TiDB

TiDB 运维指南（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 17

1.1 TiDB 简介 ····· 17

1.1.1 五大核心特性 ····· 17

1.1.2 四大核心应用场景 ····· 17

1.1.3 另请参阅 ····· 18

1.2 TiDB 整体架构 ····· 18

2 部署标准集群 20

2.1 TiDB 软件和硬件环境建议配置 ····· 20

2.1.1 操作系统及平台要求 ····· 20

2.1.2 软件配置要求 ····· 21

2.1.3 服务器建议配置 ····· 22

2.1.4 网络要求 ····· 24

2.1.5 磁盘空间要求 ····· 24

2.1.6 客户端 Web 浏览器要求 ····· 25

2.2 TiDB 环境与系统配置检查 ····· 25

2.2.1 在 TiKV 部署目标机器上添加数据盘 EXT4 文件系统挂载参数 ····· 25

2.2.2 检测及关闭系统 swap ····· 26

2.2.3 检测及关闭目标部署机器的防火墙 ····· 27

2.2.4 检测及安装 NTP 服务 ····· 27

2.2.5 检查和配置操作系统优化参数 ····· 29

2.2.6 手动配置 SSH 互信及 sudo 免密码 ····· 34

2.2.7 安装 numactl 工具 ····· 35

2.3 规划集群拓扑 ····· 36
2.3.1 最小拓扑架构 ····· 36
2.3.2 TiFlash 部署拓扑 ····· 38
2.3.3 TiCDC 部署拓扑 ····· 39
2.3.4 TiDB Binlog 部署拓扑 ····· 40
2.3.5 TiSpark 部署拓扑 ····· 43
2.3.6 跨数据中心部署拓扑 ····· 46
2.3.7 混合部署拓扑 ····· 49
2.4 安装与启动 ····· 52
2.4.1 使用 TiUP 部署 TiDB 集群 ····· 52
2.4.2 在 Kubernetes 上部署 TiDB ····· 66
2.5 验证集群运行状态 ····· 67
2.5.1 通过 TiUP 检查集群状态 ····· 67
2.5.2 通过 TiDB Dashboard 和 Grafana 检查集群状态 ····· 67
2.5.3 登录数据库执行简单 DML/DDL 操作和查询 SQL 语句 ····· 68
2.6 测试集群性能 ····· 71
2.6.1 如何用 Sysbench 测试 TiDB ····· 71
2.6.2 如何对 TiDB 进行 TPC-C 测试 ····· 75
2.6.3 如何对 TiDB 进行 CH-benCHmark 测试 ····· 77

3 数据迁移 81
3.1 数据迁移概述 ····· 81
3.1.1 迁移 Aurora MySQL 到 TiDB ····· 81
3.1.2 迁移 MySQL 到 TiDB ····· 82
3.1.3 分库分表 MySQL 合并迁移到 TiDB ····· 82
3.1.4 从文件迁移数据到 TiDB ····· 82
3.1.5 TiDB 集群增量数据同步 ····· 82
3.1.6 复杂迁移场景 ····· 82
3.2 数据迁移工具概览 ····· 83
3.2.1 TiDB Data Migration (DM) ····· 83
3.2.2 TiDB Lightning ····· 83
3.2.3 Dumpling ····· 84
3.2.4 TiCDC ····· 84

3.2.5 Backup & Restore (BR) ····· 84
3.2.6 sync-diff-inspector ····· 85
3.2.7 使用 TiUP 快速安装 ····· 85
3.2.8 探索更多 ····· 86
3.3 数据迁移场景 ····· 87
3.3.1 从 Amazon Aurora 迁移数据到 TiDB ····· 87
3.3.2 从小数据量 MySQL 迁移数据到 TiDB ····· 93
3.3.3 从大数据量 MySQL 迁移数据到 TiDB ····· 97
3.3.4 从小数据量分库分表 MySQL 合并迁移数据到 TiDB ····· 104
3.3.5 从大数据量分库分表 MySQL 合并迁移数据到 TiDB ····· 109
3.3.6 从 CSV 文件迁移数据到 TiDB ····· 117
3.3.7 从 SQL 文件迁移数据到 TiDB ····· 122
3.3.8 从 Parquet 文件迁移数据到 TiDB ····· 124
3.3.9 从 TiDB 集群迁移数据至另一 TiDB 集群 ····· 126
3.3.10 从 TiDB 集群迁移数据至兼容 MySQL 的数据库 ····· 132
3.4 复杂迁移场景 ····· 137
3.4.1 上游使用 pt-osc/gh-ost 工具的持续同步场景 ····· 137
3.4.2 下游存在更多列的迁移场景 ····· 138
3.4.3 如何过滤 binlog 事件 ····· 142
3.4.4 如何通过 SQL 表达式过滤 DML ····· 145

4 数据集成 147
4.1 数据集成概述 ····· 147
4.1.1 与 Confluent Cloud 和 Snowflake 进行数据集成 ····· 147
4.1.2 与 Apache Kafka 和 Apache Flink 进行数据集成 ····· 147
4.2 数据集成场景 ····· 147
4.2.1 与 Confluent Cloud 和 Snowflake 进行数据集成 ····· 147
4.2.2 与 Apache Kafka 和 Apache Flink 进行数据集成 ····· 160

5 运维操作 164
5.1 升级 TiDB 版本 ····· 164
5.1.1 使用 TiUP 升级 TiDB ····· 164
5.1.2 使用 TiDB Operator ····· 170
5.1.3 平滑升级 TiDB ····· 170
5.1.4 TiFlash v6.2 升级帮助 ····· 171

5.2 扩缩容 ····· 173
5.2.1 使用 TiUP 扩容缩容 TiDB 集群 ····· 173
5.2.2 使用 TiDB Operator ····· 183
5.3 备份与恢复 ····· 183
5.3.1 TiDB 备份与恢复概述 ····· 183
5.3.2 架构设计 ····· 191
5.3.3 使用 BR 进行备份与恢复 ····· 203
5.3.4 br cli 命令手册 ····· 222
5.3.5 参考指南 ····· 237
5.4 集群容灾 ····· 247
5.4.1 TiDB 容灾方案概述 ····· 247
5.4.2 基于主备集群的容灾方案 ····· 258
5.4.3 基于多副本的单集群容灾方案 ····· 268
5.4.4 基于备份与恢复的容灾方案 ····· 272
5.5 使用资源管控 (Resource Control) 实现资源隔离 ····· 274
5.5.1 使用场景 ····· 274
5.5.2 使用限制 ····· 274
5.5.3 什么是 Request Unit (RU) ····· 275
5.5.4 相关参数 ····· 275
5.5.5 使用方法 ····· 275
5.5.6 关闭资源管控特性 ····· 277
5.5.7 监控与图表 ····· 278
5.5.8 工具兼容性 ····· 278
5.5.9 常见问题 ····· 278
5.5.10 另请参阅 ····· 278
5.6 时区支持 ····· 278
5.7 日常巡检 ····· 280
5.7.1 TiDB Dashboard 关键指标 ····· 280
5.8 TiFlash 集群运维 ····· 285
5.8.1 查看 TiFlash 版本 ····· 286
5.8.2 TiFlash 重要日志介绍 ····· 286
5.8.3 TiFlash 系统表 ····· 287
5.9 TiUP 常见运维操作 ····· 287

5.9.1 查看集群列表 287
5.9.2 启动集群 288
5.9.3 查看集群状态 288
5.9.4 修改配置参数 288
5.9.5 Hotfix 版本替换 289
5.9.6 重命名集群 290
5.9.7 关闭集群 290
5.9.8 清除集群数据 291
5.9.9 销毁集群 291
5.10 在线修改集群配置 292
5.10.1 常用操作 292
5.11 Online Unsafe Recovery 使用文档 326
5.11.1 功能说明 327
5.11.2 适用场景 327
5.11.3 使用步骤 327
5.12 搭建双集群主从复制 331
5.12.1 第 1 步：搭建环境 332
5.12.2 第 2 步：迁移全量数据 333
5.12.3 第 3 步：迁移增量数据 336
5.12.4 第 4 步：模拟主集群故障 337
5.12.5 第 5 步：使用 redo log 确保数据一致性 337
5.12.6 第 6 步：恢复主集群及业务 337

6 监控与告警 338
6.1 TiDB 监控框架概述 338
6.1.1 Prometheus 在 TiDB 中的应用 338
6.1.2 Grafana 在 TiDB 中的应用 339
6.2 TiDB 集群监控 API 341
6.2.1 使用状态接口 341
6.2.2 使用 metrics 接口 344

6.3 TiDB 集群监控部署 344
6.3.1 部署 Prometheus 和 Grafana 344
6.3.2 配置 Grafana 347
6.3.3 查看组件 metrics 348
6.4 将 Grafana 监控数据导出成快照 350

6.4.1 使用方法 350
6.4.2 FAQs 351
6.5 TiDB 集群报警规则 352
6.5.1 TiDB 报警规则 352
6.5.2 PD 报警规则 355
6.5.3 TiKV 报警规则 360
6.5.4 TiFlash 报警规则 368
6.5.5 TiDB Binlog 报警规则 368
6.5.6 TiCDC 报警规则 368
6.5.7 Node_exporter 主机报警规则 368
6.5.8 Blackbox_exporter TCP、ICMP 和 HTTP 报警规则 371
6.6 TiFlash 报警规则 375
6.6.1 TiFlash_schema_error 375
6.6.2 TiFlash_schema_apply_duration 376
6.6.3 TiFlash_raft_read_index_duration 376
6.6.4 TiFlash_raft_wait_index_duration 376
6.7 自定义监控组件的配置 377
6.7.1 自定义 Prometheus 配置 377
6.7.2 自定义 Grafana 配置 378
6.7.3 自定义 Alertmanager 配置 379
6.8 备份恢复监控告警 380
6.8.1 日志备份监控 380

7 故障诊断 383
7.1 故障诊断问题汇总 383
7.1.1 TiDB 集群问题导图 383
7.1.2 TiDB 集群故障诊断 398
7.1.3 TiFlash 常见问题 401

7.2 故障场景 405
7.2.1 慢查询 405
7.2.2 TiDB OOM 故障排查 425
7.2.3 TiDB 热点问题处理 430
7.2.4 读写延迟增加 440
7.2.5 乐观事务模型下写写冲突问题排查 444
7.2.6 TiDB 磁盘 I/O 过高的处理办法 447

7.2.7 TiDB 锁冲突问题处理 ····· 449
7.2.8 数据索引一致性报错 ····· 459
7.3 故障诊断方法 ····· 461
7.3.1 SQL 诊断 ····· 461
7.3.2 Statement Summary Tables ····· 463
7.3.3 TiDB Dashboard Top SQL 页面 ····· 471
7.3.4 定位消耗系统资源多的查询 ····· 478
7.3.5 使用 PLAN REPLAYER 保存和恢复集群现场信息 ····· 480
7.4 支持资源 ····· 485
8 性能调优 486
8.1 优化手册 ····· 486
8.1.1 TiDB 性能优化概述 ····· 486
8.1.2 TiDB 性能分析和优化 ····· 490
8.1.3 OLTP 负载性能优化实践 ····· 508
8.1.4 TiFlash 性能分析和优化方法 ····· 529
8.1.5 TiCDC 性能分析和优化方法 ····· 534
8.1.6 延迟的拆解分析 ····· 536
8.2 配置调优 ····· 554
8.3 SQL 性能调优 ····· 607
8.3.1 SQL 性能调优 ····· 607
8.3.2 理解 TiDB 执行计划 ····· 607
8.3.3 SQL 优化流程 ····· 679
8.3.4 控制执行计划 ····· 765
9 教程 804
9.1 单区域多 AZ 部署 TiDB ····· 804
9.1.1 了解 Raft 协议 ····· 804
9.1.2 同区域三 AZ 方案 ····· 804
9.2 双区域多 AZ 部署 TiDB ····· 810
9.2.1 简介 ····· 810
9.2.2 架构 ····· 810
9.2.3 配置 ····· 812
9.3 单区域双 AZ 部署 TiDB ····· 816
9.3.1 简介 ····· 816

9.3.2 部署架构 816
9.3.3 配置 817
9.4 读取历史数据 823
9.4.1 使用 Stale Read 功能读取历史数据（推荐） 823
9.4.2 通过系统变量 tidb_snapshot 读取历史数据 834

10 TiDB 工具 839
10.1 TiDB 工具功能概览 839
10.1.1 部署运维工具 839
10.1.2 数据管理工具 840
10.1.3 OLAP 分析工具 - TiSpark 842
10.2 TiDB 工具的使用场景 842
10.2.1 在物理机或虚拟机上部署运维 TiDB 843
10.2.2 在 Kubernetes 上部署运维 TiDB 843
10.2.3 从 CSV 导入数据到 TiDB 843
10.2.4 从 MySQL/Aurora 导入全量数据 843
10.2.5 从 MySQL/Aurora 迁移数据 843
10.2.6 TiDB 集群备份与恢复 843
10.2.7 迁出数据到 TiDB 843
10.2.8 TiDB 增量数据订阅 843
10.3 TiDB 工具下载 843
10.3.1 TiDB 工具包下载 844
10.3.2 TiUniManager 下载 847

10.4 TiUP 848
10.4.1 TiUP 文档地图 848
10.4.2 TiUP 简介 849
10.4.3 TiUP 术语及核心概念 852
10.4.4 使用 TiUP 命令管理组件 852
10.4.5 TiUP FAQ 856
10.4.6 TiUP 故障排查 857
10.4.7 TiUP 命令参考手册 858
10.4.8 通过 TiUP 部署 TiDB 集群的拓扑文件配置 935
10.4.9 通过 TiUP 部署 DM 集群的拓扑文件配置 950
10.4.10 TiUP 镜像参考指南 958
10.4.11 TiUP 组件文档 965

10.5 TiDB Operator ····· 992
10.6 TiDB Data Migration ····· 992
10.6.1 TiDB Data Migration 简介 ····· 992
10.6.2 Data Migration 架构 ····· 994
10.6.3 TiDB Data Migration 快速上手指南 ····· 996
10.6.4 DM 数据迁移最佳实践 ····· 999
10.6.5 部署 DM 集群 ····· 1011
10.6.6 入门指南 ····· 1026
10.6.7 进阶教程 ····· 1069
10.6.8 运维管理 ····· 1110
10.6.9 参考手册 ····· 1159
10.7 TiDB Lightning ····· 1277
10.7.1 TiDB Lightning 简介 ····· 1277
10.7.2 TiDB Lightning 快速上手 ····· 1279
10.7.3 部署 TiDB Lightning ····· 1281
10.7.4 TiDB Lightning 目标数据库要求 ····· 1282
10.7.5 数据源 ····· 1284
10.7.6 Physical Import Mode ····· 1294
10.7.7 Logical Import Mode ····· 1302
10.7.8 TiDB Lightning 前置检查 ····· 1304
10.7.9 表库过滤 ····· 1311
10.7.10 TiDB Lightning 断点续传 ····· 1316
10.7.11 TiDB Lightning 并行导入 ····· 1318
10.7.12 TiDB Lightning 错误处理功能 ····· 1323
10.7.13 TiDB Lightning 故障处理 ····· 1330
10.8 使用 Dumpling 导出数据 ····· 1377
10.8.1 从 TiDB/MySQL 导出数据 ····· 1378
10.8.2 Dumpling 主要选项表 ····· 1384
10.9 TiCDC ····· 1397
10.9.1 TiCDC 简介 ····· 1397
10.9.2 TiCDC 安装部署与集群运维 ····· 1400
10.9.3 Changefeed ····· 1404
10.9.4 监控告警 ····· 1438
10.9.5 TiCDC 故障处理 ····· 1536

10.9.6 TiCDC 常见问题解答 ····· 1540
10.9.7 TiCDC 术语表 ····· 1550
10.10 TiDB Binlog ····· 1551
10.10.1 TiDB Binlog 简介 ····· 1551
10.10.2 TiDB Binlog 教程 ····· 1553
10.10.3 TiDB Binlog 集群部署 ····· 1562
10.10.4 TiDB Binlog 集群运维 ····· 1570
10.10.5 TiDB Binlog 配置说明 ····· 1572
10.10.6 TiDB Binlog 版本升级方法 ····· 1588
10.10.7 TiDB Binlog 集群监控 ····· 1590
10.10.8 Reparo 使用文档 ····· 1594
10.10.9 binlogctl 工具 ····· 1596
10.10.10Binlog Consumer Client 用户文档 ····· 1599
10.10.11TiDB Binlog Relay Log ····· 1602
10.10.12集群间双向同步 ····· 1603
10.10.13TiDB Binlog 术语表 ····· 1607
10.10.14故障诊断 ····· 1608
10.10.15TiDB Binlog 常见问题 ····· 1609
10.11 PingCAP Clinic 诊断服务 ····· 1615
10.11.1 PingCAP Clinic 诊断服务简介 ····· 1615
10.11.2 PingCAP Clinic 快速上手指南 ····· 1617
10.11.3 使用 PingCAP Clinic 诊断集群 ····· 1620
10.11.4 使用 PingCAP Clinic 生成诊断报告 ····· 1628
10.11.5 使用 PingCAP Clinic Diag 采集 SQL 查询计划信息 ····· 1632
10.11.6 PingCAP Clinic 数据采集说明 ····· 1636
10.12 TiSpark ····· 1641
10.12.1 TiSpark 用户指南 ····· 1641
10.13 sync-diff-inspector ····· 1655
10.13.1 sync-diff-inspector 用户文档 ····· 1655
10.13.2 不同库名或表名的数据校验 ····· 1662
10.13.3 分库分表场景下的数据校验 ····· 1663
10.13.4 TiDB 主从集群的数据校验 ····· 1667
10.13.5 基于 DM 同步场景下的数据校验 ····· 1669
10.14 TiUniManager ····· 1669
10.14.1 TiUniManager 概览 ····· 1669

10.14.2 TiUniManager 安装和运维指南 ····· 1673
10.14.3 TiUniManager 快速操作指南 ····· 1687
10.14.4 操作指南 ····· 1691
10.14.5 TiUniManager 常见问题 ····· 1712
10.14.6 发布版本历史 ····· 1716

11 TiDB Dashboard 1721
11.1 TiDB Dashboard 介绍 ····· 1721
11.1.1 了解集群整体运行概况 ····· 1722
11.1.2 查看组件及主机运行状态 ····· 1722
11.1.3 分析集群读写流量分布及趋势变化 ····· 1722
11.1.4 列出所有 SQL 查询的耗时等执行信息 ····· 1723
11.1.5 详细了解耗时较长的 SQL 语句的执行信息 ····· 1723
11.1.6 诊断常见集群问题并生成报告 ····· 1723
11.1.7 查询所有组件日志 ····· 1723
11.1.8 预估资源管控容量 ····· 1723
11.1.9 收集分析各个组件的性能数据 ····· 1723

11.2 运维 ····· 1723
11.2.1 部署 TiDB Dashboard ····· 1723
11.2.2 通过反向代理使用 TiDB Dashboard ····· 1727
11.2.3 TiDB Dashboard 用户管理 ····· 1731
11.2.4 提高 TiDB Dashboard 安全性 ····· 1735
11.3 访问 TiDB Dashboard ····· 1737
11.3.1 多 PD 实例访问 ····· 1737
11.3.2 浏览器兼容性 ····· 1738
11.3.3 登录 ····· 1738
11.3.4 切换语言 ····· 1740
11.3.5 登出 ····· 1741
11.4 TiDB Dashboard 概况页面 ····· 1742
11.4.1 访问 ····· 1743
11.4.2 QPS ····· 1743
11.4.3 延迟 ····· 1744
11.4.4 Top SQL 语句 ····· 1745
11.4.5 最近的慢查询 ····· 1746
11.4.6 实例 ····· 1747

11.4.7 监控和告警 1747
11.5 TiDB Dashboard 集群信息页面 1748
11.5.1 访问 1748
11.5.2 实例列表 1749
11.5.3 主机列表 1750
11.5.4 磁盘列表 1750
11.6 TiDB Dashboard Top SQL 页面 1751
11.6.1 推荐适用场景 1751
11.6.2 访问页面 1752
11.6.3 启用 Top SQL 1752
11.6.4 使用 Top SQL 1753
11.6.5 停用 Top SQL 1757
11.6.6 常见问题 1757
11.7 TiDB Dashboard 流量可视化页面 1757
11.7.1 访问页面 1758
11.7.2 界面示例 1759
11.7.3 基本概念 1759
11.7.4 使用介绍 1760
11.7.5 常见热力图解读 1766
11.7.6 解决热点问题 1769
11.8 TiDB Dashboard 监控关系图 1770
11.8.1 访问关系图 1770
11.8.2 关系图解读 1770
11.9 SQL 语句分析 1775
11.9.1 TiDB Dashboard SQL 语句分析执行详情页面 1775
11.9.2 TiDB Dashboard SQL 语句分析执行详情页面 1780
11.10 TiDB Dashboard 慢查询页面 1792
11.10.1 访问列表页面 1792
11.10.2 查看执行详情 1793
11.11 集群诊断页面 1797
11.11.1 TiDB Dashboard 集群诊断页面 1797
11.11.2 TiDB Dashboard 诊断报告 1800
11.11.3 使用 TiDB Dashboard 诊断报告定位问题 1815
11.12 TiDB Dashboard 监控页面 1819

11.12.1 访问页面 1819
11.12.2 面板监控指标详解 1821
11.13 TiDB Dashboard 日志搜索页面 1824
11.13.1 访问 1824
11.13.2 搜索结果详情 1824
11.13.3 搜索历史列表 1826
11.14 TiDB Dashboard 资源管控页面 1827
11.14.1 访问方式 1827
11.14.2 资源管控详情 1828
11.14.3 容量估算 1828
11.14.4 监控指标 1830
11.15 实例性能分析 1830
11.15.1 TiDB Dashboard 实例性能分析 - 手动分析页面 1830
11.15.2 TiDB Dashboard 实例性能分析 - 持续分析页面 1836
11.16 会话管理与配置 1841
11.16.1 分享 TiDB Dashboard 会话 1841
11.16.2 配置 TiDB Dashboard 使用 SSO 登录 1845
11.17 TiDB Dashboard 常见问题 1865
11.17.1 访问 1865
11.17.2 界面功能 1865
12 常见问题解答 (FAQ) 1869
12.1 TiDB 安装部署常见问题 1869
12.1.1 软硬件要求 FAQ 1869
12.1.2 安装部署 FAQ 1870
12.1.3 TiDB 支持在公有云上部署吗？ 1875
12.2 迁移常见问题 1876
12.2.1 全量数据导出导入 1876
12.2.2 在线数据同步 1878
12.2.3 业务流量迁入 1878
12.3 升级与升级后常见问题 1880
12.3.1 升级常见问题 1880
12.3.2 升级后常见问题 1880
12.4 TiDB 监控常见问题 1885
12.4.1 目前的监控使用方式及主要监控指标，有没有更好看的监控？ 1885

12.4.2 Prometheus 监控数据默认 15 天自动清除一次，可以自己设定成 2 个月或者手动删除吗？ ····· 1885
12.4.3 Region Health 监控项 ····· 1886
12.4.4 Statement Count 监控项中的 selectsimplefull 是什么意思？ ····· 1886
12.4.5 监控上的 QPS 和 Statement OPS 有什么区别？ ····· 1886
12.5 TiDB 集群管理常见问题 ····· 1886
12.5.1 集群日常管理 ····· 1886
12.5.2 PD 管理 ····· 1889
12.5.3 TiDB server 管理 ····· 1890
12.5.4 TiKV 管理 ····· 1893
12.5.5 TiDB 测试 ····· 1896
12.5.6 TiDB 备份恢复 ····· 1896
12.6 备份与恢复常见问题 ····· 1897
12.6.1 当误删除或误更新数据后，如何原地快速恢复？ ····· 1897
12.6.2 在 TiDB v5.4.0 及后续版本中，当在有负载的集群进行备份时，备份速度为什么会变得很慢？ · 1897
12.6.3 PITR 问题 ····· 1897
12.6.4 功能兼容性问题 ····· 1900
12.6.5 进行数据恢复的问题 ····· 1901
12.6.6 备份恢复功能相关知识 ····· 1904
13 术语表 1905
13.1 A ····· 1905
13.1.1 ACID ····· 1905
13.2 B ····· 1905
13.2.1 BR ····· 1905
13.2.2 Batch Create Table ····· 1905
13.2.3 Baseline Capturing ····· 1905
13.2.4 Bucket ····· 1905
13.3 C ····· 1906
13.3.1 Cached Table ····· 1906
13.3.2 Coalesce Partition ····· 1906
13.3.3 Continuous Profiling ····· 1906
13.4 D ····· 1906
13.4.1 Dynamic Pruning ····· 1906
13.5 I ····· 1906
13.5.1 Index Merge ····· 1906

13.5.2 In-Memory Pessimistic Lock ····· 1906
13.6 L ····· 1906
13.6.1 Leader/Follower/Learner ····· 1906
13.7 O ····· 1907
13.7.1 Old value ····· 1907
13.7.2 Operator ····· 1907
13.7.3 Operator Step ····· 1907
13.8 P ····· 1907
13.8.1 Partitioning ····· 1907
13.8.2 Pending/Down ····· 1907
13.8.3 Point get ····· 1907
13.8.4 Predicate columns ····· 1908
13.9 Q ····· 1908
13.9.1 Quota Limiter ····· 1908
13.10 R ····· 1908
13.10.1 Raft Engine ····· 1908
13.10.2 Region/Peer/Raft Group ····· 1908
13.10.3 Region Split ····· 1908
13.10.4 Restore ····· 1908
13.11 S ····· 1908
13.11.1 Scheduler ····· 1908
13.11.2 Store ····· 1909
13.12 T ····· 1909
13.12.1 Top SQL ····· 1909
13.12.2 TSO ····· 1909
13.12.3 TTL ····· 1909

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 整体架构

推荐先观看以下视频（时长约 14 分钟），快速了解 TiDB 的整体架构。

与传统的单机数据库相比，TiDB 具有以下优势：

- 纯分布式架构，拥有良好的扩展性，支持弹性的扩缩容
- 支持 SQL，对外暴露 MySQL 的网络协议，并兼容大多数 MySQL 的语法，在大多数场景下可以直接替换 MySQL
- 默认支持高可用，在少数副本失效的情况下，数据库本身能够自动进行数据修复和故障转移，对业务透明
- 支持ACID 事务，对于一些有强一致需求的场景友好，例如：银行转账
- 具有丰富的工具链生态，覆盖数据迁移、同步、备份等多种场景

在内核设计上，TiDB 分布式数据库将整体架构拆分成了多个模块，各模块之间互相通信，组成完整的TiDB 系统。对应的架构图如下：

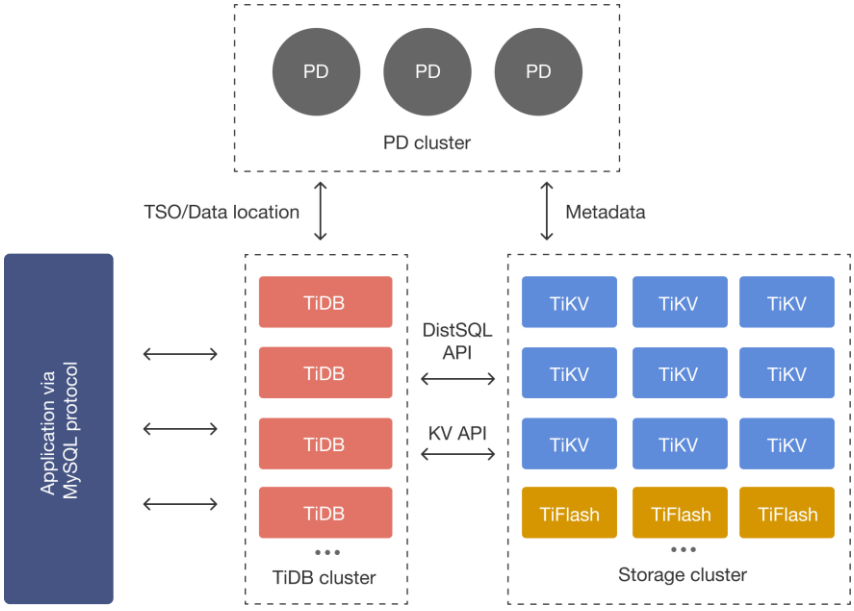


图 1: architecture

- TiDB Server：SQL 层，对外暴露 MySQL 协议的连接 endpoint，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划。TiDB 层本身是无状态的，实践中可以启动多个TiDB 实例，通过负载均衡组件（如 LVS、HAProxy 或 F5）对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个 TiDB 实例上以达到负载均衡的效果。TiDB Server 本身并不存储数据，只是解析 SQL，将实际的数据读取请求转发给底层的存储节点TiKV（或 TiFlash）。
- PD (Placement Driver) Server：整个 TiDB 集群的元信息管理模块，负责存储每个TiKV 节点实时的数据分布情况和集群的整体拓扑结构，提供TiDB Dashboard 管控界面，并为分布式事务分配事务ID。PD 不仅存储元信息，同时还会根据TiKV 节点实时上报的数据分布状态，下发数据调度命令给具体的TiKV 节点，可以说是整个集群的“大脑”。此外，PD 本身也是由至少 3 个节点构成，拥有高可用的能力。建议部署奇数个 PD 节点。
- 存储节点
  - TiKV Server：负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个TiKV 节点会负责多个Region。TiKV 的 API 在 KV 键值对层面提供对分布式事务的原生支持，默认提供了 SI (Snapshot Isolation) 的隔离级别，这也是TiDB 在 SQL 层面支持分布式事务的核心。TiDB 的 SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为对TiKV API 的实际调用。所以，数据都存储在TiKV 中。另外，TiKV 中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。
  - TiFlash：TiFlash 是一类特殊的存储节点。和普通TiKV 节点不一样的是，在TiFlash 内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。

2 部署标准集群

2.1 TiDB 软件和硬件环境建议配置

TiDB 作为一款开源一栈式实时 HTAP 数据库，可以很好地部署和运行在Intel 架构服务器环境、ARM 架构的服务器环境及主流虚拟化环境，并支持绝大多数的主流硬件网络。作为一款高性能数据库系统，TiDB 支持主流的 Linux 操作系统环境。

2.1.1 操作系统及平台要求

操作系统	支持的 CPU 架构
Red Hat Enterprise Linux 8.4 及以上的 8.x 版本	x86_64 ARM 64
Red Hat Enterprise Linux 7.3 及以上的 7.x 版本 CentOS 7.3 及以上的 7.x 版本	x86_64 ARM 64
Amazon Linux 2	x86_64 ARM 64
麒麟欧拉版V10 SP1/SP2	x86_64 ARM 64
UOS V20	x86_64 ARM 64
openEuler 22.03 LTS SP1	x86_64
macOS Catalina 及以上的版本	x86_64 ARM 64

| | Oracle Enterprise Linux 7.3 及以上的 7.x 版本 | x86_64 | | Ubuntu LTS 18.04 及以上的版本 | x86_64 | | CentOS 8 Stream |

x86_64

ARM 64

| | Debian 9 (Stretch) 及以上的版本 | x86_64 | | Fedora 35 及以上的版本 | x86_64 | | openSUSE Leap 15.3 以上的版本（不包含 Tumbleweed） | x86_64 | | SUSE Linux Enterprise Server 15 | x86_64 |

注意：

- TiDB 只支持 Red Hat 兼容内核 (RHCK) 的 Oracle Enterprise Linux，不支持 Oracle Enterprise Linux 提供的 Unbreakable Enterprise Kernel。
- 根据 CentOS Linux EOL，CentOS Linux 8 的上游支持已于 2021 年 12 月 31 日终止，但 CentOS 将继续提供对 CentOS Stream 8 的支持。
- TiDB 将不再支持 Ubuntu 16.04。强烈建议升级到 Ubuntu 18.04 或更高版本。
- 对于以上表格中所列操作系统的 32 位版本，TiDB 在这些 32 位操作系统以及对应的 CPU 架构上不保障可编译、可构建以及可部署，或 TiDB 不主动适配这些 32 位的操作系统。
- 以上未提及的操作系统版本也许可以运行 TiDB，但尚未得到 TiDB 官方支持。

2.1.1.1 编译和运行 TiDB 所依赖的库

编译和构建 TiDB 所需的依赖库	版本
Golang	1.20 及以上版本
Rust	nightly-2022-07-31 及以上版本
GCC	7.x
LLVM	13.0 及以上版本

运行时所需的依赖库：glibc（2.28-151.el8 版本）

2.1.1.2 Docker 镜像依赖

支持的 CPU 架构如下：

- x86_64，从 TiDB v6.6.0 开始，需要 x84-64-v2 指令集
- ARM 64

2.1.2 软件配置要求

2.1.2.1 中控机软件配置

软件	版本
sshpass	1.06 及以上
TiUP	1.5.0 及以上

注意：

中控机需要部署TiUP 软件来完成TiDB 集群运维管理。

2.1.2.2 目标主机建议配置软件

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

2.1.3 服务器建议配置

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

2.1.3.1 开发及测试环境

组件	CPU	内存	本地存储	网络	实例数量(最低要求)
TiDB	8 核 +	16 GB+	无特殊要求	千兆网卡	1（可与 PD 同机器）
PD	4 核 +	8 GB+	SAS, 200 GB+	千兆网卡	1（可与TiDB 同机器）
TiKV	8 核 +	32 GB+	SSD, 200 GB+	千兆网卡	3
TiFlash	32 核 +	64 GB+	SSD, 200 GB+	千兆网卡	1
TiCDC	8 核 +	16 GB+	SAS, 200 GB+	千兆网卡	1

注意：

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

- 从 v6.3.0 开始，在 Linux AMD64 架构的硬件平台部署TiFlash 时，CPU 必须支持AVX2 指令集。确保命令 `cat /proc/cpuinfo | grep avx2` 有输出。而在 Linux ARM64 架构的硬件平台部署 TiFlash 时，CPU 必须支持ARMv8 架构。确保命令 `cat /proc/cpuinfo | grep 'crc32' | grep 'asimd'` 有输出。通过使用向量扩展指令集，TiFlash 的向量化引擎能提供更好的性能。

2.1.3.2 生产环境

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

注意：

- 生产环境中的TiDB 和 PD 可以部署和运行在同一台服务器上，如对性能和可靠性有更高的要求，应尽可能分开部署。
- 生产环境强烈推荐使用更高的配置。
- TiKV 硬盘大小配置建议PCI-E SSD 不超过 2 TB，普通 SSD 不超过 1.5 TB。
- TiFlash 支持多盘部署。
- TiFlash 数据目录的第一块磁盘推荐用高性能 SSD 来缓冲TiKV 同步数据的实时写入，该盘性能应不低于TiKV 所使用的磁盘，比如PCI-E SSD。并且该磁盘容量建议不小于总容量的 10%，否则它可能成为这个节点的能承载的数据量的瓶颈。而其他磁盘可以根据需求部署多块普通 SSD，当然更好的 PCI-E SSD 硬盘会带来更好的性能。
- TiFlash 推荐与TiKV 部署在不同节点，如果条件所限必须将TiFlash 与 TiKV 部署在相同节点，则需要适当增加 CPU 核数和内存，且尽量将TiFlash 与 TiKV 部署在不同的磁盘，以免互相干扰。
- TiFlash 硬盘总容量大致为：整个 TiKV 集群的需同步数据容量 / TiKV 副本数 * TiFlash 副本数。例如整体TiKV 的规划容量为 1 TB、TiKV 副本数为 3、TiFlash 副本数为 2，则 TiFlash 的推荐总容量为 1024 GB / 3 * 2。用户可以选择同步部分表数据而非全部，具体容量可以根据需要同步的表的数据量具体分析。
- TiCDC 硬盘配置建议 1 TB+ PCIE-SSD。

2.1.4 网络要求

TiDB 作为开源一栈式实时 HTAP 数据库，其正常运行需要网络环境提供如下的网络端口配置要求，管理员可根据实际环境中TiDB 组件部署的方案，在网络侧和主机侧开放相关端口：

组件	默认端口	说明
TiDB	4000	应用及 DBA 工具访问通信端口
TiDB	10080	TiDB 状态信息上报通信端口
TiKV	20160	TiKV 通信端口
TiKV	20180	TiKV 状态信息上报通信端口
PD	2379	提供TiDB 和 PD 通信端口
PD	2380	PD 集群节点间通信端口
TiFlash	9000	TiFlash TCP 服务端口
TiFlash	3930	TiFlash RAFT 服务和 Coprocessor 服务端口
TiFlash	20170	TiFlash Proxy 服务端口
TiFlash	20292	Prometheus 拉取TiFlash Proxy metrics 端口
TiFlash	8234	Prometheus 拉取TiFlash metrics 端口
Pump	8250	Pump 通信端口
Drainer	8249	Drainer 通信端口
CDC	8300	CDC 通信接口
Monitoring	9090	Prometheus 服务通信端口
Monitoring	12020	NgMonitoring 服务通信端口
Node_exporter	9100	TiDB 集群每个节点的系统信息上报通信端口
Blackbox_exporter	9115	Blackbox_exporter 通信端口，用于TiDB 集群端口监控
Grafana	3000	Web 监控服务对外服务和客户端(浏览器) 访问端口
Alertmanager	9093	告警 web 服务端口
Alertmanager	9094	告警通信端口

2.1.5 磁盘空间要求

组件	磁盘空间要求	健康水位使用率
TiDB	日志盘建议最少预留 30 GB	低于 90%
PD	数据盘和日志盘建议最少各预留 20 GB	低于 90%
TiKV	数据盘和日志盘建议最少各预留 100 GB	低于 80%
TiFlash	数据盘建议最少预留 100 GB，日志盘建议最少预留 30 GB	低于 80%

| TiUP |

中控机：部署一个版本的TiDB 集群占用不超过 1 GB 空间，部署多个版本集群所占用的空间会相应增加

部署服务器（实际运行TiDB 各组件的机器）：TiFlash 占用约 700 MB 空间，其他组件（PD、TiDB、TiKV 等）各占用约 200 MB 空间。同时，部署过程会占用小于 1 MB 临时空间（/tmp）存放临时文件

| 不涉及| | Ngmonitoring |

Conprof：3 x 1 GB x 组件数量（表示每个组件每天占用约 1 GB，总共 3 天）+ 20 GB 预留空间

Top SQL：30 x 50 MB x 组件数量（每个组件每天占用约 50 MB，总共 30 天）

Top SQL 和 Conprof 共享预留空间

| 不涉及 |

2.1.6 客户端Web 浏览器要求

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

2.2 TiDB 环境与系统配置检查

本文介绍部署TiDB 前的环境检查操作，以下各项操作按优先级排序。

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

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

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

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

以 `/dev/nvme0n1` 数据盘为例，具体操作步骤如下：

1. 查看数据盘。

```
fdisk -l
```

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

2. 创建分区。

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

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

3. 格式化文件系统。

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

4. 查看数据盘分区 UUID。

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

```
lsblk -f
```

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

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

```
vi /etc/fstab
```

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

6. 挂载数据盘。

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

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

```
mount -t ext4
```

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

2.2.2 检测及关闭系统 swap

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

要永久关闭 swap，可执行以如下命令:

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

注意：

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

2.2.3 检测及关闭目标部署机器的防火墙

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

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

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

2. 关闭防火墙服务

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

3. 关闭防火墙自动启动服务

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

4. 检查防火墙状态

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

2.2.4 检测及安装 NTP 服务

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

采用如下步骤检查是否安装 NTP 服务以及与 NTP 服务器正常同步：

1. 执行以下命令，如果输出 running 表示 NTP 服务正在运行：

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

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

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

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

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

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

如果你使用的系统配置是 `chronyd`，请直接执行步骤 3。

2. 执行 `ntpstat` 命令检测是否与 NTP 服务器同步：

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

```
ntpstat
```

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

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

- 以下情况表示 NTP 服务未正常同步：

```
unsynchronised
```

- 以下情况表示 NTP 服务未正常运行：

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

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

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

```
chronyc tracking
```

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

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

- 如果该命令返回结果如下，则表示同步过程出错：

```
Leap status     : Not synchronised
```

- 如果该命令返回结果如下，则表示 Chrony 服务未正常运行：

```
506 Cannot talk to daemon
```

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

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

如果要在 CentOS 7 系统上手动安装 NTP 服务，可执行以下命令：

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

2.2.5 检查和配置操作系统优化参数

在生产系统的TiDB 中，建议对操作系统进行如下的配置优化：

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

采用如下步骤检查操作系统的当前配置，并配置系统优化参数：

1. 执行以下命令查看透明大页的开启状态。

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

```
[always] madvise never
```

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

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

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

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

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

3. 执行以下命令查看磁盘的唯一标识 `ID_SERIAL`。

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

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

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

4. 执行以下命令查看cpufreq 模块选用的节能策略。

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

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

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

5. 配置系统优化参数

    - 方法一：使用tuned（推荐）

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

        ```
        tuned-adm list
        ```

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

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

        2. 创建新的tuned 策略。

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

```
[main]
include=balanced

[cpu]
governor=performance

[vm]
transparent_hugepages=never

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

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

3. 应用新的tuned 策略。

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

- 方法二：使用脚本方式。如果已经使用tuned 方法，请跳过本方法。

1. 执行 grubby 命令查看默认内核版本。

> 注意：
> 需安装 grubby 软件包。

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

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

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

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

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

3. 执行 grubby --info 命令查看修改后的默认内核配置。

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

> 注意：
> --info 后需要使用实际的默认内核版本。

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

4. 修改当前的内核配置立即关闭透明大页。

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

5. 配置 udev 脚本应用 IO 调度器策略。

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

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

6. 应用 udev 脚本。

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

7. 创建 CPU 节能策略配置服务。

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

8. 应用 CPU 节能策略配置服务。

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

6. 执行以下命令验证透明大页的状态。

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

```
always madvise [never]
```

7. 执行以下命令验证数据目录所在磁盘的I/O 调度器。

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

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

8. 执行以下命令查看cpufreq 模块选用的节能策略。

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

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

9. 执行以下命令修改 sysctl 参数。

```
echo "fs.file-max = 1000000">> /etc/sysctl.conf
echo "net.core.somaxconn = 32768">> /etc/sysctl.conf
echo "net.ipv4.tcp_tw_recycle = 0">> /etc/sysctl.conf
echo "net.ipv4.tcp_syncookies = 0">> /etc/sysctl.conf
echo "vm.overcommit_memory = 1">> /etc/sysctl.conf
sysctl -p
```

10. 执行以下命令配置用户的limits.conf 文件。

```
cat << EOF >>/etc/security/limits.conf
tidb            soft    nofile          1000000
tidb            hard    nofile          1000000
tidb            soft    stack          32768
tidb            hard    stack          32768
EOF
```

2.2.6 手动配置 SSH 互信及 sudo 免密码

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

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

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

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

```
visudo
```

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

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

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

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

```
ssh 10.0.1.1
```

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

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

```
sudo -su root
```

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

2.2.7 安装numactl 工具

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

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

安装 NUMA 工具有两种方法：

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

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

方法 2：通过 tiup cluster exec 在集群上批量安装 NUMA。

1. 使用 TiUP 安装 TiDB 集群，参考使用 TiUP 部署 TiDB 集群完成 tidb-test 集群的部署。如果本地已有集群，可跳过这一步。

```
tiup cluster deploy tidb-test v6.1.0 ./topology.yaml --user root [-p] [-i /home/root/.ssh/
    ↪ gcp_rsa]
```

2. 执行 tiup cluster exec 命令，以 sudo 权限在 tidb-test 集群所有目标主机上安装 NUMA。

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

你可以执行 tiup cluster exec --help 查看的 tiup cluster exec 命令的说明信息。

2.3 规划集群拓扑

2.3.1 最小拓扑架构

本文档介绍TiDB 集群最小部署的拓扑架构。

2.3.1.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	2	16 VCore 32 GiB 100 GiB 用于存储	10.0.1.1 10.0.1.2	默认端口 全局目录配置

实例	个数	物理机配置	IP	配置
PD	3	4 VCore 8 GiB 100 GiB 用于存储	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32 GiB 2 TiB (NVMe SSD) 用于存储	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8 GiB 500 GiB (SSD) 用于存储	10.0.1.10	默认端口全局目录配置

2.3.1.1.1 拓扑模版

简单最小配置模板

详细最小配置模板

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

注意：

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

2.3.2 TiFlash 部署拓扑

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

2.3.2.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
TiFlash	1	32 VCore 64 GB 2TB (nvme ssd) * 1	10.0.1.11	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.10	默认端口全局目录配置

2.3.2.1.1 拓扑模版

- 简单TiFlash 配置模版
- 详细TiFlash 配置模版

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

2.3.2.1.2 关键参数介绍

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

注意：

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

2.3.3 TiCDC 部署拓扑

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

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

2.3.3.1 拓扑信息

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

2.3.3.1.1 拓扑模版

简单TiCDC 配置模板

详细TiCDC 配置模板

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

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

2.3.4 TiDB Binlog 部署拓扑

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

2.3.4.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32 GB	10.0.1.1 10.0.1.2 10.0.1.3	默认端口配置；开启 enable_binlog；开启 ignore-error
PD	3	4 VCore 8 GB	10.0.1.4 10.0.1.5 10.0.1.6	默认端口配置
TiKV	3	16 VCore 32 GB	10.0.1.7 10.0.1.8 10.0.1.9	默认端口配置

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

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

2.3.4.1.1 拓扑模版

简单 TiDB Binlog 配置模板（下游为 MySQL）

简单 TiDB Binlog 配置模板（下游为 file）

详细 TiDB Binlog 配置模板

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

2.3.4.1.2 关键参数介绍

拓扑配置模版的关键参数如下：

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

  开启TiDB Binlog 服务，默认为 false。

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

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

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

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

- `drainer_servers.config.syncer.to`

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

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

2.3.5 TiSpark 部署拓扑

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

关于TiSpark 的架构介绍与使用，参见TiSpark 用户指南。

> 警告：
>
> TiUP Cluster 的TiSpark 支持目前为废弃状态，不建议使用。

2.3.5.1 拓扑信息

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

实例	个数	物理机配置	IP	配置
TiSpark	3	8 VCore 16GB * 1	10.0.1.21 (master) 10.0.1.22 (worker) 10.0.1.23 (worker)	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认端口全局目录配置

2.3.5.1.1 拓扑模版

简单 TiSpark 配置模板

详细 TiSpark 配置模板

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

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

2.3.5.2 环境要求

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

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

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

2.3.6 跨数据中心部署拓扑

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

2.3.6.1 拓扑信息

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

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

2.3.6.1.1 拓扑模版

跨机房配置模板

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

2.3.6.1.2 关键参数配置

本节介绍跨数据中心部署TiDB集群的关键参数配置。

TiKV参数

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

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

- label配置

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

- TiKV 配置

  相同物理机配置相同的host 级别label 信息：

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

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

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

PD 参数

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

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

- 调整Raft Group 的副本数据量为 5，保证集群的高可用性。

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

- 拒绝异地机房TiKV 的 Raft 副本选举为 Leader。

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

> 注意：
> 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。

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

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

2.3.7 混合部署拓扑

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

2.3.7.1 拓扑信息

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

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

核

实例	个数	物理机配置	IP	配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.10	默认配置

2.3.7.1.1 拓扑模版

简单混部配置模板

详细混部配置模板

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

2.3.7.1.2 混合部署的关键参数介绍

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

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

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

    * 计算公式如下：

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

  - storage CF (all RocksDB column families) 内存自适应，配置 `storage.block-cache.capacity` 参数即可实现 CF 之间自动平衡内存使用。
    * 计算公式如下：

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

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

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

- label 调度配置

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

  - TiKV 配置

    相同物理机配置相同的host 级别label 信息：

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

  - PD 配置

    PD 需要配置labels 类型来识别并调度Region：

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

- numa_node 绑核

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

注意：

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

2.4 安装与启动

2.4.1 使用TiUP 部署TiDB 集群

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

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

2.4.1.1 第 1 步：软硬件环境需求及前置检查

软硬件环境需求

环境与系统配置检查

2.4.1.2 第 2 步：在中控机上部署TiUP 组件

在中控机上部署TiUP 组件有两种方式：在线部署和离线部署。

2.4.1.2.1 在线部署

以普通用户身份登录中控机。以 tidb 用户为例，后续安装TiUP 及集群管理操作均通过该用户完成：

1. 执行如下命令安装TiUP 工具：

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

2. 按如下步骤设置TiUP 环境变量：

    1. 重新声明全局环境变量：

    ```
    shell source .bash_profile
    ```

    2. 确认TiUP 工具是否安装：

    ```
    which tiup
    ```

3. 安装TiUP cluster 组件：

```
tiup cluster
```

4. 如果已经安装，则更新TiUP cluster 组件至最新版本：

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

预期输出“Update successfully!” 字样。

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

```
tiup --binary cluster
```

2.4.1.2.2 离线部署

离线部署TiUP 组件的操作步骤如下。

准备TiUP 离线组件包

方式一：在官方下载页面选择对应版本的 TiDB server 离线镜像包（包含 TiUP 离线组件包）。需要同时下载 TiDB-community-server 软件包和TiDB-community-toolkit 软件包。

方式二：使用 tiup mirror clone 命令手动打包离线组件包。步骤如下：

1. 在在线环境中安装TiUP 包管理器工具

    1. 执行如下命令安装TiUP 工具：

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

    2. 重新声明全局环境变量：

    ```
    source .bash_profile
    ```

    3. 确认TiUP 工具是否安装：

    ```
    which tiup
    ```

2. 使用TiUP 制作离线镜像

    1. 在一台和外网相通的机器上拉取需要的组件：

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

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

    2. 通过tar 命令将该组件包打包然后发送到隔离环境的中控机：

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

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

3. 自定义制作的离线镜像，或调整已有离线镜像中的内容

    如果从官网下载的离线镜像不满足你的具体需求，或者希望对已有的离线镜像内容进行调整，例如增加某个组件的新版本等，可以采取以下步骤进行操作：

    1. 在制作离线镜像时，可通过参数指定具体的组件和版本等信息，获得不完整的离线镜像。例如，要制作一个只包括 v1.11.3 版本TiUP 和 TiUP Cluster 的离线镜像，可执行如下命令：

    ```
    tiup mirror clone tiup-custom-mirror-v1.11.3 --tiup v1.11.3 --cluster v1.11.3
    ```

    如果只需要某一特定平台的组件，也可以通过 --os 和 --arch 参数来指定。

    2. 参考上文“使用TiUP 制作离线镜像” 第 2 步的方式，将此不完整的离线镜像传输到隔离环境的中控机。

    3. 在隔离环境的中控机上，查看当前使用的离线镜像路径。较新版本的TiUP 可以直接通过命令获取当前的镜像地址：

    ```
    tiup mirror show
    ```

    以上命令如果提示 show 命令不存在，可能当前使用的是较老版本的 TiUP。此时可以通过查看 $HOME/.tiup/tiup.toml 获得正在使用的镜像地址。将此镜像地址记录下来，后续步骤中将以变量 ${base_mirror} 指代此镜像地址。

4. 将不完整的离线镜像合并到已有的离线镜像中：

   首先将当前离线镜像中的 keys 目录复制到 $HOME/.tiup 目录中：

   ```
   cp -r ${base_mirror}/keys $HOME/.tiup/
   ```

   然后使用TiUP 命令将不完整的离线镜像合并到当前使用的镜像中：

   ```
   tiup mirror merge tiup-custom-mirror-v1.11.3
   ```

5. 上述步骤完成后，通过 tiup list 命令检查执行结果。在本文例子中，使用 tiup list tiup 和 tiup list cluster 均应能看到对应组件的 v1.11.3 版本出现在结果中。

部署离线环境TiUP 组件

将离线包发送到目标集群的中控机后，执行以下命令安装TiUP 组件：

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

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

合并离线包

如果是通过官方下载页面下载的离线软件包，需要将TiDB-community-server 软件包和TiDB-community-toolkit 软件包合并到离线镜像中。如果是通过 tiup mirror clone 命令手动打包的离线组件包，不需要执行此步骤。

执行以下命令合并离线组件到 server 目录下。

```
tar xf tidb-community-toolkit-${version}-linux-amd64.tar.gz
ls -ld tidb-community-server-${version}-linux-amd64 tidb-community-toolkit-${version}-linux-amd64
cd tidb-community-server-${version}-linux-amd64/
cp -rp keys ~/.tiup/
tiup mirror merge ../tidb-community-toolkit-${version}-linux-amd64
```

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

2.4.1.3 第 3 步：初始化集群拓扑文件

执行如下命令，生成集群初始化配置文件：

```
tiup cluster template > topology.yaml
```

针对两种常用的部署场景，也可以通过以下命令生成建议的拓扑模板：

- 混合部署场景：单台机器部署多个实例，详情参见混合部署拓扑架构。

  ```
  tiup cluster template --full > topology.yaml
  ```

- 跨机房部署场景：跨机房部署TiDB 集群，详情参见跨机房部署拓扑架构。

```
tiup cluster template --multi-dc > topology.yaml
```

执行 vi topology.yaml，查看配置文件的内容：

```
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/tidb-deploy"
  data_dir: "/tidb-data"
server_configs: {}
pd_servers:
  - host: 10.0.1.4
  - host: 10.0.1.5
  - host: 10.0.1.6
tidb_servers:
  - host: 10.0.1.7
  - host: 10.0.1.8
  - host: 10.0.1.9
tikv_servers:
  - host: 10.0.1.1
  - host: 10.0.1.2
  - host: 10.0.1.3
monitoring_servers:
  - host: 10.0.1.4
grafana_servers:
  - host: 10.0.1.4
alertmanager_servers:
  - host: 10.0.1.4
```

下表列出了常用的 7 种场景，请根据链接中的拓扑说明以及配置文件模板配置topology.yaml。如果有其他组合场景的需求，请根据多个模板自行调整。

场景	配置任务	配置文件模板	拓扑说明
OLTP 业务	部署最小拓扑架构	简单最小配置模板详细最小配置模板	最小集群拓扑，包括 tidb-server、tikv-server、pd-server。

场景	配置任务	配置文件模板	拓扑说明
HTAP 业务	部署 Ti-Flash 拓扑架构	简单 Ti-Flash 配置模版 详细 Ti-Flash 配置模版	在最小拓扑的基础上部署 Ti-Flash。Ti-Flash 是列式存储引擎，已经逐步成为集群拓扑的标配。

场景	配置任务	配置文件模板	拓扑说明
使用TiCDC进行增量同步	部署TiCDC拓扑架构	简单TiCDC配置模板 详细TiCDC配置模板	在最小拓扑的基础上部署TiCDC。TiCDC支持多种下游(TiDB/MySQL/MQ)。

场景	配置任务	配置文件模板	拓扑说明
使用 TiDB Binlog 进行增量同步	部署 TiDB Binlog 拓扑架构	简单 TiDB Binlog 配置模板（下游为 MySQL） 简单 TiDB Binlog 配置模板（下游为 file） 详细 TiDB Binlog 配置模板	在最小拓扑的基础上部署 TiDB Binlog。

场景	配置任务	配置文件模板	拓扑说明
使用 Spark 的 OLAP 业务	部署 TiSpark 拓扑架构	简单 TiSpark 配置模板 详细 TiSpark 配置模板	在最小拓扑的基础上部署 TiSpark 组件。 TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。 TiUP cluster 组件对 TiSpark 的支

拓扑持	配置文件	配置

场景	配置任务	配置文件模板	拓扑说明
单台机器，多个实例	混合部署拓扑架构	简单混部配置模板详细混部配置模板	也适用于单机多实例需要额外增加目录、端口、资源配比、label 等配置的场景。

场景	配置任务	配置文件模板	拓扑说明
跨机房部署 TiDB 集群	跨机房部署拓扑架构	跨机房配置模板	以典型的两地三中心架构为例，介绍跨机房部署架构，以及需要注意的关键设置。

注意：

- 对于需要全局生效的参数，请在配置文件中 `server_configs` 的对应组件下配置。

- 对于需要某个节点生效的参数，请在具体节点的 config 中配置。
- 配置的层次结构使用 . 表示。如：log.slow-threshold。更多格式参考 TiUP 配置参数模版。
- 如果需要指定在目标机创建的用户组名，可以参考这个例子。

更多参数说明，请参考：

- TiDB config.toml.example
- TiKV config.toml.example
- PD config.toml.example
- TiFlash config.toml.example

2.4.1.4 第 4 步：执行部署命令

注意：

通过TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：

- 如果是密钥方式，可以通过-i 或者–identity_file 来指定密钥的路径。
- 如果是密码方式，可以通过-p 进入密码交互窗口。
- 如果已经配置免密登录目标机，则不需填写认证。

一般情况下TiUP 会在目标机器上创建topology.yaml 中约定的用户和组，以下情况例外：

- topology.yaml 中设置的用户名在目标机器上已存在。
- 在命令行上使用了参数–skip-create-user 明确指定跳过创建用户的步骤。

执行部署命令前，先使用 check 及 check --apply 命令检查和自动修复集群存在的潜在风险：

1. 检查集群存在的潜在风险：

```
tiup cluster check ./topology.yaml --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

2. 自动修复集群存在的潜在风险：

```
tiup cluster check ./topology.yaml --apply --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

3. 部署TiDB 集群：

```
tiup cluster deploy tidb-test v7.0.0 ./topology.yaml --user root [-p] [-i /home/root/.ssh/
    ↪ gcp_rsa]
```

以上部署示例中：

- `tidb-test` 为部署的集群名称。
- `v7.0.0` 为部署的集群版本，可以通过执行 `tiup list tidb` 来查看TiUP 支持的最新可用版本。
- 初始化配置文件为 `topology.yaml`。
- `--user root` 表示通过root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- [-i] 及 [-p] 为可选项，如果已经配置免密登录目标机，则不需填写。否则选择其一即可，[-i] 为可登录到目标机的root 用户（或 –user 指定的其他用户）的私钥，也可使用 [-p] 交互式输入该用户的密码。

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

2.4.1.5 第 5 步：查看TiUP 管理的集群情况

```
tiup cluster list
```

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

2.4.1.6 第 6 步：检查部署的TiDB 集群情况

例如，执行如下命令检查 `tidb-test` 集群情况：

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

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

2.4.1.7 第 7 步：启动集群

安全启动是TiUP cluster 从 v1.9.0 起引入的一种新的启动方式，采用该方式启动数据库可以提高数据库安全性。推荐使用安全启动。

安全启动后，TiUP 会自动生成TiDB root 用户的密码，并在命令行界面返回密码。

> 注意：
>
> - 使用安全启动方式后，不能通过无密码的root 用户登录数据库，你需要记录命令行返回的密码进行后续操作。
> - 该自动生成的密码只会返回一次，如果没有记录或者忘记该密码，请参照忘记root 密码修改密码。

方式一：安全启动

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

预期结果如下，表示启动成功。

```
Started cluster `tidb-test` successfully.
The root password of TiDB database has been changed.
The new password is: 'y_+3Hwp=*AWz8971s6'.
Copy and record it to somewhere safe, it is only displayed once, and will not be stored.
The generated password can NOT be got again in future.
```

方式二：普通启动

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

预期结果输出 `Started cluster `tidb-test` successfully`，表示启动成功。使用普通启动方式后，可通过无密码的root 用户登录数据库。

2.4.1.8 第 8 步：验证集群运行状态

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

预期结果输出：各节点 Status 状态信息为 Up 说明集群状态正常。

2.4.1.9 探索更多

如果你已同时部署了TiFlash，接下来可参阅以下文档：

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

如果你已同时部署了TiCDC，接下来可参阅以下文档：

- Changefeed 概述
- 管理 Changefeed
- TiCDC 故障处理
- TiCDC 常见问题

2.4.2 在 Kubernetes 上部署TiDB

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

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

- TiDB on Kubernetes 用户文档

2.5 验证集群运行状态

在部署完一套 TiDB 集群后，需要检查集群是否正常运行。本文介绍如何通过 TiUP 命令、TiDB Dashboard 和 Grafana 检查集群状态，以及如何登录TiDB 数据库执行简单的 SQL 操作。

2.5.1 通过TiUP 检查集群状态

检查集群状态的命令是 `tiup cluster display <cluster-name>`，例如：

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

预期结果输出：各节点 Status 状态信息为 Up 说明集群状态正常。

2.5.2 通过TiDB Dashboard 和 Grafana 检查集群状态

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

2.5.2.1 查看TiDB Dashboard 检查TiDB 集群状态

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

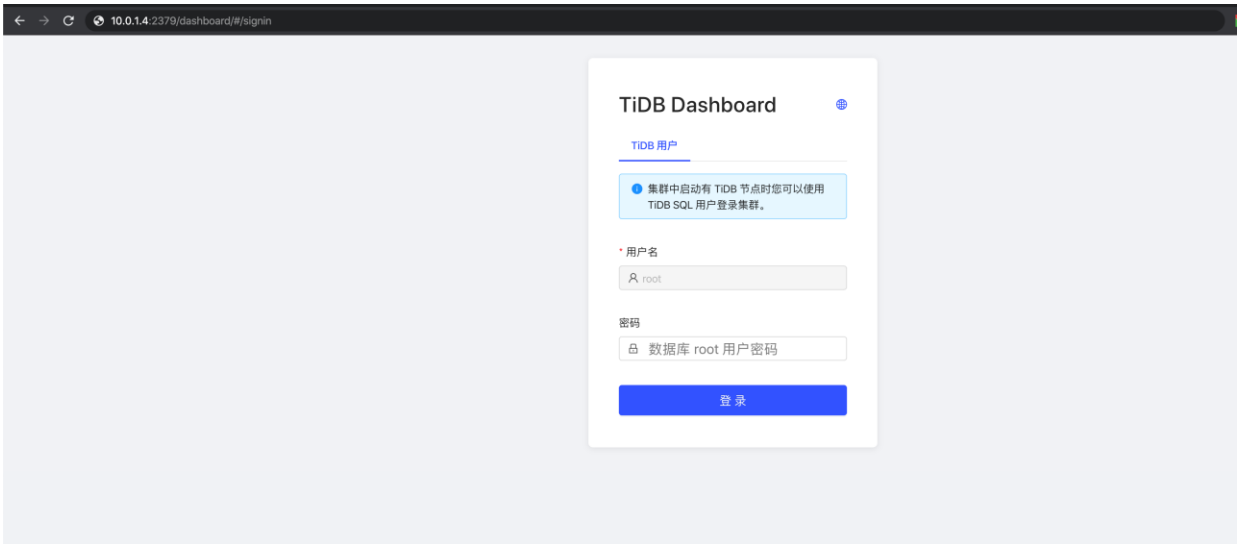


图 2: TiDB-Dashboard

2. 主页面显示TiDB 集群中节点信息

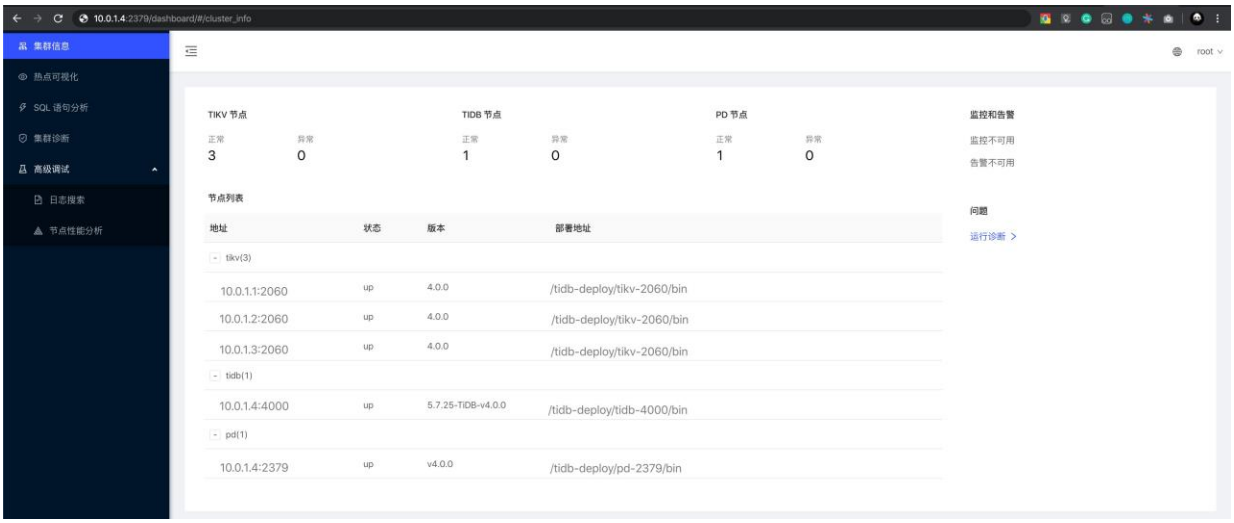


图 3: TiDB-Dashboard-status

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

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

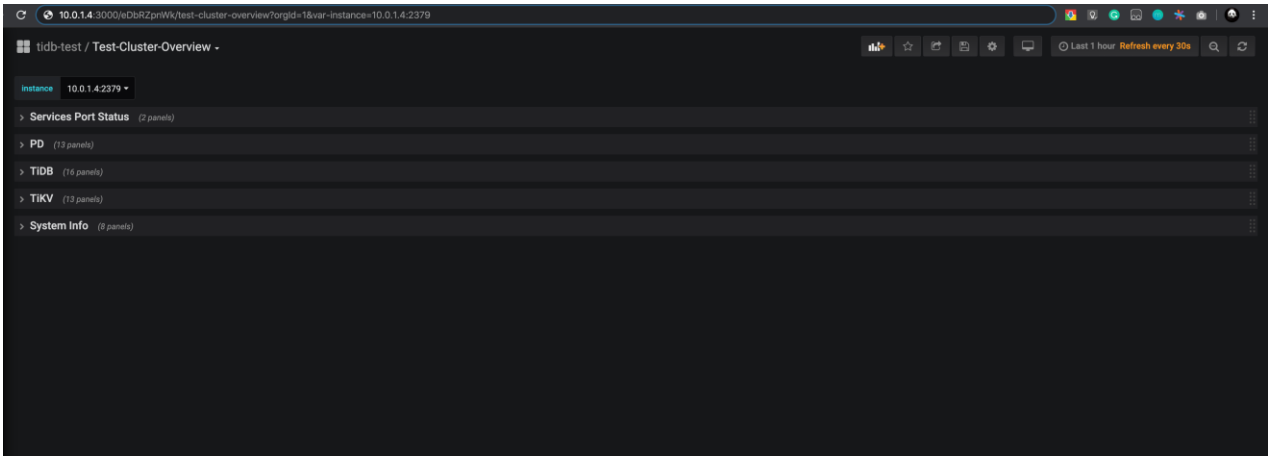


图 4: Grafana-overview

2.5.3 登录数据库执行简单 DML/DDL 操作和查询 SQL 语句

> **注意：**
> 登录数据库前，你需要安装 MySQL 客户端。

执行以下命令登录数据库：

```
mysql -u root -h ${tidb_server_host_IP_address} -P 4000
```

其中，${tidb_server_host_IP_address} 是在初始化集群拓扑文件时为 tidb_servers 配置的IP 地址之一，例如 10.0.1.7。

输出下列信息表示登录成功：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.25-TiDB-v5.0.0 TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7
    '→ compatible

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

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

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

2.5.3.1 数据库操作

- 检查TiDB 版本

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

预期结果输出：

```
*************************** 1. row ***************************
tidb_version(): Release Version: v5.0.0
Edition: Community
Git Commit Hash: 689a6b6439ae7835947fcaccf329a3fc303986cb
Git Branch: HEAD
UTC Build Time: 2020-05-28 11:09:45
GoVersion: go1.13.4
Race Enabled: false
TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306
Check Table Before Drop: false
1 row in set (0.00 sec)
```

- 创建 PingCAP database

```
create database pingcap;
```

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

```
use pingcap;
```

预期输出

```
Database changed
```

- 创建 tab_tidb 表

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

预期输出

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

- 插入数据

```
insert into `tab_tidb` values (1,'TiDB',5,'TiDB-v5.0.0');
```

预期输出

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

- 查看 tab_tidb 结果

```
select * from tab_tidb;
```

预期输出

```
+----+------+-----+--------------------+
| id | name | age | version      |
+----+------+-----+--------------------+
|  1 | TiDB |   5 | TiDB-v5.0.0 |
+----+------+-----+--------------------+
1 row in set (0.00 sec)
```

- 查看TiKV store 状态、store_id、存储情况以及启动时间

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

预期输出

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

- 退出

```
exit
```

预期输出

```
Bye
```

2.6 测试集群性能

2.6.1 如何用 Sysbench 测试 TiDB

建议使用 Sysbench 1.0 或之后的更新版本，可在 Sysbench Release 1.0.20 页面下载。

2.6.1.1 测试方案

2.6.1.1.1 TiDB 配置

升高日志级别，可以减少打印日志数量，对TiDB 的性能有积极影响。具体在TiUP 配置文件中加入：

```
server_configs:
  tidb:
    log.level: "error"
```

同时，推荐启用tidb_enable_prepared_plan_cache，并保证 --db-ps-mode 设置为 auto，这样 Sysbench 就可以使用预处理语句。关于 SQL 执行计划缓存的功能及监控，请参考执行计划缓存。

> **注意：**
>
> 不同版本 Sysbench 的 db-ps-mode 参数默认值可能会不同，建议在命令中显式指定。

2.6.1.1.2 TiKV 配置

升高TiKV 的日志级别同样有利于提高性能表现。

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

Default CF : Write CF = 4 : 1

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

```
server_configs:
  tikv:
    log-level: "error"
    rocksdb.defaultcf.block-cache-size: "24GB"
    rocksdb.writecf.block-cache-size: "6GB"
```

还可以使用共享block cache 的方式进行设置：

```
server_configs:
  tikv:
    storage.block-cache.capacity: "30GB"
```

更详细的TiKV 参数调优请参考TiKV 内存参数性能调优。

2.6.1.2 测试过程

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

2.6.1.2.1 Sysbench 配置

以下为 Sysbench 配置文件样例：

```
mysql-host={TIDB_HOST}
mysql-port=4000
```

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

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

配置文件参考示例如下：

```
mysql-host=172.16.30.33
mysql-port=4000
mysql-user=root
mysql-password=password
mysql-db=sbtest
time=600
threads=16
report-interval=10
db-driver=mysql
```

2.6.1.2.2 数据导入

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

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

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

然后退出客户端。

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

```
create database sbtest;
```

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

假设使用的 Sysbench 版本为 1.0.20，可以通过以下两种方式来修改：

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

> **注意：**
> 此操作为可选操作，仅节约了数据导入时间。

命令行输入以下命令，开始导入数据，config 文件为上一步中配置的文件：

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

2.6.1.2.3 数据预热与统计信息收集

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

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

2.6.1.2.4 Point select 测试命令

```
sysbench --config-file=config oltp_point_select --tables=32 --table-size=10000000 --db-ps-mode=
    ↪ auto --rand-type=uniform run
```

2.6.1.2.5 Update index 测试命令

```
sysbench --config-file=config oltp_update_index --tables=32 --table-size=10000000 --db-ps-mode=
    ↪ auto --rand-type=uniform run
```

2.6.1.2.6 Read-only 测试命令

```
sysbench --config-file=config oltp_read_only --tables=32 --table-size=10000000 --db-ps-mode=auto
    ↪ --rand-type=uniform run
```

2.6.1.3 常见问题

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

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

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

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

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

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

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

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

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

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

2.6.2 如何对TiDB 进行 TPC-C 测试

本文介绍如何对TiDB 进行 TPC-C 测试。

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

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

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

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

我们将以 1000 WAREHOUSE 为例进行测试。

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

本文使用 go-tpc 作为 TPC-C 测试实现，可以通过TiUP 命令下载测试程序：

```
tiup install bench
```

关于TiUP Bench 组件的详细用法可参考TiUP Bench。

假设已部署TiDB 集群，其中TiDB 节点部署在 172.16.5.140、172.16.5.141 实例上，端口都为 4000，可按如下步骤进行 TPC-C 测试。

2.6.2.1 导入数据

导入数据通常是整个 TPC-C 测试中最耗时，也是最容易出问题的阶段。

在 shell 中运行TiUP 命令：

```
tiup bench tpcc -H 172.16.5.140,172.16.5.141 -P 4000 -D tpcc --warehouses 1000 --threads 20
    ′→ prepare
```

基于不同的机器配置，这个过程可能会持续几个小时。如果是小型集群，可以使用较小的WAREHOUSE 值进行测试。

数据导入完成后，可以通过命令 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 4 check` 验证数据正确性。

2.6.2.2 运行测试

运行测试的命令是：

```
tiup bench tpcc -H 172.16.5.140,172.16.5.141 -P 4000 -D tpcc --warehouses 1000 --threads 100 --
    ′→ time 10m run
```

运行过程中控制台上会持续打印测试结果：

```
[Current] NEW_ORDER - Takes(s): 4.6, Count: 5, TPM: 65.5, Sum(ms): 4604, Avg(ms): 920, 90th(ms):
    ′→ 1500, 99th(ms): 1500, 99.9th(ms): 1500
[Current] ORDER_STATUS - Takes(s): 1.4, Count: 1, TPM: 42.2, Sum(ms): 256, Avg(ms): 256, 90th(ms)
    ′→ : 256, 99th(ms): 256, 99.9th(ms): 256
[Current] PAYMENT - Takes(s): 6.9, Count: 5, TPM: 43.7, Sum(ms): 2208, Avg(ms): 441, 90th(ms):
    ′→ 512, 99th(ms): 512, 99.9th(ms): 512
[Current] STOCK_LEVEL - Takes(s): 4.4, Count: 1, TPM: 13.8, Sum(ms): 224, Avg(ms): 224, 90th(ms):
    ′→  256, 99th(ms): 256, 99.9th(ms): 256
...
```

运行结束后，会打印测试统计结果：

```
[Summary] DELIVERY - Takes(s): 455.2, Count: 32, TPM: 4.2, Sum(ms): 44376, Avg(ms): 1386, 90th(ms
    ′→ ): 2000, 99th(ms): 4000, 99.9th(ms): 4000
[Summary] DELIVERY_ERR - Takes(s): 455.2, Count: 1, TPM: 0.1, Sum(ms): 953, Avg(ms): 953, 90th(ms
    ′→ ): 1000, 99th(ms): 1000, 99.9th(ms): 1000
[Summary] NEW_ORDER - Takes(s): 487.8, Count: 314, TPM: 38.6, Sum(ms): 282377, Avg(ms): 899, 90th
    ′→ (ms): 1500, 99th(ms): 1500, 99.9th(ms): 1500
[Summary] ORDER_STATUS - Takes(s): 484.6, Count: 29, TPM: 3.6, Sum(ms): 8423, Avg(ms): 290, 90th(
    ′→ ms): 512, 99th(ms): 1500, 99.9th(ms): 1500
```

```
[Summary] PAYMENT - Takes(s): 490.1, Count: 321, TPM: 39.3, Sum(ms): 144708, Avg(ms): 450, 90th(
    ↪ ms): 512, 99th(ms): 1000, 99.9th(ms): 1500
[Summary] STOCK_LEVEL - Takes(s): 487.6, Count: 41, TPM: 5.0, Sum(ms): 9318, Avg(ms): 227, 90th(
    ↪ ms): 512, 99th(ms): 1000, 99.9th(ms): 1000
```

测试完成之后，也可以运行 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 4 check` 进行数据正确性验证。

2.6.2.3 清理测试数据

```
tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 4 cleanup
```

2.6.3 如何对TiDB 进行 CH-benCHmark 测试

本文介绍如何对TiDB 进行 CH-benCHmark 测试。

CH-benCHmark 是包含 TPC-C 和 TPC-H 的混合负载，也是用于测试 HTAP 系统的最常见负载。更多信息，请参考 The mixed workload CH-benCHmark。

在进行 CH-benCHmark 测试前，你需要先部署TiDB 的 HTAP 组件TiFlash。部署TiFlash 并创建TiFlash 副本后，对于 TPC-C 联机交易数据，系统将实时同步最新的数据到TiFlash 组件；TiDB 优化器会自动将 TPC-H 负载的 OLAP 查询下推到TiFlash MPP 引擎进行高效执行。

本文使用 go-tpc 作为 CH 测试实现，可以通过TiUP 命令下载测试程序：

```
tiup install bench
```

关于TiUP Bench 组件的详细用法可参考TiUP Bench。

2.6.3.1 导入数据

2.6.3.1.1 导入 TPC-C 数据

导入数据通常是整个 TPC-C 测试中最耗时、也是最容易出问题的阶段。

本文以 1000 WAREHOUSE 为例，在 shell 中运行以下 TiUP 命令进行数据导入和测试。注意你需要将本文中的 172.16.5.140 和 4000 替换为你实际的TiDB host 和 port 值。

```
tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 1000 prepare -T 32
```

基于不同的机器配置，数据导入过程可能会持续几个小时。如果是小型集群，可以使用较小的WAREHOUSE 值进行测试。

数据导入完成后，可以通过命令 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 1000 ↪ check` 验证数据正确性。

2.6.3.1.2 导入 TPC-H 所需额外的表和视图

在 shell 中运行TiUP 命令：

```
tiup bench ch -H 172.16.5.140 -P 4000 -D tpcc prepare
```

日志输出如下：

```
creating nation
creating region
creating supplier
generating nation table
generate nation table done
generating region table
generate region table done
generating suppliers table
generate suppliers table done
creating view revenue1
```

2.6.3.2 创建TiFlash 副本

部署TiFlash 后，TiFlash 并不会自动同步TiKV 数据，你需要执行以下 SQL 语句创建整库的TiFlash 副本。创建 TiFlash 副本后，系统自动实时同步最新数据到TiFlash 组件。例如，当集群中部署了两个 TiFlash 节点时，如果将replica 设置为 2，执行以下 SQL 语句将创建两个TiFlash 副本。

```
ALTER DATABASE tpcc SET TIFLASH REPLICA 2;
```

可通过如下 SQL 语句确认所有表（通过WHERE 语句可以指定需要确认的表，去掉WHERE 语句则查看所有表）的 TiFlash 副本的状态是否完成同步：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'tpcc';
```

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，进度值在 0 到 1 之间，1 代表TiFlash 副本已经完成同步。

2.6.3.3 搜集统计信息

为了确保优化器能生成最优的执行计划，请执行以下 SQL 语句提前搜集统计信息：

```
analyze table customer;
analyze table district;
analyze table history;
analyze table item;
analyze table new_order;
analyze table order_line;
```

```
analyze table orders;
analyze table stock;
analyze table warehouse;
analyze table nation;
analyze table region;
analyze table supplier;
```

2.6.3.4 运行测试

以 50 TP 并发，1 AP 并发为例，运行以下测试命令：

```
tiup bench ch --host 172.16.5.140 -P4000 --warehouses 1000 run -D tpcc -T 50 -t 1 --time 1h
```

命令运行过程中，控制台上会持续打印测试结果。例如：

```
[Current] NEW_ORDER - Takes(s): 10.0, Count: 13524, TPM: 81162.0, Sum(ms): 998317.6, Avg(ms):
    ′→ 73.9, 50th(ms): 71.3, 90th(ms): 100.7, 95th(ms): 113.2, 99th(ms): 159.4, 99.9th(ms):
    ′→ 209.7, Max(ms): 243.3
[Current] ORDER_STATUS - Takes(s): 10.0, Count: 1132, TPM: 6792.7, Sum(ms): 16196.6, Avg(ms):
    ′→ 14.3, 50th(ms): 13.1, 90th(ms): 24.1, 95th(ms): 27.3, 99th(ms): 37.7, 99.9th(ms): 50.3,
    ′→ Max(ms): 52.4
 [Current] PAYMENT - Takes(s): 10.0, Count: 12977, TPM: 77861.1, Sum(ms): 773982.0, Avg(ms): 59.7,
     ′→  50th(ms): 56.6, 90th(ms): 88.1, 95th(ms): 100.7, 99th(ms): 151.0, 99.9th(ms): 201.3, Max
    ′→ (ms): 243.3
[Current] STOCK_LEVEL - Takes(s): 10.0, Count: 1134, TPM: 6806.0, Sum(ms): 31220.9, Avg(ms):
    ′→ 27.5, 50th(ms): 25.2, 90th(ms): 37.7, 95th(ms): 44.0, 99th(ms): 71.3, 99.9th(ms): 117.4,
    ′→ Max(ms): 125.8
[Current] Q11     - Count: 1, Sum(ms): 3682.9, Avg(ms): 3683.6
[Current] DELIVERY - Takes(s): 10.0, Count: 1167, TPM: 7002.6, Sum(ms): 170712.9, Avg(ms): 146.3,
    ′→  50th(ms): 142.6, 90th(ms): 192.9, 95th(ms): 209.7, 99th(ms): 251.7, 99.9th(ms): 335.5,
    ′→ Max(ms): 385.9
[Current] NEW_ORDER - Takes(s): 10.0, Count: 13238, TPM: 79429.5, Sum(ms): 1010795.3, Avg(ms):
    ′→ 76.4, 50th(ms): 75.5, 90th(ms): 104.9, 95th(ms): 117.4, 99th(ms): 159.4, 99.9th(ms):
    ′→ 234.9, Max(ms): 352.3
[Current] ORDER_STATUS - Takes(s): 10.0, Count: 1224, TPM: 7350.6, Sum(ms): 17874.1, Avg(ms):
    ′→ 14.6, 50th(ms): 13.6, 90th(ms): 23.1, 95th(ms): 27.3, 99th(ms): 37.7, 99.9th(ms): 54.5,
    ′→ Max(ms): 60.8
 [Current] PAYMENT - Takes(s): 10.0, Count: 12650, TPM: 75901.1, Sum(ms): 761981.3, Avg(ms): 60.3,
     ′→  50th(ms): 56.6, 90th(ms): 88.1, 95th(ms): 104.9, 99th(ms): 159.4, 99.9th(ms): 218.1, Max
    ′→ (ms): 318.8
[Current] STOCK_LEVEL - Takes(s): 10.0, Count: 1179, TPM: 7084.9, Sum(ms): 32829.8, Avg(ms):
    ′→ 27.9, 50th(ms): 26.2, 90th(ms): 37.7, 95th(ms): 44.0, 99th(ms): 71.3, 99.9th(ms): 100.7,
    ′→ Max(ms): 117.4
[Current] Q12     - Count: 1, Sum(ms): 9945.8, Avg(ms): 9944.7
[Current] Q13     - Count: 1, Sum(ms): 1729.6, Avg(ms): 1729.6
...
```

命令运行结束后，控制台会打印测试统计结果。例如：

```
Finished: 50 OLTP workers, 1 OLAP workers
[Summary] DELIVERY - Takes(s): 3599.7, Count: 501795, TPM: 8363.9, Sum(ms): 63905178.8, Avg(ms):
    ′→ 127.4, 50th(ms): 125.8, 90th(ms): 167.8, 95th(ms): 184.5, 99th(ms): 226.5, 99.9th(ms):
    ′→ 318.8, Max(ms): 604.0
[Summary] DELIVERY_ERR - Takes(s): 3599.7, Count: 14, TPM: 0.2, Sum(ms): 1027.7, Avg(ms): 73.4,
    ′→ 50th(ms): 71.3, 90th(ms): 109.1, 95th(ms): 109.1, 99th(ms): 113.2, 99.9th(ms): 113.2, Max
    ′→ (ms): 113.2
 [Summary] NEW_ORDER - Takes(s): 3599.7, Count: 5629221, TPM: 93826.9, Sum(ms): 363758020.7, Avg(
     ′→ ms): 64.6, 50th(ms): 62.9, 90th(ms): 88.1, 95th(ms): 100.7, 99th(ms): 130.0, 99.9th(ms):
    ′→ 184.5, Max(ms): 570.4
[Summary] NEW_ORDER_ERR - Takes(s): 3599.7, Count: 20, TPM: 0.3, Sum(ms): 404.2, Avg(ms): 20.2,
    ′→ 50th(ms): 18.9, 90th(ms): 37.7, 95th(ms): 50.3, 99th(ms): 56.6, 99.9th(ms): 56.6, Max(ms)
    ′→ : 56.6
[Summary] ORDER_STATUS - Takes(s): 3599.8, Count: 500318, TPM: 8339.0, Sum(ms): 7135956.6, Avg(ms
    ′→ ): 14.3, 50th(ms): 13.1, 90th(ms): 24.1, 95th(ms): 27.3, 99th(ms): 37.7, 99.9th(ms):
    ′→ 50.3, Max(ms): 385.9
[Summary] PAYMENT - Takes(s): 3599.8, Count: 5380815, TPM: 89684.8, Sum(ms): 269863092.5, Avg(ms)
    ′→ : 50.2, 50th(ms): 48.2, 90th(ms): 75.5, 95th(ms): 88.1, 99th(ms): 125.8, 99.9th(ms):
    ′→ 184.5, Max(ms): 1073.7
 [Summary] PAYMENT_ERR - Takes(s): 3599.8, Count: 11, TPM: 0.2, Sum(ms): 313.0, Avg(ms): 28.5, 50
     ′→ th(ms): 10.0, 90th(ms): 67.1, 95th(ms): 67.1, 99th(ms): 88.1, 99.9th(ms): 88.1, Max(ms):
    ′→ 88.1
[Summary] STOCK_LEVEL - Takes(s): 3599.8, Count: 500467, TPM: 8341.5, Sum(ms): 13208726.4, Avg(ms
    ′→ ): 26.4, 50th(ms): 25.2, 90th(ms): 37.7, 95th(ms): 44.0, 99th(ms): 62.9, 99.9th(ms):
    ′→ 96.5, Max(ms): 570.4
[Summary] STOCK_LEVEL_ERR - Takes(s): 3599.8, Count: 2, TPM: 0.0, Sum(ms): 7.6, Avg(ms): 3.7, 50
    ′→ th(ms): 3.1, 90th(ms): 4.7, 95th(ms): 4.7, 99th(ms): 4.7, 99.9th(ms): 4.7, Max(ms): 4.7
tpmC: 93826.9, efficiency: 729.6%
[Summary] Q1      - Count: 11, Sum(ms): 42738.2, Avg(ms): 3885.3
[Summary] Q10     - Count: 11, Sum(ms): 440370.3, Avg(ms): 40034.3
[Summary] Q11     - Count: 11, Sum(ms): 44208.6, Avg(ms): 4018.7
[Summary] Q12     - Count: 11, Sum(ms): 105320.3, Avg(ms): 9574.6
[Summary] Q13     - Count: 11, Sum(ms): 19199.5, Avg(ms): 1745.4
[Summary] Q14     - Count: 11, Sum(ms): 84582.1, Avg(ms): 7689.5
[Summary] Q15     - Count: 11, Sum(ms): 271944.8, Avg(ms): 24722.8
[Summary] Q16     - Count: 11, Sum(ms): 183894.9, Avg(ms): 16718.1
[Summary] Q17     - Count: 11, Sum(ms): 89018.9, Avg(ms): 8092.7
[Summary] Q18     - Count: 10, Sum(ms): 767814.5, Avg(ms): 76777.6
[Summary] Q19     - Count: 10, Sum(ms): 17099.1, Avg(ms): 1709.8
[Summary] Q2      - Count: 11, Sum(ms): 53513.6, Avg(ms): 4865.2
[Summary] Q20     - Count: 10, Sum(ms): 73717.7, Avg(ms): 7372.1
[Summary] Q21     - Count: 10, Sum(ms): 166001.4, Avg(ms): 16601.1
[Summary] Q22     - Count: 10, Sum(ms): 48268.4, Avg(ms): 4827.7
[Summary] Q3      - Count: 11, Sum(ms): 31110.1, Avg(ms): 2828.5
```

```
[Summary] Q4      - Count: 11, Sum(ms): 83814.2, Avg(ms): 7619.3
[Summary] Q5      - Count: 11, Sum(ms): 368301.0, Avg(ms): 33483.5
[Summary] Q6      - Count: 11, Sum(ms): 61702.5, Avg(ms): 5608.9
[Summary] Q7      - Count: 11, Sum(ms): 158928.2, Avg(ms): 14446.3
```

测试完成之后，也可以运行 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 1000 check` 验证数据正确性。

3 数据迁移

3.1 数据迁移概述

本文档总体介绍可用于TiDB 的数据迁移方案。数据迁移方案如下：

- 全量数据迁移。
  - 数据导入：使用TiDB Lightning 将 Aurora Snapshot，CSV 文件或 Mydumper SQL 文件的数据全量导入到 TiDB 集群。
  - 数据导出：使用 Dumpling 将 TiDB 集群的数据全量导出为CSV 文件或 Mydumper SQL 文件，从而更好地配合从 MySQL 数据库或 MariaDB 数据库进行数据迁移。
  - TiDB DM (Data migration) 也提供了适合小规模数据量数据库（例如小于 1 TiB）的全量数据迁移功能。
- 快速初始化TiDB 集群：TiDB Lightning 提供的快速导入功能可以实现快速初始化TiDB 集群的指定表的效果。请注意，使用快速初始化 TiDB 集群的功能对TiDB 集群的影响极大，在进行初始化的过程中，TiDB 集群不支持对外访问。
- 增量数据迁移：使用TiDB DM 从 MySQL，MariaDB 或 Aurora 同步 Binlog 到 TiDB，该功能可以极大降低业务迁移过程中停机窗口时间。
- TiDB 集群复制：TiDB 支持备份恢复功能，该功能可以实现将TiDB 的某个快照初始化到另一个全新的TiDB 集群。
- TiDB 集群增量数据同步：TiCDC 支持同构数据库之间的灾备场景，能够在灾难发生时保证主备集群数据的最终一致性。目前该场景仅支持TiDB 作为主备集群。

根据迁移数据所在数据库类型、部署位置、业务数据规模大小、业务需求等因素，会有不同数据迁移选择。下面展示一些常用的数据迁移场景，方便用户依据这些线索选择到最适合自己的数据迁移方案。

3.1.1 迁移Aurora MySQL 到 TiDB

从 Aurora 迁移数据到部署在AWS 的 TiDB 集群，数据迁移可以分为全量迁移和增量迁移两个步骤进行。请根据你的业务需求选择相应的步骤。

- 从 Aurora 迁移数据到TiDB

3.1.2 迁移 MySQL 到 TiDB

如果你没有使用 Cloud storage (S3) 服务，而且网络联通和延迟情况良好，那么从 MySQL 迁移数据到TiDB 时可以参照下面的方案。

- 从小数据量 MySQL 迁移数据到TiDB

如果你对数据迁移速度有要求，或者数据规模特别大（例如大于 1 TiB），并且禁止 TiDB 集群在迁移期间有其他业务写入，那么你可以先使用 Lightning 进行快速导入，然后根据业务需要选择是否使用 DM 进行增量数据 (Binlog) 同步。

- 从大数据量 MySQL 迁移数据到TiDB

3.1.3 分库分表 MySQL 合并迁移到TiDB

如果你的业务使用了基于 MySQL 分库的方案来存储数据，业务数据从 MySQL 迁移到TiDB 后，合并这些分表数据到一张合并，那么你可以使用 DM 进行分表合并迁移。

- 从小数据量分库分表 MySQL 合并迁移数据到TiDB

如果分表数据总规模特别大（例如大于 1 TiB），并且禁止 TiDB 集群在迁移期间有其他业务写入，那么你可以使用 Lightning 对分表数据进行快速合并导入，然后根据业务需要选择是否使用 DM 进行增量数据(Binlog) 的分表同步。

- 从大数据量分库分表 MySQL 合并迁移数据到TiDB

3.1.4 从文件迁移数据到TiDB

- 从 CSV 文件迁移数据到TiDB
- 从 SQL 文件迁移数据到TiDB
- 从 Parquet 文件迁移数据到TiDB

3.1.5 TiDB 集群增量数据同步

可以使用TiCDC 进行TiDB 集群间的增量数据同步。详情请参考TiCDC 简介。

3.1.6 复杂迁移场景

DM 在实时同步过程中，多个已有特性可以使得同步过程更加灵活，适应各类业务需求：

- 上游使用pt/gh-ost 工具的持续同步场景
- 下游存在更多列的迁移场景
- 如何过滤binlog 事件
- 如何通过 SQL 表达式过滤binlog

3.2 数据迁移工具概览

TiDB 提供了丰富的数据迁移相关的工具，用于全量迁移、增量迁移、备份恢复、数据同步等多种场景。

本文介绍了使用这些工具的场景、支持的上下游、优势和相关限制等信息。请根据你的需求选择合适的工具。

3.2.1 TiDB Data Migration (DM)

使用场景	用于将数据从与 MySQL 协议兼容的数据库迁移到TiDB
上游	MySQL，MariaDB，Aurora
下游	TiDB

| 主要优势 |

一体化的数据迁移任务管理工具，支持全量迁移和增量同步

支持对表与操作进行过滤

支持分库分表的合并迁移

使用限制 | 数据导入速度与TiDB Lightning 的Logical Import Mode 大致相同，而比TiDB Lightning 的Physical Import Mode 低很多。建议用于 1 TB 以内的存量数据迁移。 |

3.2.2 TiDB Lightning

使用场景	用于将数据全量导入到TiDB

| 上游（输入源文件） |

Dumpling 输出的文件

从 Amazon Aurora 或 Apache Hive 导出的 Parquet 文件

CSV 文件

从本地盘或Amazon S3 云盘读取数据

| | 下 游 | TiDB | | 主要优势 |

支持快速导入大量数据，实现快速初始化TiDB 集群的指定表

支持断点续传

支持数据过滤

| | 使用限制 |

如果使用Physical Import Mode 进行数据导入，TiDB Lightning 运行后，TiDB 集群将无法正常对外提供服务。

如果你不希望TiDB 集群的对外服务受到影响，可以参考TiDB Lightning Logical Import Mode 中的硬件需求与部署方式进行数据导入。

|

3.2.3 Dumping

使用场景	用于将数据从 MySQL/TiDB 进行全量导出
上游	MySQL，TiDB
下游（输出文件）	SQL，CSV

| 主要优势 |

支持全新的table-filter，筛选数据更加方便

支持导出到Amazon S3 云盘

| | 使用限制 |

如果导出后计划往非TiDB 的数据库恢复，建议使用 Dumpling。

如果导出后计划往另一个TiDB 恢复，建议使用BR。

|

3.2.4 TiCDC

使用场景	通过拉取 TiKV 变更日志实现的 TiDB 增量数据同步工具，具有将数据还原到与上游任意 TSO 一致状态的能力，支持
上游	TiDB
下游	TiDB，MySQL，Kafka，Confluent
主要优势	提供开放数据协议(TiCDC Open Protocol)。

| 使用限制 | TiCDC 只能同步至少存在一个有效索引的表。暂不支持以下场景：

单独使用RawKV 的 TiKV 集群。

在 TiDB 中创建 SEQUENCE 的 DDL 操作和 SEQUENCE 函数。

|

3.2.5 Backup & Restore (BR)

使用场景	通过对大数据量的TiDB 集群进行数据备份和恢复，实现数据迁移
上游	TiDB
下游（输出文件）	SST，backup.meta 文件，backup.lock 文件

| 主要优势 |

适用于向另一个TiDB 迁移数据。

支持数据冷备份到外部存储，可以用于灾备恢复。

| | 使用限制 |

BR 恢复到TiCDC / Drainer 的上游集群时，恢复数据无法由TiCDC / Drainer 同步到下游。

BR 只支持在 new_collations_enabled_on_first_bootstrap 开关值相同的集群之间进行操作。

|

3.2.6 sync-diff-inspector

使用场景	用于校验 MySQL/TiDB 中两份数据的一致性
上游	TiDB，MySQL
下游	TiDB，MySQL
主要优势	提供了修复数据的功能，适用于修复少量不一致的数据。

| 使用限制 |

对于 MySQL 和 TiDB 之间的数据同步不支持在线校验。

不支持JSON、 BIT、 BINARY、 BLOB 等类型的数据。

|

3.2.7 使用TiUP 快速安装

从TiDB 4.0 开始，TiUP 作为软件包管理器，帮助你轻松管理TiDB 生态系统中的不同集群组件。现在你可以只用一个TiUP 命令行来管理任何组件。

3.2.7.1 第 1 步：安装TiUP

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

重新声明全局环境变量：

```
source ~/.bash_profile
```

3.2.7.2 第 2 步：安装组件

你可以通过以下命令查看所有可用组件:

```
tiup list
```

以下输出为所有可用组件：

```
Available components:
Name            Owner    Description
----            -----    -----------
bench           pingcap  Benchmark database with different workloads
br              pingcap  TiDB/TiKV cluster backup restore tool
cdc             pingcap  CDC is a change data capture tool for TiDB
client          pingcap  Client to connect playground
cluster         pingcap  Deploy a TiDB cluster for production
ctl             pingcap  TiDB controller suite
dm              pingcap  Data Migration Platform manager
dmctl           pingcap  dmctl component of Data Migration Platform
errdoc          pingcap  Document about TiDB errors
pd-recover      pingcap  PD Recover is a disaster recovery tool of PD, used to recover the PD
    '→ cluster which cannot start or provide services normally
playground      pingcap  Bootstrap a local TiDB cluster for fun
tidb            pingcap  TiDB is an open source distributed HTAP database compatible with the
    '→ MySQL protocol
tidb-lightning  pingcap  TiDB Lightning is a tool used for fast full import of large amounts of
    '→ data into a TiDB cluster
tiup            pingcap  TiUP is a command-line component management tool that can help to
    '→ download and install TiDB platform components to the local system
```

选择所需要的组件进行安装

```
tiup install dumpling tidb-lightning
```

> Note:
>
> 如果需要安装特定版本，可以使用 `tiup install <component>[:version]` 命令.

3.2.7.3 第 3 步：更新TiUP 及组件(可选)

建议先查看新版本的更新日志及兼容性说明：

```
tiup update --self && tiup update dm
```

3.2.8 探索更多

- 离线方式安装TiUP
- 以二进制包形式安装各工具

3.3 数据迁移场景

3.3.1 从 Amazon Aurora 迁移数据到TiDB

本文档介绍如何从Amazon Aurora 迁移数据到TiDB，迁移过程采用 DB snapshot，可以节约大量的空间和时间成本。整个迁移包含两个过程：

- 使用 Lightning 导入全量数据到TiDB
- 使用 DM 持续增量同步到TiDB（可选）

3.3.1.1 前提条件

- 安装 Dumpling 和 Lightning。
- 获取 Dumpling 所需上游数据库权限。
- 获取 Lightning 所需下游数据库权限。

3.3.1.2 导入全量数据到TiDB

3.3.1.2.1 第 1 步：导出Aurora 快照文件到Amazon S3

1. 在 Aurora 上，执行以下命令，查询并记录当前binlog 位置：

```
mysql> SHOW MASTER STATUS;
```

你将得到类似以下的输出，请记录binlog 名称和位置，供后续步骤使用：

```
+------------------+----------+--------------+------------------+-------------------------------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------------------------------+
| mysql-bin.000002 |    52806 |              |                  |                                           |
+------------------+----------+--------------+------------------+-------------------------------------------+
1 row in set (0.012 sec)
```

2. 导出Aurora 快照文件。具体方式请参考Aurora 的官方文档：Exporting DB snapshot data to Amazon S3.

请注意，上述两步的时间间隔建议不要超过 5 分钟，否则记录的binlog 位置过旧可能导致增量同步时产生数据冲突。

完成上述两步后，你需要准备好以下信息：

- 创建快照点时，Aurora binlog 的名称及位置。
- 快照文件的 S3 路径，以及具有访问权限的 SecretKey 和 AccessKey。

3.3.1.2.2 第 2 步：导出 schema

因为Aurora 生成的快照文件并不包含建表语句文件，所以你需要使用Dumpling 自行导出schema 并使用Lightning 在下游创建 schema。你也可以跳过此步骤，并以手动方式在下游自行创建 schema。

运行以下命令，建议使用 `--filter` 参数仅导出所需表的 schema：

```
tiup dumpling --host ${host} --port 3306 --user root --password ${password} --filter 'my_db1.
    '→ table[12],mydb.*' --consistency none --no-data --output 's3://my-bucket/schema-backup'
```

命令中所用参数描述如下。如需更多信息可参考Dumpling overview。

参数	说明
-u 或 –user	Aurora MySQL 数据库的用户
-p 或 –password	MySQL 数据库的用户密码
-P 或 –port	MySQL 数据库的端口
-h 或 –host	MySQL 数据库的 IP 地址
-t 或 –thread	导出的线程数
-o 或 –output	存储导出文件的目录，支持本地文件路径或外部存储 URI 格式
-r 或 –row	单个文件的最大行数
-F	指定单个文件的最大大小，单位为 MiB，建议值 256 MiB
-B 或 –database	导出指定数据库
-T 或 –tables-list	导出指定数据表
-d 或 –no-data	不导出数据，仅导出 schema
-f 或 –filter	导出能匹配模式的表，不可与-T 一起使用，语法可参考table filter
–consistency	导出期间的一致性模式，Aurora 只能使用 none，即不加锁 dump，不保证一致性

3.3.1.2.3 第 3 步：编写 Lightning 配置文件

根据以下内容创建 `tidb-lightning.toml` 配置文件：

```
vim tidb-lightning.toml
```

```
[tidb]

### 目标 TiDB 集群信息.
host = ${host}                  # 例如：172.16.32.1
port = ${port}                  # 例如：4000
user = "${user_name}"           # 例如："root"
password = "${password}"        # 例如："rootroot"

status-port = ${status-port} # 表结构信息在从 TiDB 的“状态端口”获取例如：10080
pd-addr = "${ip}:${port}"      # 集群 PD 的地址，lightning 通过 PD 获取部分信息，例如
    '→ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    '→ 否则导入将出现异常。


[tikv-importer]
### "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
```

```
### "tidb": TB 级以下数据量也可以采用`tidb`后端模式，下游 TiDB 可正常提供服务。
    ′→ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-backends
backend = "local"

 ### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小，
     ′→建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO 会获得更好的导入性能。
sorted-kv-dir = "${path}"

[mydumper]
### 快照文件的地址
data-source-dir = "${s3_path}"  # eg: s3://my-bucket/sql-backup

[[mydumper.files]]
### 解析 parquet 文件所需的表达式
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$'
schema = '$1'
table = '$2'
type = '$3'
```

如果需要在TiDB 开启 TLS，请参考TiDB Lightning Configuration。

3.3.1.2.4 第 4 步：导入全量数据到TiDB

1. 使用 Lightning 在下游TiDB 建表:

   ```
   tiup tidb-lightning -config tidb-lightning.toml -d 's3://my-bucket/schema-backup'
   ```

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   将有权限访问该Amazon S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

   ```
   export AWS_ACCESS_KEY_ID=${access_key}
   export AWS_SECRET_ACCESS_KEY=${secret_key}
   nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
   ```

3. 导入开始后，可以采用以下任意方式查看进度：

   - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
   - 通过监控面板查看进度，请参考TiDB Lightning 监控。
   - 通过Web 页面查看进度，请参考Web 界面。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure ′→ completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

3.3.1.3 持续增量同步数据到TiDB（可选）

3.3.1.3.1 前提条件

- 安装 DM 集群
- 获取 DM 所需上下游数据库权限

3.3.1.3.2 第 1 步：创建数据源

1. 新建 `source1.yaml` 文件，写入以下内容：

```
# 唯一命名，不可重复。
source-id: "mysql-01"

# DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式
    ↪ 。若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: false


from:
  host: "${host}"          # 例如：172.16.10.81
  user: "root"
  password: "${password}" # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

2. 在终端中执行下面的命令，使用 `tiup dmctl` 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
`--master-addr`	dmctl 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`，例如：172.16.10.71:8261
`operate-source create`	向 DM 集群加载数据源

3.3.1.3.3 第 2 步：创建迁移任务

新建 task1.yaml 文件，写入以下内容：

```
### 任务名，多个同时运行的任务不能重名。
name: "test"
### 任务模式，可设为
### full：只进行全量数据迁移
### incremental： binlog 实时同步
### all： 全量 + binlog 迁移
task-mode: "incremental"
### 下游 TiDB 配置信息。
target-database:
  host: "${host}"                  # 例如：172.16.10.83
  port: 4000
  user: "root"
  password: "${password}"          # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用

### 黑白名单全局配置，各实例通过配置项名引用。
block-allow-list:                    # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  listA:                             # 名称
    do-tables:                       # 需要迁移的上游表的白名单。
    - db-name: "test_db"             # 需要迁移的表的库名。
      tbl-name: "test_table"         # 需要迁移的表的名称。

### 配置数据源
mysql-instances:
  - source-id: "mysql-01"              # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "listA"          # 引入上面黑白名单配置。
###    syncer-config-name: "global"        # syncer 配置的名称
    meta:                              # `task-mode` 为 `incremental` 且下游数据库的 `checkpoint
        ↪ ` 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `checkpoint` 为准。如果
        ↪ `meta` 项和下游数据库的 `checkpoint` 都不存在，则从上游当前最新的 binlog 位置开始迁移
        ↪ 。
      binlog-name: "mysql-bin.000004"    # “Step 1. 导出 Aurora 快照文件到 Amazon S3”
          ↪ 中记录的日志位置，当上游存在主从切换时，必须使用 gtid。
      binlog-pos: 109227
      # binlog-gtid: "09bec856-ba95-11ea-850a-58f2b4af5188:1-9"

### 【可选配置】如果增量数据迁移需要重复迁移已经在全量数据迁移中完成迁移的数据，则需要开启 safe
    ↪ mode 避免增量数据迁移报错。
   ## 该场景多见于以下情况：全量迁移的数据不属于数据源的一个一致性快照，
       ↪ 随后从一个早于全量迁移数据之前的位置开始同步增量数据。
   # syncers:              # sync 处理单元的运行配置参数。
   #  global:             # 配置名称。
```

```
  #   safe-mode: true # 设置为 true，会将来自数据源的 INSERT 改写为 REPLACE，将 UPDATE 改写为
      ↪ DELETE 与 REPLACE，从而保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入
      ↪ DML。在启动或恢复增量复制任务的前 1 分钟内 TiDB DM 会自动启动 safe mode。
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍

3.3.1.3.4 第 3 步：启动任务

在你启动数据迁移任务之前，建议使用 check-task 命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率：

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 tiup dmctl 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后，再次执行上述命令，重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

3.3.1.3.5 第 4 步：查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 tiup dmctl 执行 query-status 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

3.3.1.3.6 第 5 步：监控任务与查看日志

要查看迁移任务的历史状态以及更多的内部运行指标，可参考以下步骤。

如果使用TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的IP 及端口进入Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、DM-master 及dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 --log-file 设置。如果使用TiUP 部署 DM，则日志目录默认位于 /dm-deploy/dm-master-8261/log/。
- DM-worker 日志目录：通过 DM-worker 进程参数 --log-file 设置。如果使用TiUP 部署 DM，则日志目录默认位于 /dm-deploy/dm-worker-8262/log/。

3.3.1.4 探索更多

- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句

3.3.2 从小数据量 MySQL 迁移数据到TiDB

本文档介绍如何使用TiDB DM（以下简称 DM）以全量 + 增量的模式数据到TiDB。本文所称“小数据量”通常指 TiB 级别以下。

一般而言，受到表结构索引数目等信息、硬件以及网络环境影响，迁移速率在 30 ~ 50GB/h 不等。使用TiDB DM 迁移的流程如下图所示。

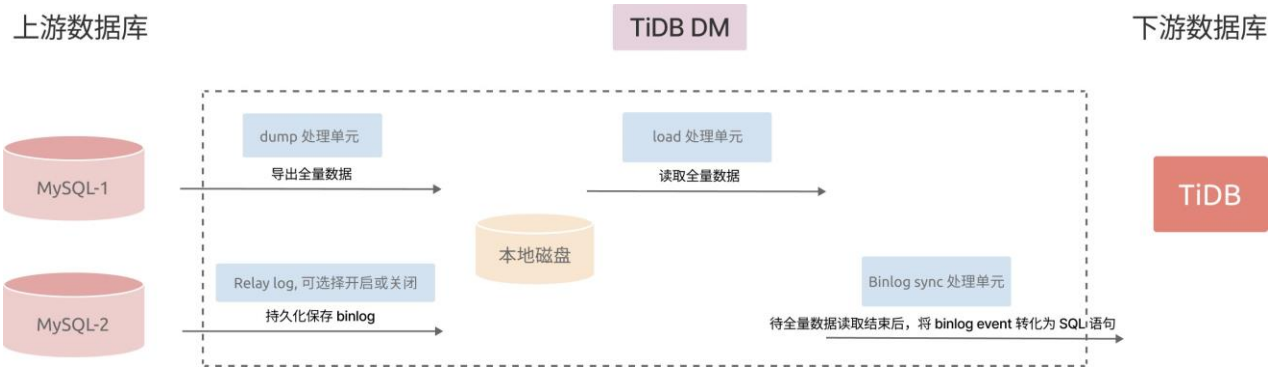


图 5: dm

3.3.2.1 前提条件

- 使用TiUP 安装 DM 集群
- DM 所需上下游数据库权限

3.3.2.2 第 1 步：创建数据源

首先，新建 source1.yaml 文件，写入以下内容：

```
### 唯一命名，不可重复。
source-id: "mysql-01"

### DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式。
    '→ 若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true

from:
```

```
  host: "${host}"          # 例如：172.16.10.81
  user: "root"
  password: "${password}" # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

其次，在终端中执行下面的命令后，使用 `tiup dmctl` 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
`--master-addr`	`dmctl` 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`，例如：172.16.10.71:8261
`operate-source create`	向 DM 集群加载数据源

3.3.2.3 第 2 步：创建迁移任务

新建 `task1.yaml` 文件，写入以下内容：

```
### 任务名，多个同时运行的任务不能重名。
name: "test"
### 任务模式，可设为
### full：只进行全量数据迁移
### incremental： binlog 实时同步
### all： 全量 + binlog 迁移
task-mode: "all"
### 下游 TiDB 配置信息。
target-database:
  host: "${host}"                  # 例如：172.16.10.83
  port: 4000
  user: "root"
  password: "${password}"          # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用


### 当前数据迁移任务需要的全部上游 MySQL 实例配置。
mysql-instances:
-
  # 上游实例或者复制组 ID。
  source-id: "mysql-01"
  # 需要迁移的库名或表名的黑白名单的配置项名称，用于引用全局的黑白名单配置，全局配置见下面的 `
      ↪ block-allow-list` 的配置。
  block-allow-list: "listA"



### 黑白名单全局配置，各实例通过配置项名引用。
```

```
block-allow-list:
  listA:                                # 名称
    do-tables:                          # 需要迁移的上游表的白名单。
    - db-name: "test_db"                # 需要迁移的表的库名。
      tbl-name: "test_table"            # 需要迁移的表的名称。
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

3.3.2.4 第 3 步：启动任务

在你启动数据迁移任务之前，建议使用 `check-task` 命令检查配置是否符合 DM 的配置要求，以避免后期报错。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 `tiup dmctl` 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--dmctl-master-addr	要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task task.yaml	用于启动数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

3.3.2.5 第 4 步：查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

3.3.2.6 第 5 步：监控任务与查看日志（可选）

要查看迁移任务的历史状态以及更多的内部运行指标，可参考以下步骤。

如果使用TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的IP 及端口进入Grafana，选择 DM 的 Dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、DM-master 及dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 `--log-file`设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
- DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

3.3.2.7 探索更多

- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句

3.3.3 从大数据量 MySQL 迁移数据到TiDB

通常数据量较低时，使用 DM 进行迁移较为简单，可直接完成全量+ 持续增量迁移工作。但当数据量较大时，DM 较低的数据导入速度(30~50 GiB/h) 可能令整个迁移周期过长。本文所称“大数据量”通常指TiB 级别以上。

因此，本文档介绍使用 Dumpling 和 TiDB Lightning 进行全量数据迁移，其本地导入(local backend) 模式导入速度可达每小时 500 GiB。完成全量数据迁移后，再使用 DM 完成增量数据迁移。

3.3.3.1 前提条件

- 部署 DM 集群。
- 安装 Dumpling 和 TiDB Lightning。
- 配置 DM 所需上下游数据库权限。
- 获取TiDB Lightning 所需下游数据库权限。
- 获取 Dumpling 所需上游数据库权限。

3.3.3.2 资源要求

操作系统：本文档示例使用的是若干新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将TiDB Lightning 和其他组件（比如 `tikv-server`）部署在同一台机器上，然后设置 `region-concurrency` 配置项的值为逻辑 CPU 数的 75%，以限制TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：因为 TiDB Lightning 对计算机资源消耗较高，建议分配 64 GB 以上的内存以及 32 核以上的 CPU，而且确保 CPU 核数和内存（GB）比为 1:2 以上，以获取最佳性能。

磁盘空间：

- Dumpling 需要能够储存整个数据源的存储空间，即可以容纳要导出的所有上游表的空间。计算方式参考下游数据库所需空间。
- TiDB Lightning 导入期间，需要临时空间来存储排序键值对，磁盘空间需要至少能存储数据源的最大单表。
- 若全量数据量较大，可适当加长上游binlog保存时间，以避免增量同步时缺必要binlog导致重做。

说明： 目前无法精确计算 Dumpling 从 MySQL 导出的数据大小， 但你可以用下面 SQL 语句统计信息表的 data_length 字段估算数据量：

```
/* 统计所有 schema 大小，单位 MiB，注意修改 ${schema_name} */
SELECT table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)/1024/1024 AS
    ↪ index_length,SUM(data_length+index_length)/1024/1024 AS SUM FROM information_schema.
    ↪ tables WHERE table_schema = "${schema_name}" GROUP BY table_schema;

/* 统计最大单表，单位 MiB，注意修改 ${schema_name} */
 SELECT table_name,table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)
    ↪ /1024/1024 AS index_length,SUM(data_length+index_length)/1024/1024 AS SUM from
    ↪ information_schema.tables WHERE table_schema = "${schema_name}" GROUP BY table_name,
    ↪ table_schema ORDER BY SUM DESC LIMIT 5;
```

3.3.3.2.1 目标TiKV 集群的磁盘空间要求

目标TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标TiKV 集群的总存储空间必须大于数据源大小× 副本数量× 2。例如，集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间。
- RocksDB 的空间放大效应。

3.3.3.3 第 1 步：从 MySQL 导出全量数据

1. 运行以下命令，从 MySQL 导出全量数据：

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MiB -B my_db1 -f 'my_db1.table
    ↪ [12]' -o 's3://my-bucket/sql-backup'
```

Dumpling 默认导出数据格式为 SQL 文件，你也可以通过设置 `--filetype` 指定导出文件的类型。

以上命令行中用到的参数描述如下。要了解更多 Dumpling 参数，请参考Dumpling 使用文档。

参数	说明
-u 或 --user	MySQL 数据库的用户
-p 或 --password	MySQL 数据库的用户密码
-P 或 --port	MySQL 数据库的端口
-h 或 --host	MySQL 数据库的 IP 地址
-t 或 --thread	导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜
-o 或 --output	存储导出文件的目录，支持本地文件路径或外部存储 URI 格式
-r 或 --row	用于指定单个文件的最大行数，指定该参数后 Dumpling 会开启表内并发加速导出，同时减少内存使用
-F	指定单个文件的最大大小，单位为 MiB。强烈建议使用 -F 参数以避免单表过大导致备份过程中断
-B 或 --database	导出指定数据库
-f 或 --filter	导出能匹配模式的表，语法可参考table-filter

请确保 ${data-path} 的空间可以容纳要导出的所有上游表，计算方式参考下游数据库所需空间。强烈建议使用 -F 参数以避免单表过大导致备份过程中断。

2. 查看在 ${data-path} 目录下的 metadata 文件，这是 Dumpling 自动生成的元信息文件，请记录其中的 binlog 位置信息，这将在第 3 步增量同步的时候使用。

```
SHOW MASTER STATUS:
Log: mysql-bin.000004
Pos: 109227
GTID:
```

3.3.3.4 第 2 步：导入全量数据到TiDB

1. 编写配置文件 tidb-lightning.toml：

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
# "tidb"：TB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-
    ↪ backends
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小
    ↪ 。建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO
    ↪ 会获得更好的导入性能
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
# 源数据目录，即第 1 步中 Dumpling 保存数据的路径。
```

```
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

[tidb]
# 目标集群的信息
host = ${host}                  # 例如：172.16.32.1
port = ${port}                  # 例如：4000
user = "${user_name}"           # 例如："root"
password = "${password}"        # 例如："rootroot"

status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”
    ↪ 获取表结构信息，例如：10080
pd-addr = "${ip}:${port}"       # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于更多TiDB Lightning的配置，请参考TiDB Lightning 配置参数。

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   若从Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和AccessKey 作为环境变量传入 Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

3. 导入开始后，可以采用以下任意方式查看进度：

   - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
   - 通过监控面板查看进度，请参考TiDB Lightning 监控。
   - 通过Web 页面查看进度，请参考Web 界面。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 tidb lightning exit。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

3.3.3.5 第 3 步：使用 DM 持续复制增量数据到TiDB

3.3.3.5.1 添加数据源

1. 新建 source1.yaml 文件，写入以下内容：

```
# 唯一命名，不可重复。
source-id: "mysql-01"

# DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式
    ↪ 。若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true


from:
  host: "${host}"           # 例如：172.16.10.81
  user: "root"
  password: "${password}"    # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

2. 在终端中执行下面的命令，使用 tiup dmctl 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
operate-source create	向 DM 集群加载数据源

3.3.3.5.2 添加同步任务

1. 编辑 task.yaml，配置增量同步模式，以及每个数据源的同步起点：

```
name: task-test                     # 任务名称，需要全局唯一。
task-mode: incremental              # 任务模式，设为 "incremental" 即只进行增量数据迁移。


# 配置下游 TiDB 数据库实例访问信息
target-database:                    # 下游数据库实例配置。
  host: "${host}"                   # 例如：127.0.0.1
  port: 4000
  user: "root"
  password: "${password}"
                                    # 推荐使用经过 dmctl 加密的密文。


#  使用黑白名单配置需要同步的表
block-allow-list:                   # 数据源数据库实例匹配的表的 block-allow-list
    ↪ 过滤规则集，如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  bw-rule-1:                        # 黑白名单配置项 ID。
```

```
    do-dbs: ["${db-name}"]           # 迁移哪些库。

# 配置数据源
mysql-instances:
  - source-id: "mysql-01"            # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "bw-rule-1"    # 引入上面黑白名单配置。
    # syncer-config-name: "global"    # 引用下面的 syncers 增量数据配置。
    meta:                            # `task-mode` 为 `incremental` 且下游数据库的 `
        '→ checkpoint` 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `
        '→ checkpoint` 为准。如果 `meta` 项和下游数据库的 `checkpoint` 都不存在，
        '→ 则从上游当前最新的 binlog 位置开始迁移。
      # binlog-name: "mysql-bin.000004" # 第 1 步中记录的日志位置，当上游存在主从切换时，
          '→ 必须使用 gtid。
      # binlog-pos: 109227
      binlog-gtid:  "09bec856-ba95-11ea-850a-58f2b4af5188:1-9"

# 【可选配置】如果增量数据迁移需要重复迁移已经在全量数据迁移中完成迁移的数据，则需要开启
    '→ safe mode 避免增量数据迁移报错。
# 该场景多见于以下情况：全量迁移的数据不属于数据源的一个一致性快照，
    '→ 随后从一个早于全量迁移数据之前的位置开始同步增量数据。
# syncers:              # sync 处理单元的运行配置参数。
#  global:             # 配置名称。
#    safe-mode: true # 设置为 true，会将来自数据源的 INSERT 改写为 REPLACE，将 UPDATE 改写为
    '→  DELETE 与 REPLACE，
    '→ 从而保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入 DML。
    '→ 在启动或恢复增量复制任务的前 1 分钟内 TiDB DM 会自动启动 safe mode。
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

在你启动数据迁移任务之前，建议使用check-task命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

2. 使用 tiup dmctl 执行以下命令启动数据迁移任务：

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更，再执行上述命令重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

3.3.3.5.3 查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

3.3.3.5.4 监控任务与查看日志

要查看迁移任务的历史状态以及更多的内部运行指标，可参考以下步骤。

如果使用TiUP 部署 DM 集群时，正确部署了 Prometheus、 Alertmanager 与 Grafana，则使用部署时填写的IP 及端口进入Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、 DM-master 及dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
- DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

3.3.3.6 探索更多

- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句

3.3.4 从小数据量分库分表 MySQL 合并迁移数据到TiDB

如果你想把上游多个 MySQL 数据库实例合并迁移到下游的同一个TiDB 数据库中，且数据量较小，你可以使用 DM 工具进行分库分表的合并迁移。本文所称“小数据量”通常指TiB 级别以下。本文举例介绍了合并迁移的操作步骤、注意事项、故障排查等。本文档适用于:

- TiB 级以内的分库分表数据合并迁移
- 基于 MySQL binlog 的增量、持续分库分表合并迁移

若要迁移分表总和 1 TiB 以上的数据，则 DM 工具耗时较长，可参考从大数据量分库分表 MySQL 合并迁移数据到 TiDB。

在本文档的示例中，数据源 MySQL 实例 1 和实例 2 均使用以下表结构，计划将 store_01 和 store_02 中 sale 开头的表合并导入下游 store.sale 表。

Schema	Tables
store_01	sale_01, sale_02
store_02	sale_01, sale_02

迁移目标库的结构如下：

Schema	Tables
store	sale

3.3.4.1 前提条件

- 使用 TiUP 安装 DM 集群
- DM 所需上下游数据库权限

3.3.4.2 分表数据冲突检查

迁移中如果涉及合库合表，来自多张分表的数据可能引发主键或唯一索引的数据冲突。因此在迁移之前，需要检查各分表数据的业务特点。详情请参考跨分表数据在主键或唯一索引冲突处理。

在本示例中：`sale_01` 和 `sale_02` 具有相同的表结构如下：

```
CREATE TABLE `sale_01` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

其中 `id` 列为主键，`sid` 列为分片键，具有全局唯一性。`id` 列具有自增属性，多个分表范围重复会引发数据冲突。`sid` 可以保证全局满足唯一索引，因此可以按照参考去掉自增主键的主键属性中介绍的操作绕过 `id` 列。在下游创建 `sale` 表时移除 `id` 列的唯一键属性：

```
CREATE TABLE `sale` (
  `id` bigint(20) NOT NULL,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  INDEX (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

3.3.4.3 第 1 步：创建数据源

新建 source1.yaml 文件，写入以下内容：

```
### 唯一命名，不可重复。
source-id: "mysql-01"

### DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式。
    '→ 若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true

from:
  host: "${host}" # 例如：172.16.10.81
  user: "root"
  password: "${password}" # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      '→ 对明文密码进行加密后使用
  port: 3306
```

在终端中执行下面的命令，使用 tiup dmctl 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
operate-source create	向 DM 集群加载数据源

重复以上操作直至所有数据源均添加完成。

3.3.4.4 第 2 步：创建迁移任务

新建task1.yaml文件，写入以下内容：

```
name: "shard_merge"
### 任务模式，可设为
### full：只进行全量数据迁移
### incremental： binlog 实时同步
### all： 全量 + binlog 迁移
task-mode: all
### 分库分表合并任务则需要配置 shard-mode。默认使用悲观协调模式 "pessimistic"，
    '→ 在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "optimistic"
### 详细信息可参考：https://docs.pingcap.com/zh/tidb/dev/feature-shard-merge/
shard-mode: "pessimistic"
meta-schema: "dm_meta"                          # 将在下游数据库创建 schema 用于存放元数据
ignore-checking-items: ["auto_increment_ID"]    # 本示例中上游存在自增主键，
    '→ 因此需要忽略掉该检查项
```

```
target-database:
  host: "${host}"                                  # 例如：192.168.0.1
  port: 4000
  user: "root"
  password: "${password}"                          # 支持但不推荐使用明文密码，建议使用 dmctl
      ↪ encrypt 对明文密码进行加密后使用

mysql-instances:
  -
    source-id: "mysql-01"                                    # 数据源 ID，即 source1.yaml 中的
        ↪ source-id
    route-rules: ["sale-route-rule"]                         # 应用于该数据源的 table route 规则
    filter-rules: ["store-filter-rule", "sale-filter-rule"] # 应用于该数据源的 binlog event
        ↪ filter 规则
    block-allow-list:  "log-bak-ignored"                     # 应用于该数据源的 Block & Allow
        ↪ Lists 规则
  -
    source-id: "mysql-02"
    route-rules: ["sale-route-rule"]
    filter-rules: ["store-filter-rule", "sale-filter-rule"]
    block-allow-list:  "log-bak-ignored"

### 分表合并配置
routes:
  sale-route-rule:
    schema-pattern: "store_*"                                # 合并 store_01 和 store_02 库到下游
        ↪ store 库
    table-pattern: "sale_*"                                  # 合并上述库中的 sale_01 和 sale_02
        ↪ 表到下游 sale 表
    target-schema: "store"
    target-table:  "sale"
    # 可选配置：提取各分库分表的源信息，并写入下游用户自建的列，用于标识合表中各行数据的来源。
        ↪ 如果配置该项，需要提前在下游手动创建合表，具体可参考下面 Table routing 的用法
    # extract-table:                                         # 提取分表去除 sale_ 的后缀信息，
        ↪ 并写入下游合表 c_table 列，例如，sale_01 分表的数据会提取 01 写入下游 c_table 列
    #   table-regexp: "sale_(.*)"
    #   target-column: "c_table"
     # extract-schema:                                        # 提取分库去除 store_ 的后缀信息，
         ↪ 并写入下游合表 c_schema 列，例如，store_02 分库的数据会提取 02 写入下游 c_schema 列
    #   schema-regexp: "store_(.*)"
    #   target-column: "c_schema"
    # extract-source:                                        # 提取数据库源实例信息写入 c_source
        ↪ 列，例如，mysql-01 数据源实例的数据会提取 mysql-01 写入下游 c_source 列
    #   source-regexp: "(.*)"
```

```
    #   target-column: "c_source"

### 过滤部分 DDL 事件
filters:
  sale-filter-rule:
    schema-pattern: "store_*"
    table-pattern: "sale_*"
    events: ["truncate table", "drop table", "delete"]
    action: Ignore
  store-filter-rule:
    schema-pattern: "store_*"
    events: ["drop database"]
    action: Ignore

### 黑白名单
block-allow-list:
  log-bak-ignored:
    do-dbs: ["store_*"]
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

若想了解配置文件中 routes、filters 等更多用法，请参考：

- Table routing
- Block & Allow Table Lists
- 如何过滤binlog事件
- 如何通过 SQL 表达式过滤 DML

3.3.4.5 第 3 步：启动任务

在你启动数据迁移任务之前，建议使用 check-task 命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 tiup dmctl 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。遇到

问题请参考故障及处理方法以及常见问题。

3.3.4.6 第 4 步：查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

3.3.4.7 第 5 步：监控任务与查看日志(可选)

你可以通过Grafana 或者日志查看迁移任务的历史状态以及各种内部运行指标。

- 通过 Grafana 查看

  如果使用TiUP 部署 DM 集群时，正确部署了 Prometheus、 Alertmanager 与 Grafana，则使用部署时填写的IP及端口进入Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

- 通过日志查看

  DM 在运行过程中，DM-worker、 DM-master 及 dmctl 都会通过日志输出相关信息，其中包含迁移任务的相关信息。各组件的日志目录如下：

  - DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
  - DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

3.3.4.8 探索更多

- 分库分表合并中的悲观/乐观模式
- 分表合并数据迁移最佳实践
- 故障及处理方法
- 性能问题及处理方法
- 常见问题

3.3.5 从大数据量分库分表 MySQL 合并迁移数据到TiDB

如果分表数据总规模特别大（例如大于 1 TiB ），并且允许 TiDB 集群在迁移期间无其他业务写入，那么你可以使用TiDB Lightning 对分表数据进行快速合并导入，然后根据业务需要选择是否使用TiDB DM 进行增量数据的分表同步。本文所称“大数据量”通常指TiB 级别以上。本文档举例介绍了导入数据的操作步骤。

如果分库分表合并迁移在 1 TiB 以内，请参考从小数据量分库分表 MySQL 合并迁移数据到TiDB，支持全量和增量且更为简单。

在本文的示例中，假设有两个数据库my_db1 和 my_db2，使用 Dumpling 分别从my_db1 中导出table1 和 table2 两个表，从my_db2 中导出table3 和 table4 两个表，然后再用TiDB Lightning 把导出的 4 个表合并导入到下游TiDB 中的同一个库my_db 的同一个表格table5 中。

本文将以三个步骤演示导入流程：

1. 使用 Dumpling 导出全量数据备份。在本文档示例中，分别从两个源数据库中各导出两个表：
    - 从实例 1 MySQL 的 my_db1 导出 table1、 table2
    - 从实例 2 MySQL 的 my_db2 导出 table3、 table4
2. 启动 TiDB Lightning 执行导入 TiDB 中的 mydb.table5
3. 使用 DM 进行增量数据迁移（可选）

3.3.5.1 前提条件

- 使用 TiUP 安装 DM 集群
- 使用 TiUP 安装 Dumpling 和 Lightning
- Dumpling 所需上游数据库权限
- TiDB Lightning 所需下游数据库权限
- TiDB Lightning 下游数据库所需空间
- DM 所需上下游数据库权限

3.3.5.1.1 分表数据冲突检查

迁移中如果涉及合库合表，来自多张分表的数据可能引发主键或唯一索引的数据冲突。因此在迁移之前，需要检查各分表数据的业务特点。详情请参考跨分表数据在主键或唯一索引冲突处理，这里做简要描述：

假设 table1~4 具有相同的表结构如下：

```
CREATE TABLE `table1` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

其中 `id` 列为主键，具有自增属性，多个分表范围重复会引发数据冲突。`sid` 列为分片键，可以保证全局满足唯一索引。因此可以移除下游 `table5` 表 `id` 列的唯一键属性：

```
CREATE TABLE `table5` (
  `id` bigint(20) NOT NULL,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  INDEX (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

3.3.5.2 第 1 步：用 Dumpling 导出全量数据备份

如果需要导出的多个分表属于同一个上游 MySQL 实例，建议直接使用 Dumpling 的 `-f` 参数一次导出多个分表的结果。如果多个分表分布在不同的 MySQL 实例，可以使用 Dumpling 分两次导出，并将两次导出的结果放置在相同的父目录下即可。下面的例子中同时用到了上述两种方式，然后将导出的数据存放在同一父目录下。

首先使用 Dumpling 从 my_db1 中导出表table1 和 table2，如下：

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MB -B my_db1 -f 'my_db1.table[12]' -
    ′→ o ${data-path}/my_db1
```

以上命令行中用到的参数描述如下。要了解更多 Dumpling 参数，请参考Dumpling 使用文档。

参数	描述
`-u` 或 `--user`	MySQL 数据库的用户
`-p` 或 `--password`	MySQL 数据库的用户密码
`-P` 或 `--port`	MySQL 数据库的端口
`-h` 或 `--host`	MySQL 数据库的 IP 地址
`-t` 或 `--thread`	导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜
`-o` 或 `--output`	存储导出文件的目录，支持本地文件路径或外部存储 URI 格式
`-r` 或 `--row`	用于指定单个文件的最大行数，指定该参数后 Dumpling 会开启表内并发加速导出，同时减少内存使用
`-F`	指定单个文件的最大大小，单位为 MiB。强烈建议使用`-F`参数以避免单表过大导致备份过程中断
`-B` 或 `--database`	导出指定数据库
`-f` 或 `--filter`	导出能匹配模式的表，语法可参考table-filter

然后使用同样的方式从 my_db2 中导出表 table3 和 table4。注意路径在相同 `${data-path}` 下的不同子目录 my_db2。

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MB -B my_db2 -f 'my_db2.table[34]' -
    ′→ o ${data-path}/my_db2
```

这样所需的全量备份数据就全部导出到了 `${data-path}` 目录中。将所有源数据表格存储在一个目录中，是为了后续方便用TiDB Lightning 导入。

第 3 步增量同步的时候所需的起始位点信息，在 `${data-path}` 目录下，my_db1 和 my_db2 的 `metadata` 文件中，这是 Dumpling 自动生成的元信息文件，请记录其中的binlog 位置信息。

3.3.5.3 第 2 步：启动TiDB Lightning 进行导入

在启动TiDB Lightning 进行迁移之前，建议先了解如何处理检查点，然后根据需要选择合适的方式进行迁移。

3.3.5.3.1 断点续传

大量数据导入一般耗时数小时甚至数天，长时间运行的进程会有一定机率发生非正常中断。如果每次重启都从头开始，之前已成功导入的数据就会前功尽弃。为此，TiDB Lightning 提供了断点续传的功能，即使TiDB Lightning 崩溃，在重启时仍然从断点开始继续工作。

若 TiDB Lightning 因不可恢复的错误而退出，例如数据出错，在重启时不会使用断点，而是直接报错离开。为保证已导入的数据安全，必须先解决掉这些错误才能继续。你可以使用 `tidb-lightning-ctl` 命令控制导入出错后的行为。该命令的选项有：

- –checkpoint-error-destroy：出现错误后，让失败的表从头开始整个导入过程。
- –checkpoint-error-ignore：如果导入表曾经出错，该命令会清除出错状态，如同错误没有发生过一样。
- –checkpoint-remove：无论是否有出错，把表的断点清除。

关于断点续传的更多信息，请参考TiDB Lightning 断点续传。

3.3.5.3.2 在下游创建 schema

在下游创建 `mydb.table5`。

```
CREATE TABLE `table5` (
  `id` bigint(20) NOT NULL,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  INDEX (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

3.3.5.3.3 执行导入操作

启动tidb-lightning的步骤如下：

1. 编写配置文件 `tidb-lightning.toml`。

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[mydumper]
data-source-dir = ${data-path}

[tikv-importer]
# "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
# "tidb"：TB 级以下数据量也可以采用`tidb`后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-
    ↪ backends
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小
    ↪ 。建议设为与 `data-source-dir` 不同的磁盘目录，独占 IO 会获得更好的导入性能
```

```
sorted-kv-dir = "${sorted-kv-dir}"

# 设置分库分表合并规则，将 my_db1 中的 table1、table2 两个表,以及 my_db2 中的 table3、table4
    ↪ 两个表，共计 2 个数据库中的 4 个表都导入到目的数据库 my_db 中的 table5 表中。
[[mydumper.files]]
pattern = '(^|/)my_db1\.table[1-2]\..*\.sql$'
schema = "my_db"
table = "table5"
type = "sql"

[[mydumper.files]]
pattern = '(^|/)my_db2\.table[3-4]\..*\.sql$'
schema = "my_db"
table = "table5"
type = "sql"

# 目标集群的信息，示例仅供参考。请把 IP 地址等信息替换成真实的信息。
[tidb]
# 目标集群的信息
host = ${host}              # 例如：172.16.32.1
port = ${port}              # 例如：4000
user = "${user_name}"       # 例如："root"
password = "${password}"    # 例如："rootroot"
status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息
    ↪ ，例如：10080
# PD 集群的地址，Lightning 通过 PD 获取部分信息。
pd-addr = "${ip}:${port}"    # 例如 172.16.31.3:2379。当 backend = "local" 时 status-port 和
    ↪ pd-addr 必须正确填写，否则导入将出现异常。
```

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   若从Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和AccessKey 作为环境变量传入 TiDB Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

3. 导入开始后，可以采用以下任意方式查看进度：

   - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
   - 通过监控面板查看进度，请参考TiDB Lightning 监控。
   - 通过Web 页面查看进度，请参考Web 界面。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

3.3.5.4 第 3 步：使用 DM 持续复制增量数据到TiDB (可选)

基于binlog从指定位置同步数据库到TiDB，可以使用 DM 来执行增量复制

3.3.5.4.1 添加数据源

新建 `source1.yaml` 文件，写入以下内容：

```
### 唯一命名，不可重复。
source-id: "mysql-01"

### DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式。
    ↪ 若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true


from:
  host: "${host}"           # 例如：172.16.10.81
  user: "root"
  password: "${password}"    # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

在终端中执行下面的命令，使用 `tiup dmctl` 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
`--master-addr`	dmctl 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`
`operate-source create`	向 DM 集群加载数据源

重复以上步骤直至所有 MySQL 实例被加入 DM。

3.3.5.4.2 添加同步任务

编辑 task.yaml，配置增量同步模式，以及每个数据源的同步起点：

```
name: task-test                 # 任务名称，需要全局唯一。
task-mode: incremental          # 任务模式，设为 "incremental" 即只进行增量数据迁移。
### 分库分表合并任务则需要配置 shard-mode。默认使用悲观协调模式 "pessimistic"，
    ′→ 在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "optimistic"
### 详细信息可参考：https://docs.pingcap.com/zh/tidb/dev/feature-shard-merge/
shard-mode: "pessimistic"

#### 配置下游 TiDB 数据库实例访问信息
target-database:                # 下游数据库实例配置。
  host: "${host}"               # 例如：127.0.0.1
  port: 4000
  user: "root"
  password: "${password}"       # 推荐使用经过 dmctl 加密的密文。

#### 使用黑白名单配置需要同步的表
block-allow-list:               # 数据源数据库实例匹配的表的 block-allow-list 过滤规则集，如果 DM
    ′→ 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  bw-rule-1:                    # 黑白名单配置项 ID。
    do-dbs: ["my_db1"]          # 迁移哪些库。这里将实例 1 的 my_db1 和实例 2 的 my_db2
        ′→ 分别配置为两条 rule，以示例如何避免实例 1 的 my_db2 被同步。
  bw-rule-2:
    do-dbs: ["my_db2"]

routes:                                 # 上游和下游表之间的路由 table routing 规则集
  route-rule-1:                         # 配置名称。将 my_db1 中的 table1 和 table2 合并导入下游
      ′→ my_db.table5
    schema-pattern: "my_db1"      # 库名匹配规则，支持通配符 "*" 和 "?"
    table-pattern: "table[1-2]"   # 表名匹配规则，支持通配符 "*" 和 "?"
    target-schema: "my_db"              # 目标库名称
    target-table: "table5"              # 目标表名称
  route-rule-2:                         # 配置名称。将 my_db2 中的 table3 和 table4 合并导入下游
      ′→        my_db.table5
    schema-pattern: "my_db2"
    table-pattern: "table[3-4]"
    target-schema: "my_db"
    target-table: "table5"

#### 配置数据源，以两个数据源为例
mysql-instances:
  - source-id: "mysql-01"               # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "bw-rule-1"       # 引入上面黑白名单配置。同步实例1的 my_db1
    route-rules: ["route-rule-1"]       # 引入上面表合并配置。
###       syncer-config-name: "global" # 引用后面的 syncers 增量数据配置。
```

```
    meta:                               # `task-mode` 为 `incremental` 且下游数据库的 `checkpoint`
        ′→ 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `checkpoint` 为准。如果 `
        ′→meta` 项和下游数据库的 `checkpoint` 都不存在，则从上游当前最新的 binlog 位置开始迁移
        ′→ 。
      binlog-name: "${binlog-name}"   # 第 1 步中 ${data-path}/my_db1/metadata 记录的日志位置，
          ′→ 当上游存在主从切换时，必须使用 gtid。
      binlog-pos: ${binlog-position}
      # binlog-gtid:                  " 例如：09bec856-ba95-11ea-850a-58f2b4af5188:1-9"
  - source-id: "mysql-02"             # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "bw-rule-2"     # 引入上面黑白名单配置。实例2的 my_db2
    route-rules: ["route-rule-2"]     # 引入上面表合并配置。

###       syncer-config-name: "global" # 引用后面的 syncers 增量数据配置。
    meta:                               # task-mode 为 incremental 且下游数据库的 checkpoint
        ′→ 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 checkpoint 为准。
      # binlog-name: "${binlog-name}"   # 第 1 步中 ${data-path}/my_db2/metadata 记录的日志位置，
          ′→ 当上游存在主从切换时，必须使用 gtid。
      # binlog-pos: ${binlog-position}
      binlog-gtid:  "09bec856-ba95-11ea-850a-58f2b4af5188:1-9"

#### 【可选配置】如果增量数据迁移需要重复迁移已经在全量数据迁移中完成迁移的数据，则需要开启 safe
    ′→  mode 避免增量数据迁移报错。
#### 该场景多见于以下情况：全量迁移的数据不属于数据源的一个一致性快照，
    ′→ 随后从一个早于全量迁移数据之前的位置开始同步增量数据。
### syncers:             # sync 处理单元的运行配置参数。
###  global:            # 配置名称。
###    safe-mode: true # 设置为 true，会将来自数据源的 INSERT 改写为 REPLACE，将 UPDATE 改写为
    ′→ DELETE 与 REPLACE，从而保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入 DML
    ′→ 。在启动或恢复增量复制任务的前 1 分钟内 TiDB DM 会自动启动 safe mode。
```

关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

在你启动数据迁移任务之前，建议使用 check-task 命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用tiup dmctl 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

3.3.5.4.3 查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

3.3.5.4.4 监控任务与查看日志

你可以通过Grafana 或者日志查看迁移任务的历史状态以及各种内部运行指标。

- 通过 Grafana 查看

  如果使用TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的IP及端口进入Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

- 通过日志查看

  DM 在运行过程中，DM-worker、DM-master 及 dmctl 都会通过日志输出相关信息，其中包含迁移任务的相关信息。各组件的日志目录如下：

  - DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
  - DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

3.3.5.5 探索更多

- 关于 Dumpling
- 关于 Lightning
- 分库分表合并中的悲观/乐观模式
- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句
- 故障及处理方法

3.3.6 从 CSV 文件迁移数据到TiDB

本文档介绍如何从CSV 文件迁移数据到TiDB。

TiDB Lightning 支持读取CSV 格式的文件，以及其他定界符格式，如TSV（制表符分隔值）。对于其他“平面文件”类型的数据导入，也可以参考本文档进行。

3.3.6.1 前提条件

- 安装TiDB Lightning。
- 获取 Lightning 所需下游数据库权限。

3.3.6.2 第 1 步：准备CSV 文件

将所有要导入的CSV 文件放在同一目录下，若要TiDB Lightning 识别所有CSV 文件，文件名必须满足以下格式：

- 包含整张表数据的CSV 文件，需命名为 `${db_name}.${table_name}.csv`。
- 如果一张表分布于多个CSV 文件，这些CSV 文件命名需加上文件编号的后缀，如`${db_name}.${table_name}.003.csv`。数字部分不需要连续，但必须递增，并且需要用零填充数字部分，保证后缀为同样长度。

3.3.6.3 第 2 步：创建目标表结构

CSV 文件自身未包含表结构信息。要将CSV 数据导入TiDB，就必须为数据提供表结构。可以通过以下任一方法创建表结构：

- 方法一：使用TiDB Lightning 创建表结构。

  编写包含 DDL 语句的 SQL 文件如下：

  - 文件名格式为 `${db_name}-schema-create.sql`，其内容需包含 `CREATE DATABASE` 语句。
  - 文件名格式为 `${db_name}.${table_name}-schema.sql`，其内容需包含 `CREATE TABLE` 语句。

- 方法二：手动在下游TiDB 建库和表。

3.3.6.4 第 3 步：编写配置文件

新建文件 `tidb-lightning.toml`，包含以下内容：

```
[lightning]
### 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
### "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
### "tidb"：TB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于导入模式更多信息请参阅：https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-
    ↪ overview#tidb-lightning-整体架构
backend = "local"
### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小，
    ↪ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO 会获得更好的导入性能
sorted-kv-dir = "/mnt/ssd/sorted-kv-dir"

[mydumper]
```

```
### 源数据目录。
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

### 定义 CSV 格式
[mydumper.csv]
### 字段分隔符，必须不为空。如果源文件中包含非字符串或数值类型的字段（如 binary，blob，bit 等），
    ↪ 则不建议源文件使用默认的“,”简单分隔符，推荐“|+|”等非常见字符组合
separator = ','
### 引用定界符，可以为零或多个字符。
delimiter = '"'
### CSV 文件是否包含表头。
### 如果为 true，则 lightning 会使用首行内容解析字段的对应关系。
header = true
### CSV 是否包含 NULL。
### 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
### 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
### 为以下值的字段将会被解析为 NULL。
null = '\N'
### 是否将字符串中包含的反斜杠（'\'）字符作为转义字符处理
backslash-escape = true
### 是否移除行尾的最后一个分隔符。
trim-last-separator = false


[tidb]
### 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"

status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息，
    ↪ 例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于配置文件更多信息，可参阅TiDB Lightning 配置参数。

3.3.6.5 第 4 步：导入性能优化（可选）

导入文件的大小统一约为 256 MiB 时，TiDB Lightning 可达到最佳工作状态。如果导入单个 CSV 大文件，TiDB Lightning 在默认配置下只能使用一个线程来处理，这会降低导入速度。

要解决此问题，可先将CSV 文件分割为多个文件。对于通用格式的CSV 文件，在没有读取整个文件的情况下，无法快速确定行的开始和结束位置。因此，默认情况下 TiDB Lightning 不会自动分割CSV 文件。但如果你确定待导入的CSV 文件符合特定的限制要求，则可以启用 `strict-format` 模式。启用后，TiDB Lightning 会将单个CSV

大文件分割为单个大小为 256 MiB 的多个文件块进行并行处理。

> 注意：
>
> 如果CSV 文件不是严格格式，但 `strict-format` 被误设为 `true`，跨多行的单个完整字段会被分割成两部分，导致解析失败，甚至不报错地导入已损坏的数据。

严格格式的CSV 文件中，每个字段仅占一行，即必须满足以下条件之一：

- delimiter 为空；
- 每个字段不包含 CR（\r）或 LF（\n）。

如果你确认满足条件，可按如下配置开启 `strict-format` 模式以加快导入速度。

```
[mydumper]
strict-format = true
```

3.3.6.6 第 5 步：执行导入

运行 `tidb-lightning`。如果直接在命令行中启动程序，可能会因为 `SIGHUP` 信号而退出，建议配合 `nohup` 或 `screen` 等工具，如：

```
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

导入开始后，可以采用以下任意方式查看进度：

- 通过 `grep` 日志关键字 `progress` 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，请参考TiDB Lightning 监控。
- 通过Web 页面查看进度，请参考Web 界面。

导入完毕后，TiDB Lightning 会自动退出。查看 `tidb-lightning.log` 日志末尾是否有 `the whole procedure ′→ completed` 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> 注意：
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

3.3.6.7 其他格式的文件

若数据源为其他格式，除文件名仍必须以 `.csv` 结尾外，配置文件 `tidb-lightning.toml` 的 `[mydumper.csv]` 格式定义同样需要做相应修改。常见的格式修改如下：

TSV：

```
### 格式示例
### ID    Region    Count
### 1     East      32
### 2     South     NULL
### 3     West      10
### 4     North     39

### 格式配置
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
null = 'NULL'
backslash-escape = false
trim-last-separator = false
```

TPC-H DBGEN：

```
### 格式示例
### 1|East|32|
### 2|South|0|
### 3|West|10|
### 4|North|39|

### 格式配置
[mydumper.csv]
separator = '|'
delimiter = ''
header = false
not-null = true
backslash-escape = false
trim-last-separator = true
```

3.3.6.8 探索更多

- CSV 支持与限制

3.3.7 从 SQL 文件迁移数据到TiDB

本文介绍如何使用 TiDB Lightning 从 MySQL SQL 文件迁移数据到TiDB。关于如何生成 MySQL SQL 文件，请参考 Dumpling 文档中的导出为 SQL 文件。

3.3.7.1 前提条件

- 使用TiUP 安装TiDB Lightning
- Lightning 所需下游数据库权限

3.3.7.2 第 1 步：准备 SQL 文件

将所有 SQL 文件放到统一目录下，例如 `/data/my_datasource/` 或 `s3://my-bucket/sql-backup`。Lightning 将递归地寻找该目录下及其子目录内的所有 `.sql` 文件。

3.3.7.3 第 2 步：定义目标表结构

要导入TiDB，就必须为 SQL 文件提供表结构。

如果使用 Dumpling 工具导出数据，则会自动导出表结构文件。此外，其他方式导出的数据可以通过以下任一方法创建表结构：

- 方法一：使用TiDB Lightning 创建表结构。
  编写包含 DDL 语句的 SQL 文件：
  - 文件名格式为 `${db_name}-schema-create.sql`，其内容需包含 `CREATE DATABASE` 语句。
  - 文件名格式为 `${db_name}.${table_name}-schema.sql`，其内容需包含 `CREATE TABLE` 语句。
- 方法二：手动在下游TiDB 建库和表。

3.3.7.4 第 3 步：编写配置文件

新建文件 `tidb-lightning.toml`，包含以下内容：

```
[lightning]
### 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
### "local"：默认使用该模式，适用于 TiB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
backend = "local"
### # "tidb"：TiB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ′→ 关于后端模式更多信息请参考 https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-
    ′→ backends 。
### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。
    ′→ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 I/O 会获得更好的导入性能
    ′→ 。
```

```
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
### 源数据目录
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'


[tidb]
### 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"

status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息，
    ′→ 例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ′→ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ′→ 否则导入将出现异常。
```

关于配置文件更多信息，可参阅TiDB Lightning Configuration。

3.3.7.5 第 4 步：执行导入

运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具。

若从 Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

同时，TiDB Lightning 还支持从 ~/.aws/credentials 读取凭证文件。

导入开始后，可以采用以下任意方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，请参考TiDB Lightning 监控。
- 通过Web 页面查看进度，请参考Web 界面。

导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure ′→ completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

3.3.8 从 Parquet 文件迁移数据到TiDB

本文介绍如何从Apache Hive 中生成 Parquet 文件以及如何使用TiDB Lightning 从 Parquet 文件迁移数据到TiDB。

如果你从Amazon Aurora 中导出 Parquet 文件，请参照从 Amazon Aurora 迁移数据到TiDB。

3.3.8.1 前提条件

- 使用TiUP 安装TiDB Lightning
- 获取TiDB Lightning 所需下游数据库权限

3.3.8.2 第 1 步：准备 Parquet 文件

本节描述如何从 Hive 中导出能被TiDB Lightning 读取的 Parquet 文件。

Hive 中每个表都能通过标注 `STORED AS PARQUET LOCATION '/path/in/hdfs'` 的形式将表数据导出到 Parquet 文件中。因此，如果你需要导出一张名叫 `test` 的表，请执行以下步骤：

1. 在 Hive 中执行如下 SQL 语句：

    ```
    CREATE TABLE temp STORED AS PARQUET LOCATION '/path/in/hdfs'
    AS SELECT * FROM test;
    ```

    执行上述语句后，表数据就成功导出到 HDFS 系统里。

2. 使用 `hdfs dfs -get` 命令将 Parquet 文件导出到本地：

    ```
    hdfs dfs -get /path/in/hdfs /path/in/local
    ```

    完成导出后，如果你需要将 HDFS 里导出的 Parquet 文件删除，可以直接将这个临时表(`temp`) 删掉：

    ```
    DROP TABLE temp;
    ```

3. 从 Hive 导出的 Parquet 文件可能不带有 `.parquet` 的后缀，无法被TiDB Lightning 正确识别。因此，在进行导入之前，需要对导出的文件进行重命名，添加 `.parquet` 后缀。

4. 将所有 Parquet 文件放到统一目录下，例如 `/data/my_datasource/` 或 `s3://my-bucket/sql-backup`。TiDB Lightning 将递归地寻找该目录及其子目录内的所有 `.parquet` 文件。

3.3.8.3 第 2 步：创建目标表结构

在将 Parquet 文件导入TiDB 前，你必须为 Parquet 文件提供表结构。你可以通过以下任一方法创建表结构：

- 方法一：使用TiDB Lightning 创建表结构。

  编写包含 DDL 语句的 SQL 文件：

  - 文件名格式为 `${db_name}-schema-create.sql`，其内容需包含 `CREATE DATABASE` 语句。
  - 文件名格式为 `${db_name}.${table_name}-schema.sql`，其内容需包含 `CREATE TABLE` 语句。

- 方法二：手动在下游TiDB 建库和表。

3.3.8.4 第 3 步：编写配置文件

新建文件 `tidb-lightning.toml`，包含以下内容：

```
[lightning]
### 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
### "local"：默认使用该模式，适用于 TiB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
backend = "local"
### # "tidb"：TiB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ′→ 关于导入模式更多信息请参阅：https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-
    ′→ overview#tidb-lightning-整体架构
### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。
    ′→ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 I/O 会获得更好的导入性能
    ′→ 。
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
### 源数据目录
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'


[tidb]
### 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"

status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息，
    ′→ 例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ′→ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ′→ 否则导入将出现异常。
```

关于配置文件更多信息，可参阅TiDB Lightning 配置参数。

3.3.8.5 第 4 步：执行导入

1. 运行 tidb-lightning。

    - 如果从Amazon S3 导入，需先将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。

        ```
        export AWS_ACCESS_KEY_ID=${access_key}
        export AWS_SECRET_ACCESS_KEY=${secret_key}
        ```

        此外，TiDB Lightning 还支持从 ~/.aws/credentials 读取凭证文件。
    - 如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具运行 tidb-lightning：

        ```
        nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
        ```

2. 导入开始后，可以采用以下任意方式查看进度：

    - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
    - 通过监控面板查看进度，请参考TiDB Lightning 监控。
    - 通过Web 页面查看进度，请参考Web 界面。

    导入完毕后，TiDB Lightning 会自动退出。

3. 检查导入是否成功。

    查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> 注意：
> 无论导入成功与否，最后一行都会显示 tidb lightning exit。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

3.3.9 从 TiDB 集群迁移数据至另一TiDB 集群

本文档介绍如何将数据从一个TiDB 集群迁移至另一TiDB。在如下场景中，你可以将数据从一个TiDB 集群迁移至另一个TiDB 集群：

- 拆库：原 TiDB 集群体量过大，或者为了避免原有的 TiDB 集群所承载的数个业务之间互相影响，将原 TiDB 集群中的部分表迁到另一个TiDB 集群。
- 迁库：是对数据库的物理位置进行迁移，比如更换数据中心。

- 升级：在对数据正确性要求严苛的场景下，可以将数据迁移到一个更高版本的TiDB 集群，确保数据安全。

本文将模拟整个迁移过程，具体包括以下四个步骤：

1. 搭建环境
2. 迁移全量数据
3. 迁移增量数据
4. 平滑切换业务

3.3.9.1 第 1 步：搭建环境

1. 部署集群。

   使用TiUP Playground 快速部署上下游测试集群。更多部署信息，请参考TiUP 官方文档。

   ```
   # 创建上游集群
   tiup --tag upstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
   # 创建下游集群
   tiup --tag downstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
   # 查看集群状态
   tiup status
   ```

2. 初始化数据。

   测试集群中默认创建了test 数据库，因此可以使用 sysbench 工具生成测试数据，用以模拟真实集群中的历史数据。

   ```
   sysbench oltp_write_only --config-file=./tidb-config --tables=10 --table-size=10000 prepare
   ```

   这里通过 sysbench 运行 oltp_write_only 脚本，其将在测试数据库中生成 10 张表，每张表包含 10000 行初始数据。tidb-config 的配置如下：

   ```
   mysql-host=172.16.6.122 # 这里需要替换为实际上游集群 ip
   mysql-port=4000
   mysql-user=root
   mysql-password=
   db-driver=mysql          # 设置数据库驱动为 mysql
   mysql-db=test            # 设置测试数据库为 test
   report-interval=10       # 设置定期统计的时间间隔为 10 秒
   threads=10               # 设置 worker 线程数量为 10
   time=0                   # 设置脚本总执行时间，0 表示不限制
   rate=100                 # 设置平均事务速率 tps = 100
   ```

3. 模拟业务负载。

   实际生产集群的数据迁移过程中，通常原集群还会写入新的业务数据，本文中可以通过 sysbench 工具模拟持续的写入负载，下面的命令会使用 10 个 worker 在数据库中的 sbtest1、sbtest2 和 sbtest3 三张表中持续写入数据，其总tps 限制为 100。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=3 run
```

4. 准备外部存储。

   在全量数据备份中，上下游集群均需访问备份文件，因此推荐使用备份存储存储备份文件，本文中通过 Minio 模拟兼容 S3 的存储服务：

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
# 配置访问 minio 的 access-key access-screct-id
export HOST_IP='172.16.6.122' # 替换为实际上游集群 ip
export MINIO_ROOT_USER='minio'
export MINIO_ROOT_PASSWORD='miniostorage'
# 创建数据目录， 其中 backup 为 bucket 的名称
mkdir -p data/backup
# 启动 minio，暴露端口在 6060
./minio server ./data --address :6060 &
```

   上述命令行启动了一个单节点的minio server 模拟 S3 服务，其相关参数为：

   - Endpoint: http://$%7BHOST_IP%7D:6060/
   - Access-key: minio
   - Secret-access-key: miniostorage
   - Bucket: backup

   相应的访问链接为：

```
s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=http://${HOST_IP}:6060&
    ↪ force-path-style=true
```

3.3.9.2 第 2 步：迁移全量数据

搭建好测试环境后，可以使用 BR 工具的备份和恢复功能迁移全量数据。BR 工具有多种使用方式，本文中使用 SQL 语句 BACKUP 和 RESTORE 进行备份恢复。

> 注意：
>
> - 在生产集群中，关闭GC 机制和备份操作会一定程度上降低集群的读性能，建议在业务低峰期进行备份，并设置合适的 RATE_LIMIT 限制备份操作对线上业务的影响。
> - 上下游集群版本不一致时，应检查 BR 工具的兼容性。本文假设上下游集群版本相同。

1. 关闭GC。

   为了保证增量迁移过程中新写入的数据不丢失，在开始备份之前，需要关闭上游集群的垃圾回收(GC)机制，以确保系统不再清理历史数据。

执行如下命令关闭GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=FALSE;
```

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

查询 tidb_gc_enable 的取值，判断GC是否已关闭：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------------------------+:
| @@global.tidb_gc_enable |
+-------------------------------------------+
|                         0 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

2. 备份数据。

   在上游集群中执行 BACKUP 语句备份数据：

```
MySQL [(none)]> BACKUP DATABASE * TO 's3://backup?access-key=minio&secret-access-key=
    '→ miniostorage&endpoint=http://${HOST_IP}:6060&force-path-style=true' RATE_LIMIT = 120
    '→  MB/SECOND;
```

```
+---------------+----------+--------------------+--------------------+---------------------------------------+
    '→
| Destination   | Size     | BackupTS           | Queue Time         | Execution Time
    '→ |
+---------------+----------+--------------------+--------------------+---------------------------------------+
    '→
| s3://backup   | 10315858 | 431434047157698561 | 2022-02-25 19:57:59 | 2022-02-25 19:57:59
    '→ |
+---------------+----------+--------------------+--------------------+---------------------------------------+
    '→
1 row in set (2.11 sec)
```

   备份语句提交成功后，TiDB 会返回关于备份数据的元信息，这里需要重点关注 BackupTS，它意味着该时间点之前数据会被备份，后边的教程中，本文将使用 BackupTS 作为数据校验截止时间和TiCDC 增量扫描的开始时间。

3. 恢复数据。

   在下游集群中执行RESTORE 语句恢复数据：

```
mysql> RESTORE DATABASE * FROM 's3://backup?access-key=minio&secret-access-key=miniostorage&
    '→ endpoint=http://${HOST_IP}:6060&force-path-style=true';
```

```
+--------------+-----------+--------------------+--------------------+----------------------------------------+
    '→
| Destination  | Size      | BackupTS           | Queue Time         | Execution Time
    '→ |
+--------------+-----------+--------------------+--------------------+----------------------------------------+
    '→
| s3://backup  | 10315858  | 431434141450371074 | 2022-02-25 20:03:59 | 2022-02-25 20:03:59
    '→ |
+--------------+-----------+--------------------+--------------------+----------------------------------------+
    '→
1 row in set (41.85 sec)
```

4. （可选）校验数据。

   通过sync-diff-inspector 工具，可以验证上下游数据在某个时间点的一致性。从上述备份和恢复命令的输出可以看到，上游集群备份的时间点为 431434047157698561，下游集群完成数据恢复的时间点为 431434141450371074。

```
sync_diff_inspector -C ./config.yaml
```

```
# Diff Configuration.
######################### Datasource config #########################
[data-sources]
[data-sources.upstream]
        host = "172.16.6.122" # 需要替换为实际上游集群 ip
        port = 4000
        user = "root"
        password = ""
        snapshot = "431434047157698561" # 配置为实际的备份时间点（参见「备份」小节的
            '→ BackupTS）
[data-sources.downstream]
        host = "172.16.6.125" # 需要替换为实际下游集群 ip
        port = 4000
        user = "root"
        password = ""

######################### Task config #########################
[task]
        output-dir = "./output"
        source-instances = ["upstream"]
        target-instance = "downstream"
        target-check-tables = ["*.*"]
```

   关于 sync-diff-inspector 的配置方法，请参考配置文件说明，在本文中，相应的配置如下：

3.3.9.3 第 3 步：迁移增量数据

1. 部署TiCDC。

   完成全量数据迁移后，就可以部署并配置TiCDC 集群同步增量数据，实际生产集群中请参考TiCDC 部署。本文在创建测试集群时，已经启动了一个TiCDC 节点，因此可以直接进行changefeed 的配置。

2. 创建同步任务。

   在上游集群中，执行以下命令创建从上游到下游集群的同步链路：

   ```
   tiup cdc cli changefeed create --server=http://172.16.6.122:8300 --sink-uri="mysql://root:
       ′→ @172.16.6.125:4000" --changefeed-id="upstream-to-downstream" --start-ts
       ′→ ="431434047157698561"
   ```

   以上命令中：

   - `--server`：TiCDC 集群中任意一个节点的地址
   - `--sink-uri`：同步任务下游的地址
   - `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 [1]+(-[a-zA-Z0-9]+)*$
   - `--start-ts`：TiCDC 同步的起点，需要设置为实际的备份时间点，也就是第 2 步：迁移全量数据中“备份数据”提到的 BackupTS

   更多关于changefeed 的配置，请参考TiCDC Changefeed 配置参数。

3. 重新开启GC。

   TiCDC 可以保证GC 只回收已经同步的历史数据。因此，创建完从上游到下游集群的changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。详情请参考TiCDC GC safepoint 的完整行为。

执行如下命令打开GC：

```
```sql
MySQL [test]> SET GLOBAL tidb_gc_enable=TRUE;
```


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

查询 `tidb_gc_enable` 的取值，判断 GC 是否已开启：

```sql
MySQL [test]> SELECT @@global.tidb_gc_enable;
```


```
+-------------------------------------------+
| @@global.tidb_gc_enable |
```

[1] a-zA-Z0-9
```

```
+----------------------------------------+
|                                      1 |
+----------------------------------------+
1 row in set (0.00 sec)
```

3.3.9.4 第 4 步：平滑切换业务

通过 TiCDC 创建上下游的同步链路后，原集群的写入数据会以非常低的延迟同步到新集群，此时可以逐步将读流量迁移到新集群了。观察一段时间，如果新集群表现稳定，就可以将写流量接入新集群，步骤如下：

1. 停止上游集群的写业务。确认上游数据已全部同步到下游后，停止上游到下游集群的changefeed。

```
# 停止旧集群到新集群的 changefeed
tiup cdc cli changefeed pause -c "upstream-to-downstream" --server=http://172.16.6.122:8300

# 查看 changefeed 状态
tiup cdc cli changefeed list
```

```
[
  {
    "id": "upstream-to-downstream",
    "summary": {
    "state": "stopped", # 需要确认这里的状态为 stopped
    "tso": 431747241184329729,
    "checkpoint": "2022-03-11 15:50:20.387", # 确认这里的时间晚于停写的时间
    "error": null
    }
  }
]
```

2. 创建下游到上游集群的changefeed。由于此时上下游数据是一致的，且没有新数据写入，因此可以不指定 start-ts，默认为当前时间：

```
tiup cdc cli changefeed create --server=http://172.16.6.125:8300 --sink-uri="mysql://root:
    ⟶ @172.16.6.122:4000" --changefeed-id="downstream -to-upstream"
```

3. 将写业务迁移到下游集群，观察一段时间后，等新集群表现稳定，便可以弃用原集群。

3.3.10 从 TiDB 集群迁移数据至兼容 MySQL 的数据库

本文档介绍如何将数据从TiDB 集群迁移至兼容 MySQL 的数据库，如Aurora、 MySQL、 MariaDB 等。本文将模拟整个迁移过程，具体包括以下四个步骤：

1. 搭建环境

2. 迁移全量数据
3. 迁移增量数据
4. 平滑切换业务

3.3.10.1 第 1 步：搭建环境

1. 部署上游TiDB 集群。

   使用TiUP Playground 快速部署上下游测试集群。更多部署信息，请参考TiUP 官方文档。

   ```
   # 创建上游集群
   tiup playground --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
   # 查看集群状态
   tiup status
   ```

2. 部署下游 MySQL 实例。

   - 在实验环境中，可以使用 Docker 快速部署 MySQL 实例，执行如下命令：

     ```
     docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -p 3306:3306 -d mysql
     ```

   - 在生产环境中，可以参考Installing MySQL 来部署 MySQL 实例。

3. 模拟业务负载。

   在测试实验环境下，可以使用 go-tpc 向上游TiDB 集群写入数据，以让TiDB 产生事件变更数据。执行如下命令，将首先在上游TiDB 创建名为tpcc 的数据库，然后使用TiUP bench 写入数据到刚创建的tpcc 数据库中。

   ```
   tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 prepare
   tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 run --time 300s
   ```

   关于 go-tpc 的更多详细内容，可以参考如何对TiDB 进行 TPC-C 测试。

3.3.10.2 第 2 步：迁移全量数据

搭建好测试环境后，可以使用Dumpling 工具导出上游集群的全量数据。

> **注意：**
>
> 在生产集群中，关闭 GC 机制和备份操作会一定程度上降低集群的读性能，建议在业务低峰期进行备份，并设置合适的 `RATE_LIMIT` 限制备份操作对线上业务的影响。

1. 关闭 GC (Garbage Collection)。

   为了保证增量迁移过程中新写入的数据不丢失，在开始全量导出之前，需要关闭上游集群的垃圾回收 (GC) 机制，以确保系统不再清理历史数据。

   执行如下命令关闭GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=FALSE;
```

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

查询 tidb_gc_enable 的取值，判断GC是否已关闭：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------------------------+:
| @@global.tidb_gc_enable |
+-------------------------------------------+
|                         0 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

2. 备份数据。

    1. 使用 Dumpling 导出 SQL 格式的数据：

    ```
    tiup dumpling -u root -P 4000 -h 127.0.0.1 --filetype sql -t 8 -o ./dumpling_output -r
        '→ 200000 -F256MiB
    ```

    2. 导出完毕后，执行如下命令查看导出数据的元信息，metadata 文件中的 Pos 就是导出快照的TSO，将其记录为 BackupTS：

    ```
    cat dumpling_output/metadata
    ```

    ```
    Started dump at: 2022-06-28 17:49:54
    SHOW MASTER STATUS:
            Log: tidb-binlog
            Pos: 434217889191428107
            GTID:

    Finished dump at: 2022-06-28 17:49:57
    ```

3. 恢复数据。

    使用开源工具 MyLoader 导入数据到下游 MySQL。MyLoader 的安装和详细用例参见 MyDumpler/MyLoader。执行以下指令，将 Dumpling 导出的上游全量数据导入到下游 MySQL 实例：

    ```
    myloader -h 127.0.0.1 -P 3306 -d ./dumpling_output/
    ```

4. （可选）校验数据。

    通过sync-diff-inspector 工具，可以验证上下游数据在某个时间点的一致性。

    ```
    sync_diff_inspector -C ./config.yaml
    ```

关于 sync-diff-inspector 的配置方法，请参考配置文件说明。在本文中，相应的配置如下：

```
# Diff Configuration.
######################### Datasource config #########################
[data-sources]
[data-sources.upstream]
        host = "127.0.0.1" # 需要替换为实际上游集群 ip
        port = 4000
        user = "root"
        password = ""
        snapshot = "434217889191428107" # 配置为实际的备份时间点（参见「备份」小节的
            ↪ BackupTS）
[data-sources.downstream]
        host = "127.0.0.1" # 需要替换为实际下游集群 ip
        port = 3306
        user = "root"
        password = ""

######################### Task config #########################
[task]
        output-dir = "./output"
        source-instances = ["upstream"]
        target-instance = "downstream"
        target-check-tables = ["*.*"]
```

3.3.10.3 第 3 步：迁移增量数据

1. 部署TiCDC。

   完成全量数据迁移后，就可以部署并配置TiCDC 集群同步增量数据，实际生产集群中请参考TiCDC 部署。本文在创建测试集群时，已经启动了一个TiCDC 节点，因此可以直接进行changefeed 的配置。

2. 创建同步任务。

   在上游集群中，执行以下命令创建从上游到下游集群的同步链路：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server=http://127.0.0.1:8300 --sink-uri
    ↪ ="mysql://root:@127.0.0.1:3306" --changefeed-id="upstream-to-downstream" --start-ts
    ↪ ="434217889191428107"
```

   以上命令中：

   - `--server`：TiCDC 集群任意一节点地址
   - `--sink-uri`：同步任务下游的地址
   - `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`
   - `--start-ts`：TiCDC 同步的起点，需要设置为实际的备份时间点，也就是第 2 步：迁移全量数据中“备份数据”提到的 BackupTS

更多关于changefeed 的配置，请参考TiCDC Changefeed 配置参数。

3. 重新开启GC。

   TiCDC 可以保证GC 只回收已经同步的历史数据。因此，创建完从上游到下游集群的changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。详情请参考TiCDC GC safepoint 的完整行为。

   执行如下命令打开GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=TRUE;
```

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

   查询 tidb_gc_enable 的取值，判断GC 是否已开启：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------------------+
| @@global.tidb_gc_enable |
+-------------------------------------+
|                       1 |
+-------------------------------------+
1 row in set (0.00 sec)
```

3.3.10.4 第 4 步：平滑切换业务

通过TiCDC 创建上下游的同步链路后，原集群的写入数据会以非常低的延迟同步到新集群，此时可以逐步将读流量迁移到新集群了。观察一段时间，如果新集群表现稳定，就可以将写流量接入新集群，步骤如下：

1. 停止上游集群的写业务。确认上游数据已全部同步到下游后，停止上游到下游集群的changefeed。

```
# 停止旧集群到新集群的 changefeed
tiup cdc cli changefeed pause -c "upstream-to-downstream" --pd=http://172.16.6.122:2379

# 查看 changefeed 状态
tiup cdc cli changefeed list
```

```
[
  {
    "id": "upstream-to-downstream",
    "summary": {
    "state": "stopped", # 需要确认这里的状态为 stopped
    "tso": 434218657561968641,
    "checkpoint": "2022-06-28 18:38:45.685", # 确认这里的时间晚于停写的时间
    "error": null
    }
  }
]
```

2. 将写业务迁移到下游集群，观察一段时间后，等新集群表现稳定，便可以弃用原集群。

3.4 复杂迁移场景

3.4.1 上游使用pt-osc/gh-ost 工具的持续同步场景

在生产业务中执行 DDL 时，产生的锁表操作会一定程度阻塞数据库的读取或者写入。为了把对读写的影响降到最低，用户往往会选择 online DDL 工具执行 DDL。常见的 Online DDL 工具有 gh-ost 和 pt-osc。

在使用 DM 完成 MySQL 到 TiDB 的数据迁移时，可以开启`online-ddl`配置，实现 DM 工具与 gh-ost 或 pt-osc 的协同。

具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到TiDB
- 从大数据量 MySQL 迁移数据到TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到TiDB

3.4.1.1 开启 DM 的 online-ddl 特性

配置 DM 的任务配置文件时，将全局参数的`online-ddl`设置为 true，具体配置示例如下图：

```
### ------------ 全局配置 ------------
#### ********* 基本信息配置 *********
name: test                          # 任务名称，需要全局唯一
task-mode: all                      # 任务模式，可设为 "full"、"incremental"、"all"
shard-mode: "pessimistic"           # 如果为分库分表合并任务则需要配置该项。默认使用悲观协调模式 "
    ↪ pessimistic"，在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "
    ↪ optimistic"
meta-schema: "dm_meta"              # 下游储存 `meta` 信息的数据库
online-ddl: true                    # 开启 DM 的 online DDL 支持特性。兼容上游使用 gh-ost 、 pt-osc
    ↪ 两种工具的自动处理
```

3.4.1.2 开启 online-ddl 的影响

当开启`online-ddl`特性后，DM 同步 gh-ost 或 pt-osc 工具所产生的 DDL 语句将会发生一些变化：

上游 gh-ost 或 pt-osc 工具工作流如下：

- 根据 DDL 目标表(real table)的表结构新建一张镜像表(ghost table)；
- 在镜像表上应用变更 DDL；
- 将 DDL 目标表的数据同步到镜像表；
- 在目标表与镜像表数据一致后，通过RENAME 语句使镜像表替换掉目标表。

`online-ddl=true` 时 DM 的同步方式：

- 不在下游新建镜像表(ghost table)；
- 记录变更 DDL；

- 仅从镜像表同步数据；
- 在下游执行 DDL 变更。

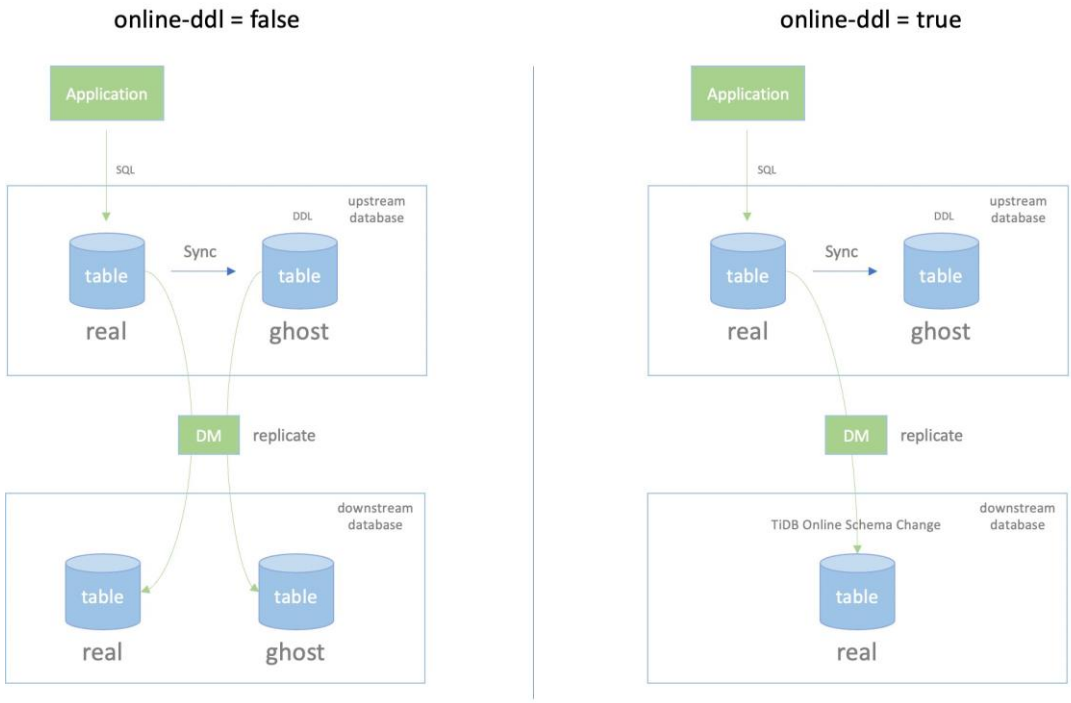


图 6: dm-online-ddl

这些变化将带来一些好处：

- 下游TiDB 无需创建和同步镜像表，节约相应存储空间和网络传输等开销；
- 在分库分表合并场景下，自动忽略各分表镜像表的RENAME 操作，保证同步正确性。

如果您想深入了解其实现原理，请阅读以下两篇技术博客：

- DM 源码阅读系列文章（八）Online Schema Change 迁移支持
- TiDB Online Schema Change 原理

3.4.1.3 探索更多

- DM 与 online DDL 工具协作细节

3.4.2 下游存在更多列的迁移场景

本文档介绍数据同步时，下游存在更多列的迁移场景需要的注意事项。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到TiDB
- 从大数据量 MySQL 迁移数据到TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到TiDB

3.4.2.1 使用 DM 迁移至存在更多列的下游

DM 同步上游的binlog 时，会尝试使用下游当前的表结构来解析binlog 并生成相应的 DML 语句。如果上游的 binlog 里数据表的列数与下游表结构的列数不一致，则会产生如下错误：

```
"errors": [
    {
        "ErrCode": 36027,
        "ErrClass": "sync-unit",
        "ErrScope": "internal",
        "ErrLevel": "high",
         "Message": "startLocation: [position: (mysql-bin.000001, 2022), gtid-set:09bec856-ba95-11
             '→ ea-850a-58f2b4af5188:1-9 ], endLocation: [position: (mysql-bin.000001, 2022),
             '→ gtid-set: 09bec856-ba95-11ea-850a-58f2b4af5188:1-9]: gen insert sqls failed,
             '→ schema: log, table: messages: Column count doesn't match value count: 3 (columns)
             '→  vs 2 (values)",
        "RawCause": "",
        "Workaround": ""
    }
]
```

例如上游表结构为：

```
### 上游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
)
```

例如下游表结构为：

```
### 下游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  `message` varchar(255) DEFAULT NULL, # 下游比上游多出的列。
  PRIMARY KEY (`id`)
)
```

当 DM 尝试使用下游表结构解析上游产生的binlog event时，DM 会报出上述 `Column count doesn't match` 错误。

此时，你可以使用 `binlog-schema` 命令来为数据源中需要迁移的表指定表结构，表结构需要对应 DM 将要开始同步的binlog event 的数据。如果你在进行分表合并的数据迁移，那么需要为每个分表按照如下步骤在 DM 中设置用于解析binlog event 的表结构。具体操作为：

1. 在DM 中，新建一个.sql 文件，并将上游表结构对应的CREATE TABLE 语句添加到该文件。例如，将以下表结构保存到 log.messages.sql 中。如果是 6.0 及以上版本，可以直接通过 --from-source/--from-target 更新，无需创建 SQL 文件。可参考管理迁移表的表结构。

```
# 上游表结构
CREATE TABLE `messages` (
`id` int(11) NOT NULL,
PRIMARY KEY (`id`)
)
```

2. 使用 binlog-schema 命令为数据源中需要迁移的表设置表结构（此时数据迁移任务应该由于上述 Column count doesn't match 错误而处于 Paused 状态）。

```
tiup dmctl --master-addr ${advertise-addr} binlog-schema update -s ${source-id} ${task-name}
    ↪ ${database-name} ${table-name} ${schema-file}
```

该命令中的参数描述如下：

参数	描述
--master↪dmctl ↪ -↪ addr↪	指定要连接的集群的任意 DM-master 节点的 ${↪ advertise↪ -↪ addr↪ }。${↪ advertise↪ -↪ addr↪ } 表示 DM-master 向外界宣告的地址。

参数	描述
binlog ′→ - ′→ schemaschema ′→ ′→ update ′→	手动更新信息
-s	指定source。
${ ′→ task ′→ - ′→ name ′→ }	指定task。表示数据同步任务配置文件task. ′→ yaml ′→ 中定义的同步任务名称。
${ ′→ databadsaetabase。 ′→ - ′→ name ′→ }	指定表示上游数据库名。

参数	描述
${ ′→ table ′→ - ′→ name ′→ }	指定 table。表示上游数据表名。
${ ′→ schema ′→ - ′→ file ′→ }	指定表的 schema 文件。表示将被设置的表结构文件。

例如：

```
tiup dmctl --master-addr 172.16.10.71:8261 binlog-schema update -s mysql-01 task-test -d log
    ′→  -t message log.message.sql
```

3. 使用 resume-task 命令恢复处于 Paused 状态的同步任务。

```
tiup dmctl --master-addr ${advertise-addr} resume-task ${task-name}
```

4. 使用 query-status 命令确认数据迁移任务是否运行正常。

```
tiup dmctl --master-addr ${advertise-addr} query-status resume-task ${task-name}
```

3.4.3 如何过滤binlog事件

本文档介绍使用 DM 持续增量数据同步时，如何过滤binlog 事件。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到TiDB
- 从大数据量 MySQL 迁移数据到TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到TiDB

3.4.3.1 配置方式

配置 DM 的任务配置文件时，增加如下filter，具体配置示例如下图：

```
filters:
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["truncate table", "drop table"]
    sql-pattern: ["^DROP\\s+PROCEDURE", "^CREATE\\s+PROCEDURE"]
    action: Ignore
```

- schema-pattern/table-pattern：对匹配上的 schema 或table 进行过滤
- events：binlog events，支持的 Event 如下表所示:

Event	分类	说明
all		匹配所有 events
all dml		匹配所有 DML events
all ddl		匹配所有 DDL events
none		不匹配任何 events
none ddl		不包含任何 DDL events
none dml		不包含任何 DML events
insert	DML	匹配 insert DML event
update	DML	匹配 update DML event
delete	DML	匹配 delete DML event
create database	DDL	匹配 create database event
drop database	DDL	匹配 drop database event
create table	DDL	匹配 create table event
create index	DDL	匹配 create index event
drop table	DDL	匹配 drop table event
truncate table	DDL	匹配 truncate table event
rename table	DDL	匹配 rename table event
drop index	DDL	匹配 drop index event
alter table	DDL	匹配 alter table event

- sql-pattern：匹配指定的 DDL SQL 语句，支持正则表达式匹配。
- action：可取值 Do 或 Ignore。
  - Do：白名单。binlog event 如果满足下面两个条件之一将会被同步：
    * 符合 events 条件；
    * sql-pattern 不为空，且对应的 SQL 可以匹配上 sql-pattern 中任意一项。
  - Ignore：黑名单。如果满足下面两个条件之一就会被过滤掉：
    * 符合 events 条件；
    * sql-pattern 不为空，且对应的 SQL 可以匹配上 sql-pattern 中任意一项

注意：如果同时配置 Do/Ignore，则 Ignore 优先级更高。binlog event 不匹配白名单或者匹配黑名单都将被直接过滤。

3.4.3.2 使用场景举例

3.4.3.2.1 过滤分库分表的所有删除操作

设置 `filter-table-rule` 和 `filter-schema-rule` 两个过滤规则，具体如下：

```
filters:
  filter-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["truncate table", "drop table", "delete"]
    action: Ignore
  filter-schema-rule:
    schema-pattern: "test_*"
    events: ["drop database"]
    action: Ignore
```

3.4.3.2.2 只迁移分库分表的 DML 操作

设置两个 `Binlog event filter rule`：

```
filters:
  do-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["create table", "all dml"]
    action: Do
  do-schema-rule:
    schema-pattern: "test_*"
    events: ["create database"]
    action: Do
```

3.4.3.2.3 过滤 TiDB 不支持的 SQL 语句

```
filters:
  filter-procedure-rule:
    schema-pattern: "*"
    sql-pattern: [".*\\s+DROP\\s+PROCEDURE", ".*\\s+CREATE\\s+PROCEDURE", "ALTER\\s+TABLE[\\s\\S
        ↪ ]*ADD\\s+PARTITION", "ALTER\\s+TABLE[\\s\\S]*DROP\\s+PARTITION"]
    action: Ignore
```

> **注意：**
>
> 全局过滤规则的设置必须尽可能严格，以避免过滤掉需要迁移的数据。

3.4.3.3 探索更多

- 如何通过 SQL 表达式过滤binlog

3.4.4 如何通过 SQL 表达式过滤 DML

本文档介绍使用 DM 持续增量数据同步时，如何更加精细的过滤binlog 事件。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到TiDB
- 从大数据量 MySQL 迁移数据到TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到TiDB

在进行增量数据迁移时，可以通过如何过滤binlog 事件功能过滤某些类型的binlog event，例如不向下游迁移 DELETE 事件以达到归档、审计等目的。但是binlog event filter 无法以更细粒度判断某一行的 DELETE 事件是否要被过滤。

为了解决上述问题，从 v2.0.5 起，DM 支持在增量数据同步阶段使用binlog value filter过滤迁移数据。DM 支持的 ROW 格式的binlog 中，binlog event 带有所有列的值。你可以基于这些值配置 SQL 表达式。如果该表达式对于某条行变更的计算结果是 TRUE，DM 就不会向下游迁移该条行变更。

与如何过滤binlog 事件类似，表达式过滤需要在数据迁移任务配置文件里配置，详见下面配置样例。完整的配置及意义，可以参考DM 完整配置文件示例：

```
name: test
task-mode: all

mysql-instances:
  - source-id: "mysql-replica-01"
    expression-filters: ["even_c"]

expression-filter:
  even_c:
    schema: "expr_filter"
    table: "tbl"
    insert-value-expr: "c % 2 = 0"
```

上面的示例配置了 even_c 规则，并让 source ID 为 mysql-replica-01 的数据源引用了该规则。even_c 规则的含义是：对于 expr_filter 库下的 tbl 表，当插入的c 的值为偶数(c % 2 = 0) 时，不将这条插入语句迁移到下游。下面展示该规则的使用效果。

在上游数据源增量插入以下数据：

```
INSERT INTO tbl(id, c) VALUES (1, 1), (2, 2), (3, 3), (4, 4);
```

随后在下游查询 tbl 表，可见只有c 的值为单数的行迁移到了下游：

```
MySQL [test]> select * from tbl;

+------+---------- +
| id   | c    |
+------+---------- +
|    1 |    1 |
|    3 |    3 |
+------+---------- +

2 rows in set (0.001 sec)
```

3.4.4.1 配置参数及规则说明

- schema：要匹配的上游数据库库名，不支持通配符匹配或正则匹配。
- table: 要匹配的上游表名，不支持通配符匹配或正则匹配。
- insert-value-expr：配置一个表达式，对 INSERT 类型的binlog event (WRITE_ROWS_EVENT) 带有的值生效。不能与 update-old-value-expr、update-new-value-expr、delete-value-expr 出现在一个配置项中。
- update-old-value-expr：配置一个表达式，对 UPDATE 类型的binlog event (UPDATE_ROWS_EVENT) 更新对应的旧值生效。不能与 insert-value-expr、delete-value-expr 出现在一个配置项中。
- update-new-value-expr：配置一个表达式，对 UPDATE 类型的binlog event (UPDATE_ROWS_EVENT) 更新对应的新值生效。不能与 insert-value-expr、delete-value-expr 出现在一个配置项中。
- delete-value-expr：配置一个表达式，对 DELETE 类型的binlog event (DELETE_ROWS_EVENT) 带有的值生效。不能与 insert-value-expr、update-old-value-expr、update-new-value-expr 出现在一个配置项中。

> 注意：
>
> - update-old-value-expr 可以与 update-new-value-expr 同时配置。
> - 当二者同时配置时，会将“更新+旧值“满足update-old-value-expr 且” 更新+新值“满足 update-new-value-expr 的行过滤掉。
> - 当只配置一者时，配置的这条表达式会决定是否过滤整个行变更，即旧值的删除和新值的插入会作为一个整体被过滤掉。

SQL 表达式可以涉及一列或多列，也可使用TiDB 支持的SQL 函数，例如c % 2 = 0、a*a + b*b = c*c、ts > NOW()。

TIMESTAMP 类型的默认时区是任务配置文件中指定的时区，默认值是下游时区。可以使用 c_timestamp = '→ '2021-01-01 12:34:56.5678+08:00' 的方式显式指定时区。

配置项 expression-filter 下可以定义多条过滤规则，上游数据源在其 expression-filters 配置项中引用需要的规则使其生效。当有多条规则生效时，匹配到任意一条规则即会导致某个行变更被过滤。

> 注意：
> 为某张表设置过多的表达式过滤会增加 DM 的计算开销，可能导致数据迁移速度变慢。

4 数据集成

4.1 数据集成概述

数据集成一般是指数据在各个独立的数据源之间流动、转换和汇集。随着数据量的爆炸式增长和数据价值被深度挖掘，对数据集成的需求越来越普遍和迫切。为了避免TiDB 成为数据孤岛，顺利与各个数据系统进行集成，TiCDC 提供将TiDB 增量数据变更日志实时同步到其他数据系统的能力。本文介绍一些常用的数据集成场景，你可以依据这些场景选择最适合自己的数据集成方案。

4.1.1 与 Confluent Cloud 和 Snowflake 进行数据集成

你可以使用TiCDC 将 TiDB 的增量数据同步到 Confluent Cloud，并借助 Confluent Cloud 的能力最终将数据分别同步到 Snowflake、 ksqlDB、 SQL Server。参见与 Confluent Cloud 和 Snowflake 进行数据集成。

4.1.2 与 Apache Kafka 和 Apache Flink 进行数据集成

你可以使用TiCDC 将TiDB 的增量数据同步到Apache Kafka，并使用Apache Flink 消费Kafka 中的数据。参见与 Apache Kafka 和 Apache Flink 进行数据集成。

4.2 数据集成场景

4.2.1 与 Confluent Cloud 和 Snowflake 进行数据集成

Confluent 是一个兼容Apache Kafka 的数据流平台，能够访问、存储和管理连续的实时流数据，具备丰富的数据集成能力。自v6.1.0 开始，TiCDC 支持将增量变更数据以Avro 格式输出到Confluent。本文档介绍如何使用TiCDC 将TiDB 的增量数据同步到 Confluent Cloud，并借助 Confluent Cloud 的能力最终将数据分别同步到 Snowflake、 ksqlDB、 SQL Server。主要内容包括：

- 快速搭建包含TiCDC 的 TiDB 集群
- 创建将数据输出到 Confluent Cloud 的 changefeed
- 创建将数据从 Confluent Cloud 输出到 Snowflake、 ksqlDB 和 SQL Server 的连接器(Connector)
- 使用 go-tpc 写入数据到上游TiDB，并观察 Snowflake、 ksqlDB 和 SQL Server 中的数据

上述过程将会基于实验环境进行，你也可以参考上述执行步骤，搭建生产级别的集群。

4.2.1.1 输出增量数据到 Confluent Cloud

4.2.1.1.1 第 1 步：搭建环境

1. 部署包含TiCDC 的 TiDB 集群。

   在实验或测试环境中，可以使用TiUP Playground 功能快速部署TiCDC，命令如下：

   ```
   tiup playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
   # 查看集群状态
   tiup status
   ```

   如果尚未安装TiUP，可以参考安装TiUP。在生产环境下，可以参考使用TiUP 安装部署TiCDC 集群，完成 TiCDC 集群部署工作。

2. 注册 Confluent Cloud 并创建 Confluent 集群。

   创建 Basic 集群并开放Internet 访问，详见 Quick Start for Confluent Cloud。

4.2.1.1.2 第 2 步：创建Access Key Pair

1. 创建 Cluster API Key。

   在 Confluent 集群控制面板中依次点击 Data integration > API keys > Create key。在弹出的 Select scope for API key 页面，选择Global access。

   创建成功后会得到一个 Key Pair 文件，内容如下：

   ```
   === Confluent Cloud API key: xxx-xxxxx ===

   API key:
   L5WWA4GK4NAT2EQV

   API secret:
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

   Bootstrap server:
   xxx-xxxxx.ap-east-1.aws.confluent.cloud:9092
   ```

2. 记录 Schema Registry Endpoints。

   在 Confluent 集群控制面板中，选择 Schema Registry > API endpoint，记录 Schema Registry Endpoints，如下：

   ```
   https://yyy-yyyyy.us-east-2.aws.confluent.cloud
   ```

3. 创建 Schema Registry API key。

   在 Confluent 集群控制面板中，选择 Schema Registry > API credentials，点击Edit 和 Create key。

   创建成功后会得到一个 Key Pair 文件，内容如下：

   ```
   === Confluent Cloud API key: yyy-yyyyy ===

   API key:
   ```

```
7NBH2CAFM2LMGTH7

API secret:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

以上步骤也可以通过 Confluent CLI 实现，详见 Connect Confluent CLI to Confluent Cloud Cluster。

4.2.1.1.3 第 3 步：创建 Kafka changefeed

1. 创建changefeed 配置文件。

   根据 Avro 协议和 Confluent Connector 的要求和规范，每张表的增量数据需要发送到独立的Topic 中，并且每个事件需要按照主键值分发 Partition。因此，需要创建一个名为 `changefeed.conf` 的配置文件，填写如下内容：

   ```
   [sink]
   dispatchers = [
   {matcher = ['*.*'], topic = "tidb_{schema}_{table}", partition="index-value"},
   ]
   ```

   关于配置文件中 `dispatchers` 的详细解释，参考自定义 Kafka Sink 的Topic 和 Partition 的分发规则。

2. 创建一个changefeed，将增量数据输出到 Confluent Cloud：

   ```
   tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
       ′→ uri="kafka://<broker_endpoint>/ticdc-meta?protocol=avro&replication-factor=3&enable-
       ′→ tls=true&auto-create-topic=true&sasl-mechanism=plain&sasl-user=<broker_api_key>&sasl
       ′→ -password=<broker_api_secret>" --schema-registry="https://<schema_registry_api_key
       ′→ >:<schema_registry_api_secret>@<schema_registry_endpoint>" --changefeed-id="
       ′→ confluent-changefeed" --config changefeed.conf
   ```

   将如下字段替换为第 2 步： 创建Access Key Pair中创建和记录的值：

   - `<broker_endpoint>`
   - `<broker_api_key>`
   - `<broker_api_secret>`
   - `<schema_registry_api_key>`
   - `<schema_registry_api_secret>`
   - `<schema_registry_endpoint>`

   其中 `<schema_registry_api_secret>` 需要经过 HTML URL 编码后再替换，替换完毕后示例如下：

   ```
   tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
       ′→ uri="kafka://xxx-xxxxx.ap-east-1.aws.confluent.cloud:9092/ticdc-meta?protocol=avro&
       ′→ replication-factor=3&enable-tls=true&auto-create-topic=true&sasl-mechanism=plain&
       ′→ sasl-user=L5WWA4GK4NAT2EQV&sasl-password=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" --
       ′→  schema-registry="https://7NBH2CAFM2LMGTH7:xxxxxxxxxxxxxxxxx@yyy-yyyyy.us-east-2.aws
       ′→ .confluent.cloud" --changefeed-id="confluent-changefeed" --config changefeed.conf
   ```

- 如果命令执行成功，将会返回被创建的changefeed 的相关信息，包含被创建的changefeed 的 ID 以及相关信息，内容如下：

```
Create changefeed successfully!
ID: confluent-changefeed
Info: {... changfeed info json struct ...}
```

- 如果命令长时间没有返回，请检查当前执行命令所在服务器到 Confluent Cloud 之间网络可达性，参考 Test connectivity to Confluent Cloud。

3. Changefeed 创建成功后，执行如下命令，查看changefeed 的状态：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed list --server="http://127.0.0.1:8300"
```

可以参考管理 Changefeed，对 changefeed 状态进行管理。

4.2.1.1.4 第 4 步：写入数据以产生变更日志

完成以上步骤后，TiCDC 会将上游TiDB 的增量数据变更日志发送到 Confluent Cloud。本小节将对 TiDB 写入数据，以产生增量数据变更日志。

1. 模拟业务负载。

在测试实验环境下，可以使用 go-tpc 向上游TiDB 集群写入数据，以让TiDB 产生事件变更数据。执行以下命令，会首先在上游TiDB 创建名为 `tpcc` 的数据库，然后使用TiUP bench 写入数据到这个数据库中。

```
tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 prepare
tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 run --time 300s
```

关于 go-tpc 的更多详细内容，可以参考如何对TiDB 进行 TPC-C 测试。

2. 观察 Confluent 中数据传输情况。

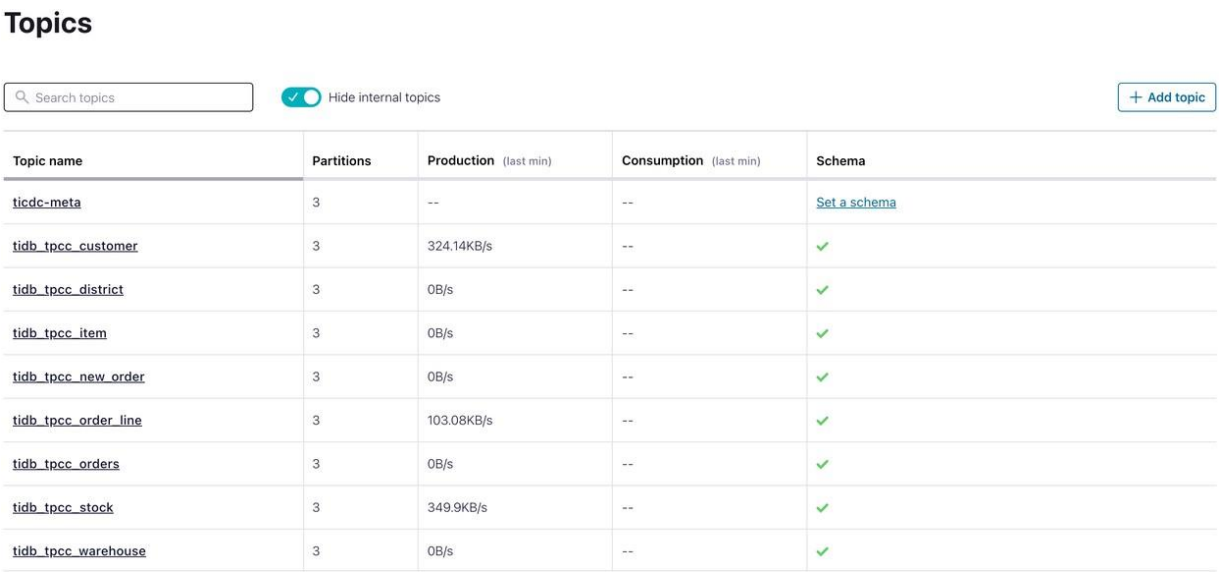


图 7: Confluent topics

在 Confluent 集群控制面板中，可以观察到相应的Topic 已经被自动创建，并有数据正在写入。至此，TiDB 数据库中的增量数据就被成功输出到了 Confluent Cloud。

4.2.1.2 与 Snowflake 进行数据集成

Snowflake 是一种云原生数据仓库。借助 Confluent 的能力，你只需要创建 Snowflake Sink Connector，就可以将 TiDB 的增量数据输出到 Snowflake。

4.2.1.2.1 准备工作

- 注册和创建 Snowflake 集群，参考Getting Started with Snowflake。
- 连接到 Snowflake 前，为 Snowflake 添加 Private Key，参考 Key Pair Authentication & Key Pair Rotation。

4.2.1.2.2 集成步骤

1. 在 Snowflake 中创建 Database 和 Schema。

   在 Snowflake 控制面板中，选择 Data > Database。创建名为 TPCC 的 Database 和名为 TiCDC 的 Schema。

2. 在 Confluent 集群控制面板中，选择 Data integration > Connectors > Snowflake Sink，进入如下页面：

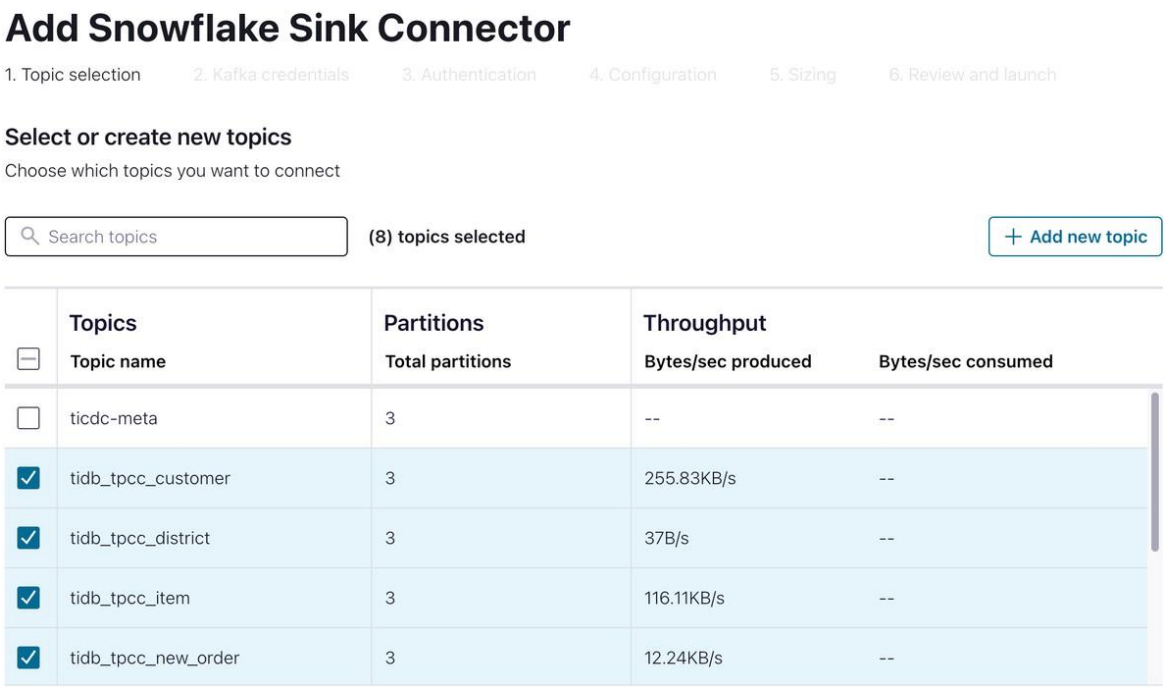


图 8: Add snowflake sink connector

3. 选择需要同步到 Snowflake 的 Topic 后，进入下一页面：

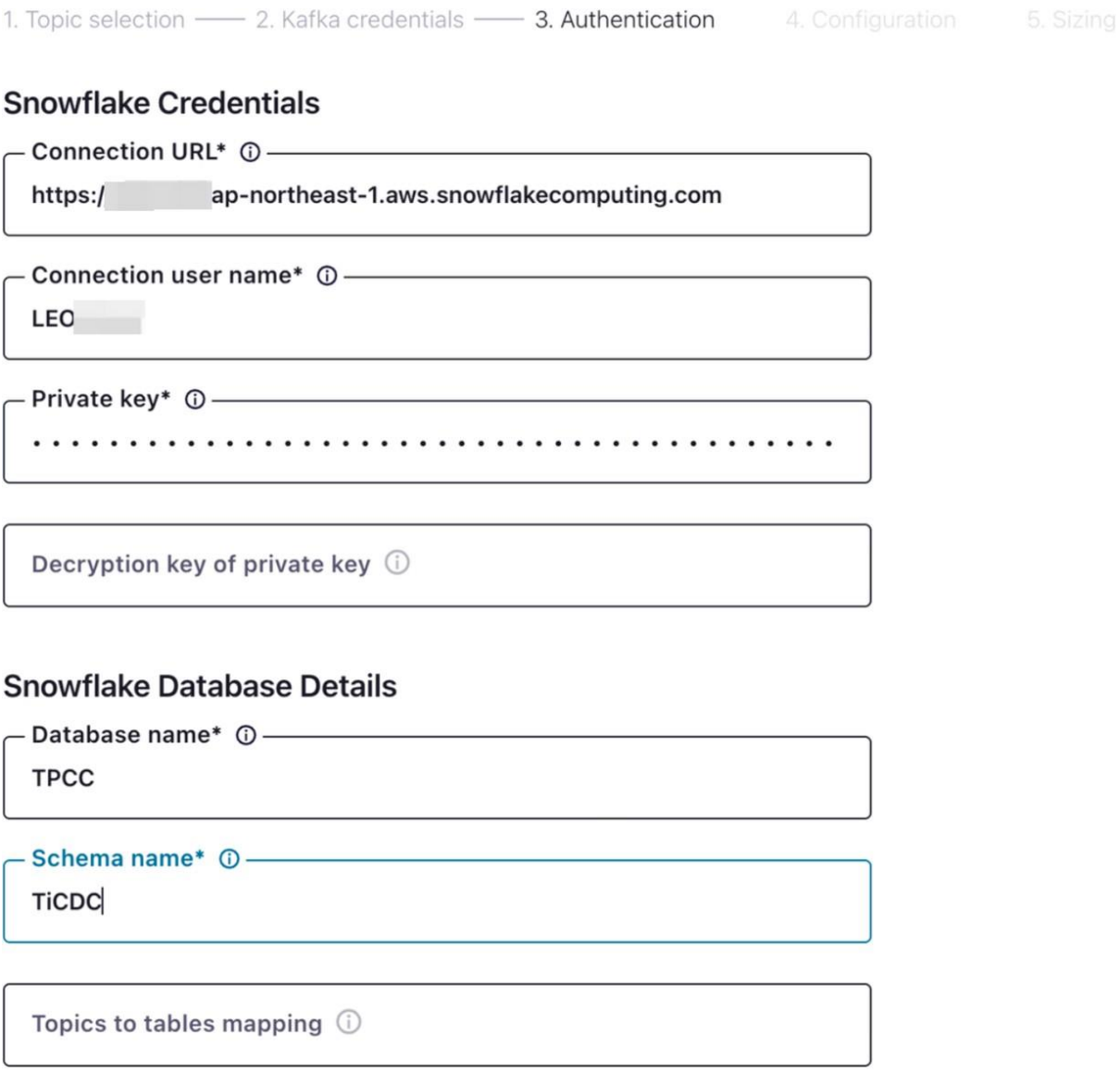


图 9: Credentials

4. 填写 Snowflake 连接认证信息，其中 Database name 和 Schema name 填写在上一步创建的 Database 和 Schema 名，随后进入下一页面：

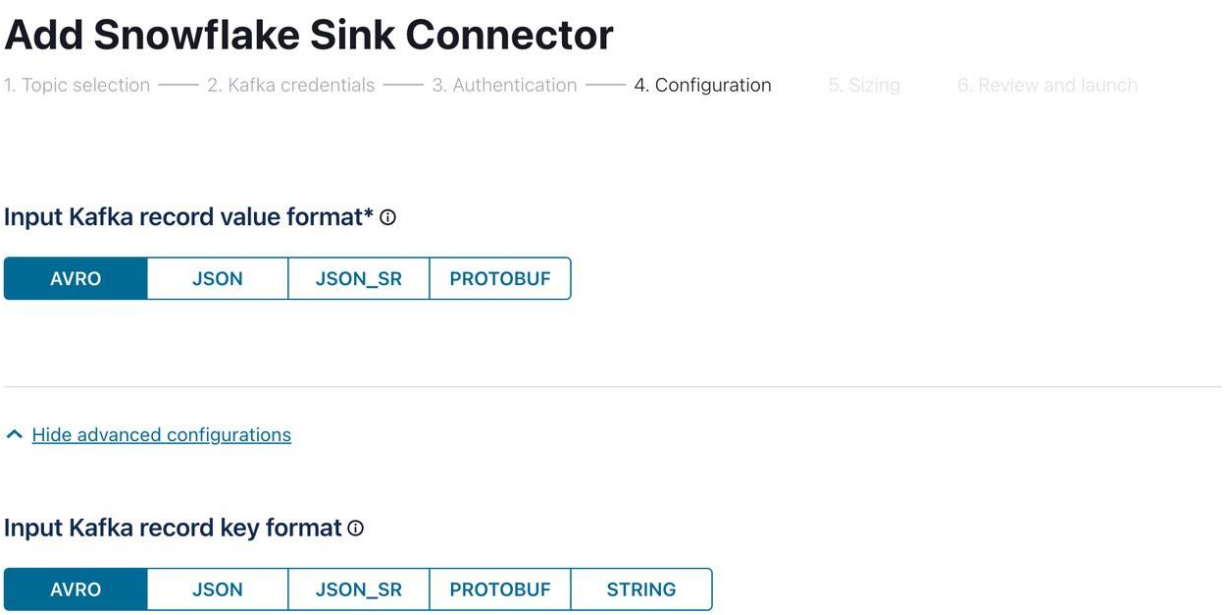


图 10: Configuration

5. 在 Configuration 页面中，`record value format` 和 `record key format` 都选择 AVRO，点击 Continue，直到 Connector 创建完成。等待 Connector 状态变为 `RUNNING`，这个过程可能持续数分钟。

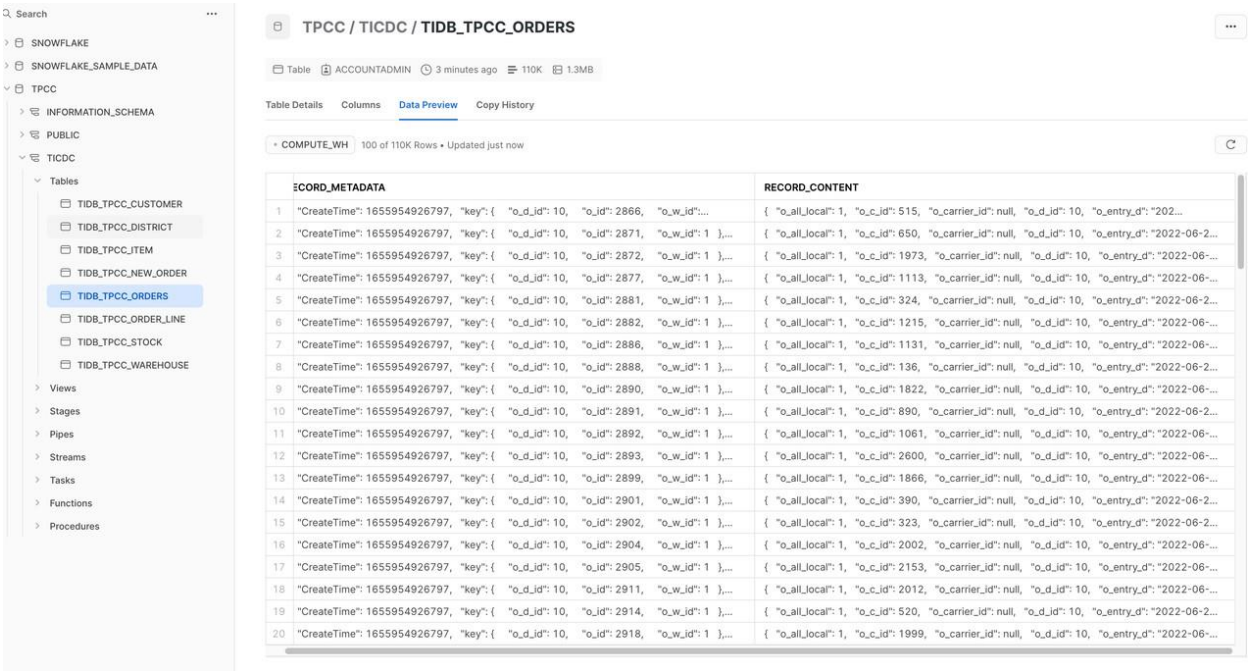


图 11: Data preview

6. 在 Snowflake 控制面板中，选择 Data > Database > TPCC > TiCDC，可以观察到TiDB 中的增量数据实时同步到了 Snowflake，如上图。但 Snowflake 中的表结构和TiDB 中的表结构不同，数据也以“追加”的方式插入 Snowflake 表。在大多数业务场景中，都希望 Snowflake 中的表数据是TiDB 表的一个副本，而不是存储TiDB 表的变更日志。该问题将在下一章节解决。

4.2.1.2.3 在 Snowflake 中创建TiDB 表对应的数据副本

在上一章节，TiDB 的增量变更日志已经被同步到 Snowflake 中，本章节将介绍如何借助 Snowflake 的 TASK 和 STREAM 功能，将实时写入 Snowflake 的数据变更日志根据 INSERT、UPDATE 和 DELETE 等事件类型分别处理，写入一个与上游TiDB 结构相同的表中，从而在 Snowflake 中创建一个数据副本。下面以 ITEM 表为例。

ITEM 表结构为：

```
CREATE TABLE `item` (
  `i_id` int(11) NOT NULL,
  `i_im_id` int(11) DEFAULT NULL,
  `i_name` varchar(24) DEFAULT NULL,
  `i_price` decimal(5,2) DEFAULT NULL,
  `i_data` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`i_id`)
);
```

Snowflake 中存在一张名为 TIDB_TEST_ITEM 的表，这张表是 Confluent Snowflake Sink Connector 自动创建的，表结构如下：

```
create or replace TABLE TIDB_TEST_ITEM
        ( RECORD_METADATA VARIANT,
        RECORD_CONTENT VARIANT
);
```

1. 根据TiDB 中的表结构，在 Snowflake 中创建结构相同的表：

```
create or replace table TEST_ITEM
    ( i_id INTEGER primary key,
    i_data VARCHAR,
    i_im_id INTEGER,
    i_name VARCHAR,
    i_price DECIMAL(36,2)
);
```

2. 为TIDB_TEST_ITEM 创建一个STREAM，将append_only 设为true，表示仅接收INSERT 事件。创建的STREAM 可以实时捕获 TIDB_TEST_ITEM 的 INSERT 事件，也就是说，当TiDB 中 ITEM 有新的变更日志时，变更日志将会被插入到 TIDB_TEST_ITEM 表，然后被 STREAM 捕获。

```
create or replace stream TEST_ITEM_STREAM on table TIDB_TEST_ITEM append_only=true;
```

3. 处理 STREAM 中的数据，根据不同的事件类型，在 TEST_ITEM 表中插入、更新或删除 STREAM 数据。

```
--将数据合并到 TEST_ITEM 表
merge into TEST_ITEM n
  using
      -- 查询 TEST_ITEM_STREAM
      (SELECT RECORD_METADATA:key as k, RECORD_CONTENT:val as v from TEST_ITEM_STREAM) stm
      -- 以 i_id 相等为条件将流和表做匹配
      on k:i_id = n.i_id
  -- 如果 TEST_ITEM 表中存在匹配 i_id 的记录，并且 v 为空，则删除这条记录
  when matched and IS_NULL_VALUE(v) = true then
      delete

  -- 如果 TEST_ITEM 表中存在匹配 i_id 的记录，并且 v 不为空，则更新这条记录
  when matched and IS_NULL_VALUE(v) = false then
      update set n.i_data = v:i_data, n.i_im_id = v:i_im_id, n.i_name = v:i_name, n.i_price
          ↪ = v:i_price

  -- 如果 TEST_ITEM 表中不存在匹配 i_id 的记录，则插入这条记录
  when not matched then
      insert
          (i_data, i_id, i_im_id, i_name, i_price)
      values
          (v:i_data, v:i_id, v:i_im_id, v:i_name, v:i_price)
;
```

在上面的语句中，我们使用了 Snowflake 的 MERGE INTO 语句，这个语句可以根据条件将流和表做匹配，然后根据不同的匹配结果，执行不同的操作，比如删除、更新或插入。在这个例子中，我们使用了三个 WHEN 子句，分别对应了三种情况：

- 当流和表匹配时，且流中的数据为空，则删除表中的记录
- 当流和表匹配时，且流中的数据不为空，则更新表中的记录
- 当流和表不匹配时，则插入流中的数据

4. 周期性执行第三步中的语句，以保证数据的实时性。可通过 Snowflake 的 SCHEDULED TASK 来实现：

```
-- 创建一个 TASK，周期性执行 MERGE INTO 语句
create or replace task STREAM_TO_ITEM
    warehouse = test
    -- 每分钟执行一次
    schedule = '1 minute'
when
    -- 当 TEST_ITEM_STREAM 中无数据时跳过
    system$stream_has_data('TEST_ITEM_STREAM')
as
-- 将数据合并到 TEST_ITEM 表，和上文中的 merge into 语句相同
merge into TEST_ITEM n
  using
```

```
    (select RECORD_METADATA:key as k, RECORD_CONTENT:val as v from TEST_ITEM_STREAM) stm
    on k:i_id = n.i_id
when matched and IS_NULL_VALUE(v) = true then
    delete
when matched and IS_NULL_VALUE(v) = false then
    update set n.i_data = v:i_data, n.i_im_id = v:i_im_id, n.i_name = v:i_name, n.i_price
        ↪ = v:i_price
when not matched then
    insert
        (i_data, i_id, i_im_id, i_name, i_price)
    values
        (v:i_data, v:i_id, v:i_im_id, v:i_name, v:i_price)
;
```

至此，你就建立了一条具备一定ETL 能力的数据通路，使得TiDB 的增量数据变更日志能够被输出到 Snowflake，并且维护一个 TiDB 表的数据副本， 实现在 Snowflake 中使用 TiDB 表的数据。最后一步操作是定期清理 TIDB_TEST_ITEM 表中的无用数据：

```
-- 每两小时清空表 TIDB_TEST_ITEM
create or replace task TRUNCATE_TIDB_TEST_ITEM
    warehouse = test
    schedule = '120 minute'
when
    system$stream_has_data('TIDB_TEST_ITEM')
as
    TRUNCATE table TIDB_TEST_ITEM;
```

4.2.1.3 与 ksqlDB 进行数据集成

ksqlDB 是一种面向流式数据处理的数据库。你可以直接在 Confluent Cloud 上创建 ksqlDB 集群，并且直接读取 TiCDC 输出到 Confluent 的增量数据。

1. 在 Confluent 集群控制面板中选择 ksqlDB，按照引导创建 ksqlDB 集群。

   等待集群状态为Running 后，进入下一步操作，这个过程可能持续数分钟。

2. 在 ksqlDB Editor 中执行如下命令，创建一个用于读取 tidb_tpcc_orders Topic 的 STREAM。

   ```
   CREATE STREAM orders (o_id INTEGER, o_d_id INTEGER, o_w_id INTEGER, o_c_id INTEGER,
       ↪ o_entry_d STRING, o_carrier_id INTEGER, o_ol_cnt INTEGER, o_all_local INTEGER) WITH
       ↪ (kafka_topic='tidb_tpcc_orders', partitions=3, value_format='AVRO');
   ```

3. 执行如下命令查询 orders STREAM 数据：

   ```
   SELECT * FROM ORDERS EMIT CHANGES;
   ```

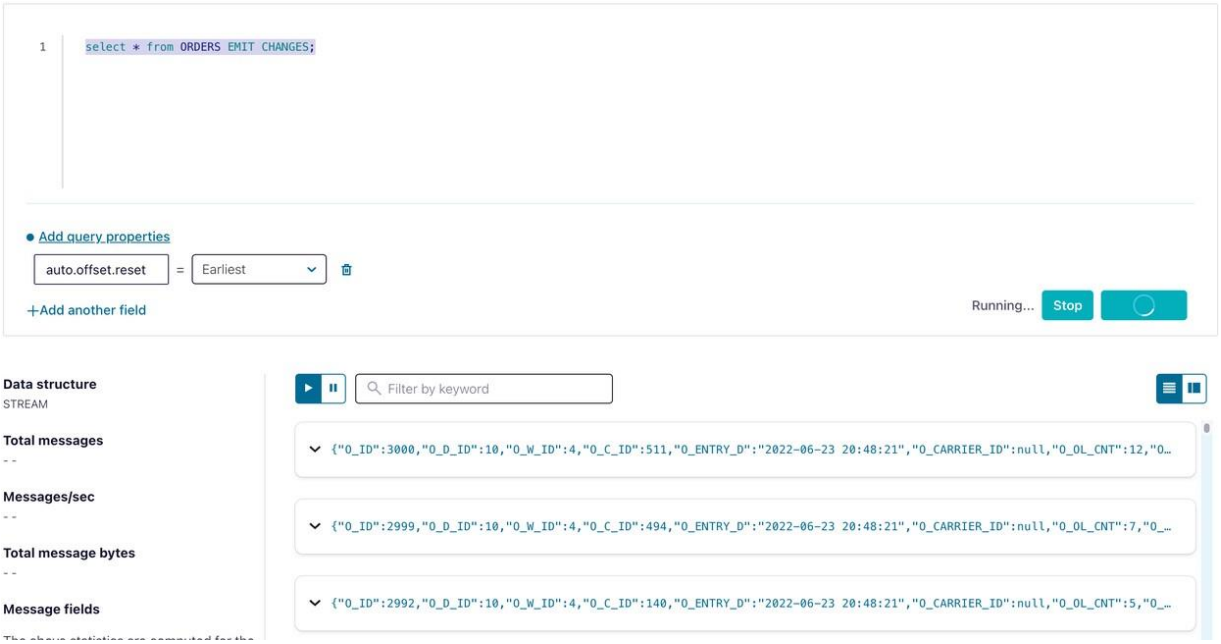


图 12: Select from orders

可以观察到TiDB 中的增量数据实时同步到了 ksqlDB，如上图。至此，就完成了 TiDB 与 ksqlDB 的数据集成。

4.2.1.4 与 SQL Server 进行数据集成

SQL Server 是 Microsoft 推出的关系型数据库软件。借助 Confluent 的能力，你只需要创建 SQL Server Sink Connector，就可以将TiDB 的增量数据输出到 SQL Server。

1. 连接 SQL Server 服务器，创建名为 `tpcc` 的数据库：

```
[ec2-user@ip-172-1-1-1 bin]$ sqlcmd -S 10.61.43.14,1433 -U admin
Password:
1> create database tpcc
2> go
1> select name from master.dbo.sysdatabases
2> go
name

--------------------------------------------------------------------------------
master
tempdb
model
msdb
rdsadmin
tpcc

(6 rows affected)
```

2. 在 Confluent 集群控制面板中，选择 Data integration > Connectors > Microsoft SQL Server Sink，进入如下页面：

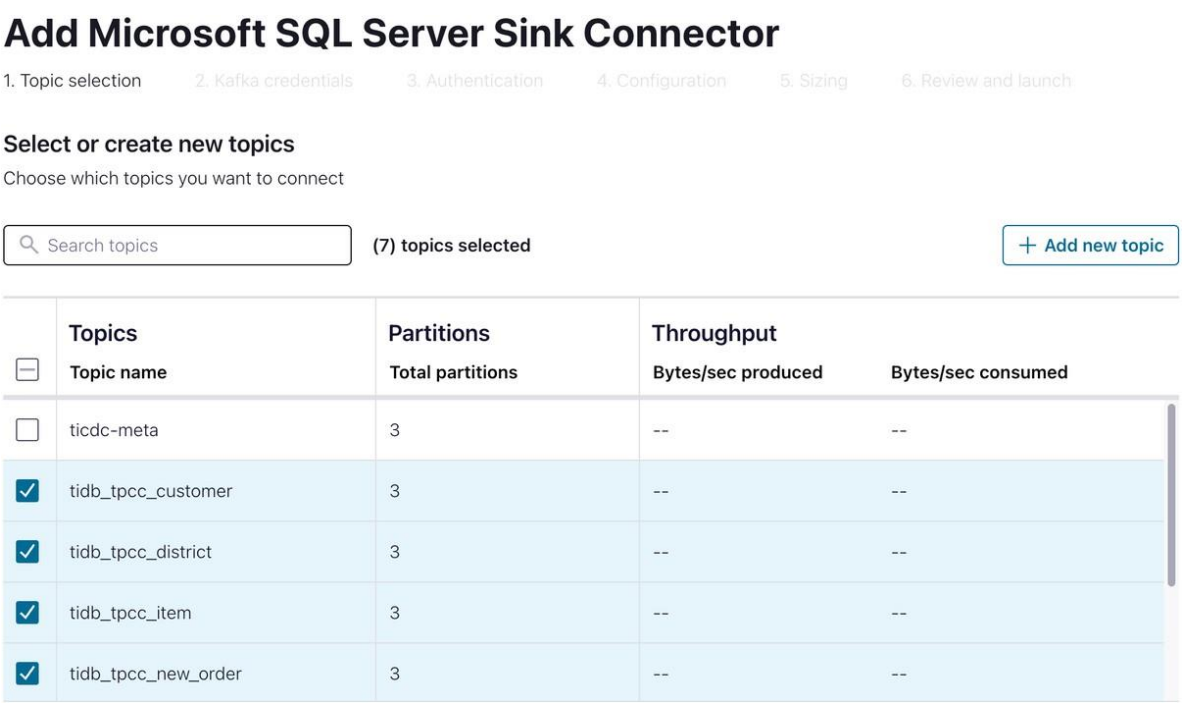


图 13: Topic selection

3. 选择需要同步到 SQL Server 的 Topic 后，进入下一页面：

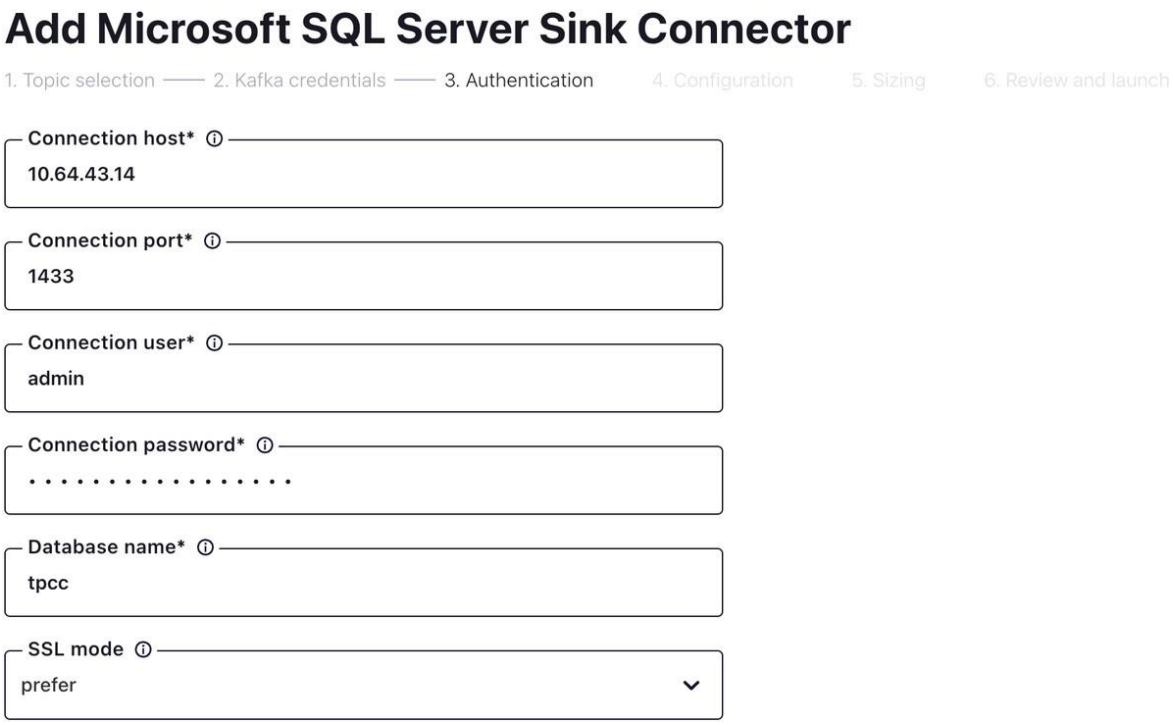


图 14: Authentication

4. 在填写 SQL Server 的连接和认证信息后，进入下一页面。
5. 在 Configuration 界面，按下表进行配置：

字段	取值
Input Kafka record value format	AVRO
Insert mode	UPSERT
Auto create table	true
Auto add columns	true
PK mode	record_key
Input Kafka record key format	AVRO
Delete on null	true

6. 配置完成后，选择 Continue，等待 Connector 状态变为RUNNING，这个过程可能持续数分钟。

Run Cancel | Disconnect Change Connection | tpcc

```
SELECT TOP (1000) [o_id]
      ,[o_d_id]
      ,[o_w_id]
      ,[o_c_id]
      ,[o_entry_d]
      ,[o_carrier_id]
      ,[o_ol_cnt]
      ,[o_all_local]
  FROM [tpcc].[dbo].[tidb_tpcc_orders]
```

Results Messages

	o_id	o_d_id	o_w_id	o_c_id	o_entry_d	o_carrier_id	o_ol_cnt	o_all_local
1	1	1	1	1860	2022-06-22 22:08:06	3	6	1
2	1	3	1	884	2022-06-22 22:08:06	10	5	1
3	2	1	1	1247	2022-06-22 22:08:06	7	10	1
4	2	3	1	752	2022-06-22 22:08:06	2	12	1
5	3	1	1	902	2022-06-22 22:08:06	5	11	1
6	3	2	1	1250	2022-06-22 22:08:06	8	7	1
7	4	1	1	1468	2022-06-22 22:08:06	7	6	1
8	4	2	1	2246	2022-06-22 22:08:06	3	9	1
9	4	3	1	358	2022-06-22 22:08:06	4	13	1
10	5	1	1	639	2022-06-22 22:08:06	9	10	1

图 15: Results

7. 连接 SQL Server。观察 TiDB 中的增量数据实时同步到了 SQL Server，如上图。至此，就完成了 TiDB 与 SQL Server 的数据集成。

4.2.2 与 Apache Kafka 和 Apache Flink 进行数据集成

本文档介绍如何使用TiCDC 将TiDB 的数据同步到Apache Kafka。主要包含以下内容：

1. 快速搭建TiCDC 集群、 Kafka 集群和 Flink 集群
2. 创建changefeed，将TiDB 增量数据输出至 Kafka
3. 使用 go-tpc 写入数据到上游TiDB
4. 使用 Kafka console consumer 观察数据被写入到指定的Topic
5. （可选）配置 Flink 集群消费 Kafka 内数据

上述过程将会基于实验环境进行。你也可以参考上述执行步骤，搭建生产级别的集群。

4.2.2.1 第 1 步：搭建环境

1. 部署包含TiCDC 的 TiDB 集群。

   在实验或测试环境中，可以使用TiUP Playground 功能，快速部署TiCDC，命令如下：

```
tiup playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
# 查看集群状态
tiup status
```

如果尚未安装TiUP，可以参考安装TiUP。在生产环境下，可以参考TiUP 安装部署TiCDC 集群，完成TiCDC 集群部署工作。

2. 部署 Kafka 集群。

   - 实验环境，可以参考Apache Kakfa Quickstart 启动 Kafka 集群。
   - 生产环境，可以参考Running Kafka in Production 完成 Kafka 集群搭建。

3. （可选）部署 Flink 集群。

   - 实验环境，可以参考Apache Flink First steps 启动 Flink 集群。
   - 生产环境，可以参考Apache Kafka Deployment 部署 Flink 生产集群。

4.2.2.2 第 2 步：创建 Kafka changefeed

1. 创建changefeed 配置文件。

   根据 Flink 的要求和规范，每张表的增量数据需要发送到独立的Topic 中，并且每个事件需要按照主键值分发 Partition。因此，需要创建一个名为 `changefeed.conf` 的配置文件，填写如下内容：

```
[sink]
dispatchers = [
{matcher = ['*.*'], topic = "tidb_{schema}_{table}", partition="index-value"},
]
```

   关于配置文件中dispatchers 的详细解释，参考自定义 Kafka Sink 的 Topic 和 Partition 的分发规则。

2. 创建一个changefeed，将增量数据输出到 Kafka：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
    ↪ uri="kafka://127.0.0.1:9092/kafka-topic-name?protocol=canal-json" --changefeed-id="
    ↪ kafka-changefeed" --config="changefeed.conf"
```

   - 如果命令执行成功，将会返回被创建的changefeed 的相关信息，包含被创建的changefeed 的 ID 以及相关信息，内容如下：

```
Create changefeed successfully!
ID: kafka-changefeed
Info: {... changfeed info json struct ...}
```

   - 如果命令长时间没有返回，你需要检查当前执行命令所在服务器到 sink-uri 中指定的 Kafka 机器的网络可达性，保证二者之间的网络连接正常。

生产环境下 Kafka 集群通常有多个broker 节点，你可以在 sink-uri 中配置多个broker 的访问地址，这有助于提升changefeed 到 Kafka 集群访问的稳定性，当部分被配置的 Kafka 节点故障的时候，changefeed 依旧可以正常工作。假设 Kafka 集群中有 3 个broker 节点，地址分别为 127.0.0.1:9092 / 127.0.0.2:9092 / 127.0.0.3:9092，可以参考如下 sink-uri 创建changefeed：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
    ↪ uri="kafka://127.0.0.1:9092,127.0.0.2:9092,127.0.0.3:9092/kafka-topic-name?protocol=
    ↪ canal-json&partition-num=3&replication-factor=1&max-message-bytes=1048576" --config
    ↪ ="changefeed.conf"
```

3. Changefeed 创建成功后，执行如下命令，查看changefeed 的状态：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed list --server="http://127.0.0.1:8300"
```

可以参考管理 Changefeed，对 changefeed 状态进行管理。

4.2.2.3 第 3 步：写入数据以产生变更日志

完成以上步骤后，TiCDC 会将上游TiDB 的增量数据变更日志发送到 Kafka，下面对 TiDB 写入数据，以产生增量数据变更日志。

1. 模拟业务负载。

   在测试实验环境下，可以使用 go-tpc 向上游TiDB 集群写入数据，以让TiDB 产生事件变更数据。如下命令，首先在上游TiDB 创建名为 tpcc 的数据库，然后使用TiUP bench 写入数据到这个数据库中。

```
tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 prepare
tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 run --time 300s
```

   关于 go-tpc 的更多详细内容，可以参考如何对TiDB 进行 TPC-C 测试。

2. 消费 Kafka Topic 中的数据。

   changefeed 正常运行时，会向 Kafka Topic 写入数据，你可以通过由 Kafka 提供的 kafka-console-consumer.sh，观测到数据成功被写入到 Kafka Topic 中：

```
./bin/kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --from-beginning --topic `
    ↪ ${topic-name}`
```

至此，TiDB 的增量数据变更日志就实时地复制到了 Kafka。下一步，你可以使用 Flink 消费 Kafka 数据。当然，你也可以自行开发适用于业务场景的 Kafka 消费端。

4.2.2.4 第 4 步：配置 Flink 消费 Kafka 数据（可选）

1. 安装 Flink Kafka Connector。

   在 Flink 生态中，Flink Kafka Connector 用于消费 Kafka 中的数据并输出到 Flink 中。Flink Kafka Connector 并不是内建的，因此在 Flink 安装完毕后，还需要将 Flink Kafka Connector 及其依赖项添加到 Flink 安装目录中。下载下列jar 文件至 Flink 安装目录下的lib 目录中，如果你已经运行了 Flink 集群，请重启集群以加载新的插件。

- flink-connector-kafka-1.15.0.jar
- flink-sql-connector-kafka-1.15.0.jar
- kafka-clients-3.2.0.jar

2. 创建一个表。

   你可以在 Flink 的安装目录执行如下命令，启动 Flink SQL 交互式客户端：

```
[root@flink flink-1.15.0]# ./bin/sql-client.sh
```

   随后，执行如下语句创建一个名为 `tpcc_orders` 的表：

```
CREATE TABLE tpcc_orders
    ( o_id INTEGER,
    o_d_id INTEGER,
    o_w_id INTEGER,
    o_c_id INTEGER,
    o_entry_d STRING,
    o_carrier_id INTEGER,
    o_ol_cnt INTEGER,
    o_all_local INTEGER
) WITH (
'connector' = 'kafka',
'topic' = 'tidb_tpcc_orders',
'properties.bootstrap.servers' = '127.0.0.1:9092',
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset',
'properties.auto.offset.reset' = 'earliest'
)
```

   请将 `topic` 和 `properties.bootstrap.servers` 参数替换为环境中的实际值。

3. 查询表内容。

   执行如下命令，查询 `tpcc_orders` 表中的数据：

```
SELECT * FROM tpcc_orders;
```

   执行成功后，可以观察到有数据输出，如下图：

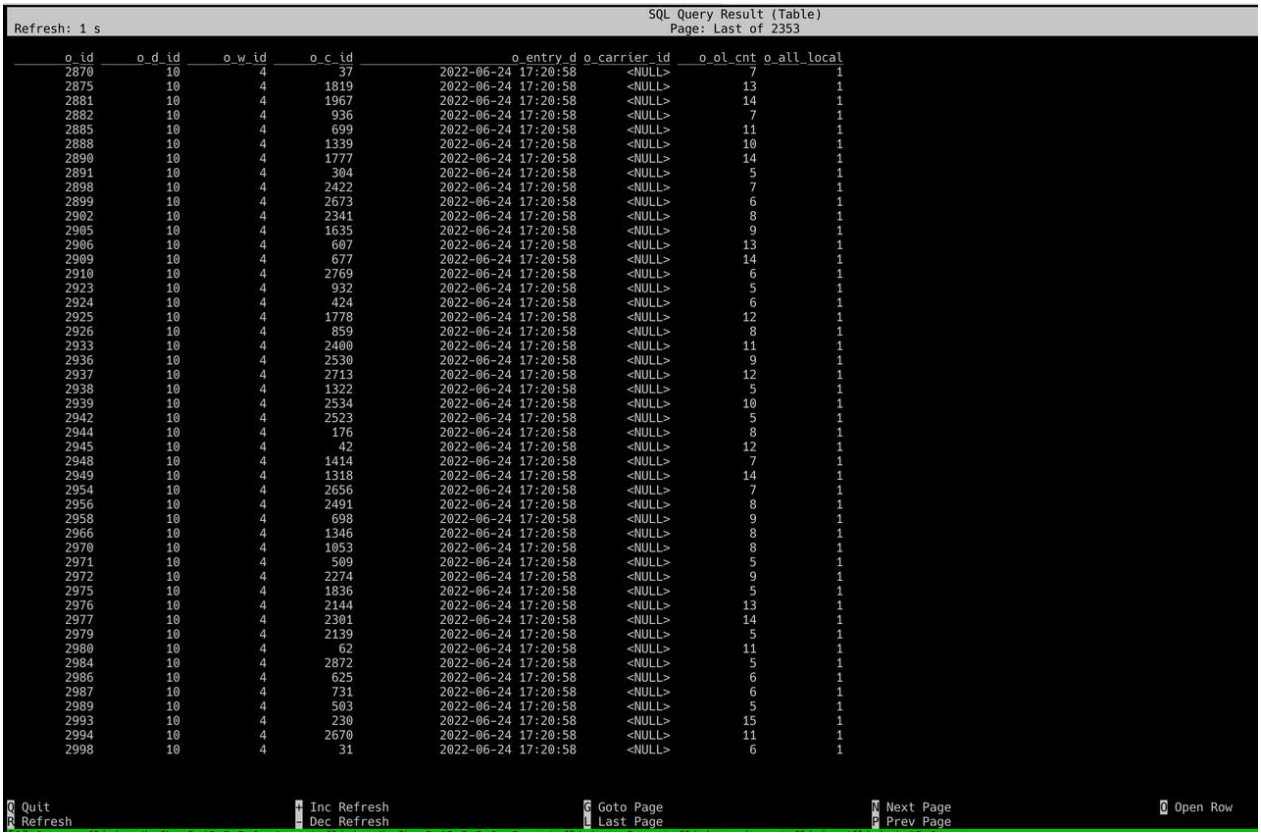


图 16: SQL query result

至此，就完成了TiDB 与 Flink 的数据集成。

5 运维操作

5.1 升级TiDB 版本

5.1.1 使用TiUP 升级TiDB

本文档适用于以下升级路径：

- 使用TiUP 从 TiDB 4.0 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 5.0-5.4 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.0 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.1 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.2 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.3 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.4 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.5 版本升级至TiDB 7.0。
- 使用TiUP 从 TiDB 6.6 版本升级至TiDB 7.0。

警告：

1. 不支持将TiFlash 组件从 5.3 之前的老版本在线升级至 5.3 及之后的版本，只能采用停机升级。如果集群中其他组件（如tidb，tikv）不能停机升级，参考不停机升级中的注意事项。
2. 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。
3. 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。

从TiDB v7.1 版本升级至更高的版本时，可以不遵循上面的限制 2 和 3，建议参考平滑升级TiDB的限制。

注意：

如果原集群是 3.0 或 3.1 或更早的版本，不支持直接升级到 7.0.0 及后续修订版本。你需要先从早期版本升级到 4.0 后，再从 4.0 升级到 7.0.0 及后续修订版本。

5.1.1.1 1. 升级兼容性说明

- TiDB 目前暂不支持版本降级或升级后回退。
- 使用TiDB Ansible 管理的 4.0 版本集群，需要先按照 4.0 版本文档的说明将集群导入到TiUP (`tiup cluster`) 管理后，再按本文档说明升级到 7.0.0 版本。
- 若要将 3.0 之前的版本升级至 7.0.0 版本：
  1. 首先通过TiDB Ansible 升级到3.0 版本。
  2. 然后按照 4.0 版本文档的说明，使用TiUP (`tiup cluster`) 将 TiDB Ansible 配置导入。
  3. 将集群升级至 4.0 版本。
  4. 按本文档说明将集群升级到 7.0.0 版本。
- 支持TiDB Binlog，TiCDC，TiFlash 等组件版本的升级。
- 将 v6.3.0 之前的TiFlash 升级至 v6.3.0 及之后的版本时，需要特别注意：在 Linux AMD64 架构的硬件平台部署 TiFlash 时，CPU 必须支持AVX2 指令集。而在 Linux ARM64 架构的硬件平台部署TiFlash 时，CPU 必须支持 ARMv8 架构。具体请参考6.3.0 版本Release Notes 中的描述。
- 具体不同版本的兼容性说明，请查看各个版本的Release Note。请根据各个版本的 Release Note 的兼容性更改调整集群的配置。
- 升级 v5.3 之前版本的集群到 v5.3 及后续版本时，默认部署的 Prometheus 会从 v2.8.1 升级到 v2.27.1，v2.27.1 提供更多的功能并解决了安全风险。Prometheus v2.27.1 相对于 v2.8.1 存在Alert 时间格式变化，详情见 Prometheus commit。

5.1.1.2 2. 升级前准备

本部分介绍实际开始升级前需要进行的更新TiUP 和TiUP Cluster 组件版本等准备工作。

5.1.1.2.1 2.1 查阅兼容性变更

查阅TiDB v7.0.0 release notes 中的兼容性变更。如果有任何变更影响到了你的升级，请采取相应的措施。

5.1.1.2.2 2.2 升级TiUP 或更新TiUP 离线镜像

升级TiUP 和 TiUP Cluster

> **注意：**
>
> 如果原集群中控机不能访问 `https://tiup-mirrors.pingcap.com` 地址，可跳过本步骤，然后更新 TiUP 离线镜像。

1. 先升级TiUP 版本（建议 `tiup` 版本不低于 `1.11.3`）：

```
tiup update --self
tiup --version
```

2. 再升级TiUP Cluster 版本（建议 `tiup cluster` 版本不低于 `1.11.3`）：

```
tiup update cluster
tiup cluster --version
```

更新TiUP 离线镜像

> **注意：**
>
> 如果原集群不是通过离线部署方式部署的，可忽略此步骤。

可以参考使用TiUP 部署TiDB 集群的步骤下载部署新版本的TiUP 离线镜像，上传到中控机。在执行`local_install.sh` 后，TiUP 会完成覆盖升级。

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

> **建议：**
>
> 关于 `TiDB-community-server` 软件包和 `TiDB-community-toolkit` 软件包的内容物，请查阅 TiDB 离线包。

覆盖升级完成后，需将 server 和toolkit 两个离线镜像合并，执行以下命令合并离线组件到 server 目录下。

```
tar xf tidb-community-toolkit-${version}-linux-amd64.tar.gz
ls -ld tidb-community-server-${version}-linux-amd64 tidb-community-toolkit-${version}-linux-amd64
cd tidb-community-server-${version}-linux-amd64/
cp -rp keys ~/.tiup/
tiup mirror merge ../tidb-community-toolkit-${version}-linux-amd64
```

离线镜像合并后，执行下列命令升级 Cluster 组件：

```
tiup update cluster
```

此时离线镜像已经更新成功。如果覆盖后发现TiUP 运行报错，可能是manifest 未更新导致，可尝试 `rm -rf ~/.tiup/manifests/*` 后再使用。

5.1.1.2.3 2.3 编辑 TiUP Cluster 拓扑配置文件

> 注意：
>
> 以下情况可跳过此步骤：
>
> - 原集群没有修改过配置参数，或通过tiup cluster 修改过参数但不需要调整。
> - 升级后对未修改过的配置项希望使用 7.0.0 默认参数。

1. 进入拓扑文件的 vi 编辑模式：

   ```
   tiup cluster edit-config <cluster-name>
   ```

2. 参考topology 配置模板的格式，将希望修改的参数填到拓扑文件的 server_configs 下面。

修改完成后 :wq 保存并退出编辑模式，输入Y 确认变更。

> 注意：
>
> 升级到 7.0.0 版本前，请确认已在 4.0 修改的参数在 7.0.0 版本中是兼容的，可参考TiKV 配置文件描述。

5.1.1.2.4 2.4 检查当前集群的健康状况

为避免升级过程中出现未定义行为或其他故障，建议在升级前对集群当前的region 健康状态进行检查，此操作可通过 check 子命令完成。

```
tiup cluster check <cluster-name> --cluster
```

执行结束后，最后会输出region status 检查结果。如果结果为“All regions are healthy”，则说明当前集群中所有 region 均为健康状态，可以继续执行升级；如果结果为“Regions are not fully healthy: m miss-peer, n pending-peer”并提示“Please fix unhealthy regions before other operations.”，则说明当前集群中有 region 处在异常状态，应先排除相应异常状态，并再次检查结果为“All regions are healthy”后再继续升级。

5.1.1.2.5 2.5 检查当前集群的 DDL 和 Backup 情况

为避免升级过程中出现未定义行为或其他故障，建议检查以下指标后再进行升级操作。

- 集群 DDL 情况：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。
- 集群 Backup 情况：建议使用 `SHOW [BACKUPS|RESTORES]` 命令查看集群中是否有正在进行的 Backup 或者 Restore 任务。如需升级，请等待 Backup 执行完成后，得到一个有效的备份后再执行升级。

5.1.1.3 3. 升级TiDB 集群

本部分介绍如何滚动升级TiDB 集群以及如何进行升级后的验证。

5.1.1.3.1 3.1 将集群升级到指定版本

升级的方式有两种：不停机升级和停机升级。TiUP Cluster 默认的升级TiDB 集群的方式是不停机升级，即升级过程中集群仍然可以对外提供服务。升级时会对各节点逐个迁移leader 后再升级和重启，因此对于大规模集群需要较长时间才能完成整个升级操作。如果业务有维护窗口可供数据库停机维护，则可以使用停机升级的方式快速进行升级操作。

不停机升级

```
tiup cluster upgrade <cluster-name> <version>
```

以升级到 7.0.0 版本为例：

```
tiup cluster upgrade <cluster-name> v7.0.0
```

注意：

- 滚动升级会逐个升级所有的组件。升级TiKV 期间，会逐个将TiKV 上的所有leader 切走再停止该TiKV 实例。默认超时时间为 5 分钟（300 秒），超时后会直接停止该实例。
- 使用 `--force` 参数可以在不驱逐leader 的前提下快速升级集群至新版本，但是该方式会忽略所有升级中的错误，在升级失败后得不到有效提示，请谨慎使用。
- 如果希望保持性能稳定，则需要保证TiKV 上的所有leader 驱逐完成后再停止该TiKV 实例，可以指定 `--transfer-timeout` 为一个更大的值，如 `--transfer-timeout 3600`，单位为秒。
- 若想将TiFlash 从 5.3 之前的版本升级到 5.3 及之后的版本，必须进行TiFlash 的停机升级。参考如下步骤，可以在确保其他组件正常运行的情况下升级TiFlash：

1. 关闭TiFlash 实例：`tiup cluster stop <cluster-name> -R tiflash`

2. 使用--offline 参数在不重启（只更新文件）的情况下升级集群：tiup cluster upgrade ′→ <cluster-name> <version> --offline，例如 tiup cluster upgrade <cluster-name ′→ > v6.3.0 --offline
3. reload 整个集群：tiup cluster reload <cluster-name>。此时，TiFlash 也会正常启动，无需额外操作。

- 在对使用TiDB Binlog 的集群进行滚动升级过程中，请避免新创建聚簇索引表。

停机升级

在停机升级前，首先需要将整个集群关停。

```
tiup cluster stop <cluster-name>
```

之后通过 upgrade 命令添加 --offline 参数来进行停机升级，其中 <cluster-name> 为集群名，<version> 为升级的目标版本，例如 v6.5.0。

```
tiup cluster upgrade <cluster-name> <version> --offline
```

升级完成后集群不会自动启动，需要使用 start 命令来启动集群。

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

5.1.1.3.2 3.2 升级后验证

执行 display 命令来查看最新的集群版本 TiDB Version：

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

```
Cluster type:       tidb
Cluster name:       <cluster-name>
Cluster version:    v7.0.0
```

5.1.1.4 4. 升级FAQ

本部分介绍使用TiUP 升级TiDB 集群遇到的常见问题。

5.1.1.4.1 4.1 升级时报错中断，处理完报错后，如何继续升级

重新执行 tiup cluster upgrade 命令进行升级，升级操作会重启之前已经升级完成的节点。如果不希望重启已经升级过的节点，可以使用 replay 子命令来重试操作，具体方法如下：

1. 使用 tiup cluster audit 命令查看操作记录：

```
tiup cluster audit
```

在其中找到失败的升级操作记录，并记下该操作记录的ID，下一步中将使用 `<audit-id>` 表示操作记录 ID 的值。

2. 使用 `tiup cluster replay <audit-id>` 命令重试对应操作：

```
tiup cluster replay <audit-id>
```

5.1.1.4.2 4.2 升级过程中 evict leader 等待时间过长，如何跳过该步骤快速升级

可以指定 `--force`，升级时会跳过 `PD transfer leader` 和 `TiKV evict leader` 过程，直接重启并升级版本，对线上运行的集群性能影响较大。命令如下，其中 `<version>` 为升级的目标版本，例如 `v6.5.0`：

```
tiup cluster upgrade <cluster-name> <version> --force
```

5.1.1.4.3 4.3 升级完成后，如何更新pd-ctl 等周边工具版本

可通过TiUP 安装对应版本的 `ctl` 组件来更新相关工具版本：

```
tiup install ctl:v7.0.0
```

5.1.2 使用TiDB Operator

5.1.3 平滑升级TiDB

> **警告：**
>
> 平滑升级目前为实验特性。

本文档介绍TiDB 的平滑升级集群功能，支持无需手动取消 DDL 的操作。

从 v7.1.0 起，当将TiDB 升级至更高的版本时，TiDB 支持平滑升级功能，取消了升级过程中的限制，提供更平滑的升级体验。此功能默认开启，且无开关控制。

5.1.3.1 功能简介

TiDB 引入平滑升级功能前，对于升级过程中的 DDL 操作有如下限制(可以参考使用TiUP 升级TiDB中警告部分)：

- 在升级过程中执行 DDL 操作，TiDB 可能会出现未定义的行为。
- 在 DDL 操作执行过程中升级TiDB，TiDB 可能会出现未定义的行为。

引入平滑升级后，TiDB 升级过程不再受上述限制。

升级过程中，TiDB 会自动进行以下操作，无需用户干预：

1. 暂停用户的 DDL 操作。

2. 执行升级过程中的系统 DDL 操作。
3. 恢复被暂停的用户的 DDL 操作。
4. 完成升级。

其中，恢复的 DDL job 仍会按升级前的顺序执行。

5.1.3.2 使用限制

使用平滑升级功能时，需要注意以下限制。

5.1.3.2.1 用户操作限制

- 在升级前，如果集群中存在正在处理的canceling DDL job，即有正在被处理的 DDL job 被用户取消了，由于处于canceling 状态的job 无法被 pause，TiDB 会尝试重试。如果重试失败，会报错并退出升级。
- 在使用TiUP 进行升级的场景下，由于TiUP 升级存在超时时间，如果在升级之前集群中有大量 DDL（超过 300 条）正在处理队列中等待执行，则此次升级可能会失败。
- 在升级过程中，不允许以下操作：
  - 对系统表（`mysql.*`、`information_schema.*`、`performance_schema.*`、`metrics_schema.*`）进行 DDL 操作。
  - 执行手动取消 DDL job 操作：`ADMIN CANCEL DDL JOBS job_id [, job_id] ...;`。
  - 导入数据。

5.1.3.2.2 工具使用限制

在升级过程中，不支持使用以下工具：

- BR：BR 可能会将处于paused 状态的 DDL 拷贝到TiDB 中，而此状态的 DDL 不能自动resume，可能导致后续 DDL 卡住的情况。
- DM 和 TiCDC：如果在升级过程中使用 DM 和 TiCDC 向 TiDB 导入 SQL，并且其中包含 DDL 操作，则该导入操作会被阻塞，并可能出现未定义错误。

5.1.4 TiFlash v6.2 升级帮助

本文介绍从TiFlash 低版本升级至 v6.2 时功能模块的变化，以及推荐的应对方法。

如需了解标准升级流程，请参考如下文档：

- 使用TiUP 升级TiDB
- 使用TiDB Operator 升级TiDB

注意：

- v6.2.0 新增了一项名为 FastScan 的实验功能，该功能在 v7.0.0 GA。FastScan 在牺牲强一致性保证的前提下可以大幅提升扫表性能。
- 不推荐跨主干版本升级包含TiFlash 的 TiDB 集群，如从 v4.x 升级至 v6.x，请先升级至 v5.x，然后再升级至 v6.x。
- v4.x. 已接近产品周期尾声，请尽早升级到 v5.x 及以上版本。具体的版本周期请参考TiDB 版本周期支持策略。
- v6.0 作为非 LTS 版本，不会推出后续的bug 修复版，请尽量使用 v6.1 及之后的 LTS 版本。
- 若想将TiFlash 从 v5.3.0 之前的版本升级到 v5.3.0 及之后的版本，必须进行TiFlash 的停机升级。参考如下步骤，可以在确保其他组件正常运行的情况下升级TiFlash：
  - 关闭TiFlash 实例：`tiup cluster stop <cluster-name> -R tiflash`
  - 使用 `--offline` 参数在不重启（只更新文件）的情况下升级集群：`tiup cluster upgrade <cluster-name> <version> --offline`，例如 `tiup cluster upgrade <cluster-name> v5.3.0 --offline`
  - reload 整个集群：`tiup cluster reload <cluster-name>`。此时，TiFlash 也会正常启动，无需额外操作。

5.1.4.1 从 v5.x 或 v6.0 升级至 v6.1

从 v5.x 或 v6.0 升级至 v6.1 时，需要注意TiFlash Proxy 和动态分区裁剪功能的变化。

5.1.4.1.1 TiFlash Proxy

TiFlash 在 v6.1.0 对 Proxy 做了升级（与 TiKV v6.0.0 对齐）。新的 Proxy 版本升级了RocksDB 版本，在升级过程中会自动将数据格式转换为新版本。

正常升级时，不会有明显风险。如果特殊场景（如测试验证）需要降级，请注意，v6.1 降级到之前的任意版本时，会无法解析新版 RocksDB 配置，从而导致 TiFlash 重启失败。请做好升级验证工作，并尽可能准备应急方案。

测试环境及特殊回退需求下的对策

强制缩容TiFlash 节点，并重新同步数据。操作步骤详见缩容TiFlash 节点。

5.1.4.1.2 动态分区裁剪

如果你没有也不打算开启动态分区裁剪，可略过本部分。

- TiDB v6.1 全新安装：默认开启动态分区裁剪(Dynamic Pruning)。
- TiDB v6.0 及之前版本：默认关闭动态分区裁剪。旧版本升级遵循已有设定，不会自动开启（相对的也不会关闭）此功能。

  升级完成之后，如果要启用动态分区裁剪特性，请确保 `tidb_partition_prune_mode` 的值为 `dynamic`，并手动更新分区表的全局统计信息。关于如何手动更新统计信息，参见动态裁剪模式。

5.1.4.2 从 v5.x 或 v6.0 升级至 v6.2

TiFlash 在 v6.2.0 将数据格式升级到V3 版本，因此，从 v5.x 或 v6.0 升级至 v6.2 时，除了需要注意TiFlash Proxy 和动态分区裁剪的变化，还应注意 PageStorage 变更数据版本带来的影响，具体如下：

- 已有节点升级 v6.2 后，随着数据不断写入，旧版本的数据会逐步转换成新版本数据。
- 新旧版本的数据格式不能做到完全的转换，这会带来一定系统开销（通常不影响业务，但需要注意）。因此升级完成后，建议使用 `COMPACT` 命令触发数据整理(Compaction) 将相关表的数据转成新版本格式。操作步骤如下：

    1. 对每张有TiFlash 副本（replica）的表执行如下命令：

    ```
    ALTER TABLE <table_name> COMPACT tiflash replica;
    ```

    2. 重启TiFlash 节点。

你可以在Grafana 监控查看是否还有表使用旧的数据版本：Tiflash summary > storage pool > Storage Pool Run Mode

- Only V2：使用 PageStorage V2 的表数量（包括分区数）
- Only V3：使用 PageStorage V3 的表数量（包括分区数）
- Mix Mode：从 V2 迁移到V3 的表数量（包括分区数）

测试环境及特殊回退需求下的对策

强制缩容TiFlash 节点，并重新同步数据。操作步骤详见缩容TiFlash 节点。

5.1.4.3 从 v6.1 升级至 v6.2

从 v6.1 升级至 v6.2 时，需要注意 PageStorage 变更数据版本带来的影响。具体请参考从 v5.x 或 v6.0 升级至 v6.2 中关于 PageStorage 的描述。

5.2 扩缩容

5.2.1 使用TiUP 扩容缩容TiDB 集群

TiDB 集群可以在不中断线上服务的情况下进行扩容和缩容。

本文介绍如何使用TiUP 扩容缩容集群中的TiDB、TiKV、PD、TiCDC 或者TiFlash 节点。如未安装TiUP，可参考部署文档中的步骤。

你可以通过 `tiup cluster list` 查看当前的集群名称列表。

例如，集群原拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD
10.0.1.5	TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	1T7iK3V

5.2.1.1 扩容TiDB/PD/TiKV 节点

如果要添加一个TiDB 节点，IP 地址为 10.0.1.5，可以按照如下步骤进行操作。

> **注意：**
>
> 添加 PD 节点和添加TiDB 节点的步骤类似。添加TiKV 节点前，建议预先根据集群的负载情况调整 PD 调度参数。

5.2.1.1.1 1. 编写扩容拓扑配置

> **注意：**
>
> - 默认情况下，可以不填写端口以及目录信息。但在单机多实例场景下，则需要分配不同的端口以及目录，如果有端口或目录冲突，会在部署或扩容时提醒。
> - 从 TiUP v1.0.0 开始，扩容配置会继承原集群配置的 global 部分。

在 scale-out.yml 文件添加扩容拓扑配置：

```
vi scale-out.yml
```

```
tidb_servers:
  - host: 10.0.1.5
    ssh_port: 22
    port: 4000
    status_port: 10080
    deploy_dir: /tidb-deploy/tidb-4000
    log_dir: /tidb-deploy/tidb-4000/log
```

TiKV 配置文件参考：

```
tikv_servers:
  - host: 10.0.1.5
    ssh_port: 22
    port: 20160
    status_port: 20180
    deploy_dir: /tidb-deploy/tikv-20160
    data_dir: /tidb-data/tikv-20160
    log_dir: /tidb-deploy/tikv-20160/log
```

PD 配置文件参考：

```
pd_servers:
  - host: 10.0.1.5
    ssh_port: 22
    name: pd-1
    client_port: 2379
    peer_port: 2380
    deploy_dir: /tidb-deploy/pd-2379
    data_dir: /tidb-data/pd-2379
    log_dir: /tidb-deploy/pd-2379/log
```

可以使用 `tiup cluster edit-config <cluster-name>` 查看当前集群的配置信息， 因为其中的 `global` 和 `server_configs` 参数配置默认会被 `scale-out.yml` 继承，因此也会在 `scale-out.yml` 中生效。

5.2.1.1.2 2. 执行扩容命令

执行 scale-out 命令前，先使用 check 及 `check --apply` 命令，检查和自动修复集群存在的潜在风险：

> 注意：
>
> 针对scale-out 命令的检查功能在tiup cluster v1.9.3 及后续版本中支持，请操作前先升级tiup cluster 版本。

（1）检查集群存在的潜在风险：

```
shell tiup cluster check <cluster-name> scale-out.yml --cluster --user root [-p] [-i /home/root/.
'→ ssh/gcp_rsa]
```

（2）自动修复集群存在的潜在风险：

```
shell tiup cluster check <cluster-name> scale-out.yml --cluster --apply --user root [-p] [-i /
'→ home/root/.ssh/gcp_rsa]
```

（3）执行 scale-out 命令扩容TiDB 集群：

```
shell tiup cluster scale-out <cluster-name> scale-out.yml [-p] [-i /home/root/.ssh/gcp_rsa]
```

以上操作示例中：

- 扩容配置文件为 `scale-out.yml`。
- `--user root` 表示通过root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- [-i] 及 [-p] 为可选项，如果已经配置免密登录目标机，则不需填写。否则选择其一即可，[-i] 为可登录到目标机的root 用户（或 –user 指定的其他用户）的私钥，也可使用 [-p] 交互式输入该用户的密码。

预期日志结尾输出 `Scaled cluster `<cluster-name>` out successfully` 信息，表示扩容操作成功。

5.2.1.1.3 3. 检查集群状态

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

打开浏览器访问监控平台http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD
10.0.1.5	TiDB + TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

5.2.1.2 扩容TiFlash 节点

如果要添加一个TiFlash 节点，其IP 地址为 `10.0.1.4`，可以按照如下步骤进行操作。

> 注意：
>
> 在原有TiDB 集群上新增TiFlash 组件需要注意：
>
> 1. 首先确认当前TiDB 的版本支持TiFlash，否则需要先升级TiDB 集群至 v5.0 以上版本。
> 2. 执 行 `tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> config set enable-placement-rules true` 命令，以开启 PD 的 Placement Rules 功能。或通过 pd-ctl 执行对应的命令。

5.2.1.2.1 1. 添加节点信息到 scale-out.yml 文件

编写 scale-out.yml 文件，添加该TiFlash 节点信息（目前只支持ip，不支持域名）：

```
tiflash_servers:
  - host: 10.0.1.4
```

5.2.1.2.2 2. 运行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yml
```

> 注意：
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 `-p` 来输入新机器的密码，或通过 `-i` 指定私钥文件。

5.2.1.2.3 3. 查看集群状态

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

打开浏览器访问监控平台http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD + TiFlash
10.0.1.5	TiDB+ TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

5.2.1.3 扩容TiCDC 节点

如果要添加TiCDC 节点，IP 地址为 10.0.1.3、10.0.1.4，可以按照如下步骤进行操作。

5.2.1.3.1 1. 添加节点信息到 scale-out.yml 文件

编写 scale-out.yml 文件：

```
cdc_servers:
  - host: 10.0.1.3
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
  - host: 10.0.1.4
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
```

5.2.1.3.2 2. 运行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yml
```

> **注意：**
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 `-p` 来输入新机器的密码，或通过 `-i` 指定私钥文件。

5.2.1.3.3 3. 查看集群状态

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

打开浏览器访问监控平台http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiFlash + TiCDC
10.0.1.5	TiDB+ TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

5.2.1.4 缩容TiDB/PD/TiKV 节点

如果要移除IP 地址为 10.0.1.5 的一个TiKV 节点，可以按照如下步骤进行操作。

> 注意：
>
> - 移除TiDB、 PD 节点和移除TiKV 节点的步骤类似。
> - 由于TiKV、 TiFlash 和 TiDB Binlog 组件是异步下线的，且下线过程耗时较长，所以TiUP 对 TiKV、 TiFlash 和TiDB Binlog 组件做了特殊处理，详情参考下线特殊处理。

> 注意：
>
> TiKV 中的 PD Client 会缓存 PD 节点的列表。当前版本的TiKV 有定期自动更新 PD 节点的机制，可以降低TiKV 缓存的 PD 节点列表过旧这一问题出现的概率。但你应尽量避免在扩容新 PD 后直接一次性缩容所有扩容前就已经存在的 PD 节点。如果需要，请确保在下线所有之前存在的 PD 节点前将 PD 的 leader 切换至新扩容的 PD 节点。

5.2.1.4.1 1. 查看节点ID 信息

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

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster display <cluster-name>

TiDB Cluster: <cluster-name>

TiDB Version: v7.0.0

ID       Role         Host     Ports                            Status  Data Dir        Deploy Dir
```

```
--  ----  ----  ------  -------  ---------
    ′→ -----------

10.0.1.3 :8300  cdc           10.0.1.3    8300                            Up       data/cdc-8300
    ′→      deploy/cdc-8300

10.0.1.4 :8300  cdc           10.0.1.4    8300                            Up       data/cdc-8300
    ′→      deploy/cdc-8300

10.0.1.4 :2379  pd            10.0.1.4    2379/2380                        Healthy data/pd-2379
    ′→      deploy/pd-2379

10.0.1.1 :20160 tikv          10.0.1.1    20160/20180                      Up       data/tikv-20160
    ′→     deploy/tikv-20160

10.0.1.2 :20160 tikv          10.0.1.2    20160/20180                      Up       data/tikv-20160
    ′→     deploy/tikv-20160

10.0.1.5 :20160 tikv         10.0.1.5    20160/20180                     Up       data/tikv-20160
    ′→     deploy/tikv-20160

10.0.1.3 :4000  tidb         10.0.1.3    4000/10080                      Up       -
    ′→ deploy/tidb-4000

10.0.1.4 :4000  tidb         10.0.1.4    4000/10080                      Up       -
    ′→ deploy/tidb-4000

10.0.1.5 :4000  tidb          10.0.1.5    4000/10080                       Up       -
    ′→ deploy/tidb-4000

10.0.1.3 :9000   tiflash       10.0.1.3    9000/8123/3930/20170/20292/8234  Up       data/tiflash
    ′→ -9000        deploy/tiflash-9000

10.0.1.4 :9000   tiflash       10.0.1.4    9000/8123/3930/20170/20292/8234  Up       data/tiflash
    ′→ -9000        deploy/tiflash-9000

10.0.1.5 :9090  prometheus    10.0.1.5    9090                            Up       data/prometheus
    ′→ -9090  deploy/prometheus-9090

10.0.1.5 :3000  grafana       10.0.1.5    3000                            Up       -
    ′→ deploy/grafana-3000

10.0.1.5:9093  alertmanager 10.0.1.5    9093/9094                       Up       data/
    ′→ alertmanager-9093 deploy/alertmanager-9093
```

5.2.1.180.2 2. 执行缩容操

```
tiup cluster scale-in <cluster-name> --node 10.0.1.5:20160
```

其中 --node 参数为需要下线节点的ID。

预期输出 Scaled cluster <cluster-name> in successfully 信息，表示缩容操作成功。

5.2.1.4.3 3. 检查集群状态

下线需要一定时间，下线节点的状态变为Tombstone 就说明下线成功。

执行如下命令检查节点是否下线成功：

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

打开浏览器访问监控平台http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiFlash + TiCDC
10.0.1.5	TiDB + Monitor（TiKV 已删除）
10.0.1.1	TiKV
10.0.1.2	TiKV

5.2.1.5 缩容TiFlash 节点

如果要缩容IP 地址为 10.0.1.4 的一个TiFlash 节点，可以按照如下步骤进行操作。

5.2.1.5.1 1. 根据TiFlash 剩余节点数调整数据表的副本数

1. 查询是否有数据表的TiFlash 副本数大于缩容后的TiFlash 节点数。tobe_left_nodes 表示缩容后的TiFlash 节点数。如果查询结果为空，可以开始执行缩容。如果查询结果不为空，则需要修改相关表的 TiFlash 副本数。

```
SELECT * FROM information_schema.tiflash_replica WHERE REPLICA_COUNT >  'tobe_left_nodes';
```

2. 对所有TiFlash 副本数大于缩容后的TiFlash 节点数的表执行以下语句，new_replica_num 必须小于等于 tobe_left_nodes：

```
ALTER TABLE <db-name>.<table-name> SET tiflash replica 'new_replica_num';
```

3. 重新执行步骤 1，确保没有数据表的TiFlash 副本数大于缩容后的TiFlash 节点数。

5.2.1.181.2 2. 执行缩容操

接下来，请任选下列方案其一进行缩容。

方案一：通过TiUP 缩容TiFlash 节点

1. 通过以下命令确定需要下线的节点名称：

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

2. 执行 scale-in 命令来下线节点，假设步骤 1 中获得该节点名为 10.0.1.4:9000

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:9000
```

方案二：手动缩容TiFlash 节点

在特殊情况下（比如需要强制下线节点），或者 TiUP 操作失败的情况下，可以使用以下方法手动下线TiFlash 节点。

1. 使用 pd-ctl 的 store 命令在 PD 中查看该TiFlash 节点对应的 store id。

   - 在pd-ctl（tidb-ansible 目录下的 resources/bin 包含对应的二进制文件）中输入 store 命令。
   - 若使用TiUP 部署，可以调用以下命令代替 pd-ctl：

   ```
   tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> store
   ```

> 注意：
> 如果集群中有多个 PD 实例，只需在以上命令中指定一个活跃 PD 实例的 IP:端口即可。

2. 在 pd-ctl 中下线该TiFlash 节点。

   - 在 pd-ctl 中输入 store delete <store_id>，其中 <store_id> 为上一步查到的该TiFlash 节点对应的 store id。
   - 若通过TiUP 部署，可以调用以下命令代替 pd-ctl：

   ```
   tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> store delete <store_id>
   ```

> 注意：
> 如果集群中有多个 PD 实例，只需在以上命令中指定一个活跃 PD 实例的 IP:端口即可。

3. 等待该TiFlash 节点对应的 store 消失或者 state_name 变成Tombstone 再关闭TiFlash 进程。
4. 手动删除TiFlash 的数据文件，具体位置可查看在集群拓扑配置文件中TiFlash 配置部分下的data_dir 目录。
5. 从 TiUP 拓扑信息中删除已经下线的TiFlash 节点信息：

```
tiup cluster scale-in <cluster-name> --node <pd_ip>:<pd_port> --force
```

> **注意：**
>
> 如果在集群中所有的TiFlash 节点停止运行之前，没有取消所有同步到TiFlash 的表，则需要手动在 PD 中清除同步规则，否则无法成功完成TiFlash 节点的下线。

手动在 PD 中清除同步规则的步骤如下：

1. 查询当前 PD 实例中所有与TiFlash 相关的数据同步规则。

```
curl http://<pd_ip>:<pd_port>/pd/api/v1/config/rules/group/tiflash
```

```
[
  {
    "group_id": "tiflash",
    "id": "table-45-r",
    "override": true,
    "start_key": "7480000000000000FF2D5F720000000000FA",
    "end_key": "7480000000000000FF2E00000000000000F8",
    "role": "learner",
    "count": 1,
    "label_constraints": [
      {
        "key": "engine",
        "op": "in",
          "values":
        [ "tiflash"
        ]
      }
    ]
  }
]
```

2. 删除所有与TiFlash 相关的数据同步规则。以 `id` 为 `table-45-r` 的规则为例，通过以下命令可以删除该规则。

```
curl -v -X DELETE http://<pd_ip>:<pd_port>/pd/api/v1/config/rule/tiflash/table-45-r
```

5.2.1.5.3 3. 查看集群状态

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

打开浏览器访问监控平台http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiCDC （ TiFlash 已删除 ）
10.0.1.5	TiDB + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

5.2.1.6 缩容TiCDC 节点

如果要缩容IP 地址为 10.0.1.4 的一个TiCDC 节点，可以按照如下步骤进行操作。

5.2.1.6.1 1. 下线该TiCDC 节点

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:8300
```

5.2.1.6.2 2. 查看集群状态

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

打开浏览器访问监控平台http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + (TiCDC 已删除)
10.0.1.5	TiDB + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

5.2.2 使用TiDB Operator

5.3 备份与恢复

5.3.1 TiDB 备份与恢复概述

基于 Raft 协议和合理的部署拓扑规划，TiDB 实现了集群的高可用，当集群中少数节点挂掉时，集群依然能对外提供服务。在此基础上，为了更进一步保证用户数据的安全，TiDB 还提供了集群的备份与恢复(Backup & Restore, BR) 功能，作为数据安全的最后一道防线，使得集群能够免于严重的自然灾害，提供业务误操作“复原”的能力。

TiDB 备份恢复功能可以用于满足以下业务的需求：

- 备份集群数据到灾备系统，并保证Recovery Point Objective (RPO) 低至 5 分钟，减少灾难场景下数据的丢失。

- 处理业务数据写错的案例，提供业务操作的“复原”能力。
- 审计业务的历史数据，满足司法审查的需求。
- 复制(Clone) 生产环境，方便问题诊断、性能调优验证、仿真测试等。

5.3.1.1 使用须知

本部分介绍使用TiDB 备份恢复功能前的注意事项，包括使用限制和使用建议。

5.3.1.1.1 使用限制

- PITR 仅支持恢复到全新的空集群。
- PITR 仅支持集群粒度的恢复，不支持对单个database 或 table 的恢复。
- PITR 不支持恢复系统表中用户表和权限表的数据。
- 不支持在一个集群上同时运行多个数据备份任务。
- PITR 数据恢复任务运行期间，不支持同时运行日志备份任务，也不支持通过TiCDC 同步数据到下游集群。

5.3.1.1.2 使用建议

进行快照备份

- 推荐在业务低峰时执行集群快照数据备份，这样能最大程度地减少对业务的影响。
- 不推荐同时运行多个集群快照数据备份任务。不同的任务并行，不仅会导致备份的性能降低，影响在线业务，还会因为任务之间缺少协调机制造成任务失败，甚至对集群的状态产生影响。

进行快照恢复

- BR 恢复数据时会尽可能多地占用恢复集群的资源，因此推荐恢复数据到新集群或离线集群。应避免恢复数据到正在提供服务的生产集群，否则，恢复期间会对业务产生不可避免的影响。

备份存储和网络配置

- 推荐使用支持Amazon S3、 GCS 或 Azure Blob Storage 协议的存储系统保存备份数据；
- 应确保 BR、 TiKV 节点和备份存储系统有足够的网络带宽，备份存储系统能提供足够的写入和读取性能，否则，它们有可能成为备份恢复时的性能瓶颈。

5.3.1.2 功能使用

根据TiDB 部署方式的不同，使用备份恢复功能的方式也不同。本文主要介绍在 On-Premise 物理部署方式下，如何使用br 命令行工具进行TiDB 的备份和恢复。

其它TiDB 的部署方式的备份恢复功能使用，可以参考：

- 备份恢复部署在TiDB Cloud 上的TiDB。推荐在 TiDB Cloud 上创建TiDB 集群，集群的运维管理将由TiDB Cloud 团队托管完成，你可以聚焦于业务。
- 备份恢复部署在 Kubernetes 上的TiDB。如果你使用TiDB Operator 在 Kubernetes 中部署了TiDB 集群，建议通过 Kubernetes CustomResourceDefinition (CRD) 来提交备份和恢复任务。

5.3.1.3 功能介绍

使用备份恢复功能，你可以进行以下两类操作：

- 对集群进行备份：你可以对集群某个时间点的全量数据进行备份（全量备份），也可以对业务写入在 TiDB 产生的数据变更记录进行备份（日志备份，日志指的是TiKV 中的 kv 变更数据的记录）。
- 恢复备份数据：
  - 你可以恢复某个全量备份，或者全量备份中的部分库/表，将目标集群恢复到该全量备份对应的数据状态。
  - 基于备份（全量和日志）数据，你可以指定任意时间点，将目标集群恢复到该时间点所对应的备份集群数据状态(Point-in-time recovery, PITR)。

5.3.1.3.1 备份集群数据

全量备份是对集群某个时间点的全量数据进行备份。TiDB 支持以下方式的全量备份：

- 快照数据备份：TiDB 集群快照数据包含某个物理时间点上集群满足事务一致性的所有数据。BR 支持备份集群快照数据，使用请参考快照备份。

全量备份一般会占用不小的存储空间，且只包含某个时间点的集群数据。如果你需要灵活的选择恢复的时间点，即实现 PITR，可以按以下说明同时使用两种备份方式：

- 开启日志备份任务后，任务会在所有TiKV 节点上持续运行，以小批量的形式定期将TiDB 变更数据备份到指定存储中。
- 定期执行快照备份，备份集群全量数据到备份存储，例如在每天零点进行集群快照备份。

备份的性能，以及对集群的影响

- 集群快照数据备份，对TiDB 集群的影响可以保持在 20% 以下；通过合理的配置TiDB 集群用于备份资源，影响可以降低到 10% 及更低；单TiKV 存储节点的备份速度可以达到 50 MB/s ~ 100 MB/s，备份速度具有可扩展性；更详细说明请参考备份性能和影响。
- 单独运行日志备份时影响约在 5%。日志备份每隔 3 ~ 5 分钟将上次刷新后产生的变更数据记录刷新到备份存储中，可以实现低至五分钟RPO 的集群容灾目标。

5.3.1.3.2 恢复备份数据

与备份功能相对应，你可以进行两种类型的恢复：全量恢复和 PITR。

- 恢复某个全量备份
  - 恢复集群快照数据备份：你可以在一个空集群或不存在数据冲突（相同 schema 或 table）的集群执行快照备份恢复，将该集群恢复到快照备份对应数据状态。使用请参考恢复快照备份。此外你可以只恢复备份数据中指定库/表的局部数据。该功能在恢复过程中过滤掉不需要的数据。使用请参考恢复备份数据中指定库表的数据。

- 恢复到集群的历史任意时间点(PITR)
  - 通过 `br restore point` 功能。你可以指定要恢复的时间点，恢复时间点之前最近的快照数据备份，以及日志备份数据。BR 会自动判断和读取恢复需要的数据，然后将这些数据依次恢复到指定的集群。

恢复的性能

- 恢复集群快照数据备份，速度可以达到单TiKV 存储节点 100 MiB/s，恢复速度具有可扩展性；BR 只支持恢复数据到新集群，会尽可能多的使用恢复集群的资源。更详细说明请参考恢复性能和影响。
- 恢复日志备份数据，速度可以达到 30 GiB/h。更详细说明请参考PITR 性能和影响。

5.3.1.4 备份存储

TiDB 支持将数据备份到Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage、 NFS，或者实现 S3 协议的其他文件存储服务。更多备份存储的详细信息，请参考如下内容：

- 使用 URI 格式指定备份存储
- 配置备份存储的访问权限

5.3.1.5 兼容性

在使用备份恢复功能之前，需要先了解 BR 工具与其他功能的兼容性以及使用限制。

5.3.1.5.1 与其他功能的兼容性

某些功能在开启或关闭状态下，会导致备份恢复功能使用出错。因此需要保证恢复集群的这些配置，与备份集群备份时的配置相同。

功能	相关 issue	解决方式
GBK charset		BR 在 v5.4.0 之前不支持恢复 `charset ′→ = ′→ GBK` 的表。并且，任何版本的 BR 都不支持恢复 `charset ′→ = ′→ GBK` 的表到 v5.4.0 之前的 TiDB 集群。

功能	相关 issue	解决方式

聚簇索引	#565	确保恢复时集群的 `tidb_enable_clustered_index` 全局变量和备份时一致，否则会导致数据不一致的问题，例如
		
		`default`
		
		`not`
		
		`found` 和数据索引不一致。

功能	相关 issue	解决方式
New collation	#352	确保恢复时集群的 `new_collations_enabled_on_first_bootstrap` 变量值和备份时的一致，否则会导致数据索引不一致和 checksum 通不过。更多信息，请参考FAQ - BR 为什么会报 `new_collations_enabled_on_first_bootstrap` 不匹配？。
全局临时表		确保使用 BR v5.3.0 及以上版本进行备份和恢复，否则会导致全局临时表的表定义错误。

功能	相关 issue	解决方式
TiDB Lightning Physical Import		上游数据库使用 TiDB Lightning Physical 方式导入的数据，无法作为数据日志备份下来。推荐在数据导入后执行一次全量备份，细节参考上游数据库使用 TiDB Lightning Physical 方式导入数据的恢复。

5.3.1.5.2 版本间兼容性

> **注意：**
>
> 在执行备份和恢复操作时，建议使用和TiDB 集群相同版本的 BR。

在执行备份和恢复操作前，BR 会比较自身版本与TiDB 集群版本。如果版本之间不兼容，BR 会提示报错并退出。如要跳过版本检查，可以设置 `--check-requirements=false`。需要注意的是，跳过检查可能会遇到数据

不兼容的问题。

备份版本（纵向） 恢复版本（横向）	恢复到 TiDB v6.0	恢复到 TiDB v6.1	恢复到 TiDB v6.2	恢复到 TiDB v6.3、v6.4 或 v6.5	恢复到 TiDB v6.6
TiDB v6.0、v6.1、v6.2、v6.3、v6.4 或 v6.5 快照备份	兼容（已知问题，如果备份数据中包含空库可能导致报错，参考 #36379）	兼容	兼容	兼容	兼容（需使用 v6.6 及以上版本的 BR）
TiDB v6.3、v6.4、v6.5 或 v6.6 日志备份	不兼容	不兼容	不兼容	兼容	兼容

5.3.1.6 探索更多

- TiDB 快照备份与恢复使用指南
- TiDB 日志备份与 PITR 使用指南
- 备份存储

5.3.2 架构设计

5.3.2.1 TiDB 备份与恢复功能架构概述

正如TiDB 备份与恢复概述所介绍，TiDB 备份恢复功能包含了多种不同类型的集群数据对象的备份与恢复实现。这些功能都以 Backup & Restore (BR) 和 TiDB Operator 为使用入口，创建相应的任务从TiKV 节点上备份数据，或者恢复数据到TiKV 节点。

关于各种备份恢复功能的实现架构，请参考以下链接：

- 全量数据备份与恢复
  - 备份集群快照数据
  - 恢复快照备份数据
- 数据变更日志备份
  - 日志备份 - 备份 kv 数据变更
- Point-in-time recovery (PITR)
  - 恢复到指定时间点

5.3.2.2 TiDB 快照备份与恢复功能架构

本文以使用 BR 工具进行备份与恢复为例，介绍TiDB 集群快照数据备份和恢复的架构设计与流程。

5.3.2.2.1 架构设计

快照数据备份和恢复的架构如下：

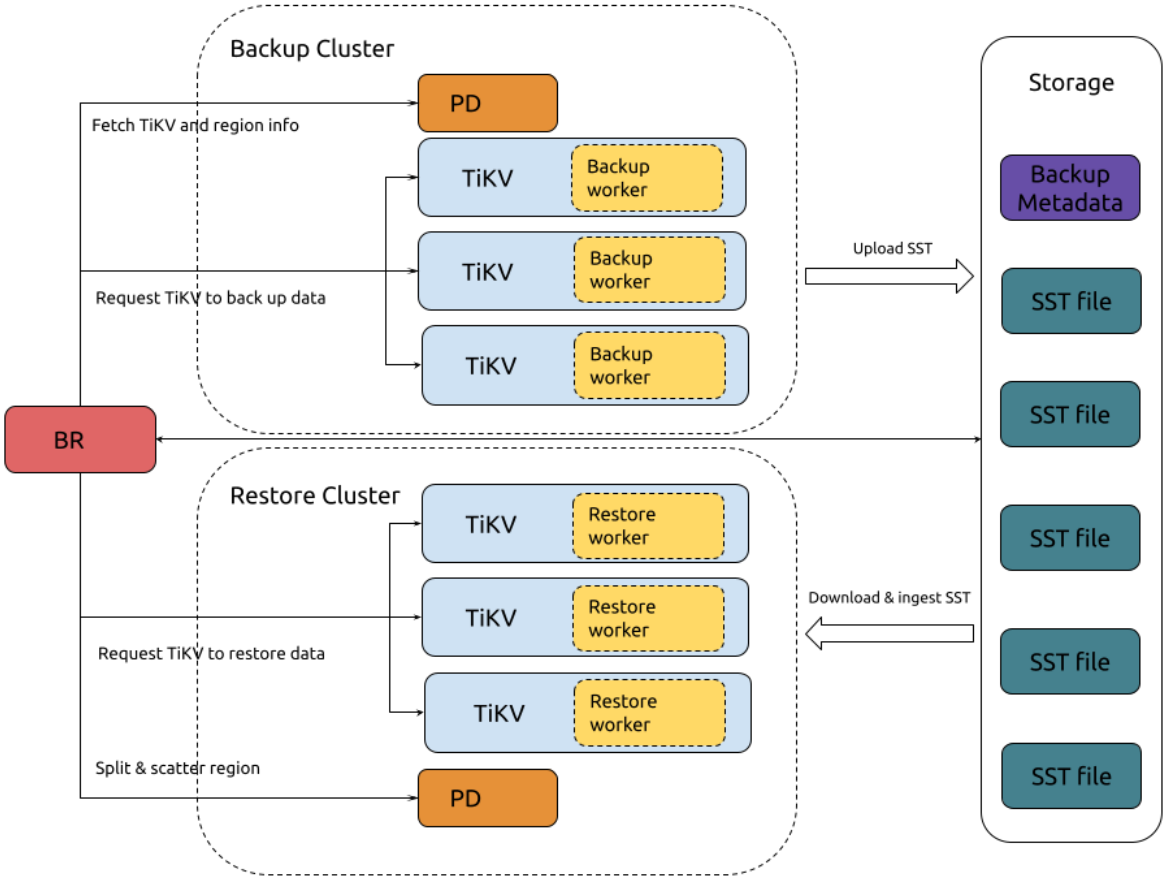


图 17: BR snapshot backup and restore architecture

5.3.2.2.2 备份流程

集群快照数据备份的流程如下：

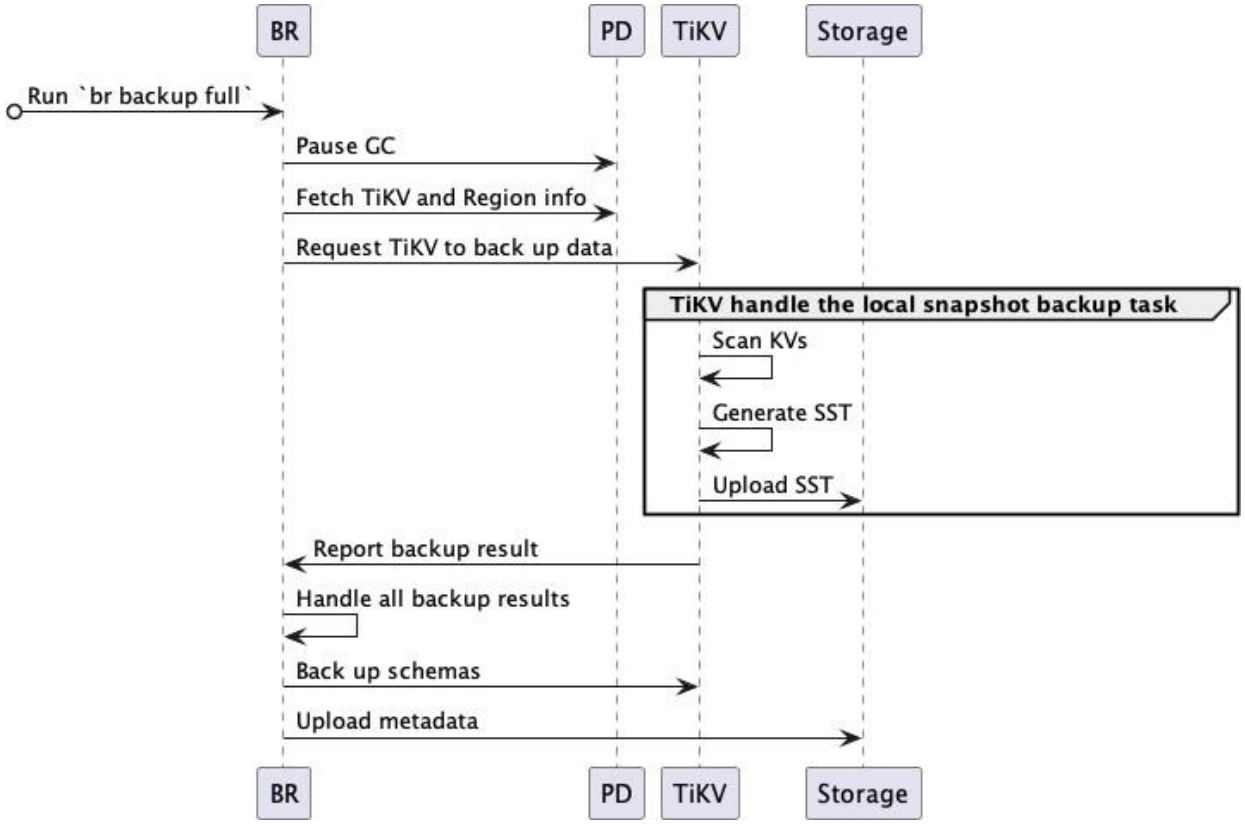


图 18: snapshot backup process design

完整的备份交互流程描述如下：

1. BR 接收备份命令 `br backup full`。
    - 获得备份快照点(backup ts)和备份存储地址。
2. BR 调度备份数据。
    - Pause GC：配置TiDB GC，防止要备份的数据被TiDB GC 机制回收。
    - Fetch TiKV and Region info：访问 PD，获取所有 TiKV 节点访问地址以及数据的Region 分布信息。
    - Request TiKV to back up data：创建备份请求，发送给TiKV 节点，备份请求包含backup ts、需要备份的region、备份存储地址。
3. TiKV 接受备份请求，初始化backup worker。
4. TiKV 备份数据。
    - Scan KVs：backup worker 从 Region (only leader) 读取backup ts 对应的数据。
    - Generate SST：backup worker 将读取到的数据保存到 SST 文件，存储在内存中。
    - Upload SST：backup worker 上传 SST 文件到备份存储中。
5. BR 从各个TiKV 获取备份结果。

- 如果局部数据因为Region 变动而备份失败，比如TiKV 节点故障，BR 将重试这些数据的备份。
- 如果任意数据被判断为不可重试的备份失败，则备份任务失败。
- 全部数据备份成功后，则在最后完成元信息备份。

6. BR 备份元信息。

   - Back up schemas：备份 table schema，同时计算 table data checksum。
   - Upload metadata：生成backup metadata，并上传到备份存储。 backup metadata 包含backup ts、 表和对应的备份文件、 data checksum 和 file checksum 等信息。

5.3.2.2.3 恢复流程

恢复集群快照备份数据的流程如下：

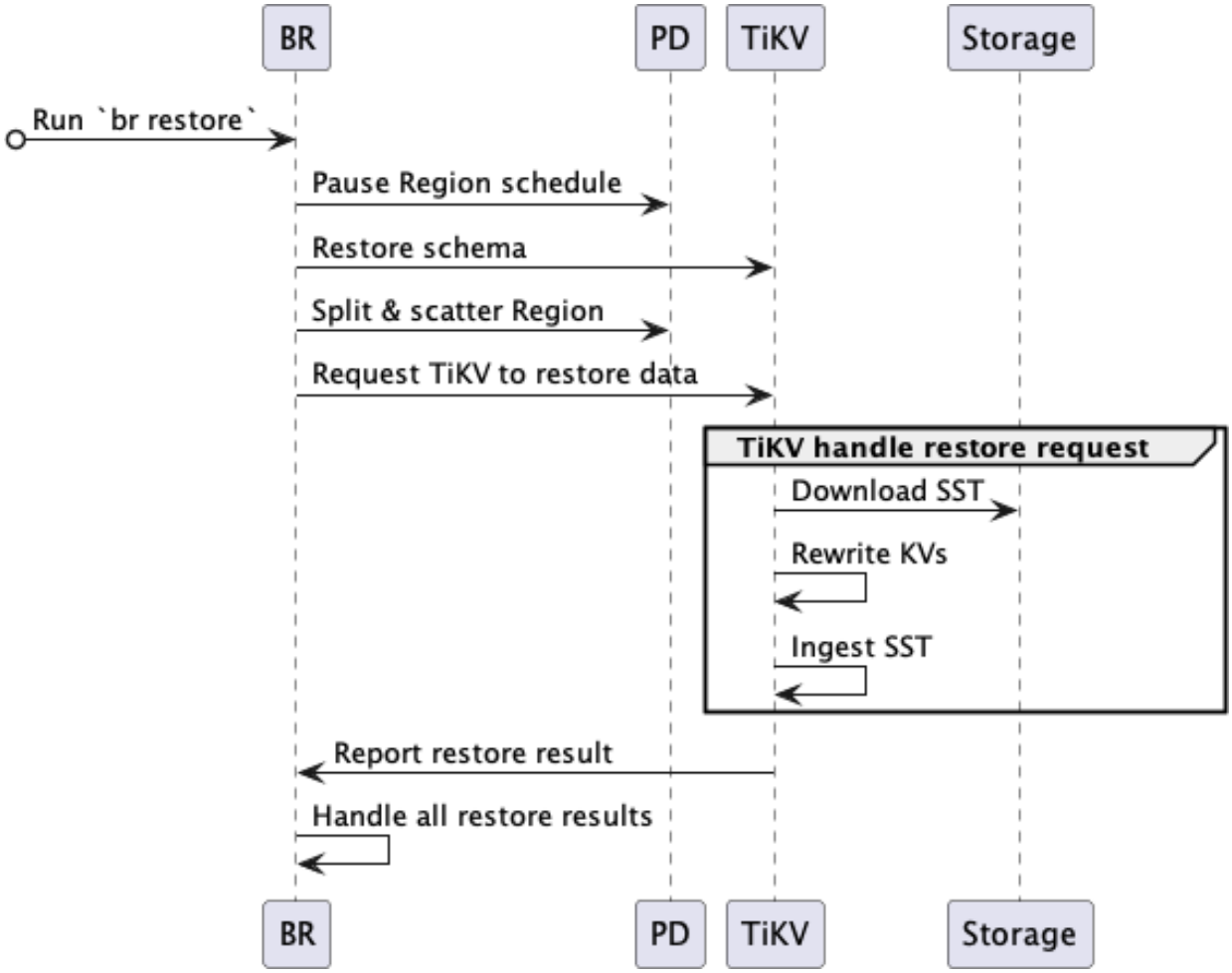


图 19: snapshot restore process design

完整的恢复交互流程描述如下：

1. BR 接收恢复命令 `br restore`。

- 获得快照备份数据存储地址、要恢复的database 或 table。
- 检查要恢复的table 是否存在及是否符合要求。

2. BR 调度恢复数据。
   - Pause Region schedule：请求 PD 在恢复期间关闭自动Region schedule。
   - Restore schema：读取备份数据的 schema、恢复的 database 和table（注意新建表的table ID 与备份数据可能不一样）。
   - Split & scatter Region：BR 基于备份数据信息，请求 PD 分配Region (split Region)，并调度 Region 均匀分布到存储节点上(scatter Region)。每个Region 都有明确的数据范围 [start key, end key)。
   - Request TiKV to restore data：根据 PD 分配的Region 结果，发送恢复请求到对应的TiKV 节点，恢复请求包含要恢复的备份数据及rewrite 规则。
3. TiKV 接受恢复请求，初始化restore worker。
   - restore worker 计算恢复数据需要读取的备份数据。
4. TiKV 恢复数据。
   - Download SST：restore worker 从备份存储中下载相应的备份数据到本地。
   - Rewrite KVs：restore worker 根据新建表table ID，对备份数据 kv 进行重写，即将原有的kv 编码中的table ID 替换为新创建的table ID。对 index ID，restore worker 也进行相同处理。
   - Ingest SST：restore worker 将处理好的 SST 文件 ingest 到 RocksDB 中。
   - Report restore result：restore worker 返回恢复结果给 BR。
5. BR 从各个TiKV 获取恢复结果。
   - 如果局部数据恢复因为 RegionNotFound 或 EpochNotMatch 等原因失败，比如TiKV 节点故障，BR 重试恢复这些数据。
   - 如果存在备份数据不可重试的恢复失败，则恢复任务失败。
   - 全部备份都恢复成功后，则整个恢复任务成功。

详细的快照数据备份恢与恢复流程设计，可以参考备份恢复设计方案。

5.3.2.2.4 备份文件

文件类型

快照备份会产生如下类型文件：

- SST 文件：存储TiKV 备份下来的数据信息。单个 SST 文件大小等于TiKV Region 的大小。
- backupmeta 文件：存储本次备份的元信息，包括备份文件数、备份文件的 Key 区间、备份文件大小和备份文件 Hash (sha256) 值。
- backup.lock 文件：用于防止多次备份到同一目录。

SST 文件的命名格式

当备份数据到Google Cloud Storage 或Azure Blob Storage 时，SST 文件以storeID_regionID_regionEpoch_keyHash_timestamp_cf 的格式命名。格式名的解释如下：

- storeID：TiKV 节点编号

- regionID：Region 编号
- regionEpoch：Region 版本号
- keyHash：Range startKey 的 Hash (sha256) 值，确保唯一性
- timestamp：TiKV 节点生成 SST 文件名时刻的 Unix 时间戳
- cf：RocksDB 的 ColumnFamily（只备份 cf 为 default 或 write 的数据）

当备份数据到Amazon S3 或网络盘上时，SST 文件以 regionID_regionEpoch_keyHash_timestamp_cf 的格式命名。

- regionID：Region 编号
- regionEpoch：Region 版本号
- keyHash：Range startKey 的 Hash (sha256) 值，确保唯一性
- timestamp：TiKV 节点生成 SST 文件名时刻的 Unix 时间戳
- cf：RocksDB 的 ColumnFamily（只备份 cf 为 default 或 write 的数据）

SST 文件存储格式

- 关于 SST 文件存储格式，可以参考RocksDB SST table 介绍。
- 关于 SST 文件中存储的备份数据编码格式，可以参考TiDB 表数据与 Key-Value 的映射关系。

备份文件目录结构

将数据备份到Google Cloud Storage 或 Azure Blob Storage 上时，SST 文件、backupmeta 文件和 backup.lock 文件在同一目录下。目录结构如下：

```
.
└── 20220621
    ├── backupmeta
    |── backup.lock
    ├── {storeID}-{regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── {storeID}-{regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    └── {storeID}-{regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
```

将数据备份到Amazon S3 或网络盘上时，SST 文件会根据 storeID 划分子目录。目录结构如下：

```
.
└── 20220621
    ├── backupmeta
    |── backup.lock
    ├── store1
    |   └── {regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── store100
    |   └── {regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── store2
    |   └── {regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── store3
    ├── store4
    └── store5
```

5.3.2.2.5 探索更多

- TiDB 快照备份与恢复使用指南

5.3.2.3 TiDB 日志备份与 PITR 功能架构

本文以使用 BR 工具进行备份与恢复为例，介绍 TiDB 集群日志备份与 Point-in-time recovery (PITR) 的架构设计与流程。

5.3.2.3.1 架构设计

日志备份和 PITR 的架构如下：

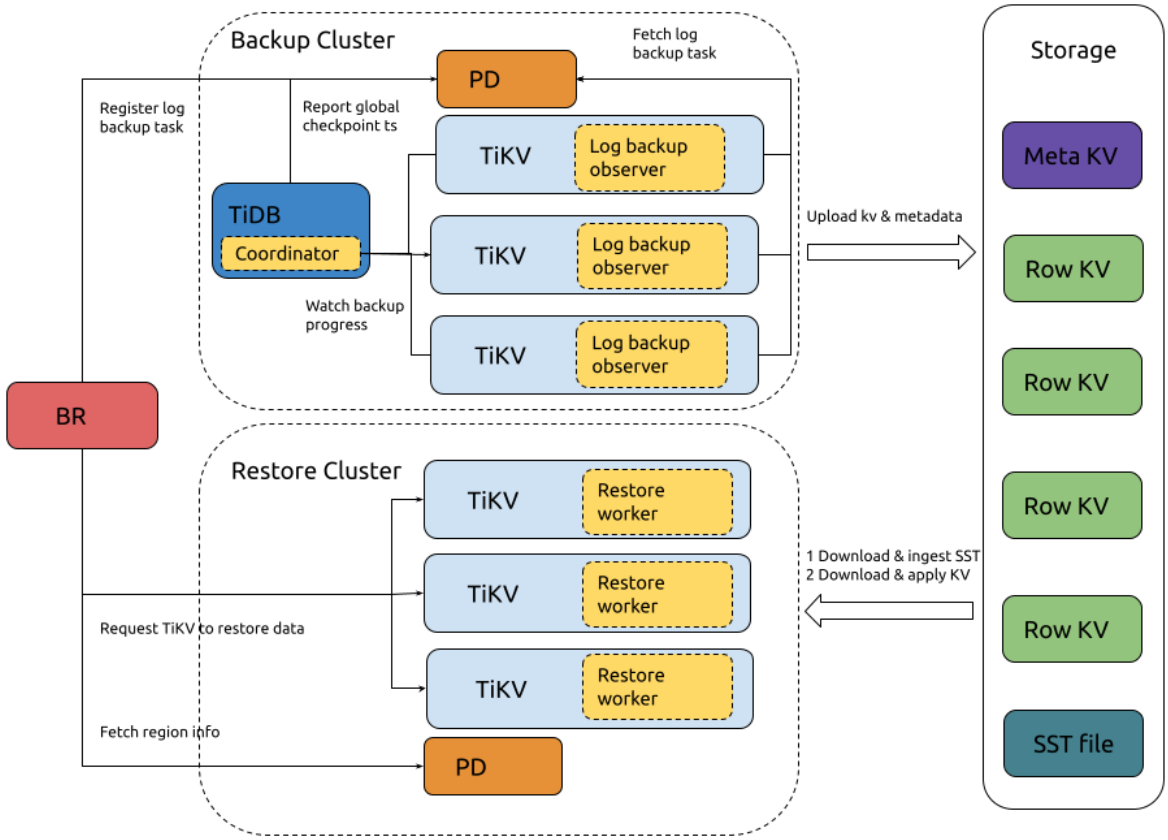


图 20: BR log backup and PITR architecture

5.3.2.3.2 日志备份

日志备份的流程如下：

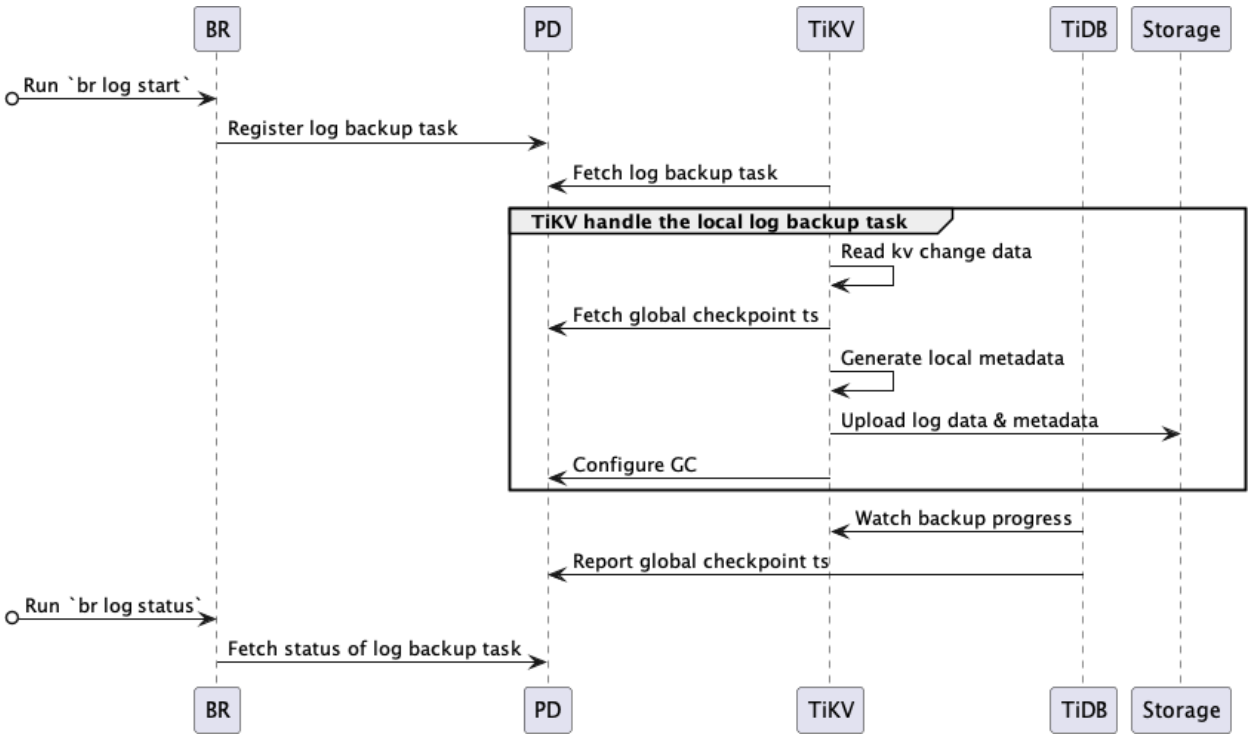


图 21: BR log backup process design

系统组件和关键概念：

- local metadata：表示单TiKV 节点备份下来的数据元信息，主要包括：local checkpoint ts、 global checkpoint ts、备份文件信息。
- local checkpoint ts (in local metadata)：表示这个TiKV 中所有小于local checkpoint ts 的日志数据已经备份到目标存储。
- global checkpoint ts：表示所有TiKV 中小于 global checkpoint ts 的日志数据已经备份到目标存储。它由运行在 TiDB 中的 Coordinator 模块收集所有TiKV 的 local checkpoint ts 计算所得，然后上报给 PD。
- TiDB Coordinator 组件：TiDB 集群的某个节点会被选举为 Coordinator，负责收集和计算整个日志备份任务的进度(global checkpoint ts)。该组件设计上无状态，在其故障后可以从存活的TiDB 节点中重新选出一个节点作为 Coordinator。
- TiKV log observer 组件：运行在 TiDB 集群的每个TiKV 节点，负责从 TiKV 读取和备份日志数据。TiKV 节点故障的话，该节点负责备份数据范围，在Region 重新选举后，会被其他TiKV 节点负责，这些节点会从 global checkpoint ts 重新备份故障范围的数据。

完整的备份交互流程描述如下：

1. BR 接收备份命令 `br log start`。

- 解析获取日志备份任务的checkpoint ts（日志备份起始位置）、备份存储地址。
- Register log backup task：在 PD 注册日志备份任务(log backup task)。

2. TiKV 监控日志备份任务的创建与更新。

- Fetch log backup task：每个TiKV 节点的log backup observer 监听 PD 中日志备份任务的创建与更新，然后备份该节点上在备份范围内的数据。

3. TiKV log backup observer 持续地备份 KV 变更日志。

- Read kv change data：读取 kv 数据变更，然后保存到自定义格式的备份文件中。
- Fetch global checkpoint ts：定期从 PD 查询 global checkpoint ts。
- Generate local metadata：定期生成 local metadata（包含local checkpoint ts、global checkpoint ts、备份文件信息）。
- Upload log data & metadata：定期将日志备份数据和local metadata 上传到备份存储中。
- Configure GC：请求 PD 阻止未备份的数据（大于 local checkpoint ts）被TiDB GC 机制回收掉。

4. TiDB Coordinator 监控日志备份进度。

- Watch backup progress：轮询所有TiKV 节点，获取各个Region 的备份进度(Region checkpoint ts)。
- Report global checkpoint ts：根据各个Region checkpoint ts，计算整个日志备份任务的进度(global checkpoint ts)，然后上报给 PD。

5. PD 持久化日志备份任务状态。可以通过 `br log status` 查询。

5.3.2.3.3 PITR

PITR 的流程如下：

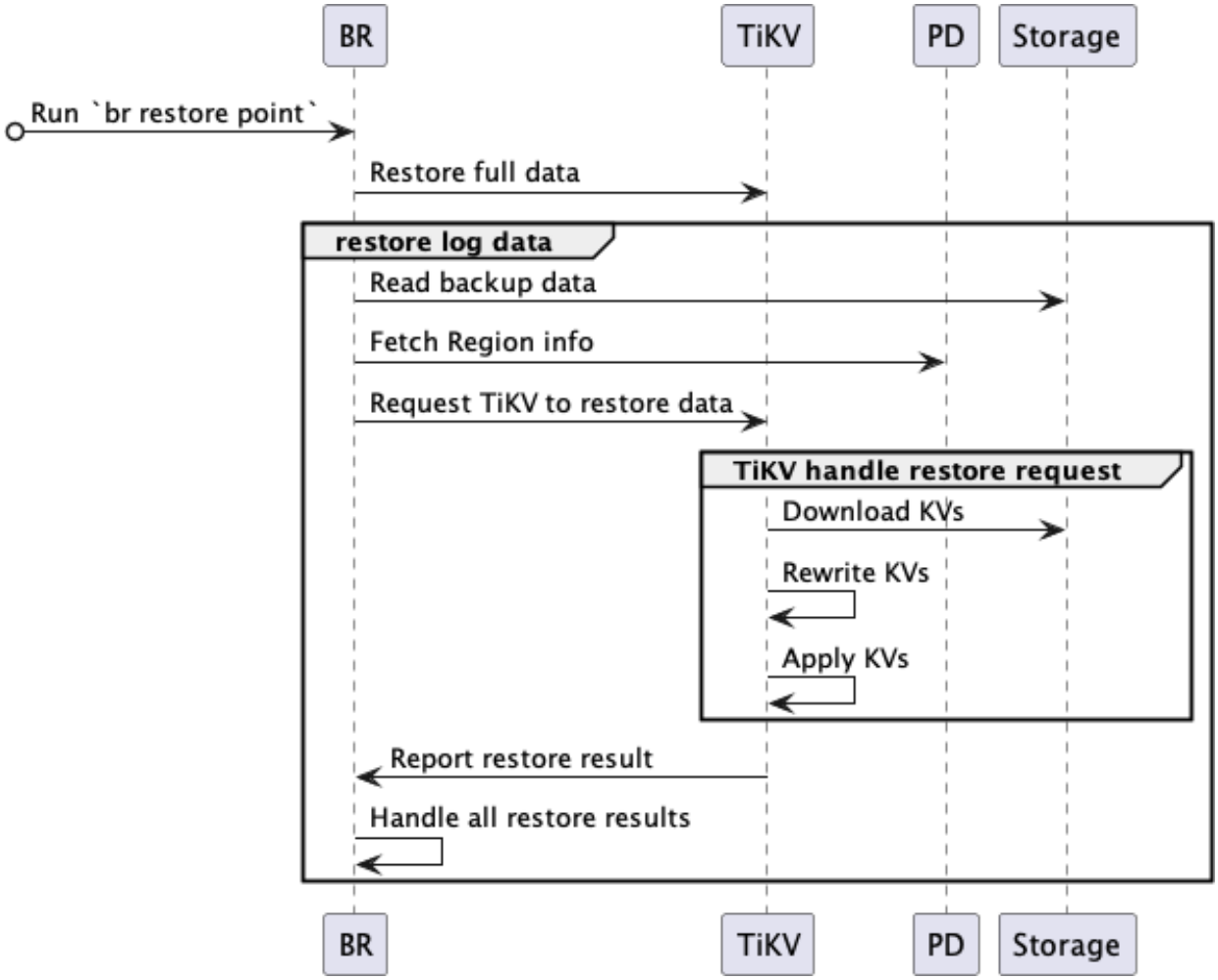


图 22: Point-in-time recovery process design

完整的 PITR 交互流程描述如下：

1. BR 接收恢复命令 `br restore point`。

- 解析获取全量备份数据地址、日志备份数据地址、恢复到的时间点。
- 查询备份数据中恢复数据对象（database 或 table），并检查要恢复的表是否存在并符合要求。

2. BR 恢复全量备份。

- 进行快照备份数据恢复，恢复流程参考恢复快照备份数据。

3. BR 恢复日志备份。

- Read backup data：读取日志备份数据，计算需要恢复的日志备份数据。

- Fetch Region info：访问 PD，获取所有 Region 和 KV range 的对应关系。
- Request TiKV to restore data：创建日志恢复请求，发送到对应的TiKV，日志恢复请求包含要恢复的日志备份数据信息。

4. TiKV 接受 BR 的恢复请求，初始化log restore worker。

- log restore worker 获取需要恢复的日志备份数据。

5. TiKV 恢复日志备份数据。

- Download KVs：log restore worker 根据日志恢复请求中要恢复的备份数据，从备份存储中下载相应的备份数据到本地。
- Rewrite KVs：log restore worker 根据恢复集群表的table ID 对备份数据的 kv 进行重写—— 将原有的kv 编码中的 table ID 替换为新创建的table ID。 对 index ID，log restore worker 也进行相同的处理。
- Apply KVs：log restore worker 将处理好的 kv 通过 raft 接口写入 store (RocksDB) 中。
- Report restore result：log restore worker 返回恢复结果给 BR。

6. BR 从各个TiKV 获取恢复结果。

- 如果局部数据恢复因为 `RegionNotFound` 或 `EpochNotMatch` 等原因失败，比如TiKV 节点故障，BR 重试恢复这些数据。
- 如果存在备份数据不可重试的恢复失败，则恢复任务失败。
- 全部备份数据都恢复成功后，则恢复任务成功。

5.3.2.3.4 日志备份文件

日志备份会产生如下类型文件：

- `{min_ts}-{uuid}.log` 文件：存储备份下来的 kv 数据变更记录。其中 `{min_ts}` 是该文件中所有 kv 数据变更记录数对应的最小ts；`{uuid}` 是生成该文件的时候随机生成的。
- `{checkpoint_ts}-{uuid}.meta` 文件：每个TiKV 节点每次上传日志备份数据时会生成一个该文件，保存本次上传的所有日志备份数据文件。其中 `{checkpoint_ts}` 是本节点的日志备份的checkpoint，所有TiKV 节点的最小的checkpoint 就是日志备份任务最新的checkpoint；`{uuid}` 是生成该文件的时候随机生成的。
- `{store_id}.ts` 文件：每个TiKV 节点每次上传日志备份数据时会使用 global checkpoint ts 更新该文件。其中 `{store_id}` 是 TiKV 的 store ID。
- `v1_stream_truncate_safepoint.txt` 文件：保存最近一次通过 `br log truncate` 删除日志备份数据后，存储中最早的日志备份数据对应的ts。

备份文件目录结构

```
.
├── v1
|   ├── backupmeta
|   |   ├── {min_restored_ts}-{uuid}.meta
|   |   ├── {checkpoint}-{uuid}.meta
```

```
|   ├── global_checkpoint
|   |   ├── {store_id}.ts
|   ├── {date}
|   |   ├── {hour}
|   |   |   ├── {store_id}
|   |   |   |   ├── {min_ts}-{uuid}.log
|   |   |   |   ├── {min_ts}-{uuid}.log
├── v1_stream_truncate_safepoint.txt
```

具体示例如下：

```
.
├── v1
|   ├── backupmeta
|   |   ├── ...
|   |   ├── 435213818858112001-e2569bda-a75a-4411-88de-f469b49d6256.meta
|   |   ├── 435214043785779202-1780f291-3b8a-455e-a31d-8a1302c43ead.meta
|   |   ├── 435214443785779202-224f1408-fff5-445f-8e41-ca4fcfbd2a67.meta
|   ├── global_checkpoint
|   |   ├── 1.ts
|   |   ├── 2.ts
|   |   ├── 3.ts
|   ├── 20220811
|   |   ├── 03
|   |   |   ├── 1
|   |   |   |   ├── ...
|   |   |   |   ├── 435213866703257604-60fcbdb6-8f55-4098-b3e7-2ce604dafe54.log
|   |   |   |   ├── 435214023989657606-72ce65ff-1fa8-4705-9fd9-cb4a1e803a56.log
|   |   |   ├── 2
|   |   |   |   ├── ...
|   |   |   |   ├── 435214102632857605-11deba64-beff-4414-bc9c-7a161b6fb22c.log
|   |   |   |   ├── 435214417205657604-e6980303-cbaa-4629-a863-1e745d7b8aed.log
|   |   |   ├── 3
|   |   |   |   ├── ...
|   |   |   |   ├── 435214495848857605-7bf65e92-8c43-427e-b81e-f0050bd40be0.log
|   |   |   |   ├── 435214574492057604-80d3b15e-3d9f-4b0c-b133-87ed3f6b2697.log
├── v1_stream_truncate_safepoint.txt
```

5.3.3 使用 BR 进行备份与恢复

5.3.3.1 TiDB 备份与恢复功能使用概述

本文介绍使用TiDB 备份与恢复功能的最佳实践，包括如何选择备份方案、如何管理备份数据，以及如何安装和部署备份恢复工具。

5.3.3.1.1 使用概览

深入TiDB 备份和恢复功能使用之前，建议先了解推荐的集群备份和恢复方案。

如何备份数据？

TiDB 支持两种类型的备份，应该使用哪种备份？ 全量备份包含集群某个时间点的全量数据，日志备份包含业务写入在TiDB 产生的数据变更记录。推荐这两种备份方式一起使用：

- 开启日志备份：运行 `br log start` 命令来启动日志备份任务，任务会在每个TiKV 节点上持续运行，以小批量的形式定期将TiDB 变更数据备份到指定存储中。
- 定期执行快照（全量）备份：运行 `br backup full` 命令来备份集群快照到备份存储，例如在每天零点进行集群快照备份。

如何管理备份数据？

BR 只提供备份和恢复的基础功能，尚不支持备份管理的功能，因此你需要自行规划备份数据的管理事项，可能包含以下的问题：

- 选择哪种备份存储系统？
- 数据备份的时候，备份数据应该放在什么目录下？
- 全量备份和日志备份的数据目录如何组织？
- 如何处理存储系统中历史备份数据？

下面是处理这些问题的推荐方式：

选择备份存储

Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 是推荐的存储系统选择，使用这些系统，你无需担心备份容量、备份带宽规划等。

如果TiDB 集群部署在自建机房中，则推荐以下方式：

- 搭建 MinIO 作为备份存储系统，使用 S3 协议将数据备份到 MinIO 中。
- 挂载 NFS（如 NAS）盘到 br 工具和所有的TiKV 实例，使用 POSIX file system 接口将备份数据写入对应的 NFS 目录中。

> 注意：
>
> 如果没有挂载 NFS 到 br 工具或TiKV 节点，或者使用了支持 S3、 GCS 或 Azure Blob Storage 协议的远端存储，那么br 工具备份的数据会在各个TiKV 节点生成。注意这不是推荐的br 工具使用方式，因为备份数据会分散在各个节点的本地文件系统中，聚集这些备份数据可能会造成数据冗余和运维上的麻烦，而且在不聚集这些数据便直接恢复的时候会遇到 `SST file not found` 报错。

组织备份数据目录

- 全量备份和日志备份保存在相同的目录下，方便统一管理，例如 backup-${cluster-id}。
- 每个全量备份保存到命名带有备份日期的目录下，例如 backup-${cluster-id}/fullbackup-202209081330。
- 日志备份数据保存在一个固定目录下，例如 backup-${cluster-id}/logbackup。日志备份程序会在 logbackup 目录中每天切分出来一个新的子目录来区分每天的日志备份数据。

处理历史备份数据

假设你设置了备份保留期，即保存固定时间的备份数据，比如 7 天。请注意备份保留期的概念，后面使用教程中也会多次遇到。

- 进行 PITR 不仅需要恢复时间点之前的全量备份，还需要全量备份和恢复时间点之间的日志备份，因此，对于超过备份保留期的日志备份，应执行 br log truncate 命令删除指定时间点之前的备份。建议只清理全量快照之前的日志备份。
- 对于超过备份保留期的全量备份，建议直接删除或者归档全量备份的目录。

如何恢复数据？

- 如果你只有全量备份数据，或者想恢复某个确定的全量备份，那么可以使用 br restore 恢复指定的全量备份。
- 如果你按照以上推荐的的方式进行备份，那么你可以使用 br restore point 恢复到备份保留期内任意时间点。

5.3.3.1.2 部署和使用 BR

使用备份恢复功能的部署要求如下：

- BR、TiKV 节点和备份存储系统需要提供大于备份速度的的网络带宽。当集群特别大的时候，备份和恢复速度上限受限于备份网络的带宽。
- 备份存储系统还需要提供足够的写入/读取性能(IOPS)，否则它有可能成为备份恢复时的性能瓶颈。
- TiKV 节点需要为备份准备至少额外的两个 CPU core 和高性能的磁盘，否则备份将对集群上运行的业务产生影响。
- 推荐 br 工具运行在 8 核 +/16 GB+ 的节点上。

目前支持以下几种方式来使用 BR。

通过命令行工具（推荐）

TiDB 支持使用br 工具进行备份恢复。

- 安装方法可以使用 TiUP 在线安装：tiup install br。
- 了解如何使用 br 命令行工具进行备份与恢复，请参阅：
  - TiDB 快照备份与恢复功能使用
  - TiDB 日志备份与 PITR 功能使用
  - TiDB 集群备份与恢复实践示例

通过 SQL 语句

TiDB 支持使用 SQL 语句进行全量快照备份和恢复：

- BACKUP 进行全量快照数据备份。
- RESTORE 进行快照备份恢复。
- SHOW BACKUPS|RESTORES 查看备份恢复的进度。

在 Kubernetes 环境下通过TiDB Operator

在 Kubernetes 环境下，支持通过TiDB Operator 支持以 S3、 GCS、 Azure blob storage 作为备份存储，并从这些存储系统中恢复备份数据。使用文档请参阅使用TiDB Operator 进行备份恢复。

5.3.3.1.3 探索更多

- TiDB 备份与恢复概述
- TiDB 备份与恢复功能架构

5.3.3.2 TiDB 快照备份与恢复使用指南

本文介绍如何使用br 命令行工具进行TiDB 快照备份和恢复。使用前，请先安装br 命令行工具。

快照备份是集群全量备份的一种实现。它基于TiDB 的多版本并发控制(MVCC) 实现，将指定快照包含的所有数据备份到目标存储中。备份下来的数据大小约等于集群（压缩后的）单副本数据大小。备份完成之后，你可以在一个空集群或不存在数据冲突（相同 schema 或 table）的集群执行快照备份恢复，将集群恢复到快照备份时的数据状态，同时恢复功能会依据集群副本设置恢复出多副本。

除了基础的备份和恢复功能，快照备份和恢复还提供以下功能：

- 备份指定时间点的快照数据
- 恢复指定数据库或表的数据

5.3.3.2.1 对集群进行快照备份

> 注意：
>
> - 以下场景采用Amazon S3 Access key 和 Secret key 授权方式来进行模拟。如果使用 IAM Role 授权，需要设置 `--send-credentials-to-tikv` 为 `false`。
> - 如果使用不同存储或者其他授权方式，请参考备份存储来进行参数调整。

使用 `br backup full` 可以进行一次快照备份。该命令的详细使用帮助可以通过执行 `br backup full --help` 查看。

```
tiup br backup full --pd "${PD_IP}:2379" \
    --backupts '2022-09-08 13:30:00' \
    --storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=$
        ′→ {secret-access-key}" \
    --ratelimit 128 \
```

以上命令中：

- `--backupts`：快照对应的物理时间点，格式可以是TSO 或者时间戳，例如 `400036290571534337` 或者 `2018-05-11 01:42:23`。如果该快照的数据被垃圾回收 (GC) 了，那么 `br backup` 命令会报错并退出。如果你没有指定该参数，那么br 会选取备份开始的时间点所对应的快照。
- `--storage`：数据备份到的存储地址。快照备份支持以 Amazon S3、 Google Cloud Storage、 Azure Blob Storage 为备份存储，以上命令以Amazon S3 为示例。详细存储地址格式请参考备份存储 URI 配置。
- `--ratelimit`：每个 TiKV 备份数据的速度上限，单位为 MiB/s。

在快照备份过程中，终端会显示备份进度条。在备份完成后，会输出备份耗时、速度、备份数据大小等信息。

```
Full Backup <---------------------------------------------------------------------------------->
    ′→ 100.00%
Checksum <------------------------------------------------------------------------------------>
    ′→ 100.00%
*** ["Full Backup success summary"] *** [backup-checksum=3.597416ms] [backup-fast-checksum
    ′→ =2.36975ms] *** [total-take=4.715509333s] [BackupTS=435844546560000000] [total-kv=1131]
    [ ′→ total-kv-size=250kB] [average-speed=53.02kB/s] [backup-data-size(after-compressed)=71.33
    ′→ kB] [Size=71330]
```

5.3.3.2.2 查询快照备份的时间点信息

出于管理备份数的需要，如果你需要查看某个快照备份对应的快照物理时间点，可以执行下面的命令：

```
tiup br validate decode --field="end-version" \
--storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ′→ secret-access-key}" | tail -n1
```

结果输出如下，对应物理时间 `2022-09-08 13:30:00 +0800 CST`：

```
435844546560000000
```

5.3.3.2.3 恢复快照备份数据

如果你需要恢复备份的快照数据，则可以使用 `br restore full`。该命令的详细使用帮助可以通过执行 `br restore full --help` 查看。

将上文备份的快照数据恢复到目标集群：

```
tiup br restore full --pd "${PD_IP}:2379" \
--storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    '→ secret-access-key}"
```

在恢复快照备份数据过程中，终端会显示恢复进度条。在完成恢复后，会输出恢复耗时、速度、恢复数据大小等信息。

```
Full Restore <---------------------------------------------------------------------------------->
    '→ 100.00%
*** ["Full Restore success summary"] *** [total-take=4.344617542s] [total-kv=5] [total-kv-size
    '→ =327B] [average-speed=75.27B/s] [restore-data-size(after-compressed)=4.813kB] [Size=4813]
    '→  [BackupTS=435844901803917314]
```

恢复备份数据中指定库表的数据

br 命令行工具支持只恢复备份数据中指定库、表的部分数据，该功能用于在恢复过程中过滤不需要的数据。

恢复单个数据库的数据

要将备份数据中的某个数据库恢复到集群中，可以使用 `br restore db` 命令。以下示例只恢复 `test` 库的数据：

```
tiup br restore db --pd "${PD_IP}:2379" \
--db "test" \
--storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    '→ secret-access-key}"
```

以上命令中 `--db` 选项指定了需要恢复的数据库名。

恢复单张表的数据

要将备份数据中的某张数据表恢复到集群中，可以使用 `br restore table` 命令。以下示例只恢复 `test.
'→ usertable` 表的数据：

```
tiup br restore table --pd "${PD_IP}:2379" \
--db "test" \
--table "usertable" \
--storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    '→ secret-access-key}"
```

以上命令中 `--db` 选项指定了需要恢复的数据库名，`--table` 选项指定了需要恢复的表名。

使用表库过滤功能恢复部分数据

要通过复杂的过滤条件恢复多个表，可以使用 `br restore full` 命令，并用 `--filter` 或 `-f` 指定表库过滤的条件。以下示例恢复符合 `db*.tbl*` 条件的表的数据：

```
tiup br restore full --pd "${PD_IP}:2379" \
--filter 'db*.tbl*' \
--storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    '→ secret-access-key}"
```

恢复 mysql 数据库下的表

自 br v5.1.0 开始，快照备份会备份mysql schema 下的系统表数据，而不会默认恢复这些数据。自 br v6.2.0 开始，在设置 --with-sys-table 下，恢复数据时将同时恢复部分系统表相关数据。

可恢复的部分系统表：

```
+-----------------------------------+
| mysql.columns_priv                |
| mysql.db                          |
| mysql.default_roles               |
| mysql.global_grants               |
| mysql.global_priv                 |
| mysql.role_edges                  |
| mysql.tables_priv                 |
| mysql.user                        |
+                                   +
 -----------------------------------
```

不能恢复以下系统表：

- 统计信息表(mysql.stat_*)
- 系统变量表(mysql.tidb、 mysql.global_variables)
- 其他系统表

当恢复系统权限相关数据的时候，请注意：

- BR 不会恢复 user 为 cloud_admin 并且 host 为 '%' 的用户数据，该用户是TiDB Cloud 预留用户。请不要在你的环境中创建 cloud_admin 的用户或者角色，因为依赖 cloud_admin 的用户的权限将不能被完整恢复。
- 在恢复数据前 BR 会检查目标集群的系统表是否跟备份数据中的系统表兼容。这里的兼容是指满足以下所有条件：
  - 目标集群需要存在备份中的系统权限表。
  - 目标集群系统权限表列数需要与备份数据中一致，列的顺序可以有差异。
  - 目标集群系统权限表的列需要与备份数据兼容。如果为带长度类型（包括整型、字符串等类型），前者长度需大于或等于后者，如果为 ENUM 类型，则应该为后者超集。

5.3.3.2.4 性能与影响

快照备份的性能与影响

TiDB 备份功能对集群性能（事务延迟和 QPS）有一定的影响，但是可以通过调整备份的线程数backup.num-
'→ threads，以及增加集群配置，来降低备份对集群性能的影响。

为了更加具体说明备份对集群的影响，下面列举了多次快照备份测试结论来说明影响的范围：

- （使用 5.3 及之前版本）在默认配置下，单TiKV 存储节点上备份线程数量是节点 CPU 总数量的 75% 时，QPS 会下降到备份之前的 35% 左右。
- （使用 5.4 及以后版本）单 TiKV 存储节点上备份的线程数量不大于 8、集群总 CPU 利用率不超过 80% 时，备份任务对集群（无论读写负载）影响最大在 20% 左右。

- （使用 5.4 及以后版本）单 TiKV 存储节点上备份的线程数量不大于 8、集群总 CPU 利用率不超过 75% 时，备份任务对集群（无论读写负载）影响最大在 10% 左右。
- （使用 5.4 及以后版本）单 TiKV 存储节点上备份的线程数量不大于 8、集群总 CPU 利用率不超过 60% 时，备份任务对集群（无论读写负载）几乎没有影响。

你可以通过如下方案手动控制备份对集群性能带来的影响。但是，这两种方案在减少备份对集群的影响的同时，也会降低备份任务的速度。

- 使用 `--ratelimit` 参数对备份任务进行限速。请注意，这个参数限制的是把备份文件存储到外部存储的速度。计算备份文件的大小时，请以备份日志中的 `backup data size(after compressed)` 为准。
- 调节TiKV 配置项`backup.num-threads`，限制备份任务使用的工作线程数量。内部测试数据表明，当备份的线程数量不大于 8、集群总 CPU 利用率不超过 60% 时，备份任务对集群（无论读写负载）几乎没有影响。

通过限制备份的线程数量可以降低备份对集群性能的影响，但是这会影响到备份的性能，以上的多次备份测试结果显示，单TiKV 存储节点上备份速度和备份线程数量呈正比。在线程数量较少的时候，备份速度约为 20 MiB/线程数。例如，单 TiKV 节点 5 个备份线程可达到 100 MiB/s 的备份速度。

快照恢复的性能与影响

- TiDB 恢复的时候会尽可能打满TiKV CPU、磁盘 IO、网络带宽等资源，所以推荐在空的集群上执行备份数据的恢复，避免对正在运行的业务产生影响。
- 备份数据的恢复速度与集群配置、部署、运行的业务都有比较大的关系。在内部多场景仿真测试中，单 TiKV 存储节点上备份数据恢复速度能够达到 100 MiB/s。在不同用户场景下，快照恢复的性能和影响应以实际测试结论为准。

5.3.3.2.5 探索更多

- TiDB 集群备份与恢复实践示例
- br 命令行手册
- 快照备份与恢复架构设计

5.3.3.3 TiDB 日志备份与 PITR 使用指南

全量备份包含集群某个时间点的全量数据，但是不包含其他时间点的更新数据，而TiDB 日志备份能够将业务写入TiDB 的数据记录及时备份到指定存储中。如果你需要灵活的选择恢复的时间点，即实现 Point-in-time recovery (PITR)，可以开启日志备份，并定期执行快照（全量）备份。

使用br 命令行工具备份与恢复数据前，请先安装br 命令行工具。

5.3.3.3.1 对集群进行备份

开启日志备份

注意：

- 以下场景采用Amazon S3 Access key 和 Secret key 授权方式来进行模拟。如果使用 IAM Role 授权，需要设置 `--send-credentials-to-tikv` 为 `false`。
- 如果使用不同存储或者其他授权方式，请参考备份存储来进行参数调整。

执行 `br log start` 命令启动日志备份任务，一个集群只能启动一个日志备份任务。

```
tiup br log start --task-name=pitr --pd "${PD_IP}:2379" \
--storage  's3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"'
```

日志备份任务启动后，会在TiDB 集群后台持续地运行，直到你手动将其暂停。在这过程中，TiDB 变更数据将以小批量的形式定期备份到指定存储中。如果你需要查询日志备份任务当前状态，执行如下命令：

```
tiup br log status --task-name=pitr --pd "${PD_IP}:2379"
```

日志备份任务状态输出如下：

```
● Total 1 Tasks.
> #1 <
    name: pitr
    status: ● NORMAL
    start: 2022-05-13 11:09:40.7 +0800
      end: 2035-01-01 00:00:00 +0800
    storage: s3://backup-101/log-backup
    speed(est.): 0.00 ops/s
checkpoint[global]: 2022-05-13 11:31:47.2 +0800; gap=4m53s
```

定期执行全量备份

快照备份功能可作为全量备份的方法，运行 `br backup full` 命令可以按照固定的周期（比如 2 天）进行全量备份。

```
tiup br backup full --pd "${PD_IP}:2379" \
--storage  's3://backup-101/snapshot-${date}?access-key=${access-key}&secret-access-key=${secret-
    ↪ access-key}"'
```

5.3.3.3.2 进行 PITR

如果你想恢复到备份保留期内的任意时间点，可以使用 `br restore point` 命令。执行该命令时，你需要指定要恢复的时间点、恢复时间点之前最近的快照备份以及日志备份数据。`br` 命令行工具会自动判断和读取恢复需要的数据，然后将这些数据依次恢复到指定的集群。

```
br restore point --pd "${PD_IP}:2379" \
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ′→ key}"' \
--full-backup-storage='s3://backup-101/snapshot-${date}?access-key=${access-key}&secret-access-
    ′→ key=${secret-access-key}"' \
--restored-ts '2022-05-15 18:00:00+0800'
```

恢复期间，可通过终端中的进度条查看进度，如下。恢复分为两个阶段：全量恢复(Full Restore) 和日志恢复 (Restore Meta Files 和Restore KV Files) 。每个阶段完成恢复后，br 命令行工具都会输出恢复耗时和恢复数据大小等信息。

```
Full Restore
    ′→ <
    ′→  100.00%
*** ["Full Restore success summary"] ****** [total-take=xxx.xxxs] [restore-data-size(after-
    ′→ compressed)=xxx.xxx] [Size=xxxx] [BackupTS={TS}] [total-kv=xxx] [total-kv-size=xxx]
    [ ′→ average-speed=xxx]
Restore Meta Files
    ′→ <
    ′→  100.00%
Restore KV Files
    ′→ <
    ′→  100.00%
*** ["restore log success summary"] [total-take=xxx.xx] [restore-from={TS}] [restore-to={TS}] [
    ′→ total-kv-count=xxx] [total-size=xxx]
```

5.3.3.3.3 清理过期的日志备份数据

如使用指南概览所述：

- 进行 PITR 不仅需要恢复时间点之前的全量备份，还需要全量备份和恢复时间点之间的日志备份，因此，对于超过备份保留期的日志备份，应执行 `br log truncate` 命令删除指定时间点之前的备份。建议只清理全量快照之前的日志备份。

你可以按照以下步骤清理超过备份保留期的备份数据：

1. 查找备份保留期之外的最近一次全量备份。
2. 使用 `validate` 指令获取该备份对应的时间点。假如需要清理 2022/09/01 之前的备份数据，则应查找该日期之前的最近一次全量备份，且保证它不会被清理。

```
FULL_BACKUP_TS=`tiup br validate decode --field="end-version" --storage "s3://backup-101/
    ′→ snapshot-${date}?access-key=${access-key}&secret-access-key=${secret-access-key}"|
    ′→ tail -n1`
```

3. 清理该快照备份 FULL_BACKUP_TS 之前的日志备份数据。

```
tiup br log truncate --until=${FULL_BACKUP_TS} --storage='s3://backup-101/logbackup?access-
    ↪ key=${access-key}&secret-access-key=${secret-access-key}"'
```

4. 清理该快照备份 FULL_BACKUP_TS 之前的快照备份数据。

```
rm -rf s3://backup-101/snapshot-${date}
```

5.3.3.3.4 PITR 的性能与影响

能力指标

- PITR 恢复速度，平均到单台TiKV 节点：全量恢复为 280 GB/h，日志恢复为 30 GB/h
- 使用 `br log truncate` 清理过期的日志备份数据速度为 600 GB/h

> 注意：
>
> 以上功能指标是根据下述两个场景测试得出的结论，如有出入，建议以实际测试结果为准：
>
> - 全量恢复速度= 全量恢复数据量/（时间*TiKV 数量）
> - 日志恢复速度= 日志恢复总量/（时间*TiKV 数量）
>
> 其中全量恢复数据量，是指单个副本中所有 KV 的逻辑大小，并不代表实际恢复的数据量。BR 恢复数据时会根据集群设置的副本数来恢复全部副本，当副本数越多时，实际恢复的数据量也就越多。所有测试集群默认设置 3 副本。如果想提升整体恢复的性能，可以通过根据实际情况调整TiKV 配置文件中的`import.num-threads` 配置项以及 BR 命令的`concurrency` 参数。

测试场景 1（TiDB Cloud 上部署）

- TiKV 节点（8 core，16 GB 内存）数量：21
- TiKV 配置项 `import.num-threads`：8
- BR 命令参数 `concurrency`：128
- Region 数量：183,000
- 集群新增日志数据：10 GB/h
- 写入(INSERT/UPDATE/DELETE) QPS：10,000

测试场景 2（本地部署）

- TiKV 节点（8 core，64 GB 内存）数量：6
- TiKV 配置项 `import.num-threads`：8
- BR 命令参数 `concurrency`：128
- Region 数量：50,000
- 集群新增日志数据：10 GB/h
- 写入(INSERT/UPDATE/DELETE) QPS：10,000

5.3.3.3.5 探索更多

- TiDB 集群备份与恢复实践示例
- br 命令行手册
- 日志备份与 PITR 架构设计

5.3.3.4 TiDB 备份与恢复实践示例

TiDB 快照备份与恢复使用指南及TiDB 日志备份与 PITR 使用指南系统介绍了TiDB 提供的备份与恢复方案，即快照（全量）备份与恢复、日志备份和恢复到指定时间点(Point-in-time recovery, PITR)。本文档将基于具体的使用场景，介绍如何快速上手使用TiDB 的备份与恢复方案。

介绍具体操作前，设想有如下使用场景，你在AWS 部署了一套TiDB 生产集群，业务团队提出如下需求：

- 及时备份用户数据变更，在数据库遭遇异常情况时，能够以最小的数据丢失代价（容忍异常前几分钟内的用户数据丢失）快速地恢复业务。
- 每个月不定期进行业务审计。接收到审计请求后，提供一个数据库来查询审计要求的一个月内某个时间点的数据。

通过TiDB 提供的 PITR 功能，你可以满足业务团队的需求。

5.3.3.4.1 部署TiDB 集群和br 命令行工具

使用 PITR 功能，需要部署 v6.2.0 或以上版本的TiDB 集群，并且更新br 命令行工具到与TiDB 集群相同的版本，本文假设使用的是 v7.0.0 版本。

下表介绍了在TiDB 集群中使用日志备份功能的推荐配置。

组件	CPU	内存	硬盘类型	AWS 机型	实例数量
TiDB	8 核 +	16 GB+	SAS	c5.2xlarge	2
PD	8 核 +	16 GB+	SSD	c5.2xlarge	3
TiKV	8 核 +	32 GB+	SSD	m5.2xlarge	3
br cli	8 核 +	16 GB+	SAS	c5.2xlarge	1
监控	8 核 +	16 GB+	SAS	c5.2xlarge	1

> 注意：
>
> - br 命令行工具执行备份恢复功能需要访问 PD 和 TiKV，请确保br 命令行工具与所有 PD 和 TiKV 连接正常。
> - br 命令行工具与 PD 所在服务器时区需要相同。

使用TiUP 部署或升级TiDB 集群：

- 如果没有部署TiDB 集群，请部署TiDB 集群。

- 如果已经部署的TiDB 集群版本低于 v6.2.0，请升级 TiDB 集群。

使用TiUP 安装或升级br 命令行工具：

- 安装：

```
tiup install br:v7.0.0
```

- 升级：

```
tiup update br:v7.0.0
```

5.3.3.4.2 配置备份存储(Amazon S3)

在开始备份任务之前需要准备好备份存储，包括：

1. 准备用于存放备份数据的 S3 bucket 和目录；
2. 配置访问 S3 中备份目录的权限；
3. 规划备份数据保存的目录结构。

配置备份存储的步骤如下：

1. 在 S3 创建用于保存备份数据的目录 `s3://tidb-pitr-bucket/backup-data`。

    1. 创建bucket。你也可以选择已有的 S3 bucket 来保存备份数据。如果没有可用的 bucket，可以参照 AWS 官方文档创建一个 S3 Bucket。本文使用的 bucket 名为 `tidb-pitr-bucket`。
    2. 创建备份数据总目录。在上一步创建的bucket（例如 `tidb-pitr-bucket`）下创建目录 `backup-data`，参考AWS 官方文档。

2. 配置 br 命令行工具和TiKV 访问 S3 中的备份目录的权限。本文推荐使用最安全的IAM 访问方式，配置过程可以参考控制存储桶访问。权限要求如下：

    - 备份集群的TiKV 和br 命令行工具需要的`s3://tidb-pitr-bucket/backup-data` 权限：`s3:ListBucket`、`s3:PutObject` 和 `s3:AbortMultipartUpload`。
    - 恢复集群的 TiKV 和 br 命令行工具需要 `s3://tidb-pitr-bucket/backup-data` 的最小权限：`s3:ListBucket`、`s3:GetObject` 和 `s3:PutObject`。

3. 规划备份数据保存的目录结构，以及快照（全量）备份和日志备份的目录。

    - 所有快照备份保存在 `s3://tidb-pitr-bucket/backup-data/snapshot-${date}` 目录下，`${date}` 为快照备份开始的时间点，如在 2022/05/12 00:01:30 开始的快照备份保存为 `s3://tidb-pitr-bucket/backup-data/snapshot-20220512000130`。
    - 日志备份保存在 `s3://tidb-pitr-bucket/backup-data/log-backup/` 目录下。

5.3.3.4.3 确定备份策略

为了满足业务最小数据丢失、快速恢复、一个月内任意时间点审计需求，你可以制定如下备份策略：

- 运行日志备份，持续不断备份数据库数据变更；
- 每隔两天在零点左右进行一次快照备份；
- 保存 30 天内的快照备份和日志备份数据，清理超过 30 天的备份数据。

5.3.3.4.4 执行日志备份

启动日志备份任务后，日志备份进程会在TiKV 集群运行，持续不断将数据库变更数据备份到 S3 中。日志备份任务启动命令：

```
tiup br log start --task-name=pitr --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/log-backup'
```

启动日志备份任务后，可以查询日志备份任务状态：

```
tiup br log status --task-name=pitr --pd="${PD_IP}:2379"

● Total 1 Tasks.
> #1 <
    name: pitr
    status: ● NORMAL
    start: 2022-05-13 11:09:40.7 +0800
      end: 2035-01-01 00:00:00 +0800
    storage: s3://tidb-pitr-bucket/backup-data/log-backup
    speed(est.): 0.00 ops/s
checkpoint[global]: 2022-05-13 11:31:47.2 +0800; gap=4m53s
```

5.3.3.4.5 执行快照备份

通过自动化运维工具（如crontab）设置定期的快照备份任务，例如：每隔两天在零点左右进行一次快照（全量）备份。下面是两次备份的示例：

- 在 2022/05/14 00:00:00 执行一次快照备份：

```
tiup br backup full --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/snapshot-20220514000000'  \
--backupts='2022/05/14 00:00:00'
```

- 在 2022/05/16 00:00:00 执行一次快照备份：

```
tiup br backup full --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/snapshot-20220516000000'  \
--backupts='2022/05/16 00:00:00'
```

5.3.3.4.6 执行 PITR

假设你接到需求，要准备一个集群查询 2022/05/15 18:00:00 时间点的用户数据。此时，你可以制定 PITR 方案，恢复 2022/05/14 的快照备份和该快照到 2022/05/15 18:00:00 之间的日志备份数据，从而收集到目标数据。执行命令如下：

```
tiup br restore point --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/log-backup'  \
--full-backup-storage='s3://tidb-pitr-bucket/backup-data/snapshot-20220514000000'  \
--restored-ts '2022-05-15 18:00:00+0800'

Full Restore
    '→ <
    '→  100.00%
[2022/05/29 18:15:39.132 +08:00] [INFO] [collector.go:69] ["Full Restore success summary"] [total
    '→ -ranges=12] [ranges-succeed=xxx] [ranges-failed=0] [split-region=xxx.xxxµs] [restore-
    '→ ranges=xxx] [total-take=xxx.xxxs] [restore-data-size(after-compressed)=xxx.xxx] [Size=
    '→ xxxx] [BackupTS={TS}] [total-kv=xxx] [total-kv-size=xxx] [average-speed=xxx]
Restore Meta Files
    '→ <
    '→  100.00%
Restore KV Files
    '→ <
    '→  100.00%
[2022/05/29 18:15:39.325 +08:00] [INFO] [collector.go:69] ["restore log success summary"] [total-
    '→ take=xxx.xx] [restore-from={TS}] [restore-to={TS}] [total-kv-count=xxx] [total-size=xxx]
```

5.3.3.4.7 清理过期备份数据

通过自动化运维工具（如 crontab）每两天定期清理过期备份数据的任务。

下面是执行过期备份数据清理任务：

- 删除早于 2022/05/14 00:00:00 的快照备份

```
shell rm s3://tidb-pitr-bucket/backup-data/snapshot-20220514000000
```

- 删除早于 2022/05/14 00:00:00 的日志备份数据

```
shell tiup br log truncate --until='2022-05-14 00:00:00 +0800' --storage='s3://tidb-pitr-bucket/
'→ backup-data/log-backup'
```

5.3.3.4.8 探索更多

- 备份存储
- 快照备份与恢复命令手册
- 日志备份与 PITR 命令手册

5.3.3.5 备份存储

TiDB 支持Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 和 NFS 作为备份恢复的存储。具体来说，可以在 `br` 的 `--storage` 或 `-s` 选项中指定备份存储的 URI。本文介绍不同外部存储服务中URI 的定义格式、存储过程中的鉴权方案以及存储服务端加密。

5.3.3.5.1 BR 向 TiKV 发送凭证

命令行参数	描述	默认值
`--send-credentials-to-tikv`	是否将 BR 获取到的权限凭证发送给TiKV。	`true`

在默认情况下，使用Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 存储时，BR 会将凭证发送到每个 TiKV 节点，以减少设置的复杂性。该操作由参数 `--send-credentials-to-tikv`（或简写为 `-c`）控制。

但是，这个操作不适合云端环境，如果采用了IAM Role 方式授权，那么每个节点都有自己的角色和权限。在这种情况下，你需要设置 `--send-credentials-to-tikv=false`（或简写为 `-c=0`）来禁止发送凭证：

```
./br backup full -c=0 -u pd-service:2379 --storage 's3://bucket-name/prefix'
```

使用 SQL 进行备份恢复时，可加上 `SEND_CREDENTIALS_TO_TIKV = FALSE` 选项：

```
BACKUP DATABASE * TO 's3://bucket-name/prefix' SEND_CREDENTIALS_TO_TIKV = FALSE;
```

5.3.3.5.2 URI 格式

格式说明

本部分介绍存储服务的 URI 格式：

```
[scheme]://[host]/[path]?[parameters]
```

- `scheme`：s3
- `host`：bucket name
- `parameters`：
  - `access-key`：访问密钥
  - `secret-access-key`：秘密访问密钥
  - `session-token`：会话令牌
  - `use-accelerate-endpoint`：是否在 Amazon S3 上使用加速端点，默认为 `false`
  - `endpoint`：Amazon S3 兼容服务自定义端点的 URL，例如 `<https://s3.example.com/>`
  - `force-path-style`：使用路径类型 (path-style)，而不是虚拟托管类型 (virtual-hosted-style)，默认为 `true`
  - `storage-class`：上传对象的存储类别，例如 `STANDARD`、 `STANDARD_IA`
  - `sse`：加密上传的服务端加密算法，可以设置为空、 `AES256` 或 `aws:kms`
  - `sse-kms-key-id`：如果 `sse` 设置为 `aws:kms`，则使用该参数指定 KMS ID
  - `acl`：上传对象的标准 ACL (Canned ACL)，例如 `private`、 `authenticated-read`

- scheme：gcs 或 gs
- host：bucket name
- parameters：
  - credentials-file：迁移工具节点上凭证 JSON 文件的路径
  - storage-class：上传对象的存储类别，例如 STANDARD 或 COLDLINE
  - predefined-acl：上传对象的预定义 ACL，例如 private 或 project-private
- scheme：azure 或 azblob
- host：container name
- parameters：
  - account-name：存储账户名
  - account-key：访问密钥
  - access-tier：上传对象的存储类别，例如 Hot、Cool、Archive，默认为 Hot

URI 示例

本部分示例以 host（上表中 bucket name、container name）为 external 为例进行介绍。

备份快照数据到Amazon S3

```
./br backup full -u "${PD_IP}:2379" \
--storage  "s3://external/backup-20220915?access-key=${access-key}&secret-access-key=${secret-
    ↪ access-key}"
```

从 Amazon S3 恢复快照备份数据

```
./br restore full -u "${PD_IP}:2379" \
--storage  "s3://external/backup-20220915?access-key=${access-key}&secret-access-key=${secret-
    ↪ access-key}"
```

备份快照数据到GCS

```
./br backup full --pd "${PD_IP}:2379" \
--storage  "gcs://external/backup-20220915?credentials-file=${credentials-file-path}"
```

从 GCS 恢复快照备份数据

```
./br restore full --pd "${PD_IP}:2379" \
--storage  "gcs://external/backup-20220915?credentials-file=${credentials-file-path}"
```

备份快照数据到Azure Blob Storage

```
./br backup full -u "${PD_IP}:2379" \
--storage  "azure://external/backup-20220915?account-name=${account-name}&account-key=${account-
    ↪ key}"
```

从 Azure Blob Storage 恢复快照备份数据中 test 数据库

```
./br restore db --db test -u "${PD_IP}:2379" \
--storage  "azure://external/backup-20220915account-name=${account-name}&account-key=${account-key
    ↪ }"
```

5.3.3.5.3 鉴权

将数据存储到云服务存储系统时，根据云服务供应商的不同，需要设置不同的鉴权参数。本部分介绍使用 Amazon S3、 GCS 及 Azure Blob Storage 时所用存储服务的鉴权方式以及如何配置访问相应存储服务的账户。

在备份之前，需要为br 命令行工具访问Amazon S3 中的备份目录设置相应的访问权限：

- 备份时TiKV 和 br 命令行工具需要的访问备份数据目录的最小权限：s3:ListBucket、 s3:PutObject 和 s3:AbortMultipartUpload。
- 恢复时TiKV 和 br 命令行工具需要的访问备份数据目录的最小权限：s3:ListBucket、 s3:GetObject 和 s3:PutObject。 br 命令行工具会将断点信息写到备份数据目录下的 ./checkpoints 子目录。在恢复日志备份数据时，br 命令行工具会将备份恢复集群的表ID 映射关系写到备份数据目录下的 ./pitr_id_maps 子目录。

如果你还没有创建备份数据保存目录，可以参考创建存储桶在指定的区域中创建一个 S3 存储桶。如果需要使用文件夹，可以参考使用文件夹在Amazon S3 控制台中组织对象在存储桶中创建一个文件夹。

配置访问Amazon S3 的账户可以通过以下两种方式：

- 方式一：指定访问密钥

  如果指定访问密钥和秘密访问密钥，将按照指定的访问密钥和秘密访问密钥进行鉴权。除了在 URI 中指定密钥外，还支持以下方式：

  - br 命令行工具读取 $AWS_ACCESS_KEY_ID 和 $AWS_SECRET_ACCESS_KEY 环境变量
  - br 命令行工具读取 $AWS_ACCESS_KEY 和 $AWS_SECRET_KEY 环境变量
  - br 命令行工具读取共享凭证文件，路径由 $AWS_SHARED_CREDENTIALS_FILE 环境变量指定
  - br 命令行工具读取共享凭证文件，路径为 ~/.aws/credentials

- 方式二：基于IAM Role 进行访问

  为运行TiKV 和 br 命令行工具的EC2 实例关联一个配置了访问 S3 访问权限的IAM role。正确设置后，br 命令行工具可以直接访问对应的 S3 中的备份目录，而不需要额外的设置。

  ```
  br backup full --pd "${PD_IP}:2379" \
  --storage "s3://${host}/${path}"
  ```

配置访问 GCS 的账户可以通过指定访问密钥的方式。如果指定了 credentials-file 参数，将按照指定的 credentials-file 进行鉴权。除了在 URI 中指定密钥文件外，还支持以下方式：

- br 命令行工具读取位于 $GOOGLE_APPLICATION_CREDENTIALS 环境变量所指定路径的文件内容
- br 命令行工具读取位于 ~/.config/gcloud/application_default_credentials.json 的文件内容

- 在 GCE 或 GAE 中运行时，从元数据服务器中获取的凭证

- 方式一：指定访问密钥

  在 URI 配置 account-name 和 account-key，则使用该参数指定的密钥。除了在 URI 中指定密钥文件外，还支持 br 命令行工具读取 $AZURE_STORAGE_KEY 的方式。

- 方式二：使用Azure AD 备份恢复

  在br 命令行工具运行环境配置环境变量$AZURE_CLIENT_ID、$AZURE_TENANT_ID 和$AZURE_CLIENT_SECRET。

  - 当集群使用TiUP 启动时，TiKV 会使用 systemd 服务。以下示例介绍如何为TiKV 配置上述三个环境变量：

    > 注意：
    > 该流程在第 3 步中需要重启TiKV。如果你的集群不适合重启，请使用指定访问密钥的方式进行备份恢复。

    1. 假设该节点上TiKV 端口为 24000，即 systemd 服务名为 tikv-24000：

    ```
    systemctl edit tikv-24000
    ```

    2. 编辑三个环境变量的信息：

    ```
    [Service]
    Environment="AZURE_CLIENT_ID=aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    Environment="AZURE_TENANT_ID=aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    Environment="AZURE_CLIENT_SECRET=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    ```

    3. 重新加载配置并重启TiKV：

    ```
    systemctl daemon-reload
    systemctl restart tikv-24000
    ```

  - 为命令行启动的 TiKV 和 br 命令行工具配置 Azure AD 的信息， 只需要确定运行环境中存在 $AZURE_CLIENT_ID、 $AZURE_TENANT_ID 和 $AZURE_CLIENT_SECRET。通过运行下列命令行，可以确认 br 命令行工具和TiKV 运行环境中是否存在这三个环境变量：

  ```
  echo $AZURE_CLIENT_ID
  echo $AZURE_TENANT_ID
  echo $AZURE_CLIENT_SECRET
  ```

  - 使用 br 命令行工具将数据备份至Azure Blob Storage：

  ```
  ./br backup full -u "${PD_IP}:2379" \
  --storage  "azure://external/backup-20220915?account-name=${account-name}"
  ```

5.3.3.5.4 存储服务端加密

Amazon S3 存储服务端加密备份数据

TiDB 备份恢复功能支持对备份到Amazon S3 的数据进行 S3 服务端加密(SSE)。S3 服务端加密也支持使用用户自行创建的AWS KMS 密钥，详细信息请参考BR S3 服务端加密。

5.3.3.5.5 存储服务其他功能支持

TiDB 备份恢复功能从 v6.3.0 支持AWS S3 Object Lock 功能。你可以在AWS 中开启 S3 Object Lock 功能来防止备份数据写入后被修改或者删除。

5.3.4 br cli 命令手册

5.3.4.1 br 命令行手册

本文介绍 br 命令的定义、组成、常用选项，以及快照备份与恢复、日志备份与 PITR (Point-in-time recovery) 功能使用的命令。

5.3.4.1.1 br 命令行描述

br 命令是由子命令、选项和参数组成的。子命令即不带 - 或者 -- 的字符。选项即以 - 或者 -- 开头的字符。参数即子命令或选项字符后紧跟的、并传递给命令和选项的字符。

以下是一条完整的 br 命令行：

```
br backup full --pd "${PD_IP}:2379" \
--storage "s3://backup-data/snapshot-202209081330/"
```

上面命令行中各部分的解释如下：

- backup：br 的子命令。
- full：br backup 的子命令。
- -s 或 --storage：备份数据的存储地址选项。"s3://backup-data/snapshot-202209081330/" 是 -s 的参数值。
- --pd：PD 访问地址选项。"${PD_IP}:2379" 是 --pd 的参数值。

命令和子命令

br 由多层命令组成。目前，br 包含的主要命令有：

- br backup：用于备份 TiDB 集群的全量数据。
- br log：用于启动和管理日志备份任务。
- br restore：用于恢复备份数据到 TiDB 集群。

br backup 和 br restore 还包含这些子命令：

- full：用于备份或恢复整个备份数据。
- db：用于备份或恢复集群中的指定数据库。
- table：用于备份或恢复集群指定数据库中的单张表。

常用选项

- --pd：PD 访问地址选项，例如 "${PD_IP}:2379"。

- `-s` 或 `--storage`：备份数据的存储地址选项。TiDB 备份恢复支持以Amazon S3、 Google Cloud Storage (GCS)、Azure Blob Storage 及 NFS 为备份存储。详细参考备份存储 URI 配置。
- `--ca`：指定 PEM 格式的受信任 CA 的证书文件路径。
- `--cert`：指定 PEM 格式的 SSL 证书文件路径。
- `--key`：指定 PEM 格式的 SSL 证书密钥文件路径。
- `--status-addr`：向 Prometheus 提供统计数据的监听地址。
- `--concurrency`：备份或恢复阶段的任务并发数。

5.3.4.1.2 全量备份命令行

使用 `br backup` 命令来备份集群全量数据。可选择添加 `full` 或 `table` 子命令来指定备份的范围：全部集群数据 (`full`) 或单张表的数据(`table`)。

- 备份集群快照数据
- 备份单个数据库的数据
- 备份单张表的数据
- 使用表库过滤功能备份多张表的数据
- 加密快照备份数据

5.3.4.1.3 日志备份命令行

使用 `br log` 命令来开启和管理日志备份任务。

- 启动日志备份
- 查询日志备份状态
- 暂停和恢复日志备份任务
- 停止和重启日志备份任务
- 清理日志备份数据
- 查看备份数据元信息

5.3.4.1.4 恢复备份数据命令行

使用 `br restore` 命令来恢复备份数据。可选择添加 `full`、`db` 或 `table` 子命令来指定恢复操作的范围：全部集群数据(`full`)、某个数据库 (`db`) 或某张数据表(`table`)。

- Point-in-time recovery
- 恢复快照备份数据
- 恢复单个数据库的快照备份数据
- 恢复单张表的快照备份数据
- 使用表库功能过滤恢复快照数据
- 恢复加密的快照备份数据

5.3.4.2 TiDB 快照备份与恢复命令行手册

本文按备份恢复的场景介绍快照备份和恢复的命令行，包括：

- 备份集群快照
- 备份单个数据库的数据
- 备份单张表的数据
- 使用表库过滤功能备份多张表的数据
- 备份数据加密
- 恢复快照备份数据
- 恢复单个数据库的数据
- 恢复单张表的数据
- 使用表库功能过滤恢复数据
- 恢复加密的快照备份数据

如果你想了解如何进行快照备份与恢复，可以参考以下教程：

- TiDB 快照备份与恢复使用指南
- TiDB 集群备份与恢复实践示例

5.3.4.2.1 备份集群快照

执行 `br backup full` 命令，可以备份TiDB 最新的或者指定时间点的快照数据。执行 `br backup full --help` 可获取该命令的使用帮助。

```
br backup full \
    --pd "${PD_IP}:2379" \
    --backupts '2022-09-08 13:30:00' \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backupfull.log
```

以上命令中：

- `--backupts`：快照对应的物理时间点。如果该快照的数据已经被 GC，那么 `br backup` 命令会报错退出；如果没有指定该参数，br 命令行工具会选取备份开始的时间点所对应的快照。
- `--ratelimit`：每个 TiKV 执行备份任务的速度上限（单位 MiB/s）。
- `--log-file`：备份日志写入的目标文件。

> 注意：
>
> BR 工具已支持自适应 GC，会自动将 backupTS（默认是最新的 PD timestamp）注册到 PD 的 `safePoint`，保证 TiDB 的GC Safe Point 在备份期间不会向前移动，即可避免手动设置GC。

备份期间终端会显示进度条，效果如下。当进度条达到 100% 时，表示备份完成。

```
Full Backup <---------/.....................................................> 17.12%.
```

5.3.4.2.2 备份TiDB 集群指定库表的数据

br 工具支持只备份集群快照和增量数据中指定库或表的局部数据。在快照备份和增量数据备份的基础上，该功能可过滤掉不需要的数据，只备份关键业务的数据。

备份单个数据库的数据

执行 `br backup db` 命令，可备份集群中指定单个数据库的数据。

下面是将数据库 `test` 备份到Amazon S3 的示例：

```
br backup db \
    --pd "${PD_IP}:2379" \
    --db test \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backuptable.log
```

db 子命令的选项为 `--db`，用来指定数据库名。其他选项的含义与备份 TiDB 集群快照相同。

备份单张表的数据

执行 `br backup table` 命令，可备份集群中指定单张表的数据。

下面是将表 `test.usertable` 备份到Amazon S3 的示例：

```
br backup table \
    --pd "${PD_IP}:2379" \
    --db test \
    --table usertable \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backuptable.log
```

`table` 子命令有 `--db` 和 `--table` 两个选项，分别用来指定数据库名和表名。其他选项的含义与备份 TiDB 集群快照相同。

使用表库过滤功能备份多张表的数据

如果你需要以更复杂的过滤条件来备份多个库或表，执行 `br backup full` 命令，并使用 `--filter` 或 `-f` 来指定表库过滤规则。

下面是将所有符合 `db*.tbl*` 条件的表的数据备份到Amazon S3 的示例：

```
br backup full \
    --pd "${PD_IP}:2379" \
    --filter 'db*.tbl*' \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backupfull.log
```

5.3.4.2.3 备份数据加密

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

br 命令行工具支持在备份端，或备份到Amazon S3 的时候在存储服务端进行备份数据加密，你可以根据自己情况选择其中一种使用。

自TiDB v5.3.0 起，你可配置下列参数在备份过程中实现数据加密：

- `--crypter.method`：加密算法，支持aes128-ctr、aes192-ctr 和aes256-ctr 三种算法，缺省值为plaintext，表示不加密
- `--crypter.key`：加密密钥，十六进制字符串格式，aes128-ctr 对应 128 位（16 字节）密钥长度，aes192-ctr 为 24 字节，aes256-ctr 为 32 字节
- `--crypter.key-file`：密钥文件，可直接将存放密钥的文件路径作为参数传入，此时 crypter.key 不需要配置

备份加密的示例如下：

```
br backup full\
    --pd ${PD_IP}:2379 \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --crypter.method aes128-ctr \
    --crypter.key 0123456789abcdef0123456789abcdef
```

> **注意：**
>
> - 密钥丢失，备份的数据将无法恢复到集群中。
> - 加密功能需在br 工具和TiDB 集群都不低于 v5.3.0 的版本上使用，且加密备份得到的数据无法在低于 v5.3.0 版本的集群上恢复。

5.3.4.2.4 恢复快照备份数据

执行 `br restore full` 命令，可将集群恢复到快照备份对应的数据状态。

```
br restore full \
    --pd "${PD_IP}:2379" \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file restorefull.log
```

以上命令中，

- `--ratelimit`：每个 TiKV 执行恢复任务的速度上限（单位 MiB/s）
- `--log-file`：备份日志写入的目标文件

恢复期间终端会显示进度条，效果如下。当进度条达到 100% 时，表示恢复完成。在完成恢复后，br 工具为了确保数据安全性，还会校验恢复数据。

```
Full Restore <---------/.....................................------------> 17.12%.
```

5.3.4.2.5 恢复备份数据中指定库表的数据

br 命令行工具支持只恢复备份数据中指定库/表的局部数据。该功能在恢复过程中过滤掉不需要的数据，可以用于往TiDB 集群上恢复指定库/表的数据。

恢复单个数据库的数据

执行 `br restore db` 命令，可将单个数据库恢复到对应的状态。

示例：恢复 S3 中的库 `test` 的数据：

```
br restore db \
    --pd "${PD_IP}:2379" \
    --db "test" \
    --ratelimit 128 \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --log-file restore_db.log
```

以上命令中 `--db` 选项指定了需要恢复的数据库名字。其余选项的含义与恢复快照备份数据相同。

> **注意：**
>
> 由于备份数据的元文件 `backupmeta` 记录了数据库名 `--db`，因此只能将数据恢复到同名的数据库，否则无法恢复成功。推荐把备份文件恢复到另一个集群的同名数据库中。

恢复单张表的数据

执行 `br restore table` 命令，可将单张表的数据恢复到对应的状态。

下面是恢复Amazon S3 中 `test.usertable` 表数据的示例：

```
br restore table \
    --pd "${PD_IP}:2379" \
    --db "test" \
    --table "usertable" \
    --ratelimit 128 \
```

```
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --log-file restore_table.log
```

以上命令中 `--table` 选项指定了需要恢复的表名。其余选项的含义与恢复单个数据库相同。

使用表库功能过滤恢复数据

如果你需要用复杂的过滤条件来恢复多个表，执行 `br restore full` 命令，并用 `--filter` 或 `-f` 指定使用表库过滤。

下面是恢复Amazon S3 中符合 `db*.tbl*` 条件的表的数据的示例：

```
br restore full \
    --pd "${PD_IP}:2379" \
    --filter 'db*.tbl*' \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --log-file restorefull.log
```

5.3.4.2.6 恢复加密的快照备份数据

> 警告：
>
> 当前该功能为实验特性，不建议在生产环境中使用。

在对数据做加密备份后，恢复操作需传入相应的解密参数，解密算法或密钥不正确则无法恢复，解密参数和加密参数一致即可。解密恢复的示例如下：

```
br restore full\
    --pd "${PD_IP}:2379" \
    --storage  "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --crypter.method aes128-ctr \
    --crypter.key 0123456789abcdef0123456789abcdef
```

5.3.4.3 TiDB 日志备份与 PITR 命令行手册

本文介绍TiDB 日志备份和 PITR (Point-in-time recovery) 命令行。

如果你想了解如何进行日志备份与 PITR，可以参考以下教程：

- TiDB 日志备份与 PITR 使用指南
- TiDB 集群备份与恢复实践示例

5.3.4.3.1 日志备份命令行介绍

你可以执行 `br log` 命令来开启和管理日志备份任务：

```
./br log --help

backup stream log from TiDB/TiKV cluster

Usage:
  br log [command]

Available Commands:
  metadata    get the metadata of log dir
  pause       pause a log backup task
  resume      resume a log backup task
  start       start a log backup task
  status      get status for the log backup task
  stop        stop a log backup task
  truncate    truncate the log data until sometime
```

各个子命令的作用如下：

- `br log start`：启动一个日志备份任务
- `br log status`：查询日志备份任务状态
- `br log pause`：暂停日志备份任务
- `br log resume`：重启暂停的备份任务
- `br log stop`：停止备份任务，并删除任务元信息
- `br log truncate`：从备份存储中清理日志备份数据
- `br log metadata`：查询备份存储中备份数据的元信息

启动日志备份

执行 `br log start` 命令，你可以在备份集群启动一个日志备份任务。该任务在TiDB 集群持续地运行，及时地将 KV 变更日志保存到备份存储中。

执行 `br log start --help` 命令可获取该子命令使用介绍：

```
./br log start --help
start a log backup task

Usage:
  br log start [flags]

Flags:
  -h, --help                help for start
  --start-ts string         usually equals last full backupTS, used for backup log. Default value
      '⟶ is current ts. support TSO or datetime, e.g. '400036290571534337' or '2018-05-11
      '⟶ 01:42:23+0800'.
```

```
  --task-name string       The task name for the backup log task.

Global Flags:
 --ca string                    CA certificate path for TLS connection
 --cert string                  Certificate path for TLS connection
 --key string                   Private key path for TLS connection
 -u, --pd strings               PD address (default [127.0.0.1:2379])
 -s, --storage string           specify the url where backup storage, eg, "s3://bucket/path/prefix"
```

以上命令行示例只展示了常用的参数，这些参数作用如下：

- `--start-ts`：指定开始备份日志的起始时间点。如果未指定，备份程序选取当前时间作为 `start-ts`。
- `task-name`：指定日志备份任务名。该名称也用于查询备份状态、暂停、重启和恢复备份任务等操作。
- `ca`、`cert`、`key`：指定使用 mTLS 加密方式与TiKV 和 PD 进行通讯。
- `--pd`：指定备份集群的 PD 访问地址。br 命令行工具需要访问 PD，发起日志备份任务。
- `--storage`：指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储，以上命令以 S3 为示例。详细参考备份存储 URI 格式。

使用示例：

```
./br log start --task-name=pitr --pd="${PD_IP}:2379" \
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"'
```

查询日志备份任务

执行 `br log status` 命令，你可以查询日志备份任务状态。

执行 `br log status --help` 命令可获取该子命令使用介绍：

```
./br log status --help
get status for the log backup task

Usage:
  br log status [flags]

Flags:
  -h, --help            help for status
  --json                Print JSON as the output.
  --task-name string    The task name for backup stream log. If default, get status of all of
      ↪ tasks (default "*")


Global Flags:
 --ca string                    CA certificate path for TLS connection
 --cert string                  Certificate path for TLS connection
 --key string                   Private key path for TLS connection
 -u, --pd strings               PD address (default [127.0.0.1:2379])
```

以上示例中，--task-name 为常用参数，它用来指定日志备份任务名。默认值为*，即显示全部任务。

使用示例：

```
./br log status --task-name=pitr --pd="${PD_IP}:2379"
```

命令输出如下：

```
● Total 1 Tasks.
> #1 <
               name: pitr
             status: ● NORMAL
              start: 2022-07-14 20:08:03.268 +0800
                end: 2090-11-18 22:07:45.624 +0800
            storage: s3://backup-101/logbackup
       speed(est.): 0.82 ops/s
checkpoint[global]: 2022-07-25 22:52:15.518 +0800; gap=2m52s
```

命令输出中的字段含义如下：

- status：任务状态，包括 NORMAL（正常）、ERROR（异常）和 PAUSE（暂停）三种状态。
- start：日志备份任务开始的时间，该值为备份任务启动时候指定的 start-ts。
- storage：备份存储。
- speed：日志备份任务的总 QPS（每秒备份的日志个数）。
- checkpoint [global]：集群中早于该 checkpoint 的数据都已经保存到备份存储，它也是备份数据可恢复的最近时间点。
- error [store]：存储节点上的日志备份组件运行遇到的异常。

暂停和恢复日志备份任务

执行 br log pause 命令，你可以暂停正在运行的日志备份任务。

执行 br log pause -help 可获取该子命令使用介绍：

```
./br log pause --help
pause a log backup task

Usage:
  br log pause [flags]

Flags:
  --gc-ttl int         the TTL (in seconds) that PD holds for BR's GC safepoint (default 86400)
  -h, --help           help for status
  --task-name string   The task name for backup stream log.


Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
```

```
 --key string                  Private key path for TLS connection
 -u, --pd strings              PD address (default [127.0.0.1:2379])
```

> **注意：**
>
> - 暂停日志备份任务后，备份程序为了防止生成变更日志的 MVCC 数据被删除，暂停任务程序会自动将当前备份点checkpoint 设置为 service safepoint，允许最多保留最近 24 小时内的 MVCC 数据。如果暂停的日志备份任务超过 24 小时未恢复，对应的数据就会被GC，不会备份。
> - 保留过多的 MVCC 数据会影响TiDB 集群的存储容量和性能，任务暂停后请及时恢复任务。

使用示例：

```
./br log pause --task-name=pitr --pd="${PD_IP}:2379"
```

执行 `br log resume` 命令，你可以恢复被暂停的日志备份任务。

执行 `br log resume --help` 命令可获取该子命令使用介绍：

```
./br log resume --help
resume a log backup task

Usage:
  br log resume [flags]

Flags:
  -h, --help            help for status
  --task-name string    The task name for backup stream log.


Global Flags:
 --ca string                   CA certificate path for TLS connection
 --cert string                 Certificate path for TLS connection
 --key string                  Private key path for TLS connection
 -u, --pd strings              PD address (default [127.0.0.1:2379])
```

暂停日志备份任务超过了 24 小时后，执行 `br log resume` 会报错，提示备份数据丢失。处理方法请参考恢复日志备份任务失败。

使用示例：

```
./br log resume --task-name=pitr --pd="${PD_IP}:2379"
```

停止和重启日志备份任务

通过执行 `br log stop` 命令，你可以停止正在进行的日志备份任务。停止的任务，可以通过 `--storage` 路径重新启动。

停止日志备份任务

执行 `br log stop` 命令，可以停止日志备份任务，该命令会清理备份集群中的任务元信息。

执行 `br log stop --help` 命令可获取该子命令使用介绍：

```
./br log stop --help
stop a log backup task

Usage:
  br log stop [flags]

Flags:
  -h, --help            help for status
  --task-name string    The task name for the backup log task.


Global Flags:
 --ca string                    CA certificate path for TLS connection
 --cert string                  Certificate path for TLS connection
 --key string                   Private key path for TLS connection
 -u, --pd strings               PD address (default [127.0.0.1:2379])
```

> **注意：**
>
> 请谨慎使用该命令，如果你只需暂时停止日志备份，请使用 `br log pause` 和 `br log resume` 命令。

使用示例：

```
./br log stop --task-name=pitr --pd="${PD_IP}:2379"
```

重新启动备份任务

当使用 `br log stop` 命令停止日志备份任务后，可在另一个 `--storage` 路径下重新创建一个新的日志备份任务，也可以在原来的 `--storage` 路径下执行 `br log start` 命令重新启动日志备份任务。如果是在原来的 `--storage` 路径重启任务，需要注意：

- 重启备份任务的 `--storage` 参数需要与停止任务之前的参数相同。
- 此时不需要填入 `--start-ts` 参数，程序将自动从上次的备份进度点开始备份数据。
- 如果停止任务后的时间过长， 多版本的数据已经被 GC， 则在重启备份任务时会报错 `BR:Backup:
  ↪ ErrBackupGCSafepointExceeded`，此时只能配置另外的日志路径来重新创建日志备份任务。

清理日志备份数据

执行 `br log truncate` 命令，你可以从备份存储中删除过期或不再需要的备份日志数据。

执行 `br log truncate --help` 命令可获取该子命令使用介绍：

```
./br log truncate --help
truncate the incremental log until sometime.

Usage:
  br log truncate [flags]

Flags:
  --dry-run         Run the command but don't really delete the files.
  -h, --help        help for truncate
  --until string    Remove all backup data until this TS.(support TSO or datetime, e.g.
      '→ '400036290571534337' or '2018-05-11 01:42:23+0800'.)
  -y, --yes         Skip all prompts and always execute the command.



Global Flags:
  -s, --storage string          specify the url where backup storage, eg, "s3://bucket/path/prefix
      '→ "
```

该命令只需要访问备份存储，不需要访问备份集群。此外，常用的参数如下：

- `--dry-run`：运行命令，但是不删除文件。
- `--until`：早于该参数指定时间点的日志备份数据会被删除。建议使用快照备份的时间点作为该参数值。
- `--storage`：指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。

使用示例：

```
./br log truncate --until='2022-07-26 21:20:00+0800' \
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    '→ key}"'
```

该子命令运行后输出以下信息：

```
Reading Metadata... DONE; take = 277.911599ms
We are going to remove 9 files, until 2022-07-26 21:20:00.0000.
Sure? (y/N) y
Clearing data files... DONE; take = 43.504161ms, kv-count = 53, kv-size = 4573(4.573kB)
Removing metadata... DONE; take = 24.038962ms
```

查看备份数据元信息

执行 `br log metadata` 命令，你可以查看备份存储中保存的日志备份的元信息，例如最早和最近的可恢复时间点。

执行 `br log metadata --help` 命令可获取该子命令使用介绍：

```
./br log metadata --help
get the metadata of log backup storage

Usage:
  br log metadata [flags]


Flags:
  -h, --help       help for metadata

Global Flags:
  -s, --storage string         specify the url where backup storage, eg, "s3://bucket/path/prefix
      '→ "
```

该命令只需要访问备份存储，不需要访问备份集群。

以上示例中，`--storage` 为常用参数，它用来指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。

使用示例：

```
./br log metadata --storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key
    '→ =${secret-access-key}"'
```

该子命令运行后输出以下信息：

```
 [2022/07/25 23:02:57.236 +08:00] [INFO] [collector.go:69] ["log metadata"] [log-min-ts
     '→ =434582449885806593] [log-min-date="2022-07-14 20:08:03.268 +0800"] [log-max-ts
    '→ =434834300106964993] [log-max-date="2022-07-25 23:00:15.618 +0800"]
```

5.3.4.3.2 恢复到指定时间点 PITR

执行 `br restore point` 命令，你可以在新集群上进行 PITR，或者只恢复日志备份数据。

执行 `br restore point --help` 命令可获取该命令使用介绍：

```
./br restore point --help
restore data from log until specify commit timestamp

Usage:
  br restore point [flags]

Flags:
  --full-backup-storage string specify the backup full storage. fill it if want restore full
      '→ backup before restore log.
  -h, --help                   help for point
  --restored-ts string         the point of restore, used for log restore. support TSO or
      '→ datetime, e.g. '400036290571534337' or '2018-05-11 01:42:23+0800'
```

```
  --start-ts string            the start timestamp which log restore from. support TSO or
      '→ datetime, e.g. '400036290571534337' or '2018-05-11 01:42:23+0800'


Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
 --key string                 Private key path for TLS connection
 -u, --pd strings             PD address (default [127.0.0.1:2379])
 -s, --storage string         specify the url where backup storage, eg, "s3://bucket/path/prefix"
```

以上示例只展示了常用的参数，这些参数作用如下：

- `--full-backup-storage`：指定快照（全量）备份的存储地址。如果你要使用 PITR，需要指定该参数，并选择恢复时间点之前最近的快照备份；如果只恢复日志备份数据，则不需要指定该参数。快照备份支持以 Amazon S3、Google Cloud Storage (GCS)、Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。
- `--restored-ts`：指定恢复到的时间点。如果没有指定该参数，则恢复到日志备份数据最后的可恢复时间点（备份数据的checkpoint）。
- `--start-ts`：指定日志备份恢复的起始时间点。如果你只恢复日志备份数据，不恢复快照备份，需要指定这个参数。
- `ca`、`cert`、`key`：指定使用 mTLS 加密方式与TiKV 和 PD 进行通讯。
- `--pd`：指定恢复集群的 PD 访问地址。
- `--storage`：指定备份存储地址。日志备份支持以 Amazon S3、Google Cloud Storage (GCS)、Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。

使用示例：

```
./br restore point --pd="${PD_IP}:2379"
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    '→ key}"'
--full-backup-storage='s3://backup-101/snapshot-202205120000?access-key=${access-key}&secret-
    '→ access-key=${secret-access-key}"'

Full Restore
    '→ <
    '→  100.00%
*** ***["Full Restore success summary"] ****** [total-take=3.112928252s] [restore-data-size(after
    '→ -compressed)=5.056kB] [Size=5056] [BackupTS=434693927394607136] [total-kv=4] [total-kv-
    '→ size=290B] [average-speed=93.16B/s]
Restore Meta Files
    '→ <
    '→  100.00%
Restore KV Files
    '→ <
    '→  100.00%
```

```
"restore log success summary"] [total-take=192.955533ms] [restore-from=434693681289625602] [
    ↪ restore-to=434693753549881345] [total-kv-count=33] [total-size=21551]
```

> 注意：
>
> - 不支持重复恢复某段时间区间的日志，如多次重复恢复 `[t1=10, t2=20)` 区间的日志数据，可能会造成恢复后的数据不正确。
> - 多次恢复不同时间区间的日志时，需保证恢复日志的连续性。如先后恢复 `[t1, t2)`、`[t2, t3)` 和 `[t3, t4)` 三个区间的日志可以保证正确性，而在恢复 `[t1, t2)` 后跳过 `[t2, t3)` 直接恢复 `[t3, t4)` 的区间可能导致恢复之后的数据不正确。

5.3.5 参考指南

5.3.5.1 BR 特性

5.3.5.1.1 备份自动调节从 v5.4 版本开始引入

在 TiDB v5.4.0 之前，默认情况下，使用 BR 进行备份任务时使用的线程数量占总逻辑 CPU 数量的 75%。在没有限速的前提下，备份会消耗大量的集群资源，这会对在线集群的性能造成相当大的影响。虽然你可以通过调节线程池的大小的方式来减少备份对集群性能的影响，但观察负载、手动调节线程池大小也是一件繁琐的事情。

为了减少备份任务对在线集群的影响，从TiDB v5.4.0 起，引入了自动调节功能，此功能默认开启。在集群资源占用率较高的情况下，备份功能可以通过该功能自动限制备份使用的资源，从而减少对集群的影响。

使用场景

如果你希望减少备份对集群的影响，那么，你可以开启自动调节功能。开启该功能后，备份功能会在不过度影响集群的前提下，以最快的速度进行数据备份。

或者，你也可以使用TiKV 配置项`backup.num-threads` 或参数 `--ratelimit` 进行备份限速。

使用方法

自动调节功能默认打开，无需额外配置。

> 注意：
>
> v5.3.x 版本的集群，在升级到 v5.4.0 及以上版本后，自动调节功能默认关闭，需手动开启。

如需开启备份自动调节功能，可以通过把TiKV 配置项`backup.enable-auto-tune` 设置为 `true` 的方式来完成。

TiKV 支持动态配置自动调节功能，因此，在开启或关闭该功能时，无需重启集群。你可以运行以下命令动态启动或停止备份自动调节功能：

```
tikv-ctl modify-tikv-config -n backup.enable-auto-tune -v <true|false>
```

在离线备份场景中，你也可以使用 tikv-ctl 把 backup.num-threads 修改为更大的数字，从而提升备份速度。

使用限制

自动调节是一个粗粒度的限流方案，它的优势在无需手动调节。但是，由于调节的粒度不够精确，该功能有可能无法彻底移除备份对集群的影响。

该功能的已知问题及其解决方案如下：

- 问题 1：对于以写负载为主的集群，自动调节可能会让工作负载和备份进入一种“正反馈循环”：备份会占用较多资源，导致工作负载使用的资源变少。此时，自动调节会误以为资源使用率下降，从而让备份运行得更加激进。在这种情况下，自动调节实际上失效。
  - 解决方法：手动调节 backup.num-threads，限制处理备份的工作线程数量。具体原理如下：
    目前，备份过程会涉及大量的 SST 解码、编码、压缩、解压，而此过程会需要消耗大量的 CPU 资源。另外，以往的测试证明，备份过程中，用于备份的线程池的 CPU 利用率接近 100%。也就是说，备份任务会占用大量 CPU 资源。通过调整备份任务使用的线程数量，TiKV 可以控制备份任务使用的 CPU 核心数，从而减少其任务对集群性能的影响。
- 问题 2：对于存在热点的集群，产生热点的TiKV 节点可能会被过度限流，从而拉慢备份的整体进度。
  - 解决方法：消除热点节点，或者在热点节点上关闭自动调节（关闭此功能可能会导致集群性能降低）。
- 问题 3：对于流量抖动非常大的场景，由于自动调节每隔auto-tune-refresh-interval（默认为一分钟）才会计算出新的限流，所以可能无法很好地应对流量抖动非常厉害的场景。
  - 解决方法：关闭自动调节。

实现原理

自动调节会通过调节备份时使用的工作线程池的大小，保证集群的 CPU 总体使用率不超过某个特定的值。

这个特性还有两个配置项未在TiKV 文档中列出，仅在内部调试使用，正常备份时无需配置这两个参数。

- backup.auto-tune-remain-threads：
  - 通过控制备份任务占用的资源，自动调节会保证该节点中至少有该数量的核心会保持空闲的状态。
  - 默认值：round(0.2 * vCPU)
- backup.auto-tune-refresh-interval：
  - 每隔该值的时间段，自动调节会刷新统计信息并重新计算备份任务使用的 CPU 核心数的上限。
  - 默认值：1m

以下是一个使用自动调节功能的示例，其中 * 代表集群中被备份任务占用的 CPU，^ 代表其它任务占用的 CPU，- 代表空闲 CPU。

```
|--------| 系统总共有 8 颗逻辑 CPU。
|****----| 默认配置 `backup.num-threads` 为 `4`。请注意，在任何时候自动调节都不会让线程池大小大于
    ′→ `backup.num-threads`。
|^^****--| 默认配置 `auto-tune-remain-threads` = round(8 * 0.2) = 2。
    ′→ 自动调节会将备份任务的线程池大小调节至 `4`。
|^^^^**--| 由于集群的工作负载加重，自动调节将备份任务的线程池大小调节至 `2`。调节后，集群中仍有 2
    ′→ 个 CPU 核心数保持空闲。
```

在监控面板的“Backup CPU Utilization”中，可以看到自动限流目前选择的线程池的大小：

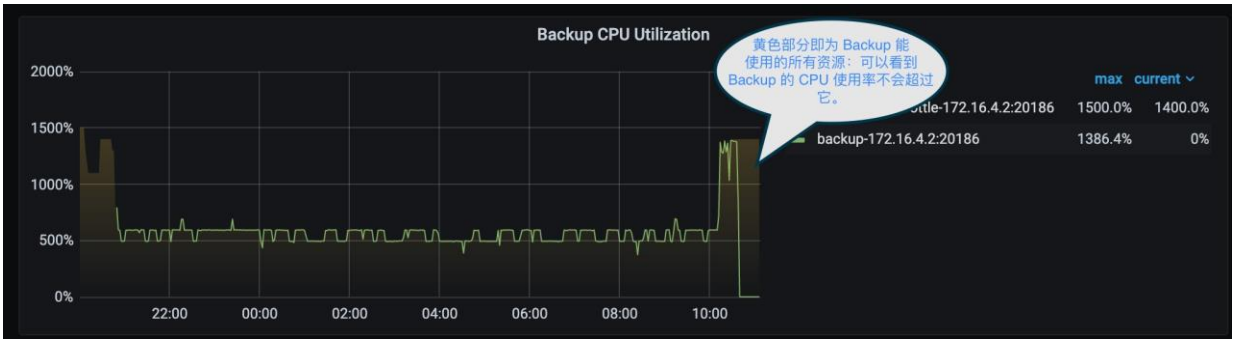


图 23: Grafana dashboard example of backup auto-tune metrics

图片中，黄色半透明的填充部分为开启自动调节后备份任务可用的线程，即备份任务能使用的所有资源。从中可以看到备份任务的 CPU 使用率不会超过黄色部分。

5.3.5.1.2 批量建表

使用 Backup & Restore (BR) 执行数据恢复任务时，BR 会先在目标TiDB 集群上创建库和表，然后再把已备份的数据恢复到表中。TiDB v6.0.0 之前，在数据恢复阶段创建表时，BR 采用了串行执行的方案。然而，当需要恢复的数据中带有大量的表（约 50000 张）时，该方案会在创建表上消耗较多时间。

为了加快创建表的速度，以减少数据恢复的时间，TiDB 在 v6.0.0 中引入了批量建表功能，此功能默认开启。

> 注意：
>
> - 为使用批量建表功能，TiDB 和 br 命令行工具都必须为 v6.0.0 或以上版本。如果TiDB 或 br 命令行工具的任意一方的版本低于 v6.0.0，br 命令行工具会采用原来的串行建表方案。
> - 如果你在使用TiUP 等集群管理工具，而且你使用的TiDB 和 br 命令行工具是 v6.0.0 或以上版本，或是从 v6.0.0 以下版本升级至该版本，此时，br 工具会默认开启批量建表功能，你不需要额外进行相关配置。

使用场景

当需要恢复的数据中带有大量的表（约 50000 张）时，你可以使用批量建表功能显著提升数据恢复的速度。

具体提速效果可参考测试结果。

使用方法

Backup & Restore (BR) 默认开启批量建表功能，在 v6.0.0 或以上版本中默认设置了 `--ddl-batch-size=128`（以 128 张表为一批，并发创建多批表），以加快恢复时的建表速度。因此，你不需要额外配置该参数。

如果需要关闭此功能，你可以参考以下命令将 `--ddl-batch-size` 的值设置为 `1`：

```
br restore full \
--storage local:///br_data/ --pd "${PD_IP}:2379" --log-file restore.log \
--ddl-batch-size=1
```

关闭批量建表功能后，BR 会采用原来的串行建表方案。

实现原理

- v6.0.0 前的串行建表方案：

  执行数据恢复任务时，BR 会先在目标TiDB 集群上创建库和表，然后再把已备份的数据恢复到表中。建表时，BR 会调用TiDB 内部API 后开始创建表，其运作方式类似 BR 执行 SQL `Create Table` 语句。建表任务由TiDB DDL owner 依次串行执行。DDL owner 每创建一张表会引起一次 DDL schema 版本的变更，而每次的 schema 版本的变更都需要同步到其他TiDB DDL worker（含 BR）。因此，当需要创建的表的数量比较多时，串行建表方案会导致建表时间过长。

- v6.0.0 起的批量建表方案：

  在默认情况下，BR 会以 128 张表为一批，并发创建多批表。采用该方案后，BR 每建一批表时，TiDB schema 版本只会变更一次。此方法极大地提高了建表速度。

功能测试

以下是在TiDB v6.0.0 集群中测试批量建表功能的内容。具体的测试环境如下：

- 集群配置：
  - 15 个TiKV 实例，每个 TiKV 实例共有 16 个 CPU 核心、80 GB 内存、16 个处理RPC 请求的线程（即`import.num-threads` = 16）
  - 3 个 TiDB 实例，每个TiDB 实例共有 16 个 CPU 核心、32 GB 内存。
  - 3 个 PD 实例，每个 PD 实例共有 16 个 CPU 核心、32 GB 内存。
- 待恢复数据的规模：16.16 TB

测试结果如下：

```
‘[2022/03/12 22:37:49.060 +08:00] [INFO] [collector.go:67] ["Full restore success summary"] [
   ’→ total-ranges=751760] [ranges-succeed=751760] [ranges-failed=0] [split-region=1h33m18
   ’→ .078448449s] [restore-ranges=542693] [total-take=1h41m35.471476438s] [restore-data-
   size( ’→ after-compressed)=8.337TB] [Size=8336694965072] [BackupTS=431773933856882690]
   [total-kv ’→ =148015861383] [total-kv-size=16.16TB] [average-speed=2.661GB/s]’
```

从结果可见，单个TiKV 实例的平均恢复速度高达 181.65 MB/s（即 `average-speed/tikv_count`）。

5.3.5.1.3 断点备份

快照备份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、节点宕机等等一些突发情况。在TiDB v6.5.0 之前，在错误被处理之后，之前备份的数据会作废，你需要重新进行备份。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备份，从TiDB v6.5.0 起，备份恢复特性引入了断点备份的功能。该功能可以在备份意外中断后保留上一次备份的大部分进度。

使用场景

如果你的TiDB 集群规模很大，无法接受备份失败后需要重新备份的情况，那么，你可以使用断点备份功能重试备份。br 工具会定期记录已备份的分片，使得在下一次重试备份时回到离上一次退出时较近的进度。

实现原理

在快照备份过程中，br 工具将表编码成对应的键空间，并生成备份的RPC 请求发送给所有TiKV 节点。接收到备份请求后，TiKV 节点会选择请求范围内的数据进行备份。每当TiKV 节点备份完一个Region 级别的数据，就会返回给br 工具这段范围相关的备份信息。

通过记录这些返回的备份信息，br 工具能够及时获知已备份完成的键范围。断点备份功能会定期将这些新增的相关备份信息上传至外部存储中，从而持久化存储该备份任务已完成的键范围。

在重试备份时，br 工具会从外部存储读取已经备份的键范围，再与所需备份的键范围做比较，获得一个差集，从而得出断点备份还需要备份的键范围。

使用限制

断点备份功能依赖于GC 机制，且无法恢复所有已备份的数据，具体描述如下。

确保在GC 前重试

在备份过程中，br 命令行工具会定期向 PD 更新备份 snapshot 的 `gc-safepoint`，从而避免在备份过程中数据被GC。当br 工具退出后，PD 中备份 snapshot 的 `gc-safepoint` 将无法及时更新。因此，在下一次重试备份前，数据有可能已经被GC。

为了避免数据被GC，当br 工具没有指定具体的 `gcttl` 时，br 工具在默认情况下会让 `gc-safepoint` 保留大约 1 小时。如果你需要延长这个时间，可以设置参数 `gcttl`。

例如，设置 `gcttl` 为 15 小时（54000 秒）来延长 `gc-safepoint` 的保留时间：

```
br backup full \
--storage local:///br_data/ --pd "${PD_IP}:2379" \
--gcttl 54000
```

> **注意：**
>
> 快照备份结束后会将备份前创建的 `gc-safepoint` 删除，不需要手动删除。

部分数据需要重新备份

在重试备份时，部分已备份的数据可能需要重新备份，包括正在备份的数据和还未被断点记录的数据。

- 由错误引起的退出，br 工具会在退出前将已备份的数据元信息持久化到外部存储中，因此只有正在备份的数据需要在下一次重试中重新备份。
- 如果br 工具进程被系统中断，那么br 将无法把已备份的数据元信息持久化到外部存储中。br 工具持久化已备份的数据元信息的周期为 30 秒，因此大约最近 30 秒内的已完成备份的数据无法持久化到外部存储，在下次重试时需要重新备份。

5.3.5.1.4 断点恢复

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、节点宕机等等一些突发情况。在TiDB v7.1.0 之前，在错误被处理之后，之前恢复的进度会作废，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。

使用场景

如果你的TiDB 集群规模很大，无法接受恢复失败后需要重新恢复的情况，那么，你可以使用断点恢复功能重试恢复。br 工具会定期记录已恢复的分片，使得在下一次重试恢复时回到离上一次退出时较近的进度。

实现原理

断点恢复的实现原理分为快照恢复和日志恢复两部分。

快照恢复

快照恢复的实现原理与快照备份类似，br 工具会批量恢复一段键范围(Region) 内的所有的 SST 文件。恢复完成后，br 工具记录下这段范围以及该范围对应的恢复集群的表ID。断点恢复功能会定期将这些新增的相关恢复信息上传至外部存储中，从而持久化存储该恢复任务已完成的键范围。

在重试恢复时，br 工具会从外部存储读取已经恢复的键范围，并匹配相关的表ID。在恢复过程中，会跳过与断点恢复记录的键范围重叠且对应表ID 相同的范围。

如果在重试恢复前，你删除了某些表，那么重试恢复时新创建的表的ID 会与之前记录在断点恢复的表ID 不同，从而绕过使用之前的断点恢复信息，重新恢复该表，也就是说，新的ID 的同个表可以不使用旧的ID 的断点恢复信息，而是重新记录新ID 对应的断点恢复信息。

由于数据采用了 MVCC（多版本并发控制）机制，带有固定 TS 的数据可以被无序且重复写入。

快照恢复在恢复库表 DDL 时添加了 `ifExists` 参数。对于已经存在的库表（视为已经创建完毕），br 工具会自动跳过恢复。

日志恢复

日志恢复需要按照时间顺序恢复TiKV 节点备份下来的数据元信息(meta-kv)。断点恢复首先会根据meta-kv 数据为备份集群和恢复集群建立一一对应的ID 映射关系，以确保在不同重试恢复过程中meta-kv 的 ID 保持一致。这样，meta-kv 就具备了重新恢复的能力。

与快照备份文件不同，日志备份文件的范围可能会重叠，因此无法直接使用键范围作为恢复进度的元信息。同时，日志备份文件数量可能非常多。不过，每个日志备份文件在日志备份元信息中的位置是固定的，因此可以为每个日志备份文件指定一个日志备份元信息中唯一的位置作为恢复进度的元信息。

日志备份元信息中包含一个存放文件元信息的数组。数组中的每个文件元信息代表一个由多个日志备份文件拼接而成的文件。文件元信息记录了这些日志备份文件在拼接文件中的偏移和大小。因此，br 工具可以使用三元组(日志备份元信息名，文件元信息数组偏移，日志备份文件数组偏移）来唯一标识一个日志备份文件。

使用限制

断点恢复功能依赖于GC 机制，且无法记录所有已恢复的数据，具体描述如下。

GC 会被停止

在日志恢复过程中，日志数据恢复的顺序是无序的，因此可能会出现一个 key 的删除记录优先于该 key 的写入记录恢复的情况。如果此时触发GC，该 key 的所有数据会被删除，导致该 key 后续恢复的写入记录无法被 GC 处理。为了避免这种情况，br 命令行工具会在日志恢复过程中暂停GC。当 br 工具中途退出后，GC 将仍然保持暂停状态。

日志恢复完成后会重新启动GC，不需要手动启动。但如果你不想再继续恢复时，请手动开启GC，开启方法如下：

br 工具暂停GC 的原理是通过执行 `SET config tikv gc.ratio-threshold = -1.0`，将gc.ratio-threshold 的值设置成负数来暂停GC。你可以通过修改 `gc.ratio-threshold` 的值手动开启GC，例如执行 `SET config tikv ′→ gc.ratio-threshold = 1.1` 将参数重置为默认值。

部分数据需要重新恢复

在重试恢复时，部分已恢复的数据可能需要重新恢复，包括正在恢复的数据和还未被断点记录的数据。

- 由错误引起的退出，br 工具会在退出前将已恢复的数据元信息持久化到外部存储中，因此只有正在恢复的数据需要在下一次重试中重新恢复。
- 如果br 工具进程被系统中断，那么br 将无法把已恢复的数据元信息持久化到外部存储中。br 工具持久化已恢复的数据元信息的周期为 30 秒，因此大约最近 30 秒内的已完成恢复的数据无法持久化到外部存储，在下次重试时需要重新恢复。

不要在恢复期间修改集群数据

在恢复失败后，请避免向集群写入或删除数据、删除或创建表。由于备份数据中可能包含重命名表的 DDL 操作，断点恢复无法确认被删除的表或已存在的表是否是外部操作引起的，这会影响下次重试恢复的准确性。

5.3.5.2 使用 Dumpling 和 TiDB Lightning 备份与恢复

本文档介绍如何使用 Dumpling 和TiDB Lightning 进行全量备份与恢复。

在备份与恢复场景中，如果需要全量备份少量数据（例如小于 50 GB），且不要求备份速度，你可以使用Dumpling 从TiDB 数据库导出数据进行备份，再使用TiDB Lightning 将数据导入至TiDB 数据库实现恢复。更多备份与恢复的相关信息，参见TiDB 备份与恢复概述。

5.3.5.2.1 前提条件

- 安装和运行 Dumpling：

```
tiup install dumpling && tiup dumpling
```

- 安装和运行TiDB Lightning：

```
tiup install tidb lightning && tiup tidb lightning
```

- 获取 Dumpling 所需上游数据库权限
- 获取TiDB Lightning 所需下游数据库权限

5.3.5.2.2 资源要求

操作系统：本文档示例使用的是若干新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将TiDB Lightning 和其他组件（比如 `tikv-server`）部署在同一台机器上，然后设置 `region-concurrency` 配置项的值为逻辑 CPU 数的 75%，以限制TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：因为 TiDB Lightning 对计算机资源消耗较高，建议分配 64 GB 以上的内存以及 32 核以上的 CPU，而且确保 CPU 核数和内存(GB) 比为 1:2 以上，以获取最佳性能。

磁盘空间：

推荐使用Amazon S3、 Google Cloud Storage (GCS) 和Azure Blob Storage 等外部存储，以便能够快速存储备份文件，且不受磁盘空间限制。

如果需要保存单次备份数据到本地磁盘，需要注意以下磁盘空间限制：

- Dumpling 需要能够储存整个数据源的存储空间，即可以容纳要导出的所有上游表的空间。计算方式参考目标数据库所需空间。
- TiDB Lightning 导入期间，需要临时空间来存储排序键值对，磁盘空间需要至少能存储数据源的最大单表。

说明：目前无法精确计算Dumpling 从TiDB 导出的数据大小，但你可以用下面SQL 语句统计信息表的`data_length`字段估算数据量：

```
/* 统计所有 schema 大小，单位 MiB，注意修改 ${schema_name} */
SELECT table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)/1024/1024 AS
    ’→ index_length,SUM(data_length+index_length)/1024/1024 AS SUM FROM information_schema.
    ’→ tables WHERE table_schema = "${schema_name}" GROUP BY table_schema;

/* 统计最大单表，单位 MiB，注意修改 ${schema_name} */
 SELECT table_name,table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)
     ’→ /1024/1024 AS index_length,SUM(data_length+index_length)/1024/1024 AS SUM from
     ’→ information_schema.tables WHERE table_schema = "${schema_name}" GROUP BY table_name,
    ’→ table_schema ORDER BY SUM DESC LIMIT 5;
```

目标TiKV 集群的磁盘空间要求

目标TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标TiKV 集群的总存储空间必须大于数据源大小× 副本数量× 2。例如，集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间。
- RocksDB 的空间放大效应。

5.3.5.2.3 使用 Dumpling 备份全量数据

1. 运行以下命令，从TiDB 导出全量数据至Amazon S3 存储路径 `s3://my-bucket/sql-backup`：

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MiB -B my_db1 -f 'my_db1.table
    ’→ [12]' -o 's3://my-bucket/sql-backup'
```

Dumpling 默认导出数据格式为 SQL 文件，你也可以通过设置 `--filetype` 指定导出文件的类型。

关于更多 Dumpling 的配置，请参考Dumpling 主要选项表。

2. 导出完成后，可以在数据存储目录 `s3://my-bucket/sql-backup` 查看导出的备份文件。

5.3.5.2.4 使用TiDB Lightning恢复全量数据

1. 编写配置文件 `tidb-lightning.toml`，将 Dumpling 备份的全量数据从 `s3://my-bucket/sql-backup` 恢复到目标TiDB 集群：

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
# "tidb"：TB 级以下数据量也可以采用`tidb`后端模式，下游 TiDB 可正常提供服务。
    ′→ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-
    ′→ backends
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小
    ′→ 。建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO
    ′→ 会获得更好的导入性能
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
# 源数据目录，即上一章节中 Dumpling 保存数据的路径。
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'


[tidb]
# 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"

status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”
    ′→ 获取表结构信息，例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ′→ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ′→ 否则导入将出现异常。
```

关于更多TiDB Lightning 的配置，请参考TiDB Lightning 配置参数。

2. 运行 `tidb-lightning`。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

若从Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和AccessKey 作为环境变量传入 Lightning 节点。同时还支持从 `~/.aws/credentials` 读取凭证文件。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

3. 导入开始后，可以通过 grep 日志关键字 `progress` 查看进度，默认 5 分钟更新一次。
4. 导入完毕后，TiDB Lightning 会自动退出。查看 `tidb-lightning.log` 日志末尾是否有 `the whole procedure completed`，如果有，数据导入成功，恢复完成。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表恢复任务完成。

如果恢复过程中遇到问题，请参见TiDB Lightning 常见问题。

5.3.5.3 备份与恢复RawKV 数据

TiKV 可以独立于TiDB，与 PD 构成 KV 数据库，此时的产品形态为RawKV。TiKV-BR 工具支持对使用RawKV 的产品进行备份和恢复。

更多信息，请参考TiKV-BR 用户文档。

5.3.5.4 TiDB 增量备份与恢复使用指南

TiDB 集群增量数据包含在某个时间段的起始和结束两个快照的变化差异的数据，以及之间的 DDL。增量备份的数据相对比全量备份数据而言数据量更小，适合配合快照备份一起使用，减少备份的数据量。进行增量备份的时候，需要保证备份时间范围内的多版本数据没有被TiDB GC 机制回收。例如，每个小时进行一次增量备份，则需要调整TiDB 集群的GC Lifetime 设置至少大于 1 小时。

> **警告：**
>
> 当前该功能已经停止开发迭代，推荐你选择日志备份与恢复功能代替。

5.3.5.4.1 对集群进行增量备份

使用 `br backup` 进行增量备份只需要指定上一次的备份时间戳 `--lastbackupts`，br 命令行工具会判定需要备份 `lastbackupts` 和当前时间之间增量数据。使用 `validate` 指令获取上一次备份的时间戳，示例如下：

```
LAST_BACKUP_TS=`tiup br validate decode --field="end-version" --storage "s3://backup-101/snapshot
    ↪ -202209081330?access-key=${access-key}&secret-access-key=${secret-access-key}"| tail -n1`
```

备份 (LAST_BACKUP_TS, current timestamp] 之间的增量数据，以及这段时间内的 DDL：

```
tiup br backup full --pd "${PD_IP}:2379" \
--storage  "s3://backup-101/snapshot-202209081330/incr?access-key=${access-key}&secret-access-key=
    ↪ ${secret-access-key}" \
--lastbackupts ${LAST_BACKUP_TS} \
--ratelimit 128
```

以上命令会中：

- --lastbackupts：上一次的备份时间戳。
- --ratelimit：每个 TiKV 执行备份任务的速度上限（单位 MiB/s）。
- storage：数据备份到存储地址。增量备份数据需要与快照备份数据保存在不同的路径下，例如上例保存在全量备份数据下的 incr 目录中。详细参考备份存储 URI 配置。

5.3.5.4.2 恢复增量备份数据

恢复增量数据的时候，需要保证备份时指定的 LAST_BACKUP_TS 之前备份的数据已经全部恢复到目标集群。同时，因为增量恢复的时候会更新数据，因此你需要保证此时不会有其他写入，避免出现冲突。

恢复全量备份数据，备份数据存储在 backup-101/snapshot-202209081330 目录下：

```
tiup br restore full --pd "${PD_IP}:2379" \
--storage  "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}"
```

恢复全量备份后的增量备份数据，备份数据存储在 backup-101/snapshot-202209081330/incr 目录下：

```
tiup br restore full --pd "${PD_IP}:2379" \
--storage  "s3://backup-101/snapshot-202209081330/incr?access-key=${access-key}&secret-access-key=
    ↪ ${secret-access-key}"
```

5.4 集群容灾

5.4.1 TiDB 容灾方案概述

本文将以如下结构系统介绍TiDB 容灾解决方案：

- 介绍容灾解决方案涉及的基本概念。
- 介绍核心组件TiDB、TiCDC 及 BR 的架构。
- 介绍各种容灾方案。
- 对比不同的容灾解决方案。

5.4.1.1 基本概念

- RTO (Recovery Time Objective)：是指灾难发生后，系统恢复服务所需的时间。
- RPO (Recovery Point Objective)：是指灾难发生后，确保对业务不产生损失的前提下，可以丢失的最大数据量。

下面的图形描述了这两个概念：

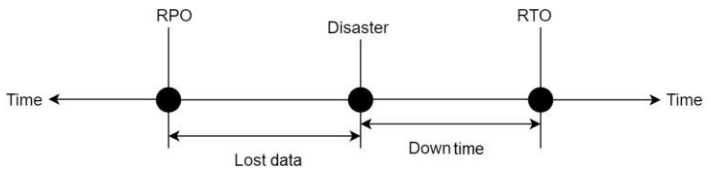


图 24: RTO and RPO

- 错误容忍目标：由于灾难可能影响的地域范围是不同的，在本文中，使用“错误容忍目标”来描述系统能够容忍的灾难的最大范围。
- 区域：本文主要讨论区域(region)级别的容灾方案，这里的区域通常是指一个物理世界中的地区或者城市。

5.4.1.2 组件架构

在介绍具体的容灾方案之前，本部分将从容灾角度介绍容灾系统中的组件架构，包括TiDB、TiCDC 和 BR。

5.4.1.2.1 TiDB 架构

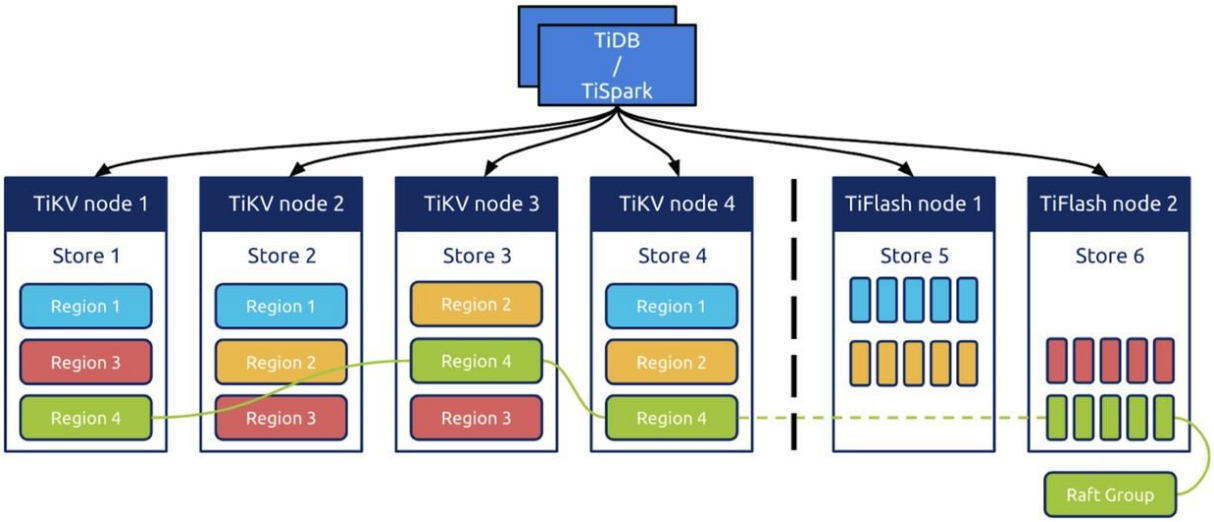


图 25: TiDB architecture

TiDB 的设计采用了计算、存储分离的架构：

- TiDB 为系统的计算层。
- TiKV 是系统的存储层，采用行存的方式保存数据库的数据记录，其中Region 是经过排序的若干行数据的集合，也是系统调度数据的单位。同一个Region 的数据保存至少 3 份，通过Raft 协议在日志层复制数据改变。
- TiFlash 副本是可选的，它是一款列式存储，用于加速分析类查询的速度。数据通过Raft group 中的learner 角色与TiKV 中的数据进行复制。

由于TiDB 保存了三份完整的数据副本，所以天然就具备了基于多副本数据复制的容灾能力。同时，由于TiDB 采用了Raft log 来进行事务日志同步，也在一定程度上具备了基于事务日志同步的容灾能力。

5.4.1.2.2 TiCDC 架构

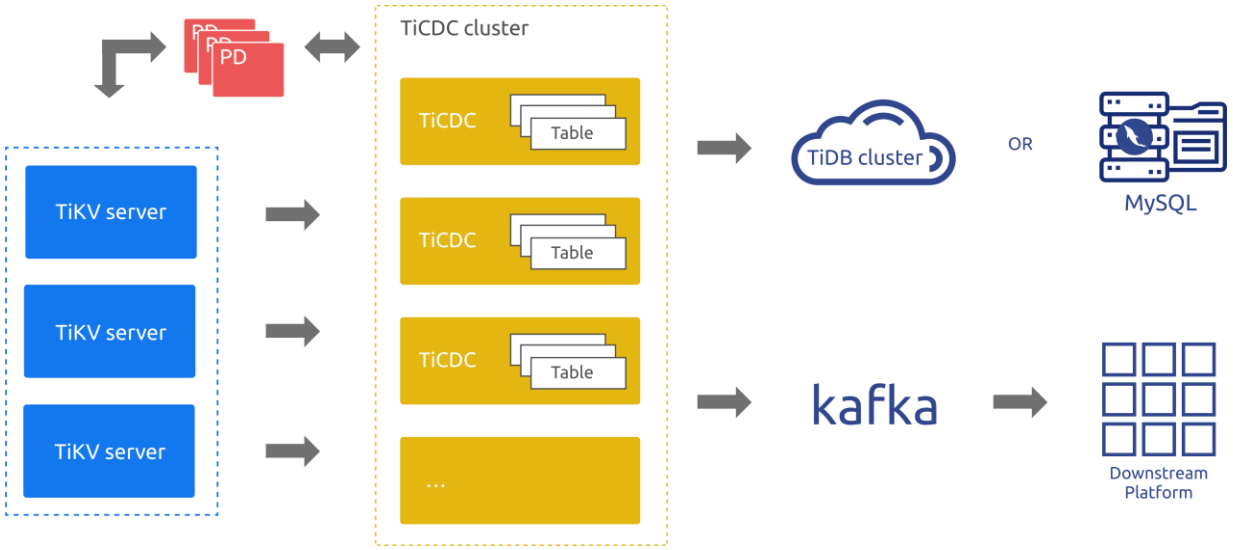


图 26: TiCDC architecture

TiCDC 作为 TiDB 的增量数据同步工具，通过 PD 内部的 etcd 实现高可用，通过多个 Capture 进程获取 TiKV 节点上的数据改变，在内部进行排序、合并等处理之后，通过多个同步任务，同时向多个下游系统进行数据同步。在上面的架构中：

- TiKV server：负责将对应节点上数据的改变推送到 TiCDC 节点。当然，如果 TiCDC 发现收到的数据改变不完整，也会主动联系 TiKV server 获取需要的数据改变。
- TiCDC：负责启动多个 Capture 进程，每个 Capture 负责拉取一部分的 KV change logs，并对获取到的数据改变进行排序，最后同步到不同的下游当中。

从上面的架构中可以看到，TiCDC 的架构和事务日志复制系统比较类似，但是扩展性更好，同时又兼顾了逻辑数据复制的很多特点。所以，TiCDC 也可以为 TiDB 在容灾场景提供很好的帮助和补充。

5.4.1.2.3 BR 架构

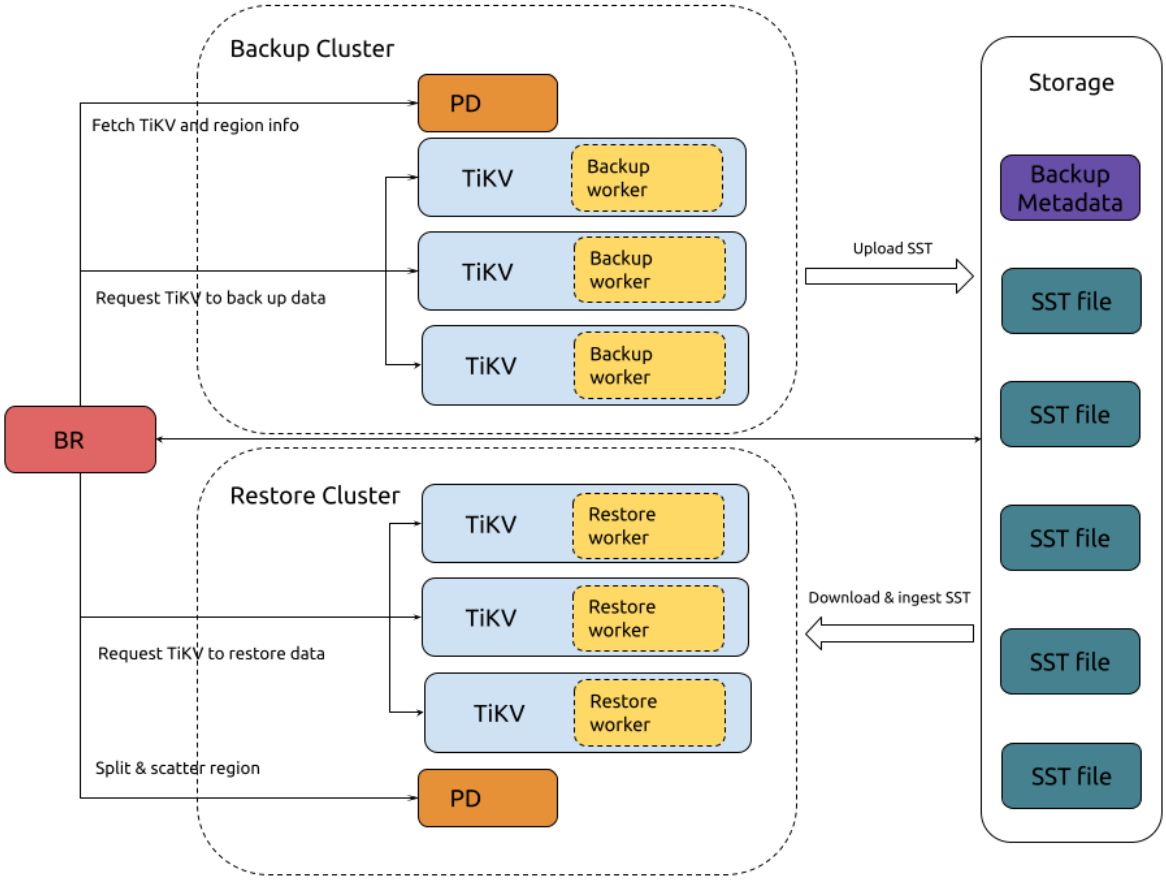


图 27: BR architecture

BR 作为TiDB 的备份恢复工具，可以对TiDB 集群进行基于时间点的全量快照备份和持续的日志备份，从而保护TiDB 集群的数据。当TiDB 集群完全不可用时，可以通过备份文件，在全新的集群中进行恢复。备份恢复通常是数据安全的最后一道防线。

5.4.1.3 方案介绍

5.4.1.3.1 基于TiCDC 的主备集群容灾方案

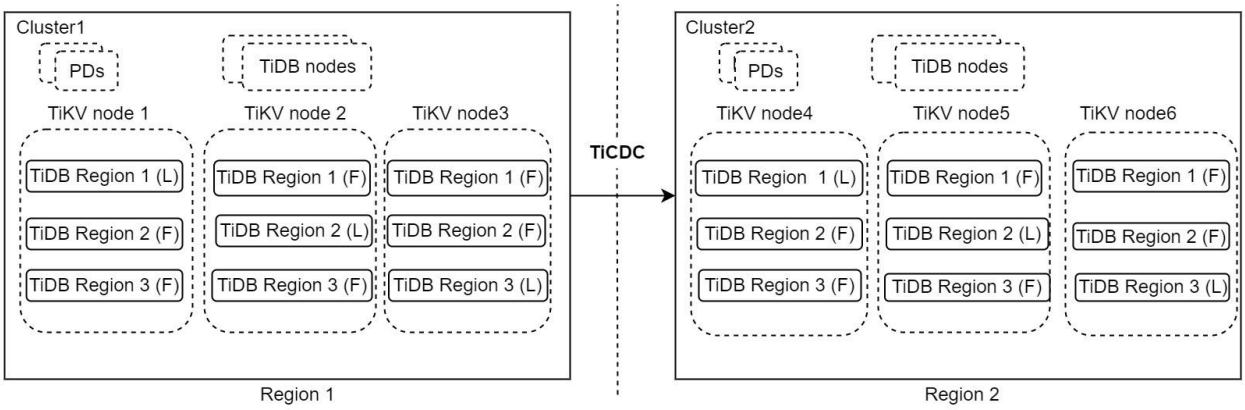


图 28: Primary-secondary cluster DR

在上面的架构中包含了两个TiDB 集群，Cluster1 为主用集群，运行在区域 1 (Region 1)，包含 3 个副本，承担读写业务。Cluster2 作为灾备集群，运行在区域 2 (Region 2)。当 Cluster1 出现灾难时，Cluster2 继续对外提供服务。两个集群之间通过TiCDC 进行数据改变的同步。这种架构，简称为“1:1” 解决方案。

这种架构看起来非常简洁，可用性比较高，最大的错误容忍目标可以做到区域级别，写能力也能够得到扩展，RPO 在秒级别，RTO 在分钟级别，甚至更低。如果RPO 为 0 并不是必须满足的要求，推荐在重要生产系统使用该容灾方案。对于该方案的详细信息，请参考基于主备集群的容灾方案。

5.4.1.3.2 基于多副本的单集群容灾方案

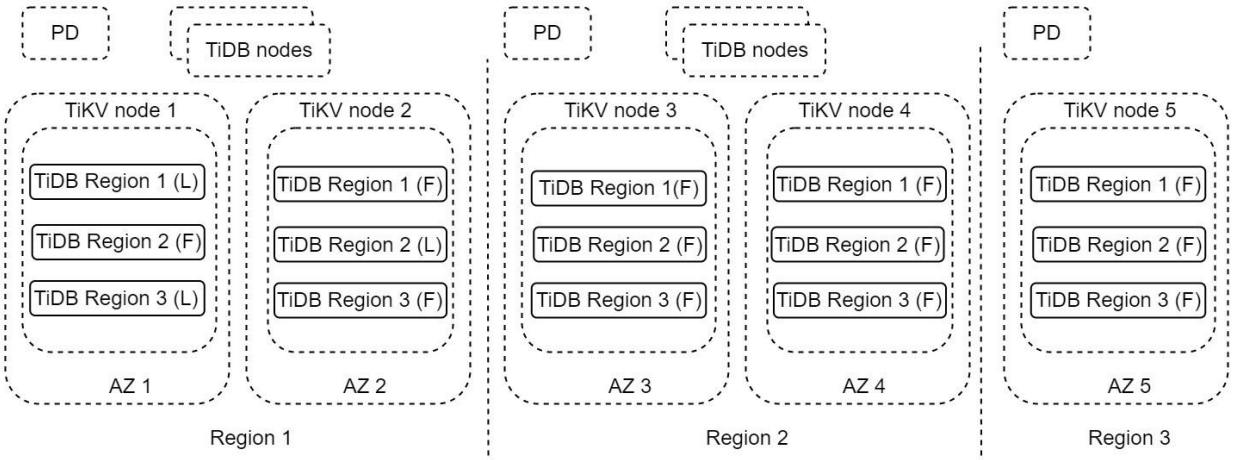


图 29: Multi-replica cluster DR

在上面的架构中，每个区域都包含两份完整的数据副本，它们位于不同的可用区(Available Zone, AZ) 当中（通常情况下，两个可用区之间的网络速度和带宽条件较好，在同一个区域中的不同AZ 中读写请求的延迟很低），整个集群横跨了三个区域。区域 1 通常是用来处理读写业务请求的主区域，当区域 1 出现灾难后完全不可用时，区域 2 可以作为灾难恢复的区域。而区域 3 (Region 3) 更多的是为了满足多数派协议而存在的一个副本。这

种架构，简称为“2-2-1”解决方案。

该方案最大的错误容忍目标可以达到区域级别，写能力也能够得到扩展，并且RPO 为 0，RTO 也可以达到分钟级别，甚至更低。如果RPO 为 0 是必须满足的要求，推荐在重要生产系统使用该容灾方案。对于该方案的详细信息，请参考基于多副本的单集群容灾方案。

5.4.1.3.3 多副本与TiCDC 相结合的容灾解决方案

以上两种容灾解决方案都可以实现区域级别的容灾，但是都无法解决多个区域同时不可用的问题。如果你的系统非常重要，需要“错误容忍目标”达到多个区域，就需要将以上两种容灾解决方案进行结合。

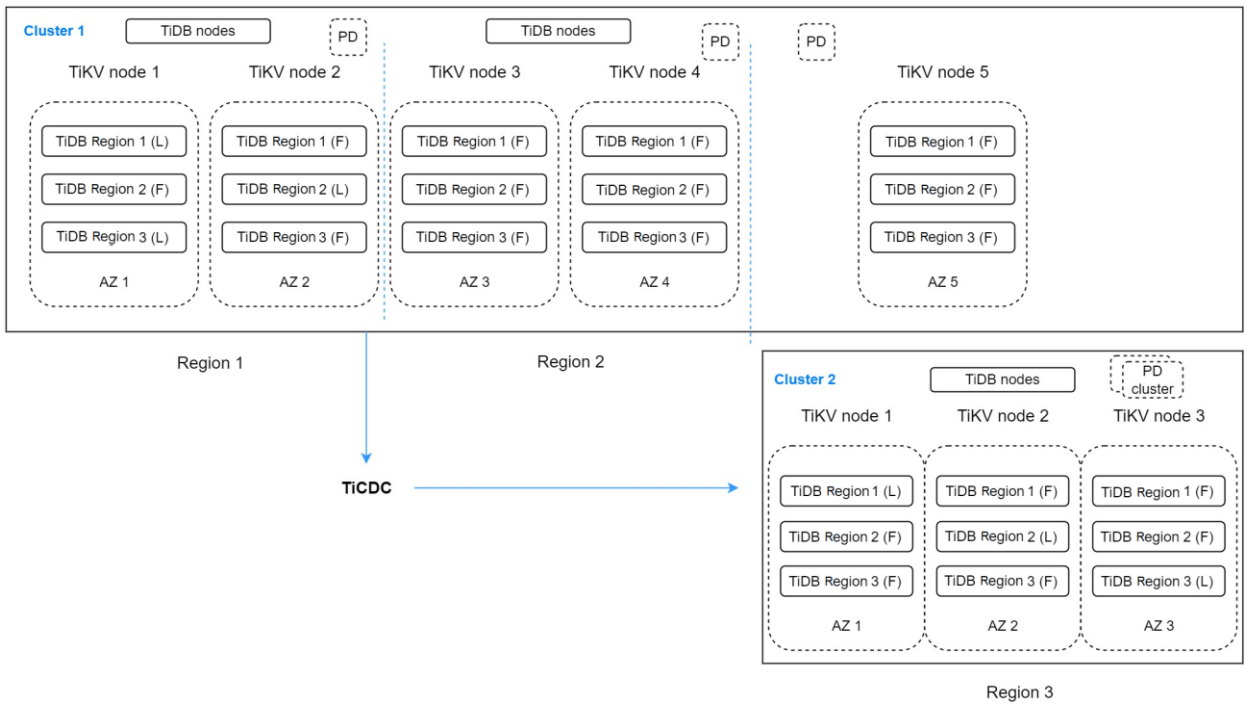


图 30: TiCDC-based multi-replica cluster DR

在上面的部署中存在两个TiDB 集群。Cluster1 有 5 个副本，跨 3 个区域。区域 1 (Region 1) 包含两个副本作为主区域，用于服务写入。区域 2 (Region 2) 有两个副本作为区域 1 的容灾区域，可以提供一些延迟不敏感的读取服务。最后一个副本用于投票，位于区域 3 (Region 3) 中。

作为区域 1 和区域 2 的容灾集群，Cluster2 在区域 3 中运行，并包含 3 个副本。TiCDC 在两个集群之间同步数据更改。这种部署可能看起来比较复杂，但它可以将容错目标提高到多区域。如果多区域故障不要求RPO 必须为 0，这种架构是一个很好的选择。这种架构，简称为“2-2-1:1”解决方案。

当然，如果“错误容忍目标”为多个区域，并且RPO 为 0 是一个必须满足的要求，你也可以考虑创建一个包含至少 9 个副本，横跨 5 个区域的集群来实现该能力。这种架构，简称为“2-2-2-2-1”解决方案。

5.4.1.3.4 基于备份与恢复的容灾解决方案

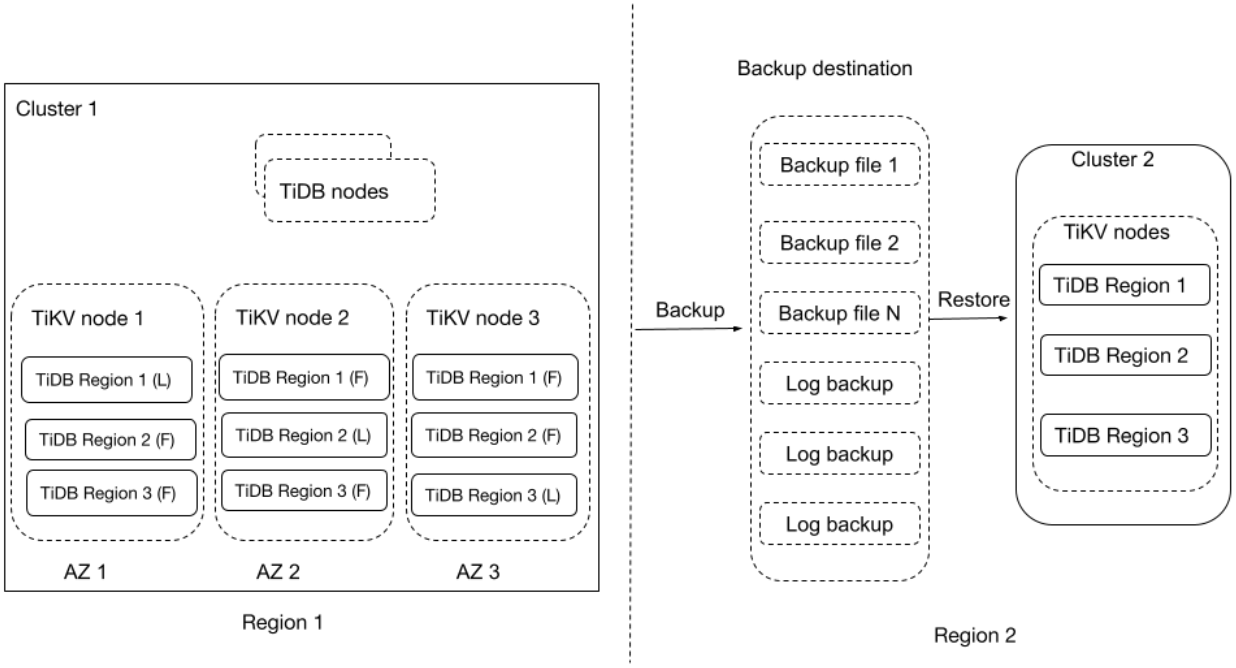


图 31: BR-based cluster DR

按照上面的部署，TiDB Cluster1 部署在区域 1 (Region 1)，BR 工具定期将集群的数据备份到区域 2 (Region 2)，并且持续将数据改变日志也备份到区域 2。当区域 1 出现灾难导致 Cluster1 无法恢复时，你可以使用备份的数据和数据改变在区域 2 恢复新的集群 Cluster2 对外提供服务。

基于备份恢复的容灾方案，目前，RPO 低于 5 分钟，而RTO 则取决于需要恢复的集群数据大小，对于 v6.5.0 版本的 BR，其恢复速度可以参考快照恢复的性能与影响和PITR 的性能与影响。通常来说，大部分客户会把跨区域的备份作为数据安全的最后一道防线，是大多数系统都需要的。对于该方案的详细信息，请参考基于备份与恢复的容灾方案。

另外，从 v6.5.0 开始，BR 支持基于AWS 上的EBS 快照的快速恢复。如果你在AWS 上运行TiDB 集群，要求备份过程对集群没有任何影响，并且要求恢复的时间尽量短，可以考虑使用该特性来降低系统的RTO。

5.4.1.3.5 其他容灾解决方案

除了以上容灾方案，针对同城双中心这种特定的场景，如果 RPO=0 是一个必须的条件，你也可以采用 DR-AUTO sync 解决方案。详细的信息请参考单区域双AZ 部署TiDB。

5.4.1.4 方案对比

最后，对本文提到的各种容灾解决方案进行对比，以方便你根据自己的业务需要选择合适的容灾方案。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
基于多副本的单集群容灾方案 (2-2-1)	高	单个区域	0	分钟级	区域之间的网络延迟要求小于 30 ms。	对灾备和响应时间有明确要求 (RPO = 0) 的重要生产系统。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
基于 TiCDC 的主备集群容灾方案 (1:1)	中等	单个区域	小于 10 秒	小于 5 分钟	区域之间的网络延迟要求小于 100 ms。	对灾备和响应时间有明确要求 (RPO > 0) 的重要生产系统。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
多副本与 TiCDC 相结合的容灾解决方案 (2-2-1:1)	高	多个区域	小于 10 秒	小于 5 分钟	对于通过多副本进行容灾的区域，网络延迟建议小于 30 ms。对于第三区域与其他区域之间，建议延迟小于 100	对灾难恢复和响应时间有严格要求的关键生产系统。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
基于备份恢复的容灾方案	低	单个区域	小于 5 分钟	小时级	无特殊要求	能够接受 RPO < 5 分钟，RTO 达到小时级别的系统。

5.4.2 基于主备集群的容灾方案

使用主、备数据库进行容灾是一种常用的容灾方式。在这种方案下，系统有一个主用集群和一个备用集群。主集群用于处理用户的请求，备集群负责备份主集群的数据。当主集群发生故障时，系统可以切换到备集群，使用备份的数据继续提供服务。这样，系统就可以在发生故障的情况下继续正常运行，避免因为故障而导致的服务中断。

主备集群容灾方案具有如下优势：

- 高可用性：主、备集群的架构可以有效提高系统的可用性，使得系统在遇到故障时能够快速恢复。
- 快速切换：在主集群发生故障的情况下，系统可以快速切换到备用集群，继续提供服务。
- 数据一致性：备用集群会近实时备份主集群的数据，因此，在故障发生后切换到备集群时，数据基本是最新的。

本文包含以下主要内容：

- 构建主备集群
- 从主集群复制数据至备集群
- 监控集群

- 容灾切换

同时，本文还介绍了如何在备用集群上进行业务查询，以及如何在主备集群间进行双向同步。

5.4.2.1 基于TiCDC构建TiDB主备集群

5.4.2.1.1 架构概览

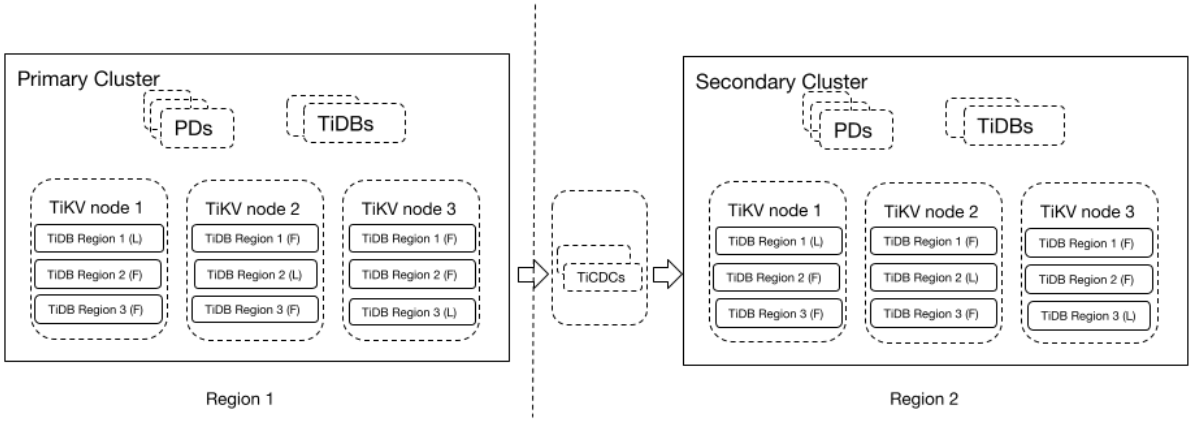


图 32: TiCDC secondary cluster architecture

上述架构包含两个TiDB集群：Primary Cluster 和 Secondary Cluster。

- Primary Cluster：主用集群，运行在区域 1 (Region 1)，三副本，用于处理读写业务。
- Secondary Cluster：备用集群，运行在区域 2 (Region 2)，通过TiCDC从 Primary Cluster 同步数据。

这种容灾架构简洁易用，可以容忍区域级别的故障，既可以保证主用集群的写入性能不会下降，还可以在备用集群处理一些延迟不敏感的只读业务。该方案的 Recovery Point Objective (RPO) 在秒级别，Recovery Time Objective (RTO) 可以达到分钟级别甚至更低。这个方案适用于重要的生产系统。

> 注意：
>
> 不要使用多个TiCDC Changefeed同步数据至备用集群，也不要在备用集群基础上运行另一个备用集群，否则，备用集群的数据事务完整性无法保证。

5.4.2.1.2 搭建主备集群

本文将TiDB 主集群和备用集群分别部署在两个不同的区域（区域 1 和区域 2）。由于主备集群之间存在一定的网络延迟，TiCDC 与TiDB 备用集群应部署在一起，以实现最好的数据同步性能。在本教程示例中，每台服务器部署一个组件节点，具体的部署拓扑如下：

区域	主机	集群	组件
区域 1	10.0.1.1/10.0.1.2/10.0.1.3	Primary	PD
区域 1	10.0.1.4/10.0.1.5	Primary	TiDB
区域 1	10.0.1.6/10.0.1.7/10.0.1.8	Primary	TiKV
区域 1	10.0.1.9	Primary	Monitor、 Grafana 或 AlterManager
区域 2	10.1.1.9/10.1.1.10	Primary	TiCDC
区域 2	10.1.1.1/10.1.1.2/10.1.1.3	Secondary	PD
区域 2	10.1.1.4/10.1.1.5	Secondary	TiDB
区域 2	10.1.1.6/10.1.1.7/10.1.1.8	Secondary	TiKV
区域 2	10.0.1.11	Secondary	Monitor、 Grafana 或 AlterManager

关于服务器配置信息，可以参考如下文档：

- TiDB 软件和硬件环境建议配置
- TiCDC 软件和硬件环境推荐配置

部署TiDB 主集群和备用集群的详细过程，可以参考部署TiDB 集群。

部署TiCDC 组件需要注意的是，Secondary Cluster 和 TiCDC 需要在一起部署和管理，并且它们之间的网络需要能够连通。

- 如果需要在已有的 Primary Cluster 上部署TiCDC，请参考部署TiCDC 组件。
- 如果部署全新的 Primary Cluster 和TiCDC 组件，则可以使用以下 TiUP 部署模版，并按照需要修改配置参数：

```
global:
user: "tidb"
ssh_port: 22
deploy_dir: "/tidb-deploy"
data_dir: "/tidb-data"
server_configs: {}
pd_servers:
- host: 10.0.1.1
- host: 10.0.1.2
- host: 10.0.1.3
tidb_servers:
- host: 10.0.1.4
- host: 10.0.1.5
tikv_servers:
- host: 10.0.1.6
```

```
- host: 10.0.1.7
- host: 10.0.1.8
monitoring_servers:
- host: 10.0.1.9
grafana_servers:
- host: 10.0.1.9
alertmanager_servers:
- host: 10.0.1.9
cdc_servers:
- host: 10.1.1.9
    gc-ttl: 86400
    data_dir: "/cdc-data"
    ticdc_cluster_id: "DR_TiCDC"
- host: 10.1.1.10
    gc-ttl: 86400
    data_dir: "/cdc-data"
    ticdc_cluster_id: "DR_TiCDC"
```

5.4.2.1.3 从主集群复制数据到备用集群

搭建好TiDB 主集群和备用集群之后，需要先将主集群的数据迁移到备用集群，然后创建同步任务从主集群复制实时变更数据到备用集群。

选择外部存储

数据迁移和实时变更数据复制都需要使用外部存储。推荐使用Amazon S3 作为存储系统。如果TiDB 集群部署在自建机房中，则推荐以下方式：

- 搭建 MinIO 作为备份存储系统，使用 S3 协议将数据备份到 MinIO 中。
- 挂载 NFS 盘（如 NAS）到 br、TiKV 和TiCDC 实例节点，使用 POSIX 文件系统接口将备份数据写入对应的 NFS 目录中。

下面以 MinIO 为示例，仅供参考。注意需要在区域 1 或者区域 2 中准备独立的服务器部署 MinIO。

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
### 配置访问 MinIO 的 access-key 和 access-secret-id
export HOST_IP='10.0.1.10' # 替换为实际部署 MinIO 的机器 IP 地址
export MINIO_ROOT_USER='minio'
export MINIO_ROOT_PASSWORD='miniostorage'
### 创建 TiCDC redo log 和 backup 数据保存的目录，其中 redo、backup 为 bucket 名字
mkdir -p data/redo
mkdir -p data/backup
### 启动 MinIO，暴露端口在 6060
nohup ./minio server ./data --address :6060 &
```

上述命令启动了一个单节点的 MinIO server 模拟 S3 服务，相关参数为：

- endpoint：http://10.0.1.10:6060/
- access-key：minio
- secret-access-key：miniostorage
- bucket：redo/backup

其访问链接为：

```
s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=http://10.0.1.10:6060&force-
    ↪ path-style=true
```

数据迁移

在TiDB 主备集群之间使用TiDB 备份恢复功能 BR 进行数据迁移。

1. 关闭垃圾回收机制(GC)。为了保证增量迁移过程中新写入的数据不丢失，在开始备份之前，需要关闭上游集群的GC 机制，以确保系统不再清理历史数据。

   执行如下命令关闭GC：

   ```
   SET GLOBAL tidb_gc_enable=FALSE;
   ```

   查询 tidb_gc_enable 的取值，以确认GC 是否已关闭：

   ```
   SELECT @@global.tidb_gc_enable;
   ```

   输出结果为 0 表明GC 已关闭：

   ```
   +-------------------------------------+
   | @@global.tidb_gc_enable |
   +-------------------------------------+
   |                       0 |
   +-------------------------------------+
   1 row in set (0.00 sec)
   ```

> 注意：
> 在生产集群中，关闭GC 机制和备份操作会一定程度上降低集群的读性能。建议在业务低峰期进行备份，并设置合适的 RATE_LIMIT 限制备份操作对线上业务的影响。

2. 备份数据。在TiDB 主集群中执行 BACKUP 语句备份数据：

   ```
   BACKUP DATABASE * TO ``s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=
       ↪ http://10.0.1.10:6060&force-path-style=true``;
   ```

   ```
   +----------------------+----------+--------------------+---------------------+---------------------------------------- +
       ↪
   | Destination          | Size     | BackupTS           | Queue Time          | Execution
       ↪ Time       |
   ```

```
+----------------------+----------+--------------------+--------------------+------------------------------------ +
    ↪
| s3://backup          | 10315858 | 431434047157698561 | 2022-02-25 19:57:59 | 2022-02-25
    ↪ 19:57:59 |
+----------------------+----------+--------------------+--------------------+------------------------------------ +
    ↪
1 row in set (2.11 sec)
```

备份语句提交成功后，TiDB 会返回关于备份数据的元信息，这里需要重点关注 BackupTS，它意味着该时间点之前的数据会被备份。本文后续步骤中，使用 BackupTS 作为实时变更数据复制的起始时间点。

3. 恢复数据。在TiDB 备用集群中执行 RESTORE 语句恢复数据：

```
RESTORE DATABASE * FROM ``s3://backup?access-key=minio&secret-access-key=miniostorage&
    ↪ endpoint=http://10.0.1.10:6060&force-path-style=true``;
```

```
+----------------------+----------+----------+--------------------+------------------------------------ +
| Destination          | Size     | BackupTS | Queue Time          | Execution Time       |
+----------------------+----------+----------+--------------------+------------------------------------ +
| s3://backup          | 10315858 | 0        | 2022-02-25 20:03:59 | 2022-02-25 20:03:59 |
+----------------------+----------+----------+--------------------+------------------------------------ +
1 row in set (41.85 sec)
```

复制实时变更数据

完成以上数据迁移的操作步骤后，从备份的 BackupTS 时间点开始同步主集群增量变更数据到备用集群。

1. 创建TiCDC 同步任务 Changefeed。

   创建 Changefeed 配置文件并保存为 changefeed.toml。

```
[consistent]
# eventual consistency: 使用 redo log，提供上游灾难情况下的最终一致性。
level = "eventual"
# 单个 redo log 文件大小，单位 MiB，默认值 64，建议该值不超过 128。
max-log-size = 64
# 刷新或上传 redo log 至 S3 的间隔，单位毫秒，默认 1000，建议范围 500-2000。
flush-interval = 2000
# 存储 redo log 的地址
storage = "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint=http
    ↪ ://10.0.1.10:6060&force-path-style=true"
```

   在主用集群中，执行以下命令创建从主用集群到备用集群的同步链路：

```
tiup cdc cli changefeed create --server=http://10.1.1.9:8300 --sink-uri="mysql://{username
    ↪ }:{password}@10.1.1.4:4000" --changefeed-id="dr-primary-to-secondary" --start-ts
    ↪ ="431434047157698561"
```

更多关于 Changefeed 的配置，请参考TiCDC Changefeed 配置参数。

2. 查询 Changefeed 是否正常运行。使用 `changefeed query` 命令可以查询特定同步任务（对应某个同步任务的信息和状态），指定 `--simple` 或 `-s` 参数会简化输出，提供基本的同步状态和checkpoint 信息。不指定该参数会输出详细的任务配置、同步状态和同步表信息。

```
tiup cdc cli changefeed query -s --server=http://10.1.1.9:8300 --changefeed-id="dr-primary-
    ↪ to-secondary"
```

```
{
"state": "normal",
"tso": 431434047157998561, # changefeed 已经同步到的时间点
"checkpoint": "2020-08-27 10:12:19.579", # TSO 对应的物理时间点
"error": null
}
```

3. 重新开启GC。

   TiCDC 可以保证未同步的历史数据不会被回收。因此，创建完从主集群到备用集群的 Changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。

执行如下命令打开GC：

```
```sql
SET GLOBAL tidb_gc_enable=TRUE;
```

查询 `tidb_gc_enable` 的取值，判断 GC 是否已开启：

```sql
SELECT @@global.tidb_gc_enable;
```

结果输出 `1` 表明 GC 已开启：

```
+-------------------------------------------+
| @@global.tidb_gc_enable |
+-------------------------------------------+
|                         1 |
+-------------------------------------------+
1 row in set (0.00 sec)
```
```

5.4.2.1.4 主备集群状态监控

TiDB 目前还没有提供 DR Dashboard，你可以通过以下 Dashboard 了解TiDB 主备集群的状态，从而决定是否需要进行容灾切换：

- TiDB 集群运行状态 Dashboard
- Changefeed 运行状态

5.4.2.1.5 容灾切换

本部分介绍容灾演练，遇到真正灾难时的主备切换，以及重建灾备集群的步骤。

计划中的主备切换

定期对非常重要的业务系统进行容灾演练，检验系统的可靠性是非常有必要的。下面是容灾演练推荐的操作步骤，因为没有考虑演练中业务写入是否为模拟、业务访问数据库是否使用proxy 服务等，与实际的演练场景会有出入，请根据你的实际情况进行修改。

1. 停止主集群上的业务写入。
2. 业务写入完全停止后，查询TiDB 集群当前最新的TSO (Position)：

```
mysql> show master status;
+-------------+--------------------+--------------+------------------+-------------------------------------------+
| File        | Position           | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-------------+--------------------+--------------+------------------+-------------------------------------------+
| tidb-binlog | 438223974697009153 |              |                  |                   |
+-------------+--------------------+--------------+------------------+-------------------------------------------+
1 row in set (0.33 sec)
```

3. 轮询 Changefeed dr-primary-to-secondary 的同步位置时间点TSO 直到满足 TSO >= Position。

```
tiup cdc cli changefeed query -s --server=http://10.1.1.9:8300 --changefeed-id="dr-primary-
    ↪ to-secondary"


{
    "state": "normal",
    "tso": 438224029039198209,  # Changefeed 已经同步到的时间点
    "checkpoint": "2022-12-22 14:53:25.307", # TSO 对应的物理时间点
    "error": null
}
```

4. 停止 Changefeed dr-primary-to-secondary。通过删除 Changefeed 的方式，暂停 Changefeed dr-primary-to
    ↪ -secondary：

```
tiup cdc cli changefeed remove --server=http://10.1.1.9:8300 --changefeed-id="dr-primary-to-
    ↪ secondary"
```

5. 创建 Changefeed dr-secondary-to-primary。不需要指定 Changefeed start-ts 参数，Changefeed 从当前时间开始同步即可。
6. 修改业务应用的数据库访问配置，并重启业务应用，使得业务访问备用集群。
7. 检查业务状态是否正常。

容灾演练后，再重复一遍以上步骤，即可恢复原有的系统主备配置。

真正灾难中主备切换

当发生真正的灾难，比如主集群所在区域停电，主备集群的同步链路可能会突然中断，从而导致备用集群数据处于事务不一致的状态。

1. 恢复备用集群到事务一致的状态。在区域 2 的任意TiCDC 节点执行以下命令，以向备用集群重放 redo log，使备用集群达到最终一致性状态：

```
tiup cdc redo apply --storage "s3://redo?access-key=minio&secret-access-key=miniostorage&
    ↪ endpoint=http://10.0.1.10:6060&force-path-style=true" --tmp-dir /tmp/redo --sink-uri
    ↪ "mysql://{username}:{password}@10.1.1.4:4000"
```

   命令中参数描述如下：

   - `--storage`：指定 redo log 所在的 S3 位置
   - `--tmp-dir`：为从 S3 下载redo log 的缓存目录
   - `--sink-uri`：指定备份集群的地址

2. 修改业务应用的数据库访问配置，并重启业务应用，使得业务访问备用集群。
3. 检查业务状态是否正常。

灾难后重建主备集群

当TiDB 主集群所遭遇的灾难解决后，或者主集群暂时不能恢复，此时TiDB 集群是脆弱的，只有一个备用集群临时作为新的主集群提供服务。为了维持系统的可靠性，需要重建灾备集群保护系统的可靠性。

目前，重建TiDB 主备集群，通用的方案是重新部署一个新的集群，组成新的容灾主备集群。操作请参考：

1. 搭建主备集群。
2. 从主集群复制数据到备用集群。
3. 完成以上操作步骤后，如果你希望新集群成为主集群，那么请参考主从切换。

> 注意：
>
> 如果在业务上能够修正灾难发生后主集群和备用集群的数据不一致的问题，那么也可以使用修正后的集群重建主备集群，而不需要重建新集群。

5.4.2.1.6 在备用集群上进行业务查询

在主备集群容灾场景中，将备用集群作为只读集群来运行一些延迟不敏感的查询是常见的需求，TiDB 主备集群容灾方案也提供了这种功能。

创建 Changefeed 时，你只需要在配置文件中开启 Syncpoint 功能，Changefeed 就会定期(`sync-point-interval`)在备用集群中通过执行 `SET GLOBAL tidb_external_ts = @@tidb_current_ts` 设置已复制完成的一致性快照点。

当业务需要从备用集群查询数据的时候，在业务应用中设置`SET GLOBAL|SESSION tidb_enable_external_ts_read ↪ = ON;` 就可以在备用集群上获得事务状态完成的数据。

```
### 从 v6.4.0 开始支持，使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者
    ′→ SUPER 权限
enable-sync-point = true

### 记录主集群和备用集群一致快照点的时间间隔，它也代表能读到完整事务的最大延迟时间，
    ′→比如在备用集群读取到主集群两分钟之前的事务数据
### 配置格式为 h m s，例如 "1h30m30s"。默认值为 10m，最小值为 30s
sync-point-interval = "10m"

### Syncpoint 功能在下游表中保存的数据的时长，超过这个时间的数据会被清理
### 配置格式为 h m s，例如 "24h30m30s"。默认值为 24h
sync-point-retention = "1h"

[consistent]
### eventual consistency： 使用 redo log，提供上游灾难情况下的最终一致性。
level = "eventual"
### 单个 redo log 文件大小，单位 MiB，默认值 64，建议该值不超过 128。
max-log-size = 64
### 刷新或上传 redo log 至 S3 的间隔，单位毫秒，默认 1000，建议范围 500-2000。
flush-interval = 2000
### 存储 redo log
storage = "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint=http
    ′→ ://10.0.1.10:6060&force-path-style=true"
```

> 注意：
>
> 在主备集群容灾架构中，每个备用集群只能被一个 Changefeed 同步数据，否则就无法保证备用集群的事务完整性。

5.4.2.1.7 在主备集群之间进行双向复制

在主备集群容灾场景中，部分用户希望让两个区域的TiDB 集群互为灾备集群：用户的业务流量按其区域属性写入对应的TiDB 集群，同时两套TiDB 集群备份对方集群的数据。

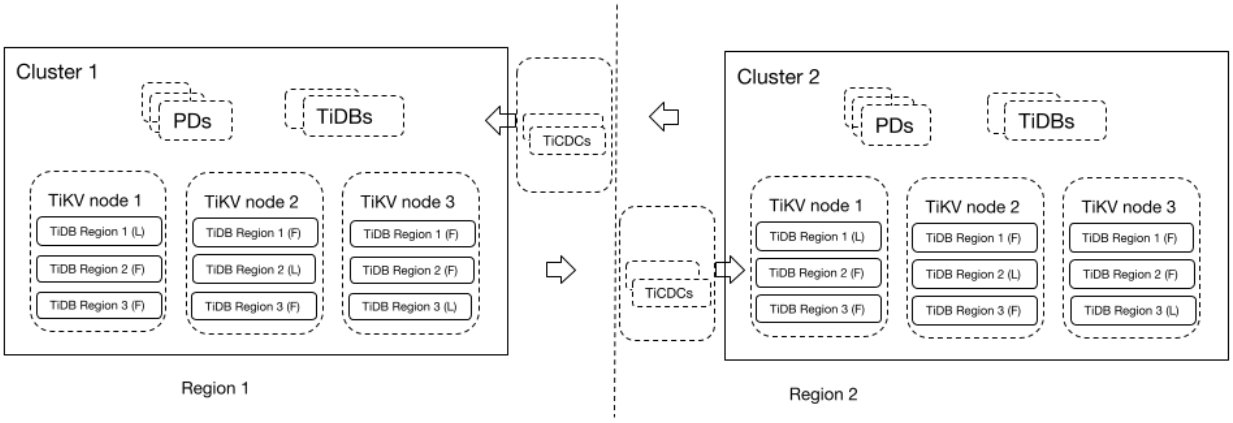


图 33: TiCDC bidirectional replication

在双向复制容灾集群方案中，两个区域的TiDB 集群互相备份对方的数据，使得它们可以在故障发生时互为灾备集群。这种方案既能满足安全性和可靠性的需求，同时也能保证数据库的写入性能。在计划中的主备切换场景中，不需要停止正在运行的 Changefeed 和启动新的 Changefeed 等操作，在运维上也更加简单。

搭建双向容灾复制集群的步骤，请参考教程TiCDC 双向复制。

5.4.2.2 常见问题处理

以上任何步骤遇到问题，可以先通过TiDB FAQ 查找问题的处理方法。如果问题仍不能解决，请在TiDB 项目中提 issue。

5.4.3 基于多副本的单集群容灾方案

本文介绍了基于多副本的单集群容灾方案，文档内容组织如下：

- 方案简介
- 搭建集群
- 配置副本
- 监控集群
- 容灾切换

5.4.3.1 简介

对于重要的生产系统，很多用户需要能够实现区域级别的容灾，并且做到RPO = 0 和分钟级别的RTO。TiDB 作为基于Raft 协议的分布式数据库，其自带的多副本特性可以用于支持区域级别的容灾目标，并同时确保数据的一致性和高可用性。而同区域可用区(Available Zone, AZ) 之间的网络延迟相对较小，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。

5.4.3.2 搭建集群和配置副本

在这一部分当中，会以一个 5 副本的集群为例，演示如何使用TiUP 创建一个跨 3 个区域的集群，以及如何控制数据和 PD 的分布位置，从而达到容灾的目的。

在下面的示例中，TiDB 集群的区域 1 作为 primary region，区域 2 作为 secondary region，而区域 3 则作为投票使用的第三个区域，一共包含 5 个副本。同理，PD 集群也包含了 5 个副本，其功能和TiDB 集群的功能基本一致。

1. 创建类似于以下的集群拓扑文件：

```
global:
  user: "root"
  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:  ["Region","AZ"] # PD 会根据 TiKV 节点的 Region 和 AZ
        ↪ 配置来进行副本的调度。

pd_servers:
  - host: tidb-dr-test1
    name: "pd-1"
  - host: tidb-dr-test2
    name: "pd-2"
  - host: tidb-dr-test3
    name: "pd-3"
  - host: tidb-dr-test4
    name: "pd-4"
  - host: tidb-dr-test5
    name: "pd-5"

tidb_servers:
  - host: tidb-dr-test1
  - host: tidb-dr-test3

tikv_servers: # 在 TiKV 节点中通过 labels 选项来对每个 TiKV 节点所在的 Region 和 AZ
    ↪ 进行标记
  - host: tidb-dr-test1
    config:
      server.labels: { Region: "Region1", AZ: "AZ1" }
  - host: tidb-dr-test2
    config:
      server.labels: { Region: "Region1", AZ: "AZ2" }
```

```
  - host: tidb-dr-test3
    config:
      server.labels: { Region: "Region2", AZ: "AZ3" }
  - host: tidb-dr-test4
    config:
      server.labels: { Region: "Region2", AZ: "AZ4" }
  - host: tidb-dr-test5
    config:
      server.labels: { Region: "Region3", AZ: "AZ5" }

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

monitoring_servers:
  - host: tidb-dr-test2

grafana_servers:
  - host: tidb-dr-test2

alertmanager_servers:
  - host: tidb-dr-test2
```

在上面的配置中，使用了以下一系列配置来针对跨区域容灾场景进行优化：

- 使用 `server.grpc-compression-type: gzip` 启用TiKV 之间的消息压缩，从而降低网络流量。
- 使 用 `raftstore.raft-min-election-timeout-ticks` 和 `raftstore.raft-max-election-timeout-ticks` 延长区域 3 参加选举的时间，从而避免该区域中的副本被选举为主节点。

2. 使用上面的配置文件创建集群：

```
tiup cluster deploy drtest v6.4.0 ./topo.yaml
tiup cluster start drtest --init
tiup cluster display drtest
```

对集群的副本数和 Leader 限制进行配置：

```
tiup ctl:v6.4.0 pd config set max-replicas 5
tiup ctl:v6.4.0 pd config set label-property reject-leader Region Region3

# 下面的步骤用于向集群中添加一些测试数据，可选
tiup bench tpcc  prepare -H 127.0.0.1 -P 4000 -D tpcc --warehouses 1
```

指定 PD leader 的优先级:

```
tiup ctl:v6.4.0 pd member leader_priority  pd-1 4
tiup ctl:v6.4.0 pd member leader_priority  pd-2 3
tiup ctl:v6.4.0 pd member leader_priority  pd-3 2
tiup ctl:v6.4.0 pd member leader_priority  pd-4 1
```

```
tiup ctl:v6.4.0 pd member leader_priority  pd-5 0
```

> **注意：**
> 优先级数值越大的节点成为leader 的可能性越高。

3. 创建placement rule，并将测试表的主副本固定在区域 1：

```
-- 创建两个 placement rules，第一个是区域 1 作为主区域，在系统正常时使用，第二个是区域 2
    ↪ 作为备区域。
-- 作为主区域，当区域 1 出现问题时，区域 2 会作为主区域。
MySQL [(none)]> CREATE PLACEMENT POLICY primary_rule_for_region1 PRIMARY_REGION="Region1"
    ↪ REGIONS="Region1, Region2,Region3";
MySQL [(none)]> CREATE PLACEMENT POLICY secondary_rule_for_region2 PRIMARY_REGION="Region2"
    ↪ REGIONS="Region1,Region2,Region3";

-- 将刚刚创建的规则 primary_rule_for_region1 应用到对应的用户表上。
ALTER TABLE tpcc.warehouse PLACEMENT POLICY=primary_rule_for_region1;
ALTER TABLE tpcc.district PLACEMENT POLICY=primary_rule_for_region1;

-- 说明：请根据需要修改上面的数据库名称、表名和 placement rule 的名称。

-- 使用类似下面的查询，用户可以查看每个区域包含的 leader 数量，以确认 leader 迁移是否完成。
SELECT STORE_ID, address, leader_count, label FROM TIKV_STORE_STATUS ORDER BY store_id;
```

下面的语句可以产生一个 SQL 脚本，把所有非系统 schema 中的表的leader 都设置到特定的区域上：

```
SET @region_name=primary_rule_for_region1;
SELECT concat('ALTER TABLE ', table_schema, '.', table_name, ' PLACEMENT POLICY=',
    ↪ @region_name, ';') FROM information_schema.tables WHERE table_schema NOT IN ('
    ↪ METRICS_SCHEMA', 'PERFORMANCE_SCHEMA', 'INFORMATION_SCHEMA','mysql');
```

5.4.3.3 监控集群

对于部署的集群，你可以通过访问集群中的Grafana 地址或者TiDB Dashboard 组件来对集群中的各个TiKV、 TiDB 和 PD 组件的各种性能指标进行监控。根据组件的状态，确定是否进行容灾切换。详细信息，请参考如下文档：

- TiDB 重要监控指标详解
- TiKV 监控指标详解
- PD 重要监控指标详解
- TiDB Dashboard 监控页面

5.4.3.4 容灾切换

本部分介绍容灾切换，包括计划内切换和计划外切换。

5.4.3.4.1 计划内切换

指根据维护需要进行的主备区域切换，可用于验证容灾系统是否可以正常工作。本部分介绍如何在计划内切换主备区域。

1. 执行如下命令，将所有用户表和 PD Leader 都切换到区域 2：

```
-- 将之前创建的规则 secondary_rule_for_region2 应用到对应的用户表上。
ALTER TABLE tpcc.warehouse PLACEMENT POLICY=secondary_rule_for_region2;
ALTER TABLE tpcc.district PLACEMENT POLICY=secondary_rule_for_region2;
```

说明：请根据需要修改上面的数据库名称、表名和placement rule 的名称。

执行如下命令，调低区域 1 的 PD 节点的优先级，并调高区域 2 的 PD 节点的优先级。

```
tiup ctl:v6.4.0 pd member leader_priority pd-1 2
tiup ctl:v6.4.0 pd member leader_priority pd-2 1
tiup ctl:v6.4.0 pd member leader_priority pd-3 4
tiup ctl:v6.4.0 pd member leader_priority pd-4 3
```

2. 观察Grafana 中 PD 和 TiKV 部分中的内容，确保 PD 的 Leader 和用户表的 Leader 已经迁移到对应的区域。另外，切换回原有区域的步骤与上面的步骤基本相同，本文不做过多的描述。

5.4.3.4.2 计划外切换

计划外切换，指灾难发生时的主备区域切换，或者为了验证容灾系统的有效性，而模拟灾难发生时的主备区域切换。

1. 执行类似下面的命令终止区域 1 上所有的TiKV、 TiDB 和 PD 节点:

```
tiup cluster stop drtest -N tidb-dr-test1:20160,tidb-dr-test2:20160,tidb-dr-test1:2379,tidb-
    ↪ dr-test2:2379
```

2. 运行类似于下面的命令切换用户表的leader 到区域 2:

```
-- 将之前创建的规则 secondary_rule_for_region2 应用到对应的用户表上。
ALTER TABLE tpcc.warehouse PLACEMENT POLICY=secondary_rule_for_region2;
ALTER TABLE tpcc.district PLACEMENT POLICY=secondary_rule_for_region2;

---可以使用类似下面的查询查看每个区域包含的 leader 数量，以确认 leader 迁移是否完成。
SELECT STORE_ID, address, leader_count, label FROM TIKV_STORE_STATUS ORDER BY store_id;
```

当区域 1 恢复正常之后，可以使用类似于上面的命令将用户表的leader 重新切换到区域 1。

5.4.4 基于备份与恢复的容灾方案

TiDB 集群自身的多副本特性可以让其容忍单个机房或地理区域的故障，并继续向外提供服务。在更大范围的自然灾害、软件漏洞、硬件故障、病毒攻击或常见的人为误操作的情况下，TiDB 的备份与恢复功能可以将数据备份到独立的灾备存储设备中，以保护用户数据不受损坏。与其他方案相比，备份与恢复功能具有灵活性、可靠性、可恢复性和成本效益等优势：

- 灵活性：可以在不同的时间点进行备份，并且可以根据需要调整备份频率。这使得备份与恢复功能更加灵活，能够更好地适应不同的业务场景。
- 可靠性：备份与恢复功能通常会将备份数据保存在独立的存储设备上，使得数据安全得到进一步升级。
- 可恢复性：任何意外情况导致的原始数据丢失或损坏，都可以通过恢复备份数据的方式来恢复。这使得备份恢复功能具有很高的可恢复性，能够保证数据库的正常使用。
- 成本效益：备份恢复功能通常具有较低的成本，可以在不增加太多费用的情况下实现对数据库的安全保护。

总的来说，备份恢复功能是数据安全的最后一道防线，可以在不增加太多成本的情况下，提高数据库的安全性和可靠性。它能够在各种意外情况发生时保护数据，使你能够安心地使用数据库，而不必担心数据丢失或损坏的风险。

5.4.4.1 进行备份恢复

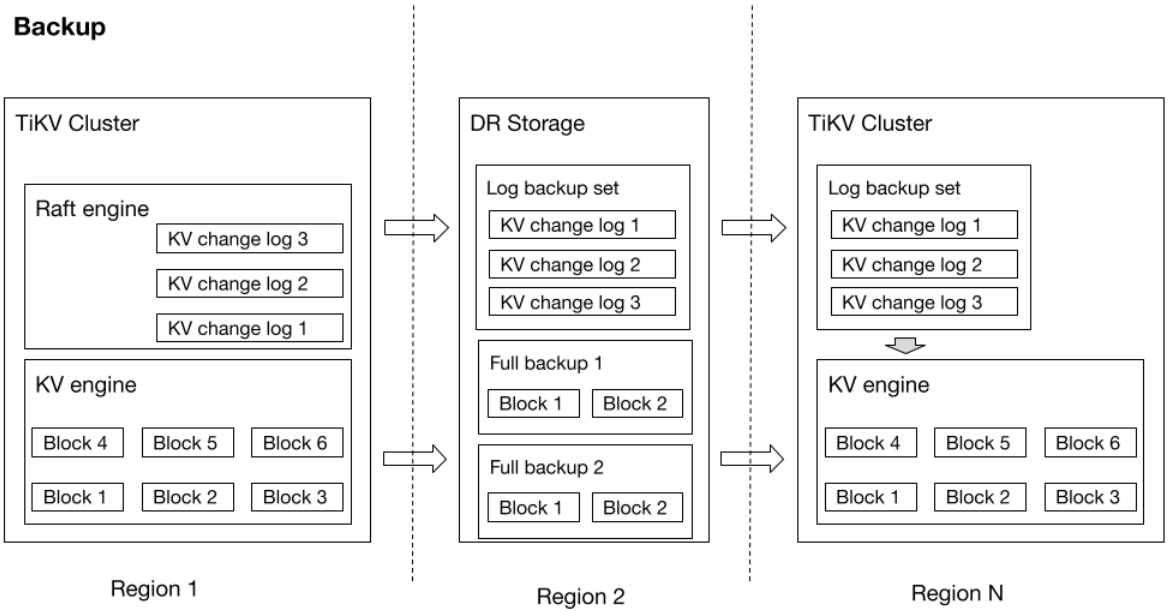


图 34: BR log backup and PITR architecture

按照上述架构，你可以将数据备份到其他区域的灾备存储设备中，并在需要时将数据恢复回来。这样，系统就能够容忍单个区域的故障，并且Recovery Point Objective (RPO) 可以达到 5 分钟，Recovery Time Objective (RTO) 通常在几十分钟到数小时之间。但是，如果数据库尺寸较大，RTO 时间可能会更长。

此外，TiDB 还提供了基于块存储快照技术的备份和恢复特性，可以将集群的恢复时间缩短到小时级别甚至 1 小时以内。TiDB 也在不断完善和优化基于存储快照技术的备份和恢复能力，以提供更好的服务。

TiDB 提供了丰富的文档，帮助你了解如何在容灾场景下使用备份和恢复功能。其中，

- TiDB 备份和恢复功能使用概述提供了备份与恢复的使用指引，包含备份策略及备份数据存放组织等。
- TiDB 备份恢复常见问题列出了使用备份与恢复功能时可能遇到的问题，以及解决方案。
- TiDB 备份与恢复功能架构描述了TiDB 备份与恢复功能的设计架构，包括备份和恢复的流程，以及备份文件设计。

5.5 使用资源管控(Resource Control) 实现资源隔离

使用资源管控特性，集群管理员可以定义资源组(Resource Group)，通过资源组限定配额。

TiDB 资源管控特性提供了两层资源管理能力，包括在TiDB 层的流控能力和TiKV 层的优先级调度的能力。两个能力可以单独或者同时开启，详情请参见参数组合效果表。将用户绑定到某个资源组后，TiDB 层会根据用户所绑定资源组设定的配额对用户的读写请求做流控，TiKV 层会根据配额映射的优先级来对请求做调度。通过流控和调度这两层控制，可以实现应用的资源隔离，满足服务质量(QoS) 要求。

- TiDB 流控：TiDB 流控使用令牌桶算法做流控。如果桶内令牌数不够，而且资源组没有指定 BURSTABLE 特性，属于该资源组的请求会等待令牌桶回填令牌并重试，重试可能会超时失败。
- TiKV 调度：你可以为资源组设置绝对优先级 (PRIORITY)，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。如果没有设置绝对优先级(PRIORITY)，TiKV 会将资源组的 RU_PER_SEC取值映射成各自资源组读写请求的优先级，并基于各自的优先级在存储层使用优先级队列调度处理请求。

5.5.1 使用场景

资源管控特性的引入对TiDB 具有里程碑的意义。它能够将一个分布式数据库集群划分成多个逻辑单元，即使个别单元对资源过度使用，也不会挤占其他单元所需的资源。利用该特性：

- 你可以将多个来自不同系统的中小型应用合入一个TiDB 集群中，个别应用的负载升高，不会影响其他业务的正常运行。而在系统负载较低的时候，繁忙的应用即使超过设定的读写配额，也仍然可以被分配到所需的系统资源，达到资源的最大化利用。
- 你可以选择将所有测试环境合入一个集群，或者将消耗较大的批量任务编入一个单独的资源组，在保证重要应用获得必要资源的同时，提升硬件利用率，降低运行成本。
- 当系统中存在多种业务负载时，可以将不同的负载分别放入各自的资源组。利用资源管控技术，确保交易类业务的响应时间不受数据分析或批量业务的影响。
- 当集群遇到突发的 SQL 性能问题，可以结合 SQL Binding 和资源组，临时限制某个 SQL 的资源消耗。

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。

5.5.2 使用限制

目前，资源管控特性具有以下限制:

- 只支持对前台客户发起的读写请求做限流和调度，不支持对 DDL 以及Auto Analyze 等后台任务的限流和调度。
- 资源管控将带来额外的调度开销。因此，开启该特性后，性能可能会有轻微下降。

5.5.3 什么是Request Unit (RU)

Request Unit (RU) 是 TiDB 对 CPU、IO 等系统资源的统一抽象的单位，目前包括 CPU、IOPS 和 IO 带宽三个指标。这三个指标的消耗会按照一定的比例统一到RU 单位上。

下表是用户请求对TiKV 存储层 CPU 和 IO 资源的消耗以及对应的RU 权重：

资源	RU 权重
CPU	1/3 RU / 毫秒
读 IO	1/64 RU / KB
写 IO	1 RU / KB
一次读请求的基本开销	0.25 RU
一次写请求的基本开销	1.5 RU

基于上表，假设某个资源组消耗的TiKV 时间是c 毫秒，r1 次请求读取了 r2 KB 数据，w1 次写请求写入了 w2 KB 数据，集群中非 witness TiKV 节点数是n，则该资源组消耗的总RU 的公式如下：

c* 1/3 + (r1 * 0.25 + r2 * 1/64) + (w1 * 1.5 + w2 * 1 * n)

5.5.4 相关参数

资源管控特性引入了两个新的全局开关变量：

- TiDB：通过配置全局变量`tidb_enable_resource_control` 控制是否打开资源组流控。
- TiKV：通过配置参数`resource-control.enabled` 控制是否使用基于资源组配额的请求调度。

从 v7.0.0 开始，两个开关都被默认打开。这两个参数的组合效果见下表：

`resource-control.enabled`	`tidb_enable_resource_control=` ON	`tidb_enable_resource_control=` OFF
`resource-control.enabled=` true	流控和调度（推荐组合）	无效配置
`resource-control.enabled=` false	仅流控（不推荐）	特性被关闭

关于资源管控实现机制及相关参数的详细介绍，请参考RFC: Global Resource Control in TiDB。

5.5.5 使用方法

下面介绍如何使用资源管控特性。

5.5.5.1 预估集群容量

在进行资源规划之前，你需要了解集群的整体容量。TiDB 提供了命令 `CALIBRATE RESOURCE` 用来估算集群容量。目前提供两种估算方式：

- 根据实际负载估算容量
- 基于硬件部署估算容量

可通过TiDB Dashboard 资源管控页面进行查看。详情请参考CALIBRATE RESOURCE 预估方式。

5.5.5.2 管理资源组

创建、修改、删除资源组，需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

你可以通过 CREATE RESOURCE GROUP 在集群中创建资源组。

对于已有的资源组，可以通过 ALTER RESOURCE GROUP 修改资源组的配额，对资源组的配额修改会立即生效。

可以使用 DROP RESOURCE GROUP 删除资源组。

5.5.5.3 创建资源组

下面举例说明如何创建资源组。

1. 创建 rg1 资源组，限额是每秒 500 RU，并且允许这个资源组的应用超额占用资源。

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1 RU_PER_SEC = 500 BURSTABLE;
```

2. 创建 rg2 资源组，RU 的回填速度是每秒 600 RU。在系统资源充足的时候，不允许这个资源组的应用超额占用资源。

```
CREATE RESOURCE GROUP IF NOT EXISTS rg2 RU_PER_SEC = 600;
```

3. 创建 rg3 资源组，设置绝对优先级为 HIGH。绝对优先级目前支持 LOW|MEDIUM|HIGH，资源组的默认绝对优先级为 MEDIUM。

```
CREATE RESOURCE GROUP IF NOT EXISTS rg3 RU_PER_SEC = 100 PRIORITY = HIGH;
```

5.5.5.4 绑定资源组

TiDB 支持如下三个级别的资源组设置：

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

5.5.5.4.1 将用户绑定到资源组

下面的示例创建一个用户 usr1 并将其绑定到资源组 rg1。其中 rg1 为创建资源组示例中创建的资源组。

```
CREATE USER 'usr1'@'%' IDENTIFIED BY '123' RESOURCE GROUP rg1;
```

下面示例使用 ALTER USER 将用户 usr2 绑定到资源组 rg2。其中 rg2 为创建资源组示例中创建的资源组。

```
ALTER USER usr2 RESOURCE GROUP rg2;
```

绑定用户后，用户新建立的会话对资源的占用会受到指定用量(RU)的限制。如果系统负载比较高，没有多余的容量，用户 usr2 的资源消耗速度会被严格控制不超过指定用量。由于 usr1 绑定的 rg1 配置了 BURSTABLE，所以 usr1 消耗速度允许超过指定用量。

如果资源组对应的请求太多导致资源组的资源不足，客户端的请求处理会发生等待。如果等待时间过长，请求会报错。

> 注意：
>
> - 使用 CREATE USER 或者 ALTER USER 将用户绑定到资源组后，只会对该用户新建的会话生效，不会对该用户已有的会话生效。
> - 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。对于没有绑定资源组的语句会自动绑定至此资源组。此资源组不支持删除，但允许修改其RU 的配置。

5.5.5.4.2 将当前会话绑定到资源组

通过把当前会话绑定到资源组，会话对资源的占用会受到指定用量(RU)的限制。

下面的示例将当前的会话绑定至资源组 rg1。

```
SET RESOURCE GROUP rg1;
```

5.5.5.4.3 将语句绑定到资源组

通过在 SQL 语句中添加RESOURCE_GROUP(resource_group_name) Hint，可以将该语句绑定到指定的资源组。此 Hint 支持 SELECT、 INSERT、 UPDATE、 DELETE 四种语句。

示例：

```
SELECT /*+ RESOURCE_GROUP(rg1) */ * FROM t limit 10;
```

5.5.6 关闭资源管控特性

1. 执行以下命令关闭资源管控特性：

   ```
   SET GLOBAL tidb_enable_resource_control = 'OFF';
   ```

2. 将TiKV 参数resource-control.enabled 设为 false，关闭按照资源组配额调度。

5.5.7 监控与图表

TiDB 会定时采集资源管控的运行时信息，并在Grafana 的 Resource Control Dashboard 中提供了相关指标的可视化图表，详见Resource Control 监控指标详解。

TiKV 中也记录了来自于不同资源组的请求 QPS，详见TiKV 监控指标详解。

TiDB Dashboard 中可以查看当前 RESOURCE_GROUPS 表中资源组的数据。详见TiDB Dashboard 资源管控页面。

5.5.8 工具兼容性

资源管控不影响数据导入导出以及其他同步工具的正常使用，BR、 TiDB Lightning、 TiCDC 等工具不支持对资源管控相关 DDL 的处理，这些工具的资源消耗也不受资源管控的限制。

5.5.9 常见问题

1. 如果我暂时不想使用资源组对资源进行管控，是否一定要关闭这个特性？

   不需要。没有指定任何资源组的用户，将被放入系统预定义的 `default` 资源组，而 `default` 资源组默认拥有无限用量。当所有用户都属于 `default` 资源组时，资源分配方式与关闭资源管控时相同。

2. 一个数据库用户是否可以绑定到不同的资源组？

   不能。一个数据库用户只能绑定到一个资源组。但是，在会话运行的过程中，可以通过 `SET RESOURCE GROUP` 设置当前会话使用的资源组。你也可以通过优化器`RESOURCE_GROUP()` Hint 为运行的语句设置资源组。

3. 当各个资源组设置的用量(RU_PER_SEC) 总和超出系统容量会发生什么？

   TiDB 在创建资源组时不会检查容量。只要系统有足够的空闲资源，TiDB 就会满足每个资源组的用量设置。当系统资源超过限制时，TiDB 会优先满足高优先级(PRIORITY) 资源组的请求。如果同一优先级的请求无法全部满足，TiDB 会根据用量(RU_PER_SEC) 的大小按比例分配。

5.5.10 另请参阅

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

5.6 时区支持

TiDB 使用的时区由 `time_zone` 全局变量和 session 变量决定。`time_zone` 的默认值是 `System`，`System` 对应的实际时区在 TiDB 集群bootstrap 初始化时设置。具体逻辑如下：

- 优先使用 TZ 环境变量
- 如果失败，则从 `/etc/localtime` 的实际软链地址提取。
- 如果上面两种都失败则使用 UTC 作为系统时区。

在运行过程中可以修改全局时区：

```
SET GLOBAL time_zone = timezone;
```

TiDB 还可以通过设置 session 变量 time_zone 为每个连接维护各自的时区。默认条件下，这个值取的是全局变量 time_zone 的值。修改 session 使用的时区：

```
SET time_zone = timezone;
```

使用以下 SQL 语句查看当前全局时区、客户端时区和系统时区的值：

```
SELECT @@global.time_zone, @@session.time_zone, @@global.system_time_zone;
```

设置 time_zone 的值的格式：

- ‘SYSTEM’ 表明使用系统时间
- 相对于 UTC 时间的偏移，比如‘+10:00’或者‘-6:00’
- 某个时区的名字，比如‘Europe/Helsinki’，‘US/Eastern’或‘MET’

NOW() 和 CURTIME() 的返回值都受到时区设置的影响。

> 注意：
>
> 只有 Timestamp 数据类型的值是受时区影响的。可以理解为，Timestamp 数据类型的实际表示使用的是（字面值+时区信息）。其它时间和日期类型，比如 Datetime/Date/Time 是不包含时区信息的，所以也不受到时区变化的影响。

```
create table t (ts timestamp, dt datetime);
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
set @@time_zone = 'UTC';
```

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

```
insert into t values ('2017-09-30 11:11:11', '2017-09-30 11:11:11');
```

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

```
set @@time_zone = '+8:00';
```

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

```
select * from t;
```

```
+---------------------|---------------------------------------+
| ts                  | dt                                    |
+---------------------|---------------------------------------+
| 2017-09-30 19:11:11 | 2017-09-30 11:11:11 |
+---------------------|---------------------------------------+
1 row in set (0.00 sec)
```

上面的例子中，无论怎么调整时区的值，Datetime 类型字段的值是不受影响的，而Timestamp 则随着时区改变，显示的值会发生变化。其实Timestamp 持久化到存储的值始终没有变化过，只是根据时区的不同显示值不同。

Timestamp 类型和 Datetime 等类型的值，两者相互转换的过程中，会涉及到时区。这种情况一律基于 session 的当前 `time_zone` 时区处理。

另外，用户在导数据的过程中，也要需注意主库和从库之间的时区设定是否一致。

5.7 日常巡检

TiDB 作为分布式数据库，对比单机数据库机制更加复杂，其自带的监控项也很丰富。为了更便捷地运维TiDB，本文介绍了运维TiDB 集群需要常关注的关键性能指标。

5.7.1 TiDB Dashboard 关键指标

从 4.

使用
面举

5.7.1.



图 35: 实例面板

以上实例面板的各指标说明如下：

- 状态：用于查看实例状态是否正常，如果在线可忽略此项。
- 启动时间：是关键指标。如果有发现启动时间有变动，那么需要排查组件重启的原因。
- 版本、部署路径、Git 哈希值：通过查看这三个指标可以避免有部署路径和版本不一致或者错误的情况。

5.7.1.2 主机面板



图 36: 主机面板

通过主机面板可以查看 CPU、内存、磁盘使用率。当任何资源的使用率超过 80% 时，推荐扩容对应组件。

5.7.1.3 SQL 分析面板



图 37: SQL 分析面板

通过 SQL 分析面板可以分析对集群影响较大的慢 SQL，然后进行对应的 SQL 优化。

5.7.1.4 Region 信息面板

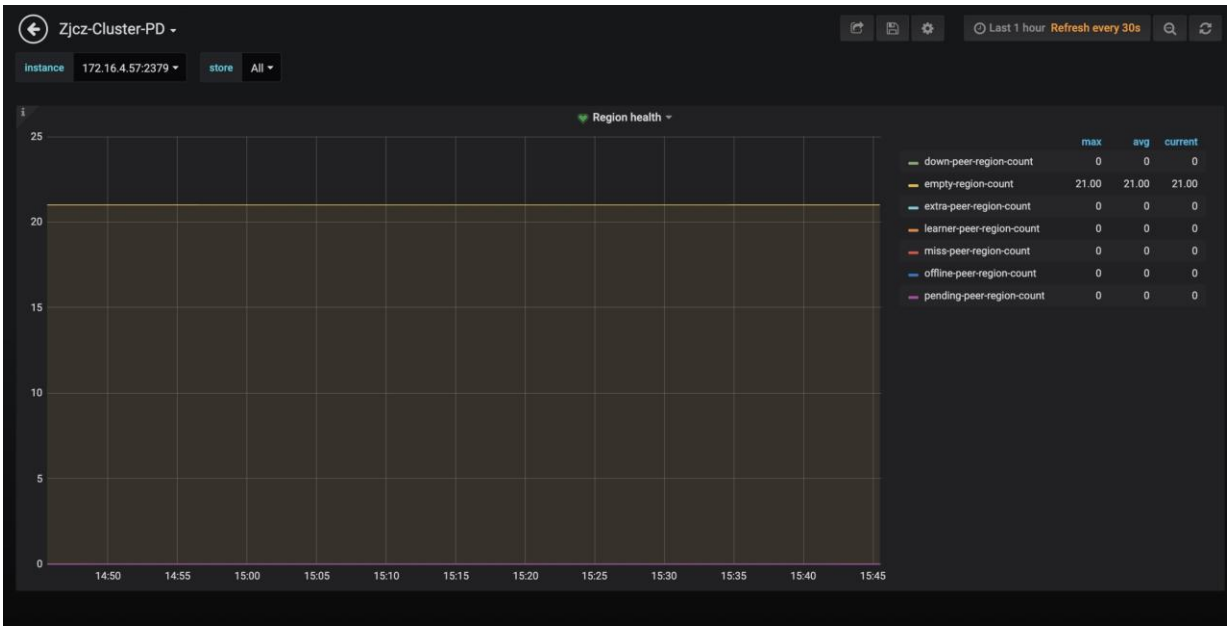


图 38: Region 信息面板

以上 Region 信息面板说明如下：

- `miss-peer-region-count`：缺副本的 Region 数量，不会一直大于 0。
- `extra-peer-region-count`：多副本的 Region 数量，调度过程中会产生。
- `empty-region-count`：空 Region 的数量，一般是 `TRUNCATE TABLE/DROP TABLE` 语句导致。如果数量较多，可以考虑开启跨表Region merge。
- `pending-peer-region-count`：Raft log 落后的Region 数量。由于调度产生少量的 pending peer 是正常的，但是如果pending peer 的数量持续（超过 30 分钟）很高，可能存在问题。
- `down-peer-region-count`：Raft leader 上报有不响应peer 的 Region 数量。
- `offline-peer-region-count`：peer 下线过程中的Region 数量。

原则上来说，该监控面板偶尔有数据是符合预期的。但长期有数据，需要排查是否存在问题。

5.7.1.5 KV Request Duration

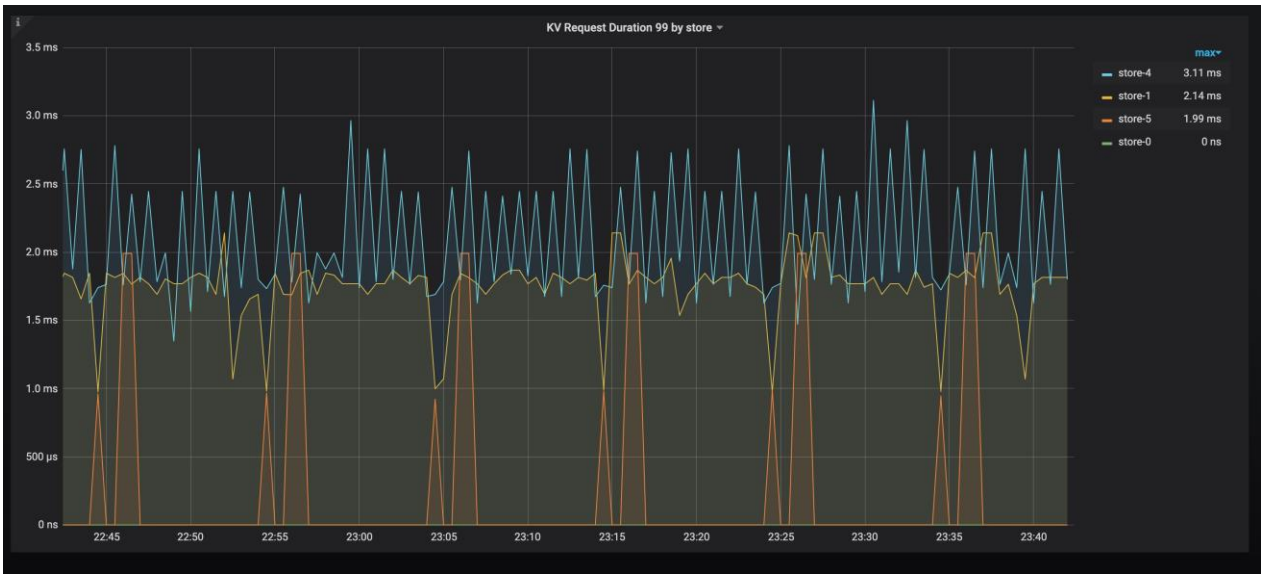


图 39: TiKV 相应时间

TiKV 当前.99（百分位）的响应时间。如果发现有明显高的节点，可以排查是否有热点，或者相关节点性能较差。

5.7.1.6 PD TSO Wait Duration

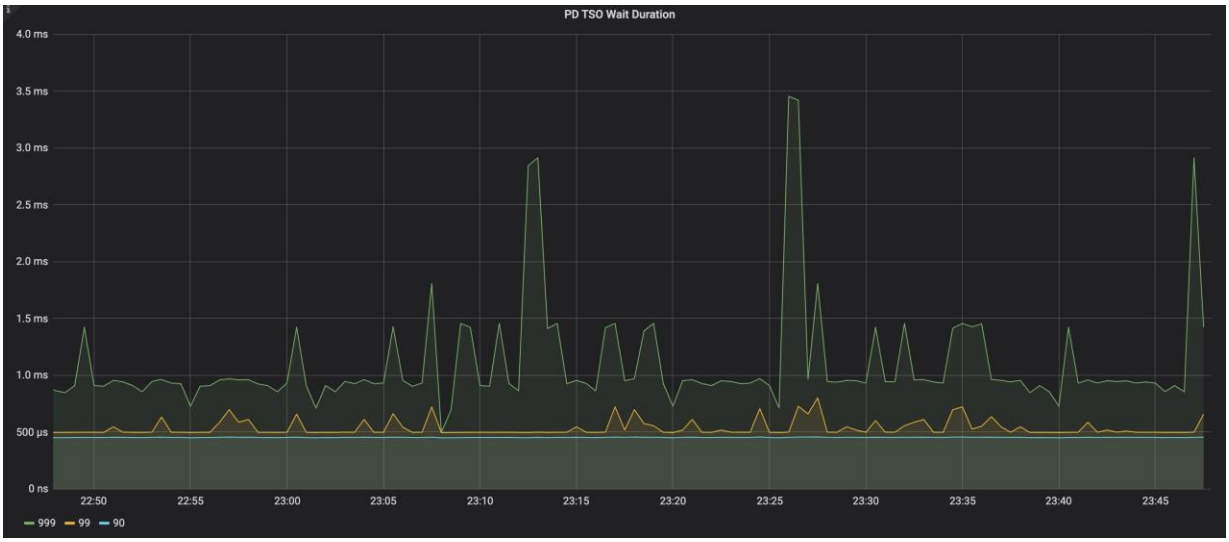


图 40: TiDB 从 PD 获取TSO 的时间

TiDB 从 PD 获取TSO 的时间。如果相关响应时间较高，一般常见原因如下：

- TiDB 到 PD 的网络延迟较高，可以手动 Ping 一下网络延迟。

- TiDB 压力较高，导致获取较慢。
- PD 服务器压力较高，导致获取较慢。

5.7.1.7 Overview 面板

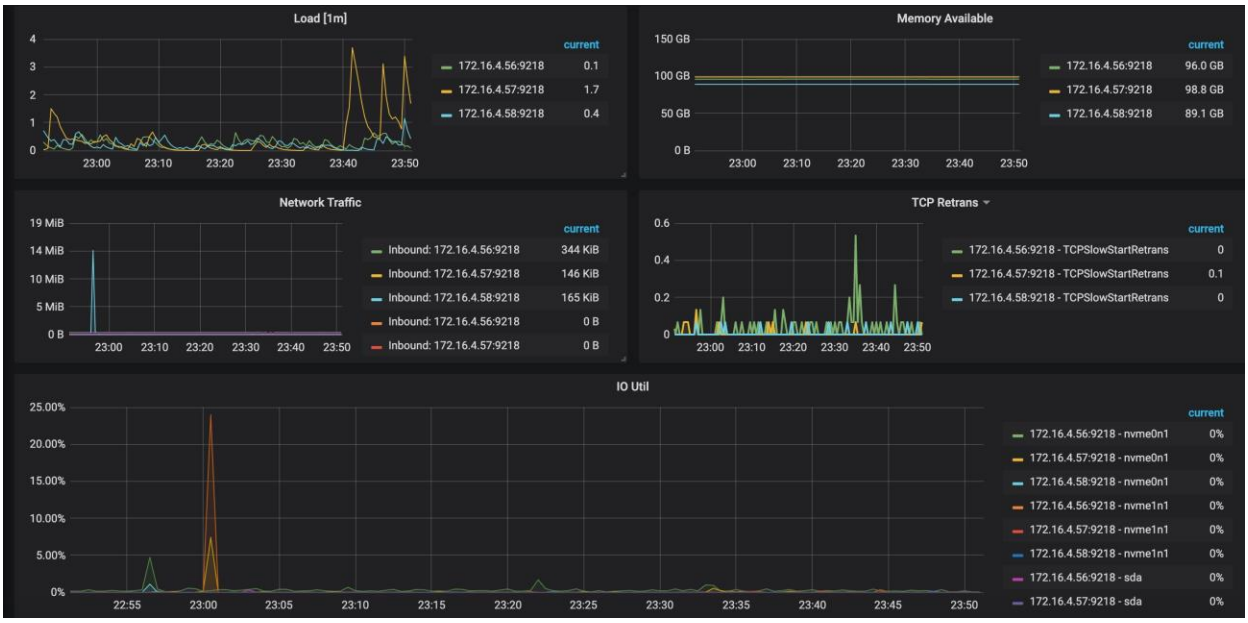


图 41: Overview 面板

以上面板展示常见的负载、内存、网络、IO 监控。发现有瓶颈时，推荐扩容或者优化集群拓扑，优化 SQL、集群参数等。

5.7.1.8 异常监控面板

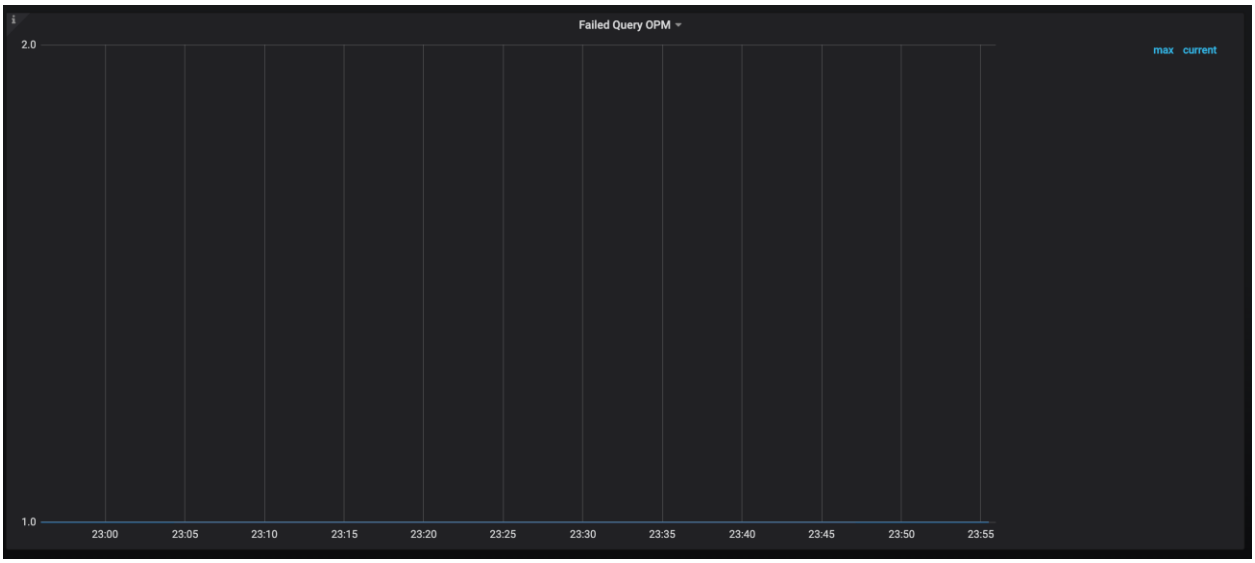


图 42: 异常监控面板

以上面板展示每个TiDB 实例上，执行 SQL 语句发生的错误，并按照错误类型进行统计，例如语法错误、主键冲突等。

5.7.1.9 GC 状态面板

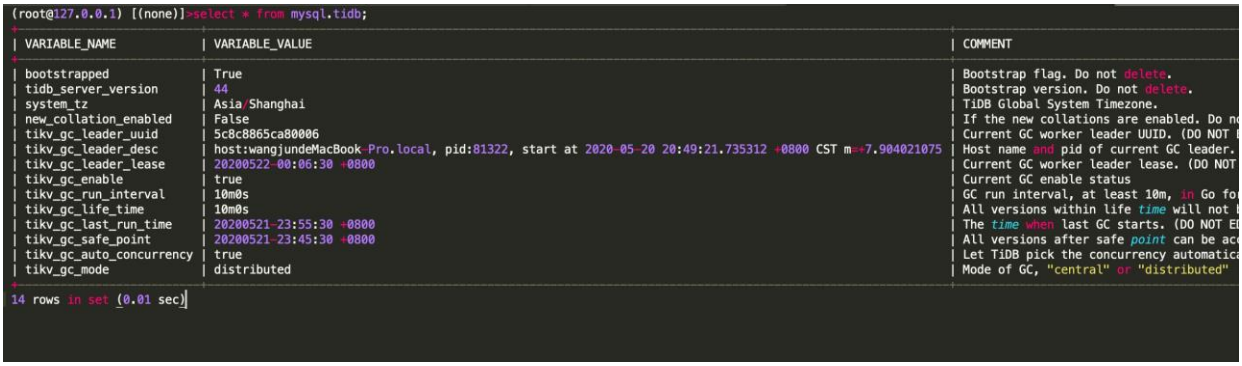


图 43: GC 状态面板

以上面板展示最后GC（垃圾清理）的时间，观察GC是否正常。如果GC 发生异常，可能会造成历史数据存留过多，影响访问效率。

5.8 TiFlash 集群运维

本文介绍TiFlash 集群运维的一些常见操作，包括查看TiFlash 版本、TiFlash 重要日志及系统表。

5.8.1 查看TiFlash 版本

查看TiFlash 版本有以下两种方法：

- 假设TiFlash 的二进制文件名为 `tiflash`，则可以通过 `./tiflash version` 方式获取TiFlash 版本。

  但是由于TiFlash 的运行依赖于动态库 `libtiflash_proxy.so`，因此需要将包含动态库 `libtiflash_proxy.so` 的目录路径添加到环境变量 `LD_LIBRARY_PATH` 后，上述命令才能正常执行。

  例如，当 `tiflash` 和 `libtiflash_proxy.so` 在同一个目录下时，切换到该目录后，可以通过如下命令查看 TiFlash 版本：

```
LD_LIBRARY_PATH=./ ./tiflash version
```

- 在 TiFlash 日志（日志路径见配置文件tiflash.toml [logger] 部分）中查看TiFlash 版本，例如：

```
<information>: TiFlash version: TiFlash 0.2.0 master-375035282451103999f3863c691e2fc2
```

5.8.2 TiFlash 重要日志介绍

日志信息	日志含义
`[INFO] [<unknown>] ′→ ["KVStore: ′→ Start to ′→ persist [region ′→ 47, applied: ′→ term 6 index ′→ 10]"] [ ′→ thread_id=23]`	在 TiFlash 中看到类似日志代表数据开始同步
`[DEBUG] [<unknown ′→ >] [" ′→ CoprocessorHandler ′→ : grpc::Status ′→ DB:: ′→ CoprocessorHandler ′→ ::execute(): ′→ Handling DAG ′→ request"] [ ′→ thread_id=30]`	该日志代表TiFlash 开始处理一个 Coprocessor 请求

日志信息	日志含义
[DEBUG] [<unknown ′→ >] [" ′→ CoprocessorHandler ′→ : grpc::Status ′→ DB:: ′→ CoprocessorHandler ′→ ::execute(): ′→ Handle DAG ′→ request done"] ′→ [thread_id=30]	该日志代表TiFlash 完成 Coprocessor 请求的处理

你可以找到一个 Coprocessor 请求的开始或结束，然后通过日志前面打印的线程号找到该 Coprocessor 请求的其他相关日志。

5.8.3 TiFlash 系统表

`information_schema.tiflash_replica` 系统表的列名及含义如下：

列名	含义
TABLE_SCHEMA	数据库名
TABLE_NAME	表名
TABLE_ID	表 ID
REPLICA_COUNT	TiFlash 副本数
LOCATION_LABELS	给 PD 的 hint，让Region 的多个副本尽可能按照 LOCATION_LABELS 里的设置分散
AVAILABLE	是否可用（0/1）
PROGRESS	同步进度 [0.0~1.0]

5.9 TiUP 常见运维操作

本文介绍了使用TiUP 运维TiDB 集群的常见操作，包括查看集群列表、启动集群、查看集群状态、修改配置参数、关闭集群、销毁集群等。

5.9.1 查看集群列表

TiUP cluster 组件可以用来管理多个TiDB 集群，在每个TiDB 集群部署完毕后，该集群会出现在TiUP 的集群列表里，可以使用list 命令来查看。

```
tiup cluster list
```

5.9.2 启动集群

启动集群操作会按 PD -> TiKV -> Pump -> TiDB -> TiFlash -> Drainer -> TiCDC -> Prometheus -> Grafana -> Alertmanager 的顺序启动整个TiDB 集群所有组件：

```
tiup cluster start ${cluster-name}
```

> 注意：
>
> 你需要将 ${cluster-name} 替换成实际的集群名字，若忘记集群名字，可通过 tiup cluster list 查看。

该命令支持通过 -R 和 -N 参数来只启动部分组件。

例如，下列命令只启动 PD 组件：

```
tiup cluster start ${cluster-name} -R pd
```

下列命令只启动 1.2.3.4 和 1.2.3.5 这两台机器上的 PD 组件：

```
tiup cluster start ${cluster-name} -N 1.2.3.4:2379,1.2.3.5:2379
```

> 注意：
>
> 若通过 -R 和 -N 启动指定组件，需要保证启动顺序正确（例如需要先启动 PD 才能启动TiKV），否则可能导致启动失败。

5.9.3 查看集群状态

集群启动之后需要检查每个组件的运行状态，以确保每个组件工作正常。TiUP 提供了display 命令，节省了登录到每台机器上去查看进程的时间。

```
tiup cluster display ${cluster-name}
```

5.9.4 修改配置参数

集群运行过程中，如果需要调整某个组件的参数，可以使用 edit-config 命令来编辑参数。具体的操作步骤如下：

1. 以编辑模式打开该集群的配置文件：

   ```
   tiup cluster edit-config ${cluster-name}
   ```

2. 设置参数：

   首先确定配置的生效范围，有以下两种生效范围：

   - 如果配置的生效范围为该组件全局，则配置到 server_configs。例如：

   ```
   server_configs:
     tidb:
       log.slow-threshold: 300
   ```

   - 如果配置的生效范围为某个节点，则配置到具体节点的 config 中。例如：

   ```
   tidb_servers:
   - host: 10.0.1.11
     port: 4000
     config:
         log.slow-threshold: 300
   ```

   参数的格式参考TiUP 配置参数模版。

   配置项层次结构使用 . 表示。

   关于组件的更多配置参数说明，可参考 tidb config.toml.example、tikv config.toml.example 和pd config
   ′→ .toml.example。

3. 执行 reload 命令滚动分发配置、重启相应组件：

   ```
   tiup cluster reload ${cluster-name} [-N <nodes>] [-R <roles>]
   ```

5.9.4.1 示例

如果要调整tidb-server 中事务大小限制参数 txn-total-size-limit 为 1G，该参数位于 performance 模块下，调整后的配置如下：

```
server_configs:
  tidb:
    performance.txn-total-size-limit: 1073741824
```

然后执行 tiup cluster reload ${cluster-name} -R tidb 命令滚动重启。

5.9.5 Hotfix 版本替换

常规的升级集群请参考升级文档，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 patch 命令：

```
tiup cluster patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup cluster patch <cluster-name> <package-path> [flags]


Flags:
  -h, --help                    帮助信息
  -N, --node strings            指定被替换的节点
      --overwrite               在未来的 scale-out 操作中使用当前指定的临时包
  -R, --role strings            指定被替换的服务类型
      --transfer-timeout int    transfer leader 的超时时间

Global Flags:
      --native-ssh         使用系统默认的 SSH 客户端
      --wait-timeout int   等待操作超时的时间
      --ssh-timeout int    SSH 连接的超时时间
  -y, --yes                跳过所有的确认步骤
```

例如，有一个TiDB 实例的hotfix 包放在 `/tmp/tidb-hotfix.tar.gz` 目录下。如果此时想要替换集群上的所有 TiDB 实例，则可以执行以下命令：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -R tidb
```

或者只替换其中一个TiDB 实例：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -N 172.16.4.5:4000
```

5.9.6 重命名集群

部署并启动集群后，可以通过 `tiup cluster rename` 命令来对集群重命名：

```
tiup cluster rename ${cluster-name} ${new-name}
```

> 注意：
>
> - 重命名集群会重启监控（Prometheus 和Grafana）。
> - 重命名集群之后Grafana 可能会残留一些旧集群名的面板，需要手动删除这些面板。

5.9.7 关闭集群

关闭集群操作会按Alertmanager -> Grafana -> Prometheus -> TiCDC -> Drainer -> TiFlash -> TiDB -> Pump -> TiKV -> PD 的顺序关闭整个TiDB 集群所有组件（同时也会关闭监控组件）：

```
tiup cluster stop ${cluster-name}
```

和 start 命令类似，stop 命令也支持通过 -R 和 -N 参数来只停止部分组件。

例如，下列命令只停止TiDB 组件：

```
tiup cluster stop ${cluster-name} -R tidb
```

下列命令只停止 1.2.3.4 和 1.2.3.5 这两台机器上的TiDB 组件：

```
tiup cluster stop ${cluster-name} -N 1.2.3.4:4000,1.2.3.5:4000
```

5.9.8 清除集群数据

此操作会关闭所有服务，并清空其数据目录或/和日志目录，并且无法恢复，需要谨慎操作。

清空集群所有服务的数据，但保留日志：

```
tiup cluster clean ${cluster-name} --data
```

清空集群所有服务的日志，但保留数据：

```
tiup cluster clean ${cluster-name} --log
```

清空集群所有服务的数据和日志：

```
tiup cluster clean ${cluster-name} --all
```

清空 Prometheus 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-role prometheus
```

清空节点 172.16.13.11:9000 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-node 172.16.13.11:9000
```

清空部署在 172.16.13.12 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-node 172.16.13.12
```

5.9.9 销毁集群

销毁集群操作会关闭服务，清空数据目录和部署目录，并且无法恢复，需要谨慎操作。

```
tiup cluster destroy ${cluster-name}
```

5.10 在线修改集群配置

在线配置变更主要是通过利用 SQL 对包括TiDB、TiKV 以及 PD 在内的各组件的配置进行在线更新。用户可以通过在线配置变更对各组件进行性能调优而无需重启集群组件。但目前在线修改TiDB 实例配置的方式和修改其他组件(TiKV, PD) 的有所不同。

5.10.1 常用操作

5.10.1.1 查看实例配置

可以通过 SQL 语句 show config 来直接查看集群所有实例的配置信息，结果如下：

```
show config;
```

```
+--
    '→     +                  +                                                    +
    '→
| Type | Instance         | Name                                                  | Value
    '→
    '→ |
+--
    '→     +                  +                                                    +
    '→ -----  ----------------- ---------------------------------------------------------------
| tidb | 127.0.0.1:4001   | advertise-address                                     | 127.0.0.1
    '→
    '→ |
| tidb | 127.0.0.1:4001   | binlog.binlog-socket                                  |
    '→
    '→ |
| tidb | 127.0.0.1:4001   | binlog.enable                                         | false
    '→
    '→ |
| tidb | 127.0.0.1:4001   | binlog.ignore-error                                   | false
    '→
    '→ |
| tidb | 127.0.0.1:4001   | binlog.strategy                                       | range
    '→
    '→ |
| tidb | 127.0.0.1:4001   | binlog.write-timeout                                  | 15s
    '→
    '→ |
| tidb | 127.0.0.1:4001   | check-mb4-value-in-utf8                               | true
    '→
    '→ |

...
```

还可以根据对应的字段进行过滤，如：

```
show config where type='tidb'
show config where instance in (...)
show config where name like '%log%'
show config where type='tikv' and name='log.level'
```

5.10.1.2 在线修改TiKV配置

> 注意：
>
> 在线修改TiKV 配置项后，同时会自动修改TiKV 的配置文件。但还需要使用 `tiup edit-config` 命令来修改对应的配置项，否则 `upgrade` 和 `reload` 等运维操作会将在线修改配置后的结果覆盖。修改配置的操作请参考：使用TiUP 修改配置。执行 `tiup edit-config` 后不需要执行 `tiup reload` 操作。

执行 SQL 语句 `set config`，可以结合实例地址或组件类型来修改单个实例配置或全部实例配置，如：

修改全部TiKV 实例配置：

> 注意：
>
> 建议使用反引号包裹变量名称。

```
set config tikv `split.qps-threshold`=1000
```

修改单个TiKV 实例配置：

```
set config "127.0.0.1:20180" `split.qps-threshold`=1000
```

设置成功会返回 `Query OK`：

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

在批量修改时如果有错误发生，会以 warning 的形式返回：

```
set config tikv `log-level`='warn';
```

```
Query OK, 0 rows affected, 1 warning (0.04 sec)
```

```
show warnings;
```

```
+--
    '→ --------+-------+-----------------------------------------------------------------------------------------------------------------------------------
    '→
| Level   | Code | Message
    '→
    '→ |
+--
    '→ --------+-------+-----------------------------------------------------------------------------------------------------------------------------------
    '→
| Warning | 1105 | bad request to http://127.0.0.1:20180/config: fail to update, error: "config
    '→ log-level can not be changed" |
+--
    '→ --------+-------+-----------------------------------------------------------------------------------------------------------------------------------
    '→
1 row in set (0.00 sec)
```

批量修改配置不保证原子性，可能出现某些实例成功，而某些失败的情况。如使用 `set tikv key=val` 命令修改整个TiKV 集群配置时，可能有部分实例失败，请执行 `show warnings` 进行查看。

如遇到部分修改失败的情况，需要重新执行对应的修改语句，或通过修改单个实例的方式完成修改。如果因网络或者机器故障等原因无法访问到的TiKV，需要等到恢复后再次进行修改。

针对TiKV 可在线修改的配置项，如果成功修改后，修改的结果会被持久化到配置文件中，后续以配置文件中的配置为准。某些配置项名称可能和TiDB 预留关键字冲突，如 `limit`、`key` 等，对于此类配置项，需要用反引号 ` 包裹起来，如 `` `raftstore.raft-log-gc-size-limit` ``。

支持的配置项列表如下：

配置项	简介
log.level	日志等级

配置项	简介
raftstore.raft-max-inflight-msgs	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
raftstore.raft-log-gc-tick-interval	删除 Raft 日志的轮询任务调度间隔时间
raftstore.raft-log-gc-threshold	允许残余的 Raft 日志个数，软限制
raftstore.raft-log-gc-count-limit	允许残余的 Raft 日志个数，硬限制

配置项	简介
raftstore.raft-log-gc-size-limit	允许残余的 Raft 日志大小，硬限制
raftstore.raft-max-size-per-msg	允许生成的单个消息包的大小，软限制
raftstore.raft-entry-max-size	单个 Raft 日志最大大小，硬限制
raftstore.raft-entry-cache-life-time	内存中日志 cache 允许的最长残留时间
raftstore.split-region-check-tick-interval	检查 Region 是否需要分裂的时间间隔

配置项	简介
raftstore.region-split-check-diff	允许 Region 数据超过指定大小的最大值
raftstore.region-compact-check-interval	检查是否需要人工触发 RocksDB compaction 的时间间隔
raftstore.region-compact-check-step	每轮校验人工 compaction 时，一次性检查的 Region 个数
raftstore.region-compact-min-tombstone	触发 RocksDB compaction 需要的 tombstone 个数

配置项	简介
raftstore.region-compact-tombstone-percent	触发 RocksDB compaction 需要的 tombstone 所占比例
raftstore.pd-heartbeat-tick-interval	触发 Region 对 PD 心跳的时间间隔
raftstore.pd-store-heartbeat-tick-interval	触发 store 对 PD 心跳的时间间隔
raftstore.snap-mgr-gc-tick-interval	触发回收过期 snapshot 文件的时间间隔
raftstore.snap-gc-timeout	一个 snapshot 文件的最长保存时间

配置项	简介
raftstore.lock-cf-compact-interval	触发对 lock CF 进行 compact 检查的时间间隔
raftstore.lock-cf-compact-bytes-threshold	触发对 lock CF 进行 compact 的大小
raftstore.max-messages-per-tick	每轮处理的消息最大个数
raftstore.max-peer-down-duration	副本允许的最长未响应时间

配置项	简介
raftstore.max-leader-missing-duration	允许副本处于无主状态的最长时间，超过将向 PD 校验自己是否已经被删除
raftstore.abnormal-leader-missing-duration	允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中

配置项	简介
raftstore.peer-stale-state-check-interval	触发检验副本是否处于无主状态的时间间隔
raftstore.consistency-check-interval	触发一致性检查的时间间隔（不建议使用该配置项，因为与 TiDB GC 操作不兼容）
raftstore.raft-store-max-leader-lease	主可信任期的最长时间
raftstore.merge-check-tick-interval	触发 Merge 完成检查的时间间隔

配置项	简介
raftstore.cleanup-import-sst-interval	触发检查过期 SST 文件的时间间隔
raftstore.local-read-batch-size	轮处理读请求的最大个数
raftstore.apply-yield-write-size	Apply 线程每一轮处理单个状态机写入的最大数据量

配置项	简介
raftstore.hibernate-timeout	启动后进入静默状态前需要等待的最短时间，在该时间段内不会进入静默状态（未 release）
raftstore.apply-pool-size	处理把数据落盘至磁盘的线程池中线程的数量，即 Apply 线程池大小

配置项	简介
raftstore.store-pool-size	处理 Raft 的线程池中线程的数量，即 Raft-store 线程池的大小
raftstore.apply-max-batch-size	Rapfftly- 状态机由 Batch-System 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。

配置项	简介
raftstore.sRtoarfte-max-batch-size	状态机由 Batch-System 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。

配置项	简介
raftstore.sync-io-pool-size	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（不支持将该配置项由非零值调整为 0，或者从 0 调整为非零值）

配置项	简介
readpool.unified.max-thread-count	统一处理读请求的线程池最多的线程数量，即 UnifyReadPool 线程池大小
readpool.unified.auto-adjust-pool-size	是否开启自适应调整 UnifyReadPool 的大小
coprocessor.split-region-on-table	开启按 table 分裂 Region 的开关
coprocessor.batch-split-limit	批量分裂 Region 的阈值

配置项	简介
coprocessRoer.grieognion-max-size	容量空间的最大值
coprocessor.region-split-size	分裂后新 Region 的大小
coprocessRoer.grieognion-max-keys	最多允许的 key 的个数
coprocessor.region-split-keys	分裂后新 Region 的 key 的个数
pessimistic-txn.wait-for-lock-timeout	悲观事务遇到锁后的最长等待时间
pessimistic-txn.wake-up-delay-duration	悲观事务被重新唤醒的时间

配置项	简介
pessimistic-txn.pipelined	是否开启流水线式加悲观锁流程
pessimistic-txn.in-memory	是否开启内存悲观锁功能
quota.foreground-cpu-time	限制处理 TiKV 前台读写请求所使用的 CPU 资源使用量，软限制
quota.foreground-write-bandwidth	限制前台事务写入的带宽，软限制

配置项	简介
quota.foreground-read-bandwidth	限制前台事务读取数据和 Coprocessor 读取数据的带宽，软限制
quota.background-cpu-time	限制后台处理 TiKV 后台读写请求所使用的 CPU 资源使用量，软限制
quota.background-write-bandwidth	限制后台事务写入的带宽，软限制，暂未生效

配置项	简介
quota.background-read-bandwidth	后台事务读取数据和 Coprocessor 读取数据的带宽，软限制，暂未生效
quota.enable-auto-tune	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

配置项	简介
quota.max	单次读写请求被强制等待的最大时间
delay-duration	跳过 Region GC 的阈值（GC 版本个数/key 个数）
gc.ratio-threshold	一轮处理 key 的个数
gc.batch-keys	一秒可写入 RocksDB 的最大字节数
gc.max-write-bytes-per-sec	是否使用 compaction filter
gc.enable-compaction-filter	

配置项	简介
gc.compaction-filter-skip-version-check	是否跳过 compaction filter 的集群版本检查（未 release）
{db-name}.max-total-wal-size	WAL 总大小限制
{db-name}.max-background-jobs	RocksDB 后台线程个数
{db-name}.max-background-flushes	RocksDB flush 线程个数
{db-name}.max-open-files	RocksDB 可以打开的文件总数
{db-name}.compaction-readahead-size	Compaction 时候 readahead 的大小

配置项	简介
{db-name}.bytes-per-sync	异步同步的限速速率
{db-name}.wal_bytes-per-sync	WAL 同步的限速速率
{db-name}.writable-file-max-buffer-size	WritableFileWrite 所使用的最大的 buffer 大小
{db-name}.{cf-name}.block-cache-size	block cache 大小
{db-name}.{cf-name}.write-buffer-size	memtable 大小
{db-name}.{cf-name}.max-write-buffer-number	最大 memtable 个数
{db-name}.{cf-name}.max-bytes-for-level-base	base (xL-1) 最大字节数

配置项	简介
{db-name}.{cf-name}.target-file-size-base	目标文件大小
{db-name}.{cf-name}.level0-file-num-compaction-trigger	触发 L0 文件最大个数
{db-name}.{cf-name}.level0-slowdown-writes-trigger	触发 L0 文件最大个数
{db-name}.{cf-name}.level0-stop-writes-trigger	完全阻停写入的 L0 文件最大个数
{db-name}.{cf-name}.max-compaction-bytes	一次 compaction 最大写入字节数
{db-name}.{cf-name}.max-bytes-for-level-multiplier	每一层的默认放大倍数

配置项	简介
{db-name}.{cf-com-name}.dispabadeti-on	自动
auto-compactio	的开关
{db-name}.{cf-com-name}.sofpt-action	pending
pending-compactio bytes-limit	bytes 的软限制
{db-name}.{cf-com-name}.harpda-ction	pending
pending-compactio bytes-limit	bytes 的硬限制
{db-name}.{cf-blob name}.tita b-run-mode	处理文件 b- 的模式
server.grpgcR- memory-pool-quota	PC 可使用的内存大小限制
server.magxR- grpc-send-msg-len	PC 可发送的最大消息长度

配置项	简介
server.raft-msg-max-batch-size	单个 gRPC 消息可包含的最大 Raft 消息个数
server.simplify-metrics	精简监控采样数据的开关
server.snap-io-max-bytes-per-sec	处理 snapshot 时最大允许使用的磁盘带宽
server.concurrent-send-snap-limit	同时发送 snapshot 的最大个数
server.concurrent-recv-snap-limit	同时接受 snapshot 的最大个数

配置项	简介
storage.block-cache.capacity	共享 cache 的大小（自 v4.0.3 起支持）
storage.scheduler-worker-pool-size	线程池中线程的数量
backup.num-threads	backup 线程的数量（自 v4.0.3 起支持）

配置项	简介
split.qps-threshold	对 Region 执行 load-base-split 的阈值。如果连续 10s 内，某个 Region 的读请求的 QPS 超过 qps-threshold，则尝试切分该 Region

配置项	简介
split.byte-threshold	对 Region 执行 load-base-split 的阈值。如果连续 10s 内，某个 Region 的读请求的流量超过 byte-threshold，则尝试切分该 Region

配置项	简介
split.region-cpu-overload-threshold-ratio	对 Region 执行 load-base-split 的阈值。如果连续 10s 内，某个 Region 的 Unified Read Pool CPU 使用时间占比超过了 region-cpu-overload-threshold-ratio，则尝试切分该 Region（自 v6.2.0 起支持）

配置项	简介
split.split-balance-score	load-base-split 的控制参数，确保 Region 切分后左右访问尽量均匀，数值越小越均匀，但也可能导致无法切分
split.split-contained-score	load-base-split 的控制参数，数值越小，Region 切分后跨 Region 的访问越少

配置项	简介
cdc.min-ts-interval	定期推进 Resolved TS 的时间间隔
cdc.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
cdc.sink-memory-quota	缓存在内存中的 TiCDC 数据变更事件占用内存的上限
cdc.incremental-scan-speed-limit	增量扫描历史数据的速度上限

配置项	简介
cdc.increment-scan-concurrency	增量扫描历史数据任务的最大并发执行个数

上述前缀为 {db-name} 或 {db-name}.{cf-name} 的是 RocksDB 相关的配置项。db-name 的取值可为 rocksdb 或 raftdb。

- 当 db-name 为 rocksdb 时，cf-name 的可取值有：defaultcf、writecf、lockcf、raftcf；
- 当 db-name 为 raftdb 时，cf-name 的可取值有：defaultcf。

具体配置项的意义可参考TiKV 配置文件描述

5.10.1.3 在线修改 PD 配置

PD 暂不支持单个实例拥有独立配置。所有实例共享一份配置，可以通过下列方式修改 PD 的配置项：

```
set config pd `log.level`='info'
```

设置成功会返回 Query OK：

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

针对 PD 可在线修改的配置项，成功修改后则会持久化到 etcd 中，不会对配置文件进行持久化，后续以 etcd 中的配置为准。同上，若和TiDB 预留关键字冲突，需要用反引号` 包裹此类配置项，例如 `schedule.leader-
'→ schedule-limit`。

支持配置项列表如下：

配置项	简介
log.level	日志级别
cluster-version	集群的版本
schedule.max-merge-region-size	控制 Region Merge 的 size 上限（单位是 MiB）
schedule.max-merge-region-keys	控制 Region Merge 的 key 数量上限
schedule.patrol-region-interval	控制 replicaChecker 检查 Region 健康状态的运行频率
schedule.split-merge-interval	控制对同一个 Region 做 split 和 merge 操作的间隔
schedule.max-snapshot-count	控制单个 store 最多同时接收或发送的 snapshot 数量

配置项	简介
schedule.max-pending-peer-count	控制单个 store 的 pending peer 上限
schedule.max-store-down-time	PD 认为失联 store 无法恢复的时间
schedule.leader-schedule-policy	用于控制 leader 调度的策略
schedule.leader-schedule-limit	可以控制同时进行 leader 调度的任务个数
schedule.region-schedule-limit	可以控制同时进行 Region 调度的任务个数
schedule.replica-schedule-limit	可以控制同时进行 replica 调度的任务个数
schedule.merge-schedule-limit	控制同时进行的 Region Merge 调度的任务
schedule.hot-region-schedule-limit	可以控制同时进行的热点调度的任务个数
schedule.hot-region-cache-hits-threshold	用于设置 Region 被视为热点的阈值
schedule.high-space-ratio	用于设置 store 空间充裕的阈值
schedule.low-space-ratio	用于设置 store 空间不足的阈值
schedule.tolerant-size-ratio	控制 balance 缓冲区大小
schedule.enable-remove-down-replica	用于开启自动删除 DownReplica 的特性
schedule.enable-replace-offline-replica	用于开启迁移 OfflineReplica 的特性
schedule.enable-make-up-replica	用于开启补充副本的特性
schedule.enable-remove-extra-replica	用于开启删除多余副本的特性
schedule.enable-location-replacement	用于开启隔离级别检查
schedule.enable-cross-table-merge	用于开启跨表 Merge
schedule.enable-one-way-merge	用于开启单向 Merge（只允许和下一个相邻的 Region Merge）
replication.max-replicas	用于设置副本的数量
replication.location-labels	用于设置 TiKV 集群的拓扑信息
replication.enable-placement-rules	开启 Placement Rules
replication.strictly-match-label	开启 label 检查
pd-server.use-region-storage	开启独立的Region 存储
pd-server.max-gap-reset-ts	用于设置最大的重置timestamp 的间隔（BR）
pd-server.key-type	用于设置集群 key 的类型
pd-server.metric-storage	用于设置集群 metrics 的存储地址
pd-server.dashboard-address	用于设置 dashboard 的地址
replication-mode.replication-mode	备份的模式

具体配置项意义可参考 PD 配置文件描述。

5.10.1.4 在线修改TiDB 配置

在线修改TiDB 配置的方式和TiKV/PD 有所不同，你可以通过修改系统变量来实现。

下面例子展示了如何通过变量 `tidb_slow_log_threshold` 在线修改配置项 `slow-threshold`。

`slow-threshold` 默认值是 300 毫秒，可以通过设置系统变量 `tidb_slow_log_threshold` 将其修改为 200 毫秒：

```
set tidb_slow_log_threshold = 200;
```

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

```
select @@tidb_slow_log_threshold;
```

```
+----------------------------+
| @@tidb_slow_log_threshold |
+----------------------------+
| 200                        |
+----------------------------+
1 row in set (0.00 sec)
```

支持在线修改的配置项和相应的TiDB 系统变量如下：

配置项	对应变量	简介
log.enable-slow-log	tidb_enable_slow_log	慢日志的开关
log.slow-threshold	tidb_slow_log_threshold	慢日志阈值
log.expensive-threshold	tidb_expensive_query_time_threshold	expensive 查询阈值

5.10.1.5 在线修改TiFlash 配置

目前，你可以通过修改系统变量 `tidb_max_tiflash_threads` 来在线修改 TiFlash 配置项 `max_threads`。`tidb_max_tiflash_threads` 表示TiFlash 中 request 执行的最大并发度。

`tidb_max_tiflash_threads` 默认值是 `-1`，表示此系统变量无效，由 TiFlash 的配置文件决定max_threads。你可以通过设置系统变量 `tidb_max_tiflash_threads` 将其修改为 `10`：

```
set tidb_max_tiflash_threads = 10;
```

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

```
select @@tidb_max_tiflash_threads;
```

```
+-----------------------------+
| @@tidb_max_tiflash_threads |
+-----------------------------+
| 10                          |
+-----------------------------+
1 row in set (0.00 sec)
```

5.11 Online Unsafe Recovery 使用文档

> 警告：
>
> - 此功能为有损恢复，无法保证数据索引一致性和事务完整性，若有问题需要额外的工具或者步骤进行相应修复。
> - 该功能自 v6.1.0 版本开始引入。在TiDB v6.1 以下版本为实验特性，行为与本文描述有区别，不推荐使用。在其他版本使用该功能时，请参考相应版本文档。

当多数副本的永久性损坏造成部分数据不可读写时，可以使用 Online Unsafe Recovery 功能进行数据有损恢复，使 TiKV 正常提供服务。

5.11.1 功能说明

在TiDB 中，根据用户定义的多种副本规则，一份数据可能会同时存储在多个节点中，从而保证在单个或少数节点暂时离线或损坏时，读写数据不受任何影响。但是，当一个Region 的多数或全部副本在短时间内全部下线时，该Region 会处于暂不可用的状态，无法进行读写操作。

如果一段数据的多数副本发生了永久性损坏（如磁盘损坏）等问题，从而导致节点无法上线时，此段数据会一直保持暂不可用的状态。这时，如果用户希望集群恢复正常使用，在用户能够容忍数据回退或数据丢失的前提下，用户理论上可以通过手动移除不可用副本的方式，使Region 重新形成多数派，进而让上层业务可以写入和读取（可能是 stale 的，或者为空）这一段数据分片。

在这个情况下，当存有可容忍丢失的数据的部分节点受到永久性损坏时，用户可以通过使用 Online Unsafe Recovery，快速简单地进行有损恢复。使用 Online Unsafe Recovery 时，PD 会自动暂停调度（包括 split 和 merge），然后收集全部节点内的数据分片元信息，用 PD 的全局视角生成一份更实时、更完整的恢复计划后，将其计划下发给各个存活的节点，使各节点执行数据恢复任务。另外，下发恢复计划后，PD 还会定期查看恢复进度，并在必要时重新向各节点分发恢复计划。

5.11.2 适用场景

Online Unsafe Recovery 功能适用于以下场景：

- 部分节点受到永久性损坏，导致节点无法重启，造成业务端的部分数据不可读、不可写。
- 可以容忍数据丢失，希望受影响的数据恢复读写。

5.11.3 使用步骤

5.11.3.1 前提条件

在使用 Online Unsafe Recovery 功能进行数据有损恢复前，请确认以下事项：

- 离线节点导致部分数据确实不可用。
- 离线节点确实无法自动恢复或重启。

5.11.3.2 第 1 步：指定无法恢复的节点

使用 PD Control 执行`unsafe remove-failed-stores <store_id>[,<store_id>,...]` 命令，指定已确定无法恢复的所有TiKV 节点，并用逗号隔开，以触发自动恢复。

```
pd-ctl -u <pd_addr> unsafe remove-failed-stores <store_id1,store_id2,...>
```

命令输出 `Success` 表示向 PD 注册任务成功。但仅表示请求已被接受，并不代表恢复成功。恢复任务在后台进行，具体进度使用`show` 查看。命令输出 `Failed` 表示注册任务失败，可能的错误有：

- `unsafe recovery is running`：已经有正在进行的恢复任务

- `invalid input store x doesn't exist`：指定的 store ID 不存在
- `invalid input store x is up and connected`：指定的 store ID 仍然是健康的状态，不应该进行恢复

可通过 `--timeout <seconds>` 指定可允许执行恢复的最长时间。若未指定，默认为 5 分钟。当超时后，恢复中断报错。

若 PD 进行过灾难性恢复 `pd-recover` 操作，丢失了无法恢复的TiKV 节点的 store 信息，因此无法确定要传的 store ID 时，可指定 `--auto-detect` 参数允许传入一个空的 store ID 列表。在该模式下，所有未在 PD store 列表中的 store ID 均被认为无法恢复，进行移除。

> 注意：
>
> - 由于此命令需要收集来自所有 Peer 的信息，可能会造成 PD 短时间内有明显的内存使用量上涨（10 万个 Peer 预计使用约 500 MiB 内存）。
> - 若执行过程中 PD 发生重启，则恢复中断，需重新触发命令。
> - 一旦执行，所指定的节点将被设为Tombstone 状态，不再允许启动。
> - 执行过程中，所有调度以及 split/merge 都会被暂停，待恢复成功或失败后自动恢复。

5.11.3.3 第 2 步：查看进度等待结束

节点移除命令运行成功后，使用 PD Control 执行`unsafe remove-failed-stores show` 命令，查看移除进度。

```
pd-ctl -u <pd_addr> unsafe remove-failed-stores show
```

恢复过程有多个可能的阶段：

- `collect report`：初始阶段，第一次接收 TiKV 的报告获得的全局信息。
- `tombstone tiflash learner`：在不健康的 Region 中，删除比其他健康 Peer 要新的TiFlash learner，防止极端情况造成panic。
- `force leader for commit merge`：特殊阶段。在有未完成的 commit merge 时出现，优先对有 commit merge 的 Region 进行 `force leader`，防止极端情况。
- `force leader`：强制不健康的 Region 在剩余的健康 Peer 中指定一个成为Raft leader。
- `demote failed voter`：将 Region 不健康的Voter 降级为 Learner，之后 Region 就可以正常地选出Raft leader。
- `create empty region`：创建一个空 Region 来补足 key range 的空洞，主要针对的是某些Region 的所有副本所在的 Store 都损坏了。

每一阶段按照JSON 格式输出，包括信息，时间，以及具体的恢复计划。例如：

```
[
    {
        "info": "Unsafe recovery enters collect report stage",
        "time": ".....",
        "details" : [
            "failed stores 4, 5, 6",
```

```
        ]
    },
    {
        "info": "Unsafe recovery enters force leader stage",
        "time": ".....",
        "actions": {
            "store 1": [
                "force leader on regions: 1001, 1002"
            ],
            "store 2": [
                "force leader on regions: 1003"
            ]
        }
    },
    {
        "info": "Unsafe recovery enters demote failed voter stage",
        "time": ".....",
        "actions": {
            "store 1": [
                "region 1001 demotes peers { id:101 store_id:4 }, { id:102 store_id:5 }",
                "region 1002 demotes peers { id:103 store_id:5 }, { id:104 store_id:6 }",
            ],
            "store 2": [
                "region 1003 demotes peers { id:105 store_id:4 }, { id:106 store_id:6 }",
            ]
        }
    },
    {
        "info": "Collecting reports from alive stores(1/3)",
        "time": ".....",
        "details": [
            "Stores that have not dispatched plan: ",
            "Stores that have reported to PD: 4",
            "Stores that have not reported to PD: 5, 6",
        ]
    }
]
```

PD 下发恢复计划后，会等待TiKV 上报执行的结果。如上述输出中最后一阶段的 `Collecting reports from alive stores` 显示 PD 下发恢复计划和接受TiKV 报告的具体状态。

整个恢复过程包括多个阶段，可能存在某一阶段的多次重试。一般情况下，预计时间为 3~10 个 store heartbeat 周期（一个 store heartbeat 默认为 10s）。当恢复完成后，命令执行结果最后一阶段显示 `"Unsafe recovery finished"`，以及受影响的 Region 所属的 table id（若无或使用 RawKV 则不显示）和受影响的 SQL 元数据 Region。如：

```
{
    "info": "Unsafe recovery finished",
    "time": ".....",
    "details": [
        "Affected table ids: 64, 27",
        "Affected meta regions: 1001",
    ]
}
```

得到受影响的table id 后，可以使用 INFORMATION_SCHEMA.TABLES 来查看受影响的表名。

```
SELECT TABLE_SCHEMA, TABLE_NAME, TIDB_TABLE_ID FROM INFORMATION_SCHEMA.TABLES WHERE TIDB_TABLE_ID
    ↪  IN (64, 27);
```

> 注意：
>
> - 恢复操作把一些 failed Voter 变成了 failed Learner，之后还需要 PD 调度经过一些时间将这些 failed Learner 移除。
> - 建议及时添加新的节点。

若执行过程中发生错误，最后一阶段会显示 "Unsafe recovery failed" 以及具体错误。如：

```
{
    "info": "Unsafe recovery failed: <error>",
    "time": "....."
}
```

5.11.3.4 第 3 步：检查数据索引一致性（RawKV 不需要）

> 注意：
> 数据可以读写并不代表没有数据丢失。

执行完成后，数据和索引可能会不一致。请使用 ADMIN CHECK 对受影响的表进行数据索引的一致性检查。

```
ADMIN CHECK TABLE table_name;
```

若结果有不一致的索引，可以通过重命名旧索引、创建新索引，然后再删除旧索引的步骤来修复数据索引不一致的问题。

1. 重命名旧索引：

```
ALTER TABLE table_name RENAME INDEX index_name TO index_name_lame_duck;
```

2. 创建新索引：

```
ALTER TABLE table_name ADD INDEX index_name (column_name);
```

3. 删除旧索引：

```
ALTER TABLE table_name DROP INDEX index_name_lame_duck;
```

5.11.3.5 第 4 步：移除无法恢复的节点（可选）

1. 缩容无法恢复的节点：

```
tiup cluster scale-in <cluster-name> -N <host> --force
```

2. 清理Tombstone 节点：

```
tiup cluster prune <cluster-name>
```

1. 删除该 PersistentVolumeClaim。

```
kubectl delete -n ${namespace} pvc ${pvc_name} --wait=false
```

2. 删除TiKV Pod，并等待新创建的 TiKV Pod 加入集群。

```
kubectl delete -n ${namespace} pod ${pod_name}
```

5.12 搭建双集群主从复制

本文档介绍如何配置一个TiDB 集群以及该集群的TiDB 或 MySQL 从集群，并将增量数据实时从主集群同步到从集群，主要包含以下内容：

1. 配置一个TiDB 集群以及该集群的TiDB 或 MySQL 从集群。
2. 将增量数据实时从主集群同步到从集群。
3. 在主集群发生灾难利用Redo log 恢复一致性数据。

如果你需要配置一个运行中的TiDB 集群和其从集群，以进行实时增量数据同步，可使用Backup & Restore (BR)和TiCDC。

5.12.1 第 1 步：搭建环境

1. 部署集群。

   使用 tiup playground 快速部署 TiDB 上下游测试集群。生产环境可以参考 tiup 官方文档根据业务需求来部署集群。

   为了方便展示和理解，我们简化部署结构，需要准备以下两台机器，分别来部署上游主集群和下游从集群。假设 IP 地址分别为:

   - NodeA: 172.16.6.123，部署上游 TiDB
   - NodeB: 172.16.6.124，部署下游 TiDB

```
# 在 NodeA 上创建上游集群
tiup --tag upstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
# 在 NodeB 上创建下游集群
tiup --tag downstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 0
# 查看集群状态
tiup status
```

2. 初始化数据。

   测试集群中默认创建了 test 数据库，因此可以使用 sysbench 工具生成测试数据，用以模拟真实集群中的历史数据。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=10 --table-size=10000 prepare
```

   这里通过 sysbench 运行 oltp_write_only 脚本，其将在测试数据库中生成 10 张表，每张表包含 10000 行初始数据。tidb-config 的配置如下：

```
mysql-host=172.16.6.122 # 这里需要替换为实际上游集群 ip
mysql-port=4000
mysql-user=root
mysql-password=
db-driver=mysql          # 设置数据库驱动为 mysql
mysql-db=test            # 设置测试数据库为 test
report-interval=10       # 设置定期统计的时间间隔为 10 秒
threads=10               # 设置 worker 线程数量为 10
time=0                   # 设置脚本总执行时间，0 表示不限制
rate=100                 # 设置平均事务速率 tps = 100
```

3. 模拟业务负载。

   实际生产集群的数据迁移过程中，通常原集群还会写入新的业务数据，本文中可以通过 sysbench 工具模拟持续的写入负载，下面的命令会使用 10 个 worker 在数据库中的 sbtest1、sbtest2 和 sbtest3 三张表中持续写入数据，其总tps 限制为 100。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=3 run
```

4. 准备外部存储。

   在全量数据备份中，上下游集群均需访问备份文件，因此推荐使用外部存储存储备份文件，本文中通过 Minio 模拟兼容 S3 的存储服务：

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
# 配置访问 minio 的 access-key access-secret-id
export `HOST_IP`='172.16.6.123' # 替换为实际部署 minio 的机器 ip
export** **MINIO_ROOT_USER**='**minio'
export MINIO_ROOT_PASSWORD='miniostorage'
# 创建 redo 和 backup 数据目录， 其中 redo, backup 为 bucket 名字
mkdir -p data/redo
mkdir -p data/backup
# 启动 minio, 暴露端口在 6060
nohup ./minio server ./data --address :6060 &
```

   上述命令行启动了一个单节点的minio server 模拟 S3 服务，其相关参数为：

   - Endpoint : http://${HOST_IP}:6060/
   - Access-key : minio
   - Secret-access-key: miniostorage
   - Bucket: redo

   其访问链接为如下:

```
s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=http://${HOST_IP}:6060&
    ↪ force-path-style=true
```

5.12.2 第 2 步：迁移全量数据

搭建好测试环境后，可以使用 BR 工具的备份和恢复功能迁移全量数据。BR 工具有多种使用方式，本文中使用 SQL 语句 BACKUP 和 RESTORE 进行备份恢复。

> 注意：
>
> - 在生产集群中，关闭GC 机制和备份操作会一定程度上降低集群的读性能，建议在业务低峰期进行备份，并设置合适的 RATE_LIMIT 限制备份操作对线上业务的影响。
> - 上下游集群版本不一致时，应检查 BR 工具的兼容性。本文假设上下游集群版本相同。

1. 关闭GC。

   为了保证增量迁移过程中新写入的数据不丢失，在开始备份之前，需要关闭上游集群的垃圾回收(GC)机制，以确保系统不再清理历史数据。

   执行如下命令关闭GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=FALSE;
```

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

查询 tidb_gc_enable 的取值，判断GC 是否已关闭：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------------------+
| @@global.tidb_gc_enable |
+-------------------------------------+
|                       0 |
+-------------------------------------+
1 row in set (0.00 sec)
```

2. 备份数据。

   在上游集群中执行 BACKUP 语句备份数据：

```
MySQL [(none)]> BACKUP DATABASE * TO ``s3://backup?access-key=minio&secret-access-key=
    '→ miniostorage&endpoint=http://${HOST_IP}:6060&force-path-style=true`` RATE_LIMIT =
    '→ 120 MB/SECOND;
```

```
+----------------------+----------+--------------------+--------------------+-------------------------------------- +
    '→
| Destination          | Size     | BackupTS           | Queue Time         | Execution
    '→ Time       |
+----------------------+----------+--------------------+--------------------+-------------------------------------- +
    '→
| local:///tmp/backup/ | 10315858 | 431434047157698561 | 2022-02-25 19:57:59 | 2022-02-25
    '→ 19:57:59 |
+----------------------+----------+--------------------+--------------------+-------------------------------------- +
    '→
1 row in set (2.11 sec)
```

   备份语句提交成功后，TiDB 会返回关于备份数据的元信息，这里需要重点关注 BackupTS，它意味着该时间点之前数据会被备份，后边的教程中，本文将使用 BackupTS 作为数据校验截止时间和TiCDC 增量扫描的开始时间。

3. 恢复数据。

   在下游集群中执行RESTORE 语句恢复数据：

```
mysql> RESTORE DATABASE * FROM ``s3://backup?access-key=minio&secret-access-key=miniostorage
    '→ &endpoint=http://${HOST_IP}:6060&force-path-style=true``;
```

```
+---------------------+----------+--------------------+--------------------+---------------------------------------- +
    ↪
| Destination           | Size     | BackupTS           | Queue Time           | Execution
    ↪ Time       |
+---------------------+----------+--------------------+--------------------+---------------------------------------- +
    ↪
| local:///tmp/backup/ | 10315858 | 431434141450371074 | 2022-02-25 20:03:59 | 2022-02-25
    ↪ 20:03:59 |
+---------------------+----------+--------------------+--------------------+---------------------------------------- +
    ↪
1 row in set (41.85 sec)
```

4. （可选）校验数据。

   通过sync-diff-inspector 工具，可以验证上下游数据在某个时间点的一致性。从上述备份和恢复命令的输出可以看到，上游集群备份的时间点为 431434047157698561，下游集群完成数据恢复的时间点为 431434141450371074。

```
sync_diff_inspector -C ./config.yaml
```

```
# Diff Configuration.
######################### Global config #########################
check-thread-count = 4
export-fix-sql = true
check-struct-only = false

######################### Datasource config #########################
[data-sources]
[data-sources.upstream]
    host = "172.16.6.123" # 替换为实际上游集群 ip
    port = 4000
    user = "root"
    password = ""
    snapshot = "431434047157698561" # 配置为实际的备份时间点
[data-sources.downstream]
    host = "172.16.6.124" # 替换为实际下游集群 ip
    port = 4000
    user = "root"
    password = ""
    snapshot = "431434141450371074" # 配置为实际的恢复时间点

######################### Task config #########################
[task]
    output-dir = "./output"
```

   关于 sync-diff-inspector 的配置方法，请参考配置文件说明。在本文中，相应的配置如下：

```
    source-instances = ["upstream"]
    target-instance = "downstream"
    target-check-tables = ["*.*"]
```

5.12.3 第 3 步：迁移增量数据

1. 部署TiCDC。

   完成全量数据迁移后，就可以部署并配置TiCDC 集群同步增量数据，实际生产集群中请参考TiCDC 部署。本文在创建测试集群时，已经启动了一个TiCDC 节点，因此可以直接进行changefeed 的配置。

2. 创建同步任务。

   创建changefeed 配置文件并保存为 changefeed.toml。

```
[consistent]
# 一致性级别，配置成 eventual 表示开启一致性复制
level = "eventual"
# 使用 S3 来存储 redo log，其他可选为 local，nfs
storage = "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint=http
    ↪ ://172.16.6.125:6060&force-path-style=true"
```

   在上游集群中，执行以下命令创建从上游到下游集群的同步链路：

```
tiup cdc cli changefeed create --server=http://172.16.6.122:8300 --sink-uri="mysql://root:
    ↪ @172.16.6.125:4000" --changefeed-id="primary-to-secondary" --start-ts
    ↪ ="431434047157698561"
```

   以上命令中：

   - --server：TiCDC 集群任意一节点的地址
   - --sink-uri：同步任务下游的地址
   - --start-ts：TiCDC 同步的起点，需要设置为实际的备份时间点（也就是第 2 步：迁移全量数据提到的 BackupTS）

   更多关于changefeed 的配置，请参考TiCDC Changefeed 配置参数。

3. 重新开启GC。

   TiCDC 可以保证未同步的历史数据不会被回收。因此，创建完从上游到下游集群的changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。详情请参考TiCDC GC safepoint 的完整行为。

执行如下命令打开GC：

```
```sql
MySQL [test]> SET GLOBAL tidb_gc_enable=TRUE;
```


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

```
```

查询 `tidb_gc_enable` 的取值，判断 GC 是否已开启：

```sql
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------------------+
| @@global.tidb_gc_enable |
+-------------------------------------+
|                       1 |
+-------------------------------------+
1 row in set (0.00 sec)
```

5.12.4 第 4 步：模拟主集群故障

模拟在业务过程中上游TiDB 发生灾难性故障无法再启动起来，这里可以直接使用 Ctrl + C 终止tiup playground 进程。

5.12.5 第 5 步：使用redo log 确保数据一致性

在正常同步过程中，为了提高TiCDC 的吞吐能力，TiCDC 会将事务并行写入下游。因此，当TiCDC 同步链路意外中断时，下游可能不会恰好停在与上游一致的状态。我们这里需要使用TiCDC 的命令行工具来向下游重放 redo log，使下游达到最终一致性状态。

```
tiup cdc redo apply --storage "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint
    ↪ =http://172.16.6.123:6060&force-path-style=true" --tmp-dir /tmp/redo --sink-uri "mysql://
    ↪ root:@172.16.6.124:4000"
```

- `--storage`：指定 redo log 所在的 S3 位置以及credential
- `--tmp-dir`：为从 S3 下载redo log 的缓存目录
- `--sink-uri`：指定下游集群的地址

5.12.6 第 6 步：恢复主集群及业务

现在从集群有了某一时刻全部的一致性数据，你需要重新搭建主从集群来保证数据可靠性。

1. 在 NodeA 重新搭建一个新的TiDB 集群作为新的主集群。

```
tiup --tag upstream playground v5.4.0 --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --
    ↪ ticdc 1
```

2. 使用 BR 将从集群数据全量备份恢复到主集群。

```
# 全量备份从集群的数据
tiup br --pd http://172.16.6.124:2379 backup full --storage ./backup
# 全量恢复从集群的数据
tiup br --pd http://172.16.6.123:2379 restore full --storage ./backup
```

3. 创建一个TiCDC 同步任务，备份主集群数据到从集群。

```
# 创建 changefeed
tiup cdc cli changefeed create --server=http://172.16.6.122:8300 --sink-uri="mysql://root:
    ↪ @172.16.6.125:4000" --changefeed-id="primary-to-secondary"
```

6 监控与告警

6.1 TiDB 监控框架概述

TiDB 使用开源时序数据库 Prometheus 作为监控和性能指标信息存储方案，使用Grafana 作为可视化组件进行展示。

6.1.1 Prometheus 在 TiDB 中的应用

Prometheus 是一个拥有多维度数据模型的、灵活的查询语句的时序数据库。Prometheus 作为热门的开源项目，拥有活跃的社区及众多的成功案例。

Prometheus 提供了多个组件供用户使用。目前，TiDB 使用了以下组件：

- Prometheus Server：用于收集和存储时间序列数据。
- Client 代码库：用于定制程序中需要的 Metric。
- Alertmanager：用于实现报警机制。

其结构如下图所示：

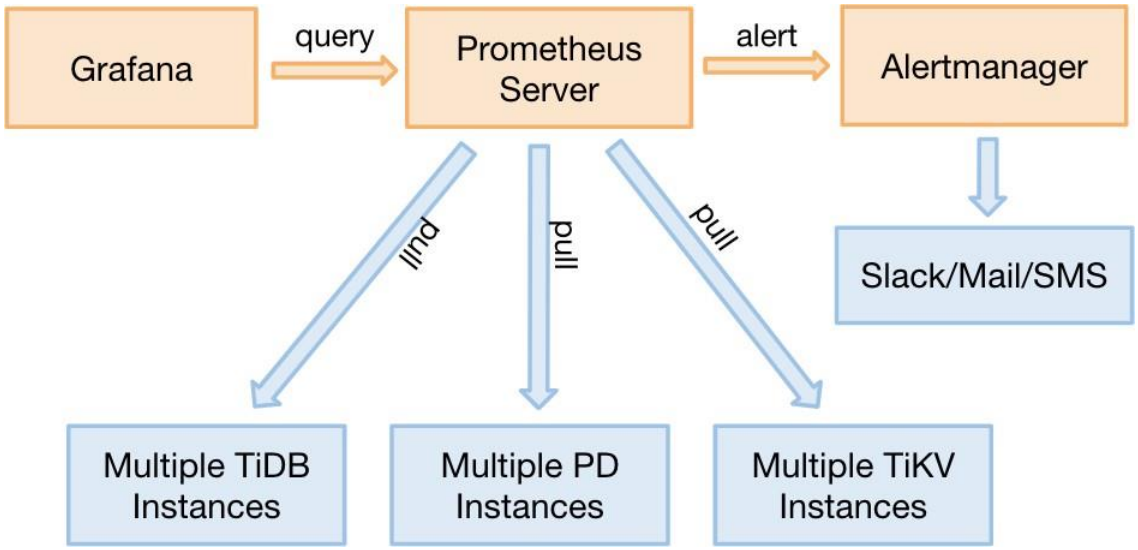


图 44: Prometheus in TiDB

6.1.2 Grafana 在 TiDB 中的应用

Grafana 是一个开源的metric 分析及可视化系统。TiDB 使用 Grafana 来展示 TiDB 集群各组件的相关监控，监控项分组如下图所示：

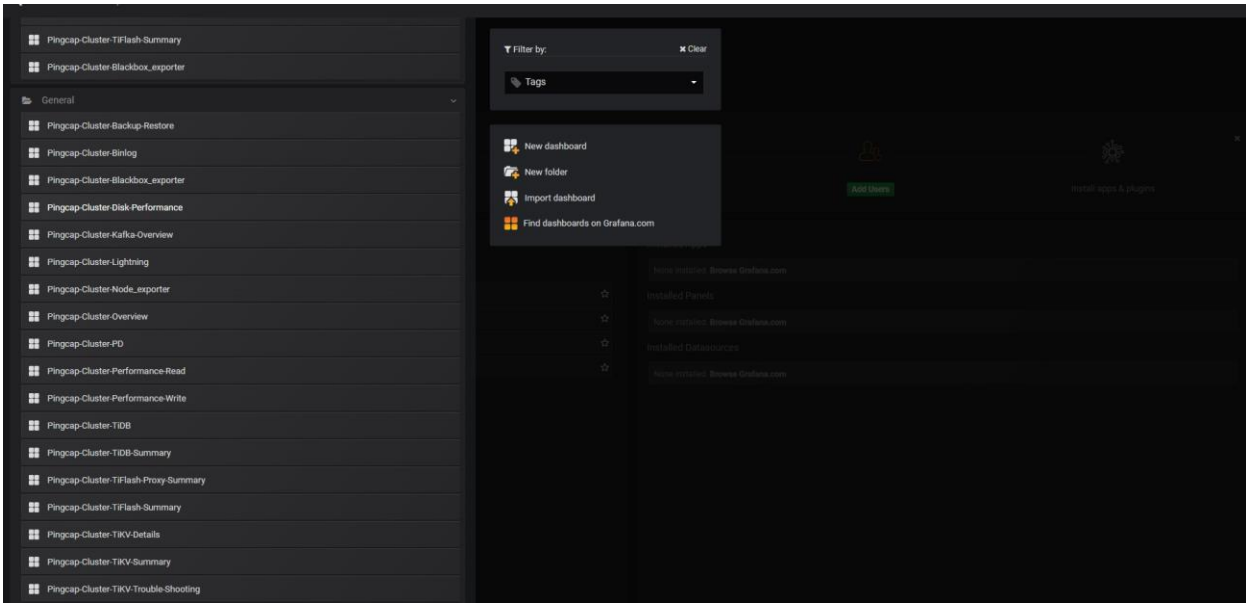


图 45: Grafana monitored_groups

- {TiDB_Cluster_name}-Backup-Restore：备份恢复相关的监控项。

- {TiDB_Cluster_name}-Binlog：TiDB Binlog 相关的监控项。
- {TiDB_Cluster_name}-Blackbox_exporter：网络探活相关监控项。
- {TiDB_Cluster_name}-Disk-Performance：磁盘性能相关监控项。
- {TiDB_Cluster_name}-Kafka-Overview：Kafka 相关监控项。
- {TiDB_Cluster_name}-Lightning：TiDB Lightning 组件相关监控项。
- {TiDB_Cluster_name}-Node_exporter：操作系统相关监控项。
- {TiDB_Cluster_name}-Overview：重要组件监控概览。
- {TiDB_Cluster_name}-PD：PD server 组件相关监控项。
- {TiDB_Cluster_name}-Performance-Read：读性能相关监控项。
- {TiDB_Cluster_name}-Performance-Write：写性能相关监控项。
- {TiDB_Cluster_name}-TiDB：TiDB server 组件详细监控项。
- {TiDB_Cluster_name}-TiDB-Summary：TiDB server 相关监控项概览。
- {TiDB_Cluster_name}-TiFlash-Proxy-Summary：数据同步到 TiFlash 的代理 server 监控项概览。
- {TiDB_Cluster_name}-TiFlash-Summary：TiFlash server 相关监控项概览。
- {TiDB_Cluster_name}-TiKV-Details：TiKV server 组件详细监控项。
- {TiDB_Cluster_name}-TiKV-Summary：TiKV server 监控项概览。
- {TiDB_Cluster_name}-TiKV-Trouble-Shooting：TiKV 错误诊断相关监控项。
- {TiDB_Cluster_name}-TiCDC：TiCDC 组件详细监控项。

每个分组包含多个监控项页签，页签中包含多个详细的监控项信息。以 Overview 监控组为例，其中包含 5 个页签，每个页签内有相应的监控指标看板，如下图所示：



图 46: Grafana Overview

要快速了解TiDB 监控与报警系统的体系、该系统背后的数据流转方式、系统管理方法、系统使用方法和常用监控指标，建议观看下面的培训视频（时长 29 分钟）。注意本视频只作为学习参考，具体的监控指标与相关报警规则，请以文档内容为准。

6.2 TiDB 集群监控API

TiDB 提供了以下几种接口来监控集群状态：

- 状态接口：通过 HTTP 接口对外汇报组件的信息。通过状态接口，你可获取当前 TiDB Server 的运行状态以及表的存储信息。
- Metrics 接口：使用 Prometheus 记录组件中各种操作的详细信息，使用Grafana 进行可视化展示。

6.2.1 使用状态接口

状态接口用于监控组件的一些基本信息，并且可以作为 keepalive 的监测接口。另外，通过 PD 的状态接口可以看到整个TiKV 集群的详细信息。

6.2.1.1 TiDB Server

- TiDB API 地址：`http://${host}:${port}`
- 默认端口：10080

6.2.1.1.1 运行状态

以下示例中，通过访问 `http://${host}:${port}/status` 获取当前TiDB Server 的状态，并判断该TiDB Server 是否存活。结果以JSON 格式返回：

```
curl http://127.0.0.1:10080/status
```

```
{
    connections: 0, # 当前 TiDB Server 上的客户端连接数
    version: "5.7.25-TiDB-v4.0.0-rc-141-g7267747ae", # TiDB 版本号
    git_hash: "7267747ae0ec624dffc3fdedb00f1ed36e10284b" # TiDB 当前代码的 Git Hash
}
```

6.2.1.1.2 存储信息

以下示例中，通过访问 `http://${host}:${port}/schema_storage/${db}/${table}` 获取指定数据表的存储信息。结果以JSON 格式返回：

```
curl http://127.0.0.1:10080/schema_storage/mysql/stats_histograms
```

```
{
    "table_schema": "mysql",
    "table_name": "stats_histograms",
    "table_rows": 0,
    "avg_row_length": 0,
    "data_length": 0,
    "max_data_length": 0,
    "index_length": 0,
    "data_free": 0
}
```

```
curl http://127.0.0.1:10080/schema_storage/test
```

```
[
    {
        "table_schema": "test",
        "table_name": "test",
        "table_rows": 0,
        "avg_row_length": 0,
        "data_length": 0,
        "max_data_length": 0,
```

```
        "index_length": 0,
        "data_free": 0
    }
]
```

6.2.1.2 PD Server

- PD API 地址：`http://${host}:${port}/pd/api/v1/${api_name}`
- 默认端口：2379
- 各类 `api_name` 详细信息：参见 PD API Doc

通过该接口可以获取当前所有TiKV 节点的状态以及负载均衡信息。下面以一个单节点的TiKV 集群为例，说明用户需要了解的信息：

```
curl http://127.0.0.1:2379/pd/api/v1/stores
```

```
{
  "count": 1,  # TiKV 节点数量
  "stores": [ # TiKV 节点的列表
    # 集群中单个 TiKV 节点的信息
    {
      "store": {
        "id": 1,
        "address": "127.0.0.1:20160",
        "version": "4.0.0-rc.2",
        "status_address": "172.16.5.90:20382",
        "git_hash": "2fdb2804bf8ffaab4b18c4996970e19906296497",
        "start_timestamp": 1590029618,
        "deploy_path": "/data2/tidb_test/v4.0.rc.2/tikv-20372/bin",
        "last_heartbeat": 1590030038949235439,
        "state_name": "Up"
      },
      "status": {
        "capacity": "3.581TiB", # 存储总容量
        "available": "3.552TiB", # 存储剩余容量
        "used_size": "31.77MiB",
        "leader_count": 174,
        "leader_weight": 1,
        "leader_score": 174,
        "leader_size": 174,
        "region_count": 531,
        "region_weight": 1,
        "region_score": 531,
        "region_size": 531,
        "start_ts": "2020-05-21T10:53:38+08:00",
```

```
        "last_heartbeat_ts": "2020-05-21T11:00:38.949235439+08:00",
        "uptime": "7m0.949235439s"
      }
    }
]
```

6.2.2 使用 metrics 接口

Metrics 接口用于监控整个集群的状态和性能。

- 如果使用其他方式部署TiDB 集群，在使用metrics 接口前，需先部署 Prometheus 和 Grafana。

成功部署 Prometheus 和 Grafana 之后，配置 Grafana。

6.3 TiDB 集群监控部署

本文档适用于希望手动部署TiDB 监控报警系统的用户。TiUP 部署方式，会同时自动部署监控报警系统，无需手动部署。

6.3.1 部署 Prometheus 和 Grafana

假设TiDB 的拓扑结构如下：

节点	主机IP	服务
Node1	192.168.199.113	PD1, TiDB, node_export, Prometheus, Grafana
Node2	192.168.199.114	PD2, node_export
Node3	192.168.199.115	PD3, node_export
Node4	192.168.199.116	TiKV1, node_export
Node5	192.168.199.117	TiKV2, node_export
Node6	192.168.199.118	TiKV3, node_export

6.3.1.1 第 1 步：下载二进制包

下载二进制包：

```
wget https://download.pingcap.org/prometheus-2.27.1.linux-amd64.tar.gz
wget https://download.pingcap.org/node_exporter-v1.3.1-linux-amd64.tar.gz
wget https://download.pingcap.org/grafana-7.5.11.linux-amd64.tar.gz
```

解压二进制包：

```
tar -xzf prometheus-2.27.1.linux-amd64.tar.gz
tar -xzf node_exporter-v1.3.1-linux-amd64.tar.gz
tar -xzf grafana-7.5.11.linux-amd64.tar.gz
```

6.3.1.2 第 2 步：在 Node1，Node2，Node3，Node4 上启动 node_exporter

```
cd node_exporter-v1.3.1-linux-amd64
```

启动 node_exporter 服务：

```
./node_exporter --web.listen-address=":9100" \
    --log.level="info" &
```

6.3.1.3 第 3 步：在 Node1 上启动 Prometheus

编辑 Prometheus 的配置文件：

```
cd prometheus-2.27.1.linux-amd64 &&
vi prometheus.yml
```

```
...

global:
  scrape_interval:     15s
  evaluation_interval: 15s
  # scrape_timeout 设置为全局默认值 (10s)
  external_labels:
    cluster: 'test-cluster'
    monitor: "prometheus"

scrape_configs:
  - job_name: 'overwritten-nodes'
    honor_labels: true # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:9100'
      - '192.168.199.114:9100'
      - '192.168.199.115:9100'
      - '192.168.199.116:9100'
      - '192.168.199.117:9100'
      - '192.168.199.118:9100'

  - job_name: 'tidb'
    honor_labels: true # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:10080'

  - job_name: 'pd'
    honor_labels: true # 不要覆盖 job 和实例的 label
```

```
    static_configs:
    - targets:
      - '192.168.199.113:2379'
      - '192.168.199.114:2379'
      - '192.168.199.115:2379'

  - job_name: 'tikv'
    honor_labels: true # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.116:20180'
      - '192.168.199.117:20180'
      - '192.168.199.118:20180'

...
```

启动 Prometheus 服务：

```
./prometheus \
    --config.file="./prometheus.yml" \
    --web.listen-address=":9090" \
    --web.external-url="http://192.168.199.113:9090/" \
    --web.enable-admin-api \
    --log.level="info" \
    --storage.tsdb.path="./data.metrics" \
    --storage.tsdb.retention="15d" &
```

6.3.1.4 第 4 步：在 Node1 上启动Grafana

编辑Grafana 的配置文件：

```
cd grafana-7.5.11 &&
vi conf/grafana.ini
```

```
...

[paths]
data = ./data
logs = ./data/log
plugins = ./data/plugins
[server]
http_port = 3000
domain = 192.168.199.113
[database]
[session]
[analytics]
```

```
check_for_updates = true
[security]
admin_user = admin
admin_password = admin
[snapshots]
[users]
[auth.anonymous]
[auth.basic]
[auth.ldap]
[smtp]
[emails]
[log]
mode = file
[log.console]
[log.file]
level = info
format = text
[log.syslog]
[event_publisher]
[dashboards.json]
enabled = false
path = ./data/dashboards
[metrics]
[grafana_net]
url = https://grafana.net

...
```

启动Grafana服务：

```
./bin/grafana-server \
    --config="./conf/grafana.ini" &
```

6.3.2 配置Grafana

本小节介绍如何配置Grafana。

6.3.2.1 第1步：添加 Prometheus 数据源

1. 登录Grafana界面。

    - 默认地址：http://localhost:3000
    - 默认账户：admin
    - 默认密码：admin

> **注意：**
>
> Change Password 步骤可以选择 Skip。

2. 点击 Grafana 侧边栏菜单 Configuration 中的 Data Source。
3. 点击 Add data source。
4. 指定数据源的相关信息：
    - 在 Name 处，为数据源指定一个名称。
    - 在 Type 处，选择 Prometheus。
    - 在 URL 处，指定 Prometheus 的 IP 地址。
    - 根据需求指定其它字段。
5. 点击 Add 保存新的数据源。

6.3.2.2 第 2 步：导入 Grafana 面板

执行以下步骤，为 PD Server、 TiKV Server 和 TiDB Server 分别导入 Grafana 面板：

1. 点击侧边栏的 Grafana 图标。
2. 在侧边栏菜单中，依次点击 Dashboards > Import 打开 Import Dashboard 窗口。
3. 点击 Upload .json File 上传对应的 JSON 文件（从 pingcap/tidb、 tikv/tikv 和 tikv/pd 下载 TiDB Grafana 配置文件）。

> **注意：**
>
> TiKV、 PD 和 TiDB 面板对应的 JSON 文件分别为 `tikv_summary.json`，`tikv_details.json`，`tikv_trouble_shooting.json`，`pd.json`，`tidb.json`，`tidb_summary.json`。

4. 点击 Load。
5. 选择一个 Prometheus 数据源。
6. 点击 Import，Prometheus 面板即导入成功。

6.3.3 查看组件 metrics

在顶部菜单中，点击 New dashboard，选择要查看的面板。

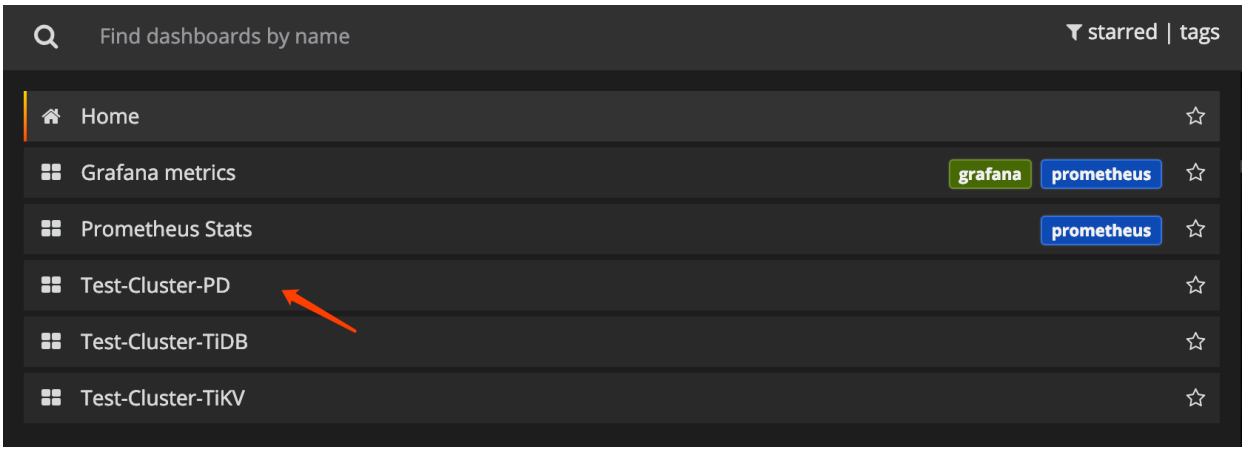


图 47: view dashboard

可查看以下集群组件信息：

- TiDB Server:
  - query 处理时间，可以看到延迟和吞吐
  - ddl 过程监控
  - TiKV client 相关的监控
  - PD client 相关的监控
- PD Server:
  - 命令执行的总次数
  - 某个命令执行失败的总次数
  - 某个命令执行成功的耗时统计
  - 某个命令执行失败的耗时统计
  - 某个命令执行完成并返回结果的耗时统计
- TiKV Server:
  - GC 监控
  - 执行 KV 命令的总次数
  - Scheduler 执行命令的耗时统计
  - Raft propose 命令的总次数
  - Raft 执行命令的耗时统计
  - Raft 执行命令失败的总次数
  - Raft 处理 ready 状态的总次数

> 警告：
>
> - 从TiDB v6.0.0 起，PingCAP 不再维护 MetricsTool 工具。从 v6.1.0 起，不再维护 MetricsTool 的文档。

- 如需导出监控数据，建议使用PingCAP Clinic 诊断服务一键导出诊断TiDB 集群时所需要的信息，包括监控数据、日志、集群拓扑、配置、参数等。

6.4 将 Grafana 监控数据导出成快照

在故障诊断中，监控数据十分重要。当你请求远程协助时，技术支持人员有时需要查看Grafana Dashboard 以确认问题所在。MetricsTool 用于将Grafana Dashboard 的快照导出为本地文件，并将快照可视化。因此，你可以在不泄露Grafana 服务器上其他敏感信息的前提下，将监控数据以快照形式分享给外部人员，同时也方便外部人员准确识读数据图表。

6.4.1 使用方法

可以通过访问 <https://metricstool.pingcap.net> 来使用 MetricsTool。它主要提供以下三种功能：

- 导出快照：提供一段在浏览器开发者工具上运行的用户脚本。你可以使用这个脚本在任意Grafana v6.x.x 服务器上下载当前 Dashboard 中所有可见面板的快照。

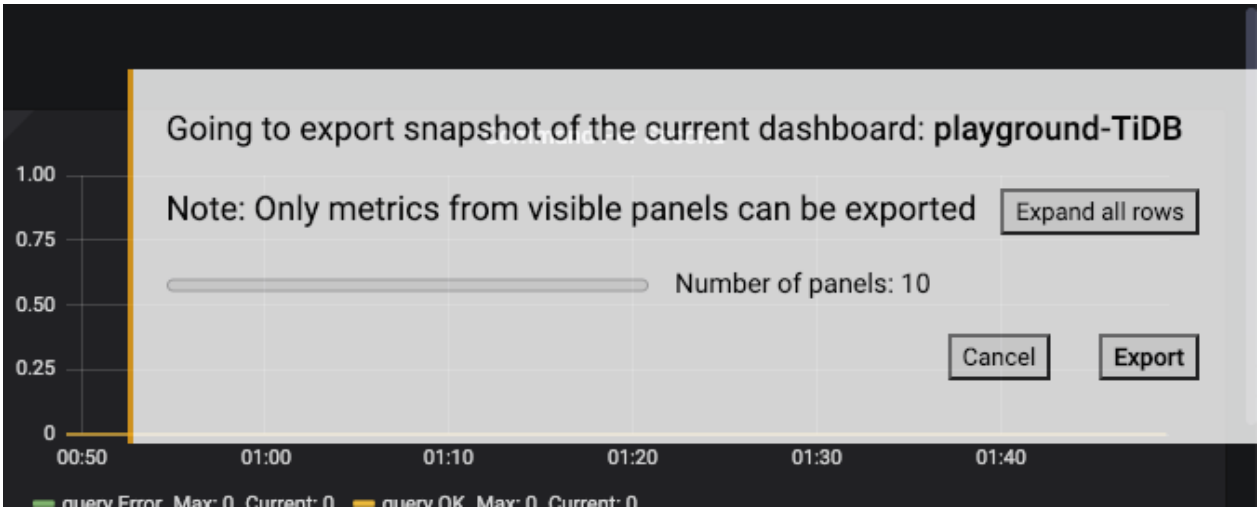


图 48: 运行用户脚本后的 MetricsTool Exporter 截图

- 快照可视化：通过网页端可视化工具Visualizer 将快照导出文件可视化。快照经过可视化后，操作起来与实际的Grafana Dashboard 无异。

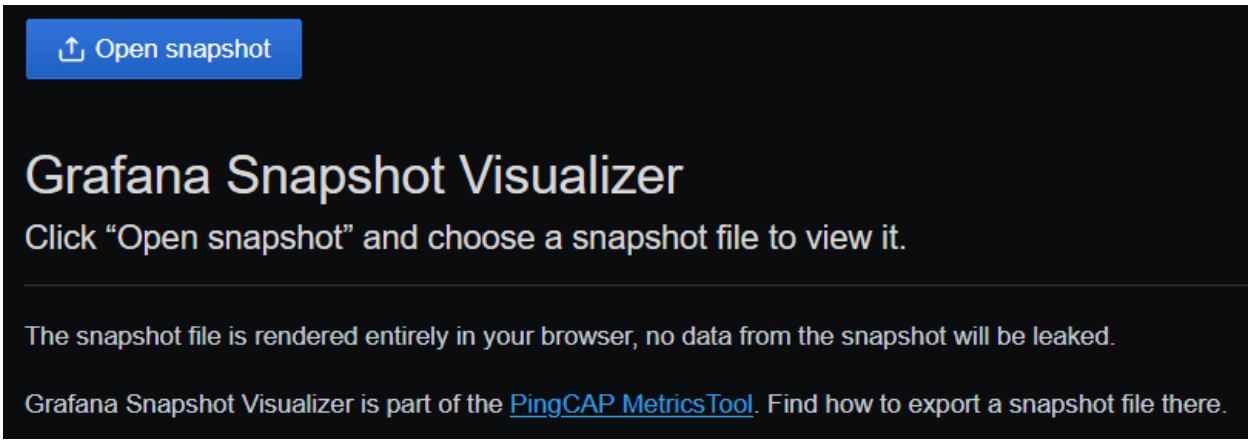


图 49: MetricsTool Visualizer 截图

- 导入快照：介绍如何将导出的快照重新导入到已有的Grafana 实例中。

6.4.2 FAQs

6.4.2.1 与直接截图及导出 PDF 相比，MetricTool 有什么优势？

MetricsTool 导出的快照文件包含快照生成时的监控指标实际数值。你可以通过Visualizer 与渲染的图表进行交互，比如切换序列、选择一个较小的时间范围以及检查特定时间点的监控数据值等，就像在操作一个实际的 Grafana Dashboard 一样，因此它比 PDF 文件和截图更强大。

6.4.2.2 快照文件里都包含什么？

快照文件包含所选时间范围内所有图表和面板数据的值，但不保存数据源的原始监控指标，所以无法在 Visualizer 中编辑查询表达式。

6.4.2.3 Visualizer 会将上传的快照文件保存到 PingCAP 的服务器上吗？

不会。快照文件解析全部在浏览器中完成，Visualizer 不会将任何信息发送给 PingCAP。你可以放心地使用 Visualizer 查看带有敏感信息的快照文件，不用担心信息会泄露给第三方。

6.4.2.4 MetricsTool 可以导出除Grafana 外其他监控工具的数据吗？

不能。目前该工具仅支持在Grafana v6.x.x 上使用。

6.4.2.5 可以在所有监控指标数据都加载完毕前就运行脚本吗？

可以。虽然脚本会弹出提示，让你等所有监控数据加载完毕后再运行，但可以手动跳过等待并导出快照，以免有些监控数据加载的时间过长。

6.4.2.6 快照文件可视化后，可以通过网页链接分享吗？

不能。但你可以分享快照文件，并说明如何使用Visualizer 查看。如果确实需要通过网页链接分享，可以尝试使用 Grafana 内置的 `snapshot.raintank.io` 服务。但在这样做之前，要确保不会泄漏隐私信息。

6.5 TiDB 集群报警规则

本文介绍了TiDB 集群中各组件的报警规则，包括TiDB、 TiKV、 PD、 TiFlash、 TiDB Binlog、 TiCDC、 Node_exporter 和 Blackbox_exporter 的各报警项的规则描述及处理方法。

按照严重程度由高到低，报警项可分为紧急级别> 严重级别> 警告级别三类。该分级适用于以下各组件的报警项。

严重程度	说明
紧急级别	最高严重程度，服务不可用，通常由于服务停止或节点故障导致，此时需要马上进行人工干预
严重级别	服务可用性下降，需要用户密切关注异常指标
警告级别	对某一问题或错误的提醒

6.5.1 TiDB 报警规则

本节介绍了TiDB 组件的报警项。

6.5.1.1 紧急级别报警项

6.5.1.1.1 TiDB_schema_error

- 报警规则：

  ```
  increase(tidb_session_schema_lease_error_total{type="outdated"}[15m])>  0
  ```

- 规则描述：

  TiDB 在一个 Lease 时间内没有重载到最新的 Schema 信息。如果TiDB 无法继续对外提供服务，则报警。

- 处理方法：

  该问题通常由于TiKV Region 不可用或超时导致，需要看TiKV 的监控指标定位问题。

6.5.1.1.2 TiDB_tikvclient_region_err_total

- 报警规则：

  ```
  increase(tidb_tikvclient_region_err_total[10m])>  6000
  ```

- 规则描述：

  TiDB server 根据自己的缓存信息访问TiKV 的 Region leader。 如果Region leader 有变化或者当前TiKV 的 Region 信息与TiDB 的缓存不一致，就会产生Region 缓存错误。如果在 10 分钟之内该错误多于 6000 次，则报警。

- 处理方法：

  查看TiKV-Details > Cluster 面板，检查leader 的分布是否均衡。

6.5.1.1.3 TiDB_domain_load_schema_total

- 报警规则：

  ```
  increase(tidb_domain_load_schema_total{type="failed"}[10m])> 10
  ```

- 规则描述：

  TiDB 重载最新的 Schema 信息失败的总次数。如果在 10 分钟之内重载失败次数超过 10 次，则报警。

- 处理方法：

  参考TiDB_schema_error 的处理方法。

6.5.1.1.4 TiDB_monitor_keep_alive

- 报警规则：

  ```
  increase(tidb_monitor_keep_alive_total[10m])< 100
  ```

- 规则描述：

  表示TiDB 的进程是否仍然存在。如果在 10 分钟之内 tidb_monitor_keep_alive_total 增加次数少于 100，则 TiDB 的进程可能已经退出，此时会报警。

- 处理方法：

  - 检查TiDB 进程是否 OOM。
  - 检查机器是否发生了重启。

6.5.1.2 严重级别报警项

6.5.1.2.1 TiDB_server_panic_total

- 报警规则：

  ```
  increase(tidb_server_panic_total[10m])> 0
  ```

- 规则描述：

  发生崩溃的TiDB 线程的数量。当出现崩溃的时候会报警。该线程通常会被恢复，否则TiDB 会频繁重启。

- 处理方法：

  收集panic 日志，定位原因。

6.5.1.3 警告级别报警项

6.5.1.3.1 TiDB_memory_abnormal

- 报警规则：

```
go_memstats_heap_inuse_bytes{job="tidb"} > 1e+10
```

- 规则描述：

  对TiDB 内存使用量的监控。如果内存使用大于 10 G，则报警。

- 处理方法：

  通过 HTTP API 来排查 goroutine 泄露的问题。

6.5.1.3.2 TiDB_query_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 1
```

- 规则描述：

  TiDB 处理请求的延时。如果延迟大于 1 秒的概率超过 99%，则报警。

- 处理方法：

  查看TiDB 的日志，搜索 SLOW_QUERY 和 TIME_COP_PROCESS 关键字，查找慢 SQL。

6.5.1.3.3 TiDB_server_event_error

- 报警规则：

```
increase(tidb_server_event_total{type=~"server_start|server_hang"}[15m])>  0
```

- 规则描述：

  TiDB 服务中发生的事件数量。当出现以下事件的时候会报警：

  1. start：TiDB 服务启动。
  2. hang：当发生了 Critical 级别的事件时（目前只有 Binlog 写不进去一种情况），TiDB 进入 hang 模式，并等待人工 Kill。

- 处理方法：

  - 重启TiDB 以恢复服务。
  - 检查TiDB Binlog 服务是否正常。

6.5.1.3.4 TiDB_tikvclient_backoff_seconds_count

- 报警规则：

  ```
  increase(tidb_tikvclient_backoff_seconds_count[10m])> 10
  ```

- 规则描述：

  TiDB 访问TiKV 发生错误时发起重试的次数。如果在 10 分钟之内重试次数多于 10 次，则报警。

- 处理方法：

  查看TiKV 的监控状态。

6.5.1.3.5 TiDB_monitor_time_jump_back_error

- 报警规则：

  ```
  increase(tidb_monitor_time_jump_back_total[10m])> 0
  ```

- 规则描述：

  如果TiDB 所在机器的时间发生了回退，则报警。

- 处理方法：

  排查 NTP 配置。

6.5.1.3.6 TiDB_ddl_waiting_jobs

- 报警规则：

  ```
  sum(tidb_ddl_waiting_jobs)> 5
  ```

- 规则描述：

  如果TiDB 中等待执行的 DDL 任务的数量大于 5，则报警。

- 处理方法：

  通过 `admin show ddl` 语句检查是否有耗时的add index 操作正在执行。

6.5.2 PD 报警规则

本节介绍了 PD 组件的报警项。

6.5.2.1 紧急级别报警项

6.5.2.1.1 PD_cluster_down_store_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_down_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

PD 长时间（默认配置是 30 分钟）没有收到TiKV/TiFlash 心跳。

- 处理方法：

  - 检查TiKV/TiFlash 进程是否正常、网络是否隔离以及负载是否过高，并尽可能地恢复服务。
  - 如果确定TiKV/TiFlash 无法恢复，可做下线处理。

6.5.2.2 严重级别报警项

6.5.2.2.1 PD_etcd_write_disk_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket[1m]))by (
↪ instance, job, le))> 1
```

- 规则描述：

fsync 操作延迟大于 1s，代表 etcd 写盘慢，这很容易引起 PD leader 超时或者TSO 无法及时存盘等问题，从而导致整个集群停止服务。

- 处理方法：

  - 排查写入慢的原因。可能是由于其他服务导致系统负载过高。可以检查 PD 本身是否占用了大量 CPU 或 IO 资源。
  - 可尝试重启 PD 或手动transfer leader 至其他的 PD 来恢复服务。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

6.5.2.2.2 PD_miss_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="miss-peer-region-count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

Region 的副本数小于 `max-replicas` 配置的值。

- 处理方法：

  - 查看是否有TiKV 宕机或在做下线操作，尝试定位问题产生的原因。
  - 观察region health 面板，查看 `miss-peer-region-count` 是否在不断减少。

6.5.2.3 警告级别报警项

6.5.2.3.1 PD_cluster_lost_connect_store_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_disconnected_count"})by (instance)> 0)and (sum(etcd_server_is_leader
'→ )by (instance)> 0)
```

- 规则描述：

  PD 在 20 秒之内未收到TiKV/TiFlash 上报心跳。正常情况下是每 10 秒收到 1 次心跳。

- 处理方法：

  - 排查是否在重启TiKV/TiFlash。
  - 检查TiKV/TiFlash 进程是否正常、网络是否隔离以及负载是否过高，并尽可能地恢复服务。
  - 如果确定TiKV/TiFlash 无法恢复，可做下线处理。
  - 如果确定TiKV/TiFlash 可以恢复，但在短时间内还无法恢复，可以考虑延长 max-down-time 配置，防止超时后TiKV/TiFlash 被判定为无法恢复并开始搬移数据。

6.5.2.3.2 PD_cluster_unhealthy_tikv_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_unhealth_count"})by (instance)> 0)and (sum(etcd_server_is_leader
'→ )by (instance)> 0)
```

- 规则描述：

  表示存在异常状态的 Store。如果这种状态持续一段时间（取决于配置的max-store-down-time，默认值为 30m），Store 可能会变为 Offline 状态并触发PD_cluster_down_store_nums 报警。

- 处理方法：

  检查 TiKV Store 的状态。

6.5.2.3.3 PD_cluster_low_space

- 报警规则：

```
(sum(pd_cluster_status{type="store_low_space_count"})by (instance)> 0)and (sum(etcd_server_is_leader
'→ )by (instance)> 0)
```

- 规则描述：

  表示TiKV/TiFlash 节点空间不足。

- 处理方法：

  - 检查集群中的空间是否普遍不足。如果是，则需要扩容。
  - 检查Region balance 调度是否有问题。如果有问题，会导致数据分布不均衡。
  - 检查是否有文件占用了大量磁盘空间，比如日志、快照、core dump 等文件。
  - 降低该节点的Region weight 来减少数据量。
  - 无法释放空间时，可以考虑主动下线该节点，防止由于磁盘空间不足而宕机。

6.5.2.3.4 PD_etcd_network_peer_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_network_peer_round_trip_time_seconds_bucket[1m]))by (
↪ To, instance, job, le))> 1
```

- 规则描述：

PD 节点之间网络延迟高，严重情况下会导致leader 超时和TSO 存盘超时，从而影响集群服务。

- 处理方法：

  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

6.5.2.3.5 PD_tidb_handle_requests_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(pd_client_request_handle_requests_duration_seconds_bucket{
↪ type="tso"}[1m]))by (instance, job, le))> 0.1
```

- 规则描述：

PD 处理TSO 请求耗时过长，一般是由于负载过高。

- 处理方法：

  - 检查服务器负载状况。
  - 使用 pprof 抓取 PD 的 CPU profile 进行分析。
  - 手动切换 PD leader。
  - 如果是环境问题，则将有问题的 PD 下线替换。

6.5.2.3.6 PD_down_peer_region_nums

- 报警规则：

```
(sum(pd_regions_status{type="down-peer-region-count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

Raft leader 上报有不响应peer 的 Region 数量。

- 处理方法：

  - 检查是否有TiKV 宕机，或刚发生重启，或者繁忙。
  - 观察region health 面板，检查 down_peer_region_count 是否在不断减少。
  - 检查是否有TiKV 之间网络不通。

6.5.2.3.7 PD_pending_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="pending-peer-region-count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft log 落后的Region 过多。由于调度产生少量的pending peer 是正常的，但是如果持续很高，就可能有问题。

- 处理方法：

  - 观察region health 面板，检查 pending_peer_region_count 是否在不断减少。
  - 检查TiKV 之间的网络状况，特别是带宽是否足够。

6.5.2.3.8 PD_leader_change

- 报警规则：

```
count(changes(pd_tso_events{type="save"}[10m])> 0)>= 2
```

- 规则描述：

  近期发生了 PD leader 切换。

- 处理方法：

  - 排除人为因素，比如重启 PD、手动 transfer leader 或调整leader 优先级等。
  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

6.5.2.3.9 TiKV_space_used_more_than_80%

- 报警规则：

```
sum(pd_cluster_status{type="storage_size"})/  sum(pd_cluster_status{type="storage_capacity"})
↪ * 100 > 80
```

- 规则描述：

  集群空间占用超过 80%。

- 处理方法：

  - 确认是否需要扩容。
  - 排查是否有文件占用了大量磁盘空间，比如日志、快照或core dump 等文件。

6.5.2.3.10 PD_system_time_slow

- 报警规则：

  ```
  changes(pd_tso_events{type="system_time_slow"}[10m])>= 1
  ```

- 规则描述：

  系统时间可能发生回退。

- 处理方法：

  检查系统时间设置是否正确。

6.5.2.3.11 PD_no_store_for_making_replica

- 报警规则：

  ```
  increase(pd_checker_event_count{type="replica_checker", name="no_target_store"}[1m])> 0
  ```

- 规则描述：

  没有合适的 store 用来补副本。

- 处理方法：

  - 检查 store 是否空间不足。
  - 根据label 配置（如果有这个配置的话）来检查是否有可以补副本的 store。

6.5.2.3.12 PD_cluster_slow_tikv_nums

- 报警规则：

  ```
  sum(pd_cluster_status{type="store_slow_count"})by (instance)> 0)and (sum(etcd_server_is_leader
  ↪ )by (instance)> 0
  ```

- 规则描述：

  某一个TiKV 被检测为慢节点。慢节点的检测由TiKV `raftstore.inspect-interval` 参数控制，参见TiKV 配置文件描述。

- 处理方法：

  - 观察TiKV-Details > PD 面板，查看 Store Slow Score 监控指标，找出指标数值超过 80 的节点，该节点即为被检测到的慢节点。
  - 观察TiKV-Details > Raft IO 面板，查看延迟是否升高。如果延迟很高，表明磁盘可能存在瓶颈。
  - 调大TiKV `raftstore.inspect-interval` 参数，提高延迟检测的超时上限。
  - 如果需要进一步分析报警的TiKV 节点的性能问题，找到优化方法，可以参考性能分析和优化方法。

6.5.3 TiKV 报警规则

本节介绍了TiKV 组件的报警项。

6.5.3.1 紧急级别报警项

6.5.3.1.1 TiKV_memory_used_too_fast

- 报警规则：

```
process_resident_memory_bytes{job=~"tikv",instance=~".*"} - (process_resident_memory_bytes{
↪ job=~"tikv",instance=~".*"} offset 5m)> 5*1024*1024*1024
```

- 规则描述：

  目前没有和内存相关的TiKV的监控，你可以通过 Node_exporter 监控集群内机器的内存使用情况。如上规则表示，如果在 5 分钟之内内存使用超过 5GB（TiKV 内存占用的太快），则报警。

- 处理方法：

  调整 `rocksdb.defaultcf` 和 `rocksdb.writecf` 的 `block-cache-size` 的大小。

6.5.3.1.2 TiKV_GC_can_not_work

- 报警规则：

```
sum(increase(tikv_gcworker_gc_tasks_vec{task="gc"}[1d]))< 1 and (sum(increase(tikv_gc_compaction_filter_per
↪ [1d]))< 1 and sum(increase(tikv_engine_event_total{db="kv", cf="write", type="compaction
↪ "}[1d]))>= 1)
```

- 规则描述：

  在 24 小时内一个TiKV实例上没有成功执行GC，说明GC不能正常工作了。短期内GC不运行不会造成太大的影响，但如果GC一直不运行，版本会越来越多，从而导致查询变慢。

- 处理方法：

  1. 执行 `SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME="tikv_gc_leader_desc"` 来找到 gc leader 对应的 `tidb-server`；
  2. 查看该 `tidb-server` 的日志，grep gc_worker tidb.log；
  3. 如果发现这段时间一直在resolve locks（最后一条日志是 `start resolve locks`）或者 delete ranges（最后一条日志是 `start delete {number} ranges`），说明 GC 进程是正常的。否则请从 PingCAP 官方或TiDB 社区获取支持。

6.5.3.2 严重级别报警项

6.5.3.2.1 TiKV_server_report_failure_msg_total

- 报警规则：

```
sum(rate(tikv_server_report_failure_msg_total{type="unreachable"}[10m]))BY (store_id)> 10
```

- 规则描述：

  表明无法连接远端的TiKV。

- 处理方法：

    1. 检查网络是否通畅。
    2. 检查远端TiKV 是否挂掉。
    3. 如果远端TiKV 没有挂掉，检查压力是否太大，参考TiKV_channel_full_total 处理方法。

6.5.3.2.2 TiKV_channel_full_total

- 报警规则：

```
sum(rate(tikv_channel_full_total[10m]))BY (type, instance)> 0
```

- 规则描述：

    该错误通常是因为Raftstore 线程卡死，TiKV 的压力已经非常大了。

- 处理方法：

    1. 观察TiKV-Details > Raft Propose 面板，查看这个报警的TiKV 节点是否明显比其他TiKV 高很多。如果是，表明这个TiKV 上有热点，需要检查热点调度是否能正常工作。
    2. 观察TiKV-Details > Raft IO 面板，查看延迟是否升高。如果延迟很高，表明磁盘可能存在瓶颈。
    3. 观察TiKV-Details > Raft process 面板， 关注 tick duration 是否很高。如果是， 需要将 TiKV 配置项raftstore.raft-base-tick-interval 设置为 "2s"。

6.5.3.2.3 TiKV_write_stall

- 报警规则：

```
delta(tikv_engine_write_stall[10m])> 0
```

- 规则描述：

    RocksDB 写入压力太大，出现了 stall。

- 处理方法：

    1. 观察磁盘监控，排除磁盘问题。
    2. 看 TiKV 是否有写入热点。
    3. 在 [rocksdb] 和 [raftdb] 配置下调大 max-sub-compactions 的值。

6.5.3.2.4 TiKV_raft_log_lag

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_log_lag_bucket[1m]))by (le, instance))>
↪ 5000
```

- 规则描述：

    这个值偏大，表明 Follower 已经远远落后于 Leader，Raft 没法正常同步了。可能的原因是 Follower 所在的 TiKV 卡住或者挂掉了。

6.5.3.2.5 TiKV_async_request_snapshot_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
'→ type="snapshot"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明Raftstore 负载压力很大，可能已经卡住。

- 处理方法：

  参考TiKV_channel_full_total 的处理方法。

6.5.3.2.6 TiKV_async_request_write_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
'→ type="write"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明Raft write 耗时很长。

- 处理方法：

  1. 观察TiKV-Details > Raft propose 面板，查看这个报警的TiKV 节点的 99% Propose wait duration per server 是否明显比其他TiKV 高很多。如果是，表明这个TiKV 上有热点，需要检查热点调度是否能正常工作。
  2. 观察TiKV-Details > Raft IO 面板，查看延迟是否升高。如果延迟很高，表明磁盘可能存在瓶颈。
  3. 如果需要进一步分析报警的TiKV 节点的性能问题，找到优化方法，可以参考性能分析和优化方法。

6.5.3.2.7 TiKV_coprocessor_request_wait_seconds

- 报警规则：

```
histogram_quantile(0.9999, sum(rate(tikv_coprocessor_request_wait_seconds_bucket[1m]))by (le
'→ , instance, req))> 10
```

- 规则描述：

  这个值偏大，表明 Coprocessor worker 压力很大。可能有比较慢的任务卡住了 Coprocessor 线程。

- 处理方法：

  1. 从TiDB 日志中查看慢查询日志，看查询是否用到了索引或全表扫，或者看是否需要做analyze。
  2. 排查是否有热点。
  3. 查看 Coprocessor 监控，看 `coporcessor table/index scan` 里 `total` 和 `process` 是否匹配。如果相差太大，表明做了太多的无效查询。看是否有 `over seek bound`，如果有，表明版本太多，GC 工作不及时，需要增大并行GC 的线程数。

6.5.3.2.8 TiKV_raftstore_thread_cpu_seconds_total

- 报警规则：

```
sum(rate(tikv_thread_cpu_seconds_total{name=~"raftstore_.*"}[1m]))by (instance, name)> 1.6
```

- 规则描述：

  Raftstore 线程压力太大。

- 处理方法：

  参考TiKV_channel_full_total 的处理方法。

6.5.3.2.9 TiKV_raft_append_log_duration_secs

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_append_log_duration_seconds_bucket[1m]))by
↪ (le, instance))> 1
```

- 规则描述：

  表示append Raft log 的耗时，如果高，通常是因为IO 太忙了。

6.5.3.2.10 TiKV_raft_apply_log_duration_secs

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_apply_log_duration_seconds_bucket[1m]))by (
↪ le, instance))> 1
```

- 规则描述：

  表示apply Raft log 耗时，如果高，通常是因为IO 太忙了。

6.5.3.2.11 TiKV_scheduler_latch_wait_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_scheduler_latch_wait_duration_seconds_bucket[1m]))by
↪ (le, instance, type))> 1
```

- 规则描述：

  Scheduler 中写操作获取内存锁时的等待时间。如果这个值高，表明写操作冲突较多，也可能是某些引起冲突的操作耗时较长，阻塞了其它等待相同锁的操作。

- 处理方法：

  1. 查看 Scheduler-All 监控中的 scheduler command duration，看哪一个命令耗时最大。
  2. 查看 Scheduler-All 监控中的 scheduler scan details，看 total 和 process 是否匹配。如果相差太大，表明有很多无效的扫描，另外观察是否有 over seek bound，如果太多，表明 GC 不及时。
  3. 查看 Storage 监控中的 storage async snapshot/write duration，看是否Raft 操作不及时。

6.5.3.2.12 TiKV_thread_apply_worker_cpu_seconds

- 报警规则：

```
max(rate(tikv_thread_cpu_seconds_total{name=~"apply_.*"}[1m]))by (instance)> 0.9
```

- 规则描述：

  Apply Raft log 线程压力太大，已经接近或超过apply 线程的处理上限。通常是因为短期内写入的数据量太多造成的。

6.5.3.3 警告级别报警项

6.5.3.3.1 TiKV_leader_drops

- 报警规则：

```
delta(tikv_pd_heartbeat_tick_total{type="leader"}[30s])< -10
```

- 规则描述：

  该问题通常是因为Raftstore 线程卡住了。

- 处理方法：

  1. 参考TiKV_channel_full_total 的处理方法。
  2. 如果TiKV 压力很小，考虑 PD 的调度是否太频繁。可以查看 PD 页面的 Operator Create 面板，排查 PD 产生调度的类型和数量。

6.5.3.3.2 TiKV_raft_process_ready_duration_secs

- 报警规则：

```
histogram_quantile(0.999, sum(rate(tikv_raftstore_raft_process_duration_secs_bucket{type='
'→ ready'}[1m]))by (le, instance, type))> 2
```

- 规则描述：

  表示处理Raft ready 的耗时。这个值大，通常是因为append log 任务卡住了。

6.5.3.3.3 TiKV_raft_process_tick_duration_secs

- 报警规则：

```
histogram_quantile(0.999, sum(rate(tikv_raftstore_raft_process_duration_secs_bucket{type='
'→ tick' }[1m]))by (le, instance, type))> 2
```

- 规则描述：

  表示处理Raft tick 的耗时，这个值大，通常是因为Region 太多导致的。

- 处理方法：

  1. 考虑使用更高等级的日志，比如 warn 或者 error。
  2. 在 [raftstore] 配置下添加 raft-base-tick-interval = “2s”。

6.5.3.3.4 TiKV_scheduler_context_total

- 报警规则：

  ```
  abs(delta( tikv_scheduler_context_total[5m]))> 1000
  ```

- 规则描述：

  Scheduler 正在执行的写命令数量。这个值高，表示任务完成得不及时。

- 处理方法：

  参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

6.5.3.3.5 TiKV_scheduler_command_duration_seconds

- 报警规则：

  ```
  histogram_quantile(0.99, sum(rate(tikv_scheduler_command_duration_seconds_bucket[1m]))by (le
  ↪ , instance, type))> 1
  ```

- 规则描述：

  表明 Scheduler 执行命令的耗时。

- 处理方法：

  参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

6.5.3.3.6 TiKV_coprocessor_outdated_request_wait_seconds

- 报警规则：

  ```
  delta(tikv_coprocessor_outdated_request_wait_seconds_count[10m])> 0
  ```

- 规则描述：

  Coprocessor 已经过期的请求等待的时间。这个值高，表示 Coprocessor 压力已经非常大了。

- 处理方法：

  参考TiKV_coprocessor_request_wait_seconds 的处理方法。

6.5.3.3.7 TiKV_coprocessor_pending_request

- 报警规则：

  ```
  delta(tikv_coprocessor_pending_request[10m])> 5000
  ```

- 规则描述：

  Coprocessor 排队的请求。

- 处理方法：

  参考TiKV_coprocessor_request_wait_seconds 的处理方法。

6.5.3.3.8 TiKV_batch_request_snapshot_nums

- 报警规则：

```
sum(rate(tikv_thread_cpu_seconds_total{name=~"cop_.*"}[1m]))by (instance)/ (count(tikv_thread_cpu_seconds_t
'→ {name=~"cop_.*"})* 0.9)/ count(count(tikv_thread_cpu_seconds_total)by (instance))> 0
```

- 规则描述：

  某个 TiKV 的 Coprocessor CPU 使用率超过了 90%。

6.5.3.3.9 TiKV_pending_task

- 报警规则：

```
sum(tikv_worker_pending_task_total)BY (instance,name)> 1000
```

- 规则描述：

  TiKV 等待的任务数量。

- 处理方法：

  观察TiKV-Details > Task 面板，查看是哪一类任务的 `Worker pending tasks` 值偏高。

6.5.3.3.10 TiKV_low_space

- 报警规则：

```
sum(tikv_store_size_bytes{type="available"})by (instance)/ sum(tikv_store_size_bytes{type="
'→ capacity"})by (instance)< 0.2
```

- 规则描述：

  TiKV 数据量超过节点配置容量或物理磁盘容量的 80%。

- 处理方法：

  确认节点空间均衡情况，做好扩容计划。

6.5.3.3.11 TiKV_approximate_region_size

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_region_size_bucket[1m]))by (le))> 1073741824
'→
```

- 规则描述：

  TiKV split checker 扫描到的最大的Region approximate size 在 1 分钟内持续大于 1 GB。

- 处理方法：

  Region 分裂的速度不及写入的速度。为缓解这种情况，建议更新到支持 batch-split 的版本(>= 2.1.0-rc1)。如暂时无法更新，可以使用`pd-ctl operator add split-region <region_id> --policy=approximate` 手动分裂Region。

6.5.4 TiFlash 报警规则

关于TiFlash 报警规则的详细描述，参见TiFlash 报警规则。

6.5.5 TiDB Binlog 报警规则

关于TiDB Binlog 报警规则的详细描述，参见TiDB Binlog 集群监控报警文档。

6.5.6 TiCDC 报警规则

关于TiCDC 报警规则的详细描述，参见TiCDC 集群监控报警。

6.5.7 Node_exporter 主机报警规则

本节介绍了 Node_exporter 主机的报警项。

6.5.7.1 紧急级别报警项

6.5.7.1.1 NODE_disk_used_more_than_80%

- 报警规则：

```
node_filesystem_avail_bytes{fstype=~"(ext.|xfs)", mountpoint!~"/boot"} / node_filesystem_size_bytes
↪ {fstype=~"(ext.|xfs)", mountpoint!~"/boot"} * 100 <= 20
```

- 规则描述：

  机器磁盘空间使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -h` 命令，查看磁盘空间使用率，做好扩容计划。

6.5.7.1.2 NODE_disk_inode_more_than_80%

- 报警规则：

```
node_filesystem_files_free{fstype=~"(ext.|xfs)"} / node_filesystem_files{fstype=~"(ext.|xfs)
↪ "} * 100 < 20
```

- 规则描述：

  机器磁盘挂载目录文件系统inode 使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -i` 命令，查看磁盘挂载目录文件系统inode 使用率，做好扩容计划。

6.5.7.1.3 NODE_disk_readonly

- 报警规则：

  ```
  node_filesystem_readonly{fstype=~"(ext.|xfs)"} == 1
  ```

- 规则描述：

  磁盘挂载目录文件系统只读，无法写入数据，一般是因为磁盘故障或文件系统损坏。

- 处理方法：

  - 登录机器创建文件测试是否正常。
  - 检查该服务器硬盘指示灯是否正常，如异常，需更换磁盘并修复该机器文件系统。

6.5.7.2 严重级别报警项

6.5.7.2.1 NODE_memory_used_more_than_80%

- 报警规则：

  ```
  (((node_memory_MemTotal_bytes-node_memory_MemFree_bytes-node_memory_Cached_bytes)/(node_memory_MemTotal_byt
  ↪ )*100))>= 80
  ```

- 规则描述：

  机器内存使用率超过 80%。

- 处理方法：

  - 在 Grafana Node Exporter 页面查看该主机的 Memory 面板，检查 Used 是否过高，Available 内存是否过低。
  - 登录机器，执行 `free -m` 命令查看内存使用情况，执行 top 看是否有异常进程的内存使用率过高。

6.5.7.3 警告级别报警项

6.5.7.3.1 NODE_node_overload

- 报警规则：

  ```
  (node_load5 / count without (cpu, mode)(node_cpu_seconds_total{mode="system"}))> 1
  ```

- 规则描述：

  机器 CPU 负载较高。

- 处理方法：

  - 在 Grafana Node exporter 页面上查看该主机的 CPU Usage 及 Load Average，检查是否过高。
  - 登录机器，执行 top 查看load average 及 CPU 使用率，看是否是异常进程的 CPU 使用率过高。

6.5.7.3.2 NODE_cpu_used_more_than_80%

- 报警规则：

  ```
  avg(irate(node_cpu_seconds_total{mode="idle"}[5m]))by(instance)* 100 <= 20
  ```

- 规则描述：

  机器 CPU 使用率超过 80%。

- 处理方法：

  - 在 Grafana Node exporter 页面上查看该主机的 CPU Usage 及 Load Average，检查是否过高。
  - 登录机器，执行 top 查看load average 及 CPU 使用率，看是否是异常进程的 CPU 使用率过高。

6.5.7.3.3 NODE_tcp_estab_num_more_than_50000

- 报警规则：

  ```
  node_netstat_Tcp_CurrEstab > 50000
  ```

- 规则描述：

  机器 establish 状态的TCP 链接超过 50,000。

- 处理方法：

  登录机器执行 ss -s 可查看当前系统 estab 状态的TCP 链接数，执行 netstat 查看是否有异常链接。

6.5.7.3.4 NODE_disk_read_latency_more_than_32ms

- 报警规则：

  ```
  ((rate(node_disk_read_time_seconds_total{device=~".+"}[5m])/  rate(node_disk_reads_completed_total
  ′→ {device=~".+"}[5m]))or  (irate(node_disk_read_time_seconds_total{device=~".+"}[5m])/ irate
  ′→ (node_disk_reads_completed_total{device=~".+"}[5m])))* 1000 > 32
  ```

- 规则描述：

  磁盘读延迟超过 32 毫秒。

- 处理方法：

  - 查看 Grafana Disk Performance Dashboard 观察磁盘使用情况。
  - 查看 Disk Latency 面板观察磁盘的读延迟。
  - 查看 Disk IO Utilization 面板观察IO 使用率。

6.5.7.3.5 NODE_disk_write_latency_more_than_16ms

- 报警规则：

```
((rate(node_disk_write_time_seconds_total{device=~".+"}[5m])/  rate(node_disk_writes_completed_total
↪ {device=~".+"}[5m]))or (irate(node_disk_write_time_seconds_total{device=~".+"}[5m])/
↪ irate(node_disk_writes_completed_total{device=~".+"}[5m])))> 16
```

- 规则描述：

  机器磁盘写延迟超过 16 毫秒。

- 处理方法：

  - 查看 Grafana Disk Performance Dashboard 观察磁盘使用情况。
  - 查看 Disk Latency 面板可查看磁盘的写延迟。
  - 查看 Disk IO Utilization 面板可查看IO 使用率。

6.5.8 Blackbox_exporter TCP、 ICMP 和 HTTP 报警规则

本节介绍了 Blackbox_exporter TCP、 ICMP 和 HTTP 的报警项。

6.5.8.1 紧急级别报警项

6.5.8.1.1 TiDB_server_is_down

- 报警规则：

```
probe_success{group="tidb"} == 0
```

- 规则描述：

  TiDB 服务端口探测失败。

- 处理方法：

  - 检查TiDB 服务所在机器是否宕机。
  - 检查TiDB 进程是否存在。
  - 检查监控机与TiDB 服务所在机器之间网络是否正常。

6.5.8.1.2 TiFlash_server_is_down

- 报警规则：

```
probe_success{group="tiflash"} == 0
```

- 规则描述：

  TiFlash 服务端口探测失败。

- 处理方法：

  - 检查TiFlash 服务所在机器是否宕机。
  - 检查TiFlash 进程是否存在。
  - 检查监控机与TiFlash 服务所在机器之间网络是否正常。

6.5.8.1.3 Pump_server_is_down

- 报警规则：

  ```
  probe_success{group="pump"} == 0
  ```

- 规则描述：

  Pump 服务端口探测失败。

- 处理方法：

  - 检查 Pump 服务所在机器是否宕机。
  - 检查 Pump 进程是否存在。
  - 检查监控机与 Pump 服务所在机器之间网络是否正常。

6.5.8.1.4 Drainer_server_is_down

- 报警规则：

  ```
  probe_success{group="drainer"} == 0
  ```

- 规则描述：

  Drainer 服务端口探测失败。

- 处理方法：

  - 检查 Drainer 服务所在机器是否宕机。
  - 检查 Drainer 进程是否存在。
  - 检查监控机与 Drainer 服务所在机器之间网络是否正常。

6.5.8.1.5 TiKV_server_is_down

- 报警规则：

  ```
  probe_success{group="tikv"} == 0
  ```

- 规则描述：

  TiKV 服务端口探测失败。

- 处理方法：

  - 检查TiKV 服务所在机器是否宕机。
  - 检查TiKV 进程是否存在。
  - 检查监控机与TiKV 服务所在机器之间网络是否正常。

6.5.8.1.6 PD_server_is_down

- 报警规则：

  ```
  probe_success{group="pd"} == 0
  ```

- 规则描述：

  PD 服务端口探测失败。

- 处理方法：

  - 检查 PD 服务所在机器是否宕机。
  - 检查 PD 进程是否存在。
  - 检查监控机与 PD 服务所在机器之间网络是否正常。

6.5.8.1.7 Node_exporter_server_is_down

- 报警规则：

  ```
  probe_success{group="node_exporter"} == 0
  ```

- 规则描述：

  Node_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Node_exporter 服务所在机器是否宕机。
  - 检查 Node_exporter 进程是否存在。
  - 检查监控机与 Node_exporter 服务所在机器之间网络是否正常。

6.5.8.1.8 Blackbox_exporter_server_is_down

- 报警规则：

  ```
  probe_success{group="blackbox_exporter"} == 0
  ```

- 规则描述：

  Blackbox_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Blackbox_exporter 服务所在机器是否宕机。
  - 检查 Blackbox_exporter 进程是否存在。
  - 检查监控机与 Blackbox_exporter 服务所在机器之间网络是否正常。

6.5.8.1.9 Grafana_server_is_down

- 报警规则：

  ```
  probe_success{group="grafana"} == 0
  ```

- 规则描述：

  Grafana 服务端口探测失败。

- 处理方法：

  - 检查Grafana 服务所在机器是否宕机。
  - 检查Grafana 进程是否存在。
  - 检查监控机与Grafana 服务所在机器之间网络是否正常。

6.5.8.1.10 Pushgateway_server_is_down

- 报警规则：

  ```
  probe_success{group="pushgateway"} == 0
  ```

- 规则描述：

  Pushgateway 服务端口探测失败。

- 处理方法：

  - 检查 Pushgateway 服务所在机器是否宕机。
  - 检查 Pushgateway 进程是否存在。
  - 检查监控机与 Pushgateway 服务所在机器之间网络是否正常。

6.5.8.1.11 Kafka_exporter_is_down

- 报警规则：

  ```
  probe_success{group="kafka_exporter"} == 0
  ```

- 规则描述：

  Kafka_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Kafka_exporter 服务所在机器是否宕机。
  - 检查 Kafka_exporter 进程是否存在。
  - 检查监控机与 Kafka_exporter 服务所在机器之间网络是否正常。

6.5.8.1.12 Pushgateway_metrics_interface

- 报警规则：

  ```
  probe_success{job="blackbox_exporter_http"} == 0
  ```

- 规则描述：

  Pushgateway 服务 http 接口探测失败。

- 处理方法：

  - 检查 Pushgateway 服务所在机器是否宕机。
  - 检查 Pushgateway 进程是否存在。
  - 检查监控机与 Pushgateway 服务所在机器之间网络是否正常。

6.5.8.2 警告级别报警项

6.5.8.2.1 BLACKER_ping_latency_more_than_1s

- 报警规则：

  ```
  max_over_time(probe_duration_seconds{job=~"blackbox_exporter.*_icmp"}[1m])>  1
  ```

- 规则描述：

  Ping 延迟超过 1 秒。

- 处理方法：

  - 在 Grafana Blackbox Exporter 页面上检查两个节点间的ping 延迟是否太高。
  - 在 Grafana Node Exporter 页面的TCP 面板上检查是否有丢包。

6.6 TiFlash 报警规则

本文介绍了TiFlash 集群的报警规则。

6.6.1 TiFlash_schema_error

- 报警规则：

  ```
  increase(tiflash_schema_apply_count{type="failed"}[15m])>  0
  ```

- 规则描述：

  出现 schema apply 错误时报警。

- 处理方法：

  可能是逻辑问题，联系TiFlash 开发人员。

6.6.2 TiFlash_schema_apply_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_schema_apply_duration_seconds_bucket[1m]))BY (le,
↪ instance))> 20
```

- 规则描述：

apply 时间超过 20 秒的概率超过 99% 时报警。

- 处理方法：

可能是TiFlash 存储引擎内部问题，联系TiFlash 开发人员。

6.6.3 TiFlash_raft_read_index_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_raft_read_index_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 3
```

- 规则描述：

read index 时间超过 3 秒的概率超过 99% 时报警。

> 注意：
> read index 请求是发送给TiKV leader 的 kvproto 请求，TiKV region 的重试，或 Store 的繁忙/网络问题都可能导致read index 请求时间过长。

- 处理方法：

可能TiKV 集群分裂/迁移频繁，导致频繁重试，可以查看TiKV 集群状态确认。

6.6.4 TiFlash_raft_wait_index_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_raft_wait_index_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 2
```

- 规则描述：

TiFlash 等待Region Raft Index 的时间超过 2 秒的概率超过 99% 时报警。

- 处理方法：

可能TiKV 和 Proxy 的通信出现问题，联系TiFlash 开发人员确认。

6.7 自定义监控组件的配置

使用TiUP部署TiDB集群时，TiUP会同时自动部署Prometheus、Grafana和Alertmanager等监控组件，并且在集群扩容中自动为新增节点添加监控配置。

需要注意的是，TiUP会使用自己的配置参数覆盖监控组件的配置，如果你直接修改监控组件的配置文件，修改的配置文件可能在对集群进行deploy/scale-out/scale-in/reload等操作中被TiUP所覆盖，导致配置不生效。

如果需要自定义Prometheus、Grafana和Alertmanager等监控组件的配置，请参考本文在TiDB集群的拓扑配置topology.yaml文件中添加对应的配置项。

> **注意：**
>
> - 在自定义监控组件的配置时，请勿直接修改监控组件的配置文件。因为在对集群进行deploy/scale-out/scale-in/reload等操作时，TiUP会使用自己的配置参数覆盖监控组件的配置。
> - 如果监控组件不是由TiUP部署和管理，可以直接修改监控组件的配置文件，无需参考本文档。
> - 本文所述功能在TiUP v1.9.0及后续版本支持，使用本功能前请检查TiUP版本号。

6.7.1 自定义 Prometheus 配置

目前，TiUP支持自定义Prometheus的rule配置和scrape配置。

6.7.1.1 自定义 Prometheus rule

1. 将自定义的rule配置文件放到TiUP所在机器的某个目录下。
2. 在TiDB集群的拓扑配置topology.yaml文件中，将自定义规则文件目录rule_dir设置为实际rule配置文件的目录。

以下为topology.yaml文件中的monitoring_servers配置示例:

```
## # Server configs are used to specify the configuration of Prometheus Server.
monitoring_servers:
  # # The ip address of the Monitoring Server.
  - host: 127.0.0.1
    rule_dir: /home/tidb/prometheus_rule    # prometheus rule dir on TiUP machine
```

上述配置后，在集群进行deploy/scale-out/scale-in/reload操作时，TiUP将读取本机/home/tidb/prometheus_rule路径下的自定义rule，然后将该配置发送到Prometheus Server，替换默认配置规则。

6.7.1.2 自定义 Prometheus scrape 配置

1. 打开TiDB 集群的拓扑配置文件topology.yaml。
2. 在 monitoring_servers 的配置部分添加additional_scrape_conf 字段。

以下为topology.yaml 文件中的monitoring_servers 配置示例:

```
monitoring_servers:
- host: xxxxxxx
  ssh_port: 22
  port: 9090
  deploy_dir: /tidb-deploy/prometheus-9090
  data_dir: /tidb-data/prometheus-9090
  log_dir: /tidb-deploy/prometheus-9090/log
  external_alertmanagers: []
  arch: amd64
  os: linux
  additional_scrape_conf:
    metric_relabel_configs:
      - source_labels: [__name__]
        separator: ;
        regex: tikv_thread_nonvoluntary_context_switches|tikv_thread_voluntary_context_switches|
            ↪ tikv_threads_io_bytes_total
        action: drop
      - source_labels: [__name__,name]
        separator: ;
        regex: tikv_thread_cpu_seconds_total;(tokio|rocksdb).+
        action: drop
```

上述配置后，在集群进行deploy/scale-out/scale-in/reload 操作时，TiUP 会将additional_scrape_conf 字段的内容会添加到 Prometheus 配置文件的对应参数中。

6.7.2 自定义Grafana 配置

目前，TiUP 支持自定义Grafana Dashboard 和其他配置。

6.7.2.1 自定义Grafana Dashboard

1. 将自定义的 Dashboard 配置文件放到TiUP 所在机器的某个目录下。
2. 在 TiDB 集群的拓扑配置 topology.yaml 文件中，将自定义规则文件目录 dashboard_dir 设置为实际放置 Dashboard 配置文件的目录。

以下为topology.yaml 文件中的 grafana_servers 配置示例：

```
## # Server configs are used to specify the configuration of Grafana Servers.
grafana_servers:
  # # The ip address of the Grafana Server.
  - host: 127.0.0.1
    dashboard_dir: /home/tidb/dashboards    # grafana dashboard dir on TiUP machine
```

上述配置后，在集群进行deploy/scale-out/scale-in/reload 操作时，TiUP 将读取本机 /home/tidb/dashboards 路径下的自定义 Dashboard，然后将该配置发送到 Grafana Server，替换默认配置规则。

6.7.2.2 自定义Grafana 其他配置

1. 打开集群配置文件topology.yaml。
2. 在 grafana_servers 的配置部分添加其他配置。

以下为topology.yaml 文件中的 [log.file] level 字段以及 smtp 配置示例：

```
## # Server configs are used to specify the configuration of Grafana Servers.
grafana_servers:
  # # The ip address of the Grafana Server.
  - host: 127.0.0.1
    config:
      log.file.level: warning
      smtp.enabled: true
      smtp.host: {IP}:{port}
      smtp.user: example@pingcap.com
      smtp.password: {password}
      smtp.skip_verify: true
```

上述配置后，在集群进行deploy/scale-out/scale-in/reload 操作时，TiUP 会将config 字段的内容会添加到 grafana 的配置文件 grafana.ini 中。

6.7.3 自定义Alertmanager 配置

目前，TiUP 支持自定义配置Alertmanager 的监听地址。

TiUP 部署的Alertmanager 默认监听alertmanager_servers.host，如果你使用代理，则无法访问Alertmanager。此时，你可以在集群配置文件topology.yaml 中添加listen_host 指定监听地址，使得Alertmanager 可以通过代理访问。推荐配置为 0.0.0.0。

以下示例将listen_host 字段设置为 0.0.0.0。

```
alertmanager_servers:
  # # The ip address of the Alertmanager Server.
  - host: 172.16.7.147
    listen_host: 0.0.0.0
```

```
# # SSH port of the server.
ssh_port: 22
```

上述配置后， 在集群进行 deploy/scale-out/scale-in/reload 操作时，TiUP 会将 listen_host 字段的内容会添加到 Alertmanager 启动参数的‘–web.listen-address’ 中。

6.8 备份恢复监控告警

本文介绍备份恢复的监控和告警，包括如何部署监控、监控指标及常用告警项。

6.8.1 日志备份监控

日志备份支持功能使用 Prometheus 采集监控指标，目前所有的监控指标都内置在TiKV 中。

6.8.1.1 部署监控

- 通过TiUP 部署的集群，Prometheus 会自动采集相关的监控指标。
- 手动部署的集群，需要参考TiDB 集群监控部署，在 Prometheus 配置文件的 `scrape_configs` 中加入TiKV 相关的job。

6.8.1.2 配置Grafana

- 通过TiUP 部署的集群，Grafana 中内置了 Backup log 的面板。
- 手动部署的集群，需要参考导入 Grafana 面板，将 tikv_details.json 文件上传到Grafana 中。之后在 TiKV-Details Dashboard 中找到 Backup Log 面板即可。

6.8.1.3 监控指标

指标	类型	说明
tikv_log_backup_interal_actor_actinHg_isdtuorgartaiomn		处理内部各种消息事件的耗时。`message :: TaskType`
tikv_log_backup_initial_scan_reasonCounter		触发增量扫的原因统计。主要是 Leader 迁移或者Region Version 变更。`reason :: {"leader-changed", "region-changed", "retry"}`
tikv_log_backup_event_handle_duraHtisotno_gsreacm		处理 KV Event 的耗时。和 `tikv_log_backup_on_event_duration_seconds` 相比，这个指标还包含了一些内部转化消耗的时间。`stage :: {"to_stream_event", "save_to_temp_file"}`
tikv_log_backup_handle_kv_batch	Histogram	由 RaftStore 发送的 KV 对的 Batch 大小统计，统计数据为Region 级别。
tikv_log_backup_initial_scan_disk_reCaodunter		增量扫期间，从硬盘读取的数据量的大小。在 Linux 系统下，这个信息来自于procfs，是实际从block device 读取的数据量的大小；配置项 `initial-scan-rate-limit` 也是施加于这个数值上。

指标	类型	说明
tikv_log_backup_incremental_scan_Hbiystteosgram		增量扫期间，实际产生的 KV 对的大小。因为压缩和读放大的缘故，这个数值和 `tikv_log_backup_initial_scan_disk_read` 不一定相同。
tikv_log_backup_skip_kv_count	Counter	日志备份期间，因为对备份没有帮助而被跳过的Raft Event 数量。
tikv_log_backup_errors	Counter	日志备份期间，遇到的可以重试或可以忽略的错误。`type :: ErrorType`
tikv_log_backup_fatal_errors	Counter	日志备份期间，遇到的不可重试或不可忽略的错误。当该类错误出现的时候，日志备份任务会被暂停。`type :: ErrorType`
tikv_log_backup_heap_memory	Gauge	日志备份期间，增量扫发现的、尚未被消费的事件占用的内存。
tikv_log_backup_on_event_duration_seconds	Histogram	将 KV Event 保存到临时文件各个阶段的耗时。`stage :: {"write_to_tempfile", "syscall_write"}`
tikv_log_backup_store_checkpoint_t	Gsauge	Store 级别的 Checkpoint TS，已经弃用。其含义更加接近于 Store 当前注册的GC Safepoint。`task :: string`
tidb_log_backup_last_checkpoint	Gauge	全局 Checkpoint TS，表示日志备份功能中已经备份的时间点。`task :: string`
tikv_log_backup_flush_duration_sec	Histogram	将本地临时文件移动到外部存储的耗时。`stage :: {"` `'→ generate_metadata", "save_files", "clear_temp_files"}`
tikv_log_backup_flush_file_size	Histogram	备份产生的文件的大小统计。
tikv_log_backup_initial_scan_duratioHnis_tsoegcram		增量扫的整体耗时统计。
tikv_log_backup_skip_retry_observe	Counter	在日志备份过程中，遇到的可忽略错误的统计，即放弃 retry 的原因。`reason :: {"region-absent", "not-leader", "stale-command"}`
tikv_log_backup_initial_scan_operatCioonusnter		增量扫过程中，RocksDB 相关的操作统计。`cf :: {"default", "write", "lock"}, op :: RocksDBOP`
tikv_log_backup_enabled	Counter	日志备份功能是否开启，若值大于 0，表示开启
tikv_log_backup_observed_region	Gauge	被监听的 Region 数量
tikv_log_backup_task_status	Gauge	日志备份任务状态，0-Running 1-Paused 2-Error `task :: string`
tikv_log_backup_pending_initial_sca	Gnauge	尚未执行的增量扫的统计。`stage :: {"queuing", "executing"}`

6.8.1.4 日志备份告警

6.8.1.4.1 配置告警

目前 Point-in-time recovery (PITR) 还未内置告警项，本节介绍如何在 PITR 中配置告警项，以及推荐的告警项规则。

告警规则配置可以参考下面的步骤：

1. 在 Prometheus 所在节点上创建告警规则的配置文件（例如 `pitr.rules.yml`），参考 Prometheus 文档和下列推荐的告警项及配置样例填写告警规则。
2. 在 Prometheus 配置文件中的 `rule_files` 字段填入告警规则文件的路径。
3. 通过向 Prometheus 进程发送 `SIGHUP` 信号（`kill -HUP pid`）或向 `http://prometheus-addr/-/reload` 发送 HTTP POST 请求（使用 HTTP 请求方式前需要在启动 Prometheus 时指定 `--web.enable-lifecycle` 参数）。

以下为推荐的告警项配置：

6.8.1.4.2 LogBackupRunningRPOMoreThan10m

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 60 > 10 and max( ′→ tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task ′→ )== 0`
- 告警级别：warning
- 说明：日志数据超过 10 分钟未持久化到存储中，该配置项主要用于提醒，大部分情况下，不会影响日志备份。

Prometheus 中的配置样例如下：

```
groups:
- name: PiTR
  rules:
  - alert: LogBackupRunningRPOMoreThan10m
    expr: max(time() - tidb_log_backup_last_checkpoint / 262144000) by (task) / 60 > 10 and
        max( ′→ tidb_log_backup_last_checkpoint) by (task) > 0 and
        max(tikv_log_backup_task_status) ′→ by (task) == 0
    labels:
      severity: warning
    annotations:
      summary: RPO of log backup is high
      message: RPO of the log backup task {{ $labels.task }} is more than 10m
```

6.8.1.4.3 LogBackupRunningRPOMoreThan30m

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 60 > 30 and max( ′→ tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task ′→ )== 0`
- 告警级别：critical
- 说明：日志数据超过 30 分钟未持久化到存储中，出现该告警表示极有可能出现异常，可以查看TiKV 日志定位原因。

6.8.1.4.4 LogBackupPausingMoreThan2h

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 3600 > 2 and max( ′→ tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task ′→ )== 1`
- 告警级别：warning
- 说明：日志备份任务处于暂停状态超过 2 小时，该告警主要用于提醒，建议尽早执行 `br log resume` 恢复任务。

6.8.1.4.5 LogBackupPausingMoreThan12h

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 3600 > 12 and ′→ max(tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by ( ′→ task)== 1`
- 告警级别：critical
- 说明：日志备份任务处于暂停状态超过 12 小时，应尽快执行 `br log resume` 恢复任务。任务处于暂停状态时间过长会有数据丢失的风险。

6.8.1.4.6 LogBackupFailed

- 表达式：`max(tikv_log_backup_task_status)by (task)== 2 and max(tidb_log_backup_last_checkpoint ′→ )by (task)> 0`
- 告警级别：critical
- 说明：日志备份任务进入失败状态，需要执行 `br log status` 查看失败原因，如有必要还需进一步查看 TiKV 日志。

6.8.1.4.7 LogBackupGCSafePointExceedsCheckpoint

- 表达式：`min(tidb_log_backup_last_checkpoint)by (instance)- max(tikv_gcworker_autogc_safe_point ′→ )by (instance)< 0`
- 告警级别：critical
- 说明：部分数据在备份前被GC，此时已有部分数据丢失，极有可能对业务产生影响。

7 故障诊断

7.1 故障诊断问题汇总

7.1.1 TiDB 集群问题导图

本篇文档总结了使用TiDB 及其组件时的常见错误。遇到相关错误时，可以通过本文档的问题导图来排查错误原因并进行处理。

7.1.1.1 1. 服务不可用

7.1.1.1.1 1.1 客户端报 `Region is Unavailable` 错误

- 1.1.1 `Region is Unavailable` 一般是由于Region 在一段时间不可用（可能会遇到 `TiKV server is busy`；或者发送给TiKV 的请求由于 `not leader` 或者 `epoch not match` 等原因被打回；又或者请求TiKV 超时等），TiDB 内部会进行 `backoff` 重试。`backoff` 的时间超过一定阈值（默认 20s）后就会报错给客户端。如果 `backoff` 在阈值内，客户端对该错误无感知。
- 1.1.2 多台TiKV 同时内存不足(OOM)，导致Region OOM 期间内没有 Leader，见案例 case-991。

- 1.1.3 TiKV 报 `TiKV server is busy` 错误， 超过 `backoff` 时间， 参考4.3 客户端报 `server is busy` 错误。`TiKV server is busy` 属于内部流控机制，后续可能不计入 `backoff` 时间。
- 1.1.4 多台 TiKV 启动不了，导致Region 没有 Leader。单台物理主机部署多个 TiKV 实例，一个物理机挂掉，由于 label 配置错误导致Region 没有 Leader，见案例 case-228。
- 1.1.5 follower apply 落后，成为 Leader 之后把收到的请求以 `epoch not match` 理由打回，见案例 case-958（TiKV 内部需要优化该机制）。

7.1.1.1.2 1.2 PD 异常导致服务不可用

查看本文档5. PD 问题。

7.1.1.2 2. 延迟明显升高

7.1.1.2.1 2.1 延迟短暂升高

- 2.1.1 TiDB 执行计划不对导致延迟升高，请参考3.3 执行计划不对。
- 2.1.2 PD 出现选举问题或者 OOM 问题，请参考5.2 PD 选举问题和5.3 PD OOM 问题。
- 2.1.3 某些TiKV 大量掉 Leader，请参考4.4 某些TiKV 大量掉 Leader。
- 2.1.4 其他原因，请参考读写延迟增加。

7.1.1.2.2 2.2 Latency 持续升高

- 2.2.1 TiKV 单线程瓶颈
  - 单个TiKV Region 过多，导致单个gRPC 线程成为瓶颈（查看监控：Grafana -> TiKV-details -> Thread CPU/gRPC CPU Per Thread），v3.x 以上版本可以开启 Hibernate Region 特性解决该问题，见案例case-612。
  - v3.0 之前版本Raftstore 单线程或者apply 单线程到达瓶颈（查看监控：Grafana -> TiKV-details -> Thread CPU/raft store CPU 和 Async apply CPU 超过 80%）。可以选择扩容 TiKV（v2.x 版本）实例，或者升级到多线程模型的 v3.x 版本。
- 2.2.2 CPU load 升高。
- 2.2.3 TiKV 写入慢，请参考4.5 TiKV 写入慢。
- 2.2.4 TiDB 执行计划不对，请参考3.3 执行计划不对。
- 2.2.5 其他原因，请参考读写延迟增加。

7.1.1.3 3. TiDB 问题

7.1.1.3.1 3.1 DDL

- 3.1.1 修改 decimal 字段长度时报错 "ERROR 1105 (HY000): unsupported modify decimal column precision"。TiDB 暂时不支持修改 decimal 字段长度。
- 3.1.2 TiDB DDL job 卡住不动/执行很慢（通过 admin show ddl jobs 可以查看 DDL 进度）：
  - 原因 1：与外部组件(PD/TiKV) 的网络问题。
  - 原因 2：早期版本（v3.0.8 之前）TiDB 内部自身负载很重（高并发下可能产生了很多协程）。
  - 原因 3：早期版本（v2.1.15 & v3.0.0-rc1 之前）PD 实例删除TiDB key 无效的问题，会导致每次 DDL 变更都需要等 2 个 lease（很慢）。
  - 其他未知原因，请上报bug。
  - 解决方法：原因 1 需要检查与外部组件的网络问题；原因 2 和 3 已经修复，需要升级到高版本；其他原因，可选择以下兜底方案进行 DDL owner 迁移。
  - DDL owner 迁移方案：
    * 如果与该 TiDB 集群可以网络互通，执行重新进行 owner 选举命令：curl -X POST http://{TiDBIP}:10080/ddl/owner/resign
    * 如果与该TiDB 集群不可以网络互通，需旁路下线，通过tidb-ctl 工具，从PD 集群的etcd 中直接删除 DDL owner，之后也会重新选举：tidb-ctl etcd delowner [LeaseID] [flags] + ownerKey
- 3.1.3 TiDB 日志中报 information schema is changed 的错误：
  - 报错的详细原因以及解决办法参见触发 Information schema is changed 错误的原因。
  - 背景知识：schema version 的增长数量与每个 DDL 变更操作的 schema state 个数一致， 例如 create table 操作会有 1 个版本变更，add column 操作会有 4 个版本变更（详情可以参考 online schema change），所以太多的 column 变更操作会导致 schema version 增长得很快。
- 3.1.4 TiDB 日志中报 information schema is out of date 的错误：
  - 原因 1：执行 DML 的 TiDB 被 graceful kill 后准备退出，且此 DML 对应的事务执行时间超过一个 DDL lease，在事务提交的时候会报此错。
  - 原因 2：TiDB 在执行 DML 时，有一段时间连不上 PD 和 TiKV，导致TiDB 在一个 DDL Lease（默认 45s）内没有加载新的 schema，或者 TiDB 断开与 PD 之间带 keep alive 设置的连接。
  - 原因 3：TiKV 压力大或网络超时，通过监控Grafana -> TiDB 和 TiKV 节点的负载情况来确认是否是该原因。
  - 解决方法：第 1 种原因，在TiDB 启动时手动重试该 DML 即可；第 2 种原因，需要检查TiDB 实例和 PD 及 TiKV 的网络波动情况；第 3 种原因，需要检查TiKV 为什么繁忙，参考4. TiKV 问题。

7.1.1.3.2 3.2 OOM 问题

- 3.2.1 现象
  - 客户端： 客户端收到 TiDB server 报错 ERROR 2013 (HY000): Lost connection to MySQL server during query

- 日志：
  - dmesg -T | grep tidb-server 结果中有事故发生附近时间点的 OOM-killer 的日志。
  - tidb.log 中可以 grep 到事故发生后附近时间的 "Welcome to TiDB" 的日志（即 TiDB server 发生重启）。
  - tidb_stderr.log 中能 grep 到 fatal error: "runtime: out of memory" 或 "cannot allocate memory"。
  - v2.1.8 及其之前的版本，tidb_stderr.log 中能 grep 到 fatal error: stack overflow。
- 监控：TiDB server 实例所在机器可用内存迅速回升

- 3.2.2 定位造成 OOM 的 SQL（目前所有版本都无法完成精准定位，需要在发现 SQL 后再做进一步分析，确认 OOM 是否的确由该 SQL 造成）：
  - >= v3.0.0 的版本，可以在tidb.log 中 grep "expensive_query"，该 log 会记录运行超时、或使用内存超过阈值的 SQL。
  - < v3.0.0 的版本，通过 grep "memory exceeds quota" 定位运行时内存超限的 SQL。

> 注意：
> 单条 SQL 内存阈值的默认值为 1GB，可通过系统变量tidb_mem_quota_query 进行设置。

- 3.2.3 缓解 OOM 问题
  - 通过开启 SWAP 的方式，可以缓解由于大查询使用内存过多而造成的 OOM 问题。但该方法由于存在I/O 开销，会在内存空间不足时对大查询性能造成一定影响。性能回退程度受剩余内存量、读写盘速度影响。
- 3.2.4 OOM 常见原因
  - SQL 中包含join，通过 explain 查看发现该join 选用 HashJoin 算法且 inner 端的表很大。
  - 单条 UPDATE/DELETE 涉及的查询数据量太大，见案例case-882。
  - SQL 中包含 Union 连接的多条子查询，见案例case-1828。

更多 OOM 的排查方法，请参考TiDB OOM 故障排查。

7.1.1.3.3 3.3 执行计划不对

- 3.3.1 现象
  - SQL 相比于之前的执行时间有较大程度变慢，执行计划突然发生改变。如果慢日志中输出了执行计划，可以直接对比执行计划。
  - SQL 执行时间相比于其他数据库（例如 MySQL）有较大差距。可以对比其他数据库执行计划，例如 Join Order 是否不同。
  - 慢日志中 SQL 执行时间 Scan Keys 数目较大。
- 3.3.2 排查执行计划问题

- `explain analyze {SQL}` 在执行时间可以接受的情况下，对比 `explain analyze` 结果中 `count` 和 execution info 中 `rows` 的数目差距。如果在 `TableScan/IndexScan` 行上发现比较大的差距，很大可能是统计信息出问题；如果在其他行上发现较大差距，则也有可能是非统计信息问题。
- `select count(*)` 在执行计划中包含 `join` 等情况下，explain analyze 可能耗时过长；此时可以通过对 `TableScan/IndexScan` 上的条件进行 `select count(*)`，并对比 `explain` 结果中的 `row count` 信息，确定是不是统计信息的问题。

- 3.3.3 缓解问题
  - v3.0 及以上版本可以使用 `SQL Bind` 功能固定执行计划。
  - 更新统计信息。在大致确定问题是由统计信息导致的情况下，先dump 统计信息保留现场。如果是由于统计信息过期导致，例如 `show stats_meta` 中 modify count/row count 大于某个值（例如 0.3）或者表中存在时间列的索引情况下，可以先尝试analyze table 恢复；如果配置了auto analyze，可以查看系统变量 `tidb_auto_analyze_ratio` 是否过大（例如大于 0.3），以及当前时间是否在 `tidb_auto_analyze_start_time` 和 `tidb_auto_analyze_end_time` 范围内。
  - 其他情况，请上报bug。

7.1.1.3.4 3.4 SQL 执行报错

- 3.4.1 客户端报 `ERROR 1265(01000)Data Truncated` 错误。原因是TiDB 内在计算 `Decimal` 类型处理精度的时候，和 MySQL 不兼容。该错误已于 v3.0.10 中修复(#14438)，具体原因如下：

  在 MySQL 内，如果两个大精度 `Decimal` 做除法运算，超出最大小数精度时(30)，会只保留 30 位且不报错。TiDB 在计算结果上，也是这样实现的，但是在内部表示 `Decimal` 的结构体内，有一个表示小数精度的字段，还是保留的真实精度。

  比如 `(0.1^30)/ 10`，TiDB 和 MySQL 的结果都为 0，是正确的，因为精度最多 30；但是 TiDB 内表示精度的那个字段，还是 31；

  多次 `Decimal` 除法计算后，虽然结果正确，但是这个精度可能越来越大，最终超过TiDB 内的另一个阈值 72，此时就会报 `Data Truncated` 的错误；`Decimal` 的乘法计算就不会有这个问题，因为绕过越界，会直接把精度设置为最大精度限制。

  解决方法：可以通过手动加 `Cast(xx as decimal(a, b))` 来绕过这个问题，a 和 b 就是目标的精度。

7.1.1.3.5 3.5 慢查询问题

要定位慢查询，参阅慢查询日志。要处理慢查询，参阅分析慢查询。

7.1.1.3.6 3.6 热点问题

TiDB 作为分布式数据库，内建负载均衡机制，尽可能将业务负载均匀地分布到不同计算或存储节点上，更好地利用上整体系统资源。然而，机制不是万能的，在一些场景下仍会有部分业务负载不能被很好地分散，影响性能，形成单点的过高负载，也称为热点。

TiDB 提供了完整的方案用于排查、解决或规避这类热点。通过均衡负载热点，可以提升整体性能，包括提高 QPS 和降低延迟等。详情参见TiDB 热点问题处理。

7.1.1.3.7 3.7 TiDB 磁盘 I/O 过高

当出现系统响应变慢的时候，如果已经排查了 CPU 的瓶颈、数据事务冲突的瓶颈后，问题仍存在，就需要从 I/O 来入手来辅助判断目前的系统瓶颈点。参考TiDB 磁盘I/O 过高的处理办法了解如何定位和处理TiDB 存储I/O 过高的问题。

7.1.1.3.8 3.8 锁冲突问题

TiDB 支持完整的分布式事务，自 v3.0 版本起，提供乐观事务与悲观事务两种事务模式。要了解如何排查锁相关的问题，以及如何处理乐观和悲观锁冲突的问题，请参考TiDB 锁冲突问题处理。

7.1.1.3.9 3.9 数据索引一致性报错

当执行事务或执行 `ADMIN CHECK [TABLE|INDEX]` 命令时，TiDB 会对数据索引的一致性进行检查。如果检查发现 record key-value 和index key-value 不一致，即存储行数据的键值对和存储其对应索引的键值对之间不一致（例如多索引或缺索引），TiDB 会报数据索引一致性错误，并在日志文件中打印相关错误日志。

要了解更多数据索引一致性报错信息以及如何绕过检查，请参考数据索引一致性报错。

7.1.1.4 4. TiKV 问题

7.1.1.4.1 4.1 TiKV panic 启动不了

- 4.1.1 `sync-log = false`，机器断电之后出现`unexpected raft log index: last_index X < applied_index → Y` 错误。符合预期，需通过 `tikv-ctl` 工具恢复Region。
- 4.1.2 虚拟机部署TiKV，`kill` 虚拟机或物理机断电，出现 `entries[X, Y] is unavailable from storage` 错误。符合预期，虚拟机的 fsync 不可靠，需通过 `tikv-ctl` 工具恢复Region。
- 4.1.3 其他原因（非预期，需报bug）。

7.1.1.4.2 4.2 TiKV OOM

- 4.2.1 `block-cache` 配置太大导致 OOM：
  - 在监控Grafana -> TiKV-details 选中对应的instance 后，查看RocksDB 的 `block cache size` 监控来确认是否是该问题。
  - 同时，请检查 `[storage.block-cache] capacity = # "1GB"` 参数是否设置合理，默认情况下 TiKV 的 `block-cache` 设置为机器总内存的 45%；在 container 部署时，需要显式指定该参数，因为TiKV 获取的是物理机的内存，可能会超出container 的内存限制。
- 4.2.2 Coprocessor 收到大量大查询，返回的数据量太大，gRPC 的发送速度跟不上 Coprocessor 往外输出数据的速度，导致 OOM：
  - 可以通过检查监控：Grafana -> TiKV-details -> coprocessor overview 的 `response size` 是否超过 `network → outbound` 流量来确认是否属于这种情况。
- 4.2.3 其他部分占用太多内存（非预期，需报bug）。

7.1.1.4.3 4.3 客户端报 server is busy 错误

通过查看监控：Grafana -> TiKV -> errors 确认具体busy 原因。server is busy 是 TiKV 自身的流控机制，TiKV 通过这种方式告知 tidb/ti-client 当前TiKV 的压力过大，稍后再尝试。

- 4.3.1 TiKV RocksDB 出现 write stall。一个 TiKV 包含两个 RocksDB 实例，一个用于存储 Raft 日志，位于 data/raft。另一个用于存储真正的数据，位于 data/db。通过 grep "Stalling" RocksDB 日志查看 stall 的具体原因，RocksDB 日志是 LOG 开头的文件，LOG 为当前日志。
  - level0 sst 太多导致 stall，可以添加参数 [rocksdb] max-sub-compactions = 2（或者 3），加快 level0 sst 往下 compact 的速度。该参数的意思是将从 level0 到 level1 的 compaction 任务最多切成 max-sub-compactions 个子任务交给多线程并发执行，见案例case-815。
  - pending compaction bytes 太多导致 stall，磁盘 I/O 能力在业务高峰跟不上写入，可以通过调大对应 Column Family (CF) 的 soft-pending-compaction-bytes-limit 和 hard-pending-compaction-bytes-limit 参数来缓解：
    * 如果pending compaction bytes 达到该阈值，RocksDB 会放慢写入速度。默认值64GB，[rocksdb.defaultcf] soft-pending-compaction-bytes-limit = "128GB"。
    * 如果 pending compaction bytes 达到该阈值，RocksDB 会 stop 写入，通常不太可能触发该情况，因为在达到 soft-pending-compaction-bytes-limit 的阈值之后会放慢写入速度。默认值 256GB，hard-pending-compaction-bytes-limit = "512GB"。
    * 如果磁盘IO 能力持续跟不上写入，建议扩容。如果磁盘的吞吐达到了上限（例如 SATA SSD 的吞吐相对 NVME SSD 会低很多）导致 write stall，但是 CPU 资源又比较充足，可以尝试采用压缩率更高的压缩算法来缓解磁盘的压力，用 CPU 资源换磁盘资源。
    * 比如default cf compaction 压力比较大，调整参数 [rocksdb.defaultcf] compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"] 改成 compression-per-level = ["no", "no", "zstd", "zstd", "zstd", "zstd", "zstd"]。
  - memtable 太多导致 stall。该问题一般发生在瞬间写入量比较大，并且 memtable flush 到磁盘的速度比较慢的情况下。如果磁盘写入速度不能改善，并且只有业务峰值会出现这种情况，可以通过调大对应 CF 的 max-write-buffer-number 来缓解：
    * 例如 [rocksdb.defaultcf] max-write-buffer-number = 8 （默认值 5），同时请求注意在高峰期可能会占用更多的内存，因为可能存在于内存中的memtable 会更多。
- 4.3.2 scheduler too busy
  - 写入冲突严重，latch wait duration 比较高，查看监控：Grafana -> TiKV-details -> scheduler prewrite 或者 scheduler commit 的 latch wait duration。scheduler 写入任务堆积，导致超过了[storage] scheduler -pending-write-threshold = "100MB" 设置的阈值。可通过查看 MVCC_CONFLICT_COUNTER 对应的 metric 来确认是否属于该情况。
  - 写入慢导致写入堆积， 该 TiKV 正在写入的数据超过了 [storage] scheduler-pending-write-threshold = "100MB" 设置的阈值。请参考4.5 TiKV 写入慢。
- 4.3.3 raftstore is busy，主要是消息的处理速度没有跟上接收消息的速度。短时间的 channel full 不会影响服务，长时间持续出现该错误可能会导致 Leader 切换走。
  - append log 遇到了 stall，参考4.3.1 客户端报 server is busy 错误。

  - append log duration 比较高，导致处理消息不及时，可以参考4.5 TiKV 写入慢分析为什么 append log duration 比较高。
  - 瞬间收到大量消息（查看 TiKV Raft messages 面板），Raftstore 没处理过来，通常情况下短时间的 channel full 不会影响服务。

- 4.3.4 TiKV Coprocessor 排队，任务堆积超过了 Coprocessor 线程数 * readpool.coprocessor.max-tasks-per-worker-[normal|low|high]。大量大查询导致 Coprocessor 出现了堆积情况，需要确认是否由于执行计划变化而导致了大量扫表操作，请参考3.3 执行计划不对。

7.1.1.4.4 4.4 某些TiKV 大量掉 Leader

- 4.4.1 TiKV 重启，导致重新选举。
  - TiKV panic 之后又被 systemd 重新拉起正常运行，可以通过查看TiKV 的日志来确认是否有 panic，这种情况属于非预期，需报bug。
  - 被第三者 stop/kill，被 systemd 重新拉起。查看 dmesg 和 TiKV log 确认原因。
  - TiKV 发生 OOM 导致重启了，参考4.2 TiKV OOM 问题。
  - 动态调整 THP 导致hung 住，见案例case-500。
- 4.4.2 查看监控：Grafana -> TiKV-details -> errors 面板 server is busy，看到 TiKV RocksDB 出现 write stall 导致发生重新选举，请参考4.3.1。
- 4.4.3 网络隔离导致重新选举。

7.1.1.4.5 4.5 TiKV 写入慢

- 4.5.1 通过查看TiKV gRPC 的 prewrite/commit/raw-put（仅限 raw kv 集群）duration 确认确实是TiKV 写入慢了。通常情况下可以按照performance-map 来定位到底哪个阶段慢了，下面列出几种常见的情况。
- 4.5.2 scheduler CPU 繁忙（仅限transaction kv）。prewrite/commit 的scheduler command duration 比scheduler latch wait duration + storage async write duration 更长，并且 scheduler worker CPU 比较高，例如超过 scheduler-worker-pool-size * 100% 的 80%，并且或者整个机器的 CPU 资源比较紧张。如果写入量很大，确认下是否 [storage] scheduler-worker-pool-size 配置得太小。其他情况，需报bug。
- 4.5.3 Append log 慢。TiKV Grafana 的 Raft IO/append log duration 比较高，通常情况下是由于写盘慢了，可以检查RocksDB - Raft 的 WAL Sync Duration max 值来确认，否则可能需要报bug。
- 4.5.4 Raftstore 线程繁忙。TiKV Grafana 的 Raft Propose/propose wait duration 明显高于append log duration。请查看以下情况：
  - [raftstore] store-pool-size 配置是否过小（该值建议在 [1,5] 之间，不建议太大）。
  - 机器的 CPU 是不是不够。
- 4.5.5 apply 慢了。TiKV Grafana 的 Raft IO/apply log duration 比较高，通常会伴随着 Raft Propose/apply wait duration 比较高。可能是以下原因引起的：
  - [raftstore] apply-pool-size 配置过小（建议在 [1, 5] 之间，不建议太大），Thread CPU/apply cpu 比较高；

  - 机器的 CPU 资源不够了；
  - Region 写入热点问题，单个 apply 线程 CPU 使用率比较高（通过修改 Grafana 表达式，加上 by (instance, name) 来看各个线程的 CPU 使用情况），暂时对于单个 Region 的热点写入没有很好的方式，最近在优化该场景；
  - 写 RocksDB 比较慢，RocksDB kv/max write duration 比较高（单个 Raft log 可能包含很多个 kv，写 RocksDB 的时候会把 128 个 kv 放在一个 write batch 写入到RocksDB，所以一次apply log 可能涉及到多次 RocksDB 的 write）；
  - 其他情况，需报bug。

- 4.5.6 Raft commit log 慢了。

  - TiKV Grafana 的 Raft IO/commit log duration 比较高（4.x 版本的Grafana 才有该metric）。每个 Region 对应一个独立的 Raft group，Raft 本身是有流控机制的，类似 TCP 的滑动窗口机制，通过参数 [raftstore] raft-max-inflight-msgs = 256 来控制滑动窗口的大小， 如果有热点写入并且 commit log duration 比较高可以适度调大该参数，比如 1024。

- 4.5.7 其他情况，请参考 Performance Map 上的写入路径来分析。

7.1.1.5 5. PD 问题

7.1.1.5.1 5.1 PD 调度问题

- 5.1.1 merge 问题：

  - 跨表空Region 无法merge，需要修改 TiKV 的 [coprocessor] split-region-on-table = false 参数来解决，4.x 版本该参数默认为 false，见案例 case-896。
  - Region merge 慢，可检查监控Grafana -> PD -> operator 面板是否有merge 的 operator 产生，可以适当调大 merge-schedule-limit 参数来加速merge。

- 5.1.2 补副本/上下线问题：

  - TiKV 磁盘使用80% 容量，PD 不会进行补副本操作，miss peer 数量上升，需要扩容TiKV，见案例case-801。
  - 下线TiKV，有Region 长时间迁移不走。v3.0.4 版本已经修复改问题，见#5526 和案例case-870。

- 5.1.3 Balance 问题：

  - Leader/Region count 分布不均，见案例 case-394, case-759。主要原因是 balance 是依赖Region/leader 的 size 去调度的，所以可能会造成count 数量的不均衡，v4.0 新增了一个参数 [leader-schedule-policy]，可以调整 Leader 的调度策略，根据“count” 或者是“size” 进行调度。

7.1.1.5.2 5.2 PD 选举问题

- 5.2.1 PD 发生 Leader 切换：

  - 磁盘问题，PD 所在的节点I/O 被打满，排查是否有其他I/O 高的组件与 PD 混部以及盘的健康情况，可通过监控Grafana -> disk performance -> latency 和 load 等指标进行验证，必要时可以使用 fio 工具对盘进行检测，见案例case-292。

- 网络问题，PD 日志中有 `lost the TCP streaming connection`，排查 PD 之间网络是否有问题，可通过监控Grafana -> PD -> etcd 的 `round trip` 来验证，见案例case-177。
- 系统load高，日志中能看到 `server is likely overloaded`，见案例 case-214。

- 5.2.2 PD 选不出 Leader 或者选举慢：
  - 选不出 Leader，PD 日志中有 `lease is not expired`，见 #10355。v3.0.x 版本和 v2.1.19 版本已修复该问题，见案例case-875。
  - 选举慢，Region 加载时间长。从 PD 日志中 `grep "regions cost"`（例如日志中可能是 “load 460927 regions cost 11.77099s”），如果出现秒级，则说明较慢，v3.0 版本可开启 Region storage（设置 `use-region-storage` 为 `true`），该特性能极大缩短加载 Region 的时间，见案例case-429。

- 5.2.3 TiDB 执行 SQL 时报 PD timeout：
  - PD 没 Leader 或者有切换，参考5.2.1 PD 选举问题和5.2.2 PD 选举问题。
  - 网络问题，排查网络相关情况。通过监控 Grafana -> blackbox_exporter -> ping latency 确定 TiDB 到 PD Leader 的网络是否正常。
  - PD panic，需报bug。
  - PD OOM，参考5.3 PD OOM 问题。
  - 其他原因，通过`curl http://127.0.0.1:2379/debug/pprof/goroutine?debug=2` 抓goroutine，报bug。

- 5.2.4 其他问题
  - PD 报 FATAL 错误，日志中有 `range failed to find revision pair`，v3.0.8 已经修复该问题，见 #2040。详情参考案例case-947。
  - 其他原因，需报bug。

7.1.1.5.3 5.3 PD OOM

- 5.3.1 使用 `/api/v1/regions` 接口时Region 数量过多，可能会导致 PD OOM，在 v3.0.8 版本中修复，见 #1986。
- 5.3.2 滚动升级的时候 PD OOM，gRPC 消息大小没限制，监控可看到TCP InSegs 较大，在 v3.0.6 版本中修复，见 #1952。

7.1.1.5.4 5.4 Grafana 显示问题

- 5.4.1 监控Grafana -> PD -> cluster -> role 显示 follower，Grafana 表达式问题，在 v3.0.8 版本修复。

7.1.1.6 6. 生态Tools 问题

7.1.1.6.1 6.1 TiDB Binlog 问题

- 6.1.1 TiDB Binlog 是将 TiDB 的修改同步给下游 TiDB 或者 MySQL 的工具，见 TiDB Binlog on GitHub。
- 6.1.2 Pump/Drainer Status 中 Update Time 正常更新，日志中也没有异常，但下游没有数据写入。
  - TiDB 配置中没有开启 binlog，需要修改 TiDB 配置 `[binlog]`。
- 6.1.3 Drainer 中的 sarama 报 EOF 错误。
  - Drainer 使用的 Kafka 客户端版本和 Kafka 版本不匹配，需要修改配置 `[syncer.to] kafka-version` 来解决。
- 6.1.4 Drainer 写 Kafka 失败然后 panic，Kafka 报 `Message was too large` 错误。
  - binlog 数据太大，造成写 Kafka 的单条消息太大，需要修改 Kafka 的下列配置来解决：

```
message.max.bytes=1073741824
replica.fetch.max.bytes=1073741824
fetch.message.max.bytes=1073741824
```

    见案例 case-789。

- 6.1.5 上下游数据不一致
  - 部分 TiDB 节点没有开启 binlog。v3.0.6 及之后的版本可以通过访问 http://127.0.0.1:10080/info/all 接口可以检查所有节点的 binlog 状态。之前的版本可以通过查看配置文件来确认是否开启了 binlog。
  - 部分 TiDB 节点进入 ignore binlog 状态。v3.0.6 及之后的版本可以通过访问 http://127.0.0.1:10080/info/all 接口可以检查所有节点的 binlog 状态。之前的版本需要看 TiDB 的日志中是否有 ignore binlog 关键字来确认是该问题。
  - 上下游 Timestamp 列的值不一致：
    * 时区问题，需要确保 Drainer 和上下游数据库时区一致，Drainer 通过 `/etc/localtime` 获取时区，不支持 TZ 环境变量，见案例 case-826。
    * TiDB 中 Timestamp 的默认值为 `null`，MySQL 5.7 中 Timestamp 默认值为当前时间（MySQL 8 无此问题），因此当上游写入 `null` 的 Timestamp 且下游是 MySQL 5.7 时，Timestamp 列数据不一致。在开启 binlog 前，在上游执行 `set @@global.explicit_defaults_for_timestamp=on;` 可解决此问题。
  - 其他情况需报 bug。
- 6.1.6 同步慢
  - 下游是 TiDB/MySQL，上游频繁进行 DDL 操作，见案例 case-1023。
  - 下游是 TiDB/MySQL，需要同步的表中存在没有主键且没有唯一索引的表，这种情况会导致 binlog 性能下降，建议加主键或唯一索引。
  - 下游输出到文件，检查目标磁盘/网络盘是否慢。
  - 其他情况需报 bug。
- 6.1.7 Pump 无法写 binlog，报 `no space left on device` 错误。

  - 本地磁盘空间不足，Pump 无法正常写binlog 数据。需要清理磁盘空间，然后重启 Pump。

- 6.1.8 Pump 启动时报错 `fail to notify all living drainer`。

  - Pump 启动时需要通知所有 Online 状态的 Drainer，如果通知失败则会打印该错误日志。
  - 可以使用binlogctl 工具查看所有 Drainer 的状态是否有异常，保证 Online 状态的 Drainer 都在正常工作。如果某个 Drainer 的状态和实际运行情况不一致，则使用binlogctl 修改状态，然后再重启 Pump。见案例fail-to-notify-all-living-drainer。

- 6.1.9 Drainer 报错 `gen update sqls failed: table xxx: row data is corruption []`。

  - 触发条件：上游做 Drop Column DDL 的时候同时在做这张表的 DML。已经在 v3.0.6 修复，见case-820。

- 6.1.10 Drainer 同步卡住，进程活跃但checkpoint 不更新。

  - 已知bug 在 v3.0.4 修复，见案例case-741。

- 6.1.11 任何组件 `panic`。

  - 需报bug。

7.1.1.6.2 6.2 DM 问题

- 6.2.1 TiDB Data Migration (DM) 是能将 MySQL/MariaDB 的数据迁移到TiDB 的迁移工具，详情见 DM on GitHub。

- 6.2.2 执行 `query-status` 或查看日志时出现 `Access denied for user 'root'@'172.31.43.27' (using password: YES)`。

  - 在所有 DM 配置文件中，数据库相关的密码都必须使用经dmctl 加密后的密文（若数据库密码为空，则无需加密）。在 v1.0.6 及以后的版本可使用明文密码。
  - 在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动同步任务过程中，DM 会自动进行相应权限的检查。
  - 同一套 DM 集群，混合部署不同版本的 DM-worker/DM-master/dmctl，见案例 AskTUG-1049。

- 6.2.3 DM 同步任务中断并包含 `driver: bad connection` 错误。

  - 发生 `driver: bad connection` 错误时，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到TiDB。
    * 1.0.0 GA 之前的版本，DM 发生该类型错误时，需要先使用 `stop-task` 命令停止任务后再使用 `start-task` 命令重启任务。
    * 1.0.0 GA 版本，增加对此类错误的自动重试机制，见#265。

- 6.2.4 同步任务中断并包含 `invalid connection` 错误。

  - 发生 `invalid connection` 错误时，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了TiDB。由于 DM 中存在同步任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 `query-status` 或 `query-error` 查询当前错误）：

    * 如果错误中仅包含 `invalid connection` 类型的错误，且当前处于增量复制阶段，则 DM 会自动进行重试。
    * 如果 DM 由于版本问题（v1.0.0-rc.1 后引入自动重试）等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

- 6.2.5 Relay 处理单元报错 `event from * in * diff from passed-in event *` 或同步任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误。

    - 在 DM 进行 relay log 拉取与增量同步过程中，如果遇到了上游超过 4 GB 的 binlog 文件，就可能出现这两个错误。原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存同步的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4 GB 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。
        * 对于 relay 处理单元，可通过官网步骤进行手动处理。
        * 对于 binlog replication 处理单元，可通过官网步骤进行手动处理。

- 6.2.6 DM 同步中断，日志报错 `ERROR 1236 (HY000)The slave is connecting using CHANGE MASTER TO MASTER_AUTO_POSITION = 1, but the master has purged binary logs containing GTIDs that the slave requires.`。

    - 检查 master 的 binlog 是否被 purge。
    - 检查 relay.meta 中记录的位点信息。
        * relay.meta 中记录空的 GTID 信息，DM-worker 进程在退出时、以及定时(30s) 会把内存中的 GTID 信息保存到 relay.meta 中，在没有获取到上游 GTID 信息的情况下，把空的 GTID 信息保存到了 relay.meta 中。见案例 case-772。
        * relay.meta 中记录的 binlog event 不完整触发 recover 流程后记录错误的 GTID 信息，该问题可能会在 1.0.2 之前的版本遇到，已在 1.0.2 版本修复。

- 6.2.7 DM 同步报错 `Error 1366: incorrect utf8 value eda0bdedb29d(\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd)`。

    - 该值 MySQL 8.0 和 TiDB 都不能写入成功，但是 MySQL 5.7 可以写入成功。可以开启 TiDB 动态参数 `tidb_skip_utf8_check` 参数，跳过数据格式检查。

7.1.1.6.3 6.3 TiDB Lightning 问题

- 6.3.1 TiDB Lightning 是快速的全量数据导入工具，见 TiDB Lightning on GitHub。

- 6.3.2 导入速度太慢。

    - `region-concurrency` 设定太高，线程间争用资源反而减低了效率。排查方法如下：
        * 从日志的开头搜寻 `region-concurrency` 能知道 TiDB Lightning 读到的参数是多少；
        * 如果 TiDB Lightning 与其他服务（如 TiKV Importer）共用一台服务器，必需手动将 `region-concurrency` 设为该服务器 CPU 数量的 75%；
        * 如果 CPU 设有限额（例如从 Kubernetes 指定的上限），TiDB Lightning 可能无法自动判断出来，此时亦需要手动调整 `region-concurrency`。

- 表结构太复杂。每条索引都会额外增加 KV 对，如果有 N 条索引，实际导入的大小就差不多是 Mydumper 文件的 N+1 倍。如果索引不太重要，可以考虑先从 schema 去掉，待导入完成后再使用 CREATE INDEX 加回去。
- TiDB Lightning 版本太旧。尝试使用最新的版本，可能会有改善。

- 6.3.3 `checksum failed: checksum mismatched remote vs local`
  - 原因 1：这张表可能本身已有数据，影响最终结果。
  - 原因 2：如果目标数据库的校验和全是 0，表示没有发生任何导入，有可能是集群太忙无法接收任何数据。
  - 原因 3：如果数据源是由机器生成而不是从 Mydumper 备份的，需确保数据符合表的限制。例如：
    * 自增(AUTO_INCREMENT)的列需要为正数，不能为 0。
    * 单一键和主键(UNIQUE and PRIMARY KEYs)不能有重复的值。
  - 解决办法：参考官网步骤处理。

- 6.3.4 `Checkpoint for … has invalid status:(`错误码`)`
  - 原因：断点续传已启用。TiDB Lightning 或 TiKV Importer 之前发生了异常退出。为了防止数据意外损坏，TiDB Lightning 在错误解决以前不会启动。错误码是小于 25 的整数，可能的取值是 0、3、6、9、12、14、15、17、18、20、21。整数越大，表示异常退出所发生的步骤在导入流程中越晚。
  - 解决办法：参考官网步骤处理。

- 6.3.5 `ResourceTemporarilyUnavailable("Too many open engines …: 8")`
  - 原因：并行打开的引擎文件(engine files)超出 tikv-importer 里的限制。这可能由配置错误引起。即使配置没问题，如果tidb-lightning 曾经异常退出，也有可能令引擎文件残留在打开的状态，占据可用的数量。
  - 解决办法：参考官网步骤处理。

- 6.3.6 `cannot guess encoding for input file, please convert to UTF-8 manually`
  - 原因：TiDB Lightning 只支持 UTF-8 和 GB-18030 编码的表架构。此错误代表数据源不是这里任一个编码。也有可能是文件中混合了不同的编码，例如在不同的环境运行过 ALTER TABLE，使表架构同时出现 UTF-8 和 GB-18030 的字符。
  - 解决办法：参考官网步骤处理。

- 6.3.7 `[sql2kv] sql encode error = [types:1292]invalid time format: '{1970 1 1 0 45 0 0}'`
  - 原因：一个timestamp 类型的时间戳记录了不存在的时间值。时间值不存在是由于夏令时切换或超出支持的范围（1970 年 1 月 1 日至 2038 年 1 月 19 日）。
  - 解决办法：参考官网步骤处理。

7.1.1.7 7. 常见日志分析

7.1.1.7.1 7.1 TiDB

- 7.1.1 GC life time is shorter than transaction duration。事务执行时间太长，超过了 GC lifetime（默认为 10 分钟），可以通过修改系统变量tidb_gc_life_time 来延长life time，通常情况下不建议修改，因为延长时限可能导致大量老版本数据的堆积（如果有大量 UPDATE 和 DELETE 语句）。
- 7.1.2 txn takes too much time。事务太长时间（超过 590s）没有提交，准备提交的时候报该错误。可以通过调大 [tikv-client] max-txn-time-use = 590 参数，以及调大 GC life time 来绕过该问题（如果确实有这个需求）。通常情况下，建议看看业务是否真的需要执行这么长时间的事务。
- 7.1.3 coprocessor.go 报 request outdated。发往 TiKV 的 Coprocessor 请求在TiKV 端排队时间超过了 60s，直接返回该错误。需要排查TiKV Coprocessor 为什么排队这么严重。
- 7.1.4 region_cache.go 大量报 switch region peer to next due to send request fail 且 error 信息是 context deadline exceeded。请求 TiKV 超时触发region cache 切换请求到其他节点，可以对日志中的 addr 字段继续 grep "<addr> cancelled"，根据 grep 结果：
  - send request is cancelled。请求发送阶段超时，可以排查监控Grafana -> TiDB -> Batch Client/Pending Request Count by TiKV 是否大于 128，确定是否因发送远超 KV 处理能力导致发送堆积。如果 Pending Request 不多，需要排查日志确认是否因为对应 KV 有运维变更，导致短暂报出；否则非预期，需报bug。
  - wait response is cancelled。请求发送到 TiKV 后超时未收到TiKV 响应。需要排查对应地址TiKV 的响应时间和对应Region 在当时的 PD 和 KV 日志，确定为什么 KV 未及时响应。
- 7.1.5 distsql.go 报 inconsistent index。数据索引疑似发生不一致，首先对报错的信息中 index 所在表执行 admin check table <TableName> 命令，如果检查失败，则先通过以下命令禁用GC，然后报 bug。

```
SET GLOBAL tidb_gc_enable = 0;
```

7.1.1.7.2 7.2 TiKV

- 7.2.1 key is locked 读写冲突，读请求碰到还未提交的数据，需要等待其提交之后才能读。少量这个错误对业务无影响，大量出现这个错误说明业务读写冲突比较严重。
- 7.2.2 write conflict 乐观事务中的写写冲突，同时多个事务对相同的 key 进行修改，只有一个事务会成功，其他事务会自动重取timestamp 然后进行重试，不影响业务。如果业务冲突很严重可能会导致重试多次之后事务失败，这种情况下建议使用悲观锁。报错以及解决方法详情，参考乐观事务模型下写写冲突问题排查。
- 7.2.3 TxnLockNotFound 事务提交太慢，过了 TTL (Time To Live) 时间之后被其他事务回滚了，该事务会自动重试，通常情况下对业务无感知。对于 0.25 MB 以内的小事务，TTL 默认时间为 3 秒。详情参见锁被清除 (LockNotFound) 错误。
- 7.2.4 PessimisticLockNotFound 类似 TxnLockNotFound，悲观事务提交太慢被其他事务回滚了。
- 7.2.5 stale_epoch 请求的 epoch 太旧了，TiDB 会更新路由之后再重新发送请求，业务无感知。epoch 在 Region 发生 split/merge 以及迁移副本的时候会变化。

- 7.2.6 `peer is not leader` 请求发到了非 Leader 的副本上，TiDB 会根据该错误更新本地路由（如果错误 response 里携带了最新 Leader 是哪个副本这一信息），并且重新发送请求到最新 Leader，一般情况下业务无感知。在 v3.0 后 TiDB 在原 Leader 请求失败时会尝试其他peer，也会导致TiKV 频繁出现 `not leader`日志，可以通过查看TiDB 对应Region 的 `switch region peer to next due to send request fail` 日志，排查发送失败根本原因，参考7.1.4 TiDB。另外也可能是由于其他原因导致一些Region 一直没有 Leader，请参考4.4 某些TiKV 大量掉 Leader。

7.1.2 TiDB 集群故障诊断

当试用TiDB 遇到问题时，请先参考本篇文档。如果问题未解决，请按文档要求收集必要的信息通过Github 提供给TiDB 开发者。

7.1.2.1 如何给TiDB 开发者报告错误

当使用TiDB 遇到问题并且通过后面所列信息无法解决时，请收集以下信息并创建新Issue:

- 具体的出错信息以及正在执行的操作
- 当前所有组件的状态
- 出问题组件log 中的 error/fatal/panic 信息
- 机器配置以及部署拓扑
- dmesg 中 TiDB 组件相关的问题

7.1.2.2 数据库连接不上

首先请确认集群的各项服务是否已经启动，包括tidb-server、 pd-server、 tikv-server。 请用ps 命令查看所有进程是否在。如果某个组件的进程已经不在了，请参考对应的章节排查错误。

如果所有的进程都在，请查看tidb-server 的日志，看是否有报错？常见的错误包括：

- InformationSchema is out of date

  无法连接tikv-server，请检查pd-server 以及tikv-server 的状态和日志。

- panic

  程序有错误，请将具体的panic log 提供给TiDB 开发者。

  如果是清空数据并重新部署服务，请确认以下信息：

- pd-server、 tikv-server 数据都已清空

  tikv-server 存储具体的数据，pd-server 存储 tikv-server 中数据的元信息。如果只清空 pd-server 或只清空 tikv-server 的数据，会导致两边数据不匹配。

- 清空pd-server 和 tikv-server 的数据并重启后，也需要重启tidb-server

  集群ID 是由pd-server 在集群初始化时随机分配，所以重新部署集群后，集群ID 会发生变化。tidb-server 业务需要重启以获取新的集群ID。

7.1.2.3 tidb-server 启动报错

tidb-server 无法启动的常见情况包括：

- 启动参数错误

  请参考TiDB 命令行参数。

- 端口被占用：`lsof -i:port`

  请确保tidb-server 启动所需要的端口未被占用。

- 无法连接pd-server

  首先检查pd-server 的进程状态和日志，确保pd-server 成功启动，对应端口已打开：`lsof -i:port`。

  若 pd-server 正常，则需要检查tidb-server 机器和pd-server 对应端口之间的连通性，确保网段连通且对应服务端口已添加到防火墙白名单中，可通过nc 或 curl 工具检查。

  例如，假设tidb 服务位于 `192.168.1.100`，无法连接的 pd 位于 `192.168.1.101`，且 2379 为其client port，则可以在 tidb 机器上执行 `nc -v -z 192.168.1.101 2379`，测试是否可以访问端口。或使用 `curl -v ′→ 192.168.1.101:2379/pd/api/v1/leader` 直接检查pd 是否正常服务。

7.1.2.4 tikv-server 启动报错

- 启动参数错误

  请参考TiKV 启动参数文档。

- 端口被占用：`lsof -i:port`

  请确保tikv-server 启动所需要的端口未被占用：`lsof -i:port`。

- 无法连接pd-server

  首先检查pd-server 的进程状态和日志。确保pd-server 成功启动，对应端口已打开：`lsof -i:port`。

  若 pd-server 正常，则需要检查tikv-server 机器和pd-server 对应端口之间的连通性，确保网段连通且对应服务端口已添加到防火墙白名单中，可通过nc 或 curl 工具检查。具体命令参考上一节。

- 文件被占用

  不要在一个数据库文件目录上打开两个tikv。

7.1.2.5 pd-server 启动报错

- 启动参数错误

  请参考 PD 命令行参数文档。

- 端口被占用：`lsof -i:port`

  请确保pd-server 启动所需要的端口未被占用：`lsof -i:port`。

7.1.2.6 TiDB/TiKV/PD 进程异常退出

- 进程是否是启动在前台
  当前终端退出给其所有子进程发送 HUP 信号，从而导致进程退出。
- 是否是在命令行用过 nohup+& 方式直接运行
  这样依然可能导致进程因终端连接突然中断，作为终端 SHELL 的子进程被杀掉。
  推荐将启动命令写在脚本中，通过脚本运行（相当于二次 fork 启动）。

7.1.2.7 TiKV 进程异常重启

- 检查 dmesg 或者 syslog 里面是否有 OOM 信息
  如果有 OOM 信息并且杀掉的进程为TiKV，请减少TiKV 的 RocksDB 的各个 CF 的 `block-cache-size` 值。
- 检查 TiKV 日志是否有panic 的 log
  提交 Issue 并附上panic 的 log。

7.1.2.8 TiDB panic

请提供panic 的 log。

7.1.2.9 连接被拒绝

- 请确保操作系统的网络参数正确，包括但不限于
  - 连接字符串中的端口和tidb-server 启动的端口需要一致
  - 请保证防火墙的配置正确

7.1.2.10 Too many open files

在启动进程之前，请确保 `ulimit -n` 的结果足够大，推荐设为unlimited 或者是大于 1000000。

7.1.2.11 数据库访问超时，系统负载高

首先检查SLOW-QUERY 日志，判断是否是因为某条 SQL 语句导致。如果未能解决，请提供如下信息：

- 部署的拓扑结构
  - tidb-server/pd-server/tikv-server 部署了几个实例
  - 这些实例在机器上是如何分布的
- 机器的硬件配置
  - CPU 核数
  - 内存大小
  - 硬盘类型（SSD 还是机械硬盘）
  - 是实体机还是虚拟机

- 机器上除了TiDB 集群之外是否还有其他服务
- pd-server 和 tikv-server 是否分开部署
- 目前正在进行什么操作
- 用 `top -H` 命令查看当前占用 CPU 的线程名
- 最近一段时间的网络/IO 监控数据是否有异常

7.1.3 TiFlash 常见问题

本文介绍了一些TiFlash 常见问题、原因及解决办法。

7.1.3.1 TiFlash 未能正常启动

该问题可能由多个因素构成，可以通过以下步骤依次排查：

1. 检查系统环境是否是 CentOS8。

   CentOS8 中缺少 `libnsl.so` 系统库，可以通过手动安装的方式解决：

   ```
   dnf install libnsl
   ```

2. 检查系统的 `ulimit` 参数设置。

   ```
   ulimit -n 1000000
   ```

3. 使用 PD Control 工具检查在该节点（相同IP 和 Port）是否有之前未成功下线的 TiFlash 实例，并将它们强制下线。（下线步骤参考手动缩容 TiFlash 节点）

如果遇到上述方法无法解决的问题，可以打包TiFlash 的log 文件夹，并在AskTUG 社区中提问。

7.1.3.2 TiFlash 副本始终处于不可用状态

该问题一般由于配置错误或者环境问题导致TiFlash 处于异常状态，可以先通过以下步骤定位问题组件：

1. 使用pd-ctl 检查 PD 的 Placement Rules 功能是否开启：

   ```
   echo 'config show replication' | /path/to/pd-ctl -u http://${pd-ip}:${pd-port}
   ```

   - 如果返回 `true`，进入下一步。
   - 如果返回 `false`，你需要先开启 Placement Rules 特性后再进入下一步。

2. 通过TiFlash-Summary 监控面板下的 UpTime 检查操作系统中TiFlash 进程是否正常。

3. 通过pd-ctl 查看TiFlash proxy 状态是否正常：

   ```
   echo "store" | /path/to/pd-ctl -u http://${pd-ip}:${pd-port}
   ```

   store.labels 中含有 `{"key": "engine", "value": "tiflash"}` 信息的为TiFlash proxy。

4. 查看 pd buddy 是否正常打印日志（日志路径的对应配置项 [flash.flash_cluster] log 设置的值，默认为TiFlash 配置文件配置的tmp 目录下 ）。
5. 检查配置的副本数是否小于等于集群TiKV 节点数。若配置的副本数超过 TiKV 节点数，则 PD 不会向TiFlash 同步数据；

```
echo 'config placement-rules show' | /path/to/pd-ctl -u http://${pd-ip}:${pd-port}
```

再确认“default: count”参数值。

> 注意：
> 开启 Placement Rules 后，原先的 `max-replicas` 及 `location-labels` 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

6. 检查TiFlash 节点对应 store 所在机器剩余的磁盘空间是否充足。默认情况下当磁盘剩余空间小于该 store 的 capacity 的 20%（通过 low-space-ratio 参数控制）时，PD 不会向TiFlash 调度数据。

7.1.3.3 部分查询返回Region Unavailable 的错误

如果在TiFlash 上的负载压力过大，会导致TiFlash 数据同步落后，部分查询可能会返回 `Region Unavailable` 的错误。

在这种情况下，可以增加TiFlash 节点分担负载压力。

7.1.3.4 数据文件损坏

可依照如下步骤进行处理：

1. 参照下线TiFlash 节点下线对应的TiFlash 节点。
2. 清除该TiFlash 节点的相关数据。
3. 重新在集群中部署TiFlash 节点。

7.1.3.5 TiFlash 分析慢

如果语句中含有 MPP 模式不支持的算子或函数等，TiDB 不会选择 MPP 模式，可能导致分析慢。此时，可以执行 `EXPLAIN` 语句检查 SQL 中是否含有TiFlash 不支持的函数或算子。

```
create table t(a datetime);
alter table t set tiflash replica 1;
insert into t values('2022-01-13');
set @@session.tidb_enforce_mpp=1;
explain select count(*) from t where subtime(a, '12:00:00') > '2022-01-01' group by a;
show warnings;
```

示例中，warning 消息显示，因为TiDB 5.4 及更早的版本尚不支持 `subtime` 函数的下推，因此TiDB 没有选择 MPP 模式。

```
+---------+-------+-----------------------------------------------------------------------------------+
> | Level   | Code | Message
    ′→ |
+---------+-------+-----------------------------------------------------------------------------------+
| Warning | 1105 | Scalar function 'subtime'(signature: SubDatetimeAndString, return type:
    ′→ datetime) is not supported to push down to tiflash now.       |
+---------+-------+-----------------------------------------------------------------------------------+
```

7.1.3.6 TiFlash 数据不同步

在部署完TiFlash 节点且进行了数据的同步操作（ALTER 操作）之后，如果实际没有数据同步到TiFlash 节点，你可以通过以下步骤确认或解决问题：

1. 检查同步操作是否执行。

   执行 `ALTER table <tbl_name> set tiflash replica <num>` 操作，查看是否有正常返回:

   - 如果有正常返回，进入下一步。
   - 如果无正常返回，请执行 `SELECT * FROM information_schema.tiflash_replica` 检查是否已经创建 TiFlash replica。如果没有，请重新执行 `ALTER table ${tbl_name} set tiflash replica ${num}`，查看是否有其他执行语句（如 `add index`），或者检查 DDL 操作是否正常。

2. 检查TiFlash Region 同步是否正常。

   查看 `progress` 是否有变化:

   - 如果有变化，说明TiFlash 同步正常，进入下一步。
   - 如果没有变化，说明TiFlash 同步异常，在 `tidb.log` 中，搜索 `Tiflash replica is not available` 相关日志。检查对应表的 `progress` 是否更新。如果无更新，请检查 `tiflash log` 来获取更多信息。例如，在 `tiflash log` 中搜索 `lag_region_info` 来判断同步落后的Region。

3. 使用 pd-ctl 检查 PD 的 Placement Rules 功能是否开启：

   ```
   echo 'config show replication' | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
   ```

   - 如果返回 `true`，进入下一步。
   - 如果返回 `false`，你需要先开启 Placement Rules 特性，然后进入下一步。

4. 检查集群副本数 `max-replicas` 配置是否合理。

   - 如果 `max-replicas` 取值未超过TiKV 节点数，进入下一步。
   - 如果 `max-replicas` 超过TiKV 节点数，PD 不会向TiFlash 同步数据。此时，请将 `max-replicas` 修改为小于等于TiKV 节点数的整数。

> 注意：
>
> `max-replicas` 的默认值是 3。在生产环境中，TiKV 节点数一般大于该值；在测试环境中，可以修改为 1。

```
curl -X POST -d
    '{ "group_id":
    "pd",
    "id": "default",
    "start_key": "",
    "end_key": "",
    "role": "voter",
    "count": 3,
    "location_labels":
    [ "host"

    ]
}' <http://172.16.x.xxx:2379/pd/api/v1/config/rule>
```

5. 检查TiDB是否为表创建 Placement rule。

   搜索 TiDB DDL Owner 的日志，检查 TiDB 是否通知 PD 添加 Placement rule。对于非分区表搜索 ConfigureTiFlashPDForTable；对于分区表，搜索 ConfigureTiFlashPDForPartitions。

   - 有关键字，进入下一步。
   - 没有关键字，收集相关组件的日志进行排查。

6. 检查 PD 是否为表设置 Placement rule。

   可以通过curl http://<pd-ip>:<pd-port>/pd/api/v1/config/rules/group/tiflash 查询比较当前PD 上的所有TiFlash 的 Placement rule。如果观察到有 id 为 table-<table_id>-r 的Rule，则表示 PD rule 设置成功。

7. 检查 PD 是否正常发起调度。

   查看 pd.log 日志是否出现 table-<table_id>-r 关键字，且之后是否出现 add operator 之类的调度行为。

   - 是，PD 调度正常。
   - 否，PD 调度异常。

7.1.3.7 TiFlash 数据同步卡住

如果TiFlash 数据一开始可以正常同步，过一段时间后全部或者部分数据无法继续同步，你可以通过以下步骤确认或解决问题：

1. 检查磁盘空间。

   检查磁盘使用空间比例是否高于 low-space-ratio 的值（默认值 0.8，即当节点的空间占用比例超过 80% 时，为避免磁盘空间被耗尽，PD 会尽可能避免往该节点迁移数据）。

   - 如果磁盘使用率大于等于 low-space-ratio，说明磁盘空间不足。此时，请删除不必要的文件，如 ${data}/flash/ 目录下的space_placeholder_file 文件（必要时可在删除文件后将reserve-space 设置为 0MB）。
   - 如果磁盘使用率小于 low-space-ratio，说明磁盘空间正常，进入下一步。

2. 检查是否有 down peer（down peer 没有清理干净可能会导致同步卡住）。

- 执行 pd-ctl region check-down-peer 命令检查是否有 down peer。
- 如果存在 down peer， 执行 pd-ctl operator add remove-peer <region-id> <tiflash-store-id> 命令将其清除。

7.1.3.8 数据同步慢

同步慢可能由多种原因引起，你可以按以下步骤进行排查。

1. 调大调度参数store limit，加快同步速度。
2. 调整TiFlash 侧负载。

   TiFlash 负载过大会引起同步慢，可通过Grafana 中的TiFlash-Summary 面板查看各个指标的负载情况：

   - Applying snapshots Count: TiFlash-summary > raft > Applying snapshots Count
   - Snapshot Predecode Duration: TiFlash-summary > raft > Snapshot Predecode Duration
   - Snapshot Flush Duration: TiFlash-summary > raft > Snapshot Flush Duration
   - Write Stall Duration: TiFlash-summary > Storage Write Stall > Write Stall Duration
   - generate snapshot CPU: TiFlash-Proxy-Details > Thread CPU > Region task worker pre-handle/ ′→ generate snapshot CPU

   根据业务优先级，调整负载情况。

7.2 故障场景

7.2.1 慢查询

7.2.1.1 慢查询日志

TiDB 会将执行时间超过tidb_slow_log_threshold（默认值为 300 毫秒）的语句输出到slow-query-file（默认值为“tidb-slow.log”）日志文件中，用于帮助用户定位慢查询语句，分析和解决 SQL 执行的性能问题。

TiDB 默认启用慢查询日志，可以修改系统变量tidb_enable_slow_log 来启用或禁用它。

7.2.1.1.1 日志示例

```
#### Time: 2019-08-14T09:26:59.487776265+08:00
#### Txn_start_ts: 410450924122144769
#### User@Host: root[root] @ localhost [127.0.0.1]
#### Conn_ID: 3086
#### Exec_retry_time: 5.1 Exec_retry_count: 3
#### Query_time: 1.527627037
#### Parse_time: 0.000054933
#### Compile_time: 0.000129729
#### Rewrite_time: 0.000000003 Preproc_subqueries: 2 Preproc_subqueries_time: 0.000000002
#### Optimize_time: 0.00000001
#### Wait_TS: 0.00001078
```

```
#### Process_time: 0.07 Request_count: 1 Total_keys: 131073 Process_keys: 131072 Prewrite_time:
    ′→ 0.335415029 Commit_time: 0.032175429 Get_commit_ts_time: 0.000177098
    ′→ Local_latch_wait_time: 0.106869448 Write_keys: 131072 Write_size: 3538944 Prewrite_region
    ′→ : 1
#### DB: test
#### Is_internal: false
#### Digest: 50a2e32d2abbd6c1764b1b7f2058d428ef2712b029282b776beb9506a365c0f1
#### Stats: t:pseudo
#### Num_cop_tasks: 1
#### Cop_proc_avg: 0.07 Cop_proc_p90: 0.07 Cop_proc_max: 0.07 Cop_proc_addr: 172.16.5.87:20171
#### Cop_wait_avg: 0 Cop_wait_p90: 0 Cop_wait_max: 0 Cop_wait_addr: 172.16.5.87:20171
#### Cop_backoff_regionMiss_total_times: 200 Cop_backoff_regionMiss_total_time: 0.2
    ′→ Cop_backoff_regionMiss_max_time: 0.2 Cop_backoff_regionMiss_max_addr: 127.0.0.1
    ′→ Cop_backoff_regionMiss_avg_time: 0.2 Cop_backoff_regionMiss_p90_time: 0.2
#### Cop_backoff_rpcPD_total_times: 200 Cop_backoff_rpcPD_total_time: 0.2
    ′→ Cop_backoff_rpcPD_max_time: 0.2 Cop_backoff_rpcPD_max_addr: 127.0.0.1
    ′→ Cop_backoff_rpcPD_avg_time: 0.2 Cop_backoff_rpcPD_p90_time: 0.2
#### Cop_backoff_rpcTiKV_total_times: 200 Cop_backoff_rpcTiKV_total_time: 0.2
    ′→ Cop_backoff_rpcTiKV_max_time: 0.2 Cop_backoff_rpcTiKV_max_addr: 127.0.0.1
    ′→ Cop_backoff_rpcTiKV_avg_time: 0.2 Cop_backoff_rpcTiKV_p90_time: 0.2
#### Mem_max: 525211
#### Disk_max: 65536
#### Prepared: false
#### Plan_from_cache: false
#### Succ: true
#### Plan: tidb_decode_plan('
    ′→ ZJAwCTMyXzcJMAkyMAlkYXRhOlRhYmxlU2Nhbl82CjEJMTBfNgkxAR0AdAEY1Dp0LCByYW5nZTpbLWluZiwraW5mXSwga2VlcCBvcmRlc
    ′→ ==')
use test;
insert into t select * from t;
```

7.2.1.1.2 字段含义说明

> **注意：**
> 慢查询日志中所有时间相关字段的单位都是“秒”

Slow Query 基础信息：

- `Time`：表示日志打印时间。
- `Query_time`：表示执行这个语句花费的时间。
- `Parse_time`：表示这个语句在语法解析阶段花费的时间。

- Compile_time：表示这个语句在查询优化阶段花费的时间。
- Optimize_time：表示这个语句在优化查询计划阶段花费的时间。
- Wait_TS：表示这个语句在等待获取事务 TS 阶段花费的时间。
- Query：表示 SQL 语句。慢日志里面不会打印 Query，但映射到内存表后，对应的字段叫 Query。
- Digest：表示 SQL 语句的指纹。
- Txn_start_ts：表示事务的开始时间戳，也是事务的唯一 ID，可以用这个值在TiDB 日志中查找事务相关的其他日志。
- Is_internal：表示是否为 TiDB 内部的 SQL 语句。true 表示TiDB 系统内部执行的 SQL 语句，false 表示用户执行的 SQL 语句。
- Index_names：表示这个语句执行用到的索引。
- Stats：表示这个语句使用到的统计信息的健康状态、内部版本号、总行数、修改行数以及加载状态。pseudo 状态表示统计信息不健康。如果有尝试使用但没有完全加载的统计信息，会在之后输出其内部状态。例如，t1:439478225786634241[105000;5000][col1:allEvicted][idx1:allEvicted] 的含义如下：
  - t1：本次查询优化过程中使用了 t1 表上的统计信息
  - 439478225786634241：其内部版本号
  - 105000：统计信息中维护的总行数
  - 5000：自上次收集统计信息以来记录的修改的行数
  - col1:allEvicted：col1 列对应的统计信息没有完全加载
  - idx1:allEvicted：idx1 索引对应的统计信息没有完全加载
- Succ：表示语句是否执行成功。
- Backoff_time：表示语句遇到需要重试的错误时在重试前等待的时间。常见的需要重试的错误有以下几种：遇到了lock、Region 分裂、tikv server is busy。
- Plan：表示语句的执行计划，用 select tidb_decode_plan('xxx...') SQL 语句可以解析出具体的执行计划。
- Binary_plan：表示以二进制格式编码后的语句的执行计划，用 select tidb_decode_binary_plan('xxx...') SQL 语句可以解析出具体的执行计划。传递的信息和 Plan 字段基本相同，但是解析出的执行计划的格式会和 Plan 字段不同。
- Prepared：表示这个语句是否是 Prepare 或 Execute 的请求。
- Plan_from_cache：表示这个语句是否命中了执行计划缓存。
- Plan_from_binding：表示这个语句是否用的绑定的执行计划。
- Has_more_results：表示这个语句的查询结果是否还有更多的数据待用户发起 fetch 命令获取。
- Rewrite_time：表示这个语句在查询改写阶段花费的时间。
- Preproc_subqueries：表示这个语句中被提前执行的子查询个数，如 where id in (select if from t) 这个子查询就可能被提前执行。
- Preproc_subqueries_time：表示这个语句中被提前执行的子查询耗时。
- Exec_retry_count：表示这个语句执行的重试次数。一般出现在悲观事务中，上锁失败时重试执行该语句。
- Exec_retry_time：表示这个语句的重试执行时间。例如某个查询一共执行了三次（前两次失败），则 Exec_retry_time 表示前两次的执行时间之和，Query_time 减去 Exec_retry_time 则为最后一次执行时间。
- KV_total：表示这个语句在 TiKV/TiFlash 上所有RPC 请求花费的时间。
- PD_total：表示这个语句在 PD 上所有RPC 请求花费的时间。
- Backoff_total：表示这个语句在执行过程中所有 backoff 花费的时间。
- Write_sql_response_total：表示这个语句把结果发送回客户端花费的时间。

- `Result_rows`：表示这个语句查询结果的行数。
- `Warnings`：表示这个语句执行过程中产生的警告，采用 JSON 格式。通常和 `SHOW WARNINGS` 语句的输出结果一致，但是可能会包含 `SHOW WARNINGS` 中没有的警告，因而可以提供更多诊断信息。这类警告将被标记为 `IsExtra: true`。
- `IsExplicitTxn`：表示这个语句是否在一个明确声明的事务中。如果是 `false`，表示这个语句的事务是 `autocommit=1`，即语句执行完成后就自动提交的事务。

和事务执行相关的字段：

- `Prewrite_time`：表示事务两阶段提交中第一阶段（prewrite 阶段）的耗时。
- `Commit_time`：表示事务两阶段提交中第二阶段（commit 阶段）的耗时。
- `Get_commit_ts_time`：表示事务两阶段提交中第二阶段（commit 阶段）获取commit 时间戳的耗时。
- `Local_latch_wait_time`：表示事务两阶段提交中第二阶段（commit 阶段）发起前在TiDB 侧等锁的耗时。
- `Write_keys`：表示该事务向 TiKV 的 Write CF 写入 Key 的数量。
- `Write_size`：表示事务提交时写 key 或 value 的总大小。
- `Prewrite_region`：表示事务两阶段提交中第一阶段（prewrite 阶段）涉及的TiKV Region 数量。每个 Region 会触发一次远程过程调用。
- `Wait_prewrite_binlog_time`：表示事务提交时用于写 binlog 的时间。
- `Resolve_lock_time`：表示事务提交时遇到锁后，清理锁或者等待锁过期的时间。

和内存使用相关的字段：

- `Mem_max`：表示执行期间 TiDB 使用的最大内存空间，单位为byte。

和硬盘使用相关的字段：

- `Disk_max`：表示执行期间 TiDB 使用的最大硬盘空间，单位为byte。

和 SQL 执行的用户相关的字段：

- `User`：表示执行语句的用户名。
- `Host`：表示执行语句的用户地址。
- `Conn_ID`：表示用户的链接 ID，可以用类似 `con:3` 的关键字在TiDB 日志中查找该链接相关的其他日志。
- `DB`：表示执行语句时使用的 database。

和 TiKV Coprocessor Task 相关的字段：

- `Request_count`：表示这个语句发送的 Coprocessor 请求的数量。
- `Total_keys`：表示 Coprocessor 扫过的 key 的数量。
- `Process_time`：执行 SQL 在TiKV 的处理时间之和，因为数据会并行的发到TiKV 执行，这个值可能会超过 `Query_time`。
- `Wait_time`：表示这个语句在 TiKV 的等待时间之和，因为TiKV 的 Coprocessor 线程数是有限的，当所有的 Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。

- `Process_keys`：表示 Coprocessor 处理的 key 的数量。相比total_keys，processed_keys 不包含 MVCC 的旧版本。如果processed_keys 和 total_keys 相差很大，说明旧版本比较多。
- `Num_cop_tasks`：表示这个语句发送的 Coprocessor 请求的数量。
- `Cop_proc_avg`：cop-task 的平均执行时间，包括一些无法统计的等待时间，如RocksDB 内的mutex。
- `Cop_proc_p90`：cop-task 的 P90 分位执行时间。
- `Cop_proc_max`：cop-task 的最大执行时间。
- `Cop_proc_addr`：执行时间最长的 cop-task 所在地址。
- `Cop_wait_avg`：cop-task 的平均等待时间，包括请求排队和获取 snapshot 时间。
- `Cop_wait_p90`：cop-task 的 P90 分位等待时间。
- `Cop_wait_max`：cop-task 的最大等待时间。
- `Cop_wait_addr`：等待时间最长的 cop-task 所在地址。
- `Rocksdb_delete_skipped_count`：RocksDB 读数据过程中已删除 Key 的扫描数。
- `Rocksdb_key_skipped_count`：RocksDB 扫数据时遇到的已删除(tombstone) Key 数量。
- `Rocksdb_block_cache_hit_count`：RocksDB 从 Block Cache 缓存中读数据的次数。
- `Rocksdb_block_read_count`：RocksDB 从文件系统中读数据的次数。
- `Rocksdb_block_read_byte`：RocksDB 从文件系统中读数据的数据量。
- `Rocksdb_block_read_time`：RocksDB 从文件系统中读数据的时间。
- `Cop_backoff_{backoff-type}_total_times`：因某种错误造成的 backoff 总次数。
- `Cop_backoff_{backoff-type}_total_time`：因某种错误造成的 backoff 总时间。
- `Cop_backoff_{backoff-type}_max_time`：因某种错误造成的最大 backoff 时间。
- `Cop_backoff_{backoff-type}_max_addr`：因某种错误造成的最大 backoff 时间的cop-task 地址。
- `Cop_backoff_{backoff-type}_avg_time`：因某种错误造成的平均 backoff 时间。
- `Cop_backoff_{backoff-type}_p90_time`：因某种错误造成的 P90 分位backoff 时间。

`backoff-type` 一般有以下几种：

- `tikvRPC`：给 TiKV 发送RPC 请求失败而产生的backoff。
- `tiflashRPC`：给 TiFlash 发送RPC 请求失败而产生的backoff。
- `pdRPC`：给 PD 发送RPC 请求失败而产生的backoff。
- `txnLock`：遇到锁冲突后产生的 backoff。
- `regionMiss`：Region 发生分裂或者合并后，TiDB 的Region 缓存信息过期导致请求失败而产生的backoff。
- `regionScheduling`：Region 还在调度中，尚未选出 Leader 导致无法处理请求而产生的backoff。
- `tikvServerBusy`：因为 TiKV 负载太高无法处理新请求而产生的backoff。
- `tiflashServerBusy`：因为 TiFlash 负载太高无法处理新请求而产生的backoff。
- `tikvDiskFull`：因为 TiKV 的磁盘满了而产生的backoff。
- `txnLockFast`：因为读数据时遇到了锁而产生的 backoff。

7.2.1.1.3 相关系统变量

- tidb_slow_log_threshold：设置慢日志的阈值，执行时间超过阈值的 SQL 语句将被记录到慢日志中。默认值是 300 ms。
- tidb_query_log_max_len：设置慢日志记录 SQL 语句的最大长度。默认值是 4096 byte。
- tidb_redact_log：设置慢日志记录 SQL 时是否将用户数据脱敏用? 代替。默认值是 0，即关闭该功能。
- tidb_enable_collect_execution_info：设置是否记录执行计划中各个算子的物理执行信息，默认值是 1。该功能对性能的影响约为 3%。开启该项后查看 `Plan` 的示例如下：

```
> select tidb_decode_plan('jAOIMAk1XzE3CTAJMQlmdW5jczpjb3VudChDb2x1bW4jNyktPkMJC/
    '→ BMNQkxCXRpbWU6MTAuOTMxNTA1bXMsIGxvb3BzOjIJMzcyIEJ5dGVzCU4vQQoxCTMyXzE4CTAJMQlpbmRleDpTdHJlY   W1BZ2dfOQkxCXQ
    '→ ');
+--
    '→
    '→
| tidb_decode_plan('jAOIMAk1XzE3CTAJMQlmdW5jczpjb3VudChDb2x1bW4jNyktPkMJC/
    '→ BMNQkxCXRpbWU6MTAuOTMxNTA1bXMsIGxvb3BzOjIJMzcyIEJ5dGVzCU4vQQoxCTMyXzE4CTAJMQlpbmRleDpTdHJlY   W1BZ2dfOQkxCXQ
    '→  |
+
    '→
|     id                    task     estRows               operator info
    '→                                                   actRows     execution info
    '→                                                                memory       disk
    '→                                 |
|     StreamAgg_17          root     1                     funcs:count(Column#7)->Column#5
    '→                                   1          time:10.931505ms, loops:2
    '→                                                      372 Bytes     N/A
    '→                                 |
|       └─IndexReader_18      root     1                     index:StreamAgg_9
    '→                                                 1          time:10.927685ms, loops:2, rpc
    '→ num: 1, rpc time:10.884355ms, proc keys:25007     206 Bytes     N/A
    '→                                 |
|         └─StreamAgg_9       cop      1                     funcs:count(1)->Column#7
    '→                                          1          time:11ms, loops:25
    '→                                                     N/A           N/A
|   '→                                 |
            └─IndexScan_16    cop      31281.857819905217    table:t, index:idx(a), range:[-inf
    '→ ,50000), keep order:false     25007      time:11ms, loops:25
    '→                                                     N/A           N/A
+   '→                                 |

    '→
```

在性能测试中可以关闭自动收集算子的执行信息：

```
set @@tidb_enable_collect_execution_info=0;
```

Plan 字段显示的格式和 EXPLAIN 或者 EXPLAIN ANALYZE 大致一致。可以查看 EXPLAIN 或者 EXPLAIN ANALYZE 文档了解更多关于执行计划的信息。

更多详细信息，可以参见TiDB 专用系统变量和语法。

7.2.1.1.4 慢日志内存映射表

用户可通过查询 INFORMATION_SCHEMA.SLOW_QUERY 表来查询慢查询日志中的内容，表中列名和慢日志中字段名一一对应，表结构可查看 SLOW_QUERY 表中的介绍。

> **注意：**
>
> 每次查询 SLOW_QUERY 表时，TiDB 都会去读取和解析一次当前的慢查询日志。

TiDB 4.0 中，SLOW_QUERY 已经支持查询任意时间段的慢日志，即支持查询已经被rotate 的慢日志文件的数据。用户查询时只需要指定 TIME 时间范围即可定位需要解析的慢日志文件。如果查询不指定时间范围，则仍然只解析当前的慢日志文件，示例如下：

不指定时间范围时，只会解析当前TiDB 正在写入的慢日志文件的慢查询数据：

```
select count(*),
       min(time),
       max(time)
from slow_query;
```

```
+----------+----------------------------+----------------------------------------------------+
| count(*) | min(time)                  | max(time)                  |
+----------+----------------------------+----------------------------------------------------+
| 122492   | 2020-03-11 23:35:20.908574 | 2020-03-25 19:16:38.229035 |
+----------+----------------------------+----------------------------------------------------+
```

指定查询 2020-03-10 00:00:00 到 2020-03-11 00:00:00 时间范围后，会定位指定时间范围内的慢日志文件后解析慢查询数据：

```
select count(*),
       min(time),
       max(time)
from slow_query
where time > '2020-03-10 00:00:00'
  and time < '2020-03-11 00:00:00';
```

```
+----------+----------------------------+----------------------------------------------------+
| count(*) | min(time)                  | max(time)                  |
+----------+----------------------------+----------------------------------------------------+
| 2618049  | 2020-03-10 00:00:00.427138 | 2020-03-10 23:00:22.716728 |
+----------+----------------------------+----------------------------------------------------+
```

> **注意：**
>
> 如果指定时间范围内的慢日志文件被删除，或者并没有慢查询，则查询结果会返回空。

TiDB 4.0 中新增了CLUSTER_SLOW_QUERY 系统表，用来查询所有 TiDB 节点的慢查询信息，表结构在 SLOW_QUERY 的基础上多增加了 INSTANCE 列，表示该行慢查询信息来自的TiDB 节点地址。使用方式和 SLOW_QUERY 系统表一样。

关于查询 CLUSTER_SLOW_QUERY 表，TiDB 会把相关的计算和判断下推到其他节点执行，而不是把其他节点的慢查询数据都取回来在一台TiDB 上执行。

7.2.1.1.5 查询 SLOW_QUERY/CLUSTER_SLOW_QUERY 示例

搜索Top N 的慢查询

查询Top 2 的用户慢查询。is_internal=false 表示排除TiDB 内部的慢查询，只看用户的慢查询：

```
select query_time, query
from information_schema.slow_query
where is_internal = false  -- 排除 TiDB 内部的慢查询 SQL
order by query_time desc
limit 2;
```

输出样例：

```
+--------------+-----------------------------------------------------------------------+
| query_time   | query                                                                 |
+--------------+-----------------------------------------------------------------------+
| 12.77583857  | select * from t_slim, t_wide where t_slim.c0=t_wide.c0;               |
|  0.734982725 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c0; |
+--------------+-----------------------------------------------------------------------+
```

搜索某个用户的Top N 慢查询

下面例子中搜索test 用户执行的慢查询 SQL，且按执行消耗时间逆序排序显式前 2 条：

```
select query_time, query, user
from information_schema.slow_query
where is_internal = false  -- 排除 TiDB 内部的慢查询 SQL
  and user = "test"        -- 查找的用户名
order by query_time desc
limit 2;
```

输出样例：

```
+--------------+-----------------------------------------------------------------------+-----------------+
    '→
| Query_time   | query                                                                 | user
    '→           |
+--------------+-----------------------------------------------------------------------+-----------------+
    '→
| 0.676408014 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c1; | test
    '→           |
```

```
+-------------+-------------------------------------------------------------------------+-----------------+
    '→
```

根据 SQL 指纹搜索同类慢查询

在得到Top N 的慢查询 SQL 后，可通过 SQL 指纹继续搜索同类慢查询 SQL。

先获取Top N 的慢查询和对应的 SQL 指纹：

```
select query_time, query, digest
from information_schema.slow_query
where is_internal = false
order by query_time desc
limit 1;
```

输出样例：

```
+-------------+------------------------------+
    '→
| query_time  | query                        | digest
    '→                                                                |
 -------------+------------------------------
+             +                              +
    '→
| 0.302558006 | select * from t1 where a=1; | 4751
    '→ cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa |
+             +                              +
    '→
```

再根据 SQL 指纹搜索同类慢查询：

```
select query, query_time
from information_schema.slow_query
where digest = "4751cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa";
```

输出样例：

```
+-----------------------------+-------------+
| query                       | query_time  |
+-----------------------------+-------------+
| select * from t1 where a=1; | 0.302558006 |
| select * from t1 where a=2; | 0.401313532 |
+-----------------------------+-------------+
```

搜索统计信息为pseudo 的慢查询 SQL 语句

```
select query, query_time, stats
from information_schema.slow_query
where is_internal = false
  and stats like '%pseudo%';
```

输出样例：

```
+------------------------------+-------------+----------------------------------+
| query                        | query_time  | stats                            |
+------------------------------+-------------+----------------------------------+
| select * from t1 where a=1; | 0.302558006 | t1:pseudo                        |
| select * from t1 where a=2; | 0.401313532 | t1:pseudo                        |
| select * from t1 where a>2; | 0.602011247 | t1:pseudo                        |
| select * from t1 where a>3; | 0.50077719  | t1:pseudo                        |
| select * from t1 join t2;    | 0.931260518 | t1:407872303825682445,t2:pseudo |
+------------------------------+-------------+----------------------------------+
```

查询执行计划发生变化的慢查询

由于统计信息过时，或者统计信息因为误差无法精确反映数据的真实分布情况时，可能导致同类型 SQL 的执行计划发生改变导致执行变慢，可以用以下 SQL 查询哪些 SQL 具有不同的执行计划：

```
select count(distinct plan_digest) as count,
       digest,
       min(query)
from cluster_slow_query
group by digest
having count > 1
limit 3\G
```

输出样例：

```
***************************[ 1. row ]***************************
count      | 2
digest     | 17b4518fde82e32021877878bec2bb309619d384fca944106fcaf9c93b536e94
min(query) | SELECT DISTINCT c FROM sbtest25 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↩ (291638, 291737)];
***************************[ 2. row ]***************************
count      | 2
digest     | 9337865f3e2ee71c1c2e740e773b6dd85f23ad00f8fa1f11a795e62e15fc9b23
min(query) | SELECT DISTINCT c FROM sbtest22 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↩ (215420, 215519)];
***************************[ 3. row ]***************************
count      | 2
digest     | db705c89ca2dfc1d39d10e0f30f285cbbadec7e24da4f15af461b148d8ffb020
min(query) | SELECT DISTINCT c FROM sbtest11 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↩ (303359, 303458)];
```

然后可以用查询结果中的 SQL 指纹进一步查询不同的plan

```
select min(plan),
       plan_digest
from cluster_slow_query
```

```
where digest='17b4518fde82e32021877878bec2bb309619d384fca944106fcaf9c93b536e94'
group by plan_digest\G
```

输出样例：

```
*************************** 1. row ***************************
  min(plan):    Sort_6                  root    100.00131380758702      sbtest.sbtest25.c:asc
        └─HashAgg_10            root    100.00131380758702      group by:sbtest.sbtest25.c,
          ′→ funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
          └─TableReader_15      root    100.00131380758702      data:TableRangeScan_14
            └─TableScan_14      cop     100.00131380758702      table:sbtest25, range

                ′→ :[502791,502890], keep order:false
plan_digest: 6afbbd21f60ca6c6fdf3d3cd94f7c7a49dd93c00fcf8774646da492e50e204ee
*************************** 2. row ***************************
  min(plan):    Sort_6                  root    1                       sbtest.sbtest25.c:asc
        └─HashAgg_12            root    1                       group by:sbtest.sbtest25.c,
          └─TableReader_13      root    1                       data:HashAgg_8
            └─HashAgg_8         cop     1                       group by:sbtest.sbtest25.c,
              └─TableScan_11    cop     1.2440069558121831      table:sbtest25, range

                ′→ :[472745,472844], keep order:false
```

查询集群各个TIDB 节点的慢查询数量

```
select instance, count(*) from information_schema.cluster_slow_query where time >= "2020-03-06
    ′→ 00:00:00" and time < now() group by instance;
```

输出样例：

```
+---------------+------------------+
| instance      | count(*) |
+---------------+------------------+
| 0.0.0.0:10081 | 124      |
| 0.0.0.0:10080 | 119771   |
+---------------+------------------+
```

查询仅出现在异常时间段的慢日志

假如发现 2020-03-10 13:24:00 ~ 2020-03-10 13:27:00 的 QPS 降低或者延迟上升等问题，可能是由于突然出现大查询导致的， 可以用下面 SQL 查询仅出现在异常时间段的慢日志， 其中 2020-03-10 13:20:00 ~ 2020-03-10 13:23:00 为正常时间段。

```
SELECT * FROM
    (SELECT /*+ AGG_TO_COP(), HASH_AGG() */ count(*),
         min(time),
         sum(query_time) AS sum_query_time,
         sum(Process_time) AS sum_process_time,
         sum(Wait_time) AS sum_wait_time,
```

```
        sum(Commit_time),
        sum(Request_count),
        sum(process_keys),
        sum(Write_keys),
        max(Cop_proc_max),
        min(query),min(prev_stmt),
        digest
    FROM information_schema.CLUSTER_SLOW_QUERY
    WHERE time >= '2020-03-10 13:24:00'
            AND time < '2020-03-10 13:27:00'
            AND Is_internal = false
    GROUP BY  digest) AS t1
WHERE t1.digest NOT IN
    (SELECT /*+ AGG_TO_COP(), HASH_AGG() */ digest
    FROM information_schema.CLUSTER_SLOW_QUERY
    WHERE time >= '2020-03-10 13:20:00'
            AND time < '2020-03-10 13:23:00'
    GROUP BY  digest)
ORDER BY  t1.sum_query_time DESC limit 10\G
```

输出样例：

```
***************************[ 1. row ]***************************
count(*)            | 200
min(time)           | 2020-03-10 13:24:27.216186
sum_query_time      | 50.114126194
sum_process_time    | 268.351
sum_wait_time       | 8.476
sum(Commit_time)    | 1.044304306
sum(Request_count)  | 6077
sum(process_keys)   | 202871950
sum(Write_keys)     | 319500
max(Cop_proc_max)   | 0.263
min(query)          | delete from test.tcs2 limit 5000;
min(prev_stmt)      |
digest              | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

7.2.1.1.6 解析其他的TiDB 慢日志文件

TiDB 通过 session 变量 `tidb_slow_query_file` 控制查询 `INFORMATION_SCHEMA.SLOW_QUERY` 时要读取和解析的文件，可通过修改改 session 变量的值来查询其他慢查询日志文件的内容：

```
set tidb_slow_query_file = "/path-to-log/tidb-slow.log"
```

7.2.1.1.7 用 pt-query-digest 工具分析TiDB 慢日志

可以用 pt-query-digest 工具分析TiDB 慢日志。

> **注意：**
>
> 建议使用pt-query-digest 3.0.13 及以上版本。

示例如下：

```
pt-query-digest --report tidb-slow.log
```

输出样例：

```
# 320ms user time, 20ms system time, 27.00M rss, 221.32M vsz
# Current date: Mon Mar 18 13:18:51 2019
# Hostname: localhost.localdomain
# Files: tidb-slow.log
# Overall: 1.02k total, 21 unique, 0 QPS, 0x concurrency _____________________
# Time range: 2019-03-18-12:22:16 to 2019-03-18-13:08:52

# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time            218s     10ms     13s    213ms     30ms       1s     19ms
# Query size        175.37k       9   2.01k  175.89  158.58  122.36  158.58
# Commit time          46ms     2ms     7ms     3ms     7ms     1ms     3ms
# Conn ID                71       1      16    8.88   15.25    4.06    9.83
# Process keys      581.87k       2 103.15k  596.43  400.73   3.91k  400.73
# Process time         31s     1ms     10s     32ms     19ms   334ms     16ms
# Request coun        1.97k       1      10    2.02    1.96    0.33    1.96
# Total keys        636.43k       2 103.16k  652.35  793.42   3.97k  400.73
# Txn start ts      374.38E       0  16.00E 375.48P   1.25P  89.05T   1.25P
# Wait time            943ms     1ms     19ms     1ms     2ms     1ms   972us
.
.
.
```

定位问题语句的方法

并不是所有 SLOW_QUERY 的语句都是有问题的。会造成集群整体压力增大的，是那些process_time 很大的语句。wait_time 很大，但process_time 很小的语句通常不是问题语句，是因为被问题语句阻塞，在执行队列等待造成的响应时间过长。

7.2.1.1.8 ADMIN SHOW SLOW 命令

除了获取TiDB 日志，还有一种定位慢查询的方式是通过 ADMIN SHOW SLOW SQL 命令：

```
ADMIN SHOW SLOW recent N;
```

```
ADMIN SHOW SLOW TOP [internal | all] N;
```

recent N 会显示最近的N 条慢查询记录，例如：

```
ADMIN SHOW SLOW recent 10;
```

top N 则显示最近一段时间（大约几天）内，最慢的查询记录。如果指定 internal 选项，则返回查询系统内部 SQL 的慢查询记录；如果指定 all 选项，返回系统内部和用户 SQL 汇总以后的慢查询记录；默认只返回用户 SQL 中的慢查询记录。

```
ADMIN SHOW SLOW TOP 3;
ADMIN SHOW SLOW TOP internal 3;
ADMIN SHOW SLOW TOP all 5;
```

由于内存限制，保留的慢查询记录的条数是有限的。当命令查询的N 大于记录条数时，返回的结果记录条数会小于N。

输出内容详细说明，如下：

列名	描述
start	SQL 语句执行开始时间
duration	SQL 语句执行持续时间
details	执行语句的详细信息
succ	SQL 语句执行是否成功，1：成功，0：失败
conn_id	session 连接ID
transaction_ts	事务提交的commit ts
user	执行该语句的用户名
db	执行该 SQL 涉及到database
table_ids	执行该 SQL 涉及到表的ID
index_ids	执行该 SQL 涉及到索引ID
internal	表示为TiDB 内部的 SQL 语句
digest	表示 SQL 语句的指纹
sql	执行的 SQL 语句

7.2.1.2 分析慢查询

处理慢查询分为两步：

1. 从大量查询中定位出哪一类查询比较慢
2. 分析这类慢查询的原因

第一步可以通过慢日志、statement-summary 方便地定位，推荐直接使用TiDB Dashboard，它整合了这两个功能，且能方便直观地在浏览器中展示出来。本文聚焦第二步。

首先将慢查询归因成两大类：

- 优化器问题：如选错索引，选错Join 类型或顺序。
- 系统性问题：将非优化器问题都归结于此类。如：某个TiKV 实例忙导致处理请求慢，Region 信息过期导致查询变慢。

实际中，优化器问题可能造成系统性问题。例如对于某类查询，优化器应使用索引，但却使用了全表扫。这可能导致这类 SQL 消耗大量资源，造成某些 KV 实例 CPU 飚高等问题。表现上看就是一个系统性问题，但本质是优化器问题。

分析优化器问题需要有判断执行计划是否合理的能力，而系统性问题的定位相对简单，因此面对慢查询推荐的分析过程如下：

1. 定位查询瓶颈：即查询过程中耗时多的部分
2. 分析系统性问题：根据瓶颈点，结合当时的监控/日志等信息，分析可能的原因
3. 分析优化器问题：分析是否有更好的执行计划

接下来会分别介绍上面几点。

7.2.1.2.1 定位查询瓶颈

定位查询瓶颈需要对查询过程有一个大致理解，TiDB 处理查询过程的关键阶段都在performance-map 图中了。

查询的耗时信息可以从下面几种方式获得：

- 慢日志（推荐直接在TiDB Dashboard 中查看）
- `explain analyze` 语句

他们的侧重点不同：

- 慢日志记录了 SQL 从解析到返回，几乎所有阶段的耗时，较为全面（在 TiDB Dashboard 中可以直观地查询和分析慢日志）；
- `explain analyze` 可以拿到 SQL 实际执行中每个执行算子的耗时，对执行耗时有更细分的统计；

总的来说，利用慢日志和 `explain analyze` 可以比较准确地定位查询的瓶颈点，帮助你判断这条 SQL 慢在哪个模块(TiDB/TiKV)，慢在哪个阶段，下面会有一些例子。

另外在 4.0.3 之后，慢日志中的 `Plan` 字段也会包含 SQL 的执行信息，也就是 `explain analyze` 的结果，这样一来 SQL 的所有耗时信息都可以在慢日志中找到。

7.2.1.2.2 分析系统性问题

对于系统性问题，我们根据执行阶段，分成三个大类：

1. TiKV 处理慢：如coprocessor 处理数据慢
2. TiDB 执行慢：主要指执行阶段，如某个Join 算子处理数据慢
3. 其他关键阶段慢：如取时间戳慢

拿到一个慢查询，我们应该先根据已有信息判断大致是哪个大类，再具体分析。

TiKV 处理慢

如果是TiKV 处理慢，可以很明显的通过explain analyze 中看出来。例如下面这个例子，可以看到StreamAgg_8 和 TableFullScan_15 这两个 tikv-task （在 task 列可以看出这两个任务类型是 cop[tikv]）花费了 170ms，而 TiDB 部分的算子耗时，减去这 170ms 后，耗时占比非常小，说明瓶颈在TiKV。

```
+--
    '→ ------------------------+---------+---------+-----------+--------------+------------------------------
    '→
| id                          | estRows | actRows | task      | access object | execution info
    '→                                                                          | operator info
    '→                   | memory    | disk |
+--
    '→ ------------------------+---------+---------+-----------+--------------+------------------------------
    '→
| StreamAgg_16                | 1.00    | 1       | root      |               | time:170.08572ms,
    '→ loops:2                                                                | funcs:count(Column#5)->
    '→ Column#3 | 372 Bytes | N/A  |
| └─TableReader_17            | 1.00    | 1       | root      |               | time:170.080369ms
    '→ , loops:2, rpc num: 1, rpc time:17.023347ms, proc keys:28672 | data:StreamAgg_8
    '→                 | 202 Bytes | N/A  |
|   └─StreamAgg_8             | 1.00    | 1       | cop[tikv] |               | time:170ms, loops
    '→ :29                                                                    | funcs:count(1)->Column#5
    '→        | N/A       | N/A  |
|     └─TableFullScan_15      | 7.00    | 28672   | cop[tikv] | table:t       | time:170ms, loops
    '→ :29                                                                    | keep order:false, stats:
    '→ pseudo  | N/A       | N/A  |
+--
    '→ ------------------------+---------+---------+-----------+--------------+------------------------------
    '→
```

另外在慢日志中，Cop_process 和 Cop_wait 字段也可以帮助判断，如下面这个例子，查询整个耗时是 180.85ms 左右，而最大的那个 coptask 就消耗了 171ms，可以说明对这个查询而言，瓶颈在 TiKV 侧。

慢日志中的各个字段的说明可以参考慢查询日志中的字段含义说明

```
#### Query_time: 0.18085
...
#### Num_cop_tasks: 1
#### Cop_process: Avg_time: 170ms P90_time: 170ms Max_time: 170ms Max_addr: 10.6.131.78
#### Cop_wait: Avg_time: 1ms P90_time: 1ms Max_time: 1ms Max_Addr: 10.6.131.78
```

根据上述方式判断是TiKV 慢后，可以依次排查TiKV 慢的原因。

TiKV 实例忙

一条 SQL 可能会去从多个TiKV 上拿数据，如果某个TiKV 响应慢，可能拖慢整个 SQL 的处理速度。

慢日志中的 Cop_wait 可以帮忙判断这个问题：

```
#### Cop_wait: Avg_time: 1ms P90_time: 2ms Max_time: 110ms Max_Addr: 10.6.131.78
```

如上图，发给 10.6.131.78 的一个 cop-task 等待了 110ms 才被执行，可以判断是当时该实例忙，此时可以打开当时的 CPU 监控辅助判断。

过期 key 多

如果 TiKV 上过期的数据比较多，在扫描的时候则需要处理这些不必要的数据，影响处理速度。

这可以通过 Total_keys 和 Processed_keys 判断，如果两者相差较大，则说明旧版本的 key 太多：

```
...
#### Total_keys: 2215187529 Processed_keys: 1108056368
...
```

其他关键阶段慢

取 TS 慢

可以对比慢日志中的 Wait_TS 和 Query_time，因为 TS 有预取操作，通常来说 Wait_TS 应该很低。

```
#### Query_time: 0.0300000
...
#### Wait_TS: 0.02500000
```

Region 信息过期

TiDB 侧 Region 信息可能过期，此时TiKV 可能返回 regionMiss 的错误，然后TiDB 会从 PD 去重新获取Region 信息，这些信息会被反应在 Cop_backoff 信息内，失败的次数和总耗时都会被记录下来。

```
#### Cop_backoff_regionMiss_total_times: 200 Cop_backoff_regionMiss_total_time: 0.2
    ′→ Cop_backoff_regionMiss_max_time: 0.2 Cop_backoff_regionMiss_max_addr: 127.0.0.1
    ′→ Cop_backoff_regionMiss_avg_time: 0.2 Cop_backoff_regionMiss_p90_time: 0.2
#### Cop_backoff_rpcPD_total_times: 200 Cop_backoff_rpcPD_total_time: 0.2
    ′→ Cop_backoff_rpcPD_max_time: 0.2 Cop_backoff_rpcPD_max_addr: 127.0.0.1
    ′→ Cop_backoff_rpcPD_avg_time: 0.2 Cop_backoff_rpcPD_p90_time: 0.2
```

子查询被提前执行

对于带有非关联子查询的语句，子查询部分可能被提前执行，如：select * from t1 where a = (select max ′→ (a)from t2)，select max(a)from t2 部分可能在优化阶段被提前执行。这种查询用 explain analyze 看不到对应的耗时，如下：

```
mysql> explain analyze select count(*) from t where a=(select max(t1.a) from t t1, t t2 where t1.
    ′→ a=t2.a);
+--
    ′→ ----------------------------+----------+---------+-----------+---------------+--------------------------------+
    ′→
| id                            | estRows  | actRows | task      | access object | execution info
    ′→           | operator info                     | memory    | disk |
```

```
+--
    '→ ---------------------------+---------+---------+-----------+--------------+--------------------------------+
    '→
| StreamAgg_59                  | 1.00     | 1       | root      |              | time:4.69267ms,
    '→  loops:2  | funcs:count(Column#10)->Column#8 | 372 Bytes | N/A  |
| └─TableReader_60              | 1.00     | 1       | root      |              | time:4.690428
    '→ ms, loops:2 | data:StreamAgg_48                | 141 Bytes | N/A  |
|   └─StreamAgg_48              | 1.00     |         | cop[tikv] |              | time:0ns,
    '→ loops:0        | funcs:count(1)->Column#10        | N/A       | N/A  |
|     └─Selection_58            | 16384.00 |         | cop[tikv] |              | time:0ns,
    '→ loops:0        | eq(test.t.a, 1)                  | N/A       | N/A  |
|       └─TableFullScan_57      | 16384.00 | -1      | cop[tikv] | table:t      | time:0s, loops
    '→ :0         | keep order:false                 | N/A       | N/A  |
+--
    '→ ---------------------------+---------+---------+-----------+--------------+--------------------------------+
    '→
5 rows in set (7.77 sec)
```

不过可以从慢日志中排查这种情况：

```
#### Query_time: 7.770634843
...
#### Rewrite_time: 7.765673663 Preproc_subqueries: 1 Preproc_subqueries_time: 7.765231874
```

可以看到有 1 个子查询被提前执行，花费了 7.76s。

TiDB 执行慢

这里我们假设TiDB 的执行计划正确（不正确的情况在分析优化器问题这一节中说明），但是执行上很慢；

解决这类问题主要靠调整参数或利用hint，并结合 explain analyze 对 SQL 进行调整。

并发太低

如果发现瓶颈在有并发的算子上，可以通过调整并发度来尝试提速，如下面的执行计划中：

```
mysql> explain analyze select sum(t1.a) from t t1, t t2 where t1.a=t2.a;
+--
    '→ --------------------------------+--------------+----------+----------+--------------+-----------------
    '→
| id                                | estRows      | actRows   | task      | access object |
    '→ execution info                                                                        |
    '→ operator info                                    | memory           | disk     |
+--
    '→ --------------------------------+--------------+----------+----------+--------------+-----------------
    '→
| HashAgg_11                        | 1.00  | 1  | root  |                                    | time
    '→ :9.666832189s, loops:2, PartialConcurrency:4, FinalConcurrency:4                 |  funcs:
    '→ sum(Column#6)->Column#5                  | 322.125 KB       | N/A     |
```

```
| └─Projection_24                  | 268435456.00 | 268435456 | root      |               | time
    ′→ :9.098644711s, loops:262145, Concurrency:4                                          | cast(
    ′→ test.t.a, decimal(65,0) BINARY)->Column#6 | 199 KB           | N/A      |
|   └─HashJoin_14                  | 268435456.00 | 268435456 | root      |               | time
    ′→ :6.616773501s, loops:262145, Concurrency:5, probe collision:0, build:881.404µs  | inner
    ′→ join, equal:[eq(test.t.a, test.t.a)]      | 131.75 KB        | 0 Bytes |
|     ├─TableReader_21(Build)      | 16384.00     | 16384     | root      |               | time
    ′→ :6.553717ms, loops:17                                                               | data:
    ′→ Selection_20                              | 33.6318359375 KB | N/A      |
|     │ └─Selection_20             | 16384.00     |           | cop[tikv] |               | time
    ′→ :0ns, loops:0                                                                       | not(
    ′→ isnull(test.t.a))                          | N/A              | N/A      |
|     │   └─TableFullScan_19       | 16384.00     | -1        | cop[tikv] | table:t2      | time
    ′→ :0s, loops:0                                                                        | keep
    ′→ order:false                               | N/A              | N/A      |
|     └─TableReader_18(Probe)      | 16384.00     | 16384     | root      |               | time
    ′→ :6.880923ms, loops:17                                                               | data:
    ′→ Selection_17                              | 33.6318359375 KB | N/A      |
|       └─Selection_17             | 16384.00     |           | cop[tikv] |               | time
    ′→ :0ns, loops:0                                                                       | not(
    ′→ isnull(test.t.a))                          | N/A              | N/A      |
|         └─TableFullScan_16       | 16384.00     | -1        | cop[tikv] | table:t1      | time
    ′→ :0s, loops:0                                                                        | keep
    ′→ order:false                               | N/A              | N/A      |
+--
    ′→  -------------------------------+--------------+-----------+-----------+---------------+----------------
    ′→
9 rows in set (9.67 sec)
```

发现耗时主要在 HashJoin_14 和 Projection_24，可以酌情通过 SQL 变量来提高他们的并发度进行提速。

system-variables 中有所有的系统变量，如想提高HashJoin_14 的并发度，则可以修改变量tidb_hash_join_concurrency
′→。

产生了落盘

执行慢的另一个原因是执行过程中，因为到达内存限制，产生了落盘，这点在执行计划和慢日志中都能看到：

```
+--
    ′→ ------------------------+-----------+---------+-----------+---------------+--------------------------------+
    ′→
| id                         | estRows    | actRows | task      | access object | execution info
    ′→               | operator info         | memory                 | disk           |
+--
    ′→ ------------------------+-----------+---------+-----------+---------------+--------------------------------+
    ′→
| Sort_4                     | 462144.00 | 462144  | root      |               | time:2.02848898s,
    ′→ loops:453  | test.t.a              | 149.68795776367188 MB | 219.3203125 MB |
```

```
| └─TableReader_8          | 462144.00 | 462144   | root      |               | time:616.211272ms,
    ↪  loops:453 | data:TableFullScan_7 | 197.49601364135742 MB | N/A             |
|   └─TableFullScan_7      | 462144.00 | -1       | cop[tikv] | table:t       | time:0s, loops:0
    ↪              | keep order:false      | N/A                   | N/A             |
+--
    ↪ ----------------------+-----------+---------+-----------+---------------+-------------------------------------------------+
    ↪
```

```
...
#### Disk_max: 229974016
...
```

做了笛卡尔积的Join

做笛卡尔积的Join 会产生左边孩子行数 * 右边孩子行数这么多数据，效率较低，应该尽量避免；

目前对于产生笛卡尔积的Join 会在执行计划中显示的标明 CARTESIAN，如下：

```
mysql> explain select * from t t1, t t2 where t1.a>t2.a;
+--
    ↪                             +             +           +               +
    ↪
| id                            | estRows     | task      | access object | operator info
    ↪                                         |
+--
    ↪ ------------------------------+-------------+-----------+---------------+                                                 +
    ↪
| HashJoin_8                    | 99800100.00 | root      |               | CARTESIAN inner join,
    ↪ other cond:gt(test.t.a, test.t.a) |
| ├─TableReader_15(Build)       | 9990.00     | root      |               | data:Selection_14
    ↪                                       |
| │ └─Selection_14              | 9990.00     | cop[tikv] |               | not(isnull(test.t.a))
    ↪                                     |
| │   └─TableFullScan_13        | 10000.00    | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo                          |
| └─TableReader_12(Probe)       | 9990.00     | root      |               | data:Selection_11
    ↪                                       |
|   └─Selection_11              | 9990.00     | cop[tikv] |               | not(isnull(test.t.a))
    ↪                                     |
|     └─TableFullScan_10        | 10000.00    | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo                          |
+--
    ↪                             +             +           +               +
    ↪
```

7.2.1.2.3 分析优化器问题

分析优化问题需要有判断执行计划是否合理的能力，这需要对优化过程和各算子有一定了解。

下面是一组例子，假设表结构为 `create table t (id int, a int, b int, c int, primary key(id), key(a), key(b, c))`：

1. `select * from t`：没有过滤条件，会扫全表，所以会用 `TableFullScan` 算子读取数据；
2. `select a from t where a=2`：有过滤条件且只读索引列，所以会用 `IndexReader` 算子读取数据；
3. `select * from t where a=2`：在 a 有过滤条件，但索引a 不能完全覆盖需要读取的内容，因此会采用 `IndexLookup`；
4. `select b from t where c=3`：多列索引没有前缀条件就用不上，所以会用 `IndexFullScan`；
5. …

上面举例了数据读入相关的算子，在理解TiDB 执行计划中描述了更多算子的情况。

另外阅读SQL 性能调优整个小节能增加你对TiDB 优化器的了解，帮助判断执行计划是否合理。

由于大多数优化器问题在SQL 性能调优已经有解释，这里就直接列举出来跳转过去：

1. 索引选择错误
2. Join 顺序错误
3. 表达式未下推

7.2.2 TiDB OOM 故障排查

本文总结了TiDB Out of Memory (OOM) 常见问题的解决思路、故障现象、故障原因、解决方法，以及需要收集的诊断信息。在遇到 OOM 问题时，你可以参考本文档来排查错误原因并进行处理。

7.2.2.1 常见故障现象

OOM 常见的故障现象包括（但不限于）：

- 客户端报错：`SQL error, errno = 2013, state = 'HY000': Lost connection to MySQL server during query`
- 查看Grafana 监控，发现以下现象：
  - TiDB > Server > Memory Usage 显示process/heapInUse 持续升高，达到阈值后掉零
  - TiDB > Server > Uptime 显示为掉零
  - TiDB-Runtime > Memory Usage 显示 estimate-inuse 持续升高
- 查看 `tidb.log`，可发现如下日志条目：
  - OOM 相关的Alarm：`[WARN] [memory_usage_alarm.go:139] ["tidb-server has the risk of OOM. Running SQLs and heap profile will be recorded in record path"]`。关于该日志的详细说明，请参考`memory-usage-alarm-ratio`。
  - 重启相关的日志条目：`[INFO] [printer.go:33] ["Welcome to TiDB."]`。

7.2.2.2 整体排查思路

在排查 OOM 问题时，整体遵循以下排查思路：

1. 确认是否属于 OOM 问题。

   执行以下命令查看操作系统日志。如果故障发生的时间点附近存在 `oom-killer` 的日志，则可以确定是 OOM 问题。

   ```
   dmesg -T | grep tidb-server
   ```

   下面是包含 `oom-killer` 的日志输出示例：

   ```
   ......
   Mar 14 16:55:03 localhost kernel: tidb-server invoked oom-killer: gfp_mask=0x201da, order=0,
       ↪  oom_score_adj=0
   Mar 14 16:55:03 localhost kernel: tidb-server cpuset=/ mems_allowed=0
   Mar 14 16:55:03 localhost kernel: CPU: 14 PID: 21966 Comm: tidb-server Kdump: loaded Not
       ↪ tainted 3.10.0-1160.el7.x86_64 #1
   Mar 14 16:55:03 localhost kernel: Hardware name: QEMU Standard PC (i440FX + PIIX, 1996),
       ↪ BIOS rel-1.14.0-0-g155821a1990b-prebuilt.qemu.org 04/01/2014
   ......
   Mar 14 16:55:03 localhost kernel: Out of memory: Kill process 21945 (tidb-server) score 956
       ↪ or sacrifice child
   Mar 14 16:55:03 localhost kernel: Killed process 21945 (tidb-server), UID 1000, total-vm
       ↪ :33027492kB, anon-rss:31303276kB, file-rss:0kB, shmem-rss:0kB
   Mar 14 16:55:07 localhost systemd: tidb-4000.service: main process exited, code=killed,
       ↪ status=9/KILL
   ......
   ```

2. 确认是 OOM 问题之后，可以进一步排查触发 OOM 的原因是部署问题还是数据库问题。

   - 如果是部署问题触发 OOM，需要排查资源配置、混合部署的影响。
   - 如果是数据库问题触发 OOM，常见原因：
     - TiDB 处理较大的数据流量，如：大查询、大写入、数据导入等。
     - TiDB 的高并发场景，多条 SQL 并发消耗资源，或者算子并发高。
     - TiDB 内存泄露，资源没有释放。

   具体排查方法请参考下面的章节。

7.2.2.3 常见故障原因和解决方法

根据 OOM 出现的原因，一般可以分为以下几种情况：

- 部署问题
- 数据库问题
- 客户端问题

7.2.2.3.1 部署问题

如果是由于部署不当导致的 OOM 问题，常见的原因有：

- 操作系统内存容量规划偏小，导致内存不足。
- TiUP `resource_control` 配置不合理。
- 在混合部署的情况下（指TiDB 和其他应用程序部署在同一台服务器上），其他应用程序抢占资源导致 TiDB 被 `oom-killer` 关闭。

7.2.2.3.2 数据库问题

本节介绍由数据库问题导致的 OOM 问题和解决办法。

> 注意：
>
> 如果 SQL 返回 `ERROR 1105 (HY000): Out Of Memory Quota![conn_id=54]`，是由于配置了`tidb_mem_quota_query` 导致， 数据库的内存使用控制行为会触发该报错。此报错为正常行为。

执行 SQL 语句时消耗太多内存

可以根据以下不同的触发 OOM 的原因，采取相应的措施减少 SQL 的内存使用：

- 如果 SQL 的执行计划不优，比如由于缺少合适的索引、统计信息过期、优化器bug 等原因，会导致选错 SQL 的执行计划，进而出现巨大的中间结果集累积在内存中。这种情况下可以考虑采取以下措施：
  - 添加合适的索引
  - 使用算子的数据落盘功能
  - 调整表之间的JOIN 顺序
  - 使用 hint 进行调优
- 一些算子和函数不支持下推到存储层，导致出现巨大的中间结果集累积。此时可能需要改写业务 SQL，或使用hint 进行调优，来使用可下推的函数或算子。
- 执行计划中存在算子 HashAgg。HashAgg 是多线程并发执行，虽然执行速度较快，但会消耗较多内存。可以尝试使用 `STREAM_AGG()` 替代。
- 调小同时读取的Region 的数量，或降低算子并发度，以避免因高并发导致的内存问题。对应的系统变量包括：
  - `tidb_distsql_scan_concurrency`
  - `tidb_index_serial_scan_concurrency`
  - `tidb_executor_concurrency`
- 问题发生时间附近，session 的并发度过高，此时可能需要添加节点进行扩容。

大事务或大写入消耗太多内存

需要提前进行内存的容量规划，这是因为执行事务时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。

针对单个大事务，可以通过拆分的方式调小事务大小。

收集和加载统计信息的过程中消耗太多内存

TiDB 节点启动后需要加载统计信息到内存中。统计信息的收集过程会消耗内存，可以通过以下方式控制内存使用量：

- 使用指定采样率、指定只收集特定列的统计信息、减少 ANALYZE 并发度等手段减少内存使用。
- TiDB v6.1.0 开始引入了系统变量tidb_stats_cache_mem_quota，可以对统计信息的内存使用进行限制。
- TiDB v6.1.0 开始引入了系统变量tidb_mem_quota_analyze，用于控制 TiDB 更新统计信息时的最大总内存占用。

更多信息请参见统计信息简介。

预处理语句(Prepared Statement) 使用过量

客户端不断创建预处理语句但未执行deallocate prepare stmt 会导致内存持续上涨，最终触发TiDB OOM。原因是预处理语句占用的内存要在 session 关闭后才会释放。这一点在长连接下尤需注意。

要解决该问题，可以考虑采取以下措施：

- 调整 session 的生命周期。
- 调整连接池的 wait_timeout 和 max_execution_time 时长。
- 使用系统变量max_prepared_stmt_count 进行限制。

系统变量配置不当

系统变量tidb_enable_rate_limit_action 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），启动该变量可能会导致内存不受tidb_mem_quota_query 控制，加剧 OOM 风险。

建议关闭该变量。从TiDB v6.3.0 开始，该变量默认关闭。

7.2.2.3.3 客户端问题

若客户端发生 OOM，则需要排查以下方面：

- 观察 Grafana TiDB Details > Server > Client Data Traffic 的趋势和速度，查看是否存在网络阻塞。
- 检查是否存在错误的JDBC 配置参数导致的应用 OOM。例如流式读取的相关参数 defaultFetchSize 配置有误，会造成数据在客户端大量缓存。

7.2.2.4 处理 OOM 问题需要收集的诊断信息

为定位 OOM 故障，通常需要收集以下信息：

- 操作系统的内存相关配置：

  - TiUP 上的配置：`resource_control.memory_limit`
  - 操作系统的配置：
    * 内存信息：`cat /proc/meminfo`
    * 相关内核参数：`vm.overcommit_memory`
  - NUMA 相关信息：
    * `numactl --hardware`
    * `numactl --show`

- 数据库的版本和内存相关配置：
  - TiDB 版本
  - `tidb_mem_quota_query`
  - `memory-usage-alarm-ratio`
  - `mem-quota-query`
  - `oom-action`
  - `tidb_enable_rate_limit_action`
  - `tidb_server_memory_limit`
  - `oom-use-tmp-storage`
  - `tmp-storage-path`
  - `tmp-storage-quota`
  - `tidb_analyze_version`
- 在 Grafana 查看 TiDB 内存的日常使用情况：TiDB > Server > Memory Usage
- 查看内存消耗较多的 SQL 语句：
  - 可以从 TiDB Dashboard 中查看 SQL 语句分析、慢查询，查看内存使用量
  - 查看 `INFORMATION_SCHEMA` 中的 `SLOW_QUERY` 和 `CLUSTER_SLOW_QUERY`
  - 各个 TiDB 节点的 `tidb_slow_query.log`
  - 执行 `grep "expensive_query" tidb.log` 查看对应的日志条目
  - 执行 `EXPLAIN ANALYZE` 查看算子的内存消耗
  - 执行 `SELECT * FROM information_schema.processlist;` 查看 SQL 对应的 `MEM` 列的值
- 执行以下命令收集内存使用率高的时候 TiDB 的 Profile 信息：

```
curl -G http://{TiDBIP}:10080/debug/zip?seconds=10" > profile.zip
```

- 执行 `grep "tidb-server has the risk of OOM" tidb.log` 查看 TiDB Server 收集的告警文件路径，例如：

```
["tidb-server has the risk of OOM. Running SQLs and heap profile will be recorded in record
    ↪ path"] ["is tidb_server_memory_limit set"=false] ["system memory total"=14388137984]
    ↪  ["system memory usage"=11897434112] ["tidb-server memory usage"=11223572312] [
    ↪ memory-usage-alarm-ratio=0.8] ["record path"="/tmp/0_tidb/
    ↪ MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-storage/record"]
```

7.2.2.5 探索更多

- TiDB 内存调优
- TiKV 内存调优

7.2.3 TiDB 热点问题处理

本文介绍如何定位和解决读写热点问题。

TiDB 作为分布式数据库，内建负载均衡机制，尽可能将业务负载均匀地分布到不同计算或存储节点上，更好地利用上整体系统资源。然而，机制不是万能的，在一些场景下仍会有部分业务负载不能被很好地分散，影响性能，形成单点的过高负载，也称为热点。

TiDB 提供了完整的方案用于排查、解决或规避这类热点。通过均衡负载热点，可以提升整体性能，包括提高 QPS 和降低延迟等。

7.2.3.1 常见热点场景

7.2.3.1.1 TiDB 编码规则回顾

TiDB 对每个表分配一个 TableID，每一个索引都会分配一个 IndexID，每一行分配一个 RowID（默认情况下，如果表使用整数型的 Primary Key，那么会用 Primary Key 的值当做 RowID）。其中 TableID 在整个集群内唯一，IndexID/RowID 在表内唯一，这些ID 都是 int64 类型。

每行数据按照如下规则进行编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_recordPrefixSep{rowID}
Value: [col1, col2, col3, col4]
```

其中 Key 的 `tablePrefix` 和 `recordPrefixSep` 都是特定的字符串常量，用于在 KV 空间内区分其他数据。

对于Index 数据，会按照如下规则编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: rowID
```

Index 数据还需要考虑 Unique Index 和非 Unique Index 两种情况，对于 Unique Index，可以按照上述编码规则。但是对于非 Unique Index，通过这种编码并不能构造出唯一的 Key，因为同一个 Index 的 `tablePrefix{tableID}_indexPrefixSep{indexID}` 都一样，可能有多行数据的 `ColumnsValue` 是一样的，所以对于非 Unique Index 的编码做了一点调整：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue_rowID
Value: null
```

7.2.3.1.2 表热点

从 TiDB 编码规则可知，同一个表的数据会在以表ID 开头为前缀的一个range 中，数据的顺序按照RowID 的值顺序排列。在表insert 的过程中如果RowID 的值是递增的，则插入的行只能在末端追加。当Region 达到一定的大小之后会进行分裂，分裂之后还是只能在range 范围的末端追加，永远只能在一个Region 上进行insert 操作，形成热点。

常见的increment 类型自增主键就是顺序递增的，默认情况下，在主键为整数型时，会用主键值当做RowID，此时 RowID 为顺序递增，在大量insert 时形成表的写入热点。

同时，TiDB 中RowID 默认也按照自增的方式顺序递增，主键不为整数类型时，同样会遇到写入热点的问题。

此外，当写入或读取数据存在热点时，即出现新建表或分区的写入热点问题和只读场景下周期性读热点问题时，你可以使用表属性控制Region 合并。具体的热点场景描述和解决方法可以查看使用表属性控制Region 合并的使用场景。

7.2.3.1.3 索引热点

索引热点与表热点类似，常见的热点场景出现在时间顺序单调递增的字段，或者插入大量重复值的场景。

7.2.3.2 确定存在热点问题

性能问题不一定是热点造成的，也可能存在多个因素共同影响，在排查前需要先确认是否与热点相关。

- 判断写热点依据：打开监控面板TiKV-Trouble-Shooting 中 Hot Write 面板，观察Raftstore CPU 监控是否存在个别 TiKV 节点的指标明显高于其他节点的现象。
- 判断读热点依据：打开监控面板TIKV-Details 中 Thread_CPU，查看 coprocessor cpu 有没有明显的某个TiKV 特别高。

7.2.3.3 使用TiDB Dashboard 定位热点表

TiDB Dashboard 中的流量可视化功能可帮助用户缩小热点排查范围到表级别。以下是流量可视化功能展示的一个热力图样例，该图横坐标是时间，纵坐标是各个表和索引，颜色越亮代表其流量越大。可在工具栏中切换显示读或写流量。

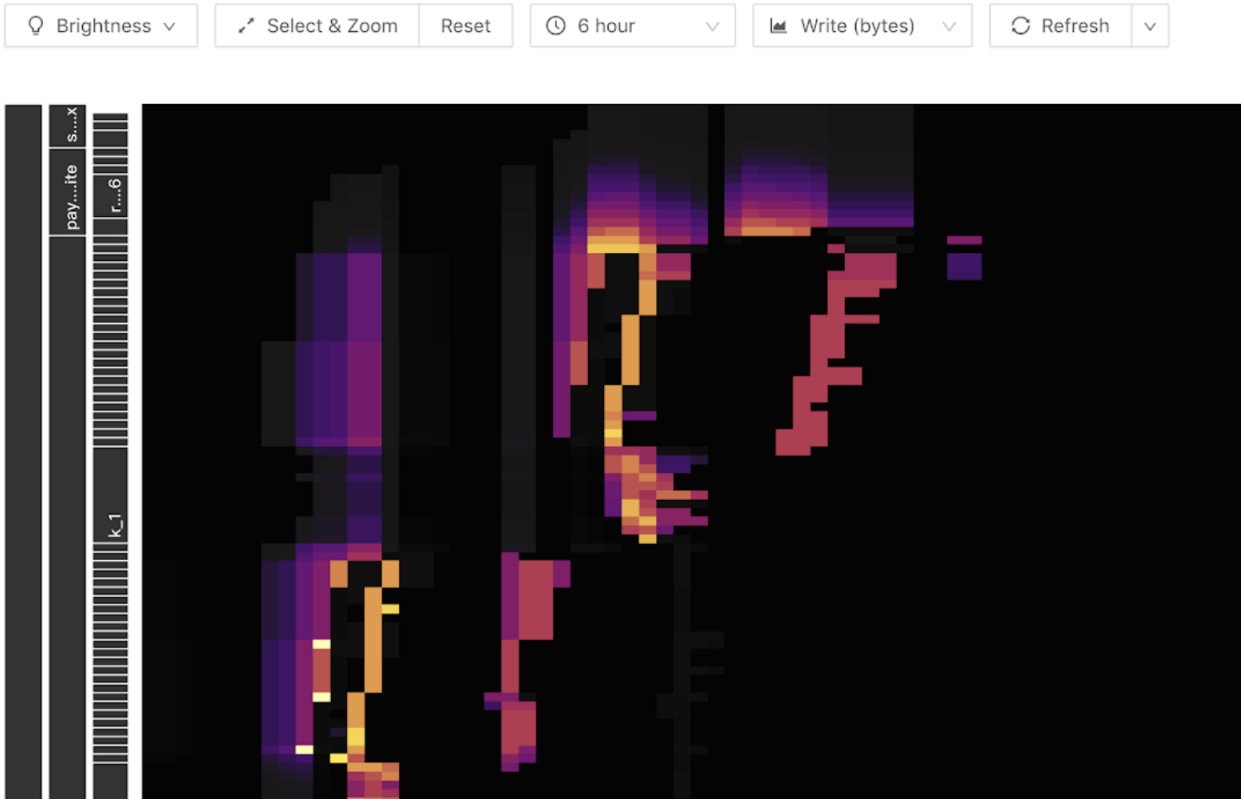


图 50: Dashboard 示例 1

当图中写入流量图出现以下明亮斜线（斜向上或斜向下）时，由于写入只出现在末端，随着表Region 数量变多，呈现出阶梯状。此时说明该表构成了写入热点：

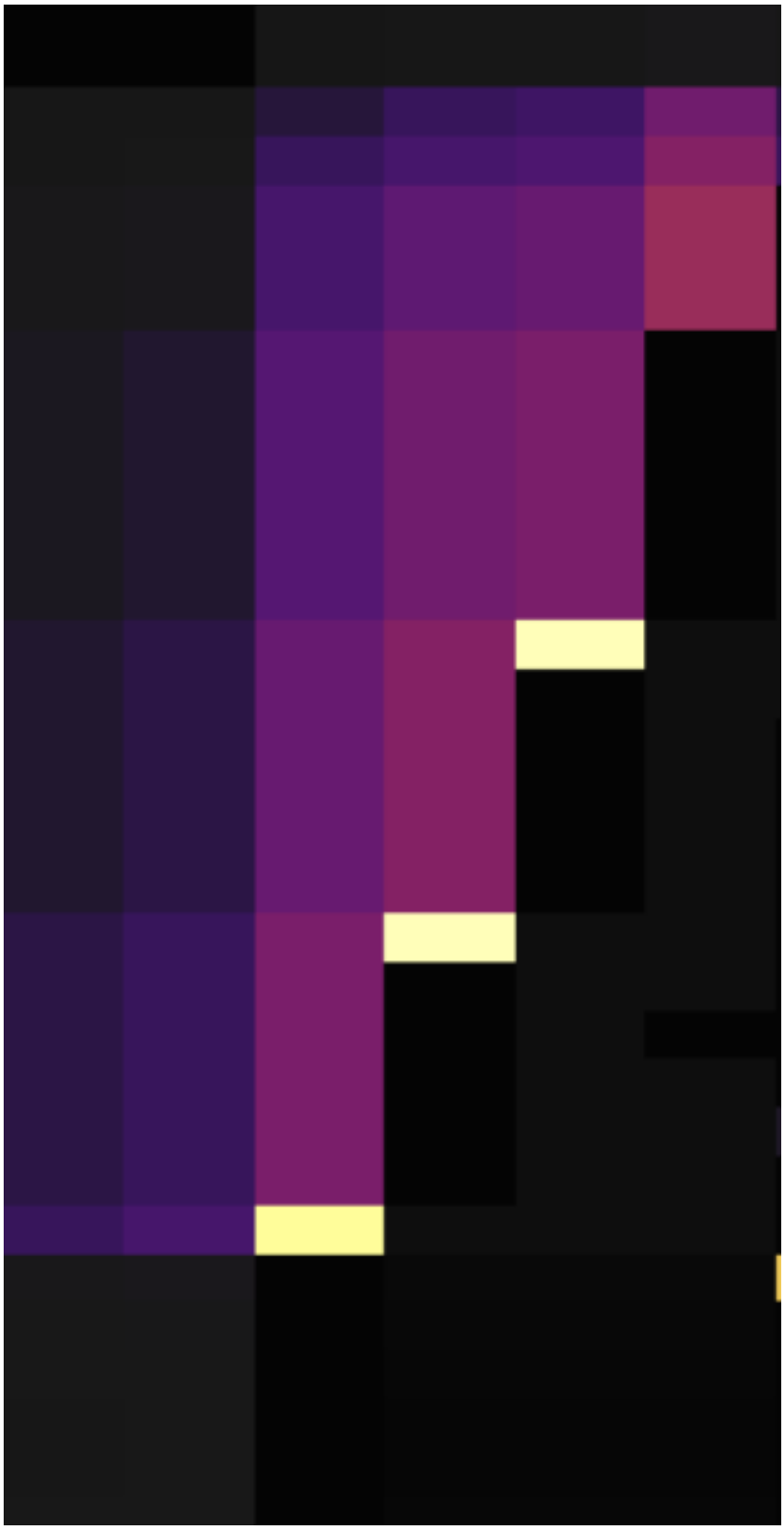


图 51: Dashboard 示例 2

对于读热点，在热力图中一般表现为一条明亮的横线，通常是有大量访问的小表，如下图所示：

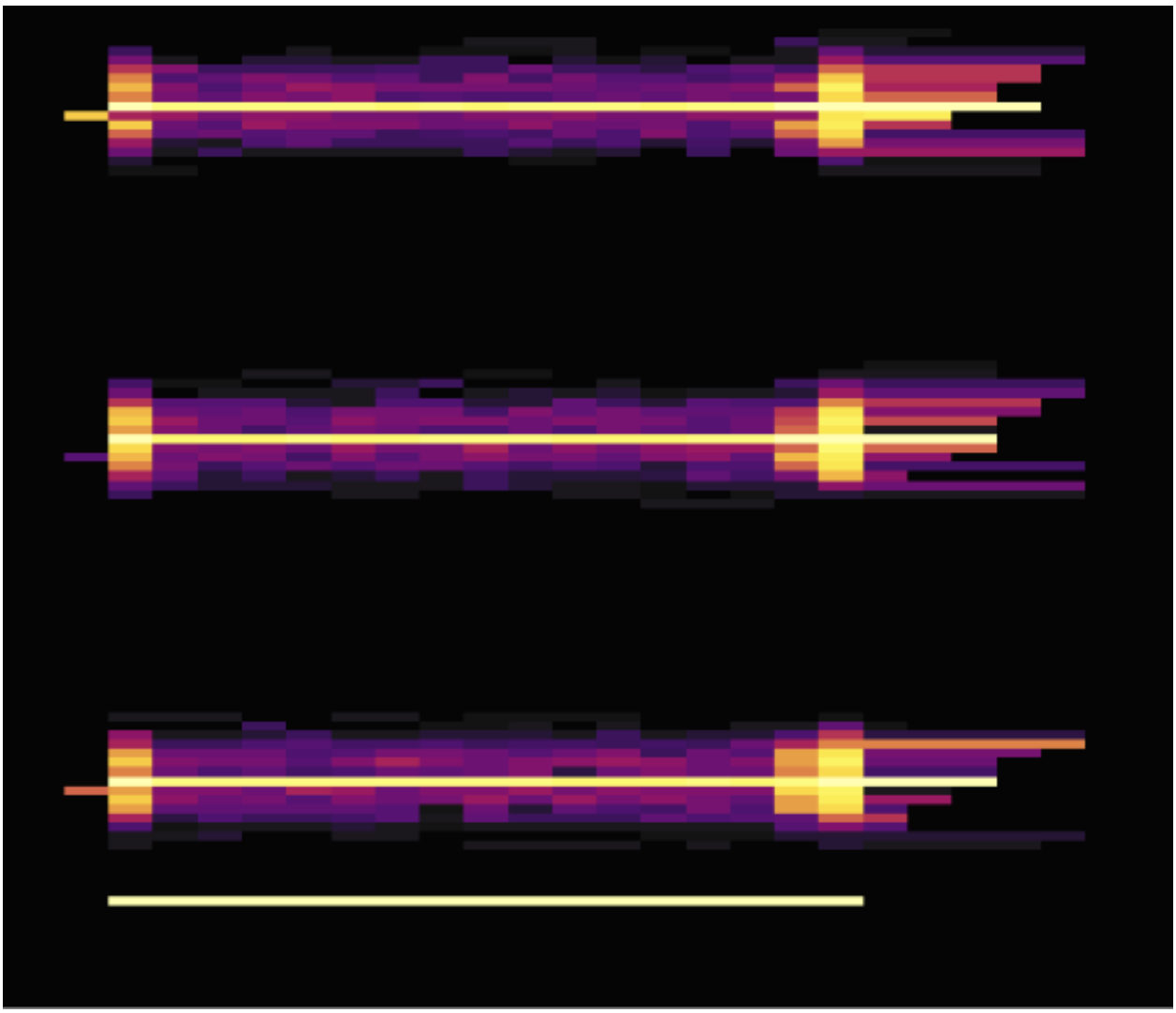


图 52: Dashboard 示例 3

将鼠标移到亮色块上，即可看到是什么表或索引具有大流量，如下图所示：

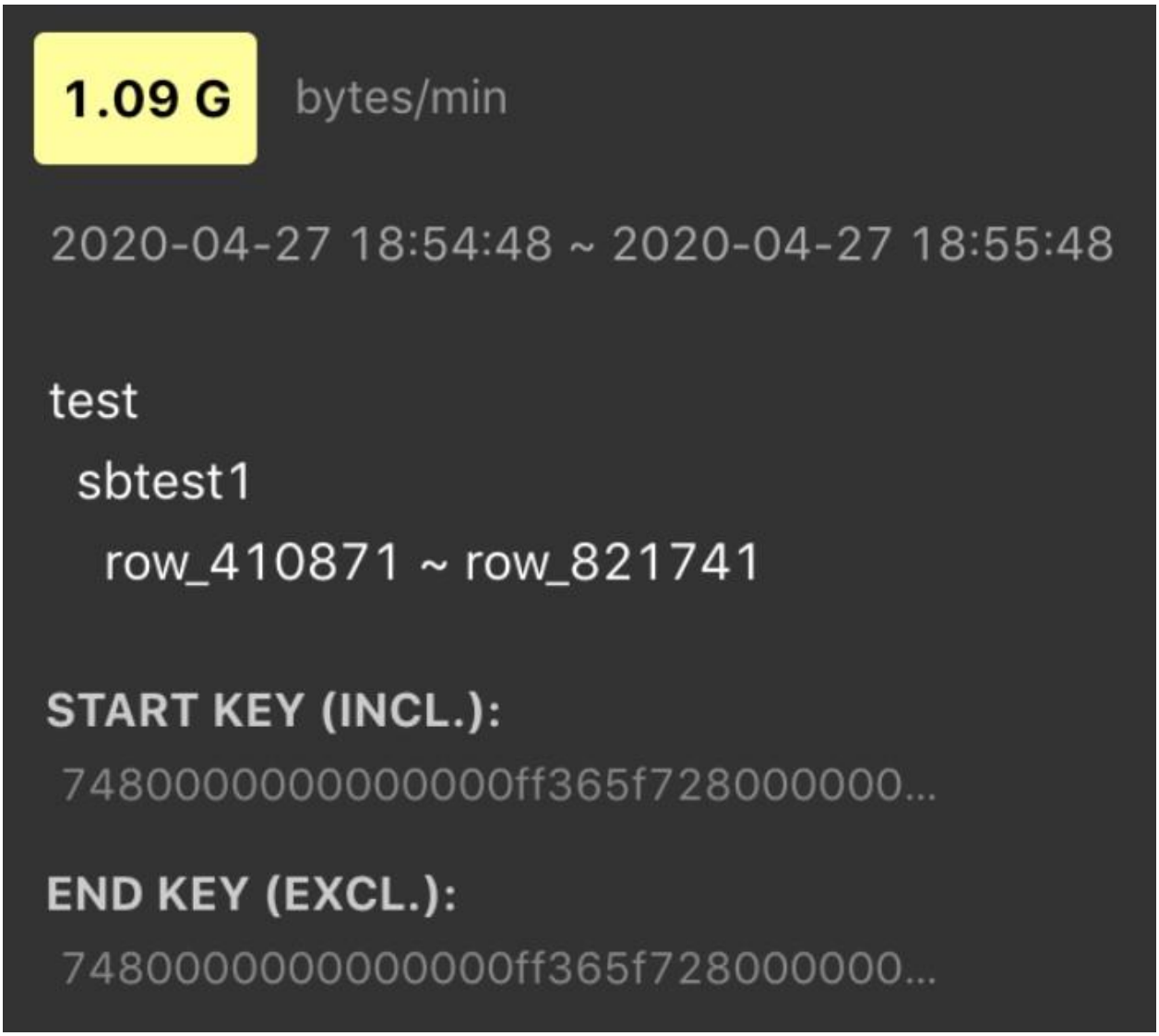


图 53: Dashboard 示例 4

7.2.3.4 使用 SHARD_ROW_ID_BITS 处理热点表

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 INSERT 时会把数据集中写入单个Region，造成写入热点。

通过设置 SHARD_ROW_ID_BITS，可以把 RowID 打散写入多个不同的Region，缓解写入热点问题。

```
SHARD_ROW_ID_BITS = 4 表示 16 个分片
SHARD_ROW_ID_BITS = 6 表示 64 个分片
SHARD_ROW_ID_BITS = 0 表示默认值 1 个分片
```

语句示例：

```
CREATE TABLE: CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4;
```

```
ALTER TABLE: ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

SHARD_ROW_ID_BITS 的值可以动态修改，每次修改之后，只对新写入的数据生效。

对于含有 CLUSTERED 主键的表，TiDB 会使用表的主键作为RowID，因为 SHARD_ROW_ID_BITS 会改变RowID 生成规则，所以此时无法使用 SHARD_ROW_ID_BITS 选项。而对于使用 NONCLUSTERED 主键的表，TiDB 会使用自动分配的 64 位整数作为RowID，此时也可以使用 SHARD_ROW_ID_BITS 特性。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

以下是两张无主键情况下使用 SHARD_ROW_ID_BITS 打散热点后的流量图，第一张展示了打散前的情况，第二张展示了打散后的情况。

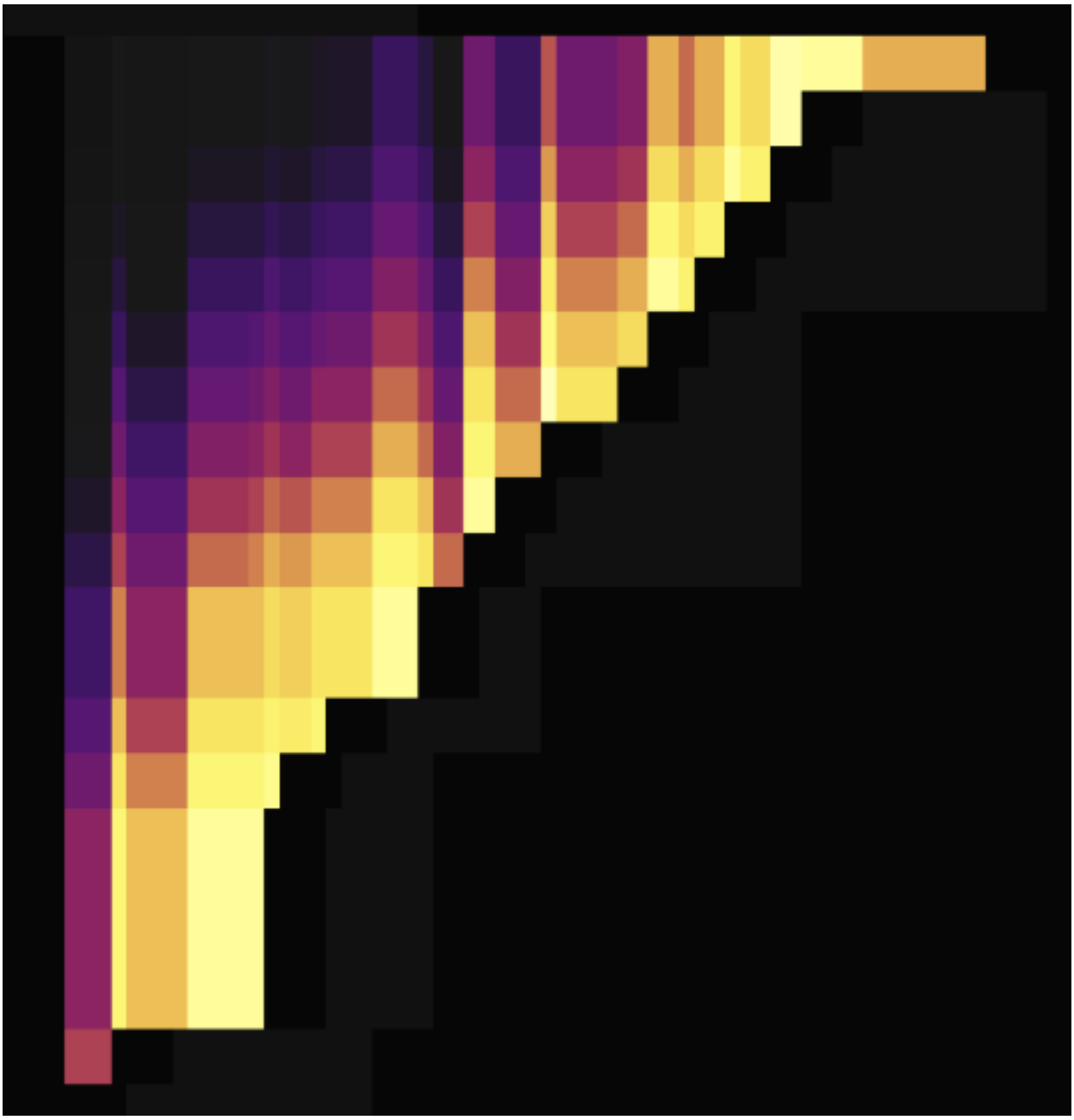


图 54: Dashboard 示例 5

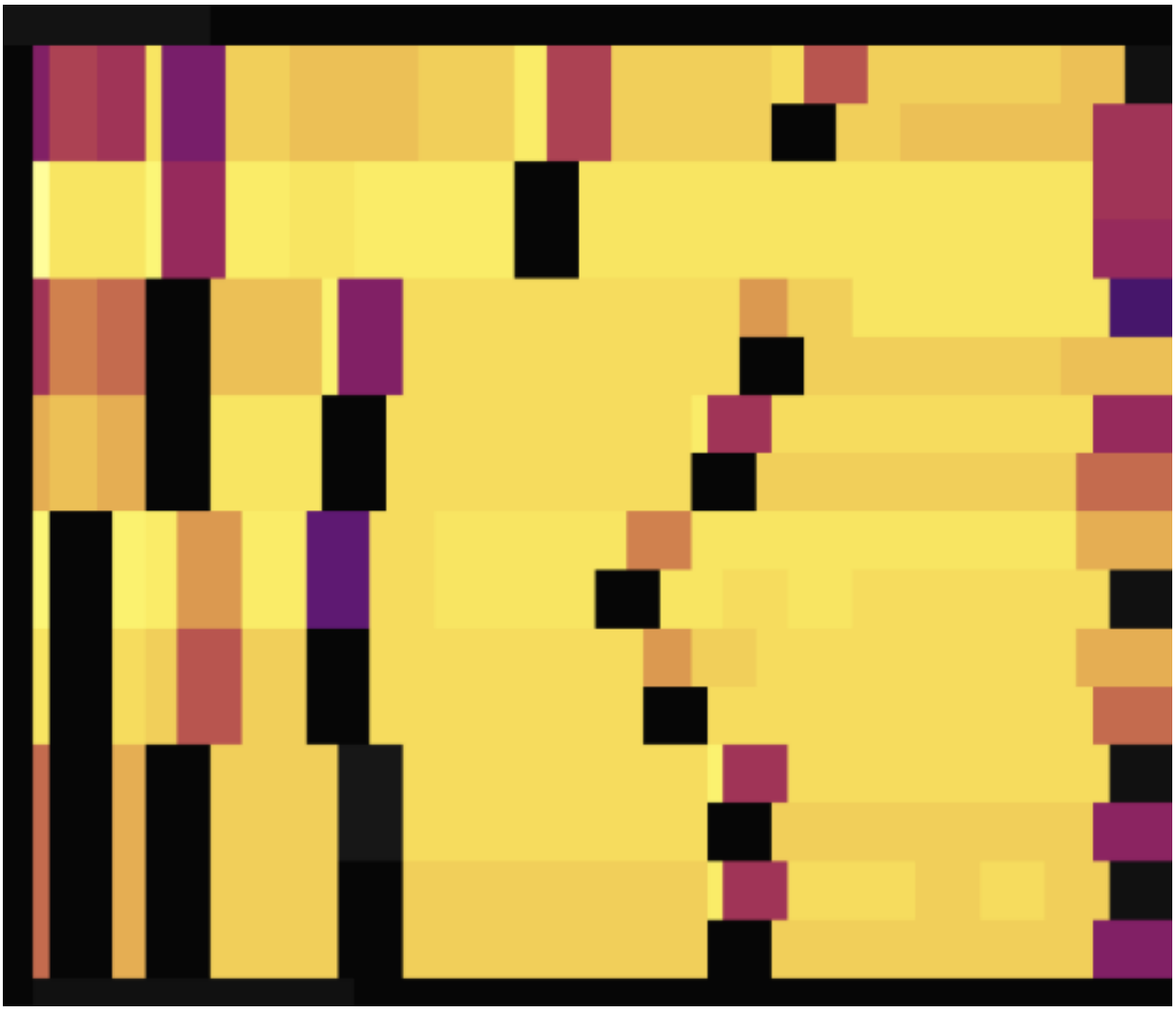


图 55: Dashboard 示例 6

从流量图可见，设置 SHARD_ROW_ID_BITS 后，流量热点由之前的只在一个Region 上变得很分散。

7.2.3.5 使用AUTO_RANDOM 处理自增主键热点表

使用 AUTO_RANDOM 处理自增主键热点表，适用于代替自增主键，解决自增主键带来的写入热点。

使用该功能后，将由TiDB 生成随机分布且空间耗尽前不重复的主键，达到离散写入、打散写入热点的目的。

注意TiDB 生成的主键不再是自增的主键，可使用 LAST_INSERT_ID() 获取上次分配的主键值。

将建表语句中的 AUTO_INCREMENT 改为 AUTO_RANDOM 即可使用该功能，适用于主键只需要保证唯一，不包含业务意义的场景。示例如下：

```
CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b varchar(255));
INSERT INTO t (b) VALUES ("foo");
SELECT * FROM t;
```

```
+-------------+---+
| a           | b |
+-------------+---+
| 1073741825 | b |
+-------------+---+
```

```
SELECT LAST_INSERT_ID();
```

```
+------------------------------ +
| LAST_INSERT_ID() |
+------------------------------ +
| 1073741825        |
+------------------------------ +
```

以下是将 AUTO_INCREMENT 表改为 AUTO_RANDOM 打散热点后的流量图，第一张是 AUTO_INCREMENT，第二张是 AUTO_RANDOM。

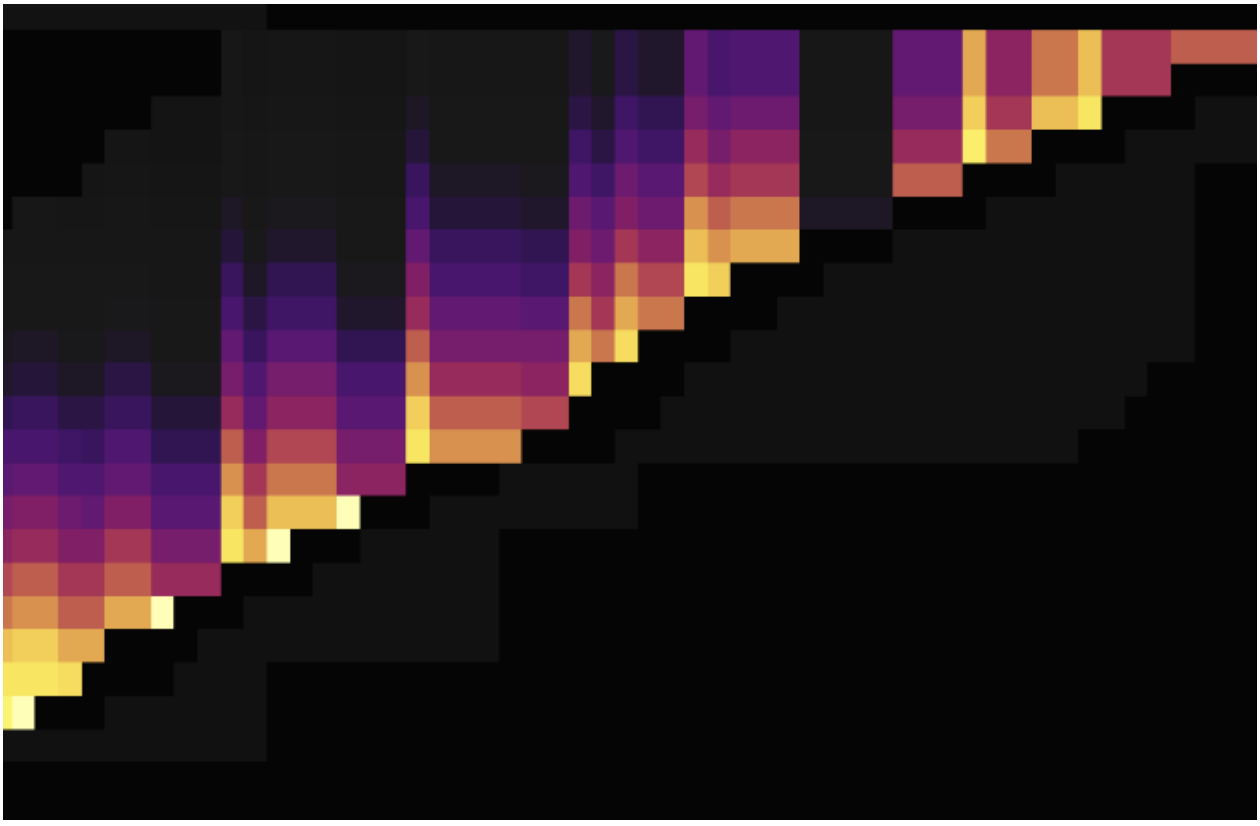


图 56: Dashboard 示例 7

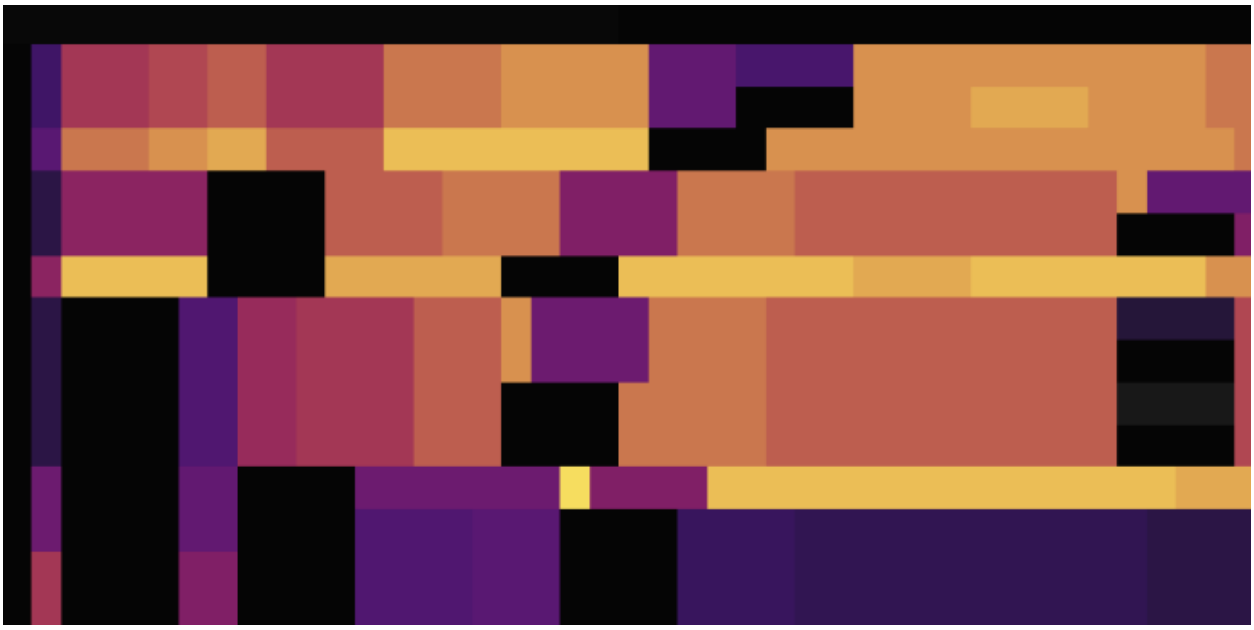


图 57: Dashboard 示例 8

由流量图可见，使用 AUTO_RANDOM 代替 AUTO_INCREMENT 能很好地打散热点。

更详细的说明参见 AUTO_RANDOM 文档。

7.2.3.6 小表热点的优化

TiDB 的 Coprocessor Cache 功能支持下推计算结果缓存。开启该功能后，将在TiDB 实例侧缓存下推给TiKV 计算的结果，对于小表读热点能起到比较好的效果。

更详细的说明参见下推计算结果缓存文档。

其他相关资料：

- TiDB 高并发写入场景最佳实践
- Split Region 使用文档

7.2.3.7 打散读热点

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%。

7.2.4 读写延迟增加

本文档介绍读写延迟增加、抖动时的排查思路，可能的原因和解决方法。

7.2.4.1 常见原因

7.2.4.1.1 TiDB 执行计划不对导致延迟增高

查询语句的执行计划不稳定，偶尔执行计划选择错误的索引，导致查询延迟增加。

现象：

- 如果慢日志中输出了执行计划，可以直接查看执行计划。用 `select tidb_decode_plan('xxx...')` 语句可以解析出具体的执行计划。
- 监控中的 key 扫描异常升高；慢日志中 SQL 执行时间 `Scan Keys` 数目较大。
- SQL 执行时间相比于其他数据库（例如 MySQL）有较大差距。可以对比其他数据库执行计划，例如 `Join Order` 是否不同。

可能的原因：

- 统计信息不准确

解决方案：

- 更新统计信息
  - 手动 `analyze table`，配合 crontab 定期 `analyze`，维持统计信息准确度。
  - 自动 `auto analyze`，调低 `analyze ratio` 阈值，提高收集频次，并设置运行时间窗口。示例如下：
    * `set global tidb_auto_analyze_ratio=0.2;`
    * `set global tidb_auto_analyze_start_time='00:00 +0800';`
    * `set global tidb_auto_analyze_end_time='06:00 +0800';`
- 绑定执行计划
  - 修改业务 SQL，使用 `use index` 固定使用列上的索引。
  - 3.0 版本下，业务可以不用修改 SQL，使用 `create global binding` 创建 `force index` 的绑定 SQL。
  - 4.0 版本支持 SQL Plan Management，可以避免因执行计划不稳定导致的性能下降。

7.2.4.1.2 PD 异常

现象：

监控中 PD TSO 的 wait duration 异常升高。wait duration 代表从开始等待 PD 返回，到等待结束的时间。

可能的原因：

- 磁盘问题。PD 所在的节点I/O 被占满，排查是否有其他I/O 高的组件与 PD 混合部署以及磁盘的健康情况，可通过监控Grafana -> disk performance -> latency 和 load 等指标进行验证，必要时可以使用 fio 工具对盘进行检测，见案例case-292。
- PD 之间的网络问题。PD 日志中有 `"lost the TCP streaming connection"`，排查 PD 之间网络是否有问题，可通过监控Grafana -> PD -> etcd 的 round trip 来验证，见案例case-177。
- 系统负载高，日志中能看到 `"server is likely overloaded"`，见案例 case-214。

- 选举不出leader。PD 日志中有 `"lease is not expired"`，见 issues https://github.com/etcd-io/etcd/issues/10355。v3.0.x 版本和 v2.1.19 版本已解决该问题，见案例case-875。
- 选举慢。Region 加载时间长，从PD 日志中`grep "regions cost"`（例如日志中可能是`load 460927 regions cost 11.77099s`），如果出现秒级，则说明较慢，v3.0 版本可开启Region Storage（设置 `use-region-storage` 为 `true`），该特性能极大缩短加载 Region 的时间，见案例case-429。
- TiDB 与 PD 之间的网络问题，应排查网络相关情况。通过监控 Grafana -> blackbox_exporter -> ping latency 确定 TiDB 到 PD leader 的网络是否正常。
- PD 报 FATAL 错误，日志中有 `"range failed to find revision pair"`。v3.0.8 中已经解决问题，见 PR https://github.com/pingcap/pd/pull/2040。详情请参考案例case-947。
- 使用 `/api/v1/regions` 接口时 Region 数量过多可能会导致 PD OOM。已于 v3.0.8 版本修复，见 https://github.com/pingcap/pd/pull/1986。
- 滚动升级的时候 PD OOM，gRPC 消息大小没限制，监控可看到TCP InSegs 较大，已于 v3.0.6 版本修复，见 https://github.com/pingcap/pd/pull/1952。
- PD panic，请提交bug。
- 其他原因，通过 `curl http://127.0.0.1:2379/debug/pprof/goroutine?debug=2` 抓取 goroutine，并提交bug。

7.2.4.1.3 TiKV 异常

现象：

监控中 KV Cmd Duration 异常升高。KV Cmd Duration 是 TiDB 发送请求给TiKV 到收到回复的延迟。

可能的原因：

- 查看 gRPC duration。gRPC duration 是请求在TiKV 端的总耗时。通过对比TiKV 的 gRPC duration 以及TiDB 中的 KV duration 可以发现潜在的网络问题。比如 gRPC duration 很短但是TiDB 的 KV duration 显示很长，说明TiDB 和 TiKV 之间网络延迟可能很高，或者TiDB 和 TiKV 之间的网卡带宽被占满。
- TiKV 重启了导致重新选举
  - TiKV panic 之后又被 `systemd` 重新拉起正常运行，可以通过查看TiKV 的日志来确认是否有 `panic`，这种情况属于非预期，需要报bug。
  - 被第三者 `stop/kill`，被 `systemd` 重新拉起。查看 `dmesg` 和 `TiKV log` 确认原因。
  - TiKV 发生 OOM 导致重启了。
  - 动态调整 THP 导致hung 住，见案例case-500。
- 查看监控：Grafana -> TiKV-details -> errors 面板 `server is busy` 看到TiKV RocksDB 出现 write stall 导致发生重新选举。
- TiKV 发生网络隔离导致重新选举。
- `block-cache` 配置太大导致 OOM，在监控 Grafana -> TiKV-details 选中对应的 instance 之后查看 RocksDB 的 `block cache size` 监控来确认是否是该问题。同时请检查 `[storage.block-cache] capacity = # "1GB"` 参数是否设置合理，默认情况下TiKV 的 `block-cache` 设置为机器总内存的 45%。在容器化部署时需要显式指定该参数，因为TiKV 获取的是物理机的内存，可能会超出单个container 的内存限制。

- Coprocessor 收到大量大查询，返回的数据量太大，gRPC 发送速度跟不上 Coprocessor 向客户端输出数据的速度导致 OOM。可以通过检查监控：Grafana -> TiKV-details -> coprocessor overview 的 `response size` 是否超过 `network outbound` 流量来确认是否属于这种情况。

7.2.4.1.4 TiKV 单线程瓶颈

TiKV 中存在一些单线程线程，可能会成为瓶颈。

- 单个 TiKV Region 过多导致单个 gRPC 线程成为瓶颈（查看 Grafana -> TiKV-details -> `Thread CPU/gRPC CPU Per` `Thread` 监控），v3.x 以上版本可以开启 Hibernate Region 特性来解决，见案例case-612。
- v3.0 之前版本 Raftstore 单线程或者 Apply 单线程到达瓶颈（Grafana -> TiKV-details -> `Thread CPU/raft store` CPU 和 `Async apply CPU` 超过 `80%`），可以选择扩容 TiKV（v2.x 版本）实例或者升级到多线程模型的 v3.x 版本。

7.2.4.1.5 CPU Load 升高

现象：

CPU 资源使用到达瓶颈

可能的原因：

- 热点问题。
- 整体负载高，排查 TiDB 的 slow query 和 expensive query。对运行的 query 进行优化，如果缺索引就加索引，如果可以批量执行就批量执行。另一个方案是对集群进行扩容。

7.2.4.2 其它原因

7.2.4.2.1 集群维护

通常大多数的线上集群有 3 或 5 个 PD 节点，如果维护的主机上有 PD 组件，需要具体考虑节点是 leader 还是 follower，关闭 follower 对集群运行没有任何影响，关闭 leader 需要先切换，并在切换时有 3 秒左右的性能抖动。

7.2.4.2.2 少数派副本离线

TiDB 集群默认配置为 3 副本，每一个 Region 都会在集群中保存 3 份，它们之间通过 Raft 协议来选举 Leader 并同步数据。Raft 协议可以保证在数量小于副本数（注意：不是节点数）一半的节点挂掉或者隔离的情况下，仍然能够提供服务，并且不丢失任何数据。对于 3 副本集群，挂掉一个节点可能会导致性能抖动，可用性和正确性理论上不会受影响。

7.2.4.2.3 新增索引

由于创建索引在扫表回填索引的时候会消耗大量资源，甚至与一些频繁更新的字段会发生冲突导致正常业务受到影响。大表创建索引的过程往往会持续很长时间，所以要尽可能地平衡执行时间和集群性能之间的关系，比如选择非高频更新时间段。

参数调整：

目前主要使用 tidb_ddl_reorg_worker_cnt 和 tidb_ddl_reorg_batch_size 这两个参数来动态调整索引创建速度，通常来说它们的值越小对系统影响越小，但是执行时间越长。

一般情况下， 先将值保持为默认的 4 和 256， 观察集群资源使用情况和响应速度， 再逐渐调大 tidb_ddl_reorg_worker_cnt 参数来增加并发， 观察监控如果系统没有发生明显的抖动， 再逐渐调大 tidb_ddl_reorg_batch_size 参数，但如果索引涉及的列更新很频繁的话就会造成大量冲突造成失败重试。

另外还可以通过调整参数 tidb_ddl_reorg_priority 为 PRIORITY_HIGH 来让创建索引的任务保持高优先级来提升速度，但在通用 OLTP 系统上，一般建议保持默认。

7.2.4.2.4 GC 压力大

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。GC 的任务便是清理不再需要的旧数据。

- Resolve Locks 阶段在TiKV 一侧会产生大量的scan_lock 请求，可以在gRPC 相关的metrics 中观察到。scan_lock 请求会对全部的Region 调用。
- Delete Ranges 阶段会往TiKV 发送少量的 unsafe_destroy_range 请求，也可能没有。可以在 gRPC 相关的 metrics 中和GC 分类下的GC tasks 中观察到。
- Do GC 阶段，默认每台TiKV 会自动扫描本机上的leader Region 并对每一个leader 进行GC，这一活动可以在 GC 分类下的GC tasks 中观察到。

7.2.5 乐观事务模型下写写冲突问题排查

本文介绍TiDB 中乐观锁下写写冲突出现的原因以及解决方案。

在 v3.0.8 版本之前，TiDB 默认采用乐观事务模型，在事务执行过程中并不会做冲突检测，而是在事务最终 COMMIT 提交时触发两阶段提交，并检测是否存在写写冲突。当出现写写冲突，并且开启了事务重试机制，则 TiDB 会在限定次数内进行重试，最终重试成功或者达到重试次数上限后，会给客户端返回结果。因此，如果 TiDB 集群中存在大量的写写冲突情况，容易导致集群的 Duration 比较高。

7.2.5.1 出现写写冲突的原因

TiDB 中使用 Percolator 事务模型来实现TiDB 中的事务。Percolator 总体上就是一个二阶段提交的实现。具体的二阶段提交过程可参考乐观事务文档。

当客户端发起 COMMIT 请求的时候，TiDB 开始两阶段提交：

1. TiDB 从所有要写入的 Key 中选择一个作为当前事务的 Primary Key
2. TiDB 向所有的本次提交涉及到的TiKV 发起prewrite 请求，TiKV 判断是否所有 Key 都可以prewrite 成功
3. TiDB 收到所有 Key 都 prewrite 成功的消息
4. TiDB 向 PD 请求commit_ts
5. TiDB 向 Primary Key 发起第二阶段提交。Primary Key 所在的TiKV 收到commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁
6. TiDB 收到两阶段提交成功的信息

写写冲突发生在prewrite 阶段，当发现有其他的事务在写当前 Key (data.commit_ts > txn.start_ts)，则会发生写写冲突。

TiDB 会根据 `tidb_disable_txn_auto_retry` 和 `tidb_retry_limit` 参数设置的情况决定是否进行重试，如果设置了不重试，或者重试次数达到上限后还是没有prewrite 成功，则向TiDB 返回 `Write Conflict` 错误。

7.2.5.2 如何判断当前集群存在写写冲突

可以通过Grafana 监控查看集群写写冲突的情况：

- 通过TiDB 监控面板中 KV Errors 监控栏中 KV Backoff OPS 监控指标项，查看TiKV 中返回错误信息的数量

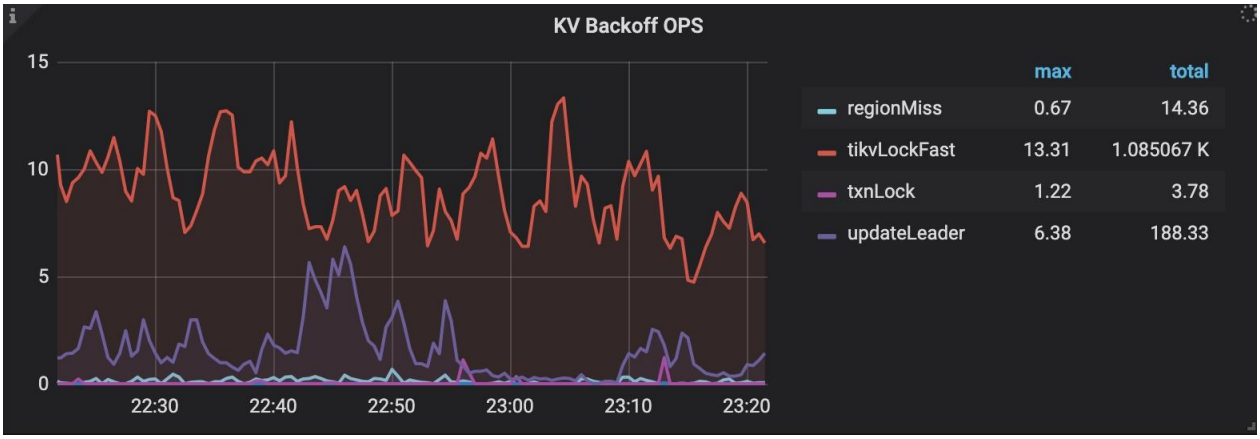


图 58: kv-backoff-ops

txnlock 表示集群中存在写写冲突，txnLockFast 表示集群中存在读写冲突。

- 通过TiDB 监控面板中 KV Errors 监控栏中 Lock Resolve OPS 监控指标项，查看事务冲突相关的数量

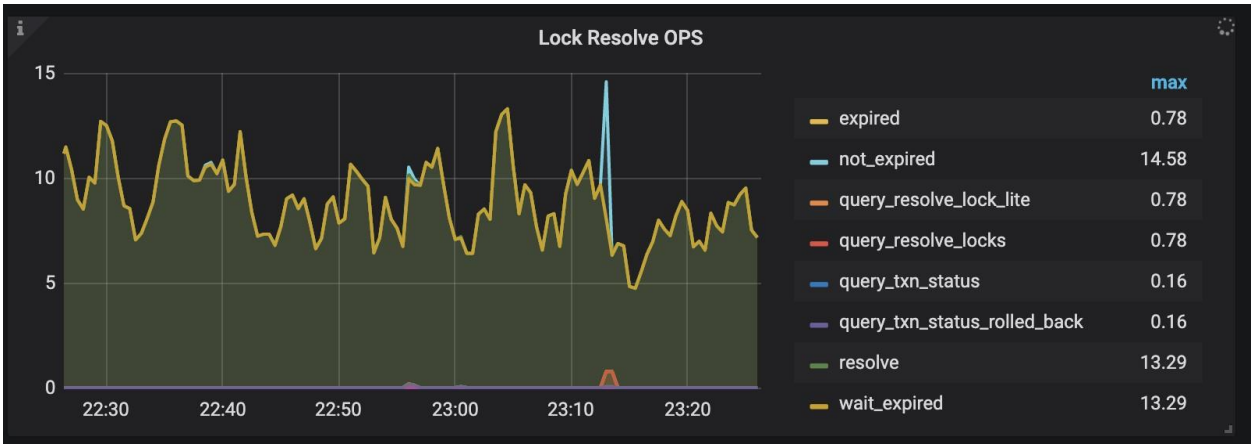


图 59: lock-resolve-ops

expired、 not_expired、 wait_expired 表示对应的lock 状态

- 通过TiDB 监控面板中 KV Errors 监控栏中 KV Retry Duration 监控指标项，查看 KV 重试请求的时间

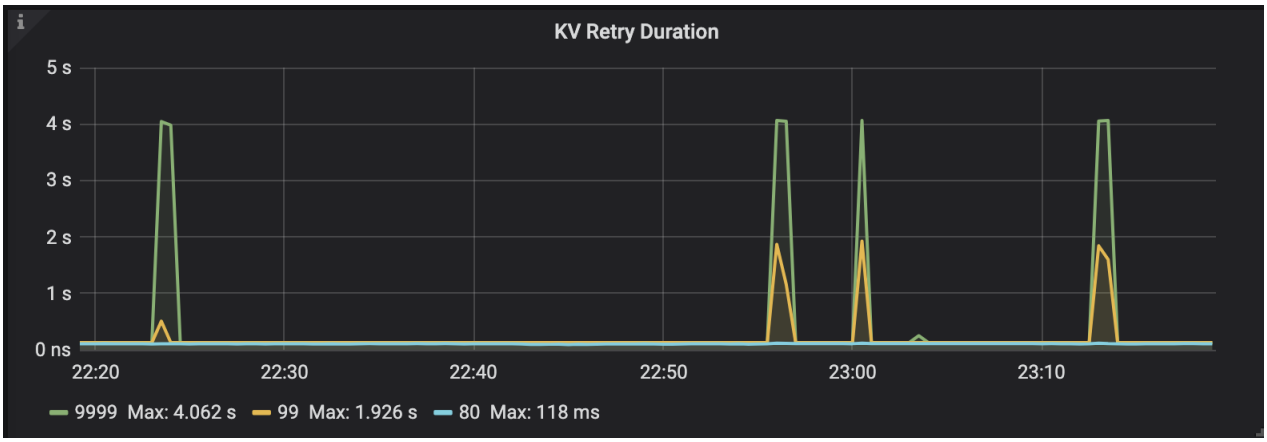


图 60: kv-retry-duration

也可以通过TiDB 日志查看是否有 `[kv:9007]Write conflict` 关键字，如果搜索到对应关键字，则可以表明集群中存在写写冲突。

7.2.5.3 如何解决写写冲突问题

如果通过以上方式判断出集群中存在大量的写写冲突，建议找到冲突的数据，以及写写冲突的原因，看是否能从应用程序修改逻辑，加上重试的逻辑。当出现写写冲突的时候，可以在TiDB 日志中看到类似的日志：

```
[2020/05/12 15:17:01.568 +08:00] [WARN] [session.go:446] ["commit failed"] [conn=3] ["finished
    ′→ txn"="Txn{state=invalid}"] [error="[kv:9007]Write conflict, txnStartTS
    ′→ =416617006551793665, conflictStartTS=416617018650001409, conflictCommitTS
    ′→ =416617023093080065, key={tableID=47, indexID=1, indexValues={string, }} primary={tableID
    ′→ =47, indexID=1, indexValues={string, }} [try again later]"]
```

关于日志的解释如下：

- `[kv:9007]Write conflict`：表示出现了写写冲突
- `txnStartTS=416617006551793665`：表示当前事务的 start_ts 时间戳，可以通过pd-ctl 工具将时间戳转换为具体时间
- `conflictStartTS=416617018650001409`：表示冲突事务的 start_ts 时间戳，可以通过pd-ctl 工具将时间戳转换为具体时间
- `conflictCommitTS=416617023093080065`：表示冲突事务的 commit_ts 时间戳，可以通过pd-ctl 工具将时间戳转换为具体时间
- `key={tableID=47, indexID=1, indexValues={string, }}`：表示当前事务中冲突的数据，tableID 表示发生冲突的表的ID，indexID 表示是索引数据发生了冲突。如果是数据发生了冲突，会打印 `handle=x` 表示对应哪行数据发生了冲突，indexValues 表示发生冲突的索引数据
- `primary={tableID=47, indexID=1, indexValues={string, }}`：表示当前事务中的 Primary Key 信息

通过pd-ctl 将时间戳转换为可读时间：

```
tiup ctl:v<CLUSTER_VERSION> pd -u https://127.0.0.1:2379 tso {TIMESTAMP}
```

通过tableID 查找具体的表名：

```
curl http://{TiDBIP}:10080/db-table/{tableID}
```

通过indexID 查找具体的索引名：

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_INDEXES WHERE TABLE_SCHEMA='{db_name}' AND TABLE_NAME='{
    ↪ table_name}' AND INDEX_ID={indexID};
```

另外在 v3.0.8 及之后版本默认使用悲观事务模式，从而避免在事务提交的时候因为冲突而导致失败，无需修改应用程序。悲观事务模式下会在每个 DML 语句执行的时候，加上悲观锁，用于防止其他事务修改相同 Key，从而保证在最后提交的prewrite 阶段不会出现写写冲突的情况。

7.2.6 TiDB 磁盘I/O 过高的处理办法

本文主要介绍如何定位和处理TiDB 存储I/O 过高的问题。

7.2.6.1 确认当前I/O 指标

当出现系统响应变慢的时候，如果已经排查了 CPU 的瓶颈、数据事务冲突的瓶颈后，就需要从I/O 来入手来辅助判断目前的系统瓶颈点。

7.2.6.1.1 从监控定位I/O 问题

最快速的定位手段是从监控来查看整体的I/O 情况，可以从集群部署工具 (TiUP) 默认会部署的监控组件Grafana 来查看对应的I/O 监控，跟I/O 相关的 Dashboard 有 `Overview`，`Node_exporter`，`Disk-Performance`。

第一类面板

在 `Overview` > `System Info` > `IO Util` 中，可以看到集群中每个机器的I/O 情况，该指标和 Linux iostat 监控中的 util 类似，百分比越高代表磁盘I/O 占用越高：

- 如果监控中只有一台机器的I/O 高，那么可以辅助判断当前有读写热点。
- 如果监控中大部分机器的I/O 都高，那么集群现在有高I/O 负载存在。

如果发现某台机器的I/O 比较高，可以从监控 `Disk-Performance Dashboard` 进一步观察I/O 的使用情况，结合 `Disk Latency`，`Disk Load` 等 metric 判断是否存在异常，必要时可以使用 fio 工具对磁盘进行检测。

第二类面板

TiDB 集群主要的持久化组件是 TiKV 集群，一个 TiKV 包含两个 RocksDB 实例：一个用于存储 Raft 日志，位于 data/raft，一个用于存储真正的数据，位于data/db。

在 `TiKV-Details` > `Raft IO` 中，可以看到这两个实例磁盘写入的相关metric：

- `Append log duration`：该监控表明了存储 Raft 日志的RocksDB 写入的响应时间，99% 的响应应该在 50ms 以内。
- `Apply log duration`：该监控表明了存储真正数据的 RocksDB 写入的响应时间，99% 的响应应该在 100ms 以内。

这两个监控还有 .. per server 的监控面板来提供辅助查看热点写入的情况。

第三类面板

在 TiKV-Details > Storage 中，有关于 storage 相关情况的监控：

- Storage command total：收到的不同命令的个数。
- Storage async write duration：包括了磁盘 sync duration 等监控项，可能和Raft IO 有关。如遇到异常情况，需要通过log 来检查相关组件的工作状态是否正常。

其他面板

此外，可能还需要一些其它内容来辅助确认瓶颈是否为I/O，并可以尝试调整一些参数。通过查看 TiKV gRPC 的 prewrite/commit/raw-put（仅限raw kv 集群）duration，确认确实是TiKV 写入慢了。常见的几种情况如下：

- append log 慢。TiKV Grafana 的 Raft I/O 和 append log duration 比较高，通常情况下是由于写盘慢了，可以检查 RocksDB - raft 的 WAL Sync Duration max 值来确认，否则可能需要报bug。
- raftstore 线程繁忙。TiKV grafana 的 Raft Propose/propose wait duration 明显高于append log duration。请查看以下两点：
  - [raftstore] 的 store-pool-size 配置是否过小（该值建议在 [1,5] 之间，不建议太大）。
  - 机器的 CPU 是不是不够了。
- apply log 慢。TiKV Grafana 的 Raft I/O 和 apply log duration 比较高，通常会伴随着Raft Propose/apply wait duration 比较高。可能的情况如下：
  - [raftstore] 的 apply-pool-size 配置过小（建议在 [1, 5] 之间，不建议太大），Thread CPU/apply cpu 比较高；
  - 机器的 CPU 资源不够了。
  - Region 写入热点问题，单个apply 线程CPU 使用率比较高（通过修改Grafana 表达式，加上by (instance, name) 来看各个线程的cpu 使用情况），暂时对于单个 Region 的热点写入没有很好的方式，最近在优化该场景。
  - 写 RocksDB 比较慢，RocksDB kv/max write duration 比较高（单个Raft log 可能包含很多个 kv，写 RocksDB 的时候会把 128 个 kv 放在一个 write batch 写入到RocksDB，所以一次 apply log 可能涉及到多次RocksDB 的 write）。
  - 其他情况，需要报bug。
- raft commit log 慢。TiKV Grafana 的 Raft I/O 和 commit log duration 比较高（4.x 版本的Grafana 才有该metric）。每个Region 对应一个独立的Raft group，Raft 本身是有流控机制的，类似TCP 的滑动窗口机制，通过参数 [raftstore] raft-max-inflight-msgs = 256 来控制滑动窗口的大小，如果有热点写入并且commit log duration 比较高可以适度调大改参数，比如 1024。

7.2.6.1.2 从 log 定位I/O 问题

- 如果客户端报 server is busy 错误，特别是 raftstore is busy 的错误信息，会和I/O 有相关性。

  可以通过查看监控：grafana -> TiKV -> errors 监控确认具体busy 原因。其中，server is busy 是 TiKV 自身的流控机制，TiKV 通过这种方式告知 tidb/ti-client 当前TiKV 的压力过大，过一会儿再尝试。

- TiKV RocksDB 日志出现 `write stall`。

  可能是 `level0 sst` 太多导致 stall。可以添加参数 `[rocksdb] max-sub-compactions = 2`（或者 3）加快 level0 sst 往下 compact 的速度，该参数的意思是将从level0 到 level1 的 compaction 任务最多切成 `max-sub-compactions` 个子任务交给多线程并发执行。

  如果磁盘I/O 能力持续跟不上写入，建议扩容。如果磁盘的吞吐达到了上限（例如 SATA SSD 的吞吐相对 NVME SSD 会低很多）导致 write stall，但是 CPU 资源又比较充足，可以尝试采用压缩率更高的压缩算法来缓解磁盘的压力，用 CPU 资源换磁盘资源。

  比如 `default cf compaction` 压力比较大时， 可以调整参数 `[rocksdb.defaultcf] compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]` 改成 `compression-per-level = ["no", "no", "zstd", "zstd", "zstd", "zstd", "zstd"]`。

7.2.6.1.3 从告警发现I/O 问题

集群部署工具(TiUP) 默认部署的告警组件，官方已经预置了相关的告警项目和阈值，I/O 相关项包括：

- TiKV_write_stall
- TiKV_raft_log_lag
- TiKV_async_request_snapshot_duration_seconds
- TiKV_async_request_write_duration_seconds
- TiKV_raft_append_log_duration_secs
- TiKV_raft_apply_log_duration_secs

7.2.6.2 I/O 问题处理方案

1. 当确认为热点I/O 问题的时候，需要参考TiDB 热点问题处理来消除相关的热点I/O 情况。
2. 当确认整体I/O 已经到达瓶颈的时候，且从业务侧能够判断I/O 的能力会持续的跟不上，那么就可以利用分布式数据库的 scale 的能力，采用扩容TiKV 节点数量的方案来获取更大的整体I/O 吞吐量。
3. 调整上述说明中的一些参数，使用计算/内存资源来换取磁盘的存储资源。

7.2.7 TiDB 锁冲突问题处理

TiDB 支持完整的分布式事务，自 v3.0 版本起，提供乐观事务与悲观事务两种事务模式。本文介绍如何使用 Lock View 排查锁相关的问题，以及如何处理使用乐观事务或者悲观事务的过程中常见的锁冲突问题。

7.2.7.1 使用 Lock View 排查锁相关的问题

自 v5.1 版本起，TiDB 支持 Lock View 功能。该功能在 `information_schema` 中内置了若干系统表，用于提供更多关于锁冲突和锁等待的信息。

> **注意：**
>
> Lock View 功能目前仅提供悲观锁的冲突和等待信息。

关于这些系统表的详细说明，请参考以下文档：

- TIDB_TRX 与 CLUSTER_TIDB_TRX：提供当前 TiDB 节点上或整个集群上所有运行中的事务的信息，包括事务是否处于等锁状态、等锁时间和事务曾经执行过的语句的 Digest 等信息。
- DATA_LOCK_WAITS：提供关于 TiKV 内的悲观锁等锁信息，包括阻塞和被阻塞的事务的 start_ts、被阻塞的 SQL 语句的 Digest 和发生等待的 key。
- DEADLOCKS 与 CLUSTER_DEADLOCKS：提供当前 TiDB 节点上或整个集群上最近发生过的若干次死锁的相关信息，包括死锁环中事务之间的等待关系、事务当前正在执行的语句的 Digest 和发生等待的 key。

> 注意：
>
> Lock View 所属的系统表中展示的 SQL 语句为归一化的 SQL 语句（即去除了格式和参数的 SQL 语句），通过内部查询从 SQL Digest 获得，因而无法获取包括格式和参数在内的完整语句。有关 SQL Digest 和归一化 SQL 语句的详细介绍，请参阅Statement Summary Tables。

以下为排查部分问题的示例。

7.2.7.1.1 死锁错误

要获取最近发生的死锁错误的信息，可查询 DEADLOCKS 或 CLUSTER_DEADLOCKS 表。

以查询 DEADLOCKS 表为例，请执行以下 SQL 语句：

```
select * from information_schema.deadlocks;
```

示例输出：

```
+--
    '->            +                              +           +                    +
    '->
| 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`
    '->  = ? ; | 748000000000000355F728000000000000002 | {"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`
    '->  = ? ; | 748000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    '-> :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
```

```
+--
    '-> -----------+----------------------------+-----------+--------------------+-------------------------------------
    '->
```

查询结果会显示死锁错误中多个事务之间的等待关系和各个事务当前正在执行的 SQL 语句的归一化形式（即去掉参数和格式的形式），以及发生冲突的 key 及其从 key 中解读出的一些信息。

例如在上述例子中，第一行意味着 ID 为 426812829645406216 的事务当前正在执行形如 `update `t` set `v` = '-> ? where `id` = ? ;` 的语句，被另一个 ID 为 426812829645406217 的事务阻塞；而 426812829645406217 同样也在执行一条形如 `update `t` set `v` = ? where `id` = ? ;` 的语句，并被ID 为 426812829645406216 的事务阻塞，两个事务因而构成死锁。

7.2.7.1.2 少数热点 key 造成锁排队

DATA_LOCK_WAITS 系统表提供TiKV 节点上的等锁情况。查询该表时，TiDB 将自动从所有TiKV 节点上获取当前时刻的等锁信息。当少数热点 key 频繁被上锁并阻塞较多事务时，你可以查询 DATA_LOCK_WAITS 表并按 key 对结果进行聚合，以尝试找出经常发生问题的 key：

```
select `key`, count(*) as `count` from information_schema.data_lock_waits group by `key` order by
    '->  `count` desc;
```

示例输出：

```
+----------------------------------------+-------+
| key                                    | count |
+----------------------------------------+-------+
| 748000000000000415F728000000000000001 |     2 |
| 748000000000000415F728000000000000002 |     1 |
+----------------------------------------+-------+
```

为避免偶然性，你可考虑进行多次查询。

如果已知频繁出问题的 key，可尝试从 TIDB_TRX 或 CLUSTER_TIDB_TRX 表中获取试图上锁该 key 的事务的信息。

需要注意 TIDB_TRX 和 CLUSTER_TIDB_TRX 表所展示的信息也是对其进行查询的时刻正在运行的事务的信息，并不展示已经结束的事务。如果并发的事务数量很大，该查询的结果集也可能很大，可以考虑添加 limit 子句，或用where 子句筛选出等锁时间较长的事务。需要注意，对 Lock View 中的多张表进行join 时，不同表之间的数据并不保证在同一时刻获取，因而不同表中的信息可能并不同步。

以用where 子句筛选出等锁时间较长的事务为例，请执行以下 SQL 语句：

```
select trx.* from information_schema.data_lock_waits as l left join information_schema.tidb_trx
    '-> as trx on l.trx_id = trx.id where l.key = "748000000000000415F728000000000000001"\G
```

示例输出：

```
*************************** 1. row ***************************
                    ID: 426831815660273668
            START_TIME: 2021-08-06 07:16:00.081000
    CURRENT_SQL_DIGEST: 06da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7
```

```
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ? ;
                  STATE: LockWaiting
     WAITING_START_TIME: 2021-08-06 07:16:00.087720
         MEM_BUFFER_KEYS: 0
        MEM_BUFFER_BYTES: 0
              SESSION_ID: 77
                    USER: root
                      DB: test
         ALL_SQL_DIGESTS: ["0fdc781f19da1c6078c9de7eadef8a307889c001e05f107847bee4cfc8f3cdf3","06
             ′→ da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7"]
*************************** 2. row ***************************
                     ID: 426831818019569665
             START_TIME: 2021-08-06 07:16:09.081000
     CURRENT_SQL_DIGEST: 06da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ? ;
                  STATE: LockWaiting
     WAITING_START_TIME: 2021-08-06 07:16:09.290271
         MEM_BUFFER_KEYS: 0
        MEM_BUFFER_BYTES: 0
              SESSION_ID: 75
                    USER: root
                      DB: test
         ALL_SQL_DIGESTS: ["0fdc781f19da1c6078c9de7eadef8a307889c001e05f107847bee4cfc8f3cdf3","06
             ′→ da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7"]
2 rows in set (0.00 sec)
```

7.2.7.1.3 事务被长时间阻塞

如果已知一个事务被另一事务（或多个事务）阻塞，且已知当前事务的 start_ts（即事务 ID），则可使用如下方式获取导致该事务阻塞的事务的信息。注意对 Lock View 中的多张表进行join 时，不同表之间的数据并不保证在同一时刻获取，因而可能不同表中的信息可能并不同步。

```
select l.key, trx.*, tidb_decode_sql_digests(trx.all_sql_digests) as sqls from information_schema
    ′→ .data_lock_waits as l join information_schema.cluster_tidb_trx as trx on l.
    ′→ current_holding_trx_id = trx.id where l.trx_id = 426831965449355272\G
```

示例输出：

```
*************************** 1. row ***************************
                   key: 7480000000000004D5F728000000000000001
              INSTANCE: 127.0.0.1:10080
                    ID: 426832040186609668
            START_TIME: 2021-08-06 07:30:16.581000
    CURRENT_SQL_DIGEST: 06da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ? ;
                STATE: LockWaiting
```

```
    WAITING_START_TIME: 2021-08-06 07:30:16.592763
        MEM_BUFFER_KEYS: 1
       MEM_BUFFER_BYTES: 19
            SESSION_ID: 113
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["0fdc781f19da1c6078c9de7eadef8a307889c001e05f107847bee4cfc8f3cdf3","
           ↪ a4e28cc182bdd18288e2a34180499b9404cd0ba07e3cc34b6b3be7b7c2de7fe9","06
           ↪ da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7"]
                  sqls: ["begin ;","select * from `t` where `id` = ? for update ;","update `t`
                      ↪ set `v` = `v` + ? where `id` = ? ;"]
1 row in set (0.01 sec)
```

上述查询中，对 CLUSTER_TIDB_TRX 表的 ALL_SQL_DIGESTS 列使用了TIDB_DECODE_SQL_DIGESTS 函数，目的是将该列（内容为一组 SQL Digest）转换为其对应的归一化 SQL 语句，便于阅读。

如果当前事务的 start_ts 未知，可以尝试从 TIDB_TRX / CLUSTER_TIDB_TRX 表或者 PROCESSLIST / CLUSTER_PROCESSLIST 表中的信息进行判断。

7.2.7.2 处理乐观锁冲突问题

以下介绍乐观事务模式下常见的锁冲突问题的处理方式。

7.2.7.2.1 读写冲突

在 TiDB 中，读取数据时，会获取一个包含当前物理时间且全局唯一递增的时间戳作为当前事务的 start_ts。事务在读取时，需要读到目标 key 的commit_ts 小于这个事务的 start_ts 的最新的数据版本。当读取时发现目标 key 上存在lock 时，因为无法知道上锁的那个事务是在 Commit 阶段还是 Prewrite 阶段，所以就会出现读写冲突的情况，如下图：

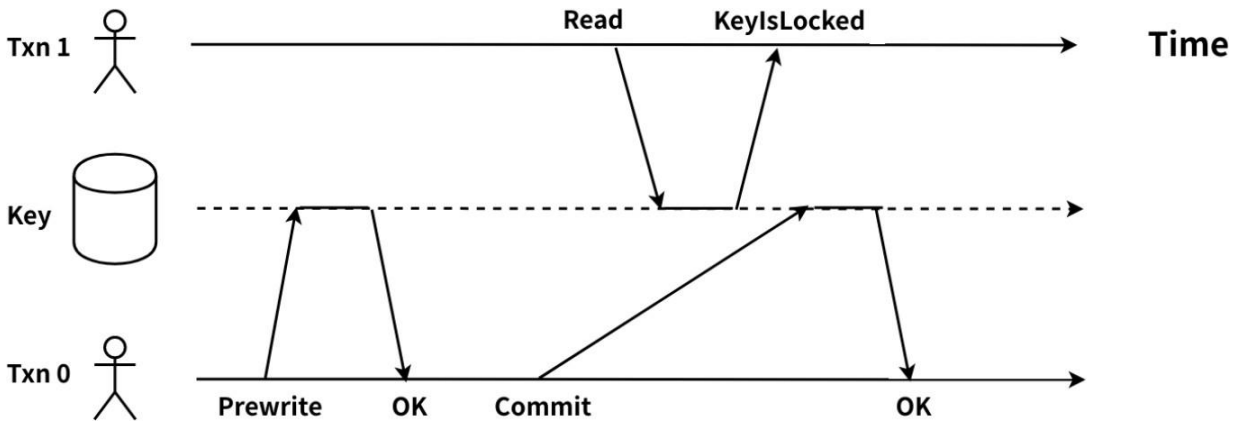


图 61: 读写冲突

分析：

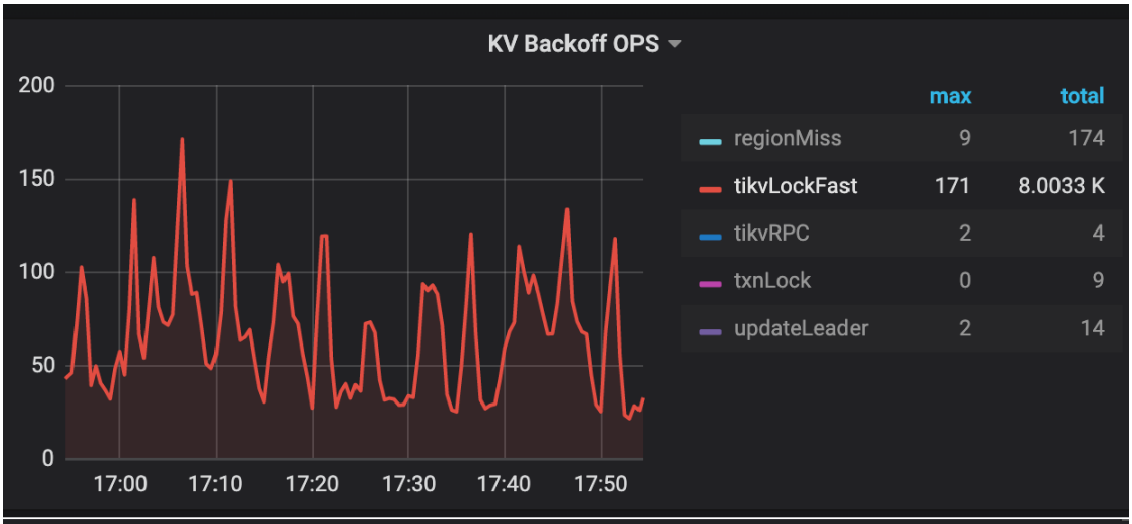
Txn0 完成了 Prewrite，在 Commit 的过程中Txn1 对该 key 发起了读请求，Txn1 需要读取 start_ts > commit_ts 最近的 key 的版本。此时，Txn1 的 `start_ts` > Txn0 的 lock_ts，需要读取的 key 上的锁信息仍未清理，故无法判断Txn0 是否提交成功，因此Txn1 与 Txn0 出现读写冲突。

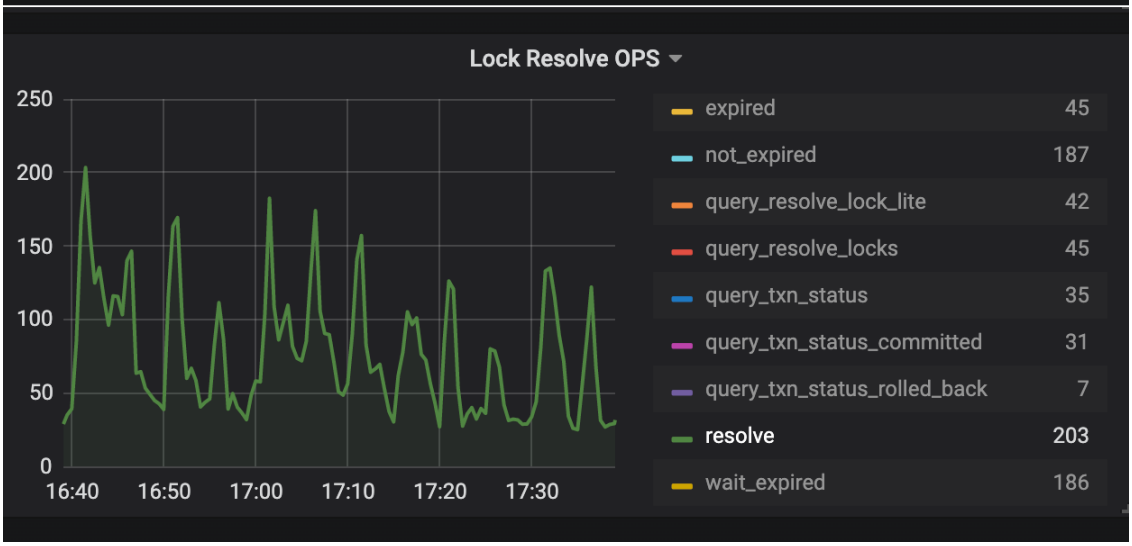
你可以通过如下两种途径来检测当前环境中是否存在读写冲突：

1. TiDB 监控及日志

    - 通过TiDB Grafana 监控分析：

      观察 KV Errors 下 Lock Resolve OPS 面板中的 not_expired/resolve 监控项以及 KV Backoff OPS 面板中的 tikvLockFast 监控项，如果有较为明显的上升趋势，那么可能是当前的环境中出现了大量的读写冲突。其中，not_expired 是指对应的锁还没有超时，resolve 是指尝试清锁的操作，tikvLockFast 代表出现了读写冲突。





    - 通过TiDB 日志分析：

      在TiDB 的日志中可以看到下列信息：

```
[INFO] [coprocessor.go:743] ["[TIME_COP_PROCESS] resp_time:406.038899ms txnStartTS
    ’→ :416643508703592451 region_id:8297 store_addr:10.8.1.208:20160 backoff_ms:255
    ’→ backoff_types:[txnLockFast,txnLockFast] kv_process_ms:333 scan_total_write:0
```

```
    ↪ scan_processed_write:0 scan_total_data:0 scan_processed_data:0 scan_total_lock:0
    ↪  scan_processed_lock:0"]
```

- txnStartTS：发起读请求的事务的 start_ts，如上面示例中的 416643508703592451
- backoff_types：读写发生了冲突，并且读请求进行了backoff 重试，重试的类型为txnLockFast
- backoff_ms：读请求backoff 重试的耗时，单位为ms，如上面示例中的 255
- region_id：读请求访问的目标region 的 id

2. 通过TiKV 日志分析：

   在TiKV 的日志可以看到下列信息：

```
[ERROR] [endpoint.rs:454] [error-response] [err=""locked primary_lock:7480000000000004
    ↪ D35F698000000000000001038000000000004C788E038000000000004C0748 lock_version:
    ↪ 411402933858205712 key: 7480000000000004D35F7280000000004C0748 lock_ttl: 3008
    ↪ txn_size: 1""]
```

   这段报错信息表示出现了读写冲突，当读数据时发现 key 有锁阻碍读，这个锁包括未提交的乐观锁和未提交的prewrite 后的悲观锁。

   - primary_lock：锁对应事务的primary lock。
   - lock_version：锁对应事务的 start_ts。
   - key：表示被锁的 key。
   - lock_ttl: 锁的 TTL。
   - txn_size：锁所在事务在其Region 的 key 数量，指导清锁方式。

处理建议：

- 在遇到读写冲突时会有backoff 自动重试机制，如上述示例中Txn1 会进行backoff 重试，单次初始 10 ms，单次最大 3000 ms，总共最大 20000 ms
- 可以使用TiDB Control 的子命令decoder 来查看指定 key 对应的行的table id 以及 rowid：

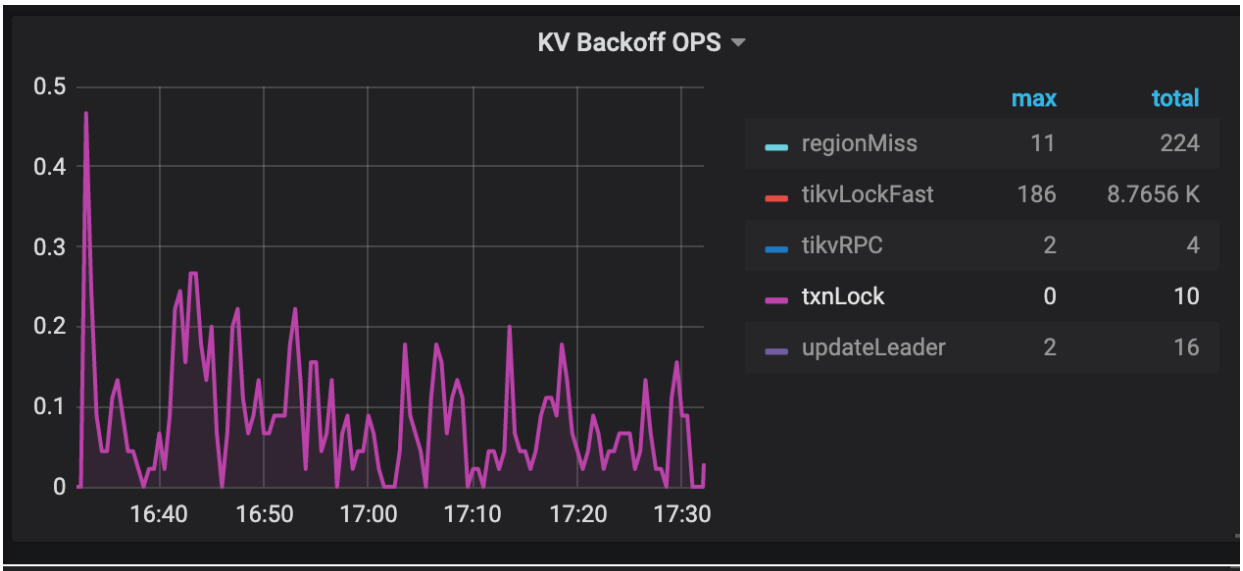
```
./tidb-ctl decoder "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\x00\x00\xfa"
format: table_row
table_id: -9223372036854775780
row_id: -9223372036854775558
```

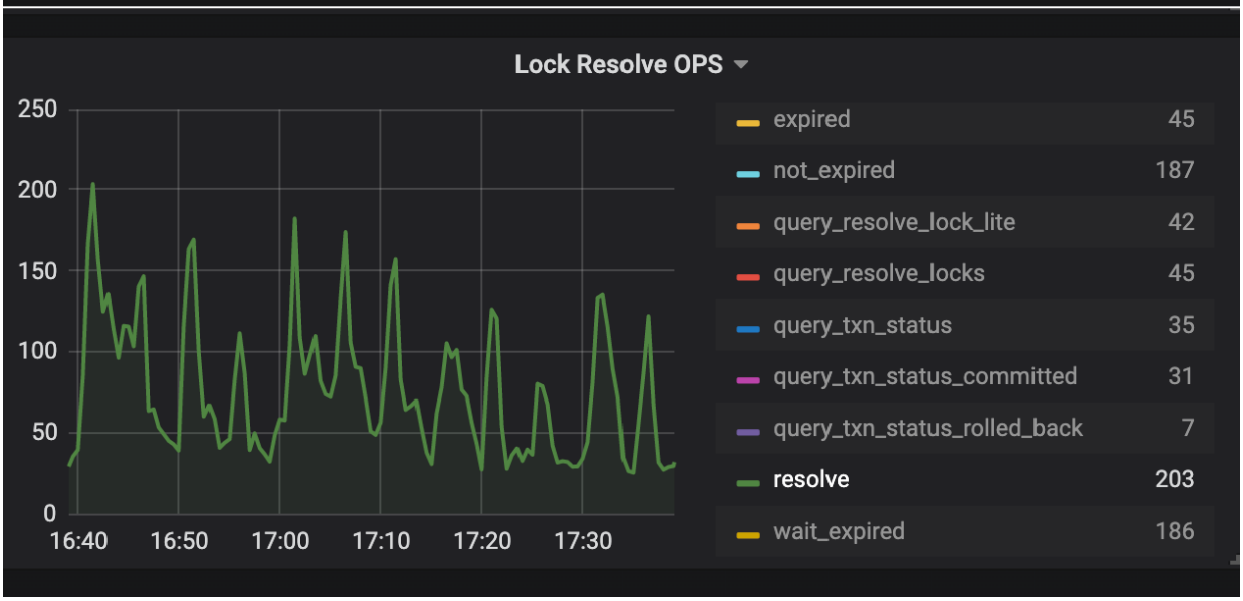
7.2.7.2.2 KeyIsLocked 错误

事务在 Prewrite 阶段的第一步就会检查是否有写写冲突，第二步会检查目标 key 是否已经被另一个事务上锁。当检测到该 key 被 lock 后，会在TiKV 端报出 KeyIsLocked。目前该报错信息没有打印到 TiDB 以及TiKV 的日志中。与读写冲突一样，在出现 KeyIsLocked 时，后台会自动进行backoff 重试。

你可以通过TiDB Grafana 监控检测 KeyIsLocked 错误：

观察 KV Errors 下 Lock Resolve OPS 面板中的resolve 监控项以及 KV Backoff OPS 面板中的txnLock 监控项，会有比较明显的上升趋势，其中resolve 是指尝试清锁的操作，txnLock 代表出现了写冲突。





处理建议：

- 监控中出现少量txnLock，无需过多关注。后台会自动进行 backoff 重试，单次初始 100 ms，单次最大 3000 ms。
- 如果出现大量的txnLock，需要从业务的角度评估下冲突的原因。
- 使用悲观锁模式。

7.2.7.2.3 锁被清除(LockNotFound) 错误

TxnLockNotFound 错误是由于事务提交的慢了，超过了 TTL 的时间。当要提交时，发现被其他事务给Rollback 掉了。在开启TiDB 自动重试事务的情况下，会自动在后台进行事务重试（注意显示和隐式事务的差别）。

你可以通过如下两种途径来查看 LockNotFound 报错信息：

1. 查看TiDB 日志

如果出现了TxnLockNotFound 的报错，会在TiDB 的日志中看到下面的信息：

```
[WARN] [session.go:446] ["commit failed"] [conn=149370] ["finished txn"="Txn{state=invalid
    ′→ }"] [error="[kv:6]Error: KV error safe to retry tikv restarts txn: Txn(Mvcc(
    ′→ TxnLockNotFound{ start_ts: 412720515987275779, commit_ts: 412720519984971777, key:
    ′→ [116, 128, 0, 0, 0, 0, 1, 111, 16, 95, 114, 128, 0, 0, 0, 0, 0, 0, 2] })) [try again
    ′→  later]"]
```

- start_ts：出现 TxnLockNotFound 报错的事务的 start_ts，如上例中的 412720515987275779
- commit_ts：出现TxnLockNotFound 报错的事务的commit_ts，如上例中的 412720519984971777

2. 查看TiKV 日志

   如果出现了TxnLockNotFound 的报错，在TiKV 的日志中同样可以看到相应的报错信息：

```
Error: KV error safe to retry restarts txn: Txn(Mvcc(TxnLockNotFound)) [ERROR [Kv.rs:708] ["
    ′→ KvService::batch_raft send response fail"] [err=RemoteStoped]
```

处理建议：

- 通过检查 start_ts 和 commit_ts 之间的提交间隔，可以确认是否超过了默认的 TTL 的时间。

  查看提交间隔：

```
tiup ctl:v<CLUSTER_VERSION> pd tso [start_ts]
tiup ctl:v<CLUSTER_VERSION> pd tso [commit_ts]
```

- 建议检查下是否是因为写入性能的缓慢导致事务提交的效率差，进而出现了锁被清除的情况。
- 在关闭TiDB 事务重试的情况下，需要在应用端捕获异常，并进行重试。

7.2.7.3 处理悲观锁冲突问题

以下介绍悲观事务模式下常见的锁冲突问题的处理方式。

> 注意：
>
> 即使设置了悲观事务模式，autocommit 事务仍然会优先尝试使用乐观事务模式进行提交，并在发生冲突后、自动重试时切换为悲观事务模式。

7.2.7.3.1 读写冲突

报错信息以及处理建议同乐观锁模式。

7.2.7.3.2 pessimistic lock retry limit reached

在冲突非常严重的场景下，或者当发生 write conflict 时，乐观事务会直接终止，而悲观事务会尝试用最新数据重试该语句直到没有 write conflict。因为TiDB 的加锁操作是一个写入操作，且操作过程是先读后写，需要 2 次 RPC。如果在这中间发生了 write conflict，那么会重试。每次重试都会打印日志，不用特别关注。重试次数由pessimistic-txn.max-retry-count 定义。

可通过查看TiDB 日志查看报错信息：

悲观事务模式下，如果发生 write conflict，并且重试的次数达到了上限，那么在TiDB 的日志中会出现含有下述关键字的报错信息。如下：

```
err="pessimistic lock retry limit reached"
```

处理建议：

- 如果上述报错出现的比较频繁，建议从业务的角度进行调整。
- 如果业务中包含对同一行（同一个 key）的高并发上锁而频繁冲突，可以尝试启用系统变量tidb_pessimistic_txn_fair_locking。需要注意启用该选项可能对存在锁冲突的事务带来一定程度的吞吐下降（平均延迟上升）的代价。对于新部署的集群，该选项默认启用(ON)。

7.2.7.3.3 Lock wait timeout exceeded

在悲观锁模式下，事务之间出现会等锁的情况。等锁的超时时间由TiDB 的innodb_lock_wait_timeout 参数来定义，这个是 SQL 语句层面的最大允许等锁时间，即一个 SQL 语句期望加锁，但锁一直获取不到，超过这个时间，TiDB 不会再尝试加锁，会向客户端返回相应的报错信息。

可通过查看TiDB 日志查看报错信息：

当出现等锁超时的情况时，会向客户端返回下述报错信息：

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

处理建议：

- 如果出现的次数非常频繁，建议从业务逻辑的角度来进行调整。

7.2.7.3.4 TTL manager has timed out

除了有不能超出GC 时间的限制外，悲观锁的 TTL 有上限，默认为 1 小时，所以执行时间超过 1 小时的悲观事务有可能提交失败。这个超时时间由TiDB 参数 performance.max-txn-ttl 指定。

可通过查看TiDB 日志查看报错信息：

当悲观锁的事务执行时间超过 TTL 时，会出现下述报错：

```
TTL manager has timed out, pessimistic locks may expire, please commit or rollback this
    ↪ transaction
```

处理建议：

- 当遇到该报错时，建议确认下业务逻辑是否可以进行优化，如将大事务拆分为小事务。在未使用大事务的前提下，大事务可能会触发TiDB 的事务限制。
- 可适当调整相关参数，使其符合事务要求。

7.2.7.3.5 Deadlock found when trying to get lock

死锁是指两个或两个以上的事务在执行过程中，由于竞争资源而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去，将永远在互相等待。此时，需要终止其中一个事务使其能够继续推进下去。

TiDB 在使用悲观锁的情况下，多个事务之间出现了死锁，必定有一个事务abort 来解开死锁。在客户端层面行为和 MySQL 一致，在客户端返回表示死锁的Error 1213。如下：

```
[err="[executor:1213]Deadlock found when trying to get lock; try restarting transaction"]
```

处理建议：

- 如果难以确认产生死锁的原因，对于 v5.1 及以后的版本，建议尝试查询 `INFORMATION_SCHEMA.DEADLOCKS` 或 `INFORMATION_SCHEMA.CLUSTER_DEADLOCKS` 系统表来获取死锁的等待链信息。详情请参考死锁错误小节和 `DEADLOCKS` 表文档。
- 如果死锁出现非常频繁，需要调整业务代码来降低发生概率。

7.2.8 数据索引一致性报错

当执行事务或执行 `ADMIN CHECK [TABLE|INDEX]` 命令时，TiDB 会对数据索引的一致性进行检查。如果检查发现record key-value 和index key-value 不一致，即存储行数据的键值对和存储其对应索引的键值对之间不一致（例如多索引或缺索引），TiDB 会报数据索引一致性错误，并在日志文件中打印相关错误日志。

本文对数据索引一致性的报错信息进行了说明，并提供了一些绕过检查的方法。遇到报错时，你可以前往 AskTUG 论坛，与社区用户交流；如果是订阅用户，请联系 PingCAP 服务与支持。

7.2.8.1 错误样例解读

当数据索引不一致时，你可以通过查看TiDB 的报错信息了解行数据和索引数据在哪一项不一致，或者查看相关错误日志进行判断。

7.2.8.1.1 执行事务中的报错

本节列出了TiDB 在执行事务过程中可能出现的数据索引不一致性的报错，并通过举例对这些信息的含义进行了解释。

Error 8133

```
ERROR 8133 (HY000): data inconsistency in table: t, index: k2, index-count:1 != record-count:0
```

上述错误表明，对于表t 中的 k2 索引，表中索引数量为 1，行记录的数量为 0，数量不一致。

Error 8138

```
ERROR 8138 (HY000): writing inconsistent data in table: t, expected-values:{KindString green} !=
↪ record-values:{KindString GREEN}
```

上述错误表明，事务试图写入的行值有误。即将写入的数据中，编码后的行数据与编码前的原始数据不符。

Error 8139

```
ERROR 8139 (HY000): writing inconsistent data in table: t, index: i1, index-handle:4 != record-
'→ handle:3, index: tables.mutation{key:kv.Key{0x74, 0x80, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x49, 0
'→ x5f, 0x69, 0x80, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1, 0x1, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0, 0
'→ x0, 0x0, 0xfc, 0x1, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0, 0x0, 0x0, 0xfc, 0x3, 0x80, 0x0, 0x0, 0
'→ x0, 0x0, 0x0, 0x0, 0x4}, flags:0x0, value:[]uint8{0x30}, indexID:1}, record:
tables.mutation{ '→ key:kv.Key{0x74, 0x80, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x49, 0x5f, 0x72, 0x80,
0x0, 0x0, 0x0, 0 '→ x0, 0x0, 0x0, 0x3}, flags:0xd, value:[]uint8{0x80, 0x0, 0x2, 0x0, 0x0, 0x0,
0x1, 0x2, 0x5, 0x0 '→ , 0xa, 0x0, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x68, 0x65, 0x6c, 0x6c, 0x6f},
indexID:0}
```

上述错误表明，即将写入的数据中，handle（即行数据的 key）值不一致。对于表 `t` 中的 `i1` 索引，该事务即将写入的某行在索引键值对中的handle 值是 4，在行记录键值对中的handle 值是 3。这行数据将不会被写入。

Error 8140

```
ERROR 8140 (HY000): writing inconsistent data in table: t, index: i2, col: c1, indexed-value:{
'→ KindString hellp} != record-value:{KindString hello}
```

上述错误表明，事务试图写入的行和索引的值不一致。对于表 `t` 中的 `i2` 索引，该事务即将写入的某行在索引键值对中的数据是 `hellp`，在行记录键值对中的数据是`hello`。这行数据将不会被写入。

Error 8141

```
ERROR 8141 (HY000): assertion failed: key: 74800000000000000405f7201330000000000000000f8, assertion
'→ : NotExist, start_ts: 430590532931813377, existing start ts: 430590532931551233, existing
'→ commit ts: 430590532931551234
```

上述错误表明， 事务提交时断言失败。根据数据索引一致的假设，TiDB 断言 key 7480000000000000405 '→ f7201330000000000000f8 不存在，提交事务时发现该 key 存在，是由 start ts 为 430590532931551233 的事务写入的。TiDB 会将该 key 的 MVCC (Multi-Version Concurrency Control) 历史输出到日志。

7.2.8.1.2 Admin check 中的报错

本节列出了执行 `ADMIN CHECK [TABLE|INDEX]` 系列语句时TiDB 可能出现的数据索引不一致报错，并通过举例对这些信息的含义进行了解释。

Error 8003

```
ERROR 8003 (HY000): table count 3 != index(idx)count 2
```

上述错误表明，在 `ADMIN CHECK` 语句所执行的表上有 3 个行键值对，但只有 2 个索引键值对。

Error 8134

```
ERROR 8134 (HY000): data inconsistency in table: t, index: c2, col: c2, handle: "2", index-values
'→ :"KindInt64 13" != record-values:"KindInt64 12", compare err:<nil>
```

上述错误表明，对于表`t` 中的 `c2` 索引，handle 为 2 的行对应的索引键值对中列`c2` 的值是 13，行记录键值对中列 `c2` 的值是 12。

Error 8223

```
ERROR 8223 (HY000): data inconsistency in table: t2, index: i1, handle: {hello, hello}, index-
'→ values:"" != record-values:"handle: {hello, hello}, values: [KindString hello KindString hello
'→ ]"
```

上述错误表明，index-values 为空，record-values 不为空，说明不存在对应的索引，但存在对应的行。

7.2.8.2 报错处理

发生报错时，不要自行处理，请从 PingCAP 官方或TiDB 社区获取支持。如果业务急需跳过此类报错，可以使用以下方法绕过检查。

7.2.8.2.1 改写 SQL

如果只有某一条 SQL 语句报错，可以尝试将其改写为其它等价的 SQL 形式，以使用不同的执行算子来尝试绕过。

7.2.8.2.2 关闭错误检查

对于事务执行中出现的一些报错，可以使用以下方法绕过检查：

- 对于错误代码为 8138、8139 和 8140 的错误，可以通过设置 `set @@tidb_enable_mutation_checker=0` 跳过检查。
- 对于错误代码为 8141 的错误，可以通过设置 `set @@tidb_txn_assertion_level=OFF` 跳过检查。

> **注意：**
>
> 关闭 tidb_enable_mutation_checker 和 tidb_txn_assertion_level 开关会关闭对所有 SQL 语句的对应检查。

对于其它错误代码，包括执行 `ADMIN CHECK [TABLE|INDEX]` 系列语句或执行事务中的报错，由于数据中已经存在不一致，无法跳过对应的检查。

7.3 故障诊断方法

7.3.1 SQL 诊断

SQL 诊断功能是在TiDB 4.0 版本中引入的特性，用于提升TiDB 问题定位的效率。TiDB 4.0 版本以前，用户需要使用不同的工具以异构的方式获取不同信息。新的 SQL 诊断系统对这些离散的信息进行了整体设计，它整合系统各个维度的信息，通过系统表的方式向上层提供一致的接口，提供监控汇总与自动诊断，方便用户查询集群信息。

SQL 诊断共分三大块：

- 集群信息表：TiDB 4.0 诊断系统添加了集群信息表，为原先离散的各实例信息提供了统一的获取方式。它将整个集群的集群拓扑、硬件信息、软件信息、内核参数、监控、系统信息、慢查询、语句、日志完全整合在表中，让用户能够统一使用 SQL 进行查询。

- 集群监控表：TiDB 4.0 诊断系统添加了集群监控系统表，所有表都在 `metrics_schema` 中，可以通过 SQL 语句来查询监控信息。比起原先的可视化监控，SQL 查询监控允许用户对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。由于TiDB 集群的监控指标数量较大，SQL 诊断还提供了监控汇总表，让用户能够更便捷地从众多监控中找出异常的监控项。
- 自动诊断：尽管用户可以手动执行 SQL 来查询集群信息表、集群监控表与汇总表来定位问题，但自动诊断可以快速对常见异常进行定位。SQL 诊断基于已有的集群信息表和监控表，提供了与之相关的诊断结果表与诊断汇总表来执行自动诊断。

7.3.1.1 集群信息表

集群信息表将一个集群中的所有实例的信息都汇聚在一起，让用户仅通过一条 SQL 就能查询整个集群相关信息。集群信息表列表如下：

- 集群拓扑表 `information_schema.cluster_info` 用于获取集群当前的拓扑信息，以及各个实例的版本、版本对应的Git Hash、各实例的启动时间、各实例的运行时间。
- 集群配置表 `information_schema.cluster_config` 用于获取集群当前所有实例的配置。对于 TiDB 4.0 之前的版本，用户必须逐个访问各个实例的 HTTP API 才能获取这些配置信息。
- 集群硬件表 `information_schema.cluster_hardware` 用于快速查询集群硬件信息。
- 集群负载表 `information_schema.cluster_load` 用于查询集群不同实例以及不同硬件类型的负载信息。
- 内核参数表 `information_schema.cluster_systeminfo` 用于查询集群不同实例的内核配置信息。目前支持查询 sysctl 的信息。
- 集群日志表 `information_schema.cluster_log` 用于集群日志查询，通过将查询条件下推到各个实例，降低日志查询对集群的影响，性能影响小于等 grep 命令。

TiDB 4.0 之前的系统表，只能查看当前实例信息，TiDB 4.0 实现了对应的集群表，可以在单个TiDB 实例上拥有整个集群的全局视图。这些表目前都位于 `information_schema` 中，查询方式与其他 `information_schema` 系统表一致。

7.3.1.2 集群监控表

为了能够动态地观察并对比不同时间段的集群情况，TiDB 4.0 诊断系统添加了集群监控系统表。所有监控表都在 `metrics_schema` 中，可以通过 SQL 的方式查询监控信息。SQL 查询监控允许用户对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。

- `information_schema.metrics_tables`：由于目前添加的系统表数量较多，因此用户可以通过该表查询这些监控表的相关元信息。

由于TiDB 集群的监控指标数量较大，因此TiDB 4.0 提供以下监控汇总表：

- 监控汇总表 `information_schema.metrics_summary` 用于汇总所有监控数据，以提升用户排查各监控指标的效率。
- 监控汇总并按 label 聚合表 `information_schema.metrics_summary_by_label` 同样用于汇总所有监控数据，但该表会对各项监控的不同的label 进行聚合统计。

7.3.1.3 自动诊断

以上集群信息表和集群监控表均需要用户手动执行 SQL 语句来排查集群问题。TiDB 4.0 中添加了自动诊断功能，根据已有的基础信息表，提供诊断相关的系统表，使诊断自动执行。自动诊断相关的系统表如下：

- 诊断结果表 `information_schema.inspection_result` 用于展示对系统的诊断结果。诊断是惰性触发，使用 `select * from inspection_result` 会触发所有诊断规则对系统进行诊断，并在结果中展示系统中的故障或风险。
- 诊断汇总表 `information_schema.inspection_summary` 用于对特定链路或模块的监控进行汇总，用户可以根据整个模块或链路的上下文来排查定位问题。

7.3.2 Statement Summary Tables

针对 SQL 性能相关的问题，MySQL 在 `performance_schema` 提供了 statement summary tables，用来监控和统计 SQL。例如其中的一张表 `events_statements_summary_by_digest`，提供了丰富的字段，包括延迟、执行次数、扫描行数、全表扫描次数等，有助于用户定位 SQL 问题。

为此，从 4.0.0-rc.1 版本开始，TiDB 在 `information_schema`（而不是 `performance_schema`）中提供与 `events_statements_summary_by_digest` 功能相似的系统表：

- `statements_summary`
- `statements_summary_history`
- `cluster_statements_summary`
- `cluster_statements_summary_history`

本文将详细介绍这些表，以及如何利用它们来排查 SQL 性能问题。

7.3.2.1 `statements_summary`

`statements_summary` 是 `information_schema` 里的一张系统表，它把 SQL 按 SQL digest 和 plan digest 分组，统计每一组的 SQL 信息。

此处的 SQL digest 与 slow log 里的 SQL digest 一样，是把 SQL 规一化后算出的唯一标识符。SQL 的规一化会忽略常量、空白符、大小写的差别。即语法一致的 SQL 语句，其 digest 也相同。

例如：

```
SELECT * FROM employee WHERE id IN (1, 2, 3) AND salary BETWEEN 1000 AND 2000;
select * from EMPLOYEE where ID in (4, 5) and SALARY between 3000 and 4000;
```

归一化后都是：

```
select * from employee where id in (...) and salary between ? and ?;
```

此处的 plan digest 是把执行计划规一化后算出的唯一标识符。执行计划的规一化会忽略常量的差别。由于相同的 SQL 可能产生不同的执行计划，所以可能分到多个组，同一个组内的执行计划是相同的。

`statements_summary` 用于保存 SQL 监控指标聚合后的结果。一般来说，每一项监控指标都包含平均值和最大值。例如执行延时对应 `AVG_LATENCY` 和 `MAX_LATENCY` 两个字段，分别是平均延时和最大延时。

为了监控指标的即时性，statements_summary 里的数据定期被清空，只展现最近一段时间内的聚合结果。清空周期由系统变量 tidb_stmt_summary_refresh_interval 设置。如果刚好在清空之后进行查询，显示的数据可能很少。

以下为查询 statements_summary 的部分结果：

```
  SUMMARY_BEGIN_TIME: 2020-01-02 11:00:00
    SUMMARY_END_TIME: 2020-01-02 11:30:00
           STMT_TYPE: Select
         SCHEMA_NAME: test
              DIGEST: 0611cc2fe792f8c146cc97d39b31d9562014cf15f8d41f23a4938ca341f54182
         DIGEST_TEXT: select * from employee where id = ?
         TABLE_NAMES: test.employee
         INDEX_NAMES: NULL
         SAMPLE_USER: root
          EXEC_COUNT: 3
         SUM_LATENCY: 1035161
         MAX_LATENCY: 399594
         MIN_LATENCY: 301353
         AVG_LATENCY: 345053
   AVG_PARSE_LATENCY: 57000
   MAX_PARSE_LATENCY: 57000
 AVG_COMPILE_LATENCY: 175458
 MAX_COMPILE_LATENCY: 175458
 ...........
             AVG_MEM: 103
             MAX_MEM: 103
            AVG_DISK: 65535
            MAX_DISK: 65535
  AVG_AFFECTED_ROWS: 0
         FIRST_SEEN: 2020-01-02 11:12:54
          LAST_SEEN: 2020-01-02 11:25:24
  QUERY_SAMPLE_TEXT: select * from employee where id=3100
   PREV_SAMPLE_TEXT:
        PLAN_DIGEST: f415b8d52640b535b9b12a9c148a8630d2c6d59e419aad29397842e32e8e5de3
               PLAN:  Point_Get_1      root     1        table:employee, handle:3100
```

> **注意：**
>
> 在 TiDB 中，statement summary tables 中字段的时间单位是纳秒(ns)，而 MySQL 中的时间单位是皮秒 (ps)。

7.3.2.2 statements_summary_history

statements_summary_history 的表结构与 statements_summary 完全相同，用于保存历史时间段的数据。通过历史数据，可以排查过去出现的异常，也可以对比不同时间的监控指标。

字段 SUMMARY_BEGIN_TIME 和 SUMMARY_END_TIME 代表历史时间段的开始时间和结束时间。

7.3.2.3 statements_summary_evicted

statements_summary 表的容量受 tidb_stmt_summary_max_stmt_count 配置控制，内部使用 LRU 算法，一旦接收到的 SQL 种类超过了 tidb_stmt_summary_max_stmt_count，表中最久未被命中的记录就会被驱逐出表。TiDB 引入了 statements_summary_evicted 表，该表记录了各个时段被驱逐 SQL 语句的具体数量。

只有当 SQL 语句被 statement summary 表驱逐的时候，statements_summary_evicted 表的内容才会更新。statements_summary_evicted 表记录发生驱逐的时间段和被驱逐 SQL 的数量。

7.3.2.4 statement summary 的 cluster 表

statements_summary、statements_summary_history 和 statements_summary_evicted 仅显示单台 TiDB server 的 statement summary 数据。若要查询整个集群的数据，需要查询 cluster_statements_summary、cluster_statements_summary_history 或 cluster_statements_summary_evicted 表。

cluster_statements_summary 显示各台TiDB server 的statements_summary 数据,cluster_statements_summary_history
′→显示各台TiDB server的 statements_summary_history 数据，而 cluster_statements_summary_evicted 则显示各台TiDB server 的 statements_summary_evicted 数据。这三张表用字段 INSTANCE 表示TiDB server 的地址，其他字段与 statements_summary、statements_summary_history 和 statements_summary_evicted 表相同。

7.3.2.5 参数配置

以下系统变量用于控制 statement summary：

- tidb_enable_stmt_summary：是否打开 statement summary 功能。1 代表打开，0 代表关闭，默认打开。statement summary 关闭后，系统表里的数据会被清空，下次打开后重新统计。经测试，打开后对性能几乎没有影响。
- tidb_stmt_summary_refresh_interval：statements_summary 的清空周期，单位是秒(s)，默认值是 1800。
- tidb_stmt_summary_history_size：statements_summary_history 保存每种 SQL 的历史的数量，也是 statements_summary_evicted 的表容量，默认值是 24。
- tidb_stmt_summary_max_stmt_count：statement summary tables 保存的 SQL 种类数量，默认 3000 条。当 SQL 种类超过该值时，会移除最近没有使用的 SQL。这些 SQL 将会被 DIGEST 为 NULL 的行和 statements_summary_evicted 统计记录。DIGEST 为 NULL 的行数据在TiDB Dashboard SQL 语句分析列表页面 中显示为 Others。
- tidb_stmt_summary_max_sql_length：字段 DIGEST_TEXT 和 QUERY_SAMPLE_TEXT 的最大显示长度，默认值是 4096。
- tidb_stmt_summary_internal_query：是否统计 TiDB 的内部 SQL。1 代表统计，0 代表不统计，默认不统计。

> 注意：

> 当一种 SQL 因为达到 tidb_stmt_summary_max_stmt_count 限制要被移除时，TiDB 会移除该 SQL 语句种类在所有时间段的数据。因此，即使一个时间段内的 SQL 种类数量没有达到上限，显示的 SQL 语句数量也会比实际的少。如遇到该情况，对性能也有一些影响，建议调大 tidb_stmt_summary_max_stmt_count 的值。

statement summary 配置示例如下：

```
set global tidb_enable_stmt_summary = true;
set global tidb_stmt_summary_refresh_interval = 1800;
set global tidb_stmt_summary_history_size = 24;
```

以上配置生效后，statements_summary 每 30 分钟清空一次，所以 statements_summary_history 保存最近 12 小时的历史数。statements_summary_evicted 保存最近 24 个发生了 evict 的时间段记录；statements_summary_evicted 则以 30 分钟为一个记录周期，表容量为 24 个时间段。

> 注意：
>
> tidb_stmt_summary_history_size、tidb_stmt_summary_max_stmt_count、tidb_stmt_summary_max_sql_length 这些配置都影响内存占用，建议根据实际情况调整（取决于 SQL 大小、SQL 数量、机器配置）不宜设置得过大。内存大小可通过 tidb_stmt_summary_history_size * tidb_stmt_summary_max_stmt_count * tidb_stmt_summary_max_sql_length * 3 来进行估算。

7.3.2.5.1 为 statement summary 设定合适的大小

在系统运行一段时间后（视系统负载而定），可以查看 statements_summary 表检查是否发生了 evict，例如：

```
select @@global.tidb_stmt_summary_max_stmt_count;
select count(*) from information_schema.statements_summary;
```

```
+-------------------------------------------+
| @@global.tidb_stmt_summary_max_stmt_count |
+-------------------------------------------+
| 3000                                      |
+-------------------------------------------+
1 row in set (0.001 sec)

+-----------------+
| count(*) |
+-----------------+
|     3001 |

+-----------------+
1 row in set (0.001 sec)
```

可以发现 statements_summary 表已经满了。再查看 statements_summary_evicted 表检查 evict 的数据。

```
select * from information_schema.statements_summary_evicted;
```

```
+---------------------+---------------------+---------------+
| BEGIN_TIME          | END_TIME            | EVICTED_COUNT |
+---------------------+---------------------+---------------+
| 2020-01-02 16:30:00 | 2020-01-02 17:00:00 |            59 |
+---------------------+---------------------+---------------+
| 2020-01-02 16:00:00 | 2020-01-02 16:30:00 |            45 |
+---------------------+---------------------+---------------+
2 row in set (0.001 sec)
```

由上可知，对最多 59 种 SQL 发生了 evict，也就是说最少应将 statement summary 的容量增大至 59 条记录。

7.3.2.6 目前的限制

由于 statement summary tables 默认都存储在内存中，TiDB server 重启后，statement summary 会全部丢失。

为解决该问题，TiDB v6.6.0 实验性地引入了statement summary 持久化功能，该功能默认为关闭。开启该功能后，历史数据不再存储在内存内，而是直接写入磁盘。TiDB server 重启后，历史数据也依然可用。

7.3.2.7 持久化 statements summary

> **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。

如目前的限制一节所描述，默认情况下statements summary 只在内存中维护，一旦TiDB server 发生重启，所有statements summary 数据都会丢失。自 v6.6.0 起，TiDB 实验性地提供了配置项tidb_stmt_summary_enable_persistent 来允许用户控制是否开启 statements summary 持久化。

如果要开启 statements summary 持久化，可以在TiDB 配置文件中添加如下配置：

```
[instance]
tidb_stmt_summary_enable_persistent = true
### 以下配置为默认值，可根据需求调整。
### tidb_stmt_summary_filename = "tidb-statements.log"
### tidb_stmt_summary_file_max_days = 3
### tidb_stmt_summary_file_max_size = 64 # MiB
### tidb_stmt_summary_file_max_backups = 0
```

开启 statements summary 持久化后，内存中只维护当前的实时数据，不再维护历史数据。历史数据生成后直接被写入磁盘文件，写入周期参考参数配置一节所描述的 tidb_stmt_summary_refresh_interval。后续针对

statements_summary_history 或 cluster_statements_summary_history 表的查询将结合内存和磁盘两处数据返回结果。

> 注意：
>
> - 当开启持久化后，由于不再于内存中维护历史数据，因此参数配置一节所描述的 tidb_stmt_summary_history_size 将不再生效，而是由 tidb_stmt_summary_file_max_days、tidb_stmt_summary_file_max_size 和 tidb_stmt_summary_file_max_backups 这三项配置来决定历史数据在磁盘上的保留数量和时间。
> - tidb_stmt_summary_refresh_interval 取值越小，数据写入到磁盘就越实时，但写入磁盘的冗余数据也会随之增多。

7.3.2.8 排查示例

下面用两个示例问题演示如何利用 statement summary 来排查。

7.3.2.8.1 SQL 延迟比较大，是不是服务端的问题？

例如客户端显示 employee 表的点查比较慢，那么可以按 SQL 文本来模糊查询：

```
SELECT avg_latency, exec_count, query_sample_text
    FROM information_schema.statements_summary
    WHERE digest_text LIKE 'select * from employee%';
```

结果如下，avg_latency 是 1 ms 和 0.3 ms，在正常范围，所以可以判定不是服务端的问题，继而排查客户端或网络问题。

```
+-------------+------------+-----------------------------------------------------------------------+
| avg_latency | exec_count | query_sample_text                                                     |
+-------------+------------+-----------------------------------------------------------------------+
|     1042040 |          2 | select * from employee where name='eric' |
|      345053 |          3 | select * from employee where id=3100     |
+-------------+------------+-----------------------------------------------------------------------+
2 rows in set (0.00 sec)
```

7.3.2.8.2 哪类 SQL 的总耗时最高？

假如上午 10:00 到 10:30 的 QPS 明显下降，可以从历史表中找出当时耗时最高的三类 SQL：

```
SELECT sum_latency, avg_latency, exec_count, query_sample_text
    FROM information_schema.statements_summary_history
    WHERE summary_begin_time='2020-01-02 10:00:00'
    ORDER BY sum_latency DESC LIMIT 3;
```

结果显示以下三类 SQL 的总延迟最高，所以这些 SQL 需要重点优化。

```
+--------------+-------------+------------+-------------------------------------------------------------------------------
    '→
| sum_latency | avg_latency | exec_count | query_sample_text
    '→                                                                |
+--------------+-------------+------------+-------------------------------------------------------------------------------
    '→
|     7855660 |     1122237 |          7 | select avg(salary) from employee where company_id=2013
    '→                  |
|     7241960 |     1448392 |          5 | select * from employee join company on employee.

    '→ company_id=company.id |
|     2084081 |     1042040 |          2 | select * from employee where name='eric'
    '→                                 |
+--------------+-------------+------------+-------------------------------------------------------------------------------
    '→
3 rows in set (0.00 sec)
```

7.3.2.9 表的字段介绍

7.3.2.9.1 `statements_summary` 字段介绍

下面介绍 `statements_summary` 表中各个字段的含义。

SQL 的基础信息：

- `STMT_TYPE`：SQL 语句的类型
- `SCHEMA_NAME`：执行这类 SQL 的当前 schema
- `DIGEST`：这类 SQL 的 digest
- `DIGEST_TEXT`：规一化后的 SQL
- `QUERY_SAMPLE_TEXT`：这类 SQL 的原 SQL 语句，多条语句只取其中一条
- `TABLE_NAMES`：SQL 中涉及的所有表，多张表用 , 分隔
- `INDEX_NAMES`：SQL 中使用的索引名，多个索引用 , 分隔
- `SAMPLE_USER`：执行这类 SQL 的用户名，多个用户名只取其中一个
- `PLAN_DIGEST`：执行计划的 digest
- `PLAN`：原执行计划，多条语句只取其中一条的执行计划
- `BINARY_PLAN`：以二进制格式编码后的原执行计划，存在多条语句时，只取其中一条语句的执行计划。用 `select tidb_decode_binary_plan('xxx...')` SQL 语句可以解析出具体的执行计划。
- `PLAN_CACHE_HITS`：这类 SQL 语句命中plan cache 的总次数
- `PLAN_IN_CACHE`：这类 SQL 语句的上次执行是否命中了plan cache

执行时间相关的信息：

- `SUMMARY_BEGIN_TIME`：当前统计的时间段的开始时间
- `SUMMARY_END_TIME`：当前统计的时间段的结束时间

- FIRST_SEEN：这类 SQL 的首次出现时间
- LAST_SEEN：这类 SQL 的最后一次出现时间

在 TiDB server 上的执行数据：

- EXEC_COUNT：这类 SQL 的总执行次数
- SUM_ERRORS：执行过程中遇到的 error 的总数
- SUM_WARNINGS：执行过程中遇到的 warning 的总数
- SUM_LATENCY：这类 SQL 的总延时
- MAX_LATENCY：这类 SQL 的最大延时
- MIN_LATENCY：这类 SQL 的最小延时
- AVG_LATENCY：这类 SQL 的平均延时
- AVG_PARSE_LATENCY：解析器的平均延时
- MAX_PARSE_LATENCY：解析器的最大延时
- AVG_COMPILE_LATENCY：优化器的平均延时
- MAX_COMPILE_LATENCY：优化器的最大延时
- AVG_MEM：使用的平均内存，单位 byte
- MAX_MEM：使用的最大内存，单位 byte
- AVG_DISK：使用的平均硬盘空间，单位 byte
- MAX_DISK：使用的最大硬盘空间，单位 byte

和 TiKV Coprocessor Task 相关的字段：

- SUM_COP_TASK_NUM：发送 Coprocessor 请求的总数
- MAX_COP_PROCESS_TIME：cop-task 的最大处理时间
- MAX_COP_PROCESS_ADDRESS：执行时间最长的 cop-task 所在地址
- MAX_COP_WAIT_TIME：cop-task 的最大等待时间
- MAX_COP_WAIT_ADDRESS：等待时间最长的 cop-task 所在地址
- AVG_PROCESS_TIME：SQL 在 TiKV 的平均处理时间
- MAX_PROCESS_TIME：SQL 在 TiKV 的最大处理时间
- AVG_WAIT_TIME：SQL 在 TiKV 的平均等待时间
- MAX_WAIT_TIME：SQL 在 TiKV 的最大等待时间
- AVG_BACKOFF_TIME：SQL 遇到需要重试的错误时在重试前的平均等待时间
- MAX_BACKOFF_TIME：SQL 遇到需要重试的错误时在重试前的最大等待时间
- AVG_TOTAL_KEYS：Coprocessor 扫过的 key 的平均数量
- MAX_TOTAL_KEYS：Coprocessor 扫过的 key 的最大数量
- AVG_PROCESSED_KEYS：Coprocessor 处理的 key 的平均数量。相比 avg_total_keys，avg_processed_keys 不包含 MVCC 的旧版本。如果 avg_total_keys 和 avg_processed_keys 相差很大，说明旧版本比较多
- MAX_PROCESSED_KEYS：Coprocessor 处理的 key 的最大数量

和事务相关的字段：

- AVG_PREWRITE_TIME：prewrite 阶段消耗的平均时间
- MAX_PREWRITE_TIME prewrite 阶段消耗的最大时间
- AVG_COMMIT_TIME：commit 阶段消耗的平均时间

- MAX_COMMIT_TIME：commit 阶段消耗的最大时间
- AVG_GET_COMMIT_TS_TIME：获取 commit_ts 的平均时间
- MAX_GET_COMMIT_TS_TIME：获取 commit_ts 的最大时间
- AVG_COMMIT_BACKOFF_TIME：commit 时遇到需要重试的错误时在重试前的平均等待时间
- MAX_COMMIT_BACKOFF_TIME：commit 时遇到需要重试的错误时在重试前的最大等待时间
- AVG_RESOLVE_LOCK_TIME：解决事务的锁冲突的平均时间
- MAX_RESOLVE_LOCK_TIME：解决事务的锁冲突的最大时间
- AVG_LOCAL_LATCH_WAIT_TIME：本地事务等待的平均时间
- MAX_LOCAL_LATCH_WAIT_TIME：本地事务等待的最大时间
- AVG_WRITE_KEYS：写入 key 的平均数量
- MAX_WRITE_KEYS：写入 key 的最大数量
- AVG_WRITE_SIZE：写入的平均数据量，单位 byte
- MAX_WRITE_SIZE：写入的最大数据量，单位 byte
- AVG_PREWRITE_REGIONS：prewrite 涉及的平均Region 数量
- MAX_PREWRITE_REGIONS：prewrite 涉及的最大Region 数量
- AVG_TXN_RETRY：事务平均重试次数
- MAX_TXN_RETRY：事务最大重试次数
- SUM_BACKOFF_TIMES：这类 SQL 遇到需要重试的错误后的总重试次数
- BACKOFF_TYPES：遇到需要重试的错误时的所有错误类型及每种类型重试的次数，格式为 类型:次数。如有多种错误则用，分隔，例如 txnLock:2,pdRPC:1
- AVG_AFFECTED_ROWS：平均影响行数
- PREV_SAMPLE_TEXT：当 SQL 是 COMMIT 时，该字段为 COMMIT 的前一条语句；否则该字段为空字符串。当 SQL 是 COMMIT 时，按digest 和 prev_sample_text 一起分组，即不同 prev_sample_text 的 COMMIT 也会分到不同的行

7.3.2.9.2 statements_summary_evicted 字段介绍

- BEGIN_TIME: 记录的开始时间；
- END_TIME: 记录的结束时间；
- EVICTED_COUNT：在记录的时间段内 evict 了多少种 SQL。

7.3.3 TiDB Dashboard Top SQL 页面

TiDB Dashboard 的 Top SQL 功能允许你可视化地监控和探索数据库中各个 SQL 语句在执行过程中的 CPU 开销情况，从而对数据库性能问题进行优化和处理。Top SQL 持续收集各个TiDB 及 TiKV 实例每秒的实时 CPU 负载等数据（按 SQL 类型分别统计），并存储至多 30 天。你可以通过Top SQL 展示的图表及表格快速分析某个TiDB 或 TiKV 实例在某段时间中高 CPU 负载是来自于哪些 SQL 语句。

Top SQL 具有以下功能：

- 通过图表及表格，可视化地展示 CPU 开销最多的 5 类 SQL 语句。
- 展示每秒请求数、平均延迟、查询计划等详细执行信息。
- 支持统计所有正在执行、尚未执行完毕的 SQL 语句。
- 支持查看集群中指定TiDB 及 TiKV 实例的情况。

7.3.3.1 推荐适用场景

Top SQL 适用于分析性能问题。以下列举了一些典型的Top SQL 适用场景：

- 通过监控图发现集群中有个别TiKV 实例的 CPU 非常高，期望了解 CPU 热点来自于哪些 SQL 语句，以便对其进行优化、更好地利用上分布式资源。
- 集群整体 CPU 占用率非常高、数据库查询缓慢，期望快速知悉目前哪些 SQL 语句开销了最多的 CPU 资源，以便对它们进行优化。
- 集群整体 CPU 占用率突然发生了显著变化，期望了解变化前后主要的 SQL 资源开销区别。
- 分析集群当前最消耗资源的 SQL 语句情况，希望对它们进行优化以便降低硬件开支。

Top SQL 不能用于解答与性能无关的问题，例如数据正确性或异常崩溃问题。

当前Top SQL 仍然处于早期阶段，功能正在持续加强。以下列举了一些目前暂不支持的场景，供参考：

- 暂时不支持分析Top 5 以外 SQL 语句的开销情况（如多业务混合时）。
- 暂时不支持按 User、Database 等不同维度分析Top N SQL 语句的开销情况。
- 暂时不支持分析并非由于 CPU 负载高导致的数据库性能问题，例如锁冲突。

7.3.3.2 访问页面

你可以通过以下任一方式访问Top SQL 页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击Top SQL。

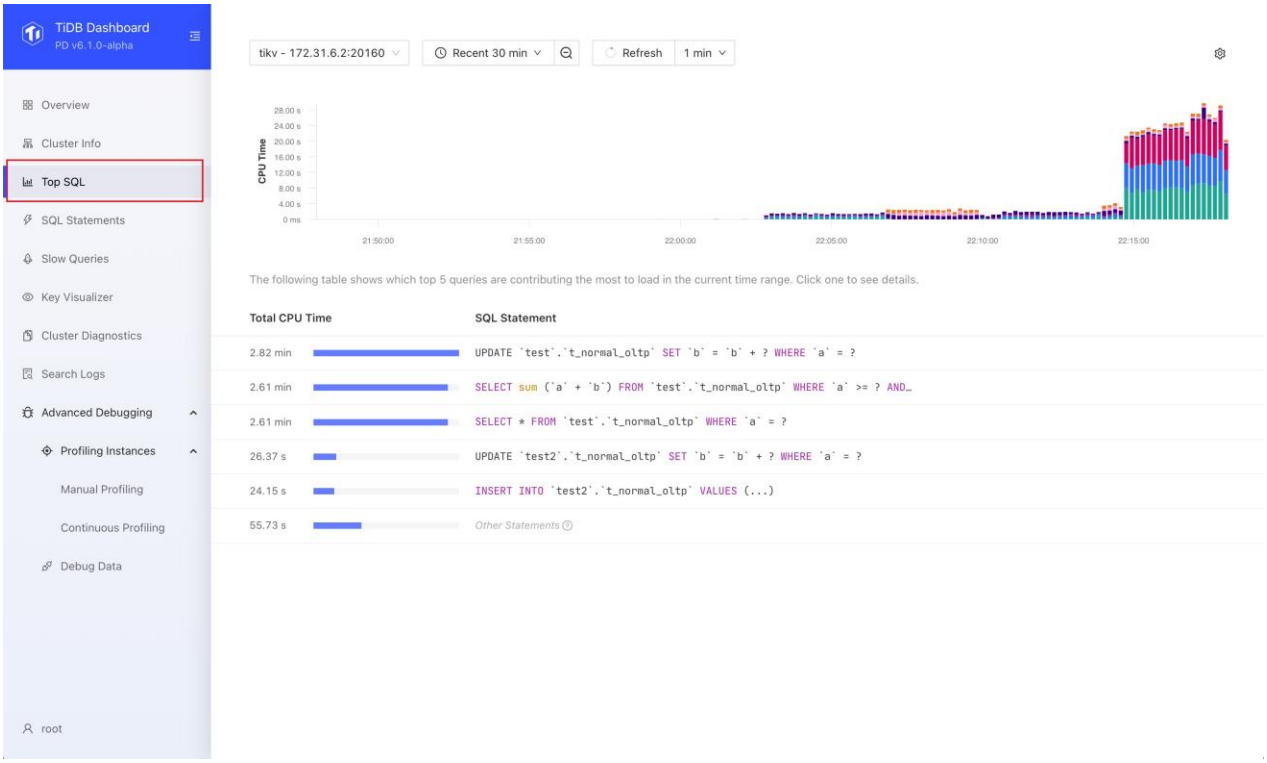


图 62: Top SQL

- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/topsql（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

7.3.3.3 启用Top SQL

> 注意：
>
> 要使用Top SQL，你需要使用 TiUP（v1.9.0 及以上版本）或 TiDB Operator（v1.3.0 及以上版本）部署或升级集群。如果你已经使用旧版本TiUP 或 TiDB Operator 进行了集群升级，请参见FAQ 进行处理。

Top SQL 开启后会对集群性能产生轻微的影响（平均 3% 以内），因此该功能默认关闭。你可以通过以下方法启用 Top SQL：

1. 访问Top SQL 页面。
2. 点击打开设置(Open Settings)。在右侧设置 (Settings) 页面，将启用特性(Enable Feature) 下方的开关打开。
3. 点击保存(Save)。

你仅能看到开启功能之后的 CPU 负载细节情况，在开启功能之前的 CPU 负载细节无法在界面上呈现。另外，数据有至多 1 分钟左右的延迟，因此你可能需要等待片刻才能看到数据。

除了通过图形化界面以外，你也可以配置TiDB 系统变量tidb_enable_top_sql 来启用Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 1;
```

7.3.3.4 使用Top SQL

以下是Top SQL 的常用步骤：

1. 访问Top SQL 页面。
2. 选择一个你想要观察负载的具体TiDB 或 TiKV 实例。

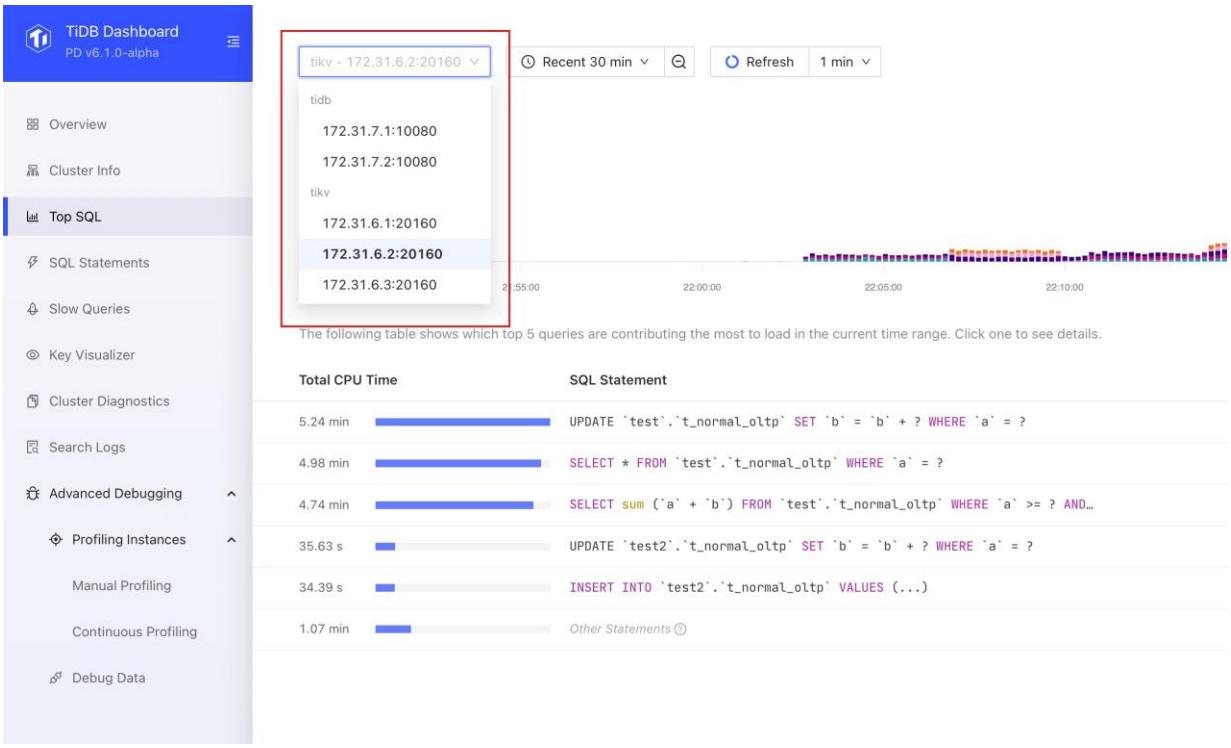


图 63: 选择实例

如果你不知道要观察哪一个TiDB 或TiKV 实例，可以选择任意一个实例。在集群 CPU 开销非常不均衡的情况下，你可以首先通过Grafana 中的 CPU 监控来确定具体期望观察的实例。

3. 观察Top SQL 呈现的Top 5 图表及表格。

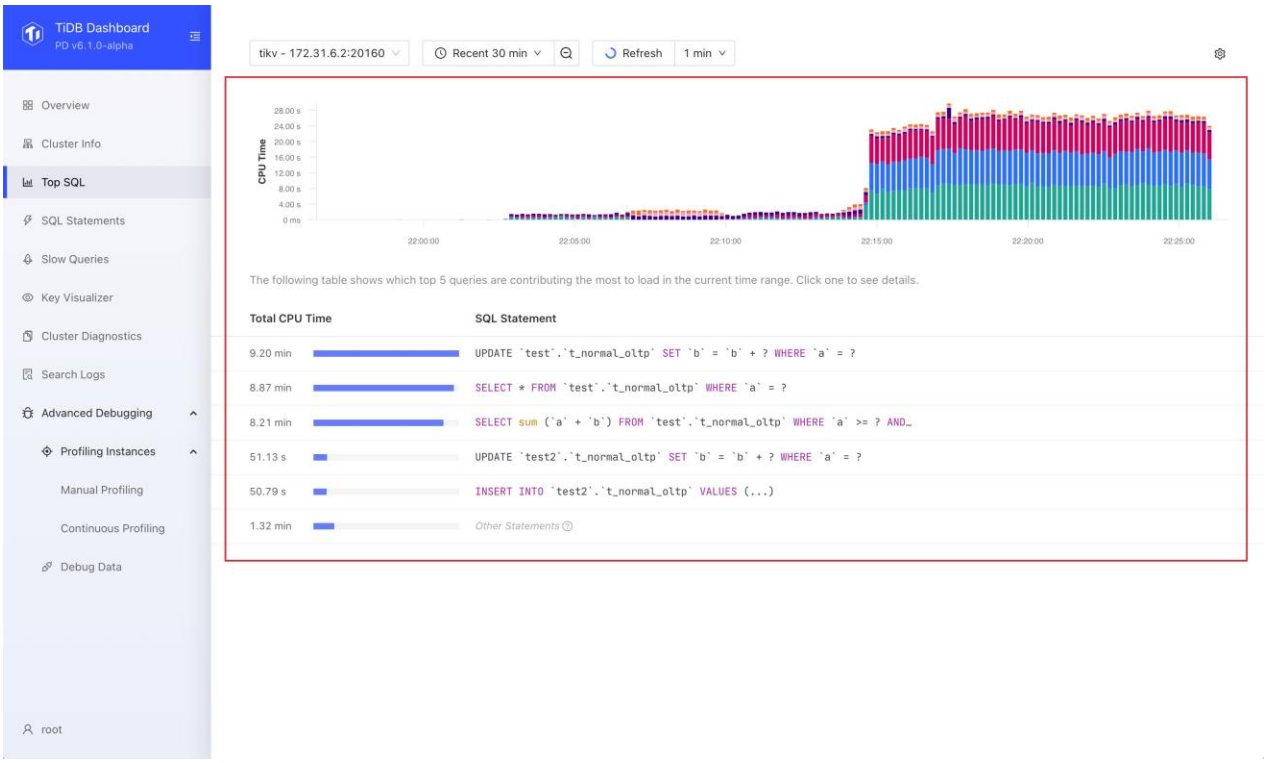


图 64: 图表表格

柱状图中色块的大小代表了 SQL 语句在该时刻消耗的 CPU 资源的多少，不同颜色区分了不同类型的 SQL 语句。大多数情况下，你都应该关注图表中相应时间范围内 CPU 资源开销较大的 SQL 语句。

4. 点击表格中的某一个 SQL 语句后，可以展开查看该语句不同执行计划的执行情况，例如 Call/sec（平均每秒请求数）、Scan Indexes/sec（平均每秒扫描索引数）等。

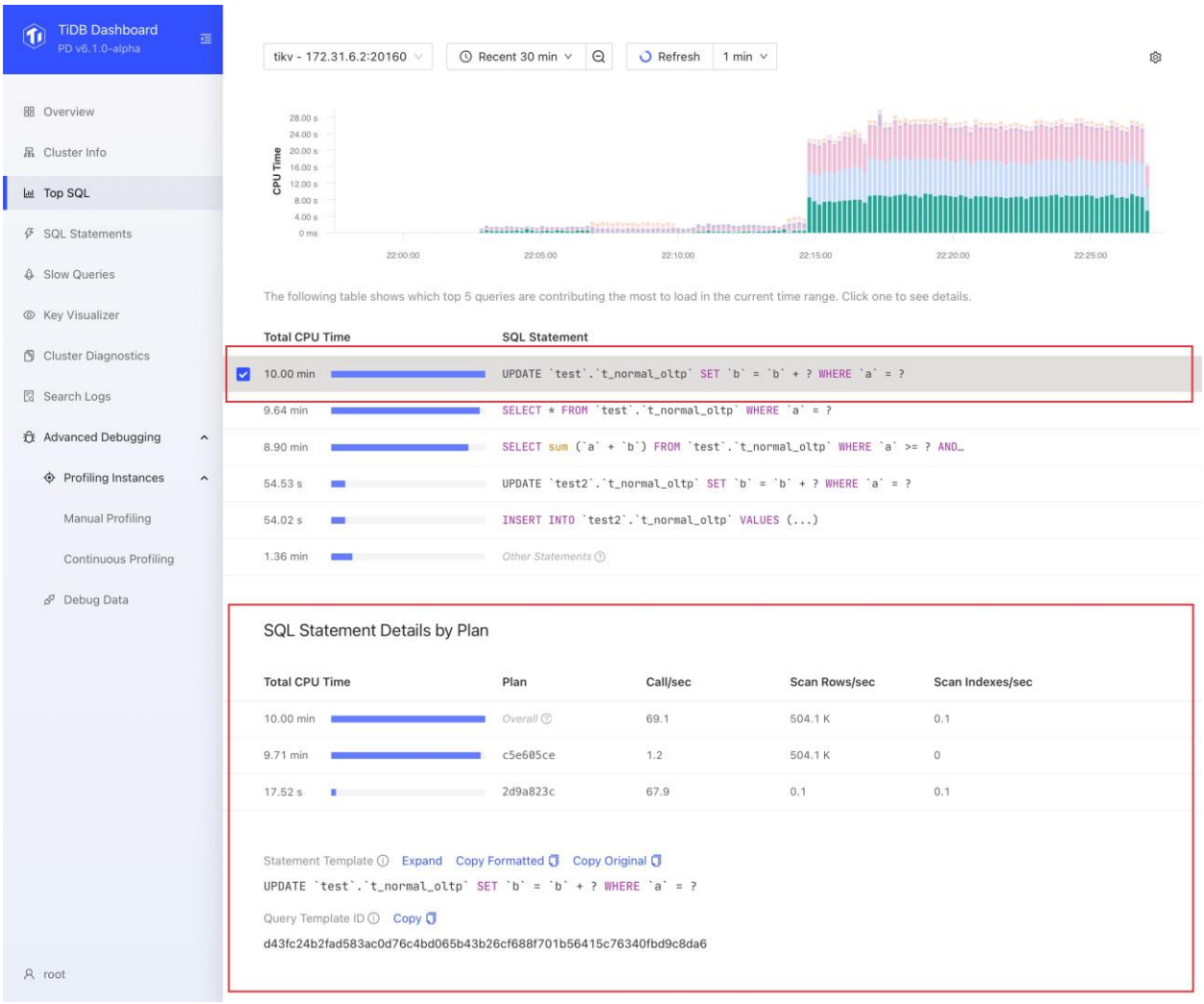


图 65: 详情

5. 基于这些初步线索，进一步在SQL 语句分析或慢查询界面中了解该 SQL 语句开销大量 CPU 资源、或扫大量数据的详细原因。

除此以外：

- 你可以在时间选择器中调整时间范围，或在图表中框选一个时间范围，来更精确、细致地观察问题。更小的时间范围将能提供细节更丰富的数据，数据最高精度可达 1 秒。

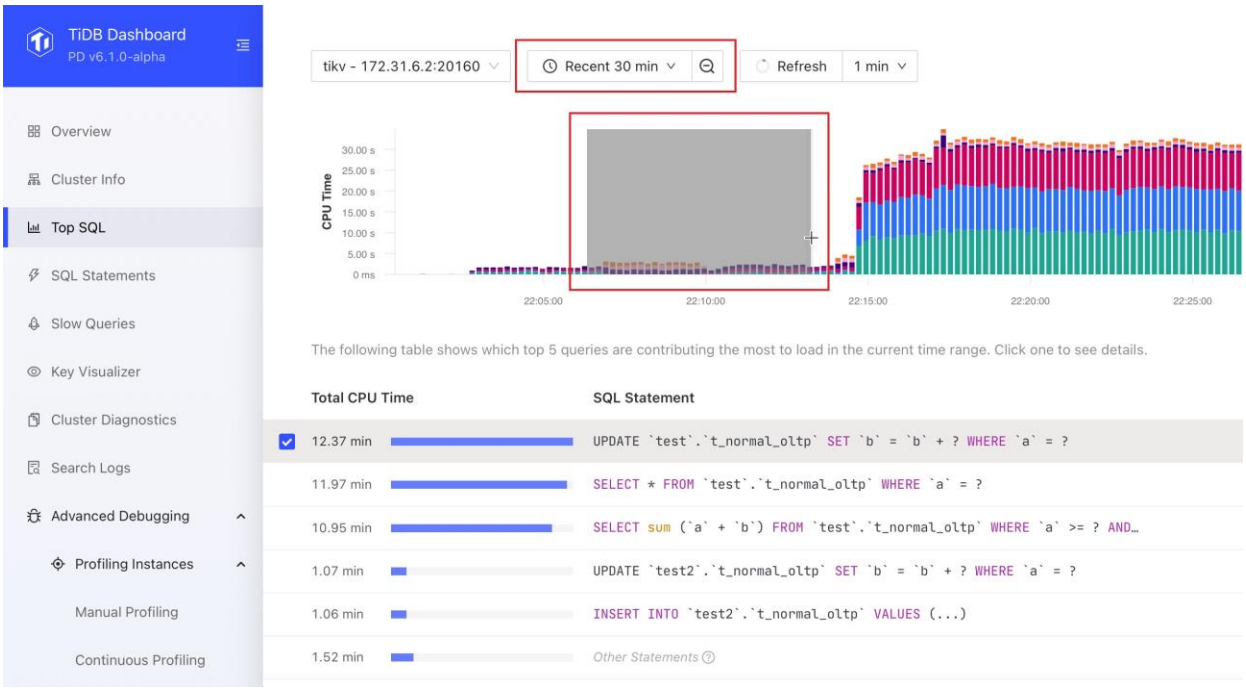


图 66: 修改时间范围

- 如果图表中显示的数据已过时，你可以点击刷新(Refresh)按钮，或在刷新(Refresh)下拉列表中选择自动刷新。

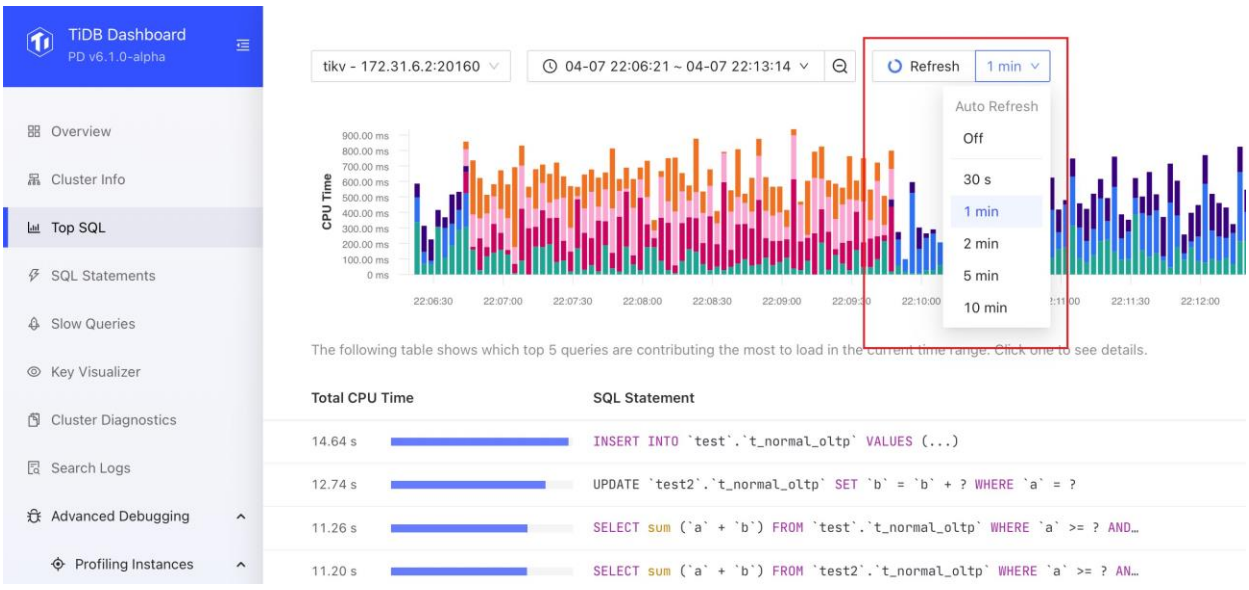


图 67: 刷新

7.3.3.5 停用Top SQL

你可以通过以下步骤停用该功能：

1. 访问Top SQL 页面。
2. 点击右上角齿轮按钮打开设置界面，将启用特性(Enable Feature) 下方的开关关闭。
3. 点击保存(Save)。
4. 在弹出的关闭Top SQL 功能(Disable Top SQL Feature) 对话框中，点击确认(Disable)。

除了通过图形化界面以外，你也可以配置TiDB 系统变量tidb_enable_top_sql 来停用Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 0;
```

7.3.3.6 常见问题

1. 界面上提示“集群中未启动必要组件 NgMonitoring”无法启用功能

请参见TiDB Dashboard FAQ。

2. 该功能开启后对集群是否有性能影响？

该功能对集群性能有轻微影响。根据我们的测算，该功能对集群的平均性能影响小于 3%。

3. 该功能目前是什么状态？

该功能是正式特性，在生产环境中可用。

4. 界面中显示的其他语句（Other Statements）是什么意思？

其他所有非Top 5 语句产生的 CPU 开销或执行情况都会被统计在该项中。你可以基于这一项了解Top 5 的 SQL 语句开销在整体所有 SQL 语句的 CPU 开销中的比例。

5. Top SQL 展示的 CPU 开销总和与进程的实际 CPU 开销是什么关系？

它们之间有很强的相关性，但不完全一致。以TiKV 为例，TiKV 的 CPU 开销还可能来自于其他副本的数据同步写入，这些开销不会被计入Top SQL。但总的来说，Top SQL 中开销比例比较大的 SQL 语句实际的 CPU 资源开销也确实会更大。

6. Top SQL 图表的纵坐标是什么意思？

代表消耗 CPU 资源的多少。消耗资源越多的 SQL 语句，该值越大。在绝大多数情况下，你都不需要关心纵坐标具体数值的含义。

7. 还没有执行完毕的 SQL 语句会被统计到吗？

会。Top SQL 图表上所展示的每一时刻 CPU 开销比例即为这一时刻所有正在运行的 SQL 语句的 CPU 开销情况。

7.3.4 定位消耗系统资源多的查询

TiDB 会将执行时间超过tidb_expensive_query_time_threshold 限制（默认值为 60s），或使用内存超过tidb_mem_quota_query 限制（默认值为 1 GB）的语句输出到tidb-server 日志文件（默认文件为“tidb.log”）中，用于在语句执行结束前定位消耗系统资源多的查询语句（以下简称为 expensive query），帮助用户分析和解决语句执行的性能问题。

注意，expensive query 日志和慢查询日志的区别是，慢查询日志是在语句执行完后才打印，expensive query 日志可以将正在执行的语句的相关信息打印出来。当一条语句在执行过程中达到资源使用阈值时（执行时间/使用内存量），TiDB 会即时将这条语句的相关信息写入日志。

7.3.4.1 Expensive query 日志示例

```
[2020/02/05 15:32:25.096 +08:00] [WARN] [expensivequery.go:167] [expensive_query] [cost_time
    ↪ =60.008338935s] [wait_time=0s] [request_count=1] [total_keys=70] [process_keys=65]
    [ ↪ num_cop_tasks=1] [process_avg_time=0s] [process_p90_time=0s] [process_max_time=0s]
    [ ↪ process_max_addr=10.0.1.9:20160] [wait_avg_time=0.002s] [wait_p90_time=0.002s] [
    ↪ wait_max_time=0.002s] [wait_max_addr=10.0.1.9:20160] [stats=t:pseudo] [conn_id=60026]
    [ ↪ user=root] [database=test] [table_ids="[122]"] [txn_start_ts=414420273735139329]
    [mem_max ↪ ="1035 Bytes (1.0107421875 KB)"] [sql="insert into t select sleep(1) from t"]
```

7.3.4.2 字段含义说明

基本字段：

- `cost_time`：日志打印时语句已经花费的执行时间。
- `stats`：语句涉及到的表或索引使用的统计信息版本。值为 pseudo 时表示无可用统计信息，需要对表或索引进行analyze。
- `table_ids`：语句涉及到的表的 ID。
- `txn_start_ts`：事务的开始时间戳，也是事务的唯一 ID，可以用这个值在TiDB 日志中查找事务相关的其他日志。
- `sql`：SQL 语句。

和内存使用相关的字段：

- `mem_max`：日志打印时语句已经使用的内存空间。该项使用两种单位标识内存使用量，分别为 Bytes 以及易于阅读的自适应单位（比如 MB、GB 等）。

和 SQL 执行的用户相关的字段：

- `user`：执行语句的用户名。
- `conn_id`：用户的连接 ID，可以用类似 `con:60026` 的关键字在TiDB 日志中查找该连接相关的其他日志。
- `database`：执行语句时使用的 database。

和 TiKV Coprocessor Task 相关的字段：

- `wait_time`：该语句在TiKV 的等待时间之和，因为TiKV 的Coprocessor 线程数是有限的，当所有的Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。
- `request_count`：该语句发送的 Coprocessor 请求的数量。
- `total_keys`：Coprocessor 扫过的 key 的数量。
- `processed_keys`：Coprocessor 处理的 key 的数量。与total_keys 相比，processed_keys 不包含 MVCC 的旧版本。如果processed_keys 和 total_keys 相差很大，说明旧版本比较多。
- `num_cop_tasks`：该语句发送的 Coprocessor 请求的数量。
- `process_avg_time`：Coprocessor 执行task 的平均执行时间。
- `process_p90_time`：Coprocessor 执行task 的 P90 分位执行时间。

- `process_max_time`：Coprocessor 执行task 的最长执行时间。
- `process_max_addr`：task 执行时间最长的 Coprocessor 所在地址。
- `wait_avg_time`：Coprocessor 上 task 的等待时间。
- `wait_p90_time`：Coprocessor 上 task 的 P90 分位等待时间。
- `wait_max_time`：Coprocessor 上 task 的最长等待时间。
- `wait_max_addr`：task 等待时间最长的 Coprocessor 所在地址。

7.3.5 使用 PLAN REPLAYER 保存和恢复集群现场信息

用户在定位排查TiDB 集群问题时，经常需要提供系统和查询计划相关的信息。为了帮助用户更方便地获取相关信息，更高效地排查集群问题，TiDB 在 v5.3.0 中引入了 PLAN REPLAYER 命令，用于“一键”保存和恢复现场问题的相关信息，提升查询计划问题诊断的效率，同时方便将问题归档管理。

PLAN REPLAYER 主要功能如下：

- 导出排查现场TiDB 集群的相关信息，导出为ZIP 格式的文件用于保存。
- 在任意TiDB 集群上导入另一TiDB 集群现场信息的ZIP 文件。

7.3.5.1 使用 PLAN REPLAYER 导出集群信息

你可以使用 PLAN REPLAYER 来保存TiDB 集群的现场信息。导出接口如下：

```
PLAN REPLAYER DUMP EXPLAIN [ANALYZE] sql-statement;
```

TiDB 根据 `sql-statement` 整理出以下集群现场信息：

- TiDB 版本信息
- TiDB 配置信息
- TiDB Session 系统变量
- TiDB 执行计划绑定信息（SQL Binding）
- `sql-statement` 中所包含的表结构
- `sql-statement` 中所包含表的统计信息
- `EXPLAIN [ANALYZE] sql-statement` 的结果
- 优化器进行查询优化的一些内部步骤的记录

> **注意：**
>
> PLAN REPLAYER 不会导出表中数据

7.3.5.1.1 PLAN REPLAYER 导出示例

```
use test;
create table t(a int, b int);
insert into t values(1,1), (2, 2), (3, 3);
```

```
analyze table t;

plan replayer dump explain select * from t;
```

PLAN REPLAYER DUMP 会将以上信息打包整理成 ZIP 文件，并返回文件标识作为执行结果。

> **注意：**
>
> ZIP 文件最多会在TiDB 集群中保存一个小时，超时后TiDB 会将其删除。

```
MySQL [test]> plan replayer dump explain select * from t;
```

```
+------------------------------------------------------------------------+
| Dump_link                                                              |
+------------------------------------------------------------------------+
| replayer_JOGvpu4t7dssySqJfTtS4A==_1635750890568691080.zip |
+------------------------------------------------------------------------+
1 row in set (0.015 sec)
```

你同样可以通过tidb_last_plan_replayer_token 这个会话变量来获取上一次 PLAN REPLAYER dump 执行的结果。

```
SELECT @@tidb_last_plan_replayer_token;
```

```
| @@tidb_last_plan_replayer_token                            |
+------------------------------------------------------------+
| replayer_Fdamsm3C7ZiPJ-LQqgVjkA==_1663304195885090000.zip |
+------------------------------------------------------------+
1 row in set (0.00 sec)
```

对于多条 SQL 的情况，你可以通过文件的方式来获取plan replayer dump 的结果，多条 SQL 语句在文件中以 ; 进行分隔。

```
plan replayer dump explain 'sqls.txt';
```

```
SELECT @@tidb_last_plan_replayer_token;
```

```
+------------------------------------------------------------+
| @@tidb_last_plan_replayer_token                            |
+------------------------------------------------------------+
| replayer_LEDKg8sb-KOu24QesiH8ig==_1663226556509182000.zip |
+------------------------------------------------------------+
1 row in set (0.00 sec)
```

因为 MySQL Client 无法下载文件，所以需要通过TiDB HTTP 接口和文件标识下载文件：

```
http://${tidb-server-ip}:${tidb-server-status-port}/plan_replayer/dump/${file_token}
```

其中，`${tidb-server-ip}:${tidb-server-status-port}` 是集群中任意TiDB server 的地址。示例如下：

```
curl http://127.0.0.1:10080/plan_replayer/dump/replayer_JOGvpu4t7dssySqJfTtS4A==
    '→ _16357508905686910 80.zip > plan_replayer.zip
```

7.3.5.2 使用 PLAN REPLAYER 导入集群信息

> **警告：**
>
> PLAN REPLAYER 在一个TiDB 集群上导入另一集群的现场信息，会修改导入集群的TiDB Session 系统变量、执行计划绑定信息、表结构和统计信息。

有 PLAN REPLAYER 导出的 ZIP 文件后，用户便可以通过 PLAN REPLAYER 导入接口在任意TiDB 集群上恢复另一集群地现场信息。语法如下：

```
PLAN REPLAYER LOAD 'file_name';
```

以上语句中，`file_name` 为要导入的 ZIP 文件名。

示例如下：

```
PLAN REPLAYER LOAD 'plan_replayer.zip';
```

导入完毕后，该TiDB 集群就载入了所需要的表结构、统计信息等其他影响构造 Plan 所需要的信息。你可以通过以下方式查看执行计划以及验证统计信息:

```
mysql> desc t;
+-------+---------+------+------+---------+-------------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------------+
| a     | int(11) | YES  |      | NULL    |       |
| b     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------------+
2 rows in set (0.01 sec)

mysql> explain select * from t where a = 1 or b =1;
+--
    '→ ----------------------+---------+-----------+---------------+-----------------------------------+
    '→
| id                    | estRows | task      | access object | operator info
    '→                       |
```

```
+--
    '→ ----------------------+---------+-----------+----------------+                                                +
    '→
| TableReader_7            | 0.01    | root      |                | data:Selection_6
    '→                      |
| └─Selection_6            | 0.01    | cop[tikv] |                | or(eq(test.t.a, 1), eq(test.t.
    '→ b, 1)) |
|   └─TableFullScan_5      | 6.00    | cop[tikv] | table:t        | keep order:false, stats:pseudo
    '→        |
+--
    '→ ----------------------+---------+-----------+----------------+                                                +
    '→
3 rows in set (0.00 sec)

mysql> show stats_meta;
+---------+------------+----------------+---------------------+--------------+-----------------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------------------+
| test    | t          |                | 2022-08-26 15:52:07 |            3 |         6 |
+---------+------------+----------------+---------------------+--------------+-----------------------+
1 row in set (0.04 sec)
```

加载并还原所需现场后，即可在该现场诊断和改进执行计划。

7.3.5.3 使用 PLAN REPLAYER CAPTURE 抓取目标计划

在用户定位 TiDB 执行计划的部分场景中，目标 SQL 语句与目标计划可能仅在查询中偶尔出现，无法使用 PLAN REPLAYER 直接抓取。此时你可以使用 PLAN REPLAYER CAPTURE 来帮助定向抓取目标 SQL 语句与目标计划的优化器信息。

PLAN REPLAYER CAPTURE 主要功能如下：

- 在TiDB 集群内部提前注册目标 SQL 语句与执行计划的 Digest，并开始匹配目标查询。
- 当目标查询匹配成功时，直接抓取其优化器相关信息，导出为ZIP 格式的文件用于保存。
- 针对匹配到的每组 SQL 和执行计划，信息只抓取一次。
- 通过系统表显示正在进行的匹配任务，以及生成的文件。
- 定时清理历史文件。

7.3.5.3.1 开启 PLAN REPLAYER CAPTURE

PLAN REPLAYER CAPTURE 功能通过系统变量tidb_enable_plan_replayer_capture 控制。要开启 PLAN REPLAYER
'→ CAPTURE，将变量值设为 ON。

7.3.5.3.2 使用 PLAN REPLAYER CAPTURE 功能

你可以通过以下方式向TiDB 集群注册目标 SQL 语句和计划的 Digest:

```
PLAN REPLAYER CAPTURE 'sql_digest' 'plan_digest';
```

当你的目标 SQL 语句对应多种执行计划，且你想抓取所有执行计划时，你可以通过以下 SQL 语句一键注册:

```
PLAN REPLAYER CAPTURE 'sql_digest' '*';
```

7.3.5.3.3 查看 PLAN REPLAYER CAPTURE 抓取任务

你可以通过以下方式查看集群中目前正在工作的 PLAN REPLAYER CAPTURE 的抓取任务:

```
mysql> PLAN PLAYER CAPTURE 'example_sql' 'example_plan';
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM mysql.plan_replayer_task;
+-------------+--------------+----------------------------+
| sql_digest  | plan_digest  | update_time                |
+-------------+--------------+----------------------------+
| example_sql | example_plan | 2023-01-28 11:58:22 |
+-------------+--------------+----------------------------+
1 row in set (0.01 sec)
```

7.3.5.3.4 查看 PLAN REPLAYER CAPTURE 抓取结果

当 PLAN REPLAYER CAPTURE 成功抓取到结果后，可以通过以下 SQL 语句查看用于下载的文件标识:

```
mysql> SELECT * FROM mysql.plan_replayer_status;
+--
    '→                                                                +
    '→
| sql_digest                                                              | plan_digest
    '→                                                                | origin_sql | token
    '→                                                                | update_time          | fail_reason |
    '→  instance        |
+--
    '→                                                                +
    '→ ------------------------------------------------------------------
| 086e3fbd2732f7671c17f299d4320689deeeb87ba031240e1e598a0ca14f808c | 042
    '→ de2a6652a6d20afc629ff90b8507b7587a1c7e1eb122c3e0b808b1d80cc02 |              |
    '→ replayer_Utah4nkz2sIEzkks7tIRog==_1668746293523179156.zip | 2022-11-18 12:38:13 | NULL
    '→        | 172.16.4.4:4022 |
| b5b38322b7be560edb04f33f15b15a885e7c6209a22b56b0804622e397199b54 | 1770
    '→ efeb3f91936e095f0344b629562bf1b204f6e46439b7d8f842319297c3b5 |              |
    '→ replayer_Z2mUXNHDjU_WBmGdWQqifw==_1668746293560115314.zip | 2022-11-18 12:38:13 | NULL
    '→        | 172.16.4.4:4022 |
```

```
| 96d00c0b3f08795fe94e2d712fa1078ab7809faf4e81d198f276c0dede818cf9 | 8892
    '→ f74ac2a42c2c6b6152352bc491b5c07c73ac3ed66487b2c990909bae83e8 |              |
    '→ replayer_RZcRHJB7BaCccxFfOIAhWg==_1668746293578282450.zip | 2022-11-18 12:38:13 | NULL
    '→         | 172.16.4.4:4022 |
+--
    '→                                                                +
    '→
3 rows in set (0.00 sec)
```

下载 PLAN REPLAYER CAPTURE 的文件方法与 PLAN REPLAYER 相同，请参考PLAN REPLAYER 导出示例。

> **注意：**
>
> PLAN REPLAYER CAPTURE 的结果文件最多会在TiDB 集群中保存一周，超时后TiDB 会将其删除。

7.3.5.4 使用 PLAN REPLAYER CONTINUOUS CAPTURE

开启 PLAN REPLAYER CONTINUOUS CAPTURE 功能后，TiDB 将以 SQL DIGEST 和 PLAN DIGEST 为维度异步地将业务 SQL 语句以 PLAN REPLAYER 的方式进行记录，对于相同 DIGEST 的 SQL 语句与执行计划，PLAN REPLAYER CONTINUOUS '→ CAPTURE 不会重复记录。

7.3.5.4.1 开启 PLAN REPLAYER CONTINUOUS CAPTURE

PLAN REPLAYER CONTINUOUS CAPTURE 功能通过系统变量tidb_enable_plan_replayer_continuous_capture 控制。要开启 PLAN REPLAYER CONTINUOUS CAPTURE，将变量值设为 ON。

7.3.5.4.2 查看 PLAN REPLAYER CONTINUOUS CAPTURE 抓取结果

查看 PLAN REPLAYER CONTINUOUS CAPTURE 抓取结果的方法同查看 PLAN REPLAYER CAPTURE 抓取结果。

7.4 支持资源

如果你在使用TiDB 的过程中遇到了问题，你可以通过以下方式向 PingCAP 或TiDB 社区寻求帮助：

- 从 PingCAP 获取支持（需要订阅TiDB 企业版）：
  - 提交工单
- 从 TiDB 社区寻求帮助：
  - AskTUG 论坛
  - Stack Overflow（在 #tidb 标签下提问）
- 了解TiDB 的实现和设计
  - TiDB Internals 论坛

8 性能调优

8.1 优化手册

8.1.1 TiDB 性能优化概述

本文介绍性能优化的基本概念，比如用户响应时间、吞吐和数据库时间，以及性能优化的通用流程。

8.1.1.1 用户响应时间和数据库时间

8.1.1.1.1 用户响应时间

用户响应时间是指应用系统为用户返回请求结果所消耗的时间。一个典型的用户请求的处理时序图如下，包含了用户和应用系统的网络延迟、应用的处理时间、应用和数据库的交互时的网络延迟和数据库的服务时间等。用户响应时间受到请求链路上各个子系统的影响，比如网络延迟和带宽、系统并发用户数和请求类型、服务器 CPU 和 IO 资源使用率等。要对整个系统进行有效的优化，你需要先定位用户响应时间的瓶颈。

你可以通过以下公式计算指定时间范围(ΔT) 内总的用户响应时间：

ΔT 时间内总的用户响应时间= 平均 TPS (Transactions Per Second) x 用户平均响应时间x ΔT。

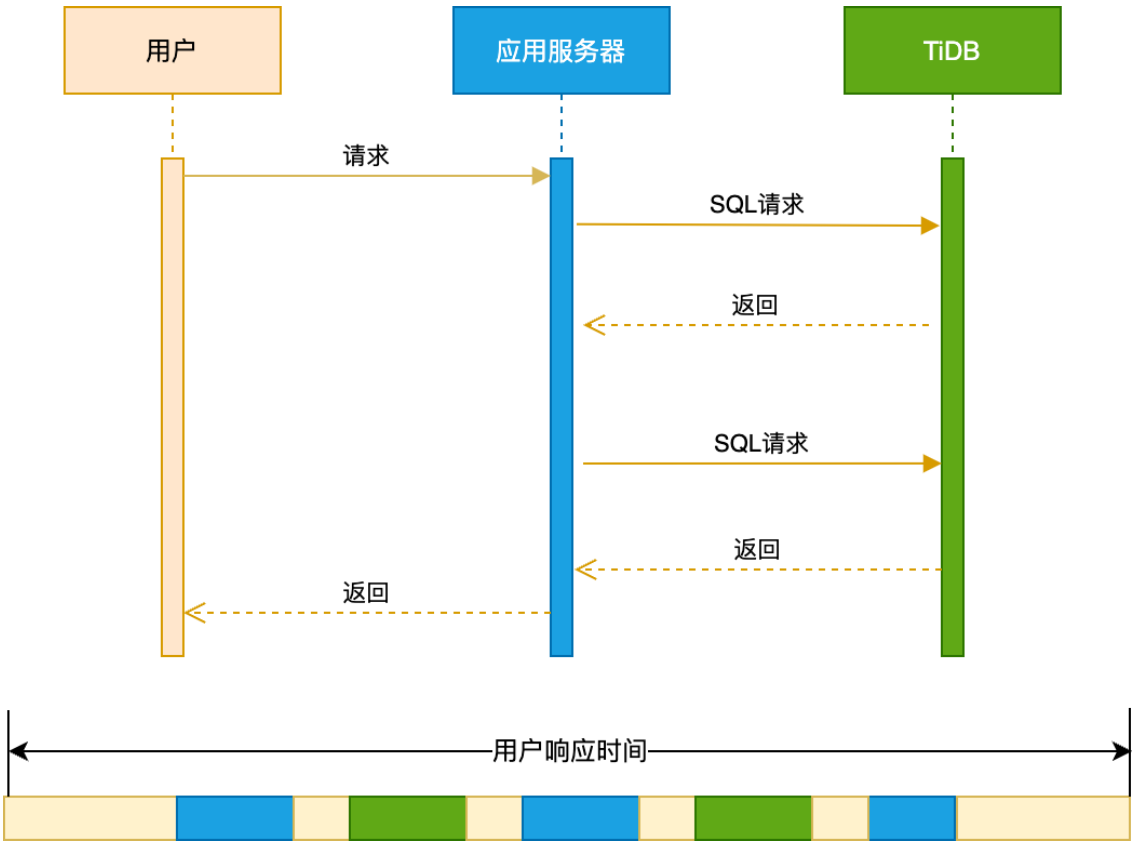


图 68: 用户响应时间

8.1.1.1.2 数据库时间

数据库时间是指数据库系统提供服务的时间，ΔT 时间内的数据库时间为数据库并发处理所有应用请求的时间总和。

你可以通过以下任一方式计算数据库时间：

- 方式一：通过 QPS 乘以平均query 延迟乘以ΔT，即 `DB Time in ΔT = QPS × avg latency × ΔT`
- 方式二：通过平均活跃会话数乘以ΔT，即 `DB Time in ΔT = avg active connections × ΔT`
- 方式三：通过TiDB 内部的Prometheus 指标tidb_server_tokens 计算，即`ΔT DB Time = rate(tidb_server_tokens ′→ )× ΔT`

8.1.1.2 用户响应时间和系统吞吐的关系

用户响应时间包含完成用户请求的服务时间、排队时间和并发等待时间，即：

```
User Response time = Service time + Queuing delay + Coherency delay
```

- Service Time（完成用户请求的服务时间）：系统处理请求时需要消耗某种资源的时间，比如数据库完成一次 SQL 请求需要消耗的 CPU 时间。
- Queuing delay（排队延迟时间）：系统处理请求时为了等待某种资源的服务，在队列中等待调度的时间。
- Coherency delay（并发等待延迟）：系统处理请求时为了访问共享资源，需要和其他并发的任务进行通信和协作的时间。

系统吞吐指系统每秒完成的请求数量。用户响应时间和吞吐通常是反比倒数的关系。随着吞吐的上升，系统资源利用率上升，请求服务的排队延迟会随之上升，当资源利用率超过某个拐点，排队延迟会急剧上升。

例如，对于运行 OLTP 负载的数据库系统，当 CPU 利用率超过 65% 之后，CPU 的排队调度延迟会明显上升。因为系统的并发请求不是完全独立的，请求之间存在共享资源的协同和争用，比如不同的数据库请求可能对同样的数据有互斥的加锁操作。当资源利用率上升时，排队和调度延迟上升，这将导致持有的共享资源无法及时释放，反过来延长了其他任务对共享资源的等待时间。

8.1.1.3 性能优化流程

性能优化流程包含以下 6 个步骤：

1. 定义优化目标
2. 建立性能基线
3. 定位用户响应时间的瓶颈
4. 提出优化方案，预估每种方案的收益、风险和成本
5. 实施优化
6. 评估优化结果

一个性能优化项目，经常需要对步骤 2 到 6 进行多次循环，才能达到优化的目标。

8.1.1.3.1 第 1 步：定义优化目标

不同类型系统优化目标不同。例如，对于一个金融核心的 OLTP 系统，优化目标可能是降低交易的长尾延迟；对于一个财务结算系统，优化目标可能是更充分利用硬件资源，缩短批量结算任务时间。

一个好的优化目标应该是容易量化的，比如：

- 好的优化目标：” 业务高峰期上午 9 点到 10 点，转账交易的p99 延迟需要小于 200 毫秒 “
- 差的优化目标：” 系统太慢了没有响应，需要优化 “

定义一个清晰的优化目标有助于指导后续的性能优化工作。

8.1.1.3.2 第 2 步：建立性能基线

为了高效地进行性能优化，你需要采集当前的性能数据以建立性能基线。需要采集的性能数据通常包含以下内容：

- 用户响应时间的平均值和长尾值、应用系统的吞吐

- 数据库时间、Query 延迟和 QPS 等数据库性能数据。

  TiDB 针对不同维度的性能数据进行了完善的测量和存储，例如慢日志、Top SQL、持续性能分析功能和流量可视化等。此外，你还可以对存储在 Prometheus 中的时序指标数据进行历史回溯和对比。

- 资源使用率，包含 CPU、IO 和网络等资源
- 配置信息，比如应用系统、数据库和操作系统的配置

8.1.1.3.3 第 3 步：定位用户响应时间的瓶颈

基于性能基线的数据，定位或者推测用户响应时间的瓶颈。

现实中的应用程序往往没有对用户请求的链路进行完整的测量和记录，因此你无法通过应用程序对用户响应时间进行自上而下有效的分解。

与之相反的是，数据库内部对于query 延迟和吞吐等性能指标记录非常完善。基于数据库时间，你可以判断用户响应时间的瓶颈是否在数据库中。

- 如果瓶颈不在数据库中，需要借助数据库外部搜集的资源利用率，或者对应用程序进行 Profile，以确定数据库外部的瓶颈。常见场景包括应用程序或者代理服务器资源不足，应用程序存在串行点无法充分利用硬件资源等。
- 如果瓶颈存在数据库中，你可以通过数据库完善的调优工具进行数据库内部性能分析和诊断。常见场景包括存在慢 SQL、应用程序使用数据库的方式不合理、数据库存在读写热点等。

具体的分析诊断方法和工具，请参考性能优化方法。

8.1.1.3.4 第 4 步：提出优化方案，评估每种方案的收益、风险和成本

通过性能分析确定系统瓶颈点之后，根据实际情况提出低成本、低风险、并能获得最大的收益的优化方案。

根据阿姆达尔定律，性能优化的最大收益，取决于优化的部分在整个系统的占比。因此，你需要根据性能数据，确认系统瓶颈和相应的占比，预估瓶颈解决或者优化之后的收益。

需要注意的是，即使某个方案针对最大瓶颈点的优化潜在收益最大，也需要同时评估该方案的风险和成本。例如：

- 对于资源过载的系统，最直接的优化方案是扩容，但是实际中可能因为扩容方案成本太高而无法被采纳。
- 当某个业务模块里的一个慢 SQL 导致整个模块的响应时间很慢时，升级到数据库新版本的方案可以解决这个慢 SQL 问题，但是同时可能影响原来没有问题的模块，因此该方案可能存在潜在的高风险。一个低风险的方案是不升级数据库版本，直接改写现有慢 SQL，在当前数据库版本中解决该问题。

8.1.1.3.5 第 5 步：实施优化

综合考量收益、风险和成本，选定一种或者多种优化方案进行实施，并对生产系统的变更进行周全的准备和详细的记录。

为了降低风险和验证优化方案的收益，建议在测试环境和准生产环境对变更的内容进行验证和完整的回归。例如，针对一个查询业务的慢 SQL，如果选定的优化方案是新建索引优化查询的访问路径，你需要确保新的索引不会在现有的数据插入业务中引入明显的写入热点，导致其他业务变慢。

8.1.1.3.6 第 6 步：评估优化结果

实施优化之后，需要评估优化结果。

- 如果达到优化目标，整个优化项目顺利完成。
- 如果未达到优化目标，你需要重复步骤 2 到 6，直到达到优化目标。

达到优化目标之后，为了应对业务的增长，你可能还需要进一步做好系统的容量规划。

8.1.2 TiDB 性能分析和优化

本文介绍了基于数据库时间的系统优化方法，以及如何利用TiDB Performance Overview 面板进行性能分析和优化。

通过本文中介绍的方法，你可以从全局、自顶向下的角度分析用户响应时间和数据库时间，确认用户响应时间的瓶颈是否在数据库中。如果瓶颈在数据库中，你可以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。

8.1.2.1 基于数据库时间的性能优化方法

TiDB 对 SQL 的处理路径和数据库时间进行了完善的测量和记录，方便定位数据库的性能瓶颈。即使在用户响应时间的性能数据缺失的情况下，基于TiDB 数据库时间的相关性能指标，你也可以达到以下两个性能分析目标：

1. 通过对比 SQL 处理平均延迟和事务中TiDB 连接的空闲时间，确定整个系统的瓶颈是否在TiDB 中。
2. 如果瓶颈在TiDB 内部，根据数据库时间概览、颜色优化法、关键指标和资源利用率、自上而下的延迟分解，定位到性能瓶颈具体在整个分布式系统的哪个模块。

8.1.2.1.1 确定整个系统的瓶颈是否在TiDB 中

- 如果事务中TiDB 连接的平均空闲时间比 SQL 平均处理延迟高，说明应用的事务处理中，主要的延迟不在数据库中，数据库时间占用户响应时间比例小，可以确认瓶颈不在数据库中。

  在这种情况下，需要关注数据库外部的组件，比如应用服务器硬件资源是否存在瓶颈，应用到数据库的网络延迟是否过高等。

- 如果 SQL 平均处理延迟比事务中TiDB 连接的平均空闲时间高，说明事务中主要的瓶颈在TiDB 内部，数据库时间占用户响应时间比例大。

8.1.2.1.2 如果瓶颈在TiDB 内部，如何定位

一个典型的 SQL 的处理流程如下所示，TiDB 的性能指标覆盖了绝大部分的处理路径，对数据库时间进行不同维度的分解和上色，用户可以快速的了解负载特性和数据库内部的瓶颈。

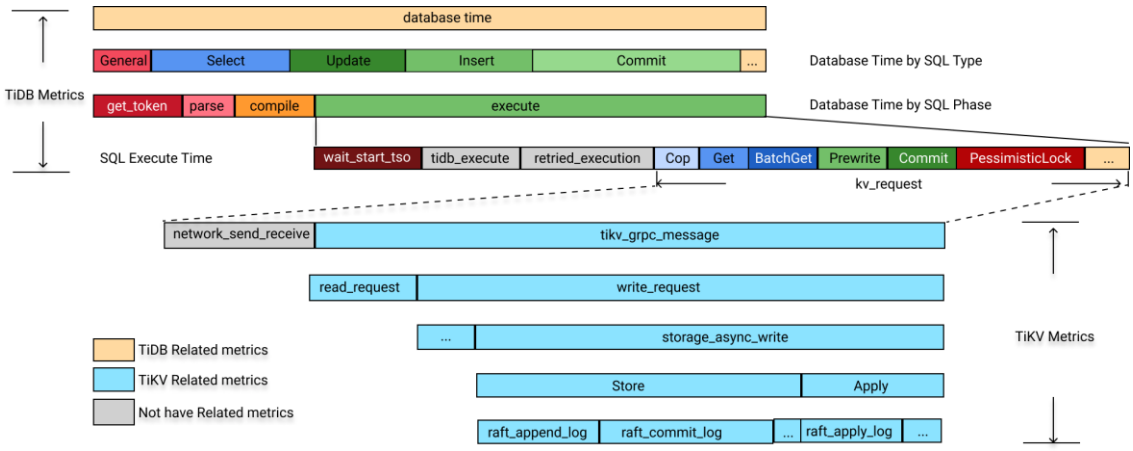


图 69: 数据库时间分解图

数据库时间是所有 SQL 处理时间的总和。通过以下三个维度对数据库时间进行分解，可以帮助你快速定位 TiDB 内部瓶颈：

- 按 SQL 处理类型分解，判断哪种类型的 SQL 语句消耗数据库时间最多。对应的分解公式为：

  ```
  DB Time = Select Time + Insert Time + Update Time + Delete Time + Commit Time + ...
  ```

- 按 SQL 处理的 4 个步骤（即 get_token/parse/compile/execute）分解，判断哪个步骤消耗的时间最多。对应的分解公式为：

  ```
  DB Time = Get Token Time + Parse Time + Comiple Time + Execute Time
  ```

- 对于 execute 耗时，按照TiDB 执行器本身的时间、TSO 等待时间、KV 请求时间和重试的执行时间，判断执行阶段的瓶颈。对应的分解公式为：

  ```
  Execute Time ~= TiDB Executor Time + KV Request Time + PD TSO Wait Time + Retried execution
  ↪ time
  ```

8.1.2.2 利用 Performance Overview 面板进行性能分析和优化

本章介绍如何利用Grafana 中的 Performance Overview 面板进行基于数据库时间的性能分析和优化。

Performance Overview 面板按总分结构对TiDB、TiKV、PD 的性能指标进行编排组织，包括以下三个部分：

- 数据库时间和 SQL 执行时间概览：通过颜色标记不同 SQL 类型，SQL 不同执行阶段、不同请求的数据库时间，帮助你快速识别数据库负载特征和性能瓶颈。
- 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：包括 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。

8.1.2.2.1 数据库时间和 SQL 执行时间概览

Database Time 指标为TiDB 每秒处理 SQL 的延迟总和，即TiDB 集群每秒并发处理应用 SQL 请求的总时间(等于活跃连接数)。

Performance Overview 面板提供了以下三个面积堆叠图，帮助你了解数据库负载的类型，快速定位数据库时间的瓶颈主要是处理什么语句，集中在哪个执行阶段，SQL 执行阶段主要等待TiKV 或者 PD 哪种请求类型。

- Database Time By SQL Type
- Database Time By SQL Phase
- SQL Execute Time Overview

颜色优化法

通过观察数据库时间分解图和执行时间概览图，你可以直观地区分正常或者异常的时间消耗，快速定位集群的异常瓶颈点，高效了解集群的负载特征。对于正常的时间消耗和请求类型，图中显示颜色为绿色系或蓝色系。如果非绿色或蓝色系的颜色在这两张图中占据了明显的比例，意味着数据库时间的分布不合理。

- Database Time By SQL Type：蓝色标识代表 Select 语句，绿色标识代表 Update、Insert、Commit 等 DML 语句。红色标识代表General 类型，包含 StmtPrepare、StmtReset、StmtFetch、StmtClose 等命令。
- Database Time By SQL Phase：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。一个常见的场景是因为无法使用执行计划缓存，导致compile 阶段的橙色占比明显。
- SQL Execute Time Overview：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（例如 Cop 和Get），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。

示例 1：TPC-C 负载

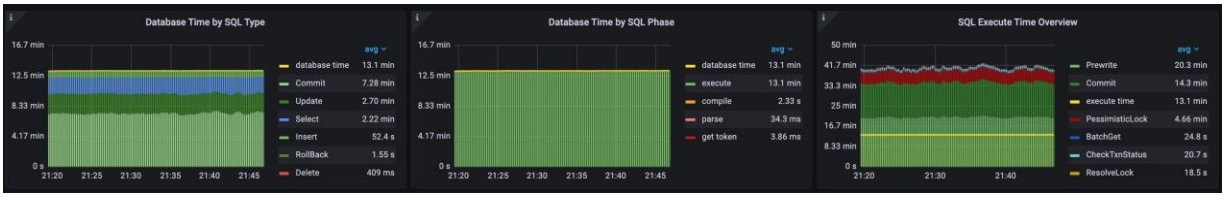


图 70: TPC-C

- Database Time by SQL Type：主要消耗时间的语句为commit、update、select 和 insert 语句。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间的 KV 请求为绿色的 Prewrite 和 Commit。

> 注意：

- KV 请求的总时间大于 execute time 为正常现象，因为 TiDB 执行器可能并发向多个 TiKV 发送 KV 请求，导致总的 KV 请求等待时间大于 execute time。TPC-C 负载中，事务提交时，TiDB 会向多个TiKV 并行发送 Prewrite 和 Commit 请求，所以这个例子中 Prewrite、Commit 和 PessimisticsLock 请求的总时间明显大于 execute time。
- execute time 也可能明显大于 KV 请求的总时间加上tso_wait 的时间，这意味着 SQL 执行阶段主要时间花在TiDB 执行器内部。两种常见的例子：
  - 例 1：TiDB 执行器从TiKV 读取大量数据之后，需要在TiDB 内部进行复杂的关联和聚合，消耗大量时间。
  - 例 2：应用的写语句锁冲突严重，频繁锁重试导致 `Retried execution time` 过长。

示例 2：OLTP 读密集负载

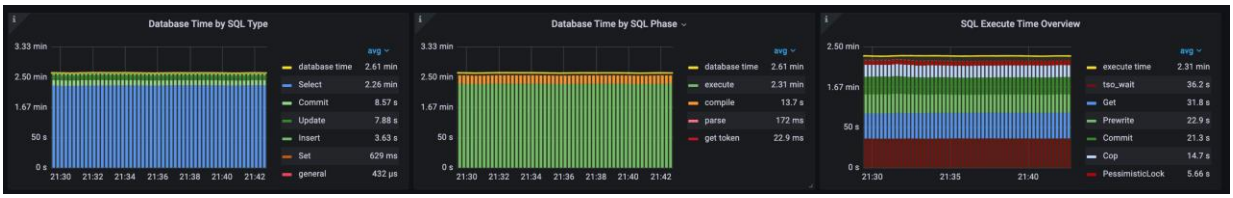


图 71: OLTP

- Database Time by SQL Type：主要消耗时间的语句为 select、commit、update 和 insert 语句。其中，select 占据绝大部分的数据库时间。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间为深褐色的 pd tso_wait、蓝色的 KV Get 和绿色的 Prewrite 和 Commit。

示例 3：只读 OLTP 负载

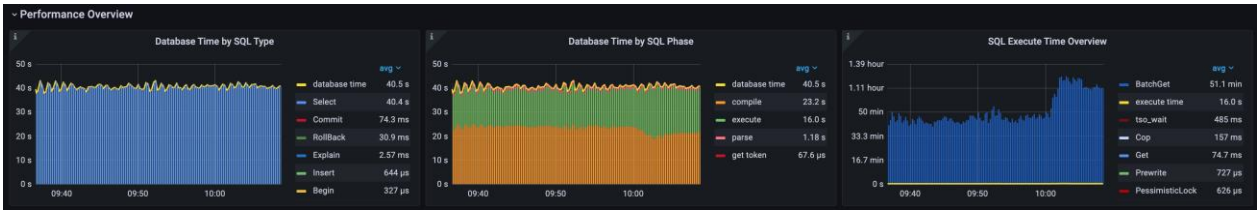


图 72: OLTP

- Database Time by SQL Type：几乎所有语句为 select。
- Database Time by SQL Phase：主要消耗时间的阶段为橙色的 compile 和绿色的 execute 阶段。compile 阶段延迟最高，代表着TiDB 生成执行计划的过程耗时过长，需要根据后续的性能数据进一步确定根源。
- SQL Execute Time Overview：执行阶段主要消耗时间的 KV 请求为蓝色 BatchGet。

> 注意：
>
> 示例 3 select 语句需要从多个TiKV 并行读取几千行数据，BatchGet 请求的总时间远大于执行时间。

示例 4：锁争用负载



图 73: OLTP

- Database Time by SQL Type：主要为 Update 语句。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间的 KV 请求为红色的悲观锁 PessimisticLock，execute time 明显大于KV 请求的总时间，这是因为应用的写语句锁冲突严重，频繁锁重试导致`Retried execution time`过长。目前 `Retried execution time` 消耗的时间，TiDB 尚未进行测量。

示例 5：HTAP CH-Benchmark 负载



图 74: HTAP

- Database Time by SQL Type：主要为 Select 语句。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间为紫色的 `tiflash_mpp` 请求，其次是 KV 请求，包括蓝色的 `Cop`，以及绿色的 `Prewrite` 和 `Commit`。

8.1.2.2.2 TiDB 关键指标和集群资源利用率

Query Per Second、 Command Per Second 和 Prepared-Plan-Cache

通过观察 Performance Overview 里的以下三个面板，可以了解应用的负载类型，与TiDB 的交互方式，以及是否能有效地利用TiDB 的执行计划缓存。

- QPS：表示 Query Per Second，包含应用的 SQL 语句类型执行次数分布。
- CPS By Type：CPS 表示 Command Per Second，Command 代表 MySQL 协议的命令类型。同样一个查询语句可以通过query 或者 prepared statement 的命令类型发送到TiDB。
- Queries Using Plan Cache OPS：TiDB 集群每秒执行计划缓存的命中次数（即 `avg-hit`）和未命中次数（即 `avg-miss`）。

  StmtExecute 每秒执行次数等于 `avg-hit + avg-miss`。执行计划缓存只支持 prepared statement 命令。当 TiDB 开启执行计划缓存时，存在三种使用情况：

  - 完全无法命中执行计划缓存：每秒命中次数 `avg-hit` 为 0，`avg-miss` 等于 StmtExecute 命令每秒执行次数。可能的原因包括：
    * 应用使用了query 命令。
    * 每次 StmtExecute 执行之后，应用调用了 StmtClose 命令，导致缓存的执行计划被清理。
    * StmtExecute 执行的所有语句都不符合缓存的条件，导致无法命中执行计划缓存。
  - 完全命中执行计划缓存：每秒命中次数 `avg-hit` 等于 StmtExecute 命令每秒执行次数，每秒未命中次数 `avg-miss` 等于 0。
  - 部分命中执行计划缓存：每秒命中次数 `avg-hit` 小于 StmtExecute 命令每秒执行次数。执行计划缓存目前存在一些限制，比如不支持子查询，该类型的 SQL 执行计划无法被缓存。

示例 1：TPC-C 负载

TPC-C 负载类型主要以 Update、Select 和Insert 语句为主。总的 QPS 等于每秒 StmtExecute 的次数，并且 StmtExecute 每秒的数据基本等于 Queries Using Plan Cache OPS 面板的 `avg-hits`。这是 OLTP 负载理想的情况，客户端执行使用 prepared statement，并且在客户端缓存了 prepared statement 对象，执行每条 SQL 语句时直接调用 statement 执行。执行时都命中执行计划缓存，不需要重新compile 生成执行计划。

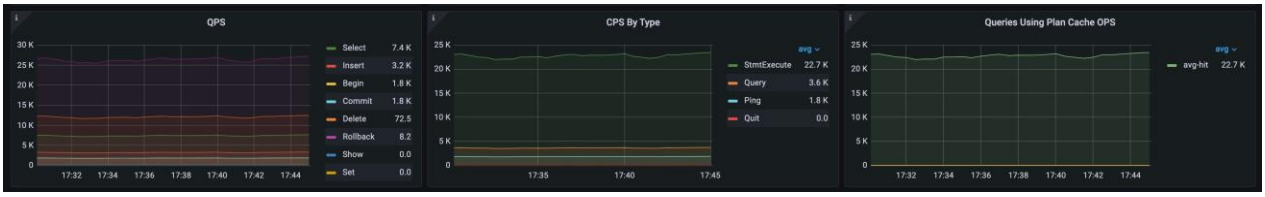


图 75: TPC-C

示例 2：只读 OLTP 负载，使用query 命令无法使用执行计划缓存

这个负载中，Commit QPS = Rollback QPS = Select QPS。应用开启了 auto-commit 并发，每次从连接池获取连接都会执行rollback，因此这三种语句的执行次数是相同的。



图 76: OLTP-Query

- QPS 面板中出现的红色加粗线为 Failed Query，坐标的值为右边的 Y 轴。非 0 代表此负载中存在错误语句。
- 总的 QPS 等于 CPS By Type 面板中的 Query，说明应用中使用了 query 命令。
- Queries Using Plan Cache OPS 面板没有数据，因为不使用prepared statement 接口，无法使用TiDB 的执行计划缓存，意味着应用的每一条query，TiDB 都需要重新解析，重新生成执行计划。通常会导致compile 时间变长以及TiDB CPU 消耗的增加。

示例 3：OLTP 负载，使用prepared statement 接口无法使用执行计划缓存

StmtPreare 次数 = StmtExecute 次数 = StmtClose 次数 ~= StmtFetch 次数，应用使用了prepare > execute > fetch > close 的 loop，很多框架都会在 execute 之后调用close，确保资源不会泄露。这会带来两个问题：

- 执行每条 SQL 语句需要 4 个命令，以及 4 次网络往返。
- Queries Using Plan Cache OPS 为 0，无法命中执行计划缓存。StmtClose 命令默认会清理缓存的执行计划，导致下一次 StmtPreare 命令需要重新生成执行计划。

> 注意：
>
> 从 TiDB v6.0.0 起，你可以通过全局变量(`set global tidb_ignore_prepared_cache_close_stmt =on;`) 控制 StmtClose 命令不清理已被缓存的执行计划，使得下一次的 SQL 的执行不需要重新生成执行计划。

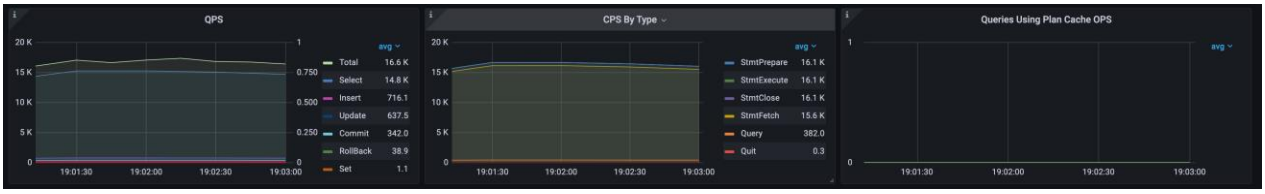


图 77: OLTP-Prepared

示例 4：Prepared Statement 存在资源泄漏

StmtPrepare 每秒执行次数远大于 StmtClose，说明应用程序存在 prepared statement 对象泄漏。

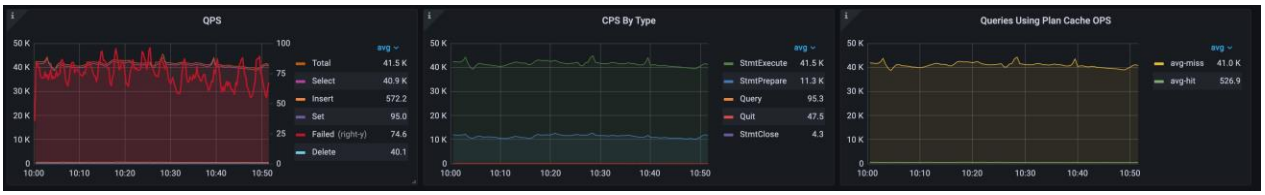


图 78: OLTP-Query

- QPS 面板中出现的红色加粗线为 Failed Query，坐标的值为右边的 Y 轴。每秒错误语句为 74.6 条。

- CPS By Type 面板中的 StmtPrepare 每秒执行次数远大于 StmtClose，说明应用程序存在 prepared statement 对象泄漏。
- Queries Using Plan Cache OPS 面板中的 `avg-miss` 几乎等于 CPS By Type 面板中的 StmtExecute，说明几乎所有的 SQL 执行都未命中执行计划缓存。

KV/TSO Request OPS 和 KV Request Time By Source

- 在 KV/TSO Request OPS 面板中，你可以查看 KV 和TSO 每秒请求的数据统计。其中，`kv request total` 代表 TiDB 到TiKV 所有请求的总和。通过观察TiDB 到 PD 和TiKV 的请求类型，可以了解集群内部的负载特征。
- 在 KV Request Time By Source 面板中，你可以查看每种 KV 请求和请求来源的时间占比。
  - kv request total time 是每秒总的 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）。

示例 1：繁忙的负载



图 79: TPC-C

在此 TPC-C 负载中：

- 每秒总的 KV 请求的数量为 79.7 K。按请求数量排序，最高的请求类型为 `Prewrite`、`Commit`、`PessimisticsLock` 和 `BatchGet` 等。
- KV 处理时间来源主要为 `Commit-external_Commit`、`Prewrite-external_Commit`，说明消耗时间最高的 KV 请求为 `Commit` 和 `Prewrite`，并且来源于外部的 Commit 语句。

示例 2：Analyze 负载



图 80: OLTP

集群中只有analyze 语句运行：

- 每秒总的 KV 请求数据是 35.5，Cop 请求次数是每秒 9.3。

- KV 处理时间主要来源为 `Cop-internal_stats`，说明 Cop 请求来源于内部的analyze 操作。

TiDB CPU，以及 TiKV CPU 和 IO 使用情况

在 TiDB CPU 和 TiKV CPU/IO MBps 这两个面板中，你可以观察到TiDB 和 TiKV 的逻辑 CPU 使用率和IO 吞吐，包含平均、最大和delta（最大 CPU 使用率减去最小 CPU 使用率），从而用来判定 TiDB 和 TiKV 总体的 CPU 使用率。

- 通过 `delta` 值，你可以判断TiDB 是否存在 CPU 使用负载不均衡（通常伴随着应用连接不均衡），TiKV 是否存在热点。
- 通过TiDB 和 TiKV 的资源使用概览，你可以快速判断集群是否存在资源瓶颈，最需要扩容的组件是TiDB 还是TiKV。

示例 1：TiDB 资源使用率高

下图负载中，每个TiDB 和 TiKV 配置 8 CPU。

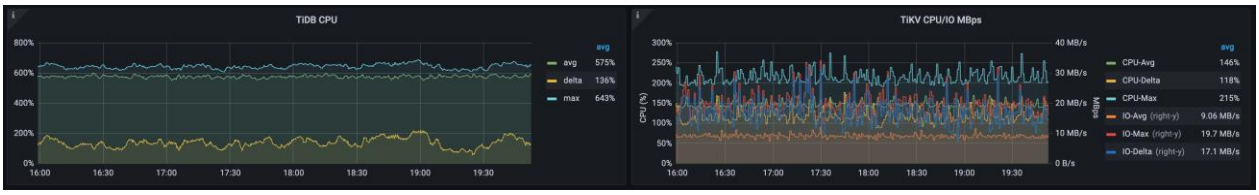


图 81: TPC-C

- TiDB 平均 CPU 为 575%。最大 CPU 为 643%，delta CPU 为 136%。
- TiKV 平均 CPU 为 146%，最大 CPU 215%。delta CPU 为 118%。TiKV 的平均IO 吞吐为 9.06 MB/s，最大 IO 吞吐为 19.7 MB/s，delta IO 吞吐为 17.1 MB/s。

由此可以判断，TiDB 的 CPU 消耗明显更高，并接近于 8 CPU 的瓶颈，可以考虑扩容TiDB。

示例 2：TiKV 资源使用率高

下图 TPC-C 负载中，每个TiDB 和 TiKV 配置 16 CPU。



图 82: TPC-C

- TiDB 平均 CPU 为 883%。最大 CPU 为 962%，delta CPU 为 153%。
- TiKV 平均 CPU 为 1288%，最大 CPU 1360%。delta CPU 为 126%。TiKV 的平均IO 吞吐为 130 MB/s，最大 IO 吞吐为 153 MB/s，delta IO 吞吐为 53.7 MB/s。

由此可以判断，TiKV 的 CPU 消耗更高，因为 TPC-C 是一个写密集场景，这是正常现象，可以考虑扩容TiKV 节点提升性能。

8.1.2.2.3 Query 延迟分解和关键的延迟指标

延迟面板通常包含平均值和 99 分位数，平均值用来定位总体的瓶颈，99 分位数用来判断是否存在延迟严重抖动的情况。判断性能抖动范围时，可能还需要需要借助 999 分位数。

Duration、 Connection Idle Duration 和 Connection Count

Duration 面板包含了所有语句的 99 延迟和每种 SQL 类型的平均延迟。Connection Idle Duration 面板包含连接空闲的平均和 99 延迟，连接空闲时包含两种状态：

- in-txn：代表事务中连接的空闲时间，即当连接处于事务中时，处理完上一条 SQL 之后，收到下一条 SQL 语句的间隔时间。
- not-in-txn：当连接没有处于事务中，处理完上一条 SQL 之后，收到下一条 SQL 语句的间隔时间。

应用进行数据库事务时，通常使用同一个数据库连接。对比query 的平均延迟和connection idle duration 的延迟，可以判断整个系统性能瓶颈或者用户响应时间的抖动是否是由TiDB 导致的。

- 如果应用负载不是只读的，包含事务，对比query 平均延迟和 `avg-in-txn` 可以判断应用处理事务时，主要的时间是花在数据库内部还是在数据库外面，借此定位用户响应时间的瓶颈。
- 如果是只读负载，不存在 `avg-in-txn` 指标，可以对比query 平均延迟和 `avg-not-in-txn` 指标。

现实的客户负载中，瓶颈在数据库外面的情况并不少见，例如：

- 客户端服务器配置过低，CPU 资源不够。
- 使用 HAProxy 作为TiDB 集群代理，但是 HAProxy CPU 资源不够。
- 使用 HAProxy 作为TiDB 集群代理，但是高负载下 HAProxy 服务器的网络带宽被打满。
- 应用服务器到数据库延迟过高，比如公有云环境应用和TiDB 集群不在同一个地区，比如数据库的 DNS 均衡器和TiDB 集群不在同一个地区。
- 客户端程序存在瓶颈，无法充分利用服务器的多 CPU 核或者多 Numa 资源，比如应用只使用一个JVM 向 TiDB 建立上千个JDBC 连接。

在 Connection Count （连接信息）面板中，你可以查看总的连接数和每个TiDB 节点的连接数，并由此判断连接总数是否正常，各TiDB 节点的连接数是否不均衡。`active connections` 记录着活跃连接数，等于每秒的数据库时间，右侧Y 轴为 `disconnection/s`，代表集群每秒断开连接的数量，用来判断应用是否使用了短连接。

示例 1：disconnection/s 过高



图 83: high disconnection/s

在此负载中：

- 所有 SQL 语句的平均延迟 10.8 ms，P99 延迟 84.1 ms。

- 事务中连接空闲时间 avg-in-txn 为 9.4 ms。
- 集群总的连接数为 3.7K，每个TiDB 节点的连接数为 1.8 K。平均活跃连接数为 40.3，大部分连接处于空闲状态。disonnnection/s 平均为 55.8，说明应用在频繁的新建和断开连接。短连接的行为会对TiDB 的资源和响应时间造成一定的影响。

示例 2：用户响应时间的瓶颈在TiDB 中



图 84: TiDB is the Bottleneck

在此 TPC-C 负载中：

- 所有 SQL 语句的平均延迟 477 us，99 延迟 3.13 ms。平均commit 语句 2.02 ms，平均insert 语句 609 us，平均查询语句 468 us。
- 事务中连接空闲时间 avg-in-txn 171 us。

由此可以判断，平均的query 延迟明显大于 avg-in-txn，说明事务处理中，主要的瓶颈在数据库内部。

示例 3：用户响应时间的瓶颈不在TiDB 中



图 85: TiDB is not the Bottleneck

在此负载中，平均query 延迟为 1.69 ms，事务中连接空闲时间 avg-in-txn 为 18 ms。说明事务中，TiDB 平均花了 1.69 ms 处理完一个 SQL 语句之后，需要等待 18 ms 才能收到下一条语句。

由此可以判断，用户响应时间的瓶颈不在TiDB 中。这个例子是在一个公有云环境下，因为应用和数据库不在同一个地区，应用和数据库之间的网络延迟高导致了超高的连接空闲时间。

Parse、 Compile 和 Execute Duration

在TiDB 中，从输入查询文本到返回结果的典型处理流程。

SQL 在 TiDB 内部的处理分为四个阶段，get token、 parse、 compile 和 execute：

- get token 阶段：通常只有几微秒的时间，可以忽略。除非TiDB 单个实例的连接数达到的token-limit 的限制，创建连接的时候被限流。

- parse 阶段：query 语句解析为抽象语法树abstract syntax tree (AST)。
- compile 阶段：根据parse 阶段输出的AST 和统计信息，编译出执行计划。整个过程主要步骤为逻辑优化与物理优化，前者通过一些规则对查询计划进行优化，例如基于关系代数的列裁剪等，后者通过统计信息和基于成本的优化器，对执行计划的成本进行估算，并选择整体成本最小的物理执行计划。
- execute 阶段：时间消耗视情况，先等待全局唯一的时间戳TSO，之后执行器根据执行计划中算子涉及的Key 范围，构建出TiKV 的API 请求，分发到TiKV。execute 时间包含TSO 等待时间、KV 请求的时间和TiDB 执行器本身处理数据的时间。

如果应用统一使用query 或者 StmtExecute MySQL 命令接口，可以使用以下公式来定位平均延迟的瓶颈。

```
avg Query Duration = avg Get Token + avg Parse Duration + avg Compile Duration + avg Execute
    ↪ Duration
```

通常 execute 阶段会占query 延迟的主要部分，在以下情况下，parse 和 compile 阶段也会占比明显。

- parse 阶段延迟占比明显：比如query 语句很长，文本解析消耗大量的 CPU。
- compile 阶段延迟占比明显：如果应用没有使用执行计划缓存，每个语句都需要生成执行计划。compile 阶段的延迟可能达到几毫秒或者几十毫秒。如果无法命中执行计划缓存，compile 阶段需要进行逻辑优化和物理优化，这将消耗大量的 CPU 和内存，并给Go Runtime 带来压力（因为TiDB 是 Go 编写的），进一步影响TiDB 其他组件的性能。这说明，OLTP 负载在TiDB 中是否能高效运行，执行计划缓存扮演了重要的角色。

示例 1：数据库瓶颈在compile 阶段

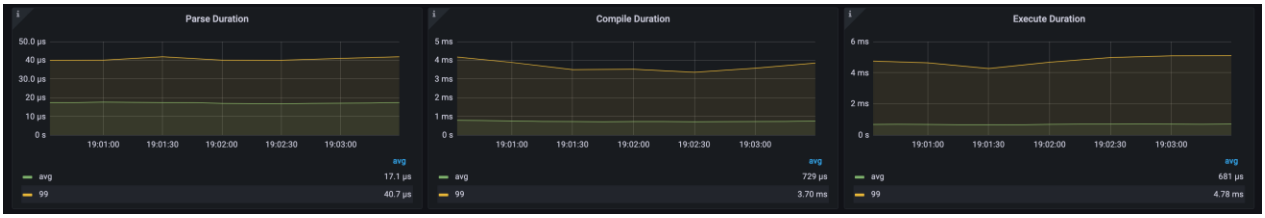


图 86: Compile

此图中parse、compile 和 execute 阶段的平均时间分别为 17.1 us、729 us 和 681 us。因为应用使用query 命令接口，无法使用执行计划缓存，所以compile 阶段延迟高。

示例 2：数据库瓶颈在 execute 阶段

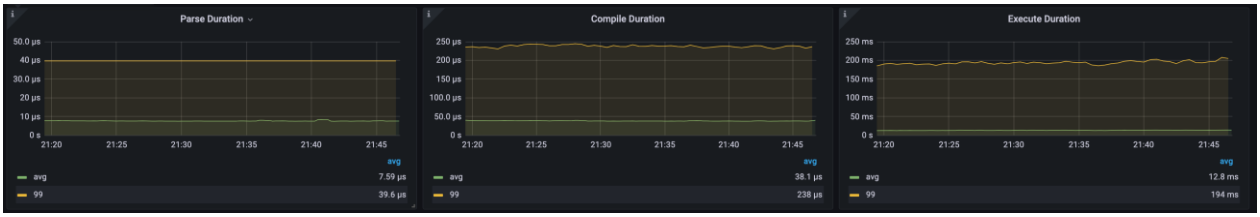


图 87: Execute

在此 TPC-C 负载中，parse、 compile 和 execute 阶段的平均时间分别为 7.39us、 38.1us 和 12.8ms。 query 延迟的瓶颈在于 execute 阶段。

KV 和 TSO Request Duration

在 execute 阶段，TiDB 会跟 PD 和 TiKV 进行交互。如下图所示，当TiDB 处理 SQL 语句请求时，在进行parse 和 compile 之前，如果需要获取TSO，会先请求生成 TSO。PD Client 不会阻塞调用者，而是直接返回一个 TSFuture，并在后台异步处理TSO 请求的收发，一旦完成立即返回给TSFuture，TSFuture 的持有者则需要调用Wait 方法来获得最终的TSO 结果。 当TiDB 完成parse 和 compile 之后，进入 execute 阶段，此时存在两个情况：

- 如果TSO 请求已经完成，Wait 方法会立刻返回一个可用的TSO 或 error
- 如果TSO 请求还未完成，Wait 方法会block 住等待一个可用的TSO 或 error（说明 gRPC 请求已发送但尚未收到返回结果，网络延迟较高）

TSO 等待的时间记录为TSO WAIT，TSO 请求的网络时间记录为TSO RPC。TiDB TSO 等待完成之后，执行过程中通常需要和TiKV 进行读写交互：

- 读的 KV 请求常见类型：Get、 BatchGet 和 Cop
- 写的 KV 请求常见类型：PessimisticLock，二阶段提交的 Prewrite 和 Commit

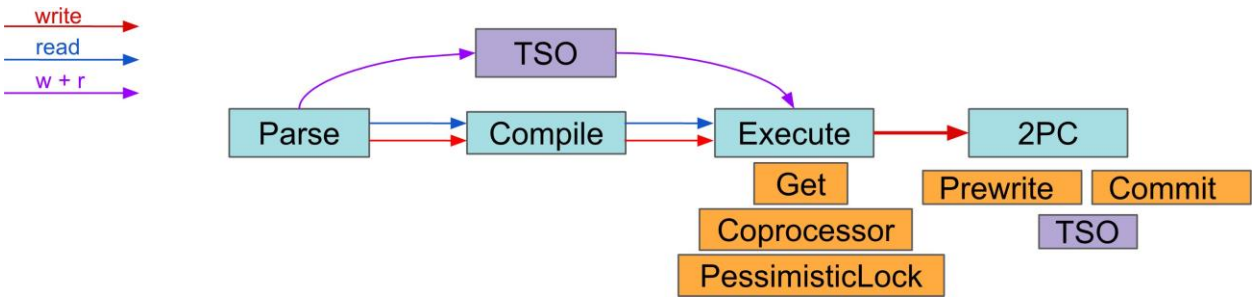


图 88: Execute

这一部分的指标对应以下三个面板：

- Avg TiDB KV Request Duration：TiDB 测量的 KV 请求的平均延迟
- Avg TiKV GRPC Duration：TiKV 内部GRPC 消息处理的平均延迟
- PD TSO Wait/RPC Duration：TiDB 执行器等待TSO 延迟(wait) 和 TSO 请求的网络延迟(rpc)。

其中，Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的关系如下

```
Avg TiDB KV Request Duration = Avg TiKV GRPC Duration + TiDB 与 TiKV 之间的网络延迟 + TiKV GRPC
    ↪ 处理时间 + TiDB GRPC 处理时间和调度延迟。
```

Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的差值跟网络流量和延迟，TiDB 和 TiKV 的资源使用情况密切相关。

- 同一个机房内，Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的差值通常应该小于 2 毫秒。

- 同一地区的不同可用区，Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的差值通常应该小于 5 毫秒。

示例 1：同机器低负载的集群

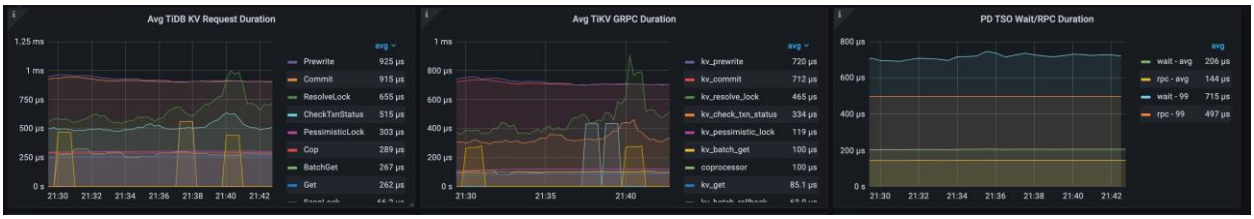


图 89: Same Data Center

在此负载中，TiDB 侧平均 Prewrite 请求延迟为 925 us，TiKV 内部 kv_prewrite 平均处理延迟为 720 us，相差 200 us 左右，是同机房内正常的延迟。TSO wait 平均延迟 206 us，rpc 时间为 144 us。

示例 2：公有云集群，负载正常

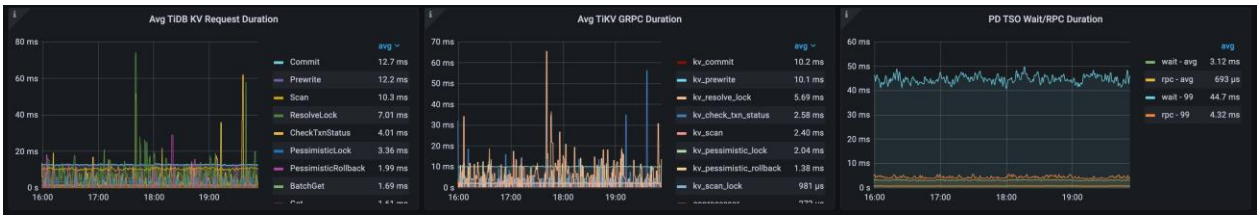


图 90: Cloud Env

在此示例中，TiDB 集群部署在同一个地区的不同机房。TiDB 侧平均 Commit 请求延迟为 12.7 ms，TiKV 内部 kv_commit 平均处理延迟为 10.2 ms，相差 2.5 ms 左右。TSO wait 平均延迟为 3.12 ms，rpc 时间为 693 us。

示例 3：公有云集群，资源严重过载

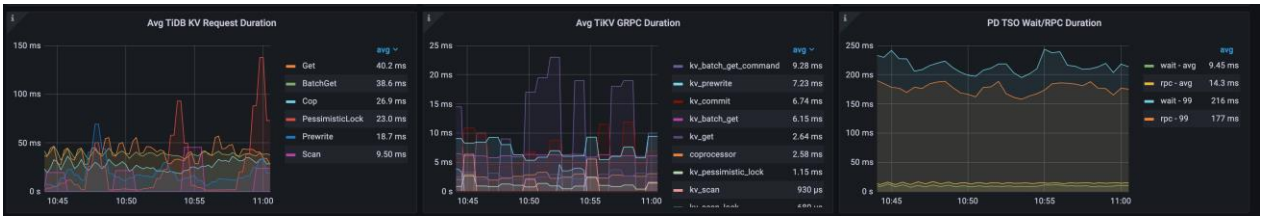


图 91: Cloud Env, TiDB Overloaded

在此示例中，TiDB 集群部署在同一个地区的不同机房，TiDB 网络和 CPU 资源严重过载。TiDB 侧平均 BatchGet 请求延迟为 38.6 ms，TiKV 内部 kv_batch_get 平均处理延迟为 6.15 ms，相差超过 32 ms，远高于正常值。TSO wait 平均延迟为 9.45 ms，rpc 时间为 14.3 ms。

Storage Async Write Duration、Store Duration 和 Apply Duration

TiKV 对于写请求的处理流程如下图

- `scheduler worker` 会先处理写请求，进行事务一致性检查，并把写请求转化成键值对，发送到`raftstore` 模块。
- `raftstore` 为TiKV 的共识模块，使用Raft 共识算法，使多个TiKV 组成的存储层可以容错。

  Raftstore 分为 store 线程和apply 线程。：

  - store 线程负载处理Raft 消息和新的 `proposals`。当收到新的 `proposals` 时，leader 节点的 store 线程会写入本地Raft DB，并将消息复制到多个 follower 节点。当这个 `proposals` 在多数实例持久化成功之后，`proposals` 成功被提交。
  - apply 线程会负载将提交的内容写入到 KV DB 中。当写操作的内容被成功的写入 KV 数据库中，apply 线程会通知外层请求写请求已经完成。

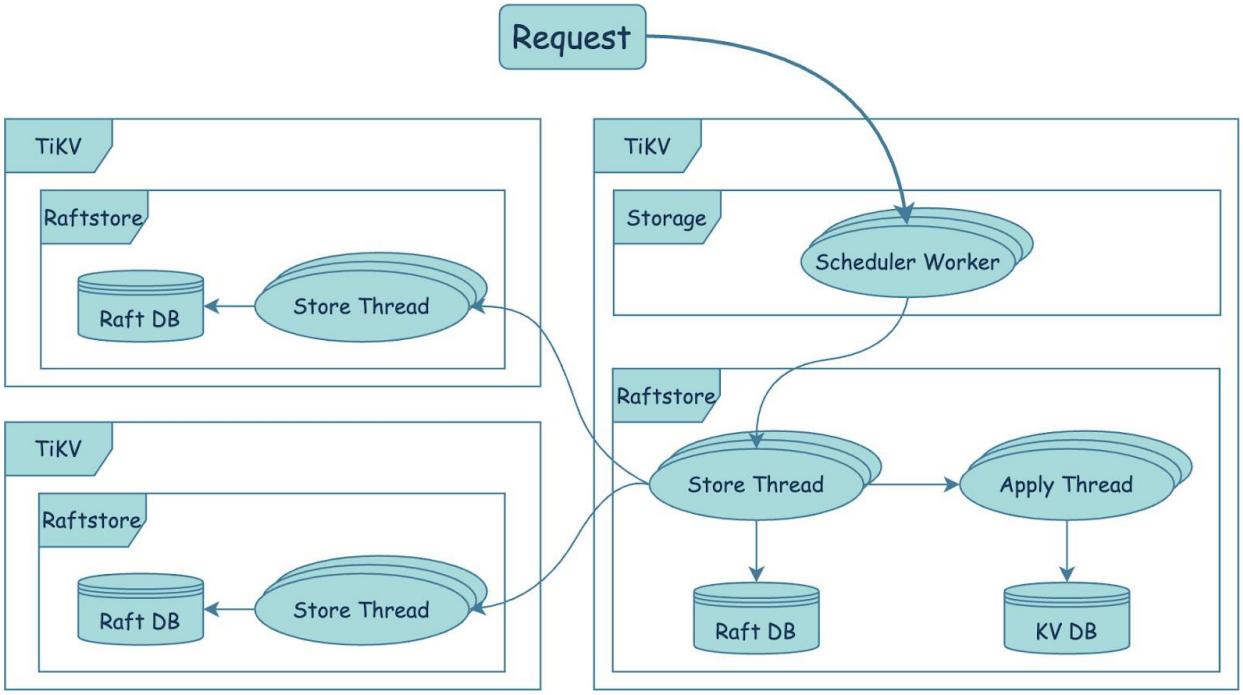


图 92: TiKV Write

Storage Async Write Duration 指标记录写请求进入raftstore 之后的延迟，采集的粒度具体到每个请求的级别。

Storage Async Write Duration 分为 Store Duration 和 Apply Duration。你可以通过以下公式定位写请求的瓶颈主要是在 Store 还是Apply 步骤。

```
avg Storage Async Write Duration  = avg Store Duration + avg Apply Duration
```

> **注意：**
>
> Store Duration 和 Apply Duration 从 v5.3.0 版本开始支持。

示例 1：同一个 OLTP 负载在 v5.3.0 和 v5.4.0 版本的对比

v5.4.0 版本，一个写密集的 OLTP 负载 QPS 比 v5.3.0 提升了 14%。应用以上公式

- v5.3.0：24.4 ms ~= 17.7 ms + 6.59 ms
- v5.4.0：21.4 ms ~= 14.0 ms + 7.33 ms

因为 v5.4.0 版本中，TiKV 对 gRPC 模块进行了优化，优化了Raft 日志复制速度，相比 v5.3.0 降低了 Store Duration。

v5.3.0：



图 93: v5.3.0

v5.4.0：



图 94: v5.4.0

示例 2：Store Duration 瓶颈明显

应用以上公式：10.1 ms ~= 9.81 ms + 0.304 ms，说明写请求的延迟瓶颈在 Store Duration。



图 95: Store

Commit Log Duration、 Append Log Duration 和 Apply Log Duration

Commit Log Duration、 Append Log Duration 和 Apply Log Duration 这三个延迟是raftstore 内部关键操作的延迟记录。这些记录采集的粒度是batch 操作级别的，每个操作会把多个写请求合并在一起，因此不能直接对应上文的 Store Duration 和 Apply Duration。

- Commit Log Duration 和 Append Log Duration 均为 store 线程的操作。Commit Log Duration 包含复制Raft 日志到其他TiKV 节点，保证raft-log 的持久化。一般包含两次Append Log Duration，一次 leader，一次 follower 的。Commit Log Duration 延迟通常会明显高于Append Log Duration，因为包含了通过网络复制 Raft 日志到其他 TiKV 的时间。
- Apply Log Duration 记录了apply 线程apply Raft 日志的延迟。

Commit Log Duration 慢的常见场景：

- TiKV CPU 资源存在瓶颈，调度延迟高
- `raftstore.store-pool-size` 设置过小或者过大（过大也可能导致性能下降）
- IO 延迟高，导致Append Log Duration 延迟高
- TiKV 之间的网络延迟比较高
- TiKV 的 gRPC 线程数设置过小或者多个 gRPC CPU 资源使用不均衡

Apply Log Duration 慢的常见场景：

- TiKV CPU 资源存在瓶颈，调度延迟高
- `raftstore.apply-pool-size` 设置过小或者过大（过大也可能导致性能下降）
- IO 延迟比较高

示例 1：同一个 OLTP 负载在 v5.3.0 和 v5.4.0 版本的对比

v5.4.0 版本，一个写密集的 OLTP 负载 QPS 比 v5.3.0 提升了 14%。对比这三个关键延迟：

Avg Duration	v5.3.0(ms)	v5.4.0(ms)
Append Log Duration	0.27	0.303
Commit Log Duration	13	8.68
Apply Log Duration	0.457	0.514

因为 v5.4.0 版本中，TiKV 对 gRPC 模块进行了优化，优化了 Raft 日志复制速度，相比 v5.3.0 降低了 Commit Log Duration 和 Store Duration。

v5.3.0：

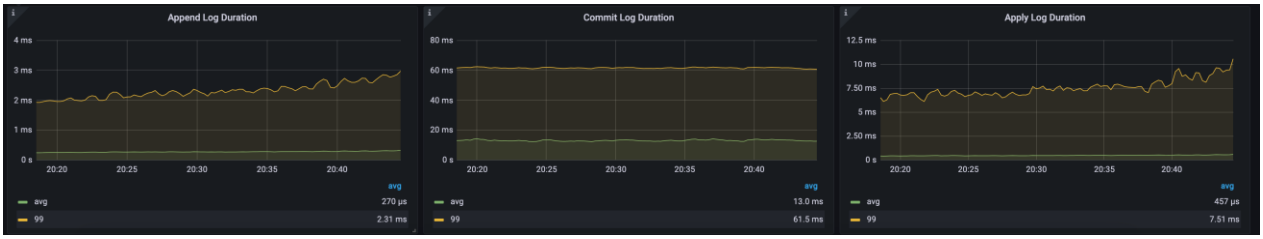


图 96: v5.3.0

v5.4.0：

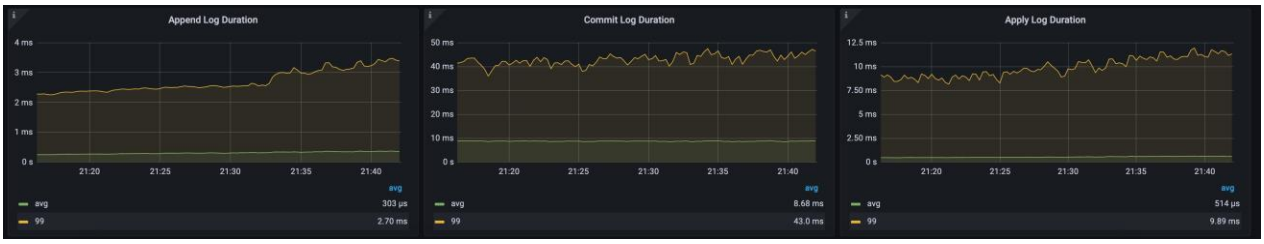


图 97: v5.4.0

示例 2：Commit Log Duration 瓶颈明显的例子



图 98: Store

- 平均 Append Log Duration = 4.38 ms
- 平均 Commit Log Duration = 7.92 ms
- 平均 Apply Log Duration = 172 us

Store 线程的 Commit Log Duration 明显比 Apply Log Duration 高，并且 Append Log Duration 比 Apply Log Duration 明显的高，说明 Store 线程在 CPU 和 IO 都可能都存在瓶颈。可能降低 Commit Log Duration 和 Append Log Duration 的方式如下：

- 如果 TiKV CPU 资源充足，考虑增加 Store 线程，即 `raftstore.store-pool-size`。
- 如果 TiDB 为 v5.4.0 及之后的版本，考虑启用 `Raft Engine`，Raft Engine 具有更轻量的执行路径，在一些场景下显著减少 IO 写入量和写入请求的长尾延迟，启用方式为设置：`raft-engine.enable: true`
- 如果 TiKV CPU 资源充足，且 TiDB 为 v5.3.0 及之后的版本，考虑启用 `StoreWriter`。启用方式：`raftstore.store-io-pool-size: 1`。

8.1.2.3 低于 v6.1.0 的 TiDB 版本如何使用 Performance overview 面板

从 v6.1.0 起，TiDB Grafana 组件默认内置了 Performance Overview 面板。Performance overview 面板兼容 TiDB v4.x 和 v5.x 版本。如果你的 TiDB 版本低于 v6.1.0，需要手动导入 `performance_overview.json`。

导入方法如图所示：

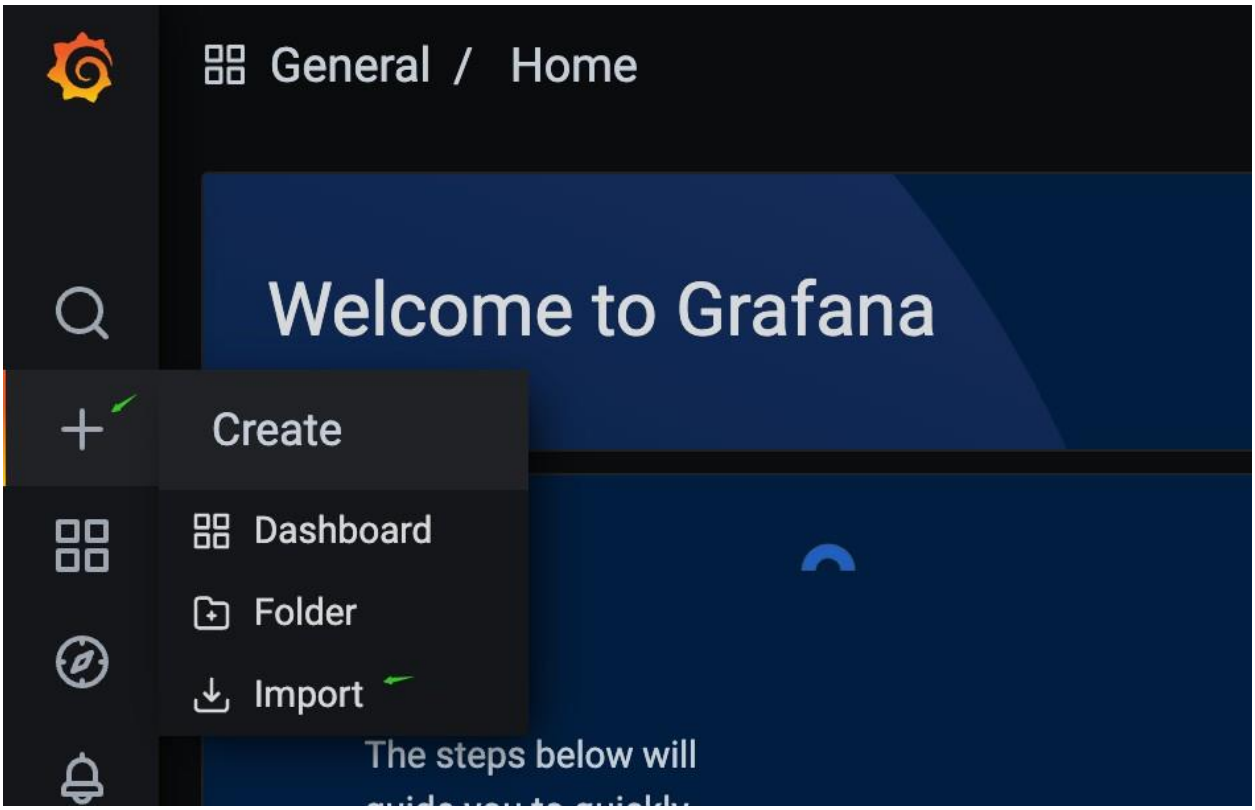


图 99: Store

8.1.3 OLTP 负载性能优化实践

TiDB 提供了完善的性能诊断和分析功能，例如 TiDB Dashboard 的Top SQL 和Continuous Profiling 功能，以及 TiDB Performance Overview 面板。

本文介绍如何综合利用这些功能，对同一个 OLTP 负载在七种不同运行场景下的性能表现进行分析和对比，并演示了具体的 OLTP 负载的优化过程，帮助你更快地对TiDB 的性能进行分析和优化。

> 注意：
>
> Top SQL 和Continuous Profiling 功能默认关闭，需要提前开启。

在这些场景中，通过使用不同的JDBC 配置运行同一个应用程序，你可以观察应用和数据库之间不同的交互方式将如何影响系统整体的性能，从而更好地掌握开发Java 应用使用TiDB 的最佳实践。

8.1.3.1 负载环境

本文使用一个银行交易系统 OLTP 仿真模拟负载进行演示。以下为该负载的仿真环境配置：

- 负载应用程序的开发语言：JAVA

- 涉及业务的 SQL 语句：共 200 条，其中 90% 都是 SELECT 语句，属于典型的读密集 OLTP 场景。
- 涉及交易的表：共 60 张，存在修改操作类的表为 12 张，其余 48 张表只读。
- 应用程序使用的隔离级别：`read committed`。
- TiDB 集群配置：3 个 TiDB 节点和 3 个 TiKV 节点，各节点分配 16 CPU。
- 客户端服务器配置：36 CPU。

8.1.3.2 场景 1：使用 Query 接口

8.1.3.2.1 应用配置

应用程序使用以下JDBC 配置，通过 Query 接口连接数据库。

```
useServerPrepStmts=false
```

8.1.3.2.2 性能分析

TiDB Dashboard

从以下 Dashboard 的 Top SQL 页面可以观察到，非业务 SQL 类型 `SELECT @@session.tx_isolation` 消耗的资源最多。 虽然TiDB 处理这类 SQL 语句的速度快，但由于执行次数最多导致总体 CPU 耗时最多。

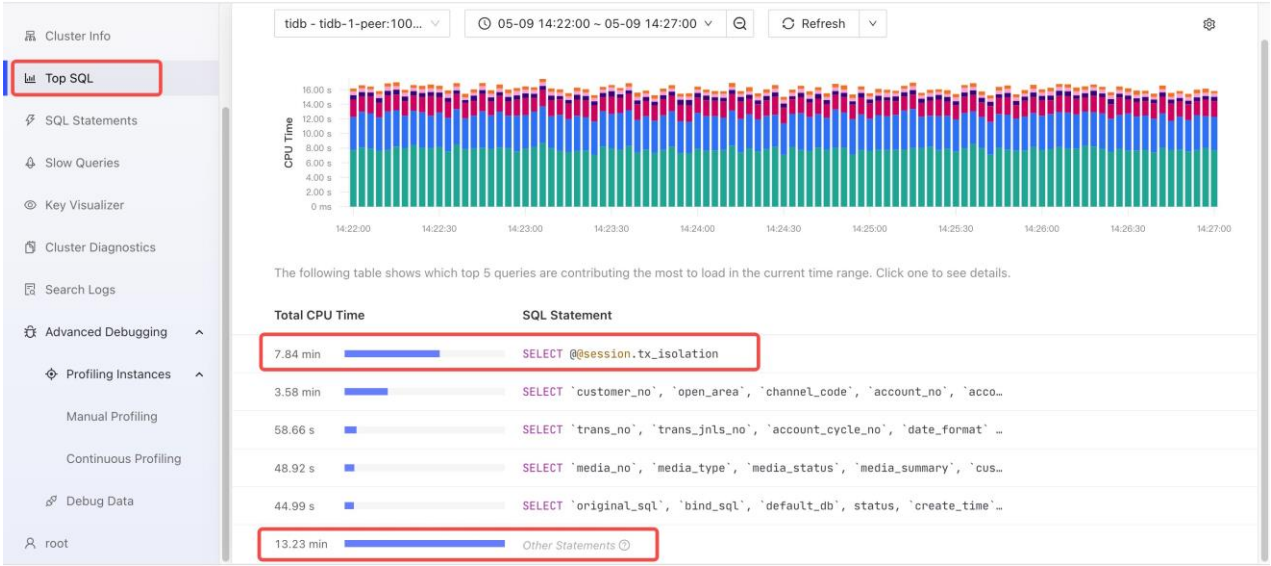


图 100: dashboard-for-query-interface

观察以下TiDB 的火焰图，可以发现，在 SQL 的执行过程中，Compile 和 Optimize 等函数的 CPU 消耗占比明显。因为应用使用了 Query 接口，TiDB 无法使用执行计划缓存，导致每个 SQL 都需要编译生成执行计划。



图 101: flame-graph-for-query-interface

- ExecuteStmt cpu = 38% cpu time = 23.84s
- Compile cpu = 27% cpu time = 17.17s
- Optimize cpu = 26% cpu time = 16.41s

Performance Overview 面板

观察以下 Performance Overview 面板中数据库时间概览和 QPS 的数据：

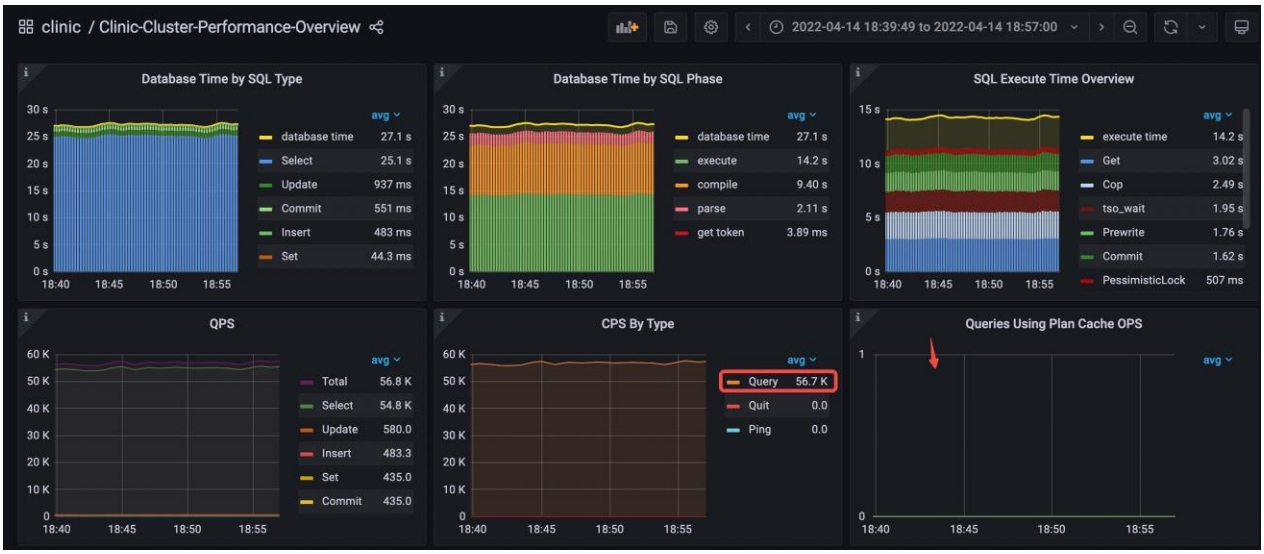


图 102: performance-overview-1-for-query-interface

- Database Time by SQL Type 中 Select 语句耗时最多
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是Get、 Cop 和 tso wait
- CPS By Type 只有query 这一种command
- Queries Using Plan Cache OPS 没有数据，说明无法命中执行计划缓存

- execute 和 compile 的延迟在query duration 中占比最高
- avg QPS = 56.8k

观察集群的资源消耗，TiDB CPU 的平均利用率为 925%，TiKV CPU 的平均利用率为 201%，TiKV IO 平均吞吐为 18.7 MB/s。 TiDB 的资源消耗明显更高。

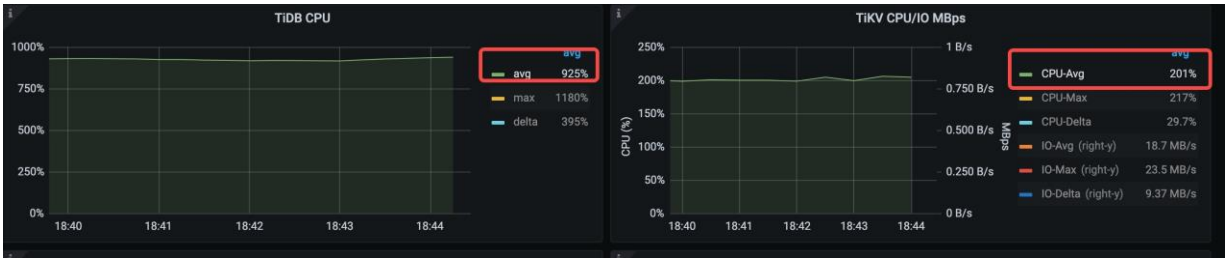


图 103: performance-overview-2-for-query-interface

8.1.3.2.3 分析结论

需要屏蔽这些大量无用的非业务 SQL 语句。

8.1.3.3 场景 2：使用maxPerformance 配置

8.1.3.3.1 应用配置

在场景 1 中的JDBC 连接串的基础上，新增一个参数 `useConfigs=maxPerformance`。这个参数可以用来屏蔽 JDBC 向数据库发送的一些查询设置类的 SQL 语句（例如 `select @@session.transaction_read_only`），完整配置如下：

```
useServerPrepStmts=false&useConfigs=maxPerformance
```

8.1.3.3.2 性能分析

TiDB Dashboard

在 Dashboard 的Top SQL 页面，可以看到原本占比最多的 `SELECT @@session.tx_isolation` 已消失。

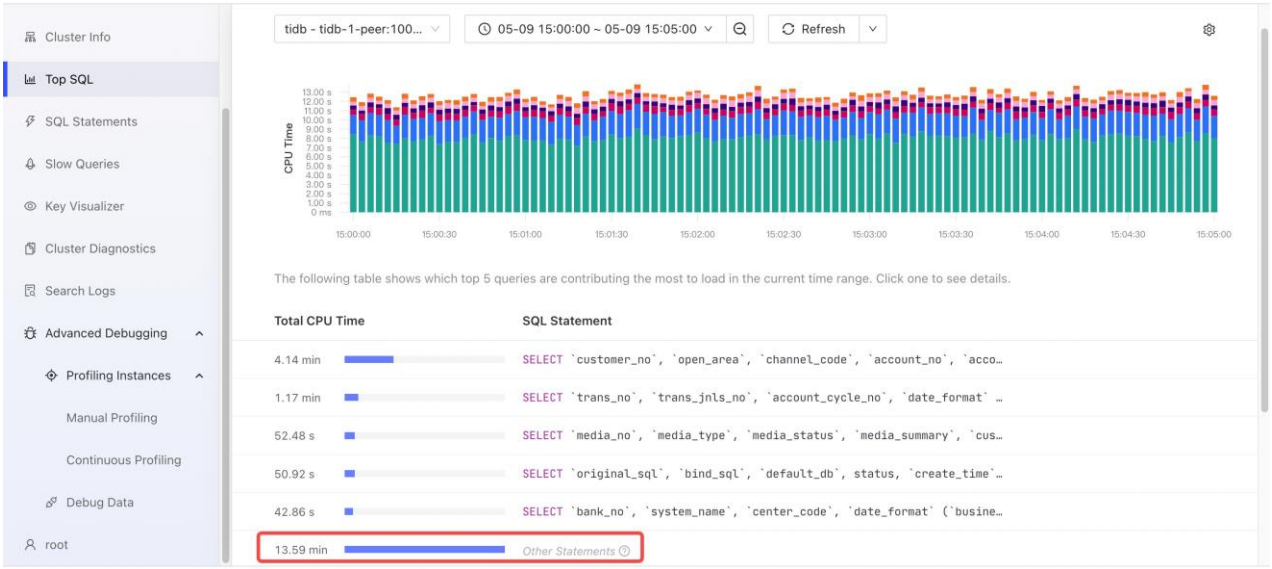


图 104: dashboard-for-maxPerformance

观察以下TiDB 的火焰图，可以发现 SQL 语句执行中 Compile 和 Optimize 等函数 CPU 消耗占比高：



图 105: flame-graph-for-maxPerformance

- ExecuteStmt cpu = 43% cpu time =35.84s
- Compile cpu = 31% cpu time =25.61s
- Optimize cpu = 30% cpu time = 24.74s

Performance Overview 面板

数据库时间概览和 QPS 的数据如下：

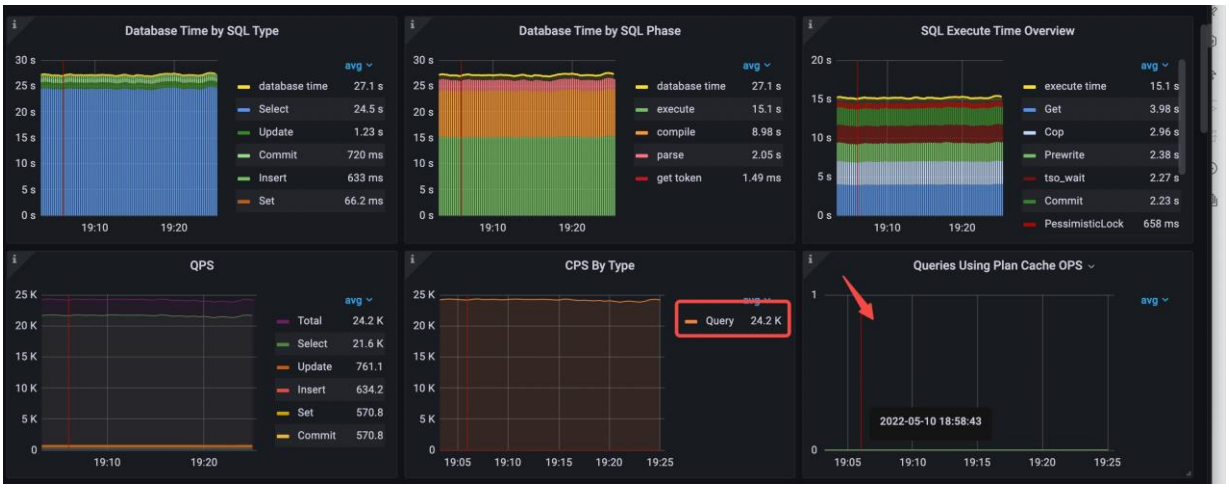


图 106: performance-overview-1-for-maxPerformance

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是Get、 Cop、 Prewrite 和 tso_wait
- execute 和 compile 的延迟在db time 中占比最高
- CPS By Type 只有query 这一种command
- avg QPS = 24.2k (56.3k->24.2k)
- 无法命中plan cache

从场景 1 到场景 2，TiDB CPU 平均利用率从 925% 下降到 874%，TiKV CPU 平均利用率从 201% 上升到 250% 左右。

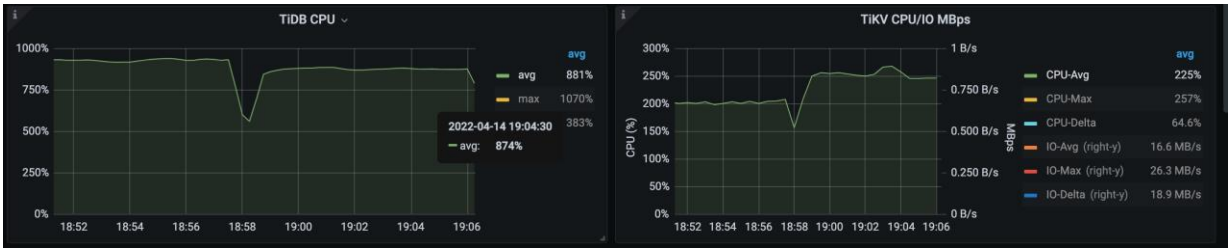


图 107: performance-overview-2-for-maxPerformance

关键延迟指标变化如下：



图 108: performance-overview-3-for-maxPerformance

- avg query duration = 1.12ms (479µs->1.12ms)
- avg parse duration = 84.7µs (37.2µs->84.7µs)
- avg compile duration = 370µs (166µs->370µs)
- avg execution duration = 626µs (251µs->626µs)

8.1.3.3.3 分析结论

和场景 1 相比，场景 2 的 QPS 有明显的下降，平均query duration 和 parse、compile 以及 execute duration 有明显的上升。这是因为场景 1 中类似 `select @@session.transaction_read_only` 这样执行次数多且处理时间快的 SQL 语句拉低了性能平均值，场景 2 屏蔽这类语句后只剩下纯业务 SQL，从而带来了 duration 平均值的上升。

当应用使用 query 接口时，TiDB 无法使用执行计划缓存，编译执行计划消耗高。此时，建议使用 Prepared Statement 预编译接口，利用TiDB 的执行计划缓存来降低compile 带来的TiDB CPU 消耗，降低延迟。

8.1.3.4 场景 3：使用 Prepared Statement 接口，未开启执行计划缓存

8.1.3.4.1 应用配置

应用程序使用以下连接配置，和场景 2 对比，JDBC 的 `useServerPrepStmts` 参数值修改为 `true`，表示启用了预编译语句的接口。

```
useServerPrepStmts=true&useConfigs=maxPerformance"
```

8.1.3.4.2 性能分析

TiDB Dashboard

观察以下TiDB 的火焰图，可以发现启用 Prepared Statement 接口之后，CompileExecutePreparedStmt 和 Optimize 的 CPU 占比依然明显。

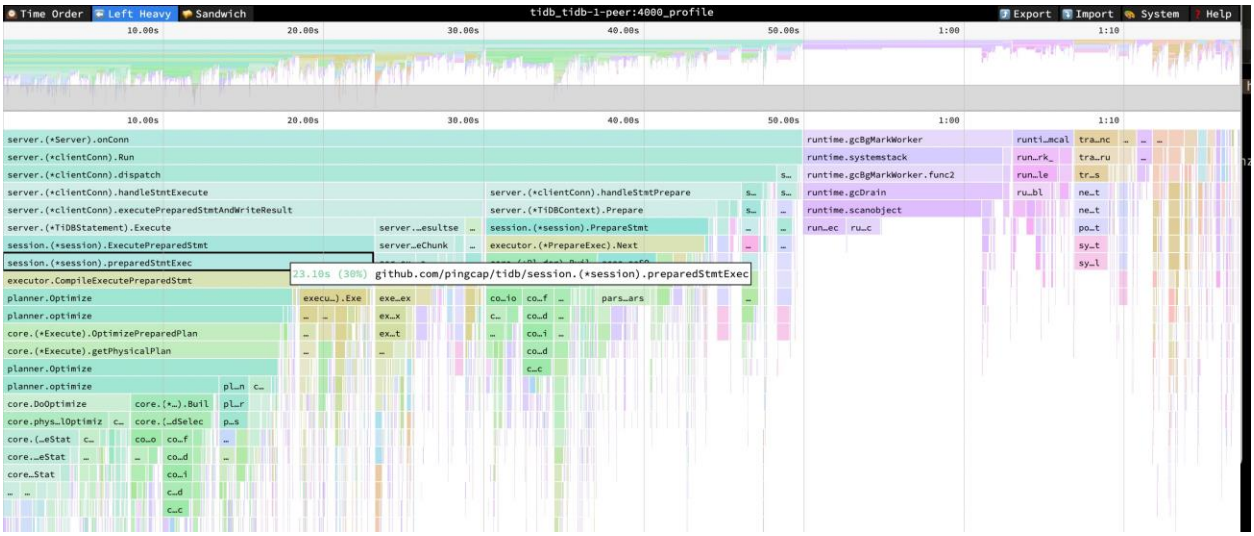


图 109: flame-graph-for-PrepStmts

- ExecutePreparedStmt cpu = 31% cpu time = 23.10s
- preparedStmtExec cpu = 30% cpu time = 22.92s
- CompileExecutePreparedStmt cpu = 24% cpu time = 17.83s
- Optimize cpu = 23% cpu time = 17.29s

Performance Overview 面板

使用 Prepared Statement 接口之后，数据库时间概览和 QPS 的数据如下：

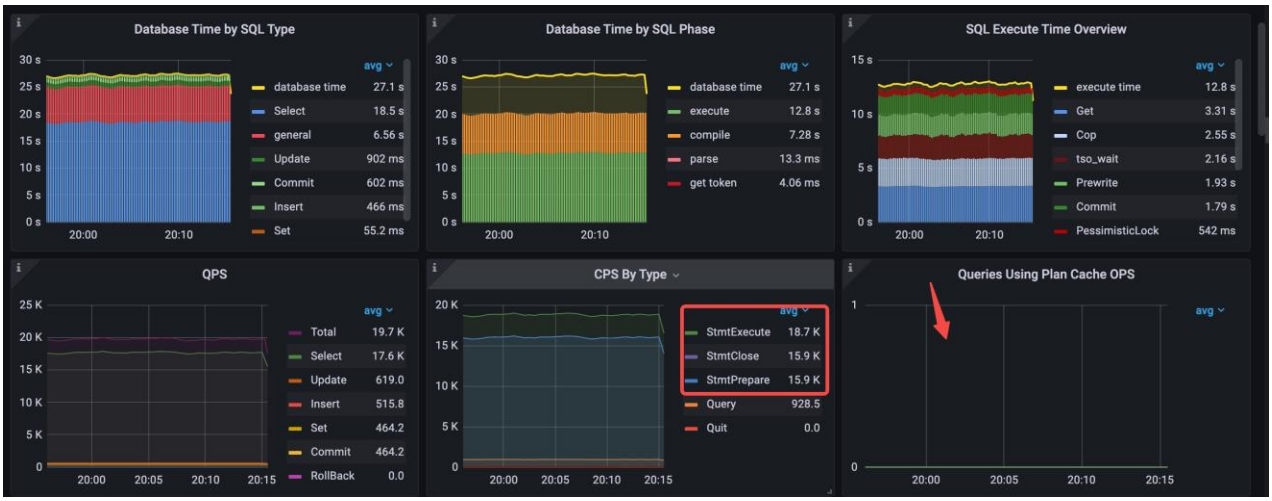


图 110: performance-overview-1-for-PrepStmts

QPS 从 24.4k 下降到 19.7k，从 CPS By Type 面板可以看到应用程序使用了三种 Prepared 命令。Database Time Overview 出现了 general 的语句类型（包含了 StmtPrepare 和 StmtClose 等命令的执行耗时），占比排名第二。这说明，即使使用了 Prepared Statement 接口，执行计划缓存也没有命中，原因在于TiDB 内部处理 StmtClose 命令时，会清理修改语句的执行计划缓存。

- Database Time by SQL Type 中 select 语句耗时最多，其次是 general 语句
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是Get、 Cop、 Prewrite 和 tso_wait
- CPS By Type 变成 3 种 command：StmtPrepare、 StmtExecute、 StmtClose
- avg QPS = 19.7k (24.4k->19.7k)
- 无法命中plan cache

TiDB CPU 平均利用率从 874% 上升到 936%



图 111: performance-overview-1-for-PrepStmts

主要延迟数据如下：

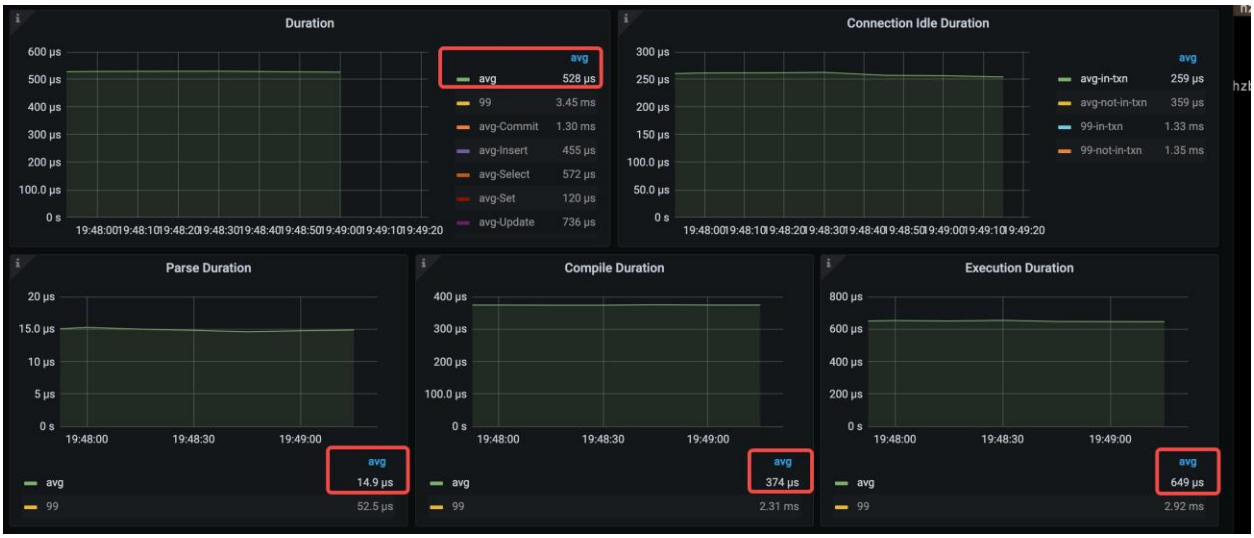


图 112: performance-overview-2-for-PrepStmts

- avg query duration = 528µs (1.12ms->528µs)
- avg parse duration = 14.9µs (84.7µs->14.9µs)
- avg compile duration = 374µs (370µs->374µs)
- avg execution duration = 649µs (626µs->649µs)

8.1.3.4.3 分析结论

和场景 2 不同的是，场景 3 启用了prepare 预编译接口但是仍然无法命中缓存。此外，场景 2 的 CPS By Type 只有query 这一种command 类型，场景 3 多了 3 种command 类型（ StmtPrepare、 StmtExecute、 StmtClose ）。与场景 2 相比，相当于多了两次网络往返的延迟。

- QPS 的降低原因分析：从 CPS By Type 面板可以看到，场景 2 只有query 这一种command 类型，但场景 3 新增了 3 种 command 类型，即 StmtPrepare、 StmtExecute 和 StmtClose。其中，StmtExecute 和 query 为常规类型 command，会被 QPS 统计，而 StmtPrepare 和 StmtClose 为非常规类型command，不会被 QPS 统计，所以 QPS 降低了。非常规类型command 的 StmtPrepare 和 StmtClose 被统计在 general SQL 类型中，因此可以看到 database overview 中多了 general 的时间，且占比在database time 的四分之一以上。
- 平均query duration 明显降低原因分析：场景 3 新增了 StmtPrepare 和 StmtClose 这两种command 类型，TiDB 内部处理时，query duration 也会单独计算，这两类命令处理速度很快，所以平均query duration 明显被拉低。

虽然场景 3 使用了 Prepare 预编译接口但是因为出现了 StmtClose 导致缓存失效，很多应用框架也会在execute 后调用close 方法来防止内存泄漏。从v6.0.0 版本开始，你可以设置全局变量`tidb_ignore_prepared_cache_close_stmt` `′→ =on;`。设置后，即使应用调用了 StmtClose 方法，TiDB 也不会清除缓存的执行计划，使得下一次的 SQL 执行能重用现有的执行计划，避免重复编译执行计划。

8.1.3.5 场景 4：使用 Prepared Statement 接口，开启执行计划缓存

8.1.3.5.1 应用配置

应用配置保持不变。设置以下参数，解决即使应用触发 StmtClose 导致无法命中缓存的问题。

- 设置 TiDB 全局变量 `set global tidb_ignore_prepared_cache_close_stmt=on;`（ TiDB v6.0.0 起正式使用，默认关闭)
- 设置TiDB配置项 `prepared-plan-cache: {enabled: true}` 开启plan cache 功能

8.1.3.5.2 性能分析

TiDB Dashboard

观察TiDB 的 CPU 火焰图，可以看到 CompileExecutePreparedStmt 和 Optimize 没有明显的 CPU 消耗。 Prepare 命令的 CPU 占比 25%，包含了 PlanBuilder 和 parseSQL 等 Prepare 解析相关的函数。

PreparseStmt cpu = 25% cpu time = 12.75s

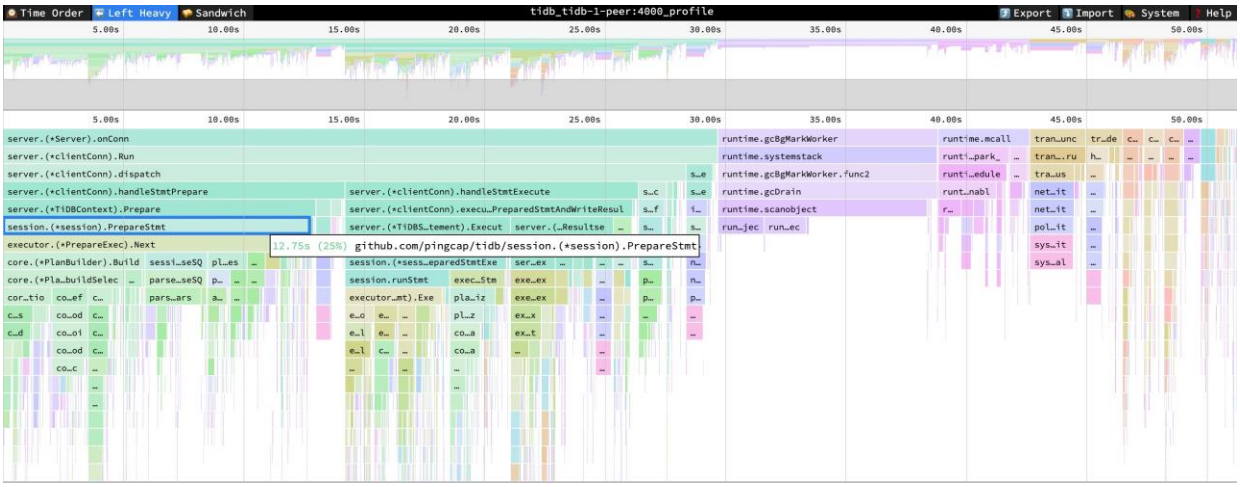


图 113: flame-graph-for-3-commands

Performance Overview 面板

在 Performance Overview 面板种，最显著的变化来自于 Compile 阶段的占比，从场景 3 每秒消耗 8.95 秒降低为 1.18 秒。执行计划缓存的命中次数大致等于 StmtExecute 次数。在 QPS 上升的前提下，每秒 Select 语句消耗的数据库时间降低了，general 类型的语句消耗时间变长。

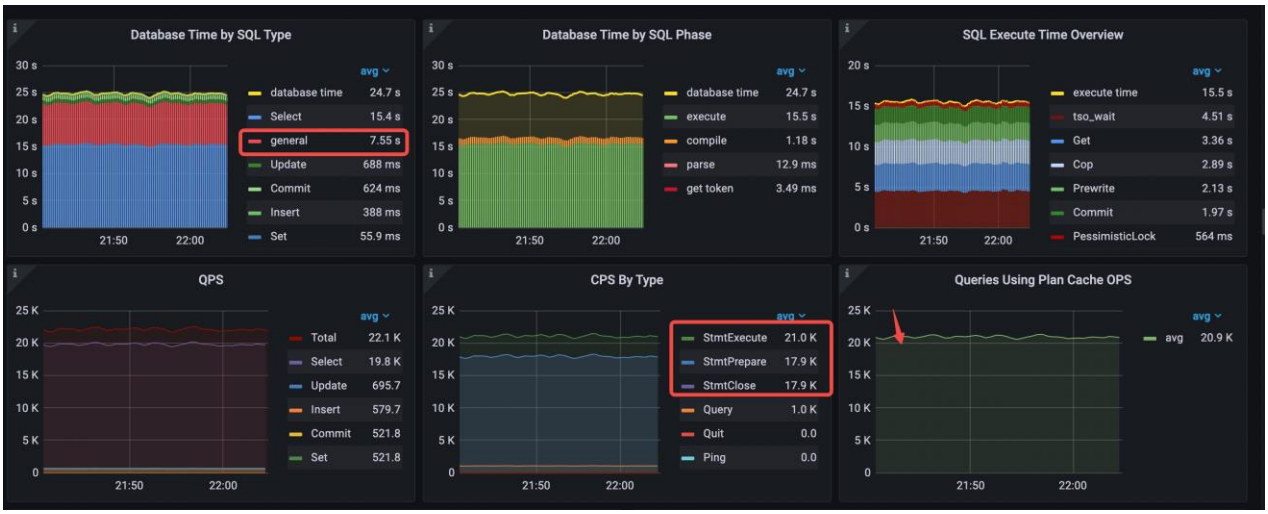


图 114: performance-overview-1-for-3-commands

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 占比最多
- SQL Execute Time Overview 中占比最多的分别是tso wait、 Get 和 Cop
- 命中 plan cache， Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 仍然是 3 种 command
- general time 相比场景 3 变长，因为 QPS 上升了
- avg QPS = 22.1k (19.7k->22.1k)

TiDB CPU 平均利用率从 936% 下降到 827%。

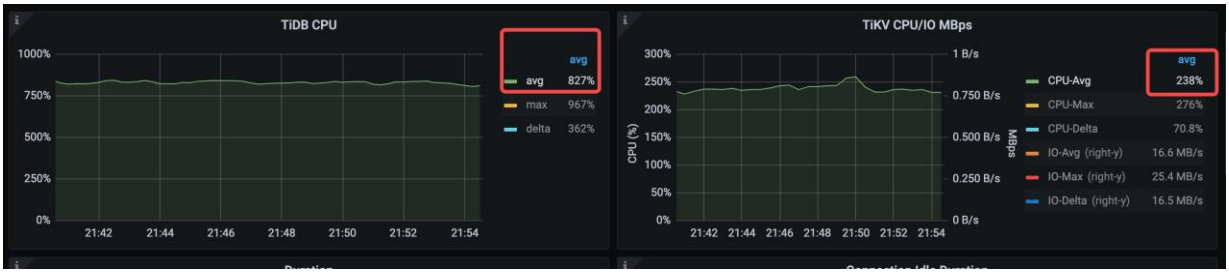


图 115: performance-overview-2-for-3-commands

Compile 平均时间显著下降，从 374 us 下降到 53.3 us，因为 QPS 的上升，平均 execute 时间有所上升。



图 116: performance-overview-3-for-3-commands

- avg query duration = 426µs (528µs->426µs)
- avg parse duration = 12.3µs (14.8µs->12.3µs)
- avg compile duration = 53.3µs (374µs->53.3µs)
- avg execution duration = 699µs (649µs->699us)

8.1.3.5.3 分析结论

和场景 3 相比，场景 4 同样存在 3 种 command 类型，不同的是场景 4 可以命中执行计划缓存，所以大大降低了 compile duration，同时降低了query duration，并且提升了 QPS。

因为 StmtPrepare 和 StmtClose 两种命令消耗的数据库时间明显，并且增加了应用程序每执行一条 SQL 语句需要跟 TiDB 交互的次数。下一个场景将通过JDBC 配置优化掉这两种命令。

8.1.3.6 场景 5：客户端缓存prepared 对象

8.1.3.6.1 应用配置

和场景4相比，新增3个JDBC参数配置cachePrepStmts=true&prepStmtCacheSize=1000&prepStmtCacheSqlLimit
↪ =20480，解释如下：

- cachePrepStmts = true：在客户端缓存 prepared statement 对象，消除 StmtPrepare 和 StmtClose 调用。
- prepStmtCacheSize：需要配置为大于 0 的值
- prepStmtCacheSqlLimit：需要设置为大于 SQL 文本的长度

完整的JDBC参数配置如下：

```
useServerPrepStmts=true&cachePrepStmts=true&prepStmtCacheSize=1000&prepStmtCacheSqlLimit=20480&
    ↪ useConfigs=maxPerformance
```

8.1.3.6.2 性能分析

TiDB Dashboard

观察以下TiDB的火焰图，可以发现 Prepare 命令的高 CPU 消耗未再出现。

- ExecutePreparedStmt cpu = 22% cpu time = 8.4s

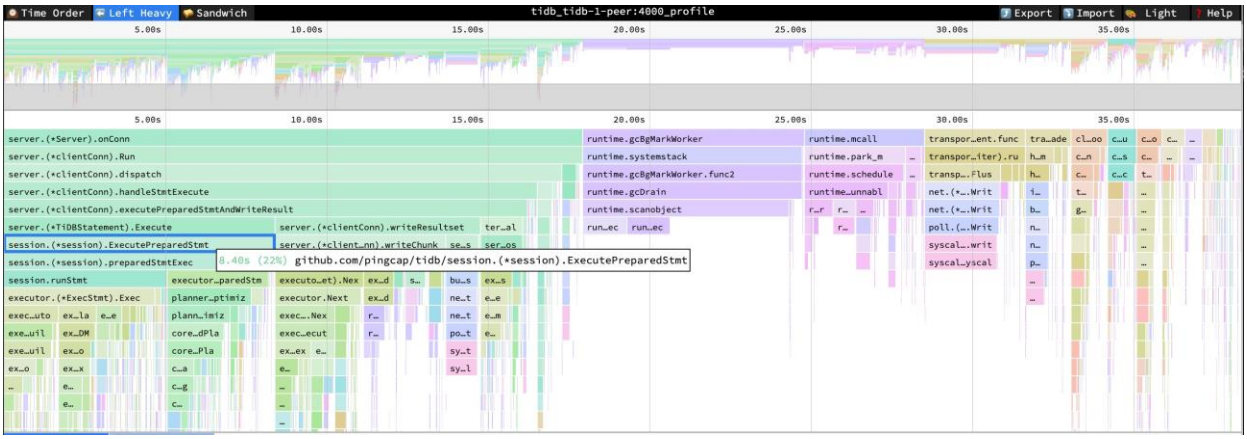


图 117: flame-graph-for-1-command

Performance Overview 面板

在Performance Overview 面板中，最显著的变化是，CPS By Type 面板中三种Stmt command 类型变成了一种，Database Time by SQL Type 面板中的 general 语句类型消失了，QPS 面板中 QPS 上升到了 30.9k。

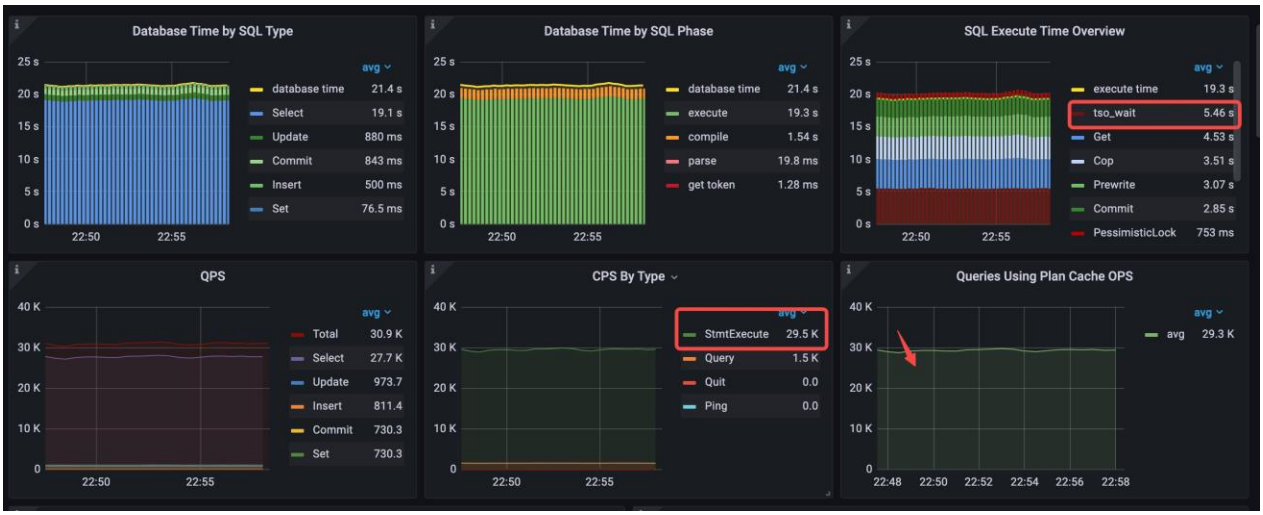


图 118: performance-overview-for-1-command

- Database Time by SQL Type 中 select 语句耗时最多，general 语句类型消失了
- Database Time by SQL Phase 中主要为 execute
- SQL Execute Time Overview 中占比最多的分别是tso wait、 Get 和 Cop
- 命中 plan cache， Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 只有一种command，即 StmtExecute
- avg QPS = 30.9k (22.1k->30.9k)

TiDB CPU 平均利用率从 827% 下降到 577%，随着 QPS 的上升，TiKV CPU 平均利用率上升为 313%。

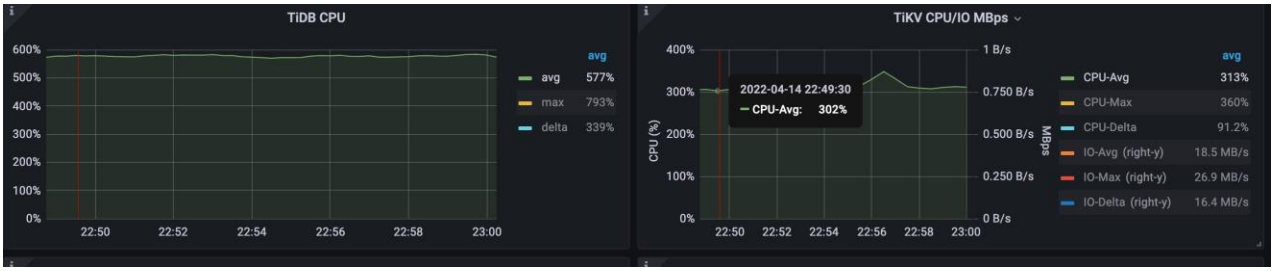


图 119: performance-overview-for-2-command

关键的延迟指标如下：

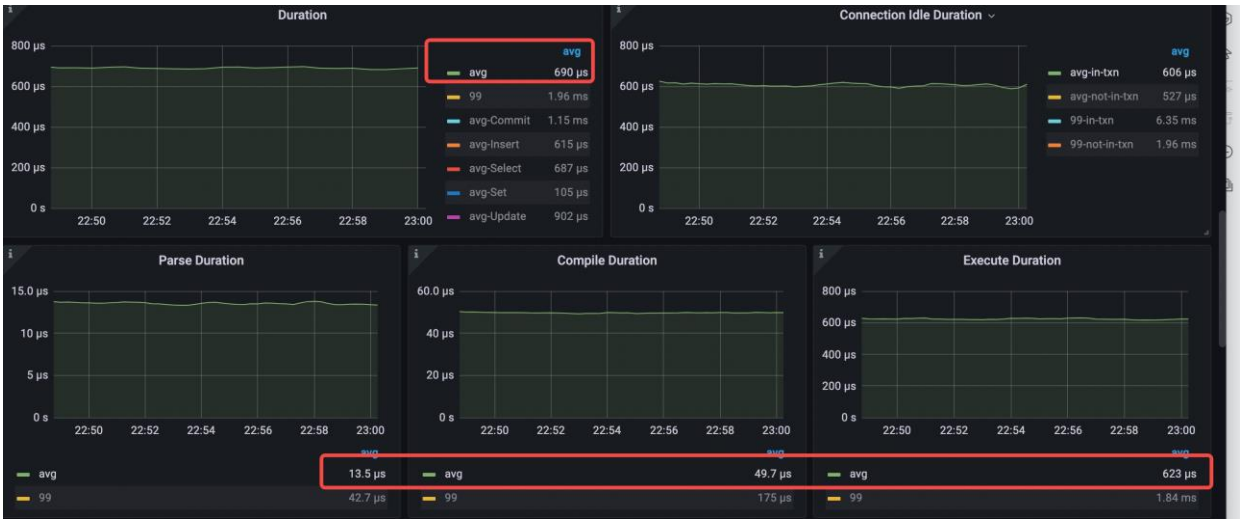


图 120: performance-overview-for-3-command

- avg query duration = 690µs (426->690µs)
- avg parse duration = 13.5µs (12.3µs->13.5µs)
- avg compile duration = 49.7µs (53.3µs->49.7µs)
- avg execution duration = 623µs (699us->623µs)
- avg pd tso wait duration = 196µs (224µs->196µs)
- connection idle duration avg-in-txn = 608µs (250µs->608µs)

8.1.3.6.3 分析结论

- 和场景 4 相比，场景 5 的 CPS By Type 只有 StmtExecute 这一种command，减少了两次的网络往返，系统总体 QPS 上升。
- 在 QPS 上升的情况下，从parse duration、compile duration、execution duration 来看延迟降低了，但query duration 反而上升了。这是因为，StmtPrepare 和 StmtClose 处理的速度非常快，消除这两种command 类型之后，平均的query duration 就会上升。
- Database Time by SQL Phase 中 execute 占比非常高接近于database time，同时 SQL Execute Time Overview 中占比最多的是tso wait，超过四分之一的 execute 时间是在等待tso。
- 每秒的tso wait 总时间为 5.46s。平均tso wait 时间为 196 us，每秒的tso cmd 次数为 28k，非常接近于 QPS 的 30.9k。因为在TiDB 对于 `read committed` 隔离级别的实现中，事务中的每个 SQL 语句都需要都到 PD 请求 tso。

TiDB v6.0 提供了 `rc read`，针对 `read committed` 隔离级别进行了减少 tso cmd 的优化。该功能由全局变量 `set global tidb_rc_read_check_ts=on;`控制。启用此变量后，TiDB 默认行为和 `repeatable-read` 隔离级别一致，只需要从 PD 获取 start-ts 和 commit-ts。事务中的语句先使用 start-ts 从 TiKV 读取数据。如果读到的数据小于 start-ts，则直接返回数据；如果读到大于 start-ts 的数据，则需要丢弃数据，并向 PD 请求TSO 再进行重试。后续语句的 for update ts 使用最新的 PD TSO。

8.1.3.7 场景 6：开启tidb_rc_read_check_ts 变量降低TSO 请求

8.1.3.7.1 应用配置

应用配置不变，和场景 5 不同的是，设置 `set global tidb_rc_read_check_ts=on;`，减少 TSO 请求。

8.1.3.7.2 性能分析

TiDB Dashboard

TiDB 的 CPU 火焰图没有明显变化。

- ExecutePreparedStmt cpu = 22% cpu time = 8.4s



图 121: flame-graph-for-rc-read

Performance Overview 面板

使用 RC read 之后，QPS 从 30.9k 上升到 34.9k，每秒消耗的tso wait 时间从 5.46 s 下降到 456 ms。

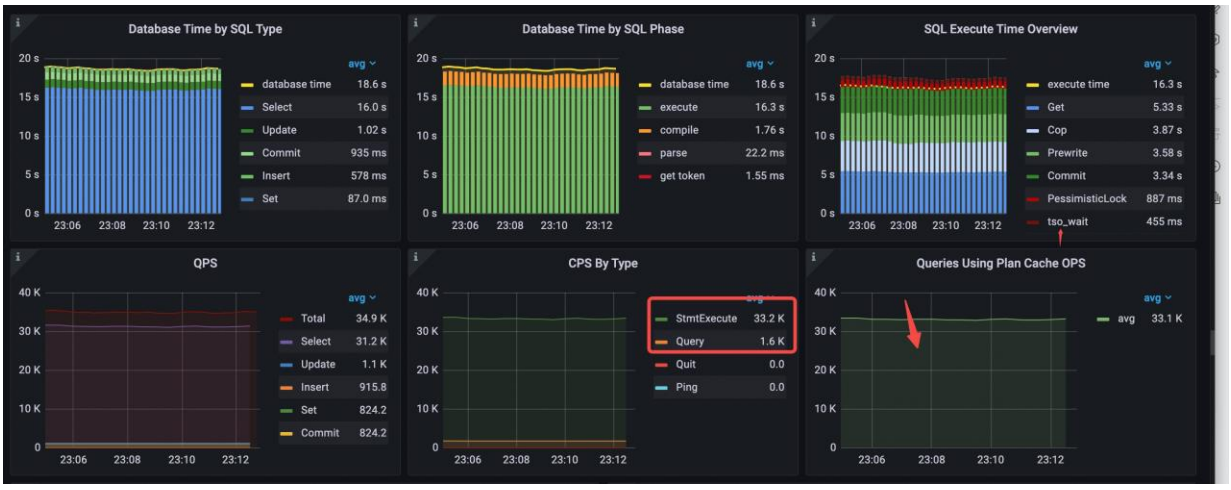


图 122: performance-overview-1-for-rc-read

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 占比最高
- SQL Execute Time Overview 中占比最多的分别是Get、 Cop 和 Prewrite
- 命中plan cache， Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 只有一种command
- avg QPS = 34.9k (30.9k->34.9k)

每秒tso cmd 从 28.3k 下降到 2.7k。

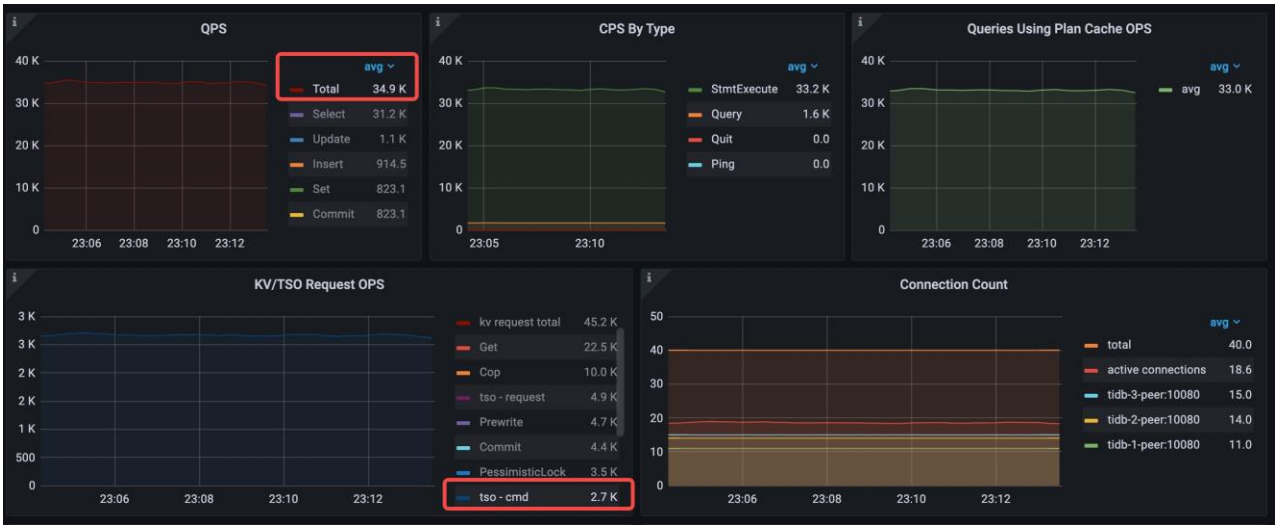


图 123: performance-overview-2-for-rc-read

平均TiDB CPU 上升为 603% (577%->603%)。

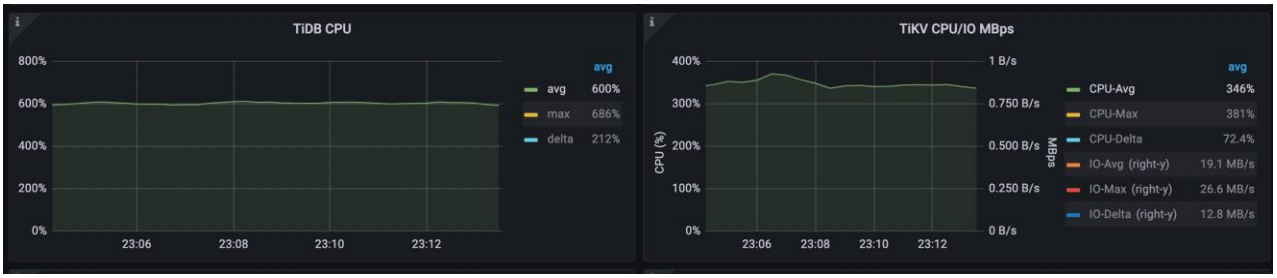


图 124: performance-overview-3-for-rc-read

关键延迟指标如下：

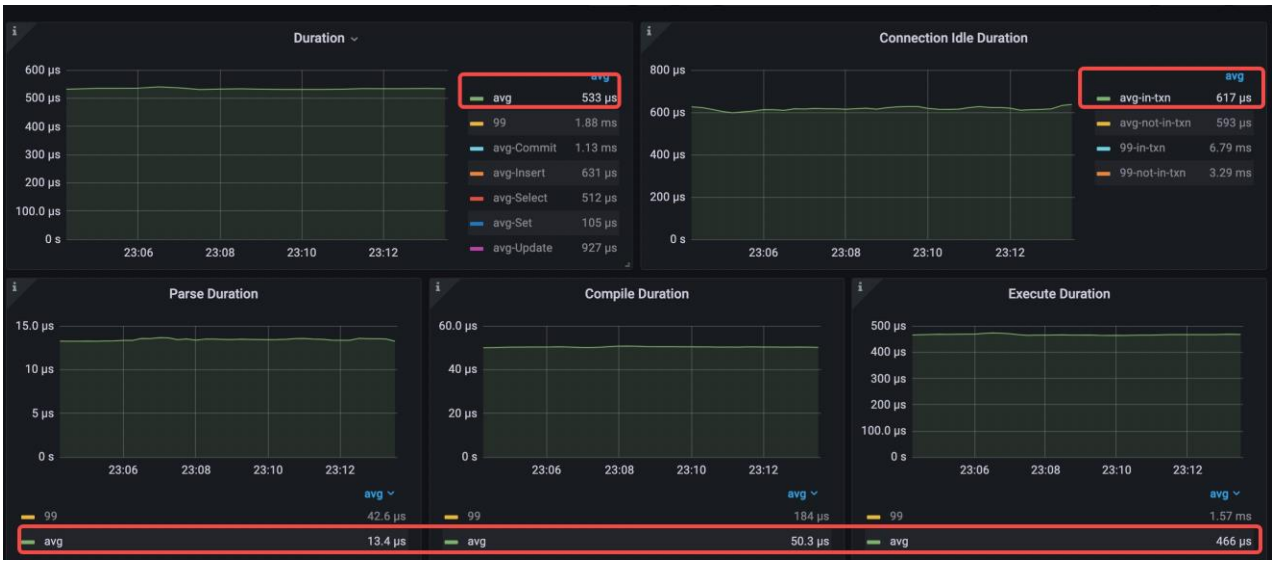


图 125: performance-overview-4-for-rc-read

- avg query duration = 533µs (690µs->533µs)
- avg parse duration = 13.4µs (13.5µs->13.4µs)
- avg compile duration = 50.3µs (49.7µs->50.3µs)
- avg execution duration = 466µs (623µs->466µs)
- avg pd tso wait duration = 171µs (196µs->171µs)

8.1.3.7.3 分析结论

通过 `set global tidb_rc_read_check_ts=on;` 启用RC Read 之后，RC Read 明显降低了tso cmd 次数从而降低了 tso wait 以及平均query duration，并且提升了 QPS。

当前数据库时间和延迟瓶颈都在 execute 阶段，而 execute 阶段占比最高的为Get 和 Cop 读请求。这个负载中，大部分表是只读或者很少修改，可以使用 v6.0.0 的小表缓存功能，使用TiDB 缓存这些小表的数据，降低 KV 读请求的等待时间和资源消耗。

8.1.3.8 场景 7：使用小表缓存

8.1.3.8.1 应用配置

应用配置不变，在场景 6 的基础上设置了对业务的只读表进行缓存 `alter table t1 cache;`。

8.1.3.8.2 性能分析

TiDB Dashboard

TiDB CPU 火焰图没有明显变化。

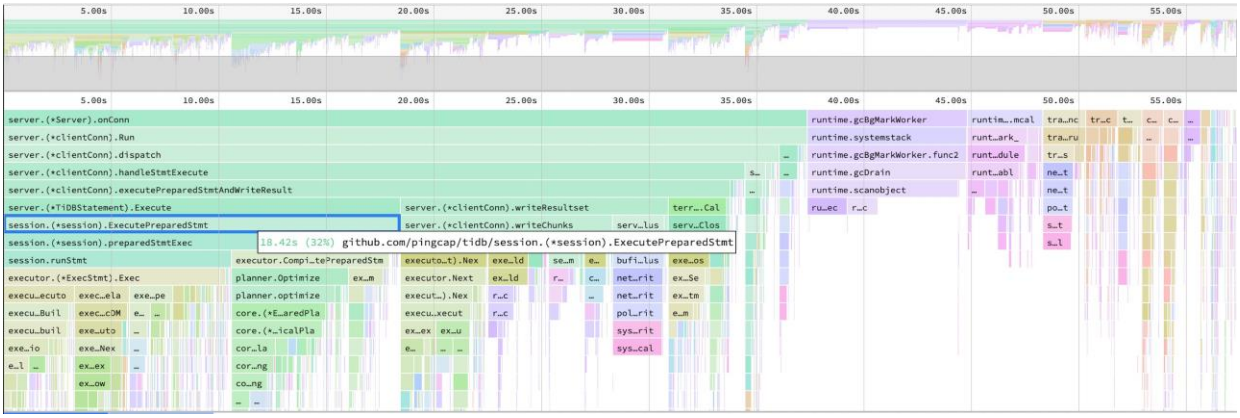


图 126: flame-graph-for-table-cache

Performance Overview 面板

QPS 从 34.9k 上升到 40.9k，execute 时间中占比最高的 KV 请求类型变成了 Prewrite 和 Commit。Get 每秒的时间从 5.33 秒下降到 1.75 秒，Cop 每秒的时间从 3.87 下降到 1.09 秒。

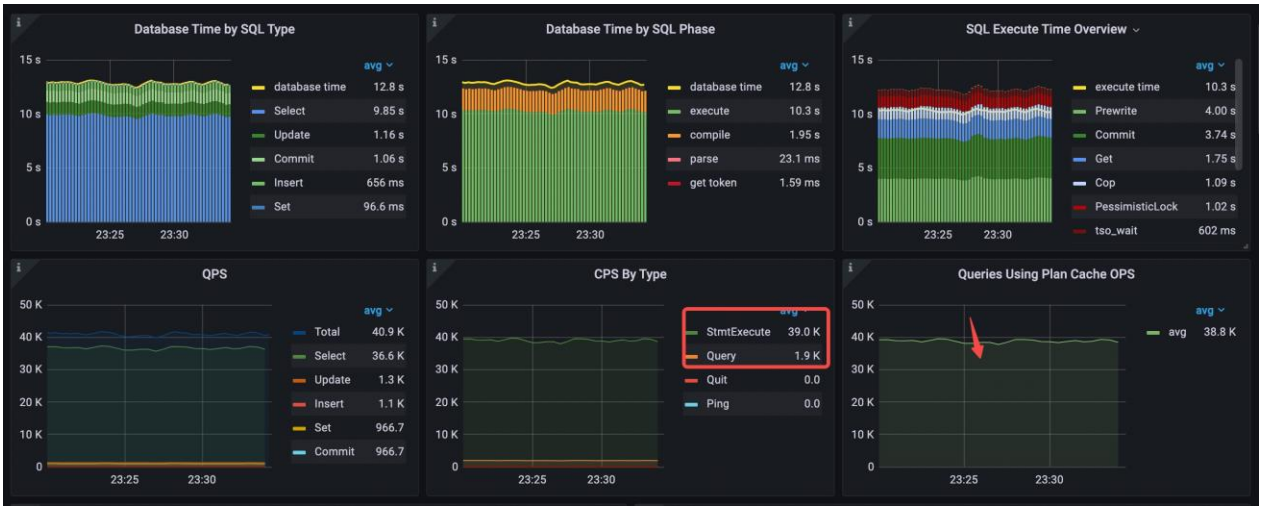


图 127: performance-overview-1-for-table-cache

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是 Prewrite、Commit 和 Get
- 命中 plan cache，Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 只有 1 种 command
- avg QPS = 40.9k (34.9k->40.9k)

TiDB CPU 平均利用率从 603% 下降到 478%，TiKV CPU 平均利用率从 346% 下降到 256%。

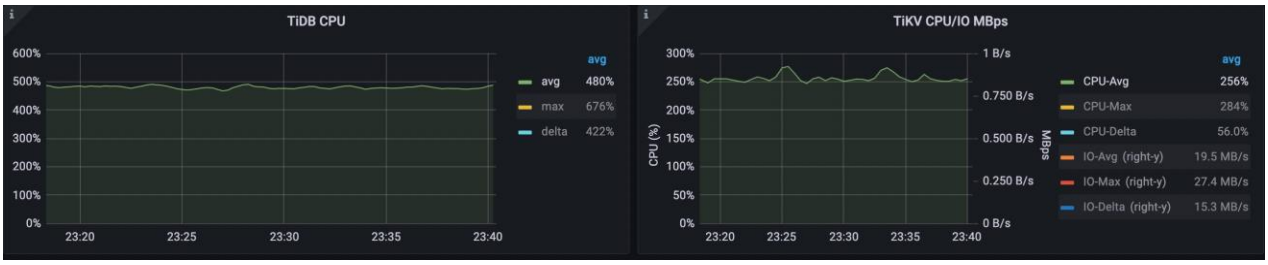


图 128: performance-overview-2-for-table-cache

Query 平均延迟从 533 us 下降到 313 us。 execute 平均延迟从 466 us 下降到 250 us。

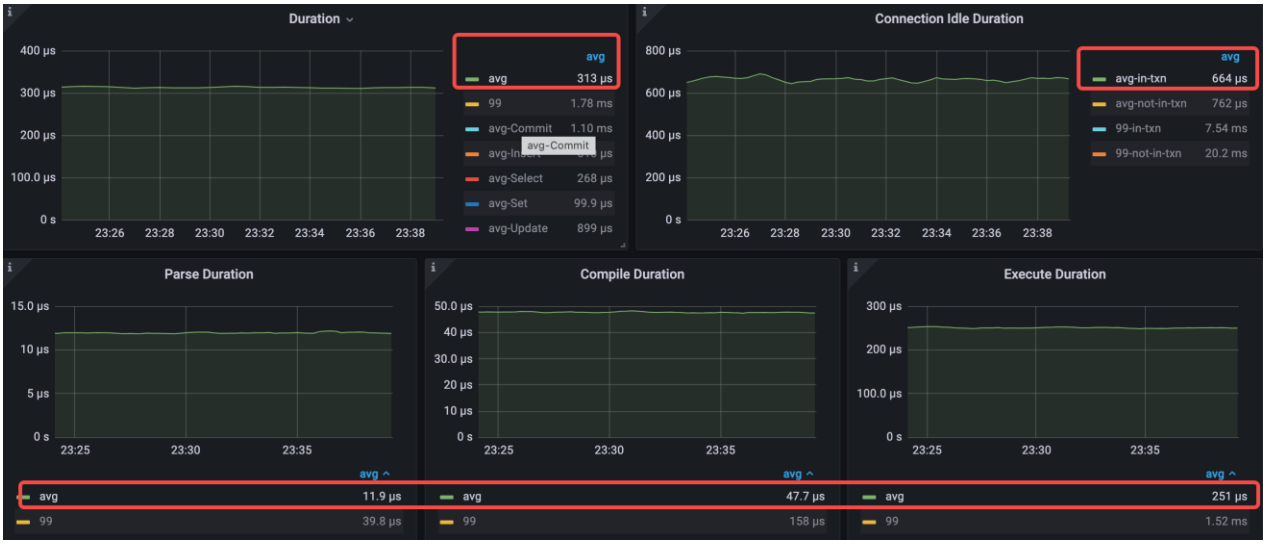


图 129: performance-overview-3-for-table-cache

- avg query duration = 313µs (533µs->313µs)
- avg parse duration = 11.9µs (13.4µs->11.9µs)
- avg compile duration = 47.7µs (50.3µs->47.7µs)
- avg execution duration = 251µs (466µs->251µs)

8.1.3.8.3 分析结论

将所有只读表进行缓存后，可以看到，execute duration 下降非常明显，原因是所有只读表都缓存在TiDB 中，不再需要到TiKV 中查询数据，所以query duration 下降，QPS 上升。

这个是比较乐观的结果，实际的业务中可能只读表的数据量较大无法全部在TiDB 中进行缓存。另一个限制是，当前小表缓存的功能虽然支持写操作，但是写操作默认需要等 3 秒，确保所有TiDB 节点的缓存失效，对于对延迟要求比较高的应用，可能暂时不太友好。

8.1.3.9 总结

以下表格展示了七个不同场景的性能表现：

指标	场景 1	场景 2	场景 3	场景 4	场景 5	场景 6	场景 7	对比场景 5 和场景 2 (%)	对比场景 7 和场景 3(%)
query duration	479µs	1120µs	528µs	426µs	690µs	533µs	313µs	-38%	-51%
QPS	56.3k	24.2k	19.7k	22.1k	30.9k	34.9k	40.9k	+28%	+108%

其中，场景 2 是应用程序使用 Query 接口的常见场景，场景 5 是应用程序使用 Prepared Statement 接口的理想场景。

- 对比场景 2 和场景 5，可以发现通过使用Java 应用开发的最佳实践以及客户端缓存 Prepared Statement 对象，每条 SQL 只需要一次命令和数据库交互，就能命中执行计划缓存，从而使 Query 延迟下降了 38%，QPS 上升 28%，同时，TiDB CPU 平均利用率从 936% 下降到 577%。
- 对比场景 2 和场景 7，可以看到在场景 5 的基础上使用了RC Read、小表缓存等TiDB 最新的优化功能，延迟降低了 51%，QPS 提升了 108%，同时，TiDB CPU 平均利用率从 936% 下降到 478%。

通过对比各场景的性能表现，可以得出以下结论：

- TiDB 的执行计划缓存对于 OLTP 发挥着至关重要的作用。而从 v6.0.0 开始引入的RC Read 和小表缓存功能，在这个负载的深度优化中，也发挥了重要的作用。
- TiDB 兼容 MySQL 协议的不同命令，最佳的性能表现来自于应用程序使用 Prepared Statement 接口，并设置以下JDBC 连接参数：

```
useServerPrepStmts=true&cachePrepStmts=true&prepStmtCacheSize=1000&prepStmtCacheSqlLimit
    ↪ =20480&useConfigs=maxPerformance
```

- 在性能分析和优化过程中， 推荐使用 TiDB Dashboard （例如 Top SQL 功能和持续性能分析功能）和 Performance Overview 面板。
  - Top SQL 功能允许你可视化地监控和探索数据库中各个 SQL 语句在执行过程中的 CPU 开销情况，从而对数据库性能问题进行优化和处理。
  - 持续性能分析功能可以持续地收集TiDB、TiKV、PD 各个实例的性能数据。应用程序使用不同接口跟 TiDB 交互时，TiDB 的 CPU 消耗有着巨大的差距。
  - Performance Overview 面板提供了数据库时间的概览和 SQL 执行时间分解信息。借助这个面板，你可以进行基于数据库时间的性能分析和诊断，确定整个系统的性能瓶颈是否处于TiDB 中。如果瓶颈在 TiDB 中，你可以通过数据库时间和延迟的分解，以及集群关键指标和资源使用情况，确认TiDB 内部的性能瓶颈，并进行针对性的优化。

综合使用以上几个功能，你可以针对现实中的应用进行高效的性能分析和优化。

8.1.4 TiFlash 性能分析和优化方法

本文介绍TiFlash 资源使用率和关键的性能指标。你可以通过 Performance Overview 面板中的TiFlash 面板，来监控和评估TiFlash 集群的性能。

8.1.4.1 TiFlash 集群资源利用率

通过以下三个指标，你可以快速判断TiFlash 集群的资源使用率：

- CPU：每个 TiFlash 实例的 CPU 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

示例：CH-benCHmark 负载资源使用率

该 TiFlash 集群包含两个节点，每个节点配置均为 16 核、48G 内存。当 CH-benCHmark 负载运行时，CPU 利用率最高可达到 1500%，内存占用最大可达 20 GB，IO 利用率达到 91%。这表明TiFlash 节点资源接近饱和状态。

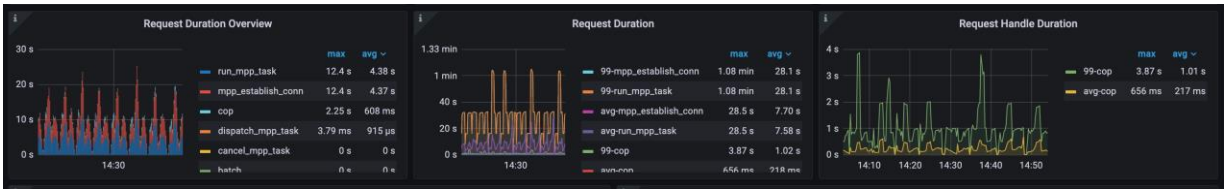


图 130: CH-TiFlash-MPP

8.1.4.2 TiFlash 关键性能指标

8.1.4.2.1 吞吐指标

通过以下指标，你可以了解TiFlash 的吞吐情况：

- MPP Query count：每个TiFlash 实例 MPP 查询数量的瞬时值，表示当前TiFlash 实例需要处理的 MPP 查询数量（包括正在处理的以及还没被调度到的）。
- Request QPS：所有 TiFlash 实例收到的coprocessor 请求数量。
  - `run_mpp_task`、`dispatch_mpp_task` 和 `mpp_establish_conn` 为 MPP 请求。
  - `batch`：batch 请求数量。
  - `cop`：直接通过 coprocessor 接口发送的coprocessor 请求数量。
  - `cop_execution`：正在执行的 coprocessor 请求数量。
  - `remote_read`、`remote_read_constructed` 和 `remote_read_sent` 为 remote read 相关指标，remote read 增多一般意味着系统出现了问题。
- 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，`join` 为关联算子，`exchange_sender` 和 `exchange_receiver` 为数据发送和接收算子。

8.1.4.2.2 延迟指标

通过以下指标，你可以了解TiFlash 的延迟处理情况：

- Request Duration Overview：每秒所有 TiFlash 实例处理所有请求类型的总时长堆叠图。

- 如果请求类型为 `run_mpp_task`、`dispatch_mpp_task` 或 `mpp_establish_conn`，说明 SQL 语句的执行已经部分或者完全下推到TiFlash 上进行，通常包含join 和数据分发的操作，这是TiFlash 最常见的服务类型。
- 如果请求类型为 `cop`，说明整个语句并没有完全下推到 TiFlash，通常TiDB 会将全表扫描算子下推到 TiFlash 上进行数据访问和过滤。在堆叠图中，如果 `cop` 占据主导地位，需要仔细权衡是否合理。
  * 如果 SQL 访问的数据量很大，优化器可能根据成本模型估算TiFlash 全表扫描的成本更低。
  * 如果表结构缺少合适的索引，即使访问的数据量很少，优化器也只能将查询下推到TiFlash 进行全表扫描。在这种情况下，创建合适的索引，通过TiKV 访问数据更加高效。

- Request Duration：所有 TiFlash 实例每种 MPP 和coprocessor 请求类型的总处理时间，包含平均和 P99 处理延迟。
- Request Handle Duration：指 `cop` 和 `batch cop` 从开始执行到执行结束的时间，不包括等待时间，只包含 `cop` 和 `batch cop` 两种类型，包含平均和 P99 延迟。

示例 1：TiFlash MPP 请求处理时间概览

如下图所示，在此负载中，`run_mpp_task` 和 `mpp_establish_conn` 请求的处理时间占比最高，表明大部分请求都是完全下推到TiFlash 上执行的 MPP 任务。

而 `cop` 请求处理时间占比较小，说明存在一部分请求是通过coprocessor 下推到TiFlash 上进行数据访问和过滤的。

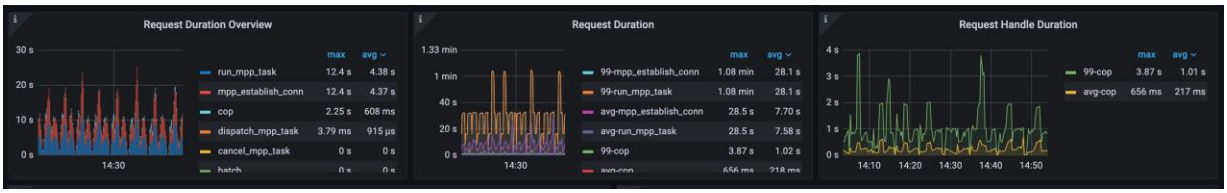


图 131: CH-TiFlash-MPP

示例 2：TiFlash cop 请求处理时间占比高

如下图所示，在此负载中，cop 请求的处理时间占比最高，可以通过查看 SQL 执行计划来确认 cop 请求产生的原因。

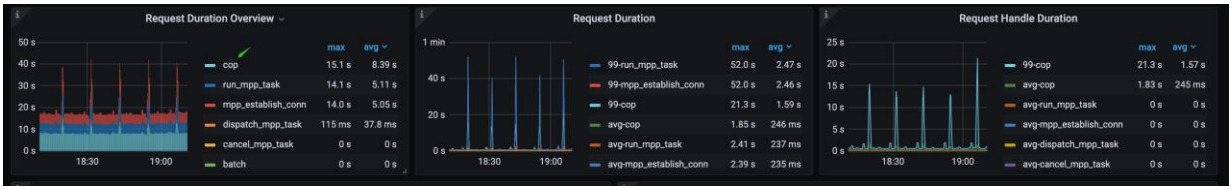


图 132: Cop

8.1.4.2.3 Raft 相关指标

通过以下指标，你可以了解TiFlash 的 Raft 同步情况：

- Raft Wait Index Duration：所有 TiFlash 实例等待本地Region index >= read_index 所花费的时间，即进行 wait_index 操作的延迟。如果Wait Index 延迟过高，说明TiKV 和 TiFlash 之间数据同步存在明显的延迟，可能的原因包括：
  - TiKV 资源过载
  - TiFlash 资源过载，特别是IO 资源
  - TiKV 和TiFlash 之间存在网络瓶颈
- Raft Batch Read Index Duration：所有 TiFlash 实例Read Index 的延迟。如果该指标过高，说明TiFlash 和 TiKV 之间的交互速度较慢，可能的原因包括：
  - TiFlash 资源过载
  - TiKV 资源过载
  - TiFlash 和TiKV 之间存在网络瓶颈

8.1.4.2.4 IO 流量指标

通过以下指标，你可以了解TiFlash 的IO 流量情况：

- Write Throughput By Instance：每个TiFlash 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。
  - File Descriptor：TiFlash 所使用的 DeltaTree 存储引擎的稳定层。
  - Page：指 Pagestore，TiFlash 所使用的 DeltaTree 存储引擎的 Delta 变更层。
- Read flow：所有 TiFlash 实例磁盘读操作的流量。
  - File Descriptor：TiFlash 所使用的 DeltaTree 存储引擎的稳定层。
  - Page：指 Pagestore，TiFlash 所使用的 DeltaTree 存储引擎的 Delta 变更层。

你可以通过 `(Read flow + Write flow)÷` 总的 `Write Throughput By Instance` 计算出整个TiFlash 集群的写放大倍数。

示例 1：CH-benCHmark 负载 OP 环境Raft 和 IO 指标

如下图所示，该TiFlash 集群的Raft Wait Index 和 Raft Batch Read Index 99 分位数较高，分别为 3.24 秒和 753 毫秒。这是因为该集群的TiFlash 负载较高，数据同步存在延迟。

该集群包含两个TiFlash 节点，每秒TiKV 同步到TiFlash 的增量数据约为 28 MB。稳定层 (File Descriptor) 的文件描述符最大写流量为 939 MB/s，最大读流量为 1.1 GiB/s，而 Delta 层 (Page) 最大写流量为 74 MB/s，最大读流量为 111 MB/s。该环境中的 TiFlash 使用独立的 NVME 盘，具有较强的IO 吞吐能力。



图 133: CH-2TiFlash-OP

示例 2： CH-benCHmark 负载公有云环境Raft 和 IO 指标

如下图所示，Raft Wait Index 等待时间 99 分位数最高为 438 毫秒，Raft Batch Read Index 等待时间 99 分位数最高为 125 毫秒。该集群只有一个 TiFlash 节点，每秒TiKV 同步到TiFlash 的增量数据约为 5 MB。稳定层 (File Descriptor) 的最大写入流量为 78 MB/s，最大读取流量为 221 MB/s，Delta 层(Page) 最大写入流量为 8 MB/s，最大读取流量为 18 MB/s。这个环境中的 TiFlash 使用的是AWS EBS 云盘，其IO 吞吐能力相对较弱。



图 134: CH-TiFlash-MPP

8.1.5 TiCDC 性能分析和优化方法

本文介绍TiCDC 资源使用率和关键的性能指标。你可以通过 Performance Overview 面板中的CDC 面板来监控和评估 TiCDC 同步数据的性能。

8.1.5.1 TiCDC 集群的资源利用率

通过以下三个指标，你可以快速判断TiCDC 集群的资源使用率：

- 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

8.1.5.2 TiCDC 数据同步关键指标

8.1.5.2.1 TiCDC 整体指标

通过以下指标，你可以了解TiCDC 数据同步的整体情况：

- Changefeed checkpoint lag：同步任务上下游数据的进度差，以时间单位秒计算。

  如果TiCDC 消费数据的速度和写入下游的速度能跟上上游的数据变更，该指标将保持在较小的延迟范围内，通常是 10 秒以内。如果TiCDC 消费数据的速度和写入下游的速度跟不上上游的数据变更，则该指标将持续增长。

  该指标增长（即TiCDC checkpoint lag 增长）的常见原因如下：

  - 系统资源不足：如果TiCDC 系统中的 CPU、内存或磁盘空间不足，可能会导致数据处理速度过慢，从而导致TiCDC Changefeed checkpoint 过长。
  - 网络问题：如果TiCDC 系统中存在网络中断、延迟或带宽不足的问题，可能会影响数据的传输速度，从而导致TiCDC Changefeed checkpoint 过长。
  - 上游 QPS 过高：如果 TiCDC 系统需要处理的数据量过大，可能会导致数据处理超时，从而导致 TiCDC Changefeed checkpoint 增长，通常一个TiCDC 节点处理的 QPS 上限为 60K 左右。
  - 数据库问题：
    * 上游 TiKV 集群 `min resolved ts` 和最新的 PD TSO 差距过大，通常是因为上游写入负载过大，TiKV 无法及时推进resolved ts 造成。
    * 下游数据库写入延迟高导致TiCDC 无法及时将数据同步到下游。

- 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。如果 TiCDC Changefeed resolved ts lag 值很高，可能意味着TiCDC 系统的 Puller 或者 Sorter 模块数据处理能力不足，或者可能存在网络延迟或磁盘读写速度慢的问题。在这种情况下，需要采取适当的措施，例如增加TiCDC 实例数量或优化网络配置，以确保TiCDC 系统的高效和稳定运行。
- The status of changefeeds：Changefeed 各状态的解释，请参考Changefeed 状态流转。

示例 1：单个TiCDC 节点上游 QPS 过高导致checkpoint lag 过高

如下图所示，因为上游 QPS 过高，该集群中只有单个TiCDC 节点，TiCDC 节点处于过载状态，CPU 使用率较高，Changefeed checkpoint lag 和 Changefeed resolved ts lag 持续增长。Changefeeds 的状态间歇性地从 0 变为 1，意味着 changefeed 不断出错。你可尝试通过增加资源解决该问题：

- 添加TiCDC 节点：将TiCDC 集群扩展到多个节点，以增加处理能力。
- 优化TiCDC 节点的资源：提高TiCDC 节点的 CPU 和内存配置，以改善性能。



图 135: TiCDC overview

8.1.5.2.2 数据流吞吐指标和下游延迟信息

通过以下指标，你可以了解数据流的吞吐和下游延迟信息：

- 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

示例 2：下游数据库写入速度对TiCDC 数据同步性能的影响

如下图所示，该环境上下游都为TiDB 集群。通过 `TiCDC Puller output events/s` 可以确认上游数据库的 QPS 值。通过 `Transaction Sink Full Flush Duration` 可以确认，第一段负载的下游数据库平均写入延迟高，第二段负载的下游平均写入延迟低。

- 在第一段负载期间，由于下游TiDB 集群写入数据缓慢，导致TiCDC 消费数据的速度跟不上上游的 QPS，引起 Changefeed checkpoint lag 不断增长。然而，Changefeed resolved ts lag 仍然在 300 毫秒以内，说明同步延迟和吞吐瓶颈不在puller 和 sorter 模块中，而在下游的 sink 模块。
- 在第二段负载期间，因为下游TiDB 集群的写入速度快，TiCDC 同步数据的速度完全追上了上游的速度，因此 Changefeed checkpoint lag 和 Changefeed resolved ts lag 保持在 500 毫秒以内，此时TiCDC 的同步速度较为理想。



图 136: TiCDC overview



图 137: data flow and txn latency

8.1.6 延迟的拆解分析

本文将TiDB 中 SQL 语句的延迟拆解成各项监控指标，并从用户角度对指标进行分析，包括：

- 通用 SQL 层
- 读请求
- 写请求
- 批量请求
- TiKV 快照
- 异步写入

这些分析可以让你深入了解TiDB 在执行 SQL 查询时的耗时情况，有助于诊断TiDB 关键运行路径的问题。除了延迟的指标拆解之外，诊断场景小节介绍了如何在真实场景中分析延迟。

建议在阅读本文前，先阅读TiDB 性能分析和优化方法。需要注意的是，在将延迟拆解成监控指标时，延迟的耗时数值采用的是均值，而非某几个特定慢查询对应的数值。许多指标都会以直方图的形式展示，以更好地展现耗时或者延迟的分布情况。你需要按如下公式使用总和(sum) 及数量(count) 来计算均值(avg)。

```
avg = ${metric_name}_sum / ${metric_name}_count
```

本文介绍的监控指标可以从TiDB 的 Prometheus 管理界面中查询到。

8.1.6.1 通用 SQL 层

通用 SQL 层带来的延迟存在于TiDB 的最顶部，所有 SQL 查询都具有这一部分的延迟。下面是通用 SQL 层操作的时间消耗图：

```
Diagram(
    NonTerminal("Token wait duration"),
    Choice(
        0,
        Comment("Prepared statement"),
        NonTerminal("Parse duration"),
    ),
    OneOrMore(
        Sequence(
        Choice(
            0,
            NonTerminal("Optimize prepared plan duration"),
            Sequence(
            Comment("Plan cache miss"),
            NonTerminal("Compile duration"),
            ),
        ),
        NonTerminal("TSO wait duration"),
        NonTerminal("Execution duration"),
        ),
        Comment("Retry"),
    ),
)
```

通用 SQL 层的延迟可以使用 `e2e duration` 指标观察。它的计算方式是：

```
e2e duration =
    tidb_server_get_token_duration_seconds +
    tidb_session_parse_duration_seconds +
    tidb_session_compile_duration_seconds +
    tidb_session_execute_duration_seconds{type="general"}
```

- `tidb_server_get_token_duration_seconds` 代表令牌(Token)等待耗时，它通常小于 1 微秒，因而足以被忽略。

- `tidb_session_parse_duration_seconds` 代表把 SQL 查询解析成抽象语法树(AST, Abstract Syntax Tree) 的耗时。要跳过这部分的耗时，可以使用PREPARE/EXECUTE 语句。
- `tidb_session_compile_duration_seconds` 代表把抽象语法树编译成执行计划的耗时。要跳过这部分的耗时，可以使用执行计划缓存。
- `tidb_session_execute_duration_seconds{type="general"}` 代表执行各种不同用户查询的耗时。这部分的耗时需要进行细粒度的拆解，以用来分析性能问题或瓶颈。

通常来说，OLTP (Online Transactional Processing) 工作负载可以分为读请求和写请求两类。下面的小节会分别对读请求和写请求进行介绍。这两类请求共享了一些关键代码，但执行方式是不一样的。

8.1.6.2 读请求

读请求只有一种处理形式。

8.1.6.2.1 点查(Point Get)

下面是点查操作的时间消耗图：

```
Diagram(
    Choice(
        0,
        NonTerminal("Resolve TSO"),
        Comment("Read by clustered PK in auto-commit-txn mode or snapshot read"),
    ),
    Choice(
        0,
        NonTerminal("Read handle by index key"),
        Comment("Read by clustered PK, encode handle by key"),
    ),
    NonTerminal("Read value by handle"),
)
```

在点查过程中，`tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算：

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
    read handle duration +
    read value duration
```

`pd_client_cmd_handle_cmds_duration_seconds{type="wait"}` 代表从 PD 中读取TSO 的耗时。在auto-commit 事务模式下从聚簇索引主键或者快照中读取时，该数值为 0。

`read handle duration` 和 `read value duration` 使用如下方式计算：

```
read handle duration = read value duration =
    tidb_tikvclient_txn_cmd_duration_seconds{type="get"} =
    send request duration =
    tidb_tikvclient_request_seconds{type="Get"} =
```

```
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_get"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

`tidb_tikvclient_request_seconds{type="Get"}` 代表通过 gRPC 发往 TiKV 的批量 get 请求耗时。关于 `tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和 `tidb_tikvclient_rpc_net_latency_seconds{store="?"}` 等批量请求客户端的耗时计算方式，请参考批量请求小节。

`tikv_grpc_msg_duration_seconds{type="kv_get"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_get"} =
    tikv_storage_engine_async_request_duration_seconds{type="snapshot"} +
    tikv_engine_seek_micro_seconds{type="seek_average"} +
    read value duration +
    read value duration(non-short value)
```

此时，请求已经到达TiKV。TiKV 在处理 get 请求时，会进行一次 seek 和一到两次read 操作。其中，短数据的键和值被编码在一个 write CF 中，因而只需要进行一次read 操作。TiKV 在处理 get 请求前会先获取一个快照。关于 TiKV 快照耗时的计算方式，请参考TiKV 快照小节。

`read value duration(from disk)` 使用如下方式计算：

```
read value duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="get/batch_get_command"})) /
        sum(rate(tikv_storage_rocksdb_perf{metric="block_read_count",req="get/
        batch_get_command"}))
```

TiKV 采用 RocksDB 作为存储引擎。如果 block cache 中找不到要求的值，TiKV 需要从磁盘中读取。对于 `tikv_storage_rocksdb_perf`，get 请求可以是 get 或者 batch_get_command。

8.1.6.2.2 批量点查(Batch Point Get)

下面是批量点查操作的时间消耗图：

```
Diagram(  NonTerminal("Resolv
  e TSO"), Choice(
    0,
    NonTerminal("Read all handles by index keys"),
    Comment("Read by clustered PK, encode handle by keys"),
  ),
  NonTerminal("Read values by handles"),
)
```

在进行批量点查时，`tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算：

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
```

```
    read handles duration +
    read values duration
```

批量点查的过程几乎与点查一致。不同的是，批量点查会同时得到多个值。

`read handles duration` 和 `read values duration` 使用如下方式计算：

```
read handles duration = read values duration =
    tidb_tikvclient_txn_cmd_duration_seconds{type="batch_get"} =
    send request duration =
    tidb_tikvclient_request_seconds{type="BatchGet"} =
    tidb_tikvclient_batch_wait_duration(transaction) +
    tidb_tikvclient_batch_send_latency(transaction) +
    tikv_grpc_msg_duration_seconds{type="kv_batch_get"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}(transaction)
```

关于`tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和`tidb_tikvclient_rpc_net_latency_seco`
`'→ {store="?"}` 等批量请求客户端耗时的计算方式，请参考批量请求小节。

耗时 `tikv_grpc_msg_duration_seconds{type="kv_batch_get"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_batch_get"} =
    tikv_storage_engine_async_request_duration_seconds{type="snapshot"} +
    n * (
        tikv_engine_seek_micro_seconds{type="seek_max"} +
        read value duration +
        read value duration(non-short value)
    )

read value duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="batch_get"})) / sum(rate(
        '→ tikv_storage_rocksdb_perf{metric="block_read_count",req="batch_get"}))
```

TiKV 会首先得到一个快照，然后从同一个快照中读取多个值。read 操作的耗时和点查中的一致。当从磁盘中读取数据时，其平均耗时可以通过带有 `req="batch_get"` 属性的 `tikv_storage_rocksdb_perf` 来计算。

8.1.6.2.3 表扫描和索引扫描(Table Scan 和 Index Scan)

下面是表扫描和索引扫描的时间消耗图：

```
Diagram(
    Stack(
        NonTerminal("Resolve TSO"),
        NonTerminal("Load region cache for related table/index ranges"),
        OneOrMore(
            NonTerminal("Wait for result"),
            Comment("Next loop: drain the result"),
        ),
```

```
    ),
)
```

在进行表扫描和索引扫描时，耗时 `tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算：

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
    req_per_copr * (
        tidb_distsql_handle_query_duration_seconds{sql_type="general"}
    )
    tidb_distsql_handle_query_duration_seconds{sql_type="general"} <= send request duration
```

表扫描和索引扫描采用相同的方式处理。`req_per_copr` 是被分配的任务数量。由于执行协处理器和返回数据在不同的线程中运行，`tidb_distsql_handle_query_duration_seconds{sql_type="general"}` 是等待时间，并且小于 `send request duration`。

`send request duration` 和 `req_per_copr` 使用如下方式计算：

```
send request duration =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="coprocessor"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}

tikv_grpc_msg_duration_seconds{type="coprocessor"} =
    tikv_coprocessor_request_wait_seconds{type="snapshot"} +
    tikv_coprocessor_request_wait_seconds{type="schedule"} +
    tikv_coprocessor_request_handler_build_seconds{type="index/select"} +
    tikv_coprocessor_request_handle_seconds{type="index/select"}

req_per_copr = rate(tidb_distsql_handle_query_duration_seconds_count) / rate(
    ↪ tidb_distsql_scan_keys_partial_num_count)
```

在TiKV 中，表扫描的类型是 `select`，而索引扫描的类型是 `index`。`select` 和 `index` 类型的内部耗时是一致的。

8.1.6.2.4 索引回表(Index Look Up)

下面是通过索引回表操作的时间消耗图：

```
Diagram(
    Stack(
        NonTerminal("Resolve TSO"),
        NonTerminal("Load region cache for related index ranges"),
        OneOrMore(
            Sequence(
                NonTerminal("Wait for index scan result"),
                NonTerminal("Wait for table scan result"),
```

```
            ),
        Comment("Next loop: drain the result"),
        ),
    ),
)
```

在通过索引回表的过程中，耗时 `tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算:

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
    req_per_copr * (
        tidb_distsql_handle_query_duration_seconds{sql_type="general"}
    ) +
    req_per_copr *
        ( tidb_distsql_handle_query_duration_seconds{sql_type="general
        "}
    )

req_per_copr = rate(tidb_distsql_handle_query_duration_seconds_count) / rate(
    ↪ tidb_distsql_scan_keys_partial_num_count)
```

一次通过索引回表的过程结合了索引查找和表查找，其中索引查找和表查找按流水线方式处理。

8.1.6.3 写请求

写请求有多个变种，因而比读请求复杂得多。下面是写请求的时间消耗图：

```
Diagram(
    NonTerminal("Execute write query"),
    Choice(
        0,
        NonTerminal("Pessimistic lock keys"),
        Comment("bypass in optimistic transaction"),
    ),
    Choice(
        0,
        NonTerminal("Auto Commit Transaction"),
        Comment("bypass in non-auto-commit or explicit transaction"),
    ),
)
```

	悲观事务	乐观事务
Auto-commit	执行 + 加锁 + 提交	执行 + 提交
非 auto-commit	执行 + 加锁	执行

一次写请求可以被分解成以下三个阶段：

- 执行阶段：执行并把更改写入TiDB 的内存中
- 加锁阶段：获取执行结果的悲观锁
- 提交阶段：通过两阶段提交协议(2PC) 来提交事务

在执行阶段，TiDB 在内存中修改数据，其延迟主要源自读取所需的数据。对于更新和删除查询，TiDB 先从TiKV 读取数据，再更新或删除内存中的数据。

带有点查和批量点查的加锁时读取操作 (SELECT FOR UPDATE) 是一个例外。该操作会在单个 RPC (Remote Procedure Call) 请求中完成读取和加锁操作。

8.1.6.3.1 加锁时点查

下面是加锁时点查的时间消耗图：

```
Diagram(
    Choice(
        0,
        Sequence(
            NonTerminal("Read handle key by index key"),
            NonTerminal("Lock index key"),
        ),
        Comment("Clustered index"),
    ),
    NonTerminal("Lock handle key"),
    NonTerminal("Read value from pessimistic lock cache"),
)
```

在加锁时点查过程中，耗时 execution(clustered PK) 和 execution(non-clustered PK or UK) 使用如下方式计算：

```
execution(clustered PK) =
    tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
execution(non-clustered PK or UK) =
    2 * tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
```

加锁时点查会锁定键并获取其对应的值。相比先执行再获取锁的方式，该操作可以节省一次来回通信。加锁时点查的耗时可以看作与加锁耗时是一样的。

8.1.6.3.2 加锁时批量点查

下面是加锁时批量点查的时间消耗图：

```
Diagram(
    Choice(
        0,
        NonTerminal("Read handle keys by index keys"),
```

```
        Comment("Clustered index"),
    ),
    NonTerminal("Lock index and handle keys"),
    NonTerminal("Read values from pessimistic lock cache"),
)
```

在加锁时批量点查过程中，耗时 `execution(clustered PK)` 和 `execution(non-clustered PK or UK)` 使用如下方式计算：

```
execution(clustered PK) =
    tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
execution(non-clustered PK or UK) =
    tidb_tikvclient_txn_cmd_duration_seconds{type="batch_get"} +
    tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
```

加锁时批量点查的执行过程与加锁时点查相似。不同的点在于，加锁时批量点查会在单个RPC 请求中读取多个值。关于 `tidb_tikvclient_txn_cmd_duration_seconds{type="batch_get"}` 耗时的计算方式，请参考批量点查小节。

8.1.6.3.3 加锁阶段

本小节介绍加锁阶段的耗时。

```
round = ceil(
    sum(rate(tidb_tikvclient_txn_regions_num_sum{type="2pc_pessimistic_lock"})) /
    sum(rate(tidb_tikvclient_txn_regions_num_count{type="2pc_pessimistic_lock"})) /
    committer-concurrency
)

lock = tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"} =
    round * tidb_tikvclient_request_seconds{type="PessimisticLock"}
```

锁是按照两阶段锁的结构来获取的，带有流量控制机制。流量控制会按照 `committer-concurrency`（默认值为 128）来限制并发在线请求的数量。为了简单说明，流量控制可以看作是请求延迟 (round) 的倍增。

`tidb_tikvclient_request_seconds{type="PessimisticLock"}` 使用如下方式计算：

```
tidb_tikvclient_request_seconds{type="PessimisticLock"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_pessimistic_lock"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

关于`tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和`tidb_tikvclient_rpc_net_latency_seco`
`{store="?"}` 等批量请求客户端耗时的计算方式，请参考批量请求小节。

耗时 `tikv_grpc_msg_duration_seconds{type="kv_pessimistic_lock"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_pessimistic_lock"} =
    tikv_scheduler_latch_wait_duration_seconds{type="acquire_pessimistic_lock"} +
    tikv_storage_engine_async_request_duration_seconds{type="snapshot"} +
    (lock in-mem key count + lock on-disk key count) * lock read duration +
    lock on-disk key count / (lock in-mem key count + lock on-disk key count) *
    lock write duration
```

- 自TiDB v6.0 起，TiKV 默认使用内存悲观锁。内存悲观锁会跳过异步写入的过程。
- `tikv_storage_engine_async_request_duration_seconds{type="snapshot"}`是快照类型耗时，详情请参考TiKV 快照小节.
- `lock in-mem key count` 和 `lock on-disk key count` 使用如下方式计算：

```
lock in-mem key count =
    sum(rate(tikv_in_memory_pessimistic_locking{result="success"})) /
    sum(rate(tikv_grpc_msg_duration_seconds_count{type="kv_pessimistic_lock"}))

lock on-disk key count =
    sum(rate(tikv_in_memory_pessimistic_locking{result="full"})) /
    sum(rate(tikv_grpc_msg_duration_seconds_count{type="kv_pessimistic_lock"}))
```

内存和磁盘中被加锁的键数量可以从内存锁计数中计算得出。TiKV 在得到锁之前会读取键对应的值，其读取耗时可以从RocksDB performance context 中计算得出：

```
lock read duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="
        ↪ acquire_pessimistic_lock"})) / sum(rate(tikv_storage_rocksdb_perf{metric="
        ↪ block_read_count",req="acquire_pessimistic_lock"}))
```

- `lock write duration` 是写入磁盘锁的耗时，具体计算方式请参考异步写入小节。

8.1.6.3.4 提交阶段

本小节介绍提交阶段的耗时。下面是提交操作的时间消耗图：

```
Diagram(
    Stack(
        Sequence(
            Choice(
                0,
                Comment("use 2pc or causal consistency"),
                NonTerminal("Get min-commit-ts"),
            ),
            Optional("Async prewrite binlog"),
            NonTerminal("Prewrite mutations"),
```

```
            Optional("Wait prewrite binlog result"),
        ),
        Sequence(
            Choice(
                1,
                Comment("1pc"),
                Sequence(
                    Comment("2pc"),
                    NonTerminal("Get commit-ts"),
                    NonTerminal("Check schema"),
                    NonTerminal("Commit PK mutation"),
                ),
                Sequence(
                    Comment("async-commit"),
                    NonTerminal("Commit mutations asynchronously"),
                ),
            ),
            Choice(
                0,
                Comment("committed"),
                NonTerminal("Async cleanup"),
            ),
            Optional("Commit binlog"),
        ),
    ),
)
```

提交的耗时使用如下方式计算：

```
commit =
    Get_latest_ts_time +
    Prewrite_time +
    Get_commit_ts_time +
    Commit_time

Get_latest_ts_time = Get_commit_ts_time =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"}

prewrite_round =
    ceil( sum(rate(tidb_tikvclient_txn_regions_num_sum{type="2pc_prewrite"}
    )) /
    sum(rate(tidb_tikvclient_txn_regions_num_count{type="2pc_prewrite"})) /
    committer-concurrency
)

commit_round =
    ceil( sum(rate(tidb_tikvclient_txn_regions_num_sum{type="2pc_commit"})) /
```

```
    sum(rate(tidb_tikvclient_txn_regions_num_count{type="2pc_commit"})) /
    committer-concurrency
)

Prewrite_time =
    prewrite_round * tidb_tikvclient_request_seconds{type="Prewrite"}

Commit_time =
    commit_round * tidb_tikvclient_request_seconds{type="Commit"}
```

提交的耗时可以拆解为以下四个指标：

- `Get_latest_ts_time` 代表异步提交或单阶段(1PC)提交事务中获取最新TSO 的耗时。
- `Prewrite_time` 代表预先写阶段的耗时。
- `Get_commit_ts_time` 代表普通两阶段(2PC)事务的耗时。
- `Commit_time` 代表提交阶段的耗时。需要注意的是，异步提交和单阶段(1PC) 事务没有此阶段。

与悲观锁一样，流量控制充当延迟的放大，即上述公式中的 `prewrite_round` 和 `commit_round`。

`tidb_tikvclient_request_seconds{type="Prewrite"}` 和 `tidb_tikvclient_request_seconds{type="Commit"}` 的耗时使用如下方式计算：

```
tidb_tikvclient_request_seconds{type="Prewrite"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_prewrite"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}

tidb_tikvclient_request_seconds{type="Commit"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_commit"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

关于`tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和`tidb_tikvclient_rpc_net_latency_seco
↪ {store="?"}` 等批量请求客户端耗时的计算方式，请参考批量请求小节。

`tikv_grpc_msg_duration_seconds{type="kv_prewrite"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_prewrite"} =
    prewrite key count * prewrite read duration +
    prewrite write duration

prewrite key count =
    sum(rate(tikv_scheduler_kv_command_key_write_sum{type="prewrite"})) /
    sum(rate(tikv_scheduler_kv_command_key_write_count{type="prewrite"}))
```

```
prewrite read duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="prewrite"})) / sum(rate(
        ↪ tikv_storage_rocksdb_perf{metric="block_read_count",req="prewrite"}))
```

与 TiKV 中的锁一样，预先写在读取和写入阶段均进行了处理。读取阶段的耗时可以从 RocksDB performance context 计算。有关写入阶段耗时的计算方式，请参考异步写入部分。

`tikv_grpc_msg_duration_seconds{type="kv_commit"}`使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_commit"} =
    commit key count * commit read duration +
    commit write duration

commit key count =
    sum(rate(tikv_scheduler_kv_command_key_write_sum{type="commit"})) /
    sum(rate(tikv_scheduler_kv_command_key_write_count{type="commit"}))

commit read duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="commit"})) / sum(rate(
        ↪ tikv_storage_rocksdb_perf{metric="block_read_count",req="commit"})) (storage)
```

`kv_commit` 的耗时与 `kv_prewrite` 几乎一致。关于写入阶段耗时的计算方式，请参考异步写入小节。

8.1.6.4 批量请求

下面是批量请求客户端的时间消耗图：

```
Diagram(
    NonTerminal("Get conn pool to the target store"),
    Choice(
        0,
        Sequence(
            Comment("Batch  enabled"),
                NonTerminal("Push request to channel"),
                NonTerminal("Wait response"),
            ),
            Sequence(
            NonTerminal("Get conn from pool"),
            NonTerminal("Call RPC"),
            Choice(
                0,
                Comment("Unary call"),
                NonTerminal("Recv first"),
            ),
        ),
    ),
)
```

- 总体的发送请求耗时看作 tidb_tikvclient_request_seconds。
- RPC 客户端为每个存储维护各自的连接池（称为 ConnArray），每个连接池都包含带有一个发送批量请求 channel 的 BatchConn
- 绝大多数情况下，当存储是TiKV 并且batch 大小为正数时，批量请求开启。
- 批量请求 channel 的大小是tikv-client.max-batch-size 的值（默认值为 128）。请求入队的耗时看作 tidb_tikvclient_batch_wait_duration。
- 一共有 CmdBatchCop、CmdCopStream 和 CmdMPPConn 三种流式请求。流式请求会引入一个额外的 recv() 调用来获取流中的第一个响应。

tidb_tikvclient_request_seconds 大致使用如下方式计算（部分延迟不包含在内）：

```
tidb_tikvclient_request_seconds{type="?"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_?"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

- tidb_tikvclient_batch_wait_duration 记录批量请求系统的等待耗时。
- tidb_tikvclient_batch_send_latency 记录批量请求系统的编码耗时。
- tikv_grpc_msg_duration_seconds{type="kv_?"} 是TiKV 的处理耗时。
- tidb_tikvclient_rpc_net_latency_seconds 记录网络延迟。

8.1.6.5 TiKV 快照

下面是TiKV 快照操作的时间消耗图：

```
Diagram(
    Choice(
        0,
        Comment("Local Read"),
        Sequence(
            NonTerminal("Propose Wait"),
            NonTerminal("Read index Read Wait"),
        ),
    ),
    NonTerminal("Fetch A Snapshot From KV Engine"),
)
```

一个TiKV 快照的总体耗时可以从 tikv_storage_engine_async_request_duration_seconds{type="snapshot"} 指标查看，它的计算方式如下：

```
tikv_storage_engine_async_request_duration_seconds{type="snapshot"} =
    tikv_coprocessor_request_wait_seconds{type="snapshot"} =
    tikv_raftstore_request_wait_time_duration_secs +
    tikv_raftstore_commit_log_duration_seconds +
    get snapshot from rocksdb duration
```

当 leader lease 过期时，TiKV 会在从 RocksDB 获取快照之前提出读索引命令。tikv_raftstore_request_wait_time_duration_secs 和 tikv_raftstore_commit_log_duration_seconds 是提交读索引命令的耗时。

从 RocksDB 获取快照通常是一个快速操作，因此 get snapshot from rocksdb duration 的耗时可以被忽略。

8.1.6.6 异步写入

异步写入是 TiKV 通过回调将数据异步写入基于 Raft 的复制状态机 (Replicated State Machine) 的过程。

- 下面是异步 IO 未开启时，异步写入过程的时间消耗图：

```
Diagram(
    NonTerminal("Propose Wait"),
    NonTerminal("Process Command"),
    Choice(
        0,
        Sequence(
            NonTerminal("Wait Current Batch"),
            NonTerminal("Write to Log Engine"),
        ),
        Sequence(
            NonTerminal("RaftMsg Send Wait"),
            NonTerminal("Commit Log Wait"),
        ),
    ),
    NonTerminal("Apply Wait"),
    NonTerminal("Apply Log"),
)
```

- 下面是异步 IO 开启时，异步写入过程的时间消耗图：

```
Diagram(
    NonTerminal("Propose Wait"),
    NonTerminal("Process Command"),
    Choice(
        0,
        NonTerminal("Wait Until Persisted by Write Worker"),
        Sequence(
            NonTerminal("RaftMsg Send Wait"),
            NonTerminal("Commit Log Wait"),
        ),
    ),
    NonTerminal("Apply Wait"),
    NonTerminal("Apply Log"),
)
```

异步写入耗时的计算方式如下：

```
async write duration(async io disabled) =
    propose +
    async io disabled commit +
    tikv_raftstore_apply_wait_time_duration_secs +
    tikv_raftstore_apply_log_duration_seconds

async write duration(async io enabled) =
    propose +
    async io enabled commit +
    tikv_raftstore_apply_wait_time_duration_secs +
    tikv_raftstore_apply_log_duration_seconds
```

异步写入可以拆解为以下三个阶段：

- 提案阶段(Propose)
- 提交阶段(Commit)
- 应用阶段(Apply)：对应上面公式中的tikv_raftstore_apply_wait_time_duration_secs + tikv_raftstore_apply_log_dur
  ↪

提案阶段耗时的计算方式如下：

```
propose =
    propose wait duration +
    propose duration

propose wait duration =
    tikv_raftstore_store_wf_batch_wait_duration_seconds

propose duration =
    tikv_raftstore_store_wf_send_to_queue_duration_seconds -
    tikv_raftstore_store_wf_batch_wait_duration_seconds
```

Raft 过程以瀑布方式记录，因此，提案阶段的耗时是根据tikv_raftstore_store_wf_send_to_queue_duration_seconds
↪ 和 tikv_raftstore_store_wf_batch_wait_duration_seconds 两个指标之间的差值计算的。

提交阶段耗时的计算方式如下：

```
async io disabled commit =
    max( persist log locally
    duration, replicate log
    duration
)

async io enabled commit =
    max( wait by write worker
    duration, replicate log
    duration
```

从 TiDB v5.3.0 开始，TiKV 支持通过 StoreWriter 线程池写入Raft 日志，即异步IO。异步IO 会改变提交过程，仅在store-io-pool-size 数值大于 0 时启用。耗时 persist log locally duration 和 wait by write worker duration 的计算方式如下：

```
persist log locally duration =
    batch wait duration +
    write to raft db duration

batch wait duration =
    tikv_raftstore_store_wf_before_write_duration_seconds -
    tikv_raftstore_store_wf_send_to_queue_duration_seconds

write to raft db duration =
    tikv_raftstore_store_wf_write_end_duration_seconds -
    tikv_raftstore_store_wf_before_write_duration_seconds

wait by write worker duration =
    tikv_raftstore_store_wf_persist_duration_seconds -
    tikv_raftstore_store_wf_send_to_queue_duration_seconds
```

是否开启异步IO 的区别在于本地持久化日志的耗时。使用异步IO 可以直接从瀑布指标中计算本地持久化日志的耗时，忽略批处理等待耗时。

replicate log duration 代表quorum 副本中日志持久化的耗时，其中包含RPC 耗时和大多数日志持久化的耗时。replicate log duration 耗时的计算方式如下：

```
replicate log duration =
    raftmsg send wait duration +
    commit log wait duration

raftmsg send wait duration =
    tikv_raftstore_store_wf_send_proposal_duration_seconds -
    tikv_raftstore_store_wf_send_to_queue_duration_seconds

commit log wait duration =
    tikv_raftstore_store_wf_commit_log_duration -
    tikv_raftstore_store_wf_send_proposal_duration_seconds
```

8.1.6.6.1 Raft DB

下面是Raft DB 的时间消耗图：

```
Diagram(
    NonTerminal("Wait for Writer Leader"),
    NonTerminal("Write and Sync Log"),
    NonTerminal("Apply Log to Memtable"),
)
```

```
write to raft db duration = raft db write duration
commit log wait duration >= raft db write duration

raft db write duration(raft engine enabled) =
    raft_engine_write_preprocess_duration_seconds +
    raft_engine_write_leader_duration_seconds +
    raft_engine_write_apply_duration_seconds

raft db write duration(raft engine disabled) =
    tikv_raftstore_store_perf_context_time_duration_secs{type="write_thread_wait"} +
    tikv_raftstore_store_perf_context_time_duration_secs{type="
        ↪ write_scheduling_flushes_compactions_time"} +
    tikv_raftstore_store_perf_context_time_duration_secs{type="write_wal_time"} +
    tikv_raftstore_store_perf_context_time_duration_secs{type="write_memtable_time"}
```

`commit log wait duration` 是 quorum 副本中最长的耗时，可能大于 `raft db write duration`。

从 TiDB v6.1.0 开始，TiKV 默认使用Raft Engine 作为日志存储引擎，这将改变写入日志的过程。

8.1.6.6.2 KV DB

下面是 KV DB 的时间消耗图：

```
Diagram(
    NonTerminal("Wait for Writer Leader"),
    NonTerminal("Preprocess"),
    Choice(
        0,
        Comment("No Need to Switch"),
        NonTerminal("Switch WAL or Memtable"),
    ),
    NonTerminal("Write and Sync WAL"),
    NonTerminal("Apply to Memtable"),
)
```

```
tikv_raftstore_apply_log_duration_seconds =
    tikv_raftstore_apply_perf_context_time_duration_secs{type="write_thread_wait"} +
    tikv_raftstore_apply_perf_context_time_duration_secs{type="
        ↪ write_scheduling_flushes_compactions_time"} +
    tikv_raftstore_apply_perf_context_time_duration_secs{type="write_wal_time"} +
    tikv_raftstore_apply_perf_context_time_duration_secs{type="write_memtable_time"}
```

在异步写入过程中，提交的日志需要应用到 KV DB 中，应用耗时可以根据RocksDB performance context 进行计算。

8.1.6.7 诊断场景

前面的部分详细介绍了 SQL 查询过程中执行时间的细粒度指标。本小节主要介绍遇到慢读取或慢写入查询时常见的指标分析过程。所有指标均可在 Performance Overview 面板的 Database Time 中查看。

8.1.6.7.1 慢读取查询

如果 SELECT 语句占 Database Time 的很大一部分，你可以认为TiDB 在读查询时速度很慢。

慢查询的执行计划可以在TiDB Dashboard 中的Top SQL 语句区域查看。要分析慢读取查询的耗时，你可以根据前面的描述分析点查、批量点查和表扫描和索引扫描的耗时情况。

8.1.6.7.2 慢写入查询

在分析慢写入查询之前，你需要查看 `tikv_scheduler_latch_wait_duration_seconds_sum{type="acquire_pessimistic_lock"} by (instance)` 指标来确认冲突的原因：

- 如果这个指标在某些特定的TiKV 实例中很高，则在热点区域可能会存在冲突。
- 如果这个指标在所有实例中都很高，则业务中可能存在冲突。

如果是业务中存在冲突，那么你可以分析加锁和提交阶段的耗时。

8.2 配置调优

8.2.0.1 操作系统性能参数调优

本文档仅用于描述如何优化 CentOS 7 的各个子系统。

> 注意：
>
> - CentOS 7 操作系统的默认配置适用于中等负载下运行的大多数服务。调整特定子系统的性能可能会对其他子系统产生负面影响。因此在调整系统之前，请备份所有用户数据和配置信息；
> - 请在测试环境下对所有修改做好充分测试后，再应用到生产环境中。

8.2.0.1.1 性能分析工具

系统调优需要根据系统性能分析的结果做指导，因此本文先列出常用的性能分析方法。

60 秒分析法

60 秒分析法由《性能之巅》的作者 Brendan Gregg 及其所在的 Netflix 性能工程团队公布。所用到的工具均可从发行版的官方源获取，通过分析以下清单中的输出，可定位大部分常见的性能问题。

- `uptime`
- `dmesg | tail`
- `vmstat 1`

- `mpstat -P ALL 1`
- `pidstat 1`
- `iostat -xz 1`
- `free -m`
- `sar -n DEV 1`
- `sar -n TCP,ETCP 1`
- `top`

具体用法可查询相应 `man` 手册。

perf

perf 是 Linux 内核提供的一个重要的性能分析工具，它涵盖硬件级别（CPU/PMU 和性能监视单元）功能和软件功能（软件计数器和跟踪点）。详细用法请参考 perf Examples。

BCC/bpftrace

CentOS 从 7.6 版本起，内核已实现对 BPF (Berkeley Packet Filter) 的支持，因此可根据上述清单的结果，选取适当的工具进行深入分析。相比perf/ftrace，BPF 提供了可编程能力和更小的性能开销。相比 kprobe，BPF 提供了更高的安全性，更适合在生产环境上使用。关于 BCC 工具集的使用请参考 BPF Compiler Collection (BCC)。

8.2.0.1.2 性能调优

性能调优将根据内核子系统进行分类描述。

处理器——动态节能技术

cpufreq 是一个动态调整CPU 频率的模块，可支持五种模式。为保证服务性能应选用performance 模式，将CPU 频率固定工作在其支持的最高运行频率上，不进行动态调节，操作命令为 `cpupower frequency-set --governor performance`。

处理器——中断亲和性

- 自动平衡：可通过 `irqbalance` 服务实现。
- 手动平衡：
  - 确定需要平衡中断的设备，从 CentOS 7.5 开始，系统会自动为某些设备及其驱动程序配置最佳的中断关联性。不能再手动配置其亲和性。目前已知的有使用 `be2iscsi` 驱动的设备，以及 NVMe 设置；
  - 对于其他设备，可查询其芯片手册，是否支持分发中断。
    * 若不支持，则该设备的所有中断会路由到同一个 CPU 上，无法对其进行修改。
    * 若支持，则计算 `smp_affinity` 掩码并设置对应的配置文件，具体请参考内核IRQ-affinity 文档。

NUMA 绑核

为尽可能的避免跨 NUMA (Non-Uniform Memory Access) 访问内存，可以通过设置线程的 CPU 亲和性来实现 NUMA 绑核。对于普通程序，可使用 `numactl` 命令来绑定，具体用法请查询 `man` 手册。对于网卡中断，请参考下文网络章节。

内存——透明大页

对于数据库应用，不推荐使用 THP，因为数据库往往具有稀疏而不是连续的内存访问模式，且当高阶内存碎片化比较严重时，分配 THP 页面会出现较大的延迟。若开启针对 THP 的直接内存规整功能，也会出现系统 CPU 使用率激增的现象，因此建议关闭 THP。

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

内存——虚拟内存参数

- `dirty_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统将开始使用 pdflush 操作将脏的page cache 写入磁盘。默认值为 20%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，降低其值有利于提高内存回收时的效率。
- `dirty_background_ratio` 百分比值。当脏的page cache 总量达到系统内存总量的这一百分比后，系统开始在后台将脏的page cache 写入磁盘。默认值为 10%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，设置较低的值有利于提高内存回收时的效率。

存储及文件系统

内核I/O 栈链路较长，包含了文件系统层、块设备层和驱动层。

I/O 调度器

I/O 调度程序确定I/O 操作何时在存储设备上运行以及持续多长时间。也称为 I/O 升降机。对于 SSD 设备，宜设置为 noop。

```
echo noop > /sys/block/${SSD_DEV_NAME}/queue/scheduler
```

格式化参数——块大小

块 (block) 是文件系统的工作单元。块大小决定了单个块中可以存储多少数据，因此决定了一次写入或读取的最小数据量。

默认块大小适用于大多数使用情况。但是，如果块大小（或多个块的大小）与通常一次读取或写入的数据量相同或稍大，则文件系统将性能更好，数据存储效率更高。小文件仍将使用整个块。文件可以分布在多个块中，但这会增加运行时开销。

使用 `mkfs` 命令格式化设备时，将块大小指定为文件系统选项的一部分。指定块大小的参数随文件系统的不同而不同。有关详细信息，请查询对应文件系统的 `mkfs` 手册页，比如 `man mkfs.ext4`。

挂载参数

noatime 读取文件时，将禁用对元数据的更新。它还启用了nodiratime 行为，该行为会在读取目录时禁用对元数据的更新。

网络

网络子系统由具有敏感连接的许多不同部分组成。因此，CentOS 7 网络子系统旨在为大多数工作负载提供最佳性能，并自动优化其性能。因此，通常无需手动调整网络性能。

网络问题通常由硬件或相关设施出现问题导致的，因此在调优协议栈前，请先排除硬件问题。

尽管网络堆栈在很大程度上是自我优化的。但是在网络数据包处理过程中，以下方面可能会成为瓶颈并降低性能：

- 网卡硬件缓存：正确观察硬件层面的丢包方法是使用 `ethtool -S ${NIC_DEV_NAME}` 命令观察 `drops` 字段。当出现丢包现象时，主要考虑是硬/软中断的处理速度跟不上网卡接收速度。若接收缓存小于最大

限制时，也可尝试增加RX 缓存来防止丢包。查询命令为：`ethtool -g ${NIC_DEV_NAME}`，修改命令为 `ethtool -G ${NIC_DEV_NAME}`。

- 硬中断：若网卡支持Receive-Side Scaling（RSS 也称为多网卡接收）功能，则观察 `/proc/interrupts` 网卡中断，如果出现了中断不均衡的情况，请参考处理器动态节能技术，处理器调优，NUMA 绑核。若不支持RSS 或 RSS 数量远小于物理 CPU 核数，则可配置Receive Packet Steering（RPS，可以看作 RSS 的软件实现），及 RPS 的扩展Receive Flow Steering (RFS)。具体设置请参考内核文档。
- 软中断：观察 `/proc/net/softnet_stat` 监控。如果除第三列的其他列的数值在增长，则应适度调大 `net.core.netdev_budget` 或 `net.core.dev_weight` 值，使 `softirq` 可以获得更多的 CPU 时间。除此之外，也需要检查 CPU 使用情况，确定哪些任务在频繁占用 CPU，能否优化。
- 应用的套接字接收队列：监控 `ss -nmp` 的 `Recv-q` 列，若队列已满，则应考虑增大应用程序套接字的缓存大小或使用自动调整缓存的方式。除此之外，也要考虑能否优化应用层的架构，降低读取套接字的间隔。
- 以太网流控：若网卡和交换机支持流控功能，可通过使能此功能，给内核一些时间来处理网卡队列中的数据，来规避网卡缓存溢出的问题。对于网卡测，可通过 `ethtool -a ${NIC_DEV_NAME}` 命令检查是否支持/使能，并通过 `ethtool -A ${NIC_DEV_NAME}` 命令开启。对于交换机，请查询其手册。
- 中断合并：过于频繁的硬件中断会降低系统性能，而过晚的硬件中断会导致丢包。对于较新的网卡支持中断合并功能，并允许驱动自动调节硬件中断数。可通过 `ethtool -c ${NIC_DEV_NAME}` 命令检查，`ethtool -C ${NIC_DEV_NAME}` 命令开启。自适应模式使网卡可以自动调节中断合并。在自适应模式下，驱动程序将检查流量模式和内核接收模式，并实时评估合并设置，以防止数据包丢失。不同品牌的网卡具有不同的功能和默认配置，具体请参考网卡手册。
- 适配器队列：在协议栈处理之前，内核利用此队列缓存网卡接收的数据，每个 CPU 都有各自的backlog 队列。此队列可缓存的最大 packets 数量为 `netdev_max_backlog`。观察 `/proc/net/softnet_stat` 第二列，当某行的第二列持续增加，则意味着 CPU [row-1] 队列已满，数据包被丢失，可通过持续加倍 `net.core.netdev_max_backlog` 值来解决。
- 发送队列：发送队列长度值确定在发送之前可以排队的数据包数量。默认值是 1000，对于 10 Gbps 足够。但若从 `ip -s link` 的输出中观察到 `TX errors` 值时，可尝试加倍该数据包数量：`ip link set dev ${ ↪ NIC_DEV_NAME} txqueuelen 2000`。
- 驱动：网卡驱动通常也会提供调优参数，请查询设备硬件手册及其驱动文档。

8.2.0.2 TiDB 内存控制文档

目前TiDB 已经能够做到追踪单条 SQL 查询过程中的内存使用情况，当内存使用超过一定阈值后也能采取一些操作来预防 OOM 或者排查 OOM 原因。你可以使用系统变量tidb_mem_oom_action 来控制查询超过内存限制后所采取的操作：

- 如果变量值为 LOG，那么当一条 SQL 的内存使用超过一定阈值（由 session 变量 `tidb_mem_quota_query` 控制）后，这条 SQL 会继续执行，但TiDB 会在 log 文件中打印一条 LOG。
- 如果变量值为 CANCEL，那么当一条 SQL 的内存使用超过一定阈值后，TiDB 会立即中断这条 SQL 的执行，并给客户端返回一个错误，错误信息中会详细写明在这条 SQL 执行过程中占用内存的各个物理执行算子的内存使用情况。

8.2.0.2.1 如何配置一条 SQL 执行过程中的内存使用阈值

使用系统变量tidb_mem_quota_query 来配置一条 SQL 执行过程中的内存使用阈值，单位为字节。例如：

配置整条 SQL 的内存使用阈值为 8GB：

```
SET tidb_mem_quota_query = 8 << 30;
```

配置整条 SQL 的内存使用阈值为 8MB：

```
SET tidb_mem_quota_query = 8 << 20;
```

配置整条 SQL 的内存使用阈值为 8KB：

```
SET tidb_mem_quota_query = 8 << 10;
```

8.2.0.2.2 如何配置tidb-server 实例使用内存的阈值

自 v6.5.0 版本起，可以通过系统变量tidb_server_memory_limit 设置tidb-server 实例的内存使用阈值。

例如，配置tidb-server 实例的内存使用总量，将其设置成为 32 GB：

```
SET GLOBAL tidb_server_memory_limit = "32GB";
```

设置该变量后，当tidb-server 实例的内存用量达到 32 GB 时，TiDB 会依次终止正在执行的 SQL 操作中内存用量最大的 SQL 操作，直至tidb-server 实例内存使用下降到 32 GB 以下。被强制终止的 SQL 操作会向客户端返回报错信息 `Out Of Memory Quota!`。

当前 `tidb_server_memory_limit` 所设的内存限制不终止以下 SQL 操作：

- DDL 操作
- 包含窗口函数和公共表表达式的 SQL 操作

> 警告：
>
> - TiDB 在启动过程中不保证tidb_server_memory_limit 限制生效。如果操作系统的空闲内存不足，TiDB 仍有可能出现 OOM。你需要确保 TiDB 实例有足够的可用内存。
> - 在内存控制过程中，TiDB 的整体内存使用量可能会略微超过 `tidb_server_memory_limit` 的限制。
> - `server-memory-quota` 配置项自 v6.5.0 起被废弃。为了保证兼容性，在升级到 v6.5.0 或更高版本的集群后，`tidb_server_memory_limit` 会继承配置项 `server-memory-quota` 的值。如果集群在升级至 v6.5.0 或更高版本前没有配置 `server-memory-quota`，`tidb_server_memory_limit` 会使用默认值，即 80%。

在tidb-server 实例内存用量到达总内存的一定比例时（比例由系统变量tidb_server_memory_limit_gc_trigger 控制），tidb-server 会尝试主动触发一次Golang GC 以缓解内存压力。为了避免实例内存在阈值上下范围不断波动导致频繁GC 进而带来的性能问题，该GC 方式 1 分钟最多只会触发 1 次。

> 注意：
>
> 在混合部署的情况下，tidb_server_memory_limit 为单个tidb-server 实例的内存使用阈值，而不是整个物理机的总内存阈值。

8.2.0.2.3 使用INFORMATION_SCHEMA 系统表查看当前tidb-server 的内存用量

要查看当前实例或集群的内存使用情况，你可以查询系统表 INFORMATION_SCHEMA.(CLUSTER_)MEMORY_USAGE。

要查看本实例或集群中内存相关的操作和执行依据， 可以查询系统表 INFORMATION_SCHEMA.(CLUSTER_)MEMORY_USAGE_OPS_HISTORY。对于每个实例，该表保留最近 50 条记录。

8.2.0.2.4 tidb-server 内存占用过高时的报警

当 tidb-server 实例的内存使用量超过内存阈值（默认为总内存量的 70%）且满足以下任一条件时，TiDB 将记录相关状态文件，并打印报警日志。

- 第一次内存使用量超过内存阈值。
- 内存使用量超过内存阈值，且距离上一次报警超过 60 秒。
- 内存使用量超过内存阈值，且(本次内存使用量 - 上次报警时内存使用量)/ 总内存量 > 10%。

你可以通过系统变量tidb_memory_usage_alarm_ratio 修改触发该报警的内存使用比率，从而控制内存报警的阈值。

当触发tidb-server 内存占用过高的报警时，TiDB 的报警行为如下：

- TiDB 将以下信息记录到TiDB 日志文件filename 所在目录中。
  - 当前正在执行的所有 SQL 语句中内存使用最高的 10 条语句和运行时间最长的 10 条语句的相关信息
  - goroutine 栈信息
  - 堆内存使用状态
- TiDB 将输出一条包含关键字 tidb-server has the risk of OOM 以及以下内存相关系统变量的日志。
  - tidb_mem_oom_action
  - tidb_mem_quota_query
  - tidb_server_memory_limit
  - tidb_analyze_version
  - tidb_enable_rate_limit_action

为避免报警时产生的状态文件累积过多，目前TiDB 默认只保留最近 5 次报警时所生成的状态文件。你可以通过配置系统变量tidb_memory_usage_alarm_keep_record_num 调整该次数。

下例通过构造一个占用大量内存的 SQL 语句触发报警，对该报警功能进行演示：

1. 配置报警比例为 0.85：

```
SET GLOBAL tidb_memory_usage_alarm_ratio = 0.85;
```

2. 创建单表 CREATE TABLE t(a int); 并插入 1000 行数据。

3. 执行 select * from t t1 join t t2 join t t3 order by t1.a。该 SQL 语句会输出 1000000000 条记录，占用巨大的内存，进而触发报警。

4. 检查 tidb.log 文件，其中会记录系统总内存、系统当前内存使用量、tidb-server 实例的内存使用量以及状态文件所在目录。

```
[2022/10/11 16:39:02.281 +08:00] [WARN] [memoryusagealarm.go:212] ["tidb-server has the risk
    ↪  of OOM because of memory usage exceeds alarm ratio. Running SQLs and heap profile
    ↪ will be recorded in record path"] ["is tidb_server_memory_limit set"=false] ["system
    ↪  memory total"=33682427904] ["system memory usage"=22120655360] ["tidb-server memory
    ↪  usage"=21468556992] [memory-usage-alarm-ratio=0.85] ["record path"=/tiup/deploy/
    ↪ tidb-4000/log/oom_record]
```

   以上 Log 字段的含义如下：

   - is tidb_server_memory_limit set：表示系统变量tidb_server_memory_limit 是否被设置
   - system memory total：表示当前系统的总内存
   - system memory usage：表示当前系统的内存使用量
   - tidb-server memory usage：表示 tidb-server 实例的内存使用量
   - memory-usage-alarm-ratio：表示系统变量tidb_memory_usage_alarm_ratio 的值
   - record path：表示状态文件存放的目录

5. 通过访问状态文件所在目录（该示例中的目录为 /tiup/deploy/tidb-4000/log/oom_record），可以看到标记了记录时间的record 目录（例：record2022-10-09T17:18:38+08:00），其中包括 goroutinue、heap、running_sql 3 个文件，文件以记录状态文件的时间为后缀。这 3 个文件分别用来记录报警时的 goroutine 栈信息，堆内存使用状态，及正在运行的SQL 信息。其中running_sql 文件内容请参考expensive-queries。

8.2.0.2.5 tidb-server 其它内存控制策略

流量控制

- TiDB 支持对读数据算子的动态内存控制功能。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，读数据的算子就会停止一个线程。
- 流控行为由参数tidb_enable_rate_limit_action 控制。
- 当流控被触发时，会在日志中打印一条包含关键字 memory exceeds quota, destroy one token now 的日志。

数据落盘

TiDB 支持对执行算子的数据落盘功能。当 SQL 的内存使用超过 Memory Quota 时，tidb-server 可以通过落盘执行算子的中间数据，缓解内存压力。支持落盘的算子有：Sort、 MergeJoin、 HashJoin、 HashAgg。

- 落盘行为由参数tidb_mem_quota_query、 tidb_enable_tmp_storage_on_oom、 tmp-storage-path、 tmp-storage-quota 共同控制。
- 当落盘被触发时，TiDB 会在日志中打印一条包含关键字 memory exceeds quota, spill to disk now 或 memory exceeds quota, set aggregate mode to spill-mode 的日志。
- Sort、 MergeJoin、 HashJoin 落盘是从 v4.0.0 版本开始引入的，HashAgg 落盘是从 v5.2.0 版本开始引入的。
- 当包含Sort、MergeJoin 或HashJoin 的SQL 语句引起内存OOM 时，TiDB 默认会触发落盘。当包含HashAgg 算子的 SQL 语句引起内存 OOM 时，TiDB 默认不触发落盘，请设置系统变量 tidb_executor_concurrency = 1 来触发 HashAgg 落盘功能。

> 注意：
>
> - HashAgg 落盘功能目前不支持distinct 聚合函数。使用distinct 函数且内存占用过大时，无法进行落盘。

本示例通过构造一个占用大量内存的 SQL 语句，对 HashAgg 落盘功能进行演示：

1. 将 SQL 语句的 Memory Quota 配置为 1GB（默认 1GB）：

```
SET tidb_mem_quota_query = 1 << 30;
```

2. 创建单表 CREATE TABLE t(a int); 并插入 256 行不同的数据。
3. 尝试执行以下 SQL 语句：

```
[tidb]> explain analyze select /*+ HASH_AGG() */ count(*) from t t1 join t t2 join t t3
    '→ group by t1.a, t2.a, t3.a;
```

该 SQL 语句占用大量内存，返回 Out of Memory Quota 错误。

```
ERROR 1105 (HY000): Out Of Memory Quota![conn_id=3]
```

4. 设置系统变量tidb_executor_concurrency 将执行器的并发度调整为1。在此配置下，内存不足时HashAgg 会自动尝试触发落盘。

```
SET tidb_executor_concurrency = 1;
```

5. 执行相同的 SQL 语句，不再返回错误，可以执行成功。从详细的执行计划可以看出，HashAgg 使用了 600MB 的硬盘空间。

```
[tidb]> explain analyze select /*+ HASH_AGG() */ count(*) from t t1 join t t2 join t t3
    '→ group by t1.a, t2.a, t3.a;
```

```
+--
    '→ ------------------------------+-------------+----------+-----------+---------------+---------------
    '→
```

```
| id                              | estRows      | actRows  | task      | access object |
    ’→ execution info
    ’→
    ’→ | operator info                                                | memory    | disk
    ’→      |
+--
    ’→  -------------------------------+-------------+----------+-----------+---------------+--------------
    ’→
| HashAgg_11                      | 204.80       | 16777216 | root      |               |
    ’→ time:1m37.4s, loops:16385
    ’→
    ’→ | group by:test.t.a, test.t.a, test.t.a, funcs:count(1)->Column#7 | 1.13 GB    |
    ’→ 600.0 MB |
| └─HashJoin_12                   | 16777216.00 | 16777216 | root      |               |
    ’→ time:21.5s, loops:16385, build_hash_table:{total:267.2µs, fetch:228.9µs, build:38.2
    ’→ µs}, probe:{concurrency:1, total:35s, max:35s, probe:35s, fetch:962.2µs}         |
    ’→ CARTESIAN inner join                                          | 8.23 KB    | 4 KB
    ’→     |
|   ├─TableReader_21(Build)       | 256.00       | 256      | root      |               |
    ’→ time:87.2µs, loops:2, cop_task: {num: 1, max: 150µs, proc_keys: 0, rpc_num: 1,
    ’→ rpc_time: 145.1µs, copr_cache_hit_ratio: 0.00}
    ’→                                        | data:TableFullScan_20
    ’→                                            | 885 Bytes | N/A       |
|   │ └─TableFullScan_20          | 256.00       | 256      | cop[tikv] | table:t3      |
    ’→ tikv_task:{time:23.2µs, loops:256}
    ’→
    ’→ | keep order:false, stats:pseudo                                | N/A        | N/A
    ’→      |
|   └─HashJoin_14(Probe)          | 65536.00     | 65536    | root      |               |
    ’→ time:728.1µs, loops:65, build_hash_table:{total:307.5µs, fetch:277.6µs, build:29.9µs
    ’→ }, probe:{concurrency:1, total:34.3s, max:34.3s, probe:34.3s, fetch:278µs}      |
    ’→ CARTESIAN inner join                                          | 8.23 KB    | 4 KB
    ’→     |
|     ├─TableReader_19(Build)     | 256.00       | 256      | root      |               |
    ’→ time:126.2µs, loops:2, cop_task: {num: 1, max: 308.4µs, proc_keys: 0, rpc_num: 1,
    ’→ rpc_time: 295.3µs, copr_cache_hit_ratio: 0.00}                                  |
    ’→ data:TableFullScan_18                                         | 885 Bytes | N/A
    ’→      |
|     │ └─TableFullScan_18        | 256.00       | 256      | cop[tikv] | table:t2      |
    ’→ tikv_task:{time:79.2µs, loops:256}
    ’→
    ’→ | keep order:false, stats:pseudo                                | N/A        | N/A
    ’→      |
|     └─TableReader_17(Probe)     | 256.00       | 256      | root      |               |
    ’→ time:211.1µs, loops:2, cop_task: {num: 1, max: 295.5µs, proc_keys: 0, rpc_num: 1,
```

```
    ’→ rpc_time: 279.7µs, copr_cache_hit_ratio: 0.00}                                                    |
    ’→ data:TableFullScan_16                                                  | 885 Bytes | N/A
    ’→      |
|       └─TableFullScan_16         | 256.00       | 256       | cop[tikv] | table:t1       |
    ’→ tikv_task:{time:71.4µs, loops:256}
    ’→
    ’→ | keep order:false, stats:pseudo                                       | N/A        | N/A
    ’→      |
+--
    ’→  ------------------------------+-------------+----------+-----------+---------------+--------------
    ’→
9 rows in set (1 min 37.428 sec)
```

8.2.0.2.6 其它

设置环境变量 GOMEMLIMIT 缓解 OOM 问题

Golang 自 Go 1.19 版本开始引入 GOMEMLIMIT 环境变量，该变量用来设置触发Go GC 的内存上限。

对于 v6.1.3 <= TiDB < v6.5.0 的版本，你可以通过手动设置 Go GOMEMLIMIT 环境变量的方式来缓解一类 OOM 问题。该类 OOM 问题具有一个典型特征：观察Grafana 监控，OOM 前的时刻，TiDB-Runtime > Memory Usage 面板中 estimate-inuse 立柱部分在整个立柱中仅仅占一半。如下图所示：

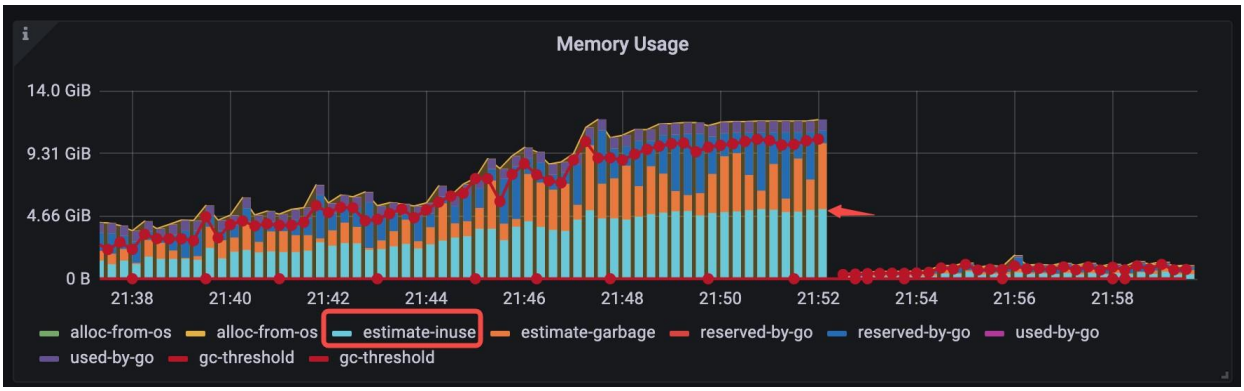


图 138: normal OOM case example

为了验证 GOMEMLIMIT 在该类场景下的效果，以下通过一个对比实验进行说明：

- 在 TiDB v6.1.2 下，模拟负载在持续运行几分钟后，TiDB server 会发生 OOM（系统内存约 48 GiB）：



图 139: v6.1.2 workload oom

- 在 TiDB v6.1.3 下，设置 GOMEMLIMIT 为 40000 MiB，模拟负载长期稳定运行、TiDB server 未发生 OOM 且进程最高内存用量稳定在 40.8 GiB 左右：

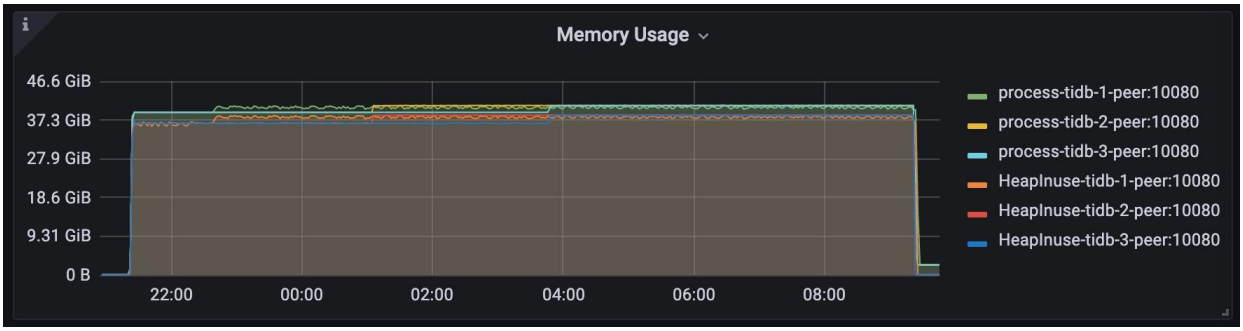


图 140: v6.1.3 workload no oom with GOMEMLIMIT

8.2.0.3 TiKV 线程池性能调优

本文主要介绍TiKV 线程池性能调优的主要手段，以及TiKV 内部线程池的主要用途。

8.2.0.3.1 线程池介绍

在TiKV 中，线程池主要由 gRPC、Scheduler、UnifyReadPool、Raftstore、StoreWriter、Apply、RocksDB 以及其它一些占用 CPU 不多的定时任务与检测组件组成，这里主要介绍几个占用 CPU 比较多且会对用户读写请求的性能产生影响的线程池。

- gRPC 线程池：负责处理所有网络请求，它会把不同任务类型的请求转发给不同的线程池。
- Scheduler 线程池：负责检测写事务冲突，把事务的两阶段提交、悲观锁上锁、事务回滚等请求转化为 key-value 对数组，然后交给Raftstore 线程进行Raft 日志复制。
- Raftstore 线程池：
  - 处理所有的Raft 消息以及添加新日志的提议(Propose)。

  - 处理 Raft 日志。如果store-io-pool-size 配置项的值为 0，Raftstore 线程将日志写入到磁盘；如果该值不为 0，Raftstore 线程将日志发送给 StoreWriter 线程处理。
  - 当日志在多数副本中达成一致后，Raftstore 线程把该日志发送给Apply 线程处理。
- StoreWriter 线程池：负责将所有Raft 日志写入到磁盘，再把结果返回到Raftstore 线程。
- Apply 线程池：当收到从Raftstore 线程池发来的已提交日志后，负责将其解析为 key-value 请求，然后写入 RocksDB 并且调用回调函数通知 gRPC 线程池中的写请求完成，返回结果给客户端。
- RocksDB 线程池：RocksDB 进行 Compact 和 Flush 任务的线程池，关于RocksDB 的架构与 Compact 操作请参考 RocksDB: A Persistent Key-Value Store for Flash and RAM Storage。
- UnifyReadPool 线程池：由 Coprocessor 线程池与 Storage Read Pool 合并而来，所有的读取请求包括 kv get、 kv batch get、 raw kv get、 coprocessor 等都会在这个线程池中执行。

8.2.0.3.2 TiKV 的只读请求

TiKV 的读取请求分为两类：

- 一类是指定查询某一行或者某几行的简单查询，这类查询会运行在 Storage Read Pool 中。
- 另一类是复杂的聚合计算、范围查询，这类请求会运行在 Coprocessor Read Pool 中。

从 TiKV 5.0 版本起，默认所有的读取请求都通过统一的线程池进行查询。如果是从TiKV 4.0 升级上来的TiKV 集群且升级前未打开 `readpool.storage` 的 `use-unified-pool` 配置，则升级后所有的读取请求仍然继续使用独立的线程池进行查询，可以将 `readpool.storage.use-unified-pool` 设置为 `true` 使所有的读取请求通过统一的线程池进行查询。

8.2.0.3.3 TiKV 线程池调优

- gRPC 线程池的大小默认配置(`server.grpc-concurrency`) 是 5。由于 gRPC 线程池几乎不会有多少计算开销，它主要负责网络IO、反序列化请求，因此该配置通常不需要调整。
  - 如果部署的机器 CPU 核数特别少（小于等于 8 ），可以考虑将该配置 (`server.grpc-concurrency`) 设置为 2。
  - 如果机器配置很高，并且TiKV 承担了非常大量的读写请求，观察到Grafana 上的监控 Thread CPU 的 gRPC poll CPU 的数值超过了 server.grpc-concurrency 大小的 80%，那么可以考虑适当调大 `server.grpc-concurrency` 以控制该线程池使用率在 80% 以下（即 Grafana 上的指标低于 80% * `server.grpc-concurrency` 的值)。
- Scheduler 线程池的大小配置(`storage.scheduler-worker-pool-size`) 在 TiKV 检测到机器 CPU 核数大于等于 16 时默认为 8，小于 16 时默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 scheduler 线程池本身不进行任何写操作。
  - 如果检测到有事务冲突，那么它会提前返回冲突结果给客户端。
  - 如果未检测到事务冲突，那么它会把需要写入的 key-value 合并成一条Raft 日志交给Raftstore 线程进行 Raft 日志复制。

  通常来说为了避免过多的线程切换，最好确保 scheduler 线程池的利用率保持在 50% ~ 75% 之间。（如果线程池大小为 8 的话，那么Grafana 上的TiKV-Details.Thread CPU.scheduler worker CPU 应当在 400% ~ 600% 之间较为合理)

- Raftstore 线程池是 TiKV 中最复杂的一个线程池，默认大小 (由 `raftstore.store-pool-size` 控制) 为 2。StoreWriter 线程池的默认大小(由 `raftstore.store-io-pool-size` 控制) 为 0。
  - 当 StoreWriter 线程池大小为 0 时，所有的写请求都会由 Raftstore 线程以 fsync 的方式写入RocksDB。此时建议采取如下调优操作：
    * 将 Raftstore 线程的整体 CPU 使用率控制在 60% 以下。当把Raftstore 线程数设为默认值 2 时，将 Grafana 监控上TiKV-Details、 Thread CPU、 Raft store CPU 面版上的数值控制在 120% 以内。由于存在 I/O 请求，理论上Raftstore 线程的 CPU 使用率总是低于 100%。
    * 不建议为了提升写性能而盲目增大Raftstore 线程池大小，这样可能会适得其反，增加磁盘负担，导致性能变差。
  - 当 StoreWriter 线程池大小不为 0 时，所有写请求都由 StoreWriter 线程以 fsync 的方式写入RocksDB。此时建议采取如下调优操作：
    * 仅在整体 CPU 资源比较充裕的情况下启用 StoreWriter 线程池，并将 StoreWriter 线程和Raftstore 线程的 CPU 使用率控制在 80% 以下。
      与写请求在Raftstore 线程完成的情况相比，理论上 StoreWriter 线程处理写请求能够显著地降低写延迟和读的尾延迟。然而，写入速度变得更快意味着Raft 日志也变得更多，从而导致Raftstore 线程、Apply 线程和 gRPC 线程的 CPU 开销增多。在这种情况下，CPU 资源不足可能会抵消优化效果，反而还可能比原来的写速度更慢，因此若是 CPU 资源不充裕则不建议开启 StoreWriter 线程。由于Raftstore 线程把绝大部分的I/O 请求交给 StoreWriter，因此 Raftstore 线程的 CPU 使用率控制在 80% 以下即可。
    * 大多数情况下将 StoreWriter 线程池的大小设为 1 或 2 即可。这是因为 StoreWriter 线程池的大小会影响Raft 日志数量，所以该值不宜过大。如果 CPU 使用率高于 80%，可以考虑再增加其大小。
    * 注意Raft 日志增多对其他线程池 CPU 开销的影响，必要的时候需要相应地增加Raftstore 线程、Apply 线程和 gRPC 线程的数量。

- UnifyReadPool 负责处理所有的读取请求。默认配置 (`readpool.unified.max-thread-count`) 大小为机器 CPU 数的 80% （如机器为 16 核，则默认线程池大小为 12 ）。

  通常建议根据业务负载特性调整其 CPU 使用率在线程池大小的 60% ~ 90% 之间（如果用户 Grafana 上 TiKV-Details.Thread CPU.Unified read pool CPU 的峰值不超过 800%，那么建议用户将 `readpool.unified.max-thread-count` 设置为 10，过多的线程数会造成更频繁的线程切换，并且抢占其他线程池的资源）。

  TiKV 从 v6.3.0 开始支持根据统一读线程池(UnifyReadPool) 的 CPU 利用率自动动态调整线程池的线程数量，可以通过配置`readpool.unified.auto-adjust-pool-size` 开启此功能。对于重读并且峰值 CPU 利用率超过 80% 的集群，建议开启此配置。

- RocksDB 线程池是RocksDB 进行 Compact 和 Flush 任务的线程池，通常不需要配置。
  - 如果机器 CPU 核数较少，可将 `rocksdb.max-background-jobs` 与 `raftdb.max-background-jobs` 同时设置为 4。
  - 如果遇到了Write Stall，可查看 Grafana 监控上RocksDB-kv 中的Write Stall Reason 有哪些指标不为 0。
    * 如果是由pending compaction bytes 相关原因引起的，可将 `rocksdb.max-sub-compactions` 设置为 2 或者 3（该配置表示单次compaction job 允许使用的子线程数量，TiKV 4.0 版本默认值为 3，3.0 版本默认值为 1 ）。
    * 如果原因是memtable count 相关，建议调大所有列的 `max-write-buffer-number`（默认为 5 ）。
    * 如果原因是level0 file limit 相关，建议调大如下参数为 64 或者更高：

```
rocksdb.defaultcf.level0-slowdown-writes-trigger
rocksdb.writecf.level0-slowdown-writes-trigger
rocksdb.lockcf.level0-slowdown-writes-trigger
rocksdb.defaultcf.level0-stop-writes-trigger
rocksdb.writecf.level0-stop-writes-trigger
rocksdb.lockcf.level0-stop-writes-trigger
```

8.2.0.4 TiKV 内存参数性能调优

本文档用于描述如何根据机器配置情况来调整 TiKV 的参数，使 TiKV 的性能达到最优。你可以在 etc/config-template.toml 找到配置文件模版，参考使用 TiUP 修改配置参数进行操作，部分配置项可以通过在线修改 TiKV 配置方式在线更新。具体配置项的含义可参考TiKV 配置文件描述。

TiKV 最底层使用的是RocksDB 做为持久化存储，所以TiKV 的很多性能相关的参数都是与RocksDB 相关的。TiKV 使用了两个RocksDB 实例，默认RocksDB 实例存储 KV 数据，Raft RocksDB 实例（简称 RaftDB）存储Raft 数据。

TiKV 使用了RocksDB 的 Column Families (CF) 特性。

- 默认RocksDB 实例将 KV 数据存储在内部的 default、write 和 lock 3 个 CF 内。
  - default CF 存储的是真正的数据，与其对应的参数位于 [rocksdb.defaultcf] 项中；
  - write CF 存储的是数据的版本信息(MVCC) 以及索引相关的数据，相关的参数位于[rocksdb.writecf ↩] 项中；
  - lock CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储Raft log。
  - default CF 主要存储的是Raft log，与其对应的参数位于 [raftdb.defaultcf] 项中。

所有的 CF 默认共同使用一个block cache 实例。通过在 [storage.block-cache] 下设置 capacity 参数，你可以配置该block cache 的大小。block cache 越大，能够缓存的热点数据越多，读取数据越容易，同时占用的系统内存也越多。

> **注意：**
>
> 在 TiKV 3.0 之前的版本中，不支持使用 shared block cache，需要为每个 CF 单独配置 block cache。

每个 CF 有各自的 write-buffer，大小通过 write-buffer-size 控制。

8.2.0.4.1 参数说明

```
#### 日志级别，可选值为：trace，debug，warn，error，info，off
log-level = "info"

[server]
#### 监听地址
#### addr = "127.0.0.1:20160"

#### gRPC 线程池大小
#### grpc-concurrency = 4
#### TiKV 每个实例之间的 gRPC 连接数
#### grpc-raft-conn-num = 10

#### TiDB 过来的大部分读请求都会发送到 TiKV 的 Coprocessor 进行处理，该参数用于设置
#### coprocessor 线程的个数，如果业务是读请求比较多，增加 coprocessor 的线程数，但应比系统的
#### CPU 核数小。例如：TiKV 所在的机器有 32 core，在重读的场景下甚至可以将该参数设置为 30。在没有
#### 设置该参数的情况下，TiKV 会自动将该值设置为 CPU 总核数乘以 0.8。
#### end-point-concurrency = 8

#### 可以给 TiKV 实例打标签，用于副本的调度
#### labels = {zone = "cn-east-1", host = "118", disk = "ssd"}

[storage]
#### 数据目录
#### data-dir = "/tmp/tikv/store"

#### 通常情况下使用默认值就可以了。在导数据的情况下建议将该参数设置为 1024000。
#### scheduler-concurrency = 102400
#### 该参数控制写入线程的个数，当写入操作比较频繁的时候，需要把该参数调大。使用 top -H -p tikv-
    ↪ pid
#### 发现名称为 sched-worker-pool 的线程都特别忙，这个时候就需要将 scheduler-worker-pool-size
#### 参数调大，增加写线程的个数。
#### scheduler-worker-pool-size = 4

[storage.block-cache]
##### 是否为 RocksDB 的所有 CF 都创建一个 `shared block cache`。
##### ## RocksDB 使用 block cache 来缓存未压缩的数据块。较大的 block cache 可以加快读取速度。
##### 推荐开启 `shared block cache` 参数。这样只需要设置全部缓存大小，使配置过程更加方便。
##### 在大多数情况下，可以通过 LRU 算法在各 CF 间自动平衡缓存用量。
##### ## `storage.block-cache` 会话中的其余配置仅在开启 `shared block cache` 时起作用。
##### 从 v6.6.0 开始，该选项永远开启且无法关闭。
#### shared = true
##### `shared block cache` 的大小。正常情况下应设置为系统全部内存的 30%-50%。
##### 如果未设置该参数，则由以下字段或其默认值的总和决定。
##### ##    * rocksdb.defaultcf.block-cache-size 或系统全部内存的 25%
```

```
#####   * rocksdb.writecf.block-cache-size 或系统全部内存的 15%
#####   * rocksdb.lockcf.block-cache-size 或系统全部内存的 2%
#####   * raftdb.defaultcf.block-cache-size 或系统全部内存的 2%
##### ## 要在单个物理机上部署多个 TiKV 节点，需要显式配置该参数。
##### 否则，TiKV 中可能会出现 OOM 错误。
#### capacity = "1GB"

[pd]
#### pd 的地址
#### endpoints = ["127.0.0.1:2379","127.0.0.2:2379","127.0.0.3:2379"]

[metric]
#### 将 metrics 推送给 Prometheus pushgateway 的时间间隔
interval = "15s"
#### Prometheus pushgateway 的地址
address = ""
job = "tikv"

[raftstore]
#### Raft RocksDB 目录。默认值是 [storage.data-dir] 的 raft 子目录。
#### 如果机器上有多块磁盘，可以将 Raft RocksDB 的数据放在不同的盘上，提高 TiKV 的性能。
#### raftdb-path = "/tmp/tikv/store/raft"

#### 当 Region 写入的数据量超过该阈值的时候，TiKV 会检查该 Region 是否需要分裂。为了减少检查过程
#### 中扫描数据的成本，导入数据过程中可以将该值设置为 32 MB，正常运行状态下使用默认值即可。
region-split-check-diff = "32MB"

[coprocessor]

##### 当区间为 [a,e) 的 Region 的大小超过 `region_max_size`，TiKV 会尝试分裂该 Region，例如分裂成
    ↪ [a,b)、[b,c)、[c,d)、[d,e) 等区间的 Region 后
##### 这些 Region [a,b), [b,c), [c,d) 的大小为 `region_split_size`（或者稍大于 `region_split_size
    ↪ `）
#### region-max-size = "144MB"
#### region-split-size = "96MB"

[rocksdb]
#### RocksDB 进行后台任务的最大线程数，后台任务包括 compaction 和 flush。具体 RocksDB
    ↪ 为什么需要进行 compaction，
#### 请参考 RocksDB 的相关资料。在写流量比较大的时候（例如导数据），建议开启更多的线程，
#### 但应小于 CPU 的核数。例如在导数据的时候，32 核 CPU 的机器，可以设置成 28。
#### max-background-jobs = 8

#### RocksDB 能够打开的最大文件句柄数。
#### max-open-files = 40960
```

```
#### RocksDB MANIFEST 文件的大小限制.# 更详细的信息请参考：https://github.com/facebook/rocksdb/
    ↪ wiki/MANIFEST
max-manifest-file-size = "20MB"

#### RocksDB write-ahead logs 目录。如果机器上有两块盘，可以将 RocksDB 的数据和 WAL 日志放在
#### 不同的盘上，提高 TiKV 的性能。
#### wal-dir = "/tmp/tikv/store"

#### 下面两个参数用于怎样处理 RocksDB 归档 WAL。
#### 更多详细信息请参考：https://github.com/facebook/rocksdb/wiki/How-to-persist-in-memory-
    ↪ RocksDB-database%3F
#### wal-ttl-seconds = 0
#### wal-size-limit = 0

#### RocksDB WAL 日志的最大总大小，通常情况下使用默认值就可以了。
#### max-total-wal-size = "4GB"

#### 开启 RocksDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[rocksdb.defaultcf]
#### 数据块大小。RocksDB 是按照 block 为单元对数据进行压缩的，同时 block 也是缓存在 block-cache
#### 中的最小单元（类似其他数据库的 page 概念）。
block-size = "64KB"

#### RocksDB 每一层数据的压缩方式，可选的值为：no,snappy,zlib,bzip2,lz4,lz4hc,zstd。
#### no:no:lz4:lz4:lz4:zstd:zstd 表示 level0 和 level1 不压缩，level2 到 level4 采用 lz4 压缩算法
    ↪ ,
#### level5 和 level6 采用 zstd 压缩算法,。
#### no 表示没有压缩，lz4 是速度和压缩比较为中庸的压缩算法，zlib 的压缩比很高，对存储空间比较友
#### 好，但是压缩速度比较慢，压缩的时候需要占用较多的 CPU 资源。不同的机器需要根据 CPU 以及 I/O
    ↪ 资
#### 源情况来配置怎样的压缩方式。例如：如果采用的压缩方式为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量
#### 写入数据的情况下（导数据），发现系统的 I/O 压力很大（使用 iostat 发现 %util 持续 100% 或者使
#### 用 top 命令发现 iowait 特别多），而 CPU 的资源还比较充裕，这个时候可以考虑将 level0 和
#### level1 开启压缩，用 CPU 资源换取 I/O 资源。如果采用的压缩方式
#### 为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量写入数据的情况下，发现系统的 I/O 压力不大，但是 CPU
#### 资源已经吃光了，top -H 发现有大量的 bg 开头的线程（RocksDB 的 compaction 线程）在运行，这
#### 个时候可以考虑用 I/O 资源换取 CPU 资源，将压缩方式改成"no:no:no:lz4:lz4:zstd:zstd"。总之，目
#### 的是为了最大限度地利用系统的现有资源，使 TiKV 的性能在现有的资源情况下充分发挥。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### RocksDB memtable 的大小。
write-buffer-size = "128MB"
```

```
#### 最多允许几个 memtable 存在。写入到 RocksDB 的数据首先会记录到 WAL 日志里面，然后会插入到
#### memtable 里面，当 memtable 的大小到达了 write-buffer-size 限定的大小的时候，当前的
#### memtable 会变成只读的，然后生成一个新的 memtable 接收新的写入。只读的 memtable 会被
#### RocksDB 的 flush 线程（max-background-flushes 参数能够控制 flush 线程的最大个数）
#### flush 到磁盘，成为 level0 的一个 sst 文件。当 flush 线程忙不过来，导致等待 flush 到磁盘的
#### memtable 的数量到达 max-write-buffer-number 限定的个数的时候，RocksDB 会将新的写入
#### stall 住，stall 是 RocksDB 的一种流控机制。在导数据的时候可以将 max-write-buffer-number
#### 的值设置的更大一点，例如 10。
max-write-buffer-number = 5

#### 当 level0 的 sst 文件个数到达 level0-slowdown-writes-trigger 指定的限度的时候，
#### RocksDB 会尝试减慢写入的速度。因为 level0 的 sst 太多会导致 RocksDB 的读放大上升。
#### level0-slowdown-writes-trigger 和 level0-stop-writes-trigger 是 RocksDB 进行流控的
#### 另一个表现。当 level0 的 sst 的文件个数到达 4（默认值），level0 的 sst 文件会和 level1 中
#### 有 overlap 的 sst 文件进行 compaction，缓解读放大的问题。
level0-slowdown-writes-trigger = 20

#### 当 level0 的 sst 文件个数到达 level0-stop-writes-trigger 指定的限度的时候，RocksDB 会
#### stall 住新的写入。
level0-stop-writes-trigger = 36

#### 当 level1 的数据量大小达到 max-bytes-for-level-base 限定的值的时候，会触发 level1 的
#### sst 和 level2 种有 overlap 的 sst 进行 compaction。
#### 黄金定律：max-bytes-for-level-base 的设置的第一参考原则就是保证和 level0 的数据量大致相
#### 等，这样能够减少不必要的 compaction。例如压缩方式为"no:no:lz4:lz4:lz4:lz4:lz4"，那么
#### max-bytes-for-level-base 的值应该是 write-buffer-size 的大小乘以 4，因为 level0 和
#### level1 都没有压缩，而且 level0 触发 compaction 的条件是 sst 的个数到达 4（默认值）。在
#### level0 和 level1 都采取了压缩的情况下，就需要分析下 RocksDB 的日志，看一个 memtable 的压
#### 缩成一个 sst 文件的大小大概是多少，例如 32MB，那么 max-bytes-for-level-base 的建议值就应
#### 该是 32MB * 4 = 128MB。
max-bytes-for-level-base = "512MB"

#### sst 文件的大小。level0 的 sst 文件的大小受 write-buffer-size 和 level0 采用的压缩算法的
#### 影响，target-file-size-base 参数用于控制 level1-level6 单个 sst 文件的大小。
target-file-size-base = "32MB"

[rocksdb.writecf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1
```

```
#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"

[raftdb]
#### RaftDB 能够打开的最大文件句柄数。
#### max-open-files = 40960

#### 开启 RaftDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[raftdb.defaultcf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1

#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"
```

8.2.0.4.2 TiKV 内存使用情况

除了以上列出的 block-cache 以及 write-buffer 会占用系统内存外：

1. 需预留一些内存作为系统的page cache
2. TiKV 在处理大的查询的时候（例如 select * from ...）会读取数据然后在内存中生成对应的数据结构返回给TiDB，这个过程中TiKV 会占用一部分内存

8.2.0.4.3 TiKV 机器配置推荐

1. 生产环境中，不建议将TiKV 部署在 CPU 核数小于 8 或内存低于 32GB 的机器上
2. 如果对写入吞吐要求比较高，建议使用吞吐能力比较好的磁盘
3. 如果对读写的延迟要求非常高，建议使用IOPS 比较高的 SSD 盘

8.2.0.5 Follower Read

当系统中存在读取热点Region 导致leader 资源紧张成为整个系统读取瓶颈时，启用 Follower Read 功能可明显降低 leader 的负担，并且通过在多个 follower 之间均衡负载，显著地提升整体系统的吞吐能力。本文主要介绍 Follower Read 的使用方法与实现机制。

8.2.0.5.1 概述

Follower Read 功能是指在强一致性读的前提下使用Region 的follower 副本来承载数据读取的任务，从而提升TiDB 集群的吞吐能力并降低leader 负载。Follower Read 包含一系列将TiKV 读取负载从Region 的 leader 副本上 offload 到 follower 副本的负载均衡机制。TiKV 的 Follower Read 可以保证数据读取的一致性，可以为用户提供强一致的数据读取能力。

> **注意：**
>
> 为了获得强一致读取的能力，在当前的实现中，follower 节点需要向leader 节点询问当前的执行进度（即 `ReadIndex`），这会产生一次额外的网络请求开销，因此目前 Follower Read 的主要优势是将集群中的读请求与写请求隔离开，并提升整体的读吞吐量。

8.2.0.5.2 使用方式

要开启TiDB 的 Follower Read 功能，将变量 `tidb_replica_read` 的值设置为对应的目标值即可：

```
set [session | global] tidb_replica_read = '<目标值>';
```

作用域：SESSION | GLOBAL

默认值：leader

该变量用于设置期待的数据读取方式。

- 当设置为默认值 `leader` 或者空字符串时，TiDB 会维持原有行为方式，将所有的读取操作都发送给leader 副本处理。
- 当设置为 `follower` 时，TiDB 会选择Region 的 follower 副本完成所有的数据读取操作。
- 当设置为 `leader-and-follower` 时，TiDB 可以选择任意副本来执行读取操作，此时读请求会在leader 和 follower 之间负载均衡。
- 当设置为 `prefer-leader` 时，TiDB 会优先选择leader 副本执行读取操作。当leader 副本的处理速度明显变慢时，例如由于磁盘或网络性能抖动，TiDB 将选择其他可用的 follower 副本来执行读取操作。
- 当设置为 `closest-replicas` 时，TiDB 会优先选择分布在同一可用区的副本执行读取操作，对应的副本可以是leader 或 follower。如果同一可用区内没有副本分布，则会从 leader 执行读取。
- 当设置为 `closest-adaptive` 时：
  - 当一个读请求的预估返回结果大于或等于变量`tidb_adaptive_closest_read_threshold` 的值时，TiDB 会优先选择分布在同一可用区的副本执行读取操作。此时，为了避免读流量在各个可用区分布不均衡，TiDB 会动态检测当前在线的所有TiDB 和 TiKV 的可用区数量分布，在每个可用区中 `closest-adaptive` 配置实际生效的TiDB 节点数总是与包含TiDB 节点最少的可用区中的TiDB 节点数相同，并将其他多出的TiDB 节点自动切换为读取leader 副本。例如，如果TiDB 分布在 3 个可用区，其中A 和 B 两个可用区各包含 3 个 TiDB 节点，C 可用区只包含 2 个 TiDB 节点，那么每个可用区中 `closest-adaptive` 实际生效的TiDB 节点数为 2，A 和 B 可用区中各有 1 个节点自动被切换为读取 leader 副本。

  - 当一个读请求的预估返回结果小于变量tidb_adaptive_closest_read_threshold 的值时，TiDB 会选择 leader 副本执行读取操作。

- 当设置为 learner 时，TiDB 会选择learner 副本执行读取操作。在读取时，如果当前Region 没有 learner 副本，TiDB 会报错。

> **注意：**
>
> 当设置为 closest-replicas 或 closest-adaptive 时，你需要配置集群以确保副本按照指定的设置分布在各个可用区。请参考通过拓扑label 进行副本调度为 PD 配置 location-labels 并为 TiDB 和 TiKV 设置正确的 labels。TiDB 依赖 zone 标签匹配位于同一可用区的TiKV，因此请务必在 PD 的 location-labels 配置中包含 zone 并确保每个TiDB 和 TiKV 节点的 labels 配置中包含 zone。如果是使用 TiDB Operator 部署的集群，请参考数据的高可用进行配置。

8.2.0.5.3 实现机制

在 Follower Read 功能出现之前，TiDB 采用 strong leader 策略将所有的读写操作全部提交到Region 的 leader 节点上完成。虽然TiKV 能够很均匀地将Region 分散到多个物理节点上，但是对于每一个Region 来说，只有leader 副本能够对外提供服务，另外的 follower 除了时刻同步数据准备着 failover 时投票切换成为leader 外，没有办法对 TiDB 的请求提供任何帮助。

为了允许在 TiKV 的 follower 节点进行数据读取，同时又不破坏线性一致性和 Snapshot Isolation 的事务隔离，Region 的 follower 节点需要使用Raft ReadIndex 协议确保当前读请求可以读到当前leader 上已经commit 的最新数据。在TiDB 层面，Follower Read 只需根据负载均衡策略将某个Region 的读取请求发送到 follower 节点。

Follower 强一致读

TiKV follower 节点处理读取请求时，首先使用Raft ReadIndex 协议与Region 当前的leader 进行一次交互，来获取当前 Raft group 最新的commit index。本地apply 到所获取的leader 最新commit index 后，便可以开始正常的读取请求处理流程。

Follower 副本选择策略

由于TiKV 的 Follower Read 不会破坏TiDB 的 Snapshot Isolation 事务隔离级别，因此TiDB 选择 follower 的策略可以采用 round robin 的方式。目前，对于 Coprocessor 请求，Follower Read 负载均衡策略粒度是连接级别的，对于一个 TiDB 的客户端连接在某个具体的Region 上会固定使用同一个 follower，只有在选中的 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。而对于非 Coprocessor 请求（点查等），Follower Read 负载均衡策略粒度是事务级别的，对于一个TiDB 的事务在某个具体的Region 上会固定使用同一个 follower，同样在 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。

8.2.0.6 Region 性能调优

本文介绍了如何通过调整Region 大小等方法对Region 进行性能调优以及如何在大Region 下使用bucket 进行并发查询优化。

8.2.0.6.1 概述

TiKV 自动将底层数据进行分片，所有数据按照 key 的范围划分为若干个Region。当某个Region 的大小超过一定限制后，TiKV 会将它分裂为多个Region。

在大量数据的场景下，可能会出现Region 数量过多，从而带来更多的资源开销和导致性能回退的问题。在固定数据量下，Region 越大，则Region 个数越少。从 v6.1.0 开始，TiDB 支持设置自定义的Region 大小。Region 默认的大小约为 96 MiB，将其调大可以减少 Region 个数。

开启Hibernate Region 或Region Merge 也可以减少过多Region 带来的性能开销。

8.2.0.6.2 使用 region-split-size 调整Region 大小

> **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。

Region 的大小可以通过coprocessor.region-split-size 进行设置。推荐的Region 大小为 96 MiB、128 MiB、256 MiB。region-split-size 越大，性能会越容易发生抖动。不推荐将Region 大小设置超过 1 GiB，强烈建议不超过 10 GiB。如果你使用了TiFlash，则Region 大小不能超过 256 MiB。如果使用 Dumpling 工具，则Region 大小不能超过 1 GiB。Region 调大以后，使用 Dumpling 工具时，需要降低并发，否则TiDB 会有 OOM 的风险。

8.2.0.6.3 使用bucket 增加并发

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

Region 调大以后，为了增加查询并发，应当设置coprocessor.enable-region-bucket 为 true。这个配置会将每个Region 划分为更小的区间bucket，并且以这个更小的区间作为并发查询单位，以提高扫描数据的并发度。bucket 的大小通过coprocessor.region-bucket-size 来控制，默认值为 96MiB。

8.2.0.7 TiFlash 性能调优

本文介绍了TiFlash 性能调优的几种方式，包括机器资源规划和TiDB 参数调优，通过这些方式，TiFlash 性能可以达到最优状态。

8.2.0.7.1 资源规划

对于希望节省机器资源，并且完全没有隔离要求的场景，可以使用TiKV 和 TiFlash 联合部署。建议为TiKV 与 TiFlash 分别留够资源，同时避免共享磁盘。

8.2.0.7.2 TiDB 相关参数调优

本部分介绍如何通过调整TiDB 相关参数来提升TiFlash 性能，具体包括如下几个方面：

- 强制开启 MPP 模式
- 聚合下推 Join 或 Union
- 开启 Distinct 优化
- 使用 ALTER TABLE...COMPACT 整理数据
- 使用 Broadcast Hash Join 代替 Shuffled Hash Join
- 设置更大的执行并发度
- 设置细粒度 Shuffle 参数

强制开启 MPP 模式

MPP 执行计划可以充分利用分布式计算资源，从而显著提高批量数据查询的效率。当查询没有生成 MPP 执行计划的时候，你可以强制开启 MPP：

tidb_enforce_mpp 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。要开启 MPP 模式查询，执行如下命令：

```
set @@tidb_enforce_mpp = ON;
```

在以下示例中，tidb_enforce_mpp 开启前，TiDB 需要从TiKV 读取数据，并在TiDB 侧执行 Join 和 Aggregation。tidb_enforce_mpp 开启后，Join 和 Aggregation 被下推到了TiFlash。需要注意的是，有时候优化器生成的执行计划并不稳定，可能生成 MPP，也可能不生成，这时候可以开启 tidb_enforce_mpp，强制优化器生成 MPP 计划。

MPP 模式开启前：

```
mysql> explain analyze select o_orderpriority, count(*) as order_count from orders where
    '-> o_orderdate >= '1995-01-01' and o_orderdate < date_add('1995-01-01', interval '3' month)
    '-> and exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate <
    '-> l_receiptdate) group by o_orderpriority;
+--
    '-> ------------------------------------+--------------+-----------+-----------+
    '->
| id                                 | estRows      | actRows   | task      | access object
    '->                                          | execution info
    '->
    '-> | operator info
    '->
    '-> | memory     | disk |
+--
    '->                                  +              +           +           +
    '->
      ----------------------------------- -------------- ----------- -----------
```

```
| Projection_9                          | 1.00          | 5          | root      |
    ′→                                                                | time:22.8s, loops:5,
    ′→ Concurrency:OFF
    ′→
    ′→ | tpch_100.orders.o_orderpriority, Column#27
    ′→
    ′→ | 1016 Bytes | N/A  |
| └─HashAgg_10                          | 1.00          | 5          | root      |
    ′→                                                                | time:22.8s, loops:5,
    ′→ partial_worker:{wall_time:22.817403422s, concurrency:5, task_num:5027, tot_wait:1m52
    ′→ .560509138s, tot_exec:1.523754659s, tot_time:1m54.086196522s, max:22.817363394s, p95
    ′→ :22.817363394s}, final_worker:{wall_time:22.817416046s, concurrency:5, task_num:20,
    ′→ tot_wait:1m54.086784741s, tot_exec:155.94µs, tot_time:1m54.086948982s, max:22.817396284s,
    ′→  p95:22.817396284s}
    ′→
    ′→ | group by:tpch_100.orders.o_orderpriority, funcs:count(1)->Column#27, funcs:firstrow(
    ′→ tpch_100.orders.o_orderpriority)->tpch_100.orders.o_orderpriority
    ′→                                    | 511.7 KB    | N/A  |
|   └─IndexHashJoin_18                  | 4429262.13    | 5145138    | root      |
    ′→                                                                | time:22.8s, loops:5028, inner:{
    ′→ total:1m52.6s, concurrency:5, task:227, construct:5.17s, fetch:1m40s, build:1.22s, join
    ′→ :7.49s}
    ′→
    ′→ | semi join, inner:IndexLookUp_15, outer key:tpch_100.orders.o_orderkey, inner key:
    ′→ tpch_100.lineitem.l_orderkey, equal cond:eq(tpch_100.orders.o_orderkey, tpch_100.lineitem
    ′→ .l_orderkey) | 46.8 MB     | N/A  |
|     ├─TableReader_37(Build)           | 5536577.67    | 5610951    | root      |
    ′→                                                                | time:4.89s, loops:5467,
    ′→ cop_task: {num: 2127, max: 408.7ms, min: 277.5µs, avg: 81.6ms, p95: 249.7ms,
    ′→ max_proc_keys: 226272, p95_proc_keys: 220128, tot_proc: 2m50.2s, tot_wait: 40ms, rpc_num:
    ′→  2127, rpc_time: 2m53.6s, copr_cache_hit_ratio: 0.02, distsql_concurrency: 15}
    ′→
    ′→ | data:Selection_36
    ′→
    ′→ | 3.42 MB     | N/A  |
|     │ └─Selection_36                 | 5536577.67    | 5610951    | cop[tikv] |
    ′→                                                                | tikv_task:{proc max:402ms, min
    ′→ :0s, avg: 79.7ms, p80:151ms, p95:245ms, iters:155122, tasks:2127}, scan_detail: {
    ′→ total_process_keys: 149478336, total_process_keys_size: 22705157593, total_keys:
    ′→ 149480413, get_snapshot_time: 187.9ms, rocksdb: {key_skipped_count: 149478336, block: {
    ′→ cache_hit_count: 393690}}}
    ′→
    ′→ | ge(tpch_100.orders.o_orderdate, 1995-01-01 00:00:00.000000), lt(tpch_100.orders.
    ′→ o_orderdate, 1995-04-01 00:00:00.000000)
    ′→                                                                  | N/A         | N/A  |
```

```
|     |   └─TableFullScan_35           | 150000000.00 | 150000000 | cop[tikv] | table:orders
    ′→                                                | tikv_task:{proc max:382ms, min:0s, avg: 74.6
    ′→ ms, p80:141ms, p95:229ms, iters:155122, tasks:2127}
    ′→
    ′→ | keep order:false
    ′→
    ′→ | N/A        | N/A  |
|     └─IndexLookUp_15(Probe)          | 22018496.89  | 14185840  | root      |
    ′→                                                                | time:1m33.4s, loops:14204,
    ′→ index_task: {total_time: 1m10.8s, fetch_handle: 1m10.7s, build: 3.9ms, wait: 13.3ms},
    ′→ table_task: {total_time: 1m44.4s, num: 1989, concurrency: 5}
    ′→
    ′→ |
    ′→
    ′→ | 17.9 KB    | N/A  |
|       ├─IndexRangeScan_12(Build)      | 27523121.11  | 22441356  | cop[tikv] | table:lineitem,
    ′→ index:PRIMARY(L_ORDERKEY, L_LINENUMBER) | time:1m8.8s, loops:23635, cop_task: {num:
    ′→ 12537, max: 179.3ms, min: 331.4µs, avg: 22.7ms, p95: 66.9ms, max_proc_keys: 13755,
    ′→ p95_proc_keys: 5088, tot_proc: 3m44s, tot_wait: 129ms, rpc_num: 12537, rpc_time: 4m44s,
    ′→ copr_cache_hit_ratio: 0.06, distsql_concurrency: 15}, tikv_task:{proc max:170ms, min:0s,
    ′→ avg: 18.8ms, p80:29ms, p95:62ms, iters:68823, tasks:12537}, scan_detail: {
    ′→ total_process_keys: 21985403, total_process_keys_size: 1165226359, total_keys: 27492546,
    ′→ get_snapshot_time: 563.1ms, rocksdb: {key_skipped_count: 21985403, block: {
    ′→ cache_hit_count: 44004354}}}     | range: decided by [eq(tpch_100.lineitem.l_orderkey,
    ′→ tpch_100.orders.o_orderkey)], keep order:false
    ′→                                                                                | N/A
    ′→         | N/A  |
|       └─Selection_14(Probe)          | 22018496.89  | 14185840  | cop[tikv] |
    ′→                                                                | time:1m34.8s, loops:16882,
    ′→ cop_task: {num: 12057, max: 240.4ms, min: 328.5µs, avg: 20.4ms, p95: 73.1ms,
    ′→ max_proc_keys: 15005, p95_proc_keys: 6118, tot_proc: 3m39.8s, tot_wait: 300ms, rpc_num:
    ′→ 12057, rpc_time: 4m5.7s, copr_cache_hit_ratio: 0.28, distsql_concurrency: 15}, tikv_task
    ′→ :{proc max:230ms, min:0s, avg: 27.5ms, p80:45ms, p95:82ms, iters:74165, tasks:12057},
    ′→ scan_detail: {total_process_keys: 19916937, total_process_keys_size: 3957799619,
    ′→ total_keys: 24053143, get_snapshot_time: 1.07s, rocksdb: {key_skipped_count: 19254567,
    ′→ block: {cache_hit_count: 37816908}}}  | lt(tpch_100.lineitem.l_commitdate, tpch_100.
    ′→ lineitem.l_receiptdate)
    ′→
    ′→ | N/A        | N/A  |
|         └─TableRowIDScan_13          | 27523121.11  | 22441356  | cop[tikv] | table:lineitem
    ′→                                                | tikv_task:{proc max:230ms, min:0s, avg: 27.2ms
    ′→ , p80:45ms, p95:81ms, iters:74165, tasks:12057}
    ′→
    ′→ | keep order:false
    ′→
```

```
    '→ | N/A         | N/A   |
+--
    '→ ------------------------------------+-------------+----------+----------+-------------------------------------------------------------------------
    '→
10 rows in set (22.82 sec)
```

开启 MPP 模式：

```
mysql> set @@tidb_enforce_mpp = ON;
Query OK, 0 rows affected (0.00 sec)
```

MPP 模式开启后：

```
mysql> explain analyze select o_orderpriority, count(*) as order_count from orders where
    '→ o_orderdate >= '1995-01-01' and o_orderdate < date_add('1995-01-01', interval '3' month)
    '→ and exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate <
    '→ l_receiptdate) group by o_orderpriority;
+--
    '→ ------------------------------------------------+-------------+-----------+---------------+-------------------------------------------------
    '→
| id                                                | estRows       | actRows    | task          |
    '→ access object  | execution info
    '→
    '→ | operator info
    '→
    '→ | memory | disk |
+--
    '→                                                +             +            +               +
    '→ ------------------------------------------------+-------------+-----------+---------------+-------------------------------------------------
    '→
| TableReader_59                                    | 1.00          | 5          | root          |
    '→                | time:5.97s, loops:2, cop_task: {num: 8, max: 0s, min: 0s, avg: 0s, p95:
    '→ 0s, copr_cache_hit_ratio: 0.00}
    '→
    '→ | data:ExchangeSender_58
    '→
    '→ | N/A     | N/A   |
| └─ExchangeSender_58                               | 1.00          | 5          | mpp[tiflash] |
    '→                | tiflash_task:{proc max:5.97s, min:5.97s, avg: 5.97s, p80:5.97s, p95:5.97
    '→ s, iters:5, tasks:3, threads:60}
    '→
    '→ | ExchangeType: PassThrough
    '→
    '→ | N/A     | N/A   |
|   └─Projection_10                                 | 1.00          | 5          | mpp[tiflash] |
    '→                | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    '→ s, iters:5, tasks:3, threads:60}
```

```
    '→
    '→ | tpch_100.orders.o_orderpriority, Column#27
    '→
    '→ | N/A    | N/A  |
|     └─Projection_54                          | 1.00         | 5          | mpp[tiflash] |
    '→               | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    '→ s, iters:5, tasks:3, threads:60}
    '→
    '→ | Column#27, tpch_100.orders.o_orderpriority
    '→
    '→ | N/A    | N/A  |
|       └─HashAgg_55                           | 1.00         | 5          | mpp[tiflash] |
    '→               | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    '→ s, iters:5, tasks:3, threads:60}
    '→
    '→ | group by:tpch_100.orders.o_orderpriority, funcs:sum(Column#32)->Column#27, funcs:
    '→ firstrow(tpch_100.orders.o_orderpriority)->tpch_100.orders.o_orderpriority, stream_count:
    '→  20 | N/A    | N/A  |
|         └─ExchangeReceiver_57                | 1.00         | 15         | mpp[tiflash] |
    '→               | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    '→ s, iters:5, tasks:3, threads:60}
    '→
    '→ | stream_count: 20
    '→
    '→ | N/A    | N/A  |
|           └─ExchangeSender_56                | 1.00         | 15         | mpp[tiflash] |
    '→               | tiflash_task:{proc max:5.96s, min:0s, avg: 1.99s, p80:5.96s, p95:5.96s,
    '→ iters:768, tasks:3, threads:60}
    '→
    '→ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.orders.o_orderpriority, collate
    '→ : utf8mb4_bin], stream_count: 20                                                        |
    '→ N/A    | N/A  |
|             └─HashAgg_52                     | 1.00         | 15         | mpp[tiflash] |
    '→               | tiflash_task:{proc max:5.96s, min:0s, avg: 1.99s, p80:5.96s, p95:5.96s,
    '→ iters:768, tasks:3, threads:60}
    '→
    '→ | group by:tpch_100.orders.o_orderpriority, funcs:count(1)->Column#32
    '→
    '→ | N/A    | N/A  |
|               └─HashJoin_41                  | 4429262.13   | 5145138    | mpp[tiflash] |
    '→               | tiflash_task:{proc max:5.95s, min:0s, avg: 1.98s, p80:5.95s, p95:5.95s,
    '→ iters:118, tasks:3, threads:60}
    '→
    '→ | semi join, equal:[eq(tpch_100.orders.o_orderkey, tpch_100.lineitem.l_orderkey)],
    '→ stream_count: 20
```

```
    ′→                                                                                                  | N/A     | N/
    ′→ A  |
|                 ├─ExchangeReceiver_26(Build)      | 480030321.60 | 379356474 | mpp[tiflash] |
     ′→                | tiflash_task:{proc max:3.26s, min:0s, avg: 1.09s, p80:3.26s, p95:3.26s,
    ′→ iters:42469, tasks:3, threads:60}
    ′→
    ′→ | stream_count: 20
    ′→
    ′→ | N/A     | N/A  |
|                 │ └─ExchangeSender_25             | 480030321.60 | 379356474 | mpp[tiflash] |
     ′→                | tiflash_task:{proc max:3.37s, min:0s, avg: 1.12s, p80:3.37s, p95:3.37s,
    ′→ iters:9292, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.lineitem.l_orderkey, collate:
    ′→ binary], stream_count: 20                                                                        |
    ′→  N/A     | N/A  |
|                 │   └─Projection_60               | 480030321.60 | 379356474 | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:819.1ms, min:0s, avg: 273ms, p80:819.1ms, p95
    ′→ :819.1ms, iters:9292, tasks:3, threads:60}
    ′→
    ′→ | tpch_100.lineitem.l_orderkey
    ′→
    ′→ | N/A     | N/A  |
|                 │     └─Selection_24              | 480030321.60 | 379356474 | mpp[tiflash] |
     ′→                | tiflash_task:{proc max:817.1ms, min:0s, avg: 272.4ms, p80:817.1ms, p95
    ′→ :817.1ms, iters:9292, tasks:3, threads:60}
    ′→
    ′→ | lt(tpch_100.lineitem.l_commitdate, tpch_100.lineitem.l_receiptdate)
    ′→
    ′→ | N/A     | N/A  |
|                 │       └─TableFullScan_23        | 600037902.00 | 600037902 | mpp[tiflash] |
    ′→ table:lineitem | tiflash_task:{proc max:328.1ms, min:0s, avg: 109.4ms, p80:328.1ms, p95
    ′→ :328.1ms, iters:9292, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs
    ′→ :73475, total_skipped_packs:13452, total_scanned_rows:600230532, total_skipped_rows
    ′→ :109887612, total_rs_index_load_time: 35ms, total_read_time: 34065ms},
    ′→ total_create_snapshot_time: 0ms} | keep order:false
    ′→
    ′→ | N/A     | N/A  |
|                 └─ExchangeReceiver_22(Probe)      | 5536577.67    | 5610951    | mpp[tiflash] |
     ′→                | tiflash_task:{proc max:2.5s, min:0s, avg: 833.7ms, p80:2.5s, p95:2.5s,
    ′→ iters:669, tasks:3, threads:60}
    ′→
    ′→ |
    ′→
    ′→ | N/A     | N/A  |
```

```
|                   └─ExchangeSender_21             | 5536577.67    | 5610951    | mpp[tiflash] |
    ′→                  | tiflash_task:{proc max:5.93s, min:0s, avg: 1.98s, p80:5.93s, p95:5.93s,
    ′→ iters:2336, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.orders.o_orderkey, collate:
    ′→ binary]
    ′→                                                                                    | N/A
    ′→    | N/A  |
|                     └─Selection_20                | 5536577.67    | 5610951    | mpp[tiflash] |
    ′→                  | tiflash_task:{proc max:325.5ms, min:0s, avg: 108.5ms, p80:325.5ms, p95
    ′→ :325.5ms, iters:2336, tasks:3, threads:60}
    ′→
    ′→ | ge(tpch_100.orders.o_orderdate, 1995-01-01 00:00:00.000000), lt(tpch_100.orders.
    ′→ o_orderdate, 1995-04-01 00:00:00.000000)
    ′→                                                          | N/A     | N/A  |
|                       └─TableFullScan_19          | 150000000.00 | 150000000 | mpp[tiflash] |
    ′→ table:orders    | tiflash_task:{proc max:238.5ms, min:0s, avg: 79.5ms, p80:238.5ms, p95
    ′→ :238.5ms, iters:2336, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs
    ′→ :18541, total_skipped_packs:186, total_scanned_rows:150000611, total_skipped_rows
    ′→ :1513328, total_rs_index_load_time: 75ms, total_read_time: 11454ms},
    ′→ total_create_snapshot_time: 0ms}       | keep order:false
    ′→
    ′→ | N/A     | N/A  |
+--
    ′→ ----------------------------------------------------+--------------+-----------+---------------+-----------------
    ′→
18 rows in set (6.00 sec)
```

聚合下推 Join 或 Union

将聚合操作下推到 Join 或 Union 之前执行，有可能能显著减少 Join 或 Union 需要处理的数据量，从而提升性能。

tidb_opt_agg_push_down 变量用来设置优化器是否执行聚合函数下推到Join 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 ON。

```
set @@tidb_opt_agg_push_down = ON;
```

在以下示例中，tidb_opt_agg_push_down 开启前，HashAgg_58 在 HashJoin_41 之后执行。tidb_opt_agg_push_down ′→ 开启后，新生成的 HashAgg_21 和 HashAgg_32 在 HashJoin_76 之前执行，显著减少了 Join 需要处理的数据量。

tidb_opt_agg_push_down 开启前:

```
mysql> explain analyze select count(*) from t1 join t2 where t1.a = t2.b group by t1.a;
+--
    ′→  ---------------------------------------+-------------+----------+-------------+---------------+------
    ′→
```

```
| id                                      | estRows       | actRows    | task          | access
    ′→ object | execution info
    ′→
    ′→ | operator info
    ′→                                                                                 |
    ′→ memory | disk |
+--
    ′→  ----------------------------------------+--------------+-----------+--------------+-------- -------+------
    ′→
| TableReader_66                            | 100450000.00 | 20         | root          |
    ′→               | time:2.13s, loops:2, cop_task: {num: 21, max: 0s, min: 0s, avg: 0s, p95:
    ′→ 0s, copr_cache_hit_ratio: 0.00}
    ′→
    ′→ | data:ExchangeSender_65
    ′→                                                                              | N/A     | N/
    ′→ A  |
| └─ExchangeSender_65                       | 100450000.00 | 20         | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:2.13s, min:2.08s, avg: 2.11s, p80:2.13s, p95:2.13s
    ′→ , iters:768, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: PassThrough
    ′→                                                                          | N/A     | N/A
    ′→ |
|   └─Projection_60                         | 100450000.00 | 20         | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:2.13s, min:2.08s, avg: 2.11s, p80:2.13s, p95:2.13s
    ′→ , iters:768, tasks:3, threads:60}
    ′→
    ′→ | Column#43
    ′→
    ′→ | N/A     | N/A  |
|     └─HashAgg_58                          | 100450000.00 | 20         | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:2.13s, min:2.08s, avg: 2.11s, p80:2.13s, p95:2.13s
    ′→ , iters:768, tasks:3, threads:60}
    ′→
    ′→ | group by:test.t1.a, funcs:count(1)->Column#43
    ′→                                                          | N/A     | N/A  |
|       └─HashJoin_41                       | 600000000.00 | 600000000 | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:1.77s, min:1.67s, avg: 1.73s, p80:1.77s, p95:1.77s
    ′→ , iters:10955, tasks:3, threads:60}
    ′→
    ′→ | inner join, equal:[eq(test.t1.a, test.t2.b)], stream_count: 20
    ′→                                      | N/A     | N/A  |
|         ├─ExchangeReceiver_22(Build)      | 100450000.00 | 99980000  | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:171ms, min:127ms, avg: 154.5ms, p80:171ms, p95:171
    ′→ ms, iters:10440, tasks:3, threads:60}
```

```
    ′→
    ′→ | stream_count: 20
    ′→                                                                                                  | N/A
    ′→    | N/A  |
|         | └─ExchangeSender_21             | 100450000.00 | 99980000  | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:498.7ms, min:0s, avg: 166.2ms, p80:498.7ms, p95
    ′→ :498.7ms, iters:1653, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: HashPartition, Hash Cols: [name: test.t1.a, collate: binary],
    ′→ stream_count: 20        | N/A     | N/A  |
|         |   └─Selection_20                | 100450000.00 | 99980000  | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:83.7ms, min:0s, avg: 27.9ms, p80:83.7ms, p95:83.7
    ′→ ms, iters:1653, tasks:3, threads:60}
    ′→
    ′→ | not(isnull(test.t1.a))
    ′→                                                                                                | N/A     | N/
    ′→ A  |
|         |     └─TableFullScan_19          | 100450000.00 | 99980000  | mpp[tiflash] | table:t1
     ′→      | tiflash_task:{proc max:41.7ms, min:0s, avg: 13.9ms, p80:41.7ms, p95:41.7ms, iters
    ′→ :1653, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:12336,
    ′→ total_skipped_packs:5759, total_scanned_rows:100083566, total_skipped_rows:46786721,
    ′→ total_rs_index_load_time: 5ms, total_read_time: 1105ms}, total_create_snapshot_time: 0ms}
    ′→  | keep order:false
    ′→                                                                                                  | N/A
    ′→    | N/A  |
|         └─ExchangeReceiver_26(Probe)      | 600000000.00 | 600000000 | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:373ms, min:71ms, avg: 253.2ms, p80:373ms, p95:373
    ′→ ms, iters:28135, tasks:3, threads:60}
    ′→
    ′→ |
    ′→
    ′→ | N/A     | N/A  |
|           └─ExchangeSender_25             | 600000000.00 | 600000000 | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:2.07s, min:0s, avg: 689.6ms, p80:2.07s, p95:2.07s,
    ′→  iters:9388, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: HashPartition, Hash Cols: [name: test.t2.b, collate: binary]
    ′→                           | N/A     | N/A  |
|             └─Selection_24                | 600000000.00 | 600000000 | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:146.9ms, min:0s, avg: 49ms, p80:146.9ms, p95:146.9
    ′→ ms, iters:9388, tasks:3, threads:60}
    ′→
    ′→ | not(isnull(test.t2.b))
    ′→                                                                                                | N/A     | N/
    ′→ A  |
```

```
|               └─ TableFullScan_23        | 600000000.00 | 600000000 | mpp[tiflash] | table:t2
    ′→      | tiflash_task:{proc max:63.8ms, min:0s, avg: 21.3ms, p80:63.8ms, p95:63.8ms, iters
    ′→ :9388, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    ′→ total_skipped_packs:439, total_scanned_rows:600007458, total_skipped_rows:3553131,
    ′→ total_rs_index_load_time: 12ms, total_read_time: 270ms}, total_create_snapshot_time: 0ms}
    ′→    | keep order:false
    ′→                                                                                | N/A
    ′→    | N/A  |
+--
    ′→  ---------------------------------------+-------------+----------+-------------+----------------+------
    ′→
13 rows in set (2.15 sec)
```

开启 tidb_opt_agg_push_down：

```
mysql> set @@tidb_opt_agg_push_down = ON;
Query OK, 0 rows affected (0.00 sec)
```

开启 tidb_opt_agg_push_down 后：

```
mysql> explain analyze select count(*) from t1 join t2 where t1.a = t2.b group by t1.a;
+--
    ′→ ------------------------------------------------------+--------------+-----------+                +----------
    ′→
| id                                                  | estRows       | actRows    | task
    ′→ | access object | execution info
    ′→
    ′→ | operator info
    ′→
    ′→ | memory | disk |
+--  ------------------------------------------------------ --------------- ------------                +----------
    ′→                                                     +               +           +
    ′→
| TableReader_85                                      | 20.00         | 20         | root
    ′→ |               | time:432.4ms, loops:2, cop_task: {num: 19, max: 0s, min: 0s, avg: 0s,
    ′→ p95: 0s, copr_cache_hit_ratio: 0.00}
    ′→
    ′→ | data:ExchangeSender_84
    ′→
    ′→ | N/A    | N/A  |
| └─ExchangeSender_84                                  | 20.00         | 20         | mpp[tiflash]
     ′→ |               | tiflash_task:{proc max:431.1ms, min:429.3ms, avg: 430.4ms, p80:431.1ms
    ′→ , p95:431.1ms, iters:16, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: PassThrough
    ′→
    ′→ | N/A    | N/A  |
```

```
|   └─Projection_83                                  | 20.00        | 20        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:430.1ms, min:429.3ms, avg: 429.7ms, p80:430.1ms
   ′→ , p95:430.1ms, iters:16, tasks:3, threads:60}
   ′→
   ′→ | Column#43
   ′→
   ′→ | N/A    | N/A  |
|     └─HashAgg_82                                  | 20.00        | 20        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:430.1ms, min:429.3ms, avg: 429.7ms, p80:430.1ms
   ′→ , p95:430.1ms, iters:16, tasks:3, threads:60}
   ′→
   ′→ | group by:test.t1.a, funcs:count(Column#44)->Column#43, stream_count: 20
   ′→                                                          | N/A    | N/A  |
|       └─ExchangeReceiver_78                       | 20.00        | 20        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:430.1ms, min:429.3ms, avg: 429.7ms, p80:430.1ms
   ′→ , p95:430.1ms, iters:16, tasks:3, threads:60}
   ′→
   ′→ | stream_count: 20
   ′→
   ′→ | N/A    | N/A  |
|         └─ExchangeSender_77                       | 20.00        | 20        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:427ms, min:0s, avg: 142.3ms, p80:427ms, p95:427
   ′→ ms, iters:60, tasks:3, threads:60}
   ′→
   ′→ | ExchangeType: HashPartition, Hash Cols: [name: test.t1.a, collate: binary],
   ′→ stream_count: 20                                  | N/A    | N/A  |
|           └─HashJoin_76                           | 20.00        | 20        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:426ms, min:0s, avg: 142ms, p80:426ms, p95:426ms
   ′→ , iters:60, tasks:3, threads:60}
   ′→
   ′→ | inner join, equal:[eq(test.t1.a, test.t2.b)]
   ′→                                                                                  | N/A
   ′→    | N/A  |
|             ├─ExchangeReceiver_36(Build)          | 20.00        | 60        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:374ms, min:0s, avg: 124.7ms, p80:374ms, p95:374
   ′→ ms, iters:46, tasks:3, threads:60}
   ′→
   ′→ |
   ′→
   ′→ | N/A    | N/A  |
|             │ └─ExchangeSender_35                 | 20.00        | 20        | mpp[tiflash]
    ′→ |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
   ′→ :373.7ms, iters:16, tasks:3, threads:60}
   ′→
   ′→ | ExchangeType: Broadcast
```

```
    ’→
    ’→ | N/A    | N/A  |
|             |   └─Projection_31                       | 20.00        | 20        | mpp[tiflash]
    ’→ |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ’→ :373.7ms, iters:16, tasks:3, threads:60}
    ’→
    ’→ | Column#44, test.t2.b
    ’→
    ’→ | N/A    | N/A  |
|             |     └─HashAgg_32                        | 20.00        | 20        | mpp[tiflash]
    ’→ |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ’→ :373.7ms, iters:16, tasks:3, threads:60}
    ’→
    ’→ | group by:test.t2.b, funcs:sum(Column#45)->Column#44, funcs:firstrow(test.t2.b)->test.t2
    ’→ .b, stream_count: 20                   | N/A    | N/A  |
|             |       └─ExchangeReceiver_34             | 20.00        | 60        | mpp[tiflash]
    ’→ |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ’→ :373.7ms, iters:16, tasks:3, threads:60}
    ’→
    ’→ | stream_count: 20
    ’→
    ’→ | N/A    | N/A  |
|             |         └─ExchangeSender_33             | 20.00        | 60        | mpp[tiflash]
    ’→ |               | tiflash_task:{proc max:368.3ms, min:0s, avg: 122.8ms, p80:368.3ms, p95
    ’→ :368.3ms, iters:3, tasks:3, threads:60}
    ’→
    ’→ | ExchangeType: HashPartition, Hash Cols: [name: test.t2.b, collate: binary],
    ’→ stream_count: 20                                | N/A    | N/A  |
|             |           └─HashAgg_21                  | 20.00        | 60        | mpp[tiflash]
    ’→ |               | tiflash_task:{proc max:367.3ms, min:0s, avg: 122.4ms, p80:367.3ms, p95
    ’→ :367.3ms, iters:3, tasks:3, threads:60}
    ’→
    ’→ | group by:test.t2.b, funcs:count(1)->Column#45
    ’→                                                                                | N/A
    ’→ | N/A  |
|             |             └─Selection_30              | 600000000.00 | 600000000 | mpp[tiflash]
    ’→ |               | tiflash_task:{proc max:77.2ms, min:0s, avg: 25.7ms, p80:77.2ms, p95
    ’→ :77.2ms, iters:9389, tasks:3, threads:60}
    ’→
    ’→ | not(isnull(test.t2.b))
    ’→
    ’→ | N/A    | N/A  |
|             |               └─TableFullScan_29        | 600000000.00 | 600000000 | mpp[tiflash]
    ’→ | table:t2      | tiflash_task:{proc max:37.2ms, min:0s, avg: 12.4ms, p80:37.2ms, p95
    ’→ :37.2ms, iters:9389, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs
```

```
    '→ :73834, total_skipped_packs:412, total_scanned_rows:600002896, total_skipped_rows
    '→ :3342145, total_rs_index_load_time: 0ms, total_read_time: 45ms},
    '→ total_create_snapshot_time: 0ms} | keep order:false
    '→
    '→ | N/A    | N/A  |
|              └─Selection_19(Probe)                    | 100450000.00 | 99980000  | mpp[tiflash]
    '→  |               | tiflash_task:{proc max:30ms, min:0s, avg: 10ms, p80:30ms, p95:30ms,
    '→ iters:1645, tasks:3, threads:60}
    '→
    '→ | not(isnull(test.t1.a))
    '→
    '→ | N/A    | N/A  |
|                └─TableFullScan_18                     | 100450000.00 | 99980000  | mpp[tiflash]
    '→  | table:t1      | tiflash_task:{proc max:22ms, min:0s, avg: 7.34ms, p80:22ms, p95:22ms,
    '→ iters:1645, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:12326,
    '→ total_skipped_packs:4762, total_scanned_rows:100045535, total_skipped_rows:38651216,
    '→ total_rs_index_load_time: 4ms, total_read_time: 355ms}, total_create_snapshot_time: 0ms}
    '→    | keep order:false
    '→
    '→ | N/A    | N/A  |
+--
    '→ ----------------------------------------------------+--------------+------------+                +----------
    '→
18 rows in set (0.46 sec)
```

开启 Distinct 优化

TiFlash 暂时还不支持部分可接受 Distinct 列的聚合函数，比如 Sum 。默认情况下，整个聚合函数运算都会在 TiDB 端执行。通过开启 Distinct 优化，部分操作可以下推到TiFlash，从而提升查询性能：

tidb_opt_distinct_agg_push_down 变量用来设置优化器是否执行带有 Distinct 的聚合函数（比如 select '→ sum(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 ON。

```
set @@tidb_opt_distinct_agg_push_down = ON;
```

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从TiFlash 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TiFlash，在 HashAgg_6 里新增里一个 group by 列 test.t.a。这边查询结果里面的两个 warnings 是警告聚合函数不能完全下推。

tidb_opt_distinct_agg_push_down 开启前：

```
mysql> explain analyze select count(distinct a) from test.t;
+--
    '→  -------------------------+-------------+-----------+-------------+---------------+-------------------
    '→
| id                           | estRows       | actRows    | task          | access object |
    '→ execution info
```

```
    '→
    '→ | operator info                                          | memory    | disk    |
+--
    '→  -------------------------+-------------+-----------+-------------+--------------+-------------------
    '→
| HashAgg_6                    | 1.00        | 1         | root        |              | time:2
    '→ m23.2s, loops:2
    '→
    '→ | funcs:sum(distinct Column#23)->Column#22               | 41.3 KB   | 0 Bytes |
| └─Projection_16              | 600000000.00 | 600000000 | root        |              | time
    '→ :2.51s, loops:586548, Concurrency:5
    '→
    '→ | cast(test.t.a, decimal(10,0) BINARY)->Column#23        | 243.2 KB | N/A      |
|   └─TableReader_11           | 600000000.00 | 600000000 | root        |              | time
    '→ :1.6s, loops:586548, cop_task: {num: 1173, max: 256.2ms, min: 25.1ms, avg: 46.9ms, p95:
    '→ 63.5ms, rpc_num: 1173, rpc_time: 55s, copr_cache_hit_ratio: 0.00, distsql_concurrency:
    '→ 15}
    '→
    '→ | data:TableFullScan_10                                 | 70.2 MB   | N/A      |
|     └─TableFullScan_10       | 600000000.00 | 600000000 | cop[tiflash] | table:t      |
    '→ tiflash_task:{proc max:9ms, min:531µs, avg: 4.49ms, p80:5.55ms, p95:6.74ms, iters:9390,
    '→ tasks:1173, threads:1173}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    '→ total_skipped_packs:1231, total_scanned_rows:600010914, total_skipped_rows:9988978,
    '→ total_rs_index_load_time: 0ms, total_read_time: 16ms}, total_create_snapshot_time: 0ms}
    '→ | keep order:false                                      | N/A       | N/A      |
+--
    '→  -------------------------+-------------+-----------+-------------+--------------+-------------------
    '→
4 rows in set, 2 warnings (2 min 23.21 sec)
```

开启 tidb_opt_distinct_agg_push_down：

```
mysql> set @@tidb_opt_distinct_agg_push_down = ON;
Query OK, 0 rows affected (0.00 sec)
```

tidb_opt_distinct_agg_push_down 开启后：

```
mysql> explain analyze select count(distinct a) from test.t;
+--
    '→  -------------------------+-------------+-----------+------------------+--------------+-------------
    '→
| id                           | estRows      | actRows   | task             | access object |
    '→ execution info
    '→
    '→ | operator info                                         | memory    | disk    |
+--
    '→  -------------------------+-------------+-----------+------------------+--------------+-------------
```

```
    '→
| HashAgg_10                    | 1.00          | 1          | root               |                |
    '→ time:233.8ms, loops:2
    '→
    '→ | funcs:sum(distinct Column#23)->Column#22                | 2.42 KB    | 0 Bytes |
| └─Projection_12               | 1.00          | 3          | root               |                |
    '→ time:233.7ms, loops:2, Concurrency:OFF
    '→
    '→ | cast(test.c.a, decimal(10,0) BINARY)->Column#23         | 380 Bytes | N/A      |
|   └─TableReader_11            | 1.00          | 3          | root               |                |
    '→ time:233.7ms, loops:2, cop_task: {num: 6, max: 0s, min: 0s, avg: 0s, p95: 0s,
    '→ copr_cache_hit_ratio: 0.00, distsql_concurrency: 15}
    '→
    '→ | data:HashAgg_6                                         | 100 Bytes | N/A      |
|     └─HashAgg_6               | 1.00          | 3          | batchCop[tiflash] |                |
    '→ tiflash_task:{proc max:225.8ms, min:210.7ms, avg: 216.9ms, p80:225.8ms, p95:225.8ms,
    '→ iters:3, tasks:3, threads:60}
    '→
    '→ | group by:test.t.a,                                     | N/A        | N/A      |
|       └─TableFullScan_9       | 600000000.00 | 600000000 | batchCop[tiflash] | table:t        |
    '→ tiflash_task:{proc max:50.3ms, min:33.7ms, avg: 44.6ms, p80:50.3ms, p95:50.3ms, iters
    '→ :9387, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73833,
    '→ total_skipped_packs:475, total_scanned_rows:600000000, total_skipped_rows:3852098,
    '→ total_rs_index_load_time: 0ms, total_read_time: 84ms}, total_create_snapshot_time: 0ms} |
    '→  keep order:false                                       | N/A        | N/A      |
+--
    '→  ---------------------------+--------------+-----------+-------------------+---------------+--------------
    '→
5 rows in set, 2 warnings (0.24 sec)
```

使用 ALTER TABLE...COMPACT 整理数据

ALTER TABLE ... COMPACT 可以触发TiFlash 节点对某个表或者某个分区进行数据整理。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。

```
ALTER TABLE employees COMPACT TIFLASH REPLICA;
```

```
ALTER TABLE employees COMPACT PARTITION pNorth, pEast TIFLASH REPLICA;
```

使用 Broadcast Hash Join 代替 Shuffled Hash Join

对于有小表的 Join 算子，Broadcast Hash Join 可以避免大表的网络传输，从而提升计算性能。

- tidb_broadcast_join_threshold_size，单位为bytes。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。

```
set @@tidb_broadcast_join_threshold_size = 2000000;
```

- tidb_broadcast_join_threshold_count，单位为行数。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。

```
set @@tidb_broadcast_join_threshold_count = 100000;
```

在以下示例中，tidb_broadcast_join_threshold_size 设置前，ExchangeSender_29 的 ExchangeType 是 HashPartition。tidb_broadcast_join_threshold_size 设置后，ExchangeSender_29 的 ExchangeType 改变为 Broadcast。

tidb_broadcast_join_threshold_size 设置前：

```
mysql> explain analyze select max(l_shipdate), max(l_commitdate), max(l_receiptdate) from
    '-> supplier,lineitem where s_suppkey = l_suppkey;
+--
    '->  ----------------------------------------+-------------+-----------+-------------+-----------------+-----
    '->
| id                                    | estRows      | actRows    | task          | access
    '-> object  | execution info
    '->
    '-> | operator info
    '->
    '-> | memory  | disk |
+--
    '->  ----------------------------------------+-------------+-----------+-------------+-----------------+-----
    '->
| HashAgg_45                            | 1.00         | 1          | root          |
    '->                | time:3.8s, loops:2, partial_worker:{wall_time:3.798284809s, concurrency
    '-> :5, task_num:1, tot_wait:18.99079929s, tot_exec:8.193µs, tot_time:18.990819019s, max
    '-> :3.798181723s, p95:3.798181723s}, final_worker:{wall_time:0s, concurrency:5, task_num:1,
    '-> tot_wait:18.991291379s, tot_exec:197.45µs, tot_time:18.991494363s, max:3.798334972s, p95
    '-> :3.798334972s}   | funcs:max(Column#28)->Column#25, funcs:max(Column#29)->Column#26,
    '-> funcs:max(Column#30)->Column#27
    '->                                                                    | 17.3 KB | N/A   |
| └─TableReader_47                       | 1.00         | 3          | root          |
    '->                | time:3.8s, loops:2, cop_task: {num: 6, max: 0s, min: 0s, avg: 0s, p95: 0
    '-> s, copr_cache_hit_ratio: 0.00}
    '->
    '-> | data:ExchangeSender_46
    '->
    '-> | N/A     | N/A  |
|   └─ExchangeSender_46                  | 1.00         | 3          | mpp[tiflash] |
    '->                | tiflash_task:{proc max:3.79s, min:3.78s, avg: 3.78s, p80:3.79s, p95:3.79
```

```
    ′→ s, iters:3, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: PassThrough
    ′→
    ′→ | N/A     | N/A  |
|     └─HashAgg_13                        | 1.00          | 3          | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:3.79s, min:3.78s, avg: 3.78s, p80:3.79s, p95:3.79
    ′→ s, iters:3, tasks:3, threads:60}
    ′→
    ′→ | funcs:max(tpch_100.lineitem.l_shipdate)->Column#28, funcs:max(tpch_100.lineitem.
    ′→ l_commitdate)->Column#29, funcs:max(tpch_100.lineitem.l_receiptdate)->Column#30 | N/A
    ′→     | N/A  |
|       └─HashJoin_44                     | 600845438.27 | 600037902 | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:3.17s, min:3.12s, avg: 3.14s, p80:3.17s, p95:3.17
    ′→ s, iters:11143, tasks:3, threads:60}
    ′→
    ′→ | inner join, equal:[eq(tpch_100.supplier.s_suppkey, tpch_100.lineitem.l_suppkey)],
    ′→ stream_count: 20                                                                  | N/A      |
    ′→  N/A  |
|         ├─ExchangeReceiver_30(Build)    | 1000000.00    | 1000000    | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:23.3ms, min:21ms, avg: 22.5ms, p80:23.3ms, p95
    ′→ :23.3ms, iters:120, tasks:3, threads:60}
    ′→
    ′→ | stream_count: 20
    ′→
    ′→ | N/A     | N/A  |
|         │ └─ExchangeSender_29           | 1000000.00    | 1000000    | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:22.7ms, min:0s, avg: 7.57ms, p80:22.7ms, p95:22.7
    ′→ ms, iters:16, tasks:3, threads:2}
    ′→
    ′→ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.supplier.s_suppkey, collate:
    ′→ binary], stream_count: 20                                                         | N/A      | N/
    ′→ A  |
|         │   └─TableFullScan_28          | 1000000.00    | 1000000    | mpp[tiflash] | table:
     ′→ supplier | tiflash_task:{proc max:9.71ms, min:0s, avg: 3.24ms, p80:9.71ms, p95:9.71ms,
    ′→ iters:16, tasks:3, threads:2}, tiflash_scan:{dtfile:{total_scanned_packs:123,
    ′→ total_skipped_packs:0, total_scanned_rows:1000000, total_skipped_rows:0,
    ′→ total_rs_index_load_time: 0ms, total_read_time: 2ms}, total_create_snapshot_time: 0ms}
    ′→                                    | keep order:false
    ′→
    ′→ | N/A     | N/A  |
|         └─ExchangeReceiver_33(Probe)    | 600037902.00 | 600037902 | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:564.3ms, min:340ms, avg: 438.5ms, p80:564.3ms,
    ′→ p95:564.3ms, iters:27583, tasks:3, threads:60}
    ′→
```

```
    '→ |
    '→
    '→ | N/A      | N/A  |
|            └─ExchangeSender_32             | 600037902.00 | 600037902 | mpp[tiflash] |
    '→                | tiflash_task:{proc max:3.69s, min:0s, avg: 1.23s, p80:3.69s, p95:3.69s,
    '→ iters:9298, tasks:3, threads:60}
    '→
    '→ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.lineitem.l_suppkey, collate:
    '→ binary]                                                                    | N/A      | N/
    '→ A  |
|              └─TableFullScan_31            | 600037902.00 | 600037902 | mpp[tiflash] | table:
     '→ lineitem | tiflash_task:{proc max:62.8ms, min:0s, avg: 20.9ms, p80:62.8ms, p95:62.8ms,
    '→ iters:9298, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73465,
    '→ total_skipped_packs:13060, total_scanned_rows:600168663, total_skipped_rows:106699256,
    '→ total_rs_index_load_time: 16ms, total_read_time: 27979ms}, total_create_snapshot_time: 0
    '→ ms}            | keep order:false
    '→
    '→ | N/A      | N/A  |
+--
    '→  ----------------------------------------+-------------+-----------+-------------+----------------+-----
    '→
11 rows in set (3.83 sec)
```

设置 tidb_broadcast_join_threshold_size：

```
mysql> set @@tidb_broadcast_join_threshold_size = 10000000;
Query OK, 0 rows affected (0.00 sec)
```

tidb_broadcast_join_threshold_size 设置后：

```
mysql> explain analyze select max(l_shipdate), max(l_commitdate), max(l_receiptdate) from
    '→ supplier,lineitem where s_suppkey = l_suppkey;
+--
    '→  ----------------------------------------+-------------+-----------+-------------+----------------+-----
    '→
| id                                      | estRows      | actRows    | task          | access
    '→ object  | execution info
    '→
    '→ | operator info
    '→
    '→ | memory  | disk |
+--
    '→  ----------------------------------------+-------------+-----------+-------------+----------------+-----
    '→
| HashAgg_43                              | 1.00         | 1          | root          |
    '→                | time:2.75s, loops:2, partial_worker:{wall_time:2.748445779s, concurrency
```

```
    ′→ :5, task_num:1, tot_wait:13.74202679s, tot_exec:8.012µs, tot_time:13.742045721s, max
    ′→ :2.748414915s, p95:2.748414915s}, final_worker:{wall_time:2.74848039s, concurrency:5,
    ′→ task_num:1, tot_wait:13.742157526s, tot_exec:53.191µs, tot_time:13.742214417s, max
    ′→ :2.748462393s, p95:2.748462393s}    | funcs:max(Column#28)->Column#25, funcs:max(Column
    ′→ #29)->Column#26, funcs:max(Column#30)->Column#27
    ′→                                                                    | 17.3 KB | N/A  |
| └─TableReader_45                          | 1.00         | 3         | root         |
    ′→                | time:2.75s, loops:2, cop_task: {num: 6, max: 0s, min: 0s, avg: 0s, p95:
    ′→ 0s, copr_cache_hit_ratio: 0.00}
    ′→
    ′→ | data:ExchangeSender_44
    ′→
    ′→ | N/A      | N/A  |
|   └─ExchangeSender_44                     | 1.00         | 3         | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:2.74s, min:2.65s, avg: 2.7s, p80:2.74s, p95:2.74s
    ′→ , iters:3, tasks:3, threads:60}
    ′→
    ′→ | ExchangeType: PassThrough
    ′→
    ′→ | N/A      | N/A  |
|     └─HashAgg_13                          | 1.00         | 3         | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:2.74s, min:2.65s, avg: 2.7s, p80:2.74s, p95:2.74s
    ′→ , iters:3, tasks:3, threads:60}
    ′→
    ′→ | funcs:max(tpch_100.lineitem.l_shipdate)->Column#28, funcs:max(tpch_100.lineitem.
    ′→ l_commitdate)->Column#29, funcs:max(tpch_100.lineitem.l_receiptdate)->Column#30 | N/A
    ′→     | N/A  |
|       └─HashJoin_42                       | 600845438.27 | 600037902 | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:2.15s, min:2.06s, avg: 2.11s, p80:2.15s, p95:2.15
    ′→ s, iters:9286, tasks:3, threads:60}
    ′→
    ′→ | inner join, equal:[eq(tpch_100.supplier.s_suppkey, tpch_100.lineitem.l_suppkey)]
    ′→                                                                                  | N/A
    ′→ | N/A  |
|       ├─ExchangeReceiver_30(Build)       | 1000000.00   | 3000000   | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:28.1ms, min:24.9ms, avg: 27ms, p80:28.1ms, p95
    ′→ :28.1ms, iters:48, tasks:3, threads:60}
    ′→
    ′→ |
    ′→
    ′→ | N/A      | N/A  |
|       │ └─ExchangeSender_29               | 1000000.00   | 1000000   | mpp[tiflash] |
    ′→                | tiflash_task:{proc max:14ms, min:0s, avg: 4.67ms, p80:14ms, p95:14ms,
    ′→ iters:16, tasks:3, threads:2}
    ′→
```

```
    '→ | ExchangeType: Broadcast
    '→
    '→ | N/A     | N/A  |
|         |   └─TableFullScan_28           | 1000000.00   | 1000000   | mpp[tiflash] | table:
    '→ supplier | tiflash_task:{proc max:9ms, min:0s, avg: 3ms, p80:9ms, p95:9ms, iters:16,
    '→ tasks:3, threads:2}, tiflash_scan:{dtfile:{total_scanned_packs:123, total_skipped_packs
    '→ :0, total_scanned_rows:1000000, total_skipped_rows:0, total_rs_index_load_time: 0ms,
    '→ total_read_time: 2ms}, total_create_snapshot_time: 0ms}
    '→                                                              | keep order:false
    '→
    '→ | N/A     | N/A  |
|         └─TableFullScan_31(Probe)        | 600037902.00 | 600037902 | mpp[tiflash] | table:
    '→ lineitem | tiflash_task:{proc max:57.9ms, min:42.9ms, avg: 51.3ms, p80:57.9ms, p95:57.9ms
    '→ , iters:9297, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73464,
    '→ total_skipped_packs:12985, total_scanned_rows:600169085, total_skipped_rows:106014866,
    '→ total_rs_index_load_time: 23ms, total_read_time: 21667ms}, total_create_snapshot_time: 0
    '→ ms}                  | keep order:false
    '→
    '→ | N/A     | N/A  |
+--
    '→  ----------------------------------------+-------------+-----------+-------------+-----------------+-----
    '→
9 rows in set (2.76 sec)
```

设置更大的执行并发度

设置更大的执行并发度，可以让TiFlash 占用更多系统 CPU 资源，从而提升查询性能。

tidb_max_tiflash_threads，单位为线程。用来设置 TiFlash 中request 执行的最大并发度。

```
set @@tidb_max_tiflash_threads = 20;
```

在以下示例中，tidb_max_tiflash_threads 设置前，所有TiFlash 的算子的执行线程数为24。tidb_max_tiflash_threads
'→ 设置后，所有TiFlash 的算子的执行线程数为 60。

tidb_max_tiflash_threads 设置前：

```
mysql> explain analyze select a, count(*) from t group by a;
+--
    '→  -----------------------------------+-------------+-----------+-------------+--------------+----------
    '→
| id                                  | estRows       | actRows    | task          | access object
    '→ | execution info
    '→
    '→ | operator info
    '→
    '→ | memory | disk |
```

```
+--
    '→  ----------------------------------+-------------+----------+-------------+------------ ---+----------
    '→
| TableReader_44                          | 20.00       | 20       | root        |
    '→ | time:655.9ms, loops:2, cop_task: {num: 14, max: 0s, min: 0s, avg: 0s, p95: 0s,
    '→ copr_cache_hit_ratio: 0.00}
    '→
    '→ | data:ExchangeSender_43
    '→
    '→ | N/A    | N/A  |
| └─ExchangeSender_43                   | 20.00       | 20       | mpp[tiflash] |
    '→               | tiflash_task:{proc max:654.6ms, min:652.4ms, avg: 653.8ms, p80:654.6ms,
    '→ p95:654.6ms, iters:11, tasks:3, threads:24}
    '→
    '→ | ExchangeType: PassThrough
    '→
    '→ | N/A    | N/A  |
|   └─Projection_5                      | 20.00       | 20       | mpp[tiflash] |
    '→               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    '→ p95:654.3ms, iters:11, tasks:3, threads:24}
    '→
    '→ | test.t.a, Column#22
    '→
    '→ | N/A    | N/A  |
|     └─Projection_39                   | 20.00       | 20       | mpp[tiflash] |
    '→               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    '→ p95:654.3ms, iters:11, tasks:3, threads:24}
    '→
    '→ | Column#22, test.t.a
    '→
    '→ | N/A    | N/A  |
|       └─HashAgg_40                   | 20.00       | 20       | mpp[tiflash] |
    '→               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    '→ p95:654.3ms, iters:11, tasks:3, threads:24}
    '→
    '→ | group by:test.t.a, funcs:sum(Column#29)->Column#22, funcs:firstrow(test.t.a)->test.t.a,
    '→  stream_count: 8                     | N/A    | N/A  |
|         └─ExchangeReceiver_42        | 20.00       | 60       | mpp[tiflash] |
    '→               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    '→ p95:654.3ms, iters:11, tasks:3, threads:24}
    '→
    '→ | stream_count: 8
    '→
    '→ | N/A    | N/A  |
|           └─ExchangeSender_41        | 20.00       | 60       | mpp[tiflash] |
```

```
    ′→               | tiflash_task:{proc max:649ms, min:0s, avg: 216.3ms, p80:649ms, p95:649ms,
    ′→  iters:3, tasks:3, threads:24}
    ′→
    ′→ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    ′→ : 8                                  | N/A     | N/A  |
|             └─HashAgg_37              | 20.00        | 60         | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:649ms, min:0s, avg: 216.3ms, p80:649ms, p95:649ms,
    ′→  iters:3, tasks:3, threads:24}
    ′→
    ′→ | group by:test.t.a, funcs:count(1)->Column#29
    ′→                                                                                  | N/A
    ′→ | N/A  |
|               └─TableFullScan_26      | 600000000.00 | 600000000 | mpp[tiflash] | table:t
    ′→       | tiflash_task:{proc max:40ms, min:0s, avg: 13.3ms, p80:40ms, p95:40ms, iters:9386,
    ′→  tasks:3, threads:24}, tiflash_scan:{dtfile:{total_scanned_packs:73833,
    ′→ total_skipped_packs:174, total_scanned_rows:600000000, total_skipped_rows:1402537,
    ′→ total_rs_index_load_time: 0ms, total_read_time: 4ms}, total_create_snapshot_time: 0ms} |
    ′→ keep order:false
    ′→
    ′→ | N/A     | N/A  |
+--
    ′→  ----------------------------------+--------------+-----------+--------------+---------------+----------
    ′→
9 rows in set (0.67 sec)
```

设置 tidb_max_tiflash_threads：

```
mysql> set @@tidb_max_tiflash_threads = 20;
Query OK, 0 rows affected (0.00 sec)
```

tidb_max_tiflash_threads 设置后：

```
mysql> explain analyze select a, count(*) from t group by a;
+--
    ′→  ----------------------------------+--------------+-----------+--------------+---------------+----------
    ′→
| id                                  | estRows      | actRows    | task          | access object
    ′→ | execution info
    ′→
    ′→ | operator info
    ′→
    ′→ | memory | disk |
+--
    ′→  ----------------------------------+--------------+-----------+--------------+---------------+----------
    ′→
| TableReader_44                      | 20.00        | 20         | root          |
    ′→ | time:357.7ms, loops:2, cop_task: {num: 19, max: 0s, min: 0s, avg: 0s, p95: 0s,
```

```
    '→ copr_cache_hit_ratio: 0.00}
    '→
    '→ | data:ExchangeSender_43
    '→
    '→ | N/A    | N/A  |
| └─ExchangeSender_43                  | 20.00        | 20        | mpp[tiflash] |
    '→               | tiflash_task:{proc max:355.3ms, min:354.6ms, avg: 355ms, p80:355.3ms, p95
    '→ :355.3ms, iters:16, tasks:3, threads:60}
    '→
    '→ | ExchangeType: PassThrough
    '→
    '→ | N/A    | N/A  |
|   └─Projection_5                     | 20.00        | 20        | mpp[tiflash] |
    '→               | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    '→ :354.3ms, iters:16, tasks:3, threads:60}
    '→
    '→ | test.t.a, Column#22
    '→
    '→ | N/A    | N/A  |
|     └─Projection_39                  | 20.00        | 20        | mpp[tiflash] |
    '→               | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    '→ :354.3ms, iters:16, tasks:3, threads:60}
    '→
    '→ | Column#22, test.t.a
    '→
    '→ | N/A    | N/A  |
|       └─HashAgg_40                  | 20.00        | 20        | mpp[tiflash] |
    '→               | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    '→ :354.3ms, iters:16, tasks:3, threads:60}
    '→
    '→ | group by:test.t.a, funcs:sum(Column#29)->Column#22, funcs:firstrow(test.t.a)->test.t.a,
    '→  stream_count: 20                      | N/A    | N/A  |
|         └─ExchangeReceiver_42       | 20.00        | 60        | mpp[tiflash] |
    '→               | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    '→ :354.3ms, iters:16, tasks:3, threads:60}
    '→
    '→ | stream_count: 20
    '→
    '→ | N/A    | N/A  |
|           └─ExchangeSender_41       | 20.00        | 60        | mpp[tiflash] |
    '→               | tiflash_task:{proc max:349.6ms, min:0s, avg: 116.5ms, p80:349.6ms, p95
    '→ :349.6ms, iters:3, tasks:3, threads:60}
    '→
    '→ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    '→ : 20                                  | N/A    | N/A  |
```

```
|             └─HashAgg_37              | 20.00         | 60         | mpp[tiflash] |
    ′→               | tiflash_task:{proc max:347.6ms, min:0s, avg: 115.9ms, p80:347.6ms, p95
    ′→ :347.6ms, iters:3, tasks:3, threads:60}
    ′→
    ′→ | group by:test.t.a, funcs:count(1)->Column#29
    ′→                                                                                | N/A
    ′→    | N/A  |
|               └─TableFullScan_26      | 600000000.00 | 600000000 | mpp[tiflash] | table:t
    ′→       | tiflash_task:{proc max:36.6ms, min:0s, avg: 12.2ms, p80:36.6ms, p95:36.6ms, iters
    ′→ :9389, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73833,
    ′→ total_skipped_packs:418, total_scanned_rows:600001386, total_skipped_rows:3381854,
    ′→ total_rs_index_load_time: 10ms, total_read_time: 61ms}, total_create_snapshot_time: 0ms}
    ′→ | keep order:false
    ′→
    ′→ | N/A    | N/A  |
+--
    ′→  ------------------------------------+--------------+-----------+--------------+---------------+----------
    ′→
9 rows in set (0.37 sec)
```

设置细粒度 Shuffle 参数

细粒度 Shuffle 可以通过参数增加窗口函数执行的并发度，让函数执行占用更多系统资源，从而提升查询性能。

tiflash_fine_grained_shuffle_stream_count，单位为线程数。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。

```
set @@tiflash_fine_grained_shuffle_stream_count = 20;
```

在以下示例中，tiflash_fine_grained_shuffle_stream_count 设置前，[ExchangeSender_11, ExchangeReceiver_12 ′→ , Sort_13, Window_22] 的 stream_count 是 8。tiflash_fine_grained_shuffle_stream_count 设置后，[ExchangeSender_11, ExchangeReceiver_12, Sort_13, Window_22] 的 stream_count 是 20。

tiflash_fine_grained_shuffle_stream_count 设置前：

```
mysql> explain analyze select *, row_number() over (partition by a) from t;
+--
    ′→  -------------------------------+--------------+-----------+--------------+---------------+-------------
    ′→
| id                                | estRows      | actRows   | task         | access object |
    ′→ execution info
    ′→
    ′→ | operator info
    ′→
    ′→ | memory | disk |
+--
    ′→  -------------------------------+--------------+-----------+--------------+---------------+-------------
    ′→
```

```
| TableReader_24                  | 600000000.00 | 600000000 | root         |               |
    ’→ time:4m30.5s, loops:585941, cop_task: {num: 9163, max: 0s, min: 0s, avg: 0s, p95: 0s,
    ’→ copr_cache_hit_ratio: 0.00}
    ’→
    ’→ | data:ExchangeSender_23
    ’→
    ’→ | N/A    | N/A  |
| └─ExchangeSender_23             | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ’→ tiflash_task:{proc max:4m30.5s, min:3m4.8s, avg: 3m36.1s, p80:4m30.5s, p95:4m30.5s, iters
    ’→ :9160, tasks:3, threads:24}
    ’→
    ’→ | ExchangeType: PassThrough
    ’→
    ’→ | N/A    | N/A  |
|   └─Window_22                   | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ’→ tiflash_task:{proc max:3m31.6s, min:2m26.2s, avg: 2m50.7s, p80:3m31.6s, p95:3m31.6s,
    ’→ iters:9160, tasks:3, threads:24}
    ’→
    ’→ | row_number()->Column#23 over(partition by test.t.a rows between current row and current
    ’→  row), stream_count: 8        | N/A    | N/A  |
|     └─Sort_13                   | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ’→ tiflash_task:{proc max:3m28.6s, min:2m24.2s, avg: 2m48.4s, p80:3m28.6s, p95:3m28.6s,
    ’→ iters:9160, tasks:3, threads:24}
    ’→
    ’→ | test.t.a, stream_count: 8
    ’→
    ’→ | N/A    | N/A  |
|       └─ExchangeReceiver_12     | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ’→ tiflash_task:{proc max:32.4s, min:32s, avg: 32.1s, p80:32.4s, p95:32.4s, iters:49307,
    ’→ tasks:3, threads:24}
    ’→
    ’→ | stream_count: 8
    ’→
    ’→ | N/A    | N/A  |
|         └─ExchangeSender_11     | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ’→ tiflash_task:{proc max:32s, min:0s, avg: 10.7s, p80:32s, p95:32s, iters:9386, tasks:3,
    ’→ threads:60}
    ’→
    ’→ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    ’→ : 8                          | N/A    | N/A  |
|           └─TableFullScan_10    | 600000000.00 | 600000000 | mpp[tiflash] | table:t       |
     ’→ tiflash_task:{proc max:113.9ms, min:0s, avg: 38ms, p80:113.9ms, p95:113.9ms, iters:9386,
    ’→ tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    ’→ total_skipped_packs:190, total_scanned_rows:600000000, total_skipped_rows:1536382,
    ’→ total_rs_index_load_time: 16ms, total_read_time: 166324ms}, total_create_snapshot_time: 0
```

```
    '→ ms} | keep order:false
    '→
    '→ | N/A    | N/A  |
+--
    '→  ------------------------------+--------------+-----------+-------------+---------------+--------------
    '→
7 rows in set (4 min 30.59 sec)
```

设置 tiflash_fine_grained_shuffle_stream_count：

```
mysql> set @@tiflash_fine_grained_shuffle_stream_count = 20;
Query OK, 0 rows affected (0.00 sec)
```

tiflash_fine_grained_shuffle_stream_count 设置后：

```
mysql> explain analyze select *, row_number() over (partition by a) from t;
+--
    '→  ------------------------------+--------------+-----------+-------------+---------------+--------------
    '→
| id                              | estRows       | actRows    | task          | access object |
    '→ execution info
    '→
    '→ | operator info
    '→
    '→ | memory | disk |
+--
    '→  ------------------------------+--------------+-----------+-------------+---------------+--------------
    '→
| TableReader_24                  | 600000000.00 | 600000000 | root          |               |
    '→ time:2m55s, loops:585941, cop_task: {num: 9163, max: 0s, min: 0s, avg: 0s, p95: 0s,
    '→ copr_cache_hit_ratio: 0.00}
    '→
    '→ | data:ExchangeSender_23
    '→
    '→ | N/A    | N/A  |
| └─ExchangeSender_23             | 600000000.00 | 600000000 | mpp[tiflash] |               |
     '→ tiflash_task:{proc max:2m55s, min:1m37s, avg: 2m28.7s, p80:2m55s, p95:2m55s, iters:9160,
    '→ tasks:3, threads:60}
    '→
    '→ | ExchangeType: PassThrough
    '→
    '→ | N/A    | N/A  |
|   └─Window_22                   | 600000000.00 | 600000000 | mpp[tiflash] |               |
     '→ tiflash_task:{proc max:2m12.9s, min:1m17s, avg: 1m54.2s, p80:2m12.9s, p95:2m12.9s, iters
    '→ :9160, tasks:3, threads:60}
    '→
```

```
    '→ | row_number()->Column#23 over(partition by test.t.a rows between current row and current
    '→  row), stream_count: 20         | N/A     | N/A   |
|     └─Sort_13                      | 600000000.00 | 600000000 | mpp[tiflash] |                |
     '→ tiflash_task:{proc max:2m10.9s, min:1m16s, avg: 1m52.5s, p80:2m10.9s, p95:2m10.9s, iters
    '→ :9160, tasks:3, threads:60}
    '→
    '→ | test.t.a, stream_count: 20
    '→
    '→ | N/A     | N/A   |
|       └─ExchangeReceiver_12       | 600000000.00 | 600000000 | mpp[tiflash] |                |
     '→ tiflash_task:{proc max:27.2s, min:25.5s, avg: 26.6s, p80:27.2s, p95:27.2s, iters:49602,
    '→ tasks:3, threads:60}
    '→
    '→ | stream_count: 20
    '→
    '→ | N/A     | N/A   |
|         └─ExchangeSender_11       | 600000000.00 | 600000000 | mpp[tiflash] |                |
    '→ tiflash_task:{proc max:25.5s, min:0s, avg: 8.51s, p80:25.5s, p95:25.5s, iters:9388, tasks
    '→ :3, threads:60}
    '→
    '→ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    '→ : 20                            | N/A     | N/A   |
|           └─TableFullScan_10      | 600000000.00 | 600000000 | mpp[tiflash] | table:t        |
     '→ tiflash_task:{proc max:167.3ms, min:0s, avg: 55.8ms, p80:167.3ms, p95:167.3ms, iters
    '→ :9388, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    '→ total_skipped_packs:408, total_scanned_rows:600002896, total_skipped_rows:3307316,
    '→ total_rs_index_load_time: 20ms, total_read_time: 179431ms}, total_create_snapshot_time: 0
    '→ ms} | keep order:false
    '→
    '→ | N/A     | N/A   |
+--
    '→  --------------------------------+--------------+-----------+--------------+---------------+--------------
    '→
7 rows in set (2 min 55.09 sec)
```

8.2.0.8 下推计算结果缓存

TiDB 从 4.0 起支持下推计算结果缓存（即 Coprocessor Cache 功能）。开启该功能后，将在 TiDB 实例侧缓存下推给 TiKV 计算的结果，在部分场景下起到加速效果。

8.2.0.8.1 配置

Coprocessor Cache 的配置均位于TiDB 的 `tikv-client.copr-cache` 配置项中。Coprocessor 的具体开启和配置方法，见TiDB 配置文件描述。

8.2.0.8.2 特性说明

- 所有 SQL 在单个TiDB 实例上的首次执行都不会被缓存。
- 缓存仅存储在TiDB 内存中，TiDB 重启后缓存会失效。
- 不同TiDB 实例之间不共享缓存。
- 缓存的是下推计算结果，即使缓存命中，后续仍有TiDB 计算。
- 缓存以Region 为单位。对Region 的写入会导致涉及该Region 的缓存失效。基于此原因，该功能主要会对很少变更的数据有效果。
- 下推计算请求相同时，缓存会被命中。通常在以下场景下，下推计算的请求是相同或部分相同的：
  - SQL 语句完全一致，例如重复执行相同的 SQL 语句。
    该场景下所有下推计算的请求都是一致的，所有请求都能利用上下推计算缓存。
  - SQL 语句包含一个变化的条件，其他部分一致，变化的条件是表主键或分区主键。
    该场景下一部分下推计算的请求会与之前出现过的一致，部分请求能利用上下推计算结果缓存。
  - SQL 语句包含多个变化的条件，其他部分一致，变化的条件完全匹配一个复合索引列。
    该场景下一部分下推计算的请求会与之前出现过的一致，部分请求能利用上下推计算结果缓存。
- 该功能对用户透明，开启和关闭都不影响计算结果，仅影响 SQL 执行时间。

8.2.0.8.3 检验缓存效果

可以通过执行 EXPLAIN ANALYZE 或查看Grafana 监控面板来检查 Coprocessor 的缓存效果。

使用 EXPLAIN ANALYZE

用户可以通过 EXPLAIN ANALYZE 语句查看读表算子中的缓存命中率，示例如下：

```
EXPLAIN ANALYZE SELECT * FROM t USE INDEX(a);
+--
    '→  ----------------------------+----------+---------+----------+------------------------+----------------
    '→
| id                           | estRows    | actRows | task      | access object          |
    '→ execution info
    '→
    '→ | operator info                   | memory                | disk |
+--
    '→  ----------------------------+----------+---------+----------+------------------------+----------------
    '→
| IndexLookUp_6                | 262400.00 | 262400  | root      |                        | time
    '→ :620.513742ms, loops:258, cop_task: {num: 4, max: 5.530817ms, min: 1.51829ms, avg:
    '→ 2.70883ms, p95: 5.530817ms, max_proc_keys: 2480, p95_proc_keys: 2480, tot_proc: 1ms,
    '→ tot_wait: 1ms, rpc_num: 4, rpc_time: 10.816328ms, copr_cache_hit_rate: 0.75} |
    '→                                  | 6.685169219970703 MB  | N/A  |
| ├─IndexFullScan_4(Build)       | 262400.00 | 262400  | cop[tikv] | table:t, index:a(a, c) |
    '→ proc max:93ms, min:1ms, p80:93ms, p95:93ms, iters:275, tasks:4
    '→
    '→ | keep order:false, stats:pseudo | 1.7549400329589844 MB | N/A  |
```

```
| └─TableRowIDScan_5(Probe)      | 262400.00 | 0        | cop[tikv] | table:t                |
    ’→ time:0ns, loops:0
    ’→
    ’→ | keep order:false, stats:pseudo | N/A                    | N/A   |
+--
    ’→  ----------------------------+----------+---------+----------+------------------------+---------------
    ’→
3 rows in set (0.62 sec)
```

执行结果的 execution info 列有 copr_cache_hit_ratio 信息，表示下推计算结果缓存的命中率。以上示例的 0.75 表示命中率大约是 75%。

查看 Grafana 监控面板

在 Grafana 监控中，tidb 命名空间下 distsql 子系统中可见 copr-cache 面板，该面板监控整个集群中下推计算结果缓存的命中次数、未命中次数和缓存丢弃次数。

8.2.0.9 垃圾回收(GC)

8.2.0.9.1 GC 机制简介

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。Garbage Collection (GC) 的任务便是清理不再需要的旧数据。

整体流程

一个 TiDB 集群中会有一个 TiDB 实例被选举为 GC leader，GC 的运行由 GC leader 来控制。

GC 会被定期触发。每次 GC 时，首先，TiDB 会计算一个称为 safe point 的时间戳，接下来 TiDB 会在保证 safe point 之后的快照全部拥有正确数据的前提下，删除更早的过期数据。每一轮 GC 分为以下三个步骤：

1. “Resolve Locks” 阶段会对所有 Region 扫描 safe point 之前的锁，并清理这些锁。
2. “Delete Ranges” 阶段快速地删除由于 DROP TABLE/DROP INDEX 等操作产生的整区间的废弃数据。
3. “Do GC” 阶段每个 TiKV 节点将会各自扫描该节点上的数据，并对每一个 key 删除其不再需要的旧版本。

默认配置下，GC 每 10 分钟触发一次，每次 GC 会保留最近 10 分钟内的数据（即默认 GC life time 为 10 分钟，safe point 的计算方式为当前时间减去 GC life time）。如果一轮 GC 运行时间太久，那么在一轮 GC 完成之前，即使到了下一次触发 GC 的时间也不会开始下一轮 GC。另外，为了使持续时间较长的事务能在超过 GC life time 之后仍然可以正常运行，safe point 不会超过正在执行中的事务的开始时间(start_ts)。

实现细节

Resolve Locks（清理锁）

TiDB 的事务是基于 Google Percolator 模型实现的，事务的提交是一个两阶段提交的过程。第一阶段完成时，所有涉及的 key 都会上锁，其中一个锁会被选为 Primary，其余的锁 (Secondary) 则会存储一个指向 Primary 的指针；第二阶段会将 Primary 锁所在的 key 加上一个 Write 记录，并去除锁。这里的 Write 记录就是历史上对该 key 进行写入或删除，或者该 key 上发生事务回滚的记录。Primary 锁被替换为何种 Write 记录标志着该事务提交成功与否。接下来，所有 Secondary 锁也会被依次替换。如果因为某些原因（如发生故障等），这些 Secondary 锁没有

完成替换、残留了下来，那么也可以根据锁中的信息取找到 Primary，并根据 Primary 是否提交来判断整个事务是否提交。但是，如果 Primary 的信息在GC 中被删除了，而该事务又存在未成功提交的 Secondary 锁，那么就永远无法得知该锁是否可以提交。这样，数据的正确性就无法保证。

Resolve Locks 这一步的任务即对 safe point 之前的锁进行清理。即如果一个锁对应的 Primary 已经提交，那么该锁也应该被提交；反之，则应该回滚。而如果 Primary 仍然是上锁的状态（没有提交也没有回滚），则应当将该事务视为超时失败而回滚。

Resolve Locks 有两种执行模式：

- LEGACY（默认模式）：由GC leader 对所有的Region 发送请求扫描过期的锁，并对扫到的锁查询 Primary 的状态，再发送请求对其进行提交或回滚。
- PHYSICAL：TiDB 绕过Raft 层直接扫描每个TiKV 节点上的数据。

> 警告：
>
> PHYSICAL模式（即启用 Green GC）目前是实验性功能，不建议在生产环境中使用。

你可以通过修改系统变量tidb_gc_scan_lock_mode 的值切换Resolve Locks 的执行模式。

Delete Ranges（删除区间）

在执行 DROP TABLE/INDEX 等操作时，会有大量连续的数据被删除。如果对每个 key 都进行删除操作、再对每个 key 进行GC 的话，那么执行效率和空间回收速度都可能非常的低下。事实上，这种时候TiDB 并不会对每个 key 进行删除操作，而是将这些待删除的区间及删除操作的时间戳记录下来。Delete Ranges 会将这些时间戳在 safe point 之前的区间进行快速的物理删除。

Do GC（进行GC 清理）

这一步即删除所有 key 的过期版本。为了保证 safe point 之后的任何时间戳都具有一致的快照，这一步删除 safe point 之前提交的数据，但是会对每个 key 保留 safe point 前的最后一次写入（除非最后一次写入是删除）。

在进行这一步时，TiDB 只需要将 safe point 发送给 PD，即可结束整轮 GC。TiKV 会自行检测到 safe point 发生了更新，会对当前节点上所有作为Region leader 进行GC。与此同时，GC leader 可以继续触发下一轮GC。

> 注意：
>
> 从 TiDB 5.0 版本起，CENTRAL GC 模式（需要TiDB 服务器发送GC 请求到各个Region）已经废弃，Do GC 这一步将只以 DISTRIBUTED GC 模式（从 TiDB 3.0 版起的默认模式）运行。

8.2.0.9.2 GC 配置

你可以通过以下系统变量进行GC 配置：

- tidb_gc_enable

- tidb_gc_run_interval
- tidb_gc_life_time
- tidb_gc_concurrency
- tidb_gc_scan_lock_mode
- tidb_gc_max_wait_time

流控

TiDB 支持GC 流控，可通过配置 gc.max-write-bytes-per-sec 限制GC worker 每秒数据写入量，降低对正常请求的影响，0 为关闭该功能。该配置可通过tikv-ctl 动态修改：

```
tikv-ctl --host=ip:port modify-tikv-config -n gc.max-write-bytes-per-sec -v 10MB
```

TiDB 5.0 引入的变化

在 TiDB 5.0 之前的版本中，GC 是通过系统表 mysql.tidb 进行配置的。从TiDB 5.0 版本起，GC 仍然可以通过系统表 mysql.tidb 进行配置，但建议你使用系统变量进行配置，这样可以确保对配置的任何更改都能得到验证，防止造成异常行为(#20655)。

TiDB 5.0 及之后的版本不再需要向各个TiKV Region 都发送触发GC 的请求，因此不再提供 CENTRAL GC 模式的支持，取而代之的是效率更高的 DISTRIBUTED GC 模式（自 TiDB 3.0 起的默认GC 模式）。

如果要了解TiDB 历史版本中GC 配置的变化信息，请使用左侧导航栏中的“TIDB 版本选择器” 切换到本文档的历史版本。

TiDB 6.1.0 引入的变化

在 TiDB 6.1.0 之前的版本中，TiDB 内部事务不会影响GC safe point 推进。从TiDB 6.1.0 版本起，计算 safe point 时会考虑内部事务的 startTS，从而解决内部事务因访问的数据被清理掉而导致失败的问题。带来的负面影响是如果内部事务运行时间过长，会导致 safe point 长时间不推进，进而会影响业务性能。

TiDB v6.1.0 引入了系统变量tidb_gc_max_wait_time 控制活跃事务阻塞GC safe point 推进的最长时间，超过该值后 GC safe point 会强制向后推进。

GC in Compaction Filter 机制

GC in Compaction Filter 机制是在分布式GC 模式(DISTRIBUTED GC mode) 的基础上，由RocksDB 的 Compaction 过程来进行GC，而不再使用一个单独的GC worker 线程。这样做的好处是避免了GC 引起的额外磁盘读取，以及避免清理掉的旧版本残留大量删除标记影响顺序扫描性能。可以由TiKV 配置文件中的以下开关控制：

```
[gc]
enable-compaction-filter = true
```

该GC 机制可通过在线配置变更开启：

```
show config where type = 'tikv' and name like '%enable-compaction-filter%';
```

```
+------+----------------------+-----------------------------+--------+
| Type | Instance             | Name                        | Value |
+------+----------------------+-----------------------------+--------+
| tikv | 172.16.5.37:20163 | gc.enable-compaction-filter | false |
| tikv | 172.16.5.36:20163 | gc.enable-compaction-filter | false |
```

```
| tikv | 172.16.5.35:20163 | gc.enable-compaction-filter | false |
+------+-------------------+-----------------------------+--------+
```

```
set config tikv gc.enable-compaction-filter = true;
show config where type = 'tikv' and name like '%enable-compaction-filter%';
```

```
+------+-------------------+-----------------------------+--------+
| Type | Instance          | Name                        | Value  |
+------+-------------------+-----------------------------+--------+
| tikv | 172.16.5.37:20163 | gc.enable-compaction-filter | true   |
| tikv | 172.16.5.36:20163 | gc.enable-compaction-filter | true   |
| tikv | 172.16.5.35:20163 | gc.enable-compaction-filter | true   |
+------+-------------------+-----------------------------+--------+
```

8.3 SQL 性能调优

8.3.1 SQL 性能调优

SQL 是一种声明性语言。一条 SQL 语句描述的是最终结果应该如何，而非按顺序执行的步骤。TiDB 会优化 SQL 语句的执行，语义上允许以任何顺序执行查询的各部分，前提是能正确返回语句所描述的最终结果。

SQL 性能优化的过程，可以理解为GPS 导航的过程。你提供地址后，GPS 软件利用各种统计信息（例如以前的行程、速度限制等元数据，以及实时交通信息）规划出一条最省时的路线。这与TiDB 中的 SQL 性能优化过程相对应。

本章节包括以下文档，可帮助你更好地理解查询执行计划：

- 理解TiDB 执行计划介绍如何使用 EXPLAIN 语句来理解TiDB 是如何执行某个查询的。
- SQL 优化流程概览介绍TiDB 可以使用的几种优化，以提高查询性能。
- 控制执行计划介绍如何控制执行计划的生成。TiDB 的执行计划非最优时，建议控制执行计划。

8.3.2 理解TiDB 执行计划

8.3.2.1 TiDB 执行计划概览

> **注意：**
>
> 使用 MySQL 客户端连接到TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 EXPLAIN 的输出结果不再换行，可按右箭头→ 键水平滚动阅读输出结果。

使用 EXPLAIN 可查看TiDB 执行某条语句时选用的执行计划。也就是说，TiDB 在考虑上数百或数千种可能的执行计划后，最终认定该执行计划消耗的资源最少、执行的速度最快。

EXPLAIN 示例如下：

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY auto_increment, a INT NOT NULL, pad1 VARCHAR(255),
    ↪ INDEX(a));
INSERT INTO t VALUES (1, 1, 'aaa'),(2,2, 'bbb');
EXPLAIN SELECT * FROM t WHERE a = 1;
```

返回的结果如下：

```
Query OK, 0 rows affected (0.96 sec)

Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0

+--
    ↪                                +         +           +                      +
                                                                                    ----------
    ↪
| id                             | estRows | task      | access object        | operator info
    ↪                                  |
+--
    ↪ -------------------------------+---------+-----------+----------------------+                    ----------
    ↪
| IndexLookUp_10                 | 10.00   | root      |                      |
    ↪                                                |
| ├─IndexRangeScan_8(Build)      | 10.00   | cop[tikv] | table:t, index:a(a) | range:[1,1], keep
    ↪ order:false, stats:pseudo |
| └─TableRowIDScan_9(Probe)      | 10.00   | cop[tikv] | table:t              | keep order:false,
    ↪ stats:pseudo               |
+--
    ↪                                +         +           +                      +
                                                                                    ----------
    ↪
3 rows in set (0.00 sec)
```

EXPLAIN 实际不会执行查询。EXPLAIN ANALYZE 可用于实际执行查询并显示执行计划。如果TiDB 所选的执行计划非最优，可用 EXPLAIN 或 EXPLAIN ANALYZE 来进行诊断。有关 EXPLAIN 用法的详细内容，参阅以下文档：

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.1.1 解读EXPLAIN 的返回结果

EXPLAIN 的返回结果包含以下字段：

- id 为算子名，或执行 SQL 语句需要执行的子任务。详见算子简介。
- estRows 为显示TiDB 预计会处理的行数。该预估数可能基于字典信息（例如访问方法基于主键或唯一键），或基于 CMSketch 或直方图等统计信息估算而来。
- task 显示算子在执行语句时的所在位置。详见Task 简介。
- access-object 显示被访问的表、分区和索引。显示的索引为部分索引。以上示例中TiDB 使用了a 列的索引。尤其是在有组合索引的情况下，该字段显示的信息很有参考意义。
- operator info 显示访问表、分区和索引的其他信息。详见operator info 结果。

> **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 版本之前的有所不同。
>
> 在 v6.4.0 之前，estRows 表示对于 Build 端子节点的每一行，Probe 端预计会处理的行数。自 v6.4.0 起，estRows 表示 Probe 端预计会处理的总行数。由于 EXPLAIN ANALYZE 中展示的实际行数（actRows 列）表示的是总行数，v6.4.0 起这些算子 estRows 的含义与 actRows 列的含义保持一致。
>
> 例如：

```
CREATE TABLE t1(a INT, b INT);
CREATE TABLE t2(a INT, b INT, INDEX ia(a));
EXPLAIN SELECT /*+ INL_JOIN(t2) */ * FROM t1 JOIN t2 ON t1.a = t2.a;
EXPLAIN SELECT (SELECT a FROM t2 WHERE t2.a = t1.b LIMIT 1) FROM t1;
```

```
-- v6.4.0 之前：
+--
    '→                                +          +           +                      +
    '→ ------------------------------- ---------- ----------- ---------------------- ----------------
| id                              | estRows  | task      | access object          |
    '→ operator info
    '→
    '→ |
+--
    '→                                +          +           +                      +
    '→ ------------------------------- ---------- ----------- ---------------------- ----------------
| IndexJoin_12                    | 12487.50 | root  |                            |
    '→ inner join, inner:IndexLookUp_11, outer key:test.t1.a, inner key:test.t2.a,
    '→  equal cond:eq(test.t1.a, test.t2.a) |
| ├─TableReader_24(Build)         | 9990.00  | root      |                        |
    '→  data:Selection_23
    '→
    '→ |
| | └─Selection_23                | 9990.00  | cop[tikv] |                        |
    '→  not(isnull(test.t1.a))
    '→
    '→ |
```

```
| |   └─TableFullScan_22          | 10000.00 | cop[tikv] | table:t1              |
    ↪ keep order:false, stats:pseudo
    ↪
    ↪ |
| └─IndexLookUp_11(Probe)         | 1.25     | root      |                       |
    ↪
    ↪
    ↪ |
|   ├─Selection_10(Build)         | 1.25     | cop[tikv] |                       |
    ↪ not(isnull(test.t2.a))
    ↪
    ↪ |
|   | └─IndexRangeScan_8          | 1.25     | cop[tikv] | table:t2, index:ia(a) |
    ↪ range: decided by [eq(test.t2.a, test.t1.a)], keep order:false, stats:
    ↪ pseudo                                  |
|   └─TableRowIDScan_9(Probe)     | 1.25     | cop[tikv] | table:t2              |
    ↪ keep order:false, stats:pseudo
    ↪
    ↪ |
+--
    ↪ ------------------------------+----------+-----------+----------------------+-----------------
    ↪
+--
    ↪ ------------------------------+----------+-----------+----------------------+-----------------
    ↪
| id                              | estRows  | task      | access object         |
    ↪ operator info
    ↪                                                                          |
+--
    ↪ ------------------------------+----------+-----------+----------------------+-----------------
    ↪
| Projection_12                   | 10000.00 | root      |                       |
    ↪ test.t2.a
    ↪                                                                        |
| └─Apply_14                      | 10000.00 | root      |                       |
    ↪ CARTESIAN left outer join
    ↪                                                      |
|   ├─TableReader_16(Build)       | 10000.00 | root      |                       |
    ↪ data:TableFullScan_15
    ↪                                                          |
|   | └─TableFullScan_15          | 10000.00 | cop[tikv] | table:t1              |
    ↪ keep order:false, stats:pseudo
    ↪                                                 |
```

```
|   └─Limit_17(Probe)             | 1.00     | root      |                         |
    '→ offset:0, count:1
    '→                                                                              |
|     └─IndexReader_21            | 1.00     | root      |                         |
    '→ index:Limit_20
    '→                                                                                 |
|       └─Limit_20                | 1.00     | cop[tikv] |                         |
    '→ offset:0, count:1
    '→                                                                              |
|         └─IndexRangeScan_19     | 1.00     | cop[tikv] | table:t2, index:ia(a) |
    '→  range: decided by [eq(test.t2.a, test.t1.b)], keep order:false, stats:
    '→ pseudo |
+--
    '→ --------------------------------+----------+-----------+-------------------------+-----------------
    '→

-- 自 v6.4.0 起:

-- 可以发现 `IndexLookUp_11`、`Selection_10`、`IndexRangeScan_8` 和 `
    '→ TableRowIDScan_9` 在 `estRows` 列显示的行数与 v6.4.0 以前的不同
+--
    '→                                 +          +           +                       +
    '→ --------------------------------+----------+-----------+-------------------------+-----------------
    '→
| id                              | estRows  | task      | access object           |
    '→ operator info
    '→
    '→ |
+--
    '→                                 +          +           +                       +
    '→ --------------------------------+----------+-----------+-------------------------+-----------------
| IndexJoin_12                    | 12487.50 | root      |                         |
    '→ inner join, inner:IndexLookUp_11, outer key:test.t1.a, inner key:test.t2.a,
    '→  equal cond:eq(test.t1.a, test.t2.a) |
| ├─TableReader_24(Build)         | 9990.00  | root      |                         |
    '→ data:Selection_23
    '→
    '→ |
| | └─Selection_23                | 9990.00  | cop[tikv] |                         |
    '→ not(isnull(test.t1.a))
    '→
    '→ |
| |   └─TableFullScan_22          | 10000.00 | cop[tikv] | table:t1                |
    '→  keep order:false, stats:pseudo
```

```
    '→
    '→ |
| └─IndexLookUp_11(Probe)          | 12487.50 | root      |                       |
    '→
    '→
    '→ |
|   ├─Selection_10(Build)          | 12487.50 | cop[tikv] |                       |
    '→  not(isnull(test.t2.a))
    '→
    '→ |
|   | └─IndexRangeScan_8           | 12500.00 | cop[tikv] | table:t2, index:ia(a) |
    '→  range: decided by [eq(test.t2.a, test.t1.a)], keep order:false, stats:
    '→ pseudo                                    |
|   └─TableRowIDScan_9(Probe)      | 12487.50 | cop[tikv] | table:t2              |
    '→  keep order:false, stats:pseudo
    '→
    '→ |
+--
    '→                                 +          +           +                       +
    '→ ---------------------------------+----------+-----------+-----------------------+-----------------
    '→

-- 可以发现 `Limit_17`、`IndexReader_21`、`Limit_20` 和 `IndexRangeScan_19` 在 `
    '→ estRows` 列显示的行数与 v6.4.0 以前的不同
+--
    '→                                 +          +           +                       +
    '→ ---------------------------------+----------+-----------+-----------------------+-----------------
    '→
| id                               | estRows  | task      | access object         |
    '→ operator info
    '→                                                                          |
+--
    '→                                    +          +           +-----------------------+
    '→                                                                                    -----------------
| Projection_12                    | 10000.00 | root      |                       |
    '→ test.t2.a
    '→                                                                       |
| └─Apply_14                       | 10000.00 | root      |                       |
    '→  CARTESIAN left outer join
    '→                                                        |
|   ├─TableReader_16(Build)        | 10000.00 | root      |                       |
    '→  data:TableFullScan_15
    '→                                                           |
|   | └─TableFullScan_15           | 10000.00 | cop[tikv] | table:t1              |
    '→  keep order:false, stats:pseudo
```

```
    '→                                                  |                                                  |
|   └─Limit_17(Probe)              | 10000.00 | root      |                                                  |
    '→ offset:0, count:1
    '→                                                  |                                                  |
|     └─IndexReader_21             | 10000.00 | root      |                                                  |
    '→ index:Limit_20
    '→                                                  |                                                  |
|       └─Limit_20                 | 10000.00 | cop[tikv] |                                                  |
    '→ offset:0, count:1
    '→                                                  |                                                  |
|         └─IndexRangeScan_19      | 10000.00 | cop[tikv] | table:t2, index:ia(a)                            |
    '→ range: decided by [eq(test.t2.a, test.t1.b)], keep order:false, stats:
    '→ pseudo                                           |                                                  |
+--
    '→                                  +          +           +                                          --+----------------------
    '→
```

算子简介

算子是为返回查询结果而执行的特定步骤。真正执行扫表（读盘或者读TiKV Block Cache）操作的算子有如下几类：

- TableFullScan：全表扫描。
- TableRangeScan：带有范围的表数据扫描。
- TableRowIDScan：根据上层传递下来的 RowID 扫描表数据。时常在索引读操作后检索符合条件的行。
- IndexFullScan：另一种“全表扫描”，扫的是索引数据，不是表数据。
- IndexRangeScan：带有范围的索引数据扫描操作。

TiDB 会汇聚TiKV/TiFlash 上扫描的数据或者计算结果，这种“数据汇聚”算子目前有如下几类：

- TableReader：将 TiKV 上底层扫表算子 TableFullScan 或 TableRangeScan 得到的数据进行汇总。
- IndexReader：将TiKV 上底层扫表算子IndexFullScan 或IndexRangeScan 得到的数据进行汇总。
- IndexLookUp：先汇总 Build 端 TiKV 扫描上来的RowID，再去 Probe 端上根据这些 RowID 精确地读取TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。
- IndexMerge：和 IndexLookupReader 类似，可以看做是它的扩展，可以同时读取多个索引的数据，有多个 Build 端，一个 Probe 端。执行过程也很类似，先汇总所有 Build 端 TiKV 扫描上来的RowID，再去 Probe 端上根据这些RowID 精确地读取TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。

算子的执行顺序

算子的结构是树状的，但在查询执行过程中，并不严格要求子节点任务在父节点之前完成。TiDB 支持同一查询内的并行处理，即子节点“流入”父节点。父节点、子节点和同级节点可能并行执行查询的一部分。

在以上示例中，`├─IndexRangeScan_8(Build)` 算子为`a(a)` 索引所匹配的行查找内部RowID。`└─TableRowIDScan_9 ′→ (Probe)` 算子随后从表中检索这些行。

Build 总是先于 Probe 执行，并且 Build 总是出现在 Probe 前面。即如果一个算子有多个子节点，子节点ID 后面有 Build 关键字的算子总是先于有 Probe 关键字的算子执行。TiDB 在展现执行计划的时候，Build 端总是第一个出现，接着才是 Probe 端。

范围查询

在 WHERE/HAVING/ON 条件中，TiDB 优化器会分析主键或索引键的查询返回。如数字、日期类型的比较符，如大于、小于、等于以及大于等于、小于等于，字符类型的 LIKE 符号等。

若要使用索引，条件必须是“Sargable”(Search ARGument ABLE) 的。例如条件 `YEAR(date_column)< 1992` 不能使用索引，但 `date_column < '1992-01-01` 就可以使用索引。

推荐使用同一类型的数据以及同一类型的字符串和排序规则进行比较，以避免引入额外的 `cast` 操作而导致不能利用索引。

可以在范围查询条件中使用 AND（求交集）和 OR（求并集）进行组合。对于多维组合索引，可以对多个列使用条件。例如对组合索引 `(a, b, c)`：

- 当 a 为等值查询时，可以继续求b 的查询范围。
- 当 b 也为等值查询时，可以继续求c 的查询范围。
- 反之，如果a 为非等值查询，则只能求a 的范围。

Task 简介

目前TiDB 的计算任务分为两种不同的task：cop task 和 root task。Cop task 是指使用TiKV 中的 Coprocessor 执行的计算任务，root task 是指在TiDB 中执行的计算任务。

SQL 优化的目标之一是将计算尽可能地下推到TiKV 中执行。TiKV 中的 Coprocessor 能支持大部分 SQL 内建函数（包括聚合函数和标量函数）、SQL `LIMIT` 操作、索引扫描和表扫描。

`operator info` 结果

`EXPLAIN` 返回结果中 `operator info` 列可显示诸如条件下推等信息。本文以上示例中，`operator info` 结果各字段解释如下：

- `range: [1,1]` 表示查询的 WHERE 字句`(a = 1)` 被下推到了TiKV，对应的task 为 `cop[tikv]`。
- `keep order:false` 表示该查询的语义不需要TiKV 按顺序返回结果。如果查询指定了排序（例如 `SELECT ′→ * FROM t WHERE a = 1 ORDER BY id`），该字段的返回结果为 `keep order:true`。
- `stats:pseudo` 表示 `estRows` 显示的预估数可能不准确。TiDB 定期在后台更新统计信息。也可以通过执行 `ANALYZE TABLE t` 来手动更新统计信息。

`EXPLAIN` 执行后，不同算子返回不同的信息。你可以使用 Optimizer Hints 来控制优化器的行为，以此控制物理算子的选择。例如 `/*+ HASH_JOIN(t1, t2)*/` 表示优化器将使用 Hash Join 算法。详细内容见Optimizer Hints。

8.3.2.1.2 算子相关的系统变量

TiDB 在 MySQL 的基础上，定义了一些专用的系统变量和语法用来优化性能。其中一些系统变量和具体的算子相关，比如算子的并发度，算子的内存使用上限，是否允许使用分区表等。这些都可以通过系统变量进行控制，从而影响各个算子执行的效率。

如果读者想要详细了解所有的系统变量及其使用规则，可以参见系统变量和语法。

8.3.2.1.3 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE
- TiDB in Action
- 系统变量

8.3.2.2 使用 EXPLAIN 解读执行计划

SQL 是一种声明性语言，因此用户无法根据 SQL 语句直接判断一条查询的执行是否有效率。用户首先要使用 EXPLAIN 语句查看当前的执行计划。

以 bikeshare 数据库示例（英文）中的一个 SQL 语句为例，该语句统计了 2017 年 7 月 1 日的行程次数：

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

```
+--
    '→                              +          +           +               +
    '→
| id                           | estRows   | task      | access object | operator info
    '→
    '→ |
+--
    '→                              +          +           +               +
    '→ ------------------------------ ---------- ----------- ----------------
| StreamAgg_20                 | 1.00      | root      |               | funcs:count(Column#13)->
    '→ Column#11
    '→                                                                                  |
| └─TableReader_21             | 1.00      | root      |               | data:StreamAgg_9
    '→
    '→ |
|   └─StreamAgg_9              | 1.00      | cop[tikv] |               | funcs:count(1)->Column
    '→ #13
    '→
    '→ |
|     └─Selection_19           | 250.00    | cop[tikv] |               | ge(bikeshare.trips.
    '→ start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.start_date, 2017-07-01
    '→ 23:59:59.000000) |
|       └─TableFullScan_18     | 10000.00 | cop[tikv] | table:trips    | keep order:false, stats:
    '→ pseudo
    '→                                                                                  |
+--
    '→                              +          +           +               +
    '→ ------------------------------ ---------- ----------- ----------------
```

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

以上是该查询的执行计划结果。从 `└─TableFullScan_18` 算子开始向上看，查询的执行过程如下（非最佳执行计划）：

1. Coprocessor (TiKV) 读取整张 `trips` 表的数据，作为一次 `TableFullScan` 操作，再将读取到的数据传递给 `Selection_19` 算子。`Selection_19` 算子仍在TiKV 内。
2. `Selection_19` 算子根据谓词`WHERE start_date BETWEEN ..` 进行数据过滤。预计大约有250 行数据满足该过滤条件（基于统计信息以及算子的执行逻辑估算而来）。`└─TableFullScan_18` 算子显示`stats:pseudo`，表示该表没有实际统计信息，执行 `ANALYZE TABLE trips` 收集统计信息后，预计的估算的数字会更加准确。
3. `COUNT` 函数随后应用于满足过滤条件的行，这一过程也是在TiKV (`cop[tikv]`) 中的 `StreamAgg_9` 算子内完成的。TiKV coprocessor 能执行一些 MySQL 内置函数，`COUNT` 是其中之一。
4. `StreamAgg_9` 算子执行的结果会被传递给 `TableReader_21` 算子（位于TiDB 进程中，即 `root` 任务）。执行计划中，`TableReader_21` 算子的 `estRows` 为 `1`，表示该算子将从每个访问的 TiKV Region 接收一行数据。这一请求过程的详情，可参阅 `EXPLAIN ANALYZE`。
5. `StreamAgg_20` 算子随后对`└─TableReader_21` 算子传来的每行数据计算 `COUNT` 函数的结果。`StreamAgg_20` 是根算子，会将结果返回给客户端。

> 注意：
>
> 要查看TiDB 中某张表的Region 信息，可执行 `SHOW TABLE REGIONS` 语句。

8.3.2.2.1 评估当前的性能

`EXPLAIN` 语句只返回查询的执行计划，并不执行该查询。若要获取实际的执行时间，可执行该查询，或使用 `EXPLAIN ANALYZE` 语句：

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

```
+--
    '→  ---------------------------+----------+----------+-----------+---------------+--------------------------
    '→
| id                          | estRows  | actRows  | task      | access object | execution info
    '→
    '→
    '→ | operator info
    '→
    '→ | memory    | disk |
```

```
+--
    '→  ---------------------------+----------+----------+-----------+--------------+--------------------------
    '→
| StreamAgg_20                  | 1.00     | 1        | root      |              | time
    '→ :1.031417203s, loops:2
    '→
    '→ | funcs:count(Column#13)->Column#11
    '→                                                                                |
    '→ 632 Bytes | N/A  |
| └─TableReader_21              | 1.00     | 56       | root      |              | time
    '→ :1.031408123s, loops:2, cop_task: {num: 56, max: 782.147269ms, min: 5.759953ms, avg:
    '→ 252.005927ms, p95: 609.294603ms, max_proc_keys: 910371, p95_proc_keys: 704775, tot_proc:
    '→ 11.524s, tot_wait: 580ms, rpc_num: 56, rpc_time: 14.111932641s} | data:StreamAgg_9
    '→
    '→ | 328 Bytes | N/A  |
|   └─StreamAgg_9               | 1.00     | 56       | cop[tikv] |              | proc max:640
    '→ ms, min:8ms, p80:276ms, p95:480ms, iters:18695, tasks:56
    '→
    '→ | funcs:count(1)->Column#13
    '→
    '→ | N/A       | N/A  |
|     └─Selection_19            | 250.00   | 11409    | cop[tikv] |              | proc max:640
    '→ ms, min:8ms, p80:276ms, p95:476ms, iters:18695, tasks:56
    '→
    '→ | ge(bikeshare.trips.start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.
    '→ start_date, 2017-07-01 23:59:59.000000) | N/A       | N/A  |
|       └─TableFullScan_18      | 10000.00 | 19117643 | cop[tikv] | table:trips  | proc max:612
    '→ ms, min:8ms, p80:248ms, p95:460ms, iters:18695, tasks:56
    '→
    '→ | keep order:false, stats:pseudo
    '→                                                                              |
    '→  N/A       | N/A  |
+--
    '→  ---------------------------+----------+----------+-----------+--------------+--------------------------
    '→
5 rows in set (1.03 sec)
```

执行以上示例查询耗时 1.03 秒，说明执行性能较为理想。

以上 EXPLAIN ANALYZE 的结果中，actRows 表明一些 estRows 预估数不准确（预估返回 10000 行数据但实际返回 19117643 行）。└─TableFullScan_18 算子的 operator info 列(stats:pseudo)信息也表明该算子的预估数不准确。

如果先执行 ANALYZE TABLE 再执行 EXPLAIN ANALYZE，预估数与实际数会更接近：

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

```
    '→ 2017-07-01 23:59:59';
```

```
Query OK, 0 rows affected (10.22 sec)

+--
    '→  ---------------------------+-------------+----------+-----------+---------------+----------------------
    '→
| id                            | estRows     | actRows  | task      | access object | execution
    '→ info
    '→
    '→ | operator info
    '→
    '→ | memory    | disk |
+--
    '→  ---------------------------+-------------+----------+-----------+---------------+----------------------
    '→
| StreamAgg_20                  | 1.00        | 1        | root      |               | time
    '→ :926.393612ms, loops:2
    '→
    '→ | funcs:count(Column#13)->Column#11
    '→                                                                                                 |
    '→ 632 Bytes | N/A  |
| └─TableReader_21              | 1.00        | 56       | root      |               | time
    '→ :926.384792ms, loops:2, cop_task: {num: 56, max: 850.94424ms, min: 6.042079ms, avg:
    '→ 234.987725ms, p95: 495.474806ms, max_proc_keys: 910371, p95_proc_keys: 704775, tot_proc:
    '→ 10.656s, tot_wait: 904ms, rpc_num: 56, rpc_time: 13.158911952s} | data:StreamAgg_9
    '→
    '→ | 328 Bytes | N/A  |
|   └─StreamAgg_9               | 1.00        | 56       | cop[tikv] |               | proc max
    '→ :592ms, min:4ms, p80:244ms, p95:480ms, iters:18695, tasks:56
    '→
    '→ | funcs:count(1)->Column#13
    '→
    '→ | N/A       | N/A  |
|     └─Selection_19            | 432.89      | 11409    | cop[tikv] |               | proc max
    '→ :592ms, min:4ms, p80:244ms, p95:480ms, iters:18695, tasks:56
    '→
    '→ | ge(bikeshare.trips.start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.
    '→ start_date, 2017-07-01 23:59:59.000000) | N/A       | N/A  |
|       └─TableFullScan_18      | 19117643.00 | 19117643 | cop[tikv] | table:trips   | proc max
    '→ :564ms, min:4ms, p80:228ms, p95:456ms, iters:18695, tasks:56
    '→
    '→ | keep order:false
    '→
    '→ | N/A       | N/A  |
```

```
+--
    '→  ---------------------------+-------------+----------+-----------+---------------+---------------------
    '→
5 rows in set (0.93 sec)
```

执行 ANALYZE TABLE 后，可以看到└─TableFullScan_18 算子的预估行数是准确的，└─Selection_19 算子的预估行数也更接近实际行数。以上两个示例中的执行计划（即TiDB 执行查询所使用的一组算子）未改变，但过时的统计信息常常导致TiDB 选择到非最优的执行计划。

除 ANALYZE TABLE 外，达到tidb_auto_analyze_ratio 阈值后，TiDB 会自动在后台重新生成统计数据。若要查看 TiDB 有多接近该阈值（即 TiDB 判断统计数据有多健康），可执行 SHOW STATS_HEALTHY 语句。

```
SHOW STATS_HEALTHY;
```

```
+-----------+------------+----------------+---------------- +
| Db_name   | Table_name | Partition_name | Healthy |
+-----------+------------+----------------+---------------- +
| bikeshare | trips      |                |     100 |
+-----------+------------+----------------+---------------- +
1 row in set (0.00 sec)
```

8.3.2.2.2 确定优化方案

当前执行计划是有效率的：

- 大部分任务是在TiKV 内处理的，需要通过网络传输给TiDB 处理的仅有 56 行数据，每行都满足过滤条件，而且都很短。
- 在 TiDB (StreamAgg_20) 中和在TiKV (└─StreamAgg_9) 中汇总行数都使用了 Stream Aggregate，该算法在内存使用方面很有效率。

当前执行计划存在的最大问题在于谓词 start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01 '→ 23:59:59' 并未立即生效，先是 TableFullScan 算子读取所有行数据，然后才进行过滤选择。可以在 SHOW CREATE TABLE trips 的返回结果中找出问题原因：

```
SHOW CREATE TABLE trips\G
```

```
*************************** 1. row ***************************
       Table: trips
Create Table: CREATE TABLE `trips` (
  `trip_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `duration` int(11) NOT NULL,
  `start_date` datetime DEFAULT NULL,
  `end_date` datetime DEFAULT NULL,
  `start_station_number` int(11) DEFAULT NULL,
  `start_station` varchar(255) DEFAULT NULL,
```

```
  `end_station_number` int(11) DEFAULT NULL,
  `end_station` varchar(255) DEFAULT NULL,
  `bike_number` varchar(255) DEFAULT NULL,
  `member_type` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`trip_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=20477318
1 row in set (0.00 sec)
```

以上返回结果显示，start_date 列没有索引。要将该谓词下推到index reader 算子，还需要一个索引。添加索引如下：

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

```
Query OK, 0 rows affected (2 min 10.23 sec)
```

> **注意：**
>
> 你可通过执行 ADMIN SHOW DDL JOBS 语句来查看 DDL 任务的进度。TiDB 中的默认值的设置较为保守，因此添加索引不会对生产环境下的负载造成太大影响。测试环境下，可以考虑调大 tidb_ddl_reorg_batch_size 和tidb_ddl_reorg_worker_cnt 的值。在参照系统上，将批处理大小设为 10240，将 worker count 并发度设置为 32，该系统可获得 10 倍的性能提升（较之使用默认值）。

添加索引后，可以使用 EXPLAIN 重复该查询。在以下返回结果中，可见TiDB 选择了新的执行计划，而且不再使用 TableFullScan 和 Selection 算子。

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

```
+--
    '→ ---------------------------+---------+-----------+-----------------------------------------+-----------
    '→
| id                          | estRows | task      | access object                           |
    '→ operator info                                                  |
+--
    '→ ---------------------------+---------+-----------+-----------------------------------------+-----------
    '→
| StreamAgg_17                | 1.00    | root      |                                         |
    '→ funcs:count(Column#13)->Column#11                              |
| └─IndexReader_18            | 1.00    | root      |
    '→ | index:StreamAgg_9                                            |
|   └─StreamAgg_9             | 1.00    | cop[tikv] |
    '→ | funcs:count(1)->Column#13                                     |
```

```
|     └─IndexRangeScan_16      | 8471.88 | cop[tikv] | table:trips, index:start_date(start_date)
    '→ | range:[2017-07-01 00:00:00,2017-07-01 23:59:59], keep order:false |
+--
    '→ ---------------------------+---------+-----------+-----------------------------------------+-----------
    '→
4 rows in set (0.00 sec)
```

若要比较实际的执行时间，可再次使用 `EXPLAIN ANALYZE` 语句：

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

```
+--
    '→ ---------------------------+---------+---------+-----------+-----------------------------------------+-
    '→
| id                          | estRows | actRows | task      | access object
    '→                            | execution info
    '→
    '→ | operator info                                                | memory    | disk |
+--
    '→ ---------------------------+---------+---------+-----------+-----------------------------------------+-
    '→
| StreamAgg_17                | 1.00    | 1       | root      |
    '→                                          | time:4.516728ms, loops:2
    '→                                                                                    |
    '→  funcs:count(Column#13)->Column#11                                | 372 Bytes | N/A  |
| └─IndexReader_18            | 1.00    | 1       | root      |
    '→                                          | time:4.514278ms, loops:2, cop_task: {num: 1,
    '→ max:4.462288ms, proc_keys: 11409, rpc_num: 1, rpc_time: 4.457148ms} | index:StreamAgg_9
    '→                                               | 238 Bytes | N/A  |
|   └─StreamAgg_9             | 1.00    | 1       | cop[tikv] |
    '→                                          | time:4ms, loops:12
    '→
    '→ | funcs:count(1)->Column#13                                | N/A       | N/A  |
|     └─IndexRangeScan_16     | 8471.88 | 11409   | cop[tikv] | table:trips, index:start_date(
    '→ start_date) | time:4ms, loops:12
    '→
    '→ | range:[2017-07-01 00:00:00,2017-07-01 23:59:59], keep order:false | N/A       | N/A  |
+--
    '→ ---------------------------+---------+---------+-----------+-----------------------------------------+-
    '→
4 rows in set (0.00 sec)
```

从以上结果可看出，查询时间已从 1.03 秒减少到 0.0 秒。

> 注意：
>
> 以上示例另一个可用的优化方案是coprocessor cache。如果你无法添加索引， 可考虑开启 coprocessor cache 功能。开启后，只要算子上次执行以来 Region 未作更改，TiKV 将从缓存中返回值。这也有助于减少 TableFullScan 和 Selection 算子的大部分运算成本。

8.3.2.3 用 EXPLAIN 查看 MPP 模式查询的执行计划

TiDB 支持使用 MPP 模式来执行查询。在 MPP 执行模式下，SQL 优化器会生成 MPP 的执行计划。注意 MPP 模式仅对有TiFlash 副本的表生效。

本文档使用的示例数据如下：

```
CREATE TABLE t1 (id int, value int);
INSERT INTO t1 values(1,2),(2,3),(1,3);
ALTER TABLE t1 set tiflash replica 1;
ANALYZE TABLE t1;
SET tidb_allow_mpp = 1;
```

8.3.2.3.1 MPP 查询片段和 MPP 任务

在 MPP 模式下，一个查询在逻辑上会被切分为多个 MPP 查询片段(query fragment)。示例如下：

```
EXPLAIN SELECT COUNT(*) FROM t1 GROUP BY id;
```

这个查询在 MPP 模式下会包含两个查询片段，一个为一阶段聚合，一个为二阶段聚合（最终聚合）。在查询执行的时候每个查询片段都会被实例化为一个或者多个 MPP 任务。

8.3.2.3.2 Exchange 算子

MPP 查询的执行计划中有两个 MPP 特有的Exchange 算子，分别为ExchangeReceiver 和 ExchangeSender。ExchangeReceiver 表示从下游查询片段读取数据，ExchangeSender 表示下游查询片段向上游查询片段发送数据。在 MPP 执行模式下，每个 MPP 查询片段的根算子均为ExchangeSender 算子，即每个查询片段以ExchangeSender 为界进行划分。一个简单的 MPP 计划如下：

```
EXPLAIN SELECT COUNT(*) FROM t1 GROUP BY id;
```

```
+--
    '→ ----------------------------------+---------+-------------------+--------------------------+-------------------------
    '→
| id                                  | estRows | task              | access object | operator
    '→ info                                       |
+--
    '→ ----------------------------------+---------+-------------------+---------------+-------------------------
    '→
```

```
| TableReader_31                      | 2.00    | root              |               | data:
    ′→ ExchangeSender_30                                  |
| └─ExchangeSender_30                 | 2.00    | batchCop[tiflash] |               |
    ′→ ExchangeType: PassThrough                          |
|   └─Projection_26                   | 2.00    | batchCop[tiflash] |               | Column#4
    ′→                                                    |
|     └─HashAgg_27                    | 2.00    | batchCop[tiflash] |               | group by:
    ′→ test.t1.id, funcs:sum(Column#7)->Column#4 |
|       └─ExchangeReceiver_29         | 2.00    | batchCop[tiflash] |               |
    ′→                                                    |
|         └─ExchangeSender_28         | 2.00    | batchCop[tiflash] |               |
    ′→ ExchangeType: HashPartition, Hash Cols: test.t1.id |
|           └─HashAgg_9               | 2.00    | batchCop[tiflash] |               | group by:
    ′→ test.t1.id, funcs:count(1)->Column#7               |
|             └─TableFullScan_25      | 3.00    | batchCop[tiflash] | table:t1      | keep order:
    ′→ false                                              |
+--
    ′→                                    +         +                   +               +
    ′→
```

以上执行计划中有两个查询片段：

- `[TableFullScan_25, HashAgg_9, ExchangeSender_28]` 为第一个查询片段，其主要完成一阶段聚合的计算。
- `[ExchangeReceiver_29, HashAgg_27, Projection_26, ExchangeSender_30]` 为第二个查询片段，其主要完成二阶段聚合的计算。

ExchangeSender 算子的 `operator info` 列输出了ExchangeType 信息。目前有以下三种ExchangeType：

- HashPartition：ExchangeSender 把数据按 Hash 值进行分区之后分发给上游的 MPP 任务的ExchangeReceiver 算子，通常在 Hash Aggregation 以及 Shuffle Hash Join 算法中使用。
- Broadcast：ExchangeSender 通过广播的方式把数据分发给上游的 MPP 任务，通常在 Broadcast Join 中使用。
- PassThrough：ExchangeSender 把数据分发给上游的 MPP Task，与 Broadcast 的区别是此时上游有且仅有一个 MPP 任务，通常用于向TiDB 返回数据。

上述例子中ExchangeSender 的 ExchangeType 为 HashPartition 以及 PassThrough，分别对应于 Hash Aggregation 运算以及向 TiDB 返回数据。

另外一个典型的 MPP 应用为join 运算。TiDB MPP 支持两种类型的join，分别为：

- Shuffle Hash Join：join 的 input 通过 HashPartition 的方式 shuffle 数据，上游的 MPP 任务进行分区内的join。
- Broadcast Join：join 中的小表以 Broadcast 的方式把数据广播到各个节点，各个节点各自进行join。

典型的 Shuffle Hash Join 执行计划如下：

```
SET tidb_broadcast_join_threshold_count=0; SET tidb_broadcast_join_threshold_size=0; EXPLAIN
    '→ SELECT COUNT(*) FROM t1 a JOIN t1 b ON a.id = b.id;
```

```
+--
    '→                                    +         +             +              +
    '→
| id                                  | estRows | task         | access object | operator info
    '→                                    |
+--
    '→                                    +         +             +              +
    '→ ------------------------------------ --------- -------------- ----------------
    '→
| StreamAgg_14                        | 1.00    | root         |               | funcs:count
    '→ (1)->Column#7                      |
| └─TableReader_48                    | 9.00    | root         |               | data:
    '→ ExchangeSender_47                  |
|   └─ExchangeSender_47               | 9.00    | cop[tiflash] |               | ExchangeType
    '→ : PassThrough                      |
|     └─HashJoin_44                   | 9.00    | cop[tiflash] |               | inner join,
    '→ equal:[eq(test.t1.id, test.t1.id)]     |
|       ├─ExchangeReceiver_19(Build)  | 6.00    | cop[tiflash] |               |
    '→                                          |
|       │ └─ExchangeSender_18         | 6.00    | cop[tiflash] |               | ExchangeType
    '→ : HashPartition, Hash Cols: test.t1.id |
|       │   └─Selection_17            | 6.00    | cop[tiflash] |               | not(isnull(
    '→ test.t1.id))                       |
|       │     └─TableFullScan_16      | 6.00    | cop[tiflash] | table:a       | keep order:
    '→ false                              |
|       └─ExchangeReceiver_23(Probe)  | 6.00    | cop[tiflash] |               |
    '→                                          |
|         └─ExchangeSender_22         | 6.00    | cop[tiflash] |               | ExchangeType
    '→ : HashPartition, Hash Cols: test.t1.id |
|           └─Selection_21            | 6.00    | cop[tiflash] |               | not(isnull(
    '→ test.t1.id))                       |
|             └─TableFullScan_20      | 6.00    | cop[tiflash] | table:b       | keep order:
    '→ false                              |
+--
    '→                                    +         +             +              +
    '→
12 rows in set (0.00 sec)
```

以上执行计划中，

- `[TableFullScan_20, Selection_21, ExchangeSender_22]` 完成表b 的数据读取并通过 HashPartition 的方式把数据 shuffle 给上游 MPP 任务。

- [TableFullScan_16, Selection_17, ExchangeSender_18] 完成表a 的数据读取并通过 HashPartition 的方式把数据 shuffle 给上游 MPP 任务。
- [ExchangeReceiver_19, ExchangeReceiver_23, HashJoin_44, ExchangeSender_47] 完成join 并把数据返回给TiDB。

典型的 Broadcast Join 执行计划如下：

```
EXPLAIN SELECT COUNT(*) FROM t1 a JOIN t1 b ON a.id = b.id;
```

```
+--
    '→                                        +         +              +               +
    '→
| id                                    | estRows | task          | access object | operator info
    '→                                    |
+--
    '→                                        +         +              +               +
    '→ ---------------------------------------- ---------- -------------- ----------------
| StreamAgg_15                          | 1.00    | root          |               | funcs:count
    '→ (1)->Column#7                        |
| └─TableReader_47                       | 9.00    | root          |               | data:
    '→ ExchangeSender_46                          |
|   └─ExchangeSender_46                  | 9.00    | cop[tiflash] |               | ExchangeType
    '→ : PassThrough                       |
|     └─HashJoin_43                      | 9.00    | cop[tiflash] |               | inner join,
    '→ equal:[eq(test.t1.id, test.t1.id)] |
|       ├─ExchangeReceiver_20(Build)     | 6.00    | cop[tiflash] |               |
    '→                                        |
|       | └─ExchangeSender_19            | 6.00    | cop[tiflash] |               | ExchangeType
    '→ : Broadcast                         |
|       |   └─Selection_18               | 6.00    | cop[tiflash] |               | not(isnull(
    '→ test.t1.id))                        |
|       |     └─TableFullScan_17         | 6.00    | cop[tiflash] | table:a       | keep order:
    '→ false                               |
|       └─Selection_22(Probe)            | 6.00    | cop[tiflash] |               | not(isnull(
    '→ test.t1.id))                        |
|         └─TableFullScan_21             | 6.00    | cop[tiflash] | table:b       | keep order:
    '→ false                               |
+--
    '→                                        +         +              +               +
    '→
```

以上执行计划中，

- [TableFullScan_17, Selection_18, ExchangeSender_19] 从小表（表 a）读数据并广播给大表（表 b）数据所在的各个节点。
- [TableFullScan_21, Selection_22, ExchangeReceiver_20, HashJoin_43, ExchangeSender_46] 完成join 并将数据返回给TiDB。

8.3.2.3.3 对 MPP 模式的查询使用 EXPLAIN ANALYZE

EXPLAIN ANALYZE 语句与 EXPLAIN 类似，但还会输出一些运行时的信息。一个简单的 EXPLAIN ANALYZE 输出信息如下：

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM t1 GROUP BY id;
```

```
+--
    ’→ ----------------------------------+---------+---------+-----------------------------+----------------+---------------
    ’→
| id                                | estRows | actRows | task              | access object |
    ’→ execution info
    ’→                                                                                                    |
    ’→ operator info                                                  | memory | disk |
+--
    ’→                                    +         +         +                     +                +
    ’→ ----------------------------------- ---------- ---------- ----------------------------- ---------------- ---------------
    ’→
| TableReader_31                      | 4.00    | 2       | root              |               |
    ’→ time:44.5ms, loops:2, cop_task: {num: 1, max: 0s, proc_keys: 0, copr_cache_hit_ratio:
    ’→ 0.00}       | data:ExchangeSender_30                                        | N/A    | N
    ’→ /A  |
| └─ExchangeSender_30                 | 4.00    | 2       | batchCop[tiflash] |               |

    ’→ tiflash_task:{time:16.5ms, loops:1, threads:1}
    ’→                                                                | ExchangeType: PassThrough, tasks:
    ’→ [2, 3, 4]                      | N/A    | N/A  |
|   └─Projection_26                   | 4.00    | 2       | batchCop[tiflash] |               |
    ’→ tiflash_task:{time:16.5ms, loops:1, threads:1}
    ’→                                                                | Column#4
    ’→                                                                   | N/A    | N/A  |
|     └─HashAgg_27                    | 4.00    | 2       | batchCop[tiflash] |               |
    ’→ tiflash_task:{time:16.5ms, loops:1, threads:1}
    ’→                                                                | group by:test.t1.id, funcs:sum(
    ’→ Column#7)->Column#4              | N/A    | N/A  |
|       └─ExchangeReceiver_29         | 4.00    | 2       | batchCop[tiflash] |               |
    ’→ tiflash_task:{time:14.5ms, loops:1, threads:20}
    ’→                                                              |
    ’→                                                                          | N/A    | N/A  |
|         └─ExchangeSender_28         | 4.00    | 0       | batchCop[tiflash] |               |
    ’→ tiflash_task:{time:9.49ms, loops:0, threads:0}
    ’→                                                                | ExchangeType: HashPartition, Hash
    ’→ Cols: test.t1.id, tasks: [1] | N/A    | N/A  |
|           └─HashAgg_9               | 4.00    | 0       | batchCop[tiflash] |               |
    ’→ tiflash_task:{time:9.49ms, loops:0, threads:0}
    ’→                                                                | group by:test.t1.id, funcs:count(1)
    ’→ ->Column#7                     | N/A    | N/A  |
|             └─TableFullScan_25      | 6.00    | 0       | batchCop[tiflash] | table:t1      |
```

```
    '→ tiflash_task:{time:9.49ms, loops:0, threads:0}, tiflash_scan:{dtfile:{total_scanned_packs
    '→ :1,...}} | keep order:false                                                    | N/A     | N/A
    '→  |
+--
    '→ ---------------------------------+---------+---------+-----------------------+---------------+-----------------------
    '→
```

与 EXPLAIN 相比，ExchangeSender 的 operator info 中多了 task id 的输出，其记录了该查询片段实例化成的 MPP 任务的任务ID。此外 MPP 算子中都会有 threads 这一列，这列记录了 MPP 在执行该算子时使用的并发数（如果集群由多个节点组成，该并发数是所有节点并发数相加的结果）。

8.3.2.3.4 MPP Version 和 Exchange 数据压缩

从 v6.6.0 开始，MPP 执行计划新增字段 MppVersion 和 Compression。

- MppVersion：MPP 执行计划的版本号，可通过系统变量mpp_version 设置。
- Compression：Exchange 算子的数据压缩模式，可通过系统变量mpp_exchange_compression_mode 设置。如果未启用数据压缩，则执行计划中不显示该字段。

```
mysql > EXPLAIN SELECT COUNT(*) AS count_order FROM  lineitem GROUP BY l_returnflag, l_linestatus
    '→ ORDER BY l_returnflag, l_linestatus;

+--
    '→                                    +             +             +               +
    '→
| id                                  | estRows       | task          | access object   |
    '→ operator info
    '→
    '→ |
+--
    '→                                    +             +             +               +
    '→
| Projection_6                        | 3.00          | root          |                 | Column
    '→ #18
    '→
    '→ |
|  └─Sort_8                           | 3.00          | root          |                 |
    '→ tpch100.lineitem.l_returnflag, tpch100.lineitem.l_linestatus
    '→
    '→ |
|    └─TableReader_36                 | 3.00          | root          |                 |
    '→ MppVersion: 1, data:ExchangeSender_35
    '→
    '→ |
```

```
|   └─ExchangeSender_35                | 3.00          | mpp[tiflash] |                |
    ’→ ExchangeType: PassThrough
    ’→
    ’→ |
|     └─Projection_31                  | 3.00          | mpp[tiflash] |                | Column
    ’→ #18, tpch100.lineitem.l_returnflag, tpch100.lineitem.l_linestatus
    ’→
    ’→ |
|       └─HashAgg_32                   | 3.00          | mpp[tiflash] |                | group
    ’→ by:tpch100.lineitem.l_linestatus, tpch100.lineitem.l_returnflag, funcs:sum(Column#23)->
    ’→ Column#18, funcs:firstrow(tpch100.lineitem.l_returnflag)->tpch100.lineitem.l_returnflag,
    ’→ funcs:firstrow(tpch100.lineitem.l_linestatus)->tpch100.lineitem.l_linestatus,
    ’→ stream_count: 20 |
|         └─ExchangeReceiver_34        | 3.00          | mpp[tiflash] |                |
    ’→ stream_count: 20
    ’→
    ’→ |
|           └─ExchangeSender_33        | 3.00          | mpp[tiflash] |                |
    ’→ ExchangeType: HashPartition, Compression: FAST, Hash Cols: [name: tpch100.lineitem.
    ’→ l_returnflag, collate: utf8mb4_bin], [name: tpch100.lineitem.l_linestatus, collate:
    ’→ utf8mb4_bin], stream_count: 20
    ’→                                                                                  |
|             └─HashAgg_14             | 3.00          | mpp[tiflash] |                | group
    ’→ by:tpch100.lineitem.l_linestatus, tpch100.lineitem.l_returnflag, funcs:count(1)->Column
    ’→ #23
    ’→
    ’→ |
|               └─TableFullScan_30     | 600037902.00 | mpp[tiflash] | table:lineitem | keep
    ’→ order:false
    ’→
    ’→ |
+--
    ’→                                    +             +              +                +
    ’→
```

例如上面的执行计划结果中，TiDB 使用了版本为 1 的 MPP 执行计划来构建 TableReader。其中类型为 HashPartition 的 ExchangeSender 算子使用 FAST 数据压缩模式，类型为 PassThrough 的 ExchangeSender 算子未启用数据压缩。

8.3.2.3.5 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划

- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.4 用 EXPLAIN 查看索引查询的执行计划

SQL 查询可能会使用索引，可以通过 EXPLAIN 语句来查看索引查询的执行计划。本文提供多个示例，以帮助用户理解索引查询是如何执行的。

TiDB 支持以下使用索引的算子来提升查询速度：

- IndexLookup
- IndexReader
- Point_Get 和 Batch_Point_Get
- IndexFullScan

本文档中的示例都基于以下数据：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 intkey INT NOT NULL,
 pad1 VARBINARY(1024),
 INDEX (intkey)
);

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    '→ LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    '→ LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    '→ LIMIT 10000;
```

8.3.2.4.1 IndexLookup

TiDB 从二级索引检索数据时会使用 IndexLookup 算子。例如，以下所有查询均会在 intkey 列的索引上使用 IndexLookup 算子：

```
EXPLAIN SELECT * FROM t1 WHERE intkey = 123;
EXPLAIN SELECT * FROM t1 WHERE intkey < 10;
EXPLAIN SELECT * FROM t1 WHERE intkey BETWEEN 300 AND 310;
EXPLAIN SELECT * FROM t1 WHERE intkey IN (123,29,98);
EXPLAIN SELECT * FROM t1 WHERE intkey >= 99 AND intkey <= 103;
```

```
+--
    '→                                +          +            +                                  +
    '→
```

```
| id                            | estRows | task      | access object                   | operator
    ′→  info                     |
+--
    ′→ ------------------------------+---------+-----------+---------------------------------+----------------------
    ′→
| IndexLookUp_10                | 1.00    | root      |                                 |
    ′→                                    |
| ├─IndexRangeScan_8(Build)     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ′→ :[123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 1.00    | cop[tikv] | table:t1                        | keep
    ′→ order:false                   |
+--
    ′→ ------------------------------+---------+-----------+---------------------------------+----------------------
    ′→
3 rows in set (0.00 sec)

+--
    ′→ ------------------------------+---------+-----------+---------------------------------+
    ′→                                                                                        ----------------------
| id                            | estRows | task      | access object                   | operator
    ′→  info                     |
+--
    ′→ ------------------------------+---------+-----------+---------------------------------+----------------------
    ′→
| IndexLookUp_10                | 3.60    | root      |                                 |
    ′→                                    |
| ├─IndexRangeScan_8(Build)     | 3.60    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ′→ :[-inf,10), keep order:false |
| └─TableRowIDScan_9(Probe)     | 3.60    | cop[tikv] | table:t1                        | keep
    ′→ order:false                   |
+--
    ′→ ------------------------------+---------+-----------+---------------------------------+
    ′→                                                                                        ----------------------
3 rows in set (0.00 sec)

+--
    ′→ ------------------------------+---------+-----------+---------------------------------+
    ′→                                                                                        ----------------------
| id                            | estRows | task      | access object                   | operator
    ′→  info                     |
+--
    ′→ ------------------------------+---------+-----------+---------------------------------+
    ′→                                                                                        ----------------------
| IndexLookUp_10                | 5.67    | root      |                                 |
    ′→                                    |
```

```
| ├─IndexRangeScan_8(Build)      | 5.67    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ′→ :[300,310], keep order:false |
| └─TableRowIDScan_9(Probe)      | 5.67    | cop[tikv] | table:t1                       | keep
    ′→ order:false                  |
+--
    ′→ ------------------------------+---------+-----------+--------------------------------+-------------------
    ′→
3 rows in set (0.00 sec)

+--
    ′→ ------------------------------+---------+-----------+--------------------------------+
    ′→
| id                            | estRows | task      | access object                  | operator
    ′→  info                                        |
+--
    ′→ ------------------------------+---------+-----------+--------------------------------+
    ′→
| IndexLookUp_10                | 4.00    | root      |                                |
    ′→                                                  |
| ├─IndexRangeScan_8(Build)      | 4.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ′→ :[29,29], [98,98], [123,123], keep order:false |
| └─TableRowIDScan_9(Probe)      | 4.00    | cop[tikv] | table:t1                       | keep
    ′→ order:false                                    |
+--
    ′→ ------------------------------+---------+-----------+--------------------------------+
    ′→
3 rows in set (0.00 sec)

+--
    ′→ ------------------------------+---------+-----------+--------------------------------+
    ′→
| id                            | estRows | task      | access object                  | operator
    ′→  info                     |
+--
    ′→ ------------------------------+---------+-----------+--------------------------------+
    ′→
| IndexLookUp_10                | 6.00    | root      |                                |
    ′→                                    |
| ├─IndexRangeScan_8(Build)      | 6.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ′→ :[99,103], keep order:false |
| └─TableRowIDScan_9(Probe)      | 6.00    | cop[tikv] | table:t1                       | keep
    ′→ order:false                  |
+--
    ′→ ------------------------------+---------+-----------+--------------------------------+
    ′→
```

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

IndexLookup 算子有以下两个子节点：

- ├─IndexRangeScan_8(Build) 算子节点对 intkey 列的索引执行范围扫描，并检索内部的 RowID 值（对此表而言，即为主键）。
- └─TableRowIDScan_9(Probe) 算子节点随后从表数据中检索整行。

IndexLookup 任务分以上两步执行。如果满足条件的行较多，SQL 优化器可能会根据统计信息选择使用 TableFullScan 算子。在以下示例中，很多行都满足intkey > 100 这一条件，因此优化器选择了TableFullScan：

```
EXPLAIN SELECT * FROM t1 WHERE intkey > 100;
```

```
+-------------------------+---------+-----------+---------------+--------------------------------+
| id                      | estRows | task      | access object | operator info                  |
+-------------------------+---------+-----------+---------------+--------------------------------+
| TableReader_7           | 898.50  | root      |               | data:Selection_6               |
| └─Selection_6           | 898.50  | cop[tikv] |               | gt(test.t1.intkey, 100)        |
|   └─TableFullScan_5     | 1010.00 | cop[tikv] | table:t1      | keep order:false               |
+-------------------------+---------+-----------+---------------+--------------------------------+
3 rows in set (0.00 sec)
```

IndexLookup 算子能在带索引的列上有效优化 LIMIT：

```
EXPLAIN SELECT * FROM t1 ORDER BY intkey DESC LIMIT 10;
```

```
+--
    ′→ ------------------------------+---------+-----------+--------------------------------+----------
    ′→
| id                              | estRows | task      | access object                  |
    ′→ operator info                       |
+--
    ′→                                 +         +           +                                +
                                                                                                ----------
    ′→
| IndexLookUp_21                  | 10.00   | root      |                                | limit
    ′→ embedded(offset:0, count:10) |
| ├─Limit_20(Build)               | 10.00   | cop[tikv] |                                | offset
    ′→ :0, count:10                 |
| │ └─IndexFullScan_18            | 10.00   | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ′→ order:true, desc             |
| └─TableRowIDScan_19(Probe)      | 10.00   | cop[tikv] | table:t1                       | keep
    ′→ order:false, stats:pseudo    |
+--
    ′→                                 +         +           +                                +
                                                                                                ----------
    ′→
4 rows in set (0.00 sec)
```

以上示例中，TiDB 从 intkey 索引读取最后 10 行，然后从表数据中检索这些行的 RowID 值。

8.3.2.4.2 IndexReader

TiDB 支持覆盖索引优化(covering index optimization)。如果 TiDB 能从索引中检索出所有行，就会跳过 IndexLookup 任务中通常所需的第二步（即从表数据中检索整行）。示例如下：

```
EXPLAIN SELECT * FROM t1 WHERE intkey = 123;
EXPLAIN SELECT id FROM t1 WHERE intkey = 123;
```

```
+--
    '-> ----------------------------+---------+-----------+--------------------------------+-----------------
    '->
| id                            | estRows | task      | access object                  | operator
    '->  info                     |
+--
    '-> ----------------------------+---------+-----------+--------------------------------+-----------------
    '->
| IndexLookUp_10                | 1.00    | root      |                                |
    '->                                     |
| ├─IndexRangeScan_8(Build)     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    '-> :[123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 1.00    | cop[tikv] | table:t1                       | keep
    '-> order:false               |
+--
    '-> ----------------------------+---------+-----------+--------------------------------+-----------------
    '->
3 rows in set (0.00 sec)

+--
    '-> ------------------------+---------+-----------+--------------------------------+-----------------
    '->
| id                          | estRows | task      | access object                  | operator info
    '->                        |
+--
    '-> ------------------------+---------+-----------+--------------------------------+-----------------
    '->
| Projection_4                | 1.00    | root      |                                | test.t1.id
    '->                          |
| └─IndexReader_6             | 1.00    | root      |                                | index:
    '-> IndexRangeScan_5             |
|   └─IndexRangeScan_5        | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    '-> :[123,123], keep order:false |
+--
    '-> ------------------------+---------+-----------+--------------------------------+
    '->                                                                                  -----------------
3 rows in set (0.00 sec)
```

以上结果中，id 也是内部的 RowID 值，因此 id 也存储在 intkey 索引中。部分└─IndexRangeScan_5 任务使用 intkey 索引后，可直接返回 RowID 值。

8.3.2.4.3 Point_Get 和 Batch_Point_Get

TiDB 直接从主键或唯一键检索数据时会使用 Point_Get 或 Batch_Point_Get 算子。这两个算子比 IndexLookup 更有效率。示例如下：

```
EXPLAIN SELECT * FROM t1 WHERE id = 1234;
EXPLAIN SELECT * FROM t1 WHERE id IN (1234,123);

ALTER TABLE t1 ADD unique_key INT;
UPDATE t1 SET unique_key = id;
ALTER TABLE t1 ADD UNIQUE KEY (unique_key);

EXPLAIN SELECT * FROM t1 WHERE unique_key = 1234;
EXPLAIN SELECT * FROM t1 WHERE unique_key IN (1234, 123);
```

```
+-------------+---------+------+---------------+------------------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+------------------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1234    |
+-------------+---------+------+---------------+------------------------+
1 row in set (0.00 sec)

+--
    '→                  +          +      +                +                                                                                +
    '→ ------------------ ---------- ------ ----------------
| id                | estRows | task | access object | operator info
    '→                                    |
+--
    '→ ------------------+----------+------+----------------+                                                                                +
    '→
| Batch_Point_Get_1 | 2.00    | root | table:t1      | handle:[1234 123], keep order:false, desc:
    '→ false |
+--
    '→ ------------------+----------+------+----------------+                                                                                +
    '→
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.27 sec)

Query OK, 1010 rows affected (0.06 sec)
Rows matched: 1010  Changed: 1010  Warnings: 0

Query OK, 0 rows affected (0.37 sec)
```

```
+-------------+---------+------+---------------------------------------+-----------------------------+
| id          | estRows | task | access object                         | operator info |
+-------------+---------+------+---------------------------------------+-----------------------------+
| Point_Get_1 | 1.00    | root | table:t1, index:unique_key(unique_key) |               |
+-------------+---------+------+---------------------------------------+-----------------------------+
1 row in set (0.00 sec)

+--
    '→                  +         +      +                                        +
    '→ ------------------ ---------- ------- ----------------------------------------
| id                | estRows | task | access object                          | operator info
    '→                |
+--
    '→                  +         +      +                                        +
    '→ ------------------ ---------- ------- ----------------------------------------
| Batch_Point_Get_1 | 2.00    | root | table:t1, index:unique_key(unique_key) | keep order:false,
    '→  desc:false |
+--
    '→                  +         +      +                                        +


    '→
1 row in set (0.00 sec)
```

8.3.2.4.4 IndexFullScan

索引是有序的，所以优化器可以使用 IndexFullScan 算子来优化常见的查询，例如在索引值上使用 MIN 或 Max 函数：

```
EXPLAIN SELECT MIN(intkey) FROM t1;
EXPLAIN SELECT MAX(intkey) FROM t1;
```

```
+--
    '→                            +         +           +                              +

    '→
| id                           | estRows | task      | access object                  | operator
    '→ info                      |
+--
    '→ -----------------------------+---------+-----------+--------------------------------+
    '→
| StreamAgg_12                 | 1.00    | root      |                                | funcs:min
    '→ (test.t1.intkey)->Column#4 |
| └─Limit_16                   | 1.00    | root      |                                 | offset
    '→ :0, count:1                    |
|   └─IndexReader_29           | 1.00    | root      |                                 | index:
    '→ Limit_28                       |
```

```
|     └─Limit_28                | 1.00    | cop[tikv] |                                 | offset
    ′→ :0, count:1                    |
|       └─IndexFullScan_27      | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ′→ order:true                      |
+--
    ′→                            +         +           +                                 +
    ′→
5 rows in set (0.00 sec)

+--
    ′→                            +         +           +                                 +
    ′→
| id                            | estRows | task      | access object                   | operator
    ′→ info                       |
+--
    ′→ ------------------------------+---------+-----------+---------------------------------+
    ′→
| StreamAgg_12                  | 1.00    | root      |                                 | funcs:max
    ′→ (test.t1.intkey)->Column#4 |
| └─Limit_16                    | 1.00    | root      |                                 | offset
    ′→ :0, count:1                    |
|   └─IndexReader_29            | 1.00    | root      |                                 | index:
    ′→ Limit_28                       |
|     └─Limit_28                | 1.00    | cop[tikv] |                                 | offset
    ′→ :0, count:1                    |
|       └─IndexFullScan_27      | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ′→ order:true, desc               |
+--
    ′→                            +         +           +                                 +
    ′→
5 rows in set (0.00 sec)
```

以上语句的执行过程中，TiDB 在每一个TiKV Region 上执行 IndexFullScan 操作。虽然算子名为 FullScan 即全扫描，TiDB 只读取第一行(└─Limit_28)。每个 TiKV Region 返回各自的 MIN 或 MAX 值给TiDB，TiDB 再执行流聚合运算来过滤出一行数据。即使表为空，带 MAX 或 MIN 函数的流聚合运算也能保证返回 NULL 值。

相反，在没有索引的值上执行 MIN 函数会在每一个TiKV Region 上执行 TableFullScan 操作。该查询会要求在TiKV 中扫描所有行，但 TopN 计算可保证每个TiKV Region 只返回一行数据给TiDB。尽管 TopN 能减少TiDB 和 TiKV 之间的多余数据传输，但该查询的效率仍远不及以上示例（MIN 能够使用索引）。

```
EXPLAIN SELECT MIN(pad1) FROM t1;
```

```
+--
    ′→ ------------------------------+---------+-----------+--------------+--------------------------------+
    ′→
| id                              | estRows | task      | access object | operator info
    ′→                     |
```

```
+--
    '→ ------------------------------+---------+-----------+---------------+--------------------------------+
    '→
| StreamAgg_13                    | 1.00    | root      |               | funcs:min(test.t1.pad1)
    '→ ->Column#4 |
| └─TopN_14                       | 1.00    | root      |               | test.t1.pad1, offset:0,
    '→  count:1    |
|   └─TableReader_23              | 1.00    | root      |               | data:TopN_22
    '→                          |
|     └─TopN_22                   | 1.00    | cop[tikv] |               | test.t1.pad1, offset:0,
    '→  count:1    |
|       └─Selection_21            | 1008.99 | cop[tikv] |               | not(isnull(test.t1.pad1
    '→ ))          |
|         └─TableFullScan_20      | 1010.00 | cop[tikv] | table:t1      | keep order:false
    '→                    |
+--    ------------------------------ ---------- ----------- ---------------                                +
    '→                                +         +           +               +----------------------------------
    '→
6 rows in set (0.00 sec)
```

执行以下语句时，TiDB 将使用 IndexFullScan 算子扫描索引中的每一行：

```
EXPLAIN SELECT SUM(intkey) FROM t1;
EXPLAIN SELECT AVG(intkey) FROM t1;
```

```
+--
    '→                              +         +           +                                +
    '→
| id                            | estRows | task      | access object                   | operator
    '→ info                         |
+--
    '→                              +         +           +                                +
    '→
| StreamAgg_20                  | 1.00    | root      |                                 | funcs:sum(
    '→ Column#6)->Column#4        |
| └─IndexReader_21              | 1.00    | root      |                                 | index:
    '→ StreamAgg_8                    |
|   └─StreamAgg_8               | 1.00    | cop[tikv] |                                 | funcs:sum(
    '→ test.t1.intkey)->Column#6 |
|     └─IndexFullScan_19        | 1010.00 | cop[tikv] | table:t1, index:intkey(intkey) | keep order
    '→ :false                       |
+--
    '→                              +         +           +                                +
    '→
4 rows in set (0.00 sec)
```

```
+--
    '→                              +         +           +                                  +
    '→
| id                          | estRows | task       | access object                    | operator
    '→ info                                                                  |
+--
    '→ ---------------------------+---------+-----------+----------------------------------+
    '→
| StreamAgg_20                | 1.00    | root       |                                  | funcs:avg(
    '→ Column#7, Column#8)->Column#4                                         |
| └─IndexReader_21            | 1.00    | root       |                                   | index:
    '→ StreamAgg_8                                                              |
|   └─StreamAgg_8             | 1.00    | cop[tikv] |                                   | funcs:
    '→  count(test.t1.intkey)->Column#7, funcs:sum(test.t1.intkey)->Column#8 |
|     └─IndexFullScan_19      | 1010.00 | cop[tikv] | table:t1, index:intkey(intkey) | keep order
    '→ :false                                                                |
+--
    '→                              +         +           +                                  +
    '→
4 rows in set (0.00 sec)
```

以上示例中，IndexFullScan 比 TableFullScan 更有效率，因为 (intkey + RowID) 索引中值的长度小于整行的长度。

以下语句不支持使用 IndexFullScan 算子，因为涉及该表中的其他列：

```
EXPLAIN SELECT AVG(intkey), ANY_VALUE(pad1) FROM t1;
```

```
+--
    '→                              +         +           +               +
    '→
| id                          | estRows | task       | access object | operator info
    '→
    '→ |
+--
    '→                              +         +           +               +
    '→ ---------------------------+---------+-----------+---------------
    '→
| Projection_4                | 1.00    | root       |               | Column#4, any_value(test.
    '→ t1.pad1)->Column#5
    '→                                                                        |
| └─StreamAgg_16              | 1.00    | root       |               | funcs:avg(Column#10,
    '→ Column#11)->Column#4, funcs:firstrow(Column#12)->test.t1.pad1
    '→                                    |
|   └─TableReader_17          | 1.00    | root       |               | data:StreamAgg_8
    '→
    '→ |
```

```
|     └─StreamAgg_8             | 1.00     | cop[tikv] |               | funcs:count(test.t1.
    '→ intkey)->Column#10, funcs:sum(test.t1.intkey)->Column#11, funcs:firstrow(test.t1.pad1)->
    '→ Column#12 |
|       └─TableFullScan_15      | 1010.00 | cop[tikv] | table:t1      | keep order:false
    '→
    '→ |
+--
    '→                              +         +           +               +
    '→
5 rows in set (0.00 sec)
```

8.3.2.4.5 其他类型查询的执行计划

- MPP 模式查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.5 用 EXPLAIN 查看JOIN 查询的执行计划

SQL 查询中可能会使用JOIN 进行表连接，可以通过 EXPLAIN 语句来查看JOIN 查询的执行计划。本文提供多个示例，以帮助用户理解表连接查询是如何执行的。

在TiDB 中，SQL 优化器需要确定数据表的连接顺序，且要判断对于某条特定的 SQL 语句，哪一种Join 算法最为高效。

本文档使用的示例数据如下：

```
CREATE TABLE t1 (id BIGINT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB,
    '→ int_col INT NOT NULL DEFAULT 0);
CREATE TABLE t2 (id BIGINT NOT NULL PRIMARY KEY auto_increment, t1_id BIGINT NOT NULL, pad1 BLOB,
    '→  pad2 BLOB, pad3 BLOB, INDEX(t1_id));
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM
    '→ dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    '→  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    '→  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    '→  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    '→  a JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
SELECT SLEEP(1);
ANALYZE TABLE t1, t2;
```

8.3.2.5.1 Index Join

如果预计需要连接的行数较少（一般小于 1 万行），推荐使用 Index Join 算法。这个算法与 MySQL 主要使用的 Join 算法类似。在下表的示例中，`TableReader_28(Build)` 算子首先读取表 `t1`，然后根据在 `t1` 中匹配到的每行数据，依次探查表 `t2` 中的数据：

> **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 版本之前的有所不同。细节请参考TiDB 执行计划概

览

```
EXPLAIN SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id;
```

```
+--
    '→ -------------------------------+-----------+------------+-----------------------------------+----------
    '→
| id                                | estRows   | task       | access object                     |
    '→ operator info
    '→
    '→ |
+--                                      +           +                                     +
    '→                                +                                                     +
    '→
| IndexJoin_11                      | 90000.00 | root       |                                | inner
    '→ join, inner:IndexLookUp_10, outer key:test.t1.id, inner key:test.t2.t1_id, equal
    cond:eq( '→ test.t1.id, test.t2.t1_id) |
| ├─TableReader_29(Build)           | 71010.00 | root       |                                | data:
    '→ TableFullScan_28
    '→
    '→ |
| │ └─TableFullScan_28              | 71010.00 | cop[tikv] | table:t1                        | keep
    '→ order:false
    '→
    '→ |
| └─IndexLookUp_10(Probe)           | 90000.00 | root       |                                |
    '→
    '→ |
|   ├─IndexRangeScan_8(Build)       | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | range:
    '→  decided by [eq(test.t2.t1_id, test.t1.id)], keep order:false
    '→                                                      |
|   └─TableRowIDScan_9(Probe)       | 90000.00 | cop[tikv] | table:t2                        | keep
    '→ order:false
    '→
    '→ |
+--
    '→                                +           +           +                             +
    '→
```

Index Join 算法对内存消耗较小，但如果需要执行大量探查操作，运行速度可能会慢于其他Join 算法。以下面这条查询语句为例：

```
SELECT * FROM t1 INNER JOIN t2 ON t1.id=t2.t1_id WHERE t1.pad1 = 'value' and t2.pad1='value';
```

在Inner Join 操作中，TiDB 会先执行Join Reorder 算法，所以不能确定会先读取 t1 还是 t2。假设 TiDB 先读取了 t1 来构建 Build 端，那么TiDB 会在探查 t2 前先根据谓词 t1.col = 'value' 筛选数据，但接下来每次探查 t2 时都要应用谓词 t2.col='value'。所以对于这条语句，Index Join 算法可能不如其他Join 算法高效。

但如果 Build 端的数据量比 Probe 端小，且 Probe 端的数据已预先建立了索引，那么这种情况下Index Join 算法效率更高。在下面这段查询语句中，因为Index Join 比 Hash Join 效率低，所以 SQL 优化器选择了 Hash Join 算法：

```
-- 删除已有索引
ALTER TABLE t2 DROP INDEX t1_id;

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ′→  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ′→ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
+--
    ′→  -------------------------+----------+---------+-----------+---------------+--------------------------
    ′→
| id                          | estRows  | actRows | task      | access object | execution info
    ′→
    ′→ | operator info
    ′→
    ′→ | memory  | disk |
+--
    ′→  -------------------------+----------+---------+-----------+---------------+--------------------------
    ′→
| IndexJoin_14                | 90000.00 | 0       | root      |               | time:330.2ms,
    ′→ loops:1, inner:{total:72.2ms, concurrency:5, task:12, construct:58.6ms, fetch:13.5ms,
    ′→ build:2.12µs}, probe:26.1ms
    ′→
    ′→ | inner join, inner:TableReader_10, outer key:test.t2.t1_id, inner key:test.t1.id, equal
    ′→ cond:eq(test.t2.t1_id, test.t1.id) | 88.5 MB | N/A  |
| ├─TableReader_20(Build)     | 90000.00 | 90000   | root      |               | time:307.2ms,
    ′→ loops:96, cop_task: {num: 24, max: 130.6ms, min: 170.9µs, avg: 33.5ms, p95: 105ms,
    ′→ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 472ms, rpc_num: 24, rpc_time: 802.4
    ′→ ms, copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ′→                                                                          | data:
    ′→ TableFullScan_19
    ′→
    ′→ | 58.6 MB | N/A  |
| │ └─TableFullScan_19        | 90000.00 | 90000   | cop[tikv] | table:t2      | tikv_task:{proc
    ′→  max:34ms, min:0s, avg: 15.3ms, p80:24ms, p95:30ms, iters:181, tasks:24}, scan_detail: {
    ′→ total_process_keys: 69744, total_process_keys_size: 217533936, total_keys: 69753,
    ′→ get_snapshot_time: 701.6µs, rocksdb: {delete_skipped_count: 97368, key_skipped_count:
    ′→ 236847, block: {cache_hit_count: 3509}}}  | keep order:false
```

```
    ′→
    ′→ | N/A      | N/A  |
| └─TableReader_10(Probe)      | 12617.92 | 0       | root      |               | time:11.9ms,
    ′→ loops:12, cop_task: {num: 42, max: 848.8µs, min: 199µs, avg: 451.8µs, p95: 846.2µs,
    ′→ max_proc_keys: 7, p95_proc_keys: 5, rpc_num: 42, rpc_time: 18.3ms, copr_cache_hit_ratio:
    ′→ 0.00, distsql_concurrency: 15}
    ′→
    ′→ | data:Selection_9
    ′→
    ′→ | N/A      | N/A  |
|   └─Selection_9              | 12617.92 | 0       | cop[tikv] |               | tikv_task:{proc
    ′→  max:0s, min:0s, avg: 0s, p80:0s, p95:0s, iters:42, tasks:42}, scan_detail: {
    ′→ total_process_keys: 56, total_process_keys_size: 174608, total_keys: 77,
    ′→ get_snapshot_time: 727.7µs, rocksdb: {block: {cache_hit_count: 154}}}
    ′→                                                                                | eq(test.
    ′→ t1.int_col, 1)
    ′→
    ′→ | N/A      | N/A  |
|     └─TableRangeScan_8       | 90000.00 | 56      | cop[tikv] | table:t1      | tikv_task:{proc
    ′→  max:0s, min:0s, avg: 0s, p80:0s, p95:0s, iters:42, tasks:42}
    ′→
    ′→ | range: decided by [test.t2.t1_id], keep order:false
    ′→                                                                       | N/A      | N/A  |
+--
     ′→ ---------------------------+----------+---------+-----------+---------------+------------+-------------
    ′→


+--
     ′→ ---------------------------+----------+---------+-----------+---------------+------------+-------------
    ′→
| id                           | estRows  | actRows | task      | access object | execution info
    ′→
    ′→ | operator info                                  | memory  | disk    |
+--
     ′→ ---------------------------+----------+---------+-----------+---------------+------------+-------------
    ′→
| HashJoin_20                  | 90000.00 | 0       | root      |               | time:313.6ms,
    ′→ loops:1, build_hash_table:{total:24.6ms, fetch:21.2ms, build:3.32ms}, probe:{concurrency
    ′→ :5, total:1.57s, max:313.5ms, probe:18.9ms, fetch:1.55s}
    ′→
    ′→ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.0 MB | 0 Bytes |
| ├─TableReader_23(Build)      | 9955.54  | 10000   | root      |               | time:23.6ms,
    ′→ loops:12, cop_task: {num: 11, max: 504.6µs, min: 203.7µs, avg: 377.4µs, p95: 504.6µs,
    ′→ rpc_num: 11, rpc_time: 3.92ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ′→
```

```
    ′→ | data:Selection_22                                | 14.9 MB | N/A      |
| | └─Selection_22             | 9955.54  | 10000    | cop[tikv] |               | tikv_task:{
    ′→ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ′→ scan_detail: {get_snapshot_time: 241.4µs, rocksdb: {block: {}}}
    ′→
    ′→ | eq(test.t1.int_col, 1)                           | N/A      | N/A      |
| |   └─TableFullScan_21       | 71010.00 | 71010    | cop[tikv] | table:t1      | tikv_task:{
    ′→ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ′→
    ′→ | keep order:false                                 | N/A      | N/A      |
| └─TableReader_25(Probe)      | 90000.00 | 90000    | root      |               | time:293.7ms,
    ′→ loops:91, cop_task: {num: 24, max: 105.7ms, min: 210.9µs, avg: 31.4ms, p95: 103.8ms,
    ′→ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 407ms, rpc_num: 24, rpc_time: 752.2
    ′→ ms, copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ′→                                                                          | data:
    ′→ TableFullScan_24                              | 58.6 MB | N/A      |
|   └─TableFullScan_24         | 90000.00 | 90000    | cop[tikv] | table:t2      | tikv_task:{
    ′→ proc max:31ms, min:0s, avg: 13ms, p80:19ms, p95:26ms, iters:181, tasks:24}, scan_detail:
    ′→ {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys: 69753,
    ′→ get_snapshot_time: 637.2µs, rocksdb: {delete_skipped_count: 97368, key_skipped_count:
    ′→ 236847, block: {cache_hit_count: 3509}}}  | keep order:false
    ′→                                    | N/A      | N/A      |
+--
     ′→ ---------------------------+----------+---------+-----------+---------------+------------ -------------
    ′→


+--
     ′→ ---------------------------+----------+---------+-----------+---------------+------------  -------------
    ′→
| id                           | estRows  | actRows | task      | access object | execution info
    ′→
    ′→ | operator info                                    | memory   | disk     |
+--
     ′→ ---------------------------+----------+---------+-----------+---------------+------------  -------------
    ′→
| HashJoin_21                  | 90000.00 | 0       | root      |               | time:331.7ms,
    ′→ loops:1, build_hash_table:{total:32.7ms, fetch:26ms, build:6.73ms}, probe:{concurrency:5,
    ′→  total:1.66s, max:331.3ms, probe:16ms, fetch:1.64s}
    ′→
    ′→ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.3 MB | 0 Bytes |
| ├─TableReader_26(Build)      | 9955.54  | 10000    | root      |               | time:30.4ms,
    ′→ loops:13, cop_task: {num: 11, max: 1.87ms, min: 844.7µs, avg: 1.29ms, p95: 1.87ms,
    ′→ rpc_num: 11, rpc_time: 13.5ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ′→
    ′→ | data:Selection_25                                | 12.2 MB | N/A      |
```

```
| | └─Selection_25              | 9955.54  | 10000    | cop[tikv] |                | tikv_task:{
    ′→ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ′→ scan_detail: {get_snapshot_time: 521µs, rocksdb: {block: {}}}
    ′→
    ′→ | eq(test.t1.int_col, 1)                              | N/A      | N/A      |
| |   └─TableFullScan_24        | 71010.00 | 71010    | cop[tikv] | table:t1       | tikv_task:{
    ′→ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ′→
    ′→ | keep order:false                                    | N/A      | N/A      |
| └─TableReader_23(Probe)       | 90000.00 | 90000    | root      |                | time:308.6ms,
    ′→ loops:91, cop_task: {num: 24, max: 123.3ms, min: 518.9µs, avg: 32.4ms, p95: 113.4ms,
    ′→ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 499ms, rpc_num: 24, rpc_time: 776ms,
    ′→  copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ′→                                                                                | data:
    ′→ TableFullScan_22                               | 58.6 MB | N/A      |
|   └─TableFullScan_22          | 90000.00 | 90000    | cop[tikv] | table:t2       | tikv_task:{
    ′→ proc max:44ms, min:0s, avg: 16.8ms, p80:27ms, p95:40ms, iters:181, tasks:24}, scan_detail
    ′→ : {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys: 69753,
    ′→ get_snapshot_time: 955.4µs, rocksdb: {delete_skipped_count: 97368, key_skipped_count:
    ′→ 236847, block: {cache_hit_count: 3509}}}  | keep order:false
    ′→                                    | N/A      | N/A      |
+--
    ′→  ----------------------------+----------+---------+-----------+---------------+--------------------------
    ′→
```

在上面所示的Index Join 操作中，`t1.int_col` 一列的索引被删除了。如果加上这个索引，操作执行速度可以从 0.3 秒提高到 0.06 秒，如下表所示：

```
-- 重新添加索引
ALTER TABLE t2 ADD INDEX (t1_id);

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ′→  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ′→ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
+--
    ′→  -------------------------------+----------+---------+-----------+-----------------------------+--------
    ′→
| id                                | estRows  | actRows | task      | access object
    ′→                | execution info
    ′→
    ′→ | operator info
    ′→
    ′→ | memory    | disk |
```

```
+--
    ’→  ------------------------------+----------+---------+-----------+------------------------ -----+--------
    ’→
| IndexJoin_12                      | 90000.00 | 0       | root      |
    ’→                                | time:65.6ms, loops:1, inner:{total:129.7ms, concurrency:5,
    ’→  task:7, construct:7.13ms, fetch:122.5ms, build:16.4µs}, probe:2.54ms
    ’→
    ’→ | inner join, inner:IndexLookUp_11, outer key:test.t1.id, inner key:test.t2.t1_id, equal
    ’→ cond:eq(test.t1.id, test.t2.t1_id) | 28.7 MB    | N/A  |
| ├─TableReader_33(Build)           | 9955.54  | 10000   | root      |
    ’→                                | time:15.4ms, loops:16, cop_task: {num: 11, max: 1.52ms,
    ’→ min: 211.5µs, avg: 416.8µs, p95: 1.52ms, rpc_num: 11, rpc_time: 4.36ms,
    ’→ copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ’→
    ’→ | data:Selection_32
    ’→
    ’→ | 13.9 MB   | N/A  |
| | └─Selection_32                  | 9955.54  | 10000   | cop[tikv] |
    ’→                                | tikv_task:{proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms,
    ’→  p95:104ms, iters:113, tasks:11}, scan_detail: {get_snapshot_time: 185µs, rocksdb: {block
    ’→ : {}}}
    ’→
    ’→ | eq(test.t1.int_col, 1)
    ’→
    ’→ | N/A       | N/A  |
| |   └─TableFullScan_31            | 71010.00 | 71010   | cop[tikv] | table:t1
    ’→                       | tikv_task:{proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101
    ’→ ms, iters:113, tasks:11}
    ’→
    ’→ | keep order:false
    ’→
    ’→ | N/A       | N/A  |
| └─IndexLookUp_11(Probe)           | 90000.00 | 0       | root      |
    ’→                                | time:115.6ms, loops:7
    ’→
    ’→ |
    ’→
    ’→ | 555 Bytes | N/A  |
|   ├─IndexRangeScan_9(Build)       | 90000.00 | 0       | cop[tikv] | table:t2, index:t1_id(
    ’→ t1_id) | time:114.3ms, loops:7, cop_task: {num: 7, max: 42ms, min: 1.3ms, avg: 16.2ms,
    ’→ p95: 42ms, tot_proc: 71ms, rpc_num: 7, rpc_time: 113.2ms, copr_cache_hit_ratio: 0.29,
    ’→ distsql_concurrency: 15}, tikv_task:{proc max:37ms, min:0s, avg: 11.3ms, p80:20ms, p95:37
    ’→ ms, iters:7, tasks:7}, scan_detail: {total_keys: 9296, get_snapshot_time: 141.9µs,
    ’→ rocksdb: {block: {cache_hit_count: 18592}}}  | range: decided by [eq(test.t2.t1_id, test.
    ’→ t1.id)], keep order:false                                                          | N/A
```

```
    ’→       | N/A  |
|   └─TableRowIDScan_10(Probe)      | 90000.00 | 0       | cop[tikv] | table:t2
    ’→                       |
    ’→
    ’→ | keep order:false
    ’→
    ’→ | N/A       | N/A  |
+--
    ’→  -------------------------------+----------+---------+-----------+------------------------
    ’→


+--
    ’→  ----------------------------+----------+---------+-----------+--------------+--------------
    ’→
| id                            | estRows  | actRows | task      | access object | execution info
    ’→
    ’→ | operator info                                    | memory  | disk    |
+--
    ’→  ----------------------------+----------+---------+-----------+--------------+--------------
    ’→
| HashJoin_32                   | 90000.00 | 0   | root  |                         | time:320.2ms,
    ’→ loops:1, build_hash_table:{total:19.3ms, fetch:16.8ms, build:2.52ms}, probe:{concurrency
    ’→ :5, total:1.6s, max:320.1ms, probe:16.1ms, fetch:1.58s}
    ’→
    ’→ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.0 MB | 0 Bytes |
| ├─TableReader_35(Build)       | 9955.54  | 10000   | root      |                | time:18.6ms,
    ’→ loops:12, cop_task: {num: 11, max: 713.8µs, min: 197.3µs, avg: 368.5µs, p95: 713.8µs,
    ’→ rpc_num: 11, rpc_time: 3.83ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ’→
    ’→ | data:Selection_34                                | 14.9 MB | N/A     |
| │ └─Selection_34              | 9955.54  | 10000   | cop[tikv] |                | tikv_task:{
    ’→ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ’→ scan_detail: {get_snapshot_time: 178.9µs, rocksdb: {block: {}}}
    ’→
    ’→ | eq(test.t1.int_col, 1)                           | N/A     | N/A     |
| │   └─TableFullScan_33        | 71010.00 | 71010   | cop[tikv] | table:t1      | tikv_task:{
    ’→ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ’→
    ’→ | keep order:false                                 | N/A     | N/A     |
| └─TableReader_37(Probe)       | 90000.00 | 90000   | root      |                | time:304.4ms,
    ’→ loops:91, cop_task: {num: 24, max: 114ms, min: 251.1µs, avg: 33.1ms, p95: 110.4ms,
    ’→ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 492ms, rpc_num: 24, rpc_time: 793ms,
    ’→  copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ’→                                                                                | data:
    ’→ TableFullScan_36                            | 58.6 MB | N/A     |
```

```
|   └─TableFullScan_36          | 90000.00 | 90000    | cop[tikv] | table:t2       | tikv_task:{
    ′→ proc max:38ms, min:3ms, avg: 14.1ms, p80:23ms, p95:35ms, iters:181, tasks:24},
    ′→ scan_detail: {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys:
    ′→ 139497, get_snapshot_time: 577.2µs, rocksdb: {delete_skipped_count: 44208,
    ′→ key_skipped_count: 253431, block: {cache_hit_count: 3527}}}  | keep order:false
    ′→                                  | N/A      | N/A      |
+--
    ′→  ---------------------------+----------+---------+-----------+--------------+-------------
    ′→
+--
    ′→  ---------------------------+----------+---------+-----------+--------------+-------------
    ′→
| id                            | estRows   | actRows | task       | access object | execution info
    ′→
    ′→ | operator info                                        | memory   | disk     |
+--
    ′→  ---------------------------+----------+---------+-----------+--------------+-------------
    ′→
| HashJoin_33                   | 90000.00 | 0    | root   |                        | time:306.3ms,
    ′→ loops:1, build_hash_table:{total:20.5ms, fetch:17.1ms, build:3.45ms}, probe:{concurrency
    ′→ :5, total:1.53s, max:305.9ms, probe:17.1ms, fetch:1.51s}
    ′→
    ′→ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.0 MB | 0 Bytes |
| ├─TableReader_42(Build)       | 9955.54   | 10000    | root       |                | time:19.6ms,
    ′→ loops:12, cop_task: {num: 11, max: 1.07ms, min: 246.1µs, avg: 600µs, p95: 1.07ms, rpc_num
    ′→ : 11, rpc_time: 6.17ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ′→
    ′→ | data:Selection_41                                    | 19.7 MB | N/A      |
| │ └─Selection_41              | 9955.54   | 10000    | cop[tikv] |                | tikv_task:{
    ′→ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ′→ scan_detail: {get_snapshot_time: 282.9µs, rocksdb: {block: {}}}
    ′→
    ′→ | eq(test.t1.int_col, 1)                               | N/A      | N/A      |
| │   └─TableFullScan_40        | 71010.00 | 71010    | cop[tikv] | table:t1       | tikv_task:{
    ′→ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ′→
    ′→ | keep order:false                                     | N/A      | N/A      |
| └─TableReader_44(Probe)       | 90000.00 | 90000    | root       |                | time:289.2ms,
    ′→ loops:91, cop_task: {num: 24, max: 108.2ms, min: 252.8µs, avg: 31.3ms, p95: 106.1ms,
    ′→ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 445ms, rpc_num: 24, rpc_time: 750.4
    ′→ ms, copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ′→                                                                          | data:
    ′→ TableFullScan_43                              | 58.6 MB | N/A      |
|   └─TableFullScan_43          | 90000.00 | 90000    | cop[tikv] | table:t2       | tikv_task:{
```

```
    '→ proc max:31ms, min:3ms, avg: 13.3ms, p80:24ms, p95:30ms, iters:181, tasks:24},
    '→ scan_detail: {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys:
    '→ 139497, get_snapshot_time: 730.2µs, rocksdb: {delete_skipped_count: 44208,
    '→ key_skipped_count: 253431, block: {cache_hit_count: 3527}}}  | keep order:false
    '→                                  | N/A      | N/A      |
+--
    '→  ---------------------------+----------+---------+-----------+---------------+--------------------------
    '→
```

> **注意：**
>
> 在上方示例中，SQL 优化器之所以选择了性能较差的 Hash Join 算法，而不是Index Join 算法，原因在于查询优化是一个 NP 完全问题，可能会选择不太理想的计划。如果需要频繁调用这个查询，建议通过执行计划管理的方式将 Hint 与 SQL 语句绑定，这样要比在发送给TiDB 的 SQL 语句中插入 Hint 更容易管理。

Index Join 相关算法

如果使用 Hint INL_JOIN 进行Index Join 操作，TiDB 会在连接外表之前创建一个中间结果的 Hash Table。TiDB 同样也支持使用 Hint INL_HASH_JOIN 在外表上建 Hash Table。以上所述的 Index Join 相关算法都由 SQL 优化器自动选择。

配置

Index Join 算法的性能受以下系统变量影响：

- tidb_index_join_batch_size（默认值：25000）- index lookup join 操作的batch 大小。
- tidb_index_lookup_join_concurrency（默认值：4）- 可以并发执行的index lookup 任务数。

8.3.2.5.2 Hash Join

在 Hash Join 操作中，TiDB 首先读取 Build 端的数据并将其缓存在 Hash Table 中，然后再读取 Probe 端的数据，使用 Probe 端的数据来探查 Hash Table 以获得所需行。与Index Join 算法相比，Hash Join 要消耗更多内存，但如果需要连接的行数很多，运行速度会比Index Join 快。TiDB 中的 Hash Join 算子是多线程的，并且可以并发执行。

下面是一个 Hash Join 示例：

```
EXPLAIN SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    '→                          +           +           +               +
    '→
| id                          | estRows    | task      | access object | operator info
    '→                                  |
```

```
+--
    '→                                +           +           +               +
    '→
| HashJoin_27                  | 142020.00 | root      |               | inner join, equal:[eq(
    '→ test.t1.id, test.t2.id)] |
| ├─TableReader_29(Build)      | 142020.00 | root      |               | data:TableFullScan_28
    '→                            |
| │ └─TableFullScan_28         | 142020.00 | cop[tikv] | table:t1      | keep order:false
    '→                                 |
| └─TableReader_31(Probe)      | 180000.00 | root      |               | data:TableFullScan_30
    '→                            |
|   └─TableFullScan_30         | 180000.00 | cop[tikv] | table:t2      | keep order:false
    '→                                 |
+--
    '→                                +           +           +               +
    '→
5 rows in set (0.00 sec)
```

TiDB 会按照以下顺序执行 HashJoin_27 算子：

1. 将 Build 端数据缓存在内存中。
2. 根据缓存数据在 Build 端构造一个 Hash Table。
3. 读取 Probe 端的数据。
4. 使用 Probe 端的数据来探查 Hash Table。
5. 将符合条件的结果返回给用户。

EXPLAIN 返回结果中的 operator info 一列记录了 HashJoin_27 的其他信息，包括该查询是 Inner Join 还是 Outer Join 以及 Join 的条件是什么等。在上面给出的示例中，该查询为 Inner Join，Join 条件是 equal:[eq(test.t1.id, '→ test.t2.id)]，与查询语句中的 WHERE t1.id = t2.id 部分对应。下面例子中其他几个 Join 算子的 operator info 和此处类似。

运行数据

如果在执行操作时，内存使用超过了 tidb_mem_quota_query 规定的值（默认为 1 GB），且 tidb_enable_tmp_storage_on_oom '→ 的值为 ON（默认为 ON），那么 TiDB 会尝试使用临时存储，在磁盘上创建 Hash Join 的 Build 端。EXPLAIN '→ ANALYZE 返回结果中的 execution info 一栏记录了有关内存使用情况等运行数据。下面的例子展示了 tidb_mem_quota_query 的值分别设为 1 GB（默认）及 500 MB 时，EXPLAIN ANALYZE 的返回结果（当内存配额设为 500 MB 时，磁盘用作临时存储区）：

```
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
SET tidb_mem_quota_query=500 * 1024 * 1024;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    '→  ---------------------------+-----------+---------+-----------+---------------+--------------------------
    '→
```

```
| id                          | estRows    | actRows | task      | access object | execution info
    ’→
    ’→ | operator info                                  | memory                | disk     |
+--
     ’→ --------------------------+-----------+---------+-----------+---------------+------------+-------------
    ’→
| HashJoin_27                 | 142020.00 | 71010   | root      |               | time:647.508572
    ’→ ms, loops:72, build_hash_table:{total:579.254415ms, fetch:566.91012ms, build:12.344295ms
    ’→ }, probe:{concurrency:5, total:3.23315006s, max:647.520113ms, probe:330.884716ms, fetch
    ’→ :2.902265344s}                                          | inner join, equal:[eq(test.
    ’→ t1.id, test.t2.id)] | 209.61642456054688 MB | 0 Bytes |
| ├─TableReader_29(Build)     | 142020.00 | 71010   | root      |               | time
    ’→ :567.088247ms, loops:72, cop_task: {num: 2, max: 569.809411ms, min: 369.67451ms, avg:
    ’→ 469.74196ms, p95: 569.809411ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc: 400
    ’→ ms, rpc_num: 2, rpc_time: 939.447231ms, copr_cache_hit_ratio: 0.00}    | data:
    ’→ TableFullScan_28                            | 210.2100534439087 MB  | N/A      |
| │ └─TableFullScan_28        | 142020.00 | 71010   | cop[tikv] | table:t1      | proc max:64ms,
    ’→  min:48ms, p80:64ms, p95:64ms, iters:79, tasks:2
    ’→
    ’→ | keep order:false                                  | N/A                   | N/A      |
| └─TableReader_31(Probe)     | 180000.00 | 90000   | root      |               | time
    ’→ :337.233636ms, loops:91, cop_task: {num: 3, max: 569.790741ms, min: 332.758911ms, avg:
    ’→ 421.543165ms, p95: 569.790741ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ’→ 500ms, rpc_num: 3, rpc_time: 1.264570696s, copr_cache_hit_ratio: 0.00} | data:
    ’→ TableFullScan_30                            | 267.1126985549927 MB  | N/A      |
|   └─TableFullScan_30        | 180000.00 | 90000   | cop[tikv] | table:t2      | proc max:84ms,
    ’→  min:72ms, p80:84ms, p95:84ms, iters:102, tasks:3
    ’→
    ’→ | keep order:false                                  | N/A                   | N/A      |
+--
     ’→ --------------------------+-----------+---------+-----------+---------------+------------+-------------
    ’→
5 rows in set (0.65 sec)

Query OK, 0 rows affected (0.00 sec)

+--
     ’→ --------------------------+-----------+---------+-----------+---------------+------------+-------------
    ’→
| id                          | estRows    | actRows | task      | access object | execution info
    ’→
    ’→ | operator info                                  | memory                | disk
    ’→                 |
+--
     ’→ --------------------------+-----------+---------+-----------+---------------+------------+-------------
```

```
    '→
| HashJoin_27                 | 142020.00 | 71010    | root      |               | time:963.983353
    '→ ms, loops:72, build_hash_table:{total:775.961447ms, fetch:503.789677ms, build:272.17177ms
    '→ }, probe:{concurrency:5, total:4.805454793s, max:963.973133ms, probe:922.156835ms, fetch
    '→ :3.883297958s}                                          | inner join, equal:[eq(test.t1.
    '→ id, test.t2.id)] | 93.53974533081055 MB  | 210.7459259033203 MB |
| ├─TableReader_29(Build)     | 142020.00 | 71010    | root      |               | time
    '→ :504.062018ms, loops:72, cop_task: {num: 2, max: 509.276857ms, min: 402.66386ms, avg:
    '→ 455.970358ms, p95: 509.276857ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc:
    '→ 384ms, rpc_num: 2, rpc_time: 911.893237ms, copr_cache_hit_ratio: 0.00}  | data:
    '→ TableFullScan_28                          | 210.20934200286865 MB | N/A
    '→ |
| │ └─TableFullScan_28        | 142020.00 | 71010    | cop[tikv] | table:t1      | proc max:88ms,
    '→  min:72ms, p80:88ms, p95:88ms, iters:79, tasks:2
    '→
    '→ | keep order:false                                | N/A                   | N/A
    '→                   |
| └─TableReader_31(Probe)     | 180000.00 | 90000    | root      |               | time
    '→ :363.058382ms, loops:91, cop_task: {num: 3, max: 412.659191ms, min: 358.489688ms, avg:
    '→ 391.463008ms, p95: 412.659191ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    '→ 484ms, rpc_num: 3, rpc_time: 1.174326746s, copr_cache_hit_ratio: 0.00} | data:
    '→ TableFullScan_30                          | 267.11340618133545 MB | N/A
    '→ |
|   └─TableFullScan_30        | 180000.00 | 90000    | cop[tikv] | table:t2      | proc max:92ms,
    '→  min:64ms, p80:92ms, p95:92ms, iters:102, tasks:3
    '→
    '→ | keep order:false                                | N/A                   | N/A
    '→                   |
+--
    '→  ---------------------------+-----------+---------+-----------+---------------+-------------------------
    '→
5 rows in set (0.98 sec)
```

配置

Hash Join 算法的性能受以下系统变量影响：

- tidb_mem_quota_query（默认值：1GB）- 如果某条查询的内存消耗超出了配额，TiDB 会尝试将 Hash Join 的 Build 端移到磁盘上以节省内存。
- tidb_hash_join_concurrency（默认值：5）- 可以并发执行的 Hash Join 任务数量。

8.3.2.5.3 Merge Join

Merge Join 是一种特殊的Join 算法。当两个关联表要Join 的字段需要按排好的顺序读取时，适用 Merge Join 算法。由于 Build 端和 Probe 端的数据都会读取，这种算法的Join 操作是流式的，类似“拉链式合并”的高效版。Merge Join 占用的内存要远低于 Hash Join，但 Merge Join 不能并发执行。

下面是一个使用 Merge Join 的例子：

```
EXPLAIN SELECT /*+ MERGE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    '→                              +           +           +               +
    '→
| id                          | estRows    | task      | access object | operator info
    '→                                        |
+--
    '→ ----------------------------+-----------+-----------+---------------+
    '→
| MergeJoin_7                 | 142020.00 | root      |               | inner join, left key:test
    '→ .t1.id, right key:test.t2.id |
| ├─TableReader_12(Build)     | 180000.00 | root      |               | data:TableFullScan_11
    '→                                    |
| │ └─TableFullScan_11        | 180000.00 | cop[tikv] | table:t2      | keep order:true
    '→                                     |
| └─TableReader_10(Probe)     | 142020.00 | root      |               | data:TableFullScan_9
    '→                                   |
|   └─TableFullScan_9         | 142020.00 | cop[tikv] | table:t1      | keep order:true
    '→                                     |
+--
    '→                              +           +           +               +
    '→
5 rows in set (0.00 sec)
```

TiDB 会按照以下顺序执行 Merge Join 算子：

1. 从 Build 端把一个Join Group 的数据全部读取到内存中。
2. 读取 Probe 端的数据。
3. 将 Probe 端的每行数据与 Build 端的一个完整Join Group 比较，依次查看是否匹配（除了满足等值条件以外，还有其他非等值条件，这里的“匹配”主要是指查看是否满足非等值条件）。Join Group 指的是所有 Join Key 上值相同的数据。

8.3.2.5.4 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.6 用 EXPLAIN 查看子查询的执行计划

TiDB 会执行多种子查询相关的优化，以提升子查询的执行性能。本文档介绍一些常见子查询的优化方式，以及如何解读 EXPLAIN 语句返回的执行计划信息。

本文档所使用的示例表数据如下：

```
CREATE TABLE t1 (id BIGINT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB,
    ↪ int_col INT NOT NULL DEFAULT 0);
CREATE TABLE t2 (id BIGINT NOT NULL PRIMARY KEY auto_increment, t1_id BIGINT NOT NULL, pad1 BLOB,
    ↪  pad2 BLOB, pad3 BLOB, INDEX(t1_id));
CREATE TABLE t3 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 t1_id INT NOT NULL,
 UNIQUE (t1_id)
);


INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM
    ↪ dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024),  RANDOM_BYTES(1024),  RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
INSERT INTO t3 SELECT NULL, id FROM t1 WHERE id < 1000;

SELECT SLEEP(1);
ANALYZE TABLE t1, t2, t3;
```

8.3.2.6.1 Inner join（无 UNIQUE 约束的子查询）

以下示例中，IN 子查询会从表 t2 中搜索一列ID。为保证语义正确性，TiDB 需要保证 t1_id 列的值具有唯一性。使用 EXPLAIN 可查看到该查询的执行计划去掉重复项并执行 Inner Join 内连接操作：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t2);
```

```
+--
    ↪                                +          +           +                                +
    ↪
| id                              | estRows  | task      | access object                | operator
    ↪  info
    ↪
    ↪ |
+--
    ↪                                +          +           +                                +
    ↪ --------------------------------- ---------- ----------- -------------------------------
| IndexJoin_15                    | 21.11    | root      |                              | inner
    ↪ join, inner:TableReader_12, outer key:test.t2.t1_id, inner key:test.t1.id, equal
    cond:eq( ↪ test.t2.t1_id, test.t1.id) |
| ├─StreamAgg_44(Build)           | 21.11    | root      |                              | group
    ↪ by:test.t2.t1_id, funcs:firstrow(test.t2.t1_id)->test.t2.t1_id
    ↪                                                    |
| │ └─IndexReader_45              | 21.11    | root      |                              | index:
    ↪ StreamAgg_34
    ↪
    ↪ |
| │   └─StreamAgg_34              | 21.11    | cop[tikv] |                              | group
    ↪ by:test.t2.t1_id,
```

```
    '→
    '→ |
| |     └─IndexFullScan_26        | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep
    '→ order:true
    '→
    '→ |
| └─TableReader_12(Probe)         | 21.11    | root      |                              | data:
    '→ TableRangeScan_11
    '→
    '→ |
|   └─TableRangeScan_11           | 21.11    | cop[tikv] | table:t1                     | range:
    '→ decided by [test.t2.t1_id], keep order:false
    '→                                                                                  |
+--
    '→                              +          +           +                             +
    '→
```

由上述查询结果可知，TiDB 通过索引连接操作| IndexJoin_14 将子查询做了连接转化。该执行计划首先在 TiKV 侧通过索引扫描算子└─IndexFullScan_31 读取 t2.t1_id 列的值，然后由└─StreamAgg_39 算子的部分任务在 TiKV 中对 t1_id 值进行去重，然后采用├─StreamAgg_49(Build) 算子的部分任务在TiDB 中对 t1_id 值再次进行去重，去重操作由聚合函数 firstrow(test.t2.t1_id) 执行；之后将操作结果与 t1 表的主键相连接，连接条件是 eq(test.t1.id, test.t2.t1_id)。

8.3.2.6.2 Inner join（有 UNIQUE 约束的子查询）

在上述示例中，为了确保 t1_id 值在与表 t1 连接前具有唯一性，需要执行聚合运算。在以下示例中，由于 UNIQUE 约束已能确保 t3.t1_id 列值的唯一：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t3);
```

```
+--
    '→                              +          +           +                             +
    '→
| id                          | estRows | task      | access object                | operator
    '→ info
    '→
    '→ |
+--
    '→                              +          +           +                             +
    '→ ----------------------------  ----------  ----------  -------------------------------
| IndexJoin_18                | 999.00 | root  |                                | inner join,
    '→  inner:TableReader_15, outer key:test.t3.t1_id, inner key:test.t1.id, equal cond:eq(test.
    '→ t3.t1_id, test.t1.id) |
| ├─IndexReader_41(Build)     | 999.00  | root      |                               | index:
    '→ IndexFullScan_40
```

```
    '→
    '→ |
| | └─IndexFullScan_40         | 999.00   | cop[tikv] | table:t3, index:t1_id(t1_id) | keep order:
    '→ false
    '→
    '→ |
| └─TableReader_15(Probe)      | 999.00   | root      |                              | data:
    '→ TableRangeScan_14
    '→
    '→ |
|   └─TableRangeScan_14        | 999.00   | cop[tikv] | table:t1                     | range:
    '→ decided by [test.t3.t1_id], keep order:false
    '→                                                                                 |
+--
    '→                              +          +           +                              +
    '→
```

从语义上看，因为约束保证了 `t3.t1_id` 列值的唯一性，TiDB 可以直接执行 `INNER JOIN` 查询。

8.3.2.6.3 Semi Join（关联查询）

在前两个示例中，通过 `StreamAgg` 聚合操作或通过 `UNIQUE` 约束保证子查询数据的唯一性之后，TiDB 才能够执行 `Inner Join` 操作。这两种连接均使用了 `Index Join`。

下面的例子中，TiDB 优化器则选择了一种不同的执行计划：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t2 WHERE t1_id != t1.int_col);
```

```
+--
    '→                              +          +           +                              +
    '→
| id                           | estRows   | task       | access object                | operator
    '→ info
    '→
    '→ |
+--
    '→                              +          +           +                              +
    '→ ------------------------------ ----------- ------------ ------------------------------
| MergeJoin_9                  | 45446.40 | root      |                              | semi join,
    '→ left key:test.t1.id, right key:test.t2.t1_id, other cond:ne(test.t2.t1_id, test.t1.
    '→ int_col) |
| ├─IndexReader_24(Build)      | 90000.00 | root      |                               | index:
    '→ IndexFullScan_23
    '→                                                                                  |
| | └─IndexFullScan_23         | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep order
    '→ :true
    '→                                                                                        |
```

```
| └─TableReader_22(Probe)       | 56808.00 | root      |                    | data:
    ↪ Selection_21
    ↪                                                                              |
|   └─Selection_21              | 56808.00 | cop[tikv] |                    | ne(test.t1
    ↪ .id, test.t1.int_col)
    ↪                                                               |
|     └─TableFullScan_20        | 71010.00 | cop[tikv] | table:t1           | keep order
    ↪ :true
    ↪                                                                               |
+--
    ↪                          +          +            +                          +
    ↪ ---------------------------------------
    ↪
```

由上述查询结果可知，TiDB 执行了 Semi Join。不同于 Inner Join，Semi Join 仅允许右键(t2.t1_id) 上的第一个值，也就是该操作将去除 Join 算子任务中的重复数据。Join 算法也包含 Merge Join，会按照排序顺序同时从左侧和右侧读取数据，这是一种高效的 Zipper Merge。

可以将原语句视为关联子查询，因为它引入了子查询外的 t1.int_col 列。然而，EXPLAIN 语句的返回结果显示的是关联子查询去关联后的执行计划。条件 t1_id != t1.int_col 会被重写为 t1.id != t1.int_col。TiDB 可以从表 t1 中读取数据并且在└─Selection_21 中执行此操作，因此这种去关联和重写操作会极大提高执行效率。

8.3.2.6.4 Anti Semi Join（NOT IN 子查询）

在以下示例中，除非子查询中存在 t3.t1_id，否则该查询将（从语义上）返回表 t3 中的所有行：

```
EXPLAIN SELECT * FROM t3 WHERE t1_id NOT IN (SELECT id FROM t1 WHERE int_col < 100);
```

```
+--
    ↪                          +          +            +                +
    ↪ --------------------------------------------------
    ↪
| id                           | estRows | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪                          +          +            +                +
    ↪ ----------------------------- ---------- ------------ ---------------- --------------------
| IndexJoin_16                 | 799.20 | root  |                  | anti semi join, inner:
    ↪ TableReader_12, outer key:test.t3.t1_id, inner key:test.t1.id, equal cond:eq(test.t3.
    ↪ t1_id, test.t1.id) |
| ├─TableReader_28(Build)       | 999.00  | root       |                | data:TableFullScan_27
    ↪
    ↪ |
| │ └─TableFullScan_27          | 999.00  | cop[tikv] | table:t3       | keep order:false
    ↪
    ↪ |
```

```
| └─TableReader_12(Probe)      | 999.00  | root      |               | data:Selection_11
    '→
    '→ |
|   └─Selection_11             | 999.00  | cop[tikv] |               | lt(test.t1.int_col, 100)
    '→
    '→ |
|     └─TableRangeScan_10      | 999.00  | cop[tikv] | table:t1      | range: decided by [test.t3
    '→ .t1_id], keep order:false
    '→                                                                                  |
+--
    '→                           +         +           +               +
    '→
```

上述查询首先读取了表 t3，然后根据主键开始探测 (probe) 表 t1。连接类型是 anti semi join，即反半连接：之所以使用anti，是因为上述示例有不存在匹配值（即 NOT IN）的情况；使用 Semi Join 则是因为仅需要匹配第一行后就可以停止查询。

8.3.2.6.5 Null-Aware Semi Join（IN 和 = ANY 子查询）

IN 和= ANY 的集合运算符号具有特殊的三值属性（true、false 和 NULL）。这意味着在该运算符所转化得到的 Join 类型中需要对Join key 两侧的 NULL 进行特殊的感知和处理。

IN 和 = ANY 算子引导的子查询会分别转为 Semi Join 和 Left Outer Semi Join。在上述Semi Join 小节中，示例中Join key 两侧的列 test.t1.id 和 test.t2.t1_id 都为 not NULL 属性，所以 Semi Join 本身不需要 Null-Aware 的性质来辅助运算，即不需要特殊处理 NULL。当前 TiDB 对于 Null-Aware Semi Join 没有特定的优化，其实现本质都是基于笛卡尔积加过滤(filter) 的模式。以下为 Null-Aware Semi Join 的例子：

```
CREATE TABLE t(a INT, b INT);
CREATE TABLE s(a INT, b INT);
EXPLAIN SELECT (a,b) IN (SELECT * FROM s) FROM t;
EXPLAIN SELECT * FROM t WHERE (a,b) IN (SELECT * FROM s);
```

```
tidb> EXPLAIN SELECT (a,b) IN (SELECT * FROM s) FROM t;
+--
    '→                           +         +           +               +
    '→
| id                           | estRows | task      | access object | operator info
    '→                                                                              |
+--
    '→                           +         +           +               +
    '→ ----------------------------- ---------- ----------- ----------------
| HashJoin_8                   | 1.00    | root      |               | CARTESIAN left outer semi
    '→ join, other cond:eq(test.t.a, test.s.a), eq(test.t.b, test.s.b) |
| ├─TableReader_12(Build)      | 1.00    | root      |               | data:TableFullScan_11
    '→                                                                        |
| │ └─TableFullScan_11         | 1.00    | cop[tikv] | table:s       | keep order:false, stats:
    '→ pseudo                                                                   |
```

```
| └─TableReader_10(Probe)      | 1.00    | root      |               | data:TableFullScan_9
    ’→                                                                          |
|   └─TableFullScan_9          | 1.00    | cop[tikv] | table:t       | keep order:false, stats:
    ’→ pseudo                                                                   |
+--
    ’→ ----------------------------+----------+-----------+----------------+
    ’→
5 rows in set (0.00 sec)

tidb> EXPLAIN SELECT * FROM t WHERE (a,b) IN (SELECT * FROM s);
+--
    ’→                             +          +           +                +
    ’→ -------------------------------- ---------- ----------- ----------------
    ’→
| id                           | estRows | task      | access object | operator info
    ’→                                                                                  |
+--
    ’→                             +          +           +                +
    ’→ -------------------------------- ---------- ----------- ----------------
| HashJoin_11                  | 1.00    | root      |               | inner join, equal:[eq(test
    ’→ .t.a, test.s.a) eq(test.t.b, test.s.b)]                                  |
| ├─TableReader_14(Build)      | 1.00    | root      |               | data:Selection_13
    ’→                                                                                  |
| │ └─Selection_13             | 1.00    | cop[tikv] |               | not(isnull(test.t.a)),
    ’→ not(isnull(test.t.b))                                                            |
| │   └─TableFullScan_12       | 1.00    | cop[tikv] | table:t       | keep order:false, stats:
    ’→ pseudo                                                                   |
| └─HashAgg_17(Probe)          | 1.00    | root      |               | group by:test.s.a, test.s
    ’→ .b, funcs:firstrow(test.s.a)->test.s.a, funcs:firstrow(test.s.b)->test.s.b |
|   └─TableReader_24           | 1.00    | root      |               | data:Selection_23
    ’→                                                                                  |
|     └─Selection_23           | 1.00    | cop[tikv] |               | not(isnull(test.s.a)),
    ’→ not(isnull(test.s.b))                                                            |
|       └─TableFullScan_22     | 1.00    | cop[tikv] | table:s       | keep order:false, stats:
    ’→ pseudo                                                                   |
+--
    ’→                             +          +           +                +
                                                                                                    ----------
    ’→
8 rows in set (0.01 sec)
```

第一个查询 `EXPLAIN SELECT (a,b)IN (SELECT * FROM s)FROM t;` 中，由于`t` 表和`s` 表的a、b 列都是 NULLABLE 的，所以 `IN` 子查询所转化的 Left Outer Semi Join 是具有 Null-Aware 性质的。具体实现是先进行笛卡尔积，然后将 `IN` 或 `= ANY` 所连接的列作为普通等值条件放到 other condition 进行过滤（filter）。

第二个查询 `EXPLAIN SELECT * FROM t WHERE (a,b)IN (SELECT * FROM s);` 中，由于`t` 表和`s` 表的a、b 列都是 NULLABLE 的，`IN` 子查询本应该转为具有 Null-Aware 性质的 Semi Join，但当前TiDB 进行了优化，直接将 Semi Join 转为了Inner Join + Aggregate 的方式来实现。这是因为在非 scalar 输出的 `IN` 子查询中，`NULL` 和 `false` 是等效

的。下推过滤的 NULL 行导致了 WHERE 子句的否定语义，因此可以事先忽略这些行。

> 注意：
>
> Exists 操作符也会被转成 Semi Join，但是 Exists 操作符号本身不具有集合运算 Null-Aware 的性质。

8.3.2.6.6 Null-Aware Anti Semi Join（NOT IN 和 != ALL 子查询）

NOT IN 和 != ALL 的集合运算运算具有特殊的三值属性（true、false 和 NULL）。这意味着在其所转化得到的 Join 类型中需要对Join key 两侧的 NULL 进行特殊的感知和处理。

NOT IN 和 != ALL 算子引导的子查询会对应地转为Anti Semi Join 和 Anti Left Outer Semi Join。在上述的Anti Semi Join 小节中，由于示例中Join key 两侧的列 test.t3.t1_id 和 test.t1.id 都是 not NULL 属性的，所以Anti Semi Join 本身不需要 Null-Aware 的性质来辅助计算，即不需要特殊处理 NULL。

在 TiDB v6.3.0 版本，TiDB 引入了针对 Null-Aware Anti Join (NAAJ) 的如下特殊优化：

- 利用 Null-Aware 的等值条件(NA-EQ) 构建哈希连接

  由于集合操作符引入的等值需要对等值两侧操作符数的 NULL 值做特殊处理，这里称需要 Null-Aware 的等值条件为 NA-EQ 条件。与 v6.3.0 之前版本不同的是，TiDB 不会再将 NA-EQ 条件处理成普通EQ 条件，而是专门放置于Join 后置的 other condition 中，匹配笛卡尔积后再判断结果集的合法性。

  在 TiDB v6.3.0 版本中，NA-EQ 这种弱化的等值条件依然会被用来构建哈希值(Hash Join)，大大减少了匹配时所需遍历的数据量，加速匹配过程。在build 表 DISTINCT 值比例趋近 1 的时候，加速效果更为显著。

- 利用两侧数据源 NULL 值的特殊性质加速匹配过程的返回

  由于Anti Semi Join 自身具有 CNF (Conjunctive normal form) 表达式的属性，其任何一侧出现的 NULL 值都会导致确定的结果。利用这个性质可以来加速整个匹配过程。

以下为 Null-Aware Anti Semi Join 的例子：

```
CREATE TABLE t(a INT, b INT);
CREATE TABLE s(a INT, b INT);
EXPLAIN SELECT (a, b) NOT IN (SELECT * FROM s) FROM t;
EXPLAIN SELECT * FROM t WHERE (a, b) NOT IN (SELECT * FROM s);
```

```
tidb> EXPLAIN SELECT (a, b) NOT IN (SELECT * FROM s) FROM t;
+--
    '→                              +          +           +               +
    '→
| id                          | estRows   | task      | access object | operator info
    '→                                                                          |
+--
    '→                              +          +           +               +
    '→
       ----------------------------  -----------  ------------  ----------------
```

```
| HashJoin_8                    | 10000.00 | root      |               | Null-aware anti left outer
    ↪  semi join, equal:[eq(test.t.b, test.s.b) eq(test.t.a, test.s.a)] |
| ├─TableReader_12(Build)       | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                                      |
| │ └─TableFullScan_11          | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo                                                               |
| └─TableReader_10(Probe)       | 10000.00 | root      |               | data:TableFullScan_9
    ↪                                                                       |
|   └─TableFullScan_9           | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                                                               |
+--
    ↪                              +          +           +               +
    ↪
5 rows in set (0.00 sec)

tidb> EXPLAIN SELECT * FROM t WHERE (a, b) NOT IN (SELECT * FROM s);
+--
    ↪                              +          +           +               +
    ↪
| id                            | estRows  | task      | access object | operator info
    ↪                                                                      |
+--
    ↪ ------------------------------+----------+-----------+---------------+
    ↪
| HashJoin_8                    | 8000.00  | root      |               | Null-aware anti semi join,
    ↪  equal:[eq(test.t.b, test.s.b) eq(test.t.a, test.s.a)] |
| ├─TableReader_12(Build)       | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                                    |
| │ └─TableFullScan_11          | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo                                                             |
| └─TableReader_10(Probe)       | 10000.00 | root      |               | data:TableFullScan_9
    ↪                                                                     |
|   └─TableFullScan_9           | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                                                             |
+--
    ↪                              +          +           +               +
    ↪
5 rows in set (0.00 sec)
```

第一个查询 `EXPLAIN SELECT (a, b)NOT IN (SELECT * FROM s)FROM t;` 中，由于 `t` 表和 `s` 表的 a、b 列都是 NULLABLE 的，所以 `NOT IN` 子查询所转化的 Left Outer Semi Join 是具有 Null-Aware 性质的。不同的是，NAAJ 优化将 NA-EQ 条件也作为了 Hash Join 的连接条件，大大加速了Join 的计算。

第二个查询 `EXPLAIN SELECT * FROM t WHERE (a, b)NOT IN (SELECT * FROM s);` 中，由于t 表和s 表的a、b 列都是 NULLABLE 的，所以 `NOT IN` 子查询所转化的Anti Semi Join 是具有 Null-Aware 性质的。不同的是，NAAJ 优化将 NA-EQ 条件也作为了 Hash Join 的连接条件，大大加速了Join 的计算。

当前TiDB仅针对Anti Semi Join和Anti Left Outer Semi Join实现了NULL感知。目前仅支持Hash Join类型且其build表只能固定为右侧表。

> **注意：**
>
> `Not Exists`操作符也会被转成Anti Semi Join，但是`Not Exists`符号本身不具有集合运算Null-Aware的性质。

8.3.2.6.7 其他类型查询的执行计划

- MPP模式查询的执行计划
- 索引查询的执行计划
- Join查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.7 用EXPLAIN查看聚合查询执行计划

SQL查询中可能会使用聚合计算，可以通过EXPLAIN语句来查看聚合查询的执行计划。本文提供多个示例，以帮助用户理解聚合查询是如何执行的。

SQL优化器会选择以下任一算子实现数据聚合：

- Hash Aggregation
- Stream Aggregation

为了提高查询效率，数据聚合在Coprocessor层和TiDB层均会执行。现有示例如下：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB);
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    '→ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    '→ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    '→ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    '→ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    '→ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    '→ JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    -> JOIN t1 b JOIN t1 c LIMIT 10000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
```

以上示例创建表格 `t1` 并插入数据后，再执行 `SHOW TABLE REGIONS` 语句。从以下 `SHOW TABLE REGIONS` 的执行结果可知，表 `t1` 被切分为多个Region：

```
SHOW TABLE t1 REGIONS;
```

```
+--
    -> ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ->
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    -> WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    -> ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ->
|        64 | t_64_         | t_64_r_31766 |        65 |               1 | 65    |          0 |
    ->          1325 |  102033520 |                   98 |            52797 |
|        66 | t_64_r_31766 | t_64_r_63531 |        67 |               1 | 67    |          0 |
    ->          1325 |   72522521 |                  104 |            78495 |
|        68 | t_64_r_63531 | t_64_r_95296 |        69 |               1 | 69    |          0 |
    ->          1325 |          0 |                  104 |            95433 |
|         2 | t_64_r_95296 |              |         3 |               1 | 3     |          0 |
    ->          1501 |          0 |                   81 |            63211 |
+--
    -> ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ->
4 rows in set (0.00 sec)
```

使用 EXPLAIN 查看以下聚合语句的执行计划。可以看到 └─StreamAgg_8 算子先执行在TiKV 内每个Region 上，然后 TiKV 的每个Region 会返回一行数据给TiDB，TiDB 在 StreamAgg_16 算子上对每个Region 返回的数据进行聚合：

```
EXPLAIN SELECT COUNT(*) FROM t1;
```

```
+--
    '→ -------------------------+-----------+-----------+----------------+                                    +
    '→
| id                          | estRows   | task      | access object | operator info
    '→                    |
+--
    '→ ---------------------------+           +           +               +                                    +
    '→
| StreamAgg_16                | 1.00      | root      |               | funcs:count(Column#7)->
    '→ Column#5 |
| └─TableReader_17             | 1.00      | root      |               | data:StreamAgg_8
    '→                 |
|   └─StreamAgg_8              | 1.00      | cop[tikv] |               | funcs:count(1)->Column#7
    '→        |
|     └─TableFullScan_15       | 242020.00 | cop[tikv] | table:t1      | keep order:false
    '→                 |
+--    --------------------------- ------------ ------------ ----------------                                    +
    '→                            +           +           +               +
    '→
4 rows in set (0.00 sec)
```

同样，通过执行 EXPLAIN ANALYZE 语句可知，actRows 与 SHOW TABLE REGIONS 返回结果中的Region 数匹配，这是因为执行使用了 TableFullScan 全表扫并且没有二级索引：

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM t1;
```

```
+--
    '→  -------------------------+-----------+---------+-----------+--------------+--------------------------
    '→
| id                          | estRows   | actRows | task      | access object | execution info
    '→
    '→ | operator info                   | memory    | disk |
+--
    '→  -------------------------+-----------+---------+-----------+--------------+--------------------------
    '→
| StreamAgg_16                | 1.00      | 1       | root      |               | time:12.609575ms
    '→ , loops:2
    '→
    '→ | funcs:count(Column#7)->Column#5 | 372 Bytes | N/A  |
```

```
| └─TableReader_17          | 1.00  | 4   | root  |                                | time:12.605155
    ′→ ms, loops:2, cop_task: {num: 4, max: 12.538245ms, min: 9.256838ms, avg: 10.895114ms, p95:
    ′→  12.538245ms, max_proc_keys: 31765, p95_proc_keys: 31765, tot_proc: 48ms, rpc_num: 4,
    ′→ rpc_time: 43.530707ms, copr_cache_hit_ratio: 0.00} | data:StreamAgg_8                 |
    ′→ 293 Bytes | N/A  |
|   └─StreamAgg_8            | 1.00      | 4       | cop[tikv] |                | proc max:12ms,
    ′→ min:12ms, p80:12ms, p95:12ms, iters:122, tasks:4
    ′→
    ′→ | funcs:count(1)->Column#7         | N/A       | N/A  |
|     └─TableFullScan_15     | 242020.00 | 121010  | cop[tikv] | table:t1       | proc max:12ms,
    ′→ min:12ms, p80:12ms, p95:12ms, iters:122, tasks:4
    ′→
    ′→ | keep order:false                | N/A       | N/A  |
+--
    ′→  -------------------------+-----------+---------+-----------+---------------+-----------------------------
    ′→
4 rows in set (0.01 sec)
```

8.3.2.7.1 Hash Aggregation

Hash Aggregation 算法在执行聚合时使用 Hash 表存储中间结果。此算法采用多线程并发优化，执行速度快，但与 Stream Aggregation 算法相比会消耗较多内存。

下面是一个使用 Hash Aggregation（即 HashAgg 算子）的例子：

```
EXPLAIN SELECT /*+ HASH_AGG() */ count(*) FROM t1;
```

```
+--
    ′→ -------------------------+-----------+-----------+---------------+--------------------------------+
    ′→
| id                        | estRows    | task      | access object | operator info
    ′→                   |
+--
    ′→ -------------------------+-----------+-----------+---------------+--------------------------------+
    ′→
| HashAgg_9                 | 1.00       | root      |               | funcs:count(Column#6)->
    ′→ Column#5 |
| └─TableReader_10          | 1.00       | root      |               | data:HashAgg_5
    ′→                   |
|   └─HashAgg_5             | 1.00       | cop[tikv] |               | funcs:count(1)->Column#6
    ′→        |
|     └─TableFullScan_8     | 242020.00 | cop[tikv] | table:t1      | keep order:false
    ′→                  |
+--                                                                                                  +
    ′→ -------------------------+-----------+-----------+---------------+--------------------------------+
    ′→
```

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

operator info 列显示，用于聚合数据的 Hash 函数为 funcs:count(1)->Column#6。

8.3.2.7.2 Stream Aggregation

Stream Aggregation 算法通常会比 Hash Aggregation 算法占用更少的内存。但是此算法要求数据按顺序发送，以便对依次到达的值实现流式数据聚合。

下面是一个使用 Stream Aggregation 的例子：

```
CREATE TABLE t2 (id INT NOT NULL PRIMARY KEY, col1 INT NOT NULL);
INSERT INTO t2 VALUES (1, 9),(2, 3),(3,1),(4,8),(6,3);
EXPLAIN SELECT /*+ STREAM_AGG() */ col1, count(*) FROM t2 GROUP BY col1;
```

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

Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

+--
    ↪                              +          +          +              +
    ↪
| id                            | estRows   | task      | access object | operator info
    ↪                                                                    |
+--
    ↪                              +          +          +              +
    ↪ ---------------------------- ---------- ---------- ---------------
| Projection_4                  | 8000.00   | root      |               | test.t2.col1, Column#3
    ↪                                                         |
| └─StreamAgg_8                 | 8000.00   | root      |               | group by:test.t2.col1,
    ↪ funcs:count(1)->Column#3, funcs:firstrow(test.t2.col1)->test.t2.col1 |
|   └─Sort_13                   | 10000.00 | root      |               | test.t2.col1
    ↪                                                                     |
|     └─TableReader_12          | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                             |
|       └─TableFullScan_11      | 10000.00 | cop[tikv] | table:t2      | keep order:false, stats:
    ↪ pseudo                                                     |
+--
    ↪                              +          +          +              +
    ↪
5 rows in set (0.00 sec)
```

以上示例中，可以在 col1 上添加索引来消除└─Sort_13 算子。添加索引后，TiDB 就可以按顺序读取数据并消除 └─Sort_13 算子。

```
ALTER TABLE t2 ADD INDEX (col1);
```

```
EXPLAIN SELECT /*+ STREAM_AGG() */ col1, count(*) FROM t2 GROUP BY col1;
```

```
Query OK, 0 rows affected (0.28 sec)

+--
    '→                              +         +           +                              +
    '→
| id                            | estRows | task      | access object                | operator info
    '→                                                                                    |
+--
    '→ ------------------------------+---------+-----------+------------------------------+
    '→
| Projection_4                  | 4.00    | root      |                              | test.t2.col1,
    '→  Column#3                                                                          |
| └─StreamAgg_14                | 4.00    | root      |                              | group by:
    '→ test.t2.col1, funcs:count(Column#4)->Column#3, funcs:firstrow(test.t2.col1)->test.t2.col1
    '→  |
|   └─IndexReader_15            | 4.00    | root      |                              | index:
    '→ StreamAgg_8
    '→                                                                                      |
|     └─StreamAgg_8             | 4.00    | cop[tikv] |                              | group by:
    '→ test.t2.col1, funcs:count(1)->Column#4
    '→                                                  |
|       └─IndexFullScan_13      | 5.00    | cop[tikv] | table:t2, index:col1(col1) | keep order:
    '→ true, stats:pseudo                                                                 |
+--
    '→                              +         +           +                              +
    '→
5 rows in set (0.00 sec)
```

8.3.2.7.3 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.8 用 EXPLAIN 查看带视图的 SQL 执行计划

EXPLAIN 语句返回的结果会显示视图引用的表和索引，而不是视图本身的名称。这是因为视图是一张虚拟表，本身并不存储任何数据。视图的定义会和查询语句的其余部分在 SQL 优化过程中进行合并。

参考bikeshare 数据库示例（英文），以下两个示例查询的执行方式类似：

```
ALTER TABLE trips ADD INDEX (duration);
CREATE OR REPLACE VIEW long_trips AS SELECT * FROM trips WHERE duration > 3600;
EXPLAIN SELECT * FROM long_trips;
EXPLAIN SELECT * FROM trips WHERE duration > 3600;
```

```
Query OK, 0 rows affected (2 min 10.11 sec)

Query OK, 0 rows affected (0.13 sec)

+--
    '→ ------------------------------+------------+-----------+-----------------------------------------+----------
    '→
| id                              | estRows    | task      | access object
    '→                          | operator info                          |
+--
    '→ ------------------------------+------------+-----------+-----------------------------------------+----------
    '→
| IndexLookUp_12                  | 6372547.67 | root      |
    '→                                        |                                         |
| ├─IndexRangeScan_10(Build)      | 6372547.67 | cop[tikv] | table:trips, index:duration(duration
    '→ ) | range:(3600,+inf], keep order:false |
| └─TableRowIDScan_11(Probe)      | 6372547.67 | cop[tikv] | table:trips
    '→                           | keep order:false                       |
+--
    '→ ------------------------------+------------+-----------+-----------------------------------------+----------
    '→
3 rows in set (0.00 sec)

+--
    '→ ------------------------------+-----------+-----------+-----------------------------------------+----------
    '→
| id                              | estRows   | task      | access object                           |
    '→  operator info                           |
+--
    '→ ------------------------------+-----------+-----------+-----------------------------------------+----------
    '→
| IndexLookUp_10                  | 833219.37 | root      |                                         |
    '→                                       |
| ├─IndexRangeScan_8(Build)       | 833219.37 | cop[tikv] | table:trips, index:duration(duration)
    '→ | range:(3600,+inf], keep order:false |
| └─TableRowIDScan_9(Probe)       | 833219.37 | cop[tikv] | table:trips
    '→ | keep order:false                      |
+--
    '→ ------------------------------+-----------+-----------+-----------------------------------------+----------
```

```
    '→
3 rows in set (0.00 sec)
```

同样，该视图中的谓词被下推至基表：

```
EXPLAIN SELECT * FROM long_trips WHERE bike_number = 'W00950';
EXPLAIN SELECT * FROM trips WHERE bike_number = 'W00950';
```

```
+--
    '→ -------------------------------+----------+------------+                                                +-------------
    '→
| id                               | estRows | task       | access object                                |
    '→ operator info                                          |
+--
    '→ -------------------------------+----------+------------+                                                +-------------
    '→
| IndexLookUp_14                   | 3.33     | root       |                                                |
    '→                                                         |
| ├─IndexRangeScan_11(Build)       | 3333.33 | cop[tikv] | table:trips, index:duration(duration) |
    '→  range:(3600,+inf], keep order:false, stats:pseudo |
| └─Selection_13(Probe)            | 3.33     | cop[tikv] |                                                |
    '→  eq(bikeshare.trips.bike_number, "W00950")          |
|   └─TableRowIDScan_12            | 3333.33 | cop[tikv] | table:trips                                  |
    '→  keep order:false, stats:pseudo                      |
+--
    '→ -------------------------------+----------+------------+                                                +-------------
    '→
4 rows in set (0.00 sec)


+--
    '→ ------------------------+--------------+------------+----------------+
    '→
| id                         | estRows      | task       | access object | operator info
    '→                                |
+--
    '→ ------------------------+--------------+------------+----------------+
    '→
| TableReader_7              | 43.00        | root       |               | data:Selection_6
    '→                               |
| └─Selection_6              | 43.00        | cop[tikv] |               | eq(bikeshare.trips.
    '→ bike_number, "W00950") |
|   └─TableFullScan_5        | 19117643.00 | cop[tikv] | table:trips    | keep order:false
    '→                               |
+--
    '→ ------------------------+--------------+------------+----------------+
    '→
```

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

执行以上第一条语句时使用了索引，满足视图定义，接着在TiDB读取行时应用了 `bike_number = 'W00950'` 条件。执行以上第二条语句时，不存在满足该语句的索引，因此使用了 `TableFullScan`。

TiDB 使用的索引可以同时满足视图定义和语句本身，如以下组合索引所示：

```
ALTER TABLE trips ADD INDEX (bike_number, duration);
EXPLAIN SELECT * FROM long_trips WHERE bike_number = 'W00950';
EXPLAIN SELECT * FROM trips WHERE bike_number = 'W00950';
```

```
Query OK, 0 rows affected (2 min 31.20 sec)

+--
    '→ -------------------------------+-----------+------------+-------------------------------------------------------------------
    '→
| id                              | estRows   | task       | access object
    '→                                          | operator info
    '→                                          |
+--
    '→ -------------------------------+-----------+------------+-------------------------------------------------------------------
    '→
| IndexLookUp_13                  | 63725.48 | root       |
    '→                                                          |
    '→                                                          |
| ├─IndexRangeScan_11(Build)      | 63725.48 | cop[tikv] | table:trips, index:bike_number(
    '→ bike_number, duration) | range:("W00950" 3600,"W00950" +inf], keep order:false |
| └─TableRowIDScan_12(Probe)      | 63725.48 | cop[tikv] | table:trips
    '→                                            | keep order:false
    '→                                       |
+--
    '→                                +           +            +
    '→ -------------------------------- ----------- ------------ -------------------------------------------------------------------
3 rows in set (0.00 sec)

+--
    '→                                +           +            +
    '→ -------------------------------- ----------- ------------ -------------------------------------------------------------------
| id                              | estRows   | task       | access object
    '→                                          | operator info                                 |
+--
    '→                                +           +            +
    '→ -------------------------------- ----------- ------------ -------------------------------------------------------------------
| IndexLookUp_10                  | 19117.64 | root       |
    '→                                                          |
    '→                                             |
```

```
| ├─IndexRangeScan_8(Build)      | 19117.64 | cop[tikv] | table:trips, index:bike_number(
    ’→ bike_number, duration) | range:["W00950","W00950"], keep order:false |
| └─TableRowIDScan_9(Probe)      | 19117.64 | cop[tikv] | table:trips
    ’→                                          | keep order:false                          |
+--
    ’→                              +          +           +
    ’→
3 rows in set (0.00 sec)
```

在第一条语句中，TiDB 能够使用组合索引的两个部分 (bike_number, duration)。在第二条语句，TiDB 仅使用了索引 (bike_number, duration) 的第一部分 bike_number。

8.3.2.8.1 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

8.3.2.9 用 EXPLAIN 查看分区查询的执行计划

使用 EXPLAIN 语句可以查看TiDB 在执行查询时需要访问的分区。由于存在分区裁剪，所显示的分区通常只是所有分区的一个子集。本文档介绍了常见分区表的一些优化方式，以及如何解读 EXPLAIN 语句返回的执行计划信息。

本文档所使用的示例数据如下：

```
CREATE TABLE t1 (
 id BIGINT NOT NULL auto_increment,
 d date NOT NULL,
 pad1 BLOB,
 pad2 BLOB,
 pad3 BLOB,
 PRIMARY KEY (id,d)
) PARTITION BY RANGE (YEAR(d))
 ( PARTITION p2016 VALUES LESS THAN
 (2017), PARTITION p2017 VALUES LESS THAN
 (2018), PARTITION p2018 VALUES LESS THAN
 (2019), PARTITION p2019 VALUES LESS THAN
 (2020), PARTITION pmax VALUES LESS THAN
 MAXVALUE
);

INSERT INTO t1 (d, pad1, pad2, pad3) VALUES
```

```
 ('2016-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2016-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2016-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-06-01', RANDOM_BYTES(102), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024));

INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ’→ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ’→ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ’→ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ’→ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;

SELECT SLEEP(1);
ANALYZE TABLE t1;
```

以下示例解释了基于新建分区表 `t1` 的一条语句：

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE d = '2017-06-01';
```

```
+--
    ’→                          +         +           +                          +
    ’→
| id                          | estRows | task      | access object              | operator info
    ’→                          |
+--
    ’→                          +         +           +                          +
    ’→
| StreamAgg_21                | 1.00    | root      |                            | funcs:count(
    ’→ Column#8)->Column#6      |
| └─TableReader_22            | 1.00    | root      |                            | data:
    ’→ StreamAgg_10             |
|   └─StreamAgg_10            | 1.00    | cop[tikv] |                            | funcs:count
    ’→ (1)->Column#8            |
```

```
|     └─Selection_20           | 8.87    | cop[tikv] |                          | eq(test.t1.d,
    '→  2017-06-01 00:00:00.000000) |
|       └─TableFullScan_19     | 8870.00 | cop[tikv] | table:t1, partition:p2017 | keep order:
    '→ false                          |
+--
    '→                              +         +           +                          +
    '→
5 rows in set (0.01 sec)
```

由上述 EXPLAIN 结果可知，从最末尾的└─TableFullScan_19 算子开始，再返回到根部的 StreamAgg_21 算子的执行过程如下：

- TiDB 成功地识别出只需要访问一个分区(p2017)，并将该信息在 access object 列中注明。
- └─TableFullScan_19 算子先对整个分区进行扫描，然后执行 └─Selection_20 算子筛选起始日期为 2017-06-01 00:00:00.000000 的行。
- 之后，└─Selection_20 算子匹配的行在 Coprocessor 中进行流式聚合，Coprocessor 本身就可以理解聚合函数 count。
- 每个 Coprocessor 请求会发送一行数据给TiDB 的 └─TableReader_22 算子，然后将数据在 StreamAgg_21 算子下进行流式聚合，再将一行数据返回给客户端。

以下示例中，分区裁剪不会消除任何分区：

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE YEAR(d) = 2017;
```

```
+--
    '→ ---------------------------------+----------+-----------+--------------------------+----------
    '→
| id                                | estRows  | task      | access object            |
    '→ operator info                    |
+--
    '→                                  +          +           +                          +----------
    '→
| HashAgg_20                        | 1.00     | root      |                          | funcs:
    '→ count(Column#7)->Column#6  |
| └─PartitionUnion_21               | 5.00     | root      |                          |
    '→                                  |
|   ├─StreamAgg_36                  | 1.00     | root      |                          | funcs:
    '→ count(Column#9)->Column#7  |
|   │ └─TableReader_37              | 1.00     | root      |                          | data:
    '→ StreamAgg_25                |
|   │   └─StreamAgg_25              | 1.00     | cop[tikv] |                          | funcs:
    '→ count(1)->Column#9         |
|   │     └─Selection_35            | 6000.00  | cop[tikv] |                          | eq(
    '→ year(test.t1.d), 2017)        |
|   │       └─TableFullScan_34      | 7500.00  | cop[tikv] | table:t1, partition:p2016 | keep
    '→ order:false                 |
```

```
|   ├─StreamAgg_55                  | 1.00      | root      |                                | funcs:
    ′→ count(Column#11)->Column#7 |
|   │ └─TableReader_56               | 1.00      | root      |                                | data:
    ′→ StreamAgg_44                |
|   │   └─StreamAgg_44               | 1.00      | cop[tikv] |                                | funcs:
    ′→ count(1)->Column#11         |
|   │     └─Selection_54             | 14192.00  | cop[tikv] |                                | eq(
    ′→ year(test.t1.d), 2017)        |                                table:t1, partition:p2017
|   │       └─TableFullScan_53       | 17740.00  | cop[tikv] |                                | keep
    ′→ order:false                 |
|   ├─StreamAgg_74                  | 1.00      | root      |                                | funcs:
    ′→ count(Column#13)->Column#7 |
|   │ └─TableReader_75               | 1.00      | root      |                                | data:
    ′→ StreamAgg_63                |
|   │   └─StreamAgg_63               | 1.00      | cop[tikv] |                                | funcs:
    ′→ count(1)->Column#13         |
|   │     └─Selection_73             | 3977.60   | cop[tikv] |                                | eq(
    ′→ year(test.t1.d), 2017)        |                                table:t1, partition:p2018
|   │       └─TableFullScan_72       | 4972.00   | cop[tikv] |                                | keep
    ′→ order:false                 |
|   ├─StreamAgg_93                  | 1.00      | root      |                                | funcs:
    ′→ count(Column#15)->Column#7 |
|   │ └─TableReader_94               | 1.00      | root      |                                | data:
    ′→ StreamAgg_82                |
|   │   └─StreamAgg_82               | 1.00      | cop[tikv] |                                | funcs:
    ′→ count(1)->Column#15         |
|   │     └─Selection_92             | 20361.60  | cop[tikv] | table:t1, partition:p2019      | eq(
    ′→ year(test.t1.d), 2017)        |
|   │       └─TableFullScan_91       | 25452.00  | cop[tikv] |                                | keep
    ′→ order:false                 |
|   └─StreamAgg_112                 | 1.00      | root      |                                | funcs:
    ′→ count(Column#17)->Column#7 |
|     └─TableReader_113              | 1.00      | root      |                                | data:
    ′→ StreamAgg_101               |
|       └─StreamAgg_101              | 1.00      | cop[tikv] |                                | funcs:
    ′→ count(1)->Column#17         |                                table:t1, partition:pmax
|         └─Selection_111            | 8892.80   | cop[tikv] |                                | eq(
    ′→ year(test.t1.d), 2017)        |
|           └─TableFullScan_110      | 11116.00  | cop[tikv] |  +                             | keep
    ′→ order:false                 |
+--
    ′→                                  +           +                                    +----------------------------------
    ′→
27 rows in set (0.00 sec)
```

由上述 EXPLAIN 结果可知：

- TiDB 认为需要访问所有分区 (p2016..pMax)。这是因为 TiDB 将谓词 YEAR (d) = 2017 视为non-sargable。这个问题并非是TiDB 特有的。
- 在扫描每个分区时，Selection 算子将筛选出年份不为 2017 的行。
- 在每个分区上会执行流式聚合，以计算匹配的行数。
- └─PartitionUnion_21 算子会合并访问每个分区后的结果。

8.3.2.9.1 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 索引合并查询的执行计划

8.3.2.10 用 EXPLAIN 查看索引合并的 SQL 执行计划

索引合并是从TiDB v4.0 起引入的一种新的表访问方式。在这种访问方式下，TiDB 优化器可以选择对一张表使用多个索引，并将每个索引的返回结果进行合并。在某些场景下，这种访问方式能够减少大量不必要的数据扫描，提升查询的执行效率。

TiDB 中的索引合并分为交集型和并集型两种类型，分别适用于由 AND 连接的表达式和由 OR 连接的表达式。其中，并集型索引合并在TiDB v4.0 作为实验功能引入，在 v5.4.0 成为正式功能(GA)。交集型索引合并从TiDB v6.5.0 起引入，且必须使用USE_INDEX_MERGE Hint 指定才能使用。

8.3.2.10.1 开启索引合并

在v5.4.0 及以上版本的新建集群中，索引合并默认开启。在其他情况下如果未开启，可将tidb_enable_index_merge ′→ 的值设为 ON 来开启索引合并功能。

```
SET session tidb_enable_index_merge = ON;
```

8.3.2.10.2 示例

```
CREATE TABLE t(a int, b int, c int, d int, INDEX idx_a(a), INDEX idx_b(b), INDEX idx_c(c), INDEX
    ′→ idx_d(d));
```

```
EXPLAIN SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE a = 1 OR b = 1;
+--
    ′→ ------------------------+----------+-----------+---------------+--------------------------------------+
    ′→
| id                         | estRows  | task      | access object | operator info
    ′→                           |
```

```
+--
    '→ ----------------------+----------+-----------+---------------+-----------------------------------+
    '→
| TableReader_7            | 19.99    | root      |               | data:Selection_6
    '→                      |
| └─Selection_6            | 19.99    | cop[tikv] |               | or(eq(test.t.a, 1), eq(test.t
    '→ .b, 1)) |
|   └─TableFullScan_5      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo       |
+--
    '→ ----------------------+----------+-----------+---------------+-----------------------------------+
    '→
EXPLAIN SELECT /*+ USE_INDEX_MERGE(t) */ * FROM t WHERE a > 1 OR b > 1;
+--
    '→ ----------------------------+----------+-----------+                       +-------------------------
    '→
| id                           | estRows | task      | access object           | operator info
    '→                                |
+--
    '→ ----------------------------+----------+-----------+                       +-------------------------
    '→
| IndexMerge_8                 | 5555.56 | root      |                         | type: union
    '→                                     |
| ├─IndexRangeScan_5(Build)    | 3333.33 | cop[tikv] | table:t, index:idx_a(a) | range:(1,+inf
    '→ ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_6(Build)    | 3333.33 | cop[tikv] | table:t, index:idx_b(b) | range:(1,+inf
    '→ ], keep order:false, stats:pseudo |
| └─TableRowIDScan_7(Probe)    | 5555.56 | cop[tikv] | table:t                 | keep order:
    '→ false, stats:pseudo                |
+--
    '→ ----------------------------+----------+-----------+                       +-------------------------
    '→
```

例如，在上述示例中，过滤条件是使用 `OR` 连接的 `WHERE` 子句。在启用索引合并前，每个表只能使用一个索引，不能将`a = 1` 下推到索引`a`，也不能将`b = 1` 下推到索引`b`。当 `t` 中存在大量数据时，全表扫描的效率会很低。

对于以上查询语句，优化器选择了并集型索引合并的方式访问表。在这种访问方式下，优化器可以选择对一张表使用多个索引，并将每个索引的返回结果进行合并，生成以上两个示例中的后一个执行计划。此时的 `IndexMerge_8` 算子的 `operator info` 中的 `type: union` 表示该算子是一个并集型索引合并算子。它有三个子节点，其中 `IndexRangeScan_5` 和 `IndexRangeScan_6` 根据范围扫描得到符合条件的所有 `RowID`，再由 `TableRowIDScan_7` 算子根据这些 `RowID` 精确地读取所有满足条件的数据。

其中对于 `IndexRangeScan/TableRangeScan` 一类按范围进行的扫表操作，`EXPLAIN` 表中 `operator info` 列相比于其他扫表操作，多了被扫描数据的范围这一信息。比如上面的例子中，`IndexRangeScan_13` 算子中的 `range:(1,+inf]` 这一信息表示该算子扫描了从 `1` 到正无穷这个范围的数据。

```
EXPLAIN SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE a > 1 AND b > 1 AND c = 1;  --
    '-> 不使用索引合并

+--
    '-> ------------------------------+---------+-----------+-------------------------+-----------------------------
    '->
| id                            | estRows | task      | access object           | operator info
    '->                              |
+--
    '-> ------------------------------+---------+-----------+-------------------------+-----------------------------
    '->
| IndexLookUp_19                | 1.11    | root      |                         |
    '->                                          |
| ├─IndexRangeScan_16(Build)    | 10.00   | cop[tikv] | table:t, index:idx_c(c) | range:[1,1],
    '-> keep order:false, stats:pseudo |
| └─Selection_18(Probe)         | 1.11    | cop[tikv] |                         | gt(test.t.a,
    '-> 1), gt(test.t.b, 1)            |
|   └─TableRowIDScan_17         | 10.00   | cop[tikv] | table:t                 | keep order:
    '-> false, stats:pseudo              |
+--
    '-> ------------------------------+---------+-----------+-------------------------+-----------------------------
    '->

EXPLAIN SELECT /*+ USE_INDEX_MERGE(t, idx_a, idx_b, idx_c) */ * FROM t WHERE a > 1 AND b > 1 AND
    '-> c = 1;  -- 使用索引合并

+--
    '-> ------------------------------+---------+-----------+-------------------------+
    '->                                                                                 -----------------------------
| id                            | estRows | task      | access object           | operator info
    '->                                    |
+--
    '-> ------------------------------+---------+-----------+-------------------------+
    '->                                                                                 -----------------------------
| IndexMerge_9                  | 1.11    | root      |                         | type:
    '-> intersection                              |
| ├─IndexRangeScan_5(Build)     | 3333.33 | cop[tikv] | table:t, index:idx_a(a) | range:(1,+inf
    '-> ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_6(Build)     | 3333.33 | cop[tikv] | table:t, index:idx_b(b) | range:(1,+inf
    '-> ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_7(Build)     | 10.00   | cop[tikv] | table:t, index:idx_c(c) | range:[1,1],
    '-> keep order:false, stats:pseudo    |
| └─TableRowIDScan_8(Probe)     | 1.11    | cop[tikv] | table:t                 | keep order:
    '-> false, stats:pseudo                |
```

```
+--
    '-> ----------------------------+---------+----------+----------------------+-----------------------------
    '->
```

在如上示例中，过滤条件是使用 AND 连接的 WHERE 子句。在启用索引合并前，只能选择使用 idx_a、idx_b 或 idx_c 三个索引中的一个。

如果三个过滤条件中的其中一个的过滤性非常好，直接选择对应的索引即可达到理想的执行效率。但如果数据分布同时满足以下三种情形，可以考虑使用交集型索引合并：

- 全表的数据量相当大，导致直接读全表的执行效率非常低下
- 每个过滤条件单独的过滤性都不够好，导致 IndexLookUp 使用单个索引的执行效率也不够理想
- 三个过滤条件整体的过滤性非常好

在交集型索引合并访问方式下，优化器可以选择对一张表使用多个索引，并将每个索引的返回结果取交集，生成以上两个示例中的后一个执行计划。此时的 IndexMerge_9 算子的 operator info 中的 type: intersection表示该算子是一个交集型索引合并算子。该执行计划的其它部分和上述并集型索引合并示例类似。

> 注意：
>
> - TiDB 的索引合并特性在 v5.4.0 及之后的版本默认开启，即tidb_enable_index_merge 为 ON。
> - 如果查询中使用了 SQL 优化器 Hint USE_INDEX_MERGE，无论 tidb_enable_index_merge 开关是否开启，都会强制使用索引合并特性。当过滤条件中有无法下推的表达式时，必须使用 Hint USE_INDEX_MERGE 才能开启索引合并。
> - 如果查询有除了全表扫描以外的单索引扫描方式可以选择，优化器不会自动选择索引合并，只能通过 Hint 指定使用索引合并。
> - 索引合并目前无法在临时表上使用。
> - 交集型索引合并目前不会被优化器自动选择，必须使用USE_INDEX_MERGE Hint 指定表名和索引名时才会被选择。

8.3.2.10.3 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划

8.3.3 SQL 优化流程

8.3.3.1 SQL 优化流程简介

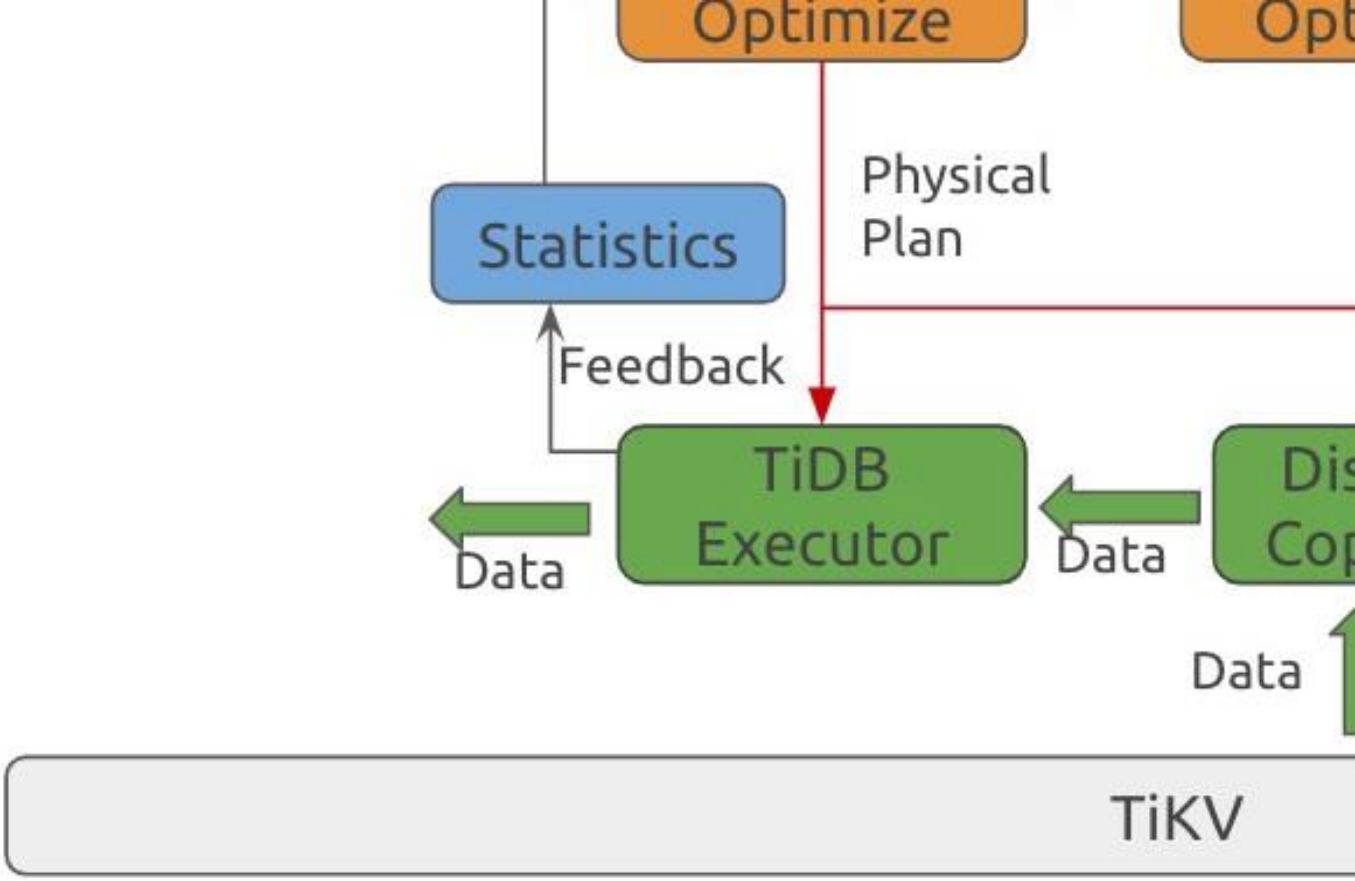


图 141: SQL Optimization

在经过了 parser 对原始查询文本的解析以及一些简单的合法性验证后，TiDB 首先会对查询做一些逻辑上的等价变化，详细的变化可以查询逻辑优化章节。

通过这些等价变化，使得这个查询在逻辑执行计划上可以变得更易于处理。在等价变化结束之后，TiDB 会得到一个与原始查询等价的查询计划结构，之后根据数据分布、以及一个算子具体的执行开销，来获得一个最终的执行计划，这部分内容可以查询物理优化章节。

同时，TiDB 在执行 PREPARE 语句时，可以选择开启缓存来降低TiDB 生成执行计划的开销，这部分内容会在执行计划缓存一节中介绍。

8.3.3.2 逻辑优化

8.3.3.2.1 逻辑优化

本章节将对一些比较关键的逻辑改写进行说明，帮助大家理解TiDB 如何生成最终的查询计划。比如在TiDB 输入 `select * from t where t.a in (select t1.a from t1 where t1.b=t.b)` 这个查询时，在最终的执行计划中将看不到这个 `t.a in (select t1.a from t1 where t1.b=t.b)` 这个 IN 子查询的存在，这便是因为TiDB 对这里进行了一些改写。

本章节会介绍如下几个关键改写：

- 子查询相关的优化
- 列裁剪

- 关联子查询去关联
- Max/Min 消除
- 谓词下推
- 分区裁剪
- TopN 和 Limit 下推
- Join Reorder

8.3.3.2.2 子查询相关的优化

本文主要介绍子查询相关的优化。

通常会遇到如下情况的子查询：

- `NOT IN (SELECT ... FROM ...)`
- `NOT EXISTS (SELECT ... FROM ...)`
- `IN (SELECT ... FROM ..)`
- `EXISTS (SELECT ... FROM ...)`
- `... >/>=/</<=/=/!= (SELECT ... FROM ...)`

有时，子查询中包含了非子查询中的列，如 `select * from t where t.a in (select * from t2 where t.b =t2.b)` 中，子查询中的 `t.b` 不是子查询中的列，而是从子查询外面引入的列。这种子查询通常会被称为关联子查询，外部引入的列会被称为关联列，关联子查询相关的优化参见关联子查询去关联。本文主要关注不涉及关联列的子查询。

子查询默认会以理解 TiDB 执行计划中提到的 `semi join` 作为默认的执行方式，同时对于一些特殊的子查询，TiDB 会做一些逻辑上的替换使得查询可以获得更好的执行性能。

`... < ALL (SELECT ... FROM ...)` 或者 `... > ANY (SELECT ... FROM ...)`

对于这种情况，可以将 `ALL` 或者 `ANY` 用 `MAX` 以及 `MIN` 来代替。不过由于在表为空时，`MAX(EXPR)` 以及 `MIN(EXPR)` 的结果会为 `NULL`，其表现形式和 `EXPR` 是有 `NULL` 值的结果一样。以及外部表达式结果为 `NULL` 时也会影响表达式的最终结果，因此这里完整的改写会是如下的形式：

- `t.id < all(select s.id from s)` 会被改写为 `t.id < min(s.id)and if(sum(s.id is null)!= 0, null, true)`。
- `t.id < any (select s.id from s)` 会被改写为 `t.id < max(s.id)or if(sum(s.id is null)!= 0, null, false)`。

`... != ANY (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数只有一种的话，那只要和这个值对比就即可。如果子查询中不同值的个数多于 1 个，那么必然会有不相等的情况出现。因此这样的子查询可以采取如下的改写手段：

- `select * from t where t.id != any (select s.id from s)` 会被改写为 `select t.* from t, (select s.id, count(distinct s.id)as cnt_distinct from s)where (t.id != s.id or cnt_distinct > 1)`

`... = ALL (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数多于一种的话，那么这个表达式的结果必然为假。因此这样的子查询在 TiDB 中会改写为如下的形式：

- select * from t where t.id = all (select s.id from s) 会被改写为 select t.* from t, (select s
  ′→ .id, count(distinct s.id)as cnt_distinct from s)where (t.id = s.id and cnt_distinct <= 1)

... IN (SELECT ... FROM ...)

对于这种情况，会将其 IN 的子查询改写为 SELECT ... FROM ... GROUP ... 的形式，然后将 IN 改写为普通的 JOIN 的形式。如 select * from t1 where t1.a in (select t2.a from t2) 会被改写为 select t1.* from t1
′→ , (select distinct(a)a from t2)t2 where t1.a = t2.a 的形式。同时这里的 DISTINCT 可以在 t2.a 具有 UNIQUE 属性时被自动消去。

```
explain select * from t1 where t1.a in (select t2.a from t2);
```

```
+--
    ′→                              +         +           +                       +
    ′→
| id                           | estRows | task       | access object           | operator info
    ′→                                                                  |
+--
    ′→                              +         +           +                       +
    ′→ ------------------------------ --------- ----------- -------------------------
| IndexJoin_12                 | 9990.00 | root       |                         | inner join, inner
    ′→ :TableReader_11, outer key:test.t2.a, inner key:test.t1.a |
| ├─HashAgg_21(Build)          | 7992.00 | root       |                         | group by:test.t2
    ′→ .a, funcs:firstrow(test.t2.a)->test.t2.a                    |
| │ └─IndexReader_28           | 9990.00 | root       |                         | index:
    ′→ IndexFullScan_27                                                 |
| │   └─IndexFullScan_27       | 9990.00 | cop[tikv] | table:t2, index:idx(a) | keep order:false
    ′→ , stats:pseudo                                               |
| └─TableReader_11(Probe)      | 7992.00 | root       |                         | data:
    ′→ TableRangeScan_10                                                |
|   └─TableRangeScan_10        | 7992.00 | cop[tikv] | table:t1                | range: decided
    ′→ by [test.t2.a], keep order:false, stats:pseudo               |
+--
    ′→                              +         +           +                       +
    ′→
```

这个改写会在 IN 子查询相对较小，而外部查询相对较大时产生更好的执行性能。因为不经过改写的情况下，我们无法使用以t2 为驱动表的 index join。同时这里的弊端便是，当改写删成的聚合无法被自动消去且 t2 表比较大时，反而会影响查询的性能。目前TiDB 中使用tidb_opt_insubq_to_join_and_agg 变量来控制这个优化的打开与否。当遇到不合适这个优化的情况可以手动关闭。

EXISTS 子查询以及 ... >/>=/</<=/=/!= (SELECT ... FROM ...)

当前对于这种场景的子查询，当它不是关联子查询时，TiDB 会在优化阶段提前展开它，将其直接替换为一个结果集直接判断结果。如下图中，EXISTS 会提前在优化阶段被执行为 TRUE，从而不会在最终的执行结果中看到它。

```
create table t1(a int);
```

```
create table t2(a int);
insert into t2 values(1);
explain select * from t1 where exists (select * from t2);
```

```
+--
    ′→ ---------------------+----------+-----------+--------------+--------------------------------- +
    ′→
| id                    | estRows  | task      | access object | operator info
    ′→ |
+--
    ′→ ---------------------+----------+-----------+--------------+--------------------------------- +
    ′→
| TableReader_12          | 10000.00 | root      |               | data:TableFullScan_11
    ′→ |
| └─TableFullScan_11      | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:pseudo
    ′→  |
+--
    ′→ ---------------------+----------+-----------+--------------+--------------------------------- +
    ′→
```

在上述优化中，优化器会自动优化语句执行。除以上情况外，你也可以在语句中添加SEMI_JOIN_REWRITE hint进一步改写语句。

如果不使用 SEMI_JOIN_REWRITE 进行改写，Semi Join 在选择 Hash Join 的执行方式时，只能够使用子查询构建哈希表，因此在子查询比外查询结果集大时，执行速度可能会不及预期。Semi Join 在选择Index Join 的执行方式时，只能够使用外查询作为驱动表，因此在子查询比外查询结果集小时，执行速度可能会不及预期。

使用 SEMI_JOIN_REWRITE 改写后，优化器便可以扩大选择范围，选择更好的执行方式。

8.3.3.2.3 列裁剪

列裁剪的基本思想在于：对于算子中实际用不上的列，优化器在优化的过程中没有必要保留它们。对这些列的删除会减少I/O 资源占用，并为后续的优化带来便利。下面给出一个列重复的例子：

假设表t 里面有a b c d 四列，执行如下语句：

```
select a from t where b > 5
```

在该查询的过程中，t 表实际上只有a, b 两列会被用到，而c, d 的数据则显得多余。对应到该语句的查询计划，Selection 算子会用到b 列，下面接着的 DataSource 算子会用到a, b 两列，而剩下c, d 两列则都可以裁剪掉，DataSource 算子在读数据时不需要将它们读进来。

出于上述考量，TiDB 会在逻辑优化阶段进行自上而下的扫描，裁剪不需要的列，减少资源浪费。该扫描过程称作“列裁剪”，对应逻辑优化规则中的 columnPruner。如果要关闭这个规则，可以在参照优化规则及表达式下推的黑名单中的关闭方法。

8.3.3.2.4 关联子查询去关联

子查询相关的优化中介绍了当没有关联列时，TiDB 是如何处理子查询的。由于为关联子查询解除关联依赖比较复杂，本文档中会介绍一些简单的场景以及这个优化规则的适用范围。

简介

以 `select * from t1 where t1.a < (select sum(t2.a)from t2 where t2.b = t1.b)` 为例，这里子查询 `t1.a < (select sum(t2.a)from t2 where t2.b = t1.b)` 中涉及了关联列上的条件 `t2.b=t1.b`，不过恰好由于这是一个等值条件，因此可以将其等价的改写为 `select t1.* from t1, (select b, sum(a)sum_a from t2 group by b)t2 where t1.b = t2.b and t1.a < t2.sum_a;`。这样，一个关联子查询就被重新改写为 JOIN 的形式。

TiDB 之所以要进行这样的改写，是因为关联子查询每次子查询执行时都是要和它的外部查询结果绑定的。在上面的例子中，如果 `t1.a` 有一千万个值，那这个子查询就要被重复执行一千万次，因为 `t2.b=t1.b` 这个条件会随着 `t1.a` 值的不同而发生变化。当通过一些手段将关联依赖解除后，这个子查询就只需要被执行一次了。

限制

这种改写的弊端在于，在关联没有被解除时，优化器是可以使用关联列上的索引的。也就是说，虽然这个子查询可能被重复执行多次，但是每次都可以使用索引过滤数据。而解除关联的变换上，通常是会导致关联列的位置发生改变而导致虽然子查询只被执行了一次，但是单次执行的时间会比没有解除关联时的单次执行时间长。

因此，在外部的值比较少的情况下，不解除关联依赖反而可能对执行性能更有帮助。这时可以通过使用 Optimizer Hint `NO_DECORRELATE` 或优化规则及表达式下推的黑名单中关闭 “子查询去关联” 优化规则的方式来关闭这个优化。在一般情况下，推荐使用 Optimizer Hint 并在需要时配合执行计划管理功能来禁止解除关联。

样例

```
create table t1(a int, b int);
create table t2(a int, b int, index idx(b));
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    '→                                  +          +          +               +
    '→
| id                                | estRows  | task      | access object | operator info
    '→                                                                          |
+--
    '→                                  +          +          +               +
    '→
| HashJoin_11                       | 9990.00  | root      |               | inner join, equal:[eq
    '→ (test.t1.b, test.t2.b)], other cond:lt(cast(test.t1.a), Column#7)  |
| ├─HashAgg_23(Build)               | 7992.00  | root      |               | group by:test.t2.b,
    '→ funcs:sum(Column#8)->Column#7, funcs:firstrow(test.t2.b)->test.t2.b |
| │ └─TableReader_24                | 7992.00  | root      |               | data:HashAgg_16
    '→                                                                          |
| │   └─HashAgg_16                  | 7992.00  | cop[tikv] |               | group by:test.t2.b,
    '→ funcs:sum(test.t2.a)->Column#8                                        |
| │     └─Selection_22              | 9990.00  | cop[tikv] |               | not(isnull(test.t2.b
    '→ ))                                                                    |
```

```
| |       └─TableFullScan_21       | 10000.00 | cop[tikv] | table:t2       | keep order:false,
    '→ stats:pseudo                                                          |
| └─TableReader_15(Probe)          | 9990.00  | root      |                | data:Selection_14
    '→                                                                       |
|   └─Selection_14                 | 9990.00  | cop[tikv] |                | not(isnull(test.t1.b
    '→ ))                                                                    |
|     └─TableFullScan_13           | 10000.00 | cop[tikv] | table:t1       | keep order:false,
    '→ stats:pseudo                                                          |
+--
    '→                                  +          +           +               +----------------------------------------------------
    '→
```

上面是优化生效的情况，可以看到 HashJoin_11 是一个普通的 inner join。

接下来，通过 Optimizer Hint NO_DECORRELATE 提示优化器不对该子查询解除关联：

```
explain select * from t1 where t1.a < (select /*+ NO_DECORRELATE() */ sum(t2.a) from t2 where t2.
    '→ b = t1.b);
```

```
+--
    '→ ---------------------------------------+----------+----------+-------------------------+----------
    '→
| id                                   | estRows    | task       | access object           |
    '→ operator info                                                                      |
+--
    '→ ---------------------------------------+------------+------------+-------------------------+----------
    '→
| Projection_10                        | 10000.00   | root       |                         |
    '→ test.t1.a, test.t1.b                                                               |
| └─Apply_12                           | 10000.00   | root       |                         |
     '→ CARTESIAN inner join, other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
|   ├─TableReader_14(Build)            | 10000.00   | root       |                         |
    '→ data:TableFullScan_13                                                              |
|   │ └─TableFullScan_13               | 10000.00   | cop[tikv]  | table:t1                |
    '→ keep order:false, stats:pseudo                                                     |
|   └─MaxOneRow_15(Probe)              | 10000.00   | root       |                         |
    '→                                                                                    |
|     └─StreamAgg_20                   | 10000.00   | root       |                         |
    '→ funcs:sum(Column#14)->Column#7                                                     |
|       └─Projection_45                | 100000.00  | root       |                         |
    '→ cast(test.t2.a, decimal(10,0) BINARY)->Column#14                                   |
|         └─IndexLookUp_44             | 100000.00  | root       |                         |
    '→                                                                                    |
|           ├─IndexRangeScan_42(Build) | 100000.00  | cop[tikv]  | table:t2, index:idx(b)  |
    '→ range: decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo       |
|           └─TableRowIDScan_43(Probe) | 100000.00  | cop[tikv]  | table:t2                |
    '→ keep order:false, stats:pseudo                                                     |
```

```
+--
    '→ ----------------------------------------+-----------+-----------+-------------------------+-----------------------------
    '→
```

也可以通过全局关闭关联规则达到同样的效果：

```
insert into mysql.opt_rule_blacklist values("decorrelate");
admin reload opt_rule_blacklist;
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+                                       +           +           +                         +-----------------------------
    '→
| id                                    | estRows   | task      | access object           |
    '→ operation info                                                                     |
+---------------------------------------+-----------+-----------+-------------------------+-----------------------------
    '→
| Projection_10                         | 10000.00  | root      |                         |
    '→ test.t1.a, test.t1.b                                                               |
|  └─Apply_12                           | 10000.00  | root      |                         |
    '→ CARTESIAN inner join, other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
|    ├─TableReader_14(Build)            | 10000.00  | root      |                         |
    '→ data:TableFullScan_13                                                              |
|    │ └─TableFullScan_13               | 10000.00  | cop[tikv] | table:t1                |
    '→ keep order:false, stats:pseudo                                                     |
|    └─MaxOneRow_15(Probe)              | 10000.00  | root      |                         |
    '→                                                                                    |
|      └─StreamAgg_20                   | 10000.00  | root      |                         |
    '→ funcs:sum(Column#14)->Column#7                                                     |
|        └─Projection_45                | 100000.00 | root      |                         |
    '→ cast(test.t2.a, decimal(10,0) BINARY)->Column#14                                   |
|          └─IndexLookUp_44             | 100000.00 | root      |                         |
    '→                                                                                    |
|            ├─IndexRangeScan_42(Build) | 100000.00 | cop[tikv] | table:t2, index:idx(b)  |
    '→ range: decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo       |
|            └─TableRowIDScan_43(Probe) | 100000.00 | cop[tikv] | table:t2                |
    '→ keep order:false, stats:pseudo                                                     |
+                                       +           +           +                         +-----------------------------
    '→
```

在执行了关闭关联规则的语句后，可以在`IndexRangeScan_25(Build)` 的`operator info` 中看到`range: decided ' → by [eq(test.t2.b, test.t1.b)]`。这部分信息就是关联依赖未被解除时，TiDB 使用关联条件进行索引范围查询的显示结果。

8.3.3.2.5 Max/Min 函数消除规则

在 SQL 中包含了 max/min 函数时，查询优化器会尝试使用 max/min 消除优化规则来将 max/min 聚合函数转换为 TopN 算子，从而能够有效地利用索引进行查询。

根据 select 语句中 max/min 函数的个数，这一优化规则有以下两种表现形式：

- 只有一个max/min 函数时的优化规则
- 存在多个max/min 函数时的优化规则

只有一个max/min 函数时的优化规则

当一个 SQL 满足以下条件时，就会应用这个规则：

- 只有一个聚合函数，且为 max 或者 min 函数。
- 聚合函数没有相应的 group by 语句。

例如：

```
select max(a) from t
```

这时 max/min 消除优化规则会将其重写为：

```
select max(a) from (select a from t where a is not null order by a desc limit 1) t
```

这个新的 SQL 语句在a 列存在索引（或 a 列是某个联合索引的前缀）时，能够利用索引只扫描一行数据来得到最大或者最小值，从而避免对整个表的扫描。

上述例子最终得到的执行计划如下：

```
mysql> explain select max(a) from t;
+                              +         +           +                          +----------------------------------------------------------
    '→
| id                           | estRows | task      | access object            | operator info
    '→                        |
+------------------------------+---------+-----------+--------------------------+                          ----------
    '→
| StreamAgg_13                 | 1.00    | root      |                          | funcs:max(test.t
    '→ .a)->Column#4       |
| └─Limit_17                   | 1.00    | root      |                          | offset:0, count
    '→ :1                    |
|   └─IndexReader_27           | 1.00    | root      |                          | index:Limit_26
    '→                       |
|     └─Limit_26               | 1.00    | cop[tikv] |                          | offset:0, count
    '→ :1                    |
|       └─IndexFullScan_25     | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep order:true
    '→ , desc, stats:pseudo |
+                              +         +           +                          +----------------------------------------------------------
    '→
5 rows in set (0.00 sec)
```

存在多个max/min 函数时的优化规则

当一个 SQL 满足以下条件时，就会应用这个规则：

- 有多个聚合函数，且所有的聚合函数都是max/min
- 聚合函数没有相应的 `group by` 语句。
- 每个 `max/min` 聚合函数参数中的列都有索引能够保序。

下面是一个简单的例子：

```
select max(a) - min(a) from t
```

优化规则会先检查a 列是否存在索引能够为其保序，如果存在，这个 SQL 会先被重写为两个子查询的笛卡尔积：

```
select max_a - min_a
from
    (select max(a) as max_a from t) t1,
    (select min(a) as min_a from t) t2
```

这样，两个子句中的 `max/min` 函数就可以使用上述“只有一个 `max/min` 函数时的优化规则”分别进行优化，最终重写为：

```
select max_a - min_a
from
    (select max(a) as max_a from (select a from t where a is not null order by a desc limit 1) t)
        '→  t1,
    (select min(a) as min_a from (select a from t where a is not null order by a asc limit 1) t)
        '→ t2
```

同样的，如果a 列能够使用索引保序，那这个优化只会扫描两行数据，避免了对整个表的扫描。但如果a 列没有可以保序的索引，这个变换会使原本只需一次的全表扫描变成两次，因此这个规则就不会被应用。

最后得到的执行计划：

```
mysql> explain select max(a)-min(a) from t;
+                                   +         +           +                          +----------------------------------------------
    '→
| id                                | estRows | task      | access object            | operator
    '→ info                       |
+-----------------------------------+---------+-----------+--------------------------+                -----------------------
    '→
| Projection_17                     | 1.00    | root      |                          | minus(
    '→ Column#4, Column#5)->Column#6 |
| └─HashJoin_18                      | 1.00    | root      |                           | CARTESIAN
    '→  inner join                |
|   ├─StreamAgg_45(Build)           | 1.00    | root      |                          | funcs:min
    '→ (test.t.a)->Column#5       |
```

```
|   | └─Limit_49                    | 1.00    | root      |                                | offset:0,
    '→ count:1                   |
|   |   └─IndexReader_59            | 1.00    | root      |                                | index:
    '→ Limit_58                  |
|   |     └─Limit_58                | 1.00    | cop[tikv] |                                | offset:0,
    '→ count:1                   |
|   |       └─IndexFullScan_57      | 1.00    | cop[tikv] | table:t, index:idx_a(a)        | keep
    '→ order:true, stats:pseudo  |
|   └─StreamAgg_24(Probe)           | 1.00    | root      |                                | funcs:max
    '→ (test.t.a)->Column#4      |
|     └─Limit_28                    | 1.00    | root      |                                | offset:0,
    '→ count:1                   |
|       └─IndexReader_38            | 1.00    | root      |                                | index:
    '→ Limit_37                  |
|         └─Limit_37                | 1.00    | cop[tikv] |                                | offset:0,
    '→ count:1                   |
|           └─IndexFullScan_36      | 1.00    | cop[tikv] | table:t, index:idx_a(a)        | keep
    '→ order:true, desc, stats:pseudo |
+-------------------------------------+---------+-----------+--------------------------------+---------------------------------------------------
    '→
12 rows in set (0.01 sec)
```

8.3.3.2.6 谓词下推

本文档介绍TiDB 逻辑优化规则中的谓词下推规则，旨在让读者对谓词下推形成理解，并了解常见的谓词下推适用及不适用的场景。

谓词下推将查询语句中的过滤表达式计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

示例

以下通过一些例子对谓词下推优化进行说明，其中示例 1、2、3 为谓词下推适用的案例，示例 4、5、6 为谓词下推不适用的案例。

示例 1: 谓词下推到存储层

```
create table t(id int primary key, a int);
explain select * from t where a < 1;
+--
    '→ ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    '→
| id                    | estRows  | task      | access object | operator info
    '→                  |
+--
    '→ ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    '→
```

```
| TableReader_7            | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6            | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，将谓词a < 1 下推到TiKV 上对数据进行过滤，可以减少由于网络传输带来的开销。

示例 2: 谓词下推到存储层

```
create table t(id int primary key, a int not null);
explain select * from t where a < substring('123', 1, 1);
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------------------------------------+
    ↪
| id                       | estRows  | task      | access object | operator info
    ↪                 |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------------------------------------+
    ↪
| TableReader_7            | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6            | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------------------------------------+
    ↪
```

该查询与示例 1 中的查询生成了完成一样的执行计划， 这是因为谓词 a < substring('123', 1, 1) 的 substring 的入参均为常量，因此可以提前计算，进而简化得到等价的谓词a < 1。进一步的，可以将a < 1 下推至TiKV 上。

示例 3: 谓词下推到join 下方

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a where t.a < 1;
+--
    ↪                              +          +          +               +
    ↪
```

```
| id                          | estRows  | task      | access object | operator info
    '→                              |
+--
    '→ ----------------------------+----------+-----------+---------------+
    '→
| HashJoin_8                  | 4154.17  | root      |               | inner join, equal:[eq(
    '→ test.t.a, test.s.a)] |
| ├─TableReader_15(Build)     | 3323.33  | root      |               | data:Selection_14
    '→                           |
| │ └─Selection_14            | 3323.33  | cop[tikv] |               | lt(test.s.a, 1)
    '→                             |
| │   └─TableFullScan_13      | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    '→ pseudo             |
| └─TableReader_12(Probe)     | 3323.33  | root      |               | data:Selection_11
    '→                           |
|   └─Selection_11            | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    '→                             |
|     └─TableFullScan_10      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo             |
+--
    '→                              +          +           +               +
    '→
7 rows in set (0.00 sec)
```

在该查询中，将谓词 `t.a < 1` 下推到join 前进行过滤，可以减少join 时的计算开销。

此外，这条 SQL 执行的是内连接，且 ON 条件是 `t.a = s.a`，可以由 `t.a < 1` 推导出谓词 `s.a < 1`，并将其下推至 join 运算前对 s 表进行过滤，可以进一步减少join 时的计算开销。

示例 4: 存储层不支持的谓词无法下推

```
create table t(id int primary key, a varchar(10) not null);
desc select * from t where truncate(a, " ") = '1';
+--
    '→                          +          +           +               +
    '→
| id                        | estRows  | task      | access object | operator info
    '→                                   |
+--
    '→                          +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
| Selection_5               | 8000.00  | root      |               | eq(truncate(cast(test.t.a,
    '→ double BINARY), 0), 1) |
| └─TableReader_7           | 10000.00 | root      |               | data:TableFullScan_6
    '→                                |
|   └─TableFullScan_6       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo                      |
```

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

在该查询中，存在谓词 truncate(a, " ")= '1'。

从 explain 结果中可以看到，该谓词没有被下推到 TiKV 上进行计算，这是因为 TiKV coprocessor 中没有对 truncate 内置函数进行支持，因此无法将其下推到TiKV 上。

示例 5: 外连接中内表上的谓词不能下推

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a where s.a is null;
+--
    ↪                            +          +           +               +
    ↪
| id                            | estRows   | task      | access object | operator info
    ↪                                    |
+--
    ↪ -----------------------------+----------+-----------+---------------+
    ↪
| Selection_7                   | 10000.00 | root      |               | isnull(test.s.a)
    ↪                                |
| └─HashJoin_8                   | 12500.00 | root      |               | left outer join, equal
    ↪ :[eq(test.t.a, test.s.a)] |
|   ├─TableReader_13(Build)      | 10000.00 | root      |               | data:TableFullScan_12
    ↪                            |
|   │ └─TableFullScan_12          | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo                  |
|   └─TableReader_11(Probe)      | 10000.00 | root      |               | data:TableFullScan_10
    ↪                            |
|     └─TableFullScan_10          | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo                  |
+--
    ↪                            +          +           +               +
    ↪
6 rows in set (0.00 sec)
```

在该查询中，内表s 上存在谓词 s.a is null。

从 explain 中可以看到，该谓词没有被下推到join 前进行计算，这是因为外连接在不满足 on 条件时会对内表填充 NULL，而在该查询中 s.a is null 用来对join 后的结果进行过滤，如果将其下推到join 前在内表上进行过滤，则下推前后不等价，因此不可进行下推。

示例 6: 谓词中包含用户变量时不能下推

```
create table t(id int primary key, a char);
set @a = 1;
```

```
explain select * from t where a < @a;
+--
    '→ ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    '→
| id                        | estRows  | task      | access object | operator info
    '→                   |
+--
    '→ ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    '→
| Selection_5               | 8000.00  | root      |               | lt(test.t.a, getvar("a"))
    '→      |
| └─TableReader_7           | 10000.00 | root      |               | data:TableFullScan_6
    '→           |
|   └─TableFullScan_6       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo |
+--
    '→ ----------------------+----------+-----------+---------------+-------------------------------------------------- +
    '→
3 rows in set (0.00 sec)
```

在该查询中，表t上存在谓词a < @a，其中 @a 为值为 1 的用户变量。

从 explain 中可以看到，该谓词没有像示例 2 中一样，将谓词简化为a < 1 并下推到TiKV 上进行计算。这是因为，用户变量 @a 的值可能会某些场景下在查询过程中发生改变，且TiKV 对于用户变量 @a 的值不可知，因此 TiDB 不会将 @a 替换为 1，且不会下推至TiKV 上进行计算。

一个帮助理解的例子如下：

```
create table t(id int primary key, a int);
insert into t values(1, 1), (2,2);
set @a = 1;
select id, a, @a:=@a+1 from t where a = @a;
+----+------+-----------------+
| id | a    | @a:=@a+1 |
+----+------+-----------------+
|  1 |    1 | 2               |
|  2 |    2 | 3               |
+----+------+-----------------+
2 rows in set (0.00 sec)
```

可以从在该查询中看到，@a 的值会在查询过程中发生改变，因此如果将a = @a 替换为a = 1 并下推至TiKV，则优化前后不等价。

8.3.3.2.7 分区裁剪

分区裁剪是只有当目标表为分区表时，才可以进行的一种优化方式。分区裁剪通过分析查询语句中的过滤条件，只选择可能满足条件的分区，不扫描匹配不上的分区，进而显著地减少计算的数据量。

例如：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY,
 pad VARCHAR(100)
)
PARTITION BY RANGE COLUMNS(id)
 ( PARTITION p0 VALUES LESS THAN (100),
 PARTITION p1 VALUES LESS THAN (200),
 PARTITION p2 VALUES LESS THAN (MAXVALUE)
);
INSERT INTO t1 VALUES (1, 'test1'),(101, 'test2'), (201, 'test3');
EXPLAIN SELECT * FROM t1 WHERE id BETWEEN 80 AND 120;
```

```
+--
    '→                              +         +           +                         +----------------------------------------------------------------------------
    '→
| id                           | estRows | task      | access object           | operator info
    '→                                  |
+--
    '→ ----------------------------+---------+-----------+-------------------------+                                                                            ----------
    '→
| PartitionUnion_8             | 80.00   | root      |                         |
    '→                                                |
| ├─TableReader_10              | 40.00   | root      |                          | data:
    '→ TableRangeScan_9                              |
| │ └─TableRangeScan_9          | 40.00   | cop[tikv] | table:t1, partition:p0 | range:[80,120],
    '→ keep order:false, stats:pseudo |
| └─TableReader_12              | 40.00   | root      |                          | data:
    '→ TableRangeScan_11                             |
|   └─TableRangeScan_11         | 40.00   | cop[tikv] | table:t1, partition:p1 | range:[80,120],
    '→ keep order:false, stats:pseudo |
+--
    '→                              +         +           +                         +----------------------------------------------------------------------------

    '→
5 rows in set (0.00 sec)
```

分区裁剪的使用场景

分区表有Range分区和hash分区两种形式，分区裁剪对两种分区表也有不同的使用场景。

分区裁剪在 Hash 分区表上的应用

Hash 分区表上可以使用分区裁剪的场景

只有等值比较的查询条件能够支持 Hash 分区表的裁剪。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t where x = 1;
```

```
+--
    '-> ----------------------+----------+-----------+-----------------------+                                +
    '->
| id                        | estRows  | task      | access object         | operator info
    '->                  |
+--
    '-> ------------------------ ---------- ----------- ------------------------                                +
    '->
| TableReader_8             | 10.00    | root      |                       | data:Selection_7
    '->               |
| └─Selection_7             | 10.00    | cop[tikv] |                       | eq(test.t.x, 1)
    '->                |
|   └─TableFullScan_6       | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    '-> stats:pseudo |
+--    ----------------------- ---------- ----------- ------------------------                                +
    '->                          +          +           +                       +
    '->
```

在这条 SQL 中，由条件 `x = 1` 可以知道所有结果均在一个分区上。数值 1 在经过 Hash 后，可以确定其在分区 p1 中。因此只需要扫描分区 p1，而无需访问一定不会出现相关结果的 p2 、p3 、p4 分区。从执行计划来看，其中只出现了一个 TableFullScan 算子，且在 access object 中指定了 p1 分区，确认 partition pruning 生效了。

Hash 分区表上不能使用分区裁剪的场景

场景一

不能确定查询结果只在一个分区上的条件：如 in, between, > < >= <= 等查询条件，不能使用分区裁剪的优化。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t where x > 2;
```

```
+--
    '->                            +          +           +                       +----------------------------
    '->
| id                            | estRows  | task      | access object         | operator info
    '->                  |
+--
    '->                            +          +           +                       +----------------
    '->
| Union_10                      | 13333.33 | root      |                       |
    '->                                |
| ├─TableReader_13              | 3333.33  | root      |                       | data:
    '-> Selection_12                |
| │ └─Selection_12              | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    '->                |
```

```
| │   └─TableFullScan_11        | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false
    ′→ , stats:pseudo |
| ├─TableReader_16              | 3333.33  | root      |                       | data:
    ′→ Selection_15              |
| │ └─Selection_15              | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ′→                 |
| │   └─TableFullScan_14        | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false
    ′→ , stats:pseudo |
| ├─TableReader_19              | 3333.33  | root      |                       | data:
    ′→ Selection_18              |
| │ └─Selection_18              | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ′→                 |
| │   └─TableFullScan_17        | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false
    ′→ , stats:pseudo |
| └─TableReader_22              | 3333.33  | root      |                       | data:
    ′→ Selection_21              |
|   └─Selection_21              | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ′→                 |
|     └─TableFullScan_20        | 10000.00 | cop[tikv] | table:t, partition:p3 | keep order:false
    ′→ , stats:pseudo |
+--
    ′→                          +          +                                   +
    ′→
```

在这条 SQL 中，`x > 2` 条件无法确定对应的 Hash Partition，所以不能使用分区裁剪。

场景二

由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t2 where x = (select * from t1 where t2.x = t1.x and t2.x < 2);
```

```
+--
    ′→                                  +          +          +                    +
    ′→
| id                                | estRows  | task      | access object          | operator
    ′→  info                             |
+--
    ′→                                  +          +          +                    +
    ′→
| Projection_13                     | 9990.00  | root      |                        | test.t2.
    ′→ x                                 |
| └─Apply_15                         | 9990.00  | root      |                        | inner
    ′→ join, equal:[eq(test.t2.x, test.t1.x)] |
```

```
| ├─TableReader_18(Build)            | 9990.00  | root      |                          | data:
'→ Selection_17                              |
| | └─Selection_17                   | 9990.00  | cop[tikv] |                          | not(
'→ isnull(test.t2.x))                        |
| |   └─TableFullScan_16             | 10000.00 | cop[tikv] | table:t2                 | keep
'→ order:false, stats:pseudo                 |
| └─Selection_19(Probe)              | 0.80     | root      |                          | not(
'→ isnull(test.t1.x))                        |
|   └─MaxOneRow_20                   | 1.00     | root      |                          |
'→                                           |
|     └─Union_21                     | 2.00     | root      |                          |
'→                                           |
|       ├─TableReader_24             | 2.00     | root      |                          | data:
'→ Selection_23                              |
|       | └─Selection_23             | 2.00     | cop[tikv] |                          | eq(test
'→ .t2.x, test.t1.x), lt(test.t2.x, 2)       |
|       |   └─TableFullScan_22       | 2500.00  | cop[tikv] | table:t1, partition:p0   | keep
'→ order:false, stats:pseudo                 |
|       └─TableReader_27             | 2.00     | root      |                          | data:
'→ Selection_26                              |
|         └─Selection_26             | 2.00     | cop[tikv] |                          | eq(test
'→ .t2.x, test.t1.x), lt(test.t2.x, 2)       |
|           └─TableFullScan_25       | 2500.00  | cop[tikv] | table:t1, partition:p1   | keep
+--'→ order:false, stats:pseudo               |      +           +                          +
'→                                    +                                                    ----------
'→
```

这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x = val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

分区裁剪在Range 分区表上的应用

Range 分区表上可以使用分区裁剪的场景

场景一

等值比较的查询条件可以使用分区裁剪。

```
create table t (x int) partition by range (x)
    ( partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x = 3;
```

```
+--
'→ ----------------------+----------+-----------+----------------------+--------------------------------+
'→
```

```
| id                      | estRows  | task      | access object          | operator info
    ′→                  |
+--
    ′→ ----------------------+----------+-----------+------------------------+-------------------------------+
    ′→
| TableReader_8           | 10.00    | root      |                        | data:Selection_7
    ′→               |
| └─Selection_7           | 10.00    | cop[tikv] |                        | eq(test.t.x, 3)
    ′→                |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false,
    ′→ stats:pseudo |
+--  ----------------------  ----------  ----------  ----------------------                                 +
    ′→                         +          +           +                       +--------------------------------
    ′→
```

使用 in 条件的等值比较查询条件也可以使用分区裁剪。

```
create table t (x int) partition by range (x)
    ( partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x in(1,13);
```

```
+--
    ′→                          +          +           +                       +-------------------------------
    ′→
| id                          | estRows  | task      | access object          | operator info
    ′→                  |
+--
    ′→                          +          +           +                       +                              ----------
    ′→ --------------------------  ----------  ----------  ----------------------
| Union_8                     | 40.00    | root      |                        |
    ′→                             |
| ├─TableReader_11             | 20.00    | root      |                        | data:Selection_10
    ′→               |
| │ └─Selection_10             | 20.00    | cop[tikv] |                        | in(test.t.x, 1,
    ′→ 13)            |
| │   └─TableFullScan_9        | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false,
    ′→  stats:pseudo |
| └─TableReader_14             | 20.00    | root      |                        | data:Selection_13
    ′→               |
|   └─Selection_13             | 20.00    | cop[tikv] |                        | in(test.t.x, 1,
    ′→ 13)            |
|     └─TableFullScan_12       | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false,
    ′→  stats:pseudo |
```

```
+--
    '-> ------------------------+----------+----------+----------------------+-----------------------------
    '->
```

在这条 SQL 中，由条件`x in(1,13)`可以知道所有结果只会分布在几个分区上。经过分析，发现所有`x = 1`的记录都在分区 `p0` 上，所有`x = 13`的记录都在分区 `p2` 上，因此只需要访问 `p0`、`p2` 这两个分区，

场景二

区间比较的查询条件如 `between,> < = >= <=` 可以使用分区裁剪。

```
create table t (x int) partition by range (x)
    ( partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x between 7 and 14;
```

```
+--
    '->                          +          +          +                      +-----------------------------
    '->
| id                          | estRows   | task      | access object          | operator info
    '->                      |
+--
    '->                          +          +          +                      +
    '-> ---------------------------+----------+----------+----------------------+
    '->
| Union_8                     | 500.00    | root      |                        |
    '->                                  |
| ├─TableReader_11             | 250.00    | root      |                        | data:Selection_10
    '->                    |
| │ └─Selection_10             | 250.00    | cop[tikv] |                        | ge(test.t.x, 7),
    '-> le(test.t.x, 14) |
| │   └─TableFullScan_9        | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    '->  stats:pseudo     |
| └─TableReader_14             | 250.00    | root      |                        | data:Selection_13
    '->                    |
|   └─Selection_13             | 250.00    | cop[tikv] |                        | ge(test.t.x, 7),
    '-> le(test.t.x, 14) |
|     └─TableFullScan_12       | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false,
    '->  stats:pseudo     |
+--
    '->                          +          +          +                      +-----------------------------
    '->
```

场景三

分区表达式为 `fn(col)` 的简单形式，查询条件是`> < = >= <=` 之一，且 `fn` 是单调函数，可以使用分区裁剪。

关于 fn 函数，对于任意x y，如果x > y，则 fn(x)> fn(y)，那么这种是严格递增的单调函数。非严格递增的单调函数也可以符合分区裁剪要求，只要函数 fn 满足：对于任意x y，如果x > y，则 fn(x)>= fn(y)。理论上，所有满足单调条件（严格或者非严格）的函数都支持分区裁剪。目前，TiDB 支持的单调函数如下：

- UNIX_TIMESTAMP()
- TO_DAYS()

例如，分区表达式是 fn(col) 形式，fn 为我们支持的单调函数 to_days，就可以使用分区裁剪：

```
create table t (id datetime) partition by range (to_days(id))
    ( partition p0 values less than (to_days('2020-04-01')),
    partition p1 values less than (to_days('2020-05-01')));
explain select * from t where id > '2020-04-18';
```

```
+--
    '→                          +           +           +                          +
    '→
| id                        | estRows   | task       | access object          | operator info
    '→                                |
+--
    '→                          +           +           +                          +
    '→ ------------------------ ----------- ------------ ------------------------
| TableReader_8             | 3333.33   | root       |                        | data:Selection_7
    '→                              |
| └─Selection_7             | 3333.33   | cop[tikv] |                         | gt(test.t.id,
    '→ 2020-04-18 00:00:00.000000) |
|   └─TableFullScan_6       | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    '→ stats:pseudo              |
+--
    '→                          +           +           +                          +
    '→
```

Range 分区表上不能使用分区裁剪的场景

由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

```
create table t1 (x int) partition by range (x)
    ( partition p0 values less than (5),
    partition p1 values less than (10));
create table t2 (x int);
explain select * from t2 where x < (select * from t1 where t2.x < t1.x and t2.x < 2);
```

```
+--
    '→                                  +           +           +                          +
    '→
```

```
| id                                | estRows  | task      | access object          | operator
    '→  info                                                |
+--
    '→ ----------------------------------+----------+-----------+------------------------+
    '→
| Projection_13                     | 9990.00  | root      |                        | test.t2.
    '→ x                                                    |
| └─Apply_15                         | 9990.00  | root      |                         |
    '→ CARTESIAN inner join, other cond:lt(test.t2.x, test.t1.x) |
|   ├─TableReader_18(Build)          | 9990.00  | root      |                         | data:
    '→ Selection_17                                         |
|   │ └─Selection_17                 | 9990.00  | cop[tikv] |                         | not(
    '→ isnull(test.t2.x))                                    |
|   │   └─TableFullScan_16           | 10000.00 | cop[tikv] | table:t2                | keep
    '→ order:false, stats:pseudo                            |
|   └─Selection_19(Probe)            | 0.80     | root      |                         | not(
    '→ isnull(test.t1.x))                                    |
|     └─MaxOneRow_20                 | 1.00     | root      |                         |
    '→                                                        |
|       └─Union_21                   | 2.00     | root      |                         |
    '→                                                        |
|         ├─TableReader_24           | 2.00     | root      |                         | data:
    '→ Selection_23                                         |
|         │ └─Selection_23           | 2.00     | cop[tikv] |                         | lt(test
    '→ .t2.x, 2), lt(test.t2.x, test.t1.x)                  |
|         │   └─TableFullScan_22     | 2.50     | cop[tikv] | table:t1, partition:p0 | keep
    '→ order:false, stats:pseudo                            |
|         └─TableReader_27           | 2.00     | root      |                         | data:
    '→ Selection_26                                         |
|           └─Selection_26           | 2.00     | cop[tikv] |                         | lt(test
    '→ .t2.x, 2), lt(test.t2.x, test.t1.x)                  |
|             └─TableFullScan_25     | 2.50     | cop[tikv] | table:t1, partition:p1 | keep
    '→ order:false, stats:pseudo                            |
+--
    '→                                  +          +           +                        +
                                                                                          ----------
    '→
14 rows in set (0.00 sec)
```

这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x > val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

8.3.3.2.8 TopN 和 Limit 下推

SQL 中的 LIMIT 子句在 TiDB 查询计划树中对应 Limit 算子节点，ORDER BY 子句在查询计划树中对应 Sort 算子节点，此外，我们会将相邻的 Limit 和 Sort 算子组合成 TopN 算子节点，表示按某个排序规则提取记录的前 N 项。

从另一方面来说，Limit 节点等价于一个排序规则为空的TopN 节点。

和谓词下推类似，TopN（及 Limit，下同）下推将查询计划树中的 TopN 计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

如果要关闭这个规则，可参照优化规则及表达式下推的黑名单中的关闭方法。

示例

以下通过一些例子对TopN 下推进行说明。

示例 1：下推到存储层 Coprocessor

```
create table t(id int primary key, a int not null);
explain select * from t order by a limit 10;
```

```
+---------------------------+----------+-----------+---------------+--------------------------------------------------------+
    ′→
| id                        | estRows  | task      | access object | operator info
    ′→                  |
+---------------------------+----------+-----------+---------------+--------------------------------------------------------+
    ′→
| TopN_7                    | 10.00    | root      |               | test.t.a, offset:0, count
    ′→ :10    |
| └─TableReader_15           | 10.00    | root      |               | data:TopN_14
    ′→                    |
|   └─TopN_14                | 10.00    | cop[tikv] |               | test.t.a, offset:0, count
    ′→ :10    |
|     └─TableFullScan_13     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ′→ pseudo |
+---------------------------+----------+-----------+---------------+--------------------------------------------------------+
    ′→
4 rows in set (0.00 sec)
```

在该查询中，将TopN 算子节点下推到TiKV 上对数据进行过滤，每个 Coprocessor 只向TiDB 传输 10 条记录。在 TiDB 将数据整合后，再进行最终的过滤。

示例 2：TopN 下推过Join 的情况（排序规则仅依赖于外表中的列）

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.a limit 10;
```

```
+                                 +          +           +               +-------------------------------------------------------
    ′→
| id                              | estRows  | task      | access object | operator info
    ′→                                    |
+                                 +          +           +               +                                                       ----------
 ------′→------------------------------ ----------- ------------ ----------------
```

```
| TopN_12                          | 10.00     | root      |               | test.t.a, offset:0,
    ′→ count:10                    |
| └─HashJoin_17                    | 12.50     | root      |               | left outer join,
    ′→ equal:[eq(test.t.a, test.s.a)] |
|   ├─TopN_18(Build)               | 10.00     | root      |               | test.t.a, offset:0,
    ′→ count:10                    |
|   │ └─TableReader_26             | 10.00     | root      |               | data:TopN_25
    ′→                                   |
|   │   └─TopN_25                  | 10.00     | cop[tikv] |               | test.t.a, offset:0,
    ′→ count:10                    |
|   │     └─TableFullScan_24       | 10000.00 | cop[tikv] | table:t
    ′→ stats:pseudo                  |                                       | data:
    └─TableReader_30(Probe)        | 10000.00 | root      |
|   ′→ TableFullScan_29                          |                           | keep order:false,
      └─TableFullScan_29           | 10000.00 | cop[tikv] | table:s
+   ′→ stats:pseudo                  |                                     +
                                  +          +           +                 ----------------------------------------------------------------
    ′→
8 rows in set (0.01 sec)
```

在该查询中，TopN 算子的排序规则仅依赖于外表t 中的列，可以将TopN 下推到Join 之前进行一次计算，以减少 Join 时的计算开销。除此之外，TiDB 同样将TopN 下推到了存储层中。

示例 3：TopN 不能下推过Join 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a order by t.id limit 10;
```

```
+                                +           +            +                +-------------------------------------------------------------
    ′→
| id                             | estRows   | task       | access object | operator info
    ′→                               |
+--------------------------------+-----------+------------+---------------+                                                             ----------
    ′→
| TopN_12                        | 10.00     | root       |               | test.t.id, offset:0,
    ′→ count:10                |
| └─HashJoin_16                  | 12500.00 | root       |               | inner join, equal:[eq(
    ′→ test.t.a, test.s.a)] |
|   ├─TableReader_21(Build)      | 10000.00 | root       |               | data:TableFullScan_20
    ′→                       |
|   │ └─TableFullScan_20         | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ′→ :pseudo             |
|   └─TableReader_19(Probe)      | 10000.00 | root       |               | data:TableFullScan_18
    ′→                       |
|     └─TableFullScan_18         | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ′→ :pseudo             |
```

```
+                                +           +           +                +-----------------------------------------------------------------
    '→
6 rows in set (0.00 sec)
```

TopN 无法下推过Inner Join。以上面的查询为例，如果先Join 得到 100 条记录，再做TopN 可以剩余 10 条记录。而如果在TopN 之前就过滤到剩余 10 条记录，做完Join 之后可能就剩下 5 条了，导致了结果的差异。

同理，TopN 无法下推到Outer Join 的内表上。在TopN 的排序规则涉及多张表上的列时，也无法下推，如`t.a+s.a`。只有当TopN 的排序规则仅依赖于外表上的列时，才可以下推。

示例 4：TopN 转换成 Limit 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.id limit 10;
```

```
+                                  +           +           +                +-----------------------------------------------------------------
    '→
| id                               | estRows   | task      | access object | operator info
    '→                                    |
+                                  +           +           +                +-----------------------------------------------------------------
 ----'→---------------------------------- ----------- ------------ -----------------
| TopN_12                          | 10.00     | root      |               | test.t.id, offset:0,
    '→ count:10                   |
|  └─HashJoin_17                    | 12.50     | root      |               | left outer join,
    '→ equal:[eq(test.t.a, test.s.a)] |
|    ├─Limit_21(Build)              | 10.00     | root      |               | offset:0, count:10
    '→                             |
|    │ └─TableReader_31             | 10.00     | root      |               | data:Limit_30
    '→                                  |
|    │   └─Limit_30                 | 10.00     | cop[tikv] |               | offset:0, count:10
    '→                             |
|    │     └─TableFullScan_29       | 10.00     | cop[tikv] | table:t       | keep order:true,
    '→ stats:pseudo                   |
|    └─TableReader_35(Probe)        | 10000.00 | root      |               | data:
    '→ TableFullScan_34                          |
|      └─TableFullScan_34           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    '→ stats:pseudo                 |
+                                  +           +           +                +-----------------------------------------------------------------
    '→
8 rows in set (0.00 sec)
```

在上面的查询中，TopN 首先推到了外表t 上。然后因为它要对 `t.id` 进行排序，而 `t.id` 是表t 的主键，可以直接按顺序读出(`keep order:true`)，从而省略了 TopN 中的排序，将其简化为 Limit。

8.3.3.2.9 Join Reorder 算法简介

在实际的业务场景中，多个表的Join 语句是很常见的，而Join 的执行效率和各个表参与Join 的顺序有关系。如 `select * from t1, t2, t3 where t1.a=t2.a and t3.a=t2.a`，这个 SQL 中可能的执行顺序有“t1 和 t2 先做 Join，然后再和t3 做 Join”以及“t2 和 t3 先做Join，然后再和t1 做 Join”两种情况。根据 t1 和 t3 的数据量及数据分布，这两种执行顺序会有不同的性能表现。

因此优化器需要实现一种决定Join 顺序的算法。目前TiDB 中存在两种Join Reorder 算法，贪心算法和动态规划算法。

- Join Reorder 贪心算法：在所有参与Join 的节点中，选择行数最小的表与其他各表分别做一次Join 的结果估算，然后选择其中结果最小的一对进行Join，再继续这个过程进入下一轮的选择和Join，直到所有的节点都完成Join。
- Join Reorder 动态规划算法：在所有参与Join 的节点中，枚举所有可能的Join 顺序，然后选择最优的Join 顺序。

Join Reorder 贪心算法实例

以三个表t1、t2、t3 的Join 为例。首先获取所有参与Join 的节点，将所有节点按照行数多少，从少到多进行排序。

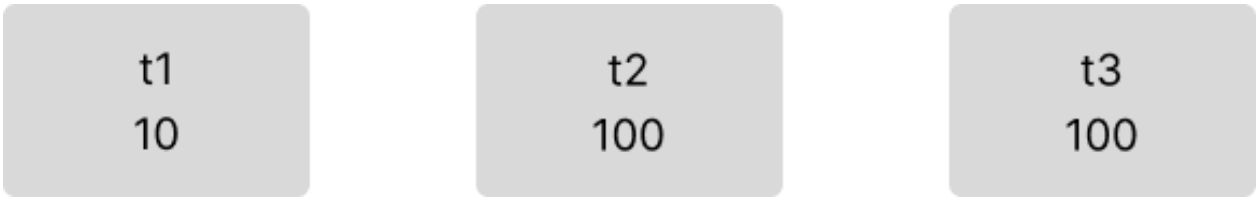


图 142: join-reorder-1

之后选定其中最小的表，将其与其他两个表分别做一次Join，观察输出的结果集大小，选择其中结果更小的一对。

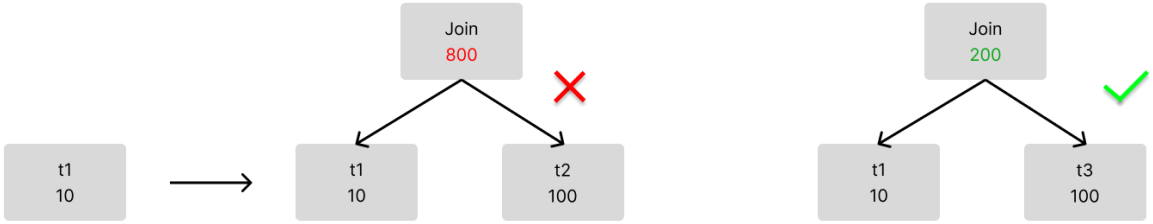


图 143: join-reorder-2

然后进入下一轮的选择，如果这时是四个表，那么就继续比较输出结果集的大小，进行选择。这里只有三个表，因此就直接得到了最终的Join 结果。

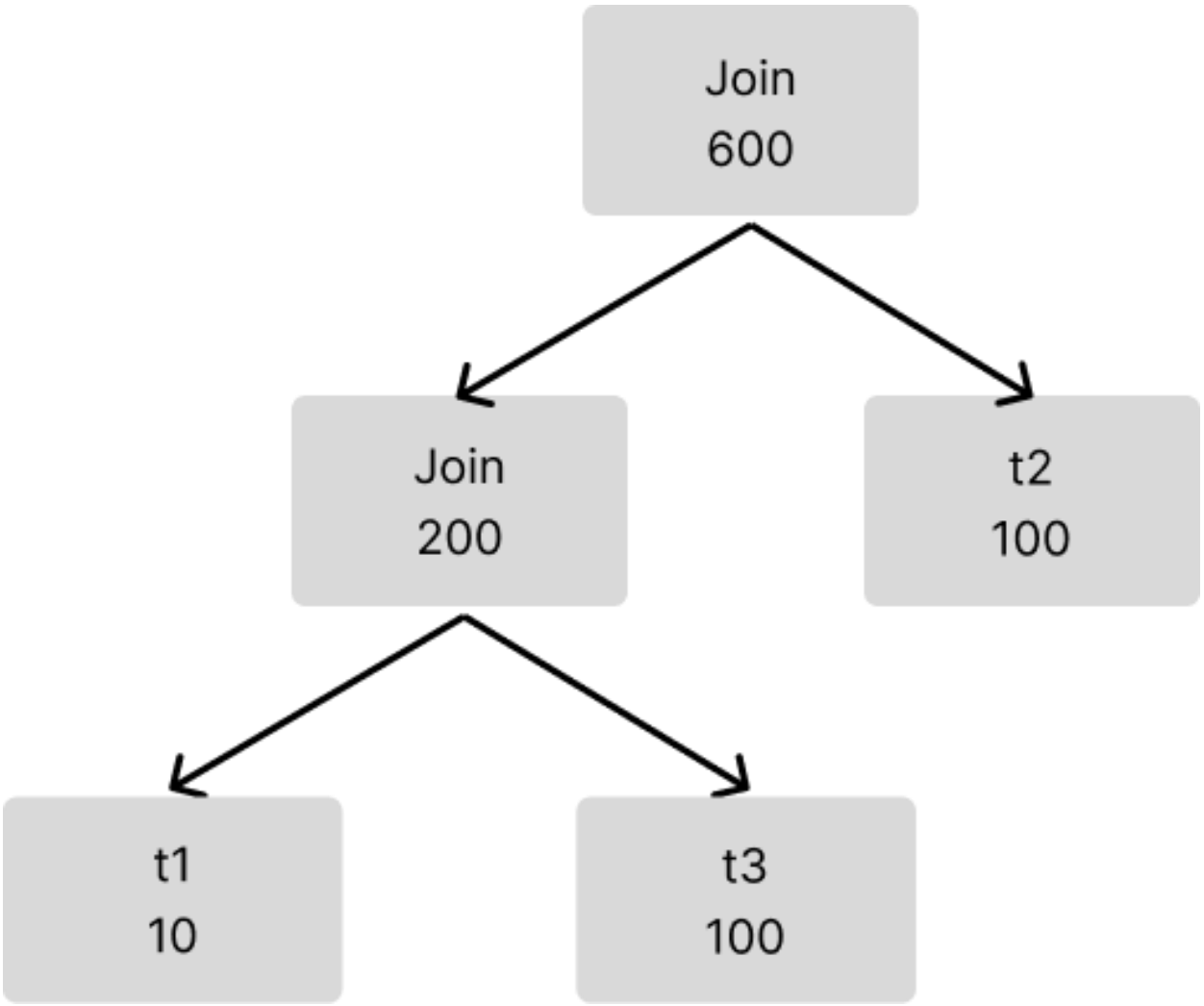


图 144: join-reorder-3

Join Reorder 动态规划算法实例

仍然以上述例子为例。动态规划算法会枚举所有的可能性，因此相对贪心算法必须从 `t1` 表开始枚举，动态规划算法可以枚举如下的Join 顺序。

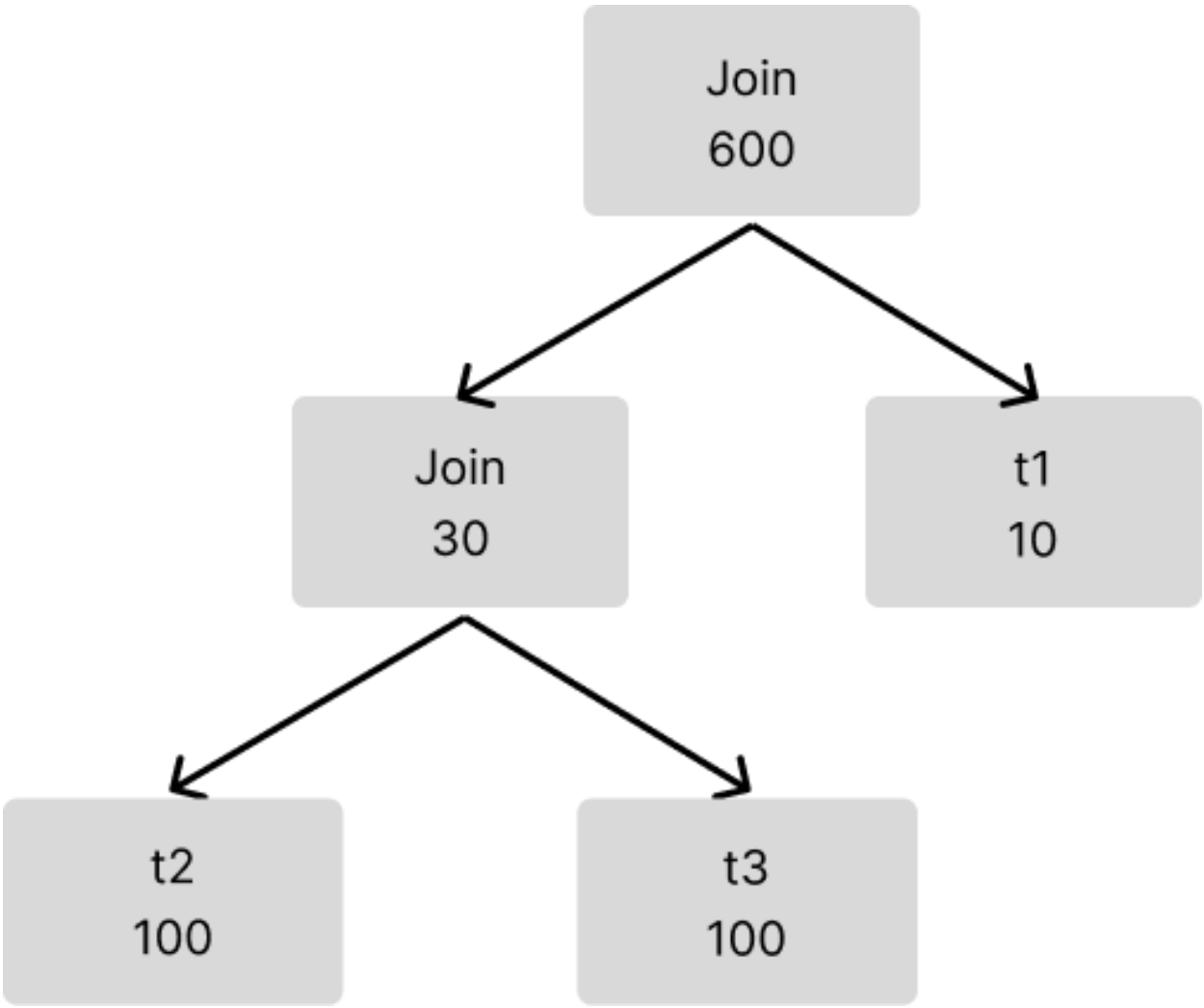


图 145: join-reorder-4

当该选择比贪心算法更优时，动态规划算法便可以选择到更优的Join 顺序。

相应地，因为会枚举所有的可能性，动态规划算法会消耗更多的时间，也会更容易受统计信息影响。

Join Reorder 算法的控制

目前Join Reorder 算法由变量`tidb_opt_join_reorder_threshold` 控制，当参与Join Reorder 的节点个数大于该阈值时选择贪心算法，反之选择动态规划算法。

Join Reorder 算法限制

当前的Join Reorder 算法存在如下限制：

- 受结果集的计算算法所限并不会保证一定会选到合适的Join order
- 是否启用 Outer Join 的Join Reorder 功能由系统变量`tidb_enable_outer_join_reorder` 控制。
- 目前动态规划算法无法进行 Outer Join 的 Join Reorder。

目前TiDB 中支持使用 `STRAIGHT_JOIN` 语法来强制指定一种Join 顺序，参见语法元素说明。

8.3.3.2.10 从窗口函数中推导TopN 或 Limit

窗口函数是一种常见的 SQL 函数。对于 ROW_NUMBER() 或者 RANK() 等编号相关的窗口函数，一种常见的用法是在进行窗口函数求值之后，对求值的结果进行过滤，例如：

```
SELECT * FROM (SELECT ROW_NUMBER() OVER (ORDER BY a) AS rownumber FROM t) dt WHERE rownumber <= 3
```

按照正常的 SQL 执行流程，TiDB 需要先对t 表的所有数据进行排序，然后为每一行都求得相应的 ROW_NUMBER() 结果，最后再进行 rownumber <= 3 的过滤。

从 v7.0.0 开始，TiDB 支持从窗口函数中推导TopN 或 Limit 算子。通过该优化规则，TiDB 可以将原始 SQL 等价改写成以下形式：

```
WITH t_topN AS (SELECT a FROM t1 ORDER BY a LIMIT 3) SELECT * FROM (SELECT ROW_NUMBER() OVER (
    ↪ ORDER BY a) AS rownumber FROM t_topN) dt WHERE rownumber <= 3
```

可以看出，改写后，TiDB 可以从窗口函数与后续的过滤条件中推导出一个TopN 算子，相比于原始 SQL 中的 Sort 算子（对应 ORDER BY），TopN 算子的运行效率远高于 Sort 算子，而且TiKV 和 TiFlash 均支持TopN 算子的下推，因此能进一步提升改写后的 SQL 的性能。

从窗口函数中推导TopN 或 Limit 默认关闭。你可以通过将 session 变量tidb_opt_derive_topn 设置为 ON 开启该功能。

开启后，如需关闭，可以进行以下操作之一：

- 设置 session 变量tidb_opt_derive_topn 为 false。
- 可参照优化规则及表达式下推的黑名单中的关闭方法。

限制

- 目前仅 ROW_NUMBER() 窗口函数支持 SQL 语句改写。
- 只有当 SQL 语句的过滤条件是针对 ROW_NUMBER() 结果而且过滤条件为< 或者 <= 时，TiDB 才支持改写 SQL 语句。

示例

以下通过一些例子对该优化规则进行说明。

不包含 PARTITION BY 的窗口函数

示例 1：不包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id int, value int);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER () AS rownumber FROM t) dt WHERE rownumber <= 3;
```

```
+                                  +         +           +               +-------------------------------------------------------------------
    ↪
| id                               | estRows | task      | access object | operator info
    ↪                                                                    |
+                                  +         +           +               +                                                                   -------------------
    ↪
 ---------------------------------- ---------- ------------ ----------------
```

```
| Projection_9                    | 2.40     | root      |                | Column#5
    ↪                                                                          |
|  └─Selection_10                  | 2.40     | root      |                | le(Column#5, 3)
    ↪                                                                   |
|    └─Window_11                   | 3.00     | root      |                | row_number()->Column
    ↪ #5 over(rows between current row and current row) |
|      └─Limit_15                  | 3.00     | root      |                | offset:0, count:3
    ↪                                                                 |
|        └─TableReader_26          | 3.00     | root      |                | data:Limit_25
    ↪                                                                     |
|          └─Limit_25              | 3.00     | cop[tikv] |                | offset:0, count:3
    ↪                                                                 |
|            └─TableFullScan_24    | 3.00     | cop[tikv] | table:t        | keep order:false,
    ↪ stats:pseudo                                                    |
+---------------------------------+----------+-----------+----------------+------------------------------------------------
    ↪
```

在该查询中，优化器从窗口函数中推导出来了 Limit 并将它下推给了TiKV。

示例 2：包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id int, value int);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (ORDER BY value) AS rownumber FROM t) dt WHERE
    ↪ rownumber <= 3;
```

```
+---------------------------------+----------+-----------+---------------+----------------------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                                                         |
+---------------------------------+----------+-----------+---------------+----------------------------------------------
    ↪
| Projection_10                   | 2.40     | root      |               | Column#5
    ↪                                                                                  |
|  └─Selection_11                  | 2.40     | root      |               | le(Column#5, 3)
    ↪                                                                       |
|    └─Window_12                   | 3.00     | root      |               | row_number()->Column
    ↪ #5 over(order by test.t.value rows between current row and current row) |
|      └─TopN_13                   | 3.00     | root      |               | test.t.value, offset
    ↪ :0, count:3                                                     |
|        └─TableReader_25          | 3.00     | root      |               | data:TopN_24
    ↪                                                                         |
|          └─TopN_24               | 3.00     | cop[tikv] |               | test.t.value, offset
    ↪ :0, count:3                                                     |
|            └─TableFullScan_23    | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                                                  |
```

```
+---------------------------------+----------+-----------+---------------+--------------------------------
    '→
```

在该查询中，优化器从窗口函数中推导出来了TopN 并将它下推给了TiKV。

包含 PARTITION BY 的窗口函数

> **注意：**
>
> 当窗口函数包含 PARTITION BY 时，该优化规则仅在partition 列是主键的前缀且主键是聚簇索引的时候才能生效。

示例 3：不包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) clustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY id1) AS rownumber FROM t) dt WHERE
    '→ rownumber <= 3;
```

```
+---------------------------------+---------+-----------+---------------+--------------------------------
    '→
| id                              | estRows | task      | access object | operator info
    '→                                                                                 |
+---------------------------------+---------+-----------+---------------+--------------------------------
    '→
| Projection_10                   | 2.40    | root      |               | Column#6
    '→                                                                                    |
| └─Selection_11                  | 2.40    | root      |               | le(Column#6, 3)
    '→                                                                                    |
|   └─Shuffle_26                  | 3.00    | root      |               | execution info:
    '→ concurrency:2, data sources:[TableReader_24]                                       |
|     └─Window_12                 | 3.00    | root      |               | row_number()->
     '→ Column#6 over(partition by test.t.id1 rows between current row and current row) |
|       └─Sort_25                 | 3.00    | root      |               | test.t.id1
    '→                                                                                      |
|         └─TableReader_24        | 3.00    | root      |               | data:Limit_23
    '→                                                                                      |
|           └─Limit_23            | 3.00    | cop[tikv] |               | partition by test.t
    '→ .id1, offset:0, count:3                                                            |
|             └─TableFullScan_22  | 3.00    | cop[tikv] | table:t       | keep order:false,
    '→ stats:pseudo                                                                       |
+---------------------------------+---------+-----------+---------------+--------------------------------
    '→
```

在该查询中，优化器从窗口函数中推导出来了 Limit 并将它下推给了 TiKV。值得一提的是这个 Limit 其实是 partition Limit，也就是说对于每个相同 id1 值组成的一组数据上都会进行一次 Limit。

示例 4：包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) clustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY id1 ORDER BY value1) AS rownumber
    '→ FROM t) dt WHERE rownumber <= 3;
```

```
+-----------------------------------+----------+-----------+--------------+--------------------------------
    '→
| id                                | estRows  | task      | access object | operator info
    '→
    '→ |
+-----------------------------------+----------+-----------+--------------+--------------------------------
    '→
| Projection_10                     | 2.40     | root      |               | Column#6
    '→
    '→ |
| └─Selection_11                    | 2.40     | root      |               | le(Column#6, 3)
    '→
    '→ |
|   └─Shuffle_23                    | 3.00     | root      |               | execution info:
    '→ concurrency:3, data sources:[TableReader_21]
    '→                                                                    |
|     └─Window_12                   | 3.00     | root      |               | row_number()->
    '→ Column#6 over(partition by test.t.id1 order by test.t.value1 rows between current row and
    '→  current row) |
|       └─Sort_22                   | 3.00     | root      |               | test.t.id1, test.t
    '→ .value1
    '→
    '→ |
|         └─TableReader_21          | 3.00     | root      |               | data:TopN_19
    '→
    '→ |
|           └─TopN_19               | 3.00     | cop[tikv] |               | partition by test.
    '→ t.id1 order by test.t.value1, offset:0, count:3
    '→                                                      |
|             └─TableFullScan_18    | 10000.00 | cop[tikv] | table:t       | keep order:false,
    '→ stats:pseudo
    '→                                                                                  |
+-----------------------------------+----------+-----------+--------------+--------------------------------
    '→
```

在该查询中，优化器从窗口函数中推导出来了TopN 并将它下推给了TiKV。需要注意的是，这个TopN 其实是 partition 的TopN, 也就是说对于每个相同 id1 值组成的一组数据上都会进行一次TopN。

示例 5：PARTITION BY 列不是主键的前缀

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) clustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY value1) AS rownumber FROM t) dt
    '→ WHERE rownumber <= 3;
```

```
+---------------------------------+----------+-----------+--------------+--------------------------------
    '→
| id                              | estRows  | task      | access object | operator info
    '→                                                                                                  |
+---------------------------------+----------+-----------+--------------+--------------------------------
    '→
| Projection_9                    | 8000.00  | root      |               | Column#6
    '→                                                                                                      |
| └─Selection_10                   | 8000.00  | root      |               | le(Column#6, 3)
    '→                                                                                                     |
|   └─Shuffle_15                   | 10000.00 | root      |               | execution info:
    '→ concurrency:5, data sources:[TableReader_13]                                                        |
|     └─Window_11                  | 10000.00 | root      |               | row_number()->Column
    '→ #6 over(partition by test.t.value1 rows between current row and current row) |
|       └─Sort_14                  | 10000.00 | root      |               | test.t.value1
    '→                                                                                                      |
|         └─TableReader_13         | 10000.00 | root      |               | data:
    '→ TableFullScan_12
    '→                                                                                                            |
|           └─TableFullScan_12     | 10000.00 | cop[tikv] | table:t       | keep order:false,
    '→ stats:pseudo                                                                                               |
+---------------------------------+----------+-----------+--------------+--------------------------------
    '→
```

在该查询中，因为partition 的列不是主键的前缀，所以 SQL 没有被改写。

示例 6：PARTITION BY 列是主键的前缀，但主键不是聚簇索引

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) nonclustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY id1) AS rownumber FROM t use index
    '→ ()) dt WHERE rownumber <= 3;
```

```
+---------------------------------+----------+-----------+--------------+--------------------------------
    '→
| id                              | estRows  | task      | access object | operator info
    '→                                                                                                  |
+---------------------------------+----------+-----------+--------------+--------------------------------
    '→
```

```
| Projection_9                      | 8000.00  | root      |               | Column#7
    ′→                                                                                                 |
| └─Selection_10                    | 8000.00  | root      |               | le(Column#7, 3)
    ′→                                                                                                 |
|   └─Shuffle_15                    | 10000.00 | root      |               | execution info:
    ′→ concurrency:5, data sources:[TableReader_13]                                                    |
|     └─Window_11                   | 10000.00 | root      |               | row_number()->Column
    ′→ #7 over(partition by test.t.id1 rows between current row and current row) |
|       └─Sort_14                   | 10000.00 | root      |               | test.t.id1
    ′→                                                                                                 |
|         └─TableReader_13          | 10000.00 | root      |               | data:
    ′→ TableFullScan_12
    ′→ |
|           └─TableFullScan_12      | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ′→ stats:pseudo                                                                                    |
+-----------------------------------+----------+-----------+---------------+--------------------------------
    ′→
```

在该查询中，即使 PARTITION 的列是主键的前缀，但是因为主键不是聚簇索引，所以 SQL 没被改写。

8.3.3.3 物理优化

8.3.3.3.1 物理优化

物理优化是基于代价的优化，为上一阶段产生的逻辑执行计划制定物理执行计划。这一阶段中，优化器会为逻辑执行计划中的每个算子选择具体的物理实现。逻辑算子的不同物理实现有着不同的时间复杂度、资源消耗和物理属性等。在这个过程中，优化器会根据数据的统计信息来确定不同物理实现的代价，并选择整体代价最小的物理执行计划。

理解TiDB 执行计划文档已对每个物理算子进行了一些介绍。在本章我们会重点介绍以下方面：

- 在索引的选择中会介绍TiDB 在一张表有多个索引时，如何选择最优的索引进行表的访问。
- 在统计信息简介中会介绍TiDB 收集了哪些统计信息来获得表的数据分布情况
- 在错误索引的解决方案中会介绍当发现TiDB 索引选错时，你应该使用那些手段来让它使用正确的索引
- 在Distinct 优化中会介绍在物理优化中会做的一个有关 DISTINCT 关键字的优化，在这一小节中会介绍它的优缺点以及如何使用它。
- 在代价模型中会介绍在物理优化时，TiDB 怎么通过代价模型来选择一个最优的执行计划。

8.3.3.3.2 索引的选择

从存储层读取数据是 SQL 计算过程中最为耗时的部分之一，TiDB 目前支持从不同的存储和不同的索引中读取数据，索引选择得是否合理将很大程度上决定一个查询的运行速度。

本章节将介绍TiDB 如何选择索引去读入数据，以及相关的一些控制索引选择的方式。

读表

在介绍索引的选择之前，首先要了解TiDB 有哪些读表的方式，这些方式的触发条件是什么，不同方式有什么区别，各有什么优劣。

读表算子

读表算子	触发条件	适用场景	说明
PointGet/BatchPoint-Get	读表的范围是一个或多个单点范围	任何场景	如果能被触发，通常被认为是最快的算子，因为其直接调用 kvget 的接口进行计算，不走 copro-cessor

读表算子	触发条件	适用场景	说明
TableReader	无	任何场景	该 TableReader 算子用于 TiKV。从 TiKV 端直接扫描表数据，一般被认为是效率最低的算子，除非在 `_tidb_rowid` ↩ 这一列上存在范围查询，或者无其他可以选择的读表算子时，才会选择这个算子

读表算子	触发条件	适用场景	说明
TableReader	表在TiFlash节点上存在副本	需要读取的列比较少，但是需要计算的行很多	该TableReader算子用于TiFlash。TiFlash是列式存储，如果需要对少量的列和大量的行进行计算，一般会选择这个算子
IndexReader	表有一个或多个索引，且计算所需的列被包含在索引里	存在较小的索引上的范围查询，或者对索引列有顺序需求的时候	当存在多个索引的时候，会根据估算代价选择合理的索引
IndexLookupReader	表有一个或多个索引，且计算所需的列不完全被包含在索引里	同IndexReader	因为计算列不完全被包含在索引里，所以读完索引后需要回表，这里会比IndexReader多一些开销

读表算子	触发条件	适用场景	说明
IndexMerge	表有多个索引或多值索引	使用多值索引或同时使用多个索引的时候	可以通过optimizer hints 指定使用该算子，或让优化器根据代价估算自动选择该算子，参见用 EXPLAIN 查看索引合并的 SQL 执行计划

> **注意：**
>
> TableReader 是基于 `_tidb_rowid` 的索引，TiFlash 是列存索引，所以索引的选择即是读表算子的选择。

索引的选择

TiDB 基于规则或基于代价来选择索引。基于的规则包括前置规则和 Skyline-Pruning。在选择索引时，TiDB 会先尝试前置规则。如果存在索引满足某一条前置规则，则直接选择该索引。否则，TiDB 会采用 Skyline-Pruning 来排除不合适的索引，然后基于每个读表算子的代价估算，选择代价最小的索引。

基于规则选择

前置规则

TiDB 采用如下的启发式前置规则来选择索引：

- 规则 1：如果存在索引满足“唯一性索引全匹配+不需要回表（即该索引生成的计划是IndexReader）”时，直接选择该索引。
- 规则 2：如果存在索引满足“唯一性索引全匹配+需要回表（即该索引生成的计划是 IndexLookupReader）”时，选择满足该条件且回表行数最小的索引作为候选索引。
- 规则 3：如果存在索引满足“普通索引不需要回表+读取行数小于一定阈值”时，选择满足该条件且读取行数最小的索引作为候选索引。

- 规则 4：如果规则 2 和 3 之中仅选出一条候选索引，则选择该候选索引。如果规则 2 和 3 均选出候选索引，则选择读取行数（读索引行数+ 回表行数）较小的索引。

上述规则中的“索引全匹配”指每个索引列上均存在等值条件。在执行 EXPLAIN FORMAT = 'verbose' ... 语句时，如果前置规则匹配了某一索引，TiDB 会输出一条 NOTE 级别的 warning 提示该索引匹配了前置规则。

在以下示例中，因为索引 idx_b 满足规则 2 中“唯一性索引全匹配+需要回表”的条件，TiDB 选择索引 idx_b 作为访问路径，SHOW WARNING 返回了索引 idx_b 命中前置规则的提示。

```
mysql> CREATE TABLE t(a INT PRIMARY KEY, b INT, c INT, UNIQUE INDEX idx_b(b));
Query OK, 0 rows affected (0.01 sec)

mysql> EXPLAIN FORMAT = 'verbose' SELECT b, c FROM t WHERE b = 3 OR b = 6;
+--
    '→ ----------------+---------+---------+------+-------------------------+-------------------------------
    '→
| id                | estRows | estCost | task | access object           | operator info
    '→               |
+--
    '→ ----------------+---------+---------+------+-------------------------+-------------------------------
    '→
| Batch_Point_Get_5 | 2.00    | 8.80    | root | table:t, index:idx_b(b) | keep order:false, desc
    '→ :false |
+--
    '→ ----------------+---------+---------+------+-------------------------+-------------------------------
    '→
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+--
    '→     +      +                                                                                          +
    '→ -----+-------+-------------------------------------------------------------------------------------------
| Level | Code | Message
    '→                                                                                                      |
+--
    '→     +      +                                                                                          +
    '→ -----+-------+-------------------------------------------------------------------------------------------
| Note  | 1105 | unique index idx_b of t is selected since the path only has point ranges with
    '→ double scan |
+--
    '→     +      +                                                                                          +
    '→ -----+-------+-------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

Skyline-Pruning

Skyline-Pruning 是一个针对索引的启发式过滤规则，能降低错误估算导致选错索引的概率。Skyline-Pruning 从以

下三个维度衡量一个索引的优劣：

- 索引的列涵盖了多少访问条件。“访问条件”指的是可以转化为某列范围的 where 条件，如果某个索引的列集合涵盖的访问条件越多，那么它在这个维度上更优。
- 选择该索引读表时，是否需要回表（即该索引生成的计划是IndexReader 还是IndexLookupReader）。不用回表的索引在这个维度上优于需要回表的索引。如果均需要回表，则比较索引的列涵盖了多少过滤条件。过滤条件指的是可以根据索引判断的 where 条件。如果某个索引的列集合涵盖的访问条件越多，则回表数量越少，那么它在这个维度上越优。
- 选择该索引是否能满足一定的顺序。因为索引的读取可以保证某些列集合的顺序，所以满足查询要求顺序的索引在这个维度上优于不满足的索引。

对于以上三种维度，如果索引 idx_a 在这三个维度上都不比 idx_b 差，且有一个维度比 idx_b 好，那么 TiDB 会优先选择 idx_a。在执行 EXPLAIN FORMAT = 'verbose' ... 语句时，如果 Skyline-Pruning 排除了某些索引，TiDB 会输出一条 NOTE 级别的 warning 提示哪些索引在 Skyline-Pruning 排除之后保留下来。

在以下示例中，索引 idx_b 和 idx_e 均劣于 idx_b_c，因而被 Skyline-Pruning 排除，SHOW WARNING 的返回结果显示了经过 Skyline-Pruning 后剩余的索引。

```
mysql> CREATE TABLE t(a INT PRIMARY KEY, b INT, c INT, d INT, e INT, INDEX idx_b(b), INDEX
    '-> idx_b_c(b, c), INDEX idx_e(e));
Query OK, 0 rows affected (0.01 sec)

mysql> EXPLAIN FORMAT = 'verbose' SELECT * FROM t WHERE b = 2 AND c > 4;
+--
    '-> ----------------------------+---------+---------+-----------+-----------------------------+-----------
    '->
| id                            | estRows | estCost | task      | access object               |
    '-> operator info                                       |
+--
    '-> ----------------------------+---------+---------+-----------+-----------------------------+-----------

    '->
| IndexLookUp_10                | 33.33   | 738.29  | root      |                             |
    '->                                                     |
| ├─IndexRangeScan_8(Build)     | 33.33   | 2370.00 | cop[tikv] | table:t, index:idx_b_c(b, c) |
    '->  range:(2 4,2 +inf], keep order:false, stats:pseudo |
| └─TableRowIDScan_9(Probe)     | 33.33   | 2370.00 | cop[tikv] | table:t                      |
    '->  keep order:false, stats:pseudo                     |
+--
    '-> ----------------------------+---------+---------+-----------+-----------------------------+-----------
    '->
3 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+--
    '-> -----+------+-----------------------------------------------------------------------------------+
    '->
```

```
| Level | Code | Message
  ↪                                                                              |
+--
  ↪ -----+-------+------------------------------------------------------------------------------+
  ↪
| Note  | 1105 | [t,idx_b_c] remain after pruning paths for t given Prop{SortItems: [], TaskTp:
  ↪ rootTask} |
+--
  ↪ -----+-------+------------------------------------------------------------------------------+
  ↪
1 row in set (0.00 sec)
```

基于代价选择

在使用 Skyline-Pruning 规则排除了不合适的索引之后，索引的选择完全基于代价估算，读表的代价估算需要考虑以下几个方面：

- 索引的每行数据在存储层的平均长度。
- 索引生成的查询范围的行数量。
- 索引的回表代价。
- 索引查询时的范围数量。

根据这些因子和代价模型，优化器会选择一个代价最低的索引进行读表。

代价选择调优的常见问题

1. 估算的行数量不准确？

   一般是统计信息过期或者准确度不够造成的，可以重新执行 `analyze table` 或者修改 `analyze table` 的参数。

2. 统计信息准确，为什么读TiFlash 更快，而优化器选择了TiKV？

   目前区别TiFlash 和 TiKV 的代价模型还比较粗糙，可以调小 `tidb_opt_seek_factor` 的值，让优化器倾向于选择TiFlash。

3. 统计信息准确，某个索引要回表，但是它比另一个不用回表的索引实际执行更快，为什么选择了不用回表的索引？

   碰到这种情况，可能是代价估算时对于回表的代价计算得过大，可以调小 `tidb_opt_network_factor`，降低回表的代价。

控制索引的选择

通过Optimizer Hints 可以实现单条查询对索引选择的控制。

- `USE_INDEX`/`IGNORE_INDEX` 可以强制优化器使用/不使用某些索引。`FORCE_INDEX` 和 `USE_INDEX` 的作用相同。
- `READ_FROM_STORAGE` 可以强制优化器对于某些表选择TiKV/TiFlash 的存储引擎进行查询。

使用多值索引

多值索引和普通索引有所不同，TiDB 目前只会使用IndexMerge 来访问多值索引。因此要想使用多值索引进行数据访问，请确保tidb_enable_index_merge 被设置为 ON。

目前TiDB 支持将 json_member_of、json_contains 和 json_overlaps 条件自动转换成IndexMerge 来访问多值索引。既可以依赖优化器根据代价自动选择，也可通过use_index_merge optimizer hint 或use_index 指定选择多值索引，见下面例子：

```
mysql> CREATE TABLE t1 (j JSON, INDEX idx((CAST(j->'$.path' AS SIGNED ARRAY)))); -- 使用 '$.path'
    '-> 作为路径创建多值索引
Query OK, 0 rows affected (0.04 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t1, idx) */ * FROM t1 WHERE (1 MEMBER OF (j->'$.path'))
    '-> ;
+--
    '->                                +          +            +
    '-> --------------------------------- ---------- ------------
| id                               | estRows | task      | access object
    '->                                                            | operator info
    '->                                                       |
+--
    '->                                +          +            +
    '-> --------------------------------- ---------- ------------
| Selection_5                      | 8000.00 | root      |
    '->                                                                                |
    '-> json_memberof(cast(1, json BINARY), json_extract(test.t1.j, "$.path")) |
| └─IndexMerge_8                    | 10.00   | root      |
    '->                                                                                | type: union
    '->                                                                  |
|   ├─IndexRangeScan_6(Build)       | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract
    '-> (`j`, _utf8'$.path') as signed array)) | range:[1,1], keep order:false, stats:pseudo
    '->                               |
|   └─TableRowIDScan_7(Probe)       | 10.00   | cop[tikv] | table:t1
    '->                                                                                | keep order:false,
    '-> stats:pseudo                                          |
+--
    '->                                +          +            +
    '-> --------------------------------- ---------- ------------
4 rows in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t1, idx) */ * FROM t1 WHERE JSON_CONTAINS((j->'$.path')
    '-> , '[1, 2, 3]');
+--
    '->                                  +          +            +
    '->
    ------------------------------------------------------------
| id                               | estRows | task      | access object
```

```
    '→                                                                | operator info
    '→                              |
+--
    '→ ------------------------------+----------+------------+-------------------------------------------------
    '→
| IndexMerge_9                    | 10.00    | root       |
    '→                                                                                 | type:
    '→ intersection                            |
| ├─IndexRangeScan_5(Build)       | 10.00    | cop[tikv] | table:t1, index:idx(cast(json_extract(`j
    '→ `, _utf8'$.path') as signed array)) | range:[1,1], keep order:false, stats:pseudo |
| ├─IndexRangeScan_6(Build)       | 10.00    | cop[tikv] | table:t1, index:idx(cast(json_extract(`j
    '→ `, _utf8'$.path') as signed array)) | range:[2,2], keep order:false, stats:pseudo |
| ├─IndexRangeScan_7(Build)       | 10.00    | cop[tikv] | table:t1, index:idx(cast(json_extract(`j
    '→ `, _utf8'$.path') as signed array)) | range:[3,3], keep order:false, stats:pseudo |
| └─TableRowIDScan_8(Probe)       | 10.00    | cop[tikv] | table:t1
    '→                                                                                 | keep order:false,
    '→ stats:pseudo                |
+--
    '→ ------------------------------+----------+------------+-------------------------------------------------
    '→
5 rows in set (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t1, idx) */ * FROM t1 WHERE JSON_OVERLAPS((j->'$.path')
    '→ , '[1, 2, 3]');
+--
    '→ --------------------------------+-----------------+            +
    '→                                                  ------------ ----------------------------------------------
| id                              | estRows | task       | access object
    '→                                                                | operator info
    '→                                                                     |
+--
    '→ --------------------------------+-----------------+            +
    '→                                                  ------------ ----------------------------------------------
| Selection_5                     | 8000.00 | root       |
    '→                                                                                 |
    '→ json_overlaps(json_extract(test.t1.j, "$.path"), cast("[1, 2, 3]", json BINARY)) |
| └─IndexMerge_10                  | 10.00    | root       |
    '→                                                                                 | type: union
    '→                                                                              |
|   ├─IndexRangeScan_6(Build)      | 10.00    | cop[tikv] | table:t1, index:idx(cast(json_extract
    '→ (`j`, _utf8'$.path') as signed array)) | range:[1,1], keep order:false, stats:pseudo
    '→                                        |
|   ├─IndexRangeScan_7(Build)      | 10.00    | cop[tikv] | table:t1, index:idx(cast(json_extract
    '→ (`j`, _utf8'$.path') as signed array)) | range:[2,2], keep order:false, stats:pseudo
    '→                                        |
```

```
|   ├─IndexRangeScan_8(Build)      | 10.00    | cop[tikv] | table:t1, index:idx(cast(json_extract
    '→ (`j`, _utf8'$.path') as signed array))| range:[3,3], keep order:false, stats:pseudo
    '→                                       |
|   └─TableRowIDScan_9(Probe)      | 10.00    | cop[tikv] | table:t1
    '→                                                                    | keep order:false,
    '→ stats:pseudo                                                       |
+--
    '→                                  +          +           +
    '→
6 rows in set, 1 warning (0.00 sec)
```

复合的多值索引，也一样可以使用IndexMerge 进行访问：

```
mysql> CREATE TABLE t2 (a INT, j JSON, b INT, INDEX idx(a, (CAST(j->'$.path' AS SIGNED ARRAY)), b
    '→ ));
Query OK, 0 rows affected (0.04 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t2, idx) */ * FROM t2 WHERE a=1 AND (1 MEMBER OF (j->'$
    '→ .path')) AND b=2;
+--
    '→                                  +          +           +
    '→ ---------------------------------- ---------- ------------
| id                                | estRows | task      | access object
    '→                                                                    | operator info
    '→                                                          |
+--
    '→                                  +          +           +
    '→ ---------------------------------- ---------- ------------
| Selection_5                       | 0.01    | root      |
    '→                                                                                  |
    '→ json_memberof(cast(1, json BINARY), json_extract(test.t2.j, "$.path")) |
| └─IndexMerge_8                    | 0.00    | root      |
    '→                                                                                  | type:
    '→  union                                                          |
|   ├─IndexRangeScan_6(Build)      | 0.00    | cop[tikv] | table:t2, index:idx(a, cast(
    '→ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 1 2,1 1 2], keep order:
    '→ false, stats:pseudo                    |
|   └─TableRowIDScan_7(Probe)      | 0.00    | cop[tikv] | table:t2
    '→                                                                               | keep order:
    '→ false, stats:pseudo                                          |
+--
    '→                                  +          +           +
    '→ ---------------------------------- ---------- ------------
4 rows in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t2, idx) */ * FROM t2 WHERE a=1 AND JSON_CONTAINS((j->'
```

```
    '→ $.path'), '[1, 2, 3]');
+--
    '→ ------------------------------+----------+------------+-------------------------------------------------
    '→
| id                            | estRows | task      | access object
    '→                                                                    | operator info
    '→                                  |
+--
    '→ ------------------------------+----------+------------+-------------------------------------------------
    '→
| IndexMerge_9                  | 0.10     | root      |
    '→                                                                                  | type:
    '→  intersection                              |
| ├─IndexRangeScan_5(Build)     | 0.10     | cop[tikv] | table:t2, index:idx(a, cast(json_extract
    '→ (`j`, _utf8'$.path') as signed array), b) | range:[1 1,1 1], keep order:false, stats:
    '→ pseudo |
| ├─IndexRangeScan_6(Build)     | 0.10     | cop[tikv] | table:t2, index:idx(a, cast(json_extract
    '→ (`j`, _utf8'$.path') as signed array), b) | range:[1 2,1 2], keep order:false, stats:
    '→ pseudo |
| ├─IndexRangeScan_7(Build)     | 0.10     | cop[tikv] | table:t2, index:idx(a, cast(json_extract
    '→ (`j`, _utf8'$.path') as signed array), b) | range:[1 3,1 3], keep order:false, stats:
    '→ pseudo |
| └─TableRowIDScan_8(Probe)     | 0.10     | cop[tikv] | table:t2
    '→                                                                        | keep order:
    '→ false, stats:pseudo                 |
+--
    '→ ------------------------------+----------+------------+
    '→                                                                    -------------------------------------------------
5 rows in set (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t2, idx) */ * FROM t2 WHERE a=1 AND JSON_OVERLAPS((j->'
    '→ $.path'), '[1, 2, 3]');
+--
    '→ ---------------------------------+-----------------+            +
    '→                                                  ------------ ------------------------------------------------
| id                              | estRows | task      | access object
    '→                                                                    | operator info
    '→                                                                  |
+--
    '→ ---------------------------------+-----------------+            +
    '→                                                  ------------ ------------------------------------------------
| Selection_5                     | 8.00    | root      |
    '→                                                                                |
    '→ json_overlaps(json_extract(test.t2.j, "$.path"), cast("[1, 2, 3]", json BINARY)) |
| └─IndexMerge_10                  | 0.10    | root      |
```

```
    '→                                                                                                  | type:
    '→  union                                                                                          |
|   ├─IndexRangeScan_6(Build)      | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(
    '→ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 1,1 1], keep order:false
    '→ , stats:pseudo                                       |
|   ├─IndexRangeScan_7(Build)      | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(
    '→ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 2,1 2], keep order:false
    '→ , stats:pseudo                                       |
|   ├─IndexRangeScan_8(Build)      | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(
    '→ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 3,1 3], keep order:false
    '→ , stats:pseudo                                       |
|   └─TableRowIDScan_9(Probe)      | 0.10    | cop[tikv] | table:t2
    '→                                                                                                  | keep order:
    '→ false, stats:pseudo                                                                            |
+--
    '→                                  +         +           +
    '→
6 rows in set, 1 warning (0.00 sec)
```

对于由多个 member of 组成的 OR 条件，也可以使用IndexMerge 进行访问：

```
mysql> CREATE TABLE t3 (a INT, j JSON, INDEX idx(a, (CAST(j AS SIGNED ARRAY))));
Query OK, 0 rows affected (0.04 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE ((a=1 AND (1 member of (j))
    '→ )) OR ((a=2 AND (2 member of (j))));
+--
    '→                                  +         +           +
    '→ ---------------------------------- ---------- ------------
    '→
| id                               | estRows | task      | access object
    '→                                        | operator info
    '→
    '→ |
+--
    '→                                  +         +           +
    '→ ---------------------------------- ---------- ------------
| Selection_5                      | 0.08    | root      |
    '→                                                    | or(and(eq(test.t3.a, 1),
    '→ json_memberof(cast(1, json BINARY), test.t3.j)), and(eq(test.t3.a, 2), json_memberof(cast
    '→ (2, json BINARY), test.t3.j))) |
| └─IndexMerge_9                   | 0.10    | root      |
    '→                                                    | type: union
    '→
    '→ |
|   ├─IndexRangeScan_6(Build)      | 0.10    | cop[tikv] | table:t3, index:idx(a, cast(`j` as
    '→ signed array)) | range:[1 1,1 1], keep order:false, stats:pseudo
```

```
    '→
    '→ |
|   ├─IndexRangeScan_7(Build)      | 0.10    | cop[tikv] | table:t3, index:idx(a, cast(`j` as
    '→ signed array)) | range:[2 2,2 2], keep order:false, stats:pseudo
    '→
    '→ |
|   └─TableRowIDScan_8(Probe)      | 0.10    | cop[tikv] | table:t3
    '→                                          | keep order:false, stats:pseudo
    '→
    '→ |
+--
    '→                                +         +           +
    '→
```

以下场景暂时无法使用IndexMerge 来访问多值索引：

```
mysql> CREATE TABLE t4 (j JSON, INDEX idx((CAST(j AS SIGNED ARRAY))));
Query OK, 0 rows affected (0.04 sec)

-- 如果查询包含多个 json_contains 通过 OR 组成的条件，则无法通过 IndexMerge 访问索引
mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE (json_contains(j, '[1, 2]')
    '→ ) OR (json_contains(j, '[3, 4]'));
+--
    '→                        +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
| id                        | estRows  | task      | access object | operator info
    '→
    '→ |
+--
    '→                        +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
| TableReader_7             | 9600.00  | root      |               | data:Selection_6
    '→
    '→ |
|  └─Selection_6             | 9600.00  | cop[tikv] |               | or(json_contains(test.t3.j,
    '→ cast("[1, 2]", json BINARY)), json_contains(test.t3.j, cast("[3, 4]", json BINARY))) |
|    └─TableFullScan_5       | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    '→ pseudo
    '→ |
+--
    '→                        +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
3 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+         +      +                              +
 ---------- ------- ------------------------------
```

```
| Level   | Code | Message                     |
+---------+------+-----------------------------+
| Warning | 1105 | IndexMerge is inapplicable |
+---------+------+-----------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE (json_contains(j, '[1, 2]')
    '-> ) OR (json_contains(j, '[3, 4]'));
+--
    '-> ------------------------+----------+-----------+----------------+
    '->                                                                    ----------------------------------------
| id                        | estRows  | task      | access object | operator info
    '->
    '-> |
+--
    '-> ------------------------+----------+-----------+----------------+
    '->                                                                    ----------------------------------------
| TableReader_7             | 9600.00  | root      |               | data:Selection_6
    '->
    '-> |
| └─Selection_6             | 9600.00  | cop[tikv] |               | or(json_contains(test.t3.j,
    '-> cast("[1, 2]", json BINARY)), json_contains(test.t3.j, cast("[3, 4]", json BINARY))) |
|   └─TableFullScan_5       | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    '-> pseudo
    '-> |
+--
    '-> ------------------------+----------+-----------+----------------+
    '->                                                                    ----------------------------------------
3 rows in set, 1 warning (0.01 sec)

-- 如果查询中包含较为复杂的多层 OR / AND 嵌套形成的表达式，则无法通过 IndexMerge 访问索引
mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE ((1 member of (j)) AND (2
    '-> member of (j))) OR ((3 member of (j)) AND (4 member of (j)));
+--
    '-> ------------------------+----------+-----------+----------------+
    '->
                                                                       ----------------------------------------
| id                        | estRows  | task      | access object | operator info
    '->
    '-> |
+--
    '-> ------------------------+----------+-----------+----------------+
    '->
                                                                       ----------------------------------------
| Selection_5               | 8000.00  | root      |               | or(and(json_memberof(cast(1,
    '-> json BINARY), test.t3.j), json_memberof(cast(2, json BINARY), test.t3.j)), and(
    '-> json_memberof(cast(3, json BINARY), test.t3.j), json_memberof(cast(4, json BINARY), test.
```

```
    '→ t3.j))) |
| └─TableReader_7          | 10000.00 | root      |               | data:TableFullScan_6
    '→
    '→ |
|   └─TableFullScan_6      | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    '→ pseudo
    '→
    '→ |
+--
    '→                         +          +           +               +
    '→
3 rows in set, 2 warnings (0.00 sec)
```

受限于多值索引的特性，当多值索引无法生效时，使用use_index 可能会返回 Can't find a proper physical '→ plan for this query 的错误，而使用use_index_merge 不会，因此建议使用 use_index_merge：

```
mysql> EXPLAIN SELECT /*+ use_index(t3, idx) */ * FROM t3 WHERE ((1 member of (j)) AND (2 member
    '→ of (j))) OR ((3 member of (j)) AND (4 member of (j)));
ERROR 1815 (HY000): Internal : Cant find a proper physical plan for this query

mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE ((1 member of (j)) AND (2
    '→ member of (j))) OR ((3 member of (j)) AND (4 member of (j)));
+--
    '→                         +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
    '→
| id                         | estRows  | task      | access object | operator info
    '→
    '→ |
+--
    '→                         +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
| Selection_5                | 8000.00  | root      |               | or(and(json_memberof(cast(1,
    '→ json BINARY), test.t3.j), json_memberof(cast(2, json BINARY), test.t3.j)), and(
    '→ json_memberof(cast(3, json BINARY), test.t3.j), json_memberof(cast(4, json BINARY), test.
    '→ t3.j))) |
| └─TableReader_7            | 10000.00 | root      |               | data:TableFullScan_6
    '→
    '→ |
|   └─TableFullScan_6        | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    '→ pseudo
    '→
    '→ |
+--
    '→                         +          +           +               +
    '→ ------------------------ ---------- ----------- ----------------
3 rows in set, 2 warnings (0.00 sec)
```

8.3.3.3.3 统计信息简介

TiDB 使用统计信息来决定索引的选择。

统计信息版本

变量tidb_analyze_version 用于控制所收集到的统计信息。目前TiDB 中支持两种统计信息：tidb_analyze_version ′→ = 1 以及 tidb_analyze_version = 2。在 v5.3.0 及之后的版本中，该变量的默认值为 2。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。

Version 2 的统计信息避免了Version 1 中因为哈希冲突导致的在较大的数据量中可能产生的较大误差，并保持了大多数场景中的估算精度。

两种版本中，TiDB 维护的统计信息如下：

信息	Version 1	Version 2
表的总行数	√	√
列的 Count-Min Sketch	√	×
索引的 Count-Min Sketch	√	×
列的Top-N	√	√（改善了维护方式和精度）
索引的Top-N	√（维护精度不足，会产生较大误差）	√（改善了维护方式和精度）
列的直方图	√	√（直方图中不包含Top-N 中出现的值）
索引的直方图	√	√（直方图的桶中记录了各自的不同值的个数，且直方图不包
列的 NULL 值个数	√	√
索引的 NULL 值个数	√	√
列的平均长度	√	√
索引的平均长度	√	√

当tidb_analyze_version = 2时，如果执行ANALYZE 语句后发生OOM，需要设置全局变量tidb_analyze_version ′→ = 1，回退到Version 1，然后根据情况进行以下操作：

- 如果ANALYZE 语句是手动执行的，你需要手动ANALYZE 每张需要的表：

```
SELECT DISTINCT(CONCAT('ANALYZE TABLE ', table_schema, '.', table_name, ';')) FROM
    ′→ information_schema.tables, mysql.stats_histograms WHERE stats_ver = 2 AND table_id =
    ′→  tidb_table_id;
```

- 如果ANALYZE 语句是开启了自动ANALYZE 后 TiDB 自动执行的，使用以下 SQL 语句生成 DROP STATS 的语句并执行：

```
sql  SELECT DISTINCT(CONCAT('DROP STATS ', table_schema, '.', table_name, ';'))FROM information_schema
′→ .tables, mysql.stats_histograms WHERE stats_ver = 2 AND table_id = tidb_table_id;
```

- 如果上一条语句返回结果太长，不方便复制粘贴，可以将结果导出到临时文件后，再执行:

```
sql  SELECT DISTINCT... INTO outfile '/tmp/sql.txt'; mysql -h ${TiDB_IP} -u user -P ${TIDB_PORT}
'→ ... < '/tmp/sql.txt';
```

本文接下来将简单介绍其中出现的直方图和 Count-Min Sketch 以及Top-N 这些数据结构，以及详细介绍统计信息的收集和维护。

直方图简介

直方图是一种对数据分布情况进行描述的工具，它会按照数据的值大小进行分桶，并用一些简单的数据来描述每个桶，比如落在桶里的值的个数。在TiDB 中，会对每个表具体的列构建一个等深直方图，区间查询的估算便是借助该直方图来进行。

等深直方图，就是让落入每个桶里的值数量尽量相等。举个例子，比方说对于给定的集合 {1.6, 1.9, 1.9, 2.0, 2.4, 2.6, 2.7, 2.7, 2.8, 2.9, 3.4, 3.5}，并且生成 4 个桶，那么最终的等深直方图就会如下图所示，包含四个桶 [1.6, 1.9]，[2.0, 2.6]，[2.7, 2.8]，[2.9, 3.5]，其桶深均为 3。

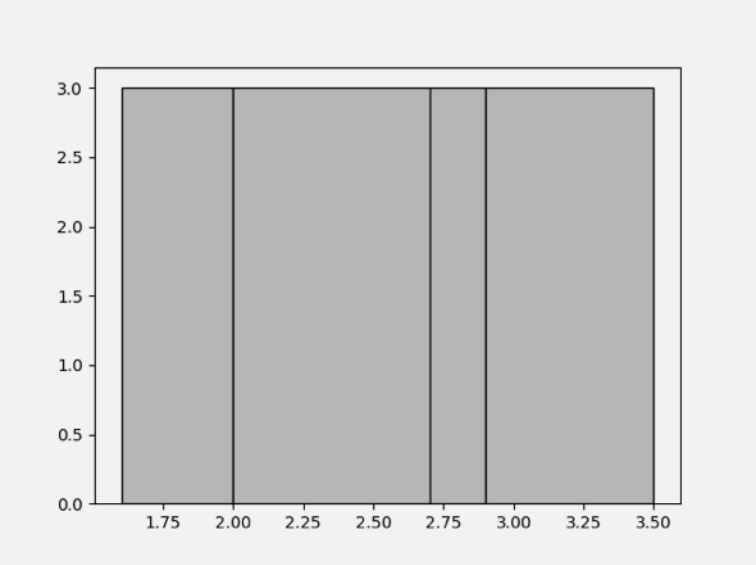


图 146: 等深直方图示例

在手动收集统计信息一节中有控制直方图桶数量上限的参数。当桶数量越多，直方图的估算精度就越高，不过也会同时增大统计信息的内存使用，可以视具体情况来做调整。

Count-Min Sketch

Count-Min Sketch 是一种哈希结构，当查询中出现诸如`a = 1` 或者 `IN` 查询（如 `a in (1, 2, 3)`）这样的等值查询时，TiDB 便会使用这个数据结构来进行估算。

由于 Count-Min Sketch 是一个哈希结构，就有出现哈希碰撞的可能。当在 `EXPLAIN` 语句中发现等值查询的估算偏离实际值较大时，就可以认为是一个比较大的值和一个比较小的值被哈希到了一起。这时有以下两种手段来避免这个情况：

- 修改手动收集统计信息中提到的 `WITH NUM TOPN` 参数。TiDB 会将出现频率前x 大的数据单独储存，之后的数据再储存到 Count-Min Sketch 中。因此可以调大这个值来避免一个比较大的值和一个比较小的值被哈希到一起。在TiDB 中，这个参数的默认值是 20，最大可以设置为 1024。

- 修改统计信息的收集-手动收集中提到的 WITH NUM CMSKETCH DEPTH 和 WITH NUM CMSKETCH WIDTH 两个参数，这两个参数会影响哈希的桶数和碰撞概率，可是适当调大来减少冲突概率，同时它会影响统计信息的内存使用，可以视具体情况来调整。在TiDB 中，DEPTH 的默认值是 5，WIDTH 的默认值是 2048。

Top-N values

Top-N 即是这个列或者这个索引中，出现次数前n 的值。TiDB 会记录前n 个值的具体的值以及出现次数。

统计信息的收集

手动收集

可以通过执行 ANALYZE 语句来收集统计信息。

> 注意：
>
> 在 TiDB 中执行 ANALYZE TABLE 语句比在 MySQL 或 InnoDB 中耗时更长。InnoDB 采样的只是少量页面，但TiDB 会完全重构一系列统计信息。适用于 MySQL 的脚本会误以为执行 ANALYZE TABLE 耗时较短。
>
> 如需更快的分析速度，可将 tidb_enable_fast_analyze 设置为 1 来打开快速分析功能。该参数的默认值为 0。
>
> 快速分析功能开启后，TiDB 会随机采样约 10000 行的数据来构建统计信息。因此在数据分布不均匀或者数据量比较少的情况下，统计信息的准确度会比较差。可能导致执行计划不优，比如选错索引。如果可以接受普通 ANALYZE 语句的执行时间，则推荐关闭快速分析功能。
>
> tidb_enable_fast_analyze 为实验性功能，目前与 tidb_analyze_version=2 的统计信息不完全匹配。因此开启 tidb_enable_fast_analyze 时需要将 tidb_analyze_version 的值设置为 1。

全量收集

可以通过以下几种语法进行全量收集。

收集 TableNameList 中所有表的统计信息：

```
ANALYZE TABLE TableNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM
    ↪ SAMPLES|WITH FLOATNUM SAMPLERATE];
```

- WITH NUM BUCKETS 用于指定生成直方图的桶数量上限。
- WITH NUM TOPN 用于指定生成 TOPN 数目的上限。
- WITH NUM CMSKETCH DEPTH 用于指定 CM Sketch 的长。
- WITH NUM CMSKETCH WIDTH 用于指定 CM Sketch 的宽。
- WITH NUM SAMPLES 用于指定采样的数目。
- WITH FLOAT_NUM SAMPLERATE 用于指定采样率。

WITH NUM SAMPLES 与 WITH FLOAT_NUM SAMPLERATE 这两种设置对应了两种不同的收集采样的算法。

- WITH NUM SAMPLES 指定了采样集的大小，在TiDB 中是以蓄水池采样的方式实现。当表较大时，不推荐使用这种方式收集统计信息。因为蓄水池采样中间结果集会产生一定的冗余结果，会对内存等资源造成额外的压力。
- WITH FLOAT_NUM SAMPLERATE 是在 v5.3.0 中引入的采样方式，指定的采样率的大小，是取值范围 (0, 1] 的参数。在TiDB 中是以伯努利采样的方式实现，更适合对较大的表进行采样，在收集效率和资源使用上更有优势。

在 v5.3.0 之前TiDB 采用蓄水池采样的方式收集统计信息。自 v5.3.0 版本起，TiDB Version 2 的统计信息默认会选取伯努利采样的方式收集统计信息。若要重新使用蓄水池采样的方式采样，可以使用 WITH NUM SAMPLES 语句。

> 注意：
>
> 目前采样率基于自适应算法进行计算。当你通过 SHOW STATS_META 可以观察到一个表的行数时，可通过这个行数去计算采集 10 万行所对应的采样率。如果你观察不到这个值，可通过 TABLE_STORAGE_STATS 表的 TABLE_KEYS 列作为另一个参考来计算采样率。
>
> 通常情况下，STATS_META 相对 TABLE_KEYS 更可信，但是通过TiDB Lightning 等方式导入数据结束后，STATS_META 结果是 0。为了处理这个情况，你可以在 STATS_META 的结果远小于 TABLE_KEYS 的结果时，使用 TABLE_KEYS 计算采样率。

收集部分列的统计信息

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，这些被用到的列称为 PREDICATE COLUMNS。

如果一个表有很多列，收集所有列的统计信息会有较大的开销。为了降低开销，你可以只收集指定列或者 PREDICATE COLUMNS 的统计信息供优化器使用。

> 注意：
>
> 收集部分列的统计信息的功能仅适用于 tidb_analyze_version = 2 的情况。

- 如果要收集指定列的统计信息，请使用以下语法：

```
ANALYZE TABLE TableName COLUMNS ColumnNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    '→ CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

其中，ColumnNameList 表示指定列的名称列表。如果需要指定多列，请使用用逗号，分隔列名。例如，ANALYZE table t columns a, b。该语法除了收集指定表中指定列的统计信息，将同时收集该表中索引列的统计信息以及所有索引的统计信息。

> **注意：**
>
> 该语法为全量收集。例如，在使用该语法收集了a 列和b 列的统计信息之后，如果还想要增加收集c 列的统计信息，需要在语法中同时指定这三列 ANALYZE TABLE t columns a , b, c，而不是只指定新增的那一列 ANALYZE TABLE t COLUMNS c。

- 如果要收集 PREDICATE COLUMNS 的统计信息，请进行以下操作：

> **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。

1. 设置系统变量tidb_enable_column_tracking 的值为 ON 开启TiDB 对 PREDICATE COLUMNS 的收集。开启后，TiDB 将每隔 100 * stats-lease 时间将 PREDICATE COLUMNS 信息写入系统表 mysql.column_stats_usage。
2. 在业务的查询模式稳定以后，使用以下语法收集 PREDICATE COLUMNS 的统计信息。

```
ANALYZE TABLE TableName PREDICATE COLUMNS [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    → CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

   该语法除了收集指定表中 PREDICATE COLUMNS 的统计信息，将同时收集该表中索引列的统计信息以及所有索引的统计信息。

> **注意：**
>
> - 如果系统表 mysql.column_stats_usage 中没有关于该表的PREDICATE COLUMNS 记录，执行以上语句会收集该表中所有列的统计信息以及所有索引的统计信息。
> - 使用该语法收集统计信息后，当执行一种新的类型的 SQL 查询时，优化器可能会暂时使用旧的或者pseudo 的列统计信息，然后在下一次收集统计信息的时候收集该列的统计信息。

- 如果要收集所有列的统计信息以及所有索引的统计信息，可以使用以下语法：

```
ANALYZE TABLE TableName ALL COLUMNS [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[
    → WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

如果要持久化 ANALYZE 语句中列的配置（包括 COLUMNS ColumnNameList、PREDICATE COLUMNS、ALL COLUMNS），请设置系统变量 tidb_persist_analyze_options 的值设置为 ON 以开启ANALYZE 配置持久化特性。开启 ANALYZE 配置持久化特性后：

- 当 TiDB 自动收集统计信息或者你手动执行 ANALYZE 语句收集统计信息但未指定列的配置时，TiDB 会继续沿用之前持久化的配置。
- 当多次手动执行 ANALYZE 语句并指定列的配置时，TiDB 会使用最新一次 ANALYZE 指定的配置项覆盖上一次记录的持久化配置。

如果你想查看一个表中哪些列是 PREDICATE COLUMNS，哪些列的统计信息已经被收集，可以使用以下语法：

```
SHOW COLUMN_STATS_USAGE [ShowLikeOrWhere];
```

SHOW COLUMN_STATS_USAGE 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
Column_name	列名
Last_used_at	该列统计信息在最近一次查询优化中被用到的时间
Last_analyzed_at	该列统计信息最近一次被收集的时间

在以下示例中，执行 ANALYZE TABLE t PREDICATE COLUMNS; 后，TiDB 将收集b，c，d 列的统计信息，其中b 列是 PREDICATE COLUMN，c 列和d 列是索引列。

```
SET GLOBAL tidb_enable_column_tracking = ON;
Query OK, 0 rows affected (0.00 sec)

CREATE TABLE t (a INT, b INT, c INT, d INT, INDEX idx_c_d(c, d));
Query OK, 0 rows affected (0.00 sec)

-- 在此查询中优化器用到了 b 列的统计信息。
SELECT * FROM t WHERE b > 1;
Empty set (0.00 sec)

-- 等待一段时间（100 * stats-lease）后，TiDB 将收集的 `PREDICATE COLUMNS` 写入 mysql.
    '→ column_stats_usage。
-- 指定 `last_used_at IS NOT NULL` 表示显示 TiDB 收集到的 `PREDICATE COLUMNS`。
SHOW COLUMN_STATS_USAGE WHERE db_name = 'test' AND table_name = 't' AND last_used_at IS NOT NULL;
+---------+------------+----------------+-------------+---------------------+---------------------+
| Db_name | Table_name | Partition_name | Column_name | Last_used_at        | Last_analyzed_at |
+---------+------------+----------------+-------------+---------------------+---------------------+
| test    | t          |                | b           | 2022-01-05 17:21:33 | NULL                |
+---------+------------+----------------+-------------+---------------------+---------------------+
1 row in set (0.00 sec)

ANALYZE TABLE t PREDICATE COLUMNS;
Query OK, 0 rows affected, 1 warning (0.03 sec)

-- 指定 `last_analyzed_at IS NOT NULL` 表示显示收集过统计信息的列。
SHOW COLUMN_STATS_USAGE WHERE db_name = 'test' AND table_name = 't' AND last_analyzed_at IS NOT
    '→ NULL;
+--
    '→ -------+------------+----------------+-------------+---------------------+---------------------+
```

```
    '→
| Db_name | Table_name | Partition_name | Column_name | Last_used_at         | Last_analyzed_at
    '→    |
+--
    '→ -------+------------+----------------+-------------+---------------------+-------------------------------- +
    '→
| test    | t          |                | b           | 2022-01-05 17:21:33 | 2022-01-05 17:23:06
    '→  |
| test    | t          |                | c           | NULL                | 2022-01-05 17:23:06
    '→  |
| test    | t          |                | d           | NULL                | 2022-01-05 17:23:06
    '→  |
+--
    '→ -------+------------+----------------+-------------+---------------------+-------------------------------- +
    '→
3 rows in set (0.00 sec)
```

收集索引的统计信息

如果要收集 TableName 中IndexNameList 里所有索引的统计信息，请使用以下语法：

```
ANALYZE TABLE TableName INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH
    '→ WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

当IndexNameList 为空时，该语法将收集 TableName 中所有索引的统计信息。

> 注意：
>
> 为了保证前后统计信息的一致性，当设置 `tidb_analyze_version=2` 时，该语句也会收集整个表的统计信息（包括所有列和所有索引的统计信息）而不限于索引的统计信息。

收集分区的统计信息

- 如果要收集 TableName 中所有的 PartitionNameList 中分区的统计信息，请使用以下语法：

```
ANALYZE TABLE TableName PARTITION PartitionNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    '→ CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

- 如果要收集 TableName 中所有的 PartitionNameList 中分区的索引统计信息，请使用以下语法：

```
ANALYZE TABLE TableName PARTITION PartitionNameList INDEX [IndexNameList] [WITH NUM BUCKETS|
    '→ TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

- 当收集分区的统计信息时，如果只收集部分列的统计信息，请使用以下语法：

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。

```
ANALYZE TABLE TableName PARTITION PartitionNameList [COLUMNS ColumnNameList|PREDICATE
    ↪ COLUMNS|ALL COLUMNS] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM
    ↪  SAMPLES|WITH FLOATNUM SAMPLERATE];
```

动态裁剪模式下的分区表统计信息

分区表在开启动态裁剪模式的情况下，TiDB 将收集表级别的汇总统计信息，以下称 GlobalStats。目前 GlobalStats 由分区统计信息合并汇总得到。在动态裁剪模式开启的情况下，任一分区上的统计信息更新都会触发 GlobalStats 的更新。

注意：

- 当触发GlobalStats 更新时：
  - 若某些分区上缺少统计信息（比如新增的未analyze 过的分区），会停止生成 GlobalStats，并通过 warning 信息提示用户缺少分区的统计信息。
  - 若某些列的统计信息合并过程中，缺少某些分区在该列上的统计信息（在不同分区上 analyze 时指定了不同的列），会停止生成 GlobalStats，并通过 warning 信息提示用户缺少列在分区上的统计信息。
- 在动态裁剪模式开启的情况下，分区和表的ANALYZE 配置需要保持一致，因此ANALYZE TABLE TableName PARTITION PartitionNameList 命令后指定的 COLUMNS 配置和 WITH 后指定的 OPTIONS 配置将被忽略，并会通过 warning 信息提示用户。

增量收集

对于类似时间列这样的单调不减列，在进行全量收集后，可以使用增量收集来单独分析新增的部分，以提高分析的速度。

注意：

1. 目前只有索引提供了增量收集的功能
2. 使用增量收集时，必须保证表上只有插入操作，且应用方需要保证索引列上新插入的值是单调不减的，否则会导致统计信息不准，影响TiDB 优化器选择合适的执行计划

可以通过以下几种语法进行增量收集。

增量收集 TableName 中所有的IndexNameList 中的索引列的统计信息：

```
ANALYZE INCREMENTAL TABLE TableName INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    ↪ CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

增量收集 TableName 中所有的 PartitionNameList 中分区的索引列统计信息：

```
ANALYZE INCREMENTAL TABLE TableName PARTITION PartitionNameList INDEX [IndexNameList] [WITH NUM
    ↪ BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

自动更新

在发生增加，删除以及修改语句时，TiDB 会自动更新表的总行数以及修改的行数。这些信息会定期持久化下来，更新的周期为 20 * stats-lease。stats-lease 配置项的默认值是 3s，如果将其指定为 0，那么统计信息将不会自动更新。

相关系统变量

系统变量名	默认值	功能
tidb_auto_analyze_ratio	0.5	自动更新阈值
tidb_auto_analyze_start_time	00:00 +0000	一天中能够进行自动更新的开始时间
tidb_auto_analyze_end_time	23:59 +0000	一天中能够进行自动更新的结束时间

当某个表tbl 的修改行数与总行数的比值大于tidb_auto_analyze_ratio,并且当前时间在tidb_auto_analyze_start_time ↪ 和 tidb_auto_analyze_end_time 之间时，TiDB 会在后台执行 ANALYZE TABLE tbl 语句自动更新这个表的统计信息。

为了避免小表因为少量数据修改而频繁触发自动更新，当表的行数小于 1000 时，TiDB 不会触发对此表的自动更新。你可以通过 SHOW STATS_META 来查看表的行数情况。

关闭自动更新

如果发现自动更新统计信息消耗过多的资源，影响在线业务，可以通过系统变量tidb_enable_auto_analyze 关闭自动更新。

终止后台的 ANALYZE 任务

从TiDB v6.0 起，TiDB 支持通过 KILL 语句终止正在后台运行的 ANALYZE 任务。如果发现正在后台运行的 ANALYZE 任务消耗大量资源影响业务，你可以通过以下步骤终止该 ANALYZE 任务：

1. 执行以下 SQL 语句：

   ```
   SHOW ANALYZE STATUS
   ```

   查看 instance 列和 process_id 列获得正在执行后台 ANALYZE 任务的TiDB 实例地址和任务 ID。

2. 终止正在后台运行的 ANALYZE 任务。

- 如果enable-global-kill 的值为 true (默认为 true)，你可以直接执行 KILL TIDB ${id}; 语句。其中，${id} 为上一步中查询得到的后台 ANALYZE 任务的 ID。

- 如果 enable-global-kill 的值为 false，你需要先使用客户端连接到执行后台 ANALYZE 任务的TiDB 实例，然后再执行 KILL TIDB ${id}; 语句。如果使用客户端连接到其他TiDB 实例，或者客户端和TiDB 中间有代理，KILL 语句不能终止后台的 ANALYZE 任务。

关于 KILL 语句的更多信息，请参考 KILL [TIDB]。

控制ANALYZE 并发度

执行ANALYZE 语句的时候，你可以通过一些参数来调整并发度，以控制对系统的影响。

tidb_build_stats_concurrency

目前ANALYZE 执行的时候会被切分成一个个小的任务，每个任务只负责某一个列或者索引。tidb_build_stats_concurrency
↪ 可以控制同时执行的任务的数量，其默认值是 4。

tidb_distsql_scan_concurrency

在执行分析普通列任务的时候，tidb_distsql_scan_concurrency 可以用于控制一次读取的Region 数量，其默认值是 15。

tidb_index_serial_scan_concurrency

在执行分析索引列任务的时候，tidb_index_serial_scan_concurrency 可以用于控制一次读取的Region 数量，其默认值是 1。

ANALYZE 配置持久化

从 v5.4.0 起，TiDB 支持 ANALYZE 配置持久化功能，方便后续收集统计信息时沿用已有配置项。

TiDB 支持持久化的配置项包括：

配置项	对应的 ANALYZE 语法
直方图桶数	WITH NUM BUCKETS
TopN 个数	WITH NUM TOPN
采样数	WITH NUM SAMPLES
采样率	WITH FLOAT_NUM SAMPLERATE

配置项	对应的 ANALYZE 语法
ANALYZE 的列的类型	AnalyzeColumnOption ::= (‘ALL COLUMNS’ | ‘PREDICATE COLUMNS’ | ‘COLUMNS’ ColumnNameList)
ANALYZE 的列	ColumnNameList ::= Identifier (‘,’ Identifier)*

开启ANALYZE配置持久化功能

ANALYZE配置持久化功能默认开启（系统变量tidb_analyze_version为默认值2,tidb_persist_analyze_options ′→ 为默认值ON），用于记录手动执行ANALYZE语句时指定的持久化配置项。记录后，当TiDB下一次自动更新统计信息或者你手动收集统计信息但未指定配置项时，TiDB会按照记录的配置项收集统计信息。

多次手动执行ANALYZE语句并指定持久化配置项时，TiDB会使用最新一次ANALYZE指定的配置项覆盖上一次记录的持久化配置。

关闭ANALYZE配置持久化功能

要关闭 ANALYZE 配置持久化功能，请设置系统变量 tidb_persist_analyze_options 为 OFF。由于 ANALYZE 配置持久化功能在 tidb_analyze_version = 1 的情况下不适用，因此设置 tidb_analyze_version = 1 同样会达到关闭配置持久化的效果。

关闭 ANALYZE 配置持久化功能后，已持久化的配置记录不会被清除。因此，当再次开启该功能时，TiDB 会继续使用之前记录的持久化配置收集统计信息。

> **注意：**
>
> 当再次开启 ANALYZE 配置持久化功能时，如果之前记录的持久化配置项已经不适用当前的数据，请手动执行 ANALYZE 语句并指定新的持久化配置项。

分区表的ANALYZE 配置持久化功能

在静态裁剪模式下ANALYZE 分区表时，配置持久化遵守：

- ANALYZE TABLE 时会持久化表级别的配置和实际被ANALYZE 的所有分区的配置
- 分区的统计信息会继承使用表级别的持久化配置
- ANALYZE TABLE ··· PARTITION ··· WITH ··· 所指定的分区配置只持久化到分区级别，不会影响表级别的持久化配置
- 当 ANALYZE 语句指定了配置，且同时存在持久化配置时，按照语句> 分 区 > 表的优先级继承和重写配置信息

在动态裁剪模式下ANALYZE 分区表时，配置持久化遵守：

- ANALYZE TABLE 时只持久化表级别的配置
- 分区的统计信息会继承使用表级别的持久化配置
- GlobalStats 会使用表级别的持久化配置
- ANALYZE TABLE ··· PARTITION ··· WITH ··· 所指定的分区配置会被忽略，且不会被持久化

统计信息收集的内存限制

> **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。

TiDB 从 v6.1.0 开始引入了统计信息收集的内存限制，你可以通过`tidb_mem_quota_analyze` 变量来控制TiDB 更新统计信息时的最大总内存占用。

要合理地配置 `tidb_mem_quota_analyze` 值，你需要考虑集群的数据。在使用默认采样率的情况下，主要考虑列的数量、列上的值的大小，以及TiDB 的内存配置。你可参考以下建议来配置变量的最大值和最小值：

> **注意：**
>
> 以下配置建议仅供参考，实际配置需要在真实场景中测试确定。

- 最小值：需要大于TiDB 从集群上列最多的表收集统计信息时使用的最大内存。一个粗略的参考信息是，在测试集上，20 列的表在默认配置下，统计信息收集的最大内存使用约为 800 MiB；160 列的表在默认配置下，统计信息收集的最大内存使用约为 5 GiB。
- 最大值：需要小于集群在不进行统计信息收集时的内存空余量。

查看ANALYZE 状态

在执行 ANALYZE 时，可以通过 SQL 语句来查看当前 ANALYZE 的状态。

语法如下：

```
SHOW ANALYZE STATUS [ShowLikeOrWhere];
```

该语句会输出 ANALYZE 的状态，可以通过使用 ShowLikeOrWhere 来筛选需要的信息。

目前 SHOW ANALYZE STATUS 会输出 11 列，具体如下：

列名	说明
table_schema	数据库名
table_name	表名
partition_name	分区名
job_info	任务具体信息。如果分析索引，该信息会包含索引名。当 tidb_analyze_version =2 时，该信息会包含采样率
processed_rows	已经分析的行数
start_time	任务开始执行的时间
end_time	任务结束执行的时间
state	任务状态，包括pending（等待）、running（正在执行）、finished（执行成功）和 failed（执行失败）
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

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。

SHOW ANALYZE STATUS 仅显示最近的任务记录。从 TiDB v6.1 起，你可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史记录。

当设置了tidb_mem_quota_analyze 且TiDB 后台的统计信息自动更新任务的内存占用超过了这个阈值时，自动更新任务会重试。失败的任务和重试的任务都可以在 SHOW ANALYZE STATUS 的结果中查看。

当tidb_max_auto_analyze_time 大于 0 时，如果后台统计信息自动更新任务的执行时间超过这个阈值，该任务会被终止。

```
mysql> SHOW ANALYZE STATUS [ShowLikeOrWhere];
+--
    ↵ -------------+-------------+-----------------+-------------------------------------------------------------------
    ↵
| Table_schema | Table_name | Partition_name | Job_info
    ↵                                                                                 |
```

```
    '→ Processed_rows | Start_time          | End_time            | State     | Fail_reason
    '→                                                                        |
+--
    '→ -------------+-------------+-----------------+
    '→
| test          | sbtest1     |                 | retry auto analyze table all columns with 100 topn
    '→ , 0.055 samplerate                         |        2000000 | 2022-05-07 16:41:09 |
    '→ 2022-05-07 16:41:20 | finished | NULL
    '→                                                                                |
| test          | sbtest1     |                 | auto analyze table all columns with 100 topn, 0.5
    '→ samplerate                                 |              0 | 2022-05-07 16:40:50 |
    '→ 2022-05-07 16:41:09 | failed    | analyze panic due to memory quota exceeds, please try
    '→ with smaller samplerate |
```

统计信息的查看

你可以通过一些语句来查看统计信息的状态。

表的元信息

你可以通过 SHOW STATS_META 来查看表的总行数以及修改的行数等信息。

语法如下：

```
SHOW STATS_META [ShowLikeOrWhere];
```

其中，ShowLikeOrWhereOpt 部分的语法图为：

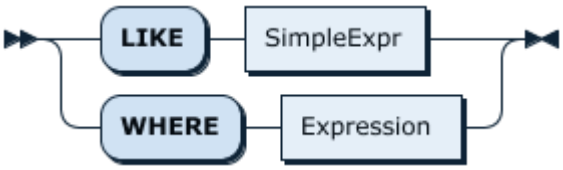


图 147: ShowLikeOrWhereOpt

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。

表的健康度信息

通过`SHOW STATS_HEALTHY` 可以查看表的统计信息健康度，并粗略估计表上统计信息的准确度。当`modify_count >= row_count` 时，健康度为 0；当 `modify_count < row_count` 时，健康度为`(1 - modify_count/row_count) * 100`。

语法如下：

```
SHOW STATS_HEALTHY [ShowLikeOrWhere];
```

`SHOW STATS_HEALTHY` 的语法图为：



图 148: ShowStatsHealthy

目前，`SHOW STATS_HEALTHY` 会输出 4 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
healthy	健康度

列的元信息

你可以通过 `SHOW STATS_HISTOGRAMS` 来查看列的不同值数量以及 NULL 数量等信息。

语法如下：

```
SHOW STATS_HISTOGRAMS [ShowLikeOrWhere];
```

该语句会输出所有列的不同值数量以及 NULL 数量等信息，你可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_HISTOGRAMS` 会输出 10 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名

列名	说明
is_index	是否是索引列
update_time	更新时间
distinct_count	不同值数量
null_count	NULL 的数量
avg_col_size	列平均长度
correlation	该列与整型主键的皮尔逊系数，表示两列之间的关联程度

直方图桶的信息

你可以通过 `SHOW STATS_BUCKETS` 来查看直方图每个桶的信息。

语法如下：

```
SHOW STATS_BUCKETS [ShowLikeOrWhere];
```

语法图：

SHOW STATS_BUCKETS:



图 149: SHOW STATS_BUCKETS

该语句会输出所有桶的信息，你可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_BUCKETS` 会输出 11 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名
is_index	是否是索引列
bucket_id	桶的编号
count	所有落在这个桶及之前桶中值的数量
repeats	最大值出现的次数
lower_bound	最小值
upper_bound	最大值
ndv	当前桶内不同值的个数。当 `tidb_analyze_version = 1` 时，该值恒为 `0`，没有实际意义。

Top-N 信息

你可以通过 `SHOW STATS_TOPN` 来查看当前TiDB 中收集的Top-N 值的信息。

语法如下：

```
SHOW STATS_TOPN [ShowLikeOrWhere];
```

目前 SHOW STATS_TOPN 会输出 7 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名
is_index	是否是索引列
value	该列的值
count	该值出现的次数

删除统计信息

可以通过执行 DROP STATS 语句来删除统计信息。

```
DROP STATS TableName;
```

该语句会删除 TableName 中所有的统计信息，如果是分区表，包括所有分区的统计信息和分区动态裁剪模式下生成的GlobalStats。

```
DROP STATS TableName PARTITION PartitionNameList;
```

该语句只删除 PartitionNameList 中对应的分区的统计信息。

```
DROP STATS TableName GLOBAL;
```

该语句只删除该表在分区动态裁剪模式下生成的GlobalStats。

统计信息的加载

默认情况下，列的统计信息占用空间大小不同，TiDB 对统计信息的加载方式也会不同。

- 对于count、 distinctCount、 nullCount 等占用空间较小的统计信息，只要有数据更新，TiDB 就会自动将对应的统计信息加载进内存供 SQL 优化阶段使用。
- 对于直方图、 TopN、 CMSketch 等占用空间较大的统计信息，为了确保 SQL 执行的性能，TiDB 会按需进行异步加载。例如，对于直方图，只有当某条 SQL 语句的优化阶段使用到了某列的直方图统计信息时，TiDB 才会将该列的直方图信息加载到内存。按需异步加载的优势是统计信息加载不会影响到 SQL 执行的性能，但在 SQL 优化时有可能使用不完整的统计信息。

从 v5.4.0 开始，TiDB 引入了统计信息同步加载的特性，支持执行当前 SQL 语句时将直方图、 TopN、 CMSketch 等占用空间较大的统计信息同步加载到内存，提高该 SQL 语句优化时统计信息的完整性。

要开启该特性，请将系统变量tidb_stats_load_sync_wait 的值设置为 SQL 优化等待加载列的完整统计信息的超时时间（单位为毫秒）。该变量默认值为 100，代表开启统计信息同步加载。

开启该特性后，你可以进一步配置该特性：

- 通过修改系统变量tidb_stats_load_pseudo_timeout 的值控制 SQL 优化等待超时后的行为。该变量默认值为 ON，代表超时后 SQL 优化过程不会使用任何列上的直方图、TopN 或 CMSketch。当设置该变量为 OFF 时，代表超时后 SQL 执行失败。
- 通过修改TiDB 配置项stats-load-concurrency 的值控制统计信息同步加载可以并发处理的最大列数。该配置项的默认值为 5。
- 通过修改TiDB 配置项stats-load-queue-size 的值设置统计信息同步加载最多可以缓存多少列的请求。该配置项的默认值为 1000。

在 TiDB 启动阶段，初始统计信息加载完成之前执行的 SQL 可能有不合理的执行计划，从而影响性能。为了避免这种情况，从 v7.1.0 开始，TiDB 引入了配置项force-init-stats。你可以控制 TiDB 启动时是否在统计信息初始化完成后再对外提供服务。该配置项默认关闭。

> **警告：**
>
> 轻量级的统计信息初始化目前为实验特性，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。

从 v7.1.0 开始，TiDB 引入了配置参数lite-init-stats 用于控制是否开启轻量级的统计信息初始化。

- 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。

lite-init-stats 默认值为 false，即关闭轻量级的统计信息初始化。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。

统计信息的导入导出

导出统计信息

统计信息的导出接口如下。

- 通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 的JSON 格式的统计信息：

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}
```

  示例如下：

```
curl -s http://127.0.0.1:10080/stats/dump/test/t1 -o /tmp/t1.json
```

- 通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 在指定时间上的JSON 格式统计信息。指定的时间应在GC SafePoint 之后。

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}/${
    ↪ yyyyMMddHHmmss}
```

导入统计信息

> **注意：**
>
> 启动 MySQL 客户端时，请使用 `--local-infile=1` 参数。

导入的统计信息一般是通过统计信息导出接口得到的json 文件。

语法如下：

```
LOAD STATS 'file_name';
```

`file_name` 为要导入的统计信息的文件名。

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。

从 v6.5.0 开始，TiDB 引入了锁定统计信息的特性。当一张表的统计信息被锁定以后，该表的统计信息将无法被修改，也无法对该表进行 ANALYZE 操作。用例如下：

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，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)
```

另请参阅

- LOAD STATS
- DROP STATS
- LOCK STATS
- UNLOCK STATS

- SHOW STATS_LOCKED

8.3.3.3.4 错误索引的解决方案

在观察到某个查询的执行速度达不到预期时，可能是它的索引使用有误，这时就需要通过一些手段来解决。通常可以先使用表的健康度信息来查看统计信息的健康度。根据健康度可以分为以下两种情况处理。

健康度较低

这意味着距离TiDB 上次 ANALYZE 很久了。这时可以先使用 ANALYZE 命令对统计信息进行更新。更新之后如果索引的使用上还是错误的，可以查看下一小节。

健康度接近 100%

这时意味着刚刚结束 ANALYZE 命令或者结束后不久。这时可能和TiDB 对行数的估算逻辑有关。

对于等值查询，错误索引可能是由Count-Min Sketch 引起的。这时可以先检查是不是这种特殊情况，然后进行对应的处理。

如果经过检查发现不是上面的可能情况，可以使用Optimizer Hints 中提到的 USE_INDEX 或者 use index 来强制选择索引。同时也可以使用执行计划管理中提到的方式来非侵入地更改查询的行为。

其他情况

除去上述情况外，也存在因为数据的更新导致现有所有索引都不再适合的情况。这时就需要对条件和数据分布进行分析，查看是否有新的索引可以加快查询速度，然后使用 ADD INDEX 命令增加新的索引。

8.3.3.3.5 Distinct 优化

本文档介绍可用于 DISTINCT 的优化，包括简单 DISTINCT 和聚合函数 DISTINCT 的优化。

简单 DISTINCT

通常简单的 DISTINCT 会被优化成GROUP BY 来执行。例如：

```
mysql> explain select DISTINCT a from t;
+--
    '→                          +         +           +               +
    '→
| id                        | estRows | task      | access object | operator info
    '→                                        |
+--
    '→                          +         +           +               +
    '→
| HashAgg_6                 | 2.40    | root      |               | group by:test.t.a, funcs:
    '→ firstrow(test.t.a)->test.t.a |
| └─TableReader_11          | 3.00    | root      |               | data:TableFullScan_10
    '→                                  |
|   └─TableFullScan_10      | 3.00    | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo                           |
+--
    '→                          +         +           +               +
    '→
```

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

聚合函数 DISTINCT

通常来说，带有DISTINCT的聚合函数会单线程的在TiDB侧执行。使用系统变量tidb_opt_distinct_agg_push_down 或者 TiDB 的配置项distinct-agg-push-down 控制优化器是否执行带有 DISTINCT 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从TiKV 读取所有数据，并在TiDB 侧执行 disctinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增了一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    '→                          +           +           +                +
    '→
| id                        | estRows   | task       | access object | operator info
    '→                              |
+--
    '→                          +           +           +                +
    '→ ------------------------ ----------- ------------ ----------------
| StreamAgg_6               | 1.00      | root       |               | funcs:count(distinct test.t.a)
    '→ ->Column#4 |
| └─TableReader_10          | 10000.00 | root       |               | data:TableFullScan_9
    '→                        |
|   └─TableFullScan_9       | 10000.00 | cop[tikv] | table:t        | keep order:false, stats:
    '→ pseudo             |
+--
    '→                          +           +           +                +
    '→ ------------------------ ----------- ------------ ----------------
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc select count(distinct a) from test.t;
+--
    '→                            +           +           +                +
    '→
| id                          | estRows   | task       | access object | operator info
    '→                                |
+--
    '→                            +           +           +                +
    '→
| HashAgg_8                   | 1.00      | root       |               | funcs:count(distinct test.t.
    '→ a)->Column#3 |
| └─TableReader_9             | 1.00      | root       |               | data:HashAgg_5
```

```
    '→                                |
|   └─HashAgg_5                | 1.00      | cop[tikv] |                | group by:test.t.a,
    '→                                |
|     └─TableFullScan_7        | 10000.00 | cop[tikv] | table:t        | keep order:false, stats:
    '→ pseudo                         |
+--  '→                                +           +           +                +
    '→
4 rows in set (0.00 sec)
```

8.3.3.3.6 代价模型

TiDB 在进行物理优化时会使用代价模型来进行索引选择和算子选择，如下图所示：

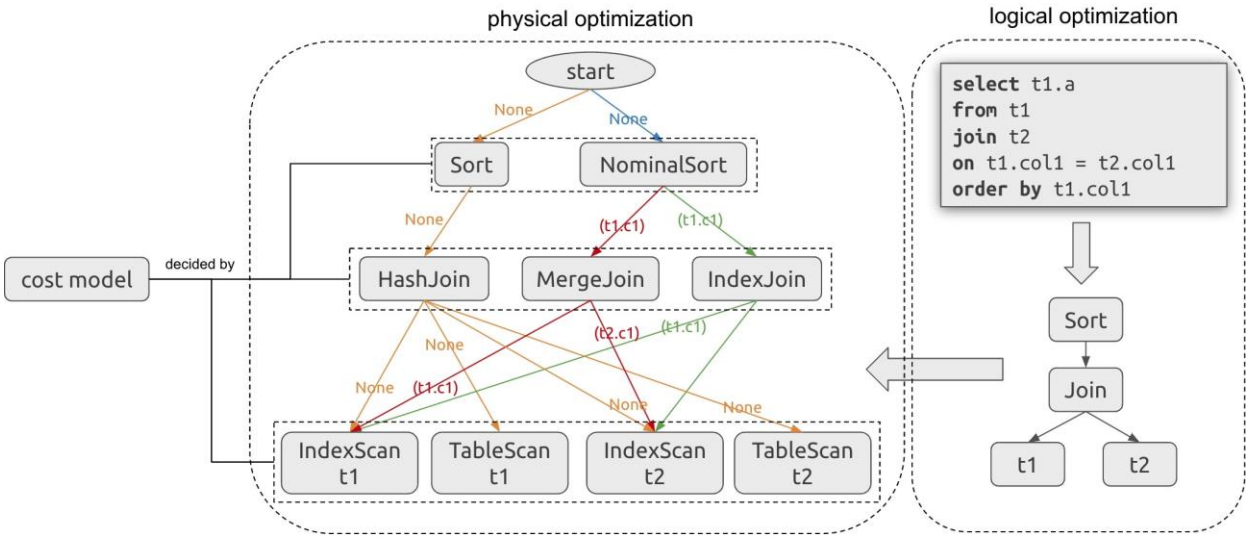


图 150: CostModel

TiDB 会计算每个索引的访问代价和计划中每个物理算子的执行代价（如 HashJoin、IndexJoin 等），选择代价最低的计划。

下面是一个简化的例子，用来解释代价模型的原理，比如有这样一张表：

```
mysql> SHOW CREATE TABLE t;
+--
    '→ ------+
    '→
| Table | Create Table
    '→
    '→ |
+--
    '→      +
    '→ ------
```

```
| t     | CREATE TABLE `t` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  `c` int(11) DEFAULT NULL,
  KEY `b` (`b`),
  KEY `c` (`c`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    '→      +
    '→ ------'
1 row in set (0.00 sec)
```

在处理查询 SELECT * FROM t WHERE b < 100 and c < 100 时，假设TiDB 对 b < 100 和 c < 100 的行数估计分别为 20 和 500，INT 类型索引行宽为 8，则TiDB 会分别计算两个索引的代价：

- 索引 b 的扫描代价= b < 100 的行数* 索引 b 的行宽= 20 * 8 = 160
- 索引 c 的扫描代价= c < 100 的行数* 索引 c 的行宽= 500 * 8 = 4000

由于扫描b 的代价更低，因此TiDB 会选择b 索引。

上述是一个简化后的例子，只是用于做原理解释，实际TiDB 的代价模型会更加复杂。

Cost Model Version 2

TiDB v6.2.0 引入了新的代价模型 Cost Model Version 2。

Cost Model Version 2 对代价公式进行了更精确的回归校准，调整了部分代价公式，比此前版本的代价公式更加准确。

你可以通过设置变量tidb_cost_model_version 来控制代价模型的版本。

> 注意：
> 切换代价模型版本可能引起执行计划的变动。

8.3.3.4 Prepare 语句执行计划缓存

TiDB 支持对 Prepare/Execute 请求的执行计划缓存。其中包括以下两种形式的预处理语句：

- 使用 COM_STMT_PREPARE 和 COM_STMT_EXECUTE 的协议功能；
- 执行 Prepare/Execute SQL 语句查询；

TiDB 优化器对这两类查询的处理是一样的：Prepare 时将参数化的 SQL 查询解析成AST（抽象语法树），每次 Execute 时根据保存的AST 和具体的参数值生成执行计划。

当开启执行计划缓存后，每条 Prepare 语句的第一次 Execute 会检查当前查询是否可以使用执行计划缓存，如果可以则将生成的执行计划放进一个由 LRU 链表构成的缓存中；在后续的 Execute 中，会先从缓存中获取

执行计划，并检查是否可用，如果获取和检查成功则跳过生成执行计划这一步，否则重新生成执行计划并放入缓存中。

对于某些非 PREPARE 语句，TiDB 可以像 Prepare/Execute 语句一样支持执行计划缓存，详情请参考非 Prepare 语句执行计划缓存。

在当前版本中，当 Prepare 语句符合以下条件任何一条，查询或者计划不会被缓存：

- SELECT、UPDATE、INSERT、DELETE、Union、Intersect、Except 以外的 SQL 语句；
- 访问分区表、临时表或访问表中包含生成列的查询；
- 查询中包含非关联子查询，例如SELECT * FROM t1 WHERE t1.a > (SELECT 1 FROM t2 WHERE t2.b < 1)；
- 执行计划中带有 PhysicalApply 算子的关联子查询，例如 SELECT * FROM t1 WHERE t1.a > (SELECT a FROM t2 WHERE t1.b > t2.b)；
- 包含 ignore_plan_cache 这一 Hint 的查询，例如 select /*+ ignore_plan_cache()*/ * from t；
- 包含除 ? 外其他变量（即系统变量或用户自定义变量）的查询，例如 select * from t where a>? and b>@x；
- 查询包含无法被缓存函数。目前不能被缓存的函数有：database()、current_user、current_role、user、connection_id、last_insert_id、row_count、version、like；
- LIMIT 后面带有变量（例如 LIMIT ? 或 LIMIT 10, ?）且变量值大于 10000 的执行计划不缓存；
- ? 直接在 Order By 后的查询，如 Order By ?，此时 ? 表示根据 Order By 后第几列排序，排序列不同的查询使用同一个计划可能导致错误结果，故不缓存；如果是普通表达式，如 Order By a+? 则会缓存；
- ? 紧跟在 Group by 后的查询，如 Group By ?，此时 ? 表示根据 Group By 后第几列聚合，聚合列不同的查询使用同一个计划可能导致错误结果，故不缓存；如果是普通表达式，如 Group By a+? 则会缓存；
- ? 出现在窗口函数 Window Frame 定义中的查询，如 (partition by year order by sale rows ? preceding)；如果 ? 出现在窗口函数的其他位置，则会缓存；
- 用参数进行 int 和 string 比较的查询，如 c_int >= ? 或者 c_int in (?, ?)等，其中 ? 为字符串类型，如 set @x='123'；此时为了保证结果和 MySQL 兼容性，需要每次对参数进行调整，故不会缓存；
- 会访问 TiFlash 的计划不会被缓存；
- 大部分情况下计划中含有 TableDual 的计划将将不会被缓存，除非当前执行的 Prepare 语句不含参数，则对应的 TableDual 计划可以被缓存。

LRU 链表是设计成 session 级别的缓存，因为 Prepare/Execute 不能跨 session 执行。LRU 链表的每个元素是一个 key-value 对，value 是执行计划，key 由如下几部分组成：

- 执行 Execute 时所在数据库的名字；
- Prepare 语句的标识符，即紧跟在 PREPARE 关键字后的名字；
- 当前的 schema 版本，每条执行成功的 DDL 语句会修改 schema 版本；
- 执行 Execute 时的 SQL Mode；
- 当前设置的时区，即系统变量 time_zone 的值；
- 系统变量 sql_select_limit 的值；

key 中任何一项变动（如切换数据库、重命名 Prepare 语句、执行 DDL、修改 SQL Mode/time_zone 的值）、或 LRU 淘汰机制触发都会导致 Execute 时无法命中执行计划缓存。

成功从缓存中获取到执行计划后，TiDB 会先检查执行计划是否依然合法，如果当前 Execute 在显式事务里执行，并且引用的表在事务前序语句中被修改，而缓存的执行计划对该表访问不包含 UnionScan 算子，则它不能被执行。

在通过合法性检测后，会根据当前最新参数值，对执行计划的扫描范围做相应调整，再用它执行获取数据。

关于执行计划缓存和查询性能有几点值得注意：

- 不管计划是否已经被缓存，都会受到 SQL Binding 的影响。对于没有被缓存的计划，即在第一次执行 Execute 时，会受到已有 SQL Binding 的影响；而对于已经缓存的计划，如果有新的 SQL Binding 被创建产生，则原有已经被缓存的计划会失效。
- 已经被缓存的计划不会受到统计信息更新、优化规则和表达式下推黑名单更新的影响，仍然会使用已经保存在缓存中的计划。
- 重启TiDB 实例时（如不停机滚动升级TiDB 集群），Prepare 信息会丢失，此时执行 execute stmt ... 可能会遇到 Prepared Statement not found 的错误，此时需要再执行一次 prepare stmt ...。
- 考虑到不同 Execute 的参数会不同，执行计划缓存为了保证适配性会禁止一些和具体参数值密切相关的激进查询优化手段，导致对特定的一些参数值，查询计划可能不是最优。比如查询的过滤条件为 where a > ? and a < ?，第一次 Execute 时参数分别为 2 和 1，考虑到这两个参数下次执行时可能会是 1 和 2，优化器不会生成对当前参数最优的 TableDual 执行计划。
- 如果不考虑缓存失效和淘汰，一份执行计划缓存会对应各种不同的参数取值，理论上也会导致某些取值下执行计划非最优。比如查询过滤条件为 where a < ?，假如第一次执行 Execute 时用的参数值为 1，此时优化器生成最优的 IndexScan 执行计划放入缓存，在后续执行 Exeucte 时参数变为 10000，此时 TableScan 可能才是更优执行计划，但由于执行计划缓存，执行时还是会使用先前生成的 IndexScan。因此执行计划缓存更适用于查询较为简单（查询编译耗时占比较高）且执行计划较为固定的业务场景。

自 v6.1.0 起，执行计划缓存功能默认打开，可以通过变量tidb_enable_prepared_plan_cache 启用或关闭这项功能。

> **注意：**
>
> 执行计划缓存功能仅针对 Prepare/Execute 请求，对普通查询无效。

在开启了执行计划缓存功能后，可以通过 SESSION 级别的系统变量last_plan_from_cache 查看上一条 Execute 语句是否使用了缓存的执行计划，例如：

```
MySQL [test]> create table t(a int);
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> prepare stmt from 'select * from t where a = ?';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> set @a = 1;
Query OK, 0 rows affected (0.00 sec)

-- 第一次 execute 生成执行计划放入缓存
MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)
```

```
MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)

-- 第二次 execute 命中缓存
MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 1                      |
+------------------------+
1 row in set (0.00 sec)
```

如果发现某一组 Prepare/Execute 由于执行计划缓存导致了非预期行为，可以通过 SQL Hint ignore_plan_cache ’→ () 让该组语句不使用缓存。还是用上述的 stmt 为例：

```
MySQL [test]> prepare stmt from 'select /*+ ignore_plan_cache() */ * from t where a = ?';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> set @a = 1;
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)

MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
```

```
+-------------------------+
| 0                       |
+-------------------------+
1 row in set (0.00 sec)
```

8.3.3.4.1 诊断 Prepared Plan Cache

对于无法进行缓存的查询或计划，可通过 SHOW WARNINGS 语句查看查询或计划是否被缓存。如果未被缓存，则可在结果中查看无法被缓存的原因。示例如下：

```
mysql> PREPARE st FROM 'SELECT * FROM t WHERE a > (SELECT MAX(a) FROM t)';  -- 该查询包含子查询，
    '→ 因此无法被缓存
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;  -- 查看查询计划无法被缓存的原因
+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
+---------+------+-----------------------------------------------+
| Warning | 1105 | skip plan-cache: sub-queries are un-cacheable |
+---------+------+-----------------------------------------------+
1 row in set (0.00 sec)

mysql> PREPARE st FROM 'SELECT * FROM t WHERE a<?';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @a='1';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE st USING @a;  -- 该优化中进行了非 INT 类型到 INT 类型的转换，
    '→ 产生的执行计划可能随着参数变化而存在风险，因此 TiDB 不缓存该计划
Empty set, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;

+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
+---------+------+-----------------------------------------------+
| Warning | 1105 | skip plan-cache: '1' may be converted to INT |
+---------+------+-----------------------------------------------+
1 row in set (0.00 sec)
```

8.3.3.4.2 Prepared Plan Cache 的内存管理

使用 Prepared Plan Cache 会有一定的内存开销，可以通过Grafana 中的 Plan Cache Memory Usage 监控查看每台 TiDB 实例上所有 SESSION 所缓存的计划占用的总内存。

> 注意：
>
> 考虑到Golang的内存回收机制以及部分未统计的内存结构，Grafana 中显示的内存与实际的堆内存使用量并不相等。经过实验验证存在约±20% 的误差。

对于每台TiDB实例上所缓存的执行计划总数量，可以通过Grafana 中的 Plan Cache Plan Num 监控查看。

Grafana 中 Plan Cache Memory Usage 和 Plan Cache Plan Num 监控如下图所示：

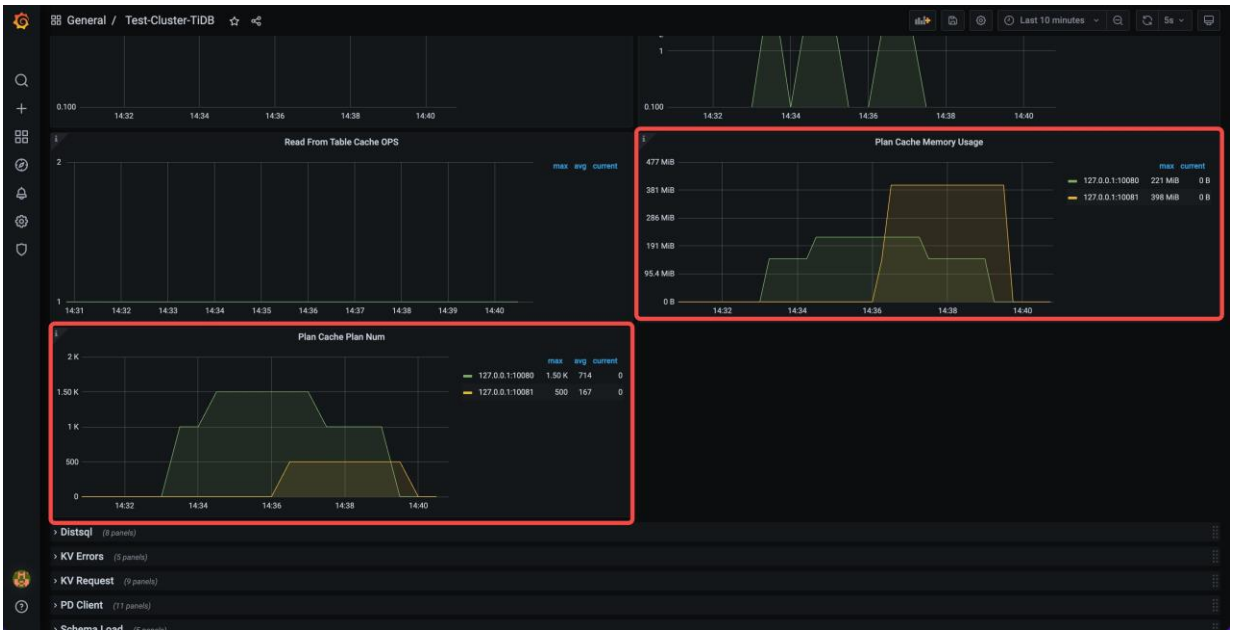


图 151: grafana_panels

从 v7.1.0 开始，你可以通过变量tidb_session_plan_cache_size 来设置每个 SESSION 最多缓存的计划数量。针对不同的环境，推荐的设置如下，你可以结合监控进行调整：

- TiDB Server 实例内存阈值<= 64 GiB 时，tidb_session_plan_cache_size = 50
- TiDB Server 实例内存阈值> 64 GiB 时，tidb_session_plan_cache_size = 100

从 v7.1.0 开始，你可以通过变量tidb_plan_cache_max_plan_size 来设置可以缓存的计划的最大大小，默认为 2 MB。超过该值的执行计划将不会被缓存到 Plan Cache 中。

当 TiDB Server 的内存余量小于一定阈值时，会触发 Plan Cache 的内存保护机制，此时会对一些缓存的计划进行逐出。

目前该阈值由变量 tidb_prepared_plan_cache_memory_guard_ratio 控制，默认为 0.1，即 10%，也就是当剩余内存不足 10%（使用内存超过 90%）时，会触发此机制。

由于内存限制，Plan Cache 可能出现 Cache Miss 的情况，可以通过Grafana 中的 Plan Cache Miss OPS 监控查看。

8.3.3.4.3 手动清空计划缓存

通过执行 `ADMIN FLUSH [SESSION | INSTANCE] PLAN_CACHE` 语句，你可以手动清空计划缓存。

该语句中的作用域 `[SESSION | INSTANCE]` 用于指定需要清空的缓存级别，可以为 `SESSION` 或 `INSTANCE`。如果不指定作用域，该语句默认清空 `SESSION` 级别的缓存。

下面是一个清空计划缓存的例子：

```
MySQL [test]> create table t (a int);
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> prepare stmt from 'select * from t';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> execute stmt;
Empty set (0.00 sec)

MySQL [test]> execute stmt;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache; -- 选择计划缓存
+                        +
| @@last_plan_from_cache |
+------------------------+
+                        +
|                      1 |
+------------------------+
1 row in set (0.00 sec)
 -------------------------

MySQL [test]> admin flush session plan_cache; -- 清空当前 session 的计划缓存
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> execute stmt;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache; -- 由于缓存被清空，此时无法再次选中
+                        +
| @@last_plan_from_cache |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.00 sec)

 -------------------------
```

TiDB 暂不支持清空 `GLOBAL` 级别的计划缓存，即不支持一次性清空整个集群的计划缓存，使用时会报错：

```
MySQL [test]> admin flush global plan_cache;
ERROR 1105 (HY000): Do not support the 'admin flush global scope.'
```

8.3.3.4.4 忽略 COM_STMT_CLOSE 指令和 DEALLOCATE PREPARE 语句

为了减少每次执行 SQL 语句的语法分析，Prepared Statement 推荐的使用方式是，prepare 一次，然后 execute 多次，最后deallocate prepare。例如：

```
MySQL [test]> prepare stmt from '...'; -- prepare 一次
MySQL [test]> execute stmt using ...; -- execute 一次
MySQL [test]> ...
MySQL [test]> execute stmt using ...;  -- execute 多次
MySQL [test]> deallocate prepare stmt; -- 使用完成后释放
```

如果你习惯于在每次 execute 后都立即执行deallocate prepare，如：

```
MySQL [test]> prepare stmt from '...'; -- 第一次 prepare
MySQL [test]> execute stmt using ...;
MySQL [test]> deallocate prepare stmt; -- 一次使用后立即释放
MySQL [test]> prepare stmt from '...'; -- 第二次 prepare
MySQL [test]> execute stmt using ...;
MySQL [test]> deallocate prepare stmt; -- 再次释放
```

这样的使用方式会让第一次执行得到的计划被立即清理，不能在第二次被复用。

为了兼容这样的使用方式，从 v6.0 起，TiDB 支持tidb_ignore_prepared_cache_close_stmt 变量。打开该变量后，TiDB 会忽略关闭 Prepare Statement 的信号，解决上述问题，如：

```
mysql> set @@tidb_ignore_prepared_cache_close_stmt=1;  -- 打开开关
Query OK, 0 rows affected (0.00 sec)

mysql> prepare stmt from 'select * from t'; -- 第一次 prepare
Query OK, 0 rows affected (0.00 sec)


mysql> execute stmt;                        -- 第一次 execute
Empty set (0.00 sec)

mysql> deallocate prepare stmt;             -- 第一次 execute 后立即释放
Query OK, 0 rows affected (0.00 sec)

mysql> prepare stmt from 'select * from t'; -- 第二次 prepare
Query OK, 0 rows affected (0.00 sec)

mysql> execute stmt;                        -- 第二次 execute
Empty set (0.00 sec)

mysql> select @@last_plan_from_cache;       -- 因为开关打开，第二次依旧能复用上一次的计划
+------------------------+
| @@last_plan_from_cache |
+------------------------+
|                      1 |
```

```
+-------------------------+
1 row in set (0.00 sec)
```

监控

在 Grafana 面板的 TiDB 页面，Executor 部分包含“Queries Using Plan Cache OPS” 和 “Plan Cache Miss OPS” 两个图表，用以检查 TiDB 和应用是否正确配置，以便 SQL 执行计划缓存能正常工作。TiDB 页面的 Server 部分还提供了“Prepared Statement Count” 图表，如果应用使用了预处理语句，这个图表会显示非零值。通过数值变化，可以判断 SQL 执行计划缓存是否正常工作。



图 152: sql_plan_cache

8.3.3.5 非 Prepare 语句执行计划缓存

> **警告：**
>
> 非 Prepare 语句执行计划缓存(Non-Prepared Plan Cache) 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。

对于某些非 PREPARE 语句，TiDB 可以像Prepare/Execute 语句一样支持执行计划缓存。这可以让这些语句跳过优化器阶段，以提升性能。

8.3.3.5.1 原理

Non-Prepared Plan Cache 为会话级别，并且与Prepared Plan Cache 共用一个缓存。Non-Prepared Plan Cache 功能的基本原理如下：

1. 开启 Non-Prepared Plan Cache 后，TiDB 首先根据AST（抽象语法树）对查询进行参数化。例如，将 SELECT * FROM t WHERE b < 10 AND a = 1 参数化为 SELECT * FROM t WHERE b < ? and a = ?。
2. 然后，使用参数化后的查询在 Plan Cache 中查找。
3. 如果能找到可以直接复用的计划，则直接使用，并跳过整个优化过程。
4. 否则，继续进行查询优化，并在最后将生成的计划放回到缓存中，以便下次复用。

8.3.3.5.2 使用方法

目前，你可以通过tidb_enable_non_prepared_plan_cache 开启或关闭Non-Prepared Plan Cache。同时，你还可以通过tidb_session_plan_cache_size 来控制Plan Cache 的大小。当缓存的计划数超过tidb_session_plan_cache_size 时，TiDB 会使用 LRU (Least Recently Used) 策略进行逐出。

从 v7.1.0 开始，你可以通过变量tidb_plan_cache_max_plan_size 来设置可以缓存的计划的最大大小，默认为 2 MB。超过该值的执行计划将不会被缓存到 Plan Cache 中。

> **注意：**
>
> tidb_session_plan_cache_size 定义的内存会被 Prepared 和 Non-Prepared Plan Cache 共享。如果集群已经开启 Prepared Plan Cache，那么开启 Non-Prepared Plan Cache 可能降低原先 Prepared Plan Cache 的命中率。

8.3.3.5.3 示例

下面是一个使用示例：

1. 创建用于测试的表t：

```
CREATE TABLE t (a INT, b INT, KEY(b));
```

2. 开启 Non-Prepared Plan Cache：

```
SET tidb_enable_non_prepared_plan_cache = ON;
```

3. 依次执行以下查询：

```
SELECT * FROM t WHERE b < 10 AND a = 1;
SELECT * FROM t WHERE b < 5 AND a = 2;
```

4. 查看第二个查询语句是否命中缓存：

```
SELECT @@last_plan_from_cache;
```

输出结果中 last_plan_from_cache 的值为 1，表示第二次执行的查询计划来自于缓存：

```
+------------------------+
| @@last_plan_from_cache |
+------------------------+
|                      1 |
+------------------------+
1 row in set (0.00 sec)
```

8.3.3.5.4 限制

缓存限制

TiDB 对参数化后形式相同的查询，只能缓存一个计划。例如，对于 `SELECT * FROM t WHERE a < 1` 和 `SELECT * FROM t WHERE a < 100000` 这两个查询语句，由于参数化后的形式相同，均为 `SELECT * FROM t WHERE a < ?`，因此它们会共用一个计划。

如果由此产生性能问题，可以使用 `ignore_plan_cache()` Hint 忽略计划缓存中的计划，让优化器每次重新为 SQL 生成执行计划。如果无法修改 SQL，可以通过创建 binding 来解决，例如 `CREATE BINDING FOR SELECT ... USING SELECT /*+ ignore_plan_cache()*/ ...`。

使用限制

由于上述风险以及执行计划缓存只在简单查询上有明显收益（如果查询较为复杂，查询本身执行时间较长，使用执行计划缓存收益不大），TiDB 目前对 Non-Prepared Plan Cache 的生效范围有严格的限制。具体限制如下：

- Prepared Plan Cache 不支持的查询或者计划，Non-Prepared Plan Cache 也不支持。
- 不支持包含 `Window` 或 `Having` 的查询。
- 不支持包含三张表及以上 `Join` 或子查询的查询。
- 不支持 `ORDER BY` 或者 `GROUP BY` 后直接带数字或者表达式的查询，如 `ORDER BY 1`、`GROUP BY a+1`。仅支持 `ORDER BY column_name` 和 `GROUP BY column_name`。
- 不支持过滤条件中包含 `JSON`、`ENUM`、`SET` 或 `BIT` 类型的列的查询，例如 `SELECT * FROM t WHERE json_col = '{}'`。
- 不支持过滤条件中出现 `NULL` 值的查询，例如 `SELECT * FROM t WHERE a is NULL`。
- 不支持参数化后参数个数超过 200 个的查询，例如 `SELECT * FROM t WHERE a in (1, 2, 3, ... 201)`。
- 不支持访问分区表、虚拟列、临时表、视图、或内存表的查询，例如 `SELECT * FROM INFORMATION_SCHEMA.COLUMNS`，其中 `COLUMNS` 为TiDB 内存表。
- 不支持带有 Hint 或有 Binding 的查询。
- 默认不支持 DML 语句或包含 `FOR UPDATE` 的查询语句。若要启用支持，你可以执行 `SET tidb_enable_non_prepared_plan_cache_for_dml = ON`。

开启此功能后，优化器会对查询进行快速判断，如果不满足 Non-Prepared Plan Cache 的支持条件，则会走正常的优化流程。

8.3.3.5.5 性能收益

在内部测试中，开启 Non-Prepared Plan Cache 功能在大多数 TP 场景下可以获得显著的性能收益。但是它也有代价，其自身也有一些额外的性能开销，包括判断查询是否支持、对查询进行参数化等。如果此功能不支持负载中的大多数查询，开启此功能反而可能影响性能。

此时，你需要观察Grafana 监控中的 Queries Using Plan Cache OPS 面板中的 `non-prepared` 指标和 Plan Cache Miss OPS 面板中的 `non-prepared-unsupported` 指标。如果大多数查询都无法被支持，只有少部分查询能命中 Plan Cache，此时你可以关闭此功能。

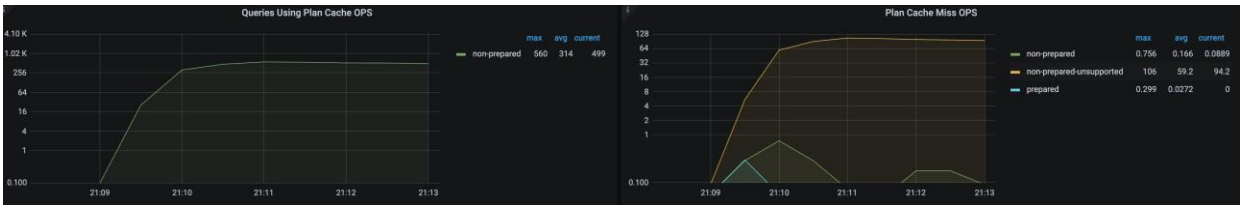


图 153: non-prepared-unsupported

8.3.3.5.6 诊断

开启 Non-Prepared Plan Cache 后，可以使用 `EXPLAIN FORMAT='plan_cache' SELECT ...` 语句验证查询是否能够命中缓存。对于无法命中缓存的查询，系统会通过 warning 的方式返回无法命中的原因。

需要注意的是，如果不加 `FORMAT='plan_cache'`，则 `EXPLAIN` 语句永远不会命中缓存。

执行下面 `EXPLAIN FORMAT='plan_cache'` 语句，查看查询是否能够命中：

```
EXPLAIN FORMAT='plan_cache' SELECT * FROM (SELECT a+1 FROM t1) t;
```

输出结果示例如下：

```
3 rows in set, 1 warning (0.00 sec)
```

通过 `SHOW warnings;` 查看无法命中缓存的查询信息：

```
SHOW warnings;
```

输出结果示例如下：

```
+--
    '-> --------+-------+                                                                                                +
    '->
| Level   | Code | Message
    '-> |
+--
    '-> --------+-------+                                                                                                +
    '->
| Warning | 1105 | skip non-prepared plan-cache: queries that have sub-queries are not supported
    '-> |
+--
    '-> --------+-------+                                                                                                +
    '->
1 row in set (0.00 sec)
```

可以看到在上述例子中，由于 Non-Prepared Plan Cache 不支持+ 操作，所以无法命中缓存。

8.3.3.5.7 监控

开启 Non-Prepared Plan Cache 后，可以在以下几个面板中查看缓存的内存使用情况、缓存中计划的个数、缓存命中的情况等信息。

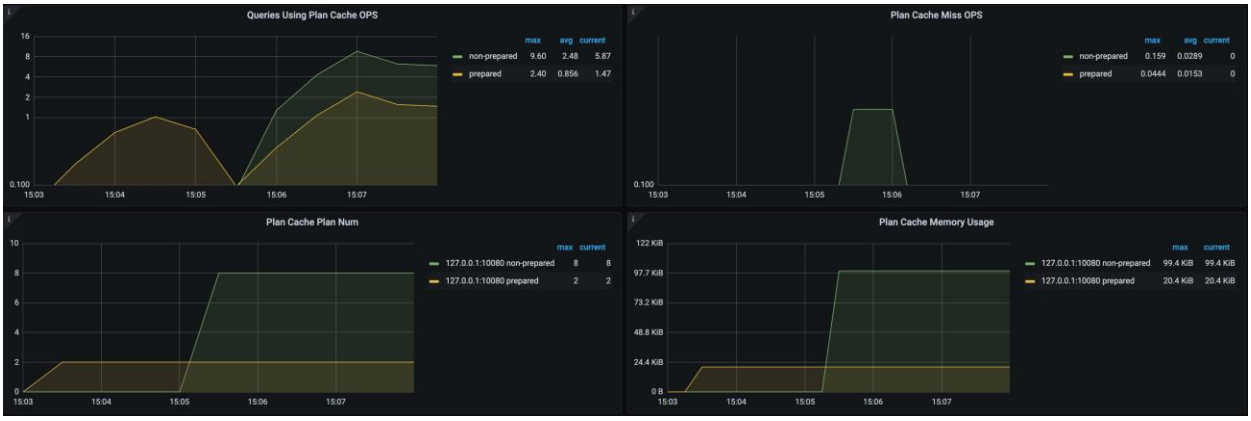


图 154: non-prepared-plan-cache

`statements_summary` 表和慢查询日志也会体现缓存的命中情况。下面是查看 `statements_summary` 表中缓存命中情况的例子：

1. 创建表`t`：

```
CREATE TABLE t (a int);
```

2. 打开 Non-Prepared Plan Cache 开关：

```
SET @@tidb_enable_non_prepared_plan_cache = ON;
```

3. 依次执行以下三个查询：

```
SELECT * FROM t WHERE a<1;
SELECT * FROM t WHERE a<2;
SELECT * FROM t WHERE a<3;
```

4. 查询 `statements_summary` 表查看查询命中缓存的情况：

```
SELECT digest_text, query_sample_text, exec_count, plan_in_cache, plan_cache_hits FROM
    -> INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE query_sample_text LIKE '%SELECT * FROM %
    -> ';
```

输出结果如下：

```
+--
    -> ------------------------------+-----------------------------------------+------------+------------
    ->
```

```
| digest_text                       | query_sample_text                          | exec_count |
    ↪ plan_in_cache | plan_cache_hits |
+--
    ↪ ----------------------------------+--------------------------------------------+------------+----------
    ↪
| SELECT * FROM `t` WHERE `a` < ? | SELECT * FROM t WHERE a<1                  |          3 |
    ↪             1 |                2 |
+--
    ↪ ----------------------------------+--------------------------------------------+------------+----------
    ↪
1 row in set (0.01 sec)
```

可以看到，查询执行了三次且命中缓存两次。

8.3.4 控制执行计划

8.3.4.1 控制执行计划

SQL 性能调优的前两个章节介绍了如何理解TiDB 的执行计划以及TiDB 如何生成一个执行计划。本章节将介绍当你确定了执行计划所存在的问题时，可以使用哪些手段来控制执行计划的生成。本章节主要包括以下三方面内容：

- Optimizer Hints中，我们会介绍如何使用 Hint 来指导TiDB 生成执行计划。
- 但是使用 Hint 会侵入性地更改 SQL，在一些场景下并不能简单的插入 Hint。在执行计划管理中，我们会介绍TiDB 如何使用另一种语法来非侵入地控制执行计划的生成，同时还会介绍后台自动对执行计划进行演进的手段。该手段可用来减轻诸如版本升级等原因造成的执行计划不稳定和集群性能下降的问题。
- 最后在优化规则及表达式下推的黑名单中，我们会介绍黑名单的使用。

除以上手段之外，执行计划还会被一部分系统变量所影响。通过在系统级或会话级对变量进行修改，能够控制执行计划的生成。自 v7.1.0 起，TiDB 引入了一个相对特殊的变量tidb_opt_fix_control。这个变量能够接受多个控制项，用来更细粒度地控制优化器的行为，避免集群升级后优化器行为变化导致的性能回退。

8.3.4.2 Optimizer Hints

TiDB 支持 Optimizer Hints 语法，它基于 MySQL 5.7 中介绍的类似comment 的语法，例如 `/*+ HINT_NAME(t1, t2)*/`。当 TiDB 优化器选择的不是最优查询计划时，建议使用 Optimizer Hints。

> **注意：**
>
> MySQL 命令行客户端在5.7.7 版本之前默认清除了Optimizer Hints。如果需要在这些早期版本的客户端中使用 `Hint` 语法，需要在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

8.3.4.2.1 语法

> **注意：**
>
> 如果需要提示优化器使用的表不在 USE DATABASE 所指定的数据库内，需要显式指定数据库名。例如：
>
> ```
> tidb> SELECT /*+ HASH_JOIN(t2, t) */ * FROM t, test2.t2;
> Empty set, 1 warning (0.00 sec)
>
> tidb> SHOW WARNINGS;
> +--
>     ↪        +       +
>     ↪ -------- ------- --------------------------------------------------------------------------
> | Level    | Code | Message
>     ↪
>     ↪ |
> +--
>     ↪        +       +
>     ↪ -------- ------- --------------------------------------------------------------------------
> | Warning | 1815 | There are no matching table names for (t2) in optimizer hint /*+
>     ↪  HASH_JOIN(t2, t) */ or /*+ TIDB_HJ(t2, t) */. Maybe you can use the table
>     ↪ alias name |
> +--
>     ↪        +       +
>     ↪ -------- ------- --------------------------------------------------------------------------
> 1 row in set (0.00 sec)
>
> tidb> SELECT /*+ HASH_JOIN(test2.t2, t) */ * FROM t, test2.t2;
> Empty set (0.00 sec)
>
> tidb> SELECT /*+ READ_FROM_STORAGE(TIFLASH[test1.t1,test2.t2]) */ t1.a FROM test1.t
>     ↪  t1, test2.t t2 WHERE t1.a = t2.a;
> Empty set (0.00 sec)
> ```
>
> 本文档中后续示例演示部分，皆是同一个数据库范围内的表。如果你使用的表不在同一个数据库内，请参照指示显式指定数据库名。

Optimizer Hints 不区分大小写，通过`/*+ ... */`注释的形式跟在SELECT、UPDATE 或 DELETE 关键字的后面。INSERT 关键字后不支持 Optimizer Hints。

多个不同的 Hint 之间需用逗号隔开，例如：

```
SELECT /*+ USE_INDEX(t1, idx1), HASH_AGG(), HASH_JOIN(t1) */ count(*) FROM t t1, t t2 WHERE t1.a
    ↪ = t2.b;
```

可以通过 Explain/Explain Analyze 语句的输出，来查看 Optimizer Hints 对查询执行计划的影响。

如果 Optimizer Hints 包含语法错误或不完整，查询语句不会报错，而是按照没有 Optimizer Hints 的情况执行。如果 Hint 不适用于当前语句，TiDB 会返回Warning，用户可以在查询结束后通过 Show Warnings 命令查看具体信息。

> 注意：
>
> 如果注释不是跟在指定的关键字后，会被当作是普通的 MySQL comment，注释不会生效，且不会上报 warning。

TiDB 目前支持的 Optimizer Hints 根据生效范围的不同可以划分为两类：第一类是在查询块范围生效的 Hint，例如/*+ HASH_AGG()*/；第二类是在整个查询范围生效的 Hint，例如/*+ MEMORY_QUOTA(1024 MB)*/。

每条语句中每一个查询和子查询都对应着一个不同的查询块，每个查询块有自己对应的名字。以下面这条语句为例：

```
SELECT * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

该查询语句有 3 个查询块，最外面一层 SELECT 所在的查询块的名字为 sel_1，两个 SELECT 子查询的名字依次为 sel_2 和 sel_3。其中数字序号根据 SELECT 出现的位置从左到右计数。如果分别用 DELETE 和 UPDATE 查询替代第一个 SELECT 查询，则对应的查询块名字分别为 del_1 和 upd_1。

8.3.4.2.2 查询块范围生效的 Hint

这类 Hint 可以跟在查询语句中任意 SELECT、UPDATE 或 DELETE 关键字的后面。通过在 Hint 中使用查询块名字可以控制 Hint 的生效范围，以及准确标识查询中的每一个表（有可能表的名字或者别名相同），方便明确 Hint 的参数指向。若不显式地在 Hint 中指定查询块，Hint 默认作用于当前查询块。以如下查询为例：

```
SELECT /*+ HASH_JOIN(@sel_1 t1@sel_1, t3) */ * FROM (SELECT t1.a, t1.b FROM t t1, t t2 WHERE t1.a
    ↪  = t2.a) t1, t t3 WHERE t1.b = t3.b;
```

该 Hint 在 sel_1 这个查询块中生效，参数分别为 sel_1 中的 t1 表（sel_2 中也有一个 t1 表）和 t3 表。

如上例所述，在 Hint 中使用查询块名字的方式有两种：第一种是作为 Hint 的第一个参数，与其他参数用空格隔开。除 QB_NAME 外，本节所列的所有 Hint 除自身明确列出的参数外都有一个隐藏的可选参数 @QB_NAME，通过使用这个参数可以指定该 Hint 的生效范围；第二种在 Hint 中使用查询块名字的方式是在参数中的某一个表名后面加 @QB_NAME，用以明确指出该参数是哪个查询块中的表。

> 注意：
>
> Hint 声明的位置必须在指定生效的查询块之中或之前，不能是在之后的查询块中，否则无法生效。

QB_NAME

当查询语句是包含多层嵌套子查询的复杂语句时，识别某个查询块的序号和名字很可能会出错，Hint QB_NAME 可以方便我们使用查询块。QB_NAME 是 Query Block Name 的缩写，用于为某个查询块指定新的名字，同时查询块原本默认的名字依然有效。例如：

```
SELECT /*+ QB_NAME(QB1) */ * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

这条 Hint 将最外层 SELECT 查询块的命名为 QB1，此时 QB1 和默认名称 sel_1 对于这个查询块来说都是有效的。

> 注意：
>
> 上述例子中，如果指定的 QB_NAME 为 sel_2，并且不给原本 sel_2 对应的第二个查询块指定新的 QB_NAME，则第二个查询块的默认名字 sel_2 会失效。

MERGE_JOIN(t1_name [, tl_name …])

MERGE_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Sort Merge Join 算法。这个算法通常会占用更少的内存，但执行时间会更久。当数据量太大，或系统内存不足时，建议尝试使用。例如：

```
SELECT /*+ MERGE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> MERGE_JOIN 的别名是 TIDB_SMJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，但推荐使用 MERGE_JOIN。

INL_JOIN(t1_name [, tl_name …])

INL_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用Index Nested Loop Join 算法。这个算法可能会在某些场景更快，消耗更少系统资源，有的场景会更慢，消耗更多系统资源。对于外表经过WHERE 条件过滤后结果集较小（小于 1 万行）的场景，可以尝试使用。例如：

```
SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

INL_JOIN() 中的参数是建立查询计划时内表的候选表，比如 INL_JOIN(t1) 只会考虑使用t1 作为内表构建查询计划。表如果指定了别名，就只能使用表的别名作为 INL_JOIN() 的参数；如果没有指定别名，则用表的本名作为其参数。比如在 SELECT /*+ INL_JOIN(t1)*/ * FROM t t1, t t2 WHERE t1.a = t2.b; 中，INL_JOIN() 的参数只能使用t 的别名t1 或 t2，不能用t。

> 注意：
>
> INL_JOIN 的别名是 TIDB_INLJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，但推荐使用 INL_JOIN。

INL_HASH_JOIN

`INL_HASH_JOIN(t1_name [, tl_name])` 提示优化器使用Index Nested Loop Hash Join 算法。该算法与Index Nested Loop Join 使用条件完全一样，两者的区别是 `INL_JOIN` 会在连接的内表上建哈希表，而 `INL_HASH_JOIN` 会在连接的外表上建哈希表，后者对于内存的使用是有固定上限的，而前者使用的内存使用取决于内表匹配到的行数。

HASH_JOIN(t1_name [, tl_name …])

`HASH_JOIN(t1_name [, tl_name ...])` 提示优化器对指定表使用 Hash Join 算法。这个算法多线程并发执行，执行速度较快，但会消耗较多内存。例如：

```
SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> HASH_JOIN 的别名是 TIDB_HJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，推荐使用 HASH_JOIN。

HASH_JOIN_BUILD(t1_name [, tl_name …])

`HASH_JOIN_BUILD(t1_name [, tl_name ...])` 提示优化器对指定表使用Hash Join 算法，同时将指定表作为Hash Join 算法的 Build 端，即用指定表来构建哈希表。例如：

```
SELECT /*+ HASH_JOIN_BUILD(t1) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

HASH_JOIN_PROBE(t1_name [, tl_name …])

`HASH_JOIN_PROBE(t1_name [, tl_name ...])` 提示优化器对指定表使用Hash Join 算法，同时将指定表作为Hash Join 算法的探测（Probe）端，即用指定表作为探测端来执行 Hash Join 算法。例如：

```
SELECT /*+ HASH_JOIN_PROBE(t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

SEMI_JOIN_REWRITE()

`SEMI_JOIN_REWRITE()` 提示优化器将查询语句中的半连接(Semi Join) 改写为普通的内连接。目前该 Hint 只作用于 `EXISTS` 子查询。

如果不使用该 Hint 进行改写，Semi Join 在选择 Hash Join 的执行方式时，只能够使用子查询构建哈希表，因此在子查询比外查询结果集大时，执行速度可能会不及预期。Semi Join 在选择Index Join 的执行方式时，只能够使用外查询作为驱动表，因此在子查询比外查询结果集小时，执行速度可能会不及预期。

在使用了 `SEMI_JOIN_REWRITE()` 进行改写后，优化器便可以扩大选择范围，选择更好的执行方式。

```
-- 不使用 SEMI_JOIN_REWRITE() 进行改写
EXPLAIN SELECT * FROM t WHERE EXISTS (SELECT 1 FROM t1 WHERE t1.a = t.a);
```

```
+--
    '→                          +          +          +                          +
    '→
```

```
| id                          | estRows | task      | access object          | operator info
    ↪                                    |
+--
    ↪ ---------------------------+---------+-----------+------------------------+
    ↪
| MergeJoin_9                 | 7992.00 | root      |                        | semi join, left
    ↪ key:test.t.a, right key:test.t1.a |
| ├─IndexReader_25(Build)     | 9990.00 | root      |                        | index:
    ↪ IndexFullScan_24                               |
| │ └─IndexFullScan_24        | 9990.00 | cop[tikv] | table:t1, index:idx(a) | keep order:true,
    ↪ stats:pseudo                          |
| └─IndexReader_23(Probe)     | 9990.00 | root      |                        | index:
    ↪ IndexFullScan_22                               |
|   └─IndexFullScan_22        | 9990.00 | cop[tikv] | table:t, index:idx(a)  | keep order:true,
    ↪ stats:pseudo                          |
+--
    ↪                              +         +           +                        +
    ↪
```

```
-- 使用 SEMI_JOIN_REWRITE() 进行改写
EXPLAIN SELECT * FROM t WHERE EXISTS (SELECT /*+ SEMI_JOIN_REWRITE() */ 1 FROM t1 WHERE t1.a = t.
    ↪ a);
```

```
+--
    ↪                              +         +           +                        +
    ↪
| id                          | estRows | task      | access object          | operator info
    ↪
    ↪ |
+--
    ↪ ---------------------------+---------+-----------+------------------------+
    ↪
| IndexJoin_16                | 1.25    | root      |                        | inner join, inner
    ↪ :IndexReader_15, outer key:test.t1.a, inner key:test.t.a, equal cond:eq(test.t1.a, test.t
    ↪ .a) |
| ├─StreamAgg_39(Build)       | 1.00    | root      |                        | group by:test.t1
    ↪ .a, funcs:firstrow(test.t1.a)->test.t1.a
    ↪                                                    |
| │ └─IndexReader_34          | 1.00    | root      |                        | index:
    ↪ IndexFullScan_33
    ↪                                                                              |
| │   └─IndexFullScan_33      | 1.00    | cop[tikv] | table:t1, index:idx(a) | keep order:true
    ↪
    ↪ |
| └─IndexReader_15(Probe)     | 1.25    | root      |                        | index:
```

```
    '→ Selection_14
    '→
    '→ |
|   └─Selection_14              | 1.25    | cop[tikv] |                          | not(isnull(test.
    '→ t.a))
    '→                                                                                 |
|     └─IndexRangeScan_13       | 1.25    | cop[tikv] | table:t, index:idx(a)    | range: decided
    '→ by [eq(test.t.a, test.t1.a)], keep order:false, stats:pseudo
    '→                                    |
+--
    '→                              +         +           +                          +
    '→
```

在上述例子中可以看到，在使用了 Hint 之后，TiDB 可以选择由表 `t1` 作为驱动表的IndexJoin 的执行方式。

SHUFFLE_JOIN(t1_name [, tl_name ⋯])

`SHUFFLE_JOIN(t1_name [, tl_name ...])` 提示优化器对指定表使用 Shuffle Join 算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ SHUFFLE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> - 使用该 Hint 前，需要保证当前TiDB 集群能够支持在查询中使用TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。
> - 该 Hint 能与HASH_JOIN_BUILD Hint 和HASH_JOIN_PROBE Hint 组合使用，达到控制 Shuffle Join 算法的 Build 端和 Probe 端的作用。

BROADCAST_JOIN(t1_name [, tl_name ⋯])

`BROADCAST_JOIN(t1_name [, tl_name ...])` 提示优化器对指定表使用 Broadcast Join 算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ BROADCAST_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> - 使用该 Hint 前，需要保证当前TiDB 集群能够支持在查询中使用TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。
> - 该 Hint 能与HASH_JOIN_BUILD Hint 和HASH_JOIN_PROBE Hint 组合使用，达到控制 Broadcast Join 算法的 Build 端和 Probe 端的作用。

NO_DECORRELATE()

NO_DECORRELATE() 提示优化器不要尝试解除指定查询块中对应子查询的关联。该 Hint 适用于包含关联列的 EXISTS、IN、ANY、ALL、SOME 和标量子查询，即关联子查询。

将该 Hint 写在一个查询块中后，对于该子查询和其外部查询块之间的关联列，优化器将不再尝试解除关联，而是始终使用Apply 算子来执行查询。

默认情况下，TiDB 会尝试对关联子查询解除关联，以达到更高的执行效率。但是在一部分场景下，解除关联反而会降低执行效率。这种情况下，可以使用该 Hint 来人工提示优化器不要进行解除关联操作。例如：

```
create table t1(a int, b int);
create table t2(a int, b int, index idx(b));
```

```
-- 不使用 NO_DECORRELATE()
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    '→                                  +          +          +               +
    '→
| id                                | estRows  | task      | access object | operator info
    '→
    '→ |
+--
    '→                                  +          +          +               +
    '→ ----------------------------------- ---------- ---------- ----------------
| HashJoin_11                       | 9990.00  | root      |               | inner join, equal:[eq
    '→ (test.t1.b, test.t2.b)], other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
| ├─HashAgg_23(Build)               | 7992.00  | root      |               | group by:test.t2.b,
    '→ funcs:sum(Column#8)->Column#7, funcs:firstrow(test.t2.b)->test.t2.b
    '→ |
| │ └─TableReader_24                | 7992.00  | root      |               | data:HashAgg_16
    '→
    '→ |
| │   └─HashAgg_16                  | 7992.00  | cop[tikv] |               | group by:test.t2.b,
    '→ funcs:sum(test.t2.a)->Column#8
    '→ |
| │     └─Selection_22              | 9990.00  | cop[tikv] |               | not(isnull(test.t2.b
    '→ ))
    '→ |
| │       └─TableFullScan_21        | 10000.00 | cop[tikv] | table:t2      | keep order:false,
    '→ stats:pseudo
    '→                                                                              |
| └─TableReader_15(Probe)           | 9990.00  | root      |               | data:Selection_14
    '→
    '→ |
|   └─Selection_14                  | 9990.00  | cop[tikv] |               | not(isnull(test.t1.b
    '→ ))
```

```
    ↪ |
|     └─TableFullScan_13           | 10000.00 | cop[tikv] | table:t1       | keep order:false,
    ↪ stats:pseudo
    ↪                                                                                  |
+--
    ↪                                  +          +           +               +
    ↪
```

从以上执行计划中可以发现，优化器自动解除了关联。解除关联之后的执行计划不包含Apply 算子，取而代之的是子查询和外部查询块之间的Join 运算，而原本的带有关联列的过滤条件 `t2.b = t1.b` 也变成了一个普通的join 条件。

```
-- 使用 NO_DECORRELATE()
explain select * from t1 where t1.a < (select /*+ NO_DECORRELATE() */ sum(t2.a) from t2 where t2.
    ↪ b = t1.b);
```

```
+--
    ↪ ----------------------------------------+----------+----------+-------------------------+
    ↪
| id                                    | estRows    | task       | access object           |
    ↪ operator info                                                                        |
+--
    ↪ ----------------------------------------+------------+------------+-------------------------+
    ↪
| Projection_10                         | 10000.00   | root       |                         |
    ↪ test.t1.a, test.t1.b                                                                 |
|  └─Apply_12                            | 10000.00   | root       |                         |
     ↪ CARTESIAN inner join, other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
|    ├─TableReader_14(Build)             | 10000.00   | root       |                         |
    ↪ data:TableFullScan_13                                                                |
|    │ └─TableFullScan_13                | 10000.00   | cop[tikv] | table:t1                |
    ↪ keep order:false, stats:pseudo                                                       |
|    └─MaxOneRow_15(Probe)               | 10000.00   | root       |                         |
    ↪                                                                                     |
|      └─StreamAgg_20                    | 10000.00   | root       |                         |
    ↪ funcs:sum(Column#14)->Column#7                                                       |
|        └─Projection_45                 | 100000.00 | root       |                         |
    ↪ cast(test.t2.a, decimal(10,0) BINARY)->Column#14                                     |
|          └─IndexLookUp_44              | 100000.00 | root       |                         |
    ↪                                                                                     |
|            ├─IndexRangeScan_42(Build)  | 100000.00 | cop[tikv] | table:t2, index:idx(b) |
    ↪ range: decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo         |
|            └─TableRowIDScan_43(Probe)  | 100000.00 | cop[tikv] | table:t2                |
    ↪ keep order:false, stats:pseudo                                                       |
+--
    ↪                                       +          +           +                      +
```

```
    ↪
```

从以上执行计划中可以发现，优化器没有解除关联。执行计划中包含 Apply 算子，而带有关联列的条件 `t2.b = t1.b` 仍然是访问 `t2` 表时的过滤条件。

HASH_AGG()

`HASH_AGG()` 提示优化器对指定查询块中所有聚合函数使用 Hash Aggregation 算法。这个算法多线程并发执行，执行速度较快，但会消耗较多内存。例如：

```
SELECT /*+ HASH_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

STREAM_AGG()

`STREAM_AGG()` 提示优化器对指定查询块中所有聚合函数使用 Stream Aggregation 算法。这个算法通常会占用更少的内存，但执行时间会更久。数据量太大，或系统内存不足时，建议尝试使用。例如：

```
SELECT /*+ STREAM_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

MPP_1PHASE_AGG()

`MPP_1PHASE_AGG()` 提示优化器对指定查询块中所有聚合函数使用一阶段聚合算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ MPP_1PHASE_AGG() */ COUNT(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

> 注意：
>
> 使用该 Hint 前，需要保证当前TiDB 集群能够支持在查询中使用TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。

MPP_2PHASE_AGG()

`MPP_2PHASE_AGG()` 提示优化器对指定查询块中所有聚合函数使用二阶段聚合算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ MPP_2PHASE_AGG() */ COUNT(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

> 注意：
>
> 使用该 Hint 前，需要保证当前TiDB 集群能够支持在查询中使用TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。

USE_INDEX(t1_name, idx1_name [, idx2_name …])

`USE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引。

下面例子的效果等价于 `SELECT * FROM t t1 use index(idx1, idx2);`：

```
SELECT /*+ USE_INDEX(t1, idx1, idx2) */ * FROM t1;
```

> **注意：**
>
> 当该 Hint 中只指定表名，不指定索引名时，表示不考虑使用任何索引，而是选择全表扫。

FORCE_INDEX(t1_name, idx1_name [, idx2_name …])

`FORCE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引。

`FORCE_INDEX(t1_name, idx1_name [, idx2_name ...])` 的使用方法、作用和 `USE_INDEX(t1_name, idx1_name [, idx2_name ...])` 相同。

以下四个查询语句的效果相同：

```
SELECT /*+ USE_INDEX(t, idx1) */ * FROM t;
SELECT /*+ FORCE_INDEX(t, idx1) */ * FROM t;
SELECT * FROM t use index(idx1);
SELECT * FROM t force index(idx1);
```

IGNORE_INDEX(t1_name, idx1_name [, idx2_name …])

`IGNORE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表忽略给出的索引。

下面例子的效果等价于 `SELECT * FROM t t1 ignore index(idx1, idx2);`：

```
SELECT /*+ IGNORE_INDEX(t1, idx1, idx2) */ * FROM t t1;
```

ORDER_INDEX(t1_name, idx1_name [, idx2_name …])

`ORDER_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引，并且按顺序读取指定的索引。

> **警告：**
>
> 这个hint 有可能会导致 SQL 语句报错，建议先进行测试。如果测试时发生报错，请移除该 Hint。如果测试时运行正常，则可以继续使用。

此 hint 通常应用在下面这种场景中：

```
CREATE TABLE t(a INT, b INT, key(a), key(b));
EXPLAIN SELECT /*+ ORDER_INDEX(t, a) */ a FROM t ORDER BY a LIMIT 10;
```

```
+--
    '→ -------------------------+---------+-----------+---------------------+                                +
    '→
| id                          | estRows | task      | access object       | operator info
    '→                 |
+--
    '→ ---------------------------+---------+-----------+---------------------+                                +
    '→
| Limit_10                    | 10.00   | root      |                     | offset:0, count:10
    '→            |
| └─IndexReader_14            | 10.00   | root      |                     | index:Limit_13
    '→                |
|   └─Limit_13                | 10.00   | cop[tikv] |                     | offset:0, count:10
    '→            |
|     └─IndexFullScan_12      | 10.00   | cop[tikv] | table:t, index:a(a) | keep order:true,
    '→ stats:pseudo |
+--    ---------------------------+---------+-----------+---------------------+                                +
    '→                            +         +           +                     +
    '→
```

优化器对该查询会生成两类计划：~~`Limit + IndexScan(keep order: true)`~~ 和 ~~`TopN + IndexScan(keep order:`~~ `false)`，当使用了 ORDER_INDEX Hint，优化器会选择前一种按照顺序读取索引的计划。

> 注意：
>
> - 如果查询本身并不需要按顺序读取索引，即在不使用 Hint 的前提下，优化器在任何情况下都不会生成按顺序读取索引的计划。此时，如果指定了 ORDER_INDEX Hint，会出现报错 `Can't find a proper physical plan for this query`，此时应考虑移除对应的 ORDER_INDEX Hint。
> - 分区表上的索引无法支持按顺序读取， 所以不应该对分区表及其相关的索引使用 ORDER_INDEX Hint。

NO_ORDER_INDEX(t1_name, idx1_name [, idx2_name …])

`NO_ORDER_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引，并且不按顺序读取指定的索引。通常应用在下面这种场景中:

以下示例中查询语句的效果等价于 `SELECT * FROM t t1 use index(idx1, idx2);`：

```
CREATE TABLE t(a INT, b INT, key(a), key(b));
EXPLAIN SELECT /*+ NO_ORDER_INDEX(t, a) */ a FROM t ORDER BY a LIMIT 10;
```

```
+--
    '-> ------------------------+----------+-----------+---------------------+------------------------------+
    '->
| id                          | estRows  | task      | access object       | operator info
    '->                  |
+--
    '-> ------------------------+----------+-----------+---------------------+------------------------------+
    '->
| TopN_7                      | 10.00    | root      |                     | test.t.a, offset:0,
    '-> count:10    |
| └─IndexReader_14            | 10.00    | root      |                     | index:TopN_13
    '->                  |
|   └─TopN_13                 | 10.00    | cop[tikv] |                     | test.t.a, offset:0,
    '-> count:10    |
|     └─IndexFullScan_12      | 10000.00 | cop[tikv] | table:t, index:a(a) | keep order:false,
    '-> stats:pseudo |
+--    ------------------------- ---------- ----------- ---------------------                              +
    '->                           +          +           +                     +------------------------------
    '->
```

和 ~~ORDER_INDEX Hint~~ 的示例相同，优化器对该查询会生成两类计划：~~Limit + IndexScan(keep order: true)~~ 和 TopN + IndexScan(keep order: false)，当使用了 NO_ORDER_INDEX Hint，优化器会选择后一种不按照顺序读取索引的计划。

AGG_TO_COP()

AGG_TO_COP() 提示优化器将指定查询块中的聚合函数下推到coprocessor。如果优化器没有下推某些适合下推的聚合函数，建议尝试使用。例如：

```
SELECT /*+ AGG_TO_COP() */ sum(t1.a) FROM t t1;
```

LIMIT_TO_COP()

LIMIT_TO_COP() 提示优化器将指定查询块中的 Limit 和 TopN 算子下推到coprocessor。优化器没有下推 Limit 或者 TopN 算子时建议尝试使用该提示。例如：

```
SELECT /*+ LIMIT_TO_COP() */ * FROM t WHERE a = 1 AND b > 10 ORDER BY c LIMIT 1;
```

READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name …]], TIKV[t2_name [, tl_name …]])

READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name ...]], TIKV[t2_name [, tl_name ...]]) 提示优化器从指定的存储引擎来读取指定的表，目前支持的存储引擎参数有 TIKV 和 TIFLASH。如果为表指定了别名，就只能使用表的别名作为 READ_FROM_STORAGE() 的参数；如果没有指定别名，则用表的本名作为其参数。例如：

```
SELECT /*+ READ_FROM_STORAGE(TIFLASH[t1], TIKV[t2]) */ t1.a FROM t t1, t t2 WHERE t1.a = t2.a;
```

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name …])

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name ...]) 提示优化器通过索引合并的方式来访问指定的表。索引合并分为并集型和交集型两种类型，详情参见用 EXPLAIN 查看索引合并的 SQL 执行计划。

若显式地指定索引列表，优化器会尝试在索引列表中选取索引来构建索引合并。若不指定索引列表，优化器会尝试在所有可用的索引中选取索引来构建索引合并。

对于交集型索引合并，索引列表是必选参数。对于并集型索引合并，Hint 中的索引列表为可选参数。示例如下。

```
SELECT /*+ USE_INDEX_MERGE(t1, idx_a, idx_b, idx_c) */ * FROM t1 WHERE t1.a > 10 OR t1.b > 10;
```

当对同一张表有多个 USE_INDEX_MERGE Hint 时，优化器会从这些 Hint 指定的索引列表的并集中尝试选取索引。

> 注意：
>
> USE_INDEX_MERGE 的参数是索引名，而不是列名。对于主键索引，索引名为 primary。

LEADING(t1_name [, tl_name ⋯])

LEADING(t1_name [, tl_name ...]) 提示优化器在生成多表连接的执行计划时，按照hint 中表名出现的顺序来确定多表连接的顺序。例如：

```
SELECT /*+ LEADING(t1, t2) */ * FROM t1, t2, t3 WHERE t1.id = t2.id and t2.id = t3.id;
```

在以上多表连接查询语句中，LEADING() 中表出现的顺序决定了优化器将会先对表 t1 和 t2 进行连接，再将结果和表 t3 进行连接。该hint 比STRAIGHT_JOIN 更为通用。

LEADING hint 在以下情况下会失效：

- 指定了多个 LEADING hint
- LEADING hint 中指定的表名不存在
- LEADING hint 中指定了重复的表名
- 优化器无法按照 LEADING hint 指定的顺序进行表连接
- 已经存在 straight_join() hint
- 查询语句中包含 outer join 且同时指定了包含笛卡尔积的情况
- 和选择join 算法的hint（即 MERGE_JOIN、 INL_JOIN、 INL_HASH_JOIN、 HASH_JOIN ）同时使用时

当出现了上述失效的情况，会输出 warning 警告。

```
-- 指定了多个 LEADING hint

SELECT /*+ LEADING(t1, t2) LEADING(t3) */ * FROM t1, t2, t3 WHERE t1.id = t2.id and t2.id = t3.id
    ↪ ;

-- 通过执行 `show warnings` 了解具体产生冲突的原因

SHOW WARNINGS;
```

```
+--
    '-> --------+-------+----------------------------------------------------------------------------------------------------------------
    '->
| Level   | Code | Message
    '->
    '-> |
+--
    '-> --------+-------+----------------------------------------------------------------------------------------------------------------
    '->
| Warning | 1815 | We can only use one leading hint at most, when multiple leading hints are used
    '-> , all leading hints will be invalid |
+--
    '->        +       +
    '-> --------+-------+----------------------------------------------------------------------------------------------------------------
    '->
```

> **注意：**
>
> 如果查询语句中包含了 outer join，你只能在hint 中指定可以用于交换连接顺序的表。如果hint 中存在不能用于交换的表，则该hint 会失效。例如在 `SELECT * FROM t1 LEFT JOIN (t2 JOIN '-> t3 JOIN t4)ON t1.a = t2.a;` 中，如果想要控制 `t2`、`t3`、`t4` 表的连接顺序，那么在使用 `LEADING` hint 时，hint 中不能出现 `t1` 表。

MERGE()

在含有公共表表达式的查询中使用 `MERGE()` hint，可关闭对当前子查询的物化过程，并将内部查询的内联展开到外部查询。该hint 适用于非递归的公共表表达式查询，在某些场景下，使用该hint 会比默认分配一块临时空间的语句执行效率更高。例如将外部查询的条件下推或在嵌套的 CTE 查询中：

```
-- 使用 hint 将外部查询条件的谓词下推
WITH CTE AS (SELECT /*+ MERGE() */ * FROM tc WHERE tc.a < 60) SELECT * FROM CTE WHERE CTE.a <18;

-- 在嵌套 CTE 查询中使用该 hint 来指定将某个 CTE 内联展开到外部查询
WITH CTE1 AS (SELECT * FROM t1), CTE2 AS (WITH CTE3 AS (SELECT /*+ MERGE() */ * FROM t2), CTE4 AS
    '->  (SELECT * FROM t3) SELECT * FROM CTE3, CTE4) SELECT * FROM CTE1, CTE2;
```

> **注意：**
>
> `MERGE()` 只适用于简单的 CTE 查询，在以下情况下无法使用该hint：
>
> - 递归的 CTE 查询
> - 子查询中有无法进行内联展开的部分，例如聚合算子、窗口函数以及 DINSTINCT 等
>
> 当 CTE 引用次数过多时，查询性能可能低于默认的物化方式。

8.3.4.2.3 全局生效的 Hint

全局生效的 Hint 和视图有关，可以使查询中定义的 Hint 能够在视图内部生效。添加这类 Hint 需要两步：先用 QB_NAME Hint 为视图内的查询块命名，再以“视图名@查询块名”的方式加入实际需要的 Hint。

第 1 步：使用 QB_NAME Hint 重命名视图内的查询块

首先使用QB_NAME Hint 重命名视图内部的查询块。其中针对视图的 QB_NAME Hint 的概念与查询块范围生效的 QB_NAME Hint相同，只是在语法上进行了相应的拓展。从 QB_NAME(QB) 拓展为 QB_NAME(QB, 视图名@查询块名 [.视图名@查询块名 .视图名@查询块名 ...])。

> 注意：
>
> @查询块名 与后面紧跟的 .视图名@查询块名 部分之间需要有一个空格， 否则 .视图名@查询块名会被视作前面@查询块名的一部分。例如，QB_NAME(v2_1, v2@SEL_1 .@SEL_1) 不能写为 QB_NAME(v2_1, v2@SEL_1.@SEL_1)。

- 对于单个视图、不包含子查询的简单语句，下面以重命名视图v 的第一个查询块为例：

```
SELECT /* 注释：当前查询块的名字为默认的 @SEL_1 */ * FROM v;
```

对于视图v 来说，从查询语句开始的首个视图是 v@SEL_1。视图 v 的第一个查询块可以声明为 QB_NAME(v_1, v@SEL_1 .@SEL_1)，也可以省略 @SEL_1 简写成 QB_NAME(v_1, v)：

```
CREATE VIEW v AS SELECT /* 注释：当前查询块的名字为默认的 @SEL_1 */ * FROM t;

-- 使用全局生效的 Hint
SELECT /*+ QB_NAME(v_1, v) USE_INDEX(t@v_1, idx) */ * FROM v;
```

- 对于嵌套视图和包含子查询的复杂语句，下面以重命名视图 v1、v2 的两个查询块为例：

```
SELECT /* 注释：当前查询块的名字为默认的 @SEL_1 */ * FROM v2 JOIN
    ( SELECT /* 注释：当前查询块的名字为默认的 @SEL_2 */ * FROM v2) vv;
```

对于第一个视图 v2 来说，从上面的语句开始的首个视图是 v2@SEL_1。对于第二个视图 v2 来说，首个视图表为 v2@SEL_2。下面的查询部分仅考虑第一个视图 v2。

视图 v2 的第一个查询块可以声明为 QB_NAME(v2_1, v2@SEL_1 .@SEL_1)，视图 v2 的第二个查询块可以声明为 QB_NAME(v2_2, v2@SEL_1 .@SEL_2)：

```
CREATE VIEW v2 AS
    SELECT * FROM t JOIN /* 注释：对于视图 v2 来说，当前查询块的名字为默认的 @SEL_1，
        因此当前查询块的视图列表是 v2@SEL_1 .@SEL_1 */
    (
        SELECT COUNT(*) FROM t1 JOIN v1 /* 注释：对于视图 v2 来说，当前查询块的名字为默认的
            @SEL_2，因此当前查询块的视图列表是 v2@SEL_1 .@SEL_2 */
    ) tt;
```

对于视图 v1 来说，从上面的语句开始的首个视图是 v2@SEL_1 .v1@SEL_2。视图 v1 的第一个查询块可以声明为 QB_NAME(v1_1, v2@SEL_1 .v1@SEL_2 .@SEL_1)，视图 v1 的第二个查询块可以声明为 QB_NAME(′→ v1_2, v2@SEL_1 .v1@SEL_2 .@SEL_2)：

```
CREATE VIEW v1 AS SELECT * FROM t JOIN /* 注释：对于视图 v1 来说，当前查询块的名字为默认的
   ′→ @SEL_1，因此当前查询块的视图列表是 v2@SEL_1 .v1@SEL_2 .@SEL_1 */
   (
        SELECT COUNT(*) FROM t1 JOIN t2 /* 注释：对于视图 v1 来说，当前查询块的名字为默认的
           ′→ @SEL_2，因此当前查询块的视图列表是 v2@SEL_1 .v1@SEL_2 .@SEL_2 */
   ) tt;
```

注意：

- 与视图相关的全局生效的 Hint 必须先定义了对应的 QB_NAME Hint 才能使用。
- 使用一个 Hint 来指定视图内的多个表名时，需要保证在同一个 Hint 中出现的表名处于同一个视图的同一个查询块中。
- 对于最外层的查询来说，在定义和视图相关的 QB_NAME Hint 时：
  - 对于 QB_NAME Hint 中视图列表序列的第一项，在不显式声明 @SEL_ 时，默认和定义 QB_NAME Hint 的查询块位置保持一致，即省略 @SEL_ 的查询 SELECT /*+ QB_NAME(qb1, v2)*/ * FROM v2 JOIN (SELECT /*+ QB_NAME(qb2, v2 ′→)*/ * FROM v2)vv; 相当于 SELECT /*+ QB_NAME(qb1, v2@SEL_1)*/ * FROM v2 ′→ JOIN (SELECT /*+ QB_NAME(qb2, v2@SEL_2)*/ * FROM v2)vv;。
  - 对于 QB_NAME Hint 中视图列表序列第一项之外的其他部分，只有 @SEL_1 可省略。即，如果声明处于当前部分的第一个查询块中，则 @SEL_1 可以省略，否则，不能省略 @SEL_。对于上面的例子：
    - 视图 v2 的第一个查询块可以声明为 QB_NAME(v2_1, v2)
    - 视图 v2 的第二个查询块可以声明为 QB_NAME(v2_2, v2.@SEL_2)
    - 视图 v1 的第一个查询块可以声明为 QB_NAME(v1_1, v2.v1@SEL_2)
    - 视图 v1 的第二个查询块可以声明为 QB_NAME(v1_2, v2.v1@SEL_2 .@SEL_2)

第 2 步：添加实际需要的 Hint

在定义好视图查询块部分的 QB_NAME Hint 后，你可以通过查询块的名字使用查询块范围生效的 Hint，以 “视图名@查询块名” 的方式加入实际需要的 Hint，使其在视图内部生效。例如：

- 指定视图 v2 中第一个查询块的 MERGE_JOIN() Hint：

```
SELECT /*+ QB_NAME(v2_1, v2) merge_join(t@v2_1) */ * FROM v2;
```

- 指定视图 v2 中第二个查询块的 MERGE_JOIN() 和 STREAM_AGG() Hint：

```
SELECT /*+ QB_NAME(v2_2, v2.@SEL_2) merge_join(t1@v2_2) stream_agg(@v2_2) */ * FROM v2;
```

- 指定视图 v1 中第一个查询块的 HASH_JOIN() Hint：

```
SELECT /*+ QB_NAME(v1_1, v2.v1@SEL_2) hash_join(t@v1_1) */ * FROM v2;
```

- 指定视图 v1 中第二个查询块的 HASH_JOIN() 和 HASH_AGG() Hint：

```
SELECT /*+ QB_NAME(v1_2, v2.v1@SEL_2 .@SEL_2) hash_join(t1@v1_2) hash_agg(@v1_2) */ * FROM
    ↪ v2;
```

8.3.4.2.4 查询范围生效的 Hint

这类 Hint 只能跟在语句中第一个 SELECT、UPDATE 或 DELETE 关键字的后面，等同于在当前这条查询运行时对指定的系统变量进行修改，其优先级高于现有系统变量的值。

> 注意：
>
> 这类 Hint 虽然也有隐藏的可选变量 @QB_NAME，但就算指定了该值，Hint 还是会在整个查询范围生效。

NO_INDEX_MERGE()

NO_INDEX_MERGE() 会关闭优化器的index merge 功能。

下面的例子不会使用index merge：

```
SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE t.a > 0 or t.b > 0;
```

除了 Hint 外，系统变量 tidb_enable_index_merge 也能决定是否开启该功能。

> 注意：
>
> - NO_INDEX_MERGE 优先级高于 USE_INDEX_MERGE，当这两类 Hint 同时存在时，USE_INDEX_MERGE 不会生效。
> - 当存在子查询时，NO_INDEX_MERGE 放在最外层才能生效。

USE_TOJA(boolean_value)

参数 boolean_value 可以是 TRUE 或者 FALSE。USE_TOJA(TRUE) 会开启优化器尝试将in (subquery) 条件转换为join 和aggregation 的功能。相对地，USE_TOJA(FALSE) 会关闭该功能。

下面的例子会将 in (SELECT t2.a FROM t2)subq 转换为等价的join 和 aggregation：

```
SELECT /*+ USE_TOJA(TRUE) */ t1.a, t1.b FROM t1 WHERE t1.a in (SELECT t2.a FROM t2) subq;
```

除了 Hint 外，系统变量 tidb_opt_insubq_to_join_and_agg 也能决定是否开启该功能。

MAX_EXECUTION_TIME(N)

MAX_EXECUTION_TIME(N) 把语句的执行时间限制在N 毫秒以内，超时后服务器会终止这条语句的执行。

下面的 Hint 设置了 1000 毫秒（即 1 秒）超时：

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 inner join t2 WHERE t1.id = t2.id;
```

除了 Hint 之外，系统变量 global.max_execution_time 也能对语句执行时间进行限制。

MEMORY_QUOTA(N)

MEMORY_QUOTA(N) 用于限制语句执行时的内存使用。该 Hint 支持 MB 和 GB 两种单位。内存使用超过该限制时会根据当前设置的内存超限行为来打出一条log 或者终止语句的执行。

下面的 Hint 设置了 1024 MB 的内存限制：

```
SELECT /*+ MEMORY_QUOTA(1024 MB) */ * FROM t;
```

除了 Hint 外，系统变量tidb_mem_quota_query 也能限制语句执行的内存使用。

READ_CONSISTENT_REPLICA()

READ_CONSISTENT_REPLICA() 会开启从数据一致的TiKV follower 节点读取数据的特性。

下面的例子会从 follower 节点读取数据：

```
SELECT /*+ READ_CONSISTENT_REPLICA() */ * FROM t;
```

除了 Hint 外，环境变量 tidb_replica_read 设为 'follower' 或者 'leader' 也能决定是否开启该特性。

IGNORE_PLAN_CACHE()

IGNORE_PLAN_CACHE() 提示优化器在处理当前 prepare 语句时不使用plan cache。

该 Hint 用于在Prepared Plan Cache 开启的场景下临时对某类查询禁用plan cache。

以下示例强制该 prepare 语句不使用plan cache：

```
prepare stmt FROM 'SELECT  /*+ IGNORE_PLAN_CACHE() */ * FROM t WHERE t.id = ?';
```

STRAIGHT_JOIN()

STRAIGHT_JOIN() 提示优化器在生成表连接顺序时按照表名在 FROM 子句中出现的顺序进行连接。

```
SELECT /*+ STRAIGHT_JOIN() */ * FROM t t1, t t2 WHERE t1.a = t2.a;
```

注意：

- STRAIGHT_JOIN 优先级高于 LEADING，当这两类 Hint 同时存在时，LEADING 不会生效。
- 建议使用 LEADING Hint，它比 STRAIGHT_JOIN Hint 更通用。

NTH_PLAN(N)

NTH_PLAN(N) 提示优化器选用在物理优化阶段搜索到的第N 个物理计划。N 必须是正整数。

如果指定的N 超出了物理优化阶段的搜索范围，TiDB 会返回 warning，并根据不存在该 Hint 时一样的策略选择最优物理计划。

该 Hint 在启用cascades planner 的情况下不会生效。

以下示例会强制优化器在物理阶段选择搜索到的第 3 个物理计划：

```
SELECT /*+ NTH_PLAN(3) */ count(*) from t where a > 5;
```

> 注意：
>
> NTH_PLAN(N) 主要用于测试用途，并且在未来不保证其兼容性，请谨慎使用。

RESOURCE_GROUP(resource_group_name)

RESOURCE_GROUP(resource_group_name) 用于使用资源管控(Resource Control) 实现资源隔离。此 Hint 将临时使用指定的资源组执行当前的语句。如果指定的资源组不存在，则该 Hint 将被忽略。

示例：

```
SELECT /*+ RESOURCE_GROUP(rg1) */ * FROM t limit 10;
```

8.3.4.3 执行计划管理(SPM)

执行计划管理，又称 SPM (SQL Plan Management)，是通过执行计划绑定，对执行计划进行人为干预的一系列功能，包括执行计划绑定、自动捕获绑定、自动演进绑定等。

8.3.4.3.1 执行计划绑定(SQL Binding)

执行计划绑定是 SPM 的基础。在优化器 Hints 中介绍了可以通过 Hint 的方式选择指定的执行计划，但有时需要在不修改 SQL 语句的情况下干预执行计划的选择。执行计划绑定功能使得可以在不修改 SQL 语句的情况下选择指定的执行计划。

> 注意：
>
> 要使用执行计划绑定，你需要拥有 SUPER 权限。如果在使用过程中系统提示权限不足，可参考权限管理补充所需权限。

创建绑定

你可以根据 SQL 或者历史执行计划为指定的 SQL 语句创建绑定。

根据 SQL 创建绑定

```
CREATE [GLOBAL | SESSION] BINDING FOR BindableStmt USING BindableStmt;
```

该语句可以在GLOBAL 或者 SESSION 作用域内为 SQL 绑定执行计划。目前，如下 SQL 类型(BindableStmt) 可创建执行计划绑定：SELECT、DELETE、UPDATE 和带有 SELECT 子查询的 INSERT/REPLACE。

> 注意：
>
> 绑定的优先级高于手工添加的 Hint，即在有绑定的时候执行带有 Hint 的语句时，该语句中控制优化器行为的 Hint 不会生效，但是其他类别的 Hint 仍然能够生效。

其中，有两类特定的语法由于语法冲突不能创建执行计划绑定，例如：

```
-- 类型一：使用 `JOIN` 关键字但不通过 `USING` 关键字指定关联列的笛卡尔积
CREATE GLOBAL BINDING for
    SELECT * FROM t t1 JOIN t t2
USING
    SELECT * FROM t t1 JOIN t t2;

-- 类型二：包含了 `USING` 关键字的 `delete` 语句
CREATE GLOBAL BINDING for
    delete FROM t1 USING t1 JOIN t2 ON t1.a = t2.a
USING
    delete FROM t1 USING t1 JOIN t2 ON t1.a = t2.a;
```

可以通过等价的 SQL 改写绕过这个语法冲突的问题。例如，上述两个例子可以改写为：

```
-- 类型一的第一种改写：为 `JOIN` 关键字添加 `USING` 子句
CREATE GLOBAL BINDING for
    SELECT * FROM t t1 JOIN t t2 USING (a)
USING
    SELECT * FROM t t1 JOIN t t2 USING (a);

-- 类型一的第二种改写：去掉 `JOIN` 关键字
CREATE GLOBAL BINDING for
    SELECT * FROM t t1, t t2
USING
    SELECT * FROM t t1, t t2;

-- 类型二的改写：去掉 `delete` 语句中的 `USING` 关键字
CREATE GLOBAL BINDING for
    delete t1 FROM t1 JOIN t2 ON t1.a = t2.a
USING
    delete t1 FROM t1 JOIN t2 ON t1.a = t2.a;
```

> 注意：
>
> 在对带 SELECT 子查询的 INSERT/REPLACE 语句创建执行计划绑定时，需要将想要绑定的优化器 Hints 指定在 SELECT 子查询中，而不是 INSERT/REPLACE 关键字后，不然优化器 Hints 不会生效。

例如：

```
-- Hint 能生效的用法
CREATE GLOBAL BINDING for
    INSERT INTO t1 SELECT * FROM t2 WHERE a > 1 AND b = 1
USING
    INSERT INTO t1 SELECT /*+ use_index(@sel_1 t2, a) */ * FROM t2 WHERE a > 1 AND b = 1;

-- Hint 不能生效的用法
CREATE GLOBAL BINDING for
    INSERT INTO t1 SELECT * FROM t2 WHERE a > 1 AND b = 1
USING
    INSERT /*+ use_index(@sel_1 t2, a) */ INTO t1 SELECT * FROM t2 WHERE a > 1 AND b = 1;
```

如果在创建执行计划绑定时不指定作用域，隐式作用域 SESSION 会被使用。TiDB 优化器会将被绑定的 SQL 进行“标准化”处理，然后存储到系统表中。在处理 SQL 查询时，只要“标准化”后的 SQL 和系统表中某个被绑定的 SQL 语句一致，并且系统变量tidb_use_plan_baselines 的值为 on（其默认值为 on），即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。

标准化：把 SQL 中的常量变成变量参数，对空格和换行符等做标准化处理，并对查询引用到的表显式指定数据库。例如：

```
SELECT * FROM t WHERE a >    1
-- 以上语句标准化后如下：
SELECT * FROM test . t WHERE a > ?
```

> 注意：
>
> 在进行标准化的时候，被逗号，连接起来的多个常量会被标准化为 ... 而不是?。
>
> 例如：
>
> ```
> SELECT * FROM t limit 10
> SELECT * FROM t limit 10, 20
> SELECT * FROM t WHERE a IN (1)
> SELECT * FROM t WHERE a IN (1,2,3)
> -- 以上语句标准化后如下：
> SELECT * FROM test . t limit ?
> SELECT * FROM test . t limit ...
> ```

```
SELECT * FROM test . t WHERE a IN ( ? )
SELECT * FROM test . t WHERE a IN ( ... )
```

因此包含单个常量的 SQL 语句和包含被逗号连接起来多个常量的 SQL 语句，在被绑定时会被 TiDB 视作不同的 SQL 语句，需要分别创建绑定。

值得注意的是，如果一条 SQL 语句在GLOBAL 和 SESSION 作用域内都有与之绑定的执行计划，因为优化器在遇到 SESSION 绑定时会忽略GLOBAL 绑定的执行计划，该语句在 SESSION 作用域内绑定的执行计划会屏蔽掉语句在 GLOBAL 作用域内绑定的执行计划。

例如：

```
-- 创建一个 global binding，指定其使用 sort merge join
CREATE GLOBAL BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id
USING
    SELECT /*+ merge_join(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;

-- 从该 SQL 的执行计划中可以看到其使用了 global binding 中指定的 sort merge join
explain SELECT * FROM t1, t2 WHERE t1.id = t2.id;

-- 创建另一个 session binding，指定其使用 hash join
CREATE BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id
USING
    SELECT /*+ hash_join(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;

-- 从该 SQL 的执行计划中可以看到其使用了 session binding 中指定的 hash join，而不是 global
    ↪ binding 中指定的 sort merge join
explain SELECT * FROM t1, t2 WHERE t1.id = t2.id;
```

第一个 SELECT 语句在执行时优化器会通过GLOBAL 作用域内的绑定为其加上 sm_join(t1, t2) hint，explain 出的执行计划中最上层的节点为 MergeJoin。而第二个 SELECT 语句在执行时优化器则会忽视GLOBAL 作用域内的绑定而使用 SESSION 作用域内的绑定为该语句加上 hash_join(t1, t2) hint，explain 出的执行计划中最上层的节点为 HashJoin。

每个标准化的 SQL 只能同时有一个通过 CREATE BINDING 创建的绑定。对相同的标准化 SQL 创建多个绑定时，会保留最后一个创建的绑定，之前的所有绑定（创建的和演进出来的）都会被删除。但 session 绑定和 global 绑定仍然允许共存，不受这个逻辑影响。

另外，创建绑定时，TiDB 要求 session 处于某个数据库上下文中，也就是执行过 use ${database} 或者客户端连接时指定了数据库。

需要注意的是原始 SQL 和绑定 SQL 在参数化以及去掉 Hint 后文本必须相同，否则创建会失败，例如：

```
CREATE BINDING FOR SELECT * FROM t WHERE a > 1 USING SELECT * FROM t use index(idx) WHERE a > 2;
```

可以创建成功，因为原始 SQL 和绑定 SQL 在参数化以及去掉 Hint 后文本都是 `SELECT * FROM test . t WHERE ' → a > ?`，而

```
CREATE BINDING FOR SELECT * FROM t WHERE a > 1 USING SELECT * FROM t use index(idx) WHERE b > 2;
```

则不可以创建成功，因为原始 SQL 在经过处理后是 `SELECT * FROM test . t WHERE a > ?`，而绑定 SQL 在经过处理后是 `SELECT * FROM test . t WHERE b > ?`。

> 注意：
>
> 对于 PREPARE/EXECUTE 语句组，或者用二进制协议执行的查询，创建执行计划绑定的对象应当是查询语句本身，而不是 PREPARE/EXECUTE 语句。

根据历史执行计划创建绑定

如需将 SQL 语句的执行计划固定为之前使用过的执行计划，可以使用 `plan_digest` 为该 SQL 语句绑定一个历史的执行计划。相比于使用 SQL 创建绑定的方式，此方式更加简便。

目前，根据历史执行计划创建绑定有一些限制：

- 该功能是根据历史的执行计划生成hint 而实现的绑定，历史的执行计划来源是Statement Summary Tables，因此在使用此功能之前需开启系统变量tidb_enable_stmt_summary。
- 对于带有子查询的查询、访问TiFlash 的查询、3 张表或更多表进行Join 的查询，目前还不支持通过历史执行计划进行绑定。

使用方式:

```
CREATE [GLOBAL | SESSION] BINDING FROM HISTORY USING PLAN DIGEST 'plan_digest';
```

该语句使用 `plan_digest` 为 SQL 语句绑定执行计划，在不指定作用域时默认作用域为 SESSION。所创建绑定的适用 SQL、优先级、作用域、生效条件等与根据 SQL 创建绑定相同。

使用此绑定方式时，你需要先从 `statements_summary` 中找到需要绑定的执行计划对应的 `plan_digest`，再通过 `plan_digest` 创建绑定。具体步骤如下：

1. 从 `Statement Summary Tables` 的记录中查找执行计划对应的 `plan_digest`。

   例如:

   ```
   CREATE TABLE t(id INT PRIMARY KEY , a INT, KEY(a));
   SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1;
   SELECT * FROM INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE QUERY_SAMPLE_TEXT = 'SELECT /*+
       ' → IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1'\G;
   ```

   以下为 `statements_summary` 部分查询结果：

```
SUMMARY_BEGIN_TIME: 2022-12-01 19:00:00
..........
      DIGEST_TEXT: select * from `t` where `a` = ?
..........
      PLAN_DIGEST: 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb
             PLAN:  id                  task         estRows operator info
                  ′→                                      actRows execution info
                  ′→
                  ′→ memory       disk
                     TableReader_7        root         10       data:Selection_6
                         ′→                            0       time:4.05ms, loops:1, cop_task: {
                         ′→ num: 1, max: 598.6µs, proc_keys: 0, rpc_num: 2, rpc_time: 609.8
                         ′→ µs, copr_cache_hit_ratio: 0.00, distsql_concurrency: 15}    176
                         ′→ Bytes    N/A
                      └─Selection_6        cop[tikv]    10       eq(test.t.a, 1)
                         ′→                            0       tikv_task:{time:560.8µs, loops
                         ′→ :0}
                         ′→
                         ′→ N/A          N/A
                        └─TableFullScan_5 cop[tikv]    10000    table:t, keep order:false,
                            ′→ stats:pseudo 0        tikv_task:{time:560.8µs, loops:0}
                            ′→
                            ′→ N/A          N/A
      BINARY_PLAN: 6
          ′→ QOYCuQDCg1UYWJsZVJlYWRlcl83Ev8BCgtTZWxlY3Rpb25fNhKOAQoPBSJQRnVsbFNjYW5fNSEBAAAAOA0
          ′→ /
          ′→ QSkAAQHwW4jDQDgCQAJKCwoJCgR0ZXN0EgF0Uh5rZWVwIG9yZGVyOmZhbHNlLCBzdGF0czpwc2V1ZG9qInRpa3ZfdGFzaz
          ′→ ////CQMEAXgJCBD///8
          ′→ BIQFzCDhVQw19BAAkBX0QUg9lcSgBfCAudC5hLCAxKWrmYQAYHOi0gc6hBB1hJAFAAVIQZGF0YTo9GgRaFAW4HDQuMDVtc
          ′→ ///////////8BGAE=
```

可以看到执行计划对应的plan_digest 为4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb ′→ 。

2. 使用 plan_digest 创建绑定。

```
CREATE BINDING FROM HISTORY USING PLAN DIGEST '4
    ′→ e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb';
```

创建完毕后可以查看绑定，验证绑定是否生效。

```
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
       ...........
  Sql_digest: 6909a1bbce5f64ade0a532d7058dd77b6ad5d5068aee22a531304280de48349f
 Plan_digest:
1 row in set (0.01 sec)

ERROR:
No query specified
```

```
SELECT * FROM t WHERE a = 1;
SELECT @@LAST_PLAN_FROM_BINDING;
```

```
+-----------------------------------------+
| @@LAST_PLAN_FROM_BINDING |
+-----------------------------------------+
|                          1 |
+-----------------------------------------+
1 row in set (0.00 sec)
```

删除绑定

你可以根据 SQL 语句或者 sql_digest 删除绑定。

根据 SQL 语句删除绑定

```
DROP [GLOBAL | SESSION] BINDING FOR BindableStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内删除指定的执行计划绑定，在不指定作用域时默认作用域为 SESSION。

一般来说，SESSION 作用域的绑定主要用于测试或在某些特殊情况下使用。若需要集群中所有的TiDB 进程都生效，则需要使用 GLOBAL 作用域的绑定。SESSION 作用域对GLOBAL 作用域绑定的屏蔽效果会持续到该 SESSION 结束。

承接上面关于 SESSION 绑定屏蔽GLOBAL 绑定的例子，继续执行：

```
-- 删除 session 中创建的 binding
DROP session binding for SELECT * FROM t1, t2 WHERE t1.id = t2.id;

-- 重新查看该 SQL 的执行计划
explain SELECT * FROM t1,t2 WHERE t1.id = t2.id;
```

在这里 SESSION 作用域内被删除掉的绑定会屏蔽GLOBAL 作用域内相应的绑定，优化器不会为 SELECT 语句添加 sm_join(t1, t2) hint，explain 给出的执行计划中最上层节点并不被hint 固定为 MergeJoin，而是由优化器经过代价估算后自主进行选择。

根据 sql_digest 删除绑定

除了可以根据 SQL 语句删除对应的绑定以外，也可以根据 sql_digest 删除绑定：

```
DROP [GLOBAL | SESSION] BINDING FOR SQL DIGEST 'sql_digest';
```

该语句用于在GLOBAL 或者 SESSION 作用域内删除 sql_digest 对应的的执行计划绑定，在不指定作用域时默认作用域为 SESSION。你可以通过查看绑定语句获取 sql_digest。

> 注意：
>
> 执行 DROP GLOBAL BINDING 会删除当前tidb-server 实例缓存中的绑定，并将系统表中对应行的状态修改为 ‘deleted’。该语句不会直接删除系统表中的记录，因为其他tidb-server 实例需要读取系统表中的‘deleted’ 状态来删除其缓存中对应的绑定。对于这些系统表中状态为‘deleted’ 的记录，后台线程每隔 100 个 bind-info-lease（默认值为 3s，合计 300s）会触发一次对 update_time 在 10 个 bind-info-lease 以前的绑定（确保所有tidb-server 实例已经读取过这个‘deleted’ 状态并更新完缓存）的回收清除操作。

变更绑定状态

根据 SQL 语句变更绑定状态

```
SET BINDING [ENABLED | DISABLED] FOR BindableStmt;
```

该语句可以在GLOBAL 作用域内变更指定执行计划的绑定状态，默认作用域为GLOBAL，该作用域不可更改。

使用时，只能将 Disabled 的绑定改为 Enabled 状态，或将 Enabled 的绑定改为 Disabled 状态。如果没有可以改变状态的绑定，则会输出一条内容为 There are no bindings can be set the status. Please check ′→ the SQL text 的警告。需要注意的是， 当绑定被设置成 Disabled 状态时， 查询语句不会使用该绑定。

根据 sql_digest 变更绑定状态

除了可以根据 SQL 语句变更对应的绑定状态以外，也可以根据 sql_digest 变更绑定状态：

```
SET BINDING [ENABLED | DISABLED] FOR SQL DIGEST 'sql_digest';
```

使用 sql_digest 所能变更的绑定状态和生效情况与根据 SQL 语句变更绑定状态相同。如果没有可以改变状态的绑定，则会输出一条内容为 can't find any binding for 'sql_digest' 的警告。

查看绑定

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhere];
```

该语句会按照绑定更新时间由新到旧的顺序输出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 11 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

列名	说明
default_db	默认数据库名
status	状态，包括 enabled（可用，从 v6.0 开始取代之前版本的using 状态）、disabled（不可用）、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
source	创建方式，包括 manual（根据 SQL 创建绑定生成）、history（根据历史执行计划创建绑定生成）、capture（由 TiDB 自动创建生成）和 evolve（由 TiDB 自动演进生成）
sql_digest	归一化后的 SQL 的 digest
plan_digest	执行计划的digest

排查绑定

绑定的排查通常有两种方式：

- 使用系统变量last_plan_from_binding 显示上一条执行语句是否采用binding 的执行计划。

```
-- 创建一个 global binding

CREATE GLOBAL BINDING for
    SELECT * FROM t
USING
    SELECT /*+ USE_INDEX(t, idx_a) */ * FROM t;
```

```
SELECT * FROM t;
SELECT @@[SESSION.]last_plan_from_binding;
```

```
+--------------------------+
| @@last_plan_from_binding |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

- 使用 `explain format = 'verbose'` 语句查看 SQL 语句的查询计划。如果 SQL 语句使用了binding，可以接着执行 `show warnings` 了解该 SQL 语句使用了哪一条binding。

```
-- 创建一个 global binding

CREATE GLOBAL BINDING for
    SELECT * FROM t
USING
    SELECT /*+ USE_INDEX(t, idx_a) */ * FROM t;

-- 使用 explain format = 'verbose' 语句查看 SQL 的执行计划

explain format = 'verbose' SELECT * FROM t;

-- 通过执行 `show warnings` 了解该 SQL 语句使用了哪一条 binding

show warnings;
```

```
+-------+------+-----------------------------------------------------------------------------------+
| Level | Code | Message                                                                           |
+-------+------+-----------------------------------------------------------------------------------+
| Note  | 1105 | Using the bindSQL: SELECT /*+ USE_INDEX(`t` `idx_a`)*/ * FROM `test`.`t` |
+-------+------+-----------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

对绑定进行缓存

每个TiDB 实例都有一个LRU (Least Recently Used) Cache 对绑定进行缓存，缓存的容量由系统变量tidb_mem_quota_binding_cache
↪ 进行控制。缓存会影响绑定的使用和查看，因此你只能使用和查看存在于缓存中的绑定。

如需查看绑定的使用情况，可以执行 `SHOW binding_cache status` 语句。该语句无法指定作用域，默认作用域为GLOBAL。该语句可查看缓存中可用绑定的数量、系统中所有可用绑定的数量、缓存中所有绑定的内存使用量及缓存的内存容量。

```
SHOW binding_cache status;
```

```
+-------------------+-------------------+--------------+---------------------------+
| bindings_in_cache | bindings_in_table | memory_usage | memory_quota |
+-------------------+-------------------+--------------+---------------------------+
|                 1 |                 1 | 159 Bytes    | 64 MB        |
+-------------------+-------------------+--------------+---------------------------+
1 row in set (0.00 sec)
```

8.3.4.3.2 自动捕获绑定(Baseline Capturing)

自动绑定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。

Plan Baseline 是一组被允许用于 SQL 语句优化器的可接受计划。在典型的应用场景中，TiDB 仅在验证计划性能良好后才将其添加到 Baseline 中。这些计划包含优化器重新生成执行计划所需的所有信息（例如，SQL 计划标识符、提示集、绑定值、优化器环境）。

使用方式

通过将 `tidb_capture_plan_baselines` 的值设置为 on（其默认值为 `off`）可以打开自动捕获绑定功能。

> 注意：
>
> 自动绑定功能依赖于Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。

开启自动绑定功能后，每隔 `bind-info-lease`（默认值为 `3s`）会遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动捕获绑定。绑定的执行计划为 Statement Summary 中记录执行这条语句时使用的执行计划。

对于以下几种 SQL 语句，TiDB 不会自动捕获绑定：

- EXPLAIN 和 EXPLAIN ANALYZE 语句；
- TiDB 内部执行的 SQL 语句，比如统计信息自动加载使用的 SELECT 查询；
- 存在 `Enabled` 或 `Disabled` 状态绑定的语句；
- 满足捕获绑定黑名单过滤条件的语句。

> 注意：
>
> 当前，绑定通过生成一组 Hints 来固定查询语句生成的执行计划，从而确保执行计划不发生变化。对于大多数 OLTP 查询，TiDB 能够保证计划前后一致，如使用相同的索引、相同的 Join 方式（如 HashJoin、IndexJoin）等。但是，受限于当前 Hints 的完善程度，对于一些较为复杂的查询，如两个表以上的Join 和复杂的 OLAP、MPP 类查询，TiDB 无法保证计划在绑定前后完全一致。

对于 PREPARE/EXECUTE 语句组，或通过二进制协议执行的查询，TiDB 会为真正的查询（而不是 PREPARE/EXECUTE 语句）自动捕获绑定。

> **注意：**
>
> 由于 TiDB 存在一些内嵌 SQL 保证一些功能的正确性，所以自动捕获绑定时会默认屏蔽内嵌 SQL。

过滤捕获绑定

使用本功能，你可以设置黑名单，将满足黑名单规则的查询排除在捕获范围之外。黑名单支持的过滤维度包括表名、频率和用户名。

使用方式

将过滤规则插入到系统表 `mysql.capture_plan_baselines_blacklist` 中，该过滤规则即刻起会在整个集群范围内生效。

```
-- 按照表名进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('table', '
    ↪ test.t');

-- 通过通配符来实现按照数据库名和表名进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('table', '
    ↪ test.table_*');
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('table', '
    ↪ db_*.table_*');

-- 按照执行频率进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('frequency',
    ↪  '2');

-- 按照用户名进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('user', '
    ↪ user1');
```

维度名称	说明	注意事项
table	按照表名进行过滤，每个过滤规则均采用 `db.table` 形式，支持通配符。详细规则可以参考直接使用表名和使用通配符。	字母大小写不敏感，如果包含非法内容，日志会输出 `[sql-bind] failed to load mysql ↪ .capture_plan_baselines_blacklist` 警告。
frequency	按照频率进行过滤，默认捕获执行超过一次的语句。可以设置较大值来捕获执行频繁的语句。	插入的值小于 1 会被认为是非法值，同时，日志会输出 `[sql-bind] frequency ↪ threshold is less than 1, ignore it` 警告。如果插入了多条频率过滤规则，频率最大的值会被用作过滤条件。

维度名称	说明	注意事项
user	按照用户名进行过滤，黑名单用户名执行的语句不会被捕获。	如果多个用户执行同一条语句，只有当他们的用户名都在黑名单的时候，该语句才不会被捕获。

> 注意：
>
> - 修改黑名单需要数据库的 super privilege 权限。
> - 如果黑名单包含了非法的过滤内容时，TiDB 会在日志中输出`[sql-bind] unknown capture filter type, ignore it` 进行提示。

升级时的计划回退防护

当需要升级TiDB 集群时，你可以利用自动捕获绑定对潜在的计划回退风险进行一定程度的防护，具体流程为：

1. 升级前打开自动捕获。

> 注意：
> 经测试，长期打开自动捕获对集群负载的性能影响很小。尽量长期打开自动捕获，以确保重要的查询（出现过两次及以上）都能被捕获到。

2. 进行TiDB 集群的升级。在升级完成后，这些通过捕获的绑定会发挥作用，确保在升级后，查询的计划不会改变。
3. 升级完成后，根据情况手动删除绑定。
    - 通过`SHOW GLOBAL BINDINGS`语句检查绑定来源：
      根据输出中的 `Source` 字段对绑定的来源进行区分，确认是通过捕获(`capture`) 生成还是通过手动创建(`manual`) 生成。
    - 确定 `capture` 的绑定是否需要保留：

```
-- 查看绑定生效时的计划
SET @@SESSION.TIDB_USE_PLAN_BASELINES = true;
EXPLAIN FORMAT='VERBOSE' SELECT * FROM t1 WHERE ...;

-- 查看绑定不生效时的计划
SET @@SESSION.TIDB_USE_PLAN_BASELINES = false;
EXPLAIN FORMAT='VERBOSE' SELECT * FROM t1 WHERE ...;
```

        - 如果屏蔽绑定前后，查询得到的计划一致，则可以安全删除此绑定。
        - 如果计划不一样，则可能需要对此计划变化的原因进行排查，如检查统计信息等操作。在这种情况下需要保留此绑定，确保计划不发生变化。

8.3.4.3.3 自动演进绑定(Baseline Evolution)

自动演进绑定，在TiDB 4.0 版本引入，是执行计划管理的重要功能之一。

由于某些数据变更后，原先绑定的执行计划可能是一个不优的计划。为了解决该问题，引入自动演进绑定功能来自动优化已经绑定的执行计划。

另外自动演进绑定还可以一定程度上避免统计信息改动后，对执行计划带来的抖动。

使用方式

通过以下语句可以开启自动演进绑定功能：

```
SET GLOBAL tidb_evolve_plan_baselines = ON;
```

tidb_evolve_plan_baselines 的默认值为 off。

> 警告：
>
> - 自动演进功能目前为实验特性，存在未知风险，不建议在生产环境中使用。
> - 此变量开关已强制关闭，直到自动演进成为正式功能GA (Generally Available)。如果你尝试打开开关，会产生报错。如果你已经在生产环境中使用了此功能，请尽快将它禁用。如发现binding 状态不如预期，请从 PingCAP 官方或TiDB 社区获取支持。

在打开自动演进功能后，如果优化器选出的最优执行计划不在之前绑定的执行计划之中，会将其记录为待验证的执行计划。每隔 bind-info-lease（默认值为 3s），会选出一个待验证的执行计划，将其和已经绑定的执行计划中代价最小的比较实际运行时间。如果待验证的运行时间更优的话（目前判断标准是运行时间小于等于已绑定执行计划运行时间的 2/3），会将其标记为可使用的绑定。以下示例描述上述过程。

假如有表t 定义如下：

```
CREATE TABLE t(a INT, b INT, KEY(a), KEY(b));
```

在表t 上进行如下查询：

```
SELECT * FROM t WHERE a < 100 AND b < 100;
```

表上满足条件a < 100 的行很少。但由于某些原因，优化器没能选中使用索引a 这个最优执行计划，而是误选了速度慢的全表扫，那么用户首先可以通过如下语句创建一个绑定：

```
CREATE GLOBAL BINDING for SELECT * FROM t WHERE a < 100 AND b < 100 USING SELECT * FROM t use
    ↪ index(a) WHERE a < 100 AND b < 100;
```

当以上查询语句再次执行时，优化器会在刚创建绑定的干预下选择使用索引a，进而降低查询时间。

假如随着在表中进行插入和修改，表中满足条件a < 100 的行变得越来越多，而满足条件b < 100 的行变得越来越少，这时再在绑定的干预下使用索引a 可能就不是最优了。

绑定的演进可以解决这类问题。当优化器感知到表数据变化后，会对这条查询生成使用索引b 的执行计划。但由于绑定的存在，这个执行计划不会被采纳和执行，不过它会被存在后台的演进列表里。在演进过程中，

如果它被验证为执行时间明显低于使用索引a 的执行时间（即当前绑定的执行计划），那么索引 b 会被加入到可用的绑定列表中。在此之后，当这条查询再次被执行时，优化器首先生成使用索引b 的执行计划，并确认它在绑定列表中，所以会采纳它并执行，进而可以在数据变化后降低这条查询的执行时间。

为了减少自动演进对集群的影响，可以通过设置 tidb_evolve_plan_task_max_time 来限制每个执行计划运行的最长时间，其默认值为 600s。实际在验证执行计划时，计划的最长运行时间还会被限制为不超过已验证执行计划的运行时间的两倍；通过 tidb_evolve_plan_task_start_time 和 tidb_evolve_plan_task_end_time 可以限制运行演进任务的时间窗口，默认值分别为 00:00 +0000 和 23:59 +0000。

注意事项

由于自动演进绑定会自动地创建新的绑定，当查询的环境发生变动时，自动创建的绑定可能会有多种行为的选择。这里列出一些注意事项：

- 自动演进只会对存在至少一个 global 绑定的标准化 SQL 进行演进。
- 由于创建新的绑定会删除之前所有绑定（对于一条标准化 SQL），自动演进的绑定也会在手动重新创建绑定后被删除。
- 所有和计算过程相关的hint，在演进时都会被保留。计算过程相关的hint 有如下几种：

Hint	说明
memory_quota	查询过程最多可以使用多少内存
use_toja	优化器是否考虑把子查询转化为join
use_cascades	是否使用cascades 优化器
no_index_merge	优化器是否考虑将 index merge 作为一个读表选项
read_consistent_replica	是否强制读表时使用 follower read
max_execution_time	查询过程最多消耗多少时间

- read_from_storage 是一个非常特别的hint，因为它指定了读表时选择从TiKV 读还是从TiFlash 读。由于TiDB 提供隔离读的功能，当隔离条件变化时，这个hint 对演进出来的执行计划影响很大，所以当最初创建的绑定中存在这个hint，TiDB 会无视其所有演进的绑定。

8.3.4.3.4 升级检查(Upgrade Checklist)

执行计划管理功能(SPM) 在版本升级过程中可能会出现一些兼容性问题导致升级失败，你需要在版本升级前做一些检查，确保版本顺利升级。

- 当你尝试从 v5.2 以前的版本（即 v4.0、v5.0、v5.1）升级到当前版本，需要注意在升级前检查自动演进的开关 tidb_evolve_plan_baselines 是否已经关闭。如果尚未关闭，则需要将其关闭后再进行升级。具体操作如下所示：

```
-- 在待升级的版本上检查自动演进的开关 `tidb_evolve_plan_baselines` 是否关闭。

SELECT @@global.tidb_evolve_plan_baselines;

-- 如果演进的开关 `tidb_evolve_plan_baselines` 尚未关闭，则需要将其关闭。
```

```
set global tidb_evolve_plan_baselines = off;
```

- 当你尝试从 v4.0 版本升级到当前版本，需要注意在升级前检查所有可用绑定对应的查询语句在新版本中是否存在语法错误。如果存在语法错误，则需要删除对应的绑定。

  具体操作如下所示：

```
-- 在待升级的版本上检查现有可用绑定对应的查询语句。

SELECT bind_sql FROM mysql.bind_info WHERE status = 'using';

-- 将上一条查询得到的结果，在新版本的测试环境中进行验证。

bind_sql_0;
bind_sql_1;
...

-- 如果报错信息是语法错误（ERROR 1064 (42000): You have an error in your SQL syntax），
    ↪ 则需要删除对应的绑定。
-- 如果是其他错误，如未找到表，则表示语法兼容，不需要进行额外的处理。
```

8.3.4.4 优化规则与表达式下推的黑名单

本文主要介绍优化规则的黑名单与表达式下推的黑名单。

8.3.4.4.1 优化规则黑名单

优化规则黑名单是针对优化规则的调优手段之一，主要用于手动禁用一些优化规则。

重要的优化规则

优化规则	规则名称	简介
列裁剪	column_prune	对于上层算子不需要的列，不在下层算子输出该列，减少计算
子查询去关联	decorrelate	尝试对相关子查询进行改写，将其转换为普通 join 或 aggregation 计算
聚合消除	aggregation_eliminate	尝试消除执行计划中的某些不必要的聚合算子
投影消除	projection_eliminate	消除执行计划中不必要的投影算子
最大最小消除	max_min_eliminate	改写聚合中的 max/min 计算，转化为 `order by + limit 1`
谓词下推	predicate_push_down	尝试将执行计划中过滤条件下推到离数据源更近的算子上
外连接消除	outer_join_eliminate	尝试消除执行计划中不必要的 left join 或者 right join
分区裁剪	partition_processor	将分区表查询改成为用 union all，并裁剪掉不满足过滤条件的分区
聚合下推	aggregation_push_down	尝试将执行计划中的聚合算子下推到更底层的计算节点
TopN 下推	topn_push_down	尝试将执行计划中的TopN 算子下推到离数据源更近的算子上
Join 重排序	join_reorder	对多表 join 确定连接顺序
从窗口函数中推导 TopN 或 Limit	derive_topn_from_window	从窗口函数中推导出 TopN 或者 Limit

禁用优化规则

当某些优化规则在一些特殊查询中的优化结果不理想时，可以使用优化规则黑名单禁用一些优化规则。

使用方法

> 注意：
>
> 以下操作都需要数据库的 super privilege 权限。每个优化规则都有各自的名字，比如列裁剪的名字是“column_prune”。所有优化规则的名字都可以在重要的优化规则表格中第二列查到。

- 如果你想禁用某些规则，可以在 `mysql.opt_rule_blacklist` 表中写入规则的名字，例如：

```
INSERT INTO mysql.opt_rule_blacklist VALUES("join_reorder"), ("topn_push_down");
```

执行以下 SQL 语句可让禁用规则立即生效，包括相应TiDB Server 的所有旧链接：

```
ADMIN reload opt_rule_blacklist;
```

> 注意：
>
> `admin reload opt_rule_blacklist` 只对执行该 SQL 语句的TiDB server 生效。若需要集群中所有TiDB server 生效，需要在每台TiDB server 上执行该 SQL 语句。

- 需要解除一条规则的禁用时，需要删除表中禁用该条规则的相应数据，再执行 `admin reload`：

```
DELETE FROM mysql.opt_rule_blacklist WHERE name IN ("join_reorder", "topn_push_down");
admin reload opt_rule_blacklist;
```

8.3.4.4.2 表达式下推黑名单

表达式下推黑名单是针对表达式下推的调优手段之一，主要用于对于某些存储类型手动禁用一些表达式。

已支持下推的表达式

目前已经支持下推的表达式信息，请参考表达式列表。

禁止特定表达式下推

当函数的计算过程由于下推而出现异常时，可通过黑名单功能禁止其下推来快速恢复业务。具体而言，你可以将上述支持的函数或运算符名加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。

`mysql.expr_pushdown_blacklist` 的 schema 如下：

```
DESC mysql.expr_pushdown_blacklist;
```

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

```
| name        | char(100)     | NO   |      | NULL               |        |
| store_type  | char(100)     | NO   |      | tikv,tiflash,tidb  |        |
| reason      | varchar(200)  | YES  |      | NULL               |        |
+-------------+---------------+------+------+--------------------+--------------- +
3 rows in set (0.00 sec)
```

以上结果字段解释如下：

- `name`：禁止下推的函数名。
- `store_type`：用于指明希望禁止该函数下推到哪些组件进行计算。组件可选 `tidb`、`tikv` 和 `tiflash`。`store_type` 不区分大小写，如果需要禁止向多个存储引擎下推，各个存储之间需用逗号隔开。
  - `store_type` 为 `tidb` 时表示在读取TiDB 内存表时，是否允许该函数在其他TiDB Server 上执行。
  - `store_type` 为 `tikv` 时表示是否允许该函数在TiKV Server 的 Coprocessor 模块中执行。
  - `store_type` 为 `tiflash` 时表示是否允许该函数在TiFlash Server 的 Coprocessor 模块中执行。
- `reason`：用于记录该函数被加入黑名单的原因。

使用方法

加入黑名单

如果要将一个或多个函数或运算符加入黑名单，执行以下步骤：

1. 向 `mysql.expr_pushdown_blacklist` 插入对应的函数名或运算符名以及希望禁止下推的存储引擎集合。
2. 执行 `admin reload expr_pushdown_blacklist;`。

移出黑名单

如果要将一个或多个函数及运算符移出黑名单，执行以下步骤：

1. 从 `mysql.expr_pushdown_blacklist` 表中删除对应的函数名或运算符名。
2. 执行 `admin reload expr_pushdown_blacklist;`。

> **注意：**
>
> `admin reload expr_pushdown_blacklist` 只对执行该 SQL 语句的TiDB server 生效。若需要集群中所有TiDB server 生效，需要在每台TiDB server 上执行该 SQL 语句。

表达式黑名单用法示例

以下示例首先将运算符< 及 > 加入黑名单，然后将运算符> 从黑名单中移出。

黑名单是否生效可以从 `explain` 结果中进行观察（参见TiDB 执行计划概览）。

1. 对于以下 SQL 语句，`where` 条件中的`a < 2` 和 `a > 2` 可以下推到TiKV 进行计算。

```
EXPLAIN SELECT * FROM t WHERE a < 2 AND a > 2;
```

```
+--
    '→ ----------------------+----------+-----------+--------------+-------------------------------------------------------------+
    '→
| id                        | estRows  | task      | access object | operator info
    '→                          |
+--
    '→ ----------------------+----------+-----------+--------------+-------------------------------------------------------------+
    '→
| TableReader_7             | 0.00     | root      |               | data:Selection_6
    '→                        |
| └─Selection_6              | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2), lt(
    '→ ssb_1.t.a, 2) |
|   └─TableFullScan_5       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo      |
+--
    '→ ----------------------+----------+-----------+--------------+-------------------------------------------------------------+
    '→
3 rows in set (0.00 sec)
```

2. 往mysql.expr_pushdown_blacklist表中插入禁用表达式，并且执行admin reload expr_pushdown_blacklist
   '→。

```
INSERT INTO mysql.expr_pushdown_blacklist VALUES('<','tikv',''), ('>','tikv','');
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
ADMIN reload expr_pushdown_blacklist;
```

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

3. 重新观察执行计划，发现表达式下推黑名单生效，where 条件中的< 和 > 没有被下推到TiKV Coprocessor
   上。

```
EXPLAIN SELECT * FROM t WHERE a < 2 and a > 2;
```

```
+--
    '→ ----------------------+----------+-----------+--------------+-------------------------------------------------------------+
    '→
| id                        | estRows  | task      | access object | operator info
    '→                          |
+--
    '→ ----------------------+----------+-----------+--------------+-------------------------------------------------------------+
    '→
```

```
| Selection_7              | 10000.00 | root      |               | gt(ssb_1.t.a, 2), lt(
    '→ ssb_1.t.a, 2) |
| └─TableReader_6          | 10000.00 | root      |               | data:TableFullScan_5
    '→                |
|   └─TableFullScan_5      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    '→ pseudo      |
+--
    '→ ----------------------+----------+-----------+---------------+-------------------------------------------------------------+
    '→
3 rows in set (0.00 sec)
```

4. 将某一表达式（>大于）禁用规则从黑名单表中删除，并且执行admin reload expr_pushdown_blacklist
   '→。

```
DELETE FROM mysql.expr_pushdown_blacklist WHERE name = '>';
```

```
Query OK, 1 row affected (0.01 sec)
```

```
ADMIN reload expr_pushdown_blacklist;
```

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

5. 重新观察执行计划，可以看到只有> 表达式被重新下推到TiKV Coprocessor，< 表达式仍然被禁用下推。

```
EXPLAIN SELECT * FROM t WHERE a < 2 AND a > 2;
```

```
+--
    '→ ------------------------+----------+-----------+---------------+-------------------------------------------------------------+
    '→
| id                         | estRows  | task      | access object | operator info
    '→                   |
+--
    '→ ------------------------+----------+-----------+---------------+-------------------------------------------------------------+
    '→
| Selection_8                | 0.00     | root      |               | lt(ssb_1.t.a, 2)
    '→                |
| └─TableReader_7            | 0.00     | root      |               | data:Selection_6
    '→                |
|   └─Selection_6            | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2)
    '→                |
|     └─TableFullScan_5      | 10000.00 | cop[tikv] | table:t       | keep order:false,
    '→ stats:pseudo |
+--
    '→ ------------------------+----------+-----------+---------------+-------------------------------------------------------------+
    '→
4 rows in set (0.00 sec)
```

9 教程

9.1 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。

9.1.1 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。

9.1.2 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。

9.1.2.1 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。

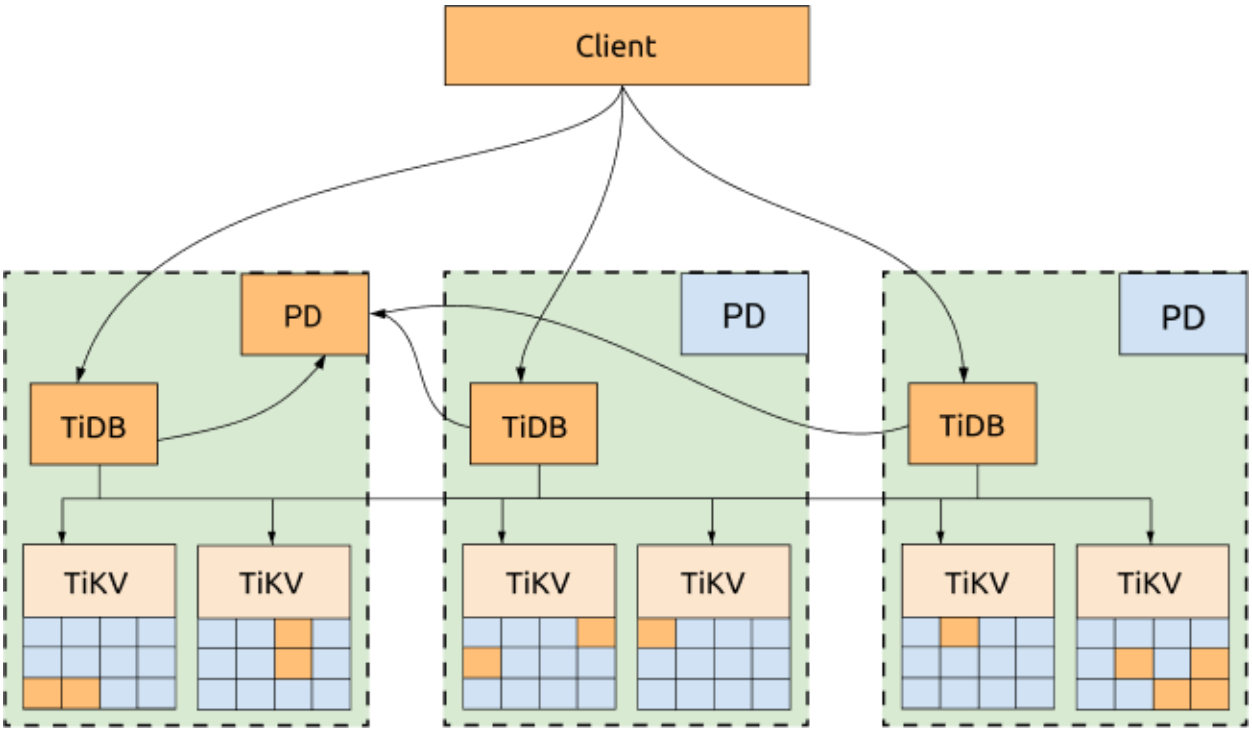


图 155: 三 AZ 部署

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

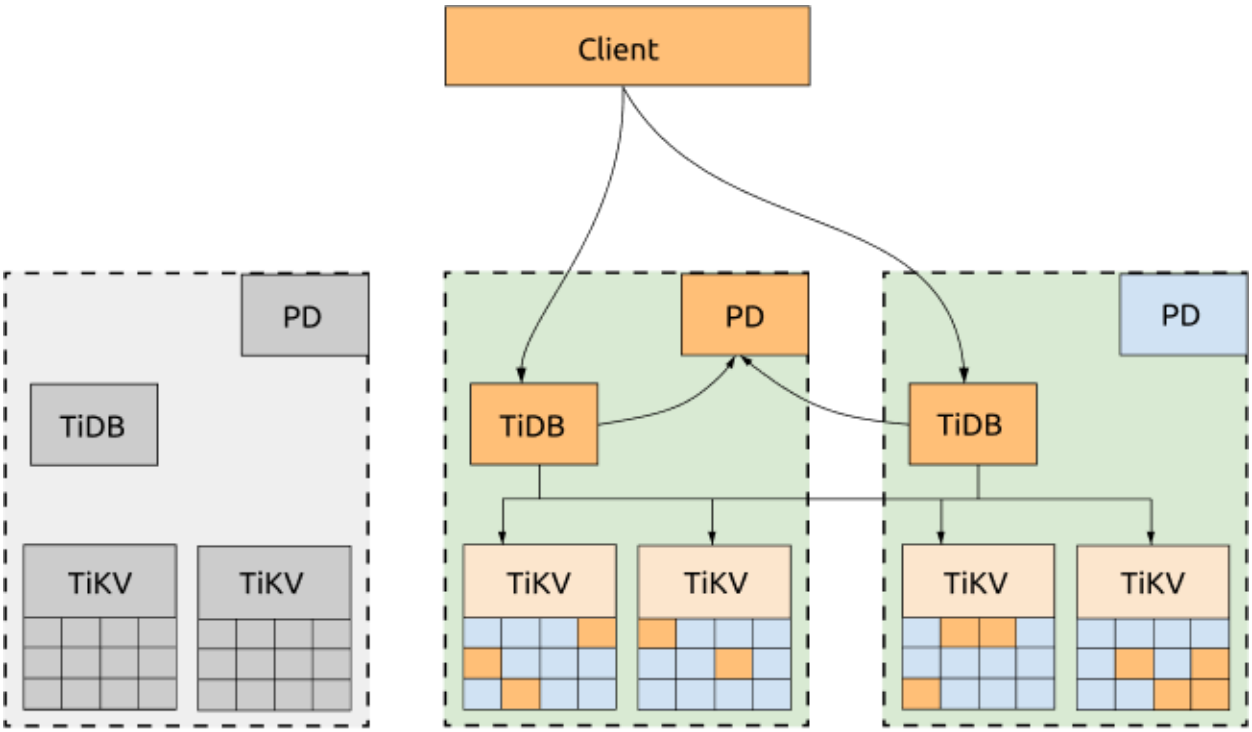


图 156: 三 AZ 部署容灾

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。

9.1.2.2 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。

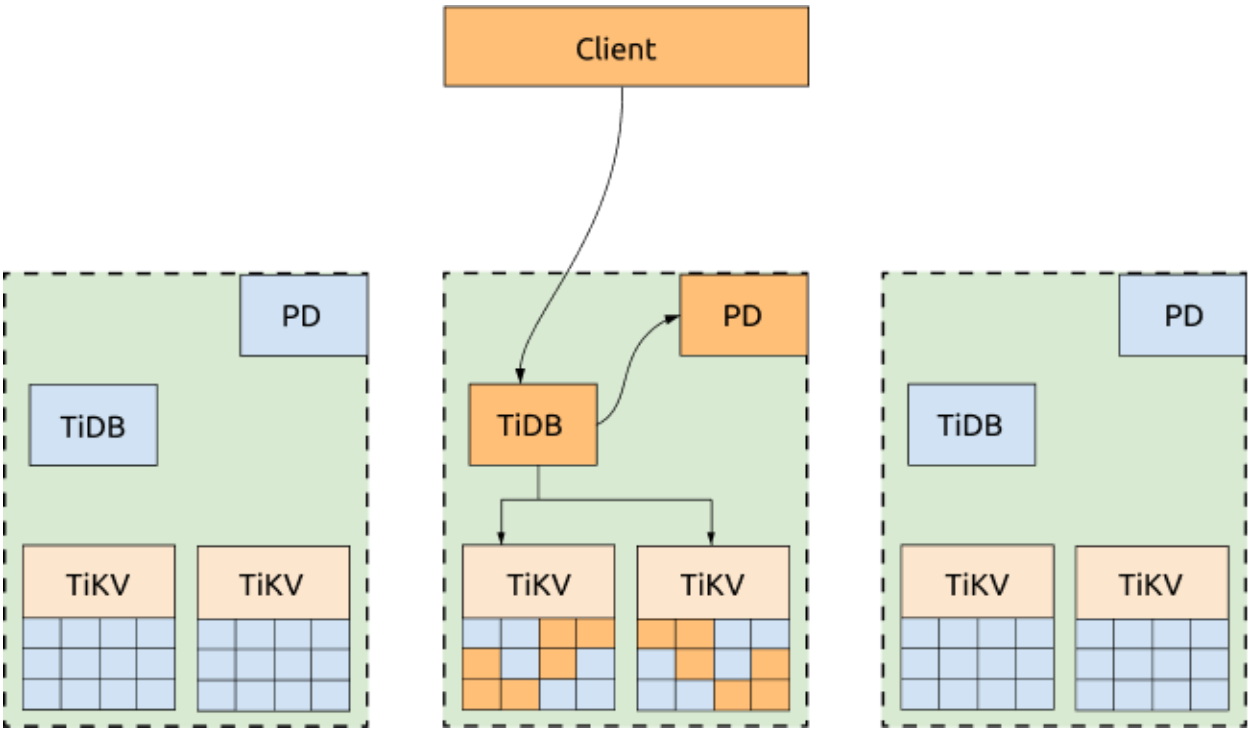


图 157: 三 AZ 部署读性能优化

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

9.1.2.3 样例部署图

9.1.2.3.1 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：

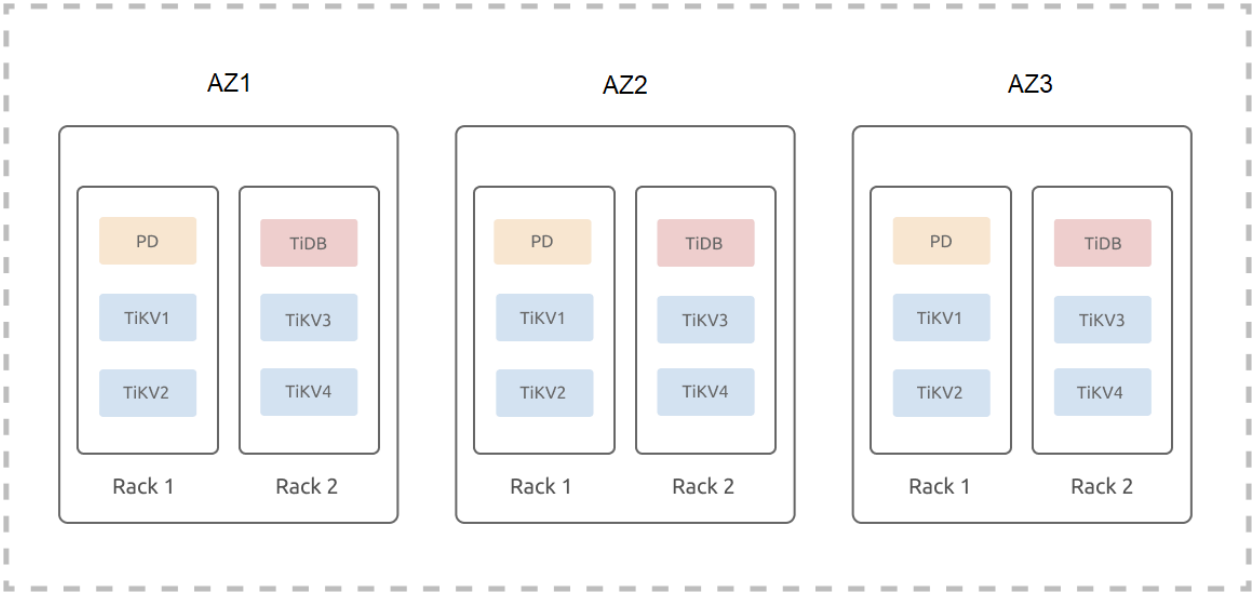


图 158: 同区域三AZ 集群部署

9.1.2.3.2 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。

9.1.2.3.3 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。

9.1.2.4 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。

9.2 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。

9.2.1 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。

9.2.2 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。

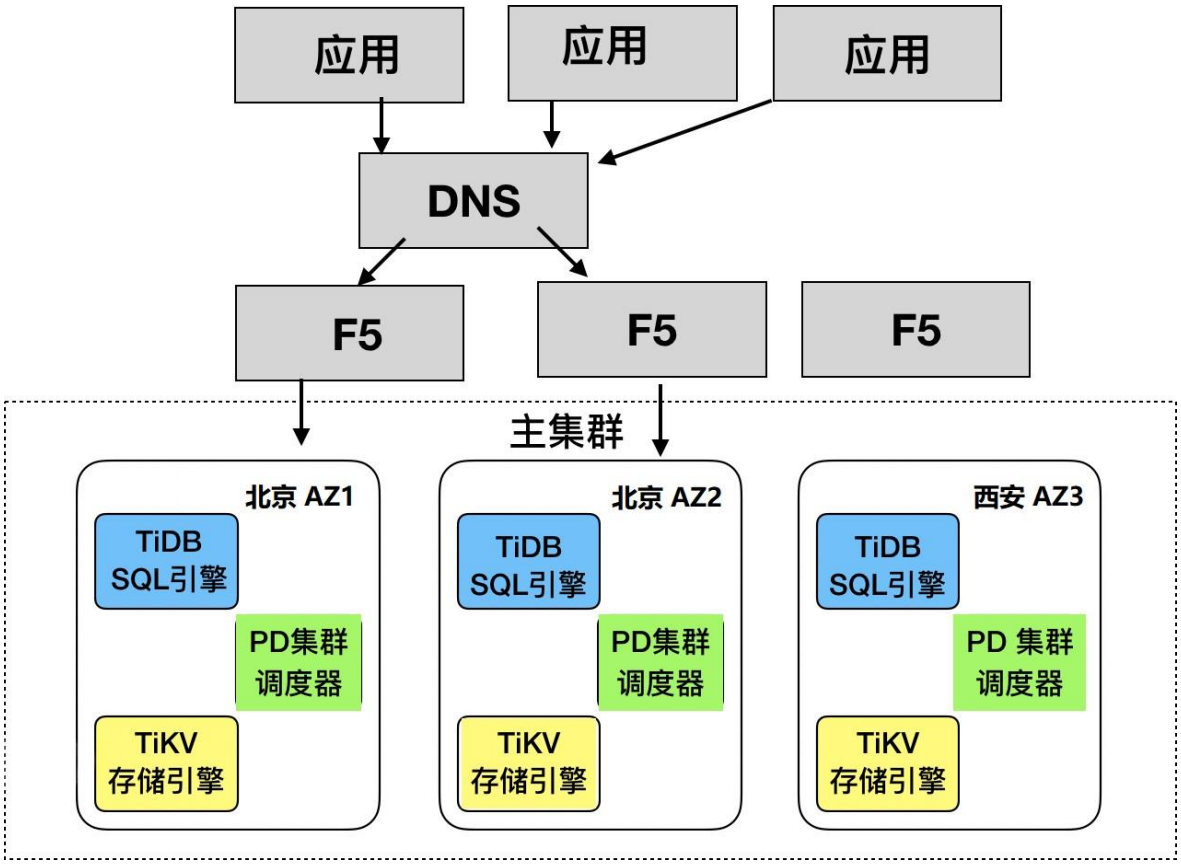


图 159: 双区域三AZ 集群架构图

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。

9.2.2.1 详细示例

北京、西安双区域三AZ 配置详解：

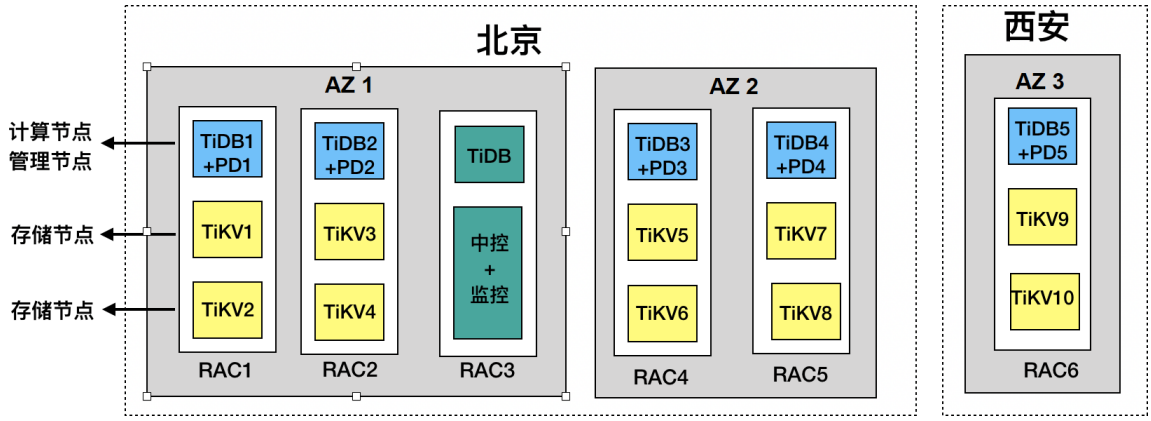


图 160: 双区域三AZ 配置详图

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。

9.2.3 配置

9.2.3.1 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
```

9.2.3.2 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。

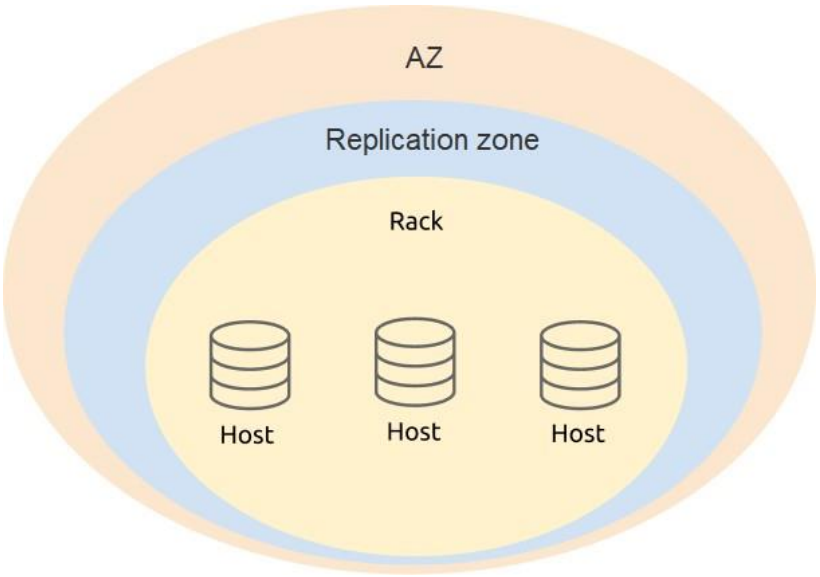


图 161: Label 逻辑定义图

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" }
```

9.2.3.3 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
```

9.3 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。

9.3.1 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。

9.3.2 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。

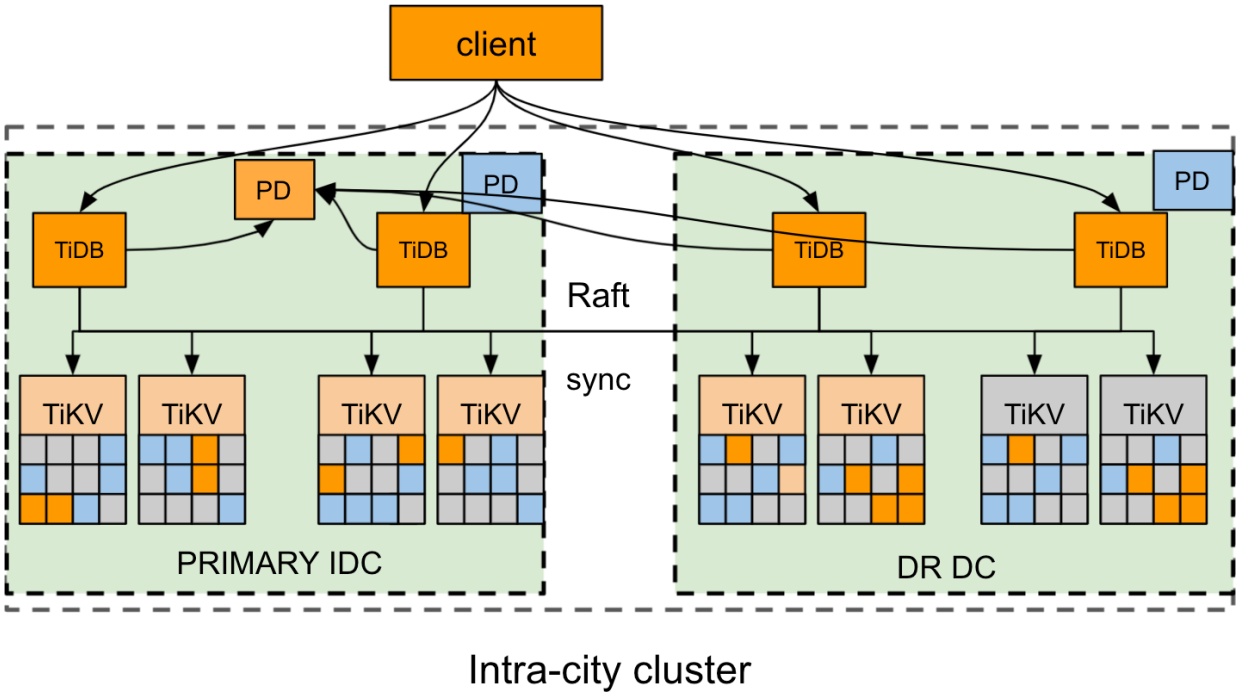


图 162: 单区域双AZ 集群架构图

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。

9.3.3 配置

9.3.3.1 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
```

9.3.3.2 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
      }
    ]
  }
]
```

9.3.3.3 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"
  }
}
```

9.3.3.3.1 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。

9.3.3.4 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）。

9.4 读取历史数据

9.4.1 使用 Stale Read 功能读取历史数据（推荐）

9.4.1.1 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。

9.4.1.1.1 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。

9.4.1.1.2 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。

9.4.1.1.3 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。

9.4.1.2 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。

9.4.1.2.1 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。

9.4.1.2.2 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。

9.4.1.3 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。

9.4.1.3.1 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。

9.4.1.3.2 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)
```

9.4.1.4 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。

9.4.1.4.1 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。

9.4.1.4.2 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`。

9.4.1.4.3 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。

9.4.2 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。

9.4.2.1 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。

9.4.2.2 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。

9.4.2.3 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。

9.4.2.4 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。

9.4.2.5 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。

10 TiDB 工具

10.1 TiDB 工具功能概览

TiDB 提供了丰富的工具，可以帮助你进行部署运维、数据管理（例如，数据迁移、备份恢复、数据校验）、在 TiKV 上运行 Spark SQL。请根据需要选择适用的工具。

10.1.1 部署运维工具

TiDB 提供了TiUP、TiDB Operator 和TiUniManager 三种部署运维工具，满足你在不同系统环境下的部署运维需求。

10.1.1.1 在物理机或虚拟机上部署运维TiDB

10.1.1.1.1 TiUP

TiUP 是在物理机或虚拟机上的TiDB 包管理器，管理着TiDB 的众多的组件，如TiDB、PD、TiKV 等。当你想要运行 TiDB 生态中任何组件时，只需要执行一行TiUP 命令即可。

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

基本信息：

- 术语及核心概念
- 使用TiUP 部署TiDB 集群
- TiUP 组件管理
- 适用TiDB 版本：v4.0 及以上

10.1.1.1.2 TiUniManager

TiUniManager 是一款以TiDB 数据库为核心的数据库管理平台，帮助用户在私有部署(on-premises) 或公有云环境中管理TiDB 集群。

TiUniManager 不仅提供对TiDB 集群的全生命周期的可视化管理，也同时一站式提供TiDB 数据库参数管理、数据库版本升级、克隆集群、主备集群切换、数据导入导出、数据同步、数据备份恢复服务，能有效提高TiDB 集群运维效率，降低企业运维成本。

基本信息：

- TiUniManager 使用场景
- TiUniManager 安装和运维指南
- TiUniManager 与 TiUP 的关系

10.1.1.2 在 Kubernetes 上部署运维TiDB - TiDB Operator

TiDB Operator 是 Kubernetes 上的TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助TiDB Operator，TiDB 可以无缝运行在公有云或私有部署的 Kubernetes 集群上。

基本信息：

- TiDB Operator 架构
- 在 Kubernetes 上部署运维TiDB 快速上手
- 适用TiDB 版本：v2.1 及以上

10.1.2 数据管理工具

TiDB 提供了丰富的数据管理工具，例如数据迁移、导入导出、备份恢复、增量同步、数据校验等。

10.1.2.1 数据迁入 - TiDB Data Migration (DM)

TiDB Data Migration (DM) 是将 MySQL/MariaDB 数据迁移到TiDB 的工具，支持全量数据的迁移和增量数据的复制。

基本信息：

- TiDB DM 的输入：MySQL/MariaDB
- TiDB DM 的输出：TiDB 集群
- 适用TiDB 版本：所有版本
- Kubernetes 支持：使用TiDB Operator 在 Kubernetes 上部署TiDB DM。

如果数据量在 TB 级别以下，推荐直接使用TiDB DM 迁移 MySQL/MariaDB 数据到TiDB（迁移的过程包括全量数据的导出导入和增量数据的复制）。

如果数据量在 TB 级别，推荐的迁移步骤如下：

1. 使用Dumpling 导出 MySQL/MariaDB 全量数据。
2. 使用TiDB Lightning 将全量导出数据导入TiDB 集群。
3. 使用TiDB DM 复制 MySQL/MariaDB 增量数据到TiDB。

> 注意：
>
> - 原 Syncer 工具已停止维护，不再推荐使用，相关场景请使用TiDB DM 的增量复制模式进行替代。

10.1.2.2 全量导出 - Dumpling

Dumpling 是一个用于从 MySQL/TiDB 进行全量逻辑导出的工具。

基本信息：

- Dumpling 的输入：MySQL/TiDB 集群
- Dumpling 的输出：SQL/CSV 文件
- 适用TiDB 版本：所有版本
- Kubernetes 支持：尚未支持

> 注意：
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 Mydumper project， 针对 TiDB 的特性进行了优化。Mydumper 已经被Dumpling 工具取代，并使用Go 语言编写，支持更多针对TiDB 特性的优化。建议切换到 Dumpling。

10.1.2.3 全量导入- TiDB Lightning

TiDB Lightning 是一个用于将全量数据导入到TiDB 集群的工具。

使用TiDB Lightning 导入数据到TiDB 时，有以下模式：

- `Physical Import Mode` 模式：TiDB Lightning 将数据解析为有序的键值对，并直接将其导入TiKV。这种模式一般用于导入大量的数据（TB 级别）到新集群，但在数据导入过程中集群无法提供正常的服务。
- `Logical Import Mode` 模式：以TiDB/MySQL 作为后端，这种模式相比 `Physical Import Mode`，导入速度较慢，但是可以在线导入，同时也支持将数据导入到 MySQL。

基本信息：

- TiDB Lightning 的输入：
  - Dumpling 输出文件
  - 其他格式兼容的CSV 文件
  - 从 Aurora 或者 Hive 导出的 Parquet 文件
- 适用TiDB 版本：v2.1 及以上
- Kubernetes 支持：使用TiDB Lightning 快速恢复 Kubernetes 上的TiDB 集群数据

> 注意：
>
> 原 Loader 工具已停止维护，不再推荐使用。相关场景请使用TiDB Lightning 的 `tidb` 模式进行替代。

10.1.2.4 备份和恢复- Backup & Restore

Backup & Restore (BR) 是一个对TiDB 进行分布式备份和恢复的工具，可以高效地对大数据量的TiDB 集群进行数据备份和恢复。

基本信息：

- 备份输出和恢复输入的文件类型
- 适用TiDB 版本：v4.0 及以上
- Kubernetes 支持：使用 BR 工具备份TiDB 集群数据到兼容 S3 的存储，使用 BR 工具恢复 S3 兼容存储上的备份数据

10.1.2.5 TiDB 增量数据同步- TiCDC

TiCDC 是一款通过拉取TiKV 变更日志实现的TiDB 增量数据同步工具，具有将数据还原到与上游任意TSO 一致状态的能力，同时提供开放数据协议(TiCDC Open Protocol)，支持其他系统订阅数据变更。

基本信息：

- TiCDC 的输入：TiDB 集群
- TiCDC 的输出：TiDB 集群、 MySQL、 Kafka、 Confluent
- 适用TiDB 版本：v4.0.6 及以上

10.1.2.6 TiDB 增量日志同步- TiDB Binlog

TiDB Binlog 是收集TiDB 的增量binlog 数据，并提供准实时同步和备份的工具。该工具可用于TiDB 集群间的增量数据同步，如将其中一个TiDB 集群作为另一个TiDB 集群的从集群。

基本信息：

- TiDB Binlog 的输入：TiDB 集群
- TiDB Binlog 的输出：TiDB 集群、 MySQL、 Kafka 或者增量备份文件
- 适用TiDB 版本：v2.1 及以上
- Kubernetes 支持：TiDB Binlog 运维文档，Kubernetes 上的TiDB Binlog Drainer 配置

10.1.2.7 数据校验- sync-diff-inspector

sync-diff-inspector 是一个用于校验 MySQL/TiDB 中两份数据是否一致的工具。该工具还提供了修复数据的功能，可用于修复少量不一致的数据。

基本信息：

- sync-diff-inspector 的输入：TiDB、 MySQL
- sync-diff-inspector 的输出：TiDB、 MySQL
- 适用TiDB 版本：所有版本

10.1.3 OLAP 分析工具- TiSpark

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合TiKV 分布式集群的优势，和TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。

10.2 TiDB 工具的使用场景

本文档从数据迁移工具的使用场景出发，介绍部分常见场景下的迁移工具的选择。

10.2.1 在物理机或虚拟机上部署运维TiDB

当需要在物理机或虚拟机上部署运维TiDB时，你可以先安装TiUP，再通过TiUP管理TiDB的众多组件，如TiDB、PD、TiKV等。

10.2.2 在Kubernetes上部署运维TiDB

当需要在Kubernetes上部署运维TiDB时，你可以先创建Kubernetes集群，部署TiDB Operator，然后使用TiDB Operator部署运维TiDB集群。

10.2.3 从CSV导入数据到TiDB

当需要将其他工具导出的格式兼容的CSV files导入到TiDB时，可使用TiDB Lightning。

10.2.4 从MySQL/Aurora导入全量数据

当需要从MySQL/Aurora导入全量数据时，可先使用Dumpling将数据导出为SQL dump files，然后再使用TiDB Lightning将数据导入到TiDB集群。

10.2.5 从MySQL/Aurora迁移数据

当既需要从MySQL/Aurora导入全量数据，又需要迁移增量数据时，可使用TiDB Data Migration (DM)完成从Amazon Aurora迁移数据到TiDB。

如果全量数据量较大（TB级别），则可先使用Dumpling与TiDB Lightning完成全量数据的迁移，再使用DM完成增量数据的迁移。

10.2.6 TiDB集群备份与恢复

当需要对TiDB集群进行备份或在之后对TiDB集群进行恢复时，可使用BR。

10.2.7 迁出数据到TiDB

当需要将TiDB集群的数据迁出到其他TiDB集群时，可使用Dumpling从TiDB将全量数据导出为SQL dump files，然后再使用TiDB Lightning将数据导入到TiDB。

如果还需要执行增量数据的迁移，则可使用TiCDC。

10.2.8 TiDB增量数据订阅

当需要订阅TiDB 增量数据的变更时，可使用TiCDC。

10.3 TiDB工具下载

本文介绍如何下载TiDB工具包以及TiUniManager。关于TiDB工具包的内容，请查看TiDB离线包。

10.3.1 TiDB 工具包下载

TiDB 工具包中包含了一些常用的TiDB 工具，例如数据导出工具 Dumpling、数据导入工具 TiDB Lightning、备份恢复工具 BR。

> 建议：
>
> 如果你的部署环境能访问互联网，无需单独下载TiDB 工具包，可以直接通过使用TiUP 命令一键部署所需的TiDB 工具。

10.3.1.1 环境要求

- 操作系统：Linux
- 架构：amd64 或 arm64

10.3.1.2 下载步骤

1. 访问TiDB 社区版页面。
2. 找到TiDB-community-toolkit 软件包，点击立即下载。

> 注意：
>
> - 点击立即下载后，默认下载当前TiDB 的最新发布版本。如需下载其它版本，请在TiDB 社区版页面底部查看其它版本下载信息。
> - 如需在 Kubernetes 上部署运维TiDB，无需下载 TiDB-community-toolkit 软件包，请参考离线安装 TiDB Operator。
> - 如需使用 PD Control 工具 `pd-ctl`，请下载 TiDB-community-server 软件包。

10.3.1.3 TiDB 工具包说明

在TiDB 工具包中，你可以依据要使用的工具，选择安装对应的离线包。

工具	离线包名称
TiUP	tiup-linux-{arch}.tar.gz tiup-{tiup-version}-linux-{arch}.tar.gz dm-{tiup-version}-linux-{arch}.tar.gz server-{version}-linux-{arch}.tar.gz
Dumpling	dumpling-{version}-linux-{arch}.tar.gz
TiDB Lightning	tidb-lightning-ctl tidb-lightning-{version}-linux-{arch}.tar.gz

工具	离线包名称
TiDB DM (Data Migration)	dm-worker-{version}-linux-{arch}.tar.gz dm-master-{version}-linux-{arch}.tar.gz dmctl-{version}-linux-{arch}.tar.gz
TiCDC	cdc-{version}-linux-{arch}.tar.gz
TiDB Binlog	pump-{version}-linux-{arch}.tar.gz drainer-{version}-linux-{arch}.tar.gz binlogctl reparo
Backup & Restore (BR)	br-{version}-linux-{arch}.tar.gz
sync-diff-inspector	sync_diff_inspector

工具	离线包名称
TiSpark	tispark-{version}-any-any.tar.gz spark-{spark-version}-any-any.tar.gz
PD Recover	pd-recover-{version}-linux-{arch}.tar.gz

注意：

以上离线包名称中，`{version}` 取决于离线包中工具的版本号，`{arch}` 取决于离线包对应的架构（amd64 或 arm64）。

10.3.2 TiUniManager 下载

TiUniManager 是为TiDB 打造的管控平台软件和数据库运维管理平台。使用下表中的链接下载TiUniManager：

安装包	操作系统	架构	SHA256 校验和
https://download.pingcap.org/em-enterprise-server-{version}-linux-amd64.tar.gz	Linux	amd64	https://download.pingcap.org/em-enterprise-server-{version}-linux-amd64.tar.sha256

注意：

- 下载链接中的 {version} 为 TiUniManager 的版本号。例如，v1.0.2 版本的下载链接为 https://download.pingcap.org/em-enterprise-server-v1.0.2-linux-amd64.tar.gz。
- TiUniManager 从 v1.0.2 起开放源代码， 因此下载链接中 {version} 支持的最低版本为 v1.0.2。你不能将 {version} 替换为 v1.0.0 或 v1.0.1。

10.4 TiUP

10.4.1 TiUP 文档地图

10.4.1.1 使用文档

- TiUP 概览：对TiUP 进行整体介绍，如何安装和基本的用法以及相关术语

- TiUP 术语：解释使用TiUP 过程中可能用到的术语，了解TiUP 的核心概念
- TiUP 组件管理：详细介绍TiUP 所有命令，如何使用TiUP 下载、更新、删除组件
- TiUP FAQ：TiUP 使用过程中的常见问题，包含TiUP 第三方组件的FAQ
- TiUP 故障排查：如果在使用TiUP 过程中遇到问题，可以参考故障排查文档的解决方案
- TiUP 参考手册：TiUP 详细参考手册，包含各类命令、组件、镜像。

10.4.1.2 资源

- AskTUG TiUP 主题
- TiUP Issues：TiUP Github Issues 列表

10.4.2 TiUP 简介

在各种系统软件和应用软件的安装管理中，包管理器均有着广泛的应用，包管理工具的出现大大简化了软件的安装和升级维护工作。例如，几乎所有使用RPM 的 Linux 都会使用yum 来进行包管理，而Anaconda 则可以非常方便地管理 Python 的环境和相关软件包。

在早期的TiDB 生态中，没有专门的包管理工具，使用者只能通过相应的配置文件和文件夹命名来手动管理，如 Prometheus 等第三方监控报表工具甚至需要额外的特殊管理，这样大大提升了运维管理难度。

从 TiDB 4.0 版本开始，TiUP 作为新的工具，承担着包管理器的角色，管理着TiDB 生态下众多的组件，如TiDB、PD、TiKV 等。用户想要运行TiDB 生态中任何组件时，只需要执行TiUP 一行命令即可，相比以前，极大地降低了管理难度。

10.4.2.1 安装TiUP

TiUP 安装过程十分简洁，无论是 Darwin 还是 Linux 操作系统，执行一行命令即可安装成功：

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

该命令将TiUP 安装在 `$HOME/.tiup` 文件夹下，之后安装的组件以及组件运行产生的数据也会放在该文件夹下。同时，它还会自动将 `$HOME/.tiup/bin` 加入到 Shell Profile 文件的 PATH 环境变量中，这样你就可以直接使用 TiUP 了。

例如，你可以查看TiUP 的版本：

```
tiup --version
```

> **注意：**
>
> 对于 v1.11.3 及以上版本的TiUP，默认不会收集使用情况信息分享给 PingCAP。若要了解所收集的信息详情及如何关闭分享行为，请参见遥测。

10.4.2.2 TiUP 生态介绍

TiUP 的直接功能是作为TiDB 生态中的包管理器，但这并不是它的最终使命。TiUP 的愿景是将TiDB 生态中所有工具的使用门槛降到极致，这个仅仅靠包管理功能是做不到的，还需要引入一些额外的包来丰富这个系统，它们一起加入到TiUP 生态中，让TiDB 的世界变得更简单。

TiUP 系列文档的主要内容就是介绍TiUP 及这些包的功能和使用方式。

在 TiUP 生态中，你可以通过在任何命令后加上 `--help` 的方式来获得帮助信息，比如通过以下命令获取TiUP 本身的帮助信息:

```
tiup --help
```

```
TiUP is a command-line component management tool that can help to download and install
TiDB platform components to the local system. You can run a specific version of a component via
"tiup <component>[:version]". If no version number is specified, the latest version installed
locally will be used. If the specified component does not have any version installed locally,
the latest stable version will be downloaded from the repository.

Usage:
  tiup [flags] <command> [args...]
  tiup [flags] <component> [args...]

Available Commands:
  install     Install a specific version of a component
  list        List the available TiDB components or versions
  uninstall   Uninstall components or versions of a component
  update      Update tiup components to the latest version
  status      List the status of instantiated components
  clean       Clean the data of instantiated components
  mirror      Manage a repository mirror for TiUP components
  help        Help about any command or component

Components Manifest:
  use "tiup list" to fetch the latest components manifest

Flags:
      --binary <component>[:version]    Print binary path of a specific version of a component <
          ↪ component>[:version]
                                        and the latest version installed will be selected if no
                                            ↪ version specified
      --binpath string                  Specify the binary path of component instance
  -h, --help                            help for tiup
  -T, --tag string                      Specify a tag for component instance
  -v, --version                         version for tiup

Component instances with the same "tag" will share a data directory ($TIUP_HOME/data/$tag):
  $ tiup --tag mycluster playground
```

```
Examples:
  $ tiup playground                    # Quick start
  $ tiup playground nightly            # Start a playground with the latest nightly version
  $ tiup install <component>[:version] # Install a component of specific version
  $ tiup update --all                  # Update all installed components to the latest version
  $ tiup update --nightly              # Update all installed components to the nightly version
  $ tiup update --self                 # Update the "tiup" to the latest version
  $ tiup list                          # Fetch the latest supported components list
  $ tiup status                        # Display all running/terminated instances
  $ tiup clean <name>                  # Clean the data of running/terminated instance (Kill
      ↪ process if it's running)
  $ tiup clean --all
                                       # Clean the data of all running/terminated instances

Use "tiup [command] --help" for more information about a command.
```

输出的帮助信息较长，你可以只关注两部分：

- 可用的命令
  - install：用于安装组件
  - list：查看可用组件列表
  - uninstall：卸载组件
  - update：更新组件版本
  - status：查看组件运行记录
  - clean：清除组件运行记录
  - mirror：从官方镜像克隆一个私有镜像
  - help：输出帮助信息
- 可用的组件
  - playground：在本机启动集群
  - client：连接本机的集群
  - cluster：部署用于生产环境的集群
  - bench：对数据库进行压力测试

> 注意：
>
> - 可用的组件会持续增加，以 `tiup list` 输出结果为准。
> - 组件的可用版本列表也会持续增加，以 `tiup list <component>` 输出结果为准。

命令和组件的区别在于，命令是TiUP 自带的，用于进行包管理的操作。而组件是TiUP 通过包管理操作安装的独立组件包。比如执行 `tiup list` 命令，TiUP 会直接运行自己内部的代码，而执行 `tiup playground` 命令则会先检查本地有没有叫做playground 的组件包，若没有则先从镜像上下载过来，然后运行这个组件包。

10.4.3 TiUP 术语及核心概念

本文主要说明TiUP 的重要术语和核心概念。

10.4.3.1 TiUP 组件

TiUP 程序只包含少数几个命令，用来下载、更新、卸载组件。TiUP 通过各种组件来扩展其功能。组件是一个可以运行的程序或脚本，通过 `tiup <component>` 运行组件时，TiUP 会添加一组环境变量，并为该程序创建好对应的数据目录，然后运行该程序。

通过运行 `tiup <component>` 命令，你可以运行支持的TiUP 组件，其中运行的逻辑为：

1. 如果用户通过 `tiup <component>[:version]` 运行指定某个组件的特定版本：
    - 组件在本地未安装任何版本，则从镜像服务器下载最新稳定版本
    - 组件在本地安装有其他版本，但是没有用户指定的版本，则从镜像服务器下载用户指定版本
    - 如果本地已经安装指定版本，则设置环境变量来运行已经安装的版本
2. 如果用户通过 `tiup <component>` 运行某个组件，且未指定任何版本：
    - 组件在本地未安装任何版本，则从镜像服务器下载最新稳定版本
    - 如果本地已经安装部分版本，则设置环境变量来运行已经安装的版本中的最新版本

10.4.3.2 TiUP 镜像仓库

TiUP 的所有组件都从镜像仓库(mirrors) 下载，镜像仓库包含各个组件的TAR 包以及对应的元信息（版本、入口启动文件、校验和）。TiUP 默认使用 PingCAP 官方的镜像仓库。用户可以通过 `TIUP_MIRRORS` 环境变量自定义镜像仓库。

镜像仓库可以是本地文件目录或在线 HTTP 服务器：

```
1. TIUP_MIRRORS=/path/to/local tiup list
2. TIUP_MIRRORS=https://private-mirrors.example.com tiup list
```

10.4.4 使用TiUP 命令管理组件

TiUP 主要通过以下一些命令来管理组件：

- list：查询组件列表，用于了解可以安装哪些组件，以及这些组件可选哪些版本
- install：安装某个组件的特定版本
- update：升级某个组件到最新的版本
- uninstall：卸载组件
- status：查看组件运行状态
- clean：清理组件实例
- help：打印帮助信息，后面跟其他TiUP 命令则是打印该命令的使用方法

本文介绍常用的组件管理操作及相应命令。

10.4.4.1 查询组件列表

你可以使用 `tiup list` 命令来查询组件列表。该命令用法如下：

- `tiup list`：查看当前有哪些组件可以安装
- `tiup list ${component}`：查看某个组件有哪些版本可以安装

你也可以在命令中组合使用以下参数(flag)：

- `--installed`：查看本地已经安装了哪些组件，或者已经安装了某个组件的哪些版本
- `--all`：显式隐藏的组件
- `--verbose`：显式所有列（安装的版本、支持的平台）

示例一：查看当前已经安装的所有组件

```
tiup list --installed
```

示例二：从服务器获取TiKV 所有可安装版本组件列表

```
tiup list tikv
```

10.4.4.2 安装组件

你可以使用 `tiup install` 命令来安装组件。该命令的用法如下：

- `tiup install <component>`：安装指定组件的最新稳定版
- `tiup install <component>:[version]`：安装指定组件的指定版本

示例一：使用TiUP 安装最新稳定版的TiDB

```
tiup install tidb
```

示例二：使用TiUP 安装 nightly 版本的TiDB

```
tiup install tidb:nightly
```

示例三：使用TiUP 安装 v7.0.0 版本的TiKV

```
tiup install tikv:v7.0.0
```

10.4.4.3 升级组件

在官方组件提供了新版之后，你可以使用 `tiup update` 命令来升级组件。除了以下几个参数，该命令的用法基本和 `tiup install` 相同：

- `--all`：升级所有组件
- `--nightly`：升级至 nightly 版本

- `--self`：升级 TiUP 自己至最新版本
- `--force`：强制升级至最新版本

示例一：升级所有组件至最新版本

```
tiup update --all
```

示例二：升级所有组件至nightly 版本

```
tiup update --all --nightly
```

示例三：升级TiUP 至最新版本

```
tiup update --self
```

10.4.4.4 运行组件

安装完成之后，你可以使用 `tiup <component>` 命令来启动相应的组件：

```
tiup [flags] <component>[:version] [args...]

Flags:
  -T, --tag string                    为组件实例指定 tag
```

该命令需要提供一个组件的名字以及可选的版本，若不提供版本，则使用该组件已安装的最新稳定版。

在组件启动之前，TiUP 会先为它创建一个目录，然后将组件放到该目录中运行。组件会将所有数据生成在该目录中，目录的名字就是该组件运行时指定的tag 名称。如果不指定 tag，则会随机生成一个tag 名称，并且在实例终止时自动删除工作目录。

如果想要多次启动同一个组件并复用之前的工作目录，就可以在启动时用 `--tag` 指定相同的名字。指定tag 后，在实例终止时就不会自动删除工作目录，方便下次启动时复用。

示例一：运行 v7.0.0 版本的TiDB

```
tiup tidb:v7.0.0
```

示例二：指定tag 运行TiKV

```
tiup --tag=experiment tikv
```

10.4.4.4.1 查询组件运行状态

你可以使用 `tiup status` 命令来查看组件的运行状态：

```
tiup status
```

运行该命令会得到一个实例列表，每行一个实例。列表中包含这些列：

- Name：实例的 tag 名称

- Component：实例的组件名称
- PID：实例运行的进程 ID
- Status：实例状态，RUNNING 表示正在运行，TERM 表示已经终止
- Created Time：实例的启动时间
- Directory：实例的工作目录，可以通过 --tag 指定
- Binary：实例的可执行程序，可以通过 --binpath 指定
- Args：实例的运行参数

10.4.4.4.2 清理组件实例

你可以使用 tiup clean 命令来清理组件实例，并删除工作目录。如果在清理之前实例还在运行，会先 kill 相关进程。该命令用法如下：

```
tiup clean [tag] [flags]
```

支持以下参数：

- --all：清除所有的实例信息

其中tag表示要清理的实例tag，如果使用了 --all 则不传递tag。

示例一：清理tag 名称为 experiment 的组件实例

```
tiup clean experiment
```

示例二：清理所有组件实例

```
tiup clean --all
```

10.4.4.4.3 卸载组件

TiUP 安装的组件会占用本地磁盘空间，如果不想保留过多老版本的组件，可以先查看当前安装了哪些版本的组件，然后再卸载某个组件。

你可以使用 tiup uninstall 命令来卸载某个组件的所有版本或者特定版本，也支持卸载所有组件。该命令用法如下：

```
tiup uninstall [component][:version] [flags]
```

支持的参数：

- --all：卸载所有的组件或版本
- --self：卸载 TiUP 自身

component 为要卸载的组件名称，version 为要卸载的版本，这两个都可以省略，省略任何一个都需要加上 --all 参数：

- 若省略版本，加 --all 表示卸载该组件所有版本

- 若版本和组件都省略，则加 `--all` 表示卸载所有组件及其所有版本

示例一：卸载 v7.0.0 版本的TiDB

```
tiup uninstall tidb:v7.0.0
```

示例二：卸载所有版本的TiKV

```
tiup uninstall tikv --all
```

示例三：卸载所有已经安装的组件

```
tiup uninstall --all
```

10.4.5 TiUP FAQ

10.4.5.1 TiUP 是否可以不使用官方镜像源？

TiUP 支持通过环境变量 `TIUP_MIRRORS` 指定镜像源，镜像源的地址可以是一个本地目录或 HTTP 服务器地址。如果用户的环境不能访问网络，可以建立自己的离线镜像源使用TiUP。

如果在使用非官方镜像之后想要切回官方镜像可以采取以下任一措施：

- 将 `TIUP_MIRRORS` 变量设置成官方镜像的地址：`https://tiup-mirrors.pingcap.com`。
- 先确保 `TIUP_MIRRORS` 变量没有设置，再使用执行 `tiup mirror set https://tiup-mirrors.pingcap.com` 命令。

10.4.5.2 如何将自己编写的组件放入TiUP 镜像仓库？

TiUP 暂时不支持外部开发的组件，但是TiUP Team 已经制定了TiUP 组件开发规范，同时正在开发tiup-publish 组件，完成tiup-publish 组件后，开发者可以通过 `tiup publish <comp> <version>` 将自己开发的组件发布到TiUP 的官方镜像仓库。

10.4.5.3 tiup-playground 和 tiup-cluster 有什么区别？

TiUP Playground 组件主要定位是快速上手和搭建单机的开发环境，支持 Linux/macOS，要运行一个指定版本的 TiUP 集群更加简单。TiUP Cluster 组件主要是部署生产环境集群，通常是一个大规模的集群，还包含运维相关操作。

10.4.5.4 怎么样编写tiup-cluster 组件的拓扑文件？

可以参考拓扑文件的样例，样例中包含了：

1. 两地三中心
2. 最小部署拓扑
3. 完整拓扑文件

可以根据自己的需求选择不同的模板，进行编辑。

10.4.5.5 同一个主机是否可以部署多个实例？

同一个主机可以使用TiUP Cluster 部署多个实例，但是需要配置不同的端口和目录信息，否则可能导致目录以及端口冲突。

10.4.5.6 是否可以检测同一个集群内的端口和目录冲突？

同一个集群的端口和目录冲突会在部署和扩容的时候进行检测，如果有目录和端口冲突，本次部署或扩容会中断。

10.4.5.7 是否可以检测不同集群的端口和目录冲突？

如果不同集群是由同一个TiUP 中控机部署的，会在部署和扩容时进行检测，如果属于不同的TiUP 中控机，目前不支持检测。

10.4.5.8 集群部署期间，TiUP 收到报错`ssh: handshake failed: read tcp 10.10.10.34:38980 -> 10.10.10.34:3600: read: connection reset by peer`

该报错可能是因为TiUP 默认并发超过 ssh 默认最大连接数导致，可尝试加大 ssh 默认连接数，然后重启 sshd 服务解决：

```
vi /etc/ssh/sshd_config
MaxSessions 1000
MaxStartups 1000
```

10.4.6 TiUP 故障排查

本文介绍TiUP 使用过程中一些常见的故障及排查方式，如果本文不包含你目前遇到的问题，可以通过以下方式求助：

1. Github Issues 新建一个Issue。
2. 在 AskTUG 提交你的问题。

10.4.6.1 1. TiUP 命令故障排查

10.4.6.1.1 1.1 使用 `tiup list` 看不到最新的组件列表

TiUP 并不会每次都从镜像服务器更新最新的组件列表，可以通过 `tiup list` 来强制刷新组件列表。

10.4.6.1.2 1.2 使用 `tiup list <component>` 看不到一个组件的最新版本信息

同1.1 一样，组件的版本信息只会在本地无缓存的情况下从镜像服务器获取，可以通过`tiup list <component>` 刷新组件列表。

10.4.6.1.3 1.3 下载组件的过程中中断

如果下载组件的过程中网络中断，可能是由于网络不稳定导致的，可以尝试重新下载，如果多次不能成功下载，请反馈到Github Issues，可能是由于 CDN 服务器导致的。

10.4.6.1.4 1.4 下载组件过程中出现checksum 错误

由于 CDN 会有短暂的缓存时间，导致新的checksum 文件和组件包不匹配，建议过 5 分钟后重试，如果依然不匹配，请反馈到Github Issues。

10.4.6.2 2. TiUP Cluster 组件故障排查

10.4.6.2.1 2.1 部署过程中提示 `unable to authenticate, attempted methods [none publickey]`

由于部署时会向远程主机上传组件包，以及进行初始化，这个过程需要连接到远程主机，该错误是由于找不到连接到远程主机的 SSH 私钥导致的。请确认你是否通过 `tiup cluster deploy -i identity_file` 指定该私钥。

1. 如果没有指定 `-i` 参数，可能是由于TiUP 没有自动找到私钥路径，建议通过 `-i` 显式指定私钥路径。
2. 如果指定了 `-i` 参数，可能是由于指定的私钥不能登录，可以通过手动执行 `ssh -i identity_file ′→ user@remote` 命令来验证。
3. 如果是通过密码登录远程主机，请确保指定了 `-p` 参数，同时输入了正确的登录密码。

10.4.6.2.2 2.2 使用TiUP Cluster 升级中断

为了避免用户误用，TiUP Cluster 不支持指定部分节点升级，所以升级失败之后，需要重新进行升级操作，包括升级过程中的幂等操作。

升级操作会分为以下几步：

1. 首先备份所有节点的老版本组件
2. 分发新的组件到远程
3. 滚动重启所有组件

如果升级操作在滚动重启时中断，可以不用重复进行 `tiup cluster upgrade` 操作，而是通过 `tiup cluster ′→ restart -N <node1> -N <node2>` 来重启未完成重启的节点。如果同一组件的未重启节点数量比较多，也可以通过 `tiup cluster restart -R <component>` 来重启某一个类型的组件。

10.4.6.2.3 2.3 升级发现 `node_exporter-9100.service/blackbox_exporter-9115.service` 不存在

这种情况可能是由于之前的集群是由TiDB Ansible 迁移过来的，且之前TiDB Ansible 未部署 exporter 导致的。要解决这种情况，可以暂时通过手动从其他节点复制缺少的文件到新的节点。后续我们会在迁移过程中补全缺失的组件。

10.4.7 TiUP 命令参考手册

10.4.7.1 TiUP 命令概览

TiUP 在TiDB 生态中承担包管理器的功能，管理着TiDB 生态下众多的组件，如TiDB、PD、TiKV 等。

10.4.7.1.1 语法

```
tiup [flags] <command> [args...]          # 执行命令
#### or
tiup [flags] <component> [args...]        # 运行组件
```

使用 `--help` 命令可以获取特定命令的信息，每个命令的摘要都显示了其参数及其用法。必须参数显示在尖括号中，可选参数显示在方括号中。

`<command>` 代表命令名字，支持的命令列表请参考下方命令清单，`<component>` 代表组件名，支持的组件列表请参考下方组件清单。

10.4.7.1.2 选项

–binary

打印指定组件的二进制文件路径：

- 执行 `tiup --binary <component>` 将打印已安装的 `<component>` 组件的最新稳定版路径，若 `<component>` 组件未安装，则报错
- 执行 `tiup --binary <component>:<version>` 将打印已经安装的 `<component>` 组件的 `<version>` 版本所在的路径，若该版本未安装，则报错
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> 注意：
>
> 该选项只能用于 `tiup [flags] <component> [args...]` 格式的命令。

–binpath (string)

指定要执行的组件的路径：执行一个组件时，如果不想使用TiUP 镜像中的二进制文件，可以使用该参数使用自定义路径的二进制文件替换之。

> 注意：
>
> 该选项只能用于 `tiup [flags] <component> [args...]` 格式的命令。

-T, –tag (string)

对启动的组件指定一个 `tag`：有的组件在执行过程中需要使用磁盘存储，TiUP 会分配一个临时目录作为该组件本次执行的存储目录，如果希望分配固定目录，可以用 `-T/--tag` 来指定目录名字，这样多次执行使用同样的 `tag` 就能读写到同一批文件。

-v, –version

打印TiUP 的版本

–help

打印帮助信息

10.4.7.1.3 命令清单

TiUP 包含众多的命令，这些命令又包含了许多子命令，具体命令及其子命令的说明请参考对应的链接：

- install：安装组件
- list：查看组件列表
- uninstall：卸载组件
- update：升级已安装的组件
- status：查看组件运行状态
- clean：清理组件数据目录
- mirror：镜像管理
- telemetry：遥测开关
- completion：TiUP 命令补全
- env：查看 TiUP 相关环境变量
- help：查看特定命令或组件的帮助文档

10.4.7.1.4 组件清单

- cluster：生产环境TiDB 集群管理
- dm：生产环境 DM 集群管理

10.4.7.2 TiUP 命令

10.4.7.2.1 tiup clean

命令 `tiup clean` 用于清除组件运行过程中产生的数据。

语法

```
tiup clean [name] [flags]
```

`[name]` 取值为status 命令输出的 `Name` 字段。若省略 `[name]`，则必须配合 `--all` 使用。

选项

–all

- 清除所有运行记录。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

```
Clean instance of `%s`, directory: %s
```

<< 返回上一页 - TiUP 命令清单

10.4.7.2.2 tiup completion

为了降低使用成本，TiUP 提供了 `tiup completion` 命令用于生成命令行自动补全的配置文件。目前支持 bash 和 zsh 两种 shell 的命令补全。

如果是bash，需要提前安装好bash-completion：

- 在 macOS 上的安装方式为：如果bash 版本小于 4.1，执行 `brew install bash-completion`；如果 bash 版本大于等于 4.1，则执行 `brew install bash-completion@2`。
- 在 Linux 上的安装方式为： 使用包管理器安装 `bash-completion` 包， 例如执行 `yum install bash-completion` 或者 `apt install bash-completion`。

语法

```
tiup completion <shell>
```

<shell> 为 shell 类型，目前支持 bash 和 zsh。

使用方式

bash

将自动补全代码写入一个文件，并且在 .bash_profile 中对其执行 source 命令：

```
tiup completion bash > ~/.tiup.completion.bash

printf "
##### tiup shell completion
source '$HOME/.tiup.completion.bash'
" >> $HOME/.bash_profile

source $HOME/.bash_profile
```

zsh

```
tiup completion zsh > "${fpath[1]}/_tiup"
```

<< 返回上一页 - TiUP 命令清单

10.4.7.2.3 tiup env

TiUP 为用户提供了灵活的定制化接口，其中一部分是使用环境变量来实现的，命令 `tiup env` 用于查询TiUP 支持用户自定义的环境变量以及它们此时的值。

语法

```
tiup env [name1...N]
```

[name1...N] 用于查看指定的环境变量，若不指定，则默认查看所有支持的环境变量。

选项

无

输出

- 若未指定 [name1...N]，则输出 “{key}”=“{value}” 列表
- 若指定了 [name1...N]，则按顺序输出 “{value}” 列表

以上输出中若 value 为空则代表未设置环境变量的值，此时TiUP 会使用默认值。

<< 返回上一页 - TiUP 命令清单

10.4.7.2.4 tiup help

TiUP 命令行界面为用户提供了丰富的帮助信息，用户可以通过 help 命令或者 --help 参数查看。tiup help <
↪ command> 命令等价于 tiup <command> --help。

语法

```
tiup help [command]
```

[command] 用于指定要查看哪个命令的帮助信息，若不指定，则查看TiUP 自身的帮助信息。

选项

无

输出

[command] 或TiUP 的帮助信息。

<< 返回上一页 - TiUP 命令清单

10.4.7.2.5 tiup install

命令 tiup install 用于组件安装，它会从镜像仓库中下载指定版本的组件包，并在本地的 TiUP 数据目录中解压，以便后续使用。另外，当TiUP 需要运行一个镜像仓库中不存在的组件时，会尝试先下载该组件，再自动运行，若仓库中不存在会报错。

语法

```
tiup install <component1>[:version] [component2...N] [flags]
```

<component1> 和 <component2> 代表组件名字，[version] 代表一个可选的版本号，若不加 version，则安装指定组件的最新稳定版本。[component2...N] 表示可同时指定多个组件或同一个组件的多个版本。

选项

无

输出

- 正常情况下输出组件的下载信息
- 若组件不存在则报错 `The component "%s" not found`
- 若版本不存在则报错 `version %s not supported by component %s`

<< 返回上一页 - TiUP 命令清单

10.4.7.2.6 tiup list

命令 `tiup list` 用于查询镜像中可用的组件列表。

语法

```
tiup list [component] [flags]
```

`[component]` 是可选的组件名称。若指定，则列出该组件的所有版本；若不指定，则列出所有组件列表。

选项

–all

- 显示所有组件。默认只显示非隐藏组件。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–installed

- 只显示已经安装的组件或版本。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–verbose

- 在组件列表中显示已安装的版本列表。默认组件列表不显示当前已安装的版本。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若未指定 `[component]`
  - 若指定–verbose：输出组件名 (Name)、已安装版本 (Installed)、组件管理员 (Owner)、组件描述 (Description) 构成的组件信息列表
  - 若不指定–verbose：输出 组件名 (Name)、组件管理员 (Owner)、组件描述 (Description) 构成的组件信息列表
- 若指定 `[component]`
  - 若 `[component]` 存在：输出版本 (Version)、是否已安装 (Installed)、发布时间 (Release)、支持的平台 (Platforms) 构成的版本信息列表
  - 若 `[component]` 不存在：报错 `failed to fetch component: unknown component`

<< 返回上一页 - TiUP 命令清单

10.4.7.2.7 tiup mirror

tiup mirror

在TiUP中，镜像是一个非常重要的概念，目前TiUP支持两种形式的镜像：

- 本地镜像：即TiUP 客户端和镜像在同一台机器上，客户端通过文件系统访问镜像
- 远程镜像：即TiUP 客户端和镜像不在同一台机器上，客户端通过网络访问镜像

命令 `tiup mirror` 用于管理镜像，提供了创建镜像，组件发布，密钥管理等多种功能。

语法

```
tiup mirror <command> [flags]
```

<command> 代表子命令，支持的子命令列表请参考下方命令清单。

选项

无

命令清单

- genkey: 生成私钥文件
- sign: 使用私钥文件对特定文件进行签名
- init: 创建一个空的镜像
- set: 设置当前镜像
- grant: 为当前镜像引入新的组件管理员
- publish: 向当前镜像推送新的组件
- modify: 修改当前镜像中的组件属性
- rotate: 更新当前镜像中的根证书
- clone: 从已有镜像克隆一个新的镜像
- merge: 合并镜像

<< 返回上一页 - TiUP 命令清单

tiup mirror clone

命令 `tiup mirror clone` 用于克隆一个已经存在的镜像或克隆部分组件生成一个新的镜像。新旧镜像的组件相同，但使用的签名密钥不同。

语法

```
tiup mirror clone <target-dir> [global version] [flags]
```

- <target-dir> 是本地存放克隆下来的镜像的路径，如果不存在则会自动创建。
- 若指定了 [global version] 参数，TiUP 会尝试克隆指定版本的所有组件。若某些组件没有指定的版本，则克隆其最新版本。

选项

-f, –full

- 是否克隆整个镜像。指定该选项后会从目标镜像完整克隆所有组件的所有版本，此时其他指定的选项将失效。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-a, –arch

- 仅克隆能在指定平台上运行的组件。
- 数据类型：STRINGS
- 该参数接受以逗号分隔的多个平台名称，例如 `amd64,arm64`。如果未指定该选项，默认克隆 AMD64 和 ARM64 平台的组件，即 `amd64,arm64`。

-o, –os

- 仅克隆能在指定操作系统上运行的组件。
- 数据类型：STRINGS
- 该参数接受以逗号分隔的多个操作系统名称，例如 `linux,darwin`。如果未指定该选项，默认克隆 Linux 和 Darwin 系统的组件，即 `linux,darwin`。

–prefix

- 匹配版本时是否前缀匹配。默认情况下必须严格匹配指定的版本才会下载，指定该选项之后，仅前缀匹配指定的版本也会被下载。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–{component}（strings，默认为空）

指定要克隆的 `{component}` 组件的版本列表。`{component}` 为组件名，可选的组件名可执行`tiup list --all` 查看。

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror genkey

在 TiUP 镜像的定义中，有三类角色：

- 镜像管理员：拥有 `root.json`、`index.json`、`snapshot.json` 以及 `timestamp.json` 的修改权限
- 组件管理员：拥有相关组件的修改权限
- 普通用户：可以下载并使用组件

由于修改文件需要相关的管理员进行签名，因此管理员必须拥有自己的私钥。命令 `tiup mirror genkey` 就是用于生成私钥的。

> 警告：
>
> 请勿通过网络传输私钥。

语法

```
tiup mirror genkey [flags]
```

选项

-n, –name

- 密钥的名字，该名字决定最终生成的文件名。生成的私钥文件路径为：`${TIUP_HOME}/keys/{name}.json`，其中 `TIUP_HOME` 为 TiUP 的 Home 目录，默认路径为 `$HOME/.tiup`，`name` 为 `-n/--name` 指定的密钥名字。
- 数据类型：`STRING`
- 如果不指定该选项，密钥名默认为 `private`。

-p, –public

- 显示当前私钥对应的公钥，当前私钥名字由 `-n/--name` 选项指定。
- 当指定了 `-p/--public` 时，不会创建新的私钥。若 `-n/--name` 指定的私钥不存在，则报错。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–save

- 将公钥信息储存为文件放置于当前目录，文件名称为 `{hash-prefix}-public.json`，其中 `hash-prefix` 为该密钥ID 的前 16 位。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若未指定 `-p/--public`：
  - 若指定的密钥已存在：`Key already exists, skipped`
  - 若指定的密钥不存在：`private key have been write to ${TIUP_HOME}/keys/{name}.json`
- 若指定 `-p/--public`：
  - 若指定的密钥不存在：`Error: open ${TIUP_HOME}/keys/{name}.json: no such file or directory`
  - 若指定的密钥存在：输出该密钥对应的公钥内容

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror grant

命令 `tiup mirror grant` 用来向当前镜像中引入一个组件管理员。

组件管理员可以使用其密钥发布新的组件，也可以对其之前发布的组件作出修改。添加组件管理员时，待添加的组件管理员需要先将其公钥发送给镜像管理员。

> **注意：**
>
> 该命令仅支持在当前镜像为本地镜像时使用。

语法

```
tiup mirror grant <id> [flags]
```

<id> 为该组件管理员的ID，该ID 需要在整个镜像中唯一，建议使用符合正则 ^[a-z\d](?:[a-z\d]|-(?=[a-z\d])){0,38}$ 的字符串。

选项

-k, –key（string，默认 ${TIUP_HOME}/keys/private.json）

指定引入的组件管理员的密钥。该密钥可以是公钥也可以是私钥。如果传入私钥，会被转换成对应的公钥储存在镜像中。

一个密钥只能被一个组件管理员使用。

-n, –name（string，默认 <id>）

指定组件管理员的名字，该名字会展示在组件列表的 Owner 字段上。若未指定 -n/--name 则使用 <id> 作为组件管理员名字。

输出

- 若执行成功：无输出
- 若管理员ID 重复：Error: owner %s exists
- 若密钥已被其他管理员使用：Error: key %s exists

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror init

命令 tiup mirror init 用于初始化一个空的镜像。初始化的镜像不包含任何组件和组件管理员，仅生成以下文件：

```
+ <mirror-dir>                                  # 镜像根目录
|-- root.json                                   # 镜像根证书
|-- 1.index.json                                # 组件/用户索引
|-- snapshot.json                               # 镜像最新快照
|-- timestamp.json                              # 镜像最新时间戳
|--+ keys                                       # 镜像私钥（可移动到其他位置）
   |-- {hash1..hashN}-root.json                 # 根证书私钥
   |-- {hash}-index.json                        # 索引私钥
   |-- {hash}-snapshot.json                     # 快照私钥
   |-- {hash}-timestamp.json                    # 时间戳私钥
```

以上文件的具体作用及内容格式请参考镜像说明。

语法

```
tiup mirror init <path> [flags]
```

<path> 为本地目录路径，可以为相对路径。TiUP 会以此路径为镜像文件存放路径，在其中生成文件。若该目录已存在，则必须保证为空，若该目录不存在，则TiUP 会自动创建。

选项

-k, –key-dir（string，默认 {path}/keys）

指定生成私钥文件的目录。若指定的文件目录不存在，则会自动创建。

输出

- 若成功：无输出
- 若 <path> 不为空：`Error: the target path '%s' is not an empty directory`
- 若 <path> 不是目录：`Error: fdopendir: not a directory`

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror merge

命令 `tiup mirror merge` 用于合并一个或多个镜像到当前镜像。

执行此命令需要满足几个条件：

- 目标镜像的所有组件的管理员ID 必须在当前镜像中存在。
- 执行该命令用户的 `${TIUP_HOME}/keys` 目录中有上述管理员ID 在当前镜像中对应的所有私钥（可以使用命令`tiup mirror set` 将当前镜像切换成目前有权限修改的镜像）。

语法

```
tiup mirror merge <mirror-dir-1> [mirror-dir-N] [flags]
```

- `<mirror-dir-1>`：要合并到当前镜像的第一个镜像
- `[mirror-dir-N]`：要合并到当前镜像的第 N 个镜像

选项

无

输出

- 成功：无输出
- 当前镜像缺失目标镜像某个组件的管理员， 或 `${TIUP_HOME}/keys` 缺失该管理员的私钥：`Error: missing owner keys for owner %s on component %s`

tiup mirror modify

命令 `tiup mirror modify` 用于修改已发布的组件。只有合法的组件管理员才可以修改组件，且只能修改其自己发布的组件。组件发布方式参考 `publish` 命令。

语法

```
tiup mirror modify <component>[:version] [flags]
```

各个参数解释如下：

- `<component>`：组件名称
- `[version]`：想要修改的版本，若不指定，则表示修改整个组件

选项

-k, –key（string，默认 ${TIUP_HOME}/keys/private.json）

组件管理员的私钥，客户端需要使用该私钥对组件信息(`{component}.json`) 进行签名。

–yank

- 将指定组件或指定版本标记为不可用：
  - 标记组件不可用之后 `tiup list` 将看不到该组件，也无法安装该组件的新版本。
  - 标记版本不可用之后 `tiup list <component>` 将看不到该版本，也无法安装该版本。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–hide

- 将该组件设置为隐藏，隐藏之后该组件将不在 `tiup list` 的列表中显示，但是可通过执行 `tiup list --all` 查看。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> 注意：
> 该选项只能应用于组件上，无法应用于组件的版本上。

输出

- 若成功：无输出
- 若该组件管理员无权修改目标组件：

- 若使用远程镜像：`Error: The server refused, make sure you have access to this component`
- 若使用本地镜像：`Error: the signature is not correct`

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror publish

命令 `tiup mirror publish` 用于发布新组件，或已有组件的新版本。只有有权限的组件管理员才可以发布组件。引入组件管理员的方式可参考grant 命令。

语法

```
tiup mirror publish <comp-name> <version> <tarball> <entry> [flags]
```

各个参数解释如下：

- `<comp-name>`：组件名，如 `tidb`，建议使用符合正则 `^[a-z\d](?:[a-z\d]|-(?=[a-z\d])){0,38}$` 的字符串
- `<version>`：当前正在发布的版本，版本号需要符合 Semantic Versioning
- `<tarball>`：`.tar.gz` 包的本地路径，需要将组件的可执行文件及依赖放在该包中，由TiUP 上传到镜像
- `<entry>`：组件的可执行文件在 `<tarball>` 中的位置

选项

-k, –key（string，默认 ${TIUP_HOME}/keys/private.json）

组件管理员的私钥，客户端需要使用该私钥对组件信息({component}.json) 进行签名。

–arch（string，默认 ${GOARCH}）

该 `<tarlball>` 中的二进制文件运行的平台，一个 `<tarball>` 只能选以下三个平台之一：

- amd64：表示在 amd64 架构的机器上运行
- arm64：表示在 arm64 架构的机器上运行
- any：表示可以在以上两种架构的机器上运行（比如脚本）

> 注意：
> 若 `--arch` 指定为 any，则 `--os` 也必须指定为 any。

–os（string，默认 ${GOOS}）

该 `<tarlball>` 中的二进制文件运行的操作系统，一个 `<tarball>` 只能选以下三个操作系统之一：

- linux：表示在 Linux 操作系统上运行
- darwin：表示在 Darwin 操作系统上运行
- any：表示可以在以上两种操作系统上运行（比如脚本）

> **注意：**
>
> 若 `--os` 指定为 any，则 `--arch` 也必须指定为 any。

–desc（string，默认为空）

该组件的描述信息。

–hide

- 是否为隐藏组件。若为隐藏组件，则不在 `tiup list` 的列表中显示，但在 `tiup list --all` 的列表中会显示。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若成功：无输出
- 若该组件管理员无权修改目标组件：
  - 若使用远程镜像：`Error: The server refused, make sure you have access to this component`
  - 若使用本地镜像：`Error: the signature is not correct`

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror rotate

TiUP 的镜像中有一个非常重要的文件：root.json，里面记录了整个系统需要使用的公钥，是TiUP 信任链的基础，它的内容主要包含几个部分：

- N 个管理员的签名，对于官方镜像，N 为 5，默认初始化的镜像N 为 3
- 用于验证以下文件的公钥：
  - root.json
  - index.json
  - snapshot.json
  - timestamp.json
- 过期时间，对于官方镜像，为root.json 创建时间后延一年

关于镜像的详细介绍可以参考镜像说明。

在某些情况下，用户需要更新root.json:

- 更换镜像的密钥
- 更新证书过期时间

更新root.json 内容之后，必须由管理员对其进行重新签名，否则客户端会拒绝，更新流程如下：

1. 更新root.json的内容
2. N 个管理员对新的root.json进行签名
3. 更新 snapshot.json，记录新的 root.json 的 version
4. 对新的 snapshot.json 进行签名
5. 更新timestamp.json，记录新的 snapshot.json 的 hash
6. 对新的timestamp.json进行签名

TiUP 使用命令 `tiup mirror rotate` 来自动化以上流程。

> 注意：
>
> - 经测试，小于TiUP v1.5.0 的版本无法正确获得新的root.json #983。
> - 使用此功能前请确保所有的TiUP 客户端升级到了 v1.5.0 或以上版本。

语法

```
tiup mirror rotate [flags]
```

该命令会启动一个编辑器，修改其内容为目标值（比如将 `expires` 字段的值向后推移），然后需要将 `version` 字段加一并保存。保存之后会启动一个临时的http 服务器，等待N 个不同的镜像管理员签名。

镜像管理员签名的方式参考sign 命令。

选项

–addr（string，默认 0.0.0.0:8080）

临时服务器的监听地址，需要确保该地址可以被其他镜像管理员访问，这样管理员才能使用sign 命令签名。

输出

- 各个镜像管理员当前的签名状态

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror set

命令 `tiup mirror set` 用于切换当前镜像，支持本地文件系统和远程网络两种镜像。

官方镜像为 `https://tiup-mirrors.pingcap.com`。

语法

```
tiup mirror set <mirror-addr> [flags]
```

`<mirror-addr>` 为镜像地址，可以有两种形式：

- 网络地址：以http 或者https 开头，如 `http://172.16.5.5:8080`，`https://tiup-mirrors.pingcap.com` 等
- 本地文件路径：镜像目录的绝对路径，比如 `/path/to/local-tiup-mirror`

选项

-r, –root（string，默认 `{mirror-dir}/root.json`）

指定根证书。

每个镜像的根证书不相同，而根证书是镜像安全性最关键的一环，在使用网络镜像时，可能遭受中间人攻击，为了避免此类攻击，推荐手动将根网络镜像的根证书下载到本地：

```
wget <mirror-addr>/root.json -O /path/to/local/root.json
```

然后进行人工查验，认定无误之后，再通过手工指定根证书的方式切换镜像：

```
tiup mirror set <mirror-addr> -r /path/to/local/root.json
```

在这种操作方式下，如果中间人在 `wget` 之前攻击了镜像，用户可发现根证书不正确。如果在 `wget` 之后攻击了镜像，TiUP 会发现镜像和根证书不符。

输出

无

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror sign

命令 `tiup mirror sign` 用于对镜像中定义的元信息文件（*.json）进行签名，这些文件可能储存在本地文件系统，也可以放在远端使用http 协议提供签名入口。

语法

```
tiup mirror sign <manifest-file> [flags]
```

`<manifest-file>` 为被签名的文件地址，可以有两种地址：

- 网络地址：http 或者https 开头，如 `http://172.16.5.5:8080/rotate/root.json`
- 本地文件路径：相对路径或绝对路径均可

如果是网络地址，该地址必须提供以下功能：

- 支持以 `http get` 访问，此时应当返回被签名文件的完整内容（包含 signatures 字段）
- 支持以 `http post` 访问，客户端会在 `http get` 返回的内容的 signatures 字段中加上本次的签名 POST 到该地址

选项

-k, –key (string，默认 ${TIUP_HOME}/keys/private.json)

指定用于签名的私钥位置。

–timeout (int，默认 10)

通过网络签名时网络的访问超时时间，单位为秒。

> **注意：**
>
> 只有当 `<manifest-file>` 为网络地址时该选项有效。

输出

- 成功：无输出
- 文件已被指定的 key 签名过：`Error: this manifest file has already been signed by specified key`
- 文件不是合法的manifest：`Error: unmarshal manifest: %s`

<< 返回上一页 - TiUP Mirror 命令清单

10.4.7.2.8 tiup status

使用命令 `tiup status` 可查看组件的运行信息：通过 `tiup [flags] <component> [args...]` 运行组件之后，可以通过该命令查看组件的运行信息。

> **注意：**
>
> 只能查询到以下两种组件的信息：
>
> - 尚在运行的组件
> - 通过 `tiup -T/--tag` 指定tag运行的组件

语法

```
tiup status [flags]
```

选项

无

输出

由以下字段构成的表格：

- Name: 通过 `-T/--tag` 指定的 Tag 名字，若未指定，则为随机字符串
- Component: 运行的组件
- PID: 对应的进程ID
- Status: 组件运行状态
- Created Time: 启动时间
- Directory: 数据目录
- Binary: 二进制文件路径

- Args: 启动参数

组件运行状态(Status)

组件可能处于如下任一运行状态：

- 在线(Up)：组件正常运行。
- 离线(Down)或无法访问(Unreachable)：组件未启动或对应主机存在网络问题。
- 已缩容下线(Tombstone)：组件上的数据已被完整迁出并缩容完毕。仅 TiKV 或TiFlash 组件存在该状态。
- 下线中(Pending Offline)：组件上的数据正在被迁出并缩容。仅 TiKV 或TiFlash 组件存在该状态。
- 未知(Unknown)：未知的组件运行状态。

> 注意：
>
> TiUP 显示的 `Pending Offline`、PD API 返回的 `Offline` 以及TiDB Dashboard 显示的 `Leaving` 这三个状态的含义相同。

组件运行状态来自于 PD 的调度信息。更详细的描述请参考TiDB 数据库的调度– 信息收集。

<< 返回上一页- TiUP 命令清单

10.4.7.2.9 tiup telemetry

在 v1.11.3 及以上的TiUP 版本中，遥测功能默认关闭，即默认不收集使用情况信息分享给 PingCAP。在 v1.11.3 以下的TiUP 版本中，遥测功能默认开启，即默认收集使用情况信息分享给 PingCAP 用于改善产品。

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

以下信息不会被分享：

- 集群的准确名字
- 集群的拓扑结构
- 集群的配置文件

TiUP 使用命令 `tiup telemetry` 来控制遥测。

语法

```
tiup telemetry <command>
```

<command> 代表子命令，支持的子命令列表请参考下方命令一节。

命令

status

命令 tiup telemetry status 查看当前的遥测设置，输出以下信息：

- status: 当前是否开启遥测 (enable|disable)
- uuid: 随机生成的遥测标示符

reset

命令 tiup telemetry reset 重置当前的遥测标示符，以一个新的随机标识符代替之。

enable

命令 tiup telemetry enable 启用遥测。

disable

命令 tiup telemetry disable 停用遥测。

<< 返回上一页 - TiUP 命令清单

10.4.7.2.10 tiup uninstall

命令 tiup uninstall 用于卸载已安装的组件。

语法

```
tiup uninstall <component1>:<version> [component2...N] [flags]
```

- <component1> 表示要卸载的组件名字
- <version> 表示要卸载的版本，如果省略，则表示卸载该组件的全部已安装版本，因为安全原因，省略 <version> 时必须加上选项 --all 明确表示需要卸载该组件的所有版本
- [component2...N] 表示可指定卸载多个组件或版本

选项

–all

- 卸载指定组件的全部已安装版本，省略 <version> 时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–self

- 卸载 TiUP 自身：删除所有从镜像上下载过来的数据，但会保留 TiUP 及其组件产生的数据，数据存放在 TIUP_HOME 环境变量指定的目录中，若未设置过 TIUP_HOME，则默认值为 ~/.tiup/。
- 数据类型：BOOLEAN

- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 正常退出：`Uninstalled component "%s" successfully!`
- 若未指定 `<version>` 也未指定 `--all`：报错 `Use "tiup uninstall tidbx --all" if you want to remove all versions.`

<< 返回上一页 - TiUP 命令清单

10.4.7.2.11 tiup update

命令 `tiup update` 用于升级已安装的组件或者自身。

语法

```
tiup update [component1][:version] [component2..N] [flags]
```

- `[component1]` 表示要升级的组件名字
- `[version]` 表示要升级的版本，如果省略，则表示升级到该组件的最新稳定版本
- `[component2...N]` 表示可指定升级多个组件或版本。如果一个组件也不指定：即`[component1][:version] [component2..N]` 为空，则需要配合使用 `--all` 选项或 `--self` 选项。

升级操作不会删除旧的版本，仍然可以在执行时指定旧版本使用。

选项

–all

- 若未指定任何组件，则必须指定该选项。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–force

- 若指定的组件版本已经安装，则默认跳过升级操作，指定该参数可强制升级已安装版本。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–nightly

- 将指定组件升级到nightly 版本。使用该参数的命令等价于 `tiup update <component>:nightly`。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–self

- 升级TiUP 自身。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 升级成功：`Updated successfully!`
- 目标版本不存在：`Error: version %s not supported by component %s`

<< 返回上一页 - TiUP 命令清单

10.4.7.3 TiUP Cluster 命令

10.4.7.3.1 TiUP Cluster

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

语法

```
tiup cluster [command] [flags]
```

`[command]` 代表命令名字，支持的命令列表请参考下方命令清单。

选项

–ssh (string，默认 builtin)

指定 SSH 客户端连接远端（部署TiDB 服务的机器）执行命令，支持以下值：

- builtin：使用tiup-cluster 内置的 easyssh 客户端
- system：使用当前操作系统默认的 SSH 客户端
- none：不使用 ssh 客户端，这种方式只支持部署到当前机器

–ssh-timeout（uint，默认 5）

设置 SSH 连接超时时间，单位为秒。

–wait-timeout（uint，默认 120）

运维过程中涉及到很多操作：指定 systemctl 启动/停止服务，等待端口上线/下线等，每个操作可能会消耗数秒。`--wait-timeout` 用于设置每个步骤的最长等待时间（单位为秒），超时后报错退出。

-y, –yes

- 跳过所有风险操作的二次确认，除非是使用脚本调用TiUP，否则不推荐使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-v, –version

- 输出TiUP Cluster 当前版本信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出相关命令的帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

命令清单

- import：导入ansible 部署的集群
- template：输出拓扑模版
- check：部署前后的集群检查
- deploy：根据指定拓扑部署集群
- list：查询已部署的集群列表
- display：展示指定集群状态
- start：启动指定集群
- stop：停止指定集群
- restart：重启指定集群
- scale-in：缩容指定集群
- scale-out：扩容指定集群
- upgrade：升级指定集群
- prune：销毁指定集群中状态为Tombstone 的实例
- edit-config：修改指定集群配置
- reload：重载指定集群配置
- patch：替换已部署集群的某个服务
- rename：重命名集群
- clean：删除指定集群数据
- destroy：销毁指定集群
- audit：查询集群操作审计日志
- replay：重试指定命令
- enable：开启指定集群或服务开机自启动
- disable：关闭指定集群或服务开机自启动
- meta backup：备份指定集群运维操作所需的TiUP meta 文件
- meta restore：恢复指定集群的TiUP meta 文件
- help：输出帮助信息

<< 返回上一页 - TiUP 组件清单

10.4.7.3.2 tiup cluster audit

命令 `tiup cluster audit` 可以用于查看历史上对所有集群执行了什么命令，以及每个命令的执行日志。

语法

```
tiup cluster audit [audit-id] [flags]
```

- 若不填写 [audit-id] 则按时间倒序输出操作记录的表格，第一列为audit-id
- 若填写 [audit-id] 则查看指定的audit-id 的执行日志

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 若指定 [audit-id]，则输出对应的执行日志
- 若不指定 [audit-id] 则输出含有以下字段的表格：
  - ID：该条记录对应的audit-id
  - Time：该条记录对应的命令执行时间
  - Command：该条记录对应的命令

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.3 tiup cluster audit cleanup

命令 tiup cluster audit cleanup 可以用于清理 tiup cluster 产生的执行日志。

语法

```
tiup cluster audit cleanup [flags]
```

选项

–retain-days

- 执行日志保留天数
- 数据类型：INT
- 默认值：60，单位为 “天”
- 默认保留 60 天的执行日志，即删除 60 天之前的执行日志。

-h, –help

- 输出帮助信息
- 数据类型：BOOLEAN
- 默认值：false

- 在命令中添加该选项，并传入 `true` 或不传值，均可开启此功能。

输出

```
clean audit log successfully
```

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.4 tiup cluster check

对于严肃的生产环境，在正式上线之前需要进行一系列检查，来确保集群拥有最好的表现。为了简化人工检查的步骤，TiUP Cluster 提供了 `check` 子命令，用于检查指定集群的机器硬件和软件环境是否满足正常运行条件。

检查项列表

操作系统版本

检查部署机操作系统发行版和版本：目前仅支持部署在 CentOS 7 的操作系统上，之后随兼容性改进可能支持更多系统版本。

CPU EPOLLEXCLUSIVE

检查部署机 CPU 是否支持EPOLLEXCLUSIVE。

numactl

检查部署机是否安装numactl，若用户配置绑核，则必须安装numactl。

系统时间

检查部署机系统时间是否同步：将部署机系统时间与中控机对比，偏差超出某一阈值（500ms）后报错。

系统时区

检查部署机系统时区是否同步：将部署机系统的时区配置进行对比，如果时区不一致则报错。

时间同步服务

检查部署机是否配置了时间同步服务：即ntpd 是否在运行

Swap 分区

检查部署机是否启用 Swap 分区：建议禁用 Swap 分区

内核参数

检查各项内核参数的值：

- net.ipv4.tcp_tw_recycle: 0
- net.ipv4.tcp_syncookies: 0
- net.core.somaxconn: 32768
- vm.swappiness: 0
- vm.overcommit_memory: 0 或 1
- fs.file-max: 1000000

THP（透明大页）

检查部署机是否启用透明大页：建议禁用透明大页。

系统限制

检查 /etc/security/limits.conf 中各项limit 值：

```
<deploy-user>     soft    nofile     1000000
<deploy-user>     hard    nofile     1000000
<deploy-user>     soft    stack      10240
```

其中 <deploy-user> 为部署、运行TiDB 集群的用户，最后一列的数值为要求达到的最小值。

SELinux

检查 SELinux 是否启用：建议用户禁用 SELinux。

防火墙

检查 FirewallD 服务是否启用：建议用户禁用 FirewallD 或为TiDB 集群各服务添加允许规则。

irqbalance

检查irqbalance 服务是否启用：建议用户启用irqbalance 服务。

磁盘挂载参数

检查 ext4 分区的挂载参数：确保挂载参数包含nodelalloc,noatime 选项。

端口占用

检查部署机上是否已有进程占用了端口：检查拓扑中定义的端口（包括自动补全的默认端口）在部署机上是否已被占用。

> 注意：
>
> 端口占用检查假设集群尚未启动，如果检查的是已经部署并启动的集群，那么端口占用检查一定会失败，因为端口确实被占用了。

CPU 核心数

检查部署机 CPU 信息：建议生产集群 CPU 逻辑核心数>= 16

> 注意：
>
> 默认不检查 CPU 核心数，需要通过选项 --enable-cpu 启用。

内存大小

检查部署机的内存大小：建议生产集群总内存容量>= 32Gb。

> **注意：**
>
> 默认不检查内存大小，需要通过选项 `--enable-mem` 启用。

fio 磁盘性能测试

使用 fio 测试data_dir 所在磁盘的性能，包括三个测试项目：

- fio_randread_write_latency
- fio_randread_write
- fio_randread

> **注意：**
>
> 默认不进行 fio 磁盘性能测试，需要通过选项 `--enable-disk` 启用。

语法

```
tiup cluster check <topology.yml | cluster-name> [flags]
```

- 若集群尚未部署，需要传递将用于部署集群的topology.yml 文件，tiup-cluster 会根据该文件的内容连接到对应机器去检查。
- 若集群已经部署，则可以使用集群的名字 `<cluster-name>` 作为检查对象。
- 如果需要检查已部署集群的扩容拓扑文件，可以将 `<scale-out.yml>` 和 `<cluster-name>` 作为检查对象。

> **注意：**
>
> 若传递的是集群名字，则需要配合 `--cluster` 选项使用。

选项

–apply

- 尝试自动修复失败的检查项，目前仅会尝试修复以下项目：
  - SELinux
  - 防火墙
  - irqbalance
  - 内核参数
  - 系统 Limits

- THP（透明大页）

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

> 注意：
>
> tiup cluster check 也支持修复已部署集群的扩容拓扑文件，命令格式：
>
> ```
> tiup cluster check <cluster-name> scale-out.yml --cluster --apply --user root [-p]
> ↪ [-i /home/root/.ssh/gcp_rsa]
> ```

–cluster

- 对已部署的集群进行检查。
- 数据类型：BOOLEAN
- 默认值：false
- 在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。
- 命令格式：

```
tiup cluster check <topology.yml | cluster-name> --cluster [flags]
```

> 注意：
>
> - 若选择的格式为tiup cluster check <cluster-name>，则必须加上该选项：tiup cluster check <cluster-name> --cluster。
> - tiup cluster check 也支持检查已部署集群的扩容拓扑文件，命令格式：
>
> shell tiup cluster check <cluster-name> scale-out.yml --cluster --user root [-p] [-i /home/root/.ssh/gcp_rsa]

-N, –node

- 指定要检查的节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为tiup-component-cluster-display 命令返回的集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认检查所有节点，即 []。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将检查它们的交集中的服务。

-R, –role

- 指定要检查的角色。该选项的值为以逗号分割的节点角色列表，角色为`tiup-component-cluster-display`命令返回的集群状态表格的第二列。
- 数据类型：`STRINGS`
- 如果不指定该选项，默认检查所有角色。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将检查它们的交集中的服务。

–enable-cpu

- 默认情况下tiup-cluster 不检查 CPU 核心数，该选项用于启用 CPU 核心数检查。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–enable-disk

- 默认情况下tiup-cluster 不进行 fio 磁盘性能测试，该选项用于启用 fio 磁盘性能测试。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–enable-mem

默认情况下tiup-cluster 不检查内存大小，该选项用于启用内存大小检查。

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

> **注意：**
>
> 仅当 `--cluster` 选项为 false 时该选项有效，否则该值固定为部署集群时拓扑文件中指定的用户名。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

> 注意：
>
> 仅当 `--cluster` 选项为 false 时该选项有效，否则该值固定为 `${TIUP_HOME}/storage/cluster/clusters/<cluster-name>/ssh/id_rsa`

-p, –password

- 在连接目标机器时使用密码登录：
  - 对于指定了 `--cluster` 的集群，密码为部署集群时拓扑文件中指定的用户的密码
  - 对于未指定 `--cluster` 的集群，密码为 `-u/--user` 参数指定的用户的密码
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

输出含有以下字段的表格：

- Node：目标节点
- Check：检查项
- Result：检查结果（Pass/Warn/Fail）
- Message：结果描述

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.5 tiup cluster clean

在测试环境中，有时候需要将集群重置回刚部署的状态，即删除所有数据，命令 `tiup cluster clean` 可以很方便的做到这一点：它会停止集群，然后删除集群上的数据。手工重启集群之后，就能得到一个全新的集群了。

> **警告：**
>
> 该命令一定会先停止集群（即使选择只清理日志也是），生产环境请勿使用。

语法

```
tiup cluster clean <cluster-name> [flags]
```

`<cluster-name>` 为要清理的集群。

选项

–all

- 同时清理数据和日志，等价于同时指定 `--data` 和 `--log`，若不指定该选项，则必须至少指定以下选项之一：
  - –data：清理数据
  - –log：清理日志
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–data

该选项开启数据清理，若不指定该选项，也不指定 `--all`，则不清理数据。

–log

- 该选项开启日志清理，若不指定该选项，也不指定 `--all`，则不清理日志。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–ignore-node（StringArray，默认为空）

指定不需要清理的节点，如需指定多个，重复使用多次该选项：`--ignore-node <node-A> --ignore-node <node-B>`。

–ignore-role（StringArray，默认为空）

指定不需要清理的角色，如需指定多个，重复使用多次该选项：`--ignore-role <role-A> --ignore-role <role-B>`。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.6 tiup cluster deploy

命令 `tiup cluster deploy` 用于部署一个全新的集群。

语法

```
tiup cluster deploy <cluster-name> <version> <topology.yaml> [flags]
```

- `<cluster-name>` 表示新集群的名字，不能和现有集群同名
- `<version>` 为要部署的TiDB 集群版本号，如 `v7.0.0`
- `<topology.yaml>` 为事先编写好的拓扑文件

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–ignore-config-check

- 在组件二进制文件部署之后，TiUP 会对TiDB，TiKV 和 PD 组件执行配置检查，检查方式为 `<binary> --config-check <config-file>`，其中 `<binary>` 为部署的二进制文件的路径，`<config-file>` 为根据用户配置生成的配置文件。如果想要跳过该项检查，可以使用该选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–no-labels

- 当两个或多个TiKV 部署到同一台机器时，会存在一个风险：由于 PD 无法感知集群的拓扑结构，可能将一个Region 的多个副本调度到一台物理机上的不同TiKV，这样这台物理机就成为了单点。为了避免这种情况，用户可以通过label 来告诉 PD 不要将相同的Region 调度到同一台机器上（配置方式参考通过拓扑 label 进行副本调度）。
- 但是对于测试环境，可能并不在意是否将一个Region 的副本调度到了同一台机器上，这个时候可以使用 `--no-labels` 来绕过检查。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–skip-create-user

- 在部署集群时，tiup-cluster 会先检查拓扑文件中指定的用户名是否存在，如果不存在就会创建一个。指定 `--skip-create-user` 选项后不再检查用户是否存在，直接跳过创建步骤。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

部署日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.7 tiup cluster destroy

当业务下线之后，如果想将集群占有的机器释放出来让给其他业务使用，需要清理掉集群上的数据以及部署的二进制文件。`tiup cluster destroy` 命令会执行以下操作销毁集群：

- 停止集群
- 对于每个服务，删除其日志目录，部署目录，数据目录
- 如果各个服务的数据目录/部署目录的父目录是由tiup-cluster 创建的，也一并删除

语法

```
tiup cluster destroy <cluster-name> [flags]
```

`<cluster-name>` 为要销毁的集群名字。

选项

–force

- 在某些情况下，有可能集群中的某些节点已经宕机，导致无法通过 SSH 连接到节点进行操作，这个时候可以通过 `--force` 选项忽略这些错误。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–retain-node-data（StringArray，默认为空）

指定需要保留数据的节点，如需指定多个，重复使用多次该选项：`--retain-node-data <node-A> --retain-node-data <node-B>`。

–retain-role-data（StringArray，默认为空）

指定需要保留数据的角色，如需指定多个，重复使用多次该选项：`--retain-role-data <role-A> --retain-role-data <role-B>`。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.8 tiup cluster disable

命令 `tiup cluster disable` 用于关闭集群服务所在机器重启之后的自启动，该命令会到指定的节点上去执行 `systemctl disable <service>` 来关闭服务的自启动。

语法

```
tiup cluster disable <cluster-name> [flags]
```

`<cluster-name>` 为要关闭自启的集群。

选项

-N, –node

- 指定要关闭自启的节点，该选项的值为以逗号分割的节点ID 列表，节点ID 为 `tiup-component-cluster-display` 命令返回的集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有节点的自启。

> 注意：
> 若同时指定了 `-R, --role`，那么将关闭它们的交集中的服务自启。

-R, –role

- 指定要关闭自启的角色，该选项的值为以逗号分割的节点角色列表，角色为 `tiup-component-cluster-display` 命令返回的集群状态表格的第二列。

- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有角色的自启。

> 注意：
> 若同时指定了 -N, --node，那么将关闭它们的交集中的服务自启。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.9 tiup cluster display

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，tiup-cluster 提供了 tiup cluster display 命令来高效完成这件工作。

语法

```
tiup cluster display <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

–dashboard

- 默认情况会展示整个集群的所有节点信息，加上该选项后仅展示dashboard 的信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-N, –node（ strings，默认为 []，表示所有节点 ）

指定要查询的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点ID 列表，如果不确定要查询节点的ID，不指定此选项，输出会显示所有节点的ID 和状态信息。

> 注意：
> 若同时指定了 -R, --role，那么将查询它们的交集中的服务状态。

-R, –role（strings，默认为 []，表示所有角色）

指定要查询的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，如果不确定要查询节点的角色，不指定此选项，输出会显示所有节点的角色和状态信息。

> 注意：
> 若同时指定了 `-N, --node`，那么将查询它们的交集中的服务状态。

–process

- 加上该选项后会增加展示节点的 CPU 和内存的使用信息，默认情况下不展示。
- 数据类型：BOOLEAN
- 默认值：`false`
- 在命令中添加该选项，并传入 `true` 或不传值，均可开启此功能。

–uptime

- 加上该选项后会增加展示节点的 `uptime` 信息，默认情况下不展示。
- 数据类型：BOOLEAN
- 默认值：`false`
- 在命令中添加该选项，并传入 `true` 或不传值，均可开启此功能。

–status-timeout

- 获取节点状态信息的超时时间。
- 数据类型：INT
- 默认值：`10`，单位为 s。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 集群名称
- 集群版本
- SSH 客户端类型
- Dashboard 地址
- 含有以下字段的表格：
  - ID：节点ID，由 `IP:PORT` 构成

- Role：该节点部署的服务角色（如TiDB、 TiKV 等）
- Host：该节点对应的机器 IP
- Ports：服务占用的端口号
- OS/Arch：该节点的操作系统和机器架构
- Status：该节点服务当前的状态
- Data Dir：服务的数据目录，- 表示没有数据目录
- Deploy Dir：服务的部署目录

节点服务的状态(Status)

节点服务可能处于如下任一状态：

- 在线(Up)：节点服务正常运行。
- 离线(Down) 或无法访问(Unreachable)：节点服务未启动或对应主机存在网络问题。
- 已缩容下线(Tombstone)：节点服务上的数据已被完整迁出并缩容完毕。仅 TiKV 或TiFlash 存在该状态。
- 下线中(Pending Offline)：节点服务上的数据正在被迁出并缩容。仅 TiKV 或TiFlash 存在该状态。
- 未知(Unknown)：未知的节点服务运行状态。

> 注意：
>
> TiUP 显示的 `Pending Offline`、 PD API 返回的 `Offline` 以及TiDB Dashboard 显示的 `Leaving` 这三个状态的含义相同。

节点服务状态来自于 PD 的调度信息。更详细的描述请参考TiDB 数据库的调度- 信息收集。

<< 返回上一页- TiUP Cluster 命令清单

10.4.7.3.10 tiup cluster edit-config

在部署集群之后，如果需要再调整集群服务的配置，则可以使用命令 `tiup cluster edit-config`，它会启动一个编辑器（默认为$EDITOR 环境变量指定的值，当EDITOR 环境变量不存在时，使用 vi 打开）允许用户修改指定集群的拓扑文件。

> 注意：
>
> - 修改配置时不能增删机器，增删机器属于集群扩容和集群缩容的功能。
> - 执行完该命令后配置只是在中控机上修改了，要应用配置需要执行`tiup cluster reload` 命令来重新加载。

语法

```
tiup cluster edit-config <cluster-name> [flags]
```

<cluster-name> 代表要操作的集群名。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 正常情况无输出
- 若修改了不能修改的字段，则保存文件时报错并提示用户重新编辑，不能修改的字段参考拓扑文件中的相关描述

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.11 tiup cluster enable

命令 tiup cluster enable 用于设置集群服务在机器重启后的自启动，该命令会到指定的节点上去执行 systemctl enable <service> 来开启服务的自启。

> 注意：
>
> 集群全部关机重启时，服务启动的顺序由节点的操作系统启动顺序决定。重启顺序不正确时，在某些情况下，重启后的集群仍然不能提供服务（比如先启动了 TiKV 但未启动 PD，systemd 重启 TiKV 多次仍未找到 PD 则会放弃）。

语法

```
tiup cluster enable <cluster-name> [flags]
```

<cluster-name> 为要启用自启的集群。

选项

-N, –node

- 指定要开启自启的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为 tiup-component-cluster-display 命令返回的集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认开启所有节点的自启。

> 注意：
> 若同时指定了 `-R, --role`，那么将开启它们的交集中的服务自启。

-R, –role

- 指定要开启自启的角色，该选项的值为以逗号分割的节点角色列表，角色为tiup-component-cluster-display 命令返回的集群状态表格的第二列。
- 数据类型：STRINGS
- 如果不指定该选项，默认开启所有角色的自启。

> 注意：
> 若同时指定了 `-N, --node`，那么将开启它们的交集中的服务自启。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.12 tiup cluster help

tiup-cluster 在命令行界面为用户提供了丰富的帮助信息，这些帮助信息可以通过 `help` 命令或者 `--help` 参数获得。`tiup cluster help <command>` 基本等价于 `tiup cluster <command> --help`。

语法

```
tiup cluster help [command] [flags]
```

`[command]` 用于指定要查看哪个命令的帮助信息，若不指定，则查看tiup-cluster 自身的帮助信息。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

[command] 或 tiup-cluster 的帮助信息。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.13 tiup cluster import

在 TiDB 4.0 以前的版本，集群多是通过 TiDB Ansible 部署的。TiUP Cluster 提供了 import 命令用于让这些集群过渡到使用 tiup-cluster 组件管理。

> 注意：
>
> - TiDB Ansible 配置导入到 TiUP 中管理后，不能再通过 TiDB Ansible 对集群进行操作，否则可能因元信息不一致造成冲突
> - 如果使用 TiDB Ansible 部署的集群存在以下情况之一，暂不支持导入：
>   - 启用了 TLS 加密功能的集群
>   - 纯 KV 集群（没有 TiDB 实例的集群）
>   - 启用了 Kafka 的集群
>   - 启用了 Spark 的集群
>   - 启用了 TiDB Lightning/ Importer 的集群
>   - 仍使用老版本 push 的方式收集监控指标（从 v3.0 起默认为 pull 模式，如果没有特意调整过则可以支持）
>   - 在 inventory.ini 配置文件中单独为机器的 node_exporter / blackbox_exporter 通过 node_exporter_port/blackbox_exporter_port 设置了非默认端口（在 group_vars 目录中统一配置的可以兼容）
> - 如果使用 TiDB Ansible 部署的集群中有部分节点未部署监控，应当先使用 TiDB Ansible 在 inventory.ini 文件的 monitored_servers 分组中补充对应节点的信息，并通过 deploy.
> ′→ yaml playbook 将补充的监控组件部署完整。否则在集群数据导入 TiUP 后进行其他运维操作时，可能会因监控组件缺失而出错。

语法

```
tiup cluster import [flags]
```

选项

-d, –dir string（string，默认当前目录）

指定 TiDB Ansible 所在目录。

–ansible-config（string，默认 “./ansible.cfg”）

指定 Ansible 的配置文件路径。

–inventory string（string，默认 “inventory.ini”）

指定 Ansible inventory 文件的名字。

–no-backup

- 默认情况下，导入成功之后会将 `--dir` 指定的目录里所有内容备份到 `${TIUP_HOME}/.tiup/storage/cluster/clusters/{cluster-name}/ansible-backup` 下。该选项用于禁用默认的备份步骤，如果该目录下有多个inventory 文件（部署了多个集群），推荐禁用默认备份。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–rename（string，默认为空）

重命名导入的集群。默认集群名为inventory 中指定的cluster_name。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

导入过程的日志信息。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.14 tiup cluster list

tiup-cluster 支持使用同一个中控机部署多套集群，而命令 `tiup cluster list` 可以查看当前登录的用户使用该中控机部署了哪些集群。

> 注意：
>
> 部署的集群数据默认放在 `~/.tiup/storage/cluster/clusters/` 目录下，因此在同一台中控机上，当前登录用户无法查看其他用户部署的集群。

语法

```
tiup cluster list [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

输出含有以下字段的表格：

- Name：集群名字
- User：部署用户
- Version：集群版本
- Path：集群部署数据在中控机上的路径
- PrivateKey：连接集群的私钥所在路径

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.15 tiup cluster meta backup

如果运维所需的TiUP meta 文件丢失，会导致无法继续使用TiUP 管理集群。你可以通过 `tiup cluster meta backup` 命令定期备份TiUP meta 文件。

语法

```
tiup cluster meta backup <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

–file（string，默认为当前目录）

指定TiUP meta 备份文件存储的目标目录。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.16 tiup cluster meta restore

当需要恢复TiUP meta 文件时，可以通过 `tiup cluster meta restore` 命令从备份文件中恢复。

语法

```
tiup cluster meta restore <cluster-name> <backup-file> [flags]
```

- `<cluster-name>` 代表需要执行操作的集群名。
- `<backup-file>` 代表TiUP meta 备份文件所在的文件路径。

> 注意：
> 恢复操作会覆盖当前的meta 文件，建议仅在meta 文件丢失的情况下进行恢复。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.17 tiup cluster patch

在集群运行过程中，如果需要动态替换某个服务的二进制文件（即替换过程中保持集群可用），那么可以使用 `tiup cluster patch` 命令，它会完成以下几件事情：

- 将用于替换的二进制包上传到目标机器
- 如果目标服务是TiKV、 TiFlash 或者TiDB Binlog 之类的存储服务，则先通过API 下线节点
- 停止目标服务
- 解压二进制包，替换服务
- 启动目标服务

语法

```
tiup cluster patch <cluster-name> <package-path> [flags]
```

- `<cluster-name>` 代表要操作的集群名
- `<package-path>` 为用于替换的二进制包路径

准备二进制包

在运行 `tiup cluster patch` 命令之前，你需要打包所需的二进制文件。请按照以下步骤操作：

1. 确定以下变量的值：
    - `${component}`：需要替换的组件名（例如 `tidb`、`tikv`、`pd`）。
    - `${version}`：组件的版本（例如 `v7.0.0`、`v6.5.1`）。
    - `${os}`：操作系统 (linux)。

- `${arch}`：组件运行的平台 (amd64、arm64)。

2. 下载当前的组件包：

```
wget https://tiup-mirrors.pingcap.com/${component}-${version}-${os}-${arch}.tar.gz -O /tmp/$
    ↪ {component}-${version}-${os}-${arch}.tar.gz
```

3. 创建临时打包目录：

```
mkdir -p /tmp/package && cd /tmp/package
```

4. 解压原来的二进制包：

```
tar xf /tmp/${component}-${version}-${os}-${arch}.tar.gz
```

5. 查看临时打包目录中的文件结构：

```
find .
```

6. 将要替换的二进制文件或配置文件复制到临时目录的对应位置。

7. 将临时目录中的所有文件打包：

```
tar czf /tmp/${component}-hotfix-${os}-${arch}.tar.gz *
```

完成上述步骤后，你可以在 `tiup cluster patch` 命令中使用 `/tmp/${component}-hotfix-${os}-${arch}.tar.gz` 作为 <package-path>。

选项

–overwrite

- 对某个组件（比如TiDB，TiKV）进行patch后，如果要在该集群扩容该组件，tiup-cluster会默认使用patch前的版本。如果希望后续扩容的时候也使用patch之后的版本，需要指定 `--overwrite` 选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–transfer-timeout（uint，默认 600）

在重启 PD 或TiKV时，会先将被重启节点的leader迁移到其他节点，迁移过程会需要一定时间，可以通过设置 `--transfer-timeout` 设置最长等待时间（单位为秒），超时之后会跳过等待直接重启服务。

> **注意：**
>
> 若出现跳过等待直接重启的情况，服务性能可能会出现抖动。

-N, –node（strings，默认为 []，未选中任何节点）

指定要替换的节点，该选项的值为以逗号分割的节点ID 列表，节点ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 `-R, --role`，那么将替换它们的交集中的服务。

-R, –role（strings，默认为 []，未选中任何角色）

指定要替换的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
> 若同时指定了 `-N, --node`，那么将替换它们的交集中的服务。

–offline

- 声明当前集群处于停止状态。指定该选项时，TiUP Cluster 仅原地替换集群组件的二进制文件，不执行迁移 Leader 以及重启服务等操作。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.18 tiup cluster prune

在缩容集群时，对于某些组件，并不会立即停止服务并删除数据，而是需要等数据调度完成之后，用户手动执行 `tiup cluster prune` 命令清理。

语法

```
tiup cluster prune <cluster-name> [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

清理过程的日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.19 tiup cluster reload

在修改集群配置之后，需要通过 tiup cluster reload 命令让集群重新加载配置才会生效，该命令会将中控机的配置发布到服务运行的远端机器，并按照升级的流程按顺序重启服务，重启过程中集群仍然可用。

语法

```
tiup cluster reload <cluster-name> [flags]
```

<cluster-name> 代表要操作的集群名。

选项

–force

- 忽略重新加载过程中的错误，强制reload。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–transfer-timeout（uint，默认 600）

在重启 PD 或TiKV 时，会先将被重启节点的leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 --transfer-timeout 设置最长等待时间（单位为秒），超时之后会跳过等待直接重启服务。

> 注意：
> 若出现跳过等待直接重启的情况，服务性能可能会出现抖动。

–ignore-config-check

- 在组件二进制文件部署之后，TiUP 会对TiDB，TiKV 和 PD 组件执行配置检查，检查方式为 <binary> --config-check <config-file>，其中 <binary> 为部署的二进制文件的路径，<config-file> 为根据用户配置生成的配置文件。如果想要跳过该项检查，可以使用该选项。
- 数据类型：BOOLEAN

- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> - 若同时指定了 `-R, --role`，那么将重启它们的交集中的服务
> - 若指定了选项 `--skip-restart`，则该选项无效

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 1. 若同时指定了 `-N, --node`，那么将重启它们的交集中的服务
> 2. 若指定了选项 `--skip-restart`，则该选项无效

–skip-restart

- 命令 `tiup cluster reload` 会执行两个操作：
  - 刷新所有节点配置
  - 重启指定节点
- 该选项指定后仅刷新配置，不重启任何节点，这样刷新的配置也不会应用，需要等对应服务下次重启才会生效。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.20 tiup cluster rename

集群名字是部署集群时指定的，在集群部署之后，如果想更改集群名字，可以通过 `tiup cluster rename` 命令来实现。

> **注意：**
>
> 如果配置了 `grafana_servers` 的 `dashboard_dir` 字段，在执行tiup cluster rename 命令进行集群重命名后，需要额外做以下操作：
>
> - 对于本地的dashboards 目录中的 `*.json` 文件，将datasource 字段的值更新为新的集群名（这是因为datasource 是以集群名命名的）
> - 执行 `tiup cluster reload -R grafana` 命令

语法

```
tiup cluster rename <old-cluster-name> <new-cluster-name> [flags]
```

- `<old-cluster-name>` 老的集群名
- `<new-cluster-name>` 新的集群名

选项

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.21 tiup cluster replay

对集群进行升级或重启等操作时，操作有可能因为环境的原因而偶然失败。这时如果重新进行操作，需要从头开始执行所有步骤。如果集群规模较大，会耗费较长时间。此时可以使用 `tiup cluster replay` 命令重试刚才失败的命令，并且跳过已经成功的步骤。

语法

```
tiup cluster replay <audit-id> [flags]
```

- `<audit-id>` 代表要重试的命令对应的audit-id。使用`tiup cluster audit` 可查看历史命令及其audit-id。

选项

-h, –help

输出帮助信息。

输出

<audit-id> 对应的命令的输出。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.22 tiup cluster restart

命令 tiup cluster restart 用于重启指定集群的所有或部分服务。

> 注意：
> 重启过程中会有一段时间服务不可用。

语法

```
tiup cluster restart <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 -R, --role，那么将重启它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
> 若同时指定了 -N, --node，那么将重启它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

重启服务的日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.23 tiup cluster scale-in

`tiup cluster scale-in` 命令用于集群缩容，缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

下线特殊处理

由于 TiKV，TiFlash 和 TiDB Binlog 组件的下线是异步的（需要先通过 API 执行移除操作）并且下线过程耗时较长（需要持续观察节点是否已经下线成功），所以对 TiKV，TiFlash 和 TiDB Binlog 组件做了特殊处理：

- 对 TiKV，TiFlash 及 TiDB Binlog 组件的操作:
  - tiup-cluster 通过 API 将其下线后直接退出而不等待下线完成
  - 执行 `tiup cluster display` 查看下线节点的状态，等待其状态变为 Tombstone
  - 执行 `tiup cluster prune` 命令清理 Tombstone 节点，该命令会执行以下操作：
    * 停止已经下线掉的节点的服务
    * 清理已经下线掉的节点的相关数据文件
    * 更新集群的拓扑，移除已经下线掉的节点
- 对其他组件的操作
  - 下线 PD 组件时，会通过 API 将指定节点从集群中删除掉（这个过程很快），然后停掉指定 PD 的服务并且清除该节点的相关数据文件
  - 下线其他组件时，直接停止并且清除节点的相关数据文件

语法

```
tiup cluster scale-in <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

-N, –node（strings，无默认值，必须非空）

选择要缩容的节点，若缩容多个节点，以逗号分割。

–force

- 在某些情况下，有可能被缩容的节点宿主机已经宕机，导致无法通过 SSH 连接到节点进行操作，这个时候可以通过 `--force` 选项强制将其从集群中移除。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> **警告：**
>
> 使用该选项强制移除正在服务和下线中的TiKV / TiFlash 节点时，这些节点会被直接删除，不等待数据调度完成，因此这个场景下，数据丢失风险非常大。不建议对未宕机的节点使用该选项。如果元数据所在的Region 发生数据丢失，整个集群将不可用且无法恢复。

–transfer-timeout（uint，默认 600）

在缩容 PD 或TiKV 时，会先将被缩容节点的leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 `--transfer-timeout` 设置最长等待时间（单位为秒），超时之后会跳过等待直接缩容服务。

> **注意：**
>
> 若出现跳过等待直接缩容的情况，服务性能可能会出现抖动。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

缩容日志

<< 返回上一页- TiUP Cluster 命令清单

10.4.7.3.24 tiup cluster scale-out

`tiup cluster scale-out` 命令用于集群扩容，扩容的内部逻辑与部署类似，tiup-cluster 组件会先建立新节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。其中 PD 节点的扩容会通过join 方式加入到集群中，并且会更新与 PD 有关联的服务的配置；其他服务直接启动加入到集群中。

语法

```
tiup cluster scale-out <cluster-name> <topology.yaml> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看
- `<topology.yaml>` 为事先编写好的扩容拓扑文件，该文件应当仅包含扩容部分的拓扑

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–no-labels

- 当两个或多个TiKV 部署到同一台机器时，会存在一个风险：由于 PD 无法感知集群的拓扑结构，可能将一个Region 的多个副本调度到一台物理机上的不同TiKV，这样这台物理机就成为了单点。为了避免这种情况，用户可以通过label 来指定 PD 不要将相同的Region 调度到同一台机器上（配置方式参考通过拓扑 label 进行副本调度）。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

但是对于测试环境，可能并不在意是否将一个 Region 的副本调度到了同一台机器上，这个时候可以使用 `--no-labels` 来绕过检查。

–skip-create-user

在扩容集群时，tiup-cluster 会先检查拓扑文件中指定的用户名是否存在，如果不存在就会创建一个。指定 `--skip-create-user` 选项后不再检查用户是否存在，直接跳过创建步骤。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

扩容日志

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.25 tiup cluster start

命令 `tiup cluster start` 用于启动指定集群的所有或部分服务。

语法

```
tiup cluster start <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

–init

以安全方式启动集群。推荐在集群第一次启动时使用，该方式会在启动时自动生成TiDB root 用户的密码，并在命令行界面返回密码。

> 注意：
>
> - 使用安全启动方式后，不能通过无密码的root 用户登录数据库，你需要记录命令行返回的密码进行后续操作。
> - 该自动生成的密码只会返回一次，如果没有记录或者忘记该密码，请参照忘记 root 密码修改密码。

-N, –node（strings，默认为 []，表示所有节点）

指定要启动的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> 若同时指定了 -R, --role，那么将启动它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要启动的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 若同时指定了 -N, --node，那么将启动它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

启动日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.26 tiup cluster stop

命令 `tiup cluster stop` 用于停止指定集群的所有服务或部分服务。

> **注意：**
>
> 核心服务停止后集群将无法提供服务。

语法

```
tiup cluster stop <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要停止的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将停止它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要停止的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将停止它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

停止服务的日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.27 tiup cluster template

部署集群之前，需要准备一份集群的拓扑文件。TiUP 内置了拓扑文件的模版，用户可以通过修改该模版来生成最终的拓扑文件。使用 `tiup cluster template` 命令可以输出TiUP 内置的模版内容。

语法

```
tiup cluster template [flags]
```

如果不指定该选项，输出的默认模版包含以下实例：

- 3 个 PD 实例
- 3 个 TiKV 实例
- 1 个 TiDB 实例
- 1 个 Prometheus 实例
- 1 个 Grafana 实例
- 1 个 Alertmanager 实例

选项

–full

- 输出详细的拓扑模版，该模版会以注释的形式带上可配置的参数。在命令中添加该选项，可开启该选项。
- 如果不指定该选项，默认输出最简单的拓扑模版。

–multi-dc

- 输出多数据中心的拓扑模版。在命令中添加该选项，可开启该选项。
- 如果不指定该选项，默认输出单地单机房的拓扑模版。

-h, –help

输出帮助信息。

输出

根据指定选项输出拓扑模版，可重定向到拓扑文件中用于部署。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.3.28 tiup cluster upgrade

命令 `tiup cluster upgrade` 用于将指定集群升级到特定版本。

语法

```
tiup cluster upgrade <cluster-name> <version> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

- <version> 为要升级到的目标版本，例如 v6.5.0。目前仅允许升级到比当前集群更高的版本，不允许升级到比当前集群更低的版本，即不允许降级。同时也不允许升级成nightly 版本

选项

–force

- 升级集群需要保证集群目前是启动的，在某些情况下，可能希望在集群未启动的状态下升级，这时候可以使用 --force 忽略升级过程的错误，强制替换二进制文件并启动集群。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

> 注意：
> 对正在提供服务的集群强制升级可能导致集群服务不可用。对于未启动的集群，升级成功后会自动启动集群。

–transfer-timeout（uint，默认 600）

在升级 PD 或TiKV 时，会先将被升级节点的leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 --transfer-timeout 设置最长等待时间（单位为秒），超时之后会跳过等待直接升级服务。

> 注意：
> 若出现跳过等待直接升级的情况，服务性能可能会出现抖动。

–ignore-config-check

- 在二进制文件更新之后，TiUP 会对TiDB，TiKV 和 PD 组件执行配置检查，检查方式为 <binary> --config-check <config-file>，其中 <binary> 为新部署的二进制文件的路径，<config-file> 为根据用户配置生成的配置文件。如果想要跳过该项检查，可以使用该选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–offline

- 声明当前集群处于停止状态。指定该选项时，TiUP Cluster 仅原地替换集群组件的二进制文件，不执行迁移 Leader 以及重启服务等操作。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

升级服务的日志。

<< 返回上一页 - TiUP Cluster 命令清单

10.4.7.4 TiUP DM 命令

10.4.7.4.1 TiUP DM

类似于TiUP Cluster 对TiDB 集群的管理，TiUP DM 是用于对 DM 集群的日常运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 DM 集群、管理 DM 集群参数。

语法

```
tiup dm [command] [flags]
```

`[command]` 代表命令名字，支持的命令列表请参考下方命令清单。

选项

–ssh (string，默认 builtin)

指定 SSH 客户端连接远端（部署TiDB 服务的机器）执行命令，支持以下几个值：

- builtin：使用tiup-cluster 内置的 easyssh 客户端
- system：使用当前操作系统默认的 ssh 客户端
- none：不使用 ssh 客户端，这种方式只支持部署到当前机器

–ssh-timeout（uint，默认 5）

设置 ssh 连接超时时间，单位为秒。

–wait-timeout（uint，默认 120）

运维过程中涉及到很多操作：指定 systemctl 启动/停止服务，等待端口上线/下线等，每个操作可能会消耗数秒，`--wait-timeout` 用于设置每个步骤的最长等待时间（单位为秒），超时后报错退出。

-y, –yes

- 跳过所有风险操作的二次确认，除非是使用脚本调用TiUP，否则不推荐使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-v, –version

- 输出tiup-dm 当前版本信息。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出相关命令的帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

命令清单

- import：导入使用 DM-Ansible 部署的 DM v1.0 集群
- template：输出拓扑模版
- deploy：根据指定拓扑部署集群
- list：查询已部署的集群列表
- display：展示指定集群状态
- start：启动指定集群
- stop：停止指定集群
- restart：重启指定集群
- scale-in：缩容指定集群
- scale-out：扩容指定集群
- upgrade：升级指定集群
- prune：销毁指定集群中状态为Tombstone 的实例
- edit-config：修改指定集群配置
- reload：重载指定集群配置
- patch：替换已部署集群的某个服务
- destroy：销毁指定集群
- audit：查询集群操作审计日志
- replay：重试指定命令
- enable：开启指定集群或服务开机自启动
- disable：关闭指定集群或服务开机自启动
- help：输出帮助信息

<< 返回上一页 - TiUP 组件清单

10.4.7.4.2 tiup dm audit

命令 `tiup dm audit` 可以用于查看执行在所有集群上的历史命令，以及每个命令的执行日志。

语法

```
tiup dm audit [audit-id] [flags]
```

- 若不填写 `[audit-id]` 则按时间倒序输出操作记录的表格，第一列为audit-id
- 若填写 `[audit-id]` 则查看指定的audit-id 的执行日志

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若指定 `[audit-id]`，则输出对应的执行日志
- 若不指定 `[audit-id]` 则输出含有以下字段的表格：
  - ID：该条记录对应的audit-id
  - Time：该条记录对应的命令执行时间
  - Command：该条记录对应的命令

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.3 tiup dm deploy

命令 `tiup dm deploy` 用于部署一个全新的集群。

语法

```
tiup dm deploy <cluster-name> <version> <topology.yaml> [flags]
```

- `<cluster-name>` 表示新集群的名字，不能和现有集群同名
- `<version>` 为要部署的 DM 集群版本号，如 `v2.0.0`
- `<topology.yaml>` 为事先编写好的拓扑文件

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

部署日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.4 tiup dm destroy

当业务下线之后，如果想将集群占有的机器释放出来让给其他业务使用，需要清理掉集群上的数据以及部署的二进制文件。`tiup dm destroy` 命令会执行以下操作销毁集群：

- 停止集群
- 对于每个服务，删除其日志目录，部署目录，数据目录
- 如果各个服务的数据目录/部署目录的父目录是由tiup-dm 创建的，也一并删除

语法

```
tiup dm destroy <cluster-name> [flags]
```

`<cluster-name>` 为要销毁的集群名字。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.5 tiup dm disable

集群服务所在的机器重启之后会自启动，命令 `tiup dm disable` 用于关闭该自启动。该命令会在指定的节点上执行 `systemctl disable <service>` 来关闭服务的自启动。

语法

```
tiup dm disable <cluster-name> [flags]
```

`<cluster-name>` 为要关闭自启的集群。

选项

-N, –node

- 指定要关闭自启的节点，该选项的值为以逗号分割的节点ID 列表，节点ID 为集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有节点的自启。

> 注意：
> 若同时指定了 -R, --role，那么将关闭它们的交集中的服务自启。

-R, –role

- 指定要关闭自启的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有角色的自启。

> 注意：
> 若同时指定了 -N, --node，那么将关闭它们的交集中的服务自启。

-h, –help

输出帮助信息。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.6 tiup dm display

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，tiup-dm 提供了 tiup dm display 命令来高效完成这件工作。

语法

```
tiup dm display <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要查询的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点ID 列表，如果不确定要查询节点的ID，不指定此选项，输出会显示所有节点的ID 和状态信息。

> 若同时指定了 `-R, --role`，那么将查询它们的交集中的服务状态。

-R, –role（strings，默认为 []，表示所有角色）

指定要查询的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，如果不确定要查询节点的角色，不指定此选项，输出会显示所有节点的角色和状态信息。

> 注意：
> 若同时指定了 `-N, --node`，那么将查询它们的交集中的服务状态。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 集群名称
- 集群版本
- SSH 客户端类型
- 含有以下字段的表格：
  - ID：节点ID，由IP:PORT构成
  - Role：该节点部署的服务角色（如TiDB和TiKV等）
  - Host：该节点对应的机器 IP
  - Ports：服务占用的端口号
  - OS/Arch：该节点的操作系统和机器架构
  - Status：该节点服务当前的状态
  - Data Dir：服务的数据目录，- 表示没有数据目录
  - Deploy Dir：服务的部署目录

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.7 tiup dm edit-config

在部署集群之后，如果需要再调整集群服务的配置，则可以使用命令 `tiup dm edit-config`，它会启动一个编辑器（默认为 `$EDITOR` 环境变量指定的值，当EDITOR环境变量不存在时，使用 vi 打开）允许用户修改指定集群的拓扑文件。

- 修改配置时不能增删机器，增删机器属于集群扩容和集群缩容的功能。
- 执行完该命令后配置只是在中控机上修改了，要应用配置需要执行 `tiup dm reload` 命令来重新加载。

语法

```
tiup dm edit-config <cluster-name> [flags]
```

`<cluster-name>` 代表要操作的集群名。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 正常情况无输出
- 若修改了不能修改的字段，则保存文件时报错并提示用户重新编辑，不能修改的字段参考拓扑文件中的相关描述

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.8 tiup dm enable

命令 `tiup dm enable` 用于设置集群服务在机器重启后的自启动，该命令会到指定的节点上去执行 `systemctl enable <service>` 来开启服务的自启。

语法

```
tiup dm enable <cluster-name> [flags]
```

`<cluster-name>` 为要启用自启的集群。

选项

-N, –node

- 指定要开启自启的节点，该选项的值为以逗号分割的节点ID 列表，节点ID 为集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认开启所有节点的自启。

> 若同时指定了 `-R, --role`，那么将开启它们的交集中的服务自启。

-R, –role

- 指定要开启自启的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。
- 数据类型：`STRINGS`
- 如果不指定该选项，默认开启所有角色的自启。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将开启它们的交集中的服务自启。

-h, –help

输出帮助信息。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.9 tiup dm help

tiup-dm 在命令行界面为用户提供了丰富的帮助信息，这些帮助信息可以通过 `help` 命令或者 `--help` 参数获得。基本上，`tiup dm help <command>` 等价于 `tiup dm <command> --help`。

语法

```
tiup dm help [command] [flags]
```

`[command]` 用于指定要查看哪个命令的帮助信息，若不指定，则查看tiup-dm 自身的帮助信息。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

`[command]` 或tiup-dm 的帮助信息。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.10 tiup dm import 仅适用于升级 DM v1.0

该命令仅适用于将 DM 集群从 v1.0 升级到 v2.0 或更高版本。

在 DM 1.0 版本，集群基本是通过TiDB Ansible 部署的，TiUP DM 提供了 `import` 命令导入 v1.0 的集群并重新部署 v2.0 的集群。

> 注意：
>
> - 不支持导入 v1.0 集群中的 DM Portal 组件
> - 导入前请先停止原集群
> - 对于需要升级到 v2.0 的数据迁移任务，请不要执行 `stop-task`
> - 仅支持导入到 v2.0.0-rc.2 或更高版本
> - `import` 命令用于将 DM v1.0 集群导入到全新的 v2.0 集群。如果需要将数据迁移任务导入到已有的 v2.0 集群，请参考TiDB Data Migration 1.0.x 到 2.0.x+ 手动升级
> - 部分组件生成的部署目录会跟原集群不一样，具体可以使用 `display` 命令查看
> - 导入前运行 `tiup update --self && tiup update dm` 确认升级TiUP DM 组件到最新版本
> - 导入后集群中仅会有一个 DM-master 节点，可参考扩容节点对 DM-master 进行扩容

语法

```
tiup dm import [flags]
```

选项

-v, –cluster-version（string，required）

重新部署的版本号，必须指定 v2.0.0-rc.2 或更高版本。

-d, –dir string（string，默认当前目录）

指定TiDB Ansible 所在目录。

–inventory string（string，默认 “inventory.ini”）

指定Ansible inventory 文件的名字。

–rename（string，默认为空）

重命名导入的集群。默认集群名为inventory 中指定的 `cluster_name`。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

导入过程的日志信息。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.11 tiup dm list

tiup-dm 支持使用同一个中控机部署多套集群，而命令 `tiup dm list` 可以查看当前登录的用户使用该中控机部署了哪些集群。

> **注意：**
>
> 部署的集群数据默认放在 `~/.tiup/storage/dm/clusters/` 目录下，因此在同一台中控机上，当前登录用户无法查看其他用户部署的集群。

语法

```
tiup dm list [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

输出含有以下字段的表格：

- Name：集群名字
- User：部署用户
- Version：集群版本
- Path：集群部署数据在中控机上的路径
- PrivateKey：连接集群的私钥所在路径

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.12 在线应用 Hotfix 到 DM 集群

在集群运行过程中，如果需要动态替换某个服务的二进制文件（即替换过程中保持集群可用），那么可以使用 `tiup dm patch` 命令，它会完成以下几件事情：

- 将用于替换的二进制包上传到目标机器
- 通过 API 下线节点
- 停止目标服务
- 解压二进制包，替换服务
- 启动目标服务

语法

```
tiup dm patch <cluster-name> <package-path> [flags]
```

- <cluster-name> 代表要操作的集群名。
- <package-path> 为用于替换的二进制包。

准备条件

执行 tiup dm patch 命令前，需要进行以下操作准备用于替换的二进制包：

- 确定当前要替换的组件名称 ${component} (dm-master，dm-worker 等) 以及其版本 ${version} (v2.0.0，v2.0.1 等)，以及其运行的平台 ${os} (linux) 和 ${arch} (amd64, arm64)
- 下载当前的组件包：wget https://tiup-mirrors.pingcap.com/${component}-${version}-${os}-${arch}.tar.gz -O /tmp/${component}-${version}-${os}-${arch}.tar.gz
- 建立临时打包目录：mkdir -p /tmp/package && cd /tmp/package
- 解压原来的二进制包：tar xf /tmp/${component}-${version}-${os}-${arch}.tar.gz
- 查看临时打包目录中的文件结构：find .
- 将要替换的二进制文件或配置文件复制到临时目录的对应位置
- 重新打包 tar czf /tmp/${component}-hotfix-${os}-${arch}.tar.gz *

完成以上操作后，/tmp/${component}-hotfix-${os}-${arch}.tar.gz 就可以作为 <package-path> 用于 patch 命令中。

选项

–overwrite

- 对某个组件（比如TiDB，TiKV）进行patch 之后，该集群扩容该组件时，tiup-dm 默认会用原来的版本。如果希望后续扩容的时候也使用patch 之后的版本的话，就需要指定 --overwrite 选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-N, –node（strings，默认为 []，未选中任何节点）

指定要替换的节点，该选项的值为以逗号分割的节点ID 列表，节点ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 -R, --role，那么将替换它们的交集中的服务。

-R, –role（strings，默认为 []，未选中任何角色）

指定要替换的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将替换它们的交集中的服务。

–offline

声明当前集群处于离线状态。指定该选项时，TiUP DM 仅原地替换集群组件的二进制文件，不重启服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

在线应用 Hotfix 示例

以下将在使用TiUP 部署的 DM 环境中演示如何应用 `v5.3.0-hotfix` 到 `v5.3.0`集群，其他部署方式可能需要调整部分操作。

> **注意：**
>
> Hotfix 仅用于紧急修复，其日常维护较为复杂，建议在正式版本发布后及时升级。

准备工作

在开始应用 Hotfix 之前，请准备好 Hotfix 补丁包 `dm-linux-amd64.tar.gz`，并确认当前环境 DM 软件版本：

```
/home/tidb/dm/deploy/dm-master-8261/bin/dm-master/dm-master  -V
```

输出示例：

```
Release Version: v5.3.0
Git Commit Hash: 20626babf21fc381d4364646c40dd84598533d66
Git Branch: heads/refs/tags/v5.3.0
UTC Build Time: 2021-11-29 08:29:49
Go Version: go version go1.16.4 linux/amd64
```

制作 Patch 补丁包并应用到 DM 集群

1. 准备当前环境版本的 DM 软件包：

```
mkdir -p /tmp/package
```

```
tar -zxvf /root/.tiup/storage/dm/packages/dm-master-v5.3.0-linux-amd64.tar.gz -C /tmp/
    ↪ package/
tar -zxvf /root/.tiup/storage/dm/packages/dm-worker-v5.3.0-linux-amd64.tar.gz -C /tmp/
    ↪ package/
```

```

```

2. 替换新的二进制文件：

```
# 解压 Hotfix 压缩包并替换
cd /root; tar -zxvf dm-linux-amd64.tar.gz
cp /root/dm-linux-amd64/bin/dm-master /tmp/package/dm-master/dm-master
cp  /root/dm-linux-amd64/bin/dm-worker  /tmp/package/dm-worker/dm-worker

# 重新打包
# 注意，其他部署方式可能有所不同
cd /tmp/package/ && tar -czvf dm-master-hotfix-linux-amd64.tar.gz dm-master/
cd /tmp/package/ && tar -czvf dm-worker-hotfix-linux-amd64.tar.gz dm-worker/
```

3. 应用补丁。

   查询当前集群状态，以名称为 dm-test 的集群为例：

```
tiup dm display dm-test
```

   输出示例：

```
Cluster type:       dm
Cluster name:       dm-test
Cluster version:    v5.3.0
Deploy user:        tidb
SSH type:           builtin
ID                  Role                 Host           Ports       OS/Arch       Status
    ⟶ Data Dir                              Deploy Dir
--                  -----                -------        ---------      --------      -------
    ⟶ --------                              ----------
172.16.100.21:9093  alertmanager         172.16.100.21  9093/9094  linux/x86_64  Up
    ⟶ /home/tidb/dm/data/alertmanager-9093  /home/tidb/dm/deploy/alertmanager-9093
172.16.100.21:8261  dm-master            172.16.100.21  8261/8291  linux/x86_64  Healthy|L
    ⟶ /home/tidb/dm/data/dm-master-8261     /home/tidb/dm/deploy/dm-master-8261
172.16.100.21:8262  dm-worker            172.16.100.21  8262       linux/x86_64  Free
    ⟶ /home/tidb/dm/data/dm-worker-8262     /home/tidb/dm/deploy/dm-worker-8262
172.16.100.21:3000  grafana              172.16.100.21  3000       linux/x86_64  Up
    ⟶ -                                     /home/tidb/dm/deploy/grafana-3000
172.16.100.21:9090  prometheus           172.16.100.21  9090       linux/x86_64  Up
    ⟶ /home/tidb/dm/data/prometheus-9090    /home/tidb/dm/deploy/prometheus-9090
Total nodes: 5
```

   将补丁应用到指定节点或指定角色，若同时使用 -R 和 -N，将会取其交集。

```
# 为指定节点应用补丁
tiup dm patch dm-test dm-master-hotfix-linux-amd64.tar.gz -N 172.16.100.21:8261
tiup dm patch dm-test dm-worker-hotfix-linux-amd64.tar.gz -N 172.16.100.21:8262

# 为指定角色应用补丁
```

```
tiup dm patch dm-test dm-master-hotfix-linux-amd64.tar.gz -R dm-master
tiup dm patch dm-test dm-worker-hotfix-linux-amd64.tar.gz -R dm-worker
```

4. 查看补丁应用结果：

```
/home/tidb/dm/deploy/dm-master-8261/bin/dm-master/dm-master  -V
```

输出示例：

```
Release Version: v5.3.0-20211230
Git Commit Hash: ca7070c45013c24d34bd9c1e936071253451d707
Git Branch: heads/refs/tags/v5.3.0-20211230
UTC Build Time: 2022-01-05 14:19:02
Go Version: go version go1.16.4 linux/amd64
```

集群信息也会有所变化：

```
tiup dm display dm-test
```

输出示例：

```
Starting component `dm`: /root/.tiup/components/dm/v1.8.1/tiup-dm display dm-test
Cluster type:       dm
Cluster name:       dm-test
Cluster version:    v5.3.0
Deploy user:        tidb
SSH type:           builtin
ID                  Role                 Host            Ports       OS/Arch       Status
    '→ Data Dir                              Deploy Dir
--                  -----                -------         ------      --------      -------
    '→ --------                              ----------
172.16.100.21:9093  alertmanager         172.16.100.21   9093/9094   linux/x86_64  Up
    '→ /home/tidb/dm/data/alertmanager-9093   /home/tidb/dm/deploy/alertmanager-9093
172.16.100.21:8261  dm-master (patched)  172.16.100.21   8261/8291   linux/x86_64  Healthy|L
    '→ /home/tidb/dm/data/dm-master-8261      /home/tidb/dm/deploy/dm-master-8261
172.16.100.21:8262  dm-worker (patched)  172.16.100.21   8262        linux/x86_64  Free
    '→ /home/tidb/dm/data/dm-worker-8262      /home/tidb/dm/deploy/dm-worker-8262
172.16.100.21:3000  grafana              172.16.100.21   3000        linux/x86_64  Up
    '→ -                                     /home/tidb/dm/deploy/grafana-3000
172.16.100.21:9090  prometheus           172.16.100.21   9090        linux/x86_64  Up
    '→ /home/tidb/dm/data/prometheus-9090     /home/tidb/dm/deploy/prometheus-9090
Total nodes: 5
```

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.13 tiup dm prune

在缩容集群后，etcd 中仍然会有少量元信息不会被清理，通常不会有问题，如果确实需要清理，可以手动执行 `tiup dm prune` 命令清理。

语法

```
tiup dm prune <cluster-name> [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

清理过程的日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.14 tiup dm reload

在修改集群配置之后，需要通过 `tiup dm reload` 命令让集群重新加载配置才会生效，该命令会将中控机的配置发布到服务运行的远端机器，并按照升级的流程按顺序重启服务，重启过程中集群仍然可用。

语法

```
tiup dm reload <cluster-name> [flags]
```

`<cluster-name>` 代表要操作的集群名。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> - 若同时指定了 `-R, --role`，那么将重启它们的交集中的服务
> - 若指定了选项 `--skip-restart`，则该选项无效

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

注意：

- 若同时指定了 -N, --node，那么将重启它们的交集中的服务
- 若指定了选项 --skip-restart，则该选项无效

–skip-restart

- 命令 tiup dm reload 会执行两个操作：
  - 刷新所有节点配置
  - 重启指定节点
- 该选项指定后仅刷新配置，不重启任何节点，这样刷新的配置也不会应用，需要等对应服务下次重启才会生效。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

tiup-dm 的执行日志。

<<返回上一页- TiUP DM 命令清单

10.4.7.4.15 tiup dm replay

对集群进行升级或重启等操作时，操作有可能因为环境的原因而偶然失败。这时如果重新进行操作，需要从头开始执行所有步骤。如果集群规模较大，会耗费较长时间。此时可以使用 tiup dm replay 命令，命令重试刚才失败的命令，并且跳过已经成功的步骤。

语法

```
tiup dm replay <audit-id> [flags]
```

- <audit-id> 代表要重试的命令对应的 audit-id。使用tiup dm audit 可查看历史命令及其 audit-id。

选项

-h, –help

输出帮助信息。

输出

<audit-id> 对应的命令的输出。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.16 tiup dm restart

命令 tiup dm restart 用于重启指定集群的所有或部分服务。

> 注意：
> 重启过程中会有一段时间服务不可用。

语法

```
tiup dm restart <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 -R, --role，那么将重启它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
> 若同时指定了 -N, --node，那么将重启它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

重启服务的日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.17 tiup dm scale-in

`tiup dm scale-in` 命令用于集群缩容，缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

语法

```
tiup dm scale-in <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，无默认值，必须非空）

选择要缩容的节点，若缩容多个节点，以逗号分割。

–force

- 在某些情况下，有可能被缩容的节点宿主机已经宕机，导致无法通过 SSH 连接到节点进行操作，这个时候可以通过 `--force` 选项强制将其从集群中移除。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

缩容日志

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.18 tiup dm scale-out

`tiup dm scale-out` 命令用于集群扩容，扩容的内部逻辑与部署类似，tiup-dm 组件会先建立新节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。

语法

```
tiup dm scale-out <cluster-name> <topology.yaml> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表
- `<topology.yaml>` 为事先编写好的扩容拓扑文件，该文件应当仅包含扩容部分的拓扑

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

扩容日志

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.19 tiup dm start

命令 `tiup dm start` 用于启动指定集群的所有或部分服务。

语法

```
tiup dm start <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要启动的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将启动它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要启动的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将启动它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

启动日志。

<<返回上一页- TiUP DM 命令清单

10.4.7.4.20 tiup dm stop

命令 `tiup dm stop` 用于停止指定集群的所有服务或部分服务。

> **注意：**
>
> 核心服务停止后集群将无法提供服务。

语法

```
tiup dm stop <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要停止的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 -R, --role，那么将停止它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要停止的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
> 若同时指定了 -N, --node，那么将停止它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

停止服务的日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.21 tiup dm template

部署集群之前，需要准备一份集群的拓扑文件。TiUP 内置了拓扑文件的模版，用户可以通过修改模版来生成最终的拓扑文件。使用 tiup dm template 命令可以输出TiUP 内置的模版内容。

语法

```
tiup dm template [flags]
```

如果不指定该选项，输出的默认模版包含以下实例：

- 3 个 DM-master 实例

- 3 个 DM-worker 实例
- 1 个 Prometheus 实例
- 1 个 Grafana 实例
- 1 个 Alertmanager 实例

选项

–full

- 输出详细的拓扑模版，该模版会以注释的形式带上可配置的参数。在命令中添加该选项，可开启该选项。
- 如果不指定该选项，默认输出最简单的拓扑模版。

-h, –help

输出帮助信息。

输出

根据指定选项输出拓扑模版到标准输出，可重定向到拓扑文件中用于部署。

<< 返回上一页 - TiUP DM 命令清单

10.4.7.4.22 tiup dm upgrade

命令 `tiup dm upgrade` 用于将指定集群升级到特定版本。

语法

```
tiup dm upgrade <cluster-name> <version> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表。
- `<version>` 为要升级到的目标版本，例如 `v6.5.0`。目前仅允许升级到比当前集群更高的版本，不允许升级到比当前集群更低的版本，即不允许降级。同时也不允许升级成nightly 版本

选项

–offline

声明当前集群处于离线状态。指定该选项时，TiUP DM 仅原地替换集群组件的二进制文件，不重启服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

升级服务的日志。

<< 返回上一页 - TiUP DM 命令清单

10.4.8 通过TiUP 部署TiDB 集群的拓扑文件配置

通过TiUP 部署或扩容TiDB 集群时，需要提供一份拓扑文件（示例）来描述集群拓扑。

同样，修改集群配置也是通过编辑拓扑文件来实现的，区别在于修改配置时仅允许修改部分字段。本文档介绍拓扑文件的各个区块以各区块中的各字段。

10.4.8.1 文件结构

一个通过TiUP 部署的TiDB 集群拓扑文件可能包含以下区块：

- global：集群全局配置，其中一些是集群的默认值，可以在实例里面单独配置
- monitored：监控服务配置，即blackbox exporter 和 node exporter，每台机器上都会部署一个 node exporter 和一个 blackbox exporter
- server_configs：组件全局配置，可单独针对每个组件配置，若在实例中存在同名配置项，那么以实例中配置的为准
- pd_servers：PD 实例的配置，用来指定 PD 组件部署到哪些机器上
- tidb_servers：TiDB 实例的配置，用来指定TiDB 组件部署到哪些机器上
- tikv_servers：TiKV 实例的配置，用来指定TiKV 组件部署到哪些机器上
- tiflash_servers：TiFlash 实例的配置，用来指定TiFlash 组件部署到哪些机器上
- pump_servers：Pump 实例的配置，用来指定 Pump 组件部署到哪些机器上
- drainer_servers：Drainer 实例的配置，用来指定 Drainer 组件部署到哪些机器上
- cdc_servers：CDC 实例的配置，用来指定 CDC 组件部署到哪些机器上
- tispark_masters：TiSpark Master 实例的配置，用来指定TiSpark Master 组件部署到哪台机器上，仅允许部署一个TiSpark Master 节点
- tispark_workers：TiSpark Worker 实例的配置，用来指定TiSpark Worker 组件部署到哪些机器上
- monitoring_servers：用来指定 Prometheus 和 NGMonitoring 部署在哪些机器上，TiUP 支持部署多台 Prometheus 实例，但真实投入使用的只有第一个
- grafana_servers：Grafana 实例的配置，用来指定Grafana 部署在哪台机器上
- alertmanager_servers：Alertemanager 实例的配置，用来指定Alertmanager 部署在哪些机器上

10.4.8.1.1 global

global 区块为集群的全局配置，包含以下字段：

- user：以什么用户来启动部署的集群，默认值：“tidb”，如果 <user> 字段指定的用户在目标机器上不存在，会自动尝试创建
- group：自动创建用户时指定用户所属的用户组，默认和 <user> 字段值相同，若指定的组不存在，则自动创建
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，默认值：22
- enable_tls：是否对集群启用 TLS。启用之后，组件之间、客户端与组件之间都必须使用生成的 TLS 证书进行连接，默认值：false
- deploy_dir：每个组件的部署目录，默认值：“deploy”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 deploy_dir，那么实际部署目录为该实例设定的 deploy_dir
  - 对于每个实例，如果用户未配置 deploy_dir，其默认值为相对路径 <component-name>-<component-port>

  - 如果 global.deploy_dir 为绝对路径， 那么组件会部署到 <global.deploy_dir>/<instance.deploy_dir> 目录
  - 如果 global.deploy_dir 为相对路径，那么组件会部署到 /home/<global.user>/<global.deploy_dir>/<instance.deploy_dir> 目录
- data_dir：数据目录，默认值：“data”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 data_dir，那么实际数据目录为该实例设定的 data_dir
  - 对于每个实例，如果用户未配置 data_dir，其默认值为 <global.data_dir>
  - 如果 data_dir 为相对路径，那么组件数据将放到 <deploy_dir>/<data_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- log_dir：日志目录，默认值：“log”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 log_dir，那么实际日志目录为该实例设定的 log_dir
  - 对于每个实例，如果用户未配置 log_dir，其默认值为 <global.log_dir>
  - 如果 log_dir 为相对路径，那么组件日志将放到 <deploy_dir>/<log_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- os：目标机器的操作系统，该字段决定了向目标机器推送适配哪个操作系统的组件，默认值：linux
- arch：目标机器的 CPU 架构，该字段决定了向目标机器推送哪个平台的二进制包，支持amd64 和arm64，默认值：amd64
- resource_control：运行时资源控制，该字段下所有配置都将写入 systemd 的 service 文件中，默认无限制。支持控制的资源如下：
  - memory_limit:限制运行时最大内存，例如“2G”表示最多使用 2GB 内存
  - cpu_quota：限制运行时最大 CPU 占用率，例如“200%”
  - io_read_bandwidth_max：读磁盘 I/O 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
  - io_write_bandwidth_max：写磁盘 I/O 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
  - limit_core：控制 core dump 的大小

global 配置示例：

```
global:
  user: "tidb"
  resource_control:
    memory_limit: "2G"
```

上述配置指定使用 tidb 用户启动集群，同时限制每个组件运行时最多只能使用 2GB 内存。

10.4.8.1.2 monitored

monitored 用于配置目标机上的监控服务：node_exporter 和blackbox_exporter。包含以下字段：

- node_exporter_port：node_exporter 的服务端口，默认值：9100
- blackbox_exporter_port：blackbox_exporter 的服务端口，默认值：9115
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成

- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成

monitored 配置示例：

```
monitored:
  node_exporter_port: 9100
  blackbox_exporter_port: 9115
```

上述配置指定了node_exporter 使用 9100 端口，blackbox_exporter 使用 9115 端口。

10.4.8.1.3 server_configs

server_configs 用于配置服务，生成各组件的配置文件，类似 global 区块，该区块内的配置可以在具体的实例中被覆盖。主要包含以下字段：

- tidb：TiDB 服务的相关配置，支持的完整配置请参考TiDB 配置文件描述
- tikv：TiKV 服务的相关配置，支持的完整配置请参考TiKV 配置文件描述
- pd：PD 服务的相关配置，支持的完整配置请参考 PD 配置文件描述
- tiflash：TiFlash 服务的相关配置，支持的完整配置请参考TiFlash 配置参数
- tiflash_learner：每个 TiFlash 中内置了一个特殊的TiKV，该配置项用于配置这个特殊的TiKV，一般不建议修改这个配置项下的内容
- pump：Pump 服务的相关配置，支持的完整配置请参考TiDB Binlog 配置说明
- drainer：Drainer 服务的相关配置，支持的完整配置请参考TiDB Binlog 配置说明
- cdc：CDC 服务的相关配置，支持的完整配置请参考TiCDC 安装部署

server_configs 配置示例：

```
server_configs:
  tidb:
    lease: "45s"
    split-table: true
    token-limit: 1000
    instance.tidb_enable_ddl: true
  tikv:
    log-level: "info"
    readpool.unified.min-thread-count: 1
```

上述配置指定了TiDB 和 TiKV 的全局配置。

10.4.8.1.4 pd_servers

pd_servers 指定了将 PD 的服务部署到哪些机器上，同时可以指定每台机器上的服务配置。pd_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略

- listen_host：当机器上有多个 IP 时，可以指定服务的监听IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 PD 实例的名字，不同实例的名字必须唯一，否则无法部署
- client_port：指定 PD 的客户端连接端口，默认值：2379
- peer_port：指定 PD 之间互相通信的端口，默认值：2380
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl。在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如“0,1”
- config：该字段配置规则和 server_configs 里的 pd 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 pd 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制。如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- name
- client_port
- peer_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

pd_servers 配置示例：

```
pd_servers:
  - host: 10.0.1.11
    config:
      schedule.max-merge-region-size: 20
      schedule.max-merge-region-keys: 200000
  - host: 10.0.1.12
```

上述配置指定了将 PD 部署到 10.0.1.11 和 10.0.1.12，并针对 10.0.1.11 的 PD 进行一些特殊配置。

10.4.8.1.5 tidb_servers

tidb_servers 指定了将TiDB 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。tidb_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiDB 服务的监听端口，用于提供给 MySQL 客户端连接，默认值：4000
- status_port：TiDB 状态服务的监听端口，用于外部通过http 请求查看TiDB 服务的状态，默认值：10080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如“0,1”
- config：该字段配置规则和 server_configs 里的 tidb 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tidb 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- status_port
- deploy_dir
- log_dir
- arch
- os

tidb_servers 配置示例：

```
tidb_servers:
  - host: 10.0.1.14
    config:
      log.level: warn
      log.slow-query-file: tidb-slow-overwrited.log
  - host: 10.0.1.15
```

10.4.8.1.6 tikv_servers

tikv_servers 约定了将TiKV服务部署到哪些机器上，同时可以指定每台机器上的服务配置。tikv_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiKV 服务的监听端口，默认值：20160
- status_port：TiKV 状态服务的监听端口，默认值：20180
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如“0,1”
- config：该字段配置规则和 server_configs 里的tikv 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tikv 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- status_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

tikv_servers 配置示例：

```
tikv_servers:
  - host: 10.0.1.14
    config:
      server.labels: { zone: "zone1", host: "host1" }
  - host: 10.0.1.15
    config:
      server.labels: { zone: "zone1", host: "host2" }
```

10.4.8.1.7 tiflash_servers

tiflash_servers 约定了将TiFlash 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。该区块是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- tcp_port：TiFlash TCP 服务的端口，默认 9000
- flash_service_port：TiFlash 提供服务的端口，TiDB 通过该端口从TiFlash 读数据，默认 3930
- metrics_port：TiFlash 的状态端口，用于输出metric 数据，默认 8234
- flash_proxy_port：内置 TiKV 的端口，默认 20170
- flash_proxy_status_port：内置 TiKV 的状态端口，默认为 20292
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照global 中配置的data_dir 生成，TiFlash的数据目录支持多个，采用逗号分割
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- tmp_path: TiFlash 临时文件的存放路径，默认使用[path 或者storage.latest.dir 的第一个目录]+ “/tmp”
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 tiflash 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tiflash 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- learner_config：每个 TiFlash 中内置了一个特殊的TiKV 模块，该配置项用于配置这个特殊的TiKV 模块，一般不建议修改这个配置项下的内容
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上字段中，data_dir 在部署完成之后只能新增目录，而下列字段部署完成之后不能再修改：

- host
- tcp_port
- http_port
- flash_service_port
- flash_proxy_port
- flash_proxy_status_port
- metrics_port
- deploy_dir
- log_dir
- tmp_path
- arch
- os

tiflash_servers 配置示例：

```
tiflash_servers:
  - host: 10.0.1.21
  - host: 10.0.1.22
```

10.4.8.1.8 pump_servers

pump_servers 约定了将TiDB Binlog 组件的 Pump 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。pump_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Pump 服务的监听端口，默认 8250
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 pump 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 pump 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

pump_servers 配置示例：

```
pump_servers:
  - host: 10.0.1.21
    config:
      gc: 7
  - host: 10.0.1.22
```

10.4.8.1.9 drainer_servers

drainer_servers 约定了将TiDB Binlog 组件的 Drainer 服务部署到哪些机器上，同时可以指定每台机器上的服务配置，drainer_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Drainer 服务的监听端口，默认 8249
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- commit_ts：[已废弃]Drainer 启动的时候会去读取checkpoint，如果读取不到，就会使用该字段做为初次启动开始的同步时间点，该字段默认为-1（从 PD 总获取最新时间戳作为commit_ts）
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如“0,1”
- config：该字段配置规则和 server_configs 里的 drainer 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 drainer 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

其中 commit_ts 字段自tiup cluster v1.9.2 开始已经废弃，不再被记录到drainer 的启动脚本中。如果你仍需要配置该参数，请参照下面的示例在 config 中配置 initial-commit-ts 字段。

drainer_servers 配置示例：

```
drainer_servers:
  - host: 10.0.1.21
    config:
      initial-commit-ts: -1
      syncer.db-type: "mysql"
      syncer.to.host: "127.0.0.1"
```

```
    syncer.to.user: "root"
    syncer.to.password: ""
    syncer.to.port: 3306
    syncer.ignore-table:
      - db-name: test
        tbl-name: log
      - db-name: test
        tbl-name: audit
```

10.4.8.1.10 cdc_servers

cdc_servers 约定了将TiCDC 服务部署到哪些机器上，同时可以指定每台机器上的服务配置，cdc_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiCDC 服务的监听端口，默认 8300
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录。若不指定该字段或指定为相对目录，数据目录则按照 global 中配置的 data_dir 生成。
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- gc-ttl：TiCDC 在 PD 设置的服务级别GC safepoint 的 TTL (Time To Live) 时长，单位为秒，默认值为 86400，即 24 小时
- tz：TiCDC 服务使用的时区。TiCDC 在内部转换timestamp 等时间数据类型和向下游同步数据时使用该时区，默认为进程运行本地时区。
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如“0,1”
- config：该字段配置规则和 server_configs 里的 cdc 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control
- ticdc_cluster_id：指定该服务器对应的 TiCDC 集群ID。若不指定该字段，则自动加入默认TiCDC 集群。该配置只在 v6.3.0 及以上TiDB 版本中才生效。

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir

- log_dir
- arch
- os
- ticdc_cluster_id

cdc_servers 配置示例：

```
cdc_servers:
  - host: 10.0.1.20
    gc-ttl: 86400
    data_dir: "/cdc-data"
  - host: 10.0.1.21
    gc-ttl: 86400
    data_dir: "/cdc-data"
```

10.4.8.1.11 tispark_masters

tispark_masters 约定了将TiSpark 的 master 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，tispark_masters 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Spark 的监听端口，节点之前通讯用，默认 7077
- web_port：Spark 的网页端口，提供网页服务，可查看任务情况，默认 8080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- java_home：指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME
- spark_config：用于配置 TiSpark 服务，生成配置文件并下发到 host 指定的机器
- spark_env：配置 Spark 启动时的环境变量
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- web_port
- deploy_dir
- arch
- os

tispark_masters 配置示例：

```
tispark_masters:
  - host: 10.0.1.21
    spark_config:
      spark.driver.memory: "2g"
      spark.eventLog.enabled: "False"
      spark.tispark.grpc.framesize: 2147483647
      spark.tispark.grpc.timeout_in_sec: 100
      spark.tispark.meta.reload_period_in_sec: 60
      spark.tispark.request.command.priority: "Low"
      spark.tispark.table.scan_concurrency: 256
    spark_env:
      SPARK_EXECUTOR_CORES: 5
      SPARK_EXECUTOR_MEMORY: "10g"
      SPARK_WORKER_CORES: 5
      SPARK_WORKER_MEMORY: "10g"
  - host: 10.0.1.22
```

10.4.8.1.12 tispark_workers

tispark_workers 约定了将TiSpark 的 worker 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，tispark_workers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Spark 的监听端口，节点之前通讯用，默认 7077
- web_port：Spark 的网页端口，提供网页服务，可查看任务情况，默认 8080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- java_home：指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- web_port
- deploy_dir
- arch
- os

tispark_workers 配置示例：

```
tispark_workers:
  - host: 10.0.1.22
  - host: 10.0.1.23
```

10.4.8.1.13 monitoring_servers

monitoring_servers 约定了将 Prometheus 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，monitoring_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，必填项
- ng_port：指定 NgMonitoring 组件监听的端口，在TiUP v1.7.0 引入，用于支持TiDB Dashboard 中持续性能分析和Top SQL 功能。默认值：12020
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Prometheus 提供服务的端口，默认值：9090
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- storage_retention：Prometheus 监控数据保留时间，默认 “30d”
- rule_dir：该字段指定一个本地目录，该目录中应当含有完整的 *.rules.yml 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Prometheus 的规则
- remote_config：用于支持将 Prometheus 数据写到远端，或从远端读取数据，该字段下有两个配置：
  - remote_write：参考 Prometheus <remote_write> 文档
  - remote_read：参考 Prometheus <remote_read> 文档
- external_alertmanagers：若配置了 external_alertmanagers，Prometheus 会将配置行为报警通知到集群外的Alertmanager。该字段为一个数组，数组的元素为每个外部的 Alertmanager，由 host 和 web_port 字段构成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir

- arch
- os

monitoring_servers 配置示例：

```
monitoring_servers:
  - host: 10.0.1.11
    rule_dir: /local/rule/dir
    remote_config:
      remote_write:
      - queue_config:
          batch_send_deadline: 5m
          capacity: 100000
          max_samples_per_send: 10000
          max_shards: 300
        url: http://127.0.0.1:8003/write
      remote_read:
      - url: http://127.0.0.1:8003/read
    external_alertmanagers:
    - host: 10.1.1.1
      web_port: 9093
    - host: 10.1.1.2
      web_port: 9094
```

10.4.8.1.14 grafana_servers

grafana_servers 约定了将 Grafana 服务部署到哪台机器上， 同时可以指定这台机器上的服务配置，grafana_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Grafana 提供服务的端口，默认值：3000
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- username：Grafana 登录界面的用户名
- password：Grafana 对应的密码
- dashboard_dir：该字段指定一个本地目录，该目录中应当含有完整的 dashboard(*.json) 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为Grafana 的 dashboards
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

注意：

如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：

1. 对于本地的dashboards 目录中的 *.json 文件，将 datasource 字段的值更新为新的集群名（这是因为 datasource 是以集群名命名的）
2. 执行 tiup cluster reload -R grafana 命令

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- arch
- os

grafana_servers 配置示例：

```
grafana_servers:
  - host: 10.0.1.11
    dashboard_dir: /local/dashboard/dir
```

10.4.8.1.15 alertmanager_servers

alertmanager_servers 约定了将Alertmanager 服务部署到哪些机器上，同时可以指定这台机器上的服务配置，alertmanager_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- web_port：指定 Alertmanager 提供网页服务的端口，默认值：9093
- cluster_port：指定 Alertmanger 和其他Alertmanager 通讯的端口，默认值：9094
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如“0,1”
- config_file：该字段指定一个本地文件，该文件会在集群配置初始化阶段被传输到目标机器上，作为 Alertmanager 的配置
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

- `resource_control`： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 `global` 中的 `resource_control` 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 `host` 指定机器。`resource_control` 的配置规则同 `global` 中的 `resource_control`

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- `host`
- `web_port`
- `cluster_port`
- `deploy_dir`
- `data_dir`
- `log_dir`
- `arch`
- `os`

`alertmanager_servers` 配置示例：

```
alertmanager_servers:
  - host: 10.0.1.11
    config_file: /local/config/file
  - host: 10.0.1.12
    config_file: /local/config/file
```

10.4.9 通过TiUP 部署 DM 集群的拓扑文件配置

在部署或扩容TiDB Data Migration (DM) 集群时，需要提供一份拓扑文件来描述集群拓扑，同样，修改配置也是通过编辑拓扑文件来实现的，区别在于修改配置时仅允许修改部分字段。

拓扑文件示例参考。

10.4.9.1 文件结构

一个 DM 集群的拓扑文件可能包含以下区块：

- global：集群全局配置，其中一些是集群的默认值，可以在实例里面单独配置
- server_configs：组件全局配置，可单独针对每个组件配置，若在实例中存在同名配置项，那么以实例中配置的为准
- master_servers：DM master 实例的配置，用来指定 DM 组件的master 服务部署到哪些机器上
- worker_servers：DM worker 实例的配置，用来指定 DM 组件的 worker 服务部署到哪些机器上
- monitoring_servers：用来指定 Prometheus 部署在哪机器上，TiUP 支持部署多台 Prometheus 实例，但真实投入使用的只有第一个
- grafana_servers：Grafana 实例的配置，用来指定Grafana 部署在哪台机器上
- alertmanager_servers：Alertemanager 实例的配置，用来指定Alertmanager 部署在哪些机器上

10.4.9.1.1 global

global 区块为集群的全局配置，包含以下字段：

- user：以什么用户来启动部署的集群，默认值：“tidb”，如果 <user> 字段指定的用户在目标机器上不存在，会自动尝试创建
- group：自动创建用户时指定用户所属的用户组，默认和 <user> 字段值相同，若指定的组不存在，则自动创建
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，默认值：22
- deploy_dir：每个组件的部署目录，默认值：“deploy”，其应用规则如下：
  - 如果在实例级别配置了绝对路径的 deploy_dir，那么实际部署目录为该实例设定的 deploy_dir
  - 对于每个实例，如果用户未配置 deploy_dir，其默认值为相对路径 <component-name>-<component-port>
  - 如果 global.deploy_dir 为绝对路径，那么组件会部署到 <global.deploy_dir>/<instance.deploy_dir> 目录
  - 如果 global.deploy_dir 为相对路径，那么组件会部署到 /home/<global.user>/<global.deploy_dir>/<instance.deploy_dir> 目录
- data_dir：数据目录，默认值：“data”，其应用规则如下：
  - 如果在实例级别配置了绝对路径的 data_dir，那么实际数据目录为该实例设定的 data_dir
  - 对于每个实例，如果用户未配置 data_dir，其默认值为 <global.data_dir>
  - 如果 data_dir 为相对路径，那么组件数据将放到 <deploy_dir>/<data_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- log_dir：数据目录，默认值：“log”，其应用规则如下：
  - 如果在实例级别配置了绝对路径的 log_dir，那么实际日志目录为该实例设定的 log_dir
  - 对于每个实例，如果用户未配置 log_dir，其默认值为 <global.log_dir>
  - 如果 log_dir 为相对路径，那么组件日志将放到 <deploy_dir>/<log_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- os：目标机器的操作系统，该字段决定了向目标机器推送适配哪个操作系统的组件，默认值：linux
- arch：目标机器的 CPU 架构，该字段决定了向目标机器推送哪个平台的二进制包，支持amd64 和arm64，默认值：amd64
- resource_control：运行时资源控制，该字段下所有配置都将写入 systemd 的 service 文件中，默认不限制，支持控制的资源：
  - memory_limit:限制运行时最大内存，例如“2G”表示最多使用 2GB 内存
  - cpu_quota：限制运行时最大 CPU 占用率，例如“200%”
  - io_read_bandwidth_max：读磁盘 IO 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
  - io_write_bandwidth_max：写磁盘 IO 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
  - limit_core：控制 core dump 大小

global 配置示例：

```
global:
  user: "tidb"
```

```
  resource_control:
    memory_limit: "2G"
```

上述配置指定使用 tidb 用户启动集群，同时限制每个组件运行时最多只能使用 2GB 内存。

10.4.9.1.2 server_configs

server_configs 用于配置服务，生成各组件的配置文件，类似 global 区块，该区块内的配置可以在具体的实例中被覆盖。主要包含以下字段：

- master：DM master 服务的相关配置，支持的完整配置请参考DM-master 配置文件介绍
- worker：DM worker 服务的相关配置，支持的完整配置请参考DM-worker 配置文件介绍

server_configs 配置示例：

```
server_configs:
  master:
    log-level: info
    rpc-timeout: "30s"
    rpc-rate-limit: 10.0
    rpc-rate-burst: 40
  worker:
    log-level: info
```

10.4.9.2 master_servers

master_servers 约定了将 DM 组件的master 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，master_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 DM master 实例的名字，不同实例的名字必须唯一，否则无法部署
- port：指定 DM master 提供给服务的端口，默认 8261
- peer_port：指定 DM master 之间互相通信的端口，默认值：8291
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如 “0,1”
- config：该字段配置规则和 server_configs 里的master 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的master 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os

- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control
- v1_source_path：从 v1.0.x 升级时，可指定该目录，该目录中应当存放有V1 的源的配置文件

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- name
- port
- peer_port
- deploy_dir
- data_dir
- log_dir
- arch
- os
- v1_source_path

master_servers 配置示例：

```
master_servers:
  - host: 10.0.1.11
    name: master1
     ssh_port: 22
    port: 8261
    peer_port: 8291
    deploy_dir: "/dm-deploy/dm-master-8261"
    data_dir: "/dm-data/dm-master-8261"
    log_dir: "/dm-deploy/dm-master-8261/log"
    numa_node: "0,1"
    # The following configs are used to overwrite the `server_configs.master` values.
    config:
      log-level: info
      rpc-timeout: "30s"
      rpc-rate-limit: 10.0
      rpc-rate-burst: 40
  - host: 10.0.1.18
    name: master2
  - host: 10.0.1.19
    name: master3
```

10.4.9.3 worker_servers

worker_servers 约定了将 DM 组件的 worker 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，worker_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 DM worker 实例的名字，不同实例的名字必须唯一，否则无法部署
- port：指定 DM worker 提供给服务的端口，默认 8262
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如 “0,1”
- config：该字段配置规则和 server_configs 里的 worker 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 worker 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- name
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

worker_servers 配置示例：

```
worker_servers:
  - host: 10.0.1.12
    ssh_port: 22
    port: 8262
    deploy_dir: "/dm-deploy/dm-worker-8262"
    log_dir: "/dm-deploy/dm-worker-8262/log"
    numa_node: "0,1"
    # config is used to overwrite the `server_configs.worker` values
    config:
      log-level: info
  - host: 10.0.1.19
```

10.4.9.3.1 monitoring_servers

monitoring_servers 约定了将 Prometheus 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，monitoring_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Prometheus 提供服务的端口，默认值：9090
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如“0,1”
- storage_retention：Prometheus 监控数据保留时间，默认“15d”
- rule_dir：该字段指定一个本地目录，该目录中应当含有完整的 *.rules.yml 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Prometheus 的规则
- remote_config：用于支持将 Prometheus 数据写到远端，或从远端读取数据，该字段下有两个配置：
  - remote_write：参考 Prometheus <remote_write> 文档
  - remote_read：参考 Prometheus <remote_read> 文档
- external_alertmanagers：若配置了 external_alertmanagers，Prometheus 会将配置行为报警通知到集群外的Alertmanager。该字段为一个数组，数组的元素为每个外部的 Alertmanager，由 host 和 web_port 字段构成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

monitoring_servers 配置示例：

```
monitoring_servers:
  - host: 10.0.1.11
    rule_dir: /local/rule/dir
```

```
  remote_config:
    remote_write:
    - queue_config:
        batch_send_deadline: 5m
        capacity: 100000
        max_samples_per_send: 10000
        max_shards: 300
      url: http://127.0.0.1:8003/write
    remote_read:
    - url: http://127.0.0.1:8003/read
  external_alertmanagers:
  - host: 10.1.1.1
    web_port: 9093
  - host: 10.1.1.2
    web_port: 9094
```

10.4.9.3.2 grafana_servers

grafana_servers 约定了将 Grafana 服务部署到哪台机器上， 同时可以指定这台机器上的服务配置，grafana_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Grafana 提供服务的端口，默认值：3000
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- username：Grafana 登录界面的用户名
- password：Grafana 对应的密码
- dashboard_dir：该字段指定一个本地目录，该目录中应当含有完整的 dashboard(*.json) 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为Grafana 的 dashboards
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global中的 resource_control

> 注意：
>
> 如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：
>
> 1. 在本地的dashboards 目录中，将datasource 字段的值更新为新的集群名（datasource 是以集群名命名的）
> 2. 执行 tiup cluster reload -R grafana 命令

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- arch
- os

grafana_servers 配置示例：

```
grafana_servers:
  - host: 10.0.1.11
    dashboard_dir: /local/dashboard/dir
```

10.4.9.3.3 alertmanager_servers

alertmanager_servers 约定了将Alertmanager 服务部署到哪些机器上，同时可以指定这台机器上的服务配置，alertmanager_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- web_port：指定 Alertmanager 提供网页服务的端口，默认值：9093
- cluster_port：指定 Alertmanager 和其他Alertmanager 通讯的端口，默认值：9094
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了numactl，在指定该参数的情况下会通过numactl 分配cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如“0,1”
- config_file：该字段指定一个本地文件，该文件会在集群配置初始化阶段被传输到目标机器上，作为 Alertmanager 的配置
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control： 针对该服务的资源控制， 如果配置了该字段， 会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- web_port
- cluster_port
- deploy_dir
- data_dir

- log_dir
- arch
- os

alertmanager_servers 配置示例：

```
alertmanager_servers:
  - host: 10.0.1.11
    config_file: /local/config/file
  - host: 10.0.1.12
    config_file: /local/config/file
```

10.4.10 TiUP 镜像参考指南

TiUP 镜像是TiUP 的组件仓库，存放了一系列的组件和这些组件的元信息。镜像有两种存在形式：

- 本地磁盘上的目录：用于服务本地的TiUP 客户端，文档中将称之为本地镜像
- 基于远程的磁盘目录启动的 HTTP 镜像：服务远程的TiUP 客户端，文档中将称之为远程镜像

10.4.10.1 镜像的创建与更新

镜像可以通过以下两种方式创建：

- 通过命令 tiup mirror init 从零生成
- 通过命令 tiup mirror clone 从已有镜像克隆

在创建镜像之后，可以通过 tiup mirror 相关命令来给镜像添加组件或删除组件，无论是通过何种方式更新镜像，TiUP 都不会从镜像中删除任何文件，而是通过增加文件并分配新版本号的方式更新。

10.4.10.2 镜像结构

一个典型的镜像目录结构如下：

```
+ <mirror-dir>                                  # 镜像根目录
|-- root.json                                   # 镜像根证书
|-- {2..N}.root.json                            # 镜像根证书
|-- {1..N}.index.json                           # 组件/用户索引
|-- {1..N}.{component}.json                     # 组件元信息
|-- {component}-{version}-{os}-{arch}.tar.gz    # 组件二进制包
|-- snapshot.json                               # 镜像最新快照
|-- timestamp.json                              # 镜像最新时间戳
|--+ commits                                    # 镜像更新日志（可删除）
   |--+ commit-{ts1..tsN}
      |-- {N}.root.json
      |-- {N}.{component}.json
      |-- {N}.index.json
```

```
      |-- {component}-{version}-{os}-{arch}.tar.gz
      |-- snapshot.json
      |-- timestamp.json
|--+ keys                                          # 镜像私钥（可移动到其他位置）
   |-- {hash1..hashN}-root.json                    # 根证书私钥
   |-- {hash}-index.json                           # 索引私钥
   |-- {hash}-snapshot.json                        # 快照私钥
   |-- {hash}-timestamp.json                       # 时间戳私钥
```

注意：

- commits 目录是在更新镜像过程中产生的日志，用于回滚镜像，磁盘空间不足时可以定期删除旧的文件夹
- keys 文件夹中存放的私钥较敏感，建议单独妥善保管

10.4.10.2.1 根证书

在TiUP 镜像中，根证书用于存放其他元信息文件的公钥，每次获取到任何元信息文件（*.json）都需要根据其文件类型（root，index，snapshot，timestamp）在当前已安装的 root.json 中找到对应的公钥，然后用公钥验证其签名是否合法。

根证书文件格式如下：

```
{
    "signatures": [                                 # 每个元信息文件有一系列的签名，
        ↪ 签名由该文件对应的几个私钥签出
        {
            "keyid": "{id-of-root-key-1}",          # 第一个参与签名私钥的 ID，该 ID
                ↪ 由私钥对应的公钥内容哈希得到
            "sig": "{signature-by-root-key-1}"      # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",          # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"      # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {                                     # 被签名的部分
        "_type": "root",                            # 该字段说明本文件的类型，root.
            ↪ json 的类型就是 root
        "expires": "{expiration-date-of-this-file}",# 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
```

```
        "roles": {                                              # root.json 中的 roles
            ↪ 用来记录对各个元文件签名的密钥
            "{role:index,root,snapshot,timestamp}": {           # 涉及的元文件类型包括 index,
                ↪ root, snapshot, timestamp
                "keys": {                                       # 只有 keys
                    ↪ 中记录的密钥签名才是合法的
                    "{id-of-the-key-1}": {                      # 用于签名 {role} 的第 1 个密钥
                        ↪ ID
                        "keytype": "rsa",                       # 密钥类型，目前固定为 rsa
                        "keyval": {                             # 密钥的 payload
                            "public": "{public-key-content}"    # 表示公钥内容
                        },
                        "scheme": "rsassa-pss-sha256"           # 目前固定为 rsassa-pss-sha256
                    },
                    "{id-of-the-key-N}": {                      # 用于签名 {role} 的第 N 个密钥
                        ↪ ID
                        "keytype": "rsa",
                        "keyval": {
                            "public": "{public-key-content}"
                        },
                        "scheme": "rsassa-pss-sha256"
                    }
                },
                "threshold": {N},                               # threshold 指示该元文件需要至少
                    ↪ N 个密钥签名
                "url": "/{role}.json"                           # url 是指该文件的获取地址，对于

                    ↪ index 文件，需要在前面加上版本号，即 /{N}.index.json
            }
        },
        "spec_version": "0.1.0",                                # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                          # 本文件的版本号，
            ↪ 每次更新文件需要创建一个新的 {N+1}.root.json，并将其 version 设置为 N + 1
    }
}
```

10.4.10.2.2 索引

索引文件记录了镜像中所有的组件以及组件的所有者信息。

其格式如下：

```
{
    "signatures": [                                             # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                     # 第一个参与签名的 key 的 ID
```

```
        "sig": "{signature-by-index-key-1}",                # 该私钥对此文件 signed
            ↪ 部分签名的结果
    },
    ...
    {
        "keyid": "{id-of-root-key-N}",                      # 第 N 个参与签名私钥的 ID
        "sig": "{signature-by-root-key-N}"                  # 该私钥对此文件 signed
            ↪ 部分签名的结果
    }
],
"signed": {
    "_type": "index",                                       # 指示该文件类型
    "components": {                                         # 组件列表
        "{component1}": {                                   # 第一个组件的名称
            "hidden": {bool},                               # 是否是隐藏组件
            "owner": "{owner-id}",                          # 组件管理员 ID
            "standalone": {bool},                           # 该组件是否可独立运行
            "url": "/{component}.json",                     # 获取组件的地址，需要加上版本号
                ↪ : /{N}.{component}.json
            "yanked": {bool}                                # 该组件是否已被标记为删除
        },
        ...
        "{componentN}": {                                   # 第 N 个组件的名称
            ...
        },
    },
    "default_components": ["{component1}.."{componentN}"], # 镜像必须包含的默认组件，
        ↪ 该字段目前固定为空（未启用）
    "expires": "{expiration-date-of-this-file}",            # 该文件的过期时间，
        ↪ 过期后客户端会拒绝此文件
    "owners": {
        "{owner1}": {                                       # 第一个属主的 ID
            "keys": {                                       # 只有 keys
                ↪ 中记录的密钥签名才是合法的
                "{id-of-the-key-1}": {                      # 该属主的第一个密钥
                    "keytype": "rsa",                       # 密钥类型，目前固定为 rsa
                    "keyval": {                             # 密钥的 payload
                        "public": "{public-key-content}"    # 表示公钥内容
                    },
                    "scheme": "rsassa-pss-sha256"           # 目前固定为 rsassa-pss-sha256
                },
                ...
                "{id-of-the-key-N}": {                      # 该属主的第 N 个密钥
                    ...
                }
```

```
                },
                "name": "{owner-name}",                          # 该属主的名字
                "threshod": {N}                                  #
                    '→ 指示该属主拥有的组件必须含有至少 N 个合法签名
            },
            ...
            "{ownerN}": {                                        # 第 N 个属主的 ID
                ...
            }
        }
        "spec_version": "0.1.0",                                 # 本文件遵循的规范版本，
            '→ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                           # 本文件的版本号，
            '→ 每次更新文件需要创建一个新的 {N+1}.index.json，并将其 version 设置为 N + 1
    }
}
```

10.4.10.2.3 组件

组件元信息文件记录了特定组件的平台以及版本信息。

其格式如下：

```
{
    "signatures": [                                              # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                      # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                 # 该私钥对此文件 signed
                '→ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                       # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                   # 该私钥对此文件 signed
                '→ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "component",                                    # 指示该文件类型
        "description": "{description-of-the-component}",         # 该组件的描述信息
        "expires": "{expiration-date-of-this-file}",             # 该文件的过期时间，
            '→ 过期后客户端会拒绝此文件
        "id": "{component-id}",                                  # 该组件的 ID，具有全局唯一性
        "nightly": "{nightly-cursor}",                           # nightly 游标，值为最新的
            '→ nightly 的版本号（如 v5.0.0-nightly-20201209）
```

```
        "platforms": {                                          # 该组件支持的平台（如 darwin/
            ↪ amd64，linux/arm64 等）
            "{platform-pair-1}": {
                "{version-1}": {                                # Semantic Version 版本号（如 v1
                    ↪ .0.0 等）
                    "dependencies": null,                       # 用于约定组件之间的依赖关系，
                        ↪ 该字段尚未使用，固定为 null
                    "entry": "{entry}",                         # 入口二进制文件位于 tar
                        ↪ 包的相对路径
                    "hashs": {                                  # tar 包的 checksum，我们使用
                        ↪ sha256 和 sha512
                        "sha256": "{sum-of-sha256}",
                        "sha512": "{sum-of-sha512}",
                    },
                    "length": {length-of-tar},                  # tar 包的长度
                    "released": "{release-time}",               # 该版本的 release 时间
                    "url": "{url-of-tar}",                      # tar 包的下载地址
                    "yanked": {bool}                            # 该版本是否已被禁用
                }
            },
            ...
            "{platform-pair-N}": {
                ...
            }
        },
        "spec_version": "0.1.0",                                # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                          # 本文件的版本号，

            ↪ 每次更新文件需要创建一个新的 {N+1}.{component}.json，并将其 version 设置为 N + 1
}
```

10.4.10.2.4 快照

快照文件记录了各个元文件当前的版本号。

其格式如下：

```
{
    "signatures": [                                             # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                     # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
```

```
            "keyid": "{id-of-root-key-N}",                    # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "snapshot",                                  # 指示该文件类型
        "expires": "{expiration-date-of-this-file}",          # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "meta": {                                             # 其他元文件的信息
            "/root.json": {
                "length": {length-of-json-file},              # root.json 的长度
                "version": {version-of-json-file}             # root.json 的 version
            },
            "/index.json": {
                "length": {length-of-json-file},
                "version": {version-of-json-file}
            },
            "/{component-1}.json": {
                "length": {length-of-json-file},
                "version": {version-of-json-file}
            },
            ...
            "/{component-N}.json": {
                ...
            }
        },
        "spec_version": "0.1.0",                              # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": 0                                          # 本文件的版本号，固定为 0
    }
```

10.4.10.2.5 时间戳

时间戳文件记录了当前快照checksum。

其文件格式如下：

```
{
    "signatures": [                                           # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                   # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",              # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
```

```
        {
            "keyid": "{id-of-root-key-N}",                    # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                # 该私钥对此文件 signed
                ↪部分签名的结果
        }
    ],
    "signed": {
        "_type": "timestamp",                                 # 指示该文件类型
        "expires": "{expiration-date-of-this-file}",          # 该文件的过期时间，
            ↪过期后客户端会拒绝此文件
        "meta": {                                             # snapshot.json 的信息
            "/snapshot.json":
                { "hashes": {
                    "sha256": "{sum-of-sha256}"               # snapshot.json 的 sha256
                },
                "length": {length-of-json-file}               # snapshot.json 的长度
            }
        },
        "spec_version": "0.1.0",                              # 本文件遵循的规范版本，
            ↪未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                        # 本文件的版本号，

            ↪每次更新文件需要覆盖 timestamp.json，并将其 version 设置为 N + 1
```

10.4.10.3 客户端工作流程

客户端通过以下逻辑保证从镜像下载到的文件是安全的：

- 客户端安装时随binary 附带了一个root.json
- 客户端运行时以已有的root.json 为基础，做如下操作：
  1. 获取root.json 中的 version，记为 N
  2. 向镜像请求 {N+1}.root.json，若成功，使用 root.json 中记录的公钥验证该文件是否合法
- 向镜像请求timestamp.json，并使用root.json 中记录的公钥验证该文件是否合法
- 检查timestamp.json 中记录的 snapshot.json 的 checksum 和本地的 snapshot.json 的 checksum 是否吻合
  - 若不吻合，则向镜像请求最新的 snapshot.json 并使用root.json 中记录的公钥验证该文件是否合法
- 对于index.json 文件，从 snapshot.json 中获取其版本号N，并请求 {N}.index.json，然后使用 root.json 中记录的公钥验证该文件是否合法
- 对于组件（如 tidb.json，tikv.json），从 snapshot.json 中获取其版本号N，并请求 {N}.{component}.json，然后使用 index.json 中记录的公钥验证该文件是否合法
- 对于组件tar 文件，从 {component}.json 中获取其url 及 checksum，请求url 得到tar 包，并验证checksum 是否正确

10.4.11 TiUP 组件文档

10.4.11.1 本地快速部署TiDB 集群

TiDB 集群是由多个组件构成的分布式系统，一个典型的 TiDB 集群至少由 3 个 PD 节点、3 个TiKV 节点和 2 个TiDB 节点构成。对于想要快速体验TiDB 的用户来说，手工部署这么多组件是非常耗时且麻烦的事情。本文介绍 TiUP 中的playground 组件，以及如何通过playground 组件快速搭建一套本地的TiDB 测试环境。

10.4.11.1.1 playground 组件介绍

playground 组件的基本用法：

```
tiup playground ${version} [flags]
```

如果直接执行 `tiup playground` 命令，则 TiUP playground 会使用本地安装的TiDB/TiKV/PD 组件或者安装这些组件的稳定版本，来启动一个由 1 个 TiKV、1 个 TiDB、1 个 PD 和 1 个 TiFlash 实例构成的集群。该命令实际做了以下事情：

- 因为该命令没有指定playground 的版本，TiUP 会先查找已安装的playground 的最新版本，假设已安装的 playground 最新版为 v1.11.3，则该命令相当于 tiup playground:v1.11.3
- 如果playground 从未安装过任何版本的TiDB/TiKV/PD 组件，TiUP 会先安装这些组件的最新稳定版，然后再启动运行这些组件的实例
- 因为该命令没有指定TiDB/PD/TiKV 各组件的版本，默认情况下，它会使用各组件的最新发布版本，假设当前为 v7.0.0，则该命令相当于 tiup playground:1.11.3 v7.0.0
- 因为该命令也没有指定各组件的个数，默认情况下，它会启动由 1 个TiDB、1 个TiKV、1 个 PD 和 1 个TiFlash 实例构成的最小化集群
- 在依次启动完各个TiDB 组件后，playground 会提醒集群启动成功，并告诉你一些有用的信息，譬如如何通过 MySQL 客户端连接集群、如何访问TiDB Dashboard 等

playground 的命令行参数说明：

```
Flags:
      --db int                    设置集群中 TiDB 节点的数量（默认为1）
      --db.host host              指定 TiDB 的监听地址
      --db.port int               指定 TiDB 的端口号
      --db.binpath string         指定 TiDB 二进制文件的位置（开发调试用，可忽略）
      --db.config string          指定 TiDB 的配置文件（开发调试用，可忽略）
      --db.timeout int            指定 TiDB 最长等待超时时间，单位为秒。若配置为 0，则永不超时。
      --drainer int               设置集群中 Drainer 数据
      --drainer.binpath string    指定 Drainer 二进制文件的位置（开发调试用，可忽略）
      --drainer.config string     指定 Drainer 的配置文件
  -h, --help                      打印帮助信息
      --host string               设置每个组件的监听地址（默认为 127.0.0.1），
          ↪ 如果要提供给别的电脑访问，可设置为 0.0.0.0
      --kv int                    设置集群中 TiKV 节点的数量（默认为1）
      --kv.binpath string         指定 TiKV 二进制文件的位置（开发调试用，可忽略）
      --kv.config string          指定 TiKV 的配置文件（开发调试用，可忽略）
      --mode string               指定 playground 的运行模式，取值选项为 'tidb'（默认）和 'tikv-
          ↪ slim'
      --pd int                    设置集群中 PD 节点的数量（默认为1）
```

```
      --pd.host host                指定 PD 的监听地址
      --pd.binpath string           指定 PD 二进制文件的位置（开发调试用，可忽略）
      --pd.config string            指定 PD 的配置文件（开发调试用，可忽略）
      --pump int                    设置集群中 Pump 节点的数量（非 0 的时候 TiDB 会开启 TiDB Binlog
          ↪ ）
      --pump.binpath string         指定 Pump 二进制文件的位置（开发调试用，可忽略）
      --pump.config string          指定 Pump 的配置文件（开发调试用，可忽略）
  -T, --tag string                  设置 playground 的 tag 信息
      --ticdc int                   设置集群中 TiCDC 节点的数量（默认为 0）
      --ticdc.binpath string        指定 TiCDC 二进制文件的位置（开发调试用，可忽略）
      --ticdc.config string         指定 TiCDC 的配置文件（开发调试用，可忽略）
      --tiflash int                 设置集群中 TiFlash 节点的数量（默认为 1）
      --tiflash.binpath string      指定 TiFlash 的二进制文件位置（开发调试用，可忽略）
      --tiflash.config string       指定 TiFlash 的配置文件（开发调试用，可忽略）
      --tiflash.timeout int         指定 TiFlash 最长等待超时时间，单位为秒，若配置为 0，则永不超时
          ↪ 。
  -v, --version                     显示 playground 的版本号
      --without-monitor             设置不使用 Prometheus 和 Grafana 的监控功能。若不添加此参数，
          ↪ 则默认开启监控功能。
```

10.4.11.1.2 使用示例

查看可用的TiDB 版本

```
tiup list tidb
```

启动一个指定版本的TiDB 集群

```
tiup playground ${version}
```

将 `${version}` 替换为所需的版本号。

启动一个每日构建版的TiDB 集群

```
tiup playground nightly
```

nightly 就是这个集群的版本号，这里指定为每日构建版本。

覆盖 PD 的默认配置

首先，你需要复制 PD 的配置模版。假设你将复制的配置文件放置在 `~/config/pd.toml`，按需修改一些内容后，执行以下命令可以覆盖 PD 的默认配置：

```
tiup playground --pd.config ~/config/pd.toml
```

替换默认的二进制文件

默认启动playground 时，各个组件都是使用官方镜像组件包中的二进制文件启动的，如果本地编译了一个临时的二进制文件想要放入集群中测试，可以使用 `--{comp}.binpath` 这个参数替换，例如执行以下命令替换 TiDB 的二进制文件：

```
tiup playground --db.binpath /xx/tidb-server
```

启动多个组件实例

默认情况下各启动一个TiDB、TiKV 和 PD 实例，如果希望启动多个，可以加上如下参数：

```
tiup playground --db 3 --pd 3 --kv 3
```

启动集群时指定tag

Playground 集群在命令行退出时，会默认清空所有的集群数据。如果想要启动一个数据不被自动删除的 Playground 集群，需要在启动时指定集群tag，指定后可以在 `~/.tiup/data` 路径下找到该集群的数据。在集群启动时指定tag 的方法如下：

```
tiup playground --tag <tagname>
```

以这种方式启动的集群，在集群关闭以后，数据文件会保留。下一次可以继续使用该tag 启动集群，从而使用从上一次集群关闭时的数据。

10.4.11.1.3 快速连接到由playground 启动的TiDB 集群

TiUP 提供了 `client` 组件，用于自动寻找并连接playground 在本地启动的TiDB 集群，使用方式为：

```
tiup client
```

该命令会在控制台上提供当前机器上由playground 启动的TiDB 集群列表，选中需要连接的TiDB 集群，点击回车后，可以打开一个自带的 MySQL 客户端以连接TiDB。

10.4.11.1.4 查看已启动集群的信息

```
tiup playground display
```

可以看到如下信息：

```
Pid    Role     Uptime
---    ----     ------
84518  pd       35m22.929404512s
84519  tikv     35m22.927757153s
84520  pump     35m22.92618275s
86189  tidb     exited
86526  tidb     34m28.293148663s
86190  drainer  35m19.91349249s
```

10.4.11.1.5 扩容集群

扩容集群的命令行参数与启动集群的相似。以下命令可以扩容两个TiDB：

```
tiup playground scale-out --db 2
```

10.4.11.1.6 缩容集群

可在 `tiup playground scale-in` 命令中指定 `pid`，以缩容对应的实例。可以通过 `tiup playground display` 命令查看 `pid`。

```
tiup playground scale-in --pid 86526
```

10.4.11.2 使用TiUP 部署运维TiDB 线上集群

本文重在介绍如何使用TiUP 的cluster 组件，如果需要线上部署的完整步骤，可参考使用TiUP 部署TiDB 集群。

与 playground 组件用于部署本地测试集群类似，cluster 组件用于快速部署生产集群。对比playground，cluster 组件提供了更强大的生产集群管理功能，包括对集群的升级、缩容、扩容甚至操作、审计等。

cluster 组件的帮助文档如下：

```
tiup cluster
```

```
Starting component `cluster`: /home/tidb/.tiup/components/cluster/v1.11.3/cluster
Deploy a TiDB cluster for production

Usage:
  tiup cluster [command]

Available Commands:
  check       对集群进行预检
  deploy      部署集群
  start       启动已部署的集群
  stop        停止集群
  restart     重启集群
  scale-in    集群缩容
  scale-out   集群扩容
  clean       清理数据
  destroy     销毁集群
  upgrade     升级集群
  display     获取集群信息
  list        获取集群列表
  audit       查看集群操作日志
  import      导入一个由 TiDB Ansible 部署的集群
  edit-config 编辑 TiDB 集群的配置
  reload      用于必要时重载集群配置
  patch       使用临时的组件包替换集群上已部署的组件
  help        打印 Help 信息

Flags:
  -c, --concurrency int     最大并行任务数（默认值为 5）
      --format string       （实验特性）输出的格式，支持 [default, json]（默认值为 "default"）
  -h, --help                帮助信息
```

```
      --ssh string           （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --ssh-timeout uint     SSH 连接超时时间
  -v, --version              版本信息
      --wait-timeout uint    等待操作超时的时间
  -y, --yes                  跳过所有的确认步骤
```

10.4.11.2.1 部署集群

部署集群的命令为 `tiup cluster deploy`，一般用法为：

```
tiup cluster deploy <cluster-name> <version> <topology.yaml> [flags]
```

该命令需要提供集群的名字、集群使用的TiDB 版本（例如 `v6.5.0`），以及一个集群的拓扑文件。

拓扑文件的编写可参考示例。以一个最简单的拓扑为例，将下列文件保存为 `/tmp/topology.yaml`：

> **注意：**
>
> TiUP Cluster 组件的部署和扩容拓扑是使用yaml 语法编写，所以需要注意缩进。

```
---

pd_servers:
  - host: 172.16.5.134
    name: pd-134
  - host: 172.16.5.139
    name: pd-139
  - host: 172.16.5.140
    name: pd-140

tidb_servers:
  - host: 172.16.5.134
  - host: 172.16.5.139
  - host: 172.16.5.140

tikv_servers:
  - host: 172.16.5.134
  - host: 172.16.5.139
  - host: 172.16.5.140

tiflash_servers:
  - host: 172.16.5.141
  - host: 172.16.5.142
  - host: 172.16.5.143
```

```
grafana_servers:
  - host: 172.16.5.134

monitoring_servers:
  - host: 172.16.5.134
```

TiUP 默认部署在amd64 架构上运行的binary，若目标机器为arm64 架构，可以在拓扑文件中进行配置：

```
global:
  arch: "arm64"           # 让所有机器默认使用 arm64 的 binary


tidb_servers:
  - host: 172.16.5.134
    arch: "amd64"         # 这台机器会使用 amd64 的 binary
  - host: 172.16.5.139
    arch: "arm64"         # 这台机器会使用 arm64 的 binary
  - host: 172.16.5.140    # 没有配置 arch 字段的机器，会使用 global 中的默认值，这个例子中是
      ↪ arm64


...
```

假如我们想要使用TiDB 的 v7.0.0 版本，集群名字为 prod-cluster，则执行以下命令：

```
tiup cluster deploy -p prod-cluster v7.0.0 /tmp/topology.yaml
```

执行过程中会再次确认拓扑结构并提示输入目标机器上的root 密码（-p 表示使用密码）：

```
Please confirm your topology:
TiDB Cluster: prod-cluster
TiDB Version: v7.0.0
Type        Host          Ports                          OS/Arch       Directories
----        ----          -----                          -------       -----------
pd          172.16.5.134  2379/2380                      linux/x86_64  deploy/pd-2379,data/pd
    ↪ -2379
pd          172.16.5.139  2379/2380                      linux/x86_64  deploy/pd-2379,data/pd
    ↪ -2379
pd          172.16.5.140  2379/2380                      linux/x86_64  deploy/pd-2379,data/pd
    ↪ -2379
tikv        172.16.5.134  20160/20180                    linux/x86_64  deploy/tikv-20160,data/
    ↪ tikv-20160
tikv        172.16.5.139  20160/20180                    linux/x86_64  deploy/tikv-20160,data/
    ↪ tikv-20160
tikv        172.16.5.140  20160/20180                    linux/x86_64  deploy/tikv-20160,data/
    ↪ tikv-20160
tidb        172.16.5.134  4000/10080                     linux/x86_64  deploy/tidb-4000
tidb        172.16.5.139  4000/10080                     linux/x86_64  deploy/tidb-4000
```

```
tidb         172.16.5.140  4000/10080                          linux/x86_64  deploy/tidb-4000
tiflash      172.16.5.141  9000/8123/3930/20170/20292/8234  linux/x86_64  deploy/tiflash-9000,data
    ′→ /tiflash-9000
tiflash      172.16.5.142  9000/8123/3930/20170/20292/8234  linux/x86_64  deploy/tiflash-9000,data
    ′→ /tiflash-9000
tiflash      172.16.5.143  9000/8123/3930/20170/20292/8234  linux/x86_64  deploy/tiflash-9000,data
    ′→ /tiflash-9000
prometheus   172.16.5.134  9090          deploy/prometheus-9090,data/prometheus-9090
grafana      172.16.5.134  3000          deploy/grafana-3000
Attention:

    1. If the topology is not what you expected, check your yaml file.
    1. Please confirm there is no port/directory conflicts in same host.
Do you want to continue? [y/N]:
```

输入密码后tiup-cluster 便会下载需要的组件并部署到对应的机器上，当看到以下提示时说明部署成功：

```
Deployed cluster `prod-cluster` successfully
```

10.4.11.2.2 查看集群列表

集群部署成功后，可以通过 `tiup cluster list` 命令在集群列表中查看该集群：

```
tiup cluster list
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster list
Name          User  Version    Path                                              PrivateKey
----          ----  -------    ----                                              ----------
prod-cluster  tidb  v7.0.0     /root/.tiup/storage/cluster/clusters/prod-cluster  /root/.tiup/
    ′→ storage/cluster/clusters/prod-cluster/ssh/id_rsa
```

10.4.11.2.3 启动集群

集群部署成功后，可以执行以下命令启动该集群。如果忘记了部署的集群的名字，可以使用`tiup cluster list` 命令查看。

```
tiup cluster start prod-cluster
```

TiUP 使用 Systemd 启动守护进程。如果进程意外退出，会在 15s 间隔后被重新拉起。

10.4.11.2.4 检查集群状态

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，TiUP 提供了 `tiup cluster display` 命令，用法如下：

```
tiup cluster display prod-cluster
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v7.0.0
ID                  Role        Host          Ports                                OS/Arch
    ′→ Status  Data Dir              Deploy Dir
--                  ----        ----          -----                                --------
    ′→
       -------  ---------             -----------
172.16.5.134:3000   grafana     172.16.5.134  3000                                 linux/x86_64  Up
    ′→      -                      deploy/grafana-3000
172.16.5.134:2379   pd          172.16.5.134  2379/2380                            linux/x86_64  Up|L
    ′→     data/pd-2379           deploy/pd-2379
172.16.5.139:2379   pd          172.16.5.139  2379/2380                            linux/x86_64  Up|
    ′→ UI   data/pd-2379           deploy/pd-2379
172.16.5.140:2379   pd          172.16.5.140  2379/2380                            linux/x86_64  Up
    ′→      data/pd-2379           deploy/pd-2379
172.16.5.134:9090   prometheus  172.16.5.134  9090                                 linux/x86_64  Up
    ′→      data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000   tidb        172.16.5.134  4000/10080                           linux/x86_64  Up
    ′→      -                      deploy/tidb-4000
172.16.5.139:4000   tidb        172.16.5.139  4000/10080                           linux/x86_64  Up
    ′→      -                      deploy/tidb-4000
172.16.5.140:4000   tidb        172.16.5.140  4000/10080                           linux/x86_64  Up
    ′→      -                      deploy/tidb-4000
172.16.5.141:9000   tiflash     172.16.5.141  9000/8123/3930/20170/20292/8234      linux/x86_64  Up
    ′→      data/tiflash-9000      deploy/tiflash-9000
172.16.5.142:9000   tiflash     172.16.5.142  9000/8123/3930/20170/20292/8234      linux/x86_64  Up
    ′→      data/tiflash-9000      deploy/tiflash-9000
172.16.5.143:9000   tiflash     172.16.5.143  9000/8123/3930/20170/20292/8234      linux/x86_64  Up
    ′→      data/tiflash-9000      deploy/tiflash-9000
172.16.5.134:20160  tikv        172.16.5.134  20160/20180                          linux/x86_64  Up
    ′→      data/tikv-20160        deploy/tikv-20160
172.16.5.139:20160  tikv        172.16.5.139  20160/20180                          linux/x86_64  Up
    ′→      data/tikv-20160        deploy/tikv-20160
172.16.5.140:20160  tikv        172.16.5.140  20160/20180                          linux/x86_64  Up
    ′→      data/tikv-20160        deploy/tikv-20160
```

Status 列用 Up 或者 Down 表示该服务是否正常。对于 PD 组件，同时可能会带有 |L 表示该 PD 是 Leader，|UI 表示该 PD 运行着TiDB Dashboard。

10.4.11.2.5 缩容节点

> **注意：**

本节只展示缩容命令的语法示例，线上扩缩容具体步骤可参考使用TiUP 扩容缩容TiDB 集群。

缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

由于TiKV、 TiFlash 和 TiDB Binlog 组件的下线是异步的（需要先通过API 执行移除操作）并且下线过程耗时较长（需要持续观察节点是否已经下线成功），所以对 TiKV、 TiFlash 和TiDB Binlog 组件做了特殊处理：

- 对TiKV、 TiFlash 及 Binlog 组件的操作
  - TiUP cluster 通过API 将其下线后直接退出而不等待下线完成
  - 等之后再执行集群操作相关的命令时，会检查是否存在已经下线完成的TiKV、 TiFlash 或者 Binlog 节点。如果不存在，则继续执行指定的操作；如果存在，则执行如下操作：
    1. 停止已经下线掉的节点的服务
    2. 清理已经下线掉的节点的相关数据文件
    3. 更新集群的拓扑，移除已经下线掉的节点
- 对其他组件的操作
  - 下线 PD 组件时，会通过API 将指定节点从集群中删除掉（这个过程很快），然后停掉指定 PD 的服务并且清除该节点的相关数据文件
  - 下线其他组件时，直接停止并且清除节点的相关数据文件

缩容命令的基本用法：

```
tiup cluster scale-in <cluster-name> -N <node-id>
```

它需要指定至少两个参数，一个是集群名字，另一个是节点ID。 节点ID 可以参考上一节使用 `tiup cluster display` 命令获取。

比如想缩容 172.16.5.140 上的TiKV 节点，可以执行：

```
tiup cluster scale-in prod-cluster -N 172.16.5.140:20160
```

通过 `tiup cluster display` 可以看到该TiKV 已经被标记为 `Offline`：

```
tiup cluster display prod-cluster
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v7.0.0
ID                    Role        Host          Ports                            OS/Arch
    ↪ Status    Data Dir              Deploy Dir
--                    ----        ----          -----                            --------
    ↪ -------   ---------             -----------
172.16.5.134:3000     grafana     172.16.5.134  3000                             linux/x86_64  Up
    ↪       -                         deploy/grafana-3000
172.16.5.134:2379     pd          172.16.5.134  2379/2380                        linux/x86_64  Up|L
    ↪       data/pd-2379              deploy/pd-2379
```

```
172.16.5.139:2379    pd           172.16.5.139  2379/2380                          linux/x86_64  Up|
    ′→ UI     data/pd-2379           deploy/pd-2379
172.16.5.140:2379    pd           172.16.5.140  2379/2380                          linux/x86_64  Up
    ′→        data/pd-2379           deploy/pd-2379
172.16.5.134:9090    prometheus   172.16.5.134  9090                               linux/x86_64  Up
    ′→        data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000    tidb         172.16.5.134  4000/10080                         linux/x86_64  Up
    ′→        -                      deploy/tidb-4000
172.16.5.139:4000    tidb         172.16.5.139  4000/10080                         linux/x86_64  Up
    ′→        -                      deploy/tidb-4000
172.16.5.140:4000    tidb         172.16.5.140  4000/10080                         linux/x86_64  Up
    ′→        -                      deploy/tidb-4000
172.16.5.141:9000    tiflash      172.16.5.141  9000/8123/3930/20170/20292/8234    linux/x86_64  Up
    ′→        data/tiflash-9000      deploy/tiflash-9000
172.16.5.142:9000    tiflash      172.16.5.142  9000/8123/3930/20170/20292/8234    linux/x86_64  Up
    ′→        data/tiflash-9000      deploy/tiflash-9000
172.16.5.143:9000    tiflash      172.16.5.143  9000/8123/3930/20170/20292/8234    linux/x86_64  Up
    ′→        data/tiflash-9000      deploy/tiflash-9000
172.16.5.134:20160   tikv         172.16.5.134  20160/20180                        linux/x86_64  Up
    ′→        data/tikv-20160        deploy/tikv-20160
172.16.5.139:20160   tikv         172.16.5.139  20160/20180                        linux/x86_64  Up
    ′→        data/tikv-20160        deploy/tikv-20160
172.16.5.140:20160   tikv         172.16.5.140  20160/20180                        linux/x86_64
    ′→ Offline  data/tikv-20160        deploy/tikv-20160
```

待 PD 将其数据调度到其他TiKV 后，该节点会被自动删除。

10.4.11.2.6 扩容节点

> **注意：**
>
> 本节只用于展示扩容命令的语法示例，线上扩缩容可参考使用TiUP 扩容缩容TiDB 集群。

扩容的内部逻辑与部署类似，TiUP cluster 组件会先保证节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。其中 PD 节点的扩容会通过join 方式加入到集群中，并且会更新与 PD 有关联的服务的配置；其他服务直接启动加入到集群中。所有服务在扩容时都会做正确性验证，最终返回是否扩容成功。

例如，在集群 `tidb-test` 中扩容一个TiKV 节点和一个 PD 节点：

1. 新建 scale.yaml 文件，添加新增的TiKV 和 PD 节点IP：

> **注意：**
>
> 需要新建一个拓扑文件，文件中只写入扩容节点的描述信息，不要包含已存在的节点。

```
---

pd_servers:
  - host: 172.16.5.140

tikv_servers:
  - host: 172.16.5.140
```

2. 执行扩容操作。TiUP cluster 根据 scale.yaml 文件中声明的端口、目录等信息在集群中添加相应的节点：

```
tiup cluster scale-out tidb-test scale.yaml
```

执行完成之后可以通过 `tiup cluster display tidb-test` 命令检查扩容后的集群状态。

10.4.11.2.7 滚动升级

> **注意：**
> 本节只用于展示命令的语法示例，线上升级请参考使用TiUP 升级TiDB。

滚动升级功能借助TiDB 的分布式能力，升级过程中尽量保证对前端业务透明、无感知。升级时会先检查各个组件的配置文件是否合理，如果配置有问题，则报错退出；如果配置没有问题，则工具会逐个节点进行升级。其中对不同节点有不同的操作。

不同节点的操作

- 升级 PD 节点
  - 优先升级非 Leader 节点
  - 所有非 Leader 节点升级完成后再升级 Leader 节点
    * 工具会向 PD 发送一条命令将 Leader 迁移到升级完成的节点上
    * 当 Leader 已经切换到其他节点之后，再对旧的 Leader 节点做升级操作
  - 同时升级过程中，若发现有不健康的节点，工具会中止本次升级并退出，此时需要由人工判断、修复后再执行升级。
- 升级TiKV 节点
  - 先在 PD 中添加一个迁移对应TiKV 上 Region leader 的调度，通过迁移 Leader 确保升级过程中不影响前端业务
  - 等待迁移 Leader 完成之后，再对该TiKV 节点进行升级更新
  - 等更新后的TiKV 正常启动之后再移除迁移 Leader 的调度
- 升级其他服务
  - 正常停止服务再更新

升级操作

升级命令参数如下：

```
Usage:
  tiup cluster upgrade <cluster-name> <version> [flags]

Flags:
      --force                在不 transfer leader 的情况下强制升级（危险操作）
  -h, --help                 帮助手册
      --transfer-timeout int transfer leader 的超时时间

Global Flags:
      --ssh string         （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --wait-timeout int 等待操作超时的时间
      --ssh-timeout int    SSH 连接的超时时间
  -y, --yes                跳过所有的确认步骤
```

例如，把集群升级到 v7.0.0 的命令为：

```
tiup cluster upgrade tidb-test v7.0.0
```

10.4.11.2.8 更新配置

如果想要动态更新组件的配置，TiUP cluster 组件为每个集群保存了一份当前的配置，如果想要编辑这份配置，则执行 `tiup cluster edit-config <cluster-name>` 命令。例如：

```
tiup cluster edit-config prod-cluster
```

然后TiUP cluster 组件会使用 vi 打开配置文件供编辑（如果你想要使用其他编辑器，请使用 `EDITOR` 环境变量自定义编辑器，例如 `export EDITOR=nano`），编辑完之后保存即可。此时的配置并没有应用到集群，如果想要让它生效，还需要执行：

```
tiup cluster reload prod-cluster
```

该操作会将配置发送到目标机器，重启集群，使配置生效。

> 注意：
>
> 对于监控组件，可以通过执行 `tiup cluster edit-config` 命令在对应实例上添加自定义配置路径来进行配置自定义，例如：

```
---

grafana_servers:
```

```
  - host: 172.16.5.134
    dashboard_dir: /path/to/local/dashboards/dir

monitoring_servers:
  - host: 172.16.5.134
    rule_dir: /path/to/local/rules/dir

alertmanager_servers:
  - host: 172.16.5.134
    config_file: /path/to/local/alertmanager.yml
```

路径内容格式如下：

- grafana_servers 的 dashboard_dir 字段指定的文件夹中应当含有完整的 *.json 文件。
- monitoring_servers 的 rule_dir 字段定义的文件夹中应当含有完整的 *.rules.yml 文件。
- alertmanager_servers 的 config_file 格式请参考Alertmanager 配置模板。

在执行 tiup reload 时，TiUP 会将中控机上对应的配置上传到目标机器对应的配置目录中，上传之前会删除目标机器中已有的旧配置文件。如果想要修改某一个配置文件，请确保将所有的（包含未修改的）配置文件都放在同一个目录中。例如，要修改Grafana 的 tidb.json 文件，可以先将Grafana 的 dashboards 目录中所有的 *.json 文件拷贝到本地目录中，再修改 tidb.json 文件。否则最终的目标机器上将缺失其他的JSON 文件。

> 注意：
>
> 如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：
>
> 1. 在本地的 dashboards 目录中，将集群名修改为新的集群名。
> 2. 在本地的 dashboards 目录中，将 datasource 更新为新的集群名（datasource 是以集群名命名的）。
> 3. 执行 tiup cluster reload -R grafana 命令。

10.4.11.2.9 更新组件

常规的升级集群可以使用upgrade 命令，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用patch 命令：

```
tiup cluster patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup cluster patch <cluster-name> <package-path> [flags]
```

```
Flags:
  -h, --help                    帮助信息
  -N, --node strings            指定被替换的节点
      --overwrite               在未来的 scale-out 操作中使用当前指定的临时包
  -R, --role strings            指定被替换的服务类型
      --transfer-timeout int    transfer leader 的超时时间

Global Flags:
      --ssh string          （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --wait-timeout int    等待操作超时的时间
      --ssh-timeout int     SSH 连接的超时时间
  -y, --yes                 跳过所有的确认步骤
```

例如，有一个TiDB 的 hotfix 包放在 /tmp/tidb-hotfix.tar.gz，如果此时想要替换集群上的所有 TiDB，则可以执行：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -R tidb
```

或者只替换其中一个TiDB：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -N 172.16.4.5:4000
```

10.4.11.2.10 导入 TiDB Ansible 集群

> **注意：**
>
> TiUP cluster 组件对TiSpark 的支持目前为实验性特性，暂不支持导入启用了TiSpark 组件的集群。

在 TiUP 之前，一般使用TiDB Ansible 部署TiDB 集群，import 命令用于将这部分集群过渡给TiUP 接管。import 命令用法如下：

```
tiup cluster import --help
```

```
Import an exist TiDB cluster from TiDB-Ansible

Usage:
  tiup cluster import [flags]

Flags:
  -d, --dir string          TiDB-Ansible 的目录，默认为当前目录
  -h, --help                import 的帮助信息
      --inventory string    inventory 文件的名字（默认为 "inventory.ini"）
      --no-backup           不备份 Ansible 目录，用于存在多个 inventory 文件的 Ansible 目录
  -r, --rename NAME         重命名被导入的集群
```

```
Global Flags:
      --ssh string         （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --wait-timeout int 等待操作超时的时间
      --ssh-timeout int    SSH 连接的超时时间
  -y, --yes                跳过所有的确认步骤
```

例如，导入一个TiDB Ansible 集群：

```
cd tidb-ansible
tiup cluster import
```

或者

```
tiup cluster import --dir=/path/to/tidb-ansible
```

10.4.11.2.11 查看操作日志

操作日志的查看可以借助audit 命令，其用法如下：

```
Usage:
  tiup cluster audit [audit-id] [flags]

Flags:
  -h, --help   help for audit
```

在不使用 [audit-id] 参数时，该命令会显示执行的命令列表，如下：

```
tiup cluster audit
```

```
Starting component `cluster`: /home/tidb/.tiup/components/cluster/v1.11.3/cluster audit
ID      Time                          Command
--      ----                          -------
4BLhr0  2022-03-29T23:55:09+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ′→ test v7.0.0 /tmp/topology.yaml
4BKWjF  2022-03-029T23:36:57+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ′→ test v7.0.0 /tmp/topology.yaml
4BKVwH  2022-03-29T23:02:08+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ′→ test v7.0.0 /tmp/topology.yaml
4BKKH1  2022-03-29T16:39:04+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster destroy
    ′→ test
4BKKDx  2022-03-29T16:36:57+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ′→ test v7.0.0 /tmp/topology.yaml
```

第一列为audit-id，如果想看某个命令的执行日志，则传入这个audit-id：

```
tiup cluster audit 4BLhr0
```

10.4.11.2.12 在集群节点机器上执行命令

exec 命令可以很方便地到集群的机器上执行命令，其使用方式如下：

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

Flags:
      --command string    需要执行的命令 (默认是 "ls")
  -h, --help              帮助信息
  -N, --node strings      指定要执行的节点 ID （节点 id 通过 display 命令获取）
  -R, --role strings      指定要执行的 role
      --sudo              是否使用 root (默认为 false)
```

例如，如果要到所有的TiDB 节点上执行 ls /tmp：

```
tiup cluster exec test-cluster --command='ls /tmp'
```

10.4.11.2.13 集群控制工具(controllers)

在 TiUP 之前，我们用 tidb-ctl、tikv-ctl、pd-ctl 等工具操控集群，为了方便下载和使用，TiUP 将它们集成到了统一的组件 ctl 中：

```
Usage:
  tiup ctl:v<CLUSTER_VERSION> {tidb/pd/tikv/binlog/etcd} [flags]

Flags:
  -h, --help   help for tiup
```

这个命令和之前的命令对应关系为：

```
tidb-ctl [args] = tiup ctl tidb [args]
pd-ctl [args] = tiup ctl pd [args]
tikv-ctl [args] = tiup ctl tikv [args]
binlogctl [args] = tiup ctl bindlog [args]
etcdctl [args] = tiup ctl etcd [args]
```

例如，以前查看 store 的命令为 pd-ctl -u http://127.0.0.1:2379 store，集成到 TiUP 中的命令为：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 store
```

10.4.11.2.14 部署机环境检查

使用 check 子命令可以对部署机的环境进行一系列检查，并输出检查结果。通过执行 check 子命令，可以发现常见的不合理配置或不支持情况。命令参数列表如下：

```
Usage:
  tiup cluster check <topology.yml | cluster-name> [flags]
```

```
Flags:
      --apply                  Try to fix failed checks
      --cluster                Check existing cluster, the input is a cluster name.
      --enable-cpu             Enable CPU thread count check
      --enable-disk            Enable disk IO (fio) check
      --enable-mem             Enable memory size check
  -h, --help                   help for check
  -i, --identity_file string   The path of the SSH identity file. If specified, public key
      ↪ authentication will be used.
  -p, --password               Use password of target hosts. If specified, password
      ↪ authentication will be used.
      --user string            The user name to login via SSH. The user must has root (or sudo)
          ↪ privilege.
```

默认情况下，此功能用于在部署前进行环境检查，通过指定 `--cluster` 参数切换模式，也可以用于对已部署集群的部署机进行检查，例如：

```
#### check deploy servers before deploy
tiup cluster check topology.yml --user tidb -p

#### check deploy servers of an existing cluster
tiup cluster check <cluster-name> --cluster
```

其中，CPU 线程数检查、内存大小检查和磁盘性能检查三项默认关闭，对于生产环境，建议将此三项检测开启并确保通过，以获得最佳性能。

- CPU：线程数大于等于 16 为通过检查
- 内存：物理内存总大小大于等于 32 GB 为通过检查
- 磁盘：对 `data_dir` 所在分区执行 `fio` 测试并记录结果

在运行检测时，若指定了 `--apply` 参数，程序将尝试对其中未通过的项目自动修复。自动修复仅限于部分可通过修改配置或系统参数调整的项目，其它未修复的项目需要根据实际情况手工处理。

环境检查不是部署集群的必需流程。对于生产环境建议在部署前执行环境检查并通过所有检测项。如果未通过全部检查项，也可能正常部署和运行集群，但可能无法获得最佳性能表现。

10.4.11.2.15 使用中控机系统自带的 SSH 客户端连接集群

在以上所有操作中，涉及到对集群机器的操作都是通过TiUP 内置的 SSH 客户端连接集群执行命令，但是在某些场景下，需要使用系统自带的 SSH 客户端来对集群执行操作，比如：

- 使用 SSH 插件来做认证
- 使用定制的 SSH 客户端

此时可以通过命令行参数 `--ssh=system` 启用系统自带命令行：

- 部署集群：`tiup cluster deploy <cluster-name> <version> <topo> --ssh=system`，其中`<cluster-name>` 为集群名称，`<version>` 为 TiDB 集群版本（例如 `v6.5.0`），`<topo>` 为拓扑文件路径
- 启动集群：`tiup cluster start <cluster-name> --ssh=system`
- 升级集群：`tiup cluster upgrade ... --ssh=system`

所有涉及集群操作的步骤都可以加上 `--ssh=system` 来使用系统自带的客户端。

也可以使用环境变量 `TIUP_NATIVE_SSH` 来指定是否使用本地 SSH 客户端，避免每个命令都需要添加 `--ssh=system` 参数：

```
export TIUP_NATIVE_SSH=true
#### 或者
export TIUP_NATIVE_SSh=1
#### 或者
export TIUP_NATIVE_SSH=enable
```

若环境变量和 `--ssh` 同时指定，则以 `--ssh` 为准。

> 注意：
>
> 在部署集群的步骤中，若需要使用密码的方式连接(-p)，或者密钥文件设置了passphrase，则需要保证中控机上安装了 sshpass，否则连接时会报错。

10.4.11.2.16 迁移中控机与备份

TiUP 相关的数据都存储在用户home 目录的 `.tiup` 目录下，若要迁移中控机只需要拷贝 `.tiup` 目录到对应目标机器即可。

1. 在原机器home 目录下执行 `tar czvf tiup.tar.gz .tiup`。
2. 把 `tiup.tar.gz` 拷贝到目标机器home 目录。
3. 在目标机器home 目录下执行 `tar xzvf tiup.tar.gz`。
4. 添加 `.tiup` 目录到 PATH 环境变量。

   如使用 bash 并且是 `tidb` 用户，在 `~/.bashrc` 中添加 `export PATH=/home/tidb/.tiup/bin:$PATH` 后执行 `source ~/.bashrc`，根据使用的 shell 与用户做相应调整。

> 注意：
>
> 为了避免中控机磁盘损坏等异常情况导致TiUP 数据丢失，建议定时备份 `.tiup` 目录。

10.4.11.2.17 备份与恢复集群部署和运维所需的meta 文件

如果运维所需的meta 文件丢失，会导致无法继续使用TiUP 管理集群，建议通过以下方式定期备份meta 文件：

```
tiup cluster meta backup ${cluster_name}
```

如果meta 文件丢失，可以使用以下方法恢复meta 文件：

```
tiup cluster meta restore ${cluster_name} ${backup_file}
```

> 注意：
> 恢复操作会覆盖当前的meta 文件，建议仅在meta 文件丢失的情况下进行恢复。

10.4.11.3 搭建私有镜像

在构建私有云时，通常会使用隔离的网络环境，此时无法访问TiUP 的官方镜像。因此，TiUP 提供了构建私有镜像的方案，它主要由mirror 指令来实现，该方案也可用于离线部署。使用私有镜像，你可以使用自己构建和打包的组件。

10.4.11.3.1 mirror 指令介绍

`mirror` 指令的帮助文档如下：

```
tiup mirror --help
```

```
The `mirror` command is used to manage a component repository for TiUP, you can use
it to create a private repository, or to add new component to an existing repository.
The repository can be used either online or offline.
It also provides some useful utilities to help manage keys, users, and versions
of components or the repository itself.

Usage:
  tiup mirror <command> [flags]

Available Commands:
  init        Initialize an empty repository
  sign        Add signatures to a manifest file
  genkey      Generate a new key pair
  clone       Clone a local mirror from remote mirror and download all selected components
  merge       Merge two or more offline mirrors
  publish     Publish a component
  show        Show the mirror address
  set         Set mirror address
  modify      Modify published component
```

```
  renew       Renew the manifest of a published component.
  grant       grant a new owner
  rotate      Rotate root.json

Flags:
  -h, --help          help for mirror
      --repo string   Path to the repository

Global Flags:
      --help Help for this command

Use "tiup mirror [command] --help" for more information about a command.
```

10.4.11.3.2 克隆镜像

执行 `tiup mirror clone` 命令，可构建本地地镜像：

```
tiup mirror clone <target-dir> [global-version] [flags]
```

- `target-dir`：指需要把克隆下来的数据放到哪个目录里。
- `global-version`：用于为所有组件快速设置一个共同的版本。

`tiup mirror clone` 命令提供了很多可选参数，日后可能会提供更多。但这些参数其实可以分为四类：

1. 克隆时是否使用前缀匹配方式匹配版本

   如果指定了 `--prefix` 参数，则会才用前缀匹配方式匹配克隆的版本号。例：指定 `--prefix` 时，填写版本“v5.0.0”将会匹配“v5.0.0-rc”，“v5.0.0”

2. 是否全量克隆

   如果指定了 `--full` 参数，则会完整地克隆官方镜像。

   > 注意：
   > 如果既不指定 `--full` 参数，又不指定 `global-version` 或克隆的component 版本，那么 TiUP 就只会克隆一些元信息。

3. 限定只克隆特定平台的包

   如果只想克隆某个平台的包，那么可以使用 `--os` 和 `--arch` 来限定：

   - 只想克隆linux 平台的，则执行 `tiup mirror clone <target-dir> [global-version] --os=linux`
   - 只想克隆 amd64 架构的， 则执行 `tiup mirror clone <target-dir> [global-version] --arch=amd64`
   - 只想克隆linux/amd64 的，则执行 `tiup mirror clone <target-dir> [global-version] --os=linux --arch=amd64`

4. 限定只克隆组件的特定版本

   如果只想克隆某个组件的某一个版本而不是所有版本，则使用 `--<component>=<version>` 来限定，例如：

   - 只想克隆TiDB 的 v7.0.0 版本，则执行 `tiup mirror clone <target-dir> --tidb v7.0.0`
   - 只想克隆TiDB 的 v7.0.0 版本，以及TiKV 的所有版本，则执行 `tiup mirror clone <target-dir> --tidb v7.0.0 --tikv all`
   - 克隆一个集群的所有组件的 v7.0.0 版本，则执行 `tiup mirror clone <target-dir> v7.0.0`

克隆完成后，签名密钥会自动设置。

管理私有仓库

你可以通过 SCP 和 NFS 文件共享方式，将 `tiup mirror clone` 克隆下来的仓库共享给其他主机，也可以通过 HTTP 或 HTTPS 协议来共享。可以使用 `tiup mirror set <location>` 指定仓库的位置。

```
tiup mirror set /shared_data/tiup
```

```
tiup mirror set https://tiup-mirror.example.com/
```

> 注意：
>
> 如果在执行了 `tiup mirror clone` 的机器上执行 `tiup mirror set`，下次执行 `tiup mirror clone` 时，机器会从本地镜像而非远程镜像进行克隆。因此，更新私有镜像前，需要执行 `tiup mirror set --reset` 来重置镜像。

还可以通过 `TIUP_MIRRORS` 环境变量来使用镜像。下面是一个使用私有仓库运行 `tiup list` 的例子。

```
export TIUP_MIRRORS=/shared_data/tiup
tiup list
```

设置 `TIUP_MIRRORS` 会永久改变镜像配置，例如 `tiup mirror set`。详情请参考 tiup issue #651。

更新私有仓库

如果使用同样的 `target-dir` 目录再次运行 `tiup mirror clone` 命令，机器会创建新的manifest，并下载可用的最新版本的组件。

> 注意：
>
> 重新创建manifest 之前，请确保所有组件和版本（包括之前下载的早期版本）都包含在内。

10.4.11.3.3 自定义仓库

你可以创建一个自定义仓库，以使用自己构建的TiDB 组件，例如TiDB、 TiKV 或 PD。你也可以创建自己的 TiUP 组件。

要创建自己的组件，请执行 `tiup package` 命令，并按照组件打包的说明进行操作。

创建自定义仓库

以下命令在 `/data/mirror` 目录下创建一个空仓库：

```
tiup mirror init /data/mirror
```

创建仓库时，密钥会被写入 `/data/mirror/keys`。

以下命令在 `~/.tiup/keys/private.json` 中创建一个私钥：

```
tiup mirror genkey
```

以下命令为 `jdoe` 授予 `/data/mirror` 路径下私钥 `~/.tiup/keys/private.json` 的所有权：

```
tiup mirror set /data/mirror
tiup mirror grant jdoe
```

使用自定义组件

1. 创建一个名为 `hello` 的自定义组件：

   ```
   $ cat > hello.c << END
   > #include <stdio.h>
   int main() {
     printf("hello\n");
     return (0);
   }
   END
   $ gcc hello.c -o hello
   $ tiup package hello --entry hello --name hello --release v0.0.1
   ```

   `package/hello-v0.0.1-linux-amd64.tar.gz` 创建成功。

2. 创建一个仓库和一个私钥，并为仓库授予所有权：

   ```
   $ tiup mirror init /tmp/m
   $ tiup mirror genkey
   $ tiup mirror set /tmp/m
   $ tiup mirror grant $USER
   ```

   ```
   tiup mirror publish hello v0.0.1 package/hello-v0.0.1-linux-amd64.tar.gz hello
   ```

3. 运行组件。如果组件还没有安装，会先下载安装：

```
$ tiup hello
```

```
The component `hello` version  is not installed; downloading from repository.
Starting component `hello`: /home/dvaneeden/.tiup/components/hello/v0.0.1/hello
hello
```

执行 `tiup mirror merge` 命令，可以将自定义组件的仓库合并到另一个仓库中。这一操作假设 `/data/my_custom_components` 中的所有组件都使用 `$USER` 签名：

```
$ tiup mirror set /data/my_mirror
$ tiup mirror grant $USER
$ tiup mirror merge /data/my_custom_components
```

10.4.11.4 使用TiUP bench 组件压测TiDB

在测试数据库性能时，经常需要对数据库进行压测，为了满足这一需求，TiUP 集成了bench 组件。TiUP bench 组件提供多种压测的 workloads，命令分别如下：

```
tiup bench tpcc    # 以 TPC-C 作为 workload 压测
tiup bench tpch    # 以 TPC-H 作为 workload 压测
tiup bench ch      # 以 CH-benCHmark 作为 workload 压测
tiup bench ycsb    # 以 YCSB 作为 workload 压测
tiup bench rawsql # 以自定义 SQL 文件作为 workload 压测
```

其中 `tpcc`, `tpch`, `ch`, `rawsql` 支持如下命令行参数。`ycsb` 使用方法较为不同，它主要通过properties 文件进行配置，详见 go-ycsb 使用说明。

```
  -t, --acThreads int          OLAP 并发线程数，仅适用于 CH-benCHmark (默认 1)
      --conn-params string     数据库连接参数，例如:
                               `--conn-params tidb_isolation_read_engines='tiflash'` 设置 TiDB
                                   ↪ 通过 TiFlash 进行查询
                               `--conn-params sslmode=disable` 设置连接 PostgreSQL 不启用加密
      --count int              总执行次数，0 表示无限次
  -D, --db string              被压测的数据库名 (默认为 "test")
  -d, --driver string          数据库驱动: mysql, postgres (默认 "mysql")
      --dropdata               在 prepare 数据之前清除历史数据
  -h, --help                   输出 bench 命令的帮助信息
  -H, --host strings           数据库的主机地址 (默认 ["127.0.0.1"])
      --ignore-error           忽略压测时数据库报出的错误
      --interval duration      两次报告输出时间的间隔 (默认 10s)
      --isolation int          隔离级别 0: Default, 1: ReadUncommitted,
                               2: ReadCommitted, 3: WriteCommitted, 4: RepeatableRead, 5:
                               Snapshot, 6: Serializable, 7: Linerizable
      --max-procs int          Go Runtime 能够使用的最大系统线程数
      --output string          输出格式 plain, table, json (默认为 "plain")
  -p, --password string        数据库密码
```

```
  -P, --port ints              数据库端口（默认 [4000]）
      --pprof string           pprof 地址
      --silence                压测过程中不打印错误信息
  -S, --statusPort int         TiDB 状态端口（默认 10080）
  -T, --threads int            压测并发线程数（默认 16）
      --time duration          总执行时长（默认 2562047h47m16.854775807s）
  -U, --user string            压测时使用的数据库用户（默认 "root"）
```

- `--host` 和 `--port` 支持以逗号分隔传入多个值，以启用客户端负载均衡。例如，当指定 `--host 172.16.4.1,172.16.4.2 --port 4000,4001` 时，负载程序将以轮询调度的方式连接到 172.16.4.1:4000, 172.16.4.1:4001, 172.16.4.2:4000, 172.16.4.2:4001 这 4 个实例上。
- `--conn-params` 需要符合query string 格式，不同数据库支持不同参数，如：
  - `--conn-params tidb_isolation_read_engines='tiflash'` 设置TiDB 通过TiFlash 进行查询。
  - `--conn-params sslmode=disable` 设置连接 PostgreSQL 不启用加密。
- 当运行 CH-benCHmark 时，可以通过 `--ap-host`, `--ap-port`, `--ap-conn-params` 来指定独立的TiDB 实例用于 OLAP 查询。

下文分别介绍如何使用TiUP 运行 TPC-C, TPC-H 以及YCSB 测试。

10.4.11.4.1 使用TiUP 运行 TPC-C 测试

TiUP bench 组件支持如下运行 TPC-C 测试的命令和参数：

```
Available Commands:
  check       检查数据一致性
  cleanup     清除数据
  prepare     准备数据
  run         开始压测

Flags:
      --check-all              运行所有的一致性检测
  -h, --help                   输出 TPC-C 的帮助信息
      --partition-type int     分区类型（默认为 1）
                               1 代表 HASH 分区类型
                               2 代表 RANGE 分区类型
                               3 代表 LIST 分区类型并按 HASH 方式划分
                               4 代表 LIST 分区类型并按 RANGE 方式划分
      --parts int              分区仓库的数量（默认为 1）
      --warehouses int         仓库的数量（默认为 10）
```

TPC-C 测试步骤

以下为简化后的关键步骤。完整的测试流程可以参考如何对TiDB 进行 TPC-C 测试

1. 通过 HASH 使用 4 个分区创建 4 个仓库：

```
tiup bench tpcc --warehouses 4 --parts 4 prepare
```

2. 运行 TPC-C 测试：

```
tiup bench tpcc --warehouses 4 --time 10m run
```

3. 检查一致性：

```
tiup bench tpcc --warehouses 4 check
```

4. 清理数据：

```
tiup bench tpcc --warehouses 4 cleanup
```

当需要测试大数据集时，直接写入数据通常较慢，此时可以使用如下命令生成CSV 数据集，然后通过TiDB Lightning 导入数据。

- 生成CSV 文件：

```
shell tiup bench tpcc --warehouses 4 prepare --output-dir data --output-type=csv
```

- 为指定的表生成CSV 文件：

```
shell tiup bench tpcc --warehouses 4 prepare --output-dir data --output-type=csv --tables history
↪ ,orders
```

10.4.11.4.2 使用TiUP 运行 TPC-H 测试

TiUP bench 组件支持如下运行 TPC-H 测试的命令和参数：

```
Available Commands:
  cleanup     清除数据
  prepare     准备数据
  run         开始压测

Flags:
      --check             检查输出数据，只有 scale 因子为 1 时有效
  -h, --help              tpch 的帮助信息
      --queries string    所有的查询语句 (默认 "q1,q2,q3,q4,q5,q6,q7,q8,q9,q10,q11,q12,q13,q14,q15
          ↪ ,q16,q17,q18,q19,q20,q21,q22")
      --sf int            scale 因子
```

10.4.11.4.3 TPC-H 测试步骤

1. 准备数据：

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

2. 运行 TPC-H 测试，根据是否检查结果执行相应命令：

   - 检查结果：

   ```
   tiup bench tpch --count=22 --sf=1 --check=true run
   ```

   - 不检查结果：

   ```
   tiup bench tpch --count=22 --sf=1 run
   ```

3. 清理数据：

```
tiup bench tpch cleanup
```

10.4.11.4.4 使用TiUP运行YCSB测试

你可以使用TiUP 对 TiDB 和 TiKV 节点分别进行YCSB 测试。

测试TiDB

1. 准备数据：

```
tiup bench ycsb load tidb -p tidb.instances="127.0.0.1:4000" -p recordcount=10000
```

2. 运行YCSB 测试：

```
# 默认读写比例为 95:5
tiup bench ycsb run tidb -p tidb.instances="127.0.0.1:4000" -p operationcount=10000
```

测试TiKV

1. 准备数据：

```
tiup bench ycsb load tikv -p tikv.pd="127.0.0.1:2379" -p recordcount=10000
```

2. 运行YCSB 测试：

```
# 默认读写比例为 95:5
tiup bench ycsb run tikv -p tikv.pd="127.0.0.1:2379" -p operationcount=10000
```

10.4.11.4.5 使用TiUP运行RawSQL测试

你可以将 OLAP 查询写到 SQL 文件中，通过 `tiup bench rawsql` 执行测试，步骤如下：

1. 准备数据和需要执行的查询：

```
-- 准备数据
CREATE TABLE t (a int);
INSERT INTO t VALUES (1), (2), (3);

-- 构造查询，保存为 demo.sql
SELECT a, sleep(rand()) FROM t WHERE a < 4*rand();
```

2. 运行RawSQL测试：

```
shell  tiup bench rawsql run --count 60 --query-files demo.sql
```

10.5 TiDB Operator

TiDB Operator 是 Kubernetes 上的TiDB集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助TiDB Operator，TiDB 可以无缝运行在公有云或私有部署的 Kubernetes 集群上。

TiDB Operator 的文档目前独立于TiDB 文档，文档名称为TiDB on Kubernetes 用户文档。要访问TiDB Operator 的文档，请点击以下链接：

- TiDB on Kubernetes 用户文档

10.6 TiDB Data Migration

10.6.1 TiDB Data Migration 简介

TiDB Data Migration (DM) 是一款便捷的数据迁移工具，支持从与 MySQL 协议兼容的数据库（MySQL、MariaDB、Aurora MySQL）到 TiDB 的全量数据迁移和增量数据同步。使用 DM 工具有利于简化数据迁移过程，降低数据迁移运维成本。

10.6.1.1 产品特性

- 与 TiDB 同样保持 MySQL 兼容性。高度兼容 MySQL 5.7 协议、MySQL 5.7 常用的功能及语法。
- 支持 DML & DDL 事件同步。支持解析和同步binlog 中的 DML 和 DDL 事件。
- 支持合库合表同步模式。可以方便的将上游各个分片 MySQL 实例的各个分表数据，合并同步到下游 TiDB 的一张表。支持自定义编写同步规则以方便各种可能的同步需求，且具备自动识别和处理上游分片 MySQL 的 DDL 变更，大幅简化运维成本。
- 内置多种过滤器以灵活适应不同场景。支持以预定义事件类型、正则表达式、SQL 表达式等多种方式在数据同步过程中对 MySQL binlog 事件进行过滤。
- 集中管理。DM 支持上千个节点的集群规模，可同时运行并集中管理大量数据迁移同步任务。

- 对第三方 Online Schema Change 工具变更过程的同步优化。在 MySQL 生态中，gh-ost 与 pt-osc 等工具被广泛使用，DM 对其变更过程进行了特殊的优化，以避免对不必要的中间数据进行迁移。详细信息可参考online-ddl。
- 高可用。支持迁移任务在不同节点自由调度，少量工作节点宕机并不会影响进行中的任务。

10.6.1.2 快速安装

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

10.6.1.3 使用限制

在使用 DM 工具之前，需了解以下限制：

- 数据库版本要求
  - MySQL 版本 5.5 ~ 5.7
  - MySQL 版本 = 8.0 （实验特性）
  - MariaDB 版本 >= 10.1.2 （实验特性）

> 注意：
> 如果上游 MySQL/MariaDB servers 间构成主从复制结构，则需要 MySQL 版本高于 5.7.1 或者 MariaDB 版本等于或高于 10.1.3。

- DDL 语法兼容性限制
  - 目前，TiDB 部分兼容 MySQL 支持的 DDL 语句。因为 DM 使用 TiDB parser 来解析处理 DDL 语句，所以目前仅支持TiDB parser 支持的 DDL 语法。详见TiDB DDL 语法支持。
  - DM 遇到不兼容的 DDL 语句时会报错。要解决此报错，需要使用 dmctl 手动处理，要么跳过该 DDL 语句，要么用指定的 DDL 语句来替换它。详见如何处理不兼容的 DDL 语句。
  - DM 不会将视图的 DDL 语句同步到下游的TiDB 集群，也不会将针对视图的 DML 语句同步到下游。在该场景下，建议用户在下游TiDB 集群中自行创建视图。
- GBK 字符集兼容性限制
  - DM 在 v5.4.0 之前不支持将 `charset=GBK` 的表迁移到TiDB。

10.6.1.4 Contributing

欢迎参与 DM 开源项目并万分感谢您的贡献，可以查看 CONTRIBUTING.md 了解更多信息。

10.6.1.5 社区技术支持

您可以通过在线文档了解和使用 DM，如果您遇到无法解决的问题，可以选择以下途径之一联系我们。

- GitHub
- AskTUG

10.6.1.6 License

DM 遵循 Apache 2.0 协议，在 LICENSE 了解更多信息。

10.6.1.7 版本变更说明

在 v5.4 之前，DM 工具的文档独立于 TiDB 文档。要访问这些早期版本的 DM 文档，请点击以下链接：

- DM v5.3 文档
- DM v2.0 文档
- DM v1.0 文档

> 注意：
>
> - DM 的 GitHub 代码仓库已于 2021 年 10 月迁移至 pingcap/tiflow。如有任何关于 DM 的问题，请在 `pingcap/tiflow` 仓库提交，以获得后续支持。
> - 在较早版本中（v1.0 和 v2.0），DM 采用独立于 TiDB 的版本号。从 DM v5.3 起，DM 采用与 TiDB 相同的版本号。DM v2.0 的下一个版本为 DM v5.3。DM v2.0 到 v5.3 无兼容性变更，升级过程与正常升级无差异。

要快速了解 DM 的原理架构、适用场景，建议先观看下面的培训视频。注意本视频只作为学习参考，具体操作步骤和最新功能，请以文档内容为准。

10.6.2 Data Migration 架构

DM 主要包括三个组件：DM-master，DM-worker 和 dmctl。

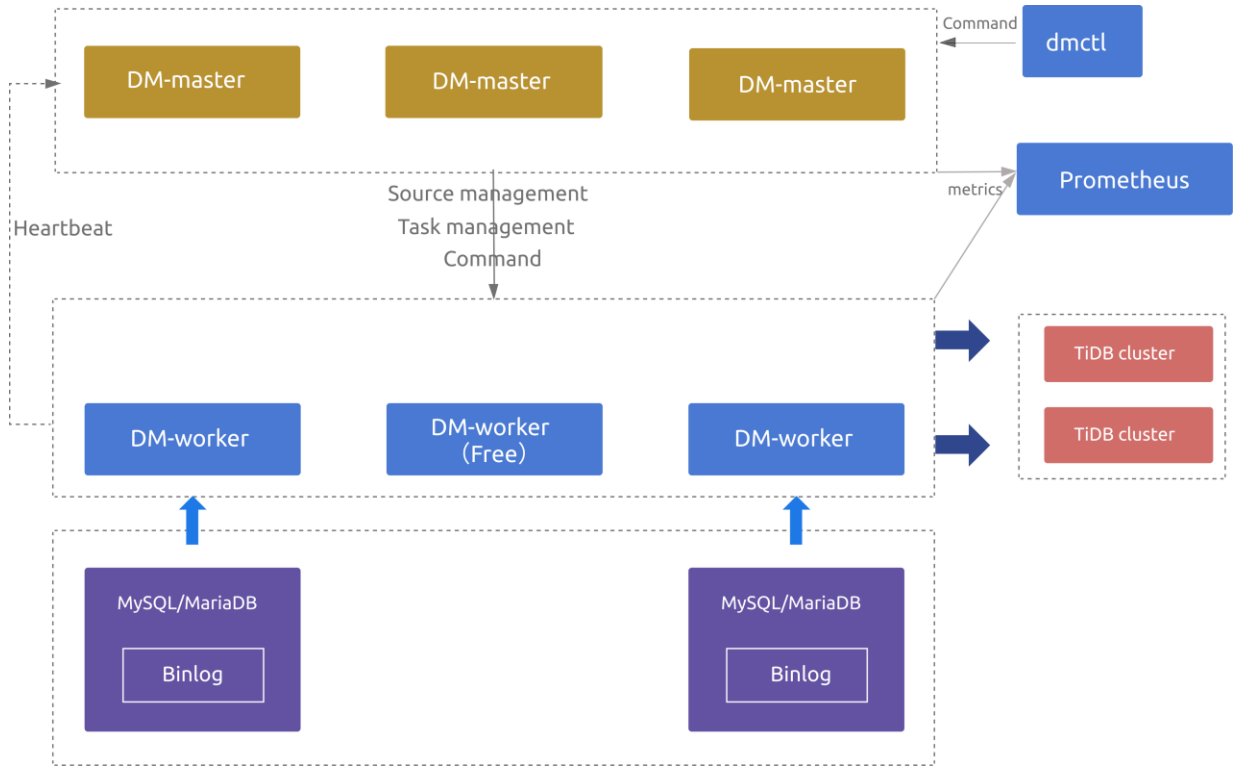


图 163: Data Migration architecture

10.6.2.1 架构组件

10.6.2.1.1 DM-master

DM-master 负责管理和调度数据迁移任务的各项操作。

- 保存 DM 集群的拓扑信息
- 监控 DM-worker 进程的运行状态
- 监控数据迁移任务的运行状态
- 提供数据迁移任务管理的统一入口
- 协调分库分表场景下各个实例分表的 DDL 迁移

10.6.2.1.2 DM-worker

DM-worker 负责执行具体的数据迁移任务。

- 将 binlog 数据持久化保存在本地
- 保存数据迁移子任务的配置信息
- 编排数据迁移子任务的运行
- 监控数据迁移子任务的运行状态

有关于 DM-worker 的更多介绍，详见DM-worker 简介。

10.6.2.1.3 dmctl

dmctl 是用来控制 DM 集群的命令行工具。

- 创建、更新或删除数据迁移任务
- 查看数据迁移任务状态
- 处理数据迁移任务错误
- 校验数据迁移任务配置的正确性

有关于dmctl 的使用介绍，详见dmctl 使用。

10.6.2.2 架构特性

10.6.2.2.1 高可用

当部署多个 DM-master 节点时，所有 DM-master 节点将使用内部嵌入的 etcd 组成集群。该 DM-master 集群用于存储集群节点信息、任务配置等元数据，同时通过 etcd 选举出leader 节点。该leader 节点用于提供集群管理、数据迁移任务管理相关的各类服务。因此，若可用的 DM-master 节点数超过部署节点的半数，即可正常提供服务。

当部署的 DM-worker 节点数超过上游 MySQL/MariaDB 节点数时，超出上游节点数的相关 DM-worker 节点默认将处于空闲状态。若某个 DM-worker 节点下线或与 DM-master leader 发生网络隔离，DM-master 能自动将与原 DM-worker 节点相关的数据迁移任务调度到其他空闲的 DM-worker 节点上（若原 DM-worker 节点为网络隔离状态，则其会自动停止相关的数据迁移任务）；若无空闲的 DM-worker 节点可供调度，则原 DM-worker 相关的数据迁移任务将暂时挂起，直到有空闲 DM-worker 节点后自动恢复。

> 注意：
>
> 当数据迁移任务处于全量导出或导入阶段时，该迁移任务暂不支持高可用，主要原因为：
>
> - 对于全量导出，MySQL 暂不支持指定从特定快照点导出，也就是说数据迁移任务被重新调度或重启后，无法继续从前一次中断时刻继续导出。
> - 对于全量导入，DM-worker 暂不支持跨节点读取全量导出数据，也就是说数据迁移任务被调度到的新 DM-worker 节点无法读取调度发生前原 DM-worker 节点上的全量导出数据。

10.6.3 TiDB Data Migration 快速上手指南

本文介绍如何快速体验使用数据迁移工具TiDB Data Migration (DM) 从 MySQL 迁移数据到TiDB。此文档用于快速体验 DM 产品功能特性，并不建议适合在生产环境中使用。

10.6.3.1 第 1 步：部署 DM 集群

1. 安装TiUP 工具并通过TiUP 快速部署dmctl。

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

2. 生成 DM 集群最小拓扑文件。

```
tiup dm template
```

3. 复制输出的配置信息，修改IP 地址后保存为 topology.yaml 文件，使用TiUP 部署 DM 集群。

```
tiup dm deploy dm-test 6.0.0 topology.yaml -p
```

10.6.3.2 第 2 步：准备数据源

你可以使用一个或多个 MySQL 实例作为上游数据源。

1. 为每一个数据源编写如下配置文件：

```
source-id: "mysql-01"

from:
  host: "127.0.0.1"
  user: "root"
  password: "fCxfQ9XKCezSzuCDOWf5dUD+LsKegSg=" # 使用 tiup dmctl --encrypt "123456" 加密。
  port: 3306
```

2. 使用如下命令将数据源增加至 DM 集群。其中，mysql-01.yaml 是上一步编写的配置文件。

```
tiup dmctl --master-addr=127.0.0.1:8261 operate-source create mysql-01.yaml # --master-addr
    ↪ 填写 master_servers 其中之一。
```

若环境中并不存在可供测试的 MySQL 实例，可以使用以下方式通过 Docker 快速创建一个测试实例。步骤如下：

1. 编写 MySQL 配置文件：

```
mkdir -p /tmp/mysqltest && cd /tmp/mysqltest

cat > my.cnf <<EOF
[mysqld]
bind-address      = 0.0.0.0
character-set-server=utf8
collation-server=utf8_bin
default-storage-engine=INNODB
transaction-isolation=READ-COMMITTED
server-id         =  100
binlog_format     = row
```

```
log_bin           = /var/lib/mysql/mysql-bin.log
show_compatibility_56 = ON
EOF
```

2. 使用 Docker 启动 MySQL 实例：

```
docker run --name mysql-01 -v /tmp/mysqltest:/etc/mysql/conf.d -e MYSQL_ROOT_PASSWORD=my-
    ↪ secret-pw -d -p 3306:3306 mysql:5.7
```

3. 待 MySQL 启动后，即可连接该实例。

> **注意：**
> 该命令仅适用于体验数据迁移过程，不能用于生产环境和压力测试。

```
mysql -uroot -p -h 127.0.0.1 -P 3306
```

10.6.3.3 第 3 步：准备下游数据库

可以选择已存在的TiDB 集群作为数据同步目标。

如果没有可以用于测试的TiDB 集群，则使用以下命令快速构建演示环境。

```
tiup playground
```

10.6.3.4 第 4 步：准备测试数据

在一个或多个数据源中创建测试表和数据。如果你使用已存在的 MySQL 数据库，且数据库中已有可用数据，可跳过这一步。

```
drop database if exists `testdm`;
create database `testdm`;
use `testdm`;
create table t1 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`),
    ↪ unique key(`uid`)) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
create table t2 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`),
    ↪ unique key(`uid`)) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
insert into t1 (id, uid, name) values (1, 10001, 'Gabriel García Márquez'), (2, 10002, 'Cien años
    ↪  de soledad');
insert into t2 (id, uid, name) values (3, 20001, 'José Arcadio Buendía'), (4, 20002, 'Úrsula
    ↪ Iguarán'), (5, 20003, 'José Arcadio');
```

10.6.3.5 第 5 步：编写数据同步任务

1. 创建任务的配置文件 `testdm-task.yaml`：

```
name: testdm
task-mode: all

target-database:
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: "" # 如果密码不为空，则推荐使用经过 dmctl 加密的密文

# 填写一个或多个所需同步的数据源信息
mysql-instances:
  - source-id: "mysql-01"
    block-allow-list:  "ba-rule1"

block-allow-list:
  ba-rule1:
    do-dbs: ["testdm"]
```

2. 使用 dmctl 创建任务：

```
tiup dmctl --master-addr 127.0.0.1:8261 start-task testdm-task.yaml
```

这样就成功创建了一个将 `mysql-01` 数据源迁移到TiDB 的任务。

10.6.3.6 第 6 步：查看迁移任务状态

在创建迁移任务之后，可以用 `dmctl query-status` 来查看任务的状态。

```
tiup dmctl --master-addr 127.0.0.1:8261 query-status testdm
```

10.6.4 DM 数据迁移最佳实践

TiDB Data Migration (DM) 是由PingCAP 开发维护的数据迁移同步工具，主要支持的源数据库类型为各类遵循MySQL 协议标准的关系型数据库，如 MySQL、 Percona MySQL、 MariaDB、 Amazon RDS for MySQL、 Amazon Aurora 等。

DM 的使用场景主要有：

- 从兼容 MySQL 的单一实例中全量和增量迁移数据到TiDB
- 将小数据量（小于 1 TB）分库分表 MySQL 合并迁移数据到TiDB
- 在业务数据中台、业务数据实时汇聚等数据中枢场景中，作为数据同步中间件来使用

本文档介绍了如何优雅高效地使用 DM，以及如何规避使用 DM 的常见误区。

10.6.4.1 性能边界定位

DM 的性能参数如下表所示。

参数	限制
最大同步节点（Work Nodes）	1000
最大同步任务数量	600
最大同步 QPS	30k QPS/worker
最大 Binlog 吞吐量	20 MB/s/worker
每个 Task 处理的表数量	无限制

- DM 支持同时管理 1000 个同步节点（Work Node），最大同步任务数量为 600 个。为了保证同步节点的高可用，应预留一部分同步节点作为备用节点。建议预留的节点数量为已开启同步任务的同步节点数量的 20% ~ 50%。
- 单机部署Work Node 数量。在服务器配置较好情况下，要保证每个Work Node 至少有 2 核 CPU 加 4G 内存的可用工作资源，并且应为主机预留 10% ~ 20% 的系统资源。
- 单个同步节点（Work Node），理论最大同步 QPS 为 30K QPS/worker（不同 Schema 和 workload 会有所差异），处理上游 Binlog 的能力最高为 20 MB/s/worker。
- 如果将 DM 作为需要长期使用的数据同步中间件，需要注意 DM 组件的部署架构。请参见DM-master 与 DM-woker 部署实践。

10.6.4.2 数据迁移前

在所有数据迁移之前，整体方案的设计是至关重要的。下面我们分别从业务侧要点及实施侧要点两个方面讲一下相关的实践经验和适用场景。

10.6.4.2.1 业务侧要点

为了让压力可以平均分配到多个节点上，在 Schema 设计上，分布式数据库与传统数据库差别很大，既要保证较低的业务迁移成本，又要保证迁移后应用逻辑的正确性。下面就从几个方面来看业务迁移前的最佳实践。

Schema 的设计中AUTO_INCREMENT 对业务的影响

TiDB 的 AUTO_INCREMENT 与 MySQL 的 AUTO_INCREMENT 整体上看是相互兼容的。但因为TiDB 作为分布式数据库，一般会有多个计算节点（client 端入口），应用数据写入时会将负载均分开，这就导致在有 AUTO_INCREMENT 列的表上，可能出现不连续的自增ID。详细原理参考AUTO_INCREMENT。

如果业务对自增ID 有强依赖，可以考虑使用SEQUENCE 函数。

是否使用聚簇索引

TiDB 在建表时可以声明为主键创建聚簇索引或非聚簇索引。下面介绍各方案的优势和劣势。

- 聚簇索引

  聚簇索引使用主键作为数据存储的handle ID（行ID），在使用主键查询时可以减少一次回表的操作，有效提升查询效能。但如果表有大量数据写入且主键使用AUTO_INCREMENT，非常容易造成数据存储的写入热点问题，导致集群整体效能不能充分利用，出现单存储节点的性能瓶颈问题。

- 非聚簇索引+shard row id bit

  使用非聚簇索引，再配合表提示 shard row id bit，可以在继续使用 AUTO_INCREMENT 的情况下有效避免数据写入热点的产生。但是由于多了一次回表操作，此时使用主键查询数据，查询性能将有所影响。

- 聚簇索引+外部分布式发号器

  如果想使用聚簇索引，但还希望ID 是单调递增的，那么可以考虑使用外部分布式发号器，如雪花算法(Snowflake)、Leaf 等，来解决问题。由应用程序产生序列ID，可以一定程度上保证ID 的单调递增性，同时也保留了使用聚簇索引带来的收益。但相关应用程序需要进行改造。

- 聚簇索引+AUTO_RANDOM

  此方案是目前分布式数据库既能避免出现写入热点问题，又能保留聚簇索引带来的查询收益的方案。整体改造也相对轻量，可以在业务切换使用TiDB 作为写库时，修改 Schema 属性来达到目的。如果在后续查询时一定要利用ID 列进行排序，可以使用AUTO_RANDOM ID 列左移 5 位来保证查询数据的顺序性。示例：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM, b varchar(255));
Select  a, a<<5 ,b from t order by a <<5 desc
```

下表汇总了不同使用场景的推荐方案和优劣势。

场景	推荐方案	优势	劣势
TiDB 未来作为主库使用，并会有大量数据写入。业务逻辑强依赖主键 ID 的连续性。	将表建立为非聚簇索引，并设置 SHARD_ROW_ID_BIT。使用 SEQUENCE 作为主键列。	可以有效避免数据写入热点，保证业务数据的连续性和单调递增。	数据写入的吞吐能力会下降（为保证数据写入连续性）；主键查询性能有所下降。

场景	推荐方案	优势	劣势
TiDB 未来作为主库使用，并会有大量数据写入。业务逻辑强依赖主键 ID 的递增特性。	将表建立为非聚簇索引，并设置 SHARD_ROW_ID_BIT；使用应用程序发号器来定义主键 ID。	可以有效避免数据写入热点；可以保证数据写入性能；可以有效保证业务数据是趋势性递增，但不能保证数据连续性。	对原有代码有一定的改造成本；外部发号器对时钟准确性有强依赖，引入新的故障风险点。

场景	推荐方案	优势	劣势
TiDB 未来作为主库使用，并会有大量数据写入。业务逻辑不依赖主键 ID 的连续性。	将表建立为聚簇索引表；主键列设置为 AUTO_RANDOM 属性。	可以有效避免数据写入热点；有非常有限的写入吞吐能力；主键查询性能优异；可以平滑将 AUTO_INCREMENT 属性切换为 AUTO_RANDOM 属性。	主键 ID 是完全随机的，业务数据排序建议使用插入时间列来完成。如果一定要使用主键 ID 排序，可以用 ID 左移 5 的方式查询，此方式查询的数据可以保证趋势递增的特性。

场景	推荐方案	优势	劣势
TiDB 未来作为只读的数据中台使用。	将表建立为非聚簇索引，并设置 SHARD_ROW_ID_BIT；使主键列维持与源数据库类型一致即可。	可以有效避免数据写入热点；改造成本低。	主键查询性能有所下降。

10.6.4.2.2 分库分表要点

分与合

DM 支持将上游分库分表的数据合并到下游TiDB 中的同一个表，这也是TiDB 推荐的一种方式。

除了数据合并场景外，另一个典型场景为数据归档场景。在此场景中，数据不断写入，随着时间流逝，大量的数据从热数据逐渐转变为温冷数据。在TiDB 中，你可以通过 Placement Rules 放置规则来按照一定规则对数据设置不同的放置规则，而最小粒度即为分区表(Partition)。

所以建议在遇到有大规模数据写入的场景，一开始就规划好未来是否需要归档或者有冷热数据分别存储在不同介质的需要。如果有，那么在迁移前请设置好分区表规则（目前TiDB 还不支持 Table Rebuild 操作）。避免因为初期考虑不周，导致后期需要重新建表及重新导入数据。

悲观 DDL 锁与乐观 DDL 锁

DM 默认会使用悲观 DDL 锁模式。在分库分表迁移与同步场景中，上游相关分表发生 Schema 变更后，会阻断后续的 DML 向下游TiDB 写入。此时需要等待上游各分表的 Schema 都变更完毕并自动确认所有分表的结构一致后，从同步阻断记录点继续数据同步。

- 如果上游 Schema 变更时间较长，可能导致上游 Binlog 被清理，此问题可以通过开启 DM 的 Relay log 功能来避免。
- 如果不希望因为上游 Schema 变更阻塞数据写入，可以考虑使用乐观 DDL 锁模式。此时 DM 在发现上游分表 Schema 变更时也不会阻断数据同步，而是会持续同步数据。但如果同步期间 DM 发现上下游数据格式不兼容，将停止同步任务。此时需要人工介入处理。

下表汇总了乐观 DDL 锁和悲观 DDL 锁的优劣势。

场景	优势	劣势
悲观 DDL 锁（默认）	最大程度保证数据同步任务的可靠性	如果分表较多，将长时间阻断数据同步任务。并有可能因为上游的 Binlog 已被清理而导致同步中断。可以通过开启 DM-worker 的 Relay log 来避免问题。详情请参考Relay log 的使用.

场景	优势	劣势
乐观 DDL 锁	数据同步任务基本不会出现相关阻塞延迟	此模式下需要保证 Schema 变更的兼容性（增加列是否具有默认值）。如果考虑不周，可能出现未被发现的上下游数据不一致问题。更多限制，请参考乐观模式下分库分表合并迁移。

10.6.4.2.3 其他限制与影响

上下游的数据类型

这里主要需要考虑上下游的数据类型问题。TiDB 目前支持绝大部分 MySQL 的数据类型。但一些特殊类型尚不支持（如空间类型）。关于数据类型的兼容性，请参考数据类型概述。

字符集与排序规则

自 TiDB v6.0.0 以后，默认使用新排序规则。如果需要TiDB 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，需要在集群创建时声明，将new_collations_enabled_on_first_bootst
↪ 的值设为 true。更详细信息请参考字符集和排序规则。

TiDB 默认使用的字符集为utf8mb4。建议同步上下游及应用统一使用utf8mb4。如果上游有显式指定的字符集或者排序规则，需要确认TiDB 是否支持。

从 v6.0.0 起，TiDB 支持GBK 字符集。有关字符集的限制详见：

- 字符集和排序规则
- GBK 兼容情况

10.6.4.2.4 实施侧要点

DM-master 与 DM-woker 部署实践

DM 整体架构分为 DM-master 与 DM-worker。

- DM-master 主要负责同步任务的元数据管理，以及 DM-worker 的中心调度，是整个 DM 平台的核心。所以 DM-master 可以部署为集群模式，以保证 DM 同步平台的可用性。
- DM-worker 负责执行上下游同步任务，是无状态节点。最多可以部署 1000 个节点。在需要将 DM 作为数据同步平台的场景，可以预留一部分空闲的 DM-worker，以保证同步任务的高可用。

同步任务规划

分库分表场景。在分库分表迁移场景，根据上游分库分表的种类进行同步任务的拆分，如 usertable_1~50 和 Logtable_1~50 是两类分表，那么就应该分别建 2 个 Task 任务进行同步。这样做的好处是可有效简化同步任务模板的复杂度，并有效控制数据同步中断的影响范围。

大规模数据迁移同步场景。可以参考以下思路进行 Task 任务拆分：

- 如果上游需要同步多个数据库，可以按照不同数据库拆分 Task。
- 根据上游写入压力拆分任务。即把上游 DML 操作频繁的表，拆分到单独的 Task 任务中，将其他没有频繁 DML 操作的表使用另一个 Task 任务进行同步。此方式可在一定程度上加速同步任务的推进能力。尤其是在上游有大量 Log 写入某张表，但业务关注的是其他表时，此方法可以有效解决此类问题。

请注意，拆分同步任务后只能保证数据同步的最终一致性，实时一致性因各种原因可能出现较大偏差。

下表给出了在不同的数据迁移与同步场景下部署 DM-master 与 DM-worker 的推荐方案。

场景	DM-master 部署	DM-worker 部署
小规模数据（1 TB 以下），一次性数据迁移场景	部署 1 个 DM-master 节点	根据上游数据源数量，部署 1 ~ N 个 DM-worker 节点。一般情况下 1 个 DM-worker 节点。
大规模数据（1 TB 以上）及分库分表，一次性数据迁移场景	推荐部署 3 个 DM-master 节点，来保证在长时间数据迁移时 DM 集群的可用性	根据数据源数量或同步任务数量部署 DM-Worker 节点。推荐多部署 1~3 个空闲 DM-Worker 节点。

场景	DM-master 部署	DM-worker 部署
长期数据同步迁移场景	务必部署 3 个 DM-master 节点。如在云上部署，尽量将 DM-master 部署在不同的可用区（AZ）	根据数据源数量或同步任务数量部署 DM-worker 节点。务必部署实际需要 DM-worker 节点数量的 1.5 ~ 2 倍的 DM-worker 节点数量。

上游数据源选择与设置

DM 支持存量数据迁移，但在做全量迁移时会对整库进行全量数据备份。DM 采用的备份方式为并行逻辑备份，备份 MySQL 期间会加上全局只读锁 `FLUSH TABLES WITH READ LOCK`。此时会短暂地阻塞上游数据库的 DML 和 DDL 操作。所以强烈建议使用上游的备库来进行全量备份，并同时在数据源开启 GTID 的功能 (`enable-gtid: true`)。这样既可避免存量迁移时对上游业务的影响，也可以在增量同步期间再切换到上游主库节点降低数据同步的延迟。切换上游 MySQL 数据源的方法，请参考切换 DM-worker 与上游 MySQL 实例的连接。

下面是一些特殊场景下的注意事项：

- 只能在上游主库进行全量备份

  在该场景中， 可以在同步任务配置中设置一致性参数为 `none`，`mydumpers.global.extra-args: "--consistency none"` 避免给主库加全局只读锁，但有可能破坏全量备份的数据一致性，导致最终上下游数据不一致。

- 利用备份快照解决存量迁移（只适用AWS 上 MySQL RDS 和 Aurora RDS 的迁移）

  如果要迁移的数据库正好为AWS MySQL RDS 或者Aurora RDS，可以利用RDS Snapshot 备份将Amazon S3 中的备份数据直接迁移到TiDB，以此保证存量数据迁移的一致性。整个操作流程以及后续增量同步方法，请参考从 Amazon Aurora 迁移数据到TiDB。

10.6.4.2.5 配置细节详解

大小写

TiDB 默认情况下对 Schema name 大小写不敏感，即 `lower_case_table_names:2`。但上游 MySQL 大多为 Linux 系统，默认对大小写敏感。此时需要注意，在 DM 数据同步任务设置时将 `case-sensitive` 设置为 `true`，保证可以正确同步上游的 Schema。

特殊情况下，比如上游一个数据库中，既有大写表如 `Table`，又有小写表如 `table`，那么 Schema 创建时将报错:

```
ERROR 1050 (42S01): Table '{tablename}' already exists
```

过滤规则

在配置数据源时即可配置过滤规则。配置方法请参考数据迁移任务配置向导。

配置过滤规则的好处有：

- 可以减少下游处理 Binglog Event 的数量，提升同步效能
- 可以减少不必要的Relay log 的落盘，节约磁盘空间

> 注意：
>
> 在分库分表场景中，如果你在数据源配置了过滤规则，请确保数据源与同步任务中设置的过滤规则相匹配。如果不匹配，将会导致同步任务长期接收不到增量数据。

Relay Log 的使用

MySQL 的主从复制在 Secondary 端会保存一份 Relay log，以此保证异步复制的可靠性与效能。DM 也支持在 DM-worker 侧保存一份Relay log，并可设置存储位置、过期清理时间等信息。此功能适用于以下场景：

- 在进行全量+增量数据迁移时，因为全量迁移数据量较大，整个过程耗费时间超过了上游 Binlog 归档的时间，导致增量同步任务不能正常启动，如果开启Relay Log 在全量同步启动同时，DM-worker 即会开始接收Relay log，避免增量任务启动失败。
- 在使用 DM 进行长期数据同步的场景中，有时因为各种原因导致同步任务长时间阻塞，此时开启了 Relay log 功能，可以有效应对同步任务阻塞而导致的上游 Binglog 被回收问题。

在使用Relay Log 功能时也会有一定的限制。DM 支持高可用，当某个 DM-worker 出现故障，会尝试将空闲的 DM-worker 实例提升为工作实例，如果此时上游 Binlog 没有包含必要的同步日志，将可能出现同步中断情况。此时需要人工干预，尽快将Relay log 复制到新的 DM-worker 节点上来，并修改相应的Relay meta 文件。具体方法请参考故障处理。

上游使用在线变更工具 PT-osc/GH-ost

在日常运维 MySQL 时，想要在线变更表结构，一般会使用 PT-osc/GH-ost 这类工具，以此保证 DDL 变更对线上业务的影响最小。但整个过程会被如实地记录到 MySQL Binlog 中，如果全部同步到下游TiDB，将产生大量的写放大，既不高效也不经济。DM 在设置 Task 任务时候可以设置支持三方数据同步工具 PT-osc 或 GH-ost，配置后将不再同步大量冗余数据，并且能保证数据同步的一致性。具体设置方式请参考迁移使用GH-ost/PT-osc 的源数据库。

10.6.4.3 数据迁移中

在这个环节基本上遇到的问题都是Troubleshooting 类的问题，在这里给大家列举一些场景的问题及处理方式。

10.6.4.3.1 上游与下游 Schema 不一致

常见报错信息：

- `messages: Column count doesn't match value count: 3 (columns)vs 2 (values)`
- `Schema/Column doesn't match`

此类问题主要原因是下游TiDB 中增加或修改了索引，或者下游比上游更多列。当出现此类的同步报错信息时，请检查是否是上下游 Schema 不一致导致的同步中断。

要解决此类问题，只需将 DM 中缓存的 Schema 信息更新成与下游TiDB Schema 一致即可。具体方法参考管理迁移表的表结构。

如果是下游比上游多列的场景，请参考下游存在更多列的迁移场景。

10.6.4.3.2 处理因为 DDL 中断的数据同步任务

DM 支持跳过或者替换导致同步任务中断的 DDL 语句。并且针对是否为分库分表合并场景有对应不同的操作，具体请参考处理出错的 DDL 语句。

10.6.4.4 数据迁移后的数据校验

在完成数据迁移后，建议对新旧数据进行数据一致性校验。TiDB 提供了相应的同步工具sync-diff-inspector 来帮助你完成数据校验工作。

通过管理 DM 中的同步任务，sync-diff-inspector 可以自动管理需要进行数据一致性检查的 Table 列表，相较之前的手动配置更加的高效。具体参考基于 DM 同步场景下的数据校验。

自 DM v6.2.0 版本开始，DM 支持在增量同步的同时进行数据校验。具体参考DM 增量数据校验。

10.6.4.5 数据长期同步

如果将 DM 作为持续的数据同步平台，建议一定要做好必要的元信息备份。一方面是保证同步集群故障重建的能力，另一方面可以实现同步任务的版本控制。具体实现方式参考导出和导入集群的数据源和任务配置。

10.6.5 部署 DM 集群

10.6.5.1 TiDB Data Migration 集群软硬件环境需求

TiDB Data Migration (DM) 支持主流的 Linux 操作系统，具体版本要求见下表：

Linux 操作系统	版本
Red Hat Enterprise Linux	7.3 及以上
CentOS	7.3 及以上
Oracle Enterprise Linux	7.3 及以上
Ubuntu LTS	16.04 及以上

DM 可以在Intel 架构服务器环境及主流虚拟化环境中部署和运行。

10.6.5.1.1 服务器建议配置

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

开发及测试环境

组件	CPU	内存	本地存储	网络	实例数量（最低要求）
DM-master	4 核 +	8 GB+	SAS，200 GB+	千兆网卡	1
DM-worker	8 核 +	16 GB+	SAS，200 GB+（大于迁移数据的大小）	千兆网卡	上游 MySQL 实例的数量

注意：

- 在功能验证的测试环境中的 DM-master 和 DM-worker 可以部署在同一台服务器上。
- 如进行性能相关的测试，避免采用低性能存储和网络硬件配置，防止对测试结果的正确性产生干扰。
- 如果仅验证功能，可以单机部署一个 DM-master，DM-worker 部署的数量至少是上游 MySQL 实例的数量。为了保证高可用性，建议部署更多的 DM-worker。
- DM-worker 在 dump 和 load 阶段需要存储全量数据，因此 DM-worker 的磁盘空间需要大于需要迁移数据的总量；如果迁移任务开启了relay log，DM-worker 也需要一定的磁盘空间来存储上游的binlog 数据。

生产环境

组件	CPU	内存	硬盘类型	网络	实例数量（最低要求）
DM-master	4 核 +	8 GB+	SAS，200 GB+	千兆网卡	3
DM-worker	16 核 +	32 GB+	SSD，200 GB+（大于迁移数据的大小）	万兆网卡	大于上游 MySQL 实例的数量
监控	8 核 +	16 GB+	SAS，200 GB+	千兆网卡	1

注意：

- 在生产环境中，不建议将 DM-master 和 DM-worker 部署和运行在同一个服务器上，以防 DM-worker 对磁盘的写入干扰 DM-master 高可用组件使用磁盘。
- 在遇到性能问题时可参照配置调优尝试修改任务配置。调优效果不明显时，可以尝试升级服务器配置。

10.6.5.1.2 下游数据库所需空间

目标TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标TiKV 集群的总存储空间必须大于数据源大小× 副本数量× 2。例如集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间
- RocksDB 的空间放大效应

可以用下面 SQL 语句统计信息表的data_length 字段估算数据量：

统计所有 schema 大小，单位 MiB，注意修改 ${schema_name}

```
select table_schema,sum(data_length)/1024/1024 as data_length,sum(index_length)/1024/1024 as
    ↪ index_length,sum(data_length+index_length)/1024/1024 as sum from information_schema.
    ↪ tables where table_schema = "${schema_name}" group by table_schema;
```

统计最大单表，单位 MiB，注意修改 ${schema_name}

```
select table_name,table_schema,sum(data_length)/1024/1024 as data_length,sum(index_length)
    ↪ /1024/1024 as index_length,sum(data_length+index_length)/1024/1024 as sum from
    ↪ information_schema.tables where table_schema = "${schema_name}" group by table_name,
    ↪ table_schema order by sum  desc limit 5;
```

10.6.5.2 使用TiUP 部署 DM 集群

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

目前TiUP 可以支持部署 v2.0 及以上版本的 DM。本文将介绍不同集群拓扑的具体部署步骤。

注意：

如果部署机器的操作系统支持 SELinux，请确保 SELinux 处于关闭状态。

10.6.5.2.1 前提条件

当 DM 执行全量数据复制任务时，每个 DM-worker 只绑定一个上游数据库。DM-worker 首先在上游导出全部数据，然后将数据导入下游数据库。因此，DM-worker 的主机空间需要容纳所有要导出的上游表，具体存储路径在后续创建迁移任务时指定。

另外，部署 DM 集群需参照DM 集群软硬件环境需求，满足相应要求。

10.6.5.2.2 第 1 步：在中控机上安装TiUP 组件

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

1. 执行如下命令安装TiUP 工具：

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

安装完成后，~/.bashrc 已将TiUP 加入到路径中，你需要新开一个终端或重新声明全局变量 source ~/.bashrc 来使用TiUP。

2. 安装TiUP DM 组件：

```
tiup install dm dmctl
```

10.6.5.2.3 第 2 步：编辑初始化配置文件

请根据不同的集群拓扑，编辑TiUP 所需的集群初始化配置文件。

请根据配置文件模板，新建一个配置文件 topology.yaml。如果有其他组合场景的需求，请根据多个模板自行调整。

可以使用 tiup dm template > topology.yaml 命令快速生成配置文件模板。

部署 3 个 DM-master、3 个 DM-worker 与 1 个监控组件的配置如下：

```
#全局变量适用于配置中的其他组件。如果组件实例中缺少一个特定值，则相应的全局变量将用作默认值。
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/dm-deploy"
  data_dir: "/dm-data"

server_configs:
  master:
    log-level: info
    # rpc-timeout: "30s"
    # rpc-rate-limit: 10.0
    # rpc-rate-burst: 40
  worker:
    log-level: info
```

```
master_servers:
  - host: 10.0.1.11
    name: master1
    ssh_port: 22
    port: 8261
    # peer_port: 8291
    # deploy_dir: "/dm-deploy/dm-master-8261"
    # data_dir: "/dm-data/dm-master-8261"
    # log_dir: "/dm-deploy/dm-master-8261/log"
    # numa_node: "0,1"
    # 下列配置项用于覆盖 `server_configs.master` 的值。
    config:
      log-level: info
      #  rpc-timeout: "30s"
      # rpc-rate-limit: 10.0
      # rpc-rate-burst: 40
  - host: 10.0.1.18
    name: master2
    ssh_port: 22
    port: 8261
  - host: 10.0.1.19
    name: master3
    ssh_port: 22
    port: 8261
#### 如果不需要确保 DM 集群高可用，则可只部署 1 个 DM-master 节点，且部署的 DM-worker
    '→ 节点数量不少于上游待迁移的 MySQL/MariaDB 实例数。
#### 如果需要确保 DM 集群高可用，则推荐部署 3 个 DM-master 节点，且部署的 DM-worker
    '→ 节点数量大于上游待迁移的 MySQL/MariaDB 实例数（如 DM-worker 节点数量比上游实例数多 2 个
    '→ ）。
worker_servers:
  - host: 10.0.1.12
    ssh_port: 22
    port: 8262
    # deploy_dir: "/dm-deploy/dm-worker-8262"
    # log_dir: "/dm-deploy/dm-worker-8262/log"
    # numa_node: "0,1"
    # 下列配置项用于覆盖 `server_configs.worker` 的值。
    config:
      log-level: info
  - host: 10.0.1.19
    ssh_port: 22
    port: 8262

monitoring_servers:
```

```
  - host: 10.0.1.13
    ssh_port: 22
    port: 9090
    # deploy_dir: "/tidb-deploy/prometheus-8249"
    # data_dir: "/tidb-data/prometheus-8249"
    # log_dir: "/tidb-deploy/prometheus-8249/log"

grafana_servers:
  - host: 10.0.1.14
    port: 3000
    # deploy_dir: /tidb-deploy/grafana-3000

alertmanager_servers:
  - host: 10.0.1.15
    ssh_port: 22
    web_port: 9093
    # cluster_port: 9094
    # deploy_dir: "/tidb-deploy/alertmanager-9093"
    # data_dir: "/tidb-data/alertmanager-9093"
    # log_dir: "/tidb-deploy/alertmanager-9093/log"
```

注意：

- 不建议在一台主机上运行太多 DM-worker。每个 DM-worker 至少应有 2 核 CPU 和 4 GiB 内存。
- 需要确保以下组件间端口可正常连通：
  - 各 DM-master 节点间的 `peer_port`（默认为 8291）可互相连通。
  - 各 DM-master 节点可连通所有 DM-worker 节点的 `port`（默认为 8262）。
  - 各 DM-worker 节点可连通所有 DM-master 节点的 `port`（默认为 8261）。
  - TiUP 节点可连通所有 DM-master 节点的 `port`（默认为 8261）。
  - TiUP 节点可连通所有 DM-worker 节点的 `port`（默认为 8262）。

更多 `master_servers.host.config` 参数说明，请参考master parameter；更多 `worker_servers.host.config` 参数说明，请参考 worker parameter。

10.6.5.2.4 第 3 步：执行部署命令

注意：

通过TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：

- 如果是密钥方式，可以通过 -i 或者 --identity_file 来指定密钥的路径；
- 如果是密码方式，可以通过 -p 进入密码交互窗口；
- 如果已经配置免密登录目标机，则不需填写认证。

```
tiup dm deploy dm-test ${version} ./topology.yaml --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

以上部署命令中：

- 通过TiUP DM 部署的集群名称为 dm-test。
- 部署版本为 ${version}，可以通过执行 tiup list dm-master 来查看TiUP 支持的最新版本。
- 初始化配置文件为 topology.yaml。
- --user root：通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- -i 及 -p：非必选项，如果已经配置免密登录目标机，则不需填写，否则选择其一即可。-i 为可登录到目标机的root 用户（或 --user 指定的其他用户）的私钥，也可使用 -p 交互式输入该用户的密码。
- TiUP DM 使用内置的 SSH 客户端，如需使用系统自带的 SSH 客户端，请参考TiUP DM 文档中使用中控机系统自带的 SSH 客户端连接集群章节进行设置。

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

10.6.5.2.5 第 4 步：查看TiUP 管理的集群情况

```
tiup dm list
```

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

```
Name  User  Version  Path                                       PrivateKey
----  ----  -------  ----                                       ----------
dm-test  tidb  ${version}  /root/.tiup/storage/dm/clusters/dm-test  /root/.tiup/storage/dm/
    ↪ clusters/dm-test/ssh/id_rsa
```

10.6.5.2.6 第 5 步：检查部署的 DM 集群情况

例如，执行如下命令检查 dm-test 集群情况：

```
tiup dm display dm-test
```

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

10.6.5.2.7 第 6 步：启动集群

```
tiup dm start dm-test
```

预期结果输出 Started cluster `dm-test` successfully 表示启动成功。

10.6.5.2.8 第 7 步：验证集群运行状态

通过以下TiUP 命令检查集群状态：

```
tiup dm display dm-test
```

在输出结果中，如果 Status 状态信息为 Up，说明集群状态正常。

10.6.5.2.9 第 8 步：使用dmctl 管理迁移任务

dmctl 是用来控制集群运行命令的工具，推荐通过TiUP 获取该工具。

dmctl 支持命令模式与交互模式，具体请见使用dmctl 运维集群。

10.6.5.3 使用TiUP 离线镜像部署 DM 集群

本文介绍如何使用TiUP 离线部署 DM 集群，具体的操作步骤如下。

10.6.5.3.1 第 1 步：准备TiUP 离线组件包

- 在线环境中安装TiUP 包管理器工具。

  1. 执行如下命令安装TiUP 工具：

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

  2. 重新声明全局环境变量：

     ```
     source .bash_profile
     ```

  3. 确认TiUP 工具是否安装：

     ```
     which tiup
     ```

- 使用TiUP 制作离线镜像。

  1. 在一台和外网相通的机器上拉取需要的组件：

     ```
     # 将 ${version} 修改成实际需要的版本
     tiup mirror clone tidb-dm-${version}-linux-amd64 --os=linux --arch=amd64 \
         --dm-master=${version} --dm-worker=${version} --dmctl=${version} \
         --alertmanager=v0.17.0 --grafana=v4.0.3 --prometheus=v4.0.3 \
         --tiup=v$(tiup --version|grep 'tiup'|awk -F ' ' '{print $1}') --dm=v$(tiup --
             ↪ version|grep 'tiup'|awk -F ' ' '{print $1}')
     ```

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

  2. 通过tar 命令将该组件包打包然后发送到隔离环境的中控机：

     ```
     tar czvf tidb-dm-${version}-linux-amd64.tar.gz tidb-dm-${version}-linux-amd64
     ```

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

10.6.5.3.2 第 2 步: 部署离线环境TiUP 组件

将离线包发送到目标集群的中控机后，执行以下命令安装TiUP 组件：

```
#### 将 ${version} 修改成实际需要的版本
tar xzvf tidb-dm-${version}-linux-amd64.tar.gz
sh tidb-dm-${version}-linux-amd64/local_install.sh
source /home/tidb/.bash_profile
```

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

若需将镜像切换到其他目录，可以通过手动执行 `tiup mirror set <mirror-dir>` 进行切换，切换后如果再想切换成官方镜像，可执行 `tiup mirror set https://tiup-mirrors.pingcap.com`。

10.6.5.3.3 第 3 步：编辑初始化配置文件

请根据不同的集群拓扑，编辑TiUP 所需的集群初始化配置文件。

请根据配置文件模板，新建一个配置文件 `topology.yaml`。如果有其他组合场景的需求，请根据多个模板自行调整。

部署 3 个 DM-master、 3 个 DM-worker 与 1 个监控组件的配置如下：

```
---
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/home/tidb/dm/deploy"
  data_dir: "/home/tidb/dm/data"
  # arch: "amd64"

master_servers:
  - host: 172.19.0.101
  - host: 172.19.0.102
  - host: 172.19.0.103

worker_servers:
  - host: 172.19.0.101
  - host: 172.19.0.102
  - host: 172.19.0.103

monitoring_servers:
  - host: 172.19.0.101

grafana_servers:
  - host: 172.19.0.101

alertmanager_servers:
```

```
  - host: 172.19.0.101
```

注意：

- 如果不需要确保 DM 集群高可用，则可只部署 1 个 DM-master 节点，且部署的 DM-worker 节点数量不少于上游待迁移的 MySQL/MariaDB 实例数。
- 如果需要确保 DM 集群高可用，则推荐部署 3 个 DM-master 节点，且部署的 DM-worker 节点数量大于上游待迁移的 MySQL/MariaDB 实例数（如 DM-worker 节点数量比上游实例数多 2 个）。
- 对于需要全局生效的参数，请在配置文件中 `server_configs` 的对应组件下配置。
- 对于需要某个节点生效的参数，请在具体节点的 `config` 中配置。
- 配置的层次结构使用 `.` 表示。如：`log.slow-threshold`。更多格式说明，请参考 TiUP 配置参数模版。
- 更多参数说明，请参考master config.toml.example、worker config.toml.example。
- 需要确保以下组件间端口可正常连通：
  - 各 DM-master 节点间的 `peer_port`（默认为 8291）可互相连通。
  - 各 DM-master 节点可连通所有 DM-worker 节点的 `port`（默认为 8262）。
  - 各 DM-worker 节点可连通所有 DM-master 节点的 `port`（默认为 8261）。
  - TiUP 节点可连通所有 DM-master 节点的 `port`（默认为 8261）。
  - TiUP 节点可连通所有 DM-worker 节点的 `port`（默认为 8262）。

10.6.5.3.4 第 4 步：执行部署命令

注意：

通过TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：

- 如果是密钥方式，可以通过 `-i` 或者 `--identity_file` 来指定密钥的路径；
- 如果是密码方式，可以通过 `-p` 进入密码交互窗口；
- 如果已经配置免密登录目标机，则不需填写认证。

```
tiup dm deploy dm-test ${version} ./topology.yaml --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

以上部署命令中：

- 通过TiUP DM 部署的集群名称为 `dm-test`。
- 部署版本为 `${version}`，可以通过执行 `tiup list dm-master` 来查看TiUP 支持的最新版本。

- 初始化配置文件为 `topology.yaml`。
- `--user root`：通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- `-i` 及 `-p`：非必选项，如果已经配置免密登录目标机，则不需填写，否则选择其一即可。`-i` 为可登录到目标机的root 用户（或 `--user` 指定的其他用户）的私钥，也可使用 `-p` 交互式输入该用户的密码。
- TiUP DM 使用内置的 SSH 客户端，如需使用系统自带的 SSH 客户端，请参考TiUP DM 文档中使用中控机系统自带的 SSH 客户端连接集群章节进行设置。

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

10.6.5.3.5 第 5 步：查看TiUP 管理的集群情况

```
tiup dm list
```

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

```
Name  User  Version  Path                                       PrivateKey
----  ----  -------  ----                                       ----------
dm-test  tidb  ${version}  /root/.tiup/storage/dm/clusters/dm-test  /root/.tiup/storage/dm/
    ↪ clusters/dm-test/ssh/id_rsa
```

10.6.5.3.6 第 6 步：检查部署的 DM 集群情况

例如，执行如下命令检查 `dm-test` 集群情况：

```
tiup dm display dm-test
```

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

10.6.5.3.7 第 7 步：启动集群

```
tiup dm start dm-test
```

预期结果输出 `Started cluster `dm-test` successfully` 表示启动成功。

10.6.5.3.8 第 8 步：验证集群运行状态

通过以下TiUP 命令检查集群状态：

```
tiup dm display dm-test
```

在输出结果中，如果 Status 状态信息为 Up，说明集群状态正常。

10.6.5.4 使用 DM binary 部署 DM 集群

本文将介绍如何使用 DM binary 快速部署 DM 集群。

> 注意：
> 对于生产环境，推荐使用TiUP 部署 DM 集群及相关监控组件。

10.6.5.4.1 下载 DM 安装包

DM 安装包位于TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

10.6.5.4.2 使用样例

假设在五台服务器上部署两个 DM-worker 实例和三个 DM-master 实例。各个节点的信息如下：

实例	服务器地址	端口
DM-master1	192.168.0.4	8261
DM-master2	192.168.0.5	8261
DM-master3	192.168.0.6	8261
DM-worker1	192.168.0.7	8262
DM-worker2	192.168.0.8	8262

下面以此为例，说明如何部署 DM。

> 注意：
>
> - 在单机部署多个 DM-master 或 DM-worker 时，需要确保每个实例的端口以及运行命令的当前目录各不相同。
> - 如果不需要确保 DM 集群高可用，则可只部署 1 个 DM-master 节点，且部署的 DM-worker 节点数量不少于上游待迁移的 MySQL/MariaDB 实例数。
> - 如果需要确保 DM 集群高可用，则推荐部署 3 个 DM-master 节点，且部署的 DM-worker 节点数量大于上游待迁移的 MySQL/MariaDB 实例数（如 DM-worker 节点数量比上游实例数多 2 个）。
> - 需要确保以下组件间端口可正常连通：
>   - 各 DM-master 节点间的 8291 端口可互相连通。
>   - 各 DM-master 节点可连通所有 DM-worker 节点的 8262 端口。
>   - 各 DM-worker 节点可连通所有 DM-master 节点的 8261 端口。

部署 DM-master

DM-master 提供命令行参数和配置文件两种配置方式。

使用命令行参数部署 DM-master

DM-master 的命令行参数说明：

```
./dm-master --help
```

```
Usage of dm-master:
  -L string
        log level: debug, info, warn, error, fatal (default "info")
  -V    prints version and exit
  -advertise-addr string
        advertise address for client traffic (default "${master-addr}")
  -advertise-peer-urls string
        advertise URLs for peer traffic (default "${peer-urls}")
  -config string
        path to config file
  -data-dir string
        path to the data directory (default "default.${name}")
  -initial-cluster string
        initial cluster configuration for bootstrapping, e.g. dm-master=http://127.0.0.1:8291
  -join string
        join to an existing cluster (usage: cluster's "${master-addr}" list, e.g.
            ↪ "127.0.0.1:8261,127.0.0.1:18261"
  -log-file string
        log file path
  -master-addr string
        master API server and status addr
  -name string
        human-readable name for this DM-master member
  -peer-urls string
        URLs for peer traffic (default "http://127.0.0.1:8291")
  -print-sample-config
        print sample config file of dm-worker
```

> 注意：
> 某些情况下，无法使用命令行参数来配置 DM-master，因为有的配置并未暴露给命令行。

使用配置文件部署 DM-master

推荐使用配置文件，把以下配置文件内容写入到 `conf/dm-master1.toml` 中。

DM-master 的配置文件：

```
#### Master Configuration.

name = "master1"

#### 日志配置
log-level = "info"
log-file = "dm-master.log"

#### DM-master 监听地址
master-addr = "192.168.0.4:8261"

#### DM-master 节点的对等 URL
peer-urls = "192.168.0.4:8291"

#### 初始集群中所有 DM-master 的 advertise-peer-urls 的值
initial-cluster = "master1=http://192.168.0.4:8291,master2=http://192.168.0.5:8291,master3=http
    ↪ ://192.168.0.6:8291"
```

在终端中使用下面的命令运行 DM-master：

> 注意：
> 执行该命令后控制台不会输出日志，可以通过 `tail -f dm-master.log` 查看运行日志。

```
./dm-master -config conf/dm-master1.toml
```

对于 DM-master2 和 DM-master3，修改配置文件中的 `name` 为 `master2` 和 `master3`，并将 `peer-urls` 的值改为 `192.168.0.5:8291` 和 `192.168.0.6:8291` 即可。

部署 DM-worker

DM-worker 提供命令行参数和配置文件两种配置方式。

使用命令行参数部署 DM-worker

查看 DM-worker 的命令行参数说明：

```
./dm-worker --help
```

```
Usage of worker:
  -L string
        log level: debug, info, warn, error, fatal (default "info")
  -V    prints version and exit
  -advertise-addr string
        advertise address for client traffic (default "${worker-addr}")
```

```
  -config string
        path to config file
  -join string
        join to an existing cluster (usage: dm-master cluster's "${master-addr}")
  -keepalive-ttl int
        dm-worker's TTL for keepalive with etcd (in seconds) (default 10)
  -log-file string
        log file path
  -name string
        human-readable name for DM-worker member
  -print-sample-config
        print sample config file of dm-worker
  -worker-addr string
        listen address for client traffic
```

> **注意：**
>
> 某些情况下，无法使用命令行参数的方法来配置 DM-worker，因为有的配置并未暴露给命令行。

使用配置文件部署 DM-worker

推荐使用配置文件来配置 DM-worker，把以下配置文件内容写入到 `conf/dm-worker1.toml` 中。

DM-worker 的配置文件：

```
#### Worker Configuration.

name = "worker1"

#### 日志配置
log-level = "info"
log-file = "dm-worker.log"

#### DM-worker 的地址
worker-addr = ":8262"

#### 对应集群中 DM-master 配置中的 master-addr
join = "192.168.0.4:8261,192.168.0.5:8261,192.168.0.6:8261"
```

在终端中使用下面的命令运行 DM-worker：

```
./dm-worker -config conf/dm-worker1.toml
```

对于 DM-worker2，修改配置文件中的 `name` 为 `worker2` 即可。

这样，DM 集群就部署成功了。

10.6.5.5 在 Kubernetes 环境中部署

10.6.6 入门指南

10.6.6.1 创建 TiDB Data Migration 数据源

> **注意：**
>
> 在创建数据源之前，你需要先使用 TiUP 部署 DM 集群。

本文档介绍如何为 TiDB Data Migration (DM) 的数据迁移任务创建数据源。

数据源包含了访问迁移任务上游所需的信息。数据迁移任务需要引用对应的数据源来获取访问配置信息。因此，在创建数据迁移任务之前，需要先创建任务的数据源。详细的数据源管理命令请参考管理上游数据源。

10.6.6.1.1 第一步：配置数据源

1. （可选）加密数据源密码

   在 DM 的配置文件中，推荐使用经 dmctl 加密后的密文密码。按照下面的示例可以获得数据源的密文密码，用于下一步编写数据源配置文件。

```
tiup dmctl encrypt 'abc!@#123'
```

```
MKxn0Qo3m3XOyjCnhEMtsUCm83EhGQDZ/T4=
```

2. 编写数据源配置文件

   每个数据源需要一个单独的配置文件来创建数据源。按照下面示例创建 ID 为“mysql-01”的数据源，创建数据源配置文件 `./source-mysql-01.yaml`：

```
source-id: "mysql-01"     # 数据源 ID，在数据迁移任务配置和 dmctl 命令行中引用该 source-id
    '→ 可以关联到对应的数据源

from:
  host: "127.0.0.1"
  port: 3306
  user: "root"
  password: "MKxn0Qo3m3XOyjCnhEMtsUCm83EhGQDZ/T4=" # 推荐使用 dmctl
      '→ 对上游数据源的用户密码加密之后的密码
  security:                                        # 上游数据源 TLS 相关配置。
      '→ 如果没有需要则可以删除
    ssl-ca: "/path/to/ca.pem"
    ssl-cert: "/path/to/cert.pem"
    ssl-key: "/path/to/key.pem"
```

10.6.6.1.2 第二步：创建数据源

使用如下命令创建数据源：

```
tiup dmctl --master-addr <master-addr> operate-source create ./source-mysql-01.yaml
```

数据源配置文件的其他配置参考数据源配置文件介绍。

命令返回结果如下：

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

10.6.6.1.3 第三步：查询创建的数据源

创建数据源后，可以使用如下命令查看创建的数据源：

- 如果知道数据源的 source-id，可以通过 dmctl config source <source-id> 命令直接查看数据源配置：

```
tiup dmctl --master-addr <master-addr> config source mysql-01
```

```
{
  "result": true,
  "msg": "",
  "cfg": "enable-gtid: false
    flavor: mysql
    source-id: mysql-01
    from:
      host: 127.0.0.1
      port: 3306
      user: root
      password: '******'
}
```

- 如果不知道数据源的 source-id，可以先通过 dmctl operate-source show 命令查看源数据库列表，从中可以找到对应的数据源。

```
tiup dmctl --master-addr <master-addr> operate-source show
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "source is added but there is no free worker to bound",
            "source": "mysql-02",
            "worker": ""
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

10.6.6.2 管理 TiDB Data Migration 上游数据源

dmctl 是运维TiDB Data Migration (DM) 集群的命令行工具。本文介绍了如何使用 dmctl 组件来管理数据源配置，包括如何加密数据库密码，数据源操作，查看数据源配置，改变数据源与 DM-worker 的绑定关系。

10.6.6.2.1 加密数据库密码

在 DM 相关配置文件中，推荐使用经dmctl 加密后的密码。对于同一个原始密码，每次加密后密码不同。

```
./dmctl -encrypt 'abc!@#123'
```

```
MKxn0Qo3m3XOyjCnhEMtsUCm83EhGQDZ/T4=
```

10.6.6.2.2 数据源操作

operate-source 命令向 DM 集群加载、列出、移除数据源。

```
help operate-source
```

```
`create`/`stop`/`show` upstream MySQL/MariaDB source.

Usage:
  dmctl operate-source <operate-type> [config-file ...] [--print-sample-config] [flags]


Flags:
  -h, --help                  help for operate-source
```

```
  -p, --print-sample-config    print sample config file of source

Global Flags:
  -s, --source strings    MySQL Source ID
```

参数解释

- create：创建一个或多个上游的数据库源。创建多个数据源失败时，会尝试回滚到执行命令之前的状态
- stop：停止一个或多个上游的数据库源。停止多个数据源失败时，可能有部分数据源已成功停止
- show：显示已添加的数据源以及对应的 DM-worker
- config-file：
  - 指定 source.yaml 的文件路径
  - 可传递多个文件路径
- --print-sample-config：打印示例配置文件。该参数会忽视其余参数

命令用法示例

使用 operate-source 命令创建数据源配置：

```
operate-source create ./source.yaml
```

其中 source.yaml 的配置参考上游数据库配置文件介绍。

结果如下：

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

10.6.6.2.3 查看数据源配置

> 注意：
>
> config 命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 get-config 命令。

如果知道 source-id，可以通过 `dmctl --master-addr <master-addr> config source <source-id>` 命令直接查看数据源配置。

```
config source mysql-replica-01
```

```
{
  "result": true,
  "msg": "",
  "cfg": "enable-gtid: false
    flavor: mysql
    source-id: mysql-replica-01
    from:
      host: 127.0.0.1
      port: 8407
      user: root
      password: '******'
}
```

如果不知道 source-id，可以先通过 `dmctl --master-addr <master-addr> operate-source show` 查看源数据库列表。

```
operate-source show
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "source is added but there is no free worker to bound",
            "source": "mysql-replica-02",
            "worker": ""
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

10.6.6.2.4 改变数据源与 DM-worker 的绑定关系

`transfer-source` 用于改变数据源与 DM-worker 的绑定关系。

```
help transfer-source
```

```
Transfers a upstream MySQL/MariaDB source to a free worker.

Usage:
  dmctl transfer-source <source-id> <worker-id> [flags]

Flags:
  -h, --help   help for transfer-source

Global Flags:
  -s, --source strings   MySQL Source ID.
```

在改变绑定关系前，DM 会检查待解绑的 worker 是否正在运行同步任务，如果正在运行则需要先暂停任务，并在改变绑定关系后恢复任务。

命令用法示例

如果不清楚 DM-worker 的绑定关系，可以通过 `dmctl --master-addr <master-addr> list-member --worker` 查看。

```
list-member --worker
```

```
{
    "result": true,
    "msg": "",
    "members": [
        {
            "worker": {
                "msg": "",
                "workers": [
                    {
                        "name": "dm-worker-1",
                        "addr": "127.0.0.1:8262",
                        "stage": "bound",
                        "source": "mysql-replica-01"
                    },
                    {
                        "name": "dm-worker-2",
                        "addr": "127.0.0.1:8263",
                        "stage": "free",
                        "source": ""
                    }
                ]
            }
        }
```

```
    ]
}
```

在本示例中 mysql-replica-01 绑定到了 dm-worker-1 上。使用如下命令可以将该数据源绑定到 dm-worker-2 上

```
transfer-source mysql-replica-01 dm-worker-2
```

```
{
    "result": true,
    "msg": ""
}
```

再次通过 dmctl --master-addr <master-addr> list-member --worker 查看，检查命令已生效。

```
list-member --worker
```

```
{
    "result": true,
    "msg": "",
    "members": [
        {
            "worker": {
                "msg": "",
                "workers": [
                    {
                        "name": "dm-worker-1",
                        "addr": "127.0.0.1:8262",
                        "stage": "free",
                        "source": ""
                    },
                    {
                        "name": "dm-worker-2",
                        "addr": "127.0.0.1:8263",
                        "stage": "bound",
                        "source": "mysql-replica-01"
                    }
                ]
            }
        }
    ]
}
```

10.6.6.3 TiDB Data Migration 数据迁移任务配置向导

本文档介绍如何配置TiDB Data Migration (DM) 的数据迁移任务。

10.6.6.3.1 配置需要迁移的数据源

配置需要迁移的数据源之前，首先应该确认已经在 DM 创建相应数据源：

- 查看数据源可以参考查看数据源配置
- 创建数据源可以参考在 DM 创建数据源
- 数据源配置可以参考数据源配置文件介绍

仿照下面的 `mysql-instances:` 示例定义数据迁移任务需要同步的单个或者多个数据源。

```
---

##### ********* 任务信息配置 *********
name: test              # 任务名称，需要全局唯一

##### ******** 数据源配置 **********
mysql-instances:
  - source-id: "mysql-replica-01" # 从 source-id = mysql-replica-01 的数据源迁移数据
  - source-id: "mysql-replica-02" # 从 source-id = mysql-replica-02 的数据源迁移数据
```

10.6.6.3.2 配置迁移的目标TiDB 集群

仿照下面的 `target-database:` 示例定义迁移的目标TiDB 集群。

```
---

##### ********* 任务信息配置 *********
name: test              # 任务名称，需要全局唯一

##### ******** 数据源配置 **********
mysql-instances:
  - source-id: "mysql-replica-01" # 从 source-id = mysql-replica-01 的数据源迁移数据
  - source-id: "mysql-replica-02" # 从 source-id = mysql-replica-02 的数据源迁移数据

##### ******** 目标 TiDB 配置 **********

target-database:        # 目标 TiDB 配置
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: ""          # 如果密码不为空，则推荐使用经过 dmctl 加密的密文
```

10.6.6.3.3 配置需要迁移的表

如果不需要过滤或迁移特定表，可以跳过该项配置。

配置从数据源迁移表的黑白名单，则需要添加两个定义，详细配置规则参考Block & Allow Lists：

1. 定义全局的黑白名单规则

```
block-allow-list:
  bw-rule-1:                            # 规则名称
    do-dbs: ["test.*", "user"]          # 迁移哪些库，支持通配符 "*" 和 "?"，do-dbs 和 ignore
        '→ -dbs 只需要配置一个，如果两者同时配置只有 do-dbs 会生效
    # ignore-dbs: ["mysql", "account"] # 忽略哪些库，支持通配符 "*" 和 "?"
    do-tables:                          # 迁移哪些表，do-tables 和 ignore-tables
        '→ 只需要配置一个，如果两者同时配置只有 do-tables 会生效
    - db-name: "test.*"
      tbl-name: "t.*"
    - db-name: "user"
      tbl-name: "information"
  bw-rule-2:                            # 规则名称
    ignore-tables:                      # 忽略哪些表
    - db-name: "user"
      tbl-name: "log"
```

2. 在数据源配置中引用黑白名单规则，过滤该数据源需要迁移的表

```
mysql-instances:
     - source-id: "mysql-replica-01" # 从 source-id = mysql-replica-01 的数据源迁移数据
       block-allow-list:  "bw-rule-1" # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
           '→ 则使用 black-white-list
     - source-id: "mysql-replica-02" # 从 source-id = mysql-replica-02 的数据源迁移数据
       block-allow-list:  "bw-rule-2" # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
           '→ 则使用 black-white-list
```

10.6.6.3.4 配置需要过滤的操作

如果不需要过滤特定库或者特定表的特定操作，可以跳过该项配置。

配置过滤特定操作，则需要添加两个定义，详细配置规则参考Binlog Event Filter：

1. 定义全局的数据源操作过滤规则

```
filters:                                    # 定义过滤数据源特定操作的规则，
    '→ 可以定义多个规则
  filter-rule-1:                            # 规则名称
    schema-pattern: "test_*"                # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                    # 匹配数据源的表名，支持通配符 "*" 和 "?"
    events: ["truncate table", "drop table"]    # 匹配上 schema-pattern 和 table-pattern
        '→ 的库或者表的操作类型
    action: Ignore                          # 迁移（Do）还是忽略(Ignore)
  filter-rule-2:
    schema-pattern: "test"
    events: ["all dml"]
```

```
    action: Do
```

2. 在数据源配置中引用数据源操作过滤规则，过滤该数据源的指定库或表的指定操作

```
mysql-instances:
  - source-id: "mysql-replica-01"     # 从 source-id = mysql-replica-01 的数据源迁移数据
    block-allow-list:  "bw-rule-1"    # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
        ↪ 则使用 black-white-list
    filter-rules: ["filter-rule-1"]  # 过滤数据源特定操作的规则，可以配置多个过滤规则
  - source-id: "mysql-replica-02"     # 从 source-id = mysql-replica-02 的数据源迁移数据
    block-allow-list:  "bw-rule-2"    # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
        ↪ 则使用 black-white-list
    filter-rules: ["filter-rule-2"]  # 过滤数据源特定操作的规则，可以配置多个过滤规则
```

10.6.6.3.5 配置需要数据源表到目标TiDB 表的映射

如果不需要将数据源表路由到不同名的目标TiDB 表，可以跳过该项配置。分库分表合并迁移的场景必须配置该规则。

配置数据源表迁移到目标TiDB 表的路由规则，则需要添加两个定义，详细配置规则参考Table Routing：

1. 定义全局的路由规则

```
routes:                             # 定义数据源表迁移到目标 TiDB 表的路由规则，
    ↪ 可以定义多个规则
  route-rule-1:                     # 规则名称
    schema-pattern: "test_*"        # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"            # 匹配数据源的表名，支持通配符 "*" 和 "?"
    target-schema: "test"           # 目标 TiDB 库名
    target-table: "t"               # 目标 TiDB 表名
  route-rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

2. 在数据源配置中引用路由规则，过滤该数据源需要迁移的表

```
mysql-instances:
  - source-id: "mysql-replica-01"                       # 从 source-id = mysql-replica-01
      ↪ 的数据源迁移数据
    block-allow-list:  "bw-rule-1"                      # 黑白名单配置名称，如果 DM 版本早于
        ↪ v2.0.0-beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-1"]                     # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
    route-rules: ["route-rule-1", "route-rule-2"]       # 数据源表迁移到目标 TiDB 表的路由规则
        ↪ ，可以定义多个规则
  - source-id: "mysql-replica-02"                       # 从 source-id = mysql-replica-02
      ↪ 的数据源迁移数据
```

```
    block-allow-list:  "bw-rule-2"                     # 黑白名单配置名称，如果 DM 版本早于
        ↪ v2.0.0-beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-2"]                    # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
```

10.6.6.3.6 配置是否进行分库分表合并

如果是分库分表合并的数据迁移场景，并且需要同步分库分表的 DDL，则必须显式配置 `shard-mode`，否则不要配置该选项。

分库分表 DDL 同步问题特别多，请确认了解 DM 同步分库分表 DDL 的原理和限制后，谨慎使用。

```
---

##### ********* 任务信息配置 *********
name: test                          # 任务名称，需要全局唯一
shard-mode: "pessimistic"           # 默认值为 "" 即无需协调。如果为分库分表合并任务，
    ↪ 请设置为悲观协调模式 "pessimistic"。在深入了解乐观协调模式的原理和使用限制后，
    ↪ 也可以设置为乐观协调模式 "optimistic"
```

10.6.6.3.7 其他配置

下面是本数据迁移任务配置向导的完整示例。完整的任务配置参见DM 任务完整配置文件介绍。

```
---

##### ********* 任务信息配置 *********
name: test                          # 任务名称，需要全局唯一
shard-mode: "pessimistic"           # 默认值为 "" 即无需协调。如果为分库分表合并任务，
    ↪ 请设置为悲观协调模式 "pessimistic"。在深入了解乐观协调模式的原理和使用限制后，
    ↪ 也可以设置为乐观协调模式 "optimistic"
task-mode: all                      # 任务模式，可设为 "full" - "只进行全量数据迁移"、 "incremental" -
    ↪  "Binlog 实时同步"、 "all" - "全量 + Binlog 迁移"
#### timezone: "UTC"                  # 指定数据迁移任务时 SQL Session 使用的时区。 DM
    ↪ 默认使用目标库的全局时区配置进行数据迁移，并且自动确保同步数据的正确性。
    ↪ 使用自定义时区依然可以确保整个流程的正确性，但一般不需要手动指定。

##### ******** 数据源配置 **********
mysql-instances:
  - source-id: "mysql-replica-01"                  # 从 source-id = mysql-replica-01
      ↪ 的数据源迁移数据
    block-allow-list:  "bw-rule-1"                 # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-
        ↪ beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-1"]                # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
```

```
    route-rules: ["route-rule-1", "route-rule-2"]    # 数据源表迁移到目标 TiDB 表的路由规则，
        ↪ 可以定义多个规则
  - source-id: "mysql-replica-02"                    # 从 source-id = mysql-replica-02
      ↪ 的数据源迁移数据
    block-allow-list:  "bw-rule-2"                   # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-
        ↪ beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-2"]                  # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
    route-rules: ["route-rule-2"]                    # 数据源表迁移到目标 TiDB 表的路由规则，
        ↪ 可以定义多个规则

##### ******** 目标 TiDB 配置 **********
target-database:       # 目标 TiDB 配置
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: ""         # 如果密码不为空，则推荐使用经过 dmctl 加密的密文

##### ******** 功能配置 **********
block-allow-list:                       # 定义数据源迁移表的过滤规则，可以定义多个规则。如果 DM
    ↪ 版本早于 v2.0.0-beta.2 则使用 black-white-list
  bw-rule-1:                            # 规则名称
    do-dbs: ["test.*", "user"]          # 迁移哪些库，支持通配符 "*" 和 "?"，do-dbs 和 ignore-dbs
        ↪ 只需要配置一个，如果两者同时配置只有 do-dbs 会生效
    # ignore-dbs: ["mysql", "account"] # 忽略哪些库，支持通配符 "*" 和 "?"
    do-tables:                          # 迁移哪些表，do-tables 和 ignore-tables 只需要配置一个，
        ↪ 如果两者同时配置只有 do-tables 会生效
    - db-name: "test.*"
      tbl-name: "t.*"
    - db-name: "user"
      tbl-name: "information"
  bw-rule-2:                           # 规则名称
    ignore-tables:                     # 忽略哪些表
    - db-name: "user"
      tbl-name: "log"

filters:                                        # 定义过滤数据源特定操作的规则，可以定义多个规则
  filter-rule-1:                                # 规则名称
    schema-pattern: "test_*"                    # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                        # 匹配数据源的表名，支持通配符 "*" 和 "?"
    events: ["truncate table", "drop table"]    # 匹配上 schema-pattern 和 table-pattern
        ↪ 的库或者表的操作类型
    action: Ignore                              # 迁移（Do）还是忽略(Ignore)
  filter-rule-2:
    schema-pattern: "test"
```

```
    events: ["all dml"]
    action: Do

routes:                                 # 定义数据源表迁移到目标 TiDB 表的路由规则，可以定义多个规则
  route-rule-1:                         # 规则名称
    schema-pattern: "test_*"            # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                # 匹配数据源的表名，支持通配符 "*" 和 "?"
    target-schema: "test"               # 目标 TiDB 库名
    target-table: "t"                   # 目标 TiDB 表名
  route-rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

10.6.6.4 TiDB Data Migration 分库分表合并

TiDB Data Migration (DM) 支持将上游 MySQL/MariaDB 各分库分表中的 DML、 DDL 数据合并后迁移到下游TiDB 的库表中。

如果你需要从小数据量分库分表 MySQL 合并迁移数据到TiDB，请参考这篇文档

10.6.6.4.1 使用限制

目前分库分表合并功能仅支持有限的场景，使用该功能前，请仔细阅读悲观模式分库分表合并迁移使用限制和乐观模式分库分表合并迁移使用限制。

10.6.6.4.2 参数配置

在任务配置文件中设置：

```
shard-mode: "pessimistic" # 默认值为 "" 即无需协调。如果为分库分表合并任务，请设置为悲观协调模式
    ↪ "pessimistic"。在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "
    ↪ optimistic"
```

10.6.6.4.3 手动处理 Sharding DDL Lock

如果分库分表合并迁移过程中发生了异常，对于部分场景，可尝试参考手动处理 Sharding DDL Lock 进行处理。

10.6.6.5 TiDB Data Migration 表路由

使用 TiDB Data Migration (DM) 迁移数据时，你可以配置表路由(Table Routing) 规则，指定将上游 MySQL/MariaDB 实例的特定表迁移到下游的特定表。

> **注意：**

- 不支持对同一个表设置多个不同的路由规则。
- Schema 的匹配规则需要单独设置，用来迁移 `CREATE/DROP SCHEMA ...`，例如下面配置表路由中的rule-2。

10.6.6.5.1 配置表路由

在迁移任务配置文件中，添加如下配置：

```
routes:
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    target-schema: "test"
    target-table: "t"
    # extract-table、extract-schema 和 extract-source 为可选配置，仅在需要提取分表、
        ↪ 分库和数据源信息时填写
    extract-table:
      table-regexp: "t_(.*)"
      target-column: "c_table"
    extract-schema:
      schema-regexp: "test_(.*)"
      target-column: "c_schema"
    extract-source:
      source-regexp: "(.*)"
      target-column: "c_source"
  rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

在简单任务场景下，推荐使用通配符匹配库表名，但需注意以下版本差异：

- 对于 v1.0.5 版及后续版本，表路由支持通配符匹配。但注意所有版本中通配符匹配中的* 符号只能有一个，且必须在末尾。
- 对于 v1.0.5 以前的版本，表路由支持通配符，但不支持 `[...]` 与 `[!...]` 表达式。

10.6.6.5.2 参数解释

- 对于匹配上schema-pattern/table-pattern 规则的上游 MySQL/MariaDB 实例的表，DM 将它们迁移到下游的 `target-schema/target-table`。
- 对于匹配上 `schema-pattern/table-pattern` 规则的分表，DM 通过 `extract-table.table-regexp` 提取分表信息，通过 `extract-schema.schema-regexp` 提取分库信息，通过 `extract-source.source-regexp` 提取数据来源信息，然后写入到下游合表对应的 `target-column` 中。

10.6.6.5.3 使用示例

下面展示了四个不同场景下的配置示例。如果你需要从小数据量分库分表 MySQL 合并迁移数据到TiDB，请参考这篇文档

分库分表合并

假设存在分库分表场景，需要将上游两个 MySQL 实例的表 test_{1,2,3...}.t_{1,2,3...} 迁移到下游TiDB 的一张表 test.t。

为了迁移到下游实例的表 test.t，需要创建以下表路由规则：

- rule-1 用来迁移匹配上 schema-pattern: "test_*" 和 table-pattern: "t_*" 的表的 DML/DDL 语句到下游的 test.t。
- rule-2 用来迁移匹配上 schema-pattern: "test_*" 的库的 DDL 语句，例如 CREATE/DROP SCHEMA xx。

> 注意：
>
> - 如果下游TiDB schema: test 已经存在，并且不会被删除，则可以省略 rule-2。
> - 如果下游TiDB schema: test 不存在，只设置了rule_1，则迁移会报错schema test doesn
>   ↪ 't exist。

```
rule-1:
  schema-pattern: "test_*"
  table-pattern: "t_*"
  target-schema: "test"
  target-table: "t"
rule-2:
  schema-pattern: "test_*"
  target-schema: "test"
```

提取分库分表数据源信息写入合表

假设存在分库分表场景，需要将上游两个 MySQL 实例的表 test_{11,12,13...}.t_{1,2,3...} 迁移到下游TiDB 的一张表 test.t，同时需要提取分库分表的源信息写入下游合表中，用于标识合表中各行数据的来源。

为了迁移到下游实例的表 test.t，需要创建和分库分表合并场景类似的表路由规则，并在其中增加 extract-
↪ table、extract-schema、extract-source 配置用于提取分库分表源数据信息：

- extract-table：对于匹配上 schema-pattern 和 table-pattern 的分表，DM 根据 table-regexp 提取分表，并将去除 t_ 后的后缀信息写入合表的 target-column，即 c_table 列中。
- extract-schema：对于匹配上 schema-pattern和 table-pattern 的分库，DM 根据 schema-regexp 提取分库，并将去除 test_ 后的后缀信息写入合表的 target-column，即 c_schema 列中。
- extract-source：对于匹配上 schema-pattern 和 table-pattern 的分表，DM 将其数据源信息写入合表的 target-column，即 c_source 列中。

```
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    target-schema: "test"
    target-table: "t"
    extract-table:
      table-regexp: "t_(.*)"
      target-column: "c_table"
    extract-schema:
      schema-regexp: "test_(.*)"
      target-column: "c_schema"
    extract-source:
      source-regexp: "(.*)"
      target-column: "c_source"
  rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

为了提取上游分表来源信息数据以写入到下游合表，必须在启动迁移任务前手动在下游创建好对应合表，合表需要包含用于存放分表源数据信息的三个扩展列 `target-column`（表名列、库名列、数据源列），扩展列必须为表末尾列且必须为字符串类型。

```
CREATE TABLE `test`.`t`
    ( a int(11) PRIMARY
    KEY,
    c_table varchar(10) DEFAULT NULL,
    c_schema varchar(10) DEFAULT NULL,
    c_source varchar(10) DEFAULT NULL
);
```

例如，上游源数据为：

数据源 `mysql-01`:

```
mysql> select * from test_11.t_1;
+---+
| a |
+---+
| 1 |
+---+
mysql> select * from test_11.t_2;
+---+
| a |
+---+
| 2 |
+---+
mysql> select * from test_12.t_1;
```

```
+---+
| a |
+---+
| 3 |
+---+
```

数据源 mysql-02:

```
mysql> select * from test_13.t_3;
+---+
| a |
+---+
| 4 |
+---+
```

则 DM 同步后合表中的数据将为：

```
mysql> select * from test.t;
+---+---------+----------+----------------- +
| a | c_table | c_schema | c_source |
+---+---------+----------+----------------- +
| 1 | 1       | 11       | mysql-01 |
| 2 | 2       | 11       | mysql-01 |
| 3 | 1       | 12       | mysql-01 |
| 4 | 3       | 13       | mysql-02 |
+---+---------+----------+----------------- +
```

错误的合表建表示例：

> 注意：
> 以下错误都可能导致分库分表数据源信息写入合表失败。

- c-table 列不在末尾

```
CREATE TABLE `test`.`t` (
    c_table varchar(10) DEFAULT NULL,
    a int(11) PRIMARY KEY,
    c_schema varchar(10) DEFAULT NULL,
    c_source varchar(10) DEFAULT NULL
);
```

- c-source 列缺失

```
CREATE TABLE `test`.`t`
    ( a int(11) PRIMARY
    KEY,
    c_table varchar(10) DEFAULT NULL,
    c_schema varchar(10) DEFAULT NULL,
);
```

- c_schema 列为非 string 类型

```
CREATE TABLE `test`.`t`
    ( a int(11) PRIMARY
    KEY,
    c_table varchar(10) DEFAULT NULL,
    c_schema int(11) DEFAULT NULL,
    c_source varchar(10) DEFAULT NULL,
);
```

分库合并

假设存在分库场景， 将上游两个 MySQL 实例 `test_{1,2,3...}.t_{1,2,3...}` 迁移到下游 TiDB 的 `test.t_{1,2,3...}`，创建一条路由规则即可：

```
  rule-1:
    schema-pattern: "test_*"
    target-schema: "test"
```

错误的表路由

假设存在下面两个路由规则，`test_1_bak.t_1_bak` 可以匹配上 `rule-1` 和 `rule-2`，违反 table 路由的限制而报错。

```
  rule-0:
    schema-pattern: "test_*"
    target-schema: "test"
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    target-schema: "test"
    target-table: "t"
  rule-2:
    schema-pattern: "test_1_bak"
    table-pattern: "t_1_bak"
    target-schema: "test"
    target-table: "t_bak"
```

10.6.6.6 TiDB Data Migration 黑白名单过滤

使用 TiDB Data Migration (DM) 迁移数据时，你可以配置上游数据库实例表的黑白名单过滤(Block & Allow List) 规则，用来过滤或者只迁移某些 `database/table` 的所有操作。

10.6.6.6.1 配置黑白名单

在迁移任务配置文件中，添加如下配置：

```
block-allow-list:             # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  rule-1:
    do-dbs: ["test*"]         # 非 ~ 字符开头，表示规则是通配符；v1.0.5 及后续版本支持通配符规则
        ↪ 。
    do-tables:
    - db-name: "test[123]"    # 匹配 test1、test2、test3。
      tbl-name: "t[1-5]"      # 匹配 t1、t2、t3、t4、t5。
    - db-name: "test"
      tbl-name: "t"
  rule-2:
    do-dbs: ["~^test.*"]      # 以 ~ 字符开头，表示规则是正则表达式。
    ignore-dbs: ["mysql"]
    do-tables:
    - db-name: "~^test.*"
      tbl-name: "~^t.*"
    - db-name: "test"
      tbl-name: "t"
    ignore-tables:
    - db-name: "test"
      tbl-name: "log"
```

在简单任务场景下，推荐使用通配符匹配库表名，但需注意以下版本差异：

- 对于 v1.0.5 版及后续版本，黑白名单支持通配符匹配。但注意所有版本中通配符匹配中的* 符号只能有一个，且必须在末尾。
- 对于 v1.0.5 以前的版本，黑白名单仅支持正则表达式。

10.6.6.6.2 参数解释

- do-dbs：要迁移的库的白名单，类似于 MySQL 中的 replicate-do-db。
- ignore-dbs：要迁移的库的黑名单，类似于 MySQL 中的 replicate-ignore-db。
- do-tables：要迁移的表的白名单，类似于 MySQL 中的 replicate-do-table。必须同时指定 db-name 与 tbl-name。
- ignore-tables：要迁移的表的黑名单，类似于 MySQL 中的 replicate-ignore-table。必须同时指定 db-name 与 tbl-name。

以上参数值以~ 开头时均支持使用正则表达式来匹配库名、表名。

10.6.6.6.3 过滤规则

- do-dbs 与 ignore-dbs 对应的过滤规则与 MySQL 中的 Evaluation of Database-Level Replication and Binary Logging Options 类似。

- do-tables 与 ignore-tables 对应的过滤规则与 MySQL 中的 Evaluation of Table-Level Replication Options 类似。

> 注意：
>
> DM 中黑白名单过滤规则与 MySQL 中相应规则存在以下区别：
>
> - MySQL 中存在 replicate-wild-do-table 与 replicate-wild-ignore-table 用于支持通配符，DM 中各配置参数直接支持以~ 字符开头的正则表达式。
> - DM 当前只支持 ROW 格式的binlog，不支持 STATEMENT/MIXED 格式的binlog，因此应与 MySQL 中 ROW 格式下的规则对应。
> - 对于DDL，MySQL 仅依据默认的database 名称（USE 语句显式指定的database）进行判断，而 DM 优先依据DDL 中的database 名称部分进行判断，并当DDL 中不包含database 名称时再依据 USE 部分进行判断。假设需要判断的 SQL 为 USE test_db_2; CREATE TABLE test_db_1.
> ↪ test_table (c1 INT PRIMARY KEY)，且MySQL 配置了replicate-do-db=test_db_1、DM 配置了 do-dbs: ["test_db_1"]，则对于 MySQL 该规则不会生效，而对于 DM 该规则会生效。

判断table test.t 是否应该被过滤的流程如下：

1. 首先进行 schema 过滤判断
    - 如果 do-dbs 不为空，判断 do-dbs 中是否存在一个匹配的 schema。
        - 如果存在，则进入table 过滤判断。
        - 如果不存在，则过滤 test.t。
    - 如果 do-dbs 为空并且 ignore-dbs 不为空，判断 ignore-dbs 中是否存在一个匹配的 schema。
        - 如果存在，则过滤 test.t。
        - 如果不存在，则进入table 过滤判断。
    - 如果 do-dbs 和 ignore-dbs 都为空，则进入table 过滤判断。
2. 进行table 过滤判断
    1. 如果 do-tables 不为空，判断 do-tables 中是否存在一个匹配的table。
        - 如果存在，则迁移 test.t。
        - 如果不存在，则过滤 test.t。
    2. 如果 ignore-tables 不为空，判断 ignore-tables 中是否存在一个匹配的table。
        - 如果存在，则过滤 test.t.
        - 如果不存在，则迁移 test.t。
    3. 如果 do-tables 和 ignore-tables 都为空，则迁移 test.t。

> 注意：
>
> 如果是判断 schema test 是否应该被过滤，则只进行 schema 过滤判断。

10.6.6.6.4 使用示例

假设上游 MySQL 实例包含以下表：

```
`logs`.`messages_2016`
`logs`.`messages_2017`
`logs`.`messages_2018`
`forum`.`users`
`forum`.`messages`
`forum_backup_2016`.`messages`
`forum_backup_2017`.`messages`
`forum_backup_2018`.`messages`
```

配置如下：

```
block-allow-list: # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  bw-rule:
    do-dbs: ["forum_backup_2018", "forum"]
    ignore-dbs: ["~^forum_backup_"]
    do-tables:
    - db-name: "logs"
      tbl-name: "~_2018$"
    - db-name: "~^forum.*"
       tbl-name: "messages"
    ignore-tables:
    - db-name: "~.*"
       tbl-name: "^messages.*"
```

应用 bw-rule 规则后：

table	是否过滤	过滤的原因
logs ′→ .messages_2016 ′→	是	schema logs 没有匹配到 do-dbs 任意一项
logs ′→ .messages_2017 ′→	是	schema logs 没有匹配到 do-dbs 任意一项
logs ′→ .messages_2018 ′→	是	schema logs 没有匹配到 do-dbs 任意一项
forum_backup_2016 ′→ .messages ′→	是	schema forum_backup_2016 没有匹配到 do-dbs 任意一项

table	是否过滤	过滤的原因
forum_backup_2017 ′→ .messages ′→	是	schema forum_backup_2017 没有匹配到 do-dbs 任意一项
forum ′→ .users ′→	是	1. schema forum 匹配到 do-dbs，进入 table 过滤判断 2. schema 和 table 没有匹配到 do-tables 和 ignore-tables 中任意一项，并且 do-tables 不为空，因此过滤
forum ′→ .messages ′→	否	1. schema forum 匹配到 do-dbs，进入 table 过滤判断 2. schema 和 table 匹配到 do-tables 的 db-name: "~^forum ′→ .*",tbl-name: ′→ "messages"
forum_backup_2018 ′→ .messages ′→	否	1. schema forum_backup_2018 匹配到 do-dbs，进入 table 过滤判断 2. schema 和 table 匹配到 do-tables 的 db-name: "~^forum ′→ .*",tbl-name: ′→ "messages"

10.6.6.7 TiDB Data Migration Binlog 事件过滤

TiDB Data Migration (DM) 的 Binlog 事件过滤(Binlog event filter) 是比迁移表黑白名单更加细粒度的过滤规则，可以指定只迁移或者过滤掉某些 schema / table 的指定类型binlog，比如 INSERT、 TRUNCATE TABLE。

10.6.6.8 配置 Binlog 事件过滤

在迁移任务配置文件中，添加如下配置：

```
filters:
  rule-1:
    schema-pattern: "test_*"
```

```
    table-pattern: "t_*"
    events: ["truncate table", "drop table"]
  sql-pattern: ["^DROP\\s+PROCEDURE", "^CREATE\\s+PROCEDURE"]
    action: Ignore
```

从 DM v2.0.2 开始，你也可以在上游数据库配置文件中配置 Binlog 事件过滤。见上游数据库配置文件介绍。

在简单任务场景下，推荐使用通配符匹配库表名，但需注意以下版本差异：

- 对于 v1.0.5 版及后续版本，Binlog 事件过滤支持通配符匹配。但注意所有版本中通配符匹配中的* 符号只能有一个，且必须在末尾。
- 对于 v1.0.5 以前的版本，Binlog Event Filter 支持通配符，但不支持 `[...]` 与 `[!...]` 表达式。

10.6.6.9 参数解释

- `schema-pattern/table-pattern`：对匹配上的上游 MySQL/MariaDB 实例的表的binlog events 或者 DDL SQL 语句通过以下规则进行过滤。
- `events`：binlog events 数组，仅支持从以下 `Event` 中选择一项或多项。

Event	分类	解释
all		代表包含下面所有的 events
all dml		代表包含下面所有 DML events
all ddl		代表包含下面所有 DDL events
none		代表不包含下面所有 events
none ddl		代表不包含下面所有 DDL events
none dml		代表不包含下面所有 DML events
insert	DML	insert DML event
update	DML	update DML event
delete	DML	delete DML event
create database	DDL	create database event
drop database	DDL	drop database event
create table	DDL	create table event
create index	DDL	create index event
drop table	DDL	drop table event
truncate table	DDL	truncate table event
rename table	DDL	rename table event
drop index	DDL	drop index event
alter table	DDL	alter table event

- `sql-pattern`：用于过滤指定的 DDL SQL 语句，支持正则表达式匹配，例如上面示例中的 `"^DROP\\s+PROCEDURE"`。
- `action`：string (`Do` / `Ignore`)；进行下面规则判断，满足其中之一则过滤，否则不过滤。
  - `Do`：白名单。binlog event 如果满足下面两个条件之一就会被过滤掉：

    * 不在该rule的 events 中。
    * 如果规则的 sql-pattern 不为空的话，对应的 SQL 没有匹配上 sql-pattern 中任意一项。
  - Ignore：黑名单。如果满足下面两个条件之一就会被过滤掉：
    * 在该rule的 events 中。
    * 如果规则的 sql-pattern 不为空的话，对应的 SQL 可以匹配上 sql-pattern 中任意一项。
  - 同一个表匹配上多个规则时，将会顺序应用这些规则，并且黑名单的优先级高于白名单，即如果同时存在规则 Ignore 和 Do 应用在某个表上，那么 Ignore 生效。

10.6.6.10 使用示例

10.6.6.10.1 过滤分库分表的所有删除操作

需要设置下面两个 Binlog event filter rule 来过滤掉所有的删除操作：

- filter-table-rule 过滤掉所有匹配到pattern test_*.t_* 的table 的 turncate table、drop table、delete statement 操作。
- filter-schema-rule 过滤掉所有匹配到pattern test_* 的 schema 的 drop database 操作。

```
filters:
  filter-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["truncate table", "drop table", "delete"]
    action: Ignore
  filter-schema-rule:
    schema-pattern: "test_*"
    events: ["drop database"]
    action: Ignore
```

10.6.6.10.2 只迁移分库分表的 DML 操作

需要设置下面两个 Binlog event filter rule 只迁移 DML 操作：

- do-table-rule 只迁移所有匹配到pattern test_*.t_* 的table 的 create table、insert、update、delete 操作。
- do-schema-rule 只迁移所有匹配到pattern test_* 的 schema 的 create database 操作。

> 注意：
>
> 迁移 create database/table 的原因是创建库和表后才能迁移 DML。

```
filters:
  do-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["create table", "all dml"]
    action: Do
  do-schema-rule:
    schema-pattern: "test_*"
    events: ["create database"]
    action: Do
```

10.6.6.10.3 过滤 TiDB 不支持的 SQL 语句

可设置如下规则过滤TiDB 不支持的 PROCEDURE 语句：

```
filters:
  filter-procedure-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    sql-pattern: ["^DROP\\s+PROCEDURE", "^CREATE\\s+PROCEDURE"]
    action: Ignore
```

10.6.6.10.4 过滤 TiDB parser 不支持的 SQL 语句

对于 TiDB parser 不支持的 SQL 语句，DM 无法解析获得 schema/table 信息， 因此需要使用全局过滤规则：schema-pattern: "*"。

> **注意：**
> 全局过滤规则的设置必须尽可能严格，以避免过滤掉需要迁移的数据。

可设置如下规则过滤某些版本的TiDB parser 不支持的 PARTITION 语句：

```
filters:
  filter-partition-rule:
    schema-pattern: "*"
    sql-pattern: ["ALTER\\s+TABLE[\\s\\S]*ADD\\s+PARTITION", "ALTER\\s+TABLE[\\s\\S]*DROP\\s+
        ↪ PARTITION"]
    action: Ignore
```

10.6.6.11 通过 SQL 表达式过滤 DML

在进行增量数据迁移时，可以通过如何过滤binlog 事件功能过滤某些类型的binlog event，例如不向下游迁移 DELETE 事件以达到归档、审计等目的。但是binlog event filter 无法以更细粒度判断某一行的 DELETE 事件是否要被过滤。

为了解决上述问题，从 v2.0.5 起，DM 支持在增量数据同步阶段使用binlog value filter过滤迁移数据。DM 支持的 ROW 格式的binlog 中，binlog event 带有所有列的值。你可以基于这些值配置 SQL 表达式。如果该表达式对于某条行变更的计算结果是 TRUE，DM 就不会向下游迁移该条行变更。

具体操作步骤和实现细节，请参考如何通过 SQL 表达式过滤 DML。

10.6.6.12 TiDB Data Migration 对 online DDL 工具的支持

在 MySQL 生态中，gh-ost 与pt-osc 等工具较广泛地被使用，TiDB Data Migration (DM) 对其提供了特殊的支持以避免对不必要的中间数据进行迁移。本文介绍了在 DM 中使用常见 online DDL 工具的方法和注意事项。

有关 DM 对 online DDL 工具支持的原理、处理流程等，可参考online-ddl。

10.6.6.12.1 使用限制

- DM 仅针对 gh-ost 与 pt-osc 做了特殊支持。
- 在开启 online-ddl 时，增量复制对应的checkpoint 应不处于 online DDL 执行过程中。如上游某次 online DDL 操作开始于binlog position-A、结束于 position-B，则增量复制的起始点应早于 position-A 或晚于 position-B，否则可能出现迁移出错，具体可参考FAQ。

10.6.6.12.2 参数配置

在 v2.0.5 及之后的版本，请在 task 配置文件中使用 online-ddl 配置项。

如上游 MySQL/MariaDB（同时）使用 gh-ost 或 pt-osc 工具，则在task 的配置文件中设置：

```
online-ddl: true
```

> **注意：**
>
> 自 v2.0.5 起，online-ddl-scheme 已被弃用，请使用 online-ddl 代替 online-ddl-scheme。如设置 online-ddl: true 会覆盖掉 online-ddl-scheme。如设置 online-ddl-scheme: "pt" 或 online-ddl-scheme: "gh-ost" 会被转换为 online-ddl: true。

在 v2.0.5 之前的版本（不含 v2.0.5），请在 task 配置文件中使用 online-ddl-scheme 配置项。

如上游 MySQL/MariaDB 使用的是 gh-ost 工具，则在task 的配置文件中设置：

```
online-ddl-scheme: "gh-ost"
```

如上游 MySQL/MariaDB 使用的是pt-osc 工具，则在task 的配置文件中设置：

```
online-ddl-scheme: "pt"
```

10.6.6.13 迁移任务操作

10.6.6.13.1 TiDB Data Migration 任务前置检查

本文介绍了TiDB Data Migration (DM) 的任务前置检查功能。此功能用于提前检测出上游 MySQL 实例配置中可能存在的一些错误。

使用场景

为了使数据迁移任务顺利进行，DM 在启动迁移任务时会自动触发任务前置检查，并返回检查结果。只有当前置检查通过后，DM 才开始执行迁移任务。

如果你想要手动触发前置检查，运行 `check-task` 命令即可。

例如：

```
tiup dmctl check-task ./task.yaml
```

检查项说明

当任务前置检查被触发时，DM 会按照配置的迁移模式对相应的检查项进行检查。

本小节列出了前置检查中的所有检查项。

> 注意：
>
> 针对前置检查中必须通过的检查项，本文在检查项名称前标注了（必须）。
>
> - 对于标注了（必须）的检查项，如果检查没有通过，DM 会在检查结束后返回错误并拒绝执行该迁移任务。此时，请依据错误信息修改配置，在满足检查项要求后重试。
> - 对于未标注（必须）的检查项，如果检查没有通过，DM 会在检查结束后返回警告。如果检查结果中只有警告没有错误，DM 将自动开始执行该迁移任务。

通用检查项

对于任何一种迁移模式，前置检查都会包含以下通用检查项：

- 数据库版本
  - MySQL 版本 > 5.5
  - MariaDB 版本 >= 10.1.2

> 警告：
>
> - 使用 DM 从 MySQL v8.0 迁移数据到TiDB 目前为实验特性（从 DM v2.0 引入），不建议在生产环境下使用。
> - 使用 DM 从 MariaDB 迁移数据到TiDB 目前为实验特性，不建议在生产环境下使用。

- 上游 MySQL 表结构的兼容性
  - 检查上游表是否设置了外键。TiDB 不支持外键，如果上游表设置了外键，则返回警告。
  - 检查上游字符集是否与TiDB 兼容，详见TiDB 支持的字符集。
  - 检查上游表中是否存在主键或唯一键约束（从 v1.0.7 版本引入）。

> 警告：
>
> - 上游使用不兼容的字符集时，下游可以使用utf8mb4 字符集建表兼容同步，但不建议这样做。建议调整上游的字符集，使用下游支持的字符集。
> - 上游表不存在主键或唯一键约束时，可能出现单行数据在下游被重复同步多次的情况，同步性能也会降低，不建议在生产环境下使用。

全量数据迁移检查项

对于全量数据迁移模式（`task-mode: full`），除了通用检查项，前置检查还会包含以下检查项：

- （必须）上游数据库的dump 权限
  - 检查是否有INFORMATION_SCHEMA 和 dump 表的 SELECT 权限。
  - 如果consistency=flush，将检查是否有RELOAD 权限。
  - 如果consistency=flush/lock，将检查是否有dump 表的 LOCK TABLES 权限。
- （必须）上游 MySQL 多实例分库分表的一致性
  - 悲观协调模式下，检查所有分表的表结构是否一致，检查内容包括：
    * Column 数量
    * Column 名称
    * Column 顺序
    * Column 类型
    * 主键
    * 唯一索引
  - 乐观协调模式下，检查所有分表结构是否满足乐观协调兼容。
  - 如果曾经通过 `start-task` 命令成功启动任务，那么将不会对一致性进行检查。
- 分表中自增主键检查
  - 分表存在自增主键时返回警告。如果存在自增主键冲突，请参照自增主键冲突处理解决。

Physical Import 检查项

在任务配置中使用 `import-mode: "physical"` 后，会增加如下的前置检查项以保证Physical Import 正常运行。如果参照提示后仍然难以完成这些前置检查，你可以尝试使用Logical Import 进行导入。

- 下游数据库中的空Region

  - 如果空Region 的数量大于 max(1000，表的数量 * 3)，即大于 “1000” 和 “3 倍表数量” 二者中的较大者，前置检查会向用户返回警告。可以调整 PD 相关参数加快空Region 的合并速度，并等待空Region 数量下降以解除警告。参见PD 调度策略最佳实践- Region Merge 速度慢

- 下游数据库中的Region 分布
  - 统计不同的TiKV 上的Region 数目。假设Region 数最少的TiKV 节点上拥有a 个 Region，Region 数最多的 TiKV 节点上拥有b 个 Region，如果a/b 小于 0.75，则前置检查会向用户返回警告。可以调整 PD 相关参数加快Region 调度速度，并等待Region 数目变化以解除警告。参见PD 调度策略最佳实践-Leader/Region 分布不均衡

- 下游数据库TiDB、 PD、 TiKV 组件的版本
  - Physical Import 需要调用TiDB、 PD、 TiKV 接口，如果版本不符合要求，会返回错误。

- 下游数据库的剩余空间
  - 估算上游数据库所有白名单表的大小之和 (source_size)， 如果下游数据库剩余空间小于 source_size，前置检查会返回错误；如果下游数据库剩余空间小于 TiKV 副本数 * source_size * 2，前置检查会返回警告。

- 下游数据库是否在运行与 Physical Import 不兼容的任务
  - 目前 Physical Import 不兼容TiCDC、 PITR 任务，如果发现下游数据库正在运行这些任务，前置检查会返回错误。

增量数据迁移检查项

对于增量数据迁移模式（task-mode: incremental），除了通用检查项，前置检查还会包含以下检查项：

- （必须）上游数据库的REPLICATION 权限
  - 检查是否有REPLICATION CLIENT 权限。
  - 检查是否有REPLICATION SLAVE 权限。

- 数据库主从配置
  - 建议上游数据库设置数据库ID server_id（非 AWS Aurora 环境建议开启GTID），防止主从复制切换出错。

- （必须）MySQL binlog 配置
  - 检查binlog 是否开启（DM 要求binlog 必须开启）。
  - 检查是否有 binlog_format=ROW（DM 只支持ROW 格式的binlog 迁移）。
  - 检查是否有 binlog_row_image=FULL（DM 只支持 binlog_row_image=FULL）。
  - 如果配置了 binlog_do_db 或者 binlog_ignore_db， 那么检查需要迁移的库表， 是否满足 binlog_do_db 和 binlog_ignore_db 的条件。

- （必须）检查上游是否处于Online-DDL 过程中，即创建了 ghost 表，但还未执行 rename 的阶段。如果处于 online-DDL 中，则检查报错，请等待 DDL 结束后重试。

全量加增量数据迁移检查项

对于全量加增量数据迁移模式（`task-mode: all`），除了通用检查项，前置检查还会包含全量数据迁移检查项，以及增量数据迁移检查项。

可忽略检查项

一般情况下前置检查项用于提前发现环境中可能存在的风险，不建议忽略。如果你的数据迁移任务面临特殊场景，你可以通过`ignore-checking-items`配置项来跳过部分检查项。

值	含义
dump_privilege	检查上游 MySQL 实例用户的 dump 相关权限
replication_privilege	检查上游 MySQL 实例用户的 replication 相关权限
version	检查上游数据库版本
server_id	检查上游数据库是否设置 server_id
binlog_enable	检查上游数据库是否已启用binlog
table_schema	检查上游 MySQL 表结构的兼容性
schema_of_shard_tables	检查上游 MySQL 多实例分库分表的表结构一致性
auto_increment_ID	检查上游 MySQL 多实例分库分表的自增主键冲突
online_ddl	检查上游是否处于Online-DDL 过程中
empty_region	Physical Import 检查空 Region 的数目
region_distribution	Physical Import 检查 Region 的分布
downstream_version	检查下游数据库 TiDB、 PD、 TiKV 的版本
free_space	检查下游数据库的剩余空间
downstream_mutex_features	检查下游数据库是否存在与 Physical Import 不兼容的任务

> 注意：
>
> 6.0 之前的版本支持忽略更多的检查项，但诸如 `binlog_row_image` 等参数，若配置错误可能导致同步时丢失数据，因此在 6.0 版本中移除了部分与数据安全相关的检查项。

配置检查参数

任务前置检查支持多线程并行。即使分表数目达到万级别，检查也可以在分钟级完成。

你可以通过数据迁移任务配置文件里 `mydumpers` 字段中的 `threads` 参数指定线程的数量。

```
mydumpers:                          # dump 处理单元的运行配置参数
  global:                           # 配置名称
    threads: 4                      # dump
        ↪ 处理单元从上游数据库实例导出数据和执行前置检查时访问上游的线程数量，默认值为 4
    chunk-filesize: 64              # dump 处理单元生成的数据文件大小，默认值为 64，单位为 MB
    extra-args: "--consistency none" # dump 处理单元的其他参数，不需要在 extra-args 中配置 table-
        ↪ list，DM 会自动生成
```

> **注意：**
>
> `threads` 参数的值决定了上游和 DM 之间的物理连接数。过大的 `threads` 可能会加大上游的负载，请注意控制 `threads` 大小。

10.6.6.13.2 创建 TiDB Data Migration 数据迁移任务

`start-task` 命令用于创建数据迁移任务。当数据迁移任务启动时，TiDB Data Migration (DM) 将自动对相应权限和配置进行前置检查。

```
help start-task
```

```
Starts a task as defined in the configuration file

Usage:
  dmctl start-task [-s source ...] [--remove-meta] <config-file> [flags]

Flags:
  -h, --help                help for start-task
      --remove-meta         whether to remove task's meta data
      --start-time string   specify the start time of binlog replication, e.g. '2021-10-21
          '→ 00:01:00' or 2021-10-21T00:01:00

Global Flags:
      --config string         Path to config file.
      --master-addr string    Master API server address, this parameter is required when
          '→ interacting with the dm-master

      --rpc-timeout string    RPC timeout, default is 10m. (default "10m")
  -s, --source strings        MySQL Source ID.
      --ssl-ca string         Path of file that contains list of trusted SSL CAs for connection.
      --ssl-cert string       Path of file that contains X509 certificate in PEM format for
          '→ connection.
      --ssl-key string        Path of file that contains X509 key in PEM format for connection.
  -V, --version               Prints version and exit.
```

命令用法示例

```
start-task [ -s "mysql-replica-01"] ./task.yaml
```

参数解释

- **-s：**
  - 可选

  - 指定在特定的一个 MySQL 源上执行 `task.yaml`
  - 如果设置，则只启动指定任务在该 MySQL 源上的子任务
- `config-file`：
  - 必选
  - 指定 `task.yaml` 的文件路径
- `remove-meta`:
  - 可选
  - 如果设置，则在启动指定任务时会移除该任务之前存在的metadata
- `start-time`:
  - 可选，格式为 `'2021-10-21 00:01:00'` 或 `2021-10-21T00:01:00`
  - 对于增量任务，可以通过该参数大致指定任务起始位点，该参数比任务配置文件中的binlog 位置优先级更高，也比下游checkpoint 中的binlog 位置优先级更高
  - 当该任务存在checkpoint 时，如果通过这种方式启动任务，DM 会自动开启 safe mode 直到同步过checkpoint，以避免重置任务到更早位置时遇到数据重复的报错。向前重置起始位点时，如果起始位点的表结构与下游当前表结构不一致可能会在同步时报错；向后重置起始位点时，需要注意跳过的binlog 可能在下游残留脏数据
  - 指定了过早的时间时，会从最早的binlog 开始同步
  - 指定了过晚的时间时， 会报错 `start-time {input-time} is too late, no binlog location matches it`

返回结果示例

```
start-task task.yaml
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

10.6.6.13.3 TiDB Data Migration 查询任务状态

本文介绍TiDB Data Migration (DM) `query-status` 命令的查询结果、任务状态与子任务状态。

查询结果

```
» query-status
```

```
{
    "result": true,     # 查询是否成功
    "msg": "",          # 查询失败原因描述
    "tasks": [          # 迁移 task 列表
        {
            "taskName": "test",          # 任务名称
            "taskStatus": "Running",     # 任务运行状态
            "sources": [                 # 该任务的上游 MySQL 列表
                "mysql-replica-01",
                "mysql-replica-02"
            ]
        },
        {
            "taskName": "test2",
            "taskStatus": "Paused",
            "sources": [
                "mysql-replica-01",
                "mysql-replica-02"
            ]
        }
    ]
}
```

关于 tasks 下的 taskStatus 状态的详细定义，请参阅任务状态。

推荐的 query-status 使用方法是：

1. 首先使用 query-status 查看各个 task 的运行状态是否正常。
2. 如果发现其中某一 task 状态有问题，通过 query-status <出错任务的 taskName> 来得到更详细的错误信息。

任务状态

DM 的迁移任务状态取决于其分配到 DM-worker 上的子任务状态，定义见下表：

任务对应的所有子任务的状态	任务状态
任一子任务处于“Paused”状态且返回结果有错误信息	Error - Some error oc-curred in subtask
任一处于 Sync 阶段的子任务处于“Run-ning”状态但其 Relay 处理单元未运行 (处于 Error/-Paused/Stopped 状态)	Error - Relay status is Error/-Paused/Stopped

任务对应的所有子任务的状态	任务状态
任一子任务处于“Paused”状态且返回结果没有错误信息	Paused
所有子任务处于“New”状态	New
所有子任务处于“Finished”状态	Finished
所有子任务处于“Stopped”状态	Stopped
其他情况	Running

详情查询结果

```
» query-status test
```

```
{
    "result": true,     # 查询是否成功
    "msg": "",          # 查询失败原因描述
    "sources": [        # 上游 MySQL 列表
        {
            "result": true,
            "msg": "",
            "sourceStatus": {                   # 上游 MySQL 的信息
                "source": "mysql-replica-01",
                "worker": "worker1", "result":
                null, "relayStatus": null
            },
            "subTaskStatus": [              # 上游 MySQL 所有子任务的信息
                {
                    "name": "test",         # 子任务名称
                    "stage": "Running",     # 子任务运行状态，包括 “New”，“Running”，“Paused
                        '→ ”，“Stopped” 以及 “Finished”
                    "unit": "Sync",         # DM 的处理单元，包括 “Check”，“Dump “，“Load”
                        '→ 以及 “Sync”
                    "result": null,         # 子任务失败时显示错误信息
                    "unresolvedDDLLockID": "test-`test`.`t_target`",     # sharding DDL lock ID,
                        '→ 可用于异常情况下手动处理 sharding DDL lock
                    "sync": {                   # 当前 `Sync` 处理单元的迁移信息
                        "masterBinlog": "(bin.000001, 3234)",                                   #
                            '→ 上游数据库当前的 binlog position
                        "masterBinlogGtid": "c0149e17-dff1-11e8-b6a8-0242ac110004:1-14",     #
                            '→ 上游数据库当前的 GTID 信息
                        "syncerBinlog": "(bin.000001, 2525)",                                   #
                            '→ 已被 `Sync` 处理单元迁移的 binlog position
                        "syncerBinlogGtid": "",                                                 #
                            '→ 使用 GTID 迁移的 binlog position
                        "blockingDDLs": [       # 当前被阻塞的 DDL 列表。该项仅在当前 DM-worker
                            '→ 所有上游表都处于 “synced“ 状态时才有数值，
                            '→ 此时该列表包含的是待执行或待跳过的 sharding DDL 语句
                            "USE `test`; ALTER TABLE `test`.`t_target` DROP COLUMN `age`;"
                        ],
                        "unresolvedGroups": [   # 没有被解决的 sharding group 信息
                            {
                                "target": "`test`.`t_target`",                  # 待迁移的下游表
                                "DDLs": [
                                    "USE `test`; ALTER TABLE `test`.`t_target` DROP COLUMN `age
                                        '→ `;"
                                ],
```

```
                        "firstPos": "(bin|000001.000001, 3130)",        # sharding DDL
                            '→ 语句起始 binlog position
                        "synced": [                                     # `Sync`
                            '→ 处理单元已经读到该 sharding DDL 的上游分表
                            "`test`.`t2`"
                            "`test`.`t3`"
                            "`test`.`t1`"
                        ],
                        "unsynced": [                                   # `Sync`
                            '→ 处理单元未读到该 sharding DDL 的上游分表。
                            '→ 如有上游分表未完成同步，`blockingDDLs` 为空
                        ]
                    }
                ],
                "synced": false,        # 增量复制是否已追上上游。由于后台 `Sync`
                    '→ 单元并不会实时刷新保存点，当前值为 `false`
                    '→ 并不一定代表发生了迁移延迟
                "totalRows": "12",      # 该子任务中迁移的行数
                "totalRps": "1",        # 该子任务中每秒迁移的行数
                "recentRps": "1"        # 该子任务中最后一秒迁移的行数
            }
        }
    ]
},
{
    "result": true,
    "msg": "",
    "sourceStatus": {
        "source": "mysql-replica-02",
        "worker": "worker2",
        "result": null,
        "relayStatus": null
    },
    "subTaskStatus": [
        {
            "name": "test",
            "stage": "Running",
            "unit": "Load",
            "result": null,
            "unresolvedDDLLockID": "",
            "load": {                                   # `Load` 处理单元的迁移信息
                "finishedBytes": "115",                 # 已全量导入的字节数
                "totalBytes": "452",                    # 总计需要导入的字节数
                "progress": "25.44 %",                  # 全量导入进度
                "bps": "2734"                           # 全量导入速度
```

```
            }
        }
    ]
},
{
    "result": true,
    "sourceStatus": {
        "source": "mysql-replica-03",
        "worker": "worker3",
        "result": null,
        "relayStatus": null
    },
    "subTaskStatus": [
        {
            "name": "test",
            "stage": "Paused",
            "unit": "Load",
            "result": {                 # 错误示例
                "isCanceled": false,
                "errors": [
                    {
                        "Type": "ExecSQL",
                        "msg": "Error 1062: Duplicate entry '1155173304420532225' for key
                            '→ 'PRIMARY'\n/home/jenkins/workspace/build_dm/go/src/
                            '→ github.com/pingcap/tidb-enterprise-tools/loader/db.go
                            '→ :160: \n/home/jenkins/workspace/build_dm/go/src/github.
                            '→ com/pingcap/tidb-enterprise-tools/loader/db.go:105: \n/
                            '→ home/jenkins/workspace/build_dm/go/src/github.com/pingcap
                            '→ /tidb-enterprise-tools/loader/loader.go:138: file test.t1
                            '→ .sql"
                    }
                ],
                "detail": null
            },
            "unresolvedDDLLockID": "",
            "load": {
                "finishedBytes": "0",
                "totalBytes": "156",
                "progress": "0.00 %",
                "bps": "0"
            }
        }
    ]
},
{
```

```
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-04",
                "worker": "worker4",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Dump",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "dump": {
                        "totalTables": "10",          # `Dump` 处理单元的迁移信息
                        "completedTables": "3",       # 需要 dump 的表数量
                                                      # 已 dump 的表数量
                        "finishedBytes": "2542",      # 已 dump 的字节数
                        "finishedRows": "32",         # 已 dump 的行数
                        "estimateTotalRows": "563",  # 预估需要 dump 的行数
                        "progress": "30.52 %",        # dump 的进度
                        "bps": "445"                  # dump 的速度
                    }
                }
            ]
        },
    ]
}
```

关于 sources 下 subTaskStatus 中 stage 状态和状态转换关系的详细信息，请参阅子任务状态。

关于 sources 下 subTaskStatus 中 unresolvedDDLLockID的操作细节，请参阅手动处理 Sharding DDL Lock。

子任务状态

状态描述

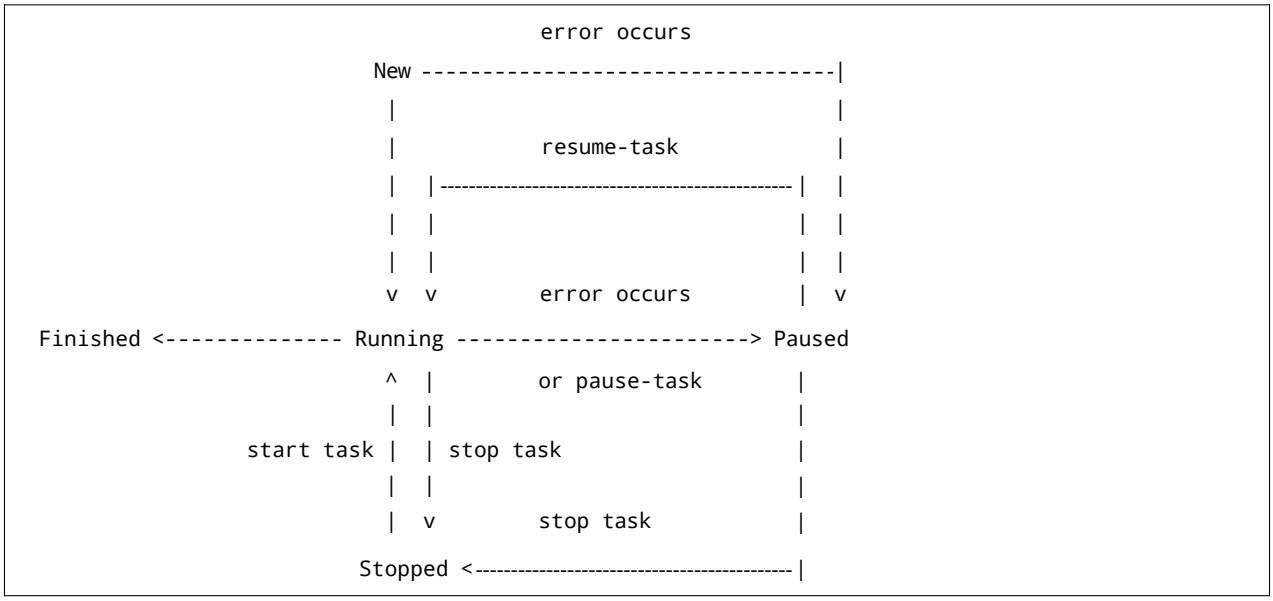
- New：
  - 初始状态。
  - 如果子任务没有发生错误，状态切换为 Running，其他情况则切换为 Paused。
- Running：正常运行状态。
- Paused：
  - 暂停状态。

- 子任务发生错误，状态切换为 Paused。
- 如在子任务为 Running 状态下执行 pause-task 命令，任务状态会切换为 Paused。
- 如子任务处于该状态，可以使用 resume-task 命令恢复任务。

- Stopped：
  - 停止状态。
  - 如在子任务为 Running 或 Paused 状态下执行 stop-task 命令，任务状态会切换为 Stopped。
  - 如子任务处于该状态，不可使用 resume-task 命令恢复任务。

- Finished：
  - 任务完成状态。
  - 只有 task-mode 为 full 的任务正常完成后，任务才会切换为该状态。

状态转换图



10.6.6.13.4 暂停 TiDB Data Migration 数据迁移任务

pause-task 命令用于暂停数据迁移任务。

> 注意：
>
> 有关 pause-task 与 stop-task 的区别如下：
>
> - 使用 pause-task 仅暂停迁移任务的执行，但仍然会在内存中保留任务的状态信息等，且可通过 query-status 进行查询；使用 stop-task 会停止迁移任务的执行，并移除内存中与该任务相关的信息，且不可再通过 query-status 进行查询，但不会移除已经写入到下游数据库中的数据以及其中的checkpoint 等 dm_meta 信息。

- 使用 pause-task 暂停迁移任务期间，由于任务本身仍然存在，因此不能再启动同名的新任务，且会阻止对该任务所需relay log 的清理；使用 stop-task 停止任务后，由于任务不再存在，因此可以再启动同名的新任务，且不会阻止对relay log 的清理。
- pause-task 一般用于临时暂停迁移任务以排查问题等；stop-task 一般用于永久删除迁移任务或通过与 start-task 配合以更新配置信息。

```
help pause-task
```

```
pause a specified running task

Usage:
 dmctl pause-task [-s source ...] <task-name | task-file> [flags]

Flags:
 -h, --help    help for pause-task

Global Flags:
 -s, --source strings    MySQL Source ID
```

命令用法示例

```
pause-task [-s "mysql-replica-01"] task-name
```

参数解释

- -s：
  - 可选
  - 指定在特定的一个 MySQL 源上暂停数据迁移任务的子任务
  - 如果设置，则只暂停该任务在指定 MySQL 源上的子任务
- task-name| task-file：
  - 必选
  - 指定任务名称或任务文件路径

返回结果示例

```
pause-task test
```

```
{
    "op": "Pause",
    "result": true,
    "msg": "",
    "sources": [
```

```
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

10.6.6.13.5 恢复 TiDB Data Migration 数据迁移任务

`resume-task` 命令用于恢复处于 Paused 状态的数据迁移任务，通常用于在人为处理完造成迁移任务暂停的故障后手动恢复迁移任务。

```
help resume-task
```

```
resume a specified paused task

Usage:
 dmctl resume-task [-s source ...] <task-name | task-file> [flags]

Flags:
 -h, --help    help for resume-task

Global Flags:
 -s, --source strings    MySQL Source ID
```

命令用法示例

```
resume-task [-s "mysql-replica-01"] task-name
```

参数解释

- `-s`:
  - 可选
  - 指定在特定的一个 MySQL 源上恢复数据迁移任务的子任务
  - 如果设置，则只恢复该任务在指定 MySQL 源上的子任务
- `task-name | task-file`:
  - 必选
  - 指定任务名称或任务文件路径

返回结果示例

```
resume-task test
```

```
{
    "op": "Resume",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

10.6.6.13.6 停止 TiDB Data Migration 数据迁移任务

stop-task 命令用于停止数据迁移任务。有关 stop-task 与 pause-task 的区别，请参考暂停数据迁移任务中的相关说明。

```
help stop-task
```

```
stop a specified task

Usage:
 dmctl stop-task [-s source ...] <task-name | task-file> [flags]

Flags:
 -h, --help   help for stop-task

Global Flags:
 -s, --source strings   MySQL Source ID
```

命令用法示例

```
stop-task [-s "mysql-replica-01"]  task-name
```

参数解释

- -s：
  - 可选
  - 指定在特定的一个 MySQL 源上停止数据迁移任务的子任务
  - 如果设置，则只停止该任务在指定 MySQL 源上的子任务
- task-name | task-file：
  - 必选

- 指定任务名称或任务文件路径

返回结果示例

```
stop-task test
```

```
{
    "op": "Stop",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

10.6.7 进阶教程

10.6.7.1 分库分表合并迁移

10.6.7.1.1 分库分表合并迁移

本文介绍了 DM 提供的分库分表的合并迁移功能，此功能可用于将上游 MySQL/MariaDB 实例中结构相同/不同的表迁移到下游TiDB 的同一个表中。DM 不仅支持迁移上游的 DML 数据，也支持协调迁移多个上游分表的 DDL 表结构变更。

简介

DM 支持对上游多个分表的数据合并迁移到TiDB 的一个表中，在迁移过程中需要协调各个分表的 DDL，以及该 DDL 前后的 DML。针对用户的使用场景，DM 支持悲观协调和乐观协调两种不同的模式。

> 注意：
>
> - 要执行分库分表合并迁移任务，必须在任务配置文件中设置 `shard-mode`。
> - DM 对分库分表的合并默认使用悲观协调模式，在文档中如果没特殊说明则默认为悲观协调模式。
> - 在没有深入了解乐观模式的原理和使用限制的情况下不建议使用该模式，否则可能造成迁移中断甚至数据不一致的严重后果。

悲观协调模式

当上游一个分表执行某一 DDL 后，这个分表的迁移会暂停，并等待其他所有分表都执行了同样的 DDL 才在下游执行该 DDL 并继续数据迁移。“悲观协调”模式的优点是可以保证迁移到下游的数据不会出错，详细的介绍请参考悲观模式下分库分表合并迁移。

乐观协调模式

在一个分表上执行的 DDL，DM 会自动修改成兼容其他分表的语句，并立即迁移到下游，不会阻挡任何分表 DML 的迁移。“乐观协调”模式的优点是处理 DDL 时不会阻塞数据的迁移，但是使用不当会有迁移中断甚至数据不一致的风险，详细的介绍请参考乐观模式下分库分表合并迁移。

悲观协调模式与乐观协调模式的对比

悲观协调模式	乐观协调模式
发起 DDL 的分表会暂停 DML 迁移	发起 DDL 的分表会继续 DML 迁移
每个分表的 DDL 执行次序和语句必须相同	每个分表只需保持表结构互相兼容即可
DDL 在整个分表群达成一致后才迁移到下游	每个分表的 DDL 会即时影响下游
错误的 DDL 操作在侦测到后可以被拦截	错误的 DDL 操作也会被迁移到下游，可能在侦测到之前已使部分上下游数据不

10.6.7.1.2 悲观模式下分库分表合并迁移

本文介绍了 DM 提供的悲观模式（默认模式）下分库分表合并迁移功能。此功能用于将上游 MySQL/MariaDB 实例中结构相同的表迁移到下游TiDB 的同一个表中。

使用限制

DM 在悲观模式下进行分表 DDL 的迁移有以下几点使用限制：

- 对于一个逻辑 sharding group（需要合并迁移到下游同一个表的所有分表组成的 group），只能使用一个仅包含这些分表所在数据源的任务进行迁移。
- 在一个逻辑 sharding group 内，所有上游分表必须以相同的顺序执行相同的 DDL 语句（库名和表名可以不同），并且只有在所有分表执行完当前一条 DDL 语句后，下一条 DDL 语句才能执行。
    - 比如，如果在table_1 表中先增加列a 后再增加列b，则在table_2 表中就不能先增加列b 后再增加列a，因为 DM 不支持以不同的顺序来执行相同的 DDL 语句。
- 在一个逻辑 sharding group 内，所有上游分表都应该执行对应的 DDL 语句。
    - 比如，若 DM-worker-2 对应的一个或多个上游分表未执行 DDL 语句，则其他已执行 DDL 语句的 DM-worker 都会暂停迁移任务，直到等到 DM-worker-2 收到上游对应的 DDL 语句。
- sharding group 数据迁移任务不支持 DROP DATABASE/TABLE 语句。
    - DM-worker 中的binlog 复制单元（sync）会自动忽略掉上游分表的 DROP DATABASE 和 DROP TABLE 语句。
- sharding group 数据迁移任务不支持 TRUNCATE TABLE 语句。
    - DM-worker 中的binlog 复制单元（sync）会自动忽略掉上游分表的 TRUNCATE TABLE 语句。

- sharding group 数据迁移任务支持 RENAME TABLE 语句，但有如下限制（online DDL 中的 RENAME 有特殊方案进行支持）：
  - 只支持 RENAME TABLE 到一个不存在的表。
  - 一条 RENAME TABLE 语句只能包含一个 RENAME 操作。
- sharding group 数据迁移任务要求一个 DDL 语句仅包含对一张表的操作。
- 增量复制任务需要确认开始迁移的binlog position 上各分表的表结构必须一致，才能确保来自不同分表的 DML 语句能够迁移到表结构确定的下游，并且后续各分表的 DDL 语句能够正确匹配与迁移。
- 如果需要变更table routing 规则，必须先等所有 sharding DDL 语句迁移完成。
  - 在 sharding DDL 语句迁移过程中，使用dmctl 尝试变更router-rules 会报错。
- 如果需要创建新表加入到一个正在执行 DDL 语句的 sharding group 中，则必须保持新表结构和最新更改的表结构一致。
  - 比如，原table_1, table_2 表初始时有(a, b) 两列，sharding DDL 语句执行后有(a, b, c) 三列，则迁移完成后新创建的表也应当有(a, b, c) 三列。
- 由于已经收到 DDL 语句的 DM-worker 会暂停任务以等待其他 DM-worker 收到对应的 DDL 语句，因此数据迁移延迟会增加。

背景

目前，DM 使用ROW 格式的binlog 进行数据迁移，且binlog 中不包含表结构信息。在ROW 格式的binlog 迁移过程中，如果不需要将多个上游表合并迁移到下游的同一个表，则只存在一个上游表的 DDL 语句会更新对应下游表结构。ROW 格式的binlog 可以认为是具有 self-description 属性。

分库分表合并迁移过程中，可以根据column values 及下游的表结构构造出相应的 DML 语句，但此时若上游的分表执行 DDL 语句进行了表结构变更，则必须对该 DDL 语句进行额外迁移处理，以避免因为表结构和binlog 数据不一致而造成迁移出错的问题。

以下是一个简化后的例子：

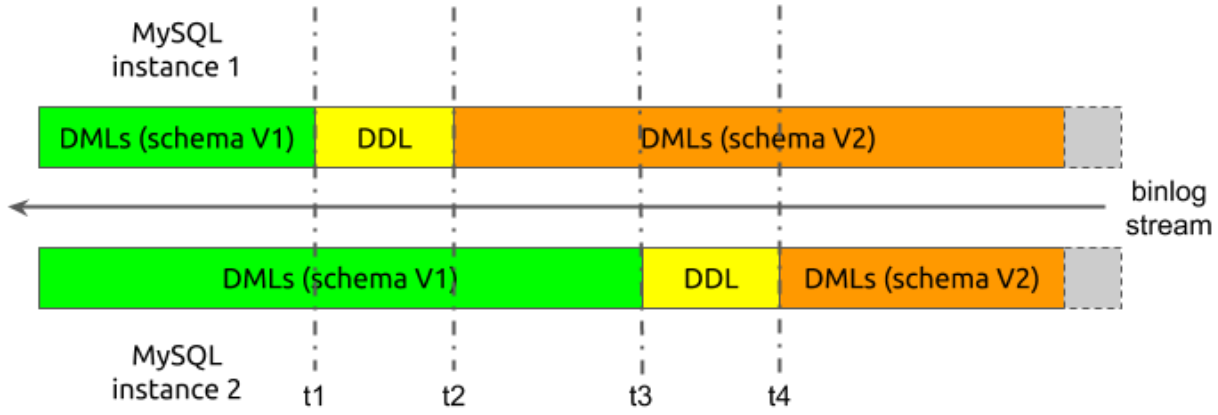


图 164: shard-ddl-example-1

在上图的例子中，分表的合库合表过程简化成了上游只有两个 MySQL 实例，每个实例内只有一个表。假设在数据迁移开始时，将两个分表的表结构版本记为 schema V1，将 DDL 语句执行完后的表结构版本记为 schema V2。

现在，假设数据迁移过程中，DM-worker 内的binlog 复制单元（sync）从两个上游分表收到的 binlog 数据有如下时序：

1. 开始迁移时，sync 从两个分表收到的都是 schema V1 版本的 DML 语句。
2. 在 t1 时刻，sync 收到实例 1 上分表的 DDL 语句。
3. 从 t2 时刻开始，sync 从实例 1 收到的是 schema V2 版本的 DML 语句；但从实例 2 收到的仍是 schema V1 版本的 DML 语句。
4. 在 t3 时刻，sync 收到实例 2 上分表的 DDL 语句。
5. 从 t4 时刻开始，sync 从实例 2 收到的也是 schema V2 版本的 DML 语句。

假设在数据迁移过程中，不对分表的 DDL 语句进行额外处理。当实例 1 的 DDL 语句迁移到下游后，下游的表结构会变更成为 schema V2 版本。但在t2 到 t3 这段时间内，sync 从实例 2 上收到的仍是 schema V1 版本的 DML 语句。当尝试把这些 schema V1 版本的 DML 语句迁移到下游时，就会由于 DML 语句与表结构的不一致而发生错误，从而无法正确迁移数据。

实现原理

基于上述例子，本部分介绍了 DM 在默认的悲观模式下合库合表过程中进行 DDL 迁移的实现原理。

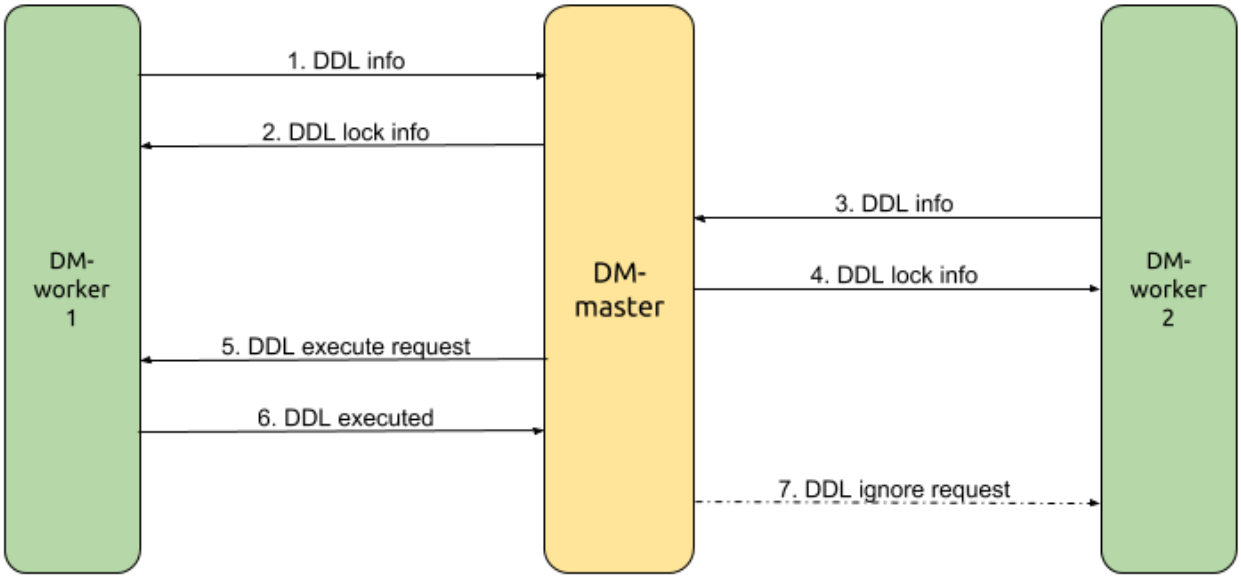


图 165: shard-ddl-flow

在这个例子中，DM-worker-1 负责迁移来自 MySQL 实例 1 的数据，DM-worker-2 负责迁移来自 MySQL 实例 2 的数据，DM-master 负责协调多个 DM-worker 间的 DDL 迁移。

从 DM-worker-1 收到 DDL 语句开始，简化后的 DDL 迁移流程为：

1. 在 t1 时刻，DM-worker-1 收到来自 MySQL 实例 1 的 DDL 语句，自身暂停该 DDL 语句对应任务的 DDL 及 DML 数据迁移，并将 DDL 相关信息发送给 DM-master。
2. DM-master 根据收到的 DDL 信息判断得知需要协调该 DDL 语句的迁移，于是为该 DDL 语句创建一个锁，并将 DDL 锁信息发回给 DM-worker-1，同时将 DM-worker-1 标记为这个锁的 owner。
3. DM-worker-2 继续进行 DML 语句的迁移，直到在t3 时刻收到来自 MySQL 实例 2 的 DDL 语句，自身暂停该 DDL 语句对应任务的数据迁移，并将 DDL 相关信息发送给 DM-master。
4. DM-master 根据收到的 DDL 信息判断得知该 DDL 语句对应的锁信息已经存在，于是直接将对应锁信息发回给 DM-worker-2。
5. 根据任务启动时的配置信息、上游 MySQL 实例分表信息、部署拓扑信息等，DM-master 判断得知自身已经收到了来自待合表的所有上游分表的 DDL 语句，于是请求 DDL 锁的 owner（DM-worker-1）向下游迁移执行该 DDL。
6. DM-worker-1 根据第二步收到的 DDL 锁信息验证 DDL 语句执行请求；向下游执行 DDL，并将执行结果反馈给 DM-master；若 DDL 语句执行成功，则自身开始继续迁移后续的（从 t2 时刻对应的binlog 开始的）DML 语句。
7. DM-master 收到来自 owner 执行 DDL 语句成功的响应，于是请求在等待该 DDL 锁的所有其他 DM-worker（DM-worker-2）忽略该 DDL 语句，直接继续迁移后续的（从 t4 时刻对应的binlog 开始的）DML 语句。

根据上面的流程，可以归纳出 DM 协调多个 DM-worker 间 sharding DDL 迁移的特点：

- 根据任务配置与 DM 集群部署拓扑信息，DM-master 内部也会建立一个逻辑 sharding group 来协调 DDL 迁移，group 中的成员为负责处理该迁移任务拆解后的各子任务的 DM-worker。
- 各 DM-worker 从 binlog event 中收到 DDL 语句后，会将 DDL 信息发送给 DM-master。
- DM-master 根据来自 DM-worker 的 DDL 信息及 sharding group 信息创建或更新 DDL 锁。
- 如果 sharding group 的所有成员都收到了某一条相同的 DDL 语句，则表明上游分表在该 DDL 执行前的 DML 语句都已经迁移完成，此时可以执行该 DDL 语句，并继续后续的 DML 迁移。
- 上游所有分表的 DDL 在经过table router 转换后需要保持一致，因此仅需 DDL 锁的 owner 执行一次该 DDL 语句即可，其他 DM-worker 可直接忽略对应的 DDL 语句。

在上面的示例中，每个 DM-worker 对应的上游 MySQL 实例中只有一个待合并的分表。但在实际场景下，一个 MySQL 实例可能有多个分库内的多个分表需要进行合并，这种情况下，sharding DDL 的协调迁移过程将更加复杂。

假设同一个 MySQL 实例中有table_1 和 table_2 两个分表需要进行合并：

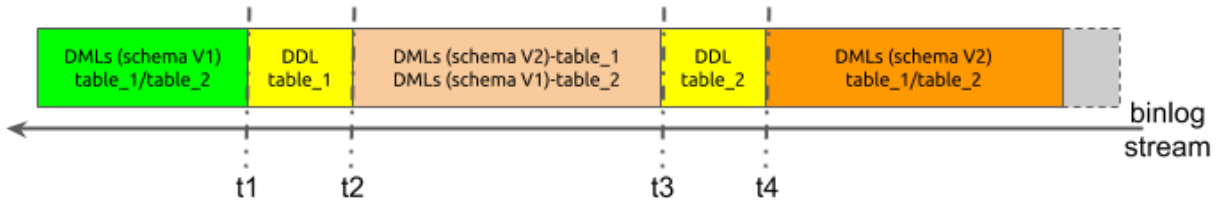


图 166: shard-ddl-example-2

在这个例子中，由于数据来自同一个 MySQL 实例，因此所有数据都是从同一个binlog 流中获得，时序如下：

1. 开始迁移时，DM-worker 内的 sync 从两个分表收到的数据都是 schema V1 版本的 DML 语句。
2. 在 t1 时刻，sync 收到table_1 分表的 DDL 语句。
3. 从 t2 到 t3 时刻，sync 收到的数据同时包含table_1 的 DML 语句（schema V2 版本）及 table_2 的 DML 语句（schema V1 版本）。
4. 在 t3 时刻，sync 收到table_2 分表的 DDL 语句。
5. 从 t4 时刻开始，sync 从两个分表收到的数据都是 schema V2 版本的 DML 语句。

假设在数据迁移过程中，不对分表的 DDL 语句进行额外处理。当table_1 的 DDL 语句迁移到下游从而变更下游表结构后，table_2 的 DML 语句（schema V1 版本）将无法正常迁移。因此，在单个 DM-worker 内部，我们也构造了与 DM-master 内类似的逻辑 sharding group，但 group 的成员是同一个上游 MySQL 实例的不同分表。

DM-worker 内协调处理 sharding group 的迁移与 DM-master 处理 DM-worker 之间的迁移不完全一致，主要原因包括：

- 当 DM-worker 收到table_1 分表的 DDL 语句时，迁移不能暂停，需要继续解析binlog 才能获得后续table_2 分表的 DDL 语句，即需要从t2 时刻继续解析直到t3 时刻。
- 在继续解析t2 到 t3 时刻的binlog 的过程中，table_1 分表的 DML 语句（schema V2 版本）不能向下游迁移；但当 sharding DDL 迁移并执行成功后，这些 DML 语句则需要迁移到下游。

DM-worker 内部 sharding DDL 迁移的简化流程为：

1. 在 t1 时刻，DM-worker 收到table_1 的 DDL 语句，并记录 DDL 信息及此时的binlog 位置点信息。
2. DM-worker 继续向前解析t2 到 t3 时刻的binlog。
3. 对于table_1 的 DML 语句（schema V2 版本），忽略；对于 table_2 的 DML 语句（schema V1 版本），正常迁移到下游。
4. 在 t3 时刻，DM-worker 收到table_2 的 DDL 语句，并记录 DDL 信息及此时的binlog 位置点信息。
5. 根据迁移任务配置信息、上游库表信息等，DM-worker 判断得知该 MySQL 实例上所有分表的 DDL 语句都已收到；于是将该 DDL 语句迁移到下游执行并变更下游表结构。
6. DM-worker 设置binlog 流的新解析起始位置点为第一步时保存的位置点。
7. DM-worker 重新开始解析从t2 到 t3 时刻的binlog。
8. 对于table_1 的 DML 语句（schema V2 版本），正常迁移到下游；对于 table_2 的 DML 语句（schema V1 版本），忽略。
9. 解析到达第四步时保存的binlog 位置点，可得知在第三步时被忽略的所有 DML 语句都已经重新迁移到下游。
10. DM-worker 继续从t4 时刻对应的binlog 位置点开始正常迁移。

综上可知，DM 在处理 sharding DDL 迁移时，主要通过两级 sharding group 来进行协调控制，简化的流程为：

1. 各 DM-worker 独立地协调对应上游 MySQL 实例内多个分表组成的 sharding group 的 DDL 迁移。
2. 当 DM-worker 收到所有分表的 DDL 语句时，向 DM-master 发送 DDL 相关信息。
3. DM-master 根据 DM-worker 发来的 DDL 信息，协调由各 DM-worker 组成的 sharing group 的 DDL 迁移。
4. 当 DM-master 收到所有 DM-worker 的 DDL 信息时，请求 DDL 锁的 owner（某个 DM-worker）执行该 DDL 语句。
5. DDL 锁的 owner 执行 DDL 语句，并将结果反馈给 DM-master；自身开始重新迁移在内部协调 DDL 迁移过程中被忽略的 DML 语句。
6. 当 DM-master 收到 owner 执行 DDL 成功的消息后，请求其他所有 DM-worker 继续开始迁移。
7. 其他所有 DM-worker 各自开始重新迁移在内部协调 DDL 迁移过程中被忽略的 DML 语句。
8. 在完成被忽略的 DML 语句的重新迁移后，所有 DM-worker 继续正常迁移。

10.6.7.1.3 乐观模式下分库分表合并迁移

本文介绍了 DM 提供的乐观模式下分库分表的合并迁移功能，此功能可用于将上游 MySQL/MariaDB 实例中结构相同/不同的表迁移到下游TiDB 的同一个表中。

> **注意：**
>
> 在没有深入了解乐观模式的原理和使用限制的情况下不建议使用该模式，否则可能造成迁移中断甚至数据不一致的严重后果。

背景

DM 支持在线上执行分库分表的 DDL 语句（通称 Sharding DDL），默认使用 “悲观模式”，即当上游一个分表执行某一 DDL 后，这个分表的迁移会暂停，等待其他所有分表都执行了同样的 DDL 才在下游执行该 DDL 并继续数据迁移。这种 “悲观协调” 模式的优点是可以保证迁移到下游的数据不会出错，缺点是会暂停数据迁移而不利于对上游进行灰度变更。有些用户可能会花较长时间在单一分表执行 DDL，验证一定时间后才会更改其他分表的结构。在悲观模式迁移的设定下，这些 DDL 会阻塞迁移，binlog 事件会大量积压。

因此，需要提供一种新的 “乐观协调” 模式，在一个分表上执行的 DDL，自动修改成兼容其他分表的语句后，立即迁移到下游，不会阻挡任何分表执行的 DML 的迁移。

乐观协调模式的配置

在任务的配置文件中指定 `shard-mode` 为 `optimistic` 则使用 “乐观协调” 模式，示例配置文件可以参考DM 任务完整配置文件介绍。

使用限制

使用 “乐观协调” 模式有一定的风险，需要严格遵照以下方针：

- 执行每个批次的 DDL 前和后，要确保每个分表的结构达成一致。

- 进行灰度 DDL 时，只集中在一个分表上测试。
- 灰度完成后，在其他分表上尽量以最简单直接的 DDL 迁移到最终的 schema，而不要重新执行灰度测试中对或错的每一步。
  - 例如：在分表执行过 `ADD COLUMN A INT; DROP COLUMN A; ADD COLUMN A FLOAT;`，在其他分表直接执行 `ADD COLUMN A FLOAT` 即可，不需要三条 DDL 都执行一遍。
- 执行 DDL 时要注意观察 DM 迁移状态。当迁移报错时，需要判断这个批次的 DDL 是否会造成数据不一致。

“乐观协调”模式下，上游执行的大部分 DDL 无需特别关注，将自动同步，本文使用“一类 DDL”指代。同时，还有一部分改变列名、列属性或列默认值的 DDL，称为“二类 DDL”，上游执行时要注意必须保证该 DDL 在各分表中按相同的顺序执行。

“二类 DDL”举例如下：

- 修改列的类型：`ALTER TABLE table_name MODIFY COLUMN column_name VARCHAR(20)`。
- 重命名列：`ALTER TABLE table_name RENAME COLUMN column_1 TO column_2;`。
- 增加没有默认值且非空的列：`ALTER TABLE table_name ADD COLUMN column_1 NOT NULL;`。
- 重命名索引：`ALTER TABLE table_name RENAME INDEX index_1 TO index_2;`。

各分表在执行以上 DDL 时，若顺序不同将导致同步中断，例如下述场景：

- 分表 1 先重命名列，再修改列类型
  1. 重命名列：`ALTER TABLE table_name RENAME COLUMN column_1 TO column_2;`。
  2. 修改列类型：`ALTER TABLE table_name MODIFY COLUMN column_3 VARCHAR(20);`。
- 分表 2 先修改列类型，再重命名列
  1. 修改列类型：`ALTER TABLE table_name MODIFY COLUMN column_3 VARCHAR(20)`。
  2. 重命名列：`ALTER TABLE table_name RENAME COLUMN column_1 TO column_2;`。

此外，不论是使用“乐观协调”或“悲观协调”，DM 仍是有以下限制：

- 不支持 `DROP TABLE/DROP DATABASE`。
- 不支持 `TRUNCATE TABLE`。
- 单条 DDL 语句要求仅包含对一张表的操作。
- TiDB 不支持的 DDL 语句在 DM 也不支持。
- 新增列的默认值不能包含 `current_timestamp`、`rand()`、`uuid()` 等，否则会造成上下游数据不一致。

风险

使用乐观模式迁移时，由于 DDL 会即时迁移到下游，若使用不当，可能导致上下游数据不一致。

使数据不一致的操作

- 各分表的表结构不兼容，例：
  - 两个分表各自添加相同名称的列，但其类型不同。
  - 两个分表各自添加相同名称的列，但其默认值不同。

  - 两个分表各自添加相同名称的生成列，但其生成表达式不同。
  - 两个分表各自添加相同名称的索引，但其键组合不同。
  - 其他同名异构的情况。
- 在分表上执行对数据具有破坏性的 DDL，然后尝试回滚，例：
  - 删除一列X，之后又把X加回来。

例子

例如以下三个分表合并迁移到TiDB：

tbl00

ID	Name
1	Sarah
5	Sophia

tbl01

ID	Name
12	Paul
16	Jessica
19	Shaun

tbl02

ID	Name
23	Bob
24	Ben

tbl

ID	Name
1	Sarah
5	Sophia
12	Paul
16	Jessica
19	Shaun
23	Bob
24	Ben

图 167: optimistic-ddl-fail-example-1

在 tbl01 新增一列 Age，默认值定为 0：

```
ALTER TABLE `tbl01` ADD COLUMN `Age` INT DEFAULT 0;
```

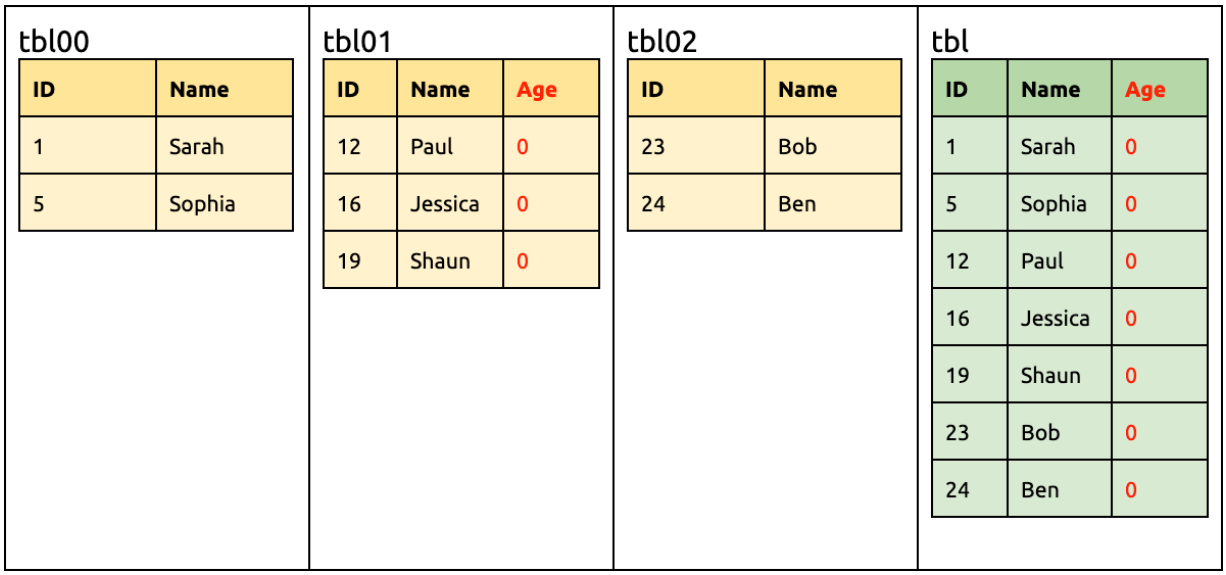


图 168: optimistic-ddl-fail-example-2

在 tbl00 新增一列 Age，但默认值定为 -1：

```
ALTER TABLE `tbl00` ADD COLUMN `Age` INT DEFAULT -1;
```

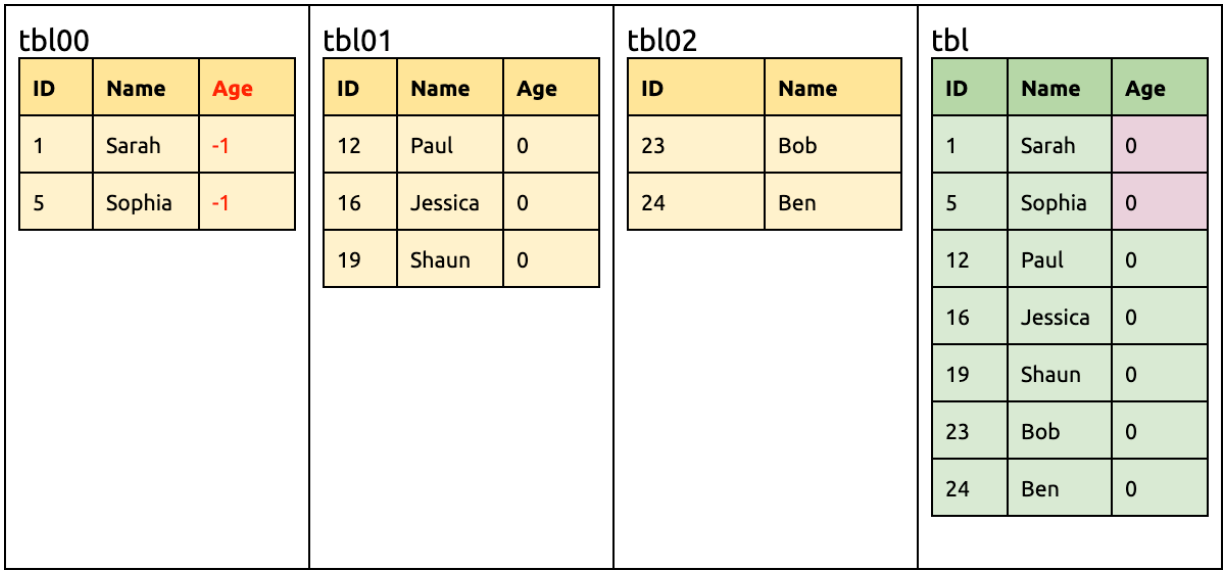


图 169: optimistic-ddl-fail-example-3

此时所有来自 tbl00 的 Age 都不一致了。这是由于 DEFAULT 0 和 DEFAULT -1 互不兼容。虽然 DM 遇到这种情况会报错，但上下游不一致的问题就需要手动去解决。

原理

在“乐观协调” 模式下，DM-worker 接收到来自上游的DDL 后，会把更新后的表结构转送给DM-master。DM-worker 会追踪各分表当前的表结构，DM-master 合并成可兼容来自每个分表 DML 的合成结构，然后把与此对应的 DDL 迁移到下游；对于 DML 会直接迁移到下游。

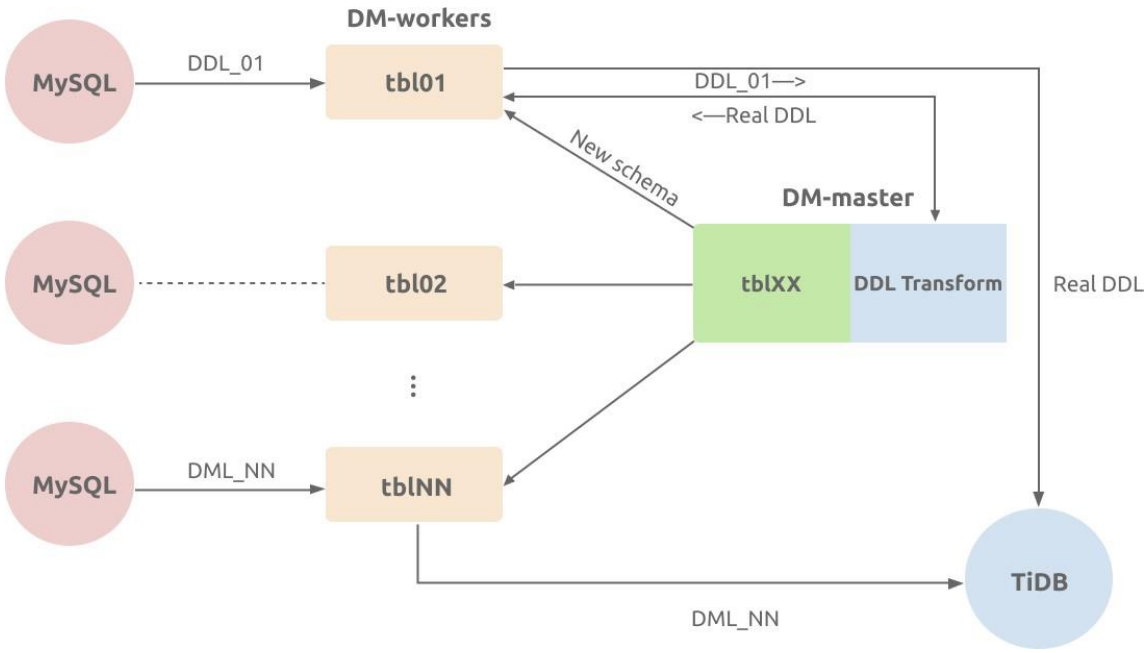


图 170: optimistic-ddl-flow

例子

例如上游 MySQL 有三个分表（`tbl00`, `tbl01` 以及 `tbl02`），使用 DM 迁移到下游TiDB 的 `tbl` 表中，如下图所示：

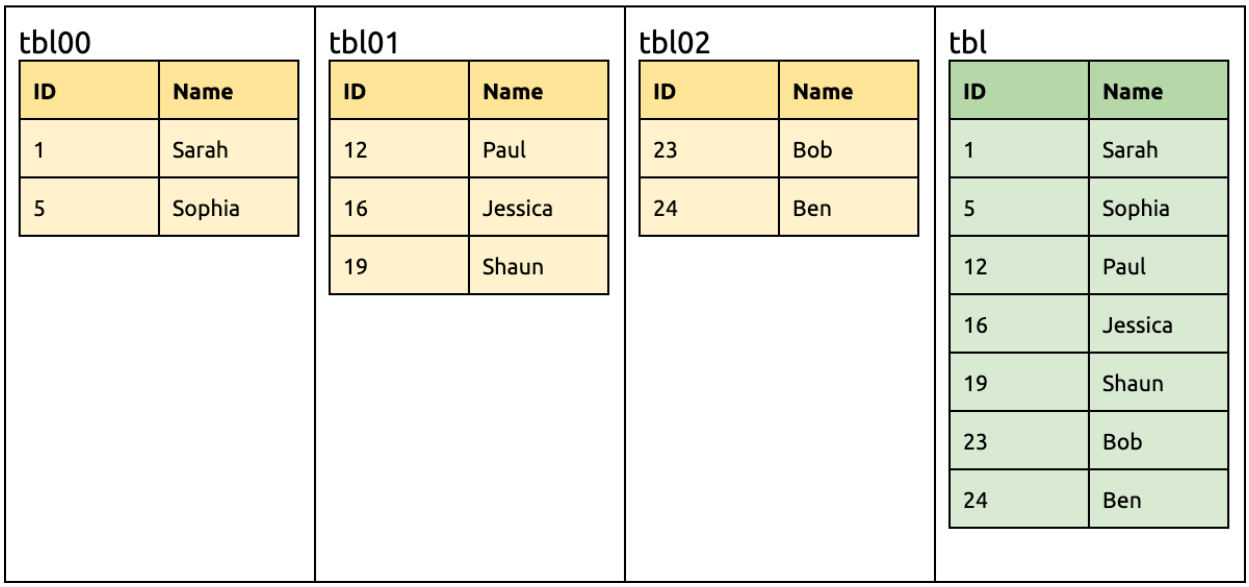


图 171: optimistic-ddl-example-1

在上游增加一列 `Level`：

```
ALTER TABLE `tbl00` ADD COLUMN `Level` INT;
```

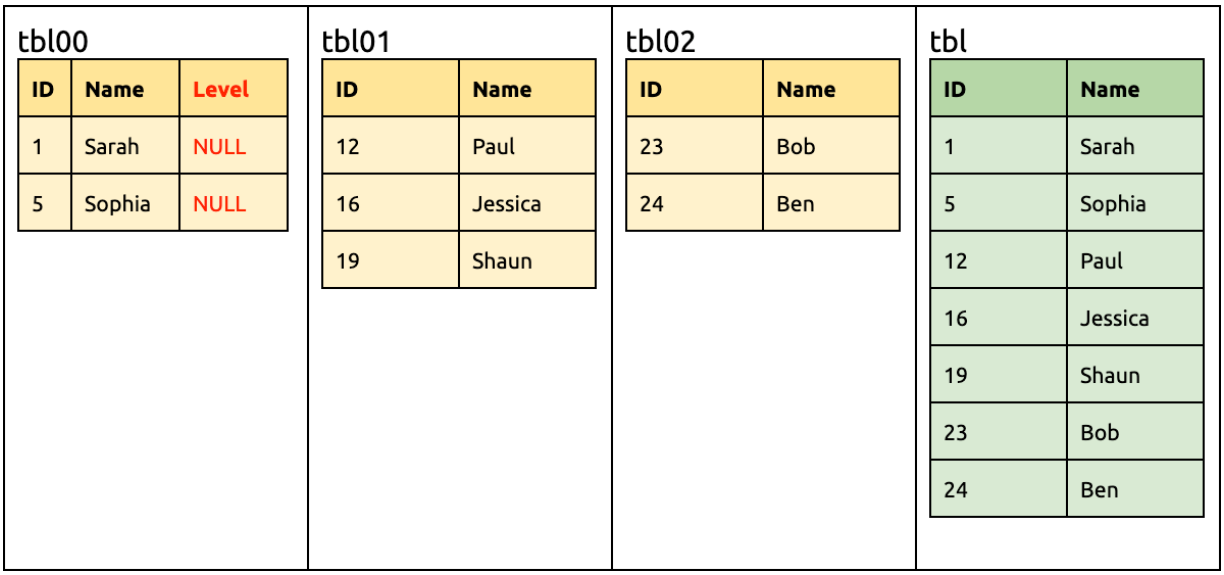


图 172: optimistic-ddl-example-2

此时下游TiDB 要准备接受来自 `tbl00` 有 `Level` 的 DML、以及来自 `tbl01` 和 `tbl02` 没有 `Level` 的 DML。

tbl00

ID	Name	Level
1	Sarah	NULL
5	Sophia	NULL

tbl01

ID	Name
12	Paul
16	Jessica
19	Shaun

tbl02

ID	Name
23	Bob
24	Ben

tbl

ID	Name	Level
1	Sarah	NULL
5	Sophia	NULL
12	Paul	NULL
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL

图 173: optimistic-ddl-example-3

这时候如下的 DML 无需修改就可以迁移到下游：

```
UPDATE `tbl00` SET `Level` = 9 WHERE `ID` = 1;
INSERT INTO `tbl02` (`ID`, `Name`) VALUES (27, 'Tony');
```

tbl00

ID	Name	Level
1	Sarah	9
5	Sophia	NULL

tbl01

ID	Name
12	Paul
16	Jessica
19	Shaun

tbl02

ID	Name
23	Bob
24	Ben
27	Tony

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	NULL
12	Paul	NULL
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 174: optimistic-ddl-example-4

在 tbl01 同样增加一列 Level：

```
ALTER TABLE `tbl01` ADD COLUMN `Level` INT;
```

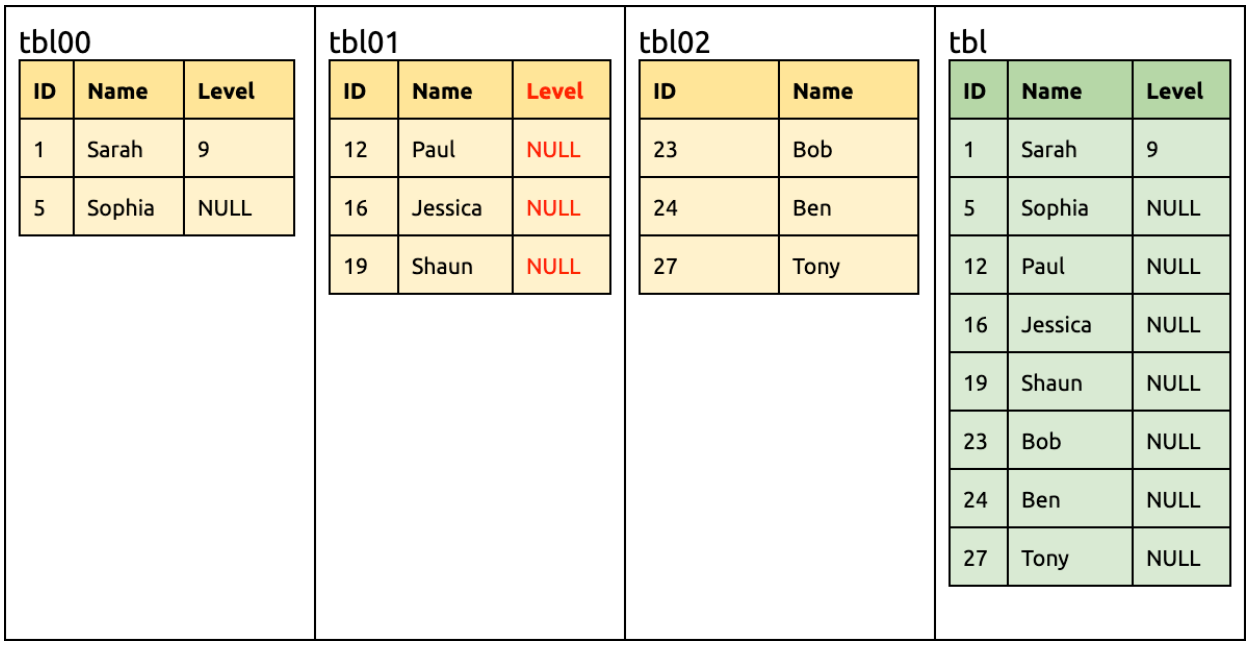


图 175: optimistic-ddl-example-5

此时下游已经有相同的 Level 列了，所以 DM-master 比较表结构之后不做任何操作。

在 tbl01 删除一列 Name：

```
ALTER TABLE `tbl01` DROP COLUMN `Name`;
```

tbl00

ID	Name	Level
1	Sarah	9
5	Sophia	NULL

tbl01

ID	Level
12	NULL
16	NULL
19	NULL

tbl02

ID	Name
23	Bob
24	Ben
27	Tony

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	NULL
12	Paul	NULL
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 176: optimistic-ddl-example-6

此时下游仍需要接收来自 `tbl00` 和 `tbl02` 含 `Name` 的 DML 语句，因此不会立刻删除该列。

同样，各种 DML 仍可直接迁移到下游：

```
INSERT INTO `tbl01` (`ID`, `Level`) VALUES (15, 7);
UPDATE `tbl00` SET `Level` = 5 WHERE `ID` = 5;
```

tbl00

ID	Name	Level
1	Sarah	9
5	Sophia	5

tbl01

ID	Level
12	NULL
15	7
16	NULL
19	NULL

tbl02

ID	Name
23	Bob
24	Ben
27	Tony

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	5
12	Paul	NULL
15	NULL	7
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 177: optimistic-ddl-example-7

在 `tbl02` 增加一列 `Level`：

```
ALTER TABLE `tbl02` ADD COLUMN `Level` INT;
```

tbl00

ID	Name	Level
1	Sarah	9
5	Sophia	5

tbl01

ID	Level
12	NULL
15	7
16	NULL
19	NULL

tbl02

ID	Name	Level
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	5
12	Paul	NULL
15	NULL	7
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 178: optimistic-ddl-example-8

此时所有分表都已有 `Level` 列。

在 `tbl00` 和 `tbl02` 各删除一列 `Name`：

```
ALTER TABLE `tbl00` DROP COLUMN `Name`;
ALTER TABLE `tbl02` DROP COLUMN `Name`;
```

tbl00

ID	Level
1	9
5	5

tbl01

ID	Level
12	NULL
15	7
16	NULL
19	NULL

tbl02

ID	Level
23	NULL
24	NULL
27	NULL

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	5
12	Paul	NULL
15	NULL	7
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 179: optimistic-ddl-example-9

到此步 Name 列也从所有分表消失了，所以可以安全从下游移除：

```
ALTER TABLE `tbl` DROP COLUMN `Name`;
```

tbl00

ID	Level
1	9
5	5

tbl01

ID	Level
12	NULL
15	7
16	NULL
19	NULL

tbl02

ID	Level
23	NULL
24	NULL
27	NULL

tbl

ID	Level
1	9
5	5
12	NULL
15	7
16	NULL
19	NULL
23	NULL
24	NULL
27	NULL

图 180: optimistic-ddl-example-10

10.6.7.1.4 手动处理 Sharding DDL Lock

DM (Data Migration) 使用 sharding DDL lock 来确保分库分表的 DDL 操作可以正确执行。绝大多数情况下，该锁定机制可自动完成；但在部分异常情况发生时，需要使用 `shard-ddl-lock` 手动处理异常的 DDL lock。

> 注意：
>
> - 本文档只适用于悲观协调模式下 sharding DDL lock 的处理。
> - 本文档的命令在交互模式中进行，因此在以下命令示例中未添加转义字符。在命令行模式中，你需要添加转义字符，防止报错。
> - 不要轻易使用 `shard-ddl-lock unlock` 命令，除非完全明确当前场景下使用这些命令可能会造成的影响，并能接受这些影响。
> - 在手动处理异常的 DDL lock 前，请确保已经了解 DM 的分库分表合并迁移原理。

命令介绍

`shard-ddl-lock`

该命令用于查看 DDL lock 和主动请求 DM-master 解除指定的 DDL lock。命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 `show-ddl-locks` 和 `unlock-ddl-lock` 命令。

```
shard-ddl-lock -h
```

```
maintain or show shard-ddl locks information

Usage:
  dmctl shard-ddl-lock [task] [flags]
  dmctl shard-ddl-lock [command]

Available Commands:
  unlock      Unlock un-resolved DDL locks forcely

Flags:
  -h, --help    help for shard-ddl-lock

Global Flags:
  -s, --source strings    MySQL Source ID.

Use "dmctl shard-ddl-lock [command] --help" for more information about a command.
```

参数解释

- `shard-ddl-lock [task] [flags]`:
  - 用于查询当前 DM-master 上存在的 DDL lock 信息
- `shard-ddl-lock [command]`
  - 用于主动请求 DM-master 解除指定的 DDL lock, command 只支持 `unlock`

命令示例

`shard-ddl-lock [task] [flags]`

使用 `shard-ddl-lock [task] [flags]` 命令，查询当前 DM-master 上存在的 DDL lock 信息。

例如：

```
shard-ddl-lock test
```

期望输出

```
{
    "result": true,                                     # 查询 lock 操作本身是否成功
    "msg": "",                                          # 查询 lock
        ↪ 操作失败时的原因或其它描述信息（如不存在任务 lock）
    "locks": [                                          # 当前存在的 lock 信息列表
        {
            "ID": "test-`shard_db`.`shard_table`",      # lock 的 ID 标识，当前由任务名与 DDL
                ↪ 对应的 schema/table 信息组成
            "task": "test",                             # lock 所属的任务名
```

```
            "mode": "pessimistic"                               # shard DDL 协调模式，可为悲观模式 "
                ↪ pessimistic" 或乐观模式 "optimistic"
            "owner": "mysql-replica-01",                        # lock 的 owner (
                ↪ 在悲观模式时为第一个遇到该 DDL 的 source ID)，在乐观模式时总为空
            "DDLs": [                                           # 在悲观模式时为 lock 对应的 DDL 列表
                ↪ ，在乐观模式时总为空
                "USE `shard_db`; ALTER TABLE `shard_db`.`shard_table` DROP COLUMN `c2`;"
            ],
            "synced": [                                         # 已经收到对应 MySQL 实例内所有分表
                ↪ DDL 的 source 列表
                "mysql-replica-01"
            ],
            "unsynced": [                                       # 尚未收到对应 MySQL 实例内所有分表
                ↪ DDL 的 source 列表
                "mysql-replica-02"
            ]
        }
    ]
}
```

shard-ddl-lock unlock

用于主动请求 DM-master 解除指定的 DDL lock，包括的操作：请求 owner 执行 DDL 操作，请求其他非 owner 的 DM-worker 跳过 DDL 操作，移除 DM-master 上的lock 信息。

> **注意：**
>
> shard-ddl-lock unlock 当前仅对悲观协调模式(pessimistic)下产生的lock 有效。

```
shard-ddl-lock unlock -h
```

```
Unlock un-resolved DDL locks forcely

Usage:
  dmctl shard-ddl-lock unlock <lock-id> [flags]


Flags:
  -a, --action string      accept skip/exec values which means whether to skip or execute ddls (
      ↪ default "skip")
  -d, --database string    database name of the table
  -f, --force-remove       force to remove DDL lock
  -h, --help               help for unlock
  -o, --owner string       source to replace the default owner
  -t, --table string       table name
```

```
Global Flags:
  -s, --source strings    MySQL Source ID.
```

shard-ddl-lock unlock 命令支持以下参数：

- -o, --owner：
  - flag 参数，string，可选
  - 不指定时，请求默认的 owner（shard-ddl-lock 返回结果中的 owner）执行 DDL 操作；指定时，请求该 MySQL source（替代默认的 owner）执行 DDL 操作
  - 除非原 owner 已经从集群中移除，否则不应该指定新的 owner
- -f, --force-remove：
  - flag 参数，boolean，可选
  - 不指定时，仅在 owner 执行 DDL 成功时移除lock 信息；指定时，即使 owner 执行 DDL 失败也强制移除 lock 信息（此后将无法再次查询或操作该lock）
- lock-id：
  - 非 flag 参数，string，必选
  - 指定需要执行unlock 操作的 DDL lock ID（即 shard-ddl-lock 返回结果中的 ID）

以下是一个使用 shard-ddl-lock unlock 命令的示例：

```
shard-ddl-lock unlock test-`shard_db`.`shard_table`
```

```
{
    "result": true,                                    # unlock lock 操作是否成功
    "msg": "",                                         # unlock lock 操作失败时的原因
}
```

支持场景

目前，使用 shard-ddl-lock unlock 命令仅支持处理以下两种 sharding DDL lock 异常情况。

场景一：部分 MySQL source 被移除

Lock 异常原因

在 DM-master 尝试自动unlock sharding DDL lock 之前，需要等待所有 MySQL source 的 sharding DDL events 全部到达（详见分库分表合并迁移原理）。如果 sharding DDL 已经在迁移过程中，同时有部分 MySQL source 被移除，且不再计划重新加载它们（按业务需求移除了这部分 MySQL source），则会由于永远无法等齐所有的 DDL 而造成 lock 无法自动unlock。

手动处理示例

假设上游有 MySQL-1（mysql-replica-01）和 MySQL-2（mysql-replica-02）两个实例， 其中 MySQL-1 中有 shard_db_1.shard_table_1 和 shard_db_1.shard_table_2 两个表，MySQL-2 中有 shard_db_2.shard_table_1 和 shard_db_2.shard_table_2 两个表。现在需要将这 4 个表合并后迁移到下游TiDB 的 shard_db.shard_table 表中。

初始表结构如下：

```
SHOW CREATE TABLE shard_db_1.shard_table_1;
```

```
+---------------+-----------------------------------------+
| Table         | Create Table                            |
+---------------+-----------------------------------------+
| shard_table_1 | CREATE TABLE `shard_table_1` (
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+---------------+-----------------------------------------+
```

上游分表将执行以下 DDL 语句变更表结构：

```
ALTER TABLE shard_db_*.shard_table_* ADD COLUMN c2 INT;
```

MySQL 及 DM 操作与处理流程如下：

1. mysql-replica-01 对应的两个分表执行了对应的 DDL 操作进行表结构变更。

   ```
   ALTER TABLE shard_db_1.shard_table_1 ADD COLUMN c2 INT;
   ```

   ```
   ALTER TABLE shard_db_1.shard_table_2 ADD COLUMN c2 INT;
   ```

2. DM-worker 接受到mysql-replica-01 两个分表的DDL 之后，将对应的DDL 信息发送给DM-master，DM-master 创建相应的 DDL lock。

3. 使用 shard-ddl-lock 查看当前的 DDL lock 信息。

   ```
   shard-ddl-lock test
   ```

   ```
   {
       "result": true,
       "msg": "",
       "locks": [
           {
               "ID": "test-`shard_db`.`shard_table`",
               "task": "test",
               "mode": "pessimistic"
               "owner": "mysql-replica-01",
               "DDLs": [
                   "USE `shard_db`; ALTER TABLE `shard_db`.`shard_table` ADD COLUMN `c2` int
                       ↪ (11);"
               ],
               "synced": [
                   "mysql-replica-01"
               ],
   ```

```
            "unsynced": [
                "mysql-replica-02"
            ]
        }
    ]
}
```

4. 由于业务需要，`mysql-replica-02` 对应的数据不再需要迁移到下游TiDB，对 `mysql-replica-02` 执行了移除操作。

5. DM-master 上 ID 为 `test-`shard_db`.`shard_table`` 的 lock 无法等到 `mysql-replica-02` 的 DDL 操作信息。`shard-ddl-lock` 返回的 unsynced 中一直包含 `mysql-replica-02` 的信息。

6. 使用 `shard-ddl-lock unlock` 来请求 DM-master 主动 unlock 该 DDL lock。

   - 如果 DDL lock 的 owner 也已经被移除，可以使用 `--owner` 参数指定其他 MySQL source 作为新 owner 来执行 DDL。
   - 当存在任意 MySQL source 报错时，`result` 将为 `false`，此时请仔细检查各 MySQL source 的错误是否是预期可接受的。

   ```
   shard-ddl-lock unlock test-`shard_db`.`shard_table`
   ```

   ```
   {
       "result": true,
       "msg": ""
   ```

7. 使用 `shard-ddl-lock` 确认 DDL lock 是否被成功unlock。

   ```
   shard-ddl-lock test
   ```

   ```
   {
       "result": true,
       "msg": "no DDL lock exists",
       "locks": [
       ]
   }
   ```

8. 查看下游TiDB 中的表结构是否变更成功。

   ```
   SHOW CREATE TABLE shard_db.shard_table;
   ```

   ```
   +-------------+-----------------------------------------------------+
   | Table       | Create Table                                        |
   +-------------+-----------------------------------------------------+
   | shard_table | CREATE TABLE `shard_table` (
     `c1` int(11) NOT NULL,
   ```

```
  `c2` int(11) DEFAULT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+-------------+-----------------------------------------------------+
```

9. 使用 query-status 确认迁移任务是否正常。

手动处理后的影响

使用 shard-ddl-lock unlock 手动执行unlock 操作后，由于该任务的配置信息中仍然包含了已下线的 MySQL source，如果不进行处理，则当下次 sharding DDL 到达时，仍会出现lock 无法自动完成迁移的情况。

因此，在手动解锁 DDL lock 后，需要再执行以下操作：

1. 使用 stop-task 停止运行中的任务。
2. 更新任务配置文件，将已下线 MySQL source 对应的信息从配置文件中移除。
3. 使用 start-task 及新任务配置文件重新启动任务。

> 注意：
>
> 在 shard-ddl-lock unlock 之后，如果已下线的 MySQL source 重新加载并尝试对其中的分表进行数据迁移，则会由于数据与下游的表结构不匹配而发生错误。

场景二：unlock 过程中部分 DM-worker 异常停止或网络中断

Lock 异常原因

在 DM-master 收到所有 DM-worker 的 DDL 信息后，执行自动unlock DDL lock 的操作主要包括以下步骤：

1. 请求lock owner 执行 DDL 操作，并更新对应分表的checkpoint。
2. 在 owner 执行 DDL 操作成功后，移除 DM-master 上保存的 DDL lock 信息。
3. 在 owner 执行 DDL 操作成功后，请求其他所有非 owner 跳过 DDL 操作并更新对应分表的checkpoint。
4. DM-master 在所有 owner/非 owner 操作成功后，移除对应的 DDL lock 信息。

上述unlock DDL lock 的操作不是原子的。如果非 owner 跳过 DDL 操作成功后，所在的 DM-worker 异常停止或与下游 TiDB 发生网络异常，造成无法成功更新checkpoint。

当非 owner 对应的 MySQL source 恢复数据迁移时，会尝试请求 DM-master 重新协调异常发生前已经协调过的 DDL、且永远无法等到其他 MySQL source 的对应 DDL，造成该 DDL 操作对应lock 的自动unlock。

手动处理示例

仍然假设是部分 MySQL source 被移除示例中的上下游表结构及合表迁移需求。

当在 DM-master 自动执行unlock 操作的过程中，owner (mysql-replica-01) 成功执行了 DDL 操作且开始继续进行后续迁移，但在请求非 owner (mysql-replica-02) 跳过 DDL 操作的过程中，由于对应的 DM-worker 发生了重启在跳过 DDL 后未能更新checkpoint。

mysql-replica-02 对应的数据迁移子任务恢复后，将在 DM-master 上创建一个新的lock，但其他 MySQL source 此时已经执行或跳过 DDL 操作并在进行后续迁移。

处理流程如下：

1. 使用 shard-ddl-lock 确认 DM-master 上存在该 DDL 操作对应的lock。

   应该仅有 mysql-replica-02 处于 synced 状态：

```
shard-ddl-lock
```

```
{
    "result": true,
    "msg": "",
    "locks": [
        {
            "ID": "test-`shard_db`.`shard_table`",
            "task": "test",
            "mode": "pessimistic"
            "owner": "mysql-replica-02",
            "DDLs": [
                "USE `shard_db`; ALTER TABLE `shard_db`.`shard_table` ADD COLUMN `c2` int
                    ↪ (11);"
            ],
            "synced": [
                "mysql-replica-02"
            ],
            "unsynced": [
                "mysql-replica-01"
            ]
        }
    ]
}
```

2. 使用 shard-ddl-lock unlock 请求 DM-master unlock 该 lock。

   - Lock 过程中会尝试再次向下游执行该 DDL 操作（重启前的原 owner 已向下游执行过该 DDL 操作），需要确保该 DDL 操作可被多次执行。

```
shard-ddl-lock unlock test-`shard_db`.`shard_table`
```

```
{
    "result": true,
    "msg": "",
}
```

3. 使用 shard-ddl-lock 确认 DDL lock 是否被成功unlock。

4. 使用 query-status 确认迁移任务是否正常。

手动处理后的影响

手动unlock sharding DDL lock 后，后续的 sharding DDL 将可以自动正常迁移。

10.6.7.2 迁移使用GH-ost/PT-osc 的源数据库

在生产业务中执行 DDL 时，产生的锁表操作会一定程度阻塞数据库的读取或者写入。为了把对读写的影响降到最低，用户往往会选择 online DDL 工具执行 DDL。常见的 Online DDL 工具有 gh-ost 和 pt-osc。

在使用 DM 完成 MySQL 到 TiDB 的数据迁移时，可以开启online-ddl配置，实现 DM 工具与 gh-ost 或 pt-osc 的协同。关于如何开启 online-ddl配置及开启该配置后的工作流程，请参考上游使用 pt-osc/gh-ost 工具的持续同步场景。本文仅介绍 DM 与 online DDL 工具协作的细节。

10.6.7.2.1 DM 与 online DDL 工具协作细节

DM 与 online DDL 工具 gh-ost 和 pt-osc 在实现 online-schema-change 过程中的协作细节如下。

online-schema-change: gh-ost

gh-ost 在实现 online-schema-change 的过程会产生 3 种 table：

- gho：用于应用 DDL，待 gho 表中数据迁移到与 origin table 一致后，通过rename 的方式替换 origin table。
- ghc：用于存放 online-schema-change 相关的信息。
- del：对 origin table 执行rename 操作而生成。

DM 在迁移过程中会把上述table 分成 3 类：

- ghostTable : _*_gho
- trashTable : _*_ghc、_*_del
- realTable : 执行 online-ddl 的 origin table

gh-ost 涉及的主要 SQL 以及 DM 的处理：

1. 创建 _ghc 表：

```
Create /* gh-ost */ table `test`.`_test4_ghc` (
                        id bigint auto_increment,
                        last_update timestamp not null DEFAULT CURRENT_TIMESTAMP ON UPDATE
                            ↪ CURRENT_TIMESTAMP,
                        hint varchar(64) charset ascii not null,
                        value varchar(4096) charset ascii not null,
                        primary key(id),
                        unique key hint_uidx(hint)
                ) auto_increment=256 ;
```

DM：不执行 _test4_ghc 的创建操作。

2. 创建 _gho 表：

```
Create /* gh-ost */ table `test`.`_test4_gho` like `test`.`test4` ;
```

DM：不执行 _test4_gho 的创建操作，根据 ghost_schema、ghost_table 以及 dm_worker 的 server_id，删除下游 dm_meta.{task_name}_onlineddl 的记录，清理内存中的相关信息。

```
DELETE FROM dm_meta.{task_name}_onlineddl WHERE id = {server_id} and ghost_schema = {
    '→ ghost_schema} and ghost_table = {ghost_table};
```

3. 在 _gho 表应用需要执行的 DDL：

```
Alter /* gh-ost */ table `test`.`_test4_gho` add column cl1 varchar(20) not null ;
```

DM：不执行 _test4_gho 的 DDL 操作，而是把该 DDL 记录到 dm_meta.{task_name}_onlineddl 以及内存中。

```
REPLACE INTO dm_meta.{task_name}_onlineddl (id, ghost_schema , ghost_table , ddls) VALUES
    '→ (. ..... );
```

4. 往 _ghc 表写入数据，以及往 _gho 表同步 origin table 的数据：

```
INSERT /* gh-ost */ INTO `test`.`_test4_ghc` VALUES (. ..... );

INSERT /* gh-ost `test`.`test4` */ ignore INTO `test`.`_test4_gho` (`id`, `date`, `
    '→ account_id`, `conversion_price`, `ocpc_matched_conversions`, `ad_cost`, `cl2`)
  (SELECT `id`, `date`, `account_id`, `conversion_price`, `ocpc_matched_conversions`, `
      '→ ad_cost`, `cl2` FROM `test`.`test4` FORCE INDEX (`PRIMARY`)
    WHERE (((`id` > _binary'1') OR ((`id` = _binary'1'))) AND ((`id` < _binary'2') OR ((`id`
        '→  = _binary'2')))) lock IN share mode
  )   ;
```

DM：只要不是 realtable 的 DML 全部不执行。

5. 数据同步完成后 origin table 与 _gho 一起改名，完成 online DDL 操作：

```
Rename /* gh-ost */ table `test`.`test4` to `test`.`_test4_del`, `test`.`_test4_gho` to `
    '→ test`.`test4`;
```

DM 执行以下两个操作:

- 把 rename 语句拆分成两个 SQL：

```
rename test.test4 to test._test4_del;
rename test._test4_gho to test.test4;
```

- 不执行 rename to _test4_del。当要执行 rename ghost_table to origin table 的时候，并不执行 rename 语句，而是把步骤 3 记录在内存中的 DDL 读取出来，然后把 ghost_table、ghost_schema 替换为 origin_table 以及对应的 schema，再执行替换后的 DDL。

```
alter table test._test4_gho add column cl1 varchar(20) not null;
--替换为
alter table test.test4 add column cl1 varchar(20) not null;
```

> 注意：
>
> 具体 gh-ost 的 SQL 会根据工具执行时所带的参数而变化。本文只列出主要的 SQL，具体可以参考 gh-ost 官方文档。

online-schema-change: pt

pt-osc 在实现 online-schema-change 的过程会产生 2 种 table：

- new：用于应用 DDL，待表中数据同步到与 origin table 一致后，再通过rename 的方式替换 origin table。
- old：对 origin table 执行rename 操作后生成。
- 3 种trigger：pt_osc_*_ins、pt_osc_*_upd、pt_osc_*_del，用于在 pt_osc 过程中，同步 origin table 新产生的数据到 new。

DM 在迁移过程中会把上述table 分成 3 类：

- ghostTable : _*_new
- trashTable : _*_old
- realTable : 执行的 online-ddl 的 origin table

pt-osc 主要涉及的 SQL 以及 DM 的处理：

1. 创建 _new 表：

```
CREATE TABLE `test`.`_test4_new` (id int(11) NOT NULL AUTO_INCREMENT,
date date DEFAULT NULL, account_id bigint(20) DEFAULT NULL, conversion_price decimal(20,3)
    '→ DEFAULT NULL,  ocpc_matched_conversions bigint(20) DEFAULT NULL, ad_cost decimal
    '→ (20,3) DEFAULT NULL,cl2 varchar(20) COLLATE utf8mb4_bin NOT NULL,cl1 varchar(20)
    '→ COLLATE utf8mb4_bin NOT NULL,PRIMARY KEY (id) ) ENGINE=InnoDB AUTO_INCREMENT=3
    '→ DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin ;
```

   DM: 不执行 _test4_new 的创建操作。根据 ghost_schema、ghost_table 以及 dm_worker 的 server_id，删除下游 dm_meta.{task_name}_onlineddl 的记录，清理内存中的相关信息。

```
DELETE FROM dm_meta.{task_name}_onlineddl WHERE id = {server_id} and ghost_schema = {
    '→ ghost_schema} and ghost_table = {ghost_table};
```

2. 在 _new 表上执行 DDL：

```
ALTER TABLE `test`.`_test4_new` add column c3 int;
```

DM: 不执行 _test4_new 的 DDL 操作，而是把该 DDL 记录到 dm_meta.{task_name}_onlineddl 以及内存中。

```
REPLACE INTO dm_meta.{task_name}_onlineddl (id, ghost_schema , ghost_table , ddls) VALUES
    '→ (. ..... );
```

3. 创建用于同步数据的 3 个 Trigger：

```
CREATE TRIGGER `pt_osc_test_test4_del` AFTER DELETE ON `test`.`test4` ..... ;
CREATE TRIGGER `pt_osc_test_test4_upd` AFTER UPDATE ON `test`.`test4` ..... ;
CREATE TRIGGER `pt_osc_test_test4_ins` AFTER INSERT ON `test`.`test4` ..... ;
```

DM: 不执行TiDB 不支持的相关Trigger 操作。

4. 往 _new 表同步 origin table 的数据：

```
INSERT LOW_PRIORITY IGNORE INTO `test`.`_test4_new` (`id`, `date`, `account_id`, `
    '→ conversion_price`, `ocpc_matched_conversions`, `ad_cost`, `cl2`, `cl1`) SELECT `id`,
    '→  `date`, `account_id`, `conversion_price`, `ocpc_matched_conversions`, `ad_cost`, `
    '→ cl2`, `cl1` FROM `test`.`test4` LOCK IN SHARE MODE /*pt-online-schema-change 3227
    '→ copy table*/
```

DM: 只要不是realTable 的 DML 全部不执行。

5. 数据同步完成后 origin table 与 _new 一起改名，完成 online DDL 操作：

```
RENAME TABLE `test`.`test4` TO `test`.`_test4_old`, `test`.`_test4_new` TO `test`.`test4`
```

DM 执行以下两个操作:

- 把 rename 语句拆分成两个 SQL。

  ```
  sql rename test.test4 to test._test4_old; rename test._test4_new to test.test4;
  ```

- 不执行 rename to _test4_old。当要执行 rename ghost_table to origin table 的时候，并不执行 rename，而是把步骤 2 记录在内存中的 DDL 读取出来，然后把 ghost_table、ghost_schema 替换为 origin_table 以及对应的 schema，再执行替换后的 DDL。

  ```
  sql  ALTER TABLE `test`.`_test4_new` add column c3 int; --替换为 ALTER TABLE `test`.`
  '→ test4` add column c3 int;
  ```

6. 删除 _old 表以及 online DDL 的 3 个 Trigger：

```
DROP TABLE IF EXISTS `test`.`_test4_old`;
DROP TRIGGER IF EXISTS `pt_osc_test_test4_del` AFTER DELETE ON `test`.`test4` ..... ;
DROP TRIGGER IF EXISTS `pt_osc_test_test4_upd` AFTER UPDATE ON `test`.`test4` ..... ;
DROP TRIGGER IF EXISTS `pt_osc_test_test4_ins` AFTER INSERT ON `test`.`test4` ..... ;
```

DM: 不执行 _test4_old 以及Trigger 的删除操作。

> **注意：**
>
> 具体 pt-osc 的 SQL 会根据工具执行时所带的参数而变化。本文只列出主要的 SQL，具体可以参考 pt-osc 官方文档。

10.6.7.2.2 其他 Online Schema Change 工具

在某些场景下，你可能需要变更 online schema change 工具的默认行为，自定义 `ghost table` 和 `trash table` 的名称；或者期望使用 `gh-ost` 和 `pt-osc` 之外的工具（原理和变更流程仍然保持一致）。此时则需要自行编写正则表达式以匹配`ghost table` 和 `trash table`。

自 v2.0.7 起，DM 实验性支持修改过的 online schema change 工具。在 DM 任务配置中设置 `online-ddl=true` 后，配合 `shadow-table-rules` 和 `trash-table-rules` 即可支持通过正则表达式来匹配修改过的临时表。

假设自定义pt-osc 的`ghost table` 规则为`_{origin_table}_pcnew`，`trash table` 规则为`_{origin_table}_pcold`，那么自定义规则需配置如下：

```
online-ddl: true
shadow-table-rules: ["^_(.+)_(?:pcnew)$"]
trash-table-rules: ["^_(.+)_(?:pcold)$"]
```

10.6.7.3 下游存在更多列的迁移场景

本文档介绍数据同步时，下游存在更多列的迁移场景需要的注意事项。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到TiDB
- 从大数据量 MySQL 迁移数据到TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到TiDB

10.6.7.3.1 使用 DM 迁移至存在更多列的下游

DM 同步上游的binlog 时，会尝试使用下游当前的表结构来解析binlog 并生成相应的 DML 语句。如果上游的 binlog 里数据表的列数与下游表结构的列数不一致，则会产生如下错误：

```
"errors": [
    {
        "ErrCode": 36027,
        "ErrClass": "sync-unit",
        "ErrScope": "internal",
        "ErrLevel": "high",
         "Message": "startLocation: [position: (mysql-bin.000001, 2022), gtid-set:09bec856-ba95-11
             ′→ ea-850a-58f2b4af5188:1-9 ], endLocation: [position: (mysql-bin.000001, 2022),
             ′→ gtid-set: 09bec856-ba95-11ea-850a-58f2b4af5188:1-9]: gen insert sqls failed,
```

```
            '→ schema: log, table: messages: Column count doesn't match value count: 3 (columns)
            '→  vs 2 (values)",
        "RawCause": "",
        "Workaround": ""
    }
]
```

例如上游表结构为：

```
#### 上游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
)
```

例如下游表结构为：

```
#### 下游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  `message` varchar(255) DEFAULT NULL, # 下游比上游多出的列。
  PRIMARY KEY (`id`)
)
```

当 DM 尝试使用下游表结构解析上游产生的binlog event时，DM 会报出上述 `Column count doesn't match` 错误。

此时，你可以使用 `binlog-schema` 命令来为数据源中需要迁移的表指定表结构，表结构需要对应 DM 将要开始同步的binlog event 的数据。如果你在进行分表合并的数据迁移，那么需要为每个分表按照如下步骤在 DM 中设置用于解析binlog event 的表结构。具体操作为：

1. 在DM 中，新建一个`.sql` 文件，并将上游表结构对应的`CREATE TABLE` 语句添加到该文件。例如，将以下表结构保存到 `log.messages.sql` 中。如果是 6.0 及以上版本，可以直接通过 `--from-source/--from-target` 更新，无需创建 SQL 文件。可参考管理迁移表的表结构。

   ```
   # 上游表结构
   CREATE TABLE `messages` (
   `id` int(11) NOT NULL,
   PRIMARY KEY (`id`)
   )
   ```

2. 使用 `binlog-schema` 命令为数据源中需要迁移的表设置表结构（此时数据迁移任务应该由于上述 `Column count doesn't match` 错误而处于 Paused 状态）。

   ```
   tiup dmctl --master-addr ${advertise-addr} binlog-schema update -s ${source-id} ${task-name}
       '→ ${database-name} ${table-name} ${schema-file}
   ```

   该命令中的参数描述如下：

参数	描述
--masterdmctl′→ -′→ addr′→	指定要连接的集群的任意 DM-master 节点的 ${′→ advertise′→ -′→ addr′→ }。${′→ advertise′→ -′→ addr′→ } 表示 DM-master 向外界宣告的地址。
binlog′→ -′→ schemaschema′→ update′→	手动更新信息
-s	指定 source。${′→ source′→ -′→ id′→ } 表示 MySQL 数据源 ID。

参数	描述
${ ′→ task ′→ - ′→ name ′→ }	指定 task。表示数据同步任务配置文件 task.′→ yaml ′→ 中定义的同步任务名称。
${ ′→ databadsaetabase ′→ - ′→ name ′→ }	指定 databadsaetabase。表示上游数据库名。
${ ′→ table ′→ - ′→ name ′→ }	指定 table。表示上游数据表名。
${ ′→ schema ′→ - ′→ file ′→ }	指定表的 schema 文件。表示将被设置的表结构文件。

例如：

```
tiup dmctl --master-addr 172.16.10.71:8261 binlog-schema update -s mysql-01 task-test -d log
    ′→  -t message log.message.sql
```

3. 使用 resume-task 命令恢复处于 Paused 状态的同步任务。

```
tiup dmctl --master-addr ${advertise-addr} resume-task ${task-name}
```

4. 使用 query-status 命令确认数据迁移任务是否运行正常。

```
tiup dmctl --master-addr ${advertise-addr} query-status resume-task ${task-name}
```

10.6.7.4 DM 增量数据校验

本文介绍了如何使用 DM 增量数据校验功能、DM 增量数据校验的原理以及相关的使用限制。

10.6.7.4.1 使用场景

在将增量数据从上游迁移到下游数据库的过程中，数据的流转有小概率导致错误或者丢失的情况。对于需要依赖于强数据一致的场景，如信贷、证券等业务，你可以在数据迁移完成之后对数据进行全量校验，确保数据的一致性。然而，在某些增量复制的业务场景下，上游和下游的写入是持续的、不会中断的，因为上下游的数据在不断变化，导致用户难以对表里面的全部数据进行一致性校验（例如使用sync-diff-inspector）。

在增量数据复制的场景下，你可以使用 DM 的增量校验功能，在数据持续写入下游的增量复制过程中确保迁移数据的完整性、一致性。

10.6.7.4.2 开启增量数据校验

你可以使用下面任意方法开启增量数据校验：

- 在任务配置中开启
- 通过 dmctl 开启

方法 1：在任务配置中开启

你可以在任务配置文件中加入以下内容来开启增量数据校验：

```
#### 给需要开启增量校验功能的上游数据库添加增量校验配置
mysql-instances:
  - source-id: "mysql1"
    block-allow-list: "bw-rule-1"
    validator-config-name: "global"
validators:
  global:
    mode: full # 也可以是 fast，默认是 none，即不开启校验
    worker-count: 4 # 后台校验的 validation worker 数量，默认是 4 个
    row-error-delay: 30m # 某一行多久没有验证通过会被标记为 error row，默认是 30m，即 30 分钟
```

示例中各配置项的含义如下：

- mode：校验模式，可以是 none、full、fast。

  - none：默认值，即不开启校验。
  - full：将变动行和下游数据库中获取的行数据进行每列对比。
  - fast：只判断这一行在下游数据库是否存在。

- worker-count：增量校验功能使用的 worker 数量（每个 worker 都是一个 goroutine）。
- row-error-delay：某一行多久没有验证通过会被标记为 error row，默认是 30 分钟。

完整配置请查阅DM 任务完整配置文件。

方法 2：通过dmctl 开启

你可以使用 dmctl validation start 命令来开启增量数据校验：

```
Usage:
  dmctl validation start [--all-task] [task-name] [flags]


Flags:
      --all-task            whether applied to all tasks
  -h, --help                help for start
      --mode string         specify the mode of validation: full (default), fast; this flag will
          '→ be ignored if the validation task has been ever enabled but currently paused (
          '→ default "full")
      --start-time string   specify the start time of binlog for validation, e.g. '2021-10-21
          '→ 00:01:00' or 2021-10-21T00:01:00
```

- --mode：指定开启的模式，可以是 fast 或者 full。
- --start-time：指定 validator 开启校验的位置，格式是：2021-10-21 00:01:00 或者 2021-10-21T00:01:00。
- task-name：需要开启增量数据校验的任务名，你也可以用 --all-task 来为当前所有任务开启增量数据校验。

示例：

```
dmctl --master-addr=127.0.0.1:8261 validation start --start-time 2021-10-21T00:01:00 --mode full
    '→ my_dm_task
```

10.6.7.4.3 使用增量数据校验

在使用增量数据校验时，通过dmctl 工具，你可以查询到增量校验当前的校验状态，也可以对校验出的错误行（error row）进行及时处理。所谓的错误行，就是在增量校验过程中，被检查出上下游数据不一致的行。

查看增量校验的状态

你可以使用两种方式查看增量校验的状态。

方式 1：用 dmctl query-status <task-name> 命令查看任务状态，如果开启了增量校验，校验结果会显示在每个 subtask 的 validation 字段里面。示例输出：

```
"subTaskStatus": [
    {
```

```
        "name": "test",
        "stage": "Running",
        "unit": "Sync",
        "result": null,
        "unresolvedDDLLockID": "",
        "sync": {
            ...
        },
        "validation": {
            "task": "test", // 任务名
            "source": "mysql-01", // source id
            "mode": "full", // 校验模式
            "stage": "Running", // 当前状态，Running 或者 Stopped
            "validatorBinlog": "(mysql-bin.000001, 5989)", // 校验到的 binlog 位置
            "validatorBinlogGtid": "1642618e-cf65-11ec-9e3d-0242ac110002:1-30", // 同上，用 GTID
                ↪ 表示
            "result": null, // 当增量校验异常时，显示异常信息
            "processedRowsStatus": "insert/update/delete: 0/0/0", // 已经处理的 binlog
                ↪ 数据行的统计信息
            "pendingRowsStatus": "insert/update/delete: 0/0/0", // 还未校验或者校验失败，
                ↪ 但还没标记为`错误行`的数据行统计信息
            "errorRowsStatus": "new/ignored/resolved: 0/0/0" // `错误行`统计信息，
                ↪ 三种状态的错误会在下文讲解
        }
    }
]
```

方式 2：使用 `dmctl validation status <taskname>` 来查询增量校验的状态：

```
dmctl validation status [--table-stage stage] <task-name> [flags]
Flags:
  -h, --help                 help for status
      --table-stage string   filter validation tables by stage: running/stopped
```

在上述命令中，你可以设置 `--table-stage` 来过滤正在校验或者已经停止校验的表。示例输出：

```
{
    "result": true,
    "msg": "",
    "validators": [
        {
            "task": "test",
            "source": "mysql-01",
            "mode": "full",
            "stage": "Running",
            "validatorBinlog": "(mysql-bin.000001, 6571)",
```

```
            "validatorBinlogGtid": "",
            "result": null,
            "processedRowsStatus": "insert/update/delete: 2/0/0",
            "pendingRowsStatus": "insert/update/delete: 0/0/0",
            "errorRowsStatus": "new/ignored/resolved: 0/0/0"
        }
    ],
    "tableStatuses": [
        {
            "source": "mysql-01", // source id
            "srcTable": "`db`.`test1`", // 源表名
            "dstTable": "`db`.`test1`", // 目标表名
            "stage": "Running", // 校验状态
            "message": "" // 具体错误信息显示
        }
    ]
}
```

如果你想要查询错误行的详细信息，比如错误原因、错误时间等，可以使用 `dmctl validation show-error` 命令：

```
Usage:
  dmctl validation show-error [--error error-state] <task-name> [flags]

Flags:
      --error string    filtering type of error: all, ignored, or unprocessed (default "
          ↪ unprocessed")
  -h, --help            help for show-error
```

示例输出：

```
{
    "result": true,
    "msg": "",
    "error": [
        {
            "id": "1", // 错误行标识符，在后续的处理错误行中用到
            "source": "mysql-replica-01", // source id
            "srcTable": "`validator_basic`.`test`", // 错误行源表
            "srcData": "[0, 0]", // 错误行具体数据
            "dstTable": "`validator_basic`.`test`", // 错误行目标表
            "dstData": "[]", // 错误行在下游的数据
            "errorType": "Expected rows not exist", // 错误原因
            "status": "NewErr", // 错误状态
            "time": "2022-07-04 13:33:02", // 错误行发现时间
            "message": "" // 额外信息
```

```
        }
    ]
}
```

处理增量校验错误行

当增量数据校验发现错误行后，你需要手动处理这些错误行。

在增量校验出现错误行时，增量校验不会停下，而是会把这些错误行记录下来，让用户自己去发现处理。错误行没有被处理时，默认状态是 unprocessed。如果你在下游手动矫正了该错误行的错误，增量校验也不会去自动获取矫正后的信息，仍会将该错误行记录在 error 中。

如果你不想在 validation status 中再看到这个错误行、或者你需要给已经解决的错误行打上标记，你可以使用 validation show-error 找到错误行的id，然后使用错误处理命令来对这些错误行进行处理或者标记。

dmctl 提供了三种错误处理命令：

- clear-error：清理掉错误行，show-error命令将不再展示该error row。

```
Usage:
  dmctl validation clear-error <task-name> <error-id|--all> [flags]

Flags:
      --all    all errors
  -h, --help   help for clear-error
```

- ignore-error：忽略该错误行，将这个错误行标记为 ignored。

```
Usage:
  dmctl validation ignore-error <task-name> <error-id|--all> [flags]

Flags:
      --all    all errors
  -h, --help   help for ignore-error
```

- resolve-error：已手动解决该错误行，将这个错误行标记为 resolved。

```
Usage:
  dmctl validation resolve-error <task-name> <error-id|--all> [flags]

Flags:
      --all    all errors
  -h, --help   help for resolve-error
```

10.6.7.4.4 停止增量数据校验

如果你需要停止增量数据校验，可以使用 validation stop 命令：

```
Usage:
  dmctl validation stop [--all-task] [task-name] [flags]

Flags:
      --all-task    whether applied to all tasks
  -h, --help        help for stop
```

用法可参考`dmctl validation start`命令。

10.6.7.4.5 原理

DM 增量校验（validator）的简要架构如下所示：

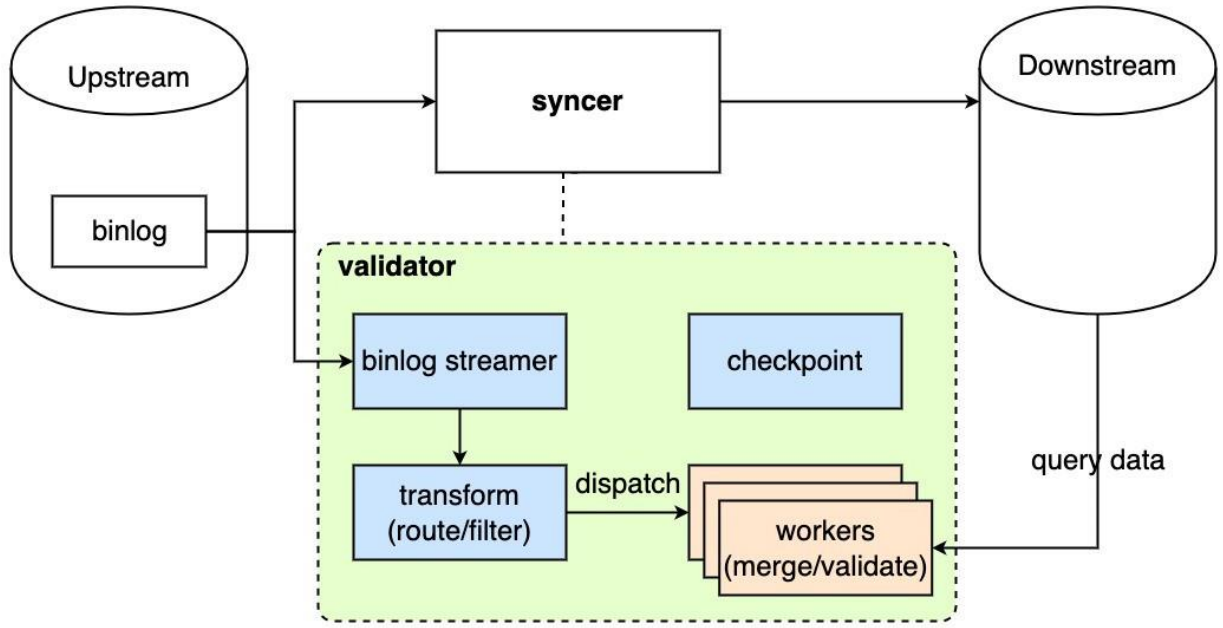


图 181: validator summary

增量数据校验的工作生命周期如下所示：

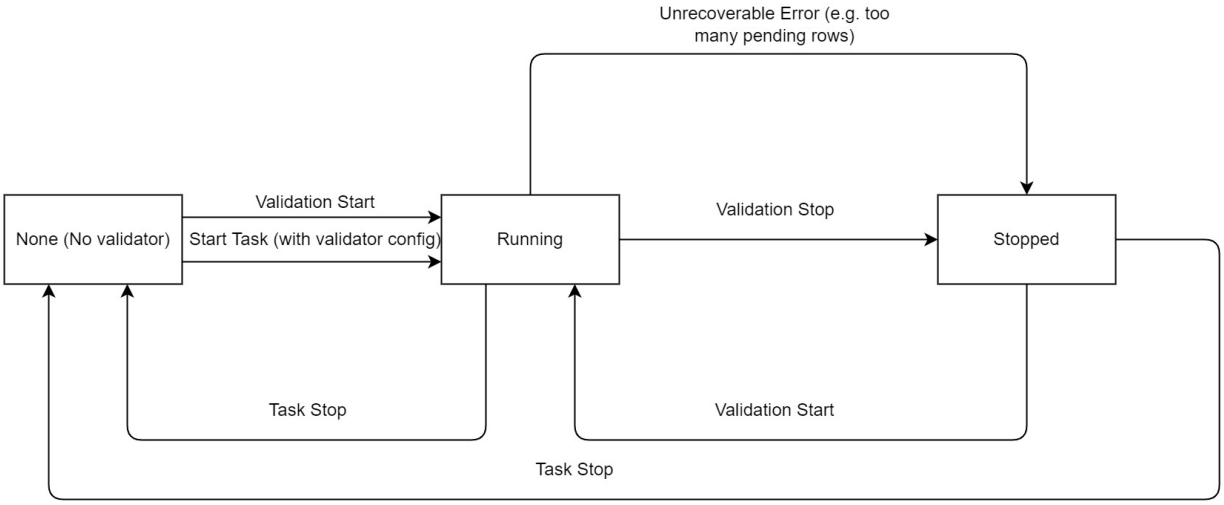


图 182: validator lifecycle

增量数据校验的具体处理流程如下：

1. validator 从上游拉取binlog 事件，获取到发生变更的数据行：
    - validator 只会校验增量复制(syncer) 完成的事件，如果该事件还没有被 syncer 处理，则 validator 会暂停，等待 syncer 处理完成。
    - 如果该事件已被 syncer 处理完成，则进行下面的步骤。
2. validator 将 binlog 解析，并通过黑白名单、过滤器的筛选，表路由的重定向（和 syncer 保持一致）之后，将这些行变动交给在后台运行的 validation worker。
3. validation worker 合并相同表、相同主键的行变动，避免进行“过期”的校验，并将这些行先缓存到内存中。
4. 当 validation worker 积攒了一定数量的行或者到了某个时间间隔之后，validation worker 根据这些行的主键信息在下游数据库中查询当下的数据，并和变动行期望的数据进行对比。
5. validation worker 进行数据校验。如果当前配置是 full mode，会将变动行和下游数据库中获取的行数据进行每列对比；如果当前配置是 fast mode，则只会判断这一行是否还存在。
    - 如果校验成功，则将该行从内存中删除。
    - 如果校验失败，不会马上报错，而是在一定间隔之后继续校验。
    - 对于某些已经在很长时间（由用户定义）内都没有校验成功的行，则将这些行定义为错误行（error row），写入到下游的 meta 库中。你可以通过查询迁移任务信息来获取错误行的数量等信息，详见查看增量校验的状态以及处理增量校验错误行。

10.6.7.4.6 使用限制

- 校验目标表必须有主键或者not null 的唯一键。
- 上游迁移 DDL 时有以下限制：
    - DDL 不能变更主键，不能调整列顺序，不能删除已有列。

  - 该表不能被 DROP。

- 不支持按照 expression 过滤事件的任务。
- 由于TiDB 和 MySQL 的浮点数精度有差异，精度范围内误差也会判断为相等（即绝对误差小于 10^-6)。
- 不支持校验的数据类型：

  - JSON
  - 二进制数据

10.6.8 运维管理

10.6.8.1 集群版本升级

10.6.8.1.1 使用TiUP 运维 DM 集群

本文介绍如何使用TiUP 的 DM 组件运维 DM 集群。

如果你还未部署 DM 集群，可参考使用TiUP 部署 DM 集群。

> 注意：
>
> - 需要确保以下组件间端口可正常连通：
>
>   - 各 DM-master 节点间的 peer_port（默认为 8291）可互相连通。
>   - 各 DM-master 节点可连通所有 DM-worker 节点的 port（默认为 8262）。
>   - 各 DM-worker 节点可连通所有 DM-master 节点的 port（默认为 8261）。
>   - TiUP 节点可连通所有 DM-master 节点的 port（默认为 8261）。
>   - TiUP 节点可连通所有 DM-worker 节点的 port（默认为 8262）。

TiUP DM 组件的帮助信息如下：

```
tiup dm --help
```

```
Deploy a DM cluster for production

Usage:
  tiup dm [flags]
  tiup dm [command]

Available Commands:
  deploy      Deploy a DM cluster for production
  start       Start a DM cluster
  stop        Stop a DM cluster
  restart     Restart a DM cluster
  list        List all clusters
```

```
  destroy     Destroy a specified DM cluster
  audit       Show audit log of cluster operation
  exec        Run shell command on host in the dm cluster
  edit-config Edit DM cluster config
  display     Display information of a DM cluster
  reload      Reload a DM cluster's config and restart if needed
  upgrade     Upgrade a specified DM cluster
  patch       Replace the remote package with a specified package and restart the service
  scale-out   Scale out a DM cluster
  scale-in    Scale in a DM cluster
  import      Import an exist DM 1.0 cluster from dm-ansible and re-deploy 2.0 version
  help        Help about any command


Flags:
  -h, --help                help for tiup-dm
      --native-ssh          Use the native SSH client installed on local system instead of the
          '→ build-in one.
      --ssh-timeout int     Timeout in seconds to connect host via SSH, ignored for operations
          '→ that don't need an SSH connection. (default 5)
  -v, --version             version for tiup-dm
      --wait-timeout int    Timeout in seconds to wait for an operation to complete, ignored for
          '→ operations that don't fit. (default 60)
  -y, --yes                 Skip all confirmations and assumes 'yes'
```

查看集群列表

集群部署成功后，可以通过 `tiup dm list` 命令在集群列表中查看该集群：

```
tiup dm list
```

```
Name  User  Version  Path                                       PrivateKey
----  ----  -------  ----                                       ----------
prod-cluster  tidb  ${version}  /root/.tiup/storage/dm/clusters/test  /root/.tiup/storage/dm/
    '→ clusters/test/ssh/id_rsa
```

启动集群

集群部署成功后，可以执行以下命令启动该集群。如果忘记了部署的集群名字，可以使用 `tiup dm list` 命令查看。

```
tiup dm start prod-cluster
```

检查集群状态

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，TiUP 提供了 `tiup dm display` 命令，用法如下：

```
tiup dm display prod-cluster
```

```
dm Cluster: prod-cluster
dm Version: ${version}
ID                  Role          Host          Ports       OS/Arch       Status     Data Dir
    ′→                            Deploy Dir
--                  -----         -----         ------      --------      ----------     ---------
    ′→                            ----------
172.19.0.101:9093   alertmanager  172.19.0.101  9093/9094   linux/x86_64  Up         /home/tidb/
    ′→ data/alertmanager-9093     /home/tidb/deploy/alertmanager-9093
172.19.0.101:8261   dm-master     172.19.0.101  8261/8291   linux/x86_64  Healthy|L  /home/tidb/
    ′→ data/dm-master-8261        /home/tidb/deploy/dm-master-8261
172.19.0.102:8261   dm-master     172.19.0.102  8261/8291   linux/x86_64  Healthy    /home/tidb/
    ′→ data/dm-master-8261        /home/tidb/deploy/dm-master-8261
172.19.0.103:8261   dm-master     172.19.0.103  8261/8291   linux/x86_64  Healthy    /home/tidb/
    ′→ data/dm-master-8261        /home/tidb/deploy/dm-master-8261
172.19.0.101  :8262 dm-worker     172.19.0.101  8262        linux/x86_64  Free       /home/tidb/
    ′→ data/dm-worker-8262        /home/tidb/deploy/dm-worker-8262
172.19.0.102  :8262 dm-worker     172.19.0.102  8262        linux/x86_64  Free       /home/tidb/
    ′→ data/dm-worker-8262        /home/tidb/deploy/dm-worker-8262
172.19.0.103  :8262 dm-worker     172.19.0.103  8262        linux/x86_64  Free       /home/tidb/
    ′→ data/dm-worker-8262        /home/tidb/deploy/dm-worker-8262
172.19.0.101:3000   grafana       172.19.0.101  3000        linux/x86_64  Up         -
    ′→                                    /home/tidb/deploy/grafana-3000
172.19.0.101:9090   prometheus    172.19.0.101  9090        linux/x86_64  Up         /home/tidb/
    ′→ data/prometheus-9090       /home/tidb/deploy/prometheus-9090
```

Status 列用 Up 或者 Down 表示该服务是否正常。对于 DM-master 组件，同时可能会带有 |L 表示该 DM-master 是 Leader，对于 DM-worker 组件，Free 表示当前 DM-worker 没有与上游绑定。

缩容节点

缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关数据文件。

缩容操作进行时，内部对 DM-master、DM-worker 组件的操作流程为：

1. 停止组件进程
2. 调用 DM-master 删除 member 的 API
3. 清除节点的相关数据文件

缩容命令的基本用法：

```
tiup dm scale-in <cluster-name> -N <node-id>
```

它需要指定至少两个参数，一个是集群名字，另一个是节点 ID。节点 ID 可以参考上一节使用 `tiup dm display` 命令获取。

比如想缩容 172.16.5.140 上的 DM-worker 节点（DM-master 的缩容类似），可以执行：

```
tiup dm scale-in prod-cluster -N 172.16.5.140:8262
```

扩容节点

扩容的内部逻辑与部署类似，TiUP DM 组件会先保证节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。

例如，在集群 `prod-cluster` 中扩容一个 DM-worker 节点（DM-master 的扩容类似）：

1. 新建 scale.yaml 文件，添加新增的 woker 节点信息：

> 注意：
> 需要新建一个拓扑文件，文件中只写入扩容节点的描述信息，不要包含已存在的节点。
> 其他更多配置项（如：部署目录等）请参考TiUP 配置参数模版。

```
---

worker_servers:
  - host: 172.16.5.140
```

2. 执行扩容操作。TiUP DM 根据 scale.yaml 文件中声明的端口、目录等信息在集群中添加相应的节点：

```
tiup dm scale-out prod-cluster scale.yaml
```

执行完成之后可以通过 `tiup dm display prod-cluster` 命令检查扩容后的集群状态。

滚动升级

> 注意：
> 从 v2.0.5 版本开始，dmctl 支持导出和导入集群的数据源和任务配置。
> 升级前，可使用 `config export` 命令导出集群的配置文件，升级后如需降级回退到旧版本，可重建旧集群后，使用 `config import` 导入之前的配置。
> 对于 v2.0.5 之前版本的集群，可使用 v2.0.5 及之后版本的dmctl 导出和导入集群配置。
> 对于 v2.0.2 之后的版本，导入集群配置时暂不支持自动恢复 relay worker 相关配置，可手动执行 `start-relay` 命令开启relay log。

滚动升级过程中尽量保证对前端业务透明、无感知，其中对不同节点有不同的操作。

升级操作

可使用 `tiup dm upgrade` 命令来升级集群。例如，以下示例将集群升级到 `${version}`，执行命令前，将 `${version}` 替换为实际需要的版本：

```
tiup dm upgrade prod-cluster `${version}`
```

更新配置

如果想要动态更新组件的配置，TiUP DM 组件为每个集群保存了一份当前的配置，如果想要编辑这份配置，则执行 `tiup dm edit-config <cluster-name>` 命令。例如：

```
tiup dm edit-config prod-cluster
```

然后 TiUP DM 组件会使用 vi 打开配置文件供编辑（如果你想要使用其他编辑器，请使用 `EDITOR` 环境变量自定义编辑器，例如 `export EDITOR=nano`），编辑完之后保存即可。此时的配置并没有应用到集群，如果想要让它生效，还需要执行：

```
tiup dm reload prod-cluster
```

该操作会将配置发送到目标机器，滚动重启集群，使配置生效。

更新组件

常规的升级集群可以使用 `upgrade` 命令，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 `patch` 命令：

```
tiup dm patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup dm patch <cluster-name> <package-path> [flags]

Flags:
  -h, --help                    help for patch
  -N, --node strings            Specify the nodes
      --overwrite               Use this package in the future scale-out operations
  -R, --role strings            Specify the role
      --transfer-timeout int    Timeout in seconds when transferring dm-master leaders (default
          ↪ 600)


Global Flags:
      --native-ssh          Use the native SSH client installed on local system instead of the
          ↪ build-in one.
      --ssh-timeout int     Timeout in seconds to connect host via SSH, ignored for operations
          ↪ that don't need an SSH connection. (default 5)
      --wait-timeout int    Timeout in seconds to wait for an operation to complete, ignored for
          ↪ operations that don't fit. (default 60)
  -y, --yes                 Skip all confirmations and assumes 'yes'
```

例如，有一个 DM-master 的 hotfix 包放在 `/tmp/dm-master-hotfix.tar.gz`，如果此时想要替换集群上的所有 DM-master，则可以执行：

```
tiup dm patch prod-cluster /tmp/dm-master-hotfix.tar.gz -R dm-master
```

或者只替换其中一个 DM-master：

```
tiup dm patch prod-cluster /tmp/dm--hotfix.tar.gz -N 172.16.4.5:8261
```

导入和升级由 DM-Ansible 部署的 DM 1.0 集群

> 注意：
>
> - TiUP 不支持在 DM 1.0 集群中导入 DM Portal 组件。
> - 导入 DM 集群前，终止运行待导入的集群。
> - 如果需要将迁移任务升级至 2.0，不可对该任务执行 `stop-run` 命令。
> - TiUP 仅支持导入数据到 v2.0.0-rc.2 或以上版本的 DM 集群。
> - `import` 命令用于将数据从 DM 1.0 集群导入到一个新的 DM 2.0 集群。如果你需要将 DM 迁移任务导入一个现有的 DM 2.0 集群，可以参考TiDB Data Migration 1.0.x 到 2.0+ 手动升级。
> - 导入集群某些组件的部署目录与原始集群的部署目录不同，可以执行 `display` 命令来查看相关信息。
> - 导入前，执行 `tiup update --self && tiup update dm`，以确保 DM 组件是最新版本。
> - 导入后，集群中只存在一个 DM-master 节点。请参考扩容节点来扩展 DM-master。

引入TiUP 前，DM-Ansible 用于部署 DM 集群。要使TiUP 接管由 DM-Ansible 部署的 DM 1.0 集群，需要执行 `import` 命令：

例如，以下命令可导入使用 DM-Ansible 部署的集群：

```
tiup dm import --dir=/path/to/dm-ansible --cluster-version ${version}
```

可以执行 `tiup list dm-master` 来查看TiUP 支持的最新集群版本。

`import` 命令的执行过程如下：

1. 根据之前使用 DM-Ansible 部署的 DM 集群，TiUP 生成一个拓扑文件 `topology.yml`。
2. 确认拓扑文件生成后，你可以用这个文件来部署 v2.0 或更高版本的 DM 集群。

部署完成后，执行 `tiup dm start` 命令来启动集群和开始 DM 内核升级流程。

查看操作日志

操作日志的查看可以借助 `audit` 命令，其用法如下：

```
Usage:
  tiup dm audit [audit-id] [flags]

Flags:
  -h, --help   help for audit
```

在不使用 `[audit-id]` 参数时，该命令会显示执行的命令列表，如下：

```
tiup dm audit
```

```
ID       Time                  Command
--       ----                  -------
4D5kQY   2020-08-13T05:38:19Z  tiup dm display test
4D5kNv   2020-08-13T05:36:13Z  tiup dm list
4D5kNr   2020-08-13T05:36:10Z  tiup dm deploy -p prod-cluster ${version} ./examples/dm/minimal.
    ↪ yaml
```

第一列为audit-id，如果想看某个命令的执行日志，则传入这个audit-id：

```
tiup dm audit 4D5kQY
```

在集群节点机器上执行命令

exec 命令可以很方便地到集群的机器上执行命令，使用方式如下：

```
Usage:
  tiup dm exec <cluster-name> [flags]

Flags:
      --command string    the command run on cluster host (default "ls")
  -h, --help              help for exec
  -N, --node strings      Only exec on host with specified nodes
  -R, --role strings      Only exec on host with specified roles
      --sudo              use root permissions (default false)
```

例如，如果要到所有的 DM 节点上执行 `ls /tmp`，则可以执行：

```
tiup dm exec prod-cluster --command='ls /tmp'
```

集群控制工具(dmctl)

TiUP 集成了 DM 的控制工具 `dmctl`：

运行命令如下：

```
tiup dmctl [args]
```

指定dmctl 版本。执行如下命令前，将${version}修改为实际需要的版本。

```
tiup dmctl: ${version} [args]
```

例如，以前添加source 命令为`dmctl --master-addr master1:8261 operate-source create /tmp/source1.yml`，集成到TiUP 中的命令为：

```
tiup dmctl --master-addr master1:8261 operate-source create /tmp/source1.yml
```

使用中控机系统自带的 SSH 客户端连接集群

在以上所有操作中，涉及到对集群机器的操作都是通过TiUP 内置的 SSH 客户端连接集群执行命令，但是在某些场景下，需要使用系统自带的 SSH 客户端来对集群执行操作，比如：

- 使用 SSH 插件来做认证
- 使用定制的 SSH 客户端

此时可以通过命令行参数 `--native-ssh` 启用系统自带命令行：

- 部署集群：`tiup dm deploy <cluster-name> <version> <topo> --native-ssh`，其中 `<cluster-name>` 为集群名称，`<version>` 为 DM 集群版本（例如 `v6.5.0`），`<topo>` 为拓扑文件路径
- 启动集群：`tiup dm start <cluster-name> --native-ssh`
- 升级集群：`tiup dm upgrade ... --native-ssh`

所有涉及集群操作的步骤都可以加上 `--native-ssh` 来使用系统自带的客户端。

也可以使用环境变量 `TIUP_NATIVE_SSH` 来指定是否使用本地 SSH 客户端，避免每个命令都需要添加 `--native-ssh` 参数：

```
export TIUP_NATIVE_SSH=true
##### 或者
export TIUP_NATIVE_SSH=1
##### 或者
export TIUP_NATIVE_SSH=enable
```

若环境变量和 `--native-ssh` 同时指定，则以 `--native-ssh` 为准。

> 注意：
>
> 在部署集群的步骤中，若需要使用密码的方式连接(-p)，或者密钥文件设置了passphrase，则需要保证中控机上安装了 sshpass，否则连接时会报错。

10.6.8.1.2 TiDB Data Migration 1.0.x 到 2.0+ 手动升级

本文档主要介绍如何手动从 DM v1.0.x 升级到 v2.0+，主要思路为利用 v1.0.x 时的全局checkpoint 信息在 v2.0+ 集群中启动一个新的增量数据复制任务。

> 注意：
>
> - DM 当前不支持在数据迁移任务处于全量导出或全量导入过程中从 v1.0.x 升级到 v2.0+。
> - 由于 DM 各组件间用于交互的 gRPC 协议进行了较大变更，因此需确保升级前后 DM 集群各组件（包括dmctl）使用相同的版本。

- 由于 DM 集群的元数据存储（如 checkpoint、 shard DDL lock 状态及 online DDL 元信息等）发生了较大变更，升级到 v2.0+ 后无法自动复用 v1.0.x 的元数据，因此在执行升级操作前需要确保：
  - 所有数据迁移任务不处于 shard DDL 协调过程中。
  - 所有数据迁移任务不处于 online DDL 协调过程中。

下面是手动升级的具体步骤。

第 1 步：准备 v2.0+ 的配置文件

准备的 v2.0+ 的配置文件包括上游数据库的配置文件以及数据迁移任务的配置文件。

上游数据库配置文件

在 v2.0+ 中将上游数据库 source 相关的配置从 DM-worker 的进程配置中独立了出来，因此需要根据v1.0.x 的 DM-worker 配置拆分得到 source 配置。

> 注意：
>
> 当前从 v1.0.x 升级到 v2.0+ 时，如在 source 配置中启用了 `enable-gtid`，则后续需要通过解析 binlog 或 relay log 文件获取binlog position 对应的GTID sets。

从 DM-Ansible 部署的 v1.0.x 升级

如果 v1.0.x 是使用 DM-Ansible 部署的，且假设在 `inventory.ini` 中有如下 `dm_worker_servers` 配置：

```
[dm_master_servers]
dm_worker1 ansible_host=172.16.10.72 server_id=101 source_id="mysql-replica-01" mysql_host
    ‘→ =172.16.10.81 mysql_user=root mysql_password='VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU='
    ‘→ mysql_port=3306
 dm_worker2 ansible_host=172.16.10.73 server_id=102 source_id="mysql-replica-02" mysql_host
     ‘→ =172.16.10.82 mysql_user=root mysql_password='VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU='
    ‘→ mysql_port=3306
```

则可以转换得到如下两个 source 配置文件：

```
##### 原 dm_worker1 对应的 source 配置，如命名为 source1.yaml
server-id: 101                                  # 对应原 `server_id`
source-id: "mysql-replica-01"                   # 对应原 `source_id`
from:
  host: "172.16.10.81"                          # 对应原 `mysql_host`
  port: 3306                                    # 对应原 `mysql_port`
  user: "root"                                  # 对应原 `mysql_user`
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="  # 对应原 `mysql_password`
```

```
##### 原 dm_worker2 对应的 source 配置，如命名为 source2.yaml
server-id: 102                                  # 对应原 `server_id`
source-id: "mysql-replica-02"                   # 对应原 `source_id`
from:
  host: "172.16.10.82"                          # 对应原 `mysql_host`
  port: 3306                                    # 对应原 `mysql_port`
  user: "root"                                  # 对应原 `mysql_user`
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="  # 对应原 `mysql_password`
```

从 Binary 部署的 v1.0.x 升级

如果 v1.0.x 是使用 Binary 部署的，且对应的 DM-worker 配置如下：

```
log-level = "info"
log-file = "dm-worker.log"
worker-addr = ":8262"

server-id = 101
source-id = "mysql-replica-01"
flavor = "mysql"

[from]
host = "172.16.10.81"
user = "root"
password = "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="
port = 3306
```

则可转换得到如下的一个 source 配置文件：

```
server-id: 101                                  # 对应原 `server-id`
source-id: "mysql-replica-01"                   # 对应原 `source-id`
flavor: "mysql"                                 # 对应原 `flavor`
from:
  host: "172.16.10.81"                          # 对应原 `from.host`
  port: 3306                                    # 对应原 `from.port`
  user: "root"                                  # 对应原 `from.user`
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="  # 对应原 `from.password`
```

数据迁移任务配置文件

对于数据迁移任务配置向导，v2.0+ 基本与 v1.0.x 保持兼容，可直接复制 v1.0.x 的配置。

第 2 步：部署 v2.0+ 集群

> **注意：**
> 如果已有其他可用的 v2.0+ 集群，可跳过此步。

使用TiUP 按所需要节点数部署新的 v2.0+ 集群。

第 3 步：下线 v1.0.x 集群

如果原 v1.0.x 集群是使用 DM-Ansible 部署的，则使用 DM-Ansible 下线 v1.0.x 集群。

如果原 v1.0.x 集群是使用 Binary 部署，则直接停止 DM-worker 与 DM-master 进程。

第 4 步：升级数据迁移任务

1. 使用operate-source 命令将准备 v2.0+ 的配置文件中得到的上游数据库 source 配置加载到 v2.0+ 集群中。
2. 在下游TiDB 中，从 v1.0.x 的数据复制任务对应的增量checkpoint 表中获取对应的全局checkpoint 信息。
    - 假设v1.0.x 的数据迁移配置中未额外指定meta-schema（或指定其值为默认的dm_meta），且对应的任务名为 task_v1，则对应的 checkpoint 信息在下游TiDB 的 `dm_meta`.`task_v1_syncer_checkpoint` 表中。
    - 使用以下 SQL 语句分别获取该数据迁移任务对应的所有上游数据库 source 的全局checkpoint 信息。

```
> SELECT `id`, `binlog_name`, `binlog_pos` FROM `dm_meta`.`task_v1_syncer_checkpoint`
    '→ WHERE `is_global`=1;
+------------------+--------------------------+----------------------+
| id               | binlog_name              | binlog_pos |
+------------------+--------------------------+----------------------+
| mysql-replica-01 | mysql-bin|000001.000123 | 15847       |
| mysql-replica-02 | mysql-bin|000001.000456 | 10485       |
+------------------+--------------------------+----------------------+
```

3. 更新 v1.0.x 的数据迁移任务配置文件以启动新的 v2.0+ 数据迁移任务。
    - 如 v1.0.x 的数据迁移任务配置文件为 task_v1.yaml，则将其复制一份为 task_v2.yaml。
    - 对 task_v2.yaml 进行以下修改：
        - 将 name 修改为一个新的、不存在的名称，如 task_v2
        - 将 task-mode 修改为 incremental
        - 根据 step.2 中获取的全局checkpoint 信息，为各 source 设置增量复制的起始点，如：

```
mysql-instances:
  - source-id: "mysql-replica-01"          # 对应 checkpoint 信息所属的 `id`
    meta:
      binlog-name: "mysql-bin.000123"      # 对应 checkpoint 信息中的 `binlog_name`,
          '→ 但不包含 `|000001` 部分
      binlog-pos: 15847                    # 对应 checkpoint 信息中的 `binlog_pos`

  - source-id: "mysql-replica-02"
    meta:
      binlog-name: "mysql-bin.000456"
      binlog-pos: 10485
```

> 注意：
> 如在 source 配置中启动了 `enable-gtid`，当前需要通过解析 binlog 或relay log 文件获取 binlog position 对应的GTID sets 并在 `meta` 中设置为 `binlog-gtid`。

4. 使用`start-task` 命令以 v2.0+ 的数据迁移任务配置文件启动升级后的数据迁移任务。
5. 使用`query-status` 命令确认数据迁移任务是否运行正常。

如果数据迁移任务运行正常，则表明 DM 升级到 v2.0+ 的操作成功。

10.6.8.2 集群运维工具

10.6.8.2.1 使用WebUI 管理 DM 迁移任务

DM WebUI 是一个TiDB Data Migration (DM) 迁移任务管理界面，方便用户以直观的方式管理大量迁移任务，无需使用dmctl 命令行，简化任务管理的操作步骤。

本文档介绍 DM WebUI 的访问方式、使用前提、各界面的使用场景以及注意事项。

> 警告：
>
> - DM WebUI 当前为实验特性，不建议在生产环境中使用。
> - DM WebUI 中 `task` 的生命周期有所改变，不建议与dmctl 同时使用。

DM WebUI 主要包含以下界面：

- 数据迁移
  - 任务列表：提供创建迁移任务的界面入口，并展示各迁移任务的详细信息。帮助用户监控、创建、删除、配置迁移任务。
  - 上游配置：用户在此页面配置同步任务的上游数据源信息。管理数据迁移环境中的上游配置，包含了，新增、删除上游配置、监控上游配置对应的同步任务状态、修改上游配置等相关的管理功能。
  - 同步详情：展示更加详细的任务状态信息。根据用户指定的筛选条件查看同步任务的具体配置和状态信息，包括上下游配置信息，上下游数据库名称，源表和目标表的关系等。
- 集群管理
  - 成员列表：展示 DM 集群中所有的master 和 worker 节点，以及 worker 节点与 source 的绑定关系。查看当前 DM 集群的配置信息和各个 worker 的状态信息，并提供基本的管理功能。

界面示例如下：

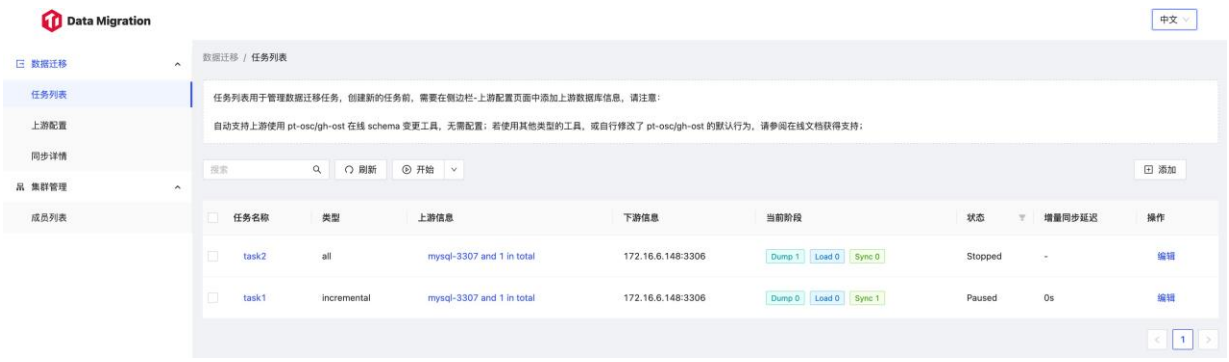


图 183: webui

访问方式

在开启OpenAPI 后，你可以从 DM 集群的任意master 节点访问 DM WebUI，访问端口与 DM OpenAPI 保持一致，默认为 8261。访问地址示例：`http://{master_ip}:{master_port}/dashboard/`。

数据迁移

数据迁移包含上游配置、任务列表、同步详情三个界面。

上游配置

创建迁移任务之前，你需要先创建同步任务的上游数据源信息。你可在上游配置页面创建上游任务的配置。创建时，请注意以下事项：

- 如果存在主从切换，请务必在上游 MySQL 开启GTID，并在创建上游配置时将GTID 设为 `True`，否则数据迁移任务将在主从切换时中断（AWS Aurora 除外）。
- 若某个上游数据库需要临时下线，可将其“停用”，但停用期间其他正在同步的 MySQL 实例不可执行 DDL 操作，否则停用的实例被启用后将无法正常同步。
- 当多个迁移任务使用同一个上游时，可能对其造成额外压力。开启relay log 可降低对上游的影响，建议开启relay log。

任务列表

你可通过任务列表界面查看迁移任务详情，并创建迁移任务。

查看迁移任务详情

在任务列表中，点击任务名称，详情页面会从右侧滑出。详情页面展示了更加详细的任务状态信息。在信息详情页面，你可以查看每一个子任务的运行情况，以及此迁移任务当前完整的配置项信息。

在 DM 中，迁移任务中的每一个子任务可能处于不同的阶段，即全量导出(dump) -> 全量导入(load) -> 增量同步(sync)。因此任务的当前阶段以子任务所处阶段的统计信息来展示，可以更加清楚的了解任务运行情况。

创建迁移任务

要在该界面创建任务，点击右上角的添加按钮即可。创建迁移任务时，你可以使用以下任一方式：

- 通过向导方式。通过WebUI 根据指引一步步填写所需信息进行任务创建，此种方式比较适合入门级用户及日常使用。

- 通过配置文件。通过直接粘贴或编写JSON 格式的任务配置文件进行创建，支持更多的参数调整，适合熟练的用户使用。

同步详情

你可以通过同步详情页面查看迁移任务中所配置迁移规则的运行情况。同步详情页面支持根据任务、数据源、表库名称进行查询。

查询结果中包含上游表至下游表的对应信息，因此请慎重使用 .* 等，以防止查询结果过多导致页面反应迟缓。

集群管理

成员列表

成员列表页面展示 DM 集群中所有的master 和 worker 节点，以及 worker 节点与 source 的绑定关系。

10.6.8.2.2 使用 dmctl 运维 TiDB Data Migration 集群

> 注意：
> 对于用 TiUP 部署的 DM 集群，推荐直接使用 `tiup dmctl` 命令。

dmctl 是用来运维 DM 集群的命令行工具，支持交互模式和命令模式。

dmctl 交互模式

进入交互模式，与 DM-master 进行交互：

> 注意：
> 交互模式下不具有bash 的特性，比如不需要通过引号传递字符串参数而应当直接传递。

```
./dmctl --master-addr 172.16.30.14:8261
```

```
Welcome to dmctl
Release Version: ${version}
Git Commit Hash: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Git Branch: release-x.x
UTC Build Time: yyyy-mm-dd hh:mm:ss
Go Version: go version gox.xx linux/amd64

» help
DM control
```

```
Usage:
  dmctl [command]

Available Commands:
  binlog           manage or show binlog operations
  binlog-schema    manage or show table schema in schema tracker
  check-task       Checks the configuration file of the task
  config           manage config operations
  decrypt      Decrypts cipher text to plain text
  encrypt        Encrypts plain text to cipher text
  help             Gets help about any command
  list-member      Lists member information
  offline-member  Offlines member which has been closed
  operate-leader  `evict`/`cancel-evict` the leader
  operate-source  `create`/`stop`/`show` upstream MySQL/MariaDB source
  pause-relay      Pauses DM-worker's relay unit
  pause-task       Pauses a specified running task or all (sub)tasks bound to a source
  purge-relay      Purges relay log files of the DM-worker according to the specified filename
  query-status     Queries task status
  resume-relay     Resumes DM-worker's relay unit
  resume-task      Resumes a specified paused task or all (sub)tasks bound to a source
  shard-ddl-lock  maintain or show shard-ddl locks information
  start-relay      Starts workers pulling relay log for a source
  start-task       Starts a task as defined in the configuration file
  stop-relay       Stops workers pulling relay log for a source
  stop-task        Stops a specified task or all (sub)tasks bound to a source
  transfer-source Transfers a upstream MySQL/MariaDB source to a free worker

Flags:
  -h, --help              help for dmctl
  -s, --source strings    MySQL Source ID.

Use "dmctl [command] --help" for more information about a command.
```

dmctl 命令模式

命令模式跟交互模式的区别是，执行命令时只需要在dmctl 命令后紧接着执行任务操作，任务操作同交互模式的参数一致。

> 注意：
>
> - 一条dmctl 命令只能跟一个任务操作
> - 从 v2.0.4 版本开始，支持从环境变量(DM_MASTER_ADDR) 里读取 `-master-addr` 参数

```
./dmctl --master-addr 172.16.30.14:8261 start-task task.yaml
./dmctl --master-addr 172.16.30.14:8261 stop-task task
./dmctl --master-addr 172.16.30.14:8261 query-status

export DM_MASTER_ADDR="172.16.30.14:8261"
./dmctl query-status
```

```
Available Commands:
  binlog          manage or show binlog operations
  binlog-schema   manage or show table schema in schema tracker
  check-task      Checks the configuration file of the task
  config          manage config operations
  decrypt      Decrypts cipher text to plain text
  encrypt        Encrypts plain text to cipher text
  help            Gets help about any command
  list-member     Lists member information
  offline-member  Offlines member which has been closed
  operate-leader  `evict`/`cancel-evict` the leader
  operate-source  `create`/`stop`/`show` upstream MySQL/MariaDB source
  pause-relay     Pauses DM-worker's relay unit
  pause-task      Pauses a specified running task or all (sub)tasks bound to a source
  purge-relay     Purges relay log files of the DM-worker according to the specified filename
  query-status    Queries task status
  resume-relay    Resumes DM-worker's relay unit
  resume-task     Resumes a specified paused task or all (sub)tasks bound to a source
  shard-ddl-lock  maintain or show shard-ddl locks information
  start-relay     Starts workers pulling relay log for a source
  start-task      Starts a task as defined in the configuration file
  stop-relay      Stops workers pulling relay log for a source
  stop-task       Stops a specified task or all (sub)tasks bound to a source
  transfer-source Transfers a upstream MySQL/MariaDB source to a free worker

Flags:
      --config string        Path to config file.
  -h, --help                 help for dmctl
      --master-addr string   Master API server address, this parameter is required when
          '→ interacting with the dm-master



      --rpc-timeout string   RPC timeout, default is 10m. (default "10m")
  -s, --source strings       MySQL Source ID.
      --ssl-ca string        Path of file that contains list of trusted SSL CAs for connection.
      --ssl-cert string      Path of file that contains X509 certificate in PEM format for
          '→ connection.
      --ssl-key string       Path of file that contains X509 key in PEM format for connection.
  -V, --version              Prints version and exit.
```

```
Use "dmctl [command] --help" for more information about a command.
```

10.6.8.3 性能调优

10.6.8.3.1 DM 5.4.0 性能测试报告

TiDB Data Migration (DM) 5.4.0 的性能测试数据与 DM 5.3.0 的测试数据持平，因此不单独提供测试报告。若要了解 DM 5.3.0 版本的性能测试目的、环境、场景和结果，请参阅 DM 5.3.0 性能测试报告。

10.6.8.3.2 DM 配置优化

本文档介绍如何对迁移任务的配置进行优化，从而提高 DM 的数据迁移性能。

全量导出

全量导出相关的配置项为 `mydumpers`，下面介绍和性能相关的参数如何配置。

`rows`

设置 `rows` 选项可以开启单表多线程并发导出，值为导出的每个chunk 包含的最大行数。开启后，DM 会在 MySQL 的单表并发导出时，优先选出一列做拆分基准，选择的优先级为主键> 唯一索引> 普通索引，选出目标列后需保证该列为整数类型（如 `INT`、`MEDIUMINT`、`BIGINT` 等）。

`rows` 的值可以设置为 10000，具体设置的值可以根据表中包含数据的总行数以及数据库的性能做调整。另外也需要设置 `threads` 来控制并发线程数量，默认值为 4，可以适当做些调整。

`chunk-filesize`

DM 全量备份时会根据 `chunk-filesize` 参数的值把每个表的数据划分成多个chunk，每个chunk 保存到一个文件中，大小约为 `chunk-filesize`。根据这个参数把数据切分到多个文件中，这样就可以利用 DM Load 处理单元的并行处理逻辑提高导入速度。该参数默认值为 64（单位为 MB），正常情况下不需要设置，也可以根据全量数据的大小做适当的调整。

> 注意：
>
> - `mydumpers` 的参数值不支持在迁移任务创建后更新，所以需要在创建任务前确定好各个参数的值。如果需要更新，则需要使用dmctl stop 任务后更新配置文件，然后再重新创建任务。
> - `mydumpers.threads` 可以使用配置项 `mydumper-thread` 替代来简化配置。
> - 如果设置了 `rows`，DM 会忽略 `chunk-filesize` 的值。

全量导入

全量导入相关的配置项为 `loaders`，下面介绍和性能相关的参数如何配置。

`pool-size`

pool-size 为 DM Load 阶段线程数量的设置，默认值为 16，正常情况下不需要设置，也可以根据全量数据的大小以及数据库的性能做适当的调整。

> 注意：
>
> - loaders 的参数值不支持在迁移任务创建后更新，所以需要在创建任务前确定好各个参数的值。如果需要更新，则需要使用dmctl stop 任务后更新配置文件，然后再重新创建任务。
> - loaders.pool-size 可以使用配置项 loader-thread 替代来简化配置。

增量复制

增量复制相关的配置为 syncers，下面介绍和性能相关的参数如何配置。

worker-count

worker-count 为 DM Sync 阶段并发迁移 DML 的线程数量设置，默认值为 16，如果对迁移速度有较高的要求，可以适当调高改参数的值。

batch

batch 为 DM Sync 阶段迁移数据到下游数据库时，每个事务包含的 DML 的数量，默认值为 100，正常情况下不需要调整。

> 注意：
>
> - syncers 的参数值不支持在迁移任务创建后更新，所以需要在创建任务前确定好各个参数的值。如果需要更新，则需要使用dmctl stop 任务后更新配置文件，然后再重新创建任务。
> - syncers.worker-count 可以使用配置项 syncer-thread 替代来简化配置。
> - worker-count 和 batch 的设置需要根据实际的场景进行调整，例如：DM 到下游数据库的网络延迟较高，可以适当调高 worker-count，调低 batch。

10.6.8.3.3 DM 集群性能测试

本文档介绍如何构建测试场景对 DM 集群进行性能测试，包括数据迁移速度、延迟等。

迁移数据流

可以使用简单的迁移数据流来测试 DM 集群的数据迁移性能，即单个 MySQL 实例到TiDB 的数据迁移：MySQL -> DM -> TiDB。

部署测试环境

- 使用TiUP 部署TiDB 测试集群，所有配置使用TiUP 提供的默认配置。

- 部署 MySQL 服务，开启 ROW 模式binlog，其他配置项使用默认配置。
- 部署 DM 集群，部署一个 DM-worker 和一个 DM-master 即可。

性能测试

迁移数据表结构

使用如下结构的表进行性能测试：

```
CREATE TABLE `sbtest` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `k` int(11) NOT NULL DEFAULT '0',
  `c` char(120) CHARSET utf8mb4 COLLATE utf8mb4_bin NOT NULL DEFAULT '',
  `pad` char(60) CHARSET utf8mb4 COLLATE utf8mb4_bin NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `k_1` (`k`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

全量导入性能测试用例

生成测试数据

使用 sysbench 在上游创建测试表，并生成全量导入的测试数据。sysbench 生成数据的命令如下所示：

```
sysbench --test=oltp_insert --tables=4 --mysql-host=172.16.4.40 --mysql-port=3306 --mysql-user=
    ↪ root --mysql-db=dm_benchmark --db-driver=mysql --table-size=50000000 prepare
```

创建数据迁移任务

1. 创建上游 MySQL 的 source，将 source-id 配置为 source-1。详细操作方法参考：加载数据源配置。
2. 创建 full 模式的 DM 迁移任务，示例任务配置文件如下：

“ ‘yaml

name: test-full
task-mode: full

使用实际测试环境中TiDB 的信息配置target-database: host: “192.168.0.1” port: 4000 user: “root” password: “”

mysql-instances: - source-id: “source-1” block-allow-list: “instance” # 如果 DM 版本早于 v2.0.0-beta.2 则使用black-white-list mydumper-config-name: “global” loader-thread: 16

配置 sysbench 生成数据所在的库的名称 block-allow-list: # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list instance: do-dbs: [“dm_benchmark”]

mydumpers: global: rows: 32000 threads: 32 “ ‘

创建数据迁移任务的详细操作参考创建数据迁移任务。

注意：

- 在 mydumpers 配置项中使用 rows 选项，可以开启单表多线程并发导出，加快数据导出速度。
- mysql-instances 配置中的 loader-thread 以及 mydumpers 配置项中的 rows 和 threads 可以做适当调整，测试在不同配置下对性能的影响。

获取测试结果

观察 DM-worker 日志，当出现 all data files have been finished 时，表示全量数据导入完成，此时可以看到消耗时间。示例日志如下：

```
[INFO] [loader.go:604] ["all data files have been finished"] [task=test] [unit=load] ["cost time
    ↪ "=52.439796ms]
```

根据测试数据的数据量和导入消耗时间，可以算出全量数据的迁移速度。

增量复制性能测试用例

初始化表

使用 sysbench 在上游创建测试表。

创建数据迁移任务

1. 创建上游 MySQL 的 source, source-id 配置为 source-1（如果在全量迁移性能测试中已经创建，则不需要再次创建）。详细操作方法参考：加载数据源配置。
2. 创建 all 模式的 DM 迁移任务，示例任务配置文件如下：

“‘yaml

name: test-all
task-mode: all

使用实际测试环境中TiDB 的信息配置target-database: host: “192.168.0.1” port: 4000 user: “root” password: “”

mysql-instances: - source-id: “source-1” block-allow-list: “instance” # 如果 DM 版本早于 v2.0.0-beta.2 则使用black-white-list syncer-config-name: “global”

配置 sysbench 生成数据所在的库的名称 block-allow-list: # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list instance: do-dbs: [“dm_benchmark”]

syncers: global: worker-count: 16 batch: 100 “‘

创建数据迁移任务的详细操作参考创建数据迁移任务。

> **注意：**
>
> syncers 配置项中的 worker-count 和 batch 可以做适当调整，测试在不同配置下性能的差异。

生成增量数据

执行 sysbench 命令在上游持续生成增量数据：

```
sysbench --test=oltp_insert --tables=4 --num-threads=32 --mysql-host=172.17.4.40 --mysql-port
    ′→ =3306 --mysql-user=root --mysql-db=dm_benchmark --db-driver=mysql --report-interval=10 --
    ′→ time=1800 run
```

> **注意：**
>
> 可以通过调整 sysbench 的语句类型，测试在不同业务场景下 DM 的数据迁移性能。

获取测试结果

通过 query-status 命令观测 DM 的迁移状态，通过Grafana 观测 DM 的监控指标。主要包括单位时间内完成的 job 数量 finished sqls jobs 等，详细的监控指标说明参考Binlog replication 监控指标。

10.6.8.3.4 TiDB Data Migration 性能问题及处理方法

本文档介绍TiDB Data Migration (DM) 中可能存在的、常见的性能问题及其处理方法。

在诊断与处理性能问题时，请确保已经正确配置并安装 DM 的监控组件，并能在Grafana 监控面板查看DM 的监控指标。

在诊断性能问题时，请先确保对应组件正在正常运行，否则可能出现监控指标异常的情况，对性能问题的诊断造成干扰。

在诊断问题前，也可以先了解 DM 的性能测试报告。

当数据迁移过程存在较大延迟时，若需快速定位瓶颈是在 DM 组件内部还是在TiDB 集群，可先排查写入 SQL 到下游部分的 DML queue remain length。

relay log 模块的性能问题及处理方法

在relay log 的监控部分，可以主要通过 binlog file gap between master and relay 监控项确认是否存在性能问题。如果该指标长时间大于 1，通常表明存在性能问题；如果该指标基本为 0，一般表明没有性能问题。

如果 binlog file gap between master and relay 基本为 0，但仍怀疑存在性能问题，则可以继续查看 binlog ′→ pos，如果该指标中 master 远大于 relay，则表明可能存在性能问题。

如果存在性能问题，则继续根据relay log 模块的主要处理流程分别进行诊断与处理。

读取binlog 数据

与 relay log 模块从上游读取binlog 数据相关的主要性能指标是 `read binlog event duration`，该指标表示从上游 MySQL/MariaDB 读取到单个binlog event 所需要的时间，理想情况下应接近于 DM-worker 与 MySQL/MariaDB 实例间的网络延迟。

对于同机房的数据迁移，这部分一般不会成为性能瓶颈；如果该值过大，请排查 DM-worker 与 MySQL/MariaDB 间的网络连通情况。

对于跨机房的数据迁移，可尝试将 DM-worker 与 MySQL/MariaDB 部署在同一机房，而仍将TiDB 集群部署在目标机房。

从上游读取binlog 数据这一流程细分后包括以下三个子流程：

- 上游 MySQL/MariaDB 从本地读取binlog 数据并通过网络进行发送。上游 MySQL/MariaDB 负载无异常时，该子流程通常不会成为瓶颈。
- binlog 数据通过网络从 MySQL/MariaDB 所在机器传输到 DM-worker 所在机器。该子流程主要由 DM-worker 与上游 MySQL/MariaDB 的网络连通情况决定。
- DM-worker 从网络数据流中读取binlog 数据，并构造成binlog event。当 DM-worker 负载无异常时，该子流程通常不会成为瓶颈。

> 注意：
>
> 如果 `read binlog event duration` 的值较大，另一个可能的原因是上游 MySQL/MariaDB 负载较低，一段时间内暂时没有需要发送给 DM 的 binlog event，relay log 模块处于等待状态，导致该值包含了额外的等待时间。

binlog 数据解码与验证

将 binlog event 读取到 DM 内存后，会进行必要的数据解码与验证，这部分通常不会存在性能瓶颈，因此监控面板上默认无对应性能指标。如果需要查看相应指标，可手动为 Prometheus 中的 `dm_relay_read_transform_duration` 添加相应的监控。

写入relay log 文件

在将binlog event 写入relay log 文件时，相关的主要性能指标是 `write relay log duration`，该指标在 `binlog event size` 不是特别大时，值应在微秒级别。如果该值过大，需排查磁盘写入性能，如尽量优先为 DM-worker 使用本地 SSD 等。

Load 模块的性能问题及处理方法

Load 模块主要操作为从本地读取 SQL 文件数据并写入到下游，对应的主要性能指标是 `transaction execution latency`，如果该值过大，则通常需要根据下游数据库的监控对下游性能进行排查。

另外，也可以查看是否 DM 到下游数据库间的网络存在较大的延迟。

Binlog replication 模块的性能问题及处理方法

在Binlog replication 的监控部分，可以主要通过 `binlog file gap between master and syncer` 监控项确认是否存在性能问题，如果该指标长时间大于 1，则通常表明存在性能问题；如果该指标基本为 0，则一般表明没有性能问题。

如果 binlog file gap between master and syncer 长时间大于 1，则可以再通过 binlog file gap between relay and syncer 判断延迟主要存在于哪个模块，如果该值基本为 0，则延迟可能存在于 relay log 模块，请先参考relay log 模块的性能问题及处理方法进行处理；否则继续对 Binlog replication 进行排查。

读取 binlog 数据

Binlog replication 模块会根据配置选择从上游 MySQL/MariaDB 或 relay log 文件中读取binlog event，对应的主要性能指标是 read binlog event duration，该值的范围一般是几微秒至几十微秒。

- 如果是从上游 MySQL/MariaDB 读取binlog event，则可参考 relay log 模块下的读取 binlog 数据进行排查与处理。
- 如果是从relay log 文件中读取，则在 binlog event size 不是特别大时，read binlog event duration 的值应在微秒级别。如果 read binlog event duration 过大，则需排查磁盘读取性能，如尽量优先为 DM-worker 使用本地 SSD 等。

binlog event 转换

Binlog replication 模块从binlog event 数据中尝试构造 DML、解析 DDL 以及进行table router 转换等，主要的性能指标是 transform binlog event duration。

这部分的耗时受上游写入的业务特点影响较大，如对于 INSERT INTO 语句，转换单个 VALUES 的时间和转换大量 VALUES 的时间差距很多，其波动范围可能从几十微秒至上百微秒，但一般不会成为系统的瓶颈。

写入 SQL 到下游

Binlog replication 模块将转换后的 SQL 写入到下游时，涉及到的性能指标主要包括 DML queue remain length 与 transaction execution latency。

DM 在从 binlog event 构造出 SQL 后，会使用 worker-count 个队列尝试并发写入到下游。但为了避免监控条目过多，会将并发队列编号按 8 取模，即所有并发队列在监控上会对应到 q_0 到 q_7 的某一项。

DML queue remain length 用于表示并发处理队列中尚未取出并开始用于向下游写入的 DML 语句数，理想情况下，各 q_* 对应的曲线应该基本一致，如果极不一致则表明并发的负载极不均衡。

如果负载不均衡，请确认需要迁移的所有表结构中都有主键或唯一键，如没有主键或唯一键则请尝试为其添加主键或唯一键；如果存在主键或唯一键时仍存在该问题，可尝试升级 DM 到 v1.0.5 及以上的版本。

当整个数据迁移链路无明显延迟时，DML queue remain length 对应曲线应基本为 0，且最大通常应不超过任务配置文件中的 batch 值。

如果确认数据迁移链路存在明显延迟，且 DML queue remain length 中各 q_* 对应的曲线基本一致且基本为 0，则表明 DM 未能及时地从上游读取数据、进行转换或进行并行分发（如瓶颈存在于relay log 模块等），请参考本文档前述各节进行排查。

如果 DML queue remain length 对应曲线不为 0（最大一般不超过 1024），则通常表明向下游写入 SQL 时存在瓶颈，可通过 transaction execution latency 查看向下游执行单个事务的耗时情况。

transaction execution latency 一般应在几十毫秒。如果该值过高，则通常需要根据下游数据库的监控对下游性能进行排查，另外也可以关注是否 DM 到下游数据库间的网络存在较大的延迟。

此外，也可通过 statement execution latency 查看向下游写入 BEGIN、INSERT/UPDATE/DELETE、COMMIT 等单条语句的耗时情况。

10.6.8.4 数据源管理

10.6.8.4.1 切换 DM-worker 与上游 MySQL 实例的连接

当需要对 DM-worker 所连接的上游 MySQL 实例进行停机维护或该实例意外宕机时，需要将 DM-worker 的连接切换到同一个主从复制集群内的另一个 MySQL 实例上。本文介绍如何将 DM-worker 的连接从一个 MySQL 实例切换到另一个 MySQL 实例上。

> 注意：
>
> - 仅支持在同一个主从复制集内的 MySQL 实例间进行切换。
> - 将要切换到的 MySQL 实例必须拥有 DM-worker 所需的binlog。
> - DM-worker 必须以GTID sets 模式运行，即对应 source 配置文件中指定 `enable-gtid: true`。
> - DM 仅支持以下两种切换场景，且必须严格按照各场景的步骤执行操作，否则可能需要根据切换后的 MySQL 实例重新搭建 DM 集群并完整重做数据迁移任务。

有关GTID sets 的概念解释，请参考 MySQL 文档。

虚拟IP 环境下切换 DM-worker 与 MySQL 实例的连接

如果 DM-worker 通过虚拟IP (VIP) 连接上游的 MySQL 实例，更改 VIP 所指向的 MySQL 实例，即是在 DM-worker 对应上游连接地址不变的情况下切换 DM-worker 实际所连接的 MySQL 实例。

> 注意：
>
> 如果不对 DM 执行必要变更，当切换VIP 所指向的 MySQL 实例时，DM 内部不同的connection 可能会同时连接到切换前后不同的 MySQL 实例，造成 DM 拉取的binlog 与从上游获取到的其他状态不一致，从而导致难以预期的异常行为甚至数据损坏。

如果 DM-worker 连接的VIP 需要指向新的 MySQL 实例，需要按以下步骤进行操作：

1. 使用 `query-status` 命令获取当前 binlog replication 处理单元已复制到下游的 binlog 对应的 GTID sets (`syncerBinlogGtid`)，记为 `gtid-S`。
2. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_purged;` 获取已经被purged 的 binlog 对应的 GTID sets，记为 `gtid-P`。
3. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_executed;` 获取所有已经执行成功的事务对应的GTID sets，记为 `gtid-E`。
4. 确保满足以下关系，否则不支持将 DM-worker 连接切换到相应的 MySQL 实例：
    - `gtid-S` 包含 `gtid-P`（`gtid-P` 可以为空）
    - `gtid-E` 包含 `gtid-S`
5. 使用 `pause-task` 命令暂停所有运行中的数据迁移任务。

6. 变更VIP 以指向新的 MySQL 实例。
7. 使用 `resume-task` 命令恢复之前的数据迁移任务。

变更 DM-worker 连接的上游 MySQL 实例地址

若要变更 DM-worker 的配置信息来使 DM-worker 连接到新的上游 MySQL 实例，需要按以下步骤进行操作：

1. 使用 `query-status` 命令获取当前 binlog replication 处理单元已复制到下游的 binlog 对应的 GTID sets (`syncerBinlogGtid`)，记为 `gtid-S`。
2. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_purged;` 获取已经被purged 的 binlog 对应的 GTID sets，记为 `gtid-P`。
3. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_executed;` 获取所有已经执行成功的事务对应的GTID sets，记为 `gtid-E`。
4. 确保满足以下关系，否则不支持将 DM-worker 连接切换到相应的 MySQL 实例：
    - `gtid-S` 包含 `gtid-P`（`gtid-P` 可以为空）
    - `gtid-E` 包含 `gtid-S`
5. 使用 `stop-task` 命令停止所有运行中的数据迁移任务。
6. 使用 `operate-source stop` 命令从 DM 集群中移除原 MySQL 实例地址对应的 source 配置。
7. 更新 source 配置文件中的 MySQL 实例地址并使用 `operate-source create` 命令将新的 source 配置重新加载到 DM 集群中。
8. 使用 `start-task` 命令重新启动数据迁移任务。

10.6.8.5 任务管理

10.6.8.5.1 使用TiDB Data Migration 处理出错的 DDL 语句

本文介绍了如何使用TiDB Data Migration (DM) 来处理出错的 DDL 语句。

目前，TiDB 并不完全兼容所有的 MySQL 语法（详见DDL 的限制）。当使用 DM 从 MySQL 迁移数据到TiDB 时，如果 TiDB 不支持对应的 DDL 语句，可能会造成错误并中断迁移任务。在这种情况下，DM 提供 `binlog` 命令来恢复迁移。

使用限制

如果业务不能接受下游TiDB 跳过异常的 DDL 语句，也不接受使用其他 DDL 语句作为替代，不接受插入其他 DDL 语句，则不适合使用此方式进行处理。

比如：`DROP PRIMARY KEY`，这种情况下，只能在下游重建一个（DDL 执行完后的）新表结构对应的表，并将原表的全部数据重新导入该新表。

支持场景

迁移过程中，上游执行了TiDB 不支持的 DDL 语句并迁移到了 DM，造成迁移任务中断。

- 如果业务能接受下游TiDB 不执行该 DDL 语句，则使用 `binlog skip <task-name>` 跳过对该 DDL 语句的迁移以恢复迁移任务。
- 如果业务能接受下游TiDB 执行其他 DDL 语句来作为替代，则使用 `binlog replace <task-name>` 替代该 DDL 的迁移以恢复迁移任务。

- 如果业务能接受下游TiDB 插入执行其他 DDL 语句，则使用 `binlog inject <task-name>` 插入其他 DDL 以恢复迁移任务。

命令介绍

使用dmctl 手动处理出错的 DDL 语句时，主要使用的命令包括 `query-status`、`binlog`。

query-status

`query-status` 命令用于查询当前 MySQL 实例内子任务及relay 单元等的状态和错误信息，详见查询状态。

binlog

`binlog` 命令管理和查看binlog 操作。命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 `handle-error` 命令。

`binlog` 命令用法如下：

```
binlog -h
```

```
manage or show binlog operations

Usage:
  dmctl binlog [command]

Available Commands:
  inject      inject the current error event or a specific binlog position (binlog-pos) with some
      '→  ddls
  list        list error handle command at binlog position (binlog-pos) or after binlog position
      '→ (binlog-pos)
  replace     replace the current error event or a specific binlog position (binlog-pos) with
      '→ some ddls
  revert      revert the current binlog operation or a specific binlog position (binlog-pos)
      '→ operation
  skip        skip the current error event or a specific binlog position (binlog-pos) event

Flags:
  -b, --binlog-pos string    position used to match binlog event if matched the binlog operation
      '→ will be applied. The format like "mysql-bin|000001.000003:3270"
  -h, --help                 help for binlog

Global Flags:
  -s, --source strings    MySQL Source ID.

Use "dmctl binlog [command] --help" for more information about a command.
```

`binlog` 支持如下子命令：

- `inject`：在 DDL binlog 位置插入 DDL 语句，binlog 位置指定方式参考 `-b, --binlog-pos`。

- `list`：查看 binlog 位置以及此位置之后的有效 inject/skip/replace 操作，binlog 位置指定方式参考 `-b, --binlog-pos`。
- `replace`：替代 DDL binlog 位置的 DDL 语句，binlog 位置指定方式参考 `-b, --binlog-pos`。
- `revert`：重置 binlog 位置的inject/skip/replace 操作，仅在先前的操作没有最终生效前执行，binlog 位置指定方式参考 `-b, --binlog-pos`
- `skip`：跳过 binlog 位置的 DDL 语句, binlog 位置指定方式参考 `-b, --binlog-pos`。

`binlog` 支持如下参数：

- `-b, --binlog-pos`：
  - 类型：string。
  - 指定binlog 位置，表示操作将在 `binlog-pos` 与 binlog event 的 position 匹配时生效。若不指定，DM 会默认置为当前出错的 DDL 语句的binlog 位置。
  - 格式：`binlog-filename:binlog-pos`，例如 `mysql-bin|000001.000003:3270`。
  - 在迁移执行出错后，binlog position 可直接从 `query-status` 返回的 `startLocation` 中的 `position` 获得；在迁移执行出错前，binlog position 可在上游 MySQL 中使用 `SHOW BINLOG EVENTS` 获得。
- `-s, --source strings`：
  - 类型：string。
  - `source` 指定预设操作将生效的 MySQL 实例。
- 其他参数参考 `-h` 提示。

使用示例

迁移中断执行跳过操作

如需在迁移中断时执行跳过操作，可使用 `binlog skip` 命令：

```
binlog skip -h
```

```
skip the current error event or a specific binlog position (binlog-pos) event

Usage:
  dmctl binlog skip <task-name> [flags]

Flags:
  -h, --help    help for skip

Global Flags:
  -b, --binlog-pos string    position used to match binlog event if matched the binlog operation
      ⟶ will be applied. The format like "mysql-bin|000001.000003:3270"
  -s, --source strings       MySQL Source ID.
```

非合库合表场景

假设现在需要将上游的 `db1.tbl1` 表迁移到下游TiDB，初始时表结构为：

```
SHOW CREATE TABLE db1.tbl1;
```

```
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| tbl1  | CREATE TABLE `tbl1` (
  `c1` int(11) NOT NULL,
  `c2` decimal(11,3) DEFAULT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-------+-----------------------------------------------------+
```

此时，上游执行以下 DDL 操作修改表结构（将列的 DECIMAL(11, 3) 修改为 DECIMAL(10, 3)）：

```
ALTER TABLE db1.tbl1 CHANGE c2 c2 DECIMAL (10, 3);
```

则会由于TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 query-status <task-name> 命令可看到如下错误：

```
ERROR 8200 (HY000): Unsupported modify column: can't change decimal column precision
```

假设业务上可以接受下游TiDB 不执行此 DDL 语句（即继续保持原有的表结构），则可以通过使用 binlog skip <task-name> 命令跳过该 DDL 语句以恢复迁移任务。操作步骤如下：

1. 使用 binlog skip <task-name> 跳过当前错误的 DDL 语句

```
» binlog skip test
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

2. 使用 query-status <task-name> 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Sync",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "sync": {
                        "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                        "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                        "blockingDDLs": [
                        ],
                        "unresolvedGroups": [
                        ],
                        "synced": true,
                        "binlogType": "remote",
                        "totalRows": "4",
                        "totalRps": "0",
                        "recentRps": "0"
                    }
                }
            ]
        }
    ]
}
```


可以看到任务运行正常，错误的 DDL 被跳过。

合库合表场景

假设现在存在如下四个上游表需要合并迁移到下游的同一个表 `shard_db`.`shard_table`，任务模式为悲观协调模式：

- MySQL 实例 1 内有 shard_db_1 库，包括 shard_table_1 和 shard_table_2 两个表。
- MySQL 实例 2 内有 shard_db_2 库，包括 shard_table_1 和 shard_table_2 两个表。

初始时表结构为：

```
SHOW CREATE TABLE shard_db.shard_table;
```

```
+       +------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+       +
 --------
    ↪
| tb    | CREATE TABLE `shard_table` (
  `id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+       +
        ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
```

此时，在上游所有分表上都执行以下 DDL 操作修改表字符集

```
ALTER TABLE `shard_db_*`.`shard_table_*` CHARACTER SET LATIN1 COLLATE LATIN1_DANISH_CI;
```

则会由于TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 query-status 命令可以看到 MySQL 实例 1 的 shard_db_1.shard_table_1 表和 MySQL 实例 2 的 shard_db_2.shard_table_1 表报错：

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_1`.`shard_table_1` CHARACTER SET UTF8
        ↪ COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_2`.`shard_table_1` CHARACTER SET UTF8
        ↪ COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

假设业务上可以接受下游TiDB 不执行此 DDL 语句（即继续保持原有的表结构），则可以通过使用 binlog skip ↪ <task-name> 命令跳过该 DDL 语句以恢复迁移任务。操作步骤如下：

1. 使用 binlog skip <task-name> 跳过 MySQL 实例 1 和实例 2 当前错误的 DDL 语句

```
» binlog skip test
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

2. 使用 `query-status <task-name>` 查看任务状态，可以看到 MySQL 实例 1 的 `shard_db_1.shard_table_2` 表和 MySQL 实例 2 的 `shard_db_2.shard_table_2` 表报错：

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_1`.`shard_table_2` CHARACTER SET
        ↪ UTF8 COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_2`.`shard_table_2` CHARACTER SET
        ↪ UTF8 COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

3. 继续使用 `binlog skip <task-name>` 跳过 MySQL 实例 1 和实例 2 当前错误的 DDL 语句

```
» binlog skip test
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
```

```
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

4. 使用 `query-status <task-name>` 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Sync",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "sync": {
                        "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                        "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                        "blockingDDLs": [
```

```
                ],
                "unresolvedGroups": [
                ],
                "synced": true,
                "binlogType": "remote",
                "totalRows": "4",
                "totalRps": "0",
                "recentRps": "0"
            }
        }
    ]
},
{
    "result": true,
    "msg": "",
    "sourceStatus": {
        "source": "mysql-replica-02",
        "worker": "worker2", "result":
        null, "relayStatus": null
    },
    "subTaskStatus": [
        {
            "name": "test",
            "stage": "Running",
            "unit": "Sync",
            "result": null,
            "unresolvedDDLLockID": "",
            "sync": {
                "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                "blockingDDLs": [
                ],
                "unresolvedGroups": [
                ],
                "synced": true,
                "binlogType": "remote",
                "totalRows": "4",
                "totalRps": "0",
                "recentRps": "0"
            }
        }
    ]
```

```
        }
    ]
}
```

可以看到任务运行正常，无错误信息。四条 DDL 全部被跳过。

迁移中断执行替代操作

如需在迁移中断时执行替代操作，可使用 `binlog replace` 命令：

```
binlog replace -h
```

```
replace the current error event or a specific binlog position (binlog-pos) with some ddls

Usage:
  dmctl binlog replace <task-name> <replace-sql1> <replace-sql2>... [flags]

Flags:
  -h, --help    help for replace

Global Flags:
  -b, --binlog-pos string    position used to match binlog event if matched the binlog operation
      ↪ will be applied. The format like "mysql-bin|000001.000003:3270"
  -s, --source strings       MySQL Source ID.
```

非合库合表场景

假设现在需要将上游的 `db1.tbl1` 表迁移到下游TiDB，初始时表结构为：

```
SHOW CREATE TABLE db1.tbl1;
```

```
+-------+--------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+--------------------------------------------------------------------------------------------------------
    ↪
| tb    | CREATE TABLE `tbl1` (
  `id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+-------+--------------------------------------------------------------------------------------------------------
    ↪
```

此时，上游执行以下 DDL 操作增加新列，并添加 UNIQUE 约束

```
ALTER TABLE `db1`.`tbl1` ADD COLUMN new_col INT UNIQUE;
```

则会由于TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 query-status 命令可看到如下错误：

```
{
    "Message": "cannot track DDL: ALTER TABLE `db1`.`tbl1` ADD COLUMN `new_col` INT UNIQUE KEY",
    "RawCause": "[ddl:8200]unsupported add column 'new_col' constraint UNIQUE KEY when altering '
        '→ db1.tbl1'",
}
```

我们将该 DDL 替换成两条等价的 DDL。操作步骤如下：

1. 使用如下命令替换错误的 DDL 语句

```
» binlog replace test "ALTER TABLE `db1`.`tbl1` ADD COLUMN `new_col` INT;ALTER TABLE `db1`.`
    '→ tbl1` ADD UNIQUE(`new_col`)";
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

2. 使用 query-status <task-name> 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
```

```
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Sync",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "sync": {
                        "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                        "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                        "blockingDDLs": [
                        ],
                        "unresolvedGroups": [
                        ],
                        "synced": true,
                        "binlogType": "remote",
                        "totalRows": "4",
                        "totalRps": "0",
                        "recentRps": "0"
                    }
                }
            ]
        }
    ]
}
```

可以看到任务运行正常，错误的 DDL 已被替换且执行成功。

合库合表场景

假设现在存在如下四个上游表需要合并迁移到下游的同一个表 `shard_db`.`shard_table`，任务模式为悲观协调模式：

- MySQL 实例 1 内有 shard_db_1 库，包括 shard_table_1 和 shard_table_2 两个表。
- MySQL 实例 2 内有 shard_db_2 库，包括 shard_table_1 和 shard_table_2 两个表。

初始时表结构为：

```sql
SHOW CREATE TABLE shard_db.shard_table;
```

```
+       +------------------------------------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
```

```
+--------+
    '→
| tb    | CREATE TABLE `shard_table` (
  `id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+       +
        ----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    '→
```

此时，在上游所有分表上都执行以下 DDL 操作增加新列，并添加 UNIQUE 约束：

```
ALTER TABLE `shard_db_*`.`shard_table_*` ADD COLUMN new_col INT UNIQUE;
```

则会由于TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 query-status 命令可以看到 MySQL 实例 1 的 shard_db_1.shard_table_1 表和 MySQL 实例 2 的 shard_db_2.shard_table_1 表报错：

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_1`.`shard_table_1` ADD COLUMN `new_col`
        '→ INT UNIQUE KEY",
    "RawCause": "[ddl:8200]unsupported add column 'new_col' constraint UNIQUE KEY when altering '
        '→ shard_db_1.shard_table_1'",
}
```

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_2`.`shard_table_1` ADD COLUMN `new_col`
        '→ INT UNIQUE KEY",
    "RawCause": "[ddl:8200]unsupported add column 'new_col' constraint UNIQUE KEY when altering '
        '→ shard_db_2.shard_table_1'",
}
```

我们将该 DDL 替换成两条等价的 DDL。操作步骤如下：

1. 使用如下命令分别替换 MySQL 实例 1 和实例 2 中错误的 DDL 语句

```
» binlog replace test -s mysql-replica-01 "ALTER TABLE `shard_db_1`.`shard_table_1` ADD
    '→ COLUMN `new_col` INT;ALTER TABLE `shard_db_1`.`shard_table_1` ADD UNIQUE(`new_col`)"
    '→ ;
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
```

```
            "worker": "worker1"
        }
    ]
}
```

```
» binlog replace test -s mysql-replica-02 "ALTER TABLE `shard_db_2`.`shard_table_1` ADD
    '→ COLUMN `new_col` INT;ALTER TABLE `shard_db_2`.`shard_table_1` ADD UNIQUE(`new_col`)"
    '→ ;
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

2. 使用 query-status <task-name> 查看任务状态，可以看到 MySQL 实例 1 的 shard_db_1.shard_table_2 表和 MySQL 实例 2 的 shard_db_2.shard_table_2 表报错：

```
{
    "Message": "detect inconsistent DDL sequence from source ... ddls: [ALTER TABLE `
        '→ shard_db`.`tb` ADD COLUMN `new_col` INT UNIQUE KEY] source: `shard_db_1`.`
        '→ shard_table_2`], right DDL sequence should be ..."
}
```

```
{
    "Message": "detect inconsistent DDL sequence from source ... ddls: [ALTER TABLE `
        '→ shard_db`.`tb` ADD COLUMN `new_col` INT UNIQUE KEY] source: `shard_db_2`.`
        '→ shard_table_2`], right DDL sequence should be ..."
}
```

3. 使用如下命令继续分别替换 MySQL 实例 1 和实例 2 中错误的 DDL 语句

```
» binlog replace test -s mysql-replica-01 "ALTER TABLE `shard_db_1`.`shard_table_2` ADD
    '→ COLUMN `new_col` INT;ALTER TABLE `shard_db_1`.`shard_table_2` ADD UNIQUE(`new_col`)"
    '→ ;
```

```
{
    "result": true,
    "msg": "",
```

```
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

```
» binlog replace test -s mysql-replica-02 "ALTER TABLE `shard_db_2`.`shard_table_2` ADD
    '→ COLUMN `new_col` INT;ALTER TABLE `shard_db_2`.`shard_table_2` ADD UNIQUE(`new_col`)"
    '→ ;
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

4. 使用 `query-status <task-name>` 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
```

```
        "subTaskStatus": [
            {
                "name": "test",
                "stage": "Running",
                "unit": "Sync",
                "result": null,
                "unresolvedDDLLockID": "",
                "sync": {
                    "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                    "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                    "blockingDDLs": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "synced": true,
                    "binlogType": "remote",
                    "totalRows": "4",
                    "totalRps": "0",
                    "recentRps": "0"
                }
            }
        ]
    },
    {
        "result": true,
        "msg": "",
        "sourceStatus": {
            "source": "mysql-replica-02",
            "worker": "worker2", "result":
            null, "relayStatus": null
        },
        "subTaskStatus": [
            {
                "name": "test",
                "stage": "Running",
                "unit": "Sync",
                "result": null,
                "unresolvedDDLLockID": "",
                "sync": {
                    "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
```

```
                    "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                    "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                    "blockingDDLs": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "synced": try,
                    "binlogType": "remote",
                    "totalRows": "4",
                    "totalRps": "0",
                    "recentRps": "0"
                }
            }
        ]
    }
]
}
```

可以看到任务运行正常，无错误信息。四条 DDL 全部被替换。

其他命令

binlog 其他命令的使用，请参考上述 `binlog skip`、`binlog replace` 命令的使用方式。

10.6.8.5.2 管理TiDB Data Migration 迁移表的表结构

dmctl 是运维TiDB Data Migration (DM) 集群的命令行工具，本文介绍如何使用dmctl 组件来管理通过 DM 迁移的表在 DM 内部的表结构。

DM 执行增量迁移时，首先读取上游的binlog，然后生成 SQL 语句执行到下游。但是，上游的 binlog 中并不记录表的完整结构信息，为了生成 SQL 语句，DM 内部维护了待迁移的表的 schema 信息，即表结构信息。

为了一些特殊场景及处理其他可能的由于 schema 不匹配导致的迁移中断等问题，DM 提供了 `binlog-schema` 命令来获取、修改、删除 DM 内部维护的表结构。

原理介绍

DM 中的 schema 信息来源包括以下几部分：

1. 执行全量数据迁移(`task-mode=all`) 时，任务将经过dump/load/sync（全量导出/全量导入/增量同步）三个阶段。dump 阶段将表结构信息随着数据一并导出，并自动在下游创建相关表。sync 阶段时以该表结构作为起始的表结构信息。
2. sync 阶段中，处理 DDL 语句（如 `ALTER TABLE`）时，DM 执行该语句的同时更新内部维护的表结构信息。
3. 如果任务是增量迁移(`task-mode=incremental`)，下游已经将待迁移的表创建完成，DM 会从下游数据库获取该表的结构信息（此行为随版本变化而不同）。

增量迁移过程的 schema 信息维护较为复杂，在整个数据链路中包含以下几类可能相同或不同的表结构。

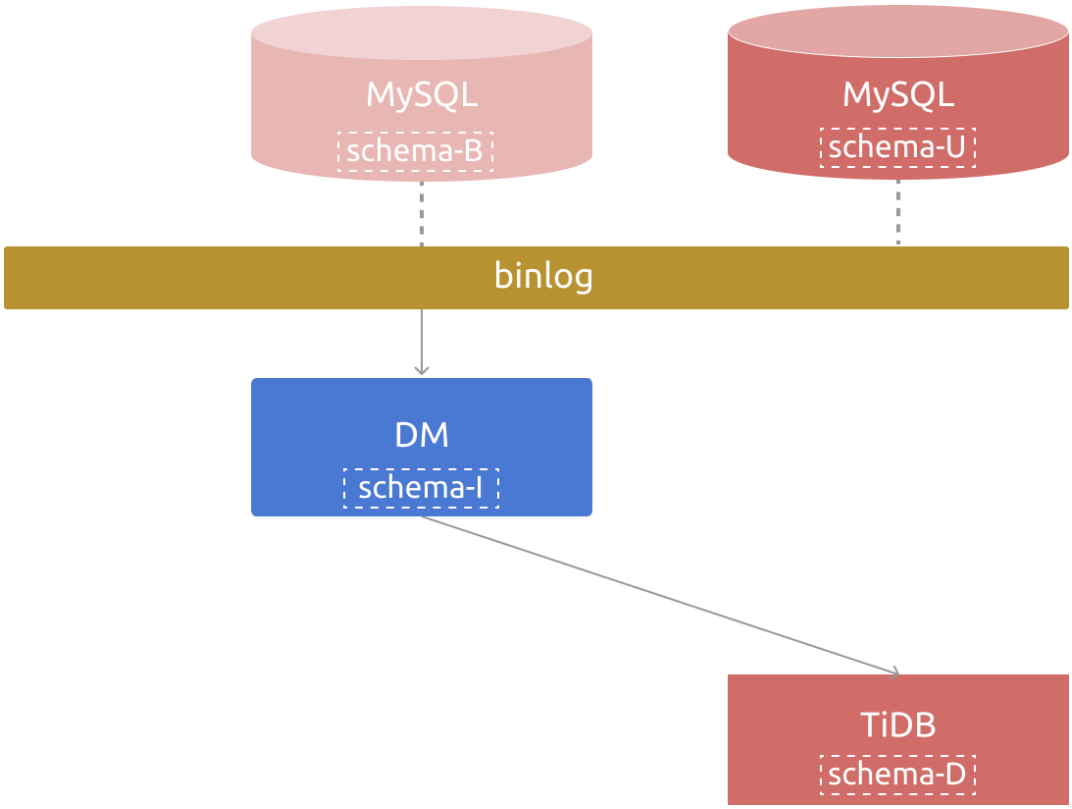


图 184: 表结构

- 上游当前时刻的表结构（记为 schema-U）。
- 当前 DM 正在消费的binlog event 的表结构（记为 schema-B，其对应于上游某个历史时刻的表结构）。
- DM 内部（schema tracker 组件）当前维护的表结构（记为 schema-I）。
- 下游TiDB 集群中的表结构（记为 schema-D）。

在大多数情况下，以上 4 类表结构一致。当上游执行 DDL 变更表结构后，schema-U 即会发生变更；DM 通过将该 DDL 应用于内部的 schema tracker 组件及下游TiDB，会先后更新 schema-I、schema-D 以与 schema-U 保持一致，因而随后能正常消费binlog 中在 DDL 之后对应表结构为 schema-B 的 binlog event。即当 DDL 被复制成功后，仍能保持 schema-U、schema-B、schema-I 及 schema-D 的一致。

需要注意以下不一致的情况：

- 在开启乐观 shard DDL 支持的数据迁移过程中，下游合并表的 schema-D 可能与部分分表对应的 schema-B 及 schema-I 并不一致，但 DM 仍保持 schema-I 与 schema-B 的一致，以确保能正常解析 DML 对应的binlog event。

- 当下游比上游多部分列时，schema-D 也可能会与 schema-B 及 schema-I 并不一致。全量数据迁移(task-mode=all) 时 DM 会自动处理不一致的情况；而增量迁移 (task-mode=incremental) 时，由于任务首次启动，内部尚无 Schema 信息，DM 将自动读取下游表结构，即 schema-D，更新自身的 schema-I (此行为随版本变化而不同)。在此之后，如果 DM 使用 schema-I 解析 schema-B 的 binlog，将会导致 Column count doesn't match value count 错误，详情可参考：下游存在更多列的迁移场景

binlog-schema 命令可以获取、修改、删除 DM 内部维护的表结构 schema-I。

> **注意：**
>
> binlog-schema 命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 operate-schema 命令。

命令介绍

```
help binlog-schema
```

```
manage or show table schema in schema tracker

Usage:
  dmctl binlog-schema [command]

Available Commands:
  delete      delete table schema structure
  list        show table schema structure
  update      update tables schema structure

Flags:
  -h, --help    help for binlog-schema

Global Flags:
  -s, --source strings    MySQL Source ID.

Use "dmctl binlog-schema [command] --help" for more information about a command.
```

> **注意：**
>
> - 由于表结构在数据迁移过程中可能会发生变更，为获取确定性的表结构，当前 binlog-schema 命令仅能在数据迁移任务处于 Paused 状态时可用。
> - 强烈建议在修改表结构前，首先获取并备份表结构，以免误操作导致数据丢失。

参数解释

- `delete`：删除表结构
- `list`：获取查看表结构
- `update`：更新设置表结构
- `-s` 或 `--source`:
  - 必选
  - 指定操作将应用到的 MySQL 源

使用示例

获取表结构

如需获取表结构，可使用 `binlog-schema list` 命令：

```
help binlog-schema list
```

```
show table schema structure

Usage:
  dmctl binlog-schema list <task-name> <database> <table> [flags]

Flags:
  -h, --help    help for list

Global Flags:
  -s, --source strings    MySQL Source ID.
```

假设要获取 `db_single` 任务对应于 `mysql-replica-01` MySQL 源的 `` `db_single`.`t1` `` 表的表结构，则执行如下命令：

```
binlog-schema list -s mysql-replica-01 task_single db_single t1
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "CREATE TABLE `t1` ( `c1` int(11) NOT NULL, `c2` int(11) DEFAULT NULL, PRIMARY
                ↪ KEY (`c1`)) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin",
            "source": "mysql-replica-01",
            "worker": "127.0.0.1:8262"
        }
    ]
}
```

更新设置表结构

如需更新设置表结构，可使用 `binlog-schema update` 命令：

```
help binlog-schema update
```

```
update tables schema structure

Usage:
  dmctl binlog-schema update <task-name> <database> <table> [schema-file] [flags]


Flags:
      --flush          flush the table info and checkpoint immediately (default true)
      --from-source    use the schema from upstream database as the schema of the specified tables
      --from-target    use the schema from downstream database as the schema of the specified
          '→ tables
  -h, --help           help for update
      --sync           sync the table info to master to resolve shard ddl lock, only for
          '→ optimistic mode now (default true)

Global Flags:
  -s, --source strings   MySQL Source ID.
```

假设要设置 db_single 任务对应于 mysql-replica-01 MySQL 源的 `db_single`.`t1` 表的表结构为如下所示：

```
CREATE TABLE `t1` (
    `c1` int(11) NOT NULL,
    `c2` bigint(11) DEFAULT NULL,
    PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin
```

则将上述 CREATE TABLE 语句保存为文件（如 db_single.t1-schema.sql）后，执行如下命令：

```
binlog-schema update -s mysql-replica-01 task_single db_single t1 db_single.t1-schema.sql
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "127.0.0.1:8262"
        }
    ]
}
```

删除表结构

如需删除表结构，可使用 binlog-schema delete 命令：

```
help binlog-schema delete
```

```
delete table schema structure

Usage:
  dmctl binlog-schema delete <task-name> <database> <table> [flags]

Flags:
  -h, --help    help for delete

Global Flags:
  -s, --source strings    MySQL Source ID.
```

> **注意：**
>
> 删除 DM 内部维护的表结构后，如果后续有该表的 DDL/DML 需要复制到下游，则 DM 会依次尝试从 checkpoint 表里 `table_info` 字段、乐观 shard DDL 协调中的元信息以及下游TiDB 中对应的该表获取表结构。

假设要删除 `db_single` 任务对应于 `mysql-replica-01` MySQL 源的 `` `db_single`.`t1` `` 表的表结构，则执行如下命令：

```
binlog-schema delete -s mysql-replica-01 task_single db_single t1
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "127.0.0.1:8262"
        }
    ]
}
```

10.6.8.6 TiDB Data Migration 导出和导入集群的数据源和任务配置

`config` 命令用于导出和导入集群的数据源和任务配置。

> **注意：**
>
> 对于 v2.0.5 版本之前的集群，可使用 v2.0.5 版本及之后的dmctl 导出和导入集群的数据源和任务配置文件。

```
» help config
Commands to import/export config

Usage:
  dmctl config [command]

Available Commands:
  export      Export the configurations of sources and tasks.
  import      Import the configurations of sources and tasks.

Flags:
  -h, --help    help for config

Global Flags:
  -s, --source strings    MySQL Source ID.

Use "dmctl config [command] --help" for more information about a command.
```

10.6.8.6.1 导出集群的数据源和任务配置

使用 `export` 子命令导出集群的数据源和任务配置到指定文件夹中。

```
config export [--dir directory]
```

参数解释

- `dir`：
  - 可选
  - 指定导出文件夹路径
  - 默认值为 `./configs`

返回结果示例

```
config export -d /tmp/configs
```

```
export configs to directory `/tmp/configs` succeed
```

10.6.8.6.2 导入集群的数据源和任务配置

使用 `import` 子命令从指定文件夹中导入集群的数据源和任务配置。

```
config import [--dir directory]
```

> **注意：**
>
> 对于 v2.0.2 版本之后的集群， 暂不支持自动导入 relay worker 的相关配置， 可以手动使用 `start-relay` 命令开启 relay log。

参数解释

- `dir`：
  - 可选
  - 指定导入文件夹路径
  - 默认值为 `./configs`

返回结果示例

```
config import -d /tmp/configs
```

```
start creating sources
start creating tasks
import configs from directory `/tmp/configs` succeed
```

10.6.8.7 TiDB Data Migration 处理告警

本文档介绍如何处理TiDB Data Migration (DM) 中的告警。

10.6.8.7.1 高可用告警

`DM_master_all_down`

当全部 DM-master 离线时触发该告警。发生该错误时，需要检查集群环境，并通过各节点日志排查错误。

`DM_worker_offline`

存在离线的 DM-worker 超过一小时会触发该告警。在高可用架构下，该告警可能不会直接中断任务，但是会提升任务中断的风险。处理告警可以查看对应 DM-worker 节点的工作状态，检查是否连通，并通过日志排查错误。

`DM_DDL_error`

处理 shard DDL 时出现错误，此时需要参考DM 故障诊断进行处理。

`DM_pending_DDL`

存在未完成的 shard DDL 并超过一小时会触发该告警。在某些应用场景下，存在未完成的 shard DDL 可能是用户所期望的。在用户预期以外的场景下，可以通过手动处理 Sharding DDL Lock解决。

10.6.8.7.2 任务状态告警

`DM_task_state`

当 DM-worker 内有子任务处于 Paused 状态超过 20 分钟时会触发该告警，此时需要参考DM 故障诊断进行处理。

10.6.8.7.3 relay log 告警

`DM_relay_process_exits_with_error`

当 relay log 处理单元遇到无法自动恢复的错误时（如找不到binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_remain_storage_of_relay_log`

当relay log 所在磁盘的剩余可用容量小于 10G 时会触发该告警，对应的处理方法包括：

- 手动清理该磁盘上其他无用数据以增加可用容量。
- 尝试调整relay log 的自动清理策略或执行手动清理。
- 使用 `pause-relay` 命令暂停relay log 的拉取，并在磁盘空间合适之后使用 `resume-relay` 命令恢复。需要注意上游数据源不要清理尚未拉取的binlog。

`DM_relay_log_data_corruption`

当 relay log 处理单元在校验从上游读取到的binlog event 且发现checksum 信息异常时会转为 Paused 状态并立即触发告警，此时需要参考DM 故障诊断进行处理。

`DM_fail_to_read_binlog_from_master`

当 relay log 处理单元在尝试从上游读取binlog event 发生错误时，会转为 Paused 状态并立即触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_fail_to_write_relay_log`

当 relay log 处理单元在尝试将binlog event 写入relay log 文件发生错误时，会转为 Paused 状态并立即触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_binlog_file_gap_between_master_relay`

当 relay log 处理单元已拉取到的最新的binlog 文件个数落后于当前上游 MySQL/MariaDB 超过 1 个（不含 1 个）且持续 10 分钟时会触发该告警，此时需要参考性能问题及处理方法对 relay log 处理单元相关的性能问题进行排查与处理。

10.6.8.7.4 Dump/Load 告警

`DM_dump_process_exists_with_error`

当 Dump 处理单元遇到无法自动恢复的错误时（如找不到binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_load_process_exists_with_error`

当 Load 处理单元遇到无法自动恢复的错误时（如找不到binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

10.6.8.7.5 Binlog replication 告警

`DM_sync_process_exists_with_error`

当 Binlog replication 处理单元遇到无法自动恢复的错误时（如找不到binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_binlog_file_gap_between_master_syncer`

当 Binlog replication 处理单元已处理到的最新的binlog 文件个数落后于当前上游 MySQL/MariaDB 超过 1 个（不含 1 个）且持续 10 分钟时 DM 会触发该告警，此时需要参考性能问题及处理方法对 Binlog replication 处理单元相关的性能问题进行排查与处理。

`DM_binlog_file_gap_between_relay_syncer`

当 Binlog replication 处理单元已处理到的最新的binlog 文件个数落后于当前relay log 处理单元超过 1 个（不含 1 个）且持续 10 分钟时 DM 会触发该告警，此时需要参考性能问题及处理方法对 Binlog replication 处理单元相关的性能问题进行排查与处理。

10.6.8.8 TiDB Data Migration 日常巡检

本文总结了TiDB Data Migration (DM) 工具日常巡检的方法：

- 方法一：执行 `query-status` 命令查看任务运行状态以及相关错误输出。详见查询状态。
- 方法二：如果使用TiUP 部署DM 集群时正确部署了Prometheus 与Grafana，如Grafana 的地址为`172.16.10.71`，可在浏览器中打开http://172.16.10.71:3000 进入Grafana，选择 DM 的 Dashboard 即可查看 DM 相关监控项。具体监控指标参照监控与告警设置。
- 方法三：通过日志文件查看 DM 运行状态和相关错误。
  - DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录位于 `{log_dir}`。
  - DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录位于 `{log_dir}`。

10.6.9 参考手册

10.6.9.1 架构组件

10.6.9.1.1 DM-worker 简介

DM-worker 是 DM (Data Migration) 的一个组件，负责执行具体的数据迁移任务。

其主要功能如下：

- 注册为一台 MySQL 或 MariaDB 服务器的 slave。
- 读取 MySQL 或 MariaDB 的 binlog event，并将这些 event 持久化保存在本地(relay log)。
- 单个 DM-worker 支持迁移一个 MySQL 或 MariaDB 实例的数据到下游的多个TiDB 实例。
- 多个 DM-Worker 支持迁移多个 MySQL 或 MariaDB 实例的数据到下游的一个TiDB 实例。

DM-worker 处理单元

DM-worker 任务包含如下多个逻辑处理单元。

Relay log

Relay log 持久化保存从上游 MySQL 或 MariaDB 读取的binlog，并对binlog replication 处理单元提供读取binlog event 的功能。

其原理和功能与 MySQL relay log 类似，详见 MySQL Relay Log。

dump 处理单元

dump 处理单元从上游 MySQL 或 MariaDB 导出全量数据到本地磁盘。

load 处理单元

load 处理单元读取dump 处理单元导出的数据文件，然后加载到下游TiDB。

Binlog replication/sync 处理单元

Binlog replication/sync 处理单元读取上游 MySQL/MariaDB 的 binlog event 或 relay log 处理单元的binlog event，将这些 event 转化为 SQL 语句，再将这些 SQL 语句应用到下游TiDB。

DM-worker 所需权限

本小节主要介绍使用 DM-worker 时所需的上下游数据库用户权限以及各处理单元所需的用户权限。

上游数据库用户权限

上游数据库(MySQL/MariaDB)用户必须拥有以下权限：

权限	作用域
SELECT	Tables
RELOAD	Global
REPLICATION SLAVE	Global
REPLICATION CLIENT	Global

如果要迁移 db1 的数据到TiDB，可执行如下的 GRANT 语句：

```
GRANT RELOAD,REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'your_user'@'your_wildcard_of_host'
GRANT SELECT ON db1.* TO 'your_user'@'your_wildcard_of_host';
```

如果还要迁移其他数据库的数据到TiDB，请确保已赋予这些库跟 db1 一样的权限。

下游数据库用户权限

下游数据库(TiDB)用户必须拥有以下权限：

权限	作用域
SELECT	Tables
INSERT	Tables
UPDATE	Tables
DELETE	Tables

权限	作用域
CREATE	Databases，tables
DROP	Databases，tables
ALTER	Tables
INDEX	Tables

对要执行迁移操作的数据库或表执行下面的 GRANT 语句：

```
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,ALTER,INDEX  ON db.table TO 'your_user'@'
    '→ your_wildcard_of_host';
GRANT ALL ON dm_meta.* TO 'your_user'@'your_wildcard_of_host';
```

处理单元所需的最小权限

处理单元	最小上游(MySQL/MariaDB)权限	最小下游(TiDB)权限	最小系统权限
Relay log	REPLICATION SLAVE (读取 binlog) REPLICATION CLIENT (show master status, show slave status)	无	本地读/写磁盘
Dump	SELECTRELOAD（获取读锁将表数据刷到磁盘，进行一些操作后，再释放读锁对表进行解锁）	无	本地写磁盘
Load	无	SELECT（查询 checkpoint 历史） CREATE（创建数据库或表） DELETE（删除 checkpoint） INSERT（插入 dump 数据）	读/写本地文件
Binlog replication	REPLICATION SLAVE（读 binlog） REPLICATION CLIENT (show master status, show slave status)	SELECT（显示索引和列）INSERT (DML)UPDATE (DML)DELETE (DML)CREATE（创建数据库或表）DROP（删除数据库或表）ALTER（修改表）INDEX（创建或删除索引）	本地读/写磁盘

> 注意：
>
> 这些权限并非一成不变。随着需求改变，这些权限也可能会改变。

10.6.9.1.2 DM 安全模式

安全模式(safe mode) 是 DM 在进行增量同步时候的一种运行模式，在安全模式中，DM 增量同步组件在同步 binlog event 时，将把所有 INSERT 和 UPDATE 操作强制进行改写后再在下游执行。

安全模式的目的是在增量同步过程中，同一条binlog event 能够在下游被重复同步且保证幂等性，从而确保增量同步能够“安全”进行。

DM 从 checkpoint 恢复数据同步任务后，可能重复执行某些binlog 事件而导致下述问题：

1. 在进行增量同步过程中，执行 DML 的操作和写checkpoint 的操作并不是同步的；写checkpoint 的操作和写下游数据的操作也并不能保证原子性。因此，当 DM 异常退出时，checkpoint 可能只记录到退出时刻之前的一个恢复点。
2. 当 DM 重启同步任务，并从checkpoint 重新开始增量数据同步时，checkpoint 之后的部分数据可能已经在异常退出前被处理过了，从而导致部分 SQL 语句重复执行。
3. 如果重复执行 INSERT 操作，会导致主键或唯一索引冲突，引发同步中断；如果重复执行 UPDATE 操作，会导致不能根据筛选条件找到之前对应的更新记录。

在安全模式下，通过改写 SQL 语句，DM 可以解决上述问题。

安全模式原理

安全模式通过 SQL 语句改写来保证binlog event 的幂等性。具体来说，在安全模式下：

- INSERT 语句会被改写成 REPLACE 语句。
- UPDATE 语句会被分析，得到该语句涉及的行的主键或唯一索引的值，然后改写成 DELETE + REPLACE 语句：先根据主键或唯一索引的定位删除对应的行，然后使用 REPLACE 语句插入一条最新值的行记录。

REPLACE 操作是 MySQL 特有的数据插入语法。使用 REPLACE 语法插入数据时，如果新插入的数据和现有数据存在主键或唯一约束冲突，MySQL 会删除所有冲突的记录，然后再执行插入记录操作，相当于“强制插入”的操作。具体请参考 MySQL 官方文档的 REPLACE 语句相关介绍。

比如，假设一张表 dummydb.dummytbl 的主键是 id，在这张表中重复执行下面的 SQL 语句：

```
INSERT INTO dummydb.dummytbl (id, int_value, str_value) VALUES (123, 999, 'abc');
UPDATE dummydb.dummytbl SET int_value = 888999 WHERE int_value = 999;     # 假设没有其他 int_value
    ↪ = 999的数据
UPDATE dummydb.dummytbl SET id = 999 WHERE id = 888;      # 更新主键操作
```

启用安全模式后，上述 SQL 语句再次在下游执行时，会被改写为下面的 SQL 语句：

```
REPLACE INTO dummydb.dummytbl (id, int_value, str_value) VALUES (123, 999, 'abc');
DELETE FROM dummydb.dummytbl WHERE id = 123;
REPLACE INTO dummydb.dummytbl (id, int_value, str_value) VALUES (123, 888999, 'abc');
DELETE FROM dummydb.dummytbl WHERE id = 888;
REPLACE INTO dummydb.dummytbl (id, int_value, str_value) VALUES (999, 888888, 'abc888');
```

上述语句中，UPDATE 被改写为DELETE + REPLACE，而不是DELETE + INSERT。如果这里使用INSERT，那么id = 999 的记录在已经存在的情况下重复插入时，数据库会报错主键冲突。因此这里使用了 REPLACE，新的记录会替换之前已经插入的记录。

通过这样的语句改写，在进行重复的插入或更新操作时，DM 都会使用执行该操作后的行数据来覆盖之前已经存在的行数据。这样保证了插入和更新操作的可重复执行。

开启安全模式

自动开启

当 DM 从 checkpoint 恢复增量同步任务（例如 worker 重启，网络中断重连等）时，会自动开启一段时间的安全模式。开启安全模式的逻辑，与checkpoint 中的 safemode_exit_point 信息相关。当一个增量同步任务异常暂停时，DM 会先尝试将内存中所有的 DML 全部同步到下游，然后记录当前内存中从上游拉取到的最新的binlog 位置点，记作 safemode_exit_point，把 safemode_exit_point 保存在异常暂停之前最后一个checkpoint 当中。

当 DM 从checkpoint 恢复增量同步任务时，会根据下面的判断逻辑来开启安全模式：

- 如果checkpoint 信息里面额外记录了 safemode_exit_point，说明该增量同步任务属于异常暂停。如果 DM 在恢复该增量同步任务时，发现待恢复的checkpoint 的 binlog 位置点在 safemode_exit_point 之前，那么说明从这个checkpoint 到 safemode_exit_point 之间的binlog 可能已经在下游被执行；恢复该任务后，部分binlog 会被重复执行。因此，DM 判断在这段binlog 范围内需要开启安全模式。在当前的binlog 位置点超过了 safemode_exit_point 后，未手动开启 safe-mode 的情况下，DM 将关闭安全模式。
- 如果checkpoint 信息里没有 safemode_exit_point 信息，则有两种情况：

    1. 这是一个全新的任务，或该任务是正常暂停。
    2. 该任务异常暂停时，记录 safemode_exit_point 失败；或是 DM 进程异常退出。

    第二种情况下，DM 并不知道哪些checkpoint 之后的binlog 已经被执行过。为了保险起见，只要没有在checkpoint 找到 safemode_exit_point 信息，DM 就会在前 2 个 checkpoint 间隔中都开启安全模式，确保这段时间里的binlog 被重复执行不会引发问题。默认的checkpoint 间隔是 30 秒，也就是说，一个正常的增量同步任务开始时，前 2 * 30 = 60 秒会自动强制开启安全模式。

如果你需要调整增量同步任务开始时安全模式的持续时间，可以设置 syncers 的配置项safe-mode-duration。

手动开启

你可以通过调整增量任务配置文件中syncer 的配置项safe-mode，控制是否需要全程开启安全模式。safe-mode 是布尔类型参数，默认为 false。如果设置为 true，则表示在增量同步的全过程中都会开启安全模式。例如，下面是一个开启了安全模式的任务配置示例：

```
syncers:                            # sync 处理单元的运行配置参数
  global:                           # 配置名称
```

```
    # 其他配置省略
    safe-mode: true                  # 增量同步任务全程开启"安全模式"
    # 其他配置省略
##### ------------ 实例配置 ------------
mysql-instances:
  -
    source-id: "mysql-replica-01"
    # 其他配置省略
    syncer-config-name: "global"            # syncers 配置的名称
```

注意事项

如果你出于安全考虑，希望全程开启安全模式，你需要注意以下事项：

- 安全模式对于增量同步有额外开销。频繁的 DELETE + REPLACE 操作会带来主键或唯一索引的频繁变化，带来了比单纯 UPDATE 语句更大的性能开销。
- 安全模式会强制替换主键相同的记录，可能导致下游的记录数据丢失。如果由于上游分片表合并导入下游的配置不当，可能导致在上游合并导入到下游数据库的时候，出现大量的主键或唯一索引冲突。如果此时全程开启安全模式，会导致下游大量的数据丢失，并且可能无法从任务中看到有任何的异常，导致大量数据不一致。
- 安全模式依赖于通过主键或唯一索引来判断冲突。如果下游数据库对应的表没有主键或唯一索引，会导致 REPLACE 语句无法实现替换插入的目的。这种情况下，即使开启了安全模式，DM 将 INSERT 语句改写成 REPLACE 并执行后，依旧会向下游插入重复记录。

因此，如果上游存在主键重复的数据，业务可以接受丢失重复数据和性能损失，你可以开启安全模式来忽略数据重复错误。

10.6.9.1.3 DM Relay Log

DM (Data Migration) 工具的relay log 由若干组有编号的文件和一个索引文件组成。这些有编号的文件包含了描述数据库更改的事件。索引文件包含所有使用过的relay log 的文件名。

在启用relay log 功能后，DM-worker 会自动将上游binlog 迁移到本地配置目录（若使用TiUP 部署 DM，则迁移目录默认为 <deploy_dir> / <relay_log>）。本地配置目录 <relay_log> 的默认值是 relay-dir，可在上游数据库配置文件中进行修改。自 v5.4.0 版本起，你可以在DM-worker 配置文件中通过 relay-dir 配置本地配置目录，其优先级高于上游数据库的配置文件。

适用场景

MySQL 的存储空间是有限制的，通常会设置binlog 的最长保存时间，当上游把binlog 清除掉之后，如果 DM 还需要对应位置的binlog 就会拉取失败，导致同步任务出错。每增加一个同步任务，DM 都会在上游建立一条连接用于拉取binlog，连接数过多会对上游造成比较大的负载。开启relay log 后，同一个上游的多个同步任务可以复用已经拉到本地的relay log，减少了对上游的压力。

在全量加增量数据迁移任务(task-mode=all) 中，DM 需要先进行全量数据迁移，再根据binlog 进行增量同步。若全量阶段持续时间较长，上游binlog 可能会被清除，导致增量同步无法进行。若提前开启了 relay log，DM 会自动在本地保留足够的日志，保证增量任务正常进行。

一般情况下建议开启relay log，但需知晓relay log 可能导致的负面作用：

由于 relay log 需要写入到磁盘中，这一过程会产生外部IO 和一些 CPU 消耗，可能导致整个同步链路变长，从而增加数据同步的时延。对时延要求十分敏感的同步任务，暂时不推荐使用relay log。

> **注意：**
>
> DM v2.0.7 及之后的版本中，对relay log 写入进行了优化，增加的时延和 CPU 消耗已相对较小。

使用 relay log

开启/关闭 relay log

在 v5.4.0 及之后的版本中，你可以通过将 `enable-relay` 设为 `true` 开启relay log。自 v5.4.0 起，DM-worker 在绑定上游数据源时，会检查上游数据源配置中的 `enable-relay` 项。如果 `enable-relay` 为 `true`，则为该数据源启用 relay log 功能。

具体配置方式参见上游数据源配置文件介绍

除此以外，你也可以通过 `start-relay` 或 `stop-relay` 命令动态调整数据源的 `enable-relay` 并即时开启或关闭 relay log。

```
start-relay -s mysql-replica-01
```

```
{
    "result": true,
    "msg": ""
}
```

> **注意：**
>
> 在 v2.0.2 及之后的 v2.0 版本，以及在 v5.3.0 版本中，上游数据源配置中的 `enable-relay` 项失效，你只能通过`start-relay` 和 `stop-relay`命令开启和关闭 relay log。加载数据源配置时，如果 DM 发现配置中的 `enable-relay` 项为 `true`，会给出如下信息提示：
>
> ```
> Please use `start-relay` to specify which workers should pull relay log of relay-
>     ↪ enabled sources.
> ```

> **警告：**
>
> 该启动方式在 v6.1 版本中标记为弃用，在未来版本可能会被移除。相关命令的输出中您会看到如下提示：`start-relay/stop-relay with worker name will be deprecated soon. You can try stopping relay first and use start-relay without worker name instead`。

start-relay 命令可以配置一个或多个 DM-worker 为指定数据源迁移relay log，但只能指定空闲或者已绑定了该上游数据源的 DM-worker。使用示例如下：

```
start-relay -s mysql-replica-01 worker1 worker2
```

```
{
    "result": true,
    "msg": ""
}
```

```
stop-relay -s mysql-replica-01 worker1 worker2
```

```
{
    "result": true,
    "msg": ""
}
```

在 v2.0.2 之前的版本（不含v2.0.2），DM-worker 在绑定上游数据源时，会检查上游数据源配置中的enable-relay 项。如果 enable-relay 为 true，则为该数据源启用 relay log 功能。

具体配置方式参见上游数据源配置文件介绍

查询relay log

使用 query-status -s 命令，可以查询relay log 的状态：

```
query-status -s mysql-replica-01
```

期望输出

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "no sub task started",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker2",
                "result": null,
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000005, 916)",
                    "masterBinlogGtid": "09bec856-ba95-11ea-850a-58f2b4af5188:1-28",
                    "relaySubDir": "09bec856-ba95-11ea-850a-58f2b4af5188.000001",
                    "relayBinlog": "(mysql-bin.000005, 4)",
                    "relayBinlogGtid": "09bec856-ba95-11ea-850a-58f2b4af5188:1-28",
                    "relayCatchUpMaster": false,
```

```
                    "stage": "Running",
                    "result": null
                }
            },
            "subTaskStatus": [
            ]
        },
        {
            "result": true,
            "msg": "no sub task started",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000005, 916)",
                    "masterBinlogGtid": "09bec856-ba95-11ea-850a-58f2b4af5188:1-28",
                    "relaySubDir": "09bec856-ba95-11ea-850a-58f2b4af5188.000001",
                    "relayBinlog": "(mysql-bin.000005, 916)",
                    "relayBinlogGtid": "",
                    "relayCatchUpMaster": true,
                    "stage": "Running",
                    "result": null
                }
            },
            "subTaskStatus": [
            ]
        }
    ]
}
```

暂停/恢复 relay log

`pause-relay` 与 `resume-relay` 命令可以分别暂停及恢复拉取relay log。这两个命令执行时都需要指定上游数据源的 `source-id`，例如：

```
pause-relay -s mysql-replica-01 -s mysql-replica-02
```

期望输出

```
{
    "op": "PauseRelay",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
```

```
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

```
resume-relay -s mysql-replica-01
```

期望输出

```
{
    "op": "ResumeRelay",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

清理 relay log

DM 提供两种清理relay log 的方式，手动清理和自动清理。两种清理方法都不会清理活跃的relay log。

> 注意：
>
> - 活跃的relay log：该relay log 正在被同步任务使用。活跃的relay log 当前只在 Syncer Unit 被更新和写入，如果一个为All 模式的同步任务在全量导出/导入阶段花费了超过数据源 purge 里配置的过期时间，该relay log 依旧会被清除。
> - 过期的relay log：该relay log 文件最后被改动的时间与当前时间的差值大于配置文件中的 `expires` 字段。

自动数据清理

启用自动数据清理需在 source 配置文件中进行以下配置：

```
##### relay log purge strategy
purge:
    interval: 3600
    expires: 24
    remain-space: 15
```

- purge.interval
  - 后台自动清理的时间间隔，以秒为单位。
  - 默认为“3600”，表示每 3600 秒执行一次后台清理任务。
- purge.expires
  - 当前 relay 处理单元没有写入、或已有数据迁移任务当前或未来不需要读取的relay log 在被后台清理前可保留的小时数。
  - 默认为“0”，表示不按relay log 的更新时间执行数据清理。
- purge.remain-space
  - 剩余磁盘空间，单位为 GB。若剩余磁盘空间小于该配置，则指定的 DM-worker 机器会在后台尝试自动清理可被安全清理的relay-log。若这一数字被设为“0”，则表示不按剩余磁盘空间来清理数据。
  - 默认为“15”，表示可用磁盘空间小于 15GB 时，DM-master 会尝试安全地清理relay log。

手动数据清理

手动数据清理是指使用dmctl 提供的 purge-relay 命令，通过指定 subdir 和binlog 文件名，来清理掉指定 binlog 之前的所有relay log。若在命令中不填 -subdir 选项，则默认清理最新relay log 子目录之前的所有relay log。

假设当前relay log 的目录结构如下：

```
tree .
```

```
.
|-- deb76a2b-09cc-11e9-9129-5242cf3bb246.000001
|   |-- mysql-bin.000001
|   |-- mysql-bin.000002
|   |-- mysql-bin.000003
|   `-- relay.meta
|-- deb76a2b-09cc-11e9-9129-5242cf3bb246.000003
|   |-- mysql-bin.000001
|   `-- relay.meta
|-- e4e0e8ab-09cc-11e9-9220-82cc35207219.000002
|   |-- mysql-bin.000001
|   `-- relay.meta
`-- server-uuid.index
```

```
cat server-uuid.index
```

```
deb76a2b-09cc-11e9-9129-5242cf3bb246.000001
e4e0e8ab-09cc-11e9-9220-82cc35207219.000002
deb76a2b-09cc-11e9-9129-5242cf3bb246.000003
```

在 dmctl 中使用 purge-relay 命令的示例如下：

- 以下命令指定的 relay log 子目录为 e4e0e8ab-09cc-11e9-9220-82cc35207219.000002，该子目录之前的 relay log 子目录为 deb76a2b-09cc-11e9-9129-5242cf3bb246.000001。所以该命令实际清空了 deb76a2b-09
  ′→ cc-11e9-9129-5242cf3bb246.000001 子目录，保留 e4e0e8ab-09cc-11e9-9220-82cc35207219.000002 和 deb76a2b-09cc-11e9-9129-5242cf3bb246.000003 子目录。

```
purge-relay -s mysql-replica-01 --filename mysql-bin.000001 --sub-dir e4e0e8ab-09cc-11e9
    ′→ -9220-82cc35207219.000002
```

- 以下命令默认 --sub-dir 为最新的 deb76a2b-09cc-11e9-9129-5242cf3bb246.000003 子目录。该目录之前的relay log 子目录为 deb76a2b-09cc-11e9-9129-5242cf3bb246.000001 和 e4e0e8ab-09cc-11e9-9220-82
  ′→ cc35207219.000002，所以该命令实际清空了这两个子目录，保留了 deb76a2b-09cc-11e9-9129-5242
  ′→ cf3bb246.000003。

```
purge-relay -s mysql-replica-01 --filename mysql-bin.000001
```

Relay log 内部机制

目录结构

Relay log 本地存储的目录结构示例如下：

```
<deploy_dir>/<relay_log>/
|-- 7e427cc0-091c-11e9-9e45-72b7c59d52d7.000001
|   |-- mysql-bin.000001
|   |-- mysql-bin.000002
|   |-- mysql-bin.000003
|   |-- mysql-bin.000004
|   `-- relay.meta
|-- 842965eb-091c-11e9-9e45-9a3bff03fa39.000002
|   |-- mysql-bin.000001
|   `-- relay.meta
`-- server-uuid.index
```

- subdir：
  - DM-worker 把从上游数据库迁移到的binlog 存储在同一目录下，每个目录都为一个 subdir。
  - subdir 的命名格式为<上游数据库 UUID>.<本地 subdir 序列号>。
  - 在上游进行primary 和 secondary 实例切换后，DM-worker 会生成一个序号递增的新 subdir 目录。
  - 在以上示例中，对于7e427cc0-091c-11e9-9e45-72b7c59d52d7.000001 这一目录，7e427cc0-091c-11
    ′→ e9-9e45-72b7c59d52d7 是上游数据库的 UUID，000001 是本地 subdir 的序列号。

- `server-uuid.index`：记录当前可用的 `subdir` 目录。
- `relay.meta`：存储每个 `subdir` 中已迁移的binlog 信息。例如，

```
cat c0149e17-dff1-11e8-b6a8-0242ac110004.000001/relay.meta
```

```
binlog-name = "mysql-bin.000010"     # 当前迁移的 binlog 名
binlog-pos = 63083620                # 当前迁移的 binlog 位置
binlog-gtid = "c0149e17-dff1-11e8-b6a8-0242ac110004:1-3328" # 当前迁移的 binlog GTID
```

也可能包含多个GTID：

```
cat 92acbd8a-c844-11e7-94a1-1866daf8accc.000001/relay.meta
```

```
binlog-name = "mysql-bin.018393"
binlog-pos = 277987307
binlog-gtid = "3ccc475b-2343-11e7-be21-6c0b84d59f30:1-14,406a3f61-690d-11e7-87c5-6
    ′→ c92bf46f384:1-94321383,53bfca22-690d-11e7-8a62-18ded7a37b78:1-495,686e1ab6-c47e-11e7
    ′→ -a42c-6c92bf46f384:1-34981190,03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474
    ′→ d3c-28c7-11e7-8352-203db246dd3d:1-170,10b039fc-c843-11e7-8f6a-1866daf8d810
    ′→ :1-308290454"
```

DM 接收binlog 的位置

- 从保存的checkpoint 中（默认位于下游dm_meta 库），获取各同步任务需要该数据源的最早位置。如果该位置晚于下述任何一个位置，则从此位置开始迁移。
- 若本地relay log 有效（有效是指relay log 具有有效的 `server-uuid.index`，`subdir` 和 `relay.meta` 文件），DM-worker 从 `relay.meta` 记录的位置恢复迁移。
- 若不存在有效的本地relay log，但上游数据源配置文件中指定了 `relay-binlog-name` 或 `relay-binlog-gtid`：
  - 在非GTID 模式下，若指定了 `relay-binlog-name`，则 DM-worker 从指定的binlog 文件开始迁移。
  - 在 GTID 模式下，若指定了 `relay-binlog-gtid`，则 DM-worker 从指定的GTID 开始迁移。
- 若不存在有效的本地relay log，而且 DM 配置文件中未指定 `relay-binlog-name` 或 `relay-binlog-gtid`：
  - 在非GTID 模式下，DM-worker 从自身所有 subtask 正在同步的最早binlog 开始迁移，直至最新。
  - 在 GTID 模式下，DM-worker 从自身所有 subtask 正在同步的最早GTID 开始迁移，直至最新。

> 注意：
> 若上游的relay log 被清理掉，则会发生错误。在这种情况下，需设置 `relay-binlog-gtid` 来指定迁移的起始位置。

探索更多

- DM 源码阅读系列文章（六）relay log 的实现 | TiDB 工具

10.6.9.1.4 Data Migration DDL 特殊处理说明

DM 同步过程中，根据 DDL 语句以及所处场景的不同，将采用不同的处理方式。

忽略的 DDL 语句

以下语句 DM 并未支持，因此解析之后直接跳过。

```
<tr>
    <th>描述</th>
    <th>SQL</th>
</tr>
<tr>
    <td>transaction</td>
    <td><code>^SAVEPOINT</code></td>
</tr>
<tr>
    <td>skip all flush sqls</td>
    <td><code>^FLUSH</code></td>
</tr>
<tr>
    <td rowspan="3">table maintenance</td>
    <td><code>^OPTIMIZE\\s+TABLE</code></td>
</tr>
<tr>
    <td><code>^ANALYZE\\s+TABLE</code></td>
</tr>
<tr>
    <td><code>^REPAIR\\s+TABLE</code></td>
</tr>
<tr>
    <td>temporary table</td>
    <td><code>^DROP\\s+(\\/\\*\\!40005\\s+)?TEMPORARY\\s+(\\*\\/\\s+)?TABLE</code></td>
</tr>
<tr>
    <td rowspan="2">trigger</td>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?TRIGGER</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+TRIGGER</code></td>
</tr>
<tr>
    <td rowspan="3">procedure</td>
    <td><code>^DROP\\s+PROCEDURE</code></td>
</tr>
<tr>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?PROCEDURE</code></td>
</tr>
```

```
<tr>
    <td><code>^ALTER\\s+PROCEDURE</code></td>
</tr>
<tr>
    <td rowspan="3">view</td>
    <td><code>^CREATE\\s*(OR REPLACE)?\\s+(ALGORITHM\\s?=.+?)?(DEFINER\\s?=.+?)?\\s+(SQL SECURITY
        ↪ DEFINER)?VIEW</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+VIEW</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+(ALGORITHM\\s?=.+?)?(DEFINER\\s?=.+?)?(SQL SECURITY DEFINER)?VIEW</code
        ↪ ></td>
</tr>
<tr>
    <td rowspan="4">function</td>
    <td><code>^CREATE\\s+(AGGREGATE)?\\s*?FUNCTION</code></td>
</tr>
<tr>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?FUNCTION</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+FUNCTION</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+FUNCTION</code></td>
</tr>
<tr>
    <td rowspan="3">tableSpace</td>
    <td><code>^CREATE\\s+TABLESPACE</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+TABLESPACE</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+TABLESPACE</code></td>
</tr>
<tr>
    <td rowspan="3">event</td>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?EVENT</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+(DEFINER\\s?=.+?)?EVENT</code></td>
</tr>
```

```
<tr>
    <td><code>^DROP\\s+EVENT</code></td>
</tr>
<tr>
    <td rowspan="7">account management</td>
    <td><code>^GRANT</code></td>
</tr>
<tr>
    <td><code>^REVOKE</code></td>
</tr>
<tr>
    <td><code>^CREATE\\s+USER</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+USER</code></td>
</tr>
<tr>
    <td><code>^RENAME\\s+USER</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+USER</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+USER</code></td>
</tr>
```

改写的 DDL 语句

以下语句在同步到下游前会进行改写。

原始语句	实际执行语句
^CREATE DATABASE...	^CREATE DATABASE...IF NOT EXISTS
^CREATE TABLE...	^CREATE TABLE..IF NOT EXISTS
^DROP DATABASE...	^DROP DATABASE...IF EXISTS
^DROP TABLE...	^DROP TABLE...IF EXISTS
^DROP INDEX...	^DROP INDEX...IF EXISTS

合库合表迁移任务

当使用悲观协调模式和乐观协调模式进行分库分表合并迁移时，DDL 同步的行为存在变更，具体请参考悲观模式和乐观模式。

Online DDL

Online DDL 特性也会对 DDL 事件进行特殊处理，详情可参考迁移使用 GH-ost/PT-osc 的源数据库。

10.6.9.2 运行机制

10.6.9.2.1 Data Migration 中的 DML 同步机制

本文介绍了 DM 核心处理单元 Sync 如何同步从数据源或relay log 中读取到的 DML。本文将介绍 DML 事件在 DM 内部的完整处理流程，包括binlog 读取、过滤、路由、转换、优化以及执行等逻辑，并详细解读 DML 优化逻辑和 DML 执行逻辑。

DML 处理流程

Sync 单元对 DML 的处理步骤如下：

1. 从 MySQL、 MariaDB 或 relay log 中，读取binlog event。
2. 转换读取到的binlog event：
    1. Binlog filter：根据 binlog 表达式过滤binlog event，通过 `filters` 配置。
    2. Table routing：根据“库/表”路由规则对“库/表”名进行转换，通过 `routes` 配置。
    3. Expression filter：根据 SQL 表达式过滤binlog event，通过 `expression-filter` 配置。
3. 优化 DML 执行：
    1. Compactor：将对同一条记录（主键相同）的多个操作合并成一个操作，通过 `syncer.compact` 开启。
    2. Causality：将不同记录（主键不同）进行冲突检测，提升同步并发度。
    3. Merger：将多条 binlog 合并成一条 DML，通过 `syncer.multiple-rows` 开启。
4. 将 DML 执行到下游。
5. 定期保存binlog position 或 GTID 到 checkpoint 中。

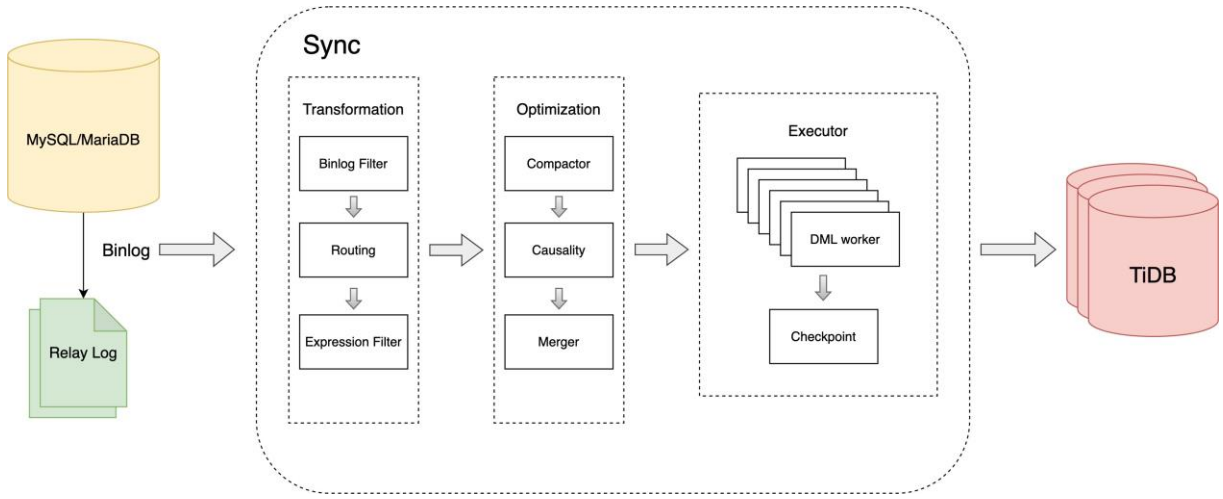


图 185: DML 处理逻辑

DML 优化逻辑

Sync 单元通过 Compactor、 Causality、 Merger 三个步骤，实现对 DML 的优化逻辑。

Compactor

DM 根据上游 binlog 记录，捕获记录的变更并同步到下游。当上游对同一条记录短时间内做了多次变更时（INSERT/UPDATE/DELETE），DM 可以通过 Compactor 将多次变更压缩成一次变更，减少下游压力，提升吞吐。例如：

```
INSERT + UPDATE => INSERT
INSERT + DELETE => DELETE
UPDATE + UPDATE => UPDATE
UPDATE + DELETE => DELETE
DELETE + INSERT => UPDATE
```

Compactor 特性默认关闭，如需启用可在迁移任务的 sync 配置模块开启，例如：

```
syncers:                                # sync 处理单元的运行配置参数
  global:                               # 配置名称
    ...                                 # 省略其他配置
    compact: true
```

Causality

MySQL binlog 顺序同步模型要求按照binlog 顺序依次同步binlog event，这样的同步模型无法满足高 QPS 低同步延迟的需求。此外，由于不是所有的binlog 涉及到的操作都存在冲突，顺序同步也是非必要的。

DM 通过冲突检测机制，识别出需要顺序执行的binlog，确保这些binlog 顺序执行的同时，最大程度地保持其他 binlog 并发执行，以此提高binlog 同步的性能。

Causality 采用一种类似并查集的算法，对每一个 DML 进行分类，将相互关联的 DML 分为一组。具体算法可参考并行执行 DML。

Merger

根据 MySQL binlog 协议，每条binlog 对应一行数据的变更操作。通过 Merger，DM 可以将多条binlog 合并成一条 DML，再执行到下游，减少网络的交互。例如：

```
  INSERT tb(a,b) VALUES(1,1);
+ INSERT tb(a,b) VALUES(2,2);
= INSERT tb(a,b) VALUES(1,1),(2,2);

  UPDATE tb SET a=1, b=1 WHERE a=1;
+ UPDATE tb SET a=2, b=2 WHERE a=2;
= INSERT tb(a,b) VALUES(1,1),(2,2) ON DUPLICATE UPDATE a=VALUES(a), b=VALUES(b)

  DELETE tb WHERE a=1
+ DELETE tb WHERE a=2
= DELETE tb WHERE (a) IN (1),(2);
```

Merger 特性默认关闭，如需启用可在迁移任务的 sync 配置模块开启，例如：

```
syncers:                                # sync 处理单元的运行配置参数
  global:                               # 配置名称
```

```
...                                    # 省略其他配置
multiple-rows: false
```

DML 执行逻辑

Sync 单元对 DML 进行优化后，再进行执行逻辑。

DML 生成

DM 内嵌一个 schema tracker，用于记录上下游的 schema 信息：

- 当 DM 收到 DDL 时，DM 更新内部 schema tracker 的表结构。
- 当收到 DML 时，DM 根据 schema tracker 的表结构生成对应的 DML。

生成 DML 的具体逻辑如下：

1. Sync 记录上游的初始表结构：
    - 当启动全量与增量任务时，Sync 使用上游全量同步时导出的表结构作为上游的初始表结构。
    - 当启动增量任务时，由于 MySQL binlog 没有记录表结构信息，Sync 使用下游对应的表的表结构作为上游的初始表结构。
2. 由于用户上下游表结构可能不一致，例如下游比上游多了额外的列，或者上下游主键不一致，为了保证数据同步的正确性，DM 记录下游对应表的主键和唯一键信息。
3. DM 生成 DML：
    - 使用 schema tracker 中记录的上游表结构生成 DML 语句的列名。
    - 使用binlog 中记录的列值生成 DML 语句的列值。
    - 使用 schema tracker 中记录的下游主键或唯一键生成 DML 语句中的 WHERE 条件。当表结构无唯一键时，DM 会使用binlog 中记录的所有列值作为 WHERE 条件。

Worker count

Causality 可以通过冲突检测算法将binlog 分成多个 group 并发地执行到下游。DM 通过设置 worker-count，控制并发的数量。当下游TiDB 的 CPU 占用不高时，增大并发的数量可以有效地提高数据同步的吞吐量。

你可以通过`syncer.worker-count` 配置项，修改并发迁移 DML 的线程数量。

Batch

DM 将多条 DML 积累到一个事务中执行到下游。当 DML Worker 收到 DML 时，将 DML 加入到缓存中。当缓存中 DML 数量达到预定阈值时，或者 DML worker 较长时间没有收到 DML 时，DML worker 将缓存中的 DML 执行到下游。

你可以通过`syncer.batch` 配置项，修改每个事务包含的 DML 的数量。

checkpoint

DML 执行和checkpoint 更新的操作不是原子的。

在 DM 中，checkpoint 默认每 30 秒更新一次。同时，由于存在多个 DML worker 进程，checkpoint 进程会计算所有 DML worker 中同步进度最早的binlog 位点，将该位点作为当前同步的checkpoint。所有早于此位点的binlog，都已保证被成功地执行到下游。

注意事项

事务一致性

DM 是按照“行级别”进行数据同步的，并不保证事务原样同步。在 DM 中，上游的一个事务会被拆成多行，分发到不同的 DML Worker 中并发执行。因此，当 DM 同步任务报错暂停，或者用户手动暂停任务时，下游可能停留在一个中间状态；即上游一个事务中的 DML 语句，可能一部分同步到下游，一部分没有，导致下游处于不一致的状态。

为了尽可能使任务暂停时下游处于一致状态，DM v5.3.0 起，任务暂停时，会等待 10 秒，使正在处理的上游事务全部同步到下游再真正暂停任务。如果上游一个事务在 10 秒内还未全部同步到下游，那么下游仍然可能处于不一致的状态。

安全模式

DML 执行和checkpoint 写操作不是同步的，并且写checkpoint 操作和写下游数据也并不能保证原子性。当 DM 因为某些原因异常退出时，checkpoint 可能只记录到退出时刻之前的一个恢复点。因此，当同步任务重启时，DM 可能会重复写入部分数据，也就是说，DM 实际上提供的是“至少一次处理”的逻辑（At-least-once processing），相同的数据可能会被处理一次以上。

为了保证数据是可重入的，DM 在异常重启时会进入安全模式。具体逻辑参阅DM 安全模式。

开启安全模式期间，为了保证数据可重入，DM 会进行如下转换：

- 将上游 INSERT 语句转换成 REPLACE 语句
- 将上游 UPDATE 语句转换成 DELETE + REPLACE 语句。

精确一次处理(Exactly-Once Processing)

目前 DM 仅保证最终一致性，尚未支持“精确一次处理”及“保持事务原有顺序同步”。

10.6.9.3 命令行

10.6.9.3.1 TiDB Data Migration 命令行参数

本文档介绍TiDB Data Migration (DM) 中各组件的主要命令行参数。

DM-master

`--advertise-addr`

- DM-master 用于接收客户端请求的外部地址
- 默认值为 `"{master-addr}"`
- 可选参数，可以为`"域名:port"` 的形式

`--advertise-peer-urls`

- DM-master 节点间通信的外部连接地址
- 默认值为 `"{peer-urls}"`
- 可选参数，可以为 `"http(s)://域名:port"` 的形式

--config

- DM-master 配置文件路径
- 默认值为 ""
- 可选参数

--data-dir

- DM-master 用于存储自身数据的目录
- 默认值为 "default.{name}"
- 可选参数

--initial-cluster

- 用于bootstrap DM-master 集群的 "{节点名}={外部地址}" 列表
- 默认值为 "{name}={advertise-peer-urls}"
- 在未指定 join 参数时需要指定该参数。 一个 3 节点集群的配置示例为 "dm-master-1=http
  ′→ ://172.16.15.11:8291,dm-master-2=http://172.16.15.12:8291,dm-master-3=http://172.16.15.13:8291"
  ′→

--join

- DM-master 节点加入到已有集群时，已有集群的 advertise-addr 地址列表
- 默认值为 ""
- 未指定 initial-cluster 参数时需要指定该参数。一个新节点加入到一个已有 2 个节点的集群的示例为 "172.16.15.11:8261,172.16.15.12:8261"

--log-file

- log 输出文件名
- 默认值为 ""
- 可选参数

-L

- log 级别
- 默认值为 "info"
- 可选参数

--master-addr

- DM-master 监听客户端请求的地址
- 默认值为 ""
- 必选参数

`--name`

- DM-master 节点名称
- 默认值为 `"dm-master-{hostname}"`
- 必选参数

`--peer-urls`

- DM-master 节点间通信的监听地址
- 默认值为 `"http://127.0.0.1:8291"`
- 必选参数

DM-worker

`--advertise-addr`

- DM-worker 用于接受客户端请求的外部地址
- 默认值为 `"{worker-addr}"`
- 可选参数，可以为`"域名:port"` 的形式

`--config`

- DM-worker 配置文件路径
- 默认值为 `""`
- 可选参数

`--join`

- DM-worker 注册到集群时，相应集群的 DM-master 节点的 `{advertise-addr}` 列表
- 默认值为 `""`
- 必选参数，一个3 DM-master 节点的集群配置示例为`"172.16.15.11:8261,172.16.15.12:8261,172.16.15.13:8261"`

`--log-file`

- log 输出文件名
- 默认值为 `""`
- 可选参数

`-L`

- log 级别
- 默认值为 `"info"`
- 可选参数

`--name`

- DM-worker 节点名称
- 默认值为 `"{advertise-addr}"`
- 必选参数

`--worker-addr`

- DM-worker 监听客户端请求的地址
- 默认值为 `""`
- 必选参数

dmctl

`--config`

- dmctl 配置文件路径
- 默认值为 `""`
- 可选参数

`--master-addr`

- dmctl 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`
- 默认值为 `""`
- 需要与 DM-master 交互时为必选参数

`--encrypt`

- 将明文数据库密码加密成密文
- 默认值为 `""`
- 指定该参数时，仅用于加密明文而不会与 DM-master 交互

`--decrypt`

- 将使用dmctl 加密过的密文解密为明文
- 默认值为 `""`
- 指定该参数时，仅用于解密密文而不会与 DM-master 交互

10.6.9.4 配置文件

10.6.9.4.1 DM 配置简介

本文档简要介绍 DM (Data Migration) 的配置文件和数据迁移任务的配置。

配置文件

- `dm-master.toml`：DM-master 进程的配置文件，包括 DM-master 的拓扑信息、日志等各项配置。配置说明详见DM-master 配置文件介绍。
- `dm-worker.toml`：DM-worker 进程的配置文件，包括 DM-worker 的拓扑信息、日志等各项配置。配置说明详见DM-worker 配置文件介绍。
- `source.yaml`：上游数据库 MySQL/MariaDB 相关配置。配置说明详见上游数据库配置文件介绍。

迁移任务配置

创建数据迁移任务

具体步骤如下：

1. 使用 dmctl 将数据源配置加载到 DM 集群；
2. 参考数据任务配置向导来创建 `your_task.yaml`；
3. 使用 dmctl 创建数据迁移任务。

关键概念

DM 配置的关键概念如下：

概念	解释	配置文件
source-id	唯一确定一个 MySQL 或 MariaDB 实例，或者一个具有主从结构的复制组，字符串长度不大于 32	`source.yaml` 的 `source-id`；`task.yaml` 的 `source-id`
DM-master ID	唯一确定一个 DM-master（取值于 `dm-master.toml` 的 `master-addr` 参数）	`dm-master.toml` 的 `master-addr`
DM-worker ID	唯一确定一个 DM-worker（取值于 `dm-worker.toml` 的 `worker-addr` 参数）	`dm-worker.toml` 的 `worker-addr`

10.6.9.4.2 TiDB Data Migration 上游数据库配置文件介绍

本文介绍TiDB Data Migration (DM) 上游数据库的配置文件，包括配置文件示例与配置项说明。

配置文件示例

上游数据库的示例配置文件如下所示：

```
source-id: "mysql-replica-01"

##### 是否开启 GTID
enable-gtid: false

##### 是否开启 relay log
enable-relay: false       # 该配置项从 DM v2.0.2 版本起弃用，使用 `start-relay` 命令开启 relay
    '→ log
relay-binlog-name: ""     # 拉取上游 binlog 的起始文件名
relay-binlog-gtid: ""     # 拉取上游 binlog 的起始 GTID
##### relay-dir: "relay-dir" # 存储 relay log 的目录，默认值为 "relay-dir"。从 v6.1
    '→ 版本起该配置标记为弃用，被 worker 配置中的同名参数取代

from:
  host: "127.0.0.1"
  port: 3306
  user: "root"
  password: "ZqMLjZ2j5khNelDEfDoUhkD5aV5fIJOe0fiog9w=" # 推荐使用 dmctl
      '→ 对上游数据库的用户密码加密之后的密码
  security:                       # 上游数据库 TLS 相关配置
    ssl-ca: "/path/to/ca.pem"
    ssl-cert: "/path/to/cert.pem"
    ssl-key: "/path/to/key.pem"

##### purge:
#####   interval: 3600
#####   expires: 0
#####   remain-space: 15

##### checker:
#####   check-enable: true
#####   backoff-rollback: 5m0s
#####   backoff-max: 5m0s       # backoff 的最大值，不能小于 1s

##### 从 DM v2.0.2 开始，Binlog event filter 也可以在上游数据库配置文件中进行配置
##### case-sensitive: false
##### filters:
##### - schema-pattern: dmctl
#####   table-pattern: t_1
#####   events: []
#####   sql-pattern:
```

```
#####    - alter table .* add column `aaa` int
#####    action: Ignore
```

> **注意：**
>
> 在 DM v2.0.1 版本中，请勿同时配置 `enable-gtid` 与 `enable-relay` 为 `true`，否则可能引发增量数据丢失问题。

配置项说明

Global 配置

配置项	说明
`source-id`	标识一个 MySQL 实例。
`enable-gtid`	是否使用GTID 方式从上游拉取binlog，默认值为 false。一般情况下不需要手动配置，如果上游数据库启用了GTID 支持，且需要做主从切换，则将该配置项设置为true。
`enable-relay`	是否开启relay log，默认值为 false。自 DM v2.0.2 版本起，该配置项弃用，需使用 `start-relay` 命令开启 relay log。
`relay-binlog-name`	拉取上游binlog 的起始文件名，例如“mysql-bin.000002”，该配置在 `enable-gtid` 为 false 的情况下生效。如果不配置该项，DM-worker 将从正在同步的最早的binlog 文件开始拉取，一般情况下不需要手动配置。
`relay-binlog-gtid`	拉取上游binlog 的起始GTID，例如“e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849”，该配置在 `enable-gtid` 为 true 的情况下生效。如果不配置该项，DM-worker 将从正在同步的最早binlog GTID 开始拉取binlog，一般情况下不需要手动配置。
`relay-dir`	存储relay log 的目录，默认值为“./relay_log”。
`host`	上游数据库的host。
`port`	上游数据库的端口。
`user`	上游数据库使用的用户名。
`password`	上游数据库的用户密码。注意：推荐使用dmctl 加密后的密码。
`security`	上游数据库 TLS 相关配置。配置的证书文件路径需能被所有节点访问。若配置为本地路径，则集群所有节点需要将证书文件拷贝一份放在各节点机器相同的路径位置上。

relay log 清理策略配置（purge 配置项）

一般情况下不需要手动配置，如果 relay log 数据量较大，磁盘空间不足，则可以通过设置该配置项来避免 relay log 写满磁盘。

配置项	说明
`interval`	定期检查relay log 是否过期的间隔时间，默认值：3600，单位：秒。
`expires`	relay log 的过期时间，默认值为 0，单位：小时。未由 relay 处理单元进行写入、或已有数据迁移任务当前或未来不需要读取的relay log 在超过过期时间后会被 DM 删除。如果不设置则 DM 不会自动清理过期的 relay log。
`remain-space`	设置最小的可用磁盘空间。当磁盘可用空间小于这个值时，DM-worker 会尝试删除relay log，默认值：15，单位：GB。

> 注意：
>
> 仅在 `interval` 不为 0 且 `expires` 和 `remain-space` 两个配置项中至少有一个不为 0 的情况下 DM 的自动清理策略才会生效。

任务状态检查配置（checker 配置项）

DM 会定期检查当前任务状态以及错误信息，判断恢复任务能否消除错误，并自动尝试恢复任务进行重试。DM 会使用指数回退策略调整检查间隔。这些行为可以通过如下配置进行调整：

配置项	说明
`check-enable`	启用自动重试功能。
`backoff-rollback`	如果指数回退策略的间隔大于该值，且任务处于正常状态，尝试减小间隔。
`backoff-max`	指数回退策略的间隔的最大值，该值必须大于 1 秒。

Binlog event filter

从 DM v2.0.2 开始，Binlog event filter 也可以在上游数据库配置文件中进行配置。

配置项	说明
`case-sensitive`	Binlog event filter 标识符是否大小写敏感。默认值：false。
`filters`	配置 Binlog event filter，含义见Binlog event filter 参数解释。

10.6.9.4.3 DM 任务完整配置文件介绍

本文档主要介绍 Data Migration (DM) 的任务完整的配置文件，包含全局配置和实例配置两部分。

关键概念

关于包括 source-id 和 DM-worker ID 在内的关键概念的介绍，请参阅关键概念。

完整配置文件示例

下面是一个完整的配置文件示例，通过该示例可以完成复杂的数据迁移功能。

```
---

##### ------------ 全局配置 ------------
###### ********* 基本信息配置 *********
name: test                          # 任务名称，需要全局唯一
task-mode: all                      # 任务模式，可设为 "full" - "只进行全量数据迁移"、 "incremental" -
    ↪  "Binlog 实时同步"、 "all" - "全量 + Binlog 实时同步"
shard-mode: "pessimistic"           # 任务协调模式，可选的模式有 ""、 "pessimistic、 "optimistic"。
    ↪ 默认值为 "" 即无需协调。如果是分库分表合并任务，请设置为悲观协调模式 "pessimistic"。
                                    # 在 v2.0.6 版本后乐观模式逐渐成熟，
                                        ↪ 深入了解乐观协调模式的原理和使用限制后，
                                        ↪ 也可以设置为乐观协调模式 "optimistic"
meta-schema: "dm_meta"              # 下游储存 `meta` 信息的数据库
##### timezone: "Asia/Shanghai"      # 指定数据迁移任务时 SQL Session 使用的时区。DM
    ↪ 默认使用目标库的全局时区配置进行数据迁移，并且自动确保同步数据的正确性。
    ↪ 使用自定义时区依然可以确保整个流程的正确性，但一般不需要手动指定。

case-sensitive: false               # schema/table 是否大小写敏感
online-ddl: true                    # 支持上游 "gh-ost" 、 "pt" 的自动处理
online-ddl-scheme: "gh-ost"         # `online-ddl-scheme` 已被弃用，建议使用 `online-ddl`。
clean-dump-file: true               # 是否清理 dump 阶段产生的文件，包括 metadata 文件、 建库建表 SQL
    ↪ 文件以及数据导入 SQL 文件
collation_compatible: "loose"       # 同步 CREATE 语句中缺省 Collation 的方式，可选 "loose" 和 "
    ↪ strict"，默认为 "loose"。"loose" 模式不会显式补充上游缺省的 Collation，"strict"
    ↪ 会显式补充上游缺省的 Collation。当使用 "strict" 模式，但下游不支持上游缺省的 Collation 时
    ↪ ，下游可能会报错。
ignore-checking-items: []           # 忽略检查项。可用值请参考 precheck 说明：https://docs.pingcap.
    ↪ com/zh/tidb/stable/dm-precheck。

target-database:                    # 下游数据库实例配置
  host: "192.168.0.1"
  port: 4000
  user: "root"
  password: "/Q7B9DizNLLTTfiZHv9WoEAKamfpIUs=" # 推荐使用经 `dmctl encrypt` 加密后的密码
  max-allowed-packet: 67108864                 # 设置 DM 内部连接 TiDB 服务器时，TiDB 客户端的 "
      ↪ max_allowed_packet" 限制（即接受的最大数据包限制），单位为字节，默认 67108864 (64 MB)
                                               # 该配置项从 DM v2.0.0 起弃用，DM 会自动获取连接
                                                   ↪ TiDB 的 "max_allowed_packet"
```

```
  session:                                    # 设置 TiDB 的 session 变量，在 v1.0.6 版本引入。
      ↪ 更多变量及解释参见 `https://docs.pingcap.com/zh/tidb/stable/system-variables`
    sql_mode: "ANSI_QUOTES,NO_ZERO_IN_DATE,NO_ZERO_DATE" # 从 DM v2.0.0 起，
        ↪ 如果配置文件中没有出现该项，DM 会自动从下游 TiDB 中获得适合用于 "sql_mode" 的值。
        ↪ 手动配置该项具有更高优先级
    tidb_skip_utf8_check: 1                           # 从 DM v2.0.0 起，
        ↪ 如果配置文件中没有出现该项，DM 会自动从下游 TiDB 中获得适合用于 "tidb_skip_utf8_check
        ↪ " 的值。手动配置该项具有更高优先级
    tidb_constraint_check_in_place: 0
  security:                       # 下游 TiDB TLS 相关配置
    ssl-ca: "/path/to/ca.pem"
    ssl-cert: "/path/to/cert.pem"
    ssl-key: "/path/to/key.pem"

###### ******** 功能配置集 **********

routes:                           # 上游和下游表之间的路由 table routing 规则集
  route-rule-1:                   # 配置名称
    schema-pattern: "test_*"   # 库名匹配规则，支持通配符 "*" 和 "?"
    table-pattern: "t_*"     # 表名匹配规则，支持通配符 "*" 和 "?"
    target-schema: "test"       # 目标库名称
    target-table: "t"             # 目标表名称
    # 可选配置：提取各分库分表的源信息，并写入下游用户自建的列，用于标识合表中各行数据的来源。
        ↪ 如果配置该项，需要提前在下游手动创建合表，具体可参考 “table routing 文档” <https://
        ↪ docs.pingcap.com/zh/tidb/dev/dm-key-features#table-routing>。
    # extract-table:                                  # 提取分表去除 t_ 的后缀信息，
        ↪ 并写入下游合表 c_table 列，例如，t_01 分表的数据会提取 01 写入下游 c_table 列
    #   table-regexp: "t_(.*)"
    #   target-column: "c_table"
    # extract-schema:                                 # 提取分库去除 test_ 的后缀信息，
        ↪ 并写入下游合表 c_schema 列，例如，test_02 分库的数据会提取 02 写入下游 c_schema 列
    #   schema-regexp: "test_(.*)"
    #   target-column: "c_schema"
     # extract-source:                                 # 提取数据库源实例信息写入 c_source
         ↪ 列，例如，mysql-replica-01 数据源实例的数据会提取 mysql-replica-01 写入下游 c_source
        ↪ 列
    #   source-regexp: "(.*)"
    #   target-column: "c_source"
  route-rule-2:
    schema-pattern: "test_*"
    target-schema: "test"

filters:                                  # 上游数据库实例匹配的表的 binlog event filter
    ↪ 规则集
  filter-rule-1:                          # 配置名称
```

```
    schema-pattern: "test_*"                    # 库名匹配规则，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                        # 表名匹配规则，支持通配符 "*" 和 "?"
    events: ["truncate table", "drop table"]    # 匹配哪些 event 类型
    action: Ignore                              # 对与符合匹配规则的 binlog 迁移（Do）还是忽略(
        ↪ Ignore)
  filter-rule-2:
    schema-pattern: "test_*"
    events: ["all dml"]
    action: Do

expression-filter:                      # 定义数据源迁移行变更的过滤规则，可以定义多个规则
  # 过滤 `expr_filter`.`tbl` 的 c 为偶数的插入
  even_c:                               # 规则名称
    schema: "expr_filter"               # 要匹配的上游数据库库名，不支持通配符匹配或正则匹配
    table: "tbl"                        # 要匹配的上游表名，不支持通配符匹配或正则匹配
    insert-value-expr: "c % 2 = 0"

block-allow-list:                       # 定义数据源迁移表的过滤规则，可以定义多个规则。如果 DM
    ↪ 版本早于 v2.0.0-beta.2 则使用 black-white-list
  bw-rule-1:                            # 规则名称
    do-dbs: ["~^test.*", "user"]   # 迁移哪些库
    ignore-dbs: ["mysql", "account"] # 忽略哪些库
    do-tables:                          # 迁移哪些表
    - db-name: "~^test.*"
      tbl-name: "~^t.*"
    - db-name: "user"
      tbl-name: "information"
  bw-rule-2:                            # 规则名称
    ignore-tables:                      # 忽略哪些表
    - db-name: "user"
      tbl-name: "log"

mydumpers:                              # dump 处理单元的运行配置参数
  global:                               # 配置名称
    threads: 4                          # dump 处理单元从上游数据库实例导出数据和 check-task
        ↪ 访问上游的线程数量，默认值为 4
    chunk-filesize: 64                  # dump 处理单元生成的数据文件大小，默认值为 64，单位为 MB
    extra-args: "--consistency none" # dump 处理单元的其他参数，不需要在 extra-args 中配置 table-
        ↪ list，DM 会自动生成

loaders:                                # load 处理单元的运行配置参数
  global:                               # 配置名称
    pool-size: 16                       # load 处理单元并发执行 dump 处理单元的 SQL 文件的线程数量，
        ↪ 默认值为 16，当有多个实例同时向 TiDB 迁移数据时可根据负载情况适当调小该值
```

```
# 保存上游全量导出数据的目录。该配置项的默认值为 "./dumped_data"。
# 支持配置为本地文件系统路径，也支持配置为 Amazon S3 路径，如：s3://dm_bucket/dumped_data?
    ↪ endpoint=s3-website.us-east-2.amazonaws.com&access_key=s3accesskey&secret_access_key=
    ↪ s3secretkey&force_path_style=true
dir: "./dumped_data"

# 全量阶段数据导入的模式。可以设置为如下几种模式：
# - "logical"(默认)。使用 TiDB Lightning logical import 进行导入。文档：https://docs.pingcap.
    ↪ com/zh/tidb/stable/tidb-lightning-logical-import-mode
# - "physical"。使用 TiDB Lightning physical import 进行导入。文档：https://docs.pingcap.com/
    ↪ zh/tidb/stable/tidb-lightning-physical-import-mode
#   当前 "physical" 为实验特性，不建议在生产环境中使用。
import-mode: "logical"
# logical import 针对冲突数据的解决方式：
# - "replace"（默认值）。表示用最新数据替代已有数据。
# - "ignore"。保留已有数据，忽略新数据。
# - "error"。插入重复数据时报错并停止同步任务。
on-duplicate-logical: "replace"
# physical import 针对冲突数据的解决方式：
# - "none"（默认）。对应 TiDB Lightning physical import 冲突数据检测的 "none" 选项
#  (https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-physical-import-mode-usage#
    ↪ 冲突数据检测)，
# 表示遇到冲突数据时不进行处理。该模式性能最佳，但下游数据库会遇到数据索引不一致的问题。
# - "manual"。对应 TiDB Lightning physical import 冲突数据检测的 "remove" 选项
#  (https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-physical-import-mode-usage#
    ↪ 冲突数据检测)。
# 在遇到冲突数据时将所有相互冲突的数据删除，并记录在 ${meta-schema}_${name}.conflict_error_v1
    ↪ 表中。
# 在本配置文件中，会记录在 dm_meta_test.conflict_error_v1 表中。全量导入阶段结束后，任务
# 会暂停并提示用户查询这张表并按照文档进行手动处理。使用 resume-task 命令让任务恢复运行并
# 进入到增量同步阶段。
on-duplicate-physical: "none"
# physical import 用作本地排序的目录位置，该选项的默认值与 dir 配置项一致。具体说明可以参见
    ↪ TiDB Lightning 对存储空间的需求：https://docs.pingcap.com/zh/tidb/stable/tidb-
    ↪ lightning-physical-import-mode#运行环境需求
sorting-dir-physical: "./dumped_data"
# 磁盘空间限制，对应 TiDB Lightning disk-quota 配置。具体说明参见文档：https://docs.pingcap.
    ↪ com/zh/tidb/stable/tidb-lightning-physical-import-mode-usage#磁盘资源配额-从-v620-
    ↪ 版本开始引入
disk-quota-physical: "0"
# physical import 在导入完成一张表后，对每一个表执行 `ADMIN CHECKSUM TABLE <table>`
    ↪ 进行数据校验的配置：
# - "required"（默认值）。表示导入完成后进行数据校验，如果校验失败会让任务暂停，
    ↪ 需要用户手动处理。
# - "optional"。表示导入完成后进行数据校验，如果校验失败会打印 warn 日志，任务不会暂停。
```

```
    # - "off"。表示导入完成后不进行数据校验。
    # Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
    checksum-physical: "required"
    # 配置在 CHECKSUM 结束后是否对所有表执行 `ANALYZE TABLE <table>` 操作。
    # - "required"（默认值）。表示导入完成后进行 ANALYZE 操作，ANALYZE 操作失败时任务暂停，
        ↪ 需要用户手动处理。
    # - "optional"。表示导入完成后进行 ANALYZE 操作，ANALYZE 操作失败时输出警告日志，任务不会暂停
        ↪ 。
    # - "off"。表示导入完成后不进行 ANALYZE 操作。
    # ANALYZE 只影响统计数据，在大部分场景下建议不开启 ANALYZE。
    analyze: "off"
    # Physical Import Mode 向 TiKV 写入 KV 数据的并发度。当 dm-worker 和 TiKV
        ↪ 网络传输速度超过万兆时，可适当增加这个值。
    # range-concurrency: 16
    # Physical Import Mode 向 TiKV 发送 KV 数据时是否启用压缩。目前仅支持 Gzip 压缩算法，可填写 "
        ↪ gzip" 或 "gz"。默认不启用压缩。
    # compress-kv-pairs: ""
    # PD server 的地址，填一个即可。该值为空时，默认使用 TiDB 查询到的 PD 地址信息。
    # pd-addr: "192.168.0.1:2379"

syncers:                            # sync 处理单元的运行配置参数
  global:                           # 配置名称
    worker-count: 16                # 应用已传输到本地的 binlog 的并发线程数量，默认值为 16。
        ↪ 调整此参数不会影响上游拉取日志的并发，但会对下游产生显著压力。
    batch: 100                      # sync 迁移到下游数据库的一个事务批次 SQL 语句数，默认值为
        ↪ 100，建议一般不超过 500。
    enable-ansi-quotes: true        # 若 `session` 中设置 `sql-mode: "ANSI_QUOTES"`，
        ↪ 则需开启此项

    # 设置为 true，则将来自上游的 `INSERT` 改写为 `REPLACE`，将 `UPDATE` 改写为 `DELETE` 与 `
        ↪ REPLACE`，保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入 DML。
    safe-mode: false
    # 自动安全模式的持续时间
    # 如不设置或者设置为 ""，则默认为 `checkpoint-flush-interval`（默认为 30s）的两倍，即 60s。
    # 如设置为 "0s"，则在 DM 自动进入安全模式的时候报错。
    # 如设置为正常值，例如 "1m30s"，则在该任务异常暂停、记录 `safemode_exit_point` 失败、或是 DM
        ↪ 进程异常退出时，把安全模式持续时间调整为 1 分 30 秒。详情可见[自动开启安全模式](https
        ↪ ://docs.pingcap.com/zh/tidb/stable/dm-safe-mode#自动开启)。
    safe-mode-duration: "60s"
    # 设置为 true，DM 会在不增加延迟的情况下，尽可能地将上游对同一条数据的多次操作压缩成一次操作
        ↪ 。
    # 如 INSERT INTO tb(a,b) VALUES(1,1); UPDATE tb SET b=11 WHERE a=1; 会被压缩成 INSERT INTO tb
        ↪ (a,b) VALUES(1,11); 其中 a 为主键
    # 如 UPDATE tb SET b=1 WHERE a=1; UPDATE tb(a,b) SET b=2 WHERE a=1; 会被压缩成 UPDATE tb(a,b)
        ↪  SET b=2 WHERE a=1; 其中 a 为主键
```

```
    # 如 DELETE FROM tb WHERE a=1; INSERT INTO tb(a,b) VALUES(1,1); 会被压缩成 REPLACE INTO tb(a,
        ↪ b) VALUES(1,1); 其中 a 为主键
    compact: false
    # 设置为 true，DM 会尽可能地将多条同类型的语句合并到一条语句中，生成一条带多行数据的 SQL 语句
        ↪ 。
    # 如 INSERT INTO tb(a,b) VALUES(1,1); INSERT INTO tb(a,b) VALUES(2,2); 会变成 INSERT INTO tb(
        ↪ a,b) VALUES(1,1),(2,2);
    # 如 UPDATE tb SET b=11 WHERE a=1; UPDATE tb(a,b) set b=22 WHERE a=2; 会变成 INSERT INTO tb(a
        ↪ ,b) VALUES(1,11),(2,22) ON DUPLICATE KEY UPDATE a=VALUES(a), b=VALUES(b); 其中 a
        ↪ 为主键
    # 如 DELETE FROM tb WHERE a=1; DELETE FROM tb WHERE a=2 会变成 DELETE FROM tb WHERE (a) IN
        ↪ (1),(2); 其中 a 为主键
    multiple-rows: false

validators:               # 增量数据校验的运行配置参数
  global:                 # 配置名称
    # full：校验每一行中每一列数据是否正确
    # fast：仅校验这一行是否有成功迁移到下游
    # none：不校验
    mode: full            # 可选填 full，fast 和 none，默认是 none，即不开启校验。
    worker-count: 4       # 后台校验的 validation worker 数量，默认是 4 个
    row-error-delay: 30m # 某一行多久没有校验通过会被标记为 error row，默认是 30 分钟

#####             实例配置
mysql-instances:          ------------
  -
    source-id: "mysql-replica-01"           # 对应 source.toml 中的 `source-id`
    meta:                                   # `task-mode` 为 `incremental` 且下游数据库的 `
        ↪ checkpoint` 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `checkpoint`
        ↪ 为准。如果 `meta` 项和下游数据库的 `checkpoint` 都不存在，则从上游当前最新的 binlog
        ↪ 位置开始迁移
      binlog-name: binlog.000001
      binlog-pos: 4
      binlog-gtid:  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7-8352-203
          ↪ db246dd3d:1-170" # 对于 source 中指定了 `enable-gtid: true` 的增量任务，
          ↪ 需要指定该值

    route-rules: ["route-rule-1", "route-rule-2"]     # 该上游数据库实例匹配的表到下游数据库的
        ↪ table routing 规则名称
    filter-rules: ["filter-rule-1", "filter-rule-2"] # 该上游数据库实例匹配的表的 binlog event
        ↪ filter 规则名称
    block-allow-list:  "bw-rule-1"                    # 该上游数据库实例匹配的表的 block-allow-
        ↪ list 过滤规则名称，如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
    expression-filters: ["even_c"]                    # 使用名为 even_c 的表达式过滤规则
```

```
  mydumper-config-name: "global"          # mydumpers 配置的名称
  loader-config-name: "global"            # loaders 配置的名称
  syncer-config-name: "global"            # syncers 配置的名称
  validator-config-name: "global"         # validators 配置的名称
-
  source-id: "mysql-replica-02" # 对应 source.toml 中的 `source-id`
  mydumper-thread: 4              # dump 处理单元用于导出数据的线程数量，等同于 mydumpers
      ′→ 配置中的 `threads`，当同时指定它们时 `mydumper-thread` 优先级更高
   loader-thread: 16               # load 处理单元用于导入数据的线程数量，等同于 loaders 配置中的
      ′→  `pool-size`，当同时指定它们时 `loader-thread` 优先级更高。当有多个实例同时向 TiDB
      ′→ 迁移数据时可根据负载情况适当调小该值
  syncer-thread: 16               # sync 处理单元用于复制增量数据的线程数量，等同于 syncers
      ′→ 配置中的 `worker-count`，当同时指定它们时 `syncer-thread` 优先级更高。
      ′→ 当有多个实例同时向 TiDB 迁移数据时可根据负载情况适当调小该值
```

配置顺序

通过上面的配置文件示例，可以看出配置文件总共分为两个部分：全局配置和实例配置，其中全局配置又分为基本信息配置和实例配置，配置顺序如下：

1. 编辑全局配置。
2. 根据全局配置编辑实例配置。

全局配置

任务基本信息配置

配置任务的基本信息，配置项的说明参见以上示例配置文件中的注释。其中 task-mode 需要特殊说明：

task-mode

- 描述：任务模式，可以通过任务模式来指定需要执行的数据迁移工作。
- 值为字符串（full，incremental 或 all）。
  - full：只全量备份上游数据库，然后将数据全量导入到下游数据库。
  - incremental：只通过 binlog 把上游数据库的增量修改复制到下游数据库，可以设置实例配置的 meta 配置项来指定增量复制开始的位置。
  - all：full + incremental。先全量备份上游数据库，将数据全量导入到下游数据库，然后从全量数据备份时导出的位置信息(binlog position) 开始通过binlog 增量复制数据到下游数据库。

功能配置集

全局配置主要包含下列功能配置集：

配置项	说明
routes	上游和下游表之间的路由table routing 规则集。如果上游与下游的库名、表名一致，则不需要配置该项。使用场景及示例配置参见Table Routing

配置项	说明
filters	上游数据库实例匹配的表的binlog event filter 规则集。如果不需要对binlog 进行过滤，则不需要配置该项。使用场景及示例配置参见Binlog Event Filter
block-allow-list	该上游数据库实例匹配的表的block & allow lists 过滤规则集。建议通过该项指定需要迁移的库和表，否则会迁移所有的库和表。使用场景及示例配置参见Block & Allow Lists
mydumpers	dump 处理单元的运行配置参数。如果默认配置可以满足需求，则不需要配置该项，也可以只使用 mydumper-thread 对 thread 配置项单独进行配置。
loaders	load 处理单元的运行配置参数。如果默认配置可以满足需求，则不需要配置该项，也可以只使用 loader-thread 对 pool-size 配置项单独进行配置。
syncers	sync 处理单元的运行配置参数。如果默认配置可以满足需求，则不需要配置该项，也可以只使用 syncer-thread 对 worker-count 配置项单独进行配置。

各个功能配置集的参数及解释参见完整配置文件示例中的注释说明。

实例配置

本小节定义具体的数据迁移子任务，DM 支持从单个或者多个上游 MySQL 实例迁移数据到同一个下游数据库实例。

在该项配置中设置数据迁移子任务中各个功能对应的配置集中的配置名称，关于这些配置项的更多配置细节，参见功能配置集的相关配置项，对应关系如下：

配置项	相关配置项
route-rules	routes
filter-rules	filters
block-allow-list	block-allow-list
mydumper-config-name	mydumpers
loader-config-name	loaders
syncer-config-name	syncers

10.6.9.4.4 DM-master 配置文件介绍

本文介绍 DM-master 的配置文件，包括配置文件示例与配置项说明。

配置文件示例

DM-master 的示例配置文件如下所示：

```
name = "dm-master"
```

```
##### log configuration
log-level = "info"
log-file = "dm-master.log"

##### DM-master listening address
master-addr = ":8261"
advertise-addr = "127.0.0.1:8261"

##### URLs for peer traffic
peer-urls = "http://127.0.0.1:8291"
advertise-peer-urls = "http://127.0.0.1:8291"

##### cluster configuration
initial-cluster = "master1=http://127.0.0.1:8291,master2=http://127.0.0.1:8292,master3=http
    ↪ ://127.0.0.1:8293"
join = ""

ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/cert.pem"
ssl-key = "/path/to/key.pem"
cert-allowed-cn = ["dm"]
```

配置项说明

Global 配置

配置项	说明
name	标识一个 DM-master。
log-level	日志级别：debug、info、warn、error、fatal。默认为 info。
log-file	日志文件，如果不配置，日志会输出到标准输出中。
master-addr	DM-master 服务的地址，可以省略IP 信息，例如：“:8261”。
advertise-addr	DM-master 向外界宣告的地址。
peer-urls	DM-master 节点的对等 URL。
advertise-peer-urls	DM-master 向外界宣告的对等 URL。默认为 peer-urls 的值。
initial-cluster	初始集群中所有 DM-master 的 advertise-peer-urls 的值。
join	集群里已有的 DM-master 的 advertise-peer-urls 的值。如果是新加入的 DM-master 节点，使用 join 替代 initial-cluster。
ssl-ca	DM-master 组件用于与其它组件连接的 SSL CA 证书所在的路径
ssl-cert	DM-master 组件用于与其它组件连接的PEM 格式的 X509 证书所在的路径

配置项	说明
ssl-key	DM-master 组件用于与其它组件连接的PEM 格式的 X509 密钥所在的路径
cert-allowed-cn	证书检查 Common Name 列表

10.6.9.4.5 DM-worker 配置文件介绍

本文介绍 DM-worker 的配置文件，包括配置文件示例与配置项说明。

配置文件示例

```
##### Worker Configuration.

name = "worker1"

##### Log configuration.
log-level = "info"
log-file = "dm-worker.log"

##### DM-worker listen address.
worker-addr = ":8262"
advertise-addr = "127.0.0.1:8262"
join = "http://127.0.0.1:8261,http://127.0.0.1:8361,http://127.0.0.1:8461"

keepalive-ttl = 60
relay-keepalive-ttl = 1800 # 版本 2.0.2 新增
##### relay-dir = "relay_log" # 版本 5.4.0 新增。使用相对路径时注意结合部署、启动方式确认路径位置
    ↪ 。

ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/cert.pem"
ssl-key = "/path/to/key.pem"
cert-allowed-cn = ["dm"]
```

配置项说明

Global 配置

配置项	说明
name	标识一个 DM-worker。
log-level	日志级别：debug、 info、 warn、 error、 fatal。默认为 info。
log-file	日志文件，如果不配置日志会输出到标准输出中。
worker-addr	DM-worker 服务的地址，可以省略IP 信息，例如：“:8262”。
advertise-addr	DM-worker 向外界宣告的地址。

配置项	说明
join	对应一个或多个 DM-master 配置中的master-addr。
keepalive-ttl	当绑定的上游数据源没有启用relay log 时，DM-worker 向 DM-master 保持存活的周期，单位为秒。默认是 60 秒。
relay-keepalive ′→ -ttl	当绑定的上游数据源启用relay log 时，DM-worker 向 DM-master 保持存活的周期，单位为秒。默认是 1800 秒。在版本 2.0.2 新增。
relay-dir	当绑定的上游数据源启用relay log 时，DM-worker 将 relay log 保存在该路径下。该配置优先级比上游数据源配置更高。在版本 5.4.0 新增。
ssl-ca	DM-worker 组件用于与其它组件连接的 SSL CA 证书所在的路径
ssl-cert	DM-worker 组件用于与其它组件连接的PEM 格式的 X509 证书所在的路径
ssl-key	DM-worker 组件用于与其它组件连接的PEM 格式的 X509 密钥所在的路径
cert-allowed-cn	证书检查 Common Name 列表

10.6.9.4.6 DM Table Selector

Table Selector 提供了一种基于通配符来匹配指定 schema/table 的功能。

通配符

table selector 在 schema-pattern/table-pattern 中可以使用以下两个通配符：

- 星号(*)
  - 匹配零个或者多个字符。例如，doc* 匹配 doc 和 document，但是不匹配 dodo。
  - * 只能放在pattern 的最后一位，例如，支持 doc*，但是不支持 do*c。
- 问号(?)
  - 匹配任意一个空字符除外的字符。

匹配规则

- schema-pattern 限制不能为空；
- table-pattern 可以设置为空。设置为空时，将只根据 schema-pattern 对 schema 进行匹配，然后返回匹配结果；
- table-pattern 不为空时，分别根据 schema-pattern 和 table-pattern 进行匹配，两个都匹配则结果为匹配。

使用示例

- 匹配所有库名以 schema_ 开头的 schema 和table

```
schema-pattern: "schema_*"
table-pattern: ""
```

- 匹配所有库名以 schema_ 为前缀，并且表名以 table_ 前缀的表

```
schema-pattern = "schema_*"
table-pattern = "table_*"
```

10.6.9.5 使用 OpenAPI 运维 TiDB Data Migration 集群

TiDB Data Migration (DM) 提供 OpenAPI 功能，你可以通过 OpenAPI 方便地对 DM 集群进行查询和运维操作。OpenAPI 的功能范围和dmctl 工具相当。

如需开启 OpenAPI，可通过以下方法：

- 如果你的 DM 集群是通过二进制直接部署的，则在 DM-master 的配置文件中添加如下配置：

```
openapi = true
```

- 如果你的 DM 集群是通过TiUP 部署的，则在拓扑文件中添加如下配置：

```
server_configs:
  master:
    openapi: true
```

> 注意：
>
> - DM 提供符合 OpenAPI 3.0.0 标准的 Spec 文档，其中包含了所有API 的请求参数和返回体，你可自行复制到如 Swagger Editor 等工具中在线预览文档。
> - 部署 DM-master 后，你可访问 http://{master-addr}/api/v1/docs 在线预览文档。

你可以通过 OpenAPI 完成 DM 集群的如下运维操作：

10.6.9.5.1 集群相关API

- 获取 DM-master 节点信息
- 下线 DM-master 节点
- 获取 DM-worker 节点信息
- 下线 DM-worker 节点

10.6.9.5.2 数据源相关API

- 创建数据源
- 获取数据源
- 删除数据源
- 更新数据源
- 启用数据源
- 停用数据源
- 获取数据源状态
- 获取数据源列表
- 对数据源开启relay-log 功能
- 对数据源停止relay-log 功能
- 清除数据源不需要的relay-log 文件
- 更改数据源和 DM-worker 的绑定关系
- 获取数据源的数据库名列表
- 获取数据源的指定数据库的表名列表

10.6.9.5.3 同步任务相关API

- 创建同步任务
- 获取同步任务
- 删除同步任务
- 更新同步任务
- 开始同步任务
- 停止同步任务
- 获取同步任务状态
- 获取同步任务列表
- 获取同步任务的同步规则列表
- 获取同步任务关联数据源的数据库名列表
- 获取同步任务关联数据源的数据表名列表
- 获取同步任务关联数据源的数据表的创建语句
- 更新同步任务关联数据源的数据表的创建语句
- 删除同步任务关联数据源的数据表

本文档以下部分描述当前提供的API 的具体使用方法。

10.6.9.5.4 API 统一错误格式

对API 发起的请求后，如发生错误，返回错误信息的格式如下所示：

```
{
    "error_msg": "",
    "error_code": ""
}
```

如上所示，`error_msg` 描述错误信息，`error_code` 则是对应的错误码。

10.6.9.5.5 获取 DM-master 节点信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/cluster/masters
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/cluster/masters' \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "name": "master1",
      "alive": true,
      "leader": true,
      "addr": "127.0.0.1:8261"
    }
  ]
}
```

10.6.9.5.6 下线 DM-master 节点

该接口是一个同步接口，请求成功后返回体的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/cluster/masters/{master-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/cluster/masters/master1' \
  -H 'accept: */*'
```

10.6.9.5.7 获取 DM-worker 节点信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/cluster/workers
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/cluster/workers' \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "name": "worker1",
      "addr": "127.0.0.1:8261",
      "bound_stage": "bound",
      "bound_source_name": "mysql-01"
    }
  ]
}
```

10.6.9.5.8 下线 DM-worker 节点

该接口是一个同步接口，请求成功后返回体的 Status Code 是 204。

请求 URI

```
DELETE  /api/v1/cluster/workers/{worker-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/cluster/workers/worker1' \
  -H 'accept: */*'
```

10.6.9.5.9 创建数据源

该接口是一个同步接口，请求成功会返回对应数据源信息。

请求 URI

```
POST /api/v1/sources
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
```

```
  "user": "root",
  "password": "123456",
  "enable": true,
  "enable_gtid": false,
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge":
    { "interval":
    3600,
    "expires": 0,
    "remain_space": 15
  }
}'
```

```
{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
  "user": "root",
  "password": "123456",
  "enable": true,
  "enable_gtid": false,
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge":
    { "interval":
    3600,
    "expires": 0,
    "remain_space": 15
  },
  "status_list": [
    {
      "source_name": "mysql-replica-01",
      "worker_name": "worker-1",
      "relay_status": {
```

```
        "master_binlog": "(mysql-bin.000001, 1979)",
        "master_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_dir": "./sub_dir",
        "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_catch_up_master": true,
        "stage": "Running"
      },
      "error_msg": "string"
    }
  ]
}
```

10.6.9.5.10 获取数据源

该接口是一个同步接口，请求成功会返回数据源列表信息。

请求 URI

```
GET /api/v1/sources/{source-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/source-1?with_status=true' \
  -H 'accept: application/json'
```

```
{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
  "user": "root",
  "password": "123456",
  "enable_gtid": false,
  "enable": false,
  "flavor": "mysql",
  "task_name_list": [
    "task1"
  ],
  "security":
    { "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge": {
```

```
    "interval": 3600,
    "expires": 0,
    "remain_space": 15
  },
  "status_list": [
    {
      "source_name": "mysql-replica-01",
      "worker_name": "worker-1",
      "relay_status": {
        "master_binlog": "(mysql-bin.000001, 1979)",
        "master_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_dir": "./sub_dir",
        "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_catch_up_master": true,
        "stage": "Running"
      },
      "error_msg": "string"
    }
  ],
  "relay_config":
    { "enable_relay":
    true,
    "relay_binlog_name": "mysql-bin.000002",
    "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
    "relay_dir": "./relay_log"
  }
}
```

10.6.9.5.11 删除数据源

该接口是一个同步接口，请求成功后返回的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/sources/{source-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01?force=true' \
  -H 'accept: application/json'
```

10.6.9.5.12 更新数据源

该接口是一个同步接口，请求成功会返回对应的数据源信息。

> **注意：**
>
> 更新数据源配置时，须确保当前数据源下没有任何正在运行的同步任务。

请求 URI

```
PUT /api/v1/sources/{source-name}
```

使用样例

```
curl -X 'PUT' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "source": {
    "source_name": "mysql-01",
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "enable_gtid": false,
    "enable": false,
    "flavor": "mysql",
    "task_name_list": [
      "task1"
    ],
    "security":
      { "ssl_ca_content": "",
      "ssl_cert_content": "",
      "ssl_key_content": "",
      "cert_allowed_cn": [
        "string"
      ]
    },
    "purge":
      { "interval":
      3600,
      "expires": 0,
      "remain_space": 15
    },
    "relay_config":
      { "enable_relay": true,
      "relay_binlog_name": "mysql-bin.000002",
      "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
      "relay_dir": "./relay_log"
```

```
    }
  }
}'
```

```
{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
  "user": "root",
  "password": "123456",
  "enable": true,
  "enable_gtid": false,
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge":
    { "interval":
    3600,
    "expires": 0,
    "remain_space": 15
  }
}
```

10.6.9.5.13 启用数据源

这是一个同步接口，请求成功后会启用此数据源，并批量开始数据迁移任务中依赖该数据源的所有子任务。

请求 URI

```
POST /api/v1/sources/{source-name}/enable
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/enable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json'
```

10.6.9.5.14 停用数据源

这是一个同步接口，请求成功后会停用此数据源，并批量停止数据迁移任务中依赖该数据源的所有子任务。

请求 URI

```
POST /api/v1/sources/{source-name}/disable
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/disable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json'
```

10.6.9.5.15 获取数据源列表

该接口是一个同步接口，请求成功会返回数据源列表信息。

请求 URI

```
GET /api/v1/sources
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources?with_status=true' \
  -H 'accept: application/json'
```

```
{
  "data": [
    {
      "enable_gtid": false,
      "host": "127.0.0.1",
      "password": "******",
      "port": 3306,
      "purge": {
        "expires": 0,
        "interval": 3600,
        "remain_space": 15
      },
      "security": null,
      "source_name": "mysql-01",
      "user": "root"
    },
    {
      "enable_gtid": false,
      "host": "127.0.0.1",
      "password": "******",
      "port": 3307,
      "purge": {
        "expires": 0,
        "interval": 3600,
        "remain_space": 15
```

```
      },
      "security": null,
      "source_name": "mysql-02",
      "user": "root"
    }
  ],
  "total": 2
}
```

10.6.9.5.16 获取数据源状态

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/sources/{source-name}/status
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-replica-01/status' \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "source_name": "mysql-replica-01",
      "worker_name": "worker-1",
      "relay_status": {
        "master_binlog": "(mysql-bin.000001, 1979)",
        "master_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_dir": "./sub_dir",
        "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_catch_up_master": true,
        "stage": "Running"
      },
      "error_msg": "string"
    }
  ]
}
```

10.6.9.5.17 对数据源开启relay-log 功能

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取数据源状态接口获取最新的状态。

请求 URI

POST /api/v1/sources/{source-name}/relay/enable

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/relay/enable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "worker_name_list":
    [ "worker-1"
  ],
  "relay_binlog_name": "mysql-bin.000002",
  "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
  "relay_dir": "./relay_log"
}'
```

10.6.9.5.18 对数据源停止relay-log功能

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取数据源状态接口获取最新的状态。

请求 URI

POST /api/v1/sources/{source-name}/relay/disable

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/relay/disable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "worker_name_list":
    [ "worker-1"
  ]
}'
```

10.6.9.5.19 清除数据源不需要的relay-log文件

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取数据源状态接口获取最新的状态。

请求 URI

POST /api/v1/sources/{source-name}/relay/purge

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/relay/purge' \
  -H 'accept: */*' \
```

```
  -H 'Content-Type: application/json' \
  -d '{
  "relay_binlog_name": "mysql-bin.000002",
  "relay_dir": "string"
}'
```

10.6.9.5.20 更改数据源和 DM-worker 的绑定关系

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取 DM-worker 节点信息接口获取最新的状态。

请求 URI

```
POST /api/v1/sources/{source-name}/transfer
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/transfer' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "worker_name": "worker-1"
}'
```

10.6.9.5.21 获取数据源的数据库名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET /api/v1/sources/{source-name}/schemas
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/source-1/schemas' \
  -H 'accept: application/json'
```

```
[
  "db1"
]
```

10.6.9.5.22 获取数据源的指定数据库的表名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET /api/v1/sources/{source-name}/schemas/{schema-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/source-1/schemas/db1'  \
  -H 'accept: application/json'
```

```
[
  "table1"
]
```

10.6.9.5.23 创建同步任务

这是一个同步接口，请求成功的 Status Code 是 200，请求成功会返回对应的同步任务信息。

请求 URI

```
POST /api/v1/tasks
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/tasks' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "task": {
    "name": "task-1",
    "task_mode": "all",
    "shard_mode": "pessimistic",
    "meta_schema": "dm-meta",
    "enhance_online_schema_change": true,
    "on_duplicate": "overwrite",
    "target_config": {
      "host": "127.0.0.1",
      "port": 3306,
      "user": "root",
      "password": "123456",
      "security": {
        "ssl_ca_content": "",
        "ssl_cert_content": "",
        "ssl_key_content": "",
        "cert_allowed_cn": [
          "string"
        ]
      }
    },
    "binlog_filter_rule":
      { "rule-1": {
```

```
      "ignore_event":
        [ "all dml"
      ],
      "ignore_sql":
        [ "^Drop"
      ]
    },
    "rule-2":
      { "ignore_event":
      [
        "all dml"
      ],
      "ignore_sql":
        [ "^Drop"
      ]
    },
    "rule-3":
      { "ignore_event":
      [
        "all dml"
      ],
      "ignore_sql":
        [ "^Drop"
      ]
    }
  },
  "table_migrate_rule": [
    {
      "source": {
        "source_name": "source-name",
        "schema": "db-*",
        "table": "tb-*"
      },
      "target":
        { "schema":
        "db1",
        "table": "tb1"
      },
      "binlog_filter_rule":
        [ "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
  ],
  "source_config":
    { "full_migrate_conf": {
      "export_threads": 4,
```

```
        "import_threads": 16,
        "data_dir": "./exported_data",
        "consistency": "auto"
      },
      "incr_migrate_conf":
        { "repl_threads": 16,
        "repl_batch": 100
      },
      "source_conf": [
        {
          "source_name": "mysql-replica-01",
          "binlog_name": "binlog.000001",
          "binlog_pos": 4,
          "binlog_gtid":  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
              ↪ -8352-203db246dd3d:1-170"
        }
      ]
    }
  }
}'
```

```
{
  "name": "task-1",
  "task_mode": "all",
  "shard_mode": "pessimistic",
  "meta_schema": "dm-meta",
  "enhance_online_schema_change": true,
  "on_duplicate": "overwrite",
  "target_config": {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "security": {
      "ssl_ca_content": "",
      "ssl_cert_content": "",
      "ssl_key_content": "",
      "cert_allowed_cn": [
        "string"
      ]
    }
  },
  "binlog_filter_rule":
    { "rule-1": {
      "ignore_event": [
```

```
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-2":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-3":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target":
      { "schema":
      "db1",
      "table": "tb1"
    },
    "binlog_filter_rule":
      [ "rule-1",
      "rule-2",
      "rule-3",
    ]
  }
],
"source_config":
  { "full_migrate_conf":
  {
    "export_threads": 4,
    "import_threads": 16,
```

```
      "data_dir": "./exported_data",
      "consistency": "auto"
    },
    "incr_migrate_conf":
      { "repl_threads": 16,
      "repl_batch": 100
    },
    "source_conf": [
      {
        "source_name": "mysql-replica-01",
        "binlog_name": "binlog.000001",
        "binlog_pos": 4,
        "binlog_gtid":  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
            ↪ -8352-203db246dd3d:1-170"
      }
    ]
  }
}
```

10.6.9.5.24 获取同步任务

这是一个同步接口，请求成功的 Status Code 是 200。

请求 URI

```
GET /api/v1/tasks/{task-name}?with_status=true
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1?with_status=true'  \
  -H 'accept: application/json'
```

```
{
  "name": "task-1",
  "task_mode": "all",
  "shard_mode": "pessimistic",
  "meta_schema": "dm-meta",
  "enhance_online_schema_change": true,
  "on_duplicate": "overwrite",
  "target_config": {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "security": {
      "ssl_ca_content": "",
```

```
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  }
},
"binlog_filter_rule":
  { "rule-1": {
    "ignore_event":
      [ "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-2":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-3":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target":
      { "schema":
      "db1",
      "table": "tb1"
    },
    "binlog_filter_rule": [
```

```
        "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
  ],
  "source_config":
    { "full_migrate_conf": {
      "export_threads": 4,
      "import_threads": 16,
      "data_dir": "./exported_data",
      "consistency": "auto"
    },
    "incr_migrate_conf":
      { "repl_threads": 16,
      "repl_batch": 100
    },
    "source_conf": [
      {
        "source_name": "mysql-replica-01",
        "binlog_name": "binlog.000001",
        "binlog_pos": 4,
        "binlog_gtid":  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
            ↪ -8352-203db246dd3d:1-170"
      }
    ]
  }
}
```

10.6.9.5.25 删除同步任务

该接口是一个同步接口，请求成功后返回的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/tasks/{task-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1' \
  -H 'accept: application/json'
```

10.6.9.5.26 更新同步任务

该接口是一个同步接口，请求成功会返回对应同步任务的信息。

> **注意：**
>
> 更新同步任务配置时，须确保该任务处于暂停状态，并已经运行到增量同步的阶段，且仅有部分字段可以更新。

请求 URI

```
PUT /api/v1/tasks/{task-name}
```

使用样例

```
curl -X 'PUT' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "task": {
    "name": "task-1",
    "task_mode": "all",
    "shard_mode": "pessimistic",
    "meta_schema": "dm-meta",
    "enhance_online_schema_change": true,
    "on_duplicate": "overwrite",
    "target_config": {
      "host": "127.0.0.1",
      "port": 3306,
      "user": "root",
      "password": "123456",
      "security": {
        "ssl_ca_content": "",
        "ssl_cert_content": "",
        "ssl_key_content": "",
        "cert_allowed_cn": [
          "string"
        ]
      }
    },
    "binlog_filter_rule":
      { "rule-1": {
        "ignore_event":
          [ "all dml"
        ],
        "ignore_sql":
          [ "^Drop"
        ]
```

```
  },
  "rule-2":
    { "ignore_event":
    [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-3":
    { "ignore_event":
    [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target":
      { "schema":
      "db1",
      "table": "tb1"
    },
    "binlog_filter_rule":
      [ "rule-1",
      "rule-2",
      "rule-3",
    ]
  }
],
"source_config":
  { "full_migrate_conf": {
    "export_threads": 4,
    "import_threads": 16,
    "data_dir": "./exported_data",
    "consistency": "auto"
  },
  "incr_migrate_conf":
    { "repl_threads": 16,
```

```
        "repl_batch": 100
      },
      "source_conf": [
        {
          "source_name": "mysql-replica-01",
          "binlog_name": "binlog.000001",
          "binlog_pos": 4,
          "binlog_gtid":  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
              '→ -8352-203db246dd3d:1-170"
        }
      ]
    }
  }
}'
```

```
{
  "name": "task-1",
  "task_mode": "all",
  "shard_mode": "pessimistic",
  "meta_schema": "dm-meta",
  "enhance_online_schema_change": true,
  "on_duplicate": "overwrite",
  "target_config": {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "security": {
      "ssl_ca_content": "",
      "ssl_cert_content": "",
      "ssl_key_content": "",
      "cert_allowed_cn": [
        "string"
      ]
    }
  },
  "binlog_filter_rule":
    { "rule-1": {
      "ignore_event":
        [ "all dml"
      ],
      "ignore_sql":
        [ "^Drop"
      ]
    },
```

```
  "rule-2":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-3":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target":
      { "schema":
      "db1",
      "table": "tb1"
    },
    "binlog_filter_rule":
      [ "rule-1",
      "rule-2",
      "rule-3",
    ]
  }
],
"source_config":
  { "full_migrate_conf":
  {
    "export_threads": 4,
    "import_threads": 16,
    "data_dir": "./exported_data",
    "consistency": "auto"
  },
  "incr_migrate_conf":
    { "repl_threads":
    16,
    "repl_batch": 100
```

```
    },
    "source_conf": [
      {
        "source_name": "mysql-replica-01",
        "binlog_name": "binlog.000001",
        "binlog_pos": 4,
        "binlog_gtid":  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
            ↪ -8352-203db246dd3d:1-170"
      }
    ]
  }
}
```

10.6.9.5.27 开始同步任务

这是一个异步接口，请求成功的 Status Code 是 200。可通过获取同步任务状态接口获取最新的任务状态。

请求 URI

```
POST /api/v1/tasks/{task-name}/start
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/start' \
  -H 'accept: */*'
```

10.6.9.5.28 停止同步任务

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取同步任务状态接口获取最新的任务状态。

请求 URI

```
POST /api/v1/tasks/{task-name}/stop
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/stop' \
  -H 'accept: */*'
```

10.6.9.5.29 获取同步任务状态

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/tasks/task-1/status
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/status?stage=running'  \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "name": "string",
      "source_name": "string",
      "worker_name": "string",
      "stage": "runing",
      "unit": "sync",
      "unresolved_ddl_lock_id": "string",
      "load_status": {
        "finished_bytes": 0,
        "total_bytes": 0, "progress":
        "string", "meta_binlog":
        "string", "meta_binlog_gtid":
        "string"
      },
      "sync_status":
        { "total_events": 0,
        "total_tps": 0,
        "recent_tps": 0,
        "master_binlog": "string",
        "master_binlog_gtid": "string",
        "syncer_binlog": "string",
        "syncer_binlog_gtid": "string",
        "blocking_ddls": [
          "string"
        ],
        "unresolved_groups": [
          {
            "target": "string",
            "ddl_list": [
              "string"
            ],
            "first_location": "string",
            "synced": [
              "string"
            ],
            "unsynced":
              [ "string"
```

```
            ]
          }
        ],
        "synced": true,
        "binlog_type": "string",
        "seconds_behind_master": 0
      }
    }
  ]
}
```

10.6.9.5.30 获取同步任务列表

该接口是一个同步接口，请求成功会返回对应的同步任务列表。

请求 URI

```
GET /api/v1/tasks
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks' \
  -H 'accept: application/json'
```

```
{
  "total": 2,
  "data": [
    {
      "name": "task-1",
      "task_mode": "all",
      "shard_mode": "pessimistic",
      "meta_schema": "dm-meta",
      "enhance_online_schema_change": true,
      "on_duplicate": "overwrite",
      "target_config": {
        "host": "127.0.0.1",
        "port": 3306,
        "user": "root",
        "password": "123456",
        "security": {
          "ssl_ca_content": "",
          "ssl_cert_content": "",
          "ssl_key_content": "",
          "cert_allowed_cn": [
            "string"
          ]
```

```
  }
},
"binlog_filter_rule":
  { "rule-1": {
    "ignore_event":
      [ "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-2":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  },
  "rule-3":
    { "ignore_event": [
      "all dml"
    ],
    "ignore_sql":
      [ "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target":
      { "schema":
      "db1",
      "table": "tb1"
    },
    "binlog_filter_rule":
      [ "rule-1",
      "rule-2",
      "rule-3",
    ]
  }
```

```
      ],
      "source_config":
        { "full_migrate_conf": {
          "export_threads": 4,
          "import_threads": 16,
          "data_dir": "./exported_data",
          "consistency": "auto"
        },
        "incr_migrate_conf":
          { "repl_threads": 16,
          "repl_batch": 100
        },
        "source_conf": [
          {
            "source_name": "mysql-replica-01",
            "binlog_name": "binlog.000001",
            "binlog_pos": 4,
            "binlog_gtid":  "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
                ↪ -8352-203db246dd3d:1-170"
          }
        ]
      }
    }
  ]
}
```

10.6.9.5.31 获取同步任务的同步规则列表

该接口是一个同步接口，请求成功会返回对应同步任务的同步规则列表。

请求 URI

```
GET  /api/v1/tasks/{task-name}/sources/{source-name}/migrate_targets
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/migrate_targets'  \
  -H 'accept: application/json'
```

```
{
  "total": 0,
  "data": [
    {
      "source_schema": "db1",
      "source_table": "tb1",
      "target_schema": "db1",
```

```
      "target_table": "tb1"
    }
  ]
}
```

10.6.9.5.32 获取同步任务关联数据源的数据库名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET  /api/v1/tasks/{task-name}/sources/{source-name}/schemas
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas'  \
  -H 'accept: application/json'
```

```
[
  "db1"
]
```

10.6.9.5.33 获取同步任务关联数据源的数据表名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET  /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas/db1'  \
  -H 'accept: application/json'
```

```
[
  "table1"
]
```

10.6.9.5.34 获取同步任务关联数据源的数据表的创建语句

该接口是一个同步接口，请求成功会返回对应的创建语句。

请求 URI

```
GET  /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}/{table-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas/db1/table1'  \
  -H 'accept: application/json'
```

```
{
  "schema_name": "db1",
  "table_name": "table1",
  "schema_create_sql": "CREATE TABLE `t1` (`id` int(11) NOT NULL AUTO_INCREMENT,PRIMARY KEY (`id
      ↪ `) /*T![clustered_index] CLUSTERED */) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=
      ↪ utf8mb4_bin"
}
```

10.6.9.5.35 更新同步任务关联数据源的数据表的创建语句

该接口是一个同步接口，返回体的 Status Code 是 200。

请求 URI

```
POST  /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}/{table-name}
```

使用样例

```
curl -X 'PUT' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/task-1/schemas/db1/table1'  \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
   "sql_content": "CREATE TABLE `t1` ( `c1` int(11) DEFAULT NULL, `c2` int(11) DEFAULT NULL, `c3`
       ↪ int(11) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;",
  "flush": true,
  "sync": true
}'
```

10.6.9.5.36 删除同步任务关联数据源的数据表

该接口是一个同步接口，返回体的 Status Code 是 200。

请求 URI

```
DELETE  /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}/{table-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas/db1/table1'  \
  -H 'accept: */*'
```

10.6.9.6 TiDB Data Migration 兼容性目录

DM 数据同步软件支持从不同类型的数据源迁移到TiDB 集群。针对各种数据源类型，产品支持程度可以分为四个级别：

- 正式支持：该场景经过验证，并且通过完整的测试流程。
- 实验支持：虽然通过部分验证，但测试尚未覆盖所有预设场景或用户较少，存在少量场景下可能出错的风险。
- 未测试：DM 在迭代过程中尽量保证 MySQL 协议的兼容性，但由于资源限制，无法测试所有 MySQL 衍生版本。因此虽然技术原理上兼容，但是并未经完整测试，需要使用前自行验证。
- 不兼容：已发现明确不兼容的情况，不建议在生产环境中使用。

10.6.9.6.1 数据源

	数据源	级别	备注
	MySQL ≤ 5.5	未测试	
	MySQL 5.6	正式支持	
	MySQL 5.7	正式支持	
	MySQL 8.0	实验支持	
	MariaDB < 10.1.2	不兼容	时间类型的binlog 不兼容
	MariaDB 10.1.2 ~ 10.5.10	实验支持	
	MariaDB > 10.5.10	不兼容	检查环节存在权限报错

10.6.9.6.2 目标数据库

> **警告：**
>
> 不建议使用 DM 5.3.0，因为当使用GTID 同步且未开启Relay log 的情况下，低概率会出现数据不同步。

目标数据库	级别	DM 版本
TiDB 6.0	正式支持	最低 5.3.1
TiDB 5.4	正式支持	最低 5.3.1
TiDB 5.3	正式支持	最低 5.3.1
TiDB 5.2	正式支持	最低 2.0.7，建议 5.4 版本
TiDB 5.1	正式支持	最低 2.0.4，建议 5.4 版本
TiDB 5.0	正式支持	最低 2.0.4，建议 5.4 版本
TiDB 4.x	正式支持	最低 2.0.1，建议 2.0.7 版本
TiDB 3.x	正式支持	最低 2.0.1，建议 2.0.7 版本
MySQL	实验支持	
MariaDB	实验支持	

10.6.9.7 安全

10.6.9.7.1 为 DM 的连接开启加密传输

本文介绍如何为 DM 的连接开启加密传输，包括 DM-master，DM-worker，dmctl 组件之间的连接以及 DM 组件与上下游数据库之间的连接。

为 DM-master，DM-worker，dmctl 组件之间的连接开启加密传输

本节介绍如何为 DM-master，DM-worker，dmctl 组件之间的连接开启加密传输。

配置开启加密传输

1. 准备证书。

   推荐为 DM-master、DM-worker 分别准备一个 Server 证书，并保证可以相互验证，而dmctl 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，cfssl 及 easy-rsa 等基于 openssl 的工具。

   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。

> 注意：
>
> DM-master、DM-worker 与 dmctl 三个组件可使用同一套证书。

- DM-master

  在 DM-master 配置文件或命令行参数中设置：

```
ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/master-cert.pem"
ssl-key = "/path/to/master-key.pem"
```

- DM-worker

  在 DM-worker 配置文件或命令行参数中设置：

```
ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/worker-cert.pem"
ssl-key = "/path/to/worker-key.pem"
```

- dmctl

  若 DM 集群各个组件间开启加密传输后，在使用dmctl 工具连接集群时，需要指定 Client 证书，示例如下：

```
./dmctl --master-addr=127.0.0.1:8261 --ssl-ca /path/to/ca.pem --ssl-cert /path/to/
    ↪ client-cert.pem --ssl-key /path/to/client-key.pem
```

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，还需要校验调用者身份以防止拥有有效证书的非法访问者进行访问（例如：DM-worker 只能被 DM-master 访问，需阻止拥有合法证书但非 DM-master 的其他访问者访问 DM-worker）。

如希望进行组件调用者身份认证，需要在生成证书时通过 Common Name (CN) 标识证书使用者身份，并在被调用者配置检查证书 Common Name 列表时检查调用者身份。

- DM-master

  在 config 文件或命令行参数中设置：

```
cert-allowed-cn = ["dm"]
```

- DM-worker

  在 config 文件或命令行参数中设置：

```
cert-allowed-cn = ["dm"]
```

证书重加载

DM-master、DM-worker 和 dmctl 都会在每次新建相互通讯的连接时重新读取当前的证书和密钥文件内容，实现证书和密钥的重加载。

当 ssl-ca、ssl-cert 或 ssl-key 的文件内容更新后，可通过重启 DM 组件使其重新加载证书与密钥内容并重新建立连接。

DM 组件与上下游数据库之间的连接开启加密传输

本节介绍如何为 DM 组件与上下游数据库之间的连接开启加密传输。

为上游数据库连接开启加密传输

1. 配置上游数据库，启用加密连接支持并设置 Server 证书，具体可参考 Using encrypted connections
2. 在 source 配置文件中设置 MySQL Client 证书：

> **注意：**
>
> 请确保所有 DM-master 与 DM-worker 组件能通过指定路径读取到证书与密钥文件的内容。

```
from:
    security:
        ssl-ca: "/path/to/mysql-ca.pem"
        ssl-cert: "/path/to/mysql-client-cert.pem"
        ssl-key: "/path/to/mysql-client-key.pem"
```

为下游TiDB 连接开启加密传输

1. 配置下游TiDB 启用加密连接支持，具体可参考配置TiDB 启用加密连接支持
2. 在 task 配置文件中设置TiDB Client 证书：

> 注意：
> 请确保所有 DM-master 与 DM-worker 组件能通过指定路径读取到证书与密钥文件的内容。

```
target-database:
    security:
        ssl-ca: "/path/to/tidb-ca.pem"
        ssl-cert: "/path/to/tidb-client-cert.pem"
        ssl-key: "/path/to/tidb-client-key.pem"
```

10.6.9.7.2 TiDB Data Migration 生成自签名证书

本文档提供使用 openssl 为 TiDB Data Migration (DM) 生成自签名证书的一个示例，你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	DM-master1
node2	172.16.10.12	DM-master2
node3	172.16.10.13	DM-master3
node4	172.16.10.14	DM-worker1
node5	172.16.10.15	DM-worker2
node6	172.16.10.16	DM-worker3

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。

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 CA 密钥：

```
openssl genrsa -out ca-key.pem 4096
```

2. 生成 CA 证书：

```
openssl req -new -x509 -days 1000 -key ca-key.pem -out ca.pem
```

3. 验证 CA 证书：

```
openssl x509 -text -in ca.pem -noout
```

签发各个组件的证书

集群中可能使用到的证书

- master certificate 由 DM-master 使用，为其他组件验证 DM-master 身份。
- worker certificate 由 DM-worker 使用，为其他组件验证 DM-worker 身份。
- client certificate 由 dmctl 使用，用于 DM-master、 DM-worker 验证客户端。

为 DM-master 签发证书

给 DM-master 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 master-key.pem 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，根据前述的集群拓扑编辑 Subject Alternative Name (SAN) 的信息：

```
[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 172.16.10.11
IP.3 = 172.16.10.12
IP.4 = 172.16.10.13
```

   目前支持以下 SAN 检查项：

   - IP

- DNS
- URI

> **注意：**
>
> 如果要使用 0.0.0.0 等特殊IP 用于连接通讯，也需要将其加入到 alt_names 中。

4. 保存 openssl.cnf 文件后，生成证书请求文件（在这一步中提供 Common Name (e.g. server FQDN or YOUR name)[]: 输入时，可以为该证书指定 Common Name (CN)，如 dm。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 master-key.pem -out master-cert.pem -config openssl.cnf
```

5. 签发生成证书：

```
openssl x509 -req -days 365 -CA ca.pem -CAkey ca-key.pem -CAcreateserial -in master-cert.pem
    -out master-cert.pem -extensions v3_req -extfile openssl.cnf
```

6. 验证证书携带 SAN 字段信息（可选）：

```
openssl x509 -text -in master-cert.pem -noout
```

7. 确认在当前目录下得到如下文件：

```
ca.pem
master-cert.pem
master-key.pem
```

> **注意：**
>
> 为 DM-worker 组件签发证书的过程类似，此文档不再赘述。

为 dmctl 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.pem 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.pem -out client-cert.pem
```

3. 签发生成证书：

```
openssl x509 -req -days 365 -CA ca.pem -CAkey ca-key.pem -CAcreateserial -in client-cert.pem
    ↪ -out client-cert.pem
```

10.6.9.8 监控告警

10.6.9.8.1 DM 监控指标

使用TiUP 部署 DM 集群的时候，会默认部署一套监控系统。

Task

在 Grafana dashboard 中，DM 默认名称为 DM-task。

Overview

overview 下包含运行当前选定task 的所有 DM-worker/master instance/source 的部分监控指标。当前默认告警规则只针对于单个 DM-worker/master instance/source。

metric 名称	说明	告警说明	告警级别
task state	迁移子任务的状态	N/A	N/A
storage capacity	relay log 占有的磁盘的总容量	N/A	N/A
storage remain	relay log 占有的磁盘的剩余可用容量	N/A	N/A
binlog file gap between master and relay	relay 与上游 master 相比落后的 binlog file 个数	N/A	N/A
load progress	load unit 导入过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
binlog file gap between master and syncer	与上游master 相比 binlog replication unit 落后的binlog file 个数	N/A	N/A

	metric 名称	说明	告警说明	告警级别
	shard lock resolving	当前子任务是否正在等待 shard DDL 迁移，大于 0 表示正在等待迁移	N/A	N/A

Operate error

metric 名称	说明	告警说明	告警级别
before any operate error	在进行操作之前出错的次数	N/A	N/A
source bound error	数据源绑定操作出错次数	N/A	N/A
start error	子任务启动的出错次数	N/A	N/A
pause error	子任务暂停的出错次数	N/A	N/A
resume error	子任务恢复的出错次数	N/A	N/A
auto-resume error	子任务自动恢复的出错次数	N/A	N/A
update error	子任务更新的出错次数	N/A	N/A
stop error	子任务停止的出错次数	N/A	N/A

HA 高可用

	metric 名称	说明	告警说明	告警级别
	number of dm-masters start leader compo-nents per minute	每分钟内 DM-master 尝试启用 leader 相关组件次数	N/A	N/A
	number of workers in different state	不同状态下有多少个 DM-worker	存在离线的 DM-worker 超过一小时	critical
	workers' state	DM-worker 的状态	N/A	N/A

metric 名称	说明	告警说明	告警级别
number of worker event error	不同类型的 DM-worker 错误出现次数	N/A	N/A
shard ddl error per minute	每分钟内不同类型的 shard DDL 错误次数	发生 shard DDL 错误	critical
number of pending shard ddl	未完成的 shard DDL 数目	存在未完成的 shard DDL 数目超过一小时	critical

Task 状态

metric 名称	说明	告警说明	告警级别
task state	迁移子任务的状态	当子任务状态处于 Paused 超过 20 分钟时	critical

Dump/Load unit

下面metrics 仅在 `task-mode` 为 `full` 或者 `all` 模式下会有值。

metric 名称	说明	告警说明	告警
dump progress	dump unit 导出过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
load progress	load unit 导入过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
checksum progress	load unit 导入完成后，数据校验过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
total bytes for load unit	load unit 导入过程中源数据解析、生成数据 KV、生成索引 KV 阶段处理的字节数	N/A	N/A
chunk process duration	load unit 处理数据源文件 chunk 的耗时，单位：秒	N/A	N/A
dump process exits with error	dump unit 在 DM-worker 内部遇到错误并且退出了	立即告警	critic
load process exits with error	load unit 在 DM-worker 内部遇到错误并且退出了	立即告警	critic

Binlog replication

下面metrics 仅在 `task-mode` 为 `incremental` 或者 `all` 模式下会有值。

metric 名称	说明	告警说明	告警级别
remaining time to sync	预计 Syncer 还需要多少分钟可以和上游 master 完全同步，单位：分钟	N/A	N/A
replicate lag gauge	上游 master 到下游的 binlog 复制延迟时间，单位：秒	N/A	N/A
replicate lag histogram	上游 master 到下游的 binlog 复制延迟分布，单位：秒。注意由于统计机制不同，数据会有误差	N/A	N/A
process exist with error	binlog replication unit 在 DM-worker 内部遇到错误并且退出了	立即告警	critical
binlog file gap between master and syncer	与上游 master 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog file gap between relay and syncer	与 relay 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog event QPS	单位时间内接收到的 binlog event 数量(不包含需要跳过的 event)	N/A	N/A

metric 名称	说明	告警说明	告警级别
skipped binlog event QPS	单位时间内接收到的需要跳过的 binlog event 数量	N/A	N/A
read binlog event duration	binlog replication unit 从 relay log 或上游 MySQL 读取 binlog 的耗时，单位：秒	N/A	N/A
transform binlog event duration	binlog replication unit 解析 binlog 并将 binlog 转换成 SQL 语句的耗时，单位：秒	N/A	N/A
dispatch binlog event duration	binlog replication unit 调度一条 binlog event 的耗时，单位：秒	N/A	N/A
transaction execu-tion latency	binlog replication unit 执行事务到下游的耗时，单位：秒	N/A	N/A
binlog event size	binlog replication unit 从 relay log 或上游 MySQL 读取的单条binlog event 的大小	N/A	N/A
DML queue remain length	剩余 DML job 队列的长度	N/A	N/A
total sqls jobs	单位时间内新增的 job 数量	N/A	N/A
finished sqls jobs	单位时间内完成的 job 数量	N/A	N/A
statement execu-tion latency	binlog replication unit 执行语句到下游的耗时，单位：秒	N/A	N/A

metric 名称	说明	告警说明	告警级别
add job duration	binlog replication unit 增加一条 job 到队列的耗时，单位：秒	N/A	N/A
DML conflict detect duration	binlog replication unit 检测 DML 间冲突的耗时，单位：秒	N/A	N/A
skipped event duration	binlog replication unit 跳过 binlog event 的耗时，单位：秒	N/A	N/A
unsynced tables	当前子任务内还未收到 shard DDL 的分表数量	N/A	N/A
shard lock resolving	当前子任务是否正在等待 shard DDL 迁移，大于 0 表示正在等待迁移	N/A	N/A
ideal QPS	在 DM 运行耗时为 0 时可以达到的最高 QPS	N/A	N/A
binlog event row	一个 binlog 事件中的行数	N/A	N/A
finished transaction total	执行完毕的事务数量	N/A	N/A
replication transaction batch	执行到下游的事务里中 sql 行数	N/A	N/A
flush checkpoints time interval	检查点刷新时间间隔，单位：秒	N/A	N/A

Relay log

metric 名称	说明	告警说明	告警级别
storage capacity	relay log 占有的磁盘的总容量	N/A	N/A
storage remain	relay log 占有的磁盘的剩余可用容量	小于 10G 的时候需要告警	critical
process exits with error	relay log 在 DM-worker 内部遇到错误并且退出了	立即告警	critical
relay log data corruption	relay log 文件损坏的个数	立即告警	emergency
fail to read binlog from master	relay 从上游的 MySQL 读取 binlog 时遇到的错误数	立即告警	critical
fail to write relay log	relay 写 binlog 到磁盘时遇到的错误数	立即告警	critical
binlog file index	relay log 最大的文件序列号。如 value = 1 表示 relay-log.000001	N/A	N/A
binlog file gap between master and relay	relay 与上游 master 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog pos	relay log 最新文件的写入 offset	N/A	N/A
read binlog event duration	relay log 从上游的 MySQL 读取 binlog 的时延，单位：秒	N/A	N/A

metric 名称	说明	告警说明	告警级别
write relay log duration	relay log 每次写 binlog 到磁盘的时延，单位：秒	N/A	N/A
binlog event size	relay log 写到磁盘的单条binlog 的大小	N/A	N/A

Instance

在 Grafana dashboard 中，instance 的默认名称为 `DM-instance`。

Relay log

metric 名称	说明	告警说明	告警级别
storage capacity	relay log 占有的磁盘的总容量	N/A	N/A
storage remain	relay log 占有的磁盘的剩余可用容量	小于 10G 的时候需要告警	critical
process exits with error	relay log 在 DM-worker 内部遇到错误并且退出了	立即告警	critical
relay log data corruption	relay log 文件损坏的个数	立即告警	emergency
fail to read binlog from master	relay 从上游的 MySQL 读取 binlog 时遇到的错误数	立即告警	critical
fail to write relay log	relay 写 binlog 到磁盘时遇到的错误数	立即告警	critical
binlog file index	relay log 最大的文件序列号。如 value = 1 表示 relay-log.000001	N/A	N/A

metric 名称	说明	告警说明	告警级别
binlog file gap between master and relay	relay 与上游 master 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog pos	relay log 最新文件的写入 offset	N/A	N/A
read binlog duration	relay log 从上游的 MySQL 读取 binlog 的时延，单位：秒	N/A	N/A
write	relay log 每次写	N/A	N/A
relay log duration	binlog 到磁盘的时延，单位：秒	N/A	N/A
binlog size	relay log 写到磁盘的单条binlog 的大小	N/A	N/A

task

metric 名称	说明	告警说明	告警级别
task state	迁移子任务的状态	当子任务状态处于 paused 超过 10 分钟时	critical
load progress	load unit 导入过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
binlog file gap between master and syncer	与上游master 相比 binlog replication unit 落后的binlog file 个数	N/A	N/A

metric 名称	说明	告警说明	告警级别
shard lock resolving	当前子任务是否正在等待 shard DDL 迁移，大于 0 表示正在等待迁移	N/A	N/A

10.6.9.8.2 DM 告警信息

使用TiUP 部署 DM 集群的时候，会默认部署一套告警系统。

DM 的告警规则及其对应的处理方法可参考告警处理。

DM 的告警信息与监控指标均基于 Prometheus，告警规则与监控指标的对应关系可参考DM 监控指标。

10.6.9.9 TiDB Data Migration 故障及处理方法

本文档介绍TiDB Data Migration (DM) 的错误系统及常见故障的处理方法。

10.6.9.9.1 DM 错误系统

在 DM 的错误系统中，对于一条特定的错误，通常主要包含以下信息：

- code：错误码。

  同一种错误都使用相同的错误码。错误码不随 DM 版本改变。

  在 DM 迭代过程中，部分错误可能会被移除，但错误码不会。新增的错误会使用新的错误码，不会复用已有的错误码。

- class：发生错误的类别。

  用于标记出现错误的系统子模块。

  下表展示所有的错误类别、错误对应的系统子模块、错误样例：

  |

  错误类别

错误对应的系统子模块 ↩	错误样例
:———	:———————
database	执行数据库操作出现错误	[code=10003:class=database:scope=downstream:level=medium] ↩ database driver: invalid connection
functional	系统底层的基础函数错误	[code=11005:class=functional:scope=internal:level=high] ↩ not allowed operation: alter multiple tables in one statement
config	配置错误	[code=20005:class=config:scope=internal:level=medium] empty source-id not ↩ valid

```
binlog-op | binlog 操作出现错误 | [code=22001:class=binlog-op:scope=internal:level=high] empty
↪ UUIDs not valid |
checkpoint | checkpoint 相关操作出现错误 | [code=24002:class=checkpoint:scope=internal:level=
↪ high] save point bin.1234 is older than current pos bin.1371 |
task-check | 进行任务检查时出现错误| [code=26003:class=task-check:scope=internal:level=medium
↪ ] new table router error |
relay-event-lib| 执行 relay 模块基础功能时出现错误 | [code=28001:class=relay-event-lib:scope=
↪ internal:level=high] parse server-uuid.index |
relay-unit | relay 处理单元内出现错误| [code=30015:class=relay-unit:scope=upstream:level=high]
↪  TCPReader get event: ERROR 1236 (HY000): Could not open log file |
dump-unit | dump 处理单元内出现错误| [code=32001:class=dump-unit:scope=internal:level=high]
↪ mydumper runs with error: CRITICAL **: 15:12:17.559: Error connecting to database: Access
↪  denied for user 'root'@'172.17.0.1' (using password: NO) |
load-unit | load 处理单元内出现错误 | [code=34002:class=load-unit:scope=internal:level=high]
↪ corresponding ending of sql: ')' not found |
sync-unit | sync 处理单元内出现错误 | [code=36027:class=sync-unit:scope=internal:level=high]
↪ Column count doesn't match value count: 9 (columns)vs 10 (values) |
dm-master | DM-master 服务内部出现错误 | [code=38008:class=dm-master:scope=internal:level
↪ =high] grpc request error: rpc error: code = Unavailable desc = all SubConns are in
↪ TransientFailure, latest connection error: connection error: desc = "transport: Error
↪ while dialing dial tcp 172.17.0.2:8262: connect: connection refused" |
dm-worker | DM-worker 服务内部出现错误| [code=40066:class=dm-worker:scope=internal:level=high]
↪  ExecuteDDL timeout, try use query-status to query whether the DDL is still blocking |
dm-tracer | DM-tracer 服务内部出现错误| [code=42004:class=dm-tracer:scope=internal:level=medium
↪ ] trace event test.1 not found |
schema-tracker | 增量复制时记录 schema 变更出现错误 | [code=44006:class=schema-tracker:scope=
↪ internal:level=high],"cannot track DDL: ALTER TABLE test DROP COLUMN col1" |
scheduler | 数据迁移任务调度相关操作出错操作 | [code=46001:class=scheduler:scope=internal:
↪ level=high],"the scheduler has not started" |
dmctl | dmctl 内部或与其他组件交互出现错误| [code=48001:class=dmctl:scope=internal:level=high
↪ ],"can not create grpc connection" |
```

- `scope`：错误作用域。

  用于标识错误发生时 DM 作用对象的范围和来源，包括未设置(`not-set`)、上游数据库 (`upstream`)、下游数据库(`downstream`)、内部 (`internal`) 四种类型。

  如果错误发生的逻辑直接涉及到上下游数据库请求，作用域会设置为 `upstream` 或 `downstream`，其他出错场景目前都设置为 `internal`。

- `level`：错误级别。

  错误的严重级别，包括低级别(`low`)、中级别 (`medium`)、高级别 (`high`) 三种。

  低级别通常是用户操作、输入错误，不影响正常迁移任务；中级别通常是用户配置等错误，会影响部分新启动服务，不影响已有系统迁移状态；高级别通常是用户需要关注的一些错误，可能存在迁移任务中断等风险，需要用户进行处理。

- `message`：错误描述。

错误的详细描述信息。对于错误调用链上每一层额外增加的错误message，采用 errors.Wrap 的模式来叠加和保存错误message。wrap 最外层的message 是 DM 内部对该错误的描述，wrap 最内层的message 是该错误最底层出错位置的错误描述。

- workaround: 错误处理方法（可选）。

  对该错误的处理方法。对于部分明确的错误（如：配置信息错误等），DM 会在 workaround 中给出相应的人为处理方法。

- 错误堆栈信息（可选）。

  DM 根据错误的严重程度和必要性来选择是否输出错误堆栈。错误堆栈记录了错误发生时完整的堆栈调用信息。如果用户通过错误基本信息和错误message 描述不能完全诊断出错误发生的原因，可以通过错误堆栈进一步跟进出错时代码的运行路径。

可在 DM 代码仓库已发布的错误码中查询完整的错误码列表。

10.6.9.9.2 DM 故障诊断

如果在运行 DM 工具时出现了错误，请尝试以下解决方案：

1. 执行 query-status 命令查看任务运行状态以及相关错误输出。
2. 查看与该错误相关的日志文件。日志文件位于 DM-master、DM-worker 部署节点上，通过DM 错误系统获取错误的关键信息，然后查看常见故障处理方法以寻找相应的解决方案。
3. 如果该错误还没有相应的解决方案，并且你无法通过查询日志或监控指标自行解决此问题，请从 PingCAP 官方或TiDB 社区获取支持。
4. 一般情况下，错误处理完成后，只需使用dmctl 重启任务即可。

```
resume-task ${task name}
```

但在某些情况下，你还需要重置数据迁移任务。有关何时需要重置以及如何重置，详见重置数据迁移任务。

10.6.9.9.3 常见故障处理方法

|

错误码

错误说明	解决方法
code=10001	数据库操作异常	进一步分析错误信息和错误堆栈
code=10002	数据库底层的 bad connection 错误，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到TiDB。	DM 提供针对此类错误的自动恢复。如果长时间未恢复，需要用户检查网络或TiDB 状态。
code=10003	数据库底层 invalid connection 错误，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如	

网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了TiDB。 | DM 提供针对此类错误的自动恢复。如果未能正常恢复，需要用户进一步检查错误信息并根据具体场景进行分析。 |
code=10005 | 数据库查询类语句出错| |
code=10006 | 数据库 EXECUTE 类型语句出错，包括 DDL 和 INSERT/UPDATE/DELETE 类型的 DML。更详细的错误信息可通过错误message 获取。错误message 中通常包含操作数据库所返回的错误码和错误信息。 | |
code=11006 | DM 内置的parser 解析不兼容的 DDL 时出错| 可参考Data Migration 故障诊断-处理不兼容的 DDL 语句 提供的解决方案|
code=20010 | 处理任务配置时，解密数据库的密码出错| 检查任务配置中提供的下游数据库密码是否有使用 dmctl 正确加密 |
code=26002 | 任务检查创建数据库连接失败。更详细的错误信息可通过错误message 获取。错误message 中包含操作数据库所返回的错误码和错误信息。 | 检查 DM-master 所在的机器是否有权限访问上游|
code=32001 | dump 处理单元异常| 如果报错 `msg` 包含 `mydumper: argument list too long.`，则需要用户根据 block-allow-list，在 `task.yaml` 的dump 处理单元的 `extra-args` 参数中手动加上 `--regex` 正则表达式设置要导出的库表。例如，如果要导出所有库中表名字为 `hello` 的表，可加上 `--regex '.*\\.hello$'`，如果要导出所有表，可加上 `--regex '.*'`。 |
code=38008 | DM 组件间的 gRPC 通信出错 | 检查 `class`，定位错误发生在哪些组件的交互环节，根据错误 message 判断是哪类通信错误。如果是 gRPC 建立连接出错，可检查通信服务端是否运行正常。 |

迁移任务中断并包含 `invalid connection` 错误

原因

发生 `invalid connection` 错误时，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了TiDB。

解决方案

由于 DM 中存在迁移任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 `query-status` 查询当前错误）。

- 如果错误中仅包含 `invalid connection` 类型的错误且当前处于增量复制阶段，则 DM 会自动进行重试。
- 如果 DM 由于版本问题等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

迁移任务中断并包含 `driver: bad connection` 错误

原因

发生 `driver: bad connection` 错误时，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到TiDB。

解决方案

当前版本 DM 会自动进行重试，如果由于版本问题等未自动重试，可先使用 `stop-task` 停止任务后再使用 `start-task` 重启任务。

relay 处理单元报错 `event from * in * diff from passed-in event *` 或迁移任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误

原因

在 DM 进行relay log 拉取与增量复制过程中，如果遇到了上游超过 4GB 的binlog 文件，就可能出现这两个错误。

原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存迁移的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4G 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。

解决方案

对于relay 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的binlog 文件的大小超出了 4GB。
2. 停止 DM-worker。
3. 将上游对应的binlog 文件复制到relay log 目录作为relay log 文件。
4. 更新 relay log 目录内对应的 `relay.meta` 文件以从下一个 binlog 开始拉取。如果 DM worker 已开启 `enable_gtid`，那么在修改 `relay.meta` 文件时， 同样需要修改下一个 binlog 对应的 GTID。如果未开启 `enable_gtid` 则无需修改GTID。

   例如：报错时有 `binlog-name = "mysql-bin.004451"` 与 `binlog-pos = 2453`，则将其分别更新为 `binlog-name = "mysql-bin.004452"` 和 `binlog-pos = 4`，同时更新 `binlog-gtid = "f0e914ef-54cf-11e7-813d-6c92bf2fa791:1-138218058"`。
5. 重启 DM-worker。

对于binlog replication 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的binlog 文件的大小超出了 4GB。
2. 通过 `stop-task` 停止迁移任务。
3. 将下游 `dm_meta` 数据库中 global checkpoint 与每个table 的 checkpoint 中的 `binlog_name` 更新为出错的binlog 文件，将 `binlog_pos` 更新为已迁移过的一个合法的position 值，比如 4。

   例如： 出错任务名为 `dm_test`， 对应的 `source-id` 为 `replica-1`， 出错时对应的 binlog 文件为 `mysql-bin|000001.004451`， 则执行 `UPDATE dm_test_syncer_checkpoint SET binlog_name='mysql-bin|000001.004451', binlog_pos = 4 WHERE id='replica-1';`。
4. 在迁移任务配置中为 `syncers` 部分设置 `safe-mode: true` 以保证可重入执行。
5. 通过 `start-task` 启动迁移任务。
6. 通过 `query-status` 观察迁移任务状态，当原造成出错的relay log 文件迁移完成后，即可还原 `safe-mode` 为原始值并重启迁移任务。

执行 `query-status` 或查看日志时出现 `Access denied for user 'root'@'172.31.43.27' (using password: YES)`

在所有 DM 配置文件中，数据库相关的密码都推荐使用经dmctl 加密后的密文（若数据库密码为空，则无需加密）。有关如何使用 dmctl 加密明文密码，参见使用 dmctl 加密数据库密码。

此外，在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动迁移任务过程中，DM 会自动进行相应权限的前置检查，详见上游 MySQL 实例配置前置检查。

load 处理单元报错 `packet for query is too large. Try adjusting the 'max_allowed_packet' variable`

原因

- MySQL client 和 MySQL/TiDB Server 都有 `max_allowed_packet` 配额的限制，如果在使用过程中违反其中任何一个 `max_allowed_packet` 配额，客户端程序就会收到对应的报错。目前最新版本的 DM 和 TiDB Server 的默认 `max_allowed_packet` 配额都为 `64M`。
- DM 的全量数据导入处理模块不支持对dump 处理模块导出的 SQL 文件进行切分。

解决方案

- 推荐在 DM 的dump 处理单元提供的配置 `extra-args` 中设置 `statement-size`:

  依据默认的 `--statement-size` 设置，DM 的 dump 处理单元默认生成的 `Insert Statement` 大小一般会在 `1M` 左右，使用默认值就可以确保绝大部分情况load 处理单元不会报错 `packet for query is too large . Try adjusting the 'max_allowed_packet' variable`。

  有时候在dump 过程中会出现下面的 `WARN` log。这个 `WARN` log 不影响dump 的过程，只是说明dump 的表可能是宽表。

  ```
  Row bigger than statement_size for xxx
  ```

- 如果宽表的单行超过了 `64M`，需要修改以下两项配置，并且确保其生效。
  - 在 TiDB Server 执行 `set @@global.max_allowed_packet=134217728`（`134217728 = 128M`）
  - 根据实际情况为 DM 的任务配置文件中的 `target-database` 增加配置 `max-allowed-packet`: 134217728 (128M)，执行 `stop-task` 后再重新 `start-task`。

10.6.9.10 TiDB Data Migration 术语表

本文档介绍TiDB Data Migration (TiDB DM) 相关术语。

10.6.9.10.1 B

Binlog

在 TiDB DM 中，Binlog 通常指 MySQL/MariaDB 生成的binary log 文件，具体请参考 MySQL Binary Log 与 MariaDB Binary Log。

Binlog event

MySQL/MariaDB 生成的 Binlog 文件中的数据变更信息，具体请参考 MySQL Binlog Event 与 MariaDB Binlog Event。

Binlog event filter

比 Block & allow table list 更加细粒度的过滤功能，具体可参考Binlog Event Filter。

Binlog position

特定 Binlog event 在 Binlog 文件中的位置偏移信息，具体请参考 MySQL `SHOW BINLOG EVENTS` 与 MariaDB `SHOW BINLOG EVENTS`。

Binlog replication 处理单元/ sync 处理单元

DM-worker 内部用于读取上游 Binlog 或本地 Relay log 并迁移到下游的处理单元，每个 Subtask 对应一个 Binlog replication 处理单元。在当前文档中，有时也称作 Sync 处理单元。

Block & allow table list

针对上游数据库实例表的黑白名单过滤功能，具体可参考Block & Allow Table Lists。该功能与 MySQL Replication Filtering 及 MariaDB Replication Filters 类似。

10.6.9.10.2 C

Checkpoint

TiDB DM 在全量导入与增量复制过程中的断点信息，用于在重新启动或恢复任务时从之前已经处理过的位置继续执行。

- 对于全量导入，Checkpoint 信息对应于每个数据文件已经被成功导入的数据对应的文件内偏移量等信息，其在每个导入数据的事务中迁移更新；
- 对于增量复制，Checkpoint 信息对应于已经成功解析并导入到下游的Binlog event 对应的Binlog position 等信息，其在 DDL 导入成功后或距上次更新时间超过 30 秒等条件下更新。

另外，Relay 处理单元对应的 `relay.meta` 内记录的信息也相当于 Checkpoint，其对应于 Relay 处理单元已经成功从上游拉取并写入到Relay log 的Binlog event 对应的Binlog position 或GTID 信息。

10.6.9.10.3 D

Dump 处理单元

DM-worker 内部用于从上游导出全量数据的处理单元，每个 Subtask 对应一个 Dump 处理单元。

10.6.9.10.4 F

复制/增量复制

使用TiDB Data Migration 工具将上游数据库的增量数据复制到下游数据库的过程。

本用户手册中，在明确提到是“增量”的情况下，将使用“复制”或“增量复制”进行文档描述。

10.6.9.10.5 G

GTID

MySQL/MariaDB 的全局事务ID，当启用该功能后会在 Binlog 文件中记录GTID 相关信息，多个GTID 即组成为GTID Set，具体请参考 MySQL GTID Format and Storage 与 MariaDB Global Transaction ID。

10.6.9.10.6 L

Load 处理单元

DM-worker 内部用于将全量导出数据导入到下游的处理单元，每个 Subtask 对应一个 Load 处理单元。在当前文档中，有时也称作Import 处理单元。

10.6.9.10.7 Q

迁移/全量迁移

使用TiDB Data Migration 工具将上游数据库的全量数据迁移到下游数据库的过程。

本用户手册中，在明确提到是“全量”的情况下，将使用“迁移”或“全量迁移”进行文档描述；在明确提到是“全量+增量”的情况下，也将统一使用“迁移”进行文档描述。

10.6.9.10.8 R

Relay log

DM-worker 从上游 MySQL/MariaDB 拉取 Binlog 后存储在本地的文件，当前其格式为标准的 Binlog 格式，可使用版本兼容的mysqlbinlog 等工具进行解析。其作用与 MySQL Relay Log 及 MariaDB Relay Log 相近。

有关TiDB DM 内 Relay log 的目录结构、初始迁移规则、数据清理等内容，可参考TiDB DM Relay Log。

Relay 处理单元

DM-worker 内部用于从上游拉取 Binlog 并写入数据到Relay log 的处理单元，每个 DM-worker 实例内部仅存在一个该处理单元。

10.6.9.10.9 S

Safe mode

指增量复制过程中，用于支持在表结构中存在主键或唯一索引的条件下可重复导入 DML 的模式。该模式的主要特点是：将来自上游的 `INSERT` 改写为 `REPLACE`，将 `UPDATE` 改写为 `DELETE` 与 `REPLACE` 后再向下游执行。

该模式会在满足如下任一条件时启用：

- 任务配置文件中设置 `safe-mode: true` 时会始终启用
- 合库合表模式下，DDL 尚未在所有分表完成同步时保持启用
- 在全量迁移任务中的dump 处理单元配置 `--consistency none` 后，不能确定导出开始时的binlog 变动是否影响了导出数据。Safe mode 会在增量复制这部分binlog 时保持启用
- 任务出错停止并恢复后，对有些数据的操作可能会被执行两次时保持启用

Shard DDL

指合库合表迁移过程中，在上游各分表(shard) 上执行的需要TiDB DM 进行协调迁移的 DDL。在当前文档中，有时也称作 Sharding DDL。

Shard DDL lock

用于协调 Shard DDL 迁移的锁机制，具体原理可查看悲观模式下分库分表合并迁移实现原理。在当前文档中，有时也称作 Sharding DDL lock。

Shard group

指合库合表迁移过程中，需要合并迁移到下游同一张表的所有上游分表(shard)，TiDB DM 内部具体实现时使用了两级抽象的 Shard group，具体可查看悲观模式下分库分表合并迁移实现原理。在当前文档中，有时也称作 Sharding group。

Subtask

数据迁移子任务，即数据迁移任务运行在单个 DM-worker 实例上的部分。根据任务配置的不同，单个数据迁移任务可能只有一个子任务，也可能有多个子任务。

Subtask status

数据迁移子任务所处的状态，目前包括 New、 Running、 Paused、 Stopped 及 Finished 5 种状态。有关数据迁移任务、子任务状态的更多信息可参考任务状态。

10.6.9.10.10 T

Table routing

用于支持将上游 MySQL/MariaDB 实例的某些表迁移到下游指定表的路由功能，可以用于分库分表的合并迁移，具体可参考Table routing。

Task

数据迁移任务，执行 `start-task` 命令成功后即启动一个数据迁移任务。根据任务配置的不同，单个数据迁移任务既可能只在单个 DM-worker 实例上运行，也可能同时在多个 DM-worker 实例上运行。

Task status

数据迁移子任务所处的状态，由Subtask status 整合而来，具体信息可查看任务状态。

10.6.9.11 使用示例

10.6.9.11.1 使用 DM 迁移数据

本文介绍如何使用 DM 工具迁移数据。

第 1 步：部署 DM 集群

推荐使用TiUP 部署 DM 集群；也可以使用binary 部署 DM 集群用于体验或测试。

> 注意：
>
> - 在 DM 所有的配置文件中，对于数据库密码推荐使用dmctl 加密后的密文。如果数据库密码为空，则不需要加密。关于如何使用dmctl 加密明文密码，参考使用 dmctl 加密数据库密码。
> - 上下游数据库用户必须拥有相应的读写权限。

第 2 步：检查集群信息

使用TiUP 部署 DM 集群后，相关配置信息如下：

- DM 集群相关组件配置信息

组件	主机	端口
dm_worker1	172.16.10.72	8262
dm_worker2	172.16.10.73	8262
dm_master	172.16.10.71	8261

- 上下游数据库实例相关信息

数据库实例	主机	端口	用户名	加密密码
上游 MySQL-1	172.16.10.81	3306	root	VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU=
上游 MySQL-2	172.16.10.82	3306	root	VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU=
下游 TiDB	172.16.10.83	4000	root	

上游 MySQL 数据库实例用户所需权限参见上游 MySQL 实例配置前置检查介绍。

第 3 步：创建数据源

1. 将 MySQL-1 的相关信息写入到 `conf/source1.yaml` 中：

```
# MySQL1 Configuration.

source-id: "mysql-replica-01"

# DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是在上游 MySQL 已开启 GTID
    ↪ 模式。
enable-gtid: false

from:
  host: "172.16.10.81"
  user: "root"
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="
  port: 3306
```

2. 在终端中执行下面的命令，使用 `tiup dmctl` 将 MySQL-1 的数据源配置加载到 DM 集群中：

```
tiup dmctl --master-addr 172.16.10.71:8261 operate-source create conf/source1.yaml
```

3. 对于 MySQL-2，修改配置文件中的相关信息，并执行相同的 dmctl 命令。

第 4 步：配置任务

假设需要将 MySQL-1 和 MySQL-2 实例的 `test_db` 库的 `test_table` 表以全量 + 增量的模式迁移到下游 TiDB 的 `test_db` 库的 `test_table` 表。

编辑任务配置文件 `task.yaml`：

```
##### 任务名，多个同时运行的任务不能重名。
name: "test"
##### 全量+增量 (all) 迁移模式。
task-mode: "all"
##### 下游 TiDB 配置信息。
target-database:
  host: "172.16.10.83"
  port: 4000
  user: "root"
  password: ""

##### 当前数据迁移任务需要的全部上游 MySQL 实例配置。
mysql-instances:
-
  # 上游实例或者复制组 ID，参考 `inventory.ini` 的 `source_id` 或者 `dm-master.toml` 的 `source-
      ↪ id 配置`。
  source-id: "mysql-replica-01"
  # 需要迁移的库名或表名的黑白名单的配置项名称，用于引用全局的黑白名单配置，全局配置见下面的 `
      ↪ block-allow-list` 的配置。
  block-allow-list: "global"          # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  # dump 处理单元的配置项名称，用于引用全局的 dump 处理单元配置。
  mydumper-config-name: "global"

-
  source-id: "mysql-replica-02"
  block-allow-list: "global"          # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  mydumper-config-name: "global"

##### 黑白名单全局配置，各实例通过配置项名引用。
block-allow-list:                     # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  global:
    do-tables:                        # 需要迁移的上游表的白名单。
    - db-name: "test_db"              # 需要迁移的表的库名。
      tbl-name: "test_table"          # 需要迁移的表的名称。

##### dump 处理单元全局配置，各实例通过配置项名引用。
mydumpers:
  global:
    extra-args: ""
```

第 5 步：启动任务

为了提前发现数据迁移任务的一些配置错误，DM 中增加了前置检查功能：

- 启动数据迁移任务时，DM 自动检查相应的权限和配置。

- 也可使用 check-task 命令手动前置检查上游的 MySQL 实例配置是否符合 DM 的配置要求。

> **注意：**
>
> 第一次启动数据迁移任务时，必须确保上游数据库已配置。否则，启动任务时会报错。

使用 tiup dmctl 执行以下命令启动数据迁移任务。其中，task.yaml 是之前编辑的配置文件。

```
tiup dmctl --master-addr 172.16.10.71:8261 start-task ./task.yaml
```

- 如果执行该命令后返回的结果如下，则表明任务已成功启动。

```
{
    "result": true,
    "msg": "",
    "workers": [
        {
            "result": true,
            "worker": "172.16.10.72:8262",
            "msg": ""
        },
        {
            "result": true,
            "worker": "172.16.10.73:8262",
            "msg": ""
        }
    ]
}
```

- 如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。

第 6 步：查询任务

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 tiup dmctl 执行以下命令进行查询：

```
tiup dmctl --master-addr 172.16.10.71:8261 query-status
```

第 7 步：停止任务

如果不再需要进行数据迁移，可以使用 tiup dmctl 执行以下命令停止迁移任务：

```
tiup dmctl --master-addr 172.16.10.71:8261 stop-task test
```

其中的 `test` 是 `task.yaml` 配置文件中 `name` 配置项设置的任务名。

第 8 步：监控任务与查看日志

如果使用TiUP 部署DM 集群时，正确部署了Prometheus、Alertmanager 与Grafana，且其地址均为`172.16.10.71`。可在浏览器中打开http://172.16.10.71:9093 进入Alertmanager 查看DM 告警信息；可在浏览器中打开http://172.16.10.71:3000 进入Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、 DM-master 及dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录位于 `{log_dir}`。
- DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用TiUP 部署 DM，则日志目录位于 `{log_dir}`。

10.6.9.11.2 创建TiDB Data Migration 数据迁移任务

本文档介绍在TiDB Data Migration (DM) 集群部署成功后，如何快速创建简单的数据迁移任务。

使用样例

在本地部署两个开启binlog 的 MySQL 实例和一个TiDB 实例；使用 DM 集群的一个 DM-master 来管理集群和数据迁移任务。各个节点的信息如下：

实例	服务器地址	端口
MySQL1	127.0.0.1	3306
MySQL2	127.0.0.1	3307
TiDB	127.0.0.1	4000
DM-master	127.0.0.1	8261

下面以此为例，说明如何创建数据迁移任务。

运行上游 MySQL

准备 2 个可运行的 MySQL 实例，也可以使用 Docker 快速启动 MySQL，示例命令如下：

```
docker run --rm --name mysql-3306 -p 3306:3306 -e MYSQL_ALLOW_EMPTY_PASSWORD=true mysql:5.7.22 --
    '→ log-bin=mysql-bin --port=3306 --bind-address=0.0.0.0 --binlog-format=ROW --server-id=1 --
    '→ gtid_mode=ON --enforce-gtid-consistency=true > mysql.3306.log 2>&1 &
docker run --rm --name mysql-3307 -p 3307:3307 -e MYSQL_ALLOW_EMPTY_PASSWORD=true mysql:5.7.22 --
    '→ log-bin=mysql-bin --port=3307 --bind-address=0.0.0.0 --binlog-format=ROW --server-id=1 --
    '→ gtid_mode=ON --enforce-gtid-consistency=true > mysql.3307.log 2>&1 &
```

准备数据

- 向 mysql-3306 写入示例数据。

```
drop database if exists `sharding1`;
create database `sharding1`;
use `sharding1`;
```

```
create table t1 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
create table t2 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
insert into t1 (id, uid, name) values (1, 10001, 'Gabriel García Márquez'), (2 ,10002, 'Cien
    ↪  años de soledad');
insert into t2 (id, uid, name) values (3,20001, 'José Arcadio Buendía'), (4,20002, 'Úrsula
    ↪ Iguarán'), (5,20003, 'José Arcadio');
```

- 向 mysql-3307 写入示例数据。

```
drop database if exists `sharding2`;
create database `sharding2`;
use `sharding2`;
create table t2 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
create table t3 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
insert into t2 (id, uid, name, info) values (6, 40000, 'Remedios Moscote', '{}');
insert into t3 (id, uid, name, info) values (7, 30001, 'Aureliano José', '{}'), (8, 30002, '
    ↪ Santa Sofía de la Piedad', '{}'), (9, 30003, '17 Aurelianos', NULL);
```

运行下游TiDB

使用以下命令运行一个TiDB server：

```
wget https://download.pingcap.org/tidb-community-server-v7.0.0-linux-amd64.tar.gz
tar -xzvf tidb-latest-linux-amd64.tar.gz
mv tidb-latest-linux-amd64/bin/tidb-server ./
./tidb-server -P 4000 --store mocktikv --log-file "./tidb.log" &
```

> 警告：
> 本文档中TiDB 的部署方法并不适用于生产或开发环境。

配置 MySQL 数据源

运行数据迁移任务前，需要对 source 进行配置，也就是 MySQL 的相关设置。

对密码进行加密

> 注意：

- 如果数据库没有设置密码，则可以跳过该步骤。
- DM v1.0.6 及其以后版本可以使用明文密码配置 source 信息。

为了安全，可配置及使用加密后的密码。使用 dmctl 对 MySQL/TiDB 的密码进行加密，以密码为“123456” 为例：

```
./dmctl encrypt "123456"
```

```
fCxfQ9XKCezSzuCD0Wf5dUD+LsKegSg=
```

记录该加密后的密码，用于下面新建 MySQL 数据源。

编写 source 配置文件

把以下配置文件内容写入到 conf/source1.yaml 中。

MySQL1 的配置文件：

```
##### MySQL1 Configuration.

source-id: "mysql-replica-01"

##### 是否开启 GTID
enable-gtid: true

from:
  host: "127.0.0.1"
  user: "root"
  password: "fCxfQ9XKCezSzuCD0Wf5dUD+LsKegSg="
  port: 3306
```

对于 MySQL2 数据源，将以上内容复制到文件 conf/source2.yaml 中，将 conf/source2.yaml 配置文件中的 name 修改为 mysql-replica-02，并将 password 和 port 改为相应的值。

创建 source

在终端中执行下面的命令，使用dmctl 将 MySQL1 的数据源配置加载到 DM 集群中：

```
./dmctl --master-addr=127.0.0.1:8261 operate-source create conf/source1.yaml
```

对于 MySQL2，将上面命令中的配置文件替换成 MySQL2 对应的配置文件。

创建数据迁移任务

在导入准备数据后，MySQL1 和 MySQL2 实例中有若干个分表，这些分表的结构相同，所在库的名称都以“sharding” 开头，表名称都以“t” 开头，并且主键或唯一键不存在冲突（即每张分表的主键或唯一键各不相同）。现在需要把这些分表迁移到 TiDB 中的 db_target.t_target 表中。

首先创建任务的配置文件：

```
---
name: test
task-mode: all
shard-mode: "pessimistic"

target-database:
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: "" # 如果密码不为空，则推荐使用经过 dmctl 加密的密文

mysql-instances:
  - source-id: "mysql-replica-01"
    block-allow-list:  "instance"    # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
    route-rules: ["sharding-route-rules-table", "sharding-route-rules-schema"]
    mydumper-thread: 4
    loader-thread: 16
    syncer-thread: 16

  - source-id: "mysql-replica-02"
    block-allow-list:  "instance" #  如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
    route-rules: ["sharding-route-rules-table", "sharding-route-rules-schema"]
    mydumper-thread: 4
    loader-thread: 16
    syncer-thread: 16

block-allow-list:                  # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
  instance:
    do-dbs: ["~^sharding[\\d]+"]
    do-tables:
    -  db-name: "~^sharding[\\d]+"
       tbl-name: "~^t[\\d]+"

routes:
  sharding-route-rules-table:
    schema-pattern: sharding*
    table-pattern: t*
    target-schema: db_target
    target-table: t_target

  sharding-route-rules-schema:
    schema-pattern: sharding*
    target-schema: db_target
```

将以上配置内容写入到 conf/task.yaml 文件中，使用dmctl 创建任务：

```
./dmctl --master-addr 127.0.0.1:8261 start-task conf/task.yaml
```

结果如下：

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

这样就成功创建了一个将 MySQL1 和 MySQL2 实例中的分表数据迁移到TiDB 的任务。

数据校验

修改上游 MySQL 分表中的数据，然后使用sync-diff-inspector 校验上下游数据是否一致，如果一致则说明迁移任务运行正常。

10.6.9.11.3 分表合并数据迁移最佳实践

本文阐述了使用TiDB Data Migration（以下简称 DM）对分库分表进行合并迁移的场景中，DM 相关功能的支持和限制，旨在给出一个业务的最佳实践（使用默认的“悲观协调”模式）。

独立的数据迁移任务

在分库分表合并迁移的实现原理部分，我们介绍了 sharding group 的概念，简单来说可以理解为需要合并到下游同一个表的所有上游表即组成一个 sharding group。

当前的 sharding DDL 算法为了能协调在不同分表执行 DDL 对 schema 变更的影响，加入了一些使用限制。而当这些使用限制由于某些异常原因被打破时，我们需要手动处理 Sharding DDL Lock 甚至是完整重做整个数据迁移任务。

因此，为了减小异常发生时对数据迁移的影响，我们推荐将每一个 sharding group 拆分成一个独立的数据迁移任务。这样当异常发生时，可能只有少部分迁移任务需要进行手动处理，其他数据迁移任务可以不受影响。

手动处理 sharding DDL lock

从分库分表合并迁移的实现原理部分我们可以知道，DM 中的 sharding DDL lock 是用于协调不同上游分表向下游执行 DDL 的一种机制，本身并不是异常。

因此，当通过 `shard-ddl-lock` 查看到 DM-master 上存在 sharding DDL lock 时，或通过 `query-status` 查看到某些 DM-worker 有 `unresolvedGroups` 或 `blockingDDLs` 时，并不要急于使用 `shard-ddl-lock unlock` 尝试去手动解除 sharding DDL lock。

只有在确认当前未能自动解除 sharding DDL lock 是文档中所列的支持场景之一时，才能参照对应支持场景中的手动处理示例进行处理。对于其他未被支持的场景，我们建议完整重做整个数据迁移任务，即清空下游数据库中的数据以及该数据迁移任务相关的 `dm_meta` 信息后，重新执行全量数据及增量数据的迁移。

跨分表数据在主键或唯一索引冲突处理

来自多张分表的数据可能会引发主键或者唯一索引的数据冲突，这需要结合分表逻辑对每个主键或唯一索引进行检查。我们在此列举主键或唯一索引的三种情况：

- 分片键：通常来讲，相同的分片键始终会划分到同一张分表之中，因此分片键不会产生数据冲突。
- 自增主键：每个分表的自增主键会单独计数，因此会出现范围重叠的情况，这需要参照下一节自增主键冲突处理来解决。
- 其他主键或唯一索引：需要根据业务逻辑判断。如果出现数据冲突，也可参照下一节自增主键冲突处理预先在下游创建表结构。

自增主键冲突处理

推荐使用以下两种处理方式。

去掉自增主键的主键属性

假设上游分表的表结构如下：

```
CREATE TABLE `tbl_no_pk` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uk_c2` bigint(20) NOT NULL,
  `content_c3` text,
  PRIMARY KEY (`auto_pk_c1`),
  UNIQUE KEY `uk_c2` (`uk_c2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

如果满足下列条件：

- `auto_pk_c1` 列对业务无意义，且不依赖该列的 `PRIMARY KEY` 属性。
- `uk_c2` 列有 `UNIQUE KEY` 属性，且能保证在所有上游分表间全局唯一。

则可以用以下步骤处理合表时可能由 `auto_pk_c1` 导致的 `ERROR 1062 (23000): Duplicate entry '***' for ↩ key 'PRIMARY'` 问题：

1. 在开始执行全量数据迁移前，在下游数据库创建用于合表迁移的表，但将 `auto_pk_c1` 的 `PRIMARY KEY` 属性修改为普通索引。

```
CREATE TABLE `tbl_no_pk_2` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uk_c2` bigint(20) NOT NULL,
  `content_c3` text,
  INDEX (`auto_pk_c1`),
  UNIQUE KEY `uk_c2` (`uk_c2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

2. 在 task.yaml 文件中增加如下配置跳过自增主键冲突检查：

```
ignore-checking-items: ["auto_increment_ID"]
```

3. 启动数据迁移任务，执行全量与增量数据迁移。
4. 通过 query-status 验证数据迁移任务是否正常，在下游数据库中验证合表中是否已经存在了来自上游的数据。

使用联合主键

假设上游分表的表结构如下：

```
CREATE TABLE `tbl_multi_pk` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uuid_c2` bigint(20) NOT NULL,
  `content_c3` text,
  PRIMARY KEY (`auto_pk_c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

如果满足下列条件：

- 业务不依赖 auto_pk_c1 的 PRIMARY KEY 属性。
- auto_pk_c1 与 uuid_c2 的组合能确保全局唯一。
- 业务能接受将 auto_pk_c1 与 uuid_c2 组成联合 PRIMARY KEY。

则可以用以下步骤处理合表时可能由 auto_pk_c1 导致的 ERROR 1062 (23000): Duplicate entry '***' for key 'PRIMARY' 问题：

1. 在开始执行全量数据迁移前，在下游数据库创建用于合表迁移的表，但不为 auto_pk_c1 指定 PRIMARY KEY 属性，而是将 auto_pk_c1 与 uuid_c2 一起组成 PRIMARY KEY。

```
CREATE TABLE `tbl_multi_pk_c2` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uuid_c2` bigint(20) NOT NULL,
  `content_c3` text,
  PRIMARY KEY (`auto_pk_c1`,`uuid_c2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

2. 启动数据迁移任务，执行全量与增量数据迁移。
3. 通过 query-status 验证数据迁移任务是否正常，在下游数据库中验证合表中是否已经存在了来自上游的数据。

上游RDS 封装分库分表的处理

上游数据源为RDS 且使用了其分库分表功能的情况下，MySQL binlog 中的表名在 SQL client 连接时可能并不可见。例如在UCloud 分布式数据库 UDDB 中，其binlog 表名可能会多出 _0001 的后缀。这需要根据binlog 中的表名规律，而不是 SQL client 所见的表名，来配置table routing 规则。

合表迁移过程中在上游增/删表

从分库分表合并迁移的实现原理部分我们可以知道，sharding DDL lock 的协调依赖于是否已收到上游已有各分表对应的 DDL，且当前 DM 在合表迁移过程中暂时不支持在上游动态增加或删除分表。如果需要在合表迁移过程中，对上游执行增、删分表操作，推荐按以下方式进行处理。

在上游增加分表

如果需要在上游增加新的分表，推荐按以下顺序执行操作：

1. 等待在上游分表上执行过的所有 sharding DDL 协调同步完成。
2. 通过 stop-task 停止任务。
3. 在上游添加新的分表。
4. 确保 task.yaml 配置能使新添加的分表与其他已有的分表能合并到同一个下游表。
5. 通过 start-task 启动任务。
6. 通过 query-status 验证数据迁移任务是否正常，在下游数据库中验证合表中是否已经存在了来自上游的数据。

在上游删除分表

如果需要在上游删除原有的分表，推荐按以下顺序执行操作：

1. 在上游删除原有的分表， 并通过 SHOW BINLOG EVENTS 获取该 DROP TABLE 语句在 binlog 中对应的 End_log_pos，记为 Pos-M。
2. 通过 query-status 获取当前 DM 已经处理完成的binlog event 对应的position（syncerBinlog），记为 Pos-S。
3. 当 Pos-S 比 Pos-M 更大后，则说明 DM 已经处理完 DROP TABLE 语句，且该表在被删除前的数据都已经迁移到下游，可以进行后续操作；否则，继续等待 DM 进行数据迁移。
4. 通过 stop-task 停止任务。
5. 确保 task.yaml 配置能忽略上游已删除的分表。
6. 通过 start-task 启动任务。
7. 通过 query-status 验证数据迁移任务是否正常。

数据迁移限速/流控

当将多个上游 MySQL/MariaDB 实例的数据合并迁移到下游同一个TiDB 集群时，由于每个与上游 MySQL/MariaDB 对应的 DM-worker 都会并发地进行全量与增量数据迁移，默认的并发度（全量阶段的 pool-size 与增量阶段的 worker-count）通过多个 DM-worker 累加后，可能会给下游造成过大的压力，此时应根据TiDB 监控及 DM 监控进行初步的性能分析后，适当地调整各并发度参数的大小。后续 DM 也将考虑支持部分自动的流控策略。

10.6.9.12 异常解决

10.6.9.12.1 Data Migration 常见问题

DM 是否支持迁移阿里RDS 以及其他云数据库的数据？

DM 仅支持解析标准版本的 MySQL/MariaDB 的binlog，对于阿里云RDS 以及其他云数据库没有进行过测试，如果确认其binlog 为标准格式，则可以支持。

已知问题的兼容情况：

- 阿里云RDS
  - 即使上游表没有主键，阿里云RDS 的binlog 中也会包含隐藏的主键列，与上游表结构不一致。
- 华为云RDS
  - 不支持，详见：华为云数据库RDS 是否支持直接读取 Binlog 备份文件。

task 配置中的黑白名单的正则表达式是否支持非获取匹配（?!）？

目前不支持，DM 仅支持 golang 标准库的正则，可以通过re2-syntax 了解 golang 支持的正则表达式。

如果在上游执行的一个 statement 包含多个 DDL 操作，DM 是否支持迁移？

DM 会尝试将包含多个 DDL 变更操作的单条语句拆分成只包含一个 DDL 操作的多条语句，但是可能没有覆盖所有的场景。建议在上游执行的一条 statement 中只包含一个 DDL 操作，或者在测试环境中验证一下，如果不支持，可以给 DM 提 issue。

如何处理不兼容的 DDL 语句？

你需要使用dmctl 手动处理TiDB 不兼容的 DDL 语句（包括手动跳过该 DDL 语句或使用用户指定的 DDL 语句替换原 DDL 语句，详见处理出错的 DDL 语句）。

> 注意：
> TiDB 目前并不兼容 MySQL 支持的所有 DDL 语句。

DM 是否会将视图的 DDL 语句和对视图的 DML 语句同步到下游的TiDB 中？

目前 DM 不会将视图的 DDL 语句同步到下游的TiDB 集群，也不会将针对视图的 DML 语句同步到下游。

如何重置数据迁移任务？

当数据迁移过程中发生异常且无法恢复时，需要重置数据迁移任务，对数据重新进行迁移：

1. 使用 `stop-task` 停止异常的数据迁移任务。
2. 清理下游已迁移的数据。
3. 从下面两种方式中选择其中一种重启数据迁移任务：

- 修改任务配置文件以指定新的任务名，然后使用 `start-task {task-config-file}` 重启迁移任务。
- 使用 `start-task --remove-meta {task-config-file}` 重启数据迁移任务。

设置了 `online-ddl-scheme: "gh-ost"`，gh-ost 表相关的 DDL 报错该如何处理？

```
[unit=Sync] ["error information"="{\"msg\":\"[code=36046:class=sync-unit:scope=internal:level=
    ↪ high] online ddls on ghost table `xxx`.`_xxxx_gho`\\ngithub.com/pingcap/dm/pkg/terror.(*
    ↪ Error).Generate ......
```

出现上述错误可能有以下原因：

DM 在最后 `rename ghost_table to origin table` 的步骤会把内存的 DDL 信息读出，并且还原为 origin table 的 DDL。而内存中的 DDL 信息是在 `alter ghost_table` 的时候进行处理，记录 ghost_table DDL 的信息；或者是在重启 dm-worker 启动task 的时候，从 `dm_meta.{task_name}_onlineddl` 中读取出来。

因此，如果在增量复制过程中，指定的Pos 跳过了`alter ghost_table` 的DDL，但是该Pos 仍在gh-ost 的online-ddl 的过程中，就会因为 ghost_table 没有正确写入到内存以及 `dm_meta.{task_name}_onlineddl`，而导致该问题。

可以通过以下方式绕过这个问题：

1. 取消task 的 `online-ddl-schema` 的配置。
2. 把`_{table_name}_gho`、`_{table_name}_ghc`、`_{table_name}_del` 配置到`block-allow-list.ignore-tables` 中。
3. 手工在下游的TiDB 执行上游的 DDL。
4. 待 Pos 复制到 gh-ost 整体流程后的位置，再重新启用 `online-ddl-schema` 以及注释掉 `block-allow-list.
    ↪ ignore-tables`。

如何为已有迁移任务增加需要迁移的表？

假如已有数据迁移任务正在运行，但又有其他的表需要添加到该迁移任务中，可根据当前数据迁移任务所处的阶段按下列方式分别进行处理。

> 注意：
> 向已有数据迁移任务中增加需要迁移的表操作较复杂，请仅在确有强烈需求时进行。

迁移任务当前处于 Dump 阶段

由于 MySQL 不支持指定 snapshot 来进行导出，因此在导出过程中不支持更新迁移任务并重启以通过断点继续导出，故无法支持在该阶段动态增加需要迁移的表。

如果确实需要增加其他的表用于迁移，建议直接使用新的配置文件重新启动迁移任务。

迁移任务当前处于 Load 阶段

多个不同的数据迁移任务在导出时，通常对应于不同的binlog position，如将它们在 Load 阶段合并导入，则无法就binlog position 达成一致，因此不建议在 Load 阶段向数据迁移任务中增加需要迁移的表。

迁移任务当前处于 Sync 阶段

当数据迁移任务已经处于 Sync 阶段时，在配置文件中增加额外的表并重启任务，DM 并不会为新增的表重新执行全量导出与导入，而是会继续从之前的断点进行增量复制。

因此，如果需要新增的表对应的全量数据尚未导入到下游，则需要先使用单独的数据迁移任务将其全量数据导出并导入到下游。

将已有迁移任务对应的全局checkpoint （ is_global=1 ）中的 position 信息记为 checkpoint-T，如 (mysql-bin.000100, 1234)。将需要增加到迁移任务的表在全量导出的 metedata（或另一个处于 Sync 阶段的数据迁移任务的checkpoint ）的position 信息记为 checkpoint-S，如 (mysql-bin.000099, 5678)。则可通过以下步骤将表增加到迁移任务中：

1. 使用 stop-task 停止已有迁移任务。如果需要增加的表属于另一个运行中的迁移任务，则也将其停止。
2. 使用 MySQL 客户连接到下游 TiDB 数据库， 手动更新已有迁移任务对应的 checkpoint 表中的信息为 checkpoint-T 与 checkpoint-S 中的较小值（在本例中，为 (mysql-bin.000099, 5678) ）。
    - 需要更新的checkpoint 表为 {dm_meta} 库中的 {task-name}_syncer_checkpoint。
    - 需要更新的checkpoint 行为 id={source-id} 且 is_global=1。
    - 需要更新的checkpoint 列为 binlog_name 与 binlog_pos。
3. 在迁移任务配置中为 syncers 部分设置 safe-mode: true 以保证可重入执行。
4. 通过 start-task 启动迁移任务。
5. 通过 query-status 观察迁移任务状态，当 syncerBinlog 超过 checkpoint-T 与 checkpoint-S 中的较大值后（在本例中，为 (mysql-bin.000100, 1234) ），即可还原 safe-mode 为原始值并重启迁移任务。

全量导入过程中遇到报错 packet for query is too large. Try adjusting the 'max_allowed_packet' variable

尝试将

- TiDB Server 的全局变量 max_allowed_packet
- 任务配置文件中的配置项 target-database.max-allowed-packet（详情参见DM 任务完整配置文件介绍 ）

设置为比默认 67108864 (64M) 更大的值。

2.0+ 集群运行 1.0 已有数据迁移任务时报错 Error 1054: Unknown column 'binlog_gtid' in 'field list'

在 DM 2.0 之后，为checkpoint 等元信息表引入了更多的字段。如果通过 start-task 直接使用 1.0 集群的任务配置文件从增量复制阶段继续运行，则会出现 Error 1054: Unknown column 'binlog_gtid' in 'field list' 错误。

对于此错误，可手动将 DM 1.0 的数据迁移任务导入到 2.0+ 集群。

TiUP 无法部署 DM 的某个版本（如 v2.0.0-hotfix ）

你可以通过 tiup list dm-master 命令查看TiUP 支持部署的 DM 版本。该命令未展示的版本不能由TiUP 管理。

DM 同步报错 `parse mydumper metadata error: EOF`

该错误需要查看报错信息以及日志进一步分析。报错原因可能是dump 单元由于缺少权限没有产生正确的 metadata 文件。

DM 分库分表同步中没有明显报错，但是下游数据丢失

需要检查配置项 `block-allow-list` 和 `table-route`：

- `block-allow-list` 填写的是上游数据库表，可以在 `do-tables` 前通过加 “~” 来进行正则匹配。
- `table-route` 不支持正则，采用的是通配符模式，所以 `table_parttern_[0-63]` 只会匹配table_parttern_0 到 table_pattern_6 这 7 张表。

DM 上游无写入，replicate lag 监控无数据

在DM v1.0 中，需要开启`enable-heartbeat` 才会产生该监控数据。v2.0 及以后版本中，尚未启用该功能，replicate lag 监控无数据是预期行为。

DM v2.0.0 启动任务时出现 `fail to initial unit Sync of subtask`，报错信息的 RawCause 显示 `context dead-line exceeded`

该问题是 DM v2.0.0 的已知问题，在同步任务的表数目较多时触发，将在 v2.0.1 修复。使用TiUP 部署的用户可以升级到开发版nightly 解决该问题，或者访问GitHub 上 DM 仓库的release 页面下载 v2.0.0-hotfix 版本手动替换可执行文件。

DM 同步中报错 `duplicate entry`

用户需要首先确认任务中没有配置 `disable-detect`（v2.0.7 及之前版本），没有其他同步程序或手动插入数据，表中没有配置相关的 DML 过滤器。

为了便于排查问题，用户收集到下游TiDB 相关general log 后可以在AskTUG 社区联系专家进行排查。收集general log 的方式如下：

```
##### 开启 general log
curl -X POST -d "tidb_general_log=1" http://{TiDBIP}:10080/settings
##### 关闭 general log
curl -X POST -d "tidb_general_log=0" http://{TiDBIP}:10080/settings
```

在发生 `duplicate entry` 报错时，确认日志中包含冲突数据的记录。

监控中部分面板显示 `No data point`

请参照DM 监控指标查看各面板含义，部分面板没有数据是正常现象。例如没有发生错误、不存在 DDL lock、没有启用relay 功能等情况，均可能使得对应面板没有数据。

DM v1.0 在任务出错时使用 `sql-skip` 命令无法跳过某些语句

首先需要检查执行 `sql-skip` 之后binlog 位置是否在推进，如果是的话表示 `sql-skip` 已经生效。重复出错的原因是上游发送了多个不支持的 DDL，可以通过 `sql-skip -s <sql-pattern>` 进行模式匹配。

对于类似下面这种报错（报错中包含 `parse statement`）：

```
if the DDL is not needed, you can use a filter rule with \"*\" schema-pattern to ignore it.\n\t :
    ↪ parse statement: line 1 column 11 near \"EVENT `event_del_big_table`
    \r\nDISABLE\" %!!( ↪ MISSING)(EXTRA string=ALTER EVENT `event_del_big_table` \r\nDISABLE
```

出现报错的原因是TiDB parser 无法解析上游的 DDL，例如 `ALTER EVENT`，所以 `sql-skip` 不会按预期生效。可以在任务配置文件中添加Binlog 过滤规则进行过滤，并设置 `schema-pattern: "*"`。从 DM 2.0.1 版本开始，已预设过滤了 `EVENT` 相关语句。

在 DM v6.0 版本之后 `sql-skip`、`handle-error` 均已经被 `binlog` 替代，使用 `binlog` 命令可以跳过该类错误。

DM 同步时下游长时间出现REPLACE 语句

请检查是否符合safe mode 触发条件。如果任务发生错误并自动恢复，或者发生高可用调度，会满足“启动或恢复任务的前 1 分钟”这一条件，因此启用 safe mode。

可以检查 DM-worker 日志，在其中搜索包含 `change count` 的行，该行的 `new count` 非零时会启用 safe mode。检查 safe mode 启用时间以及启用前是否有报错，以定位启用原因。

使用 DM v2.0 同步数据时重启 DM 进程，出现全量数据导入失败错误

在 DM v2.0.1 及更早版本中，如果全量导入操作未完成时发生重启，重启后的上游数据源与 DM worker 的绑定关系可能会发生变化。例如，可能出现dump 单元的中间数据在 DM worker A 机器上，但却由 DM worker B 进行 load 单元的情况，进而导致操作失败。

该情况有两种解决方案：

- 如果数据量较小（`TB` 级以下）或任务有合库合表：清空下游数据库的已导入数据，同时清空导出数据目录，使用dmctl 删除并 `start-task --remove-meta` 重建任务。后续尽量保证全量导出导入阶段 DM 没有冗余 worker 以及避免在该时段内重启或升级 DM 集群。
- 如果数据量较大（数 `TB` 或更多）：清空下游数据库的已导入数据，将 lightning 部署到数据所在的 DM worker 节点，使用lightning local backend 模式导入 DM dump 单元导出的数据。全量导入完成后，修改任务的 `task-mode` 为 `incremental`，修改 `mysql-instance.meta.pos` 为dump 单元导出数据 `metadata` 中记录的位置，启动一个增量任务。

使用 DM 同步数据时重启 DM 进程， 增量任务出现 `ERROR 1236 (HY000): The slave is connecting using CHANGE MASTER TO MASTER_AUTO_POSITION = 1, but the master has purged binary logs containing GTIDs that the slave requires.` 错误

该错误表明全量迁移期间，dump 单元记录metadata 中的binlog 位置已经被上游清理。

解决方案：出现该问题时只能清空下游数据库已同步数据，并在停止任务后加上 `--remove-meta` 参数重建任务。

如要提前避免该问题，需要进行以下配置：

1. 在 DM 全量迁移未完成时调大上游 MySQL 的 `expire_logs_days` 变量，保证全量进行结束时metadata 中的 binlog 位置到当前时间的binlog 都还没有被清理掉。如果数据量较大，应该使用dumpling + lightning 的方式加快全量迁移速度。
2. DM 任务开启relay log 选项，保证binlog 被清理后 DM 仍有relay log 可读取。

使用TiUP v1.3.0, v1.3.1 部署 DM 集群，DM 集群的 grafana 监控报错显示 `failed to fetch dashboard`

该问题为TiUP 已知bug，在TiUP v1.3.2 中已进行修复。可采取以下任一方法解决：

- 方法一：使用 `tiup update --self && tiup update dm` 升级TiUP 到更新版本，随后先缩容再扩容集群中的 grafana 节点，重建 grafana 服务。

- 方法二：

    1. 备份 deploy/grafana-$port/bin/public 文件夹。
    2. 下载 TiUP DM 离线镜像包，并进行解压，将其中的 grafana-v4.0.3-**.tar.gz 文件解压后，用解压出的 public/ 文件夹替换前面所描述的文件夹，运行 `tiup dm restart $cluster_name -R grafana` 重启 grafana 服务监控。

在 DM v2.0 中，同时开启relay 与 gtid 同步 MySQL 时，运行 `query-status` 发现 syncer checkpoint 中 GTID 不连续

该问题为 DM 已知bug，在完全满足以下两个条件时将会触发，DM 将在 v2.0.2 修复该问题：

1. DM 配置的 source 同时设置了 `enable-relay` 与 `enable-gtid` 为 `true`
2. DM 同步上游为 MySQL 从库，并且该从库通过 `show binlog events in '<newest-binlog>' limit 2` 查询出的 `previous_gtids` 区间不连续，例如：

```
mysql> show binlog events in 'mysql-bin.000005' limit 2;
+------------------+------+---------------+-----------+-------------+-----------------------------------
    '→
| Log_name         | Pos  | Event_type    | Server_id | End_log_pos | Info
    '→                                                                |
+------------------+------+---------------+-----------+-------------+-----------------------------------
    '→
| mysql-bin.000005 |    4 | Format_desc   |    123452 |         123 | Server ver: 5.7.32-35-log,
    '→ Binlog ver: 4                          |
| mysql-bin.000005 |  123 | Previous_gtids |    123452 |         194 | d3618e68-6052-11eb-a68b
    '→ -0242ac110002:6-7                          |
+------------------+------+---------------+-----------+-------------+-----------------------------------
    '→
```

使用dmctl 的 `query-status <task>` 指令查询任务信息，如果已经出现 `subTaskStatus.sync.syncerBinlogGtid` 不连续但 `subTaskStatus.sync.masterBinlogGtid` 连续的情况，例如下述例子：

```
query-status test
{
    ...
    "sources": [
        {
            ...
            "sourceStatus":
                { "source": "mysql1",
                ...
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000006, 744)",
                    "masterBinlogGtid":  "f8004e25-6067-11eb-9fa3-0242ac110003:1-50",
                    ...
                }
            },
```

```
            "subTaskStatus": [
                {
                    ...
                    "sync": {
                        ...
                        "masterBinlog": "(mysql-bin.000006, 744)",
                        "masterBinlogGtid": "f8004e25-6067-11eb-9fa3-0242ac110003:1-50",
                        "syncerBinlog": "(mysql-bin|000001.000006, 738)",
                        "syncerBinlogGtid":  "f8004e25-6067-11eb-9fa3-0242ac110003:1-20:40-49",
                        ...
                        "synced": false,
                        "binlogType": "local"
                    }
                }
            ]
        },
        {
            ...
            "sourceStatus":
                { "source": "mysql2",
                ...
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000007, 1979)",
                    "masterBinlogGtid":  "ddb8974e-6064-11eb-8357-0242ac110002:1-25",
                    ...
                }
            },
            "subTaskStatus": [
                {
                    ...
                    "sync": {
                        "masterBinlog": "(mysql-bin.000007, 1979)",
                        "masterBinlogGtid": "ddb8974e-6064-11eb-8357-0242ac110002:1-25",
                        "syncerBinlog": "(mysql-bin|000001.000008, 1979)",
                        "syncerBinlogGtid": "ddb8974e-6064-11eb-8357-0242ac110002:1-25",
                        ...
                        "synced": true,
                        "binlogType": "local"
                    }
                }
            ]
        }
    ]
}
```

其中mysql1 的 syncerBinlogGtid 不连续，已有数据丢失需要按下述方案之一处理：

- 如果全量导出任务metadata 中的position 到当前时间的上游数据库的binlog 仍未被清理：
  1. 停止当前任务并删除所有GTID 不连续的 source
  2. 设置所有 source 的 enable-relay 为 false
  3. 针对GTID 不连续的 source（上例 mysql1），在对应的任务配置文件 task.yaml 中，把 task-mode 修改为 incremental 并配置增量任务起始点 mysql-instances.meta 为各个全量导出任务metadata 的 binlog name，position 和 gtid 信息
  4. 配置 task.yaml 中的 syncers.safe-mode 为 true 并重启任务
  5. 待增量同步追上后，停止任务并在任务配置文件中设置 safe-mode 为 false
  6. 再次重启任务
- 如果上游数据库binlog 已被清理但是本地relay log 仍未被清理：
  1. 停止当前任务
  2. 针对GTID 不连续的 source（上例 mysql1），在对应的任务配置文件 task.yaml 中，把 task-mode 修改为 incremental 并配置增量任务起始点 mysql-instances.meta 为各个全量导出任务metadata 的 binlog name，position 和 gtid 信息
  3. 修改其中的GTID 信息的 1-y 为 previous_gtids 的前段值，例如，上述例子需要改为 6-y
  4. 配置 task.yaml 中的 syncers.safe-mode 为 true 并重启任务
  5. 待增量同步追上后，停止任务并在任务配置文件中设置 safe-mode 为 false
  6. 再次重启任务
  7. 重启 source 并关闭 gtid 或 relay
- 如果上述条件均不满足或任务同步数据量较小：
  1. 清空下游数据库中数据
  2. 重启 source 并关闭 gtid 或 relay
  3. 重建任务并通过 start-task task.yaml --remove-meta 重新同步

上述处理方案中，针对正常同步的 source（如上例 mysql2），重设增量任务时起始点需设置 mysql-instances.
′→ meta 为 subTaskStatus.sync 的 syncerBinlog 与 syncerBinlogGtid。

在 DM 2.0 中开启heartbeat，虚拟IP 环境下切换 DM-worker 与 MySQL 实例的连接，遇到“heartbeat config is different from previous used: serverID not equal” 错误

heartbeat 功能在 DM v2.0 及之后版本已经默认关闭，如果用户在同步任务配置文件中开启会干扰高可用特性，在配置文件中关闭该项（通过设置 enable-heartbeat: false，然后更新任务配置）即可解决。DM 将会在后续版本强制关闭该功能。

DM-master 在重启后无法加入集群，报错信息为 “fail to start embed etcd, RawCause: member xxx has already been bootstrapped”

DM-master 会在启动时将 etcd 信息记录在当前目录。如果重启后当前目录发生变化，会导致 DM 缺失 etcd 信息，从而启动失败。

推荐使用TiUP 运维 DM 避免这一问题。在需要使用二进制部署的场合，需要在 DM-master 配置文件中使用绝对路径配置data-dir 项，或者注意运行命令的当前目录。

使用dmctl 执行命令时无法连接 DM-master

在使用dmctl 执行相关命令时，发现连接 DM-master 失败（即使已在命令中指定 `--master-addr` 的参数值），报错内容类似`RawCause: context deadline exceeded, Workaround: please check your network connection.`，但使用 `telnet <master-addr>` 之类的命令检查网络却没有发现异常。

这种情况可以检查下环境变量 `https_proxy`（注意，这里是 https ）。如果配置了该环境变量，dmctl 会自动去连接 `https_proxy` 指定的主机及端口，而如果该主机没有相应的 `proxy` 转发服务，则会导致连接失败。

解决方案：确认 `https_proxy` 是否必须要配置，如果不是必须的，取消该设置即可。如果环境必须，那么在原命令前加环境变量设置 `https_proxy="" ./dmctl --master-addr "x.x.x.x:8261"` 即可。

> 注意：
>
> 关于 `proxy` 的环境变量有 `http_proxy`，`https_proxy`，`no_proxy` 等。如果依据上述解决方案处理后仍无法连接，可以考虑检查 `http_proxy` 和 `no_proxy` 的参数配置是否有影响。

v2.0.2 - v2.0.6 版本执行 start-relay 命令报错该如何处理？

```
flush local meta, Rawcause: open relay-dir/xxx.000001/relay.metayyyy: no such file or directory
```

上述报错在以下情况下有可能会被触发：

- DM 从 v2.0.1 及之前的版本升级到 v2.0.2 - v2.0.6 版本，且升级之前曾开启过relay log，升级完后重新开启。
- 使用 stop-relay 命令暂停relay log 后重新开启relay log。

可以通过以下方式解决该问题：

- 重启relay log:

  ```
  » stop-relay -s sourceID workerName
  » start-relay -s sourceID workerName
  ```

- 升级 DM 至 v2.0.7 或之后版本。

Load 单元报错 `Unknown character set`

由于TiDB 只支持部分 MySQL 字符集，因此，在全量导入中，如果创建表结构时使用了TiDB 不支持的字符集，DM 会报这个错误。你可以结合数据内容选择TiDB 支持的字符集，预先在下游创建表结构以绕过这个错误。

10.6.9.12.2 TiDB Data Migration 故障及处理方法

本文档介绍TiDB Data Migration (DM) 的错误系统及常见故障的处理方法。

DM 错误系统

在 DM 的错误系统中，对于一条特定的错误，通常主要包含以下信息：

- code：错误码。

  同一种错误都使用相同的错误码。错误码不随 DM 版本改变。

  在 DM 迭代过程中，部分错误可能会被移除，但错误码不会。新增的错误会使用新的错误码，不会复用已有的错误码。

- class：发生错误的类别。

  用于标记出现错误的系统子模块。

  下表展示所有的错误类别、错误对应的系统子模块、错误样例：

  |

  错误类别

错误对应的系统子模块	错误样例
database	执行数据库操作出现错误	[code=10003:class=database:scope=downstream:level=medium] database driver: invalid connection
functional	系统底层的基础函数错误	[code=11005:class=functional:scope=internal:level=high] not allowed operation: alter multiple tables in one statement
config	配置错误	[code=20005:class=config:scope=internal:level=medium] empty source-id not valid
binlog-op	binlog 操作出现错误	[code=22001:class=binlog-op:scope=internal:level=high] empty UUIDs not valid
checkpoint	checkpoint 相关操作出现错误	[code=24002:class=checkpoint:scope=internal:level=high] save point bin.1234 is older than current pos bin.1371
task-check	进行任务检查时出现错误	[code=26003:class=task-check:scope=internal:level=medium] new table router error
relay-event-lib	执行 relay 模块基础功能时出现错误	[code=28001:class=relay-event-lib:scope=internal:level=high] parse server-uuid.index
relay-unit	relay 处理单元内出现错误	[code=30015:class=relay-unit:scope=upstream:level=high] TCPReader get event: ERROR 1236 (HY000): Could not open log file
dump-unit	dump 处理单元内出现错误	[code=32001:class=dump-unit:scope=internal:level=high] mydumper runs with error: CRITICAL **: 15:12:17.559: Error connecting to database: Access denied for user 'root'@'172.17.0.1' (using password: NO)
load-unit	load 处理单元内出现错误	[code=34002:class=load-unit:scope=internal:level=high] corresponding ending of sql: ')' not found
sync-unit	sync 处理单元内出现错误	[code=36027:class=sync-unit:scope=internal:level=high] Column count doesn't match value count: 9 (columns)vs 10 (values)
dm-master	DM-master 服务内部出现错误	[code=38008:class=dm-master:scope=internal:level=high] grpc request error: rpc error: code = Unavailable desc = all SubConns are in TransientFailure, latest connection error: connection error: desc = "transport: Error while dialing dial tcp 172.17.0.2:8262: connect: connection refused"
dm-worker	DM-worker 服务内部出现错误	[code=40066:class=dm-worker:scope=internal:level=high] ExecuteDDL timeout, try use query-status to query whether the DDL is still blocking

dm-tracer | DM-tracer 服务内部出现错误 | [code=42004:class=dm-tracer:scope=internal:level=medium
↪] trace event test.1 not found |

schema-tracer | 增量复制时记录 schema 变更出现错误 | [code=44006:class=schema-tracer:scope=
↪ internal:level=high],"cannot track DDL: ALTER TABLE test DROP COLUMN col1" |

scheduler | 数据迁移任务调度相关操作出错操作 | [code=46001:class=scheduler:scope=internal:
↪ level=high],"the scheduler has not started" |

dmctl | dmctl 内部或与其他组件交互出现错误 | [code=48001:class=dmctl:scope=internal:level=high
↪],"can not create grpc connection" |

- scope：错误作用域。

  用于标识错误发生时 DM 作用对象的范围和来源，包括未设置(not-set)、上游数据库 (upstream)、下游数据库(downstream)、内部 (internal) 四种类型。

  如果错误发生的逻辑直接涉及到上下游数据库请求，作用域会设置为 upstream 或 downstream，其他出错场景目前都设置为 internal。

- level：错误级别。

  错误的严重级别，包括低级别(low)、中级别 (medium)、高级别 (high) 三种。

  低级别通常是用户操作、输入错误，不影响正常迁移任务；中级别通常是用户配置等错误，会影响部分新启动服务，不影响已有系统迁移状态；高级别通常是用户需要关注的一些错误，可能存在迁移任务中断等风险，需要用户进行处理。

- message：错误描述。

  错误的详细描述信息。对于错误调用链上每一层额外增加的错误message，采用 errors.Wrap 的模式来叠加和保存错误message。wrap 最外层的message 是 DM 内部对该错误的描述，wrap 最内层的message 是该错误最底层出错位置的错误描述。

- workaround: 错误处理方法（可选）。

  对该错误的处理方法。对于部分明确的错误（如：配置信息错误等），DM 会在 workaround 中给出相应的人为处理方法。

- 错误堆栈信息（可选）。

  DM 根据错误的严重程度和必要性来选择是否输出错误堆栈。错误堆栈记录了错误发生时完整的堆栈调用信息。如果用户通过错误基本信息和错误message 描述不能完全诊断出错误发生的原因，可以通过错误堆栈进一步跟进出错时代码的运行路径。

可在 DM 代码仓库已发布的错误码中查询完整的错误码列表。

DM 故障诊断

如果在运行 DM 工具时出现了错误，请尝试以下解决方案：

1. 执行 query-status 命令查看任务运行状态以及相关错误输出。
2. 查看与该错误相关的日志文件。日志文件位于 DM-master、DM-worker 部署节点上，通过DM 错误系统获取错误的关键信息，然后查看常见故障处理方法以寻找相应的解决方案。
3. 如果该错误还没有相应的解决方案，并且你无法通过查询日志或监控指标自行解决此问题，请从 PingCAP 官方或TiDB 社区获取支持。

4. 一般情况下，错误处理完成后，只需使用dmctl 重启任务即可。

```
resume-task ${task name}
```

但在某些情况下，你还需要重置数据迁移任务。有关何时需要重置以及如何重置，详见重置数据迁移任务。

常见故障处理方法

|

错误码

错误说明	解决方法
code=10001	数据库操作异常	进一步分析错误信息和错误堆栈
code=10002	数据库底层的 bad connection 错误，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到TiDB。	DM 提供针对此类错误的自动恢复。如果长时间未恢复，需要用户检查网络或TiDB 状态。
code=10003	数据库底层 invalid connection 错误，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了TiDB。	DM 提供针对此类错误的自动恢复。如果未能正常恢复，需要用户进一步检查错误信息并根据具体场景进行分析。
code=10005	数据库查询类语句出错	
code=10006	数据库 EXECUTE 类型语句出错，包括 DDL 和 INSERT/UPDATE/DELETE 类型的 DML。更详细的错误信息可通过错误message 获取。错误message 中通常包含操作数据库所返回的错误码和错误信息。	
code=11006	DM 内置的parser 解析不兼容的 DDL 时出错	可参考Data Migration 故障诊断-处理不兼容的 DDL 语句 提供的解决方案
code=20010	处理任务配置时，解密数据库的密码出错	检查任务配置中提供的下游数据库密码是否有使用 dmctl 正确加密
code=26002	任务检查创建数据库连接失败。更详细的错误信息可通过错误message 获取。错误message 中包含操作数据库所返回的错误码和错误信息。	检查 DM-master 所在的机器是否有权限访问上游
code=32001	dump 处理单元异常	如果报错 msg 包含 mydumper: argument list too long.，则需要用户根据 block-allow-list，在 task.yaml 的dump 处理单元的 extra-args 参数中手动加上 --regex 正则表达式设置要导出的库表。例如，如果要导出所有库中表名字为 hello 的表，可加上 --regex '.*\\.hello$'，如果要导出所有表，可加上 --regex '.*'。
code=38008	DM 组件间的 gRPC 通信出错	检查 class，定位错误发生在哪些组件的交互环节，根据错误 message 判断是哪类通信错误。如果是 gRPC 建立连接出错，可检查通信服务端是否运行正常。

迁移任务中断并包含 invalid connection 错误

原因

发生 invalid connection 错误时，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了TiDB。

解决方案

由于 DM 中存在迁移任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 query-status 查询当前错误）。

- 如果错误中仅包含 `invalid connection` 类型的错误且当前处于增量复制阶段，则 DM 会自动进行重试。
- 如果 DM 由于版本问题等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

迁移任务中断并包含 `driver: bad connection` 错误

原因

发生 `driver: bad connection` 错误时，通常表示 DM 到下游TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到TiDB。

解决方案

当前版本 DM 会自动进行重试，如果由于版本问题等未自动重试，可先使用 `stop-task` 停止任务后再使用 `start-task` 重启任务。

relay 处理单元报错 `event from * in * diff from passed-in event *` 或迁移任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误

原因

在 DM 进行relay log 拉取与增量复制过程中，如果遇到了上游超过 4GB 的binlog 文件，就可能出现这两个错误。

原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存迁移的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4G 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。

解决方案

对于relay 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的binlog 文件的大小超出了 4GB。
2. 停止 DM-worker。
3. 将上游对应的binlog 文件复制到relay log 目录作为relay log 文件。
4. 更新 relay log 目录内对应的 `relay.meta` 文件以从下一个 binlog 开始拉取。如果 DM worker 已开启 `enable_gtid`，那么在修改 `relay.meta` 文件时， 同样需要修改下一个 binlog 对应的 GTID。如果未开启 `enable_gtid` 则无需修改GTID。

   例如：报错时有 `binlog-name = "mysql-bin.004451"` 与 `binlog-pos = 2453`，则将其分别更新为 `binlog-name = "mysql-bin.004452"` 和 `binlog-pos = 4`，同时更新 `binlog-gtid = "f0e914ef-54cf-11e7-813d-6c92bf2fa791:1-138218058"`。
5. 重启 DM-worker。

对于binlog replication 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的binlog 文件的大小超出了 4GB。
2. 通过 `stop-task` 停止迁移任务。

3. 将下游 dm_meta 数据库中 global checkpoint 与每个table 的 checkpoint 中的 binlog_name 更新为出错的binlog 文件，将 binlog_pos 更新为已迁移过的一个合法的position 值，比如 4。

   例如： 出错任务名为 dm_test， 对应的 source-id 为 replica-1， 出错时对应的 binlog 文件为 mysql-bin|000001.004451， 则执行 UPDATE dm_test_syncer_checkpoint SET binlog_name='mysql-bin|000001.004451', binlog_pos = 4 WHERE id='replica-1';。

4. 在迁移任务配置中为 syncers 部分设置 safe-mode: true 以保证可重入执行。
5. 通过 start-task 启动迁移任务。
6. 通过 query-status 观察迁移任务状态，当原造成出错的relay log 文件迁移完成后，即可还原 safe-mode 为原始值并重启迁移任务。

执行 query-status 或查看日志时出现 Access denied for user 'root'@'172.31.43.27' (using password: YES)

在所有 DM 配置文件中，数据库相关的密码都推荐使用经dmctl 加密后的密文（若数据库密码为空，则无需加密）。有关如何使用 dmctl 加密明文密码，参见使用 dmctl 加密数据库密码。

此外，在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动迁移任务过程中，DM 会自动进行相应权限的前置检查，详见上游 MySQL 实例配置前置检查。

load 处理单元报错 packet for query is too large. Try adjusting the 'max_allowed_packet' variable

原因

- MySQL client 和 MySQL/TiDB Server 都有 max_allowed_packet 配额的限制，如果在使用过程中违反其中任何一个 max_allowed_packet 配额，客户端程序就会收到对应的报错。目前最新版本的 DM 和 TiDB Server 的默认 max_allowed_packet 配额都为 64M。
- DM 的全量数据导入处理模块不支持对dump 处理模块导出的 SQL 文件进行切分。

解决方案

- 推荐在 DM 的dump 处理单元提供的配置 extra-args 中设置 statement-size:

  依据默认的 --statement-size 设置，DM 的 dump 处理单元默认生成的 Insert Statement 大小一般会在 1M 左右，使用默认值就可以确保绝大部分情况load 处理单元不会报错 packet for query is too large. Try adjusting the 'max_allowed_packet' variable。

  有时候在dump 过程中会出现下面的 WARN log。这个 WARN log 不影响dump 的过程，只是说明dump 的表可能是宽表。

  ```
  Row bigger than statement_size for xxx
  ```

- 如果宽表的单行超过了 64M，需要修改以下两项配置，并且确保其生效。
  - 在 TiDB Server 执行 set @@global.max_allowed_packet=134217728（134217728 = 128M）
  - 根据实际情况为 DM 的任务配置文件中的 target-database 增加配置 max-allowed-packet: 134217728 (128M)，执行 stop-task 后再重新 start-task。

10.6.9.13 TiDB Data Migration 版本发布历史

从 DM v5.4.0 起，TiDB Data Migration 的 Release Notes 合并入相同版本号的TiDB Release Notes。

- 如需阅读 v5.4.0 及之后版本的 DM Release Notes，请查看对应版本的 TiDB Release Notes 中 DM 相关的内容。
- 如需阅读 v5.3.0 及更早版本的 DM Release Notes，请参考以下链接：

10.6.9.13.1 5.3

- 5.3.0

10.6.9.13.2 2.0

- 2.0.7
- 2.0.6
- 2.0.5
- 2.0.4
- 2.0.3
- 2.0.2
- 2.0.1
- 2.0 GA
- 2.0.0-rc.2
- 2.0.0-rc

10.6.9.13.3 1.0

- 1.0.7
- 1.0.6
- 1.0.5
- 1.0.4
- 1.0.3
- 1.0.2

10.7 TiDB Lightning

10.7.1 TiDB Lightning 简介

TiDB Lightning 是用于从静态文件导入 TB 级数据到TiDB 集群的工具，常用于TiDB 集群的初始化数据导入。

要快速了解 Lightning 的基本原理和使用方法，建议先观看下面的培训视频（时长 32 分钟）。注意本视频只为学习参考，具体操作步骤和最新功能，请以文档内容为准。

TiDB Lightning 支持以下文件类型：

- Dumpling 生成的文件

- CSV 文件
- Amazon Aurora 生成的Apache Parquet 文件

TiDB Lightning 支持从以下位置读取：

- 本地
- Amazon S3
- Google GCS

10.7.1.1 TiDB Lightning 整体架构

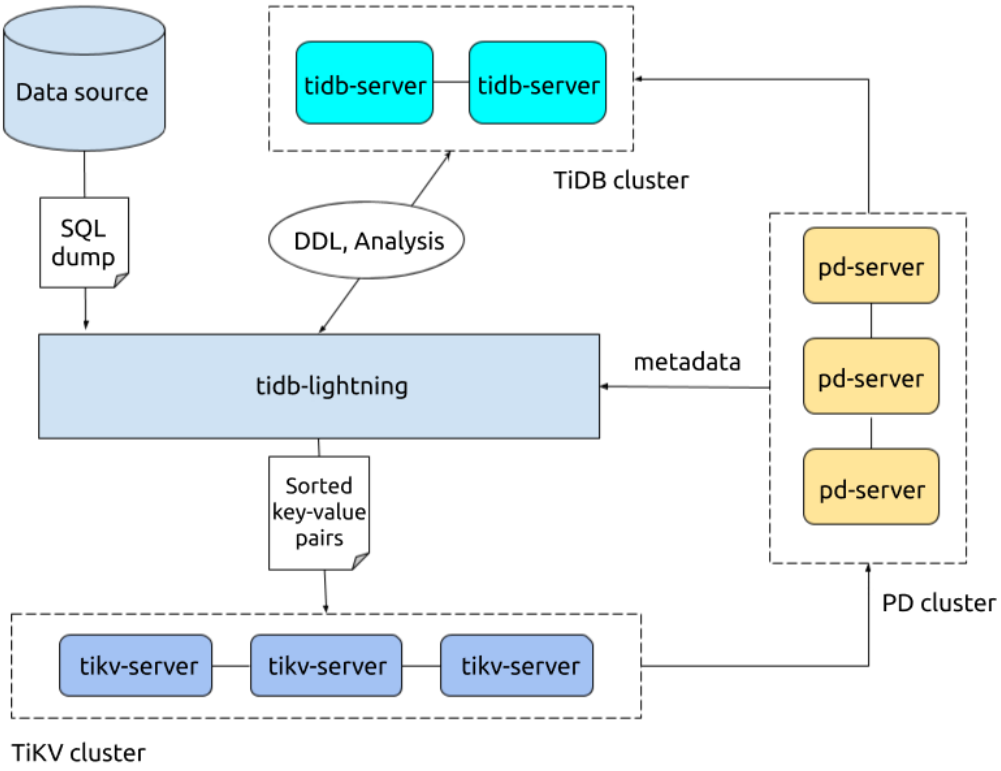


图 186: TiDB Lightning 整体架构

TiDB Lightning 目前支持两种导入方式，通过 backend 配置区分。不同的模式决定TiDB Lightning 如何将数据导入到目标TiDB 集群。

- Physical Import Mode：TiDB Lightning 首先将数据编码成键值对并排序存储在本地临时目录，然后将这些键值对上传到各个TiKV 节点，最后调用TiKV Ingest 接口将数据插入到TiKV 的 RocksDB 中。如果用于初始化导

入，请优先考虑使用 Physical Import Mode，其拥有较高的导入速度。Physical Import Mode 对应的后端模式为 `local`。

- Logical Import Mode：TiDB Lightning 先将数据编码成 SQL，然后直接运行这些 SQL 语句进行数据导入。如果需要导入的集群为生产环境线上集群，或需要导入的目标表中已包含有数据，则应使用 Logical Import Mode。Logical Import Mode 对应的后端模式为 `tidb`。

导入模式	Physical Import Mode	Logical Import Mode
后端	`local`	`tidb`
速度	快 (100 ~ 500 GiB/小时)	慢 (10 ~ 50 GiB/小时)
资源使用率	高	低
占用网络带宽	高	低
导入时是否满足ACID	否	是
目标表	必须为空	可以不为空
支持TiDB 集群版本	>= v4.0.0	全部
导入期间是否允许TiDB 对外提供服务	受限制	是

> **注意：**
>
> 以上性能数据用于对比两种模式的导入性能差异，实际导入速度受硬件配置、表结构、索引数量等多方面因素影响。

10.7.2 TiDB Lightning 快速上手

本教程假设使用的是若干新的、纯净版 CentOS 7 实例，你可以（使用VMware、VirtualBox 及其他工具）在本地虚拟化或在供应商提供的平台上部署一台小型的云虚拟主机。因为TiDB Lightning 对计算机资源消耗较高，建议分配 16 GB 以上的内存以及 32 核以上的 CPU 以获取最佳性能。

> **警告：**
>
> 本教程中的部署方法只适用于测试及功能体验，并不适用于生产或开发环境。

10.7.2.1 准备全量备份数据

我们使用dumpling 从 MySQL 导出数据，如下：

```
tiup dumpling -h 127.0.0.1 -P 3306 -u root -t 16 -F 256MB -B test -f 'test.t[12]' -o /data/
    ↪ my_database/
```

其中：

- `-B test`：从 `test` 数据库导出。
- `-f test.t[12]`：只导出 `test.t1` 和 `test.t2` 这两个表。
- `-t 16`：使用 16 个线程导出数据。
- `-F 256MB`：将每张表切分成多个文件，每个文件大小约为 256 MB。

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

10.7.2.2 部署 TiDB Lightning

10.7.2.2.1 第 1 步：部署 TiDB 集群

在开始数据导入之前，需先部署一套要进行导入的 TiDB 集群。本教程以 TiDB v5.4.0 版本为例，具体部署方法可参考使用 TiUP 部署 TiDB 集群。

10.7.2.2.2 第 2 步：下载 TiDB Lightning 安装包

TiDB Lightning 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。

10.7.2.2.3 第 3 步：启动 `tidb-lightning`

1. 将安装包里的 `bin/tidb-lightning` 及 `bin/tidb-lightning-ctl` 上传至部署 TiDB Lightning 的服务器。
2. 将数据源也上传到同样的服务器。
3. 配置 `tidb-lightning.toml`。

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# 选择使用的导入模式
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径需要是一个空目录
sorted-kv-dir = "/mnt/ssd/sorted-kv-dir"

[mydumper]
# 源数据目录。
data-source-dir = "/data/my_datasource/"

# 配置通配符规则，默认规则会过滤 mysql、 sys、 INFORMATION_SCHEMA、 PERFORMANCE_SCHEMA、
    ↪ METRICS_SCHEMA、 INSPECTION_SCHEMA 系统数据库下的所有表
# 若不配置该项，导入系统表时会出现“找不到 schema”的异常
filter = ['*.*', '!mysql.*', '!sys.*', '!INFORMATION_SCHEMA.*', '!PERFORMANCE_SCHEMA.*', '!
    ↪ METRICS_SCHEMA.*', '!INSPECTION_SCHEMA.*']
```

```
[tidb]
# 目标集群的信息
host = "172.16.31.2"
port = 4000
user = "root"
password = "rootroot"
# 表架构信息在从 TiDB 的“状态端口”获取。
status-port = 10080
# 集群 pd 的地址
pd-addr = "172.16.31.3:2379"
```

4. 配置合适的参数运行 tidb-lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
#!/bin/bash
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out &
```

10.7.2.2.4 第 4 步：检查数据

导入完毕后，TiDB Lightning 会自动退出。若导入成功，日志的最后一行会显示 tidb lightning exit。

如果出错，请参见TiDB Lightning 常见问题。

10.7.2.3 总结

本教程对TiDB Lightning 进行了简单的介绍，并快速部署了一套简单的TiDB Lightning 集群，将全量备份数据导入到 TiDB 集群中。

关于TiDB Lightning 的详细功能和使用，参见TiDB Lightning 简介。

10.7.3 部署TiDB Lightning

本文主要介绍TiDB Lightning 进行数据导入的硬件需求，以及手动部署TiDB Lightning 的方式。Lightning 不同的导入模式，其硬件要求有所不同，请先阅读：

- Physical Import Mode 必要条件及限制
- Logical Import Mode 必要条件及限制

10.7.3.1 使用TiUP 联网部署（推荐）

1. 执行如下命令安装TiUP 工具：

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

安装完成后，~/.bashrc 已将TiUP 加入到路径中，你需要新开一个终端或重新声明全局变量 source ~/.bashrc 来使用TiUP。(也可能是~/.profile，以 TiUP 输出为准。)

2. 安装TiUP Lightning组件：

```
tiup install tidb-lightning
```

10.7.3.2 手动部署

10.7.3.2.1 下载TiDB Lightning安装包

参考工具下载文档下载TiDB Lightning安装包（TiDB Lightning完全兼容较低版本的TiDB集群，建议选择最新稳定版本）。

解压 Lightning 压缩包即可获得 `tidb-lightning` 可执行文件。

```
tar -zxvf tidb-lightning-${version}-linux-amd64.tar.gz
chmod +x tidb-lightning
```

10.7.3.2.2 升级TiDB Lightning

你可以通过替换二进制文件升级TiDB Lightning，无需其他配置。重启 TiDB Lightning 的具体操作参见FAQ。

如果当前有运行的导入任务，推荐任务完成后再升级TiDB Lightning。否则，你可能需要从头重新导入，因为无法保证断点可以跨版本工作。

10.7.4 TiDB Lightning目标数据库要求

使用TiDB Lightning导入数据前，先检查环境是否满足要求，这有助于减少导入过程的错误，避免导入失败的情况。

10.7.4.1 目标数据库权限要求

TiDB Lightning导入数据时，根据导入方式和启用特性等，需要下游数据库用户具备不同的权限，可参考下表：

```
<td></td>
<td>特性</td>
<td>作用域</td>
<td>所需权限</td>
<td>备注</td>
```

```
<td rowspan="2">必需</td>
<td rowspan="2">基本功能</td>
<td>目标 table</td>
<td>CREATE,SELECT,INSERT,UPDATE,DELETE,DROP,ALTER</td>
<td>DROP 仅 tidb-lightning-ctl 在执行 checkpoint-destroy-all 时需要</td>
```

```
<td>目标 database</td>
<td>CREATE</td>
<td></td>
```

```
<td rowspan="4">必需</td>
<td>Logical Import Mode</td>
<td>information_schema.columns</td>
<td>SELECT</td>
<td></td>
```

```
<td  rowspan="3">Physical Import Mode</td>
<td>mysql.tidb</td>
<td>SELECT</td>
<td></td>
```

```
<td>-</td>
<td>SUPER</td>
<td></td>
```

```
<td>-</td>
<td>RESTRICTED_VARIABLES_ADMIN,RESTRICTED_TABLES_ADMIN</td>
<td>当目标 TiDB 开启 SEM</td>
```

```
<td>推荐</td>
<td>冲突检测，max-error</td>
<td>lightning.task-info-schema-name 配置的 schema</td>
<td>SELECT,INSERT,UPDATE,DELETE,CREATE,DROP</td>
<td>如不需要，该值必须设为""</td>
```

```
<td>可选</td>
<td>并行导入</td>
<td>lightning.meta-schema-name 配置的 schema</td>
<td>SELECT,INSERT,UPDATE,DELETE,CREATE,DROP</td>
<td>如不需要，该值必须设为""</td>
```

```
<td>可选</td>
<td>checkpoint.driver = "mysql"</td>
<td>checkpoint.schema 设置</td>
<td>SELECT,INSERT,UPDATE,DELETE,CREATE,DROP</td>
<td>使用数据库而非文件形式存放 checkpoint 信息时需要</td>
```

10.7.4.2 目标数据库所需空间

目标TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标TiKV 集群的总存储空间必须大于数据源大小× 副本数量× 2。例如集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间

- RocksDB 的空间放大效应

目前无法精确计算 Dumpling 从 MySQL 导出的数据大小，但你可以用下面 SQL 语句统计信息表的data_length 字段估算数据量：

统计所有 schema 大小，单位 MiB，注意修改 ${schema_name}

```
SELECT table_schema, SUM(data_length)/1024/1024 AS data_length, SUM(index_length)/1024/1024 AS
    ↪ index_length, SUM(data_length+index_length)/1024/1024 AS sum FROM information_schema.
    ↪ tables WHERE table_schema = "${schema_name}" GROUP BY table_schema;
```

统计最大单表，单位 MiB，注意修改 ${schema_name}

```
SELECT table_name, table_schema, SUM(data_length)/1024/1024 AS data_length, SUM(index_length)
    ↪ /1024/1024 AS index_length, SUM(data_length+index_length)/1024/1024 AS sum FROM
    ↪ information_schema.tables WHERE table_schema = "${schema_name}" GROUP BY table_name,
    ↪ table_schema ORDER BY sum  DESC LIMIT 5;
```

10.7.5 数据源

10.7.5.1 TiDB Lightning 数据源

TiDB Lightning 支持从多种类型的文件导入数据到TiDB 集群。通过以下配置为TiDB Lightning 指定数据文件所在位置。

```
[mydumper]
#### 本地源数据目录或 S3 等外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/
    ↪ tidb/v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"
```

TiDB Lightning 运行时将查找 data-source-dir 中所有符合命令规则的文件。

文件类型	分类	命名规则
Schema 文件	包含 DDL 语句 CREATE ↪ ↪ TABLE ↪ ↪ 的文件	${ ↪ db_name ↪ }. ↪ ${ ↪ table_name ↪ }- ↪ schema ↪ . ↪ sql ↪

文件类型	分类	命名规则
Schema 文件	包含 CREATE DATABASE 语句的文件	${db_name}-schema.sql
数据文件	包含整张表的数据文件，该文件会被导入	${db_name}.${table_name}.${csv|sql|parquet}
数据文件	表如果一个表分布于多个数据文件，这些文件命名需加上文件编号的后缀	${db_name}.${table_name}.001.${csv|sql|parquet}

文件类型	分类	命名规则
压缩文件	上述所有类型文件如带压缩文件名后缀，如 gzip、snappy 或 zstd，TiDB Lightning 会流式解压后进行导入	${db_name}.${table_name}.${csv|sql|parquet}.{compress}

TiDB Lightning 尽量并行处理数据，由于文件必须顺序读取，所以数据处理协程是文件级别的并发（通过 region-concurrency 配置控制）。因此导入大文件时性能比较差。通常建议单个文件尺寸为 256MiB，以获得最好的性能。

10.7.5.1.1 CSV

表结构

CSV 文件是没有表结构的。要导入TiDB，就必须为其提供表结构。可以通过以下任一方法实现：

- 创建包含 DDL 语句的 ${db_name}.${table_name}-schema.sql 和 ${db_name}-schema-create.sql。
- 在 TiDB 中手动创建。

配置

CSV 格式可在 tidb-lightning.toml 文件中 [mydumper.csv] 下配置。大部分设置项在 MySQL [LOAD DATA] 语句中都有对应的项目。

```
[mydumper.csv]
```

```
#### 字段分隔符，支持一个或多个字符，默认值为 ','。如果数据中可能有逗号，
    ↪ 建议源文件导出时分隔符使用非常见组合字符例如'|+|'。
separator = ','
#### 引用定界符，设置为空表示字符串未加引号。
delimiter = '"'
#### 行尾定界字符，支持一个或多个字符。设置为空（默认值）表示 "\n"（换行）和 "\r\n" （回车+换行
    ↪ ），均表示行尾。
terminator = ""
#### CSV 文件是否包含表头。
#### 如果为 true，首行将会被跳过，且基于首行映射目标表的列。
header = true
#### CSV 是否包含 NULL。
#### 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
#### 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
#### 为以下值的字段将会被解析为 NULL。
null = '\N'
#### 是否解析字段内的反斜线转义符。
backslash-escape = true
#### 是否移除以分隔符结束的行。
trim-last-separator = false
```

对于诸如 `separator`，`delimiter` 和 `terminator` 等取值为字符串的配置项，如果需要设置的字符串中包含特殊字符，可以通过使用反斜杠`\` 转义的方式进行输入，输入的转义序列必须被包含在一对双引号" 之间。例如，设置 `separator = "\u001f"` 表示使用ASCII 字符 0X1F 作为字符串定界符。

你也可以使用单引号字符串 `'...'` 禁止对字符进行转义。

另外，设置 `separator = '\n'` 表示使用两个字符`\` + `n` 作为字符串定界符，而不是转义后的换行符 `\n`。

更多详细的内容请参考 TOML v1.0.0 标准。

`separator`

- 指定字段分隔符。
- 可以为一个或多个字符，不能为空。
- 常用值：
  - CSV 用 `','`
  - TSV 用 `"\t"`
  - “˜0001” 表示使用ASCII 字符 0x01
- 对应 LOAD DATA 语句中的 `FIELDS TERMINATED BY` 项。

`delimiter`

- 指定引用定界符。

- 如果 `delimiter` 为空，所有字段都会被取消引用。
- 常用值：
  - `'"'` 使用双引号引用字段，和RFC 4180 一致。
  - `''` 不引用
- 对应 LOAD DATA 语句中的 `FIELDS ENCLOSED BY` 项。

`terminator`

- 指定行尾定界符。
- 如果 `terminator` 为空，则“\n”（换行）和“\r\n”（回车+换行）均表示行尾。
- 对应 LOAD DATA 语句中的 `LINES TERMINATED BY` 项。

`header`

- 是否所有CSV 文件都包含表头行。
- 如为true，第一行会被用作列名。如为 false，第一行并无特殊性，按普通的数据行处理。

`not-null` 和 `null`

- `not-null` 决定是否所有字段不能为空。
- 如果 `not-null` 为 false，设定了 `null` 的字符串会被转换为 SQL NULL 而非具体数值。
- 引用不影响字段是否为空。
  例如有如下CSV 文件：

```
A,B,C
\N,"\N",
```

  在默认设置（`not-null = false; null = '\N'`）下，列 A and B 导入TiDB 后都将会转换为 NULL。列 C 是空字符串 `''`，但并不会解析为 NULL。

`backslash-escape`

- 是否解析字段内的反斜线转义符。
- 如果 `backslash-escape` 为true，下列转义符会被识别并转换。

转义符	转换为
\0	空字符(U+0000)
\b	退格(U+0008)
\n	换行(U+000A)
\r	回车(U+000D)
\t	制表符(U+0009)
\Z	Windows EOF (U+001A)

其他情况下（如 \"）反斜线会被移除，仅在字段中保留其后面的字符（"），这种情况下，保留的字符仅作为普通字符，特殊功能（如界定符）都会失效。

- 引用不会影响反斜线转义符的解析与否。
- 对应 LOAD DATA 语句中的 FIELDS ESCAPED BY '\' 项。

trim-last-separator

- 将 separator 字段当作终止符，并移除尾部所有分隔符。

  例如有如下CSV 文件：

```
A,,B,,
```

- 当 trim-last-separator = false，该文件会被解析为包含 5 个字段的行 ('A', '', 'B', '', '')。
- 当 trim-last-separator = true，该文件会被解析为包含 3 个字段的行 ('A', '', 'B')。
- 此配置项已被弃用，建议使用兼容性更好的 terminator。

  如果有如下旧的配置：

```
separator = ','
trim-last-separator = true
```

  建议修改为：

```
separator = ','
terminator = ",\n" # 请根据文件实际使用的换行符指定为 ",\n" 或 ",\r\n"
```

不可配置项

TiDB Lightning 并不完全支持 LOAD DATA 语句中的所有配置项。例如：

- 不可使用行前缀(LINES STARTING BY)。
- 不可跳过表头(IGNORE n LINES)。如有表头，必须是有效的列名。

启用严格格式

导入文件的大小统一约为 256 MB 时，TiDB Lightning 可达到最佳工作状态。如果导入单个 CSV 大文件，TiDB Lightning 只能使用一个线程来处理，这会降低导入速度。

要解决此问题，可先将CSV 文件分割为多个文件。对于通用格式的CSV 文件，在没有读取整个文件的情况下无法快速确定行的开始和结束位置。因此，默认情况下 TiDB Lightning 不会自动分割CSV 文件。但如果你确定待导入的CSV 文件符合特定的限制要求，则可以启用 strict-format 设置。启用后，TiDB Lightning 会将单个CSV 大文件分割为单个大小为 256 MB 的多个文件块进行并行处理。

```
[mydumper]
strict-format = true
```

严格格式的CSV 文件中，每个字段仅占一行，即必须满足以下条件之一：

- delimiter 为空；
- 每个字段不包含 `terminator` 对应的字符串。在默认配置下，对应每个字段不包含 CR(`\r`) 或 LF(`\n`)。

如果 CSV 文件不是严格格式但 `strict-format` 被误设为 `true`，跨多行的单个完整字段会被分割成两部分，导致解析失败，甚至不报错地导入已损坏的数据。

常见配置示例

CSV

默认设置已按照 RFC 4180 调整。

```
[mydumper.csv]
separator = ',' # 如果数据中可能有逗号，建议源文件导出时分隔符使用非常见组合字符例如'|+|'
delimiter = '"'
header = true
not-null = false
null = '\N'
backslash-escape = true
```

示例内容：

```
ID,Region,Count
1,"East",32
2,"South",\N
3,"West",10
4,"North",39
```

TSV

```
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
null = 'NULL'
backslash-escape = false
```

示例内容：

```
ID      Region      Count
1       East        32
2       South       NULL
3       West        10
4       North       39
```

TPC-H DBGEN

```
[mydumper.csv]
separator = '|'
delimiter = ''
terminator = "|\n"
header = false
not-null = true
backslash-escape = false
```

示例内容：

```
1|East|32|
2|South|0|
3|West|10|
4|North|39|
```

10.7.5.1.2 SQL

TiDB Lightning 在处理 SQL 文件时，由于无法对单个文件进行快速分割，因此无法通过增加并发提高单个文件的导入速度。鉴于此，导出数据为 SQL 文件时应尽量避免单个 SQL 文件过大，通常单文件在 256MiB 左右可以达到最佳性能。

10.7.5.1.3 Parquet

TiDB Lightning 目前仅支持由Amazon Aurora 或者 Hive 导出快照生成的 Parquet 文件。要识别其在 S3 的文件组织形式，需要使用如下配置匹配到所有的数据文件：

```
[[mydumper.files]]
#### 解析 AWS Aurora parquet 文件所需的表达式
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$'
schema = '$1'
table = '$2'
type = '$3'
```

注意，此处仅说明Aurora snapshot 导出的parquet 文件如何匹配。Schema 文件需要单独导出及处理。

关于 `mydumper.files`，请参考自定义文件匹配。

10.7.5.1.4 压缩文件

TiDB Lightning 目前支持由 Dumpling 导出的压缩文件或满足符合上文命名规则的压缩文件，目前支持 `gzip`、`snappy`、`zstd` 压缩算法的压缩文件。在文件名符合命名规则时，TiDB Lightning 会自动识别压缩算法在流式解压后导入，无需额外配置。

> **注意**

- 由于TiDB Lightning 无法对单个大压缩文件进行并发解压，因此压缩文件的大小会直接影响导入速度。建议压缩数据文件解压后的源文件大小不超过 256 MiB。
- TiDB Lightning 仅支持导入各自独立压缩的数据文件，不支持导入多个数据文件组成的单个压缩文件集合包。
- TiDB Lightning 不支持二次压缩的 `parquet` 文件，例如 `db.table.parquet.snappy`。如需压缩 `parquet` 文件，你可以配置 `parquet` 文件数据存储的压缩格式。
- TiDB v6.4.0 及之后版本的TiDB Lightning 支持后缀为压缩算法 `gzip`、`snappy`、`zstd`，以及后缀名 `.bak` 的数据文件。其他后缀名会报错。你需要提前修改文件名，或将该类文件移出导入数据目录来避免此类错误。

10.7.5.1.5 自定义文件匹配

TiDB Lightning 仅识别符合命名要求的数据文件，但在某些情况下已提供的数据文件并不符合要求，因此可能出现TiDB Lightning 在极短的时间结束，处理文件数量为 0 的情况。

为了解决此类问题，TiDB Lightning 提供了 `[[mydumper.files]]` 配置用于通过自定义表达式匹配数据文件。

以 AWS Aurora 导出至 S3 的快照文件为例，Parquet 文件的完整路径为：`S3://some-bucket/some-subdir/some-database/some-database.some-table/part-00000-c5a881bb-58ff-4ee6-1111-b41ecff340a3-c000.gz.parquet`。

通常 `data-source-dir` 会被配置为`S3://some-bucket/some-subdir/some-database/` 以导入 `some-database` 库。

根据上述 Parquet 文件的路径， 你可以编写正则表达式 `(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$`， 得到的 match group 中 index=1 的内容为 `some-database`，index=2 的内容为 `some-table`，index=3 的内容为 `parquet`。

根据上述正则表达式及相应的index 编写配置文件，TiDB Lightning 即可识别非默认命名规则的文件，最终实际配置如下：

```
[[mydumper.files]]
#### 解析 AWS Aurora parquet 文件所需的表达式
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$'
schema = '$1'
table = '$2'
type = '$3'
```

- schema：目标库名称，值可以为：
  - 正则表达式匹配到的 group 序号，例如“$1”。
  - 直接填写期望导入的库名，例如“db1”。所有匹配到的文件均会导入“db1”。
- table：目标表名称，值可以为：
  - 正则表达式匹配到的 group 序号，例如“$2”。
  - 直接填写期望导入的库名，例如“table1”。所有匹配到的文件均会导入“table1”。
- type：文件类型，支持`sql`，`parquet`，`csv`，值可以为：

  - 正则表达式匹配到的 group 序号，例如“$3”。

- key：文件的序号，即前文所述${db_name}.${table_name}.001.csv中的001。
  - 正则表达式匹配到的 group 序号，例如“$4”。

10.7.5.1.6 从 Amazon S3 导入数据

如下为从Amazon S3 导入数据的示例，更多配置参数描述，可参考外部存储 URI 格式。

- 使用本地已设置的权限访问 S3：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup'
```

- 使用路径类型的请求模式：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d  's3://my-bucket/sql-backup?force-path-style=true&endpoint=http://10.154.10.132:8088'
```

- 使用 AWS IAM 角色的 ARN 来访问 S3 数据：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d  's3://my-bucket/test-data?role-arn=arn:aws:iam::888888888888:role/my-role'
```

- 使用 AWS IAM 用户密钥来访问 S3 数据：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/test-data?access_key={my_access_key}&secret_access_key={
        ↪ my_secret_access_key}'
```

- 使用 AWS IAM 角色的密钥以及会话令牌来访问 S3 数据：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/test-data?access_key={my_access_key}&secret_access_key={
        ↪ my_secret_access_key}&session-token={my_session_token}'
```

10.7.5.1.7 更多

- 使用 Dumpling 导出到 CSV 文件
- LOAD DATA

10.7.6 Physical Import Mode

10.7.6.1 Physical Import Mode 简介

Physical Import Mode 是TiDB Lightning 支持的一种数据导入方式。Physical Import Mode 不经过 SQL 接口，而是直接将数据以键值对的形式插入TiKV 节点，是一种高效、快速的导入模式。使用 Physical Import Mode 时，单个 Lightning 实例可导入的数据量为 10 TiB，理论上导入的数据量可以随着 Lightning 实例数量的增加而增加，目前已经有多个用户验证基于并行导入功能可以导入的数据量达 20 TiB。

使用前请务必自行阅读必要条件及限制。

Physical Import Mode 对应的后端模式为 `local`。

10.7.6.1.1 原理说明

1. 在导入数据之前，`tidb-lightning` 会自动将TiKV 节点切换为“导入模式” (import mode)，优化写入效率并停止自动压缩。以下为各版本的TiDB Lightning 决定暂停调度的策略：

    - 自 v7.1.0 开始，你可以通过`pause-pd-scheduler-scope` 来控制是否暂停全局调度。
    - v6.2.0 ~ v7.0.0 版本的TiDB Lightning 会根据TiDB 集群的版本决定是否暂停全局调度。当TiDB 集群版本 >= v6.1.0，只会暂停目标表数据范围所在 Region 的调度，并在目标表导入完成后恢复调度。其他版本则会暂停全局调度。
    - 当 TiDB Lightning 版本 < v6.2.0 时，`tidb-lightning` 会暂停全局调度。

2. `tidb-lightning` 在目标数据库建立表结构，并获取其元数据。

    如果将 `add-index-by-sql` 设置为 `true`，`tidb-lightning` 会使用 SQL 接口添加索引，并且会在导入数据前移除目标表的所有次级索引。

3. 每张表都会被分割为多个连续的区块，这样来自大表(200 GB+) 的数据就可以多个并发导入。

4. `tidb-lightning` 会为每一个区块准备一个“引擎文件(engine file)” 来处理键值对。`tidb-lightning` 会并发读取 SQL dump，将数据源转换成与TiDB 相同编码的键值对，然后将这些键值对排序写入本地临时存储文件中。

5. 当一个引擎文件数据写入完毕时，`tidb-lightning` 便开始对目标TiKV 集群数据进行分裂和调度，然后导入数据到TiKV 集群。

    引擎文件包含两种：数据引擎与索引引擎，各自又对应两种键值对：行数据和次级索引。通常行数据在数据源里是完全有序的，而次级索引是无序的。因此，数据引擎文件在对应区块写入完成后会被立即上传，而所有的索引引擎文件只有在整张表所有区块编码完成后才会执行导入。

    注意当 `tidb-lightning` 使用 SQL 接口添加索引时（即 `add-index-by-sql` 设置为 `true`），索引引擎将不会写入数据，因为此时目标表的次级索引已经在第 2 步中被移除。

6. 整张表相关联的所有引擎文件完成导入后，`tidb-lightning` 会对比本地数据源及下游集群的校验和(checksum)，确保导入的数据无损，然后添加在第 2 步中被移除的次级索引或让TiDB 分析(ANALYZE) 这些新增的数据，以优化日后的操作。同时，`tidb-lightning` 调整 `AUTO_INCREMENT` 值防止之后新增数据时发生冲突。

    表的自增ID 是通过行数的上界估计值得到的，与表的数据文件总大小成正比。因此，最后的自增ID 通常比实际行数大得多。这属于正常现象，因为在TiDB 中自增ID 不一定是连续分配的。

7. 在所有步骤完毕后，tidb-lightning 自动将TiKV 切换回“普通模式”(normal mode)，并恢复可能被暂停的全局调度，此后TiDB 集群可以正常对外提供服务。

10.7.6.1.2 必要条件及限制

运行环境需求

操作系统：建议使用新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将TiDB Lightning 和其他组件（比如 tikv-server）部署在同一台机器上，然后设置 region-concurrency 配置项的值为逻辑 CPU 数的 75%，以限制TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：

建议使用 32 核以上的 CPU 和 64 GiB 以上内存以获得更好的性能。

> 注意：
>
> 导入大量数据时，一个并发对内存的占用在 2 GiB 左右，也就是说总内存占用最大可达到 region-concurrency * 2 GiB。region-concurrency 默认与逻辑 CPU 的数量相同。如果内存的大小（GiB）小于逻辑 CPU 数量的两倍或运行时出现 OOM，需要手动调低 region-concurrency 参数以避免TiDB Lightning OOM。

存储空间：配置项 sorted-kv-dir 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。建议与 data-source-dir 使用不同的存储设备，独占IO 会获得更好的导入性能，且建议优先考虑配置闪存等高性能存储介质。

网络：建议使用 10 Gbps 以太网卡。

版本要求

- TiDB Lightning 版本≥ 4.0.3。
- TiDB 集群版本≥ v4.0.0。
- 如果目标TiDB 集群是 v3.x 或以下的版本，需要使用Importer-backend 来完成数据的导入。在这个模式下，tidb-lightning 需要将解析的键值对通过 gRPC 发送给 tikv-importer 并由 tikv-importer 完成数据的导入。

使用限制

- 请勿直接使用 Physical Import Mode 向已经投入生产的TiDB 集群导入数据，这将对在线业务产生严重影响。如需向生产集群导入数据，请参考导入时限制调度范围从集群降低到表级别。
- 默认情况下，不应同时启动多个TiDB Lightning 实例向同一TiDB 集群导入数据，而应考虑使用并行导入特性。
- 使用多个TiDB Lightning 向同一目标导入时，请勿混用不同的 backend，即不可同时使用 Physical Import Mode 和 Logical Import Mode 导入同一TiDB 集群。

- 在导入数据的过程中，请勿在目标表进行写操作，否则会导致导入失败或数据不一致。导入期间也不建议进行读操作，因为读取的数据可能不一致。请在导入操作完成后再进行读写操作。
- 单个 Lightning 进程导入单表不应超过 10 TB。使用并行导入时，Lightning 实例不应超过 10 个。

与其他组件一同使用的注意事项

- TiDB Lightning 与 TiFlash 一起使用时需要注意：
  - 无论是否已为一张表创建TiFlash 副本，你都可以使用TiDB Lightning 导入数据至该表。但该场景下，TiDB Lightning 导入数据耗费的时间更长，具体取决于 TiDB Lightning 部署机器的网卡带宽、TiFlash 节点的 CPU 及磁盘负载及TiFlash 副本数等因素。
- TiDB Lightning 字符集相关的注意事项：
  - TiDB Lightning 在 v5.4.0 之前不支持导入 `charset=GBK` 的表。
- TiDB Lightning 与 TiCDC 一起使用时需要注意：
  - TiCDC 无法捕获 Physical Import Mode 插入的数据。

10.7.6.2 使用 Physical Import Mode

本文档介绍如何编写Physical Import Mode 的配置文件，如何进行性能调优、使用磁盘资源配额等内容。

10.7.6.2.1 配置及使用

可以通过以下配置文件使用 Physical Import Mode 执行数据导入：

```
[lightning]
#### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14

#### 启动之前检查集群是否满足最低需求。
check-requirements = true

[mydumper]
#### 本地源数据目录或外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/tidb/
    ↪ v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"

[tikv-importer]
#### 导入模式配置，设为 local 即使用 Physical Import Mode
backend = "local"
```

```
#### 冲突数据处理方式
duplicate-resolution = 'remove'

#### 本地进行 KV 排序的路径。
sorted-kv-dir = "./some-dir"

#### 限制 TiDB Lightning 向每个 TiKV 节点写入的带宽大小，默认为 0，表示不限制。
#### store-write-bwlimit = "128MiB"

#### Physical Import Mode 是否通过 SQL 方式添加索引。默认为 `false`，表示 TiDB Lightning
    '→ 会将行数据以及索引数据都编码成 KV pairs 后一同导入 TiKV，实现机制和历史版本保持一致。
     '→ 如果设置为 `true`，即 TiDB Lightning 会导完数据后，再使用 add index 的 SQL 来添加索引。
#### 通过 SQL 方式添加索引的优点是将导入数据与导入索引分开，可以快速导入数据，即使导入数据后，
    '→ 索引添加失败，也不会影响数据的一致性。
#### add-index-by-sql = false

[tidb]
#### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
#### 设置连接 TiDB 的密码，可为明文或 Base64 编码。
password = ""
#### 必须配置。表结构信息从 TiDB 的“status-port”获取。
status-port = 10080
#### 必须配置。pd-server 的地址，填一个即可。
pd-addr = "172.16.31.4:2379"
#### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
#### 设置 TiDB 库的日志等级。
log-level = "error"

[post-restore]
 #### 配置是否在导入完成后对每一个表执行 `ADMIN CHECKSUM TABLE <table>` 操作来验证数据的完整性。
#### 可选的配置项：
#### - "required"（默认）。在导入完成后执行 CHECKSUM 检查，如果 CHECKSUM 检查失败，则会报错退出。
#### - "optional"。在导入完成后执行 CHECKSUM 检查，如果报错，会输出一条 WARN 日志并忽略错误。
#### - "off"。导入结束后不执行 CHECKSUM 检查。
#### 默认值为 "required"。从 v4.0.8 开始，checksum 的默认值由此前的 "true" 改为 "required"。
#### # 注意：
#### 1. Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
#### 2. 考虑到与旧版本的兼容性，依然可以在本配置项设置 `true` 和 `false` 两个布尔值，其效果与 `
    '→ required` 和 `off` 相同。
checksum = "required"
#### 配置是否在 CHECKSUM 结束后对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
```

```
#### 此配置的可选配置项与 `checksum` 相同，但默认值为 "optional"。
analyze = "optional"
```

Lightning 的完整配置文件可参考完整配置及命令行参数。

10.7.6.2.2 冲突数据检测

冲突数据，即两条或两条以上的记录存在 PK/UK 列数据重复的情况。当数据源中的记录存在冲突数据，将导致该表真实总行数和使用唯一索引查询的总行数不一致的情况。冲突数据检测支持三种策略：

- record: 仅将冲突记录添加到目的TiDB 中的 lightning_task_info.conflict_error_v1 表中。注意，该方法要求目的TiKV 的版本为 v5.2.0 或更新版本。如果版本过低，则会启用 'none' 模式。
- remove: 推荐方式。记录所有的冲突记录，和 'record' 模式相似。但是会删除所有的冲突记录，以确保目的TiDB 中的数据状态保持一致。
- none: 关闭冲突数据检测。该模式是三种模式中性能最佳的，但是可能会导致目的TiDB 中出现数据不一致的情况。

在 v5.3 版本之前，Lightning 不具备冲突数据检测特性，若存在冲突数据将导致导入过程最后的checksum 环节失败；开启冲突检测特性的情况下，无论 record 还是 remove 策略，只要检测到冲突数据，Lightning 都会跳过最后的checksum 环节（因为必定失败）。

假设一张表 order_line 的表结构如下：

```
CREATE TABLE IF NOT EXISTS `order_line` (
  `ol_o_id` int(11) NOT NULL,
  `ol_d_id` int(11) NOT NULL,
  `ol_w_id` int(11) NOT NULL,
  `ol_number` int(11) NOT NULL,
  `ol_i_id` int(11) NOT NULL,
  `ol_supply_w_id` int(11) DEFAULT NULL,
  `ol_delivery_d` datetime DEFAULT NULL,
  `ol_quantity` int(11) DEFAULT NULL,
  `ol_amount` decimal(6,2) DEFAULT NULL,
  `ol_dist_info` char(24) DEFAULT NULL,
  PRIMARY KEY (`ol_w_id`,`ol_d_id`,`ol_o_id`,`ol_number`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

若在导入过程中检测到冲突数据，则可以查询 lightning_task_info.conflict_error_v1 表得到以下内容：

```
mysql> select table_name,index_name,key_data,row_data from conflict_error_v1 limit 10;
+--
    '→ --------------------+-------------+--------------------+
    '→
|  table_name          | index_name | key_data | row_data
    '→                                                                                |
+--
    '→                    +            +          +
    '→
       -------------------- ------------- --------------------
```

```
| `tpcc`.`order_line` | PRIMARY    | 21829216 | (2677, 10, 10, 11, 75656, 10,  NULL, 5, 5831.97, "
    '→ HT5DN3EVb6kWTd4L37bsbogj") |
| `tpcc`.`order_line` | PRIMARY    | 49931672 | (2677, 10, 10, 11, 75656, 10,  NULL, 5, 5831.97, "
    '→ HT5DN3EVb6kWTd4L37bsbogj") |
| `tpcc`.`order_line` | PRIMARY    | 21829217 | (2677, 10, 10, 12, 76007, 10,  NULL, 5, 9644.36, "
    '→ bHuVoRfidQOq2rJ6ZC9Hd12E") |
| `tpcc`.`order_line` | PRIMARY    | 49931673 | (2677, 10, 10, 12, 76007, 10,  NULL, 5, 9644.36, "
    '→ bHuVoRfidQOq2rJ6ZC9Hd12E") |
| `tpcc`.`order_line` | PRIMARY    | 21829218 | (2677, 10, 10, 13, 85618, 10,  NULL, 5, 7427.98, "
    '→ t3rsesgi9rVAKi9tf6an5Rpv") |
| `tpcc`.`order_line` | PRIMARY    | 49931674 | (2677, 10, 10, 13, 85618, 10,  NULL, 5, 7427.98, "
    '→ t3rsesgi9rVAKi9tf6an5Rpv") |
| `tpcc`.`order_line` | PRIMARY    | 21829219 | (2677, 10, 10, 14, 15873, 10,  NULL, 5, 133.21, "
    '→ z1vH0e31tQydJGhfNYNa4ScD")  |
| `tpcc`.`order_line` | PRIMARY    | 49931675 | (2677, 10, 10, 14, 15873, 10,  NULL, 5, 133.21, "
    '→ z1vH0e31tQydJGhfNYNa4ScD")  |
| `tpcc`.`order_line` | PRIMARY    | 21829220 | (2678, 10, 10, 1, 44644, 10,  NULL, 5, 8463.76, "
    '→ TWKJBt5iJA4eF7FIVxnugNmz")  |
| `tpcc`.`order_line` | PRIMARY    | 49931676 | (2678, 10, 10, 1, 44644, 10,  NULL, 5, 8463.76, "
    '→ TWKJBt5iJA4eF7FIVxnugNmz")  |
+--
    '→                    +            +----------------------------------------------------------------------------------
    '→
10 rows in set (0.14 sec)
```

根据上述信息人工甄别需要保留的重复数据，手动插回原表即可。

10.7.6.2.3 导入时暂停 PD 调度的范围

自 TiDB Lightning v6.2.0 版本起，TiDB Lightning 提供机制控制导入数据过程对在线业务的影响。TiDB Lightning 不会暂停全局的调度，而是只暂停目标表数据范围所在region 的调度，降低了对在线业务的影响。

自 TiDB v7.1.0 版本起，你可以通过TiDB Lightning 的参数pause-pd-scheduler-scope 来控制暂停调度的范围。默认为 "table"，即只暂停目标表数据所在 Region 的调度。当集群没有业务流量时，建议设置为 "global" 以减少来自调度器对导入的干扰。

> **注意：**
>
> TiDB Lightning 不支持导入数据到已有业务写入的数据表。
>
> TiDB 集群版本需大于等于 v6.1.0，更低的版本 TiDB Lightning 会保持原有行为，暂停全局调度，数据导入期间会给在线业务带来严重影响。

TiDB Lightning 默认情况下会在最小范围内暂停集群调度，无需额外配置。但默认配置下，TiDB 集群仍然会因为数据导入太快，使在线业务的性能受到影响，所以你需要额外配置几个选项来控制导入速度和其他可能影响

集群性能的因素：

```
[tikv-importer]
#### 限制 TiDB Lightning 向每个 TiKV 节点写入的带宽大小。
store-write-bwlimit = "128MiB"

[tidb]
#### 使用更小的并发以降低计算 checksum 和执行 analyze 对事务延迟的影响。
distsql-scan-concurrency = 3

[cron]
#### 避免将 TiKV 切换到 import 模式。
switch-mode = '0'
```

在测试中用 TPCC 测试模拟在线业务，同时用TiDB Lightning 向 TiDB 集群导入数据，测试导入数据对 TPCC 测试结果的影响。测试结果如下：

线程数	TPM	P99	P90	AVG
1	20%~30%	60%~80%	30%~50%	30%~40%
8	15%~25%	70%~80%	35%~45%	20%~35%
16	20%~25%	55%~85%	35%~40%	20%~30%
64	无显著影响			
256	无显著影响			

表格中的百分比含义为TiDB Lightning 导入对 TPCC 结果的影响大小。对于 TPM，数值表示 TPM 下降的百分比；对于延迟 P99、 P90、 AVG，数值表示延迟上升的百分比。

测试结果表明，TPCC 并发越小，TiDB Lightning 导入对TPCC 结果影响越大。当TPCC 并发达到64 或以上时，Lightning 导入对 TPCC 结果无显著影响。

因此，如果你的TiDB 生产集群上有延迟敏感型业务，并且并发较小，强烈建议不使用TiDB Lightning 导入数据到该集群，这会给在线业务带来较大影响。

10.7.6.2.4 性能调优

提高 Lightning Physical Import Mode 导入性能最直接有效的方法：

- 升级 Lightning 所在节点的硬件，尤其重要的是 CPU 和 sorted-key-dir 所在存储设备的性能。
- 使用并行导入特性实现水平扩展。

当然，Lightning 也提供了部分并发相关配置以影响 Physical Import Mode 的导入性能。但是从长期实践的经验总结来看，以下四个配置项一般保持默认值即可，调整其数值并不会带来显著的性能提升，可作为了解内容阅读。

```
[lightning]
#### 引擎文件的最大并行数。
#### 每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”。
```

```
#### 这两项设置控制两种引擎文件的最大并发数。
index-concurrency = 2
table-concurrency = 6

#### 数据的并发数。默认与逻辑 CPU 的数量相同。
#### region-concurrency =

#### I/O 最大并发数。I/O 并发量太高时，会因硬盘内部缓存频繁被刷新
#### 而增加 I/O 等待时间，导致缓存未命中和读取速度降低。
#### 对于不同的存储介质，此参数可能需要调整以达到最佳效率。
io-concurrency = 5
```

导入时，每张表被切分成一个用于存储索引的 “索引引擎” 和若干存储行数据的 “数据引擎”，index-concurrency用于调整 “索引引擎” 的并发度。

在调整 index-concurrency 时，需要注意 index-concurrency * 每个表对应的源文件数量 > region-concurrency 以确保 cpu 被充分利用，一般比例大概在 1.5 ~ 2 左右为优。index-concurrency 不应该设置的过大，但不低于 2 (默认)，过大会导致太多导入的pipeline 变差，大量index-engine 的 import 阶段堆积。

table-concurrency 同理，需要确保table-concurrency * 每个表对应的源文件数量 > region-concurrency 以确保cpu 被充分利用。推荐值为region-concurrency * 4 / 每个表对应的源文件数量左右，最少设置为 4.

如果表非常大，Lightning 会按照 100 GiB 的大小将表分割成多个批次处理，并发度由 table-concurrency 控制。

上述两个参数对导入速度影响不大，使用默认值即可。

io-concurrency 用于控制文件读取并发度，默认值为 5。可以认为在某个时刻只有 5 个句柄在执行读操作。由于文件读取速度一般不会是瓶颈，所以使用默认值即可。

读取文件数据后，lightning 还需要做后续处理，例如将数据在本地进行编码和排序。此类操作的并发度由region-concurrency 配置控制。region-concurrency 的默认值为CPU 核数，通常无需调整，建议不要将Lightning 与其它组件部署在同一主机，如果客观条件限制必须混合部署，则需要根据实际负载调低 region-concurrency。

此外，TiKV 的num-threads 配置也可能影响性能，新集群建议设置为 CPU 核数。

10.7.6.2.5 磁盘资源配额从 v6.2.0 版本开始引入

TiDB Lightning 在使用物理模式导入数据时，会在本地磁盘创建大量的临时文件，用来对原始数据进行编码、排序、分割。当用户本地磁盘空间不足时，TiDB Lightning 会由于写入文件失败而报错退出。

为了减少这种情况的发生，你可以为TiDB Lightning 配置磁盘配额(disk quota)。当磁盘配额不足时，TiDB Lightning 会暂停读取源数据以及写入临时文件的过程，优先将已经完成排序的 key-value 写入到TiKV，TiDB Lightning 删除本地临时文件后，再继续导入过程。

为TiDB Lightning 开启磁盘配额，你需要在配置文件中加入配置项：

```
[tikv-importer]
#### 默认为 MaxInt64，即 9223372036854775807 字节
disk-quota = "10GB"
backend = "local"
```

```
[cron]
#### 检查磁盘配额的时间间隔，默认为 60 秒。
check-disk-quota = "30s"
```

disk-quota 配置项可以设置磁盘配额，默认为 MaxInt64，即 9223372036854775807 字节，这个值远远大于实际情况中可能需要的磁盘空间，相当于没开启。

check-disk-quota 配置项是检查磁盘配额的时间间隔，默认为 60 秒。由于检查临时文件使用空间的过程需要加锁，会使所有的导入线程都暂停，如果在每次写入之前都检查一次磁盘空间的使用情况，则会大大降低写入文件的效率（相当于单线程写入）。为了维持高效的写入，磁盘配额不会在每次写入之前检查，而是每隔一段时间暂停所有线程的写入并检查当前磁盘空间的使用情况。也就是说，当 check-disk-quota 配置项设置为一个非常大的值时，磁盘的使用空间有可能会大大超出磁盘配额，这样的情况下，磁盘配额功能可以说是不生效的。因此，check-disk-quota 的值建议不要设置太大，而具体设置多少则需要由TiDB Lightning 具体运行的环境决定，因为不同的环境下，TiDB Lightning 写入临时文件的速度是不一样的。理论上来说，写入临时文件的速度越快，check-disk-quota 需要设置得越小。

10.7.7 Logical Import Mode

10.7.7.1 Logical Import Mode 简介

Logical Import Mode 是 TiDB Lightning 支持的一种数据导入方式。在 Logical Import Mode 下，TiDB Lightning 先将数据编码成 SQL，然后直接运行这些 SQL 语句进行数据导入。对于已有数据、对外提供服务的TiDB 集群，推荐使用 Logical Import Mode 导入数据。Logical Import Mode 的行为与正常执行 SQL 并无差异，可保证ACID。

Logical Import Mode 对应的后端模式为 tidb。

10.7.7.1.1 必要条件

操作系统：

建议使用新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将TiDB Lightning 和其他组件（比如 tikv-server）部署在同一台机器上，然后设置 region-concurrency 配置项的值为逻辑 CPU 数的 75%，以限制TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：

建议使用 4 核以上的 CPU 和 8 GiB 以上内存以获得更好的性能。根据长期的实践经验，Lightning 的 Logical Import Mode 没有显著（5 GiB 以上）的内存占用，但上调 region-concurrency 默认值将导致内存量增加。

网络：建议使用 1 Gbps 或 10 Gbps 以太网卡。

10.7.7.1.2 使用限制

使用多个TiDB Lightning 向同一目标导入时，请勿混用不同的backend，即不可同时使用 Physical Import Mode 和 Logical Import Mode 导入同一TiDB 集群。

10.7.7.2 使用 Logical Import Mode

本文档介绍如何编写Logical Import Mode 的配置文件，如何进行性能调优等内容。

10.7.7.2.1 配置及使用

可以通过以下配置文件使用 Logical Import Mode 执行数据导入：

```
[lightning]
#### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14

#### 启动之前检查集群是否满足最低需求。
check-requirements = true

[mydumper]
#### 本地源数据目录或外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/tidb/
    ↪ v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"

[tikv-importer]
#### 导入模式配置，设为 tidb 即使用 Logical Import Mode
backend = "tidb"

#### Logical Import Mode 插入重复数据时执行的操作。
#### - replace：新数据替代已有数据
#### - ignore：保留已有数据，忽略新数据
#### - error：中止导入并报错
on-duplicate = "replace"

[tidb]
#### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
#### 设置连接 TiDB 的密码，可为明文或 Base64 编码。
password = ""
#### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
#### 设置 TiDB 库的日志等级。
log-level = "error"
```

TiDB Lightning 的完整配置文件可参考完整配置及命令行参数。

10.7.7.2.2 冲突数据检测

冲突数据，即两条或两条以上的记录存在主键或唯一键列数据重复的情况。当数据源中的记录存在冲突数据，将导致该表真实总行数和使用唯一索引查询的总行数不一致的情况。TiDB Lightning 的 Logical Import Mode 通

过 on-duplicate 配置冲突数据检测的策略，TiDB Lightning 根据策略使用不同的 SQL 语句进行插入。

策略	冲突时默认行为	对应 SQL 语句
replace	新数据替代旧数据	REPLACE INTO ...
ignore	保留旧数据，忽略新数据	INSERT IGNORE INTO ...
error	中止导入	INSERT INTO ...

10.7.7.2.3 性能调优

- TiDB Lightning 的 Logical Import Mode 性能很大程度上取决于目标TiDB 集群的写入性能，当遇到性能瓶颈时可参考TiDB 相关性能优化文档。
- 如果发现目标TiDB 集群的的写入尚未达到瓶颈，可以考虑增加 Lightning 配置中 region-concurrency 的值。region-concurrency 默认值为 CPU 核数，其含义在 Physical Import Mode 和 Logical Import Mode 下有所不同，Logical Import Mode 的 region-concurrency 表示写入并发数。配置示例：

```
[lightning]
region-concurrency = 32
```

- 调整目标TiDB 集群的raftstore.apply-pool-size 和raftstore.store-pool-size 参数也可能提升导入速度。

10.7.8 TiDB Lightning 前置检查

从TiDB 5.3.0 开始，TiDB Lightning 增加了前置检查功能，可以在执行迁移前检查配置。默认开启。该功能会自动进行一些磁盘空间和执行配置的常规检查，主要目的是确保后续的整个导入过程顺利。

下表介绍了各检查项和详细解释。

检查项	支持版本	解释
集群版本/状态是否正常	>= 5.3.0	检查配置中集群是否可以连接，Physical Import Mode 还会检查 TiKV/PD/TiFlash 版本是否支持。
是否有权限读取数据	>= 5.3.0	检查当从云存储（Amazon S3）读取数据的时候，是否有对应的权限，确保不会因权限缺失导致导入中断。

检查项	支持版本	解释
导入空间是否足够	>= 5.3.0	检查 TiKV 集群是否有足够空间导入数据。检查时会对数据源进行采样，通过采样结果预估索引大小占比。由于估算中考虑了索引，因此可能会出现尽管数据源大小低于本地盘可用空间，但依然无法通过检测的情况。
	1306	Physical Import Mode

检查项	支持版本	解释
Region 分布状态	>= 5.3.0	检查 TiKV 集群的 Region 分布是否均匀，以及是否存在大量空 region，如果空 Region 的数量大于 max(1000, 表的数量 * 3)，即大于“1000”和“3 倍表数量”二者中的较大者，TiDB Lightning 无法执行导入。

检查项	支持版本	解释
数据文件是否有大 CSV 文件	>= 5.3.0	当备份文件中出现大于 10 GiB 的 CSV 文件且无法进行自动切分 (Strict-For-mat=false) 的时候，会导致导入性能下降。该检查的目的是提醒用户确保数据格式的情况下，开启自动切分 CSV 功能。

检查项	支持版本	解释
是否可以从断点恢复	>= 5.3.0	该检查是确保断点恢复过程中，不会出现对源文件和数据库中 schema 进行修改，导致导入错误数据的情况。

检查项	支持版本	解释
是否可以导入数据到已存在的数据表中	>= 5.3.0	当导入到已创建好的数据表中时，该检查尽可能的检查此次导入的源文件是否和已存在的数据表匹配。检查列数是否匹配，如果源文件存在列名，则检查列名是否匹配。当源文件存在缺省列，则检查缺省列在数据表中是否存在
	1310	

检查项	支持版本	解释
导入的目标表是否为空	>= 5.3.1	如果导入的目标表不为空，则 TiDB Lightning 会自动报错退出；如果开启了并行导入模式 (incremental-import = true)，则会跳过此检查项。

10.7.9 表库过滤

TiDB 数据迁移工具默认情况下作用于所有数据库，但实际使用中，往往只需要作用于其中的部分子集。例如，用户只想处理 `foo*` 和 `bar*` 形式的表，而无需对其他表进行操作。

从 TiDB 4.0 起，所有TiDB 数据迁移工具都使用一个通用的过滤语法来定义子集。本文档介绍如何使用表库过滤功能。

10.7.9.1 使用表库过滤

10.7.9.1.1 命令行

在命令行中使用多个 `-f` 或 `--filter` 参数，即可在TiDB 数据迁移工具中应用表库过滤规则。每个过滤规则均采用 `db.table` 形式，支持通配符（详情见下一节）。以下为各个工具中的使用示例：

- BR：

```
./br backup full -f 'foo*.*' -f 'bar*.*' -s 'local:///tmp/backup'
```

```
./br restore full -f 'foo*.*' -f 'bar*.*' -s 'local:///tmp/backup'
```

- Dumpling：

```
./dumpling -f 'foo*.*' -f 'bar*.*' -P 3306 -o /tmp/data/
```

- TiDB Lightning：

```
./tidb-lightning -f 'foo*.*' -f 'bar*.*' -d /tmp/data/ --backend tidb
```

10.7.9.1.2 TOML 配置文件

在 TOML 文件中，表库过滤规则以字符串数组的形式指定。以下为各个工具中的使用示例：

- TiDB Lightning：

```
[mydumper]
filter = ['foo*.*', 'bar*.*']
```

- TiCDC：

```
[filter]
rules = ['foo*.*', 'bar*.*']

[[sink.dispatchers]]
matcher = ['db1.*', 'db2.*', 'db3.*']
dispatcher = 'ts'
```

10.7.9.2 表库过滤语法

10.7.9.2.1 直接使用表名

每条表库过滤规则由“库”和“表”组成，两部分之间以英文句号(.)分隔。只有表名与规则完全相符的表才会被接受。

```
db1.tbl1
db2.tbl2
db3.tbl3
```

表名只由有效的标识符组成，例如：

- 数字（0 到 9）
- 字母（a 到 z，A 到 Z）

- $
- _
- 非 ASCII 字符（U+0080 到 U+10FFFF）

其他ASCII 字符均为保留字。部分标点符号有特殊含义，详情见下一节。

10.7.9.2.2 使用通配符

表名的两个部分均支持使用通配符（详情见 fnmatch(3)）。

- `*`：匹配零个或多个字符。
- `?`：匹配一个字符。
- `[a-z]`：匹配“a”和“z”之间的一个字符。
- `[!a-z]`：匹配不在“a”和“z”之间的一个字符。

```
db[0-9].tbl[0-9a-f][0-9a-f]
data.*
*.backup_*
```

此处，“字符”指的是一个 Unicode 码位，例如：

- U+00E9 “é” 是 1 个字符。
- U+0065，U+0301 “é” 是 2 个字符。
- U+1F926 U+1F3FF U+200D U+2640 U+FE0F “🤦🏿‍♀️” 是 5 个字符。

10.7.9.2.3 使用文件导入

如需导入一个文件作为过滤规则，请在规则的开头加上一个“@”来指定文件名。库表过滤解析器将导入文件中的每一行都解析为一条额外的过滤规则。

例如，`config/filter.txt` 文件有以下内容：

```
employees.*
*.WorkOrder
```

以下两条表库过滤命令是等价的：

```
./dumpling -f '@config/filter.txt'
./dumpling -f 'employees.*' -f '*.WorkOrder'
```

导入的文件里不能使用过滤规则导入另一个文件。

10.7.9.2.4 注释与空行

导入的过滤规则文件中，每一行开头和结尾的空格都会被去除。此外，空行（空字符串）也将被忽略。

行首的# 表示该行是注释，会被忽略。而不在行首的# 则会被认为是语法错误。

```
### 这是一行注释
db.table   # 这一部分不是注释，且可能引起错误
```

10.7.9.2.5 排除规则

在一条过滤规则的开头加上`!`，则表示符合这条规则的表不会被TiDB 数据迁移工具处理。通过应用排除规则，库表过滤可以作为屏蔽名单来使用。

```
*.*
#^ 注意：必须先添加 *.* 规则来包括所有表
!*.Password
!employees.salaries
```

10.7.9.2.6 转义字符

如果需要将特殊字符转化为标识符，可以在特殊字符前加上反斜杠`\`。

```
db\.with\.dots.*
```

为了简化语法并向上兼容，不支持下列字符序列：

- 在行尾去除空格后使用`\`（使用`[ ]`来匹配行尾的空格）。
- 在`\`后使用数字或字母(`[0-9a-zA-Z]`)。特别是类似 C 的转义序列，如 `\0`、`\r`、`\n`、`\t` 等序列，目前在表库过滤规则中无意义。

10.7.9.2.7 引号包裹的标识符

除了`\`之外，还可以用`"`和`` ` ``来控制特殊字符。

```
"db.with.dots"."tbl\1"
`db.with.dots`.`tbl\2`
```

也可以通过输入两次引号，将引号包含在标识符内。

```
"foo""bar".`foo``bar`
### 等价于：
foo\"bar.foo\`bar
```

用引号包裹的标识符不可以跨越多行。

用引号只包裹标识符的一部分是无效的，例如：

```
"this is "invalid*.*
```

10.7.9.2.8 正则表达式

如果你需要使用较复杂的过滤规则，可以将每个匹配模型写为正则表达式，以`/`为分隔符：

```
/^db\d{2,}$/./^tbl\d{2,}$/
```

这类正则表示使用Go dialect。只要标识符中有一个子字符串与正则表达式匹配，则视为匹配该模型。例如，`/b/` 匹配 db01。

> 注意：
>
> 正则表达式中的每一个`/`都需要转义为`\/`，包括在`[...]`里面的`/`。不允许在`\Q...\E`之间放置一个未转义的`/`。

10.7.9.3 使用多个过滤规则

当表的名称与过滤列表中所有规则均不匹配时，默认情况下这些表被忽略。

要建立一个屏蔽名单，必须使用显式的 `*.*` 作为第一条过滤规则，否则所有表均被排除。

```
### 所有表均被过滤掉
./dumpling -f '!*.Password'

### 只有 “Password” 表被过滤掉，其余表仍保留
./dumpling -f '*.*' -f '!*.Password'
```

如果一个表的名称与过滤列表中的多个规则匹配，则以最后匹配的规则为准。例如：

```
### rule 1
employees.*
### rule 2
!*.dep*
### rule 3
*.departments
```

过滤结果如下：

表名	规则 1	规则 2	规则 3	结果
irrelevant.table				默认（拒绝）
employees.employees	√			规则 1（接受）
employees.dept_emp	√	√		规则 2（拒绝）
employees.departments	√	√	√	规则 3（接受）
else.departments		√	√	规则 3（接受）

> 注意：
>
> 在TiDB 数据迁移工具的默认配置中，系统库总是被排除。系统库有以下六个：
>
> - INFORMATION_SCHEMA

- PERFORMANCE_SCHEMA
- METRICS_SCHEMA
- INSPECTION_SCHEMA
- mysql
- sys

10.7.10 TiDB Lightning 断点续传

大量的数据导入一般耗时数小时至数天，长时间运行的进程会有一定机率发生非正常中断。如果每次重启都从头开始，就会浪费掉之前已成功导入的数据。为此，TiDB Lightning 提供了 “断点续传” 的功能，即使 tidb-lightning 崩溃，在重启时仍然接着之前的进度继续工作。

本文主要介绍TiDB Lightning 断点续传的启用与配置、断点的存储，以及断点续传的控制。

10.7.10.1 断点续传的启用与配置

```
[checkpoint]
### 启用断点续传。
### 导入时，TiDB Lightning 会记录当前进度。
### 若 TiDB Lightning 或其他组件异常退出，在重启时可以避免重复再导入已完成的数据。
enable = true

### 存储断点的方式
### - file：存放在本地文件系统（要求 v2.1.1 或以上）
### - mysql：存放在兼容 MySQL 的数据库服务器
driver = "file"

### 存储断点的架构名称（数据库名称）
### 仅在 driver = "mysql" 时生效
### schema = "tidb_lightning_checkpoint"

### 断点的存放位置
### # 若 driver = "file"，此参数为断点信息存放的文件路径。
### 如果不设置该参数则默认为 `/tmp/CHECKPOINT_SCHEMA.pb`
### # 若 driver = "mysql"，此参数为数据库连接参数 (DSN)，格式为“用户:密码@tcp(地址:端口)/”。
### 默认会重用 [tidb] 设置目标数据库来存储断点。
### 为避免加重目标集群的压力，建议另外使用一个兼容 MySQL 的数据库服务器。
### dsn = "/tmp/tidb_lightning_checkpoint.pb"

### 导入成功后是否保留断点。默认为删除。
### 保留断点可用于调试，但有可能泄漏数据源的元数据。
### keep-after-success = false
```

10.7.10.2 断点的存储

TiDB Lightning 支持两种存储方式：本地文件或 MySQL 数据库。

- 若 `driver = "file"`，断点会存放在一个本地文件，其路径由 `dsn` 参数指定。由于断点会频繁更新，建议将这个文件放到写入次数不受限制的盘上，例如RAM disk。
- 若 `driver = "mysql"`，断点可以存放在任何兼容 MySQL 5.7 或以上的数据库中，包括 MariaDB 和 TiDB。在没有选择的情况下，默认会存在目标数据库里。

目标数据库在导入期间会有大量的操作，若使用目标数据库来存储断点会加重其负担，甚至有可能造成通信超时丢失数据。因此，强烈建议另外部署一台兼容 MySQL 的临时数据库服务器。此数据库也可以安装在 `tidb-lightning` 的主机上。导入完毕后可以删除。

10.7.10.3 断点续传的控制

若 `tidb-lightning` 因不可恢复的错误而退出（例如数据出错），重启时不会使用断点，而是直接报错离开。为保证已导入的数据安全，这些错误必须先解决掉才能继续。使用 `tidb-lightning-ctl` 工具可以标示已经恢复。

10.7.10.3.1 `--checkpoint-error-destroy`

```
tidb-lightning-ctl  --checkpoint-error-destroy='`schema`.`table`'
```

该命令会让失败的表从头开始整个导入过程。选项中的架构和表名必须以反引号(`) 包裹，而且区分大小写。

- 如果导入 `` `schema`.`table` `` 这个表曾经出错，这条命令会：
    1. 从目标数据库移除(DROP) 这个表，清除已导入的数据。
    2. 将断点重设到“未开始”的状态。
- 如果 `` `schema`.`table` `` 没有出错，则无操作。

传入“all” 会对所有表进行上述操作。这是最方便、安全但保守的断点错误解决方法：

```
tidb-lightning-ctl  --checkpoint-error-destroy=all
```

10.7.10.3.2 `--checkpoint-error-ignore`

```
tidb-lightning-ctl --checkpoint-error-ignore='`schema`.`table`' &&
tidb-lightning-ctl --checkpoint-error-ignore=all
```

如果导入 `` `schema`.`table` `` 这个表曾经出错，这条命令会清除出错状态，如同没事发生过一样。传入“all”会对所有表进行上述操作。

> 注意：
>
> 除非确定错误可以忽略，否则不要使用这个选项。如果错误是真实的话，可能会导致数据不完全。启用校验和(CHECKSUM) 可以防止数据出错被忽略。

10.7.10.3.3 --checkpoint-remove

```
tidb-lightning-ctl --checkpoint-remove='`schema`.`table`' &&
tidb-lightning-ctl --checkpoint-remove=all
```

无论是否有出错，把表的断点清除。

10.7.10.3.4 --checkpoint-dump

```
tidb-lightning-ctl --checkpoint-dump=output/directory
```

将所有断点备份到传入的文件夹，主要用于技术支持。此选项仅于 `driver = "mysql"` 时有效。

10.7.11 TiDB Lightning 并行导入

TiDB Lightning 的Physical Import Mode 从 v5.3.0 版本开始支持单表或多表数据的并行导入。通过支持同步启动多个实例，并行导入不同的单表或多表的不同数据，使TiDB Lightning 具备水平扩展的能力，可大大降低导入大量数据所需的时间。

在技术实现上，TiDB Lightning 通过在目标TiDB 中记录各个实例以及每个导入表导入数据的元信息，协调不同实例的Row ID 分配范围、全局 Checksum 的记录和TiKV 及 PD 的配置变更与恢复。

TiDB Lightning 并行导入可以用于以下场景：

- 并行导入分库分表的数据。在该场景中，来自上游多个数据库实例中的多个表，分别由不同的TiDB Lightning 实例并行导入到下游TiDB 数据库中。
- 并行导入单表的数据。在该场景中，存放在某个目录中或云存储（如 Amazon S3）中的多个单表文件，分别由不同的TiDB Lightning 实例并行导入到下游TiDB 数据库中。该功能为 v5.3.0 版本引入的新功能。

> 注意：
>
> - 并行导入只支持初始化TiDB 的空表，不支持导入数据到已有业务写入的数据表，否则可能会导致数据不一致的情况。
> - 并行导入一般用于 Physical Import 模式，需要设置 `incremental-import = true`
> - 并行导入一般用于 Physical Import 模式；虽然也能用于 Logical Import 模式，但是一般不会带来明显的性能提升。

10.7.11.1 使用说明

使用TiDB Lightning并行导入需要设置 `incremental-import = true`。TiDB Lightning在启动时，会在下游TiDB中注册元信息，并自动检测是否有其他实例向目标集群导入数据。如果有，则自动进入并行导入模式。

但是在并行导入时，需要注意以下情况：

- 解决主键或者唯一索引的冲突
- 导入性能优化

10.7.11.1.1 解决主键或者唯一索引的冲突

在使用Physical Import Mode 并行导入时，需要确保多个TiDB Lightning 的数据源之间，以及它们和TiDB 的目标表中的数据没有主键或者唯一索引的冲突，并且导入的目标表不能有其他应用进行数据写入。否则，TiDB Lightning将无法保证导入结果的正确性，并且导入完成后相关的数据表将处于数据索引不一致的状态。

10.7.11.1.2 导入性能优化

由于TiDB Lightning 需要将生成的 Key-Value 数据上传到对应Region 的每一个副本所在的TiKV 节点，其导入速度受目标集群规模的限制。在通常情况下，建议确保目标TiDB 集群中的TiKV 实例数量与TiDB Lightning 的实例数量大于n:1 (n 为 Region 的副本数量)。同时，在使用TiDB Lightning并行导入模式时，为达到最优性能，建议进行如下限制：

- 每个TiDB Lightning 部署在单独的机器上面。TiDB Lightning 默认会消耗所有的 CPU 资源，在单台机器上面部署多个实例并不能提升性能。
- 每个TiDB Lightning 实例导入的源文件总大小不超过 5 TiB
- TiDB Lightning 实例的总数量不超过 10 个

在使用TiDB Lightning并行导入分库分表数据的时候，请根据数据量大小选择使用的TiDB Lightning 实例数量。

- 如果 MySQL 数据量小于 2 TiB，可以使用 1 个 TiDB Lightning 实例进行并行导入
- 如果 MySQL 数据量超过 2 TiB，并且 MySQL 实例总数小于 10 个，建议每个 MySQL 实例对应 1 个TiDB Lightning 实例，而且并行TiDB Lightning 实例数量不要超过 10 个
- 如果 MySQL 数据量超过 2 TiB，并且 MySQL 实例总数超过 10 个，建议这些 MySQL 实例导出的数据平均分配 5 到 10 个 TiDB Lightning 实例进行导入

接下来，本文档将以两个并行导入的示例，详细介绍了不同场景下并行导入的操作步骤：

- 示例 1：使用 Dumpling + TiDB Lightning 并行导入分库分表数据至TiDB
- 示例 2：使用TiDB Lightning 并行导入单表数据

10.7.11.1.3 使用限制

TiDB Lightning 在运行时，需要独占部分资源，因此如果需要在单台机器上面部署多个TiDB Lightning 实例(不建议生产环境使用) 或多台机器共享磁盘存储时，需要注意如下使用限制：

- 每个TiDB Lightning 实例的tikv-importer.sorted-kv-dir 必须设置为不同的路径。多个实例共享相同的路径会导致非预期的行为，可能导致导入失败或数据出错。
- 每个TiDB Lightning 的 checkpoint 需要分开存储。checkpoint 的详细配置见TiDB Lightning 断点续传。
  - 如果设置checkpoint.driver = “file”（默认值），需要确保每个实例设置的 checkpoint 的路径不同。
  - 如果设置checkpoint.driver = “mysql”，需要为每个实例设置不同的 schema。
- 每个TiDB Lightning 的 log 文件应该设置为不同的路径。共享同一个log 文件将不利于日志的查询和排查问题。
- 如果开启Web 界面或 Debug API，需要为每个实例的 `lightning.status-addr` 设置不同地址，否则，TiDB Lightning 进程会由于端口冲突无法启动。

10.7.11.2 示例 1：使用 Dumpling + TiDB Lightning 并行导入分库分表数据至TiDB

在本示例中，假设上游为包含 10 个分表的 MySQL 集群，总共大小为 10 TiB。使用 5 个 TiDB Lightning 实例并行导入，每个实例导入 2 TiB 数据，预计可以将总导入时间（不包含 Dumping 导出的耗时）由约 40 小时降至约 10 小时。

假设上游的库名为 `my_db`，每个分表的名字为 `my_table_01 ~ my_table_10`，需要合并导入至下游的 `my_db.my_table` 表中。下面介绍具体的操作步骤。

10.7.11.2.1 第 1 步：使用 Dumpling 导出数据

在部署TiDB Lightning 的 5 个节点上面分别导出两个分表的数据：

- 如果两个分表位于同一个 MySQL 实例中，可以直接使用 Dumpling 的 `--filter` 参数一次性导出。此时在使用TiDB Lightning 导入时，指定 `data-source-dir` 为 Dumpling 数据导出的目录即可；
- 如果两个分表的数据分布在不同的 MySQL 节点上，则需要使用 Dumpling 分别导出，两次导出数据需要放置在同一父目录下不同子目录里，然后在使用TiDB Lightning 导入时，`data-source-dir` 指定为此父级目录。

使用 Dumpling 导出数据的步骤，请参考Dumpling。

10.7.11.2.2 第 2 步：配置TiDB Lightning 的数据源

创建 `tidb-lightning.toml` 配置文件，并加入如下内容：

```
[lightning]
status-addr = ":8289"

[mydumper]
### 设置为 Dumpling 导出数据的路径，如果 Dumpling 执行了多次并分属不同的目录，
    ′→ 请将多次导出的数据置放在相同的父目录下并指定此父目录即可。
data-source-dir = "/path/to/source-dir"

[tikv-importer]
### 是否允许向已存在数据的表导入数据。默认值为 false。
### 当使用并行导入模式时，由于多个 TiDB Lightning 实例同时导入一张表，因此此开关必须设置为 true。
```

```
incremental-import = true
### "local": Physical Import Mode，默认使用。适用于 TB 级以上大数据量，但导入期间下游 TiDB
    ↪ 无法对外提供服务。
### "tidb": Logical Import Mode。TB 级以下数据量也可以采用，下游 TiDB 可正常提供服务。
backend = "local"

### 设置本地排序数据的路径
sorted-kv-dir = "/path/to/sorted-dir"
```

如果数据源存放在Amazon S3 或GCS 等外部存储中，需要额外的连接配置，你可以为这类配置指定参数。如下例子假设数据源存放在Amazon S3 中：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/tmp/
    ↪ sorted-kvs \
    -d 's3://my-bucket/sql-backup'
```

更多参数设置，请参考外部存储 URI 格式。

10.7.11.2.3 第 3 步：开启TiDB Lightning 进行数据导入

在使用TiDB Lightning 并行导入时，对每个TiDB Lightning 节点机器配置的需求与非并行导入模式相同，每个TiDB Lightning 节点需要消耗相同的资源，建议部署在不同的机器上。详细的部署步骤，请参考TiDB Lightning 部署与执行

依次在每台机器上面运行TiDB Lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
### !/bin/bash
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out &
```

在并行导入的场景下，TiDB Lightning 在启动任务之后，会自动进行下列检查：

- 检查本地盘空间（即 `sort-kv-dir` 配置）以及TiKV 集群是否有足够空间导入数据，空间大小的详细说明参考TiDB Lightning 下游数据库所需空间和TiDB Lightning 运行时资源要求。检查时会对数据源进行采样，通过采样结果预估索引大小占比。由于估算中考虑了索引，因此可能会出现尽管数据源大小低于本地盘可用空间，但依然无法通过检测的情况。
- 检查TiKV 集群的region 分布是否均匀，以及是否存在大量空region，如果空region 的数量大于max(1000, 表的数量*3)，即大于“1000”和“3 倍表数量”二者中的最大者，则无法执行导入。
- 检查数据源导入数据是否有序， 并且根据检查结果自动调整 `mydumper.batch-size` 的大小。因此 `mydumper.batch-size` 配置不再对用户开放。

你也可以通过 `lightning.check-requirements` 配置来关闭检查，执行强制导入。更多详细检查内容，可以查看Lightning 执行前检查项。

10.7.11.2.4 第 4 步：查看进度

开始导入后，可以通过以下任意方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度。详情请参见TiDB Lightning 监控。

等待所有的TiDB Lightning 运行结束，则整个导入完成。

10.7.11.3 示例 2：使用TiDB Lightning 并行导入单表数据

TiDB Lightning 也支持并行导入单表的数据。例如，将存放在Amazon S3 中的多个单表文件，分别由不同的TiDB Lightning 实例并行导入到下游TiDB 数据库中。该方法可以加快整体导入速度。TiDB Lightning 使用云端存储时的配置参数与 BR 大致相同，关于详细的参数配置，可以参考外部存储 URI 格式。

> **注意：**
>
> 在本地环境下，可以使用 Dumpling 的 `--filesize` 或 `--where` 参数，预先将单表的数据划分成不同的部分导出至多台机器的本地磁盘，此时依然可以使用并行导入功能，其配置与示例 1 相同。

假设通过 Dumpling 导出的源文件存放在Amazon S3 云存储中，数据文件为 `my_db.my_table.00001.sql ~ my_db.my_table.10000.sql` 共计 10000 个 SQL 文件。如果希望使用 2 个TiDB Lightning 实例加速导入，则需要在配置文件中增加如下设置：

```
[[mydumper.files]]
### db schema 文件
pattern = '(?i)^(?:[^/]*/)*my_db-schema-create\.sql'
schema = "my_db"
type = "schema-schema"

[[mydumper.files]]
### table schema 文件
pattern = '(?i)^(?:[^/]*/)*my_db\.my_table-schema\.sql'
schema = "my_db"
table = "my_table"
type = "table-schema"

[[mydumper.files]]
### 只导入 00001~05000 这些数据文件并忽略其他文件
pattern = '(?i)^(?:[^/]*/)*my_db\.my_table\.(0[0-4][0-9][0-9][0-9]|05000)\.sql'
schema = "my_db"
table = "my_table"
type = "sql"

[tikv-importer]
### 是否允许向已存在数据的表导入数据。默认值为 false。
### 当使用并行导入模式时，由于多个 TiDB Lightning 实例同时导入一张表，因此此开关必须设置为 true。
```

```
incremental-import = true
```

另外一个实例的配置修改为只导入 05001 ~ 10000 数据文件即可。

其他步骤请参考示例 1 中的相关步骤。

10.7.11.4 错误处理

10.7.11.4.1 部分TiDB Lightning 节点异常终止

在并行导入过程中，如果一个或多个TiDB Lightning 节点异常终止，需要首先根据日志中的报错明确异常退出的原因，然后根据错误类型做不同处理：

- 如果是正常退出，例如手动 Kill 或内存溢出被操作系统终止等，可以在适当调整配置后直接重启TiDB Lightning，无须任何其他操作。
- 如果是不影响数据正确性的报错，例如网络超时，请按以下步骤解决：
    1. 在每一个失败的节点上，执行checkpoint-error-ignore 命令，值设置为 `all`，以清除断点续传源数据中记录的错误。
    2. 重启这些异常的节点，从断点位置继续导入。
- 如果在日志中看到影响数据正确性的报错，如checksum mismatched，表示源文件中有非法的数据，请按以下步骤解决：
    1. 在每一个 Lightning 节点（包括成功导入数据的节点）上执行checkpoint-error-destroy 命令，以清除失败的表中已导入的数据，并将这些表的checkpoint 状态重置为“not yet started”。
    2. 使用`filter` 参数在所有TiDB Lightning 节点（包括任务正常结束的节点）上重新配置和导入失败表的数据。重新配置任务时，不要将checkpoint-error-destroy 命令放在每一个 Lightning 节点的启动脚本中，否则会删除多个并行导入任务使用的共享元数据，可能会导致数据导入过程出现问题。例如，如果启动了第二个 Lightning 导入任务，它将删除第一个数据导入任务写入的元数据，导致数据导入异常。

10.7.11.4.2 导入过程中报错“Target table is calculating checksum. Please wait until the checksum is finished and try again”

在部分并行导入场景，如果表比较多或者一些表的数据量比较小，可能会出现当一个或多个任务开始处理某个表的时候，此表对应的其他任务已经完成，并开始数据一致性校验。此时，由于数据一致性校验不支持写入其他数据，对应的任务会返回“Target table is calculating checksum. Please wait until the checksum is finished and try again”错误。等校验任务完成，重启这些失败的任务，报错会消失，数据的正确性也不会受影响。

10.7.12 TiDB Lightning 错误处理功能

从 TiDB 5.4.0 开始，你可以配置TiDB Lightning 以跳过诸如无效类型转换、唯一键冲突等错误，让导入任务持续进行，就如同出现错误的行数据不存在一样。你可以依据生成的报告，手动修复这些错误。该功能适用于以下场景：

- 要导入的数据有少许错误
- 手动定位错误比较困难
- 如果遇到错误就重启TiDB Lightning，代价太大

本文介绍了类型错误处理功能 (`lightning.max-error`) 和重复问题处理功能 (`tikv-importer.duplicate-resolution`) 的使用方法，以及保存这些错误的数据库 (`lightning.task-info-schema-name`)，并提供了一个示例。

10.7.12.1 类型错误(Type error)

你可以通过修改配置项 `lightning.max-error` 来增加数据类型相关的容错数量。如果设置为 `N`，那么 TiDB Lightning 允许数据源中出现N 个错误，而且会跳过这些错误，一旦超过这个错误数就会退出。默认值为 `0`，表示不允许出现错误。

这些错误会被记录到数据库中。在导入完成后，你可以查看数据库中的数据，手动进行处理。请参见错误报告。

```
[lightning]
max-error = 0
```

该配置对下列错误有效：

- 无效值。例如：在INT 列设置了 `'Text'`
- 数字溢出。例如：在TINYINT 列设置了 500
- 字符串溢出。例如: 在 VARCHAR(5) 列中设置了'非常长的文字'
- 零日期时间，如 `'0000-00-00'` 和 `'2021-12-00'`
- 在 NOT NULL 列中设置了 NULL
- 生成的列表达式求值失败
- 列计数不匹配。行中数值的数量和列的数量不一致
- `on-duplicate = "error"` 时，TiDB 后端的唯一键/主键冲突
- 其他 SQL 错误

下列错误是致命错误，不能通过配置 `max-error` 跳过：

- 原始CSV、 SQL 或者 Parquet 文件中的语法错误，例如未闭合的引号
- I/O、 网络、或系统权限错误

在 Physical Import Mode 下，唯一键/主键的冲突是单独处理的。相关内容将在接下来的章节进行介绍。

10.7.12.2 错误报告

如果TiDB Lightning 在运行过程中收集到报错的记录，则在退出时会同时在终端和日志中输出各个类型报错数量的统计信息。

- 输出在终端的报错统计如下表所示：

#	ERROR TYPE	ERROR COUNT	ERROR DATA TABLE
1	Data Type	1000	lightning_task_info.type_error_v1

- 输出在TiDB Lightning 的 log 文件的结尾如下：

```
[2022/03/13 05:33:57.736 +08:00] [WARN] [errormanager.go:459] ["Detect 1000 data type errors
    ↪  in total, please refer to table `lightning_task_info`.`type_error_v1` for more
    ↪ details"]
```

所有错误都会写入下游TiDB 集群 `lightning_task_info` 数据库中的表中。在导入完成后，如果收集到报错的数据，你可以根据数据库中记录的内容，手动进行处理。

你可以使用 `lightning.task-info-schema-name` 配置更改数据库名称。

```
[lightning]
task-info-schema-name = 'lightning_task_info'
```

在此数据库中，TiDB Lightning 创建了 3 个表：

```
CREATE TABLE syntax_error_v1
    ( task_id    bigint NOT NULL,
    create_time datetime(6) NOT NULL DEFAULT now(6),
    table_name  varchar(261) NOT NULL,
    path        varchar(2048) NOT NULL,
    offset      bigint NOT NULL,
    error       text NOT NULL,
    context     text
);
CREATE TABLE type_error_v1
    ( task_id    bigint NOT NULL,
    create_time datetime(6) NOT NULL DEFAULT now(6),
    table_name  varchar(261) NOT NULL,
    path        varchar(2048) NOT NULL,
    offset      bigint NOT NULL,
    error       text NOT NULL,
    row_data    text NOT NULL
);
CREATE TABLE conflict_error_v1
    ( task_id    bigint NOT NULL,
    create_time datetime(6) NOT NULL DEFAULT now(6),
    table_name  varchar(261) NOT NULL,
    index_name  varchar(128) NOT NULL,
    key_data    text NOT NULL,
    row_data    text  NOT  NULL,
    raw_key     mediumblob NOT NULL,
    raw_value   mediumblob NOT NULL,
```

```
    raw_handle  mediumblob  NOT  NULL,
    raw_row     mediumblob  NOT  NULL,
    KEY (task_id, table_name)
);
```

type_error_v1 记录由 `max-error` 配置项管理的所有类型错误(Type error)。每个错误一行。

conflict_error_v1 记录所有后端中的唯一键/主键冲突，每对冲突有两行。

列名	语法	类型	冲突	说明
task_id	√	√	√	生成此错误的TiDB Lightning 任务 ID
create_time	√	√	√	记录错误的时间
table_name	√	√	√	包含错误的表的名称，格式为 ``db`.`tbl``
path	√	√		包含错误文件的路径
offset	√	√		文件中发现错误的字节位置
error	√	√		错误信息
context	√			围绕错误的文本
index_name			√	冲突中唯一键的名称。主键冲突为 'PRIMARY'
key_data			√	导致错误的行的格式化键句柄。该内容仅供人参考，机器不可读
row_data		√	√	导致错误的格式化行数据。该内容仅供人参考，机器不可读
raw_key			√	冲突的 KV 对的键
raw_value			√	冲突的 KV 对的值
raw_handle			√	冲突行的行句柄
raw_row			√	冲突行的编码值

注意：

错误报告记录的是文件偏移量，不是行号或列号，因为行号或列号的获取效率很低。你可以使用下列命令在字节位实现快速跳转（以 183 为例）：

- shell：输出前面几行

```
head -c 183 file.csv | tail
```

- shell，输出后面几行

```
tail -c +183 file.csv | head
```

- vim：`:goto 183` 或 `183go`

10.7.12.3 示例

在该示例中，我们准备了一个包含一些已知错误的数据源。以下是处理这些错误的具体步骤：

1. 准备数据库和表结构：

```
mkdir example && cd example
echo 'CREATE SCHEMA example;' > example-schema-create.sql
echo 'CREATE TABLE t(a TINYINT PRIMARY KEY, b VARCHAR(12) NOT NULL UNIQUE);' > example.t-
    '→ schema.sql
```

2. 准备数据：

```
cat <<EOF > example.t.1.sql
    INSERT INTO t (a, b) VALUES
    (0, NULL),              -- 列不为空
    (1, 'one'),
    (2, 'two'),
    (40, 'forty'),          -- 与下面的 `40` 冲突
    (54, 'fifty-four'),     -- 与下面的 ``'fifty-four'`` 冲突
    (77, 'seventy-seven'), -- 字符串长度超过 12 个字符
    (600, 'six hundred'),   -- 数字超出了 TINYINT 数据类型支持的范围
    (40, 'forty'),         -- 与上面的 `40` 冲突
    (42, 'fifty-four');     -- 与上面的 ``'fifty-four'`` 冲突
EOF
```

3. 配置TiDB Lightning，启用严格 SQL 模式，使用 Local 后端模式进行导入，通过删除解决重复项，并最多跳过 10 个错误：

```
cat <<EOF > config.toml
    [lightning]
    max-error = 10
    [tikv-importer]
    backend = 'local'
    sorted-kv-dir = '/tmp/lightning-tmp/'
    duplicate-resolution = 'remove'
    [mydumper]
    data-source-dir = '.'
    [tidb]
    host = '127.0.0.1'
    port = 4000
    user = 'root'
    password = ''
    sql-mode = 'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE'
EOF
```

4. 运行TiDB Lightning。因为已跳过所有错误，该命令执行完会成功退出：

```
tiup tidb-lightning -c config.toml
```

5. 验证导入的表仅包含两个正常行：

```
$ mysql -u root -h 127.0.0.1 -P 4000 -e 'select * from example.t'
+---+---------+
| a | b    |
+---+---------+
| 1 | one |
| 2 | two |
+---+---------+
```

6. 检查 `type_error_v1` 表是否捕获了涉及类型转换的三行：

```
$ mysql -u root -h 127.0.0.1 -P 4000 -e 'select * from lightning_task_info.type_error_v1;' -
    '→ E
*************************** 1. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.620090
 table_name: `example`.`t`
       path: example.t.1.sql
     offset: 46
      error: failed to cast value as varchar(12) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin
          '→ for column `b` (#2): [table:1048]Column 'b' cannot be null
   row_data: (0,NULL)
*************************** 2. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.627496
 table_name: `example`.`t`
       path: example.t.1.sql
     offset: 183
      error: failed to cast value as varchar(12) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin
          '→ for column `b` (#2): [types:1406]Data Too Long, field len 12, data len 13
   row_data: (77,'seventy-seven')
*************************** 3. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.629929
 table_name: `example`.`t`
       path: example.t.1.sql
     offset: 253
      error: failed to cast value as tinyint(4) for column `a` (#1): [types:1690]constant
          '→ 600 overflows tinyint
   row_data: (600,'six hundred')
```

7. 检查 `conflict_error_v1` 表是否捕获了具有唯一键/主键冲突的四行：

```
$ mysql -u root -h 127.0.0.1 -P 4000 -e 'select * from lightning_task_info.conflict_error_v1
    '→ ;' --binary-as-hex -E
*************************** 1. row ***************************
```

```
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.669601
 table_name: `example`.`t`
 index_name: PRIMARY
   key_data: 40
   row_data: (40, "forty")
    raw_key: 0x7480000000000000C15F728000000000000028
 raw_value: 0x800001000000020500666F727479 raw_handle:
 0x7480000000000000C15F728000000000000028
    raw_row: 0x800001000000020500666F727479
*************************** 2. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.674798
 table_name: `example`.`t`
 index_name: PRIMARY
   key_data: 40
   row_data: (40, "fourty")
    raw_key: 0x7480000000000000C15F728000000000000028
 raw_value: 0x800001000000020600666F75727479
 raw_handle: 0x7480000000000000C15F728000000000000028
    raw_row: 0x800001000000020600666F75727479
*************************** 3. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.680332
 table_name: `example`.`t`
 index_name: b
   key_data: 54
   row_data: (54, "fifty-four")
    raw_key: 0x7480000000000000C15F698000000000000001016669667479 2D666FFF7572000000000000F9
  raw_value: 0x0000000000000036
 raw_handle: 0x7480000000000000C15F728000000000000036
    raw_row: 0x800001000000020A0066696674792D666F7572
*************************** 4. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.681073
 table_name: `example`.`t`
 index_name: b
   key_data: 42
   row_data: (42, "fifty-four")
    raw_key: 0x7480000000000000C15F698000000000000001016669667479 2D666FFF7572000000000000F9
  raw_value: 0x000000000000002A
 raw_handle: 0x7480000000000000C15F72800000000000002A
    raw_row: 0x800001000000020A0066696674792D666F7572
```

10.7.13 TiDB Lightning 故障处理

本文档总结了使用TiDB Lightning 过程中常见的运行故障及解决方案。

10.7.13.1 TiDB Lightning 导入速度太慢

TiDB Lightning 的正常速度为每条线程每 2 分钟导入一个 256 MB 的数据文件，如果速度远慢于这个数值就是有问题。导入的速度可以检查日志提及 `restore chunk ... takes` 的记录，或者观察Grafana 的监控信息。

导入速度太慢一般有几个原因：

原因 1：`region-concurrency` 设定太高，线程间争用资源反而减低了效率。

1. 从日志的开头搜寻 `region-concurrency` 能知道TiDB Lightning 读到的参数是多少。
2. 如果TiDB Lightning 与其他服务（如 TiKV Importer）共用一台服务器，必需手动将 `region-concurrency` 设为该服务器 CPU 数量的 75%。
3. 如果 CPU 设有限额（例如从 Kubernetes 指定的上限），TiDB Lightning 可能无法自动判断出来，此时亦需要手动调整 `region-concurrency`。

原因 2：表结构太复杂。

每条索引都会额外增加键值对。如果有N 条索引，实际导入的大小就差不多是 Dumpling 文件的 N+1 倍。如果索引不太重要，可以考虑先从 schema 去掉，待导入完成后再使用 `CREATE INDEX` 加回去。

原因 3：单个文件过大。

把源数据分割为单个大小约为 256 MB 的多个文件时，TiDB Lightning 会并行处理数据，达到最佳效果。如果导入的单个文件过大，TiDB Lightning 可能无响应。

如果源数据是CSV 格式文件，并且所有的CSV 文件内都不存在包含字符换行符的字段(U+000A 及 U+000D)，则可以启用 `strict-format`，TiDB Lightning 会自动分割大文件。

```
[mydumper]
strict-format = true
```

原因 4：TiDB Lightning 版本太旧。

建议试试最新的版本，可能会有改善。

10.7.13.2 `tidb-lightning` 进程意外退出

这种情况可能是启动方式不正确，导致收到 SIGHUP 信号而退出。此时 `tidb-lightning.log` 通常有如下日志：

```
[2018/08/10 07:29:08.310 +08:00] [INFO] [main.go:41] ["got signal to exit"] [signal=hangup]
```

不推荐在命令行中直接使用 nohup 启动进程，推荐使用脚本启动 `tidb-lightning`。

另外，如果从TiDB Lightning 的 log 的最后一条日志显示遇到的错误是“Context canceled”，需要在日志中搜索第一条“ERROR”级别的日志。在这条日志之前，通常也会紧跟有一条“got signal to exit”，表示 Lightning 是收到中断信号然后退出的。

10.7.13.3 使用TiDB Lightning后，TiDB集群变慢，CPU占用高

如果 `tidb-lightning` 异常退出，集群可能仍处于“导入模式”(import mode)，该模式不适用于生产环境。此时可执行以下命令查看当前使用的模式：

```
tidb-lightning-ctl --config tidb-lightning.toml --fetch-mode
```

可执行以下命令强制切换回“普通模式”(normal mode)：

```
tidb-lightning-ctl --config tidb-lightning.toml --switch-mode=normal
```

10.7.13.4 TiDB Lightning报错

10.7.13.4.1 `could not find first pair, this shouldn't happen`

报错原因是遍历本地排序的文件时出现异常，可能在TiDB Lightning打开的文件数量超过系统的上限时发生报错。在Linux系统中，可以使用 `ulimit -n` 命令确认此值是否过小。建议在导入期间将此设置调整为1000000（即 `ulimit -n 1000000`）。

10.7.13.4.2 `checksum failed: checksum mismatched remote vs local`

原因：本地数据源跟目标数据库某个表的校验和不一致。这通常有更深层的原因，可以通过检查日志中包含 `checksum mismatched` 的行进一步定位。

包含 `checksum mismatched` 的行中有 `total_kvs: x vs y` 的信息，x表示导入集群在完成导入后计算出的键值对（KV pairs）数目，y表示本地数据源产生的键值对数目。

- x大，即导入集群键值对更多：
  - 可能这张表在导入前已有数据，因此影响了数据校验。这也包括TiDB Lightning之前失败停机过，但没有正确重启。
- y大，即本地数据源键值对更多：
  - 如果目标数据库的校验和全是0，表示没有发生任何导入，有可能是集群太忙无法接收任何数据。
  - 可能导出数据中包含重复数据，例如唯一键和主键(UNIQUE and PRIMARY KEYs)有重复的值、下游表结构为大小写不敏感而数据为大小写敏感。
- 其他情况
  - 如果数据源是由机器生成而不是从Dumpling备份的，需确保数据符合表的限制，例如自增(AUTO_INCREMENT)的列需要为正数，不能为0。

解决办法：

1. 使用 `tidb-lightning-ctl` 把出错的表删除，检查表结构与数据，重启TiDB Lightning重新导入此前出错的表。

   ```
   tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
   ```

2. 把断点存放在外部数据库（修改 `[checkpoint] dsn`），减轻目标集群压力。
3. 参考如何正确重启TiDB Lightning中的解决办法。

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

原因：断点续传已启用。TiDB Lightning 或 TiKV Importer 之前发生了异常退出。为了防止数据意外损坏，TiDB Lightning 在错误解决以前不会启动。

错误码是小于 25 的整数，可能的取值是 0、3、6、9、12、14、15、17、18、20、21。整数越大，表示异常退出所发生的步骤在导入流程中越晚。

解决办法：

如果错误原因是非法数据源，使用 `tidb-lightning-ctl` 删除已导入数据，并重启TiDB Lightning。

```
tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
```

其他解决方法请参考断点续传的控制。

10.7.13.4.4 ResourceTemporarilyUnavailable("Too many open engines …: …")

原因：并行打开的引擎文件(engine files) 超出 `tikv-importer` 里的限制。这可能由配置错误引起。即使配置没问题，如果 `tidb-lightning` 曾经异常退出，也有可能令引擎文件残留在打开的状态，占据可用的数量。

解决办法：

1. 提高 `tikv-importer.toml` 内 `max-open-engines` 的值。这个设置主要由内存决定，计算公式为：

   最大内存使用量≈ `max-open-engines` × `write-buffer-size` × `max-write-buffer-number`

2. 降低 `table-concurrency` + `index-concurrency`，使之低于 `max-open-engines`。

3. 重启 `tikv-importer` 来强制移除所有引擎文件(默认值为 `./data.import/`)。这样也会丢弃导入了一半的表，所以启动TiDB Lightning 前必须清除过期的断点记录：

   ```
   tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
   ```

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

原因：TiDB Lightning 只支持 UTF-8 和 GB-18030 编码的表架构。此错误代表数据源不是这里任一个编码。也有可能是文件中混合了不同的编码，例如，因为在不同的环境运行过 `ALTER TABLE`，使表架构同时出现 UTF-8 和 GB-18030 的字符。

解决办法：

1. 编辑数据源，保存为纯 UTF-8 或 GB-18030 的文件。
2. 手动在目标数据库创建所有的表。
3. 设置 `[mydumper] character-set = "binary"` 跳过这个检查。但是这样可能使数据库出现乱码。

10.7.13.4.6 [sql2kv] sql encode error = [types:1292]invalid time format: '{1970 1 1 …}'

原因: 一个 `timestamp` 类型的时间戳记录了不存在的时间值。时间值不存在是由于夏时制切换或超出支持的范围（1970 年 1 月 1 日至 2038 年 1 月 19 日）。

解决办法:

1. 确保TiDB Lightning 与数据源时区一致。

    - 手动部署的话，通过设定 $TZ 环境变量强制时区设定。
      强制使用Asia/Shanghai 时区：

    ```
    TZ='Asia/Shanghai' bin/tidb-lightning -config tidb-lightning.toml
    ```

2. 导出数据时，必须加上 `--skip-tz-utc` 选项。

3. 确保整个集群使用的是同一最新版本的 `tzdata` (2018i 或更高版本)。

    如果你使用的是 CentOS 机器，你可以运行 `yum info tzdata` 命令查看 `tzdata` 的版本及是否有更新。然后运行 `yum upgrade tzdata` 命令升级 `tzdata`。

10.7.13.4.7 `[Error 8025: entry too large, the max entry size is 6291456]`

原因：TiDB Lightning 生成的单行 KV 超过了TiDB 的限制。

解决办法:

目前无法绕过TiDB 的限制，只能忽略这张表，确保其它表顺利导入。

10.7.13.4.8 switch-mode 时遇到 `rpc error: code = Unimplemented ...`

原因：集群中有不支持 switch-mode 的节点。目前已知的组件中，4.0.0-rc.2 之前的TiFlash 不支持 switch-mode 操作。

解决办法：

- 如果集群中有TiFlash 节点，可以将集群更新到 4.0.0-rc.2 或更新版本。
- 如果不方便升级，可以临时禁用TiFlash。

10.7.13.4.9 `tidb lightning encountered error: TiDB version too old, expected '>=4.0.0', found '3.0.18'`

TiDB Lightning Local-backend 只支持导入到 v4.0.0 及以上版本的TiDB 集群。如果尝试使用 Local-backend 导入到 v2.x 或 v3.x 的集群，就会报以上错误。此时可以修改配置使用Importer-backend 或TiDB-backend 进行导入。

部分 `nightly` 版本的TiDB 集群的版本可能类似 4.0.0-beta.2。这种版本的 TiDB Lightning 实际支持 Local-backend，如果使用 `nightly` 版本遇到该报错，可以通过设置配置 `check-requirements = false` 跳过版本检查。在设置此参数之前，请确保TiDB Lightning 的配置支持对应的版本，否则无法保证导入成功。

10.7.13.4.10 `restore table test.district failed: unknown columns in header [...]`

出现该错误通常是因为CSV 格式的数据文件不包含header（第一行也是数据），因此需要在 TiDB Lightning 的配置文件中增加如下配置项：

```
[mydumper.csv]
header = false
```

10.7.13.4.11 Unknown character set

由于TiDB 只支持部分 MySQL 字符集，因此，在导入流程中，如果创建表结构时使用了TiDB 不支持的字符集，TiDB Lightning 会报这个错误。你可以结合数据内容选择TiDB 支持的字符集，预先在下游创建表结构以绕过这个错误。

10.7.13.4.12 invalid compression type ...

TiDB v6.4.0 及之后版本的TiDB Lightning 不支持带有非 .bak 后缀的数据文件并报错。你需要提前修改文件名，或将该类文件移出导入数据目录来避免此类错误。更多详情请参考压缩导出的数据文件。

10.7.13.5 TiDB Lightning 配置参数

你可以使用配置文件或命令行配置TiDB Lightning。本文主要介绍 TiDB Lightning 的全局配置、任务配置，以及如何使用命令行进行参数配置。

10.7.13.5.1 配置文件

TiDB Lightning 的配置文件分为“全局”和“任务”两种类别，二者在结构上兼容。只有当服务器模式开启时，全局配置和任务配置才会有区别；默认情况下，服务器模式为禁用状态，此时TiDB Lightning 只会执行一个任务，且全局和任务配置使用同一配置文件。

TiDB Lightning 全局配置

```
###### tidb-lightning 全局配置

[lightning]
#### 用于进度展示 web 界面、拉取 Prometheus 监控项、暴露调试数据和提交导入任务（服务器模式下）的
    ↪ HTTP 端口。设置为 0 时为禁用状态。
status-addr = ':8289'

#### 服务器模式，默认值为 false，命令启动后会开始导入任务。如果改为 true，命令启动后会等待用户在
    ↪ web 界面上提交任务。
#### 详情参见“TiDB Lightning web 界面”文档
server-mode = false

#### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14
```

TiDB Lightning 任务配置

```
###### tidb-lightning 任务配置

[lightning]
```

```
#### 启动之前检查集群是否满足最低需求。
#### check-requirements = true

#### 引擎文件的最大并行数。
#### 每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”。
#### 这两项设置控制两种引擎文件的最大并发数。
#### 这两项设置的值会影响 tikv-importer 的内存和磁盘用量。
#### 两项数值之和不能超过 tikv-importer 的 max-open-engines 的设定。
index-concurrency = 2
table-concurrency = 6

#### 数据的并发数。默认与逻辑 CPU 的数量相同。
#### 混合部署的情况下可以将其大小配置为逻辑 CPU 数的 75%，以限制 CPU 的使用。
#### region-concurrency =

#### I/O 最大并发数。I/O 并发量太高时，会因硬盘内部缓存频繁被刷新
#### 而增加 I/O 等待时间，导致缓存未命中和读取速度降低。
#### 对于不同的存储介质，此参数可能需要调整以达到最佳效率。
io-concurrency = 5

#### TiDB Lightning 停止迁移任务之前能容忍的最大非严重 (non-fatal errors) 错误数。
#### 在忽略非严重错误所在的行数据之后，迁移任务可以继续执行。
#### 将该值设置为 N，表示 TiDB Lightning 会在遇到第 (N+1) 个错误时停止迁移任务。
#### 被忽略的行会被记录到位于目标集群的 "task info" 数据库中。最大错误数可以通过下面参数配置。
#### 默认值为 MaxInt64 字节，即 9223372036854775807 字节。
max-error = 0
#### 参数 task-info-schema-name 指定用于存储 TiDB Lightning 执行结果的数据库。
#### 要关闭该功能，需要将该值设置为空字符串。
#### task-info-schema-name = 'lightning_task_info'

#### 在并行导入模式下，在目标集群保存各个 TiDB Lightning 实例元信息的 schema 名字，默认为 "
    ′→ lightning_metadata"。
#### 如果未开启并行导入模式，无须设置此配置项。
#### **注意：**
#### - 对于参与同一批并行导入的每个 TiDB Lightning 实例，该参数设置的值必须相同，
    ′→ 否则将无法确保导入数据的正确性。
#### - 如果开启并行导入模式，需要确保导入使用的用户（对于 tidb.user 配置项）
    ′→ 有权限创建和访问此配置对应的库。
#### - TiDB Lightning 在导入完成后会删除此 schema，因此不要使用已存在的库名配置该参数。
meta-schema-name = "lightning_metadata"

[security]
#### 指定集群中用于 TLS 连接的证书和密钥。
#### CA 的公钥证书。如果留空，则禁用 TLS。
#### ca-path = "/path/to/ca.pem"
```

```
#### 此服务的公钥证书。
#### cert-path = "/path/to/lightning.pem"
#### 该服务的密钥。
#### key-path = "/path/to/lightning.key"

[checkpoint]
#### 是否启用断点续传。
#### 导入数据时，TiDB Lightning 会记录当前表导入的进度。
#### 所以即使 TiDB Lightning 或其他组件异常退出，在重启时也可以避免重复再导入已完成的数据。
enable = true
#### 存储断点的数据库名称。
schema = "tidb_lightning_checkpoint"
#### 存储断点的方式。
#### - file：存放在本地文件系统。
#### - mysql：存放在兼容 MySQL 的数据库服务器。
driver = "file"

#### dsn 是数据源名称 (data source name)，表示断点的存放位置。
#### 若 driver = "file"，则 dsn 为断点信息存放的文件路径。
#若不设置该路径，则默认存储路径为“/tmp/CHECKPOINT_SCHEMA.pb”。
#### 若 driver = "mysql"，则 dsn 为“用户:密码@tcp(地址:端口)/”格式的 URL。
#### 若不设置该 URL，则默认会使用 [tidb] 部分指定的 TiDB 服务器来存储断点。
#### 为减少目标 TiDB 集群的压力，建议指定另一台兼容 MySQL 的数据库服务器来存储断点。
#### dsn = "/tmp/tidb_lightning_checkpoint.pb"

#### 所有数据导入成功后是否保留断点。设置为 false 时为删除断点。
#### 保留断点有利于进行调试，但会泄漏关于数据源的元数据。
#### keep-after-success = false

[tikv-importer]
#### "local"：Physical Import Mode，默认使用。适用于 TB 级以上大数据量，但导入期间下游 TiDB
    ↪ 无法对外提供服务。
#### "tidb"：Logical Import Mode。TB 级以下数据量可以采用，下游 TiDB 可正常提供服务。
#### backend = "local"
#### 是否允许启动多个 TiDB Lightning 实例（**physical import mode**）并行导入到一个或多个目标表。
    ↪ 默认取值为 false。注意，这个参数**不是用于增量导入数据**，仅限目标表为空的场景使用。
#### 多个 TiDB Lightning 实例（physical import mode）同时导入一张表时，此开关必须设置为 true。
    ↪ 但前提是目标表不能存在数据，即所有的数据都只能是由 TiDB Lightning 导入。
#### incremental-import = false
#### 当后端是 “importer” 时，tikv-importer 的监听地址（需改为实际地址）。
addr = "172.16.31.10:8287"
#### Logical Import Mode 插入冲突数据时执行的操作。关于冲突检测详细信息请查阅：https://docs.
    ↪ pingcap.com/zh/tidb/dev/tidb-lightning-logical-import-mode-usage#冲突数据检测
#### - replace：新数据替代已有数据
#### - ignore：保留已有数据，忽略新数据
```

```
### - error：中止导入并报错
### on-duplicate = "replace"

### Physical Import Mode 设置是否检测和解决重复的记录（唯一键冲突）。
### 目前支持三种解决方法：
### - record：数据写入目标表后，将目标表中重复记录添加到目标 TiDB 中的 `lightning_task_info.
    ↪ conflict_error_v1` 表中。注意，该方法要求目标 TiKV 的版本为 v5.2.0 或更新版本。
    ↪ 如果版本过低，则会启用下面的 'none' 模式。
### - none：不检测重复记录。该模式是三种模式中性能最佳的，但是如果数据源存在重复记录，会导致
    ↪ TiDB 中出现数据不一致的情况。
### - remove：记录所有目标表中的重复记录，和 'record' 模式相似。但是会删除目标表所有的重复记录
    ↪ ，以确保目标 TiDB 中的数据状态保持一致。
### duplicate-resolution = 'none'
### Physical Import Mode 一次请求中发送的 KV 数量。
### send-kv-pairs = 32768
### Physical Import Mode 向 TiKV 发送 KV 时是否启用压缩。目前只支持 Gzip 压缩算法，可填写 "gzip"
    ↪ 或者 "gz"。默认不启用压缩。
### compress-kv-pairs = ""
### Physical Import Mode 本地进行 KV 排序的路径。如果磁盘性能较低（如使用机械盘），建议设置成与
    ↪ `data-source-dir` 不同的磁盘，这样可有效提升导入性能。
### sorted-kv-dir = ""
### Physical Import Mode TiKV 写入 KV 数据的并发度。当 TiDB Lightning 和 TiKV
    ↪ 直接网络传输速度超过万兆的时候，可以适当增加这个值。
### range-concurrency = 16
### Physical Import Mode 限制 TiDB Lightning 向每个 TiKV 节点写入的带宽大小，默认为 0，
    ↪ 表示不限制。
### store-write-bwlimit = "128MiB"

### 使用 Physical Import Mode 时，配置 TiDB Lightning 本地临时文件使用的磁盘配额 (disk quota)。
    ↪ 当磁盘配额不足时，TiDB Lightning 会暂停读取源数据以及写入临时文件的过程，
    ↪ 优先将已经完成排序的 key-value 写入到 TiKV，TiDB Lightning 删除本地临时文件后，
    ↪ 再继续导入过程。
### 需要同时配合把 `backend` 设置为 `local` 模式才能生效。
### 默认值为 MaxInt64 字节，即 9223372036854775807 字节。
### disk-quota = "10GB"

### Physical Import Mode 是否通过 SQL 方式添加索引。默认为 `false`，表示 TiDB Lightning
    ↪ 会将行数据以及索引数据都编码成 KV pairs 后一同导入 TiKV，实现机制和历史版本保持一致。
    ↪ 如果设置为 `true`，即 TiDB Lightning 会在导入数据完成后，使用 add index 的 SQL 来添加索引
    ↪ 。
### 通过 SQL 方式添加索引的优点是将导入数据与导入索引分开，可以快速导入数据，即使导入数据后，
    ↪ 索引添加失败，也不会影响数据的一致性。
### add-index-by-sql = false

### 在使用 TiDB Lightning 导入多租户的 TiDB cluster 的场景下，指定对应的 key space 名称。
```

```
    ↪ 默认取值为空字符串，表示 TiDB Lightning 会自动获取导入对应租户的 key space 名称；
    ↪ 如果指定了值，则使用指定的 key space 名称来导入。
#### keyspace-name = ""

#### Physical Import Mode 下，用于控制 TiDB Lightning 暂停 PD 调度的范围，可选值包括：
#### - "table"：仅暂停目标表数据所在 Region 的调度。默认值为 "table"。
#### - "global"：暂停全局调度。当导入数据到无业务流量的集群时，建议设置为 "global"，
    ↪ 以避免其他调度的干扰。
#### 该参数自 v7.1.0 版本开始引入。注意："table" 选项仅适用于 TiDB v6.1.0 及以上版本的目标集群。
#### pause-pd-scheduler-scope = "table"

#### Physical Import Mode 下，用于控制批量 Split Region 时的 Region 个数。每个 TiDB Lightning
    ↪ 实例最多同时 Split Region 的个数为：
#### region-split-batch-size * region-split-concurrency * table-concurrency
#### 该参数自 v7.1.0 版本开始引入，默认值为 `4096`。
#### region-split-batch-size = 4096

#### Physical Import Mode 下，用于控制 Split Region 时的并发度。默认值为 CPU 核数。
#### 该参数自 v7.1.0 版本开始引入。
#### region-split-concurrency =

#### Physical Import Mode 下，用于控制 split 和 scatter 操作后等待 Region 上线的重试次数，
    ↪ 默认值为 `1800`。重试符合指数回退策略，最大重试间隔为 2 秒。若两次重试之间有任何 Region
    ↪ 上线，该次操作不会被计为重试次数。
#### 该参数自 v7.1.0 版本开始引入。
#### region-check-backoff-limit = 1800

[mydumper]
#### 设置文件读取的区块大小，确保该值比数据源的最长字符串长。
read-block-size = "64KiB" # 默认值

#### 引擎文件需按顺序导入。由于并行处理，多个数据引擎几乎在同时被导入，
#### 这样形成的处理队列会造成资源浪费。因此，为了合理分配资源，TiDB Lightning
#### 稍微增大了前几个区块的大小。该参数也决定了比例系数，即在完全并发下
#### “导入”和“写入”过程的持续时间比。这个值可以通过计算 1 GiB 大小的
#### 单张表的（导入时长/写入时长）得到。在日志文件中可以看到精确的时间。
#### 如果“导入”更快，区块大小的差异就会更小；比值为 0 时则说明区块大小一致。
#### 取值范围为（0 <= batch-import-ratio < 1）。
batch-import-ratio = 0.75

#### 本地源数据目录或外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/tidb/
    ↪ v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"

#### 指定包含 `CREATE TABLE` 语句的表结构文件的字符集。只支持下列选项：
```

```
#### - utf8mb4：表结构文件必须使用 UTF-8 编码，否则会报错。
#### - gb18030：表结构文件必须使用 GB-18030 编码，否则会报错。
#### - auto：自动判断文件编码是 UTF-8 还是 GB-18030，两者皆非则会报错（默认）。
#### - binary：不尝试转换编码。
character-set = "auto"

#### 指定源数据文件的字符集，Lightning 会在导入过程中将源文件从指定的字符集转换为 UTF-8 编码。
#### 该配置项目前仅用于指定 CSV 文件的字符集。只支持下列选项：
#### - utf8mb4：源数据文件使用 UTF-8 编码。
#### - GB18030：源数据文件使用 GB-18030 编码。
#### - GBK：源数据文件使用 GBK 编码（GBK 编码是对 GB-2312 字符集的拓展，也被称为 Code Page
    ↪ 936）。
#### - binary：不尝试转换编码（默认）。
#### 留空此配置将默认使用 "binary"，即不尝试转换编码。
#### 需要注意的是，Lightning 不会对源数据文件的字符集做假定，仅会根据此配置对数据进行转码并导入。
#### 如果字符集设置与源数据文件的实际编码不符，可能会导致导入失败、导入缺失或导入数据乱码。
data-character-set = "binary"
#### 指定在源数据文件的字符集转换过程中，出现不兼容字符时的替换字符。
#### 此项不可与字段分隔符、引用界定符和换行符号重复。
#### 默认值为 "\uFFFD"，即 UTF-8 编码中的 "error" Rune 或 Unicode replacement character。
#### 改变默认值可能会导致潜在的源数据文件解析性能下降。
data-invalid-char-replace = "\uFFFD"

#### “严格”格式的导入数据可加快处理速度。
#### strict-format = true 要求：
#### 在 CSV 文件的所有记录中，每条数据记录的值不可包含字符换行符（U+000A 和 U+000D，即 \r 和 \n）
#### 甚至被引号包裹的字符换行符都不可包含，即换行符只可用来分隔行。
#### 导入数据源为严格格式时，TiDB Lightning 会快速定位大文件的分割位置进行并行处理。
#### 但是如果输入数据为非严格格式，可能会将一条完整的数据分割成两部分，导致结果出错。
#### 为保证数据安全而非追求处理速度，默认值为 false。
strict-format = false

#### 如果 strict-format = true，TiDB Lightning 会将 CSV 大文件分割为多个文件块进行并行处理。max-
    ↪ region-size 是分割后每个文件块的最大大小。
#### max-region-size = "256MiB" # 默认值

#### 只导入与该通配符规则相匹配的表。详情见相应章节。
filter = ['*.*', '!mysql.*', '!sys.*', '!INFORMATION_SCHEMA.*', '!PERFORMANCE_SCHEMA.*', '!
    ↪ METRICS_SCHEMA.*', '!INSPECTION_SCHEMA.*']

#### 配置 CSV 文件的解析方式。
[mydumper.csv]
#### 字段分隔符，支持一个或多个字符，默认值为 ','。
separator = ','
#### 引用定界符，设置为空表示字符串未加引号。
```

```
delimiter = '"'
### 行尾定界字符，支持一个或多个字符。设置为空（默认值）表示 "\n"（换行）和 "\r\n"（回车+换行
    ↪ ），均表示行尾。
terminator = ""
### CSV 文件是否包含表头。
### 如果 header = true，将把首行的内容作为表头处理，不作为数据导入。如果设置为 false，首行也作为
    ↪ CSV 数据导入，此时请确保 CSV 文件的列顺序与目标表的列顺序一致，否则可能会导致数据差异。
header = true
### CSV 表头是否匹配目标表的表结构。
### 默认为 true，表示在导入数据时，会根据 CSV 表头的字段名去匹配目标表对应的列名，这样即使 CSV
    ↪ 文件和目标表列的顺序不一致也能按照对应的列名进行导入。
### 如果 CSV 表头中的字段名和目标表的列名不匹配（例如，CSV
    ↪ 表头中的某些字段名在目标表中可能找不到对应的同名列）但列的顺序是一致的，请将该配置设置为
    ↪ false。
### 这时，在导入的时候，会直接忽略 CSV 表头的内容，以避免导入错误。在这种情况下，直接把 CSV
    ↪ 数据按照目标表列的顺序导入。
#### 因此，如果列的顺序不一致，请手动调整一致后再导入，否则可能会导致数据差异。
#### 注意：只有在 header = true 时，该参数才会生效。如果 header = false ，表示 CSV 文件没有表头，
    ↪ 此时不需要考虑相关列名匹配的问题。
header-schema-match = true
### CSV 文件是否包含 NULL。
### 如果 not-null = true，CSV 所有列都不能解析为 NULL。
not-null = false
### 如果 not-null = false（即 CSV 可以包含 NULL），
### 为以下值的字段将会被解析为 NULL。
null = '\N'
### 是否对字段内“\“进行转义
backslash-escape = true
### 如果有行以分隔符结尾，删除尾部分隔符。
trim-last-separator = false

#### [[mydumper.files]]
### 解析 AWS Aurora parquet 文件所需的表达式
#### pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet)
    ↪ )$'
#### schema = '$1'
#### table = '$2'
#### type = '$3'

[tidb]
### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
### 设置连接 TiDB 的密码，可为明文或 Base64 编码。
```

```
password = ""
### 表结构信息从 TiDB 的“status-port”获取。
status-port = 10080
### pd-server 的地址，填一个即可。
pd-addr = "172.16.31.4:2379"
### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
### 设置 TiDB 库的日志等级。
log-level = "error"

### 设置 TiDB 会话变量，提升 Checksum 和 Analyze 的速度。
### 各参数定义可参阅”控制 Analyze 并发度“文档
build-stats-concurrency = 20
distsql-scan-concurrency = 15
index-serial-scan-concurrency = 20
checksum-table-concurrency = 2

### 解析和执行 SQL 语句的默认 SQL 模式。
sql-mode = "ONLY_FULL_GROUP_BY,NO_ENGINE_SUBSTITUTION"
### `max-allowed-packet` 设置数据库连接允许的最大数据包大小，
### 对应于系统参数中的 `max_allowed_packet`。 如果设置为 0，
### 会使用下游数据库 global 级别的 `max_allowed_packet`。
max-allowed-packet = 67_108_864

### SQL 连接是否使用 TLS。可选值为：
####  * ""            - 如果填充了 [tidb.security] 部分，则强制使用 TLS（与 "cluster" 情况相同
    ↪ ），否则与 "false" 情况相同
####  * "false"       - 禁用 TLS
####  * "cluster"     - 强制使用 TLS 并使用 [tidb.security] 部分中指定的 CA 验证服务器的证书
#### * "skip-verify" - 强制使用 TLS，但不验证服务器的证书（不安全！）
####  * "preferred"   - 与 "skip-verify" 相同，但是如果服务器不支持 TLS，则会退回到未加密的连接
#### tls = ""
### 指定证书和密钥用于 TLS 连接 MySQL。
### 默认为 [security] 部分的副本。
#### [tidb.security]
### CA 的公钥证书。设置为空字符串可禁用 SQL 的 TLS。
#### ca-path = "/path/to/ca.pem"
### 该服务的公钥证书。默认为 `security.cert-path` 的副本
#### cert-path = "/path/to/lightning.pem"
### 此服务的私钥。默认为 `security.key-path` 的副本
#### key-path = "/path/to/lightning.key"

### 对于 Physical Import Mode，数据导入完成后，TiDB Lightning 可以自动执行 Checksum 和 Analyze
    ↪ 操作。
### 在生产环境中，建议总是开启 Checksum 和 Analyze。
### 执行的顺序为：Checksum -> Analyze。
```

```
### 注意：对于 Logical Import Mode，无须执行这两个阶段，因此在实际运行时总是会直接跳过。
[post-restore]
### 配置是否在导入完成后对每一个表执行 `ADMIN CHECKSUM TABLE <table>` 操作来验证数据的完整性。
### 可选的配置项：
### - "required"（默认）。在导入完成后执行 CHECKSUM 检查，如果 CHECKSUM 检查失败，则会报错退出。
### - "optional"。在导入完成后执行 CHECKSUM 检查，如果报错，会输出一条 WARN 日志并忽略错误。
### - "off"。导入结束后不执行 CHECKSUM 检查。
### 默认值为 "required"。从 v4.0.8 开始，checksum 的默认值由此前的 "true" 改为 "required"。
### # 注意：
### 1. Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
### 2. 考虑到与旧版本的兼容性，依然可以在本配置项设置 `true` 和 `false` 两个布尔值，其效果与 `
    ↪ required` 和 `off` 相同。
checksum = "required"
### 配置是否在 CHECKSUM 结束后对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
### 此配置的可选配置项与 `checksum` 相同，但默认值为 "optional"。
analyze = "optional"


### 设置周期性后台操作。
### 支持的单位：h（时）、m（分）、s（秒）。
[cron]
### TiDB Lightning 自动刷新导入模式状态的持续时间，该值应小于 TiKV 对应的设定值。
switch-mode = "5m"
### 在日志中打印导入进度的持续时间。
log-progress = "5m"

### 使用 Physical Import Mode 时，检查本地磁盘配额的时间间隔，默认为 60 秒。
### check-disk-quota = "60s"
```

10.7.13.5.2 命令行参数

tidb-lightning

使用 tidb-lightning 可以对下列参数进行配置：

参数	描述	对应配置项
–config file	从 file 读取全局设置。如果没有指定则使用默认设置。	
-V	输出程序的版本	
-d directory	读取数据的本地目录或外部存储 URI	mydumper '→ . '→ data '→ - '→ source '→ - '→ dir
-L level	日志的等级：debug、info、warn、error 或 fatal (默认为 info)	lightning '→ . '→ log '→ - '→ level '→
-f rule	表库过滤的规则 (可多次指定)	mydumper '→ . '→ filter '→

参数	描述	对应配置项
–backend backend	选择导入的模式：local ′→ 为 Physical Import Mode，tidb为 Logical Import Mode	local
–log-file file	日志文件路径（默认值为 /tmp/ ′→ lightning ′→ . ′→ log ′→ .{ ′→ timestamp ′→ }，设置为 ‘-’ 表示日志输出到终端）	lightning ′→ . ′→ log ′→ - ′→ file ′→
–status-addr ip:port	TiDB Light-ning 服务器的监听地址	lightning ′→ . ′→ status ′→ - ′→ port ′→
–importer host:port	TiKV Importer 的地址	tikv- ′→ importer ′→ . ′→ addr ′→

参数	描述	对应配置项
–pd-urls host:port	PD endpoint 的地址	tidb. ′→ pd- ′→ addr ′→
–tidb-host host	TiDB Server 的 host	tidb. ′→ host ′→
–tidb-port port	TiDB Server 的端口（默认为 4000）	tidb. ′→ port ′→
–tidb-status port	TiDB Server 的状态端口的（默认为 10080）	tidb. ′→ status ′→ - ′→ port ′→
–tidb-user user	连接到 TiDB 的用户名	tidb. ′→ user ′→
–tidb-password pass-word	连接到 TiDB 的密码，可为明文或 Base64 编码	tidb. ′→ password ′→
–enable-checkpoint bool	是否启用断点 (默认值为 true)	checkpoint ′→ . ′→ enable ′→

参数	描述	对应配置项
–analyze level	导入后分析表信息，可选值为 required、optional（默认值）、off	post-restore.analyze
–checksum level	导入后比较校验和，可选值为 required（默认值）、optional、off	post-restore.checksum
–check-requirement bool	开始之前检查集群版本兼容性（默认值为 true）	lightning.check-requirements
–ca file	TLS 连接的 CA 证书路径	security.ca-path
–cert file	TLS 连接的证书路径	security.cert-path

参数	描述	对应配置项
–key file	TLS 连接的私钥路径	`security ′→ . ′→ key ′→ - ′→ path ′→`
–server-mode	在服务器模式下启动 TiDB Light-ning	`lightning ′→ . ′→ server ′→ - ′→ mode ′→`

如果同时对命令行参数和配置文件中的对应参数进行更改，命令行参数将优先生效。例如，在 `cfg.toml` 文件中，不管对日志等级做出什么修改，运行 `./tidb-lightning -L debug --config cfg.toml` 命令总是将日志级别设置为“debug”。

`tidb-lightning-ctl`

使用 `tidb-lightning-ctl` 可以对下列参数进行配置：

参数	描述
–compact	执行 full compact
–switch-mode mode	将每个TiKV Store 切换到指定模式（ normal 或 import ）
–fetch-mode	打印每个TiKV Store 的当前模式
–import-engine uuid	将 TiKV Importer 上关闭的引擎文件导入到TiKV 集群
–cleanup-engine uuid	删除TiKV Importer 上的引擎文件
–checkpoint-dump folder	将当前的断点以CSV 格式存储到文件夹中
–checkpoint-error-destroy tablename	删除断点，如果报错则删除该表
–checkpoint-error-ignore tablename	忽略指定表中断点的报错
–checkpoint-remove tablename	无条件删除表的断点

tablename 必须是`db`.`tbl` 中的限定表名（包括反引号），或关键词 `all`。

此外，上表中所有 `tidb-lightning` 的参数也适用于 `tidb-lightning-ctl`。

10.7.13.6 TiDB Lightning 命令行参数

你可以使用配置文件或命令行配置TiDB Lightning。本文主要介绍 TiDB Lightning 的命令行参数。

10.7.13.6.1 命令行参数

`tidb-lightning`

使用 `tidb-lightning` 可以对下列参数进行配置：

参数	描述	对应配置项
–config file	从 file 读取全局设置。如果没有指定则使用默认设置。	
-V	输出程序的版本	

-d directory	读取数据的本地目录或外部存储 URI	`mydumper`
		 `'→ .`
		 `'→ data`
		 `'→ -`
		 `'→ source`
		 `'→ -`
		 `'→ dir`

-L level	日志的等级：debug、info、warn、error 或 fatal (默认为 info)	`lightning`
		 `'→ .`
		 `'→ log`
		 `'→ -`
		 `'→ level`
		 `'→`

-f rule	表库过滤的规则 (可多次指定)	`mydumper`
		 `'→ .`
		 `'→ filter`
		 `'→`

参数	描述	对应配置项
–backend backend	选择导入的模式：local ′→ 为Physical′→ Import Mode，tidb ′→ 为Logical Import Mode	tikv-′→ importer′→ .′→ backend
–log-file file	日志文件路径（默认值为 /tmp/′→ lightning′→ .′→ log′→ .{′→ timestamp′→ }，设置为 ‘-’ 表示日志输出到终端）	lightning′→ .′→ log′→ -′→ file
–status-addr ip:port	TiDB Lightning 服务器的监听地址	lightning′→ .′→ status′→ -′→ port′→
–importer host:port	TiKV Importer 的地址	tikv-′→ importer′→ .′→ addr′→
–pd-urls host:port	PD endpoint 的地址	tidb.′→ pd-′→ addr′→

参数	描述	对应配置项
–tidb-host host	TiDB Server 的 host	tidb. ′→ host ′→
–tidb-port port	TiDB Server 的端口（默认为 4000）	tidb. ′→ port ′→
–tidb-status port	TiDB Server 的状态端口的（默认为 10080）	tidb. ′→ status ′→ - ′→ port ′→
–tidb-user user	连接到 TiDB 的用户名	tidb. ′→ user ′→
–tidb-password pass-word	连接到 TiDB 的密码，可为明文或 Base64 编码	tidb. ′→ password ′→
–enable-checkpoint bool	是否启用断点 (默认值为 true)	checkpoint ′→ . ′→ enable ′→
–analyze level	导入后分析表信息，可选值为 re-quired、optional（默认值）、off	post- ′→ restore ′→ . ′→ analyze ′→

参数	描述	对应配置项
–checksum level	导入后比较校验和，可选值为 required（默认值）、optional、off	post-restore.checksum
–check-requirement bool	开始之前检查集群版本兼容性（默认值为 true）	lightning.check-requirements
–ca file	TLS 连接的 CA 证书路径	security.ca-path
–cert file	TLS 连接的证书路径	security.cert-path
–key file	TLS 连接的私钥路径	security.key-path
–server-mode	在服务器模式下启动 TiDB Light-ning	lightning.server-mode

如果同时对命令行参数和配置文件中的对应参数进行更改，命令行参数将优先生效。例如，在 `cfg.toml` 文件

中，不管对日志等级做出什么修改，运行 `./tidb-lightning -L debug --config cfg.toml` 命令总是将日志级别设置为“debug”。

`tidb-lightning-ctl`

所有 `tidb-lightning` 的参数也适用于 `tidb-lightning-ctl`。此外，使用 `tidb-lightning-ctl` 还可以对下列参数进行配置：

参数	描述
–compact	执行 full compact
–switch-mode mode	将每个TiKV Store 切换到指定模式（ normal 或 import ）
–fetch-mode	打印每个TiKV Store 的当前模式
–import-engine uuid	将 TiKV Importer 上关闭的引擎文件导入到TiKV 集群
–cleanup-engine uuid	删除TiKV Importer 上的引擎文件
–checkpoint-dump folder	将当前的断点以CSV 格式存储到文件夹中
–checkpoint-error-destroy tablename	删除断点，如果报错则删除该表
–checkpoint-error-ignore tablename	忽略指定表中断点的报错
–checkpoint-remove tablename	无条件删除表的断点

tablename 必须是`db`.`tbl` 中的限定表名（包括反引号），或关键词 `all`。

10.7.13.7 TiDB Lightning 监控告警

`tidb-lightning` 支持使用 Prometheus 采集监控指标(metrics)。本文主要介绍TiDB Lightning 的监控配置与监控指标。

10.7.13.7.1 监控配置

如果是手动部署TiDB Lightning，则参照以下步骤进行配置。

只要 Prometheus 能发现 `tidb-lightning` 和 `tikv-importer` 的监控地址，就能收集对应的监控指标。

监控的端口可在 `tidb-lightning.toml` 中配置：

```
[lightning]
#### 用于调试和 Prometheus 监控的 HTTP 端口。输入 0 关闭。
pprof-port = 8289

...
```

监控的端口也可在 `tikv-importer.toml` 配置：

```
#### 状态服务器的监听地址
status-server-address = '0.0.0.0:8286'
```

配置Prometheus 后，`tidb-lightning` 才能发现服务器。配置方法如下，将服务器地址直接添加至`scrape_configs` 部分：

```
...
scrape_configs:
  - job_name: 'lightning'
    static_configs:
      - targets: ['192.168.20.10:8289']
  - job_name: 'tikv-importer'
    static_configs:
      - targets: ['192.168.20.9:8286']
```

10.7.13.7.2 Grafana 面板

Grafana 的可视化面板可以让你在网页上监控 Prometheus 指标。

第一行：速度面板

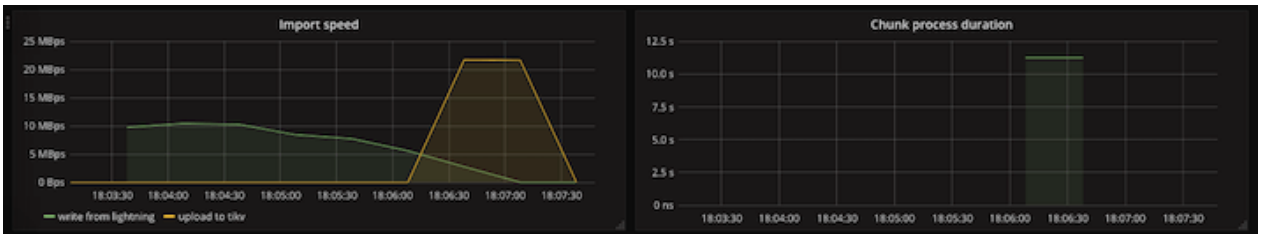


图 187: 第一行速度面板

面板名称	序列	描述
Import speed	write from lightning	从 TiDB Lightning 向 TiKV Importer 发送键值对的速度，取决于每个表的复杂性

面板名称	序列	描述
Import speed	upload to tikv	从 TiKV Importer 上传 SST 文件到所有TiKV副本的总体速度
Chunk process duration		完全编码单个数据文件所需的平均时间

有时导入速度会降到 0，这是为了平衡其他部分的速度，属于正常现象。

第二行：进度面板

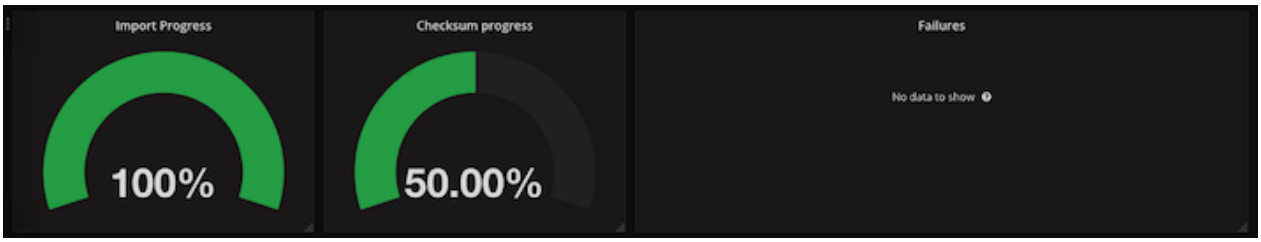


图 188: 第二行进度面板

面板名称	描述
Import progress	已编码的文件所占百分比
Checksum progress	已导入的表所占百分比
Failures	导入失败的表的数量以及故障点，通常为空

第三行：资源使用面板

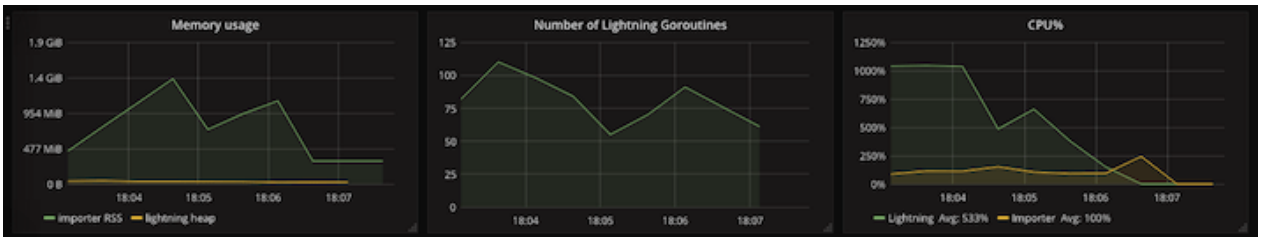


图 189: 第三行资源使用面板

面板名称	描述
Memory usage	每个服务占用的内存
Number of Lightning Goroutines	TiDB Lightning 使用的运行中的 goroutines 数量
CPU%	每个服务使用的逻辑 CPU 数量

第四行：配额使用面板

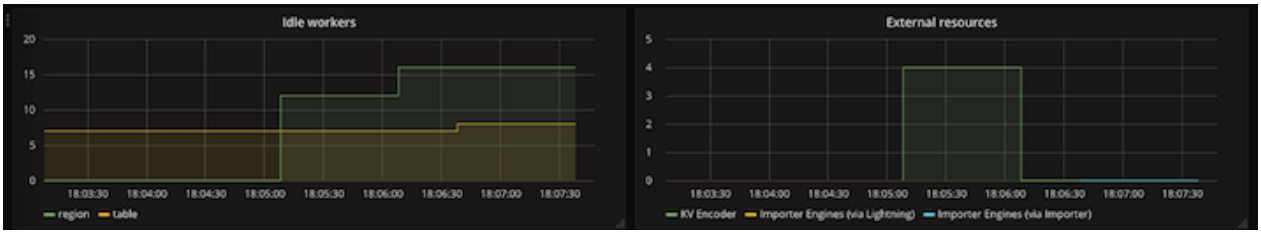


图 190: 第四行配额使用面板

面板名称	序列	描述
Idle workers	io	未使用的 `io-concurrency` 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢

面板名称	序列	描述
Idle workers	closed-engine	已关闭但未清理的引擎数量，通常接近 `index-concurrency` 与 `table-concurrency` 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟
Idle workers	table	未使用的 `table-concurrency` 的数量，通常为 0，直到进程结束
Idle workers	index	未使用的 `index-concurrency` 的数量，通常为 0，直到进程结束

面板名称	序列	描述
Idle workers	region	未使用的 region-concurrency 的数量，通常为 0，直到进程结束
External resources	KV Encoder	已激活的 KV encoder 的数量，通常与 region-concurrency 的数量相同，直到进程结束
External resources	Importer Engines	打开的引擎文件数量，不应超过 max-open-engines 的设置

第五行：读取速度面板

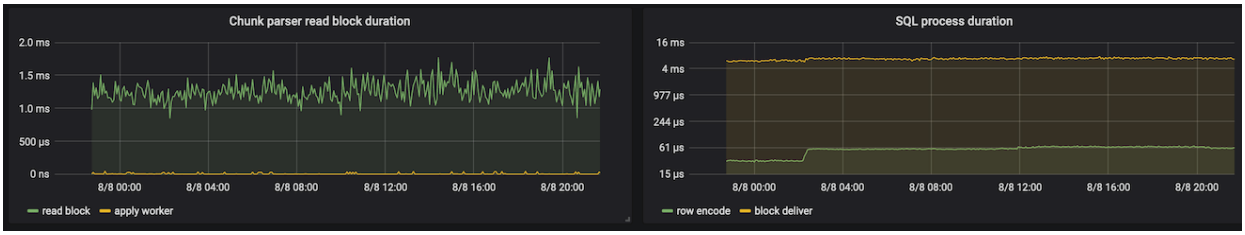


图 191: 第五行读取速度面板

面板名称	序列	描述
Chunk parser read block duration	read block	读取一个字节块来准备解析时所消耗的时间
Chunk parser read block duration	apply worker	等待 `io-concurrency` 空闲所消耗的时间
SQL process duration	row encode	解析和编码单行所消耗的时间
SQL process duration	block deliver	将一组键值对发送到TiKV Importer 所消耗的时间

如果上述项的持续时间过长，则表示TiDB Lightning 使用的磁盘运行太慢或I/O 太忙。

第六行：存储空间面板

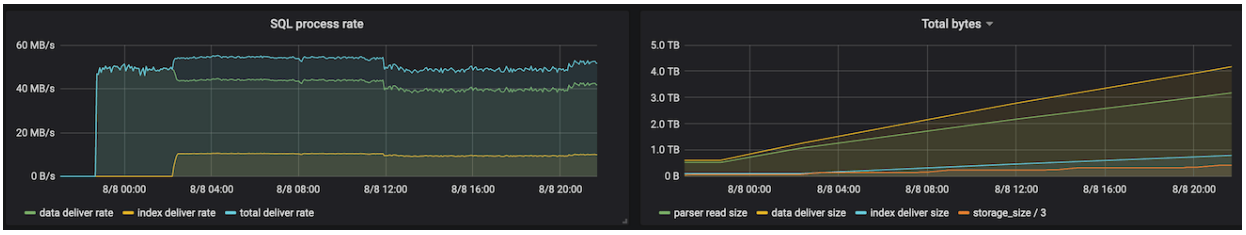


图 192: 第六行存储空间面板

面板名称	序列	描述
SQL process rate	data deliver rate	向 TiKV Importer 发送数据键值对的速度
SQL process rate	index deliver rate	向 TiKV Importer 发送索引键值对的速度
SQL process rate	total deliver rate	发送数据键值对及索引键值对的速度之和
Total bytes	parser read size	TiDB Lightning 正在读取的字节数
Total bytes	data deliver size	已发送到TiKV Importer 的数据键值对的字节数
Total bytes	index deliver size	已发送到TiKV Importer 的索引键值对的字节数
Total bytes	storage_size/3	TiKV 集群占用的存储空间大小的 1/3（3 为默认的副本数量）

第七行：导入速度面板



图 193: 第七行导入速度面板

面板名称	序列	描述
Delivery duration	Range delivery	将一个range 的键值对上传到TiKV 集群所消耗的时间
Delivery duration	SST delivery	将单个 SST 文件上传到TiKV 集群所消耗的时间

面板名称	序列	描述
SST process duration	Split SST	将键值对流切分成若干 SST 文件所消耗的时间
SST process duration	SST upload	上传单个 SST 文件所消耗的时间
SST process duration	SST ingest	ingest 单个 SST 文件所消耗的时间
SST process duration	SST size	单个 SST 文件的大小

10.7.13.7.3 监控指标

本节将详细描述 `tikv-importer` 和 `tidb-lightning` 的监控指标。

`tikv-importer`

`tikv-importer` 的监控指标皆以 `tikv_import_*` 为前缀。

- `tikv_import_rpc_duration`（直方图）
  完成一次RPC 用时直方图。标签：
  - request：所执行RPC 请求的类型
    * `switch_mode` — 将一个TiKV 节点切换为import/normal 模式
    * `open_engine` — 打开引擎文件
    * `write_engine` — 接收数据并写入引擎文件
    * `close_engine` — 关闭一个引擎文件
    * `import_engine` — 导入一个引擎文件到TiKV 集群中
    * `cleanup_engine` — 删除一个引擎文件
    * `compact_cluster` — 显式压缩TiKV 集群
    * `upload` — 上传一个 SST 文件
    * `ingest` — Ingest 一个 SST 文件
    * `compact` — 显式压缩一个TiKV 节点
  - result：RPC 请求的执行结果
    * ok
    * error
- `tikv_import_write_chunk_bytes`（直方图）
  从TiDB Lightning 接收的键值对区块大小（未压缩）的直方图。
- `tikv_import_write_chunk_duration`（直方图）
  从 `tidb-lightning` 接收每个键值对区块所需时间的直方图。
- `tikv_import_upload_chunk_bytes`（直方图）
  上传到TiKV 的每个 SST 文件区块大小（压缩）的直方图。
- `tikv_import_range_delivery_duration`（直方图）
  将一个range 的键值对发送至 `dispatch-job` 任务所需时间的直方图。
- `tikv_import_split_sst_duration`（直方图）
  将range 从引擎文件中分离到单个 SST 文件中所需时间的直方图。

- tikv_import_sst_delivery_duration（直方图）
  将 SST 文件从 dispatch-job 任务发送到 ImportSSTJob 任务所需时间的直方图
- tikv_import_sst_recv_duration（直方图）
  ImportSSTJob 任务接收从 dispatch-job 任务发送过来的 SST 文件所需时间的直方图。
- tikv_import_sst_upload_duration（直方图）
  从 ImportSSTJob 任务上传 SST 文件到TiKV 节点所需时间的直方图。
- tikv_import_sst_chunk_bytes（直方图）
  上传到TiKV 节点的 SST 文件（压缩）大小的直方图。
- tikv_import_sst_ingest_duration（直方图）
  将 SST 文件传入至TiKV 所需时间的直方图。
- tikv_import_each_phase（测量仪）
  表示运行阶段。值为 1 时表示在阶段内运行，值为 0 时表示在阶段内运行。标签：
  - phase：prepare / import
- tikv_import_wait_store_available_count（计数器）
  计算出现TiKV 节点没有充足空间上传 SST 文件现象的次数。标签：
  - store_id：TiKV 存储ID。
- tikv_import_upload_chunk_duration（直方图）
  上传到TiKV 的每个区块所需时间的直方图。

tidb-lightning

tidb-lightning 的监控指标皆以 lightning_* 为前缀。

- lightning_importer_engine（计数器）
  计算已开启及关闭的引擎文件数量。标签：
  - type:
    * open
    * closed
- lightning_idle_workers（计量表盘）
  计算闲置的 worker。标签：
  - name：
    * table — 未使用的 table-concurrency 的数量，通常为 0，直到进程结束
    * index — 未使用的 index-concurrency 的数量，通常为 0，直到进程结束
    * region — 未使用的 region-concurrency 的数量，通常为 0，直到进程结束

    * io — 未使用的 io-concurrency 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢
    * closed-engine — 已关闭但未清理的引擎数量， 通常接近 index-concurrency 与 table-concurrency 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟

- lightning_kv_encoder（计数器）

  计算已开启及关闭的 KV 编码器。KV 编码器是运行于内存的TiDB 实例，用于将 SQL 的 INSERT 语句转换成键值对。此度量的净值（开启减掉关闭）在正常情况下不应持续增长。标签：

  - type:
    * open
    * closed

- lightning_tables（计数器）

  计算处理过的表及其状态。标签：

  - state：表的状态，表明当前应执行的操作
    * pending — 等待处理
    * written — 所有数据已编码和传输
    * closed — 所有对应的引擎文件已关闭
    * imported — 所有引擎文件已上传到目标集群
    * altered_auto_inc — 自增ID 已改
    * checksum — 已计算校验和
    * analyzed — 已进行统计信息分析
    * completed — 表格已完全导入并通过验证
  - result：当前操作的执行结果
    * success — 成功
    * failure — 失败（未完成）

- lightning_engines（计数器）

  计算处理后引擎文件的数量以及其状态。标签：

  - state：引擎文件的状态，表明当前应执行的操作
    * pending — 等待处理
    * written — 所有数据已编码和传输
    * closed — 引擎文件已关闭
    * imported — 当前引擎文件已上传到目标集群
    * completed — 当前引擎文件已完全导入
  - result：当前操作的执行结果
    * success — 成功
    * failure — 失败（未完成）

- lightning_chunks（计数器）

  计算处理过的 Chunks 及其状态。标签：

  - state: 单个 Chunk 的状态，表明该 Chunk 当前所处的阶段
    * `estimated` — （非状态）当前任务中 Chunk 的数量
    * `pending` — 已载入但未执行
    * `running` — 正在编码和发送数据
    * `finished` — 该 Chunk 已处理完毕
    * `failed` — 处理过程中发生错误

- `lightning_import_seconds`（直方图）
  导入每个表所需时间的直方图。
- `lightning_row_read_bytes`（直方图）
  单行 SQL 数据大小的直方图。
- `lightning_row_encode_seconds`（直方图）
  解码单行 SQL 数据到键值对所需时间的直方图。
- `lightning_row_kv_deliver_seconds`（直方图）
  发送一组与单行 SQL 数据对应的键值对所需时间的直方图。
- `lightning_block_deliver_seconds`（直方图）
  每个键值对中的区块传送到 `tikv-importer` 所需时间的直方图。
- `lightning_block_deliver_bytes`（直方图）
  发送到Importer 的键值对中区块（未压缩）的大小的直方图。
- `lightning_chunk_parser_read_block_seconds`（直方图）
  数据文件解析每个 SQL 区块所需时间的直方图。
- `lightning_checksum_seconds`（直方图）
  计算表中 Checksum 所需时间的直方图。
- `lightning_apply_worker_seconds`（直方图）
  获取闲置 worker 等待时间的直方图(参见 `lightning_idle_workers` 计量表盘)。标签：

  - name：
    * table
    * index
    * region
    * io
    * closed-engine

10.7.13.8 TiDB Lightning Web 界面

TiDB Lightning 支持在网页上查看导入进度或执行一些简单任务管理，这就是TiDB Lightning 的服务器模式。本文将介绍服务器模式下的Web 界面和一些常见操作。

启用服务器模式的方式有如下几种：

1. 在启动 tidb-lightning 时加上命令行参数 --server-mode。

```
tiup tidb-lightning --server-mode --status-addr :8289
```

2. 在配置文件中设置 lightning.server-mode。

```
[lightning]
server-mode = true
status-addr = ':8289'
```

TiDB Lightning 启动后，可以访问 http://127.0.0.1:8289 来管理程序（实际的 URL 取决于你的 status-addr 设置）。

服务器模式下，TiDB Lightning 不会立即开始运行，而是通过用户在 web 页面提交（多个）任务来导入数据。

10.7.13.8.1 TiDB Lightning Web 首页

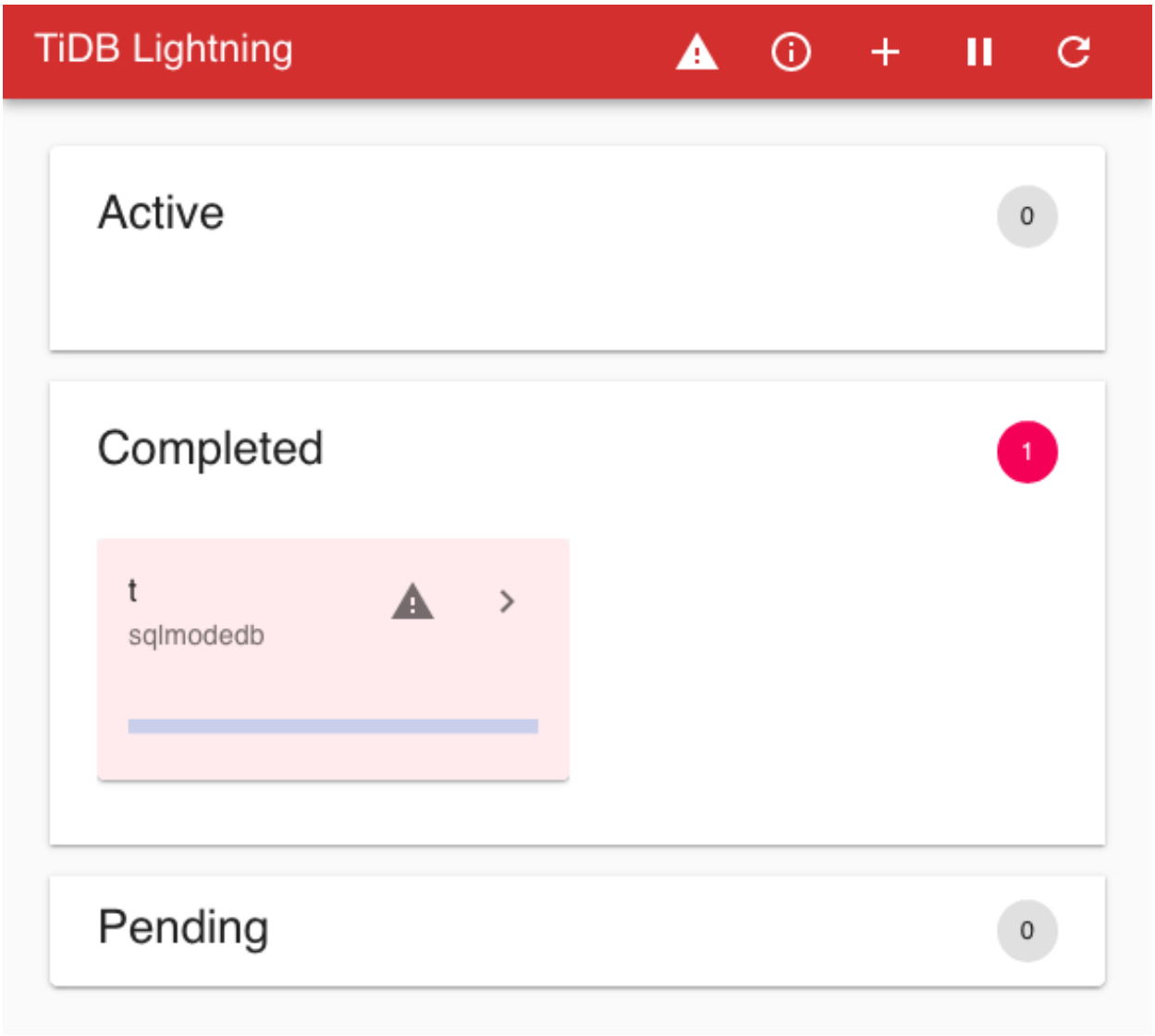


图 194: TiDB Lightning Web 首页

标题栏上图标所对应的功能，从左到右依次为：

图标	功能
“TiDB Lightning”	点击即返回首页
□	显示前一个任务的所有错误信息
ⓘ	列出当前及队列中的任务，可能会出现一个标记提示队列中任务的数量
+	提交单个任务
□/▸	暂停/继续当前操作
□	设置网页自动刷新

标题栏下方的三个面板显示了不同状态下的所有表：

- Active：当前正在导入这些表
- Completed：这些表导入成功或失败
- Pending：这些表还没有被处理

每个面板都包含用于描述表状态的卡片。

10.7.13.8.2 提交任务

点击标题栏的+图标提交任务。

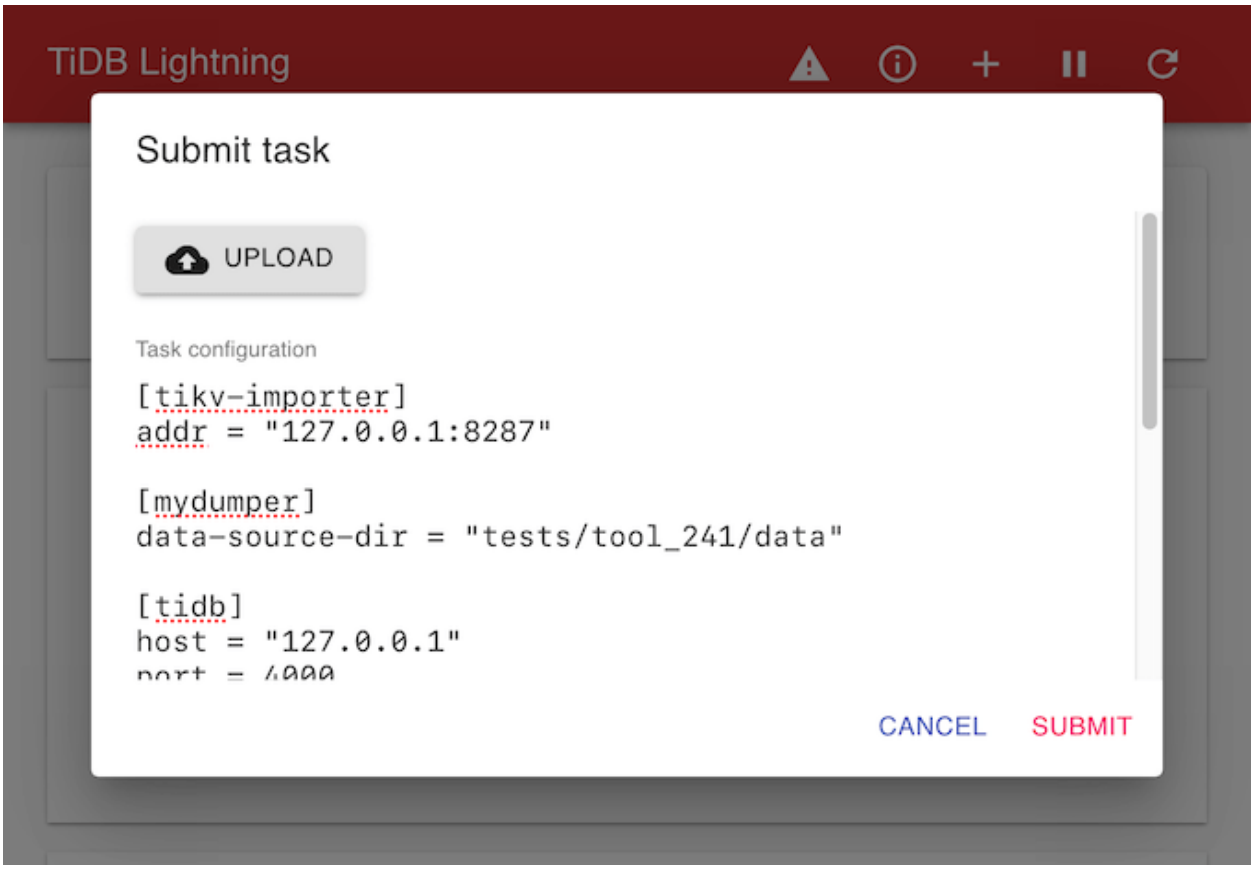


图 195: 提交任务对话框

任务(task) 为 TOML 格式的文件，具体参考TiDB Lightning 任务配置。你也可以点击 UPLOAD 上传一个本地的 TOML 文件。

点击 SUBMIT 运行任务。如果当前有任务正在运行，新增任务会加入队列并在当前任务结束后执行。

10.7.13.8.3 查看导入进度

点击首页表格卡片上的>图标，查看表格导入的详细进度。



图 196: 表格导入进度

该页显示每张表的引擎文件的导入过程。

点击标题栏上的TiDB Lightning 返回首页。

10.7.13.8.4 管理任务

单击标题栏上的ⓘ 图标来管理当前及队列中的任务。

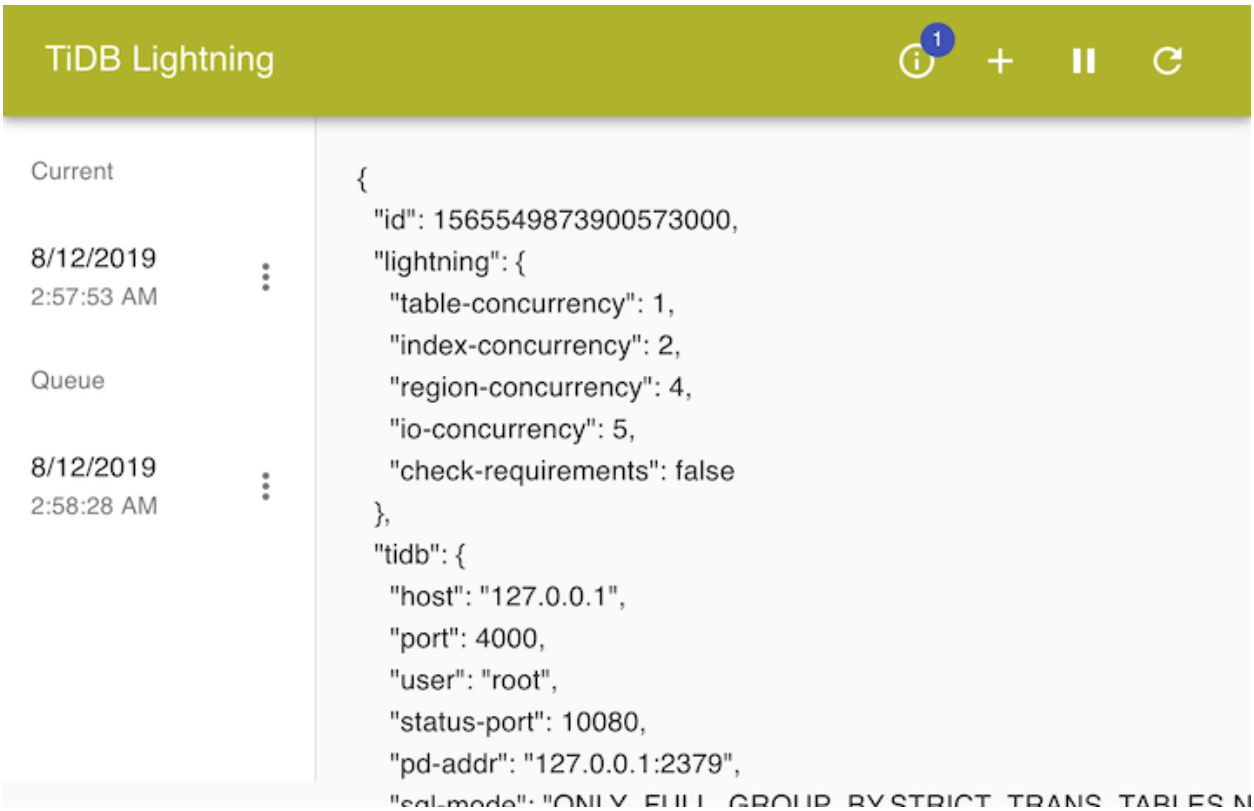


图 197: 任务管理页面

每个任务都是依据提交时间来标记。点击该任务将显示JSON 格式的配置文件。

点击任务上的▫ 可以对该任务进行管理。你可以立即停止任务，或重新排序队列中的任务。

10.7.13.9 TiDB Lightning 常见问题

本文列出了一些使用TiDB Lightning 时可能会遇到的问题与答案。

10.7.13.9.1 TiDB Lightning 对TiDB/TiKV/PD 的最低版本要求是多少？

TiDB Lightning 的版本应与集群相同。如果使用Local-backend 模式，最低版本要求为4.0.0。如果使用Importer-backend 或TiDB-backend 模式最低版本要求是 2.0.9，但建议使用最新的稳定版本 3.0。

10.7.13.9.2 TiDB Lightning 支持导入多个库吗？

支持。

10.7.13.9.3 TiDB Lightning 对下游数据库的账号权限要求是怎样的？

详细权限描述参考TiDB Lightning 使用前提。

10.7.13.9.4 TiDB Lightning 在导数据过程中某个表报错了，会影响其他表吗？进程会马上退出吗？

如果只是个别表报错，不会影响整体。报错的那个表会停止处理，继续处理其他的表。

10.7.13.9.5 如何正确重启TiDB Lightning？

如果使用Importer-backend，根据 `tikv-importer` 的状态，重启TiDB Lightning 的基本顺序如下：

如果 `tikv-importer` 仍在运行：

1. 结束 `tidb-lightning` 进程。
2. 执行修改操作（如修复数据源、更改设置、更换硬件等）。
3. 如果上面的修改操作更改了任何表，你还需要清除对应的断点。
4. 重启 `tidb-lightning`。

如果 `tikv-importer` 需要重启：

1. 结束 `tidb-lightning` 进程。
2. 结束 `tikv-importer` 进程。
3. 执行修改操作（如修复数据源、更改设置、更换硬件等）。
4. 重启 `tikv-importer`。
5. 重启 `tidb-lightning` 并等待，直到程序因校验和错误（如果有的话）而失败。
    - 重启 `tikv-importer` 将清除所有仍在写入的引擎文件，但是 `tidb-lightning` 并不会感知到该操作。从 v3.0 开始，最简单的方法是让 `tidb-lightning` 继续，然后再重试。
6. 清除失败的表及断点。
7. 再次重启 `tidb-lightning`。

如果使用 Local-backend 和TiDB-backend，操作和Importer-backend 的 `tikv-importer` 仍在运行时相同。

10.7.13.9.6 如何校验导入的数据的正确性？

TiDB Lightning 默认会对导入数据计算校验和(checksum)，如果校验和不一致就会停止导入该表。可以在日志看到相关的信息。

TiDB 也支持从 MySQL 命令行运行 `ADMIN CHECKSUM TABLE` 指令来计算校验和。

```
ADMIN CHECKSUM TABLE `schema`.`table`;
```

```
+---------+------------+--------------------+-----------+-------------+
| Db_name | Table_name | Checksum_crc64_xor | Total_kvs | Total_bytes |
+---------+------------+--------------------+-----------+-------------+
| schema  | table      | 5505282386844578743 |         3 |          96 |
+---------+------------+--------------------+-----------+-------------+
1 row in set (0.01 sec)
```

10.7.13.9.7 TiDB Lightning 支持哪些格式的数据源？

目前，TiDB Lightning 支持：

- 导入Dumpling、CSV 或Amazon Aurora Parquet 输出格式的数据源。
- 从本地盘或Amazon S3 云盘读取数据。

10.7.13.9.8 我已经在下游创建好库和表了，TiDB Lightning 可以忽略建库建表操作吗？

自 v5.1 起，TiDB Lightning 可以自动识别下游的库和表。如果你使用低于 v5.1 的 TiDB Lightning，需在配置文档中的 `[mydumper]` 部分将 `no-schema` 设置为 `true` 即可。`no-schema=true` 会默认下游已经创建好所需的数据库和表，如果没有创建，会报错。

10.7.13.9.9 有些不合法的数据，能否通过关掉严格 SQL 模式(Strict SQL Mode) 来导入？

可以。TiDB Lightning 默认的 `sql_mode` 为 `"STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"`。

这个设置不允许一些非法的数值，例如 `1970-00-00` 这样的日期。可以修改配置文件 `[tidb]` 下的 `sql-mode` 值。

```
...
[tidb]
sql-mode = ""
...
```

10.7.13.9.10 可以启用一个 `tikv-importer`，同时有多个 `tidb-lightning` 进程导入数据吗？

只要每个TiDB Lightning 操作的表互不相同就可以。

10.7.13.9.11 如何正确结束 `tikv-importer` 进程？

手动部署：如果 `tikv-importer` 正在前台运行，可直接按 Ctrl+C 退出。否则，可通过 `ps aux | grep tikv-importer` 获取进程ID，然后通过 `kill «pid»` 结束进程。

10.7.13.9.12 如何正确结束 `tidb-lightning` 进程？

根据部署方式，选择相应操作结束进程。

手动部署：如果 `tidb-lightning` 正在前台运行，可直接按 Ctrl+C 退出。否则，可通过 `ps aux | grep tidb-lightning` 获取进程ID，然后通过 `kill -2 «pid»` 结束进程。

10.7.13.9.13 TiDB Lightning 可以使用千兆网卡吗？

使用TiDB Lightning 的 SST Mode 建议配置万兆网卡。

千兆网卡的总带宽只有 120 MB/s，而且需要与整个 TiKV 集群共享。在使用TiDB Lightning Physical Import Mode 导入时，极易用尽所有带宽，继而因 PD 无法联络集群使集群断连。

10.7.13.9.14 为什么TiDB Lightning需要在TiKV集群预留这么多空间？

当使用默认的 3 副本设置时，TiDB Lightning 需要TiKV 集群预留数据源大小 6 倍的空间。多出来的 2 倍是算上下列没储存在数据源的因素的保守估计：

- 索引会占据额外的空间
- RocksDB 的空间放大效应

10.7.13.9.15 TiDB Lightning使用过程中是否可以重启TiKV Importer？

不能，TiKV Importer 会在内存中存储一些引擎文件，重启后，`tidb-lightning` 会因连接失败而停止。此时，你需要清除失败的断点，因为这些TiKV Importer 特有的信息丢失了。你可以在之后重启TiDB Lightning。

10.7.13.9.16 如何清除所有与TiDB Lightning相关的中间数据？

1. 删除断点文件。

    ```
    tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-remove=all
    ```

    如果出于某些原因而无法运行该命令，你可以尝试手动删除 `/tmp/tidb_lightning_checkpoint.pb` 文件。

2. 如果使用 Local-backend，删除配置中 `sorted-kv-dir` 对应的目录；如果使用Importer-backend，删除 `tikv-importer` 所在机器上的整个 `import` 文件目录。

3. 如果需要的话，删除TiDB 集群上创建的所有表和库。

4. 清理残留的元信息。如果存在以下任意一种情况，需要手动清理元信息库：

    - 对于v5.1.x 和v5.2.x 版本的TiDB Lightning, tidb-lightning-ctl 命令没有同时清理存储在目标集群的metadata 库，需要手动清理。
    - 如果手动删除过断点文件，则需要手动清理下游的元信息库，否则可能影响后续导入的正确性。

    使用下面命令清理元信息：

    ```
    DROP DATABASE IF EXISTS `lightning_metadata`;
    ```

10.7.13.9.17 如何获取TiDB Lightning运行时的 goroutine 信息

1. 如果TiDB Lightning的配置文件中已经指定了`status-port`，可以跳过此步骤。否则，需要向 TiDB Lightning 发送 USR1 信号以开启 `status-port`。
    首先通过 `ps` 等命令获取TiDB Lightning 的进程 PID，然后运行如下命令：

    ```
    kill -USR1 <lightning-pid>
    ```

    查看TiDB Lightning 的日志，其中 `starting HTTP server` / `start HTTP server` / `started HTTP server` 的日志会显示新开启的 `status-port`。

2. 访问 `http://<lightning-ip>:<status-port>/debug/pprof/goroutine?debug=2` 可获取 goroutine 信息。

10.7.13.9.18 为什么TiDB Lightning 不兼容 Placement Rules in SQL？

TiDB Lightning 不兼容 Placement Rules in SQL。当 TiDB Lightning 导入的数据中包含放置策略(placement policy) 时，TiDB Lightning 会报错。

不兼容的原因如下：

使用 Placement Rules in SQL，你可以从表或分区级别控制某些 TiKV 节点的数据存储位置。TiDB Lightning 从文本文件中读取数据，并导入到目标TiDB 集群中。如果导出的数据文件中包含了放置规则(placement rules) 的定义，在导入过程中，TiDB Lightning 必须根据该定义在目标集群中创建相应的放置规则策略。然而，当源集群和目标集群的拓扑结构不同时，这可能会导致问题。

假设源集群有如下拓扑结构：

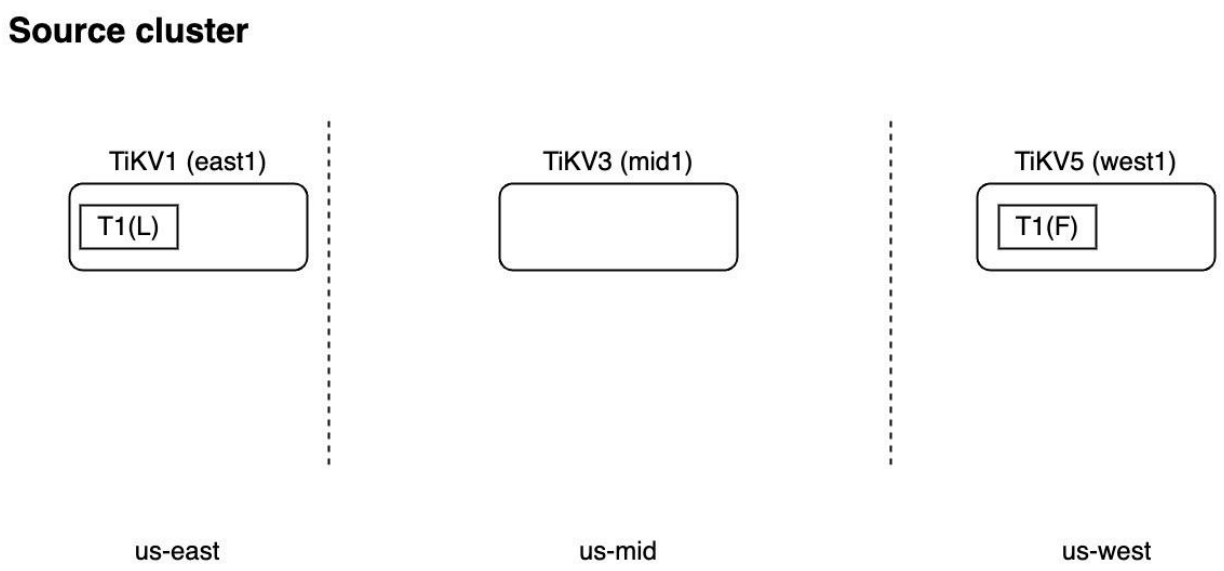


图 198: TiDB Lightning FAQ - 源集群拓扑结构

源集群中设置了这样的放置策略：

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east" REGIONS="us-east,us-west";
```

场景 1：目标集群中有 3 个副本，且拓扑结构与源集群不同。在这种情况下，当TiDB Lightning 在目标集群中创建放置策略时，TiDB Lightning 不会报错，但目标集群中的语义是错误的。

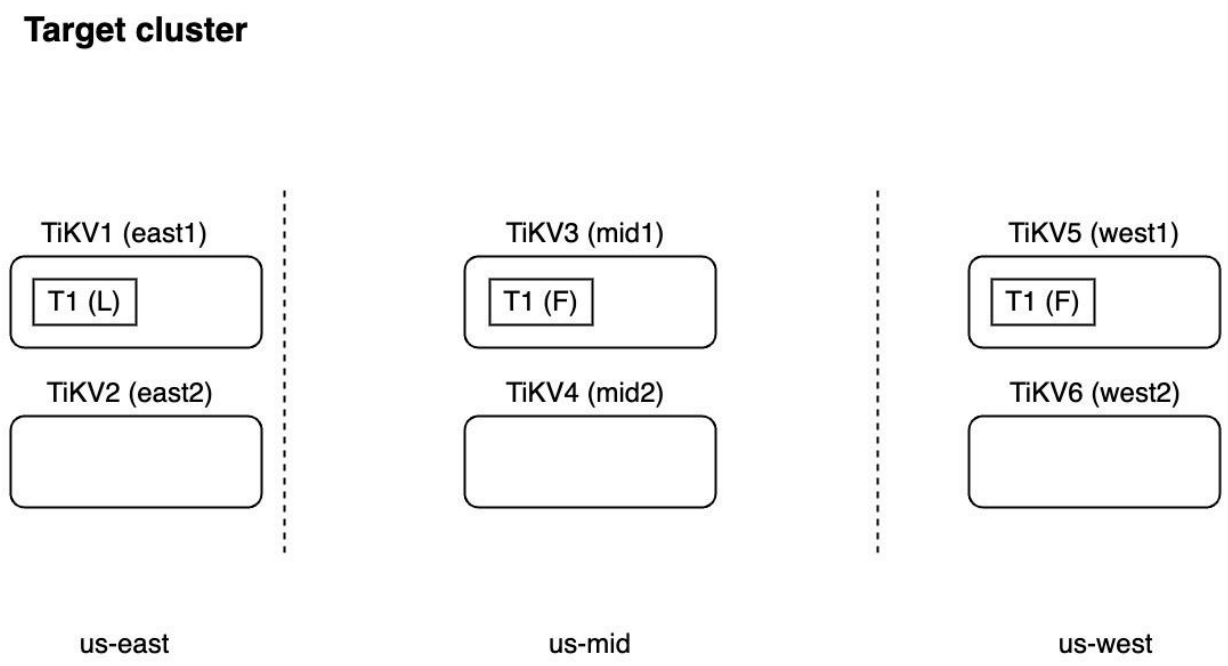


图 199: TiDB Lightning FAQ - 场景 1

场景 2：目标集群将 Follower 副本放置在区域“us-mid”的另一个TiKV 节点上，且在拓扑结构中没有节点位于区域“us-west”中。在这种情况下，当TiDB Lightning 在目标集群中创建放置策略时，TiDB Lightning 将报错。

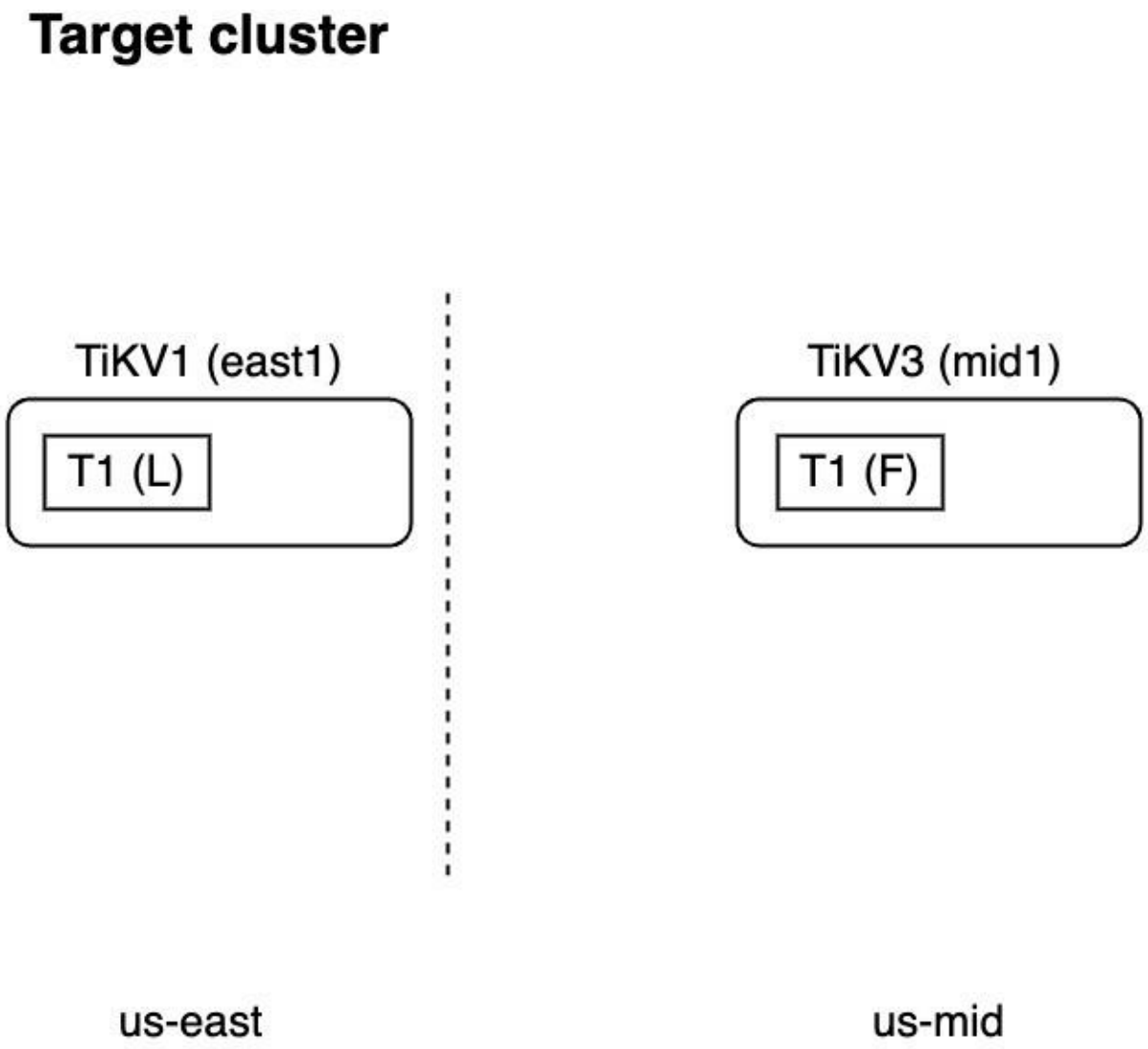


图 200: TiDB Lightning FAQ - 场景 2

解决方法：

如果要在使用TiDB Lightning 同时使用 Placement Rules in SQL，你需要在导入数据到目标表之前，确保已经在目标 TiDB 集群中创建了相关的label 和对象。因为 Placement Rules in SQL 作用于 PD 和 TiKV 层，TiDB Lightning 可以根据获取到的信息，决定应该使用哪个TiKV 来存储导入的数据。使用这种方法后，Placement Rules in SQL 对 TiDB Lightning 是透明无感的。

具体操作步骤如下：

1. 规划数据分布的拓扑结构。
2. 为 TiKV 和 PD 配置必要的label。
3. 创建放置规则策略，并将策略应用到目标表上。

4. 使用TiDB Lightning 导入数据到目标表。

10.7.13.10 TiDB Lightning 术语表

本术语表提供了TiDB Lightning 相关的术语和定义，这些术语会出现在TiDB Lightning 的日志、监控指标、配置和文档中。

10.7.13.10.1 A

Analyze

统计信息分析。指重建TiDB 表中的统计信息，即运行 `ANALYZE TABLE` 语句。

因为 TiDB Lightning local backend 直接导入数据到TiKV，统计信息不会自动更新，所以TiDB Lightning 在导入后显式地分析每个表。如果不需要该操作，可以将 `post-restore.analyze` 设置为 `false`。

AUTO_INCREMENT_ID

用于为自增列分配默认值的自增ID 计数器。每张表都有一个相关联的 `AUTO_INCREMENT_ID` 计数器。在TiDB 中，该计数器还用于分配行ID。

因为TiDB Lightning local backend 直接导入数据到TiKV，`AUTO_INCREMENT_ID` 计数器不会自动更新，所以TiDB Lightning 显式地将 `AUTO_INCREMENT_ID` 改为一个有效值。即使表中没有自增列，这步仍是会执行。

10.7.13.10.2 B

Backend

也称作 Back end（后端），用于接受 TiDB Lightning 解析结果。

详情参阅TiDB Lightning Backends。

10.7.13.10.3 C

Checkpoint

断点。用于保证TiDB Lightning 在导入数据时不断地将进度保存到本地文件或远程数据库中。这样即使进程崩溃，TiDB Lightning 也能从中间状态恢复。

详情参见TiDB Lightning 断点续传。

Checksum

校验和。一种用于验证导入数据正确性的方法。

在 TiDB Lightning 中，表的校验和是由 3 个数字组成的集合，由该表中每个键值对的内容计算得出。这些数字分别是：

- 键值对的数量
- 所有键值对的总长度
- 每个键值对 CRC-64-ECMA 按位异或的结果

TiDB Lightning 通过比较每个表的本地校验和和远程校验和来验证导入数据的正确性。如果有任一对校验和不匹配，导入进程就会停止。如果你需要跳过校验和检查，可以将 `post-restore.checksum` 设置为 `false`。

遇到校验和不匹配的问题时，参考常见问题进行处理。

Chunk

一段连续的源数据，通常相当于数据源中的单个文件。

如果单个文件太大，TiDB Lightning 可以将单个文件拆分成多个文件块。

Compaction

压缩。指将多个小 SST 文件合并为一个大 SST 文件并清理已删除的条目。TiDB Lightning 导入数据时，TiKV 在后台会自动压缩数据。

> 注意：
>
> 出于遗留原因，你仍然可以将TiDB Lightning 配置为在每次导入表时进行显式压缩，但是官方不推荐采用该操作，且该操作的相关设置默认是禁用的。
>
> 技术细节参阅RocksDB 关于压缩的说明。

10.7.13.10.4 D

Data engine

数据引擎。用于对实际的行数据进行排序的引擎。

当一个表数据很多的时候，表的数据会被放置在多个数据引擎中以改善任务流水线并节省TiKV Importer 的空间。默认条件下，每 100 GB 的 SQL 数据会打开一个新的数据引擎（可通过 `mydumper.batch-size` 配置项进行更改）。

TiDB Lightning 同时处理多个数据引擎（可通过 `lightning.table-concurrency` 配置项进行更改）。

10.7.13.10.5 E

Engine

引擎。在TiKV Importer 中，一个引擎就是一个用于排序键值对的RocksDB 实例。

TiDB Lightning 通过引擎将数据传送到TiKV Importer 中。TiDB Lightning 先打开一个引擎，向其发送未排序的键值对，然后关闭引擎。随后，引擎会对收到的键值对进行排序操作。这些关闭的引擎可以进一步上传至TiKV store 中为Ingest 做准备。

引擎使用TiKV Importer 的 `import-dir` 作为临时存储，有时也会被称为引擎文件(engine files)。

另见数据引擎和索引引擎。

10.7.13.10.6 F

Filter

配置列表，用于指定需要导入或不允许导入的表。

详情见表库过滤。

10.7.13.10.7 I

Import mode

导入模式。指通过降低读取速度和减少空间使用，来优化TiKV 写入的配置模式。

导入过程中，TiDB Lightning 自动在导入模式和普通模式中来回切换。如果 TiKV 卡在导入模式，你可以使用 `tidb-lightning-ctl` 强制切换回普通模式。

Index engine

索引引擎。用于对索引进行排序的引擎。

不管表中有多少索引，每张表都只对应一个索引引擎。

TiDB Lightning 可同时处理多个索引引擎（可通过 `lightning.index-concurrency` 配置项进行更改）。由于每张表正好对应一个索引引擎，`lightning.index-concurrency` 配置项也限定了可同时处理的表的最大数量。

Ingest

指将SST 文件的全部内容插入到RocksDB（TiKV）store 中的操作。

与逐个插入键值对相比，Ingest 的效率非常高。因此，该操作直接决定了TiDB Lightning 的性能。

技术细节参阅RocksDB 关于创建、Ingest SST 文件的 wiki 页面。

10.7.13.10.8 K

KV pair

即 key-value pair（键值对）。

KV encoder

用于将 SQL 或CSV 行解析为键值对的例程。多个 KV encoder 会并行运行以加快处理速度。

10.7.13.10.9 L

Local checksum

本地校验和。在将键值对发送到TiKV Importer 前，由TiDB Lightning 计算的表的校验和。

10.7.13.10.10 N

Normal mode

普通模式。未启用导入模式时的模式。

10.7.13.10.11 P

Post-processing

指整个数据源被解析发送到TiKV Importer 之后的一段时间。此时 TiDB Lightning 正在等待TiKV Importer 上传、Ingest SST 文件。

10.7.13.10.12 R

Remote checksum

远程校验和。指导入TiDB 后所计算的表的校验和。

10.7.13.10.13 S

Scattering

指随机再分配Region 中leader 和peer 的操作。Scattering 确保导入的数据在TiKV store 中均匀分布，这样可以降低 PD 调度的压力。

Splitting

指 TiKV Importer 在上传之前会将单个引擎文件拆分为若干小SST 文件的操作。这是因为引擎文件通常很大（约为 100 GB），在 TiKV 中不适合视为单一的Region。拆分的文件大小可通过 `import.region-split-size` 配置项更改。

SST file

Sorted string table file（排序字符串表文件）。SST 文件是一种在RocksDB 中（因而也是TiKV 中）键值对集合在本地的存储形式。

TiKV Importer 从关闭的引擎中生成 SST 文件。这些 SST 文件接着被上传、ingest 到 TiKV store 中。

10.8 使用 Dumpling 导出数据

使用数据导出工具 Dumpling，你可以把存储在 TiDB 或 MySQL 中的数据导出为 SQL 或 CSV 格式，用于逻辑全量备份。Dumpling 也支持将数据导出到Amazon S3 中。

要快速了解 Dumpling 的基本功能，建议先观看下面的培训视频（时长 28 分钟）。注意本视频只作为功能介绍、学习参考，具体操作步骤和最新功能，请以文档内容为准。

你可以通过下列任意方式获取 Dumpling：

- TiUP 执行 `tiup install dumpling` 命令。获取后，使用 `tiup dumpling ...` 命令运行 Dumpling。
- 下载包含 Dumpling 的tidb-toolkit 安装包。

更多详情，可以使用-help 选项查看，或参考Dumpling 主要选项表。

使用 Dumpling 时，需要在已经启动的集群上执行导出命令。

TiDB 还提供了其他工具，你可以根据需要选择使用：

- 如果需要直接备份 SST 文件（键值对），或者对延迟不敏感的增量备份，请使用备份工具BR。

- 如果需要实时的增量备份，请使用TiCDC。
- 所有的导出数据都可以用TiDB Lightning 导回到TiDB。

> **注意：**
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 mydumper project，针对TiDB 的特性进行了优化。关于 Mydumper 的更多信息，请参考 v4.0 版 Mydumper 使用文档。Mydumper 目前已经不再开发新功能，其绝大部分功能已经被Dumpling 取代，请切换到 Dumpling。

相比 Mydumper，Dumpling 做了如下改进：

- 支持导出多种数据形式，包括 SQL/CSV。
- 支持全新的table-filter，筛选数据更加方便。
- 支持导出到Amazon S3 云盘。
- 针对TiDB 进行了更多优化：
  - 支持配置TiDB 单条 SQL 内存限制。
  - 针对TiDB v4.0.0 及更新版本，如果 Dumpling 能够直接连接到 PD，则支持自动调整 TiDB GC 时间。
  - 使用TiDB 的隐藏列 `_tidb_rowid` 优化了单表内数据的并发导出性能。
  - 对于TiDB 可以设置tidb_snapshot 的值指定备份数据的时间点，从而保证备份的一致性，而不是通过 `FLUSH TABLES WITH READ LOCK` 来保证备份一致性。

> **注意：**
>
> 在以下情况下，Dumpling 无法连接到 PD：
>
> - TiDB 集群正在 Kubernetes 上运行（Dumpling 本身在 Kubernetes 环境中运行时除外）。
> - TiDB 集群正在TiDB Cloud 上运行。
>
> 在这种情况下，你需要手动调整TiDB GC 时间，以避免导出失败。

10.8.1 从 TiDB/MySQL 导出数据

10.8.1.1 需要的权限

- SELECT：导出目标表时需要。
- RELOAD：使用consistency flush 时需要。注意，只有TiDB 支持该权限，当上游为RDS 或采用托管服务时，可忽略该权限。
- LOCK TABLES：使用consistency lock 时需要，需要导出的库表都有该权限。
- REPLICATION CLIENT：导出 metadata 记录数据快照点时需要，可选，如果不需要导出metadata，可忽略该权限。

10.8.1.2 导出为 SQL 文件

本文假设在 127.0.0.1:4000 有一个TiDB 实例，并且这个TiDB 实例中有无密码的root 用户。

Dumpling 默认导出数据格式为 SQL 文件。也可以通过设置 --filetype sql 导出数据到 SQL 文件：

```
dumpling -u root -P 4000 -h 127.0.0.1 --filetype sql -t 8 -o /tmp/test -r 200000 -F256MiB
```

以上命令中：

- -h、-P、-u 分别代表地址、端口、用户。如果需要密码验证，可以使用 -p $YOUR_SECRET_PASSWORD 将密码传给 Dumpling。
- -o（或 --output）用于选择存储导出文件的目录，支持本地文件路径或外部存储 URI 格式。
- -t 用于指定导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜设置过大。一般不超过 64。
- -r 用于指定单个文件的最大行数，指定该参数后 Dumpling 会开启表内并发加速导出，同时减少内存使用。当上游为 TiDB 且版本为 v3.0 或更新版本时，设置 -r 参数大于 0 表示使用TiDB region 信息划分表内并发，具体取值不影响划分算法。对上游为 MySQL 且表的主键是int 的场景，该参数也有表内并发效果。
- -F 选项用于指定单个文件的最大大小，单位为 MiB，可接受类似 5GiB 或 8KB 的输入。如果你想使用 TiDB Lightning 将该文件加载到TiDB 实例中，建议将 -F 选项的值保持在 256 MiB 或以下。

> 注意：
>
> 如果导出的单表大小超过 10 GB，强烈建议使用 -r 和 -F 参数。

10.8.1.3 导出为CSV 文件

你可以通过使用 --filetype csv 导出数据到CSV 文件。

当你导出CSV 文件时，你可以使用 --sql <SQL> 导出指定 SQL 选择出来的记录。例如，导出 test.sbtest1 中所有 id < 100 的记录：

```
./dumpling -u root -P 4000 -h 127.0.0.1 -o /tmp/test --filetype csv --sql 'select * from `test`.`
    ′→ sbtest1` where id < 100' -F 100MiB --output-filename-template 'test.sbtest1.{{.Index}}'
```

以上命令中：

- --sql 选项仅仅可用于导出 CSV 文件的场景。上述命令将在要导出的所有表上执行 SELECT * FROM <
  ′→ table-name> WHERE id < 100 语句。如果部分表没有指定的字段，那么导出会失败。
- 使用 --sql 配置导出时，Dumpling 无法获知导出的表库信息，此时可以使用--output-filename-template
  ′→ 选项来指定 CSV 文件的文件名格式，以方便后续使用TiDB Lightning 导入数据文件。例如 --output
  ′→ -filename-template='test.sbtest1.{{.Index}}' 指定导出的 CSV 文件为 test.sbtest1.000000000、test.sbtest1.000000001 等。
- 你可以使用 --csv-separator、--csv-delimiter 等选项，配置 CSV 文件的格式。具体信息可查阅Dumpling 主要选项表。

> **注意：**
>
> Dumpling 导出不区分字符串与关键字。如果导入的数据是 Boolean 类型的 `true` 和 `false`，导出时会被转换为 `1` 和 `0`。

10.8.1.4 压缩导出的数据文件

你可以使用 `--compress <format>` 压缩导出的CSV、SQL 数据与表结构文件。该参数支持 `gzip`、`snappy`、`zstd` 压缩算法。默认不压缩。

- 该选项只能压缩单个数据与表结构文件，无法直接压缩整个文件夹生成单个压缩集合包。
- 该选项可以节省磁盘空间，但也会导致导出速度变慢，并增加 CPU 消耗。对导出速度要求较高的场景需慎用。
- TiDB Lightning v6.5.0 及以上版本支持直接使用 Dumpling 压缩文件作为数据源导入，无需额外配置。

10.8.1.5 输出文件格式

- `metadata`：此文件包含导出的起始时间，以及 master binary log 的位置。

```
cat metadata
```

```
Started dump at: 2020-11-10 10:40:19
SHOW MASTER STATUS:
        Log: tidb-binlog
        Pos: 420747102018863124

Finished dump at: 2020-11-10 10:40:20
```

- `{schema}-schema-create.sql`：创建 schema 的 SQL 文件。

```
cat test-schema-create.sql
```

```
CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4 */;
```

- `{schema}.{table}-schema.sql`：创建 table 的 SQL 文件

```
cat test.t1-schema.sql
```

```
CREATE TABLE `t1` (
  `id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

- `{schema}.{table}.{0001}.{sql|csv}`：数据源文件

```
cat test.t1.0.sql
```

```
/*!40101 SET NAMES binary*/;
INSERT INTO `t1` VALUES
(1);
```

- *-schema-view.sql、*-schema-trigger.sql、*-schema-post.sql：其他导出文件

10.8.1.6 导出到Amazon S3 云盘

Dumpling 在 v4.0.8 及更新版本支持导出到Amazon S3 云盘。如果需要将数据备份到Amazon S3 后端存储，那么需要在 -o 参数中指定Amazon S3 的存储路径。

可以参照AWS 官方文档- 如何创建 S3 存储桶在指定的 Region 区域中创建一个 S3 桶 Bucket。如有需要，还可以参照AWS 官方文档- 创建文件夹在 Bucket 中创建一个文件夹 Folder。

将有权限访问该Amazon S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Dumpling 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

Dumpling 同时还支持从 ~/.aws/credentials 读取凭证文件。Dumpling 导出到Amazon S3 的配置参数与 BR 大致相同，更多参数描述，请参考外部存储 URI 格式。

```
./dumpling -u root -P 4000 -h 127.0.0.1 -r 200000 -o "s3://${Bucket}/${Folder}"
```

10.8.1.7 筛选导出的数据

10.8.1.7.1 使用 --where 选项筛选数据

默认情况下，Dumpling 会导出排除系统数据库（包括 mysql 、 sys 、 INFORMATION_SCHEMA 、 PERFORMANCE_SCHEMA、METRICS_SCHEMA 和 INSPECTION_SCHEMA ）外所有其他数据库。你可以使用 --where <SQL where expression> 来指定要导出的记录。

```
./dumpling -u root -P 4000 -h 127.0.0.1 -o /tmp/test --where "id < 100"
```

上述命令将会导出各个表的id < 100 的数据。注意 --where 参数无法与 --sql 一起使用。

10.8.1.7.2 使用 --filter 选项筛选数据

Dumpling 可以通过 --filter 指定table-filter 来筛选特定的库表。table-filter 的语法与 .gitignore 相似，详细语法参考表库过滤。

```
./dumpling -u root -P 4000 -h 127.0.0.1 -o /tmp/test -r 200000 --filter "employees.*" --filter
    ′→ "*.WorkOrder"
```

上述命令将会导出 employees 数据库的所有表，以及所有数据库中的 WorkOrder 表。

10.8.1.7.3 使用 -B 或 -T 选项筛选数据

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

> 注意：
>
> - --filter 选项与 -T 选项不可同时使用。
> - -T 选项只能接受完整的 库名.表名 形式，不支持只指定表名。例：Dumpling 无法识别 -T WorkOrder。

例如通过指定：

- -B employees 导出 employees 数据库
- -T employees.WorkOrder 导出 employees.WorkOrder 数据表

10.8.1.8 通过并发提高 Dumpling 的导出效率

默认情况下，导出的文件会存储到 ./export-<current local time> 目录下。常用选项如下：

- -t 用于指定导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜设置过大。
- -r 选项用于指定单个文件的最大记录数，或者说，数据库中的行数。开启后 Dumpling 会开启表内并发，提高导出大表的速度。当上游为TiDB 且版本为 v3.0 或更新版本时，设置 -r 参数大于 0 表示使用TiDB region 信息划分表内并发，具体取值不影响划分算法。对上游为 MySQL 且表的主键是int 的场景，该参数也有表内并发效果。
- --compress <format> 选项可以用于压缩导出的数据，支持 gzip、snappy、zstd 压缩算法。压缩可以显著降低导出数据的大小，同时如果存储的写入I/O 带宽不足，可以使用该选项来加速导出。但该选项也有副作用，由于该选项会对每个文件单独压缩，因此会增加 CPU 消耗。

利用以上选项可以提高 Dumpling 的导出速度。

10.8.1.9 调整 Dumpling 的数据一致性选项

> 注意：
>
> 数据一致性选项的默认值为 auto。在大多数场景下，你不需要调整该选项。

Dumpling 通过 --consistency <consistency level> 标志控制导出数据“一致性保证”的方式。在使用snapshot 来保证一致性的时候，可以使用 --snapshot 选项指定要备份的时间戳。还可以使用以下的一致性级别：

- `flush`：使用 `FLUSH TABLES WITH READ LOCK` 短暂地中断备份库的 DML 和 DDL 操作、保证备份连接的全局一致性和记录 POS 信息。所有的备份连接启动事务后释放该锁。推荐在业务低峰或者 MySQL 备份库上进行全量备份。
- `snapshot`：获取指定时间戳的一致性快照并导出。
- `lock`：为待导出的所有表上读锁。
- `none`：不做任何一致性保证。
- `auto`：对 MySQL 使用 `flush`，对 TiDB 使用 `snapshot`。

操作完成之后，你可以在 `/tmp/test` 查看导出的文件：

```
$ ls -lh /tmp/test | awk '{print $5 "\t" $9}'

140B  metadata
66B   test-schema-create.sql
300B  test.sbtest1-schema.sql
190K  test.sbtest1.0.sql
300B  test.sbtest2-schema.sql
190K  test.sbtest2.0.sql
300B  test.sbtest3-schema.sql
190K  test.sbtest3.0.sql
```

10.8.1.10 导出TiDB 的历史数据快照

Dumping 可以通过 `--snapshot` 指定导出某个tidb_snapshot 的数据。

`--snapshot` 选项可设为TSO（`SHOW MASTER STATUS` 输出的 `Position` 字段）或有效的 `datetime` 时间（`YYYY-MM-DD hh:mm:ss` 形式），例如：

```
./dumpling --snapshot 417773951312461825
./dumpling --snapshot "2020-07-02 17:12:45"
```

即可导出TSO 为 `417773951312461825` 或 `2020-07-02 17:12:45` 时的TiDB 历史数据快照。

10.8.1.11 控制导出TiDB 大表（超过 1 TB）时的内存使用

Dumping 导出TiDB 较大单表（超过 1 TB）时，可能会因为导出数据过大导致 TiDB 内存溢出(OOM)，从而使连接中断导出失败。可以通过以下参数减少TiDB 的内存使用。

- 设置 `-r` 参数，可以划分导出数据区块减少TiDB 扫描数据的内存开销，同时也可开启表内并发提高导出效率。当上游为 TiDB 且版本为 v3.0 或更新版本时，设置 `-r` 参数大于 0 表示使用TiDB region 信息划分表内并发，具体取值不影响划分算法。
- 调小 `--tidb-mem-quota-query` 参数到 8589934592 (8GB) 或更小。可控制TiDB 单条查询语句的内存使用。
- 调整`--params "tidb_distsql_scan_concurrency=5"` 参数，即设置导出时的session 变量tidb_distsql_scan_concurrency 从而减少TiDB scan 操作的并发度。

10.8.1.12 手动设置TiDB GC时间

如果导出的TiDB 版本为 v4.0.0 或更新版本，并且 Dumpling 可以访问TiDB 集群的 PD 地址，Dumpling 会自动配置延长 GC 时间且不会对原集群造成影响。

但是，在以下场景中，Dumpling 无法自动调整GC 时间：

- 数据量非常大（超过 1 TB）。
- Dumpling 无法直接连接到 PD，例如 TiDB 集群运行在TiDB Cloud 上，或者TiDB 集群运行在 Kubernetes 上且与 Dumpling 分离。

在这些场景中，你必须提前手动调长GC 时间，以避免因为导出过程中发生GC 导致导出失败。

使用以下 SQL 语句手动调整GC 时间：

```
SET GLOBAL tidb_gc_life_time = '720h';
```

在 Dumpling 退出后，无论导出是否成功，都必须将GC 时间恢复为其原始值（默认值为 `10m`）。

```
SET GLOBAL tidb_gc_life_time = '10m';
```

10.8.2 Dumpling 主要选项表

主要选项	用途	默认值
-V 或 –version	输出 Dumpling 版本并直接退出	
-B 或 –database	导出指定数据库	
-T 或 –tables-list	导出指定数据表	

主要选项	用途	默认值
-f 或 –filter	导出能匹配模式的表，语法可参考table-filter	[*.*,!/^(′→ mysql ′→ | ′→ sys ′→ | ′→ INFORMATION_SCHEMA ′→ | ′→ PERFORMANCE_SCHEMA ′→ | ′→ METRICS_SCHEMA ′→ | ′→ INSPECTION_SCHEMA ′→) ′→ $ ′→ /.*] ′→ (导出除系统库外的所有库表)
–case-sensitive	table-filter 是否大小写敏感	false，大小写不敏感
-h 或 –host	连接的数据库主机的地址	“127.0.0.1”
-t 或 –threads	备份并发线程数	4

主要选项	用途	默认值
-r 或 –rows	将 table 划分成 row 行数据，一般针对大表操作并发生成多个文件。当上游为 TiDB 且版本为 v3.0 或更新版本时，设置 -r 参数大于 0 表示使用 TiDB region 信息划分表内并发，具体取值不影响划分算法。	

主要选项	用途	默认值
-L 或 –logfile	日志输出地址，为空时会输出到控制台	“”
–loglevel	日志级别 {debug,info,warn,error,dpanic,panic,fatal}	“info”
–logfmt	日志输出格式 {text,json}	“text”
-d 或 –no-data	不导出数据，适用于只导出 schema 场景	
–no-header	导出 csv 格式的 table 数据，不生成 header	
-W 或 –no-views	不导出 view	true
-m 或 –no-schemas	不导出 schema，只导出数据	

主要选项	用途	默认值
-s 或 –statement-size	控制 INSERT ′→ SQL 语句的大小，单位 bytes	
-F 或 –filesize	将 table 数据划分出来的文件大小，需指明单位 (如 128B, 64 ′→ KiB ′→ , 32 ′→ MiB ′→ , 1.5 ′→ GiB ′→)	
–filetype	导出文件类型 (csv/sql)	“sql”

主要选项	用途	默认值
-o 或 –output	导出本地文件路径或外部存储 URI 格式	“./export-${time}”
-S 或 –sql	根据指定的 sql 导出数据，该选项不支持并发导出	

主要选项	用途	默认值
–consistency	flush: dump 前用 FTWRL snapshot: 通过 TSO 来指定 dump 某个快照时间点的 TiDB 数据 lock: 对需要 dump 的所有表执行 lock ′→ ′→ tables ′→ ′→ read ′→ 命令 none: 不加锁 dump，无法保证一致性 auto: 对 MySQL 使用 –139consistency flush；对 TiDB	“auto”

主要选项	用途	默认值
–snapshot	snapshot tso，只在 consistency=snapshot 下生效	
–where	对备份的数据表通过 where 条件指定范围	
-p 或 –password	连接的数据库主机的密码	
-P 或 –port	连接的数据库主机的端口	4000
-u 或 –user	连接的数据库主机的用户名	“root”
–dump-empty-database	导出空数据库的建库语句	true

主要选项	用途	默认值
–ca	用于 TLS 连接的 certifi-cate au-thor-ity 文件的地址	
–cert	用于 TLS 连接的 client certifi-cate 文件的地址	
–key	用于 TLS 连接的 client pri-vate key 文件的地址	
–csv-delimiter	csv 文件中字符类型变量的定界符	’ “’

主要选项	用途	默认值
–csv-separator	csv 文件中各值的分隔符，如果数据中可能有逗号，建议源文件导出时分隔符使用非常见组合字符	‘,’
–csv-null-value	csv 文件空值的表示	“\N”
–escape-backslash	使用反斜杠 (\) 来转义导出文件中的特殊字符	true

主要选项	用途	默认值
–output-filename-template	以 golang tem-plate 格式表示的数据文件名格式支持 {{. ′→ DB ′→ }} ′→ 、 {{. ′→ Table ′→ }} ′→ 、 {{. ′→ Index ′→ }} ′→ 三个参数分别表示数据文件的库名、表名、分块 ID	‘{{.DB}}.{{.Table}}.{{.Index}}’

主要选项	用途	默认值
–status-addr	Dumpling 的服务地址，包含了 Prometheus 拉取 metrics 信息及 pprof 调试的地址	“:8281”

主要选项	用途	默认值
–tidb-mem-quota-query	单条 dumpling 命令导出 SQL 语句的内存限制，单位为 byte。对于 v4.0.10 或以上版本，若不设置该参数，默认使用 TiDB 中的 mem- '→ quota '→ - '→ query '→ 配置项值作为内存限制值。对于 v4.0.10 以下版本，该参数值默认为 32	34359738368

主要选项	用途	默认值
–params	为需导出的数据库连接指定 ses-sion 变量，可接受的格式: “char-ac-ter_set_client=latin1,character_set_connection=latin1”	
-c 或 –compress	压缩 Dumpling 导出的 CSV、SQL 数据与表结构文件为指定格式，支持 “gzip”、“snappy” 和 “zstd” 压缩算法	“”

10.9 TiCDC

10.9.1 TiCDC 简介

TiCDC 是一款TiDB 增量数据同步工具，通过拉取上游TiKV 的数据变更日志，TiCDC 可以将数据解析为有序的行级变更数据输出到下游。

10.9.1.1 TiCDC 适用场景

- 提供多TiDB 集群，跨区域数据高可用和容灾方案，保证在灾难发生时保证主备集群数据的最终一致性。
- 提供同步实时变更数据到异构系统的服务，为监控、缓存、全文索引、数据分析、异构数据库使用等场景提供数据源。

10.9.1.2 TiCDC 主要特性

10.9.1.2.1 核心能力

- 提供TiDB -> TiDB 之间数据容灾复制的能力，实现秒级别RPO 和分钟级别RTO
- 提供TiDB 之间双向复制的能力，支持通过TiCDC 构建多写多活的TiDB 集群
- 提供TiDB -> MySQL（或其他兼容 MySQL 协议的数据库）的低延迟的增量数据同步能力
- 提供TiDB -> Kafka 增量数据同步能力，推荐的数据格式包含Canal-JSON，Avro 等
- 提供表级别数据同步能力，支持同步过程中过滤数据库、表、DML、DDL 的能力
- 高可用架构，无单点故障；支持动态添加、删除TiCDC 节点
- 支持通过Open API 进行集群管理，包括查询任务状态；动态修改任务配置；动态创建、删除任务等

10.9.1.2.2 数据同步顺序性

- TiCDC 对于所有的 DDL/DML 都能对外输出至少一次。
- TiCDC 在TiKV/TiCDC 集群故障期间可能会重复发相同的 DDL/DML。对于重复的 DDL/DML：
  - MySQL sink 可以重复执行 DDL，对于在下游可重入的 DDL（譬如truncate table）直接执行成功；对于在下游不可重入的 DDL（譬如 create table），执行失败，TiCDC 会忽略错误继续同步。
  - Kafka sink
    * Kafka sink 提供不同的数据分发策略，可以按照表、主键或 ts 等策略分发数据到不同 Kafka partition。使用表、主键分发策略，可以保证某一行的更新数据被顺序的发送到相同partition。
    * 对所有的分发策略，我们都会定期发送 Resolved TS 消息到所有的 topic/partition，表示早于该 Resolved TS 的消息都已经发送到topic/partition，消费程序可以利用Resolved TS 对多个topic/partition 的消息进行排序。
    * Kafka sink 会发送重复的消息，但重复消息不会破坏Resolved TS 的约束，比如在changefeed 暂停重启后，可能会按顺序发送msg1、msg2、msg3、msg2、msg3。你可以在 Kafka 消费端进行过滤。

10.9.1.2.3 数据同步一致性

- MySQL sink
  - TiCDC 开启redo log 后保证数据复制的最终一致性
  - TiCDC 保证单行的更新与上游更新顺序一致。
  - TiCDC 不保证下游事务的执行顺序和上游完全一致。

> 注意：
>
> 从 v6.2 版本起，你可以通过配置 sink uri 参数`transaction-atomicity`来控制TiCDC 是否拆分单表事务。拆分事务可以大幅降低 MySQL sink 同步大事务的延时和内存消耗。

10.9.1.3 TiCDC 架构

TiCDC 作为TiDB 的增量数据同步工具，通过 PD 内部的 etcd 实现高可用，通过多个TiCDC 进程获取TiKV 节点上的数据改变，在内部进行排序、合并等处理之后，通过多个同步任务(Changefeed)，同时向多个下游系统进行数据同步。

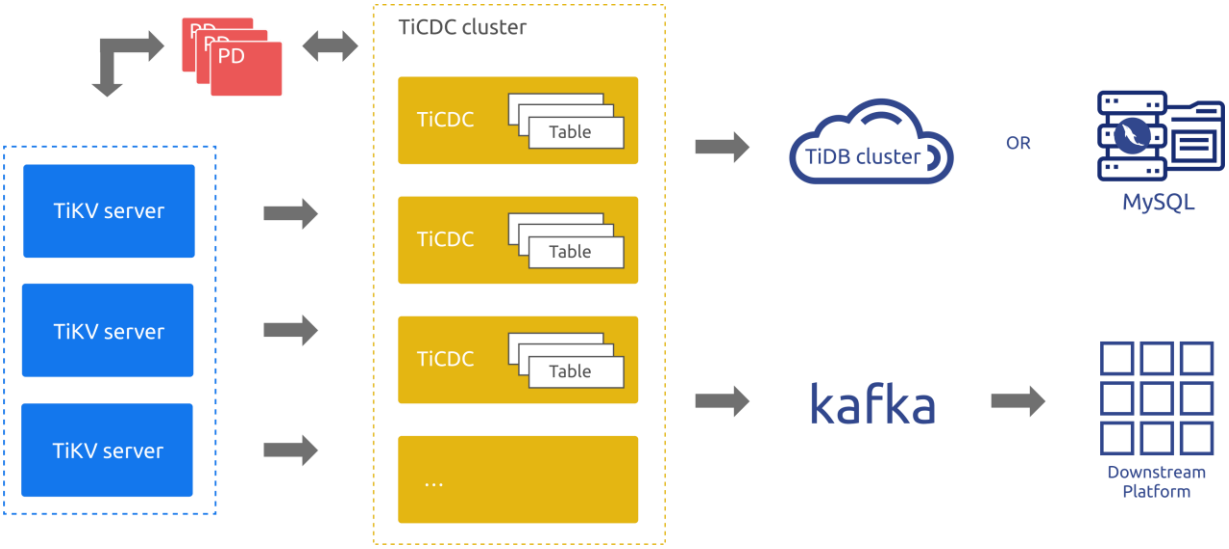


图 201: TiCDC architecture

在以上架构图中：

- TiKV Server：代表 TiDB 集群中的TiKV 节点，当数据发生改变时TiKV 节点会主动将发生的数据改变以变更日志（KV change logs，简称change logs）的方式发送给TiCDC 节点。当然，当TiCDC 节点发现收到的change logs 并不是连续的，也会主动发起请求，获得需要的change logs。
- TiCDC：代表运行了运行TiCDC 进程的各个节点。每个节点都运行一个TiCDC 进程，每个进程会从TiKV 节点中拉取一个或者多个表中的数据改变，并通过 Sink 模块同步到下游系统。
- PD：代表 TiDB 集群中的调度模块，负责集群数据的事实调度，这个模块通常是由 3 个 PD 节点构成的，内部通过 etcd 集群来实现选举等高可用相关的能力。TiCDC 集群使用了 PD 集群内置的 etcd 集群来保存自己的元数据信息，例如：节点的状态信息，changefeed 配置信息等。

另外，从上面的架构图中也可以看到，目前TiCDC 支持将数据同步到TiDB，MySQL 数据库，以及 Kafka 等。

10.9.1.4 最佳实践

- 使用TiCDC 在两个TiDB 集群间同步数据时，当上下游的延迟超过 100 ms 时，推荐将 TiCDC 部署在下游TiDB 集群所在的区域(IDC, region)。
- TiCDC 同步的表需要至少存在一个有效索引的表，有效索引的定义如下：
  - 主键(PRIMARY KEY) 为有效索引。
  - 唯一索引 (UNIQUE INDEX) 中每一列在表结构中明确定义非空 (NOT NULL) 且不存在虚拟生成列 (VIRTUAL GENERATED COLUMNS)。
- 容灾场景下使用TiCDC 需要配置redo log 实现最终一致性。
- 使用 TiCDC 同步单行较大 (> 1k) 的宽表时，推荐设置 TiCDC 参数per-table-memory-quota，使得 `per-table-memory-quota = ticdcTotalMemory / (tableCount * 2)`。`ticdcTotalMemory` 是一个 TiCDC 节点的内存，`tableCount` 是一个TiCDC 节点同步的目标表的数量。

> **注意：**
>
> 从 TiCDC v4.0.8 版本开始，可通过修改任务配置来同步没有有效索引的表，但在数据一致性的保证上有所减弱。具体使用方法和注意事项参考同步没有有效索引的表。

10.9.1.4.1 暂不支持的场景

目前TiCDC 暂不支持的场景如下：

- 暂不支持单独使用RawKV 的 TiKV 集群。
- 暂不支持在TiDB 中创建 SEQUENCE 的 DDL 操作和SEQUENCE 函数。在上游TiDB 使用 SEQUENCE 时，TiCDC 将会忽略掉上游执行的 SEQUENCE DDL 操作/函数，但是使用 SEQUENCE 函数的 DML 操作可以正确地同步。

10.9.2 TiCDC 安装部署与集群运维

本文档介绍部署TiCDC 集群的软硬件环境要求，如何安装部署TiCDC 集群，以及如何对TiCDC 集群进行运维操作。你可以选择在安装TiDB 集群的同时部署TiCDC，也可以对原有TiDB 集群新增TiCDC 组件。

10.9.2.1 软件和硬件环境推荐配置

在生产环境中，TiCDC 的软件和硬件配置推荐如下：

Linux 操作系统	版本
Red Hat Enterprise Linux	7.3 及以上
CentOS	7.3 及以上

CPU	内存	硬盘类型	网络	TiCDC 集群实例数量（生产环境最低要求）
16 核 +	64 GB+	SSD	万兆网卡（2 块最佳）	2

更多信息参见TiDB 软件和硬件环境建议配置。

10.9.2.2 使用TiUP 部署包含TiCDC 组件的全新TiDB 集群

在使用TiUP 部署全新TiDB 集群时，支持同时部署TiCDC 组件。你需要在TiUP 启动TiDB 集群时的配置文件中加入 TiCDC 相关的部分，以下是一个示例：

```
cdc_servers:
  - host: 10.0.1.20
    gc-ttl: 86400
    data_dir: "/cdc-data"
  - host: 10.0.1.21
    gc-ttl: 86400
    data_dir: "/cdc-data"
```

更多参考：

- 详细配置参数，请参考编辑初始化配置文件。
- 具体可配置字段，请参考通过TiUP 配置 cdc_servers。
- 部署集群的具体步骤，请参考使用TiUP 部署TiDB 集群。

> 注意：
>
> 在安装之前，请确认TiUP 中控机与TiCDC 目标主机的SSH 互信及 sudo 免密已经完成配置。

10.9.2.3 使用TiUP 在原有TiDB 集群上新增或扩容TiCDC 组件

扩容的方式与部署TiCDC 集群的方式类似，推荐使用TiUP 工具完成。

1. 编写一个名为 scale-out.yml 的配置文件，包含需要扩容的节点的配置信息。下面是一个示例：

```
cdc_servers:
  - host: 10.1.1.1
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
  - host: 10.1.1.2
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
  - host: 10.0.1.4:8300
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
```

2. 在TiUP 中控机上执行类似下面的命令进行扩容：

```
tiup cluster scale-out <cluster-name> scale-out.yml
```

更多用例说明，请参考扩容TiCDC 节点。

10.9.2.4 使用TiUP 在原有TiDB 集群上移除或缩容TiCDC 组件

推荐使用TiUP 完成对TiCDC 集群节点的缩容。使用类似下面的命令完成缩容:

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:8300
```

更多用例说明，请参考缩容TiCDC 节点。

10.9.2.5 使用TiUP 升级TiCDC 集群

TiUP 支持升级TiDB 集群，包括TiCDC 组件。执行升级指令时，TiUP 会自动升级TiCDC 组件，无需额外操作。操作示例如下：

```
tiup update --self && \
tiup update --all && \
tiup cluster upgrade <cluster-name> <version> --transfer-timeout 600
```

> 注意：
>
> 命令中的 `<cluster-name>` 需要替换为集群名字，`<version>` 需要替换为目标版本号，例如 v7.0.0。

10.9.2.5.1 升级的注意事项

升级TiCDC 集群时，需要注意以下事项：

- TiCDC v4.0.2 对 changefeed 的配置做了调整，请参阅配置文件兼容注意事项。
- 升级期间遇到的问题及其解决办法，请参阅使用TiUP 升级TiDB。
- TiCDC 自 v6.3.0 起支持滚动升级，使用TiUP 升级TiCDC 节点期间，能够保证同步延迟稳定，不发生剧烈波动。满足以下条件将自动启用滚动升级：
  - TiCDC 版本大于等于 v6.3.0。
  - TiUP 版本大于等于 v1.11.3。
  - 集群中至少有两个正在运行的TiCDC 实例。

10.9.2.6 使用TiUP变更TiCDC集群配置

本节介绍如何使用TiUP的`tiup cluster edit-config`命令来修改TiCDC的配置。在以下例子中，假设需要把TiCDC的`gc-ttl`从默认值`86400`修改为`172800`，即48小时。

1. 执行`tiup cluster edit-config`命令，注意将`<cluster-name>`替换成实际的集群名：

```
tiup cluster edit-config <cluster-name>
```

2. 在vi编辑器页面，修改`server-configs`下的`cdc`配置：

```
server_configs:
  tidb: {}
  tikv: {}
  pd: {}
  tiflash: {}
  tiflash-learner: {}
  pump: {}
  drainer: {}
  cdc:
    gc-ttl: 172800
```

   以上把TiCDC的`gc-ttl`的值设置为48小时。

3. 执行`tiup cluster reload -R cdc`命令重新加载配置。

10.9.2.7 使用TiUP终止和启动TiCDC节点

使用TiUP可以方便地终止和启动TiCDC节点，命令如下：

- 终止TiCDC节点：`tiup cluster stop -R cdc`
- 启动TiCDC节点：`tiup cluster start -R cdc`
- 重启TiCDC节点：`tiup cluster restart -R cdc`

10.9.2.8 使用加密传输(TLS)功能

请参阅为TiDB组件间通信开启加密传输。

10.9.2.9 使用TiCDC命令行工具来查看集群状态

执行以下命令来查看TiCDC集群运行状态，注意需要将`v<CLUSTER_VERSION>`替换为TiCDC集群版本，例如`v6.5.0`：

```
tiup ctl:v<CLUSTER_VERSION> cdc capture list --server=http://10.0.10.25:8300
```

```
[
  {
    "id": "806e3a1b-0e31-477f-9dd6-f3f2c570abdd",
```

```
    "is-owner": true,
    "address": "127.0.0.1:8300",
    "cluster-id": "default"
  },
  {
    "id": "ea2a4203-56fe-43a6-b442-7b295f458ebc",
    "is-owner": false,
    "address": "127.0.0.1:8301",
    "cluster-id": "default"
  }
]
```

- `id`：表示服务进程的 ID。
- `is-owner`：表示该服务进程是否为 owner 节点。
- `address`：该服务进程对外提供接口的地址。
- `cluster-id`：该 TiCDC 的集群ID，默认值为 `default`。

10.9.3 Changefeed

10.9.3.1 Changefeed 概述

Changefeed 是 TiCDC 中的单个同步任务。Changefeed 将一个TiDB 集群中数张表的变更数据输出到一个指定的下游。TiCDC 集群可以运行和管理多个 Changefeed。

10.9.3.1.1 Changefeed 状态流转

同步任务状态标识了同步任务的运行情况。在TiCDC 运行过程中，同步任务可能会运行出错、手动暂停、恢复，或达到指定的 `TargetTs`，这些行为都可以导致同步任务状态发生变化。本节描述 TiCDC 同步任务的各状态以及状态之间的流转关系。

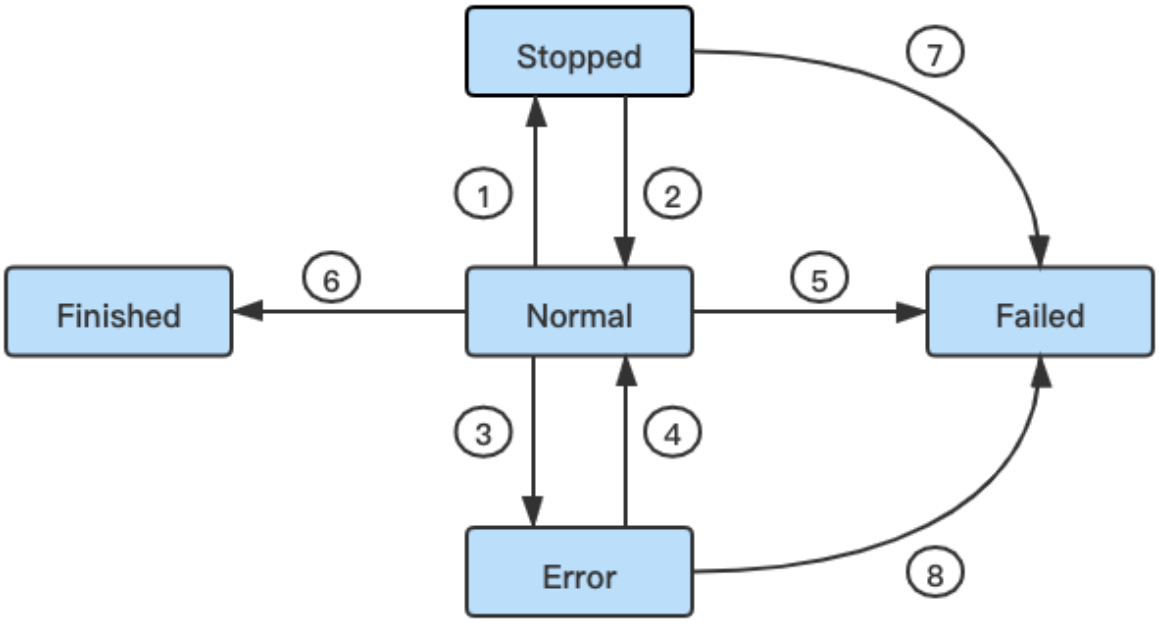


图 202: TiCDC state transfer

以上状态流转图中的状态说明如下：

- Normal：同步任务正常进行，checkpoint-ts 正常推进。
- Stopped：同步任务停止，由于用户手动暂停 (pause) changefeed。处于这个状态的changefeed 会阻挡GC 推进。
- Error：同步任务报错，由于某些可恢复的错误导致同步无法继续进行，处于这个状态的changefeed 会不断尝试继续推进，直到状态转为 Normal。处于这个状态的 changefeed 会阻挡GC 推进。
- Finished：同步任务完成，同步任务进度已经达到预设的 TargetTs。处于这个状态的 changefeed 不会阻挡 GC 推进。
- Failed：同步任务失败。由于发生了某些不可恢复的错误，导致同步无法继续进行，并且无法恢复。处于这个状态的changefeed 不会阻挡GC 推进。

以上状态流转图中的编号说明如下：

- ① 执行 `changefeed pause` 命令。
- ② 执行 `changefeed resume` 恢复同步任务。
- ③ changefeed 运行过程中发生可恢复的错误，自动进行恢复。
- ④ 执行 `changefeed resume` 恢复同步任务。
- ⑤ changefeed 运行过程中发生不可恢复的错误。
- ⑥ changefeed 已经进行到预设的 TargetTs，同步自动停止。
- ⑦ changefeed 停滞时间超过 `gc-ttl` 所指定的时长，不可被恢复。
- ⑧ changefeed 尝试自动恢复过程中发生不可恢复的错误。

10.9.3.1.2 操作 Changefeed

通过 TiCDC 提供的命令行工具 `cdc cli`，你可以管理 TiCDC 集群和同步任务，具体可参考管理 TiCDC Changefeed。你也可以通过 HTTP 接口，即 TiCDC OpenAPI 来管理 TiCDC 集群和同步任务，详见 TiCDC OpenAPI。

如果你使用的 TiCDC 是用 TiUP 部署的，可以通过 `tiup ctl:v<CLUSTER_VERSION> cdc` 来使用 TiCDC 命令行工具，注意需要将 `<version>` 替换为 TiCDC 集群版本。你也可以通过直接执行 `cdc cli` 直接使用命令行工具。

10.9.3.2 创建 Changefeed

10.9.3.2.1 同步数据到 MySQL 兼容数据库

本文介绍如何使用 TiCDC 创建一个将增量数据复制到下游 TiDB 数据库，或其他兼容 MySQL 协议数据库的 Changefeed。同时介绍了如何使用 TiCDC 灾难场景的最终一致性复制功能。

创建同步任务，复制增量数据到 MySQL 兼容数据库

使用以下命令来创建同步任务：

```
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"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"sink-uri":"mysql://root:123456@127.0.0.1:3306/","opts":{},"create-time":"2020-03-12T22
    ′→ :04:08.103600025+08:00","start-ts":415241823337054209,"target-ts":0,"admin-job-type":0,"
    ′→ sort-engine":"unified","sort-dir":".","config":{"case-sensitive":true,"filter":{"rules
    ′→ ":["*.*"],"ignore-txn-start-ts":null,"ddl-allow-list":null},"mounter":{"worker-num":16},"
    ′→ sink":{"dispatchers":null},"scheduler":{"type":"table-number","polling-time":-1}},"state
    ′→ ":"normal","history":null,"error":null}
```

- `--server`：TiCDC 集群中任意一个 TiCDC 服务器的地址。
- `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该 ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为 ID。
- `--sink-uri`：同步任务下游的地址，详见 Sink URI 配置 `mysql/tidb`。
- `--start-ts`：指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标 TSO。TiCDC 集群拉取数据直到这个 TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件，详见：TiCDC Changefeed 配置参数。

Sink URI 配置 `mysql/tidb`

Sink URI 用于指定 TiCDC 目标系统的连接信息，遵循以下格式：

```
[scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
```

> **注意：**
>
> `/path` 不适用于 MySQL sink。

一个通用的配置样例如下所示：

```
--sink-uri="mysql://root:123456@127.0.0.1:3306"
```

URI 中可配置的参数如下：

参数	描述
`root`	下游数据库的用户名。
`123456`	下游数据库密码。（可采用 Base64 进行编码）
`127.0.0.1`	下游数据库的IP。
`3306`	下游数据的连接端口。
`worker-count`	向下游执行 SQL 的并发度（可选，默认值为 16）。
`cache-prep-stmts`	向下游执行 SQL 时是否使用prepared statement 并且开启客户端的 prepared statement 缓存（可选，默认值为 `true`）。
`max-txn-row`	向下游执行 SQL 的batch 大小（可选，默认值为 256）。
`ssl-ca`	连接下游 MySQL 实例所需的 CA 证书文件路径（可选）。
`ssl-cert`	连接下游 MySQL 实例所需的证书文件路径（可选）。
`ssl-key`	连接下游 MySQL 实例所需的证书密钥文件路径（可选）。
`time-zone`	连接下游 MySQL 实例时使用的时区名称，从 v4.0.8 开始生效（可选。如果不指定该参数，使用 TiCDC 服务进程的时区；如果指定该参数但使用空值，则表示连接 MySQL 时不指定时区，使用下游默认时区）。
`transaction-atomicity`	指定事务的原子性级别（可选，默认值为 none）。当该值为 `table` 时 TiCDC 保证单表事务的原子性，当该值为 none 时 TiCDC 会拆分单表事务。

若需要对 Sink URI 中的数据库密码使用 Base64 进行编码，可以参考如下命令：

```
echo -n '123456' | base64    # 假设待编码的密码为 123456
```

编码后的密码如下：

```
MTIzNDU2
```

> **注意：**
>
> 当 Sink URI 中包含特殊字符时，如`! * ' ( ) ; : @ & = + $ , / ? % # [ ]`，需要对 URI 特殊字符进行转义处理。你可以使用 URI Encoder 工具对 URI 进行转义。

灾难场景的最终一致性复制

从 v6.1.1 版本开始，容灾场景下的最终一致性复制功能GA。从 v5.3.0 开始，TiCDC 支持将上游TiDB 的增量数据备份到下游集群的对象存储或 NFS 文件系统。当上游集群出现了灾难，完全无法使用时，TiCDC 可以将下游集群恢复到最近的一致状态，即提供灾备场景的最终一致性复制能力，确保应用可以快速切换到下游集群，避免数据库长时间不可用，提高业务连续性。

目前，TiCDC 支持将 TiDB 集群的增量数据复制到 TiDB 或兼容 MySQL 的数据库系统（包括 Aurora、MySQL 和 MariaDB）。如果 TiCDC 在上游发生灾难前正常运行，且上游TiDB 集群没有出现数据复制延迟大幅度增加的情况，灾难发生后，下游集群可以在 5 分钟之内恢复集群，并且最多丢失出现问题前 10 秒钟的数据，即RTO <= 5 min，P95 RPO <= 10s。

当上游TiDB 集群出现以下情况时，会导致TiCDC 延迟上升，进而影响RPO：

- TPS 短时间内大幅度上升
- 上游出现大事务或者长事务
- Reload 或 Upgrade 上游TiKV 集群或TiCDC 集群
- 执行耗时很长的 DDL 语句，例如：add index
- 使用过于激进的 PD 调度策略，导致频繁region leader 迁移或region merge/split

> 注意：
>
> TiCDC 最终一致性功能从 v6.1.1 开始支持兼容Amazon S3 协议的对象存储，从 v6.1.4 开始支持兼容 GCS 和 Azure 协议的对象存储。

使用前提

- 准备高可用的对象存储或 NFS 系统，用于存储TiCDC 的实时增量数据备份文件，在上游发生灾难情况下，该文件存储可以访问。
- TiCDC 对需要具备灾难场景最终一致性的changefeed 开启该功能，开启方式是在changefeed 配置文件中增加以下配置：

```
[consistent]
##### 一致性级别，选项有：
##### - none： 默认值，非灾难场景，只有在任务指定 finished-ts 情况下保证最终一致性。
##### - eventual： 使用 redo log，提供上游灾难情况下的最终一致性。
level = "eventual"

##### 单个 redo log 文件大小，单位 MiB，默认值 64，建议该值不超过 128。
max-log-size = 64

##### 刷新或上传 redo log 至 S3 的间隔，单位毫秒，建议该参数 >= 2000。
flush-interval = 2000
```

```
##### redo log 备份文件的地址，支持的 scheme 包括 nfs（NFS 目录）和 Amazon S3、GCS 和 Azure（
    ↪ 上传至对象存储）。
storage = "$SCHEME://logbucket/test-changefeed?endpoint=http://$ENDPOINT/"
```

灾难恢复

当上游发生灾难后，需要通过 cdc redo 命令在下游手动恢复。恢复流程如下：

1. 确保TiCDC 进程已经退出，防止在数据恢复过程中上游恢复服务，TiCDC 重新开始同步数据。
2. 使用cdc binary 进行数据恢复，具体命令如下：

```
cdc redo apply --tmp-dir="/tmp/cdc/redo/apply" \
    --storage="s3://logbucket/test-changefeed?endpoint=http://10.0.10.25:24927/"  \
    --sink-uri="mysql://normal:123456@10.0.10.55:3306/"
```

以上命令中：

- tmp-dir：指定用于下载 TiCDC 增量数据备份文件的临时目录。
- storage：指定存储 TiCDC 增量数据备份文件的地址，为对象存储 URI 或者 NFS 目录。
- sink-uri：数据恢复的目标地址。scheme 仅支持 mysql。

10.9.3.2.2 同步数据到 Kafka

本文介绍如何使用TiCDC 创建一个将增量数据复制到 Kafka 的 Changefeed。

创建同步任务，复制增量数据 Kafka

使用以下命令来创建同步任务：

```
cdc cli changefeed create \
    --server=http://10.0.10.25:8300 \
    --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=canal-json&kafka-version=2.4.0&
        ↪ partition-num=6&max-message-bytes=67108864&replication-factor=1" \
    --changefeed-id="simple-replication-task"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"sink-uri":"kafka://127.0.0.1:9092/topic-name?protocol=canal-json&kafka-version=2.4.0&
    ↪ partition-num=6&max-message-bytes=67108864&replication-factor=1","opts":{},"create-time
    ↪ ":"2020-03-12T22:04:08.103600025+08:00","start-ts":415241823337054209,"target-ts":0,"
    ↪  admin-job-type":0,"sort-engine":"unified","sort-dir":".","config":{"case-sensitive":true
    ↪  ,"filter":{"rules":["*.*"],"ignore-txn-start-ts":null,"ddl-allow-list":null},"mounter":{"
    ↪  worker-num":16},"sink":{"dispatchers":null},"scheduler":{"type":"table-number","polling-
    ↪ time":-1}},"state":"normal","history":null,"error":null}
```

- --server：TiCDC 集群中任意一个TiCDC 服务器的地址。

- `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为ID。
- `--sink-uri`：同步任务下游的地址，详见：Sink URI 配置 Kafka。
- `--start-ts`：指定 changefeed 的开始TSO。TiCDC 集群将从这个TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标TSO。TiCDC 集群拉取数据直到这个TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件，详见：TiCDC Changefeed 配置参数。

Sink URI 配置 kafka

Sink URI 用于指定TiCDC 目标系统的连接信息，遵循以下格式：

```
[scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
```

一个通用的配置样例如下所示：

```
--sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=canal-json&kafka-version=2.4.0&partition-
    ↪ num=6&max-message-bytes=67108864&replication-factor=1"
```

URI 中可配置的的参数如下：

参数	描述
`127.0.0.1`	下游 Kafka 对外提供服务的IP。
`9092`	下游 Kafka 的连接端口。
`topic-name`	变量，使用的 Kafka topic 名字。
`kafka-version`	下游 Kafka 版本号（可选，默认值 `2.4.0`，目前支持的最低版本为 `0.11.0.2`，最高版本为 `3.2.0`。该值需要与下游 Kafka 的实际版本保持一致）。
`kafka-client-id`	指定同步任务的 Kafka 客户端的ID（可选，默认值为 `TiCDC_sarama_producer_`同步任务的 `ID`）。
`partition-num`	下游 Kafka partition 数量（可选，不能大于实际partition 数量，否则创建同步任务会失败，默认值 `3`）。
`max-message-bytes`	每次向 Kafka broker 发送消息的最大数据量（可选，默认值 `10MB`）。从 v5.0.6 和 v4.0.6 开始，默认值分别从 64MB 和 256MB 调整至 10 MB。
`replication-factor`	Kafka 消息保存副本数（可选，默认值 `1`）。
`required-acks`	在 `Produce` 请求中使用的配置项，用于告知broker 需要收到多少副本确认后才进行响应。可选值有：`0`（`NoResponse`：不发送任何响应，只有 `TCP ACK`），`1`（`WaitForLocal`：仅等待本地提交成功后再响应）和 `-1`（`WaitForAll`：等待所有同步副本提交后再响应。最小同步副本数量可通过broker 的 `min.insync.replicas` 配置项进行配置）。（可选，默认值为 `-1`）。
`compression`	设置发送消息时使用的压缩算法（可选值为 none、`lz4`、`gzip`、`snappy` 和 `zstd`，默认值为 none）。
`protocol`	输出到 Kafka 的消息协议，可选值有 `canal-json`、`open-protocol`、`canal`、`avro`、`maxwell`。
`auto-create-topic`	当传入的 `topic-name` 在 Kafka 集群不存在时，TiCDC 是否要自动创建该topic（可选，默认值 `true`）。

参数	描述
enable-tidb-extension	可选，默认值是 false。当输出协议为 canal-json 时，如果该值为 true，TiCDC 会发送WATERMARK 事件，并在 Kafka 消息中添加TiDB 扩展字段。从 6.1.0 开始，该参数也可以和输出协议 avro 一起使用。如果该值为 true，TiCDC 会在 Kafka 消息中添加三个 TiDB 扩展字段。
max-batch-size	从 v4.0.9 开始引入。当消息协议支持把多条变更记录输出至一条 Kafka 消息时，该参数用于指定这一条 Kafka 消息中变更记录的最多数量。目前，仅当 Kafka 消息的 protocol 为 open-protocol 时有效（可选，默认值 16）。
enable-tls	连接下游 Kafka 实例是否使用 TLS（可选，默认值 false）。
ca	连接下游 Kafka 实例所需的 CA 证书文件路径（可选）。
cert	连接下游 Kafka 实例所需的证书文件路径（可选）。
key	连接下游 Kafka 实例所需的证书密钥文件路径（可选）。
insecure-skip-verify	连接下游 Kafka 实例时是否跳过证书验证（可选，默认值 false）。
sasl-user	连接下游 Kafka 实例所需的 SASL/PLAIN 或 SASL/SCRAM 认证的用户名（authcid）（可选）。
sasl-password	连接下游 Kafka 实例所需的 SASL/PLAIN 或 SASL/SCRAM 认证的密码（可选）。如有特殊字符，需要用 URL encode 转义。
sasl-mechanism	连接下游 Kafka 实例所需的 SASL 认证方式的名称，可选值有 plain、scram-sha-256、scram-sha-512 和 gssapi。
sasl-gssapi-auth-type	gssapi 认证类型，可选值有 user 和 keytab（可选）。
sasl-gssapi-keytab-path	gssapi keytab 路径（可选）。
sasl-gssapi-kerberos-config-path	gssapi kerberos 配置路径（可选）。
sasl-gssapi-service-name	gssapi 服务名称（可选）。
sasl-gssapi-user	gssapi 认证使用的用户名（可选）。
sasl-gssapi-password	gssapi 认证使用的密码（可选）。如有特殊字符，需要用 URL encode 转义。
sasl-gssapi-realm	gssapi realm 名称（可选）。
sasl-gssapi-disable-pafxfast	gssapi 是否禁用 PA-FX-FAST（可选）。
dial-timeout	和下游 Kafka 建立连接的超时时长，默认值为 10s。
read-timeout	读取下游 Kafka 返回的 response 的超时时长，默认值为 10s。
write-timeout	向下游 Kafka 发送 request 的超时时长，默认值为 10s。
avro-decimal-handling-mode	仅在输出协议是 avro 时有效。该参数决定了如何处理 DECIMAL 类型的字段，值可以是 string 或 precise，表明映射成字符串还是浮点数。
avro-bigint-unsigned-handling-mode	仅在输出协议是 avro 时有效。该参数决定了如何处理 BIGINT UNSIGNED 类型的字段，值可以是 string 或 long，表明映射成字符串还是 64 位整形。

最佳实践

- TiCDC 推荐用户自行创建 Kafka Topic，你至少需要设置该 Topic 每次向 Kafka broker 发送消息的最大数据量和下游 Kafka partition 的数量。在创建changefeed 的时候，这两项设置分别对应 max-message-bytes 和 partition-num 参数。

- 如果你在创建 changefeed 时， 使用了尚未存在的 Topic， 那么 TiCDC 会尝试使用 `partition-num` 和 `replication-factor` 参数自行创建Topic。建议明确指定这两个参数。
- 在大多数情况下，建议使用 `canal-json` 协议。

> **注意：**
>
> 当 `protocol` 为 `open-protocol` 时，TiCDC 会尽量避免产生长度超过 `max-message-bytes` 的消息。但如果单条数据变更记录需要超过 `max-message-bytes` 个字节来表示，为了避免静默失败，TiCDC 会试图输出这条消息并在日志中输出Warning。

TiCDC 使用 Kafka 的认证与授权

使用 Kafka 的 SASL 认证时配置样例如下所示：

- SASL/PLAIN

```
shell  --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&sasl-user=alice-user&sasl
'→ -password=alice-secret&sasl-mechanism=plain"
```

- SASL/SCRAM

SCRAM-SHA-256、 SCRAM-SHA-512 与 PLAIN 方式类似，只需要将 `sasl-mechanism` 指定为对应的认证方式即可。

- SASL/GSSAPI

SASL/GSSAPI `user` 类型认证：

```
shell --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&sasl-mechanism=gssapi&
'→ sasl-gssapi-auth-type=user&sasl-gssapi-kerberos-config-path=/etc/krb5.conf&sasl-gssapi-service
'→ -name=kafka&sasl-gssapi-user=alice/for-kafka&sasl-gssapi-password=alice-secret&sasl-gssapi-
'→ realm=example.com"
```

`sasl-gssapi-user` 和 `sasl-gssapi-realm` 的值与 kerberos 中指定的principle 有关。例如，principle 为 `alice/for-
'→ kafka@example.com`，则 `sasl-gssapi-user` 和 `sasl-gssapi-realm` 的值应该分别指定为 `alice/for-kafka` 和 `example.com`。

SASL/GSSAPI `keytab` 类型认证：

```
shell --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&sasl-mechanism=gssapi
'→ &sasl-gssapi-auth-type=keytab&sasl-gssapi-kerberos-config-path=/etc/krb5.conf&sasl-gssapi-
'→ service-name=kafka&sasl-gssapi-user=alice/for-kafka&sasl-gssapi-keytab-path=/var/lib/secret/
'→ alice.key&sasl-gssapi-realm=example.com"
```

SASL/GSSAPI 认证方式详见 Configuring GSSAPI。

- TLS/SSL 加密

如果 Kafka broker 启用了 TLS/SSL 加密，则需要在 `--sink-uri` 中增加 `enable-tls=true` 参数值。如果需要使用自签名证书，则还需要在 `--sink-uri` 中指定 `ca`、`cert` 跟 `key` 几个参数。

- ACL 授权

TiCDC 能够正常工作所需的最小权限集合如下：

```
- 对 Topic [资源类型](https://docs.confluent.io/platform/current/kafka/authorization.html#
    ↪ resources)的 `Create` 和 `Write` 权限。
- 对 Cluster 资源类型的 `DescribeConfigs` 权限。
```

TiCDC 集成 Kafka Connect (Confluent Platform)

如要使用 Confluent 提供的 data connectors 向关系型或非关系型数据库传输数据，请选择 `avro` 协议，并在 `schema-registry` 中提供 Confluent Schema Registry 的 URL。

配置样例如下所示：

```
--sink-uri="kafka://127.0.0.1:9092/topic-name?&protocol=avro&replication-factor=3"  --schema-
    ↪ registry="http://127.0.0.1:8081" --config changefeed_config.toml
```

```
[sink]
dispatchers = [
 {matcher = ['*.*'], topic = "tidb_{schema}_{table}"},
]
```

集成具体步骤详见与 Confluent Cloud 进行数据集成。

自定义 Kafka Sink 的 Topic 和 Partition 的分发规则

Matcher 匹配规则

以上一节示例配置文件中的dispatchers 配置项为例：

- 对于匹配了matcher 规则的表，按照对应的topic 表达式指定的策略进行分发。例如表test3.aa，按照topic 表达式 2 分发；表test5.aa，按照topic 表达式 3 分发。
- 对于匹配了多个matcher 规则的表，以靠前的 matcher 对应的topic 表达式为准。例如表 test1.aa，按照 topic 表达式 1 分发。
- 对于没有匹配任何matcher 的表，将对应的数据变更事件发送到–sink-uri 中指定的默认topic 中。例如表 test10.aa 发送到默认topic。
- 对于匹配了matcher 规则但是没有指定topic 分发器的表，将对应的数据变更发送到–sink-uri 中指定的默认 topic 中。例如表test6.aa 发送到默认topic。

Topic 分发器

Topic 分发器用topic = “xxx” 来指定，并使用topic 表达式来实现灵活的topic 分发策略。topic 的总数建议小于 1000。

Topic 表达式的基本规则为 `[prefix]{schema}[middle][{table}][suffix]`，详细解释如下：

- `prefix`：可选项，代表 Topic Name 的前缀。
- `{schema}`：必选项，用于匹配库名。
- `middle`：可选项，代表库表名之间的分隔符。
- `{table}`：可选项，用于匹配表名。
- `suffix`：可选项，代表 Topic Name 的后缀。

其中 `prefix`、`middle` 以及 `suffix` 仅允许出现大小写字母（`a-z`、`A-Z`）、数字（`0-9`）、点号（`.`）、下划线（`_`）和中划线（`-`）；`{schema}`、`{table}` 均为小写，诸如 `{Schema}` 以及 `{TABLE}` 这样的占位符是无效的。

一些示例如下：

- `matcher = ['test1.table1', 'test2.table2'], topic = "hello_{schema}_{table}"`
  - 对于表 `test1.table1` 对应的数据变更事件，发送到名为 `hello_test1_table1` 的topic 中
  - 对于表 `test2.table2` 对应的数据变更事件，发送到名为 `hello_test2_table2` 的topic 中
- `matcher = ['test3.*', 'test4.*'], topic = "hello_{schema}_world"`
  - 对于 `test3` 下的所有表对应的数据变更事件，发送到名为 `hello_test3_world` 的topic 中
  - 对于 `test4` 下的所有表对应的数据变更事件，发送到名为 `hello_test4_ world` 的topic 中
- `matcher = ['*.*'], topic = "{schema}_{table}"`
  - 对于 TiCDC 监听的所有表， 按照 “库名 _ 表名” 的规则分别分发到独立的 topic 中； 例如对于 `test.account` 表，TiCDC 会将其数据变更日志分发到名为 `test_account` 的Topic 中。

DDL 事件的分发

库级别 DDL

诸如 `create database`、`drop database` 这类和某一张具体的表无关的 DDL，称之为库级别 DDL。对于库级别 DDL 对应的事件，被发送到 `--sink-uri` 中指定的默认topic 中。

表级别 DDL

诸如 `alter table`、`create table` 这类和某一张具体的表相关的 DDL，称之为表级别 DDL。对于表级别 DDL 对应的事件，按照dispatchers 的配置，被发送到相应的topic 中。

例如，对于 `matcher = ['test.*'], topic = {schema}_{table}` 这样的dispatchers 配置，DDL 事件分发情况如下：

- 若 DDL 事件中涉及单张表，则将 DDL 事件原样发送到相应的topic 中。
  - 对于 DDL 事件 `drop table test.table1`，该事件会被发送到名为 `test_table1` 的 topic 中。
- 若 DDL 事件中涉及多张表（`rename table` / `drop table` / `drop view` 都可能涉及多张表），则将单个 DDL 事件拆分为多个发送到相应的topic 中。
  - 对于 DDL 事件 `rename table test.table1 to test.table10, test.table2 to test.table20`，则将 `rename table test.table1 to test.table10` 的 DDL 事件发送到名为 `test_table1` 的 topic 中，将 `rename table test.table2 to test.table20` 的 DDL 事件发送到名为 `test.table2` 的 topic 中。

Partition 分发器

partition 分发器用partition = “xxx” 来指定，支持default、 ts、 index-value、 table 四种partition 分发器，分发规则如下：

- default：有多个唯一索引（包括主键）时按照 table 模式分发；只有一个唯一索引（或主键）按照 index-value 模式分发；如果开启了 old value 特性，按照table 分发
- ts：以行变更的commitTs 做 Hash 计算并进行 event 分发
- index-value：以表的主键或者唯一索引的值做 Hash 计算并进行 event 分发
- table：以表的 schema 名和table 名做 Hash 计算并进行 event 分发

> 注意：
>
> 从 v6.1 开始，为了明确配置项的含义，用来指定 partition 分发器的配置项由原来的 `dispatcher` 改为 `partition`，`partition` 为 `dispatcher` 的别名。例如，以下两条规则完全等价：
>
> ```
> [sink]
> dispatchers = [
>    {matcher = ['*.*'], dispatcher = "ts"},
>    {matcher = ['*.*'], partition = "ts"},
> ]
> ```
>
> 但是 `dispatcher` 与 `partition` 不能出现在同一条规则中。例如，以下规则非法：
>
> ```
> {matcher = ['*.*'], dispatcher = "ts", partition = "table"},
> ```

> 警告：
>
> 当开启Old Value 功能时 (`enable-old-value = true`)，使用 index-value 分发器可能导致无法确保相同索引值的行变更顺序。因此，建议使用default 分发器。
>
> 具体原因请参考TiCDC 在开启 Old Value 功能后更新事件格式有何变化？

横向扩展大单表的负载到多个TiCDC 节点

该功能可以按照大单表的数据量和每分钟的修改行数将表的同步范围切分为多个，并使各个范围之间所同步的数据量和修改行数基本相同。该功能将这些范围分布到多个TiCDC 节点进行同步，使得多个TiCDC 节点可以同时同步大单表。该功能可以解决以下两个问题：

- 单个TiCDC 节点不能及时同步大单表。
- TiCDC 节点之间资源（CPU、内存等）消耗不均匀。

> **警告：**
>
> TiCDC v7.0.0 仅支持在 Kafka 同步任务上开启大单表的横向扩展功能。

配置样例如下所示：

```
[scheduler]
##### 默认值为 "false"，设置为 "true" 以打开该功能。
enable-table-across-nodes = true
##### 打开该功能后，该功能只对 Region 个数大于 `region-threshold` 值的表生效。
region-threshold = 100000
##### 打开该功能后，该功能会对每分钟修改行数大于 `write-key-threshold` 值的表生效。
##### 注意：
##### * 该参数默认值为 0，代表该功能默认不会按表的修改行数来切分表的同步范围。
##### * 你可以根据集群负载来配置该参数，如 30000，代表当表每分钟的更新行数超过 30000 时，
    ↪ 该功能将会切分表的同步范围。
##### * 当 `region-threshold` 和 `write-key-threshold` 同时配置时，
#####   TiCDC 将优先检查修改行数是否大于 `write-key-threshold`，
#####   如果不超过，则再检查 Region 个数是否大于 `region-threshold`。
write-key-threshold = 30000
```

一个表包含的Region 个数可用如下 SQL 查询：

```
SELECT COUNT(*) FROM INFORMATION_SCHEMA.TIKV_REGION_STATUS WHERE DB_NAME="database1" AND
    ↪ TABLE_NAME="table1" AND IS_INDEX=0;
```

10.9.3.2.3 同步数据到存储服务

从 TiDB v6.5.0 开始，TiCDC 支持将行变更事件保存至存储服务，如Amazon S3、 GCS、 Azure Blob Storage 和 NFS。 本文介绍如何使用TiCDC 创建同步任务(Changefeed) 将增量数据同步到这类存储服务，并介绍数据的存储方式。具体如下：

- 如何将变更数据同步至存储服务。
- 变更数据如何在存储服务中保存。

同步变更数据至存储服务

使用以下命令来创建同步任务：

```
cdc cli changefeed create \
    --server=http://10.0.10.25:8300 \
    --sink-uri="s3://logbucket/storage_test?protocol=canal-json" \
    --changefeed-id="simple-replication-task"
```

输出结果如下：

```
Info: {"upstream_id":7171388873935111376,"namespace":"default","id":"simple-replication-task","
    ↪ sink_uri":"s3://logbucket/storage_test?protocol=canal-json","create_time":"2022-11-29T18
    ↪ :52:05.566016967+08:00","start_ts":437706850431664129,"engine":"unified","config":{"
    ↪ case_sensitive":true,"enable_old_value":true,"force_replicate":false,"
    ↪ ignore_ineligible_table":false,"check_gc_safe_point":true,"enable_sync_point":false,"
    ↪ sync_point_interval":600000000000,"sync_point_retention":86400000000000,"filter":{"rules
    ↪ ":["*.*"],"event_filters":null},"mounter":{"worker_num":16},"sink":{"protocol":"canal-
    ↪ json","schema_registry":"","csv":{"delimiter":",","quote":"\"","null":"\\N","
    ↪ include_commit_ts":false},"column_selectors":null,"transaction_atomicity":"none","
    ↪ encoder_concurrency":16,"terminator":"\r\n","date_separator":"none","
    ↪ enable_partition_separator":false},"consistent":{"level":"none","max_log_size":64,"
    ↪ flush_interval":2000,"storage":""}},"state":"normal","creator_version":"v6.5.0-master-
    ↪ dirty"}
```

- `--server`：TiCDC 集群中任意一个TiCDC 服务器的地址。
- `--changefeed-id`：同步任务的 ID。格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为ID。
- `--sink-uri`：同步任务下游的地址。具体可参考配置 Sink URI。
- `--start-ts`：指定 changefeed 的开始TSO。TiCDC 集群将从这个TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标TSO。TiCDC 集群拉取数据直到这个TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件，详见TiCDC Changefeed 配置参数。

配置 Sink URI

本章节介绍如何在 Sink URI 中配置存储服务Amazon S3、GCS、Azure Blob Storage 以及 NFS。Sink URI 用于指定TiCDC 下游系统的连接信息，遵循以下格式：

```
[scheme]://[host]/[path]?[query_parameters]
```

URI 的 `[query_parameters]` 中可配置的参数如下：

参数	描述	默认值	取值范围
`worker-count`	向下游存储服务保存数据变更记录的并发度	16	`[1, 512]`
`flush-interval`	向下游存储服务保存数据变更记录的间隔	5s	`[2s, 10m]`
`file-size`	单个数据变更文件的字节数超过 `file-size` 时将其保存至存储服务中	67108864	`[1048576, 536870912]`
`protocol`	输出到存储服务的消息协议	N/A	`canal-json` 和 `csv`
`enable-tidb-extension`	`protocol` 参数为 `canal-json` 时，如果该值为 `true`，TiCDC 会发送WATERMARK 事件，并在 canal-json 消息中添加TiDB 扩展字段。	`false`	`false` 和 `true`

> **注意：**
>
> `flush-interval` 与 `file-size` 二者只要满足其一就会向下游写入数据变更文件。
>
> `protocol` 是必选配置， 如果 TiCDC 在创建 changefeed 时未解析到该配置， 将会返回 CDC:
> ↪ `ErrSinkUnknownProtocol` 错误。

配置外部存储

Amazon S3 配置样例如下：

```
--sink-uri="s3://bucket/prefix?protocol=canal-json"
```

GCS 配置样例如下：

```
--sink-uri="gcs://bucket/prefix?protocol=canal-json"
```

Azure Blob Storage 配置样例如下：

```
--sink-uri="azure://bucket/prefix?protocol=canal-json"
```

> **建议：**
>
> Amazon S3、 GCS 以及 Azure Blob Storage 的 URI 参数与 BR 中这三种外部存储的 URI 参数相同。详细参数说明请参考BR 备份存储服务的 URI 格式。

配置 NFS

NFS 配置样例如下：

```
--sink-uri="file:///my-directory/prefix?protocol=canal-json"
```

存储路径组织结构

本章节详细介绍数据变更记录、元数据与 DDL 事件的存储路径组织结构。

数据变更记录

数据变更记录将会存储到以下路径：

```
{scheme}://{prefix}/{schema}/{table}/{table-version-separator}/{partition-separator}/{date-
    ↪ separator}/CDC{num}.{extension}
```

- `scheme`：存储服务类型。例如：s3、gcs、azure、file。
- `prefix`：用户指定的父目录。例如：s3://bucket/bbb/ccc。

- `schema`：表所属的库名。例如：s3://bucket/bbb/ccc/test。
- `table`：表名。例如：s3://bucket/bbb/ccc/test/table1。
- `table-version-separator`：将文件路径按照表的版本进行分隔。例如：s3://bucket/bbb/ccc/test/table1/9999。
- `partition-separator`：将文件路径按照表的分区号进行分隔。例如：s3://bucket/bbb/ccc/test/table1/9999/20。
- `date-separator`：将文件路径按照事务提交的日期进行分隔，可选值如下：
  - `none`：不以 `date-separator` 分隔文件路径。例如：`test.table1` 版本号为 `9999` 的所有文件都存到 `s3://bucket/bbb/ccc/test/table1/9999` 路径下。
  - `year`：以事务提交的年份分隔文件路径。例如：s3://bucket/bbb/ccc/test/table1/9999/2022。
  - `month`：以事务提交的年份和月份分隔文件路径。例如：s3://bucket/bbb/ccc/test/table1/9999/2022-01。
  - `day`：以事务提交的年月日来分隔文件路径。例如：s3://bucket/bbb/ccc/test/table1/9999/2022-01-02。
- `num`：存储数据变更记录的目录下文件的序号。例如：s3://bucket/bbb/ccc/test/table1/9999/2022-01-02/CDC000005.csv。
- `extension`：文件的扩展名。v6.5.0 支持CSV 和 Canal-JSON 格式。

> 注意：
>
> 表的版本可能在以下三种情况下发生变化：
>
> - 发生 DDL 操作后，表的版本为该 DDL 在上游TiDB 执行结束的TSO。但是，表版本的变化并不意味着表结构的变化。例如，在表中的某一列添加注释，不会导致 `schema.json` 文件内容发生变化。
> - 进程重启，表的版本为进程重启时 Changefeed 的 checkpoint TSO。在有很多表的情况下，重启时需要遍历所有目录并找到上一次重启时每张表写入的位置，这样的操作耗时较长。因此，TiCDC 选择在一个以checkpoint TSO 为版本的新目录下写入数据，而不是在旧版本的目录下继续写入数据。
> - 发生表调度后，表的版本为该表调度到当前节点时 Changefeed 的checkpoint TSO。

Index 文件

Index 文件用于防止已写入的数据被错误覆盖，与数据变更记录存储在相同路径：

```
{scheme}://{prefix}/{schema}/{table}/{table-version-separator}/{partition-separator}/{date-
    ↪ separator}/CDC.index
```

Index 文件记录了当前目录下所使用到的最大文件名，比如：

```
CDC000005.csv
```

上述内容表明该目录下 `CDC000001.csv` 到 `CDC000004.csv` 文件已被占用，当TiCDC 集群中发生表调度或者节点重启时，新的节点会读取Index 文件，并判断 `CDC000005.csv` 是否被占用。如果未被占用，则新节点会从 `CDC000005.csv` 开始写文件。如果已被占用，则从 `CDC000006.csv` 开始写文件，这样可防止覆盖其他节点写入的数据。

元数据

元数据信息将会存储到以下路径：

```
{scheme}://{prefix}/metadata
```

元数据信息以JSON 格式存储到如下的文件中：

```
{
    "checkpoint-ts":433305438660591626
}
```

- checkpoint-ts：commit-ts 小于等于此 checkpoint-ts 的事务都被写入下游存储当中。

DDL 事件

当 DDL 事件引起表的版本变更时，TiCDC 将会切换到新的路径下写入数据变更记录，例如 test.table1 的版本从 9999 变更为 10000 时将会在 s3://bucket/bbb/ccc/test/table1/10000/2022-01-02/CDC000001.csv 路径中写入数据。并且，当 DDL 事件发生时，TiCDC 将生成一个 schema.json 文件存储表结构信息。

表结构信息将会存储到以下路径：

```
{scheme}://{prefix}/{schema}/{table}/{table-version-separator}/schema.json
```

一个示例 schema.json 文件如下：

```
{
    "Table":"table1",
    "Schema":"test",
    "Version":1,
    "TableVersion":10000,
    "Query": "ALTER TABLE test.table1 ADD OfficeLocation blob(20)",
    "TableColumns":[
        {
            "ColumnName":"Id",
            "ColumnType":"INT",
            "ColumnNullable":"false",
            "ColumnIsPk":"true"
        },
        {
            "ColumnName":"LastName",
            "ColumnType":"CHAR",
            "ColumnLength":"20"
        },
        {
            "ColumnName":"FirstName",
            "ColumnType":"VARCHAR",
            "ColumnLength":"30"
        },
        {
```

```
            "ColumnName":"HireDate",
            "ColumnType":"DATETIME"
        },
        {
            "ColumnName":"OfficeLocation",
            "ColumnType":"BLOB",
            "ColumnLength":"20"
        }
    ],
    "TableColumnsTotal":"5"
}
```

- Table：表名。
- Schema：表所属的库名。
- Version：Storage sink 协议版本号。
- TableVersion：表的版本号。
- Query：DDL 语句。
- TableColumns：该数组表示表中每一列的详细信息。
  - ColumnName：列名。
  - ColumnType：该列的类型。详见数据类型。
  - ColumnLength：该列的长度。详见数据类型。
  - ColumnPrecision：该列的精度。详见数据类型。
  - ColumnScale：该列小数位的长度。详见数据类型。
  - ColumnNullable：值为 true 时表示该列可以含 NULL 值。
  - ColumnIsPk：值为 true 时表示该列是主键的一部分。
- TableColumnsTotal：TableColumns 数组的大小。

数据类型

本章节主要介绍 schema.json 文件中使用的各种数据类型。数据类型定义为 T(M[, D])，详见数据类型概述。

整数类型

TiDB 中整数类型可被定义为 IT[(M)] [UNSIGNED]，其中：

- IT 为整数类型，包括 TINYINT、 SMALLINT、 MEDIUMINT、 INT、 BIGINT 和 BIT。
- M 为该类型的显示宽度。

schema.json 文件中对整数类型定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{IT} [UNSIGNED]",
    "ColumnPrecision":"{M}"
}
```

小数类型

TiDB 中的小数类型可被定义为 DT[(M,D)][UNSIGNED]，其中：

- DT 为小数类型，包括 FLOAT、 DOUBLE、 DECIMAL 和 NUMERIC。
- M 为该类型数据的精度，即整数位加上小数位的总长度。
- D 为小数位的长度。

schema.json 文件中对小数类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{DT} [UNSIGNED]",
    "ColumnPrecision":"{M}",
    "ColumnScale":"{D}"
}
```

时间和日期类型

TiDB 中的日期类型可被定义为 DT，其中：

- DT 为日期类型，包括 DATE 和 YEAR。

schema.json 文件中对日期类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{DT}"
}
```

TiDB 中的时间类型可被定义为 TT[(M)]，其中：

- TT 为时间类型，包括 TIME、 DATETIME 和 TIMESTAMP。
- M 为秒的精度，取值范围为 0~6。

schema.json 文件中对时间类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{TT}",
    "ColumnScale":"{M}"
}
```

字符串类型

TiDB 中的字符串类型可被定义为 ST[(M)]，其中：

- ST 为字符串类型，包括 CHAR、 VARCHAR、 TEXT、 BINARY、 BLOB、 JSON 等。

- M 表示字符串的最大长度。

schema.json 文件中对字符串类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{ST}",
    "ColumnLength":"{M}"
}
```

Enum/Set 类型

schema.json 文件中对Enum/Set 类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{ENUM/SET}",
}
```

10.9.3.3 管理 Changefeed

本文介绍 Changefeed 相关的各种管理方法，其中大部分功能是通过TiCDC 的命令行工具来完成的。如果你需要，也可以通过TiCDC 暴露的 HTTP 接口实现类似功能，详细信息参考TiCDC OpenAPI。

10.9.3.3.1 创建同步任务

使用以下命令来创建同步任务：

```
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"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"upstream_id":7178706266519722477,"namespace":"default","id":"simple-replication-task","
    ′→ sink_uri":"mysql://root:xxxxx@127.0.0.1:4000/?time-zone=","create_time":"2022-12-19T15
    ′→ :05:46.679218+08:00","start_ts":438156275634929669,"engine":"unified","config":{"
    ′→ case_sensitive":true,"enable_old_value":true,"force_replicate":false,"
    ′→ ignore_ineligible_table":false,"check_gc_safe_point":true,"enable_sync_point":true,"
    ′→ bdr_mode":false,"sync_point_interval":30000000000,"sync_point_retention":3600000000000,"
    ′→ filter":{"rules":["test.*"],"event_filters":null},"mounter":{"worker_num":16},"sink":{"
    ′→ protocol":"","schema_registry":"","csv":{"delimiter":",","quote":"\"","null":"\\N","
    ′→ include_commit_ts":false},"column_selectors":null,"transaction_atomicity":"none","
    ′→ encoder_concurrency":16,"terminator":"\r\n","date_separator":"none","
    ′→ enable_partition_separator":false},"consistent":{"level":"none","max_log_size":64,"
    ′→ flush_interval":2000,"storage":""}},"state":"normal","creator_version":"v6.5.0"}
```

10.9.3.3.2 查询同步任务列表

使用以下命令来查询同步任务列表：

```
cdc cli changefeed list --server=http://10.0.10.25:8300
```

```
[{
    "id": "simple-replication-task",
    "summary": {
      "state": "normal",
      "tso": 417886179132964865,
      "checkpoint": "2020-07-07 16:07:44.881",
      "error": null
    }
}]
```

- checkpoint 即为TiCDC 已经将该时间点前的数据同步到了下游。
- state 为该同步任务的状态：
  - normal：正常同步
  - stopped：停止同步（手动暂停）
  - error：停止同步（出错）
  - removed：已删除任务（只在指定 --all 选项时才会显示该状态的任务。未指定时，可通过 query 查询该状态的任务）
  - finished：任务已经同步到指定 target-ts，处于已完成状态（只在指定 --all 选项时才会显示该状态的任务。未指定时，可通过 query 查询该状态的任务）。

10.9.3.3.3 查询特定同步任务

使用 changefeed query 命令可以查询特定同步任务（对应某个同步任务的信息和状态），指定 --simple 或 -s 参数会简化输出，提供最基本的同步状态和checkpoint 信息。不指定该参数会输出详细的任务配置、同步状态和同步表信息。

```
cdc cli changefeed query -s --server=http://10.0.10.25:8300 --changefeed-id=simple-replication-
    ↪ task
```

```
{
 "state": "normal",
 "tso": 419035700154597378,
 "checkpoint": "2020-08-27 10:12:19.579",
 "error": null
}
```

以上命令中：

- state 代表当前changefeed 的同步状态，各个状态必须和 changefeed list 中的状态相同。

- `tso` 代表当前changefeed 中已经成功写入下游的最大事务TSO。
- `checkpoint` 代表当前changefeed 中已经成功写入下游的最大事务TSO 对应的时间。
- `error` 记录当前changefeed 是否有错误发生。

```
cdc cli changefeed query --server=http://10.0.10.25:8300 --changefeed-id=simple-replication-task
```

```
{
  "info": {
    "sink-uri": "mysql://127.0.0.1:3306/?max-txn-row=20\u0026worker-number=4",
    "opts": {},
    "create-time": "2020-08-27T10:33:41.687983832+08:00",
    "start-ts": 419036036249681921,
    "target-ts": 0,
    "admin-job-type": 0,
    "sort-engine": "unified",
    "sort-dir": ".", "config":
    {
      "case-sensitive": true,
      "enable-old-value": false,
      "filter": {
        "rules": [
          "*.*"
        ],
        "ignore-txn-start-ts": null,
        "ddl-allow-list": null
      },
      "mounter":
        { "worker-num":
        16
      },
      "sink":
        { "dispatchers": null,
      },
      "scheduler": {
        "type": "table-number",
        "polling-time": -1
      }
    },
    "state": "normal",
    "history": null,
    "error": null
  },
  "status": {
    "resolved-ts": 419036036249681921,
    "checkpoint-ts": 419036036249681921,
    "admin-job-type": 0
```

```
  },
  "count": 0,
  "task-status": [
    {
      "capture-id": "97173367-75dc-490c-ae2d-4e990f90da0f",
      "status": {
        "tables": {
          "47": {
            "start-ts": 419036036249681921
          }
        },
        "operation": null,
        "admin-job-type": 0
      }
    }
  ]
}
```

以上命令中：

- `info` 代表查询changefeed 的同步配置。
- `status` 代表查询changefeed 的同步状态信息。
  - `resolved-ts` 代表当前changefeed 中已经成功从TiKV 发送到TiCDC 的最大事务TS。
  - `checkpoint-ts` 代表当前changefeed 中已经成功写入下游的最大事务TS。
  - `admin-job-type` 代表一个changefeed 的状态：
    * `0`：状态正常。
    * `1`：任务暂停，停止任务后所有同步 `processor` 会结束退出，同步任务的配置和同步状态都会保留，可以从 `checkpoint-ts` 恢复任务。
    * `2`：任务恢复，同步任务从 `checkpoint-ts` 继续同步。
    * `3`：任务已删除，接口请求后会结束所有同步 `processor`，并清理同步任务配置信息。同步状态保留，只提供查询，没有其他实际功能。
- `task-status` 代表查询changefeed 所分配的各个同步子任务的状态信息。

10.9.3.3.4 停止同步任务

使用以下命令来停止同步任务：

```
cdc cli changefeed pause --server=http://10.0.10.25:8300 --changefeed-id simple-replication-task
```

以上命令中：

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

10.9.3.3.5 恢复同步任务

使用以下命令恢复同步任务：

```
cdc cli changefeed resume --server=http://10.0.10.25:8300 --changefeed-id simple-replication-task
```

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。
- `--overwrite-checkpoint-ts`：从 v6.2 开始支持指定changefeed 恢复的起始TSO。TiCDC 集群将从这个TSO 开始拉取数据。该项支持 now 或一个具体的TSO（如 434873584621453313），指定的 TSO 应在(GC safe point, CurrentTSO] 范围内。如未指定该参数，默认从当前的 `checkpoint-ts` 同步数据。
- `--no-confirm`：恢复同步任务时无需用户确认相关信息。默认为 false。

> 注意：
>
> - 若 `--overwrite-checkpoint-ts` 指定的TSO `t2` 大于changefeed 的当前checkpoint TSO `t1`（可通过 `cdc cli changefeed query` 命令获取），则会导致 `t1` 与 `t2` 之间的数据不会同步到下游，造成数据丢失。
> - 若 `--overwrite-checkpoint-ts` 指定的TSO `t2` 小于changefeed 的当前checkpoint TSO `t1`，则会导致TiCDC 集群从一个旧的时间点 `t2` 重新拉取数据，可能会造成数据重复（例如 TiCDC 下游为 MQ sink）。

10.9.3.3.6 删除同步任务

使用以下命令删除同步任务：

```
cdc cli changefeed remove --server=http://10.0.10.25:8300 --changefeed-id simple-replication-task
```

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

10.9.3.3.7 更新同步任务配置

TiCDC 从 4.0.4 开始支持非动态修改同步任务配置，修改changefeed 配置需要按照暂停任务 -> 修改配置 -> ′→ 恢复任务的流程。

```
cdc cli changefeed pause -c test-cf --server=http://10.0.10.25:8300
cdc cli changefeed update -c test-cf --server=http://10.0.10.25:8300 --sink-uri="mysql
    ′→ ://127.0.0.1:3306/?max-txn-row=20&worker-number=8" --config=changefeed.toml
cdc cli changefeed resume -c test-cf --server=http://10.0.10.25:8300
```

当前支持修改的配置包括：

- changefeed 的 `sink-uri`
- changefeed 配置文件及文件内所有配置
- changefeed 是否使用文件排序和排序目录
- changefeed 的 `target-ts`

10.9.3.3.8 管理同步子任务处理单元(processor)

- 查询 processor 列表：

```
cdc cli processor list --server=http://10.0.10.25:8300
```

```
[
        {
                "id": "9f84ff74-abf9-407f-a6e2-56aa35b33888",
                "capture-id": "b293999a-4168-4988-a4f4-35d9589b226b",
                "changefeed-id": "simple-replication-task"
        }
]
```

- 查询特定 processor，对应于某个节点处理的同步子任务信息和状态：

```
cdc cli processor query --server=http://10.0.10.25:8300 --changefeed-id=simple-replication-
    ↪ task --capture-id=b293999a-4168-4988-a4f4-35d9589b226b
```

```
{
  "status": {
    "tables": {
      "56": {    # 56 表示同步表 id，对应 TiDB 中表的 tidb_table_id
        "start-ts": 417474117955485702
      }
    },
    "operation": null,
    "admin-job-type": 0
  },
  "position": {
    "checkpoint-ts": 417474143881789441,
    "resolved-ts": 417474143881789441,
    "count": 0
  }
}
```

以上命令中：

- status.tables 中每一个作为 key 的数字代表同步表的id，对应TiDB 中表的tidb_table_id。
- resolved-ts 代表当前 Processor 中已经排序数据的最大TSO。
- checkpoint-ts 代表当前 Processor 已经成功写入下游的事务的最大TSO。

10.9.3.3.9 输出行变更的历史值从 v4.0.5 版本开始引入

默认配置下，同步任务输出的TiCDC Open Protocol 行变更数据只包含变更后的值，不包含变更前行的值，因此该输出数据不满足TiCDC Open Protocol 的消费端使用行变更历史值的需求。

从 v4.0.5 开始，TiCDC 支持输出行变更数据的历史值。若要开启该特性，需要在changefeed 的配置文件的根级别指定以下配置：

```
enable-old-value = true
```

从v5.0 开始默认启用该特性，开启该特性后TiCDC Open Protocol 的输出格式参考TiCDC 开放数据协议- Row Changed Event。

10.9.3.3.10 同步启用了TiDB 新的 Collation 框架的表

从 v4.0.15、v5.0.4、v5.1.1 和 v5.2.0 开始，TiCDC 支持同步启用了TiDB 新的 Collation 框架的表。

10.9.3.3.11 同步没有有效索引的表

从 v4.0.8 开始，TiCDC 支持通过修改任务配置来同步没有有效索引的表。若要开启该特性，需要在 changefeed 配置文件的根级别进行如下指定：

```
enable-old-value = true
force-replicate = true
```

> 警告：
>
> 对于没有有效索引的表，INSERT 和 REPLACE 等操作不具备可重入性，因此会有数据冗余的风险。TiCDC 在同步过程中只保证数据至少分发一次，因此开启该特性同步没有有效索引的表，一定会导致数据冗余出现。如果不能接受数据冗余，建议增加有效索引，譬如增加具有 AUTO RANDOM 属性的主键列。

10.9.3.3.12 Unified Sorter 功能

> 注意：
>
> 从 v6.0.0 开始，TiCDC 内部默认使用 DB Sorter 引擎来对数据进行排序，不再使用 Unified Sorter。建议用户不再主动配置 Sorter 项。

Unified Sorter 是TiCDC 中的排序引擎功能，用于缓解以下场景造成的内存溢出问题：

- 如果TiCDC 数据订阅任务的暂停中断时间长，其间积累了大量的增量更新数据需要同步。
- 从较早的时间点启动数据订阅任务，业务写入量大，积累了大量的更新数据需要同步。

对v4.0.13 版本之后的cdc cli 创建的changefeed，默认开启Unified Sorter。对v4.0.13 版本前已经存在的changefeed，则使用之前的配置。

要确定一个changefeed 上是否开启了 Unified Sorter 功能，可执行以下示例命令查看（假设 PD 实例的IP 地址为 http://10.0.10.25:2379 ）：

```
cdc cli --server="http://10.0.10.25:8300" changefeed query --changefeed-id=simple-replication-
    ↪ task | grep 'sort-engine'
```

以上命令的返回结果中，如果 `sort-engine` 的值为“unified”，则说明 Unified Sorter 已在该changefeed 上开启。

> 注意：
>
> - 如果服务器使用机械硬盘或其他有延迟或吞吐有瓶颈的存储设备，Unified Sorter 性能会受到较大影响。
> - Unified Sorter 默认使用 `data_dir` 储存临时文件。建议保证硬盘的空闲容量大于等于 500 GiB。对于生产环境，建议保证每个节点上的磁盘可用空间大于（业务允许的最大）`checkpoint-ts` 延迟 * 业务高峰上游写入流量。此外，如果在 `changefeed` 创建后预期需要同步大量历史数据，请确保每个节点的空闲容量大于等于要追赶的同步数据。

10.9.3.4 Changefeed 日志过滤器

TiCDC 支持基于表和事件两个维度的过滤功能，本文分别介绍两种过滤器的使用方法。

10.9.3.4.1 Table Filter 表过滤器

TiCDC 支持基于库表名的过滤功能，你可以通过下列配置来选定或过滤掉指定的表：

```
[filter]
#### 过滤器规则
rules = ['*.*', '!test.*']
```

常见的过滤器规则示例：

- `rules = ['*.*']`
  - 同步所有的表（不包含系统表）
- `rules = ['test1.*']`
  - 同步库 `test1` 下的所有表
- `rules = ['*.*', '!scm1.tbl2']`
  - 同步所有的表但排除表 `scm1.tbl2`
- `rules = ['scm1.tbl2', 'scm1.tbl3']`
  - 只同步表 `scm1.tbl2` 和 `scm1.tbl3`
- `rules = ['scm1.tidb_*']`
  - 同步库 `scm1` 下所有表名前缀为 `tidb_` 的表

更多用法说明参见：库表过滤语法

10.9.3.4.2 Event Filter 事件过滤器从 v6.2.0 版本开始引入

TiCDC 在 v6.2.0 中新增了事件过滤器功能，你可以通过配置该规则来过滤符合指定条件的 DML 和 DDL 事件。

以下是事件过滤器的配置规则示例：

```
[filter]
#### 事件过滤器的规则应该写在 filter 配置项之下，可以同时配置多个事件过滤器。

[[filter.event-filters]]
matcher = ["test.worker"] # 该过滤规则只应用于 test 库中的 worker 表
ignore-event = ["insert"] # 过滤掉 insert 事件
ignore-sql = ["^drop", "add column"] # 过滤掉以 "drop" 开头或者包含 "add column" 的 DDL
ignore-delete-value-expr = "name = 'john'" # 过滤掉包含 name = 'john' 条件的 delete DML
ignore-insert-value-expr = "id >= 100" # 过滤掉包含 id >= 100 条件的 insert DML
ignore-update-old-value-expr = "age < 18 or name = 'lili'" # 过滤掉旧值 age < 18 或 name = 'lili'
    ↪ 的 update DML
ignore-update-new-value-expr = "gender = 'male' and age > 18" # 过滤掉新值 gender = 'male' 且 age
    ↪ > 18 的 update DML
```

配置参数说明：

- `matcher`：该事件过滤器所要匹配的数据库名和表名，其匹配规则和表库过滤规则相一致。
- `ignore-event`：要过滤的事件类型，它是一个字符串数组，可以配置多个事件类型。目前支持的类型如下表所示:

Event	分类	别名	说明
all dml			匹配所有 DML events
all ddl			匹配所有 DDL events
insert	DML		匹配 insert DML event
update	DML		匹配 update DML event
delete	DML		匹配 delete DML event
create schema	DDL	create database	匹配 create database event
drop schema	DDL	drop database	匹配 drop database event
create table	DDL		匹配 create table event
drop table	DDL		匹配 drop table event
rename table	DDL		匹配 rename table event
truncate table	DDL		匹配 truncate table event
alter table	DDL		匹配 alter table event (包含 alter table 的所有子句和 create/drop index)
add table partition	DDL		匹配 add table partition event
drop table partition	DDL		匹配 drop table partition event
truncate table partition	DDL		匹配 truncate table partition event
create view	DDL		匹配 create view event
drop view	DDL		匹配 drop view event

- `ignore-sql`：要过滤的 DDL 语句的正则表达式。该参数接受一个字符串数组，数组中可以配置多条正则表达式。该配置仅对 DDL 事件生效。
- `ignore-delete-value-expr`：配置一个 SQL 表达式，对带有指定值的 DELETE 类型的 DML 事件生效。
- `ignore-insert-value-expr`：配置一个 SQL 表达式，对带有指定值的INSERT 类型的 DML 事件生效。
- `ignore-update-old-value-expr`：配置一个 SQL 表达式，对带有指定旧值的 UPDATE 类型的 DML 事件生效。
- `ignore-update-new-value-expr`：配置一个 SQL 表达式，对带有指定新值的 UPDATE 类型的 DML 事件生效。

> 注意：
>
> - TiDB 在更新聚簇索引的列值时，会将一个 UPDATE 事件拆分成为 DELETE 和 INSERT 事件，TiCDC 无法将该类事件识别为 UPDATE 事件，因此无法正确地进行过滤。
> - 在配置 SQL 表达式时，请确保符合matcher 规则的所有表均包含了对应 SQL 表达式中指明的所有列，否则同步任务将无法创建成功。此外，若在同步的过程中表的结构发生变化，不再包含 SQL 表达式中的列，那么同步任务将会进入无法自动恢复的错误状态，你需要手动修改配置并进行恢复操作。

10.9.3.4.3 DDL 白名单

目前TiCDC 在同步 DDL 时使用白名单策略，只有在白名单内部的 DDL 会同步到下游，不在白名单内的 DDL 不会被TiCDC 同步到下游。

以下为TiCDC 支持同步的 DDL 的列表。

- create database
- drop database
- create table
- drop table
- add column
- drop column
- create index / add index
- drop index
- truncate table
- modify column
- rename table
- alter column default value
- alter table comment
- rename index
- add partition
- drop partition
- truncate partition
- create view
- drop view
- alter table character set

- alter database character set
- recover table
- add primary key
- drop primary key
- rebase auto id
- alter table index visibility
- exchange partition
- reorganize partition
- alter table ttl
- alter table remove ttl

10.9.3.5 TiCDC 双向复制

从 v6.5.0 版本开始，TiCDC 支持在两个TiDB 集群之间进行双向复制。基于该功能，你可以使用TiCDC 来构建TiDB 集群的多写多活解决方案。

本文档以在两个TiDB 集群之间进行双向复制为例，介绍双向复制的使用方法。

10.9.3.5.1 部署双向复制

TiCDC 复制功能只会将指定时间点之后的增量变更复制到下游集群。开始双向复制之前，需要采取以下步骤：

1. （可选）根据实际需要，使用数据导出工具Dumpling 和导入工具TiDB Lightning 将两个集群的数据导入到对方集群。
2. 在这两个TiDB 集群之间部署两套TiCDC 集群，集群的拓扑如下图所示，图中箭头所指的方向即为该 TiCDC 集群同步数据的流向。

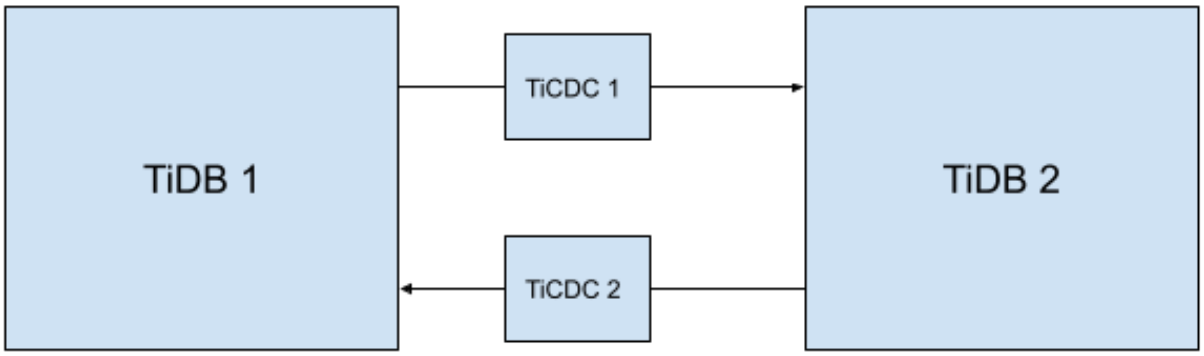


图 203: TiCDC bidirectional replication

3. 指定上下游集群的数据同步的开始时间点。
    1. 确认上下游集群的时间点。如果进行双集群容灾，那么建议确保两个集群在某个对应的时刻的数据是一致的，例如TiDB A 在 `ts=1` 时刻与TiDB B 在 `ts=2` 的时刻数据是一致的。

2. 在创建同步任务时，分别把对应集群的同步任务的 `--start-ts` 参数指定为对应的 `tso`，即上游为 TiDB A 的同步任务需设置参数 `--start-ts=1`，上游为 TiDB B 的同步任务需设置参数 `--start-ts=2`。

4. 在创建同步任务的 `--config` 参数所指定的配置文件中，添加如下配置:

```
# 是否启用 bdr 模式
bdr-mode = true
```

5. （可选）如果需要追踪数据写入源，可以使用TiDB 系统变量`tidb_source_id` 为集群设置不同的数据源 ID。

这样，以上搭建好的集群即可对数据进行双向复制。

10.9.3.5.2 执行 DDL

开启双向复制功能后，TiCDC 不会同步任何 DDL。用户需要自行在上下游集群中分别执行 DDL。

需要注意的是，某些 DDL 会造成表结构变更或者数据更改时序问题，从而导致数据同步后出现不一致的情况。因此，在开启双向同步功能后，只有下表中的 DDL 可以在业务不停止数据写入的情况下执行。

事件	是否会引起 changefeed 错误	说明
create database	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
drop database	是	需要手动重启 changefeed，指定 `--overwrite-checkpoint-ts` 为该条 DDL 的 commitTs 来恢复
create table	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
drop table	是	需要手动重启 changefeed，指定 `--overwrite-checkpoint-ts` 为该条 ddl 的 commitTs 来恢复
alter table comment	否	
rename index	否	
alter table index visibility	否	
add partition	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
drop partition	否	
create view	否	
drop view	否	
alter column default value	否	
reorganize partition	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
alter table ttl	否	

事件	是否会引起 changefeed 错误	说明
alter table remove ttl	否	
add not unique index	否	
drop not unique index	否	

如果需要执行以上列表中不存在的 DDL，需要采取以下步骤：

1. 暂停所有集群中需要执行 DDL 的对应的表的写入操作。
2. 等待所有集群中对应表的所有写入已经同步到其他集群后，手动在每一个TiDB 集群上单独执行所有的 DDL。
3. 等待 DDL 完成之后，重新恢复写入。

10.9.3.5.3 停止双向复制

在业务数据停止写入之后，你可以在两个集群中都插入一行特殊的值，通过检查这两行特殊的值来确保数据达到了一致的状态。

检查完毕之后，停止同步任务即可停止双向复制。

10.9.3.5.4 使用限制

- DDL 的限制见执行 DDL 小节。
- 双向复制的集群不具备检测写冲突的功能，写冲突将会导致未定义问题。你需要在业务层面保证不出现写冲突。
- TiCDC 双向复制功能支持超过 2 个集群的双向同步，但是不支持多个集群级联模式的同步，即TiDB A -> TiDB B -> TiDB C -> TiDB A 的环形复制方式。在这种部署方式下，如果其中一个链路出现问题则会影响整个数据同步链路。因此，如果需要部署多个集群之间的双向复制，每个集群都需要与其他集群两两相连，即 `TiDB A <-> TiDB B`，`TiDB B <-> TiDB C`，`TiDB C <-> TiDB A`。

10.9.3.6 TiCDC 单行数据正确性校验

从 v7.1.0 开始，TiCDC 引入了单行数据正确性校验功能。该功能基于 Checksum 算法，校验一行数据从TiDB 写入、通过TiCDC 同步，到写入 Kafka 集群的过程中数据内容是否发生错误。TiCDC 数据正确性校验功能仅支持下游是 Kafka 的 Changefeed，目前支持 Avro 协议。

10.9.3.6.1 实现原理

在启用单行数据 Checksum 正确性校验功能后，TiDB 使用 CRC32 算法计算该行数据的 Checksum 值，并将其一并写入TiKV。TiCDC 从 TiKV 读取数据，根据相同的算法重新计算 Checksum，如果该值与 TiDB 写入的值相同，则可以证明数据在TiDB 至 TiCDC 的传输过程中是正确的。

TiCDC 将数据编码成特定格式并发送至 Kafka。Kafka Consumer 读取数据后，可以使用与TiDB 相同的算法计算得到新的 Checksum，将此值与数据中携带的 Checksum 值进行比较，若二者一致，则可证明从TiCDC 至 Kafka Consumer 的传输链路上的数据是正确的。

关于 Checksum 值的计算规则，请参考Checksum 计算规则。

10.9.3.6.2 启用功能

TiCDC 数据正确性校验功能默认关闭，要使用该功能，请执行以下步骤：

1. 首先，你需要在上游TiDB 中开启行数据 Checksum 功能(tidb_enable_row_level_checksum)：

```
SET GLOBAL tidb_enable_row_level_checksum = ON;
```

上述配置仅对新创建的会话生效，因此需要重新连接TiDB。

2. 在创建 Changefeed 的 --config 参数所指定的配置文件中，添加如下配置：

```
[integrity]
integrity-check-level = "correctness"
corruption-handle-level = "warn"
```

3. 当使用Avro 作为数据编码格式时，你需要在sink-uri 中设置enable-tidb-extension=true。同时，为了防止数值类型在网络传输过程中发生精度丢失，导致 Checksum 校验失败，还需要设置avro-decimal-handling-mode=string 和avro-bigint-unsigned-handling-mode=string。下面是一个配置示例：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro-
    ↪ checksum" --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=avro&enable-tidb-
    ↪ extension=true&avro-decimal-handling-mode=string&avro-bigint-unsigned-handling-mode=
    ↪ string" --schema-registry=http://127.0.0.1:8081 --config changefeed_config.toml
```

通过上述配置，Changefeed 会在每条写入 Kafka 的消息中携带该消息对应数据的 Checksum，你可以根据此 Checksum 的值进行数据一致性校验。

> 注意：
> 对于已有 Changefeed，如果未设置 avro-decimal-handling-mode 和 avro-bigint-unsigned-handling-mode，开启 Checksum 校验功能时会引起 Schema 不兼容问题。可以通过修改 Schema Registry 的兼容性为 NONE 解决该问题。详情可参考 Schema 兼容性。

10.9.3.6.3 关闭功能

TiCDC 默认关闭单行数据的 Checksum 校验功能。若要在开启此功能后将其关闭，请执行以下步骤：

1. 首先， 按照TiCDC 更新同步任务配置的说明， 按照 暂停任务 -> 修改配置 -> 恢复任务 的流程更新 Changefeed 的配置内容。在 Changefeed 的 --config 参数所指定的配置文件中调整 [integrity] 的配置内容为：

```
[integrity]
integrity-check-level = "none"
corruption-handle-level = "warn"
```

2. 在上游TiDB 中关闭行数据 Checksum 功能(tidb_enable_row_level_checksum)，执行如下 SQL 语句：

```
SET GLOBAL tidb_enable_row_level_checksum = OFF;
```

上述配置仅对新创建的会话生效。在所有写入TiDB 的客户端都完成数据库连接重建后，Changefeed 写入 Kafka 的消息中将不再携带该条消息对应数据的 Checksum 值。

10.9.3.6.4 Checksum 计算规则

Checksum 计算算法的伪代码如下：

```
fn checksum(columns)
    { let result = 0
    for column in sort_by_schema_order(columns)
        { result = crc32.update(result,
        encode(column))
    }
    return result
}
```

- columns 应该按照column ID 排序。在Avro schema 中，各个字段已经按照column ID 的顺序排序，因此可以直接按照此顺序排序 columns。
- encode(column) 函数将column 的值编码为字节，编码规则取决于该column 的数据类型。具体规则如下：
  - TINYINT、SMALLINT、INT、BIGINT、MEDIUMINT 和YEAR 类型会被转换为 UINT64 类型，并按照小端序编码。例如，数字 0x0123456789abcdef 会被编码为 hex'0x0123456789abcdef'。
  - FLOAT 和 DOUBLE 类型会被转换为 DOUBLE 类型，然后转换为IEEE754 格式的 UINT64 类型。
  - BIT、ENUM 和 SET 类型会被转换为 UINT64 类型。
    - BIT 类型按照二进制转换为 UINT64 类型。
    - ENUM 和 SET 类型按照其对应的INT 值转换为 UINT64 类型。例如，SET('a','b','c') 类型column 的数据值为 'a,c'，则该值将被编码为 0b101。
  - TIMESTAMP、DATE、DURATION、DATETIME、JSON 和 DECIMAL 类型会被转换为 STRING 类型，然后转换为 UTF8 编码的字节。
  - VARBIANRY、BINARY 和 BLOB（包括 TINY、MEDIUM 和 LONG）类型会直接使用它的字节。
  - VARCHAR、CHAR 和TEXT（包括TINY、MEDIUM 和 LONG）类型会被编码为 UTF8 编码的字节。
  - NULL 和 GEOMETRY 类型不会被纳入到 Checksum 计算中，返回空字节。

> 注意：
>
> 开启 Checksum 校验功能后，DECIMAL 和 UNSIGNED BIGINT 类型的数据会被转换为字符串类型。因此在下游消费者代码中需要将其转换为对应的数值类型，然后进行 Checksum 相关计算。

Golang 消费者代码实现了解码从 Kafka 读取到的数据、按照 schema fields 排序以及 Checksum 计算等步骤。详情请参考 avro/decoder.go。

10.9.4 监控告警

10.9.4.1 TiCDC 基本监控指标

从 v7.0.0 版本开始，使用TiUP 一键部署Grafana 时，会自动在Grafana 监控页面新增TiCDC Summary Dashboard。通过该监控面板，你可以快速地了解TiCDC 服务器运行状态和同步任务的基本情况。

下图显示了TiCDC Dashboard 的监控栏：

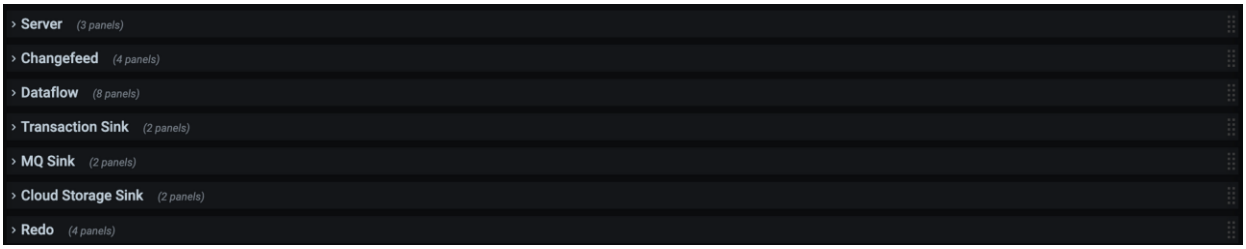


图 204: TiCDC Summary Dashboard - Overview

各监控栏说明如下：

- Server：集群中 TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务延迟和状态信息
- Dataflow：TiCDC 内部各个模块处理数据改变的各种统计信息
- Transaction Sink：下游为 MySQL 或者TiDB 时的写延迟信息
- MQ Sink: 下游为 MQ 系统时的写延迟信息
- Cloud Storage Sink：下游为 Cloud Storage 时的写速率信息
- Redo：开启Redo 功能时的写延迟信息

10.9.4.1.1 Server 监控栏

Server 监控栏示例如下：

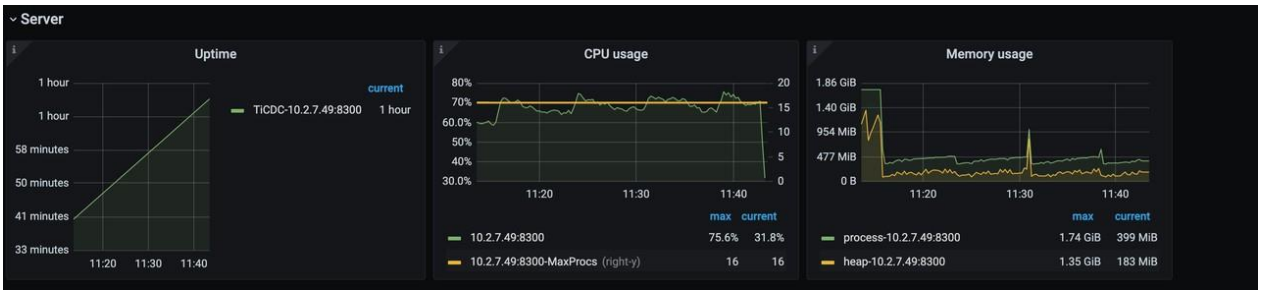


图 205: TiCDC Summary Dashboard - Server metrics

- Uptime：TiCDC 节点已经运行的时间。
- CPU usage：TiCDC 节点的 CPU 使用量。
- Memory usage：TiCDC 节点的内存使用量。

10.9.4.1.2 Changefeed 监控栏

Changefeed 监控栏示例如下：



图 206: TiCDC Summary Dashboard - Changefeed metrics

- Changefeed checkpoint lag：这个指标代表上游 TiDB 集群和下游系统之间的数据复制延迟，延迟以时间为单位。该指标反映了 Changefeed 整体的数据同步状况是否健康，通常情况下，lag 越小，说明同步任务状态越好。而当lag 上升时，通常说明 Changefeed 的同步能力或者下游系统的消费能力无法匹配上游的写入速度。
- Changefeed resolved ts lag：这个指标代表了上游TiDB 集群与TiCDC 节点之间的数据延迟，延迟以时间为单位。该指标能够反映 Changefeed 拉取上游数据变更的能力，当lag 上升时，说明 Changefeed 无法及时地拉取上游产生的数据变更。

10.9.4.1.3 Dataflow 监控栏



图 207: TiCDC Summary Dashboard - Puller metrics

- 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。该指标能够反映 TiCDC 拉取上游变更事件的速度。
- Puller output events：TiCDC 节点中 Puller 模块输出到 Sorter 模块的数据变更总数。



图 208: TiCDC Summary Dashboard - Sorter metrics

- Sorter output events/s：TiCDC 节点中 Sorter 模块每秒输出到 Sink 模块的数据变更个数。值得注意的是，Sorter 的数据输出速率会受到 Sink 模块的影响，因此在发现 Sorter 模块输出速率比 Puller 模块低时，不一定是因为 Sorter 模块排序速度过慢；而应该先观察 Sink 模块的相关指标，确认是否是因为 Sink 模块 Flush 数据的耗时较长，导致 Sorter 模块输出降低。
- Sorter output event：TiCDC 节点中 Sorter 模块输出到 Sink 模块的数据变更总个数。



图 209: TiCDC Summary Dashboard - Mounter metrics

- Mounter output events/s：TiCDC 节点中 Mounter 模块每秒解码的数据变更的个数。当上游发生的数据变更包含较多字段时，Mounter 的解码速度可能会受到影响。
- Mounter output event：TiCDC 节点中 Mounter 模块解码的数据变更的总个数。

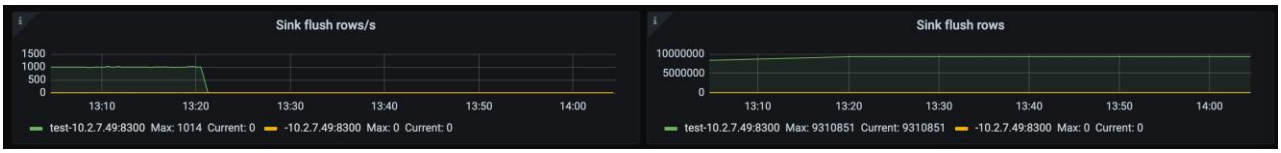


图 210: TiCDC Summary Dashboard - Sink metrics

- Sink flush rows/s：TiCDC 节点中 Sink 模块每秒往下游输出的数据变更个数。该指标反映的是同步任务向下游进行同步的速率，当 Sink flush rows/s 小于 Puller output events/s 时，同步延迟可能会上升。
- Sink flush rows：TiCDC 节点中 Sink 模块输出的数据变更的总个数。

10.9.4.1.4 Transaction Sink 监控栏

Transaction Sink 监控栏示意如下，该监控栏只有下游为 MySQL 或者TiDB 时才有数据。

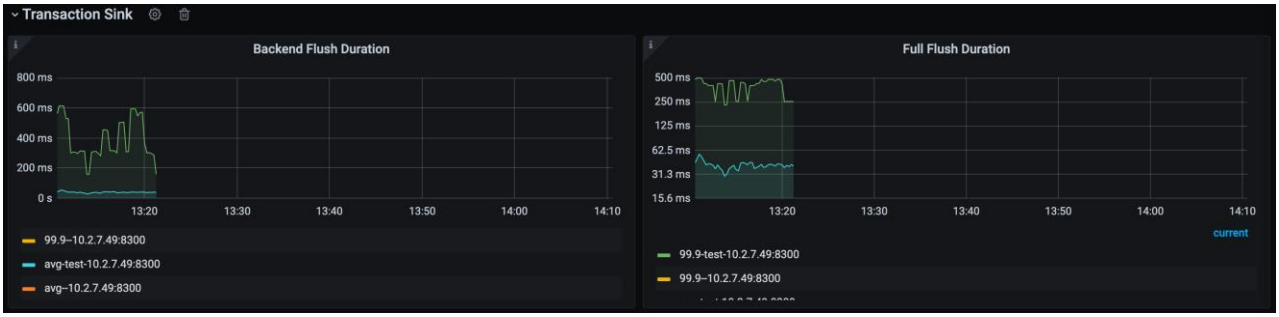


图 211: TiCDC Summary Dashboard - Transaction Sink metrics

- Backend Flush Duration：TiCDC Transaction Sink 模块在向下游执行一条 SQL 语句的耗时。通过观察该指标，能够判断下游的性能是否为同步速度的瓶颈。一般来说，p999 应该维持在 500 ms 内为佳，超过该值时，同步速度可能就会受到影响，引起 Changefeed checkpoint lag 上升。
- Full Flush Duration：TiCDC 中每个事务从 Sorter 排序完成直到发送到下游之间的总耗时。用该值减去 Backend Flush Duration 的值，即可得出一个事务在被执行到下游之前的总排队时长。如果排队时长较高，可以考虑给同步任务分配更多的内存 Quota。

10.9.4.1.5 MQ Sink 监控栏

MQ Sink 监控栏示意如下，该监控栏只有下游为 Kafka 时才有数据。



图 212: TiCDC Summary Dashboard - Transaction Sink metrics

- Worker Send Message Duration Percentile：TiCDC MQ Sink 的 Worker 往下游发送数据的延迟。
- Kafka Ongoing Bytes：TiCDC MQ Sink 往下游发送数据的速率。

10.9.4.1.6 Cloud Storage Sink 监控栏

Cloud Storage Sink 监控栏示意如下，该监控栏只有下游为 Cloud Storage 时才有数据。

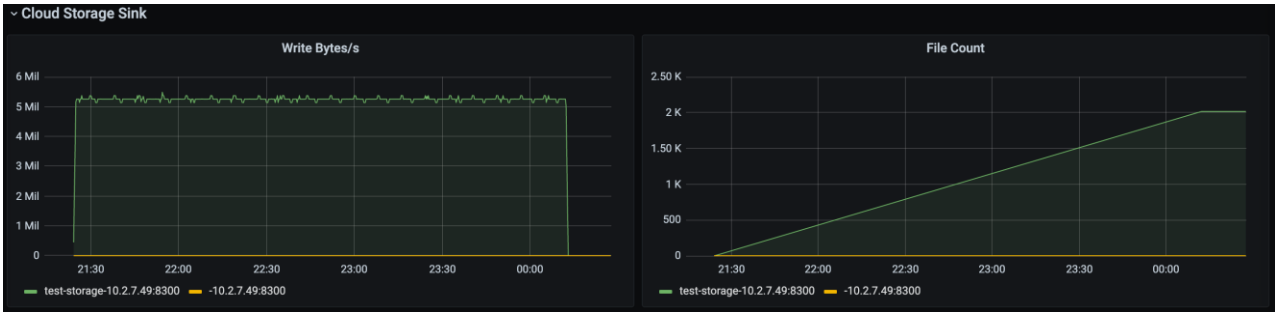


图 213: TiCDC Summary Dashboard - Transaction Sink metrics

- Write Bytes/s：Cloud Storage Sink 模块往下游写数据的速率。
- File Count：Cloud Storage Sink 模块写文件的总数量。

10.9.4.1.7 Redo 监控栏

Redo 监控栏示意如下，该监控栏只有在开启了Redo Log 功能时才有数据。



图 214: TiCDC Summary Dashboard - Transaction Sink metrics

- Redo Write rows/s：Redo 模块每秒写数据的行数。当开启 Redo 功能时，如果发现同步任务的延迟上升，那么可以观察该指标是否和 Puller Output event/s 的值有较大差距。如果是，那么可能是由于Redo 模块的写入能力不足造成延迟上升。
- Redo Write Byte/s：Redo 模块每秒写数据的速率。
- Redo flush log duration：Redo 模块往下游刷写数据的耗时。若该指标值较高，则可能是它影响了同步的速度。

- Redo flushall duration：数据变更停留在Redo 模块中的总时长。

10.9.4.2 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：

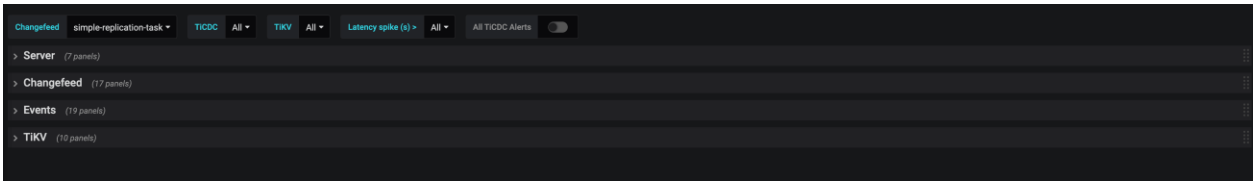


图 215: TiCDC Dashboard - Overview

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

10.9.4.2.1 Server 面板

Server 面板示例如下：



图 216: 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

10.9.4.2.2 Changefeed 面板

Changefeed 面板示例如下：



图 217: 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 218: 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

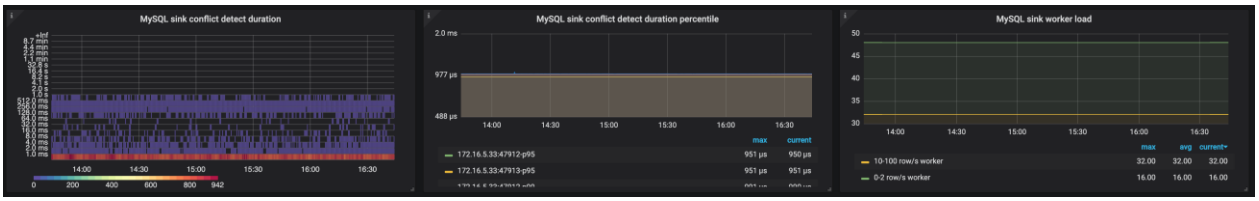


图 219: TiCDC Dashboard - Changefeed metrics 3

- 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

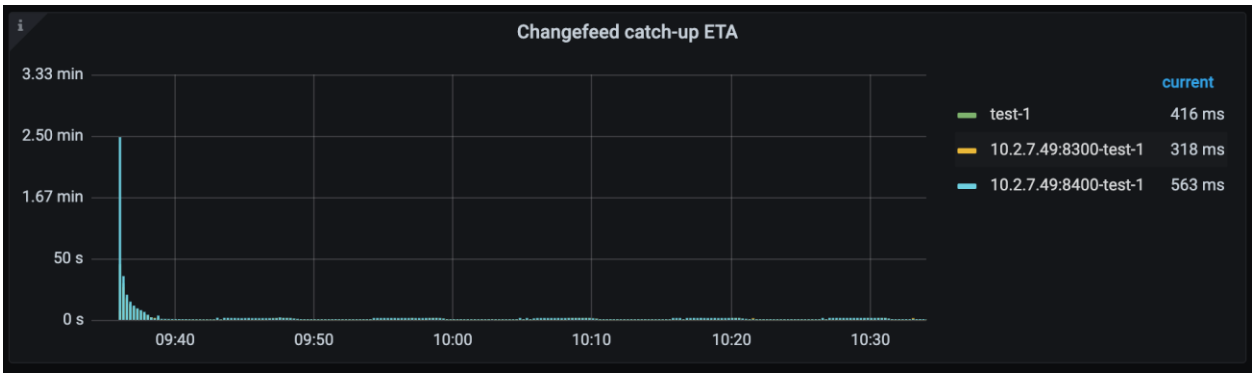


图 220: TiCDC Dashboard - Changefeed metrics 4

- 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。）

10.9.4.2.3 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

10.9.4.2.4 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

10.9.4.3 TiCDC 集群监控报警规则

本文介绍了TiCDC 组件的报警项及相应的报警规则。根据严重级别，报警项按照严重程度由高到低依次为：重要级别(Critical)、警告级别(Warning)。

10.9.4.3.1 重要级别报警项

对于重要级别的报警，需要密切关注异常指标。

`cdc_checkpoint_high_delay`

- 报警规则：

  (time() - ticdc_processor_checkpoint_ts / 1000) > 600

- 规则描述：

  TiCDC 某个同步任务延迟超过 10 分钟。

- 处理方法：

  参考TiCDC 同步任务出现中断的处理方法。

`cdc_resolvedts_high_delay`

- 报警规则：

  (time() - ticdc_processor_resolved_ts / 1000) > 300

- 规则描述：

  TiCDC 某个同步任务的resolved ts 延迟超过 5 分钟。

- 处理方法：

  该告警与同步任务中断类似，可参考TiCDC 同步任务出现中断的处理方法。

`ticdc_processor_exit_with_error_count`

- 报警规则：

  `changes(ticdc_processor_exit_with_error_count[1m])> 0`

- 规则描述：

  TiCDC 某个同步任务报错退出。

- 处理方法：

  参考TiCDC 同步任务出现中断的处理方法。

10.9.4.3.2 警告级别报警项

警告级别的报警是对某一问题或错误的提醒。

`cdc_multiple_owners`

- 报警规则：

  `sum(rate(ticdc_owner_ownership_counter[30s]))>= 2`

- 规则描述：

  TiCDC 集群有多个 owner。

- 处理方法：

  收集TiCDC 日志，定位原因。

`ticdc_mounter_unmarshal_and_mount_time_more_than_1s`

- 报警规则：

  `histogram_quantile(0.9, rate(ticdc_mounter_unmarshal_and_mount_bucket[1m]))* 1000 > 1000`

- 规则描述：

  TiCDC 某一同步任务解码变更数据的耗时超过 1 秒。

- 处理方法：

  收集TiCDC 日志，定位原因。

`cdc_sink_execute_duration_time_more_than_10s`

- 报警规则：

  `histogram_quantile(0.9, rate(ticdc_sink_txn_exec_duration_bucket[1m]))> 10`

- 规则描述：

  TiCDC 某一同步任务写下游执行时间超过 10 秒。

- 处理方法：

  检查下游是否出现问题。

`cdc_processor_checkpoint_tso_no_change_for_1m`

- 报警规则：

  `changes(ticdc_processor_checkpoint_ts[1m])< 1`

- 规则描述：

  TiCDC 某一个同步任务进度超过 1 分钟没有推进。

- 处理方法：

  参考TiCDC 同步任务出现中断的处理方法。

`ticdc_puller_entry_sorter_sort_bucket`

- 报警规则：

  `histogram_quantile(0.9, rate(ticdc_puller_entry_sorter_sort_bucket{}[1m]))> 1`

- 规则描述：

  TiCDC puller entry sorter 排序延迟太高。

- 处理方法：

  收集TiCDC 日志，定位原因。

`ticdc_puller_entry_sorter_merge_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(ticdc_puller_entry_sorter_merge_bucket{}[1m]))> 1
  ```

- 规则描述：

  TiCDC puller entry sorter merge 延迟太高。

- 处理方法：

  收集TiCDC 日志，定位原因。

`tikv_cdc_min_resolved_ts_no_change_for_1m`

- 报警规则：

  ```
  changes(tikv_cdc_min_resolved_ts[1m])< 1 and ON (instance)tikv_cdc_region_resolve_status{
  ↪ status="resolved"} > 0
  ```

- 规则描述：

  TiKV CDC 模块最小的resolved ts 1 分钟没推进。

- 处理方法：

  收集TiKV 日志，定位原因。

`tikv_cdc_scan_duration_seconds_more_than_10min`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(tikv_cdc_scan_duration_seconds_bucket{}[1m]))> 600
  ```

- 规则描述：

  TiKV CDC 模块的增量扫描耗时超过 10 分钟。

- 处理方法：

  收集TiCDC 监控和TiKV 日志，定位原因。

`ticdc_sink_mysql_execution_error`

- 报警规则：

  ```
  changes(ticdc_sink_mysql_execution_error[1m])> 0
  ```

- 规则描述：

  TiCDC 某一同步任务写下游 MySQL 时遇到错误。

- 处理方法：

  MySQL 报错的情况较多，参考TiCDC 故障处理。

`ticdc_memory_abnormal`

- 报警规则：

  `go_memstats_heap_alloc_bytes{job="ticdc"} > 1e+10`

- 规则描述：

  TiCDC 堆内存使用量超过 10 GiB。

- 处理方法：

  收集TiCDC 监控和TiCDC 日志，定位原因。

10.9.4.4 TiCDC 架构设计与原理

10.9.4.4.1 TiCDC 软件架构

TiCDC 集群由多个TiCDC 对等节点组成，是一种分布式无状态的架构设计。TiCDC 集群及节点内部组件的设计如下图所示：

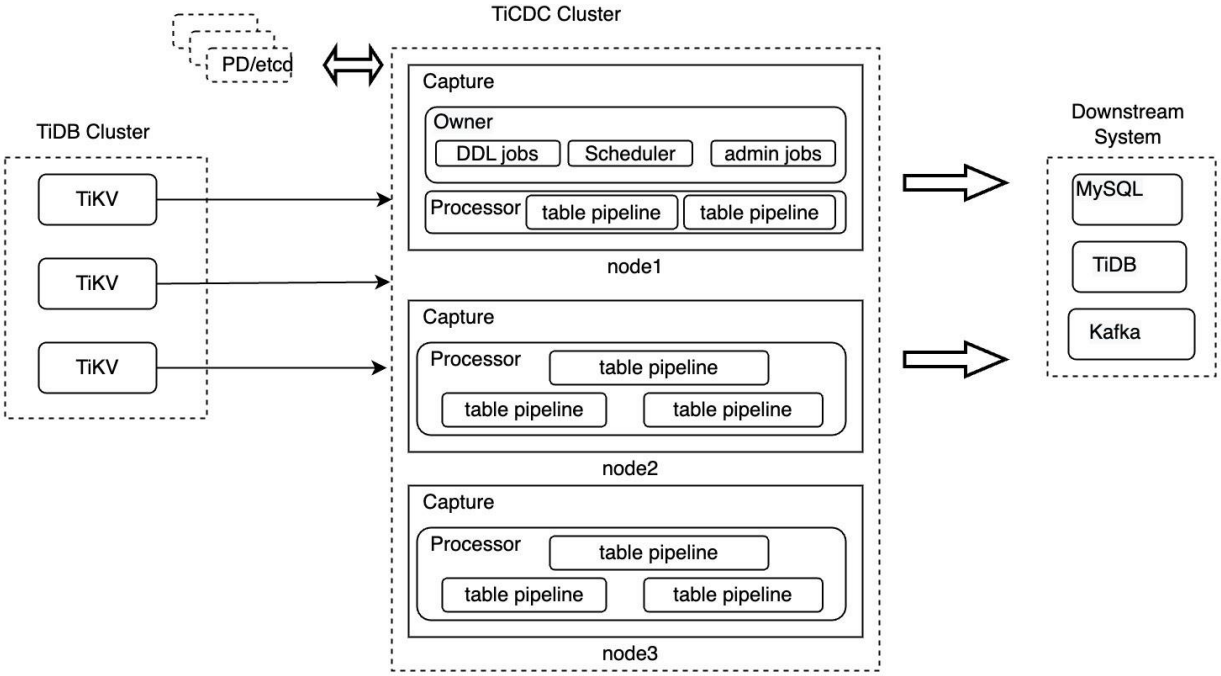


图 221: TiCDC architecture

10.9.4.4.2 组件介绍

上图中，TiCDC 集群由多个运行了TiCDC 实例的节点组成，每个 TiCDC 实例都运行一个 Capture 进程。多个 Capture 进程中会有一个被选举成为 Owner Capture，负责完成 TiCDC 集群的负载调度、DDL 语句同步和其它管理任务。

每个 Capture 进程包含一个或多个 Processor 线程，用于同步上游TiDB 集群中的表数据。由于TiCDC 同步数据的最小单位是表，所以 Processor 是由多条table pipeline 构成的。

每条pipeline 包含 Puller、 Sorter、 Mounter 和 Sink 模块，如下图。

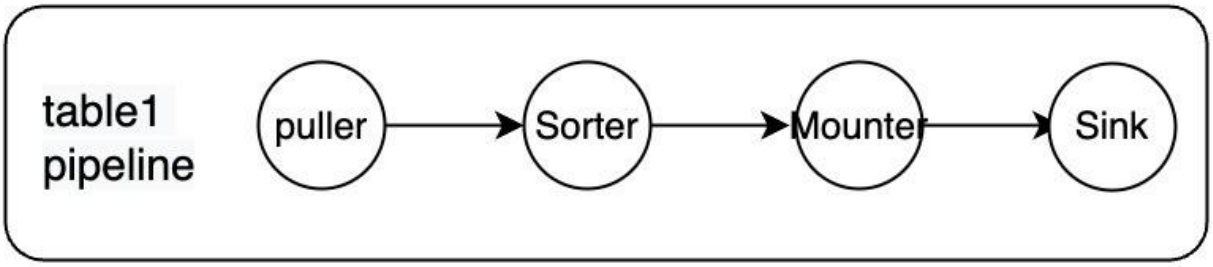


图 222: TiCDC architecture

各个模块之间是串行的关系，组合在一起完成从上游拉取、排序、加载和同步数据到下游的过程，其中：

- Puller：从 TiKV 节点获取 DDL 和行变更信息。
- Sorter：将接收到的变更在 TiCDC 内按照时间戳进行升序排序。
- Mounter：将变更按照对应的 Schema 信息转换成TiCDC 可以处理的格式。
- Sink：将对应的变更应用到下游系统。

为了实现高可用，每个TiCDC 集群都包含多个TiCDC 节点，这些节点定期向 PD 集群中的 etcd 集群汇报自己的状态，并选举出其中一个节点作为TiCDC 集群的 Owner。Owner 采用 etcd 统一存储状态来进行调度，并将调度结果直接写入 etcd。Processor 按照状态完成对应的任务，如果 Processor 所在节点出现异常，集群会将表调度到其他节点。如果 Owner 节点出现异常，其他节点的 Capture 进程会选举出新的 Owner，如下图所示：

TiCDC architecture

10.9.4.4.3 Changefeed 和 Task

TiCDC 中的 Changefeed 和 Task 是两个逻辑概念，前者是分配同步任务的方式，后者是同步任务拆分后的子任务名称，具体描述如下：

- Changefeed：代表一个同步任务，携带了需要同步的表信息，以及对应的下游信息和一些其他的配置信息。
- Task：当 TiCDC 接收到对应的同步任务之后，就会把这个任务拆分为若干个称之为 Task 的子任务，分配给 TiCDC 集群的各个节点上的 Capture 进程进行处理。

例如：

```
cdc cli changefeed create --server="http://127.0.0.1:8300" --sink-uri="kafka://127.0.0.1:9092/cdc
    ′→ -test?kafka-version=2.4.0&partition-num=6&max-message-bytes=67108864&replication-factor
    ′→ =1"
cat changefeed.toml
......
[sink]
dispatchers = [
    {matcher = ['test1.tab1', 'test2.tab2'], topic = "{schema}_{table}"},
    {matcher = ['test3.tab3', 'test4.tab4'], topic = "{schema}_{table}"},
]
```

关于命令中参数的详细释义，参考TiCDC Changefeed 配置参数。

以上命令创建了一个到 Kafka 集群的 Changefeed 任务， 需要同步 `test1.tab1`、 `test1.tab2`、 `test3.tab3` 和 `test4.tab4` 四张表。TiCDC 接收到这个命令之后的处理流程如下：

1. TiCDC 将这个任务发送给 Owner Capture 进程。
2. Owner Capture 进程将这个任务的相关定义信息保存在 PD 的 etcd 中。
3. Owner Capture 将这个任务拆分成若干个 Task，并通知其他 Capture 进程各个 Task 需要完成的任务。
4. 各个 Capture 进程开始从对应的TiKV 节点拉取信息，进行处理后完成同步。

如果将 Changefeed 和 Task 也包含到上文中提及的架构图，完整的TiCDC 架构图如下：

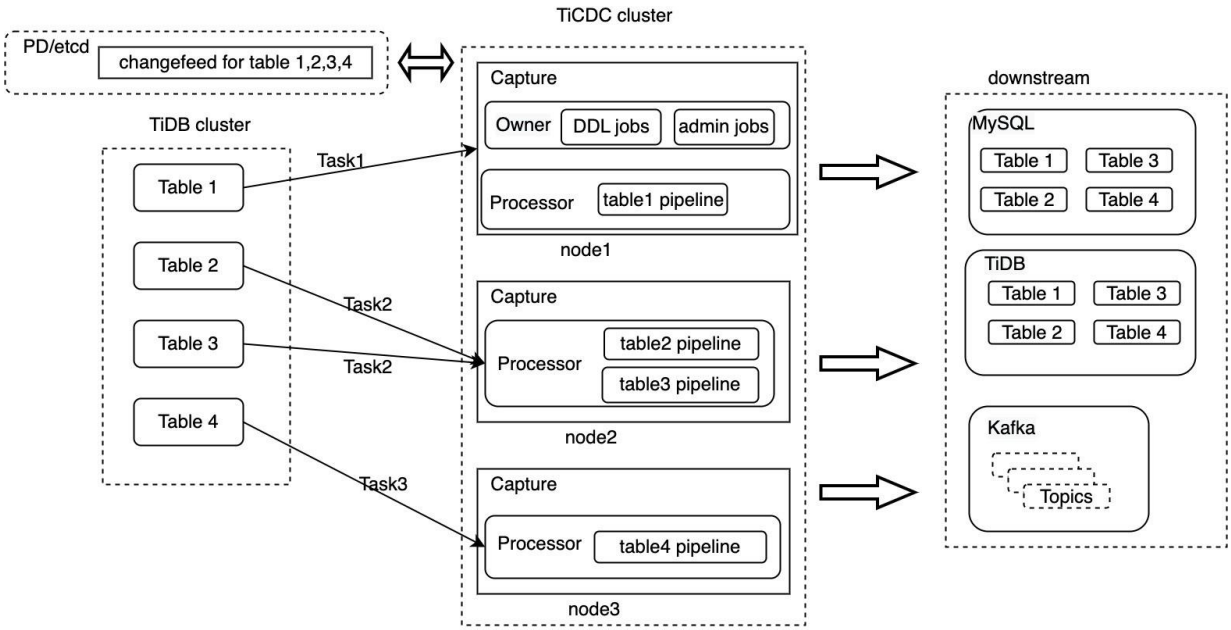


图 223: TiCDC architecture

上图创建了一个 Changefeed，需要同步 4 张表，这个 Changefeed 被拆分成了 3 个任务，均匀的分发到了 TiCDC 集群的 3 个 Capture 节点上，在TiCDC 对这些数据进行了处理之后，数据同步到了下游的系统。

目前TiCDC 支持的下游系统包含 MySQL、 TiDB 和 Kafka。上面的过程只是讲解了 Changefeed 级别数据流转的基本过程，接下来，本文将以处理表 `table1` 的任务 Task1 为例，从更加详细的层面来描述TiCDC 处理数据的过程：

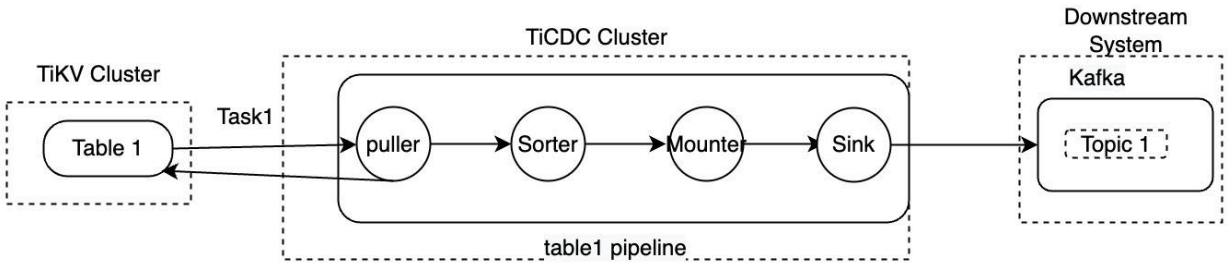


图 224: TiCDC architecture

1. 推流：发生数据改变时，TiKV 集群将数据主动推送给 Puller 模块。
2. 增量扫：Puller 模块在发现收到的数据改变不连续的时候，向TiKV 节点主动拉取需要的数据。
3. 排序：Sorter 模块对获取的数据按照时间进行排序，并将排好序的数据发送给 Mounter。
4. 装载：Mounter 模块收到数据变更后，根据表的 schema 信息，将数据变更装载成TiCDC sink 可以理解的格式。
5. 同步：Sink 模块根据下游的类型将数据变更同步到下游。

由于TiCDC 的上游是支持事务的分布式关系型数据库TiDB，在同步数据的时候，如何保证数据的一致性，以及在同步多张表的时候，如何保证事务的一致性，都是很大的挑战。下面的章节会介绍TiCDC 在确保事务特性时所使用的关键技术和概念。

10.9.4.4.4 TiCDC 关键概念

当 TiCDC 对应的下游系统是关系型数据库时，TiCDC 在同步数据的时候会确保单表事务的数据一致性，以及多表事务的最终一致性。另外，TiCDC 会确保上游TiDB 集群发生过的数据变更能够被TiCDC 向任意下游最少同步一次。

架构相关概念

- Capture：TiCDC 节点的运行进程，多个 Capture 进程构成了TiCDC 集群，Capture 进程负责TiKV 的数据变更的同步，包括接收和主动拉取两种方式，并向下游同步数据。
- Capture Owner：是一种 Capture 的角色，每个TiCDC 集群同一时刻最多只存在一个 Capture Owner 角色，负责集群内部的调度。
- Processor：是 Capture 内部的逻辑线程，每个 Processor 负责处理同一个同步流中一个或多个table 的数据。一个 Capture 节点可以运行多个 Processor。
- ChangeFeed：一个由用户启动的从上游 TiDB 同步到下游的任务，其中包含多个 Task，Task 会分布在不同的 Capture 节点进行数据同步处理。

时间戳相关概念

由于 TiCDC 需要确保数据被至少一次同步到下游， 并且确保一定的一致性， 因此引入了一系列时间戳（Timestamp，简称TS）来对数据同步的状态进行描述。

ResolvedTS

这个时间戳在TiKV和TiCDC中都存在。

- TiKV 节点中的ResolvedTS：代表某一个Region leader 上开始时间最早的事务的开始时间，即：ResolvedTS = max(ResolvedTS, min(StartTS))。因为每个 TiDB 集群包含多个TiKV 节点，所有TiKV 节点上的Region leader 的ResolvedTS 的最小值，被称为Global ResolvedTS。TiDB 集群确保Global ResolvedTS 之前的事务都被提交了，或者可以认为这个时间戳之前不存在未提交的事务。
- TiCDC 节点中的ResolvedTS：
  - table ResolvedTS：每张表都会有的表级别ResolvedTS，表示这张表已经从TiKV 接收到的数据改变的最低水位，可以简单的认为和TiKV 节点上这张表的各个Region 的 ResolvedTS 的最小值是相同的。
  - global ResolvedTS：各个 TiCDC 节点上的 Processor ResolvedTS 的最小值。由于 TiCDC 每个节点上都会存在一个或多个 Processor，每个 Processor 又对应多个table pipeline。

  对于 TiCDC 节点来说，TiKV 节点发送过来的 ResolvedTS 信息是一种特殊的事件，它只包含一个格式为 <resolvedTS: 时间戳> 的特殊事件。通常情况下，ResolvedTS 满足以下约束：

```
table ResolvedTS >= global ResolvedTS
```

CheckpointTS

这个时间戳只在TiCDC 中存在，它表示TiCDC 已经同步给下游的数据的最低水位线，即TiCDC 认为在这个时间戳之前的数据已经被同步到下游系统了。由于TiCDC 同步数据的单位是表，所以table CheckpointTS 表示表级别的同步数据的水位线，Processor CheckpointTS 表示各个 Processor 中最小的table CheckpointTS；Global checkpointTS 表示各个 Processor checkpointTS 中最低的checkpointTS。通常情况下，Checkpoint TS 满足以下约束：

```
table CheckpointTS >= global CheckpointTS
```

因为TiCDC 只会复制小于 global ResolvedTS 的数据到下游，所以存在下面的约束：

```
table ResolvedTS >= global ResolvedTS >= table CheckpointTS >= global CheckpointTS
```

随着数据的改变和事务的提交，TiKV 节点上的ResolvedTS 会不断的向前推进，TiCDC 节点的 Puller 模块也会不断的收到TiKV 推流过来的数据，并且根据收到的信息决定是否执行增量扫的过程，从而确保数据改变都能够被发送到TiCDC 节点上。Sorter 模块则负责将 Puller 模块收到的信息按照时间戳进行升序排序，从而确保数据在表级别是满足一致性的。接下来，Mounter 模块把上游的数据改变装配成 Sink 模块可以消费的格式，并发送给 Sink，而 Sink 则负责把 CheckpointTS 到 ResolvedTS 之间的数据改变，按照发生的TS 顺序同步到下游，并在下游返回后推进 CheckpointTS。

上面的内容只介绍了和 DML 语句相关的数据改变，并没有包含 DDL 相关的内容。下面对 DDL 语句相关的关键概念进行介绍。

Barrier TS

当系统发生 DDL 变更或者用户使用了TiCDC 的 Syncpoint 时会产生的一个时间戳。

- DDL：这个时间戳会用来确保在这个 DDL 语句之前的改变都被应用到下游，之后执行对应的 DDL 语句，在 DDL 语句同步完成之后再同步其他的数据改变。由于 DDL 语句的处理是 Owner 角色的 Capture 负责的，DDL 语句对应的 Barrier TS 只会由 Owner 节点产生。

- Syncpoint：当你启用 TiCDC 的 Syncpoint 特性后，TiCDC 会根据你指定的间隔产生一个 Barrier TS。等所有表都同步到这个 Barrier TS 后，TiCDC 将此刻的 global CheckpointTS 作为 Primary Ts 插入下游的TiDB 记录tsMap 信息的表中，然后TiCDC 才会继续向下游同步数据。

一个 Barrier TS 被生成后, TiCDC 会保证只有小于 Barrier TS 的数据会被复制到下游，并且保证小于 Barrier TS 的数据全部被复制到下游之前，同步任务不会再推进。Owner 不断地比较 global CheckpointTS 和 Barrier TS 的大小，确定小于 Barrier TS 的数据是否已经被同步完成。如果 global CheckpointTS = Barrier TS，执行完对应的操作（如 DDL 或者记录 global CheckpointTS 到下游）后，同步继续；否则需要继续等待所有小于 Barrier TS 数据的同步完成。

10.9.4.4.5 主要流程

本章节将介绍TiCDC 软件的常见操作所对应的主要流程，以帮助你更好的理解TiCDC 的工作原理。

注意，下面所描述的启动流程只存在于TiCDC 进程内部，对于用户是完全透明的。因此，你在启动TiCDC 进程时无需关心自己启动的是什么节点。

启动TiCDC 节点

- 启动非 Owner 的 TiCDC 节点：
    1. 启动 Capture 进程。
    2. 启动processor。
    3. 接收 Owner 下发的 Task 调度命令。
    4. 根据调度命令启动或停止tablePipeline。
- 启动 Owner 的 TiCDC 节点：
    1. 启动 Capture 进程。
    2. 当选 Owner 并启动对应的线程。
    3. 读取 Changefeed 信息。
    4. 启动 Changefeed 管理逻辑。
    5. 根据 Changefeed 配置和最新 CheckpointTS，读取 TiKV 中的 schema 信息，确定需要被同步的表。
    6. 读取各 Processor 当前同步的表的列表，分发需要添加的表。
    7. 更新进度信息。

停止TiCDC 节点

通常来说，停止TiCDC 节点是为了对TiCDC 进行升级，或者对TiCDC 所在的节点进行一些计划的维护操作。停止 TiCDC 节点的流程如下：

1. 节点收到停止的信号。
2. 节点将自己的服务状态置为不可用状态。
3. 节点停止接收新的复制表任务。
4. 通知 Owner 节点将自己节点的数据复制任务转移到其他节点。
5. 复制任务转移到其他节点后，该节点停止完成。

10.9.4.5 TiCDC Server 配置

本文介绍TiCDC 的命令行参数和配置文件定义。

10.9.4.5.1 cdc server 命令行参数说明

对于 cdc server 命令中可用选项解释如下：

- addr：TiCDC 的监听地址，提供服务的 HTTP API 查询地址和 Prometheus 查询地址，默认为 127.0.0.1:8300。
- advertise-addr：TiCDC 对外开放地址，供客户端访问。如果未设置该参数值，地址默认与 addr 相同。
- pd：TiCDC 监听的 PD 节点地址，用 , 来分隔多个 PD 节点地址。
- config：可选项，表示 TiCDC 使用的配置文件地址。TiCDC 从 v5.0.0 开始支持该选项，TiUP 从 v1.4.0 开始支持在部署TiCDC 时使用该配置。配置文件的格式说明详见：TiCDC Changefeed 配置参数
- data-dir：指定 TiCDC 使用磁盘储存文件时的目录。目前TiCDC 内部的排序引擎和redo log 等特性会使用该目录储存临时文件，建议确保该目录所在设备的可用空间大于等于 500 GiB。如果你使用TiUP，本选项可以通过配置cdc_servers 中的 data_dir 来指定或默认使用 global 中 data_dir 路径。
- gc-ttl：TiCDC 在 PD 设置的服务级别GC safepoint 的 TTL (Time To Live) 时长，和TiCDC 同步任务所能够停滞的时长。单位为秒，默认值为 86400，即 24 小时。注意：TiCDC 同步任务的停滞会影响TiCDC GC safepoint 的推进，即会影响上游TiDB GC 的推进，详情可以参考TiCDC GC safepoint 的完整行为。
- log-file：TiCDC 进程运行时日志的输出地址，未设置时默认为标准输出(stdout)。
- log-level：TiCDC 进程运行时的日志级别，默认为 "info"。
- ca：TiCDC 创建 TLS 连接时使用的 CA 证书文件路径，PEM 格式，可选。
- cert：TiCDC 创建 TLS 连接时使用的证书文件路径，PEM 格式，可选。
- cert-allowed-cn：TiCDC 创建 TLS 连接时使用的通用名称文件路径，可选。
- key：TiCDC 创建 TLS 连接时使用的证书密钥文件路径，PEM 格式，可选。
- tz：TiCDC 服务使用的时区。TiCDC 在内部转换 TIMESTAMP 等时间数据类型和向下游同步数据时使用该时区，默认为进程运行本地时区。（注意如果同时指定 tz 参数和 sink-uri 中的 time-zone 参数，TiCDC 进程内部使用 tz 指定的时区，sink 向下游执行时使用 time-zone 指定的时区）
- cluster-id：TiCDC 集群的ID。可选，默认值为 default。cluster-id 是 TiCDC 集群的唯一标识，拥有相同 cluster-id 的 TiCDC 节点同属一个集群。长度最大为 128，需要符合正则表达式 ^[a-zA-Z0-9]+(-[a-zA-Z0-9]+)*$，且不能是以下值：owner，capture，task，changefeed，job，meta。

10.9.4.5.2 cdc server 配置文件说明

对于 cdc server 命令中config 参数指定的配置文件说明如下：

```
addr = "127.0.0.1:8300"
advertise-addr = ""
log-file = ""
log-level = "info"
data-dir = ""
gc-ttl = 86400 # 24 h
tz = "System"
cluster-id = "default"

[security]
  ca-path = ""
  cert-path = ""
  key-path = ""
```

```
capture-session-ttl = 10 # 10s
owner-flush-interval = 50000000 # 50 ms
processor-flush-interval = 50000000 # 50 ms
per-table-memory-quota = 10485760 # 10 MiB

[log]
  error-output = "stderr"
  [log.file]
    max-size = 300 # 300 MiB
    max-days = 0
    max-backups = 0


#### [kv-client]
####    worker-concurrent = 8
####    worker-pool-size = 0
####    region-scan-limit = 40
####    region-retry-duration = 60000000000
```

10.9.4.6 TiCDC Changefeed 命令行参数和配置参数

10.9.4.6.1 TiCDC Changefeed 命令行参数

本章节将以创建同步任务为例，介绍TiCDC Changefeed 的命令行参数：

```
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"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"upstream_id":7178706266519722477,"namespace":"default","id":"simple-replication-task","
    ′→ sink_uri":"mysql://root:xxxxx@127.0.0.1:4000/?time-zone=","create_time":"2022-12-19T15
    ′→ :05:46.679218+08:00","start_ts":438156275634929669,"engine":"unified","config":{"
    ′→ case_sensitive":true,"enable_old_value":true,"force_replicate":false,"
    ′→ ignore_ineligible_table":false,"check_gc_safe_point":true,"enable_sync_point":true,"
    ′→ bdr_mode":false,"sync_point_interval":30000000000,"sync_point_retention":3600000000000,"
    ′→ filter":{"rules":["test.*"],"event_filters":null},"mounter":{"worker_num":16},"sink":{"
    ′→ protocol":"","schema_registry":"","csv":{"delimiter":",","quote":"\"","null":"\\N","
    ′→ include_commit_ts":false},"column_selectors":null,"transaction_atomicity":"none","
    ′→ encoder_concurrency":16,"terminator":"\r\n","date_separator":"none","
    ′→ enable_partition_separator":false},"consistent":{"level":"none","max_log_size":64,"
    ′→ flush_interval":2000,"storage":""}},"state":"normal","creator_version":"v6.5.0"}
```

- `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为ID。

- `--sink-uri`：同步任务下游的地址，需要按照以下格式进行配置，目前scheme 支持mysql、tidb 和kafka。

```
[scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
```

  URI 中包含特殊字符时，如`! * ' ( ) ; : @ & = + $ , / ? % # [ ]`，需要对 URI 特殊字符进行转义处理。你可以在 URI Encoder 中对 URI 进行转义。

- `--start-ts`：指定 changefeed 的开始TSO。TiCDC 集群将从这个TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标TSO。TiCDC 集群拉取数据直到这个TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件。

10.9.4.6.2 TiCDC Changefeed 配置文件说明

本章节详细介绍了同步任务的配置。

```
#### 指定该 Changefeed 在 Capture Server 中内存配额的上限。对于超额使用部分，
#### 会在运行中被 Go runtime 优先回收。默认值为 `1073741824`，即 1 GB。
#### memory-quota = 1073741824

#### 指定配置文件中涉及的库名、表名是否为大小写敏感
#### 该配置会同时影响 filter 和 sink 相关配置，默认为 true
case-sensitive = true

#### 是否输出 old value，从 v4.0.5 开始支持，从 v5.0 开始默认为 true
enable-old-value = true

#### 是否开启 Syncpoint 功能，从 v6.3.0 开始支持，该功能默认关闭。
#### 从 v6.4.0 开始，使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者
    ↪ SUPER 权限。
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### enable-sync-point = false

#### Syncpoint 功能对齐上下游 snapshot 的时间间隔
#### 配置格式为 h m s，例如 "1h30m30s"
#### 默认值为 10m，最小值为 30s
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### sync-point-interval = "5m"

#### Syncpoint 功能在下游表中保存的数据的时长，超过这个时间的数据会被清理
#### 配置格式为 h m s，例如 "24h30m30s"
#### 默认值为 24h
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### sync-point-retention = "1h"

[mounter]
```

```
#### mounter 解码 KV 数据的线程数，默认值为 16
#### worker-num = 16

[filter]
#### 忽略指定 start_ts 的事务
#### ignore-txn-start-ts = [1, 2]

#### 过滤器规则
####  过滤规则语法：https://docs.pingcap.com/zh/tidb/stable/table-filter#表库过滤语法
rules = ['*.*', '!test.*']

#### 忽略特定 start_ts 的事务
#### 默认值为空列表。
#### IgnoreTxnStartTs = []

#### 事件过滤器规则
#### 事件过滤器的详细配置规则可参考：https://docs.pingcap.com/zh/tidb/stable/ticdc-filter
#### 第一个事件过滤器规则
#### [[filter.event-filters]]
#### matcher = ["test.worker"] # matcher 是一个白名单，表示该过滤规则只应用于 test 库中的 worker
    ↪ 表
#### ignore-event = ["insert"] # 过滤掉 insert 事件
#### ignore-sql = ["^drop", "add column"] # 过滤掉以 "drop" 开头或者包含 "add column" 的 DDL
#### ignore-delete-value-expr = "name = 'john'" # 过滤掉包含 name = 'john' 条件的 delete DML
#### ignore-insert-value-expr = "id >= 100" # 过滤掉包含 id >= 100 条件的 insert DML
#### ignore-update-old-value-expr = "age < 18" # 过滤掉旧值 age < 18 的 update DML
#### ignore-update-new-value-expr = "gender = 'male'" # 过滤掉新值 gender = 'male' 的 update DML

#### 第二个事件过滤器规则
#### [[filter.event-filters]]
#### matcher = ["test.fruit"] # 该事件过滤器只应用于 test.fruit 表
#### ignore-event = ["drop table", "delete"] # 忽略 drop table 的 DDL 事件和 delete 类型的 DML
    ↪ 事件
#### ignore-sql = ["^drop table", "alter table"] # 忽略以 drop table 开头的，或者包含 alter table
    ↪  的 DDL 语句
#### ignore-insert-value-expr = "price > 1000 and origin = 'no where'" # 忽略包含 price > 1000 和
    ↪  origin = 'no where' 条件的 insert DML

[scheduler]
#### 将表按 Region 个数划分成多个同步范围，这些范围可由多个 TiCDC 节点同步。
#### 注意：该功能只在 Kafka changefeed 上生效，暂不支持 MySQL changefeed。
#### 默认为 "false"。设置为 "true" 以打开该功能。
enable-table-across-nodes = false

#### 打开该功能后，该功能会对 Region 个数大于 `region-threshold` 值的表生效。
```

```
region-threshold = 100000

#### 打开该功能后，该功能会对每分钟修改行数大于 `write-key-threshold` 值的表生效。
#### 注意：
#### * `write-key-threshold` 参数默认值为 0，代表该功能默认不会按表的修改行数来切分表的同步范围。
#### * 你可以根据集群负载来配置该参数，如 30000，代表当表每分钟的更新行数超过 30000 时，
    '→ 该功能将会切分表的同步范围。
#### * 当 `region-threshold` 和 `write-key-threshold` 同时配置时，
####   TiCDC 将优先检查修改行数是否大于 `write-key-threshold`，
####   如果不超过，则再检查 Region 个数是否大于 `region-threshold`。
write-key-threshold = 0

[sink]
#### 对于 MQ 类的 Sink，可以通过 dispatchers 配置 event 分发器
#### 支持 partition 及 topic（从 v6.1 开始支持）两种 event 分发器。二者的详细说明见下一节。
#### matcher 的匹配语法和过滤器规则语法相同，matcher 匹配规则的详细说明见下一节。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### dispatchers = [
####     {matcher = ['test1.*', 'test2.*'], topic = "Topic 表达式 1", partition = "ts" },
####     {matcher = ['test3.*', 'test4.*'], topic = "Topic 表达式 2", partition = "index-value" },
####     {matcher = ['test1.*', 'test5.*'], topic = "Topic 表达式 3", partition = "table"},
####     {matcher = ['test6.*'], partition = "ts"}
#### ]

#### protocol 用于指定传递到下游的协议格式
#### 当下游类型是 Kafka 时，支持 canal-json、avro 两种协议。
#### 当下游类型是存储服务时，目前仅支持 canal-json、csv 两种协议。
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### protocol = "canal-json"

#### 以下三个配置项仅在同步到存储服务的 sink 中使用，在 MQ 和 MySQL 类 sink 中无需设置。
#### 换行符，用来分隔两个数据变更事件。默认值为空，表示使用 "\r\n" 作为换行符。
#### terminator = ''

#### 文件路径的日期分隔类型。可选类型有 `none`、`year`、`month` 和 `day`。默认值为 `none`，
    '→ 即不使用日期分隔。详见 <https://docs.pingcap.com/zh/tidb/dev/ticdc-sink-to-cloud-storage#
    '→ 数据变更记录>。
#### 注意：该参数只有当下游为存储服务时，才会生效。
date-separator = 'none'

#### 是否使用 partition 作为分隔字符串。默认值为 true，即一张表中各个 partition
    '→ 的数据会分不同的目录来存储。建议保持该配置项为 true 以避免下游分区表可能丢数据的问题 <
    '→ https://github.com/pingcap/tiflow/issues/8581>。使用示例详见 <https://docs.pingcap.com/zh
    '→ /tidb/dev/ticdc-sink-to-cloud-storage#数据变更记录>。
#### 注意：该参数只有当下游为存储服务时，才会生效。
```

```
enable-partition-separator = true

#### Schema 注册表的 URL。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### schema-registry = "http://localhost:80801/subjects/{subject-name}/versions/{version-number}/
    ↪ schema"

#### 编码数据时所用编码器的线程数。
#### 默认值为 16。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### encoder-concurrency = 16

#### 是否开启 Kafka Sink V2。Kafka Sink V2 内部使用 kafka-go 实现。
#### 默认值为 false。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### enable-kafka-sink-v2 = false

#### 是否只向下游同步有内容更新的列。
#### 默认值为 false。
#### 注意：该参数只有当下游为消息队列，并且使用 Open Protocol 或 Canal-JSON 时，才会生效。
#### only-output-updated-columns = false

#### 从 v6.5.0 开始，TiCDC 支持以 CSV 格式将数据变更记录保存至存储服务中，在 MQ 和 MySQL 类 sink
    ↪ 中无需设置。
#### [sink.csv]
#### 字段之间的分隔符。必须为 ASCII 字符，默认值为 `,`。
#### delimiter = ','
#### 用于包裹字段的引号字符。空值代表不使用引号字符。默认值为 `"`。
#### quote = '"'
#### CSV 中列为 NULL 时将以什么字符来表示。默认值为 `\N`。
#### null = '\N'
#### 是否在 CSV 行中包含 commit-ts。默认值为 false。
#### include-commit-ts = false

#### sink.consistent 中的字段用于配置 Changefeed 的数据一致性。详细的信息，请参考 <https://docs.
    ↪ pingcap.com/tidb/stable/ticdc-sink-to-mysql#eventually-consistent-replication-in-disaster
    ↪ -scenarios>。
#### 注意：一致性相关参数只有当下游为数据库并且开启 redo log 功能时，才会生效。
[sink.consistent]
#### 数据一致性级别。默认值为 "none"，可选值为 "none" 和 "eventual"。
#### 设置为 "none" 时将关闭 redo log。
level = "none"
#### redo log 的最大日志大小，单位为 MB。默认值为 64。
max-log-size = 64
#### 两次 redo log 刷新的时间间隔，单位为毫秒。默认值为 2000。
```

```
flush-interval = 2000
#### redo log 使用存储服务的 URI。默认值为空。
storage = ""
#### 是否将 redo log 存储到文件中。默认值为 false。
use-file-backend = false

[integrity]
#### 是否开启单行数据的 Checksum 校验功能，默认值为 "none"，即不开启。可选值为 "none" 和 "
    ↪ correctness"。
integrity-check-level = "none"
#### 当单行数据的 Checksum 校验失败时，Changefeed 打印错误行数据相关日志的级别。默认值为 "warn"，
    ↪ 可选值为 "warn" 和 "error"。
corruption-handle-level = "warn"
```

10.9.4.7 输出数据协议

10.9.4.7.1 TiCDC Avro Protocol

Avro 是由 Apache Avro™ 定义的一种数据交换格式协议，Confluent Platform 选择它作为默认的数据交换格式。通过本文，你可以了解TiCDC 对 Avro 数据格式的实现，包括TiDB 扩展字段、Avro 数据格式定义，以及和 Confluent Schema Registry 的交互。

> **警告：**
>
> 当开启Old Value 功能时 (`enable-old-value = true`)，Avro 协议数据格式无法输出更新事件的旧值。
>
> 具体原因请参考TiCDC 在开启 Old Value 功能后更新事件格式有何变化？

使用Avro

当使用 Message Queue (MQ) 作为下游 Sink 时，你可以在 `sink-uri` 中指定使用Avro。TiCDC 获取TiDB 的 DML 事件，并将这些事件封装到Avro Message，然后发送到下游。当 Avro 检测到 schema 变化时，会向 Schema Registry 注册最新的 schema。

使用Avro 时的配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro" --sink-uri
    ↪ ="kafka://127.0.0.1:9092/topic-name?protocol=avro" --schema-registry=http
    ↪ ://127.0.0.1:8081 --config changefeed_config.toml
```

```
[sink]
dispatchers = [
 {matcher = ['*.*'], topic = "tidb_{schema}_{table}"},
]
```

`--schema-registry` 的值支持https 协议和username:password 认证，比如`--schema-registry=https://username:password@schema-registry-uri.com`，username 和password 必须经过 URL 编码。

TiDB 扩展字段

默认情况下，Avro 只收集在 DML 事件中发生数据变更的行的所有数据信息，不收集数据变更的类型和TiDB专有的 CommitTS 事务唯一标识信息。为了解决这个问题，TiCDC 在 Avro 协议格式中附加了TiDB 扩展字段。当 `sink-uri` 中设置 `enable-tidb-extension` 为 `true`（默认为 `false`）后，TiCDC 生成Avro 消息时会新增三个字段：

- `_tidb_op`：DML 的类型，“c” 表示插入，“u” 表示更新。
- `_tidb_commit_ts`：事务唯一标识信息。
- `_tidb_commit_physical_time`：事务标识信息中现实时间的时间戳。

配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro-enable-
    ↪ extension" --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=avro&enable-tidb-
    ↪ extension=true" --schema-registry=http://127.0.0.1:8081 --config changefeed_config.toml
```

```
shell [sink] dispatchers = [ {matcher = ['*.*'], topic = "tidb_{schema}_{table}"}, ]
```

数据格式定义

TiCDC 会将一个 DML 事件转换为一个 kafka 事件，其中事件的 key 和 value 都按照Avro 协议进行编码。

Key 数据格式

```
{
    "name":"{{TableName}}",
    "namespace":"{{Namespace}}",
    "type":"record",
    "fields":[
        {{ColumnValueBlock}},
        {{ColumnValueBlock}},
    ]
}
```

- `{{TableName}}` 是事件来源表的名称。
- `{{Namespace}}` 是 Avro 的命名空间。
- `{{ColumnValueBlock}}` 是每列数据的格式定义。

Key 中的 `fields` 只包含主键或唯一索引列。

Value 数据格式

```
{
    "name":"{{TableName}}",
    "namespace":"{{Namespace}}",
    "type":"record",
```

```
    "fields":[
        {{ColumnValueBlock}},
        {{ColumnValueBlock}},
    ]
}
```

Value 数据格式默认与 Key 数据格式相同，但是Value 的 `fields` 中包含了所有的列，而不仅仅是主键列。

如果开启了TiDB 扩展字段，那么Value 数据格式将会变成：

```
{
    "name":"{{TableName}}",
    "namespace":"{{Namespace}}",
    "type":"record",
    "fields":[
        {{ColumnValueBlock}},
        {{ColumnValueBlock}},
        {
            "name":"_tidb_op",
            "type":"string"
        },
        {
            "name":"_tidb_commit_ts",
            "type":"long"
        },
        {
            "name":"_tidb_commit_physical_time",
            "type":"long"
        }
    ]
}
```

相比于不打开TiDB 扩展字段选项，多出了 `_tidb_op`,`_tidb_commit_ts`,`_tidb_commit_physical_time` 三个字段的定义。

Column 数据格式

Column 数据格式即 Key/Value 数据格式中的 `{{ColumnValueBlock}}` 部分，TiCDC 会根据 SQL Type 生成对应的 Column 数据格式。基础的 Column 数据格式是：

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{ "tidb_ty
            pe":"{{TIDB_TYPE}}"
        },
        "type":"{{AVRO_TYPE}}"
    }
```

```
}
```

如果一列可以为 NULL，那么 Column 数据格式是：

```
{
    "default":null,
    "name":"{{ColumnName}}",
    "type":[
        "null",
        {
            "connect.parameters":{ "tidb_type"
                :"{{TIDB_TYPE}}"
            },
            "type":"{{AVRO_TYPE}}"
        }
    ]
}
```

- `{{ColumnName}}` 表示列名。
- `{{TIDB_TYPE}}` 表示对应到TiDB 中的类型，与原始的 SQL Type 不是一一对应关系。
- `{{AVRO_TYPE}}` 表示 avro spec 中的类型。

SQL TYPE	TIDB_TYPE	AVRO_TYPE	说明
BOOL	INT	int	
TINYINT	INT	int	当 TINYINT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED。
SMALLINT	INT	int	当 SMALLINT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED.
MEDIUMINT	INT	int	当 MEDIUMINT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED。
INT	INT	int	当 INT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED，AVRO_TYPE 为 long。
BIGINT	BIGINT	long	当 BIGINT 为无符号值时，TIDB_TYPE 为 BIGINT UNSIGNED。当 `avro-bigint-unsigned-handling-mode` 为 string 时，AVRO_TYPE 也为 string。
TINYBLOB	BLOB	bytes	
BLOB	BLOB	bytes	
MEDIUMBLOB	BLOB	bytes	
LONGBLOB	BLOB	bytes	
BINARY	BLOB	bytes	
VARBINARY	BLOB	bytes	
TINYTEXT	TEXT	string	
TEXT	TEXT	string	
MEDIUMTEXT	TEXT	string	
LONGTEXT	TEXT	string	
CHAR	TEXT	string	
VARCHAR	TEXT	string	
FLOAT	FLOAT	double	

SQL TYPE	TIDB_TYPE	AVRO_TYPE	说明
DOUBLE	DOUBLE	double	
DATE	DATE	string	
DATETIME	DATETIME	string	
TIMESTAMP	TIMESTAMP	string	
TIME	TIME	string	
YEAR	YEAR	int	
BIT	BIT	bytes	
JSON	JSON	string	
ENUM	ENUM	string	
SET	SET	string	
DECIMAL	DECIMAL	bytes	当 `avro-decimal-handling-mode` 为 string 时，AVRO_TYPE 也为 string。

对于 Avro 协议，另外两个 `sink-uri` 参数 `avro-decimal-handling-mode` 和 `avro-bigint-unsigned-handling-mode` 也会影响 Column 数据格式:

- `avro-decimal-handling-mode` 决定了如何处理 DECIMAL 字段，它有两个选项：
  - string：Avro 将 DECIMAL 字段以 string 的方式处理。
  - precise：Avro 将 DECIMAL 字段以字节的方式处理。
- `avro-bigint-unsigned-handling-mode` 决定了如何处理 BIGINT UNSIGNED 字段，它有两个选项：
  - string：Avro 将 BIGINT UNSIGNED 字段以 string 的方式处理。
  - long：Avro 将 BIGINT UNSIGNED 字段以 64 位有符号整数处理，大于 9223372036854775807 的值会发生溢出。

配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro-string-
    ↪ option" --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=avro&avro-decimal-handling
    ↪ -mode=string&avro-bigint-unsigned-handling-mode=string" --schema-registry=http
    ↪ ://127.0.0.1:8081 --config changefeed_config.toml
```

```
[sink]
dispatchers = [
 {matcher = ['*.*'], topic = "tidb_{schema}_{table}"},
]
```

大多数的 SQL Type 都会映射成基础的 Column 数据格式，但有一些类型会在基础数据格式上拓展，提供更多的信息。

BIT(64)

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{ "tidb_type":"BI
            T",
            "length":"64"
        },
        "type":"bytes"
    }
}
```

ENUM/SET(a,b,c)

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{ "tidb_type":"EN
            UM/SET", "allowed":"a,b,c"
        },
        "type":"string"
    }
}
```

DECIMAL(10, 4)

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{ "tidb_type":"DE
            CIMAL",
        },
        "logicalType":"decimal",
        "precision":10,
        "scale":4,
        "type":"bytes"
    }
}
```

DDL 事件与 Schema 变更

Avro 并不会向下游生成DDL 事件。Avro 会在每次DML 事件发生时检测是否发生schema 变更，如果发生了schema 变更，Avro 会生成新的 schema，并尝试向 Schema Registry 注册。注册时，Schema Registry 会做兼容性检测，如果此次 schema 变更没有通过兼容性检测，注册将会失败，TiCDC 并不会尝试解决 schema 的兼容性问题。

同时，即使 schema 变更通过兼容性检测并成功注册新版本，数据的生产者和消费者可能仍然需要升级才能正确工作。

比如，Confluent Schema Registry 默认的兼容性策略是 BACKWARD，在这种策略下，如果你在源表增加一个非空列，Avro 在生成新 schema 向 Schema Registry 注册时将会因为兼容性问题失败，这个时候changefeed 将会进入 error 状态。

如需了解更多 schema 相关信息，请参阅 Schema Registry 的相关文档。

Topic 分发

Schema Registry 支持三种 Subject Name Strategy：TopicNameStrategy、 RecordNameStrategy 和 TopicRecordNameStrategy。目前TiCDC Avro 只支持TopicNameStrategy 一种，这意味着一个 kafka topic 只能接收一种数据格式的数据，所以 TiCDC Avro 禁止将多张表映射到同一个topic。 在创建changefeed 时，如果配置的分发规则中，topic 规则不包含 `{schema}` 和 `{table}` 占位符，将会报错。

兼容性说明

- 在升级 TiCDC 集群到 v7.0.0 时，如果使用 Avro 同步的表包含 FLOAT 类型数据，请在升级前手动调整 Confluent Schema Registry 的兼容性策略为 None，使 changefeed 能够成功更新 schema。否则，在升级之后 changefeed 将无法更新 schema 并进入错误状态。详情请参考GitHub issue #8490。

10.9.4.7.2 TiCDC Canal-JSON Protocol

Canal-JSON 是由 Alibaba Canal 定义的一种数据交换格式协议。通过本文，你可以了解TiCDC 对 Canal-JSON 数据格式的实现，包括TiDB 扩展字段、 Canal-JSON 数据格式定义，以及和官方实现进行对比等相关内容。

使用 Canal-JSON

当使用 MQ (Message Queue) 作为下游 Sink 时，你可以在 `sink-uri` 中指定使用 Canal-JSON，TiCDC 将以Event 为基本单位封装构造 Canal-JSON Message，向下游发送 TiDB 的数据变更事件。

Event 分为三类：

- DDL Event：代表 DDL 变更记录，在上游成功执行 DDL 语句后发出，DDL Event 会被发送到索引为 0 的 MQ Partition。
- DML Event：代表一行数据变更记录，在行变更发生时该类Event 被发出，包含变更后该行的相关信息。
- WATERMARK Event：代表一个特殊的时间点，表示在这个时间点前收到的Event 是完整的。仅适用于TiDB 扩展字段，当你在 `sink-uri` 中设置 `enable-tidb-extension=true` 时生效。

使用 `Canal-JSON` 时的配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-canal-json" --
    ↪ sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&protocol=canal-json"
```

TiDB 扩展字段

Canal-JSON 协议本是为 MySQL 设计的，其中并不包含 TiDB 专有的 CommitTS 事务唯一标识等重要字段。为了解决这个问题，TiCDC 在 Canal-JSON 协议格式中附加了TiDB 扩展字段。在 `sink-uri` 中设置 `enable-tidb-extension` 为 `true`（默认为 `false`）后，TiCDC 生成 Canal-JSON 消息时的行为如下：

- TiCDC 发送的 DML Event 和 DDL Event 类型消息中，将会含有一个名为 `_tidb` 的字段。
- TiCDC 将会发送WATERMARK Event 消息。·

配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-canal-json-enable
    ↪ -tidb-extension" --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&
    ↪ protocol=canal-json&enable-tidb-extension=true"
```

Message 格式定义

下面介绍 DDL Event、 DML Event 和 WATERMARK Event 的格式定义，以及消费端的数据解析。

DDL Event

TiCDC 会把一个 DDL Event 编码成如下 Canal-JSON 格式：

```
{
    "id": 0,
    "database": "test",
    "table": "",
    "pkNames": null,
    "isDdl": true,
    "type": "QUERY",
    "es": 1639633094670,
    "ts": 1639633095489,
    "sql": "drop database if exists test",
    "sqlType": null,
    "mysqlType": null,
    "data": null,
    "old": null,
    "_tidb": {     // TiDB 的扩展字段
        "commitTs": 163963309467037594
    }
}
```

以上JSON 数据的字段解释如下：

字段	类型	说明
id	Number	TiCDC 默认值为 0
database	String	Row 所在的 Database 的名字
table	String	Row 所在的 Table 的名字
pkNames	Array	组成 primary key 的所有列的名字
isDdl	Bool	该条消息是否为 DDL 事件
type	String	Canal-JSON 定义的事件类型
es	Number	产生该条消息的事件发生时的 13 位（毫秒级）时间戳
ts	Number	TiCDC 生成该条消息时的 13 位（毫秒级）时间戳
sql	String	当 isDdl 为 true 时，记录对应的 DDL 语句
sqlType	Object	当 isDdl 为 false 时，记录每一列数据类型在Java 中的类型表示
mysqlType	object	当 isDdl 为 false 时，记录每一列数据类型在 MySQL 中的类型表示
data	Object	当 isDdl 为 false 时，记录每一列的名字及其数据值

字段	类型	说明
old	Object	仅当该条消息由 Update 类型事件产生时，记录每一列的名字，和 Update 之前的数据值
_tidb	Object	TiDB 扩展字段，仅当 `enable-tidb-extension` 为true 时才会存在。其中的 `commitTs` 值为造成Row 变更的事务的TSO

DML Event

对于一行 DML 数据变更事件，TiCDC 会将其编码成如下形式:

```
{
    "id": 0,
    "database": "test",
    "table": "tp_int",
    "pkNames": [
        "id"
    ],
    "isDdl": false,
    "type": "INSERT",
    "es": 1639633141221,
    "ts": 1639633142960,
    "sql": "",
    "sqlType": {
        "c_bigint": -5,
        "c_int": 4,
        "c_mediumint": 4,
        "c_smallint": 5,
        "c_tinyint": -6,
        "id": 4
    },
    "mysqlType":
        { "c_bigint":
        "bigint", "c_int":
        "int",
        "c_mediumint": "mediumint",
        "c_smallint": "smallint",
        "c_tinyint": "tinyint",
        "id": "int"
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "2147483647",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "127",
```

```
            "id": "2"
        }
    ],
    "old": null,
    "_tidb": {      // TiDB 的扩展字段
        "commitTs": 163963314122145239
    }
}
```

WATERMARK Event

仅当 enable-tidb-extension 为 true 时，TiCDC 才会发送WATERMARK Event，其 type 字段值为 TIDB_WATERMARK。该类型事件具有 _tidb 字段，当前只含有 watermarkTs，其值为该 Event 发送时的TSO。

当你收到一个该类型的事件，所有 commitTs 小于 watermarkTs 的事件均已发送完毕。因为TiCDC 提供 At Least Once 语义，可能出现重复发送数据的情况。如果后续收到有 commitTs 小于 watermarkTs 的事件，可以忽略。

WATERMARK Event 的示例如下：

```
{
    "id": 0,
    "database": "",
    "table": "",
    "pkNames": null,
    "isDdl": false,
    "type": "TIDB_WATERMARK",
    "es": 1640007049196,
    "ts": 1640007050284,
    "sql": "",
    "sqlType": null,
    "mysqlType": null,
    "data": null,
    "old": null,
    "_tidb": {      // TiDB 的扩展字段
        "watermarkTs": 429918007904436226
    }
}
```

消费端数据解析

从上面的示例中可知，Canal-JSON 具有统一的数据格式，针对不同的事件类型，有不同的字段填充规则。消费者可以使用统一的方法对该JSON 格式的数据进行解析，然后通过判断字段值的方式，来确定具体事件类型：

- 当 isDdl 为 true 时，该消息含有一条 DDL Event。
- 当 isDdl 为 false 时，需要对 type 字段加以判断。如果 type 为 TIDB_WATERMARK，可得知其为 WATERMARK Event，否则就是 DML Event。

字段说明

Canal-JSON 格式会在 `mysqlType` 字段和 `sqlType` 字段中记录对应的数据类型。

MySQL Type 字段

Canal-JSON 格式会在 `mysqlType` 字段中记录每一列的 MySQL Type 的字符串表示。相关详情可以参考TiDB Data Types。

SQL Type 字段

Canal-JSON 格式会在 `sqlType` 字段中记录每一列的Java SQL Type，即每条数据在JDBC 中对应的数据类型，其值可以通过 MySQL Type 和具体数据值计算得到。具体对应关系如下:

MySQL Type	Java SQL Type Code
Boolean	-6
Float	7
Double	8
Decimal	3
Char	1
Varchar	12
Binary	2004
Varbinary	2004
Tinytext	2005
Text	2005
Mediumtext	2005
Longtext	2005
Tinyblob	2004
Blob	2004
Mediumblob	2004
Longblob	2004
Date	91
Datetime	93
Timestamp	93
Time	92
Year	12
Enum	4
Set	-7
Bit	-7
JSON	12

整数类型

你需要考虑整数类型是否有 `Unsigned` 约束，以及当前取值大小，分别对应不同的Java SQL Type Code。如下表所示。

MySQL Type String	Value Range	Java SQL Type Code
tinyint	[-128, 127]	-6
tinyint unsigned	[0, 127]	-6

MySQL Type String	Value Range	Java SQL Type Code
tinyint unsigned	[128, 255]	5
smallint	[-32768, 32767]	5
smallint unsigned	[0, 32767]	5
smallint unsigned	[32768, 65535]	4
mediumint	[-8388608, 8388607]	4
mediumint unsigned	[0, 8388607]	4
mediumint unsigned	[8388608, 16777215]	4
int	[-2147483648, 2147483647]	4
int unsigned	[0, 2147483647]	4
int unsigned	[2147483648, 4294967295]	-5
bigint	[-9223372036854775808, 9223372036854775807]	-5
bigint unsigned	[0, 9223372036854775807]	-5
bigint unsigned	[9223372036854775808, 18446744073709551615]	3

TiCDC 涉及的Java SQL Type 及其 Code 映射关系如下表所示。

Java SQL Type	Java SQL Type Code
CHAR	1
DECIMAL	3
INTEGER	4
SMALLINT	5
REAL	7
DOUBLE	8
VARCHAR	12
DATE	91
TIME	92
TIMESTAMP	93
BLOB	2004
CLOB	2005
BIGINT	-5
TINYINT	-6
Bit	-7

想要了解Java SQL Type 的更多信息，请参考Java SQL Class Types。

TiCDC Canal-JSON 和 Canal 官方实现对比

TiCDC 对 Canal-JSON 数据格式的实现，包括 `Update` 类型事件和 `mysqlType` 字段，和官方有些许不同。主要差异见下表。

差异点	TiCDC	Canal
`Update` 类型事件	`old` 字段包含所有列数据	`old` 字段仅包含被修改的列数据
`mysqlType` 字段	对于含有参数的类型，没有类型参数信息	对于含有参数的类型，会包含完整的参数信息

Update 类型事件

对于 Update 类型事件，Canal 官方实现中，old 字段仅包含被修改的列数据，而TiCDC 的实现则包含所有列数据。

假设在上游TiDB 按顺序执行如下 SQL 语句:

```
create table tp_int
(
    id          int auto_increment,
    c_tinyint   tinyint   null,
    c_smallint  smallint  null,
    c_mediumint mediumint null,
    c_int       int       null,
    c_bigint    bigint    null,
    constraint pk
        primary key (id)
);

insert into tp_int(c_tinyint, c_smallint, c_mediumint, c_int, c_bigint)
values (127, 32767, 8388607, 2147483647, 9223372036854775807);

update tp_int set c_int = 0, c_tinyint = 0 where c_smallint = 32767;
```

对于 update 语句，TiCDC 将会输出一条 type 为 UPDATE 的事件消息，如下所示。该 update 语句仅对 c_int 和 c_tinyint 两列进行了修改。输出事件消息的 old 字段，则包含所有列数据。

```
{
    "id": 0,
    ...
    "type": "UPDATE",
    ...
    "sqlType": {
        ...
    },
    "mysqlType": {
        ...
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ],
```

```
    "old": [                                    // TiCDC 输出事件消息的 `old` 字段，则包含所有列数据
        ↪ 。
        {
            "c_bigint": "9223372036854775807",
            "c_int": "2147483647",              // 修改的列
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "127",                 // 修改的列
            "id": "2"
        }
    ]
}
```

官方 Canal 输出事件消息的 old 字段仅包含被修改的列数据。示例如下。

```
{
    "id": 0,
    ...
    "type": "UPDATE",
    ...
    "sqlType": {
        ...
    },
    "mysqlType": {
        ...
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ],
    "old": [
        ↪ 仅包含被修改的列的数据。                  // Canal 输出事件消息的 `old` 字段，
        {
            "c_int": "2147483647",
            "c_tinyint": "127",
                                                // 修改的列
        }                                       // 修改的列
    ]


}
```

mysqlType 字段

对于 mysqlType 字段，Canal 官方实现中，对于含有参数的类型，会包含完整的参数信息，TiCDC 实现则没有类型参数信息。

在下面示例的表定义 SQL 语句中，如decimal / char / varchar / enum 等类型，都含有参数。对比TiCDC 和 Canal 官方实现分别生成的 Canal-JSON 格式数据可知，在 mysqlType 字段中的数据，TiCDC 实现只包含基本 MySQL Type。如果业务需要类型参数信息，需要你自行通过其他方式实现。

假设在上游数据库按顺序执行如下 SQL 语句:

```
create table t (
    id     int auto_increment,
    c_decimal     decimal(10, 4) null,
    c_char        char(16)       null,
    c_varchar     varchar(16)    null,
    c_binary      binary(16)     null,
    c_varbinary   varbinary(16) null,


    c_enum enum('a','b','c') null,
    c_set  set('a','b','c')  null,
    c_bit  bit(64)             null,
    constraint pk
        primary key (id)
);

insert into t (c_decimal, c_char, c_varchar, c_binary, c_varbinary, c_enum, c_set, c_bit)
values (123.456, "abc", "abc", "abc", "abc", 'a', 'a,b', b'1000001');
```

TiCDC 输出内容如下：

```
{
    "id": 0,
    ...
    "isDdl": false,
    "sqlType": {
        ...
    },
    "mysqlType":
        { "c_binary":
        "binary", "c_bit":
        "bit",
        "c_char": "char",
        "c_decimal": "decimal",
        "c_enum": "enum",
        "c_set": "set",
        "c_varbinary": "varbinary",
        "c_varchar": "varchar",
        "id": "int"
    },
    "data": [
```

```
        {
            ...
        }
    ],
    "old": null,
}
```

Canal 官方实现输出内容如下：

```
{
    "id": 0,
    ...
    "isDdl": false,
    "sqlType": {
        ...
    },
    "mysqlType": {
        "c_binary": "binary(16)",
        "c_bit": "bit(64)",
        "c_char": "char(16)",
        "c_decimal": "decimal(10, 4)",
        "c_enum": "enum('a','b','c')",
        "c_set": "set('a','b','c')",
        "c_varbinary": "varbinary(16)",
        "c_varchar": "varchar(16)",
        "id": "int"
    },
    "data": [
        {
            ...
        }
    ],
    "old": null,
}
```

TiCDC Canal-JSON 改动说明

Delete 类型事件中 Old 字段的变化说明

TiCDC 实现的 Canal-JSON 格式，v5.4.0 及以后版本的实现，和之前的有些许不同，具体如下：

- Delete 类型事件，Old 字段的内容发生了变化。

如下是一个 DELETE 事件的数据内容，在 v5.4.0 前的实现中，“old”的内容和“data”相同，在 v5.4.0 及之后的实现中，“old”将被设为null。你可以通过“data”字段获取到被删除的数据。

```
{
```

```
    "id": 0,
    "database": "test",
    ...
    "type": "DELETE",
    ...
    "sqlType": {
        ...
    },
    "mysqlType": {
        ...
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ],
    "old": null,

    // 以下示例是 v5.4.0 之前的实现，`old` 内容等同于 `data` 内容
    "old": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ]
}
```

10.9.4.7.3 TiCDC Open Protocol

概述

TiCDC Open Protocol 是一种行级别的数据变更通知协议，为监控、缓存、全文索引、分析引擎、异构数据库的主从复制等提供数据源。TiCDC 遵循TiCDC Open Protocol，向 MQ(Message Queue) 等第三方数据媒介复制TiDB 的数据变更。

TiCDC Open Protocol 以 Event 为基本单位向下游复制数据变更事件，Event 分为三类：

- Row Changed Event：代表一行的数据变化，在行发生变更时该Event 被发出，包含变更后该行的相关信息。
- DDL Event：代表 DDL 变更，在上游成功执行 DDL 后发出，DDL Event 会广播到每一个 MQ Partition 中。
- Resolved Event：代表一个特殊的时间点，表示在这个时间点前的收到的Event 是完整的。

协议约束

- 在绝大多数情况下，一个版本的Row Changed Event 只会发出一次，但是特殊情况（节点故障、网络分区等）下，同一版本的Row Changed Event 可能会多次发送。
- 同一张表中的每一个版本第一次发出的Row Changed Event 在 Event 流中一定是按TS (timestamp) 顺序递增的。
- Resolved Event 会被周期性的广播到各个 MQ Partition，Resolved Event 意味着任何TS 小于Resolved Event TS 的 Event 已经发送给下游。
- DDL Event 将被广播到各个 MQ Partition。
- 一行数据的多个Row Changed Event 一定会被发送到同一个 MQ Partition 中。

Message 格式定义

一个 Message 中包含一个或多个Event，按照以下格式排列：

Key:

Offset(Byte)	0~7	8~15	16~(15+ 长度 1)	…	…
参数	协议版本号	长度 1	Event Key1	长度 N	Event KeyN

Value:

Offset(Byte)	0~7	8~(7+ 长度 1)	…	…
参数	长度 1	Event Value1	长度 N	Event ValueN

- 长度N代表第N 个 Key/Value 的长度
- 长度及协议版本号均为大端序int64 类型
- 当前协议版本号为 1

Event 格式定义

本部分介绍Row Changed Event、 DDL Event 和 Resolved Event 的格式定义。

Row Changed Event

- Key:

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":1
}
```

参数	类型	说明
TS	Number	造成 Row 变更的事务的TS
Schema Name	String	Row 所在的 Schema 的名字
Table Name	String	Row 所在的 Table 的名字

- Value:

  `Insert` 事件，输出新增的行数据。

```
{
    "u":{
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>, "f":<Flag>,
            "v":<Column Value>
        },
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>, "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

  `Update` 事件，输出新增的行数据(“u”)以及修改前的行数据(“p”)，仅当 Old Value 特性开启时，才会输出修改前的行数据。

```
{
    "u":{
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    },
    "p":{
```

```
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

Delete 事件，输出被删除的行数据。当 Old Value 特性开启时，Delete 事件中包含被删除的行数据中的所有列；当 Old Value 特性关闭时，Delete 事件中仅包含HandleKey 列。

```
{
    "d":{
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>, "f":<Flag>,
            "v":<Column Value>
        },
        <Column
            Name>:{ "t":<Column
            Type>, "h":<Where
            Handle>, "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

参数	类型	说明
Column Name	String	列名
Column Type	Number	列类型，详见：Column 的类型码
Where Handle	Bool	表示该列是否可以作为 Where 筛选条件，当该列在表内具有唯一性时，Where Handle 为 true。
Flag	Number	列标志位，详见：列标志位
Column Value	Any	列值

DDL Event

- Key:

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":2
}
```

参数	类型	说明
TS	Number	进行 DDL 变更的事务的TS
Schema Name	String	DDL 变更的 Schema 的名字，可能为空字符串
Table Name	String	DDL 变更的 Table 的名字，可能为空字符串

- Value:

```
{
    "q":<DDL Query>,
    "t":<DDL Type>
}
```

参数	类型	说明
DDL Query	String	DDL Query SQL
DDL Type	String	DDL 类型，详见：DDL 的类型码

Resolved Event

- Key:

```
{
    "ts":<TS>,
    "t":3
}
```

参数	类型	说明
TS	Number	Resolved TS，任意小于该TS 的 Event 已经发送完毕

- Value: None

Event 流的输出示例

本部分展示并描述Event 流的输出日志。

假设在上游执行以下 SQL 语句，MQ Partition 数量为 2：

```
CREATE TABLE test.t1(id int primary key, val varchar(16));
```

如以下执行日志中的 Log 1、 Log 3 所示，DDL Event 将被广播到所有 MQ Partition，Resolved Event 会被周期性地广播到各个 MQ Partition：

```
 1. [partition=0] [key="{\"ts\":415508856908021766,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":2}"] [
    ↪ value="{\"q\":\"CREATE TABLE test.t1(id int primary key, val varchar(16))\",\"t\":3}"]
2. [partition=0] [key="{\"ts\":415508856908021766,\"t\":3}"] [value=]
3. [partition=1] [key="{\"ts\":415508856908021766,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":2}"] [
    ↪ value="{\"q\":\"CREATE TABLE test.t1(id int primary key, val varchar(16))\",\"t\":3}"]
4. [partition=1] [key="{\"ts\":415508856908021766,\"t\":3}"] [value=]
```

在上游执行以下 SQL 语句：

```
BEGIN;
INSERT INTO test.t1(id, val) VALUES (1, 'aa');
INSERT INTO test.t1(id, val) VALUES (2, 'aa');
UPDATE test.t1 SET val = 'bb' WHERE id = 2;
INSERT INTO test.t1(id, val) VALUES (3, 'cc');
COMMIT;
```

- 如以下执行日志中的 Log 5 和 Log 6 所示，同一张表内的Row Changed Event 可能会根据主键被分派到不同的 Partition，但同一行的变更一定会分派到同一个 Partition，方便下游并发处理。
- 如 Log 6 所示，在一个事务内对同一行进行多次修改，只会发出一个Row Changed Event。
- Log 8 是 Log 7 的重复 Event。 Row Changed Event 可能重复，但每个版本的Event 第一次发出的次序一定是有序的。

```
5. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":1},\"val\":{\"t\":15,\"v\":\"YWE=\"}}}"]
6. [partition=1] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":2},\"val\":{\"t\":15,\"v\":\"YmI=\"}}}"]
7. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"Y2M=\"}}}"]
8. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"Y2M=\"}}}"]
```

在上游执行以下 SQL 语句：

```
BEGIN;
DELETE FROM test.t1 WHERE id = 1;
UPDATE test.t1 SET val = 'dd' WHERE id = 3;
UPDATE test.t1 SET id = 4, val = 'ee' WHERE id = 2;
COMMIT;
```

- Log 9 是 Delete 类型的Row Changed Event，这种类型的Event 只包含主键列或唯一索引列。

- Log 13 和 Log 14 是 Resolved Event。 Resolved Event 意味着在这个 Partition 中，任意小于 Resolved TS 的 Event（包括 Row Changed Event 和 DDL Event）已经发送完毕。

```
9. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ′→ value="{\"d\":{\"id\":{\"t\":3,\"h\":true,\"v\":1}}}"]
10. [partition=1] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ′→ value="{\"d\":{\"id\":{\"t\":3,\"h\":true,\"v\":2}}}"]
 11. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
      ′→ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"ZGQ=\"}}}"]
 12. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
      ′→ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":4},\"val\":{\"t\":15,\"v\":\"ZWU=\"}}}"]
13. [partition=0] [key="{\"ts\":415508881038376963,\"t\":3}"] [value=]
14. [partition=1] [key="{\"ts\":415508881038376963,\"t\":3}"] [value=]
```

消费端协议解析

目前 TiCDC 没有提供 Open Protocol 协议解析的标准实现，但是提供了Golang 版本和Java 版本的解析例子。你可以参考本文档提供的数据格式和以下例子实现消费端协议解析。

- Golang 例子
- Java 例子

Column 的类型码

Column 的类型码用于标识Row Changed Event 中列的数据类型。

类型	Code	输出示例	说明
TINYINT/BOOL	1	{ “t” :1, “v” :1}	
SMALLINT	2	{ “t” :2, “v” :1}	
INT	3	{ “t” :3, “v” :123}	
FLOAT	4	{ “t” :4, “v” :153.123}	
DOUBLE	5	{ “t” :5, “v” :153.123}	
NULL	6	{ “t” :6, “v” :null}	
TIMESTAMP	7	{ “t” :7, “v” : “1973-12-30 15:30:00” }	
BIGINT	8	{ “t” :8, “v” :123}	
MEDIUMINT	9	{ “t” :9, “v” :123}	
DATE	10/14	{ “t” :10, “v” : “2000-01-01” }	
TIME	11	{ “t” :11, “v” : “23:59:59” }	
DATETIME	12	{ “t” :12, “v” : “2015-12-20 23:58:58” }	
YEAR	13	{ “t” :13, “v” :1970}	

类型	Code	输出示例	说明
VARCHAR/VARBINARY	15/253	{“t”:15,“v”:“ali测试”}/{“t”:15,“v”:“\\x89PNG\\r\\n\\x1a\\n”}	编码为 UTF-8；当上游类型为 VARBINARY 时，将对不可见的字符转义
BIT	16	{“t”:16,“v”:81}	
JSON	245	{“t”:245,“v”:“{\”key1\“:\”value1\“}”}	
DECIMAL	246	{“t”:246,“v”:“129012.1230000”}	
ENUM	247	{“t”:247,“v”:1}	
SET	248	{“t”:248,“v”:3}	
TINYTEXT/TINYBLOB	249	{“t”:249,“v”:“5rWL6K+VdGV4dA==”}	编码为 Base64
MEDIUMTEXT/MEDIUMBLOB	250	{“t”:250,“v”:“5rWL6K+VdGV4dA==”}	编码为 Base64

类型	Code	输出示例	说明
LONGTEXT/LONGBLOB	251	{“t”:251,“v”:“5rWL6K+VdGV4dA==”}	编码为 Base64
TEXT/BLOB	252	{“t”:252,“v”:“5rWL6K+VdGV4dA==”}	编码为 Base64
CHAR/BINARY	254	{“t”:254,“v”:“测试”}/{“t”:254,“v”:“\\x89PNG\\r\\n\\x1a\\n”}	编码为 UTF-8；当上游类型为 BINARY 时，将对不可见的字符转义
GEOMETRY	255		尚不支持

DDL 的类型码

DDL 的类型码用于标识 DDL Event 中的 DDL 语句的类型。

类型	Code
Create Schema	1
Drop Schema	2
Create Table	3
Drop Table	4
Add Column	5
Drop Column	6
Add Index	7
Drop Index	8
Add Foreign Key	9
Drop Foreign Key	10
Truncate Table	11
Modify Column	12
Rebase Auto ID	13
Rename Table	14
Set Default Value	15
Shard RowID	16
Modify Table Comment	17
Rename Index	18
Add Table Partition	19
Drop Table Partition	20
Create View	21
Modify Table Charset And Collate	22
Truncate Table Partition	23
Drop View	24
Recover Table	25
Modify Schema Charset And Collate	26
Lock Table	27
Unlock Table	28
Repair Table	29
Set TiFlash Replica	30
Update TiFlash Replica Status	31
Add Primary Key	32
Drop Primary Key	33
Create Sequence	34
Alter Sequence	35
Drop Sequence	36

列标志位

列标志位以 Bit flags 形式标记列的相关属性。

位移	值	名称	说明
1	0x01	BinaryFlag	该列是否为二进制编码列

位移	值	名称	说明
2	0x02	HandleKeyFlag	该列是否为 Handle 列
3	0x04	GeneratedColumnFlag	该列是否为生成列
4	0x08	PrimaryKeyFlag	该列是否为主键列
5	0x10	UniqueKeyFlag	该列是否为唯一索引列
6	0x20	MultipleKeyFlag	该列是否为组合索引列
7	0x40	NullableFlag	该列是否为可空列
8	0x80	UnsignedFlag	该列是否为无符号列

示例：

若某列 Flag 值为 85，则代表这一列为可空列、唯一索引列、生成列、二进制编码列。

```
85 == 0b_101_0101
   == NullableFlag | UniqueKeyFlag | GeneratedColumnFlag | BinaryFlag
```

若某列 Flag 值为 46，则代表这一列为组合索引列、主键列、生成列、Handle 列。

```
46 == 0b_010_1110
   == MultipleKeyFlag | PrimaryKeyFlag | GeneratedColumnFlag | HandleKeyFlag
```

注意：

- BinaryFlag 仅在列为 BLOB/TEXT（包括 TINYBLOB/TINYTEXT、BINARY/CHAR 等）类型时才有意义。当上游列为 BLOB 类型时，BinaryFlag 置 1；当上游列为TEXT 类型时，BinaryFlag 置 0。
- 若要同步上游的一张表，TiCDC 会选择一个有效索引作为 Handle Index。Handle Index 包含的列的 HandleKeyFlag 置 1。

10.9.4.7.4 TiCDC CSV Protocol

当使用云存储服务作为下游 sink 时，你可以使用CSV 格式将 DML 事件发送到下游云存储服务。

警告：

当开启Old Value 功能时 (`enable-old-value = true`)，CSV 协议数据格式无法输出更新事件的旧值。

具体原因请参考TiCDC 在开启 Old Value 功能后更新事件格式有何变化？

使用CSV

使用CSV 时的配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="csv-test" --sink-uri="
    ↪ s3://bucket/prefix" --config changefeed.toml
```

changefeed.toml 文件内容如下：

```
[sink]
protocol = "csv"
terminator = "\n"

[sink.csv]
delimiter = ','
quote = '"'
null = '\N'
include-commit-ts = true
```

数据保存的事务性约束

- 单个CSV 文件中后一行数据的commit-ts 大于等于前一行数据的commit-ts。
- 单表的同一事务不会存储在不同的CSV 文件中。
- 相同事务涉及的不同表会存储在不同的CSV 文件中。

数据存储路径结构

关于数据存储路径结构的更多信息，请参考同步数据到存储服务。

数据格式定义

CSV 文件中，单行的每一列定义如下：

- 第一列：DML 操作指示符，取值包括I、U 和D。I 表示 INSERT，U 表示 UPDATE，D 表示 DELETE。
- 第二列：表名。
- 第三列：库名。
- 第四列：commit ts，即原始事务的 commit ts。该列为可选配置。
- 第五列至最后一列：变更数据的列，可为一列或多列。

假设某张表 hr.employee 的定义如下：

```
CREATE TABLE `employee` (
  `Id` int NOT NULL,
  `LastName` varchar(20) DEFAULT NULL,
  `FirstName` varchar(30) DEFAULT NULL,
  `HireDate` date DEFAULT NULL,
  `OfficeLocation` varchar(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

该表上的 DML 事件以CSV 格式存储后如下所示：

```
"I","employee","hr",433305438660591626,101,"Smith","Bob","2014-06-04","New York"
"U","employee","hr",433305438660591627,101,"Smith","Bob","2015-10-08","Los Angeles"
"D","employee","hr",433305438660591629,101,"Smith","Bob","2017-03-13","Dallas"
"I","employee","hr",433305438660591630,102,"Alex","Alice","2017-03-14","Shanghai"
"U","employee","hr",433305438660591630,102,"Alex","Alice","2018-06-15","Beijing"
```

数据类型映射

MySQL 类型	CSV 类型	示例	描述
BOOLEAN/TINYINT/SMALLINT/INT/MEDIUMINT/BIGINT	Integer	123	-
FLOAT/DOUBLE	Float	153.123	-
NULL	Null	\N	-
TIMESTAMP/DATETIME	String	"1973-12-30 15:30:00.123456"	格式：yyyy-MM-dd HH:m
DATE	String	"2000-01-01"	格式：yyyy-MM-dd
TIME	String	"23:59:59"	格式：HH:mm:ss
YEAR	Integer	1970	-
VARCHAR/JSON/TINYTEXT/MEDIUMTEXT/LONGTEXT/TEXT/CHAR	String	"test"	以 UTF-8 编码输出
VARBINARY/TINYBLOB/MEDIUMBLOB/LONGBLOB/BLOB/BINARY	String	"6Zi/5pav"	以 Base64 编码输出
BIT	Integer	81	-
DECIMAL	String	"129012.1230000"	-
ENUM	String	"a"	-
SET	String	"a,b"	-

10.9.4.8 TiCDC OpenAPI v2

TiCDC 提供 OpenAPI 功能，你可以通过 OpenAPI v2 对 TiCDC 集群进行查询和运维操作。OpenAPI 的功能是cdc cli工具的一个子集。

> **注意：**
>
> TiCDC OpenAPI v1 将在未来版本中被删除。推荐使用TiCDC OpenAPI v2。

你可以通过 OpenAPI 完成TiCDC 集群的如下运维操作：

- 获取TiCDC 节点状态信息
- 检查TiCDC 集群的健康状态
- 创建同步任务
- 删除同步任务
- 更新同步任务配置
- 查询同步任务列表
- 查询特定同步任务
- 暂停同步任务

- 恢复同步任务
- 查询同步子任务列表
- 查询特定同步子任务
- 查询TiCDC 服务进程列表
- 驱逐 owner 节点
- 动态调整TiCDC Server 日志级别

所有API的请求体与返回值统一使用JSON 格式数据。请求如果成功，则统一返回 200 OK。本文档以下部分描述当前提供的API 的具体使用方法。

在下文的示例描述中，假设TiCDC server 的监听IP 地址为 127.0.0.1，端口为 8300。在启动 TiCDC server 时，可以通过 --addr=ip:port 指定TiCDC 绑定的IP 地址和端口。

10.9.4.8.1 API 统一错误格式

对API 发起请求后，如发生错误，返回错误信息的格式如下所示：

```
{
  "error_msg": "",
  "error_code": ""
}
```

如上所示，error_msg 描述错误信息，error_code 则是对应的错误码。

10.9.4.8.2 API List 接口统一返回格式

一个API 请求如果返回是一个资源列表（例如，返回所有的服务进程 Captures），TiCDC 统一的返回格式如下：

```
{
  "total": 2,
  "items": [
    {
      "id": "d2912e63-3349-447c-90ba-wwww",
      "is_owner": true,
      "address": "127.0.0.1:8300"
    },
    {
      "id": "d2912e63-3349-447c-90ba-xxxx",
      "is_owner": false,
      "address": "127.0.0.1:8302"
    }
  ]
}
```

如上所示:

- total: 表示有一共有多少个资源。
- items: 当次请求返回的资源在这个数组里，数组所有的元素都是同种资源。

10.9.4.8.3 获取TiCDC 节点状态信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v2/status
```

使用样例

以下请求会获取IP 地址为 127.0.0.1，端口号为 8300 的TiCDC 节点的状态信息。

```
curl -X GET http://127.0.0.1:8300/api/v2/status
```

```
{
  "version": "v7.0.0-master-dirty",
  "git_hash": "10413bded1bdb2850aa6d7b94eb375102e9c44dc",
  "id": "d2912e63-3349-447c-90ba-72a4e04b5e9e",
  "pid": 1447,
  "is_owner": true,
  "liveness": 0
}
```

以上返回信息的字段解释如下：

- version：当前 TiCDC 版本号。
- git_hash：Git 哈希值。
- id：该节点的 capture ID。
- pid：该节点 capture 进程的 PID。
- is_owner：表示该节点是否是 owner。
- liveness: 该节点是否在线。0 表示正常，1 表示处于 graceful shutdown 状态。

10.9.4.8.4 检查TiCDC 集群的健康状态

该接口是一个同步接口，在集群健康的时候会返回 200 OK。

请求 URI

```
GET /api/v2/health
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v2/health
```

如果集群健康，则返回 200 OK 和一个空的json {}：

```
{}
```

如果集群不健康，则返回错误信息。

10.9.4.8.5 创建同步任务

该接口用于向TiCDC 提交一个同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
POST /api/v2/changefeeds
```

参数说明

```
{
  "changefeed_id": "string",
  "replica_config": {
    "bdr_mode": true,
    "case_sensitive": true,
    "check_gc_safe_point": true,
    "consistent": {
      "flush_interval": 0,
      "level": "string",
      "max_log_size": 0,
      "storage": "string"
    },
    "enable_old_value": true,
    "enable_sync_point": true,
    "filter": {
      "do_dbs":
        [ "string"
      ],
      "do_tables": [
        {
          "database_name": "string",
          "table_name": "string"
        }
      ],
      "event_filters": [
        {
          "ignore_delete_value_expr": "string",
          "ignore_event": [
            "string"
          ],
          "ignore_insert_value_expr": "string",
          "ignore_sql": [
            "string"
          ],
          "ignore_update_new_value_expr": "string",
          "ignore_update_old_value_expr": "string",
          "matcher": [
```

```
          "string"
        ]
      }
    ],
    "ignore_dbs":
      [ "string"
    ],
    "ignore_tables": [
      {
        "database_name": "string",
        "table_name": "string"
      }
    ],
    "ignore_txn_start_ts":
      [ 0
    ],
    "rules":
      [ "strin
      g"
    ]
  },
  "force_replicate": true,
  "ignore_ineligible_table": true,
  "memory_quota": 0,
  "mounter":
    { "worker_num": 0
  },
  "sink":
    { "column_selectors":
    [
      {
        "columns":
          [ "string"
        ],
        "matcher":
          [ "string"
        ]
      }
    ],
    "csv": {
      "delimiter": "string",
      "include_commit_ts": true,
      "null": "string",
      "quote": "string"
    },
    "date_separator": "string",
    "dispatchers": [
```

```
        {
          "matcher":
            [ "string"
          ],
          "partition": "string",
          "topic": "string"
        }
      ],
      "enable_partition_separator": true,
      "encoder_concurrency": 0,
      "protocol": "string",
      "schema_registry": "string",
      "terminator": "string",
      "transaction_atomicity": "string"
    },
    "sync_point_interval": "string",
    "sync_point_retention": "string"
  },
  "sink_uri": "string",
  "start_ts": 0,
  "target_ts": 0
}
```

参数说明如下：

参数名	说明
changefeed_id	STRING 类型，同步任务的ID。（非必选）
replica_config	同步任务的配置参数。（非必选）
sink_uri	STRING 类型，同步任务下游的地址。（必选）
start_ts	UINT64 类型，指定changefeed 的开始TSO。TiCDC 集群将从这个TSO 开始拉取数据。默认为当前时间。（非必选）
target_ts	UINT64 类型，指定changefeed 的目标TSO。达到这个 TSO 后，TiCDC 集群将停止拉取数据。默认为空，即TiCDC 不会自动停止。（非必选）

changefeed_id、start_ts、target_ts、sink_uri 的含义和格式与使用cli 创建同步任务中所作的解释相同，具体解释请参见该文档。需要注意，当在 sink_uri 中指定证书的路径时，须确保已将对应证书上传到对应的 TiCDC server 上。

replica_config 参数说明如下：

参数名	说明
bdr_mode	BOOLEAN 类型，是否开启双向同步复制。默认值为 false。（非必选）
case_sensitive	BOOLEAN 类型，过滤表名时大小写是否敏感，默认值为 true。（非必选）
check_gc_safe_point	BOOLEAN 类型，是否检查同步任务的开始时间早于 GC 时间，默认值为 true。（非必选）

参数名	说明
consistent	Redo log 配置。（非必选）
enable_old_value	BOOLEAN 类型，是否输出 old value 值（即变更前的值），默认值为 true。（非必选）
enable_sync_point	BOOLEAN 类型，是否开启 sync point 功能。（非必选）
filter	filter 配置。（非必选）
force_replicate	BOOLEAN 类型，该值默认为 false，当指定为 true 时，同步任务会尝试强制同步没有唯一索引的表。（非必选）
ignore_ineligible_tabl	BOOLEAN 类型，该值默认为 false，当指定为 true 时，同步任务会忽略无法进行同步的表。（非必选）
memory_quota	UINT64 类型，同步任务的内存quota。（非必选）
mounter	同步任务 mounter 配置。（非必选）
sink	同步任务的sink配置。（非必选）
sync_point_interval	STRING 类型，注意返回值为 UINT64 类型的纳秒级时间，sync point 功能开启时，对齐上下游 snapshot 的时间间隔。默认值为 10m，最小值为 30s。（非必选）
sync_point_retention	STRING 类型，注意返回值为 UINT64 类型的纳秒级时间，sync point 功能开启时，在下游表中保存的数据的时长，超过这个时间的数据会被清理。默认值为 24h。（非必选）

consistent 参数说明如下：

参数名	说明
flush_interval	UINT64 类型，redo log 文件 flush 间隔。（非必选）
level	STRING 类型，同步数据的一致性级别。（非必选）
max_log_size	UINT64 类型，redo log 的最大值。（非必选）
storage	STRING 类型，存储的目标地址。（非必选）

filter 参数说明如下：

参数名	说明
do_dbs	STRING ARRAY 类型，需要同步的数据库。（非必选）
do_tables	需要同步的表。（非必选）
ignore_dbs	STRING ARRAY 类型，不同步的数据库。（非必选）
ignore_tables	不同步的表。（非必选）
event_filters	event 过滤配置。（非必选）
ignore_txn_start_ts	UINT64 ARRAY 类型，指定之后会忽略指定 start_ts 的事务，如 [1, 2]。（非必选）
rules	STRING ARRAY 类型，表库过滤的规则，如 ['foo*.*', 'bar*.*']。详情请参考表库过滤。（非必选）

filter.event_filters 参数说明如下，可参考日志过滤器。

参数名	说明
ignore_delete_value_expr ′→	STRING ARRAY 类型，如 "name = 'john'" 表示过滤掉包含 name = 'john' 条件的 DELETE DML。（非必选）
ignore_event	STRING ARRAY 类型，如 ["insert"] 表示过滤掉INSERT 事件。（非必选）
ignore_insert_value_expr ′→	STRING ARRAY 类型，如 "id >= 100" 表示过滤掉包含 id >= 100 条件的 INSERT DML。（非必选）
ignore_sql	STRING ARRAY 类型，如 ["^drop", "add column"] 表示过滤掉以 DROP 开头或者包含 ADD COLUMN 的 DDL。（非必选）
ignore_update_new_value_exSpTrRING ′→	ARRAY 类型，如 "gender = 'male'" 表示过滤掉新值 gender = 'male' 的 UPDATE DML。（非必选）
ignore_update_old_value_exSpTrRING ′→	ARRAY 类型，如 "age < 18" 表示过滤掉旧值 age < 18 的 UPDATE DML。（非必选）
matcher	STRING ARRAY 类型，是一个白名单，如 ["test.worker"]，表示该过滤规则只应用于 test 库中的 worker 表。（非必选）

mounter 参数说明如下：

参数名	说明
worker_num	INT 类型。Mounter 线程数，Mounter 用于解码TiKV 输出的数据，默认值为 16。（非必选）

sink 参数说明如下：

参数名	说明
column_selectors	column selector 配置。（非必选）
csv	CSV 配置。（非必选）
date_separator	STRING 类型，文件路径的日期分隔类型。可选类型有 none、year、month 和 day。默认值为 none，即不使用日期分隔。（非必选）
dispatchers	事件分发配置数组。（非必选）
encoder_concurrency ′→	INT 类型。MQ sink 中编码器的线程数。默认值为 16。（非必选）
protocol	STRING 类型，对于 MQ 类的 Sink，可以指定消息的协议格式。目前支持以下协议：canal-json、open-protocol、canal、avro 和 maxwell。
schema_registry	STRING 类型，schema registry 地址。（非必选）
terminator	STRING 类型，换行符，用来分隔两个数据变更事件。默认值为空，表示使用 "\r\n" 作为换行符。（非必选）
transaction_atomicitySTRING ′→	类型，事务一致性等级。（非必选）

sink.column_selectors 是一个数组，元素参数说明如下：

参数名	说明
columns	STRING ARRAY 类型，column 数组。

参数名	说明
matcher	STRING ARRAY 类型，matcher 配置，匹配语法和过滤器规则的语法相同。

sink.csv 参数说明如下：

参数名	说明
delimiter	STRING 类型，字段之间的分隔符。必须为ASCII 字符，默认值为 ,。
include_commit_ts	BOOLEAN 类型，是否在CSV 行中包含commit-ts。默认值为 false。
null	STRING 类型，如果这一列是null，那这一列该如何表示。默认是用 \N 来表示。
quote	STRING 类型，用于包裹字段的引号字符。空值代表不使用引号字符。默认值为"。

sink.dispatchers：对于 MQ 类的 Sink，可以通过该参数配置 event 分发器，支持以下分发器：default、ts、rowid、table 。分发规则如下：

- default：有多个唯一索引（包括主键）时按照 table 模式分发；只有一个唯一索引（或主键）时按照 rowid 模式分发；如果开启了 old value 特性，按照table 模式分发。
- ts：以行变更的 commitTs 做 Hash 计算并进行 event 分发。
- rowid：以所选的 HandleKey 列名和列值做 Hash 计算并进行 event 分发。
- table：以表的 schema 名和table 名做 Hash 计算并进行 event 分发。

sink.dispatchers 是一个数组，元素参数说明如下：

参数名	说明
matcher	STRING ARRAY 类型，匹配语法和过滤器规则的语法相同。
partition	STRING 类型，事件分发的目标partition。
topic	STRING 类型，事件分发的目标topic。

使用样例

以下请求会创建一个ID 为 test5，sink_uri 为 blackhole:// 的同步任务。

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v2/changefeeds -d
    '→ '{"changefeed_id":"test5","sink_uri":"blackhole://"}'
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

响应体格式

```
{
  "admin_job_type": 0,
  "checkpoint_time": "string",
  "checkpoint_ts": 0,
  "config": {
    "bdr_mode": true,
```

```
"case_sensitive": true,
"check_gc_safe_point": true,
"consistent": {
  "flush_interval": 0,
  "level": "string",
  "max_log_size": 0,
  "storage": "string"
},
"enable_old_value": true,
"enable_sync_point": true,
"filter": {
  "do_dbs":
    [ "string
    "
  ],
  "do_tables": [
    {
      "database_name": "string",
      "table_name": "string"
    }
  ],
  "event_filters": [
    {
      "ignore_delete_value_expr": "string",
      "ignore_event": [
        "string"
      ],
      "ignore_insert_value_expr": "string",
      "ignore_sql": [
        "string"
      ],
      "ignore_update_new_value_expr": "string",
      "ignore_update_old_value_expr": "string",
      "matcher": [
        "string"
      ]
    }
  ],
  "ignore_dbs":
    [ "string"
  ],
  "ignore_tables": [
    {
      "database_name": "string",
      "table_name": "string"
    }
```

```
  ],
  "ignore_txn_start_ts":
    [ 0
  ],
  "rules":
    [ "strin
    g"
  ]
},
"force_replicate": true,
"ignore_ineligible_table": true,
"memory_quota": 0,
"mounter":
  { "worker_num": 0
},
"sink":
  { "column_selectors":
  [
    {
      "columns":
        [ "string"
      ],
      "matcher":
        [ "string"
      ]
    }
  ],
  "csv": {
    "delimiter": "string",
    "include_commit_ts": true,
    "null": "string",
    "quote": "string"
  },
  "date_separator": "string",
  "dispatchers": [
    {
      "matcher":
        [ "string"
      ],
      "partition": "string",
      "topic": "string"
    }
  ],
  "enable_partition_separator": true,
  "encoder_concurrency": 0,
  "protocol": "string",
  "schema_registry": "string",
```

```
      "terminator": "string",
      "transaction_atomicity": "string"
    },
    "sync_point_interval": "string",
    "sync_point_retention": "string"
  },
  "create_time": "string",
  "creator_version": "string",
  "error": {
    "addr": "string",
    "code": "string",
    "message": "string"
  },
  "id": "string",
  "resolved_ts": 0,
  "sink_uri": "string",
  "start_ts": 0,
  "state": "string",
  "target_ts": 0,
  "task_status": [
    {
      "capture_id": "string",
      "table_ids": [
        0
      ]
    }
  ]
}
```

参数说明如下：

参数名	说明
admin_job_type	INTEGER 类型，admin 事件类型
checkpoint_time	STRING 类型，同步任务当前checkpoint 的格式化时间表示
checkpoint_ts	STRING 类型，同步任务当前checkpoint 的 TSO 表示
config	同步任务配置，结构和含义与创建同步任务中的 replica_config 配置项相同
create_time	STRING 类型，同步任务创建的时间
creator_version	STRING 类型，同步任务创建时 TiCDC 的版本
error	同步任务错误
id	STRING 类型，同步任务 ID
resolved_ts	UINT64 类型，同步任务 resolved ts
sink_uri	STRING 类型，同步任务的 sink uri
start_ts	UINT64 类型，同步任务 start ts
state	STRING 类型，同步任务状态，状态可分为 normal、stopped、error、failed、finished
target_ts	UINT64 类型，同步任务的 target ts

参数名	说明
task_status	同步任务分发的详细状态

task_status 参数说明如下：

参数名	说明
capture_id	STRING 类型，Capture ID
table_ids	UINT64 ARRAY 类型，该 Capture 上正在同步的table 的 ID

error 参数说明如下：

参数名	说明
addr	STRING 类型，Capture 地址
code	STRING 类型，错误码
message	STRING 类型，错误的详细信息

10.9.4.8.6 删除同步任务

该接口是幂等的（即其任意多次执行所产生的影响均与一次执行的影响相同），用于删除一个 changefeed 同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
DELETE /api/v2/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要删除的同步任务(changefeed) 的 ID

使用样例

以下请求会删除ID 为 test1 的同步任务。

```
curl -X DELETE http://127.0.0.1:8300/api/v2/changefeeds/test1
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

10.9.4.8.7 更新同步任务配置

该接口用于更新一个同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

修改changefeed 配置需要按照暂停任务 -> 修改配置 -> 恢复任务的流程。

请求 URI

```
PUT /api/v2/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要更新的同步任务(changefeed) 的 ID

请求体参数

```
{
  "replica_config":
    { "bdr_mode": true,
    "case_sensitive": true,
    "check_gc_safe_point": true,
    "consistent": {
      "flush_interval": 0,
      "level": "string",
      "max_log_size": 0,
      "storage": "string"
    },
    "enable_old_value": true,
    "enable_sync_point": true,
    "filter": {
      "do_dbs":
        [ "string"
      ],
      "do_tables": [
        {
          "database_name": "string",
          "table_name": "string"
        }
      ],
      "event_filters": [
        {
          "ignore_delete_value_expr": "string",
          "ignore_event": [
            "string"
          ],
          "ignore_insert_value_expr": "string",
          "ignore_sql": [
            "string"
          ],
          "ignore_update_new_value_expr": "string",
```

```
        "ignore_update_old_value_expr": "string",
        "matcher": [
          "string"
        ]
      }
    ],
    "ignore_dbs":
      [ "string"
    ],
    "ignore_tables": [
      {
        "database_name": "string",
        "table_name": "string"
      }
    ],
    "ignore_txn_start_ts":
      [ 0
    ],
    "rules":
      [ "strin
      g"
    ]
  },
  "force_replicate": true,
  "ignore_ineligible_table": true,
  "memory_quota": 0,
  "mounter":
    { "worker_num": 0
  },
  "sink":
    { "column_selectors":
    [
      {
        "columns":
          [ "string"
        ],
        "matcher":
          [ "string"
        ]
      }
    ],
    "csv": {
      "delimiter": "string",
      "include_commit_ts": true,
      "null": "string",
      "quote": "string"
    },
```

```
      "date_separator": "string",
      "dispatchers": [
        {
          "matcher":
            [ "string"
          ],
          "partition": "string",
          "topic": "string"
        }
      ],
      "enable_partition_separator": true,
      "encoder_concurrency": 0,
      "protocol": "string",
      "schema_registry": "string",
      "terminator": "string",
      "transaction_atomicity": "string"
    },
    "sync_point_interval": "string",
    "sync_point_retention": "string"
  },
  "sink_uri": "string",
  "target_ts": 0
}
```

目前仅支持通过API 修改同步任务的如下配置。

参数名	说明
target_ts	UINT64 类型，指定 changefeed 的目标 TSO。(非必选)
sink_uri	STRING 类型，同步任务下游的地址。(非必选)
replica_config	sink 的配置参数, 必须是完整的配置。(非必选)

以上参数含义与创建同步任务中的参数相同，此处不再赘述。

使用样例

以下请求会更新ID 为 test1 的同步任务的 target_ts 为 32。

```
curl -X PUT -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v2/changefeeds/
    ↪ test1 -d '{"target_ts":32}'
```

若是请求成功，则返回 200 OK，若请求失败，则返回错误信息和错误码。响应的 JSON 格式以及字段含义与创建同步任务中的响应参数相同，此处不再赘述。

10.9.4.8.8 查询同步任务列表

该接口是一个同步接口，请求成功会返回TiCDC 集群中所有同步任务(changefeed) 的基本信息。

请求 URI

`GET /api/v2/changefeeds`

参数说明

查询参数

参数名	说明
`state`	非必选，指定后将会只返回该状态的同步任务的信息

`state` 可选值为 `all`、`normal`、`stopped`、`error`、`failed`、`finished`。

若不指定该参数，则默认返回处于 `normal`、`stopped`、`failed` 状态的同步任务基本信息。

使用样例

以下请求查询所有状态(state) 为 `normal` 的同步任务的基本信息。

```
curl -X GET http://127.0.0.1:8300/api/v2/changefeeds?state=normal
```

```
{
  "total": 2,
  "items": [
    {
      "id": "test",
      "state": "normal",
      "checkpoint_tso": 439749918821711874,
      "checkpoint_time": "2023-02-27 23:46:52.888",
      "error": null
    },
    {
      "id": "test2",
      "state": "normal",
      "checkpoint_tso": 439749918821711874,
      "checkpoint_time": "2023-02-27 23:46:52.888",
      "error": null
    }
  ]
}
```

以上返回的信息的说明如下：

- `id`：同步任务的 ID。
- `state`：同步任务当前所处的状态。
- `checkpoint_tso`：同步任务当前 checkpoint 的 TSO 表示。
- `checkpoint_time`：同步任务当前 checkpoint 的格式化时间表示。
- `error`：同步任务的错误信息。

10.9.4.8.9 查询特定同步任务

该接口是一个同步接口，请求成功会返回指定同步任务(changefeed)的详细信息。

请求 URI

```
GET /api/v2/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要查询的同步任务(changefeed)的 ID

使用样例

以下请求会查询ID 为 test1 的同步任务的详细信息。

```
curl -X GET http://127.0.0.1:8300/api/v2/changefeeds/test1
```

响应的JSON 格式以及字段含义与创建同步任务中的响应参数相同，此处不再赘述。

10.9.4.8.10 暂停同步任务

该接口暂停一个同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
POST /api/v2/changefeeds/{changefeed_id}/pause
```

参数说明

路径参数

参数名	说明
changefeed_id	需要暂停的同步任务(changefeed)的 ID

使用样例

以下请求会暂停ID 为 test1 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v2/changefeeds/test1/pause
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

10.9.4.8.11 恢复同步任务

该接口恢复一个同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

POST /api/v2/changefeeds/{changefeed_id}/resume

参数说明

路径参数

参数名	说明
changefeed_id	需要恢复的同步任务(changefeed) 的 ID

请求体参数

```
{
  "overwrite_checkpoint_ts": 0
}
```

参数名	说明
overwrite_checkpoint_ts	UINT64 类型，恢复同步任务(changefeed) 时重新指定的checkpoint TSO

使用样例

以下请求会恢复ID 为 test1 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v2/changefeeds/test1/resume -d '{}'
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

10.9.4.8.12 查询同步子任务列表

该接口是一个同步接口，请求成功会返回当前TiCDC 集群中的所有同步子任务(processor) 的基本信息。

请求 URI

GET /api/v2/processors

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v2/processors
```

```
{
  "total": 3,
  "items": [
    {
      "changefeed_id": "test2",
      "capture_id": "d2912e63-3349-447c-90ba-72a4e04b5e9e"
    },
    {
```

```
      "changefeed_id": "test1",
      "capture_id":  "d2912e63-3349-447c-90ba-72a4e04b5e9e"
    },
    {
      "changefeed_id": "test",
      "capture_id":  "d2912e63-3349-447c-90ba-72a4e04b5e9e"
    }
  ]
}
```

以上返回的信息的说明如下：

- changefeed_id：同步任务的 ID。
- capture_id：Capture 的ID。

10.9.4.8.13 查询特定同步子任务

该接口是一个同步接口，请求成功会返回指定同步子任务(processor)的详细信息。

请求 URI

```
GET /api/v2/processors/{changefeed_id}/{capture_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要查询的子任务的 Changefeed ID
capture_id	需要查询的子任务的 Capture ID

使用样例

以下请求查询 changefeed_id 为 test、 capture_id 为 561c3784-77f0-4863-ad52-65a3436db6af 的同步子任务。一个同步子任务通过 changefeed_id 和 capture_id 来标识。

```
curl -X GET http://127.0.0.1:8300/api/v2/processors/test/561c3784-77f0-4863-ad52-65a3436db6af
```

```
{
  "table_ids":
    [ 80
  ]
}
```

以上返回的信息的说明如下：

- table_ids：在这个 capture 上同步的table 的 ID。

10.9.4.8.14 查询TiCDC服务进程列表

该接口是一个同步接口，请求成功会返回当前TiCDC集群中的所有服务进程(capture)的基本信息。

请求 URI

```
GET /api/v2/captures
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v2/captures
```

```
{
  "total": 1,
  "items": [
    {
      "id": "d2912e63-3349-447c-90ba-72a4e04b5e9e",
      "is_owner": true,
      "address": "127.0.0.1:8300"
    }
  ]
}
```

以上返回的信息的说明如下：

- id：capture 的ID。
- is_owner：该 capture 是否是 owner 。
- address：该 capture 的地址。

10.9.4.8.15 驱逐 owner 节点

该接口是一个异步的请求，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
POST /api/v2/owner/resign
```

使用样例

以下请求会驱逐TiCDC 当前的 owner 节点，并会触发新一轮的选举，产生新的 owner 节点。

```
curl -X POST http://127.0.0.1:8300/api/v2/owner/resign
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

10.9.4.8.16 动态调整TiCDC Server日志级别

该接口是一个同步接口，请求成功会返回 200 OK。

请求 URI

`POST /api/v2/log`

请求参数

请求体参数

参数名	说明
`log_level`	想要设置的日志等级

`log_level` 支持zap 提供的日志级别：“debug”、“info”、“warn”、“error”、“dpanic”、“panic”、“fatal”。

使用样例

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v2/log -d '{"
    ′→ log_level":"debug"}'
```

如果请求成功，则返回 `200 OK`。如果请求失败，则返回错误信息和错误码。

10.9.4.9 TiCDC OpenAPI v1

> **注意：**
>
> TiCDC OpenAPI v1 将在未来版本中被删除。推荐使用TiCDC OpenAPI v2。

TiCDC 提供OpenAPI 功能，你可以通过OpenAPI 对TiCDC 集群进行查询和运维操作。OpenAPI 的总体功能和`cdc cli` 工具类似。

你可以通过 OpenAPI 完成TiCDC 集群的如下运维操作：

- 获取TiCDC 节点状态信息
- 检查TiCDC 集群的健康状态
- 创建同步任务
- 删除同步任务
- 更新同步任务配置
- 查询同步任务列表
- 查询特定同步任务
- 暂停同步任务
- 恢复同步任务
- 查询同步子任务列表
- 查询特定同步子任务
- 查询TiCDC 服务进程列表
- 驱逐 owner 节点
- 手动触发表的负载均衡
- 手动调度表到其他节点
- 动态调整TiCDC Server 日志级别

所有API的请求体与返回值统一使用JSON格式数据。本文档以下部分描述当前提供的API的具体使用方法。

在下文的示例描述中，假设TiCDC server的监听IP地址为 `127.0.0.1`，端口为 `8300`（在启动 TiCDC server 时可以通过 `--addr=ip:port` 指定绑定的IP和端口）。

10.9.4.9.1 API 统一错误格式

对API 发起的请求后，如发生错误，返回错误信息的格式如下所示：

```
{
    "error_msg": "",
    "error_code": ""
}
```

如上所示，`error_msg` 描述错误信息，`error_code` 则是对应的错误码。

10.9.4.9.2 获取TiCDC 节点状态信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/status
```

使用样例

以下请求会获取IP地址为 `127.0.0.1`，端口号为 `8300` 的TiCDC 节点的状态信息。

```
curl -X GET http://127.0.0.1:8300/api/v1/status
```

```
{
    "version": "v5.2.0-master-dirty",
    "git_hash": "f191cd00c53fdf7a2b1c9308a355092f9bf8824e",
    "id": "c6a43c16-0717-45af-afd6-8b3e01e44f5d",
    "pid": 25432,
    "is_owner": true
}
```

以上返回信息的字段解释如下：

- `version`：当前 TiCDC 版本号。
- `git_hash`：Git 哈希值。
- `id`：该节点的 capture ID。
- `pid`：该节点 capture 进程的 PID。
- `is_owner`：表示该节点是否是 owner。

10.9.4.9.3 检查TiCDC集群的健康状态

该接口是一个同步接口，在集群健康的时候会返回 200 OK。

请求 URI

```
GET /api/v1/health
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v1/health
```

10.9.4.9.4 创建同步任务

该接口是一个异步接口，请求成功会返回 202 Accepted。该返回结果只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds
```

参数说明

使用API创建同步任务可选的参数不如使用 cli 命令创建同步任务的参数完备，以下是该API支持的参数。

请求体参数

参数名	说明
changefeed_id	STRING 类型，同步任务的ID。（非必选）
start_ts	UINT64 类型，指定changefeed 的开始TSO。（非必选）
target_ts	UINT64 类型，指定changefeed 的目标TSO。（非必选）
sink_uri	STRING 类型，同步任务下游的地址。（必选）
force_replicate	BOOLEAN 类型，是否强制同步没有唯一索引的表。（非必选）
ignore_ineligible_table	BOOLEAN 类型，是否忽略无法进行同步的表。（非必选）
filter_rules	STRING 类型数组，表库过滤的规则。（非必选）
ignore_txn_start_ts	UINT64 类型数组，忽略指定 start_ts 的事务。（非必选）
mounter_worker_num	INT 类型，Mounter 线程数。（非必选）
sink_config	sink 的配置参数。（非必选）

changefeed_id、start_ts、target_ts、sink_uri 的含义和格式与使用cli 创建同步任务中所作的解释相同，具体解释请参见该文档。需要注意，当在 sink_uri 中指定证书的路径时，须确保已将对应证书上传到对应的 TiCDC server 上。

下面会对一些需要补充说明的参数进行进一步阐述。

force_replicate：该值默认为 false，当指定为 true 时，同步任务会尝试强制同步没有唯一索引的表。

ignore_ineligible_table：该值默认为 false，当指定为 true 时，同步任务会忽略无法进行同步的表。

filter_rules：表库过滤的规则，如 filter_rules = ['foo*.*', 'bar*.*'] 详情参考表库过滤。

ignore_txn_start_ts：指定之后会忽略指定 start_ts 的事务，如 ignore-txn-start-ts = [1, 2]。

mounter_worker_num：Mounter 线程数，Mounter 用于解码TiKV 输出的数据，默认值为 16。

sink_config：sink 的配置参数，如下

```
{
  "dispatchers":[
    {"matcher":["test1.*", "test2.*"], "dispatcher":"ts"},
    {"matcher":["test3.*", "test4.*"], "dispatcher":"rowid"}
  ],
  "protocol":"canal-json"
}
```

dispatchers：对于 MQ 类的 Sink，可以通过 dispatchers 配置 event 分发器，支持default、 ts、 rowid、 table 四种分发器，分发规则如下：

- default：有多个唯一索引（包括主键）时按照table 模式分发；只有一个唯一索引（或主键）按照rowid 模式分发；如果开启了 old value 特性，按照table 分发。
- ts：以行变更的commitTs 做 Hash 计算并进行 event 分发。
- rowid：以所选的 HandleKey 列名和列值做 Hash 计算并进行 event 分发。
- table：以表的 schema 名和table 名做 Hash 计算并进行 event 分发。

matcher：匹配语法和过滤器规则语法相同。

protocol：对于 MQ 类的 Sink，可以指定消息的协议格式。目前支持 canal-json、 open-protocol、 canal、 avro 和 maxwell 五种协议。

使用样例

以下请求会创建一个ID 为 test5，sink_uri 为 blackhole:// 的同步任务。

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/changefeeds -d
    ↪ '{"changefeed_id":"test5","sink_uri":"blackhole://"}'
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.5 删除同步任务

该接口是一个异步接口，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
DELETE /api/v1/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要删除的同步任务(changefeed) 的 ID

使用样例

以下请求会删除ID 为 test1 的同步任务。

```
curl -X DELETE http://127.0.0.1:8300/api/v1/changefeeds/test1
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.6 更新同步任务配置

该接口是一个异步接口，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

修改changefeed 配置需要按照暂停任务 -> 修改配置 -> 恢复任务的流程。

请求 URI

```
PUT /api/v1/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要更新的同步任务(changefeed) 的 ID

请求体参数

目前仅支持通过API 修改同步任务的如下配置。

参数名	说明
target_ts	UINT64 类型，指定 changefeed 的目标 TSO。(非必选)
sink_uri	STRING 类型，同步任务下游的地址。(非必选)
filter_rules	STRING 类型数组，表库过滤的规则。(非必选)
ignore_txn_start_ts	UINT64 类型数组，忽略指定 start_ts 的事务。 (非必选)
mounter_worker_num	INT 类型，mounter 线程数。(非必选)
sink_config	sink 的配置参数。 (非必选)

以上参数含义与创建同步任务中的参数相同，此处不再赘述。

使用样例

以下请求会更新ID 为 test1 的同步任务的 mounter_worker_num 为 32。

```
curl -X PUT -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/changefeeds/
    ↪ test1 -d '{"mounter_worker_num":32}'
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.7 查询同步任务列表

该接口是一个同步接口，请求成功会返回TiCDC 集群中所有同步任务(changefeed) 的基本信息。

请求 URI

```
GET /api/v1/changefeeds
```

参数说明

查询参数

参数名	说明
state	非必选，指定后将会只返回该状态的同步任务的信息

state 可选值为all、 normal、 stopped、 error、 failed、 finished。

若不指定该参数，则默认返回处于normal、 stopped、 failed 状态的同步任务基本信息。

使用样例

以下请求查询所有状态(state) 为 normal 的同步任务的基本信息。

```
curl -X GET http://127.0.0.1:8300/api/v1/changefeeds?state=normal
```

```
[
    {
        "id": "test1",
        "state": "normal",
        "checkpoint_tso": 426921294362574849,
        "checkpoint_time": "2021-08-10 14:04:54.242",
        "error": null
    },
    {
        "id": "test2",
        "state": "normal",
        "checkpoint_tso": 426921294362574849,
        "checkpoint_time": "2021-08-10 14:04:54.242",
        "error": null
    }
]
```

此处对以上返回的信息做进一步阐述：

- id：同步任务的 ID
- state：同步任务当前所处的状态。
- checkpoint_tso：同步任务当前 checkpoint 的TSO 表示。
- checkpoint_time：同步任务当前 checkpoint 的格式化时间表示。
- error：同步任务的错误信息。

10.9.4.9.8 查询特定同步任务

该接口是一个同步接口，请求成功会返回指定同步任务(changefeed)的详细信息。

请求 URI

```
GET /api/v1/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要查询的同步任务(changefeed)的 ID

使用样例

以下请求会查询ID 为 test1 的同步任务的详细信息。

```
curl -X GET http://127.0.0.1:8300/api/v1/changefeeds/test1
```

```
{
    "id": "test1",
    "sink_uri": "blackhole://",
    "create_time": "2021-08-10 11:41:30.642",
    "start_ts": 426919038970232833,
    "target_ts": 0,
    "checkpoint_tso": 426921014615867393,
    "checkpoint_time": "2021-08-10 13:47:07.093",
    "sort_engine": "unified",
    "state": "normal",
    "error": null,
    "error_history": null,
    "creator_version": "",
    "task_status": [
        {
            "capture_id": "d8924259-f52f-4dfb-97a9-c48d26395945",
            "table_ids": [
                63,
                65
            ],
            "table_operations": {}
        }
    ]
}
```

10.9.4.9.9 暂停同步任务

该接口是一个异步接口，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/pause
```

参数说明

路径参数

参数名	说明
changefeed_id	需要暂停的同步任务(changefeed) 的 ID

使用样例

以下请求会暂停ID 为 test1 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v1/changefeeds/test1/pause
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.10 恢复同步任务

该接口是一个异步接口，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/resume
```

参数说明

路径参数

参数名	说明
changefeed_id	需要恢复的同步任务(changefeed) 的 ID

使用样例

以下请求会恢复ID 为 test1 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v1/changefeeds/test1/resume
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.11 查询同步子任务列表

该接口是一个同步接口，请求成功会返回当前TiCDC 集群中的所有同步子任务(processor) 的基本信息。

请求 URI

```
GET /api/v1/processors
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v1/processors
```

```
[
    {
        "changefeed_id": "test1",
        "capture_id":  "561c3784-77f0-4863-ad52-65a3436db6af"
    }
]
```

10.9.4.9.12 查询特定同步子任务

该接口是一个同步接口，请求成功会返回指定同步子任务(processor)的详细信息。

请求 URI

```
GET /api/v1/processors/{changefeed_id}/{capture_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要查询的子任务的 Changefeed ID
capture_id	需要查询的子任务的 Capture ID

使用样例

以下请求查询 changefeed_id 为 test、 capture_id 为 561c3784-77f0-4863-ad52-65a3436db6af 的同步子任务。一个同步子任务通过 changefeed_id 和 capture_id 来标识。

```
curl -X GET http://127.0.0.1:8300/api/v1/processors/test1/561c3784-77f0-4863-ad52-65a3436db6af
```

```
{
    "checkpoint_ts": 426919123303006208,
    "resolved_ts": 426919123369066496,
    "table_ids": [
        63,
        65
    ],
    "error": null
}
```

10.9.4.9.13 查询TiCDC 服务进程列表

该接口是一个同步接口，请求成功会返回当前TiCDC 集群中的所有服务进程(capture)的基本信息。

请求 URI

```
GET /api/v1/captures
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v1/captures
```

```
[
    {
        "id": "561c3784-77f0-4863-ad52-65a3436db6af",
        "is_owner": true,
        "address": "127.0.0.1:8300"
    }
]
```

10.9.4.9.14 驱逐 owner 节点

该接口是一个异步的请求，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/owner/resign
```

使用样例

以下请求会驱逐TiCDC 当前的 owner 节点，并会触发新一轮的选举，产生新的 owner 节点。

```
curl -X POST http://127.0.0.1:8300/api/v1/owner/resign
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.15 手动触发表的负载均衡

该接口是一个异步的请求，请求成功会返回 202 Accepted它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/tables/rebalance_table
```

参数说明

路径参数

参数名	说明
changefeed_id	进行调度的 Changefeed ID

使用样例

以下请求会触发ID 为 test1 的 changefeed 表的负载均衡。

```
curl -X POST http://127.0.0.1:8300/api/v1/changefeeds/test1/tables/rebalance_table
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.16 手动调度表到其他节点

该接口是一个异步的请求，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/tables/move_table
```

参数说明

路径参数

参数名	说明
changefeed_id	进行调度的 Changefeed ID

请求体参数

参数名	说明
target_capture_id	目标 Capture ID
table_id	需要调度的 Table ID

使用样例

以下请求会将ID 为 test1 的 changefeed 中 ID 为 49 的 table 调度到ID 为 6f19a6d9-0f8c-4dc9-b299-3ba7c0f216f5 的 capture 上去。

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/changefeeds/
    ′→ changefeed-test1/tables/move_table -d '{"capture_id":"6f19a6d9-0f8c-4dc9-b299-3
    ′→ ba7c0f216f5","table_id":49}'
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

10.9.4.9.17 动态调整TiCDC Server 日志级别

该接口是一个同步接口，请求成功会返回 200 OK。

请求 URI

```
POST /api/v1/log
```

请求参数

请求体参数

参数名	说明
log_level	想要设置的日志等级

log_level 支持zap 提供的日志级别：“debug”、“info”、“warn”、“error”、“dpanic”、“panic”、“fatal”。

使用样例

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/log -d '{"
    ↪ log_level":"debug"}'
```

若是请求成功，则返回 200 OK，若请求失败，则返回错误信息和错误码。

10.9.4.10 Storage sink 消费程序设计

本文介绍如何设计和实现一个TiDB 数据变更的消费程序。

> **注意：**
>
> 当前 Storage sink 无法处理 DROP DATABASE DDL，请你尽量避免执行该语句。如果需要执行 DROP DATABASE DDL，请在下游 MySQL 手动执行。

TiCDC 不提供消费存储服务的数据的标准实现。本文介绍一个基于Golang 的消费示例程序，该示例程序能够读取存储服务中的数据并写入到兼容 MySQL 的下游数据库。你可以参考本文提供的数据格式和以下示例代码实现消费端。

Golang 示例代码

10.9.4.10.1 Consumer 设计

下图是 Consumer 的整体消费流程：

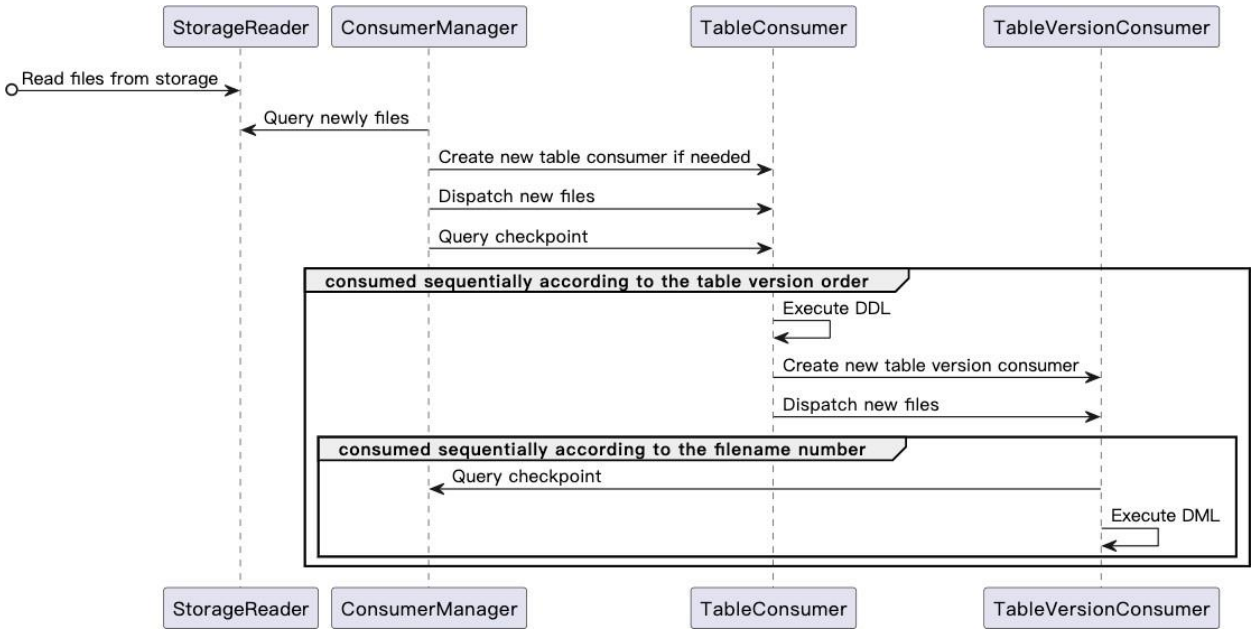


图 225: TiCDC storage consumer overview

以下是 Consumer 消费流程中的组件和功能定义，及其功能注释：

```
type StorageReader struct {
}

// Read the files from storage
// Add newly added files and delete files that not exist in storage
func (c *StorageReader) ReadFiles() {}

// Query newly added files and the latest checkpoint from storage, one file can only be returned
    ↪ once
func (c *StorageReader) ExposeNewFiles() (int64, []string) {}


// ConsumerManager is responsible for assigning tasks to TableConsumer.
// Different consumers can consume data concurrently,
// but data of one table must be processed by the same TableConsumer.
type ConsumerManager struct {
  // StorageCheckpoint is recorded in metadata file and it can be fetched by calling `
      ↪ StorageReader.ExposeNewFiles()`.
  // It indicates that the data whose transaction commit time is less than this checkpoint has
      ↪ been stored in storage
  StorageCheckpoint int64
  // it indicates where the consumer has consumed
  // ConsumerManager periodically collects TableConsumer.Checkpoint,
```

```
  // then Checkpoint is updated to the minimum value of all TableConsumer.Checkpoint
  Checkpoint int64

  tableFiles[schema][table]*TableConsumer
}

// Query newly files from StorageReader
// For new created table, create a TableConsumer for the new table
// If any, send new files to the corresponding TableConsumer
func (c *ConsumerManager) Dispatch() {}


type TableConsumer struct {
  // it indicates where this TableConsumer has consumed
  // Its initial value is ConsumerManager.Checkpoint
  // TableConsumer.Checkpoint is equal to TableVersionConsumer.Checkpoint
  Checkpoint int64

  schema,table string
  // Must be consumed sequentially according to the table version order
  verConsumers map[version int64]*TableVersionConsumer
  currentVer, previousVer int64
}

// Send newly files to the corresponding TableVersionConsumer
// For any DDL, assign a TableVersionConsumer for the new table version
func (tc *TableConsumer) Dispatch() {}

// If DDL query is empty or its tableVersion is less than TableConsumer.Checkpoint,
// - ignore this DDL, and consume the data under the table version
// Otherwise,
// - execute DDL first, and then consume the data under the table version
// - But for dropped table, self recycling after drop table DDL is executed
func (tc *TableConsumer) ExecuteDDL() {}


type TableVersionConsumer struct {
  // it indicates where the TableVersionConsumer has consumed
  // Its initial value is TableConsumer.Checkpoint
  Checkpoint int64

  schema,table,version string
  // For same table version, data in different partitions can be consumed concurrently
  # partitionNum int64
  // Must be consumed sequentially according to the data file number
```

```
  fileSet map[filename string]*TableVersionConsumer
  currentVersion
}

// If data commit ts is less than TableConsumer.Checkpoint
// or bigger than ConsumerManager.StorageCheckpoint,
// - ignore this data
// Otherwise,
// - process this data and write it to MySQL
func (tc *TableVersionConsumer) ExecuteDML() {}
```

10.9.4.10.2 DDL 事件的处理

举例来说，第一次遍历目录时目录内容如下：

```
├── metadata
└── test
    ├── tbl_1
    │   └── 437752935075545091
    │       ├── CDC000001.json
    │       └── schema.json
```

此时，首先需要解析 schema.json 文件中的表结构信息，从中获取 DDL Query 语句，并将其分为两种情况处理：

- 如果 DDL Query 为空或者 TableVersion 小于 consumer checkpoint 则跳过该语句。
- 否则，在下游 MySQL 数据库中执行获取到的 DDL 语句。

接着再开始同步数据文件 CDC000001.json。

例如，以下的 test/tbl_1/437752935075545091/schema.json 文件中 DDL Query 不为空：

```
{
    "Table":"test",
    "Schema":"tbl_1",
    "Version": 1,
    "TableVersion":437752935075545091,
    "Query": "create table tbl_1 (Id int primary key, LastName char(20), FirstName varchar(30),
        ↪ HireDate datetime, OfficeLocation Blob(20))",
    "TableColumns":[
        {
            "ColumnName":"Id",
            "ColumnType":"INT",
            "ColumnNullable":"false",
            "ColumnIsPk":"true"
        },
        {
```

```
            "ColumnName":"LastName",
            "ColumnType":"CHAR",
            "ColumnLength":"20"
        },
        {
            "ColumnName":"FirstName",
            "ColumnType":"VARCHAR",
            "ColumnLength":"30"
        },
        {
            "ColumnName":"HireDate",
            "ColumnType":"DATETIME"
        },
        {
            "ColumnName":"OfficeLocation",
            "ColumnType":"BLOB",
            "ColumnLength":"20"
        }
    ],
    "TableColumnsTotal":"5"
}
```

当程序再次遍历目录，发现该表新增了一个版本目录。程序先消费完 `test/tbl_1/437752935075545091` 目录下的所有文件，然后再消费新目录下的数据。

```
├── metadata
└── test
    ├── tbl_1
    │   ├── 437752935075545091
    │   │   ├── CDC000001.json
    │   │   └── schema.json
    │   └── 437752935075546092
    │   │   └── CDC000001.json
    │   │   └── schema.json
```

消费逻辑跟上述一致，先解析 `schema.json` 文件中的表结构信息，从中获取 DDL Query 语句并按不同情况处理，然后同步数据文件 `CDC000001.json`。

10.9.4.10.3 DML 事件的处理

在处理完 DDL 事件后，可以在 `{schema}/{table}/{table-version-separator}/` 目录下，根据具体的文件格式（CSV 或 Canal-JSON）并按照文件序号依次处理 DML 事件。

因为 TiCDC 提供 At Least Once 语义，可能出现重复发送数据的情况，所以需要在消费程序中比较数据事件的 commit ts 和 consumer checkpoint，并在 commit ts 小于 consumer checkpoint 的情况下进行去重处理。

10.9.4.11 TiCDC 兼容性

本文介绍了与TiCDC 有关的一系列兼容性问题及其处理方案。

10.9.4.11.1 命令行参数和配置文件兼容性

- TiCDC v4.0.0 中移除了 `ignore-txn-commit-ts`，添加了 `ignore-txn-start-ts`，使用 start_ts 过滤事务。
- TiCDC v4.0.2 中移除了 `db-dbs/db-tables/ignore-dbs/ignore-tables`，添加了 `rules`，使用新版的数据库和数据表过滤规则，详细语法参考表库过滤。
- 自 TiCDC v6.2.0 开始，`cdc cli` 将通过 TiCDC 的 Open API 直接与 TiCDC server 进行交互，而不再需要访问 PD。`cdc cli` 子命令中的 `--pd` 参数被废除，增加了 `--server` 参数，用于指定TiCDC Server 地址。请使用 `--server` 参数替代 `--pd` 参数。
- 从 v6.4.0 开始，TiCDC 使用 Syncpoint 功能需要同步任务拥有下游集群的 `SYSTEM_VARIABLES_ADMIN` 或者 SUPER 权限。

10.9.4.11.2 兼容性问题处理

本节介绍了兼容性相关的问题。

使用TiCDC v5.0.0-rc 版本的 `cdc cli` 工具操作 v4.0.x 集群导致不兼容问题

使用TiCDC v5.0.0-rc 版本的 `cdc cli` 工具操作 v4.0.x 版本的TiCDC 集群时，可能会遇到如下异常情况：

- 若 TiCDC 集群版本为 v4.0.8 或以下，使用 v5.0.0-rc 版本的 `cdc cli` 创建同步任务changefeed 时，可能导致 TiCDC 集群陷入异常状态，导致同步卡住。
- 若 TiCDC 集群版本为 v4.0.9 或以上，使用 v5.0.0-rc 版本的 `cdc cli` 创建同步任务 changefeed，会导致 Old Value 和 Unified Sorter 特性被非预期地默认开启。

处理方案：

使用和TiCDC 集群版本对应的 `cdc` 可执行文件进行如下操作：

1. 删除使用 v5.0.0-rc 版本创建的changefeed，例如：`tiup cdc:v4.0.9 cli changefeed remove -c xxxx --pd=xxxxx --force`。
2. 如果TiCDC 同步已经卡住，重启TiCDC 集群，例如：`tiup cluster restart <cluster_name> -R cdc`。
3. 重新创建changefeed，例如：`tiup cdc:v4.0.9 cli changefeed create --sink-uri=xxxx --pd=xxx`。

> 注意：
>
> 上述问题仅在 `cdc cli` 的版本是 v5.0.0-rc 时存在。未来其他 v5.0.x 版本的 `cdc cli` 可以兼容 v4.0.x 版本的集群。

`sort-dir` 及 `data-dir` 配置项的兼容性说明

`sort-dir` 配置项用于给TiCDC 内部的排序器指定临时文件目录，其作用在各版本有过如下兼容性更改：

版本	sort-engine 的使用	说明	使用建议

版本	sort-engine 的使用	说明	使用建议
v4.0.11 及之前的 v4.0 版本，v5.0.0-rc	作为 changefeed 配置项，给 file sorter 和 unified Sorter 指定临时文件目录	在这些版本中，file sorter 和 unified sorter 均不是正式功能 (GA)，不推荐在生产环境中使用。如果有多个 changefeed 被配	不推荐在生产环境中使用 Unified Sorter

版本	sort-engine 的使用	说明	使用建议
v4.0.12，v4.0.13，v5.0.0 及 v5.0.1	作为 change-feed 配置项或 cdc server 配置项	在默认情况下 change-feed 的 sort-dir 配置不会生效，而 cdc server 的 sort-dir 配置默认为 /tmp/cdc_sort。	需要通过 cdc server 命令行参数（或 TiUP）配置 sort-dir。

仅

版本	sort-engine 的使用	说明	使用建议
v4.0.14 及之后的 v4.0 版本，v5.0.3 及之后的 v5.0 版本，更新的版本	sort-dir 被弃用，建议配置 data-dir	data-dir 可以通过最新版本的 TiUP 进行配置。这些版本中 unified sorter 是默认开启的，升级时请确保 data-dir	需要通过 cdc server 命令行参数（或 TiUP）配置 data-dir

已经被

版本	sort-engine 的使用	说明	使用建议
		正	

版本	sort-engine 的使用	说明	使用建议
v6.0.0 及之后版本	data-dir 被用来存放 TiCDC 生成的临时文件	在该版本之后，TiCDC 默认采用 db sorter 作为内部的排序引擎，它使用 data-dir 作为磁盘目录。	需要通过 cdc server 命令行参数（或 TiUP 配置 data-dir

全局临时表兼容性说明

TiCDC 从 v5.3.0 开始支持全局临时表。

你需要使用TiCDC v5.3.0 及以上版本同步全局临时表到下游。低于该版本，会导致表定义错误。

如果TiCDC 的上游集群包含全局临时表，下游集群也必须是TiDB 5.3.0 及以上版本，否则同步报错。

10.9.5 TiCDC 故障处理

本文档总结了使用TiCDC 过程中常见的运行故障及解决方案。

> **注意：**
>
> 本文档 `cdc cli` 命令中指定 PD 地址为 `--server=http://127.0.0.1:8300`，在使用时需根据实际地址进行替换。

10.9.5.1 TiCDC 同步任务出现中断

10.9.5.1.1 如何判断TiCDC 同步任务出现中断？

- 通过Grafana 检查同步任务的 `changefeed checkpoint` 监控项。注意选择正确的 `changefeed id`。如果该值不发生变化或者查看 `checkpoint lag` 是否不断增大，可能同步任务出现中断。
- 通过Grafana 检查 `exit error count` 监控项，该监控项大于 0 代表同步任务出现错误。
- 通过 `cdc cli changefeed list` 和 `cdc cli changefeed query` 命令查看同步任务的状态信息。任务状态为 `stopped` 代表同步中断，`error` 项会包含具体的错误信息。任务出错后可以在TiCDC server 日志中搜索 `error on running processor` 查看错误堆栈，帮助进一步排查问题。
- 部分极端异常情况下TiCDC 出现服务重启，可以在TiCDC server 日志中搜索 `FATAL` 级别的日志排查问题。

10.9.5.1.2 如何查看TiCDC 同步任务是否被人为终止？

可以使用 `cdc cli` 查询同步任务是否被人为终止。例如：

```
cdc cli changefeed query --server=http://127.0.0.1:8300 --changefeed-id 28c43ffc-2316-4f4f-a70b-
    ↪ d1a7c59ba79f
```

上述命令的输出中 `admin-job-type` 标志这个同步的任务的状态：

- 0: 任务进行中，没有被人为停止。
- 1: 任务暂停，停止任务后所有同步 `processor` 会结束退出，同步任务的配置和同步状态都会保留，可以从 `checkpoint-ts` 恢复任务。
- 2: 任务恢复，同步任务从 `checkpoint-ts` 继续同步。
- 3: 任务已删除，接口请求后会结束所有同步 `processor`，并清理同步任务配置信息。同步状态保留，只提供查询，没有其他实际功能。

10.9.5.1.3 如何处理TiCDC 同步任务的中断？

目前已知可能发生的同步中断包括以下场景：

- 下游持续异常，TiCDC 多次重试后仍然失败。
  - 该场景下TiCDC 会保存任务信息，由于TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务checkpoint 之后的数据不会被TiKV GC 清理掉。
  - 处理方法：在下游恢复正常后，通过 HTTP 接口恢复同步任务。
- 因下游存在不兼容的 SQL 语句，导致同步不能继续。
  - 该场景下TiCDC 会保存任务信息，由于TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务checkpoint 之后的数据不会被TiKV GC 清理掉。
  - 处理方法：
    1. 先通过 `cdc cli changefeed query` 查询同步任务状态信息，记录 `checkpoint-ts` 值。
    2. 使用新的任务配置文件，增加`ignore-txn-start-ts` 参数跳过指定 `start-ts` 对应的事务。
    3. 通过 HTTP API 停止旧的同步任务，使用 `cdc cli changefeed create`，指定新的任务配置文件，指定 `start-ts` 为刚才记录的 `checkpoint-ts`，启动新的同步任务恢复同步。
- 在 v4.0.13 及之前的版本中TiCDC 同步分区表存在问题，导致同步停止。
  - 该场景下TiCDC 会保存任务信息，由于TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务checkpoint 之后的数据不会被TiKV GC 清理掉。
  - 处理方法：
    1. 通过 `cdc cli changefeed pause -c <changefeed-id>` 暂停同步。
    2. 等待约一分钟后，通过 `cdc cli changefeed resume -c <changefeed-id>` 恢复同步。

10.9.5.1.4 同步任务中断，尝试再次启动后TiCDC 发生 OOM，应该如何处理？

升级TiDB 集群和TiCDC 集群到最新版本。该 OOM 问题在 v4.0.14 及之后的 v4.0 版本，v5.0.2 及之后的 v5.0 版本，更新的版本上已得到缓解。

10.9.5.2 如何处理TiCDC 创建同步任务或同步到 MySQL 时遇到 `Error 1298: Unknown or incorrect time zone: 'UTC'` 错误？

这是因为下游 MySQL 没有加载时区，可以通过mysql_tzinfo_to_sql 命令加载时区，加载后就可以正常创建任务或同步任务。

```
mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root mysql -p
```

显示类似于下面的输出则表示导入已经成功：

```
Enter password:
Warning: Unable to load '/usr/share/zoneinfo/iso3166.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leap-seconds.list' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone1970.tab' as time zone. Skipping it.
```

如果下游是特殊的 MySQL 环境（某种公有云RDS 或某些 MySQL 衍生版本等），使用上述方式导入时区失败，就需要通过 sink-uri 中的 `time-zone` 参数指定下游的 MySQL 时区。可以首先在 MySQL 中查询其使用的时区：

```
show variables like '%time_zone%';
```

```
+------------------+---------------+
| Variable_name    | Value  |
+------------------+---------------+
| system_time_zone | CST    |
| time_zone        | SYSTEM |
+------------------+---------------+
```

然后在创建同步任务和创建TiCDC 服务时使用该时区：

```
cdc cli changefeed create --sink-uri="mysql://root@127.0.0.1:3306/?time-zone=CST" --server=http
    ↪ ://127.0.0.1:8300
```

> 注意：
>
> CST 可能是以下四个不同时区的缩写：
>
> - 美国中部时间：Central Standard Time (USA) UT-6:00
> - 澳大利亚中部时间：Central Standard Time (Australia) UT+9:30
> - 中国标准时间：China Standard Time UT+8:00
> - 古巴标准时间：Cuba Standard Time UT-4:00
>
> 在中国，CST 通常表示中国标准时间，使用时请注意甄别。

10.9.5.3 如何处理升级TiCDC 后配置文件不兼容的问题？

请参阅配置文件兼容注意事项。

10.9.5.4 TiCDC 启动任务的 start-ts 时间戳与当前时间差距较大， 任务执行过程中同步中断， 出现错误 `[CDC:ErrBufferReachLimit]`，怎么办？

自 v4.0.9 起可以尝试开启unified sorter 特性进行同步；或者使用 BR 工具进行一次增量备份和恢复，然后从新的时间点开启TiCDC 同步任务。TiCDC 将会在后续版本中对该问题进行优化。

10.9.5.5 当 changefeed 的下游为类 MySQL 数据库时，TiCDC 执行了一个耗时较长的 DDL 语句，阻塞了所有其他 changefeed，应该怎样处理？

1. 首先暂停执行耗时较长的 DDL 的changefeed。此时可以观察到，这个 changefeed 暂停后，其他的 changefeed 不再阻塞了。
2. 在 TiCDC log 中搜寻 `apply job` 字段，确认耗时较长的 DDL 的 `start-ts`。

3. 手动在下游执行该 DDL 语句，执行完毕后进行下面的操作。
4. 修改changefeed 配置，将上述 `start-ts` 添加到 `ignore-txn-start-ts` 配置项中。
5. 恢复被暂停的changefeed。

10.9.5.6 TiCDC 集群升级到v4.0.8 之后，changefeed 报错`[CDC:ErrKafkaInvalidConfig]Canal requires old value to be enabled`，为什么？

自 v4.0.8 起，如果changefeed 使用 `canal-json`、`canal` 或者 `maxwell` 协议输出，TiCDC 会自动开启 Old Value 功能。但是，当TiCDC 是从较旧版本升级到 v4.0.8 或以上版本时，在changefeed 使用 `canal-json`、`canal` 或 `maxwell` 协议的同时TiCDC 的 Old Value 功能会被禁用。此时，会出现该报错。可以按照以下步骤解决该报错：

1. 将changefeed 配置文件中 `enable-old-value` 的值设为 `true`。
2. 使用 `cdc cli changefeed pause` 暂停同步任务。

```
cdc cli changefeed pause -c test-cf --server=http://127.0.0.1:8300
```

3. 使用 `cdc cli changefeed update` 更新原有changefeed 的配置。

```
cdc cli changefeed update -c test-cf --server=http://127.0.0.1:8300 --sink-uri="mysql
    ↪ ://127.0.0.1:3306/?max-txn-row=20&worker-number=8" --config=changefeed.toml
```

4. 使用 `cdc cli changefeed resume` 恢复同步任务。

```
cdc cli changefeed resume -c test-cf --server=http://127.0.0.1:8300
```

10.9.5.7 使用 TiCDC 创建 changefeed 时报错 `[tikv:9006]GC life time is shorter than transaction duration, transaction starts at xx, GC safe point is yy`，该如何处理？

执行 `pd-ctl service-gc-safepoint --pd <pd-addrs>` 命令查询当前的GC safepoint 与 service GC safepoint。如果 GC safepoint 小于TiCDC changefeed 同步任务的开始时间戳 `start-ts`，可以直接在 `cdc cli create changefeed` 命令后加上 `--disable-gc-check` 参数创建changefeed。

如果 `pd-ctl service-gc-safepoint --pd <pd-addrs>` 的结果中没有 `gc_worker service_id`：

- 如果 PD 的版本<= v4.0.8，详见 PD issue #3128。
- 如果 PD 是由 v4.0.8 或更低版本滚动升级到新版，详见 PD issue #3366。
- 对于其他情况，请将上述命令执行结果反馈到AskTUG 论坛。

10.9.5.8 使用TiCDC 同步消息到 Kafka 时 Kafka 报错 `Message was too large`，该如何处理？

v4.0.8 或更低版本的TiCDC，仅在 Sink URI 中为 Kafka 配置 `max-message-bytes` 参数不能有效控制输出到 Kafka 的消息大小，需要在 Kafka server 配置中加入如下配置以增加 Kafka 接收消息的字节数限制。

```
### broker 能接收消息的最大字节数
message.max.bytes=2147483648
### broker 可复制的消息的最大字节数
replica.fetch.max.bytes=2147483648
```

```
### 消费者端的可读取的最大消息字节数
fetch.message.max.bytes=2147483648
```

10.9.5.9 TiCDC 同步时，在下游执行 DDL 语句失败会有什么表现，如何恢复？

如果某条DDL 语句执行失败，同步任务(changefeed) 会自动停止，checkpoint-ts 断点时间戳为该条出错DDL 语句的结束时间戳(finish-ts) 减去一。如果希望让TiCDC 在下游重试执行这条 DDL 语句，可以使用 `cdc cli changefeed resume` 恢复同步任务。例如：

```
cdc cli changefeed resume -c test-cf --server=http://127.0.0.1:8300
```

如果希望跳过这条出错的 DDL 语句，可以将changefeed 的 start-ts 设为报错时的checkpoint-ts 加上一，然后通过 `cdc cli changefeed create` 新建同步任务。假设报错时的checkpoint-ts 为 `415241823337054209`，可以进行如下操作来跳过该 DDL 语句：

```
cdc cli changefeed remove --server=http://127.0.0.1:8300 --changefeed-id simple-replication-task
cdc cli changefeed create --server=http://127.0.0.1:8300 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task" --sort-engine="unified" --start-
    ↪ ts 415241823337054210
```

10.9.6 TiCDC 常见问题解答

本文档总结了使用TiCDC 时经常遇到的问题。

> **注意：**
>
> 本文档 `cdc cli` 命令中指定 server 地址为 `--server=http://127.0.0.1:8300`，在使用时你需要根据实际地址进行替换。

10.9.6.1 TiCDC 创建任务时如何选择 start-ts？

首先需要理解同步任务的 `start-ts` 对应于上游TiDB 集群的一个TSO，同步任务会从这个 TSO 开始请求数据。所以同步任务的 `start-ts` 需要满足以下两个条件：

- `start-ts` 的值需要大于TiDB 集群当前的 `tikv_gc_safe_point`，否则创建任务时会报错。
- 启动任务时，需要保证下游已经具有 `start-ts` 之前的所有数据。对于同步到消息队列等场景，如果不需要保证上下游数据的一致，可根据业务场景放宽此要求。

如果不指定 `start-ts` 或者指定 `start-ts=0`，在启动任务的时候会去 PD 获取一个当前TSO，并从该 TSO 开始同步。

10.9.6.2 为什么TiCDC 创建任务时提示部分表不能同步？

在使用 `cdc cli changefeed create` 创建同步任务时会检查上游表是否符合同步要求。如果存在表不满足同步限制，会提示 `some tables are not eligible to replicate` 并列出这些不满足的表。如果选择 `Y` 或 `y` 则会继续创建同步任务，并且同步过程中自动忽略这些表的所有更新。如果选择其他输入，则不会创建同步任务。

10.9.6.3 如何查看TiCDC 同步任务的状态？

可以使用 `cdc cli` 查询同步任务的状态。例如：

```
cdc cli changefeed list --server=http://127.0.0.1:8300
```

上述命令输出如下：

```
[{
    "id": "4e24dde6-53c1-40b6-badf-63620e4940dc",
    "summary": {
      "state": "normal",
      "tso": 417886179132964865,
      "checkpoint": "2020-07-07 16:07:44.881",
      "error": null
    }
}]
```

- `checkpoint`：即为 TiCDC 已经将该时间点前的数据同步到了下游。
- `state` 为该同步任务的状态：
    - `normal`：正常同步。
    - `stopped`：停止同步（手动暂停或出错）。
    - `removed`：已删除任务。

> 注意：
>
> 该功能在TiCDC 4.0.3 版本引入。

10.9.6.4 TiCDC 的 `gc-ttl` 是什么？

从 TiDB v4.0.0-rc.1 版本起，PD 支持外部服务设置服务级别GC safepoint。任何一个服务可以注册更新自己服务的 GC safepoint。PD 会保证任何晚于该GC safepoint 的 KV 数据不会在TiKV 中被GC 清理掉。

在 TiCDC 中启用了这一功能，用来保证TiCDC 在不可用、或同步任务中断情况下，可以在TiKV 内保留TiCDC 需要消费的数据不被GC 清理掉。

启动TiCDC server 时可以通过 `gc-ttl` 指定GC safepoint 的 TTL，也可以通过 TiUP 修改TiCDC 的 `gc-ttl`，默认值为 24 小时。在TiCDC 中这个值有如下两重含义：

- 当 TiCDC 服务全部停止后，由TiCDC 在 PD 所设置的GC safepoint 保存的最长时间。

- TiCDC 中某个同步任务中断或者被手动停止时所能停滞的最长时间，若同步任务停滞时间超过 `gc-ttl` 所设置的值，那么该同步任务就会进入 `failed` 状态，无法被恢复，并且不会继续影响GC safepoint 的推进。

以上第二种行为是在TiCDC v4.0.13 版本及之后版本中新增的。目的是为了防止TiCDC 中某个同步任务停滞时间过长，导致上游TiKV 集群的GC safepoint 长时间不推进，保留的旧数据版本过多，进而影响上游集群性能。

> **注意：**
>
> 在某些应用场景中，比如使用Dumpling/BR 全量同步后使用TiCDC 接增量同步时，默认的gc-ttl 为 24 小时可能无法满足需求。此时应该根据实际情况，在启动TiCDC server 时指定 `gc-ttl` 的值。

10.9.6.5 TiCDC GC safepoint 的完整行为是什么？

TiCDC 服务启动后，如果有任务开始同步，TiCDC owner 会根据所有同步任务最小的checkpoint-ts 更新到 PD service GC safepoint，service GC safepoint 可以保证该时间点及之后的数据不被GC 清理掉。如果TiCDC 中某个同步任务中断、或者被用户主动停止，则该任务的checkpoint-ts 不会再改变，PD 对应的 service GC safepoint 最终会停滞在该任务的checkpoint-ts 处不再更新。

如果该同步任务停滞的时间超过了 `gc-ttl` 指定的时长，那么该同步任务就会进入 `failed` 状态，并且无法被恢复，PD 对应的 service GC safepoint 就会继续推进。

TiCDC 为 service GC safepoint 设置的存活有效期为 24 小时，即TiCDC 服务中断 24 小时内恢复能保证数据不因GC 而丢失。

10.9.6.6 如何理解TiCDC 时区和上下游数据库系统时区之间的关系？

	上游时区	TiCDC 时区	下游时区
配置方式	见时区支持	启动 ticdc server 时的 `--tz` 参数	sink-uri 中的 `time-zone` 参数

	上游时区	TiCDC 时区	下游时区
说明	上游 TiDB 的时区，影响 timestamp 类型的 DML 操作和与 timestamp 类型列相关的 DDL 操作。	TiCDC 会将假设上游 TiDB 的时区和 TiCDC 时区配置相同，对 timestamp 类型的列进行相关处理。	下游 MySQL 将按照下游的时区设置对 DML 和 DDL 操作中包含的 timestamp 进行处理。

注意：

请谨慎设置TiCDC server 的时区，因为该时区会用于时间类型的转换。上游时区、TiCDC 时区和下游时区应该保持一致。TiCDC server 时区使用的优先级如下：

- 最优先使用 `--tz` 传入的时区。

- 没有 `--tz` 参数，会尝试读取 TZ 环境变量设置的时区。
- 如果还没有 TZ 环境变量，会从TiCDC server 运行机器的默认时区。

10.9.6.7 创建同步任务时，如果不指定 `--config` 配置文件，TiCDC 的默认的行为是什么？

在使用 `cdc cli changefeed create` 命令时如果不指定 `--config` 参数，TiCDC 会按照以下默认行为创建同步任务：

- 同步所有的非系统表
- 开启 old value 功能
- 只同步包含有效索引的表

10.9.6.8 TiCDC 是否支持输出 Canal 格式的变更数据？

支持。要开启 Canal 格式输出，只需在 `--sink-uri` 中指定protocol 为 `canal` 即可，例如：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --sink-uri="kafka://127.0.0.1:9092/cdc-
    ↪ test?kafka-version=2.4.0&protocol=canal" --config changefeed.toml
```

注意：

- 该功能在TiCDC 4.0.2 版本引入。
- 目前TiCDC 仅支持将 Canal 格式的变更数据输出到 MQ 类的 Sink（例如 Kafka）。

更多信息请参考TiCDC Changefeed 配置参数。

10.9.6.9 为什么TiCDC 到 Kafka 的同步任务延时越来越大？

- 请参考如何查看TiCDC 同步任务的状态？检查下同步任务的状态是否正常。
- 请适当调整 Kafka 的以下参数：
  - `message.max.bytes`，将 Kafka 的 `server.properties` 中该参数调大到 1073741824 (1 GB)。
  - `replica.fetch.max.bytes`，将 Kafka 的 `server.properties` 中该参数调大到 1073741824 (1 GB)。
  - `fetch.message.max.bytes`，适当调大`consumer.properties` 中该参数，确保大于`message.max.bytes`。

10.9.6.10 TiCDC 把数据同步到 Kafka 时，能在TiDB 中控制单条消息大小的上限吗？

对于Avro 和 Canal-JSON 格式，消息是以行变更为单位发送的，一条 Kafka Message 仅包含一条行变更。一般情况下，消息的大小不会超过 Kafka 单条消息上限，因此，一般不需要限制单条消息大小。如果单条 Kafka 消息大小确实超过 Kakfa 上限，请参考为什么TiCDC 到 Kafka 的同步任务延时越来越大。

对于 Open Protocol 格式，一条 Kafka Message 可能包含多条行变更。因此，有可能存在某条 Kafka Message 消息过大。可以通过 `max-message-bytes` 控制每次向 Kafka broker 发送消息的最大数据量（可选，默认值 10 MB），通过 `max-batch-size` 参数指定每条 kafka 消息中变更记录的最大数量（可选，默认值 16）。

10.9.6.11 在一个事务中对一行进行多次修改，TiCDC 会输出多条行变更事件吗？

不会，在进行事务操作时，对于在一个事务内多次修改同一行的情况，TiDB 仅会将最新一次的修改结果发送给 TiKV。因此TiCDC 仅能获取到最新一次修改的结果。

10.9.6.12 TiCDC 把数据同步到 Kafka 时，一条消息中会不会包含多种数据变更？

会，一条消息中可能出现多个 `update` 或 `delete`，`update` 和 `delete` 也有可能同时存在。

10.9.6.13 TiCDC 把数据同步到 Kafka 时，如何查看TiCDC Open protocol 输出变更数据中的时间戳、表名和库名？

这些信息包含在 Kafka 消息的 Key 中，比如：

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":1
}
```

更多信息请参考Open protocol Event 格式定义

10.9.6.14 TiCDC 把数据同步到 Kafka 时，如何确定一条消息中包含的数据变更发生在哪个时间点？

把 Kafka 消息的 Key 中的 `ts` 右移 18 位即得unix timestamp。

10.9.6.15 TiCDC Open protocol 如何标示null 值？

Open protocol 的输出中type = 6 即为null，比如：

类型	Code	输出示例	说明
Null	6	`{"t":6,"v":null}`	

更多信息请参考Open protocol Event 格式定义。

10.9.6.16 如何区分TiCDC Open Protocol 中的Row Changed Event 是 INSERT 事件还是 UPDATE 事件？

如果没有开启 Old Value 功能，你无法区分TiCDC Open Protocol 中的Row Changed Event 是 INSERT 事件还是 UPDATE 事件。如果开启了 Old Value 功能，则可以通过事件中的字段判断事件类型：

- 如果同时存在 "p" 和 "u" 字段为 UPDATE 事件
- 如果只存在 "u" 字段则为 INSERT 事件
- 如果只存在 "d" 字段则为 DELETE 事件

更多信息请参考Open protocol Row Changed Event 格式定义。

10.9.6.17 TiCDC 占用多少 PD 的存储空间

TiCDC 使用 PD 内部的 etcd 来存储元数据并定期更新。因为 etcd 的多版本并发控制 (MVCC) 以及 PD 默认的 compaction 间隔是 1 小时，TiCDC 占用的 PD 存储空间与 1 小时内元数据的版本数量成正比。在 v4.0.5、v4.0.6、v4.0.7 三个版本中TiCDC 存在元数据写入频繁的问题，如果 1 小时内有 1000 张表创建或调度，就会用尽 etcd 的存储空间，出现 `etcdserver: mvcc: database space exceeded` 错误。出现这种错误后需要清理 etcd 存储空间，参考 etcd maintenance space-quota。如果你的 TiCDC 版本为 v4.0.5、v4.0.6 或 v4.0.7，建议升级到 v4.0.9 及以后版本。

10.9.6.18 TiCDC 支持同步大事务吗？有什么风险吗？

TiCDC 对大事务（大小超过 5 GB）提供部分支持，根据场景不同可能存在以下风险：

- 可能导致主从同步延迟大幅增高。
- 当TiCDC 内部处理能力不足时，可能出现同步任务报错 `ErrBufferReachLimit`。
- 当 TiCDC 内部处理能力不足或TiCDC 下游吞吐能力不足时，可能出现内存溢出(OOM)。

从 v6.2 版本开始，TiCDC 支持拆分单表事务功能，可大幅降低同步大事务的延时和内存消耗。因此，在业务对事务原子性要求不高的场景下，建议通过设置 sink uri 参数`transaction-atomicity` 打开拆分事务功能以解决可能出现的同步延迟和 OOM 问题。

如果实际同步过程中仍然遇到了上述错误，建议将包含大事务部分的增量数据通过 BR 进行增量恢复，具体操作如下：

1. 记录因为大事务而终止的changefeed 的 `checkpoint-ts`，将这个 TSO 作为 BR 增量备份的 `--lastbackupts`，并执行增量备份。
2. 增量备份结束后，可以在 BR 日志输出中找到类似 `["Full backup Failed summary : total backup ranges: 0, total success: 0, total failed: 0"] [BackupTS=421758868510212097]` 的日志，记录其中的 BackupTS。
3. 执行增量恢复。
4. 建立一个新的changefeed，从 BackupTS 开始同步任务。
5. 删除旧的changefeed。

10.9.6.19 TiCDC 是否会将有损 DDL 产生的数据变更同步到下游？

有损 DDL 是指在TiDB 中执行可能会导致数据改变的 DDL。一些常见的有损 DDL 操作包括：

- 修改列的类型，例如：INT -> VARCHAR
- 修改列的长度，例如：VARCHAR(20) -> VARCHAR(10)
- 修改列的精度，例如：DECIMAL(10, 3) -> DECIMAL(10, 2)
- 修改列的符号（有符号数/无符号数），例如：INT UNSIGNED -> INT SIGNED

在 TiDB v7.1.0 之前，TiCDC 会将一条新旧数据相同的 DML 事件同步到下游。当下游是 MySQL 时，这些 DML 事件不会产生任何数据变更，只有下游接收并执行该 DDL 语句后，数据才会发生变更。但是当下游是 Kafka 或者云存储时，TiCDC 会写入一条无用的数据到下游。

从TiDB v7.1.0 开始，TiCDC 会过滤掉这些无用的 DML 事件，不再将它们同步到下游。

10.9.6.20 同步 DDL 到下游 MySQL 5.7 时为什么时间类型字段默认值不一致？

比如上游 TiDB 的建表语句为 `create table test (id int primary key, ts timestamp)`，TiCDC 同步该语句到下游 MySQL 5.7，MySQL 使用默认配置， 同步得到的表结构如下所示，timestamp 字段默认值会变成 CURRENT_TIMESTAMP：

```
mysql root@127.0.0.1:test> show create table test;
+-------+-----------------------------------------------------------------------------------------+
| Table | Create Table                                                                            |
+-------+-----------------------------------------------------------------------------------------+
| test  | CREATE TABLE `test` (                                                                   |
|       |   `id` int(11) NOT NULL,                                                                |
|       |   `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,        |
|       |   PRIMARY KEY (`id`)                                                                    |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=latin1                                                  |
+-------+-----------------------------------------------------------------------------------------+
1 row in set
```

产生表结构不一致的原因是 `explicit_defaults_for_timestamp` 的默认值在 TiDB 和 MySQL 5.7 不同。从 TiCDC v5.0.1/v4.0.13 版本开始， 同步到 MySQL 会自动设置 session 变量 `explicit_defaults_for_timestamp = ON`，保证同步时间类型时上下游行为一致。对于 v5.0.1/v4.0.13 以前的版本，同步时间类型时需要注意 `explicit_defaults_for_timestamp` 默认值不同带来的兼容性问题。

10.9.6.21 使用 TiCDC 创建同步任务时将 `safe-mode` 设置为 `true` 后，为什么上游的 `INSERT/UPDATE` 语句经 TiCDC 同步到下游后变为了 `REPLACE INTO`？

TiCDC 提供至少一次的数据同步保证，当下游有重复数据时，会引起写冲突。为了避免该问题，TiCDC 会将 `INSERT` 和 `UPDATE` 语句转成 `REPLACE INTO` 语句。该行为由 `safe-mode` 参数来控制。

在 v6.1.3 版本之前，`safe-mode` 默认为 `true`，即所有的 `INSERT` 和 `UPDATE` 语句都转成 `REPLACE INTO` 语句。在 v6.1.3 及之后版本，系统能自动判断下游是否存在重复数据，`safe-mode` 默认更改为 `false`，当系统判断下游无重复数据时，会直接同步 `INSERT` 和 `UPDATE` 语句。

10.9.6.22 数据同步下游的 Sink 为 TiDB 或 MySQL 时，下游数据库的用户需要哪些权限？

Sink 为 TiDB 或 MySQL 时，下游数据库的用户需要以下权限：

- Select
- Index
- Insert
- Update
- Delete
- Create
- Drop
- Alter
- Create View

如果要同步 `recover table` 到下游 TiDB，需要有 Super 权限。

10.9.6.23 为什么TiCDC 需要使用磁盘，什么时候会写磁盘，TiCDC 能否利用内存缓存提升同步性能？

TiCDC 需要磁盘是为了缓冲上游写入高峰时下游消费不及时堆积的数据。TiCDC 正常运行期间都需要写入磁盘，但这通常不是同步吞吐和同步延时的瓶颈，写磁盘对延时影响在百毫秒内。TiCDC 也利用了内存来提升加速读取磁盘中的数据，以提升同步性能。

10.9.6.24 为什么在上游使用了TiDB Lightning 和 BR 恢复了数据之后，TiCDC 同步会出现卡顿甚至卡住？

目前TiCDC 还没完全适配TiDB Lightning 和 BR，请避免在使用 TiCDC 同步的表上使用TiDB Lightning 和 BR。

10.9.6.25 为什么恢复暂停的changefeed 后，changefeed 同步延迟越来越高，数分钟后才恢复正常？

当 changefeed 启动时，为了补齐changefeed 暂停期间产生的增量数据日志，TiCDC 需要扫描TiKV 中数据的历史版本，待扫描完毕后，才能够继续推进复制过程，扫描过程可能长达数分钟到数十分钟。

10.9.6.26 在两个异地TiDB 集群之间同步数据，如何部署TiCDC？

建议将TiCDC 部署在下游TiDB 集群。这是因为，如果上下游网络延迟较大，例如超过 100 ms 时，由于 MySQL 传输协议的原因，TiCDC 向下游执行 SQL 的延迟会急剧增加，导致系统的吞吐下降。部署在下游能够极大缓解该问题。

10.9.6.27 如何理解 DML 和 DDL 语句之间的执行顺序？

按照 DML -> DDL -> DML 的顺序执行。在数据同步过程中，为了确保 DML 事件在下游执行时有对应正确的表结构，需要协调 DDL 和 DML 的执行顺序。目前TiCDC 采用了简洁的方式处理该问题，会将 DDL ts 之前的 DML 都同步到下游之后，再同步 DDL。

10.9.6.28 如何对比上下游数据的一致性？

如果下游是TiDB 集群或者 MySQL，我们推荐使用sync diff inspector 工具进行数据对比。

10.9.6.29 单表数据同步只能在一个TiCDC 节点上运行，TiCDC 是否考虑使用多个节点同步多表数据？

目前正在开发中，未来TiCDC 会支持按照TiKV Region 粒度来同步数据变更日志，实现处理能力上的可扩展性。

10.9.6.30 上游有运行时间比较长的未提交事务，TiCDC 同步是否会被卡住？

TiDB 有事务超时的机制，当事务运行超过`max-txn-ttl` 后，会被TiDB 强制回滚。TiCDC 遇到未提交的事务，会等待其提交后再继续同步其数据，因此会出现同步延迟。

10.9.6.31 TiCDC 在开启 Old Value 功能后更新事件格式有何变化？

在下面的说明中，有效索引的定义如下：

- 主键(`PRIMARY KEY`) 为有效索引。
- 唯一索引(`UNIQUE INDEX`) 中每一列在表结构中明确定义非空(`NOT NULL`) 且不存在虚拟生成列(`VIRTUAL GENERATED COLUMNS`)。

聚簇索引指的是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。详见聚簇索引。

在开启Old Value 功能后，TiCDC 的表现如下：

- 对于非有效索引列的更新事件，输出的数据中会同时包含新值和旧值。
- 对于有效索引列的更新事件，输出的数据视不同情况而定：
  - 更新唯一索引列(UNIQUE INDEX)，并且该表不存在主键时，输出的数据中会同时包含新值和旧值。
  - 当上游TiDB 集群未开启聚簇索引，更新非INT 类型的主键列时，输出的数据中会同时包含新值和旧值。
  - 在其他情况下，更新事件会被拆分为对旧数据的删除事件和对新数据的插入事件。

以上行为的变化可能导致以下问题：

10.9.6.31.1 当更新有效索引列的事件同时包含新值和旧值时，Kafka Sink 的分发行为可能无法保证将具有相同索引列的更新事件分发到同一分区

Kafka Sink 的 index-value 分发行为是根据索引列的值来分发的。当更新事件同时包含新值和旧值时，索引列的值会发生变化，从而导致相同索引列的更新事件被分发到不同的分区。下面是一个示例：

在关闭TiDB 聚簇索引功能时，创建表t：

```
CREATE TABLE t (a VARCHAR(255) PRIMARY KEY NONCLUSTERED);
```

执行如下 DML 语句：

```
INSERT INTO t VALUES ("2");
UPDATE t SET a="1" WHERE a="2";
INSERT INTO t VALUES ("2");
UPDATE t SET a="3" WHERE a="2";
```

- 在未开启 Old Value 功能时，更新事件被拆分为删除旧值事件和插入新值事件。Kafka Sink 的 index-value 分发模式针对每个事件计算相应的分区。上述 DML 产生的事件会被发送到如下分区：

partition-1	partition-2	partition-3
INSERT a = 2	INSERT a = 1	INSERT a = 3
DELETE a = 2		
INSERT a = 2		
DELETE a = 2		

  因为 Kafka 的每个分区内可以保证消息的顺序，Kafka 消费者可以独立地消费每个分区中的数据，最终结果和 DML 执行顺序相同。

- 在开启 Old Value 功能时，Kafka Sink 的 index-value 分发模式会将相同索引列的更新事件分发到不同的分区。因此上述 DML 会被分发到如下分区（更新事件同时包含新值和旧值）：

partition-1	partition-2	partition-3
INSERT a = 2 INSERT a = 2	UPDATE a = 1 WHERE a = 2	UPDATE a = 3 WHERE a = 2

因为 Kafka 的各个分区之间不能保证消息的顺序，因此上述 DML 在消费过程中可能无法保证索引列的更新顺序。如果需要在输出的数据中会同时包含新值和旧值时保证索引列的更新顺序，建议在开启 Old Value 功能时，使用default 分发模式。

10.9.6.31.2 对于非有效索引列的更新事件和有效索引列的更新事件同时包含新值和旧值时，Kafka Sink 的 Avro 格式无法正确输出旧值

在 Avro 实现中，Kafka 消息的Value 格式只包含当前列值，因此当一个事件既有新值也有旧值时，无法正确输出旧值。如果需要输出旧值，建议关闭 Old Value 功能以获取拆分后的删除和插入事件。

10.9.6.31.3 对于非有效索引列的更新事件和有效索引列的更新事件同时包含新值和旧值时，Cloud Storage Sink 的 CSV 格式无法正确输出旧值

因为 CSV 文件的列数是固定的，当一个事件既有新值也有旧值时，无法正确输出旧值。如果需要输出旧值，建议使用 Canal-JSON 格式。

10.9.7 TiCDC 术语表

本术语表提供TiCDC 相关的术语和定义，这些术语会出现在TiCDC 的日志、监控指标、配置和文档中。

10.9.7.1 B

10.9.7.1.1 变更数据

从上游TiDB 集群写入TiCDC 的数据，包括 DML 操作引发的数据变更和 DDL 操作引发的表结构变更。

10.9.7.2 C

10.9.7.2.1 Capture

单个TiCDC 实例。多个 Capture 组成一个TiCDC 集群，Capture 上运行集群中的同步任务。

10.9.7.2.2 Changefeed

TiCDC 中的单个同步任务。同步任务将一个TiDB 集群中数张表的变更数据输出到一个指定的下游中。

10.9.7.3 O

10.9.7.3.1 Owner

一个特殊角色的Capture，负责管理 TiCDC 集群和调度TiCDC 集群中的同步任务。该角色由 Capture 选举产生，在任意时刻最多只存在一个。

10.9.7.4 P

10.9.7.4.1 Processor

TiCDC 同步任务会在TiCDC 实例上分配数据表，Processor 指这些数据表的同步处理单元。处理任务包括变更数据的拉取、排序、还原和分发。

10.10 TiDB Binlog

10.10.1 TiDB Binlog 简介

TiDB Binlog 是一个用于收集TiDB 的 binlog，并提供准实时备份和同步功能的商业工具。

TiDB Binlog 支持以下功能场景：

- 数据同步：同步TiDB 集群数据到其他数据库
- 实时备份和恢复：备份TiDB 集群数据，同时可以用于TiDB 集群故障时恢复

> 注意：
>
> TiDB Binlog 与 TiDB v5.0 版本开始引入的一些特性不兼容，无法一起使用，详情参照注意事项。
> 建议使用TiCDC 替代TiDB Binlog。

要快速了解 Binlog 的基本原理和使用方法，建议先观看下面的培训视频（时长 32 分钟）。注意本视频只为学习参考，具体操作步骤和最新功能，请以文档内容为准。

10.10.1.1 TiDB Binlog 整体架构

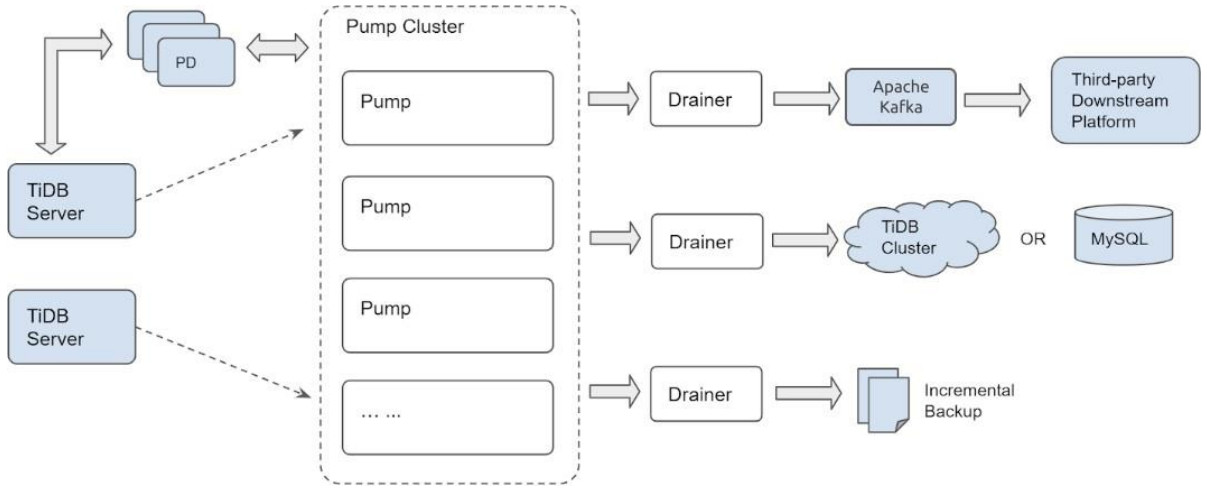


图 226: TiDB Binlog 架构

TiDB Binlog 集群主要分为 Pump 和 Drainer 两个组件，以及binlogctl 工具：

10.10.1.1.1 Pump

Pump 用于实时记录TiDB 产生的 Binlog，并将 Binlog 按照事务的提交时间进行排序，再提供给 Drainer 进行消费。

10.10.1.1.2 Drainer

Drainer 从各个 Pump 中收集 Binlog 进行归并，再将 Binlog 转化成 SQL 或者指定格式的数据，最终同步到下游。

10.10.1.1.3 binlogctl 工具

`binlogctl` 是一个TiDB Binlog 配套的运维工具，具有如下功能：

- 获取TiDB 集群当前的TSO
- 查看 Pump/Drainer 状态
- 修改 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer

10.10.1.2 主要特性

- 多个 Pump 形成一个集群，可以水平扩容。
- TiDB 通过内置的 Pump Client 将 Binlog 分发到各个 Pump。
- Pump 负责存储 Binlog，并将 Binlog 按顺序提供给 Drainer。
- Drainer 负责读取各个 Pump 的 Binlog，归并排序后发送到下游。
- Drainer 支持relay log 功能，通过relay log 保证下游集群的一致性状态。

10.10.1.3 注意事项

- TiDB Binlog 和 TiDB 在 v5.1 版本中解决了 v5.0 版本中引入的聚簇索引与TiDB Binlog 不兼容问题。在升级TiDB Binlog 和 TiDB Server 到 v5.1 版本后：开启TiDB Binlog 后，TiDB 支持创建聚簇索引表；聚簇索引表的数据插入、删除和更新操作支持通过TiDB Binlog 同步到下游。对于同步聚簇索引表时需注意：
  - 如果从 v5.0 版本手动控制组件升级顺序进行升级，请确保先将TiDB Binlog 升级至 v5.1 版本后再将 TiDB Server 升级至 v5.1 版本。
  - 推荐将上下游的TiDB 系统变量`tidb_enable_clustered_index` 配置为一致的值来保证上下游TiDB 聚簇索引表结构一致。
- TiDB Binlog 与TiDB v5.0 版本开始引入的以下特性不兼容，无法一起使用：
  - TiDB 聚簇索引特性：开启TiDB Binlog 后TiDB 不允许创建非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；
  - TiDB 系统变量tidb_enable_async_commit：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。
  - TiDB 系统变量tidb_enable_1pc：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。
- Drainer 支持将 Binlog 同步到 MySQL、TiDB、Kafka 或者本地文件。如果需要将 Binlog 同步到其他 Drainer 不支持的类型的系统中，可以设置 Drainer 将 Binlog 同步到 Kafka，然后根据 binlog consumer protocol 进行定制处理，参考Binlog Consumer Client 用户文档。
- 如果TiDB Binlog 用于增量恢复，可以设置配置项 `db-type="file"`，Drainer 会将binlog 转化为指定的proto buffer 格式的数据，再写入到本地文件中。这样就可以使用Reparo 恢复增量数据。

  关于 `db-type` 的取值，应注意：
  - 如果TiDB 版本< 2.1.9，则 `db-type="pb"`。
  - 如果TiDB 版本>= 2.1.9，则 `db-type="file"` 或 `db-type="pb"`。
- 如果下游为 MySQL/TiDB，数据同步后可以使用sync-diff-inspector 进行数据校验。

10.10.2 TiDB Binlog 教程

本文档主要介绍如何使用TiDB Binlog 将数据从TiDB 推送到 MariaDB 实例。文中的TiDB Binlog 集群包含 Pump 和 Drainer 的单个节点，TiDB 集群包含TiDB、TiKV 和 Placement Driver (PD) 各组件的单个节点。

希望上手实践TiDB Binlog 工具的用户需要对TiDB 架构有一定的了解，最好有创建过TiDB 集群的经验。该文档也有助于简单快速了解TiDB Binlog 架构以及相关概念。

> **警告：**
>
> 该文档中部署TiDB 的操作指导不适用于在生产或研发环境中部署TiDB 的情况。

该文档假设用户使用的是现代 Linux 发行版本中的x86-64。示例中使用的是VMware 中运行的 CentOS 7 最小化安装。建议在一开始就进行清洁安装，以避免受现有环境中未知情况的影响。如果不想使用本地虚拟环境，也可以使用云服务启动 CentOS 7 VM。

10.10.2.1 TiDB Binlog 简介

TiDB Binlog 用于收集TiDB 中二进制日志数据、提供实时数据备份和同步以及将TiDB 集群的数据增量同步到下游。

TiDB Binlog 支持以下功能场景：

- 增量备份，将TiDB 集群中的数据增量同步到另一个集群，或通过 Kafka 增量同步到选择的下游。
- 当使用TiDB DM (Data Migration) 将数据从上游 MySQL 或者 MariaDB 迁移到TiDB 集群时，可使用TiDB Binlog 保持 TiDB 集群与其一个独立下游 MySQL 或 MariaDB 实例或集群同步。当TiDB 集群上游数据迁移过程中出现问题，下游数据同步过程中可使用TiDB Binlog 恢复数据到原先的状态。

更多信息参考TiDB Binlog Cluster 版本用户文档。

10.10.2.2 架构

TiDB Binlog 集群由 Pump 和 Drainer 两个组件组成。一个 Pump 集群中有若干个 Pump 节点。TiDB 实例连接到各个 Pump 节点并发送binlog 数据到 Pump 节点。Pump 集群连接到 Drainer 节点，Drainer 将接收到的更新数据转换到某个特定下游（例如 Kafka、另一个 TiDB 集群或 MySQL 或 MariaDB Server）指定的正确格式。

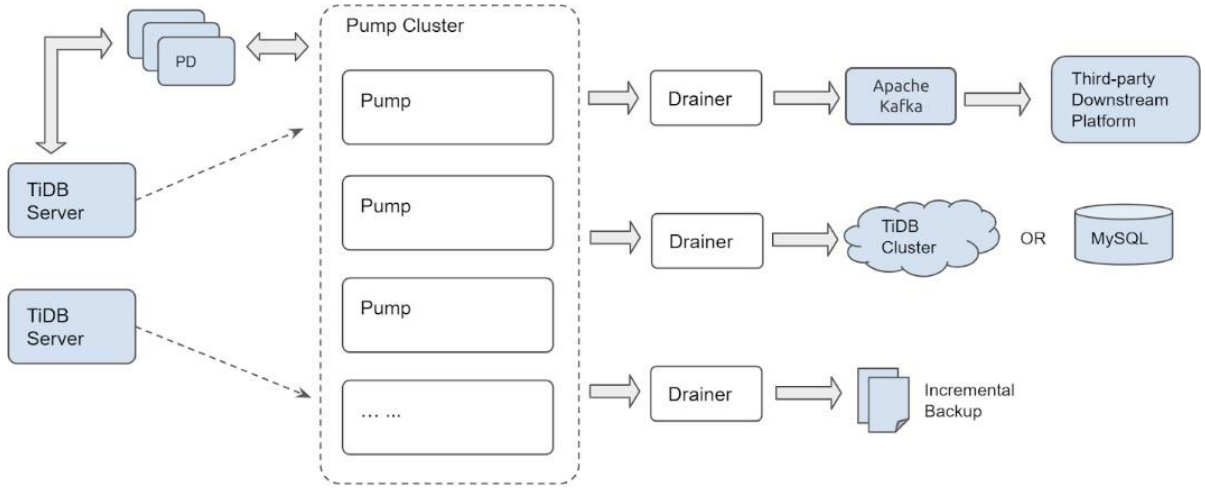


图 227: TiDB Binlog architecture

Pump 的集群架构能确保TiDB 或 Pump 集群中有新的实例加入或退出时更新数据不会丢失。

10.10.2.3 安装

由于 RHEL/CentOS 7 的默认包装库中包括 MariaDB Server，本示例选择的是 MariaDB Server。后续除了安装服务器，也需要安装客户端。安装指令如下：

```
sudo yum install -y mariadb-server
```

```
curl -LO https://download.pingcap.org/tidb-community-server-v7.0.0-linux-amd64.tar.gz | tar xzf -
    ↪ &&
cd tidb-latest-linux-amd64
```

预期输出：

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  368M  100  368M    0     0  8394k      0  0:00:44  0:00:44 --:--:-- 11.1M
```

10.10.2.4 配置

通过执行以下步骤配置一个TiDB 集群，该集群包括 pd-server、tikv-server 和 tidb-server 各组件的单个实例。

1. **填充配置文件：**

```
printf > pd.toml %s\\n 'log-file="pd.log"' 'data-dir="pd.data"' &&
printf > tikv.toml %s\\n 'log-file="tikv.log"' '[storage]' 'data-dir="tikv.data"' '[pd]' '
    ↪ endpoints=["127.0.0.1:2379"]' '[rocksdb]' max-open-files=1024 '[raftdb]' max-open-
    ↪ files=1024 &&
printf > pump.toml %s\\n 'log-file="pump.log"' 'data-dir="pump.data"' 'addr="127.0.0.1:8250"
    ↪ ' 'advertise-addr="127.0.0.1:8250"' 'pd-urls="http://127.0.0.1:2379"' &&
printf > tidb.toml %s\\n 'store="tikv"' 'path="127.0.0.1:2379"' '[log.file]' 'filename="tidb
    ↪ .log"' '[binlog]' 'enable=true' &&
printf > drainer.toml %s\\n 'log-file="drainer.log"' '[syncer]' 'db-type="mysql"' '[syncer.
    ↪ to]' 'host="127.0.0.1"' 'user="root"' 'password=""' 'port=3306'
```

2. **查看配置细节：**

```
for f in *.toml; do echo "$f:"; cat "$f"; echo; done
```

预期输出：

```
drainer.toml:
log-file="drainer.log"
[syncer]
db-type="mysql"
[syncer.to]
host="127.0.0.1"
user="root"
password=""
port=3306
```

```
pd.toml:
log-file="pd.log"
data-dir="pd.data"

pump.toml:
log-file="pump.log"
data-dir="pump.data"
addr="127.0.0.1:8250"
advertise-addr="127.0.0.1:8250"
pd-urls="http://127.0.0.1:2379"

tidb.toml:
store="tikv"
path="127.0.0.1:2379"
[log.file]
filename="tidb.log"
[binlog]
enable=true

tikv.toml:
log-file="tikv.log"
[storage]
data-dir="tikv.data"
[pd]
endpoints=["127.0.0.1:2379"]
[rocksdb]
max-open-files=1024
[raftdb]
max-open-files=1024
```

10.10.2.5 启动程序

现在可启动各个组件。推荐启动顺序依次为 Placement Driver (PD)、 TiKV、 Pump（TiDB 发送 binlog 日志必须连接 Pump 服务）、 TiDB。

1. 启动所有服务：

```
./bin/pd-server --config=pd.toml &>pd.out &
```

```
[1] 20935
```

```
./bin/tikv-server --config=tikv.toml &>tikv.out &
```

```
[2] 20944
```

```
./pump --config=pump.toml &>pump.out &
```

```
[3] 21050
```

```
sleep 3 &&
./bin/tidb-server --config=tidb.toml &>tidb.out &
```

```
[4] 21058
```

2. 如果执行 jobs，可以看到后台正在运行的程序，列表如下：

```
jobs
```

```
[1]   Running                 ./bin/pd-server --config=pd.toml &>pd.out &
[2]   Running                 ./bin/tikv-server --config=tikv.toml &>tikv.out &
[3]-  Running                 ./pump --config=pump.toml &>pump.out &
[4]+  Running                 ./bin/tidb-server --config=tidb.toml &>tidb.out &
```

如果有服务启动失败（例如出现“Exit 1” 而不是“Running”），尝试重启单个组件。

10.10.2.6 连接

按以上步骤操作后，TiDB 的 4 个组件开始运行。接下来可以使用以下 MariaDB 或 MySQL 命令行客户端，通过 4000 端口连接到TiDB 服务：

```
mysql -h 127.0.0.1 -P 4000 -u root -e 'select tidb_version();'
```

预期输出：

```
*************************** 1. row ***************************
tidb_version(): Release Version: v3.0.0-beta.1-154-gd5afff70c
Git Commit Hash: d5afff70cdd825d5fab125c8e52e686cc5fb9a6e
Git Branch: master
UTC Build Time: 2019-04-24 03:10:00
GoVersion: go version go1.12 linux/amd64
Race Enabled: false
TiKV Min Version: 2.1.0-alpha.1-ff3dd160846b7d1aed9079c389fc188f7f5ea13e
Check Table Before Drop: false
```

连接后TiDB 集群已开始运行，pump 读取集群中的binlog 数据，并在其数据目录中将binlog 数据存储为relay log。下一步是启动一个可供 drainer 写入的 MariaDB Server。

1. 启动 drainer：

```
sudo systemctl start mariadb &&
./drainer --config=drainer.toml &>drainer.out &
```

如果你的操作系统更易于安装 MySQL，只需保证监听 3306 端口。另外，可使用密码为空的 “root” 用户连接到 MySQL，或调整 `drainer.toml` 连接到 MySQL。

```
mysql -h 127.0.0.1 -P 3306 -u root
```

预期输出：

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 20
Server version: 5.5.60-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

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

MariaDB [(none)]>
```

```
show databases;
```

预期输出：

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
| tidb_binlog        |
+--------------------+
5 rows in set (0.01 sec)
```

如下表格是包含 `checkpoint` 表格的 `tidb_binlog` 数据库。`drainer` 使用 `checkpoint` 表格，记录 TiDB 集群中的 binlog 已经更新到了哪个位置。

```
use tidb_binlog;
```

```
Database changed
```

```
select * from checkpoint;
```

```
+--------------------+-------------------------------------------+
| clusterID          | checkPoint                                |
+--------------------+-------------------------------------------+
| 6678715361817107733 | {"commitTS":407637466476445697,"ts-map":{}} |
+--------------------+-------------------------------------------+
1 row in set (0.00 sec)
```

打开另一个连接到TiDB 的客户端，创建一个表格并插入几行数据。建议在 GNU Screen 软件中操作，从而同时打开多个客户端。

```
mysql -h 127.0.0.1 -P 4000 --prompt='TiDB [\d]> ' -u root
```

```
create database tidbtest;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
use tidbtest;
```

```
Database changed
```

```
create table t1 (id int unsigned not null AUTO_INCREMENT primary key);
```

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

```
insert into t1 () values (),(),(),(),();
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
select * from t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
+----+
5 rows in set (0.00 sec)
```

切换回 MariaDB 客户端可看到新的数据库、新的表格和最近插入的行数据。

```
use tidbtest;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
show tables;
```

```
+--------------------+
| Tables_in_tidbtest |
+--------------------+
| t1                 |
+--------------------+
1 row in set (0.00 sec)
```

```
select * from t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
+----+
5 rows in set (0.00 sec)
```

可看到查询 MariaDB 时插入到TiDB 相同的行数据，表明TiDB Binlog 安装成功。

10.10.2.7 binlogctl

加入到集群的 Pump 和 Drainer 的数据存储在 Placement Driver (PD) 中。binlogctl 可用于查询和修改状态信息。更多信息请参考binlogctl guide。

使用 `binlogctl` 查看集群中 Pump 和 Drainer 的当前状态：

```
./binlogctl -cmd drainers
```

```
[2019/04/11 17:44:10.861 -04:00] [INFO] [nodes.go:47] ["query node"] [type=drainer] [node="{
    '→ NodeID: localhost.localdomain:8249, Addr: 192.168.236.128:8249, State: online,
    '→ MaxCommitTS: 407638907719778305, UpdateTime: 2019-04-11 17:44:10 -0400 EDT}"]
```

```
./binlogctl -cmd pumps
```

```
[2019/04/11 17:44:13.904 -04:00] [INFO] [nodes.go:47] ["query node"] [type=pump] [node="{NodeID:
    '→ localhost.localdomain:8250, Addr: 192.168.236.128:8250, State: online, MaxCommitTS:
    '→ 407638914024079361, UpdateTime: 2019-04-11 17:44:13 -0400 EDT}"]
```

如果结束 Drainer 进程，集群会改进程设置“已暂停（即集群等待 Drainer 重新加入）” 的状态。

```
pkill drainer &&
./binlogctl -cmd drainers
```

预期输出：

```
[2019/04/11 17:44:22.640 -04:00] [INFO] [nodes.go:47] ["query node"] [type=drainer] [node="{
    ↪ NodeID: localhost.localdomain:8249, Addr: 192.168.236.128:8249, State: paused,
    ↪ MaxCommitTS: 407638915597467649, UpdateTime: 2019-04-11 17:44:18 -0400 EDT}"]
```

使用binlogctl 的“NodeIDs” 可控制单个对应节点。在该情况下，Drainer 的节点ID 是“localhost.localdomain:8249”，Pump 的节点ID 是 “localhost.localdomain:8250”。

本文档中的binlogctl 主要用于集群重启。如果在TiDB 集群中终止并尝试重启所有的进程，由于 Pump 无法连接 Drainer 且认为 Drainer 依旧“在线”，Pump 会拒绝启动。这里的进程并不包括下游的 MySQL 或 MariaDB 或 Drainer。

以下有三个方案可解决上述问题：

- 使用binlogctl 停止 Drainer，而不是结束进程：

```
./binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=drainers &&
./binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=pause-drainer --node-id=localhost.
    ↪ localdomain:8249
```

- 在启动 Pump 之前先启动 Drainer。
- 在启动 PD 之后但在启动 Drainer 和 Pump 之前，使用binlogctl 更新已暂定 Drainer 的状态。

```
./binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=update-drainer --node-id=localhost.
    ↪ localdomain:8249 --state=paused
```

10.10.2.8 清理

在 shell 终端里可启动创建集群的所有进程（pd-server 、 tikv-server、 pump、 tidb-server、 drainer）。可通过在 shell 终端中执行 `pkill -P $$` 停止TiDB 集群服务和TiDB Binlog 进程。按一定的顺序停止这些进程有利于留出足够的时间彻底关闭每个组件。

```
for p in tidb-server drainer pump tikv-server pd-server; do pkill "$p"; sleep 1; done
```

预期输出：

```
[4]-  Done                    ./bin/tidb-server --config=tidb.toml &>tidb.out
[5]+  Done                    ./drainer --config=drainer.toml &>drainer.out
[3]+  Done                    ./pump --config=pump.toml &>pump.out
[2]+  Done                    ./bin/tikv-server --config=tikv.toml &>tikv.out
[1]+  Done                    ./bin/pd-server --config=pd.toml &>pd.out
```

如果需要所有服务退出后重启集群，可以使用一开始启动服务的命令。如以上binlogctl 部分所述，需要先启动 Drainer 再启动 Pump，最后启动 tidb-server。

```
./bin/pd-server --config=pd.toml &>pd.out &
./bin/tikv-server --config=tikv.toml &>tikv.out &
./drainer --config=drainer.toml &>drainer.out &
sleep 3
./pump --config=pump.toml &>pump.out &
sleep 3
./bin/tidb-server --config=tidb.toml &>tidb.out &
```

如果有组件启动失败，请尝试单独重启该组件。

10.10.2.9 总结

本文档介绍了如何通过设置 TiDB Binlog，使用单个 Pump 和 Drainer 组成的集群同步 TiDB 集群数据到下游的 MariaDB。可以发现，TiDB Binlog 是用于获取处理TiDB 集群中更新数据的综合性平台工具。

在更稳健的开发、测试或生产部署环境中，可以使用多个 TiDB 服务以实现高可用性和扩展性。使用多个 Pump 实例可以避免 Pump 集群中的问题影响发送到TiDB 实例的应用流量。或者可以使用增加的 Drainer 实例同步数据到不同的下游或实现数据增量备份。

10.10.3 TiDB Binlog 集群部署

10.10.3.1 服务器要求

Pump 和 Drainer 均可部署和运行在Intel x86-64 架构的 64 位通用硬件服务器平台上。在开发、测试和生产环境下，对服务器硬件配置的要求和建议如下：

服务	部署数量	CPU	磁盘	内存
Pump	3	8 核 +	SSD, 200 GB+	16G
Drainer	1	8 核 +	SAS, 100 GB+（如果输出 binlog 为本地文件，磁盘大小视保留数据天数而定）	16G

10.10.3.2 使用TiUP 部署TiDB Binlog

推荐使用TiUP 部署TiDB Binlog。即在使用 TiUP 部署TiDB 时，在拓扑文件中添加TiDB Binlog 的 `drainer` 和 `pump` 节点信息后，再随TiDB 一起部署。详细部署方式参考TiUP 部署 TiDB 集群。

10.10.3.3 使用 Binary 部署TiDB Binlog

10.10.3.3.1 下载TiDB Binlog 安装包

TiDB Binlog 安装包位于TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

10.10.3.3.2 使用样例

假设有三个 PD，一个 TiDB，另外有两台机器用于部署 Pump，一台机器用于部署 Drainer。各个节点信息如下：

```
TiDB="192.168.0.10"
PD1="192.168.0.16"
PD2="192.168.0.15"
PD3="192.168.0.14"
Pump="192.168.0.11"
Pump="192.168.0.12"
Drainer="192.168.0.13"
```

下面以此为例，说明 Pump/Drainer 的使用。

1. 使用 binary 部署 Pump

    - Pump 命令行参数说明（以在“192.168.0.11”上部署为例）

```
Usage of Pump:
-L string
    日志输出信息等级设置：debug，info，warn，error，fatal（默认 "info"）
-V
    打印版本信息
-addr string
    Pump 提供服务的 RPC 地址(-addr="192.168.0.11:8250")
-advertise-addr string
    Pump 对外提供服务的 RPC 地址(-advertise-addr="192.168.0.11:8250")
-config string
    配置文件路径，如果你指定了配置文件，Pump 会首先读取配置文件的配置；
    如果对应的配置在命令行参数里面也存在，Pump
        ↪ 就会使用命令行参数的配置来覆盖配置文件里的配置。
-data-dir string
    Pump 数据存储位置路径
-gc int
    Pump 只保留多少天以内的数据（默认 7）
-heartbeat-interval int
    Pump 向 PD 发送心跳间隔（单位 秒）
-log-file string
    log 文件路径
-log-rotate string
    log 文件切换频率，hour/day
-metrics-addr string
    Prometheus Pushgateway 地址，不设置则禁止上报监控信息
-metrics-interval int
    监控信息上报频率（默认 15，单位 秒）
-node-id string
    Pump 节点的唯一识别 ID，如果不指定，程序会根据主机名和监听端口自动生成
-pd-urls string
    PD 集群节点的地址（-pd-urls="http://192.168.0.16:2379,http://192.168.0.15:2379,http
        ↪ ://192.168.0.14:2379"）
```

```
-fake-binlog-interval int
    Pump 节点生成 fake binlog 的频率（默认 3，单位 秒）
```

- Pump 配置文件（以在“192.168.0.11”上部署为例）

```
# Pump Configuration

# Pump 绑定的地址
addr = "192.168.0.11:8250"

# Pump 对外提供服务的地址
advertise-addr = "192.168.0.11:8250"

# Pump 只保留多少天以内的数据（默认 7）
gc = 7

# Pump 数据存储位置路径
data-dir = "data.pump"

# Pump 向 PD 发送心跳的间隔（单位 秒）
heartbeat-interval = 2

# PD 集群节点的地址（英文逗号分割，中间不加空格）
pd-urls = "http://192.168.0.16:2379,http://192.168.0.15:2379,http://192.168.0.14:2379"

# [security]
# 如无特殊安全设置需要，该部分一般都注解掉
# 包含与集群连接的受信任 SSL CA 列表的文件路径
# ssl-ca = "/path/to/ca.pem"
# 包含与集群连接的 PEM 形式的 X509 certificate 的路径
# ssl-cert = "/path/to/drainer.pem"
# 包含与集群链接的 PEM 形式的 X509 key 的路径
# ssl-key = "/path/to/drainer-key.pem"

# [storage]
# 设置为 true（默认值）来保证可靠性，确保 binlog 数据刷新到磁盘
# sync-log = true

# 当可用磁盘容量小于该设置值时，Pump 将停止写入数据
# 42 MB -> 42000000, 42 mib -> 44040192
# default: 10 gib
# stop-write-at-available-space = "10 gib"

# Pump 内嵌的 LSM DB 设置，除非对该部分很了解，否则一般注解掉
# [storage.kv]
# block-cache-capacity = 8388608
```

```
# block-restart-interval = 16
# block-size = 4096
# compaction-L0-trigger = 8
# compaction-table-size = 67108864
# compaction-total-size = 536870912
# compaction-total-size-multiplier = 8.0
# write-buffer = 67108864
# write-L0-pause-trigger = 24
# write-L0-slowdown-trigger = 17
```

- 启动示例

```
./pump -config pump.toml
```

如果命令行参数与配置文件中的参数重合，则使用命令行设置的参数的值。

2. 使用 binary 部署 Drainer

- Drainer 命令行参数说明（以在“192.168.0.13”上部署为例）

```
Usage of Drainer
-L string
    日志输出信息等级设置：debug，info，warn，error，fatal（默认 "info"）
-V
    打印版本信息
-addr string
    Drainer 提供服务的地址(-addr="192.168.0.13:8249")
-c int
    同步下游的并发数，该值设置越高同步的吞吐性能越好（default 1）
-cache-binlog-count int
    缓存中的 binlog 数目限制（默认 8）
    如果上游的单个 binlog 较大导致 Drainer 出现 OOM 时，可尝试调小该值减少内存使用
-config string
    配置文件路径，Drainer 会首先读取配置文件的配置；
    如果对应的配置在命令行参数里面也存在，Drainer
        '→ 就会使用命令行参数的配置来覆盖配置文件里面的配置
-data-dir string
    Drainer 数据存储位置路径（默认 "data.drainer"）
-dest-db-type string
    Drainer 下游服务类型（默认为 mysql，支持 tidb、kafka、file）
-detect-interval int
    向 PD 查询在线 Pump 的时间间隔（默认 10，单位 秒）
-disable-detect
    是否禁用冲突监测
-disable-dispatch
    是否禁用拆分单个 binlog 的 SQL 的功能，如果设置为 true，则每个 binlog
    按顺序依次还原成单个事务进行同步（下游服务类型为 MySQL，该项设置为 False）
```

```
-ignore-schemas string
    db 过滤列表 (默认 "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql,test"),
    不支持对 ignore schemas 的 table 进行 rename DDL 操作
-initial-commit-ts (默认为 `-1`)
    如果 Drainer 没有相关的断点信息，可以通过该项来设置相关的断点信息
    该参数值为 `-1` 时，Drainer 会自动从 PD 获取一个最新的时间戳
-log-file string
    log 文件路径
-log-rotate string
    log 文件切换频率，hour/day
-metrics-addr string
    Prometheus Pushgateway 地址，不设置则禁止上报监控信息
-metrics-interval int
    监控信息上报频率 (默认 15，单位：秒)
-node-id string
    drainer 节点的唯一识别 ID，如果不指定，程序会根据主机名和监听端口自动生成
-pd-urls string
    PD 集群节点的地址 (-pd-urls="http://192.168.0.16:2379,http://192.168.0.15:2379,http
        ↪ ://192.168.0.14:2379")
-safe-mode
    是否开启安全模式使得下游 MySQL/TiDB 可被重复写入
    即将 insert 语句换为 replace 语句，将 update 语句拆分为 delete + replace 语句
-txn-batch int
    输出到下游数据库一个事务的 SQL 数量 (默认 1)
```

- Drainer 配置文件（以在“192.168.0.13”上部署为例）

```
# Drainer Configuration.

# Drainer 提供服务的地址("192.168.0.13:8249")
addr = "192.168.0.13:8249"

# Drainer 对外提供服务的地址
advertise-addr = "192.168.0.13:8249"

# 向 PD 查询在线 Pump 的时间间隔 (默认 10，单位 秒)
detect-interval = 10

# Drainer 数据存储位置路径 (默认 "data.drainer")
data-dir = "data.drainer"

# PD 集群节点的地址 (英文逗号分割，中间不加空格)
pd-urls = "http://192.168.0.16:2379,http://192.168.0.15:2379,http://192.168.0.14:2379"

# log 文件路径
log-file = "drainer.log"
```

```
# Drainer 从 Pump 获取 binlog 时对数据进行压缩，值可以为 "gzip"，如果不配置则不进行压缩
# compressor = "gzip"

# [security]
# 如无特殊安全设置需要，该部分一般都注解掉
# 包含与集群连接的受信任 SSL CA 列表的文件路径
# ssl-ca = "/path/to/ca.pem"
# 包含与集群连接的 PEM 形式的 X509 certificate 的路径
# ssl-cert = "/path/to/pump.pem"
# 包含与集群链接的 PEM 形式的 X509 key 的路径
# ssl-key = "/path/to/pump-key.pem"

# Syncer Configuration
[syncer]
# 如果设置了该项，会使用该 sql-mode 解析 DDL 语句，此时如果下游是 MySQL 或 TiDB 则
# 下游的 sql-mode 也会被设置为该值
# sql-mode = "STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"

# 输出到下游数据库一个事务的 SQL 语句数量（默认 20）
txn-batch = 20

# 同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）
worker-count = 16

# 是否禁用拆分单个 binlog 的 SQL 的功能，如果设置为 true，则按照每个 binlog
# 顺序依次还原成单个事务进行同步（下游服务类型为 MySQL，该项设置为 False）
disable-dispatch = false

# safe mode 会使写下游 MySQL/TiDB 可被重复写入
# 会用 replace 替换 insert 语句，用 delete + replace 替换 update 语句
safe-mode = false

# Drainer 下游服务类型（默认为 mysql）
# 参数有效值为 "mysql"，"tidb"，"file"，"kafka"
db-type = "mysql"

# 事务的 commit ts 若在该列表中，则该事务将被过滤，不会同步至下游
ignore-txn-commit-ts = []

# db 过滤列表（默认 "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql,test"），
# 不支持对 ignore schemas 的 table 进行 rename DDL 操作
ignore-schemas = "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"

# replace-do-db 配置的优先级高于 replace-do-table。如果配置了相同的库名，
```

```
    '→ 支持使用正则表达式进行配置。
# 以 '~' 开始声明使用正则表达式

# replicate-do-db = ["~^b.*","s1"]

# [syncer.relay]
# 保存 relay log 的目录，空值表示不开启。
# 只有下游是 TiDB 或 MySQL 时该配置才生效。
# log-dir = ""
# 每个文件的大小上限
# max-file-size = 10485760

# [[syncer.replicate-do-table]]
# db-name ="test"
# tbl-name = "log"

# [[syncer.replicate-do-table]]
# db-name ="test"
# tbl-name = "~^a.*"

# 忽略同步某些表
# [[syncer.ignore-table]]
# db-name = "test"
# tbl-name = "log"

# db-type 设置为 mysql 时，下游数据库服务器参数
[syncer.to]
host = "192.168.0.13"
user = "root"
# 如果你不想在配置文件中写明文密码，则可以使用 `./binlogctl -cmd encrypt -text string`
    '→ 生成加密的密码
# 如果配置了 encrypted_password 且非空，那么配置的 password 不生效。encrypted_password
    '→ 和 password 无法同时生效。
password = ""
encrypted_password = ""
port = 3306

[syncer.to.checkpoint]
# 当 checkpoint type 是 mysql 或 tidb 时可以开启该选项，以改变保存 checkpoint 的数据库
# schema = "tidb_binlog"
# 目前只支持 mysql 或者 tidb 类型。可以去掉注释来控制 checkpoint 保存的位置。
# db-type 默认的 checkpoint 保存方式是:
# mysql/tidb -> 对应的下游 mysql/tidb
# file/kafka -> file in `data-dir`
# type = "mysql"
```

```
# host = "127.0.0.1"
# user = "root"
# password = ""
# 使用 `./binlogctl -cmd encrypt -text string` 加密的密码
# encrypted_password 非空时 password 会被忽略
# encrypted_password = ""
# port = 3306

# db-type 设置为 file 时，存放 binlog 文件的目录
# [syncer.to]
# dir = "data.drainer"

# db-type 设置为 kafka 时，Kafka 相关配置
# [syncer.to]
# kafka-addrs 和 zookeeper-addrs 只需要一个，两者都有时程序会优先用 zookeeper 中的
    '→ kafka 地址
# zookeeper-addrs = "127.0.0.1:2181"
# kafka-addrs = "127.0.0.1:9092"
# kafka-version = "0.8.2.0"
# 配置单条 broker request 中的最大 message 数（即 binlog 数），不配置或配置小于等于 0
    '→ 时会使用默认值 1024
# kafka-max-messages = 1024
# 配置单条 broker request 的最大 size（单位为 Byte），默认为 1 GiB，最大可配置为 2 GiB
# kafka-max-message-size = 1073741824

# 保存 binlog 数据的 Kafka 集群的 topic 名称，默认值为 <cluster-id>_obinlog
# 如果运行多个 Drainer 同步数据到同一个 Kafka 集群，每个 Drainer 的 topic-name
    '→ 需要设置不同的名称
# topic-name = ""
```

- 启动示例

> **注意：**
> 如果下游为 MySQL/TiDB，为了保证数据的完整性，在 Drainer 初次启动前需要获取 initial-commit-ts 的值，并进行全量数据的备份与恢复。

初次启动时使用参数 initial-commit-ts，命令如下：

```
./drainer -config drainer.toml -initial-commit-ts {initial-commit-ts}
```

如果命令行参数与配置文件中的参数重合，则使用命令行设置的参数的值。

> **注意：**

- 在运行TiDB 时，需要保证至少一个 Pump 正常运行。
- 通过给TiDB 增加启动参数 `enable-binlog` 来开启binlog 服务。尽量保证同一集群的所有 TiDB 都开启了binlog 服务，否则在同步数据时可能会导致上下游数据不一致。如果要临时运行一个不开启binlog 服务的TiDB 实例，需要在TiDB 的配置文件中设置 `run_ddl= false`。
- Drainer 不支持对ignore schemas（在过滤列表中的 schemas）的 table 进行 rename DDL 操作。
- 在已有的TiDB 集群中启动 Drainer，一般需要全量备份并且获取快照时间戳，然后导入全量备份，最后启动 Drainer 从对应的快照时间戳开始同步增量数据。
- 下游使用 MySQL 或 TiDB 时应当保证上下游数据库的 sql_mode 具有一致性，即下游数据库同步每条 SQL 语句时的 sql_mode 应当与上游数据库执行该条 SQL 语句时的 sql_mode 保持一致。可以在上下游分别执行 `select @@sql_mode;` 进行查询和比对。
- 如果存在上游 TiDB 能运行但下游 MySQL 不支持的 DDL 语句时（例如下游 MySQL 使用 InnoDB 引擎时同步语句 `CREATE TABLE t1(a INT)ROW_FORMAT=FIXED;`），Drainer 也会同步失败，此时可以在 Drainer 配置中跳过该事务，同时在下游手动执行兼容的语句，详见跳过事务。

10.10.4 TiDB Binlog 集群运维

本文首先介绍 Pump 和 Drainer 的状态及启动、退出的内部处理流程，然后说明如何通过binlogctl 工具或者直接在 TiDB 执行 SQL 操作来管理binlog 集群，最后的FAQ 部分会介绍一些常见问题以及处理方法。

10.10.4.1 Pump/Drainer 的状态

Pump/Drainer 中状态的定义：

- online：正常运行中
- pausing：暂停中
- paused：已暂停
- closing：下线中
- offline：已下线

这些状态由 Pump/Drainer 服务自身进行维护，并定时将状态信息更新到 PD 中。

10.10.4.2 Pump/Drainer 的启动、退出流程

10.10.4.2.1 Pump

- 启动：Pump 启动时会通知所有 online 状态的 Drainer，如果通知成功，则 Pump 将状态设置为 online，否则 Pump 将报错，然后将状态设置为paused 并退出进程。
- 退出：Pump 进程正常退出前要选择进入暂停或者下线状态；非正常退出（kill -9、进程panic、宕机）都依然保持 online 状态。

- 暂停：使用 kill（非 kill -9）、Ctrl+C 或者使用binlogctl 的pause-pump 命令都可以暂停 Pump。接收到暂停指令后，Pump 会变更状态为pausing，并停止接受binlog 的写请求，也停止向 Drainer 提供binlog 数据。安全退出所有线程后，更新状态为paused 然后退出进程。
- 下线：仅在使用binlogctl 的 offline-pump 命令的情况下才会下线 Pump。接收到下线指令后，Pump 会变更状态为closing，并停止接受binlog 的写请求。Pump 继续向 Drainer 提供binlog，等待所有binlog 数据都被 Drainer 消费后再将状态设置为 offline 并退出进程。

10.10.4.2.2 Drainer

- 启动：Drainer 启动时将状态设置为 online，并尝试从所有非 offline 状态的 Pump 获取 binlog，如果获取 binlog 失败，会不断尝试重新获取。
- 退出：Drainer 进程正常退出前要选择进入暂停或者下线状态；非正常退出（kill -9、进程panic、宕机）都依然保持 online 状态。
  - 暂停：使用 kill（非 kill -9）、Ctrl+C 或者使用binlogctl 的pause-drainer 命令都可以暂停 Drainer。接收到指令后，Drainer 会变更状态为pausing，并停止从 Pump 获取binlog。安全退出所有线程后，更新状态为paused 然后退出进程。
  - 下线：仅在使用 binlogctl 的 offline-drainer 命令的情况下才会下线 Drainer。接收到下线指令后，Drainer 变更状态为closing，并停止从 Pump 获取binlog。安全退出所有线程后，更新状态为 offline 然后退出进程。

关于 Pump/Drainer 暂停、下线、状态查询、状态修改等具体的操作方法，参考如下binlogctl 工具的使用方法介绍。

10.10.4.3 使用binlogctl 工具管理 Pump/Drainer

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

详细的介绍和使用方法请参考binlogctl 工具。

10.10.4.4 使用TiDB SQL 管理 Pump/Drainer

要查看和管理binlog 相关的状态，可在TiDB 中执行相应的 SQL 语句。

- 查看TiDB 是否开启binlog，0 代表关闭，1 代表开启

```
show variables like "log_bin";
```

```
+---------------+-------------+
| Variable_name | Value |
+---------------+-------------+
| log_bin       |  0    |
+---------------+-------------+
```

- 查看 Pump/Drainer 状态

```
show pump status;
```

```
+--------|----------------|--------|--------------------|---------------------------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time       |
+--------|----------------|--------|--------------------|---------------------------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------------------------|
```

```
show drainer status;
```

```
+----------|----------------|--------|--------------------|---------------------------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time       |
+----------|----------------|--------|--------------------|---------------------------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------------------------|
```

- 异常情况下修改 Pump/Drainer 状态

```
change pump to node_state ='paused' for node_id 'pump1';
```

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

```
change drainer to node_state ='paused' for node_id 'drainer1';
```

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

该 SQL 的功能和binlogctl 中的update-pump 和 update-drainer 命令的功能一样，因此也只有在 Pump/Drainer 异常的情况下使用。

注意：

1. 查看binlog 开启状态以及 Pump/Drainer 状态的功能在TiDB v2.1.7 及以上版本中支持。
2. 修改 Pump/Drainer 状态的功能在TiDB v3.0.0-rc.1 及以上版本中支持。该功能只修改 PD 中存储的 Pump/Drainer 状态，如果需要暂停/下线节点，仍然需要使用 `binlogctl`。

10.10.5 TiDB Binlog 配置说明

本文档介绍TiDB Binlog 的各项配置说明。

10.10.5.1 Pump

本节介绍 Pump 的配置项。可以在 Pump Configuration 中查看完整的 Pump 配置文件示例。

10.10.5.1.1 addr

- HTTP API 的监听地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8250"`

10.10.5.1.2 advertise-addr

- 对外可访问的 HTTP API 地址。这个地址会被注册到 PD，格式为 `host:port`。
- 默认：与 addr 的配置相同。

10.10.5.1.3 socket

- HTTP API 监听的 Unix socket 地址。
- 默认：`""`

10.10.5.1.4 pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址时出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

10.10.5.1.5 data-dir

- 本地存放binlog 及其索引的目录。
- 默认：`"data.pump"`

10.10.5.1.6 heartbeat-interval

- 心跳间隔，即每隔指定秒数向 PD 汇报最新的状态。
- 默认：2

10.10.5.1.7 gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

10.10.5.1.8 gc

- 指定binlog 可在本地存储的天数（整型）。超过指定天数的 binlog 会被自动删除。
- 默认：7

10.10.5.1.9 log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认："""

10.10.5.1.10 log-level

- Log 等级。
- 默认："info"

10.10.5.1.11 node-id

- Pump 节点的ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 node-1:8250。

10.10.5.1.12 security

以下是与安全相关的配置项。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 /path/to/ca.pem。
- 默认："""

ssl-cert

- 包含PEM 格式编码的X509 证书文件路径，例如 /path/to/pump.pem。
- 默认："""

ssl-key

- 包含PEM 格式编码的X509 Key 文件路径，例如 /path/to/pump-key.pem。
- 默认："""

10.10.5.1.13 storage

以下是与存储相关的配置项。

sync-log

- 指定是否在每次批量写入binlog 后使用 fsync 以确保数据安全。
- 默认：true

kv_chan_cap

- 在 Pump 接收写入请求时会先将请求放入一个缓冲区，该项指定缓冲区能存放的请求数量。

- 默认：1048576（即 2 的 20 次方）

slow_write_threshold

- 写入单个binlog的耗时超过该项设定的秒数就被认为是慢写入，并输出一条包含 "take a long time to write binlog" 的日志。
- 默认：1

stop-write-at-available-space

- 可用存储空间低于指定值时不再接收binlog写入请求。可以用例如 900 MB、5 GB、12 GiB 的格式指定空间大小。如果集群中 Pump 节点多于一个，那么在某个 Pump 节点因为空间不足而拒绝写入时，TiDB 端会自动写入到其他 Pump 节点。
- 默认：10 GiB

kv

目前 Pump 的存储是基于GoLevelDB 实现的。storage 下还有一个 kv 子分组可以用于调整GoLevelDB 的配置，支持的配置项包括：

- block-cache-capacity
- block-restart-interval
- block-size
- compaction-L0-trigger
- compaction-table-size
- compaction-total-size
- compaction-total-size-multiplier
- write-buffer
- write-L0-pause-trigger
- write-L0-slowdown-trigger

配置具体含义可在GoLevelDB 文档中查看。

10.10.5.2 Drainer

本节介绍 Drainer 的配置项。可以在 Drainer Configuration 中查看完整的配置文件示例。

10.10.5.2.1 addr

- HTTP API 的监听的地址，格式为 host:port。
- 默认："127.0.0.1:8249"

10.10.5.2.2 advertise-addr

- 对外可访问的 HTTP API 地址，这个地址会被注册到 PD，格式为 host:port。
- 默认：设定成与 addr 相同的配置

10.10.5.2.3 log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认："""

10.10.5.2.4 log-level

- Log 等级。
- 默认："info"

10.10.5.2.5 node-id

- Drainer 节点ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 node-1:8249。

10.10.5.2.6 data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认："data.drainer"

10.10.5.2.7 detect-interval

- 每隔指定秒数从 PD 更新一次 Pumps 信息，以获取节点加入或离开等事件。
- 默认：5

10.10.5.2.8 pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址出错时会自动尝试连接另一个地址。
- 默认："http://127.0.0.1:2379"

10.10.5.2.9 initial-commit-ts

- 指定从哪个事务提交时间点（事务的 commit ts）之后开始同步。这个配置仅适用于初次开始同步的 Drainer 节点。如果下游已经有checkpoint 存在，则会根据checkpoint 里记录的时间进行同步。
- commit ts（即 commit timestamp）是 TiDB 事务的提交时间点。该时间点是从 PD 获取的全局唯一递增的时间戳，作为当前事务的唯一ID。典型的 initial-commit-ts 配置可以通过以下方式获得：
  - BR 备份的元信息（即 backupmeta）中记录的backup TS
  - Dumpling 备份的元信息（即metadata）中记录的 Pos
  - PD Control 中 tso 命令返回的结果
- 默认：-1。Drainer 会从 PD 得到一个最新的timestamp 作为初始时间。即从当前的时间点开始同步。

10.10.5.2.10 synced-check-time

- 通过 HTTP API 访问 `/status` 路径可以查询 Drainer 同步的状态。`synced-check-time` 指定距离上次成功同步的时间超过多少分钟可以认为是 `synced`，即同步完成。
- 默认：5

10.10.5.2.11 compressor

- 指定 Pump 与 Drainer 间的数据传输所用的压缩算法。目前仅支持一种算法，即 `gzip`。
- 默认：`""`，表示不压缩。

10.10.5.2.12 security

以下是与 Drainer 安全相关的配置。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。
- 默认：`""`

ssl-cert

- 包含PEM 格式编码的X509 证书文件路径，例如 `/path/to/drainer.pem`。
- 默认：`""`

ssl-key

- 包含PEM 格式编码的X509 Key 文件路径，例如 `/path/to/drainer-key.pem`。
- 默认：`""`

10.10.5.2.13 syncer

syncer 分组包含一些与同步下游相关的配置项。

db-type

下游类型，目前支持的选项有：

- `mysql`
- `tidb`
- `kafka`
- `file`

默认：`mysql`

sql-mode

- 当下游为 mysql/tidb 类型时，可以指定 SQL mode，如果超过一个mode，则用逗号隔开。
- 默认：""

ignore-txn-commit-ts

- 同步时，该项所指定的commit timestamp 的binlog 会被忽略，例如[416815754209656834，421349811963822081]
  ′→ 。
- 默认：[]

ignore-schemas

- 同步时忽略指定数据库的变更。如果超过一个需忽略的数据库，则用逗号隔开。如果一个binlog 中的变更全部被过滤掉，则忽略整个binlog。
- 默认："INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"

ignore-table

同步时忽略指定表格的变更。在 toml 中可以用以下方式指定多个需忽略的表格：

```
[[syncer.ignore-table]]
db-name = "test"
tbl-name = "log"

[[syncer.ignore-table]]
db-name = "test"
tbl-name = "audit"
```

如果一个binlog 中的变更全部被过滤掉，则忽略整个binlog。

默认：[]

replicate-do-db

- 指定要同步的数据库，例如 [db1, db2]。
- 默认：[]

replicate-do-table

指定要同步的表格，示例如下：

```
[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "log"

[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "~^a.*"
```

默认：[]

txn-batch

- 当下游为 mysql/tidb 类型时，会将 DML 分批执行。这个配置可以用于设置每个事务中包含多少个 DML。
- 默认：20

worker-count

- 当下游为 mysql/tidb 类型时，会并发执行 DML，worker-count 可以指定并发数。
- 默认：16

disable-dispatch

- 关掉并发，强制将 worker-count 置为 1。
- 默认：false

safe-mode

如果打开 Safe mode，Drainer 会对同步的变更作以下修改，使其变成可重入的操作：

- Insert 变为 Replace Into
- Update 变为 Delete 和 Replace Into

默认：false

10.10.5.2.14 syncer.to

不同类型的下游配置都在 syncer.to 分组。以下按配置类型进行介绍。

mysql/tidb

用于连接下游数据库的配置项：

- host：如果没有设置，会尝试检查环境变量 MYSQL_HOST，默认值为 "localhost"。
- port：如果没有设置，会尝试检查环境变量 MYSQL_PORT，默认值为 3306。
- user：如果没有设置，会尝试检查环境变量 MYSQL_USER，默认值为 "root"。
- password：如果没有设置，会尝试检查环境变量 MYSQL_PSWD，默认值为 ""。
- read-timeout：指定下游数据库连接的 IO 读取超时时间，默认值为 1m。如果 drainer 在一些耗时长的 DDL 上不断失败，你可以将这个变量设置为更大的值。

file

- dir：指定用于保存 binlog 的目录。如果不指定该项，则使用 data-dir。

kafka

当下游为 kafka 时，有效的配置包括：

- zookeeper-addrs
- kafka-addrs
- kafka-version
- kafka-max-messages
- kafka-max-message-size
- topic-name

10.10.5.2.15 syncer.to.checkpoint

- type：指定用哪种方式保存同步进度，目前支持的选项为 mysql、tidb 和 file。

  该配置选项默认与下游类型相同。例如 file 类型的下游checkpoint 进度保存在本地文件 <data-dir>/savepoint 中，mysql 类型的下游进度保存在下游数据库。当明确指定要使用 mysql 或 tidb 保存同步进度时，需要指定以下配置项：

- schema：默认为 "tidb_binlog"。

> **注意：**
> 在同个 TiDB 集群中部署多个 Drainer 时，需要为每个 Drainer 节点指定不同的 checkpoint schema，否则两个实例的同步进度会互相覆盖。

- host
- user
- password
- port

10.10.5.3 TiDB Binlog 配置说明

本文档介绍TiDB Binlog 的各项配置说明。

10.10.5.3.1 Pump

本节介绍 Pump 的配置项。可以在 Pump Configuration 中查看完整的 Pump 配置文件示例。

addr

- HTTP API 的监听地址，格式为 host:port。
- 默认："127.0.0.1:8250"

advertise-addr

- 对外可访问的 HTTP API 地址。这个地址会被注册到 PD，格式为 host:port。
- 默认：与 addr 的配置相同。

socket

- HTTP API 监听的 Unix socket 地址。
- 默认：""

pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址时出错时会自动尝试连接另一个地址。
- 默认："http://127.0.0.1:2379"

data-dir

- 本地存放binlog 及其索引的目录。
- 默认："data.pump"

heartbeat-interval

- 心跳间隔，即每隔指定秒数向 PD 汇报最新的状态。
- 默认：2

gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

gc

- 指定binlog 可在本地存储的天数（整型）。超过指定天数的 binlog 会被自动删除。
- 默认：7

log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认：""

log-level

- Log 等级。
- 默认："info"

node-id

- Pump 节点的ID，用于在集群中识别这个进程。

- 默认：主机名:端口号，例如 node-1:8250。

security

以下是与安全相关的配置项。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 /path/to/ca.pem。
- 默认：""

ssl-cert

- 包含PEM 格式编码的X509 证书文件路径，例如 /path/to/pump.pem。
- 默认：""

ssl-key

- 包含PEM 格式编码的X509 Key 文件路径，例如 /path/to/pump-key.pem。
- 默认：""

storage

以下是与存储相关的配置项。

sync-log

- 指定是否在每次批量写入binlog 后使用 fsync 以确保数据安全。
- 默认：true

kv_chan_cap

- 在 Pump 接收写入请求时会先将请求放入一个缓冲区，该项指定缓冲区能存放的请求数量。
- 默认：1048576（即 2 的 20 次方）

slow_write_threshold

- 写入单个binlog 的耗时超过该项设定的秒数就被认为是慢写入，并输出一条包含 "take a long time to write binlog" 的日志。
- 默认：1

stop-write-at-available-space

- 可用存储空间低于指定值时不再接收binlog 写入请求。可以用例如 900 MB、5 GB、12 GiB 的格式指定空间大小。如果集群中 Pump 节点多于一个，那么在某个 Pump 节点因为空间不足而拒绝写入时，TiDB 端会自动写入到其他 Pump 节点。
- 默认：10 GiB

kv

目前 Pump 的存储是基于GoLevelDB 实现的。 `storage` 下还有一个 kv 子分组可以用于调整GoLevelDB 的配置，支持的配置项包括：

- block-cache-capacity
- block-restart-interval
- block-size
- compaction-L0-trigger
- compaction-table-size
- compaction-total-size
- compaction-total-size-multiplier
- write-buffer
- write-L0-pause-trigger
- write-L0-slowdown-trigger

配置具体含义可在GoLevelDB 文档中查看。

10.10.5.3.2 Drainer

本节介绍 Drainer 的配置项。 可以在 Drainer Configuration 中查看完整的配置文件示例。

addr

- HTTP API 的监听的地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8249"`

advertise-addr

- 对外可访问的 HTTP API 地址，这个地址会被注册到 PD，格式为 `host:port`。
- 默认：设定成与 `addr` 相同的配置

log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认：`""`

log-level

- Log 等级。
- 默认：`"info"`

node-id

- Drainer 节点ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 `node-1:8249`。

data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认：`"data.drainer"`

detect-interval

- 每隔指定秒数从 PD 更新一次 Pumps 信息，以获取节点加入或离开等事件。
- 默认：5

pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

initial-commit-ts

- 指定从哪个事务提交时间点（事务的 commit ts）之后开始同步。这个配置仅适用于初次开始同步的 Drainer 节点。如果下游已经有checkpoint 存在，则会根据checkpoint 里记录的时间进行同步。
- commit ts（即 commit timestamp）是 TiDB 事务的提交时间点。该时间点是从 PD 获取的全局唯一递增的时间戳，作为当前事务的唯一ID。典型的 `initial-commit-ts` 配置可以通过以下方式获得：
  - BR 备份的元信息（即 backupmeta）中记录的backup TS
  - Dumpling 备份的元信息（即metadata）中记录的 Pos
  - PD Control 中 `tso` 命令返回的结果
- 默认：-1。Drainer 会从 PD 得到一个最新的timestamp 作为初始时间。即从当前的时间点开始同步。

synced-check-time

- 通过 HTTP API 访问 `/status` 路径可以查询 Drainer 同步的状态。`synced-check-time` 指定距离上次成功同步的时间超过多少分钟可以认为是 `synced`，即同步完成。
- 默认：5

compressor

- 指定 Pump 与 Drainer 间的数据传输所用的压缩算法。目前仅支持一种算法，即 `gzip`。
- 默认：`""`，表示不压缩。

security

以下是与 Drainer 安全相关的配置。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。

- 默认：""

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/drainer.pem`。
- 默认：""

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/drainer-key.pem`。
- 默认：""

syncer

syncer 分组包含一些与同步下游相关的配置项。

db-type

下游类型，目前支持的选项有：

- `mysql`
- `tidb`
- `kafka`
- `file`

默认：`mysql`

sql-mode

- 当下游为 `mysql/tidb` 类型时，可以指定 SQL mode，如果超过一个mode，则用逗号隔开。
- 默认：""

ignore-txn-commit-ts

- 同步时，该项所指定的commit timestamp 的binlog 会被忽略，例如`[416815754209656834, 421349811963822081]`。
- 默认：`[]`

ignore-schemas

- 同步时忽略指定数据库的变更。如果超过一个需忽略的数据库，则用逗号隔开。如果一个binlog 中的变更全部被过滤掉，则忽略整个binlog。
- 默认：`"INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"`

ignore-table

同步时忽略指定表格的变更。在 `toml` 中可以用以下方式指定多个需忽略的表格：

```
[[syncer.ignore-table]]
db-name = "test"
tbl-name = "log"

[[syncer.ignore-table]]
db-name = "test"
tbl-name = "audit"
```

如果一个binlog 中的变更全部被过滤掉，则忽略整个binlog。

默认：[]

replicate-do-db

- 指定要同步的数据库，例如 [db1, db2]。
- 默认：[]

replicate-do-table

指定要同步的表格，示例如下：

```
[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "log"

[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "~^a.*"
```

默认：[]

txn-batch

- 当下游为 mysql/tidb 类型时，会将 DML 分批执行。这个配置可以用于设置每个事务中包含多少个 DML。
- 默认：20

worker-count

- 当下游为 mysql/tidb 类型时，会并发执行 DML，worker-count 可以指定并发数。
- 默认：16

disable-dispatch

- 关掉并发，强制将 worker-count 置为 1。
- 默认：false

safe-mode

如果打开 Safe mode，Drainer 会对同步的变更作以下修改，使其变成可重入的操作：

- `Insert` 变为 `Replace Into`
- `Update` 变为 `Delete` 和 `Replace Into`

默认：`false`

syncer.to

不同类型的下游配置都在 `syncer.to` 分组。以下按配置类型进行介绍。

mysql/tidb

用于连接下游数据库的配置项：

- `host`：如果没有设置，会尝试检查环境变量 `MYSQL_HOST`，默认值为 `"localhost"`。
- `port`：如果没有设置，会尝试检查环境变量 `MYSQL_PORT`，默认值为 `3306`。
- `user`：如果没有设置，会尝试检查环境变量 `MYSQL_USER`，默认值为 `"root"`。
- `password`：如果没有设置，会尝试检查环境变量 `MYSQL_PSWD`，默认值为 `""`。
- `read-timeout`：指定下游数据库连接的 IO 读取超时时间，默认值为 `1m`。如果 drainer 在一些耗时长的 DDL 上不断失败，你可以将这个变量设置为更大的值。

file

- `dir`：指定用于保存 binlog 的目录。如果不指定该项，则使用 `data-dir`。

kafka

当下游为 `kafka` 时，有效的配置包括：

- `zookeeper-addrs`
- `kafka-addrs`
- `kafka-version`
- `kafka-max-messages`
- `kafka-max-message-size`
- `topic-name`

syncer.to.checkpoint

- `type`：指定用哪种方式保存同步进度，目前支持的选项为 `mysql`、`tidb` 和 `file`。

  该配置选项默认与下游类型相同。例如 `file` 类型的下游checkpoint 进度保存在本地文件 `<data-dir>/′→ savepoint` 中，`mysql` 类型的下游进度保存在下游数据库。当明确指定要使用 `mysql` 或 `tidb` 保存同步进度时，需要指定以下配置项：

- `schema`：默认为 `"tidb_binlog"`。

> **注意：**
> 在同个 TiDB 集群中部署多个 Drainer 时，需要为每个 Drainer 节点指定不同的 checkpoint schema，否则两个实例的同步进度会互相覆盖。

- host
- user
- password
- port

10.10.6 TiDB Binlog 版本升级方法

如未特别指明，文中出现的TiDB Binlog 均指最新的Cluster 版本。

本文介绍通过手动部署的TiDB Binlog 的版本升级方法，另外有一小节介绍如何从更早的不兼容版本（Kafka/Local版本）升级到最新版本。

10.10.6.1 手动部署

10.10.6.1.1 升级 Pump

对集群里的每个 Pump 逐一升级，确保集群中总有 Pump 可以接收TiDB 发来的 Binlog。

1. 用新版本的 pump 替换原来的文件
2. 重启 Pump 进程

10.10.6.1.2 升级 Drainer

1. 用新版本的 drainer 替换原来的文件
2. 重启 Drainer 进程

10.10.6.2 从 Kafka/Local 版本升级到 Cluster 版本

新版本的TiDB（v2.0.8-binlog、v2.1.0-rc.5 及以上版本）不兼容 Kafka 版本以及 Local 版本的TiDB Binlog，集群升级到新版本后只能使用 Cluster 版本的TiDB Binlog。如果在升级前已经使用了 Kafka / Local 版本的TiDB Binlog，必须将其升级到 Cluster 版本。

TiDB Binlog 版本与TiDB 版本的对应关系如下：

TiDB Binlog 版本	TiDB 版本	说明
Local	TiDB 1.0 及更低版本	

TiDB Binlog 版本	TiDB 版本	说明
Kafka	TiDB 1.0 ~ TiDB 2.1 RC5	TiDB 1.0 支持 local 版本和 Kafka 版本的 TiDB Bin-log。
Cluster	TiDB v2.0.8-binlog，TiDB 2.1 RC5 及更高版本	TiDB v2.0.8-binlog 是一个支持 Clus-ter 版本 TiDB Binlog 的 2.0 特殊版本。

10.10.6.2.1 升级流程

> **注意：**
>
> 如果能接受重新导全量数据，则可以直接废弃老版本，按TiDB Binlog 集群部署中的步骤重新部署。

如果想从原来的checkpoint 继续同步，使用以下升级流程：

1. 部署新版本 Pump。

2. 暂停TiDB 集群业务。
3. 更新TiDB 以及配置，写 Binlog 到新的 Pump Cluster。
4. TiDB 集群重新接入业务。
5. 确认老版本的 Drainer 已经将老版本的 Pump 的数据完全同步到下游。

   查询 Drainer 的 `status` 接口，示例命令如下：

```
curl 'http://172.16.10.49:8249/status'
```

```
{"PumpPos":{"172.16.10.49:8250":{"offset":32686}},"Synced": true ,"DepositWindow":{"Upper
    ↪ ":398907800202772481,"Lower":398907799455662081}}
```

   如果返回的 Synced 为true，则可以认为 Binlog 数据已经全部同步到了下游。

6. 启动新版本 Drainer；
7. 下线老版本的 Pump、 Drainer 以及依赖的 Kafka 和 ZooKeeper。

10.10.7 TiDB Binlog 集群监控

成功部署TiDB Binlog 集群后，可以进入 Grafana Web 界面（默认地址: http://grafana_ip:3000，默认账号：admin，密码：admin）查看 Pump 和 Drainer 的运行状态。

10.10.7.1 监控指标

10.10.7.1.1 Pump

metric 名称	说明
Storage Size	记录磁盘的总空间大小(capacity)，以及可用磁盘空间大小(available)
Metadata	记录每个 Pump 的可删除binlog 的最大tso (gc_tso)，以及保存的binlog 的最大的commit tso (max_commit_tso)。
Write Binlog QPS by Instance	每个 Pump 接收到的写binlog 请求的 QPS

metric 名称	说明
Write Binlog Latency	记录每个 Pump 写 binlog 的延迟时间
Storage Write Binlog Size	Pump 写 binlog 数据的大小
Storage Write Binlog Latency	Pump 中的 storage 模块写 binlog 数据的延迟
Pump Storage Error By Type	Pump 遇到的 error 数量，按照 error 的类型进行统计
Query TiKV	Pump 通过 TiKV 查询事务状态的次数

10.10.7.1.2 Drainer

metric 名称	说明
Checkpoint TSO	Drainer 已经同步到下游的 binlog 的最大 TSO 对应的时间。可以通过该指标估算同步延迟时间
Pump Handle TSO	记录 Drainer 从各个 Pump 获取到的 binlog 的最大 TSO 对应的时间
Pull Binlog QPS by Pump NodeID	Drainer 从每个 Pump 获取 binlog 的 QPS
95% Binlog Reach Duration By Pump	记录 binlog 从写入 Pump 到被 Drainer 获取到这个过程的延迟时间
Error By Type	Drainer 遇到的 error 数量，按照 error 的类型进行统计
SQL Query Time	Drainer 在下游执行 SQL 的耗时
Drainer Event	各种类型 event 的数量，event 包括 ddl、 insert、 delete、 update、 flush、 savepoint
Execute Time	写入 binlog 到同步下游模块所消耗的时间
95% Binlog Size	Drainer 从各个 Pump 获取到 binlog 数据的大小
DDL Job Count	Drainer 处理的 DDL 的数量
Queue Size	Drainer 内部工作队列大小

10.10.7.2 监控报警规则

本节介绍了 TiDB Binlog 组件的报警项及相应的报警规则。根据严重级别，报警项可分为三类，按照严重程度由高到低依次为：紧急级别(Emergency)、 重要级别(Critical)、 警告级别(Warning)。

10.10.7.2.1 紧急级别报警项

紧急级别的报警通常由于服务停止或节点故障导致，此时需要马上进行人工干预操作。

`binlog_pump_storage_error_count`

- 报警规则：

  ```
  changes(binlog_pump_storage_error_count[1m])> 0
  ```

- 规则描述：

  Pump 写 binlog 到本地存储时失败。

- 处理方法：

  确认 `pump_storage_error` 监控是否存在错误，查看 Pump 日志确认原因。

10.10.7.2.2 重要级别报警项

对于重要级别的报警，需要密切关注异常指标。

`binlog_drainer_checkpoint_high_delay`

- 报警规则：

  ```
  (time()- binlog_drainer_checkpoint_tso / 1000)> 3600
  ```

- 规则描述：

  Drainer 同步落后延迟超过 1 个小时。

- 处理方法：

  - 判断从 Pump 获取数据是否太慢：
    监控 Pump handle tso 可以看每个 Pump 最近一条消息的时间，是不是有延迟特别大的 Pump，确认对应 Pump 正常运行。
  - 根据 Drainer event 和 Drainer execute latency 来判断是否下游同步太慢：
    * 如果 Drainer execute time 过大，则检查到目标库网络带宽和延迟，以及目标库状态。
    * 如果 Drainer execute time 不大，Drainer event 过小，则增加 work count 和 batch 进行重试。
  - 如果上面都不满足或者操作后没有改观，请从 PingCAP 官方或TiDB 社区获取支持。

10.10.7.2.3 警告级别报警项

警告级别的报警是对某一问题或错误的提醒。

`binlog_pump_write_binlog_rpc_duration_seconds_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(binlog_pump_rpc_duration_seconds_bucket{method="WriteBinlog"}[5
  ↪ m]))> 1
  ```

- 规则描述：

  Pump 处理TiDB 写 Binlog 请求耗时过大。

- 处理方法：
  - 确认磁盘性能压力，通过 node exported 查看 disk performance 监控。
  - 如果 disk latency 和 util 都很低，请从 PingCAP 官方或 TiDB 社区获取支持。

binlog_pump_storage_write_binlog_duration_time_bucket

- 报警规则：

```
histogram_quantile(0.9, rate(binlog_pump_storage_write_binlog_duration_time_bucket{type="
↪ batch"}[5m]))> 1
```

- 规则描述：

  Pump 写本地 binlog 到本地盘的耗时。

- 处理方法：

  确认 Pump 本地盘情况，进行修复。

binlog_pump_storage_available_size_less_than_20G

- 报警规则：

```
binlog_pump_storage_storage_size_bytes{type="available"} < 20 * 1024 * 1024 * 1024
```

- 规则描述：

  Pump 剩余可用磁盘空间不足 20 G。

- 处理方法：

  监控确认 Pump 的 gc_tso 是否正常。如果不正常，调整 Pump 的 GC 时间配置或者下线对应 Pump。

binlog_drainer_checkpoint_tso_no_change_for_1m

- 报警规则：

```
changes(binlog_drainer_checkpoint_tso[1m])< 1
```

- 规则描述：

  Drainer 的 checkpoint 在 1 分钟内没有更新。

- 处理方法：

  确认所有非下线的 Pump 是否正常运行。

binlog_drainer_execute_duration_time_more_than_10s

- 报警规则：

```
histogram_quantile(0.9, rate(binlog_drainer_execute_duration_time_bucket[1m]))> 10
```

- 规则描述：

  Drainer 同步到 TiDB 的事务耗时。如果这个值过大，会影响 Drainer 同步。

- 处理方法：
  - 查看 TiDB 集群的状态。
  - 查看 Drainer 日志或监控，如果是 DDL 操作导致了该问题，则忽略。

10.10.8 Reparo 使用文档

Reparo 是 TiDB Binlog 的一个配套工具，用于增量的恢复。使用TiDB Binlog 中的 Drainer 将 binlog 按照protobuf 格式输出到文件，通过这种方式来备份增量数据。当需要恢复增量数据时，使用Reparo 解析文件中的binlog，并将其应用到TiDB / MySQL 中。

Reparo 的安装包 reparo 位于TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

10.10.8.1 Reparo 使用

10.10.8.1.1 命令行参数说明

```
Usage of Reparo:
-L string
    日志输出信息等级设置：debug, info, warn, error, fatal（默认值：info）。
-V 打印版本信息。
-c int
    同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）。
-config string
    配置文件路径，如果指定了配置文件，Reparo 会首先读取配置文件的配置；
        ↪ 如果对应的配置在命令行参数里面也存在，Reparo
        ↪ 就会使用命令行参数的配置来覆盖配置文件里面的。
-data-dir string
    Drainer 输出的 protobuf 格式 binlog 文件的存储路径（默认值： data.drainer）。
-dest-type string
    下游服务类型。 取值为 print, mysql（默认值：print）。当值为 print 时，只做解析打印到标准输出
        ↪ ，不执行 SQL；如果为 mysql，则需要在配置文件内配置 host、port、user、password 等信息
        ↪ 。
-log-file string
    log 文件路径。
-log-rotate string
    log 文件切换频率，取值为 hour、 day。
-start-datetime string
    用于指定开始恢复的时间点，格式为 “2006-01-02 15:04:05”。如果不设置该参数则从最早的 binlog
        ↪ 文件开始恢复。
-stop-datetime string
    用于指定结束恢复的时间点，格式同上。如果不设置该参数则恢复到最后一个 binlog 文件。
-safe-mode bool
    指定是否开启安全模式，开启后可支持反复同步。
-txn-batch int
    输出到下游数据库一个事务的 SQL 语句数量（默认 20）。
```

10.10.8.1.2 配置文件说明

```
### Drainer 输出的 protobuf 格式 binlog 文件的存储路径。
```

```
data-dir = "./data.drainer"

### 日志输出信息等级设置：debug，info，warn，error，fatal（默认值：info）。
log-level = "info"

### 使用 start-datetime 和 stop-datetime 来选择恢复指定时间范围内的 binlog，格式为 “2006-01-02
    ′→ 15:04:05”。
### start-datetime = ""
### stop-datetime = ""

### start-tso、stop-tso 分别对应 start-datetime 和 stop-datetime，也是用于恢复指定时间范围内的
    ′→ binlog，用 tso 的值来设置。如果已经设置了 start-datetime 和 stop-datetime，就不需要再设置
    ′→  start-tso 和 stop-tso。
### 在从全量或者上次增量位置继续同步时，start-tso 应当指定为全量 tso + 1 或者上次增量的 stop-tso
    ′→ + 1
### start-tso = 0
### stop-tso = 0

### 下游服务类型。 取值为 print，mysql（默认值：print）。当值为 print 时，只做解析打印到标准输出
    ′→ ，不执行 SQL；如果为 mysql，则需要在 [dest-db] 中配置 host、port、user、password 等信息。
dest-type = "mysql"

### 输出到下游数据库一个事务的 SQL 语句数量（默认 20）。
txn-batch = 20

### 同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）。
worker-count = 16

### 安全模式配置。取值为 true 或 false（默认值：false）。当值为 true 时，Reparo 会将 update
    ′→ 语句拆分为 delete + replace 语句。
safe-mode = false

### replicate-do-db 和 replicate-do-table 用于指定恢复的库和表，replicate-do-db 的优先级高于
    ′→ replicate-do-table。支持使用正则表达式来配置，需要以 '~' 开始声明使用正则表达式。
### 注：replicate-do-db 和 replicate-do-table 使用方式与 Drainer 的使用方式一致。
### replicate-do-db = ["~^b.*","s1"]
### [[replicate-do-table]]
### db-name ="test"
### tbl-name = "log"
### [[replicate-do-table]]
### db-name ="test"
### tbl-name = "~^a.*"

### 如果 dest-type 设置为 mysql，需要配置 dest-db。
[dest-db]
```

```
host = "127.0.0.1"
port = 3309
user = "root"
password = ""
```

10.10.8.1.3 启动示例

```
./reparo -config reparo.toml
```

> 注意：
>
> - data-dir 用于指定 Drainer 输出的binlog 文件目录。
> - start-datatime 和 start-tso 效果一样，只是时间格式上的区别，用于指定开始恢复的时间点；如果不指定，则默认在第一个binlog 文件开始恢复。
> - stop-datetime 和 stop-tso 效果一样，只是时间格式上的区别，用于指定结束恢复的时间点；如果不指定，则恢复到最后一个binlog 文件的结尾。
> - dest-type 指定目标类型，取值为 `mysql`、`print`。当值为 `mysql` 时，可以恢复到 MySQL/TiDB 等使用或兼容 MySQL 协议的数据库，需要在配置下面的 [dest-db] 填写数据库信息；当取值为 `print` 的时候，只是打印binlog 信息，通常用于debug，以及查看binlog 的内容，此时不需要填写 `[dest-db]`。
> - replicate-do-db 用于指定恢复的库，不指定的话，则全部都恢复。
> - replicate-do-table 用于指定要恢复的表，不指定的话，则全部都恢复。

10.10.9 binlogctl 工具

Binlog Control（以下简称 binlogctl）是 TiDB Binlog 的命令行工具，用于管理TiDB Binlog 集群。

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

使用binlogctl 的场景：

- 同步出现故障/检查运行情况，需要查看 Pump/Drainer 的状态
- 维护集群，需要暂停/下线 Pump/Drainer
- Pump/Drainer 异常退出，状态没有更新，或者状态不符合预期，对业务造成影响

10.10.9.1 binlogctl 下载

> **注意：**
> 建议使用的 Control 工具版本与集群版本保持一致。

binlogctl 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。

10.10.9.2 binlogctl 使用说明

命令行参数：

```
Usage of binlogctl:
-V
输出 binlogctl 的版本信息
-cmd string
    命令模式，包括 "generate_meta"（已废弃）, "pumps", "drainers", "update-pump" ,"update-drainer
        ↪ ", "pause-pump", "pause-drainer", "offline-pump", "offline-drainer"
-data-dir string
    保存 Drainer 的 checkpoint 的文件的路径（默认 "binlog_position"）（已废弃）
-node-id string
    Pump/Drainer 的 ID
-pd-urls string
    PD 的地址，如果有多个，则用"," 连接（默认 "http://127.0.0.1:2379"）
-ssl-ca string
    SSL CAs 文件的路径
-ssl-cert string
        PEM 格式的 X509 认证文件的路径
-ssl-key string
        PEM 格式的 X509 key 文件的路径
-time-zone string
    如果设置时区，在 "generate_meta" 模式下会打印出获取到的 tso 对应的时间。例如 "Asia/Shanghai"
        ↪ 为 CST 时区，"Local" 为本地时区
-show-offline-nodes
    在用 `-cmd pumps` 或 `-cmd drainers` 命令时使用，这两个命令默认不显示 offline 的节点，
命令示例:↪ 仅当明确指定 `-show-offline-nodes` 时会显示
```

- 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。

10.10.10 Binlog Consumer Client 用户文档

目前 Drainer 提供了多种输出方式，包括 MySQL、TiDB、file 等。但是用户往往有一些自定义的需求，比如输出到 Elasticsearch、Hive 等，这些需求 Drainer 现在还没有实现，因此 Drainer 增加了输出到 Kafka 的功能，将binlog 数据解析后按一定的格式再输出到 Kafka 中，用户编写代码从 Kafka 中读出数据再进行处理。

10.10.10.1 配置 Kafka Drainer

修改 Drainer 的配置文件，设置输出为 Kafka，相关配置如下：

```
[syncer]
db-type = "kafka"

[syncer.to]
### Kafka 地址
kafka-addrs = "127.0.0.1:9092"
### Kafka 版本号
kafka-version = "2.4.0"
```

10.10.10.2 自定义开发

10.10.10.2.1 数据格式

首先需要了解 Drainer 写入到 Kafka 中的数据格式：

```
// Column 保存列的数据，针对数据的类型，保存在对应的变量中
message Column {
  // 数据是否为 null
  optional bool is_null = 1 [ default = false ];
  // 保存 int 类型的数据
  optional int64 int64_value = 2;
  // 保存 uint、enum, set 类型的数据
  optional uint64 uint64_value = 3;
  // 保存 float、double 类型的数据
  optional double double_value = 4;
  // 保存 bit、blob、binary、json 类型的数据
  optional bytes bytes_value = 5;
  // 保存 date、time、decimal、text、char 类型的数据
  optional string string_value = 6;
}

// ColumnInfo 保存列的信息，包括列名、类型、是否为主键
message ColumnInfo {
  optional string name = 1 [ (gogoproto.nullable) = false ];
  // MySQL 中小写的列字段类型
  //  https://dev.mysql.com/doc/refman/8.0/en/data-types.html
```

```
  // numeric 类型：int bigint smallint tinyint float double decimal bit
  // string 类型：text longtext mediumtext char tinytext varchar
  // blob longblob mediumblob binary tinyblob varbinary
  // enum set
  // json 类型：json
  optional string mysql_type = 2 [ (gogoproto.nullable) = false ];
  optional bool is_primary_key = 3 [ (gogoproto.nullable) = false ];
}

// Row 保存一行的具体数据
message Row { repeated Column columns = 1; }

// MutationType 表示 DML 的类型
enum MutationType
  { Insert = 0;
  Update = 1;
  Delete = 2;
}

// Table 包含一个表的数据变更
message Table {
  optional string schema_name = 1;
  optional string table_name = 2;
  repeated ColumnInfo column_info = 3;
  repeated TableMutation mutations = 4;
}

// TableMutation 保存一行数据的变更
message TableMutation {
  required MutationType type = 1;
  // 修改后的数据
  required Row row = 2;
  // 修改前的数据，只对 Update MutationType 有效
  optional Row change_row = 3;
}

// DMLData 保存一个事务所有的 DML 造成的数据变更
message DMLData {
  // `tables` 包含事务中所有表的数据变更
  repeated Table tables = 1;
}

// DDLData 保存 DDL 的信息
message DDLData {
  // 当前使用的数据库
```

```
  optional string schema_name = 1;
  // 相关表
  optional string table_name = 2;
  // `ddl_query` 是原始的 DDL 语句 query
  optional bytes ddl_query = 3;
}

// BinlogType 为 Binlog 的类型，分为 DML 和 DDL
enum BinlogType {
  DML = 0; //  Has `dml_data`
  DDL = 1; //  Has `ddl_query`
}

// Binlog 保存一个事务所有的变更，Kafka 中保存的数据为该结构数据序列化后的结果
message Binlog {
  optional BinlogType type = 1 [ (gogoproto.nullable) = false ];
  optional int64 commit_ts = 2 [ (gogoproto.nullable) = false ];
  optional DMLData dml_data = 3;
  optional DDLData ddl_data = 4;
}
```

查看数据格式的具体定义，参见 secondary_binlog.proto。

10.10.10.2.2 Driver

TiDB-Tools 项目提供了用于读取 Kafka 中binlog 数据的 Driver，具有如下功能：

- 读取 Kafka 的数据
- 根据commit ts 查找binlog 在 kafka 中的储存位置

使用该 Driver 时，用户需要配置如下信息：

- KafkaAddr：Kafka 集群的地址
- CommitTS：从哪个 commit ts 开始读取binlog
- Offset：从 Kafka 哪个 offset 开始读取，如果设置了 CommitTS 就不用配置该参数
- ClusterID：TiDB 集群的cluster ID
- Topic: Kafka Topic 名称，如果Topic 名称为空，将会使用drainer <ClusterID>_obinlog 中的默认名称

用户以包的形式引用 Driver 的代码即可使用，可以参考 Driver 中提供的示例代码来学习如何使用 Driver 以及 binlog 数据的解析，目前提供了两个例子：

- 使用该 Driver 将数据同步到 MySQL，该示例包含将 binlog 转化为 SQL 的具体方法
- 使用该 Driver 将数据打印出来

Driver 项目地址：Binlog Slave Driver。

注意：

- 示例代码仅仅用于示范如何使用 Driver，如果需要用于生产环境需要优化代码。
- 目前仅提供了 golang 版本的 Driver 以及示例代码。如果需要使用其他语言，用户需要根据 binlog 的 proto 文件生成相应语言的代码文件，并自行开发程序读取 Kafka 中的binlog 数据、解析数据、输出到下游。也欢迎用户优化 example 代码，以及提交其他语言的示例代码到TiDB-Tools。

10.10.11 TiDB Binlog Relay Log

Drainer 同步binlog 时会拆分上游的事务，并将拆分的事务并发同步到下游。在极端情况下，上游集群不可用并且 Drainer 异常退出后，下游集群（MySQL 或 TiDB）可能处于数据不一致的中间状态。在此场景下，Drainer 借助relay log 可以确保将下游集群同步到一个一致的状态。

10.10.11.1 Drainer 同步时的一致性状态

下游集群达到一致的状态是指：下游集群的数据等同于上游设置了 `tidb_snapshot = ts` 的快照。

checkpoint 状态一致性是指：Drainer checkpoint 通过 `consistent` 保存了同步的一致性状态。Drainer 运行时 `consistent` 为 `false`，Drainer 正常退出后 `consistent` 更新为 `true`。

查询下游checkpoint 表的示例如下：

```
mysql> select  * from tidb_binlog.checkpoint;
+---------------------+------------------------------------------------------------------+
| clusterID           | checkPoint                                                       |
+---------------------+------------------------------------------------------------------+
| 6791641053252586769 | {"consistent":false,"commitTS":414529105591271429,"ts-map":{}} |
+---------------------+------------------------------------------------------------------+
```

10.10.11.2 工作原理

Drainer 开启relay log 后会先将binlog event 写到磁盘上，然后再同步给下游集群。如果上游集群不可用，Drainer 可以通过读取relay log 把下游集群恢复到一个一致的状态。

注意：

若同时丢失relay log 数据，该方法将不可用，不过这是概率极小的事件。此外可以使用 NFS 等网络文件系统来保证relay log 的数据安全。

10.10.11.2.1 Drainer 从 relay log 消费 binlog 的触发场景

如果 Drainer 启动时无法连接到上游集群的 PD，并且探测到 checkpoint 的 `consistent = false`，此时会尝试读取relay log，并将下游集群恢复到一致的状态。然后 Drainer 进程将checkpoint 的 `consistent` 设置为 `true` 后主动退出。

10.10.11.2.2 Relay log 的清理（GC）机制

Drainer 在将数据同步到下游之前，会先将数据写入到 relay log 文件中。当一个 relay log 文件大小达到 10MB（默认）并且当前事务的binlog 数据被写入完成后，Drainer 就会开始将数据写入到下一个relay log 文件中。当 Drainer 将数据成功同步到下游后，就会自动清除当前正在写入的relay log 文件以外其他已完成同步的relay log 文件。

10.10.11.3 配置

在 Drainer 中添加以下配置来开启relay log 功能：

```
[syncer.relay]
### 保存 relay log 的目录，空值表示不开启。
### 只有下游是 TiDB 或 MySQL 时该配置才有生效。
log-dir = "/dir/to/save/log"
### 单个 relay log 文件大小限制（单位：字节）。
### 超出该值后会将 binlog 数据写入到下一个 relay log 文件。
max-file-size = 10485760
```

10.10.12 集群间双向同步

> **警告：**
>
> 目前双向同步属于实验特性，尚未经过完备的测试，不建议在生产环境中使用该功能。

本文档介绍如何将一个TiDB 集群的数据双向同步到另一个TiDB 集群、双向同步的实现原理、如何开启双向同步、以及如何同步 DDL 操作。

10.10.12.1 使用场景

当用户需要在两个TiDB 集群之间双向同步数据时，可使用TiDB Binlog 进行操作。例如要将集群A 的数据同步到集群B，而且要将集群B 的数据同步到集群A。

> **注意：**
>
> 集群间双向同步的前提条件是，写入两个集群的数据必须保证无冲突，即在两个集群中，不会同时修改同一张表的同一主键和具有唯一索引的行。

使用场景示例图如下：

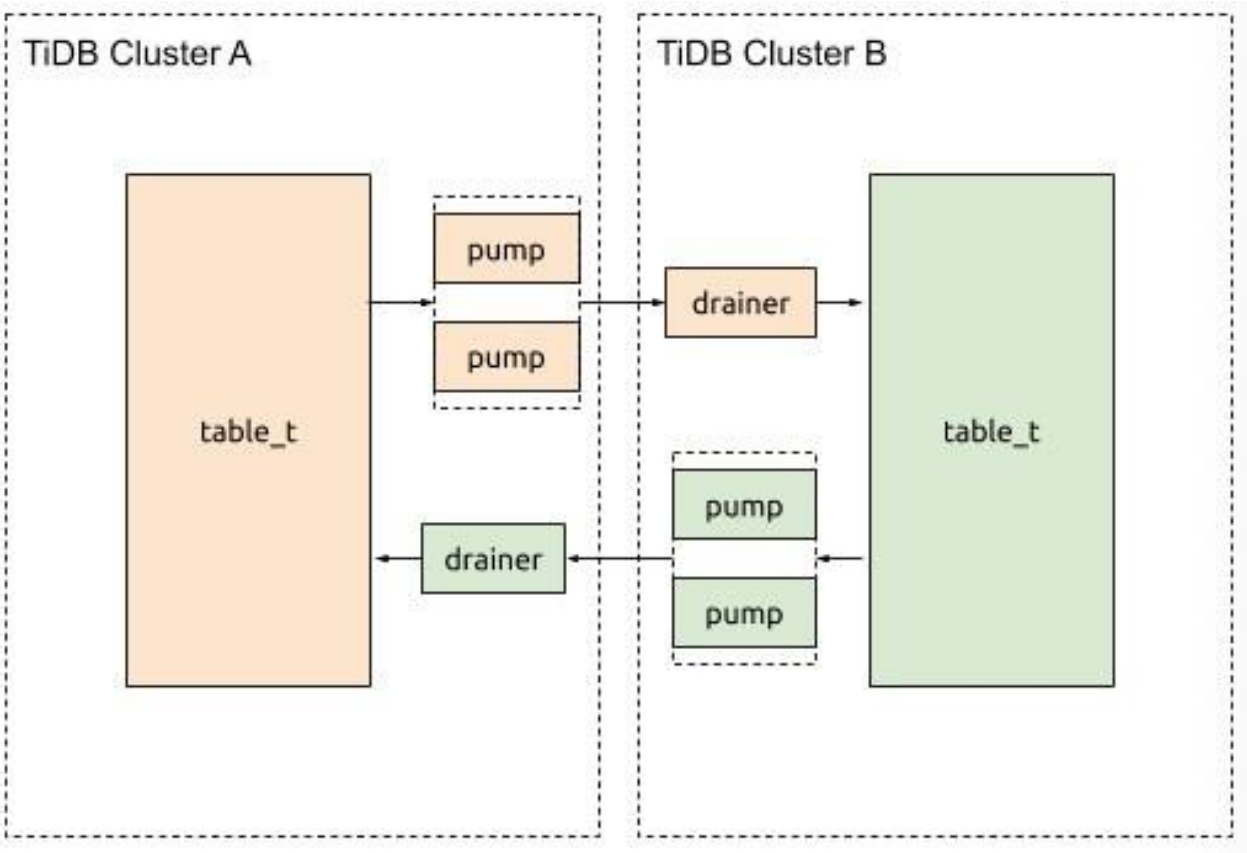


图 228: 使用场景示例图

10.10.12.2 实现原理

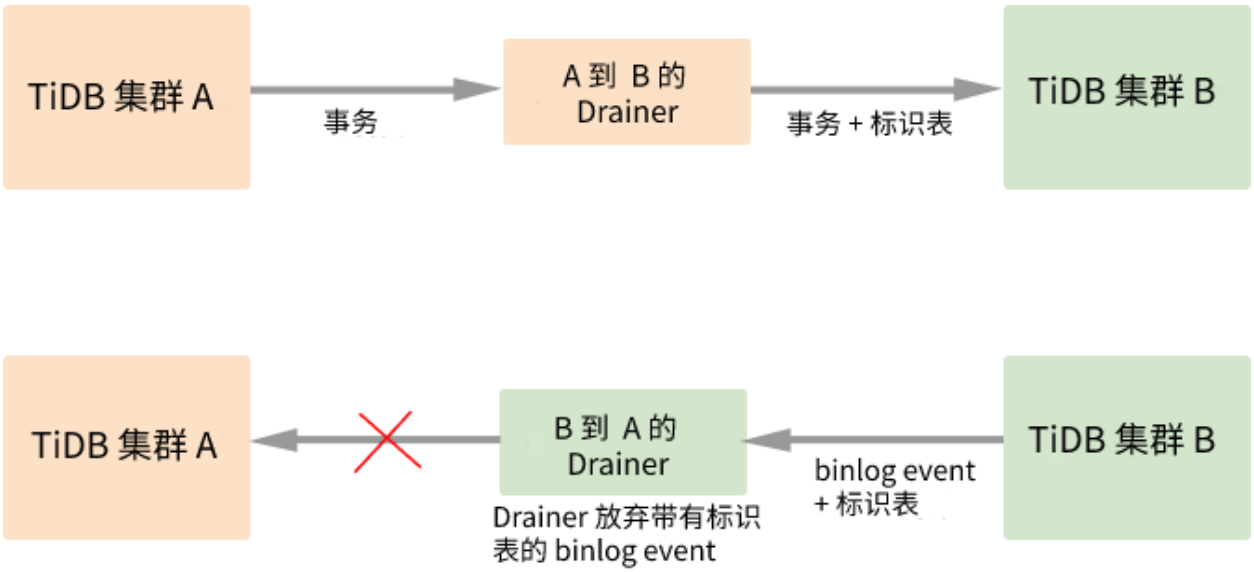


图 229: 原理示例图

在 A 和 B 两个集群间开启双向同步，则写入集群A 的数据会同步到集群B 中，然后这部分数据又会继续同步到集群A，这样就会出现无限循环同步的情况。如上图所示，在同步数据的过程中 Drainer 对 binlog 加上标记，通过过滤掉有标记的binlog 来避免循环同步。详细的实现流程如下：

1. 为两个集群分别启动TiDB Binlog 同步程序。
2. 待同步的事务经过A 的 Drainer 时，Drainer 为事务加入_drainer_repl_mark 标识表，并在表中写入本次 DML event 更新，将事务同步至集群B。
3. 集群 B 向集群 A 返回带有 _drainer_repl_mark 标识表的 binlog event。集群 B 的 Drainer 在解析该 binlog event 时发现带有 DML event 的标识表，放弃同步该binlog event 到集群A。

将集群B 中的数据同步到集群A 的流程与以上流程相同，两个集群可以互为上下游。

> 注意：
>
> - 更新 _drainer_repl_mark 标识表时，一定要有数据改动才会产生binlog。
> - DDL 操作没有事务概念，因此采取单向同步的方案，见同步 DDL。

Drainer 与下游的每个连接可以使用一个ID 以避免冲突。channel_id 用来表示进行双向同步的一个通道。A 和 B 两个集群进行双向同步的配置应使用相同的值。

当有添加或者删除列时，要同步到下游的数据可能会出现多列或者少列的情况。Drainer 通过添加配置来允许这种情况，会忽略多了的列值或者给少了的列写入默认值。

10.10.12.3 标识表

_drainer_repl_mark 标识表的结构如下：

```
CREATE TABLE `_drainer_repl_mark` (
  `id` bigint(20) NOT NULL,
  `channel_id` bigint(20) NOT NULL DEFAULT '0',
  `val` bigint(20) DEFAULT '0',
  `channel_info` varchar(64) DEFAULT NULL,
  PRIMARY KEY (`id`,`channel_id`)
);
```

Drainer 使用如下 SQL 语句更新 _drainer_repl_mark 可保证数据改动，从而保证产生binlog：

```
update drainer_repl_mark set val = val + 1 where id = ? && channel_id = ?;
```

10.10.12.4 同步 DDL

因为 Drainer 无法为 DDL 操作加入标识表，所以采用单向同步的方式来同步 DDL 操作。

比如集群A到集群B开启了 DDL 同步，则集群B到集群A会关闭 DDL 同步。即 DDL 操作全部在A上执行。

> 注意：
>
> DDL 操作无法在两个集群上同时执行。执行 DDL 时，若同时有 DML 操作或者 DML binlog 没同步完，会可能出现 DML 同步的上下游表结构不一致的情况。

10.10.12.5 配置并开启双向同步

若要在集群A和集群B间进行双向同步，假设统一在集群A上执行 DDL。在集群 A 到集群B的同步路径上，向 Drainer 添加以下配置：

```
[syncer]
loopback-control = true
channel-id = 1 # 互相同步的两个集群配置相同的 ID。
sync-ddl = true # 需要同步 DDL 操作

[syncer.to]
### 1 表示 SyncFullColumn，2 表示 SyncPartialColumn。
### 若设为 SyncPartialColumn，Drainer 会允许下游表结构比当前要同步的数据多或者少列
### 并且去掉 SQL mode 的 STRICT_TRANS_TABLES，来允许少列的情况，并插入零值到下游。
sync-mode = 2

### 忽略 checkpoint 表。
[[syncer.ignore-table]]
db-name = "tidb_binlog"
tbl-name = "checkpoint"
```

在集群B到集群A的同步路径上，向 Drainer 添加以下配置：

```
[syncer]
loopback-control = true
channel-id = 1 # 互相同步的两个集群配置相同的 ID。
sync-ddl = false #  不需要同步 DDL 操作。

[syncer.to]
### 1 表示 SyncFullColumn，2 表示 SyncPartialColumn。
### 若设为 SyncPartialColumn，Drainer 会允许下游表结构比当前要同步的数据多或者少列
### 并且去掉 SQL mode 的 STRICT_TRANS_TABLES，来允许少列的情况，并插入零值到下游。
sync-mode = 2

### 忽略 checkpoint 表。
[[syncer.ignore-table]]
db-name = "tidb_binlog"
tbl-name = "checkpoint"
```

10.10.13 TiDB Binlog 术语表

本文档介绍TiDB Binlog 相关术语。

10.10.13.1 Binlog

在 TiDB Binlog 中，Binlog 通常指TiDB 写的二进制日志数据，也指 Drainer 写到 Kafka 或者文件的二进制日志数据，前者与后者的格式不同。此外，TiDB 的 binlog 格式与 MySQL 的 binlog 格式也不同。

10.10.13.2 Binlog event

TiDB 写的 DML Binlog 有 3 种 event，分别为：INSERT、UPDATE 和 DELETE。在 Drainer 监控面板上可以看到同步数据对应的不同 event 的个数。

10.10.13.3 Checkpoint

Checkpoint 指保存的断点信息，记录了 Drainer 同步到下游的commit-ts。 Drainer 重启时可以读取checkpoint，并从对应的commit-ts 开始同步数据到下游。

10.10.13.4 Safe mode

Safe mode 指增量复制过程中，当表结构中存在主键或唯一索引时，用于支持幂等导入 DML 的模式。

该模式将来自上游的 INSERT 改写为 REPLACE，将 UPDATE 改写为 DELETE 与 REPLACE，然后再向下游执行。在 Drainer 启动的前 5 分钟，会自动启动 Safe mode；另外也可以在配置文件中通过 safe-mode 参数手动开启，但该配置仅在下游是 MySQL 或 TiDB 时有效。

10.10.14 故障诊断

10.10.14.1 TiDB Binlog 故障诊断

本文总结了在TiDB Binlog 的使用过程中遇到问题的诊断流程，并指引用户通过监控、状态、日志等信息查找相应的解决方案。

如果你在使用TiDB Binlog 时出现了异常，请尝试以下方式排查问题：

1. 查看各个监控指标是否异常，参见TiDB Binlog 集群监控。
2. 使用binlogctl 工具查看各个 Pump、 Drainer 的状态是否有异常。
3. 查看 Pump、 Drainer 日志中是否有 ERROR、 WARN，并根据详细的日志信息初步判断问题原因。

通过以上方式定位到问题后，在FAQ 以及常见错误及修复中查找解决方案，如果没有查找到解决方案或者提供的解决方案无法解决问题，请提交issue，或从 PingCAP 官方或TiDB 社区获取支持。

10.10.14.2 TiDB Binlog 常见错误修复

本文档介绍TiDB Binlog 中常见的错误以及修复方法。

10.10.14.2.1 Drainer 同步数据到 Kafka 时报错 “kafka server: Message was too large, server rejected it to avoid allocation error”

报错原因：如果在TiDB 中执行了大事务，则生成的binlog 数据会比较大，可能超过了 Kafka 的消息大小限制。

解决方法：需要调整 Kafka 的配置参数，如下所示。

```
message.max.bytes=1073741824
replica.fetch.max.bytes=1073741824
fetch.message.max.bytes=1073741824
```

10.10.14.2.2 Pump 报错 “no space left on device”

报错原因：本地磁盘空间不足， Pump 无法正常写binlog 数据。

解决方法：需要清理磁盘空间，然后重启 Pump。

10.10.14.2.3 Pump 启动时报错 “fail to notify all living drainer”

报错原因：Pump 启动时需要通知所有 Online 状态的 Drainer，如果通知失败则会打印该错误日志。

解决方法：可以使用binlogctl 工具查看所有 Drainer 的状态是否有异常，保证 Online 状态的 Drainer 都在正常工作。如果某个 Drainer 的状态和实际运行情况不一致，则使用binlogctl 修改状态，然后再重启 Pump。

10.10.14.2.4 TiDB Binlog 同步中发现数据丢失

需要确认所有TiDB 实例均开启了TiDB Binlog，并且运行状态正常。如果集群版本大于 v3.0，可以使用 `curl {TiDB_IP}:{STATUS_PORT}/info/all` 命令确认所有TiDB 实例上的TiDB Binlog 状态。

10.10.14.2.5 当上游事务较大时，Pump 报错 `rpc error: code = ResourceExhausted desc = trying to send message larger than max (2191430008 vs. 2147483647)`

出现该错误的原因是TiDB 发送给 Pump 的 gRPC message 超过限值。可以在启动 Pump 时通过指定 `max-message-size` 来调整 Pump 可接受 gRPC message 的最大大小。

10.10.14.2.6 Drainer 输出 file 格式的增量数据，数据有什么清理机制吗？数据会被删除吗？

- 在 v3.0.x 版本的 Drainer 中，file 格式的增量数据没有任何清理机制。
- 在 v4.0.x 版本中，有基于时间的数据清理机制，详见 Drainer 的 `retention-time` 配置项。

10.10.15 TiDB Binlog 常见问题

本文介绍TiDB Binlog 使用过程中的常见问题及解决方案。

10.10.15.1 开启binog 对 TiDB 的性能有何影响？

- 对于查询无影响。
- 对于有写入或更新数据的事务有一点性能影响。延迟上，在 Prewrite 阶段要并发写一条p-binlog 成功后才可以提交事务，一般写binlog 比 KV Prewrite 快，所以不会增加延迟。可以在 Pump 的监控面板看到写 binlog 的响应时间。

10.10.15.2 TiDB Binlog 的同步延迟一般为多少？

TiDB Binlog 的同步延迟为秒级别，在非业务高峰时延迟一般为 3 秒左右。

10.10.15.3 Drainer 同步下游TiDB/MySQL 的账号需要哪些权限？

Drainer 同步账号需要有如下权限：

- Insert
- Update
- Delete
- Create
- Drop
- Alter
- Execute
- Index
- Select
- Create View

10.10.15.4 Pump 磁盘快满了怎么办？

确认 GC 正常：

- 确认 pump 监控面板 gc_tso 时间是否与配置一致。

如 gc 正常以下调整可以降低单个pump 需要的空间大小：

- 调整 pump GC 参数减少保留数据天数。
- 添加 pump 结点。

10.10.15.5 Drainer 同步中断怎么办？

使用以下binlogctl 命令查看 Pump 状态是否正常，以及是否全部非 `offline` 状态的 Pump 都在正常运行。

```
binlogctl -cmd pumps
```

查看 Drainer 监控与日志是否有对应报错，根据具体问题进行处理。

10.10.15.6 Drainer 同步下游TiDB/MySQL 慢怎么办？

特别关注以下监控项：

- 通过 Drainer 监控drainer event，可以看到 Drainer 当前每秒同步Insert/Update/Delete 事件到下游的速度。
- 通过 Drainer 监控 sql query time，可以看到 Drainer 在下游执行 SQL 的响应时间。

同步慢的可能原因与解决方案：

- 同步的数据库包含没有主键或者唯一索引的表，需要给表加上主键。
- Drainer 与下游之间延迟大，可以调大 Drainer `worker-count` 参数（跨机房同步建议将 Drainer 部署在下游）。
- 下游负载不高，可以尝试调大 Drainer `worker-count` 参数。

10.10.15.7 假如有一个 Pump crash 了会怎样？

Drainer 会因为获取不到这个 Pump 的数据没法同步数据到下游。如果这个 Pump 能恢复，Drainer 就能恢复同步。

如果 Pump 没法恢复，可采用以下方式进行处理：

1. 使用binlogctl 将该 Pump 状态修改为 `offline`（丢失这个 Pump 的数据）
2. Drainer 获取到的数据会丢失这个 Pump 上的数据，下游跟上游数据会不一致，需要重新做全量+增量同步。具体步骤如下：
    1. 停止当前 Drainer。
    2. 上游做全量备份。
    3. 清理掉下游数据，包括checkpoint 表 `tidb_binlog.checkpoint`。
    4. 使用上游的全量备份恢复下游数据。
    5. 部署 Drainer，使用 `initialCommitTs=`{从全量备份获取快照的时间戳}。

10.10.15.8 什么是checkpoint？

Checkpoint 记录了 Drainer 同步到下游的commit-ts，Drainer 重启时可以读取checkpoint 接着从对应commit-ts 同步数据到下游。Drainer 日志 `["write save point"] [ts=411222863322546177]` 表示保存对应时间戳的checkpoint。

下游类型不同，checkpoint 的保存方式也不同：

- 下游 MySQL/TiDB 保存在 `tidb_binlog.checkpoint` 表。
- 下游 kafka/file 保存在对应配置目录里的文件。

因为 kafka/file 的数据内容包含了commit-ts，所以如果checkpoint 丢失，可以消费下游最新的一条数据看写到下游数据的最新commit-ts。

Drainer 启动的时候会去读取checkpoint，如果读取不到，就会使用配置的 `initial-commit-ts` 做为初次启动开始的同步时间点。

10.10.15.9 Drainer 机器发生故障，下游数据还在，如何在新机器上重新部署 Drainer？

如果下游数据还在，只要保证能从对应checkpoint 接着同步即可。

假如checkpoint 还在，可采用以下方式进行处理：

1. 部署新的 Drainer 并启动即可（参考checkpoint 介绍，Drainer 可以读取checkpoint 接着同步）。
2. 使用binlogctl 将老的 Drainer 状态修改成 `offline`。

假如checkpoint 不在，可以如下处理：

1. 获取之前 Drainer 的 checkpoint `commit-ts`， 做为新部署 Drainer 的 `initial-commit-ts` 配置来部署新的 Drainer。
2. 使用binlogctl 将老的 Drainer 状态修改成 `offline`。

10.10.15.10 如何用全量+ binlog 备份文件来恢复一个集群？

1. 清理集群数据并使用全部备份恢复数据。
2. 使用 reparo 设置 `start-tso = {全量备份文件快照时间戳+1}`，`end-ts = 0`（或者指定时间点），恢复到备份文件最新的数据。

10.10.15.11 主从同步开启 `ignore-error` 触发critical error 后如何重新部署？

TiDB 配置开启 `ignore-error` 写 binlog 失败后触发critical error 告警，后续都不会再写binlog，所以会有binlog 数据丢失。如果要恢复同步，需要如下处理：

1. 停止当前 Drainer。
2. 重启触发critical error 的 `tidb-server` 实例重新开始写binlog（触发critical error 后不会再写binlog 到pump）。
3. 上游做全量备份。
4. 清理掉下游数据包括checkpoint 表 `tidb_binlog.checkpoint`。
5. 使用上游的全量备份恢复下游。
6. 部署 Drainer，使用 `initialCommitTs= {从全量备份获取快照的时间戳}`。

10.10.15.12 同步时出现上游数据库支持但是下游数据库执行会出错的 DDL，应该怎么办？

1. 查看drainer.log日志，查找 `exec failed` 找到 Drainer 退出前最后一条执行失败的 DDL。
2. 将 DDL 改为下游兼容支持的版本，在下游数据库中手动执行。
3. 查看drainer.log日志，查找执行失败的 DDL 语句，可以查询到该 DDL 的commit-ts。例如：

```
[2020/05/21 09:51:58.019 +08:00] [INFO] [syncer.go:398] ["add ddl item to syncer, you can
    '→ add this commit ts to `ignore-txn-commit-ts` to skip this ddl if needed"] [sql="
    '→ ALTER TABLE `test` ADD INDEX (`index1`)"] ["commit ts"=416815754209656834]。
```

4. 编辑 `drainer.toml` 配置文件，在 `ignore-txn-commit-ts` 项中添加该commit-ts，重启 Drainer。

在绝大部分情况下，TiDB 和 MySQL 的语句都是兼容的。用户需要注意的是上下游的 `sql_mode` 应当保持一致。

10.10.15.13 在什么情况下暂停和下线 Pump/Drainer？

首先需要通过以下内容来了解 Pump/Drainer 的状态定义和启动、退出的流程。

暂停主要针对临时需要停止服务的场景，例如：

- 版本升级：停止进程后使用新的binary 启动服务。
- 服务器维护：需要对服务器进行停机维护。退出进程，等维护完成后重启服务。

下线主要针对永久（或长时间）不再使用该服务的场景，例如：

- Pump 缩容：不再需要那么多 Pump 服务了，所以下线部分服务。
- 同步任务取消：不再需要将数据同步到某个下游，所以下线对应的 Drainer。
- 服务器迁移：需要将服务迁移到其他服务器。下线服务，在新的服务器上重新部署。

10.10.15.14 可以通过哪些方式暂停 Pump/Drainer？

- 直接 kill 进程。

> 注意：
> 不能使用 `kill -9` 命令，否则 Pump/Drainer 无法对信号进行处理。

- 如果 Pump/Drainer 运行在前台，则可以通过按下 Ctrl+C 来暂停。
- 使用binlogctl 的 `pause-pump` 或 `pause-drainer` 命令。

10.10.15.15 可以使用binlogctl的 update-pump/update-drainer 命令来下线 Pump/Drainer 服务吗？

不可以。使用 update-pump/update-drainer 命令会直接修改 PD 中保存的状态信息，并且不会通知 Pump/Drainer 做相应的操作。使用不当时，可能会干扰数据同步，某些情况下还可能会造成数据不一致的严重后果。例如：

- 当 Pump 正常运行或者处于暂停状态时，如果使用 update-pump 将该 Pump 设置为 offline，那么 Drainer 会放弃获取处于 offline 状态的 Pump 的 binlog 数据，导致该 Pump 最新的binlog 数据没有同步到 Drainer，造成上下游数据不一致。
- 当 Drainer 正常运行时，使用 update-drainer 命令将该 Drainer 设置为 offline。如果这时启动一个 Pump 节点，Pump 只会通知 online 状态的 Drainer，导致该 Drainer 没有及时获取到该 Pump 的binlog 数据，造成上下游数据不一致。

10.10.15.16 可以使用binlogctl的 update-pump/update-drainer 命令来暂停 Pump/Drainer 服务吗？

不可以。 update-pump/update-drainer 命令直接修改 PD 中保存的状态信息。执行这个命令并不会通知 Pump/Drainer 做相应的操作，而且使用不当会使数据同步中断，甚至造成数据丢失。

10.10.15.17 什么情况下使用binlogctl的 update-pump 命令设置 Pump 状态为 paused？

在某些异常情况下，Pump 没有正确维护自己的状态，实际上状态应该为 paused。这时可以使用 update-pump 对状态进行修正，例如：

- Pump 异常退出（可能由panic 或者误操作执行 kill -9 命令直接 kill 掉进程导致），Pump 保存在 PD 中的状态仍然为 online。如果暂时不需要重启 Pump 恢复服务，可以使用 update-pump 把该 Pump 状态设置为 paused，避免对 TiDB 写 binlog 和 Drainer 获取binlog 造成干扰。

10.10.15.18 什么情况下使用binlogctl的 update-drainer 命令设置 Drainer 状态为 paused？

在某些异常情况下，Drainer 没有正确维护自己的状态，，对数据同步造成了影响，实际上状态应该为 paused。这时可以使用 update-drainer 对状态进行修正，例如：

- Drainer 异常退出（出现panic 直接退出进程，或者误操作执行 kill -9 命令直接 kill 掉进程），Drainer 保存在 PD 中的状态仍然为 online。当 Pump 启动时无法正常通知该 Drainer（报错 notify drainer ...），导致 Pump 无法正常运行。这个时候可以使用 update-drainer 将 Drainer 状态更新为 paused，再启动 Pump。

10.10.15.19 可以通过哪些方式下线 Pump/Drainer？

目前只可以使用binlogctl的 offline-pump 和 offline-drainer 命令来下线 Pump 和 Drainer。

10.10.15.20 什么情况下使用binlogctl的 update-pump 命令设置 Pump 状态为 offline?

> 警告：
>
> 仅在可以容忍binlog 数据丢失、上下游数据不一致或者确认不再需要使用该Pump 存储的binlog 数据的情况下，才能使用 update-pump 修改 Pump 状态为 offline。

可以使用 update-pump 修改 Pump 状态为 offline 的情况有：

- 在某些情况下，Pump 异常退出进程，且无法恢复服务，同步就会中断。如果希望恢复同步且可以容忍部分binlog 数据丢失，可以使用 update-pump 命令将该 Pump 状态设置为 offline，则 Drainer 会放弃拉取该 Pump 的 binlog 然后继续同步数据。
- 有从历史任务遗留下来且不再使用的 Pump 且进程已经退出（例如测试使用的服务），之后不再需要使用该服务，使用 update-pump 将该 Pump 设置为 offline。

在其他情况下一定要使用 offline-pump 命令让 Pump 走正常的下线处理流程。

10.10.15.21 Pump 进程已经退出， 且状态为 paused， 现在不想使用这个 Pump 节点了， 能否用 binlogctl 的 update-pump 命令设置节点状态为 offline ？

Pump 以 paused 状态退出进程时，不保证所有binlog 数据被下游 Drainer 消费。所以这样做会有上下游数据不一致的风险。正确的做法是重新启动 Pump，然后使用 offline-pump 下线该 Pump。

10.10.15.22 什么情况下使用binlogctl 的 update-drainer 命令设置 Drainer 状态为 offline ？

- 有从历史任务遗留下来且不再使用的 Drainer 且进程已经退出（例如测试使用的服务），之后不再需要使用该服务，使用 update-drainer 将该 Drainer 设置为 offline。

10.10.15.23 可以使用 change pump、change drainer 等 SQL 操作来暂停或者下线 Pump/Drainer 服务吗？

目前还不支持。这种 SQL 操作会直接修改 PD 中保存的状态，在功能上等同于使用binlogctl 的 update-pump、update-drainer 命令。如果需要暂停或者下线，仍然要使用binlogctl。

10.10.15.24 TiDB 写入binlog 失败导致TiDB 卡住，日志中出现 listener stopped, waiting for manual stop

在TiDB v3.0.12 以及之前，binlog 写入失败会导致TiDB 报 fatal error。但是 TiDB 不会自动退出只是停止服务，看起来像服务卡住。TiDB 日志中可看到 listener stopped, waiting for manual stop。

遇到该问题需要根据具体情况判断是什么原因导致binlog 写入失败。如果是binlog 写入下游缓慢导致的，可以考虑扩容 Pump 或增加写binlog 的超时时间。

TiDB 在 v3.0.13 版本中已优化此逻辑，写入binlog 失败将使事务执行失败返回报错，而不会导致TiDB 卡住。

10.10.15.25 TiDB 向 Pump 写入了重复的binlog ？

TiDB 在写入binlog 失败或者超时的情况下，会重试将binlog 写入到下一个可用的 Pump 节点直到写入成功。所以如果写入到某个 Pump 节点较慢，导致TiDB 超时（默认 15s），此时 TiDB 判定写入失败并尝试写入下一个 Pump 节点。如果超时的 Pump 节点实际也写入成功，则会出现同一条binlog 被写入到多个 Pump 节点。Drainer 在处理binlog 的时候，会自动去重TSO 相同的binlog，所以这种重复的写入对下游无感知，不会对同步逻辑产生影响。

10.10.15.26 在使用全量+ 增量方式恢复的过程中，Reparo 中断了，可以使用日志里面最后一个TSO 恢复同步吗？

可以。Reparo 不会在启动时自动开启 safe-mode 模式，需要手动操作：

1. Reparo 中断后，记录日志中最后一个TSO，记为 checkpoint-tso。
2. 修改Reparo 配置文件，将配置项 start-tso 设为 checkpoint-tso + 1，将 stop-tso 设为 checkpoint-tso + 80,000,000,000（大概是 checkpoint-tso 延后 5 分钟），将 safe-mode 设置为 true。启动 Reparo，Reparo 会将数据同步到 stop-tso 后自动停止。
3. Reparo 自动停止后，将 start-tso 设置为 checkpoint tso + 80,000,000,001，将 stop-tso 设置为 0，将 safe-mode 设为 false。启动 Reparo 继续同步。

10.11 PingCAP Clinic 诊断服务

10.11.1 PingCAP Clinic 诊断服务简介

PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）是 PingCAP 为 TiDB 集群提供的诊断服务，支持对使用TiUP 或 TiDB Operator 部署的集群进行远程定位集群问题和本地快速检查集群状态，用于从全生命周期确保TiDB 集群稳定运行、预测可出现的集群问题、降低问题出现概率、快速定位并修复问题。

PingCAP Clinic 服务提供以下两个组件进行集群诊断：

- Diag 诊断客户端：部署在集群侧的开源工具，用于采集集群的诊断数据(collect)、上传诊断数据到 Clinic Server、对集群进行本地快速健康检查 (check)。如需了解 Diag 工具可采集的详细的数据列表，请参阅PingCAP Clinic 数据采集说明。

> 注意：
>
> Diag 诊断客户端支持TiDB v4.0 及以上的集群，不支持使用TiDB Ansible 部署的集群。

- Clinic Server：部署在云端的云服务。Clinic Server 提供 SaaS 模式的诊断服务，不仅能接收上传到该组件的诊断数据，也可以提供在线诊断环境，用于存储、查看和诊断已上传的诊断数据，并提供集群诊断报告。根据数据存储的位置不同，Clinic Server 分为以下两个独立的服务：
  - Clinic Server 中国区，数据存储在AWS 中国区（北京）。
  - Clinic Server 美国区，数据存储在AWS 美国区。

10.11.1.1 使用场景

- 远程定位集群问题

  当集群出现无法快速修复的问题时，请从 PingCAP 官方或TiDB 社区获取支持。当申请远程协助时，你需要先保存问题现场的各种诊断数据，然后将其转发给相关技术人员。此时，你可以使用 Diag 诊断客户端，对诊断数据进行一键采集，快速收集完整的诊断数据，替代复杂的手动数据采集操作。随后，你可以将其诊断数据上传到 Clinic Server，供 PingCAP 技术人员查看。Clinic Server 为诊断数据提供了安全的存储，并支持在线诊断，提升了技术人员进行问题定位的效率。

- 快速检查集群状态

  即使集群可以正常运行，也需要定期检查集群是否有潜在的稳定性风险。PingCAP Clinic 提供的本地和 Server 端的快速诊断功能，用于检查集群潜在的健康风险。

10.11.1.2 工作原理

本章节主要介绍 Diag 诊断客户端（以下简称为 Diag）采集集群诊断数据的工作原理。

首先，Diag 需要从部署工具TiUP (tiup-cluster) 或 TiDB Operator (tidb-operator) 获取集群拓扑信息，然后通过不同的数据采集方式来采集不同类型的诊断数据，具体采集方式如下：

- 通过 SCP 传输服务器文件

  对于使用TiUP 部署的集群，Diag 可通过 SCP (Secure copy protocol) 直接从目标组件的节点采集日志文件和配置文件。

- 通过 SSH 远程执行命令采集数据

  对于TiUP 部署的集群，Diag 可以通过 SSH (Secure Shell) 连接到目标组件系统，并可执行Insight 等命令获取系统信息，包括内核日志、内核参数、系统和硬件的基础信息等。

- 通过 HTTP 调用采集数据

  - 通过调用TiDB 组件的 HTTP 接口，Diag 可获取TiDB、TiKV、PD 等组件的实时配置采样信息与实时性能采样信息。
  - 通过调用 Prometheus 的 HTTP 接口，Diag 可获取报警信息和metrics 监控数据。

- 通过 SQL 语句查询数据库参数

  通过 SQL 语句，Diag 可以查询TiDB 数据库的系统参数等信息。对于这种方式，你需要在采集数据时额外提供访问TiDB 数据库的用户名和密码。

10.11.1.3 Clinic Server 使用限制

> 注意：
>
> - Clinic Server 诊断服务在 2022 年 7 月 15 日至 2023 年 7 月 14 日期间提供免费服务。后续如需收取相关费用，PingCAP Clinic 运营团队将在 2023 年 7 月 14 日前通过邮件通知用户。
> - 如果需要调整使用限制，可以联系 PingCAP 技术支持。

诊断服务类型	使用限制
每个组织最多可以创建的集群数量	10 个
诊断数据存储容量	50 GB/集群
诊断数据最长存储时间	180 天
数据包最大大小	3 GB
诊断数据重建保存时间	最长 3 天

10.11.1.4 探索更多

- 在 TiUP 部署环境使用 PingCAP Clinic
  - 快速上手 PingCAP Clinic
  - 使用 PingCAP Clinic 诊断集群
  - 使用 PingCAP Clinic 生成诊断报告
  - PingCAP Clinic 数据采集说明
- 在 TiDB Operator 部署环境使用 PingCAP Clinic
  - 使用 PingCAP Clinic
  - PingCAP Clinic 数据采集说明

10.11.2 PingCAP Clinic 快速上手指南

本指南介绍如何使用 PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）快速采集、上传、查看集群诊断数据。

PingCAP Clinic 由 Diag 诊断客户端（以下简称为 Diag）和 Clinic Server 云服务平台（以下简称为 Clinic Server）组成。关于 Diag 和 Clinic Server 的详细介绍，请参考PingCAP Clinic 诊断服务简介。

10.11.2.1 使用场景

- 当集群出现问题，需要远程咨询 PingCAP 技术支持时，为了提高定位和解决问题的效率，你可以使用 Diag 采集诊断数据，上传数据到 Clinic Server 并获取数据链接，然后将数据链接提供给技术支持人员。
- 在集群正常运行时，需要检查集群的运行状态，你可以使用 Diag 采集诊断数据，上传数据到 Clinic Server 并查看 Health Report 的结果。

> 注意：
>
> - 本文档提供的采集和上传数据的方式仅适用于使用TiUP 部署的集群。如需查看适用于使用 Operator 部署的集群，请参阅在 TiDB Operator 部署环境使用 PingCAP Clinic。
> - 通过 PingCAP Clinic 采集和上传的数据仅用于定位和诊断集群问题。

10.11.2.2 准备工作

在开始体验 PingCAP Clinic 功能之前，你需要先安装数据采集组件 Diag 并准备数据上传环境。

1. 在安装了TiUP 的中控机上，一键安装 Diag：

```
tiup install diag
```

2. 登录 Clinic Server。

   登录 Clinic Server 中国区，选择 Sign in with AskTUG 进入 TiDB 社区 AskTUG 的登录界面。如果你尚未注册 AskTUG 账号，可以在该界面进行注册。

   登录 Clinic Server 美国区，选择 Sign in with TiDB Account 进入 TiDB Cloud Account 的登录界面。如果你尚未注册 TiDB Cloud 账号，可以在该界面进行注册。

> 注意：
>
> Clinic Server 只是通过TiDB Cloud 账号进行 SSO 登录，并不要求用户必须使用 TiDB Cloud 服务。

3. 在 Clinic Server 中，依据页面提示创建组织(Organization)。组织是一系列TiDB 集群的集合。你可以在所创建的组织上上传诊断数据。

4. 获取用于上传数据的Access Token（以下简称为 Token）。使用 Diag 上传数据时，你需要通过Token 进行用户认证，以保证数据上传到组织后被安全地隔离。获取一个Token 后，你可以重复使用该Token，具体获取方法如下：

   点击 Cluster 页面右下角的图标，选择Get Access Token For Diag Tool，在弹出窗口中点击 + 符号获取Token 后，复制并保存Token 信息。

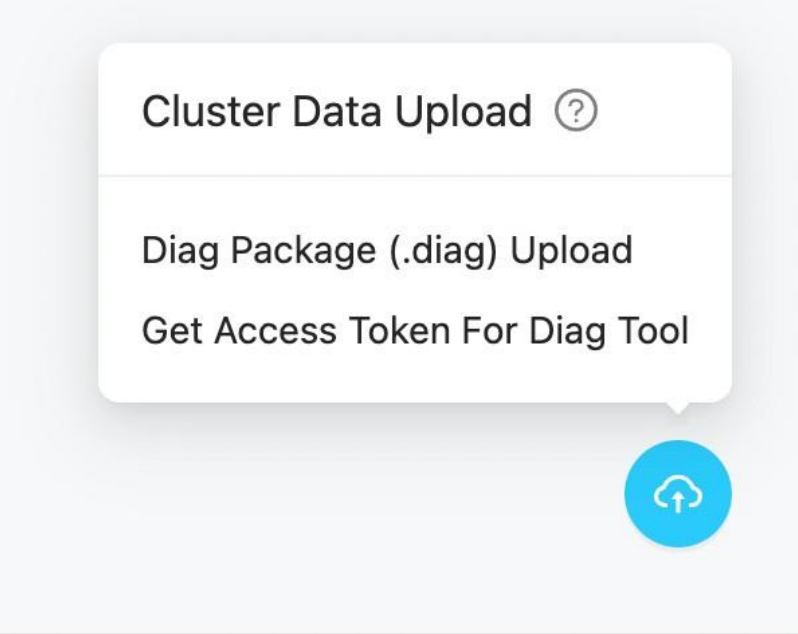


图 230: Token 示例

> 注意：
>
> - 为了确保数据的安全性，TiDB 只在创建Token 时显示Token 信息。如果丢失了Token 信息，你可以删除旧Token 后重新创建。
> - 该Token 只用于数据上传，访问数据时不需要使用Token。

5. 在 Diag 中设置Token 和 region：

   - 设置 clinic.token：

     ```
     tiup diag config clinic.token ${token-value}
     ```

   - 设置 clinic.region：

     region 决定数据打包时使用的加密证书和上传的目标 Clinic Server 地址。参考以下命令，根据你的 Clinic Server 在 Diag 中设置 clinic.region。

> 注意：
>
> - Diag v0.9.0 及以后的版本支持自行设置 region。
> - 对于 Diag v0.9.0 之前的版本，数据默认上传到中国区的 Clinic Server。
> - 如果你的 Diag 是 v0.9.0 之前的版本，你可以通过 tiup update diag 命令将其升级至最新版后设置 region。

对于 Clinic Server 中国区，参考以下命令，将 region 设置为 CN：

```
tiup diag config clinic.region CN
```

对于 Clinic Server 美国区，参考以下命令，将 region 设置为 US：

```
tiup diag config clinic.region US
```

6. 开启日志脱敏配置（可选步骤）。

   TiDB 在提供详细的日志信息时可能会打印数据库的敏感信息（例如用户数据）。如果希望本地日志及上传到 Clinic Server 的日志中不带有敏感信息，你可以开启日志脱敏配置。具体操作请参考日志脱敏。

10.11.2.3 体验步骤

1. 运行 Diag，采集诊断数据。

   例如，如需采集从当前时间的 4 小时前到 2 小时前的诊断数据，可以运行以下命令：

   ```
   tiup diag collect ${cluster-name} -f="-4h" -t="-2h"
   ```

   运行 Diag 数据采集命令后，Diag 不会立即开始采集数据，而会在输出中提供预估数据量大小和数据存储路径，并询问你是否进行数据收集。如果确认要开始采集数据，请输入Y。

   采集完成后，Diag 会提示采集数据所在的文件夹路径。

2. 将采集到的数据上传到 Clinic Server。

> 注意：
>
> 上传数据（数据包文件夹打包压缩后的文件）的大小不得超过 3 GB，否则会导致上传失败。

- 如果你的集群所在的网络可以访问互联网，你可以通过以下命令上传已采集的数据包文件夹：

```
tiup diag upload ${filepath}
```

完成上传后，Diag 会提示诊断数据的下载路径 Download URL。

> 注意：
> 使用该方式进行上传时，你需要使用 Diag v0.9.0 及以上版本。Diag 会在运行时提示版本信息。如果你早期安装过 v0.9.0 之前版本的 Diag，你可以通过 `tiup update diag` 命令将其一键升级至最新版。

- 如果你所在的集群无法访问互联网，需要先打包数据后进行上传。具体步骤，请参阅上传方式 2：打包后上传。

3. 完成数据上传后，通过上传输出结果中的 Download URL 获取诊断数据的链接。

   诊断数据默认包括集群名称、集群拓扑信息、诊断数据包中的日志内容和基于诊断数据包中的metrics信息重建的Grafana Dashboard 信息。

   你可以通过这些数据自己查找并诊断集群问题，或者，你也可以将链接发给与你对接的 PingCAP 技术支持人员，以协助远程定位集群问题。

4. 查看 Health Report 结果。

   数据上传到 Clinic Server 后，后台将自动处理数据，在大概 5 ~ 15 分钟后生成 Health Report。用户可以打开诊断数据链接，点击下方的“Health Report”入口查看报告内容。

10.11.2.4 探索更多

- 在 TiUP 部署环境使用 PingCAP Clinic
  - PingCAP Clinic 诊断服务简介
  - 使用 PingCAP Clinic 诊断集群
  - 使用 PingCAP Clinic 生成诊断报告
  - PingCAP Clinic 数据采集说明
- 在 TiDB Operator 部署环境使用 PingCAP Clinic
  - 使用 PingCAP Clinic
  - PingCAP Clinic 数据采集说明

10.11.3 使用 PingCAP Clinic 诊断集群

对于使用TiUP 部署的TiDB 集群和 DM 集群，PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）可以通过 Diag 诊断客户端（以下简称为 Diag）与 Clinic Server 云诊断平台（以下简称为 Clinic Server）实现远程定位集群问题和本地快速检查集群状态。

注意：

- 本文档仅适用于使用TiUP 部署的集群。如需查看适用于使用 Operator 部署的集群，请参阅在 TiDB Operator 部署环境使用 PingCAP Clinic。
- PingCAP Clinic 暂时不支持对使用TiDB Ansible 部署的集群进行数据采集。

10.11.3.1 使用场景

- 远程定位集群问题
  - 当集群出现问题，需要远程咨询 PingCAP 技术支持时，你可以先使用 Diag 采集诊断数据，然后将其数据上传到 Clinic Server，最后把数据链接提供给技术支持人员，协助远程定位集群问题。
  - 当集群出现问题，但无法马上进行问题分析时，你可以先使用 Diag 采集数据，并将其数据保存下来，用于自己后期进行问题分析。
- 本地快速检查集群状态

  即使集群可以正常运行，也需要定期检查集群是否有潜在的稳定性风险。PingCAP Clinic 提供的本地快速诊断功能，用于检查集群潜在的健康风险，本地诊断只覆盖配置项检查。如果需要更全面的检查，推荐上传诊断数据包到 Clinic Server，使用 Clinic Server 提供的 Health Report 对 Metrics、日志和配置项进行全面的快速检查。

10.11.3.2 准备工作

在使用 PingCAP Clinic 功能之前，你需要先安装数据采集组件 Diag 并准备数据上传环境。

1. 安装 Diag。
   - 如果你的中控机上已经安装了TiUP，可以使用以下命令一键安装 Diag：

     ```
     tiup install diag
     ```

   - 若已安装了 Diag，你可以通过以下命令，将本地的 Diag 一键升级至最新版本：

     ```
     tiup update diag
     ```

注意：

- 对于离线集群，你需要离线部署 Diag 诊断客户端。具体方法，请参照离线部署TiUP组件：方式2。
- Diag 诊断客户端仅包含在 v6.0.0 及后续版本的TiDB Server 离线镜像包中。

2. 获取并设置用于上传数据的Access Token（以下简称为 Token）。

   使用 Diag 上传采集到的数据时，你需要通过Token 进行用户认证，以保证数据上传到组织后被安全地隔离。获取一个Token 后，你可以重复使用该Token。如果你已经获取过并在 Diag 上设置过Token，可跳过此步骤。

   首先，通过以下方法获取Token：

   - 登录 Clinic Server。

     Clinic Server 中国区，数据存储在亚马逊云服务中国区。

     Clinic Server 美国区，数据存储在亚马逊云服务美国区。

   - 点击 Cluster 页面右下角的图标，选择Get Access Token For Diag Tool，在弹出窗口中点击 + 符号获取 Token 后，复制并保存Token 信息。

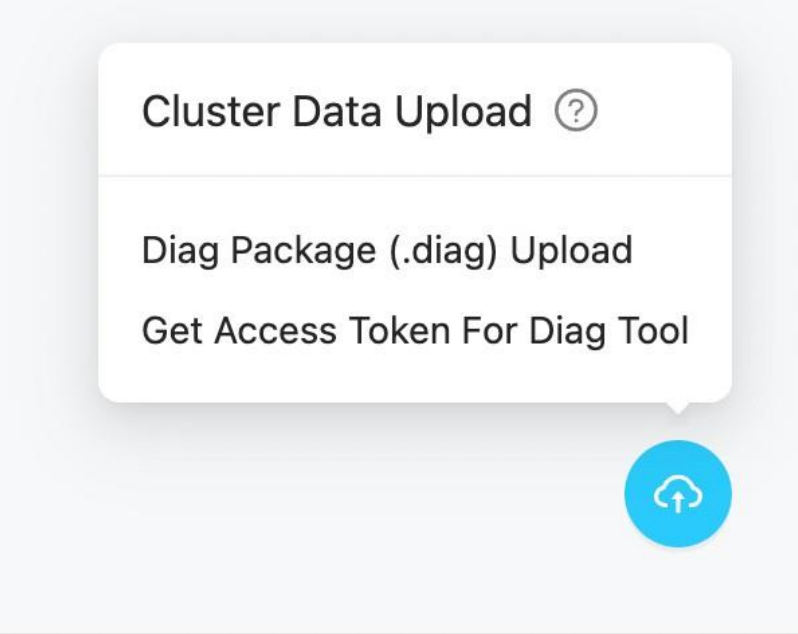


图 231: Token 示例

> 注意：
>
> - 如果你第一次访问 Clinic Server，请参考快速上手指南：准备数据上传环境的相关步骤。
> - 为了确保数据的安全性，TiDB 只在创建Token 时显示Token 信息。如果丢失了Token 信息，你可以删除旧Token 后重新创建。
> - Token 只用于上传数据。

然后，参考以下命令，在 Diag 中设置该Token：

```
tiup diag config clinic.token ${token-value}
```

3. 在 Diag 中设置 region。

`region` 决定数据打包时使用的加密证书和上传的目标 Clinic Server 地址。参考以下命令，根据你的 Clinic Server 在 Diag 中设置 `clinic.region`。

> 注意：
>
> - Diag v0.9.0 及以后的版本支持 `region` 设置。
> - 对于 Diag v0.9.0 之前的版本，数据默认上传到 Clinic Server 中国区。
> - 如果你的 Diag 是 v0.9.0 之前的版本，你可以通过 `tiup update diag` 命令将其升级至最新版本后设置 `region`。

对于 Clinic Server 中国区，参考以下命令，将 `region` 设置为 `CN`：

```
tiup diag config clinic.region CN
```

对于 Clinic Server 美国区，参考以下命令，将 `region` 设置为 `US`：

```
tiup diag config clinic.region US
```

4. 开启日志脱敏配置（可选步骤）。

   TiDB 在提供详细的日志信息时可能会打印数据库的敏感信息（例如用户数据）。如果希望本地日志及上传到 Clinic Server 的日志中不带有敏感信息，你可以开启日志脱敏配置。具体操作请参考日志脱敏。

10.11.3.3 远程定位集群问题

你可以使用 Diag 快速抓取TiDB 集群和 DM 集群的诊断数据，其中包括监控数据、配置信息等。

10.11.3.3.1 第 1 步：确定需要采集的数据

如需查看 Diag 支持采集的数据的详细列表，请参阅PingCAP Clinic 数据采集说明。

建议收集监控数据、配置信息等全量诊断数据，有助于提升后续诊断效率。具体方法，请参考采集集群的数据。

10.11.3.3.2 第 2 步：采集数据

你可以使用 Diag 采集使用TiUP 部署的TiDB 集群和 DM 集群的数据。

1. 运行 Diag 数据采集命令。

   例如，如需采集集群从当前时间的 4 小时前到 2 小时前的诊断数据，可以运行以下命令：

```
tiup diag collect ${cluster-name} -f="-4h" -t="-2h"
```

```
tiup diag collectdm ${dm-cluster-name} -f="-4h" -t="-2h"
```

   采集参数说明：

- `-f/--from`：指定采集时间的起始点。如果不指定该参数，默认起始点为当前时间的 2 小时前。如需修改时区，可使用 `-f="12:30 +0800"` 语法。如果没有在该参数中指定时区信息，如 `+0800`，则默认时区为 UTC。
- `-t/--to`：指定采集时间的结束点。如果不指定该参数，默认结束点为当前时刻。如需修改时区，可使用 `-f="12:30 +0800"` 语法。如果没有在该参数中指定时区信息，如 `+0800`，则默认时区为 UTC。

参数使用提示：

除了指定采集时间，你还可以使用Diag指定更多参数。如需查看所有参数，请使用`tiup diag collect -h` 或 `tiup diag collectdm -h` 命令。

> 注意：
>
> - Diag 默认不收集系统变量数据(db_vars)。如需收集该数据，你需要额外提供开启了系统变量可读权限的数据库用户名和密码。
> - Diag 默认不收集性能数据(perf)和 Debug 数据(debug)。
> - 如需收集全量诊断数据， 可以使用命令 `tiup diag collect <cluster-name> --include="system,monitor,log,config,db_vars,perf,debug"`。

- `-l`：传输文件时的带宽限制，单位为 Kbit/s，默认值为 100000（即 scp 的 `-l` 参数）。
- `-N/--node`：支持只收集指定节点的数据，格式为 `ip:port`。
- `--include`：只收集特定类型的数据，可选值为 `system`，`monitor`，`log`，`config`，`db_vars`。如需同时列出多种类型的数据，你可以使用逗号，来分割不同的数据类型。
- `--exclude`：不收集特定类型的数据，可选值为 `system`，`monitor`，`log`，`config`，`db_vars`。如需同时列出多种类型的数据，你可以使用逗号，来分割不同的数据类型。

运行 Diag 数据采集命令后，Diag 不会立即开始采集数据，而会在输出中提供预估数据量大小和数据存储路径，并询问你是否进行数据收集。例如：

```
Estimated size of data to collect:
Host                Size       Target
----                ----       ------
172.16.7.129:9090   43.57 MB   1775 metrics, compressed
172.16.7.87         0 B        /tidb-deploy/tidb-4000/log/tidb_stderr.log
... ...
172.16.7.179
                    325 B      /tidb-deploy/tikv-20160/conf/tikv.toml
Total               2.01 GB    (inaccurate)
These data will be stored in /home/user/diag-fNTnz5MGhr6
Do you want to continue? [y/N]: (default=N)
```

2. 如果确认要开始采集数据，请输入Y。

   采集数据需要一定的时间，具体所需时间与需要收集的数据量有关。例如，在测试环境中收集 1 GB 数据，大概需要 10 分钟。

   采集完成后，Diag 会提示采集数据所在的文件夹路径。例如：

```
Collected data are stored in /home/user/diag-fNTnz5MGhr6
```

10.11.3.3.3 第 3 步：本地查看数据（可选步骤）

已收集的数据会根据其数据来源存储于独立的子目录中，这些子目录以机器名和端口号来命名。每个节点的配置、日志等文件的存放位置与在真实服务器中存放的相对路径相同，其中：

- 系统和硬件的基础信息：位于 `insight.json`
- 系统 `/etc/security/limits.conf` 中的内容：位于 `limits.conf`
- 内核参数列表：位于 `sysctl.conf`
- 内核日志：位于 `dmesg.log`
- 采集时的网络连接情况：位于 `ss.txt`
- 配置数据：位于每节点目录下的 `config.json`
- 集群本身的元信息：位于 `meta.yaml`（此文件位于采集数据存储目录的顶层）
- 监控数据：位于 `/monitor` 文件目录。默认经过压缩，无法直接查看。如需直接查看监控指标的JSON 文件内容，可在采集时通过 `--compress-metrics=false` 参数禁用压缩。

10.11.3.3.4 第 4 步：上传数据

如需将集群诊断数据提供给 PingCAP 技术支持人员，请先将数据上传到 Clinic Server 后，再把获取到的数据访问链接发送给技术支持人员。Clinic Server 为 PingCAP Clinic 的云服务，可提供安全的诊断数据存储和共享。

根据集群的网络连接情况，你可以选择以下上传方式之一：

- 方式 1：如果集群所在的网络能访问互联网，你可以通过上传命令直接上传数据。
- 方式 2：如果集群所在的网络不能访问互联网，你需要打包后再上传数据。

> 注意：
>
> 如果在上传前没有在 Diag 中设置Token 或 `region`，Diag 会提示上传失败，并提醒你进行设置。关于Token 获取方法，请参考准备工作：第2 步。

方式 1：直接上传

如果你的集群所在的网络可以访问互联网，你可以直接通过以下命令上传在第 2 步：采集数据中收集的数据包文件夹：

```
bash tiup diag upload
```

完成上传后，Diag 会提示诊断数据的下载路径 Download URL。你可以打开 Download URL 中的链接查看数据，也可以将 Download URL 中的链接发给与你对接的 PingCAP 技术支持人员。

方式 2：打包后上传

如果你的集群所在的网络无法访问互联网，你需要先在内网打包数据后，再将其数据包发送到网络连通的设备上进行上传。具体操作方法如下：

1. 打包在第2 步：采集数据中采集的数据，并对其数据包进行压缩和加密：

```
tiup diag package ${filepath}
```

打包时，Diag 会同时对数据进行压缩和加密。在测试环境中，800 MB 数据压缩后变为 57 MB。示例输出如下：

```
Starting component `diag`: /root/.tiup/components/diag/v0.7.0/diag package diag-fNTnz5MGhr6
packaged data set saved to /home/user/diag-fNTnz5MGhr6.diag
```

完成打包后，数据包为 .diag 格式。只有上传到 Clinic Server 后，该数据包才能被解密并查看。如需直接转发已收集的数据，而不在 Clinic Server 中查看，你可以自行压缩后转发数据。

2. 使用可以访问互联网的机器上传数据压缩包。

```
tiup diag upload ${filepath}
```

完成上传后，Diag 会提示诊断数据的下载路径 Download URL。你可以打开 Download URL 中的链接，在 Clinic Server 页面进行数据查看，也可以将 Download URL 中的链接发给与你对接的 PingCAP 技术支持人员。

10.11.3.4 本地快速检查集群状态

你可以使用 Diag 对集群状态进行快速诊断。即使集群可以正常运行，也需要定期检查集群是否有潜在的稳定性风险。PingCAP Clinic 提供的本地快速诊断功能，用于检查集群潜在的健康风险，本地诊断只覆盖配置项检查。如果需要更全面的检查，推荐上传诊断数据包到 Clinic Server，使用 Clinic Server 提供的 Health Report 对 Metrics、日志和配置项进行全面的快速检查。

1. 采集配置数据：

```
tiup diag collect ${cluster-name} --include="config"
```

配置文件数据较小，采集后会默认存放至当前路径下。在测试环境中，对于一个 18 个节点的集群，配置文件数据量小于 10 KB。

2. 诊断配置数据：

```
tiup diag check ${subdir-in-output-data}
```

其中，${subdir-in-output-data} 为采集数据的存放路径，其路径中存放 meta.yaml 文件。

3. 查看诊断结果：

诊断结果会在命令行中返回，示例如下：

```
Starting component `diag`: /root/.tiup/components/diag/v0.7.0/diag check diag-fNTnz5MGhr6

# 诊断结果
lili 2022-01-24T09:33:57+08:00

## 1. 诊断集群名称等基础信息
- Cluster ID: 7047403704292855808
- Cluster Name: lili
- Cluster Version: v5.3.0
```

```
## 2. 诊断数据来源信息
- Sample ID: fNTnz5MGhr6
- Sampling Date: 2022-01-24T09:33:57+08:00
- Sample Content:: [system monitor log config]

## 3. 诊断结果信息，包括发现的可能的配置问题
In this inspection, 22 rules were executed.

The results of **1** rules were abnormal and needed to be further discussed with support
    ↪ team.

The following is the details of the abnormalities.

### 诊断结果摘要
The configuration rules are all derived from PingCAP's OnCall Service.

If the results of the configuration rules are found to be abnormal, they may cause the
    ↪ cluster to fail.

There were **1** abnormal results.

#### 诊断结果文档的保存路径
Rule Name: tidb-max-days
- RuleID: 100
- Variation:  TidbConfig.log.file.max-days
- For more information, please visit: https://s.tidb.io/msmo6awg
- Check Result:


  TidbConfig_172.16.7.87:4000    TidbConfig.log.file.max-days:0    warning
  TidbConfig_172.16.7.86:4000    TidbConfig.log.file.max-days:0    warning
  TidbConfig_172.16.7.179:4000    TidbConfig.log.file.max-days:0    warning

Result report and record are saved at diag-fNTnz5MGhr6/report-220125153215
```

在上述示例诊断结果信息的最后一部分（即“#### 诊断结果文档的保存路径”）中，对于被发现的每一条潜在的配置问题，Diag 都会提供对应的知识库链接，以便查看详细的配置建议。在上面示例中，相关链接为 `https://s.tidb.io/msmo6awg`。

10.11.3.5 常见问题

1. 如果数据上传失败了，可以重新上传吗？

   可以。数据上传支持断点上传，如果上传失败了，可以直接再次上传。

2. 数据上传后，无法打开返回的数据访问链接，怎么办？

   你可以先尝试登录 Clinic Server。如果登录后依然无法打开链接，请确认你是否拥有访问该数据的权限。如果没有权限，你需要联系数据所有人给你添加权限后，重新登录 Clinic Server 并访问数据链接。

3. 上传到 Clinic Server 的数据后会保存多久？

   最长 180 天，用户可以随时通过 Clinic Server 页面删除自己上传的集群诊断数据。

10.11.4 使用 PingCAP Clinic 生成诊断报告

PingCAP Clinic Server 云诊断平台（以下简称为 Clinic Server）可以基于上传的诊断数据生成诊断报告，包括 Benchmark Report 和 Comparison Report，用于在特定场景下检查集群健康风险，并协助定位集群问题。本文介绍Benchmark Report 和Comparison Report 报告的使用场景和使用方法。

> **注意：**
>
> 使用 PingCAP Clinic 生成的报告是基于诊断数据中的 Metrics 数据和日志数据。要获取完整的诊断建议，你需要完整收集集群诊断数据。

10.11.4.1 使用场景

- 压力测试场景

  在性能测试或新业务压力接入测试阶段，需要对系统性能进行评估，并通过调整上层业务请求和下层数据库配制参数来提升整体性能。通过Benchmark Report，可以检查当前系统性能状态，发现风险问题，提出业务层和集群配置的优化建议。

- 性能对比检查场景

  很多场景需要对比集群各种关键指标，通过Comparison Report，可以快速了解机器不同时间段运行状态的差异。相对于手动对比，Comparison Report 能够提升效率和准确性。对比的内容包括：QPS 差异、延迟差异、性能指标差异、系统资源使用差异和关键日志差异等。需要使用 Comparison Report 对比性能的场景有：

  - 集群健康巡检时，对比本次性能与上次巡检的差异
  - 集群修改了关键参数或配置后，对比修改前后的差异
  - 集群升级后，对比升级前后的差异
  - 集群接入新业务后，对比接入新业务前后的差异

10.11.4.2 前提条件

要生成 PingCAP Clinic 报告，需要使用 Diag 诊断客户端采集集群诊断数据，并上传到 Clinic Server。具体使用 Diag 采集诊断数据的方法，在TiUP 部署环境参见使用 PingCAP Clinic 诊断集群，TiDB Operator 部署环境参见使用 PingCAP Clinic。

10.11.4.3 使用方法

10.11.4.3.1 Benchmark Report

生成报告

- 生成默认报告

  Benchmark Report 会在数据包上传到 Clinic Server 后自动生成。由于生成报告需要先将相关诊断数据从数据包中解压提取，所以生成报告时间会在 5 分钟以上（数据包越大，所需时间越长）。

- 生成自定义报告

  默认生成的报告是基于数据包的起止时间，如果需要自定义生成报告的时间，可以按如下步骤进行自定义：

  1. 登录 Clinic Server（中国区、美国区）
  2. 打开需要生成诊断报告的集群页面
  3. 点击集群页面上的 Benchmark Report
  4. 在 Time Range 部分选择报告的开始(Start date) 和结束时间(End date)
  5. 点击 Create Report 生成自定义报告

解读报告

打开 Clinic Server 集群页，点击页面上的 Benchmark Report，在 All Reports 列表中点击View 可以查看相应的报告，报告内容示例如下：

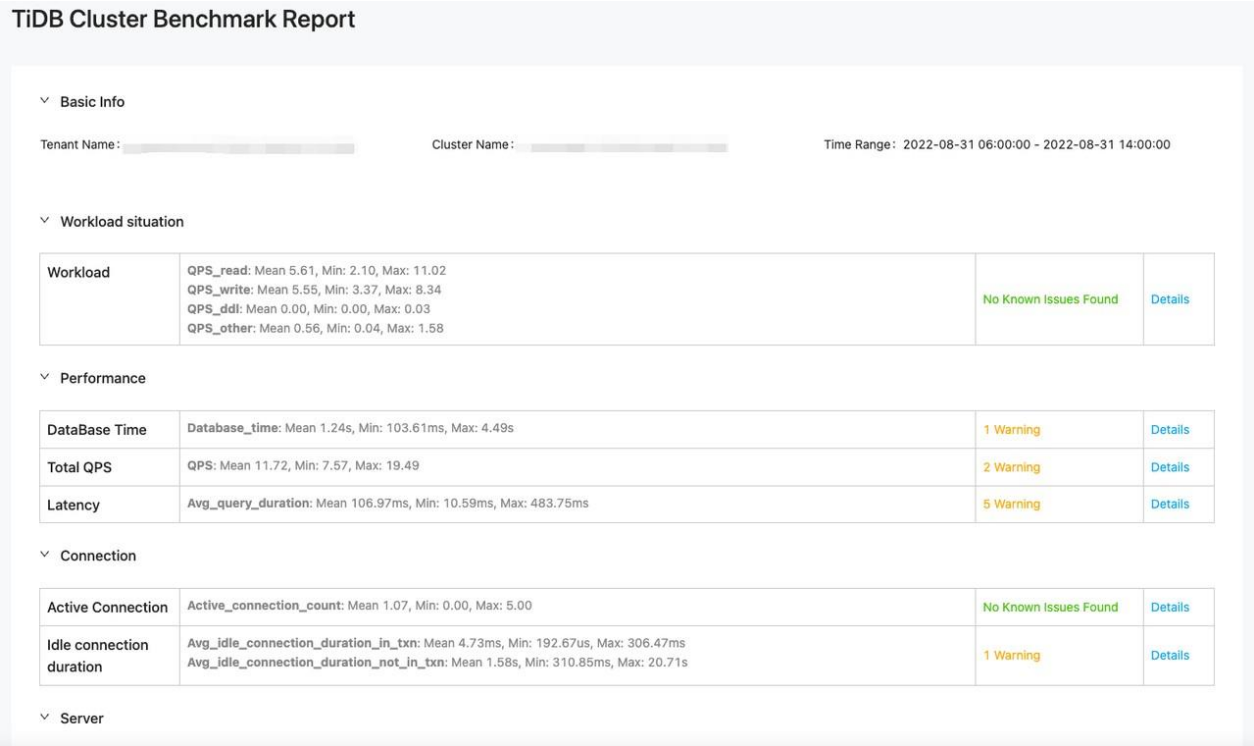


图 232: benchmark-report

> 注意：
>
> 报告功能会持续优化，实际生成的内容可能比下面的说明更丰富。

报告分为以下几个部分：

- Workload situation：业务侧性能指标结果和建议。该部分的建议主要是针对业务请求的调整。
- Performance：数据库主要性能指标分析和建议。该部分分析节点是否均衡、数据库时间 (DB Time) 中主要阶段的时延是否在合理区间，并基于业务压力和时延数据提供配置优化的建议。
- Connection：连接数据分析和建议。该部分分析各节点连接是否均衡、空闲连接是否异常。
- Server：各节点资源使用情况分析。该部分分析各节点资源使用的现状和趋势。对于资源使用不均衡，或在短时间内资源使用可能会超过阈值的情况显示Warning。

10.11.4.3.2 Comparison Report

生成报告

生成 Comparison Report 的步骤如下：

1. 登录 Clinic Server（中国区、美国区）
2. 打开需要生成诊断报告的集群页面
3. 点击集群页面上的 Comparison Report
4. 选择两个对比时间段 T1 Range 和 T2 Range
5. 点击 Create Report 生成自定义报告

生成 Comparison Report 大约需要 5 ~ 30 分钟。因为报告需要处理日志数据，所以报告生成时间与对比时间段中日志数据量相关。

解读报告

Comparison Report 会基于所选的两个时间段的 Metrics 和日志进行对比分析。报告中 Metrics 的计算方法是对两个时间段相应的指标时间序列数据取均值，然后将均值进行对比。

打开 Clinic Server 集群页，点击页面上的 Comparison Report，在 All Reports 列表中点击View 可以查看相应的报告，报告内容示例如下：

TiDB 集群诊断数据对比报告

基本信息

报告区间

T1.START_TIME	T1.END_TIME	T2.START_TIME	T2.END_TIME
2022-08-31 12:00:00	2022-08-31 16:00:00	2022-08-31 17:30:00	2022-08-31 19:40:00

主要指标对比

Metrics	T1 value	T2 value	Diff	View
QPS	12.98	15.75	21.35%	T1 T2
Connnection_count	20.61	23.15	12.31%	T1 T2
Active_connection_count	1.27	1.20	-5.37%	T1 T2
CPS	13.07	15.96	22.09%	T1 T2
Database_time	1.27s	1.20s	-5.37%	T1 T2
Avg_cluster_duration	96.76ms	74.13ms	-23.39%	T1 T2
P99_cluster_duration	474.62ms	405.31ms	-14.60%	T1 T2
TiDB_CPU_max	59.72%	60.82%	1.84%	T1 T2
TiKV_CPU_max	434.00%	812.08%	87.11%	T1 T2
TiKV_IO_MBps_max	12.18MB/s	312.46MB/s	2466.19%	T1 T2
TiKV_disk_usage_max	47.05%	47.07%	0.03%	T1 T2
Error_log_count	1142	502	-56.04%	T1 T2

图 233: comparison-report

注意：

报告功能会持续优化，实际生成的内容可能比下面的说明更丰富。

报告分为以下几个部分：

- 基本信息：列出 Comparison Report 对比的两个时间段信息。
- 主要指标对比：列出最关键的 12 个指标在前后两个时间段的对比。
- 业务指标对比：列出以下 5 个关键业务指标的对比：
  - QPS：每秒钟的请求次数。可以点击 More 对比不同请求类型的 QPS
  - Connection_count：连接数量。可以点击 More 对比不同TiDB 节点的连接数量
  - Active_connection_count：活跃连接数量
  - CPS：每秒处理的命令数。可以点击 More 对比不同命令类型的 CPS
  - Plan_cache_OPS：所有 TiDB 实例每秒使用 Plan Cache 的查询数量
- 性能指标对比：列出系统中最重要的二十多个性能指标的对比，包括以下几个部分：
  - 基础指标：数据库时间和延迟等最基础的性能指标对比
  - 基于 SQL Phase 拆分的数据库时间对比
  - 数据库时间中耗时最长的 SQL 类型对比
  - SQL_execute_time 指标中耗时最长的请求类型对比
  - Duration 指标中耗时最长的 SQL 类型对比
  - KV request 细节数据对比
  - 其他性能指标对比
- 资源指标对比：列出TiDB、TiKV、TiFlash 和 PD 节点的资源指标对比。不仅包括节点本身的对比，还会计算同类型节点资源使用的最大差值delta，基于delta 进行对比，展示节点不均衡状态的变化。
- 日志聚类对比：列出不同类型日志在前后两个时间段的数量，可以通过日志数量的变化了解系统内部任务运行情况的变化。

10.11.4.4 探索更多

- 在 TiUP 部署环境使用 PingCAP Clinic
  - 快速上手 PingCAP Clinic
  - PingCAP Clinic 诊断服务简介
  - 使用 PingCAP Clinic 诊断集群
  - PingCAP Clinic 数据采集说明
- 在 TiDB Operator 部署环境使用 PingCAP Clinic
  - 使用 PingCAP Clinic
  - PingCAP Clinic 数据采集说明

10.11.5 使用 PingCAP Clinic Diag 采集 SQL 查询计划信息

TiDB 在 v5.3.0 中引入了PLAN REPLAYER 命令，能够快速保存和恢复查询计划相关的信息，简化了排查优化器相关问题时提取排查信息的步骤。Clinic Diag 诊断客户端（下称 Diag）集成了 PLAN REPLAYER 功能，你可以使用 Diag 方便快捷地保存查询计划相关的数据。

10.11.5.1 使用说明

在排查TiDB v4.0 及以上版本的集群问题时，你可以使用 Diag 将排查现场的相关数据以ZIP 格式导出，并将导出的数据快速上传到 PingCAP Clinic 供技术支持人员查看。在使用 Diag 采集数据前，你需要提供需要采集的 SQL 语句文件。Diag 采集的数据相较于TiDB 中 `PLAN REPLAYER` 功能采集的数据多了日志信息和集群信息，具体见全量数据采集的输出结果。

> 警告：
>
> - 当前该功能为实验特性，不建议在生产环境中使用。
> - 暂时不支持采集使用TiDB Operator 部署的TiDB 集群的数据。

10.11.5.2 使用方法

本部分介绍如何使用 Diag 采集 SQL 查询计划信息。你需要先安装 Diag，再使用 Diag 采集数据。

10.11.5.2.1 安装 Diag

- 通过TiUP 安装 Diag，默认安装最新版本：

```
tiup install diag
```

- 若已安装 Diag，请确保 Diag 版本 >= 0.7.3。

  你可以通过如下命令检查 Diag 版本：

```
tiup diag --version
```

  如果 Diag 版本不满足要求，可以通过如下命令升级到最新版本：

```
tiup update diag
```

10.11.5.2.2 全量数据采集

使用如下 `diag collect` 命令进行数据采集。你需要将 `<cluster-name>` 和 `<statement-filepath>` 占位符替换为实际的值：

```
diag collect <cluster-name> --profile=tidb-plan-replayer --explain-sql=<statement-filepath>
```

> 注意：
>
> - 通过 `diag` 采集数据前，你需要通过 `--explain-sql` 指定 `sql-statement` 文件：
>   - 以上命令中 `<statement-filepath>` 是指 `sql-statement` 文件的路径。

  - 该文件包含了需要采集的 SQL 语句。
  - 如果有多条 SQL 语句需要分析，可以使用分号；分隔。
  - 因为以上 diag 命令会新建会话进行查询，所以 SQL 文件里的 SQL 语句必须显示指明所使用的数据库，如 SELECT * FROM test.t1。
- PLAN REPLAYER 不会导出表中数据。
- 采集数据时只会执行 EXPLAIN，不会真正执行查询。因此采集时对数据库性能影响较小。

sql-statement 的文件内容示例：

```
SELECT * FROM test.t1;SELECT * FROM test.t2;
```

--explain-sql 指定的 sql-statement 文件内容说明：

- 如果有多条 SQL 语句需要分析，可以使用分号；分隔。
- 因为以上 diag 命令会新建会话进行查询，所以 SQL 文件里的 SQL 语句必须显示指明所使用的数据库，如 SELECT * FROM test.t1。

输出结果

全量数据采集的输出结果包含以下集群现场信息：

序号	采集内容	调用的 Diagcollector	输出文件
1	TiDB 配置信息	config	tidb.toml
2	TiDB Session 系统变量	plan_replayer	plan_replayer.zip/variables.toml
3	TiDB 执行计划绑定信息 (SQL Binding)	sql_bind	sql_bind/global_bind.csv

序号	采集内容	调用的 Diagcollector	输出文件
4	sql-statement 中所包含的表结构	plan_replayer	replayer_ .zip /schema /<db.table>.schema.txt
5	sql-statement 中所包含表的统计信息	plan_replayer	replayer_ .zip /stats /<db.table>.json
6	EXPLAIN sql-statement 的结果	explain	explain /sql0

| 7 | 默认采集的集群信息

集群基础信息

Diag 本次采集记录

| default | cluster.json, meta.yaml, $collectid_diag_audit.log |

10.11.5.2.3 自定义数据采集

你可以自定义配置文件使 Diag 采集上述输出结果中的部分数据。下面为一个示例的配置文件 tidb-plan-replayer.toml：

```
name = "tidb-plan-replayer"
version = "0.0.1"
maintainers = [
    "pingcap"
]

### 要采集的数据类型
collectors =
    [ "config",
    "sql_bind",
    "plan_replayer"
]

### 要采集的组件
roles = [
    "tidb"
]
```

通过 --profile 参数指定配置文件的路径来进行自定义采集。你需要将 <cluster-name>、<profile-filepath> 和 <statement-filepath> 替换为实际的值：

```
diag collect <cluster-name> --profile=<profile-filepath> --explain-sql=<statement-filepath>
```

10.11.5.2.4 将结果导入到TiDB 集群

采集结果中的 plan_replayer.zip 可以通过 PLAN REPLAYER LOAD 语句直接导入到TiDB 集群中进行信息查看，具体方法可参考使用 PLAN REPLAYER 导入集群信息

10.11.6 PingCAP Clinic 数据采集说明

本文提供了 PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）在使用 TiUP 部署的TiDB 集群和 DM 集群中能够采集的诊断数据类型，并列出了各个采集项对应的采集参数。当执行 Clinic Diag 诊断客户端（以下简称为 Diag）数据采集命令时，你可以依据需要采集的数据类型，在命令中添加所需的采集参数。

通过 PingCAP Clinic 在使用TiUP 部署的集群中采集的数据仅用于诊断和分析集群问题。

Clinic Server 是部署在云端的云服务，根据数据存储的位置不同，分为以下两个独立的服务：

- Clinic Server 中国区：如果你把采集的数据上传到了 Clinic Server 中国区，这些数据将存储于 PingCAP 设立在 AWS 中国区（北京）的 S3 服务。PingCAP 对数据访问权限进行了严格的访问控制，只有经授权的内部技术人员可以访问该数据。

- Clinic Server 美国区：如果你把采集的数据上传到了 Clinic Server 美国区，这些数据将存储于 PingCAP 设立在 AWS 美国区的 S3 服务。PingCAP 对数据访问权限进行了严格的访问控制，只有经授权的内部技术人员可以访问该数据。

10.11.6.1 TiDB 集群

本节列出了 Diag 在使用TiUP 部署的TiDB 集群中能够采集的诊断数据类型。

10.11.6.1.1 TiDB 集群信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
集群基础信息，包括集群ID	`cluster.json`	每次收集默认采集
集群详细信息	`meta.yaml`	每次收集默认采集

10.11.6.1.2 TiDB 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`tidb.log`	`--include=log`
Error 日志	`tidb_stderr.log`	`--include=log`
慢日志	`tidb_slow_query.log`	`--include=log`
配置文件	`tidb.toml`	`--include=config`
实时配置	`config.json`	`--include=config`

10.11.6.1.3 TiKV 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`tikv.log`	`--include=log`
Error 日志	`tikv_stderr.log`	`--include=log`
配置文件	`tikv.toml`	`--include=config`
实时配置	`config.json`	`--include=config`

10.11.6.1.4 PD 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`pd.log`	`--include=log`
Error 日志	`pd_stderr.log`	`--include=log`
配置文件	`pd.toml`	`--include=config`

诊断数据类型	输出文件	PingCAP Clinic 采集参数
实时配置 tiup ctl ′→ :v< ′→ CLUSTER_VERSION ′→ > pd ′→ -u ′→ http ′→ ://${ ′→ pd IP ′→ }:${ ′→ PORT} ′→ ′→ store 的输出结果	config. ′→ json store. ′→ json	--include= ′→ config --include= ′→ config
tiup ctl ′→ :v< ′→ CLUSTER_VERSION ′→ > pd ′→ -u ′→ http ′→ ://${ ′→ pd IP ′→ }:${ ′→ PORT} ′→ ′→ config ′→ ′→ placement ′→ - ′→ rules ′→ show 的输出结果	placement ′→ -rule ′→ json	--include= ′→ config

10.11.6.1.5 TiFlash 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	tiflash.log	--include=log
Error 日志	tiflash_stderr.log	--include=log

诊断数据类型	输出文件	PingCAP Clinic 采集参数
配置文件	tiflash-learner.toml, tiflash-preprocessed.toml, tiflash.toml	--include=config
实时配置	config.json	--include=config

10.11.6.1.6 TiCDC 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	ticdc.log	--include=log
Error 日志	ticdc_stderr.log	--include=log
配置文件	ticdc.toml	--include=config
Debug 数据	info.txt, status.txt, changefeeds.txt, captures.txt, processors.txt	--include=debug（默认不采集）

10.11.6.1.7 Prometheus 监控数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
所有的 Metrics 数据	{metric_name}.json	--include=monitor
Alert 列表	alerts.json	--include=monitor

10.11.6.1.8 TiDB 系统变量

诊断数据类型	输出文件	PingCAP Clinic 采集参数
获取TiDB 系统变量（默认不采集，采集需要额外提供数据库账号）	mysql.tidb.csv global_variables.csv	--include=db_vars（默认不采 --include=db_vars（默认不采

10.11.6.1.9 集群节点的系统信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
内核日志	dmesg.log	--include=system

诊断数据类型	输出文件	PingCAP Clinic 采集参数
系统和硬件的基础信息	`insight.json`	`--include=system`
系统 `/etc/security/limits.conf` 中的内容	`limits.conf`	`--include=system`
内核参数列表	`sysctl.conf`	`--include=system`
socket 统计信息（即 ss 的命令结果）	`ss.txt`	`--include=system`

10.11.6.2 DM 集群

本节列出了 Diag 在使用TiUP 部署的 DM 集群中能够采集的诊断数据类型。

10.11.6.2.1 DM 集群信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
集群基础信息，包括集群ID	`cluster.json`	每次收集默认采集
集群详细信息	`meta.yaml`	每次收集默认采集

10.11.6.2.2 dm-master 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`m-master.log`	`--include=log`
Error 日志	`dm-master_stderr.log`	`--include=log`
配置文件	`dm-master.toml`	`--include=config`

10.11.6.2.3 dm-worker 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`dm-worker.log`	`--include=log`
Error 日志	`dm-worker_stderr.log`	`--include=log`
配置文件	`dm-work.toml`	`--include=config`

10.11.6.2.4 Prometheus 监控数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
所有的 Metrics 数据	`{metric_name}.json`	`--include=monitor`
Alert 列表	`alerts.json`	`--include=monitor`

10.11.6.2.5 集群节点的系统信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
内核日志	`dmesg.log`	`--include=system`
系统和硬件基础信息	`insight.json`	`--include=system`
系统 `/etc/security/limits.conf` 中的内容	`limits.conf`	`--include=system`
内核参数列表	`sysctl.conf`	`--include=system`
socket 统计信息（即 ss 的命令结果）	`ss.txt`	`--include=system`

10.12 TiSpark

10.12.1 TiSpark 用户指南

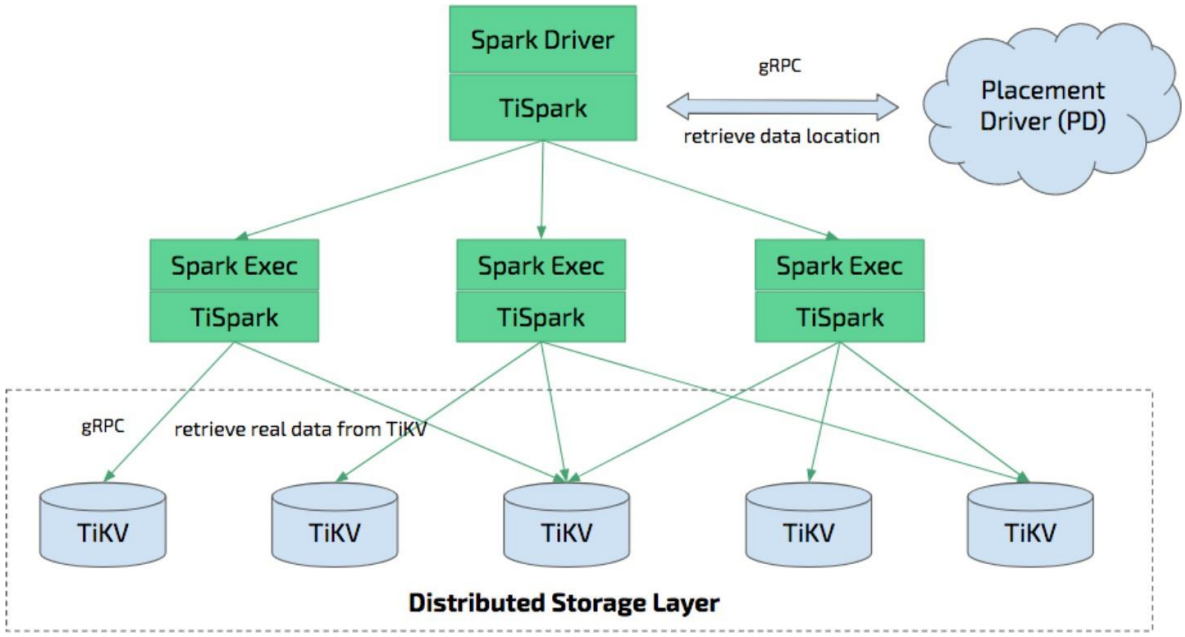


图 234: TiSpark 架构

10.12.1.1 TiSpark vs TiFlash

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合TiKV 分布式集群的优势，和TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。

TiFlash 也是一个解决 HTAP 需求的产品。TiFlash 和 TiSpark 都允许使用多个主机在 OLTP 数据上执行 OLAP 查询。TiFlash 是列式存储，这提供了更高效的分析查询。TiFlash 和TiSpark 可以同时使用。

10.12.1.2 TiSpark 是什么

TiSpark 依赖于TiKV 集群和 Placement Driver (PD)，也需要你搭建一个 Spark 集群。本文简单介绍如何部署和使用 TiSpark。本文假设你对 Spark 有基本认知。你可以参阅Apache Spark 官网了解 Spark 的相关信息。

TiSpark 深度整合了 Spark Catalyst 引擎，可以对计算进行精确的控制，使 Spark 能够高效地读取TiKV 中的数据。TiSpark 还提供索引支持，帮助实现高速点查。

TiSpark 通过将计算下推到TiKV 中提升了数据查询的效率，减少了 Spark SQL 需要处理的数据量，通过TiDB 内置的统计信息选择最优的查询计划。

TiSpark 和 TiDB 可以让用户无需创建和维护ETL，直接在同一个平台上进行事务和分析两种任务。这简化了系统架构，降低了运维成本。

用户可以在TiDB 上使用 Spark 生态圈的多种工具进行数据处理，例如：

- TiSpark：数据分析和ETL。
- TiKV：数据检索。
- 调度系统：生成报表。

除此之外，TiSpark 还提供了分布式写入TiKV 的功能。与使用 Spark 结合JDBC 写入TiDB 的方式相比，分布式写入 TiKV 能够实现事务（要么全部数据写入成功，要么全部都写入失败）。

> 警告：
>
> 由于 TiSpark 直接访问 TiKV，所以 TiDB Server 使用的访问控制机制并不适用于 TiSpark。TiSpark v2.5.0 及以上版本实现了部分鉴权与授权功能，具体信息请参考安全。

10.12.1.3 版本要求

- TiSpark 支持 Spark 2.3 或以上版本。
- TiSpark 需要JDK 1.8 以及 Scala 2.11/2.12 版本。
- TiSpark 可以运行在任何 Spark 模式上，如 YARN、Mesos 以及 Standalone。

10.12.1.4 推荐配置

> 警告：
>
> 此文所描述的使用TiUP 部署TiSpark 的方式已被废弃。

TiSpark 作为 Spark 的TiDB 连接器，需要 Spark 集群的支持。本文仅提供部署 Spark 的参考建议，对于硬件以及部署细节，请参考 Spark 官方文档。

对于独立部署的 Spark 集群，可以参考如下建议配置：

- 建议为 Spark 分配 32G 以上的内存，并为操作系统和缓存保留至少 25% 的内存。
- 建议每台机器至少为 Spark 分配 8 到 16 核 CPU。起初，你可以设定将所有 CPU 核分配给 Spark。

可以参考如下的 spark-env.sh 配置文件：

```
SPARK_EXECUTOR_MEMORY = 32g
SPARK_WORKER_MEMORY = 32g
SPARK_WORKER_CORES = 8
```

10.12.1.5 获取TiSpark

TiSpark 是 Spark 的第三方jar 包，提供读写TiKV 的能力。

10.12.1.5.1 获取mysql-connector-j

由于GPL 许可证的限制，TiSpark 不再提供mysql-connector-java 的依赖。以下版本将不再包含mysql-connector-java：

- TiSpark > 3.0.1
- TiSpark > 2.5.1 (TiSpark 2.5.x)
- TiSpark > 2.4.3 (TiSpark 2.4.x)

在使用TiSpark 写入与鉴权时，仍需要 mysql-connector-java 依赖，因此你需要手动下载，并使用以下方式引入：

- 将 mysql-connector-java 放入 Spark jars 包中。
- 在你提交 Spark 任务时，引入 mysql-connector-java，详见以下示例:

```
spark-submit --jars tispark-assembly-3.0_2.12-3.1.0-SNAPSHOT.jar,mysql-connector-java-8.0.29.jar
```

10.12.1.5.2 选择TiSpark 版本

你可以根据TiDB 和 Spark 版本选择相应的TiSpark 版本。

TiSpark 版本	TiDB、 TiKV、 PD 版本	Spark 版本	Scala 版本
2.4.x-scala_2.11	5.x, 4.x	2.3.x, 2.4.x	2.11
2.4.x-scala_2.12	5.x, 4.x	2.4.x	2.12
2.5.x	5.x, 4.x	3.0.x, 3.1.x	2.12
3.0.x	5.x, 4.x	3.0.x, 3.1.x, 3.2.x	2.12
3.1.x	6.x, 5.x, 4.x	3.0.x, 3.1.x, 3.2.x, 3.3.x	2.12

10.12.1.6 获取TiSpark jar 包

你能用以下方式获取jar 包：

- 从maven 中央仓库获取，你可以搜索 pingcap 关键词。
- 从 TiSpark releases 获取。
- 通过以下步骤从源码构建：

1. 下载TiSpark 源码：

```
git clone https://github.com/pingcap/tispark.git
cd tisapark
```

2. 在TiSpark 根目录运行如下命令：

```
// add -Dmaven.test.skip=true to skip the tests
mvn clean install -Dmaven.test.skip=true
// or you can add properties to specify spark version
mvn clean install -Dmaven.test.skip=true -Pspark3.2.1
```

> 注意：
> 目前，你只能使用java8 构架TiSpark。运行 `mvn -version` 来检查java 版本。

10.12.1.6.1 TiSpark jar 包的artifact ID

注意不同版本的TiSpark artifact ID 也不同：

TiSpark 版本	Artifact ID
2.4.x-${scala_version}, 2.5.0	tispark-assembly
2.5.1	tispark-assembly-${spark_version}
3.0.x, 3.1.x	tispark-assembly-${spark_version}-${scala_version}

10.12.1.7 快速开始

本章节将以 spark-shell 为例，介绍如何使用TiSpark。请保证您已下载 Spark。

10.12.1.7.1 启动 spark-shell

在 `spark-defaults.conf` 中添加如下配置：

```
spark.sql.extensions  org.apache.spark.sql.TiExtensions
spark.tispark.pd.addresses  ${your_pd_adress}
spark.sql.catalog.tidb_catalog  org.apache.spark.sql.catalyst.catalog.TiCatalog
spark.sql.catalog.tidb_catalog.pd.addresses  ${your_pd_adress}
```

启动 spark-shell：

```
spark-shell --jars tispark-assembly-{version}.jar
```

10.12.1.7.2 获取TiSpark版本

在spark-shell中运行如下命令，可获取TiSpark版本信息：

```
spark.sql("select ti_version()").collect
```

10.12.1.7.3 使用TiSpark读取数据

可以通过Spark SQL从TiKV读取数据：

```
spark.sql("use tidb_catalog")
spark.sql("select count(*) from ${database}.${table}").show
```

10.12.1.7.4 使用TiSpark写入数据

通过Spark DataSource API，可以在保证ACID前提下写入数据到TiKV：

```
val tidbOptions: Map[String, String] =
  Map( "tidb.addr" -> "127.0.0.1",
  "tidb.password" -> "",
  "tidb.port" -> "4000",
  "tidb.user" -> "root"
)

val customerDF = spark.sql("select * from customer limit 100000")

customerDF.write
.format("tidb")
.option("database", "tpch_test")
.option("table", "cust_test_select")
.options(tidbOptions)
.mode("append")
.save()
```

详见 Data Source API User Guide。

在TiSpark 3.1之后，你还能通过Spark SQL写入TiSpark 3.1。详见Insert SQL。

10.12.1.7.5 通过JDBC数据源写入数据

你同样可以在不使用TiSpark的情况下使用Spark JDBC数据源写入TiDB。

由于这超过了TiSpark的范畴，本文仅简单提供示例，详情请参考JDBC To Other Databases。

```
import org.apache.spark.sql.execution.datasources.jdbc.JDBCOptions

val customer = spark.sql("select * from customer limit 100000")
// 为了平衡各节点以及提高并发数，你可以将数据源重新分区
```

```
val df = customer.repartition(32)
df.write
.mode(saveMode = "append")
.format("jdbc")
.option("driver", "com.mysql.jdbc.Driver")
// 替换为你的主机名和端口地址，并确保开启了重写批处理
.option("url", "jdbc:mysql://127.0.0.1:4000/test?rewriteBatchedStatements=true")
.option("useSSL", "false")
// 作为测试，建议设置为 150
.option(JDBCOptions.JDBC_BATCH_INSERT_SIZE, 150)
.option("dbtable", s"cust_test_select") // 数据库名和表名
.option("isolationLevel", "NONE") // 如果需要写入较大 Dataframe，推荐将 isolationLevel 设置为
    ↪ NONE
.option("user", "root") // TiDB 用户名
.save()
```

为了避免大事务导致 OOM 以及 TiDB 报 `ISOLATION LEVEL does not support` 错误（TiDB 目前仅支持 `REPEATABLE-READ`），推荐设置 `isolationLevel` 为 `NONE`。

10.12.1.7.6 使用 TiSpark 删除数据

可以使用 Spark SQL 删除 TiKV 数据：

```
spark.sql("use tidb_catalog")
spark.sql("delete from ${database}.${table} where xxx")
```

详情请参考 delete feature。

10.12.1.7.7 与其他数据源一起使用

你可以使用多个 catalog 从不同数据源读取数据：

```
// 从 Hive 读取
spark.sql("select * from spark_catalog.default.t").show

// Join Hive 表和 TiDB 表
spark.sql("select t1.id,t2.id from spark_catalog.default.t t1 left join tidb_catalog.test.t t2").
    ↪ show
```

10.12.1.8 TiSpark 配置

可以将如下参数配置在 `spark-defaults.conf` 中，也可以参考 Spark 其他配置，用同样的方式传入。

Key	Default value	Description
spark. ′→ tispark. ′→ pd. ′→ addresses	127.0.0.P1D:集群的地址，通过逗号分隔。 ′→	
spark. ′→ tispark. ′→ grpc. ′→ framesize	21474836g4R7PC 的最大回复大小，单位bytes（默认 2G）。 ′→	
spark. ′→ tispark. ′→ grpc. ′→ timeout_in_sec ′→	10	gRPC 超时时间，单位秒。
spark. ′→ tispark. ′→ plan. ′→ allow_agg_pushdown ′→	true	是否运行聚合下推（为了避免TiKV 节点繁忙）。
spark. ′→ tispark. ′→ plan. ′→ allow_index_read ′→	true	是否在执行计划中开启index（可能导致TiKV 压力过大）。
spark. ′→ tispark. ′→ index. ′→ scan_batch_size ′→	20000 ′→	在 index scan 的一次batch 中 row keys 的数量。
spark. ′→ tispark. ′→ index. ′→ scan_concurrency ′→	5	在 index scan 中获取row keys 的最大线程数（每个JVM 的所有任务共享）。
spark. ′→ tispark. ′→ table. ′→ scan_concurrency ′→	512	在 table scan 中的最大线程数（每个JVM 的所有任务共享）。

Key	Default value	Description
spark.tispark.request.command.priority	Low	可选项有 Low、 Normal和High。修改配置会影响影响 TiKV 的资源分配。建议使用 Low，因为 OLTP 的负载不会被影响。
spark.tispark.coprocess.codec_format	chblock	Coprocessor 的编码格式。可选项为 default、 chblock 和 chunk。
spark.tispark.coprocess.streaming	false	是否在获取响应时使用 streaming（实验性质）。
spark.tispark.plan.unsupported_pushdown_exprs		表达式清单，多个表达式用逗号分隔。为了防止老版本的TiKV 不支持某些表达式，你可以禁止下推它们。
spark.tispark.plan.downgrade.index_threshold	10000000	如果 index scan 请求的范围超过了此限制，该Region 请求会被降级为table scan。降级默认关闭。
spark.tispark.show_rowid	false	是否在row_id 存在时显示它。
spark.tispark.db_prefix		TiDB 数据库的前缀。该配置可用于在TiSpark 2.4 中区分同名的TiDB 和 Hive 数据库。
spark.tispark.request.isolation.level	SI	是否为底层的TiKV 解锁。当你使用“RC” 级别，你将忽略锁，得到比 tso 更小的最新版本数据。当你使用“SI” 级别，你将进行解锁并根据该锁对应的事务提交与否获取数据。

Key	Default value	Description
spark.tispark.coprocessor.chunk_batch_size	1024	从 coprocessor 获取的一个batch 的 Row 数量。
spark.tispark.isolation_read_engines	tikv	TiSpark 读引擎，多个引擎使用逗号分隔。未配置的存储引擎不会被读取。
spark.tispark.stale_read	optional	stale read 时间戳。详情请参考 stale read。
spark.tispark.tikv.tls_enable	false	是否开启TiSpark TLS。
spark.tispark.tikv.trust_cert_collection		TiKV Client 的受信任根证书，用于验证 PD 证书。如 /home/tispark/config/root.pem，该文件需包含 X.509 证书。
spark.tispark.tikv.key_cert_chain		TiKV Client 的X.509 格式的客户端证书链。如 /home/tispark/config/client.pem。
spark.tispark.tikv.key_file		TiKV Client 的 PKCS#8 私钥文件。如 /home/tispark/client_pkcs8.key。
spark.tispark.tikv.jks_enable	false	是否使用 JAVA key store 而不是X.509 证书。

Key	Default value	Description
spark.tispark.tikv.jks_trust_path		TiKV Client JKS 格式的受信任根证书。由 keytool 生成，如 /home/tispark/config/tikv-truststore。
spark.tispark.tikv.jks_trust_password		spark.tispark.tikv.jks_trust_path 的密码。
spark.tispark.tikv.jks_key_path		TiKV Client JKS 格式的客户端证书。由 keytool 生成，如 /home/tispark/config/tikv-clientstore。
spark.tispark.tikv.jks_key_password		spark.tispark.tikv.jks_key_path 的密码。
spark.tispark.jdbc.tls_enable	false	是否开启JDBC connector TLS。
spark.tispark.jdbc.server_cert_store		JDBC 的受信任根证书。由 keytool 生产的Java keystore (JKS) 格式的证书。如 /home/tispark/config/jdbc-truststore。默认值为 “”，表示TiSpark 不会校验 TiDB 服务端。
spark.tispark.jdbc.server_cert_password		spark.tispark.jdbc.server_cert_store 的密码。
spark.tispark.jdbc.client_cert_store		JDBC 的 PKCS#12 格式的客户端证书。这是由 keytool 生成的JKS 格式证书。如 /home/tispark/config/jdbc-clientstore。默认值为 “”，表示TiDB 不会校验 TiSpark。

Key	Default value	Description
spark.tispark.jdbc.client_cert_password		spark.tispark.jdbc.client_cert_store 的密码。
spark.tispark.tikv.tls_reload_interval	10s	重载证书的时间间隔。默认值为 10s。
spark.tispark.tikv.conn_recycle_time	60s	清理TiKV 失效连接的时间间隔。默认时间为 60s。当重载证书开启时此配置才会生效。
spark.tispark.host_mapping		路由映射配置。用于配置公有IP 地址和私有IP 地址的映射。当TiDB 在私有网络上运行时，你可以将一系列内部IP 地址映射到公网IP 地址以便 Spark 集群访问。其格式为 {Intranet IP1}:{Public IP1};{Intranet IP2}:{Public IP2}，例如 192.168.0.2:8.8.8.8;192.168.0.3:9.9.9.9。
spark.tispark.new_collation_enable		当 TiDB 开启 new collation，推荐将此配置设为true。当 TiDB 关闭 new collation，推荐将此配置设置为 false。在未配置的情况下，TiSpark 会依据TiDB 版本自动配置 new collation。其规则为：当 TiDB 版本大于等于 v6.0.0 时为 true；否则为 false。
spark.tispark.replica_read	leader	读取副本的类型。可选值为 leader、follower、learner。可以同时指定多个类型，TiSpark 会根据顺序选择。
spark.tispark.replica_read.label		目标TiKV 节点的标签。格式为 label_x=value_x,label_y=value_y，各项之间为“逻辑与”的关系。

10.12.1.8.1 TLS 配置

TiSpark TLS 分为两部分：TiKV Client TLS 以及JDBC connector TLS。前者用于创建和 TiKV/PD 的 TLS 连接，后者用于创建与TiDB 的 TLS 连接。

当配置TiKV Client TLS 时，你需要以X.509 格式的证书配置 tikv.trust_cert_collection、tikv.key_cert_chain 和 tikv.key_file；或者以 JKS 格式的证书配置 tikv.jks_enable，tikv.jks_trust_path 和 tikv.jks_key_path。

当配置 JDBC connector TLS 时， 你需要配置 spark.tispark.jdbc.tls_enable， 而 jdbc.server_cert_store 和 jdbc.client_cert_store 则是可选的。

TiSpark 目前仅持 TLS 1.2 and TLS 1.3。

- 如下是使用X.509 证书配置TiKV Client TLS 的例子：

```
spark.tispark.tikv.tls_enable                                    true
spark.tispark.tikv.trust_cert_collection                         /home/tispark/root.pem
spark.tispark.tikv.key_cert_chain                                /home/tispark/client.pem
spark.tispark.tikv.key_file                                      /home/tispark/client.key
```

- 如下是使用JKS 配置 TiKV Client TLS 的例子：

```
spark.tispark.tikv.tls_enable                                    true
spark.tispark.tikv.jks_enable                                    true
spark.tispark.tikv.jks_key_path                                  /home/tispark/config/tikv-
    ′→ truststore
spark.tispark.tikv.jks_key_password                              tikv_trustore_password
spark.tispark.tikv.jks_trust_path                                /home/tispark/config/tikv-
    ′→ clientstore
spark.tispark.tikv.jks_trust_password                            tikv_clientstore_password
```

当你同时配置JKS 和 X.509 证书时，JKS 优先级更高。因此，当你只想使用普通的pem 证书时，不要同时设置 `spark.tispark.tikv.jks_enable=true`。

- 下面是一个配置JDBC connector TLS 的例子：

```
spark.tispark.jdbc.tls_enable                                    true
spark.tispark.jdbc.server_cert_store                             /home/tispark/jdbc-truststore
spark.tispark.jdbc.server_cert_password                          jdbc_truststore_password
spark.tispark.jdbc.client_cert_store                             /home/tispark/jdbc-clientstore
spark.tispark.jdbc.client_cert_password                          jdbc_clientstore_password
```

- 对于如何开启TiDB TLS，请参考 Enable TLS between TiDB Clients and Servers。
- 对于如何生成JAVA key store，请参考 Connecting Securely Using SSL。

10.12.1.8.2 时区配置

使用 `-Duser.timezone` 系统参数来配置时区（比如 `-Duser.timezone=GMT-7`）。时区会影响 `Timestamp` 数据类型。

请不要使用 `spark.sql.session.timeZone`。

10.12.1.9 特性

TiSpark 的主要特性如下：

特性支持	TiSpark 2.4.x	TiSpark 2.5.x	TiSpark 3.0.x	TiSpark 3.1.x
SQL select without tidb_catalog	✓	✓		
SQL select with tidb_catalog		✓	✓	✓
DataFrame append	✓	✓	✓	✓
DataFrame reads	✓	✓	✓	✓
SQL show databases	✓	✓	✓	✓
SQL show tables	✓	✓	✓	✓
SQL auth		✓	✓	✓
SQL delete			✓	✓
SQL insert				✓
TLS			✓	✓
DataFrame auth				✓

10.12.1.9.1 Expression index 支持

TiDB v5.0 开始支持expression index。

TiSpark 目前支持从 `expression index` 的表中获取数据，但 `expression index` 不会被TiSpark 执行计划使用。

10.12.1.9.2 TiFlash 支持

TiSpark 能够通过配置 `spark.tispark.isolation_read_engines` 从TiFlash 读取数据。

10.12.1.9.3 分区表支持

读分区表

TiSpark 目前支持读取range 与 hash 分区表。

TiSpark 目前不支持 `partition table` 语法 `select col_name from table_name partition(partition_name)`。但是，你仍可以使用 `where` 条件过滤分区。

TiSpark 会根据分区类型、分区表达式以及具体 SQL 决定是否进行分区裁剪。目前，TiSpark 仅支持在range 分区下，且在下列任一条件下进行分区裁剪：

- 列表达式，如 `partition by col1`。
- 形如 `YEAR($col)` 的year 函数，其中 col 为列名且类型为datetime、date 或能被解析为datetime、date 的 string 字面量。
- 形如 `TO_DAYS($col)` 的 to_days 函数，其中col 为列名且类型为datetime、 date 或能被解析为datetime、 date 的 string 字面量。

如果分区裁剪未被应用，TiSpark 将会读取所有分区表。

写分区表

目前, TiSpark 仅支持写入range 与 hash 分区表，且需满足以下任一条件：

- 列表达式，如 `partition by col1`。

- 形如 YEAR($col) 的year 函数，其中 col 为列名且类型为datetime、date 或能被解析为datetime、date 的 string 字面量。

> 注意：
> 目前，TiSpark 只支持在utf8mb4_bin 字符集下写入分区表。

有两种方式写入分区表：

- 使用支持replace 和 append 语义的 Datasource API 写入分区表。
- 使用 Spark SQL 删除语句。

10.12.1.9.4 安全

从 TiSpark v2.5.0 起，你可以通过TiDB 对 TiSpark 进行鉴权与授权。

该功能默认关闭。要开启该功能，请在 Spark 配置文件 spark-defaults.conf 中添加以下配置项：

```
// 开启鉴权与授权功能
spark.sql.auth.enable true

// 配置 TiDB 信息
spark.sql.tidb.addr $your_tidb_server_address
spark.sql.tidb.port $your_tidb_server_port
spark.sql.tidb.user $your_tidb_server_user
spark.sql.tidb.password $your_tidb_server_password
```

更多详细信息，请参考TiSpark 鉴权与授权指南。

10.12.1.9.5 其他特性

- 下推
- TiSpark 删除数据
- 历史读
- TiSpark with multiple catalogs
- TiSpark TLS
- TiSpark 执行计划

10.12.1.10 统计信息

TiSpark 可以使用TiDB 的统计信息：

- 选择代价最低的索引访问
- 估算数据大小以决定是否进行广播优化

如果你希望TiSpark 使用统计信息支持，需要确保所涉及的表已经被分析。参考统计信息简介了解如何进行表分析。

从 TiSpark 2.0 开始，统计信息将会默认被读取。

10.12.1.11 FAQ

详情请参考TiSpark FAQ。

10.13 sync-diff-inspector

10.13.1 sync-diff-inspector 用户文档

sync-diff-inspector 是一个用于校验 MySQL / TiDB 中两份数据是否一致的工具。该工具提供了修复数据的功能 (适用于修复少量不一致的数据)。

主要功能：

- 对比表结构和数据
- 如果数据不一致，则生成用于修复数据的 SQL 语句
- 支持不同库名或表名的数据校验
- 支持分库分表场景下的数据校验
- 支持TiDB 主从集群的数据校验
- 支持从 TiDB DM 拉取配置的数据校验

你可通过以下方式下载 sync-diff-inspector：

- Binary 包。sync-diff-inspector 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。
- Docker 镜像。执行以下命令进行下载：

```
docker pull pingcap/tidb-tools:latest
```

10.13.1.1 sync-diff-inspector 的使用限制

- 对于 MySQL 和 TiDB 之间的数据同步不支持在线校验，需要保证上下游校验的表中没有数据写入，或者保证某个范围内的数据不再变更，通过配置 `range` 来校验这个范围内的数据。
- FLOAT、 DOUBLE 等浮点数类型在TiDB 和 MySQL 中的实现方式不同，在计算checksum 时会分别取 6 位和 15 位有效数字。如果不使用该特性，需要设置 `ignore-columns` 忽略这些列的检查。
- 支持对不包含主键或者唯一索引的表进行校验，但是如果数据不一致，生成的用于修复的 SQL 可能无法正确修复数据。

10.13.1.2 sync-diff-inspector 所需的数据库权限

sync-diff-inspector 需要获取表结构信息、查询数据，需要的数据库权限如下：

- 上游数据库
  - SELECT（查数据进行对比）
  - SHOW_DATABASES（查看库名）
  - RELOAD（查看表结构）
- 下游数据库
  - SELECT（查数据进行对比）
  - SHOW_DATABASES（查看库名）
  - RELOAD（查看表结构）

10.13.1.3 配置文件说明

sync-diff-inspector 的配置总共分为五个部分：

- Global config：通用配置，包括校验的线程数量、是否输出修复 SQL、是否比对数据、是否跳过校验上游或下游不存在的表等。
- Datasource config：配置上下游数据库实例。
- Routes：上游多表名通过正则匹配下游单表名的规则。（可选）
- Task config：配置校验哪些表，如果有的表在上下游有一定的映射关系或者有一些特殊要求，则需要对指定的表进行配置。
- Table config：对具体表的特殊配置，例如指定范围、忽略的列等等。（可选）

下面是一个完整配置文件的说明：

- 提示：配置名后带s 的配置项允许拥有多个配置值，因此需要使用方括号 [] 来包含配置值。

```
### Diff Configuration.

########################## Global config ##########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4

### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句。
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false

### 如果开启，会跳过校验上游或下游不存在的表。
```

```
skip-non-existing-table = false

########################### Datasource config ########################
[data-sources]
[data-sources.mysql1] # mysql1 是该数据库实例唯一标识的自定义 id，用于下面 task.source-instances/
    ′→ task.target-instance 中
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = "" # 设置连接上游数据库的密码，可为明文或 Base64 编码。

    # ( 可选 ) 使用映射规则来匹配上游多个分表，其中 rule1 和 rule2 在下面 Routes 配置栏中定义
    route-rules = ["rule1", "rule2"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = "" # 设置连接下游数据库的密码，可为明文或 Base64 编码。

    # ( 可选 ) 使用 TLS 连接 TiDB
    # security.ca-path = ".../ca.crt"
    # security.cert-path = ".../cert.crt"
    # security.key-path = ".../key.crt"

    # ( 可选 ) 使用 TiDB 的 snapshot 功能，如果开启的话会使用历史数据进行对比
    # snapshot = "386902609362944000"
    # 当 snapshot 设置为 "auto" 时，使用 TiCDC 在上下游的同步时间点，具体参考 <https://github.com
        ′→ /pingcap/tidb-tools/issues/663>
    # snapshot = "auto"

############################## Routes ############################
### 如果需要对比大量的不同库名或者表名的表的数据，或者用于校验上游多个分表与下游总表的数据，
    ′→ 可以通过 table-rule 来设置映射关系
### 可以只配置 schema 或者 table 的映射关系，也可以都配置
[routes]
[routes.rule1] # rule1 是该配置的唯一标识的自定义 id，用于上面 data-sources.route-rules 中
schema-pattern = "test_*"   # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "t_*"     # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"       #  目标库名
target-table = "t" # 目标表名

[routes.rule2]
schema-pattern = "test2_*"     #  匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "t2_*"          # 匹配数据源的表名，支持通配符 "*" 和 "?"
```

```
target-schema = "test2"          # 目标库名
target-table = "t2" # 目标表名

########################### Task config ########################
### 配置需要对比的*目标数据库*中的表
[task]
    # output-dir 会保存如下信息
    # 1 sql：检查出错误后生成的修复 SQL 文件，并且一个 chunk 对应一个文件
    # 2 log：sync-diff.log 保存日志信息
    # 3 summary：summary.txt 保存总结
    # 4 checkpoint：a dir 保存断点续传信息
    output-dir = "./output"

    # 上游数据库，内容是 data-sources 声明的唯一标识 id
    source-instances = ["mysql1"]

    # 下游数据库，内容是 data-sources 声明的唯一标识 id
    target-instance = "tidb0"

    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    # 使用 ? 来匹配任意一个字符；使用 * 来匹配任意；详细匹配规则参考 golang regexp pkg: https://
        '→ github.com/google/re2/wiki/Syntax
    target-check-tables = ["schema*.table*", "!c.*", "test2.t2"]

    # ( 可选 ) 对部分表的额外配置，其中 config1 在下面 Table config 配置栏中定义
    target-configs = ["config1"]

########################### Table config ########################
### 对部分表进行特殊的配置，配置的表必须包含在 task.target-check-tables 中
[table-configs.config1] # config1 是该配置的唯一标识自定义 id，用于上面 task.target-configs 中
### 目标表名称，可以使用正则来匹配多个表，但不允许存在一个表同时被多个特殊配置匹配。
target-tables = ["schema*.test*", "test2.t2"]
# ( 可选 ) 指定检查的数据的范围，需要符合 sql 中 where 条件的语法
range = "age > 10 AND age < 20"
# ( 可选 ) 指定用于划分 chunk 的列，如果不配置该项，sync-diff-inspector 会选取一些合适的列 (
    '→ 主键 / 唯一键 / 索引 )
index-fields = ["col1","col2"]
# ( 可选 ) 忽略某些列的检查，例如 sync-diff-inspector 目前还不支持的一些类型 ( json，bit，blob 等
    '→ ) ，
### 或者是浮点类型数据在 TiDB 和 MySQL 中的表现可能存在差异，可以使用 ignore-columns
    '→ 忽略检查这些列
ignore-columns = ["",""]
# ( 可选 ) 指定划分该表的 chunk 的大小，若不指定可以删去或者将其配置为 0。
chunk-size = 0
# ( 可选 ) 指定该表的 collation，若不指定可以删去或者将其配置为空字符串。
```

```
collation = ""
```

10.13.1.4 运行 sync-diff-inspector

执行如下命令：

```
./sync_diff_inspector --config=./config.toml
```

该命令最终会在 `config.toml` 中的 `output-dir` 输出目录输出本次比对的检查报告 `summary.txt` 和日志 `sync_diff.log`。在输出目录下还会生成由 `config.toml` 文件内容哈希值命名的文件夹，该文件夹下包括断点续传checkpoint 结点信息以及数据存在不一致时生成的 SQL 修复数据。

10.13.1.4.1 前台输出

sync-diff-inspector 在执行过程中会往 `stdout` 发送进度信息。进度信息包括表的结构比较结果、表的数据比较结果以及进度条。

> **注意：**
>
> 为了达成显示效果，请保持显示窗口宽度在 80 字符以上。

```
A total of 2 tables need to be compared

Comparing the table structure of ``sbtest`.`sbtest96`` ... equivalent
Comparing the table structure of ``sbtest`.`sbtest99`` ... equivalent
Comparing the table data of ``sbtest`.`sbtest96`` ... failure
Comparing the table data of ``sbtest`.`sbtest99`` ...

Progress [============================================================>--] 98% 193/200
```

```
A total of 2 tables need to be compared

Comparing the table structure of ``sbtest`.`sbtest96`` ... equivalent
Comparing the table structure of ``sbtest`.`sbtest99`` ... equivalent
Comparing the table data of ``sbtest`.`sbtest96`` ... failure
Comparing the table data of ``sbtest`.`sbtest99`` ... failure

Progress [============================================================>] 100% 0/0
The data of `sbtest`.`sbtest99` is not equal
The data of `sbtest`.`sbtest96` is not equal

The rest of tables are all equal.
```

```
A total of 2 tables have been compared, 0 tables finished, 2 tables failed, 0 tables skipped.
The patch file has been generated in
        'output/fix-on-tidb2/'
You can view the comparison details through 'output/sync_diff.log'
```

10.13.1.4.2 输出文件

输出文件目录结构如下：

```
output/
|-- checkpoint # 保存断点续传信息
| |-- bbfec8cc8d1f58a5800e63aa73e5 # config hash 占位文件，标识该输出目录（output/）
    ′→ 对应的配置文件
| |-- DO_NOT_EDIT_THIS_DIR
| └-- sync_diff_checkpoints.pb # 断点续传信息
|
|-- fix-on-target # 保存用于修复不一致的 SQL 文件
| |-- xxx.sql
| |-- xxx.sql
| └-- xxx.sql
|
|-- summary.txt # 保存校验结果的总结
└-- sync_diff.log # 保存 sync-diff-inspector 执行过程中输出的日志信息
```

日志

sync-diff-inspector 的日志存放在${output}/sync_diff.log 中，其中${output} 是config.toml 文件中output-dir 的值。

校验进度

sync-diff-inspector 会在运行时定期（间隔 10s）输出校验进度到 checkpoint 中 (位于 ${output}/checkpoint/
′→ sync_diff_checkpoints.pb 中，其中 ${output} 是 config.toml 文件中 output-dir 的值。

校验结果

当校验结束时，sync-diff-inspector 会输出一份校验报告，位于 ${output}/summary.txt 中，其中 ${output} 是 config.toml 文件中 output-dir 的值。

```
+--------------------+---------+--------------------+----------------+---------+----------------------+
|        TABLE       | RESULT  | STRUCTURE EQUALITY | DATA DIFF ROWS | UPCOUNT | DOWNCOUNT |
+--------------------+---------+--------------------+----------------+---------+----------------------+
| `sbtest`.`sbtest99` | succeed | true               | +97/-97        |  999999 |    999999 |
| `sbtest`.`sbtest96` | succeed | true               | +0/-101        |  999999 |   1000100 |
| `sbtest`.`sbtest97` | skipped | false              | +999999/-0     |  999999 |         0 |
+--------------------+---------+--------------------+----------------+---------+----------------------+
Time Cost: 16.75370462s
Average Speed: 113.277149MB/s
```

- TABLE：该列表示对应的数据库及表名
- RESULT：校验是否完成。如果设置了 skip-non-existing-table = true，对于上游或下游不存在的表，该列的值将为 skipped
- STRUCTURE EQUALITY：表结构是否相同
- DATA DIFF ROWS：即 rowAdd/rowDelete，表示该表修复需要增加/删除的行数

SQL 修复

校验过程中遇到不同的行，会生成修复数据的 SQL 语句。一个chunk 如果出现数据不一致，就会生成一个以 chunk.Index 命名的 SQL 文件。文件位于 ${output}/fix-on-${instance} 文件夹下。其中 ${instance} 为 config.toml 中 task.target-instance 的值。

一个 SQL 文件会包含该chunk 的所属表以及表示的范围信息。对每个修复 SQL 语句，有三种情况：

- 下游数据库缺失行，则是REPLACE 语句
- 下游数据库冗余行，则是 DELETE 语句
- 下游数据库行部分数据不一致，则是REPLACE 语句，但会在 SQL 文件中通过注释的方法标明不同的列

```
-- table: sbtest.sbtest99
-- range in sequence: (3690708) < (id) <= (3720581)
/*
  DIFF COLUMNS ▫    `K`   ▫                `C`                ▫              `PAD`
═══════════════╪═══════╪══════════════════════════════════════╪═══════════════
    '→
  source data  ▫ 2501808 ▫ 'hello'                            ▫ 'world'
═══════════════╪═══════╪══════════════════════════════════════╪═══════════════
    '→
  target data  ▫ 5003616 ▫ '0709824117-9809973320-4456050422' ▫
      '→ '1714066100-7057807621-1425865505'
═══════════════╪═══════╪══════════════════════════════════════╪═══════════════
    '→
*/
REPLACE INTO `sbtest`.`sbtest99`(`id`,`k`,`c`,`pad`) VALUES (3700000,2501808,'hello','world');
```

10.13.1.5 注意事项

- sync-diff-inspector 在校验数据时会消耗一定的服务器资源，需要避免在业务高峰期间校验。
- 在数据对比前，需要注意表中的collation 设置。如果表的主键或唯一键为 varchar 类型，且上下游数据库中collation 设置不同，可能会因为排序问题导致最终校验结果不正确，需要在 sync-diff-inspector 的配置文件中增加collation 设置。
- sync-diff-inspector 会优先使用TiDB 的统计信息来划分chunk，需要尽量保证统计信息精确，可以在业务空闲期手动执行 analyze table {table_name}。
- table-rule 的规则需要特殊注意，例如设置了 schema-pattern="test1"，table-pattern = "t_1"，target-schema="test2"，target-table = "t_2"，会对比 source 中的表 test1.t_1 和 target 中的表 test2.t_2。sync-diff-inspector 默认开启 sharding，如果 source 中还有表 test2.t_2，则会把 source 端的表 test1.t_1 和表 test2.t_2 作为 sharding 与 target 中的表 test2.t_2 进行一致性校验。
- 生成的 SQL 文件仅作为修复数据的参考，需要确认后再执行这些 SQL 修复数据。

10.13.2 不同库名或表名的数据校验

当你在使用TiDB DM 等同步工具时，可以设置 `route-rules` 将数据同步到下游指定表中。sync-diff-inspector 通过设置 `rules` 提供了校验不同库名、表名的表的功能。

下面是一个简单的配置文件说明，要了解完整配置，请参考sync-diff-inspector 用户文档。

```
########################### Datasource config ###########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = ""

############################## Routes ############################
[routes.rule1]
schema-pattern = "test_1"   # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "t_1"     # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test_2"      #  目标库名
target-table = "t_2" # 目标表名
```

使用该配置会对下游的 `test_2.t_2` 与实例 `mysql1` 中的 `test_1.t_1` 进行校验。

如果需要校验大量的不同库名或者表名的表，也可以通过 `rules` 设置映射关系来简化配置。可以只配置schema 或者table 的映射关系，也可以都配置。例如上游库 `test_1` 中的所有表都同步到了下游的 `test_2` 库中，可以使用如下配置进行校验：

```
########################### Datasource config ###########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
```

```
    password = ""

############################# Routes ############################
[routes.rule1]
schema-pattern = "test_1"  # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "*"            # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test_2"     # 目标库名
target-table = "t_2"           # 目标表名
```

10.13.2.1 注意事项

如果上游数据库有 `test_2.t_2` 也会被下游数据库匹配到。

10.13.3 分库分表场景下的数据校验

sync-diff-inspector 支持对分库分表场景进行数据校验。例如有多个 MySQL 实例，当你使用同步工具TiDB DM 同步到一个TiDB 时，可以使用 sync-diff-inspector 对上下游数据进行校验。

10.13.3.1 使用 datasource config 进行配置

使用 `Datasource config` 对 `table-0` 进行特殊配置，设置对应 `rules`，配置上游表与下游表的映射关系。这种配置方式需要对所有分表进行设置，适合上游分表数量较少，且分表的命名规则没有规律的场景。场景如图所示：

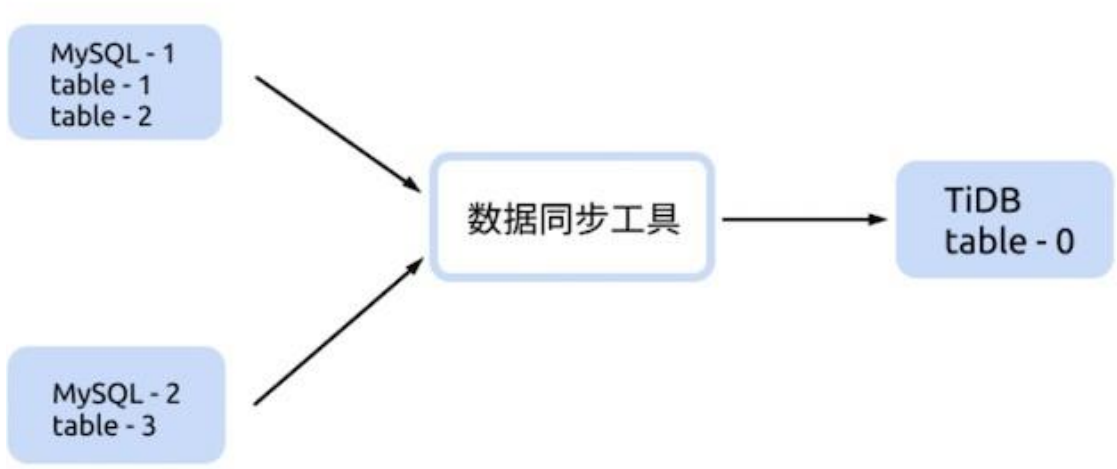


图 235: shard-table-sync-1

sync-diff-inspector 完整的示例配置如下：

```
### Diff Configuration.

############################ Global config ########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4

### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false


############################ Datasource config ########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.mysql2]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule2"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = ""

############################ Routes ########################
[routes.rule1]
schema-pattern = "test"   #  匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "table-[1-2]" # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"      #  目标库名
target-table = "table-0"       # 目标表名
```

```
[routes.rule2]
schema-pattern = "test"       # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "table-3"     # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"        # 目标库名
target-table = "table-0"      # 目标表名

########################### Task config ##########################
[task]
    output-dir = "./output"

    source-instances = ["mysql1", "mysql2"]

    target-instance = "tidb0"

    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    target-check-tables = ["test.table-0"]
```

当上游分表较多，且所有分表的命名都符合一定的规则时，则可以使用 `table-rules` 进行配置。场景如图所示：

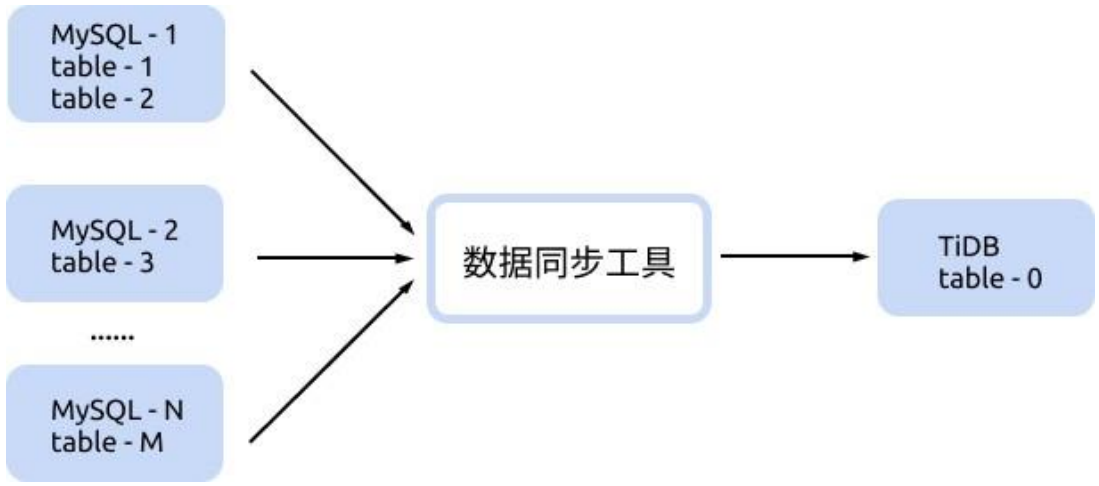


图 236: shard-table-sync-2

sync-diff-inspector 完整的示例配置如下：

```
### Diff Configuration.

########################### Global config ##########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4
```

```
### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false


########################### Datasource config ########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.mysql2]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = ""

############################## Routes ############################
[routes.rule1]
schema-pattern = "test"   # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "table-*"   # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"     #  目标库名
target-table = "table-0"    #  目标表名

########################### Task config ########################
[task]
    output-dir = "./output"

    source-instances = ["mysql1", "mysql2"]

    target-instance = "tidb0"
```

```
    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    target-check-tables = ["test.table-0"]
```

10.13.3.2 注意事项

如果上游数据库有 `test.table-0` 也会被下游数据库匹配到。

10.13.4 TiDB 主从集群的数据校验

当你使用TiCDC 搭建TiDB 的主从集群时，可能会需要在不停止同步的情况下对上下游的数据进行一致性验证。在普通的同步模式中，TiCDC 只提供数据的最终一致性的保证，而无法确保在同步的过程中数据的一致性。因此，对动态变更的数据进行一致性校验非常困难，为了满足这一需求，TiCDC 提供了 Syncpoint 功能。

Syncpoint 通过利用TiDB 提供的 snapshot 特性，让TiCDC 在同步过程中维护了一个上下游具有一致性 snapshot 的 `ts-map`。把校验动态数据的一致性问题转化为了校验静态 snapshot 数据的一致性问题，达到了接近数据一致性实时校验的效果。

要开启 SyncPoint 功能，你可以在创建同步任务时把 TiCDC 的配置项 `enable-sync-point` 设置为 `true`。开启 Syncpoint 功能后，TiCDC 在数据的同步过程中会根据你所配置的TiCDC 参数 `sync-point-interval` 来定时对齐上下游的 snapshot，并将上下游的 TSO 对应关系保存在下游的 `tidb_cdc.syncpoint_v1` 表中。

然后，你只需要在 sync-diff-inspector 中配置 `snapshot` 即可对TiDB 主从集群的数据进行校验。以下TiCDC 配置示例为创建的同步任务开启 Syncpoint 功能：

```
### 开启 SyncPoint
enable-sync-point = true

### 每隔 5 分钟对齐一次上下游的 snapshot
sync-point-interval = "5m"

### 每隔 1 小时清理一次下游 tidb_cdc.syncpoint_v1 表中的 ts-map 数据
sync-point-retention = "1h"
```

10.13.4.1 获取ts-map

在下游TiDB 中执行以下 SQL 语句，从结果中可以获取上游TSO (primary_ts) 和下游TSO (secondary_ts) 信息。

```
select * from tidb_cdc.syncpoint_v1;
```

```
+------------------+----------------+--------------------+--------------------+---------------------------------------+
    ′→
| ticdc_cluster_id | changefeed     | primary_ts         | secondary_ts       | created_at
    ′→          |
+------------------+----------------+--------------------+--------------------+---------------------------------------+
    ′→
```

```
| default          | test-2         | 435953225454059520 | 435953235516456963 | 2022-09-13
    ↪ 08:40:15 |
+------------------+----------------+--------------------+--------------------+---------------------------------------+
    ↪
```

以上 syncpoint_v1 表中各列所代表的信息如下:

- ticdc_cluster_id：插入该条记录的 TiCDC 集群的ID。
- changefeed：插入该条记录的 Changefeed 的ID。由于不同的TiCDC 集群可能会存在重名的 Changefeed，所以需要通过TiCDC 集群ID 和 Changefeed 的 ID 来确认一个 Changefeed 所插入的 ts-map。
- primary_ts：上游数据库 snapshot 的时间戳。
- secondary_ts：下游数据库 snapshot 的时间戳。
- created_at：插入该条记录的时间。

10.13.4.2 配置 snapshot

使用上一步骤获取的ts-map 信息来配置上下游数据库的 snapshot 信息。其中的 Datasource config 部分示例配置如下：

```
############################ Datasource config ########################
[data-sources.uptidb]
    host = "172.16.0.1"
    port = 4000
    user = "root"
    password = ""
    snapshot = "435953225454059520"

[data-sources.downtidb]
    host = "172.16.0.2"
    port = 4000
    user = "root"
    password = ""
    snapshot = "435953235516456963"
```

10.13.4.3 注意事项

- TiCDC 在创建 Changefeed 前，请确保 TiCDC 的配置项 enable-sync-point 已设置为 true，这样才会开启 Syncpoint 功能，在下游保存 ts-map。完整的配置请参考TiCDC 同步任务配置文件描述。
- 需要调整TiKV 的 GC 时间，保证在校验时 snapshot 对应的历史数据不会被执行GC。建议调整为 1 个小时，在校验后再还原GC 设置。
- 以上配置只展示了 Datasource config 部分，完整配置请参考sync-diff-inspector 用户文档。
- 从 v6.4.0 开始，TiCDC 使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者 SUPER 权限。

10.13.5 基于 DM 同步场景下的数据校验

当你在使用TiDB DM 等同步工具时，需要校验DM 同步后数据的一致性。你可以从DM-master 拉取指定task-name 的配置，进行数据校验。

下面是一个简单的配置文件说明，要了解完整配置，请参考sync-diff-inspector 用户文档。

```
### Diff Configuration.

########################## Global config #########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4

### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false

### dm-master 的地址，格式为 "http://127.0.0.1:8261"
dm-addr = "http://127.0.0.1:8261"

### 指定 DM 的 `task-name`
dm-task = "test"

########################## Task config ###########################
[task]
    output-dir = "./output"

    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    target-check-tables = ["hb_test.*"]
```

该配置在dm-task = “test” 中，会对该任务下hb_test 库的所有表进行检验，自动从 DM 配置中获取上游对下游库名的正则匹配，以校验 DM 同步后数据的一致性。

10.14 TiUniManager

10.14.1 TiUniManager 概览

TiUniManager 是为分布式数据库TiDB 打造的管控平台软件和数据库运维管理平台，主要为TiDB 提供数据库集群管理功能、主机管理功能和平台管理功能，涵盖了数据库运维人员 (DBA) 在TiDB 上进行的常用运维操作，帮助 DBA 对 TiDB 进行自动化、自助化和可视化管理。

TiUniManager 可帮助 DBA 避免因人工操作失误导致的数据库故障，保障数据库安全、稳定、高效地运行，降低运维TiDB 的难度，提升 DBA 工作效率。

自 v1.0.2 版本起，TiUniManager 正式开放源码，详见GitHub 仓库tiunimanager。

10.14.1.1 软件架构

TiUniManager 的软件架构图如下。

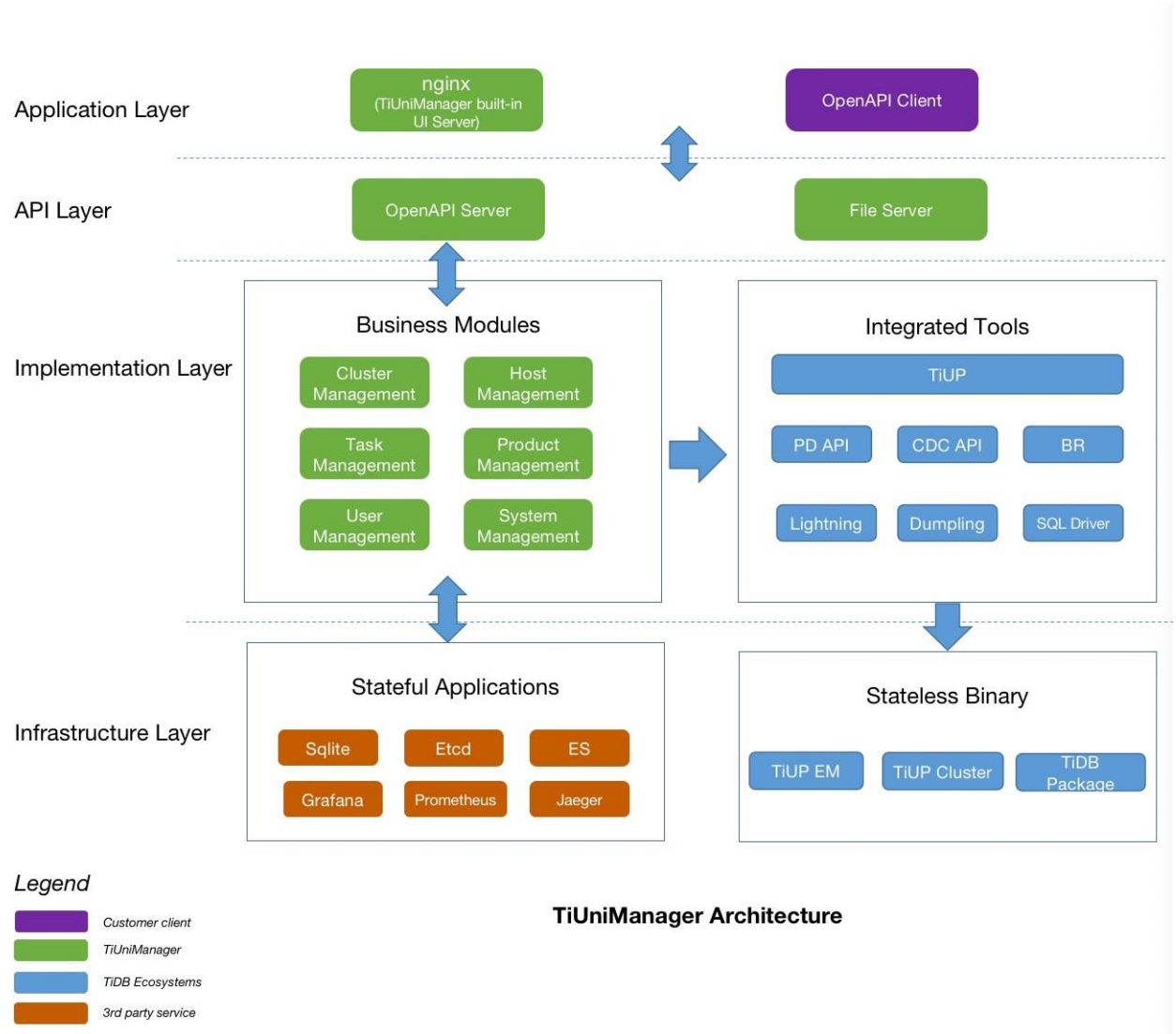


图 237: TiUniManager 架构

10.14.1.1.1 Web UI

TiUniManager Web UI 即用户可见的TiUniManager 界面，展示用户输入以及服务端返回的数据。用户可与该界面进行交互，通过输入数据来获取想要看到的数据，通过界面操作完成对TiDB 的部署运维相关工作。

10.14.1.1.2 OpenAPI

TiUniManager 提供数据库管理操作的 OpenAPI，帮助用户灵活完成自动化任务或第三方集成。当前 OpenAPI 为实验特性，不建议在生产环境中使用。

10.14.1.1.3 业务逻辑

TiUniManager 平台核心功能实现层，包括一键创建TiDB 集群、删除集群、扩容集群、恢复集群、查看集群详情、集群监控报警、错误日志、慢查询日志等。实现层负责以下方面：

- 负责后台定期备份TiDB 数据库并保存到分布式文件系统。
- 负责管理平台中所有主机信息的导入，主机的上线，记录主机资源使用情况。
- 负责管理平台自身的配置信息，元信息的备份，性能监控等。

10.14.1.2 核心特性

本节介绍TiUniManager 的核心特性。

10.14.1.2.1 易用性

TiUniManager 提供Web UI 界面交互模式，UI 界面即开即用。用户根据规划添加主机形成统一的资源池，通过TiUniManager 可一键创建指定规格的TiDB 实例，数分钟内就可部署完成集群并对外提供服务。

数据库管理员(DBA) 根据业务发展情况可随时弹性扩容TiDB 容量及服务能力。TiUniManager 集成 DBA 常用的运维操作，所有操作都通过可视化的方式操作，极大提升 DBA 运维TiDB 效率及易用性。

10.14.1.2.2 高可靠性

TiUniManager 支持以下特性，保障数据高可靠：

- 支持自动备份TiDB 数据库。用户可以设置自动备份的周期和时间。
- 支持按备份文件进行数据恢复，且支持恢复到新的实例。为避免在恢复数据过程中影响业务，数据验证通过后TiUniManager 将业务流量切换到新实例，完成数据回溯。
- 支持将数据备份文件存储在分布式文件系统，通过多副本冗余和副本分布的不同策略（不同机架等）确保数据不会丢失。
- 支持将TiDB 部署在不同的可用区。TiDB 自身采用多副本的数据存储，TiUniManager 根据调度策略将TiDB 的副本放在不同的可用区，实现跨机房数据可靠性功能。

10.14.1.2.3 高可用性

TiUniManager 组件均提供高可用功能，无单点故障，确保平台高可用。

TiDB 自身采用多副本的数据存储，TiUniManager 根据调度策略将TiDB 的副本放在不同的可用区，并采用业界先进的Raft 多数派选举算法确保数据 100% 强一致性和高可用。可将副本跨地域部署在不同的数据中心，主副本故障时自动切换，无需人工介入，自动保障业务的连续性，实现真正意义上的异地多活。

10.14.1.3 使用场景

本节介绍TiUniManager 的使用场景。

10.14.1.3.1 资源管理

- 支持导入主机
- 查看主机资源列表

10.14.1.3.2 集群管理

- 支持一键创建（部署）TiDB 集群，创建过程中可选择可用区、版本、规格、节点个数等
- 支持一键接管已有TiDB 集群
- 支持一键删除TiDB 集群
- 支持一键弹性扩容TiDB 集群，可分别扩展计算和存储能力
- 支持一键备份与恢复集群数据
- 支持一键查看监控报警
- 支持一键升级集群版本
- 支持日志管理，包括服务日志和慢查询日志等
- 支持通过集群ID、名称、标签等参数搜索实例

10.14.1.3.3 数据备份与恢复

- 支持设置自动备份策略
- 支持一键手工备份
- 支持一键删除备份记录
- 支持从备份记录恢复至新集群

10.14.1.3.4 数据导入与导出

- 支持从CSV 文件导入数据到TiDB 集群
- 支持从 SQL 文件导入数据到TiDB 集群
- 支持从TiDB 集群导出数据到CSV 文件
- 支持从TiDB 集群导出数据库到 SQL 文件

10.14.1.3.5 数据同步

- 支持从TiDB 集群向下游（Kafaka、 MySQL、 TiDB ）数据库同步数据

> 注意：
> 数据同步功能仅支持TiDB 5.2.2 及以上版本的集群。

10.14.1.3.6 克隆集群

- 支持克隆一个已有的TiDB 集群，并创建一个新的TiDB 集群
- 支持同步克隆和快照克隆两种数据策略

> 注意：
> 同步克隆功能仅支持TiDB 5.2.2 及以上版本的集群。

10.14.1.3.7 集群参数管理

- 提供TiDB 版本对应的参数组模板。关于支持的TiDB 版本，参见TiUniManager 支持的TiDB 版本。
- 提供统一界面查看、修改TiDB 集群参数

10.14.1.3.8 工作流任务管理

- 支持查看从TiUniManager 发起的所有工作流任务记录列表
- 支持查看工作流任务详情

10.14.1.3.9 系统管理

- 支持TiUniManager 系统监控和TiUniManager 主机监控
- 支持查看TiUniManager 系统日志
- 支持查看TiUniManager 微服务Trace 信息

10.14.2 TiUniManager 安装和运维指南

本文档介绍TiUniManager 的安装部署流程以及运维方法，指导数据库管理员完成TiUniManager 的部署和配置。

10.14.2.1 软硬件环境配置

本节介绍部署TiUniManager 前需要满足的软硬件环境配置要求。

10.14.2.1.1 Linux 操作系统版本

要部署和运行TiUniManager 服务，确保 Linux 操作系统的版本满足以下要求：

Linux 操作系统	版本
Red Hat Enterprise Linux	7.3 及以上的 7.x 版本
CentOS	7.3 及以上的 7.x 版本

10.14.2.1.2 中控机软件配置

TiUniManager 中控机是运行 TiUniManager 服务的中央控制节点。你可通过登录 TiUniManager 中控机上的 Web console 或 OpenAPI 完成对TiDB 集群的日常管理。中控机的软件配置要求如下：

软件	版本
sshpass	1.06 及以上
TiUP	1.9.0 及以上

10.14.2.1.3 服务器配置

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

开发及测试环境

在开发及测试环境中，对服务器硬件配置要求如下：

组件	CPU	内存	硬盘类型	网络	实例数量（最低要求）
TiUniManager	8 核 +	16 GB+	SAS	万兆网卡（2 块最佳）	1

生产环境

在生产环境中，对服务器硬件配置要求如下：

组件	CPU	内存	硬盘类型	网络	实例数量（最低要求）
TiUniManager	48 核 +	128 GB+	SAS/SSD	万兆网卡（2 块最佳）	1

10.14.2.1.4 网络要求

TiUniManager 正常运行需要网络环境提供如下端口配置，管理员可根据实际环境中TiUniManager 组件部署的方案，在网络侧和主机侧开放相关端口：

组件	默认端口	说明
Web server	4180 或 4183	HTTP 端口：4180HTTPS 端口：4183
OpenAPI server	4100 或 4103	OpenAPI 服务端口: 4100OpenAPI 监控端口：4103
Cluster server	4101 或 4104	集群服务端口
File server	4102 或 4105	文件上传或下载的服务器端口
etcd	4106 或 4107	etcd 服务端口
Elasticsearch server	4108	Elasticsearch 服务端口
Kibana server	4109	Kibana 服务端口
Prometheus	4110	Prometheus 服务端口
Grafana server	4111	Grafana 服务端口
Alertmanager server	4112 或 4113	告警管理服务端口
Jaeger(tracer server)	4114 到 4123	Jaeger 服务端口
node_exporter	4124	TiUniManager 主机系统上报信息的通信端口

10.14.2.2 客户端Web 浏览器要求

你可在较新版本的常见桌面浏览器中使用TiUniManager，浏览器的版本要求如下：

- Chrome > 79
- Firefox > 72
- Microsoft Edge > 79
- Safari > 14

> **注意：**
> 若使用旧版本浏览器或其他浏览器访问TiUniManager，部分界面可能无法正常工作。

10.14.2.3 拓扑模板

在线部署TiUniManager 前，你需要准备好YAML 拓扑文件。TiUniManager 离线包中包含YAML 拓扑文件模板。本节介绍用于部署TiUniManager 拓扑配置模版。见TiUniManager 拓扑配置模版config.yaml（单机版）。

如果TiUniManager 中控机通过用户名密钥访问TiDB 资源机，需要参照指定TiUniManager 中控机登录TiDB 资源机的用户名和密钥，在配置文件 `config.yaml` 中指定用户名和密钥。

10.14.2.4 离线部署TiUniManager

本节介绍如何在离线环境部署TiUniManager。当前TiUniManager 只支持通过离线部署。

1. 通过 `https://download.pingcap.org/em-enterprise-server-${version}-linux-amd64.tar.gz` 下载 TiUniManager 离线安装包。

   下载链接中的 `${version}` 为TiUniManager 的版本号。例如，`v1.0.2` 版本的下载链接为 `https://download.pingcap.org/em-enterprise-server-v1.0.2-linux-amd64.tar.gz`。在下载时，你需要将链接中的 `${version}` 替换为目标版本号。

2. 发送TiUniManager 离线安装包至TiUniManager 中控机。

   离线安装包放置于TiUniManager 中控机，使用具有 sudo 权限的账号执行后续操作。

3. 解压TiUniManager 离线包。

```
tar xzvf em-enterprise-server-${version}-linux-amd64.tar.gz
```

4. 安装TiUniManager。进入解压后的目录，执行 `install.sh` 脚本。

```
sudo sh em-enterprise-server-${version}-linux-amd64/install.sh <TiUniManager 中控机 IP>
```

5. 声明环境变量。

```
# 切换到 tidb 账号下
su - tidb

# 声明环境变量，使 tiup 工具生效
source /home/tidb/.bash_profile
```

6. 生成tidb 帐户下的密钥。

```
# 切换到 tidb 账号下
su - tidb

# 生成 rsa 密钥
ssh-keygen -t rsa

# 复制密钥到 tiup_rsa
cp /home/tidb/.ssh/id_rsa /home/tidb/.ssh/tiup_rsa
```

7. 编辑拓扑配置文件。根据实际环境，你可编辑位于 /home/tidb/ 下的拓扑配置文件 config.yaml。

8. 导入TiDB Server 离线镜像包。

   离线环境下，需要在TiUniManager 中控机本地目录上导入TiDB 离线镜像包，否则无法通过TiUniManager 中控机完成对TiDB 集群的日常管理。

```
# 切换到 tidb 账号下
su - tidb

# 下载解压 TiDB Server 离线镜像包，将 ${version} 手动替换为实际的 TiDB 版本号。

wget https://download.pingcap.org/tidb-community-server-${version}-linux-amd64.tar.gz
tar xzvf tidb-community-server-${version}-linux-amd64.tar.gz

# 导入离线镜像包
TIUP_HOME=/home/tidb/.tiup tiup mirror merge tidb-community-server-${version}-linux-amd64
```

   在 TiUniManager 中控机上查看本地TiDB 镜像源。

```
# 切换到 tidb 账号下
su - tidb

# 查看 TiDB 镜像源地址
tiup mirror show
```

9. 执行命令部署TiUniManager。

```
# 切换到 tidb 账号下
su - tidb

# 部署名称为 "em-test" 的 TiUniManager，注意这里的版本号不带 v，比如 v1.0.0 的版本号，
    ↪ 正确的输入是 1.0.0
TIUP_HOME=/home/tidb/.em tiup em deploy em-test <版本号> config.yaml -u <具有sudo权限的账号>
    ↪  -p

# 启动 TiUniManager
TIUP_HOME=/home/tidb/.em tiup em start em-test
```

10.14.2.5 指定TiUniManager中控机登录TiDB资源机的帐户和密钥

默认情况下，你可以通过TiUniManager Web控制台资源管理>导入主机页面提供的主机资源模板填写TiDB资源机帐户和密码。

如果用户环境中不允许账号密码方式登录TiDB资源机，可在 `config.yaml` 文件中进行配置，以帐户密钥方式登录TiDB资源机。`config.yaml` 中相关配置项如下：

配置描述	配置参数名	示例值
login-host-user	登录TiDB资源机的用户名	tidb
login-private-key-path	位于TiUniManager中控机上登录TiDB资源机使用的私钥路径	/home/tidb/.ssh/id_rsa
login-public-key-path	位于TiUniManager中控机登录TiDB资源机使用的公钥路径	/home/tidb/.ssh/id_rsa.pub

10.14.2.6 升级TiUniManager（适用于 v1.0.1 及以上版本）

本节适用于安装过TiUniManager v1.0.1 及以上版本，并打算升级TiUniManager 的用户。如首次安装可跳过本节。

> 注意：
>
> - 本节介绍的方法仅适用于从 TiUniManager v1.0.1 升级至 v1.0.2 及以上版本，不适用于从 v1.0.0 升级至 v1.0.1 及以上版本。
> - 如果你已安装TiUniManager v1.0.0 版本，想要升级至 v1.0.1 及以上版本，则需要先参考升级TiUniManager v1.0.0 至 v1.0.1 进行升级，再参考本节内容从 v1.0.1 升级至较新的目标版本。
> - 在 v1.0.0 和 v1.0.1 中，TiUniManager 原名为 TiDB Enterprise Manager。自 v1.0.2 起正式更名为 TiUniManager。

1. 升级TiUniManager 离线包。

   参考离线部署 TiUniManager 中第 1-3 步：通过 TiUniManager 产品团队获得最新 TiUniManager 离线包，发送 TiUniManager 离线安装包至目标集群的 TiUniManager 中控机，解压离线包。然后执行下面命令更新 TiUniManager 离线包。

   ```
   # user 为之前部署 TiUniManager 的帐户，默认为 tidb。${version} 为 TiUniManager 的版本号，
       ↪ 需要手动替换为实际的版本号。
   sudo sh em-enterprise-server-${version}-linux-amd64/update.sh <user>
   ```

2. 执行升级TiUniManager 命令。

   ```
   # 切换到 tidb 账号下
   su - tidb

    # 将集群 <cluster-name> 升级到特定版本 <version>
   # 目前仅支持原地停机升级，并且暂不支持版本降级和回退。为安全起见，请按照 停止集群 > 备份
       ↪ tiunimanager 元数据 > 升级集群的流程操作
   TIUP_HOME=/home/tidb/.em tiup em upgrade <cluster-name> <version>
   ```

10.14.2.7 升级TiUniManager v1.0.0 至 v1.0.1

本节介绍的方法适用于从v1.0.0 升级至v1.0.1。如需升级至更高版本，请先升级至v1.0.1，再参考升级TiUniManager（适用于 v1.0.1 及以上版本）进行升级。如首次安装可跳过本节。

> 注意：
>
> 在 v1.0.0 和 v1.0.1 中，TiUniManager 原名为TiDB Enterprise Manager。自 v1.0.2 起正式更名为TiUniManager。

1. 升级TiUniManager 离线包。

   参考离线部署 TiUniManager 中第 1-3 步：通过 TiUniManager 产品团队获得最新 TiUniManager 离线包，发送 TiUniManager 离线安装包至目标集群的 TiUniManager 中控机，解压离线包。然后执行下面命令更新 TiUniManager 离线包。

```
# user 为之前部署 TiUniManager 的帐户，默认为 tidb
sudo sh em-enterprise-server-v1.0.1-linux-amd64/update.sh <user>
```

2. 做升级TiUniManager 之前的准备。

   执行以下命令为TiUniManager 升级做准备。以下命令会自动备份TiUniManager v1.0.0 中的数据。请注意，执行完此命令后直至升级完成，请不要再通过TiUniManager 界面做其它操作，以避免备份未包含最新数据导致升级后数据丢失的情况。

```
# user 为之前部署 TiUniManager v1.0.0 的帐户，默认为 tidb
# cluster-name 为当前部署的 TiUniManager 名称，默认为 em-test
sudo sh em-enterprise-server-v1.0.1-linux-amd64/prepare_upgrade.sh <user> <TiUniManager
    ↪ 中控机 IP> <cluster-name>
```

3. 检查新的TiUniManager 部署拓扑文件。

   如果你在部署TiUniManager v1.0.0 时曾对 config.yaml 文件做了自定义修改，请参考下列子步骤查看 v1.0.0 的配置文件 config.yaml 和待部署 v1.0.1 配置文件 config-v1.0.1.yaml 的区别，并做相应修改。

   1. 切换至 tidb 账号。

```
su - tidb
```

   2. 查看配置变更。

      对比 config.yaml 和 config-v1.0.1.yaml 生效内容的差异。如果差异仅为 db_path 所在行，则可跳过本步骤，无须执行后续子步骤（备份 db_path 以及更新 config-v1.0.1.yaml），可直接进入删除 TiUniManager v1.0.0 的步骤。如果差异不仅有 db_path 所在行，还存在其它差异，则需要继续执行后续子步骤（备份 db_path 以及更新 config-v1.0.1.yaml）。

```
diff config.yaml config-v1.0.1.yaml
```

   3. 记下 config-v1.0.1.yaml 中的数据备份路径 db_path。

   db_path 为 TiUniManager 备份数据的路径。在“升级TiUniManager 之前的准备”这一步中TiUniManager 已经自动在 config-v1.0.1.yaml 中写好了默认的 v1.0.1 数据备份路径，此前已备份的 v1.0.0 数据将恢复到该路径下。

```
查看 `config-v1.0.1.yaml` 中的 `db_path`，记下该行内容，在以下子步骤 5 中更新 `config-v1
    ↪ .0.1.yaml` 时需要用到该行内容。示例如下：

```yaml
db_path:  "/home/tidb/em-backup-20220511-154415/em.db"
```
```

4. 覆盖 config-v1.0.1.yaml。

将TiUniManager v1.0.0 的 config.yaml 文件复制一份并覆盖 v1.0.1 的 config-v1.0.1.yaml。

```shell
cp config.yaml config-v1.0.1.yaml
```

5. 更新 config-v1.0.1.yaml。
   覆盖配置文件后，将子步骤 3 中拷贝的 db_path 填写至 config-v1.0.1.yaml，注意空格的缩进。示例如下：

```
em_cluster_servers:
  - host: 127.0.0.1
    db_path:  "/home/tidb/em-backup-20220511-154415/em.db"
```

4. 删除TiUniManager v1.0.0。

```
# 切换到 tidb 账号下
su - tidb

# 查看现有 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em list

# 删除指定的 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em destroy <cluster-name>
```

5. 部署TiUniManager v1.0.1。

```
# 切换到 tidb 账号下
su - tidb

# 更新 em 工具
TIUP_HOME=/home/tidb/.em tiup update em
TIUP_HOME=/home/tidb/.em tiup update --all

# 部署名称为 "em-test" 的 TiUniManager
TIUP_HOME=/home/tidb/.em tiup em deploy em-test 1.0.1 config-v1.0.1.yaml -u <具有sudo
    ↪ 权限的账号> -p

# 启动 TiUniManager
TIUP_HOME=/home/tidb/.em tiup em start em-test

# 将 db_path 注释掉。如果不注释，后续对 em 进行重启、扩容等操作时 TiUniManager
    ↪ 可能会将数据变更还原
sed -i "s/  db_path:/# db_path:/" /home/tidb/.em/storage/em/clusters/em-test/meta.yaml
```

6. 重建资源机器的 filebeat。

```
# 切换到 tidb 账号下
su - tidb

# 重建资源机器 filebeat
TIUP_HOME=/home/tidb/.em tiup em scale-out em-test /home/tidb/em-v1.0.0-backup/resource-
    ↪ filebeat-scale-out-topology.yaml --user tidb -i /home/tidb/.ssh/tiup_rsa --wait-
    ↪ timeout 360 --yes
```

7. 使集群日志功能生效。

   待所有上述步骤完成后，大部分的升级已完成。由于在升级过程中，TiUniManager 对资源机器的 filebeat 进行了重装。为了使得查看TiDB 集群日志功能在TiUniManager 中生效，你需要手动在TiUniManager 界面操作重启TiDB 集群以刷新配置。为避免影响用户服务，请尽量在业务低峰期操作。

10.14.2.8 备份恢复TiUniManager 元信息

为提高 TiUniManager 可用性，应对容灾场景，你可对 TiUniManager 元信息进行备份，并恢复元信息到新的 TiUniManager 中控机。

1. 备份TiUniManager 元信息。

```
# 切换到 tidb 账号下
su - tidb

# 将当前 TiUniManager 系统 <cluster-name> 的元数据备份到中控机的 <target-path>
TIUP_HOME=/home/tidb/.em tiup em backup <cluster-name> <target-path> -N <tiunimanager-ip>
```

2. 恢复TiUniManager 元信息。

```
# 切换到 tidb 账号下
su - tidb

db_path: "/home/tidb/em.db"
```

   从备份的元数据中恢复到新集群， 流程和部署新集群相同。唯一的区别是在集群 yaml 配置中，`em_cluster_servers` 里增加了 `db_path: "/home/tidb/em.db"`，详细见 TiUniManager 根据元数据恢复新集群拓扑配置模版 em.yaml（单机版）。

10.14.2.9 修改默认的集群备份路径

TiUniManager 默认的集群备份路径相关配置参数如下：

配置描述	配置参数名	参考值
TiDB 集群备份的存储类型（仅支持 NFS 或 S3）	BackupStorageType	‘s3’ 或 ‘nfs’
TiDB 集群备份的存储路径（S3 bucket 路径，或 NFS share 的绝对路径）	BackupStoragePath	‘s3://bucket-name/path/backup’（S3 路径示例）‘/mnt/nfs-path’（NFS 路径示例）
TiDB 集群备份在 S3 共享存储时，S3 的 AccessKey	BackupS3AccessKey	

配置描述	配置参数名	参考值
TiDB 集群备份在 S3 共享存储时，S3 的 SecretAccessKey	BackupS3SecretAccessKey	
TiDB 集群备份在 S3 共享存储时，S3 的 Endpoint（域名）	BackupS3Endpoint	

当前不支持通过TiUniManager 界面修改备份路径。如需修改备份路径，需要通过 OpenAPI 修改配置参数，以修改配置参数“BackupS3AccessKey” 为例：

1. 登录获取user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=BackupS3AccessKey" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值 BackupS3AccessKey 的值为 newValue。你可以将 newValue 替换为你期待的备份路径。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ BackupS3AccessKey\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer 6
    ↪ ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

10.14.2.10 修改默认的数据导入导出路径

TiUniManager 默认的导入导出路径在TiUniManager 中控机上，细节如下：

配置描述	配置参数名	参考值
TiUniManager 共享存储中导入文件存储位置（建议配置为 NFS 共享存储）	ImportShareStoragePath	/ ↪ home ↪ / ↪ tidb ↪ / ↪ import ↪ （备注：中控机 tidb 账号拥有该路径的读写权限）

配置描述	配置参数名	参考值
共享存储中导出文件存储的位置（建议配置为 NFS 共享存储）	TiUniManaEgxeprortSha/reStoragePath	′→ home ′→ / ′→ tidb ′→ / ′→ export ′→ （备注：中控机 tidb 账号拥有该路径的读写权限）

当前TiUniManager 不支持通过TiUniManager 界面修改导入导出的路径。如需修导入导出路径，需要通过 OpenAPI 对配置进行修改，以修改“ImportShareStoragePath”为例：

1. 登录获取user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ′→ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ′→  "admin" }'
```

> **注意：**
> 你需要将以上命令中 `172.16.6.206:4180` 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=ImportShareStoragePath" -H "
    ′→ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数 `ImportShareStoragePath` 的值为 `newValue`，你可以将 `newValue` 替换为你期待的导入导出路径。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ ImportShareStoragePath\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

10.14.2.11 删除TiUniManager

如要删除TiUniManager 服务，请执行以下命令：

```
su - tidb

### 查看现有 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em list

### 删除指定的 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em destroy <cluster-name>
```

10.14.2.12 指定外部的Elasticsearch 服务

Elasticsearch 是一个分布式、高扩展、高实时的搜索与数据分析引擎。它能很方便的使大量数据具有搜索、分析和探索的能力。目前TiUniManager 主要依赖Elasticsearch 来做日志数据的存储和检索。

TiUniManager 可以通过 config.yaml 文件中全局的 external_elasticsearch_url 参数来指定外部的Elasticsearch 服务。默认值为空。

external_elasticsearch_url 参数值格式为 <IP:Port>，例如：127.0.0.1:9200。

在 config.yaml 文件中，external_elasticsearch_url 和 elasticsearch_servers 都是用来指定 Elasticsearch 部署信息的， 不能同时指定。 如果指定了 external_elasticsearch_url 的值， 则需要注释掉 elasticsearch_servers 组件的配置信息注释。

配置格式参考TiUniManager 拓扑配置模版config.yaml（单机版）。

10.14.2.13 手动指定部署的Elasticsearch 组件堆内存大小

部署TiUniManager 的拓扑文件支持根据真实部署中控机资源大小指定Elasticsearch 组件的堆内存大小。通过 elasticsearch_servers 下的 heap_size 参数进行指定。例如 16g。

heap_size 是选填参数，不手工指定会使用默认值。默认值为 4g。

配置格式参考TiUniManager 拓扑配置模版config.yaml（单机版）。

10.14.2.14 安装 Kibana 组件（可选）

Kibana 是一个针对Elasticsearch 的开源分析及可视化平台，用来搜索、查看交互存储在Elasticsearch 索引中的数据。使用 Kibana，可以通过各种图表进行高级数据分析及展示。

目前TiUniManager 默认安装 Kibana，集成 Kibana 主要是做日志的展示，在控制台页面系统管理下的系统日志进行查看。

如果不想安装 kibana 组件，则直接在 config.yaml 文件中将 kibana_servers 组件相关的配置注释掉即可。

10.14.2.15 安全加固

完成 TiUniManager 安装后，建议用户通过系统防火墙为 Elasticsearch 和 Kibana 添加网络访问控制，仅允许 TiUniManager 中控机与上述组件通信，参考命令如下所示：

```
### 允许来自 TiUniManager 中控机的连接
iptables -A INPUT -p tcp -s <tiunimanager addr> --dport <elasticsearch port> -j ACCEPT
### 拒绝其它 IP 的访问
iptables -A INPUT -p tcp --dport <elasticsearch port> -j DROP
```

10.14.3 TiUniManager 快速操作指南

本文档介绍TiUniManager 的常见操作场景以及基本的集群操作，包括将主机节点加入TiDB 集群、创建TiDB 集群、删除集群、接管集群。

在阅读本文档前，你需要先阅读TiUniManager 安装和运维手册完成TiUniManager 的安装。

10.14.3.1 将主机节点加入TiDB 集群

系统初始化或者扩容机器时，系统管理员需将主机信息导入TiUniManager 平台，TiUniManager 将导入的机器加入到集群中，由平台统一管理。

10.14.3.1.1 前置条件

将主机节点加入TiDB 集群前，需要保证：

- 已经登录TiUniManager 控制台
- 已安装主机的操作系统和所依赖的软件，并通过测试

> 注意：
>
> - 你需要按照主机模板完整、正确地填写字段信息。详情参见本节中主机模板字段说明。
> - 导入主机时，TiDB 会对主机进行检查，参见TiDB 环境与系统配置检查。

TiUniManager 中控机通过 SSH 连接主机，默认连接端口为 22。如果环境中 SSH 端口不为默认的 22，可通过 OpenAPI 修改 config_default_ssh_port 参数来配置主机的默认登陆端口，该参数默认值为 22。以下示例通过 OpenAPI 修改 config_default_ssh_port 参数的值，从而修改主机的默认登陆端口：

1. 登录TiUniManager 获取用户Token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪ "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看 config_default_ssh_port 的参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=config_default_ssh_port" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 将 config_default_ssh_port 的值修改为 newValue。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ config_default_ssh_port\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪  6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

10.14.3.1.2 操作步骤

1. 进入资源管理页面。
2. 点击导入主机按钮。
3. 点击下载主机模板按钮。
4. 双击打开模板，并按照模板填入相应信息并保存。
5. 点击上传按钮，选择步骤 5 中编辑的文件，并上传该文件。
6. 点击确认按钮，确认导入主机信息。

10.14.3.1.3 主机模板字段说明

Hostname	主机名
IP	主机IP 地址
Login username	主机用户名（选填，当通过用户名密码方式导入主机时，才需要填写）
Login password	主机密码（选填，当通过用户名密码方式导入主机时，才需要填写）
Region	区域 ID (产品初始化时设定的 Region ID）
Zone	可用区ID (产品初始化时设定的 Zone ID）
Rack	机架ID （产品初始化时设定的Rack ID）
Arch	体系架构
OS Version	操作系统版本
Kernel	内核版本
vCPU	CPU 核数
Memory	内存大小
NIC	网卡规格
Cluster Purpose	区分主机用来部署什么类型的集群，包括TiDB、 DM、 TiUniManager
Host Purpose	主机用途：用来区分部署的组件类型，包括：Compute、 Storage、 Schedule 三种用途。多种用途以逗号“,”连接，例如‘Compute,Storage,Schedule’

Hostname	主机名
Disk Type	磁盘类型，包括：NVMe SSD、 SSD、 SATA 三种类型
Disks	磁盘信息，包括磁盘名称、容量、状态、路径，示例：{“name”：“sda”,“capacity”:256,“status”：“Available”,“path”：“/mnt/sda”}

如果通过用户名密钥方式导入主机，需要在TiUniManager 安装前配置用户名和密钥路径，参见指定TiUniManager 中控机登录TiDB 资源机的账户和密钥。

10.14.3.2 创建集群

TiUniManager 部署完成后，你可以通过TiUniManager 创建TiDB 集群，并自定义集群配置。

10.14.3.2.1 前置条件

- 已登录TiUniManager 控制台
- 已完成主机资源导入

10.14.3.2.2 操作步骤

1. 进入集群管理> 集群页面。
2. 点击创建集群按钮，跳转至创建实例的页面。
3. 选择集群创建模式。
4. 选择集群主机所在厂商、区域。
5. 输入以下数据库基本信息：
    - 数据库类型
    - CPU 体系架构
    - 数据库版本
    - 参数组
6. 设置数据库产品各组件的以下配置：
    - 所在可用区
    - 实例规格
    - 实例数量
7. 输入集群的以下基本信息：
    - 集群名称。集群名称必须是 4 - 64 个字符，可包含大小写字母、数字和连字符，并以字母或数字开头。
    - 集群标签
    - 数据库管理员Root 的密码。密码必须是 8-64 个字符，可包含大小写字母、数字和可见的特殊字符（包括 !@#$%^&*()_+=）
    - 是否独占部署
8. 点击提交按钮，确认主机资源库存满足集群要求后，点击确认创建按钮。

10.14.3.3 删除集群

由于业务或其他原因，如果你不再需要某个已创建的实例，可以将此集群删除。

10.14.3.3.1 前置条件

- 已登录TiUniManager 控制台。
- 待删除的TiDB 集群已存在。

> **注意：**
>
> 删除集群同时将删除集群上自动备份的数据。如果需要某个备份数据，应在删除实例之前将该备份恢复到新实例上，数据恢复操作可参见备份管理- 数据恢复。

10.14.3.3.2 操作步骤

1. 进入集群管理> 集群页面。
2. 选择待删除的集群，点击集群ID 进入集群详情页面。
3. 点击删除按钮。
4. 选择是否在删除集群前完成一次数据备份。
5. 选择是否保留手动备份数据。
6. 输入“delete”以确认删除，点击确认删除按钮。

10.14.3.4 接管集群

通过TiUniManager 接管已经存在的TiDB 集群，统一管理新老集群，提高集群管理效率。

10.14.3.4.1 前置条件

已经登录TiUniManager 控制台。

10.14.3.4.2 操作步骤

1. 进入集群管理> 集群页面。
2. 点击接管集群按钮，进入接管集群页面。
3. 输入接管集群的基本信息：集群名称、数据库用户名root、数据库密码。
4. 输入接管集群中控配配置信息：
   - 接管集群中控机主机IP 地址
   - 接管集群中控机 SSH 端口号
   - 接管集群中控机 SSH 用户名
   - 接管集群中控机 SSH 密码
   - 接管集群中控机TiUP 路径（即 `.tiup` 目录所在路径，不含路径结尾的`/`，例如 `/root/.tiup` 不能填写为 `/root/.tiup/` ）
5. 导入接管集群的主机（导入主机流程参见将主机节点加入TiDB 集群 ）。

10.14.4 操作指南

10.14.4.1 TiUniManager 登录与初始化

本文档介绍如何登录TiUniManager、进行产品初始化，以及如何退出登录。

10.14.4.1.1 登录TiUniManager 控制台

使用TiUniManager 进行操作前，你需要先登录TiUniManager 控制台。

要登录TiUniManager 控制台，你需要先参考TiUniManager 安装手册开启TiUniManager 服务。

> 注意：
> 当前TiUniManager 暂不支持找回密码。

按照以下步骤进行登录：

1. 在浏览器中输入TiUniManager 地址，跳转至登录页面。
2. 在登录页面输入用户名和密码。默认用户名为admin，密码为admin。
3. 点击登录按钮进行登录并跳转至概览页面。

10.14.4.1.2 退出登录

已登录TiUniManager 控制台的用户可以退出登录，返回登录页面。

> 注意：
> 退出登录后，TiUniManager 会清除缓存数据。如果再次登录同一用户，之前未保存的编辑信息会丢失。

操作步骤：点击用户图标，出现退出登录按钮，点击该按钮可退出登录并返回登录页面。

10.14.4.1.3 产品初始化

首次登录TiUniManager 系统后，需要完成对TiDB 所在数据中心、TiDB 产品组件、TiDB 版本信息的初始化。

数据中心当前默认为本地数据中心(Local)，本地数据中心下按照区域 (Region) - 可用区(Available Zone) 2 层层级来组织主机资源。

以下为主机资源组织结构示例：

```
- 本地数据中心（Local）
    - 区域 Region 1
        - 可用区 Zone1_1
```

```
      - 可用区 Zone1_2
  - 区域 Region 2
      - 可用区 Zone2_1
      - 可用区 Zone2_2
```

操作步骤

1. 登录TiUniManager 控制台。
2. 进入数据中心初始化页面
    1. 自定义填写本地数据中心的厂商名称。
    2. 自定义填写区域ID、区域名称。
    3. 自定义填写可用区ID、可用区名称。
    4. 自定义填写TiDB、PD、TiKV、TiFlash 实例规格。
3. 点击下一步进入产品组件初始化页面。
    5. 自定义填写组件名称。
    6. 设置组件端口范围。
4. 点击下一步进入产品版本初始化页面。选择希望安装的TiDB 版本和架构。
5. 点击完成按钮。

10.14.4.2 TiUniManager 主机资源管理

本文档介绍如何通过TiUniManager 管理主机资源，包括导入、查看、删除主机资源。

10.14.4.2.1 导入主机

系统初始化或者扩容机器时，系统管理员需将机器信息导入TiUniManager 平台，TiUniManager 将导入的机器加入到集群中，由平台统一管理。

前置条件

将主机节点加入TiDB 集群前，需要保证：

- 已经登录TiUniManager 控制台
- 已安装主机的操作系统和所依赖的软件，并通过测试

> 注意：
>
> - 你需要按照主机模板完整、正确地填写字段信息。详情参见本节中主机模板字段说明。
> - 导入主机时，TiDB 会对主机进行检查，参见TiDB 环境与系统配置检查。

TiUniManager 中控机通过 SSH 连接主机，默认连接端口为 22。如果环境中 SSH 端口不为默认的 22，可通过 OpenAPI 修改 config_default_ssh_port 参数来配置主机的默认登陆端口，该参数默认值为 22。以下示例通过 OpenAPI 修改 config_default_ssh_port 参数的值，从而修改主机的默认登陆端口：

1. 登录TiUniManager 获取用户Token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    '→ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    '→ "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看 config_default_ssh_port 的参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=config_default_ssh_port" -H "
    '→ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 将 config_default_ssh_port 的值修改为 newValue。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    '→ config_default_ssh_port\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    '→ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

操作步骤

1. 进入资源管理页面。
2. 点击导入主机按钮。
3. 点击下载主机模板按钮。
4. 双击打开模板，并按照模板填入相应信息并保存。
5. 点击上传按钮，选择步骤 5 中编辑的文件，并上传该文件。
6. 点击确认按钮，确认导入主机信息。

主机模板字段说明

Hostname	主机名
IP	主机IP 地址
Login username	主机用户名（选填，当通过用户名密码方式导入主机时，才需要填写）
Login password	主机密码（选填，当通过用户名密码方式导入主机时，才需要填写）
Region	区域 ID (产品初始化时设定的 Region ID）
Zone	可用区ID (产品初始化时设定的 Zone ID）
Rack	机架ID（产品初始化时设定的Rack ID）
Arch	体系架构
OS Version	操作系统版本
Kernel	内核版本
vCPU	CPU 核数
Memory	内存大小

Hostname	主机名
NIC	网卡规格
Cluster Purpose	区分主机用来部署什么类型的集群，包括TiDB、 DM、 TiUniManager
Host Purpose	主机用途：用来区分部署的组件类型，包括：Compute、 Storage、 Schedule 三种用途。多种用途以逗号“,”连接，例如‘Compute,Storage,Schedule’
Disk Type	磁盘类型，包括：NVMe SSD、 SSD、 SATA 三种类型
Disks	磁盘信息，包括磁盘名称、容量、状态、路径，示例：{“name”：“sda”，“capacity”：256，“status”：“Available”，“path”：“/mnt/sda”}

如果通过用户名密钥方式导入主机，需要在TiUniManager 安装前配置用户名和密钥路径，参见指定TiUniManager 中控机登录TiDB 资源机的账户和密钥。

10.14.4.2.2 查看主机列表

查看TiUniManager 当前管理的所有主机信息列表。

操作步骤如下：

1. 登录控制台。
2. 进入主机管理页面，在该页面即可看到主机列表。

10.14.4.2.3 删除主机

从 TiUniManager 主机资源池中删除主机。

操作步骤如下：

1. 登录控制台。
2. 进入资源管理页面。
3. 点击主机条目的删除。
4. 输入“delete”以确认删除操作。
5. 点击确认删除按钮。

10.14.4.3 TiUniManager 集群管理

本文档介绍如何通过TiUniManager 对TiDB 集群进行操作和管理。

10.14.4.3.1 创建集群

TiUniManager 部署完成后，你可以通过TiUniManager 创建TiDB 集群，并自定义集群配置。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager 控制台
- 已完成主机资源导入

创建集群的操作步骤如下：

1. 登录控制台。
2. 进入集群管理> 集群页面。
3. 点击创建集群按钮，跳转至创建实例的页面。
4. 选择集群创建模式。
5. 选择集群主机所在厂商、区域。
6. 输入以下数据库基本信息：
    - 数据库类型
    - CPU 体系架构
    - 数据库版本
    - 参数组
7. 设置数据库产品各组件的以下配置：
    - 所在可用区
    - 实例规格
    - 实例数量
8. 输入集群的以下基本信息：
    - 集群名称。集群名称必须是 4-64 个字符，可包含大小写字母、数字和连字符，并以字母或数字开头
    - 集群标签
    - 数据库管理员Root 的密码。密码必须是 8-64 个字符，可包含大小写字母、数字和可见的特殊字符（包括 `!@#$%^&*()_+=` ）
    - 是否独占部署
9. 点击提交按钮，确认主机资源库存满足集群要求后，点击确认创建按钮。

10.14.4.3.2 删除集群

由于业务或其他原因，如果你不再需要某个已创建的实例，可以将某个集群删除。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager 控制台。
- 待删除的TiDB 集群已存在。

> 注意：
>
> 删除集群同时将删除集群上自动备份的数据。如果需要某个备份数据，应在删除实例之前将该备份恢复到新实例上。

删除集群的操作步骤如下：

1. 登录控制台。
2. 进入集群管理> 集群页面。
3. 选择待删除的集群，点击集群ID 进入集群详情页面。

4. 点击删除按钮。
5. 选择是否在删除集群前完成一次数据备份。
6. 选择是否保留手动备份数据。
7. 输入“delete”以确认删除，点击确认删除按钮。

10.14.4.3.3 扩容集群

集群运行过程中，如果由于业务或其他需求，你需要为集群添加计算引擎、存储引擎或调度引擎。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager 控制台。
- 集群已创建。

> 注意：
>
> - 集群配置由平台配置所限定。
> - 推荐的 PD 实例数量为 1 个、3 个、5 个或 7 个。
> - TiKV 实例数量不能低于TiKV 副本数。

扩容的操作步骤如下：

1. 登录控制台。
2. 进入集群管理> 集群页面。
3. 选择待扩容的集群，点击集群ID 进入集群详情页面。
4. 点击扩容按钮进入扩容页面。
5. 根据业务需要，选择要扩容引擎数量或新增不同规格的引擎数量。
6. 点击提交按钮，确认资源库存满足扩容要求后，点击确认扩容。

10.14.4.3.4 缩容集群

在集群运行过程中，由于业务或其他需求，你需要从集群中删除多余的计算引擎、存储引擎或调度引擎。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager 控制台。
- 集群已创建。

> 注意：
>
> - 集群配置由平台配置所限定。
> - 当TiDB、 PD、 TiKV 的实例数量最小为 1。当实例数等于 1 时，不能进行缩容操作。

- 推荐的 PD 实例数量为 1 个、3 个、5 个或 7 个。
- TiKV 实例数量不能小于TiKV 副本数。对TiKV 进行缩容时，缩容后的TiKV 实例数量不能小于 TiKV 副本数。

缩容集群的操作步骤如下：

1. 登录控制台。
2. 进入集群管理> 集群页面。
3. 选择待删除的集群，点击集群ID 进入集群详情页面。
4. 点击待删除实例所在行的删除按钮。
5. 点击确认按钮。

10.14.4.3.5 查看集群列表

查看实例的基本信息、运行情况，并使用集群删除按钮按需删除集群。

查看集群列表的操作步骤如下：

1. 登录控制台。
2. 进集群管理> 集群页面即可查看集群列表，若暂无集群，则列表为空。

10.14.4.3.6 搜索集群

在集群列表页面，你根据以下指标搜索集群：

- 集群运行状态（可选择）
- 集群ID（需要输入集群ID，暂不支持模糊搜索）
- 集群名称（需要输入集群名称，暂不支持模糊搜索）
- 标签（需要输入标签，暂不支持模糊搜索）

搜索集群前，确保以下条件已满足：

- 已登录平台控制台。
- 集群已创建。

搜索集群的操作步骤如下：

1. 进入集群详情页面。
2. 选择搜索指标。
3. 点击搜索按钮，确认搜索，集群详情列表展示搜索结果。

10.14.4.3.7 查看集群详情

你可通过TiUniManager查看集群配置、运行情况、计算、存储、调度等各引擎列表及信息详情。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager控制台。
- 集群已创建。

查看集群详情的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID进入集群详情页面。

10.14.4.3.8 性能分析

你可通过TiUniManager查看集群的SQL语句分析、慢SQL查询分析、流量可视化、集群诊断。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager控制台。
- 集群已创建。

性能分析的操作步骤如下：

1. 进入集群管理> 集群页面
2. 点击集群ID进入集群详情页面，再点击性能分析标签页，你可看到SQL语句分析、慢查询日志、流量可视化、诊断四个子标签页。
3. 点击语句分析页签，进入SQL语句分析标签页。了解SQL 语句分析详情。
4. 点击慢查询分析页签，进入慢查询分析标签页。了解慢查询分析详情。
5. 点击流量可视化页签，进入流量可视化标签页。了解流量可视化详情
6. 点击集群诊断页签，进入集群诊断标签页。了解集群诊断详情。

10.14.4.3.9 日志管理

在集群详情页面，你可以查看该集群各组件日志，也可以选择时间范围进行日志查询，或按照日志级别、实例IP地址查看日志。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager控制台。
- 集群已创建。

日志管理的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID进入集群详情页面。

3. 点击日志管理页签，进入日志管理标签页，默认以时间倒序排序展示集群日志，并可按照以下维度进行日志过滤：

    - 按时间范围过滤
    - 按组件（计算、存储、调度）过滤
    - 按日志级别过滤
    - 按日志内容关键词过滤
    - 按实例IP 地址过滤

10.14.4.3.10 集群监控

在TiUniManager 中，你可通过集成的Grafana 面板查看集群整体、组件、节点实例的运行情况（组件分别展示：PD、TiDB、TiKV）。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager 控制台。
- 集群已创建。

查看集群监控的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID 进入集群详情页面，并点击集群监控页签。
3. 在集群监控页签中，默认显示TiDB-Summary Dashboard。
4. 可通过选择 Dashboard 一级目录，列出所有可用 Dashboard。

10.14.4.3.11 告警管理

TiUniManager 集成了TiDB 集群组件的报价规则，按照严重程度由高到低罗列。报警项可分为紧急级别> 严重级别 > 警告级别三类。该分级适用于以下各组件的报警项。

TiDB 集群告警规则可参见在线文档：TiDB 集群报警规则

进行告警管理前，确保以下条件已满足：

- 已登录TiUniManager 控制台。
- 集群已创建。

告警管理的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID 进入集群详情页面，并点击告警管理页签。

10.14.4.3.12 参数管理

你可通过TiUniManager 查看集群实例的参数信息，并修改集群的参数运行值。

进行操作前，确保以下条件已满足：

- 已登录TiUniManager 控制台
- 集群已创建

参数管理的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID 进入集群详情页面，并点击参数管理页签。
3. 选择要修改的参数项，点击编辑进行修改，完成后点击确认。
4. 点击保存按钮，使参数运行值生效。

10.14.4.3.13 备份管理- 数据备份

TiUniManager 提供自动定期备份功能。集群创建成功并开始正常运行后，你可对集群设置定期备份时间，TiUniManager 会对集群数据进行定时备份，备份的数据将保存到你所设置的存储位置中（S3 兼容存储和 NFS 共享存储）。你也可以逐条删除不需要的备份数据文件。

进行操作前，确保以下条件已满足：

- 已经登录TiUniManager 控制台。
- 集群已创建。

> 注意：
>
> - 备份保存的默认文件夹不可修改。

TiUniManager 默认的集群备份路径相关配置参数如下：

配置描述	配置参数名	参考值
TiDB 集群备份的存储类型（仅支持 NFS 或 S3）	BackupStorageType	‘s3’ 或 ‘nfs’
TiDB 集群备份的存储路径（S3 bucket 路径，或 NFS share 的绝对路径）	BackupStoragePath	‘bucketPath/backup’ (S3 路径示例） ‘/mnt/nfspath’ (NFS 路径示例）
TiDB 集群备份在 S3 共享存储时，S3 的 AccessKey	BackupS3AccessKey	”
TiDB 集群备份在 S3 共享存储时，S3 的 SecretAccessKey	BackupS3SecretAccessK”ey	
TiDB 集群备份在 S3 共享存储时,S3 的 Endpoint（域名）	BackupS3Endpoint	”

当前不支持通过TiUniManager 界面修改备份路径。如需修改备份路径，需要通过 OpenAPI 修改配置参数，以修改配置参数 BackupS3AccessKey 为例：

1. 登录TiUniManager获取user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    '→ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    '→  "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=BackupS3AccessKey" -H "
    '→ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    '→ BackupS3AccessKey\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer 6
    '→ ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

数据备份的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID 进入集群详情页面，并点击备份管理页签。
3. 点击备份计划按钮，配置自动备份策略。
4. 点击手动备份按钮，立即开始手动备份。
5. 选择备份并点击删除按钮，确认删除选择的备份。
6. 查询某时间段的备份：
    - 默认以时间倒序排序展示备份
    - 查询某时间段的备份

10.14.4.3.14 备份管理- 数据恢复

使用TiUniManager 的备份数据恢复功能，将一个TiDB 集群的备份数据恢复到一个新集群。

进行操作前，确保以下条件已满足：

- 已经登录控制台。
- 集群的备份记录已存在。

> **注意：**
> 备份数据恢复到新创建的集群上，不会覆盖已创建实例。

数据恢复的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID 进入集群详情页面，并点击备份管理页签。
3. 选择要恢复的备份记录，点击恢复，跳转恢复集群页面。
4. 设置新集群的名称、标签和数据库密码。
5. 选择新集群的Region 区域。
6. 确认新集群的实例配置规格与数量。
7. 点击提交按钮，确认创建新集群并随后开始自动恢复实例。

10.14.4.3.15 数据同步

通过TiUniManager，你可以使用TiDB 增量数据同步功能，从配置了TiCDC 组件的上游TiDB 集群，将增量数据同步到下游系统（ TiDB 数据库、 MySQL 数据库或 Kafka ）。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

> 注意：
> 数据同步功能仅支持TiDB v5.2.2 及其以上版本。

数据同步的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击集群ID 进入集群详情页面，并点击数据同步页签。
3. 点击创建按钮创建数据同步任务。
4. 填写数据同步任务基本信息：
   - 任务名称
   - 同步任务开始TSO 值（ 填入 “0” 则系统将从当前时刻获取起始TSO ）
   - 过滤规则（ 参见表库过滤语法 ）
5. 填写下游信息：
   - 下游类型（ TiDB、 MySQL、 Kafka)
   - 下游 URL 或 IP 地址
   - 下游数据库服务端口
   - 下游数据库用户名
   - 下游数据库密码
   - 同步任务并发数
6. 点击提交按钮。

10.14.4.3.16 查看参数组

TiUniManager v1.0.0 为TiDB v5.1.x、 v5.2.x、 v5.3.x、 v5.4.x 提供内置的默认参数组（暂不支持TiDB v6.0 参数组），你可查看参数组下的参数详情。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

查看参数组步骤如下：

1. 进入集群管理> 参数组页面，查看所有的参数组列表。
2. 点击参数组名称，查看参数组中详情：参数类别、参数名称、参数列表、默认值、参数描述。

10.14.4.3.17 编辑、复制、应用参数组

TiUniManager 为TiDB v5.1.x、 v5.2.x、 v5.3.x、 v5.4.x 提供内置的默认参数组，你可直接应用默认参数组到集群实例，也可复制默认参数组并根据实际运行情况对参数进行调优。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

编辑、复制、应用参数组的操作步骤如下：

1. 进入集群管理> 参数组页面。
2. 点击待修改的参数组对应的编辑操作，对参数组默认值进行修改。
3. 点击待复制的参数组对应的复制操作，将参数组复制产生一份新的参数组。
4. 点击参数组对应的应用操作，可将参数组应用到适配的集群。

10.14.4.3.18 接管集群

你可通过TiUniManager 接管已有的TiDB 集群。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

接管集群的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击接管集群按钮，进入接管集群页面。
3. 输入接管集群的基本信息：集群名称、数据库用户名root、数据库密码。
4. 输入接管集群中控配配置信息：
   - 接管集群中控机主机IP 地址
   - 接管集群中控机 SSH 端口号
   - 接管集群中控机 SSH 用户名
   - 接管集群中控机 SSH 密码
   - 接管集群中控机TiUP 路径（即 `.tiup` 目录所在路径，不含结尾的`/`，例如 `/root/.tiup` ）
5. 导入接管集群的主机（导入主机流程参见导入主机 ）。

10.14.4.3.19 重启集群

你可在TiUniManager 上重启指定的TiDB 集群。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

重启集群的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 点击待重启集群的重启操作。
3. 点击确认按钮开始重启集群。

10.14.4.3.20 停止集群

你可在TiUniManager 上停止指定的TiDB 集群。

进行操作前，确保以下条件已满足：

- 已经登录TiUniManager 控制台。

停止集群的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 选择集群，点击停止。
3. 点击确认按钮，开始停止集群。

10.14.4.3.21 克隆集群

你可通过TiUniManager 从一个TiDB 集群克隆出一个新集群。新集群与源集群之间数量关系可配置为：

- 全量：克隆集群并与源集群建立主备关系，主集群实时向备集群同步数据
- 快照：克隆集群，但不建立主备关系

进行操作前，确保以下条件已满足：

- 已经登录TiUniManager 控制台。
- 已有TiDB 集群。

> 注意：
> 克隆集群功能仅支持TiDB v5.2.2 及其以上版本。

克隆集群的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 选择要产生克隆集群的源集群，进入集群详情页面。
3. 点击克隆集群按钮。
4. 选择克隆策略。
5. 填写新集群配置信息。
6. 点击提交按钮。

10.14.4.3.22 切换主备集群

你可通过TiUniManager 选择一个备集群，并将其切换为主集群，此时集群的原主集群将自动被切换为备集群。

进行操作前，确保以下条件已满足：

- 当前TiDB 集群为备集群。

> 注意：
> 主备集群切换功能仅支持TiDB v5.2.2 及其以上版本。

切换的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 选择备集群，进入备集群详情页面。
3. 点击备集群后的切换。
4. 确认备注集群切换信息后点击确认按钮

10.14.4.3.23 原地升级集群

你可通过TiUniManager 将指定的TiDB 集群升级为更高版本的集群。

进行操作前，确保以下条件已满足：

- 已经登录TiUniManager 控制台。

原地升级集群的操作步骤如下：

1. 进入集群管理> 集群页面。
2. 选择要升级的集群，进入集群详情页面。
3. 点击数据库版本编号后的升级操作，进入集群版本升级页面。
4. 选择目标版本、升级类型、升级方式，点击下一步按钮。
5. 查看对比和合并配置参数，选择参数值为原版本值以及新版本值，点击下一步按钮。
6. 确认合并后的配置参数，点击确认按钮。

10.14.4.3.24 巡检集群

你可通过TiUniManager 巡检功能检查集群各个维度是否健康，包括集群资源分配是否泄露，Region 是否健康，参数配置是否正常等等。

进行操作前，确保以下条件已满足：

- 已经登录TiUniManager 控制台

TiUniManager 于每天 23:00 自动进行集群巡检。

查看巡检结果的方法：在工作流任务管理中查看任务名为“CheckPlatform”的任务流，点击checkHosts 步骤可以查看，如下图所示：

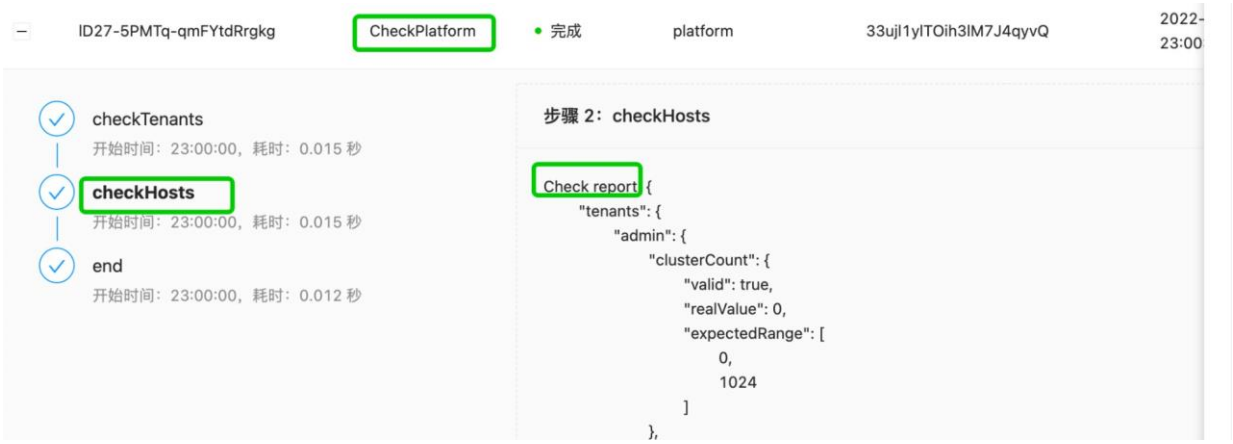


图 238: 集群巡检图

10.14.4.4 TiUniManager 数据导入与导出

本文档介绍如何使用TiUniManager 从 TiDB 集群导入和导出数据。

10.14.4.4.1 导入数据

DBA 管理员可从 S3 兼容存储、本地存储、或TiUniManager 共享存储上，导入 SQL 文件、CSV 文件的数据到TiDB 集群。

TiUniManager 共享存储是用户事先设定好、用于保存导入导出文件的文件资源池，该文件资源池是一个可挂载于TiUniManager 中控机的 NFS 目录，你可在安装TiUniManager 后设置该路径。

导入前准备

进行操作前，确保已经登录TiUniManager 控制台。

> 注意：
>
> - 从本地上传的源数据文件大小不能超过 2 GiB
> - TiUniManager 最多支持 3 个导入导出任务并行运行，后续任务在队列中等待

TiUniManager 默认的导入导出路径是在TiUniManager 中控机上，细节如下：

配置描述	配置参数名	参考值
TiUniManager 共享存储中导入文件的存储位置（建议配置为 NFS 共享存储）	ImportShareStorag/ehPoamteh/tidb/import	（备注：中控机 tidb 账号拥有该路径的读写权限）
TiUniManager 共享存储中导出文件的存储位置（建议配置为 NFS 共享存储）	ExportShareStorag/ehPoamteh/tidb/export	（备注：中控机 tidb 账号拥有该路径的读写权限）

当前TiUniManager 不支持通过前端界面修改导入路径。如需修导入路径，需要通过 OpenAPI 对配置进行修改，以修改 ImportShareStoragePath 为例：

1. 登录获取user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    '→ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    '→ "admin" }'
```

> 注意：
> 你需要将以上命令中·172.16.6.206:4180 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=ImportShareStoragePath" -H "
    '→ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值为 newValue（根据实际的环境设置为新的存储路径）。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    '→ ImportShareStoragePath\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    '→ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

导入数据操作步骤

1. 进入集群管理> 导入导出页面。
2. 点击导入数据按钮，进入导入页面。
3. 输入目标集群信息：集群ID、数据库用户名、数据库密码。
4. 选择源数据。存储位置可以是本地文件或TiUniManager 共享存储或 S3 兼容存储。
5. 输入本次导入备注信息。
6. 点击创建导入任务开始导入。

10.14.4.4.2 导出数据

DBA 管理员可从TiDB 集群将数据以 SQL 文件或CSV 文件格式，导出至 S3 兼容存储、本地存储或TiUniManager 共享存储，并可对导出数据进行筛选。数据筛选的方式包括：

- 按 SQL 语句对导出数据进行筛选
- 按指定库表对导出数据进行筛选

对于保存在TiUniManager 共享存储上的导入数据，你可进一步下载至本地。

导出前准备

在导出数据前，确保已经登录TiUniManager 控制台。

> 注意：
>
> - TiUniManager 最多支持 3 个导入导出任务并行运行，后续任务在队列中等待。
> - 从TiUniManager 共享存储下载的导出数据文件大小不能超过 2 GiB。

当前TiUniManager 前端不支持修改导入导出路径，如需修改导入导出路径，需要通过 OpenAPI 对配置进行修改，以修改 `ExportShareStoragePath` 为例：

1. 登录获取user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    '→ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    '→  "admin" }'
```

> 注意：
> 你需要将以上命令中`172.16.6.206:4180` 替换为实际环境中TiUniManager 中控机的IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=ExportShareStoragePath" -H "
    '→ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值为 newValue（根据实际环境设置为新的存储路径）。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    '→ ExportShareStoragePath\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    '→ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

导出数据操作步骤

1. 进入集群管理> 导入导出页面。
2. 点击导出数据按钮，进入导出数据页面。
3. 输入源集群信息：集群ID、数据库用户名、数据库密码。
4. 选择导出目标位置：TiUniManager 共享存储或 S3 兼容存储。
5. 设置导出选项：导出文件格式，是否筛选数据及筛选条件。
6. 记录本次导出备注信息。
7. 点击创建导出任务开始导出数据。

10.14.4.4.3 查看数据的导入导出记录

你可以通过TiUniManager 查看、删除数据的导入导出记录，并下载记录文件至本地。

在查看数据的导入导出记录前，确保已经登录TiUniManager 控制台。

查看数据的导入导出记录的操作步骤如下：

1. 进入集群管理> 导入导出页面。
2. 查看导入导出记录列表。
3. 选择导出记录的下载，可下载导出记录至本地。
4. 选择记录的删除，可删除导入导出记录。

10.14.4.5 TiUniManager 工作流任务查看

TiUniManager 提供工作流任务页面，你可以从该页面查看在TiUniManager 控制台发起的用户任务进展，以及分析任务失败的原因。

查看工作流任务列表及详情前，确保已经登录TiUniManager 控制台。

查看工作流任务列表及详情的步骤如下：

1. 进入工作量任务管理页面查看任务列表，页面以时间倒序排序展示集群工作量任务。
2. 选择要查看详情的任务，点击+ 图标，可进一步查看任务包含的详细步骤和执行结果。

10.14.4.6 TiUniManager 系统管理

本文档介绍如何通过TiUniManager 管理系统。

10.14.4.6.1 查看系统监控

你可能需要查看TiUniManager 系统监控，了解TiUniManager 系统运行情况、TiUniManager 所在主机运行情况。

操作步骤如下：

1. 登录TiUniManager 控制台。
2. 进入系统管理> 系统监控页面。
3. 选择EM Server，查看 TiUniManager 系统运行情况的 Dashboard。
4. 选择 Node Exporter，查看TiUniManager 所在主机运行情况的 Dashboard。

10.14.4.6.2 查看系统日志

查看TiUniManager 系统日志的步骤如下：

1. 登录TiUniManager 控制台。
2. 进入系统管理> 系统日志页面查看系统日志。

10.14.4.6.3 TiUniManager 告警规则

TiUniManager 默认包含以下告警规则，以便接受相应的告警通知。

work flow error alert

- 报警规则：`sum(increase(em_work_flow_total{flow_status="Error"}[5m]))by(service, biz_type, flow_name, flow_status)> 3`
- 规则描述：错误状态工作流数量超过 3 时告警。
- 处理方法：可能是逻辑问题。查看失败工作流，或者联系TiUniManager 开发人员。

work flow node error alert

- 报警规则：`sum(increase(em_work_flow_node_total{flow_node_status="Error"}[5m]))by(service, biz_type, flow_name, flow_node, flow_node_status)> 3`
- 规则描述：错误状态工作流节点超过 3 时告警。
- 处理方法：可能是逻辑问题。查看失败工作流，或者联系TiUniManager 开发人员。

cluster-server request error rate

- 报警规则：`sum(increase(em_micro_requests_total{service="cluster-server", code!="0"}[1m])/ increase(em_micro_requests_total{service="cluster-server"}[1m]))by(service, method)> 0.05`
- 规则描述：接口失败率超过 5% 时告警。
- 处理方法：可以查看系统日志或系统追踪，或者联系TiUniManager 开发人员。

openapi-server request error rate

- 报警规则：`sum(increase(em_http_requests_total{service="openapi-server", code!="200"}[1m]) / increase(em_http_requests_total{service="openapi-server"}[1m]))by(service, handler, method)> 0.05`
- 规则描述：接口失败率超过 5% 时告警。
- 处理方法：可以查看系统日志或系统追踪，或者联系TiUniManager 开发人员。

10.14.4.6.4 TiUniManager 告警设置

TiUniManager 告警设置支持钉钉、Email 等告警通道，具体见List of notifiers supported by Grafana。在设置TiUniManager 告警前，确保已登录TiUniManager 控制台。

以下示例基于Grafana v7.5.15，展示如何在 Grafana 上配置钉钉的告警通道。

1. 打开 Notification channels 配置页面，点击 New channel 创建通道。

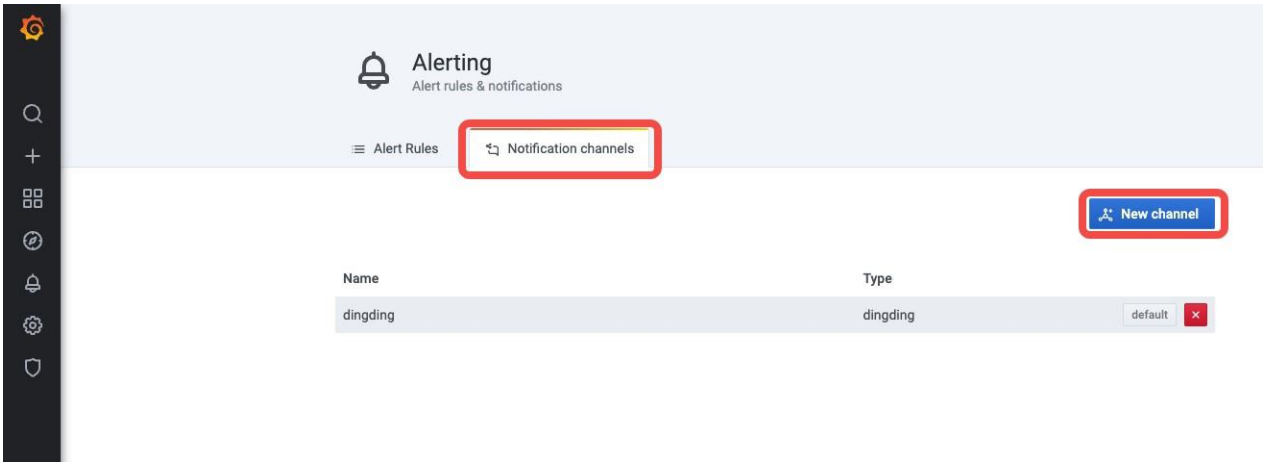


图 239: Notification channels - New channel

2. 配置消息接收方，示例如下：

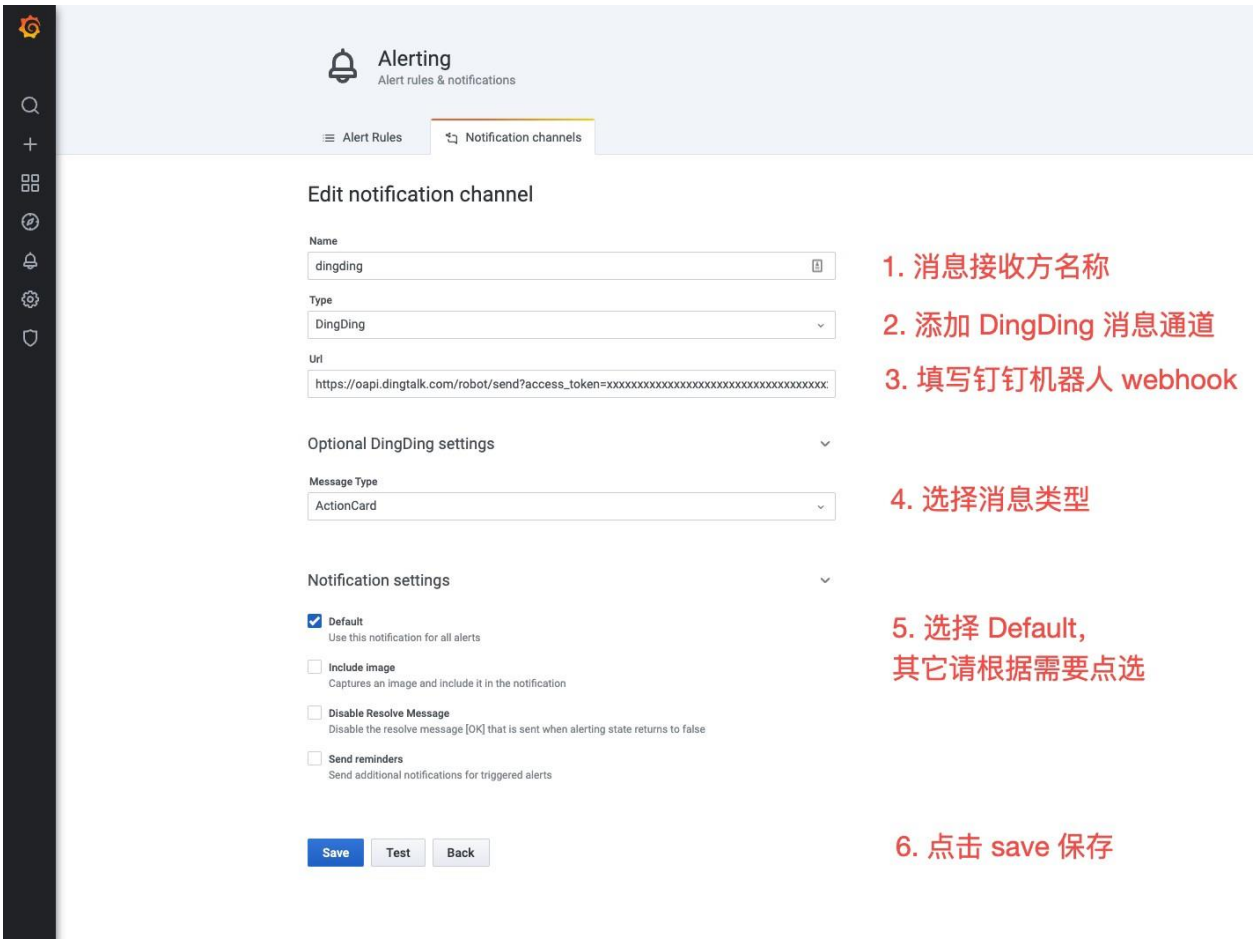


图 240: Notification channels - Edit config

若要接入钉钉自定义机器人，参考钉钉文档- 自定义机器人接入。

10.14.5 TiUniManager 常见问题

本文档介绍TiUniManager 的常见问题，帮助用户更好了解TiUniManager。

10.14.5.1 产品基本FAQ

10.14.5.1.1 TiUniManager 管理的集群可以部署在哪里？

TiUniManager 管理的集群可以部署在用户自建的数据中心或公有云。

10.14.5.1.2 TiUniManager 面向哪几类用户？

TiUniManager 面向三类用户：IT 系统管理员、数据库管理员 DBA、数据库应用开发人员。

10.14.5.1.3 TiUniManager 支持的TiDB 版本有哪些？

见TiUniManager 支持的TiDB 版本。

10.14.5.1.4 TiUniManager 与 TiUP 的关系是什么？

TiUniManager 使用 TiDB 及其生态工具、API 提供数据库管理产品功能，TiUP 是 TiUniManager 使用的工具之一。TiUniManager 产品架构图如下：

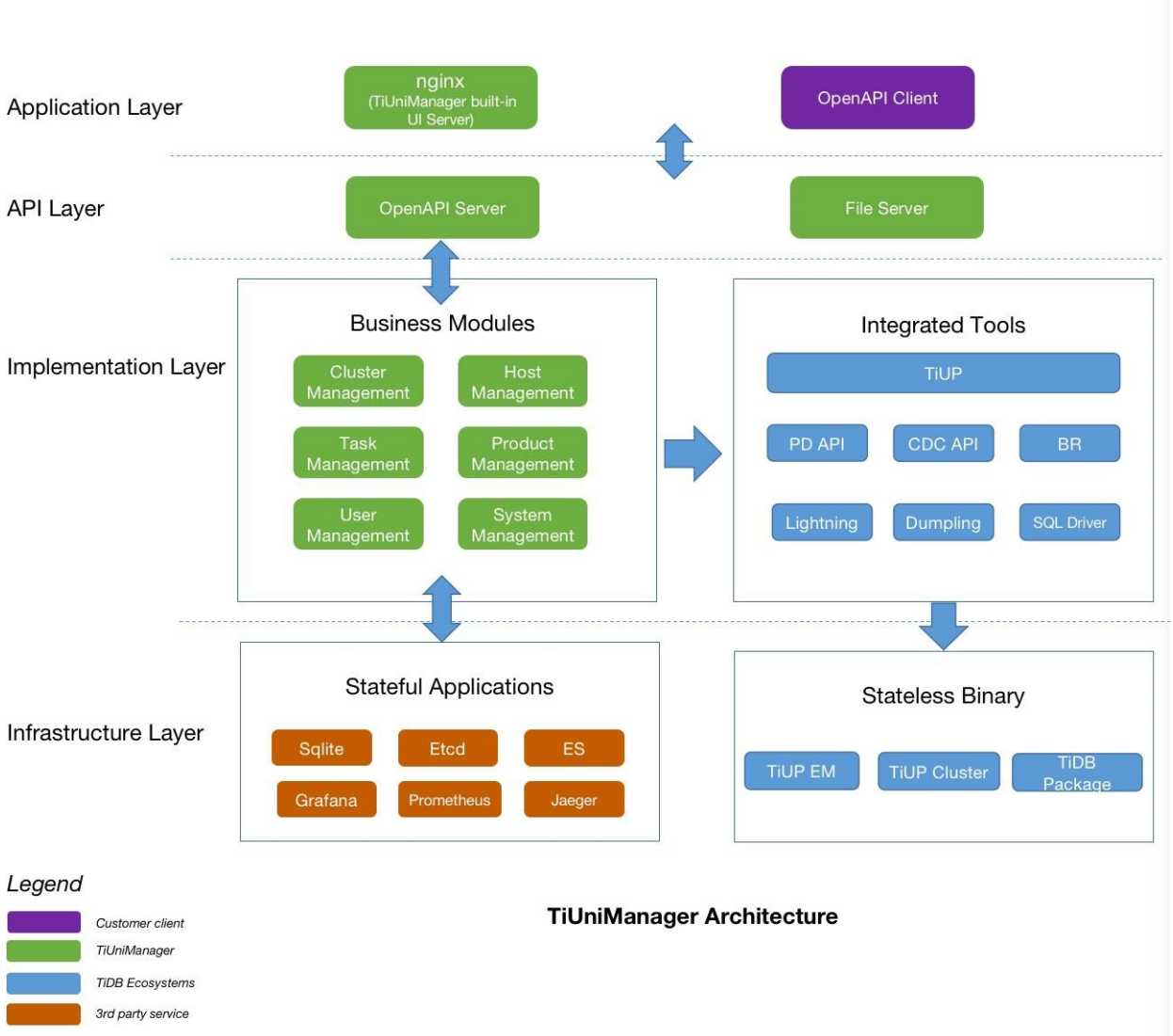


图 241: TiUniManager 架构图

10.14.5.2 主机资源管理FAQ

10.14.5.2.1 何时导入主机资源到TiUniManager？

在创建新集群或导入现有集群前， 你需要将集群所需的主机资源导入到 TiUniManager。具体步骤参见 TiUniManager 资源管理。

10.14.5.3 TiUniManager 安装部署FAQ

10.14.5.3.1 TiUniManager 支持离线安装吗？

支持

10.14.5.3.2 TiUniManager 首次部署后，需要如何做产品初始化吗？

TiUniManager 首次部署后，需要对TiDB 组件规格、数据中心 Zone 信息、主机规格信息进行初始化。

10.14.5.3.3 如何在命令行查看EM 服务和EM 部署的TiDB 集群信息？

首先请先通过 `su - tidb` 命令切换到 `tidb` 账号下。

所有与EM 工具相关的命令，请以 `TIUP_HOME=/home/tidb/.em tiup em <cmd>` 的方式执行，例如 `TIUP_HOME=/home/tidb/.em tiup em list`。

所有与EM 部署的TiDB 集群相关的命令，请以直接以 `tiup cluster <cmd>` 的方式执行，例如 `tiup cluster list`。

10.14.5.4 集群部署FAQ

10.14.5.4.1 TiUniManager 支持在离线环境中部署TiDB 集群吗？

支持

10.14.5.4.2 TiKV 实例数与TiKV 的关系？

创建集群时，TiKV 副本数建议为 1 个、3 个、5 个、或 7 个。设置的TiKV 实例数不能小于TiKV 副本数。

10.14.5.4.3 建议的 PD 实例数量？

建议的 PD 实例数量为 1 个、3 个、5 个、或 7 个。

10.14.5.5 备份与恢复FAQ

10.14.5.5.1 备份支持的存储类型有哪些？

TiUniManager 支持将数据备份至 S3 兼容存储和 NFS 共享存储。

10.14.5.5.2 如何设置TiUniManager 的备份路径？

TiUniManager 目前不支持通过前端界面修改备份路径，修改备份路径方法参考TiUniManager 备份管理。

10.14.5.6 数据导入与导出FAQ

10.14.5.6.1 可以从哪些上游导入数据到TiUniManager？

你可以从 S3 兼容存储、本地存储、或TiUniManager 共享存储上导入集群。

10.14.5.6.2 导出的数据可以保存在哪些存储上？

导出的数据可以保存至 S3 兼容存储或TiUniManager 共享存储，对于保存在TiUniManager 共享存储上的导入数据，你可进一步下载至本地。

10.14.5.6.3 从本地上传的导入文件大小有限制吗？

从本地上传的源数据文件大小不能超过 2 GB。

10.14.5.6.4 什么是TiUniManager 共享存储？

TiUniManager 共享存储是用户事先设定好用于保存导入导出文件的文件资源池，该文件资源池是一个可挂载于 TiUniManager 中控机的 NFS 目录，该 NFS 路径由用户在TiUniManager 安装后进行设置。

10.14.5.6.5 如何修改TiUniManager 共享存储路径？

参见TiUniManager 数据导入与导出。

10.14.5.7 集群扩容缩容FAQ

10.14.5.7.1 对组件实例进行缩容时有何限制？

TiDB、 PD、 TiKV 的实例数量最小为 1。当实例数等于 1 时，不能进行缩容操作。

10.14.5.7.2 TiKV 实例数与TiKV 副本数有何关系？

TiKV 实例数量不能低于TiKV 副本数。对TiKV 进行缩容时，缩容后的TiKV 实例数量不能小于TiKV 副本数。

10.14.5.8 接管集群FAQ

10.14.5.8.1 TiUniManager 接管集群前有什么注意事项吗？

建议用户在TiUniManager 接管集群前，对集群进行一次数据备份。

10.14.5.8.2 接管集群时需要用户填写的“TiUP 部署路径”是什么？

“TiUP 部署路径”是被接管集群中TiUP 的 home 路径。填写该路径时，请勿填写路径结尾的/。

例如，请勿填写为 `/root/.tiup/`。

10.14.6 发布版本历史

10.14.6.1 TiUniManager 版本发布历史

TiUniManager 历史版本发布声明如下：

- v1.0.2
- v1.0.1
- v1.0.0

10.14.6.1.1 TiUniManager 支持的TiDB 版本

当前TiUniManager 仅支持x86_64 版本的TiDB。

TiUniManager 版本	支持的 TiDB 版本（仅 x86_64）
v1.0.2	v6.1.0 v6.0.0 v5.4.0 v5.3.0、v5.3.1 v5.2.0、v5.2.1、v5.2.2、v5.2.3 v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4 v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6
	v1.0.1	v6.0.0 v5.4.0 v5.3.0、v5.3.1 v5.2.0、v5.2.1、v5.2.2、v5.2.3 v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4 v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6
	v1.0.0	v5.4.0 v5.3.0、v5.3.1 v5.2.0、v5.2.1、v5.2.2、v5.2.3 v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4 v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6

10.14.6.2 TiUniManager v1.0.2 Release Notes

发布日期：2022 年 6 月 16 日

发布版本：1.0.2

自 v1.0.2 起，产品名由 TiDB Enterprise Manager 改为TiUniManager，并正式开放源代码。在v1.0.1 版的基础上，v1.0.2 有如下更新：

- 新增对TiDB v6.1.0 集群的支持
- 增加修改密码功能，首次安装后必须修改密码才能完成登录

TiUniManager v1.0.2 支持的TiDB 版本如下：

- v6.1.0
- v6.0.0
- v5.4.0
- v5.3.0、 v5.3.1
- v5.2.0、 v5.2.1、 v5.2.2、 v5.2.3
- v5.1.0、 v5.1.1、 v5.1.2、 v5.1.3、 v5.1.4
- v5.0.0、 v5.0.1、 v5.0.2、 v5.0.3、 v5.0.4、 v5.0.5、 v5.0.6

10.14.6.3 TiDB Enterprise Manager v1.0.1 Release Notes

发布日期：2022 年 5 月 30 日

发布版本：1.0.1

> 注意：
>
> - TiDB Enterprise Manager v1.0.1 未开源。若要获取 v1.0.1 的安装包或相关支持，请联系TiUniManager 团队。
> - 自 v1.0.2 起，TiDB Enterprise Manager 更名为TiUniManager。

该版本在TiDB Enterprise Manager 1.0.0 的基础上，进行了以下更新：

- 新增对TiDB v6.0.0 集群的支持
- 优化 workflow 逻辑，支持重启继续执行
- 修复一些安全基线漏洞，提升了TiDB Enterprise Manager 安全性
- 替换Elasticsearch、 Filebeat、 Kibana、 Grafana 为 OSS 版本

TiDB Enterprise Manager v1.0.1 支持的TiDB 版本如下：

- v6.0.0
- v5.4.0

- v5.3.0、v5.3.1
- v5.2.0、v5.2.1、v5.2.2、v5.2.3
- v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4
- v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6

10.14.6.4 TiDB Enterprise Manager v1.0.0 Release Notes

发布日期：2022 年 4 月 7 日

TiDB Enterprise Manager **版本：**1.0.0

> **注意：**
>
> - TiDB Enterprise Manager v1.0.0 未开源。若要获取 v1.0.0 的安装包或相关支持，请联系TiUni-Manager 团队。
> - 自 v1.0.2 起，TiDB Enterprise Manager 正式更名为TiUniManager。

TiDB Enterprise Manager 是一款以TiDB 数据库为核心的数据库管理平台，帮助用户在私有部署(on-premises) 或公有云环境中管理TiDB 集群。

TiDB Enterprise Manager 不仅提供对TiDB 集群的全生命周期的可视化管理，也同时一站式提供TiDB 数据库参数管理、数据库版本升级、克隆集群、主备集群切换、数据导入导出、数据同步、数据备份恢复服务，能有效提高TiDB 集群运维效率，降低企业运维成本。

10.14.6.4.1 用户价值

分布式数据库管理面临集群规模化、管理场景多元化挑战，传统的命令行管理方式面临部署成本高、技术门槛高、维护成本高等诸多挑战，TiDB Enterprise Manager 能有效解决以下场景中的分布式数据库管理难题：

- 主机资源管理
- 部署集群
- 统一入口管理多套集群
- 升级集群
- 参数管理
- 备份恢复
- 数据导入导出
- 安全与合规
- 自动化与第三方集成
- 管理任务历史记录可查可追溯

10.14.6.4.2 主要功能

集群主机资源管理

- 统一构建TiDB 集群IT 基础设施资源池，主机资源（vCPU、内存、磁盘）总量与分配用量一目了然。

- 支持按集群类型、组件类型对主机资源分类，让最合适的主机资源匹配不同TiDB 组件，提升主机资源利用率。
- 支持按集群独立部署与混合部署对资源进行分类，支持在不同主机资源上灵活快速部署多套TiDB 集群。
- 支持主机位置信息，区域（ Region) - 可用区（ Available Zone) - 机架（ Rack ）三层结构清楚定义主机位置。

集群全生命周期管理

- 支持一站式管理多套TiDB 集群。

  一个Web 入口统一管理多套TiDB 集群，同时查看多套集群日志、监控和告警。

- 支持快速灵活部署 OLTP 集群和 HTAP 集群：

  - 一页配置集群，避免直接修改拓扑和配置文件，有效降低集群配置错误
  - 一键创建集群，缩短 88% 集群创建时间
  - 规范创建流程，降低技术门槛，自动检查创建流程每一步结果，确保数据库集群健康上线，及时响应业务需求

- 支持扩容、缩容TiDB 集群：

  - 支持用户根据业务需要灵活调整TiDB 集群规模，让数据库服务能力与业务规模保持动态适配
  - 集群扩缩容过程中，确保TiDB 组件实例数量符合最佳实践
  - 支持异步完成TiDB 集群缩容，确保Region 数据迁移与集群缩容有序完成

- 支持一键重启、停止、删除TiDB 集群：

  - 支持一键完成TiDB 集群日常操作
  - 删除集群前，提醒用户进行最终备份，确保数据安全
  - 优化删除集群操作的用户交互，防呆设计有效避免集群误删除

- 统一集成数据库监控告警和TiDB Dashboard。

  统一 Web 入口实时监控多套TiDB 数据库性能，及时发现热点和 SQL 慢查询，帮助数据库管理员跟踪数据库核心指标，提前发现数据库潜在风险。

- 支持统一查看TiDB 集群日志。

  统一查看多套TiDB 集群日志，支持按组件、日志级别、时间范围、日志内容、IP 地址过滤日志，帮助数据库管理员快速完成根因定位。

- 支持接管由TiUP 安装的TiDB 集群。 一键接管存量TiDB 集群，统一入口管理新老集群，提高集群管理效率。

克隆集群与一键主备切换

- 支持一键克隆TiDB 集群，快速构建主备集群

  支持一键克隆TiDB 集群，快速构建主备集群，自动构建数据同步任务，确保主备集群间数据保持同步，有效缩短 90% 主备集群创建时间。

- 支持一键切换主备集群：

- 帮助管理员快速完成主备集群切换，自动实现数据同步任务切换，提高数据库容灾能力及业务连续性
- 主备切换过程中确保至少有一个集群始终可读，确保数据库服务不中断
- 主备切换过程中确保数据不丢失

TiDB 集群升级

- 支持TiDB 集群可视化原地升级
  - 支持全流程可视化集群原地升级，简化数据库管理员的业务操作流程，提高集群升级成功率；
  - 提供集群升级前后参数对比，允许数据库管理员核查、选择参数，避免升级前后参数变化对集群性能的影响。

参数管理

- 提供TiDB 版本对应参数组模板：
  - 根据TiDB 最佳实践经验，内置TiDB 版本对应的参数组模板，帮助数据库管理员快速完成集群参数优化
  - 支持用户可自定义参数组，沉淀积累运营中的最佳参数组合
  - 支持用户快速应用参数组至不同集群
- 提供统一界面查看、修改TiDB 集群参数：
  - 统一管理不同接口的参数（SQL、TiUP、各组件 API），屏蔽参数管理接口细节
  - 帮助数据库管理员随时查看、调整TiDB 集群运行参数，降低参数管理维护难度
  - 通过参数值有效校验和定期巡检，减少参数错误对集群性能带来的负面影响

数据备份与恢复

- 支持手动备份TiDB 集群，帮助数据库管理员随时触发备份请求，支持查看备份进度。
- 支持设置TiDB 集群的自动备份策略，帮助数据库管理员设置周期性、自动化的备份策略，满足数据安全、审计的合规要求。
- 支持从备份数据恢复创建新TiDB 集群，备份数据与创建集群与无缝对接，快速恢复集群与数据。
- 支持备份历史记录，方便数据库管理员查看历史备份记录，检查备份任务状态。

数据导入与导出

- 支持数据导入TiDB 集群，从不同存储（S3 兼容存储、NFS 存储、本地存储）快速导入全量数据到TiDB 集群。
- 支持从TiDB 集群导出数据：
  - 从 TiDB 集群导出 SQL 或 CSV 格式的数据，可存储与 S3 兼容的存储或 NFS 存储
  - 支持按库、表过滤导出数据
  - 支持按 SQL 语句过滤导出数据

- 支持跨TiDB 集群导入和导出数据。

  不同TiDB 集群之间，通过TiDB Enterprise Manager 共享存储实现跨集群的数据全量导入与导出，完成数据流动与融合，打破数据孤岛，实现数据价值挖掘。

- 支持导入导出历史记录，方便数据库管理员查看数据导入导出历史记录，检查任务状态。

数据同步

支持TiCDC 数据同步：

- 快速构建从TiDB 集群到下游系统(TiDB、 MySQL、 Kafka ）的数据同步任务
- 支持数据同步任务按库、表过滤
- 支持数据同步任务管理：创建、 暂停、 恢复、 删除

工作流任务管理

- 自动记录管理员操作，历史任务事后可追查、可回溯，帮助数据库管理员查看集群历史任务记录，检查工作流执行进度与结果，快速分析操作失败原因。
- 规范不同场景下数据库管理员操作流程。规范数据库管理员在各场景下的操作流程，避免人工操作错误与遗漏。
- 支持异步工作流任务。管理员任务工作量支持异步，确保用户操作体验。

OpenAPI（实验特性）

提供集群全生命周期 OpenAPI：

- 帮助管理员完成数据库管理自动化
- 实现TiDB 管理信息与第三方平台集成

11 TiDB Dashboard

11.1 TiDB Dashboard 介绍

TiDB Dashboard 是TiDB 自 4.0 版本起提供的图形化界面，可用于监控及诊断 TiDB 集群。TiDB Dashboard 内置于TiDB 的 PD 组件中，无需独立部署。

> 注意：
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将TiDB Dashboard 作为独立的 Pod 部署。具体信息，参考TiDB Operator 部署独立的TiDB Dashboard。

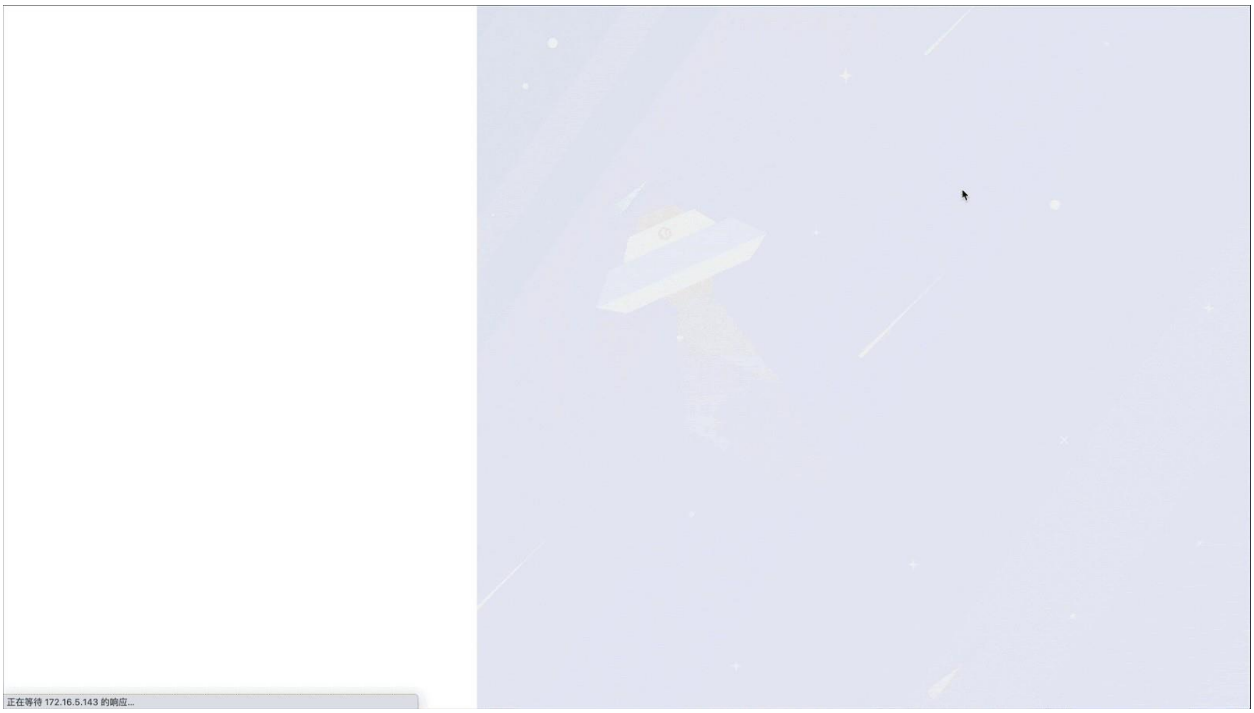


图 242: 界面

TiDB Dashboard 在 GitHub 上开源。

以下列出了TiDB Dashboard 的主要功能，可分别点击小节内的链接进一步了解详情。

11.1.1 了解集群整体运行概况

查看集群整体 QPS 数值、执行耗时、消耗资源最多的几类 SQL 语句等概况信息。

参阅概况页面了解详情。

11.1.2 查看组件及主机运行状态

查看整个集群中TiDB、 TiKV、 PD、 TiFlash 组件的运行状态及其所在主机的运行状态。

参阅集群信息页面了解详情。

11.1.3 分析集群读写流量分布及趋势变化

通过热力图形式可视化地展示整个集群中读写流量随时间的变化情况，及时发现业务模式的变化，或定位性能不均衡的热点所在。

参阅流量可视化页面了解详情。

11.1.4 列出所有 SQL 查询的耗时等执行信息

列出所有 SQL 语句在集群上执行情况，了解各个阶段的执行时间、总运行次数等信息，帮助用户分析和定位集群中最消耗资源的查询，优化整体性能。

参阅SQL 语句分析页面了解详情。

11.1.5 详细了解耗时较长的 SQL 语句的执行信息

列出所有耗时较长的 SQL 语句文本及其执行信息，帮助用户定位 SQL 语句性能缓慢或发生性能抖动的原因。

参阅慢查询页面了解详情。

11.1.6 诊断常见集群问题并生成报告

自动判断集群中是否存在一些常见的风险（如配置不一致）或问题，生成报告并给出操作建议，或对比集群在不同时间段的各个指标状态，供用户分析可能存在问题的方向。

参阅集群诊断页面了解详情。

11.1.7 查询所有组件日志

按关键字、时间范围等条件快速搜索集群中所有运行实例的日志，并可打包下载到本地。

参阅日志搜索页面了解详情。

11.1.8 预估资源管控容量

为使用资源管控(Resource Control) 特性实现资源隔离，集群管理员可以定义资源组(Resource Group)，通过资源组限定配额。

在进行资源规划之前，你需要了解集群的整体容量。参阅资源管控页面了解详情。

11.1.9 收集分析各个组件的性能数据

高级调试功能：无需第三方工具，在线地对各个组件进行性能分析，剖析组件实例在分析时间段内执行的各种内部操作及比例。

参阅实例性能分析页面了解详情。

11.2 运维

11.2.1 部署 TiDB Dashboard

TiDB Dashboard 界面内置于 TiDB 4.0 或更高版本的 PD 组件中，无需额外部署。只需部署标准 TiDB 集群，TiDB Dashboard 就会原生集成。

> **注意：**
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将TiDB Dashboard 作为独立的 Pod 部署。具体信息，参考TiDB Operator 部署独立的TiDB Dashboard。

请参阅下列文档了解如何部署标准TiDB 集群：

- 快速试用TiDB 集群
- 生产环境部署
- Kubernetes 环境部署

> **注意：**
>
> TiDB Dashboard 目前不能在低于 4.0 版本的集群中部署或使用。

11.2.1.1 多 PD 实例部署

当集群中部署了多个 PD 实例时，其中仅有一个 PD 实例会固定地提供TiDB Dashboard 服务。

各个 PD 首次运行时会自动协商出其中某一个实例提供TiDB Dashboard 服务。协商完毕后，无论重启或扩容，都会固定在这个实例上运行TiDB Dashboard 服务，除非该实例被手动缩容。其他 PD 实例不会运行TiDB Dashboard 服务。这个协商过程无需用户介入，会自动完成。

当用户访问不提供TiDB Dashboard 服务的 PD 实例时，浏览器将会收到重定向指令，自动引导用户重新访问提供 了TiDB Dashboard 服务的 PD 实例，从而能正常使用。流程如下图所示。

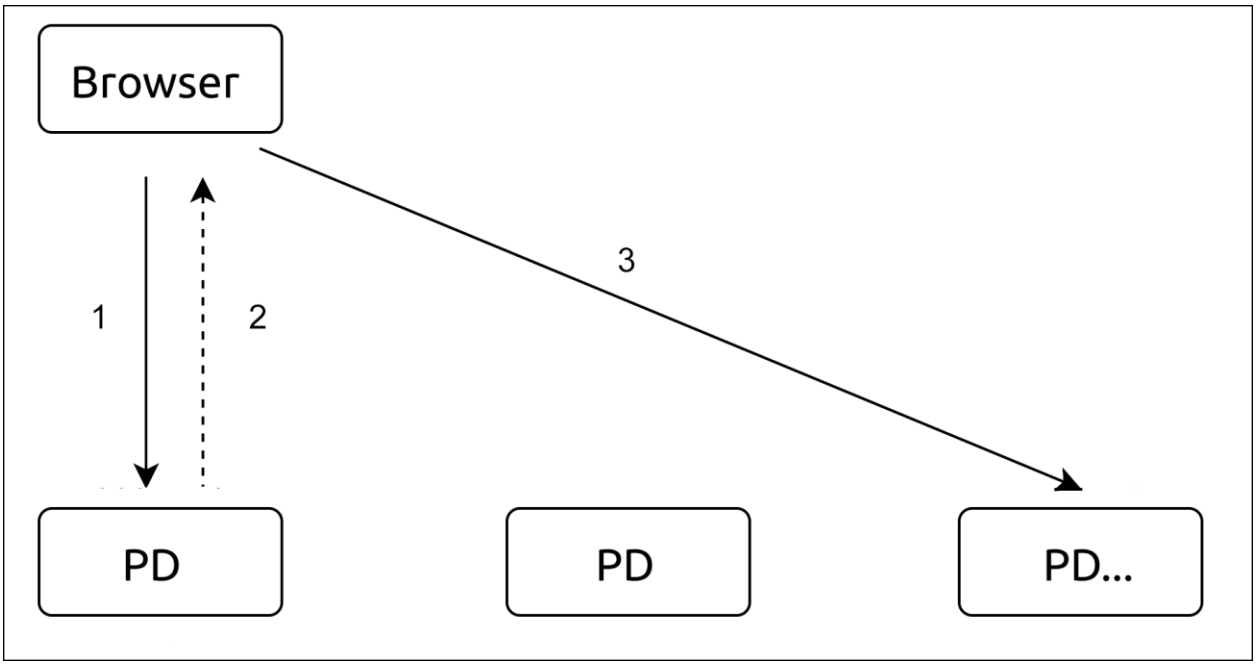


图 243: 流程示意

注意：

提供 TiDB Dashboard 服务的 PD 实例不一定与 PD leader 一致。

11.2.1.1.1 查询实际运行TiDB Dashboard 服务的 PD 实例

使用 TiUP 部署时，对于已启动的集群，可通过 `tiup cluster display` 命令查看哪个 PD 节点提供了 TiDB Dashboard 服务（将 `CLUSTER_NAME` 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出样例如下：

```
http://192.168.0.123:2379/dashboard/
```

注意：

该功能在TiUP Cluster v1.0.3 或更高版本部署工具中提供。

升级TiUP Cluster 步骤

```
tiup update --self
tiup update cluster --force
```

11.2.1.1.2 切换其他 PD 实例提供TiDB Dashboard 服务

使用TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:v<CLUSTER_VERSION> pd` 命令切换其他 PD 实例运行TiDB Dashboard，或在禁用 TiDB Dashboard 的情况下重新指定一个 PD 实例运行TiDB Dashboard：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 config set dashboard-address http
    ↪ ://9.9.9.9:2379
```

其中：

- 将 `127.0.0.1:2379` 替换为任意 PD 实例的IP 和端口
- 将 `9.9.9.9:2379` 替换为想运行TiDB Dashboard 服务的新 PD 实例的IP 和端口

修改完毕后，可使用 `tiup cluster display` 命令确认修改是否生效（将 `CLUSTER_NAME` 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

> 警告：
>
> 切换TiDB Dashboard 将会丢失之前TiDB Dashboard 实例所存储的本地数据，包括流量可视化历史、历史搜索记录等。

11.2.1.2 禁用TiDB Dashboard

使用TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:v<CLUSTER_VERSION> pd` 命令在所有 PD 实例上禁用TiDB Dashboard（将 `127.0.0.1:2379` 替换为任意 PD 实例的IP 和端口）：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 config set dashboard-address none
```

禁用TiDB Dashboard 后，查询哪个 PD 实例提供TiDB Dashboard 服务将会失败：

```
Error: TiDB Dashboard is disabled
```

浏览器访问任意 PD 实例的TiDB Dashboard 地址也将提示失败：

```
Dashboard is not started.
```

11.2.1.3 重新启用TiDB Dashboard

使用TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:v<CLUSTER_VERSION> pd` 命令，要求 PD 重新协商出某一个实例运行TiDB Dashboard（将 `127.0.0.1:2379` 替换为任意 PD 实例的IP 和端口）：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 config set dashboard-address auto
```

修改完毕后，使用`tiup cluster display` 命令查看PD 自动协商出的TiDB Dashboard 实例地址（将`CLUSTER_NAME` 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

还可以通过手动指定哪个 PD 实例运行TiDB Dashboard 服务的方式重新启用TiDB Dashboard，具体操作参见上文切换其他 PD 实例提供TiDB Dashboard 服务。

> **警告：**
>
> 若新启用的TiDB Dashboard 实例与禁用前的实例不一致，将会丢失之前TiDB Dashboard 实例所存储的本地数据，包括流量可视化历史、历史搜索记录等。

11.2.1.4 下一步

- 参阅访问TiDB Dashboard 章节了解如何访问及登录集群上的TiDB Dashboard 界面。
- 参阅提高TiDB Dashboard 安全性章节了解如何增强TiDB Dashboard 的安全性，如配置防火墙等。

11.2.2 通过反向代理使用TiDB Dashboard

你可以使用反向代理将TiDB Dashboard 服务安全从内部网络提供给外部网络。

11.2.2.1 操作步骤

11.2.2.1.1 第 1 步：获取实际TiDB Dashboard 地址

当集群中部署有多个 PD 实例时，其中仅有一个 PD 实例会真正运行TiDB Dashboard，因此需要确保反向代理的上游 (Upstream) 指向了正确的地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署章节。

使用TiUP 部署工具时，操作命令如下（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际TiDB Dashboard 地址。样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> **注意：**
>
> 该功能在TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

11.2.2.1.2 第 2 步：配置反向代理

使用 HAProxy 反向代理

HAProxy 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理TiDB Dashboard 为例，在 HAProxy 配置文件中，新增如下配置：

“ ‘haproxy frontend tidb_dashboard_front bind *:8033 use_backend tidb_dashboard_back if { path /dashboard } or { path_beg /dashboard/ }

backend tidb_dashboard_back mode http server tidb_dashboard 192.168.0.123:2379 “ ‘

其中 192.168.0.123:2379 需替换为第 1 步：获取实际TiDB Dashboard 地址中取得的TiDB Dashboard 实际地址中的 IP 及端口部分。

> 警告：
>
> 请务必保留 use_backend 指令中的 if 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高TiDB Dashboard 安全性。

2. 重启 HAProxy，以使配置生效。
3. 测试反向代理是否生效：访问 HAProxy 所在机器的 8033 端口下 /dashboard/ 地址，如http://example.com:8033/dashboard/，即可访问TiDB Dashboard。

使用 NGINX 反向代理

NGINX 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理TiDB Dashboard 为例，在 NGINX 配置文件中，新增如下配置：

```
nginx  server {  listen 8033; location /dashboard/ { proxy_pass http://192.168.0.123:2379/dashboard
'→ /; } }
```

其中 http://192.168.0.123:2379/dashboard/ 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> 警告：
>
> 请务必保留 proxy_pass 指令中的 /dashboard/ 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高TiDB Dashboard 安全性。

2. 重新载入 NGINX 以使配置生效：

```
shell  sudo nginx -s reload
```

3. 测试反向代理是否生效：访问 NGINX 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com
'→ :8033/dashboard/，即可访问 TiDB Dashboard。

11.2.2.2 自定义路径前缀

TiDB Dashboard 默认在 /dashboard/ 路径下提供服务，即使是反向代理也是如此，例如 http://example.com:8033/dashboard/。若要配置反向代理以非默认的路径提供 TiDB Dashboard 服务，例如 http://example.com:8033/foo/ 或 http://example.com:8033/，可参考以下步骤。

11.2.2.2.1 第 1 步：修改 PD 配置指定TiDB Dashboard 服务路径前缀

修改 PD 配置中 [dashboard] 类别的 public-path-prefix 配置项，可指定服务路径前缀。该配置修改后需要重启 PD 实例生效。

以TiUP 部署且希望运行在 http://example.com:8033/foo/ 为例，可指定以下配置：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /foo
```

使用TiUP 部署全新集群时修改配置

若要全新部署集群，可在TiUP 拓扑文件 topology.yaml 中加入上述配置项后进行部署，具体步骤参阅TiUP 部署文档。

使用TiUP 修改已部署集群的配置

1. 以编辑模式打开该集群的配置文件（将 CLUSTER_NAME 替换为集群名称）

```
shell tiup cluster edit-config CLUSTER_NAME
```

2. 在 server_configs 的 pd 配置下修改或新增配置项，若没有 server_configs 请在最顶层新增：

```
yaml server_configs: pd:     dashboard.public-path-prefix: /foo
```

修改完成后的配置文件类似于：

```
yaml server_configs: pd:     dashboard.public-path-prefix: /foo global: user: tidb ...
```

或

```
yaml monitored: ... server_configs: tidb: ... tikv: ... pd:     dashboard.public-path-
↪ prefix: /foo ...
```

3. 滚动重启所有 PD 实例生效配置（将 CLUSTER_NAME 替换为集群名称）

```
shell tiup cluster reload CLUSTER_NAME -R pd
```

详情请参阅TiUP 常见运维操作- 修改配置参数。

若希望运行在根路径（如 http://example.com:8033/）下，相应的配置为：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /
```

> **警告：**
>
> 修改自定义路径前缀生效后，直接访问将不能正常使用TiDB Dashboard，您只能通过和路径前缀匹配的反向代理访问。

11.2.2.2.2 第 2 步：修改反向代理配置

使用 HAProxy 反向代理

以 `http://example.com:8033/foo/` 为例，HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back if { path /foo } or { path_beg /foo/ }

backend tidb_dashboard_back
  mode http
  http-request set-path %[path,regsub(^/foo/?,/dashboard/)]
  server tidb_dashboard 192.168.0.123:2379
```

其中 `192.168.0.123:2379` 需替换为第 1 步：获取实际TiDB Dashboard 地址中取得的TiDB Dashboard 实际地址中的 IP 及端口部分。

> **警告：**
>
> 请务必保留 `use_backend` 指令中的 `if` 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高TiDB Dashboard 安全性。

若希望运行在根路径（如 `http://example.com:8033/`），HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back

backend tidb_dashboard_back
  mode http
  http-request set-path /dashboard%[path]
  server tidb_dashboard 192.168.0.123:2379
```

修改配置并重启 HAProxy 后即可生效。

使用 NGINX 反向代理

以 `http://example.com:8033/foo/` 为例，相应的 NGINX 配置为：

```
server
  { listen
  8033;
  location /foo/ {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

其中 `http://192.168.0.123:2379/dashboard/` 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> **警告：**
>
> 请务必保留 `proxy_pass` 指令中的 `/dashboard/` 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 `http://example.com:8033/`），NGINX 配置为：

```
server
  { listen
  8033;
  location / {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

修改配置并重启 NGINX 后即可生效：

```
sudo nginx -s reload
```

11.2.2.3 下一步

参阅提高TiDB Dashboard 安全性文档了解如何增强TiDB Dashboard 的安全性，如配置防火墙等。

11.2.3 TiDB Dashboard 用户管理

TiDB Dashboard 与 TiDB 使用相同的用户权限体系和登录验证方式。你可以通过控制和管理TiDB SQL 用户，从而限制和约束这些用户对TiDB Dashboard 的访问。本文描述了TiDB SQL 用户访问TiDB Dashboard 所需的最小权限，并提供了如何创建最小权限 SQL 用户、如何通过RBAC 授权 SQL 用户登录的示例。

要了解如何控制和管理TiDB SQL 用户，请参见TiDB 用户账户管理。

11.2.3.1 所需权限说明

- 当所连接的TiDB 服务器未启用安全增强模式(SEM) 时，要访问TiDB Dashboard，SQL 用户应当拥有以下所有权限：

  - PROCESS
  - SHOW DATABASES
  - CONFIG
  - DASHBOARD_CLIENT

- 当所连接的TiDB 服务器启用了安全增强模式(SEM) 时，要访问TiDB Dashboard，SQL 用户应当拥有以下所有权限：

  - PROCESS
  - SHOW DATABASES
  - CONFIG
  - DASHBOARD_CLIENT
  - RESTRICTED_TABLES_ADMIN
  - RESTRICTED_STATUS_ADMIN
  - RESTRICTED_VARIABLES_ADMIN

- 若希望 SQL 用户在登录TiDB Dashboard 后允许修改界面上的各项配置，SQL 用户还应当拥有以下权限：

  - SYSTEM_VARIABLES_ADMIN

- 若希望 SQL 用户在登录TiDB Dashboard 后允许使用快速绑定执行计划功能。

  - SYSTEM_VARIABLES_ADMIN
  - SUPER

> 注意：
>
> 拥有 `ALL PRIVILEGES` 或 `SUPER` 等粗粒度高权限的用户同样可以登录TiDB Dashboard。出于最小权限原则，强烈建议创建用户时仅使用上述精细权限，从而防止用户执行非预期操作。请参阅权限管理了解这些权限的详细信息。

如果登录TiDB Dashboard 时指定的 SQL 用户未满足上述权限需求，则登录将失败，如下图所示：

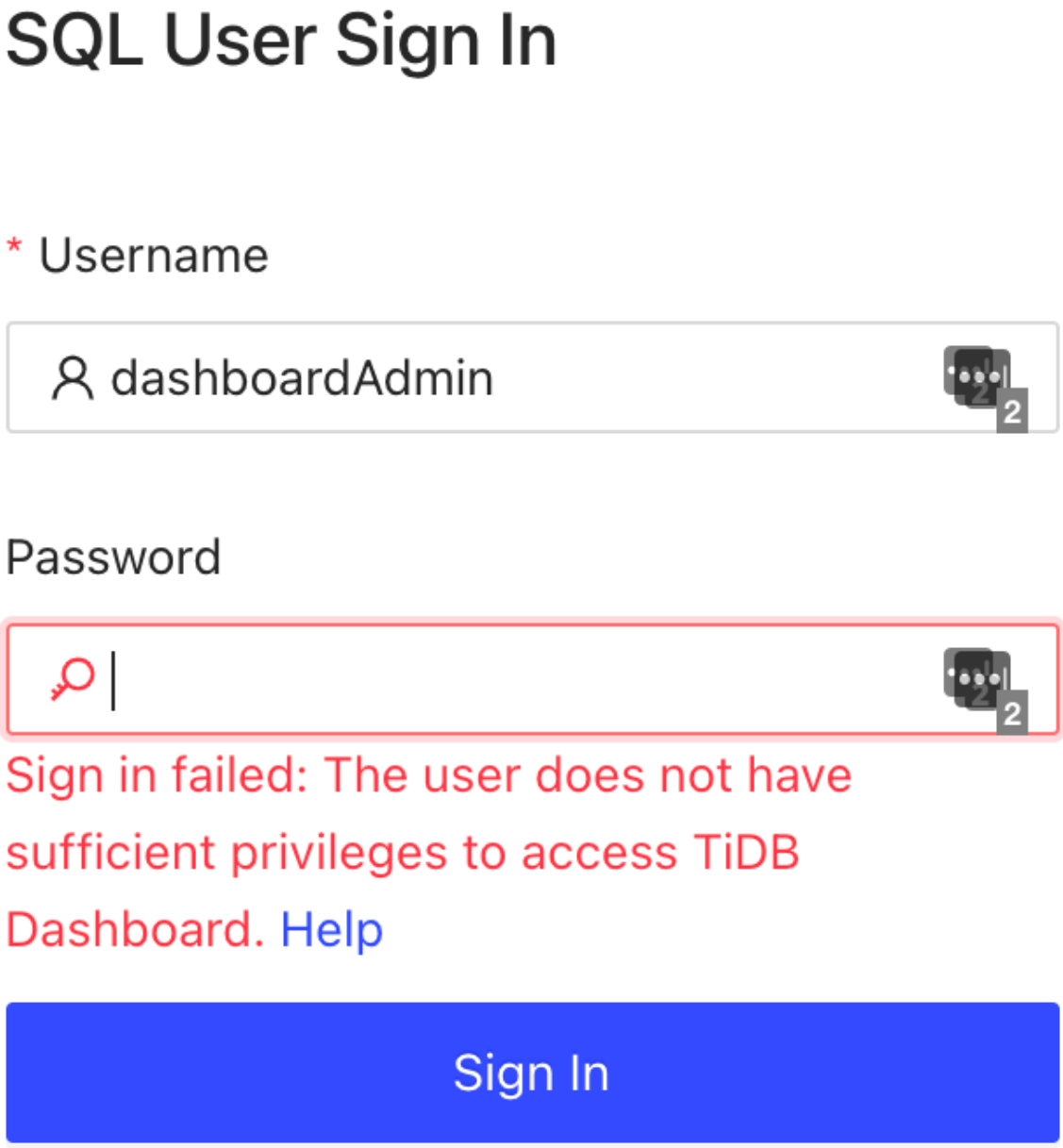


图 244: insufficient-privileges

11.2.3.2 示例：创建一个最小权限 SQL 用户用于登录TiDB Dashboard

- 当所连接的TiDB 服务器未启用安全增强模式(SEM)时，你可以通过执行以下示例 SQL 语句创建一个允许登录TiDB Dashboard 的 SQL 用户 dashboardAdmin：

```
CREATE USER 'dashboardAdmin'@'%' IDENTIFIED BY '<YOUR_PASSWORD>';
GRANT PROCESS, CONFIG ON *.* TO 'dashboardAdmin'@'%';
GRANT SHOW DATABASES ON *.* TO 'dashboardAdmin'@'%';
```

```
GRANT DASHBOARD_CLIENT ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使自定义的 SQL 用户能修改 TiDB Dashboard 界面上的各项配置，可以增加以下权限
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使用快速绑定执行计划（具体参见 https://docs.pingcap.com/zh/tidb/dev/dashboard-
    ↪ statement-details#快速绑定执行计划）功能，可以增加以下权限
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT SUPER ON *.* TO 'dashboardAdmin'@'%';
```

- 当所连接的TiDB 服务器启用了安全增强模式(SEM) 时，先关闭 SEM，然后执行以下示例 SQL 语句创建一个允许登录TiDB Dashboard 的 SQL 用户 dashboardAdmin，创建完成后，再重新开启 SEM：

```
CREATE USER 'dashboardAdmin'@'%' IDENTIFIED BY '<YOUR_PASSWORD>';
GRANT PROCESS, CONFIG ON *.* TO 'dashboardAdmin'@'%';
GRANT SHOW DATABASES ON *.* TO 'dashboardAdmin'@'%';
GRANT DASHBOARD_CLIENT ON *.* TO 'dashboardAdmin'@'%';
GRANT RESTRICTED_STATUS_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT RESTRICTED_TABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT RESTRICTED_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使自定义的 SQL 用户能修改 TiDB Dashboard 界面上的各项配置，可以增加以下权限
GRANT SUPER ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使用快速绑定执行计划（具体参见 https://docs.pingcap.com/zh/tidb/dev/dashboard-
    ↪ statement-details#快速绑定执行计划）功能，可以增加以下权限
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT SUPER ON *.* TO 'dashboardAdmin'@'%';
```

11.2.3.3 示例：通过RBAC 授权 SQL 用户登录TiDB Dashboard

以下示例演示了如何在基于角色的访问控制(RBAC) 机制下创建角色及用户来登录TiDB Dashboard。

1. 创建一个包含TiDB Dashboard 所有功能所需权限的角色 dashboard_access：

```
CREATE ROLE 'dashboard_access';
GRANT PROCESS, CONFIG ON *.* TO 'dashboard_access'@'%';
GRANT SHOW DATABASES ON *.* TO 'dashboard_access'@'%';
GRANT DASHBOARD_CLIENT ON *.* TO 'dashboard_access'@'%';
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboard_access'@'%';
GRANT SUPER ON *.* TO 'dashboard_access'@'%';
```

2. 为其他用户授权 dashboard_access 角色并设置为默认启用：

```
CREATE USER 'dashboardAdmin'@'%' IDENTIFIED BY '<YOUR_PASSWORD>';
GRANT 'dashboard_access' TO 'dashboardAdmin'@'%';
```

```
-- 需要默认启用 dashboard_access 角色
SET DEFAULT ROLE dashboard_access to 'dashboardAdmin'@'%';
```

完成以上步骤后，可以用 dashboardAdmin 用户登录TiDB Dashboard。

11.2.3.4 登录TiDB Dashboard

创建满足TiDB Dashboard 权限要求的 SQL 用户后，你可以使用该用户登录TiDB Dashboard。

11.2.4 提高TiDB Dashboard 安全性

尽管访问TiDB Dashboard 需要登录，但它被设计为默认供可信的用户实体访问。当你希望将TiDB Dashboard 提供给外部网络用户或不可信用户访问时，需要注意采取以下措施以避免安全漏洞。

11.2.4.1 提高TiDB 用户安全性

11.2.4.1.1 为 root 用户设置强密码

TiDB Dashboard 的账号体系与TiDB SQL 用户一致。默认部署情况下，TiDB 的 root 用户没有密码，因而访问TiDB Dashboard 也不需要密码验证。这将会给恶意访问者极大的集群权限，包括执行特权 SQL 等。

建议的措施：

- 为TiDB 的 root 用户设置一个强密码（请参见TiDB 用户账户管理了解详情），或禁用 root 账户。

11.2.4.1.2 为 TiDB Dashboard 创建最小权限用户

TiDB Dashboard 的账号体系与 TiDB SQL 用户一致，并基于 TiDB SQL 用户的权限进行 TiDB Dashboard 授权验证。TiDB Dashboard 所需的权限较少，甚至可以只有只读权限。可以基于最小权限原则配置合适的用户访问TiDB Dashboard，减少高权限用户的使用场景。

建议的措施：

- 为访问TiDB Dashboard 创建一个最小权限的 SQL 用户，并用该用户登录TiDB Dashboard，避免使用高权限用户，提升安全性。请参见TiDB Dashboard 用户管理了解详情。

11.2.4.2 使用防火墙阻止不可信访问

> 注意：
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将TiDB Dashboard 作为独立的 Pod 部署。在TiDB Operator 环境，可直接访问该 Pod 的 IP 来打开TiDB Dashboard，该端口不与其他 PD 内部特权接口关联，对外提供该端口不需要额外的防火墙操作。具体信息，参考TiDB Operator 部署独立的TiDB Dashboard。

TiDB Dashboard 通过 PD Client 端口提供服务，默认为 http://IP:2379/dashboard/。尽管 TiDB Dashboard 需要验证身份，但 PD Client 端口上承载的其他 PD 内部特权接口不需要验证身份，且能进行特权操作，例如 http://IP:2379/pd/api/v1/members。因此，将 PD Client 端口直接暴露给外部网络具有极大的风险。

建议的措施：

1. 使用防火墙禁止组件外部网络或不可信网络访问任何 PD 组件的 Client 端口。

注意，TiDB、TiKV 等组件需要通过 PD Client 端口与 PD 组件进行通信，因此请勿对组件内部网络阻止访问，这将导致集群不可用。

2. 参见通过反向代理使用TiDB Dashboard 了解如何配置反向代理将TiDB Dashboard 服务在另一个端口上安全地提供给外部网络。

11.2.4.2.1 如何在多 PD 实例部署时开放TiDB Dashboard 端口访问

> 警告：
>
> 本章节描述了一个不安全的访问方案，仅供测试环境使用。在生产环境中，不要使用本方案。请参见本文的其余章节。

在测试环境中，您可能需要配置防火墙开放TiDB Dashboard 端口供外部访问。

当部署了多个 PD 实例时，其中仅有一个 PD 实例会真正运行TiDB Dashboard，访问其他 PD 实例时会发生浏览器重定向，因此需要确保防火墙配置了正确的IP 地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署章节。

使用TiUP 部署工具时，可使用以下命令查看实际运行TiDB Dashboard 的 PD 实例地址（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际TiDB Dashboard 地址。

> 注意：
>
> 该功能在TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

以下是一个样例输出：

```
http://192.168.0.123:2379/dashboard/
```

在这个样例中，需要为防火墙配置开放IP 192.168.0.123 的 2379 端口入站访问，并通过 http://192.168.0.123:2379/dashboard/ 访问TiDB Dashboard。

11.2.4.3 反向代理仅代理TiDB Dashboard

如前文所述，PD Client 端口下提供的服务不仅有TiDB Dashboard（位于 http://IP:2379/dashboard/），还有其他 PD 内部特权接口（如http://IP:2379/pd/api/v1/members）。因此，使用反向代理将 TiDB Dashboard 提供给外部网络时，应当确保仅提供 /dashboard 前缀下的服务，而非该端口下所有服务，避免外部网络能通过反向代理访问到 PD 内部特权接口。

建议的措施：

- 参见通过反向代理使用TiDB Dashboard 了解安全且推荐的反向代理配置。

11.2.4.4 为反向代理开启 TLS

为了进一步加强传输层安全性，可以为反向代理开启 TLS，甚至可以引入 mTLS 实现访问用户的证书验证。

请参阅 NGINX 文档或 HAProxy 文档了解如何为它们开启 TLS。

11.2.4.5 其他建议的安全措施

- 为TiDB 组件间通信开启加密传输
- 为TiDB 客户端服务端间通信开启加密传输

11.3 访问TiDB Dashboard

通过浏览器访问http://127.0.0.1:2379/dashboard/（将 127.0.0.1:2379 替换为实际 PD 实例的地址和端口）即可打开 TiDB Dashboard。

> 注意：
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将TiDB Dashboard 作为独立的 Pod 部署。在TiDB Operator 环境，可直接访问该 Pod 的IP 来打开TiDB Dashboard。具体信息，参考 TiDB Operator 部署独立的TiDB Dashboard。

11.3.1 多 PD 实例访问

当集群中部署有多个 PD 实例、且您可以直接访问到每个 PD 实例地址和端口时，可以简单地将http://127.0.0.1:2379/dashboard/ 地址中的 127.0.0.1:2379 替换为集群中任意一个 PD 实例的地址和端口进行访问。

> **注意：**
>
> 当处于防火墙或反向代理等环境下、无法直接访问每个 PD 实例时，可能会无法访问 TiDB Dashboard。这通常是防火墙或反向代理没有正确配置导致的。可参阅通过反向代理使用 TiDB Dashboard 或提高 TiDB Dashboard 安全性章节了解如何在多 PD 实例情况下正确配置防火墙或反向代理规则。

11.3.2 浏览器兼容性

TiDB Dashboard 可在常见的、更新及时的桌面浏览器中使用，具体版本号为：

- Chrome >= 77
- Firefox >= 68
- Edge >= 17

> **注意：**
>
> 若使用旧版本浏览器或其他浏览器访问TiDB Dashboard，部分界面可能不能正常工作。

11.3.3 登录

访问TiDB Dashboard 将会显示用户登录界面。

- 可使用TiDB 的 root 用户登录。
- 如果创建了自定义 SQL 用户，也可以使用自定义的 SQL 用户和密码登录。

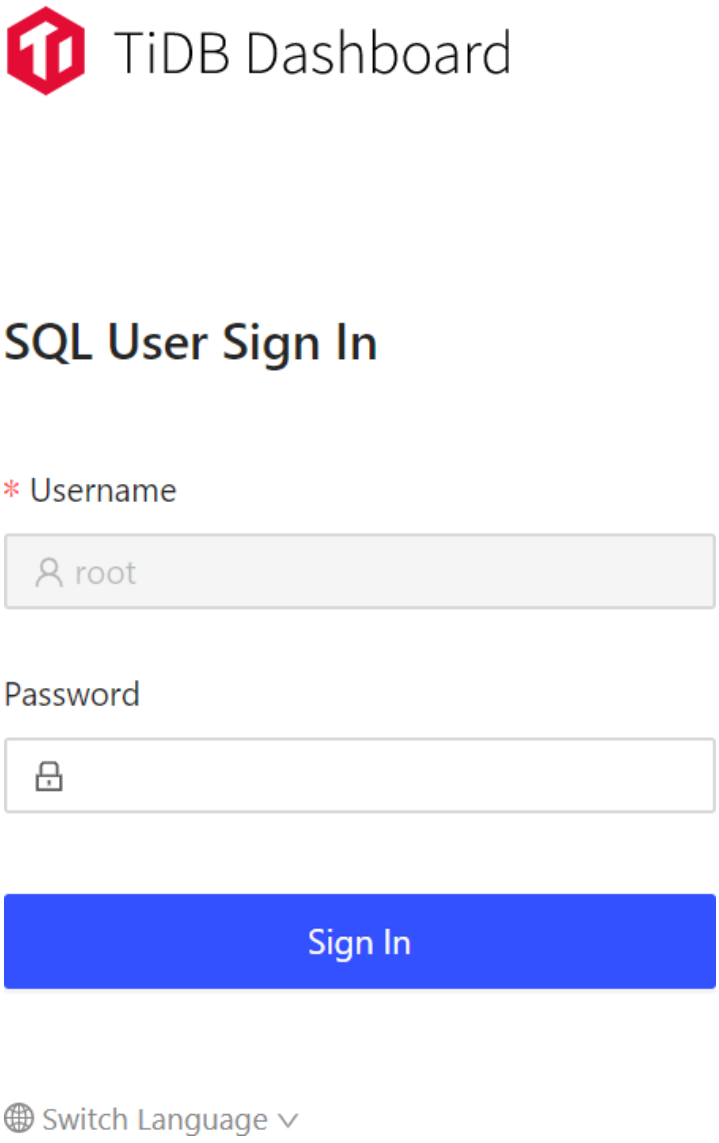


图 245: 登录界面

如果存在以下情况，则可能会登录失败：

- TiDB root 用户不存在
- PD 未启动或无法访问
- TiDB 未启动或无法访问
- root 密码错误

登录后，24 小时内将保持自动登录状态。参见登出章节了解如何登出用户。

11.3.4 切换语言

TiDB Dashboard 目前支持以下语言：

- 简体中文
- 英文

在登录界面中，可点击 Switch Language 下拉框切换界面显示语言：

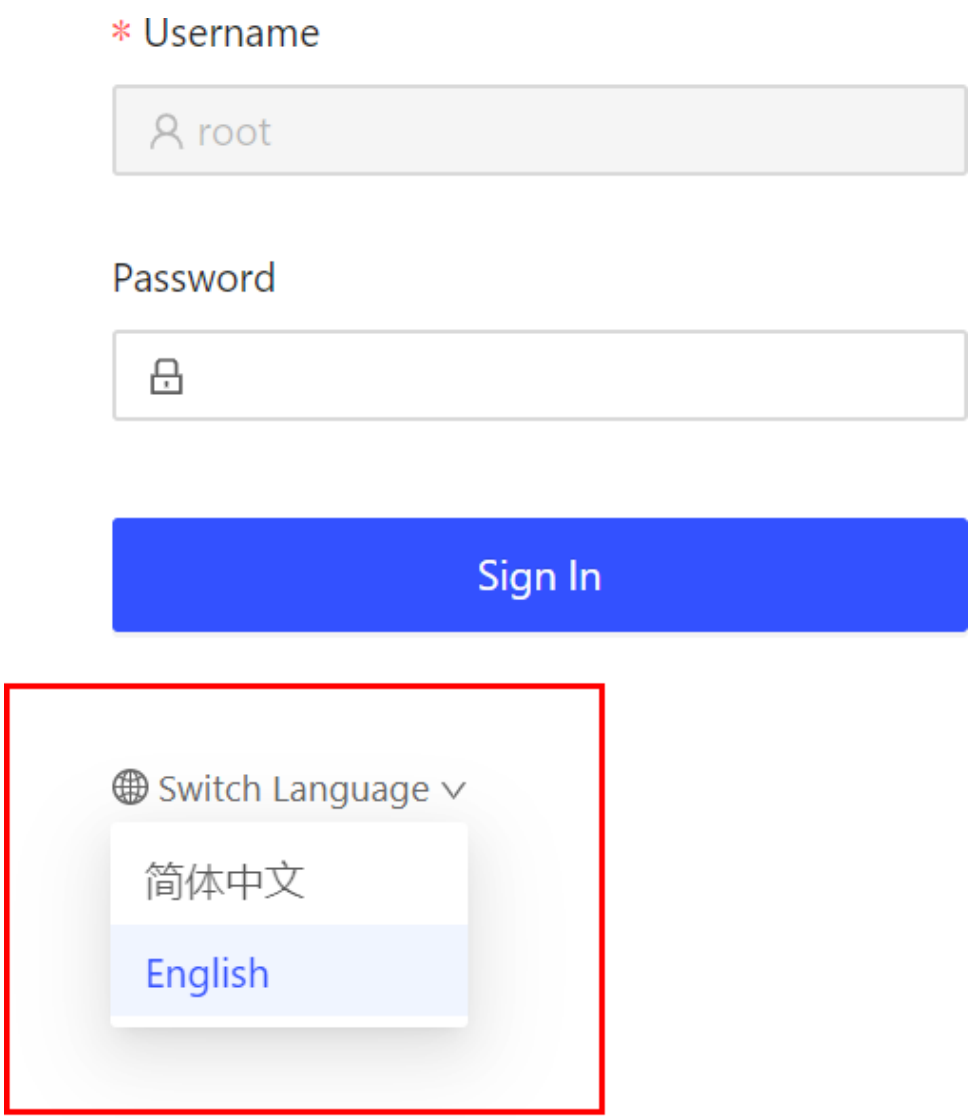


图 246: 切换语言

11.3.5 登出

登录后，在左侧导航处点击登录用户名，可切换到用户页面。在用户页面点击登出(Logout) 按钮即可登出当前用户。登出后，需重新输入用户名密码。

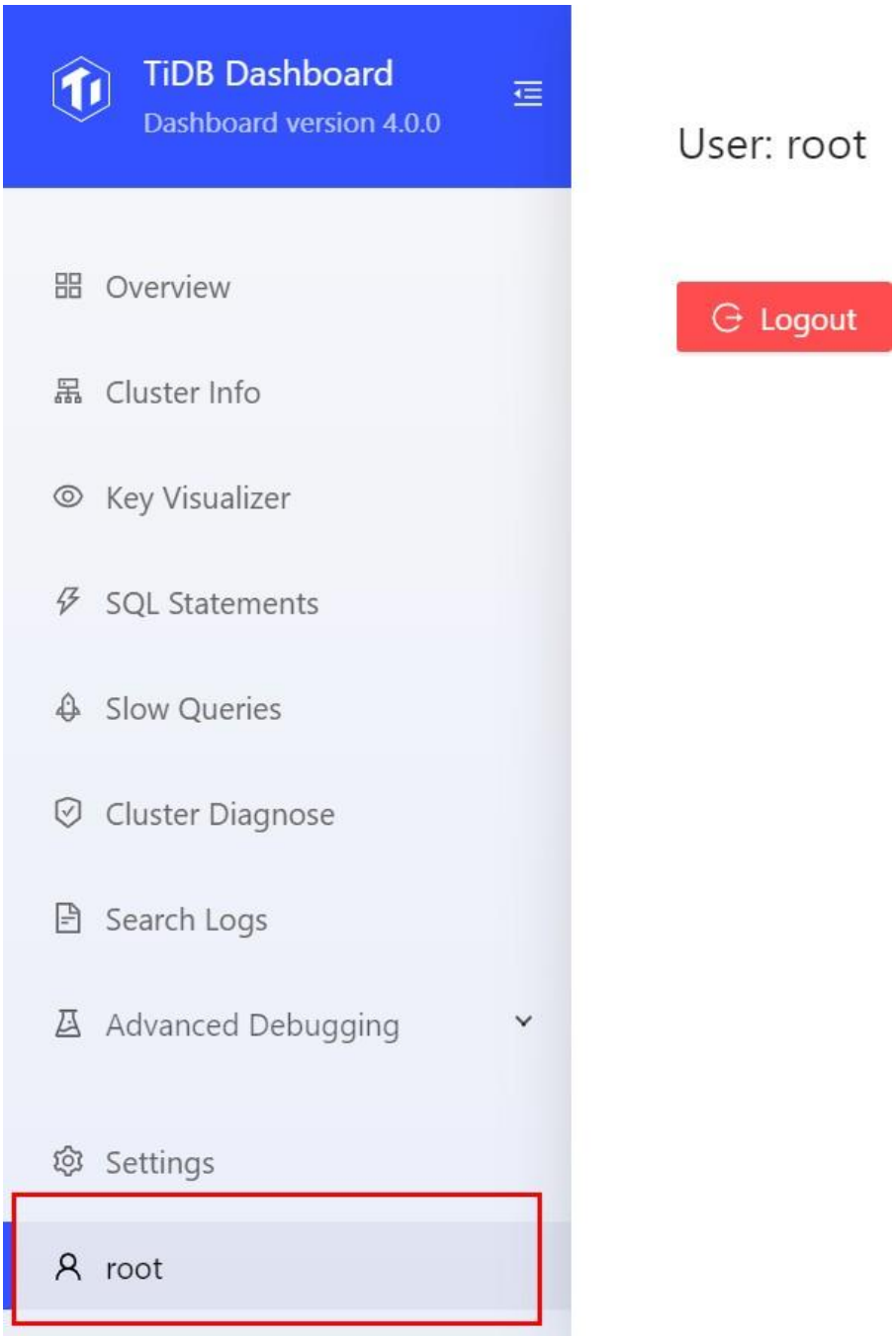


图 247: 登出

11.4 TiDB Dashboard 概况页面

该页面显示了整个集群的概况，包含以下信息：

- 整个集群的 QPS
- 整个集群的查询延迟

- 最近一段时间内累计耗时最多的若干 SQL 语句
- 最近一段时间内运行时间超过一定阈值的慢查询
- 各个实例的节点数和状态
- 监控及告警信息

11.4.1 访问

登录TiDB Dashboard 后默认进入该页面，也可以左侧导航条点击概况(Overview) 进入：

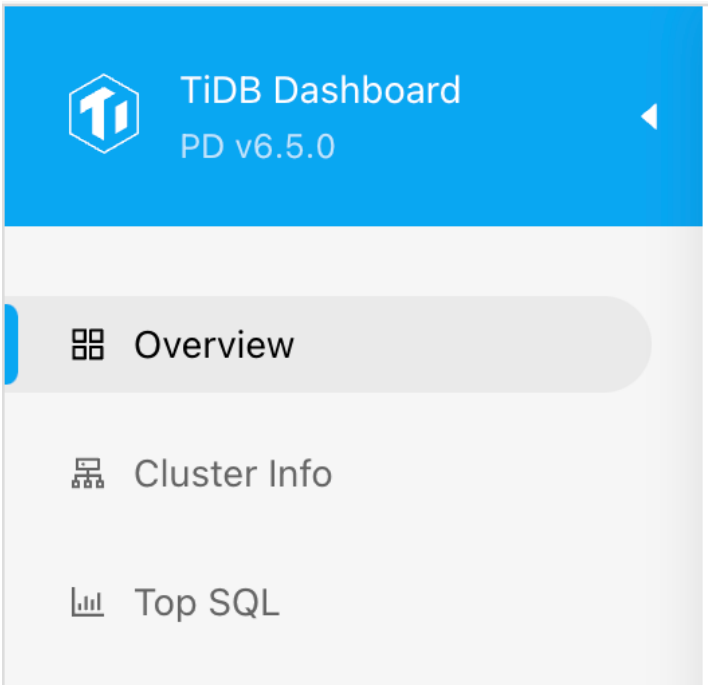


图 248: 访问

11.4.2 QPS

该区域显示最近一小时整个集群的每秒成功和失败查询数量：

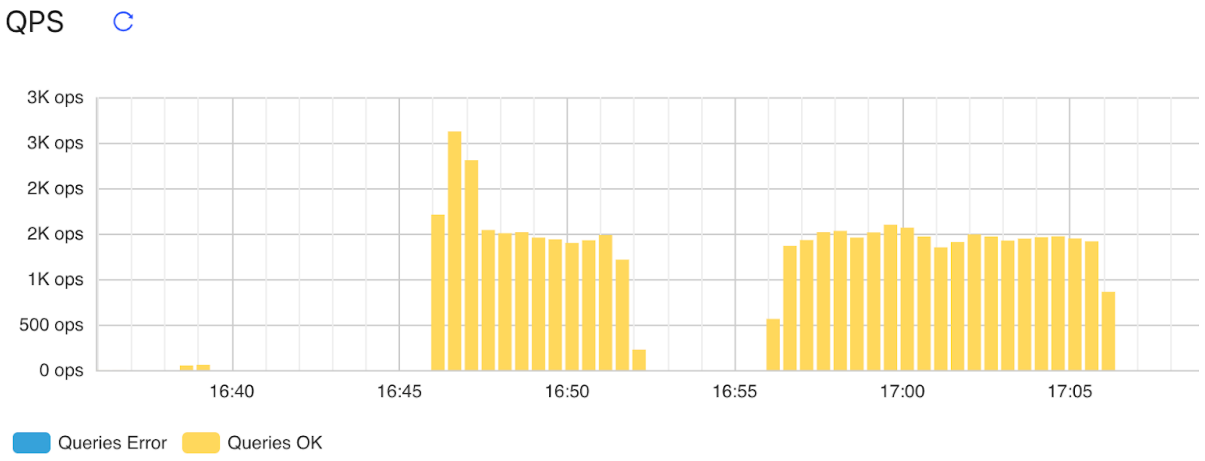


图 249: 界面

> **注意：**
>
> 该功能仅在部署了 Prometheus 监控组件的集群上可用，未部署监控组件的情况下会显示为失败。

11.4.3 延迟

该区域显示最近一小时整个集群中 99.9%、99% 和 90% 查询的延迟：

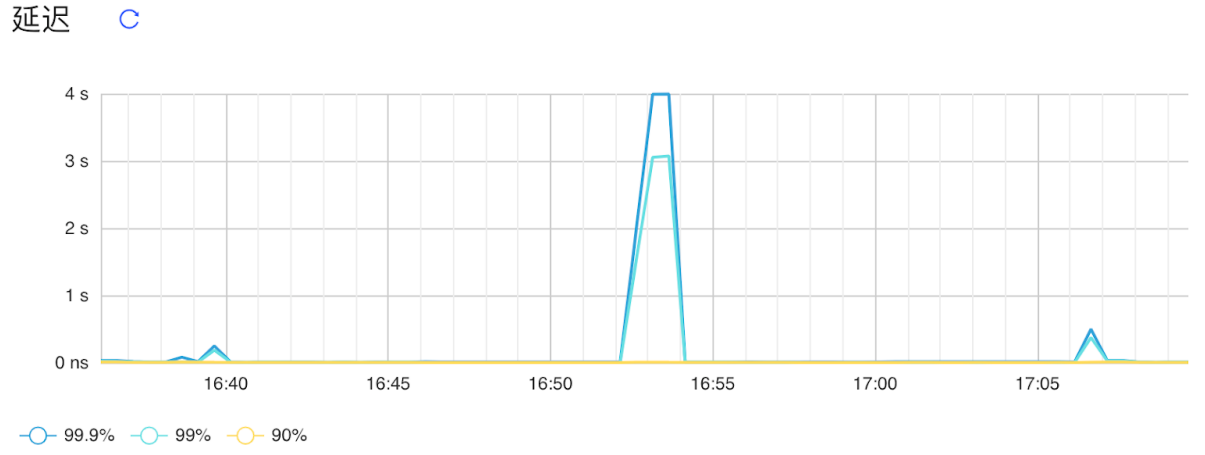


图 250: 界面

> 注意：
>
> 该功能仅在部署了 Prometheus 监控组件的集群上可用，未部署监控组件的情况下会显示为失败。

11.4.4 Top SQL 语句

该区域显示最近一段时间内整个群集中累计耗时最长的 10 类 SQL 语句。查询参数不一样但结构一样的 SQL 会归为同一类 SQL 语句，在同一行中显示：

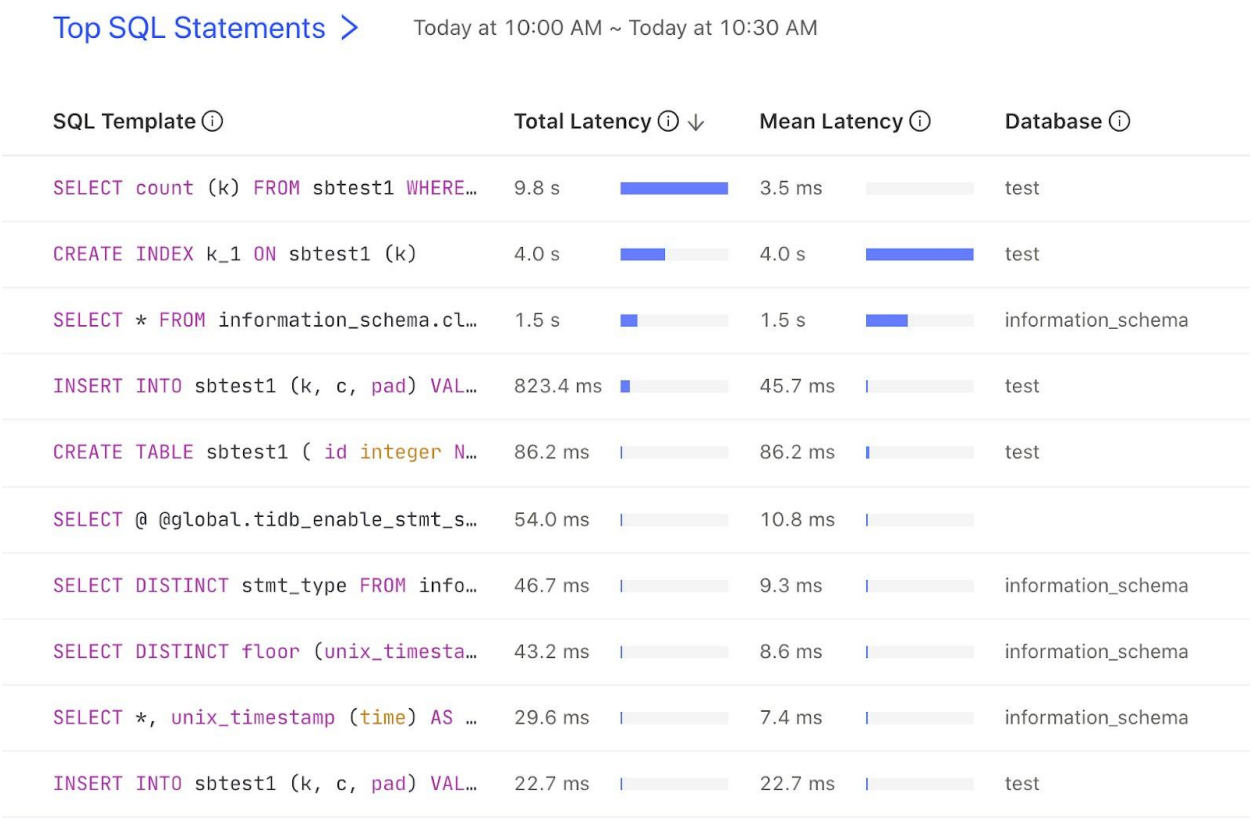


图 251: 界面

该区域显示的内容与SQL 语句分析页面一致，可点击Top SQL 语句(Top SQL Statements)标题查看完整列表。关于该表格中各列详情，见SQL 语句分析页面。

> 注意：
>
> 该功能仅在开启了 SQL 语句分析功能的集群上可用。

11.4.5 最近的慢查询

该区域默认显示最近 30 分钟内整个集群中最新的 10 条慢查询：



图 252: 界面

默认情况下运行时间超过 300ms 的 SQL 查询即会被计为慢查询并显示在该表格中。可通过调整tidb_slow_log_threshold 变量或TiDB slow-threshold 参数调整阈值。

该区域显示的内容与慢查询页面一致，可点击最近的慢查询(Recent Slow Queries) 标题查看完整列表。关于该表格中各列详情，见慢查询页面。

> 注意：
>
> 该功能仅在配置开启了慢查询日志的集群中可用，使用 TiUP 部署的集群默认开启慢查询日志。

11.4.6 实例

该区域汇总显示了整个集群中TiDB、 TiKV、 PD、 TiFlash 的总实例数量及异常实例数量：

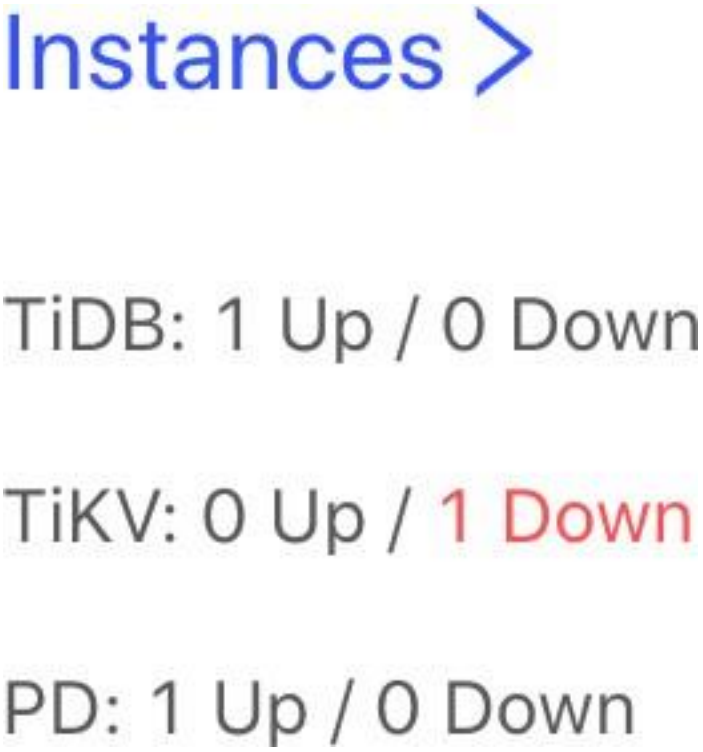


图 253: 界面

状态描述如下：

- Up：实例运行正常（含下线中的存储实例）。
- Down：实例运行异常，例如网络无法连接、进程已崩溃等。

点击实例标题可进入集群信息页面查看各个实例的详细运行状态。

11.4.7 监控和告警

该区域提供了便捷的链接方便用户查看详细监控或告警：

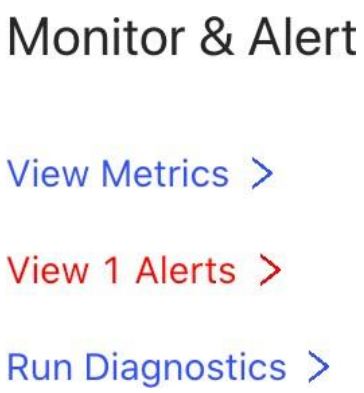


图 254: 界面

- 查看监控链接：点击后跳转至Grafana 页面，可查看集群详细监控信息。关于Grafana 监控面板中各个详细监控指标的解释，参见监控指标文档。
- 查看告警链接：点击后跳转至AlertManager 页面，可查看集群详细告警信息。当集群中已有告警时，告警数量将会直接显示在链接文本上。
- 运行诊断链接：点击后跳转至集群诊断页面，参见集群诊断页面了解详情。

> **注意：**
>
> 查看监控链接仅在集群中部署了 Grafana 节点时可用，查看告警链接仅在集群中部署了 AlertManager 节点时可用。

11.5 TiDB Dashboard 集群信息页面

该页面上允许用户查看整个集群中TiDB、 TiKV、 PD、 TiFlash 组件的运行状态及其所在主机的运行状态。

11.5.1 访问

可以通过以下两种方法访问集群信息页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击集群信息(Cluster Info)。
- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/cluster_info/instance（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

11.5.2 实例列表

点击实例(Instances)可查看实例列表：

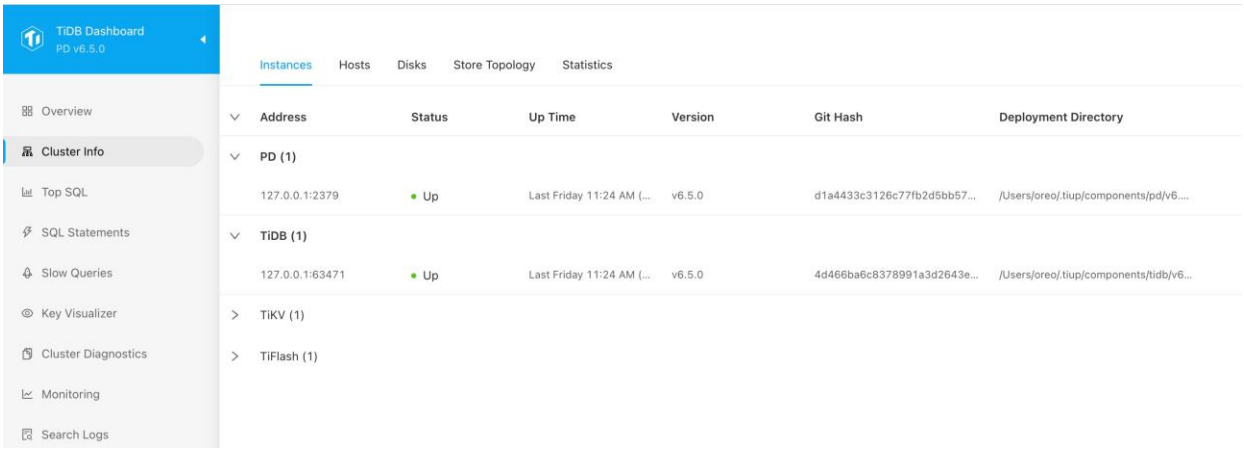


图 255: 实例

实例列表列出了该集群中TiDB、 TiKV、 PD 和 TiFlash 组件所有实例的概况信息。

表格包含以下列：

- 地址(Address)：实例地址
- 状态(Status)：实例的运行状态
- 启动时间(Up Time)：实例的启动时间
- 版本(Version)：实例版本号
- Git 哈希值(Git Hash)：实例二进制对应的 Git 哈希值
- 部署路径(Deployment Directory)：实例二进制文件所在目录路径

11.5.2.1 实例运行状态(Status)

实例可能处于如下任一运行状态：

- 在线(Up)：实例正常运行。
- 离线(Down)或无法访问(Unreachable)：实例未启动或对应主机存在网络问题。
- 已缩容下线(Tombstone)：实例上的数据已被完整迁出并缩容完毕。仅 TiKV 或TiFlash 实例存在该状态。
- 下线中(Leaving)：实例上的数据正在被迁出并缩容。仅 TiKV 或TiFlash 实例存在该状态。
- 未知(Unknown)：未知的实例运行状态。

> 注意：
>
> - TiDB Dashboard 显示的 `Leaving`、 PD API 返回的 `Offline` 以及TiUP 显示的 `Pending Offline` 这三个状态的含义相同。
> - 表格中部分列仅在实例处于在线状态(Up)时能显示。

实例运行状态来自于 PD 的调度信息。更详细的描述请参考TiDB 数据库的调度- 信息收集。

11.5.3 主机列表

点击主机(Hosts)可查看主机列表：

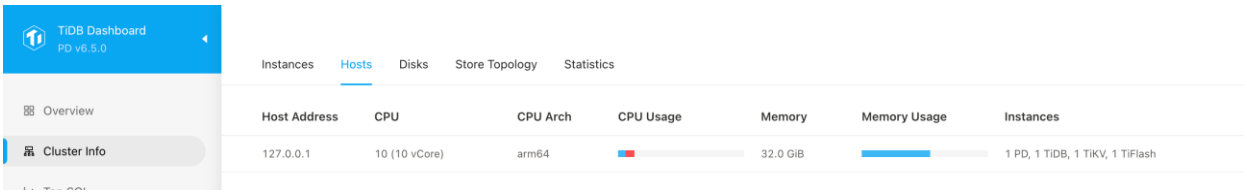


图 256: 主机

主机列表列出了该集群中TiDB、 TiKV、 PD 和TiFlash 组件所有实例对应主机的运行情况。

表格包含以下列：

- 主机地址(Host Address)：主机IP 地址
- CPU：主机 CPU 逻辑核心数
- CPU 使用率(CPU Usage)：主机当前 1 秒的用户态和内核态 CPU 使用率
- 物理内存(Memory)：主机总计的物理内存大小
- 内存使用率(Memory Usage)：主机当前内存使用率

> 注意：
>
> 主机列表信息由各个实例进程给出，因此当主机上所有实例都处于离线状态时，该主机信息将无法显示。

11.5.4 磁盘列表

点击磁盘(Disks)可查看磁盘列表：

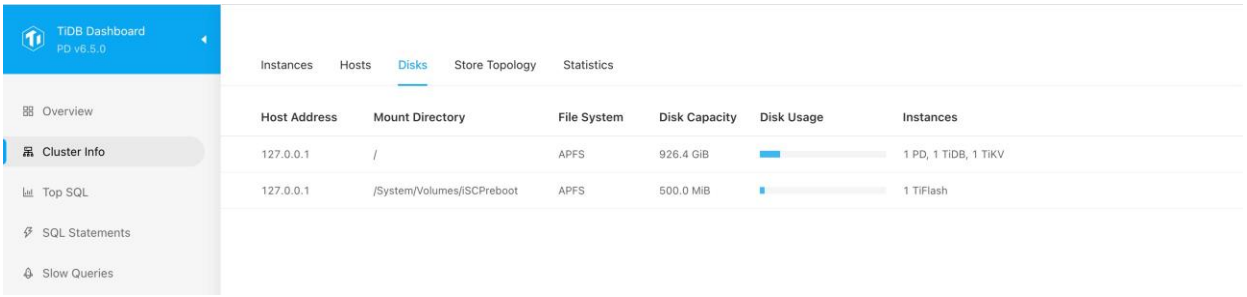


图 257: 磁盘

磁盘列表列出了该集群中TiDB、 TiKV、 PD 和 TiFlash 组件所有实例对应主机磁盘的情况。

表格包含以下列：

- 主机地址(Host Address)：主机IP 地址
- 磁盘挂载点(Mount Directory)：主机上运行实例所在磁盘的挂载路径
- 文件系统(File System)：主机上运行实例所在磁盘的文件系统类型
- 磁盘容量(Disk Capacity)：主机上运行实例所在磁盘的总空间大小
- 磁盘使用率(Disk Usage)：主机上运行实例所在磁盘的空间使用率
- 实例(Instance)：主机上运行的实例

11.6 TiDB Dashboard Top SQL 页面

TiDB Dashboard 的 Top SQL 功能允许你可视化地监控和探索数据库中各个 SQL 语句在执行过程中的 CPU 开销情况，从而对数据库性能问题进行优化和处理。Top SQL 持续收集各个TiDB 及 TiKV 实例每秒的实时 CPU 负载等数据（按 SQL 类型分别统计），并存储至多 30 天。你可以通过Top SQL 展示的图表及表格快速分析某个TiDB 或 TiKV 实例在某段时间中高 CPU 负载是来自于哪些 SQL 语句。

Top SQL 具有以下功能：

- 通过图表及表格，可视化地展示 CPU 开销最多的 5 类 SQL 语句。
- 展示每秒请求数、平均延迟、查询计划等详细执行信息。
- 支持统计所有正在执行、尚未执行完毕的 SQL 语句。
- 支持查看集群中指定TiDB 及 TiKV 实例的情况。

11.6.1 推荐适用场景

Top SQL 适用于分析性能问题。以下列举了一些典型的Top SQL 适用场景：

- 通过监控图发现集群中有个别TiKV 实例的 CPU 非常高，期望了解 CPU 热点来自于哪些 SQL 语句，以便对其进行优化、更好地利用上分布式资源。
- 集群整体 CPU 占用率非常高、数据库查询缓慢，期望快速知悉目前哪些 SQL 语句开销了最多的 CPU 资源，以便对它们进行优化。
- 集群整体 CPU 占用率突然发生了显著变化，期望了解变化前后主要的 SQL 资源开销区别。
- 分析集群当前最消耗资源的 SQL 语句情况，希望对它们进行优化以便降低硬件开支。

Top SQL 不能用于解答与性能无关的问题，例如数据正确性或异常崩溃问题。

当前Top SQL 仍然处于早期阶段，功能正在持续加强。以下列举了一些目前暂不支持的场景，供参考：

- 暂时不支持分析Top 5 以外 SQL 语句的开销情况（如多业务混合时）。
- 暂时不支持按 User、 Database 等不同维度分析Top N SQL 语句的开销情况。
- 暂时不支持分析并非由于 CPU 负载高导致的数据库性能问题，例如锁冲突。

11.6.2 访问页面

你可以通过以下任一方式访问Top SQL 页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击Top SQL。

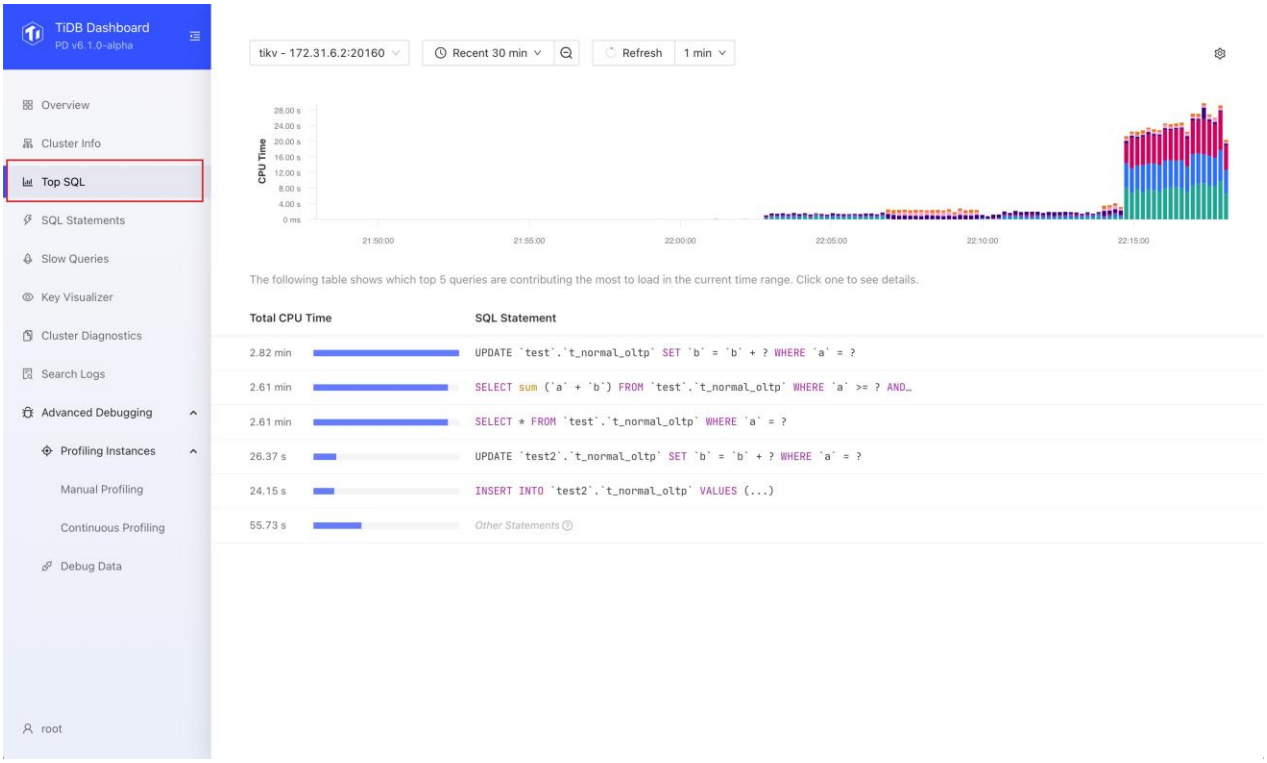


图 258: Top SQL

- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/topsql（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

11.6.3 启用Top SQL

> **注意：**
>
> 要使用Top SQL，你需要使用 TiUP（v1.9.0 及以上版本）或 TiDB Operator（v1.3.0 及以上版本）部署或升级集群。如果你已经使用旧版本TiUP 或TiDB Operator 进行了集群升级，请参见FAQ 进行处理。

Top SQL 开启后会对集群性能产生轻微的影响（平均 3% 以内），因此该功能默认关闭。你可以通过以下方法启用 Top SQL：

1. 访问Top SQL 页面。
2. 点击打开设置(Open Settings)。在右侧设置 (Settings) 页面，将启用特性(Enable Feature) 下方的开关打开。
3. 点击保存(Save)。

你仅能看到开启功能之后的 CPU 负载细节情况，在开启功能之前的 CPU 负载细节无法在界面上呈现。另外，数据有至多 1 分钟左右的延迟，因此你可能需要等待片刻才能看到数据。

除了通过图形化界面以外，你也可以配置TiDB 系统变量`tidb_enable_top_sql` 来启用Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 1;
```

11.6.4 使用Top SQL

以下是Top SQL 的常用步骤：

1. 访问Top SQL 页面。
2. 选择一个你想要观察负载的具体TiDB 或TiKV 实例。

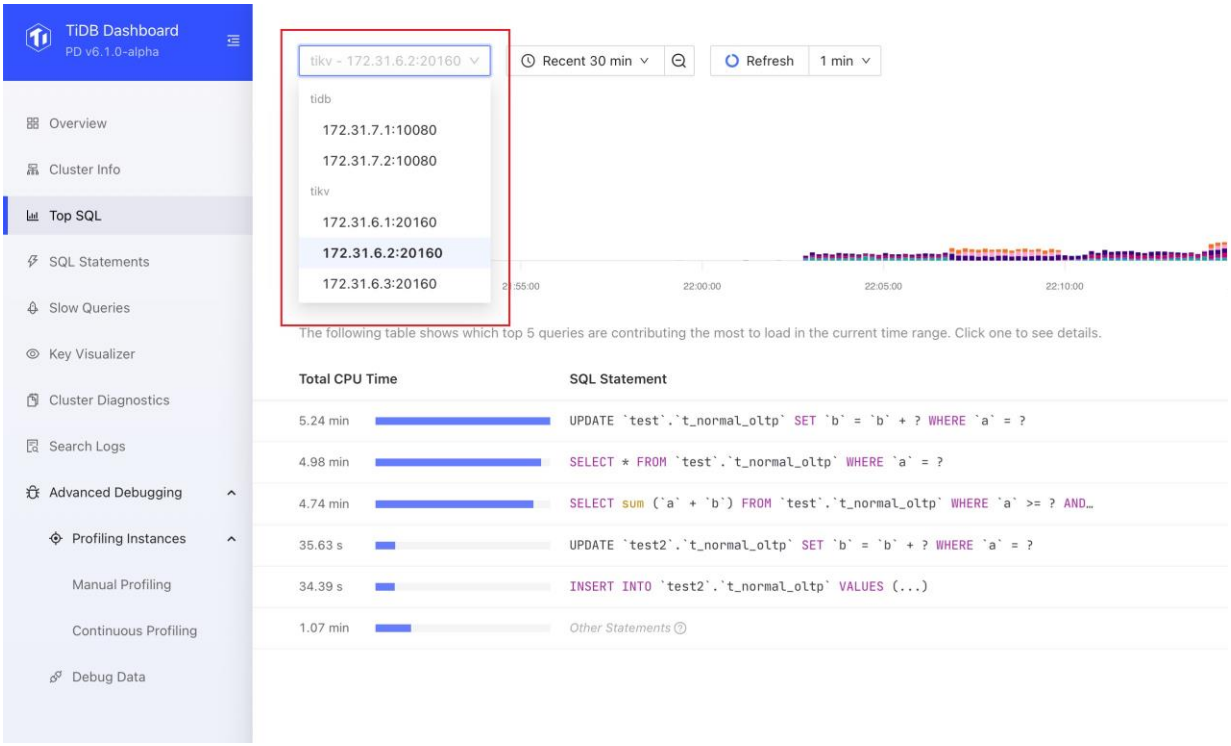


图 259: 选择实例

如果你不知道要观察哪一个TiDB 或TiKV 实例，可以选择任意一个实例。在集群 CPU 开销非常不均衡的情况下，你可以首先通过Grafana 中的 CPU 监控来确定具体期望观察的实例。

3. 观察 Top SQL 呈现的Top 5 图表及表格。

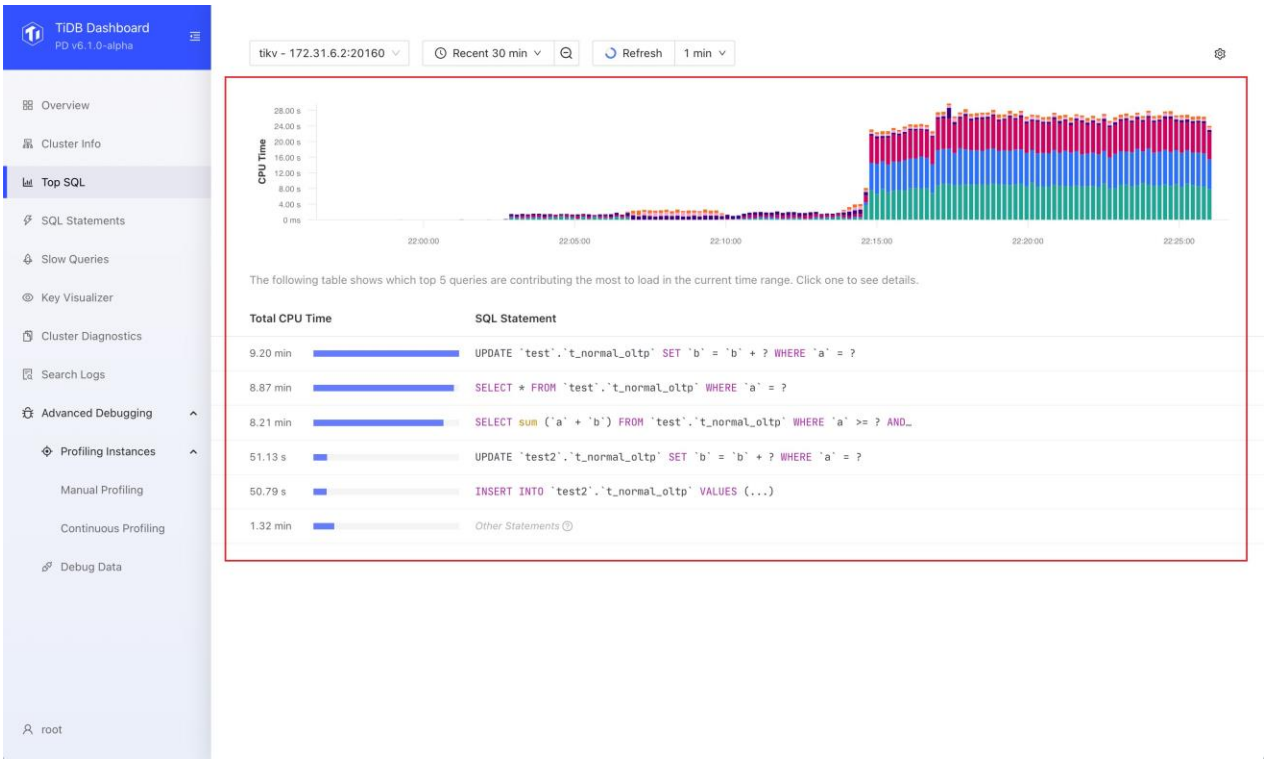


图 260: 图表表格

柱状图中色块的大小代表了 SQL 语句在该时刻消耗的 CPU 资源的多少，不同颜色区分了不同类型的 SQL 语句。大多数情况下，你都应该关注图表中相应时间范围内 CPU 资源开销较大的 SQL 语句。

4. 点击表格中的某一个 SQL 语句后，可以展开查看该语句不同执行计划的执行情况，例如 Call/sec（平均每秒请求数）、Scan Indexes/sec（平均每秒扫描索引数）等。

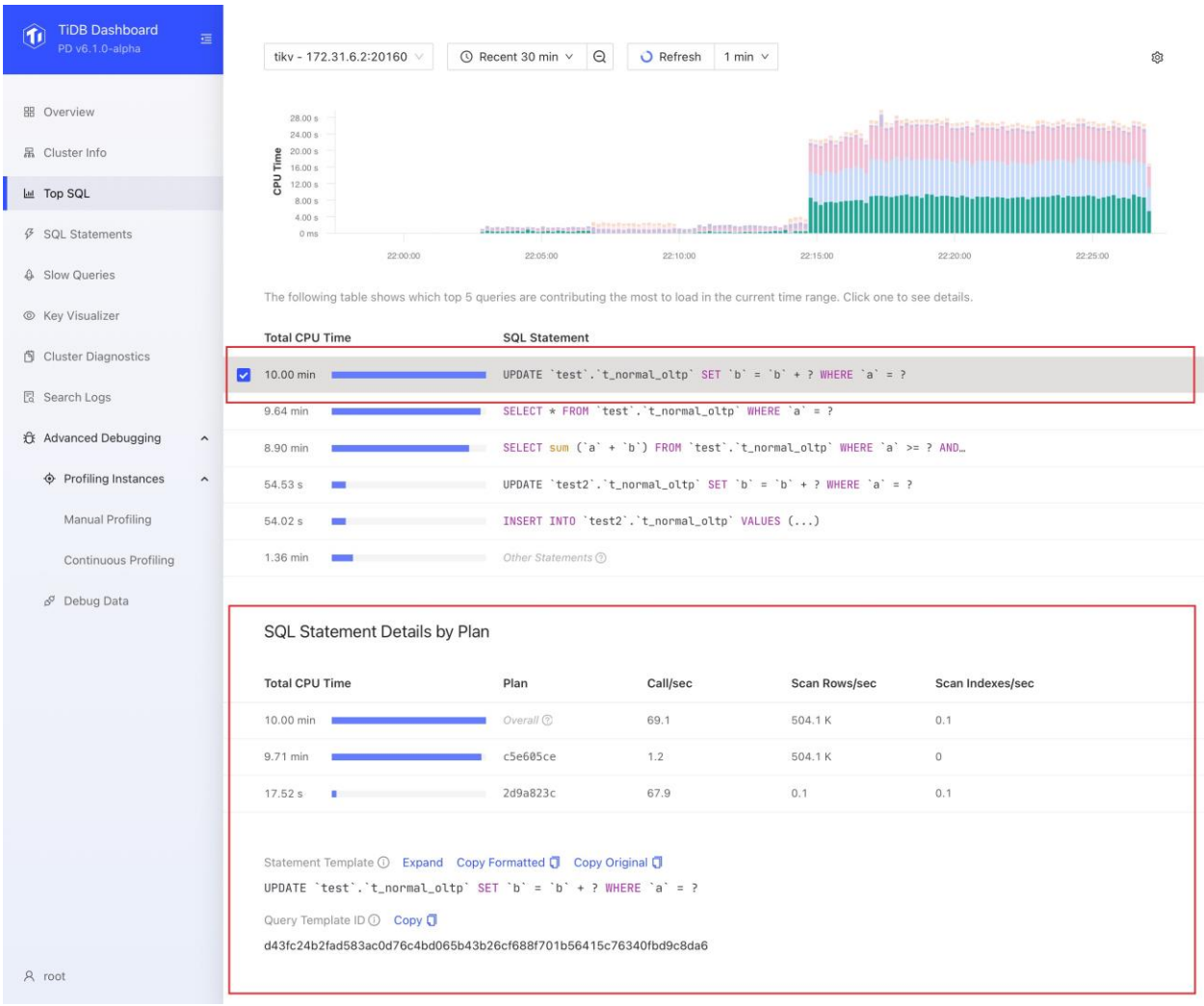


图 261: 详情

5. 基于这些初步线索，进一步在SQL 语句分析或慢查询界面中了解该 SQL 语句开销大量 CPU 资源、或扫大量数据的详细原因。

除此以外：

- 你可以在时间选择器中调整时间范围，或在图表中框选一个时间范围，来更精确、细致地观察问题。更小的时间范围将能提供细节更丰富的数据，数据最高精度可达 1 秒。

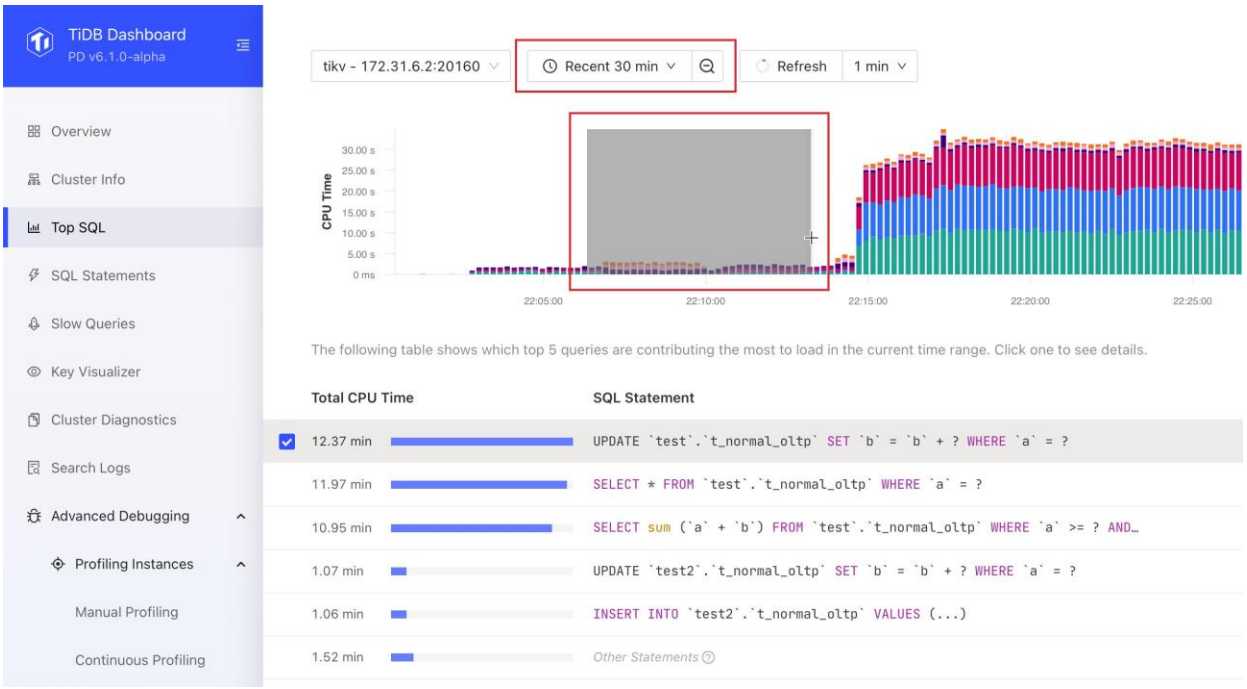


图 262: 修改时间范围

- 如果图表中显示的数据已过时，你可以点击刷新(Refresh)按钮，或在刷新(Refresh)下拉列表中选择自动刷新。

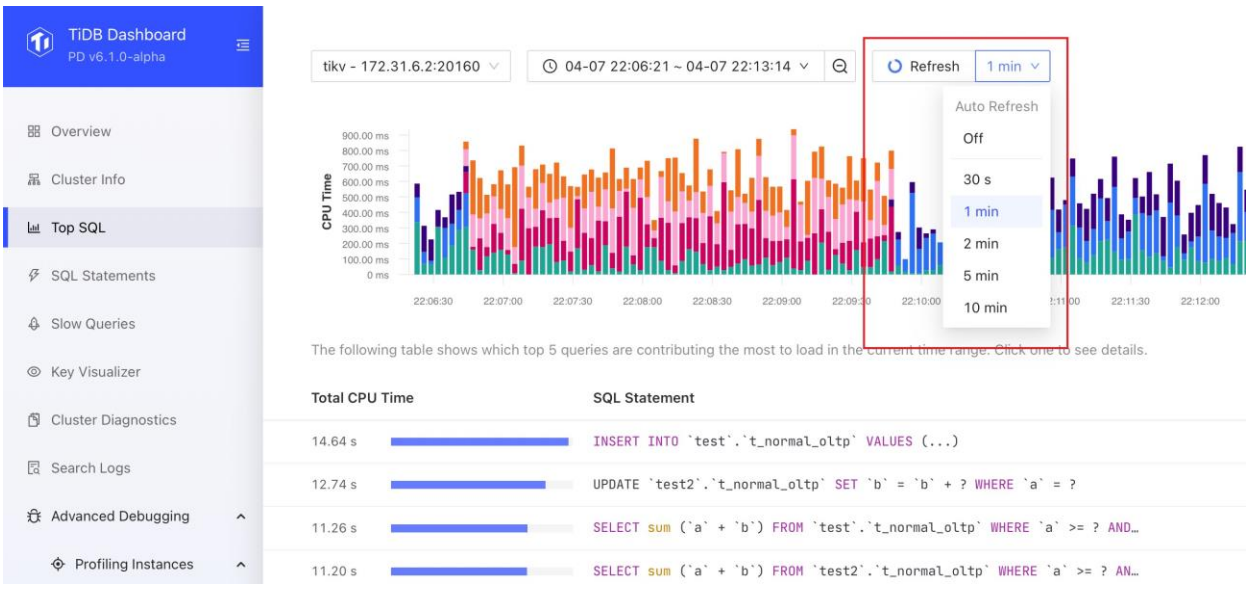


图 263: 刷新

11.6.5 停用Top SQL

你可以通过以下步骤停用该功能：

1. 访问Top SQL 页面。
2. 点击右上角齿轮按钮打开设置界面，将启用特性(Enable Feature) 下方的开关关闭。
3. 点击保存(Save)。
4. 在弹出的关闭Top SQL 功能(Disable Top SQL Feature) 对话框中，点击确认(Disable)。

除了通过图形化界面以外，你也可以配置TiDB 系统变量tidb_enable_top_sql 来停用Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 0;
```

11.6.6 常见问题

1. 界面上提示“集群中未启动必要组件 NgMonitoring”无法启用功能

请参见TiDB Dashboard FAQ。

2. 该功能开启后对集群是否有性能影响？

该功能对集群性能有轻微影响。根据我们的测算，该功能对集群的平均性能影响小于 3%。

3. 该功能目前是什么状态？

该功能是正式特性，在生产环境中可用。

4. 界面中显示的其他语句（Other Statements）是什么意思？

其他所有非Top 5 语句产生的 CPU 开销或执行情况都会被统计在该项中。你可以基于这一项了解Top 5 的 SQL 语句开销在整体所有 SQL 语句的 CPU 开销中的比例。

5. Top SQL 展示的 CPU 开销总和与进程的实际 CPU 开销是什么关系？

它们之间有很强的相关性，但不完全一致。以TiKV 为例，TiKV 的 CPU 开销还可能来自于其他副本的数据同步写入，这些开销不会被计入Top SQL。但总的来说，Top SQL 中开销比例比较大的 SQL 语句实际的 CPU 资源开销也确实会更大。

6. Top SQL 图表的纵坐标是什么意思？

代表消耗 CPU 资源的多少。消耗资源越多的 SQL 语句，该值越大。在绝大多数情况下，你都不需要关心纵坐标具体数值的含义。

7. 还没有执行完毕的 SQL 语句会被统计到吗？

会。Top SQL 图表上所展示的每一时刻 CPU 开销比例即为这一时刻所有正在运行的 SQL 语句的 CPU 开销情况。

11.7 TiDB Dashboard 流量可视化页面

流量可视化页面(Key Visualizer) 可用于分析TiDB 集群的使用模式和排查流量热点。该页面可视化地呈现了TiDB 集群一段时间的流量情况。

11.7.1 访问页面

可以通过以下两种方法访问 Key Visualizer 流量可视化页面：

- 登录TiDB Dashboard 后，点击左侧导航条的流量可视化(Key Visualizer)：

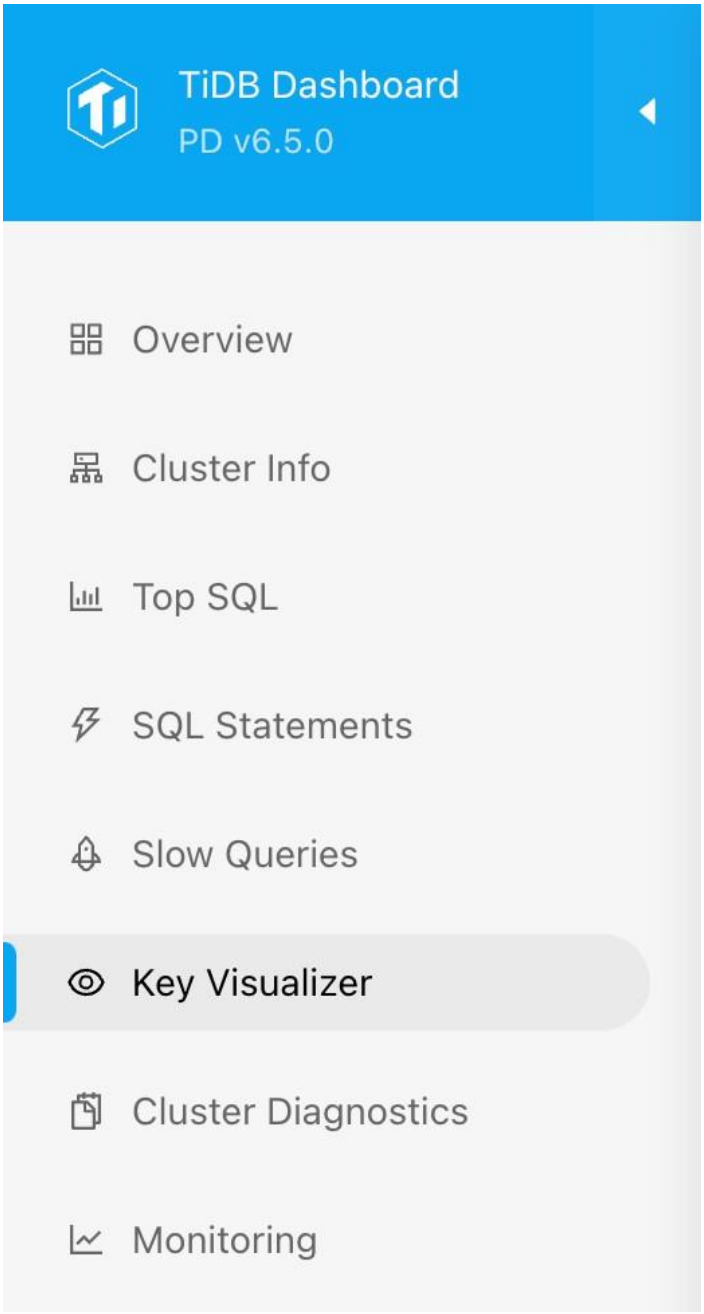


图 264: 访问

- 在浏览器中访问 `http://127.0.0.1:2379/dashboard/#/keyviz`（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

11.7.2 界面示例

流量可视化页面示例如下：

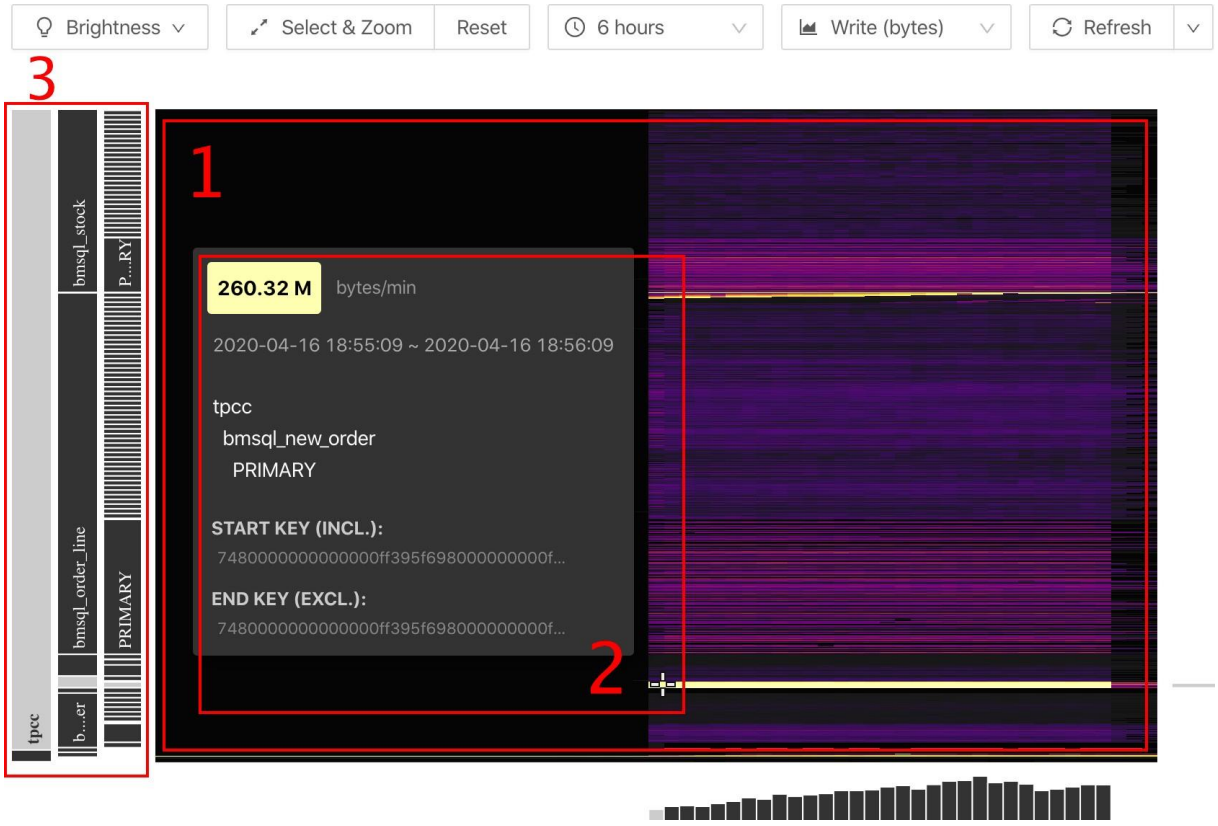


图 265: Key Visualizer 示例图

从以上流量可视化界面可以观察到以下信息：

- 一个大型热力图，显示整体访问流量随时间的变化情况。
- 热力图某个坐标的详细信息。
- 左侧为表、索引等标识信息。

11.7.3 基本概念

本节介绍流量可视化涉及的一些基本概念。

11.7.3.1 Region

在 TiDB 集群中，数据以分布式的方式存储在所有的TiKV 实例中。TiKV 在逻辑上是一个巨大且有序的 KV Map。整个 Key-Value 空间分成很多Region，每一个Region 是一系列连续的 Key。

> 注意：
>
> 关于Region 的详细介绍，请参考三篇文章了解TiDB 技术内幕- 说存储

11.7.3.2 热点

在使用TiDB 的过程中，热点是一个典型的现象，它表现为大量的流量都在读写一小块数据。由于连续的数据往往由同一个TiKV 实例处理，因此热点对应的TiKV 实例的性能就成为了整个业务的性能瓶颈。常见的热点场景有使用自增主键连续写入相邻数据导致的写入表数据热点、时间索引下写入相邻时间数据导致的写入表索引热点等。

> 注意：
>
> 热点问题详情请参阅TiDB 热点问题详解。

11.7.3.3 热力图

热力图是流量可视化页面的核心，它显示了一个指标随时间的变化。热力图的横轴X 是时间，纵轴Y 则是按Key 排序的连续Region，横跨TiDB 集群上所有数据库和数据表。颜色越暗(cold) 表示该区域的Region 在这个时间段上读写流量较低，颜色越亮(hot) 表示读写流量越高，即越热。

11.7.3.4 Region 压缩

一个TiDB 集群中，Region 的数量可能多达数十万。在屏幕上是难以显示这么多Region 信息的。因此，在一张热力图中，Region 会被压缩到约 1500 个连续范围，每个范围称为 Bucket。在一个热力图上，热的实例更需要关注，因此Region 压缩总是倾向于将流量较小的大量Region 压缩为一个 Bucket，而尽量让高流量的 Region 独立成 Bucket。

11.7.4 使用介绍

本节介绍如何使用流量可视化页面。

11.7.4.1 设置

首次使用流量可视化页面需要先通过设置页面手动开启此功能。参考页面指引，点击打开设置(Open Settings) 即可打开设置页面：

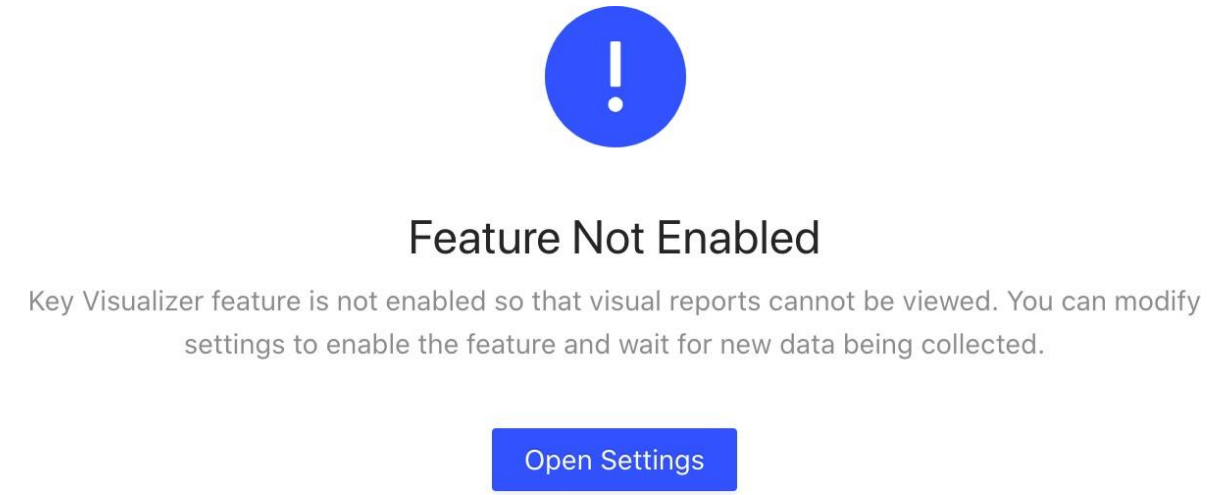


图 266: 功能未开启

在功能已经开启时，可通过右上角的设置图标打开设置页面：

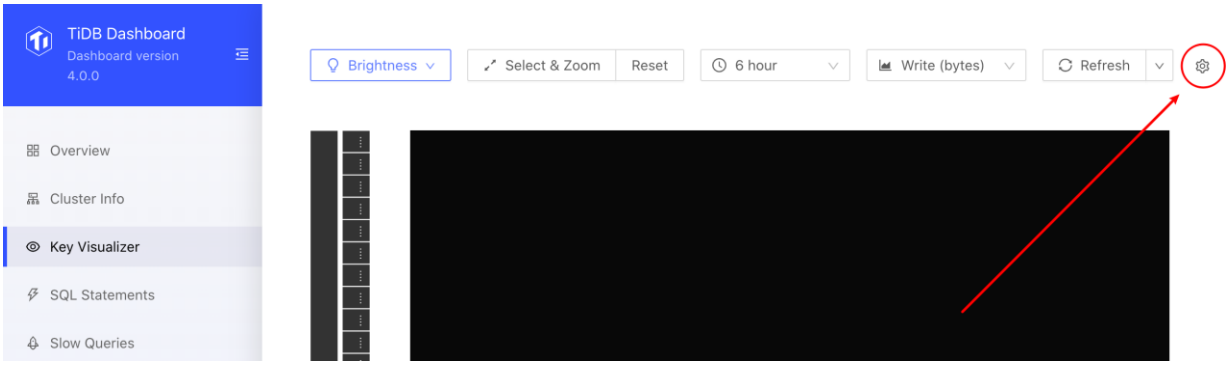


图 267: 设置按钮

设置页面如下图所示：

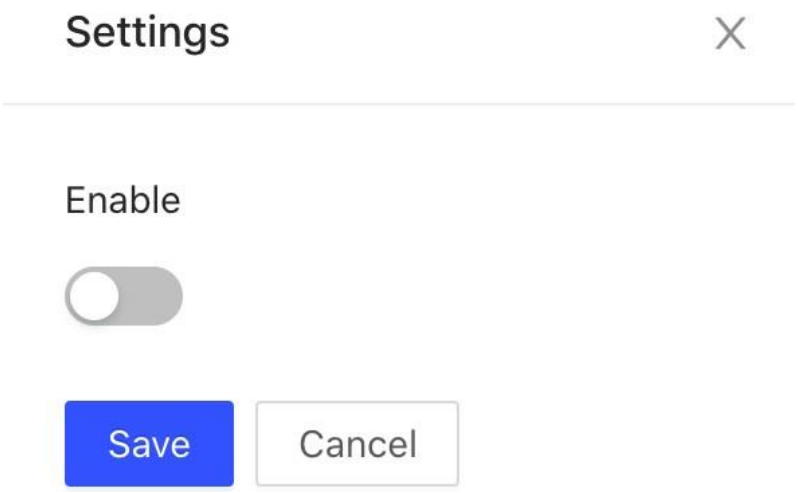


图 268: 设置页面

通过开关设定好是否开启收集，并点击保存(Save) 后生效。开启后，在界面上观察到工具栏已经可以使用：

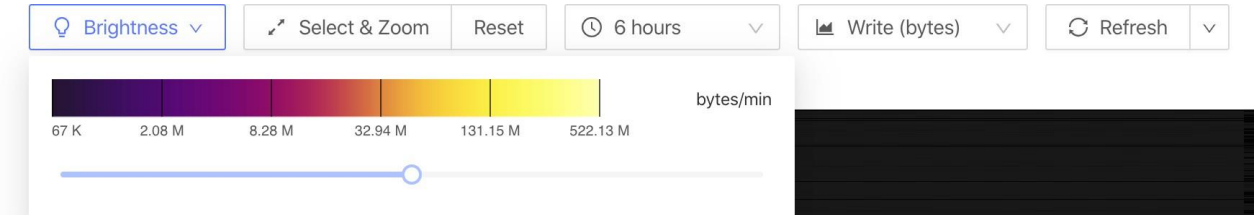


图 269: 工具栏

功能开启后，后台会持续收集数据，稍等一段时间即可看到热力图。

11.7.4.2 观察某一段时间或者Region 范围

打开流量可视化页面时，默认会显示最近六小时整个数据库内的热力图。其中，越靠近右侧（当前时间）时，每列 Bucket 对应的时间间隔会越小。如果你想观察某个特定时间段或者特定的Region 范围，则可以通过放大来获得更多细节。具体操作描述如下：

- 在热力图中向上或向下滚动。
- 点击框选(Select & Zoom) 按钮，然后点击并拖动以选择要放大的区域。

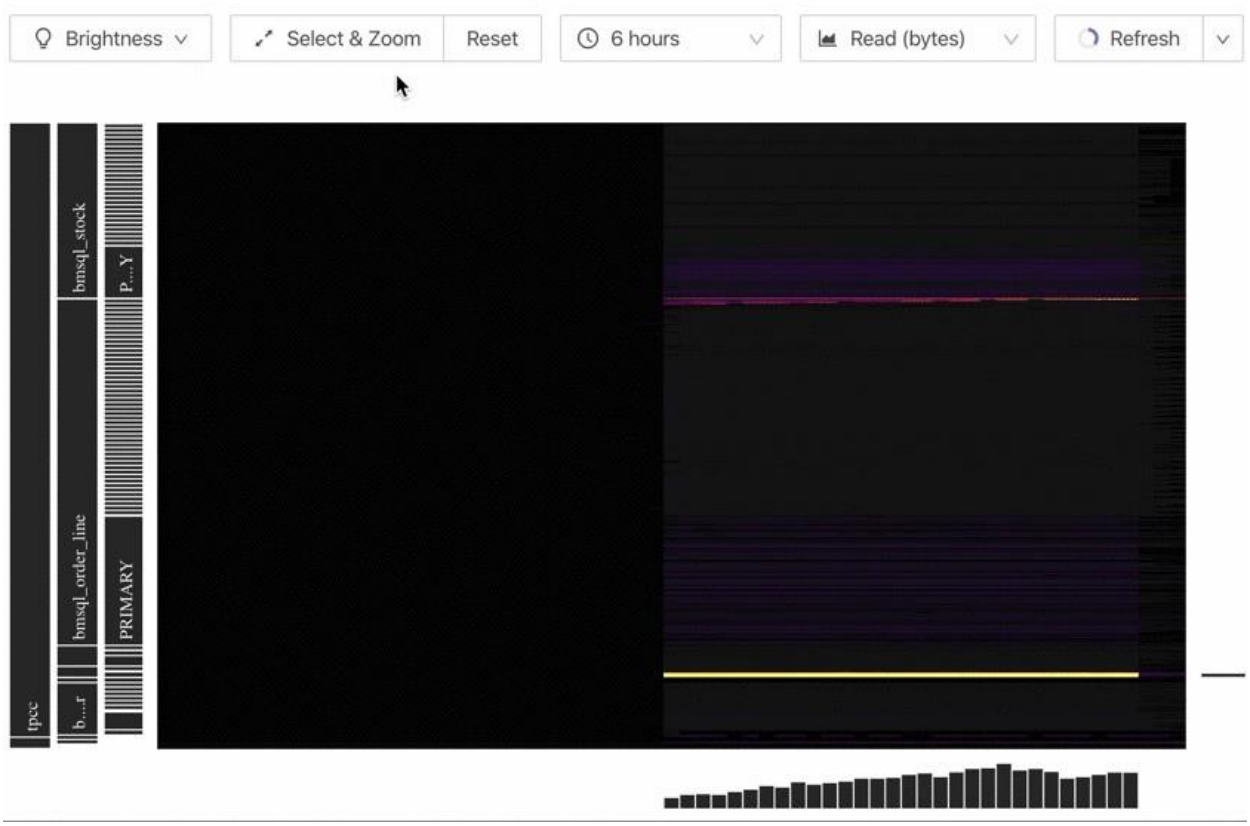


图 270: 框选

- 点击重置(Reset)按钮，将Region 范围重置为整个数据库。
- 点击「时间选择框」(界面的 6 hour 处)，重新选择观察时间段。

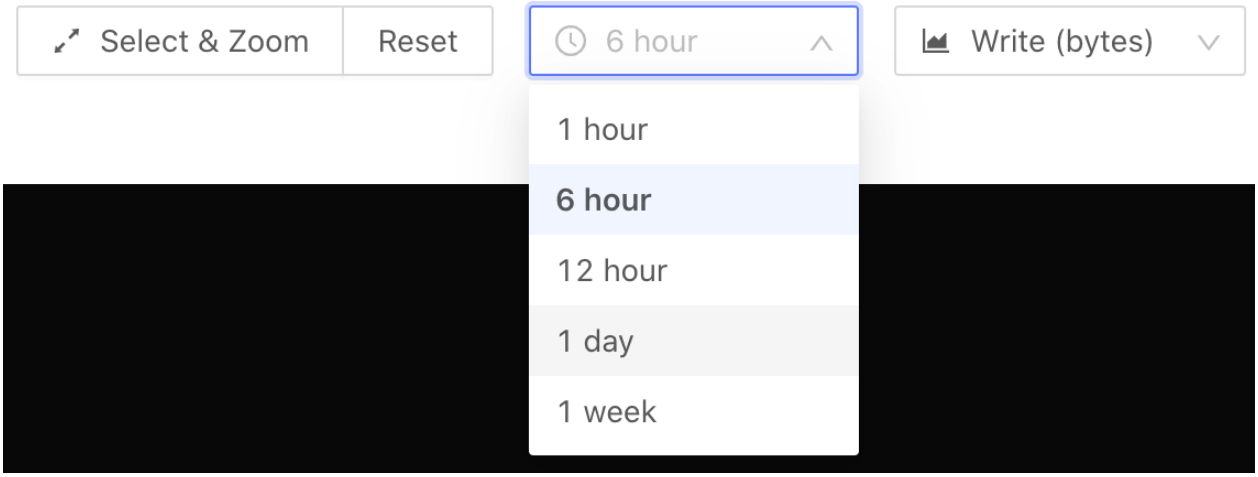


图 271: 时间选择

> 注意：
>
> 进行后三种操作，将引起热力图的重新绘制。你可能观察到热力图与放大前有较大差异。这是一个正常的现象，可能是由于在进行局部观察时，Region 压缩的粒度发生了变化，或者是局部范围内，“热”的基准发生了改变。

11.7.4.3 调整亮度

热力图使用颜色的明暗来表达一个 Bucket 的流量高低，颜色越暗 (cold) 表示该区域的Region 在这个时间段上读写流量较低，颜色越亮(hot) 表示读写流量越高，即越热。如果热力图中的颜色太亮或太暗，则可能很难观察到细节。此时，可以点击调整亮度(Brightness) 按钮，然后通过滑块来调节页面的亮度。

> 注意：
>
> 在显示一个区域内的热力图时，会根据区域内的流量情况来界定冷热。当整个区域流量较为平均时，即使整体流量在数值上很低，你依然有可能观察到较大的亮色区域。请注意一定要结合数值一起分析。

11.7.4.4 选择指标

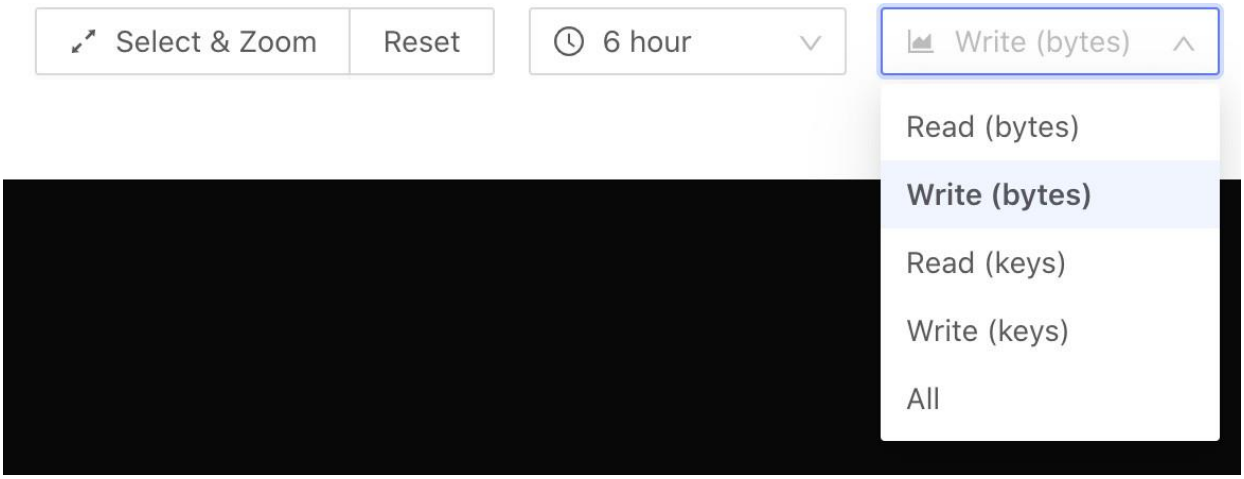


图 272: 指标选择

你可以通过「指标选择框」（以上界面中 Write (bytes) 处）来查看你关心的指标：

- `Read (bytes)`：读取字节量
- `Write (bytes)`：写入字节量

- `Read (keys)`：读取次数
- `Write (keys)`：写入次数
- `All`：所有（读写流量的总和）

11.7.4.5 刷新与自动刷新

可以通过点击刷新(Refresh)按钮来重新获得基于当前时间的热力图。当需要实时观察数据库的流量分布情况时，可以点击按钮右侧的向下箭头，选择一个固定的时间间隔让热力图按此间隔自动刷新。

> 注意：
> 如果进行了时间范围或者Region 范围的调整，自动刷新会被关闭。

11.7.4.6 查看详情

可以将鼠标悬停在你所关注的 Bucket 上，来查看这个区域的详细信息：

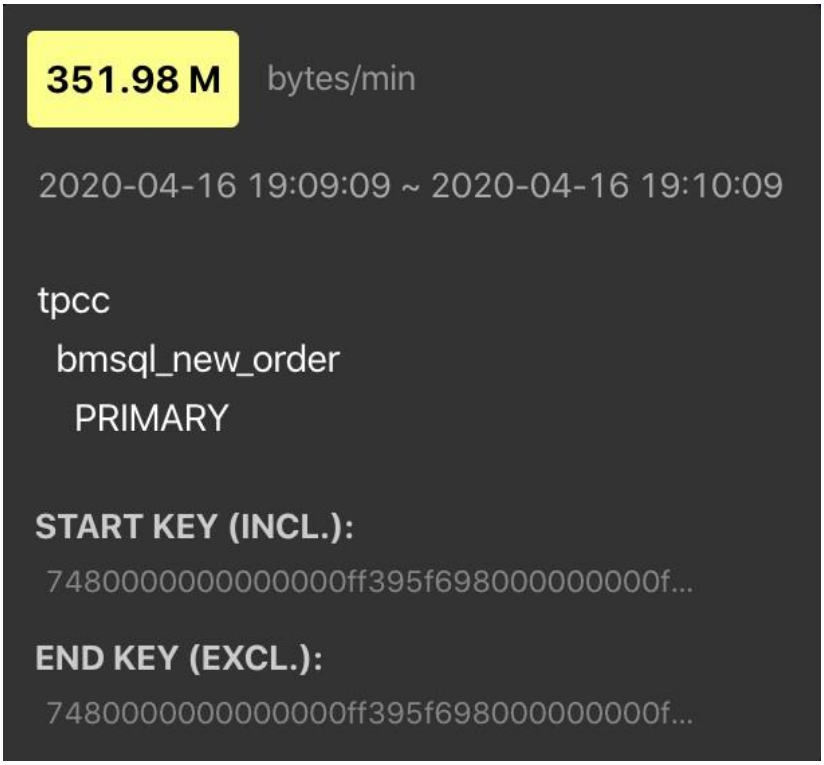


图 273: Bucket 详细信息

如果需要复制某个信息，可以进行点击 Bucket。此时相关详细信息的页面会被暂时钉住。点击你关注的信息，即可将其复制到剪切板：

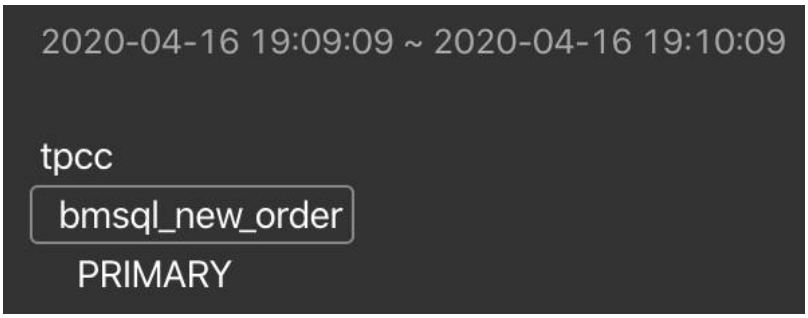


图 274: 复制 Bucket 详细信息

11.7.5 常见热力图解读

本章节选取了 Key Visualizer 中常见的四种热力图进行解读。

11.7.5.1 均衡：期望结果

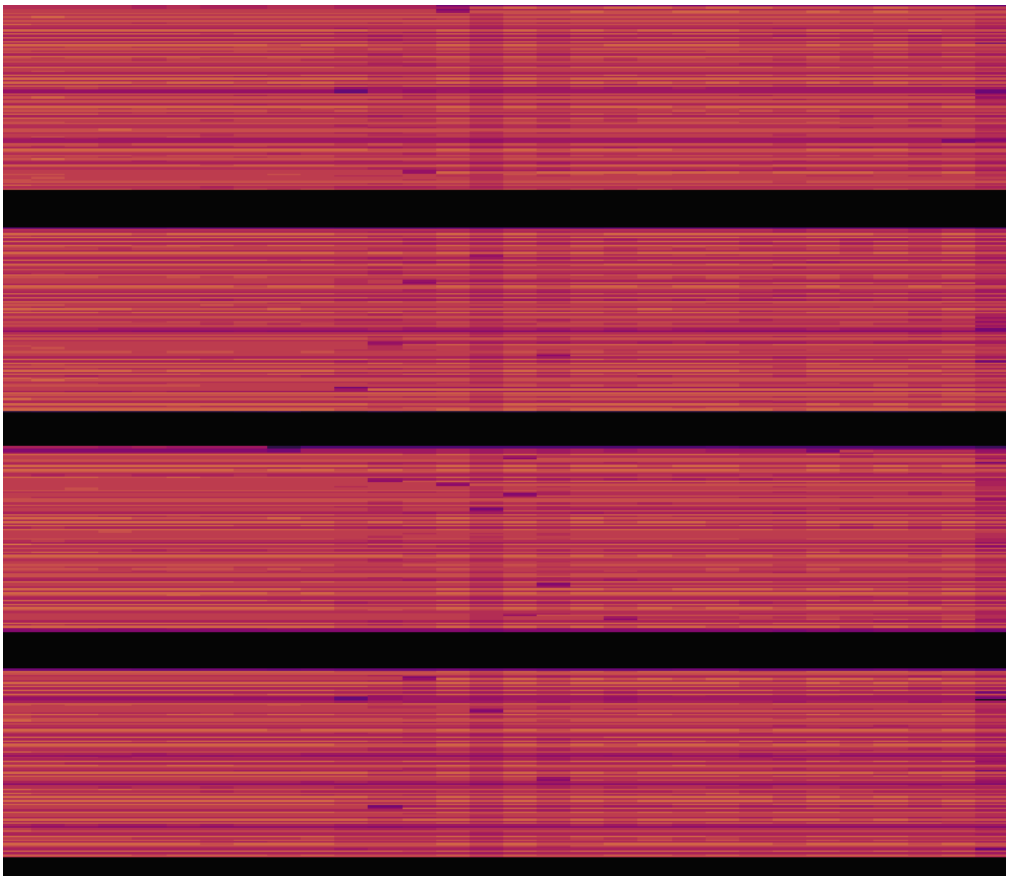


图 275: 均衡结果图

如上图所示，热力图颜色均匀或者深色和亮色混合良好，说明读取或写入在时间和Region 空间范围上都分布得比较均衡，访问压力均匀地分摊在所有的机器上。这种负载是最适合分布式数据库的。

11.7.5.2 X 轴明暗交替：需要关注高峰期的资源情况

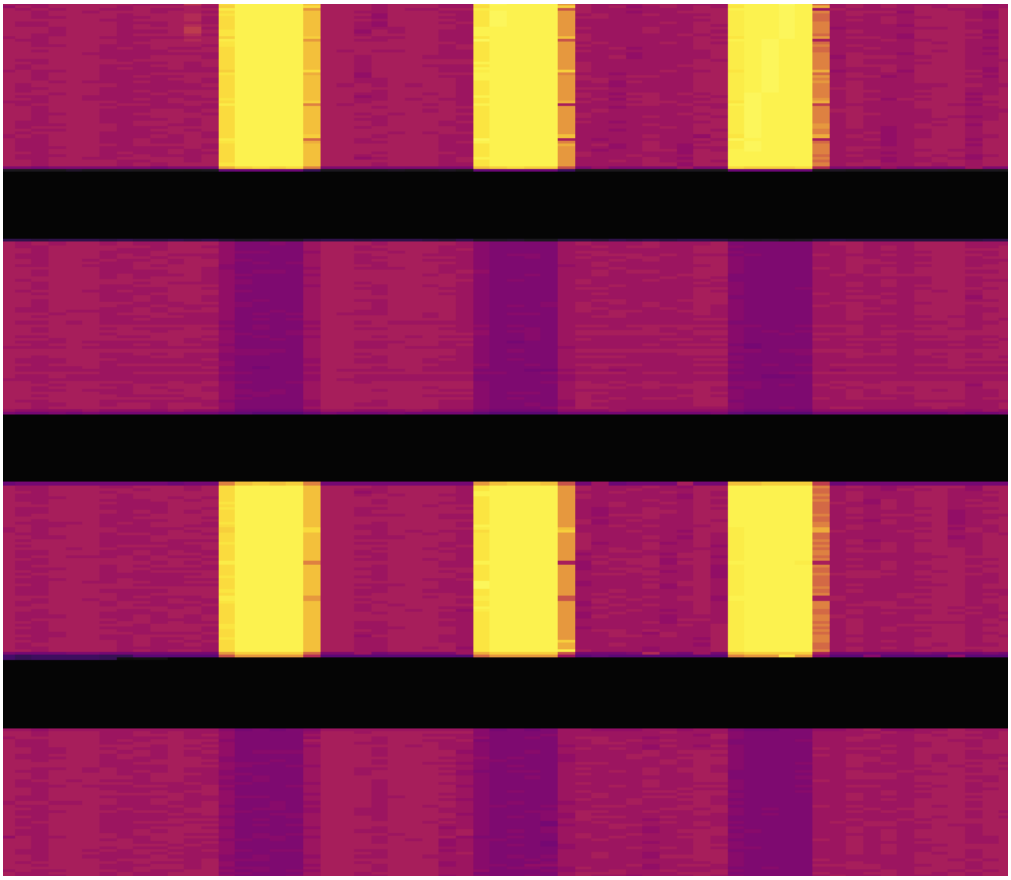


图 276: X 轴明暗交替

如上图所示，热力图在X 轴（时间）上表现出明暗交替，但Y 轴 (Region) 则比较均匀，说明读取或写入负载具有周期性的变化。这种情况可能出现在周期性的定时任务场景，如大数据平台每天定时从TiDB 中抽取数据。一般来说可以关注一下使用高峰时期资源是否充裕。

11.7.5.3 Y 轴明暗交替：需要关注产生的热点聚集程度

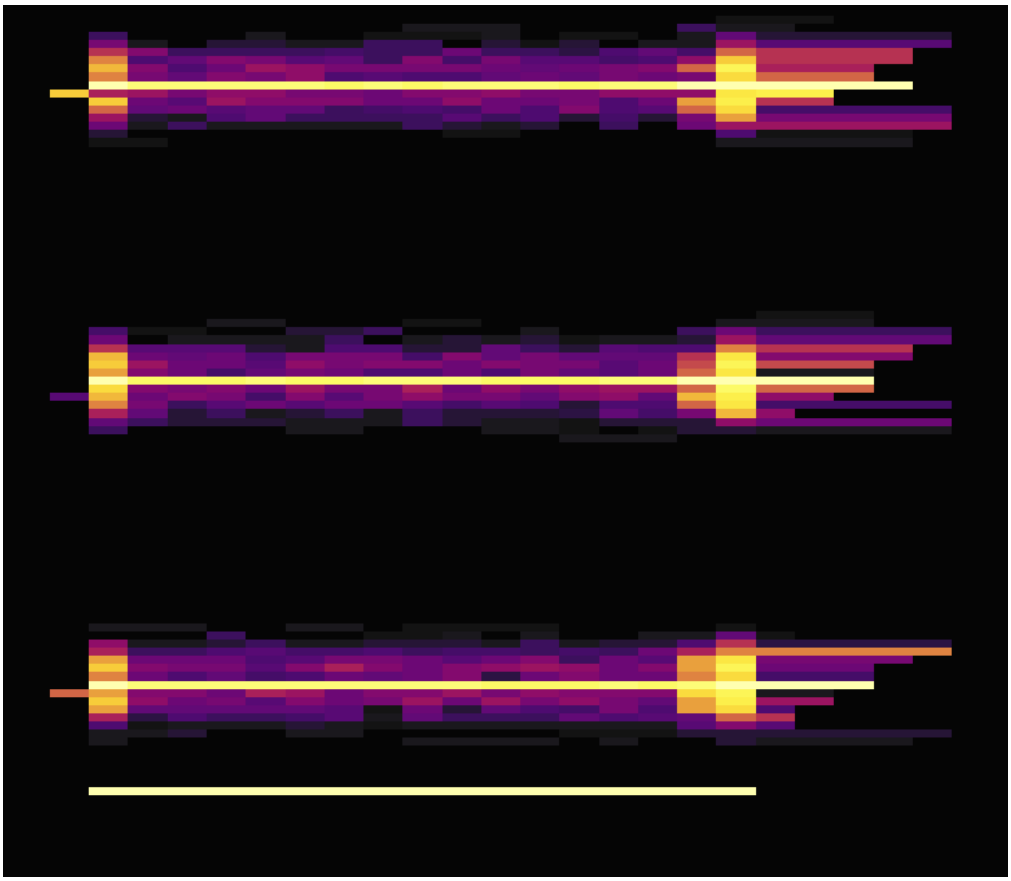


图 277: Y 轴明暗交替

如上图所示，热力图包含几个明亮的条纹，从Y 轴来看条纹周围都是暗的，这表明明亮条纹区域的Region 有很高的读写流量，可以从业务角度观察一下是否符合预期。例如，所有业务都关联用户表的情况下，用户表的整体流量就会很高，那么在热力图中表现为亮色区域就非常合理。

另外，明亮区域的高度（Y 轴方向的粗细）非常关键。由于 TiKV 自身拥有以Region 为单位的热点平衡机制，因此涉及热点的Region 越多其实越能有利于在所有TiKV 实例上均衡流量。明亮条纹越粗、数量越多则意味着热点越分散、更多的TiKV 能得到利用；明亮条纹越细、数量越少意味着热点越集中、热点TiKV 越显著、越需要人工介入并关注。

11.7.5.4 明亮斜线：需要关注业务模式

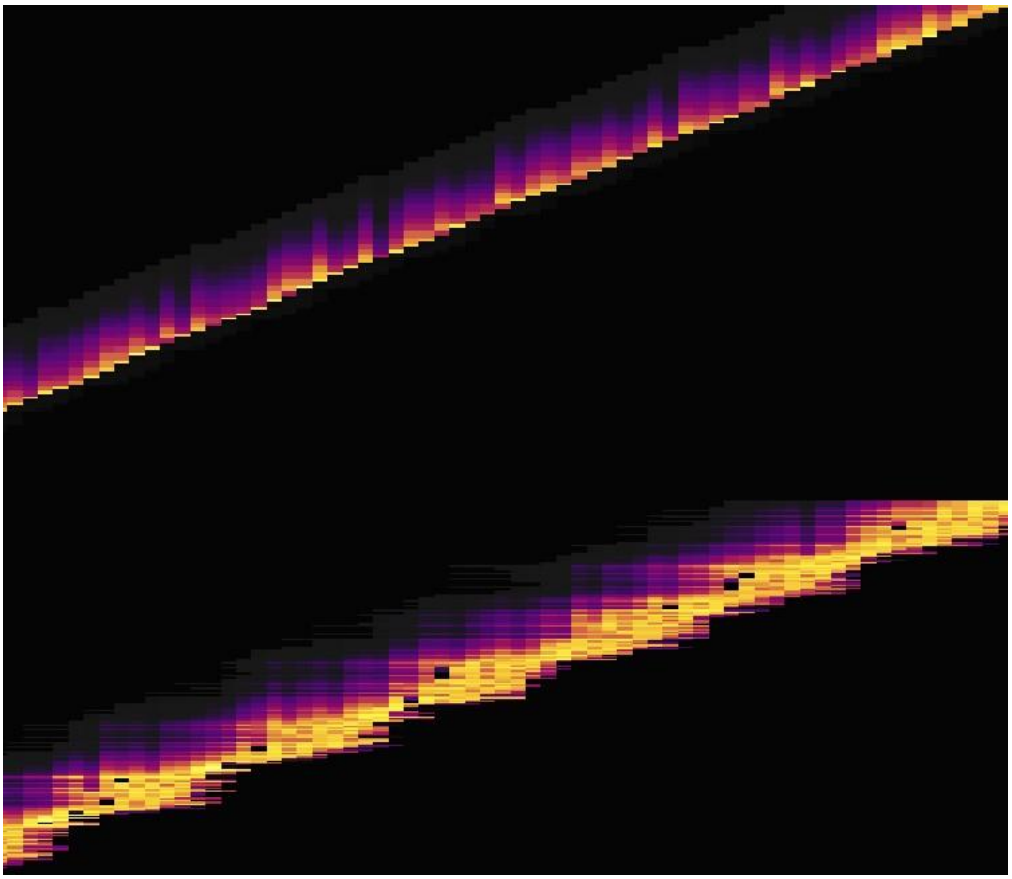


图 278: 明亮斜线

如上图所示，热力图显示了明亮的斜线，表明读写的Region 是连续的。这种场景常常出现在带索引的数据导入或者扫描阶段。例如，向自增ID 的表进行连续写入等等。图中明亮部分对应的Region 是读写流量的热点，往往会成为整个集群的性能问题所在。这种时候，可能需要业务重新调整主键，尽可能打散以将压力分散在多个Region 上，或者选择将业务任务安排在低峰期。

> **注意：**
>
> 这里只是列出了几种常见的热力图模式。流量可视化页面中实际展示的是整个集群上所有数据库、数据表的热力图，因此非常有可能在不同的区域观察到不同的热力图模式，也可能观察到多种热力图模式的混合结果。使用的时候应当视实际情况灵活判断。

11.7.6 解决热点问题

TiDB 内置了不少帮助缓解常见热点问题的功能，深入了解请参考TiDB 高并发写入场景最佳实践。

11.8 TiDB Dashboard 监控关系图

TiDB Dashboard 监控关系图是TiDB v4.0.7 起提供的新功能，可以将集群中各个内部流程的耗时监控数据绘制为关系图，帮助用户快速了解集群中各个环节的耗时及关系。

11.8.1 访问关系图

登录TiDB Dashboard 后点击左侧导航的集群诊断(Cluster Diagnostics) 可以进入此功能页面：

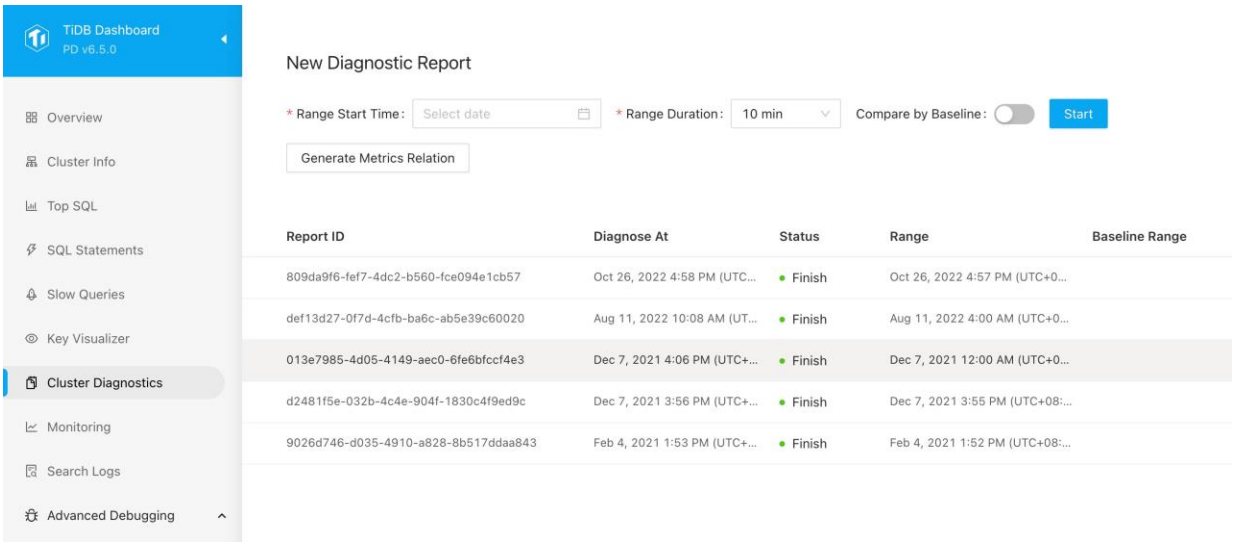


图 279: 生成监控关系图首页

设置区间起始时间(Range Start Time) 和区间长度(Range Duration) 参数后，点击生成监控关系图(Generate Metrics Relation) 按钮后，会进入监控关系图页面。

11.8.2 关系图解读

下面是一份监控耗时关系图示例，描述的是某个TiDB 集群在 2020-07-29 16:36:00 开始往后 5 分钟内，TiDB 集群中各个监控的总耗时比例，以及各项监控之间的关系。

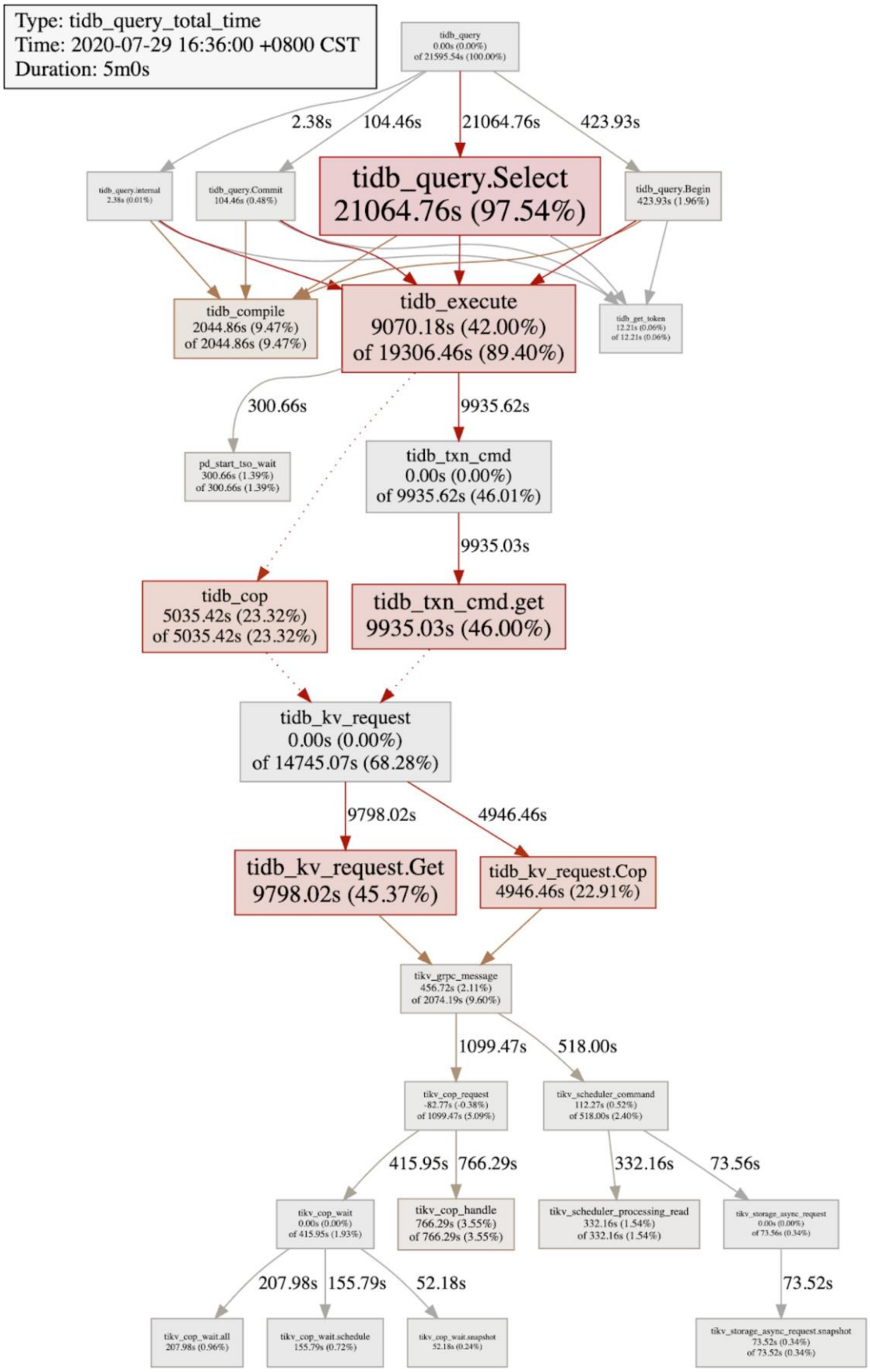


图 280: 监控关系图示例

例如以下tidb_execute 节点监控图表示：tidb_execute 监控项的总耗时为19306.46 秒，占总查询耗时的89.4%，其中 tidb_execute 节点自身的耗时是 9070.18 秒，占总查询耗时的 42%。将鼠标悬停在该方框上，可以看到监控项的注释说明，总耗时、平均耗时、平均 P99 耗时等详细信息。

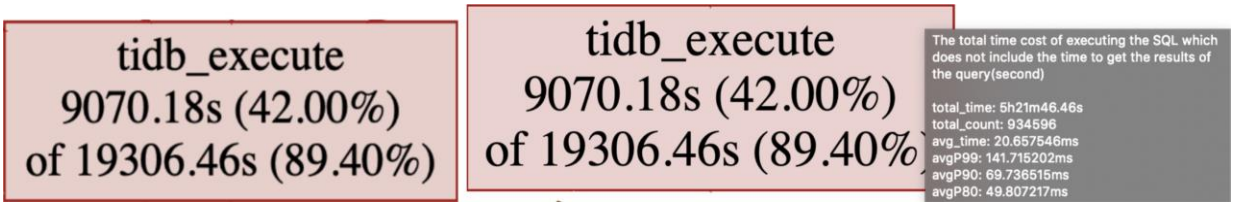


图 281: 监控关系图tidb_execute 节点示例

11.8.2.1 节点的含义

每个方框节点代表一个监控项，包含了以下信息：

- 监控项的名称
- 监控项的总耗时
- 监控项总耗时和查询总耗时的比例

节点监控的总耗时 = 节点自身的耗时 + 子节点的耗时，所以某些节点监控图会显示节点自身的耗时和总耗时的比例。例如 tidb_execute 监控：

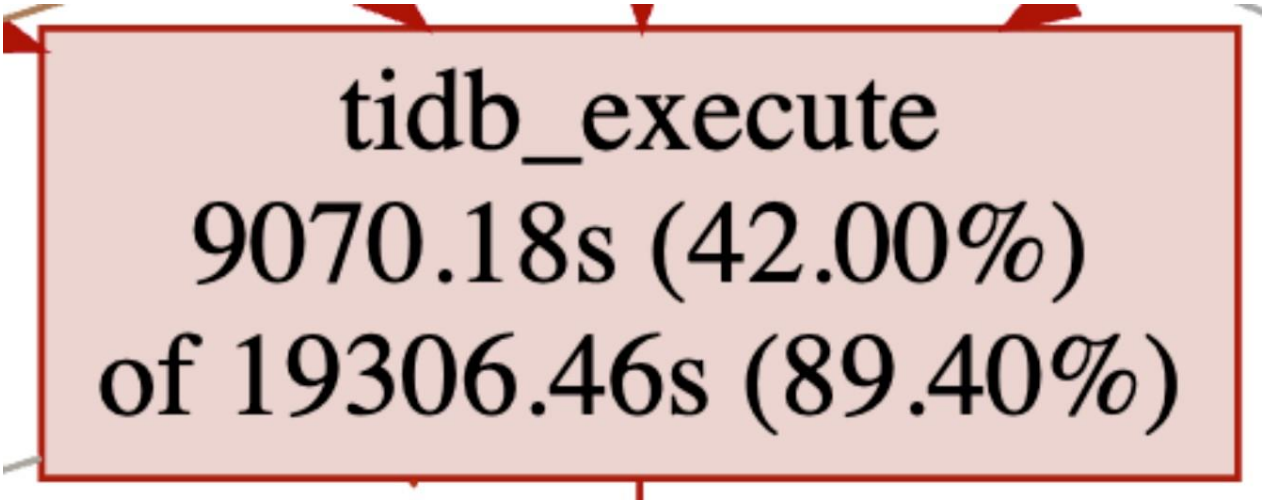


图 282: 监控关系图tidb_execute 节点示例

- tidb_execute 是监控项的名字。该监控是指一条 SQL 请求在TiDB 执行引擎中的执行耗时。
- 19306.46s 表示 tidb_execute 监控项消耗的总时间为 19306.46 秒。89.40% 表示 19306.46 秒占所有 SQL 查询总耗时（包括用户 SQL 和TiDB 内部的 SQL）的比例为 89.40%。查询总耗时是 tidb_query 监控项的总耗时。

- `9070.18s` 表示 `tidb_execute` 节点自身总执行耗时是 9070.18 秒，其余部分是被其子节点消耗的时间。`42.00%` 表示 9070.18 秒占所有查询总耗时的比例为 42.00%。

将鼠标悬停在该节点后，会显示监控项的更多详细信息：

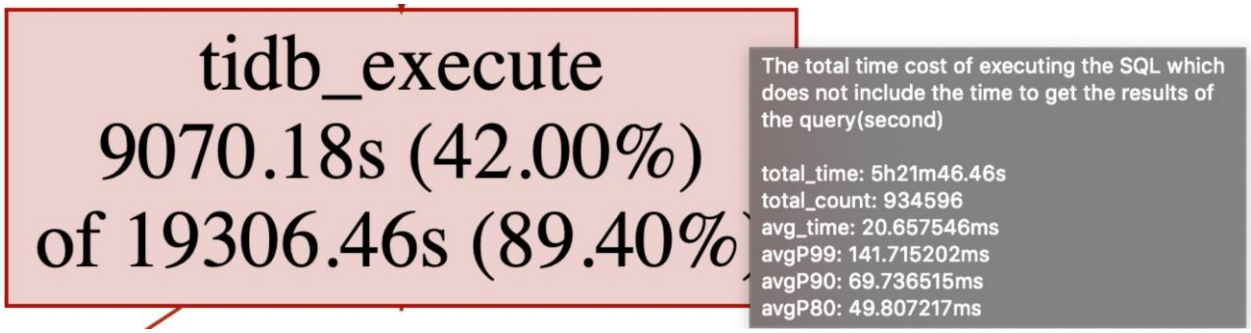


图 283: 监控关系图tidb_execute 节点注释

上图信息为该项监控的注释说明，包括总耗时、总次数、平均耗时和平均 P99、P90、P80 耗时。

11.8.2.2 监控项之间的父子关系

下面以 `tidb_execute` 监控为例介绍该监控项相关的子节点：

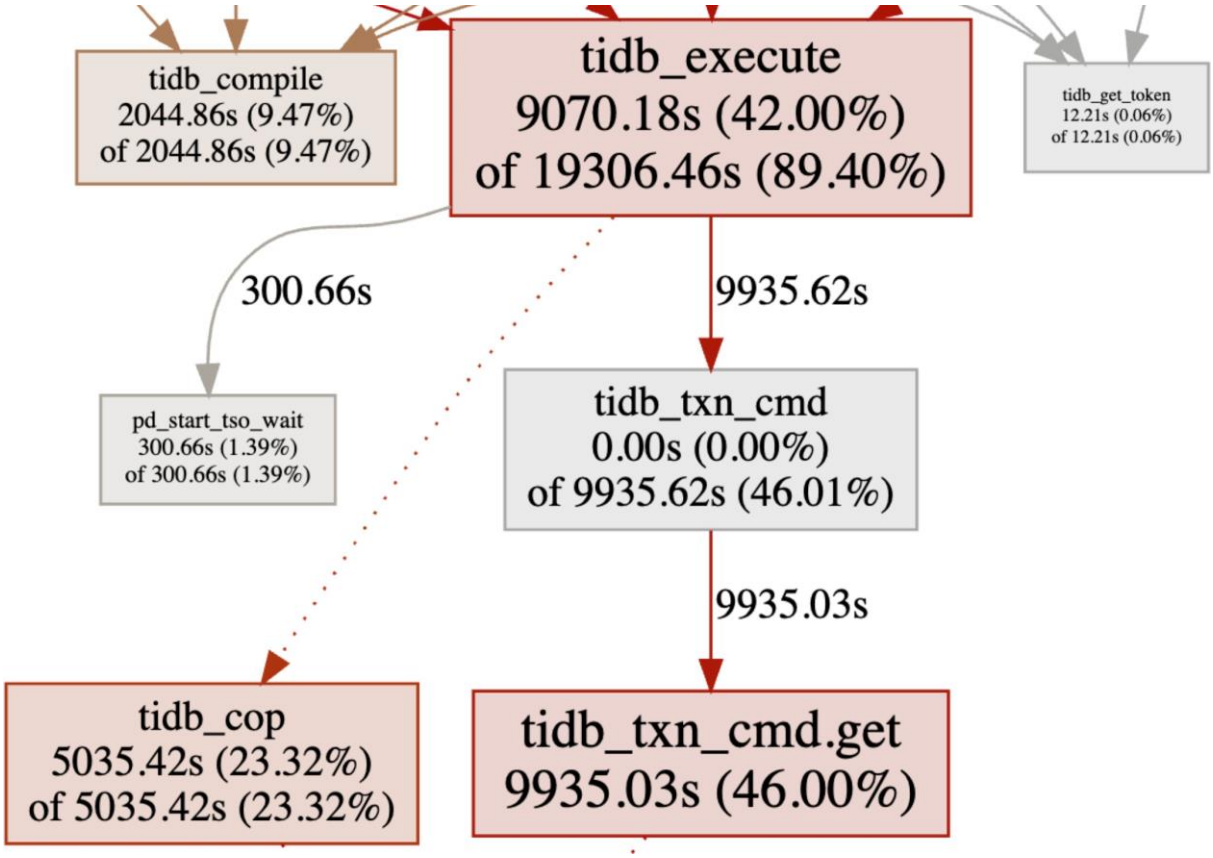


图 284: 监控关系图tidb_execute 节点注释

可以看到，tidb_execute 包含两个子节点，分别是：

- pd_start_tso_wait：等待事务 start_tso 的耗时，其总耗时是 300.66 秒。
- tidb_txn_cmd：TiDB 执行事务相关命令的耗时，其总耗时是 9935.62 秒。

另外，tidb_execute 还有一条虚线箭头指向 tidb_cop 监控，这里虚线箭头的含义如下：

tidb_execute 包含 tidb_cop 监控的耗时，但是cop 请求有可能并发执行。例如对两个表的进行 join 查询的 execute 耗时为 60 秒，其中 join 的两个表会并行地执行cop 扫表请求。假如cop 请求执行时间分别为 40 秒和 30 秒，那cop 请求的总耗时是 70 秒，但是 execute 执行耗时只有 60 秒。所以如果父节点的耗时不完全包含子节点的耗时，就用虚线箭头来指向子节点。

> 注意：
>
> 当节点有虚线箭头指向的子节点时，该节点的本身的耗时是不准确的。例如 tidb_execute 监控中，tidb_execute 节点本身的耗时为 9070.18 = 19306.46 - 300.66 - 9935.62。这里 tidb_cop 节点的耗时并不会计入子节点耗时的计算，但实际上，tidb_execute 监控本身的耗时 9070.18 秒中包含了 tidb_cop 一部分监控节点的耗时，但无法确认具体包含了多少耗时。

11.8.2.3 tidb_kv_request 及其父节点

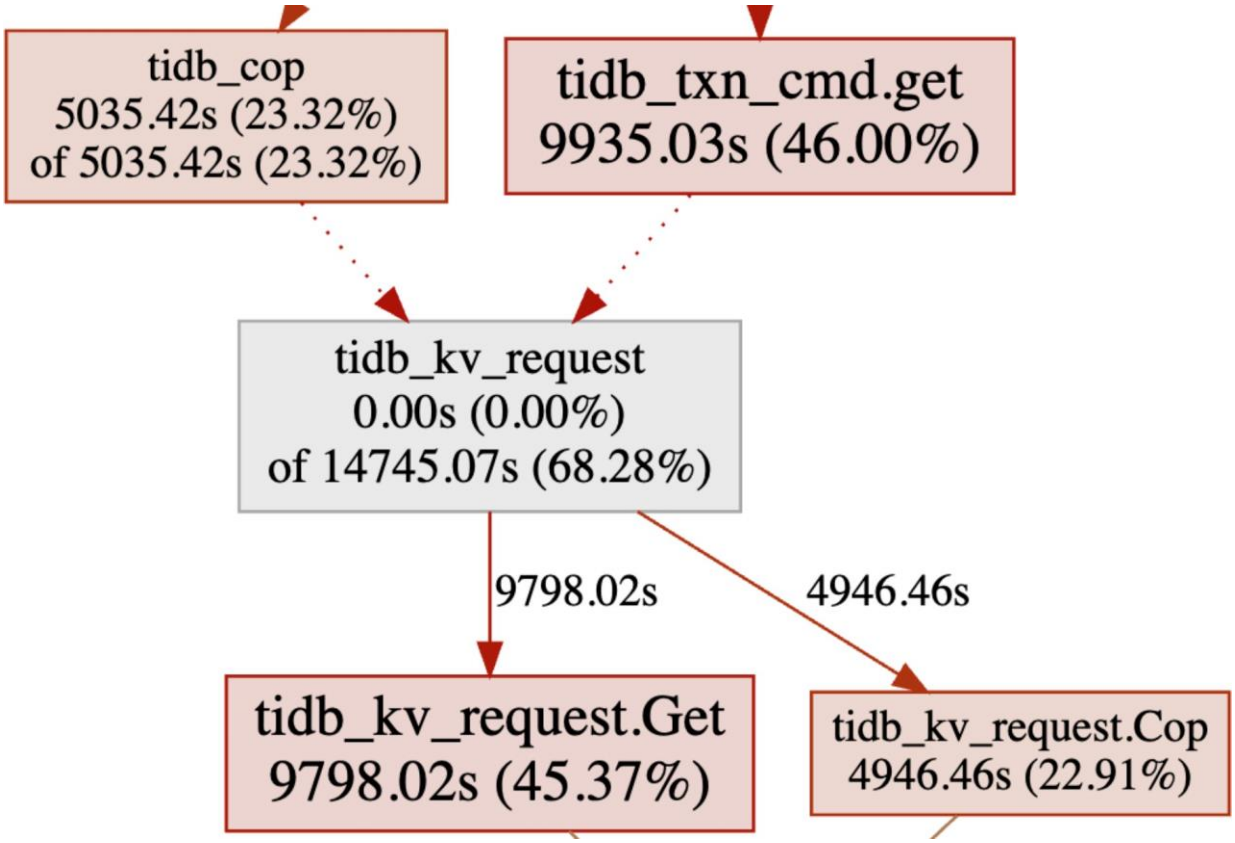


图 285: 监控关系图虚线节点关系

tidb_kv_request 的父节点 tidb_cop 和 tidb_txn_cmd.get 都用虚拟箭头指向 tidb_kv_request，这里表示：

- tidb_cop 的耗时包含部分 tidb_kv_request 的耗时
- tidb_txn_cmd.get 的耗时也包含部分的 tidb_kv_request 的耗时。

但是 tidb_cop 具体有多少耗时是 tidb_kv_request 消耗的，无法进行确认。

- tidb_kv_request.Get：TiDB 发送Get 类型的 kv 请求的耗时。
- tidb_kv_request.Cop：TiDB 发送 Cop 类型的 kv 请求的耗时。

tidb_kv_request 与 tidb_kv_request.Get 和 tidb_kv_request.Cop 并不是父节点包含子节点的关系，而是组成关系。子节点的名称前缀是父节点的名称加上 .xxx，即为父节点的子类。这里可以理解为，TiDB 发送 kv 请求的总耗时为 14745.07 秒，其中 Get 和 Cop 类型的 kv 请求的总耗时分别为 9798.02 秒和 4946.46 秒。

11.9 SQL 语句分析

11.9.1 TiDB Dashboard SQL 语句分析执行详情页面

该页面可以查看所有 SQL 语句在集群上执行情况，常用于分析总耗时或单次耗时执行耗时较长的 SQL 语句。

在该页面中，结构一致的 SQL 查询（即使查询参数不一致）都会被归为同一个 SQL 语句，例如 `SELECT * FROM employee WHERE id IN (1, 2, 3)` 和 `select * from EMPLOYEE where ID in (4, 5)` 都属于同一 SQL 语句 `select * from employee where id in (...)`。

11.9.1.1 访问列表页面

可以通过以下两种方法访问 SQL 语句分析页面：

- 登录 TiDB Dashboard 后，在左侧导航栏中点击 SQL 语句分析(SQL Statements)。
- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/statement （将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

SQL 语句分析页面所展示的所有数据都来自于 TiDB Statement 系统表，参见 Statement Summary Tables 文档了解该系统表的详细情况。

> **注意：**
>
> 在 SQL 语句分析页面的 Mean Latency 列中，蓝色条表示平均耗时。如果某条 SQL 语句的蓝色条上有黄色的线，黄色线的左右两侧分别代表最近一个收集周期内该 SQL 语句耗时的最小值和最大值。

11.9.1.1.1 修改列表过滤条件

页面顶部可修改显示的时间范围、按执行 SQL 所在数据库进行过滤，或按 SQL 类型进行过滤，如下所示。默认显示最近一个收集周期（默认最近 30 分钟）内的所有 SQL 语句执行情况。

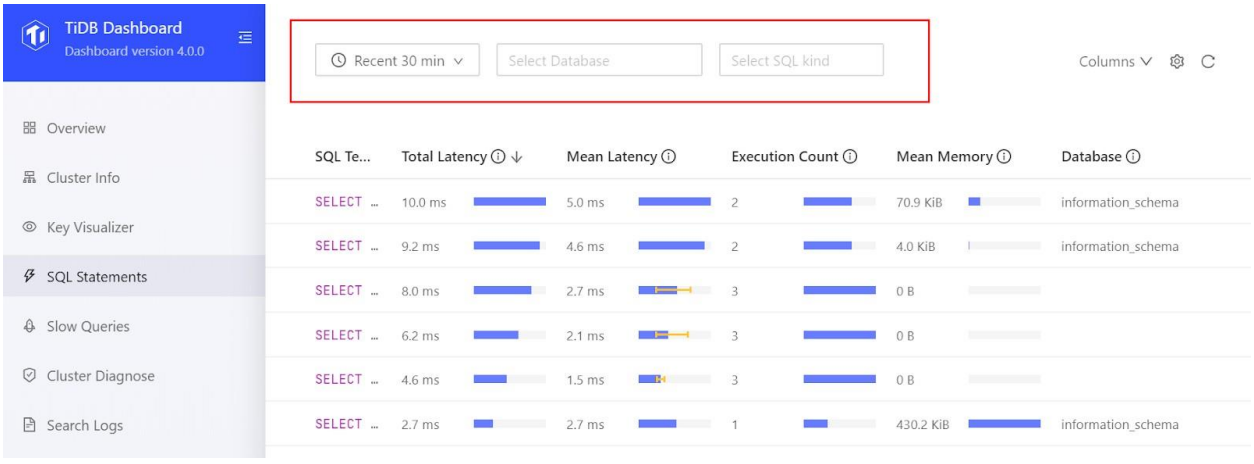


图 286: 修改过滤条件

11.9.1.1.2 显示其他字段

页面顶部选择列(Columns)选项中可选择显示更多列，可将鼠标移动到列名右侧的(i)图标处查看列的说明：

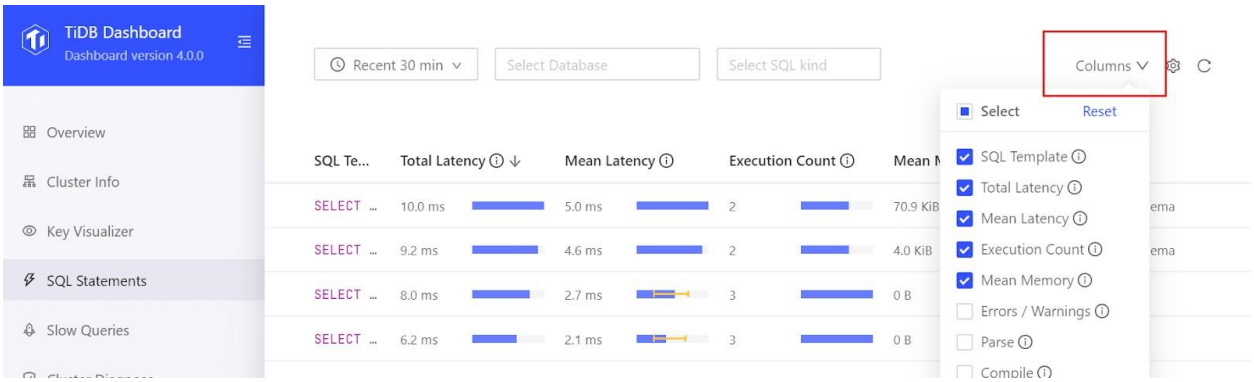


图 287: 选择列

11.9.1.1.3 修改列表排序依据

列表默认以累计耗时(Total Latency) 从高到低进行排序，点击不同的列标题可以修改排序依据或切换排序顺序：

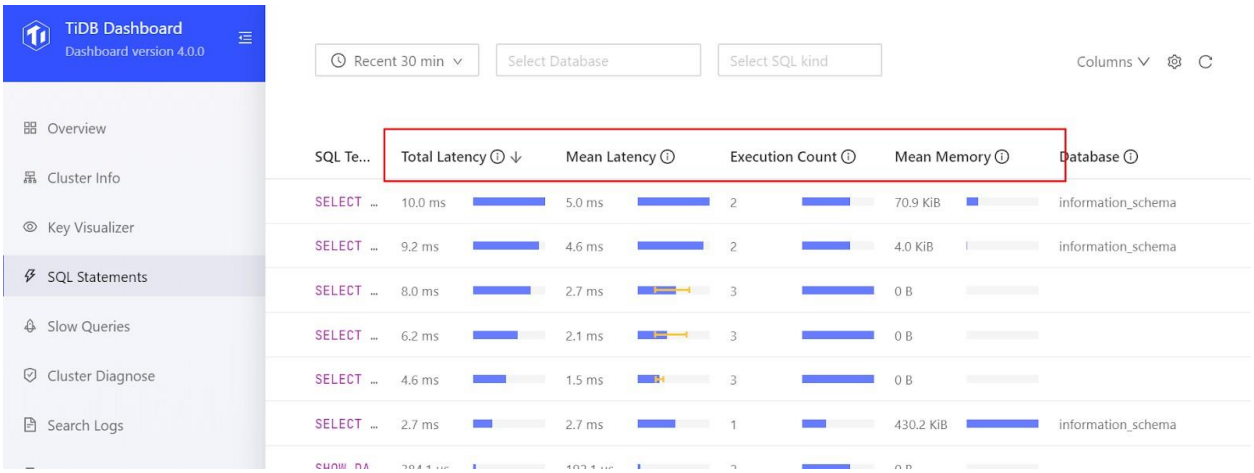


图 288: 修改列排序

11.9.1.1.4 修改数据收集设置

在列表页面，点击顶部右侧的设置(Settings)按钮，即可对 SQL 语句分析功能进行设置：

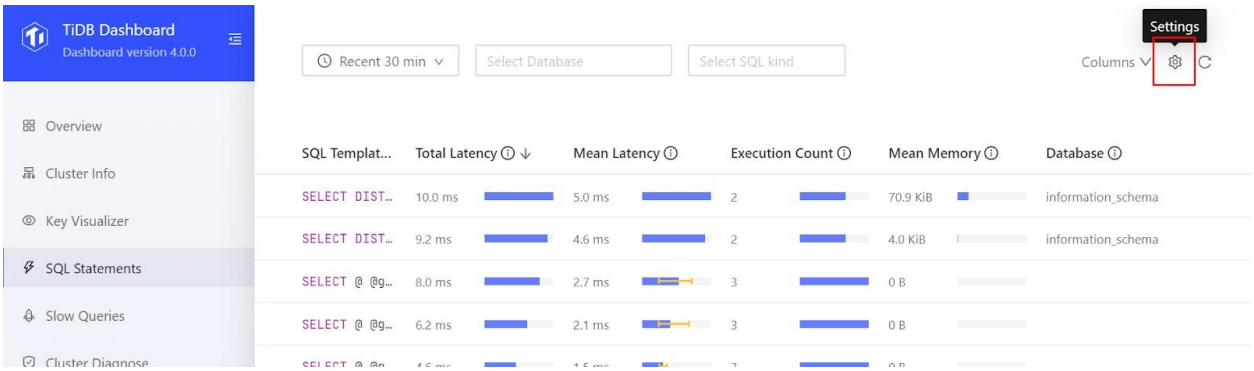


图 289: 设置入口

点击后设置界面如下图所示：

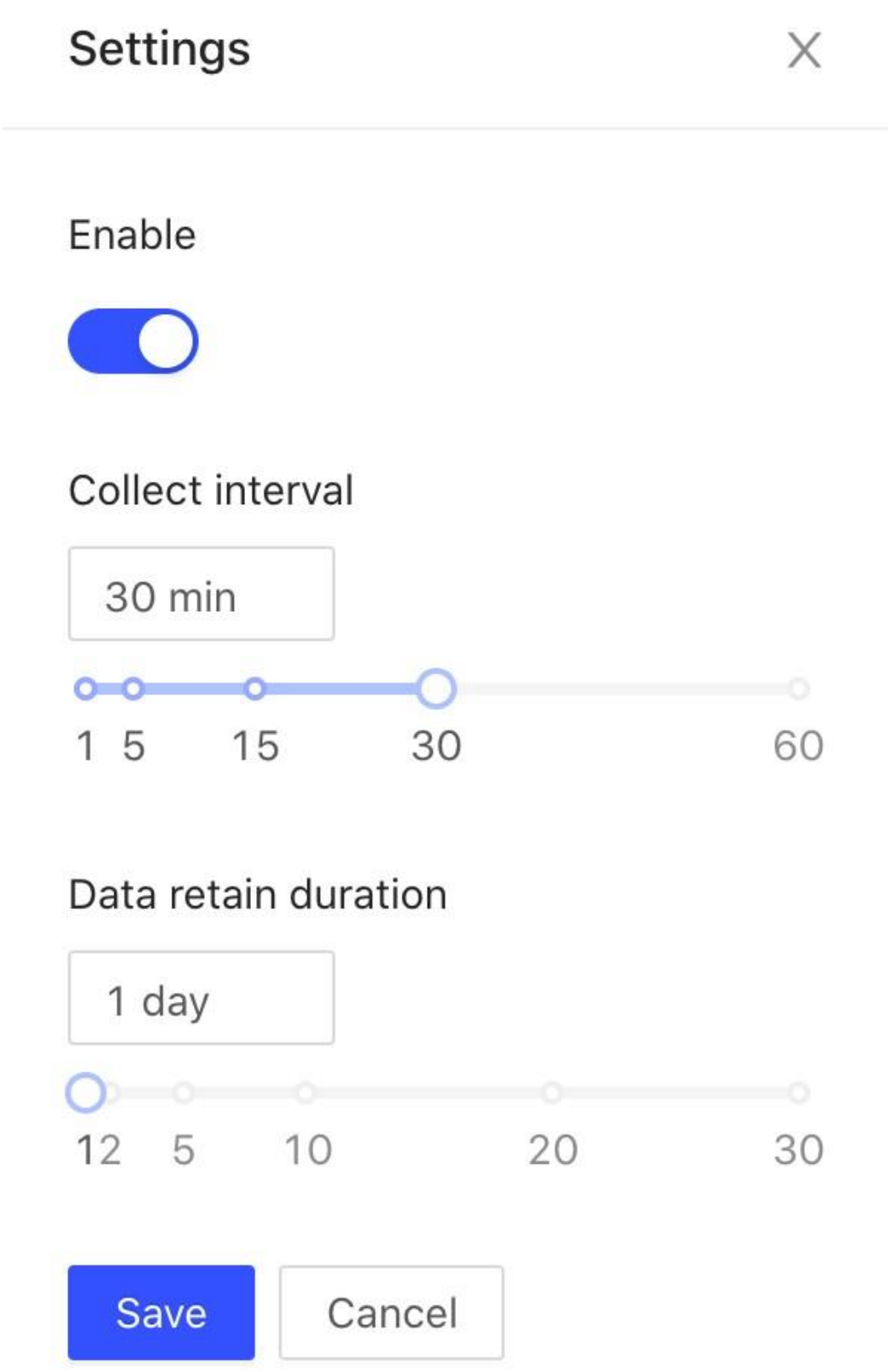

Settings
Enable
Collect interval
30 min
1 5 15 30 60
Data retain duration
1 day
12 5 10 20 30
Save
Cancel

图 290: 设置

在设置中可以选择关闭或开启 SQL 语句分析功能。在开启 SQL 语句分析功能时可以修改以下选项：

- 数据收集周期：默认 30 分钟，每次进行 SQL 语句分析的时间长度。SQL 语句分析功能每次对一段时间范围内的所有 SQL 语句进行汇总统计。

  如果这个时间范围过长，则统计的粒度粗，不利用定位问题；如果太短，则统计的粒度细，方便定位问题，但会导致在相同的数据保留时间内产生更多的记录，产生更多的内存占用。因此需要根据实际情况调整，在需要定位问题时适当地将值调低。

- 数据保留时间：默认 1 天，统计信息保留的时间，超过这个时间的数据会被从系统表中删除。

参见Statement Summary Tables 参数设置文档了解详细情况。

> 注意：
>
> 由于 Statement 系统表只存放在内存中，关闭此功能后，系统表中的数据会将清空。
>
> 数据收集周期和保留时间的值会影响内存占用，因此建议根据实际情况调整，保留时间不宜设置过大。

11.9.1.1.5 Others

`tidb_stmt_summary_max_stmt_count` 控制 Statement Summary 系统表保存的 SQL 种类数量。当 SQL 种类超过该值时，会移除最近没有出现的 SQL。这些 SQL 将会被 `DIGEST` 为 `NULL` 的行数据统计。`DIGEST` 为 `NULL` 的行数据在 TiDB Dashboard SQL 语句分析列表页面中将会显示为 `Others`，如下所示：

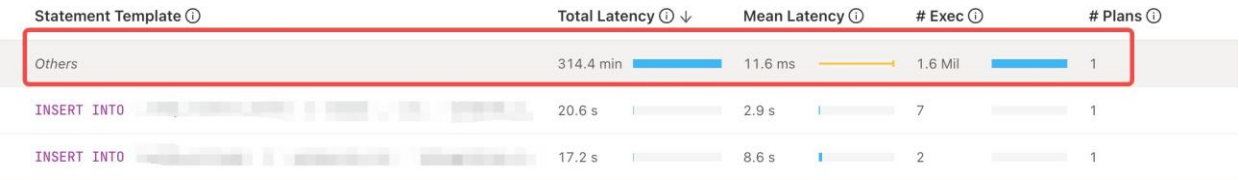


图 291: Others 行

11.9.1.2 下一步

阅读查看执行详情章节了解如何进一步查看 SQL 语句的详细执行情况。

11.9.2 TiDB Dashboard SQL 语句分析执行详情页面

在列表中点击任意一行可以进入该 SQL 语句的详情页查看更详细的信息，此信息包括三大部分：

- SQL 语句概况：包括 SQL 模板、SQL 模板 ID、当前查看的时间范围、执行计划个数、执行所在的数据库以及快速绑定执行计划功能（下图区域 1）

- 执行计划列表：如果一个 SQL 语句有多个执行计划，则显示执行计划列表。除了文本信息显示，TiDB 自 v6.2.0 开始引入图形化执行计划，通过图形化的执行计划，你可以更清晰地了解一个语句的具体算子和对应的内容。可以选择不同的执行计划，在列表和图形下方会显示该执行计划详情（下图区域 2）
- 执行计划详情：显示选中的执行计划的详细信息，具体见下一小节（下图区域 3）

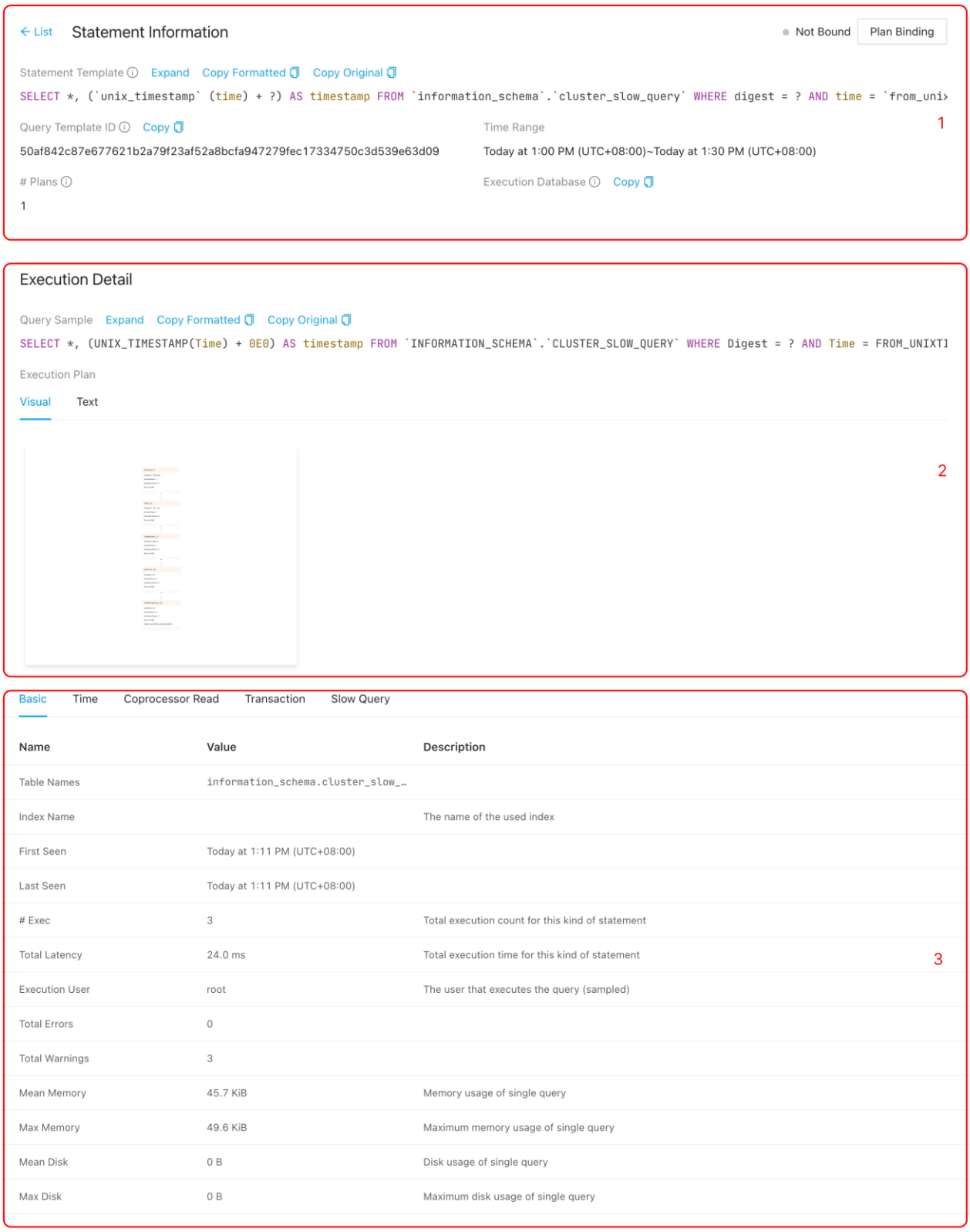

List
Statement Information
Not Bound
Plan Binding
Statement Template
Expand
Copy Formatted
Copy Original
SELECT *, (`unix_timestamp` (time) + ?) AS timestamp FROM `information_schema`.`cluster_slow_query` WHERE digest = ? AND time = `from_uni
Query Template ID
Copy
50af842c87e677621b2a79f23af52a8bcfa947279fec17334750c3d539e63d09
Time Range
Today at 1:00 PM (UTC+08:00)~Today at 1:30 PM (UTC+08:00)
Plans
1
Execution Database
Copy
1
Execution Detail
Query Sample
Expand
Copy Formatted
Copy Original
SELECT *, (UNIX_TIMESTAMP(Time) + 0E0) AS timestamp FROM `INFORMATION_SCHEMA`.`CLUSTER_SLOW_QUERY` WHERE Digest = ? AND Time = FROM_UNIXTI
Execution Plan
Visual
Text
2
Basic
Time
Coprocessor Read
Transaction
Slow Query
Name
Value
Description
Table Names
information_schema.cluster_slow_…
Index Name
The name of the used index
First Seen
Today at 1:11 PM (UTC+08:00)
Last Seen
Today at 1:11 PM (UTC+08:00)
Exec
3
Total execution count for this kind of statement
Total Latency
24.0 ms
Total execution time for this kind of statement
3
Execution User
root
The user that executes the query (sampled)
Total Errors
0
Total Warnings
3
Mean Memory
45.7 KiB
Memory usage of single query
Max Memory
49.6 KiB
Maximum memory usage of single query
Mean Disk
0 B
Disk usage of single query
Max Disk
0 B
Maximum disk usage of single query

图 292: 详情

11.9.2.1 快速绑定执行计划

自 v6.6.0 起，TiDB 引入了快速绑定执行计划的功能。你可以在TiDB Dashboard 中，快速完成 SQL 语句与特定计划的绑定。

11.9.2.1.1 使用方式

绑定一个执行计划

1. 点击执行计划绑定(Plan Binding)，弹出弹窗。

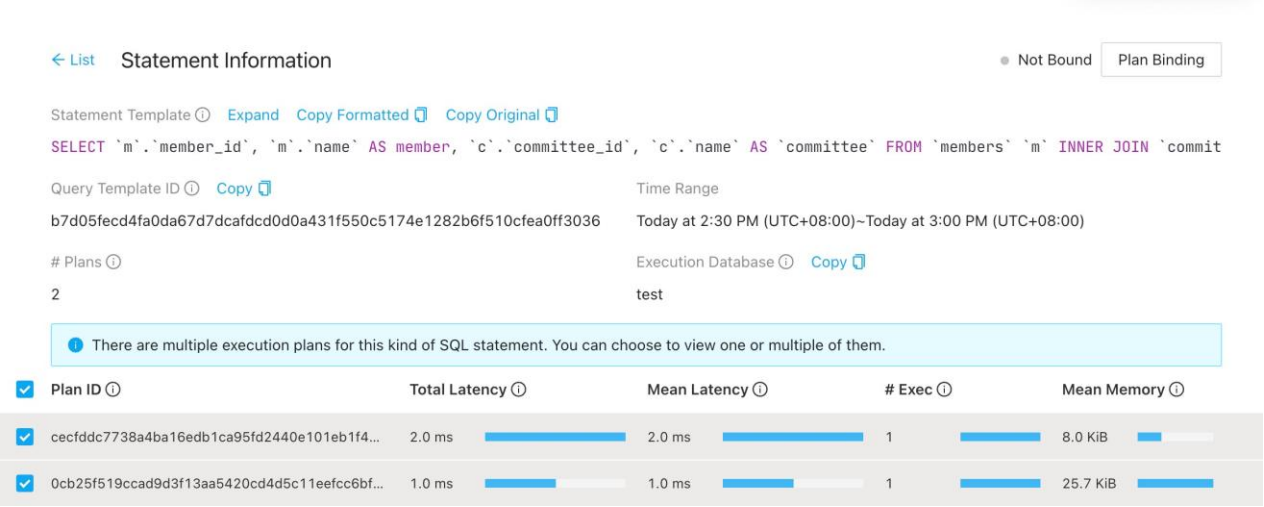


图 293: 快速绑定执行计划-未绑定-入口

2. 选择一个需要绑定的执行计划，点击绑定(Bind)。

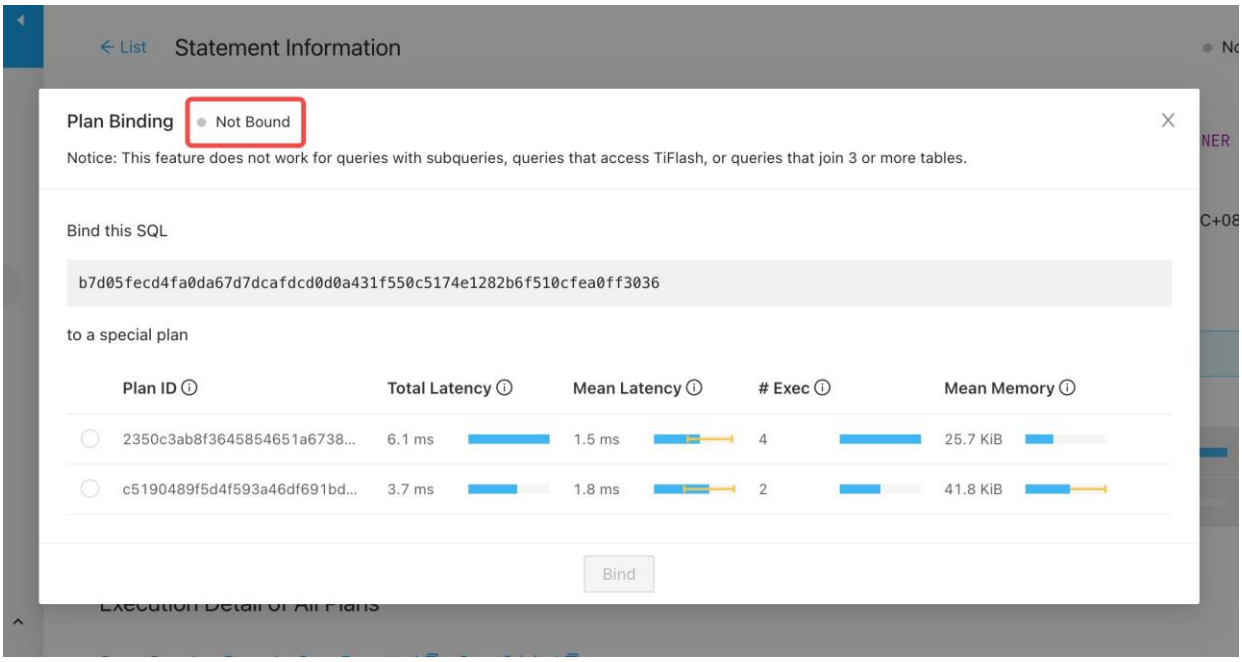


图 294: 快速绑定执行计划-弹窗

3. 绑定成功后，可以看到已绑定(Bound)提示。

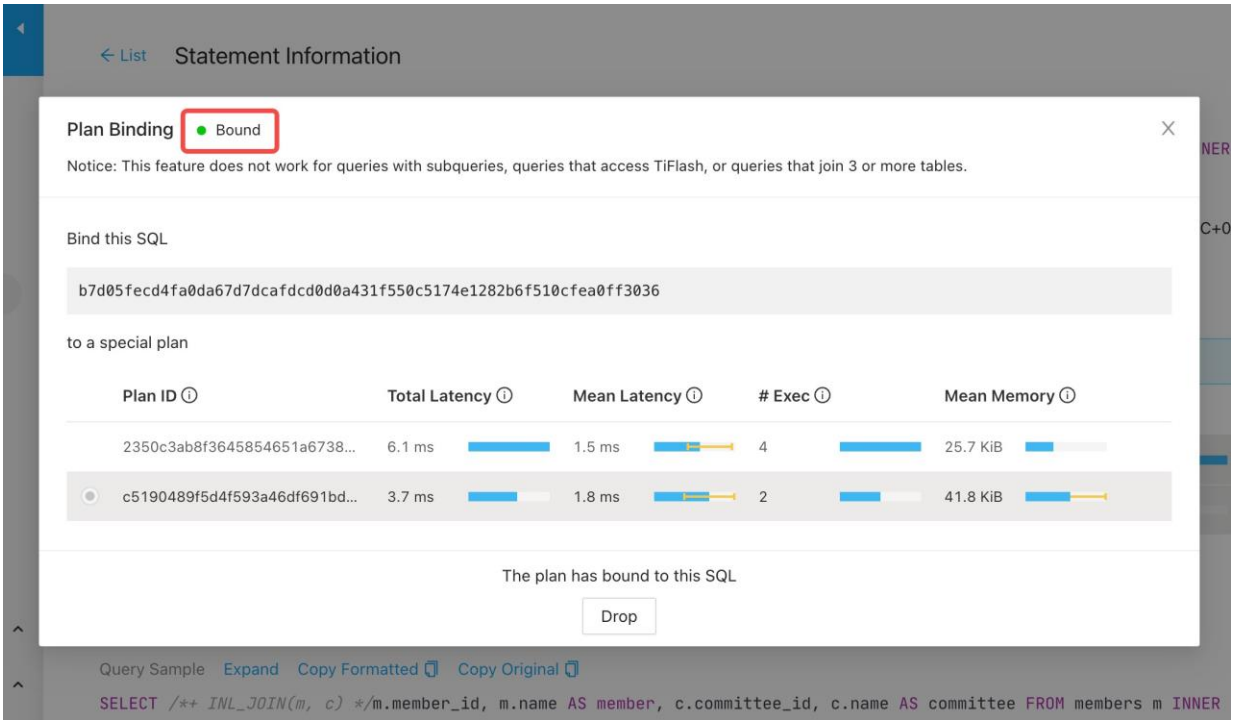


图 295: 快速绑定执行计划-弹窗-绑定成功

取消已有的执行计划绑定

1. 在已经绑定了执行计划的 SQL 语句页面上，点击执行计划绑定(Plan Binding)，弹出弹窗。

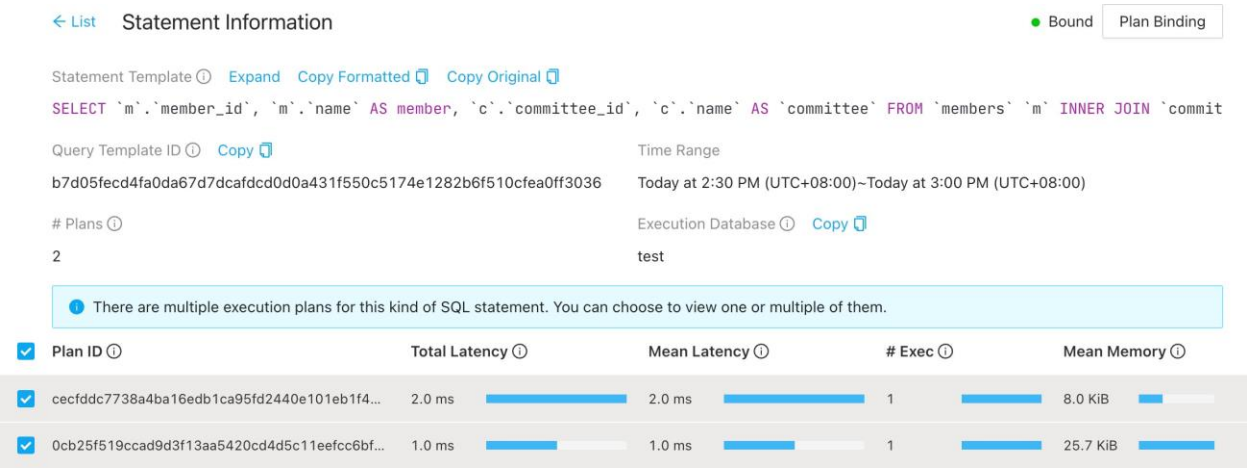


图 296: 快速绑定执行计划-已绑定-入口

2. 点击取消绑定(Drop)。

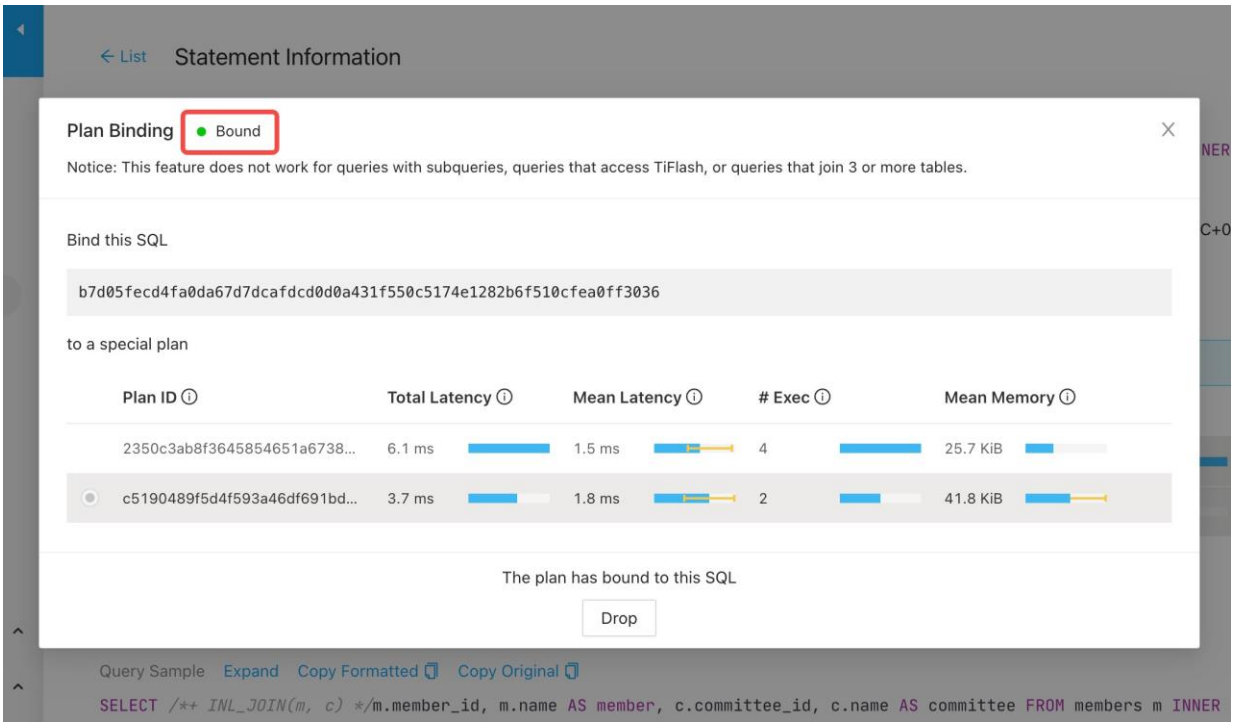


图 297: 快速绑定执行计划-弹窗-绑定成功

3. 取消成功后，可以看到未绑定(Not Bound) 提示。

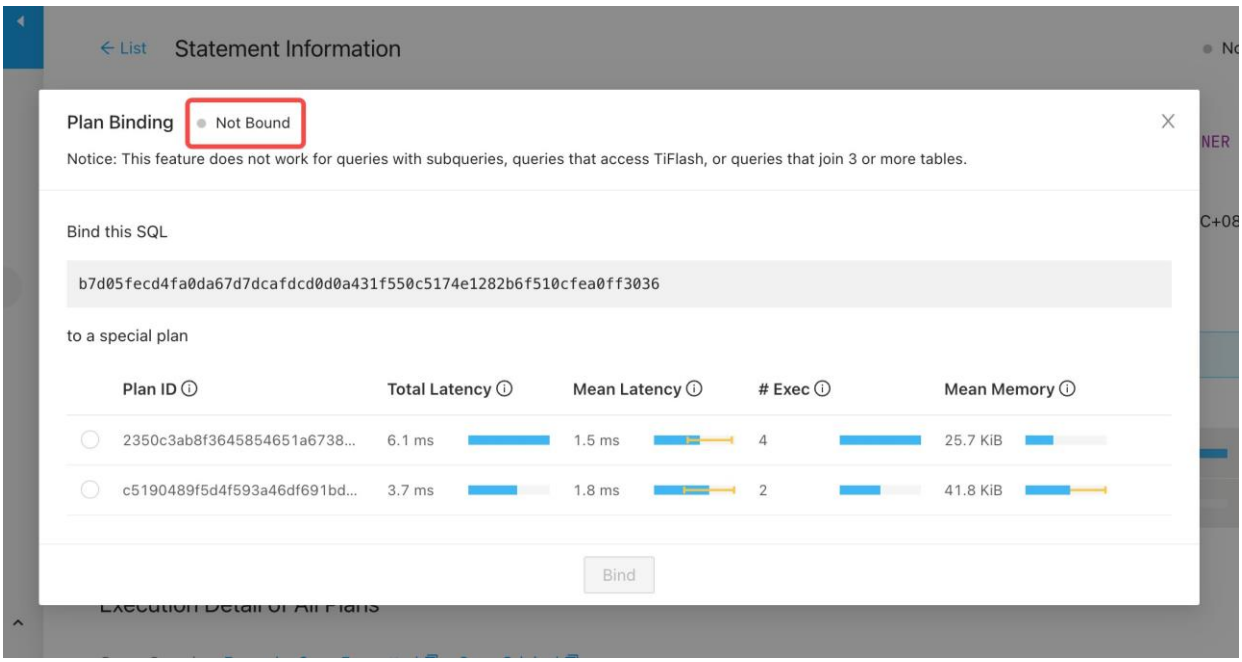


图 298: 快速绑定执行计划-弹窗

11.9.2.1.2 使用限制

目前还不支持通过TiDB Dashboard 绑定下列语句的执行计划：

- 非 SELECT、DELETE、UPDATE、INSERT、REPLACE 类型语句
- 带有子查询的查询
- 访问TiFlash 的查询
- 对三张表或更多表进行Join 的查询

目前该功能需用户拥有 SUPER 权限才可使用。如果在使用过程中提示权限不足，请参考TiDB Dashboard 用户管理补充所需权限。

11.9.2.2 执行计划详情

执行计划详情包括以下内容：

- SQL 样本：该计划对应的实际执行的某一条 SQL 语句文本。时间范围内任何出现过的 SQL 都可能作为 SQL 样本。
- 执行计划：执行计划的完整内容，有图形和文本两种展示形式。参阅理解TiDB 执行计划文档了解如何解读执行计划。如果选择了多个执行计划，则显示的是其中任意一个。
- 其他关于该 SQL 的基本信息、执行时间、Coprocessor 读取、事务、慢查询等信息，可点击相应标签页标题切换。

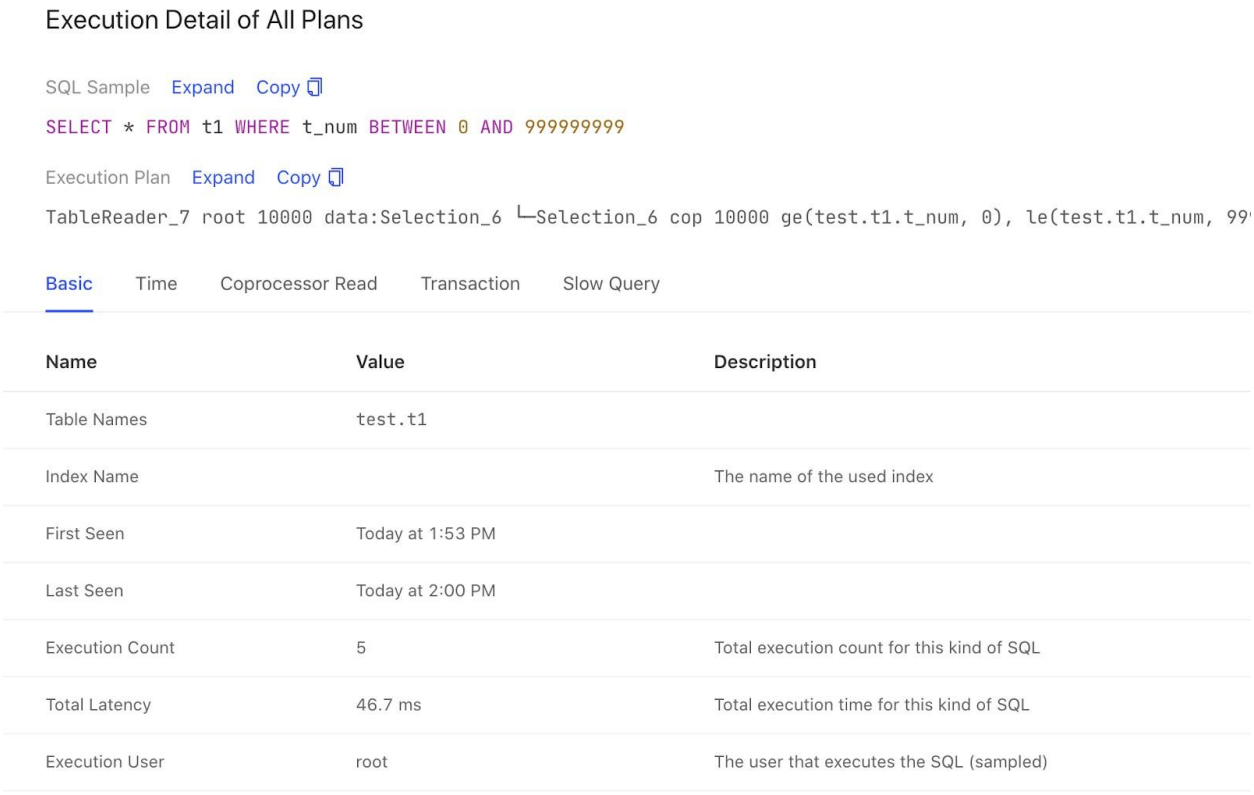


图 299: 执行计划详情

11.9.2.2.1 基本信息

包含关于表名、索引名、执行次数、累计耗时等信息。描述(Description) 列对各个字段进行了具体描述。

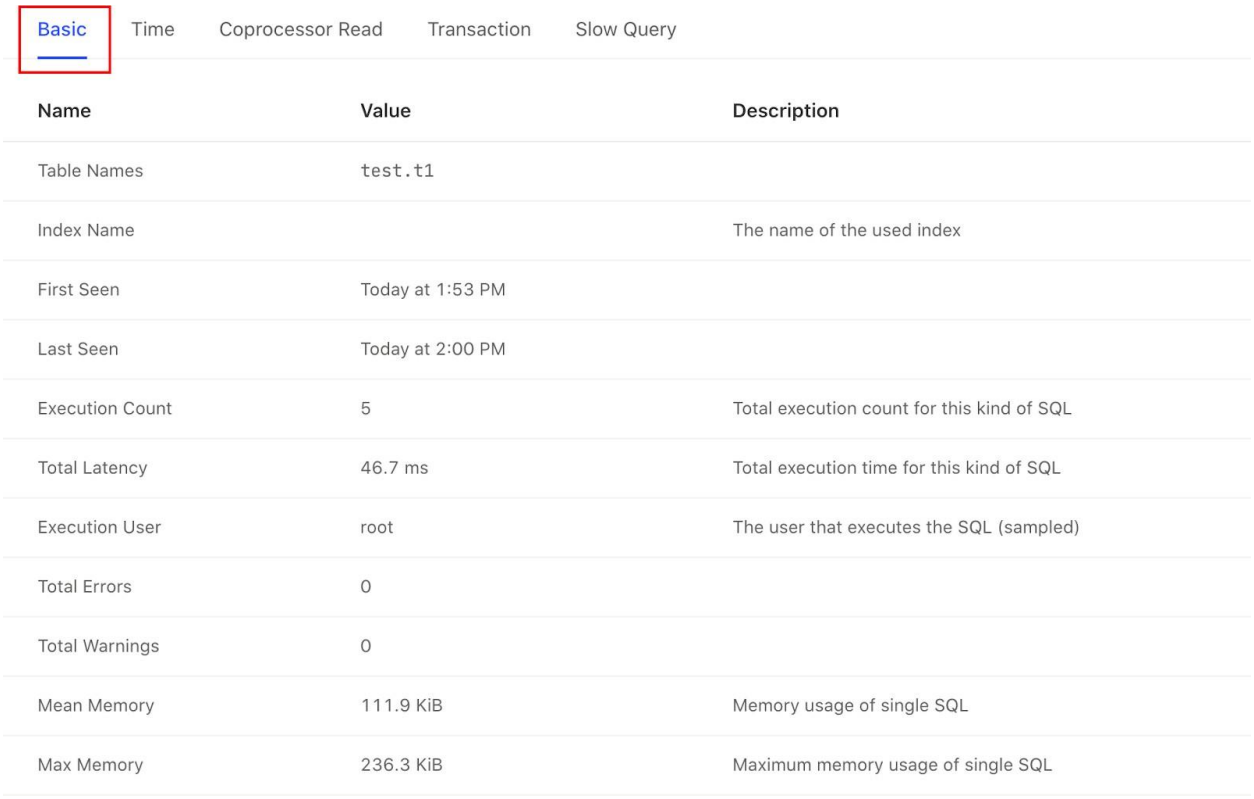


图 300: 基本信息

11.9.2.2.2 执行时间

显示执行计划执行的各阶段所耗费时间。

> **注意：**
>
> 由于单个 SQL 语句内部可能有并行执行的操作，因此各阶段累加时间可能超出该 SQL 语句的实际执行时间。

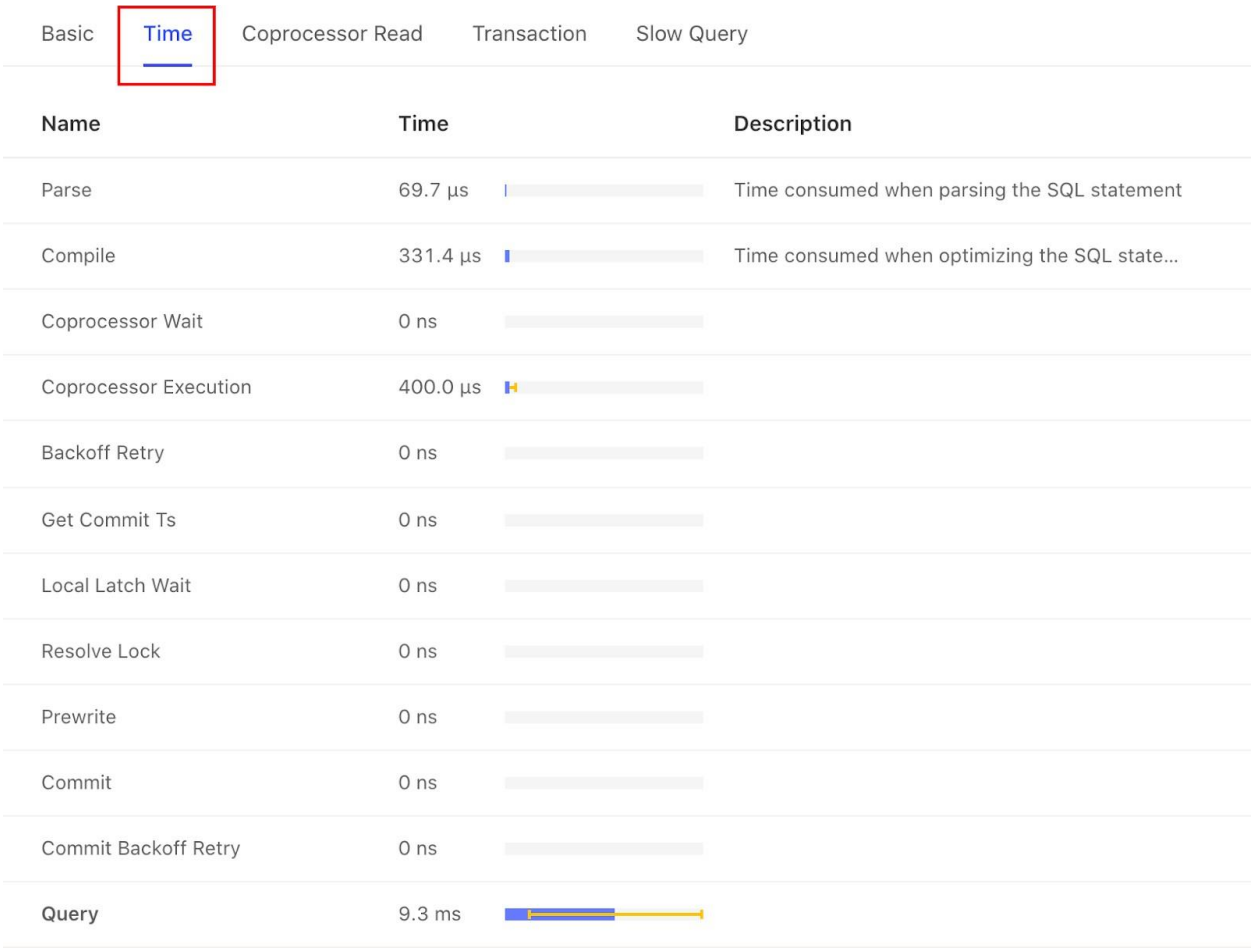


图 301: 执行时间

11.9.2.2.3 Coprocessor 读取

显示 Coprocessor 读取的相关信息。

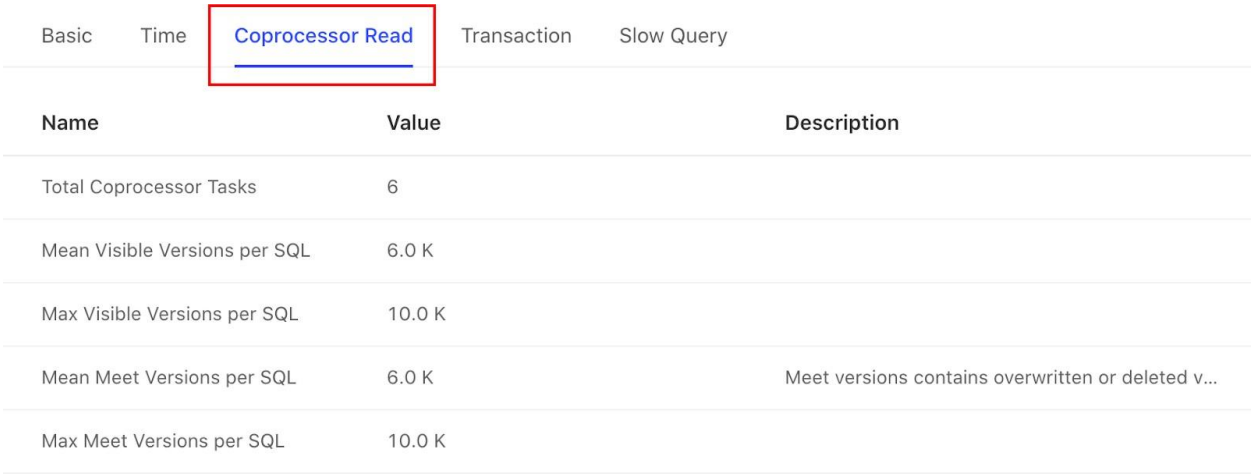


图 302: Coprocessor 读取

11.9.2.2.4 事务

显示执行计划与事务相关的信息，比如平均写入 key 个数，最大写入 key 个数等。

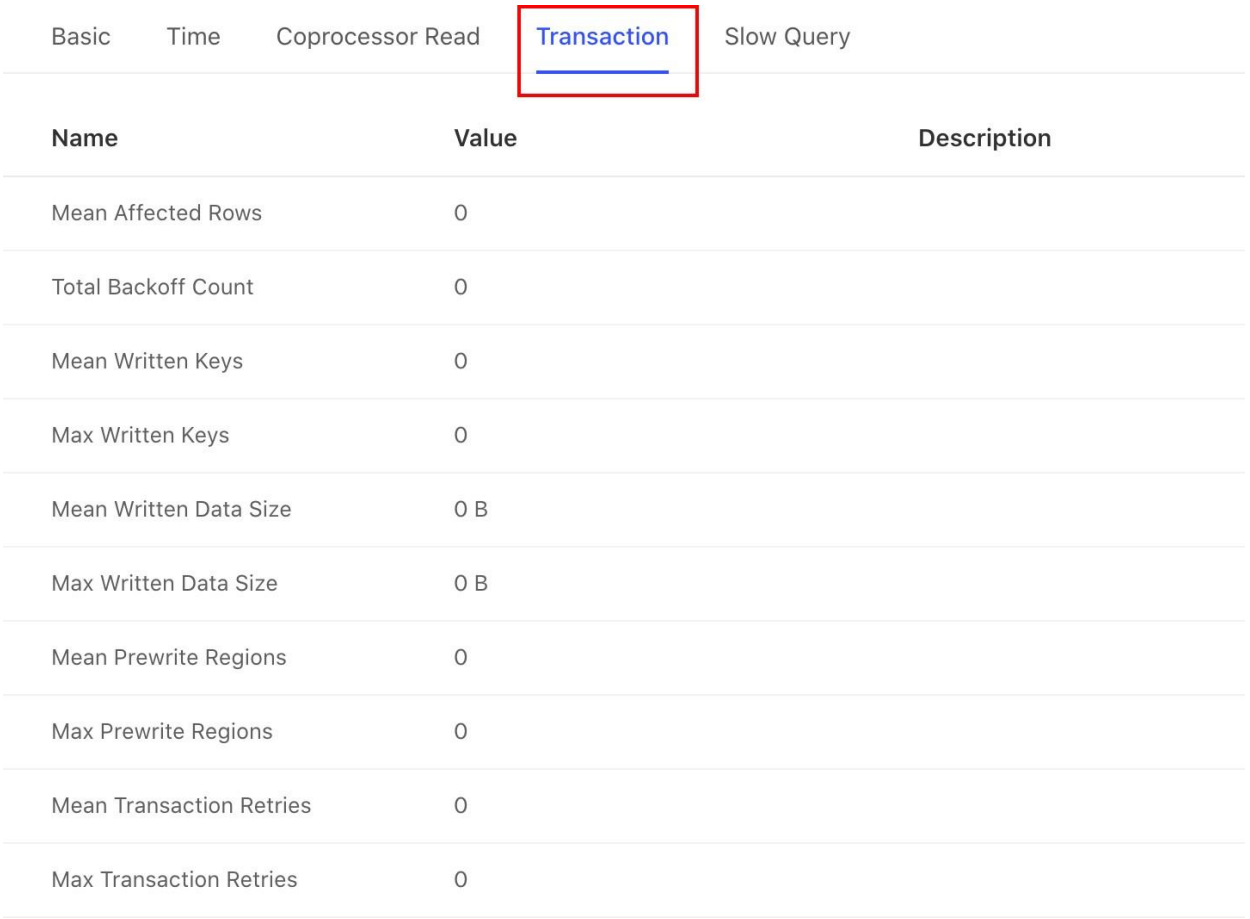


图 303: 事务

11.9.2.2.5 慢查询

如果该执行计划执行过慢，则在慢查询标签页下可以看到其关联的慢查询记录。

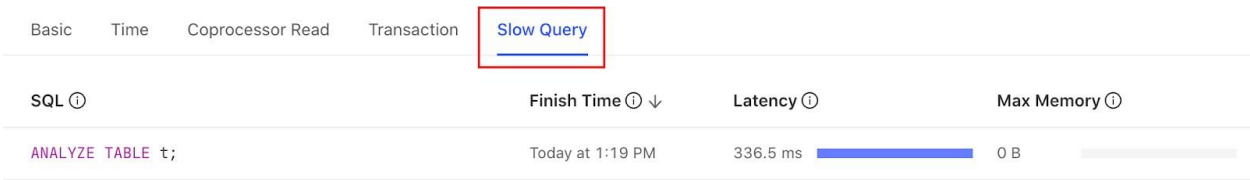


图 304: 慢查询

该区域显示的内容结构与慢查询页面一致，详见慢查询页面。

11.10 TiDB Dashboard 慢查询页面

该页面上能检索和查看集群中所有慢查询。

默认情况下，执行时间超过 300ms 的 SQL 查询就会被视为慢查询，被记录到慢查询日志中，并可通过本功能对记录到的慢查询进行查询。可调整`tidb_slow_log_threshold` SESSION 变量或TiDB `slow-threshold` 参数调整慢查询阈值。

> 注意：
>
> 若关闭了慢查询日志，则本功能不可用。慢查询日志默认开启，可通过修改 TiDB 系统变量`tidb_enable_slow_log`开启或禁用。

11.10.1 访问列表页面

可以通过以下两种方法访问慢查询页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击慢查询(Slow Queries)。
- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/slow_query（将 `127.0.0.1:2379` 替换为你的实际 PD 地址和端口）。

慢查询页面所展示的所有数据都来自于TiDB 慢查询系统表及慢查询日志，参见慢查询日志文档了解详细情况。

11.10.1.1 修改列表过滤条件

可按时间范围、慢查询语句关联的数据库、SQL 关键字、SQL 类型、显示的慢查询语句数量等条件过滤，筛选慢查询句。如下所示，默认显示 30 分钟内最近 100 条慢查询。

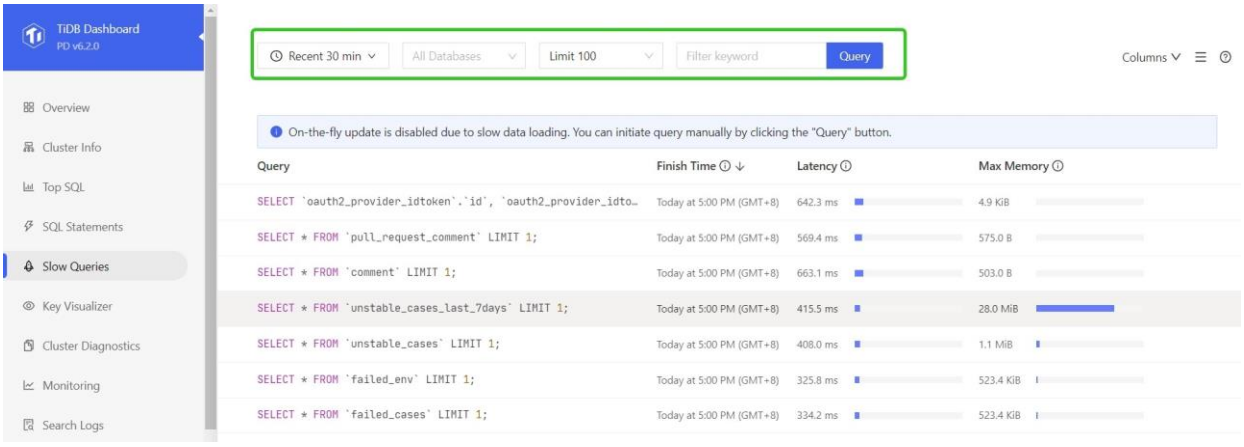


图 305: 修改列表过滤条件

11.10.1.2 显示更多列信息

页面顶部选择列(Columns) 选项中可选择显示更多列，可将鼠标移动到列名右侧的(i) 图标处查看列的说明：

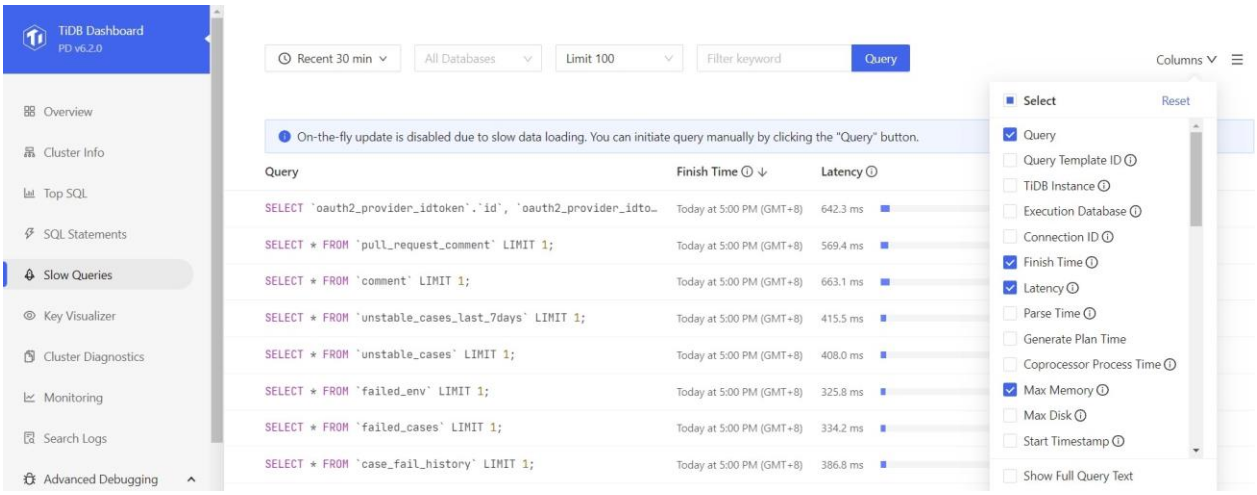


图 306: 显示更多列信息

11.10.1.3 修改列表排序依据

列表默认以结束运行时间(Finish Time) 逆序排序，点击不同的列标题可以修改排序依据或切换排序顺序：

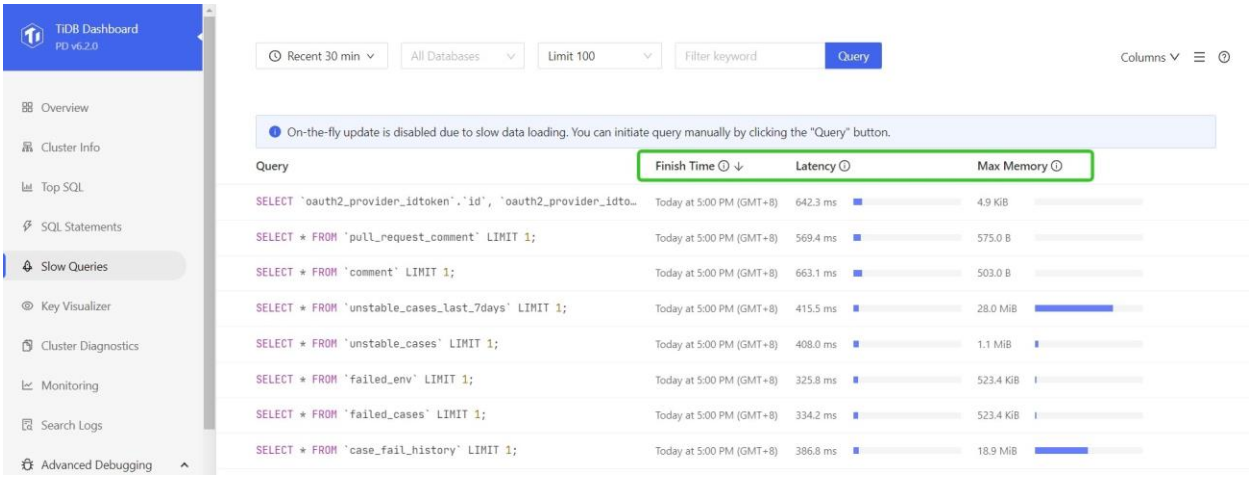


图 307: 修改列表排序依据

11.10.2 查看执行详情

在列表中点击任意一行可以显示该慢查询的详细执行信息，包含：

- SQL：慢查询 SQL 文本（下图中区域 1）
- 执行计划：慢查询的执行计划（下图中区域 2）

- 其他分类好的 SQL 执行信息（下图中区域 3）

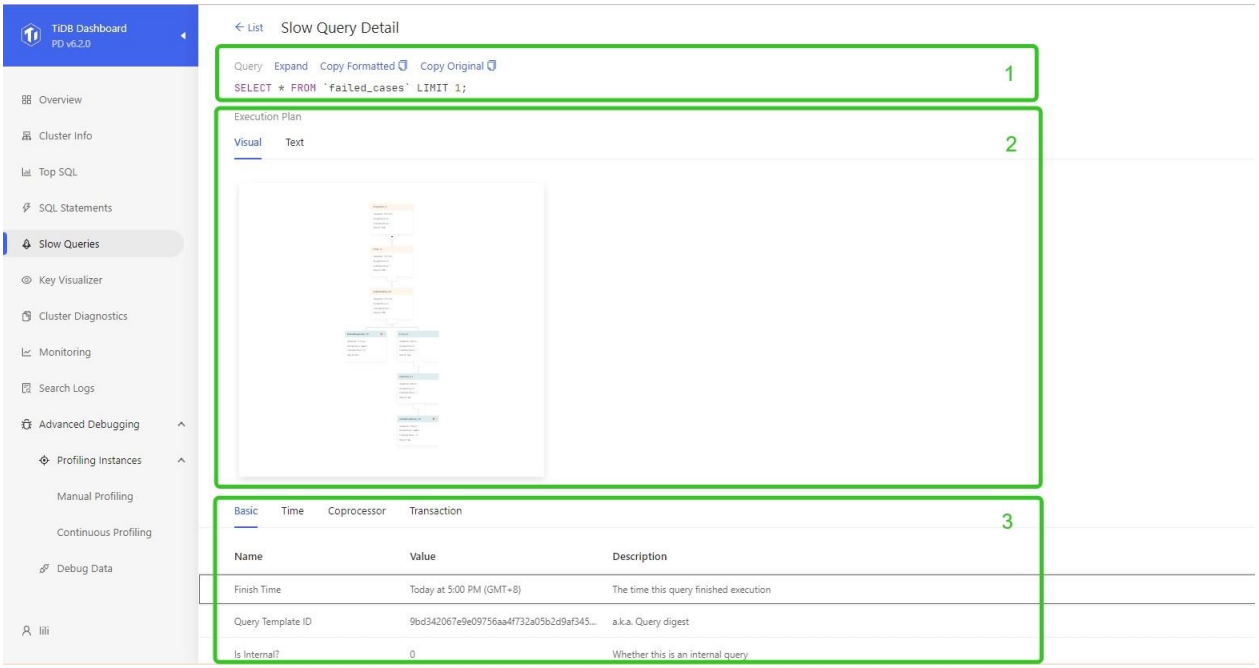


图 308: 查看执行详情

11.10.2.1 SQL

点击展开(Expand) 可以展开相应项的完整内容，点击复制(Copy) 可以复制内容到剪贴板。

11.10.2.2 执行计划

TiDB Dashboard 提供两种方式查看执行计划：图形和文本。通过图形化的执行计划，你可以更清晰地了解一个语句的具体算子和对应的内容。参阅理解 TiDB 执行计划。

11.10.2.2.1 图形化执行计划介绍

下图为一个执行计划的图形化展示。

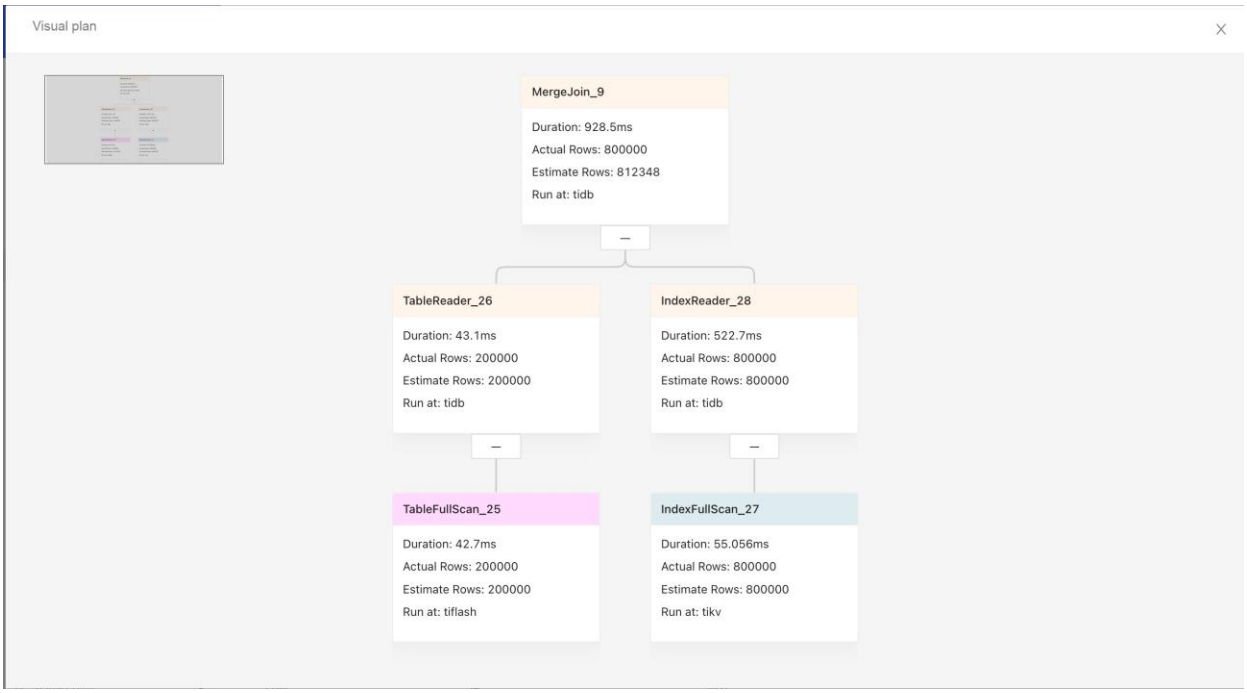


图 309: 可视化执行计划弹窗

- 执行计划的展示顺序是从左到右，从上到下。
- 上面的节点是父算子，下面的节点是子算子。
- 节点顶栏的颜色代表算子执行的组件：黄色代表TiDB，蓝色代表TiKV，粉色代表TiFlash。
- 节点的顶栏为算子的名称，正文为算子的基本信息。

点击节点区域，右侧将弹出算子的详细信息。

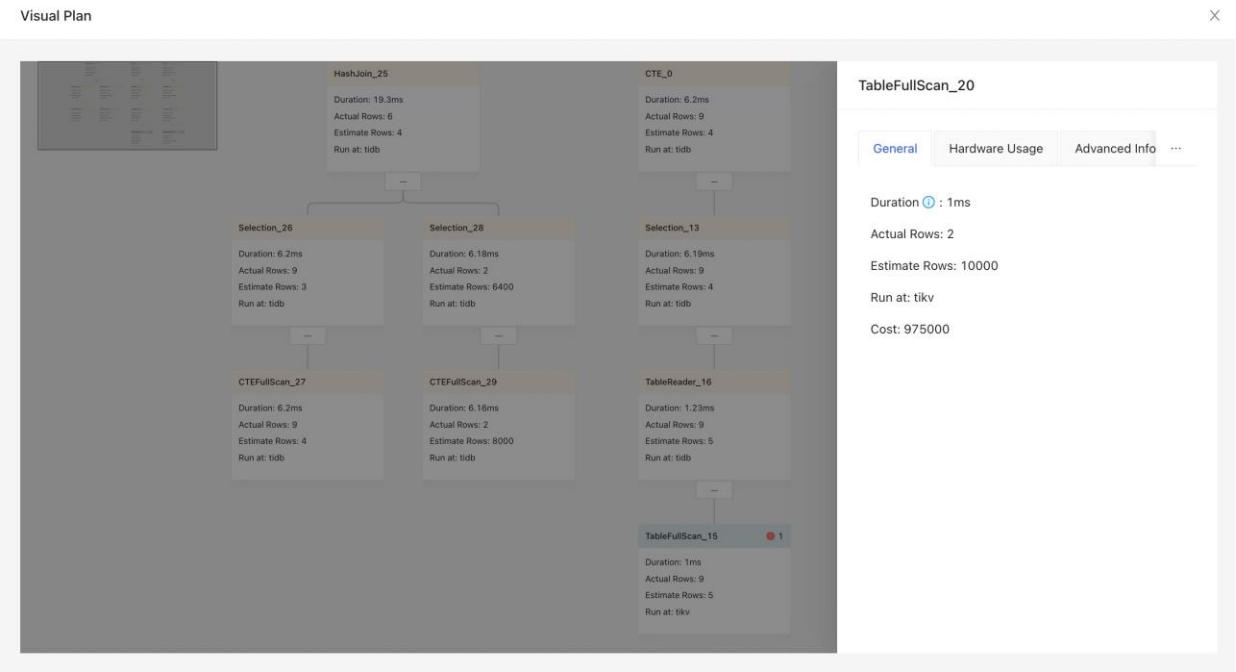


图 310: 可视化执行计划弹窗-侧栏

11.10.2.3 SQL 执行相关信息

点击标签页标题可切换显示不同分类的 SQL 执行信息：

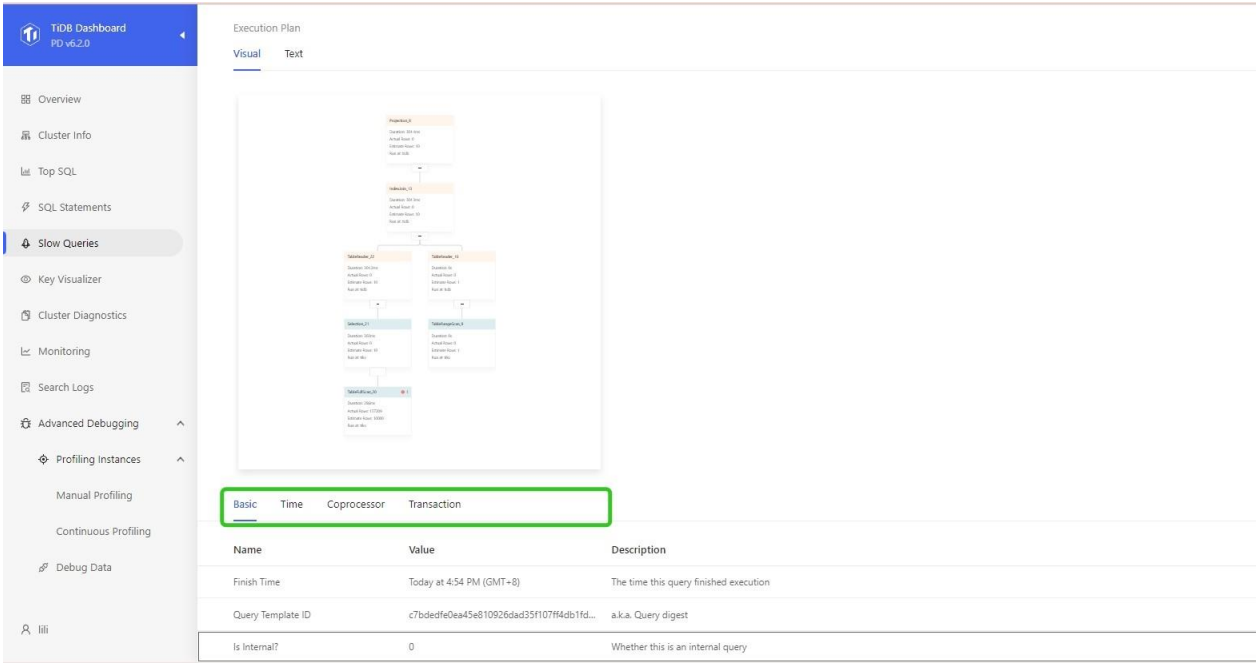


图 311: 显示不同分类执行信息

11.11 集群诊断页面

11.11.1 TiDB Dashboard 集群诊断页面

集群诊断是在指定的时间范围内，对集群可能存在的问题进行诊断，并将诊断结果和一些集群相关的负载监控信息汇总成一个诊断报告。诊断报告是网页形式，通过浏览器保存后可离线浏览和传阅。

> **注意：**
>
> 集群诊断功能依赖于集群中部署有 Prometheus 监控组件，参见TiUP 部署文档了解如何部署监控组件。若集群中没有部署监控组件，生成的诊断报告中将提示生成失败。

11.11.1.1 访问

可以通过以下两种方法访问集群诊断页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击集群诊断(Cluster Diagnostics)。

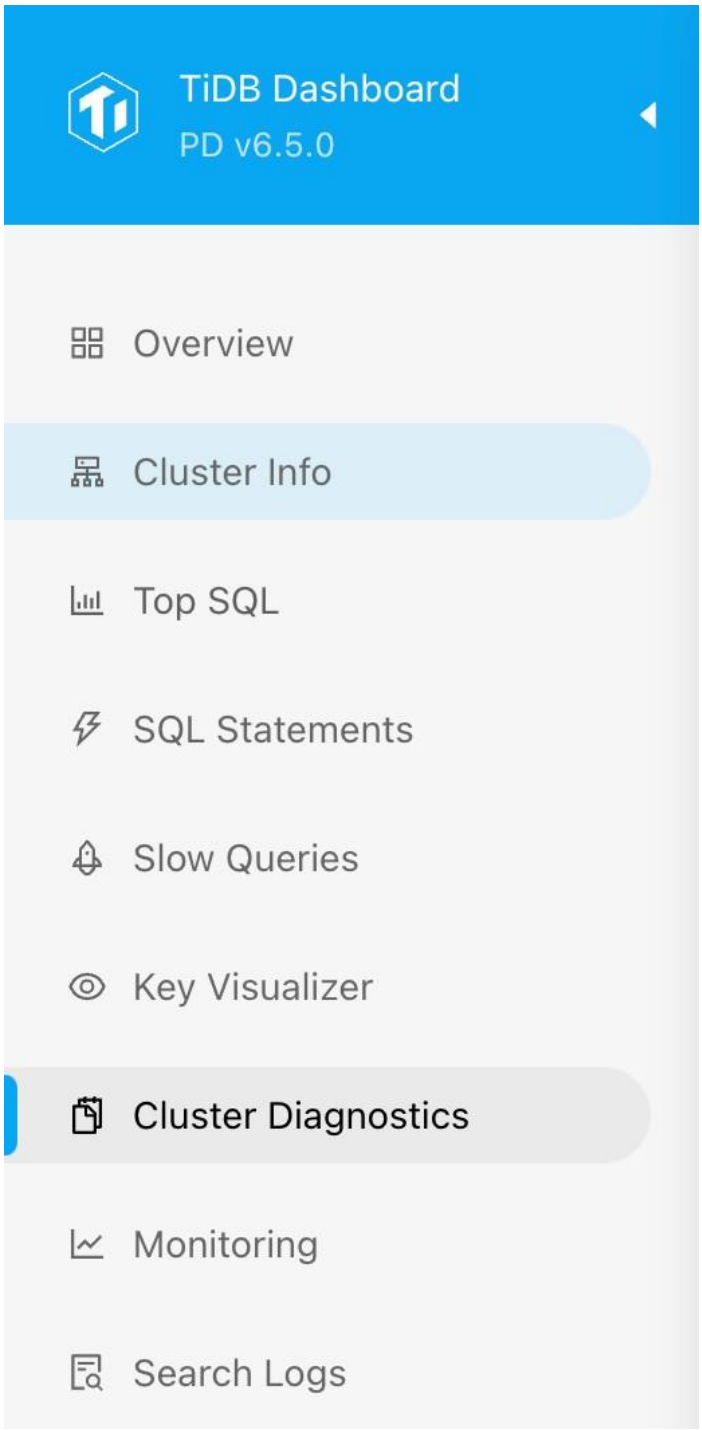


图 312: 访问

- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/diagnose（将 `127.0.0.1:2379` 替换为你的实际 PD 地址和端口）。

11.11.1.2 生成诊断报告

如果想对一个时间范围内的集群进行诊断，查看集群的负载等情况，可以使用以下步骤来生成一段时间范围

的诊断报告：

1. 设置区间的开始时间，例如 2022-05-21 14:40:00。
2. 设置区间长度。例如 10 min。
3. 点击开始(Start)。

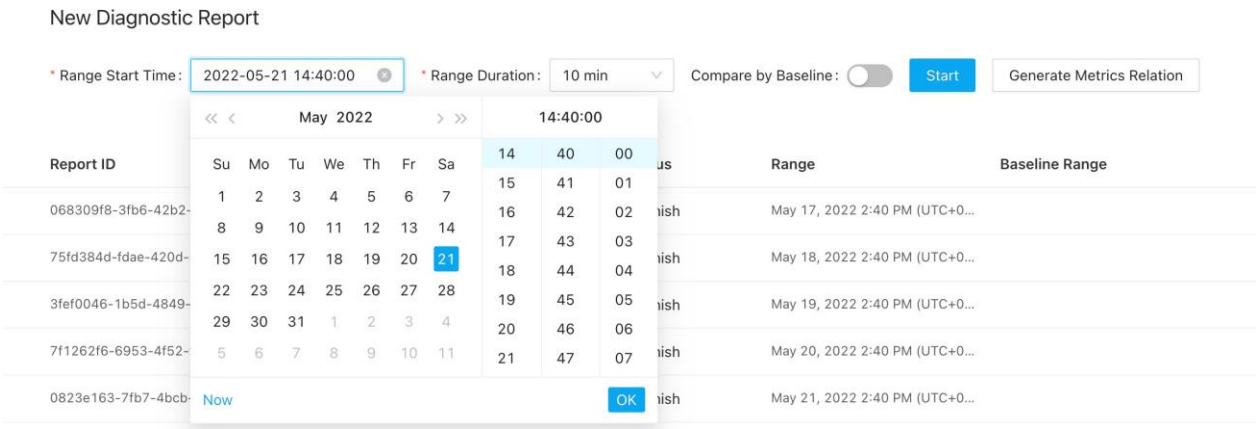


图 313: 生成单个时间段的诊断报告

> **注意：**
> 建议生成报告的时间范围在 1 min ~ 60 min 内，目前不建议生成超过 1 小时范围的报告。

以上操作会生成 2022-05-21 14:40:00 至 2022-05-21 14:50:00 时间范围的诊断报告。点击开始(start) 后，会看到以下界面，生成进度(Progress) 是生成报告的进度条，生成报告完成后，点击查看报告(View Full Report) 即可。

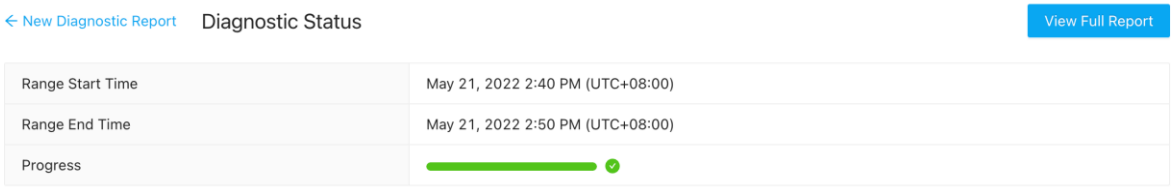


图 314: 生成报告的进度

11.11.1.3 生成对比诊断报告

如果系统在某个时间点发生异常，如 QPS 抖动或者延迟变高，可以生成一份异常时间范围和正常时间范围的对比报告，例如：

- 系统异常时间段：2022-05-21 14:40:00 ~ 2022-05-21 14:45:00，系统异常时间。
- 系统正常时间段：2022-05-21 14:30:00 ~ 2022-05-21 14:35:00，系统正常时间。

生成以上两个时间范围的对比报告的步骤如下：

1. 设置区间的开始时间，即异常时间段的开始时间，如 2022-05-21 14:40:00。
2. 设置区间长度(Range Duration)。一般指系统异常的持续时间，例如 5 min。
3. 开启与基线区间对比开关(Compare by Baseline)。
4. 设置基线开始时间 (Baseline Range Start Time)，即想要对比的系统正常时段的开始时间，如 2022-05-21 14:30:00。
5. 点击开始(Start)。

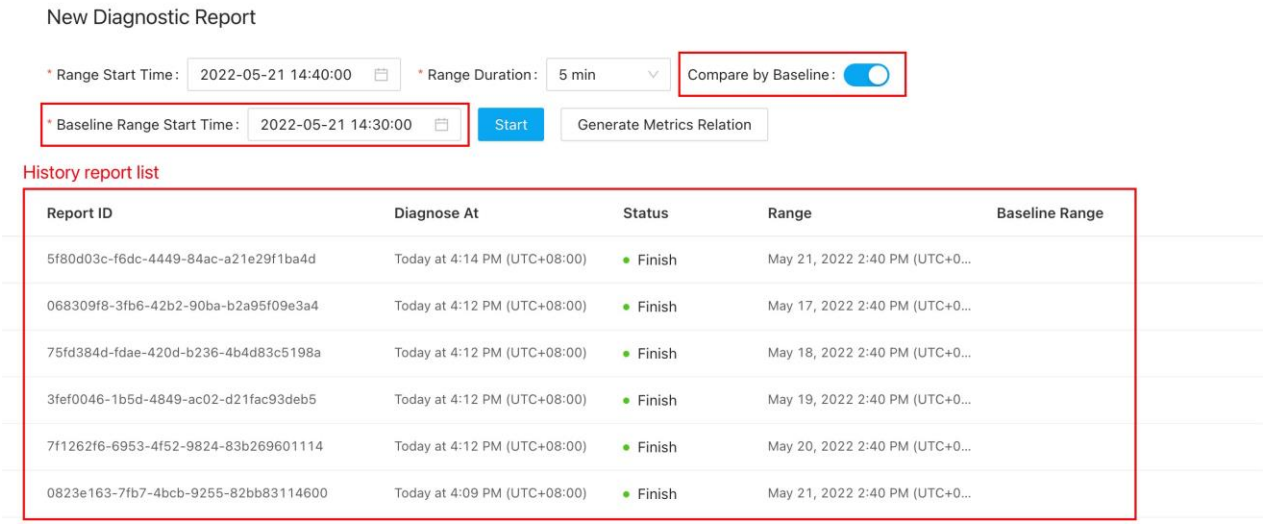


图 315: 生成对比报告

然后同样等报告生成完成后点击查看报告(View Full Report) 即可。

另外，已生成的诊断报告会显式在诊断报告主页的列表里面，可以点击查看之前生成的报告，不用重复生成。

11.11.2 TiDB Dashboard 诊断报告

本文档主要介绍诊断报告的内容以及查看技巧，访问集群诊断和生成报告请参考诊断报告访问文档。

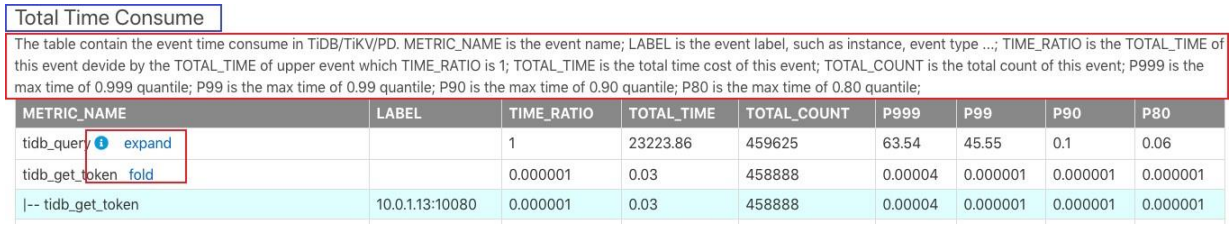
11.11.2.1 查看报告

诊断报告由以下几部分组成：

- 基本信息：包括生成报告的时间范围，集群的硬件信息，集群的拓扑版本信息。
- 诊断信息：显示自动诊断的结果。
- 负载信息：包括服务器，TIDB/PD/TiKV 相关的 CPU、内存等负载信息。

- 概览信息：包括TiDB/PD/TiKV 的各个模块的耗时信息和错误信息。
- TiDB/PD/TiKV 监控信息：包括各个组件的监控信息。
- 配置信息：包括各个组件的配置信息。

报告中报表示例如下：



Total Time Consume

The table contain the event time consume in TiDB/TiKV/PD. METRIC_NAME is the event name; LABEL is the event label, such as instance, event type ...; TIME_RATIO is the TOTAL_TIME of this event devide by the TOTAL_TIME of upper event which TIME_RATIO is 1; TOTAL_TIME is the total time cost of this event; TOTAL_COUNT is the total count of this event; P999 is the max time of 0.999 quantile; P99 is the max time of 0.99 quantile; P90 is the max time of 0.90 quantile; P80 is the max time of 0.80 quantile;

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token fold		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
|-- tidb_get_token	10.0.1.13:10080	0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001

图 316: 示例报表

上图中，最上面蓝框内的Total Time Consume 是报表名。 下方红框内的内容是对该报表意义的解释，以及报表中各个字段的含义。

报表中小按钮图标的解释如下：

- i 图标：鼠标移动到i 图标处会显示该行的说明注释。
- expand：点击 expand 会看到这项监控更加详细的信息。如是上图中 `tidb_get_token` 的详细信息包括各个 TiDB 实例的延迟监控信息。
- fold：和 expand 相反，用于把监控的详细信息折叠起来。

所有监控基本上和TiDB Grafna 监控面板上的监控内容相对应，发现某个模块异常后，可以在TiDB Grafna 监控面板上查看更多详细的监控信息。

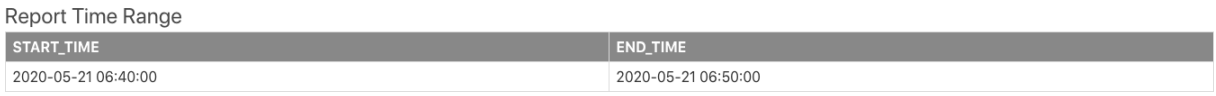
另外，报表中统计的 `TOTAL_TIME` 和 `TOTAL_COUNT` 由于是从 Prometheus 读取的监控数据，其统计会有一些计算上的精度误差。

以下介绍诊断报告的各部分内容。

11.11.2.1.1 基本信息

Report Time Range

Report Time Range 表显示生成报告的时间范围，包括开始时间和结束时间。



Report Time Range

START_TIME	END_TIME
2020-05-21 06:40:00	2020-05-21 06:50:00

图 317: report time range 报表

Cluster Hardware

Cluster Hardware 表显示集群中各服务器的硬件信息，包括 CPU、 Memory、 磁盘等信息。

Cluster Hardware

HOST	INSTANCE	CPU_CORES	MEMORY (GB)	DISK (GB)	UPTIME (DAY)
10.0.1.14	tiflash*1	20/40	125.64	nvme01: 498.638	2.9132028513567314
10.0.1.11	pd*1	20/40	125.64	sda1: 965.834	2.921851028772416
10.0.1.12	tikv*1	20/40	125.64	nvme0n1: 122.78 sda1: 965.834	2.9197623066052247
10.0.1.13	tidb*1	20/40	125.64	sda1: 498.638	2.917983893669314

图 318: Cluster Hardware 报表

上表中各个字段含义如下：

- HOST：服务器的 IP 地址。
- INSTANCE：该服务器部署的实例数量，如pd * 1 代表这台服务器部署了1 个 PD 实例。如tidb * 2 pd * 1 表示这台服务器部署了 2 个 TiDB 实例和 1 个 PD 实例。
- CPU_CORES：表示服务器 CPU 的核心数，物理核心/逻辑核心。
- MEMORY：表示服务器的内存大小，单位是 GB。
- DISK：表示服务器磁盘大小，单位是 GB。
- UPTIME：服务器的启动时间，单位是 DAY。

Cluster Info

Cluster Info 为集群拓扑信息。表中信息来自TiDB 的 information_schema.cluster_info 系统表。

Cluster Info

TYPE	INSTANCE	STATUS_ADDRESS	VERSION	GIT_HASH	START_TIME	UPTIME
pd	10.0.1.11:2379	10.0.1.11:2379	4.1.0-alpha	a80b99ef0d70807d106bdc1b7c6e97a118679c7e	2020-05-18T13:47:01Z	65h40m54.520385474s
tidb	10.0.1.13:4000	10.0.1.13:10080	4.0.0-beta.2	ac30f5322e253125002663ad7c789807acfc9305	2020-05-18T13:47:10Z	65h40m45.520383242s
tiflash	10.0.1.14:3930	10.0.1.14:20292	v4.1.0-alpha-37-gacf3f673d	acf3f673dfbc3e6ffff0dda575760670993fefa6	2020-05-18T13:47:18Z	65h40m37.520390633s
tikv	10.0.1.12:20160	10.0.1.12:20180	4.1.0-alpha	6b09cadbf55311ac07fc51e1b43e3b57b54e71f2	2020-05-18T13:47:04Z	65h40m51.520389948s

图 319: Cluster Info 报表

上表中各个字段含义如下：

- TYPE：节点类型。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。
- VERSION：对应节点的语义版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。

11.11.2.1.2 诊断信息

TiDB 内置自动诊断的结果，具体各字段含义以及介绍可以参考information_schema.inspection_result 系统表的内容。

11.11.2.1.3 负载信息

Node Load Info

Node Load Info 表显示服务器节点的负载信息，包括时间范围内，服务器以下指标的平均值 (AVG)、最大值 (MAX)、最小值(MIN)：

- CPU 使用率，最大值是 100%
- 内存使用率
- 磁盘 I/O 使用率
- 磁盘写延迟
- 磁盘读延迟
- 磁盘每秒的读取字节数
- 磁盘每秒的写入字节数
- 节点网络每分钟收到的字节数
- 节点网络每分钟发送的字节数
- 节点正在使用的TCP 连接数
- 节点所有的TCP 连接数

Node Load Info

METRIC_NAME	instance	AVG	MAX	MIN
node_cpu_usage expand		40.42%	99.6%	6.98%
node_mem_usage expand		66.9%	96.28%	33.65%
node_disk_io_utilization expand		16%	97%	0%
Disk write latency expand		4000 us	50 ms	0 us
Disk read latency expand		2000 us	6000 us	0 us
tikv_disk_read_bytes expand		605.980 KB	15.842 MB	0.000 KB
tikv_disk_write_bytes expand		1.129 MB	12.235 MB	0.000 KB
node_network_in_traffic expand		225.706 KB	1.427 MB	0.000 KB
node_network_out_traffic expand		165.807 KB	1.659 MB	0.000 KB
node_tcp_in_use expand		19	47	5
node_tcp_connections expand		30	61	6

图 320: Node Load Info 报表

Instance CPU Usage

Instance CPU Usage 表显示各个TiDB/PD/TiKV 进程的 CPU 使用率的平均值(AVG)，最大值(MAX)，最小值(MIN)，这里进程 CPU 使用率最大值是 100% * CPU 逻辑核心数。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:10089	tidb	1891%	1906%	1878%
172.16.5.40:22151	tikv	517%	571%	459%
172.16.5.40:20151	tikv	498%	562%	444%
172.16.5.40:23151	tikv	352%	399%	311%
172.16.5.40:24799	pd	39%	45%	36%

图 321: Instance CPU Usage 报表

Instance Memory Usage

Instance Memory Usage 表显示各个TiDB/PD/TiKV 进程占用内存字节数的平均值(AVG)，最大值(MAX)，最小值(MIN)。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:20151	tikv	9.562 GB	9.605 GB	9.523 GB
172.16.5.40:23151	tikv	9.528 GB	9.574 GB	9.496 GB
172.16.5.40:22151	tikv	9.433 GB	9.498 GB	9.345 GB
172.16.5.40:10089	tidb	904.500 MB	911.660 MB	894.336 MB
172.16.5.40:24799	pd	61.089 MB	61.270 MB	60.980 MB

图 322: Instance Memory Usage 报表

TiKV Thread CPU Usage

TiKV Thread CPU Usage 表显示TiKV 内部各个模块线程的 CPU 使用率的平均值(AVG)、最大值 (MAX)、和最小值 (MIN)。这里进程 CPU 使用率最大值为 100% * 对应配置的线程数量。

TiKV Thread CPU Usage

METRIC_NAME	INSTANCE	AVG	MAX	MIN	CONFIG_KEY	CURRENT_CONFIG_VALUE
grpc expand		14.83%	16%	12.96%		
schedule worker fold		8.76%	9.47%	7.56%		
|-- sched_worker	10.0.1.12:20180	8.76%	9.47%	7.56%	storage.scheduler-worker-pool-size	4
storage read pool expand		3.24%	3.64%	2.67%		
storage read pool normal expand		3.22%	3.64%	2.67%		
Async apply fold		3.14%	3.4%	2.71%		
|-- Async apply	10.0.1.12:20180	3.14%	3.4%	2.71%	raftstore.apply-pool-size	2
raftstore expand		2.7%	2.93%	2.38%		
rocksdb expand		0.54%	2.13%	0%		
unified read pool expand		0.45%	4.24%	0%		
split_check expand		0.26%	0.84%	0%		
gc expand		0.04%	0.4%	0%		
storage read pool high expand		0.02%	0.04%	0%		
snapshot expand		0.004%	0.02%	0%		
cop normal						
cop high						
cop low						
storage read pool low expand		0%	0%	0%		
cop						

图 323: TiKV Thread CPU Usage 报表

上表中的字段解释如下：

- CONFIG_KEY：表示对应模块的相关线程数配置。
- CURRENT_CONFIG_VALUE：表示配置在生成报表时刻的当前值。

> **注意：**
>
> CURRENT_CONFIG_VALUE 是生成报告时的值，并不是报告时间范围内的值。目前不能获取历史时间某些配置的值。

TiDB/PD Goroutines Count

TiDB/PD Goroutines Count 表显示 TiDB/PD goroutines 数量的平均值 (AVG)，最大值 (MAX)，和最小值 (MIN)。如果 goroutines 数量超过 2000，说明该进程并发太高，会对整体请求的延迟有影响。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:10089	tidb	1434.33	1473	1394
172.16.5.40:24799	pd	157	157	157

图 324: TiDB/PD goroutines count 报表

11.11.2.1.4 概览信息

Time Consumed by Each Component

Time Consumed by Each Component 显示包括集群中TiDB、 PD、 TiKV 各个模块的监控耗时以及各项耗时的占比。默认时间单位是秒。用户可以用该表快速定位哪些模块的耗时较多。

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token expand		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
tidb_parse expand		0.00001	0.29	4603	0.0006	0.0004	0.0002	0.0001
tidb_compile expand		0.0008	17.82	463488	0.001	0.0009	0.0003	0.0002
tidb_execute expand		1	23237.39	463491	90.81	0.35	0.1	0.08
tidb_distsql_execution expand		0.0002	4.53	1064	0.13	0.1	0.02	0.01
tidb_cop expand		0.0005	12.39	1075	2.03	1.84	0.02	0.01
Transaction expand		1	23158.73	460023	8.16	7.91	5.38	0.06
tidb_transaction_local_latch_wait		0	0	0				
tidb_kv_backoff expand		0.00003	0.58	291	0.002	0.002	0.002	0.002
KV request expand		1.65	38325.58	2300949	2.03	1.84	0.04	0.03
Slow query expand		0.01	323.65	280	65.5	65.21	62.26	58.98
tidb_slow_query_cop_process		0	0	280	0.001	0.001	0.0009	0.0008
tidb_slow_query_cop_wait		0	0	280	0.001	0.001	0.0009	0.0008
DDL job		0	0	0				
tidb_ddl_worker		0	0	0				
tidb_ddl_update_self_version		0	0	0				
tidb_owner_handle_syncer		0	0	0				
Batch add index								
tidb_ddl_deploy_syncer		0	0	0				
Schema load expand		0.000008	0.19	27	0.06	0.06	0.06	0.05
TiDB meta operation expand		0.0001	3.21	1200	0.13	0.1	0.01	0.005
TiDB auto id request		0	0	0				
TiDB auto analyze expand		0.003	78.35	1	81.88	81.51	77.82	73.73
TiDB GC expand		0.000000001	0.00002	3	1	0.99	0.9	0.8
tidb_gc_push_task		0	0	0				
tidb_batch_client_unavailable		0	0	0				
tidb_batch_client_wait		0	0	0				
pd_tso_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_tso_wait expand		0.07	1662.29	920545	0.03	0.01	0.004	0.003
PD client cmd expand		0.15	3370.87	2761712	0.03	0.01	0.004	0.003
pd_client_request_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_grpc_completed_commands expand		0.00005	1.24	5255	0.01	0.01	0.006	0.005
pd_operator_finish								
pd_operator_step_finish								
Etcd transactions expand		0.000008	0.17	207	0.008	0.008	0.006	0.002
pd_region_heartbeat expand		0.00006	1.33	195	0	0	0	0
etcd_wal_fsync expand		0.00001	0.33	288	0.02	0.01	0.006	0.004
pd_peer_round_trip								

图 325: Total Time Consume 报表

上表各列的字段含义如下：

- METRIC_NAME：监控项的名称。
- Label：监控的 label 信息，点击 expand 后可以查看该项监控更加详细的各项label 的监控信息。
- TIME_RATIO：该项为 TIME_RATIO 为 1 的监控行总时间与监控消耗的总时间的比例。如 kv_request 的总耗时占 tidb_query 总耗时的 1.65 = 38325.58/23223.86。因为 KV 请求会并行执行，所以所有 KV 请求的总时间有可能超过总查询(tidb_query) 的执行时间。
- TOTAL_TIME：该项监控的总耗时。

- TOTAL_COUNT：该项监控执行的总次数。
- P999：该项监控的 P999 最大时间。
- P99：该项监控的 P99 最大时间。
- P90：该项监控的 P90 最大时间。
- P80：该项监控的 P80 最大时间。

以上监控中相关模块的耗时关系如下所示：

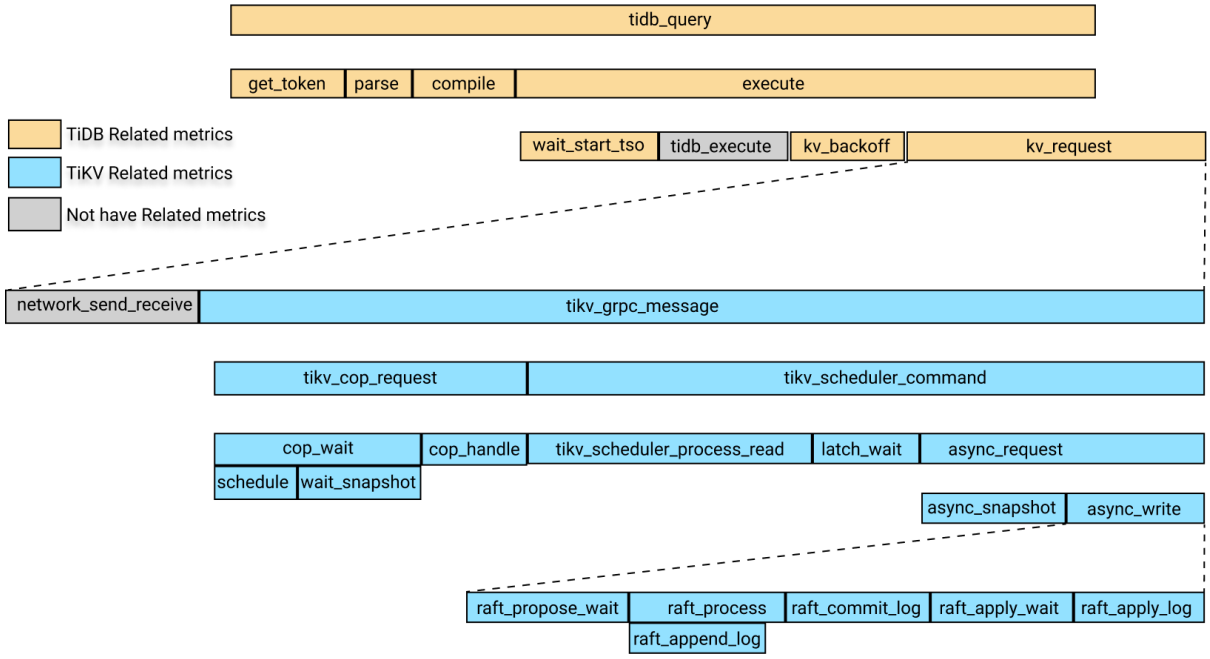


图 326: 各个模块耗时关系图

上图中，黄色部分是TiDB 相关的监控，蓝色部分是TiKV 相关的监控，灰色部分暂时没有具体对应的监控项。

上图中，tidb_query 的耗时包括以下部分的耗时：

- get_token
- parse
- compile
- execute

其中 execute 的耗时包括以下部分：

- wait_start_tso
- TiDB 层的执行时间，目前暂无监控
- KV 请求的时间
- tidb_kv_backoff 的时间，这是 KV 请求失败后进行backoff 的时间

其中，KV 请求时间包含以下部分：

- 请求的网络发送以及接收耗时，目前该项暂无监控，可以大致用 KV 请求时间减去 tikv_grpc_message 的时间
- tikv_grpc_message 的耗时

其中，tikv_grpc_message 耗时包含以下部分：

- Coprocessor request 耗时，指用于处理 COP 类型的请求，该耗时包括以下部分：
  - tikv_cop_wait，请求排队等待的耗时
  - Coprocessor handling request，处理 COP 请求的耗时
- tikv_scheduler_command 耗时，该耗时包含以下部分：
  - tikv_scheduler_processing_read，处理读请求的耗时
  - tikv_storage_async_request 中获取 snapshot 的耗时（snapshot 是该项监控的label）
  - 处理写请求的耗时，该耗时包括以下部分：
    * tikv_scheduler_latch_wait，等待 latch 的耗时
    * tikv_storage_async_request 中 write 的耗时（write 是该监控的label）

其中，tikv_storage_async_request 中的 write 耗时是指raft kv 写入的耗时，包括以下部分：

- tikv_raft_propose_wait
- tikv_raft_process，该耗时主要时间包括：
  - tikv_raft_append_log
  - tikv_raft_commit_log
  - tikv_raft_apply_wait
  - tikv_raft_apply_log

用户可以根据上述耗时之间的关系，利用 TOTAL_TIME 以及 P999，P99 的时间大致定位哪些模块耗时比较长，然后再看相关的监控。

> 注意：
>
> 由于Raft KV 可能会将多个请求作为一个batch 来写入，所以 TOTAL_TIME 不适用于来衡量Raft KV 的写入相关监控项的耗时，这些监控项具体是 tikv_raft_process、tikv_raft_append_log、tikv_raft_commit_log、tikv_raft_apply_wait、tikv_raft_apply_log。此时用 P999 和 P99 的时间来对比各个模块的耗时更加合理。
>
> 原因是，假如有 10 个 async write 请求，Raft KV 内部将 10 个请求打包成一个batch 执行，执行时间为 1 秒，所以每个请求的执行时间为 1 秒，10 个请求的总时间是 10 秒。但是Raft KV 处理的总时间是 1 秒。如果用 TOTAL_TIME 来衡量，用户可能不明白剩余的 9 秒耗时在哪些模块下。这里从总请求数(TOTAL_COUNT) 也能看出Raft KV 的监控和其他监控的差异。

Errors Occurred in Each Component

Errors Occurred in Each Component 表显示包括TiDB 和 TiKV 出现错误的总数。例如写binlog 失败、`tikv server is busy`、`TiKV channel full`、`tikv write stall` 等错误，具体各项错误含义可以看行注释。

METRIC_NAME	LABEL	TOTAL_COUNT
tidb_binlog_error_total_count		0
tidb_handshake_error_total_count		0
tidb_transaction_retry_error_total_count		
tidb_kv_region_error_total_count expand		903
tidb_schema_lease_error_total_count		0
tikv_grpc_error_total_count		0
tikv_critical_error_total_count		
tikv_scheduler_is_busy_total_count		
tikv_channel_full_total_count		
tikv_coprocessor_request_error_total_count expand		1
tikv_engine_write_stall		0
tikv_server_report_failures_total_count expand		1396
tikv_storage_async_request_error expand		1176
tikv_lock_manager_detect_error_total_count		
tikv_backup_errors_total_count		
node_network_in_errors_total_count		0
node_network_out_errors_total_count		0

图 327: Errors Occurred in Each Component 报表

11.11.2.1.5 TiDB/PD/TiKV 的具体监控信息

这部分包括了TiDB/PD/TiKV 更多的具体的监控信息。

TiDB 相关监控信息

Time Consumed by TiDB Component

Time Consumed by TiDB Component 表显示 TiDB 的各项监控耗时以及各项耗时的占比。和Time Consumed by Each Component 表类似，但是这个表的label 信息会更丰富，细节更多。

TiDB Server Connections

TiDB Server Connections 表显示TiDB 各个实例的客户端连接数。

TiDB Transaction

TiDB Transaction 表显示TiDB 事务相关的监控。

METRIC_NAME	LABEL	TOTAL_VALUE	TOTAL_COUNT	P999	P99	P90	P80
tidb_transaction_retry_num		0	0	0	0	0	0
tidb_transaction_statement_num expand		6854588	6852371	4	4	3	2
tidb_txn_region_num expand		1070770	706887	3	2	2	2
tidb_txn_kv_write_num expand		513386	181312	234	2	2	2
tidb_txn_kv_write_size expand		266.772 MB	181296	116.913 KB	1.996 KB	1.905 KB	1.805 KB
tidb_load_safepoint_total_num expand		16					
tidb_lock_resolver_total_num expand		420514					

图 328: Transaction 报表

- TOTAL_VALUE：该项监控在报告时间段内所有值的和 (SUM)。
- TOTAL_COUNT：该项监控出现的总次数。
- P999: 该项监控的 P999 最大值。
- P99: 该项监控的 P99 最大值。
- P90: 该项监控的 P90 最大值。
- P80: 该项监控的 P80 最大值。

示例：

上表中，在报告时间范围的 `tidb_txn_kv_write_size` 表示一共约有 181296 次事务的 KV 写入，总 kv 写入大小是 266.772 MB，其中单次事务的 KV 写入的 P999、 P99、 P90、 P80 的最大值分别为 116.913 KB、 1.996 KB、 1.905 KB、 1.805 KB。

DDL Owner

MIN_TIME	DDL OWNER
2020-05-21 14:40:00	10.0.1.13:10080

图 329: TiDB DDL Owner 报表

上表表示从 `2020-05-21 14:40:00` 开始，集群的 DDL owner 在 `10.0.1.13:10080` 节点。如果 owner 发生变更，上表会有多行数据，其中 `MinTime` 列表示已知对应 Owner 的最小时间。

> 注意：
>
> 如果 owner 信息为空，不代表这个时间段内一定没有 owner，因为这里是依靠 `ddl_worker` 的监控信息来判断 DDL owner 的。也可能是这个时间段内 `ddl_worker` 没有做任何 DDL job 导致 owner 信息为空。

TiDB 中其他部分监控表如下：

- Statistics Info：TiDB 统计信息的相关监控。
- Top 10 Slow Query：报表时间范围内 Top 10 的慢查询信息。
- Top 10 Slow Query Group By Digest：报表时间范围内 Top 10 的慢查询信息，并按照 SQL 指纹聚合。
- Slow Query With Diff Plan：报表时间范围内执行计划发生变更的 SQL 语句。

PD 相关监控信息

PD 模块相关监控的报表如下：

- Time Consumed by PD Component：PD 中相关模块的耗时监控
- Scheduled Leader/Region：报表时间范围内集群发生的 `balance-region` 和 `balance leader` 监控，比如从 `tikv_note_1` 上调度走了多少个leader，调度进了多少个leader。
- Cluster Status：集群的状态信息，包括总 TiKV 数量、总集群存储容量、Region 数量、离线TiKV 的数量等信息。
- Store Status：记录各个 TiKV 节点的状态信息，包括Region score、 leader score、 Region/leader 的数量。
- etcd Status：PD 内部的 etcd 相关信息。

TiKV 相关监控信息

TIKV 模块的相关监控报表如下：

- Time Consumed by TiKV Component：TiKV 中相关模块的耗时监控。
- Time Consumed by RocksDB：TiKV 中 RocksDB 的耗时监控。
- TiKV Error：TiKV 中各个模块相关的 error 信息。
- TiKV Engine Size：TiKV 中各个节点column family 的存储数据大小。
- Coprocessor Info：TiKV 中 Coprocessor 模块相关的监控。
- Raft Info：TiKV 中 Raft 模块的相关监控信息。
- Snapshot Info：TiKV 中 snapshot 相关监控信息。
- GC Info：TiKV 中 GC 相关的监控信息。
- Cache Hit：TiKV 中 Rocksdb 的各个缓存的命中率监控信息。

11.11.2.1.6 配置信息

配置信息中，部分模块的配置信息可以显示报告时间范围内的配置值，有部分配置则因为无法获取到历史的配置值，所以是生成报告时刻的当前配置值。

在报告时间范围内，以下表包括部分配置的在报告时间范围的开始时间的值：

- Scheduler Initial Config：PD 调度相关配置在报告开始时间的初始值。
- TiDB GC Initial Config：TiDB GC 相关配置在报告开始时间的初始值。
- TiKV RocksDB Initial Config：TiKV RocksDB 相关配置在报告开始时间的初始值。
- TiKV RaftStore Initial Config：TiKV RaftStore 相关配置在报告开始时间的初始值。

在报表时间范围内，如若有些配置被修改过，以下表包括部分配置被修改的记录：

- Scheduler Config Change History
- TiDB GC Config Change History

- TiKV RocksDB Config Change History
- TiKV RaftStore Config Change History

示例：

APPROXIMATE_CHANGE_TIME	CONFIG_ITEM	VALUE
2020-05-22T20:00:00+08:00	leader-schedule-limit	4
2020-05-22T20:07:00+08:00	leader-schedule-limit	8

图 330: Scheduler Config Change History 报表

上面报表显示，`leader-schedule-limit` 配置参数在报告时间范围内有被修改过：

- `2020-05-22T20:00:00+08:00`，即报告的开始时间 `leader-schedule-limit` 的配置值为 `4`，这里并不是指该配置被修改了，只是说明在报告时间范围的开始时间其配置值是 `4`。
- `2020-05-22T20:07:00+08:00`，`leader-schedule-limit` 的配置值为`8`，说明在`2020-05-22T20:07:00+08:00` 左右，该配置的值被修改了。

下面的报表是生成报告时，TiDB、 PD、 TiKV 的在生成报告时刻的当前配置：

- TiDB’s Current Config
- PD’s Current Config
- TiKV’s Current Config

11.11.2.2 对比报告

生成两个时间段的对比报告，其内容和单个时间段的报告是一样的，只是加入了对比列显示两个时间段的差别。下面主要介绍对比报告中的一些特有表以及如何查看对比报表。

首先在基本信息中的 Compare Report Time Range 报表会显示出对比的两个时间段：

Compare Report Time Range

T1.START_TIME	T1.END_TIME	T2.START_TIME	T2.END_TIME
2020-05-21 06:30:00	2020-05-21 06:35:00	2020-05-21 06:40:00	2020-05-21 06:45:00

图 331: Compare Report Time Range 报表

其中 `t1` 是正常时间段，或者叫参考时间段，`t2` 是异常时间段。

下面是一些慢查询相关的报表：

- Slow Queries In Time Range t2：仅出现在 `t2` 时间段但没有出现在 `t1` 时间段的慢查询。
- Top 10 slow query in time range t1：`t1` 时间段的Top10 慢查询。
- Top 10 slow query in time range t2：`t2` 时间段的Top10 慢查询。

11.11.2.2.1 DIFF_RATIO 介绍

本部分以 Instance CPU Usage 为例介绍 DIFF_RATIO。

Instance CPU Usage

INSTANCE	JOB	t1.AVG	t1.MAX	t1.MIN	t2.AVG	t2.MAX	t2.MIN	AVG_DIFF_RATIO	MAX_DIFF_RATIO	MIN_DIFF_RATIO
172.16.5.40:10089	tidb	410%	410%	410%	1240%	1240%	1240%	2.02	2.02	2.02
172.16.5.40:20151	tikv	221%	221%	221%	617%	617%	617%	1.79	1.79	1.79
172.16.5.40:20379	pd	82%	82%	82%	78%	78%	78%	-0.05	-0.05	-0.05

图 332: Compare Instance CPU Usage 报表

- t1.AVG、t1.MAX、t1.Min 分别是 t1 时间段内 CPU 使用率的平均值、最大值、最小值。
- t2.AVG、t2.MAX、t2.Min 分别是 t2 时间段内 CPU 使用率的平均值、最大值、最小值。
- AVG_DIFF_RATIO 表示 t1 和 t2 时间段平均值的 DIFF_RATIO。
- MAX_DIFF_RATIO 表示 t1 和 t2 时间段最大值的 DIFF_RATIO。
- MIN_DIFF_RATIO 表示 t1 和 t2 时间段最小值的 DIFF_RATIO。

DIFF_RATIO 表示两个时间段的差异大小，有以下几个取值方式：

- 如果该监控仅在 t2 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 1。
- 如果监控项仅在 t1 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 -1。
- 如果t2时间段的值比t1 时间段的值大，则 DIFF_RATIO = (t2.value / t1.value)- 1。
- 如果 t2 时间段的值比 t1 时间段的值小，则 DIFF_RATIO = 1 - (t1.value / t2.value)。

例如上表中，tidb 节点的平均 CPU 使用率在 t2 时间段比 t1 时间段高 2.02 倍，2.02 = 1240/410 - 1。

11.11.2.2.2 Maximum Different Item 报表介绍

Maximum Different Item 的报表是对比两个时间段的监控项后，按照监控项的差异大小排序，通过这个表可以很快发现两个时间段哪些监控的差异最大。示例如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	28916.85	61.33	1773532	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	7219.00		7219	TOTAL_COUNT
overview, total_time_consume	tidb_parse	general	1201.00		1201	TOTAL_COUNT
TiDB, tidb_time_consume	tidb_slow_query_cop_wait	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	tidb_slow_query_cop_process	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	Slow query	172.16.5.40:10089	799.00	1	800	TOTAL_COUNT
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	534.43	15.822 MB	8.273 GB	TOTAL_VALUE
overview, total_time_consume	tidb_distsql_execution	dag	470.06	0.68	320.32	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	412.29	0.07%	28.93%	MIN
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	198.39	1067055.38	212755381.33	TOTAL_VALUE
overview, total_time_consume Expand	tidb_ddl_worker	drop view	139.00		139	TOTAL_COUNT
overview, total_error Expand	tikv_storage_async_request_error	snapshot	89.00		89	TOTAL_COUNT
overview, total_time_consume Expand	Coprocessor handling request	index	78.99	0.67	53.59	TOTAL_TIME
overview, total_time_consume	tidb_cop	172.16.5.40:10089	75.11	5.93	451.31	TOTAL_TIME
overview, total_time_consume	KV request	Cop	75.10	5.93	451.26	TOTAL_TIME
overview, total_time_consume Expand	Coprocessor request	select	73.12	4.73	350.6	TOTAL_TIME
PD, pd_time_consume Expand	pd_region_heartbeat	172.16.5.40:20150,1	-73.00	73		TOTAL_COUNT
overview, total_time_consume Expand	tikv_grpc_messge	coprocessor	71.75	5.56	404.49	TOTAL_TIME
TiDB, tidb_time_consume	tidb_ddl_update_self_version	172.16.5.40:10089,ok	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	TiDB meta operation	update_ddl_job	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	tidb_owner_handle_syncer	update_global_version	71.00		71	TOTAL_COUNT
TiDB, tidb_time_consume Expand	tidb_query	172.16.5.40:10089,internal	-62.84	15.96	0.25	P999
TiKV, scheduler_info Expand	Scheduler stage	172.16.5.40:20151,scan_lock,snapshot_ok	55.45		55.45	TOTAL_VALUE
overview, total_time_consume Expand	tikv_grpc_messge	kv_scan_lock	55.00		55	TOTAL_COUNT
TiKV, tikv_time_consume Expand	tikv_scheduler_command	172.16.5.40:20151,scan_lock	55.00		55	TOTAL_COUNT
overview, total_time_consume Expand	PD client cmd	get_region	44.00		44	TOTAL_COUNT
overview, total_time_consume Expand	tikv_cop_wait	select	39.40	171	6908	TOTAL_COUNT
TiDB, tidb_time_consume	Schema load	172.16.5.40:10089	37.50	0.02	0.77	TOTAL_TIME
overview, total_time_consume	Schema load	172.16.5.40:10089	30.25	0.008	0.25	P999

图 333: Maximum Different Item 报表

- `Table`：表示这个监控项来自于对比报告中报表，如 `TiKV, coprocessor_info` 表示是 TiKV 组件下的 `coprocessor_info` 报表。
- `METRIC_NAME`：监控项名，点击 expand 可以查看该监控的不同label 的差异对比。
- `LABEL`：监控项对应的 label。比如 `TiKV Coprocessor scan` 监控项有两个label，分别是instance、 req、 tag、 sql_type，分别表示为 TiKV 地址、 请求类型、 操作类型和操作的column family。
- `MAX_DIFF`：差异大小，取值为 `t1.VALUE` 和 `t2.VALUE` 的 `DIFF_RATIO` 计算。

可以从上表中发现，`t2` 时间段比 `t1` 时间段多出了大量的 Coprocessor 请求，TiDB 的解析 SQL (parse) 时间也多了很多。

11.11.3 使用TiDB Dashboard 诊断报告定位问题

本文介绍如何使用TiDB Dashboard 诊断报告定位问题。

11.11.3.1 对比诊断功能示例

对比报告中对比诊断的功能，通过对比两个时间段的监控项差异来尝试帮助 DBA 定位问题。先看以下示例。

11.11.3.1.1 大查询/写入导致 QPS 抖动或延迟上升诊断

示例 1

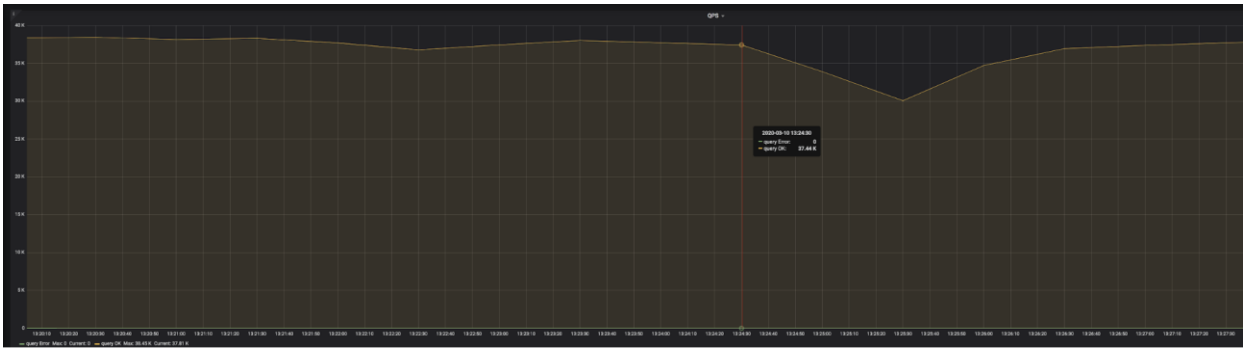


图 334: QPS 图

上图是 go-ycsb 压测的监控。可以发现，在 2020-03-10 13:24:30 时，QPS 突然开始下降，3 分钟后，QPS 开始恢复正常。

生成以下两个时间范围的对比报告：

- t1: 2020-03-10 13:21:00 - 2020-03-10 13:23:00，正常时间段，又叫参考时间段。
- t2: 2020-03-10 13:24:30 - 2020-03-10 13:27:30，QPS 开始下降的异常时间段。

这里两个时间间隔都是 3 分钟，因为抖动的影响范围为 3 分钟。因为诊断时会用一些监控的平均值做对比，所有间隔时间太长会导致平均值差异不明显，无法准确定位问题。

生成报告后查看 Compare Diagnose 报表内容如下：

Compare Diagnose

Automatically diagnose the cluster problem by compare with the refer time.

RULE	DETAIL
big-query fold	may have big query in diagnose time range
|--	tidb_qps,172.16.5.40:10089: ↓ 0.93 (34927.00 / 37658.00)
|--	tidb_query_duration,172.16.5.40:10089: ↑ 1.54 (0.25 / 0.16)
|--	tidb_cop_duration,172.16.5.40:10089: ↑ 2.48 (0.16 / 0.06)
|--	tidb_kv_write_num,172.16.5.40:10089: ↑ 7.61 (1766.40 / 232.25)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:22151: ↑ 136446.22 (98242004.00 / 720.01)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:23151: ↑ 65079325.09 (65079325.09 / 0.00)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:20151: ↑ 46976.83 (101469210.67 / 2159.98)
|--	pd_operator_step_finish_total_count,TransferLeader: ↑ 2.45 (36.00 / 14.67)
|--	try to check the slow query only appear in diagnose time range with sql: SELECT * FROM (SELECT count(*), min(time), sum(query_time) AS sum_query_time, sum(Process_time) AS sum_process_time, sum(Wait_time) AS sum_wait_time, sum(Commit_time), sum(Request_count), sum(process_keys), sum(Write_keys), max(Cop_proc_max), min(query),min(prev_stmt), digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:24:30' AND time < '2020-03-10 13:27:30' AND Is_internal = false GROUP BY digest) AS t1 WHERE t1.digest NOT IN (SELECT digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:21:00' AND time < '2020-03-10 13:24:00' GROUP BY digest) ORDER BY t1.sum_query_time DESC limit 10;

图 335: 对比诊断结果

上面诊断结果显示，在诊断的时间内可能有大查询，下面的每一行的含义是：

- tidb_qps：QPS 下降 0.93 倍。
- tidb_query_duration：P999 的查询延迟上升 1.54 倍。
- tidb_cop_duration：P999 的 COP 请求的处理延迟上升 2.48 倍。
- tidb_kv_write_num：P999 的 tidb 的事务写入 kv 数量上升 7.61 倍。
- tikv_cop_scan_keys_total_nun：TiKV 的 coprocessor 扫描 key/value 的数量分别在 3 台 TiKV 上有很大的提升。
- pd_operator_step_finish_total_count 中，transfer leader 的数量上升 2.45 倍，说明异常时间段的调度比正常时间段要高。
- 提示可能有慢查询，并提示用 SQL 查询TiDB 的慢日志。在TiDB 中执行结果如下：

```
SELECT * FROM (SELECT count(*), min(time), sum(query_time) AS sum_query_time, sum(Process_time)
    '→ AS sum_process_time, sum(Wait_time) AS sum_wait_time, sum(Commit_time), sum(Request_count
    '→ ), sum(process_keys), sum(Write_keys), max(Cop_proc_max), min(query),min(prev_stmt),
    '→ digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:24:30' AND
    '→  time < '2020-03-10 13:27:30' AND Is_internal = false GROUP BY digest) AS t1 WHERE t1.
    '→ digest NOT IN (SELECT digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '
    '→ 2020-03-10 13:21:00' AND time < '2020-03-10 13:24:00' GROUP BY digest) ORDER BY t1.
    '→ sum_query_time DESC limit 10\G
***************************[ 1. row ]***************************
count(*)            | 196
min(time)           | 2020-03-10 13:24:30.204326
sum_query_time      | 46.878509117
sum_process_time    | 265.924
sum_wait_time       | 8.308
sum(Commit_time)    | 0.926820886
sum(Request_count) | 6035
```

```
sum(process_keys)   | 201453000
sum(Write_keys)     | 274500
max(Cop_proc_max)   | 0.263
min(query)          | delete from test.tcs2 limit 5000;
min(prev_stmt)      |
digest              | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

可以发现，从 13:24:30 开始有一个批量删除的大写入，一共执行了 196 次，每次删除 5000 行数据，总共耗时 46.8 秒。

示例 2

如果大查询一直没执行完，就不会记录慢日志，但仍可以进行诊断，示例如下：



图 336: QPS 图

上图中，也是在跑 go-ycsb 的压测。可以发现，在 2020-03-08 01:46:30 时，QPS 突然开始下降，并且一直没有恢复。

生成以下两个时间范围的对比报告：

- t1: 2020-03-08 01:36:00 - 2020-03-08 01:41:00，正常时间段，又叫参考时间段。
- t2: 2020-03-08 01:46:30 - 2020-03-08 01:51:30，QPS 开始下降的异常时间段。

生成报告后看 Compare Diagnose 报表的内容如下：

Compare Diagnose

Automatically diagnose the cluster problem by compare with the refer time.

RULE	DETAIL
big-query fold	may have big query in diagnose time range
|--	tidb_qps,172.16.5.40:10089: ↓ 0.85 (31881.00 / 37674.00)
|--	tidb_query_duration,172.16.5.40:10089: ↑ 1.35 (0.06 / 0.04)
|--	tidb_cop_duration,172.16.5.40:10089: ↑ 3.78 (0.08 / 0.02)
|--	tidb_kv_write_num,172.16.5.40:10089: ↑ 511.74 (511.74 / 0.00)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:20151: ↑ 1277.21 (6270285.33 / 4909.36)
|--	try to check the slow query only appear in diagnose time range with sql: SELECT * FROM information_schema.cluster_log WHERE type='tidb' AND time >= '2020-03-08 01:46:30' AND time < '2020-03-08 01:51:30' AND level = 'warn' AND message LIKE '%expensive_query%'

图 337: 对比诊断结果

上面诊断结果的最后一行显示可能有慢查询，并提示用 SQL 查询TiDB 日志中的 expensive query。 在TiDB 中执行结果如下：

```
> SELECT * FROM information_schema.cluster_log WHERE type='tidb' AND time >= '2020-03-08 01:46:30
    '→ ' AND time < '2020-03-08 01:51:30' AND level = 'warn' AND message LIKE '%expensive_query%
    '→ '\G
TIME     | 2020/03/08 01:47:35.846
TYPE     | tidb
INSTANCE | 172.16.5.40:4009
LEVEL    | WARN
MESSAGE  | [expensivequery.go:167] [expensive_query] [cost_time=60.085949605s] [process_time=2.52
    '→ s] [wait_time=2.52s] [request_count=9] [total_keys=996009] [process_keys=996000] [
    '→ num_cop_tasks=9] [process_avg_time=0.28s] [process_p90_time=0.344s] [process_max_time
    '→ =0.344s] [process_max_addr=172.16.5.40:20150] [wait_avg_time=0.000777777s] [wait_p90_time
    '→  =0.003s] [wait_max_time=0.003s] [wait_max_addr=172.16.5.40:20150] [stats=t_wide:pseudo]
    [ '→ conn_id=19717] [user=root] [database=test] [table_ids="[80,80]"] [txn_start_ts
    '→ =415132076148785201] [mem_max="23583169 Bytes (22.490662574768066 MB)"] [sql="select
    '→ count(*) from t_wide as t1 join t_wide as t2 where t1.c0>t2.c1 and t1.c2>0"]
```

以上查询结果显示，在 172.16.5.40:4009 这台TiDB 上，2020/03/08 01:47:35.846 有一个已经执行了 60s 但还没有执行完的 expensive_query。

11.11.3.2 用对比报告定位问题

诊断有可能是误诊，使用对比报告或许可以帮助 DBA 更快速的定位问题。参考以下示例。

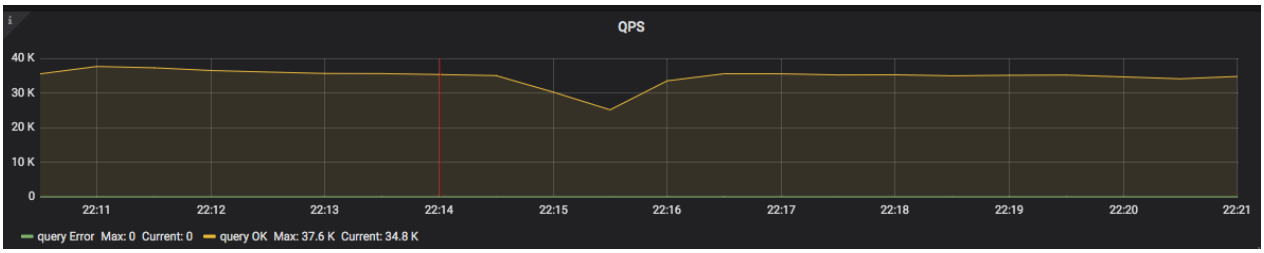


图 338: QPS 图

上图中，也是在跑 go-ycsb 的压测，可以发现，在 2020-05-22 22:14:00 时，QPS 突然开始下降，大概在持续 3 分钟后恢复。

生成以下 2 个时间范围的对比报告：

- t1: 2020-05-22 22:11:00 - 2020-05-22 22:14:00，正常时间段。
- t2: 2020-05-22 22:14:00 - 2020-05-22 22:17:00，QPS 开始下降的异常时间段。

生成对比报告后，查看 Max diff item 报表，该报表对比两个时间段的监控项后，按照监控项的差异大小排序，这个表的结果如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiDB, transaction	Transaction KV write count	172.16.5.40:10089	-21616.00	43234	2	P999
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	10499.24	64	672015.48	TOTAL_VALUE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	4264.45	15431.85	65823838.67	TOTAL_VALUE
TiDB, transaction	Transaction KV write size	172.16.5.40:10089	-2702.69	5.278 MB	1.999 KB	P999
overview, total_time_consume	Coprocessor handling request	select	2132.50	0.06	128.01	TOTAL_TIME
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	1535.00	1.792 MB	2.688 GB	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	1447.00		1447	TOTAL_COUNT
TiKV, scheduler_info Expand	tikv_scheduler_keys_written	172.16.5.40:20151,commit	-860.00	861	1	P99
TiKV, tikv_time_consume	Coprocessor request	172.16.5.40:20151,select	853.87	0.15	128.23	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	667.27	0.11%	73.51%	AVG
TiKV, tikv_time_consume	Coprocessor handling request	172.16.5.40:20151,index	444.00	0.002	0.89	P999
TiKV, tikv_time_consume Expand	tikv_grpc_messge	172.16.5.40:20151,coprocessor	419.97	0.33	138.92	TOTAL_TIME
load, node_load_info	tikv_disk_read_bytes	172.16.5.40:19110,sda	-273.41	0.267 KB	0.000 KB	MAX
overview, total_time_consume Expand	KV request	Cop	244.29	0.66	161.89	TOTAL_TIME

图 339: 对比结果

从上面结果可以看出 t2 时间段新增了很多 Coprocessor 请求，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 - t2 整个时间段内都在进行 go-ycsb 的压测，然后在 t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了出现很多的cop 请求。

这种大查询执行时间超过慢日志的阈值后也会记录在慢日志里面，可以继续查看 Slow Queries In Time Range t2 报表查看是否有一些慢查询。一些在 t1 时间段存在的查询，可能在 t2 时间段内就变成了慢查询，是因为t2 时间段的其他负载影响导致该查询的执行变慢。

11.12 TiDB Dashboard 监控页面

从 TiDB Dashboard 监控页面，你可以查看性能分析和优化工具 Performance Overview 面板。借助 Performance Overview 面板，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，你可以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。详情请参考TiDB 性能分析和优化方法。

11.12.1 访问页面

登录 Dashboard 后点击左侧导航的 Monitoring （监控）可以进入此功能页面：

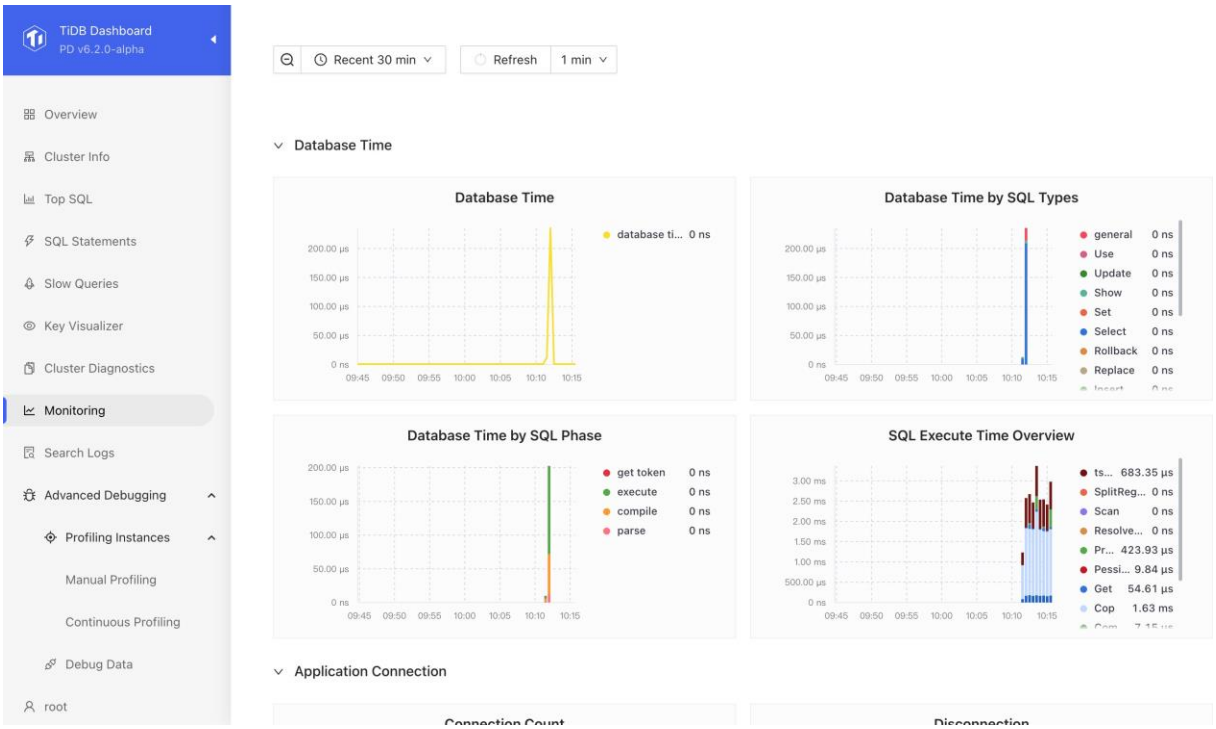


图 340: Monitoring page

如果你使用TiUP 部署TiDB，你也可以在Grafana 上查看 Performance Overview 面板。监控架构参见TiDB 监控框架概述。



图 341: performance overview

11.12.2 面板监控指标详解

Performance Overview 面板按总分结构对TiDB、TiKV 和 PD 的性能指标进行了编排组织，包含以下三部分内容：

- 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。

以下为 Performance Overview 面板监控说明：

11.12.2.1 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

11.12.2.2 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。

11.12.2.3 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。

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，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。

11.12.2.4 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

11.12.2.5 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）

11.12.2.6 Queries Using Plan Cache OPS

Queries Using Plan Cache OPS：所有 TiDB 实例每秒使用 Plan Cache 的查询数量

11.12.2.7 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。

11.12.2.8 Connection Count

- total：所有TiDB 的连接数
- active connections：所有TiDB 总的活跃连接数
- 各个TiDB 的连接数

11.12.2.9 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

11.12.2.10 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

11.12.2.11 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

11.12.2.12 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
- 99-not-in-txn：没有处于事务中，空闲连接的 P99 持续时间

11.12.2.13 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。

11.12.2.14 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。

11.12.2.15 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。

11.12.2.16 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

11.12.2.17 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

11.12.2.18 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。

11.13 TiDB Dashboard 日志搜索页面

该页面上允许用户在集群中搜索所有节点上的日志，在页面上预览搜索结果和下载日志。

11.13.1 访问

登录 Dashboard 后点击左侧导航的日志搜索可以进入此功能页面：

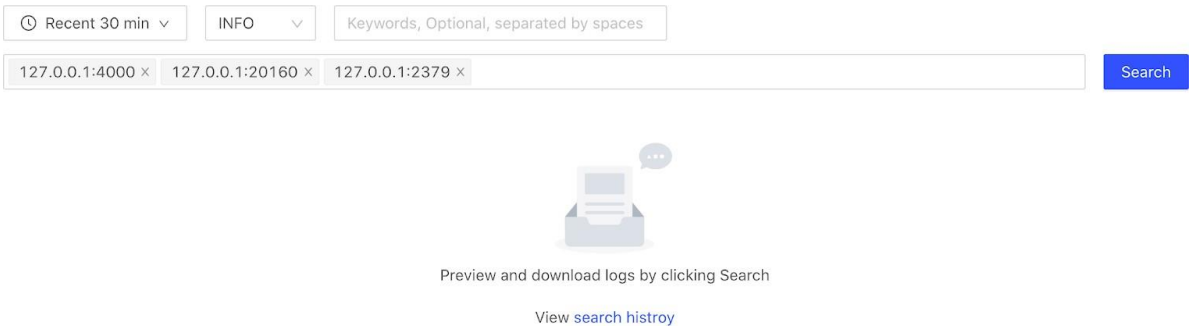


图 342: 日志搜索首页

该页面提供 4 个搜索参数，包括：

- 时间范围：限定搜索时间范围内的日志，默认值为最近 30 分钟
- 日志等级：限定最低日志等级，搜索该日志等级以上的所有日志，默认值为INFO 等级
- 关键词：任何合法的任何字符串，可选，关键词之间以空格分割，支持正则表达式（大小写不敏感）
- 组件：选择要搜索的集群组件，多选，非空，默认选中所有组件

点击搜索按钮后，会进入搜索结果的详情页。

11.13.2 搜索结果详情

搜索结果详情页面如下图所示：

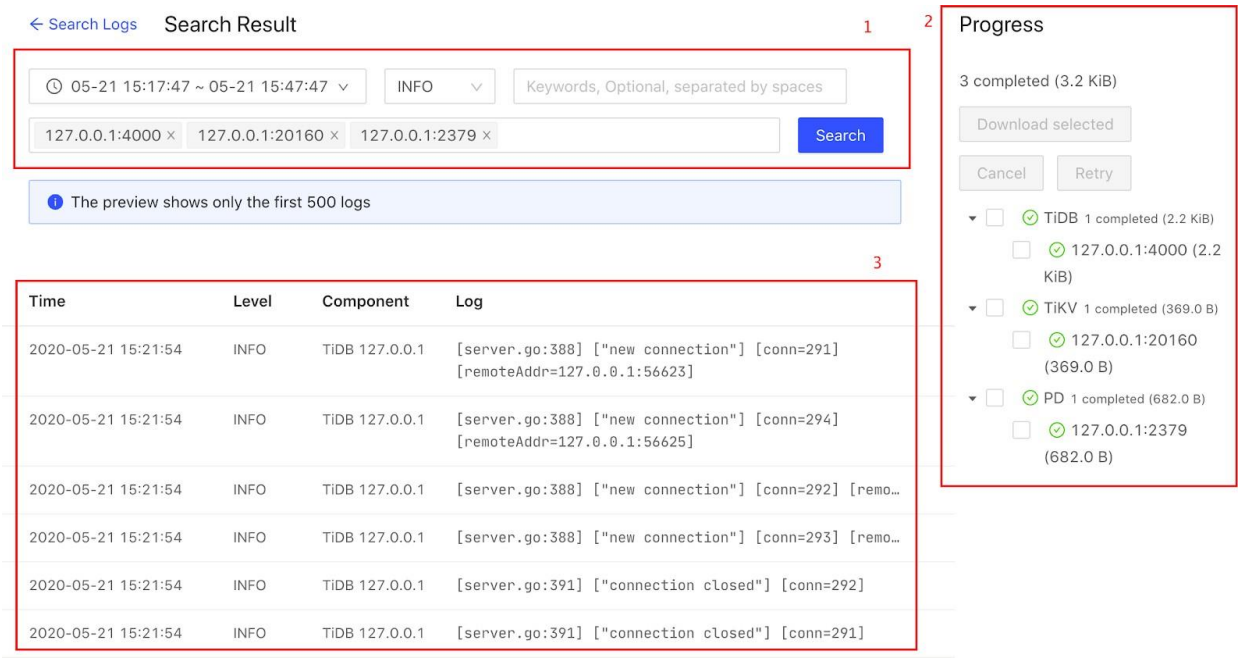


图 343: 搜索结果

整个页面分为三个部分：

- 搜索参数选项：与搜索页面的参数选项相同，可以在表单中重新选择参数并开始一次新的搜索（对应上图中框选的区域 1）
- 搜索进度：在页面右侧，展示当前搜索的进度，包括每个节点日志搜索状态和统计等（对应上图中框选的区域 2）
- 搜索结果（对应上图中框选的区域 3）：
  - 时间：日志产生的时间，时区与前端用户所处时区相同
  - 日志等级：日志等级
  - 组件类型：显示组件名及其地址
  - 日志：每条日志记录的正文部分，不包含日志的时间和日志等级。过长的日志会自动截断，鼠标单击该行可以展开查看完整内容，完整日志最长显示 512 个字符

> 注意：
>
> 在该页面上最多只显示 500 条搜索结果，完整的搜索结果可以通过下载得到。

11.13.2.1 搜索进度

在搜索进度区域中，对一个节点的搜索称为一次搜索任务，对搜索任务的状态分为：

- 运行中：开始搜索后，所有任务会进入运行中状态

- 成功：任务完成后自动转到成功，此时日志已缓存在 Dashboard 后端所在的本地磁盘中，可以提供给前端下载
- 失败：用户主动取消，或者某种原因报错退出的任务进入失败状态。任务失败时会自动清理本地临时文件

搜索进度区域包含三个控制按钮：

- 下载选中日志：下载被勾选组件的日志（只有已完成的才能被勾选），返回 tar 文件，解压得到一个或多个 zip 文件（每个组件对应一个 zip 文件），解压 zip 得到 log 文本文件
- 取消：取消所有正在运行的任务，只能在有运行中的任务时才能点击
- 重试：重试所有失败的任务，只有在有失败任务且无正在运行的任务时才能点击

11.13.3 搜索历史列表

在日志搜索首页点击查看搜索历史链接，进入搜索历史列表页面：

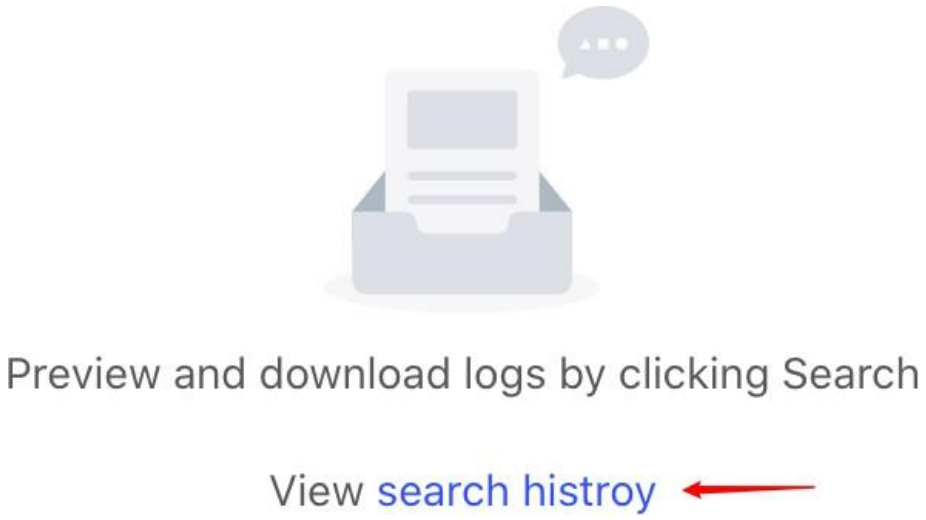


图 344: 搜索历史入口

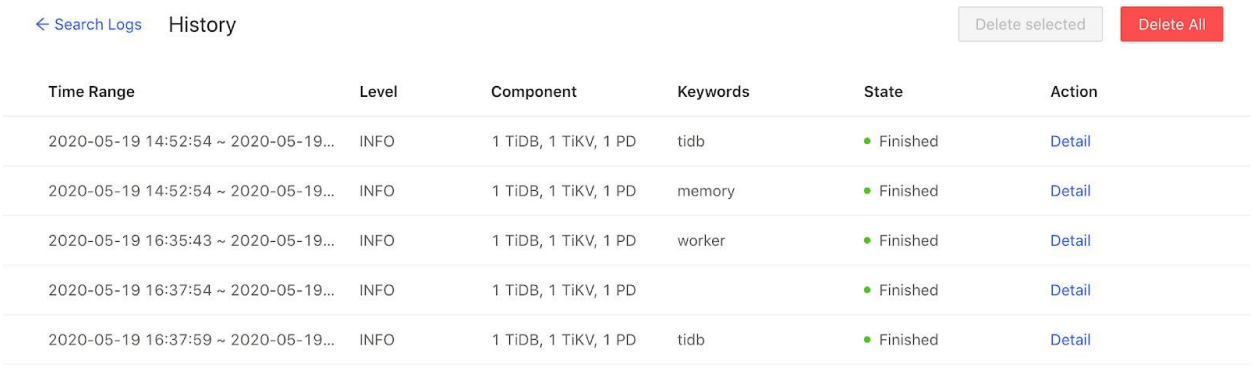


图 345: 搜索历史列表

该列表显示每次搜索的日志的时间范围、日志等级、组件、关键字以及搜索状态等信息。点击操作列的查看详情链接将跳转到此次搜索的结果详情页面。

可以对不再需要的搜索历史执行删除操作，点击右上角的删除全部任务，或者先选中需要删除的某些行后再点击删除选中的任务进行删除：

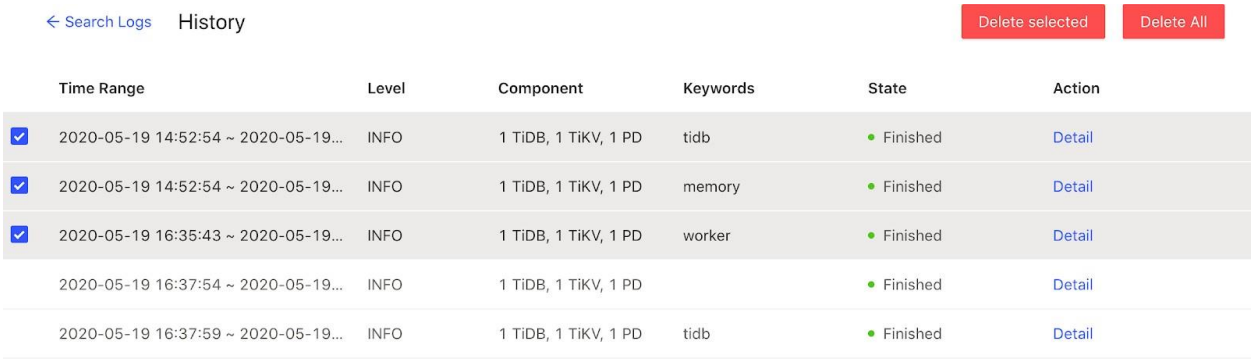


图 346: 删除搜索历史

11.14 TiDB Dashboard 资源管控页面

为使用资源管控(Resource Control)特性实现资源隔离，集群管理员可以定义资源组(Resource Group)，通过资源组限定配额。在进行资源规划之前，你需要了解集群的整体容量。该页面可以帮助你查看资源管控相关信息，以便预估集群容量，更好地进行资源配置。

11.14.1 访问方式

可以通过以下两种方法访问资源管控页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击资源管控(Resource Manager)。

- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/resource_manager（将 127.0.0.1:2379 替换为你的实际 PD 地址和端口）。

11.14.2 资源管控详情

资源管控详情页面如下图所示：

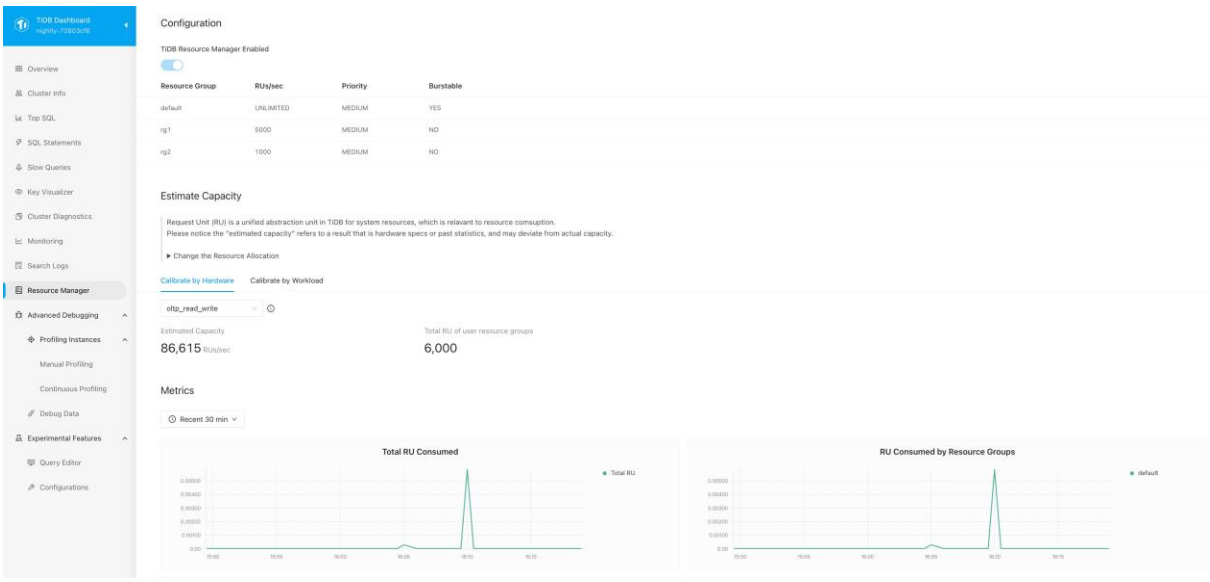


图 347: TiDB Dashboard: Resource Manager

资源管控详情页包含以下三个部分：

- 配置(Configuration)：数据来自于TiDB 的 RESOURCE_GROUPS 表中所有资源组的信息。参见 RESOURCE_GROUPS 文档。
- 容量估算(Estimate Capacity)：在进行资源规划之前，你需要了解集群的整体容量。目前提供两种估算方式：
  - 基于硬件部署估算容量
  - 根据实际负载估算容量
- 监控指标(Metrics)：通过观察面板上的指标，可以了解当前集群整体的资源消耗状态。

11.14.3 容量估算

在进行资源规划之前，你需要了解集群的整体容量。目前提供两种估算方式预估当前集群的Request Unit (RU)的容量：

- 基于硬件部署估算容量(Calibrate by Hardware)
  目前提供了以下负载类型供选择：

- `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。

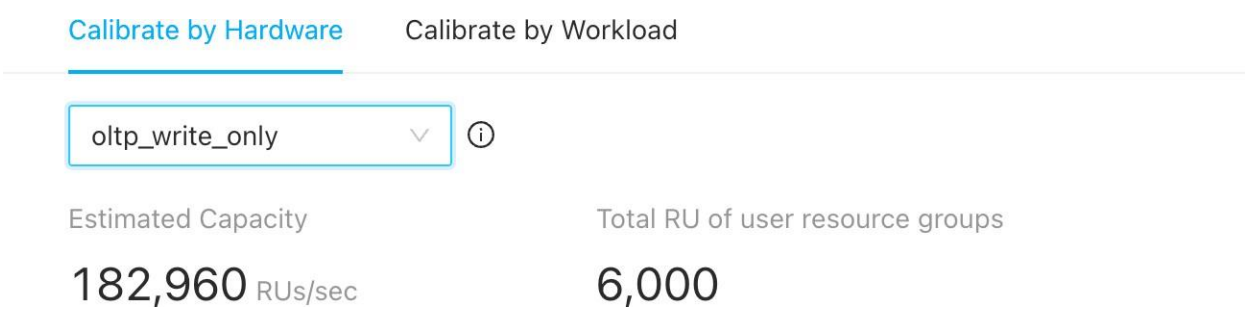


图 348: 基于硬件部署估算容量

用户资源分组总请求单元 (`Total RU of user resource groups`) 表示当前除 `` `default` `` 用户外的 RU 总量。
↳ 当该数值小于容量估算值时，系统会发出提醒。系统预定义的 `` `default` `` 资源组默认拥有无限用量。
↳ 当所有用户都属于 `` `default` `` 资源组时，资源分配方式与关闭资源管控时相同。

- 根据实际负载估算容量(Calibrate by Workload)

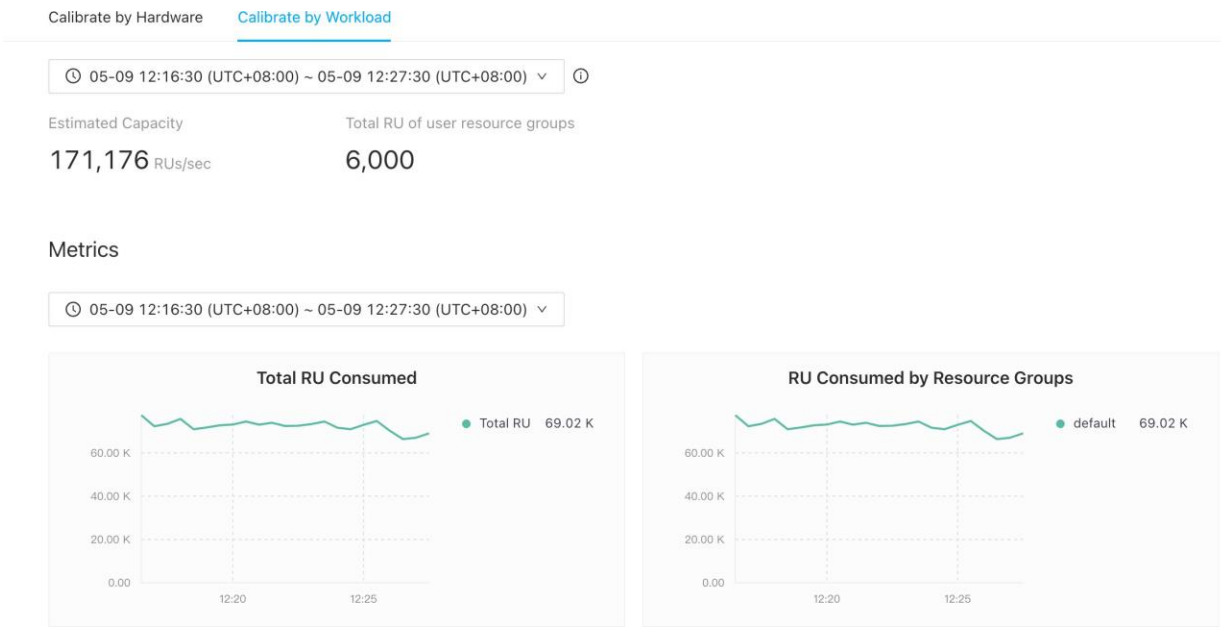


图 349: 根据实际负载估算容量

可以选择 10 分钟至 24 小时的时间范围进行预估。时区与前端用户所处时区相同。

- 如果时间窗口范围不满足 10 分钟至 24 小时的条件，会报错 `Error 1105 (HY000): the duration of calibration is too short, which could lead to inaccurate output. Please make the duration between 10m0s and 24h0m0s`。
- 如果时间窗口范围内的负载过低，会报错 `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`。

可以通过监控指标中的 CPU Usage 选择合适的时间范围。

11.14.4 监控指标

通过观察面板上的指标，可以了解当前集群整体的资源消耗状态。监控指标及其含义如下：

- Total RU Consumed：实时统计的 Request Unit 总消耗量
- RU Consumed by Resource Groups：以资源组为单位进行实时统计的Request Unit 消耗数量
- TiDB
  - CPU Quota：TiDB 最大 CPU 占用率
  - CPU Usage：所有 TiDB 实例 CPU 占用率
- TiKV
  - CPU Quota：TiKV 最大 CPU 占用率
  - CPU Usage：所有 TiKV 实例 CPU 占用率
  - IO MBps：所有 TiKV 实例的I/O 吞吐量

11.15 实例性能分析

11.15.1 TiDB Dashboard 实例性能分析- 手动分析页面

> 注意：
> 该功能面向数据库技术专家，建议非专家用户在 PingCAP 专业技术人员的指导下使用该功能。

该页面允许用户按需地一键收集TiDB、TiKV、PD、TiFlash 各个实例的当前性能数据。收集到的性能数据可显示为火焰图或有向无环图形式。

通过这些性能数据，技术专家可以分析实例当前的 CPU、内存等资源消耗细节，帮助解决诸如当前 CPU 开销较高、内存占用较大、进程不明原因卡死等复杂问题。

启动分析后，TiDB Dashboard 将会收集当前一小段时间（默认 30 秒）的性能数据，因而只能用于分析集群当前正在面临的问题，对于当前已经不再复现的问题没有显著效果。若你想要收集或分析过去任意时刻性能数据，不希望每次都人工介入按需分析，请使用持续性能分析功能。

11.15.1.1 支持的性能数据

目前支持收集并分析以下性能数据：

- CPU：TiDB、 TiKV、 PD、 TiFlash 实例上各个内部函数的 CPU 开销情况

> ARM 环境中暂不支持对TiKV 和 TiFlash 的 CPU 开销情况进行分析。

- Heap：TiDB、 PD 实例上各个内部函数的内存占用开销情况
- Mutex：TiDB、 PD 实例上各个处于等待状态的 Mutex 情况
- Goroutine：TiDB、 PD 实例上各个Goroutine 的运行状态及调用栈情况

11.15.1.2 访问页面

可以通过以下两种方法访问实例性能分析页面：

- 登录TiDB Dashboard 后，在左侧导航栏中点击高级调试(Advanced Debugging) > 实例性能分析(Profile Instances) > 手动分析(Manual Profiling)：

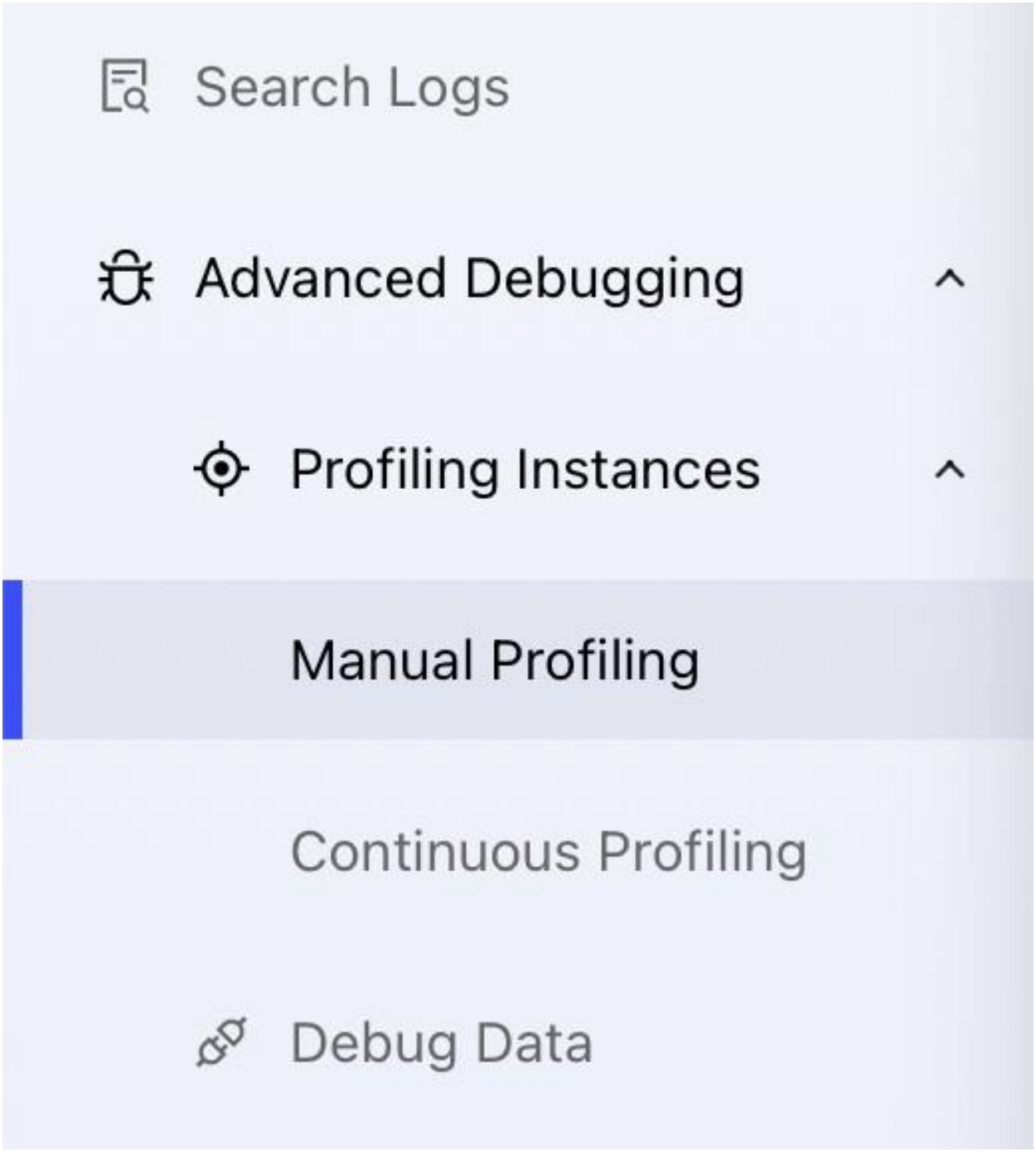


图 350: 访问页面

- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/instance_profiling（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

11.15.1.3 开始性能分析

在实例性能分析页面，选择至少一个目标实例和分析类型，确定性能分析时长（默认为 30 秒）。点击开始分析 (Start Profiling)：

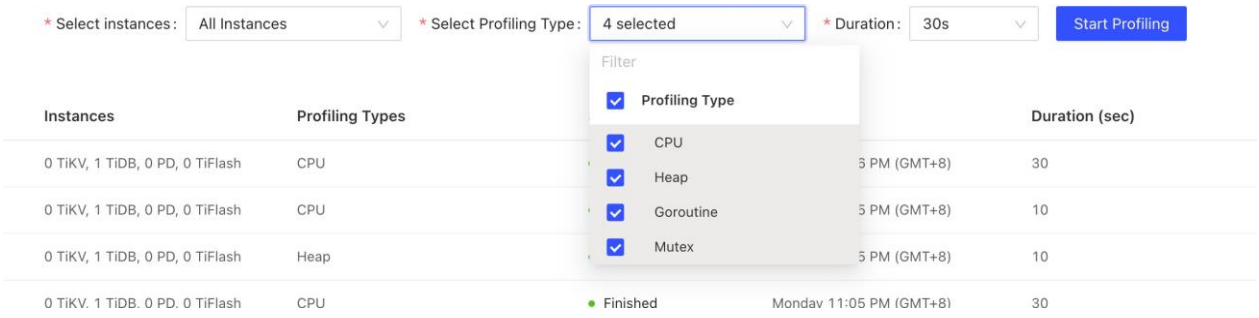


图 351: 开始分析

在已经启用了持续性能分析的集群上，手动性能分析不再可用。若要在启用了持续性能分析功能的集群上获取当前时刻的集群性能数据，请查看持续性能分析页面上最近一次的分析结果。

11.15.1.4 查看性能分析状态

开始性能分析后，可以看到实时性能分析状态和进度：

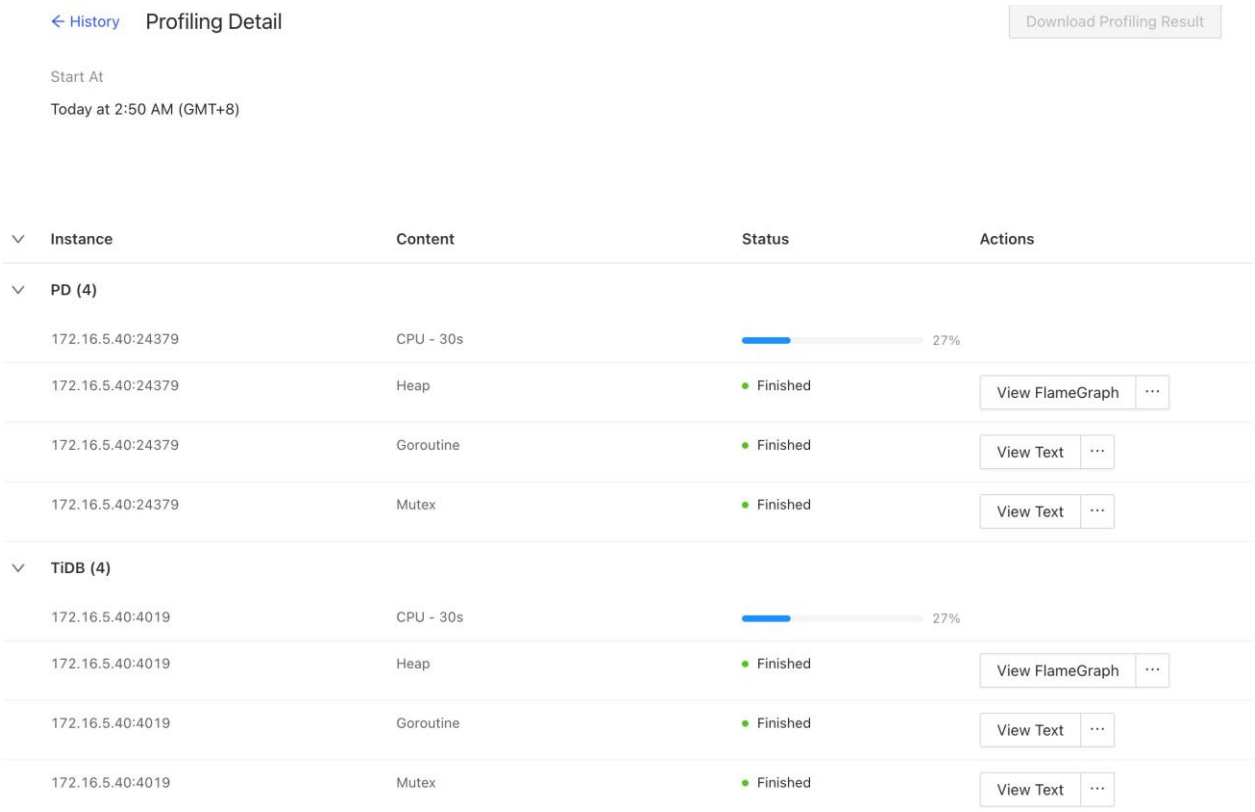


图 352: 实时状态

性能分析会在后台运行，刷新或退出当前页面不会终止已经运行的性能分析任务。

11.15.1.5 下载性能数据

所有实例的性能分析都完成后，可点击右上角下载按钮(Download Profiling Result)打包下载所有性能数据：

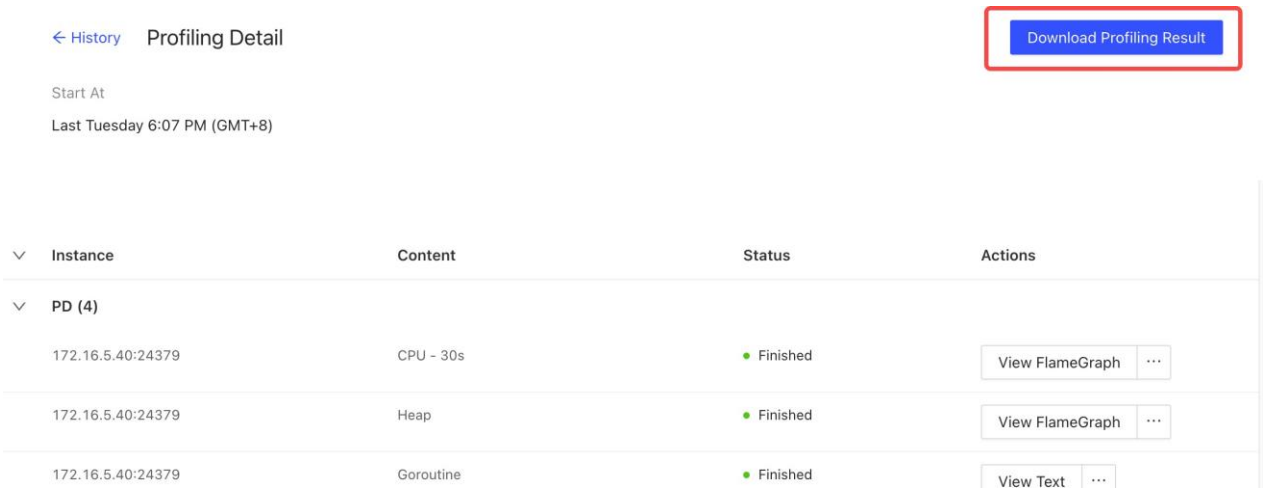


图 353: 下载分析结果

也可以点击列表中的单个实例查看其性能数据，或者悬浮到··· 按钮上下载原始数据：

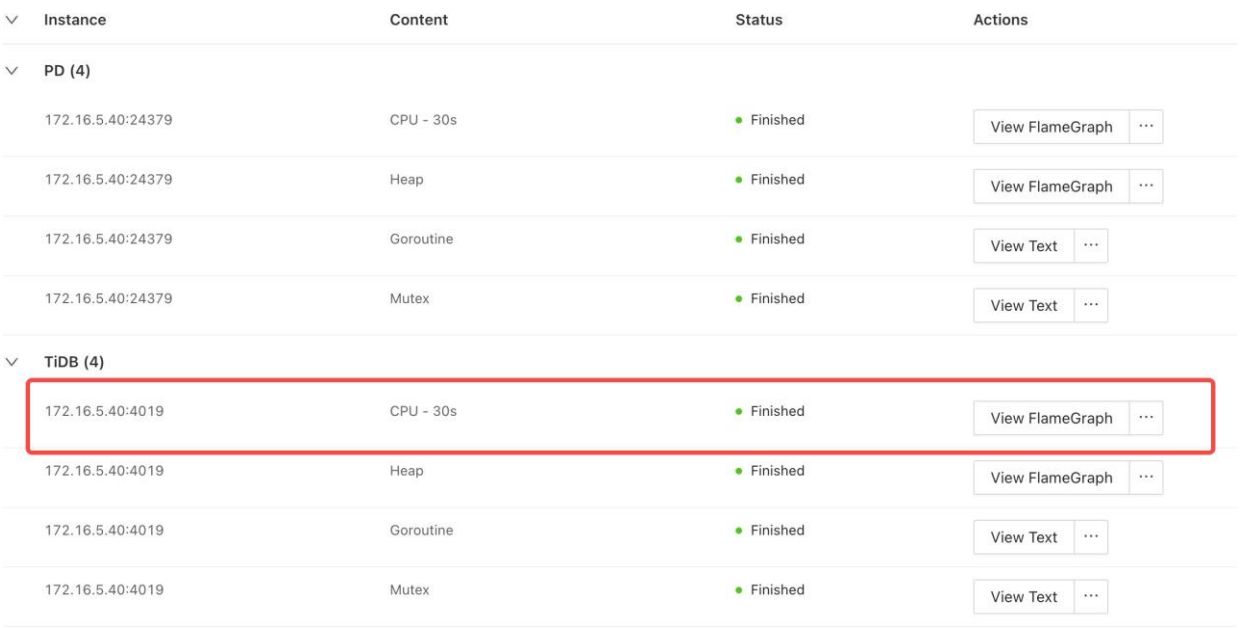


图 354: 在线查看分析结果

11.15.1.6 查看历史性能数据

在性能分析页面下方，列出了你手动发起的性能分析历史。点击任意一行，即可查看其状态详情：

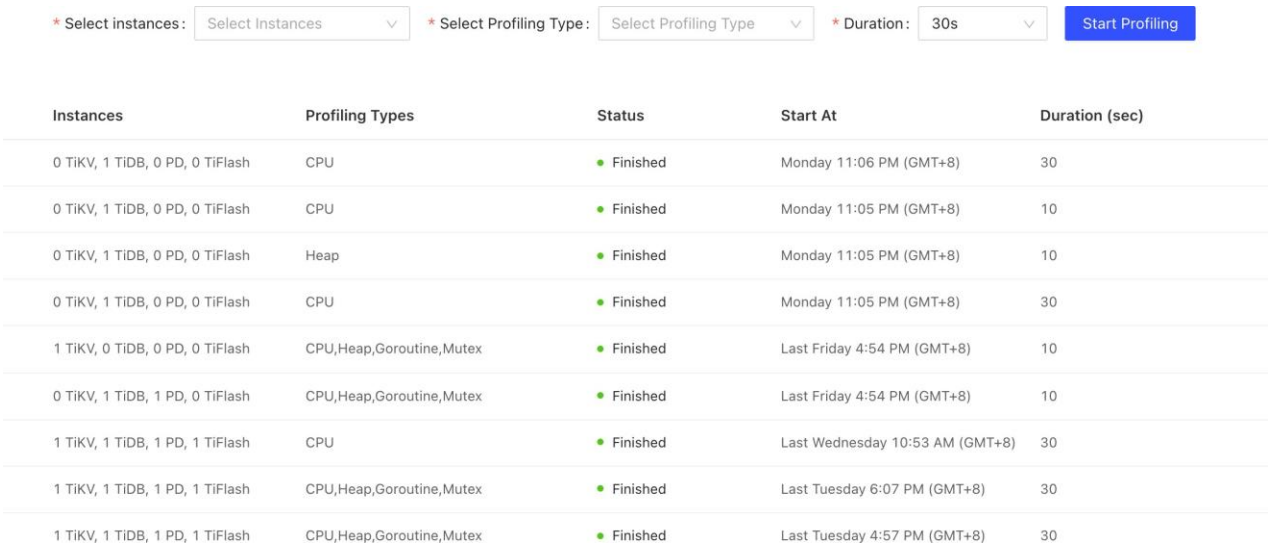


图 355: 历史列表

关于状态详情页的操作，参见查看性能分析状态章节。

11.15.2 TiDB Dashboard 实例性能分析- 持续分析页面

> **注意：**
>
> 该功能面向数据库技术专家，建议非专家用户在 PingCAP 专业技术人员的指导下使用该功能。

持续性能分析功能可以持续地收集TiDB、TiKV、PD 各个实例的性能数据。收集到的性能数据可显示为火焰图或有向无环图形式。

通过这些性能数据，技术专家可以分析实例的 CPU、内存等资源消耗细节，帮助解决诸如某一时刻 CPU 开销较高、内存占用较大、进程不明原因卡死等复杂问题。即使这类问题无法得到复现，技术专家也可以通过查看当时记录下的历史性能数据，还原问题现场，对当时的问题进行细致分析，从而有效缩短平均故障恢复时间。

11.15.2.1 与手动性能分析的区别

持续性能分析是手动性能分析的强化功能，它们都能用于收集和分析各个实例不同维度的性能数据，主要区别如下：

- 手动性能分析仅在用户发起分析的时刻收集一小段时间（如 30 秒）的性能数据，持续性能分析开启后会持续收集性能数据。

- 手动性能分析只能用于分析集群当前的问题，持续性能分析既可以用于分析集群当前问题，也可以分析集群历史问题。
- 手动性能分析允许仅收集特定实例的特定性能数据，持续性能分析会收集所有实例的所有性能数据。
- 由于持续性能分析存储了更多性能数据，因此会占用更大的磁盘空间。

11.15.2.2 支持的性能数据

所有手动性能分析中的性能数据都会在该功能中收集：

- CPU：TiDB、TiKV、TiFlash、PD 实例上各个内部函数的 CPU 开销情况
- Heap：TiDB、PD 实例上各个内部函数的内存占用开销情况
- Mutex：TiDB、PD 实例上各个处于等待状态的 Mutex 情况
- Goroutine：TiDB、PD 实例上各个Goroutine 的运行状态及调用栈情况

11.15.2.3 访问页面

你可以通过以下任一方式访问持续性能分析页面：

- 登录 TiDB Dashboard 后， 在左侧导航栏中点击高级调试 (Advanced Debugging) > 实例性能分析 (Profiling Instances) > 持续分析(Continuous Profiling)。

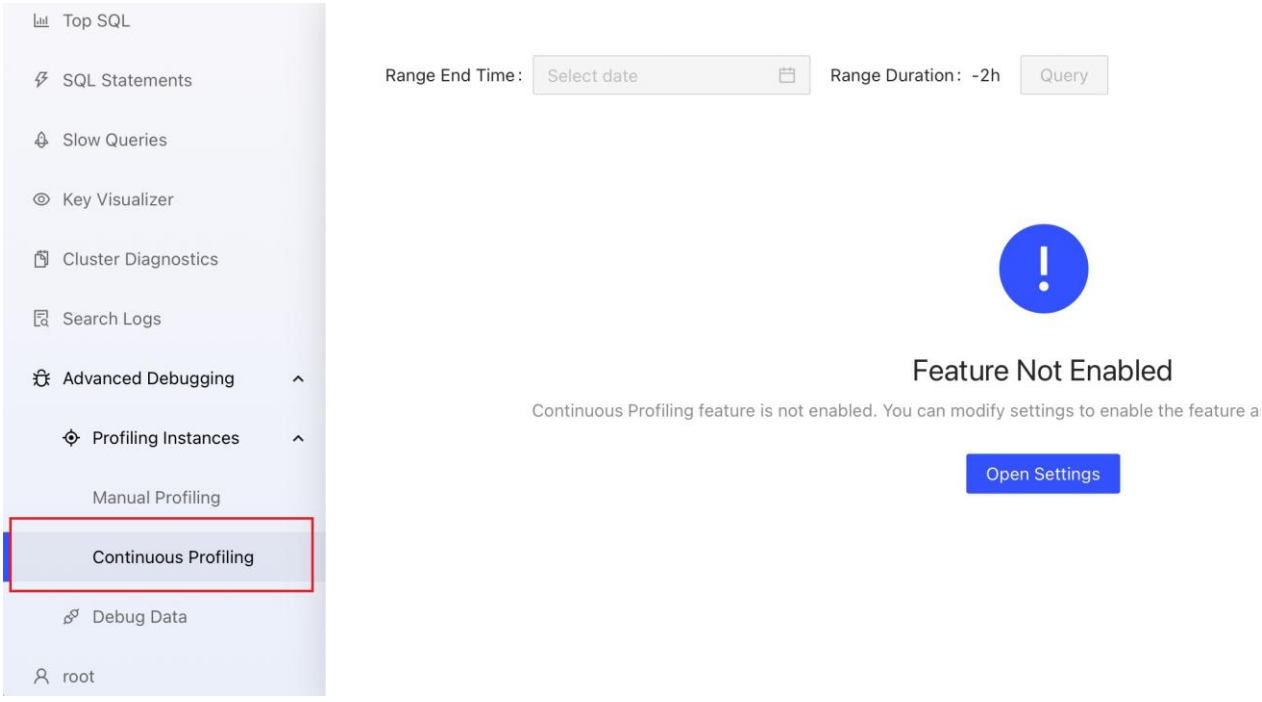


图 356: 访问页面

- 在浏览器中访问http://127.0.0.1:2379/dashboard/#/continuous_profiling（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

11.15.2.4 启用持续性能分析

> **注意：**
>
> 要使用持续性能分析，你需要使用TiUP（v1.9.0 及以上版本）或 TiDB Operator（v1.3.0 及以上版本）部署或升级集群。如果你已经使用旧版本 TiUP 或 TiDB Operator 进行了集群升级，请参见FAQ 进行处理。

该功能启用后，TiDB Dashboard 会在后台持续收集性能数据，用户无需保持网页处于打开状态。后台收集的性能数据可设置保留时长，超过保留时长的性能数据将会被自动清理。

你可以通过以下步骤启用该功能：

1. 访问持续性能分析页面。
2. 点击打开设置(Open Settings)。在右侧设置 (Settings) 页面，将启用特性(Enable Feature) 下方的开关打开。设置保留时间(Retention Period)，默认值为 3 天。
3. 点击保存(Save)。

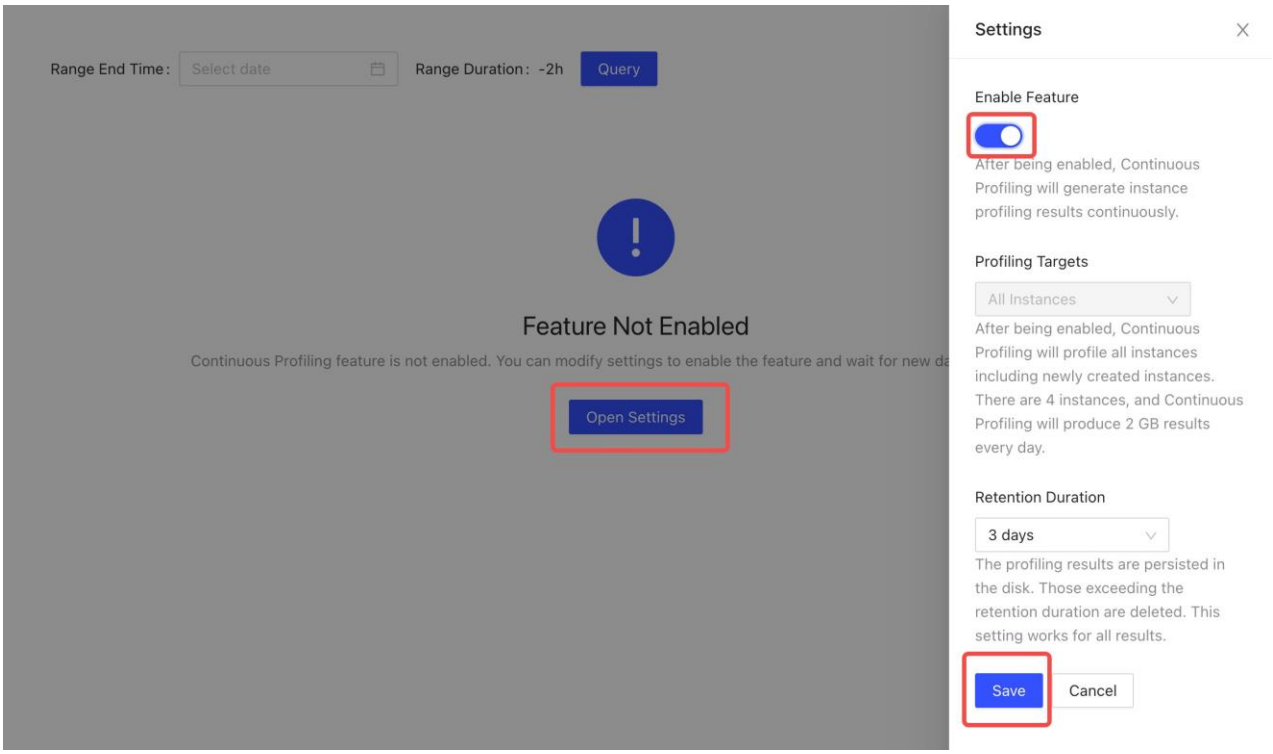


图 357: 启用功能

11.15.2.5 查看集群当前性能数据

在已经启用了持续性能分析的集群上，手动性能分析不再可用。若要查看当前时刻集群性能数据，请点击最近一次分析结果。

11.15.2.6 查看集群历史性能数据

你可以在列表中看到自启用该功能以来所有自动收集的性能数据：

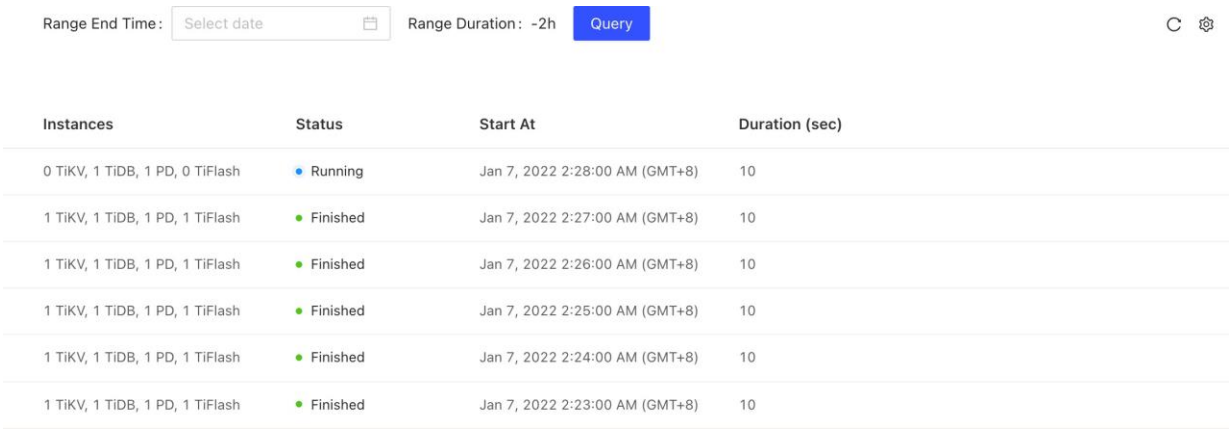


图 358: 历史结果

11.15.2.7 下载性能数据

进入某次分析结果后，可点击右上角下载按钮(Download Profiling Result)打包下载所有性能数据：

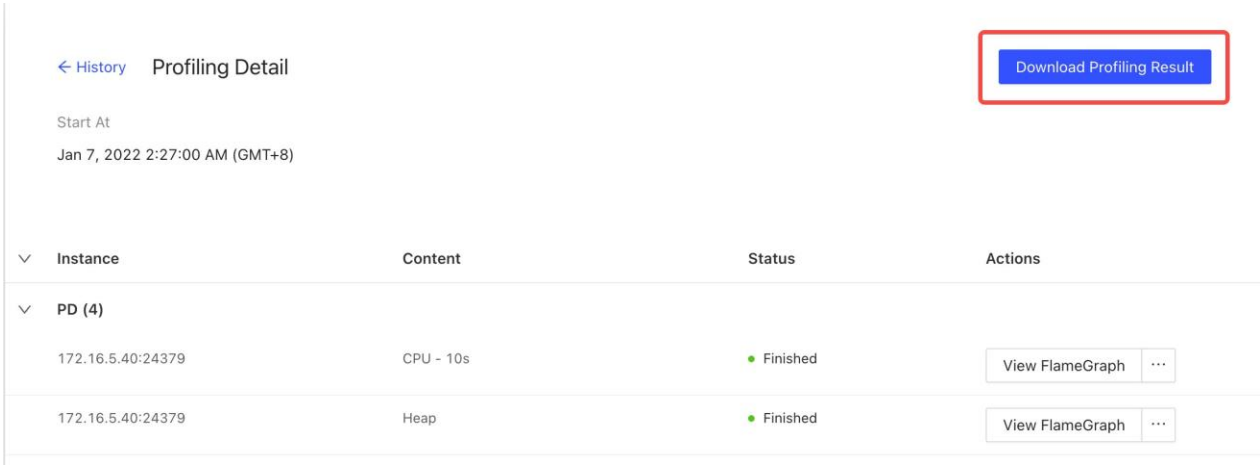


图 359: 下载某次分析结果

也可以点击列表中的单个实例查看其性能数据，或者悬浮在··· 按钮上下载原始数据：

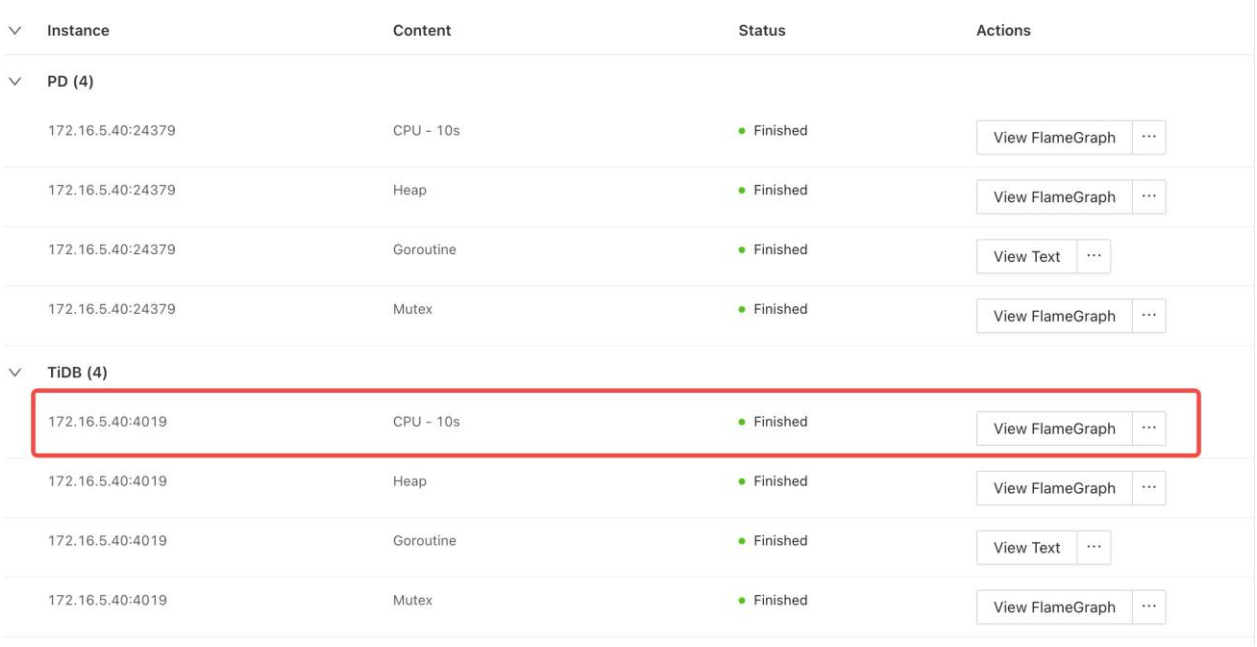


图 360: 查看单个实例分析结果

11.15.2.8 停用持续性能分析

你可以通过以下步骤停用该功能：

1. 访问持续性能分析页面。
2. 点击右上角齿轮按钮打开设置界面，将启用特性(Enable Feature) 下方的开关关闭。
3. 点击保存(Save)。
4. 在弹出的关闭持续分析功能(Disable Continuous Profiling Feature) 对话框中，点击确认(Disable)。

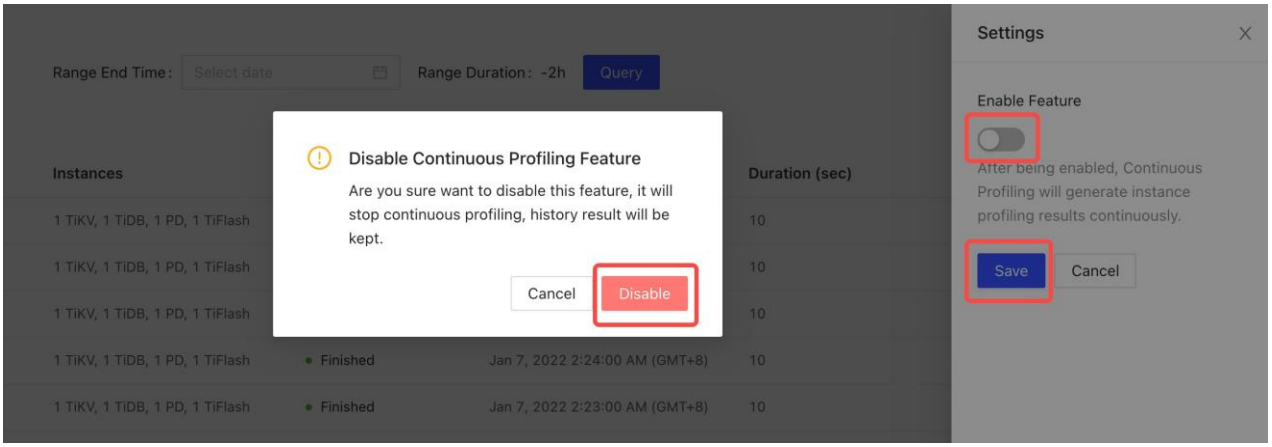


图 361: 停用功能

11.15.2.9 常见问题

1. 界面上提示“集群中未启动必要组件 NgMonitoring”无法启用功能

请参见TiDB Dashboard FAQ。

2. 该功能开启后对集群是否有性能影响？

依据测试结果，该功能开启后对集群的平均性能影响小于 1%。

3. 该功能目前是什么状态？

该功能是正式特性，在生产环境中可用。

11.16 会话管理与配置

11.16.1 分享TiDB Dashboard 会话

你可以将当前的TiDB Dashboard 会话安全地分享给其他用户访问，这样其他用户无需要知道登录账号密码即可访问TiDB Dashboard 并进行操作。

11.16.1.1 分享者操作步骤

1. 登录TiDB Dashboard。
2. 点击边栏左下角的用户名访问配置界面。
3. 点击分享当前会话(Share Current Session)。

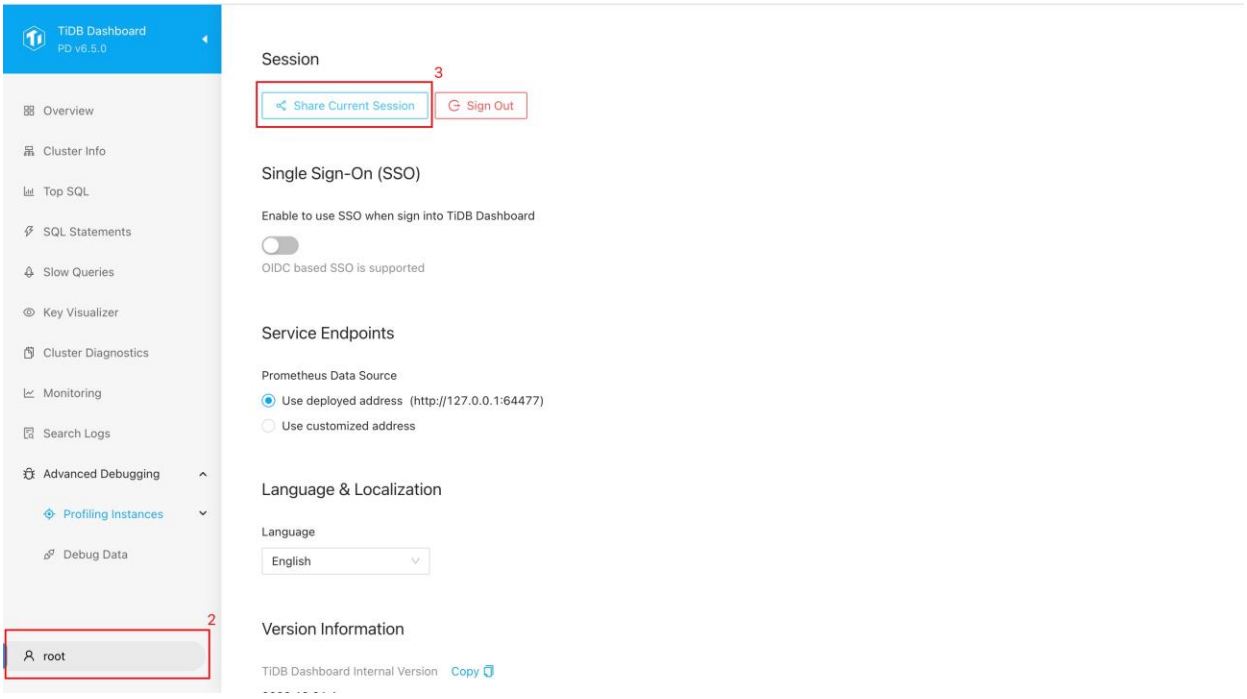


图 362: 操作示例

注意：
出于安全考虑，已分享的会话中不能使用分享功能将该会话再次分享给其他人。

4. 在弹出的对话框中，对分享进行细节配置：

- 有效时间：分享的会话在多少时间内有效。登出当前会话不影响已分享会话的有效时间。
- 以只读权限分享：分享的会话为只读，例如不允许进行配置修改等操作。

5. 点击生成授权码(Generate Authorization Code)。

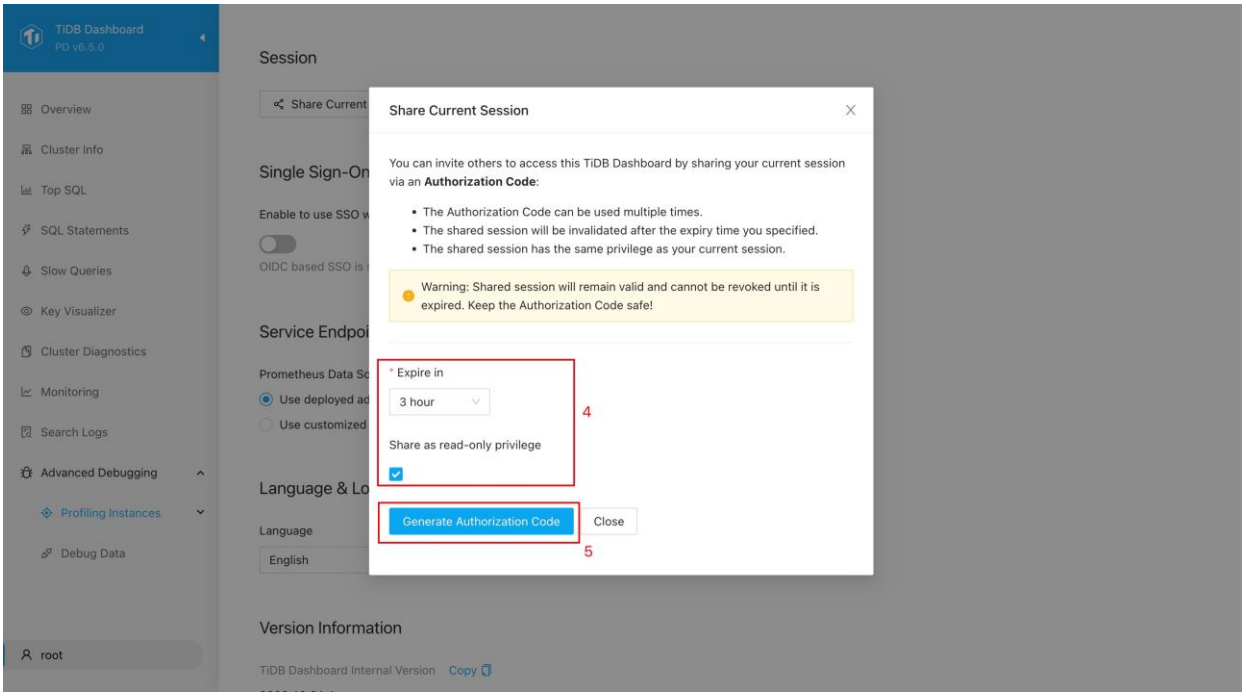


图 363: 操作示例

6. 将生成出来的授权码提供给要分享的用户。

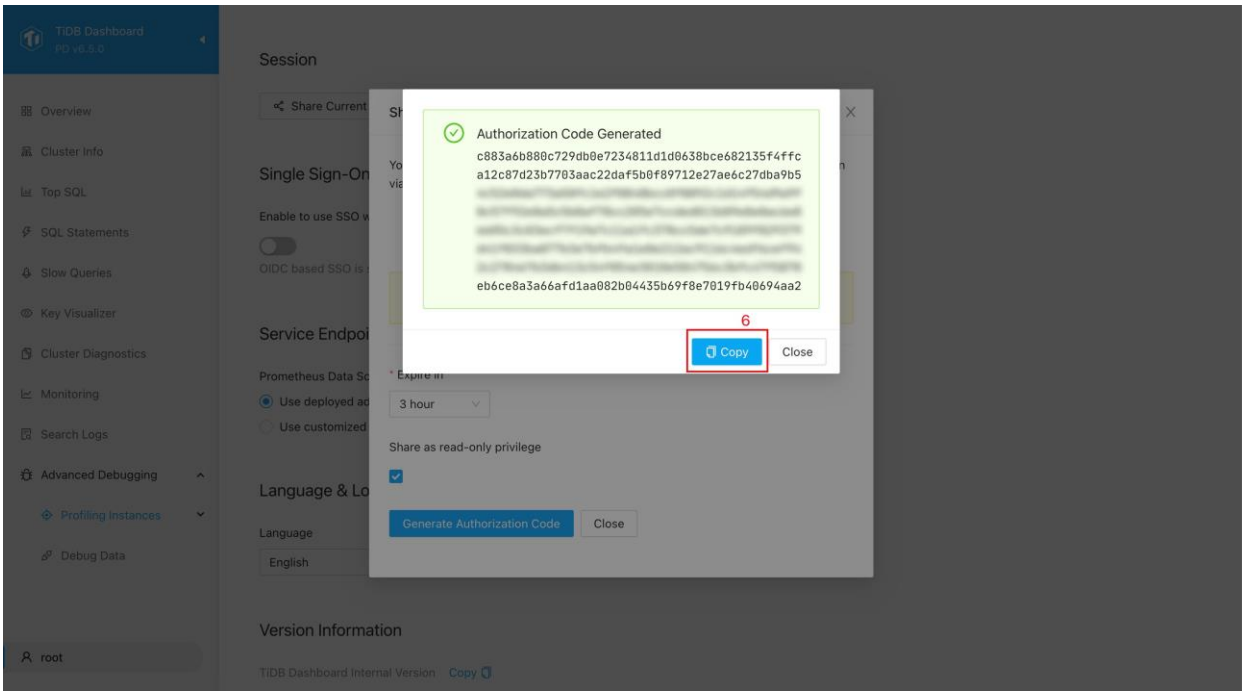


图 364: 操作示例

警告：

请妥善地保管授权码。不要将授权码分发给不受信任的用户，否则他们也将具备访问和操作 TiDB Dashboard 的能力。

11.16.1.2 受邀请者操作步骤

1. 在 TiDB Dashboard 登录界面上，点击使用其他登录方式(Use Alternative Authentication)。

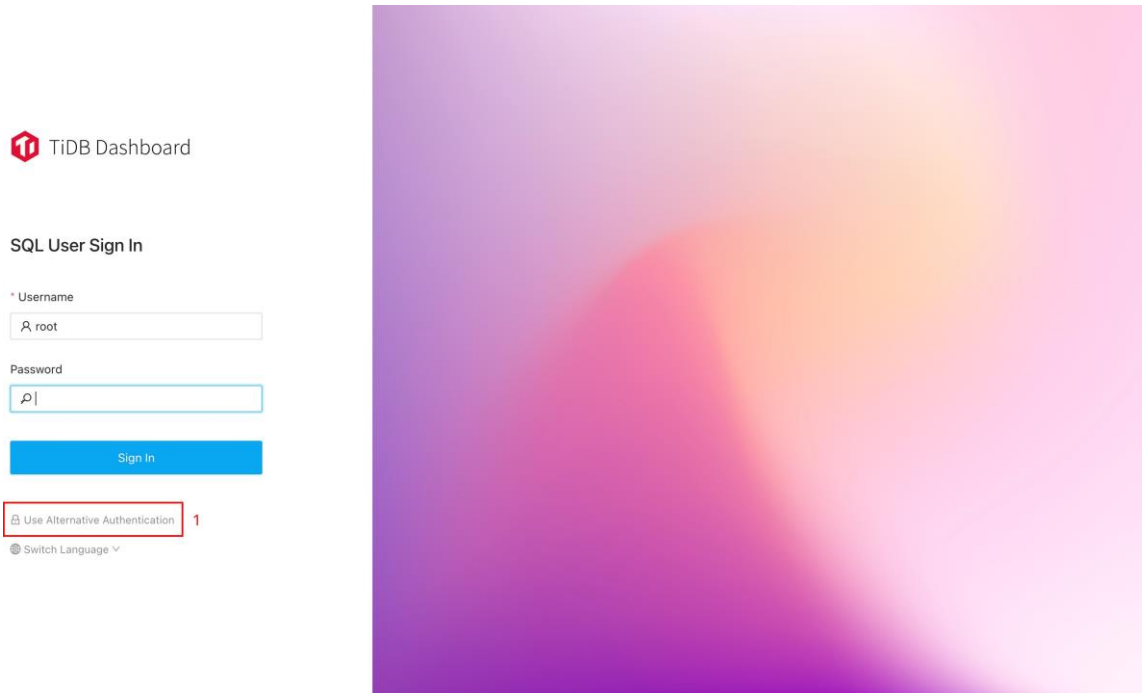


图 365: 操作示例

2. 选择使用授权码(Authorization Code) 登录。



图 366: 操作示例

3. 输入从分享者取得的授权码。
4. 点击登录(Sign In)。

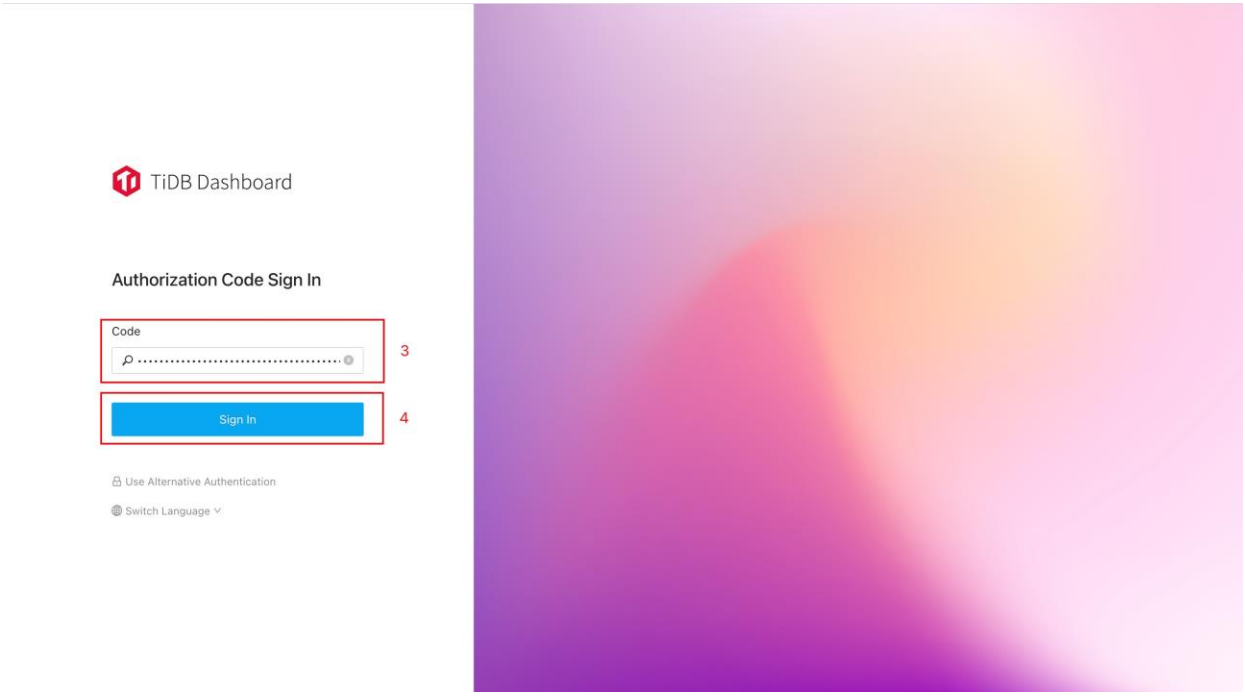


图 367: 操作示例

11.16.2 配置 TiDB Dashboard 使用 SSO 登录

TiDB Dashboard 支持基于 OIDC 协议的单点登录(Single Sign-On)。配置 TiDB Dashboard 启用 SSO 登录后，你可以通过配置的 SSO 服务进行登录鉴权，无需输入 SQL 用户名和密码即可登录到TiDB Dashboard。

11.16.2.1 配置 OIDC SSO

11.16.2.1.1 启用 SSO

1. 登录TiDB Dashboard。
2. 点击边栏左下角的用户名访问配置界面。
3. 在单点登录(Single Sign-On) 区域下，开启允许使用 SSO 登录到TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 在表单中填写 OIDC Client ID 和 OIDC Discovery URL 字段。
   一般可以从 SSO 服务的提供商处获取到这两个字段信息：
   - OIDC Client ID 有时也被称为 OIDC Token Issuer

- OIDC Discovery URL 有时也被称为 OIDC Token Audience。

5. 将 SQL 登录密码录入到 TiDB Dashboard 中，以便在 SSO 鉴权通过后完成登录。点击授权登录为该用户 (Authorize Impersonation) 录入密码。

   这是因为TiDB Dashboard SSO 的原理是在 SSO 成功鉴权后，采用 TiDB Dashboard 内加密存储的 SQL 登录密码进行替代登录。

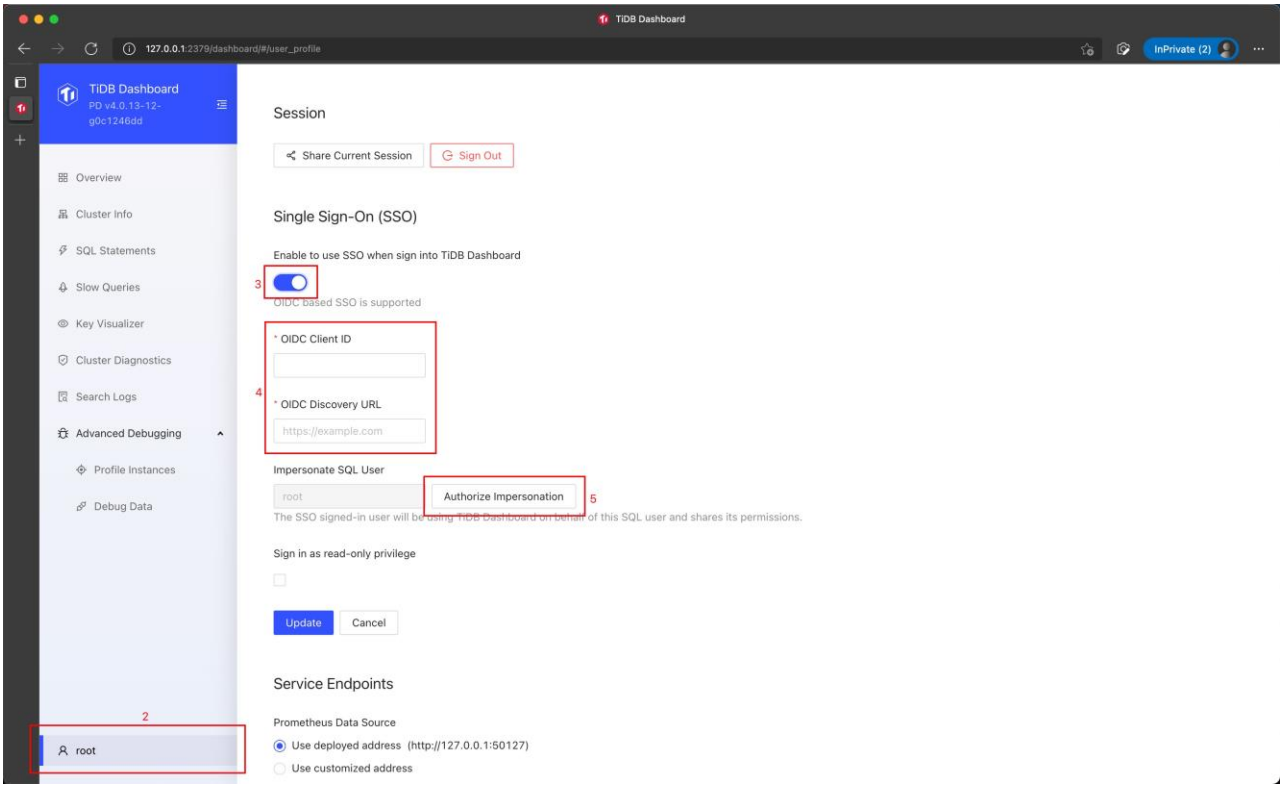


图 368: 操作示例

> **注意：**
> 你录入的密码将被加密存储。若 SQL 用户密码后续发生了变更，将导致 SSO 登录失败。这时可以重新录入密码使 SSO 登录恢复正常。

6. 在对话框中填写完密码后，点击授权并保存(Authorize and Save)。

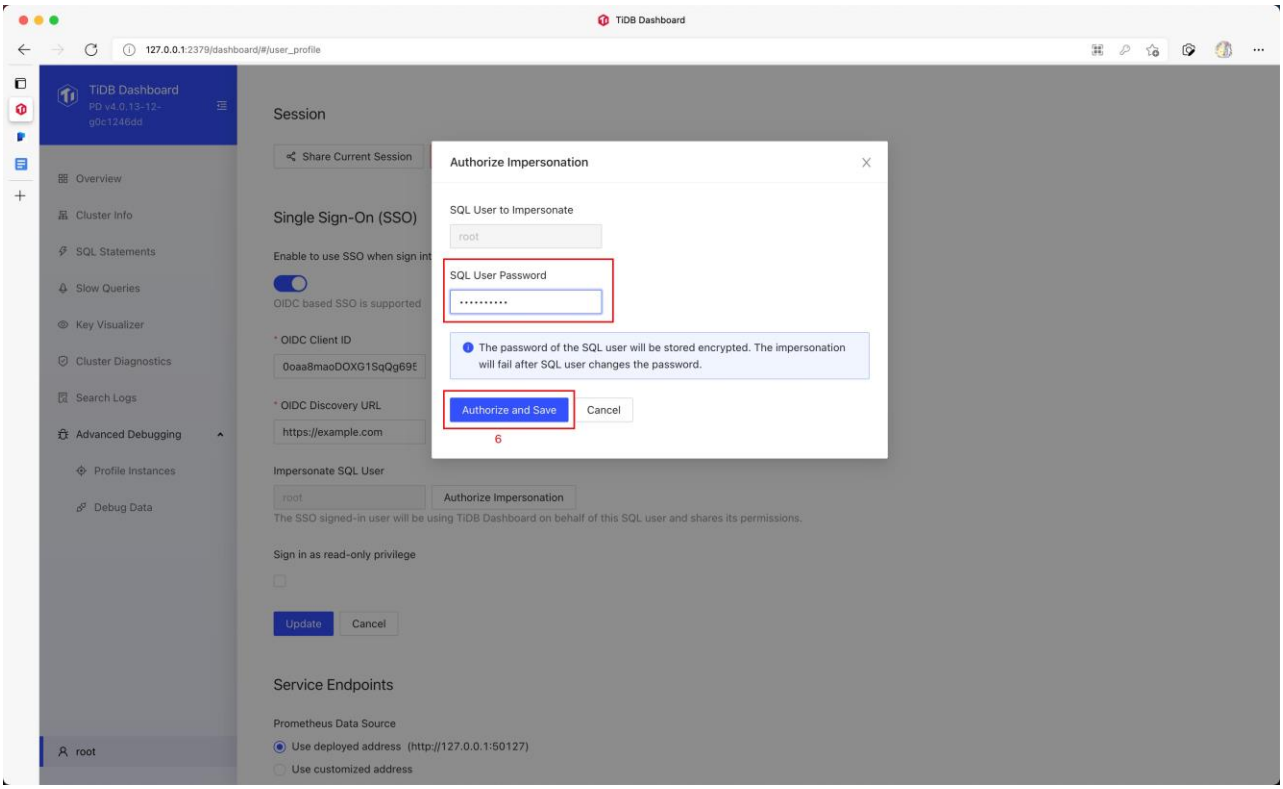


图 369: 操作示例

7. 点击更新(Update)保存配置。

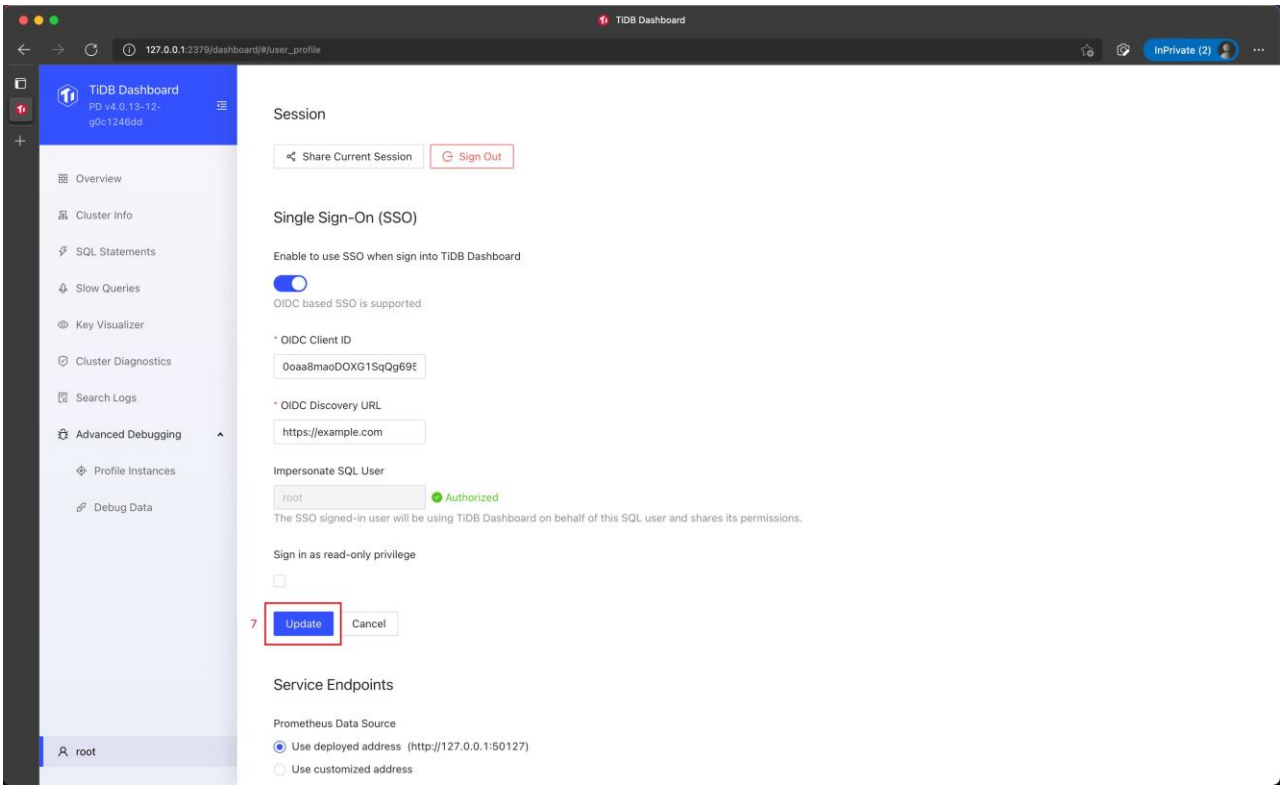


图 370: 操作示例

至此TiDB Dashboard 中已经成功开启了 SSO 登录。

> **注意：**
>
> 出于安全原因，部分 SSO 服务还需要你进一步在 SSO 服务中配置受信任的登录和登出跳转地址，请参见 SSO 服务的具体帮助完成配置。

11.16.2.1.2 禁用 SSO

你可以随时禁用 SSO。禁用后，之前已录入并存储在本地的替代登录 SQL 密码将被彻底清除。禁用步骤如下：

1. 登录TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录(Single Sign-On) 区域下，关闭允许使用 SSO 登录到TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 点击更新(Update) 保存配置。

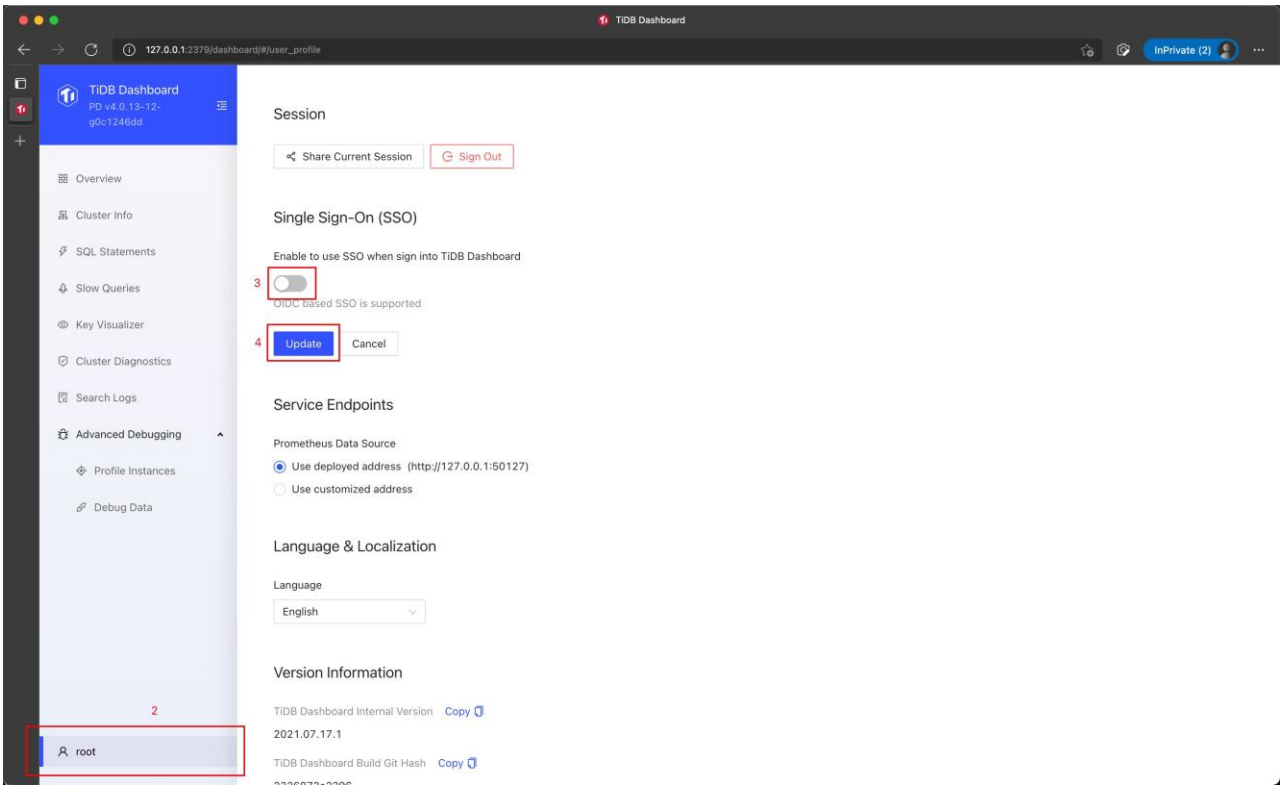


图 371: 操作示例

11.16.2.1.3 密码发生变更后重新录入密码

若替代登录的 SQL 用户密码发生了变更，则 SSO 登录将会失败。此时，你可以将新的登录密码录入到TiDB Dashboard 中以恢复正常 SSO 登录功能，步骤如下：

1. 登录TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录(Single Sign-On) 区域下，点击授权登录为该用户(Authorize Impersonation) 来录入新的密码。

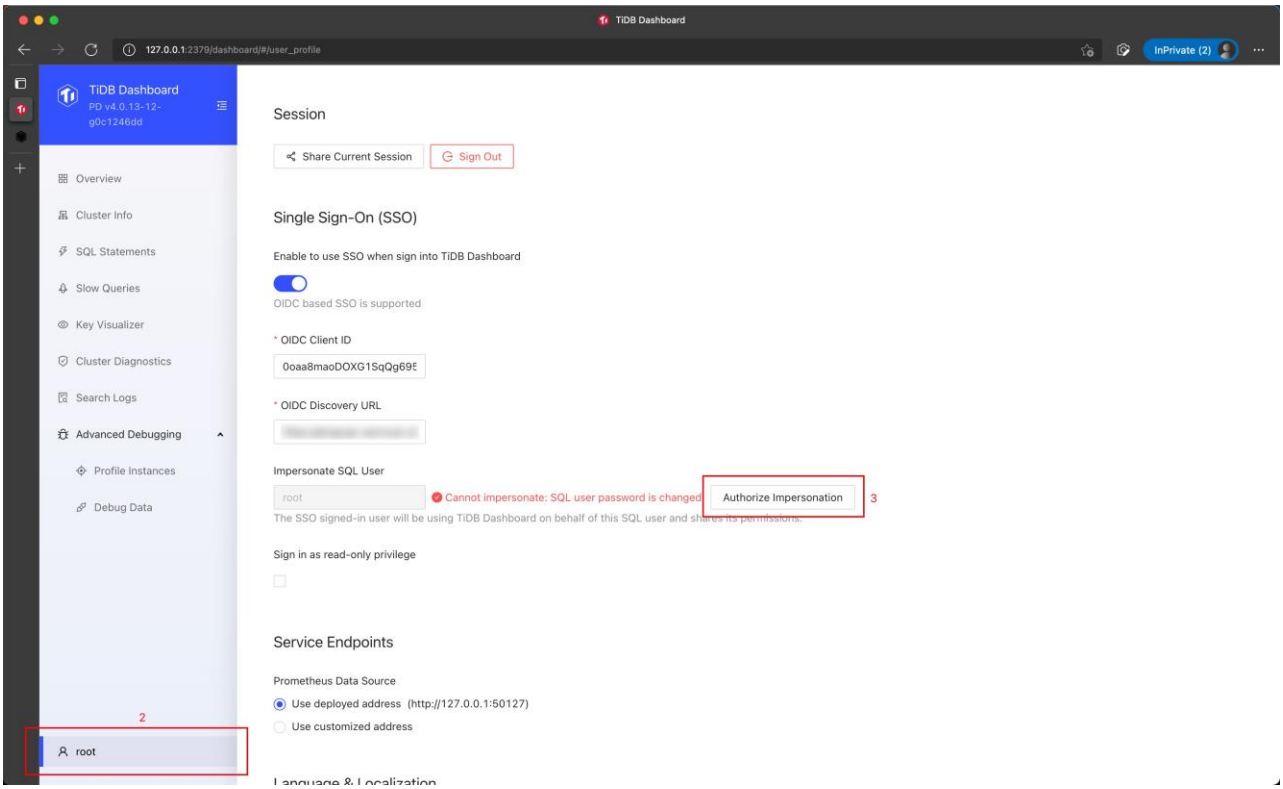


图 372: 操作示例

4. 在对话框中填写完毕密码后，点击授权并保存(Authorize and Save)。

11.16.2.2 使用 SSO 登录

若TiDB Dashboard 已经完成了 SSO 的配置，你可使用以下步骤完成登录：

1. 在 TiDB Dashboard 登录界面上，点击使用公司账号 SSO 登录(Sign in via Company Account)。

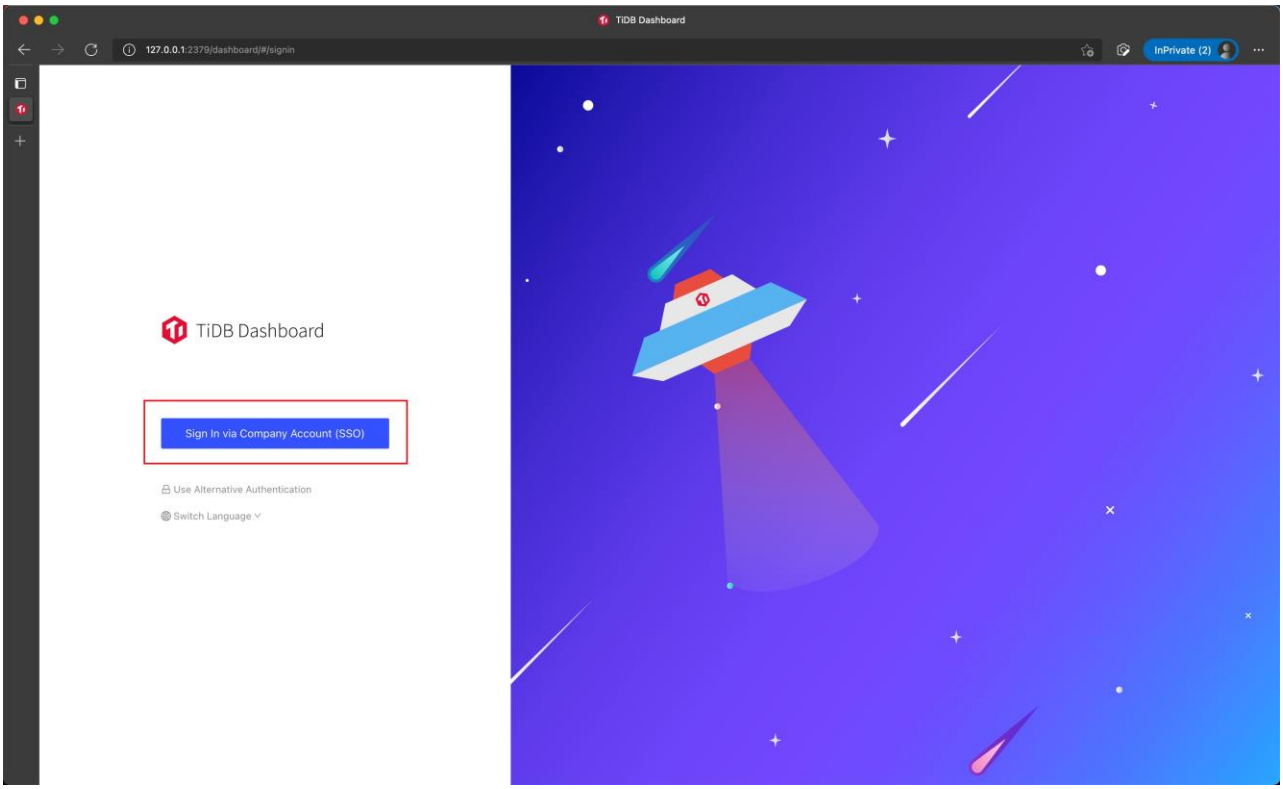


图 373: 操作示例

2. 在配置 SSO 的系统中进行登录。
3. 你将被重定向回TiDB Dashboard 完成登录。

11.16.2.3 示例一：使用 Okta 进行TiDB Dashboard SSO 登录认证

Okta 是一个提供OIDC SSO 的身份认证服务。以下步骤展示了如何配置Okta 及TiDB Dashboard，使得TiDB Dashboard 可以通过 Okta 进行 SSO 登录。

11.16.2.3.1 步骤一：配置 Okta

首先需要在 Okta 中创建一个用于集成 SSO 的 Application Integration。

1. 访问 Okta 管理后台。
2. 点击左侧边栏的Applications > Applications。
3. 点击 Create App Integration。

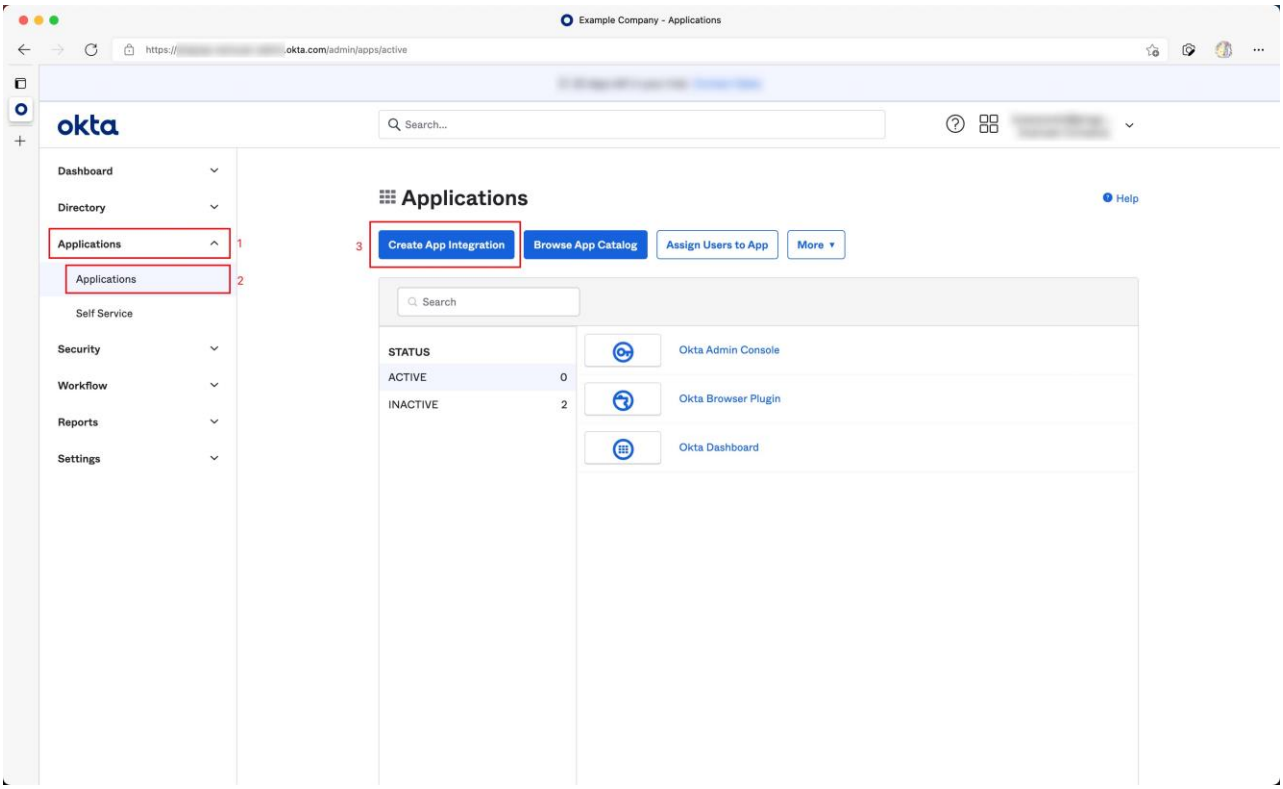


图 374: 操作示例

4. 在弹出的对话框中，Sign-in method 字段选择 OIDC - OpenID Connect。
5. Application Type 字段选择 Single-Page Application。
6. 对话框中点击 Next 按钮。

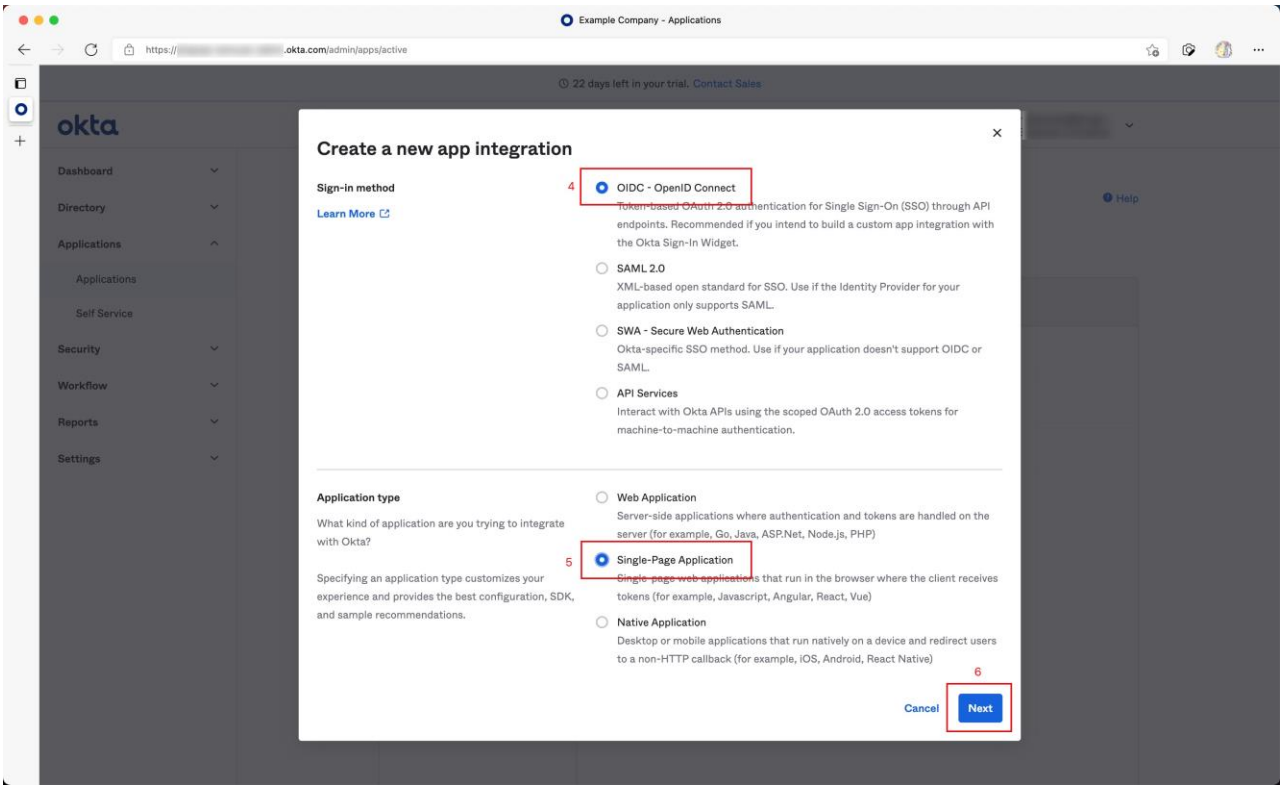


图 375: 操作示例

7. Sign-in redirect URIs 字段填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问TiDB Dashboard 所使用的域名（或 IP）及端口。

8. Sign-out redirect URIs 字段填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/
```

类似地，将 DASHBOARD_IP:PORT 替换为实际的域名（或IP）及端口。

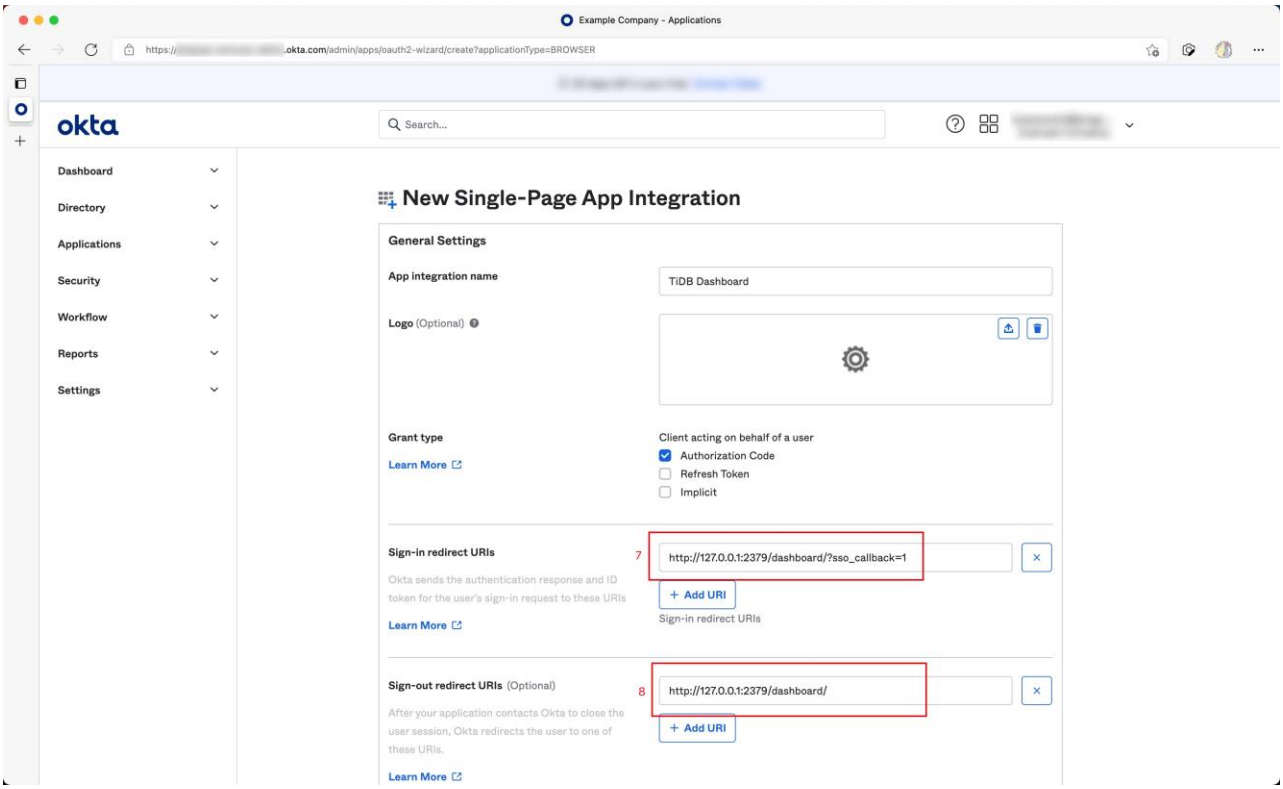


图 376: 操作示例

9. 在 Assignments 中按你的实际需求配置组织中哪些用户可以通过这个 SSO 登录 TiDB Dashboard，然后点击 Save 保存配置。

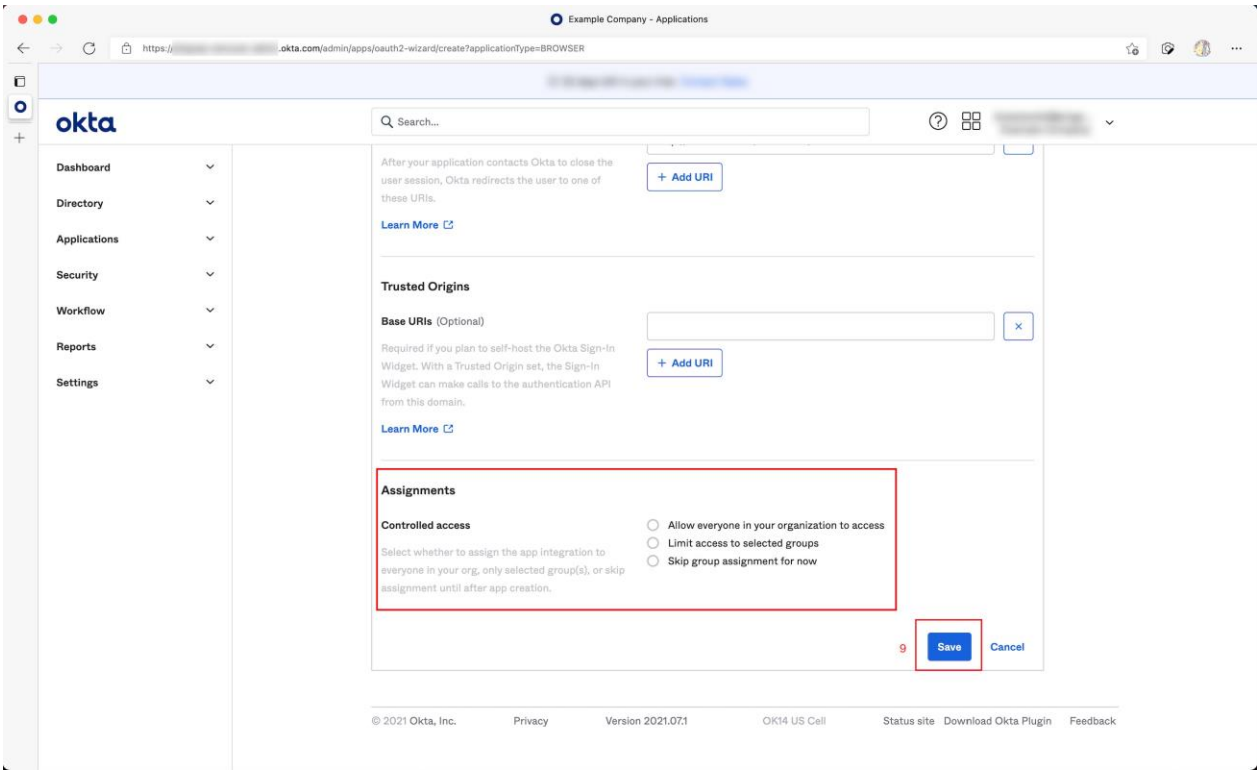


图 377: 操作示例

11.16.2.3.2 步骤二：获取TiDB Dashboard 所需的配置参数并填入TiDB Dashboard

1. 在 Okta 创建的App Integration 中，点击 Sign On。

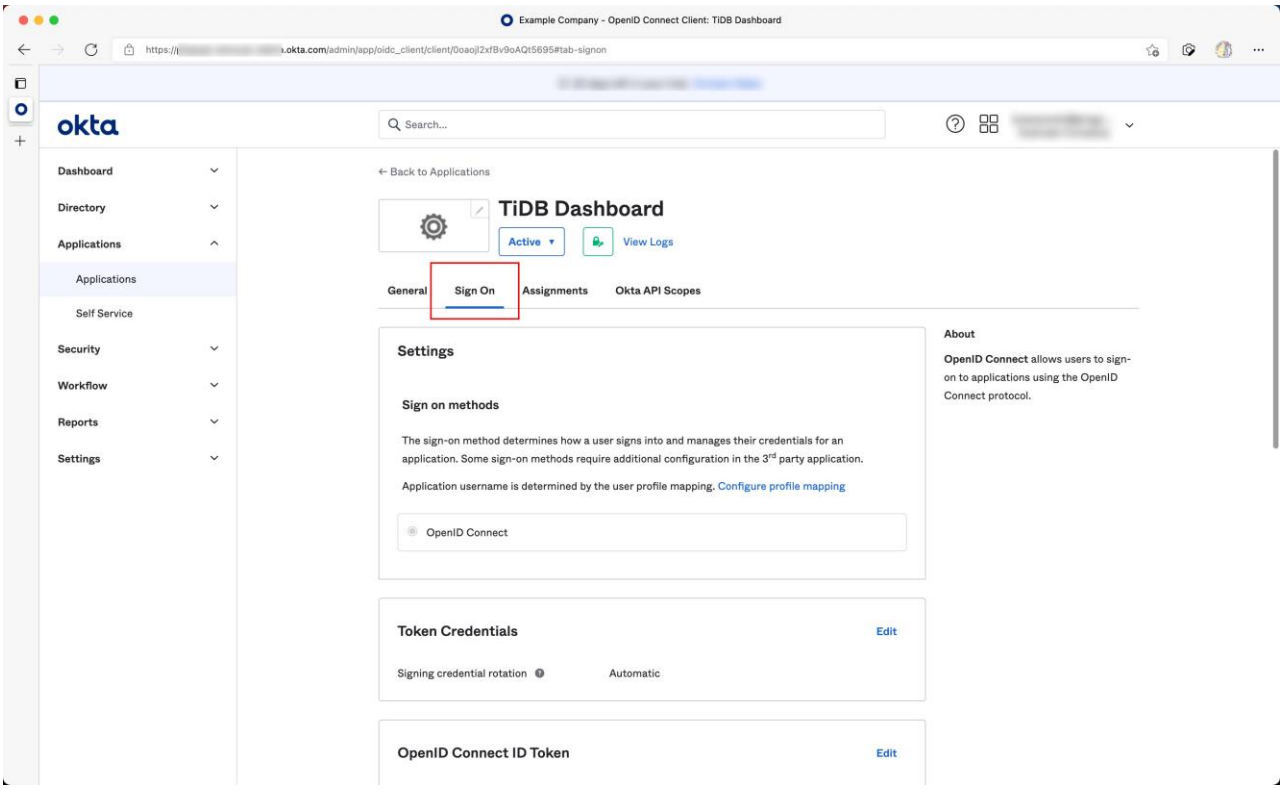


图 378: 操作示例 1

2. OpenID Connect ID Token 区域中有Issuer 和 Audience 字段，复制这两个字段的值。

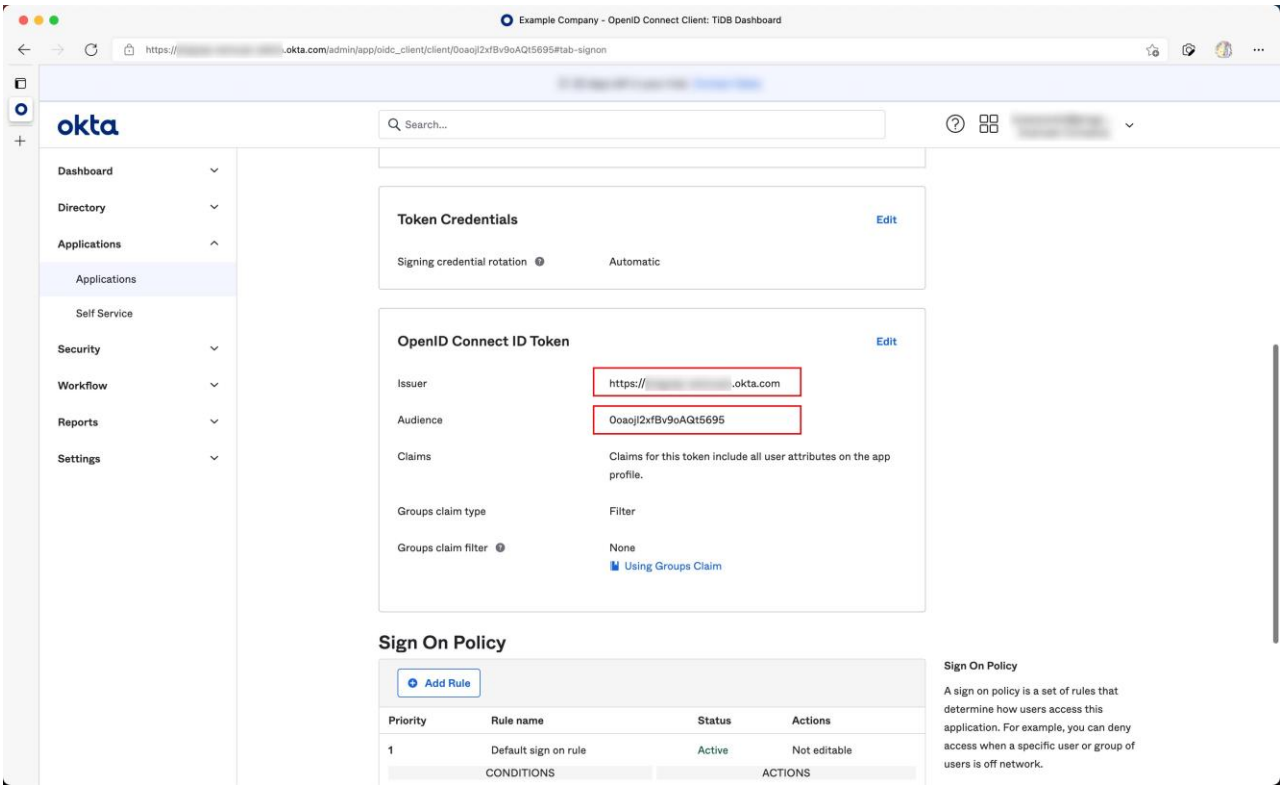


图 379: 操作示例 2

3. 打开TiDB Dashboard 配置界面，将上一步获取到的 Issuer 填入 OIDC Client ID，将 Audience 填入 OIDC Discovery URL 后，完成授权并保存配置。示例如下：

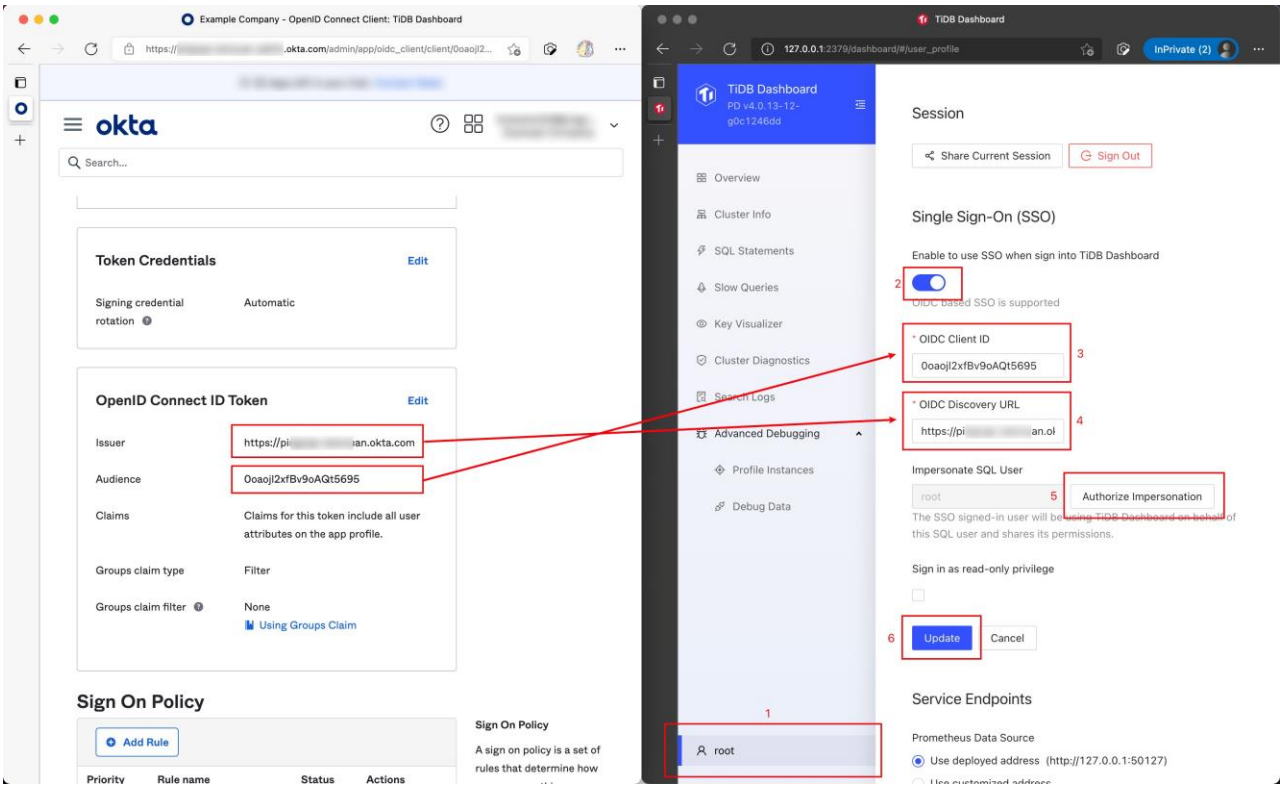


图 380: 操作示例 3

至此，TiDB Dashboard 已被配置为使用 Okta 进行 SSO 登录。

11.16.2.4 示例二：使用Auth0 进行TiDB Dashboard SSO 登录认证

和 Okta 类似，Auth0 也可以提供 OIDC SSO 的身份认证服务。

11.16.2.4.1 步骤一：配置Auth0

1. 访问Auth0 的管理后台。
2. 点击左侧边栏的Applications > Applications。
3. 点击 Create Application，在弹出窗口中输入 Name，例如 “TiDB Dashboard”。在 Choose an application type 下选择 “Single Page Web Application”。

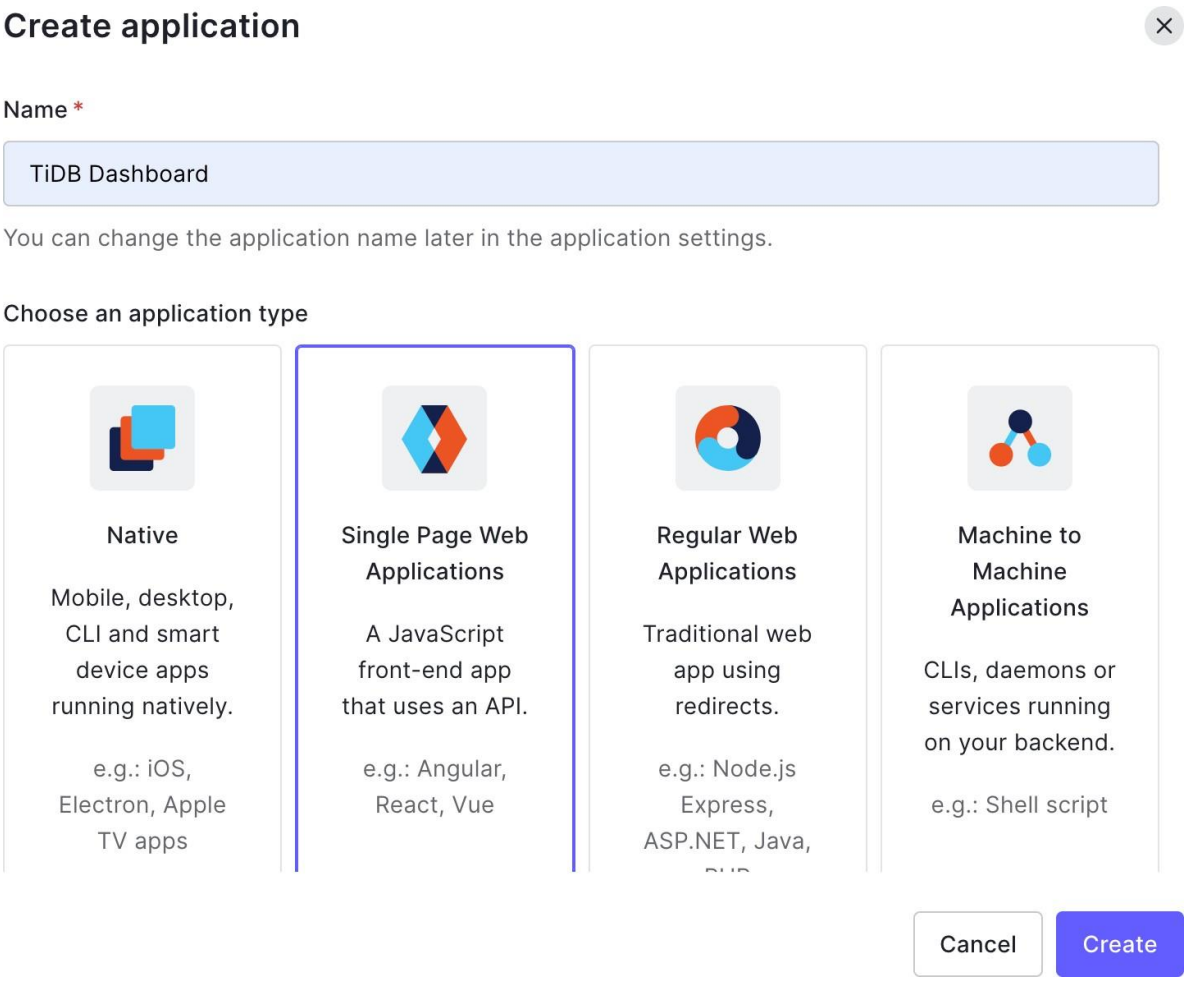


图 381: Create Application

4. 点击 Settings 栏。

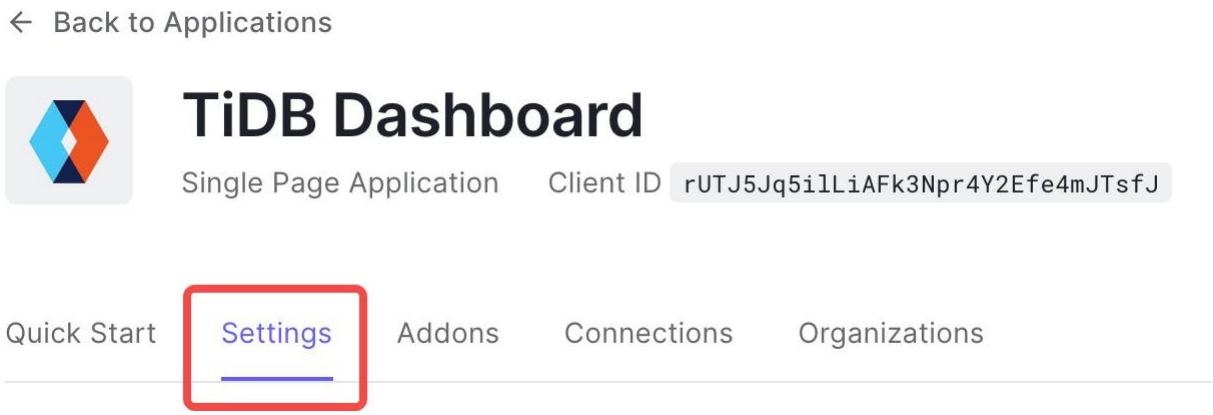


图 382: Settings

5. 在 Allowed Callback URLs 字段中填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

在以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问TiDB Dashboard 所使用的域名（或 IP）及端口。

6. 在 Allowed Logout URLs 字段中填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/
```

类似地，将 DASHBOARD_IP:PORT 替换为实际的域名（或IP）及端口。

Allowed Callback URLs

http://localhost:3001/dashboard/?sso_callback=1,
http://127.0.0.1:2379/dashboard/?sso_callback

After the user authenticates we will only call back to any of these URLs. You can specify multiple valid URLs by comma-separating them (typically to handle different environments like QA or testing). Make sure to specify the protocol (`https://`) otherwise the callback may fail in some cases. With the exception of custom URI schemes for native clients, all callbacks should use protocol `https://` . You can use Organization URL parameters in these URLs.

Allowed Logout URLs

http://localhost:3001/dashboard/,
http://127.0.0.1:2379/dashboard/

A set of URLs that are valid to redirect to after logout from Auth0. After a user logs out from Auth0 you can redirect them with the `returnTo` query parameter. The URL that you use in `returnTo` must be listed here. You can specify multiple valid URLs by comma-separating them. You can use the star symbol as a wildcard for subdomains (`*.google.com`). Query strings and hash information are not taken into account when validating these URLs. Read more about this at https://auth0.com/docs/login/logout

图 383: Settings

7. 其它设置保持默认，点击 Save Changes 保存。

11.16.2.4.2 步骤二：获取TiDB Dashboard 所需的配置参数并填入TiDB Dashboard

1. 将 Auth0 Settings 栏 Basic Information 项的 Client ID 字段的值填入TiDB Dashboard 的 OIDC Client ID，将 Domain 字段的值，加上`https://` 前缀和 `/` 后缀后填入OIDC Discovery URL 中，例如`https://example.us.auth0.com/`。完成授权并保存配置即可。

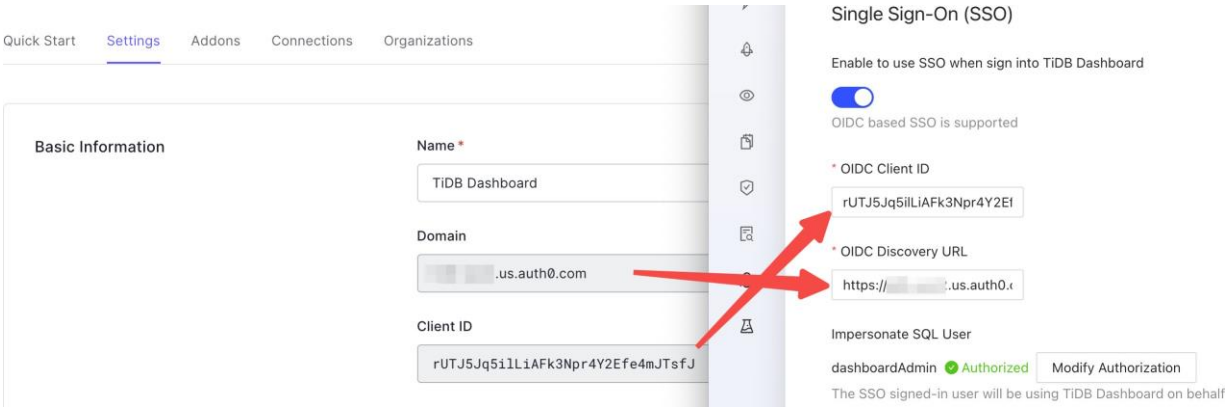


图 384: Settings

至此，TiDB Dashboard 已被配置为使用Auth0 进行 SSO 登录。

11.16.2.5 示例三：使用 Casdoor 进行TiDB Dashboard SSO 登录认证

Casdoor 是一个开源的、可以部署在私人服务器上的 SSO 平台。它与TiDB Dashboard 的 SSO 功能兼容。以下步骤展示了如何配置 Casdoor 及 TiDB Dashboard，使得 TiDB Dashboard 可以通过 Casdoor 进行 SSO 登录。

11.16.2.5.1 步骤一：配置 Casdoor

1. 部署并访问 Casdoor 的管理后台。
2. 点击上方菜单栏的应用。
3. 点击添加按钮。

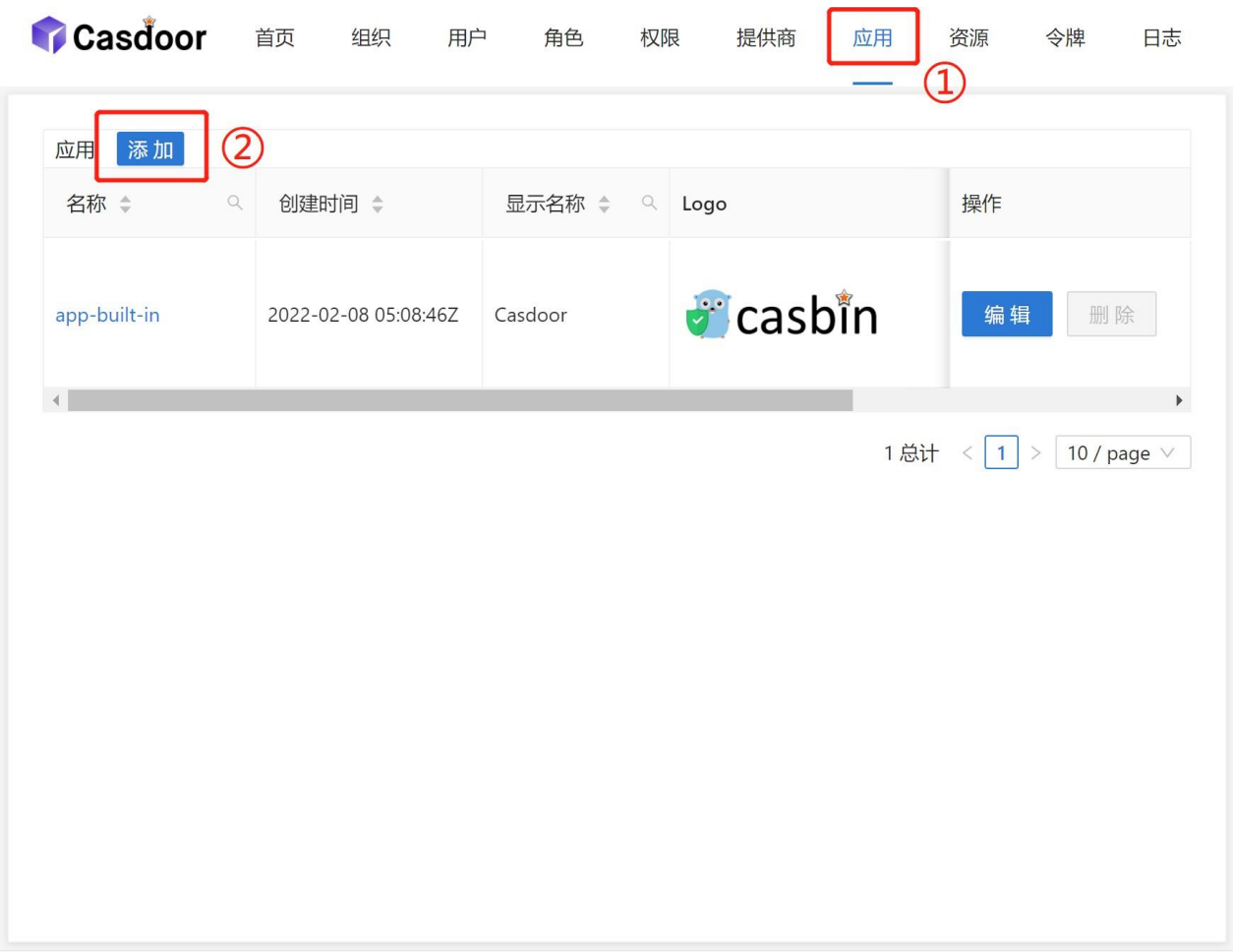


图 385: Settings

4. 填写名称和显示名称，比如：TiDB Dashboard。
5. 在回调 URLs 中添加如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

在以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问TiDB Dashboard 所使用的域名（或 IP）及端口。

回调URLs ⓘ :

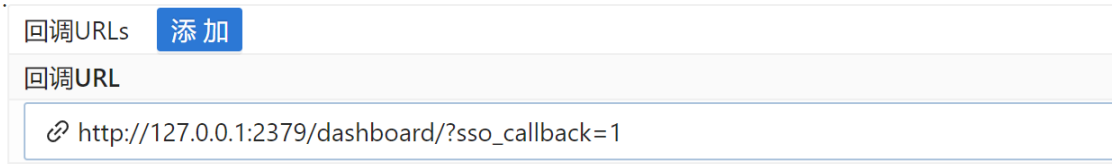


图 386: Settings

6. 其他设置保留默认值，点击保存& 退出。
7. 保存页面上的客户端ID。

11.16.2.5.2 步骤二：获取TiDB Dashboard 所需的配置参数并填入TiDB Dashboard

1. 将上一步获得的客户端ID 字段的值填入TiDB Dashboard 的 OIDC Client ID。
2. 将 Casdoor 部署地址加上 `https://` 前缀和 `/` 后缀后填入 OIDC Discovery URL 中，例如 `https://casdoor.` `example.com/`。完成授权并保存配置即可。

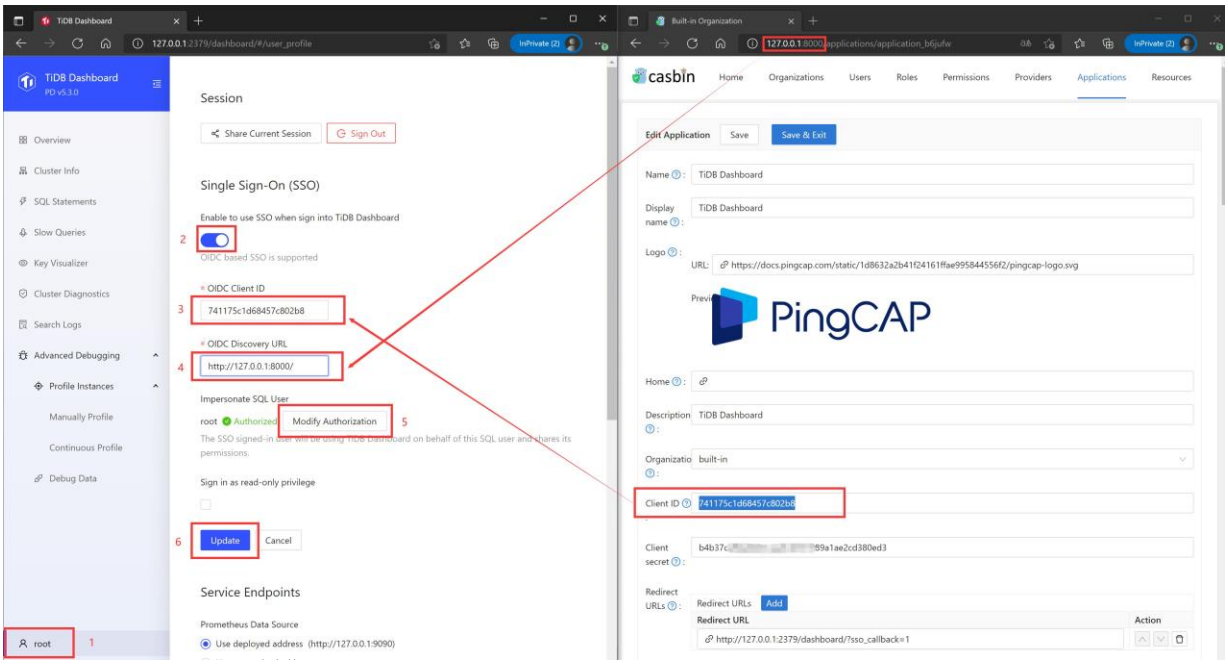


图 387: Settings

至此，TiDB Dashboard 已被配置为使用 Casdoor 进行 SSO 登录。

11.17 TiDB Dashboard 常见问题

本文汇总了使用TiDB Dashboard 过程中的常见问题与解决办法。若无法找到对应问题，或者根据指引操作后问题仍然存在，请从 PingCAP 官方或TiDB 社区获取支持。

11.17.1 访问

11.17.1.1 已配置防火墙或反向代理，但访问后被跳转到一个内部地址无法访问TiDB Dashboard

集群部署有多个 PD 实例的情况下，只有其中某一个 PD 实例会真正运行TiDB Dashboard 服务，访问其他 PD 实例时会发生浏览器端重定向。若防火墙或反向代理没有为此进行正确配置，就可能出现访问后被重定向到一个被防火墙或反向代理保护的内部地址的情况。

- 参阅TiDB Dashboard 多 PD 实例部署章节了解多 PD 实例下TiDB Dashboard 的工作原理。
- 参阅通过反向代理使用TiDB Dashboard 章节了解如何正确配置反向代理。
- 参阅提高TiDB Dashboard 安全性章节了解如何正确配置防火墙。

11.17.1.2 双网卡部署时无法通过另一个网卡访问TiDB Dashboard

PD 中的TiDB Dashboard 出于安全考虑仅监听部署时所指定的IP 地址（即只监听在一个网卡上），而非0.0.0.0，因此当主机上安装了多个网卡时，通过另一个网卡将无法访问。

当你使用 `tiup cluster` 或 `tiup playground` 命令部署时，目前尚没有方法改变该行为。推荐使用反向代理将 TiDB Dashboard 安全地暴露给另一个网卡，具体参见通过反向代理使用TiDB Dashboard 章节。

11.17.2 界面功能

11.17.2.1 概况页面中 QPS 及 Latency 显示 prometheus_not_found 错误

QPS 及 Latency 监控依赖于集群中已正常部署 Prometheus 监控实例，没有部署的情况下就会显示为错误。向集群中新部署 Prometheus 实例即可解决该问题。

若已经部署 Prometheus 监控实例但仍然显示为错误，可能的原因是您使用的部署工具（TiUP 或 TiDB Operator）版本比较旧，没有自动汇报监控地址，导致TiDB Dashboard 无法感知并查询监控数据。可以升级到最新的部署工具并重试。

以下给出TiUP 部署工具的操作方法，对于其他部署工具，请参阅工具对应文档。

1. 升级TiUP、TiUP Cluster：

```
bash  tiup update --self tiup update cluster --force
```

2. 升级后，部署包含监控节点的新集群时，应当能正常显示监控。
3. 升级后，对于现有集群，可通过再次启动集群的方法汇报监控地址（将 CLUSTER_NAME 替换为实际集群名称）：

```
bash  tiup cluster start CLUSTER_NAME
```

即使集群已经启动，请仍然执行该命令。该命令不会影响集群上正常的业务，但会刷新并上报监控地址，从而能让监控在TiDB Dashboard 中正常显示。

11.17.2.2 慢查询页面显示 invalid connection 错误

可能的原因是你开启了TiDB 的 prepared-plan-cache 功能。prepared-plan-cache 是实验性功能，在某些版本的TiDB 中可能无法正常运行，开启后可能会导致TiDB Dashboard（及其他应用）出现该问题。你可以通过系统变量tidb_enable_prepared_plan_cache 关闭这项功能。

11.17.2.3 界面提示集群中未启动必要组件 NgMonitoring

NgMonitoring 是 TiDB v5.4.0 及以上集群中内置的高级监控组件，用于支撑TiDB Dashboard 的持续性能分析和Top SQL 等功能。使用较新版本TiUP 部署或升级集群时，NgMonitoring 会自动部署；使用TiDB Operator 部署集群时，需要依据启用持续性能分析手动部署 NgMonitoring。

如果界面提示集群中未启动必要组件 NgMonitoring，可按以下方式排查部署问题。

使用TiUP 部署的集群

第 1 步：检查TiUP Cluster 版本

1. 检查TiUP Cluster 版本，NgMonitoring 组件需要较高版本的部署工具支持（TiUP v1.9.0 及以上）：

```shell
tiup cluster --version
```

上述命令可查看 TiUP Cluster 的具体版本。例如：

```
tiup version 1.9.0 tiup
Go Version: go1.17.2
Git Ref: v1.9.0
```

2. 如果TiUP 版本低于 v1.9.0，升级 TiUP 和 TiUP Cluster 版本至最新。

```shell
tiup update --all
```

第 2 步：在中控机上，通过TiUP 添加ng_port 配置项，然后重启 Prometheus 节点。

1. 以编辑模式打开集群的配置文件：

```shell
tiup cluster edit-config ${cluster-name}
```

2. 在 monitoring_servers 下面增加 ng_port:12020 参数：

```
```
monitoring_servers:
- host: 172.16.6.6
  ng_port: 12020
```
```

3. 重启 Prometheus 节点：

```
```shell
tiup cluster reload ${cluster-name} --role prometheus
```
```

如果执行完上述步骤后依然提示 NgMonitoring 未启动，请从 PingCAP 官方或TiDB 社区获取支持。

使用TiDB Operator 部署的集群

请参见TiDB Operator 文档中启用持续性能分析的步骤部署 NgMonitoring 组件。

使用TiUP Playground 启动的集群

TiUP Playground (>= v1.8.0) 在启动集群时会自动启动 NgMonitoring 组件。可使用以下命令更新TiUP Playground 到最新版：

```
tiup update --self
tiup update playground
```

11.17.2.4 慢查询页面显示 unknown field 错误

集群升级后，如果慢查询页面出现 unknown field 错误，是由于升级后新版本TiDB Dashboard 字段与浏览器缓存内的用户偏好设置的字段不兼容导致的。该问题已修复。如果你的集群版本低于 v5.0.3 或 v4.0.14，需要执行以下步骤清理浏览器缓存：

1. 打开TiDB Dashboard 页面。
2. 打开浏览器的开发者工具。各浏览器的打开方式不同。
   - Firefox：菜单 > Web 开发者 > 切换工具箱（译者注：此处修改为最新的 Firefox Quantum），或者工具栏 > 切换工具箱。
   - Chrome：菜单 > 更多工具 > 开发者工具。
   - Safari：Develop > Show Web Inspector。如果你看不到 Develop 菜单，选择 Preferences > Advanced，然后点击 Show Develop menu in menu bar 复选框。

   以 Chrome 为例：

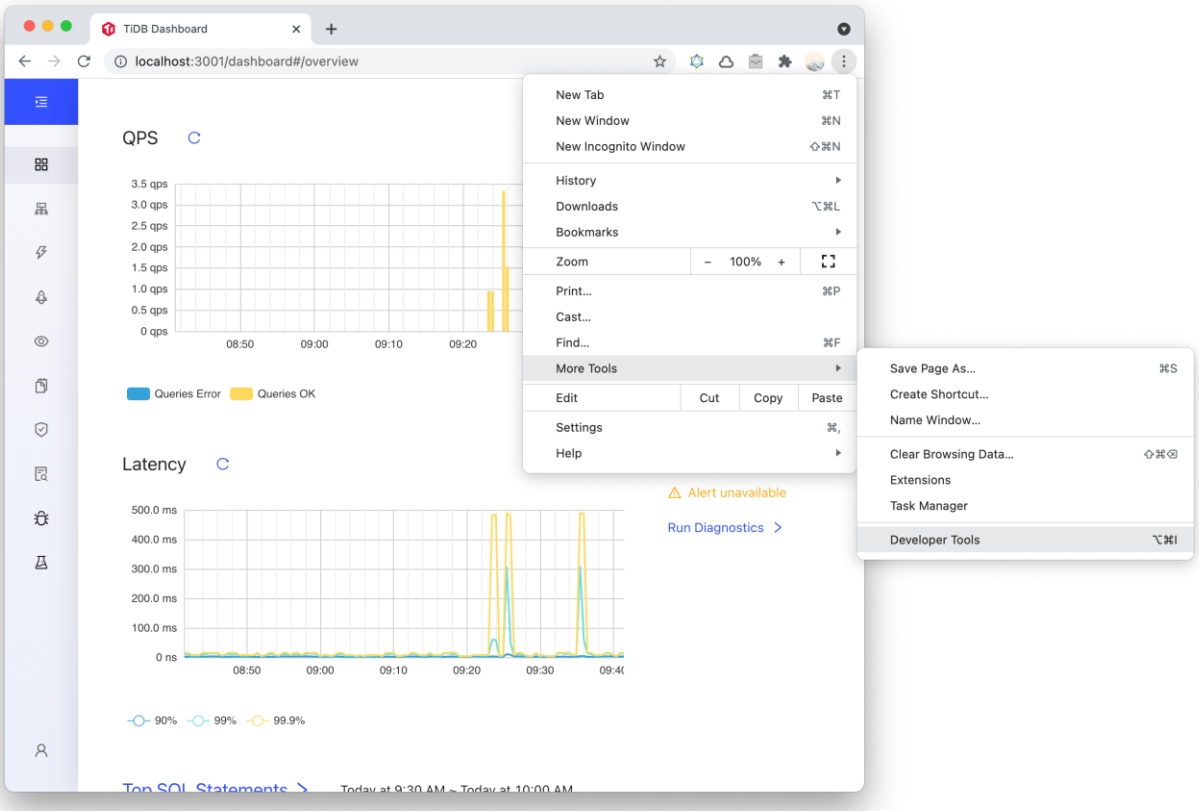


图 388: 打开开发者工具

3. 选中 Application 面板，展开 Local Storage 菜单并选中 TiDB Dashboard 页面的域名，点击 Clear All。

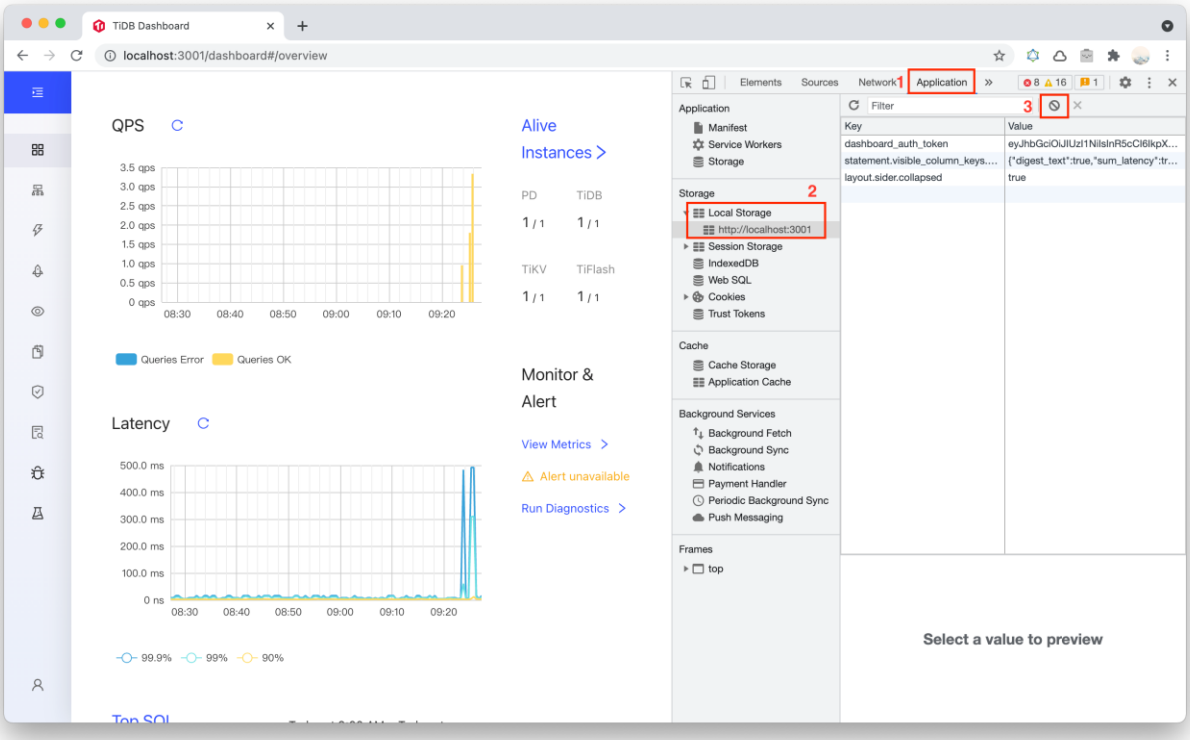


图 389: 清理 Local Storage

12 常见问题解答(FAQ)

12.1 TiDB 安装部署常见问题

本文介绍TiDB 集群安装部署的常见问题、原因及解决方法。

12.1.1 软硬件要求FAQ

12.1.1.1 TiDB 支持哪些操作系统？

关于TiDB 支持的操作系统，参见TiDB 软件和硬件环境建议配置。

12.1.1.2 TiDB 对开发、测试、生产环境的服务器硬件配置有什么要求？

TiDB 支持部署和运行在Intel x86-64 架构的 64 位通用硬件服务器平台。对于开发、测试、生产环境的服务器硬件配置，参见TiDB 软件和硬件环境建议配置- 服务器建议配置。

12.1.1.3 两块网卡的目的是？万兆的目的是？

作为一个分布式集群，TiDB 对时间的要求还是比较高的，尤其是 PD 需要分发唯一的时间戳，如果 PD 时间不统一，如果有 PD 切换，将会等待更长的时间。两块网卡可以做bond，保证数据传输的稳定，万兆可以保证数据传输的速度，千兆网卡容易出现瓶颈，我们强烈建议使用万兆网卡。

12.1.1.4 SSD 不做RAID 是否可行？

资源可接受的话，我们建议做RAID 10，如果资源有限，也可以不做RAID。

12.1.1.5 TiDB 集群各个组件的配置推荐？

- TiDB 需要 CPU 和内存比较好的机器，参考官网配置要求，如果后期需要开启TiDB Binlog，根据业务量的评估和GC 时间的要求，也需要本地磁盘大一点，不要求 SSD 磁盘；
- PD 里面存了集群元信息，会有频繁的读写请求，对磁盘I/O 要求相对比较高，磁盘太差会影响整个集群性能，推荐 SSD 磁盘，空间不用太大。另外集群Region 数量越多对 CPU、内存的要求越高；
- TiKV 对 CPU、内存、磁盘要求都比较高，一定要用 SSD 磁盘。

详情可参考TiDB 软硬件环境需求。

12.1.2 安装部署FAQ

如果用于生产环境，推荐使用TiUP 部署TiDB 集群。

12.1.2.1 为什么修改了TiKV/PD 的 toml 配置文件，却没有生效？

这种情况一般是因为没有使用 `--config` 参数来指定配置文件（目前只会出现在binary 部署的场景），TiKV/PD 会按默认值来设置。如果要使用配置文件，请设置 TiKV/PD 的 `--config` 参数。对于 TiKV 组件，修改配置后重启服务即可；对于 PD 组件，只会在第一次启动时读取配置文件，之后可以使用pd-ctl 的方式来修改配置，详情可参考 PD 配置参数。

12.1.2.2 TiDB 监控框架 Prometheus + Grafana 监控机器建议单独还是多台部署？

监控机建议单独部署。建议 CPU 8 core，内存 16 GB 以上，硬盘 500 GB 以上。

12.1.2.3 有一部分监控信息显示不出来？

查看访问监控的机器时间跟集群内机器的时间差，如果比较大，更正时间后即可显示正常。

12.1.2.4 supervise/svc/svstat 服务具体起什么作用？

- supervise 守护进程
- svc 启停服务
- svstat 查看进程状态

12.1.2.5 inventory.ini 变量参数解读

变量	含义
cluster_name	集群名称，可调整
tidb_version	TiDB 版本
deploy_method	部署方式，默认为 binary，可选 docker
process_supervision	进程监管方式，默认为 systemd，可选 supervise
timezone	修改部署目标机器时区，默认为 Asia/Shanghai，可调整，与 set_timezone 变量结合使用

变量	含义
set_timezone	默认为 True，即修改部署目标机器时区，关闭可修改为 False
enable_elk	目前不支持，请忽略
enable_firewall	开启防火墙，默认不开启
enable_ntpd	检测部署目标机器 NTP 服务，默认为 True，请勿关闭

变量	含义
machine_benchmark	检测部署目标机器磁盘 IOPS，默认为 True，请勿关闭
set_hostname_by_ip	根据 IP 修改部署目标机器主机名，默认为 False
enable_binlog	是否部署 pump 并开启 bin-log，默认为 False，依赖 Kafka 集群，参见 zookeeper_addrs 变量
zookeeper_addrs	Kafka 集群的 ZooKeeper 地址

变量	含义
enable_slow_query_log	慢查询日志记录到单独文件（{{ deploy_dir }}/log/tidb_slow_query.log），默认为 False，记录到 tidb 日志
deploy_without_tidb	模式，不部署 TiDB 服务，仅部署 PD、TiKV 及监控服务，请将 inventory.ini 文件中 tidb_servers 主机组 IP 设置为空。

12.1.2.6 如何单独记录TiDB 中的慢查询日志，如何定位慢查询 SQL？

1. TiDB 中，对慢查询的定义在TiDB 的配置文件中。`slow-threshold: 300`，这个参数是配置慢查询记录阈值的，单位是ms。
2. 如果出现了慢查询，可以从Grafana 监控定位到出现慢查询的tidb-server 以及时间点，然后在对应节点查找日志中记录的 SQL 信息。
3. 除了日志，还可以通过 `ADMIN SHOW SLOW` 命令查看，详情可参考ADMIN SHOW SLOW 命令。

12.1.2.7 首次部署TiDB 集群时，没有配置tikv 的 Label 信息，在后续如何添加配置 Label？

TiDB 的 Label 设置是与集群的部署架构相关的，是集群部署中的重要内容，是 PD 进行全局管理和调度的依据。如果集群在初期部署过程中没有设置 Label，需要在后期对部署结构进行调整，就需要手动通过 PD 的管理工具 pd-ctl 来添加location-labels 信息，例如：`config set location-labels "zone,rack,host"`（根据实际的 label 层级名字配置）。

pd-ctl 的使用参考 PD Control 使用说明。

12.1.2.8 为什么测试磁盘的dd 命令用 `oflag=direct` 这个选项？

Direct 模式就是把写入请求直接封装成I/O 指令发到磁盘，这样是为了绕开文件系统的缓存，可以直接测试磁盘的真实的I/O 读写能力。

12.1.2.9 如何用 fio 命令测试TiKV 实例的磁盘性能？

- 随机读测试：

```
./fio -ioengine=psync -bs=32k -fdatasync=1 -thread -rw=randread -size=10G -filename=
    ↪ fio_randread_test.txt -name='fio randread test' -iodepth=4 -runtime=60 -numjobs=4 -
    ↪ group_reporting --output-format=json --output=fio_randread_result.json
```

- 顺序写和随机读混合测试：

```
./fio -ioengine=psync -bs=32k -fdatasync=1 -thread -rw=randrw -percentage_random=100,0 -size
    ↪ =10G -filename=fio_randread_write_test.txt -name='fio mixed randread and sequential
    ↪ write test' -iodepth=4 -runtime=60 -numjobs=4 -group_reporting --output-format=json
    ↪ --output=fio_randread_write_test.json
```

12.1.3 TiDB 支持在公有云上部署吗？

TiDB 支持在Google Cloud GKE、 AWS EKS 和阿里云ACK 上部署使用。

此外，TiDB 云上部署也已在京东云、UCloud 上线。

12.2 迁移常见问题

本文介绍TiDB 数据迁移中的常见问题。

如果要查看迁移相关工具的常见问题，请参考以下链接：

- 备份与恢复常见问题
- TiDB Binlog 常见问题
- TiDB Lightning 常见问题
- Data Migration 常见问题
- TiCDC 常见问题

12.2.1 全量数据导出导入

12.2.1.1 如何将一个运行在 MySQL 上的应用迁移到TiDB 上？

TiDB 支持绝大多数 MySQL 语法，一般不需要修改代码。

12.2.1.2 导入导出速度慢，各组件日志中出现大量重试、EOF 错误并且没有其他错误

在没有其他逻辑出错的情况下，重试、EOF 可能是由网络问题引起的，建议首先使用相关工具排查网络连通状况。以下示例使用iperf 进行排查：

- 在出现重试、EOF 错误的服务器端节点执行以下命令：

```
iperf3 -s
```

- 在出现重试、EOF 错误的客户端节点执行以下命令：

```
iperf3 -c <server-IP>
```

下面是一个网络连接良好的客户端节点的输出：

```
$ iperf3 -c 192.168.196.58
Connecting to host 192.168.196.58, port 5201
[  5] local 192.168.196.150 port 55397 connected to 192.168.196.58 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  18.0 MBytes   150 Mbits/sec
[  5]   1.00-2.00   sec  20.8 MBytes   175 Mbits/sec
[  5]   2.00-3.00   sec  18.2 MBytes   153 Mbits/sec
[  5]   3.00-4.00   sec  22.5 MBytes   188 Mbits/sec
[  5]   4.00-5.00   sec  22.4 MBytes   188 Mbits/sec
[  5]   5.00-6.00   sec  22.8 MBytes   191 Mbits/sec
[  5]   6.00-7.00   sec  20.8 MBytes   174 Mbits/sec
[  5]   7.00-8.00   sec  20.1 MBytes   168 Mbits/sec
[  5]   8.00-9.00   sec  20.8 MBytes   175 Mbits/sec
[  5]   9.00-10.00  sec  21.8 MBytes   183 Mbits/sec
```

```
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.00  sec   208 MBytes   175 Mbits/sec                  sender
[  5]   0.00-10.00  sec   208 MBytes   174 Mbits/sec                  receiver

iperf Done.
```

如果输出显示网络带宽较低、带宽波动大，各组件日志中就可能出现大量重试、EOF 错误。此时你需要咨询网络服务供应商以提升网络质量。

如果输出的各指标良好，请尝试更新各组件版本。如果更新后仍无法解决问题，请移步 AskTUG 论坛寻求帮助。

12.2.1.3 不小心把 MySQL 的 user 表导入到TiDB 了，或者忘记密码，无法登录，如何处理？

重启TiDB 服务，配置文件中增加 `-skip-grant-table=true` 参数，无密码登录集群后，可以根据情况重建用户，或者重建mysql.user 表，具体表结构搜索官网。

12.2.1.4 如何导出TiDB 数据？

你可以通过以下方式导出TiDB 数据：

- 参考 MySQL 使用 mysqldump 导出某个表的部分数据，使用mysqldump 加 where 条件导出。
- 使用 MySQL client 将 select 的结果输出到一个文件。

12.2.1.5 如何从 DB2、 Oracle 数据库迁移到TiDB？

DB2、 Oracle 到TiDB 数据迁移（增量+ 全量），通常做法有：

- 使用 Oracle 官方迁移工具，如 OGG、 Gateway（透明网关）、 CDC (Change Data Capture)。
- 自研数据导出导入程序实现。
- 导出(Spool) 成文本文件，然后通过 Load infile 进行导入。
- 使用第三方数据迁移工具。

目前看来 OGG 最为合适。

12.2.1.6 用Sqoop 批量写入TiDB 数据，虽然配置了`--batch` 选项，但还是会遇到`java.sql.BatchUpdateExecption:statement count 5001 exceeds the transaction limitation` 的错误，该如何解决？

- 在 Sqoop 中，`--batch` 是指每个批次提交 100 条 statement，但是默认每个 statement 包含 100 条 SQL 语句，所以此时 100 * 100 = 10000 条 SQL 语句，超出了TiDB 的事务限制 5000 条，可以增加选项 `-Dsqoop.export.records.per.statement=10` 来解决这个问题，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10  \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
```

```
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大tidb 的单个事物语句数量限制，不过这个会导致内存上涨。

12.2.1.7 Dumpling 导出时引发上游数据库 OOM 或报错“磁盘空间不足”

该问题可能有如下原因：

- 数据库主键分布不均匀，例如启用了 SHARD_ROW_ID_BITS
- 上游数据库为TiDB，导出表是分区表

在上述情况下，Dumpling 划分导出子范围时，会划分出过大的子范围，从而向上游发送结果过大的查询。请联系 AskTUG 社区专家获取实验版本的 Dumpling。

12.2.1.8 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考TiDB 历史数据回溯。

12.2.2 在线数据同步

12.2.2.1 有没有现成的同步方案，可以将数据同步到 Hbase、 Elasticsearh 等其他存储？

没有，目前依赖程序自行实现。

12.2.3 业务流量迁入

12.2.3.1 如何快速迁移业务流量？

我们建议通过TiDB Data Migration 进行 MySQL -> TiDB 的业务数据的迁移；业务读写可以按照需求分阶段通过修改网络配置进行流量迁移，建议 DB 上层部署一个稳定的网络 LB（HAproxy、LVS、F5、DNS 等），这样直接修改网络配置就能实现无缝流量迁移。

12.2.3.2 TiDB 总读写流量有限制吗？

TiDB 读流量可以通过增加TiDB server 进行扩展，总读容量无限制，写流量可以通过增加TiKV 节点进行扩容，基本上写容量也没有限制。

12.2.3.3 Transaction too large 是什么原因，怎么解决？

TiDB 限制了单条 KV entry 不超过 6MB，可以修改配置文件中的txn-entry-size-limit 配置项进行调整，最大可以修改到 120MB。

分布式事务要做两阶段提交，而且底层还需要做Raft 复制。如果一个事务非常大，提交过程会非常慢，事务写冲突概率会增加，而且事务失败后回滚会导致不必要的性能开销。所以我们设置了 key-value entry 的总大小

默认不超过 100MB。如果业务需要使用大事务，可以修改配置文件中的 `txn-total-size-limit` 配置项进行调整，最大可以修改到 10G。实际的大小限制还受机器的物理内存影响。

在 Google 的 Cloud Spanner 上面，也有类似的限制。

12.2.3.4 如何批量导入？

导入数据的时候，可以分批插入，每批最好不要超过 1w 行。

12.2.3.5 TiDB 中删除数据后会立即释放空间吗？

DELETE，TRUNCATE 和 DROP 都不会立即释放空间。对于TRUNCATE 和 DROP 操作，在达到TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的GC 机制会删除数据并释放空间。对于 DELETE 操作TiDB 的GC 机制会删除数据，但不会释放空间，而是当后续数据写入RocksDB 且进行compact 时对空间重新利用。

12.2.3.6 Load 数据时可以对目标表执行 DDL 操作吗？

不可以，加载数据期间不能对目标表执行任何 DDL 操作，这会导致数据加载失败。

12.2.3.7 TiDB 是否支持replace into 语法？

支持。

12.2.3.8 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。

12.2.3.9 数据删除最高效最快的方式？

在删除大量数据的时候，建议使用 `Delete from t where xx limit 5000`（xx 建议在满足业务过滤逻辑下，尽量加上强过滤索引列或者直接使用主键选定范围，如 `id >= 5000*n+m and id <= 5000*(n+1)+m` 这样的方案，通过循环来删除，用 `Affected Rows == 0` 作为循环结束条件，这样避免遇到事务大小的限制。如果一次删除的数据量非常大，这种循环的方式会越来越慢，因为每次删除都是从前向后遍历，前面的删除之后，短时间内会残留不少删除标记（后续会被GC 掉），影响后面的 Delete 语句。如果有可能，建议把 Where 条件细化。可以参考官网最佳实践。

12.2.3.10 TiDB 如何提高数据加载速度？

主要有两个方面：

- 目前已开发分布式导入工具TiDB Lightning，需要注意的是数据导入过程中为了性能考虑，不会执行完整的事务流程，所以没办法保证导入过程中正在导入的数据的ACID 约束，只能保证整个导入过程结束以后导入数据的ACID 约束。因此适用场景主要为新数据的导入（比如新的表或者新的索引），或者是全量的备份恢复（先 Truncate 原表再导入）。
- TiDB 的数据加载与磁盘以及整体集群状态相关，加载数据时应关注该主机的磁盘利用率，TiClient Error/Backoff/Thread CPU 等相关metric，可以分析相应瓶颈。

12.3 升级与升级后常见问题

本文介绍TiDB 升级与升级后的常见问题与解决办法。

12.3.1 升级常见问题

本小节列出了TiDB 升级相关的常见问题与解决办法。

12.3.1.1 滚动升级有那些影响？

滚动升级TiDB 期间，业务运行会受到一定影响。因此，不建议在业务高峰期进行滚动升级。需要配置最小集群拓扑(TiDB * 2、 PD * 3、 TiKV * 3)，如果集群环境中有 Pump 和 Drainer 服务，建议先停止 Drainer，然后滚动升级（升级TiDB 时会升级 Pump）。

12.3.1.2 集群在执行 DDL 请求期间可以进行升级操作吗？

- 如果升级前TiDB 的版本低于 v7.1.0：
  - 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。
  - 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。
- 如果升级前TiDB 的版本为 v7.1.0 或更高的版本：
  - 不用遵循限制低版本升级时的限制，即在升级时可以接收用户 DDL 任务。建议参考平滑升级TiDB。

12.3.1.3 Binary 如何升级？

不推荐使用 Binary 来升级集群。建议使用TiUP 升级TiDB 进行升级，确保分布式系统版本一致性和兼容性。

12.3.2 升级后常见问题

本小节列出了一些升级后可能会遇到的问题与解决办法。

12.3.2.1 执行 DDL 操作时遇到的字符集(charset) 问题

TiDB 在 v2.1.0 以及之前版本（包括 v2.0 所有版本）中，默认字符集是 UTF8。从 v2.1.1 开始，默认字符集变更为 UTF8MB4。如果在 v2.1.0 及之前版本中，建表时显式指定了table 的 charset 为 UTF8，那么升级到 v2.1.1 之后，执行 DDL 操作可能会失败。

要避免该问题，需注意以下两个要点：

- 在 v2.1.3 之前，TiDB 不支持修改column 的 charset。所以，执行 DDL 操作时，新column 的 charset 需要和旧column 的 charset 保持一致。
- 在 v2.1.3 之前，即使column 的 charset 和 table 的 charset 不一样，`show create table` 也不会显示column 的 charset，但可以通过 HTTP API 获取table 的元信息来查看column 的 charset，下文提供了示例。

12.3.2.1.1 unsupported modify column charset utf8mb4 not match origin utf8

- 升级前：v2.1.0 及之前版本

```
create table t(a varchar(10)) charset=utf8;
```

```
Query OK, 0 rows affected
Time: 0.106s
```

```
show create table t
```

```
+-------+-------------------------------------------------------------+
| Table | Create Table                                                |
+-------+-------------------------------------------------------------+
| t     | CREATE TABLE `t` (                                          |
|       |   `a` varchar(10) DEFAULT NULL                              |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin |
+-------+-------------------------------------------------------------+
1 row in set
Time: 0.006s
```

- 升级后：v2.1.1、v2.1.2 会出现下面的问题，v2.1.3 以及之后版本不会出现下面的问题。

```
alter table t change column a a varchar(20);
```

```
ERROR 1105 (HY000): unsupported modify column charset utf8mb4 not match origin utf8
```

解决方案：显式指定column charset，保持和原来的charset 一致即可。

```
alter table t change column a a varchar(22) character set utf8;
```

- 根据要点 1，此处如果不指定column 的 charset，会用默认的 UTF8MB4，所以需要指定 column charset 保持和原来一致。
- 根据要点 2，用 HTTP API 获取table 元信息，然后根据column 名字和 Charset 关键字搜索即可找到column 的 charset。

```
curl "http://$IP:10080/schema/test/t" | python -m json.tool
```

这里用了python 的格式化json 的工具，也可以不加，此处只是为了方便注释。

```
{
    "ShardRowIDBits": 0,
    "auto_inc_id": 0,
    "charset": "utf8",    # table 的 charset
    "collate": "",
```

```
    "cols": [            # 从这里开始列举 column 的相关信息
        {
            ...
            "id": 1,
            "name": {
                "L": "a",
                "O": "a"    # column 的名字
            },
            "offset": 0,
            "origin_default": null,
            "state": 5,
            "type": {
                "Charset": "utf8",    # column a 的 charset
                "Collate": "utf8_bin",
                "Decimal": 0,
                "Elems": null,
                "Flag": 0,
                "Flen": 10,
                "Tp": 15
            }
        }
    ],
    ...
}
```

12.3.2.1.2 unsupported modify charset from utf8mb4 to utf8

- **升级前：** v2.1.1，v2.1.2

```
create table t(a varchar(10)) charset=utf8;
```

```
Query OK, 0 rows affected
Time: 0.109s
```

```
show create table t;
```

```
+-------+-------------------------------------------------------------+
| Table | Create Table                                                |
+-------+-------------------------------------------------------------+
| t     | CREATE TABLE `t` (                                          |
|       |   `a` varchar(10) DEFAULT NULL                              |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin |
+-------+-------------------------------------------------------------+
```

上面 show create table 只显示出了table 的 charset，但其实column 的 charset 是 UTF8MB4，这可以通过 HTTP API 获取 schema 来确认。这是一个bug，即此处建表时column 的 charset 应该要和table 保持一致为 UTF8，该问题在 v2.1.3 中已经修复。

- 升级后：v2.1.3 及之后版本

```
show create table t;
```

```
+-------+-------------------------------------------------------------------+
| Table | Create Table                                                      |
+-------+-------------------------------------------------------------------+
| t     | CREATE TABLE `t` (                                                |
|       |   `a` varchar(10) CHARSET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin             |
+-------+-------------------------------------------------------------------+
1 row in set
Time: 0.007s
```

```
alter table t change column a a varchar(20);
```

```
ERROR 1105 (HY000): unsupported modify charset from utf8mb4 to utf8
```

解决方案：

- 因为在 v2.1.3 之后，TiDB 支持修改column 和table 的charset，所以这里推荐修改 table 的charset 为 UTF8MB4。

```
alter table t convert to character set utf8mb4;
```

- 也可以像问题 1 一样指定column 的 charset，保持和column 原来的charset (UTF8MB4) 一致即可。

```
alter table t change column a a varchar(20) character set utf8mb4;
```

12.3.2.1.3 ERROR 1366 (HY000): incorrect utf8 value f09f8c80(□) for column a

TiDB 在 v2.1.1 及之前版本中，如果charset 是 UTF8，没有对 4-byte 的插入数据进行 UTF8 Unicode encoding 检查。在 v2.1.2 及之后版本中，添加了该检查。

- 升级前：v2.1.1 及之前版本

```
create table t(a varchar(100) charset utf8);
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

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

- 升级后：v2.1.2 及之后版本

```
insert t values (unhex('f09f8c80'));
```

```
ERROR 1366 (HY000): incorrect utf8 value f09f8c80(□) for column a
```

解决方案：

- v2.1.2 版本：该版本不支持修改column charset，所以只能跳过 UTF8 的检查。

```
set @@session.tidb_skip_utf8_check=1;
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

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

- v2.1.3 及之后版本：建议修改column 的 charset 为 UTF8MB4。或者也可以设置 `tidb_skip_utf8_check` 变量跳过 UTF8 的检查。如果跳过 UTF8 的检查，在需要将数据从TiDB 同步回 MySQL 的时候，可能会失败，因为 MySQL 会执行该检查。

```
alter table t change column a a varchar(100) character set utf8mb4;
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

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

关于 `tidb_skip_utf8_check` 变量，具体来说是指跳过 UTF8 和 UTF8MB4 类型对数据的合法性检查。如果跳过这个检查，在需要将数据从TiDB 同步回 MySQL 的时候，可能会失败，因为 MySQL 执行该检查。如果只想跳过 UTF8 类型的检查，可以设置 `tidb_check_mb4_value_in_utf8` 变量。

`tidb_check_mb4_value_in_utf8` 在 v2.1.3 版本加入 `config.toml` 文件，可以修改配置文件里面的 `check-mb4-value-in-utf8` 后重启集群生效。

`tidb_check_mb4_value_in_utf8` 在 v2.1.5 版本开始可以用 HTTP API 来设置，也可以用 session 变量来设置。

- HTTP API（HTTP API 只在单台服务器上生效）
  - 执行下列命令启用 HTTP API：

    ```
    curl -X POST -d "check_mb4_value_in_utf8=1" http://{TiDBIP}:10080/settings
    ```

  - 执行下列命令禁用 HTTP API：

    ```
    curl -X POST -d "check_mb4_value_in_utf8=0" http://{TiDBIP}:10080/settings
    ```

- Session 变量
  * 执行下列命令启用 Session 变量：

```
set @@session.tidb_check_mb4_value_in_utf8 = 1;
```

  * 执行下列命令禁用 Session 变量：

```
set @@session.tidb_check_mb4_value_in_utf8 = 0;
```

- v2.1.7 及之后版本，如果对表和column 的字符集没有严格要求为 UTF8，也不想修改客户端代码去跳过 UTF8 检查或者手动修改column 的 charset，可以在配置文件中把 `treat-old-version-utf8-as-utf8mb4` 打开。该配置的作用是自动把 v2.1.7 版本之前创建的旧版本的表和column 的 UTF8 字符集转成 UTF8MB4。这个转换是在TiDB load schema 时在内存中将 UTF8 转成 UTF8MB4，不会对实际存储的数据做任何修改。在配置文件中关闭 `treat-old-version-utf8-as-utf8mb4` 并重启TiDB 后，以前字符集为 UTF8 的表和column 的字符集仍然还是 UTF8。

> **注意：**
>
> `treat-old-version-utf8-as-utf8mb4` 参数默认打开，如果客户端强制需要用 UTF8 而不用 UTF8MB4，需要在配置文件中关闭。

12.4 TiDB 监控常见问题

本文介绍在监控TiDB 集群时的常见问题、原因及解决方法。

- Prometheus 监控框架详情可见TiDB 监控框架概述。
- 监控指标解读详细参考重要监控指标详解。

12.4.1 目前的监控使用方式及主要监控指标，有没有更好看的监控？

TiDB 使用 Prometheus + Grafana 组成TiDB 数据库系统的监控系统。用户在Grafana 上通过dashboard 可以监控到TiDB 的各类运行指标，包括

- 系统资源的监控指标
- 客户端连接与 SQL 运行的指标
- 内部通信和Region 调度的指标

通过这些指标，可以让数据库管理员更好的了解到系统的运行状态，运行瓶颈等内容。在监控指标的过程中，我们按照TiDB 不同的模块，分别列出了各个模块重要的指标项，一般用户只需要关注这些常见的指标项。具体指标请参见官方文档。

12.4.2 Prometheus 监控数据默认 15 天自动清除一次，可以自己设定成 2 个月或者手动删除吗？

可以的，在 Prometheus 启动的机器上，找到启动脚本，然后修改启动参数，然后重启 Prometheus 生效。

```
--storage.tsdb.retention="60d"
```

12.4.3 Region Health 监控项

TiDB-2.0 版本中，PD metric 监控页面中，对Region 健康度进行了监控，其中Region Health 监控项是对所有Region 副本状况的一些统计。其中miss 是缺副本，extra 是多副本。同时也增加了按 Label 统计的隔离级别，level-1 表示这些Region 的副本在第一级 Label 下是物理隔离的，没有配置location label 时所有Region 都在level-0。

12.4.4 Statement Count 监控项中的 selectsimplefull 是什么意思？

代表全表扫，但是可能是很小的系统表。

12.4.5 监控上的 QPS 和 Statement OPS 有什么区别？

QPS 会统计执行的所有 SQL 命令，包括use database、 load data、 begin、 commit、 set、 show、 insert、 select 等。

Statement OPS 只统计 select、 update、 insert 等业务相关的，所以 Statement OPS 的统计和业务比较相符。

12.5 TiDB 集群管理常见问题

本文介绍管理TiDB 集群时的常见问题、原因及解决方法。

12.5.1 集群日常管理

本小节介绍集群日程管理中的常见问题、原因及解决方法。

12.5.1.1 TiDB 如何登录？

和 MySQL 登录方式一样，可以按照下面例子进行登录。

```
mysql -h 127.0.0.1 -uroot -P4000
```

12.5.1.2 TiDB 如何修改数据库系统变量？

和 MySQL 一样，TiDB 也分为静态参数和固态参数，静态参数可以直接通过 `SET GLOBAL xxx = n` 的方式进行修改，不过新参数值只限于该实例生命周期有效。

12.5.1.3 TiDB (TiKV) 有哪些数据目录？

默认在`--data-dir` 目录下，其中包括backup、 db、 raft、 snap 四个目录，分别存储备份、 数据、 raft 数据及镜像数据。

12.5.1.4 TiDB 有哪些系统表？

和 MySQL 类似，TiDB 中也有系统表，用于存放数据库运行时所需信息，具体信息参考TiDB 系统表文档。

12.5.1.5 TiDB 各节点服务器下是否有日志文件，如何管理？

默认情况下各节点服务器会在日志中输出标准错误，如果启动的时候通过 `--log-file` 参数指定了日志文件，那么日志会输出到指定的文件中，并且按天做rotation。

12.5.1.6 TiDB、TiKV、PD 节点的各类文件存放在哪里？

如需快速了解TiDB 节点、TiKV 节点、PD 节点的配置文件、数据文件及日志文件的相关介绍与其存放位置，建议观看下面的培训视频（时长 9 分钟）。

12.5.1.7 如何规范停止TiDB？

- 若使用了load balancer（推荐）：先停止load balancer，然后执行 SHUTDOWN 语句。此时 TiDB 会根据graceful-wait-before-shutdown 设置值等待所有会话断开，然后停止运行。
- 若未使用load balancer：执行 SHUTDOWN 语句，TiDB 组件会做 graceful shutdown。

12.5.1.8 TiDB 里面可以执行 kill 命令吗？

- Kill DML 语句：

  查询 `information_schema.cluster_processlist`，获取正在执行 DML 语句的TiDB 实例地址和 session ID，然后执行 kill 命令。

  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，直接执行 `kill session_id` 即可。

  对于TiDB v6.1.0 之前的版本，或未启用Global Kill 功能时，`kill session_id` 默认不生效，客户端需要连接到正在执行 DML 语句的TiDB 实例，然后执行 `kill tidb session_id` 才能 kill DML 语句。如果客户端连接到其他TiDB 实例或者客户端和TiDB 集群之间有代理，`kill tidb session_id` 可能会被路由到其他的 TiDB 实例，从而错误地终止其他会话。具体可以参考 KILL。

- Kill DDL 语句：执行 `admin show ddl jobs`，查找需要 kill 的 DDL job ID，然后执行 `admin cancel ddl jobs 'job_id' [, 'job_id'] ...`。具体可以参考 ADMIN。

12.5.1.9 TiDB 是否支持会话超时？

TiDB 目前支持wait_timeout 和interactive_timeout 两种超时。

12.5.1.10 TiDB 的版本管理策略是怎么样的？

关于TiDB 版本的管理策略，可以参考TiDB 版本规则。

12.5.1.11 部署和维护TiDB 集群的运营成本如何？

TiDB 提供了一些特性和工具，可以帮助你以低成本管理集群：

- 在运维方面，TiUP 作为包管理器，简化了部署、扩缩容、升级和其他运维任务。

- 在监控方面，TiDB 监控框架使用 Prometheus 存储监控和性能指标，使用Grafana 可视化这些指标。Grafana 内置了数十个面板，覆盖了数百个指标。
- 在故障诊断方面，TiDB 集群问题导图汇总了TiDB 服务器和其他组件的常见问题。你可以使用这个导图来诊断和解决遇到的相关问题。

12.5.1.12 分不清TiDB master 版本之间的区别，应该怎么办？

TiDB 目前社区非常活跃，同时，我们还在不断的优化和修改 BUG，因此 TiDB 的版本更新周期比较快，会不定期有新版本发布，请关注我们的版本发布时间线。此外TiDB 安装推荐使用TiUP 进行安装或使用TiDB Operator 进行安装。TiDB 的版本号目前实现了统一管理，你可以通过如下任意方式查看TiDB 的版本号：

- 通过 `select tidb_version()` 进行查看
- 通过执行 `tidb-server -V` 进行查看

12.5.1.13 有没有图形化部署TiDB 的工具？

有。你可以使用TiUniManager，它是一款为分布式数据库TiDB 打造的管控平台软件和数据库运维管理平台，为TiDB 提供数据库集群管理功能、主机管理功能和平台管理功能，涵盖了数据库运维人员(DBA) 在 TiDB 上进行的常用运维操作，帮助 DBA 对 TiDB 进行自动化、自助化和可视化管理。

12.5.1.14 如何扩容TiDB 集群？

可以在不影响线上服务的情况下，对TiDB 集群进行扩容。

- 如果是使用TiUP 部署的集群，可以参考使用TiUP 扩容TiDB 集群。
- 如果是使用TiDB Operator 在 Kubernetes 上部署的集群，可以参考在 Kubernetes 中手动扩容TiDB 集群。

12.5.1.15 TiDB 如何进行水平扩展？

当您的业务不断增长时，数据库可能会面临三方面瓶颈，第一是存储空间，第二是计算资源，第三是读写容量，这时可以对TiDB 集群做水平扩展。

- 如果是存储资源不够，可以通过添加TiKV Server 节点来解决，新节点启动后，PD 会自动将其他节点的部分数据迁移过去，无需人工介入。
- 如果是计算资源不够，可以查看TiDB Server 和 TiKV Server 节点的 CPU 消耗情况，再考虑添加TiDB Server 节点或者是TiKV Server 节点来解决，如添加TiDB Server 节点，将其添加到前端 Load Balancer 配置之中即可。
- 如果是容量跟不上，一般可以考虑同时增加TiDB Server 和 TiKV Server 节点。

12.5.1.16 Percolator 用了分布式锁，crash 的客户端会保持锁，会造成锁没有release？

详细可参考 Percolator 和 TiDB 事务算法。

12.5.1.17 TiDB 为什么选用 gRPC 而不选用 Thrift，是因为 Google 在用吗？

不只是因为Google 在用，有一些比较好的特性我们需要，比如流控、加密还有 Streaming。

12.5.1.18 like(bindo.customers.name, jason%, 92) 这个 92 代表什么？

那个是转义字符，默认是(ASCII 92)。

12.5.1.19 为什么 information_schema.tables.data_length 记录的大小和TiKV 监控面板上的 store size 不一样？

这是因为两者计算的角度不一样。information_schema.tables.data_length 是通过统计信息（平均每行的大小）得到的估算值。TiKV 监控面板上的 store size 是单个TiKV 实例的数据文件（RocksDB 的 SST 文件）的大小总和。由于多版本和TiKV 会压缩数据，所以两者显示的大小不一样。

12.5.1.20 为什么事务没有使用异步提交或一阶段提交？

在以下情况中，即使通过系统变量开启了异步提交和一阶段提交，TiDB 也不会使用这些特性：

- 如果开启了TiDB Binlog，受 TiDB Binlog 的实现原理限制，TiDB 不会使用异步提交或一阶段提交特性。
- TiDB 只在事务写入不超过 256 个键值对，以及所有键值对里键的总大小不超过 4 KB 时，才会使用异步提交或一阶段提交特性。这是因为对于写入量大的事务，异步提交不能明显提升执行性能。

12.5.2 PD 管理

本小节介绍 PD 管理中的常见问题、原因及解决方法。

12.5.2.1 访问 PD 报错：TiKV cluster is not bootstrapped

PD 的大部分API 需要在初始化TiKV 集群以后才能使用，如果在部署新集群的时候只启动了 PD，还没有启动TiKV，这时候访问 PD 就会报这个错误。遇到这个错误应该先把要部署的TiKV 启动起来，TiKV 会自动完成初始化工作，然后就可以正常访问 PD。

12.5.2.2 PD 启动报错：etcd cluster ID mismatch

PD 启动参数中的 --initial-cluster 包含了某个不属于该集群的成员。遇到这个错误时请检查各个成员的所属集群，剔除错误的成员后即可正常启动。

12.5.2.3 PD 能容忍的时间同步误差是多少？

理论上，时间同步误差越小越好。PD 可容忍任意时长的误差，但是，时间同步误差越大意味着 PD 分配的时间戳与真实的物理时间相差越大，这个差距会影响读历史版本等功能。

12.5.2.4 Client 连接是如何寻找 PD 的？

Client 连接只能通过TiDB 访问集群，TiDB 负责连接 PD 与TiKV，PD 与TiKV 对 Client 透明。当TiDB 连接任意一台 PD 的时候，PD 会告知TiDB 当前的leader 是谁，如果此台 PD 不是leader，TiDB 将会重新连接至leader PD。

12.5.2.5 TiKV 节点(Store) 各状态(Up, Disconnect, Offline, Down, Tombstone) 之间的关系是什么？

使用 pd-ctl 可以查看TiKV 节点的状态信息。如需查看各个状态之间的关系，请参考TiKV Store 状态之间的关系。

12.5.2.6 PD 参数中leader-schedule-limit 和 region-schedule-limit 调度有什么区别？

- leader-schedule-limit 调度是用来均衡不同TiKV 的 leader 数，影响处理查询的负载。
- region-schedule-limit 调度是均衡不同TiKV 的副本数，影响不同节点的数据量。

12.5.2.7 每个region 的 replica 数量可配置吗？调整的方法是？

可以，目前只能调整全局的replica 数量。首次启动时 PD 会读配置文件(conf/pd.yml)，使用其中的max-replicas 配置，之后修改需要使用pd-ctl 配置命令 `config set max-replicas $num`，配置后可通过 `config show all` 来查看已生效的配置。调整的时候，不会影响业务，会在后台添加，注意总TiKV 实例数总是要大于等于设置的副本数，例如 3 副本需要至少 3 个 TiKV。增加副本数量之前需要预估额外的存储需求。pd-ctl 的详细用法可参考 PD Control 使用说明。

12.5.2.8 缺少命令行集群管理工具，整个集群的健康度当前是否正常，不好确认？

可以通过pd-ctl 等工具来判断集群大概的状态，详细的集群状态还是需要通过监控来确认。

12.5.2.9 集群下线节点后，怎么删除老集群节点监控信息？

下线节点一般指TiKV 节点通过pd-ctl 或者监控判断节点是否下线完成。节点下线完成后，手动停止下线节点上相关的服务。从 Prometheus 配置文件中删除对应节点的node_exporter 信息。

12.5.3 TiDB server 管理

本小节介绍TiDB server 管理中的常见问题、原因及解决方法。

12.5.3.1 TiDB 的 lease 参数应该如何设置？

启动TiDB Server 时，需要通过命令行参数设置lease 参数(`--lease=60`)，其值会影响 DDL 的速度（只会影响当前执行 DDL 的 session，其他的 session 不会受影响）。在测试阶段，lease 的值可以设为 1s，加快测试进度；在生产环境下，我们推荐这个值设为分钟级（一般可以设为 60），这样可以保证 DDL 操作的安全。

12.5.3.2 DDL 在正常情况下的耗时是多少？

一般情况下处理一个 DDL 操作（之前没有其他 DDL 操作在处理）的耗时基本可以分如下为三种：

- add index 操作，且此操作对应表数据行数比较少，耗时约为 3s。
- add index 操作，且此操作对应表数据行数比较多，耗时具体由表中数据行数和当时 QPS 情况定（add index 操作优先级比一般 SQL 低）。
- 其他 DDL 操作耗时约为 1s。

此外，如果接收 DDL 请求的TiDB 和 DDL owner 所处的TiDB 是一台，那么上面列举的第一和第三种可能的耗时应该在几十到几百毫秒。

12.5.3.3 为什么有的时候执行 DDL 会很慢？

可能原因如下：

- 多个 DDL 语句一起执行的时候，后面的几个 DDL 语句会比较慢。原因是当前TiDB 集群中 DDL 操作是串行执行的。
- 在正常集群启动后，第一个 DDL 操作的执行时间可能会比较久，一般在 30s 左右，这个原因是刚启动时 TiDB 在竞选处理 DDL 的 leader。
- 由于停TiDB 时不能与 PD 正常通信（包括停电情况）或者用 `kill -9` 指令停TiDB 导致TiDB 没有及时从 PD 清理注册数据，那么会影响TiDB 启动后 10min 内的 DDL 语句处理时间。这段时间内运行 DDL 语句时，每个 DDL 状态变化都需要等待 2 * lease（默认 lease = 45s）。
- 当集群中某个TiDB 与 PD 之间发生通信问题，即TiDB 不能从 PD 及时获取或更新版本信息，那么这时候 DDL 操作的每个状态处理需要等待 2 * lease。

12.5.3.4 TiDB 可以使用 S3 作为后端存储吗？

不可以，目前TiDB 只支持分布式存储引擎和GolevelDB/RocksDB/BoltDB 引擎。

12.5.3.5 Information_schema 能否支持更多真实信息？

Information_schema 库里面的表主要是为了兼容 MySQL 而存在，有些第三方软件会查询里面的信息。在目前 TiDB 的实现中，里面大部分只是一些空表。后续随着TiDB 的升级，会提供更多的参数信息。当前TiDB 支持的 Information_schema 请参考TiDB 系统数据库说明文档。

12.5.3.6 TiDB Backoff type 主要原因？

TiDB-server 与 TiKV-server 随时进行通信，在进行大量数据操作过程中，会出现 `Server is busy` 或者 `backoff.maxsleep 20000ms` 的日志提示信息，这是由于TiKV-server 在处理过程中系统比较忙而出现的提示信息，通常这时候可以通过系统资源监控到TiKV 主机系统资源使用率比较高的情况出现。如果这种情况出现，可以根据资源使用情况进行相应的扩容操作。

12.5.3.7 TiDB TiClient type 主要原因？

TiClient Region Error 该指标描述的是在TiDB-server 作为客户端通过 KV 接口访问TiKV-server 进行数据操作过程中，TiDB-server 操作TiKV-server 中的Region 数据出现的错误类型与metric 指标，错误类型包括not_leader、stale_epoch。出现这些错误的情况是当 TiDB-server 根据自己的缓存信息去操作 Region leader 数据的时候，Region leader 发生了迁移或者 TiKV 当前的 Region 信息与 TiDB 缓存的路由信息不一致而出现的错误提示。一般这种情况下，TiDB-server 都会自动重新从 PD 获取最新的路由数据，重做之前的操作。

12.5.3.8 TiDB 同时支持的最大并发连接数？

默认情况下，每个TiDB 服务器的最大连接数没有限制。如有需要，可以在 `config.toml` 文件中设置 `instance.max_connections`（或者系统变量 `max_connections`）来限制最大连接数。如果并发量过大导致响应时间增加，建议通过添加TiDB 节点进行扩容。

12.5.3.9 如何查看某张表创建的时间？

information_schema 库中的tables 表里的create_time 即为表的真实创建时间。

12.5.3.10 TiDB 的日志中EXPENSIVE_QUERY 是什么意思？

TiDB 在执行 SQL 时，预估出来每个 operator 处理了超过 10000 条数据就认为这条query 是 expensive query。可以通过修改tidb-server 配置参数来对这个门限值进行调整，调整后需要重新启动tidb-server。

12.5.3.11 如何预估TiDB 中一张表的大小？

要预估TiDB 中一张表的大小，你可以参考使用以下查询语句：

```
SELECT
    db_name,
    table_name,
    ROUND(SUM(total_size / cnt), 2) Approximate_Size,
    ROUND(SUM(total_size / cnt / (SELECT
                    ROUND(AVG(value), 2)
                FROM
                    METRICS_SCHEMA.store_size_amplification
                WHERE
                    value > 0)),
            2) Disk_Size
FROM
    (SELECT
        db_name,
            table_name,
            region_id,
            SUM(Approximate_Size) total_size,
            COUNT(*) cnt
    FROM
        information_schema.TIKV_REGION_STATUS
    WHERE
        db_name = @dbname
            AND table_name IN (@table_name)
    GROUP BY db_name , table_name , region_id) tabinfo
GROUP BY db_name , table_name;
```

在使用以上语句时，你需要根据实际情况填写并替换掉语句里的以下字段：

- `@dbname`：数据库名称。
- `@table_name`：目标表的名称。

此外，以上语句中：

- `store_size_amplification` 表示集群压缩比的平均值。除了使用 `SELECT * FROM METRICS_SCHEMA.store_size_amplification;` 语句进行查询以外， 你还可以查看 Grafana 监控 PD - statistics balance 面板下各节点的 `Size amplification` 指标来获取该信息， 集群压缩比的平均值即为所有节点的 `Size amplification` 平均值。
- `Approximate_Size` 表示压缩前表的单副本大小，该值为估算值，并非准确值。
- `Disk_Size` 表示压缩后表的大小，可根据 `Approximate_Size` 和 `store_size_amplification` 得出估算值。

12.5.4 TiKV 管理

本小节介绍TiKV 管理中的常见问题、原因及解决方法。

12.5.4.1 如何为合规性或多租户应用程序指定数据位置？

可以使用放置规则(Placement Rules) 为合规性或多租户应用程序指定数据位置。

Placement Rules in SQL 用于控制任何连续数据范围的属性，例如副本数量、 Raft 角色、放置位置以及规则生效的键范围。

12.5.4.2 TiKV 集群副本建议配置数量是多少，是不是最小高可用配置（3 个）最好？

如果是测试环境 3 副本足够；在生产环境中，不可让集群副本数低于 3，需根据架构特点、业务系统及恢复能力的需求，适当增加副本数。值得注意的是，副本升高，性能会有下降，但是安全性更高。

12.5.4.3 TiKV 启动报错：cluster ID mismatch

TiKV 本地存储的cluster ID 和指定的 PD 的 cluster ID 不一致。在部署新的 PD 集群的时候，PD 会随机生成一个 cluster ID，TiKV 第一次初始化的时候会从 PD 获取cluster ID 存储在本地，下次启动的时候会检查本地的cluster ID 与 PD 的 cluster ID 是否一致，如果不一致则会报错并退出。出现这个错误一个常见的原因是，用户原先部署了一个集群，后来把 PD 的数据删除了并且重新部署了新的 PD，但是 TiKV 还是使用旧的数据重启连到新的 PD 上，就会报这个错误。

12.5.4.4 TiKV 启动报错：duplicated store address

启动参数中的地址已经被其他的TiKV 注册在 PD 集群中了。造成该错误的常见情况：TiKV `--data-dir` 指定的路径下没有数据文件夹（删除或移动后没有更新–data-dir），用之前参数重新启动该 TiKV。请尝试用pd-ctl 的 store delete 功能，删除之前的 store，然后重新启动 TiKV 即可。

12.5.4.5 TiKV master 和 slave 用的是一样的压缩算法，为什么效果不一样？

目前来看master 有些文件的压缩率会高一些，这个取决于底层数据的分布和RocksDB 的实现，数据大小偶尔有些波动是正常的，底层存储引擎会根据需要调整数据。

12.5.4.6 TiKV block cache 有哪些特性？

TiKV 使用了RocksDB 的 Column Family (CF) 特性，KV 数据最终存储在默认RocksDB 内部的default、 write、 lock 3 个 CF 内。

- default CF 存储的是真正的数据，与其对应的参数位于 `[rocksdb.defaultcf]` 项中。
- write CF 存储的是数据的版本信息(MVCC)、索引、小表相关的数据，相关的参数位于 `[rocksdb.writecf]` 项中。
- lock CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储Raft log。 default CF 主要存储的是Raft log，与其对应的参数位于 `[raftdb.defaultcf]` 项中。

- 所有 CF 共享一个 Block-cache，用于缓存数据块，加速 RocksDB 的读取速度。Block-cache 的大小通过参数 `block-cache-size` 控制，`block-cache-size` 越大，能够缓存的热点数据越多，对读取操作越有利，同时占用的系统内存也会越多。
- 每个 CF 有各自的Write-buffer，大小通过 `write-buffer-size` 控制。

12.5.4.7 TiKV channel full 是什么原因？

- Raftstore 线程太忙，或者因I/O 而卡住。可以看一下Raftstore 的 CPU 使用情况。
- TiKV 过忙（CPU、磁盘 I/O 等），请求处理不过来。

12.5.4.8 TiKV 频繁切换Region leader 是什么原因？

- 网络问题导致节点间通信卡了，查看Report failures 监控。
- 原主 Leader 的节点卡了，导致没有及时给 Follower 发送消息。
- Raftstore 线程卡了。

12.5.4.9 如果一个节点挂了会影响服务吗？影响会持续多久？

TiDB 使用 Raft 在多个副本之间做数据同步（默认为每个Region 3 个副本）。当一份备份出现问题时，其他的副本能保证数据的安全。根据Raft 协议，当某个节点挂掉导致该节点里的 Leader 失效时，在最大 2 * lease time（leasetime 是 10 秒）时间后，通过Raft 协议会很快将一个另外一个节点里的 Follower 选为新的Region Leader 来提供服务。

12.5.4.10 TiKV 在分别在哪些场景下占用大量IO、内存、CPU（超过参数配置的多倍）？

在大量写入、读取的场景中会占用大量的磁盘IO、内存、CPU。在执行很复杂的查询，比如会产生很大中间结果集的情况下，会消耗很多的内存和 CPU 资源。

12.5.4.11 TiKV 是否可以使用 SAS/SATA 盘或者进行 SSD/SAS 混合部署？

不可以使用。TiDB 在进行 OLTP 场景中，数据访问和操作需要高IO 磁盘的支持。TiDB 作为强一致的分布式数据库，存在一定的写放大，如副本复制、存储底层 Compaction，因此，TiDB 部署的最佳实践中推荐用户使用 NVMe SSD 磁盘作为数据存储磁盘。另外，TiKV 与 PD 不能混合部署。

12.5.4.12 数据表 Key 的Range 范围划分是在数据接入之前就已经划分好了吗？

不是的，这个和 MySQL 分表规则不一样，需要提前设置好，TiKV 是根据Region 的大小动态分裂的。

12.5.4.13 Region 是如何进行分裂的？

Region 不是前期划分好的，但确实有Region 分裂机制。当Region 的大小超过参数 `region-max-size` 或 `region-max-keys` 的值时，就会触发分裂，分裂后的信息会汇报给 PD。

12.5.4.14 TiKV 是否有类似 MySQL 的 `innodb_flush_log_trx_commit` 参数，来保证提交数据不丢失？

是的。TiKV 单机的存储引擎目前使用两个RocksDB 实例，其中一个存储raft-log。TiKV 有个 sync-log 参数，在true 的情况下，每次提交都会强制刷盘到raft-log，如果发生crash 后，通过raft-log 进行 KV 数据的恢复。

12.5.4.15 对 WAL 存储有什么推荐的硬件配置，例如 SSD，RAID 级别，RAID 卡 cache 策略，NUMA 设置，文件系统选择，操作系统的IO 调度策略等？

WAL 属于顺序写，目前我们并没有单独对他进行配置，建议 SSD。RAID 如果允许的话，最好是RAID 10，RAID 卡 cache、操作系统I/O 调度目前没有针对性的最佳实践，Linux 7 以上默认配置即可。NUMA 没有特别建议，NUMA 内存分配策略可以尝试使用 `interleave = all`，文件系统建议 ext4。

12.5.4.16 在最严格的 `sync-log = true` 数据可用模式下，写入性能如何？

一般来说，开启 `sync-log` 会让性能损耗 30% 左右。关闭 `sync-log` 时的性能表现，请参见TiDB Sysbench 性能测试报告。

12.5.4.17 是否可以利用TiKV 的Raft + 多副本达到完全的数据可靠，单机存储引擎是否需要最严格模式？

通过使用Raft 一致性算法，数据在各TiKV 节点间复制为多副本，以确保某个节点挂掉时数据的安全性。只有当数据已写入超过 50% 的副本时，应用才返回ACK（三副本中的两副本）。但理论上两个节点也可能同时发生故障，所以除非是对性能要求高于数据安全的场景，一般都强烈推荐开启 `sync-log`。

另外，还有一种 `sync-log` 的替代方案，即在Raft group 中用五个副本而非三个。这将允许两个副本同时发生故障，而仍然能保证数据安全性。

对于单机存储引擎也同样推荐打开 `sync-log` 模式。否则如果节点宕机可能会丢失最后一次写入数据。

12.5.4.18 使用Raft 协议，数据写入会有多次网络的roundtrip，实际写入延迟如何？

理论上，和单机数据库相比，数据写入会多四个网络延迟。

12.5.4.19 有没有类似 MySQL 的InnoDB Memcached plugin，可以直接使用 KV 接口，可以不需要独立的 Cache？

TiKV 支持单独进行接口调用，理论上也可以起个实例做为 Cache，但 TiDB 最大的价值是分布式关系型数据库，我们原则上不对TiKV 单独进行支持。

12.5.4.20 Coprocessor 组件的主要作用？

- 减少TiDB 与 TiKV 之间的数据传输。
- 计算下推，充分利用TiKV 的分布式计算资源。

12.5.4.21 IO error: No space left on device While appending to file

这是磁盘空间不足导致的，需要加节点或者扩大磁盘空间。

12.5.4.22 为什么TiKV 容易出现 OOM？

TiKV 的内存占用主要来自于RocksDB 的 block-cache，默认为系统总内存的 40%。当TiKV 容易出现 OOM 时，检查 `block-cache-size` 配置是否过高。还需要注意，当单机部署了多个TiKV 实例时，需要显式地配置该参数，以防止多个实例占用过多系统内存导致 OOM。

12.5.4.23 TiDB 数据和RawKV 数据可存储于同一个TiKV 集群里吗？

不可以。TiDB 数据（或使用其他事务API 生成的数据）依赖于一种特殊的键值格式，和RawKV API 数据（或其他基于RawKV 的服务生成的数据）并不兼容。

12.5.5 TiDB 测试

本小节介绍TiDB 测试中的常见问题、原因及解决方法。

12.5.5.1 TiDB Sysbench 基准测试结果如何？

很多用户在接触TiDB 都习惯做一个基准测试或者TiDB 与 MySQL 的对比测试，官方也做了一个类似测试。我们汇总很多测试结果后，发现虽然测试的数据有一定的偏差，但结论或者方向基本一致，由于TiDB 与 MySQL 由于架构上的差别非常大，很多方面是很难找到一个基准点，所以官方的建议两点：

- 大家不要用过多精力纠结这类基准测试上，应该更多关注TiDB 的场景上的区别。
- 大家可以直接参考TiDB Sysbench 性能测试报告。

12.5.5.2 TiDB 集群容量 QPS 与节点数之间关系如何，和 MySQL 对比如何？

- 在 10 节点内，TiDB 写入能力(Insert TPS) 和节点数量基本成 40% 线性递增，MySQL 由于是单节点写入，所以不具备写入扩展能力。
- MySQL 读扩容可以通过添加从库进行扩展，但写流量无法扩展，只能通过分库分表，而分库分表有很多问题，具体参考方案虽好，成本先行：数据库 Sharding+Proxy 实践解析。
- TiDB 不管是读流量、还是写流量都可以通过添加节点快速方便的进行扩展。

12.5.5.3 我们的 DBA 测试过 MySQL 性能，单台TiDB 的性能没有 MySQL 性能那么好？

TiDB 设计的目标就是针对 MySQL 单台容量限制而被迫做的分库分表的场景，或者需要强一致性和完整分布式事务的场景。它的优势是通过尽量下推到存储节点进行并行计算。对于小表（比如千万级以下），不适合 TiDB，因为数据量少，Region 有限，发挥不了并行的优势。其中最极端的例子就是计数器表，几行记录高频更新，这几行在TiDB 里，会变成存储引擎上的几个 KV，然后只落在一个 Region 里，而这个Region 只落在一个节点上。加上后台强一致性复制的开销，TiDB 引擎到TiKV 引擎的开销，最后表现出来的就是没有单个 MySQL 好。

12.5.6 TiDB 备份恢复

本小节介绍TiDB 备份恢复中的常见问题、原因及解决方法。

12.5.6.1 TiDB 主要备份方式？

目前，数据量大时（大于 1 TB）推荐使用Backup & Restore (BR) 进行备份。其他场景推荐使用Dumpling 进行备份。

尽管TiDB 也支持使用MySQL 官方工具`mysqldump` 进行数据备份和恢复，但其性能低于Dumpling，并且`mysqldump` 备份和恢复大量数据的耗费更长。

其他备份恢复相关问题，可以参考备份与恢复常见问题。

12.5.6.2 备份和恢复的速度如何？

使用BR进行备份和恢复时，备份速度大约为每个TiKV 实例 40 MB/s，恢复速度大约为每个 TiKV 实例 100 MB/s。

12.6 备份与恢复常见问题

本文列出了在使用 Backup & Restore (BR) 完成备份与恢复任务时，可能会遇到的问题及相应的解决方法。

如果遇到未包含在此文档且无法解决的问题，可以在AskTUG 社区中提问。

12.6.1 当误删除或误更新数据后，如何原地快速恢复？

从 TiDB v6.4.0 引入了完整的 Flashback 功能，可以支持原地快速恢复GC 时间内的数据到指定时间点。在误操作场景下，推荐使用 Flashback 来恢复数据，具体可以参考 Flashback 集群和 Flashback 数据库语法。

12.6.2 在TiDB v5.4.0 及后续版本中，当在有负载的集群进行备份时，备份速度为什么会变得很慢？

从 TiDB v5.4.0 起，TiKV 的备份新增了自动调节功能。对于 v5.4.0 及以上版本，该功能会默认开启。当集群负载较高时，该功能会自动限制备份任务使用的资源，从而减少备份对在线集群的性能造成的影响。如需了解关于自动调节功能的更多信息，请参见自动调节。

TiKV 支持动态配置自动调节功能。因此，在开启或关闭该功能时，你不需要重启集群。以下为该功能的具体使用方法：

- 关闭功能：把TiKV 配置项backup.enable-auto-tune 设置为 false。
- 开启功能：把 backup.enable-auto-tune 设置为 true。对于 v5.3.x 版本的集群，当TiDB 升级到 v5.4.0 及以上版本后，自动调节功能会默认关闭。这时，你可以通过该方式手动开启此功能。

如需了解通过 tikv-ctl 在线修改自动调节功能的命令行，请参阅自动调节功能的使用方法。

另外，自动调节功能减少了进行备份任务时默认使用的工作线程数量（详见backup.num-threads）。因此，你通过 Grafana 监控面板看到的备份速度、CPU 使用率、I/O 资源利用率都会小于 v5.4 之前的版本。在 v5.4.0 之前的版本中，backup.num-threads 的默认值为 CPU * 0.75，即处理备份任务的工作线程数量占了 75% 的逻辑 CPU，最大值为 32；在 v5.4.0 及之后的版本中，该配置项的默认值为 CPU * 0.5，最大值为 8。

在离线备份场景中，你也可以使用 tikv-ctl 把 backup.num-threads 修改为更大的数字，从而提升备份任务的速度。

12.6.3 PITR 问题

12.6.3.1 PITR 功能和 flashback 集群有什么区别?

从使用场景角度来看，PITR 通常用于在集群完全停止服务或数据损坏且无法使用其他方案恢复时，将集群的数据恢复到指定的时间点。使用 PITR 时，你需要通过一个新的集群来完成数据恢复。而 flashback 集群则通常用于发生误操作或其他因素导致的数据错误时，将集群的数据恢复到数据错误发生前的最近时间点。

在大多数情况下，因为 flashback 具有更短的RPO（接近零）和 RTO，flashback 比 PITR 更适合由于人为错误导致的数据错误场景。但当集群完全不可用时，flashback 集群功能也无法运行，此时 PITR 是恢复集群的唯一方案。因此，相比于 flashback，虽然 PITR 的 RPO（最长 5 分钟）和 RTO 时间较长，但 PITR 是制定数据库灾难恢复策略时必须考虑的数据安全基础。

12.6.3.2 上游数据库使用TiDB Lightning Physical 方式导入数据时，为什么无法使用日志备份功能？

目前日志备份功能还没有完全适配TiDB Lightning，导致 TiDB Lightning Physical 方式导入的数据无法备份到日志中。

在创建日志备份任务的上游集群中，请尽量避免使用TiDB Lightning Physical 方式导入数据。可以选择使用TiDB Lightning Logical 方式导入数据。若确实需要使用 Physical 导入方式，可在导入完成之后做一次快照备份操作，这样，PITR 就可以恢复到快照备份之后的时间点。

12.6.3.3 索引加速功能为什么与 PITR 功能不兼容？

Issue 链接：#38045

当前通过索引加速功能创建的索引数据无法被 PITR 备份。

因此，在 PITR 恢复完成后，BR 会将通过索引加速功能创建的索引数据删除，再重新创建。如果在日志备份期间通过索引加速功能创建的索引很多或索引数据很大，建议在创建索引后进行一次全量备份。

12.6.3.4 集群已经恢复了网络分区故障，为什么日志备份任务进度checkpoint 仍然不推进？

Issue 链接：#13126

在集群出现网络分区故障后，备份任务难以继续备份日志，并且在超过一定的重试时间后，任务会被置为 ERROR 状态。此时备份任务已经停止，需要手动执行 `br log resume` 命令来恢复日志备份任务。

12.6.3.5 执行 PITR 恢复时遇到 `execute over region id` 报错，该如何处理？

Issue 链接：#37207

该场景发生在全量数据导入时开启了日志备份，并使用 PITR 恢复全量导入时间段的日志。经过测试发现，当存在长时间（24 小时）大量热点写入，且平均单台TiKV 节点写入 OPS > 50k/s（可以通过Grafana 中 TiKV-Details -> Backup Log -> Handle Event Rate 确认该数值），那么有几率会遇到这个情况。

当前版本中建议在集群初始化后，进行一次有效快照备份，并且以此作为基础进行 PITR 恢复。

12.6.3.6 在使用 `br restore point` 命令恢复下游集群后，TiFlash 引擎数据没有恢复？

PITR 目前不支持在恢复阶段直接将数据写入TiFlash，在数据恢复完成后，br 会执行 `ALTER TABLE table_name ′→ SET TIFLASH REPLICA ***`，因此 TiFlash 副本在 PITR 完成恢复之后并不能马上可用，而是需要等待一段时间从 TiKV 节点同步数据。要查看同步进度，可以查询 `INFORMATION_SCHEMA.tiflash_replica` 表中的 `progress` 信息。

12.6.3.7 日志备份任务的 `status` 变为 ERROR，该如何处理？

在运行日志备份过程中，遇到错误后经过重试也无法恢复的故障场景，任务会被设置为 ERROR 状态，如：

```
br log status --pd x.x.x.x:2379

• Total 1 Tasks.
> #1 <
                    name: task1
```

```
                  status: ○ ERROR
                   start: 2022-07-25 13:49:02.868 +0000
                     end: 2090-11-18 14:07:45.624 +0000
                 storage: s3://tmp/br-log-backup0ef49055-5198-4be3-beab-d382a2189efb/Log
              speed(est.): 0.00 ops/s
      checkpoint[global]: 2022-07-25 14:46:50.118 +0000; gap=11h31m29s
          error[store=1]: KV:LogBackup:RaftReq
error-happen-at[store=1]: 2022-07-25 14:54:44.467 +0000; gap=11h23m35s
   error-message[store=1]: retry time exceeds: and error failed to get initial snapshot: failed to
       ′→  get the snapshot (region_id = 94812): Error during requesting raftstore: message: "
       ′→ read index not ready, reason can not read index due to merge, region 94812"
       ′→ read_index_not_ready { reason: "can not read index due to merge" region_id: 94812 }:
       ′→ failed to get initial snapshot: failed to get the snapshot (region_id = 94812): Error
       ′→ during requesting raftstore: message: "read index not ready, reason can not read index
       ′→ due to merge, region 94812" read_index_not_ready { reason: "can not read index due to
       ′→ merge" region_id: 94812 }: failed to get initial snapshot: failed to get the snapshot
       ( ′→ region_id = 94812): Error during requesting raftstore: message: "read index not
       ready, ′→ reason can not read index due to merge, region 94812" read_index_not_ready
       { reason: " ′→ can not read index due to merge" region_id: 94812 }
```

此时，你需要根据错误提示来确认故障的原因并恢复故障。确认故障恢复之后，可执行下面的命令来恢复备份任务：

```
br log resume --task-name=task1 --pd x.x.x.x:2379
```

备份任务恢复后，再次查询 `br log status`，任务状态变为正常，备份任务继续执行。

```
• Total 1 Tasks.
> #1 <
               name: task1
             status: • NORMAL
              start: 2022-07-25 13:49:02.868 +0000
                end: 2090-11-18 14:07:45.624 +0000
            storage: s3://tmp/br-log-backup0ef49055-5198-4be3-beab-d382a2189efb/Log
         speed(est.): 15509.75 ops/s
checkpoint[global]: 2022-07-25 14:46:50.118 +0000; gap=6m28s
```

注意：

由于此功能会备份集群的多版本数据，当任务发生错误且状态变更为 `ERROR` 时，同时会将当前任务的备份进度点的数据设为一个 `safe point`，`safe point` 的数据将保证 24 小时不被GC掉。所以，当任务恢复之后，会从上一个备份点继续备份日志。如果任务失败时间超过 24 小时，前一次备份进度点的数据就已经GC，此时恢复任务操作会提示失败。这种场景下，只能执行 `br log stop` 命令先停止本次任务，然后重新开启新的备份任务。

12.6.3.8 执行 br log resume 命令恢复处于暂停状态的任务时报 ErrBackupGCSafepointExceeded 错误，该如何处理？

```
Error: failed to check gc safePoint, checkpoint ts 433177834291200000: GC safepoint
    ↪ 433193092308795392 exceed TS 433177834291200000: [BR:Backup:ErrBackupGCSafepointExceeded]
    ↪ backup GC safepoint exceeded
```

暂停日志备份任务后，备份程序为了防止生成变更日志的 MVCC 数据被GC，暂停任务程序会自动将当前备份点checkpoint 设置为 service safepoint，允许保留最近 24 小时内的 MVCC 数据。当超过 24 小时后，备份点 checkpoint 的 MVCC 数据已经被GC，此时程序会拒绝恢复备份任务。

此场景的处理办法是：先执行 br log stop 命令来删除当前的任务，然后执行 br log start 重新创建新的日志备份任务，同时做一个全量备份，便于后续做 PITR 恢复操作。

12.6.4 功能兼容性问题

12.6.4.1 为什么 BR 恢复的数据无法同步到TiCDC / Drainer 的上游集群？

- BR 恢复的数据无法被同步到下游，因为恢复时 BR 直接导入 SST 文件，而下游集群目前没有办法获得上游的 SST 文件。
- 在 4.0.3 版本之前，恢复时产生的 DDL jobs 还可能会让TiCDC / Drainer 执行异常的 DDL。所以，如果一定要在 TiCDC / Drainer 的上游集群执行恢复，请将 BR 恢复的所有表加入TiCDC / Drainer 的阻止名单。

TiCDC 可以通过配置项中的 filter.rules 项完成，Drainer 则可以通过syncer.ignore-table 完成。

12.6.4.2 恢复时为什么会报 new_collations_enabled_on_first_bootstrap 不匹配？

从 TiDB v6.0.0 版本开始，new_collations_enabled_on_first_bootstrap 配置项的默认值由 false 改为 true。BR 会备份上游集群的 new_collations_enabled_on_first_bootstrap 配置项。当上下游集群的此项配置相同时，BR 才会将上游集群的备份数据安全地恢复到下游集群中。若上下游的该配置不相同，BR 会拒绝恢复，并报此配置项不匹配的错误。

如果需要将旧版本的备份数据恢复到TiDB v6.0.0 或更新版本的TiDB 集群中，你需要检查上下游集群中的该配置项是否相同：

- 若该配置项相同，则可在恢复命令中添加 --check-requirements=false 以跳过此项配置检查。
- 若该配置项不相同，且进行强行恢复，BR 会报数据校验错误。

12.6.4.3 恢复 Placement Rule 到集群时为什么会报错？

BR 在 v6.0.0 之前不支持放置规则，在 v6.0.0 及以上版本开始支持并提供了命令行选项 --with-tidb-placement-mode=strict/ignore 来控制放置规则的导入模式。默认值为 strict，代表导入并检查放置规则，当--with-tidb-placement-mode 设置为 ignore 时，BR 会忽略所有的放置规则。

12.6.5 进行数据恢复的问题

12.6.5.1 使用 BR 时遇到错误信息 `Io(Os...)`，该如何处理？

这类问题几乎都是 TiKV 在写盘的时候遇到的系统调用错误。例如遇到 `Io(Os { code: 13, kind: PermissionDenied...})` 或者 `Io(Os { code: 2, kind: NotFound...})`。

首先检查备份目录的挂载方式和文件系统，尝试备份到其它文件夹或者硬盘。

目前已知备份到 samba 搭建的网盘时可能会遇到 `Code: 22(invalid argument)` 错误。

12.6.5.2 恢复时遇到错误信息 `rpc error: code = Unavailable desc =...`，该如何处理？

该问题一般是因为恢复数据的时候，恢复集群的性能不足导致的。可以从恢复集群的监控或者TiKV 日志来辅助确认。

要解决这类问题，可以尝试扩大集群资源，以及调小恢复时的并发度(concurrency)，打开限速(ratelimit) 设置。

12.6.5.3 恢复备份数据时，BR 会报错 `entry too large, the max entry size is 6291456, the size of data is 7690800`

你可以尝试降低并发批量建表的大小，将 `--ddl-batch-size` 设置为 128 或者更小的值。

在`--ddl-batch-size` 的值大于 1 的情况下，使用 BR 恢复数据时，TiDB 会把执行创建表任务的 DDL job 队列写到 TiKV 上。由于TiDB 能够一次性发送的job message 的最大值默认为 6 MB（不建议修改此值，具体内容，参考txn-entry-size-limit 和raft-entry-max-size），TiDB 单次发送的所有表的 schema 大小总和也不应该超过 6 MB。因此，如果你设置的 `--ddl-batch-size` 的值过大，TiDB 单次发送的批量表的 schema 大小就会超出规定值，从而导致 BR 报 `entry too large, the max entry size is 6291456, the size of data is 7690800` 错误。

12.6.5.4 使用local storage 的时候，备份的文件会存在哪里？

> 注意：
>
> 如果没有挂载 NFS 到br 工具或TiKV 节点，或者使用了支持 S3、GCS 或Azure Blob Storage 协议的远端存储，那么br 工具备份的数据会在各个TiKV 节点生成。注意这不是推荐的备份和恢复使用方式，因为备份数据会分散在各个节点的本地文件系统中，聚集这些备份数据可能会造成数据冗余和运维上的麻烦，而且在不聚集这些数据便直接恢复的时候会遇到 `SST file not found` 报错。

在使用local storage 的时候，会在运行br 命令行工具的节点生成 `backupmeta`，在各个 Region 的 Leader 所在TiKV 节点生成备份文件。

12.6.5.5 恢复的时候，报错 `could not read local://...:download sst failed`，该如何处理？

在恢复的时候，每个节点都必须能够访问到所有的备份文件(SST files)。默认情况下，假如使用 local storage，备份文件会分散在各个节点中，此时是无法直接恢复的，必须将每个TiKV 节点的备份文件拷贝到其它所有TiKV 节点才能恢复。所以建议使用 S3/GCS/Azure Blob Storage/NFS 作为备份存储。

12.6.5.6 遇到 Permission denied 或者 No such file or directory 错误，即使用root 运行 br 命令行工具也无法解决，该如何处理？

需要确认TiKV 是否有访问备份目录的权限。如果是备份，确认是否有写权限；如果是恢复，确认是否有读权限。

在进行备份操作时，如果使用本地磁盘或 NFS 作为存储介质，请确保执行br 命令行工具的用户和启动TiKV 的用户相同（如果br 工具和TiKV 位于不同的机器，则需要用户的 UID 相同），否则备份可能会出现该问题。

使用root 运行br 命令行工具仍旧有可能会因为磁盘权限而失败，因为备份文件(SST) 的保存是由TiKV 执行的。

> **注意：**
>
> 在恢复的时候也可能遇到同样的问题。
>
> 使用 br 命令行工具进行数据的恢复时，检验读权限的时机是在第一次读取 SST 文件时，考虑到执行 DDL 的耗时，这个时刻可能会离开始运行br 命令行工具的时间很远。这样可能会出现等了很长时间之后遇到 Permission denied 错误失败的情况。
>
> 因此，最好在恢复前提前检查权限。

你可以按照如下步骤进行权限检查：

1. 执行 Linux 原生的进程查询命令

```
ps aux | grep tikv-server
```

命令输出示例如下:

```
tidb_ouo  9235 10.9  3.8 2019248 622776 ?      Ssl  08:28   1:12 bin/tikv-server --addr
    '→ 0.0.0.0:20162 --advertise-addr 172.16.6.118:20162 --status-addr 0.0.0.0:20188 --
    '→ advertise-status-addr 172.16.6.118:20188 --pd 172.16.6.118:2379 --data-dir /home/
    '→ user1/tidb-data/tikv-20162 --config conf/tikv.toml --log-file /home/user1/tidb-
    '→ deploy/tikv-20162/log/tikv.log
tidb_ouo  9236  9.8  3.8 2048940 631136 ?      Ssl  08:28   1:05 bin/tikv-server --addr
    '→ 0.0.0.0:20161 --advertise-addr 172.16.6.118:20161 --status-addr 0.0.0.0:20189 --
    '→ advertise-status-addr 172.16.6.118:20189 --pd 172.16.6.118:2379 --data-dir /home/
    '→ user1/tidb-data/tikv-20161 --config conf/tikv.toml --log-file /home/user1/tidb-
    '→ deploy/tikv-20161/log/tikv.log
```

或者执行以下命令：

```
ps aux | grep tikv-server | awk '{print $1}'
```

命令输出示例如下:

```
tidb_ouo
tidb_ouo
```

2. 使用TiUP 命令查询集群的启动信息

```
tiup cluster list
```

命令输出示例如下：

```
[root@Copy-of-VM-EE-CentOS76-v1 br]# tiup cluster list
Starting component `cluster`: /root/.tiup/components/cluster/v1.5.2/tiup-cluster list
Name          User      Version  Path
    ↪ PrivateKey
----          ----      -------  ----
    ↪ ----------
tidb_cluster  tidb_ouo  v5.0.2   /root/.tiup/storage/cluster/clusters/tidb_cluster  /root/.
    ↪ tiup/storage/cluster/clusters/tidb_cluster/ssh/id_rsa
```

3. 检查备份目录的权限，例如 backup 目录是备份数据存储目录。命令示例如下：

```
ls -al backup
```

命令输出示例如下：

```
[root@Copy-of-VM-EE-CentOS76-v1 user1]# ls -al backup
total 0
drwxr-xr-x  2 root root   6 Jun 28 17:48 .
drwxr-xr-x 11 root root 310 Jul  4 10:35 ..
```

由以上命令输出结果可知，`tikv-server` 实例由用户 `tidb_ouo` 启动，但该账号没有 `backup` 目录的写入权限，所以备份失败。

12.6.5.7 恢复集群的时候，在 MySQL 下的业务表为什么没有恢复？

自 br v5.1.0 开始，全量备份会备份 mysql schema 下的表。br v6.2.0 以前的版本，在默认设置不会恢复 mysql schema 下的表。

如果需要恢复 `mysql` 下的用户创建的表（非系统表），可以通过table filter 来显式地包含目标表。以下示例中命令会在执行正常的恢复的同时恢复 `mysql.usertable`。

```
br restore full -f '*.*' -f '!mysql.*' -f 'mysql.usertable' -s $external_storage_url --with-sys-
    ↪ table
```

在上面的命令中，

- `-f '*.*'` 用于覆盖掉默认的规则。
- `-f '!mysql.*'` 指示 BR 不要恢复 `mysql` 中的表，除非另有指定。
- `-f 'mysql.usertable'` 则指定需要恢复 `mysql.usertable`。

如果只需要恢复 `mysql.usertable`，而无需恢复其他表，可以使用以下命令：

```
br restore full -f 'mysql.usertable' -s $external_storage_url --with-sys-table
```

此外请注意，即使设置了table filter，BR 也不会恢复以下系统表：

- 统计信息表（mysql.stat_*）
- 系统变量表（mysql.tidb、mysql.global_variables）
- 其他系统表

12.6.6 备份恢复功能相关知识

12.6.6.1 备份数据有多大，备份会有副本吗？

备份的时候仅仅在每个Region 的 Leader 处生成该Region 的备份文件。因此备份的大小等于数据大小，不会有多余的副本数据。所以最终的总大小大约是TiKV 数据总量除以副本数。

但是假如想要从本地恢复数据，因为每个TiKV 都必须要能访问到所有备份文件，在最终恢复的时候会有等同于恢复时TiKV 节点数量的副本。

12.6.6.2 BR 备份恢复后，为什么监控显示磁盘使用空间不一致？

这个情况多数是因为备份时集群的数据压缩比率和恢复时的默认值不一致导致的，只要恢复的checksum 阶段顺利通过，可以忽略这个问题，不影响正常使用。

12.6.6.3 使用 BR 恢复数据后是否需要对表执行 ANALYZE 以更新TiDB 在表和索引上留下的统计信息？

BR 不会备份统计信息（v4.0.9 除外）。所以在恢复存档后需要手动执行 ANALYZE TABLE 或等待TiDB 自动进行 ANALYZE。

BR v4.0.9 备份统计信息使br 工具消耗过多内存，为保证备份过程正常，从 v4.0.10 开始默认关闭备份统计信息的功能。

如果不对表执行 ANALYZE，TiDB 会因统计信息不准确而选不中最优化的执行计划。如果查询性能不是重点关注项，可以忽略 ANALYZE。

12.6.6.4 可以同时启动多个恢复任务对单个集群进行恢复吗？

强烈不建议在单个集群中同时启动多个恢复任务进行数据恢复，原因如下：

- BR 在恢复数据时，会修改 PD 的一些全局配置。如果同时使用多个 BR 命令进行恢复，这些配置可能会被错误地覆写，导致集群状态异常。
- 因为 BR 在恢复数据的时候会占用大量集群资源，事实上并行恢复能获得的速度提升也非常有限。
- 多个恢复任务同时进行的场景没有经过测试，无法保证成功。

12.6.6.5 BR 会备份表的 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 信息吗？恢复出来的表会有多个Region 吗？

会，BR 会备份表的SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 信息，并恢复成多个Region。

13 术语表

13.1 A

13.1.1 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。

13.2 B

13.2.1 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。

13.2.2 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。

13.2.3 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。

13.2.4 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。

13.3 C

13.3.1 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。

13.3.2 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。

13.3.3 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。

13.4 D

13.4.1 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。

13.5 I

13.5.1 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。

13.5.2 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。

13.6 L

13.6.1 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。

13.7 O

13.7.1 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”。

13.7.2 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。

13.7.3 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

13.8 P

13.8.1 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。

13.8.2 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。

13.8.3 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。

13.8.4 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。

13.9 Q

13.9.1 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。

13.10 R

13.10.1 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。

13.10.2 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。

13.10.3 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。

13.10.4 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。

13.11 S

13.11.1 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)

13.11.2 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。

13.12 T

13.12.1 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。

13.12.2 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。

13.12.3 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。