲观事务并且设置了`@@tidb_constraint_check_in_place_pes <→ =OFF`，业务中存在唯一索引上的写冲突，此时悲观事务不能保证执行成功。可以在应用测重试事务，或将该变量设置成 `ON` 绕过。详见约束。

- Error Number: 9008

  同时向TiKV发送的请求过多，超过了限制。请调大 `tidb_store_limit` 或将其设置为 `0` 来取消对请求流量的限制。

- Error Number: 9010

  TiKV无法处理这条raft log，请检查TiKV Server状态/监控/日志。

- Error Number: 9012

  请求TiFlash超时。请检查TiFlash Server状态/监控/日志以及TiDB Server与TiFlash Server之间的网络。

- Error Number: 9013

  TiFlash 操作繁忙。该错误一般出现在数据库负载比较高时。请检查TiFlash Server 的状态/监控/日志。

4.10.1.1 MySQL 原生报错汇总

- Error Number: 2013 (HY000)

  完整的报错信息为 `ERROR 2013 (HY000): Lost connection to MySQL server during query`。

  排查方法如下：

  - 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。

- Error Number: 1105 (HY000)

  完整的报错信息为 `ERROR 1105 (HY000): other error: unknown error Wire Error(InvalidEnumValue (4004))`

  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。

- Error Number: 1148 (42000)

  完整的报错信息为 `ERROR 1148 (42000): the used command is not allowed with this TiDB version`。

  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。

- Error Number: 9001 (HY000)

  完整的报错信息为 `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。

- TiDB 日志中的报错信息：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。

4.10.2 故障诊断

参见故障诊断文档。

4.11 通过拓扑label 进行副本调度

> **注意：**
>
> TiDB 在 v5.3.0 中引入了Placement Rules in SQL。使用该功能，你可以更方便地配置表和分区的位置。在未来版本中，Placement Rules in SQL 可能取代通过 PD 配置放置规则的功能。

为了提升TiDB 集群的高可用性和数据容灾能力，我们推荐让TiKV 节点尽可能在物理层面上分散，例如让TiKV 节点分布在不同的机架甚至不同的机房。PD 调度器根据TiKV 的拓扑信息，会自动在后台通过调度使得Region 的各个副本尽可能隔离，从而使得数据容灾能力最大化。

要让这个机制生效，需要在部署时进行合理配置，把集群的拓扑信息（特别是TiKV 的位置）上报给 PD。阅读本章前，请先确保阅读TiUP 部署方案。

4.11.1 根据集群拓扑配置labels

4.11.1.1 设置TiKV 和 TiFlash 的 labels

TiKV 和 TiFlash 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 和 TiFlash 在启动后，会将自身的标签上报给 PD，因此可以使用标签来标识 TiKV 和 TiFlash 节点的地理位置。

比如集群的拓扑结构分成四层：机房(zone) -> 数据中心(dc) -> 机架(rack) -> 主机(host)，就可以使用这 4 个标签来设置TiKV 和 TiFlash 的位置。

使用命令行参数的方式启动一个TiKV 实例：

```
tikv-server --labels zone=<zone>,dc=<dc>,rack=<rack>,host=<host>
```

使用配置文件的方式：

```
[server]
[server.labels]
zone = "<zone>"
dc = "<dc>"
rack = "<rack>"
host = "<host>"
```

TiFlash 支持通过tiflash-learner.toml（tiflash-proxy 的配置文件）的方式设置labels：

```
[server]
[server.labels]
zone = "<zone>"
dc = "<dc>"
rack = "<rack>"
host = "<host>"
```

4.11.1.2 设置TiDB 的 labels（可选）

如果需要使用 Follower Read 的优先读同一区域副本的功能，需要为TiDB 节点配置相关的 labels。

TiDB 支持使用配置文件的方式设置 labels：

```
[labels]
zone = "<zone>"
dc = "<dc>"
rack = "<rack>"
```

```
host = "<host>"
```

> **注意：**
>
> 目前，TiDB 依赖zone 标签匹配选择同一区域的副本。如果需要使用此功能，需要在PD location <⟶ -labels 配置中包含 zone，并在 TiDB、TiKV 和TiFlash 设置的 labels 中包含 zone。关于如何设置TiKV 和 TiFlash 的 labels，可参考设置 TiKV 和 TiFlash 的 labels。

4.11.1.3 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，是一个字符串数组。该配置的每一项与 TiKV labels 的 key 是对应的，而且其中每个 key 的顺序代表不同标签的级别关系（从左到右，隔离级别依次递减）。

location-labels 没有默认值，你可以根据具体需求来设置该值，包括 zone、rack、host 等等。同时，location <⟶ -labels 对标签级别的数量也没有限制（即不限定于 3 个），只要其级别与 TiKV 服务器的标签匹配，则可以配置成功。

> **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。
> - 如果你使用 Placement Rules in SQL，只需要配置 TiKV 的 labels 即可。Placement Rules in SQL 目前不兼容 PD location-labels 设置，会忽略该设置。不建议 location-labels 与 Placement Rules in SQL 混用，否则可能产生非预期的结果。

你可以根据集群状态来选择不同的配置方式：

- 在集群初始化之前，可以通过 PD 的配置文件进行配置：

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

- 如果需要在 PD 集群初始化完成后进行配置，则需要使用pd-ctl 工具进行在线更改：

```
pd-ctl config set location-labels zone,rack,host
```

4.11.1.4 设置 PD 的 isolation-level 配置

在配置了 location-labels 的前提下，用户可以还通过 isolation-level 配置来进一步加强对TiKV 集群的拓扑隔离要求。假设按照上面的说明通过 location-labels 将集群的拓扑结构分成三层：机房(zone) -> 机架(rack) -> 主机(host)，并对 isolation-level 作如下配置：

```
[replication]
isolation-level = "zone"
```

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 isolation-level zone
```

其中，isolation-level 配置是一个字符串，需要与 location-labels 的其中一个 key 对应。该参数限制TiKV 拓扑集群的最小且强制隔离级别要求。

> **注意：**
>
> isolation-level 默认情况下为空，即不进行强制隔离级别限制，若要对其进行设置，必须先配置 PD 的 location-labels 参数，同时保证 isolation-level 的值一定为 location-labels 中的一个。

4.11.1.5 使用TiUP 进行配置（推荐）

如果使用TiUP 部署集群，可以在初始化配置文件中统一进行location 相关配置。TiUP 会负责在部署时生成对应的 TiKV、PD 和 TiFlash 配置文件。

下面的例子定义了 zone 和 host 两层拓扑结构。集群的TiKV 和TiFlash 分布在三个zone，z1、z2 和z3。每个zone 内有四台主机，z1 两台主机分别部署两个TiKV 实例，另外两台分别部署一个TiFlash 实例，z2 和 z3 其中两台主机分别部署一个TiKV 实例，另外两台分别部署一个TiFlash 实例。以下例子中 tikv-n 代表第n 个 TiKV 节点的IP 地址，tiflash-n 代表第n 个 TiFlash 节点的IP 地址。

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

tikv_servers:
## z1
  - host: tikv-1
    config:
      server.labels:
        zone: z1
        host: h1
   - host: tikv-2
    config:
      server.labels:
```

```
        zone: z1
        host: h1
  - host: tikv-3
    config:
      server.labels:
        zone: z1
        host: h2
  - host: tikv-4
    config:
      server.labels:
        zone: z1
        host: h2
## z2
  - host: tikv-5
    config:
      server.labels:
        zone: z2
        host: h1
   - host: tikv-6
    config:
      server.labels:
        zone: z2
        host: h2
## z3
  - host: tikv-7
    config:
      server.labels:
        zone: z3
        host: h1
  - host: tikv-8
    config:
      server.labels:
        zone: z3
        host: h2

tiflash_servers:
## z1
  - host: tiflash-1
    learner_config:
      server.labels:
        zone: z1
        host: h3
   - host: tiflash-2
    learner_config:
      server.labels:
```

```
        zone: z1
        host: h4
## z2
  - host: tiflash-3
    learner_config:
      server.labels:
        zone: z2
        host: h3
   - host: tiflash-4
    learner_config:
      server.labels:
        zone: z2
        host: h4
## z3
  - host: tiflash-5
    learner_config:
      server.labels:
        zone: z3
        host: h3
  - host: tiflash-6
    learner_config:
      server.labels:
        zone: z3
        host: h4
```

详情参阅TiUP 跨数据中心部署拓扑。

> **注意：**
>
> 如果你未在配置文件中配置 `replication.location-labels` 项，使用该拓扑配置文件部署集群时可能会报错。建议在部署集群前，确认 `replication.location-labels` 已配置。

4.11.2 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 4 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。

但假如 isolation-level 设置不为空，值为 zone，这样就规定了 Region 副本在物理层面上的最低隔离要求，也就是说 PD 一定会保证同一Region 的副本分散于不同的zone 之上。即便遵循此隔离限制会无法满足 max-replicas 的多副本要求，PD 也不会进行相应的调度。例如，当前存在TiKV 集群的三个机房z1/z2/z3，在三副本的设置下，PD 会将同一Region 的三个副本分别分散调度至这三个机房。若此时z1 整个机房发生了停电事故并在一段时间后（由max-store-down-time 控制，默认为 30 分钟）仍然不能恢复，PD 会认为z1 上的 Region 副本不再可用。但由于 isolation-level 设置为了 zone，PD 需要严格保证不同的Region 副本不会落到同一zone 上。此时的z2 和 z3 均已存在副本，则 PD 在 isolation-level 的最小强制隔离级别限制下便不会进行任何调度，即使此时仅存在两个副本。

类似地，isolation-level 为 rack 时，最小隔离级别便为同一机房的不同rack。在此设置下，如果能在zone 级别保证隔离，会首先保证zone 级别的隔离。只有在zone 级别隔离无法完成时，才会考虑避免出现在同一 zone 同一 rack 的调度，并以此类推。

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。

4.12 TiDB 白名单插件

TiDB 提供了白名单插件。使用该插件，你不需要在自己的业务层开发管理白名单功能，也不需要使用 MySQL 的用户功能逐个添加用户、管理权限，而是可以直接通过配置插件管控白名单，提高管理效率。

本文档介绍了如何使用白名单插件。

4.12.1 获取插件

请联系 PingCAP 销售获取插件安装包。

4.12.2 部署插件

你可以使用TiDB Operator 或者TiUP 部署白名单插件。

4.12.2.1 使用TiDB Operator 部署插件

配置TidbCluster CR。

```
tidb:
  additionalContainers:
  - command:
    - sh
    - -c
    - touch /var/log/tidb/tidb-whitelist.log; tail -n0 -F /var/log/tidb/tidb-whitelist.log;
    image: busybox:1.26.2
```

```
    imagePullPolicy: IfNotPresent
    name:  whitelistlog
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 20m
        memory: 5Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy:  File
    volumeMounts:
    - mountPath: /var/log/tidb
      name: slowlog
  baseImage: &lt;your_private_docker_hub_username>/tidb-ent:v5.4.0
  plugins:
  - whitelist-1
```

4.12.2.2 使用TiUP 部署插件

4.12.2.2.1 在已部署集群上追加插件

1. 在所有目标服务器建立文件夹。

```
tiup cluster exec cluster_name --command "mkdir {{.DeployDir}}/plugin" -R tidb
```

2. 传输插件到所有目标服务器。

```
tiup cluster push cluster_name ~/Download/whitelist-1.so {{.DeployDir}}/plugin/whitelist-1.
    <↩> so -R tidb
```

3. 修改TiDB 配置。

```
tiup cluster edit-config cluster_name
```

在每个TiDB 节点下的config 字段中，配置plugin.dir 为对应绝对路径，plugin.load 为插件名。

“‘yaml tidb_servers:

- host: 172.16.7.213 ssh_port: 22 port: 4000 status_port: 10080 deploy_dir: /tidb-deploy/tidb-4000 log_dir: /tidb-deploy/tidb-4000/log arch: amd64 os: linux config: plugin.dir: /tidb-deploy/tidb-4000/plugin plugin.load: whitelist-1 “‘

4. 滚动重启所有TiDB 节点。

```
tiup cluster reload cluster_name -R tidb
```

建议：

- 步骤 3 中，如果所有节点部署路径完全相同，可直接在 server_configs.tidb 中进行全局配置，避免手动修改每一个节点。如果路径不完全相同，可以修改步骤 1、2 中 plugin 的路径为某一相同路径。
- 如果不是所有TiDB 节点都要部署插件，通过-N 参数来指定节点（ tiup cluster display 显示的ID ）。

4.12.2.2.2 安装时部署

安装时部署与“在已部署集群上追加插件” 的操作步骤相似：在部署前的拓扑文件中配置好plugin.load 和 plugin.dir，在启动集群前执行 `tiup cluster exec` 和 `tiup cluster push`。

4.12.2.2.3 扩容

按照常规操作正常扩容节点，然后参考“在已部署集群上追加插件”的操作步骤，通过-N 参数指定刚扩容的节点。

请注意，你需要在扩容进行期间将插件安装文件（如上面例子中的whitelist-1.so ）上传到目标服务器，而且要确保在扩容结束之前传完，否则扩容会报错。

如果扩容报错，你可以执行以下操作重新安装插件：

1. 重新进行扩容。
2. 在扩容完成后上传插件安装文件。
3. 执行 `tiup cluster start cluster_name` 命令重启。

4.12.2.2.4 升级集群

1. 删除老插件。

```
tiup cluster exec cluster_name --command "rm {{.DeployDir}}/plugin/whitelist-1.so" -R tidb
```

2. 传输新插件到所有目标服务器。为了避免修改配置，使用相同名称。

```
tiup cluster push cluster_name ~/Download/whitelist-1_new.so {{.DeployDir}}/plugin/whitelist
    <-> -1.so -R tidb
```

   说明：域名不能包含“-” （如 tidb-server1 ），否则该命令会执行失败。

3. 升级集群。

```
tiup cluster upgrade cluster_name version
```

4.12.3 查看状态

使用 MySQL 的 `show plugins` 命令，可以查看当前插件加载和启用状态信息：

- Name: 插件名称

- Status: 插件状态, 由加载状态(uninitialized, ready) 和启用状态(enable, disable) 组成
- Type: 插件类型
- Library: 插件二进制文件位置
- Version: 插件版本

```
mysql> show plugins;
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
| Name      | Status       | Type  | Library                                                                                                   | License | Version |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
| whitelist | Ready-enable | Audit | /media/genius/OS/project/src/github.com/pingcap/enterprise-plugin/whitelist/whitelist-1.so                |         | 1       |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 217: Show Plugins

4.12.4 启用和禁用插件

开启白名单以后，TiDB 会根据用户IP 进行过滤，只有属于白名单的IP 才允许登录。启用或禁用命令在可以在任意一个TiDB 节点执行。执行成功后，会作用于集群中所有TiDB 节点。

如不希望过滤IP，可以禁用白名单插件。

要禁用和重启用白名单插件，分别执行以下相应的admin plugins 命令即可：

```
mysql> admin plugins disable whitelist;
mysql> admin plugins enable whitelist;
```

```
mysql> admin plugins enable whitelist;
Query OK, 0 rows affected (0.00 sec)

mysql> show plugins;
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
| Name      | Status       | Type  | Library                                                                                                   | License | Version |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
| whitelist | Ready-enable | Audit | /media/genius/OS/project/src/github.com/pingcap/enterprise-plugin/whitelist/whitelist-1.so                |         | 1       |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
1 row in set (0.01 sec)

mysql> admin plugins disable whitelist;
Query OK, 0 rows affected (0.00 sec)

mysql> show plugins;
+-----------+---------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
| Name      | Status        | Type  | Library                                                                                                   | License | Version |
+-----------+---------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
| whitelist | Ready-disable | Audit | /media/genius/OS/project/src/github.com/pingcap/enterprise-plugin/whitelist/whitelist-1.so                |         | 1       |
+-----------+---------------+-------+-----------------------------------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 218: Disable Plugins

你可以使用 MySQL 的 `show plugins` 命令查看插件的启用状态信息。

4.12.5 使用和验证插件

白名单插件启用后，TiDB 会从 `mysql.whitelist` 系统表读取白名单信息。

```
mysql> show create table whitelist;
+-----------+-------------------------------------------------------------
-----------------------------------------------------+
| Table     | Create Table
                                                     |
+-----------+-------------------------------------------------------------
-----------------------------------------------------+
| whitelist | CREATE TABLE `whitelist` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(16) DEFAULT NULL,
  `list` text DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-----------+-------------------------------------------------------------
-----------------------------------------------------+
1 row in set (0.00 sec)
```

图 219: Plugin Allowlist

白名单表的list字段格式为一个字符串，以单个英文逗号“,”为分隔符，连接一串的IP 网段。IP 网段以CIDR 格式表示，比如 192.0.2.1/24。具体使用方法，请参考Go 的 ParseCIDR 函数。

白名单插件一般配合前端Web 使用。手动的验证操作如下：

1. 添加一条白名单记录，允许 192.0.2.0 网段和 127.0.0.0 网段的IP 访问服务器：

```
mysql> insert into mysql.whitelist (name, list) values ('default',
    <-> '192.0.2.1/24,127.0.0.1/24');
Query OK, 1 row affected (0.04 sec)
```

2. 配置生效后，从非白名单的IP 机器去连接TiDB 的时候会报错：

```
bin git:(master) mysql -h zenlife.tk -P 4000 -u root
ERROR 2013 (HY000): Lost connection to MySQL server at 'reading initial communication packet
    <-> ', system error: 0
```

4.13 TiDB 审计插件

TiDB 审计插件(TiDB Audit Plugin) 记录TiDB 服务器上的活动，这些活动应遵守企业的审计规范。对于每个用户session，该插件记录用户访问服务器的信息（即用户名和 IP 地址）、执行的 SQL 查询、访问的数据库和表、环境变量变更、执行的 DDL 操作等信息。审计信息记录在一个日志文件中以备将来使用。

本文介绍了如何部署和使用TiDB 审计插件。

4.13.1 获取插件

请联系 PingCAP 销售获取插件安装包文件。

4.13.2 部署审计插件

你可以使用TiDB Operator 或者TiUP 来部署审计插件。

4.13.2.1 使用TiDB Operator 部署审计插件

配置TidbCluster CR。

```
tidb:
  additionalContainers:
  - command:
    - sh
    - -c
    - touch /var/log/tidb/tidb-audit.log; tail -n0 -F /var/log/tidb/tidb-audit.log;
    image: busybox:1.26.2
    imagePullPolicy: IfNotPresent
    name: auditlog
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 20m
        memory: 5Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy:  File
    volumeMounts:
    - mountPath: /var/log/tidb
      name: slowlog
  baseImage: &lt;your_private_docker_hub_username>/tidb-ent:v5.4.0
  plugins:
  - audit-1
```

4.13.2.2 使用TiUP 部署审计插件

本节介绍了几种使用TiUP 部署审计插件的场景：

- 在已部署集群上追加插件
- 安装时部署
- 扩容
- 升级集群

4.13.2.2.1 在已部署集群上追加插件

1. 在所有目标服务器上建立文件夹。

```
tiup cluster exec cluster_name --command "mkdir {{.DeployDir}}/plugin" -R tidb
```

2. 传输插件到所有目标服务器。

```
tiup cluster push cluster_name ~/Download/audit-1.so {{.DeployDir}}/plugin/audit-1.so -R
    ↪ tidb
```

3. 修改TiDB 配置。

```
tiup cluster edit-config cluster_name
```

在每个TiDB 节点下的config 字段中，配置plugin.dir 为对应的绝对路径，plugin.load 为插件名。

```
tidb_servers:
-  host: 172.16.7.213
   ssh_port: 22
   port: 4000
   status_port: 10080
   deploy_dir: /tidb-deploy/tidb-4000
   log_dir: /tidb-deploy/tidb-4000/log
   arch: amd64
   os: linux
   config:
     plugin.dir: /tidb-deploy/tidb-4000/plugin
     plugin.load: audit-1
```

4. 滚动重启所有TiDB 节点。

```
tiup cluster reload cluster_name -R tidb
```

建议：

- 在步骤 3 中，如果所有节点部署路径完全相同，可直接在 server_configs.tidb 中进行全局配置，避免手动修改每一个节点。如果路径不完全相同，可以修改步骤 1、2 中 plugin 的路径为某一相同路径。
- 如果不是所有TiDB 节点都要部署插件，则通过-N 参数来指定节点（ tiup cluster display 显示的ID ）。

4.13.2.2.2 安装时部署

安装时部署与“在已部署集群上追加插件” 的操作步骤相似：在部署前的拓扑文件中配置好plugin.load 和 plugin.dir，在启动集群前执行 `tiup cluster exec` 和 `tiup cluster push`。

4.13.2.2.3 扩容

按照常规操作正常扩容节点，然后参考“在已部署集群上追加插件”的操作步骤，通过-N 参数指定刚扩容的节点。

请注意，你需要在扩容进行期间将插件安装文件（如上面例子中的audit-1.so）上传到目标服务器，而且要确保在扩容结束之前传完，否则扩容会报错。

如果扩容报错，你可以执行以下操作重新安装插件：

1. 重新进行扩容。
2. 在扩容完成后上传插件安装文件。
3. 执行 tiup cluster start cluster_name 命令重启。

4.13.2.2.4 升级集群

1. 删除老插件。

```
tiup cluster exec cluster_name --command "rm {{.DeployDir}}/plugin/audit-1.so" -R tidb
```

2. 传输新插件到所有目标服务器。为了避免修改配置，使用相同名称。

```
tiup cluster push cluster_name ~/Download/audit-1_new.so {{.DeployDir}}/plugin/audit-1.so -R
    <⟷ tidb
```

   说明：域名不能包含“-” （如 tidb-server1），否则该命令会执行失败。

3. 升级集群。

```
tiup cluster upgrade cluster_name version
```

4.13.2.3 插件的Events

TiDB 审计插件有以下几种Events。

4.13.2.3.1 CONNECTION: Connection events and authentication events.

CONNECTION logs 示例：

- A successful authentication

  [2022/04/18 21:50:29.669 +08:00] [INFO] [logger.go:70] [ID=16502898290] [TIMESTAMP=2022/04/18 21:50:29.669 +08:00] [EVENT_CLASS=CONNECTION] [EVENT_SUBCLASS=Disconnect] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=] [ROWS=0] [CLIENT_PORT=39290] [CONNECTION_ID=11] [CONNECTION_TYPE=Socket] [SERVER_ID=1] [SERVER_PORT=4000] [DURATION=4522.55501] [SERVER_OS_LOGIN_USER=] [OS_VERSION=“Centos 6”] [CLIENT_VERSION=] [SERVER_VERSION=v5.2.0] [AUDIT_VERSION=] [SSL_VERSION=v1.2.0] [PID=32638] [Reason=]

- A failed authentication

  [2022/04/18 21:50:40.198 +08:00] [INFO] [logger.go:70] [ID=16502898400] [TIMESTAMP=2022/04/18 21:50:40.198 +08:00] [EVENT_CLASS=CONNECTION] [EVENT_SUBCLASS=Reject] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=ccd] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=] [ROWS=0] [CLIENT_PORT=39296] [CONNECTION_ID=15] [CONNECTION_TYPE=Socket] [SERVER_ID=1] [SERVER_PORT=4000] [DURATION=0] [SERVER_OS_LOGIN_USER=] [OS_VERSION=”Centos 6“] [CLIENT_VERSION=] [SERVER_VERSION=v5.2.0] [AUDIT_VERSION=] [SSL_VERSION=v1.2.0] [PID=32638] [Reason=“[server:1045]Access denied for user ‘ccd’@‘127.0.0.1’ (using password: NO)”]

4.13.2.3.2 TABLE_ACCESS: SQL queries related to tables.

TABLE_ACCESS logs 示例：

- Queries on a table (`select * from t`)

  [2022/04/18 22:00:47.085 +08:00] [INFO] [logger.go:70] [ID=16502904470] [TIMESTAMP=2022/04/18 22:00:47.085 +08:00] [EVENT_CLASS=TABLE_ACCESS] [EVENT_SUBCLASS=Select] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[test]”] [TABLES=“t”] [SQL_TEXT=“select * from t”] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=Select]

- Drop a table (`drop table t`)

  [2022/04/18 22:02:40.775 +08:00] [INFO] [logger.go:70] [ID=16502905600] [TIMESTAMP=2022/04/18 22:02:40.775 +08:00] [EVENT_CLASS=TABLE_ACCESS] [EVENT_SUBCLASS=DropTable] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[test]”] [TABLES=“[t,t1]”] [SQL_TEXT=“drop table t，t1”] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=DropTable]

> 注意：
>
> - 默认情况下，TABLE_ACCESS events 不会被记录在日志中。请参考本文档中 “修改 TABLE_ACCESS 相关配置” 了解如何配置。
> - 失败的 statements 不会被记录到日志中。

4.13.2.3.3 GENERAL: SQL queries that don’t access tables.

GENERAL logs 示例：

- `show tables` statement:

  [2022/04/18 22:10:34.845 +08:00] [INFO] [logger.go:70] [ID=16502910340] [TIMESTAMP=2022/04/18 22:10:34.845 +08:00] [EVENT_CLASS=GENERAL] [EVENT_SUBCLASS=] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=“show tables”] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=Show]

- `begin` statement:

  [2022/04/18 22:06:53.198 +08:00] [INFO] [logger.go:70] [ID=16502908130] [TIMESTAMP=2022/04/18 22:06:53.198 +08:00] [EVENT_CLASS=GENERAL] [EVENT_SUBCLASS=] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1]

[USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=begin] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=Begin]

4.13.3 查看状态

通过 MySQL 的 `show plugins` 命令可以查看当前插件加载和启用状态信息：

- Name：插件名称
- Status：插件状态，由加载状态 (uninitialized, ready) 和启用状态(enable, disable) 组成
- Type：插件类型
- Library：插件二进制文件位置
- Version：插件版本

```
mysql> show plugins;
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| Name  | Status       | Type  | Library                                                                           | License | Version |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| audit | Ready-enable | Audit | /home/robi/code/go/src/github.com/pingcap/enterprise-plugin/audit/audit-1.so |         | 1       |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 220: Show Plugins

4.13.4 启用和禁用插件

审计插件加载后，默认会对 CONNECTION 和GENERAL 事件进行审计。启用或禁用命令在任意一个 TiDB 节点执行成功后，会作用于集群中所有TiDB 节点。

执行以下命令启用审计插件：

```
mysql> admin plugins enable audit;
```

如果不想对这两种事件做审计，执行如下命令禁用审计插件：

```
mysql> admin plugins disable audit;
```

```
mysql> admin plugins disable audit;
Query OK, 0 rows affected (0.03 sec)

mysql> show plugins;
+-------+---------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| Name  | Status        | Type  | Library                                                                           | License | Version |
+-------+---------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| audit | Ready-disable | Audit | /home/robi/code/go/src/github.com/pingcap/enterprise-plugin/audit/audit-1.so |         | 1       |
+-------+---------------+-------+-----------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)

mysql> admin plugins enable audit;
Query OK, 0 rows affected (0.00 sec)

mysql> show plugins;
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| Name  | Status       | Type  | Library                                                                           | License | Version |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| audit | Ready-enable | Audit | /home/robi/code/go/src/github.com/pingcap/enterprise-plugin/audit/audit-1.so |         | 1       |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 221: Disable Plugins

4.13.5 修改TABLE_ACCESS相关配置

对于TABLE_ACCESS 事件，默认不会进行审计。如需使用，需要进行相关配置。

TABLE_ACCESS 支持通过用户名+数据库+表名+语句类型这 4 个维度的组合对审计进行细粒度控制。修改配置需要运行 flush，使其在所有 TiDB 节点生效。

4.13.5.1 修改配置表tidb_audit_table_access

在首次使用审计插件时，会在TiDB 的 MySQL 库中创建一个名为tidb_audit_table_access 的插件配置表。你可以通过修改配置表来控制审计插件对table_access 的审计。

```
mysql> show create table mysql.tidb_audit_table_access;
CREATE TABLE `tidb_audit_table_access` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user` varchar(32) NOT NULL COMMENT 'username need be audit(regex)',
  `db` varchar(32) NOT NULL COMMENT 'db name need be audit(regex)',
  `tbl` varchar(32) NOT NULL COMMENT 'table name need be audit(regex)',
  `access_type` varchar(128) NOT NULL COMMENT 'table access types split by comma',
  `create_time` timestamp NULL DEFAULT NULL COMMENT 'record create time',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT
      <⟶ 'record last update time',
  PRIMARY KEY (`id`),
  UNIQUE KEY `user` (`user`,`db`,`tbl`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

相关参数解释如下：

- user：开启table_access 事件审计的用户。
- db：开启table_access 事件审计的数据库。
- tbl：开启table_access 事件审计的表名。
- access_type：开启table_access 事件审计语句类型，类型同上面提到的 subClass，支持多个逗号分割，空字符串则代表所有都审计。

注意：user、db、tbl 使用正则且区分大小写，.* 代表table-access 对所有user、db 或 tbl 都开启。

例如，如果希望对任意用户访问任意库和表的所有 SQL 进行审计，可以插入以下数据：

```
mysql> insert into mysql.tidb_audit_table_access (user, db, tbl, access_type) values ('.*', '.*',
    <⟶ '.*', '');
```

如果希望对任意用户访问ac 库 account 表的所有 SQL 进行审计，可以插入以下数据：

```
mysql> insert into mysql.tidb_audit_table_access (user, db, tbl, access_type) values (".*", 'ac',
    <⟶ ^account$, '');
```

如果希望对以 op_ 开头的用户名访问全部数据库 account 表的所有 SQL 进行审计，可以插入以下数据：

```
mysql> insert into mysql.tidb_audit_table_access (user, db, tbl, access_type) values ('op_.*',
    <-> '.*', '.*', '');
```

注意匹配的规则不是通配符风格的。例如，如果 user 设置为 abc*，会匹配到 “ab”，“abc”，“abcc”，“abccc” 等。

支持的access_type 请参考下面列表中的 SubEvent，支持多个逗号分割。空字符串则代表所有都审计。

- Insert
- Select
- Update
- Replace
- Delete
- LoadData
- Alter
- DropTable
- CreateTable

4.13.5.2 刷新配置

对mysql.tidb_audit_table_access 表进行修改后，需要执行 flush tidb plugins audit; 命令进行刷新，以通知集群中所有 TiDB 节点从表中load 最新配置。如该命令执行失败，则需要进行重试执行 flush 操作。

```
mysql> flush tidb plugins audit;
Query OK, 0 rows affected (0.00 sec)
```

4.13.6 使用插件输出审计日志

审计插件加载后，会根据配置把不同类型的审计日志输出到和TiDB 配置项 slow-query-file 相同的目录下，日志文件名称为 tidb-audit.log。

4.13.6.1 支持的审计事件类型

下表列出了目前支持的审计事件类型。

Event Class

SubEvent

开关

备注

CONNECTION

Connected

Disconnect

ChangeUser

PreAuth

Reject

默认开启。

只能通过禁用插件关闭。

连接事件和鉴权事件。

TABLE_ACCESS

Insert

Select

Update

Replace

Delete

LoadData

Alter

DropTable

CreateTable

默认关闭。

需要通过修改配置表指定db - user - table - 事件来开启。

带 table 的 SQL。

会对 SQL sql 中的参数脱敏处理替换为“?”。

GENERAL

- 默认开启。
  只能通过禁用插件关闭。
  没有 table 访问的其他 SQL。

4.13.7 日志输出

所有事件类型的日志输出中，头部分是通用的字段, 在跳过前 3 个由日志库输出的字段后，有以下字段：

- ID：事件ID
- TIMESTAMP：事件时间
- EVENT_CLASS：事件类型
- EVENT_SUBCLASS：事件子类型
- STATUS_CODE：语句响应状态
- COST_TIME：语句耗时
- HOST：服务器 IP

- CLIENT_IP：客户端 IP
- USER：登录用户名
- DATABASE：事件相关 db
- TABLES：事件相关的表名
- SQL_TEXT：脱敏后的 SQL
- ROWS：影响行数，无则为 0

日志输出中，会出现冗余或没数据也固定输出的情况，主要目的是方便当前日志分析系统处理。

CONNECTION、 TABLE_ACCESS、 GENERAL 还提供了补充输出的内容。

4.13.7.1 CONNECTION 相关日志

CONNECTION 连接事件补充输出的内容如下:

- CLIENT_PORT：客户端端口号
- CONNECTION_ID：连接 ID
- CONNECTION_TYPE：是通过 socket 还是 unix-socket 连接
- SERVER_ID：TiDB Server ID
- SERVER_PORT：TiDB server 监听 MySQL 协议端口
- SERVER_OS_LOGIN_USER：TiDB 进程启动系统用户名
- OS_VERSION：TiDB Server 所在操作系统版本
- SSL_VERSION：TiDB 当前 SSL 版本
- PID：TiDB 进程 PID

示例：

[2019/07/24 22:13:30.099 +08:00] [INFO] [logger.go:71] [ID=“15640224010”] [TIMESTAMP=2019/07/24 22:13:30.099 +08:00] [EVENT_CLASS=CONNECTION] [EVENT_SUBCLASS=Disconnect] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[test]”] [TABLES=“[]”] [SQL_TEXT=] [ROWS=0] [CLIENT_PORT=45814] [CONNECTION_ID=1] [CONNECTION_TYPE=Socket] [SERVER_ID=1] [SERVER_PORT=4000] [DURATION=1233372.167907] [SERVER_OS_LOGIN_USER=robi] [OS_VERSION=“Linux 5.0.0-20-generic.x86_64”] [CLIENT_VERSION=] [SERVER_VERSION=v2.1.15-16-g136e334df-dirty] [AUDIT_VERSION=] [SSL_VERSION=v1.2.0] [PID=24108]

4.13.7.1.1 TABLE_ACCESS 相关日志

TABLE_ACCESS 事件补充输出的内容如下：

- CONNECTION_ID：连接 ID
- COMMAND：MySQL 协议的command 类型
- SQL_STATEMENTS：SQL 语句类型
- PID：TiDB 进程所在 PID

示例：

[2019/07/24 22:35:01.640 +08:00] [INFO] [logger.go:71] [ID=“15640226010”] [TIMESTAMP=2019/07/24 22:35:01.640 +08:00] [EVENT_CLASS=TABLE_ACCESS] [EVENT_SUBCLASS=Insert] [STATUS_CODE=0] [COST_TIME=31581.623] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[mysql]”] [TABLES=“[t1]”] [SQL_TEXT=“insert into t1 (a) values (?)”] [ROWS=1] [CONNECTION_ID=2] [CLIENT_PORT=47430] [PID=24108] [COMMAND=Query] [SQL_STATEMENTS=Insert]

4.13.7.1.2 GENERAL 相关日志

GENERAL 事件补充输出的内容如下：

- CONNECTION_ID：连接 ID
- COMMAND：MySQL 协议的command 类型
- SQL_STATEMENTS：SQL 语句类型
- PID：TiDB 进程所在 PID

示例：

[2019/07/24 22:31:22.612 +08:00] [INFO] [logger.go:71] [ID=“15640334010”] [TIMESTAMP=2019/07/24 22:31:22.612 +08:00] [EVENT_CLASS=GENERAL] [EVENT_SUBCLASS=] [STATUS_CODE=0] [COST_TIME=67.6] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=“set @@tidb_max_chunk_size = ?”] [ROWS=0] [CONNECTION_ID=2] [CLIENT_PORT=47430] [PID=24108] [COMMAND=Query] [SQL_STATEMENTS=Set]

4.13.8 清除审计日志

TiUP 对于审计插件产生的审计日志不会自动清除。如果需要清除审计日志，执行以下命令：

```
tiup cluster clean &lt;cluster-name> --audit-log
```

> 注意：
>
> - 执行以上命令默认删除所有审计日志。
> - 清除操作不可逆，请先备份数据后使用。

4.14 内部组件介绍

4.14.1 TiDB 后端任务分布式框架

> 警告：
> 当前该功能为实验特性，不建议在生产环境中使用。

TiDB 采用计算存储分离架构，具有出色的扩展性和弹性的扩缩容能力。从 v7.1.0 开始，TiDB 引入了一个后端任务分布式执行框架，以进一步发挥分布式架构的资源优势。该框架的目标是实现对所有后端任务的统一调度与分布式执行，并为接入的后端任务提供统一的资源管理能力，从整体和单个后端任务两个维度提供资源管理的能力，更好地满足用户对于资源使用的预期。

本文档介绍了TiDB 后端任务分布式框架的使用场景与限制、使用方法和实现原理。

> 注意：
>
> 本框架不支持 SQL 查询的分布式执行。

4.14.1.1 使用场景与限制

在数据库中，除了核心的事务型负载任务(TP) 和分析型查询任务(AP)，也存在着其他重要任务，如 DDL 语句、Load Data、TTL、Analyze 和 Backup/Restore 等，即后端任务。这些任务需要处理数据库对象（表）中的大量数据，通常具有如下特点：

- 需要处理一个 schema 或者一个数据库对象（表）中的所有数据。
- 可能需要周期执行，但频率较低。
- 如果资源控制不当，容易对事务型任务和分析型任务造成影响，影响数据库的服务质量。

启用TiDB 后端任务分布式框架能够解决上述问题，并且具有以下三个优势：

- 提供高扩展性、高可用性和高性能的统一能力支持。
- 支持后端任务分布式执行，可以在整个TiDB 集群可用的计算资源范围内进行灵活的调度，从而更好地利用TiDB 集群内的计算资源。
- 提供统一的资源使用和管理能力，从整体和单个后端任务两个维度提供资源管理的能力。

目前，后端任务分布式框架仅支持分布式执行 ADD INDEX，即 DDL 创建索引的场景。例如以下 SQL 语句：

```
ALTER TABLE t1 ADD INDEX idx1(c1);
CREATE INDEX idx1 ON table t1(c1);
```

4.14.1.2 启用前提

使用分布式框架前，你需要启动Fast Online DDL 模式。

1. 调整 Fast Online DDL 相关的系统变量：
    - `tidb_ddl_enable_fast_reorg`：从 TiDB v6.5.0 开始默认打开，用于启用快速模式。
    - `tidb_ddl_disk_quota`：用于控制快速模式可使用的本地磁盘最大配额。
2. 调整 Fast Online DDL 相关的配置项：
    - `temp-dir`：指定快速模式能够使用的本地盘路径。

> 注意：
>
> 在升级到 v6.5.0 及以上版本时，建议你检查 TiDB 的 `temp-dir` 路径是否正确挂载了 SSD 磁盘。该参数是TiDB 的配置参数，设置后需要重启TiDB 才能生效。因此，在升级前提前进行设置，可以避免再次重启。

4.14.1.3 启用步骤

1. 启用分布式框架，只需将tidb_enable_dist_task 设置为 ON：

```
SET GLOBAL tidb_enable_dist_task = ON;
```

   在运行后端任务时，框架支持的语句会采用分布式方式执行。

2. 根据实际需求，调整可能影响 DDL 任务分布式执行的系统变量：

   - tidb_ddl_reorg_worker_cnt：使用默认值 4 即可，建议最大不超过 16。
   - tidb_ddl_reorg_priority
   - tidb_ddl_error_count_limit
   - tidb_ddl_reorg_batch_size：使用默认值即可，建议最大不超过 1024。

> 建议：
> 对于分布式执行 ADD INDEX 语句，只需要设置 tidb_ddl_reorg_worker_cnt。

4.14.1.4 实现原理

TiDB 后端任务分布式框架的架构图如下：

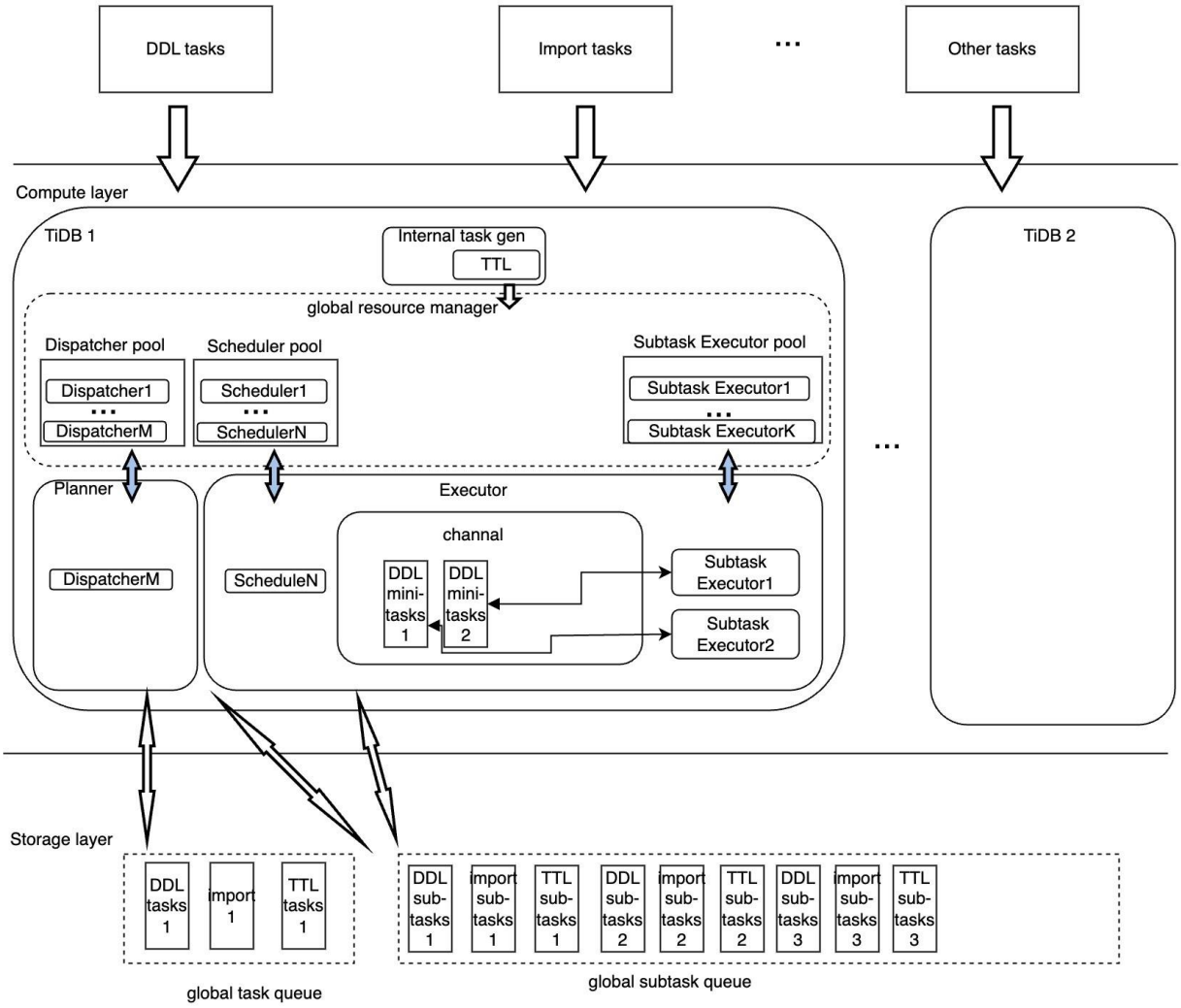


图 222: 后端任务分布式框架的架构

根据上图，分布式框架中任务的执行主要由以下模块负责：

- Dispatcher：负责生成每个任务的分布式执行计划，管理执行过程，转换任务状态以及收集和反馈运行时任务信息等。
- Scheduler：以 TiDB 节点为单位来同步分布式任务的执行，提高后端任务执行效率。
- Subtask Executor：是实际的分布式子任务执行者，并将子任务的执行情况返回给 Scheduler，由 Scheduler 统一更新子任务的执行状态。
- 资源池：通过对上述各种模块中计算资源进行池化，提供量化资源的使用与管理的基础。

4.14.1.5 另请参阅

- DDL 执行原理及最佳实践

5 常见问题解答(FAQ)

5.1 TiDB 产品常见问题

5.1.1 1.1 TiDB 介绍及整体架构

5.1.1.1 1.1.1 TiDB 是什么？

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

5.1.1.2 1.1.2 TiDB 整体架构

参见TiDB 整体架构，以及TiDB 数据库的存储、 计算与调度。

5.1.1.3 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。

5.1.1.4 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。

5.1.1.5 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。

5.1.1.6 1.1.6 TiDB 和 MySQL 兼容性如何？

TiDB 支持绝大部分 MySQL 5.7 的语法，但目前还不支持触发器、存储过程、自定义函数等。详情参见与 MySQL 兼容性对比。

5.1.1.7 1.1.7 TiDB 支持分布式事务吗？

支持。无论是一个地方的几个节点，还是跨多个数据中心的多个节点，TiDB 均支持ACID 分布式事务。

TiDB 事务模型灵感源自Google Percolator 模型，主体是一个两阶段提交协议，并进行了一些实用的优化。该模型依赖于一个时间戳分配器，为每个事务分配单调递增的时间戳，这样就检测到事务冲突。在TiDB 集群中，PD 承担时间戳分配器的角色。

5.1.1.8 1.1.8 TiDB 支持哪些编程语言？

只要支持 MySQL Client/Driver 的编程语言，都可以直接使用TiDB。

5.1.1.9 1.1.9 TiDB 是否支持其他存储引擎？

是的，除了TiKV 之外，TiDB 还支持一些单机存储引擎，比如 UniStore 和 MockTiKV。注意，TiDB 后续版本可能不再支持 MockTiKV。

要查看TiDB 支持的存储引擎，可使用以下命令：

```
./bin/tidb-server -h
```

返回结果如下：

```
Usage of ./bin/tidb-server:
  -L string
        log level: info, debug, warn, error, fatal (default "info")
  -P string
        tidb server port (default "4000")
  -V    print version information and exit (default false)
.........
  -store string
        registered store name, [tikv, mocktikv, unistore] (default "unistore")
  ......
```

5.1.1.10 1.1.10 除了官方文档，有没有其他TiDB 知识获取途径？

- 官方文档：获取TiDB 相关知识最主要、最及时的途径。
- 官方博客：了解产品技术解读、观点洞察、案例实践。
- AskTUG 社区论坛：与社区用户、技术专家互动交流。
- PingCAP Education：学习线上课程，获得数据库能力认证。

5.1.1.11 1.1.11 TiDB 用户名长度限制？

在 TiDB 中，用户名最长为 32 个字符。

5.1.1.12 1.1.12 TiDB 中列数和行大小的限制是多少？

- TiDB 中默认的最大列数为 1017。你可以调整这个限制，最大可调整到 4096 列。
- TiDB 中默认单行大小不超过 6 MB。你可以调整这个限制，最大可调整到 120 MB。

更多信息，请参考TiDB 限制。

5.1.1.13 1.1.13 TiDB 是否支持XA？

虽然TiDB 的JDBC 驱动用的就是 MySQL 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。

5.1.1.14 1.1.14 TiDB 如何在不影响性能的情况下支持对列存储引擎(TiFlash) 的高并发 INSERT 或 UPDATE 操作？

- TiFlash 引入了 DeltaTree 这种特殊结构来处理列存引擎的修改。
- TiFlash 作为Raft Group 中的 Learner 角色，不参与log commit 选举，也不会写入数据。这意味着 DML 操作不需要等待TiFlash 的确认，所以TiFlash 不会影响 OLTP 的性能。另外，TiFlash 和TiKV 分开部署在不同的实例上，不会相互影响。

5.1.1.15 1.1.15 TiFlash 是否具有最终一致性？

是的，TiFlash 默认保持数据强一致性。

5.1.2 1.2 TiDB 原理

5.1.2.1 1.2.1 存储TiKV 详细解读

三篇文章了解TiDB 技术内幕- 说存储

5.1.2.2 1.2.2 计算TiDB 详细解读

三篇文章了解TiDB 技术内幕- 说计算

5.1.2.3 1.2.3 调度 PD 详细解读

三篇文章了解TiDB 技术内幕- 谈调度

5.2 SQL 操作常见问题

本文档介绍TiDB 中常见的 SQL 操作问题。

5.2.1 TiDB 是否支持二级键？

支持。你可以在具有唯一二级索引的非主键列上设置`NOT NULL` 约束。在这种情况下，该列用作二级键。

5.2.2 TiDB 在对大表执行 DDL 操作时，性能表现如何？

TiDB 在对大表执行 DDL 操作时，一般不会有什么问题。TiDB 支持在线 DDL 操作，且这些 DDL 操作不会阻塞 DML 操作。

对于添加列、删除列或删除索引等 DDL 操作，TiDB 可以快速完成这些操作。

对于添加索引等 DDL 操作，TiDB 需要进行回填(backfill) 操作，这个过程需要较长的时间（取决于表的大小）和额外的资源消耗。对在线业务的影响可调节。TiDB 可以通过多线程进行backfill，资源消耗可通过以下系统变量进行设置：

- tidb_ddl_reorg_worker_cnt
- tidb_ddl_reorg_priority
- tidb_ddl_error_count_limit
- tidb_ddl_reorg_batch_size

5.2.3 如何选择正确的查询计划？是否需要使用优化器提示？还是可以使用提示？

TiDB 包含一个基于成本的优化器。在大多数情况下，优化器会为你选择最优的查询计划。如果优化器工作欠佳，你可以使用优化器提示来干预优化器。

另外，你还可以使用执行计划绑定来为特定的 SQL 语句固定查询计划。

5.2.4 如何阻止特定的 SQL 语句执行（或者将某个 SQL 语句加入黑名单）？

你可以使用MAX_EXECUTION_TIME Hint 来创建SQL 绑定，将特定语句的执行时间限制为一个较小的值（例如 1ms）。这样，语句就会在超过限制时自动终止。

例如，要阻止执行 SELECT * FROM t1, t2 WHERE t1.id = t2.id，可以使用以下 SQL 绑定将语句的执行时间限制为 1ms：

```
CREATE GLOBAL BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id
USING
    SELECT /*+ MAX_EXECUTION_TIME(1) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> MAX_EXECUTION_TIME 的精度大约为 100ms。在TiDB 终止 SQL 语句之前，TiKV 中的任务可能已经开始执行。为了减少这种情况下TiKV 的资源消耗，建议将系统变量tidb_enable_paging 的值设置为 ON。

删除该 SQL 绑定可以移除限制。

```
DROP GLOBAL BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id;
```

5.2.5 TiDB 对哪些 MySQL variables 兼容？

详细可参考系统变量。

5.2.6 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，就无法确定该语句返回结果的顺序。在以下示例中，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)
```

在以下示例中，`order by a` 能确定a 列的顺序，但不能确定b 列的顺序。

```
> select * from t order by a;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    1 |
+------+------+
3 rows in set (0.00 sec)
```

在 TiDB 中，你还可以使用系统变量tidb_enable_ordered_result_mode 来指定是否对最终的输出结果进行自动排序。

5.2.7 TiDB 是否支持 SELECT FOR UPDATE？

支持。当TiDB 使用悲观锁（自 TiDB v3.0.8 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。

详情参考SELECT 语句语法元素说明。

5.2.8 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。

5.2.9 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。

在使用乐观事务并开启事务重试的情况下，默认限制 5000，可通过stmt-count-limit 调整。

5.2.10 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 server 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。详情可参考自增ID和AUTO_INCREMENT。

5.2.11 如何在TiDB 中修改 sql_mode？

TiDB 支持在会话或全局作用域上修改sql_mode 系统变量。

- 对全局作用域变量的修改，设置后将作用于集群中的其它服务器，并且重启后更改依然有效。因此，你无需在每台TiDB 服务器上都更改 sql_mode 的值。
- 对会话作用域变量的修改，设置后只影响当前会话，重启后更改消失。

5.2.12 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？

问题原因：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。

解决办法：

- 增加选项 -Dsqoop.export.records.per.statement=10，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10  \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大TiDB 的单个事物语句数量限制，不过此操作会导致内存增加。详情参见SQL 语句的限制。

5.2.13 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考使用 AS OF TIMESTAMP 语法读取历史数据。

5.2.14 TiDB 中删除数据后会立即释放空间吗？

在TiDB 中使用 DELETE，TRUNCATE 和 DROP 语句删除数据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，但不会立即释放空间，而是等到后续进行compaction 时释放空间。

5.2.15 删除数据后查询速度为何会变慢？

删除大量数据后，会有很多无用的 key 存在，影响查询效率。要解决该问题，可以尝试开启Region Merge 功能，具体可参考最佳实践中的删除数据部分。

5.2.16 对数据做删除操作之后，空间回收比较慢，如何处理？

TiDB 采用了多版本并发控制(MVCC)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。为了使并发事务能查看到早期版本的数据，删除数据时 TiDB 不会立即回收空间，而是等待一段时间后再进行垃圾回收(GC)。要配置历史数据的保留时限，你可以修改系统变量`tidb_gc_life_time`的值（默认值为 `10m0s`）。

5.2.17 SHOW PROCESSLIST 是否显示系统进程号？

TiDB 中的 SHOW PROCESSLIST 与 MySQL 中的 SHOW PROCESSLIST 显示内容基本一致，不会显示系统进程号。而返回结果中的ID 表示当前的 session ID。其中TiDB 的 SHOW PROCESSLIST 和 MySQL 的 SHOW PROCESSLIST 区别如下：

- 由于TiDB 是分布式数据库，TiDB server 实例是无状态的 SQL 解析和执行引擎（详情可参考TiDB 整体架构），用户使用 MySQL 客户端登录的是哪个 TiDB server，SHOW PROCESSLIST 就会显示当前连接的这个 TiDB server 中执行的 session 列表，不是整个集群中运行的全部 session 列表；而 MySQL 是单机数据库，SHOW PROCESSLIST 列出的是当前整个 MySQL 数据库的全部执行 SQL 列表。
- 在查询执行期间，TiDB 中的 State 列不会持续更新。由于TiDB 支持并行查询，每个语句可能同时处于多个状态，因此很难显示为某一种状态。

5.2.18 在TiDB 中如何控制或改变 SQL 提交的执行优先级？

TiDB 支持改变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
- DELAYED：该语句为正常优先级语句，TiDB 不强制改变这条语句的优先级，与 tidb_force_priority 设置为 NO_PRIORITY 相同

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 | DELAYED COUNT(*) FROM table_name;
INSERT HIGH_PRIORITY | LOW_PRIORITY | DELAYED INTO table_name insert_values;
DELETE HIGH_PRIORITY | LOW_PRIORITY | DELAYED FROM table_name;
UPDATE HIGH_PRIORITY | LOW_PRIORITY | DELAYED table_reference SET assignment_list WHERE
    ↪ where_condition;
REPLACE HIGH_PRIORITY | LOW_PRIORITY | DELAYED 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。

5.2.19 在 TiDB 中 auto analyze 的触发策略是怎样的？

触发策略：如果一张新表达到 1000 条记录，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）比例大于 `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，即默认开启触发 auto analyze。为了保险起见，在开启 auto analyze 的时候，`tidb_auto_analyze_ratio` 的最小值为 0.3。但是该变量值不能大于等于 `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。

你可以用系统变量`tidb_enable_auto_analyze` 关闭auto analyze。

5.2.20 可以使用 Optimizer Hints 控制优化器行为吗？

在TiDB 中，你可以用多种方法控制查询优化器的默认行为，包括使用 Optimizer Hints 和 SQL 执行计划管理(SPM)。基本用法同 MySQL 中的一致，还包含若干TiDB 特有的用法，例如：`select column_name from table_name use <-> index（index_name）where where_condition;`。

5.2.21 DDL 执行

本节列出了 DDL 语句执行的相关问题。DDL 执行原理的详细说明，参见TiDB 中 DDL 执行原理及最佳实践。

5.2.21.1 各类 DDL 操作的预估耗时是多长？

假设 DDL 操作没有被阻塞，各个TiDB server 能够正常更新 Schema 版本，DDL Owner 节点正常运行。在此情况下，各类 DDL 操作的预估耗时如下：

DDL 操作类型	预估耗时
Reorg DDL：例如 `ADD INDEX`、`MODIFY COLUMN`（Reorg 类型的数据更改）	取决于数据量、系统负载以及 DDL 参数的设置
General DDL（除 Reorg DDL 外的 DDL 类型）：例如 `CREATE DATABASE`、`CREATE TABLE`、`DROP DATABASE`、`DROP TABLE`、`TRUNCATE TABLE`、`ALTER TABLE ADD`、`ALTER TABLE DROP`、`MODIFY COLUMN`（只更改元数据）、`DROP INDEX`	1 秒左右

> **注意：**
>
> 以上为各类操作的预估耗时，请以实际操作耗时为准。

5.2.22 执行 DDL 会慢的可能原因

- 在一个用户会话中，DDL 语句之前有非auto-commit 的 DML 语句，并且该 DML 语句的提交操作比较慢，会导致 DDL 语句执行慢。即执行 DDL 语句前，会先提交之前没有提交的 DML 语句。

- 多个 DDL 语句一起执行的时候，后面的几个 DDL 语句可能会比较慢，因为可能需要排队等待。排队场景包括：
  - 同一类型 DDL 语句需要排队（例如 CREATE TABLE 和 CREATE DATABASE 都是General DDL，两个操作同时执行时，需要排队）。自 TiDB v6.2.0 起，支持并行 DDL 语句，但为了避免 DDL 使用过多TiDB 的计算资源，也有并发度限制，因此会有一定的排队情况。
  - 对同一张表上执行的 DDL 操作存在依赖关系，后面的 DDL 语句需要等待前面的 DDL 操作完成。
- 在集群正常启动后，第一个 DDL 操作的执行时间可能会比较久，可能是因为 DDL 模块在进行 DDL Owner 的选举。
- 终止TiDB 时，TiDB 不能与 PD 正常通信（包括停电的情况），或者用 `kill -9` 命令终止TiDB 导致TiDB 没有及时从 PD 清理注册数据。
- 集群中某个TiDB 与 PD 或者TiKV 之间发生通信问题，即TiDB 不能及时获取最新版本信息。

5.2.22.1 触发 Information schema is changed 错误的原因？

TiDB 在执行SQL 语句时，会根据隔离级别确定一个对象的schema 版本来处理该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，如果遇到正在执行中的DDL 操作，TiDB 可能会报`Information schema is changed` 的错误。

从 v6.4.0 开始，TiDB 实现了元数据锁机制，可以让 DML 语句的执行和 DDL Schema 变更协同进行，可以避免大部分 `Information schema is changed` 错误的发生。

报错的可能原因如下：

- 原因 1：正在执行的 DML 所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。可以通过命令 `admin show ddl job` 查看正在执行的 DDL 操作。
- 原因 2：这个 DML 执行时间很久，而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 1024（此为默认值，可以通过 `tidb_max_delta_schema_count` 变量修改）。
- 原因 3：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。
- 原因 4：TiDB 重启后执行第一个 DDL 操作前，执行 DML 操作，并且在执行过程中遇到了第 1 个 DDL 操作（即在执行第 1 个 DDL 操作前，启动该 DML 对应的事务，且在该 DDL 变更第一个 schema 版本后，提交该 DML 对应的事务），那么这个 DML 会报此错。

以上原因中，只有原因 1 与表有关。原因 1 和原因 2 都不会导致业务问题，相应的 DML 会在失败后重试。对于原因 3，需要检查TiDB 实例和 PD 及 TiKV 的网络情况。

> 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。

5.2.22.2 触发 Information schema is out of date 错误的原因？

在 v6.5.0 之前，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。

5.2.22.3 高并发情况下执行 DDL 时报错的原因？

高并发场景下执行 DDL 语句（比如批量建表）时，极少部分的 DDL 语句可能会由于并发执行时 key 冲突而执行失败。

并发执行 DDL 语句时，建议将 DDL 语句数量保持在 20 以下，否则你需要在应用端重试失败的 DDL 语句。

5.2.22.4 DDL 执行被阻塞的原因

在 TiDB v6.2.0 前，TiDB 按照 DDL 语句类型将 DDL 分配到两个先入先出的队列中，即Reorg DDL 进入 Reorg 队列中，General DDL 进入 general 队列中。由于先入先出以及同一张表上的 DDL 语句需要串行执行，多个 DDL 语句在执行过程中可能会出现阻塞的问题。

例如对于以下 DDL 语句：

- DDL 1：`CREATE INDEX idx on t(a int);`
- DDL 2：`ALTER TABLE t ADD COLUMN b int;`
- DDL 3：`CREATE TABLE t1(a int);`

由于队列先入先出的限制，DDL 3 需要等待 DDL 2 执行。同时又因为同一张表上的 DDL 语句需要串行执行，DDL 2 需要等待 DDL 1 执行。因此，DDL 3 需要等待 DDL 1 先执行完，即使它们操作在不同的表上。

在 TiDB v6.2.0 及之后的版本中，TiDB DDL 模块采用了并发框架。在并发的框架下，不再有同一个队列先进先出的问题，而是从所有 DDL 任务中选出可以执行的 DDL 来执行。并且对Reorg worker 的数量进行了扩充，大概为节点 CPU/4，这使得在并发框架中 TiDB 可以同时为多张表建索引。

无论是新集群还是从旧版本升级的集群，在TiDB v6.2 及以上版本中，TiDB 都会自动使用并发框架，用户无需进行调整。

5.2.22.5 定位 DDL 执行卡住的问题

1. 先排除 DDL 语句通常执行慢的可能原因。
2. 使用以下任一方法找出 DDL owner 节点：
    - 通过 `curl http://{TiDBIP}:10080/info/all` 获取当前集群的 Owner
    - 通过监控 DDL > DDL META OPM 查看某个时间段的 Owner

- 如果Owner 不存在，尝试手动触发Owner 选举：`curl -X POST http://{TiDBIP}:10080/ddl/owner/resign`。
- 如果 Owner 存在，导出Goroutine 堆栈并检查可能卡住的地方。

5.2.23 SQL 优化

5.2.23.1 TiDB 执行计划解读

详细解读理解TiDB 执行计划。

5.2.23.2 统计信息收集

详细解读统计信息。

5.2.23.3 Count 如何加速？

Count 就是暴力扫表，提高并发度能显著提升扫表速度。如要调整并发度，可以使用tidb_distsql_scan_concurrency <⟶ 变量，但调整并发度需要同时考虑 CPU 和 I/O 资源。TiDB 每次执行查询时，都要访问TiKV。在数据量小的情况下，MySQL 的数据都在内存里，而TiDB 还需要进行一次网络访问。

加速建议：

- 提升硬件配置，可以参考部署建议。
- 提升并发度，默认是 10，可以尝试提升到 50，但是一般提升幅度在 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。
- 参考下推计算结果缓存。

5.2.23.4 查看当前 DDL 的进度？

通过 ADMIN SHOW DDL 语句查看当前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
```

从以上返回结果可知，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。

5.2.23.5 如何查看 DDL job？

可以使用 ADMIN SHOW DDL 语句查看正在运行的 DDL 作业。

- ADMIN SHOW DDL JOBS：用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。
- ADMIN SHOW DDL JOBS 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。

5.2.23.6 TiDB 是否支持基于 COST 的优化(CBO)？如果支持，实现到什么程度？

是的，TiDB 基于成本的优化器(CBO) 对代价模型、统计信息进行持续优化。除此之外，TiDB 还支持hash join、soft-merge join 等 join 算法。

5.2.23.7 如何确定某张表是否需要做analyze ？

可以通过 `SHOW STATS_HEALTHY` 来查看 Healthy 字段，一般该字段值小于等于 60 的表需要做analyze。

5.2.23.8 SQL 的执行计划展开成了树，ID 的序号有什么规律吗？这棵树的执行顺序会是怎么样的？

ID 没什么规律，只要是唯一就行。不过在生成执行计划时，有一个计数器，生成一个计划ID 后序号就加 1，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。要理解执行计划，请参考理解TiDB 执行计划。

5.2.23.9 TiDB 执行计划中，task cop 在一个root 下，这个是并行的吗？

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。所以从这个过程来看，root task 和cop task 是并行的，同时存在数据上下游关系。

在执行的过程中，某些时间段也可能是并行的，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。

5.2.24 数据库优化

5.2.24.1 TiDB 参数及调整

详情参考TiDB 配置参数。

5.2.24.2 如何避免热点问题并实现负载均衡？TiDB 中是否有热分区或热范围问题？

要了解热点问题的场景，请参考常见热点问题。TiDB 的以下特性旨在帮助解决热点问题：

- `SHARD_ROW_ID_BITS` 属性。设置该属性后，行ID 会被打散并写入多个Region，以缓解写入热点问题。
- `AUTO_RANDOM` 属性，用于解决自增主键带来的热点问题。
- Coprocessor Cache，针对小表的读热点问题。
- Load Base Split，针对因 Region 访问不均衡（例如小表全表扫）而导致的热点问题。
- 缓存表，针对被频繁访问但更新较少的小热点表。

如果你遇到因热点引起的性能问题，可参考处理热点问题。

5.2.24.3 TiKV 性能参数调优

详情参考TiKV 性能参数调优。

5.3 高可用常见问题

本文档介绍高可用相关的常见问题。

5.3.1 TiDB 数据是强一致的吗？

通过使用Raft 一致性算法，数据在各TiKV 节点间复制为多副本，以确保某个节点宕机时数据的安全性。

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Raft 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，以此来实现强一致。

5.3.2 TiDB 是否提供三中心跨机房多活部署的推荐方案？

从架构来看，TiDB 支持真正意义上的跨中心异地多活。从实现层面看，多地部署方案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TiDB 已经有相似的客户部署方案，参见两地三中心部署方案。

5.4 高可靠常见问题

本文档介绍高可靠相关的常见问题。

5.4.1 TiDB 是否支持数据加密？

支持。要加密传输中的数据，可以在 TiDB 客户端和服务器之间启用 TLS。要加密存储引擎中的数据，可以启用透明数据加密(TDE)。

5.4.2 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 后支持通过TiDB 配置文件中的`server-version` 配置项来修改 server 版本号。

对于 v4.0 及以上版本的集群，如果使用TiUP 部署集群，可以通过 `tiup cluster edit-config <cluster-name>` 修改配置文件中以下部分来设置合适的版本号：

```
server_configs:
  tidb:
    server-version: 'YOUR_VERSION_STRING'
```

修改完成后，使用 `tiup cluster reload <cluster-name> -R tidb` 命令使得以上修改生效，以避免出现安全漏洞扫描不通过的问题。

5.4.3 TiDB 支持哪些认证协议？过程是怎样的？

TiDB 和 MySQL 一样，在用户登录认证时使用 SASL 认证协议对密码进行处理。

客户端连接TiDB 的时候，使用challenge-response（挑战-应答）的认证模式，过程如下：

1. 客户端连接服务器。
2. 服务器发送随机字符串 `challenge` 给客户端。
3. 客户端发送 `username` + `response` 给服务器。
4. 服务器验证 `response`。

5.4.4 如何修改用户名密码和权限？

因为TiDB 是分布式数据库，想要在TiDB 中修改用户密码，建议使用 `ALTER USER` 的方法，例如 `ALTER USER 'test'@'localhost' IDENTIFIED BY 'mypass';`。

不推荐使用 `UPDATE mysql.user` 的方法，因为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，建议参考TiDB 用户账户管理文档中的方法。

6 术语表

6.1 A

6.1.1 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。

6.2 B

6.2.1 BR

TiDB 备份恢复功能用户文档中的名词 BR 根据上下文不同有不同的解释，比较常见的指代用法：

- TiDB 备份恢复功能，包含br CLI、 TiDB Operator、 TiDB Cloud 提供的备份和恢复功能集合。
- 架构中的 BR 功能组件。

名词br 一般用来指代br CLI 工具。

6.2.2 Batch Create Table

批量建表(Batch Create Table)是在TiDB v6.0.0中引入的新功能，此功能默认开启。当需要恢复的数据中带有大量的表（约50000张）时，批量建表功能显著提升数据恢复的速度。详情参见批量建表。

6.2.3 Baseline Capturing

自动捕获绑定(Baseline Capturing)会对符合捕获条件的查询进行捕获，为符合条件的查询生成相应的绑定。通常用于升级时的计划回退防护。

6.2.4 Bucket

一个Region在逻辑上划分为多个小范围，称为bucket。TiKV按bucket收集查询统计数据，并将bucket的情况报告给PD。详情参见Bucket设计文档。

6.3 C

6.3.1 Cached Table

缓存表(Cached Table)是指TiDB把整张表的数据加载到服务器的内存中，直接从内存中获取表数据，避免从TiKV获取表数据，从而提升读性能。详情参见缓存表。

6.3.2 Coalesce Partition

Coalesce Partition是一种减少Hash分区表或Key分区表中分区数量的方法。详情参见管理Hash分区和Key分区。

6.3.3 Continuous Profiling

持续性能分析(Continuous Profiling)是从TiDB v5.3起引入的一种从系统调用层面解读资源开销的方法。引入该方法后，TiDB可提供数据库源码级性能观测，通过火焰图的形式帮助研发、运维人员定位性能问题的根因。详情参见TiDB Dashboard实例性能分析-持续分析页面。

6.4 D

6.4.1 Dynamic Pruning

动态裁剪(Dynamic Pruning)是TiDB访问分区表的两种模式之一。在动态裁剪模式下，TiDB的每个算子都支持直接访问多个分区，省略Union操作，提高执行效率，还避免了Union并发管理的问题。

6.5 I

6.5.1 Index Merge

索引合并(Index Merge)是在TiDB v4.0 版本中作为实验特性引入的一种查询执行方式的优化，可以大幅提高查询在扫描多列数据时条件过滤的效率。自 v5.4 版本起，Index Merge 成为正式功能，详情参见用 EXPLAIN 查看索引合并的 SQL 执行计划。

6.5.2 In-Memory Pessimistic Lock

内存悲观锁(In-Memory Pessimistic Lock) 是在TiDB v6.0.0 中引入的新功能。开启内存悲观锁功能后，悲观锁通常只会被存储在Region leader 的内存中，而不会将锁持久化到磁盘，也不会通过Raft 协议将锁同步到其他副本，因此可以大大降低悲观事务加锁的开销，提升悲观事务的吞吐并降低延迟。

6.6 L

6.6.1 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。

6.7 O

6.7.1 Old value

Old value 特指在TiCDC 输出的增量变更日志中的“原始值”。可以通过配置来指定TiCDC 输出的增量变更日志是否包含“原始值”。

6.7.2 Operator

Operator 是应用于一个Region 的，服务于某个调度目的的一系列操作的集合。例如“将 Region 2 的 Leader 迁移至 Store 5”，“将 Region 2 的副本迁移到 Store 1, 4, 5”等。

Operator 可以是由 Scheduler 通过计算生成的，也可以是由外部API 创建的。

6.7.3 Operator Step

Operator Step 是 Operator 执行过程的一个步骤，一个 Operator 常常会包含多个 Operator Step。

目前 PD 可生成的 Step 包括：

- `TransferLeader`：将 Region Leader 迁移至指定 Peer
- AddPeer：在指定 Store 添加 Follower
- RemovePeer：删除一个 Region Peer

- AddLearner：在指定 Store 添加 Region Learner
- PromoteLearner：将指定 Learner 提升为 Follower
- SplitRegion：将指定 Region 一分为二

6.8 P

6.8.1 Partitioning

Partitioning（分区）指通过 RANGE、LIST、HASH 和 KEY 等分区方法在物理上将一张表划分为较小的分区。

6.8.2 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。

6.8.3 Point get

点查(point get) 是指通过主键或唯一索引直接读取一行的查询方式。点查的返回结果最多是一行数据。

6.8.4 Predicate columns

执行 SQL 语句时，优化器在大多数情况下只会用到部分列（例如，WHERE、JOIN、ORDER BY、GROUP BY 子句中出现的列）的统计信息，这些用到的列称为 PREDICATE COLUMNS。详情参见收集部分列的统计信息。

6.9 Q

6.9.1 Quota Limiter

前台限流(Quota Limiter) 是在 TiDB v6.0.0 版本中作为实验特性引入的功能。当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果 TiKV 前台处理的读写请求量过大，会占用 TiKV 后台处理请求所需的 CPU 资源，最终影响 TiKV 性能的稳定性。此时，开启前台限流相关的quota 相关配置项可以限制前台各类请求占用的 CPU 资源。

6.10 R

6.10.1 Raft Engine

一种内置的持久化存储引擎，有着日志结构的设计，为 TiKV 提供 multi-Raft 日志存储。从 v5.4 起，TiDB 支持使用 Raft Engine 作为 TiKV 的日志存储引擎。详情参见Raft Engine。

6.10.2 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。

6.10.3 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。

6.10.4 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。

6.11 S

6.11.1 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）

6.11.2 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。

6.12 T

6.12.1 Top SQL

Top SQL 用于找到一段时间内对某个TiDB 或TiKV 节点消耗负载较大的 SQL 查询。详情参见Top SQL 用户文档。

6.12.2 TSO

因为TiKV是一个分布式的储存系统，它需要一个全球性的授时服务TSO (Timestamp Oracle)，来分配一个单调递增的时间戳。这样的功能在TiKV 中是由 PD 提供的，在Google 的 Spanner 中是由多个原子钟和GPS 来提供的。

6.12.3 TTL

Time to Live (TTL) 提供了行级别的生命周期控制策略。通过为表设置 TTL 属性，TiDB 可以周期性地自动检查并清理表中的过期数据。

© 2023 PingCAP 公司保留所有权利。除非版权法允许，否则在未得到本公司事先给出的书面许可的情况下，严禁复制、改编或翻译本文。