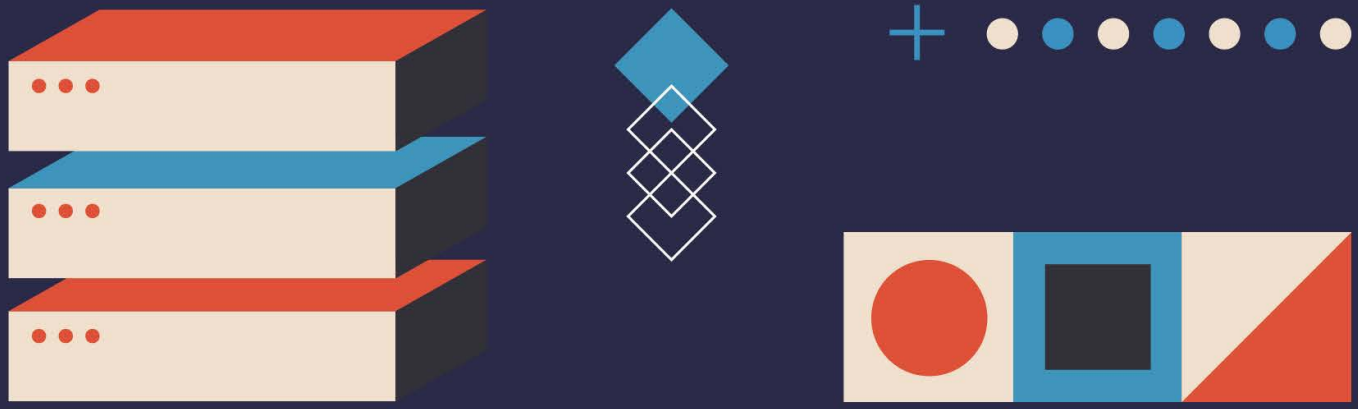
TiDB

TiDB 用户指南（企业版）

产品版本： v6.1 | 文档版本： 20220613



PingCAP

北京平凯星辰科技发展有限公司

目录

1 关于 TiDB 27

1.1 TiDB 简介 · · · · · 27

1.1.1 五大核心特性 · · · · · 27

1.1.2 四大核心应用场景 · · · · · 27

1.1.3 另请参阅 · · · · · 28

1.2 TiDB v6.1.0 Release Notes · · · · · 28

1.2.1 新功能 · · · · · 29

1.2.2 兼容性变更 · · · · · 33

1.2.3 改进提升 · · · · · 51

1.2.4 错误修复 · · · · · 51

1.3 TiDB 软件和硬件环境建议配置 · · · · · 53

1.3.1 Linux 操作系统版本要求 · · · · · 53

1.3.2 软件配置要求 · · · · · 54

1.3.3 服务器建议配置 · · · · · 54

1.3.4 网络要求 · · · · · 56

1.3.5 磁盘空间要求 · · · · · 56

1.3.6 客户端 Web 浏览器要求 · · · · · 57

1.4 与 MySQL 兼容性对比 · · · · · 57

1.4.1 不支持的功能特性 · · · · · 57

1.4.2 与 MySQL 有差异的特性详细说明 · · · · · 58

1.5 使用限制 · · · · · 62
1.5.1 标识符长度限制 · · · · · 62
1.5.2 Databases、Tables、Views、Connections 总个数限制 · · · · · 62
1.5.3 单个 Database 的限制 · · · · · 63
1.5.4 单个 Table 的限制 · · · · · 63
1.5.5 单行的限制 · · · · · 63
1.5.6 单列的限制 · · · · · 63
1.5.7 字符串类型限制 · · · · · 63
1.5.8 SQL Statements 的限制 · · · · · 64
2 SQL 参考指南 64
2.1 SQL 语言结构和语法 · · · · · 64
2.1.1 属性 · · · · · 64
2.1.2 字面值 · · · · · 73
2.1.3 Schema 对象名 · · · · · 78
2.1.4 关键字 · · · · · 79
2.1.5 表达式语法 (Expression Syntax) · · · · · 94
2.1.6 注释语法 · · · · · 95
2.2 SQL 语句 · · · · · 97
2.2.1 ADD COLUMN · · · · · 97
2.2.2 ADD INDEX · · · · · 100
2.2.3 ADMIN · · · · · 102
2.2.4 ADMIN CANCEL DDL · · · · · 107
2.2.5 ADMIN CHECKSUM TABLE · · · · · 108
2.2.6 ADMIN CHECK [TABLE|INDEX] · · · · · 109
2.2.7 ADMIN SHOW DDL [JOBS|QUERIES] · · · · · 110
2.2.8 ADMIN SHOW TELEMETRY · · · · · 113
2.2.9 ALTER DATABASE · · · · · 123
2.2.10 ALTER INDEX · · · · · 123
2.2.11 ALTER INSTANCE · · · · · 126
2.2.12 ALTER PLACEMENT POLICY · · · · · 127
2.2.13 ALTER TABLE · · · · · 129
2.2.14 ALTER USER · · · · · 134

2.2.15 ANALYZE · · · · · 136
2.2.16 BATCH · · · · · 138
2.2.17 BEGIN · · · · · 139
2.2.18 CHANGE COLUMN · · · · · 140
2.2.19 CHANGE DRAINER · · · · · 143
2.2.20 CHANGE PUMP · · · · · 144
2.2.21 COMMIT · · · · · 145
2.2.22 CREATE [GLOBAL|SESSION] BINDING · · · · · 146
2.2.23 CREATE DATABASE · · · · · 150
2.2.24 CREATE PLACEMENT POLICY · · · · · 152
2.2.25 CREATE ROLE · · · · · 153
2.2.26 CREATE SEQUENCE · · · · · 156
2.2.27 CREATE TABLE LIKE · · · · · 164
2.2.28 CREATE TABLE · · · · · 166
2.2.29 CREATE USER · · · · · 170
2.2.30 CREATE VIEW · · · · · 172
2.2.31 DEALLOCATE · · · · · 174
2.2.32 DELETE · · · · · 175
2.2.33 DESC · · · · · 177
2.2.34 DESCRIBE · · · · · 177
2.2.35 DO · · · · · 177
2.2.36 DROP [GLOBAL|SESSION] BINDING · · · · · 178
2.2.37 DROP COLUMN · · · · · 182
2.2.38 DROP DATABASE · · · · · 184
2.2.39 DROP INDEX · · · · · 185
2.2.40 DROP PLACEMENT POLICY · · · · · 187
2.2.41 DROP ROLE · · · · · 188
2.2.42 DROP SEQUENCE · · · · · 191
2.2.43 DROP STATS · · · · · 192
2.2.44 DROP TABLE · · · · · 193
2.2.45 DROP USER · · · · · 195
2.2.46 DROP VIEW · · · · · 196

2.2.47 EXECUTE · · · · · 198
2.2.48 EXPLAIN ANALYZE · · · · · 199
2.2.49 EXPLAIN · · · · · 205
2.2.50 FLASHBACK TABLE · · · · · 212
2.2.51 FLUSH PRIVILEGES · · · · · 214
2.2.52 FLUSH STATUS · · · · · 215
2.2.53 FLUSH TABLES · · · · · 216
2.2.54 GRANT <privileges> · · · · · 218
2.2.55 GRANT <role> · · · · · 220
2.2.56 INSERT · · · · · 222
2.2.57 KILL · · · · · 224
2.2.58 LOAD DATA · · · · · 226
2.2.59 LOAD STATS · · · · · 228
2.2.60 MODIFY COLUMN · · · · · 229
2.2.61 PREPARE · · · · · 232
2.2.62 RECOVER TABLE · · · · · 233
2.2.63 RENAME INDEX · · · · · 236
2.2.64 RENAME TABLE · · · · · 237
2.2.65 REPLACE · · · · · 238
2.2.66 RESTORE · · · · · 240
2.2.67 REVOKE <privileges> · · · · · 243
2.2.68 REVOKE <role> · · · · · 245
2.2.69 ROLLBACK · · · · · 248
2.2.70 SELECT · · · · · 249
2.2.71 SET DEFAULT ROLE · · · · · 255
2.2.72 SET [NAMES|CHARACTER SET] · · · · · 258
2.2.73 SET PASSWORD · · · · · 261
2.2.74 SET ROLE · · · · · 264
2.2.75 SET TRANSACTION · · · · · 266
2.2.76 SET [GLOBAL|SESSION] <variable> · · · · · 268
2.2.77 SHOW [BACKUPS|RESTORES] · · · · · 272
2.2.78 SHOW ANALYZE STATUS · · · · · 273

2.2.79 SHOW [GLOBAL|SESSION] BINDINGS · · · · · 275
2.2.80 SHOW BUILTINS · · · · · 282
2.2.81 SHOW CHARACTER SET · · · · · 288
2.2.82 SHOW COLLATION · · · · · 289
2.2.83 SHOW [FULL] COLUMNS FROM · · · · · 291
2.2.84 SHOW CONFIG · · · · · 296
2.2.85 SHOW CREATE PLACEMENT POLICY · · · · · 300
2.2.86 SHOW CREATE SEQUENCE · · · · · 301
2.2.87 SHOW CREATE TABLE · · · · · 302
2.2.88 SHOW CREATE USER · · · · · 303
2.2.89 SHOW DATABASES · · · · · 305
2.2.90 SHOW DRAINER STATUS · · · · · 306
2.2.91 SHOW ENGINES · · · · · 307
2.2.92 SHOW ERRORS · · · · · 308
2.2.93 SHOW [FULL] FIELDS FROM · · · · · 309
2.2.94 SHOW GRANTS · · · · · 309
2.2.95 SHOW INDEX [FROM|IN] · · · · · 311
2.2.96 SHOW INDEXES [FROM|IN] · · · · · 311
2.2.97 SHOW KEYS [FROM|IN] · · · · · 314
2.2.98 SHOW MASTER STATUS · · · · · 314
2.2.99 SHOW PLACEMENT · · · · · 315
2.2.100 SHOW PLACEMENT FOR · · · · · 316
2.2.101 SHOW PLACEMENT LABELS · · · · · 318
2.2.102 SHOW PLUGINS · · · · · 319
2.2.103 SHOW PRIVILEGES · · · · · 321
2.2.104 SHOW [FULL] PROCESSLIST · · · · · 323
2.2.105 SHOW PROFILES · · · · · 324
2.2.106 SHOW PUMP STATUS · · · · · 325
2.2.107 SHOW SCHEMAS · · · · · 325
2.2.108 SHOW STATS_HEALTHY · · · · · 326
2.2.109 SHOW STATS_HISTOGRAMS · · · · · 329
2.2.110 SHOW STATS_META · · · · · 332

2.2.111 SHOW [GLOBAL|SESSION] STATUS · · · · · 335
2.2.112 SHOW TABLE NEXT_ROW_ID · · · · · 339
2.2.113 SHOW TABLE REGIONS · · · · · 340
2.2.114 SHOW TABLE STATUS · · · · · 346
2.2.115 SHOW [FULL] TABLES · · · · · 348
2.2.116 SHOW [GLOBAL|SESSION] VARIABLES · · · · · 351
2.2.117 SHOW WARNINGS · · · · · 356
2.2.118 SHUTDOWN · · · · · 358
2.2.119 Split Region 使用文档 · · · · · 359
2.2.120 START TRANSACTION · · · · · 368
2.2.121 TABLE · · · · · 369
2.2.122 TRACE · · · · · 370
2.2.123 TRUNCATE · · · · · 374
2.2.124 UPDATE · · · · · 376
2.2.125 USE · · · · · 379
2.2.126 WITH · · · · · 381
2.3 数据类型 · · · · · 383
2.3.1 数据类型概述 · · · · · 383
2.3.2 数据类型的默认值 · · · · · 383
2.3.3 数值类型 · · · · · 383
2.3.4 日期和时间类型 · · · · · 387
2.3.5 字符串类型 · · · · · 393
2.4 函数与操作符 · · · · · 396
2.4.1 函数和操作符概述 · · · · · 396
2.4.2 表达式求值的类型转换 · · · · · 396
2.4.3 操作符 · · · · · 397
2.4.4 控制流程函数 · · · · · 399
2.4.5 字符串函数 · · · · · 399
2.4.6 数值函数与操作符 · · · · · 401
2.4.7 日期和时间函数 · · · · · 402
2.4.8 位函数和操作符 · · · · · 404
2.4.9 Cast 函数和操作符 · · · · · 404

2.4.10 加密和压缩函数 · · · · · 405
2.4.11 锁函数 · · · · · 406
2.4.12 信息函数 · · · · · 406
2.4.13 GROUP BY 聚合函数 · · · · · 407
2.4.14 窗口函数 · · · · · 410
2.4.15 其他函数 · · · · · 411
2.4.16 精度数学 · · · · · 411
2.4.17 集合运算 · · · · · 415
2.4.18 下推到 TiKV 的表达式列表 · · · · · 417
2.4.19 TiDB 特有的函数 · · · · · 423
2.5 聚簇索引 · · · · · 431
2.5.1 使用场景 · · · · · 432
2.5.2 使用方法 · · · · · 432
2.5.3 限制 · · · · · 435
2.5.4 兼容性 · · · · · 435
2.6 约束 · · · · · 437
2.6.1 非空约束 · · · · · 437
2.6.2 CHECK 约束 · · · · · 437
2.6.3 唯一约束 · · · · · 438
2.6.4 主键约束 · · · · · 439
2.6.5 外键约束 · · · · · 440
2.7 SQL 模式 · · · · · 441
2.7.1 重要的 sql_mode 值 · · · · · 441
2.7.2 SQL mode 列表，如下 · · · · · 441
2.8 表属性 · · · · · 461
2.8.1 使用方法 · · · · · 462
2.8.2 覆盖关系 · · · · · 462
2.8.3 使用表属性控制 Region 合并 · · · · · 463
2.9 事务 · · · · · 464
2.9.1 TiDB 事务概览 · · · · · 464
2.9.2 TiDB 事务隔离级别 · · · · · 473
2.9.3 TiDB 乐观事务模型 · · · · · 475
2.9.4 TiDB 悲观事务模式 · · · · · 479
2.9.5 非事务 DML 语句 · · · · · 486

2.10 垃圾回收 (GC) · · · · · 496
2.10.1 GC 机制简介 · · · · · 496
2.10.2 GC 配置 · · · · · 498
2.11 视图 · · · · · 500
2.11.1 查询视图 · · · · · 500
2.11.2 查看视图的相关信息 · · · · · 500
2.11.3 示例 · · · · · 503
2.11.4 局限性 · · · · · 504
2.11.5 扩展阅读 · · · · · 504
2.12 分区表 · · · · · 504
2.12.1 分区类型 · · · · · 504
2.12.2 分区管理 · · · · · 514
2.12.3 分区裁剪 · · · · · 516
2.12.4 分区选择 · · · · · 518
2.12.5 分区的约束和限制 · · · · · 520
2.13 临时表 · · · · · 533
2.13.1 使用场景 · · · · · 533
2.13.2 临时表类型 · · · · · 533
2.13.3 本地临时表 · · · · · 534
2.13.4 全局临时表 · · · · · 536
2.13.5 限制临时表的内存占用 · · · · · 538
2.13.6 与其他 TiDB 功能的兼容性限制 · · · · · 538
2.13.7 TiDB 数据迁移工具支持 · · · · · 538
2.13.8 另请参阅 · · · · · 539
2.14 缓存表 · · · · · 539
2.14.1 使用场景 · · · · · 539
2.14.2 使用示例 · · · · · 539
2.14.3 缓存表大小限制 · · · · · 543
2.14.4 与其他 TiDB 功能的兼容性限制 · · · · · 543
2.14.5 TiDB 数据迁移工具兼容性 · · · · · 544
2.14.6 另请参阅 · · · · · 544

2.15 字符集和排序 · · · · · 544
2.15.1 字符集和排序规则 · · · · · 544
2.15.2 GBK · · · · · 553
2.16 Placement Rules in SQL · · · · · 555
2.16.1 指定放置规则 · · · · · 556
2.16.2 查看放置规则 · · · · · 556
2.16.3 放置选项参考 · · · · · 557
2.16.4 示例 · · · · · 558
2.16.5 工具兼容性 · · · · · 561
2.16.6 使用限制 · · · · · 561
2.17 系统表 · · · · · 561
2.17.1 mysql Schema · · · · · 561
2.17.2 INFORMATION_SCHEMA · · · · · 562
2.17.3 Metrics Schema · · · · · 649
3 账户与安全 655
3.1 权限管理 · · · · · 655
3.1.1 与 MySQL 安全特性差异 · · · · · 655
3.1.2 权限管理 · · · · · 656
3.1.3 TiDB 用户账户管理 · · · · · 665
3.1.4 基于角色的访问控制 · · · · · 668
3.1.5 TiDB 证书鉴权使用指南 · · · · · 674
3.2 安全加固 · · · · · 683
3.2.1 为 TiDB 客户端服务端间通信开启加密传输 · · · · · 683
3.2.2 为 TiDB 组件间通信开启加密传输 · · · · · 687
3.2.3 生成自签名证书 · · · · · 690
3.2.4 静态加密 · · · · · 693
3.2.5 为 TiDB 落盘文件开启加密 · · · · · 698
3.2.6 日志脱敏 · · · · · 698
4 使用插件 699

4.1 TiDB 审计插件 · · · · · 699
4.1.1 获取插件 · · · · · 699
4.1.2 部署审计插件 · · · · · 700
4.1.3 查看状态 · · · · · 703
4.1.4 启用和禁用插件 · · · · · 704
4.1.5 修改 TABLE_ACCESS 相关配置 · · · · · 704
4.1.6 使用插件输出审计日志 · · · · · 706
4.1.7 日志输出 · · · · · 707
4.1.8 清除审计日志 · · · · · 709
4.2 TiDB 白名单插件 · · · · · 709
4.2.1 获取插件 · · · · · 709
4.2.2 部署插件 · · · · · 709
4.2.3 查看状态 · · · · · 711
4.2.4 启用和禁用插件 · · · · · 712
4.2.5 使用和验证插件 · · · · · 712
5 命令行工具 713
5.1 TiKV Control 使用说明 · · · · · 713
5.1.1 通过 TiUP 使用 TiKV Control · · · · · 714
5.1.2 通用参数 · · · · · 715
5.1.3 各项子命令及部分参数、选项 · · · · · 716
5.2 PD Control 使用说明 · · · · · 726
5.2.1 安装方式 · · · · · 726
5.2.2 简单例子 · · · · · 727
5.2.3 命令行参数 (flags) · · · · · 727
5.2.4 命令 (command) · · · · · 728
5.2.5 jq 格式化 JSON 输出示例 · · · · · 748
5.3 TiDB Control 使用说明 · · · · · 750
5.3.1 获取 TiDB Control · · · · · 750
5.3.2 使用介绍 · · · · · 751
5.4 PD Recover 使用文档 · · · · · 757
5.4.1 安装 PD Recover · · · · · 757
5.4.2 快速开始 · · · · · 757
5.4.3 常见问题 · · · · · 759

6 命令行参数 759

6.1 TiDB 配置参数 759

6.1.1 --advertise-address 759

6.1.2 --config 759

6.1.3 --config-check 759

6.1.4 --config-strict 760

6.1.5 --cors 760

6.1.6 --host 760

6.1.7 --enable-binlog 760

6.1.8 -L 760

6.1.9 --lease 760

6.1.10 --log-file 760

6.1.11 --log-slow-query 760

6.1.12 --metrics-addr 761

6.1.13 --metrics-interval 761

6.1.14 -P 761

6.1.15 --path 761

6.1.16 --proxy-protocol-networks 761

6.1.17 --proxy-protocol-header-timeout 762

6.1.18 --report-status 762

6.1.19 --run-ddl 762

6.1.20 --socket string 762

6.1.21 --status 762

6.1.22 --status-host 762

6.1.23 --store 763

6.1.24 --token-limit 763

6.1.25 -V 763

6.1.26 --plugin-dir 763

6.1.27 --plugin-load 763

6.1.28 --affinity-cpus 763

6.1.29 --repair-mode 763

6.1.30 --repair-list 763

6.2 TiKV 配置参数 · · · · · 764
6.2.1 -A, --addr · · · · · 764
6.2.2 --advertise-addr · · · · · 764
6.2.3 --status-addr · · · · · 764
6.2.4 --advertise-status-addr · · · · · 764
6.2.5 -C, --config · · · · · 764
6.2.6 --capacity · · · · · 765
6.2.7 --data-dir · · · · · 765
6.2.8 -L · · · · · 765
6.2.9 --log-file · · · · · 765
6.2.10 --pd · · · · · 765
6.3 TiFlash 命令行参数 · · · · · 765
6.3.1 server --config-file · · · · · 765
6.3.2 dttool migrate · · · · · 766
6.3.3 dttool bench · · · · · 767
6.3.4 dttool inspect · · · · · 767
6.4 PD 配置参数 · · · · · 767
6.4.1 --advertise-client-urls · · · · · 767
6.4.2 --advertise-peer-urls · · · · · 768
6.4.3 --client-urls · · · · · 768
6.4.4 --peer-urls · · · · · 768
6.4.5 --config · · · · · 768
6.4.6 --data-dir · · · · · 768
6.4.7 --initial-cluster · · · · · 768
6.4.8 --join · · · · · 769
6.4.9 -L · · · · · 769
6.4.10 --log-file · · · · · 769
6.4.11 --log-rotate · · · · · 769
6.4.12 --name · · · · · 769
6.4.13 --cacert · · · · · 769
6.4.14 --cert · · · · · 769
6.4.15 --key · · · · · 770

6.4.16 --metrics-addr 770
6.4.17 --force-new-cluster 770
6.4.18 -V, --version 770

7 配置文件参数 770

7.1 TiDB 配置文件描述 770
7.1.1 log 774
7.1.2 log.file 776
7.1.3 security 777
7.1.4 performance 778
7.1.5 opentracing 782
7.1.6 opentracing.sampler 783
7.1.7 opentracing.reporter 783
7.1.8 tikv-client 784
7.1.9 tikv-client.copr-cache 从 v4.0.0 版本开始引入 785
7.1.10 binlog 786
7.1.11 status 786
7.1.12 pessimistic-txn 787
7.1.13 experimental 788
7.2 TiKV 配置文件描述 788
7.2.1 全局配置 788
7.2.2 log 从 v5.4.0 版本开始引入 789
7.2.3 log.file 从 v5.4.0 版本开始引入 789
7.2.4 server 790
7.2.5 readpool.unified 793
7.2.6 readpool.storage 793
7.2.7 readpool.coprocessor 795
7.2.8 storage 796
7.2.9 storage.block-cache 798
7.2.10 storage.flow-control 799
7.2.11 storage.io-rate-limit 800
7.2.12 raftstore 800
7.2.13 coprocessor 809

7.2.14 rocksdb · · · · · 811
7.2.15 rocksdb.titan · · · · · 815
7.2.16 rocksdb.defaultcf | rocksdb.writecf | rocksdb.lockcf · · · · · 816
7.2.17 rocksdb.defaultcf.titan · · · · · 821
7.2.18 raftdb · · · · · 823
7.2.19 raft-engine · · · · · 823
7.2.20 security · · · · · 825
7.2.21 security.encryption · · · · · 826
7.2.22 import · · · · · 826
7.2.23 gc · · · · · 827
7.2.24 backup · · · · · 827
7.2.25 cdc · · · · · 827
7.2.26 resolved-ts · · · · · 828
7.2.27 pessimistic-txn · · · · · 828
7.2.28 quota · · · · · 829
7.2.29 causal-ts 从 v6.1.0 版本开始引入 · · · · · 830
7.3 TiFlash 配置参数 · · · · · 831
7.3.1 PD 调度参数 · · · · · 831
7.3.2 TiFlash 配置参数 · · · · · 832
7.4 PD 配置文件描述 · · · · · 837
7.4.1 security · · · · · 839
7.4.2 log · · · · · 840
7.4.3 log.file · · · · · 841
7.4.4 metric · · · · · 841
7.4.5 schedule · · · · · 841
7.4.6 replication · · · · · 844
7.4.7 label-property · · · · · 846
7.4.8 dashboard · · · · · 846
7.5 在线修改集群配置 · · · · · 847
7.5.1 常用操作 · · · · · 847
8 系统变量 875

8.1 变量参考 · · · · · 876
8.1.1 allow_auto_random_explicit_insert 从 v4.0.3 版本开始引入 · · · · · 876
8.1.2 auto_increment_increment · · · · · 877
8.1.3 auto_increment_offset · · · · · 877
8.1.4 autocommit · · · · · 878
8.1.5 character_set_client · · · · · 878
8.1.6 character_set_connection · · · · · 878
8.1.7 character_set_database · · · · · 878
8.1.8 character_set_results · · · · · 878
8.1.9 character_set_server · · · · · 878
8.1.10 cte_max_recursion_depth · · · · · 879
8.1.11 datadir · · · · · 879
8.1.12 ddl_slow_threshold · · · · · 879
8.1.13 default_authentication_plugin · · · · · 879
8.1.14 foreign_key_checks · · · · · 879
8.1.15 hostname · · · · · 879
8.1.16 init_connect · · · · · 880
8.1.17 innodb_lock_wait_timeout · · · · · 880
8.1.18 interactive_timeout · · · · · 880
8.1.19 last_plan_from_binding 从 v4.0 版本开始引入 · · · · · 880
8.1.20 last_plan_from_cache 从 v4.0 版本开始引入 · · · · · 880
8.1.21 license · · · · · 880
8.1.22 max_execution_time · · · · · 881
8.1.23 plugin_dir · · · · · 881
8.1.24 plugin_load · · · · · 881
8.1.25 port · · · · · 881
8.1.26 rand_seed1 · · · · · 881
8.1.27 rand_seed2 · · · · · 882
8.1.28 require_secure_transport 从 v6.1.0 版本开始引入 · · · · · 882
8.1.29 skip_name_resolve 从 v5.2.0 版本开始引入 · · · · · 882
8.1.30 socket · · · · · 882
8.1.31 sql_mode · · · · · 883

8.1.32 sql_select_limit 从 v4.0.2 版本开始引入 883
8.1.33 system_time_zone 883
8.1.34 tidb_allow_batch_cop 从 v4.0 版本开始引入 883
8.1.35 tidb_allow_fallback_to_tikv 从 v5.0 版本开始引入 883
8.1.36 tidb_allow_function_for_expression_index 从 v5.2.0 版本开始引入 884
8.1.37 tidb_allow_mpp 从 v5.0 版本开始引入 884
8.1.38 tidb_allow_remove_auto_inc 从 v2.1.18 和 v3.0.4 版本开始引入 884
8.1.39 tidb_analyze_version 从 v5.1.0 版本开始引入 884
8.1.40 tidb_auto_analyze_end_time 884
8.1.41 tidb_auto_analyze_ratio 885
8.1.42 tidb_auto_analyze_start_time 885
8.1.43 tidb_max_auto_analyze_time 从 v6.1.0 版本开始引入 885
8.1.44 tidb_backoff_lock_fast 885
8.1.45 tidb_backoff_weight 886
8.1.46 tidb_batch_insert 886
8.1.47 tidb_broadcast_join_threshold_count 从 v5.0 版本开始引入 886
8.1.48 tidb_broadcast_join_threshold_size 从 v5.0 版本开始引入 886
8.1.49 tidb_build_stats_concurrency 887
8.1.50 tidb_capture_plan_baselines 从 v4.0 版本开始引入 887
8.1.51 tidb_check_mb4_value_in_utf8 887
8.1.52 tidb_committer_concurrency 从 v6.1.0 版本开始引入 887
8.1.53 tidb_checksum_table_concurrency 887
8.1.54 tidb_config 888
8.1.55 tidb_constraint_check_in_place 888
8.1.56 tidb_current_ts 888
8.1.57 tidb_ddl_error_count_limit 889
8.1.58 tidb_ddl_reorg_batch_size 889
8.1.59 tidb_ddl_reorg_priority 889
8.1.60 tidb_ddl_reorg_worker_cnt 889
8.1.61 tidb_disable_txn_auto_retry 889
8.1.62 tidb_distsql_scan_concurrency 890
8.1.63 tidb_dml_batch_size 890

8.1.64 tidb_enable_1pc 从 v5.0 版本开始引入 ····· 890
8.1.65 tidb_enable_amend_pessimistic_txn 从 v4.0.7 版本开始引入 ····· 891
8.1.66 tidb_enable_async_commit 从 v5.0 版本开始引入 ····· 891
8.1.67 tidb_enable_auto_analyze 从 v6.1.0 版本开始引入 ····· 892
8.1.68 tidb_enable_auto_increment_in_generated ····· 892
8.1.69 tidb_enable_batch_dml 从 v6.1.0 版本开始引入 ····· 892
8.1.70 tidb_enable_cascades_planner ····· 892
8.1.71 tidb_enable_chunk_rpc 从 v4.0 版本开始引入 ····· 893
8.1.72 tidb_enable_clustered_index 从 v5.0 版本开始引入 ····· 893
8.1.73 tidb_enable_collect_execution_info ····· 893
8.1.74 tidb_enable_column_tracking 从 v5.4.0 版本开始引入 ····· 893
8.1.75 tidb_enable_enhanced_security ····· 893
8.1.76 tidb_restricted_read_only 从 v5.2.0 版本开始引入 ····· 894
8.1.77 tidb_enable_fast_analyze ····· 894
8.1.78 tidb_enable_index_merge 从 v4.0 版本开始引入 ····· 895
8.1.79 tidb_enable_legacy_instance_scope 从 v6.0.0 版本开始引入 ····· 895
8.1.80 tidb_enable_list_partition 从 v5.0 版本开始引入 ····· 895
8.1.81 tidb_enable_mutation_checker（从 v6.0.0 版本开始引入）····· 895
8.1.82 tidb_enable_noop_functions 从 v4.0 版本开始引入 ····· 896
8.1.83 tidb_enable_outer_join_reorder 从 v6.1.0 版本开始引入 ····· 896
8.1.84 tidb_enable_paging 从 v5.4.0 版本开始引入 ····· 896
8.1.85 tidb_enable_parallel_apply 从 v5.0 版本开始引入 ····· 897
8.1.86 tidb_enable_prepared_plan_cache 从 v6.1.0 版本开始引入 ····· 897
8.1.87 tidb_enable_pseudo_for_outdated_stats 从 v5.3.0 版本开始引入 ····· 897
8.1.88 tidb_enable_rate_limit_action ····· 897
8.1.89 tidb_enable_slow_log ····· 898
8.1.90 tidb_enable_stmt_summary 从 v3.0.4 版本开始引入 ····· 898
8.1.91 tidb_enable_strict_double_type_check 从 v5.0 版本开始引入 ····· 898
8.1.92 tidb_enable_table_partition ····· 899
8.1.93 tidb_enable_telemetry 从 v4.0.2 版本开始引入 ····· 899
8.1.94 tidb_enable_top_sql 从 v5.4.0 版本开始引入 ····· 899
8.1.95 tidb_enable_tso_follower_proxy 从 v5.3.0 版本开始引入 ····· 899

8.1.96 tidb_enable_vectorized_expression 从 v4.0 版本开始引入 ····· 900
8.1.97 tidb_enable_window_function ····· 900
8.1.98 tidb_enforce_mpp 从 v5.1 版本开始引入 ····· 900
8.1.99 tidb_evolve_plan_baselines 从 v4.0 版本开始引入 ····· 900
8.1.100 tidb_evolve_plan_task_end_time 从 v4.0 版本开始引入 ····· 901
8.1.101 tidb_evolve_plan_task_max_time 从 v4.0 版本开始引入 ····· 901
8.1.102 tidb_evolve_plan_task_start_time 从 v4.0 版本开始引入 ····· 901
8.1.103 tidb_executor_concurrency 从 v5.0 版本开始引入 ····· 901
8.1.104 tidb_expensive_query_time_threshold ····· 902
8.1.105 tidb_force_priority ····· 902
8.1.106 tidb_gc_concurrency 从 v5.0 版本开始引入 ····· 903
8.1.107 tidb_gc_enable 从 v5.0 版本开始引入 ····· 903
8.1.108 tidb_gc_life_time 从 v5.0 版本开始引入 ····· 903
8.1.109 tidb_gc_max_wait_time 从 v6.1.0 版本开始引入 ····· 903
8.1.110 tidb_gc_run_interval 从 v5.0 版本开始引入 ····· 904
8.1.111 tidb_gc_scan_lock_mode 从 v5.0 版本开始引入 ····· 904
8.1.112 tidb_general_log ····· 904
8.1.113 tidb_hash_join_concurrency ····· 905
8.1.114 tidb_hashagg_final_concurrency ····· 905
8.1.115 tidb_hashagg_partial_concurrency ····· 906
8.1.116 tidb_ignore_prepared_cache_close_stmt（从 v6.0 版本开始引入） ····· 906
8.1.117 tidb_index_join_batch_size ····· 906
8.1.118 tidb_index_lookup_concurrency ····· 906
8.1.119 tidb_index_lookup_join_concurrency ····· 907
8.1.120 tidb_index_lookup_size ····· 907
8.1.121 tidb_index_serial_scan_concurrency ····· 907
8.1.122 tidb_init_chunk_size ····· 907
8.1.123 tidb_isolation_read_engines 从 v4.0 版本开始引入 ····· 908
8.1.124 tidb_log_file_max_days 从 v5.3.0 版本开始引入 ····· 908
8.1.125 tidb_low_resolution_tso ····· 908
8.1.126 tidb_nontransactional_ignore_error 从 v6.1.0 版本开始引入 ····· 908
8.1.127 tidb_max_chunk_size ····· 908

8.1.128 tidb_max_delta_schema_count · · · · · 908
8.1.129 tidb_mem_oom_action 从 v6.1.0 版本开始引入 · · · · · 909
8.1.130 tidb_mem_quota_analyze 从 v6.1.0 版本开始引入 · · · · · 909
8.1.131 tidb_mem_quota_apply_cache 从 v5.0 版本开始引入 · · · · · 909
8.1.132 tidb_mem_quota_binding_cache（从 v6.0 版本开始引入） · · · · · 910
8.1.133 tidb_mem_quota_query · · · · · 910
8.1.134 tidb_memory_usage_alarm_ratio · · · · · 910
8.1.135 tidb_metric_query_range_duration 从 v4.0 版本开始引入 · · · · · 911
8.1.136 tidb_metric_query_step 从 v4.0 版本开始引入 · · · · · 911
8.1.137 tidb_multi_statement_mode 从 v4.0.11 版本开始引入 · · · · · 911
8.1.138 tidb_enable_new_only_full_group_by_check 从 v6.1.0 版本开始引入 · · · · · 912
8.1.139 tidb_opt_agg_push_down · · · · · 912
8.1.140 tidb_opt_correlation_exp_factor · · · · · 912
8.1.141 tidb_opt_correlation_threshold · · · · · 912
8.1.142 tidb_opt_distinct_agg_push_down · · · · · 913
8.1.143 tidb_opt_enable_correlation_adjustment · · · · · 914
8.1.144 tidb_opt_insubq_to_join_and_agg · · · · · 914
8.1.145 tidb_opt_limit_push_down_threshold · · · · · 914
8.1.146 tidb_opt_prefer_range_scan 从 v5.0 版本开始引入 · · · · · 915
8.1.147 tidb_opt_write_row_id · · · · · 915
8.1.148 tidb_partition_prune_mode 从 v5.1 版本开始引入 · · · · · 916
8.1.149 tidb_persist_analyze_options 从 v5.4.0 版本开始引入 · · · · · 916
8.1.150 tidb_placement_mode（从 v6.0.0 版本开始引入） · · · · · 916
8.1.151 tidb_pprof_sql_cpu 从 v4.0 版本开始引入 · · · · · 916
8.1.152 tidb_prepared_plan_cache_memory_guard_ratio 从 v6.1.0 版本开始引入 · · · · · 916
8.1.153 tidb_prepared_plan_cache_size 从 v6.1.0 版本开始引入 · · · · · 917
8.1.154 tidb_projection_concurrency · · · · · 917
8.1.155 tidb_query_log_max_len · · · · · 917
8.1.156 tidb_rc_read_check_ts（从 v6.0.0 版本开始引入） · · · · · 918
8.1.157 tidb_read_staleness 从 v5.4.0 版本开始引入 · · · · · 918
8.1.158 tidb_record_plan_in_slow_log · · · · · 918
8.1.159 tidb_redact_log · · · · · 918

8.1.160 tidb_regard_null_as_point 从 v5.4.0 版本开始引入 ····· 919
8.1.161 tidb_replica_read 从 v4.0 版本开始引入 ····· 919
8.1.162 tidb_retry_limit ····· 919
8.1.163 tidb_row_format_version ····· 920
8.1.164 tidb_scatter_region ····· 920
8.1.165 tidb_skip_ascii_check 从 v5.0 版本开始引入 ····· 920
8.1.166 tidb_skip_isolation_level_check ····· 920
8.1.167 tidb_skip_utf8_check ····· 921
8.1.168 tidb_slow_log_threshold ····· 921
8.1.169 tidb_max_tiflash_threads 从 v6.1.0 版本开始引入 ····· 921
8.1.170 tidb_slow_query_file ····· 921
8.1.171 tidb_snapshot ····· 922
8.1.172 tidb_stats_cache_mem_quota 从 v6.1.0 版本开始引入 ····· 922
8.1.173 tidb_stats_load_sync_wait 从 v5.4.0 版本开始引入 ····· 922
8.1.174 tidb_stats_load_pseudo_timeout 从 v5.4.0 版本开始引入 ····· 923
8.1.175 tidb_stmt_summary_history_size 从 v4.0 版本开始引入 ····· 923
8.1.176 tidb_stmt_summary_internal_query 从 v4.0 版本开始引入 ····· 923
8.1.177 tidb_stmt_summary_max_sql_length 从 v4.0 版本开始引入 ····· 923
8.1.178 tidb_stmt_summary_max_stmt_count 从 v4.0 版本开始引入 ····· 923
8.1.179 tidb_stmt_summary_refresh_interval 从 v4.0 版本开始引入 ····· 924
8.1.180 tidb_top_sql_max_meta_count 从 v6.0.0 版本开始引入 ····· 924
8.1.181 tidb_top_sql_max_time_series_count 从 v6.0.0 版本开始引入 ····· 924
8.1.182 tidb_store_limit 从 v3.0.4 和 v4.0 版本开始引入 ····· 924
8.1.183 tidb_sysdate_is_now（从 v6.0.0 版本开始引入） ····· 924
8.1.184 tidb_table_cache_lease（从 v6.0.0 版本开始引入） ····· 925
8.1.185 tidb_tmp_table_max_size 从 v5.3 版本开始引入 ····· 925
8.1.186 tidb_tso_client_batch_max_wait_time 从 v5.3.0 版本开始引入 ····· 925
8.1.187 tidb_txn_assertion_level（从 v6.0.0 版本开始引入） ····· 926
8.1.188 tidb_txn_mode ····· 926
8.1.189 tidb_use_plan_baselines（从 v4.0 版本开始引入） ····· 926
8.1.190 tidb_wait_split_region_finish ····· 927
8.1.191 tidb_wait_split_region_timeout ····· 927

8.1.192 tidb_window_concurrency 从 v4.0 版本开始引入 · · · · · 927
8.1.193 time_zone · · · · · 927
8.1.194 timestamp · · · · · 928
8.1.195 transaction_isolation · · · · · 928
8.1.196 tx_isolation · · · · · 928
8.1.197 version · · · · · 928
8.1.198 version_comment · · · · · 928
8.1.199 version_compile_os · · · · · 928
8.1.200 version_compile_machine · · · · · 928
8.1.201 wait_timeout · · · · · 929
8.1.202 warning_count · · · · · 929
8.1.203 windowing_use_high_precision · · · · · 929

9 Placement Rules 使用文档 929
9.1 规则系统介绍 · · · · · 929
9.1.1 规则字段 · · · · · 930
9.1.2 规则分组字段 · · · · · 931
9.2 配置规则操作步骤 · · · · · 931
9.2.1 开启 Placement Rules 特性 · · · · · 931
9.2.2 关闭 Placement Rules 特性 · · · · · 932
9.2.3 使用 pd-ctl 设置规则 · · · · · 932
9.2.4 使用 pd-ctl 设置规则分组 · · · · · 934
9.2.5 使用 pd-ctl 批量更新分组及组内规则 · · · · · 934
9.2.6 使用 pd-ctl 查看和修改所有配置 · · · · · 935
9.2.7 使用 tidb-ctl 查询表相关的 key range · · · · · 935
9.3 典型场景示例 · · · · · 936
9.3.1 场景一：普通的表使用 3 副本，元数据使用 5 副本提升集群容灾能力 · · · · · 936
9.3.2 场景二：5 副本按 2-2-1 的比例放置在 3 个数据中心，且第 3 个中心不产生 Leader · · · · · 936
9.3.3 场景三：为某张表添加 2 个 TiFlash Learner 副本 · · · · · 937
9.3.4 场景四：为某张表在有高性能磁盘的北京节点添加 2 个 Follower 副本 · · · · · 938
9.3.5 场景五：将某张表迁移至 TiFlash 集群 · · · · · 938

10 读取历史数据 939

10.1 使用 Stale Read 功能读取历史数据（推荐） · · · · · 939
10.1.1 Stale Read 功能的使用场景 · · · · · 939
10.1.2 使用 AS OF TIMESTAMP 语法读取历史数据 · · · · · 940
10.1.3 通过系统变量 tidb_read_staleness 读取历史数据 · · · · · 945
10.2 通过系统变量 tidb_snapshot 读取历史数据 · · · · · 947
10.2.1 功能说明 · · · · · 947
10.2.2 操作流程 · · · · · 948
10.2.3 历史数据保留策略 · · · · · 948
10.2.4 示例 · · · · · 949
10.2.5 历史数据恢复策略 · · · · · 951
11 监控与告警 951
11.1 TiDB 监控框架概述 · · · · · 951
11.1.1 Prometheus 在 TiDB 中的应用 · · · · · 952
11.1.2 Grafana 在 TiDB 中的应用 · · · · · 952
11.2 TiDB 集群监控 API · · · · · 954
11.2.1 使用状态接口 · · · · · 954
11.2.2 使用 metrics 接口 · · · · · 957
11.3 TiDB 集群监控部署 · · · · · 957
11.3.1 部署 Prometheus 和 Grafana · · · · · 957
11.3.2 配置 Grafana · · · · · 960
11.3.3 查看组件 metrics · · · · · 961
11.4 将 Grafana 监控数据导出成快照 · · · · · 963
11.4.1 使用方法 · · · · · 963
11.4.2 FAQs · · · · · 964
11.5 TiDB 集群报警规则 · · · · · 965
11.5.1 TiDB 报警规则 · · · · · 965
11.5.2 PD 报警规则 · · · · · 968
11.5.3 TiKV 报警规则 · · · · · 973
11.5.4 TiFlash 报警规则 · · · · · 980
11.5.5 TiDB Binlog 报警规则 · · · · · 980
11.5.6 TiCDC 报警规则 · · · · · 980
11.5.7 Node_exporter 主机报警规则 · · · · · 980
11.5.8 Blackbox_exporter TCP、ICMP 和 HTTP 报警规则 · · · · · 983

11.6 TiFlash 报警规则 · · · · · 987
11.6.1 TiFlash_schema_error · · · · · 988
11.6.2 TiFlash_schema_apply_duration · · · · · 988
11.6.3 TiFlash_raft_read_index_duration · · · · · 988
11.6.4 TiFlash_raft_wait_index_duration · · · · · 989
11.7 错误码与故障诊断 · · · · · 989
11.7.1 错误码 · · · · · 989
11.7.2 故障诊断 · · · · · 995
11.8 监控指标 · · · · · 996
11.8.1 Overview 面板重要监控指标详解 · · · · · 996
11.8.2 Performance Overview 面板重要监控指标详解 · · · · · 998
11.8.3 TiDB 重要监控指标详解 · · · · · 1002
11.8.4 PD 重要监控指标详解 · · · · · 1006
11.8.5 TiKV 监控指标详解 · · · · · 1015
11.8.6 TiFlash 集群监控 · · · · · 1034
11.8.7 TiCDC 重要监控指标详解 · · · · · 1037
11.9 TiDB Dashboard · · · · · 1042
11.9.1 TiDB Dashboard 介绍 · · · · · 1042
11.9.2 运维 · · · · · 1043
11.9.3 访问 TiDB Dashboard · · · · · 1057
11.9.4 TiDB Dashboard 概况页面 · · · · · 1061
11.9.5 TiDB Dashboard 集群信息页面 · · · · · 1067
11.9.6 TiDB Dashboard Top SQL 页面 · · · · · 1070
11.9.7 TiDB Dashboard 流量可视化页面 · · · · · 1077
11.9.8 TiDB Dashboard 监控关系图 · · · · · 1090
11.9.9 SQL 语句分析 · · · · · 1095
11.9.10 TiDB Dashboard 慢查询页面 · · · · · 1109
11.9.11 集群诊断页面 · · · · · 1113
11.9.12 TiDB Dashboard 日志搜索页面 · · · · · 1135
11.9.13 实例性能分析 · · · · · 1139
11.9.14 会话管理与配置 · · · · · 1150
11.9.15 TiDB Dashboard 常见问题 · · · · · 1175

12 最佳实践 1180
12.1 TiDB 最佳实践 1180
12.1.1 前言 1180
12.1.2 基本概念 1180
12.1.3 场景与实践 1183
12.1.4 TiDB 的最佳适用场景 1185
12.2 开发 Java 应用使用 TiDB 的最佳实践 1185
12.2.1 Java 应用中的数据库相关组件 1186
12.2.2 JDBC 1188
12.2.3 连接池 1192
12.2.4 数据访问框架 1193
12.2.5 Spring Transaction 1194
12.2.6 其他 1195
12.2.7 总结 1195
12.3 操作系统性能参数调优 1196
12.3.1 性能分析工具 1196
12.3.2 性能调优 1197
12.4 HAProxy 在 TiDB 中的最佳实践 1199
12.4.1 HAProxy 简介 1200
12.4.2 HAProxy 部分核心功能介绍 1200
12.4.3 准备环境 1200
12.4.4 部署 HAProxy 1201
12.5 性能调优 1206
12.5.1 优化手册 1206
12.5.2 软件调优 1247
12.5.3 SQL 性能优化 1265
12.5.4 SQL 性能调优 1265
12.5.5 理解 TiDB 执行计划 1265
12.5.6 SQL 优化流程 1326
12.5.7 控制执行计划 1385
13 常见问题 1410

13.1 FAQ · · · · · 1410
13.1.1 一、TiDB 介绍、架构、原理 · · · · · 1410
13.1.2 二、云上部署 · · · · · 1412
13.1.3 三、故障排除 · · · · · 1413
13.2 SQL 操作常见问题 · · · · · 1414
13.2.1 TiDB 对哪些 MySQL variables 兼容？ · · · · · 1414
13.2.2 省略 ORDER BY 条件时 TiDB 中返回结果的顺序与 MySQL 中的不一致 · · · · · 1415
13.2.3 TiDB 是否支持 SELECT FOR UPDATE？ · · · · · 1416
13.2.4 TiDB 的 codec 能保证 UTF8 的字符串是 memcomparable 的吗？我们的 key 需要支持 UTF8，有什么编码建议吗？ · · · · · 1416
13.2.5 一个事务中的语句数量最大是多少？ · · · · · 1416
13.2.6 TiDB 中，为什么出现后插入数据的自增 ID 反而小？ · · · · · 1416
13.2.7 如何在 TiDB 中修改 sql_mode？ · · · · · 1416
13.2.8 用 Sqoop 批量写入 TiDB 数据，虽然配置了 --batch 选项，但还是会遇到 java.sql.BatchUpdateExecption: statement count 5001 exceeds the transaction limitation 的错误，该如何解决？ · · · · · 1417
13.2.9 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？ · · · · · 1417
13.2.10 TiDB 中删除数据后会立即释放空间吗？ · · · · · 1417
13.2.11 数据删除后查询速度为何会变慢？ · · · · · 1417
13.2.12 对数据做删除操作之后，空间回收比较慢，如何处理？ · · · · · 1417
13.2.13 SHOW PROCESSLIST 是否显示系统进程号？ · · · · · 1417
13.2.14 在 TiDB 中如何控制或改变 SQL 提交的执行优先级？ · · · · · 1418
13.2.15 在 TiDB 中 auto analyze 的触发策略是怎样的？ · · · · · 1418
13.2.16 可以使用 Hints 控制优化器行为吗？ · · · · · 1418
13.2.17 触发 Information schema is changed 错误的原因？ · · · · · 1419
13.2.18 触发 Information schema is out of date 错误的原因？ · · · · · 1419
13.2.19 高并发情况下执行 DDL 时报错的原因？ · · · · · 1419
13.2.20 SQL 优化 · · · · · 1420
13.2.21 数据库优化 · · · · · 1421
13.3 高可用常见问题 · · · · · 1421
13.3.1 TiDB 数据是强一致的吗？ · · · · · 1422
13.3.2 官方有没有三中心跨机房多活部署的推荐方案？ · · · · · 1422

13.4 高可靠常见问题 · · · · · 1422
13.4.1 我们的安全漏洞扫描工具对 MySQL version 有要求，TiDB 是否支持修改 server 版本号呢？ · · · · · 1422
13.4.2 TiDB 支持哪些认证协议，过程是怎样的？ · · · · · 1422
13.4.3 如何修改用户名密码和权限？ · · · · · 1422

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 v6.1.0 Release Notes

发版日期：2022 年 6 月 13 日

TiDB 版本：6.1.0

在 6.1.0 版本中，你可以获得以下关键特性：

- List 和 List COLUMNS 分区方式 GA，与 MySQL 5.7 兼容
- TiFlash 分区表动态裁剪 GA
- 支持兼容 MySQL 的用户级别锁管理
- 支持非事务性 DML 语法（目前仅支持 DELETE）
- TiFlash 支持按需触发物理数据整理（Compaction）
- MPP 实现窗口函数框架
- TiCDC 支持将 changelogs 以 Avro 协议输出到 Kafka
- DM 合库合表迁移场景的乐观 DDL 协调模式 GA

1.2.1 新功能

1.2.1.1 SQL

- List 和 List COLUMNS 分区方式正式 GA，与 MySQL 5.7 兼容。

  用户文档：List 分区，List COLUMNS 分区

- 支持通过 SQL 语句对 TiFlash 副本立即触发物理数据整理 (Compaction)（实验特性）

  在当前 TiFlash 后台自动整理物理数据（Compaction）机制基础上，新增 compact 命令，帮助刷新旧格式数据，提升读写性能。推荐在升级至 v6.1.0 之后，执行该语句以清理数据。此语句是对标准 SQL 语法的扩展，对 MySQL 客户端保持兼容。升级之外场景一般不需要特别关注。

  用户文档，#4145

- 实现窗口函数框架，支持以下分析函数：

  - RANK()
  - DENSE_RANK()
  - ROW_NUMBER()

用户文档，#33072

1.2.1.2 可观测性

- 持续性能分析支持 ARM 架构和 TiFlash 组件。

  用户文档

- Grafana 中新增 Performance Overview 面板，提供系统级别的总体性能诊断入口。

  Performance Overview 是 TiDB 监控 Grafana 可视化组件中的一个新增面板，基于数据库时间分析法和颜色优化法，按照自上而下的性能分析方法论对 TiDB 的性能指标做了重新梳理，为 TiDB 用户提供一个系统级别的总体性能诊断入口。通过 Performance Overview 面板，你可以直观地看到整个系统的性能瓶颈在哪里，数量级地缩短了性能诊断时间并降低了性能分析和诊断难度。

  用户文档

1.2.1.3 性能

- 支持自定义 Region 大小（实验特性）

  设置更大的 Region 可以有效减少 Region 数量，降低 Region 管理成本，提升集群性能和稳定性。该特性引入 bucket 概念，即将每个 Region 划分为更小的区间 bucket。使用 bucket 作为并发查询单位能够优化 Region 调大时的查询性能，动态调整热点 Region 的大小来保证热点调度效率和负载均衡。该特性目前属于实验特性，不建议在生产环境使用。

  用户文档，#11515

- Raft Engine 存储引擎 GA

  TiDB 从 v6.1.0 开始默认使用引擎 Raft Engine 作为 TiKV 的日志存储引擎。新引擎与 RocksDB 相比，可以减少至多 40% 的 TiKV I/O 写流量和 10% 的 CPU 使用，同时在特定负载下提升 5% 左右前台吞吐，减少 20% 长尾延迟。

  用户文档，#95

- 支持 join order hint 语法

  - LEADING hint：提示优化器使用指定的顺序作为连接前缀，好的连接前缀顺序可以在连接初期快速地降低数据量，提升查询性能。
  - STRAIGHT_JOIN hint：提示优化器按照表在 FROM 子句中的出现顺序进行连接。

  该特性提供更多的手段帮助用户固定连接顺序，合理运用 hint 语法，可以有效提升 SQL 性能和集群稳定性。

用户文档：LEADING，STRAIGHT_JOIN，#29932

- TiFlash 新增对以下函数的支持：

  - FROM_DAYS
  - TO_DAYS
  - TO_SECONDS
  - WEEKOFYEAR

用户文档，#4679，#4678，#4677

- 支持分区表动态裁剪

  默认开启分区表动态裁剪功能，以提升数据分析场景下的性能。v6.0.0 以前版本的用户升级完成后建议及时手动刷新既存分区表的统计信息，以达到最好的性能表现（全新安装，或在 v6.1.0 升级完成后新创建的分区表无需此动作）。

  用户文档：MPP 模式访问分区表，动态裁剪模式，#3873

1.2.1.4 稳定性

- SST 故障自动恢复

  当 RocksDB 后台检测到故障 SST 文件后，TiKV 会尝试通过调度所涉及的故障 Peer，并利用其它副本恢复该节点数据，用户可以通过参数 `background-error-recovery-window` 设置允许的最长恢复时间。如果恢复操作未能在设置的时间窗口内完成，TiKV 将会崩溃。该特性对于可恢复存储故障进行自动检测和恢复，提升了集群稳定性。

  用户文档，#10578

- 支持非事务性 DML 语法

  在大批量的数据处理场景，单一大事务 SQL 处理有可能对集群稳定性和性能造成影响。TiDB 从 v6.1.0 支持对 DELETE 语句提供拆分后批量处理的语法格式，拆分后的语句将牺牲事务原子性和隔离性，但是对于集群的稳定性有很大提升，详细语法请参考BATCH。

  用户文档

- TiDB 支持设置最大 GC 等待时间

  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，并通过 Garbage Collection (GC) 的任务定期清理不再需要的旧数据。定期 GC 清理有助于回收存储空间，提升集群性能和稳定性，TiDB 默认每 10 分钟自动执行 GC 清理。为了保证长时间执行的事务可以访问到对应的历史数据，当有执行中的事务时，GC 清理会被推迟。为了保证 GC 清理不会被无限制推迟，TiDB 从 v6.1.0 引入了系统变量`tidb_gc_max_wait_time` 来控制 GC 推迟的最大等待时间，超过该参数允许的时间，GC 将会被强制执行。该参数默认为 24 小时。通过该系统变量，用户可以有效地控制 GC 等待时长与长事务的关系，提升集群的稳定性。

  用户文档

- 支持设置统计信息自动采集任务的最长执行时间

  通过采集统计信息，数据库可以有效掌握数据的分布情况，从而生成合理的执行计划，提升 SQL 的执行效率。TiDB 在后台会定期对频繁变更的数据对象进行统计信息采集，但在业务高峰期时进行统计信息采集可能会对集群资源造成挤压，影响业务的稳定运行。TiDB 从 v6.1.0 开始提供系统变量`tidb_max_auto_analyze_time` 用来控制后台统计信息采集的最长执行时间，默认为 12 小时。当业务没有遇到资源瓶颈的情况时，建议不要修改该参数，确保数据对象的统计信息及时采集。但是当业务压力大，资源不足的时候，可以通过该变量减少统计信息采集的时长，避免统计信息采集对核心业务造成资源争抢。

  用户文档

1.2.1.5 易用性

- 支持集群在 Region 多副本丢失状态下在线一键式恢复数据

  在 TiDB v6.1.0 之前，当出现多个 TiKV 因物理机故障导致 Region 的多副本丢失时，用户需要停机所有 TiKV，使用 TiKV Control 逐一对 TiKV 进行恢复。从 TiDB v6.1.0 起，该过程被完全自动化且不需要停机，恢复过程中不影响其他正常的在线业务。通过 PD Control 触发在线有损恢复数据，大幅简化了恢复步骤，缩短了恢复所需时间，提供了更加友好的恢复摘要信息。

  用户文档，#10483

- 历史统计信息采集持久化

  支持通过 `SHOW ANALYZE STATUS` 语句查询集群级别的统计信息收集任务。在 TiDB v6.1.0 之前，`SHOW ↪ ANALYZE STATUS` 语句仅显示实例级别的统计信息收集任务，且 TiDB 重启后历史任务记录会被清空，因此用户无法查看历史统计信息的采集时间和相关细节。从 TiDB v6.1.0 起，该信息被持久化保存，集群重启后依然可查询，可以为统计信息异常引起的查询性能问题提供排查依据。

  用户文档

- TiDB/TiKV/TiFlash 参数支持在线修改

在 v6.1.0 之前的版本中，配置变更后，必须重启 TiDB 集群，配置才会生效，这对在线业务会造成一定的影响。TiDB v6.1.0 引入了在线修改配置功能，参数修改后，无需重启，即可生效。具体优化如下：

```
* TiDB 将部分配置项转化为系统变量，所有变量支持在线变更，并支持持久化。请注意，转化后，
    ↪ 原有配置项将被废弃。详细变更列表请查看[配置文件参数](#配置文件参数)。
* TiKV 支持部分参数在线变更。详细变更列表请查看[其他](#其他)。
```

```
* TiFlash 配置项 `max_threads` 转化为系统变量 `tidb_max_tiflash_threads`，
    ↪ 从而支持配置项在线变更和持久化。转化后，原有配置项不会废弃。
```

对于从 v6.1.0 之前版本升级到 v6.1.0 的集群（包括滚动升级和停机升级），请注意：

```
* 若升级前集群指定的配置文件中，存在已经配置的项，则升级过程中 TiDB
    ↪ 将会将配置项的值自动更新为对应系统变量的值，以保证升级后，
    ↪ 系统的行为不会因为参数的优化发生变化。
* 上述自动更新仅在升级过程中发生一次，升级完成之后，被废弃的配置项不再有任何效果。
```

通过本次优化，客户可以在线修改参数生效并持久化，方便客户进行日常运维，避免重启生效对在线业务造成的影响。

用户文档

- 支持全局 kill 查询或连接

  支持通过 enable-global-kill 配置项（默认开启）设置全局 kill 开关。

  在 TiDB v6.1.0 之前，当某个特定操作占用大量资源引发集群稳定性问题时，你需要先登陆到对应的 TiDB 节点，然后运行 kill [TiDB] id 命令终止对应的连接及操作。在 TiDB 节点多的情况下，这种方式使用不便，并且容易误操作。从 v6.1.0 起，当开启 enable-global-kill 配置项时，你可以在任意 TiDB 节点运行 kill 命令终止指定的连接及操作，而无需担心客户端和 TiDB 中间有代理时错误地终止其他查询或会话。目前 TiDB 暂时不支持用 Ctrl+C 终止查询或会话。

  用户文档，#8854

- TiKV API V2（实验特性）

  在 v6.1.0 之前，TiKV 作为 Raw Key Value 存储时，由于仅存储了客户端传入的原始数据，因此只提供基本的 Key Value 读写能力。

  TiKV API V2 提供了新的 Raw Key Value 存储格式与访问接口，包括：

  - 数据以 MVCC 方式存储，并记录了数据的变更时间戳。这个特性将为实现 Change Data Capture、增量备份与恢复等打下基础。
  - 数据根据不同的使用方式划分范围，支持单一集群 TiDB、事务 KV、RawKV 应用共存。

由于底层存储格式发生了重大变化，启用 API V2 后，不能将 TiKV 集群回退到 v6.1.0 之前的版本，否则可能导致数据损坏。

```
[用户文档](#api-version-从-v610-版本开始引入)，[#11745](https://github.com/tikv/tikv/issues
    ↪ /11745)
```

1.2.1.6 MySQL 兼容性

- 支持兼容 MySQL 的用户级别锁管理

  用户级别锁是 MySQL 通过内置函数提供的用户命名锁管理系统。它们可以提供锁阻塞、等待、等锁管理能力。用户级别锁在 ORM 框架中也有较为广泛的应用，例如 RoR、Elixir 和 Ecto 等。TiDB 从 v6.1.0 版本开始支持兼容 MySQL 的用户级别锁管理，支持 GET_LOCK、RELEASE_LOCK、RELEASE_ALL_LOCKS 函数。

  用户文档，#14994

1.2.1.7 数据迁移

- DM 合库合表迁移场景的乐观 DDL 协调模式 GA

  对于合库合表迁移任务，DM 在现有乐观 DDL 协调策略的基础上增加了大量场景测试，足以覆盖 90% 日常使用场景。相比悲观协调策略，乐观协调在使用上更为简单、高效，在仔细阅读注意事项后可优先使用。

  用户文档

- DM 的 WebUI 支持根据指定参数条件启动任务

  开始一个迁移任务时，允许指定 “开始时间” 和 “safe-mode 持续时间”。这在创建具有大量 source 的增量迁移任务时尤其有用，无需再为每个 source 精确指定 binlog 起始同步位置。

  用户文档，#5442

1.2.1.8 数据共享订阅

- 支持与更丰富的第三方数据生态系统进行数据共享
  - TiCDC 支持将 TiDB 数据库的增量数据以 Avro 格式发送到 Kafka，通过 Confluent 与 KSQL、Snowflake 等第三方系统进行数据共享。

    用户文档，#5338
  - TiCDC 支持将 TiDB 数据库的增量数据按表分发到不同的 Kafka Topic 中，结合 Canal-json 格式可以将数据直接与 Flink 共享。

    用户文档，#4423
  - TiCDC 支持 SASL GSSAPI 认证类型。增加了使用 Kafka 的 SASL 认证示例。

    用户文档，#4423
- TiCDC 支持同步使用 GBK 编码的上游表。

  用户文档，#4806

1.2.2 兼容性变更

1.2.2.1 系统变量

变量名	修改类型	描述
tidb_enable_list_partition ↪	修改	默认值从 OFF 改为 ON。

变量名	修改类型	描述
tidb_mem_quota_query	修改	增加 GLOBAL 作用域，变量值可以持久化到集群。
tidb_query_log_max_len	修改	变量作用域由 INSTANCE 修改为 GLOBAL，变量值可以持久化到集群。取值范围修改为 [0, 1073741824]。
require_secure_transport	新增	由 TiDB 配置项 require-secure-transport 转化而来。
tidb_committer_concurrency	新增	由 TiDB 配置项 committer-concurrency 转化而来。

变量名	修改类型	描述
tidb_enable_auto_analyze	新增	由 TiDB 配置项 run-auto-analyze 转化而来。
tidb_enable_batch_dml	新增	由 TiDB 配置项 enable-batch-dml 转化而来。
tidb_enable_new_only_full_group_by_check	新增	控制 TiDB 执行 ONLY_FULL_GROUP_BY 检查时的行为。

变量名	修改类型	描述
tidb_enable_outer_join_reorder ↪	新增	控制 TiDB 的 Join Reorder 算法支持 Outer Join，默认开启。对于从旧版本升级上来的集群，该变量的默认值也会是 TRUE。
tidb_enable_prepared_plan_cache ↪	新增	由 TiDB 配置项 prepared ↪ - ↪ plan ↪ - ↪ cache ↪ . ↪ enabled ↪ 转化而来。
tidb_gc_max_wait_time ↪	新增	用于指定活跃事务阻碍 GC safe point 推进的最大时间。

变量名	修改类型	描述
tidb_max_auto_analyze_time ↪	新增	用于指定自动 ANALYZE 的最大执行时间。
tidb_max_tiflash_threads ↪	新增	由 TiFlash 配置项 max_threads ↪ 转化而来，表示 TiFlash 中 request 执行的最大并发度。
tidb_mem_oom_action ↪	新增	由 TiDB 配置项 oom-action ↪ 转化而来。

变量名	修改类型	描述
tidb_mem_quota_analyze	新增	控制 TiDB 更新统计信息时总的内存占用，包括用户执行的 ANALYZE TABLE 和 TiDB 后台自动执行的统计信息更新任务。
tidb_nontransactional_ignore_error	新增	设置是否在非事务语句中立刻返回错误。
tidb_prepared_plan_cache_memory_guard_ratio	新增	由 TiDB 配置项 prepared-plan-cache.memory-guard-ratio 转化而来。

变量名	修改类型	描述
tidb_prepared_plan_cache_size ↪	新增	由 TiDB 配置项 prepared ↪ - ↪ plan ↪ - ↪ cache ↪ . ↪ capacity ↪ 转化而来。
tidb_stmt_summary_internal_quota ↪	新增	控制 TiDB 内部统计信息缓存使用内存的上限。

1.2.2.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	committer ↪ - ↪ concurrency ↪	删除	转化为系统变量 tidb_committer_concurrency ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	enable-batch-dml	删除	转化为系统变量 tidb_enable_batch_dml。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	lower-case-table-names	删除	TiDB 目前只支持 lower_case_table_name=2，如果升级前设置了其他值，升级到 v6.1.0 后该值会丢失。
TiDB	mem-quota-query	删除	转化为系统变量 tidb_mem_quota_query。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	oom-↪ action	删除	转化为系统变量 tidb_mem_oom_action ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	prepared-↪ plan-↪ cache.↪ capacity	删除	转化为系统变量 tidb_prepared_plan_cache_size ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	prepared-↪ plan-↪ cache.↪ enabled	删除	转化为系统变量 tidb_enable_prepared_plan_cache ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	query ↪ - ↪ log ↪ - ↪ max ↪ - ↪ len ↪	删除	转化为系统变量 tidb_query_log_max_len ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	require ↪ - ↪ secure ↪ - ↪ transport ↪	删除	转化为系统变量 require_secure_transport ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	run- ↪ auto ↪ - ↪ analyze ↪	删除	转化为系统变量 tidb_enable_auto_analyze ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	enable-global-kill	新增	当该配置项值默认为 true，KILL 语句和 KILL TIDB 语句均能跨节点终止查询或连接，无需担心错误地终止其他查询或连接。
TiDB	enable-stats-cache-mem-quota	新增	控制 TiDB 是否开启统计信息缓存的内存上限。
TiKV	raft-engine.enable	修改	默认值从 false 修改为 true。

配置文件	配置项	修改类型	描述
TiKV	region-max-keys	修改	默认值从 1440000 修改为 region-split-keys / 2 * 3。
TiKV	region-max-size	修改	默认值从 144 MB 修改为 region-split-size / 2 * 3。
TiKV	coprocessor.enable-region-bucket	新增	是否将 Region 划分为更小的区间 bucket。
TiKV	coprocessor.region-bucket-size	新增	设置 enable-region-bucket 启用时 bucket 的预期大小。

配置文件	配置项	修改类型	描述
TiKV	causal-ts.renew-batch-min-size	新增	时间戳缓存的最小数量。
TiKV	causal-ts.renew-interval	新增	刷新本地缓存时间戳的周期。
TiKV	max-snapshot-file-raw-size	新增	当 snapshot 文件大于该配置项指定的大小时，snapshot 文件会被切割为多个文件。
TiKV	raft-engine.memory-limit	新增	指定 Raft Engine 使用内存的上限。

配置文件	配置项	修改类型	描述
TiKV	`storage.background.error-recovery-window`	新增	RocksDB 检测到可恢复的后台错误后，所允许的最长恢复时间。
TiKV	`storage.api.version`	新增	TiKV 作为 Raw Key Value 存储数据时使用的存储格式与接口版本。
PD	`schedule.max.store.preparing.time`	新增	控制 store 上线阶段的最长等待时间。
TiCDC	`enable.tls`	新增	控制是否使用 TLS 连接 Kafka。

配置文件	配置项	修改类型	描述
TiCDC	sasl-gssapi-user	新增	支持 Kafka SASL/GSSAPI 认证所需要的参数。详情见Sink URI 配置 kafka。
	sasl-gssapi-password		
	sasl-gssapi-auth-type		
	sasl-gssapi-service-name		
	sasl-gssapi-realm		
	sasl-gssapi-key-tab-path		
	sasl-gssapi-kerberos-config-path		

配置文件	配置项	修改类型	描述
TiCDC	avro-decimal-handling-mode	新增	控制 Avro 格式的输出细节。
TiCDC	dispatchers.topic	新增	控制 TiCDC 将增量数据分发到不同 Kafka Topic 的策略
TiCDC	dispatchers.partition	新增	dispatchers.partition 是原 dispatchers.dispatcher 配置项的别名，用于控制增量数据的 Kafka Partition 分发策略。

配置文件	配置项	修改类型	描述
TiCDC	schema-registry	新增	用于指定存储 Avro Schema 的 Schema Registry Endpoint。
DM	dmctl start-relay 命令中的 worker 参数	删除	不推荐使用的方式，将通过更为简单的实现替代。
DM	source 配置中的 relay-dir	删除	由 worker 配置文件中的同名配置项替代。
DM	task 配置中的 is-sharding	删除	由 shard-mode 配置项替代。
DM	task 配置中的 auto-fix-gtid	删除	该配置已在 5.x 版本废弃，v6.1.0 正式移除。

配置文件	配置项	修改类型	描述
DM	source 配置中的 meta-dir、charset	删除	该配置已在 5.x 版本废弃，v6.1.0 正式移除。

1.2.2.3 其他

- 默认开启 Prepared Plan Cache

  在新集群默认打开 Prepared Plan Cache 的开关，对 `Prepare` / `Execute` 请求的执行计划进行缓存，以便在后续执行时跳过查询计划优化这个步骤，获得性能上的提升。升级上来的集群会继承配置文件内的配置。新集群会使用新的默认值，即默认打开该特性，并且每个 Session 最大缓存计划数为 100（capacity=100）。开启后会有一定的内存消耗，详情见Prepared Plan Cache 的内存管理。

- 在 TiDB v6.1.0 之前，`SHOW ANALYZE STATUS` 显示实例级别的任务，且 TiDB 重启后任务记录会被清空。从 TiDB v6.1.0 起，`SHOW ANALYZE STATUS` 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。当 `tidb_analyze_version = 2` 时 `Job_info` 列增加了 `analyze option` 的信息。

- TiKV 中损坏的 SST 文件会导致 TiKV 进程崩溃。在 TiDB v6.1.0 之前，损坏的 SST 文件会导致 TiKV 进程立即崩溃，从 TiDB v6.1.0 起，TiKV 进程会在 SST 文件损坏的 1 小时之后崩溃。

- TiKV 中以下配置项支持在线修改：

  - `raftstore.raft-entry-max-size`
  - `quota.foreground-cpu-time`
  - `quota.foreground-write-bandwidth`
  - `quota.foreground-read-bandwidth`
  - `quota.max-delay-duration`
  - `server.grpc-memory-pool-quota`
  - `server.max-grpc-send-msg-len`
  - `server.raft-msg-max-batch-size`

- v6.1.0 中，部分配置项转化为系统变量。对于从 v6.1.0 之前版本升级到 v6.1.0 的集群（包括滚动升级和停机升级），请注意：

  - 若升级前集群指定的配置文件中，存在已经配置的项，则升级过程中 TiDB 将会将配置项的值自动更新为对应系统变量的值，以保证升级后，系统的行为不会因为参数的优化发生变化。
  - 上述自动更新仅在升级过程中发生一次，升级完成之后，被废弃的配置项不再有任何效果。

- DM WebUI 移除 Dashboard 页面。

- TiCDC 启用 `dispatchers.topic`、`dispatchers.partition` 配置后不支持降级到 v6.1.0 以前的版本。

- TiCDC 使用 Avro 协议的 Changefeed 不支持降级到 v6.1.0 以前的版本。

1.2.3 改进提升

- TiDB
  - 提升 UnionScanRead 算子的性能 #32433
  - 优化 Explain 输出中 task type 的显示（增加 MPP task type）#33332
  - 支持使用 rand() 作为列的默认值 #10377
  - 支持使用 uuid() 作为列的默认值 #33870
  - 支持将表或列的字符集从 latin1 修改为 utf8/utf8mb4 #34008
- TiKV
  - 提升引入内存悲观锁后 CDC 旧数据的命中率 #12279
  - 健康检查可以检测到无法正常工作的 Raftstore，使得 TiKV client 可以及时更新 Region Cache #12398
  - 支持设置 Raft Engine 的内存限制 #12255
  - TiKV 自动检测和删除损坏的 SST 文件，提高产品可服务性 #10578
  - CDC 支持 RawKV #11965
  - 支持将较大的 snapshot 切割为多个文件 #11595
  - 将 snapshot 垃圾回收 (GC) 从 Raftstore 中迁移到后台线程，防止 snapshot GC 阻塞 Raftstore 消息循环 #11966
  - 支持动态设置 gRPC 可发送的最大消息长度 (max-grpc-send-msg-len) 和单个 gPRC 消息可包含的最大 Raft 消息个数 (raft-msg-max-batch-size) #12334
  - 支持通过 Raft 执行在线有损恢复 #10483
- PD
  - 支持设置 Region Label 的 Time to live (TTL) #4694
  - 支持 Region Buckets #4668
  - 默认关闭编译 swagger server #4932
- TiFlash
  - 优化聚合算子的内存统计，从而能在 Merge 阶段选择更高效的算法 #4451
- Tools
  - Backup & Restore (BR)
    - 支持备份恢复空库 #33866
  - TiDB Lightning
    - 优化 Scatter Region 为批量模式，提升 Scatter Region 过程的稳定性 #33618

1.2.4 错误修复

- TiDB
  - 修复 IN 函数处理 BIT 数据类型时可能会导致 TiDB panic 的问题 #33070
  - 修复 UnionScan 无法保序导致的查询结果不正确的问题 #33175

  - 修复特定情况下 Merge Join 执行结果错误的问题 #33042
  - 修复动态裁减模式下 `index join` 的结果可能会错误的问题 #33231
  - 修复分区表的一些分区被 DROP 删除后数据可能不被 GC 垃圾回收的问题 #33620
  - 修复集群的 PD 节点被替换后一些 DDL 语句会卡住一段时间的问题 #33908
  - 修复了查询 `INFORMATION_SCHEMA.CLUSTER_SLOW_QUERY` 表导致 TiDB 服务器 OOM 的问题，在 Grafana dashboard 中查看慢查询记录的时候可能会触发该问题 #33893
  - 修复系统变量 `max_allowed_packet` 不生效的问题 #31422
  - 修复 TopSQL 模块的内存泄露问题 #34525，#34502
  - 修复 Plan Cache 对于 PointGet 计划有时候会出错的问题 #3237
  - 修复在 RC 隔离情况下 Plan Cache 启用时可能导致查询结果错误的问题 #34447

- TiKV

  - 修复下线一个 TiKV 实例导致 Raft log lag 越来越大的问题 #12161
  - 修复待 merge 的 Region 无效会导致 TiKV panic 且非预期地销毁 peer 的问题 #12232
  - 修复从 v5.3.1、v5.4.0 升级到 v6.0.0 及以上版本时 TiKV 报 `failed to load_latest_options` 错误的问题 #12269
  - 修复内存资源不足时 append Raft log 导致 OOM 的问题 #11379
  - 修复销毁 peer 和批量分裂 Region 之间的竞争导致的 TiKV panic #12368
  - 修复 `stats_monitor` 线程陷入死循环导致短期内 TiKV 内存占用陡增的问题 #12416
  - 修复进行 Follower Read 时，可能会报 `invalid store ID 0` 错误的问题 #12478

- PD

  - 修复 `not leader` 的 status code 有误的问题 #4797
  - 修复在某些特殊情况下 TSO fallback 的问题 #4884
  - 修复已清除的 `tombstone store` 信息在切换 PD leader 后再次出现的问题 #4941
  - 修复 PD leader 转移后调度不能立即启动的问题 4769

- TiDB Dashboard

  - 修复 Top SQL 功能无法统计到其开启时刻正在运行的 SQL 的问题 #33859

- TiFlash

  - 修复大量 INSERT 和 DELETE 操作后可能导致 TiFlash 数据不一致的问题 #4956

- Tools

  - TiCDC

    * 优化了 ddl schema 缓存方式，降低了内存消耗 #1386
    * 避免大事务导致的 OOM 问题 #5280
    * 修复了增量扫描特殊场景下的数据丢失问题 #5468

  - TiDB Data Migration (DM)

    * 修复 `start-time` 时区问题，从使用下游时区改为使用上游时区 #5471
    * 修复任务自动恢复后，DM 会占用更多磁盘空间的问题 #3734，#5344
    * 修复 checkpoint flush 可能导致失败行数据被跳过的问题 #5279

    * 修复了某些情况下，过滤 DDL 并在下游手动执行会导致同步任务不能自动重试恢复的问题 #5272
    * 修复在未设置 `case-sensitive: true` 时无法同步大写表的问题 #5255
    * 修复了在 `SHOW CREATE TABLE` 语句返回的索引中，主键没有排在第一位导致的 DM worker panic 的问题 #5159
    * 修复了当开启 GTID 模式或者任务自动恢复时，可能出现一段时间 CPU 占用高并打印大量日志的问题 #5063
    * 修复 DM Web UI offline 选项及其他使用问题 #4993
    * 修复上游 GTID 配置为空时，增量任务启动失败的问题 #3731
    * 修复空配置可能导致 dm-master panic 的问题 #3732

- TiDB Lightning

    * 修复前置检查中没有检查本地磁盘空间以及集群是否可用的问题 #34213
    * 修复 schema 路由错误的问题 #33381
    * 修复 TiDB Lightning panic 时 PD 配置未正确恢复的问题 #31733
    * 修复由 `auto_increment` 列的数据越界导致 local 模式导入失败的问题 #29737
    * 修复 `auto_random`、`auto_increment` 列为空时 local 模式导入失败的问题 #34208

1.3 TiDB 软件和硬件环境建议配置

TiDB 作为一款开源分布式 NewSQL 数据库，可以很好地部署和运行在 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。

1.3.1 Linux 操作系统版本要求

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
Oracle Enterprise Linux	7.3 及以上的 7.x 版本
Amazon Linux	2
Ubuntu LTS	16.04 及以上的版本

注意：

- TiDB 只支持 Red Hat 兼容内核 (RHCK) 的 Oracle Enterprise Linux，不支持 Oracle Enterprise Linux 提供的 Unbreakable Enterprise Kernel。
- TiDB 在 CentOS 7.3 的环境下进行过大量的测试，同时社区也有很多该操作系统部署的最佳实践，因此，建议使用 CentOS 7.3 以上的 7.x Linux 操作系统来部署 TiDB。
- 以上 Linux 操作系统可运行在物理服务器以及 VMware、KVM 及 XEN 主流虚拟化环境上。
- 目前尚不支持 Red Hat Enterprise Linux 8.0、CentOS 8 Stream 和 Oracle Enterprise Linux 8.0，因为目前对这些平台的测试还在进行中。

- 不计划支持 CentOS 8 Linux，因为 CentOS 的上游支持已于 2021 年 12 月 31 日终止。
- TiDB 将不再支持 Ubuntu 16.04。强烈建议升级到 Ubuntu 18.04 或更高版本。

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

1.3.2 软件配置要求

1.3.2.1 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。

1.3.2.2 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

1.3.3 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：

1.3.3.1 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。

1.3.3.2 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+	SAS	万兆网卡（2 块最佳）	2
PD	8 核 +	16 GB+	SSD	万兆网卡（2 块最佳）	3
TiKV	16 核 +	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。

1.3.4 网络要求

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

组件	默认端口	说明
TiDB	4000	应用及 DBA 工具访问通信端口
TiDB	10080	TiDB 状态信息上报通信端口
TiKV	20160	TiKV 通信端口
TiKV	20180	TiKV 状态信息上报通信端口
PD	2379	提供 TiDB 和 PD 通信端口
PD	2380	PD 集群节点间通信端口
TiFlash	9000	TiFlash TCP 服务端口
TiFlash	8123	TiFlash HTTP 服务端口
TiFlash	3930	TiFlash RAFT 服务和 Coprocessor 服务端口
TiFlash	20170	TiFlash Proxy 服务端口
TiFlash	20292	Prometheus 拉取 TiFlash Proxy metrics 端口
TiFlash	8234	Prometheus 拉取 TiFlash metrics 端口
Pump	8250	Pump 通信端口
Drainer	8249	Drainer 通信端口
CDC	8300	CDC 通信接口
Monitoring	9090	Prometheus 服务通信端口
Monitoring	20120	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

1.3.5 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 |

1.3.6 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。

1.4 与 MySQL 兼容性对比

- TiDB 高度兼容 MySQL 5.7 协议、 MySQL 5.7 常用的功能及语法。 MySQL 5.7 生态中的系统工具 (PHPMyAdmin、Navicat、 MySQL Workbench、 mysqldump、 Mydumper/Myloader)、 客户端等均适用于 TiDB。
- 但 TiDB 尚未支持一些 MySQL 功能，可能的原因如下：
  - 有更好的解决方案，例如 JSON 取代 XML 函数。
  - 目前对这些功能的需求度不高，例如存储流程和函数。
  - 一些功能在分布式系统上的实现难度较大。
- 除此以外，TiDB 不支持 MySQL 复制协议，但提供了专用工具用于与 MySQL 复制数据
  - 从 MySQL 复制：TiDB Data Migration (DM) 是将 MySQL/MariaDB 数据迁移到 TiDB 的工具，可用于增量数据的复制。
  - 向 MySQL 复制：TiCDC 是一款通过拉取 TiKV 变更日志实现的 TiDB 增量数据同步工具，可通过 MySQL sink 将 TiDB 增量数据复制到 MySQL。

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

1.4.1 不支持的功能特性

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 外键约束 #18209
- 全文语法与索引 #1793
- 空间类型的函数（即 GIS/GEOMETRY ）、 数据类型和索引 #6347

- 非 ascii、latin1、binary、utf8、utf8mb4、gbk 的字符集
- SYS schema
- MySQL 追踪优化器
- XML 函数
- X-Protocol #1109
- Savepoints #6840
- 列级权限 #9766
- XA 语法（TiDB 内部使用两阶段提交，但并没有通过 SQL 接口公开）
- CREATE TABLE tblName AS SELECT stmt 语法 #4754
- CHECK TABLE 语法 #4673
- CHECKSUM TABLE 语法 #1895
- REPAIR TABLE 语法
- OPTIMIZE TABLE 语法
- HANDLER 语句
- CREATE TABLESPACE 语句

1.4.2 与 MySQL 有差异的特性详细说明

1.4.2.1 自增 ID

- TiDB 的自增列既能保证唯一，也能保证在单个 TiDB server 中自增，但不保证多个 TiDB server 中自增，不保证自动分配的值的连续性。不建议将缺省值和自定义值混用，若混用可能会收到 Duplicated Error 的错误信息。
- TiDB 可通过 tidb_allow_remove_auto_inc 系统变量开启或者关闭允许移除列的 AUTO_INCREMENT 属性。删除列属性的语法是：ALTER TABLE MODIFY 或 ALTER TABLE CHANGE。
- TiDB 不支持添加列的 AUTO_INCREMENT 属性，移除该属性后不可恢复。

自增 ID 详情可参阅AUTO_INCREMENT。

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

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

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

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

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

1.4.2.2 Performance schema

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

1.4.2.3 查询计划

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

1.4.2.4 内建函数

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

1.4.2.5 DDL 的限制

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

- 不能在单条 ALTER TABLE 语句中完成多个操作。例如，不能在单个语句中添加多个列或索引，否则，可能会输出 Unsupported multi schema change 的错误。
- ALTER TABLE 不支持少部分类型的变更。比如，TiDB 不支持从 DECIMAL 到 DATE 的变更。当遇到不支持的类型变更时，TiDB 将会报 Unsupported modify column: type %d not match origin %d 的错误。更多细节，请参考ALTER TABLE。
- TiDB 中，ALGORITHM={INSTANT,INPLACE,COPY} 语法只作为一种指定，并不更改 ALTER 算法，详情参阅ALTER TABLE。
- 不支持添加或删除 CLUSTERED 类型的主键。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 不支持指定不同类型的索引 (HASH|BTREE|RTREE|FULLTEXT)。若指定了不同类型的索引，TiDB 会解析并忽略这些索引。
- 分区表支持 HASH、RANGE 和 LIST 分区类型。对于不支持的分区类型，TiDB 可能会报 Warning: Unsupported partition type %s, treat as normal table 错误，其中 %s 为不支持的具体分区类型。

- 分区表还支持 ADD、 DROP、 TRUNCATE 操作。其他分区操作会被忽略。TiDB 不支持以下分区表语法：
  - PARTITION BY KEY
  - SUBPARTITION
  - {CHECK|TRUNCATE|OPTIMIZE|REPAIR|IMPORT|DISCARD|REBUILD|REORGANIZE|COALESCE} PARTITION

  更多详情，请参考分区表文档。

1.4.2.6 ANALYZE TABLE

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

更多信息统计的差异请参阅ANALYZE TABLE。

1.4.2.7 SELECT 的限制

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

详情参见SELECT。

1.4.2.8 UPDATE 语句

详情参见UPDATE。

1.4.2.9 视图

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

1.4.2.10 临时表

详见TiDB 本地临时表与 MySQL 临时表的兼容性。

1.4.2.11 字符集和排序规则

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

1.4.2.12 存储引擎

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

1.4.2.13 SQL 模式

TiDB 支持大部分SQL 模式。不支持的 SQL 模式如下：

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

1.4.2.14 默认设置

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

1.4.2.15 日期时间处理的区别

1.4.2.15.1 时区

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

1.4.2.16 类型系统

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

1.4.2.17 MySQL 弃用功能导致的不兼容问题

TiDB 不支持 MySQL 中标记为弃用的功能，包括：

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

1.5 使用限制

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

1.5.1 标识符长度限制

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

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

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

标识符类型	最大个数
Connections	unlimited

1.5.3 单个 Database 的限制

类型	最大限制
Tables	unlimited

1.5.4 单个 Table 的限制

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

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

1.5.5 单行的限制

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

1.5.6 单列的限制

类型	最大限制
Size	6MB

1.5.7 字符串类型限制

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

类型	最大限制
TEXT	6MB 字节
BLOB	6MB 字节

1.5.8 SQL Statements 的限制

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

2 SQL 参考指南

2.1 SQL 语言结构和语法

2.1.1 属性

2.1.1.1 AUTO_INCREMENT

本文介绍列属性 AUTO_INCREMENT 的基本概念、实现原理、自增相关的特性，以及使用限制。

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

2.1.1.1.1 基本概念

AUTO_INCREMENT 是用于自动填充缺省列值的列属性。当 INSERT 语句没有指定 AUTO_INCREMENT 列的具体值时，系统会自动地为该列分配一个值。

出于性能原因，自增编号是系统批量分配给每台 TiDB 服务器的值（默认 3 万个值），因此自增编号能保证唯一性，但分配给 INSERT 语句的值仅在单台 TiDB 服务器上具有单调性。

```
CREATE TABLE t(id int PRIMARY KEY AUTO_INCREMENT, c int);
```

```
INSERT INTO t(c) VALUES (1);
INSERT INTO t(c) VALUES (2);
INSERT INTO t(c) VALUES (3), (4), (5);
```

```
SELECT * FROM t;
+----+---+
| id | c |
```

```
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
+----+---+
5 rows in set (0.01 sec)
```

此外，AUTO_INCREMENT 还支持显式指定列值的插入语句，此时 TiDB 会保存显式指定的值：

```
INSERT INTO t(id, c) VALUES (6, 6);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
| 6  | 6 |
+----+---+
6 rows in set (0.01 sec)
```

以上用法和 MySQL 的 AUTO_INCREMENT 用法一致。但在隐式分配的具体值方面，TiDB 和 MySQL 之间具有较为显著的差异。

2.1.1.1.2 实现原理

TiDB 实现 AUTO_INCREMENT 隐式分配的原理是，对于每一个自增列，都使用一个全局可见的键值对用于记录当前已分配的最大 ID。由于分布式环境下的节点通信存在一定开销，为了避免写请求放大的问题，每个 TiDB 节点在分配 ID 时，都申请一段 ID 作为缓存，用完之后再去取下一段，而不是每次分配都向存储节点申请。例如，对于以下新建的表：

```
CREATE TABLE t(id int UNIQUE KEY AUTO_INCREMENT, c int);
```

假设集群中有两个 TiDB 实例 A 和 B，如果向 A 和 B 分别对 t 执行一条插入语句：

```
INSERT INTO t (c) VALUES (1)
```

实例 A 可能会缓存 [1,30000] 的自增 ID，而实例 B 则可能缓存 [30001,60000] 的自增 ID。各自实例缓存的 ID 将随着执行将来的插入语句被作为缺省值，顺序地填充到 AUTO_INCREMENT 列中。

2.1.1.1.3 基本特性

唯一性保证

> 警告：
>
> 在集群中有多个 TiDB 实例时，如果表结构中有自增 ID，建议不要混用显式插入和隐式分配 (即自增列的缺省值和自定义值)，否则可能会破坏隐式分配值的唯一性。

例如在上述示例中，依次执行如下操作：

1. 客户端向实例 B 插入一条将 id 设置为 2 的语句 `INSERT INTO t VALUES (2, 1)`，并执行成功。
2. 客户端向实例 A 发送 INSERT 语句 `INSERT INTO t (c)(1)`，这条语句中没有指定 id 的值，所以会由 A 分配。当前 A 缓存了 `[1, 30000]` 这段 ID，可能会分配 2 为自增 ID 的值，并把本地计数器加 1。而此时数据库中已经存在 id 为 2 的数据，最终返回 `Duplicated Error` 错误。

单调性保证

TiDB 保证 AUTO_INCREMENT 自增值在单台服务器上单调递增。以下示例在一台服务器上生成连续的 AUTO_INCREMENT 自增值 1-3：

```
CREATE TABLE t (a int PRIMARY KEY AUTO_INCREMENT, b timestamp NOT NULL DEFAULT NOW());
INSERT INTO t (a) VALUES (NULL), (NULL), (NULL);
SELECT * FROM t;
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
+---+---------------------+
| a | b                   |
+---+---------------------+
| 1 | 2020-09-09 20:38:22 |
| 2 | 2020-09-09 20:38:22 |
| 3 | 2020-09-09 20:38:22 |
+---+---------------------+
3 rows in set (0.00 sec)
```

如果在另一台服务器上执行插入操作，那么 AUTO_INCREMENT 值的顺序可能会剧烈跳跃，这是由于每台服务器都有各自缓存的 AUTO_INCREMENT 自增值。

```
INSERT INTO t (a) VALUES (NULL);
SELECT * FROM t;
```

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

```
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
+---------+---------------------+
4 rows in set (0.00 sec)
```

以下示例在最先的一台服务器上执行一个插入 INSERT 操作，生成 AUTO_INCREMENT 值 4。因为这台服务器上仍有剩余的 AUTO_INCREMENT 缓存值可用于分配。在该示例中，值的顺序不具有全局单调性：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
+---------+---------------------+
5 rows in set (0.00 sec)
```

AUTO_INCREMENT 缓存不会持久化，重启会导致缓存值失效。以下示例中，最先的一台服务器重启后，向该服务器执行一条插入操作：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
| 2030001 | 2020-09-09 20:54:11 |
+---------+---------------------+
6 rows in set (0.00 sec)
```

TiDB 服务器频繁重启可能导致 AUTO_INCREMENT 缓存值被快速消耗。在以上示例中，最先的一台服务器本来有可用的缓存值 [5-3000]。但重启后，这些值便丢失了，无法进行重新分配。

用户不应指望 AUTO_INCREMENT 值保持连续。在以下示例中，一台 TiDB 服务器的缓存值为 [2000001-2030000]。当手动插入值 2029998 时，TiDB 取用了一个新缓存区间的值：

```
INSERT INTO t (a) VALUES (2029998);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.00 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.02 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
| 2030001 | 2020-09-09 20:54:11 |
| 2029998 | 2020-09-09 21:08:11 |
| 2029999 | 2020-09-09 21:08:11 |
| 2030000 | 2020-09-09 21:08:11 |
| 2060001 | 2020-09-09 21:08:11 |
| 2060002 | 2020-09-09 21:08:11 |
+---------+---------------------+
11 rows in set (0.00 sec)
```

以上示例插入 2030000 后，下一个值为 2060001，即顺序出现跳跃。这是因为另一台 TiDB 服务器获取了中间缓存区间 [2030001-2060000]。当部署有多台 TiDB 服务器时，AUTO_INCREMENT 值的顺序会出现跳跃，因为对缓存值的请求是交叉出现的。

缓存大小控制

TiDB 自增 ID 的缓存大小在早期版本中是对用户透明的。从 v3.1.2、v3.0.14 和 v4.0.rc-2 版本开始，TiDB 引入了 AUTO_ID_CACHE 表选项来允许用户自主设置自增 ID 分配缓存的大小。例如：

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 100;
Query OK, 0 rows affected (0.02 sec)

INSERT INTO t VALUES();
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

SELECT * FROM t;
+---+
| a |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
```

此时如果将该列的自增缓存无效化，重新进行隐式分配：

```
DELETE FROM t;
Query OK, 1 row affected (0.01 sec)

RENAME TABLE t to t1;
Query OK, 0 rows affected (0.01 sec)

INSERT INTO t1 VALUES()
Query OK, 1 row affected (0.00 sec)

SELECT * FROM t;
+-----+
| a   |
+-----+
| 101 |
+-----+
1 row in set (0.00 sec)
```

可以看到再一次分配的值为 101，说明该表的自增 ID 分配缓存的大小为 100。

此外如果在批量插入的 INSERT 语句中所需连续 ID 长度超过 AUTO_ID_CACHE 的长度时，TiDB 会适当调大缓存以便能够保证该语句的正常插入。

自增步长和偏移量设置

从 v3.0.9 和 v4.0.rc-1 开始，和 MySQL 的行为类似，自增列隐式分配的值遵循 session 变量 `@@auto_increment_increment` 和 `@@auto_increment_offset` 的控制，其中自增列隐式分配的值 (ID) 将满足式子 `(ID - auto_increment_offset )% auto_increment_increment == 0`。

2.1.1.1.4 使用限制

目前在 TiDB 中使用 AUTO_INCREMENT 有以下限制：

- 定义的列必须为主键或者索引的首列。
- 只能定义在类型为整数、 FLOAT 或 DOUBLE 的列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- 不支持使用 ALTER TABLE 来添加 AUTO_INCREMENT 属性。
- 支持使用 ALTER TABLE 来移除 AUTO_INCREMENT 属性。但从 TiDB 2.1.18 和 3.0.4 版本开始，TiDB 通过 session 变量 @@tidb_allow_remove_auto_inc 控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除列的 AUTO_INCREMENT 属性，默认是不允许移除。

2.1.1.2 AUTO_RANDOM 从 v3.1.0 版本开始引入

> **注意：**
>
> AUTO_RANDOM 属性已于 v4.0.3 版本成为正式功能。

2.1.1.2.1 使用场景

AUTO_RANDOM 用于解决大批量写数据入 TiDB 时因含有整型自增主键列的表而产生的热点问题。详情参阅 TiDB 高并发写入场景最佳实践。

以下面语句建立的表为例：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_INCREMENT, b varchar(255))
```

在以上语句所建的表上执行大量未指定主键值的 INSERT 语句，示例如下：

```
INSERT INTO t(b) VALUES ('a'), ('b'), ('c')
```

如以上语句，由于未指定主键列的值（a 列），TiDB 会使用连续自增的行值作为行 ID，可能导致单个 TiKV 节点上产生写入热点，进而影响对外提供服务的性能。要避免这种写入热点，可以在执行建表语句时为 a 列指定 AUTO_RANDOM 属性而不是 AUTO_INCREMENT 属性。示例如下：

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

或者

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

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

- 隐式分配：如果该 INSERT 语句没有指定整型主键列（a 列）的值，或者指定为 NULL，TiDB 会为该列自动分配值。该值不保证自增，不保证连续，只保证唯一，避免了连续的行 ID 带来的热点问题。
- 显式插入：如果该 INSERT 语句显式指定了整型主键列的值，和 AUTO_INCREMENT 属性类似，TiDB 会保存该值。注意，如果未在系统变量 @@sql_mode 中设置 NO_AUTO_VALUE_ON_ZERO，即使显式指定整型主键列的值为 0，TiDB 也会为该列自动分配值。

> 注意：
>
> 从 v4.0.3 开始，要使用显式插入的功能，需要将系统变量 @@allow_auto_random_explicit_insert 的值设置为 1（默认值为 0）。默认不支持显式插入的具体原因请参考使用限制一节。

自动分配值的计算方式如下：

该行值在二进制形式下，除去符号位的最高五位（称为 shard bits）由当前事务的开始时间决定，剩下的位数按照自增的顺序分配。

若要使用一个不同的 shard bits 的数量，可以在 AUTO_RANDOM 后面加一对括号，并在括号中指定想要的 shard bits 数量。示例如下：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM(3), b varchar(255))
```

以上建表语句中，shard bits 的数量为 3。shard bits 的数量的取值范围是 [1, 16)。

创建完表后，使用 SHOW WARNINGS 可以查看当前表可支持的最大隐式分配的次数：

```
SHOW WARNINGS
```

```
+-------+------+-----------------------------------------------------------+
| Level | Code | Message                                                   |
+-------+------+-----------------------------------------------------------+
| Note  | 1105 | Available implicit allocation times: 1152921504606846976 |
+-------+------+-----------------------------------------------------------+
```

> 注意：
>
> 为保证可隐式分配的次数最大，从 v4.0.3 开始，AUTO_RANDOM 列类型只能为 BIGINT。

另外，要查看某张含有 AUTO_RANDOM 属性的表的 shard bits 数量，可以在系统表 information_schema.tables 中 TIDB_ROW_ID_SHARDING_INFO 一列看到模式为 PK_AUTO_RANDOM_BITS=x 的值，其中 x 为 shard bits 的数量。

AUTO_RANDOM 列隐式分配的值会影响 last_insert_id()。可以使用 SELECT last_insert_id() 获取上一次 TiDB 隐式分配的 ID，例如：

```
INSERT INTO t (b) VALUES ("b")
SELECT * FROM t;
SELECT last_insert_id()
```

可能得到的结果如下：

```
+------------+---+
| a          | b |
+------------+---+
| 1073741825 | b |
+------------+---+

+------------------+
| last_insert_id() |
+------------------+
| 1073741825       |
+------------------+
```

2.1.1.2.2 兼容性

TiDB 支持解析版本注释语法。示例如下：

```
CREATE TABLE t (a bigint PRIMARY KEY /*T![auto_rand] auto_random */)
```

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM)
```

以上两个语句含义相同。

在 SHOW CREATE TABLE 的结果中，AUTO_RANDOM 属性会被注释掉。注释会附带一个特性标识符，例如 /*T![auto_rand] auto_random */。其中 auto_rand 表示 AUTO_RANDOM 的特性标识符，只有实现了该标识符对应特性的 TiDB 版本才能够正常解析 SQL 语句片段。

该功能支持向前兼容，即降级兼容。没有实现对应特性的 TiDB 版本则会忽略表（带有上述注释）的 AUTO_RANDOM 属性，因此能够使用含有该属性的表。

2.1.1.2.3 使用限制

目前在 TiDB 中使用 AUTO_RANDOM 有以下限制：

- 该属性必须指定在 BIGINT 类型的主键列上，否则会报错。此外，当主键属性为 NONCLUSTERED 时，即使是整型主键列，也不支持使用 AUTO_RANDOM。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 不支持使用 ALTER TABLE 来修改 AUTO_RANDOM 属性，包括添加或移除该属性。
- 支持将 AUTO_INCREMENT 属性改为 AUTO_RANDOM 属性。但在 AUTO_INCREMENT 的列数据最大值已接近 BIGINT 类型最大值的情况下，修改可能会失败。
- 不支持修改含有 AUTO_RANDOM 属性的主键列的列类型。
- 不支持与 AUTO_INCREMENT 同时指定在同一列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- AUTO_RANDOM 列的数据很难迁移到 AUTO_INCREMENT 列上，因为 AUTO_RANDOM 列自动分配的值通常都很大。
- 插入数据时，不建议自行显式指定含有 AUTO_RANDOM 列的值。不恰当地显式赋值，可能会导致该表提前耗尽用于自动分配的数值。

2.1.1.3 SHARD_ROW_ID_BITS

本文介绍表属性 SHARD_ROW_ID_BITS，它用来设置隐式 _tidb_rowid 分片数量的 bit 位数。

2.1.1.3.1 基本概念

对于非整数主键或没有主键的表，TiDB 会使用一个隐式的自增 rowid。大量执行 INSERT 插入语句时会把数据集中写入单个 Region，造成写入热点。

通过设置 SHARD_ROW_ID_BITS，可以把 rowid 打散写入多个不同的 Region，缓解写入热点问题。但是设置的过大会造成 RPC 请求数放大，增加 CPU 和网络开销。

- SHARD_ROW_ID_BITS = 4 表示 16 个分片
- SHARD_ROW_ID_BITS = 6 表示 64 个分片
- SHARD_ROW_ID_BITS = 0 表示默认值 1 个分片

2.1.1.3.2 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;

2.1.2 字面值

TiDB 字面值包括字符字面值、数值字面值、时间日期字面值、十六进制、二进制字面值和 NULL 字面值。以下分别对这些字面值进行一一介绍。

2.1.2.1 String Literals

String Literals 是一个 bytes 或者 characters 的序列，两端被单引号 ' 或者双引号 " 包围，例如：

```
'example string'
"example string"
```

如果字符串是连续的，会被合并为一个独立的 string。以下表示是一样的：

```
'a string'
'a' ' ' 'string'
"a" ' ' "string"
```

如果开启了 ANSI_QUOTES SQL MODE，那么只有单引号内的会被认为是 String Literals，对于双引号内的字符串，会被认为是一个 identifier。

字符串分为以下两种：

- 二进制字符串 (binary string)：由字节序列构成，它的 charset 和 collation 都是 binary，在互相比较时利用字节作为单位。
- 非二进制字符串：由字符序列构成，有除 binary 以外的多种 charset 和 collation，在相互比较时用字符（一个字符可能包含多个字节，取决于 charset 的选择）作为单位。

一个 String Literal 可以拥有一个可选的 `character set introducer` 和 `COLLATE clause`，可以用来指定特定的 charset 和 collation。

```
[_charset_name]'string' [COLLATE collation_name]
```

例如：

```
SELECT _latin1'string';
SELECT _binary'string';
SELECT _utf8'string' COLLATE utf8_bin;
```

你可以使用 `N'literal'` 或者 `n'literal'` 来创建使用 national character set 的字符串，下列语句是一样的：

```
SELECT N'some text';
SELECT n'some text';
SELECT _utf8'some text';
```

要在字符串中表示某些特殊字符，可以利用转义字符进行转义：

转义字符	含义
\0	ASCII NUL (X’00’) 字符
\’	单引号
\“	双引号
\b	退格符号
\n	换行符
\r	回车符
\t	tab 符（制表符）
\z	ASCII 26 (Ctrl + Z)
\\	反斜杠 \
\%	%
_	_

如果要在 ' 包围的字符串中表示 "，或者在 " 包围的字符串中表示 '，可以不使用转义字符。

更多细节见 MySQL 官方文档。

2.1.2.2 Numeric Literals

数值字面值包括 integer 跟 Decimal 类型跟浮点数字面值。

integer 可以包括 . 作为小数点分隔，数字前可以有 - 或者 + 来表示正数或者负数。

精确数值字面值可以表示为如下格式：`1, .2, 3.4, -5, -6.78, +9.10.`

科学记数法也是被允许的，表示为如下格式：`1.2E3, 1.2E-3, -1.2E3, -1.2E-3`。

更多细节见 MySQL 官方文档。

2.1.2.3 Date and Time Literals

Date 跟 Time 字面值有几种格式，例如用字符串表示，或者直接用数字表示。在 TiDB 里面，当 TiDB 期望一个 Date 的时候，它会把 '2017-08-24'，'20170824'，20170824 当做是 Date。

TiDB 的 Date 值有以下几种格式：

- 'YYYY-MM-DD' 或者 'YY-MM-DD'，这里的 - 分隔符并不是严格的，可以是任意的标点符号。比如 '2017-08-24'，'2017&08&24'，'2012@12^31' 都是一样的。唯一需要特别对待的是 '.' 号，它被当做是小数点，用于分隔整数和小数部分。

  Date 和 Time 部分可以被 'T' 分隔，它的作用跟空格符是一样的，例如 2017-8-24 10:42:00 跟 2017-8-24 T10:42:00 是一样的。

- 'YYYYMMDDHHMMSS' 或者 'YYMMDDHHMMSS'，例如 '20170824104520' 和 '170824104520' 被当做是 '2017-08-24 10:45:20'，但是如果你提供了一个超过范围的值，例如'170824304520'，那这就不是一个有效的 Date 字面值。需要注意 YYYYMMDD HHMMSS，YYYYMMDD HH:MM:DD，YYYY-MM-DD HHMMSS 等不正确的格式会插入失败。

- YYYYMMDDHHMMSS 或者 YYMMDDHHMMSS，注意这里没有单引号或者双引号，是一个数字。例如 20170824104520 表示为 '2017-08-24 10:45:20'。

DATETIME 或者 TIMESTAMP 值可以接一个小数部分，用来表示微秒（精度最多到小数点后 6 位），用小数点 . 分隔。

如果 Date 的 year 部分只有两个数字，这是有歧义的（推荐使用四个数字的格式），TiDB 会尝试用以下的规则来解释：

- year 值如果在 70-99 范围，那么被转换成 1970-1999。
- year 值如果在 00-69 范围，那么被转换成 2000-2069。

对于小于 10 的 month 或者 day 值，'2017-8-4' 跟 '2017-08-04' 是一样的。对于 Time 也是一样，比如 '2017-08-24 1:2:3' 跟 '2017-08-24 01:02:03'是一样的。

在需要 Date 或者 Time 的语境下，对于数值，TiDB 会根据数值的长度来选定指定的格式：

- 6 个数字，会被解释为 YYMMDD。
- 12 个数字，会被解释为 YYMMDDHHMMSS。
- 8 个数字，会解释为 YYYYMMDD。
- 14 个数字，会被解释为 YYYYMMDDHHMMSS。

对于 Time 类型，TiDB 用以下格式来表示：

- 'D HH:MM:SS'，或者 'HH:MM:SS'，'HH:MM'，'D HH:MM'，'D HH'，'SS'。这里的 D 表示 days，合法的范围是 0-34。
- 数值 HHMMSS，例如 231010 被解释为'23:10:10'。
- 数值 SS，MMSS，HHMMSS 都是可以被当做 Time。

Time 类型的小数点也是 .，精度最多小数点后 6 位。

更多细节见 MySQL 官方文档。

2.1.2.4 Boolean Literals

常量 TRUE 和 FALSE 等于 1 和 0，它是大小写不敏感的。

```
SELECT TRUE, true, tRuE, FALSE, FaLsE, false;
```

```
+------+------+------+-------+-------+-------+
| TRUE | true | tRuE | FALSE | FaLsE | false |
+------+------+------+-------+-------+-------+
|    1 |    1 |    1 |     0 |     0 |     0 |
+------+------+------+-------+-------+-------+
1 row in set (0.00 sec)
```

2.1.2.5 Hexadecimal Literals

十六进制字面值是有 X 和 0x 前缀的字符串，后接表示十六进制的数字。注意 0x 是大小写敏感的，不能表示为 0X。

例如：

```
X'ac12'
X'12AC'
x'ac12'
x'12AC'
0xac12
0x12AC
```

以下是不合法的十六进制字面值：

```
X'1z' (z 不是合法的十六进制值)
0X12AC (0X 必须用小写的 0x)
```

对于使用 X'val' 格式的十六进制字面值，val 必须包含偶数个字符，如果 val 的长度是奇数（比如 X’A’、X’11A’），可以在前面补一个 0 来避免语法错误。

```
select X'aff';
```

```
ERROR 1105 (HY000): line 0 column 13 near ""hex literal: invalid hexadecimal format, must even
    ↪ numbers, but 3 (total length 13)
```

```
select X'0aff';
```

```
+---------+
| X'0aff' |
+---------+
| 0x0aff  |
+---------+
1 row in set (0.00 sec)
```

默认情况，十六进制字面值是一个二进制字符串。

如果需要将一个字符串或者数字转换为十六进制字面值，可以使用内建函数 HEX()：

```
SELECT HEX('TiDB');
```

```
+-------------+
| HEX('TiDB') |
+-------------+
| 54694442    |
+-------------+
1 row in set (0.01 sec)
```

```
SELECT X'54694442';
```

```
+-------------+
| X'54694442' |
+-------------+
| TiDB        |
+-------------+
1 row in set (0.00 sec)
```

2.1.2.6 Bit-Value Literals

位值字面值用 b 或者 0b 做前缀，后接以 0 和 1 组成的二进制数字。其中 0b 是区分大小写的，0B 则会报错。

合法的 Bit-value：

- b’01’
- B’01’
- 0b01

非法的 Bit-value：

- b'2'（2 不是二进制数值，必须为 0 或 1）
- 0B01（0B 必须是小写 0b）

默认情况，位值字面值是一个二进制字符串。

Bit-value 是作为二进制返回的，所以输出到 MySQL Client 可能会无法显示，如果要转换为可打印的字符，可以使用内建函数 BIN() 或者 HEX()：

```
CREATE TABLE t (b BIT(8));
INSERT INTO t SET b = b'00010011';
INSERT INTO t SET b = b'1110';
INSERT INTO t SET b = b'100101';
```

```
SELECT b+0, BIN(b), HEX(b) FROM t;
```

```
+------+--------+--------+
| b+0  | BIN(b) | HEX(b) |
+------+--------+--------+
|   19 | 10011  | 13     |
|   14 | 1110   | E      |
|   37 | 100101 | 25     |
+------+--------+--------+
3 rows in set (0.00 sec)
```

2.1.2.7 NULL Values

NULL 代表数据为空，它是大小写不敏感的，与 \N (大小写敏感) 同义。

> **注意：**
>
> NULL 跟 0 并不一样，跟空字符串 '' 也不一样。

2.1.3 Schema 对象名

本文介绍 TiDB SQL 语句中的模式对象名。

模式对象名用于命名 TiDB 中所有的模式对象，包括 database、table、index、column、alias 等等。在 SQL 语句中，可以通过标识符 (identifier) 来引用这些对象。

标识符可以被反引号包裹，即 SELECT * FROM t 也可以写成 SELECT * FROM `t`。但如果标识符中存在至少一个特殊符号，或者它是一个保留关键字，那就必须使用反引号包裹来引用它所代表的模式对象。

```
SELECT * FROM `table` WHERE `table`.id = 20;
```

如果 SQL MODE 中设置了 ANSI_QUOTES，那么 TiDB 会将被双引号 " 包裹的字符串识别为 identifier。

```
MySQL [test]> CREATE TABLE "test" (a varchar(10));
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 19 near ""test" (a varchar
    ↪ (10))"

MySQL [test]> SET SESSION sql_mode='ANSI_QUOTES';
Query OK, 0 rows affected (0.000 sec)

MySQL [test]> CREATE TABLE "test" (a varchar(10));
Query OK, 0 rows affected (0.012 sec)
```

如果要在被引用的标识符中使用反引号这个字符，则需要重复两次反引号，例如创建一个表 a ‘b：

```
CREATE TABLE `a``b` (a int);
```

在 select 语句中，alias 部分可以用标识符或者字符串：

```
SELECT 1 AS `identifier`, 2 AS 'string';
```

```
+------------+--------+
| identifier | string |
+------------+--------+
|          1 |      2 |
+------------+--------+
1 row in set (0.00 sec)
```

更多细节，请参考 MySQL 文档。

2.1.3.1 Identifier Qualifiers

Object Names（对象名字）有时可以被限定或者省略。例如在创建表的时候可以省略数据库限定名：

```
CREATE TABLE t (i int);
```

如果之前没有使用 USE 或者连接参数来设定数据库，会报 ERROR 1046 (3D000): No database selected 错误。此时可以指定数据库限定名：

```
CREATE TABLE test.t (i int);
```

. 的左右两端可以出现空格，table_name.col_name 等于 table_name . col_name。

如果要引用这个模式对象，那么请使用：

```
`table_name`.`col_name`
```

而不是：

```
`table_name.col_name`
```

更多细节，请参考 MySQL 文档。

2.1.4 关键字

本文介绍 TiDB 的关键字，对保留字和非保留字作出区分，并汇总所有的关键字以供查询使用。

关键字是 SQL 语句中具有特殊含义的单词，例如 SELECT，UPDATE，DELETE 等等。它们之中有的能够直接作为标识符，被称为非保留关键字（简称非保留字），但有需要经过特殊处理才能作为标识符的字，被称为保留关键字（简称保留字）。

对于保留字，必须使用反引号包裹，才能作为标识符被使用。例如：

```
CREATE TABLE select (a INT);
```

```
ERROR 1105 (HY000): line 0 column 19 near " (a INT)" (total length 27)
```

```
CREATE TABLE `select` (a INT);
```

```
Query OK, 0 rows affected (0.09 sec)
```

而非保留字则不需要反引号也能直接作为标识符。例如 BEGIN 和 END 是非保留字，以下语句能够正常执行：

```
CREATE TABLE `select` (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.09 sec)
```

有一种特殊情况，如果使用了限定符 .，那么也不需要用反引号：

```
CREATE TABLE test.select (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.08 sec)
```

下表列出了 TiDB 中所有的关键字。其中保留字用 (R) 来标识。窗口函数的保留字用 (R-Window) 来标识：

A

- ACCOUNT
- ACTION
- ADD (R)
- ADMIN (R)
- ADVISE
- AFTER
- AGAINST
- AGO
- ALGORITHM
- ALL (R)
- ALTER (R)
- ALWAYS
- ANALYZE (R)
- AND (R)
- ANY
- AS (R)
- ASC (R)
- ASCII
- AUTO_ID_CACHE
- AUTO_INCREMENT
- AUTO_RANDOM

- AUTO_RANDOM_BASE
- AVG
- AVG_ROW_LENGTH

B

- BACKEND
- BACKUP
- BACKUPS
- BEGIN
- BETWEEN (R)
- BIGINT (R)
- BINARY (R)
- BINDING
- BINDINGS
- BINLOG
- BIT
- BLOB (R)
- BLOCK
- BOOL
- BOOLEAN
- BOTH (R)
- BTREE
- BUCKETS (R)
- BUILTINS (R)
- BY (R)
- BYTE

C

- CACHE
- CANCEL (R)
- CAPTURE
- CASCADE (R)
- CASCADED
- CASE (R)
- CHAIN
- CHANGE (R)
- CHAR (R)
- CHARACTER (R)
- CHARSET
- CHECK (R)
- CHECKPOINT
- CHECKSUM
- CIPHER

- CLEANUP
- CLIENT
- CMSKETCH (R)
- COALESCE
- COLLATE (R)
- COLLATION
- COLUMN (R)
- COLUMNS
- COLUMN_FORMAT
- COMMENT
- COMMIT
- COMMITTED
- COMPACT
- COMPRESSED
- COMPRESSION
- CONCURRENCY
- CONFIG
- CONNECTION
- CONSISTENT
- CONSTRAINT (R)
- CONTEXT
- CONVERT (R)
- CPU
- CREATE (R)
- CROSS (R)
- CSV_BACKSLASH_ESCAPE
- CSV_DELIMITER
- CSV_HEADER
- CSV_NOT_NULL
- CSV_NULL
- CSV_SEPARATOR
- CSV_TRIM_LAST_SEPARATORS
- CUME_DIST (R-Window)
- CURRENT
- CURRENT_DATE (R)
- CURRENT_ROLE (R)
- CURRENT_TIME (R)
- CURRENT_TIMESTAMP (R)
- CURRENT_USER (R)
- CYCLE

D

- DATA
- DATABASE (R)

- DATABASES (R)
- DATE
- DATETIME
- DAY
- DAY_HOUR (R)
- DAY_MICROSECOND (R)
- DAY_MINUTE (R)
- DAY_SECOND (R)
- DDL (R)
- DEALLOCATE
- DECIMAL (R)
- DEFAULT (R)
- DEFINER
- DELAYED (R)
- DELAY_KEY_WRITE
- DELETE (R)
- DENSE_RANK (R-Window)
- DEPTH (R)
- DESC (R)
- DESCRIBE (R)
- DIRECTORY
- DISABLE
- DISCARD
- DISK
- DISTINCT (R)
- DISTINCTROW (R)
- DIV (R)
- DO
- DOUBLE (R)
- DRAINER (R)
- DROP (R)
- DUAL (R)
- DUPLICATE
- DYNAMIC

E

- ELSE (R)
- ENABLE
- ENCLOSED (R)
- ENCRYPTION
- END
- ENFORCED
- ENGINE
- ENGINES

- ENUM
- ERROR
- ERRORS
- ESCAPE
- ESCAPED (R)
- EVENT
- EVENTS
- EVOLVE
- EXCEPT (R)
- EXCHANGE
- EXCLUSIVE
- EXECUTE
- EXISTS (R)
- EXPANSION
- EXPIRE
- EXPLAIN (R)
- EXTENDED

F

- FALSE (R)
- FAULTS
- FIELDS
- FILE
- FIRST
- FIRST_VALUE (R-Window)
- FIXED
- FLOAT (R)
- FLUSH
- FOLLOWING
- FOR (R)
- FORCE (R)
- FOREIGN (R)
- FORMAT
- FROM (R)
- FULL
- FULLTEXT (R)
- FUNCTION

G

- GENERAL
- GENERATED (R)
- GLOBAL
- GRANT (R)

- GRANTS
- GROUP (R)
- GROUPS (R-Window)

H

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

I

- IDENTIFIED
- IF (R)
- IGNORE (R)
- IMPORT
- IMPORTS
- IN (R)
- INCREMENT
- INCREMENTAL
- INDEX (R)
- INDEXES
- INFILE (R)
- INNER (R)
- INSERT (R)
- INSERT_METHOD
- INSTANCE
- INT (R)
- INT1 (R)
- INT2 (R)
- INT3 (R)
- INT4 (R)
- INT8 (R)
- INTEGER (R)
- INTERVAL (R)
- INTO (R)
- INVISIBLE
- INVOKER
- IO

- IPC
- IS (R)
- ISOLATION
- ISSUER

J

- JOB (R)
- JOBS (R)
- JOIN (R)
- JSON

K

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

L

- LABELS
- LAG (R-Window)
- LANGUAGE
- LAST
- LASTVAL
- LAST_BACKUP
- LAST_VALUE (R-Window)
- LEAD (R-Window)
- LEADING (R)
- LEFT (R)
- LESS
- LEVEL
- LIKE (R)
- LIMIT (R)
- LINEAR (R)
- LINES (R)
- LIST
- LOAD (R)
- LOCAL
- LOCALTIME (R)
- LOCALTIMESTAMP (R)
- LOCATION
- LOCK (R)
- LOGS

- LONG (R)
- LONGBLOB (R)
- LONGTEXT (R)
- LOW_PRIORITY (R)

M

- MASTER
- MATCH (R)
- MAXVALUE (R)
- MAX_CONNECTIONS_PER_HOUR
- MAX_IDXNUM
- MAX_MINUTES
- MAX_QUERIES_PER_HOUR
- MAX_ROWS
- MAX_UPDATES_PER_HOUR
- MAX_USER_CONNECTIONS
- MB
- MEDIUMBLOB (R)
- MEDIUMINT (R)
- MEDIUMTEXT (R)
- MEMORY
- MERGE
- MICROSECOND
- MINUTE
- MINUTE_MICROSECOND (R)
- MINUTE_SECOND (R)
- MINVALUE
- MIN_ROWS
- MOD (R)
- MODE
- MODIFY
- MONTH

N

- NAMES
- NATIONAL
- NATURAL (R)
- NCHAR
- NEVER
- NEXT
- NEXTVAL
- NO
- NOCACHE

- NOCYCLE
- NODEGROUP
- NODE_ID (R)
- NODE_STATE (R)
- NOMAXVALUE
- NOMINVALUE
- NONE
- NOT (R)
- NOWAIT
- NO_WRITE_TO_BINLOG (R)
- NTH_VALUE (R-Window)
- NTILE (R-Window)
- NULL (R)
- NULLS
- NUMERIC (R)
- NVARCHAR

O

- OFFSET
- ON (R)
- ONLINE
- ONLY
- ON_DUPLICATE
- OPEN
- OPTIMISTIC (R)
- OPTIMIZE (R)
- OPTION (R)
- OPTIONALLY (R)
- OR (R)
- ORDER (R)
- OUTER (R)
- OUTFILE (R)
- OVER (R-Window)

P

- PACK_KEYS
- PAGE
- PARSER
- PARTIAL
- PARTITION (R)
- PARTITIONING
- PARTITIONS
- PASSWORD

- PERCENT_RANK (R-Window)
- PER_DB
- PER_TABLE
- PESSIMISTIC (R)
- PLUGINS
- PRECEDING
- PRECISION (R)
- PREPARE
- PRE_SPLIT_REGIONS
- PRIMARY (R)
- PRIVILEGES
- PROCEDURE (R)
- PROCESS
- PROCESSLIST
- PROFILE
- PROFILES
- PUMP (R)

Q

- QUARTER
- QUERIES
- QUERY
- QUICK

R

- RANGE (R)
- RANK (R-Window)
- RATE_LIMIT
- READ (R)
- REAL (R)
- REBUILD
- RECOVER
- REDUNDANT
- REFERENCES (R)
- REGEXP (R)
- REGION (R)
- REGIONS (R)
- RELEASE (R)
- RELOAD
- REMOVE
- RENAME (R)
- REORGANIZE
- REPAIR

- REPEAT (R)
- REPEATABLE
- REPLACE (R)
- REPLICA
- REPLICATION
- REQUIRE (R)
- RESPECT
- RESTORE
- RESTORES
- RESTRICT (R)
- REVERSE
- REVOKE (R)
- RIGHT (R)
- RLIKE (R)
- ROLE
- ROLLBACK
- ROUTINE
- ROW (R)
- ROWS (R-Window)
- ROW_COUNT
- ROW_FORMAT
- ROW_NUMBER (R-Window)
- RTREE

S

- SAMPLES (R)
- SECOND
- SECONDARY_ENGINE
- SECONDARY_LOAD
- SECONDARY_UNLOAD
- SECOND_MICROSECOND (R)
- SECURITY
- SELECT (R)
- SEND_CREDENTIALS_TO_TIKV
- SEPARATOR
- SEQUENCE
- SERIAL
- SERIALIZABLE
- SESSION
- SET (R)
- SETVAL
- SHARD_ROW_ID_BITS
- SHARE
- SHARED

- SHOW (R)
- SHUTDOWN
- SIGNED
- SIMPLE
- SKIP_SCHEMA_FILES
- SLAVE
- SLOW
- SMALLINT (R)
- SNAPSHOT
- SOME
- SOURCE
- SPATIAL (R)
- SPLIT (R)
- SQL (R)
- SQL_BIG_RESULT (R)
- SQL_BUFFER_RESULT
- SQL_CACHE
- SQL_CALC_FOUND_ROWS (R)
- SQL_NO_CACHE
- SQL_SMALL_RESULT (R)
- SQL_TSI_DAY
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR
- SSL (R)
- START
- STARTING (R)
- STATS (R)
- STATS_AUTO_RECALC
- STATS_BUCKETS (R)
- STATS_HEALTHY (R)
- STATS_HISTOGRAMS (R)
- STATS_META (R)
- STATS_PERSISTENT
- STATS_SAMPLE_PAGES
- STATUS
- STORAGE
- STORED (R)
- STRAIGHT_JOIN (R)
- STRICT_FORMAT
- SUBJECT

- SUBPARTITION
- SUBPARTITIONS
- SUPER
- SWAPS
- SWITCHES
- SYSTEM_TIME

T

- TABLE (R)
- TABLES
- TABLESPACE
- TABLE_CHECKSUM
- TEMPORARY
- TEMPTABLE
- TERMINATED (R)
- TEXT
- THAN
- THEN (R)
- TIDB (R)
- TIFLASH (R)
- TIKV_IMPORTER
- TIME
- TIMESTAMP
- TINYBLOB (R)
- TINYINT (R)
- TINYTEXT (R)
- TO (R)
- TOPN (R)
- TRACE
- TRADITIONAL
- TRAILING (R)
- TRANSACTION
- TRIGGER (R)
- TRIGGERS
- TRUE (R)
- TRUNCATE
- TYPE

U

- UNBOUNDED
- UNCOMMITTED
- UNDEFINED
- UNICODE

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

V

- VALIDATION
- VALUE
- VALUES (R)
- VARBINARY (R)
- VARCHAR (R)
- VARCHARACTER (R)
- VARIABLES
- VARYING (R)
- VIEW
- VIRTUAL (R)
- VISIBLE

W

- WARNINGS
- WEEK
- WEIGHT_STRING
- WHEN (R)
- WHERE (R)
- WIDTH (R)
- WINDOW (R-Window)
- WITH (R)
- WITHOUT
- WRITE (R)

X

- X509
- XOR (R)

Y

- YEAR
- YEAR_MONTH (R)

Z

- ZEROFILL (R)

2.1.5 表达式语法 (Expression Syntax)

表达式是一个或多个值、操作符或函数的组合。在 TiDB 中，表达式主要使用在 SELECT 语句的各个子句中，包括 Group by 子句、Where 子句、Having 子句、Join 条件以及窗口函数等。此外，部分 DDL 语句也会使用到表达式，例如建表时默认值的设置、生成列的设置，分区规则等。

表达式包含几种类型：

- 标识符，可参考模式对象名。
- 谓词、数值、字符串、日期表达式等，这些类型的字面值也是表达式。
- 函数调用，窗口函数等。可参考函数和操作符概述和窗口函数。
- 其他，包括 paramMarker（即 ? ）、系统变量和用户变量、CASE 表达式等。

以下规则是表达式的语法，该语法基于 TiDB parser 的 parser.y 文件中所定义的规则。此外，下列语法图的可导航版本请参考 TiDB SQL 语法图。

```
Expression ::=
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        ↪ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    ↪ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?

PredicateExpr ::=
    BitExpr ( BetweenOrNotOp BitExpr 'AND' BitExpr )* ( InOrNotOp ( '(' ExpressionList ')' |
        ↪ SubSelect ) | LikeOrNotOp SimpleExpr LikeEscapeOpt | RegexpOrNotOp SimpleExpr )?

BitExpr ::=
    BitExpr ( ( '|' | '&' | '<<' | '>>' | '*' | '/' | '%' | 'DIV' | 'MOD' | '^' ) BitExpr | ( '+'
        ↪  | '-' ) ( BitExpr | "INTERVAL" Expression TimeUnit ) )
|   SimpleExpr

SimpleExpr ::=
    SimpleIdent ( ( '->' | '->>' ) stringLit )?
|   FunctionCallKeyword
|   FunctionCallNonKeyword
|   FunctionCallGeneric
```

```
|   SimpleExpr ( 'COLLATE' CollationName | pipes SimpleExpr )
|   WindowFuncCall
|   Literal
|   paramMarker
|   Variable
|   SumExpr
|   ( '!' | '~' | '-' | '+' | 'NOT' | 'BINARY' ) SimpleExpr
|   'EXISTS'? SubSelect
|   ( ( '(' ( ExpressionList ',' )? | 'ROW' '(' ExpressionList ',' ) Expression | builtinCast '('
    ↪  Expression 'AS' CastType | ( 'DEFAULT' | 'VALUES' ) '(' SimpleIdent | 'CONVERT' '('
    ↪ Expression ( ',' CastType | 'USING' CharsetName ) ) ')'
|   'CASE' ExpressionOpt WhenClause+ ElseOpt 'END'
```

2.1.6 注释语法

本文档介绍 TiDB 支持的注释语法。

TiDB 支持三种注释风格：

- 用 # 注释一行：

```
SELECT 1+1;     # 注释文字
```

```
+------+
| 1+1  |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

- 用 -- 注释一行：

```
SELECT 1+1;     -- 注释文字
```

```
+------+
| 1+1  |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

用 -- 注释时，必须要在其之后留出至少一个空格，否则注释不生效：

```
SELECT 1+1--1;
```

```
+--------+
| 1+1--1 |
+--------+
|      3 |
+--------+
1 row in set (0.01 sec)
```

- 用 `/* */` 注释一块，可以注释多行：

```
SELECT 1 /* 这是行内注释文字 */ + 1;
```

```
+--------+
| 1  + 1 |
+--------+
|      2 |
+--------+
1 row in set (0.01 sec)
```

```
SELECT 1+
/*
/*> 这是一条
/*> 多行注释
/*> */
    1;
```

```
+--------------------+
| 1+


        1 |
+--------------------+
|                  2 |
+--------------------+
1 row in set (0.001 sec)
```

2.1.6.1 MySQL 兼容的注释语法

TiDB 也跟 MySQL 保持一致，支持一种 C 风格注释的变体：

```
/*! Specific code */
```

或者

```
/*!50110 Specific code */
```

和 MySQL 一样，TiDB 会执行注释中的语句。

例如：`SELECT /*! STRAIGHT_JOIN */ col1 FROM table1,table2 WHERE ...`

在 TiDB 中，这种写法等价于 `SELECT STRAIGHT_JOIN col1 FROM table1,table2 WHERE ...`

如果注释中指定了 Server 版本号，例如 `/*!50110 KEY_BLOCK_SIZE=1024 */`，在 MySQL 中表示只有 MySQL 的版本大于等于 5.1.10 才会处理这个 comment 中的内容。但是在 TiDB 中，这个 MySQL 版本号不会起作用，所有的 comment 都被会处理。

2.1.6.2 TiDB 可执行的注释语法

TiDB 也有独立的注释语法，称为 TiDB 可执行注释语法。主要分为两种：

- `/*T! Specific code */`：该语法只能被 TiDB 解析执行，而在其他数据库中会被忽略。
- `/*T![feature_id] Specific code */`：该语法用于保证 TiDB 不同版本之间的兼容性。只有在当前版本中实现了 `feature_id` 对应的功能特性的 TiDB，才会试图解析该注释里的 SQL 片段。例如 v3.1.1 中引入了 AUTO_RANDOM 特性，该版本能够将 `/*T![auto_rand] auto_random */` 解析为 `auto_random`；而 v3.0.0 中没有实现 AUTO_RANDOM 特性，则上述 SQL 语句片段会被忽略。注意前几个字符 `/*T![` 中，各字符之间没有任何空格。

2.1.6.3 优化器注释语法

还有一种注释会被当做是优化器 Hint 特殊对待：

```
SELECT /*+ hint */ FROM ...;
```

TiDB 支持的相关优化器 hint 详见Optimizer Hints。

> 注意：
>
> 在 MySQL 客户端中，TiDB 可执行注释语法会被默认当成注释被清除掉。在 MySQL 客户端 5.7.7 之前的版本中，Hint 也会被默认当成注释被清除掉。推荐在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

更多细节，请参考 MySQL 文档。

2.2 SQL 语句

2.2.1 ADD COLUMN

`ALTER TABLE.. ADD COLUMN` 语句用于在已有表中添加列。在 TiDB 中，`ADD COLUMN` 为在线操作，不会阻塞表中的数据读写。

2.2.1.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AddColumnSpec ( ',' AddColumnSpec )*

TableName ::=
    Identifier ('.' Identifier)?

AddColumnSpec
         ::= 'ADD' 'COLUMN' 'IF NOT EXISTS'? ColumnName ColumnType ColumnOption+ ( 'FIRST' | '
             ↪ AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               ↪ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               ↪ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
           | 'STORAGE' StorageMedia
           | 'AUTO_RANDOM' ( '(' LengthNum ')' )?
```

2.2.1.2 示例

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

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

```
INSERT INTO t1 VALUES (NULL);
```

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

```
SELECT * FROM t1;
```

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

```
ALTER TABLE t1 ADD COLUMN c1 INT NOT NULL;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  0 |
+----+----+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD c2 INT NOT NULL AFTER c1;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+----+
| id | c1 | c2 |
+----+----+----+
|  1 |  0 |  0 |
+----+----+----+
1 row in set (0.00 sec)
```

2.2.1.3 MySQL 兼容性

- 不支持在一条语句中同时添加多列。
- 不支持将新添加的列设为 PRIMARY KEY。
- 不支持将新添加的列设为 AUTO_INCREMENT。
- 对添加生成列有局限性，具体可参考：生成列局限性。

2.2.1.4 另请参阅

- ADD INDEX
- CREATE TABLE

2.2.2 ADD INDEX

`ALTER TABLE.. ADD INDEX` 语句用于在已有表中添加一个索引。在 TiDB 中，`ADD INDEX` 为在线操作，不会阻塞表中的数据读写。

> **警告：**
>
> - 在升级 TiDB 集群的过程中，请勿执行 DDL 语句，否则可能会出现行为未定义的问题。
> - 集群中有 DDL 语句正在被执行时（通常为 `ADD INDEX` 和列类型变更等耗时较久的 DDL 语句），请勿进行升级操作。在升级前，建议使用`ADMIN SHOW DDL` 命令查看集群中是否有正在进行的 DDL Job。如需升级，请等待 DDL 执行完成或使用`ADMIN CANCEL DDL` 命令取消该 DDL Job 后再进行升级。

2.2.2.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AddIndexSpec ( ',' AddIndexSpec )*

AddIndexSpec
         ::= 'ADD' ( ( 'PRIMARY' 'KEY' | ( 'KEY' | 'INDEX' ) 'IF NOT EXISTS'? | 'UNIQUE' ( 'KEY'
             ↪ | 'INDEX' )? ) ( ( Identifier? 'USING' | Identifier 'TYPE' ) IndexType )? | '
             ↪ FULLTEXT' ( 'KEY' | 'INDEX' )? IndexName ) '(' IndexPartSpecification ( ','
             ↪ IndexPartSpecification )* ')' IndexOption*

IndexPartSpecification
         ::= ( ColumnName ( '(' LengthNum ')' )? | '(' Expression ')' ) ( 'ASC' | 'DESC' )

IndexOption
         ::= 'KEY_BLOCK_SIZE' '='? LengthNum
           | IndexType
           | 'WITH' 'PARSER' Identifier
           | 'COMMENT' stringLit
           | 'VISIBLE'
           | 'INVISIBLE'

IndexType
         ::= 'BTREE'
           | 'HASH'
```

```
          | 'RTREE'
```

2.2.2.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+------------------------+---------+-----------+-----------------------+-------------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                              |
+------------------------+---------+-----------+-----------------------+-------------------------------------
    ↪
```

```
| IndexReader_6           | 0.01    | root      |                         | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5      | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-------------------------+---------+-----------+------------------------+-----------------------------------------
    ↪
2 rows in set (0.00 sec)
```

2.2.2.3 MySQL 兼容性

- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。
- 不支持降序索引（类似于 MySQL 5.7）。
- 目前尚不支持在一条中同时添加多个索引。
- 无法向表中添加 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

2.2.2.4 另请参阅

- 索引的选择
- 错误索引的解决方案
- CREATE INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

2.2.3 ADMIN

ADMIN 语句是 TiDB 扩展语法，用于查看 TiDB 自身的状态，并对 TiDB 中的表数据进行校验。

2.2.3.1 ADMIN 与 DDL 相关的扩展语句

语句	功能描述
ADMIN CANCEL DDL JOBS	取消当前正在运行的 DDL 作业
ADMIN CHECKSUM TABLE	计算表中所有行和索引的 CRC64 校验和
[ADMIN CHECK [TABLE|INDEX]](#admin-check-[table	index])
[ADMIN SHOW DDL [JOBS|QUERIES]](#admin-show-ddl-[jobs	queries])

2.2.3.2 admin reload 语句

```
ADMIN RELOAD expr_pushdown_blacklist;
```

以上语句用于重新加载表达式下推的黑名单。

```
ADMIN RELOAD opt_rule_blacklist;
```

以上语句用于重新加载逻辑优化规则的黑名单。

2.2.3.3 admin plugin 语句

```
ADMIN PLUGINS ENABLE plugin_name [, plugin_name] ...;
```

以上语句用于启用 plugin_name 插件。

```
ADMIN PLUGINS DISABLE plugin_name [, plugin_name] ...;
```

以上语句用于禁用 plugin_name 插件。

2.2.3.4 admin ... bindings 语句

```
ADMIN FLUSH bindings;
```

以上语句用于持久化 SQL Plan 绑定的信息。

```
ADMIN CAPTURE bindings;
```

以上语句可以将出现超过一次的 selectexecution-plan 语句生成 SQL Plan 的绑定。

```
ADMIN EVOLVE bindings;
```

开启自动绑定功能后，每隔 bind-info-lease（默认值为 3s）触发一次 SQL Plan 绑定信息的演进。以上语句用于主动触发此演进，SQL Plan 绑定详情可参考：执行计划管理。

```
ADMIN RELOAD bindings;
```

以上语句用于重新加载 SQL Plan 绑定的信息。

2.2.3.5 admin repair table 语句

```
ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT;
```

ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT 用于在极端情况下，对存储层中的表的元信息进行非可信的覆盖。“非可信”是指需要人为保证原表的元信息可以完全由 CREATE TABLE STATEMENT 提供。该语句需要打开配置文件项中的repair-mode 开关，并且需要确保所修复的表名在repair-table-list 名单中。

2.2.3.6 admin show slow 语句

```
ADMIN SHOW SLOW RECENT N;
```

```
ADMIN SHOW SLOW TOP [INTERNAL | ALL] N;
```

这两种语句的具体操作详情可参考：admin show slow 语句。

2.2.3.7 语句概览

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

2.2.3.8 使用示例

执行以下命令，可查看正在执行的 DDL 任务中最近 10 条已经完成的 DDL 任务。未指定 NUM 时，默认只显示最近 10 条已经执行完的 DDL 任务。

```
admin show ddl jobs;
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                         | END_TIME                           | STATE
    ↪           |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1         | add index     | write reorganization | 32        | 37       | 0
    ↪           | 2019-01-10 12:38:36.501 +0800 CST |                                    |
    ↪ running       |
| 44     | test    | t1         | add index     | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
| 43     | test    | t1         | add index     | public               | 32        | 37       | 6
    ↪           | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    ↪         |
| 42     | test    | t1         | drop index    | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    ↪         |
```

```
| 41     | test    | t1         | add index     | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    ↪         |
| 40     | test    | t1         | drop column   | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    ↪         |
| 39     | test    | t1         | add column    | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:32:55.42 +0800 CST  | 2019-01-10 12:32:56.24 +0800 CST  | synced
    ↪         |
| 38     | test    | t1         | create table  | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:32:41.956 +0800 CST | 2019-01-10 12:32:43.956 +0800 CST | synced
    ↪         |
| 36     | test    |            | drop table    | none                 | 32        | 34       | 0
    ↪          | 2019-01-10 11:29:59.982 +0800 CST | 2019-01-10 11:30:00.45 +0800  CST | synced
    ↪         |
| 35     | test    |            | create table  | public               | 32        | 34       | 0
    ↪          | 2019-01-10 11:29:40.741 +0800 CST | 2019-01-10 11:29:41.682 +0800 CST | synced
    ↪         |
| 33     | test    |            | create schema | public               | 32        | 0        | 0
    ↪          | 2019-01-10 11:29:22.813 +0800 CST | 2019-01-10 11:29:23.954 +0800 CST | synced
    ↪         |
+--------+---------+------------+---------------------+----------------+-----------+----------+-----------+------
    ↪
```

执行以下命令，可查看正在执行的 DDL 任务中最近 5 条已经执行完的 DDL 任务：

```
admin show ddl jobs 5;
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                         | END_TIME                           | STATE
    ↪          |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1         | add index     | write reorganization | 32        | 37       | 0
    ↪          | 2019-01-10 12:38:36.501 +0800 CST |                                    |
    ↪ running       |
| 44     | test    | t1         | add index     | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
| 43     | test    | t1         | add index     | public               | 32        | 37       | 6
    ↪          | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    ↪         |
| 42     | test    | t1         | drop index    | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
```

```
    ↪          |
| 41     | test    | t1          | add index      | public                | 32        | 37       | 0
    ↪           | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    ↪          |
| 40     | test    | t1          | drop column    | none                  | 32        | 37       | 0
    ↪           | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    ↪          |
+--------+---------+------------+--------------------+---------------+-----------+----------+-----------+------
    ↪
```

执行以下命令，可查看 test 数据库中未执行完成的 DDL 任务，包括正在执行中以及最近 5 条已经执行完但是执行失败的 DDL 任务。

```
admin show ddl jobs 5 where state!='synced' and db_name='test';
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                         | END_TIME                           | STATE
    ↪           |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1          | add index     | write reorganization | 32        | 37       | 0
    ↪           | 2019-01-10 12:38:36.501 +0800 CST |                                    |
    ↪ running       |
| 44     | test    | t1          | add index     | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
```

- JOB_ID：每个 DDL 操作对应一个 DDL 作业，JOB_ID 全局唯一。
- DB_NAME：执行 DDL 操作的数据库的名称。
- TABLE_NAME：执行 DDL 操作的表的名称。
- JOB_TYPE：DDL 操作的类型。
- SCHEMA_STATE：schema 的当前状态。如果 JOB_TYPE 是 add index，则为 index 的状态；如果是 add column，则为 column 的状态，如果是 create table，则为 table 的状态。常见的状态有以下几种：
  - none：表示不存在。一般 drop 操作或者 create 操作失败回滚后，会变为 none 状态。
  - delete only、write only、delete reorganization、write reorganization：这四种状态是中间状态。由于中间状态转换很快，一般操作中看不到这几种状态，只有执行 add index 操作时能看到处于 write reorganization 状态，表示正在添加索引数据。
  - public：表示存在且可用。一般 create table 和 add index/column 等操作完成后，会变为 public 状态，表示新建的 table/column/index 可以正常读写了。
- SCHEMA_ID：执行 DDL 操作的数据库的 ID。

- TABLE_ID：执行 DDL 操作的表的 ID。
- ROW_COUNT：执行 add index 操作时，当前已经添加完成的数据行数。
- START_TIME：DDL 操作的开始时间。
- END_TIME：DDL 操作的结束时间。
- STATE：DDL 操作的状态。常见的状态有以下几种：
  - none：表示该操作任务已经进入 DDL 作业队列中，但尚未执行，因为还在排队等待前面的 DDL 作业完成。另一种原因可能是执行 drop 操作后，会变为 none 状态，但是很快会更新为 synced 状态，表示所有 TiDB 实例都已经同步到该状态。
  - running：表示该操作正在执行。
  - synced：表示该操作已经执行成功，且所有 TiDB 实例都已经同步该状态。
  - rollback done：表示该操作执行失败，回滚完成。
  - rollingback：表示该操作执行失败，正在回滚。
  - cancelling：表示正在取消该操作。这个状态只有在用 ADMIN CANCEL DDL JOBS 命令取消 DDL 作业时才会出现。

2.2.3.9 MySQL 兼容性

ADMIN 语句是 TiDB 对于 MySQL 语法的扩展。

2.2.4 ADMIN CANCEL DDL

ADMIN CANCEL DDL 语句用于取消当前正在运行的 DDL 作业。可以通过 ADMIN SHOW DDL JOBS 语句获取 DDL 作业的 job_id。

2.2.4.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

NumList ::=
    Int64Num ( ',' Int64Num )*
```

2.2.4.2 示例

可以通过 ADMIN CANCEL DDL JOBS 语句取消当前正在运行的 DDL 作业，并返回对应作业是否取消成功：

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

如果取消失败，会显示失败的具体原因。

> 注意：
>
> - 只有该操作可以取消 DDL 作业，其他所有的操作和环境变更（例如机器重启、集群重启）都不会取消 DDL 作业。
> - 该操作可以同时取消多个 DDL 作业，可以通过 `ADMIN SHOW DDL JOBS` 语句来获取 DDL 作业的 `job_id`。
> - 如果希望取消的作业已经执行完毕，取消操作将失败。

2.2.4.3 MySQL 兼容性

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

2.2.4.4 另请参阅

- `ADMIN SHOW DDL [JOBS|QUERIES]`

2.2.5 ADMIN CHECKSUM TABLE

`ADMIN CHECKSUM TABLE` 语句用于计算表中所有行和索引的 CRC64 校验和。在 TiDB Lightning 等程序中，可通过此语句来确保导入操作成功。

2.2.5.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

TableNameList ::=
    TableName ( ',' TableName )*
```

2.2.5.2 示例

计算表 `t1` 的校验和：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment);
INSERT INTO t1 VALUES (1),(2),(3);
ADMIN CHECKSUM TABLE t1;
```

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment);
Query OK, 0 rows affected (0.11 sec)

INSERT INTO t1 VALUES (1),(2),(3);
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0

ADMIN CHECKSUM TABLE t1;
+---------+------------+---------------------+-----------+-------------+
| Db_name | Table_name | Checksum_crc64_xor  | Total_kvs | Total_bytes |
+---------+------------+---------------------+-----------+-------------+
| test    | t1         | 10909174369497628533 |         3 |          75 |
+---------+------------+---------------------+-----------+-------------+
1 row in set (0.00 sec)
```

2.2.5.3 MySQL 兼容性

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

2.2.6 ADMIN CHECK [TABLE|INDEX]

`ADMIN CHECK [TABLE|INDEX]` 语句用于校验表中数据和对应索引的一致性。

2.2.6.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

TableNameList ::=
    TableName ( ',' TableName )*
```

2.2.6.2 示例

可以通过 ADMIN CHECK TABLE 语句校验 tbl_name 表中所有数据和对应索引的一致性：

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

若通过一致性校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name;
```

以上语句用于对 tbl_name 表中 idx_name 索引对应列数据和索引数据进行一致性校验。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name (lower_val, upper_val) [, (lower_val, upper_val)] ...;
```

以上语句用于对 tbl_name 表中 idx_name 索引对应列数据和索引数据进行一致性校验，并且指定了需要检查的数据范围。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

2.2.6.3 MySQL 兼容性

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

2.2.6.4 另请参阅

- ADMIN REPAIR

2.2.7 ADMIN SHOW DDL [JOBS|QUERIES]

ADMIN SHOW DDL [JOBS|QUERIES] 语句显示了正在运行和最近完成的 DDL 作业的信息。

2.2.7.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

NumList ::=
    Int64Num ( ',' Int64Num )*

WhereClauseOptional ::=
    WhereClause?
```

2.2.7.2 示例

2.2.7.2.1 ADMIN SHOW DDL

可以通过 ADMIN SHOW DDL 语句查看当前正在运行的 DDL 作业：

```
ADMIN SHOW DDL;
```

```
ADMIN SHOW DDL;
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-----------------------
    ↪
| SCHEMA_VER | OWNER_ID                             | OWNER_ADDRESS | RUNNING_JOBS | SELF_ID
    ↪                               | QUERY |
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-----------------------
    ↪
|         26 | 2d1982af-fa63-43ad-a3d5-73710683cc63 | 0.0.0.0:4000  |              | 2d1982af-
    ↪ fa63-43ad-a3d5-73710683cc63 |       |
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-----------------------
    ↪
1 row in set (0.00 sec)
```

2.2.7.2.2 ADMIN SHOW DDL JOBS

ADMIN SHOW DDL JOBS 语句用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。

```
ADMIN SHOW DDL JOBS;
```

```
ADMIN SHOW DDL JOBS;
+--
    ↪ ------+---------+-------------------+--------------+---------------------+-----------+----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME         | JOB_TYPE     | SCHEMA_STATE          | SCHEMA_ID |
    ↪ TABLE_ID | ROW_COUNT | CREATE_TIME          | START_TIME          | END_TIME            |
    ↪ STATE    |
+--
    ↪ ------+---------+-------------------+--------------+---------------------+-----------+----------+------
    ↪
|     59 | test    | t1                 | add index    | write reorganization |         1 |
    ↪       55 |     88576 | 2020-08-17 07:51:58 | 2020-08-17 07:51:58 | NULL                |
    ↪ running |
|     60 | test    | t2                 | add index    | none                 |         1 |
    ↪       57 |         0 | 2020-08-17 07:51:59 | 2020-08-17 07:51:59 | NULL                |
    ↪ none     |
```

```
|     58 | test    | t2                 | create table | public               |         1 |
    ↪       57 |         0 | 2020-08-17 07:41:28 | 2020-08-17 07:41:28 | 2020-08-17 07:41:28 |
    ↪ synced  |
|     56 | test    | t1                 | create table | public               |         1 |
    ↪       55 |         0 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 |
    ↪ synced  |
|     54 | test    | t1                 | drop table   | none                 |         1 |
    ↪       50 |         0 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 |
    ↪ synced  |
|     53 | test    | t1                 | drop index   | none                 |         1 |
    ↪       50 |         0 | 2020-08-17 07:35:44 | 2020-08-17 07:35:44 | 2020-08-17 07:35:44 |
    ↪ synced  |
|     52 | test    | t1                 | add index    | public               |         1 |
    ↪       50 |    451010 | 2020-08-17 07:34:43 | 2020-08-17 07:34:43 | 2020-08-17 07:35:16 |
    ↪ synced  |
|     51 | test    | t1                 | create table | public               |         1 |
    ↪       50 |         0 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 |
    ↪ synced  |
|     49 | test    | t1                 | drop table   | none                 |         1 |
    ↪       47 |         0 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 |
    ↪ synced  |
|     48 | test    | t1                 | create table | public               |         1 |
    ↪       47 |         0 | 2020-08-17 07:33:37 | 2020-08-17 07:33:37 | 2020-08-17 07:33:37 |
    ↪ synced  |
|     46 | mysql   | stats_extended     | create table | public               |         3 |
    ↪       45 |         0 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 |
    ↪ synced  |
|     44 | mysql   | opt_rule_blacklist | create table | public               |         3 |
    ↪       43 |         0 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 |
    ↪ synced  |
+--
    ↪ ------+---------+--------------------+--------------+----------------------+-----------+----------+------
    ↪
12 rows in set (0.00 sec)
```

由上述 ADMIN 查询结果可知：

- job_id 为 59 的 DDL 作业当前正在进行中（STATE 列显示为 running）。SCHEMA_STATE 列显示了表当前处于 write reorganization 状态，一旦任务完成，将更改为 public，以便用户会话可以公开观察到状态变更。end_time 列显示为 NULL，表明当前作业的完成时间未知。
- job_id 为 60 的 JOB_TYPE 显示为 add index，表明正在排队等待 job_id 为 59 的作业完成。当作业 59 完成时，作业 60 的 STATE 将更改为 running。
- 对于破坏性的更改（例如删除索引或删除表），当作业完成时，SCHEMA_STATE 将变为 none。对于附加更改，SCHEMA_STATE 将变为 public。

若要限制表中显示的行数，可以指定 NUM 和 WHERE 条件：

```
ADMIN SHOW DDL JOBS [NUM] [WHERE where_condition];
```

- NUM：用于查看已经执行完成的 DDL 作业队列中最近 NUM 条结果；未指定时，默认值为 10。
- WHERE：WHERE 子句，用于添加过滤条件。

2.2.7.2.3 ADMIN SHOW DDL JOB QUERIES

ADMIN SHOW DDL JOB QUERIES 语句用于查看 job_id 对应的 DDL 任务的原始 SQL 语句：

```
ADMIN SHOW DDL JOBS;
ADMIN SHOW DDL JOB QUERIES 51;
```

```
ADMIN SHOW DDL JOB QUERIES 51;
+--------------------------------------------------------------+
| QUERY                                                        |
+--------------------------------------------------------------+
| CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment) |
+--------------------------------------------------------------+
1 row in set (0.02 sec)
```

只能在 DDL 历史作业队列中最近十条结果中搜索与 job_id 对应的正在运行中的 DDL 作业。

2.2.7.3 MySQL 兼容性

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

2.2.7.4 另请参阅

- ADMIN CANCEL DDL

2.2.8 ADMIN SHOW TELEMETRY

ADMIN SHOW TELEMETRY 语句用于查看通过遥测功能收集到并分享给 PingCAP 的使用信息。

2.2.8.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow | 'TELEMETRY' ) | 'CHECK' ( 'TABLE'
        ↪ TableNameList | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? )
        ↪ | 'RECOVER' 'INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier |
        ↪  'TABLE' 'LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' '
        ↪ JOBS' NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | '
        ↪ BINDINGS' ) | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE'
        ↪ TableName CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

2.2.8.2 示例

```
ADMIN SHOW TELEMETRY\G
```

```
*************************** 1. row ***************************
 TRACKING_ID: a1ba1d97-b940-4d5b-a9d5-ddb0f2ac29e7
 LAST_STATUS: {
  "check_at": "2021-08-11T08:23:38+02:00",
  "is_error": false,
  "error_msg": "",
  "is_request_sent": true
}
DATA_PREVIEW: {
  "hardware": [
    {
      "instanceType": "tidb",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "4000",
      "cpu": {
        "cache": "8192",
        "cpuFrequency": "2301.00MHz",
        "cpuLogicalCores": "8",
        "cpuPhysicalCores": "4"
      },
      "memory": {
        "capacity": "16410021888"
      },
      "disk": {
        "ebbca862689fa9fef7c55c3112e375c4ce575fe4": {
          "deviceName": "ebbca862689fa9fef7c55c3112e375c4ce575fe4",
          "free": "624438726656",
          "freePercent": "0.61",
          "fstype": "btrfs",
          "opts": "bind,rw,relatime",
          "path": "fb365c1216b59e1cfc86950425867007a60f4435",
          "total": "1022488477696",
          "used": "397115568128",
          "usedPercent": "0.39"
        },
        "nvme0n1p1": {
          "deviceName": "nvme0n1p1",
          "free": "582250496",
          "freePercent": "0.93",
          "fstype": "vfat",
          "opts": "rw,relatime",
          "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
```

```
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2": {
        "deviceName": "nvme0n1p2",
        "free": "701976576",
        "freePercent": "0.74",
        "fstype": "ext4",
        "opts": "rw,relatime",
        "path": "/boot",
        "total": "1023303680",
        "used": "250863616",
        "usedPercent": "0.26"
      }
    }
  },
  {
    "instanceType": "pd",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "2379",
    "cpu": {
      "cache": "8192",
      "cpuFrequency": "2301.00MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4"
    },
    "memory": {
      "capacity": "16410021888"
    },
    "disk": {
      "ebbca862689fa9fef7c55c3112e375c4ce575fe4": {
        "deviceName": "ebbca862689fa9fef7c55c3112e375c4ce575fe4",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "opts": "bind,rw,relatime",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "397115568128",
        "usedPercent": "0.39"
      },
      "nvme0n1p1": {
        "deviceName": "nvme0n1p1",
        "free": "582250496",
```

```
        "freePercent": "0.93",
        "fstype": "vfat",
        "opts": "rw,relatime",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2": {
        "deviceName": "nvme0n1p2",
        "free": "701976576",
        "freePercent": "0.74",
        "fstype": "ext4",
        "opts": "rw,relatime",
        "path": "/boot",
        "total": "1023303680",
        "used": "250863616",
        "usedPercent": "0.26"
      }
    }
  },
  {
    "instanceType": "tikv",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "20160",
    "cpu": {
      "cpuFrequency": "3730MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4",
      "cpuVendorId": "GenuineIntel",
      "l1CacheLineSize": "64",
      "l1CacheSize": "32768",
      "l2CacheLineSize": "64",
      "l2CacheSize": "262144",
      "l3CacheLineSize": "64",
      "l3CacheSize": "8388608"
    },
    "memory": {
      "capacity": "16803861504"
    },
    "disk": {
      "36e7dfacbb83843f83075d78aeb4cf850a4882a1": {
        "deviceName": "36e7dfacbb83843f83075d78aeb4cf850a4882a1",
        "free": "624438726656",
        "freePercent": "0.61",
```

```
        "fstype": "btrfs",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "398049751040",
        "usedPercent": "0.39"
      },
      "nvme0n1p1": {
        "deviceName": "nvme0n1p1",
        "free": "582250496",
        "freePercent": "0.93",
        "fstype": "vfat",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2": {
        "deviceName": "nvme0n1p2",
        "free": "701976576",
        "freePercent": "0.69",
        "fstype": "ext4",
        "path": "/boot",
        "total": "1023303680",
        "used": "321327104",
        "usedPercent": "0.31"
      }
    }
  },
  {
    "instanceType": "tiflash",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "3930",
    "cpu": {
      "cpuFrequency": "3400MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4",
      "l1CacheLineSize": "64",
      "l1CacheSize": "32768",
      "l2CacheLineSize": "64",
      "l2CacheSize": "262144",
      "l3CacheLineSize": "64",
      "l3CacheSize": "8388608"
    },
    "memory": {
      "capacity": "16410021888"
```

```
    },
    "disk": {
      "36e7dfacbb83843f83075d78aeb4cf850a4882a1": {
        "deviceName": "36e7dfacbb83843f83075d78aeb4cf850a4882a1",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "398049751040",
        "usedPercent": "0.39"
      }
    }
  }
],
"instances": [
  {
    "instanceType": "tidb",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "4000",
    "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "statusPort": "10080",
    "version": "5.1.1",
    "gitHash": "797bddd25310ed42f0791c8eccb78be8cce2f502",
    "startTime": "2021-08-11T08:23:38+02:00",
    "upTime": "22.210217487s"
  },
  {
    "instanceType": "pd",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "2379",
    "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "statusPort": "2379",
    "version": "5.1.1",
    "gitHash": "7cba1912b317a533e18b16ea2ba9a14ed2891129",
    "startTime": "2021-08-11T08:23:32+02:00",
    "upTime": "28.210220368s"
  },
  {
    "instanceType": "tikv",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "20160",
    "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "statusPort": "20180",
    "version": "5.1.1",
```

```
      "gitHash": "4705d7c6e9c42d129d3309e05911ec6b08a25a38",
      "startTime": "2021-08-11T08:23:33+02:00",
      "upTime": "27.210221447s"
    },
    {
      "instanceType": "tiflash",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "3930",
      "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "statusPort": "20292",
      "version": "v5.1.1",
      "gitHash": "c8fabfb50fe28db17cc5118133a69be255c40efd",
      "startTime": "2021-08-11T08:23:40+02:00",
      "upTime": "20.210222452s"
    }
  ],
  "hostExtra": {
    "cpuFlags": [
      "fpu",
      "vme",
      "de",
      "pse",
      "tsc",
      "msr",
      "pae",
      "mce",
      "cx8",
      "apic",
      "sep",
      "mtrr",
      "pge",
      "mca",
      "cmov",
      "pat",
      "pse36",
      "clflush",
      "dts",
      "acpi",
      "mmx",
      "fxsr",
      "sse",
      "sse2",
      "ss",
      "ht",
      "tm",
```

```
        "pbe",
        "syscall",
        "nx",
        "pdpe1gb",
        "rdtscp",
        "lm",
        "constant_tsc",
        "art",
        "arch_perfmon",
        "pebs",
        "bts",
        "rep_good",
        "nopl",
        "xtopology",
        "nonstop_tsc",
        "cpuid",
        "aperfmperf",
        "pni",
        "pclmulqdq",
        "dtes64",
        "monitor",
        "ds_cpl",
        "vmx",
        "est",
        "tm2",
        "ssse3",
        "sdbg",
        "fma",
        "cx16",
        "xtpr",
        "pdcm",
        "pcid",
        "sse4_1",
        "sse4_2",
        "x2apic",
        "movbe",
        "popcnt",
        "tsc_deadline_timer",
        "aes",
        "xsave",
        "avx",
        "f16c",
        "rdrand",
        "lahf_lm",
        "abm",
```

```
        "3dnowprefetch",
        "cpuid_fault",
        "epb",
        "invpcid_single",
        "ssbd",
        "ibrs",
        "ibpb",
        "stibp",
        "ibrs_enhanced",
        "tpr_shadow",
        "vnmi",
        "flexpriority",
        "ept",
        "vpid",
        "ept_ad",
        "fsgsbase",
        "tsc_adjust",
        "sgx",
        "bmi1",
        "avx2",
        "smep",
        "bmi2",
        "erms",
        "invpcid",
        "mpx",
        "rdseed",
        "adx",
        "smap",
        "clflushopt",
        "intel_pt",
        "xsaveopt",
        "xsavec",
        "xgetbv1",
        "xsaves",
        "dtherm",
        "ida",
        "arat",
        "pln",
        "pts",
        "hwp",
        "hwp_notify",
        "hwp_act_window",
        "hwp_epp",
        "md_clear",
        "flush_l1d",
```

```
      "arch_capabilities"
    ],
    "cpuModelName": "Intel(R) Core(TM) i7-1051OU CPU @ 1.8OGHz",
    "os": "linux",
    "platform": "fedora",
    "platformFamily": "fedora",
    "platformVersion": "34",
    "kernelVersion": "5.13.5-200.fc34.x86_64",
    "kernelArch": "x86_64",
    "virtualizationSystem": "kvm",
    "virtualizationRole": "host"
  },
  "reportTimestamp": 1628663040,
  "trackingId": "a1ba1d97-b940-4d5b-a9d5-ddb0f2ac29e7",
  "featureUsage": {
    "txn": {
      "asyncCommitUsed": true,
      "onePCUsed": true,
      "txnCommitCounter": {
        "twoPC": 9,
        "asyncCommit": 0,
        "onePC": 0
      }
    },
    "clusterIndex": {},
    "temporaryTable": false,
    "cte": {
      "nonRecursiveCTEUsed": 0,
      "recursiveUsed": 0,
      "nonCTEUsed": 13
    }
  },
  "windowedStats": [],
  "slowQueryStats": {
    "slowQueryBucket": {}
  }
}
1 row in set (0.0259 sec)
```

2.2.8.3 MySQL 兼容性

`ADMIN` 语句是 TiDB 对 MySQL 语法的扩展。

2.2.8.4 另请参阅

- 遥测
- 系统变量 tidb_enable_telemetry

2.2.9 ALTER DATABASE

ALTER DATABASE 用于修改指定或当前数据库的默认字符集和排序规则。ALTER SCHEMA 跟 ALTER DATABASE 操作效果一样。

2.2.9.1 语法图

```
AlterDatabaseStmt ::=
    'ALTER' 'DATABASE' DBName? DatabaseOptionList

DatabaseOption ::=
    DefaultKwdOpt ( CharsetKw '='? CharsetName | 'COLLATE' '='? CollationName | 'ENCRYPTION' '='?
        ↪ EncryptionOpt )
```

2.2.9.2 示例

修改数据库 test 的字符集为 utf8mb4：

```
ALTER DATABASE test DEFAULT CHARACTER SET = utf8mb4;
```

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

目前 TiDB 只支持部分的字符集和排序规则，详情参阅字符集支持。

2.2.9.3 MySQL 兼容性

ALTER DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.9.4 另请参阅

- CREATE DATABASE
- SHOW DATABASES

2.2.10 ALTER INDEX

ALTER INDEX 语句用于修改索引的可见性，可以将索引设置为 Visible 或者 Invisible。设置为 Invisible 的索引即不可见索引 (Invisible Index) 由 DML 语句维护，不会被查询优化器使用。

2.2.10.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AlterIndexSpec ( ',' AlterIndexSpec )*

AlterIndexSpec
         ::= 'ALTER' 'INDEX' Identifier ( 'VISIBLE' | 'INVISIBLE' )
```

2.2.10.2 示例

可以通过 `ALTER TABLE ... ALTER INDEX ...` 语句，修改索引的可见性：

```
CREATE TABLE t1 (c1 INT, UNIQUE(c1));
ALTER TABLE t1 ALTER INDEX c1 INVISIBLE;
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
                                      |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `c1` int(11) DEFAULT NULL,
  UNIQUE KEY `c1` (`c1`) /*!80000 INVISIBLE */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

优化器将无法使用 `c1` 这个不可见的索引：

```
EXPLAIN SELECT c1 FROM t1 ORDER BY c1;
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| Sort_4                  | 10000.00 | root      |               | test.t1.c1:asc
    ↪                 |
| └─TableReader_8         | 10000.00 | root      |               | data:TableFullScan_7
    ↪           |
|   └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

作为对比，c2 是可见的索引，优化器将可以使用索引：

```
EXPLAIN SELECT c2 FROM t1 ORDER BY c2;
```

```
+--
    ↪ ----------------------+----------+-----------+-----------------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object         | operator info
    ↪                 |
+--
    ↪ ----------------------+----------+-----------+-----------------------+------------------------------+
    ↪
| IndexReader_13          | 10000.00 | root      |                       | index:IndexFullScan_12
    ↪         |
| └─IndexFullScan_12      | 10000.00 | cop[tikv] | table:t1, index:c2(c2) | keep order:true,
    ↪ stats:pseudo |
+--
    ↪ ----------------------+----------+-----------+-----------------------+------------------------------+
    ↪
2 rows in set (0.00 sec)
```

即使用 SQL Hint USE INDEX 强制使用索引，优化器也无法使用不可见索引，否则 SQL 语句会报错：

```
SELECT * FROM t1 USE INDEX(c1);
```

```
ERROR 1176 (42000): Key 'c1' doesn't exist in table 't1'
```

> 注意：
>
> “不可见”是仅仅对优化器而言的，不可见索引仍然可以被修改或删除。

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.02 sec)
```

2.2.10.3 MySQL 兼容性

- TiDB 中的不可见索引是基于 MySQL 8.0 中的同等特性构建的。
- 与 MySQL 类似，TiDB 不允许将主键索引设为不可见。
- MySQL 中提供的优化器开关 `use_invisible_indexes=on` 可将所有的不可见索引重新设为可见。该功能在 TiDB 中不可用。

2.2.10.4 另请参阅

- CREATE TABLE
- CREATE INDEX
- ADD INDEX
- DROP INDEX
- RENAME INDEX

2.2.11 ALTER INSTANCE

`ALTER INSTANCE` 语句用于对单个 TiDB 实例进行变更操作。目前 TiDB 仅支持 `RELOAD TLS` 子句。

2.2.11.1 RELOAD TLS

`ALTER INSTANCE RELOAD TLS` 语句用于从原配置的证书 (`ssl-cert`)、密钥 (`ssl-key`) 和 CA (`ssl-ca`) 的路径重新加证书、密钥和 CA。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

在重加载遇到错误时默认会报错返回且继续使用变更前的密钥和证书，但在添加可选的 `NO ROLLBACK ON ERROR` 后遇到错误将不报错并以关闭 TLS 安全连接功能的方式处理后续请求。

2.2.11.2 语法图

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

InstanceOption ::=
    'RELOAD' 'TLS' ('NO' 'ROLLBACK' 'ON' 'ERROR')?
```

2.2.11.3 示例

```
ALTER INSTANCE RELOAD TLS;
```

2.2.11.4 MySQL 兼容性

仅支持从原配置路径重加载，不支持动态修改加载路径，也不支持动态启用启动 TiDB 时未开启的 TLS 加密连接功能。

2.2.11.5 另请参阅

为 TiDB 客户端服务端间通信开启加密传输

2.2.12 ALTER PLACEMENT POLICY

`ALTER PLACEMENT POLICY` 用于修改已创建的放置策略。此修改会自动更新至所有使用这些放置策略的表和分区。

`ALTER PLACEMENT POLICY` 会完全替换之前定义的规则，而不会和之前的规则合并，比如在下面的例子中，`FOLLOWERS=4` 就被 `ALTER PLACEMENT POLICY` 语句覆盖了：

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=4;
ALTER PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1";
```

2.2.12.1 语法图

```
AlterPolicyStmt ::=
    "ALTER" "PLACEMENT" "POLICY" IfExists PolicyName PlacementOptionList

PolicyName ::=
    Identifier

PlacementOptionList ::=
    PlacementOption
|   PlacementOptionList PlacementOption
|   PlacementOptionList ',' PlacementOption

PlacementOption ::=
    CommonPlacementOption
|   SugarPlacementOption
|   AdvancedPlacementOption

CommonPlacementOption ::=
    "FOLLOWERS" EqOpt LengthNum

SugarPlacementOption ::=
    "PRIMARY_REGION" EqOpt stringLit
|   "REGIONS" EqOpt stringLit
|   "SCHEDULE" EqOpt stringLit
```

```
AdvancedPlacementOption ::=
    "LEARNERS" EqOpt LengthNum
|   "CONSTRAINTS" EqOpt stringLit
|   "LEADER_CONSTRAINTS" EqOpt stringLit
|   "FOLLOWER_CONSTRAINTS" EqOpt stringLit
|   "LEARNER_CONSTRAINTS" EqOpt stringLit
```

2.2.12.2 示例

> **注意：**
>
> 如要查看所在集群中可用的区域，见SHOW PLACEMENT LABELS。如果未看到任何可用的区域，此 TiKV 集群在部署时可能未正确设置标签 (label)。

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1";
CREATE TABLE t1 (i INT) PLACEMENT POLICY=p1; -- 绑定放置策略 p1 到表 t1。
ALTER PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1,us-west-2"
    ↪ FOLLOWERS=4; -- t1 上的放置规则会自动更新。
SHOW CREATE PLACEMENT POLICY p1\G;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

***************************[ 1. row ]***************************
Policy        | p1
Create Policy | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-
    ↪ west-1,us-west-2" FOLLOWERS=4
1 row in set (0.00 sec)
```

2.2.12.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.12.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- DROP PLACEMENT POLICY

2.2.13 ALTER TABLE

ALTER TABLE 语句用于对已有表进行修改，以符合新表结构。ALTER TABLE 语句可用于：

- ADD，DROP，或RENAME 索引
- ADD，DROP，MODIFY 或CHANGE 列
- COMPACT 表数据

2.2.13.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName (
        AlterTableSpecListOpt AlterTablePartitionOpt |
        'ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )? AnalyzeOptionListOpt |
        'COMPACT' 'TIFLASH' 'REPLICA'
    )

TableName ::=
    Identifier ('.' Identifier)?

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' |  '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
```

```
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'SHARD_ROW_ID_BITS' ) EqOpt
    ↪ LengthNum
|   ( 'CACHE' | 'NOCACHE' )
```

2.2.13.2 示例

创建一张表，并插入初始数据：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

执行以下查询需要扫描全表，因为 `c1` 列未被索引：

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪               |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

你可以使用`ALTER TABLE .. ADD INDEX` 语句在 `t1` 表上添加索引。添加后，`EXPLAIN` 的分析结果显示 `SELECT * FROM t1 WHERE c1 = 3;` 查询已使用效率更高的索引范围扫描：

```
ALTER TABLE t1 ADD INDEX (c1);
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
Query OK, 0 rows affected (0.30 sec)
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                                |
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| IndexReader_6          | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                        |
| └─IndexRangeScan_5     | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
2 rows in set (0.00 sec)
```

TiDB 允许用户为 DDL 操作指定使用某一种 ALTER 算法。这仅为一种指定，并不改变实际的用于更改表的算法。如果你只想在群集的高峰时段允许即时 DDL 更改，则 ALTER 算法会很有用。示例如下：

```
ALTER TABLE t1 DROP INDEX c1, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.24 sec)
```

如果某一 DDL 操作要求使用 INPLACE 算法，而用户指定 ALGORITHM=INSTANT，会导致报错：

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=INSTANT;
```

```
ERROR 1846 (0A000): ALGORITHM=INSTANT is not supported. Reason: Cannot alter table by INSTANT.
    ↪ Try ALGORITHM=INPLACE.
```

但如果为 INPLACE 操作指定 ALGORITHM=COPY，会产生警告而非错误，这是因为 TiDB 将该指定解读为该算法或更好的算法。由于 TiDB 使用的算法可能不同于 MySQL，所以这一行为可用于 MySQL 兼容性。

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=COPY;
SHOW WARNINGS;
```

```
Query OK, 0 rows affected, 1 warning (0.25 sec)
+--
    ↪ -----+------+-----------------------------------------------------------------------------------
    ↪
```

```
| Level | Code | Message
    ↪                                                                                     |
+--
    ↪ -----+------+------------------------------------------------------------------------------------
    ↪
| Error | 1846 | ALGORITHM=COPY is not supported. Reason: Cannot alter table by COPY. Try
    ↪ ALGORITHM=INPLACE. |
+--
    ↪ -----+------+------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

2.2.13.3 MySQL 兼容性

TiDB 中的 ALTER TABLE 语法主要存在以下限制：

- 不支持在单个 ALTER TABLE 语句中进行多个更改。
- 不支持主键列上Reorg-Data 类型的变更。
- 不支持分区表上的列类型变更。
- 不支持生成列上的列类型变更。
- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为 TiDB 中的 CAST 函数与 MySQL 的行为存在兼容性问题。
- 不支持空间数据类型。
- ALTER TABLE t CACHE | NOCACHE 不是 MySQL 标准语法，而是 TiDB 扩展功能，参见缓存表。

其它限制可参考：TiDB 中 DDL 语句与 MySQL 的兼容性情况。

2.2.13.4 另请参阅

- 与 MySQL 兼容性对比
- ALTER TABLE ⋯ COMPACT
- ADD COLUMN
- DROP COLUMN
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

2.2.13.5 ALTER TABLE ⋯ COMPACT

> **警告：**
>
> 该语句目前是实验性功能，不建议在生产环境中使用。

TiDB 存储节点在后台会自动发起数据整理（Compaction）。数据整理时，表中的物理数据会被重写，如清理已删除的数据、合并多版本数据等，从而可以获得更高的访问性能，并减少磁盘空间占用。使用 `ALTER TABLE ... COMPACT` 语句可以立即对指定的表进行数据整理，而无需等待后台触发。

该语句执行时不会阻塞现有 SQL 语句的执行或 TiDB 功能的使用，包括事务、DDL、GC 等，也不会改变通过 SQL 语句访问获得的数据内容。但执行过程中会占用一定量的 IO 及 CPU 资源，可能对业务延迟造成影响。

该语句会等待表中所有副本都数据整理完毕后才结束运行并返回。在执行过程中，你可以通过`KILL` 语句安全地中断本张表的数据整理过程。中断不会破坏数据一致性或丢失数据，也不会影响后续重新发起或自动触发后台数据整理。

目前该语句仅支持对 TiFlash 进行数据整理，不支持对 TiKV 进行数据整理。

2.2.13.5.1 语法图

```
AlterTableCompactStmt ::=
    'ALTER' 'TABLE' TableName 'COMPACT' 'TIFLASH' 'REPLICA'
```

2.2.13.5.2 示例

对表中 TiFlash 副本进行数据整理

假设目前有一张 `employees` 表具有 4 个分区，且具有 2 个 TiFlash 副本：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);

ALTER TABLE employees SET TIFLASH REPLICA 2;
```

执行以下语句可对 `employees` 表上所有分区的 2 个 TiFlash 副本立即进行数据整理：

```
ALTER TABLE employees COMPACT TIFLASH REPLICA;
```

2.2.13.5.3 并发度

ALTER TABLE ... COMPACT 语句会同时对表中所有副本发起数据整理操作。

为了避免数据整理操作对在线业务造成显著影响，在同一时间，每个 TiFlash 实例默认只会处理一张表的数据整理操作（后台自动触发的数据整理除外）。这意味着，若你同时对多张表执行 ALTER TABLE ... COMPACT 语句，则它们在同一个 TiFlash 实例上会排队依次执行，而非同时执行。

你可以修改 TiFlash 配置文件参数manual_compact_pool_size，以更大资源占用为代价、获得更大的表级别并发度。例如，参数指定为 2 时，可以同时对 2 张表进行数据整理。

2.2.13.5.4 MySQL 兼容性

ALTER TABLE ... COMPACT 语法是 TiDB 引入的对标准 SQL 语法的扩展。尽管没有对应的 MySQL 语法，但你仍然可通过 MySQL 各版本客户端，或各个遵循 MySQL 协议的数据库驱动执行该语句。

2.2.13.5.5 TiDB Binlog 及 TiCDC 兼容性

ALTER TABLE ... COMPACT 语句不会导致逻辑数据变化，因而不会被 TiDB Binlog 及 TiCDC 同步到下游。

2.2.13.5.6 另请参阅

- ALTER TABLE
- KILL

2.2.14 ALTER USER

ALTER USER 语句用于更改 TiDB 权限系统内的已有用户。和 MySQL 一样，在 TiDB 权限系统中，用户是用户名和用户名所连接主机的组合。因此，可创建一个用户 'newuser2'@'192.168.1.1'，使其只能通过 IP 地址 192.168.1.1 进行连接。相同的用户名从不同主机登录时可能会拥有不同的权限。

2.2.14.1 语法图

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

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

UserSpec ::=
    Username AuthOption

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces

AuthOption ::=
```

```
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        ↪ AuthString | 'AS' HashString )? ) )?
```

2.2.14.2 示例

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

```
Query OK, 1 row affected (0.01 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*5806
    ↪ E04BBEE79E1899964C6A04D68BCA69B1A879' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK
    ↪  |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
ALTER USER 'newuser' IDENTIFIED BY 'newnewpassword';
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*
    ↪ FB8A1EA1353E8775CA836233E367FBDFCB37BE73' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

2.2.14.3 MySQL 兼容性

- 在 MySQL 中，ALTER 语句用于更改属性，例如使密码失效。但 TiDB 尚不支持此功能。

2.2.14.4 另请参阅

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

2.2.15 ANALYZE

ANALYZE 语句用于更新 TiDB 在表和索引上留下的统计信息。执行大批量更新或导入记录后，或查询执行计划不是最佳时，建议运行 ANALYZE。

当 TiDB 逐渐发现这些统计数据与预估不一致时，也会自动更新其统计数据。

目前 TiDB 收集统计信息分为全量收集和增量收集两种方式，分别通过 ANALYZE TABLE 和 ANALYZE INCREMENTAL TABLE 语句来实现。关于这两种语句的详细使用方式，可参考统计信息简介。

2.2.15.1 语法图

```
AnalyzeTableStmt ::=
    'ANALYZE' ( 'TABLE' ( TableNameList ( 'ALL COLUMNS' | 'PREDICATE COLUMNS' ) | TableName ( '
        ↪ INDEX' IndexNameList? | AnalyzeColumnOption | 'PARTITION' PartitionNameList ( 'INDEX'
        ↪  IndexNameList? | AnalyzeColumnOption )? )? ) | 'INCREMENTAL' 'TABLE' TableName ( '
        ↪ PARTITION' PartitionNameList )? 'INDEX' IndexNameList? ) AnalyzeOptionListOpt

AnalyzeOptionListOpt ::=
( WITH AnalyzeOptionList )?

AnalyzeOptionList ::=
AnalyzeOption ( ',' AnlyzeOption )*

AnalyzeOption ::=
( NUM ( 'BUCKETS' | 'TOPN' | ( 'CMSKETCH' ( 'DEPTH' | 'WIDTH' ) ) | 'SAMPLES' ) ) | ( FLOATNUM '
    ↪ SAMPLERATE' )

AnalyzeColumnOption ::=
( 'ALL COLUMNS' | 'PREDICATE COLUMNS' | 'COLUMNS' ColumnNameList )

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
```

```
    Identifier ( '.' Identifier )?

ColumnNameList ::=
    Identifier ( ',' Identifier )*

IndexNameList ::=
    Identifier ( ',' Identifier )*

PartitionNameList ::=
    Identifier ( ',' Identifier )*
```

2.2.15.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                               |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6         | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5    | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------+
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                 |
+-----------------------+---------+-----------+-----------------------+-----------------------------+
    ↪
| IndexReader_6         | 1.00    | root      |                       | index:IndexRangeScan_5
    ↪        |
| └─IndexRangeScan_5    | 1.00    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false |
+-----------------------+---------+-----------+-----------------------+-----------------------------+
    ↪
2 rows in set (0.00 sec)
```

2.2.15.3 MySQL 兼容性

- ANALYZE TABLE 在语法上与 MySQL 类似。但 ANALYZE TABLE 在 TiDB 上的执行时间可能长得多，因为它的内部运行方式不同。
- 在 MySQL 上不支持 ANALYZE INCREMENTAL TABLE 语句，它的使用可参考增量收集文档。

TiDB 与 MySQL 在以下方面存在区别：所收集的统计信息，以及查询执行过程中统计信息是如何被使用的。虽然 TiDB 中的 ANALYZE 语句在语法上与 MySQL 类似，但存在以下差异：

- 执行 ANALYZE TABLE 时，TiDB 可能不包含最近提交的更改。若对行进行了批量更改，在执行 ANALYZE TABLE 之前，你可能需要先执行 sleep(1)，这样统计信息更新才能反映这些更改。参见 #16570。
- ANALYZE TABLE 在 TiDB 中的执行时间比在 MySQL 中的执行时间要长得多。但你可以通过执行 SET GLOBAL ↪ tidb_enable_fast_analyze=1 来启用快速分析，这样能部分抵消这种执行上的性能差异。快速分析利用了采样，会导致统计信息的准确性降低。因此快速分析仍是一项实验特性。

2.2.15.4 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE

2.2.16 BATCH

BATCH 语句将一个 DML 语句拆成多个语句在内部执行，因此不保证事务的原子性和隔离性，是一个“非事务”语句。

目前 BATCH 语句仅支持 DELETE。

BATCH 语句在某一列将 DML 语句涉及的范围划分为多个区间，在每个区间执行一条 SQL。

详细的说明和使用限制见非事务语句。

2.2.16.1 语法图

```
NonTransactionalDeleteStmt ::=
    'BATCH' ( 'ON' ColumnName )? 'LIMIT' NUM DryRunOptions? DeleteFromStmt

DryRunOptions ::=
    'DRY' 'RUN' 'QUERY'?
```

2.2.16.2 MySQL 兼容性

BATCH 语句是 TiDB 独有的语句，与 MySQL 不兼容。

2.2.16.3 另请参阅

- 非事务语句

2.2.17 BEGIN

BEGIN 语句用于在 TiDB 内启动一个新事务，类似于 START TRANSACTION 和 SET autocommit=0 语句。

在没有 BEGIN 语句的情况下，每个语句默认在各自的事务中自动提交，从而确保 MySQL 兼容性。

2.2.17.1 语法图

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( 'WITH' 'TIMESTAMP' 'BOUND' TimestampBound
    ↪  )? ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' )?
```

2.2.17.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
BEGIN;
```

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

```
INSERT INTO t1 VALUES (1);
```

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

```
COMMIT;
```

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

2.2.17.3 MySQL 兼容性

TiDB 支持 `BEGIN PESSIMISTIC` 或 `BEGIN OPTIMISTIC` 的语法扩展，用户可以为某一个事务覆盖默认的事务模型。

2.2.17.4 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION
- TiDB 乐观事务模型
- TiDB 悲观事务模式

2.2.18 CHANGE COLUMN

`ALTER TABLE.. CHANGE COLUMN` 语句用于在已有表上更改列，包括对列进行重命名，和将数据改为兼容类型。

从 v5.1.0 版本起，TiDB 开始支持 Reorg 数据的类型变更，包括但不限于：

- 从 varchar 转换为 bigint
- decimal 精度修改
- 从 varchar(10) 到 varchar(5) 的长度压缩

2.2.18.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName ChangeColumnSpec ( ',' ChangeColumnSpec )*

ChangeColumnSpec
         ::= 'CHANGE' ColumnKeywordOpt 'IF EXISTS' ColumnName ColumnName ColumnType ColumnOption*
             ↪  ( 'FIRST' | 'AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'
```

```
ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               ↪ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               ↪ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
           | 'STORAGE' StorageMedia
           | 'AUTO_RANDOM' ( '(' LengthNum ')' )?

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

2.2.18.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

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

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

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

```
ALTER TABLE t1 CHANGE col1 col2 INT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
ALTER TABLE t1 CHANGE col2 col3 BIGINT, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
ALTER TABLE t1 CHANGE col3 col4 BIGINT, CHANGE id id2 INT NOT NULL;
```

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

```
CREATE TABLE t (a int primary key);
ALTER TABLE t CHANGE COLUMN a a VARCHAR(10);
```

```
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (c1 INT, c2 INT, c3 INT) partition by range columns(c1) ( partition p0 values less
    ↪  than (10), partition p1 values less than (maxvalue));
ALTER TABLE t CHANGE COLUMN c1 c1 DATETIME;
```

```
ERROR 8200 (HY000): Unsupported modify column: table is partition table
```

```
CREATE TABLE t (a INT, b INT as (a+1));
ALTER TABLE t CHANGE COLUMN b b VARCHAR(10);
```

```
ERROR 8200 (HY000): Unsupported modify column: column is generated
```

```
CREATE TABLE t (a DECIMAL(13, 7));
ALTER TABLE t CHANGE COLUMN a a DATETIME;
```

```
ERROR 8200 (HY000): Unsupported modify column: change from original type decimal(13,7) to
    ↪ datetime is currently unsupported yet
```

2.2.18.3 MySQL 兼容性

- 不支持在单个 ALTER TABLE 语句中进行多个更改。
- 不支持主键列上Reorg-Data 类型的变更。
- 不支持分区表上的列类型变更。
- 不支持生成列上的列类型变更。
- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为 TiDB 中 CAST 函数与 MySQL 的行为存在兼容性问题。

2.2.18.4 另请参阅

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

2.2.19 CHANGE DRAINER

CHANGE DRAINER 语句用于修改集群中 Drainer 的状态信息。

> **注意：**
>
> Drainer 在正常运行时会自动上报状态到 PD，仅在 Drainer 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Drainer 状态信息。

2.2.19.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-04-30 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

可以看出 drainer1 已经超过一天没有更新状态，该 Drainer 处于异常状态，但是 State 仍然为 Online，使用 CHANGE DRAINER 将该 Drainer 状态修改为 paused：

```
CHANGE DRAINER TO NODE_STATE ='paused' FOR NODE_ID 'drainer1';
```

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

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Paused | 408553768673342532 | 2019-04-30 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

2.2.19.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.19.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS

2.2.20 CHANGE PUMP

`CHANGE PUMP` 语句用于修改集群中 Pump 的状态信息。

> **注意：**
>
> Pump 在正常运行时会自动上报状态到 PD，仅在 Pump 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Pump 状态信息。

2.2.20.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

可以看出 pump1 已经超过一天没有更新状态，该 Pump 处于异常状态，但是 State 仍然为 `Online`，使用 `CHANGE PUMP` 将该 Pump 状态修改为 `paused`：

```
CHANGE PUMP TO NODE_STATE ='paused' FOR NODE_ID 'pump1';
```

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

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Paused | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
```

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

2.2.20.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.20.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE DRAINER STATUS

2.2.21 COMMIT

COMMIT 语句用于在 TiDB 服务器内部提交事务。

在不使用 BEGIN 或 START TRANSACTION 语句的情况下，TiDB 中每一个查询语句本身也会默认作为事务处理，自动提交，确保了与 MySQL 的兼容。

2.2.21.1 语法图

```
CommitStmt ::=
    'COMMIT' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

2.2.21.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
START TRANSACTION;
```

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

```
INSERT INTO t1 VALUES (1);
```

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

```
COMMIT;
```

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

2.2.21.3 MySQL 兼容性

- 目前，TiDB 不使用元数据锁 (MDL) 来防止 DDL 语句修改事务使用的表。如果表定义被更改，提交事务将导致报错 `Information schema is changed`。此时，事务会自动回滚。
- TiDB 3.0.8 及更新版本默认使用悲观事务模式。在乐观事务模型下，需要考虑到修改的行已被另一个事务修改，导致 `COMMIT` 语句可能执行失败的情况。
- 启用乐观事务模型后，`UNIQUE` 和 `PRIMARY KEY` 约束检查将延迟直至语句提交。当 `COMMIT` 语句失败时，这可能导致其他问题。可通过设置 `tidb_constraint_check_in_place=TRUE` 来改变该行为。
- TiDB 解析但忽略 `ROLLBACK AND [NO] RELEASE` 语法。在 MySQL 中，使用该语法可在提交事务后立即断开客户端会话。在 TiDB 中，建议使用客户端程序的 `mysql_close()` 来实现该功能。
- TiDB 解析但忽略 `ROLLBACK AND [NO] CHAIN` 语法。在 MySQL 中，使用该语法可在提交当前事务时立即以相同的隔离级别开启新事务。在 TiDB 中，推荐直接开启新事务。

2.2.21.4 另请参阅

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

2.2.22 CREATE [GLOBAL|SESSION] BINDING

`CREATE [GLOBAL|SESSION] BINDING` 语句用于在 TiDB 中创建新的执行计划绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

`BINDING` 语句可以在 `GLOBAL` 或者 `SESSION` 作用域内创建执行计划绑定。在不指定作用域时，默认的作用域为 `SESSION`。

被绑定的 SQL 语句会被参数化后存储到系统表中。在处理 SQL 查询时，只要参数化后的 SQL 语句和系统表中某个被绑定的 SQL 语句一致，并且系统变量 `tidb_use_plan_baselines` 的值为 `ON`（其默认值为 `ON`），即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。

2.2.22.1 语法图

```
CreateBindingStmt ::=
    'CREATE' GlobalScope 'BINDING' 'FOR' BindableStmt 'USING' BindableStmt

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

2.2.22.2 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
```

```
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object        |
    ↪ execution info                                                                  | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                      | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                                       | range:[123,123],
    ↪ keep order:false | N/A              | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                                     | keep order:false
    ↪                     | N/A              | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+-----------+---------+-----------+--------------+--------------------------------
    ↪
| id                      | estRows   | actRows | task      | access object | execution info
    ↪                                                                        | operator info      |
```

```
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+--------------+-------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |              | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |              | time:224ms, loops
    ↪ :298                                                        | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1     | time:220ms, loops
    ↪ :298                                                        | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+--------------+-------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object       |
    ↪ execution info                                                          | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
```

```
    ↪                                    | 109.1484375 KB | N/A   |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                          | range:[123,123], keep
    ↪ order:false | N/A             | N/A   |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    ↪ , loops:4                                                          | keep order:false
    ↪                    | N/A             | N/A   |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
3 rows in set (0.01 sec)
```

2.2.22.3 MySQL 兼容性

`CREATE [GLOBAL|SESSION] BINDING` 语句是 TiDB 对 MySQL 语法的扩展。

2.2.22.4 另请参阅

- DROP [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

2.2.23 CREATE DATABASE

`CREATE DATABASE` 语句用于在 TiDB 上创建新数据库。按照 SQL 标准，“数据库”一词在 MySQL 术语中最接近“schema”。

2.2.23.1 语法图

```
CreateDatabaseStmt ::=
    'CREATE' 'DATABASE' IfNotExists DBName DatabaseOptionListOpt

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

DBName ::=
    Identifier

DatabaseOptionListOpt ::=
    DatabaseOptionList?
```

2.2.23.2 语法说明

CREATE DATABASE 用于创建数据库，并可以指定数据库的默认属性（如数据库默认字符集、排序规则）。CREATE SCHEMA 跟 CREATE DATABASE 操作效果一样。

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification] ...

create_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
```

当创建已存在的数据库且不指定使用 IF NOT EXISTS 时会报错。

create_specification 选项用于指定数据库具体的 CHARACTER SET 和 COLLATE。目前 TiDB 只支持部分的字符集和排序规则，请参照字符集支持。

2.2.23.3 示例

```
CREATE DATABASE mynewdatabase;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE mynewdatabase;
```

```
Database changed
```

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

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

```
SHOW TABLES;
```

```
+-------------------------+
| Tables_in_mynewdatabase |
+-------------------------+
| t1                      |
+-------------------------+
1 row in set (0.00 sec)
```

2.2.23.4 MySQL 兼容性

CREATE DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.23.5 另请参阅

- USE
- ALTER DATABASE
- DROP DATABASE
- SHOW DATABASES

2.2.24 CREATE PLACEMENT POLICY

`CREATE PLACEMENT POLICY` 用于创建命名的放置策略，随后可以将该策略分配给表、分区或数据库。

2.2.24.1 语法图

```
CreatePolicyStmt ::=
    "CREATE" "PLACEMENT" "POLICY" IfNotExists PolicyName PlacementOptionList

PolicyName ::=
    Identifier

PlacementOptionList ::=
    PlacementOption
|   PlacementOptionList PlacementOption
|   PlacementOptionList ',' PlacementOption

PlacementOption ::=
    CommonPlacementOption
|   SugarPlacementOption
|   AdvancedPlacementOption

CommonPlacementOption ::=
    "FOLLOWERS" EqOpt LengthNum

SugarPlacementOption ::=
    "PRIMARY_REGION" EqOpt stringLit
|   "REGIONS" EqOpt stringLit
|   "SCHEDULE" EqOpt stringLit

AdvancedPlacementOption ::=
    "LEARNERS" EqOpt LengthNum
|   "CONSTRAINTS" EqOpt stringLit
|   "LEADER_CONSTRAINTS" EqOpt stringLit
|   "FOLLOWER_CONSTRAINTS" EqOpt stringLit
|   "LEARNER_CONSTRAINTS" EqOpt stringLit
```

2.2.24.2 示例

> **注意：**
>
> 如要查看所在集群中可用的区域，见SHOW PLACEMENT LABELS。如果未看到任何可用的区域，此 TiKV 集群在部署时可能未正确设置标签 (label)。

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW CREATE PLACEMENT POLICY p1;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

+--------+-------------------------------------------------------------------------------------------------------+
    ↪
| Policy | Create Policy
    ↪                                                                                                      |
+--------+-------------------------------------------------------------------------------------------------------+
    ↪
| p1     | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1"
    ↪ FOLLOWERS=4 |
+--------+-------------------------------------------------------------------------------------------------------+
    ↪
1 row in set (0.00 sec)
```

2.2.24.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.24.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- ALTER PLACEMENT POLICY
- DROP PLACEMENT POLICY

2.2.25 CREATE ROLE

CREATE ROLE 语句是基于角色的访问控制 (RBAC) 操作的一部分，用于创建新角色并将新角色分配给用户。

2.2.25.1 语法图

```
CreateRoleStmt ::=
    'CREATE' 'ROLE' IfNotExists RoleSpec (',' RoleSpec)*

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

RoleSpec ::=
    Rolename
```

2.2.25.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
```

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

SHOW GRANTS;
+--------------------------------------------+
| Grants for User                            |
+--------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'       |
| GRANT Select ON test.* TO 'jennifer'@'%'   |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+--------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+--------------------------------------------+
| Grants for User                            |
+--------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'       |
| GRANT Select ON test.* TO 'jennifer'@'%'   |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+--------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
```

```
1 row in set (0.00 sec)
```

2.2.25.3 MySQL 兼容性

CREATE ROLE 语句与 MySQL 8.0 的“角色”功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.25.4 另请参阅

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

2.2.26 CREATE SEQUENCE

CREATE SEQUENCE 语句用于在 TiDB 中创建序列对象。序列是一种与表、视图对象平级的数据库对象，用于生成自定义的序列化 ID。

2.2.26.1 语法图

```
CreateSequenceStmt ::=
    'CREATE' 'SEQUENCE' IfNotExists TableName CreateSequenceOptionListOpt
        ↪ CreateTableOptionListOpt

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

CreateSequenceOptionListOpt ::=
    SequenceOption*

SequenceOptionList ::=
    SequenceOption

SequenceOption ::=
    ( 'INCREMENT' ( '='? | 'BY' ) | 'START' ( '='? | 'WITH' ) | ( 'MINVALUE' | 'MAXVALUE' | '
        ↪ CACHE' ) '='? ) SignedNum
|   'NOMINVALUE'
|   'NO' ( 'MINVALUE' | 'MAXVALUE' | 'CACHE' | 'CYCLE' )
|   'NOMAXVALUE'
```

```
|   'NOCACHE'
|   'CYCLE'
|   'NOCYCLE'
```

2.2.26.2 语法说明

```
CREATE [TEMPORARY] SEQUENCE [IF NOT EXISTS] sequence_name
    [ INCREMENT [ BY | = ] increment ]
    [ MINVALUE [=] minvalue | NO MINVALUE | NOMINVALUE ]
    [ MAXVALUE [=] maxvalue | NO MAXVALUE | NOMAXVALUE ]
    [ START [ WITH | = ] start ]
    [ CACHE [=] cache | NOCACHE | NO CACHE]
    [ CYCLE | NOCYCLE | NO CYCLE]
    [table_options]
```

2.2.26.3 参数说明

参数	默认值	描述
TEMPORARY	false	TiDB 暂时不支持 TEMPORARY 选项，仅在语法上做兼容。
INCREMENT	1	指定序列的步长。其正负值可以控制序列的增长方向。

参数	默认值	描述
MINVALUE	或 -9223372036854775807	指定序列的最小值。当 INCREMENT > 0 时，默认值为 1；当 INCREMENT < 0 时，默认值为 -9223372036854775807。
MAXVALUE	9223372036854775806 或 -1	指定序列的最大值。当 INCREMENT > 0 时，默认值为 9223372036854775806；当 INCREMENT < 0 时，默认值为 -1。

参数	默认值	描述
START ↪	MINVALUE ↪ 或 MAXVALUE ↪	指定序列的初始值。当 INCREMENT ↪ > 0 时，默认值为 MINVALUE ↪ ; 当 INCREMENT ↪ < 0 时，默认值为 MAXVALUE ↪ 。
CACHE ↪	1000	指定每个 TiDB 本地缓存序列的大小。

参数	默认值	描述
CYCLE ↪	NO ↪ CYCLE ↪	指定序列用完之后是否要循环使用。在 CYCLE ↪ 的情况下，当 INCREMENT ↪ > 0 时，序列用完后的后续起始值为 MINVALUE ↪ ；当 INCREMENT ↪ < 0 时，序列用完后的后续起始值为 MAXVALUE ↪ 。

2.2.26.4 SEQUENCE 函数

主要通过表达式函数来操纵序列的使用。

- NEXTVAL 或 NEXT VALUE FOR

本质上都是 nextval() 函数，获取序列对象的下一个有效值，其参数为序列的 identifier。

- LASTVAL

  lastval() 函数，用于获取本会话上一个使用过的值。如果没有值，则为 NULL，其参数为序列的 identifier。

- SETVAL

  setval() 函数，用于设置序列的增长。其第一参数为序列的 identifier，第二个参数为 num。

> **注意：**
>
> 在 TiDB 序列的实现中，SETVAL 函数并不能改变序列增长的初始步调或循环步调。在 SETVAL 之后只会返回符合步调规律的下一个有效的序列值。

2.2.26.5 示例

- 创建一个默认参数的序列对象。

```
CREATE SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.06 sec)
```

- 使用 nextval() 函数获取序列对象的下一个值。

```
SELECT nextval(seq);
```

```
+--------------+
| nextval(seq) |
+--------------+
|            1 |
+--------------+
1 row in set (0.02 sec)
```

- 使用 lastval() 函数获取本会话上一次调用序列对象所产生的值。

```
SELECT lastval(seq);
```

```
+--------------+
| lastval(seq) |
+--------------+
|            1 |
+--------------+
1 row in set (0.02 sec)
```

- 使用 `setval()` 函数设置序列对象当前值的位置。

```
SELECT setval(seq, 10);
```

```
+-----------------+
| setval(seq, 10) |
+-----------------+
|              10 |
+-----------------+
1 row in set (0.01 sec)
```

- 也可使用 `next value for` 语法获取序列的下一个值。

```
SELECT next value for seq;
```

```
+--------------------+
| next value for seq |
+--------------------+
|                 11 |
+--------------------+
1 row in set (0.00 sec)
```

- 创建一个默认自定义参数的序列对象。

```
CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;
```

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

- 当本会话还未使用过序列对象时，`lastval()` 函数返回 `NULL` 值。

```
SELECT lastval(seq2);
```

```
+---------------+
| lastval(seq2) |
+---------------+
|          NULL |
+---------------+
1 row in set (0.01 sec)
```

- 序列对象 `nextval()` 的第一个有效值为 `start` 值。

```
SELECT nextval(seq2);
```

```
+---------------+
| nextval(seq2) |
+---------------+
|             3 |
```

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

- 使用 `setval()` 虽然可以改变序列对象当前值的位置，但是无法改变下一个值的等差规律。

```
SELECT setval(seq2, 6);
```

```
+-----------------+
| setval(seq2, 6) |
+-----------------+
|               6 |
+-----------------+
1 row in set (0.00 sec)
```

- 使用 `nextval()` 下一个值获取时，会遵循序列定义的等差规律。

```
SELECT next value for seq2;
```

```
+---------------------+
| next value for seq2 |
+---------------------+
|                   7 |
+---------------------+
1 row in set (0.00 sec)
```

- 可以将序列的下一个值作为列的默认值来使用。

```
CREATE TABLE t(a int default next value for seq2);
```

```
Query OK, 0 rows affected (0.02 sec)
```

- 下列示例中，因为没有指定值，会直接获取 `seq2` 的默认值来使用。

```
INSERT into t values();
```

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

```
SELECT * from t;
```

```
+------+
| a    |
+------+
|    9 |
+------+
1 row in set (0.00 sec)
```

- 下列示例中，因为没有指定值，会直接获取 seq2 的默认值来使用。由于 seq2 的下一个值超过了上述示例 (CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;) 的定义范围，所以会显示报错。

```
INSERT into t values();
```

```
ERROR 4135 (HY000): Sequence 'test.seq2' has run out
```

2.2.26.6 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

除了 SETVAL 函数外，其他函数的 “步调 (progressions)” 与 MariaDB 一致。这里的步调是指，序列中的数在定义之后会产生一定的等差关系。SETVAL 虽然可以将序列的当前值进行移动设置，但是后续出现的值仍会遵循原有的等差关系。

示例如下：

```
1, 3, 5, ...            // 序列遵循起始为 1、步长为 2 的等差关系。
select setval(seq, 6)   // 设置序列的当前值为 6。
7, 9, 11, ...           // 后续产生值仍会遵循这个等差关系。
```

在 CYCLE 模式下，序列的起始值第一轮为 start，后续轮次将会是 MinValue (increment > 0) 或 MaxValue (increment < 0)。

2.2.26.7 另请参阅

- DROP SEQUENCE
- SHOW CREATE SEQUENCE

2.2.27 CREATE TABLE LIKE

CREATE TABLE LIKE 语句用于复制已有表的定义，但不复制任何数据。

2.2.27.1 语法图

```
CreateTableLikeStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName LikeTableWithOrWithoutParen OnCommitOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

LikeTableWithOrWithoutParen ::=
    'LIKE' TableName
|   '(' 'LIKE' TableName ')'
```

```
OnCommitOpt ::=
    ('ON' 'COMMIT' 'DELETE' 'ROWS')?
```

2.2.27.2 示例

```
CREATE TABLE t1 (a INT NOT NULL);
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

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

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

```
CREATE TABLE t2 LIKE t1;
```

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

```
SELECT * FROM t2;
```

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

2.2.27.3 Region 的预切分

如果被复制的表定义了 PRE_SPLIT_REGIONS 属性，则通过 CREATE TABLE LIKE 语句复制的表，会继承该属性并在建表时预切分 Region。关于 PRE_SPLIT_REGIONS 属性的说明，参见CREATE TABLE 语句。

2.2.27.4 MySQL 兼容性

CREATE TABLE LIKE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.27.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE

2.2.28 CREATE TABLE

`CREATE TABLE` 语句用于在当前所选数据库中创建新表，与 MySQL 中 `CREATE TABLE` 语句的行为类似。另可参阅单独的 `CREATE TABLE LIKE` 文档。

2.2.28.1 语法图

```
CreateTableStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName ( TableElementListOpt
        ↪ CreateTableOptionListOpt PartitionOpt DuplicateOpt AsOpt CreateTableSelectOpt |
        ↪ LikeTableWithOrWithoutParen ) OnCommitOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

TableElementListOpt ::=
    ( '(' TableElementList ')' )?

TableElementList ::=
    TableElement ( ',' TableElement )*

TableElement ::=
    ColumnDef
|   Constraint

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

ColumnOptionListOpt ::=
    ColumnOption*

ColumnOptionList ::=
    ColumnOption*
```

```
ColumnOption ::=
    'NOT'? 'NULL'
|   'AUTO_INCREMENT'
|   PrimaryOpt 'KEY'
|   'UNIQUE' 'KEY'?
|   'DEFAULT' DefaultValueExpr
|   'SERIAL' 'DEFAULT' 'VALUE'
|   'ON' 'UPDATE' NowSymOptionFraction
|   'COMMENT' stringLit
|   ConstraintKeywordOpt 'CHECK' '(' Expression ')' EnforcedOrNotOrNotNullOpt
|   GeneratedAlways 'AS' '(' Expression ')' VirtualOrStored
|   ReferDef
|   'COLLATE' CollationName
|   'COLUMN_FORMAT' ColumnFormat
|   'STORAGE' StorageMedia
|   'AUTO_RANDOM' OptFieldLen

CreateTableOptionListOpt ::=
    TableOptionList?

PartitionOpt ::=
    ( 'PARTITION' 'BY' PartitionMethod PartitionNumOpt SubPartitionOpt PartitionDefinitionListOpt
        ↪  )?

DuplicateOpt ::=
    ( 'IGNORE' | 'REPLACE' )?

TableOptionList ::=
    TableOption ( ','? TableOption )*

TableOption ::=
    PartDefOption
|   DefaultKwdOpt ( CharsetKw EqOpt CharsetName | 'COLLATE' EqOpt CollationName )
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'AVG_ROW_LENGTH' | 'CHECKSUM' | '
    ↪ TABLE_CHECKSUM' | 'KEY_BLOCK_SIZE' | 'DELAY_KEY_WRITE' | 'SHARD_ROW_ID_BITS' | '
    ↪ PRE_SPLIT_REGIONS' ) EqOpt LengthNum
|   ( 'CONNECTION' | 'PASSWORD' | 'COMPRESSION' ) EqOpt stringLit
|   RowFormat
|   ( 'STATS_PERSISTENT' | 'PACK_KEYS' ) EqOpt StatsPersistentVal
|   ( 'STATS_AUTO_RECALC' | 'STATS_SAMPLE_PAGES' ) EqOpt ( LengthNum | 'DEFAULT' )
|   'STORAGE' ( 'MEMORY' | 'DISK' )
|   'SECONDARY_ENGINE' EqOpt ( 'NULL' | StringName )
|   'UNION' EqOpt '(' TableNameListOpt ')'
|   'ENCRYPTION' EqOpt EncryptionOpt
```

```
OnCommitOpt ::=
    ('ON' 'COMMIT' 'DELETE' 'ROWS')?
```

TiDB 支持以下 table_option。TiDB 会解析并忽略其他 table_option 参数，例如 AVG_ROW_LENGTH、CHECKSUM、COMPRESSION、CONNECTION、DELAY_KEY_WRITE、ENGINE、KEY_BLOCK_SIZE、MAX_ROWS、MIN_ROWS、ROW_FORMAT 和 STATS_PERSISTENT。

参数	含义	举例
AUTO_INCREMENT	自增字段初始值	AUTO_INCREMENT = 5
SHARD_ROW_ID_BITS	用来设置隐式 _tidb_rowid 的分片数量的 bit 位数	SHARD_ROW_ID_BITS = 4
PRE_SPLIT_REGIONS	用来在建表时预先均匀切分 2^(PRE_SPLIT_REGIONS) 个 Region	PRE_SPLIT_REGIONS = 4
AUTO_ID_CACHE	用来指定 Auto ID 在 TiDB 实例中 Cache 的大小，默认情况下 TiDB 会根据 Auto ID 分配速度自动调整	AUTO_ID_CACHE = 200
AUTO_RANDOM_BASE	用来指定 AutoRandom 自增部分的初始值，该参数可以被认为属于内部接口的一部分，对于用户而言请忽略	AUTO_RANDOM_BASE = 0
CHARACTER SET	指定该表所使用的字符集	CHARACTER SET = ‘utf8mb4’
COLLATE	指定该表所使用的字符集排序规则	COLLATE = ‘utf8mb4_bin’
COMMENT	注释信息	COMMENT = ‘comment info’

> **注意：**
>
> 在 TiDB 配置文件中，split-table 默认开启。当该配置项开启时，建表操作会为每个表建立单独的 Region，详情参见TiDB 配置文件描述。

2.2.28.2 示例

创建一张简单表并插入一行数据：

```
CREATE TABLE t1 (a int);
DESC t1;
SHOW CREATE TABLE t1\G
INSERT INTO t1 (a) VALUES (1);
SELECT * FROM t1;
```

```
mysql> drop table if exists t1;
Query OK, 0 rows affected (0.23 sec)

mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.09 sec)
```

```
mysql> DESC t1;
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
1 row in set (0.00 sec)

mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)

mysql> INSERT INTO t1 (a) VALUES (1);
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM t1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

删除一张表。如果该表不存在，就建一张表：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE IF NOT EXISTS t1 (
 id BIGINT NOT NULL PRIMARY KEY auto_increment,
 b VARCHAR(200) NOT NULL
);
DESC t1;
```

```
mysql> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected (0.22 sec)
mysql> CREATE TABLE IF NOT EXISTS t1 (
    ->  id BIGINT NOT NULL PRIMARY KEY auto_increment,
    ->  b VARCHAR(200) NOT NULL
    -> );
Query OK, 0 rows affected (0.08 sec)
mysql> DESC t1;
+-------+--------------+------+------+---------+----------------+
```

```
| Field | Type         | Null | Key  | Default | Extra          |
+-------+--------------+------+------+---------+----------------+
| id    | bigint(20)   | NO   | PRI  | NULL    | auto_increment |
| b     | varchar(200) | NO   |      | NULL    |                |
+-------+--------------+------+------+---------+----------------+
2 rows in set (0.00 sec)
```

2.2.28.3 MySQL 兼容性

- 支持除空间类型以外的所有数据类型。
- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。
- 为了与 MySQL 兼容，index_col_name 属性支持 length 选项，最大长度默认限制为 3072 字节。此长度限制可以通过配置项 max-index-length 更改，具体请参阅TiDB 配置文件描述。
- 为了与 MySQL 兼容，TiDB 会解析但忽略 index_col_name 属性的 [ASC | DESC] 索引排序选项。
- COMMENT 属性不支持 WITH PARSER 选项。
- TiDB 在单个表中最多支持 512 列。InnoDB 中相应的数量限制为 1017，MySQL 中的硬限制为 4096。详情参阅TiDB 使用限制。
- 当前仅支持 Range、 Hash 和 Range Columns（单列）类型的分区表，详情参阅分区表。
- TiDB 会解析并忽略 CHECK 约束，与 MySQL 5.7 相兼容。详情参阅CHECK 约束。
- TiDB 会解析并存储外键约束，但不会在 DML 语句中强制对外键进行约束检查。详情外键约束。

2.2.28.4 另请参阅

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

2.2.29 CREATE USER

CREATE USER 语句用于创建带有指定密码的新用户。和 MySQL 一样，在 TiDB 权限系统中，用户是用户名和用户名所连接主机的组合。因此，可创建一个用户 'newuser2'@'192.168.1.1'，使其只能通过 IP 地址 192.168.1.1 进行连接。相同的用户名从不同主机登录时可能会拥有不同的权限。

2.2.29.1 语法图

```
CreateUserStmt ::=
    'CREATE' 'USER' IfNotExists UserSpecList RequireClauseOpt ConnectionOptions
        ↪ PasswordOrLockOptions

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

UserSpecList ::=
```

```
    UserSpec ( ',' UserSpec )*

UserSpec ::=
    Username AuthOption

AuthOption ::=
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        ↪ AuthString | 'AS' HashString )? ) )?

StringName ::=
    stringLit
|   Identifier
```

2.2.29.2 示例

创建一个密码为 newuserpassword 的用户。

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

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

创建一个只能在 192.168.1.1 登录的用户，密码为 newuserpassword。

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

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

创建一个要求在登录时使用 TLS 连接的用户。

```
CREATE USER 'newuser3'@'%' REQUIRE SSL IDENTIFIED BY 'newuserpassword';
```

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

创建一个要求在登录时提供指定客户端证书的用户。

```
CREATE USER 'newuser4'@'%' REQUIRE ISSUER '/C=US/ST=California/L=San Francisco/O=PingCAP'
    ↪ IDENTIFIED BY 'newuserpassword';
```

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

2.2.29.3 MySQL 兼容性

- TiDB 不支持 WITH MAX_QUERIES_PER_HOUR、WITH MAX_UPDATES_PER_HOUR、WITH MAX_USER_CONNECTIONS 等 CREATE 选项。
- TiDB 不支持 DEFAULT ROLE 选项。
- TiDB 不支持 PASSWORD EXPIRE、PASSWORD HISTORY 等有关密码限制的 CREATE 选项。
- TiDB 不支持 ACCOUNT LOCK 和 ACCOUNT UNLOCK 选项。
- 对于 TiDB 尚不支持的 CREATE 选项。这些选项可被解析，但会被忽略。

2.2.29.4 另请参阅

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

2.2.30 CREATE VIEW

使用 CREATE VIEW 语句将 SELECT 语句保存为类似于表的可查询对象。TiDB 中的视图是非物化的，这意味着在查询视图时，TiDB 将在内部重写查询，以将视图定义与 SQL 查询结合起来。

2.2.30.1 语法图

```
CreateViewStmt ::=
    'CREATE' OrReplace ViewAlgorithm ViewDefiner ViewSQLSecurity 'VIEW' ViewName ViewFieldList '
        ↪ AS' CreateViewSelectOpt ViewCheckOption

OrReplace ::=
    ( 'OR' 'REPLACE' )?

ViewAlgorithm ::=
    ( 'ALGORITHM' '=' ( 'UNDEFINED' | 'MERGE' | 'TEMPTABLE' ) )?

ViewDefiner ::=
    ( 'DEFINER' '=' Username )?

ViewSQLSecurity ::=
    ( 'SQL' 'SECURITY' ( 'DEFINER' | 'INVOKER' ) )?

ViewName ::= TableName

ViewFieldList ::=
    ( '(' Identifier ( ',' Identifier )* ')' )?

ViewCheckOption ::=
    ( 'WITH' ( 'CASCADED' | 'LOCAL' ) 'CHECK' 'OPTION' )?
```

2.2.30.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE c1 > 2;
```

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

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

```
INSERT INTO t1 (c1) VALUES (6);
```

```
Query OK, 1 row affected (0.01 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
```

```
|  4 |  4 |
|  5 |  5 |
|  6 |  6 |
+----+----+
4 rows in set (0.00 sec)
```

```
INSERT INTO v1 (c1) VALUES (7);
```

```
ERROR 1105 (HY000): insert into view v1 is not supported now.
```

2.2.30.3 MySQL 兼容性

- 目前 TiDB 中的任何视图都不可被插入，也不可被更新（即不支持 INSERT VIEW，也不支持 UPDATE VIEW）。WITH CHECK OPTION 只做了语法兼容但不生效。
- 目前 TiDB 中的视图不支持 ALTER VIEW，但可以使用 CREATE OR REPLACE 替代。
- 目前 ALGORITHM 字段在 TiDB 中只做了语法兼容但不生效，TiDB 目前只支持 MERGE 算法。

2.2.30.4 另请参阅

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

2.2.31 DEALLOCATE

DEALLOCATE 语句用于为服务器端预处理语句提供 SQL 接口。

2.2.31.1 语法图

```
DeallocateStmt ::=
    DeallocateSym 'PREPARE' Identifier

DeallocateSym ::=
    'DEALLOCATE'
|   'DROP'

Identifier ::=
    identifier
|   UnReservedKeyword
|   NotKeywordToken
|   TiDBKeyword
```

2.2.31.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

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

```
SET @number = 5;
```

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

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

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

2.2.31.3 MySQL 兼容性

DEALLOCATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.31.4 另请参阅

- PREPARE
- EXECUTE

2.2.32 DELETE

DELETE 语句用于从指定的表中删除行。

2.2.32.1 语法图

```
DeleteFromStmt ::=
    'DELETE' TableOptimizerHints PriorityOpt QuickOptional IgnoreOptional ( 'FROM' ( TableName
        ↪ TableAsNameOpt IndexHintListOpt WhereClauseOptional OrderByOptional LimitClause |
        ↪ TableAliasRefList 'USING' TableRefs WhereClauseOptional ) | TableAliasRefList 'FROM'
        ↪ TableRefs WhereClauseOptional )
```

2.2.32.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
DELETE FROM t1 WHERE id = 4;
```

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

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  5 |  5 |
+----+----+
4 rows in set (0.00 sec)
```

2.2.32.3 MySQL 兼容性

DELETE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.32.4 另请参阅

- INSERT
- SELECT
- UPDATE
- REPLACE

2.2.33 DESC

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

2.2.34 DESCRIBE

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

2.2.35 DO

DO 语句用于执行表达式，但不返回任何结果。大部分情况下，DO 相当于不返回结果的 SELECT expr, ...,。

> **注意：**
>
> DO 只能执行表达式，所以不是所有能够用 SELECT 的地方都能用 DO 替换。例如 DO id FROM t1 就是不是合法的 SQL 语句，因为它引用了一张表。

DO 在 MySQL 中的一个主要应用场景是存储过程或者触发器。因为 TiDB 当前不支持存储过程和触发器，所以 DO 的实际使用场景较少。

2.2.35.1 语法图

```
DoStmt   ::= 'DO' ExpressionList

ExpressionList ::=
    Expression ( ',' Expression )*

Expression ::=
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        ↪ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    ↪ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?
```

2.2.35.2 示例

这条 SELECT 语句会暂停执行，但同时也会返回一个结果集。

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

```
+----------+
| SLEEP(5) |
+----------+
|        0 |
+----------+
1 row in set (5.00 sec)
```

如果使用 DO 的话，语句同样会暂停，但不会返回结果集。

```
DO SLEEP(5);
```

```
Query OK, 0 rows affected (5.00 sec)
```

```
DO SLEEP(1), SLEEP(1.5);
```

```
Query OK, 0 rows affected (2.50 sec)
```

2.2.35.3 MySQL 兼容性

DO 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.35.4 另请参阅

- SELECT

2.2.36 DROP [GLOBAL|SESSION] BINDING

DROP BINDING 语句用于删除指定的 SQL 绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内删除执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

2.2.36.1 语法图

```
DropBindingStmt ::=
    'DROP' GlobalScope 'BINDING' 'FOR' BindableStmt ( 'USING' BindableStmt )?

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

2.2.36.2 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
```

```
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object        |
    ↪ execution info                                                                  | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                      | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                                       | range:[123,123],
    ↪ keep order:false | N/A              | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                                     | keep order:false
    ↪                     | N/A              | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+-----------+---------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows   | actRows | task      | access object | execution info
    ↪                                                                          | operator info      |
```

```
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+--------------+-----------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |              | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |              | time:224ms, loops
    ↪ :298                                                        | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1     | time:220ms, loops
    ↪ :298                                                        | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+--------------+-----------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object       |
    ↪ execution info                                                          | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
```

```
    ↪                                    | 109.1484375 KB | N/A   |
| ├─IndexRangeScan_8(Build)     | 583.00   | 297      | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                          | range:[123,123], keep
    ↪ order:false | N/A             | N/A   |
| └─TableRowIDScan_9(Probe)     | 583.00   | 297      | cop[tikv] | table:t1             | time:0s
    ↪ , loops:4                                                          | keep order:false
    ↪                    | N/A             | N/A   |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
3 rows in set (0.01 sec)

SHOW SESSION BINDINGS\G
Empty set (0.00 sec)
```

2.2.36.3 MySQL 兼容性

`DROP [GLOBAL|SESSION] BINDING` 语句是 TiDB 对 MySQL 语法的扩展。

2.2.36.4 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

2.2.37 DROP COLUMN

`DROP COLUMN` 语句用于从指定的表中删除列。在 TiDB 中，`COLUMN` 为在线操作，不会阻塞表中的数据读写。

2.2.37.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName DropColumnSpec ( ',' DropColumnSpec )*

DropColumnSpec
         ::= 'DROP' 'COLUMN'? 'IF EXISTS'? ColumnName ( 'RESTRICT' | 'CASCADE' )?

ColumnName
         ::= Identifier ( '.' Identifier ( '.' Identifier )? )?
```

2.2.37.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, col1 INT NOT NULL, col2 INT NOT NULL
    ↪ );
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (col1,col2) VALUES (1,1),(2,2),(3,3),(4,4),(5,5);
```

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

```
SELECT * FROM t1;
```

```
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    2 |    2 |
|  3 |    3 |    3 |
|  4 |    4 |    4 |
|  5 |    5 |    5 |
+----+------+------+
5 rows in set (0.01 sec)
```

```
ALTER TABLE t1 DROP COLUMN col1, DROP COLUMN col2;
```

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

```
SELECT * FROM t1;
```

```
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    2 |    2 |
|  3 |    3 |    3 |
|  4 |    4 |    4 |
|  5 |    5 |    5 |
+----+------+------+
5 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP COLUMN col1;
```

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

```
SELECT * FROM t1;
```

```
+----+------+
| id | col2 |
+----+------+
|  1 |    1 |
|  2 |    2 |
|  3 |    3 |
|  4 |    4 |
|  5 |    5 |
+----+------+
5 rows in set (0.00 sec)
```

2.2.37.3 MySQL 兼容性

- 目前不支持在一条语句中同时删除多个列。
- 目前不支持删除主键列或组合索引相关列。

2.2.37.4 另请参阅

- ADD COLUMN
- SHOW CREATE TABLE
- CREATE TABLE

2.2.38 DROP DATABASE

DROP DATABASE 语句用于永久删除指定的数据库，以及删除所有在 schema 中创建的表和视图。与被删数据库相关联的用户权限不受影响。

2.2.38.1 语法图

```
DropDatabaseStmt ::=
    'DROP' 'DATABASE' IfExists DBName

IfExists ::= ( 'IF' 'EXISTS' )?
```

2.2.38.2 示例

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

```
DROP DATABASE test;
```

```
Query OK, 0 rows affected (0.25 sec)
```

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
+--------------------+
3 rows in set (0.00 sec)
```

2.2.38.3 MySQL 兼容性

DROP DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.38.4 另请参阅

- CREATE DATABASE
- ALTER DATABASE

2.2.39 DROP INDEX

DROP INDEX 语句用于从指定的表中删除索引，并在 TiKV 中将空间标记为释放。

2.2.39.1 语法图

```
AlterTableDropIndexStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' AlterTableDropIndexSpec ( ',' AlterTableDropIndexSpec )*
```

```
AlterTableDropIndexSpec
        ::= 'DROP' ( 'FOREIGN'? 'KEY' | 'INDEX' ) ( 'IF' 'EXISTS' )? Identifier
```

2.2.39.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

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

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪               |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+------------------------+---------+-----------+-----------------------+--------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                              |
+------------------------+---------+-----------+-----------------------+--------------------------------
    ↪
```

```
| IndexReader_6          | 0.01    | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5     | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+------------------------+---------+-----------+------------------------+-----------------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

2.2.39.3 MySQL 兼容性

- 不支持删除 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

2.2.39.4 另请参阅

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

2.2.40 DROP PLACEMENT POLICY

DROP PLACEMENT POLICY 用于删除已创建的放置策略。

2.2.40.1 语法图

```
DropPolicyStmt ::=
    "DROP" "PLACEMENT" "POLICY" IfExists PolicyName

PolicyName ::=
    Identifier
```

2.2.40.2 示例

删除放置规则时，确保该策略未被任何表或分区引用，否则会删除失败。

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=4;
CREATE TABLE t1 (a INT PRIMARY KEY) PLACEMENT POLICY=p1;
DROP PLACEMENT POLICY p1;  -- 该语句执行失败，因为放置规则 p1 被引用。
```

```
-- 查看引用放置规则的表和分区。
SELECT table_schema, table_name FROM information_schema.tables WHERE tidb_placement_policy_name='
    ↪ p1';
SELECT table_schema, table_name FROM information_schema.partitions WHERE
    ↪ tidb_placement_policy_name='p1';

ALTER TABLE t1 PLACEMENT POLICY=default;  -- 移除表 t1 上的默认放置规则。
DROP PLACEMENT POLICY p1;  -- 执行成功。
```

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

Query OK, 0 rows affected (0.11 sec)

ERROR 8241 (HY000): Placement policy 'p1' is still in use

+--------------+------------+
| table_schema | table_name |
+--------------+------------+
| test         | t1         |
+--------------+------------+
1 row in set (0.00 sec)

Empty set (0.01 sec)

Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.21 sec)
```

2.2.40.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.40.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- ALTER PLACEMENT POLICY

2.2.41 DROP ROLE

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

2.2.41.1 语法图

```
DropRoleStmt ::=
    'DROP' 'ROLE' ( 'IF' 'EXISTS' )? RolenameList

RolenameList ::=
    Rolename ( ',' Rolename )*
```

2.2.41.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
```

```
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

删除角色 `analyticsteam`：

```
$ mysql -uroot

DROP ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
```

Jennifer 不再具有与 analyticsteam 关联的默认角色，或不能再将 analyticsteam 设为启用角色：

```
$ mysql -ujennifer

SHOW GRANTS;
+--------------------------------------+
| Grants for User                      |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)

SET ROLE analyticsteam;
ERROR 3530 (HY000): `analyticsteam`@`%` is is not granted to jennifer@%
```

2.2.41.3 MySQL 兼容性

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

2.2.41.4 另请参阅

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

2.2.42 DROP SEQUENCE

DROP SEQUENCE 语句用于删除序列对象。

2.2.42.1 语法图

```
DropSequenceStmt ::=
    'DROP' 'SEQUENCE' IfExists TableNameList

IfExists ::= ( 'IF' 'EXISTS' )?
```

```
TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

2.2.42.2 示例

```
DROP SEQUENCE seq;
```

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

```
DROP SEQUENCE seq, seq2;
```

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

2.2.42.3 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

2.2.42.4 另请参阅

- CREATE SEQUENCE
- SHOW CREATE SEQUENCE

2.2.43 DROP STATS

DROP STATS 语句用于从当前所选定的数据库中删除选定表的统计信息。

2.2.43.1 语法图

```
DropStatsStmt ::=
    'DROP' 'STATS' TableName

TableName ::=
    Identifier ('.' Identifier)?
```

2.2.43.2 示例

```
CREATE TABLE t(a INT);
```

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

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          |                | 2020-05-25 20:34:33 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
1 row in set (0.00 sec)
```

```
DROP STATS t;
```

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

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

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

2.2.43.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.43.4 另请参阅

- 统计信息简介

2.2.44 DROP TABLE

`DROP TABLE` 语句用于从当前所选的数据库中删除表。如果表不存在则会报错，除非使用 `IF EXISTS` 修饰符。

2.2.44.1 语法图

```
DropTableStmt ::=
    'DROP' OptTemporary TableOrTables IfExists TableNameList RestrictOrCascadeOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

TableOrTables ::=
    'TABLE'
|   'TABLES'

TableNameList ::=
    TableName ( ',' TableName )*
```

2.2.44.2 删除临时表

删除普通表和临时表的用法如下：

- `DROP TEMPORARY TABLE` 只能删除本地临时表
- `DROP GLOBAL TEMPORARY TABLE` 只能删除全局临时表
- `DROP TABLE` 可以删除普通表或临时表

2.2.44.3 示例

```
CREATE TABLE t1 (a INT);
```

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

```
DROP TABLE t1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
DROP TABLE table_not_exists;
```

```
ERROR 1051 (42S02): Unknown table 'test.table_not_exists'
```

```
DROP TABLE IF EXISTS table_not_exists;
```

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

```
CREATE VIEW v1 AS SELECT 1;
```

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

```
DROP TABLE v1;
```

```
Query OK, 0 rows affected (0.23 sec)
```

2.2.44.4 MySQL 兼容性

- 在尝试删除不存在的表时，使用 `IF EXISTS` 删除表不会返回警告。Issue #7867
- 目前 `RESTRICT` 和 `CASCADE` 仅在语法上支持。

2.2.44.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- SHOW TABLES

2.2.45 DROP USER

DROP USER 语句用于从 TiDB 系统数据库中删除用户。如果用户不存在，使用关键词 IF EXISTS 可避免出现警告。执行 DROP USER 语句需要拥有 CREATE USER 权限。

2.2.45.1 语法图

```
DropUserStmt ::=
    'DROP' 'USER' ( 'IF' 'EXISTS' )? UsernameList

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces
```

2.2.45.2 示例

```
DROP USER 'idontexist';
```

```
ERROR 1396 (HY000): Operation DROP USER failed for idontexist@%
```

```
DROP USER IF EXISTS 'idontexist';
```

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

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

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

```
GRANT ALL ON test.* TO 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

```
REVOKE ALL ON test.* FROM 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------+
| Grants for newuser@%                |
+-------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+-------------------------------------+
1 row in set (0.00 sec)
```

```
DROP USER 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

2.2.45.3 MySQL 兼容性

- 在 TiDB 中删除不存在的用户时，使用 `IF EXISTS` 可避免出现警告。Issue #10196

2.2.45.4 另请参阅

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

2.2.46 DROP VIEW

`DROP VIEW` 语句用于从当前所选定的数据库中删除视图对象。视图所引用的基表不受影响。

2.2.46.1 语法图

```
DropViewStmt ::=
    'DROP' 'VIEW' ( 'IF' 'EXISTS' )? TableNameList RestrictOrCascadeOpt

TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

2.2.46.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE c1 > 2;
```

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

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

```
DROP VIEW v1;
```

```
Query OK, 0 rows affected (0.23 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

2.2.46.3 MySQL 兼容性

`DROP VIEW` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.46.4 See also

- CREATE VIEW
- DROP TABLE

2.2.47 EXECUTE

`EXECUTE` 语句为服务器端预处理语句提供 SQL 接口。

2.2.47.1 语法图

```
ExecuteStmt ::=
    'EXECUTE' Identifier ( 'USING' UserVariable ( ',' UserVariable )* )?
```

2.2.47.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

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

```
SET @number = 5;
```

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

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

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

2.2.47.3 MySQL 兼容性

EXECUTE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.47.4 另请参阅

- PREPARE
- DEALLOCATE

2.2.48 EXPLAIN ANALYZE

EXPLAIN ANALYZE 语句的工作方式类似于 EXPLAIN，主要区别在于前者实际上会执行语句。这样可以将查询计划中的估计值与执行时所遇到的实际值进行比较。如果估计值与实际值显著不同，那么应考虑在受影响的表上运行 ANALYZE TABLE。

> **注意：**
>
> 在使用 EXPLAIN ANALYZE 执行 DML 语句时，数据的修改操作会被正常执行。但目前 DML 语句还无法展示执行计划。

2.2.48.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|    'DESC'

ExplainStmt ::=
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        ↪ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        ↪ ExplainableStmt ) )
```

```
ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

2.2.48.2 EXPLAIN ANALYZE 输出格式

和 EXPLAIN 不同，EXPLAIN ANALYZE 会执行对应的 SQL 语句，记录其运行时信息，和执行计划一并返回出来。因此，可以将 EXPLAIN ANALYZE 视为 EXPLAIN 语句的扩展。EXPLAIN ANALYZE 语句的返回结果相比 EXPLAIN，增加了 actRows，execution info，memory，disk 这几列信息：

属性名	含义
actRows	算子实际输出的数据条数。
execution info	算子的实际执行信息。time 表示从进入算子到离开算子的全部 wall time，包括所有子算子操作的全部执行时间。如果该算子被父算子多次调用 (loops)，这个时间就是累积的时间。loops 是当前算子被父算子调用的次数。
memory	算子占用内存空间的大小。
disk	算子占用磁盘空间的大小。

2.2.48.3 示例

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

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+---------+------+--------------+-------------------------------------------------------------
    ↪
| id          | estRows | actRows | task | access object | execution info
    ↪                                                    | operator info | memory | disk |
+-------------+---------+---------+------+--------------+-------------------------------------------------------------
    ↪
```

```
| Point_Get_1 | 1.00    | 1       | root | table:t1      | time:757.205µs, loops:2, Get:{num_rpc
    ↪ :1, total_time:697.051µs} | handle:1      | N/A    | N/A  |
+-------------+---------+---------+------+---------------+-----------------------------------------------
    ↪
1 row in set (0.01 sec)
```

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

```
+------------------+----------+---------+-----------+---------------+--------------------------------------
    ↪
| id               | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                  | memory    | disk |
+------------------+----------+---------+-----------+---------------+--------------------------------------
    ↪
| TableReader_5     | 10000.00 | 3       | root      |               | time:278.2µs, loops:2,
    ↪ cop_task: {num: 1, max: 437.6µs, proc_keys: 3, rpc_num: 1, rpc_time: 423.9µs,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:TableFullScan_4           | 251 Bytes | N/A  |
| └─TableFullScan_4 | 10000.00 | 3       | cop[tikv] | table:t1      | tikv_task:{time:0s, loops
    ↪ :1}, scan_detail: {total_process_keys: 3, total_process_keys_size: 111, total_keys: 4,
    ↪ rocksdb: {delete_skipped_count: 0, key_skipped_count: 3, block: {cache_hit_count: 0,
    ↪ read_count: 0, read_byte: 0 Bytes}}} | keep order:false, stats:pseudo | N/A       | N/A
    ↪ |
+------------------+----------+---------+-----------+---------------+--------------------------------------
    ↪
2 rows in set (0.00 sec)
```

2.2.48.4 算子执行信息介绍

`execution info` 信息除了基本的 `time` 和 `loop` 信息外，还包含算子特有的执行信息，主要包含了该算子发送 RPC 请求的耗时信息以及其他步骤的耗时。

2.2.48.4.1 Point_Get

`Point_Get` 算子可能包含以下执行信息：

- `Get:{num_rpc:1, total_time:697.051µs}`：向 TiKV 发送 Get 类型的 RPC 请求的数量 (num_rpc) 和所有 RPC 请求的总耗时 (total_time)。
- `ResolveLock:{num_rpc:1, total_time:12.117495ms}`：读数据遇到锁后，进行 resolve lock 的时间。一般在读写冲突的场景下会出现。
- `regionMiss_backoff:{num:11, total_time:2010 ms},tikvRPC_backoff:{num:11, total_time:10691 ms}`：RPC 请求失败后，会在等待 backoff 的时间后重试，包括了 backoff 的类型（如 regionMiss，tikvRPC），backoff 等待的总时间 (total_time) 和 backoff 的总次数 (num)。

2.2.48.4.2 Batch_Point_Get

Batch_Point_get 算子的执行信息和 Point_Get 算子类似，不过 Batch_Point_Get 一般向 TiKV 发送 BatchGet 类型的 RPC 请求来读取数据。

BatchGet:{num_rpc:2, total_time:83.13µs}：向 TiKV 发送 BatchGet 类型的 RPC 请求的数量 (num_rpc) 和所有 RPC 请求的总耗时 (total_time)。

2.2.48.4.3 TableReader

TableReader 算子可能包含以下执行信息：

```
cop_task: {num: 6, max: 1.07587ms, min: 844.312µs, avg: 919.601µs, p95: 1.07587ms, max_proc_keys:
    ↪  16, p95_proc_keys: 16, tot_proc: 1ms, tot_wait: 1ms, rpc_num: 6, rpc_time: 5.313996ms,
    ↪ copr_cache_hit_ratio: 0.00}
```

- cop_task：包含 cop task 的相关信息，如：
  - num：cop task 的数量
  - max,min,avg,p95：所有 cop task 中执行时间的最大值，最小值，平均值和 P95 值。
  - max_proc_keys, p95_proc_keys：所有 cop task 中 tikv 扫描 kv 数据的最大值，P95 值，如果 max 和 p95 的值差距很大，说明数据分布不太均匀。
  - rpc_num, rpc_time：向 TiKV 发送 Cop 类型的 RPC 请求总数量和总时间。
  - copr_cache_hit_ratio：cop task 请求的 Coprocessor Cache 缓存命中率。Coprocessor Cache 配置。
- backoff：包含不同类型的 backoff 以及等待总耗时。

2.2.48.4.4 Insert

Insert 算子可能包含以下执行信息：

```
prepare:109.616µs, check_insert:{total_time:1.431678ms, mem_insert_time:667.878µs, prefetch:763.8
    ↪ µs, rpc:{BatchGet:{num_rpc:1, total_time:699.166µs},Get:{num_rpc:1, total_time:378.276µs
    ↪ }}}
```

- prepare：准备写入前的耗时，包括表达式，默认值相关的计算等。
- check_insert：这个信息一般出现在 insert ignore 和 insert on duplicate 语句中，包含冲突检查和写入 TiDB 事务缓存的耗时。注意，这个耗时不包含事务提交的耗时。具体包含以下信息：
  - total_time：check_insert 步骤的总耗时。
  - mem_insert_time：将数据写入 TiDB 事务缓存的耗时。
  - prefetch：从 TiKV 中获取需要检查冲突的数据的耗时，该步骤主要是向 TiKV 发送 BatchGet 类型的 RPC 请求的获取数据。
  - rpc：向 TiKV 发送 RPC 请求的总耗时，一般包含 BatchGet 和 Get 两种类型的 RPC 耗时，其中：
    * BatchGet 请求是 prefetch 步骤发送的 RPC 请求。
    * Get 请求是 insert on duplicate 语句在执行 duplicate update 时发送的 RPC 请求。
- backoff：包含不同类型的 backoff 以及等待总耗时。

2.2.48.4.5 IndexJoin

IndexJoin 算子有 1 个 outer worker 和 N 个 inner worker 并行执行，其 join 结果的顺序和 outer table 的顺序一致，具体执行流程如下：

1. Outer worker 读取 N 行 outer table 的数据，然后包装成一个 task 发送给 result channel 和 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收 task，然后根据 task 生成需要读取 inner table 的 key ranges 范围，然后读取相应范围的 inner table 行数据，并生成一个 inner table row 的 hash table。
3. IndexJoin 的主线程从 result channel 中接收 task，然后等待 inner worker 执行完这个 task。
4. IndexJoin 的主线程用 outer table rows 和 inner table rows 的 hash table 做 join。

IndexJoin 算子包含以下执行信息：

```
inner:{total:4.297515932s, concurrency:5, task:17, construct:97.96291ms, fetch:4.164310088s,
    ↪ build:35.219574ms}, probe:53.574945ms
```

- inner：inner worker 的执行信息，具体如下：
  - total：inner worker 的总耗时。
  - concurrency：inner worker 的数量。
  - task：inner worker 处理 task 的总数量。
  - construct：inner worker 读取 task 对应的 inner table rows 之前的准备时间。
  - fetch：inner worker 读取 inner table rows 的总耗时。
  - build: inner worker 构造 inner table rows 对应的 hash table 的总耗时。
- probe：IndexJoin 主线程用 outer table rows 和 inner table rows 的 hash table 做 join 的总耗时。

2.2.48.4.6 IndexHashJoin

IndexHashJoin 算子和 IndexJoin 算子执行流程类似，也有 1 个 outer worker 和 N 个 inner worker 并行执行，但是其 join 结果的顺序是不和 outer table 一致。具体执行流程如下：

1. Outer worker 读取 N 行 out table 的数据，然后包装成一个 task 发送给 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收 task，然后做以下三件事情，其中步骤 a 和 b 是并行执行。
   a. 用 outer table rows 生成一个 hash table。
   b. 根据 task 生成 key 的范围后，读取 inner table 相应范围的行数据。
   c. 用 inner table rows 和 outer table rows 的 hash table 做 join，然后把 join 结果发送给 result channel。
3. IndexHashJoin 的主线程从 result channel 中接收 join 结果。

IndexHashJoin 算子包含以下执行信息：

```
inner:{total:4.429220003s, concurrency:5, task:17, construct:96.207725ms, fetch:4.239324006s,
    ↪ build:24.567801ms, join:93.607362ms}
```

- inner：inner worker 的执行信息，具体如下：

  - `total`：inner worker 的总耗时。
  - `concurrency`：inner worker 的数量。
  - `task`：inner worker 处理 task 的总数量。
  - `construct`：inner worker 读取 task 对应的 inner table rows 之前的准备时间。
  - `fetch`：inner worker 读取 inner table rows 的总耗时。
  - build: inner worker 构造 outer table rows 对应的 hash table 的总耗时。
  - `join`: inner worker 用 inner table rows 和 outer table rows 的 hash table 做 join 的总耗时。

2.2.48.4.7 HashJoin

HashJoin 算子有一个 inner worker，一个 outer worker 和 N 个 join worker，其具体执行逻辑如下：

1. inner worker 读取 inner table rows 并构造 hash table。
2. outer worker 读取 outer table rows, 然后包装成 task 发送给 join worker。
3. 等待第 1 步的 hash table 构造完成。
4. join worker 用 task 里面的 outer table rows 和 hash table 做 join，然后把 join 结果发送给 result channel。
5. HashJoin 的主线程从 result channel 中接收 join 结果。

HashJoin 算子包含以下执行信息：

```
build_hash_table:{total:146.071334ms, fetch:110.338509ms, build:35.732825ms}, probe:{concurrency
    ↪ :5, total:857.162518ms, max:171.48271ms, probe:125.341665ms, fetch:731.820853ms}
```

- build_hash_table: 读取 inner table 的数据并构造 hash table 的执行信息：
  - `total`：总耗时。
  - `fetch`：读取 inner table 数据的总耗时。
  - `build`：构造 hash table 的总耗时。
- probe: join worker 的执行信息：
  - `concurrency`：join worker 的数量。
  - `total`：所有 join worker 执行的总耗时。
  - `max`：单个 join worker 执行的最大耗时。
  - probe: 用 outer table rows 和 hash table 做 join 的总耗时。
  - `fetch`：join worker 等待读取 outer table rows 数据的总耗时。

2.2.48.4.8 lock_keys 执行信息

在悲观事务中执行 DML 语句时，算子的执行信息还可能包含 `lock_keys` 的执行信息，示例如下：

```
lock_keys: {time:94.096168ms, region:6, keys:8, lock_rpc:274.503214ms, rpc_count:6}
```

- `time`：执行 `lock_keys` 操作的总耗时。
- `region`：执行 `lock_keys` 操作涉及的 Region 数量。
- `keys`：需要 Lock 的 Key 的数量。
- `lock_rpc`：向 TiKV 发送 Lock 类型的 RPC 总耗时。因为可以并行发送多个 RPC 请求，所以总 RPC 耗时可能比 `lock_keys` 操作总耗时大。
- `rpc_count`：向 TiKV 发送 Lock 类型的 RPC 总数量。

2.2.48.4.9 commit_txn 执行信息

在 autocommit=1 的事务中执行写入类型的 DML 语句时，算子的执行信息还会包括事务提交的耗时信息，示例如下：

```
commit_txn: {prewrite:48.564544ms, wait_prewrite_binlog:47.821579, get_commit_ts:4.277455ms,
    ↪ commit:50.431774ms, region_num:7, write_keys:16, write_byte:536}
```

- prewrite：事务 2PC 提交阶段中 prewrite 阶段的耗时。
- wait_prewrite_binlog:：等待写 prewrite Binlog 的耗时。
- get_commit_ts：获取事务提交时间戳的耗时。
- commit：事务 2PC 提交阶段中，commit 阶段的耗时。
- write_keys：事务中写入 key 的数量。
- write_byte：事务中写入 key-value 的总字节数量，单位是 byte。

2.2.48.4.10 其它常见执行信息

Coprocessor 算子通常包含 cop_task 和 tikv_task 两部分执行时间信息。前者是 TiDB 端记录的时间，从发出请求到接收回复；后者是 TiKV Coprocessor 算子自己记录的时间。两者相差较大可能说明在等待、gRPC 或网络上耗时较长。

2.2.48.5 MySQL 兼容性

EXPLAIN ANALYZE 是 MySQL 8.0 的功能，但该语句在 MySQL 中的输出格式和可能的执行计划都与 TiDB 有较大差异。

2.2.48.6 另请参阅

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

2.2.49 EXPLAIN

EXPLAIN 语句仅用于显示查询的执行计划，而不执行查询。EXPLAIN ANALYZE 可执行查询，补充 EXPLAIN 语句。如果 EXPLAIN 的输出与预期结果不匹配，可考虑在查询的每个表上执行 ANALYZE TABLE。

语句 DESC 和 DESCRIBE 是 EXPLAIN 的别名。EXPLAIN <tableName> 的替代用法记录在SHOW [FULL] COLUMNS FROM 下。

TiDB 支持 EXPLAIN [options] FOR CONNECTION connection_id，但与 MySQL 的 EXPLAIN FOR 有一些区别，请参见EXPLAIN FOR CONNECTION。

2.2.49.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|   'DESC'

ExplainStmt ::=
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        ↪ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        ↪ ExplainableStmt ) )

ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

2.2.49.2 EXPLAIN 输出格式

> **注意：**
>
> 使用 MySQL 客户端连接到 TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 `EXPLAIN` 的输出结果不再换行，可按右箭头 → 键水平滚动阅读输出结果。

目前 TiDB 的 `EXPLAIN` 会输出 5 列，分别是：`id`，`estRows`，`task`，`access object`，`operator info`。执行计划中每个算子都由这 5 列属性来描述，`EXPLAIN` 结果中每一行描述一个算子。每个属性的具体含义如下：

属性名	含义
id	算子的 ID，是算子在整个执行计划中唯一的标识。在 TiDB 2.1 中，ID 会格式化地显示算子的树状结构。数据从孩子结点流向父亲结点，每个算子的父亲结点有且仅有一个。
estRows	算子预计将会输出的数据条数，基于统计信息以及算子的执行逻辑估算而来。在 4.0 之前叫 count。
task	算子属于的 task 种类。目前的执行计划分成为两种 task，一种叫 root task，在 tidb-server 上执行，一种叫 cop task，在 TiKV 或者 TiFlash 上并行执行。当前的执行计划在 task 级别的拓扑关系是一个 root task 后面可以跟许多 cop task，root task 使用 cop task 的输出结果作为输入。cop task 中执行的也即是 TiDB 下推到 TiKV 或者 TiFlash 上的任务，每个 cop task 分散在 TiKV 或者 TiFlash 集群中，由多个进程共同执行。
access object	算子所访问的数据项信息。包括表 `table`，表分区 `partition` 以及使用的索引 `index`（如果有）。只有直接访问数据的算子才拥有这些信息。

属性名	含义
operator info	算子的其它信息。各个算子的 operator info 各有不同，可参考下面的示例解读。

2.2.49.3 示例

```
EXPLAIN SELECT 1;
```

```
+------------------+---------+------+---------------+---------------+
| id               | estRows | task | access object | operator info |
+------------------+---------+------+---------------+---------------+
| Projection_3     | 1.00    | root |               | 1->Column#1   |
| └─TableDual_4    | 1.00    | root |               | rows:1        |
+------------------+---------+------+---------------+---------------+
2 rows in set (0.00 sec)
```

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

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

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
DESC SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
DESCRIBE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
EXPLAIN INSERT INTO t1 (c1) VALUES (4);
```

```
+----------+---------+------+---------------+---------------+
| id       | estRows | task | access object | operator info |
+----------+---------+------+---------------+---------------+
| Insert_1 | N/A     | root |               | N/A           |
+----------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

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

```
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                       | estRows | task      | access object | operator info
    ↪                  |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| Update_4                 | N/A     | root      |               | N/A
    ↪                            |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    ↪               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

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

```
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                       | estRows | task      | access object | operator info
    ↪                  |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
```

```
| Delete_4                  | N/A     | root      |               | N/A
    ↪                             |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    ↪               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+---------------------------+---------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.01 sec)
```

如果未指定 FORMAT，或未指定 FORMAT ="row"，那么 EXPLAIN 语句将以表格格式输出结果。更多信息，可参阅 Understand the Query Execution Plan。

除 MySQL 标准结果格式外，TiDB 还支持 DotGraph。需按照下列所示指定 FORMAT ="dot"：

```
create table t(a bigint, b bigint);
explain format = "dot" select A.a, B.b from t A join t B on A.a > B.b where A.a < 10;
```

```
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
| dot contents
    ↪
    ↪ |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
|
digraph Projection_8 {
subgraph cluster8{
node [style=filled, color=lightgrey]
color=black
label = "root"
"Projection_8" -> "HashJoin_9"
"HashJoin_9" -> "TableReader_13"
"HashJoin_9" -> "Selection_14"
"Selection_14" -> "TableReader_17"
}
subgraph cluster12{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_12" -> "TableFullScan_11"
}
subgraph cluster16{
node [style=filled, color=lightgrey]
```

```
color=black
label = "cop"
"Selection_16" -> "TableFullScan_15"
}
"TableReader_13" -> "Selection_12"
"TableReader_17" -> "Selection_16"
}
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

如果你的计算机上安装了 dot 程序（在 graphviz 包中），可使用以下方法生成 PNG 文件：

```
dot xx.dot -T png -O
```

The `xx.dot` is the result returned by the above statement.

如果你的计算机上未安装 dot 程序，可将结果复制到本网站以获取树形图：

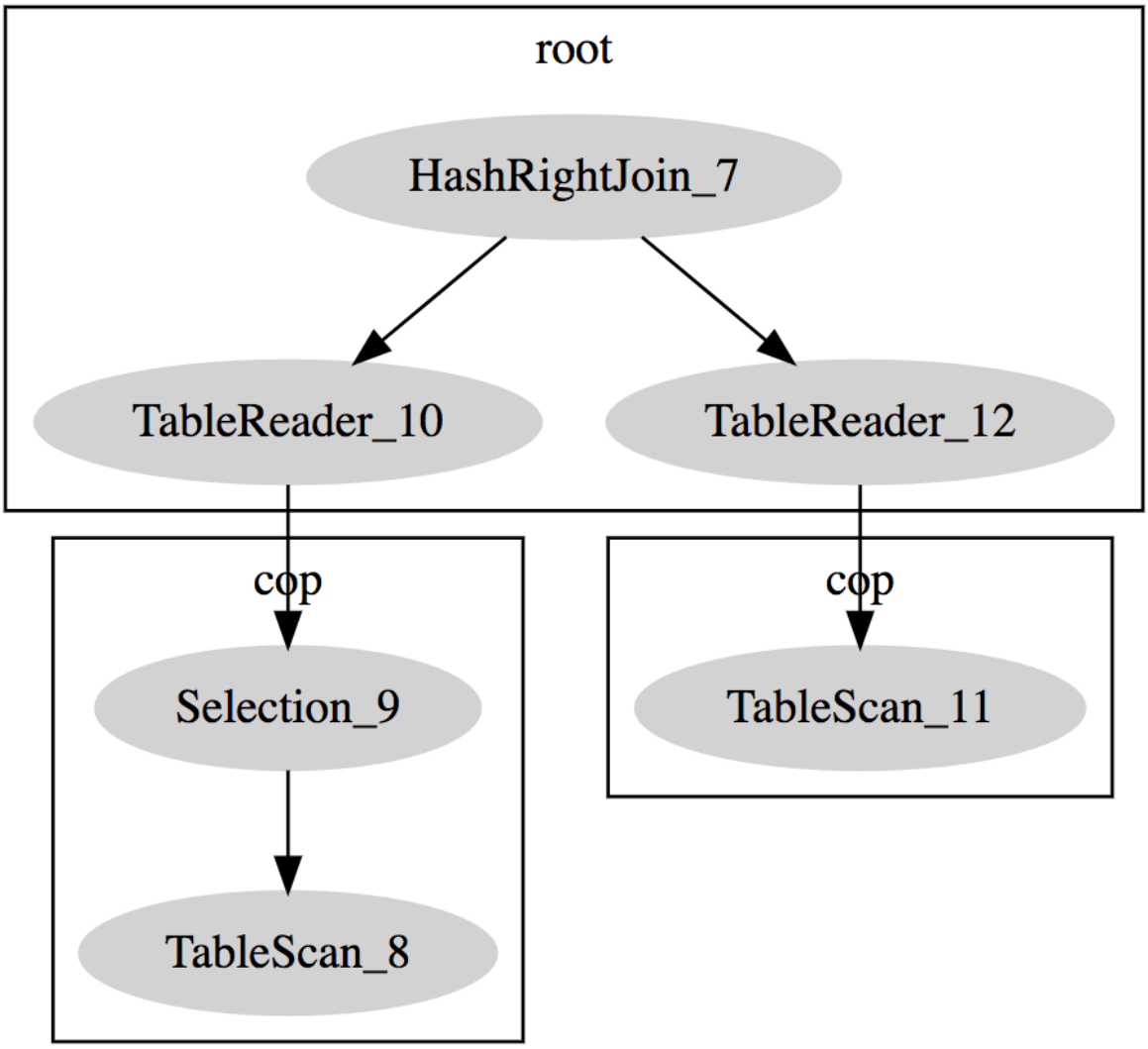


图 1: Explain Dot

2.2.49.4 MySQL 兼容性

- EXPLAIN 的格式和 TiDB 中潜在的执行计划都与 MySQL 有很大不同。
- TiDB 不像 MySQL 那样支持 EXPLAIN FORMAT = JSON。

2.2.49.5 EXPLAIN FOR CONNECTION

EXPLAIN FOR CONNECTION 用于获得一个连接中当前正在执行 SQL 的执行计划或者是最后执行 SQL 的执行计划，其输出格式与 EXPLAIN 完全一致。但 TiDB 中的实现与 MySQL 不同，除了输出格式之外，还有以下区别：

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

- MySQL 的文档中指出，MySQL 要求登录用户与被查询的连接相同，或者拥有 PROCESS 权限，而 TiDB 则要求登录用户与被查询的连接相同，或者拥有 SUPER 权限。

2.2.49.6 另请参阅

- 理解 TiDB 执行计划
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE

2.2.50 FLASHBACK TABLE

在 TiDB 4.0 中，引入了 FLASHBACK TABLE 语法，其功能是在 Garbage Collection (GC) life time 时间内，可以用 FLASHBACK TABLE 语句来恢复被 DROP 或 TRUNCATE 删除的表以及数据。

可以使用系统变量`tidb_gc_life_time`配置数据的历史版本的保留时间（默认值是 `10m0s`）。可以使用以下 SQL 语句查询当前的 `safePoint`，即 GC 已经清理到的时间点：

```
sql  SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
```

只要被 DROP 或 TRUNCATE 删除的表是在 `tikv_gc_safe_point` 时间之后，都能用 FLASHBACK TABLE 语法来恢复。

2.2.50.1 语法

```
FLASHBACK TABLE table_name [TO other_table_name]
```

2.2.50.1.1 语法图

```
FlashbackTableStmt ::=
    'FLASHBACK' 'TABLE' TableName FlashbackToNewName

TableName ::=
    Identifier ( '.' Identifier )?

FlashbackToNewName ::=
    ( 'TO' Identifier )?
```

2.2.50.2 注意事项

如果删除了一张表并过了 GC lifetime，就不能再用 FLASHBACK TABLE 语句来恢复被删除的数据了，否则会返回错误，错误类似于 `Can't find dropped/truncated table 't' in GC safe point 2020-03-16 16:34:52 +0800 CST`。

在开启 TiDB Binlog 时使用 FLASHBACK TABLE 需要注意以下情况：

- 下游从集群也支持 FLASHBACK TABLE

- 从集群的 GC lifetime 一定要长于主集群的 GC lifetime。上下游同步存在的延迟可能也会造成下游恢复数据失败。

如果 Binlog 同步出错，则需要在 Binlog 过滤掉该表，同时手动全量重新导入该表的数据。

2.2.50.3 示例

- 恢复被 DROP 删除的表数据：

```
DROP TABLE t;
```

```
FLASHBACK TABLE t;
```

- 恢复被 TRUNCATE 的表数据，由于被 TRUNCATE 的表还存在，所以需要重命名被恢复的表，否则会报错表 t 已存在。

```
TRUNCATE TABLE t;
```

```
FLASHBACK TABLE t TO t1;
```

2.2.50.4 工作原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 `mysql.gc_delete_range` 表。TiDB 后台的 GC Worker 会定期从 `mysql.gc_delete_range` 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，FLASHBACK TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 `mysql.gc_delete_range` 表中相应的行记录即可。恢复表的元信息可以用 TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。下面是 `FLASHBACK TABLE t TO t1` 的工作流程：

1. 从 DDL History job 中查找 `drop table` 或者 `truncate table` 类型的操作，且操作的表名是 t 的第一个 DDL job，若没找到，则返回错误。
2. 检查 DDL job 的开始时间，是否在 `tikv_gc_safe_point` 之前。如果是`tikv_gc_safe_point` 之前，说明被 DROP 或 TRUNCATE 删除的表已经被 GC 清理掉，返回错误。
3. 用 DDL job 的开始时间作为 snapshot 读取历史数据，读取表的元信息。
4. 删除 `mysql.gc_delete_range` 中和表 t 相关的 GC 任务。
5. 将表的元信息中的 name 修改成 `t1`，并用该元信息新建一个表。注意：这里只是修改了表名，但是 table ID 不变，依旧是之前被删除的表 t 的 table ID。

可以发现，从表 t 被删除，到表 t 被 FLASHBACK 恢复到 `t1`，一直都是对表的元信息进行操作，而表的用户数据一直未被修改过。被恢复的表 `t1` 和之前被删除的表 t 的 table ID 相同，所以表 `t1` 才能读取表t 的用户数据。

> **注意：**
>
> 不能用 FLASHBACK 多次恢复同一个被删除的表，因为 FLASHBACK 所恢复表的 table ID 还是被删除表的 table ID，而 TiDB 要求所有还存在的表 table ID 必须全局唯一。

FLASHBACK TABLE 是通过快照读获取表的元信息后，再走一次类似于 CREATE TABLE 的建表流程，所以 FLASHBACK TABLE 实际上也是一种 DDL 操作。

2.2.50.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.51 FLUSH PRIVILEGES

FLUSH PRIVILEGES 语句可触发 TiDB 从权限表中重新加载权限的内存副本。在对如 mysql.user 一类的表进行手动编辑后，应当执行 FLUSH PRIVILEGES。使用如 GRANT 或 REVOKE 一类的权限语句后，不需要执行 FLUSH PRIVILEGES 语句。执行 FLUSH PRIVILEGES 语句的用户需要拥有 RELOAD 权限。

2.2.51.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

2.2.51.2 示例

```
FLUSH PRIVILEGES;
```

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

2.2.51.3 MySQL 兼容性

FLUSH PRIVILEGES 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.51.4 另请参阅

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

2.2.52 FLUSH STATUS

FLUSH STATUS 语句用于提供 MySQL 兼容性，但在 TiDB 上并无作用。因为 TiDB 使用 Prometheus 和 Grafana 而非 SHOW STATUS 来进行集中度量收集。

2.2.52.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

2.2.52.2 示例

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher_list    |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| Ssl_cipher         |                                      |
+--------------------+--------------------------------------+
6 rows in set (0.01 sec)
```

```
show global status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
```

```
| Ssl_version        |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

```
flush status;
```

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

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| ddl_schema_version | 141                                  |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

2.2.52.3 MySQL 兼容性

- FLUSH STATUS 语句仅用于提供 MySQL 兼容性。

2.2.52.4 另请参阅

- SHOW [GLOBAL|SESSION] STATUS

2.2.53 FLUSH TABLES

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

2.2.53.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?
```

```
FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|    LogTypeOpt 'LOGS'
|    TableOrTables TableNameListOpt WithReadLockOpt

LogTypeOpt ::=
    ( 'BINARY' | 'ENGINE' | 'ERROR' | 'GENERAL' | 'SLOW' )?

TableOrTables ::=
    'TABLE'
|   'TABLES'

TableNameListOpt ::=
    TableNameList?

WithReadLockOpt ::=
    ( 'WITH' 'READ' 'LOCK' )?
```

2.2.53.2 示例

```
FLUSH TABLES;
```

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

```
FLUSH TABLES WITH READ LOCK;
```

```
ERROR 1105 (HY000): FLUSH TABLES WITH READ LOCK is not supported.  Please use @@tidb_snapshot
```

2.2.53.3 MySQL 兼容性

- TiDB 没有 MySQL 中的表缓存这一概念。所以，`FLUSH TABLES` 因 MySQL 兼容性会在 TiDB 中解析出但会被忽略掉。
- 因为 TiDB 目前不支持锁表，所以`FLUSH TABLES WITH READ LOCK` 语句会产生错误。建议使用Historical reads来实现锁表。

2.2.53.4 另请参阅

- Read historical data

2.2.54 GRANT <privileges>

GRANT <privileges> 语句用于为 TiDB 中已存在的用户分配权限。TiDB 中的权限系统同 MySQL 一样，都基于数据库/表模式来分配凭据。执行 GRANT <privileges> 语句需要拥有分配的权限，并且拥有 GRANT OPTION 权限。

2.2.54.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        ↪ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
    'ALL' 'PRIVILEGES'?
|    'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|   'TRIGGER'
|   'DELETE'
|   'DROP' 'ROLE'?
|    'PROCESS'
|   'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|   'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|    'GRANT' 'OPTION'
|    'REFERENCES'
|    'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|    'USAGE'
|   'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|   'EVENT'
|   'SHUTDOWN'

ObjectType ::=
    'TABLE'?
```

```
PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

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

2.2.54.2 示例

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

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

```
GRANT ALL ON test.* TO 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

2.2.54.3 MySQL 兼容性

- 与 MySQL 类似，USAGE 权限表示登录 TiDB 服务器的能力。
- 目前不支持列级权限。
- 与 MySQL 类似，不存在 NO_AUTO_CREATE_USER sql 模式时，GRANT 语句将在用户不存在时自动创建一个空密码的新用户。删除此 sql-mode（默认情况下已启用）会带来安全风险。

2.2.54.4 另请参阅

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

2.2.55 GRANT <role>

GRANT <role> 语句用于将之前创建的角色授予给现有用户。用户可以通过 SET ROLE <rolename> 语句拥有角色权限，或者通过 SET ROLE ALL 语句拥有被授予的所有角色。

2.2.55.1 语法图

```
GrantRoleStmt ::=
    'GRANT' RolenameList 'TO' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

2.2.55.2 示例

创建新角色 analyticsteam 和新用户 jennifer：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)
CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)
GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
```

```
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与某一角色相关联，这样该用户无需执行 SET ROLE 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
```

```
| t1             |
+----------------+
1 row in set (0.00 sec)
```

2.2.55.3 MySQL 兼容性

GRANT <role> 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.55.4 另请参阅

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

2.2.56 INSERT

使用 INSERT 语句在表中插入新行。

2.2.56.1 语法图

```
InsertIntoStmt ::=
    'INSERT' TableOptimizerHints PriorityOpt IgnoreOptional IntoOpt TableName
        ↪ PartitionNameListOpt InsertValues OnDuplicateKeyUpdate

TableOptimizerHints ::=
    hintComment?

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IgnoreOptional ::=
    'IGNORE'?

IntoOpt  ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?
```

```
InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        ↪ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*

OnDuplicateKeyUpdate ::=
    ( 'ON' 'DUPLICATE' 'KEY' 'UPDATE' AssignmentList )?
```

2.2.56.2 示例

```
CREATE TABLE t1 (a INT);
```

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

```
CREATE TABLE t2 LIKE t1;
```

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

```
INSERT INTO t1 VALUES (1);
```

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

```
INSERT INTO t1 (a) VALUES (1);
```

```
Query OK, 1 row affected (0.01 sec)
```

```
INSERT INTO t2 SELECT * FROM t1;
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

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

```
SELECT * FROM t2;
```

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

```
INSERT INTO t2 VALUES (2),(3),(4);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t2;
```

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

2.2.56.3 MySQL 兼容性

INSERT 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.56.4 另请参阅

- DELETE
- SELECT
- UPDATE
- REPLACE

2.2.57 KILL

KILL 语句可以终止当前 TiDB 集群中任意一个 TiDB 实例中的某个连接。

2.2.57.1 语法图

```
KillStmt ::= 'KILL' 'TIDB'? ( 'CONNECTION' | 'QUERY' )? CONNECTION_ID
```

2.2.57.2 示例

查询当前集群中所有所有活跃查询，并终止其中某一个连接：

```
SELECT ID, USER, INSTANCE, INFO FROM INFORMATION_SCHEMA.CLUSTER_PROCESSLIST;
```

```
+--------------------+------+----------------+-------------------------------------------------------------------
    ↪
| ID                 | USER | INSTANCE       | INFO
    ↪                                                                                 |
+--------------------+------+----------------+-------------------------------------------------------------------
    ↪
| 8306449708033769879 | root | 127.0.0.1:10082 | select sleep(30), 'foo'
    ↪                                                          |
| 5857102839209263511 | root | 127.0.0.1:10080 | select sleep(50)
    ↪                                                                    |
| 5857102839209263513 | root | 127.0.0.1:10080 | SELECT ID, USER, INSTANCE, INFO FROM
    ↪ INFORMATION_SCHEMA.CLUSTER_PROCESSLIST |
+--------------------+------+----------------+-------------------------------------------------------------------
    ↪
```

```
KILL 5857102839209263511;
```

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

2.2.57.3 MySQL 兼容性

- MySQL 的 KILL 语句仅能终止当前连接的 MySQL 实例上的连接，TiDB 的 KILL 语句能终止整个集群中任意一个 TiDB 实例上的连接。
- 暂时不支持使用 MySQL 命令行 ctrl+c 终止查询或连接。

2.2.57.4 行为变更说明

TiDB 从 v6.1.0 起新增 Global Kill 功能（由enable-global-kill 配置项控制，默认启用）。启用 Global Kill 功能时，KILL 语句和 KILL TIDB 语句均能跨节点终止查询或连接，且无需担心错误地终止其他查询或连接。当你使用客户端连接到任何一个 TiDB 节点执行 KILL 语句或 KILL TIDB 语句时，该语句会被转发给对应的 TiDB 节点。当客户端和 TiDB 中间有代理时，KILL 及 KILL TIDB 语句也会被转发给对应的 TiDB 节点执行。

对于 TiDB v6.1.0 之前的版本，或未启用 Global Kill 功能时：

- KILL 语句与 MySQL 不兼容，负载均衡器后面通常放有多个 TiDB 服务器，这种不兼容有助于防止在错误的 TiDB 服务器上终止连接。你需要显式地增加 TIDB 后缀，通过执行 KILL TIDB 语句来终止当前连接的 TiDB 实例上的其他连接。
- 强烈不建议在配置文件里设置compatible-kill-query = true，除非你确定客户端将始终连接到同一个 TiDB 节点。这是因为当你在默认的 MySQL 客户端按下 ctrl+c 时，客户端会开启一个新连接，并在这个新连接中执行 KILL 语句。此时，如果客户端和 TiDB 中间有代理，新连接可能会被路由到其他的 TiDB 节点，从而错误地终止其他会话。
- KILL TIDB 语句是 TiDB 的扩展语法，其功能与 MySQL 命令 KILL [CONNECTION|QUERY] 和 MySQL 命令行 ctrl+c 相同。在同一个 TiDB 节点上，你可以安全地使用 KILL TIDB 语句。

2.2.57.5 另请参阅

- SHOW [FULL] PROCESSLIST
- CLUSTER_PROCESSLIST

2.2.58 LOAD DATA

LOAD DATA 语句用于将数据批量加载到 TiDB 表中。

2.2.58.1 语法图

```
LoadDataStmt ::=
    'LOAD' 'DATA' LocalOpt 'INFILE' stringLit DuplicateOpt 'INTO' 'TABLE' TableName CharsetOpt
        ↪ Fields Lines IgnoreLines ColumnNameOrUserVarListOptWithBrackets LoadDataSetSpecOpt
```

2.2.58.2 参数说明

用户可以使用 LocalOpt 参数来指定导入的数据文件位于客户端或者服务端。目前 TiDB 只支持从客户端进行数据导入，因此在导入数据时 LocalOpt 应设置成 Local。

用户可以使用 Fields 和 Lines 参数来指定如何处理数据格式，使用 FIELDS TERMINATED BY 来指定每个数据的分隔符号，使用 FIELDS ENCLOSED BY 来指定消除数据的包围符号。如果用户希望以某个字符为结尾切分每行数据，可以使用 LINES TERMINATED BY 来指定行的终止符。

例如对于以下格式的数据：

```
"bob","20","street 1"\r\n
"alice","33","street 1"\r\n
```

如果想分别提取 bob、20、street 1，可以指定数据的分隔符号为 ','，数据的包围符号为 '\"'。可以写成：

```
FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
```

如果不指定处理数据的参数，将会按以下参数处理

```
FIELDS TERMINATED BY '\t' ENCLOSED BY ''
LINES TERMINATED BY '\n'
```

用户可以通过 IGNORE number LINES 参数来忽略文件开始的 number 行，例如可以使用 IGNORE 1 LINES 来忽略文件的首行。

另外，TiDB 目前对参数 DuplicateOpt、CharsetOpt、LoadDataSetSpecOpt 仅支持语法解析。

2.2.58.3 示例

```
CREATE TABLE trips (
    ->  trip_id bigint NOT NULL PRIMARY KEY AUTO_INCREMENT,
    ->  duration integer not null,
    ->  start_date datetime,
    ->  end_date datetime,
    ->  start_station_number integer,
    ->  start_station varchar(255),
    ->  end_station_number integer,
    ->  end_station varchar(255),
    ->  bike_number varchar(255),
    ->  member_type varchar(255)
    -> );
```

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

通过 LOAD DATA 导入数据，指定数据的分隔符为逗号，忽略包围数据的引号，并且忽略文件的第一行数据。

如果此时遇到 ERROR 1148 (42000): the used command is not allowed with this TiDB version 报错信息。可以参考以下文档解决：

ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
    ↪ IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station,
    ↪ end_station_number, end_station, bike_number, member_type);
```

```
Query OK, 815264 rows affected (39.63 sec)
Records: 815264  Deleted: 0  Skipped: 0  Warnings: 0
```

LOAD DATA 也支持使用十六进制 ASCII 字符表达式或二进制 ASCII 字符表达式作为 FIELDS ENCLOSED BY 和 FIELDS TERMINATED BY 的参数。示例如下：

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY x'2c' ENCLOSED BY b'100010' LINES TERMINATED BY '
    ↪ \r\n' IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station
    ↪ , end_station_number, end_station, bike_number, member_type);
```

以上示例中 x'2c' 是字符，的十六进制表示，b'100010' 是字符 " 的二进制表示。

2.2.58.4 MySQL 兼容性

LOAD DATA 语句应该完全兼容 MySQL。若发现任何兼容性差异，请在 GitHub 上提 issue。

> **注意：**
>
> 在 TiDB 的早期版本中，LOAD DATA 语句每 20000 行进行一次提交。新版本的 TiDB 默认在一个事务中提交所有行。从 TiDB 4.0 及以前版本升级后，可能出现 `ERROR 8004 (HY000)at line 1: Transaction is too large, size: 100000058` 错误。
>
> 要解决该问题，建议调大 `tidb.toml` 文件中的 `txn-total-size-limit` 值。如果无法增加此限制，还可以将`tidb_dml_batch_size` 的值设置为 20000 来恢复升级前的行为。

2.2.58.5 另请参阅

- INSERT
- 乐观事务模型
- TiDB Lightning

2.2.59 LOAD STATS

LOAD STATS 语句用于将统计信息加载到 TiDB 中。

2.2.59.1 语法图

```
LoadStatsStmt ::=
    'LOAD' 'STATS' stringLit
```

2.2.59.2 参数说明

用户直接指定统计信息文件路径，统计信息文件可通过访问 API `http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}` 进行下载。

路径可以是相对路径，也可以是绝对路径，如果是相对路径，会从启动 `tidb-server` 的路径为起点寻找对应文件。

下面是一个绝对路径的例子：

```
LOAD STATS '/tmp/stats.json';
```

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

2.2.59.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.59.4 另请参阅

- 统计信息

2.2.60 MODIFY COLUMN

`ALTER TABLE .. MODIFY COLUMN` 语句用于修改已有表上的列，包括列的数据类型和属性。若要同时重命名，可改用`CHANGE COLUMN` 语句。

从 v5.1.0 版本起，TiDB 开始支持 Reorg 类型变更，包括但不限于：

- 从 `VARCHAR` 转换为 `BIGINT`
- `DECIMAL` 精度修改
- 从 `VARCHAR(10)` 到 `VARCHAR(5)` 的长度压缩

2.2.60.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName ModifyColumnSpec ( ',' ModifyColumnSpec )*

ModifyColumnSpec
         ::= 'MODIFY' ColumnKeywordOpt 'IF EXISTS' ColumnName ColumnType ColumnOption* ( 'FIRST'
             ↪ | 'AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               ↪ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               ↪ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
```

```
            | 'STORAGE' StorageMedia
            | 'AUTO_RANDOM' ( '(' LengthNum ')' )?

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

2.2.60.2 示例

2.2.60.2.1 Meta-Only Change

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

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

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

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

```
ALTER TABLE t1 MODIFY col1 BIGINT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

2.2.60.2.2 Reorg-Data Change

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

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

```
INSERT INTO t1 (col1) VALUES (12345),(67890);
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 MODIFY col1 VARCHAR(5);
```

```
Query OK, 0 rows affected (2.52 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` varchar(5) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

注意：

- 当所变更的类型与已经存在的数据行产生冲突时，TiDB 会进行报错处理。在上述例子中，TiDB 将进行如下报错：

```
alter table t1 modify column col1 varchar(4); ERROR 1406 (22001): Data Too Long,
↪ field len 4, data len 5
```

- 由于和 Async Commit 功能兼容，DDL 在开始进入到 Reorg Data 前会有一定时间（约 2.5s）的等待处理：

```
Query OK, 0 rows affected (2.52 sec)
```

2.2.60.3 MySQL 兼容性

- 不支持使用单个 ALTER TABLE 语句修改多个列，例如：

```
ALTER TABLE t1 MODIFY col1 BIGINT, MODIFY id BIGINT NOT NULL;
ERROR 1105 (HY000): Unsupported multi schema change
```

- 不支持修改主键列上需要 Reorg-Data 的类型，但是支持修改 Meta-Only 的类型。例如：

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a VARCHAR(10);
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a INT(10) UNSIGNED;
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a bigint;
Query OK, 0 rows affected (0.01 sec)
```

- 不支持修改生成列的类型。例如：

```
CREATE TABLE t (a INT, b INT as (a+1));
ALTER TABLE t MODIFY COLUMN b VARCHAR(10);
ERROR 8200 (HY000): Unsupported modify column: column is generated
```

- 不支持修改分区表上的列类型。例如：

```
CREATE TABLE t (c1 INT, c2 INT, c3 INT) partition by range columns(c1) ( partition p0 values
    ↪  less than (10), partition p1 values less than (maxvalue));
ALTER TABLE t MODIFY COLUMN c1 DATETIME;
ERROR 8200 (HY000): Unsupported modify column: table is partition table
```

- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为 TiDB cast 函数与 MySQL 的行为有一些兼容性问题。例如：

```
CREATE TABLE t (a DECIMAL(13, 7));
ALTER TABLE t MODIFY COLUMN a DATETIME;
ERROR 8200 (HY000): Unsupported modify column: change from original type decimal(13,7) to
    ↪ datetime is currently unsupported yet
```

2.2.60.4 另请参阅

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

2.2.61 PREPARE

PREPARE 语句为服务器端预处理语句提供 SQL 接口。

2.2.61.1 语法图

```
PreparedStmt ::=
    'PREPARE' Identifier 'FROM' PrepareSQL

PrepareSQL ::=
    stringLit
|   UserVariable
```

2.2.61.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

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

```
SET @number = 5;
```

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

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

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

2.2.61.3 MySQL 兼容性

PREPARE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.61.4 另请参阅

- EXECUTE
- DEALLOCATE

2.2.62 RECOVER TABLE

RECOVER TABLE 的功能是恢复被删除的表及其数据。在 DROP TABLE 后，在 GC life time 时间内，可以用 RECOVER TABLE 语句恢复被删除的表以及其数据。

2.2.62.1 语法

```
RECOVER TABLE table_name
```

```
RECOVER TABLE BY JOB JOB_ID
```

2.2.62.1.1 语法图

```
RecoverTableStmt ::=
    'RECOVER' 'TABLE' ( 'BY' 'JOB' Int64Num | TableName Int64Num? )

TableName ::=
    Identifier ( '.' Identifier )?

Int64Num ::= NUM

NUM ::= intLit
```

2.2.62.2 注意事项

如果删除表后并过了 GC lifetime，就不能再用 RECOVER TABLE 来恢复被删除的表了，执行 RECOVER TABLE 语句会返回类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。

对于 3.0.0 及之后的 TiDB 版本，不推荐在使用 TiDB Binlog 的情况下使用 RECOVER TABLE 功能。

TiDB Binlog 在 3.0.1 支持 RECOVER TABLE 后，可在下面的情况下使用 RECOVER TABLE：

- 3.0.1+ 版本的 TiDB Binlog
- 主从集群都使用 TiDB 3.0
- 从集群 GC lifetime 一定要长于主集群（不过由于上下游同步的延迟，可能也会造成下游 recover 失败）

2.2.62.2.1 TiDB Binlog 同步错误处理

当使用 TiDB Binlog 同步工具时，上游 TiDB 使用 RECOVER TABLE 后，TiDB Binlog 可能会因为下面几个原因造成同步中断：

- 下游数据库不支持 RECOVER TABLE 语句。类似错误：`check the manual that corresponds to your MySQL server version for the right syntax to use near 'RECOVER TABLE table_name'`。
- 上下游数据库的 GC lifetime 不一样。类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。
- 上下游数据库的同步延迟。类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。

只能通过重新全量导入被删除的表来恢复 TiDB Binlog 的数据同步。

2.2.62.3 示例

- 根据表名恢复被删除的表。

```
DROP TABLE t;
```

```
RECOVER TABLE t;
```

根据表名恢复被删除的表需满足以下条件：

  - 最近 DDL JOB 历史中找到的第一个 DROP TABLE 操作，且
  - DROP TABLE 所删除的表的名称与 RECOVER TABLE 语句指定表名相同

- 根据删除表时的 DDL JOB ID 恢复被删除的表。

  如果第一次删除表 t 后，又新建了一个表 t，然后又把新建的表 t 删除了，此时如果想恢复最开始删除的表 t, 就需要用到指定 DDL JOB ID 的语法了。

```
DROP TABLE t;
```

```
ADMIN SHOW DDL JOBS 1;
```

上面这个语句用来查找删除表 t 时的 DDL JOB ID，这里是 53：

```
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE   | SCHEMA_STATE | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                         | STATE  |
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
| 53     | test    |            | drop table | none         | 1         | 41       | 0
    ↪         | 2019-07-10 13:23:18.277 +0800 CST | synced |
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
```

```
RECOVER TABLE BY JOB 53;
```

根据删除表时的 DDL JOB ID 恢复被删除的表，会直接用 DDL JOB ID 找到被删除表进行恢复。如果指定的 DDL JOB ID 的 DDL JOB 不是 DROP TABLE 类型，会报错。

2.2.62.4 原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 mysql.gc_delete_range 表。TiDB 后台的 GC Worker 会定期从 mysql.gc_delete_range 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，RECOVER TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 mysql.gc_delete_range 表中相应的行记录就可以了。恢复表的元信息可以用 TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。

TiDB 中表的恢复是通过快照读获取表的元信息后，再走一次类似于 CREATE TABLE 的建表流程，所以 RECOVER
↪ TABLE 实际上也是一种 DDL。

2.2.62.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.63 RENAME INDEX

`ALTER TABLE .. RENAME INDEX` 语句用于对已有索引进行重命名。这在 TiDB 中是即时操作的，仅需更改元数据。

2.2.63.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName RenameIndexSpec ( ',' RenameIndexSpec )*

RenameIndexSpec
         ::= 'RENAME' ( 'KEY' | 'INDEX' ) Identifier 'TO' Identifier
```

2.2.63.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL, INDEX col1 (c1));
```

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

```
SHOW CREATE TABLE t1;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `col1` (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 RENAME INDEX col1 TO c1;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
```

```
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `c1` (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

2.2.63.3 MySQL 兼容性

`RENAME INDEX` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.63.4 另请参阅

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

2.2.64 RENAME TABLE

`RENAME TABLE` 语句用于对已有表进行重命名。

2.2.64.1 语法图

```
RenameTableStmt ::=
    'RENAME' 'TABLE' TableToTable ( ',' TableToTable )*

TableToTable ::=
    TableName 'TO' TableName
```

2.2.64.2 示例

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

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW TABLES;
```

```
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

```
RENAME TABLE t1 TO t2;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SHOW TABLES;
```

```
+----------------+
| Tables_in_test |
+----------------+
| t2             |
+----------------+
1 row in set (0.00 sec)
```

2.2.64.3 MySQL 兼容性

RENAME TABLE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.64.4 另请参阅

- CREATE TABLE
- SHOW TABLES
- ALTER TABLE

2.2.65 REPLACE

从语义上看，REPLACE 语句是 DELETE 语句和 INSERT 语句的结合，可用于简化应用程序代码。

2.2.65.1 语法图

```
ReplaceIntoStmt ::=
    'REPLACE' PriorityOpt IntoOpt TableName PartitionNameListOpt InsertValues

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IntoOpt ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?
```

```
InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        ↪ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*
```

2.2.65.2 示例

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

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
+----+----+
3 rows in set (0.00 sec)
```

```
REPLACE INTO t1 (id, c1) VALUES(3, 99);
```

```
Query OK, 2 rows affected (0.01 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 | 99 |
+----+----+
3 rows in set (0.00 sec)
```

2.2.65.3 MySQL 兼容性

REPLACE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.65.4 另请参阅

- DELETE
- INSERT
- SELECT
- UPDATE

2.2.66 RESTORE

RESTORE 语句用于执行分布式恢复，把 BACKUP 语句生成的备份文件恢复到 TiDB 集群中。

RESTORE 语句使用的引擎与 BR 相同，但恢复过程是由 TiDB 本身驱动，而非单独的 BR 工具。BR 工具的优势和警告也适用于 RESTORE 语句。需要注意的是，RESTORE 语句目前不遵循 ACID 原则。

执行 RESTORE 语句前，确保集群已满足以下要求：

- 集群处于“下线”状态，当前的 TiDB 会话是唯一在访问待恢复表的活跃 SQL 连接。
- 执行全量恢复时，确保即将恢复的表不存在于集群中，因为现有的数据可能被覆盖，从而导致数据与索引不一致。
- 执行增量恢复时，表的状态应该与创建备份时 LAST_BACKUP 时间戳的状态完全一致。

执行 RESTORE 需要 RESTORE_ADMIN 或 SUPER 权限。此外，执行恢复操作的 TiDB 节点和集群中的所有 TiKV 节点都必须有对目标存储的读权限。

RESTORE 语句开始执行后将会被阻塞，直到整个恢复任务完成、失败或取消。因此，执行 RESTORE 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和 RESTORE 任务。如果 TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 RESTORE 将等待前面所有的任务完成后再执行。

RESTORE 只能在“tikv”存储引擎上使用，如果使用“unistore”存储引擎，RESTORE 操作会失败。

2.2.66.1 语法图

```
RestoreStmt ::=
    "RESTORE" BRIETables "FROM" stringLit RestoreOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

RestoreOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
```

```
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean

Boolean ::=
    NUM | "TRUE" | "FALSE"
```

2.2.66.2 示例

2.2.66.2.1 从备份文件恢复

```
RESTORE DATABASE * FROM 'local:///mnt/backup/2020/04/';
```

```
+--
    ↪ ---------------------------+-----------+----------+--------------------+-------------------+
    ↪
| Destination                  | Size      | BackupTS | Queue Time         | Execution Time
    ↪      |
+--
    ↪ ---------------------------+-----------+----------+--------------------+-------------------+
    ↪
| local:///mnt/backup/2020/04/ | 248665063 | 0        | 2020-04-21 17:16:55 | 2020-04-21 17:16:55
    ↪  |
+--
    ↪ ---------------------------+-----------+----------+--------------------+-------------------+
    ↪
1 row in set (28.961 sec)
```

上述示例中，所有数据均从本地的备份文件中恢复到集群中。RESTORE 从 SST 文件里读取数据，SST 文件存储在所有 TiDB 和 TiKV 节点的 /mnt/backup/2020/04/ 目录下。

输出结果的第一行描述如下：

列名	描述
Destination	读取的目标存储 URL
Size	备份文件的总大小，单位为字节
BackupTS	不适用
Queue Time	RESTORE 任务开始排队的时间戳（当前时区）
Execution Time	RESTORE 任务开始执行的时间戳（当前时区）

2.2.66.2.2 部分恢复

你可以指定恢复部分数据库或部分表数据。如果备份文件中缺失了某些数据库或表，缺失的部分将被忽略。此时，RESTORE 语句不进行任何操作即完成执行。

```
RESTORE DATABASE `test` FROM 'local:///mnt/backup/2020/04/';
```

```
RESTORE TABLE `test`.`sbtest01`, `test`.`sbtest02` FROM 'local:///mnt/backup/2020/04/';
```

2.2.66.2.3 外部存储

BR 支持从 Amazon S3 或 Google Cloud Storage (GCS) 恢复数据：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/?region=us-west-2';
```

有关详细的 URL 语法，见外部存储。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/?region=us-west-2'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

2.2.66.2.4 性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个 TiKV 节点的平均下载速度。

默认情况下，每个 TiKV 节点上运行 128 个恢复线程。可以通过 CONCURRENCY 选项来调整这个值。

在恢复完成之前，RESTORE 将对备份文件中的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 64
    CHECKSUM = FALSE;
```

2.2.66.2.5 增量恢复

增量恢复没有特殊的语法。TiDB 将识别备份文件属于全量备份或增量备份，然后执行对应的恢复操作，用户只需按照正确顺序进行增量恢复。

假设按照如下方式创建一个备份任务：

```
BACKUP DATABASE `test` TO 's3://example-bucket/full-backup'  SNAPSHOT = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-1' SNAPSHOT = 414971854848000
    ↪ LAST_BACKUP = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-2' SNAPSHOT = 416353458585600
    ↪ LAST_BACKUP = 414971854848000;
```

在恢复备份时，需要采取同样的顺序：

```
RESTORE DATABASE * FROM 's3://example-bucket/full-backup';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-1';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-2';
```

2.2.66.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.66.4 另请参阅

- BACKUP
- SHOW RESTORES

2.2.67 REVOKE <privileges>

REVOKE <privileges> 语句用于删除已有用户的权限。执行 REVOKE <privileges> 语句需要拥有分配的权限，并且拥有 GRANT OPTION 权限。

2.2.67.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        ↪ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
    'ALL' 'PRIVILEGES'?
|   'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|    'TRIGGER'
|   'DELETE'
|    'DROP' 'ROLE'?
|    'PROCESS'
|    'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|    'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|   'GRANT' 'OPTION'
|   'REFERENCES'
|   'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|   'USAGE'
```

```
|    'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|    'EVENT'
|   'SHUTDOWN'

ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

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

2.2.67.2 示例

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

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

```
GRANT ALL ON test.* TO 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

```
REVOKE ALL ON test.* FROM 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------+
| Grants for newuser@%                |
+-------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+-------------------------------------+
1 row in set (0.00 sec)
```

```
DROP USER 'newuser';
```

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

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

2.2.67.3 MySQL 兼容性

`REVOKE <privileges>` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.67.4 另请参阅

- GRANT <privileges>
- SHOW GRANTS
- Privilege Management

2.2.68 REVOKE <role>

`REVOKE <role>` 语句用于从指定的用户（或用户列表）中收回之前授予的角色。

2.2.68.1 语法图

```
RevokeRoleStmt ::=
    'REVOKE' RolenameList 'FROM' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

2.2.68.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)
```

```
SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与某一角色相关联，这样该用户无需执行 SET ROLE 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

收回角色 analyticsteam：

```
$ mysql -uroot

REVOKE analyticsteam FROM jennifer;
Query OK, 0 rows affected (0.01 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
```

```
+------------------------------------+
| Grants for User                    |
+------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+------------------------------------+
1 row in set (0.00 sec)
```

2.2.68.3 MySQL 兼容性

`REVOKE <role>` 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.68.4 另请参阅

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

2.2.69 ROLLBACK

`ROLLBACK` 语句用于还原 TiDB 内当前事务中的所有更改，作用与 `COMMIT` 语句相反。

2.2.69.1 语法图

```
RollbackStmt ::=
    'ROLLBACK' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

2.2.69.2 示例

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
BEGIN;
```

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

```
INSERT INTO t1 VALUES (1);
```

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

```
ROLLBACK;
```

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

```
SELECT * FROM t1;
```

```
Empty set (0.01 sec)
```

2.2.69.3 MySQL 兼容性

- TiDB 不支持 savepoint 或 `ROLLBACK TO SAVEPOINT` 语法。
- TiDB 解析但忽略 `ROLLBACK AND [NO] RELEASE` 语法。在 MySQL 中，使用该语法可在回滚事务后立即断开客户端会话。在 TiDB 中，建议使用客户端程序的 `mysql_close()` 来实现该功能。
- TiDB 解析但忽略 `ROLLBACK AND [NO] CHAIN` 语法。在 MySQL 中，使用该语法可在回滚当前事务时立即以相同的隔离级别开启新事务。在 TiDB 中，推荐直接开启新事务。

2.2.69.4 另请参阅

- COMMIT
- BEGIN
- START TRANSACTION

2.2.70 SELECT

`SELECT` 语句用于从 TiDB 读取数据。

2.2.70.1 语法图

SelectStmt:

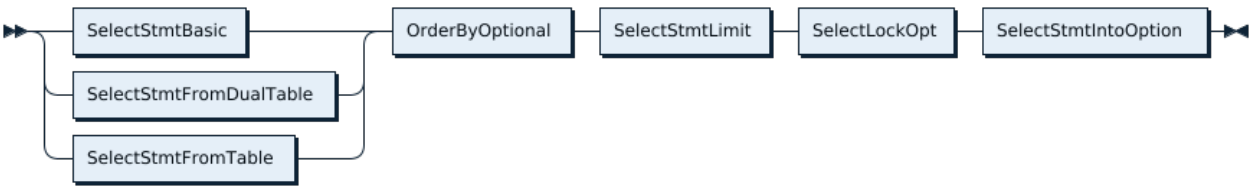


图 2: SelectStmt

FromDual:

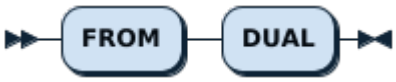


图 3: FromDual

WhereClauseOptional:

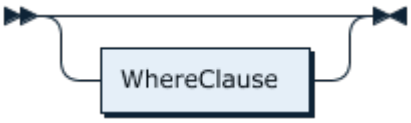


图 4: WhereClauseOptional

SelectStmtOpts:

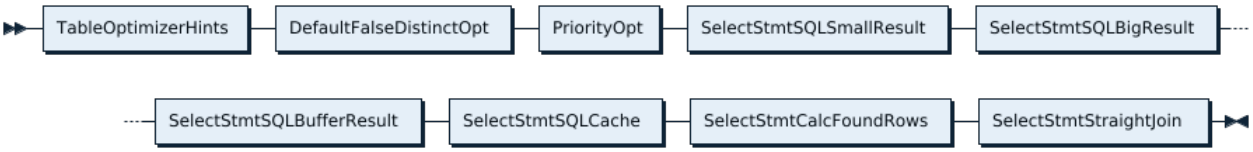


图 5: SelectStmtOpts

SelectStmtFieldList:

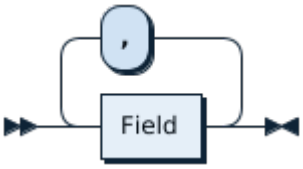


图 6: SelectStmtFieldList

TableRefsClause:

```
TableRefsClause ::=
    TableRef AsOfClause? ( ',' TableRef AsOfClause? )*

AsOfClause ::=
    'AS' 'OF' 'TIMESTAMP' Expression
```

WhereClauseOptional:

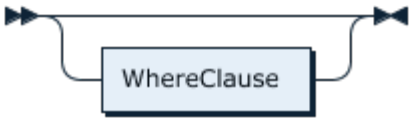


图 7: WhereClauseOptional

SelectStmtGroup:



图 8: SelectStmtGroup

HavingClause:

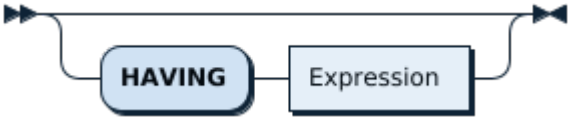


图 9: HavingClause

OrderByOptional:



图 10: OrderByOptional

SelectStmtLimit:

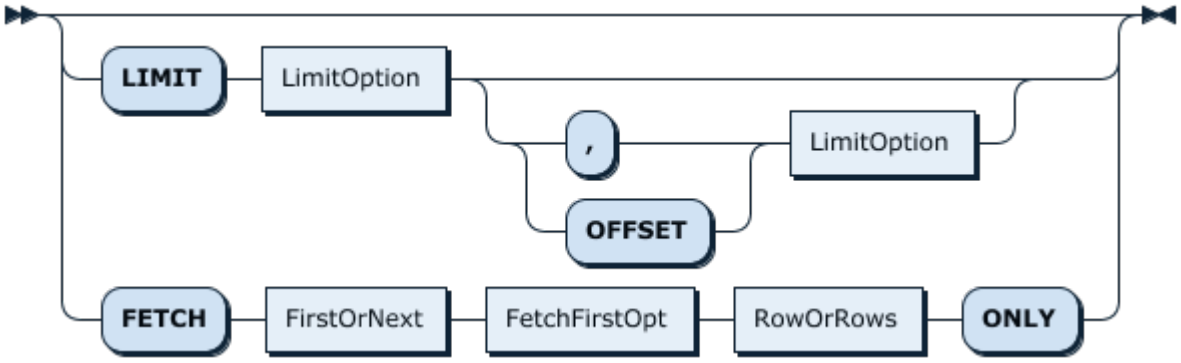


图 11: SelectStmtLimit

FirstOrNext:

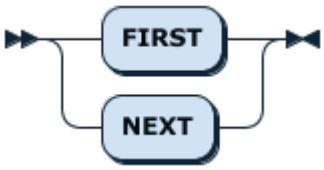


图 12: FirstOrNext

FetchFirstOpt:



图 13: FetchFirstOpt

RowOrRows:

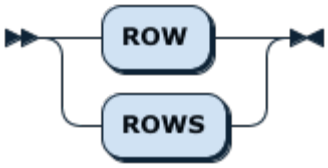


图 14: RowOrRows

SelectLockOpt:

```
SelectLockOpt ::=
    ( ( 'FOR' 'UPDATE' ( 'OF' TableList )? 'NOWAIT'? )
|   ( 'LOCK' 'IN' 'SHARE' 'MODE' ) )?

TableList ::=
    TableName ( ',' TableName )*
```

WindowClauseOptional

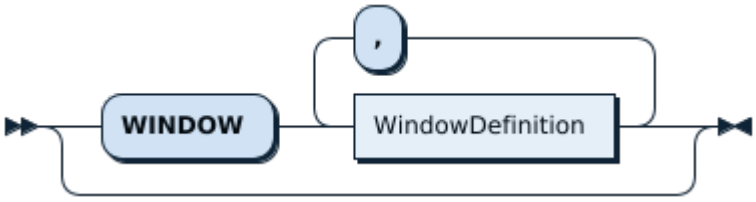


图 15: WindowClauseOptional

2.2.70.2 语法元素说明

语法元素	说明
TableOptimizerHints	用于控制优化器行为的 Hint，具体可参见Optimizer Hints
ALL、DISTINCT、DISTINCTROW	查询结果集中可能会包含重复值。指定 DISTINCT/DISTINCTROW 则在查询结果中过滤掉重复的行；指定 ALL 则列出所有的行。默认为 ALL。
HIGH_PRIORITY	该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
SQL_CALC_FOUND_ROWS	TiDB 不支持该语法，并报错（若tidb_enable_noop_functions 值设为 1 则不会报错）
SQL_CACHE、 SQL_NO_CACHE	是否把请求结果缓存到 TiKV (RocksDB) 的 BlockCache 中。对于一次性的大数据量的查询，比如 count(*) 查询，为了避免冲掉 BlockCache 中用户的热点数据，建议填上 SQL_NO_CACHE
STRAIGHT_JOIN	STRAIGHT_JOIN 会强制优化器按照 FROM 子句中所使用的表的顺序做联合查询。当优化器选择的 Join 顺序并不优秀时，你可以使用这个语法来加速查询的执行
select_expr	投影操作列表，一般包括列名、表达式，或者是用 ‘*’ 表示全部列
FROM table_references	表示数据来源，数据来源可以是一个表（select * from t;）或者是多个表(select * from t1 join t2;)或者是 0 个表(select 1+1 from dual;,等价于 select 1+1;)
WHERE where_condition	Where 子句用于设置过滤条件，查询结果中只会包含满足条件的数据
GROUP BY	GroupBy 子句用于对查询结果集进行分组
HAVING where_condition	Having 子句与 Where 子句作用类似，Having 子句可以让过滤 GroupBy 后的各种数据，Where 子句用于在聚合前过滤记录。
ORDER BY	OrderBy 子句用于指定结果排序顺序，可以按照列、表达式或者是 select_expr 列表中某个位置的字段进行排序。
LIMIT	Limit 子句用于限制结果条数。Limit 接受一个或两个数字参数，如果只有一个参数，那么表示返回数据的最大行数；如果是两个参数，那么第一个参数表示返回数据的第一行的偏移量（第一行数据的偏移量是 0），第二个参数指定返回数据的最大条目数。另支持 FETCH FIRST/NEXT n ROW/ROWS ONLY 语法，与 LIMIT n 效果相同，其中 n 可省略，省略时与 LIMIT 1 效果相同。
Window window_definition	窗口函数的相关语法，用来进行一些分析型计算的操作，详情可见窗口函数

语法元素	说明
FOR UPDATE	对查询结果集所有行上锁（对于在查询条件内，但是不在结果集的行，将不会加锁，如事务启动后由其他事务写入的行），以监测其他事务对这些的并发修改。使用乐观事务模型时，语句执行期间不会检测锁，因此，不会像 PostgreSQL 之类的数据库一样，在当前事务结束前阻止其他事务执行 UPDATE、DELETE 和 SELECT FOR UPDATE。在事务的提交阶段 SELECT FOR UPDATE 读到的行，也会进行两阶段提交，因此，它们也可以参与事务冲突检测。如发生写入冲突，那么包含 SELECT FOR UPDATE 语句的事务会提交失败。如果没有冲突，事务将成功提交，当提交结束时，这些被加锁的行，会产生一个新版本，可以让其他尚未提交的事务，在将来提交时发现写入冲突。若使用悲观事务，则行为与其他数据库基本相同，不一致之处参考和 MySQL InnoDB 的差异。TiDB 支持 FOR UPDATE NOWAIT 语法，详情可见TiDB 中悲观事务模式的行为。
LOCK IN SHARE MODE	TiDB 出于兼容性解析这个语法，但是不做任何处理

2.2.70.3 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

2.2.70.4 MySQL 兼容性

- 不支持 SELECT ... INTO @variable 语法。
- 不支持 SELECT ... GROUP BY ... WITH ROLLUP 语法。
- 不支持 MySQL 5.7 中支持的 SELECT .. GROUP BY expr 语法，而是匹配 MySQL 8.0 的行为，不按照默认的顺序进行排序。

2.2.70.5 另请参阅

- INSERT
- DELETE
- UPDATE
- REPLACE

2.2.71 SET DEFAULT ROLE

SET DEFAULT ROLE 语句默认设置将特定角色应用于用户。因此，用户不必执行 SET ROLE <rolename> 或 SET ROLE ALL 语句，也可以自动具有与角色相关联的权限。

2.2.71.1 语法图

SetDefaultRoleStmt:

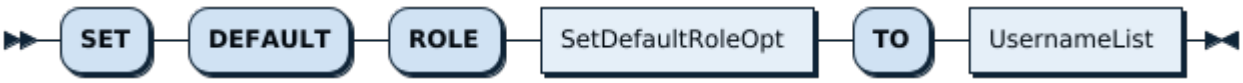


图 16: SetDefaultRoleStmt

SetDefaultRoleOpt:

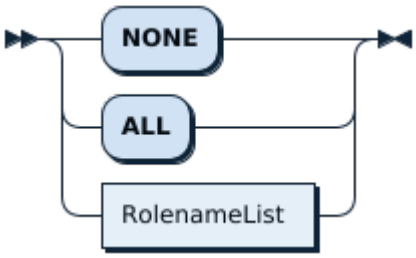


图 17: SetDefaultRoleOpt

RolenameList:

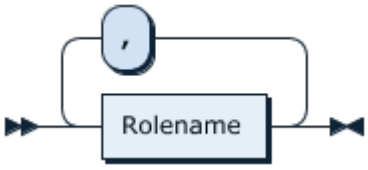


图 18: RolenameList

UsernameList:

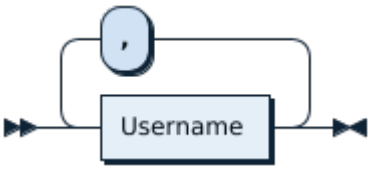


图 19: UsernameList

2.2.71.2 示例

创建新角色 analyticsteam 和新用户 jennifer：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)
CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)
GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
```

```
| GRANT USAGE ON *.* TO 'jennifer'@'%'         |
| GRANT Select ON test.* TO 'jennifer'@'%'     |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+----------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+----------------------------------------------+
| Grants for User                              |
+----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'         |
| GRANT Select ON test.* TO 'jennifer'@'%'     |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+----------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

`SET DEFAULT ROLE` 语句不会自动将相关角色授予 (GRANT) 用户。若尝试为 `jennifer` 尚未被授予的角色执行 `SET DEFAULT ROLE` 语句会导致以下错误：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
```

```
ERROR 3530 (HY000): `analyticsteam`@`%` is is not granted to jennifer@%
```

2.2.71.3 MySQL 兼容性

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

2.2.71.4 另请参阅

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

2.2.72 SET [NAMES|CHARACTER SET]

SET NAMES，SET CHARACTER SET 和 SET CHARSET 语句用于修改当前连接的变量 character_set_client，character_set_results 和 character_set_connection。

2.2.72.1 语法图

SetNamesStmt:



图 20: SetNamesStmt

VariableAssignmentList:

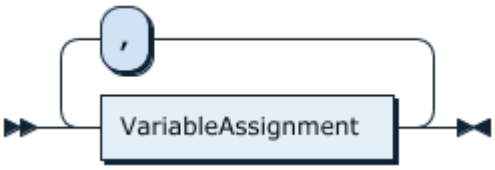


图 21: VariableAssignmentList

VariableAssignment:

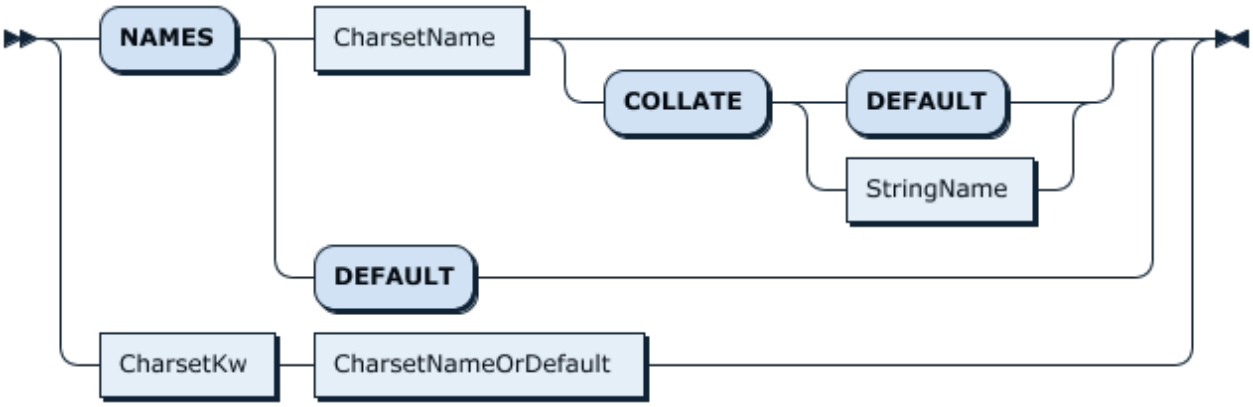


图 22: VariableAssignment

CharsetName:



图 23: CharsetName

StringName:



图 24: StringName

CharsetKw:

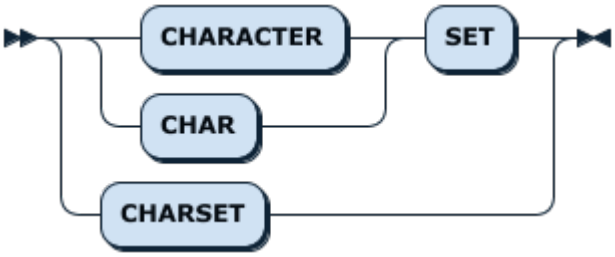


图 25: CharsetKw

CharsetNameOrDefault:

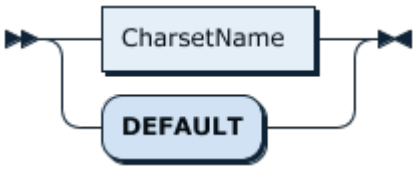


图 26: CharsetNameOrDefault

2.2.72.2 示例

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_connection | utf8mb4                                                  |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8mb4                                                  |
| character_set_client     | utf8mb4                                                  |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
| character_set_server     | utf8mb4                                                  |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.01 sec)
```

```
SET NAMES utf8;
```

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

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_connection | utf8                                                     |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8                                                     |
| character_set_client     | utf8                                                     |
| character_set_server     | utf8mb4                                                  |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.00 sec)
```

```
SET CHARACTER SET utf8mb4;
```

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

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_set_connection | utf8mb4                                                  |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8mb4                                                  |
| character_set_client     | utf8mb4                                                  |
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
| character_set_server     | utf8mb4                                                  |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.00 sec)
```

2.2.72.3 MySQL 兼容性

`SET [NAMES|CHARACTER SET]` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.72.4 另请参阅

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

2.2.73 SET PASSWORD

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

2.2.73.1 语法图

SetStmt:

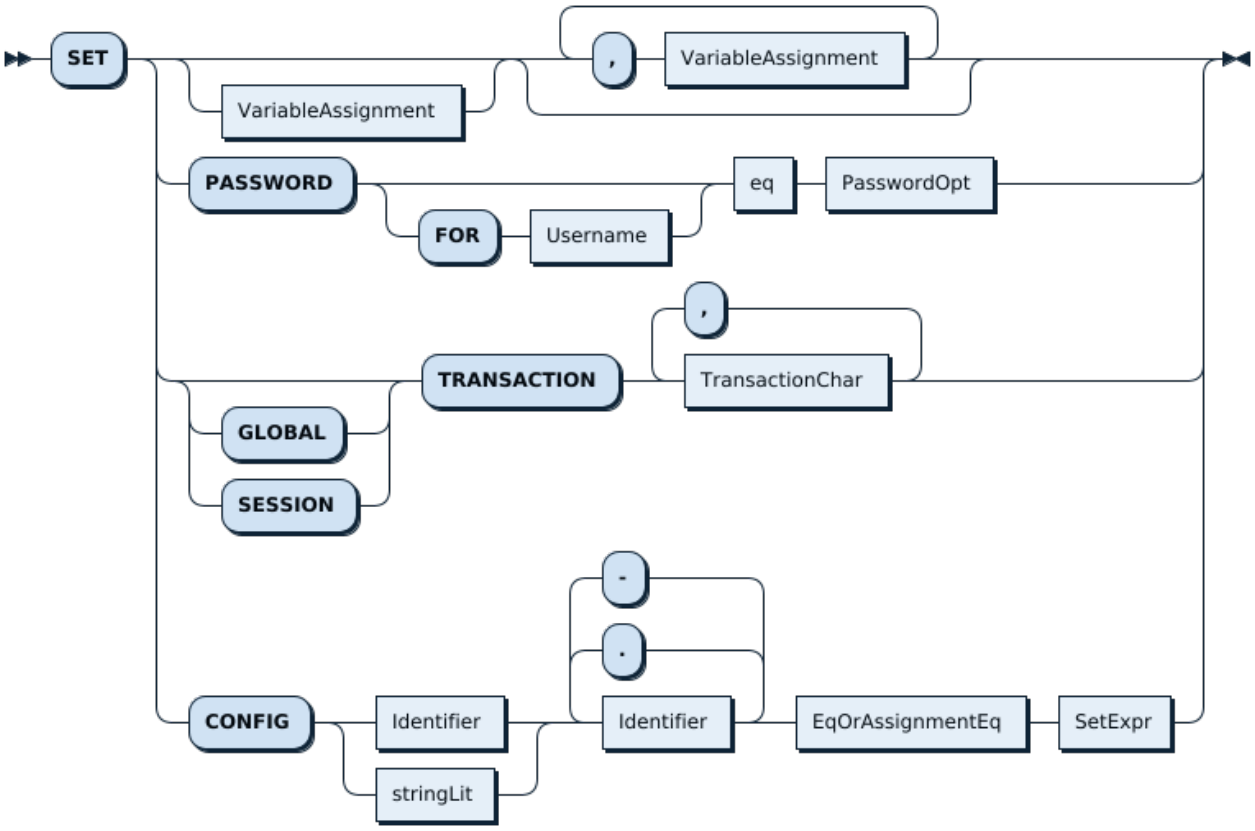


图 27: SetStmt

2.2.73.2 示例

```
SET PASSWORD='test';
```

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

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

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

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
```

```
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SET PASSWORD FOR 'newuser' = 'test';
```

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

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SET PASSWORD FOR 'newuser' = PASSWORD('test');
```

上述语法是早期 MySQL 版本的过时语法。

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

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
```

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

2.2.73.3 MySQL 兼容性

`SET PASSWORD` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.73.4 另请参阅

- CREATE USER
- Privilege Management

2.2.74 SET ROLE

`SET ROLE` 用于在当前用户会话中启用角色。使用 `SET ROLE` 启用角色后，用户可以使用这些角色的权限。

2.2.74.1 语法图

SetRoleStmt:



图 28: SetRoleStmt

SetRoleOpt:

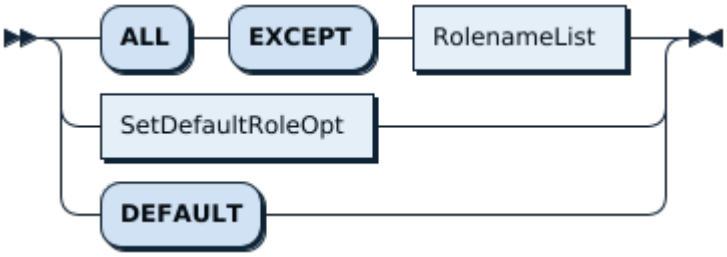


图 29: SetRoleOpt

SetDefaultRoleOpt:

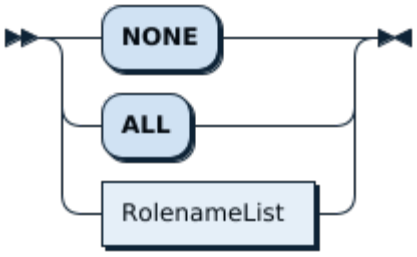


图 30: SetDefaultRoleOpt

2.2.74.2 示例

创建一个用户 `'u1'@'%'`, 创建三个角色 `'r1'@'%'`, `'r2'@'%'`, `'r3'@'%'` 并将这些角色授予给 `'u1'@'%'`。将 `'u1'@'%'` 的默认启用角色设置为 `'r1'@'%'`。

```
CREATE USER 'u1'@'%';
CREATE ROLE 'r1', 'r2', 'r3';
GRANT 'r1', 'r2', 'r3' TO 'u1'@'%';
SET DEFAULT ROLE 'r1' TO 'u1'@'%';
```

使用 `'u1'@'%'` 登录，执行 `SET ROLE` 将启用角色设置为 `ALL`。

```
SET ROLE ALL;
SELECT CURRENT_ROLE();
```

```
+----------------------------+
| CURRENT_ROLE()             |
+----------------------------+
| `r1`@`%`,`r2`@`%`,`r3`@`%` |
+----------------------------+
1 row in set (0.000 sec)
```

执行 `SET ROLE` 将启用角色设置为 `'r2'` 和 `'r3'`。

```
SET ROLE 'r2', 'r3';
SELECT CURRENT_ROLE();
```

```
+-------------------+
| CURRENT_ROLE()    |
+-------------------+
| `r2`@`%`,`r3`@`%` |
+-------------------+
1 row in set (0.000 sec)
```

执行 `SET ROLE` 将启用角色设置为 `DEFAULT`。

```
SET ROLE DEFAULT;
SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
| `r1`@`%`       |
+----------------+
1 row in set (0.000 sec)
```

执行 SET ROLE 将启用角色设置为 NONE。

```
SET ROLE NONE;
SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
|                |
+----------------+
1 row in set (0.000 sec)
```

2.2.74.3 MySQL 兼容性

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

2.2.74.4 另请参阅

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

2.2.75 SET TRANSACTION

SET TRANSACTION 语句用于在 GLOBAL 或 SESSION 的基础上更改当前的隔离级别，是 SET transaction_isolation ↪ ='new-value' 的替代语句，提供 MySQL 和 SQL 标准的兼容性。

2.2.75.1 语法图

```
SetStmt ::=
    'SET' ( VariableAssignmentList |
    'PASSWORD' ('FOR' Username)? '=' PasswordOpt |
    ( 'GLOBAL'| 'SESSION' )? 'TRANSACTION' TransactionChars |
```

```
    'CONFIG' ( Identifier | stringLit) ConfigItemName EqOrAssignmentEq SetExpr )

TransactionChars ::=
    ( 'ISOLATION' 'LEVEL' IsolationLevel | 'READ' 'WRITE' | 'READ' 'ONLY' AsOfClause? )

IsolationLevel ::=
    ( 'REPEATABLE' 'READ' | 'READ' ( 'COMMITTED' | 'UNCOMMITTED' ) | 'SERIALIZABLE' )

AsOfClause ::=
    ( 'AS' 'OF' 'TIMESTAMP' Expression)
```

2.2.75.2 示例

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.00 sec)
```

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

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

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
+-----------------------+----------------+
1 row in set (0.01 sec)
```

```
SET SESSION transaction_isolation = 'REPEATABLE-READ';
```

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

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
```

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

2.2.75.3 MySQL 兼容性

- TiDB 支持仅在语法中将事务设置为只读的功能。
- 不支持隔离级别 READ-UNCOMMITTED 和 SERIALIZABLE。
- 通过快照隔离 (Snapshot Isolation) 技术，实现乐观事务的 REPEATABLE-READ 隔离级别，和 MySQL 兼容。
- 在悲观事务中，TiDB 支持与 MySQL 兼容的 REPEATABLE-READ 和 READ-COMMITTED 两种隔离级别。具体描述详见Isolation Levels。

2.2.75.4 另请参阅

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

2.2.76 SET [GLOBAL|SESSION] <variable>

SET [GLOBAL|SESSION] 语句用于在 SESSION 或 GLOBAL 的范围内，对某个 TiDB 的内置变量进行更改。

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

2.2.76.1 语法图

SetStmt:

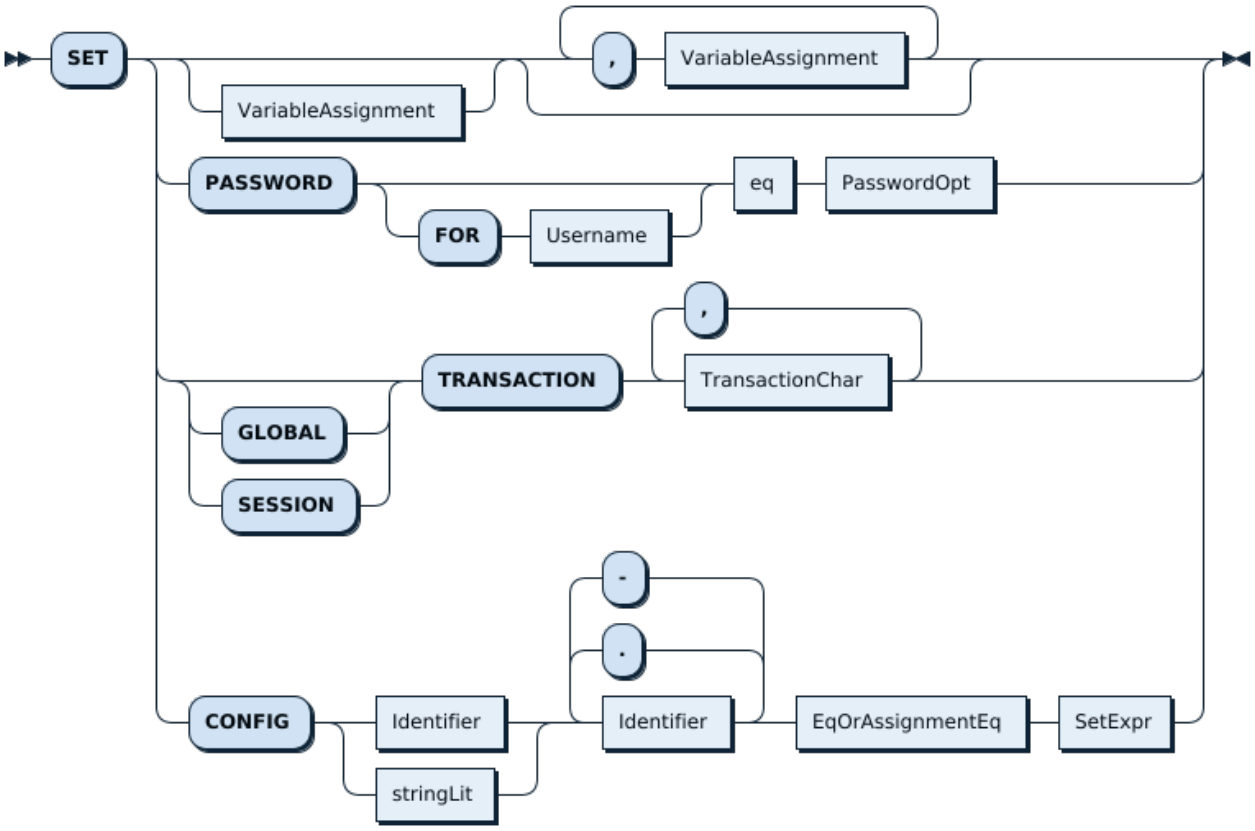


图 31: SetStmt

VariableAssignment:

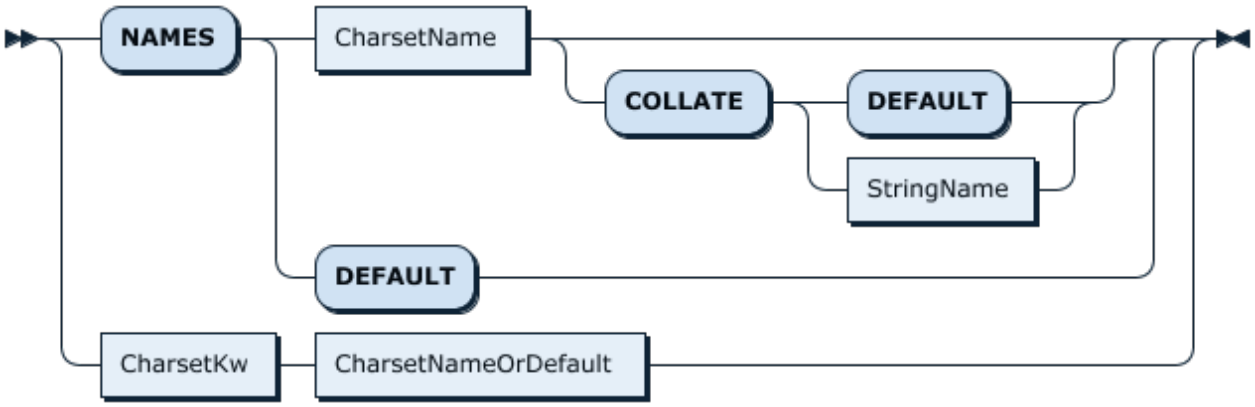


图 32: VariableAssignment

2.2.76.2 示例

获取 `sql_mode` 的值：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

更新全局的 `sql_mode`：

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

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

检查更新之后的 `sql_mode` 的取值，可以看到 SESSION 级别的值没有更新：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+-----------------------------------------+
1 row in set (0.00 sec)
```

```sql
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

SET SESSION 则可以立即生效：

```sql
SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER';
```

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

```sql
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+-----------------------------------------+
1 row in set (0.00 sec)
```

2.2.76.3 MySQL 兼容性

使用 SET [GLOBAL|SESSION] <variable> 更改系统变量上，TiDB 与 MySQL 存在以下差异

- 与 MySQL 不同，TiDB 中使用 SET GLOBAL 所作的修改会应用于集群中的全部 TiDB 实例。而在 MySQL 中，修改不会应用于副本。
- TiDB 中的若干变量可读又可设置，这是与 MySQL 相兼容的要求，因为应用程序和连接器常读取 MySQL 变量。例如：JDBC 连接器同时读取和设置缓存查询的参数，尽管并不依赖这一行为。
- 即使在 TiDB 服务器重启后，SET GLOBAL 的更改也仍然有效。这样，TiDB 中的 SET GLOBAL 更类似于 MySQL 8.0 及更高版本中的 SET PERSIST。

2.2.76.4 另请参阅

- SHOW [GLOBAL|SESSION] VARIABLES

2.2.77 SHOW [BACKUPS|RESTORES]

SHOW [BACKUPS|RESTORES] 语句会列出所有队列中或正在执行的 BACKUP 和RESTORE 任务。

查询 BACKUP 任务时，使用 SHOW BACKUPS 语句。查询 RESTORE 任务时，使用 SHOW RESTORES 语句。执行两个语句均需要 SUPER 权限。

2.2.77.1 语法图

```
ShowBRIEStmt ::=
    "SHOW" ("BACKUPS" | "RESTORES") ShowLikeOrWhere?

ShowLikeOrWhere ::=
    "LIKE" SimpleExpr
|   "WHERE" Expression
```

2.2.77.2 示例

在一个连接中，执行以下命令备份数据库：

```
BACKUP DATABASE `test` TO 's3://example-bucket/backup-01/?region=us-west-1';
```

在备份完成之前，在新的连接中执行 SHOW BACKUPS：

```
SHOW BACKUPS;
```

```
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
| Destination                    | State    | Progress | Queue_Time          | Execution_Time
    ↪       | Finish_Time | Connection |
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
| s3://example-bucket/backup-01/ | Backup   | 98.38    | 2020-04-12 23:09:03 | 2020-04-12 23:09:25
    ↪  |         NULL |          4 |
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
1 row in set (0.00 sec)
```

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL（为避免泄露密钥，所有参数均不显示）
State	任务状态

列名	描述
Progress	当前状态的进度（百分比）
Queue Time	任务开始排队的时间
Execution Time	任务开始执行的时间；对于队列中任务，该值为 0000-00-00 00:00:00
Finish_Time	（暂不适用）
Connection	运行任务的连接 ID

连接 ID 可用于在KILL TIDB QUERY 语句中取消备份/恢复任务：

```
KILL TIDB QUERY 4;
```

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

2.2.77.2.1 过滤

在 LIKE 子句中使用通配符，可以按目标存储 URL 筛选任务：

```
SHOW BACKUPS LIKE 's3://%';
```

使用 WHERE 子句，可以按列筛选任务：

```
SHOW BACKUPS WHERE `Progress` < 25.0;
```

2.2.77.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.77.4 另请参阅

- BACKUP
- RESTORE

2.2.78 SHOW ANALYZE STATUS

SHOW ANALYZE STATUS 语句提供 TiDB 正在执行的统计信息收集任务以及有限条历史任务记录。

从 TiDB v6.1.0 起，执行 SHOW ANALYZE STATUS 语句将显示集群级别的任务，且 TiDB 重启后仍能看到重启之前的任务记录。在 TiDB v6.1.0 之前，执行 SHOW ANALYZE STATUS 语句仅显示实例级别的任务，且 TiDB 重启后任务记录会被清空。

从 TiDB v6.1.0 起，你可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史记录。

2.2.78.1 语法图

```
ShowAnalyzeStatusStmt ::= 'SHOW' 'ANALYZE' 'STATUS' ShowLikeOrWhereOpt

ShowLikeOrWhereOpt ::= 'LIKE' SimpleExpr | 'WHERE' Expression
```

2.2.78.2 示例

```
mysql> create table t(x int, index idx(x)) partition by hash(x) partitions 2;
Query OK, 0 rows affected (0.69 sec)

mysql> set @@tidb_analyze_version = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> analyze table t;
Query OK, 0 rows affected (0.20 sec)

mysql> show analyze status;
+--
    ↪ ------------+------------+----------------+-------------------+---------------+---------------------+---
    ↪
| Table_schema | Table_name | Partition_name | Job_info          | Processed_rows | Start_time
    ↪          | End_time            | State    | Fail_reason | Instance       | Process_ID |
+--
    ↪ ------------+------------+----------------+-------------------+---------------+---------------------+---
    ↪
| test         | t          | p1             | analyze index idx |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns   |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns   |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--
    ↪ ------------+------------+----------------+-------------------+---------------+---------------------+---
    ↪
4 rows in set (0.01 sec)

mysql> set @@tidb_analyze_version = 2;
Query OK, 0 rows affected (0.00 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.03 sec)

mysql> show analyze status;
+--
    ↪ ------------+------------+----------------+-------------------------------------------------------------
    ↪
| Table_schema | Table_name | Partition_name | Job_info
    ↪                                                          | Processed_rows | Start_time
    ↪          | End_time            | State    | Fail_reason | Instance       | Process_ID |
```

```
+--
    ↪ ------------+------------+---------------+-----------------------------------------------------------------
    ↪
| test         | t          | p1             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze index idx
    ↪                                                 |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx
    ↪                                                 |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns
    ↪                                                    |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns
    ↪                                                    |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--
    ↪ ------------+------------+---------------+-----------------------------------------------------------------
    ↪
6 rows in set (0.00 sec)
```

2.2.78.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.78.4 另请参阅

- ANALYZE_STATUS 表

2.2.79 SHOW [GLOBAL|SESSION] BINDINGS

SHOW BINDINGS 语句用于显示创建过的 SQL 绑定的相关信息。BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内显示执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

2.2.79.1 语法图

ShowStmt:

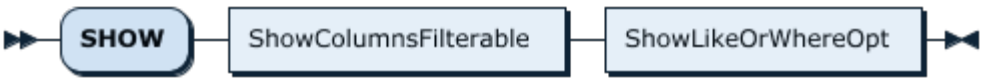


图 33: ShowStmt

ShowTargetFilterable:

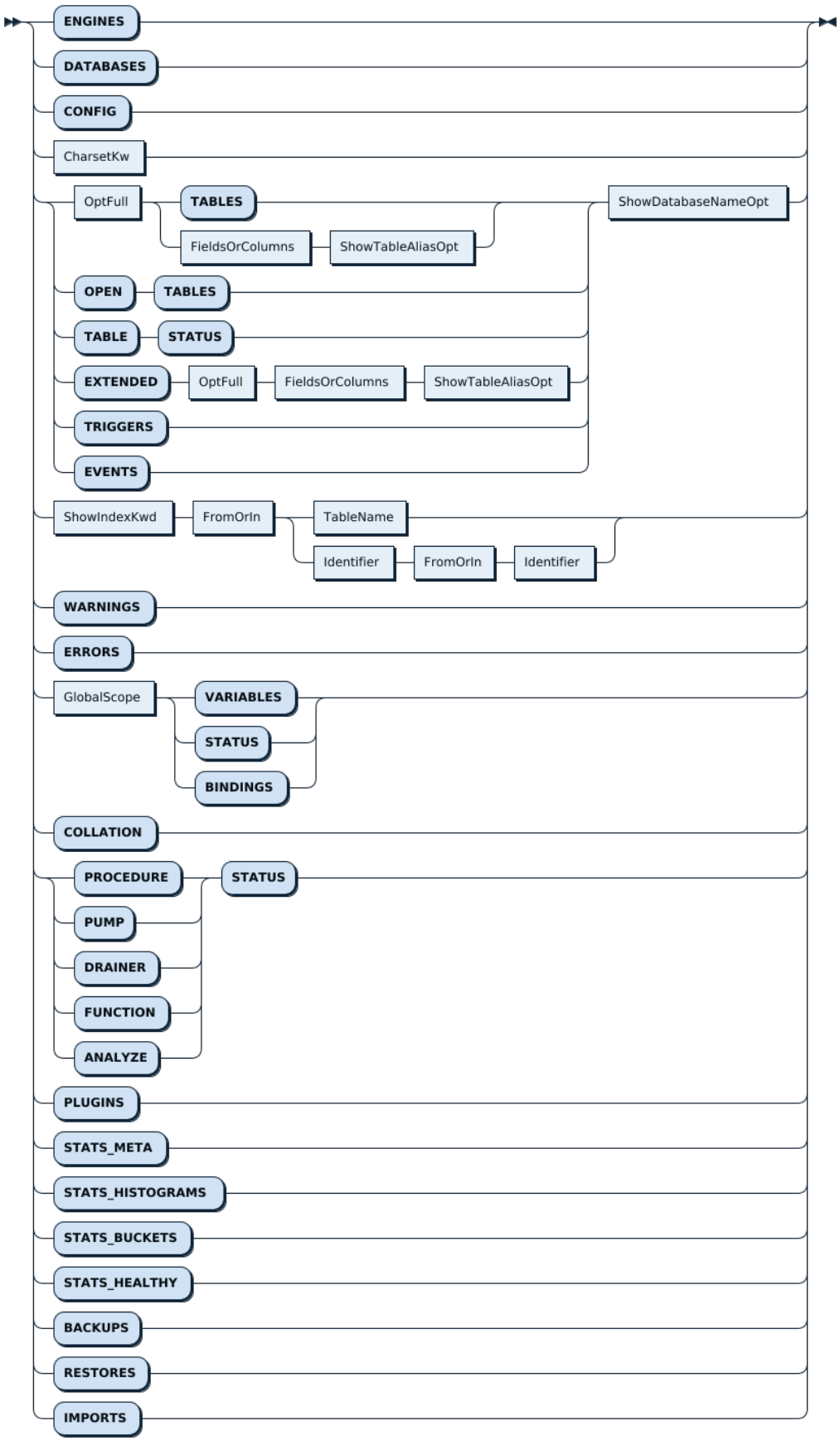


图 34: ShowTargetFilterable

GlobalScope:



图 35: GlobalScope

ShowLikeOrWhereOpt

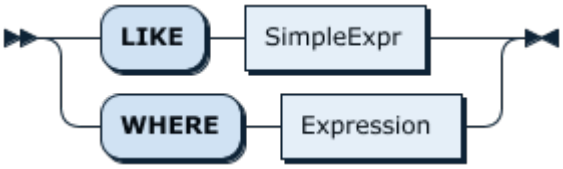


图 36: ShowLikeOrWhereOpt

2.2.79.2 语法说明

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhereOpt];
```

该语句会输出 GLOBAL 或者 SESSION 作用域内的执行计划绑定，在不指定作用域时默认作用域为 SESSION。目前 SHOW BINDINGS 会输出 8 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL
default_db	默认数据库名
status	状态，包括 using（正在使用）、deleted（已删除）、invalid（无效）、rejected（演进时被拒绝）和 pending verify（等待演进验证）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则

列名	说明
source	创建方式，包括 manual（由 create [global] binding 生成）、capture（由 tidb 自动创建生成）和 evolve（由 tidb 自动演进生成）

2.2.79.3 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

```
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object       |
    ↪ execution info                                                                | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                             | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                           | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
3 rows in set (0.02 sec)
```

```
CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪                                                                  | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

2.2.79.4 MySQL 兼容性

`SHOW [GLOBAL|SESSION] BINDINGS` 语句是 TiDB 对 MySQL 语法的扩展。

2.2.79.5 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- DROP [GLOBAL|SESSION] BINDING
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

2.2.80 SHOW BUILTINS

SHOW BUILTINS 语句用于列出 TiDB 中所有的内置函数。

2.2.80.1 语法图

ShowBuiltinsStmt:

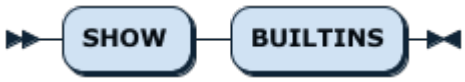


图 37: ShowBuiltinsStmt

2.2.80.2 示例

```
SHOW BUILTINS;
```

```
+-----------------------------+
| Supported_builtin_functions |
+-----------------------------+
| abs                         |
| acos                        |
| adddate                     |
| addtime                     |
| aes_decrypt                 |
| aes_encrypt                 |
| and                         |
| any_value                   |
| ascii                       |
| asin                        |
| atan                        |
| atan2                       |
| benchmark                   |
| bin                         |
| bit_count                   |
| bit_length                  |
| bitand                      |
| bitneg                      |
```

```
| bitor                     |
| bitxor                    |
| case                      |
| ceil                      |
| ceiling                   |
| char_func                 |
| char_length               |
| character_length          |
| charset                   |
| coalesce                  |
| coercibility              |
| collation                 |
| compress                  |
| concat                    |
| concat_ws                 |
| connection_id             |
| conv                      |
| convert                   |
| convert_tz                |
| cos                       |
| cot                       |
| crc32                     |
| curdate                   |
| current_date              |
| current_role              |
| current_time              |
| current_timestamp         |
| current_user              |
| curtime                   |
| database                  |
| date                      |
| date_add                  |
| date_format               |
| date_sub                  |
| datediff                  |
| day                       |
| dayname                   |
| dayofmonth                |
| dayofweek                 |
| dayofyear                 |
| decode                    |
| default_func              |
| degrees                   |
| des_decrypt               |
| des_encrypt               |
```

```
| div                        |
| elt                        |
| encode                     |
| encrypt                    |
| eq                         |
| exp                        |
| export_set                 |
| extract                    |
| field                      |
| find_in_set                |
| floor                      |
| format                     |
| format_bytes               |
| format_nano_time           |
| found_rows                 |
| from_base64                |
| from_days                  |
| from_unixtime              |
| ge                         |
| get_format                 |
| get_lock                   |
| getparam                   |
| getvar                     |
| greatest                   |
| gt                         |
| hex                        |
| hour                       |
| if                         |
| ifnull                     |
| in                         |
| inet6_aton                 |
| inet6_ntoa                 |
| inet_aton                  |
| inet_ntoa                  |
| insert_func                |
| instr                      |
| intdiv                     |
| interval                   |
| is_free_lock               |
| is_ipv4                    |
| is_ipv4_compat             |
| is_ipv4_mapped             |
| is_ipv6                    |
| is_used_lock               |
| isfalse                    |
```

```
| isnull                    |
| istrue                    |
| json_array                |
| json_array_append         |
| json_array_insert         |
| json_contains             |
| json_contains_path        |
| json_depth                |
| json_extract              |
| json_insert               |
| json_keys                 |
| json_length               |
| json_merge                |
| json_merge_patch          |
| json_merge_preserve       |
| json_object               |
| json_pretty               |
| json_quote                |
| json_remove               |
| json_replace              |
| json_search               |
| json_set                  |
| json_storage_size         |
| json_type                 |
| json_unquote              |
| json_valid                |
| last_day                  |
| last_insert_id            |
| lastval                   |
| lcase                     |
| le                        |
| least                     |
| left                      |
| leftshift                 |
| length                    |
| like                      |
| ln                        |
| load_file                 |
| localtime                 |
| localtimestamp            |
| locate                    |
| log                       |
| log10                     |
| log2                      |
| lower                     |
```

```
| lpad                      |
| lt                        |
| ltrim                     |
| make_set                  |
| makedate                  |
| maketime                  |
| master_pos_wait           |
| md5                       |
| microsecond               |
| mid                       |
| minus                     |
| minute                    |
| mod                       |
| month                     |
| monthname                 |
| mul                       |
| name_const                |
| ne                        |
| nextval                   |
| not                       |
| now                       |
| nulleq                    |
| oct                       |
| octet_length              |
| old_password              |
| or                        |
| ord                       |
| password_func             |
| period_add                |
| period_diff               |
| pi                        |
| plus                      |
| position                  |
| pow                       |
| power                     |
| quarter                   |
| quote                     |
| radians                   |
| rand                      |
| random_bytes              |
| regexp                    |
| release_all_locks         |
| release_lock              |
| repeat                    |
| replace                   |
```

```
| reverse                    |
| right                      |
| rightshift                 |
| round                      |
| row_count                  |
| rpad                       |
| rtrim                      |
| schema                     |
| sec_to_time                |
| second                     |
| session_user               |
| setval                     |
| setvar                     |
| sha                        |
| sha1                       |
| sha2                       |
| sign                       |
| sin                        |
| sleep                      |
| space                      |
| sqrt                       |
| str_to_date                |
| strcmp                     |
| subdate                    |
| substr                     |
| substring                  |
| substring_index            |
| subtime                    |
| sysdate                    |
| system_user                |
| tan                        |
| tidb_decode_key            |
| tidb_decode_plan           |
| tidb_is_ddl_owner          |
| tidb_parse_tso             |
| tidb_version               |
| time                       |
| time_format                |
| time_to_sec                |
| timediff                   |
| timestamp                  |
| timestampadd               |
| timestampdiff              |
| to_base64                  |
| to_days                    |
```

```
| to_seconds                    |
| trim                          |
| truncate                      |
| ucase                         |
| unaryminus                    |
| uncompress                    |
| uncompressed_length           |
| unhex                         |
| unix_timestamp                |
| upper                         |
| user                          |
| utc_date                      |
| utc_time                      |
| utc_timestamp                 |
| uuid                          |
| uuid_short                    |
| validate_password_strength    |
| version                       |
| week                          |
| weekday                       |
| weekofyear                    |
| weight_string                 |
| xor                           |
| year                          |
| yearweek                      |
+-------------------------------+
268 rows in set (0.00 sec)
```

2.2.80.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.81 SHOW CHARACTER SET

`SHOW CHARACTER SET` 语句提供 TiDB 中可用字符集的静态列表。此列表不反映当前连接或用户的任何属性。

2.2.81.1 语法图

ShowCharsetStmt:



图 38: ShowCharsetStmt

CharsetKw:

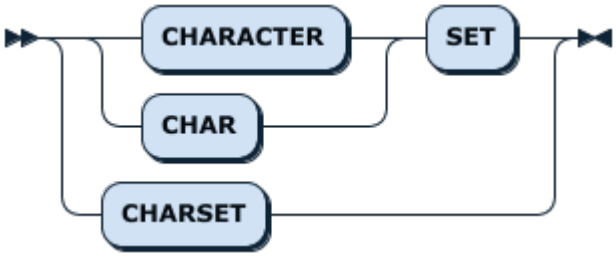


图 39: CharsetKw

2.2.81.2 示例

```
SHOW CHARACTER SET;
```

```
+---------+---------------+-------------------+--------+
| Charset | Description   | Default collation | Maxlen |
+---------+---------------+-------------------+--------+
| utf8    | UTF-8 Unicode | utf8_bin          |      3 |
| utf8mb4 | UTF-8 Unicode | utf8mb4_bin       |      4 |
| ascii   | US ASCII      | ascii_bin         |      1 |
| latin1  | Latin1        | latin1_bin        |      1 |
| binary  | binary        | binary            |      1 |
+---------+---------------+-------------------+--------+
5 rows in set (0.00 sec)
```

2.2.81.3 MySQL 兼容性

SHOW CHARACTER SET 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可以参考与 MySQL 兼容性对比。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.81.4 另请参阅

- SHOW COLLATION
- 字符集和排序规则

2.2.82 SHOW COLLATION

SHOW COLLATION 语句用于提供一个静态的排序规则列表，确保与 MySQL 客户端库的兼容性。

> **注意：**
>
> SHOW COLLATION 所展示的排序规则列表与 TiDB 集群是否开启新排序规则框架有关，详情请见TiDB 字符集和排序规则。

2.2.82.1 语法图

ShowCollationStmt:

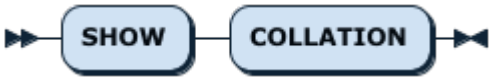


图 40: ShowCollationStmt

2.2.82.2 示例

若未开启新排序规则框架，仅展示二进制排序规则：

```
SHOW COLLATION;
```

```
+-------------+---------+------+---------+----------+---------+
| Collation   | Charset | Id   | Default | Compiled | Sortlen |
+-------------+---------+------+---------+----------+---------+
| utf8mb4_bin | utf8mb4 |   46 | Yes     | Yes      |       1 |
| latin1_bin  | latin1  |   47 | Yes     | Yes      |       1 |
| binary      | binary  |   63 | Yes     | Yes      |       1 |
| ascii_bin   | ascii   |   65 | Yes     | Yes      |       1 |
| utf8_bin    | utf8    |   83 | Yes     | Yes      |       1 |
+-------------+---------+------+---------+----------+---------+
5 rows in set (0.02 sec)
```

若开启了新排序规则框架，则在二进制排序规则之外，额外支持 `utf8_general_ci` 和 `utf8mb4_general_ci` 两种大小写和口音不敏感的排序规则：

```
SHOW COLLATION;
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| ascii_bin          | ascii   |   65 | Yes     | Yes      |       1 |
| binary             | binary  |   63 | Yes     | Yes      |       1 |
| gbk_bin            | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci     | gbk     |   28 | Yes     | Yes      |       1 |
| latin1_bin         | latin1  |   47 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |   83 | Yes     | Yes      |       1 |
| utf8_general_ci    | utf8    |   33 |         | Yes      |       1 |
| utf8_unicode_ci    | utf8    |  192 |         | Yes      |       1 |
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci | utf8mb4 |  224 |         | Yes      |       1 |
```

```
+--------------------+---------+------+---------+----------+---------+
11 rows in set (0.001 sec)
```

2.2.82.3 MySQL 兼容性

SHOW COLLATION 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可参考与 MySQL 兼容性对比。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

2.2.82.4 另请参阅

- SHOW CHARACTER SET
- 字符集和排序规则

2.2.83 SHOW [FULL] COLUMNS FROM

SHOW [FULL] COLUMNS FROM <table_name> 语句用于以表格格式描述表或视图中的列。可选关键字 FULL 用于显示当前用户对该列的权限，以及表定义中的 comment。

SHOW [FULL] FIELDS FROM <table_name>、DESC <table_name>、DESCRIBE <table_name> 和 EXPLAIN <table_name> 语句都是 SHOW [FULL] COLUMNS FROM 的别名。

> **注意：**
>
> DESC TABLE <table_name>、DESCRIBE TABLE <table_name> 和 EXPLAIN TABLE <table_name> 与上面的语句并不等价，它们是DESC SELECT * FROM <table_name> 的别名。

2.2.83.1 语法图

ShowStmt:

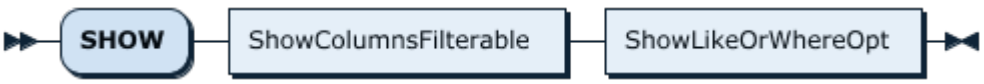


图 41: ShowStmt

ShowColumnsFilterable:



图 42: ShowColumnsFilterable

OptFull:



图 43: OptFull

FieldsOrColumns:



图 44: FieldsOrColumns

ShowTableAliasOpt:

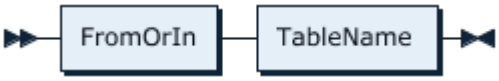


图 45: ShowTableAliasOpt

FromOrIn:

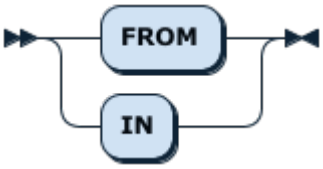


图 46: FromOrIn

TableName:

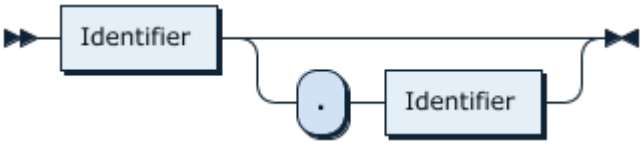


图 47: TableName

ShowDatabaseNameOpt:

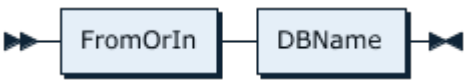


图 48: ShowDatabaseNameOpt

DBName:



图 49: DBName

ShowLikeOrWhereOpt:

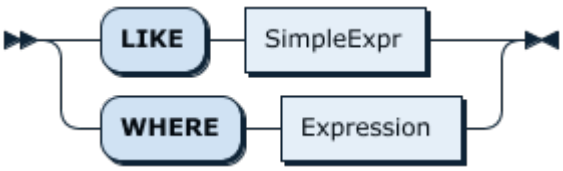


图 50: ShowLikeOrWhereOpt

2.2.83.2 示例

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

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

```
show columns from v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
desc v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
describe v1;
```

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

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

```
explain v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
show fields from v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
show full columns from v1;
```

```
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
| Field | Type      | Collation | Null | Key  | Default | Extra | Privileges
    ↪                     | Comment |
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
| 1     | bigint(1) | NULL      | YES  |      | NULL    |       | select,insert,update,references
    ↪  |         |
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
1 row in set (0.00 sec)
```

```
show full columns from mysql.user;
```

```
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
| Field                 | Type          | Collation   | Null | Key  | Default | Extra |
    ↪ Privileges                      | Comment |
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
```

```
| Host                  | char(64)      | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    ↪ insert,update,references |         |
| User                  | char(32)      | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    ↪ insert,update,references |         |
| authentication_string | text          | utf8mb4_bin | YES  |      | NULL    |       | select,
    ↪ insert,update,references |         |
| Select_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Insert_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Update_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Delete_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Drop_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Process_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Grant_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| References_priv       | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Alter_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Show_db_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Super_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_tmp_table_priv | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Lock_tables_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Execute_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_view_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Show_view_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_routine_priv   | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Alter_routine_priv    | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
```

```
| Index_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_user_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Event_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Trigger_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_role_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Drop_role_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Account_locked        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Shutdown_priv         | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Reload_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| FILE_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Config_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
+-----------------------+---------------+-------------+------+------+---------+-------+-------------------------
    ↪
33 rows in set (0.01 sec)
```

2.2.83.3 MySQL 兼容性

SHOW [FULL] COLUMNS FROM 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.83.4 另请参阅

- SHOW CREATE TABLE

2.2.84 SHOW CONFIG

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

SHOW CONFIG 语句用于展示 TiDB 各个组件当前正在应用的配置，请注意，配置与系统变量作用于不同维度，请不要混淆，如果希望获取系统变量信息，请使用SHOW VARIABLES 语法。

2.2.84.1 语法图

ShowStmt:

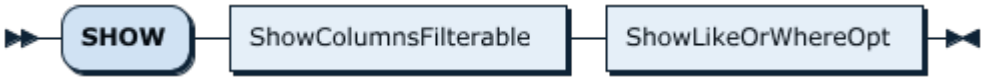


图 51: ShowStmt

ShowTargetFilterable:

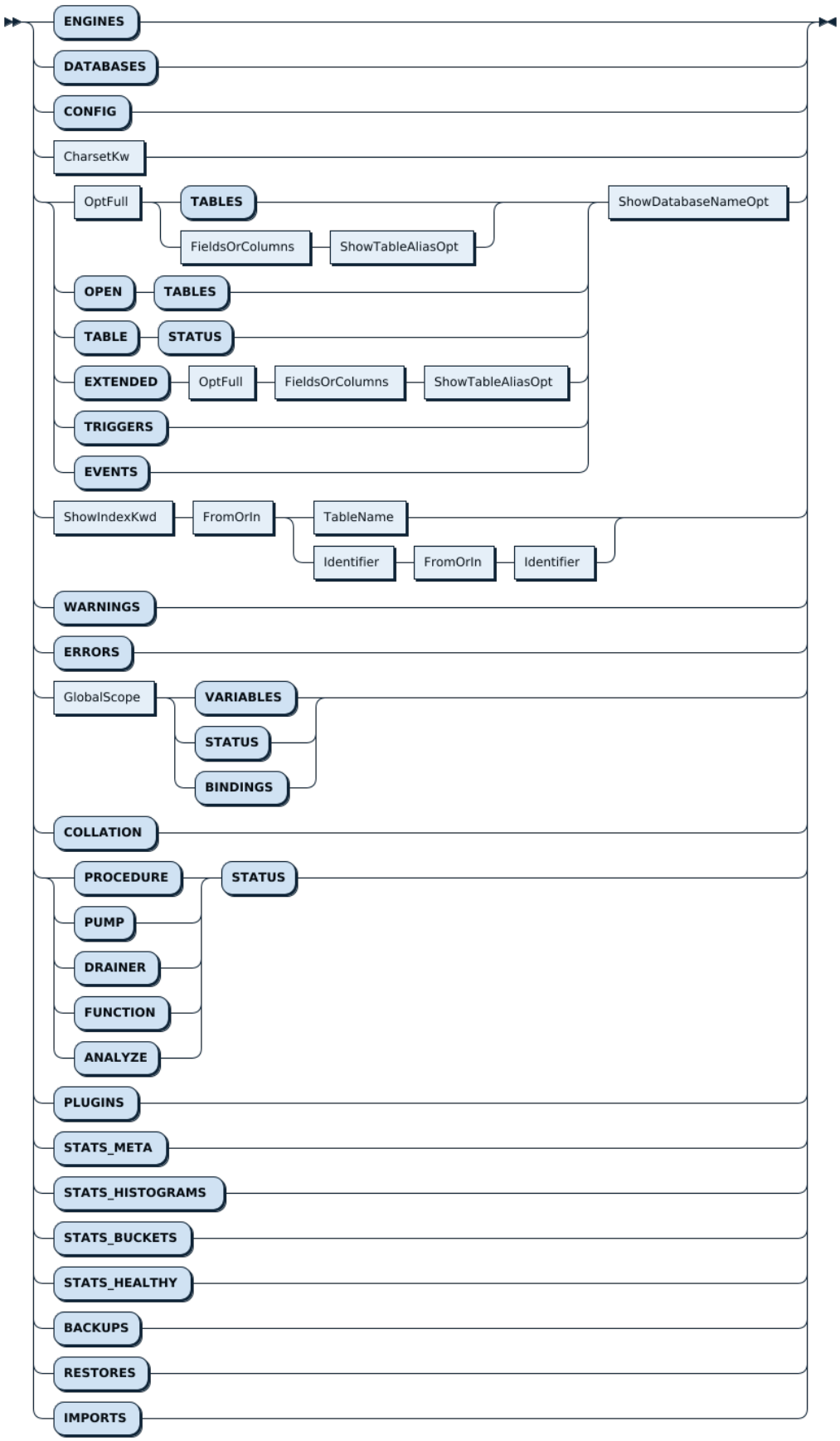


图 52: ShowTargetFilterable

2.2.84.2 示例

显示所有配置：

```
SHOW CONFIG;
```

```
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| Type | Instance       | Name                                            | Value
    ↪                                                          |
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| tidb | 127.0.0.1:4000 | advertise-address                               | 127.0.0.1
    ↪                                                        |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                            |
    ↪                                                          |
| tidb | 127.0.0.1:4000 | binlog.enable                                   | false
    ↪                                                          |
...
120 rows in set (0.01 sec)
```

显示 type 是 tidb 的配置：

```
SHOW CONFIG WHERE type = 'tidb' AND name = 'advertise-address';
```

```
+------+----------------+------------------+-----------+
| Type | Instance       | Name             | Value     |
+------+----------------+------------------+-----------+
| tidb | 127.0.0.1:4000 | advertise-address | 127.0.0.1 |
+------+----------------+------------------+-----------+
1 row in set (0.05 sec)
```

也可以用 LIKE 子句来显示 type 是 tidb 的配置：

```
SHOW CONFIG LIKE 'tidb';
```

```
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| Type | Instance       | Name                                            | Value
    ↪                                                          |
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| tidb | 127.0.0.1:4000 | advertise-address                               | 127.0.0.1
    ↪                                                        |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                            |
    ↪                                                          |
| tidb | 127.0.0.1:4000 | binlog.enable                                   | false
    ↪                                                          |
```

```
...
40 rows in set (0.01 sec)
```

2.2.84.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.84.4 另请参阅

- SHOW VARIABLES

2.2.85 SHOW CREATE PLACEMENT POLICY

`SHOW CREATE PLACEMENT POLICY` 语句可用于查看放置策略当前的定义，并在另一个 TiDB 集群中重新创建该策略。

2.2.85.1 语法图

```
ShowCreatePlacementPolicyStmt ::=
    "SHOW" "CREATE" "PLACEMENT" "POLICY" PolicyName

PolicyName ::=
    Identifier
```

2.2.85.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW CREATE PLACEMENT POLICY p1\G;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

***************************[ 1. row ]***************************
Policy        | p1
Create Policy | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-
    ↪ west-1" FOLLOWERS=4
1 row in set (0.00 sec)
```

2.2.85.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.85.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- ALTER PLACEMENT POLICY
- DROP PLACEMENT POLICY

2.2.86 SHOW CREATE SEQUENCE

`SHOW CREATE SEQUENCE` 语句用于查看一个序列的详细信息，类似于 `SHOW CREATE TABLE` 语句。

2.2.86.1 语法图

ShowCreateSequenceStmt:



图 53: ShowCreateSequenceStmt

TableName:

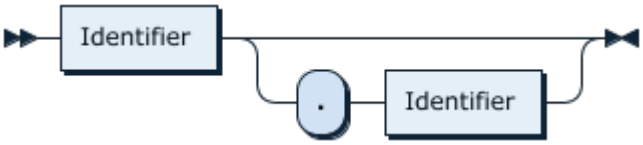


图 54: TableName

2.2.86.2 示例

```
CREATE SEQUENCE seq;
```

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

```
SHOW CREATE SEQUENCE seq;
```

```
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
```

```
| seq   | CREATE SEQUENCE `seq` start with 1 minvalue 1 maxvalue 9223372036854775806 increment by
    ↪  1 cache 1000 nocycle ENGINE=InnoDB |
+-------+-----------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

2.2.86.3 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

2.2.86.4 另请参阅

- CREATE SEQUENCE
- DROP SEQUENCE

2.2.87 SHOW CREATE TABLE

`SHOW CREATE TABLE` 语句用于显示用 SQL 重新创建已有表的确切语句。

2.2.87.1 语法图

ShowCreateTableStmt:

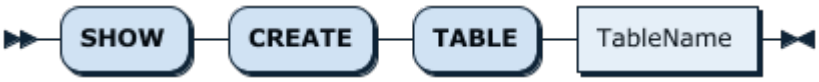


图 55: ShowCreateTableStmt

TableName:

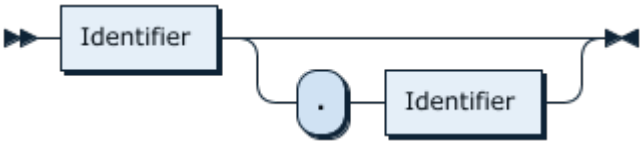


图 56: TableName

2.2.87.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

2.2.87.3 MySQL 兼容性

`SHOW CREATE TABLE` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.87.4 另请参阅

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

2.2.88 SHOW CREATE USER

`SHOW CREATE USER` 语句用于显示如何使用 `CREATE USER` 语法来重新创建用户。

2.2.88.1 语法图

ShowCreateUserStmt:

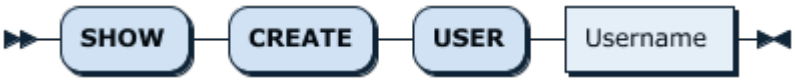


图 57: ShowCreateUserStmt

Username:

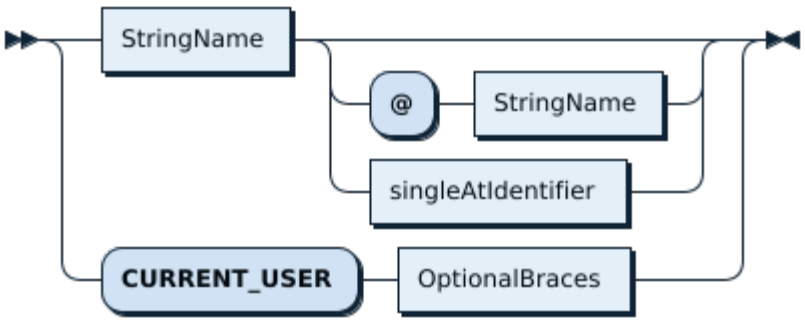


图 58: Username

2.2.88.2 示例

```
SHOW CREATE USER 'root';
```

```
+-------------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for root@%
    ↪
    ↪ |
+-------------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'root'@'%' IDENTIFIED WITH 'mysql_native_password' AS '' REQUIRE NONE PASSWORD
    ↪ EXPIRE DEFAULT ACCOUNT UNLOCK |
+-------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

mysql> SHOW GRANTS FOR 'root';
+----------------------------------------+
| Grants for root@%                      |
+----------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+----------------------------------------+
1 row in set (0.00 sec)
```

2.2.88.3 MySQL 兼容性

- SHOW CREATE USER 的输出结果旨在匹配 MySQL，但 TiDB 尚不支持若干 CREATE 选项。尚未支持的选项在语句执行过程中会被解析但会被跳过执行。详情可参阅 [security compatibility]。

2.2.88.4 另请参阅

- CREATE USER

- SHOW GRANTS
- DROP USER

2.2.89 SHOW DATABASES

`SHOW DATABASES` 语句用于显示当前用户有权访问的数据库列表。当前用户无权访问的数据库将从列表中隐藏。`information_schema` 数据库始终出现在列表的最前面。

`SHOW SCHEMAS` 是 `SHOW DATABASES` 语句的别名。

2.2.89.1 语法图

ShowDatabasesStmt:



图 59: ShowDatabasesStmt

ShowLikeOrWhereOpt:

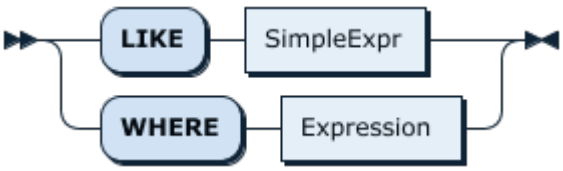


图 60: ShowLikeOrWhereOpt

2.2.89.2 示例

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

```
CREATE DATABASE mynewdb;
```

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

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mynewdb            |
| mysql              |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

2.2.89.3 MySQL 兼容性

SHOW DATABASES 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.89.4 另请参阅

- SHOW SCHEMAS
- DROP DATABASE
- CREATE DATABASE

2.2.90 SHOW DRAINER STATUS

SHOW DRAINER STATUS 语句用于显示集群中所有 Drainer 的状态信息。

2.2.90.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

2.2.90.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.90.3 另请参阅

- SHOW PUMP STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

2.2.91 SHOW ENGINES

`SHOW ENGINES` 语句用于列出所有支持的存储引擎，该语法仅提供 MySQL 兼容性。

2.2.91.1 语法图

ShowEnginesStmt:

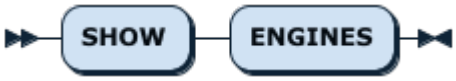


图 61: ShowEnginesStmt

```
SHOW ENGINES;
```

2.2.91.2 示例

```
SHOW ENGINES;
```

```
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
| Engine | Support | Comment                                                        | Transactions |
    ↪ XA   | Savepoints |
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES          |
    ↪ YES  | YES        |
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
1 row in set (0.00 sec)
```

2.2.91.3 MySQL 兼容性

- `SHOW ENGINES` 语句始终只返回 InnoDB 作为其支持的引擎。但 TiDB 内部通常使用 TiKV 作为存储引擎。

2.2.92 SHOW ERRORS

SHOW ERRORS 语句用于显示已执行语句中的错误。一旦先前的语句成功执行，就会清除错误缓冲区，这时 SHOW ERRORS 会返回一个空集。

当前的 sql_mode 很大程度决定了哪些语句会产生错误与警告。

2.2.92.1 语法图

ShowErrorsStmt:

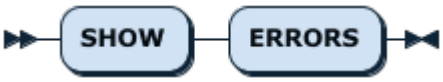


图 62: ShowErrorsStmt

2.2.92.2 示例

```
select invalid;
```

```
ERROR 1054 (42S22): Unknown column 'invalid' in 'field list'
```

```
create invalid;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 14 near "invalid"
```

```
SHOW ERRORS;
```

```
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
| Level | Code | Message
    ↪
    ↪ |
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
| Error | 1054 | Unknown column 'invalid' in 'field list'
    ↪
    ↪ |
| Error | 1064 | You have an error in your SQL syntax; check the manual that corresponds to your
    ↪ TiDB version for the right syntax to use line 1 column 14 near "invalid"  |
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
CREATE invalid2;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 15 near "invalid2"
```

```
SELECT 1;
```

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

```
SHOW ERRORS;
```

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

2.2.92.3 MySQL 兼容性

SHOW ERRORS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.92.4 另请参阅

- SHOW WARNINGS

2.2.93 SHOW [FULL] FIELDS FROM

SHOW [FULL] FIELDS FROM 是SHOW [FULL] COLUMNS FROM 的别名。包含该语句提供了 MySQL 兼容性。

2.2.94 SHOW GRANTS

SHOW GRANTS 语句用于显示与用户关联的权限列表。与在 MySQL 中一样，USAGE 权限表示登录 TiDB 的能力。

2.2.94.1 语法图

ShowGrantsStmt:

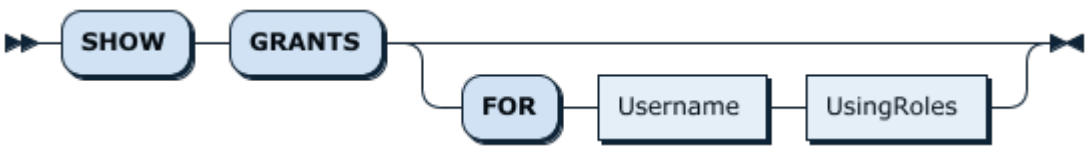


图 63: ShowGrantsStmt

Username:

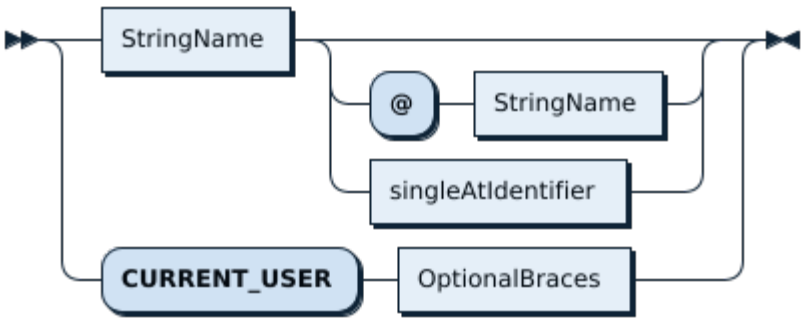


图 64: Username

UsingRoles:



图 65: UsingRoles

RolenameList:

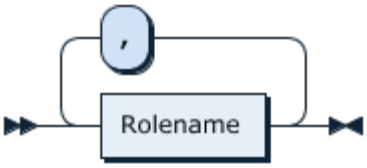


图 66: RolenameList

Rolename:

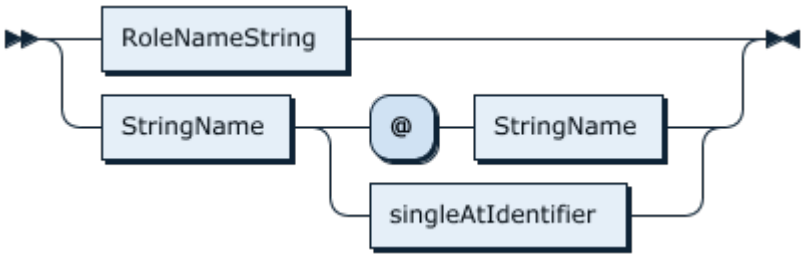


图 67: Rolename

2.2.94.2 示例

```
SHOW GRANTS;
```

```
+----------------------------------------+
| Grants for User                        |
```

```
+----------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+----------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW GRANTS FOR 'u1';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'u1' on host '%'
```

```
CREATE USER u1;
```

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

```
GRANT SELECT ON test.* TO u1;
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
SHOW GRANTS FOR u1;
```

```
+-----------------------------------+
| Grants for u1@%                   |
+-----------------------------------+
| GRANT USAGE ON *.* TO 'u1'@'%'    |
| GRANT Select ON test.* TO 'u1'@'%' |
+-----------------------------------+
2 rows in set (0.00 sec)
```

2.2.94.3 MySQL 兼容性

`SHOW GRANTS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.94.4 另请参阅

- SHOW CREATE USER
- GRANT

2.2.95 SHOW INDEX [FROM|IN]

`SHOW INDEX [FROM|IN]` 语句是`SHOW INDEXES [FROM|IN]` 的别名。包含该语句提供了 MySQL 兼容性。

2.2.96 SHOW INDEXES [FROM|IN]

`SHOW INDEXES [FROM|IN]` 语句用于列出指定表上的索引。`SHOW INDEX [FROM | IN]` 和 `SHOW KEYS [FROM | IN]` 是该语句的别名。包含该语句提供了 MySQL 兼容性。

2.2.96.1 语法图

ShowIndexStmt:

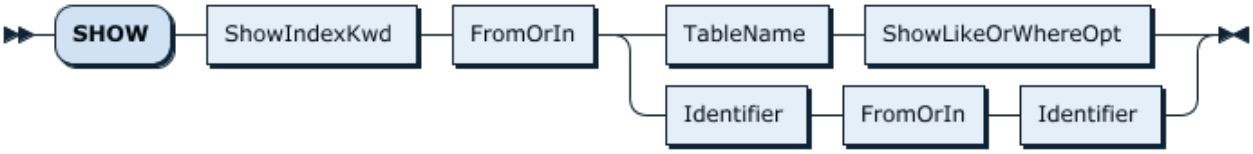


图 68: ShowIndexStmt

ShowIndexKwd:

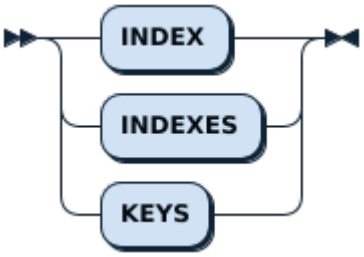


图 69: ShowIndexKwd

FromOrIn:

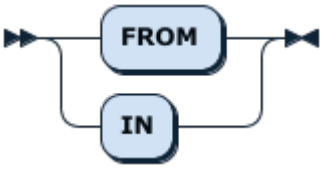


图 70: FromOrIn

TableName:

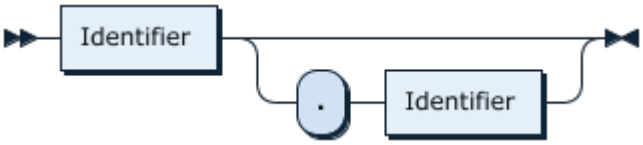


图 71: TableName

ShowLikeOrWhereOpt:

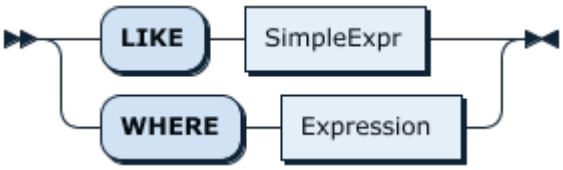


图 72: ShowLikeOrWhereOpt

2.2.96.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT, INDEX(col1));
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW INDEXES FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

```
SHOW INDEX FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               | YES
    | NULL       |
```

```
+-------+------------+----------+--------------+-------------+----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

```
SHOW KEYS FROM t1;
```

```
+-------+------------+----------+--------------+-------------+----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

2.2.96.3 MySQL 兼容性

MySQL 中的 `Cardinality` 列返回该索引上不同值的个数，而 TiDB 中的 `Cardinality` 列始终返回 0。

2.2.96.4 另请参阅

- SHOW CREATE TABLE
- DROP INDEX
- CREATE INDEX

2.2.97 SHOW KEYS [FROM|IN]

`SHOW KEYS [FROM|IN]` 语句是`SHOW INDEXES [FROM|IN]` 语句的别名。包含该语句提供了 MySQL 兼容性。

2.2.98 SHOW MASTER STATUS

`SHOW MASTER STATUS` 语句用于显示集群当前最新的 TSO 信息。

2.2.98.1 示例

```
SHOW MASTER STATUS;
```

```
+-------------+--------------------+--------------+------------------+-------------------+
| File        | Position           | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-------------+--------------------+--------------+------------------+-------------------+
| tidb-binlog | 416916363252072450 |              |                  |                   |
+-------------+--------------------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

2.2.98.2 MySQL 兼容性

`SHOW MASTER STATUS` 语句与 MySQL 兼容，但是执行结果有差异，在 MySQL 中执行结果为 binlog 的位置信息，而在 TiDB 中为最新的 TSO 信息。

2.2.98.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

2.2.99 SHOW PLACEMENT

`SHOW PLACEMENT` 汇总了所有放置策略 (placement policy)，并用统一的形式呈现相关信息。

本语句返回结果中的 `Scheduling_State` 列标识了 Placement Driver (PD) 在当前对象上的调度进度，有以下可能结果：

- `PENDING`: PD 没有进行调度。可能的原因之一是放置规则虽然语法上正确，但集群拓扑并不满足。比如指定 `FOLLOWERS=4` 但只有 3 个可用作 follower 的 TiKV 实例。
- `INPROGRESS`: PD 正在进行调度。
- `SCHEDULED`: PD 调度完成。

2.2.99.1 语法图

```
ShowStmt ::=
    "PLACEMENT"
```

2.2.99.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW PLACEMENT;
```

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

Query OK, 0 rows affected (0.00 sec)

+--
    ↪ -------------+-------------------------------------------------------------------------+------------------
    ↪
| Target         | Placement                                                               |
    ↪ Scheduling_State |
+--
    ↪ -------------+-------------------------------------------------------------------------+------------------
    ↪
| POLICY p1      | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 | NULL
    ↪              |
| DATABASE test | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    ↪ INPROGRESS       |
| TABLE test.t1 | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    ↪ INPROGRESS       |
+--
    ↪ -------------+-------------------------------------------------------------------------+------------------
    ↪
4 rows in set (0.00 sec)
```

2.2.99.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.99.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT FOR
- CREATE PLACEMENT POLICY

2.2.100 SHOW PLACEMENT FOR

SHOW PLACEMENT FOR 用于汇总所有放置策略 (placement policy)，并用统一的形式呈现特定表、数据库或分区的信息。

本语句返回结果中的 Scheduling_State 列标识了 Placement Driver (PD) 在当前对象上的调度进度，有以下可能的结果：

- PENDING: PD 没有进行调度。可能的原因之一是放置规则虽然语法上正确，但集群拓扑并不满足。比如指定 FOLLOWERS=4 但只有 3 个可用作 follower 的 TiKV 实例。
- INPROGRESS: PD 正在进行调度。
- SCHEDULED: PD 调度完成。

2.2.100.1 语法图

```
ShowStmt ::=
    "PLACEMENT" "FOR" ShowPlacementTarget

ShowPlacementTarget ::=
    DatabaseSym DBName
|   "TABLE" TableName
|   "TABLE" TableName "PARTITION" Identifier
```

2.2.100.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
use test;
ALTER DATABASE test PLACEMENT POLICY=p1;
CREATE TABLE t1 (a INT);
SHOW PLACEMENT FOR DATABASE test;
SHOW PLACEMENT FOR TABLE t1;
SHOW CREATE TABLE t1\G;
CREATE TABLE t3 (a INT) PARTITION BY RANGE (a) (PARTITION p1 VALUES LESS THAN (10), PARTITION p2
    ↪ VALUES LESS THAN (20));
SHOW PLACEMENT FOR TABLE t3 PARTITION p1\G;
```

```
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

+--
    ↪ -------------+-------------------------------------------------------------------------+------------------+
    ↪
| Target        | Placement                                                               |
    ↪ Scheduling_State |
+--
    ↪ -------------+-------------------------------------------------------------------------+------------------+
    ↪
| DATABASE test | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    ↪ INPROGRESS       |
+--
    ↪ -------------+-------------------------------------------------------------------------+------------------+
    ↪
1 row in set (0.00 sec)

+---------------+-------------+------------------+
```

```
| Target        | Placement   | Scheduling_State |
+---------------+-------------+------------------+
| TABLE test.t1 | FOLLOWERS=4 | INPROGRESS       |
+---------------+-------------+------------------+
1 row in set (0.00 sec)

***************************[ 1. row ]***************************
Table        | t1
Create Table | CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /*T![placement] PLACEMENT POLICY=`p1`
    ↪  */
1 row in set (0.00 sec)

***************************[ 1. row ]***************************
Target           | TABLE test.t3 PARTITION p1
Placement        | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4
Scheduling_State | PENDING
1 row in set (0.00 sec)
```

2.2.100.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.100.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY

2.2.101 SHOW PLACEMENT LABELS

SHOW PLACEMENT LABELS 汇总可用于放置规则 (Placement Rules) 的标签 (label) 和值。

2.2.101.1 语法图

```
ShowStmt ::=
    "PLACEMENT" "LABELS"
```

2.2.101.2 示例

```
SHOW PLACEMENT LABELS;
```

```
+--------+----------------+
| Key    | Values         |
+--------+----------------+
| region | ["us-east-1"]  |
| zone   | ["us-east-1a"] |
+--------+----------------+
2 rows in set (0.00 sec)
```

2.2.101.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.101.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY

2.2.102 SHOW PLUGINS

`SHOW PLUGINS` 用于查看 TiDB 安装的插件，各个插件运行的状态以及插件版本信息。

2.2.102.1 语法图

ShowStmt:

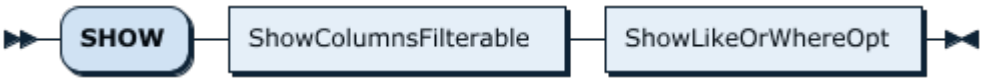


图 73: ShowStmt

ShowTargetFilterable:

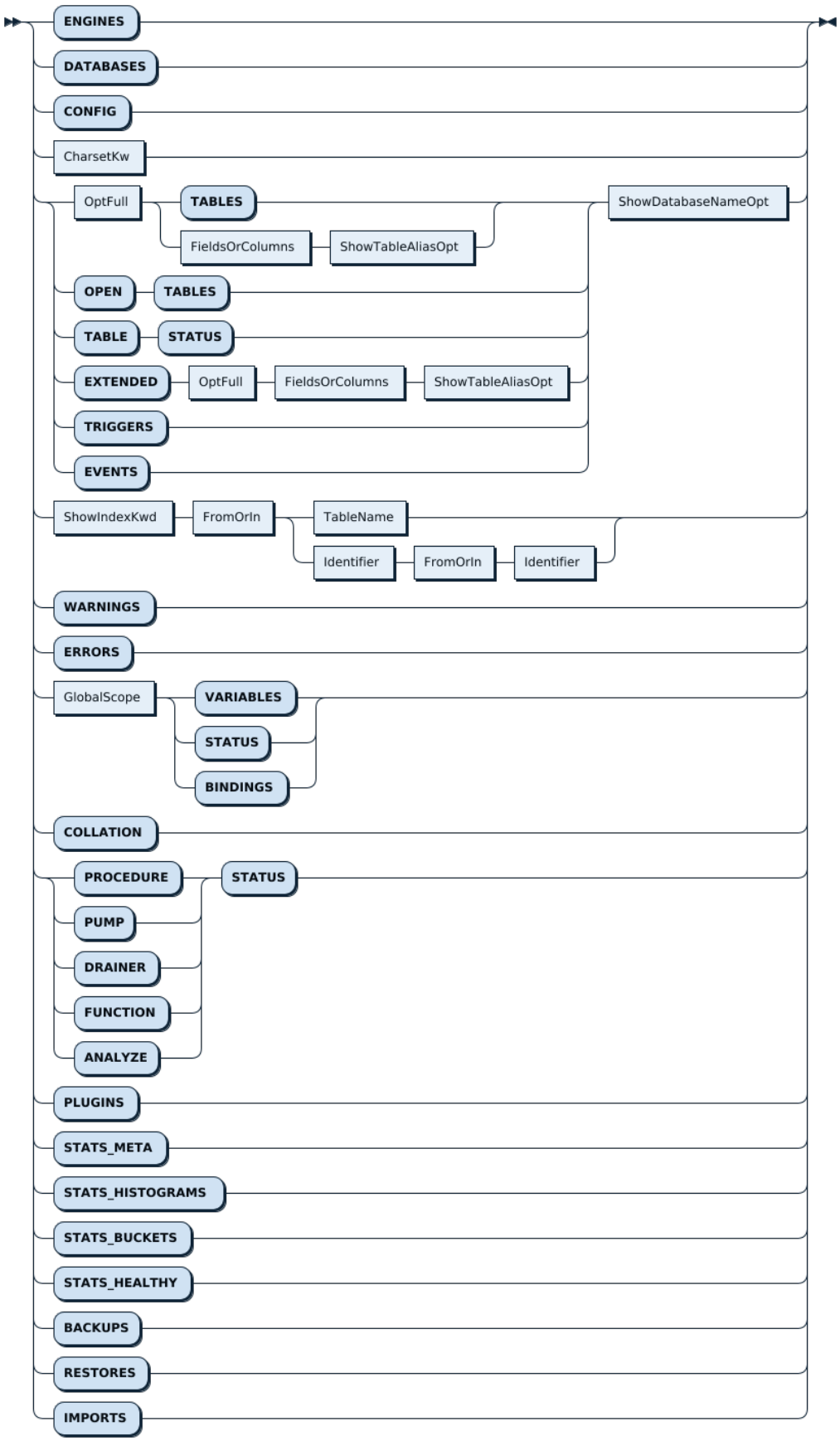


图 74: ShowTargetFilterable

2.2.102.2 示例

```
SHOW PLUGINS;
```

```
+-------+--------------+-------+----------------------------+---------+---------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+---------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |         | 1       |
+-------+--------------+-------+----------------------------+---------+---------+
1 row in set (0.000 sec)
```

SHOW PLUGINS 也支持模糊匹配：

```
SHOW PLUGINS LIKE 'a%';
```

```
+-------+--------------+-------+----------------------------+---------+---------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+---------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |         | 1       |
+-------+--------------+-------+----------------------------+---------+---------+
1 row in set (0.000 sec)
```

2.2.102.3 MySQL 兼容性

SHOW PLUGINS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.103 SHOW PRIVILEGES

SHOW PRIVILEGES 语句用于显示 TiDB 中可分配权限的列表。此列表为静态列表，不反映当前用户的权限。

2.2.103.1 语法图

ShowStmt:

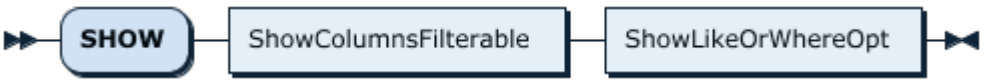


图 75: ShowStmt

2.2.103.2 示例

```
show privileges;
```

```
+-------------------------+---------------------------------------+-------------------------------------------------------------
    ↪
| Privilege               | Context                               | Comment
    ↪                                                 |
```

```
+-------------------------+--------------------------------------+--------------------------------
    ↪
| Alter                   | Tables                               | To alter the table
    ↪                                    |
| Alter                   | Tables                               | To alter the table
    ↪                                    |
| Alter routine           | Functions,Procedures                 | To alter or drop stored
    ↪ functions/procedures          |
| Create                  | Databases,Tables,Indexes             | To create new databases and
    ↪ tables                      |
| Create routine          | Databases                            | To use CREATE FUNCTION/
    ↪ PROCEDURE                       |
| Create temporary tables | Databases                            | To use CREATE TEMPORARY TABLE
    ↪                            |
| Create view             | Tables                               | To create new views
    ↪                                   |
| Create user             | Server Admin                         | To create new users
    ↪                                   |
| Delete                  | Tables                               | To delete existing rows
    ↪                                |
| Drop                    | Databases,Tables                     | To drop databases, tables,
    ↪ and views                   |
| Event                   | Server Admin                         | To create, alter, drop and
    ↪ execute events              |
| Execute                 | Functions,Procedures                 | To execute stored routines
    ↪                             |
| File                    | File access on server                | To read and write files on
    ↪ the server                  |
| Grant option            | Databases,Tables,Functions,Procedures | To give to other users those
    ↪ privileges you possess   |
| Index                   | Tables                               | To create or drop indexes
    ↪                              |
| Insert                  | Tables                               | To insert data into tables
    ↪                             |
| Lock tables             | Databases                            | To use LOCK TABLES (together
    ↪ with SELECT privilege)   |
| Process                 | Server Admin                         | To view the plain text of
    ↪ currently executing queries |
| Proxy                   | Server Admin                         | To make proxy user possible
    ↪                            |
| References              | Databases,Tables                     | To have references on tables
    ↪                           |
| Reload                  | Server Admin                         | To reload or refresh tables,
    ↪ logs and privileges      |
```

```
| Replication client      | Server Admin                          | To ask where the slave or
    ↪ master servers are          |
| Replication slave       | Server Admin                          | To read binary log events
    ↪ from the master             |
| Select                  | Tables                                | To retrieve rows from table
    ↪                           |
| Show databases          | Server Admin                          | To see all databases with
    ↪ SHOW DATABASES              |
| Show view               | Tables                                | To see views with SHOW CREATE
    ↪  VIEW                     |
| Shutdown                | Server Admin                          | To shut down the server
    ↪                               |
| Super                   | Server Admin                          | To use KILL thread, SET
    ↪ GLOBAL, CHANGE MASTER, etc.   |
| Trigger                 | Tables                                | To use triggers
    ↪                                       |
| Create tablespace       | Server Admin                          | To create/alter/drop
    ↪ tablespaces                       |
| Update                  | Tables                                | To update existing rows
    ↪                               |
| Usage                   | Server Admin                          | No privileges - allow connect
    ↪  only                     |
+-------------------------+---------------------------------------+-------------------------------------------
    ↪
32 rows in set (0.00 sec)
```

2.2.103.3 MySQL 兼容性

`SHOW PRIVILEGES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.103.4 另请参阅

- SHOW GRANTS
- GRANT <privileges>

2.2.104 SHOW [FULL] PROCESSLIST

`SHOW [FULL] PROCESSLIST` 语句列出连接到相同 TiDB 服务器的当前会话。`Info` 列包含查询文本，除非指定可选关键字 `FULL`，否则文本会被截断。

2.2.104.1 语法图

ShowProcesslistStmt:



图 76: ShowProcesslistStmt

OptFull:



图 77: OptFull

2.2.104.2 示例

```
SHOW PROCESSLIST;
```

```
+------+------+-----------------+------+---------+------+------------+------------------+
| Id   | User | Host            | db   | Command | Time | State      | Info             |
+------+------+-----------------+------+---------+------+------------+------------------+
|    5 | root | 127.0.0.1:45970 | test | Query   |    0 | autocommit | SHOW PROCESSLIST |
+------+------+-----------------+------+---------+------+------------+------------------+
1 rows in set (0.00 sec)
```

2.2.104.3 MySQL 兼容性

- TiDB 中的 State 列是非描述性的。在 TiDB 中，将状态表示为单个值更复杂，因为查询是并行执行的，而且每个 Go 线程在任一时刻都有不同的状态。

2.2.104.4 另请参阅

- KILL [TIDB]

2.2.105 SHOW PROFILES

SHOW PROFILES 语句目前只会返回空结果。

2.2.105.1 语法图

ShowStmt:

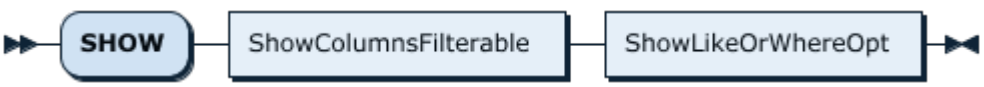


图 78: ShowStmt

2.2.105.2 示例

```
SHOW PROFILES
```

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

2.2.105.3 MySQL 兼容性

该语句仅与 MySQL 兼容，无其他作用。执行 SHOW PROFILES 始终返回空结果。

2.2.106 SHOW PUMP STATUS

SHOW PUMP STATUS 语句用于显示集群中所有 Pump 的状态信息。

2.2.106.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

2.2.106.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.106.3 另请参阅

- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

2.2.107 SHOW SCHEMAS

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

2.2.108 SHOW STATS_HEALTHY

`SHOW STATS_HEALTHY` 语句可以预估统计信息的准确度，也就是健康度。健康度低的表可能会生成次优查询执行计划。

可以通过执行 `ANALYZE` 表命令来改善表的健康度。当表的健康度下降到低于`tidb_auto_analyze_ratio` 时，则会自动执行 `ANALYZE` 命令。

2.2.108.1 语法图

ShowStmt

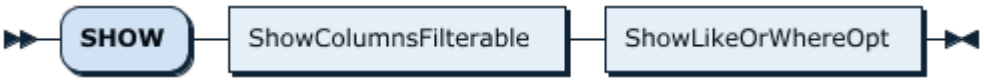


图 79: ShowStmt

ShowTargetFilterable

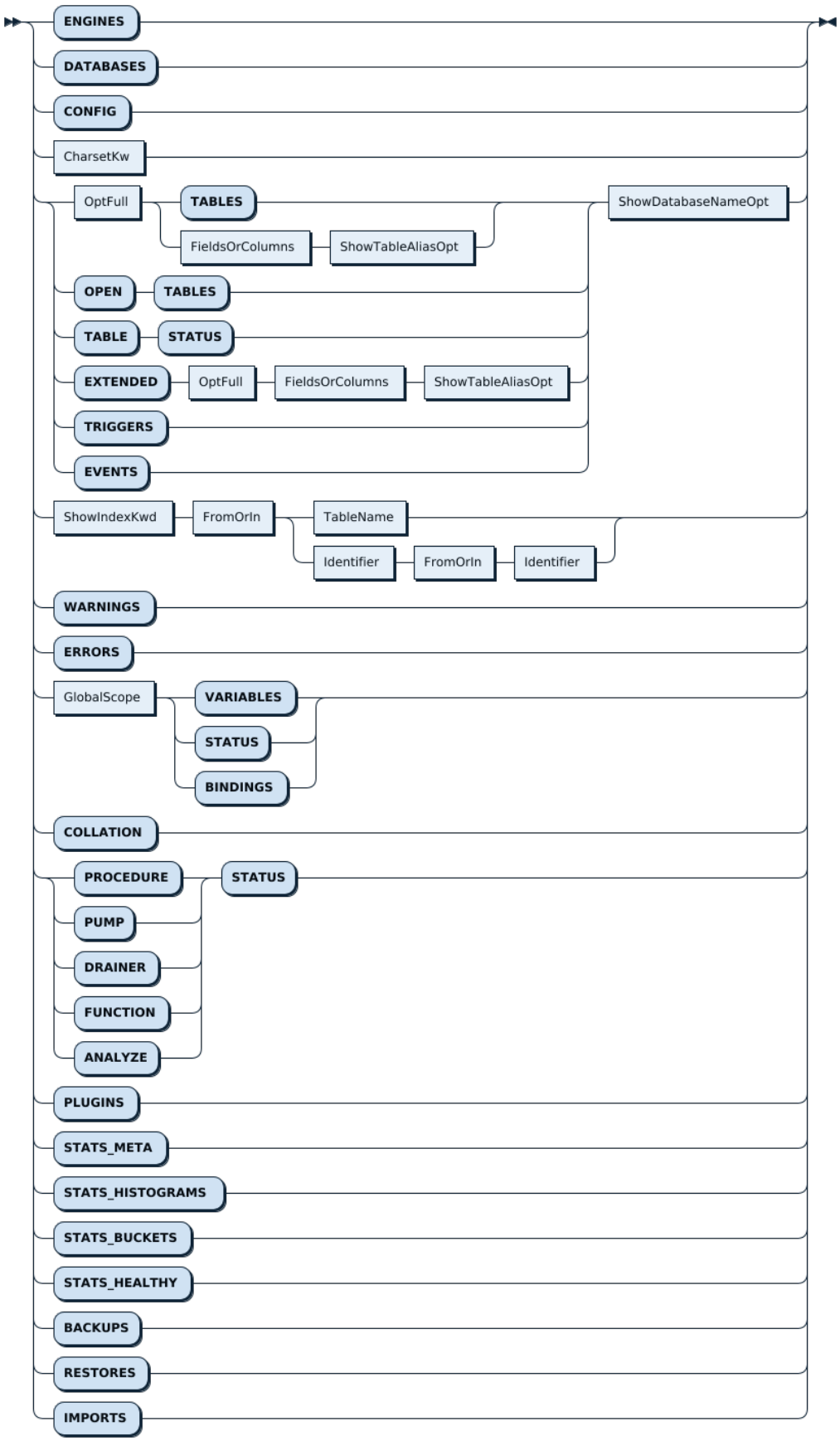


图 80: ShowTargetFilterable

ShowLikeOrWhereOpt

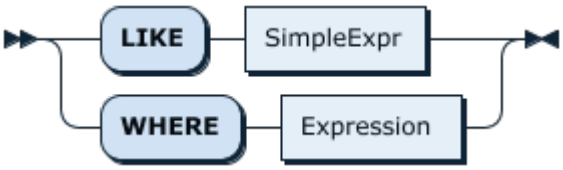


图 81: ShowLikeOrWhereOpt

2.2.108.2 示例

加载示例数据并运行 ANALYZE 命令：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad VARBINARY(255),
 INDEX(b)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
SHOW STATS_HEALTHY; # should be 100% healthy
```

```
SHOW STATS_HEALTHY;
```

```
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |     100 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

执行批量更新来删除大约 30% 的记录，然后检查统计信息的健康度：

```
DELETE FROM t1 WHERE id BETWEEN 101010 AND 201010; # delete about 30% of records
SHOW STATS_HEALTHY;
```

```
SHOW STATS_HEALTHY;
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |      50 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

2.2.108.3 MySQL 兼容性

`SHOW STATS_HEALTHY` 语句是 TiDB 对 MySQL 语法的扩展。

2.2.108.4 另请参阅

- ANALYZE
- 统计信息简介

2.2.109 SHOW STATS_HISTOGRAMS

你可以使用 `SHOW STATS_HISTOGRAMS` 语句查看统计信息中直方图的相关信息。

2.2.109.1 语法图

ShowStmt

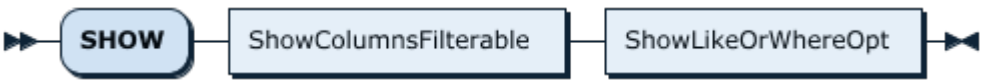


图 82: ShowStmt

ShowTargetFiltertable

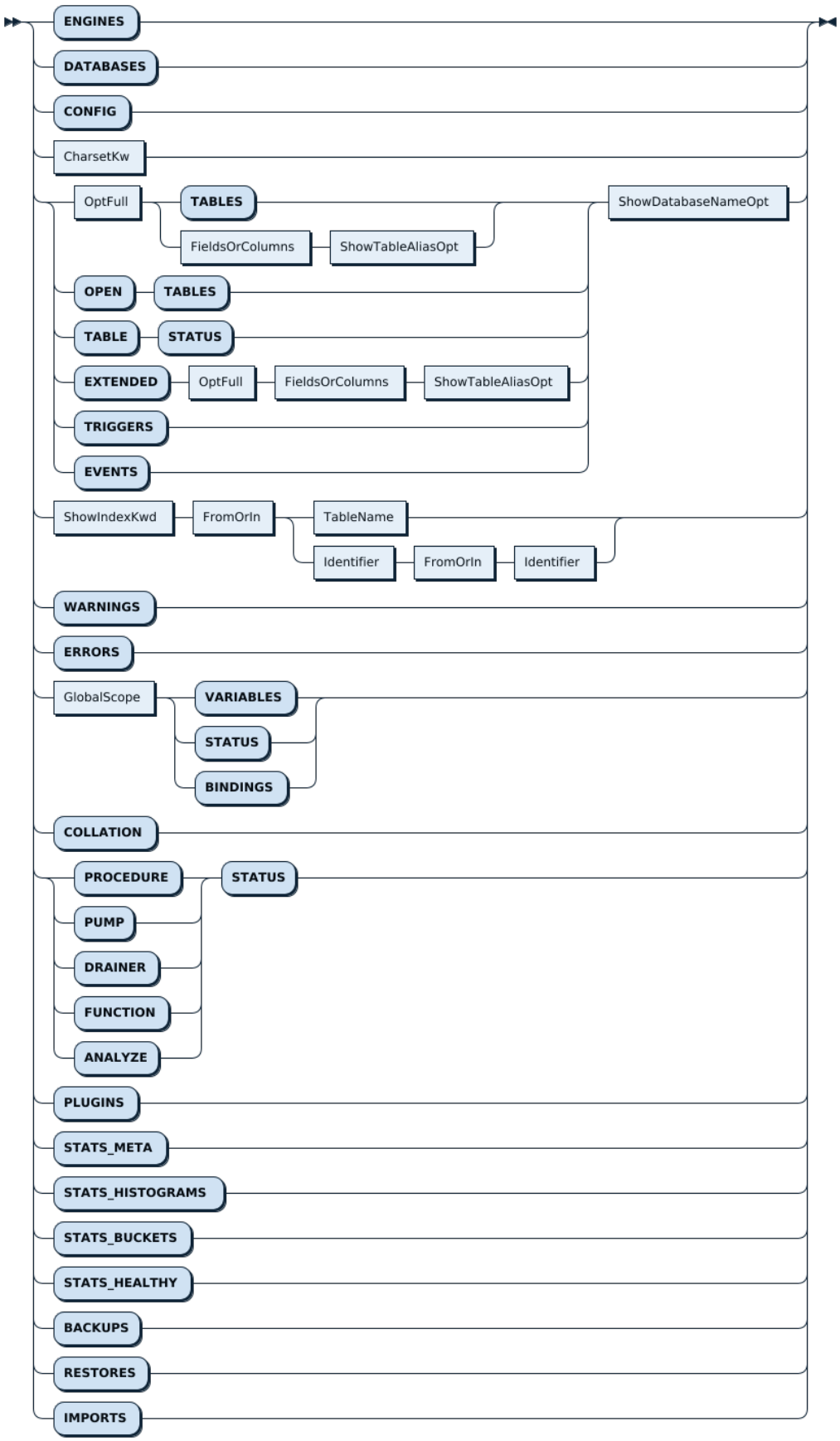


图 83: ShowTargetFilterable

ShowLikeOrWhereOpt

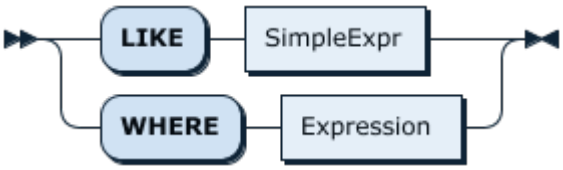


图 84: ShowLikeOrWhereOpt

2.2.109.2 示例

```
show stats_histograms;
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time         |
    ↪ Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| test    | t          |                | a           |        0 | 2020-05-25 19:20:00 |
    ↪              7 |          0 |            1 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |            8 |           0 |
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |         1.67 |           1 |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
3 rows in set (0.00 sec)
```

```
show stats_histograms where table_name = 't2';
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time         |
    ↪ Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |         1.67 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |            8 |           0 |
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+---------------------+-----------+---
    ↪
2 rows in set (0.00 sec)
```

2.2.109.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.109.4 另请参阅

- ANALYZE
- 统计信息介绍

2.2.110 SHOW STATS_META

你可以通过 `SHOW STATS_META` 来查看表的总行数以及修改的行数等信息，可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_META` 会输出 6 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
update_time	更新时间
modify_count	修改的行数
row_count	总行数

> **注意：**
>
> 在 TiDB 根据 DML 语句自动更新总行数以及修改的行数时，`update_time` 也会被更新，因此并不能认为 `update_time` 是最近一次发生 Analyze 的时间。

2.2.110.1 语法图

ShowStmt

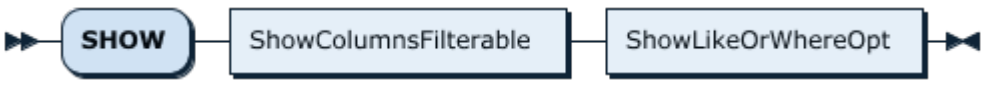


图 85: ShowStmt

ShowTargetFiltertable

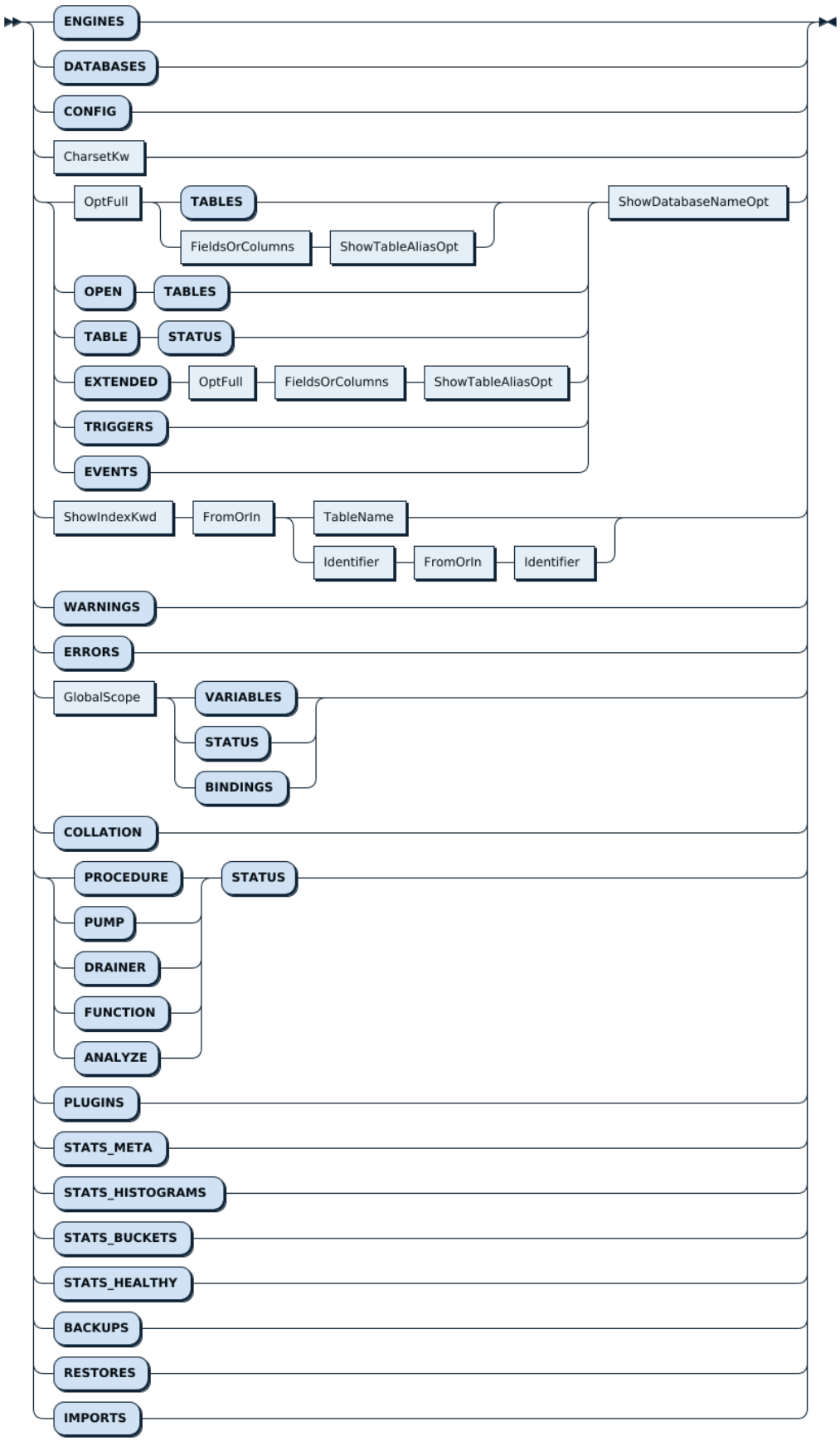


图 86: ShowTargetFilterable

ShowLikeOrWhereOpt

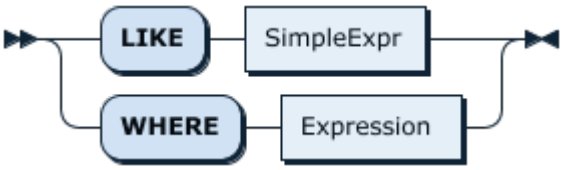


图 87: ShowLikeOrWhereOpt

2.2.110.2 示例

```
show stats_meta;
```

```
+---------+------------+----------------+---------------------+--------------+------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+------------+
| test    | t0         |                | 2020-05-15 16:58:00 |            0 |         0 |
| test    | t1         |                | 2020-05-15 16:58:04 |            0 |         0 |
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |         0 |
| test    | s          |                | 2020-05-22 19:46:43 |            0 |         0 |
| test    | t          |                | 2020-05-25 12:04:21 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+------------+
5 rows in set (0.00 sec)
```

```
show stats_meta where table_name = 't2';
```

```
+---------+------------+----------------+---------------------+--------------+------------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+------------+
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+------------+
1 row in set (0.00 sec)
```

2.2.110.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.110.4 另请参阅

- ANALYZE
- 统计信息介绍

2.2.111 SHOW [GLOBAL|SESSION] STATUS

`SHOW [GLOBAL|SESSION] STATUS` 语句用于提供 MySQL 兼容性，对 TiDB 没有作用。因为 TiDB 使用 Prometheus 和 Grafana 而非 `SHOW STATUS` 来进行集中度量收集。

2.2.111.1 语法图

ShowStmt:

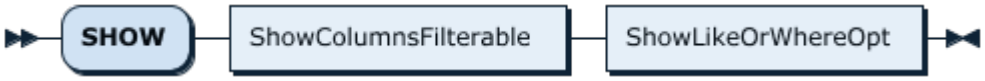


图 88: ShowStmt

ShowTargetFilterable:

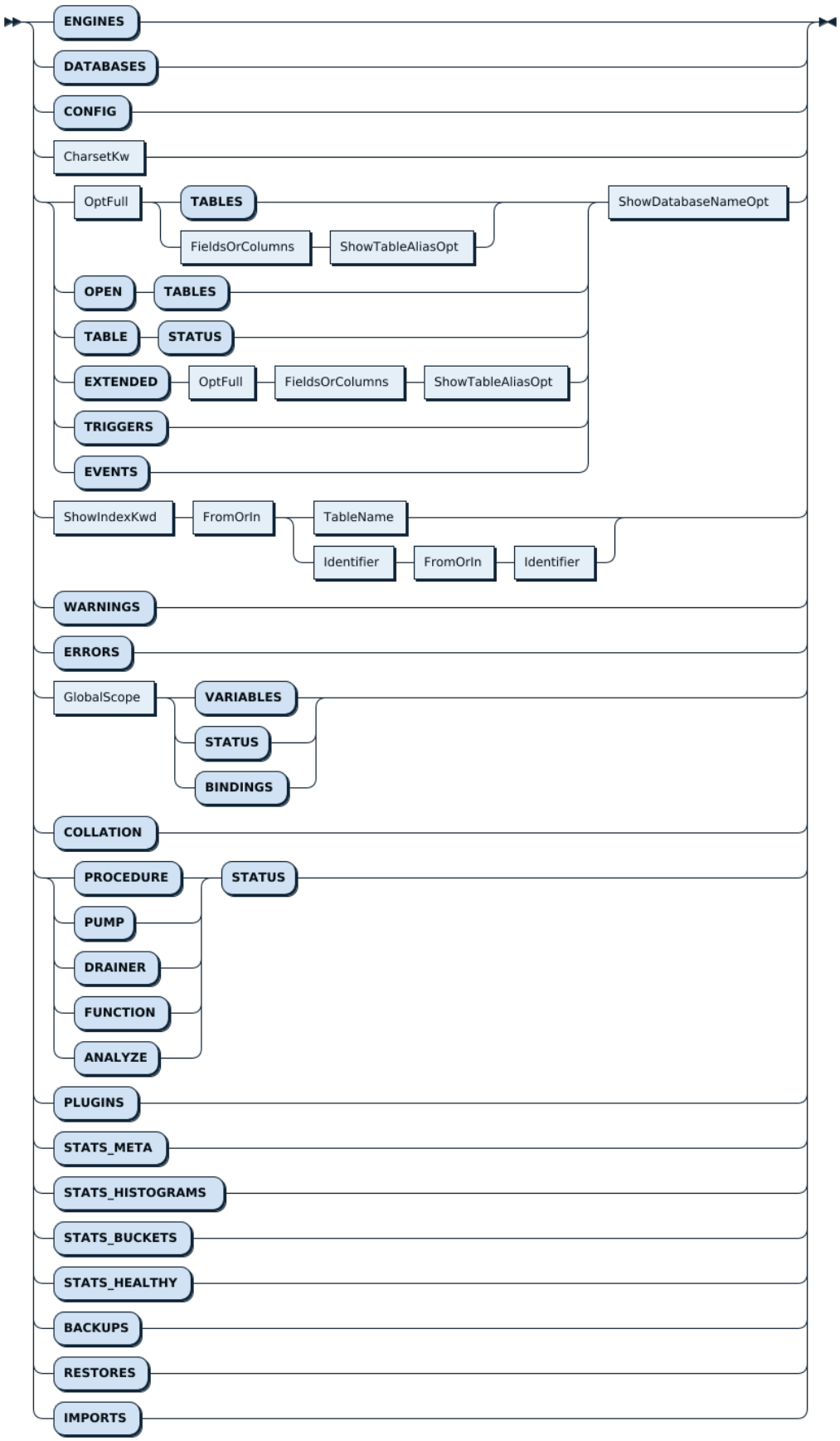


图 89: ShowTargetFilterable

GlobalScope:



图 90: GlobalScope

2.2.111.2 示例

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher_list    |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| Ssl_cipher         |                                      |
+--------------------+--------------------------------------+
6 rows in set (0.01 sec)
```

```
show global status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

2.2.111.3 MySQL 兼容性

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

2.2.111.4 另请参阅

- FLUSH STATUS

2.2.112 SHOW TABLE NEXT_ROW_ID

`SHOW TABLE NEXT_ROW_ID` 语句用于显示用表中某些特殊列的详情，主要包含以下几种类型：

- TiDB 创建的 AUTO_INCREMENT 类型列，即 _tidb_rowid 列
- 用户创建的 AUTO_INCREMENT 类型列
- 用户创建的AUTO_RANDOM 类型列
- 用户创建的SEQUENCE 对象信息

2.2.112.1 语法图

ShowTableNextRowIDStmt:



图 91: ShowTableNextRowIDStmt

TableName:

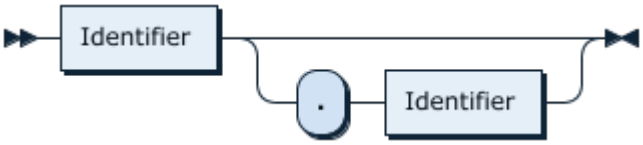


图 92: TableName

2.2.112.2 示例

对于新建的表，由于没有任何的 Row ID 分配，NEXT_GLOBAL_ROW_ID 值为 1

```
create table t(a int);
Query OK, 0 rows affected (0.06 sec)
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------+
| test    | t          | _tidb_rowid |                  1 |
+---------+------------+-------------+--------------------+
1 row in set (0.00 sec)
```

表中写入了数据，负责写入的 TiDB Server 一次性向存储层请求了 30000 个 ID 缓存起来，NEXT_GLOBAL_ROW_ID 值为 30001

```
insert into t values (), (), ();
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------+
| test    | t          | _tidb_rowid |              30001 |
+---------+------------+-------------+--------------------+
1 row in set (0.00 sec)
```

2.2.112.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.112.4 另请参阅

- CREATE TABLE
- AUTO_RANDOM
- CREATE_SEQUENCE

2.2.113 SHOW TABLE REGIONS

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

2.2.113.1 语法

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

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

2.2.113.1.1 语法图

ShowTableRegionStmt:

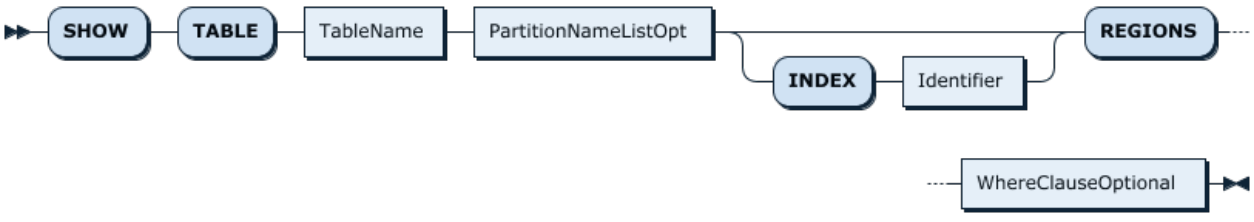


图 93: ShowTableRegionStmt

TableName:

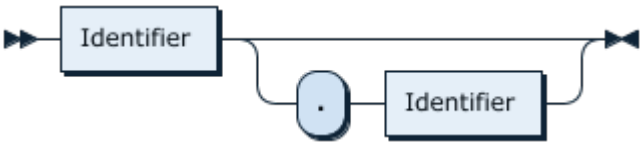


图 94: TableName

PartitionNameListOpt:

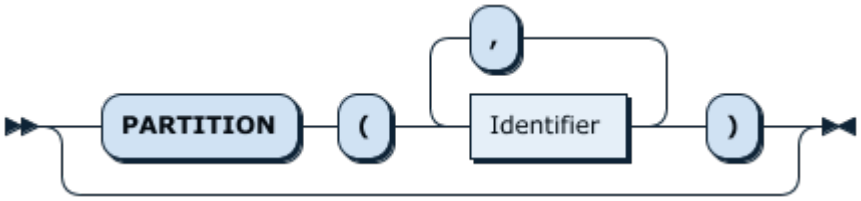


图 95: PartitionNameListOpt

WhereClauseOptional:

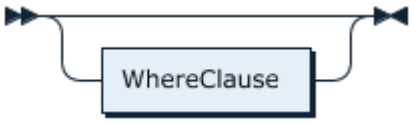


图 96: WhereClauseOptional

WhereClause:



图 97: WhereClause

`SHOW TABLE REGIONS` 会返回如下列：

- `REGION_ID`：Region 的 ID。
- `START_KEY`：Region 的 Start key。
- `END_KEY`：Region 的 End key。
- `LEADER_ID`：Region 的 Leader ID。
- `LEADER_STORE_ID`：Region leader 所在的 store (TiKV) ID。
- `PEERS`：Region 所有副本的 ID。
- `SCATTERING`：Region 是否正在调度中。1 表示正在调度。
- `WRITTEN_BYTES`：估算的 Region 在 1 个心跳周期内的写入数据量大小，单位是 byte。
- `READ_BYTES`：估算的 Region 在 1 个心跳周期内的读数据量大小，单位是 byte。
- `APPROXIMATE_SIZE(MB)`：估算的 Region 的数据量大小，单位是 MB。

- APPROXIMATE_KEYS：估算的 Region 内 Key 的个数。

> **注意：**
>
> WRITTEN_BYTES，READ_BYTES，APPROXIMATE_SIZE(MB)，APPROXIMATE_KEYS 的值是 PD 根据 Region 的心跳汇报信息统计，估算出来的数据，所以不是精确的数据。

2.2.113.2 示例

创建一个示例表，并在若干 Region 中填充足够的数据量：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad1 VARBINARY(1024),
 pad2 VARBINARY(1024),
 pad3 VARBINARY(1024)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
SELECT SLEEP(5);
SHOW TABLE t1 REGIONS;
```

结果显示示例表被切分成多个 Regions。REGION_ID、START_KEY 和 END_KEY 可能不完全匹配：

```
SHOW TABLE t1 REGIONS;
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
|        94 | t_75_         | t_75_r_31717 |        95 |               1 | 95    |          0 |
    ↪             0 |           0 |                  112 |           207465 |
|        96 | t_75_r_31717 | t_75_r_63434 |        97 |               1 | 97    |          0 |
    ↪             0 |           0 |                   97 |                0 |
|         2 | t_75_r_63434 |              |         3 |               1 | 3     |          0 |
    ↪      269323514 |    66346110 |                  245 |           162020 |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
3 rows in set (0.00 sec)
```

解释：

上面 START_KEY 列的值 t_75_r_31717 和 END_KEY 列的值 t_75_r_63434 表示主键在 31717 和 63434 之间的数据存储在该 Region 中。t_75_ 是前缀，表示这是表格 (t) 的 Region，75 是表格的内部 ID。若 START_KEY 或 END_KEY 的一对键值为空，分别表示负无穷大或正无穷大。

TiDB 会根据需要自动重新平衡 Regions。建议使用 SPLIT TABLE REGION 语句手动进行平衡：

```
SPLIT TABLE t1 BETWEEN (31717) AND (63434) REGIONS 2;
```

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  1 |                    1 |
+--------------------+----------------------+
1 row in set (42.34 sec)
SHOW TABLE t1 REGIONS;
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
|        94 | t_75_         | t_75_r_31717 |        95 |               1 | 95    |          0 |
    ↪             0 |           0 |                  112 |           207465 |
```

```
|        98 | t_75_r_31717 | t_75_r_47575 |        99 |                1 | 99    |          0 |
    ↪           1325 |           0 |                    53 |            12052 |
|        96 | t_75_r_47575 | t_75_r_63434 |        97 |                1 | 97    |          0 |
    ↪           1526 |           0 |                    48 |                0 |
|         2 | t_75_r_63434 |              |         3 |                1 | 3     |          0 |
    ↪              0 |    55752049 |                    60 |                0 |
+-----------+--------------+--------------+-----------+------------------+-------+------------+---------------+---
    ↪
4 rows in set (0.00 sec)
```

上面的输出结果显示 Region 96 被切分，并创建一个新的 Region 98。切分操作不会影响表中的其他 Region。输出结果同样证实：

- TOTAL_SPLIT_REGION 表示新切的 Region 数量。以上示例新切了 1 个 Region。
- SCATTER_FINISH_RATIO 表示新切的 Region 的打散成功率，1.0 表示都已经打散了。

更详细的示例如下：

```
show table t regions;
```

```
+-----------+--------------+--------------+-----------+------------------+--------------+------------+-----------
    ↪
| REGION_ID | START_KEY    | END_KEY      | LEADER_ID | LEADER_STORE_ID | PEERS         |
    ↪ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+--------------+--------------+-----------+------------------+--------------+------------+-----------
    ↪
| 102       | t_43_r       | t_43_r_20000 | 118       | 7                | 105, 118, 119 | 0
    ↪           | 0             | 0          | 1                    | 0                |
| 106       | t_43_r_20000 | t_43_r_40000 | 120       | 7                | 107, 108, 120 | 0
    ↪           | 23            | 0          | 1                    | 0                |
| 110       | t_43_r_40000 | t_43_r_60000 | 112       | 9                | 112, 113, 121 | 0
    ↪           | 0             | 0          | 1                    | 0                |
| 114       | t_43_r_60000 | t_43_r_80000 | 122       | 7                | 115, 122, 123 | 0
    ↪           | 35            | 0          | 1                    | 0                |
| 3         | t_43_r_80000 |              | 93        | 8                | 5, 73, 93     | 0
    ↪           | 0             | 0          | 1                    | 0                |
| 98        | t_43_        | t_43_r       | 99        | 1                | 99, 100, 101  | 0
    ↪           | 0             | 0          | 1                    | 0                |
+-----------+--------------+--------------+-----------+------------------+--------------+------------+-----------
    ↪
6 rows in set
```

解释：

- Region 102 的 START_KEY 和 END_KEY 中，t_43 是表数据前缀和 table ID，_r 是表 t record 数据的前缀，索引数据的前缀是 _i，所以 Region 102 的 START_KEY 和 END_KEY 表示用来存储 [-inf, 20000) 之前的 record 数据。其他 Region (106, 110, 114, 3) 的存储范围依次类推。

- Region 98 用来存储索引数据存储。表 t 索引数据的起始 key 是 t_43_i，处于 Region 98 的存储范围内。

查看表 t 在 store 1 上的 region，用 where 条件过滤。

```
show table t regions where leader_store_id =1;
```

```
+-----------+-----------+---------+-----------+----------------+-------------+------------+---------------+----
    ↪
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS        | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+-----------+---------+-----------+----------------+-------------+------------+---------------+----
    ↪
| 98        | t_43_     | t_43_r  | 99        | 1              | 99, 100, 101 | 0          | 0
    ↪             | 0          | 1                    | 0               |
+-----------+-----------+---------+-----------+----------------+-------------+------------+---------------+----
    ↪
```

用 SPLIT TABLE REGION 语法切分索引数据的 Region，下面语法把表 t 的索引 name 数据在 [a,z] 范围内切分成 2 个 Region。

```
split table t index name between ("a") and ("z") regions 2;
```

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
| 2                  | 1.0                  |
+--------------------+----------------------+
1 row in set
```

现在表 t 一共有 7 个 Region，其中 5 个 Region (102, 106, 110, 114, 3) 用来存表 t 的 record 数据，另外 2 个 Region (135, 98) 用来存 name 索引的数据。

```
show table t regions;
```

```
+-----------+----------------------------+----------------------------+-----------+----------------+----------
    ↪
| REGION_ID | START_KEY                  | END_KEY                    | LEADER_ID |
    ↪ LEADER_STORE_ID | PEERS         | SCATTERING | WRITTEN_BYTES | READ_BYTES |
    ↪ APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+----------------------------+----------------------------+-----------+----------------+----------
    ↪
| 102       | t_43_r                     | t_43_r_20000               | 118       | 7
    ↪               | 105, 118, 119 | 0          | 0             | 0          | 1
    ↪                   | 0               |
| 106       | t_43_r_20000               | t_43_r_40000               | 120       | 7
    ↪               | 108, 120, 126 | 0          | 0             | 0          | 1
    ↪                   | 0               |
```

```
| 110        | t_43_r_40000                | t_43_r_60000                | 112        | 9
    ↪                | 112, 113, 121 | 0          | 0             | 0          | 1
    ↪                    | 0               |
| 114        | t_43_r_60000                | t_43_r_80000                | 122        | 7
    ↪                | 115, 122, 123 | 0          | 35            | 0          | 1
    ↪                    | 0               |
| 3          | t_43_r_80000                |                             | 93         | 8
    ↪                | 73, 93, 128   | 0          | 0             | 0          | 1
    ↪                    | 0               |
| 135        | t_43_i_1_                   | t_43_i_1_016d80000000000000 | 139        | 2
    ↪                | 138, 139, 140 | 0          | 35            | 0          | 1
    ↪                    | 0               |
| 98         | t_43_i_1_016d80000000000000 | t_43_r                      | 99         | 1
    ↪                | 99, 100, 101  | 0          | 0             | 0          | 1
    ↪                    | 0               |
+------------+-----------------------------+-----------------------------+------------+-----------------+----------
    ↪
7 rows in set
```

2.2.113.3 MySQL 兼容性

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

2.2.113.4 另请参阅

- SPLIT REGION
- CREATE TABLE

2.2.114 SHOW TABLE STATUS

`SHOW TABLE STATUS` 语句用于显示 TiDB 中表的各种统计信息。如果显示统计信息过期，建议运行`ANALYZE TABLE`。

2.2.114.1 语法图

ShowTableStatusStmt:

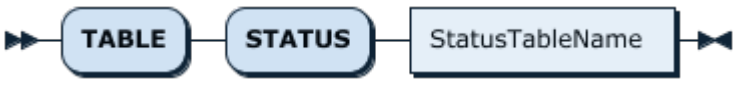


图 98: ShowTableStatusStmt

FromOrIn:

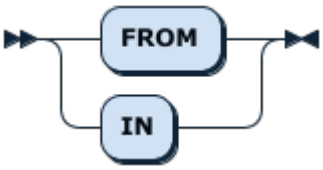


图 99: FromOrIn

StatusTableName:



图 100: StatusTableName

2.2.114.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

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

```
SHOW TABLE STATUS LIKE 't1';
```

```
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 0
 Avg_row_length: 0
    Data_length: 0
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: 30001
    Create_time: 2019-04-19 08:32:06
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_bin
       Checksum:
 Create_options:
```

```
        Comment:
1 row in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW TABLE STATUS LIKE 't1';
```

```
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 5
 Avg_row_length: 16
    Data_length: 80
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: 30001
    Create_time: 2019-04-19 08:32:06
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_bin
       Checksum:
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

2.2.114.3 MySQL 兼容性

`SHOW TABLE STATUS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.114.4 另请参阅

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

2.2.115 SHOW [FULL] TABLES

`SHOW [FULL] TABLES` 语句用于显示当前所选数据库中表和视图的列表。可选关键字 `FULL` 说明表的类型是 `BASE TABLE` 还是 `VIEW`。

若要在不同的数据库中显示表，可使用 `SHOW TABLES IN DatabaseName` 语句。

2.2.115.1 语法图

ShowTablesStmt:



图 101: ShowTablesStmt

OptFull:



图 102: OptFull

ShowDatabaseNameOpt:

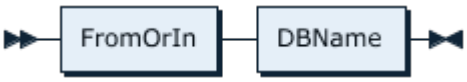


图 103: ShowDatabaseNameOpt

ShowLikeOrWhereOpt:

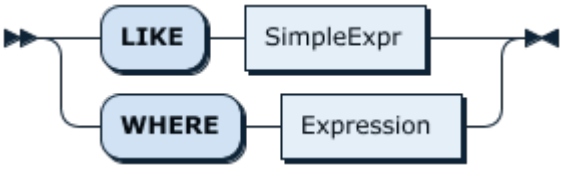


图 104: ShowLikeOrWhereOpt

2.2.115.2 示例

```
mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.12 sec)

mysql> CREATE VIEW v1 AS SELECT 1;
Query OK, 0 rows affected (0.10 sec)

mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
```

```
+----------------+
| t1             |
| v1             |
+----------------+
2 rows in set (0.00 sec)

mysql> SHOW FULL TABLES;
+----------------+------------+
| Tables_in_test | Table_type |
+----------------+------------+
| t1             | BASE TABLE |
| v1             | VIEW       |
+----------------+------------+
2 rows in set (0.00 sec)

mysql> SHOW TABLES IN mysql;
+-------------------------+
| Tables_in_mysql         |
+-------------------------+
| GLOBAL_VARIABLES        |
| bind_info               |
| columns_priv            |
| db                      |
| default_roles           |
| expr_pushdown_blacklist |
| gc_delete_range         |
| gc_delete_range_done    |
| global_priv             |
| help_topic              |
| opt_rule_blacklist      |
| role_edges              |
| stats_buckets           |
| stats_feedback          |
| stats_histograms        |
| stats_meta              |
| stats_top_n             |
| tables_priv             |
| tidb                    |
| user                    |
+-------------------------+
20 rows in set (0.00 sec)
```

2.2.115.3 MySQL 兼容性

`SHOW [FULL] TABLES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.115.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

2.2.116 SHOW [GLOBAL|SESSION] VARIABLES

`SHOW [GLOBAL|SESSION] VARIABLES` 语句用于显示 `GLOBAL` 或 `SESSION` 范围的变量列表。如果未指定范围，则应用默认范围 `SESSION`。

2.2.116.1 语法图

ShowStmt:

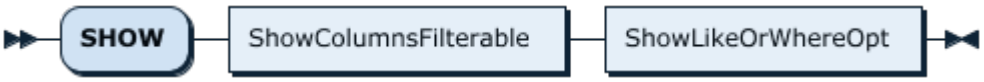


图 105: ShowStmt

ShowTargetFilterable:

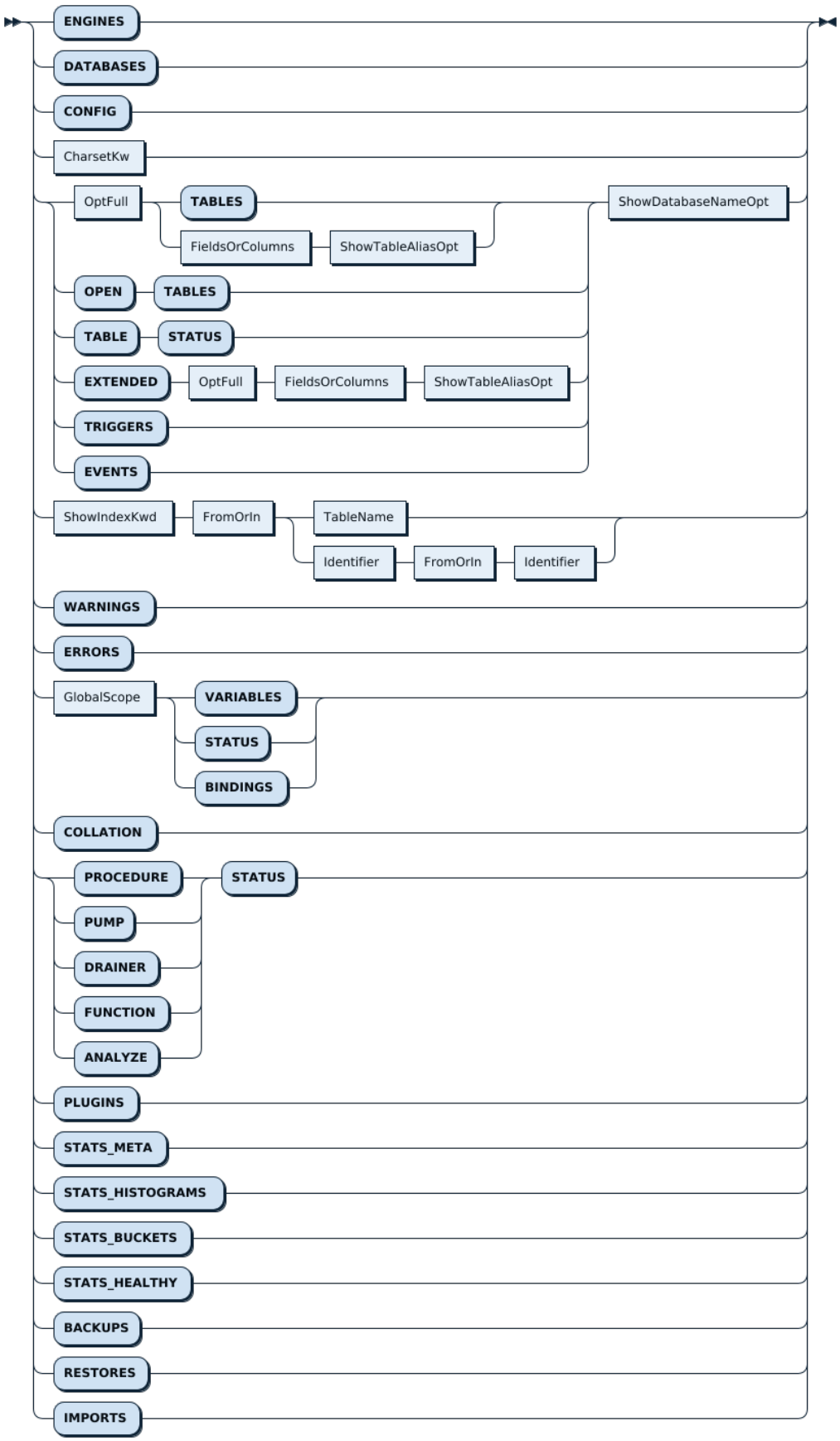


图 106: ShowTargetFilterable

GlobalScope:



图 107: GlobalScope

2.2.116.2 示例

查看 TiDB 定义的专用系统变量，关于这些变量的含义参见系统变量和语法。

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

```
+-----------------------------------+---------------------+
| Variable_name                     | Value               |
+-----------------------------------+---------------------+
| tidb_allow_batch_cop              | 0                   |
| tidb_allow_remove_auto_inc        | 0                   |
| tidb_auto_analyze_end_time        | 23:59 +0000         |
| tidb_auto_analyze_ratio           | 0.5                 |
| tidb_auto_analyze_start_time      | 00:00 +0000         |
| tidb_backoff_lock_fast            | 100                 |
| tidb_backoff_weight               | 2                   |
| tidb_batch_commit                 | 0                   |
| tidb_batch_delete                 | 0                   |
| tidb_batch_insert                 | 0                   |
| tidb_build_stats_concurrency      | 4                   |
| tidb_capture_plan_baselines       | off                 |
| tidb_check_mb4_value_in_utf8      | 1                   |
| tidb_checksum_table_concurrency   | 4                   |
| tidb_config                       |                     |
| tidb_constraint_check_in_place    | 0                   |
| tidb_current_ts                   | 0                   |
| tidb_ddl_error_count_limit        | 512                 |
| tidb_ddl_reorg_batch_size         | 256                 |
| tidb_ddl_reorg_priority           | PRIORITY_LOW        |
| tidb_ddl_reorg_worker_cnt         | 4                   |
| tidb_disable_txn_auto_retry       | 1                   |
| tidb_distsql_scan_concurrency     | 15                  |
| tidb_dml_batch_size               | 20000               |
| tidb_enable_cascades_planner      | 0                   |
| tidb_enable_chunk_rpc             | 1                   |
| tidb_enable_collect_execution_info| 1                   |
```

```
| tidb_enable_fast_analyze            | 0                    |
| tidb_enable_index_merge             | 0                    |
| tidb_enable_noop_functions          | 0                    |
| tidb_enable_radix_join              | 0                    |
| tidb_enable_slow_log                | 1                    |
| tidb_enable_stmt_summary            | 1                    |
| tidb_enable_table_partition          | on                   |
| tidb_enable_vectorized_expression   | 1                    |
| tidb_enable_window_function         | 1                    |
| tidb_evolve_plan_baselines          | off                  |
| tidb_evolve_plan_task_end_time      | 23:59 +0000          |
| tidb_evolve_plan_task_max_time      | 600                  |
| tidb_evolve_plan_task_start_time    | 00:00 +0000          |
| tidb_expensive_query_time_threshold | 60                   |
| tidb_force_priority                 | NO_PRIORITY          |
| tidb_general_log                    | 0                    |
| tidb_hash_join_concurrency          | 5                    |
| tidb_hashagg_final_concurrency      | 4                    |
| tidb_hashagg_partial_concurrency    | 4                    |
| tidb_index_join_batch_size          | 25000                |
| tidb_index_lookup_concurrency       | 4                    |
| tidb_index_lookup_join_concurrency  | 4                    |
| tidb_index_lookup_size              | 20000                |
| tidb_index_serial_scan_concurrency  | 1                    |
| tidb_init_chunk_size                | 32                   |
| tidb_isolation_read_engines         | tikv, tiflash, tidb  |
| tidb_low_resolution_tso             | 0                    |
| tidb_max_chunk_size                 | 1024                 |
| tidb_max_delta_schema_count         | 1024                 |
| tidb_mem_quota_hashjoin             | 34359738368          |
| tidb_mem_quota_indexlookupjoin      | 34359738368          |
| tidb_mem_quota_indexlookupreader    | 34359738368          |
| tidb_mem_quota_mergejoin            | 34359738368          |
| tidb_mem_quota_nestedloopapply      | 34359738368          |
| tidb_mem_quota_query                | 1073741824           |
| tidb_mem_quota_sort                 | 34359738368          |
| tidb_mem_quota_topn                 | 34359738368          |
| tidb_metric_query_range_duration    | 60                   |
| tidb_metric_query_step              | 60                   |
| tidb_opt_agg_push_down              | 0                    |
| tidb_opt_concurrency_factor         | 3                    |
| tidb_opt_copcpu_factor              | 3                    |
| tidb_opt_correlation_exp_factor     | 1                    |
| tidb_opt_correlation_threshold      | 0.9                  |
| tidb_opt_cpu_factor                 | 3                    |
```

```
| tidb_opt_desc_factor                | 3                   |
| tidb_opt_disk_factor                | 1.5                 |
| tidb_opt_distinct_agg_push_down     | 0                   |
| tidb_opt_insubq_to_join_and_agg     | 1                   |
| tidb_opt_join_reorder_threshold     | 0                   |
| tidb_opt_memory_factor              | 0.001               |
| tidb_opt_network_factor             | 1                   |
| tidb_opt_scan_factor                | 1.5                 |
| tidb_opt_seek_factor                | 20                  |
| tidb_opt_write_row_id               | 0                   |
| tidb_optimizer_selectivity_level    | 0                   |
| tidb_pprof_sql_cpu                  | 0                   |
| tidb_projection_concurrency         | 4                   |
| tidb_query_log_max_len              | 4096                |
| tidb_record_plan_in_slow_log        | 1                   |
| tidb_replica_read                   | leader              |
| tidb_retry_limit                    | 10                  |
| tidb_row_format_version             | 2                   |
| tidb_scatter_region                 | 0                   |
| tidb_skip_isolation_level_check     | 0                   |
| tidb_skip_utf8_check                | 0                   |
| tidb_slow_log_threshold             | 300                 |
| tidb_slow_query_file                | tidb-slow.log       |
| tidb_snapshot                       |                     |
| tidb_stmt_summary_history_size      | 24                  |
| tidb_stmt_summary_internal_query    | 0                   |
| tidb_stmt_summary_max_sql_length    | 4096                |
| tidb_stmt_summary_max_stmt_count    | 3000                |
| tidb_stmt_summary_refresh_interval  | 1800                |
| tidb_store_limit                    | 0                   |
| tidb_txn_mode                       |                     |
| tidb_use_plan_baselines             | on                  |
| tidb_wait_split_region_finish       | 1                   |
| tidb_wait_split_region_timeout      | 300                 |
| tidb_window_concurrency             | 4                   |
+-------------------------------------+---------------------+
108 rows in set (0.01 sec)
```

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

```
+---------------+--------+
| Variable_name | Value  |
+---------------+--------+
| time_zone     | SYSTEM |
+---------------+--------+
```

```
1 row in set (0.00 sec)
```

2.2.116.3 MySQL 兼容性

`SHOW [GLOBAL|SESSION] VARIABLES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.116.4 另请参阅

- `SET [GLOBAL|SESSION]`

2.2.117 SHOW WARNINGS

`SHOW WARNINGS` 语句用于显示当前客户端连接中已执行语句的报错列表。与在 MySQL 中一样，`sql_mode` 极大地影响哪些语句会导致错误与警告。

2.2.117.1 语法图

ShowWarningsStmt:



图 108: ShowWarningsStmt

2.2.117.2 示例

```
CREATE TABLE t1 (a INT UNSIGNED);
```

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

```
INSERT INTO t1 VALUES (0);
```

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

```
SELECT 1/a FROM t1;
```

```
+------+
| 1/a  |
+------+
| NULL |
+------+
1 row in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+---------+------+---------------+
| Level   | Code | Message       |
+---------+------+---------------+
| Warning | 1365 | Division by 0 |
+---------+------+---------------+
1 row in set (0.00 sec)
```

```
INSERT INTO t1 VALUES (-1);
```

```
ERROR 1264 (22003): Out of range value for column 'a' at row 1
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    0 |
+------+
1 row in set (0.00 sec)
```

```
SET sql_mode='';
```

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

```
INSERT INTO t1 VALUES (-1);
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
SHOW WARNINGS;
```

```
+---------+------+---------------------------+
| Level   | Code | Message                   |
+---------+------+---------------------------+
| Warning | 1690 | constant -1 overflows int |
+---------+------+---------------------------+
1 row in set (0.00 sec)
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    0 |
```

```
|    0 |
+------+
2 rows in set (0.00 sec)
```

2.2.117.3 MySQL 兼容性

SHOW WARNINGS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.117.4 另请参阅

- SHOW ERRORS

2.2.118 SHUTDOWN

SHUTDOWN 语句用于在 TiDB 中执行停机操作。执行 SHUTDOWN 语句需要用户拥有 SHUTDOWN privilege。

2.2.118.1 语法图

Statement:



图 109: Statement

2.2.118.2 示例

```
SHUTDOWN;
```

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

2.2.118.3 MySQL 兼容性

> **注意：**
>
> 由于 TiDB 是分布式数据库，因此 TiDB 中的停机操作停止的是客户端连接的 TiDB 实例，而不是整个 TiDB 集群。

SHUTDOWN 语句与 MySQL 不完全兼容。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

2.2.119 Split Region 使用文档

在 TiDB 中新建一个表后，默认会单独切分出 1 个 Region 来存储这个表的数据，这个默认行为由配置文件中的 `split-table` 控制。当这个 Region 中的数据超过默认 Region 大小限制后，这个 Region 会开始分裂成 2 个 Region。

上述情况中，如果在新建的表上发生大批量写入，则会造成热点，因为开始只有一个 Region，所有的写请求都发生在该 Region 所在的那台 TiKV 上。

为解决上述场景中的热点问题，TiDB 引入了预切分 Region 的功能，即可以根据指定的参数，预先为某个表切分出多个 Region，并打散到各个 TiKV 上去。

2.2.119.1 语法图

SplitRegionStmt:



图 110: SplitRegionStmt

SplitSyntaxOption:

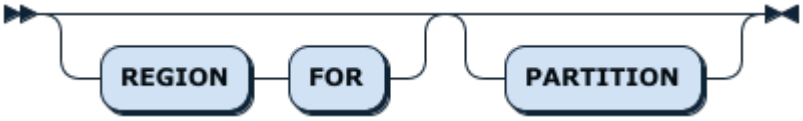


图 111: SplitSyntaxOption

TableName:

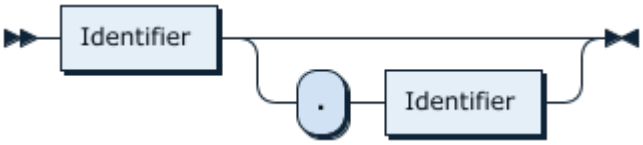


图 112: TableName

PartitionNameListOpt:

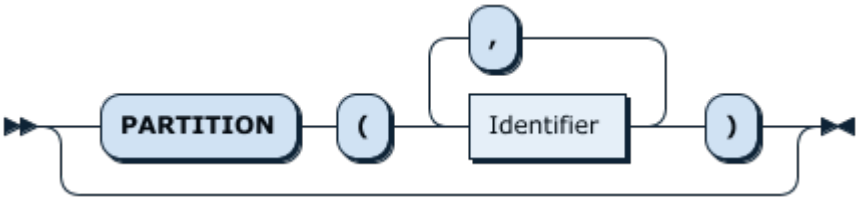


图 113: PartitionNameListOpt

SplitOption:

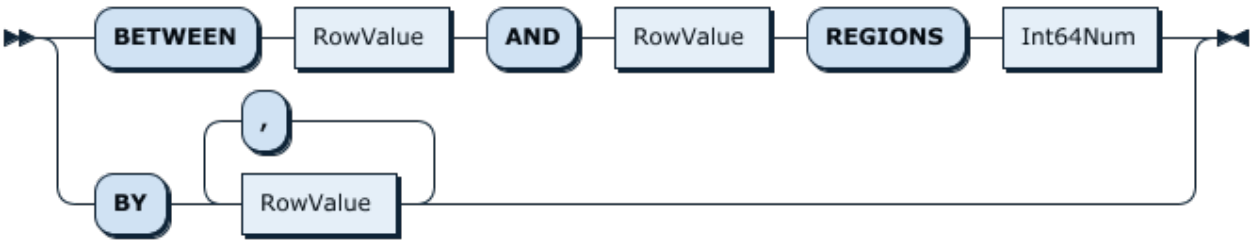


图 114: SplitOption

RowValue:



图 115: RowValue

Int64Num:

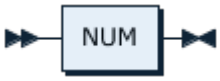


图 116: Int64Num

2.2.119.2 Split Region 的使用

Split Region 有 2 种不同的语法，具体如下：

- 均匀切分的语法：

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

`BETWEEN lower_value AND upper_value REGIONS region_num` 语法是通过指定数据的上、下边界和 Region 数量，然后在上、下边界之间均匀切分出 `region_num` 个 Region。

- 不均匀切分的语法：

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

`BY value_list…` 语法将手动指定一系列的点，然后根据这些指定的点切分 Region，适用于数据不均匀分布的场景。

`SPLIT` 语句的返回结果示例如下：

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
| 4                  | 1.0                  |
+--------------------+----------------------+
```

- TOTAL_SPLIT_REGION：表示新增预切分的 Region 数量。
- SCATTER_FINISH_RATIO：表示新增预切分 Region 中，打散完成的比率。如 1.0 表示全部完成。0.5表示只有一半的 Region 已经打散完成，剩下的还在打散过程中。

> 注意：
>
> 以下会话变量会影响 SPLIT 语句的行为，需要特别注意：
>
> - tidb_wait_split_region_finish：打散 Region 的时间可能较长，由 PD 调度以及 TiKV 的负载情况所决定。这个变量用来设置在执行 SPLIT REGION 语句时，是否同步等待所有 Region 都打散完成后再返回结果给客户端。默认 1 代表等待打散完成后再返回结果。0 代表不等待 Region 打散完成就返回结果。
> - tidb_wait_split_region_timeout：这个变量用来设置 SPLIT REGION 语句的执行超时时间，单位是秒，默认值是 300 秒，如果超时还未完成 Split 操作，就返回一个超时错误。

2.2.119.2.1 Split Table Region

表中行数据的 key 由 table_id 和 row_id 编码组成，格式如下：

```
t[table_id]_r[row_id]
```

例如，当 table_id 是 22，row_id 是 11 时：

```
t22_r11
```

同一表中行数据的 table_id 是一样的，但 row_id 肯定不一样，所以可以根据 row_id 来切分 Region。

均匀切分

由于 row_id 是整数，所以根据指定的 lower_value、upper_value 以及 region_num，可以推算出需要切分的 key。TiDB 先计算 step(step = (upper_value - lower_value)/region_num)，然后在 lower_value 和 upper_value 之间每隔 step 区间切一次，最终切出 region_num 个 Region。

例如，对于表 t，如果想要从 minInt64~maxInt64 之间均匀切割出 16 个 Region，可以用以下语句：

```
SPLIT TABLE t BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS 16;
```

该语句会把表 t 从 minInt64 到 maxInt64 之间均匀切割出 16 个 Region。如果已知主键的范围没有这么大，比如只会在 0~1000000000 之间，那可以用 0 和 1000000000 分别代替上面的 minInt64 和 maxInt64 来切分 Region。

```
SPLIT TABLE t BETWEEN (0) AND (1000000000) REGIONS 16;
```

不均匀切分

如果已知数据不是均匀分布的，比如想要 -inf ~ 10000 切一个 Region，10000 ~ 90000 切一个 Region，90000 ~ +inf 切一个 Region，可以通过手动指定点来切分 Region，示例如下：

```
SPLIT TABLE t BY (10000), (90000);
```

2.2.119.2.2 Split Index Region

表中索引数据的 key 由 table_id、index_id 以及索引列的值编码组成，格式如下：

```
t[table_id]_i[index_id][index_value]
```

例如，当 table_id 是 22，index_id 是 5，index_value 是 abc 时：

```
t22_i5abc
```

同一表中同一索引数据的 table_id 和 index_id 是一样的，所以要根据 index_value 切分索引 Region。

均匀切分

索引均匀切分与行数据均匀切分的原理一样，只是计算 step 的值较为复杂，因为 index_value 可能不是整数。

upper 和 lower 的值会先编码成 byte 数组，去掉 lower 和 upper byte 数组的最长公共前缀后，从 lower 和 upper 各取前 8 字节转成 uint64，再计算 step = (upper - lower)/num。计算出 step 后再将 step 编码成 byte 数组，添加到之前 upper和 lower的最长公共前缀后面组成一个 key 后去做切分。示例如下：

如果索引 idx 的列也是整数类型，可以用如下 SQL 语句切分索引数据：

```
SPLIT TABLE t INDEX idx BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS 16;
```

该语句会把表 t 中 idx 索引数据 Region 从 minInt64 到 maxInt64 之间均匀切割出 16 个 Region。

如果索引 idx1 的列是 varchar 类型，希望根据前缀字母来切分索引数据：

```
SPLIT TABLE t INDEX idx1 BETWEEN ("a") AND ("z") REGIONS 25;
```

该语句会把表 t 中 idx1 索引数据的 Region 从 a~z 切成 25 个 Region，region1 的范围是 [minIndexValue, b)，region2 的范围是 [b, c)，……，region25 的范围是 [y, maxIndexValue)。对于 idx1 索引以 a 为前缀的数据都会写到 region1，以 b 为前缀的索引数据都会写到 region2，以此类推。

上面的切分方法，以 y 和 z 前缀的索引数据都会写到 region 25, 因为 z 并不是一个上界，真正的上界是 z 在 ASCII 码中的下一位 {，所以更准确的切分方法如下：

```
SPLIT TABLE t INDEX idx1 BETWEEN ("a") AND ("{") REGIONS 26;
```

该语句会把表 t 中 idx1 索引数据的 Region 从 a~{ 切成 26 个 Region，region1 的范围是 [minIndexValue, b)，region2 的范围是 [b, c)，……，region25 的范围是 [y,z)，region26 的范围是 [z, maxIndexValue)。

如果索引 idx2 的列是 timestamp/datetime 等时间类型，希望根据时间区间，按年为间隔切分索引数据，示例如下：

```
SPLIT TABLE t INDEX idx2 BETWEEN ("2010-01-01 00:00:00") AND ("2020-01-01 00:00:00") REGIONS 10;
```

该语句会把表 t 中 idx2 的索引数据 Region 从 2010-01-01 00:00:00 到 2020-01-01 00:00:00 切成 10 个 Region。region1 的范围是从 [minIndexValue, 2011-01-01 00:00:00)，region2 的范围是 [2011-01-01 00:00:00, 2012-01-01 00:00:00)……

如果希望按照天为间隔切分索引，示例如下：

```
SPLIT TABLE t INDEX idx2 BETWEEN ("2020-06-01 00:00:00") AND ("2020-07-01 00:00:00") REGIONS 30;
```

该语句会将表 t 中 idx2 索引位于 2020 年 6 月份的数据按天为间隔切分成 30 个 Region。

其他索引列类型的切分方法也是类似的。

对于联合索引的数据 Region 切分，唯一不同的是可以指定多个 column 的值。

比如索引 idx3 (a, b) 包含 2 列，a 是 timestamp，b 是 int。如果只想根据 a 列做时间范围的切分，可以用切分单列时间索引的 SQL 语句来切分，lower_value 和 upper_velue 中不指定 b 列的值即可。

```
SPLIT TABLE t INDEX idx3 BETWEEN ("2010-01-01 00:00:00") AND ("2020-01-01 00:00:00") REGIONS 10;
```

如果想在时间相同的情况下，根据 b 列再做一次切分，在切分时指定 b 列的值即可。

```
SPLIT TABLE t INDEX idx3 BETWEEN ("2010-01-01 00:00:00", "a") AND ("2010-01-01 00:00:00", "z")
    ↪ REGIONS 10;
```

该语句在 a 列时间前缀相同的情况下，根据 b 列的值从 a~z 切了 10 个 Region。如果指定的 a 列的值不相同，那么可能不会用到 b 列的值。

不均匀切分

索引数据也可以根据用户指定的索引值来做切分。

假如有 idx4 (a,b)，其中 a 列是 varchar 类型，b 列是 timestamp 类型。

```
SPLIT TABLE t1 INDEX idx4 BY ("a", "2000-01-01 00:00:01"), ("b", "2019-04-17 14:26:19"), ("c", ""
    ↪ );
```

该语句指定了 3 个值，会切分出 4 个 Region，每个 Region 的范围如下。

```
region1  [ minIndexValue               , ("a", "2000-01-01 00:00:01"))
region2  [("a", "2000-01-01 00:00:01") , ("b", "2019-04-17 14:26:19"))
region3  [("b", "2019-04-17 14:26:19") , ("c", "")                   )
region4  [("c", "")                    , maxIndexValue               )
```

2.2.119.2.3 Split 分区表的 Region

预切分分区表的 Region 在使用上和普通表一样，差别是会为每一个 partition 都做相同的切分。

- 均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE t [PARTITION] [(partition_name_list...)] [INDEX index_name] BETWEEN
    ↪ (lower_value) AND (upper_value) REGIONS region_num
```

- 不均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE table_name [PARTITION (partition_name_list...)] [INDEX index_name]
    ↪ BY (value_list) [, (value_list)] ...
```

Split 分区表的 Region 示例

1. 首先创建一个分区表。如果你要建一个 Hash 分区表，分成 2 个 partition，示例语句如下：

```
create table t (a int,b int,index idx(a)) partition by hash(a) partitions 2;
```

此时建完表后会为每个 partition 都单独 split 一个 Region，用 SHOW TABLE REGIONS 语法查看该表的 Region 如下：

```
show table t regions;
```

```
+--
    ↪ ---------+-----------+---------+-----------+----------------+------------------+------------+------
    ↪
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS             |
    ↪ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+-----------+---------+-----------+----------------+------------------+------------+------
    ↪
| 1978      | t_1400_   | t_1401_ | 1979      | 4              | 1979, 1980, 1981 | 0
    ↪           | 0             | 0          | 1                  | 0                |
| 6         | t_1401_   |         | 17        | 4              | 17, 18, 21       | 0
    ↪           | 223           | 0          | 1                  | 0                |
+--
    ↪ ---------+-----------+---------+-----------+----------------+------------------+------------+------
    ↪
```

2. 用 SPLIT 语法为每个 partition 切分 Region。如果你要将各个 partition 的 [0,10000] 范围内的数据切分成 4 个 Region，示例语句如下：

```
split partition table t between (0) and (10000) regions 4;
```

其中，0 和 10000 分别代表你想要打散的热点数据对应的上、下边界的 row_id。

> 注意：
> 此示例仅适用于数据热点均匀分布的场景。如果热点数据在你指定的数据范围内是不均匀分布的，请参考Split 分区表的 Region 中不均匀切分的语法。

3. 用 SHOW TABLE REGIONS 语法查看该表的 Region。如下会发现该表现在一共有 10 个 Region，每个 partition 分别有 5 个 Region，其中 4 个 Region 是表的行数据，1 个 Region 是表的索引数据。

```
show table t regions;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+----------------+--------------------+---------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS
    ↪            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    ↪ APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+----------------+--------------------+---------
    ↪
| 1998      | t_1400_r      | t_1400_r_2500 | 2001      | 5               | 2000, 2001, 2015
    ↪  | 0          | 132           | 0          | 1                    | 0
    ↪                |
| 2006      | t_1400_r_2500 | t_1400_r_5000 | 2016      | 1               | 2007, 2016, 2017
    ↪  | 0          | 35            | 0          | 1                    | 0
    ↪                |
| 2010      | t_1400_r_5000 | t_1400_r_7500 | 2012      | 2               | 2011, 2012, 2013
    ↪  | 0          | 35            | 0          | 1                    | 0
    ↪                |
| 1978      | t_1400_r_7500 | t_1401_       | 1979      | 4               | 1979, 1980, 1981
    ↪  | 0          | 621           | 0          | 1                    | 0
    ↪                |
| 1982      | t_1400_       | t_1400_r      | 2014      | 3               | 1983, 1984, 2014
    ↪  | 0          | 35            | 0          | 1                    | 0
    ↪                |
| 1990      | t_1401_r      | t_1401_r_2500 | 1992      | 2               | 1991, 1992, 2020
    ↪  | 0          | 120           | 0          | 1                    | 0
    ↪                |
| 1994      | t_1401_r_2500 | t_1401_r_5000 | 1997      | 5               | 1996, 1997, 2021
    ↪  | 0          | 129           | 0          | 1                    | 0
    ↪                |
| 2002      | t_1401_r_5000 | t_1401_r_7500 | 2003      | 4               | 2003, 2023, 2022
    ↪  | 0          | 141           | 0          | 1                    | 0
    ↪                |
| 6         | t_1401_r_7500 |               | 17        | 4               | 17, 18, 21
    ↪       | 0          | 601           | 0          | 1                    | 0
    ↪                |
| 1986      | t_1401_       | t_1401_r      | 1989      | 5               | 1989, 2018, 2019
    ↪  | 0          | 123           | 0          | 1                    | 0
    ↪                |
+--
    ↪ ---------+--------------+--------------+-----------+----------------+--------------------+---------
```

```
    ↪
```

4. 如果你要给每个分区的索引切分 Region，如将索引 idx 的 [1000,10000] 范围切分成 2 个 Region，示例语句如下：

```
split partition table t index idx between (1000) and (10000) regions 2;
```

Split 单个分区的 Region 示例

可以单独指定要切分的 partition，示例如下：

1. 首先创建一个分区表。如果你要建一个 Range 分区表，分成 3 个 partition，示例语句如下：

```
create table t ( a int, b int, index idx(b)) partition by range( a ) (
    partition p1 values less than (10000),
    partition p2 values less than (20000),
    partition p3 values less than (MAXVALUE) );
```

2. 如果你要将 p1 分区的 [0,10000] 范围内的数据预切分 2 个 Region，示例语句如下：

```
split partition table t partition (p1) between (0) and (10000) regions 2;
```

3. 如果你要将 p2 分区的 [10000,20000] 范围内的数据预切分 2 个 Region，示例语句如下：

```
split partition table t partition (p2) between (10000) and (20000) regions 2;
```

4. 用 SHOW TABLE REGIONS 语法查看该表的 Region 如下：

```
show table t regions;
```

```
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
| REGION_ID | START_KEY      | END_KEY        | LEADER_ID | LEADER_STORE_ID | PEERS
    ↪            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    ↪ APPROXIMATE_KEYS |
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
| 2040      | t_1406_        | t_1406_r_5000  | 2045      | 3               | 2043, 2045,
    ↪ 2044 | 0           | 0              | 0          | 1                 | 0
    ↪                |
| 2032      | t_1406_r_5000  | t_1407_        | 2033      | 4               | 2033, 2034,
    ↪ 2035 | 0           | 0              | 0          | 1                 | 0
    ↪                |
| 2046      | t_1407_        | t_1407_r_15000 | 2048      | 2               | 2047, 2048,
    ↪ 2050 | 0           | 35             | 0          | 1                 | 0
    ↪                |
```

```
| 2036       | t_1407_r_15000 | t_1408_         | 2037       | 4                 | 2037, 2038,
    ↪ 2039 | 0           | 0              | 0           | 1                       | 0
    ↪                  |
| 6          | t_1408_         |                  | 17         | 4                 | 17, 18, 21
    ↪        | 0           | 214            | 0           | 1                       | 0
    ↪                  |
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+-------------------+-------
    ↪
```

5. 如果你要将 p1 和 p2 分区的索引 idx 的 [0,20000] 范围预切分 2 个 Region，示例语句如下：

```
split partition table t partition (p1,p2) index idx between (0) and (20000) regions 2;
```

2.2.119.3 pre_split_regions

使用带有 SHARD_ROW_ID_BITS 的表时，如果希望建表时就均匀切分 Region，可以考虑配合 PRE_SPLIT_REGIONS 一起使用，用来在建表成功后就开始预均匀切分 2^(PRE_SPLIT_REGIONS) 个 Region。

> **注意：**
>
> PRE_SPLIT_REGIONS 必须小于等于 SHARD_ROW_ID_BITS。

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

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

2.2.119.3.1 pre_split_regions 示例

```
create table t (a int, b int,index idx1(a)) shard_row_id_bits = 4 pre_split_regions=2;
```

该语句在建表后，会对这个表 t 预切分出 4 + 1 个 Region。4 (2^2) 个 Region 是用来存 table 的行数据的，1 个 Region 是用来存 idx1 索引的数据。

4 个 table Region 的范围区间如下：

```
region1:   [ -inf      ,  1<<61 )
region2:   [ 1<<61     ,  2<<61 )
region3:   [ 2<<61     ,  3<<61 )
region4:   [ 3<<61     ,  +inf  )
```

2.2.119.4 注意事项

Split Region 语句切分的 Region 会受到 PD 中 Region merge 调度的控制，需要动态修改 Region merge 相关的配置项，避免新切分的 Region 不久后又被 PD 重新合并的情况。

2.2.119.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

2.2.119.6 另请参阅

- SHOW TABLE REGIONS
- Session 变量：tidb_scatter_region，tidb_wait_split_region_finish 和 tidb_wait_split_region_timeout.

2.2.120 START TRANSACTION

START TRANSACTION 语句用于在 TiDB 内部启动新事务。它类似于语句 BEGIN。

在没有 START TRANSACTION 语句的情况下，每个语句都会在各自的事务中自动提交，这样可确保 MySQL 兼容性。

2.2.120.1 语法图

BeginTransactionStmt:

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( ( 'WITH' 'TIMESTAMP' 'BOUND'
    ↪ TimestampBound )? | AsOfClause ) ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' | 'WITH' 'CAUSAL' '
    ↪ CONSISTENCY' 'ONLY' )?

AsOfClause ::=
    ( 'AS' 'OF' 'TIMESTAMP' Expression)
```

2.2.120.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
START TRANSACTION;
```

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

```
INSERT INTO t1 VALUES (1);
```

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

```
COMMIT;
```

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

2.2.120.3 MySQL 兼容性

- 执行 START TRANSACTION 在 TiDB 中开启事务并立即生成快照。而在 MySQL 中，执行 START TRANSACTION 会开启事务但不会立即生成快照。TiDB 中的 START TRANSACTION 等同于 MySQL 中的 START TRANSACTION WITH CONSISTENT SNAPSHOT。
- 为与 MySQL 兼容，TiDB 会解析 START TRANSACTION READ ONLY 语句，但解析后 TiDB 仍允许写入操作。

2.2.120.4 另请参阅

- COMMIT
- ROLLBACK
- BEGIN
- START TRANSACTION WITH CAUSAL CONSISTENCY ONLY

2.2.121 TABLE

当不需要聚合或复杂的过滤操作时，可以用 TABLE 语句代替 SELECT * FROM。

2.2.121.1 语法图

```
TableStmt ::=
    "TABLE" Table ( "ORDER BY" Column )? ( "LIMIT" NUM )?
```

2.2.121.2 示例

```
CREATE TABLE t1(id INT PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.31 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3);
```

```
Query OK, 3 rows affected (0.06 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
TABLE t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
+----+
3 rows in set (0.01 sec)
```

```
TABLE t1 ORDER BY id DESC;
```

```
+----+
| id |
+----+
|  3 |
|  2 |
|  1 |
+----+
3 rows in set (0.01 sec)
```

```
TABLE t1 LIMIT 1;
```

```
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

2.2.121.3 MySQL 兼容性

TABLE 语句是从 MySQL 8.0.19 开始引入的。

2.2.121.4 另请参阅

- SELECT
- TABLE statements in MySQL

2.2.122 TRACE

TRACE 语句用于提供查询执行的详细信息，可通过 TiDB 服务器状态端口所公开的图形界面进行查看。

2.2.122.1 语法图

TraceStmt:

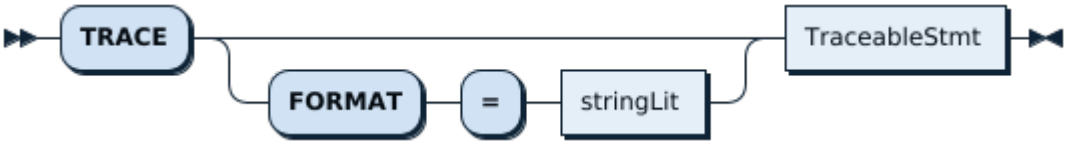


图 117: TraceStmt

TraceableStmt:

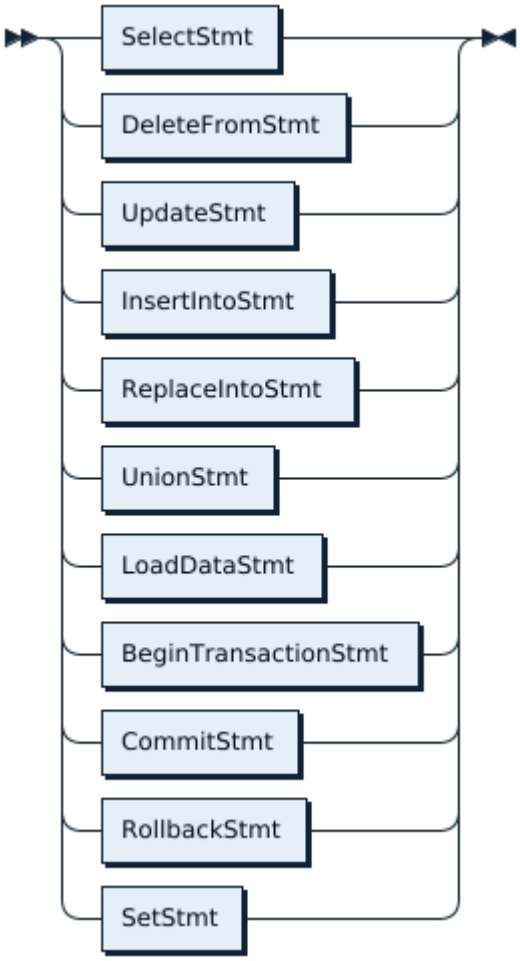


图 118: TraceableStmt

2.2.122.2 示例

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

```
+-----------------------------------------+-----------------+------------+
| operation                               | startTS         | duration   |
```

```
+-----------------------------------------+-----------------+------------+
| trace                                   | 17:03:31.938237 | 886.086µs  |
|   ├─session.Execute                     | 17:03:31.938247 | 507.812µs  |
|   │ ├─session.ParseSQL                  | 17:03:31.938254 | 22.504µs   |
|   │ ├─executor.Compile                  | 17:03:31.938321 | 278.931µs  |
|   │ │ └─session.getTxnFuture            | 17:03:31.938337 | 1.515µs    |
|   │ └─session.runStmt                   | 17:03:31.938613 | 109.578µs  |
|   │   ├─TableReaderExecutor.Open        | 17:03:31.938645 | 50.657µs   |
|   │   │ └─distsql.Select                | 17:03:31.938666 | 21.066µs   |
|   │   │   └─RPCClient.SendRequest       | 17:03:31.938799 | 158.411µs  |
|   │   └─session.CommitTxn               | 17:03:31.938705 | 12.06µs    |
|   │     └─session.doCommitWitRetry      | 17:03:31.938709 | 2.437µs    |
|   ├─*executor.TableReaderExecutor.Next  | 17:03:31.938781 | 224.327µs  |
|   └─*executor.TableReaderExecutor.Next  | 17:03:31.939019 | 6.266µs    |
+-----------------------------------------+-----------------+------------+
13 rows in set (0.00 sec)
```

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

可将 JSON 格式的跟踪文件粘贴到跟踪查看器中。查看器可通过 TiDB 状态端口访问：

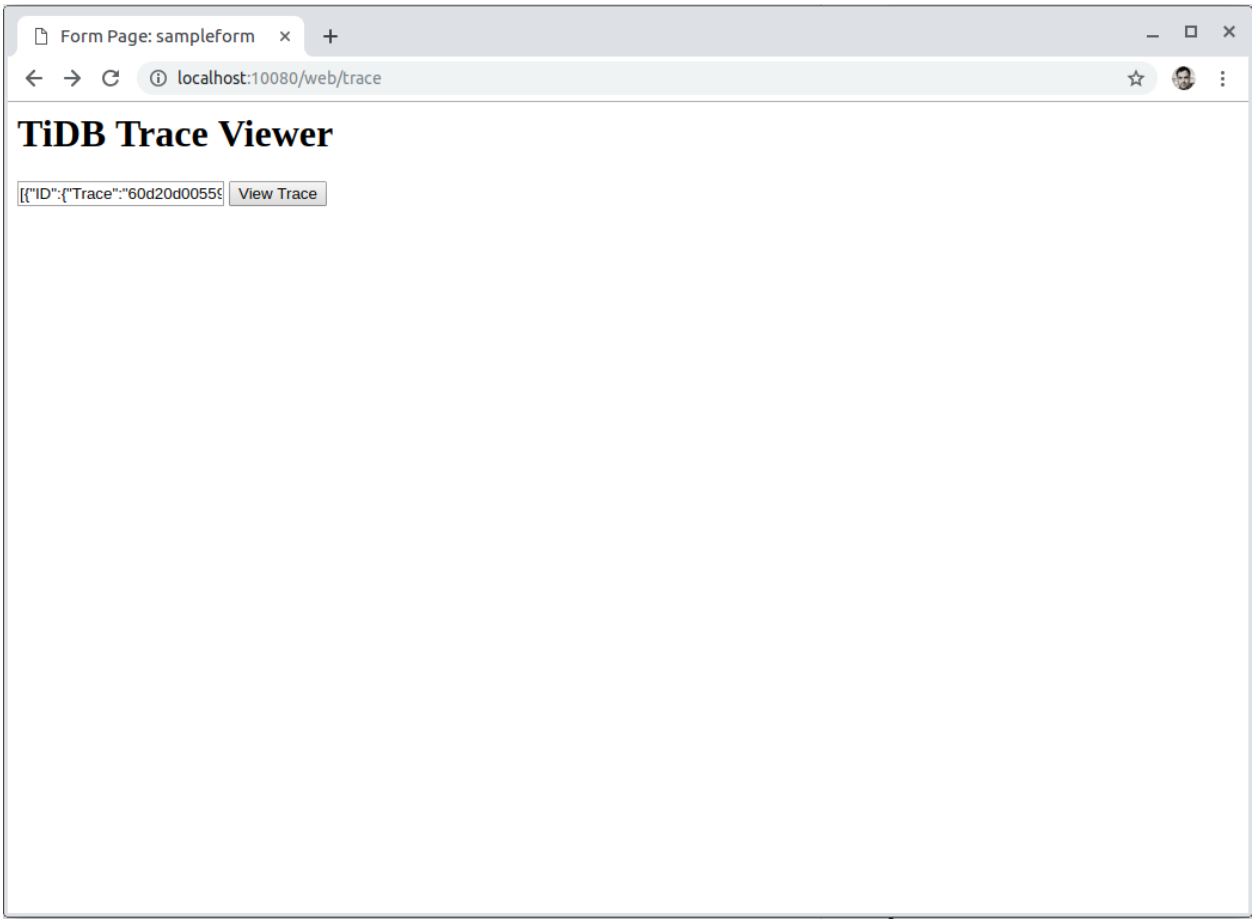


图 119: TiDB Trace Viewer-1

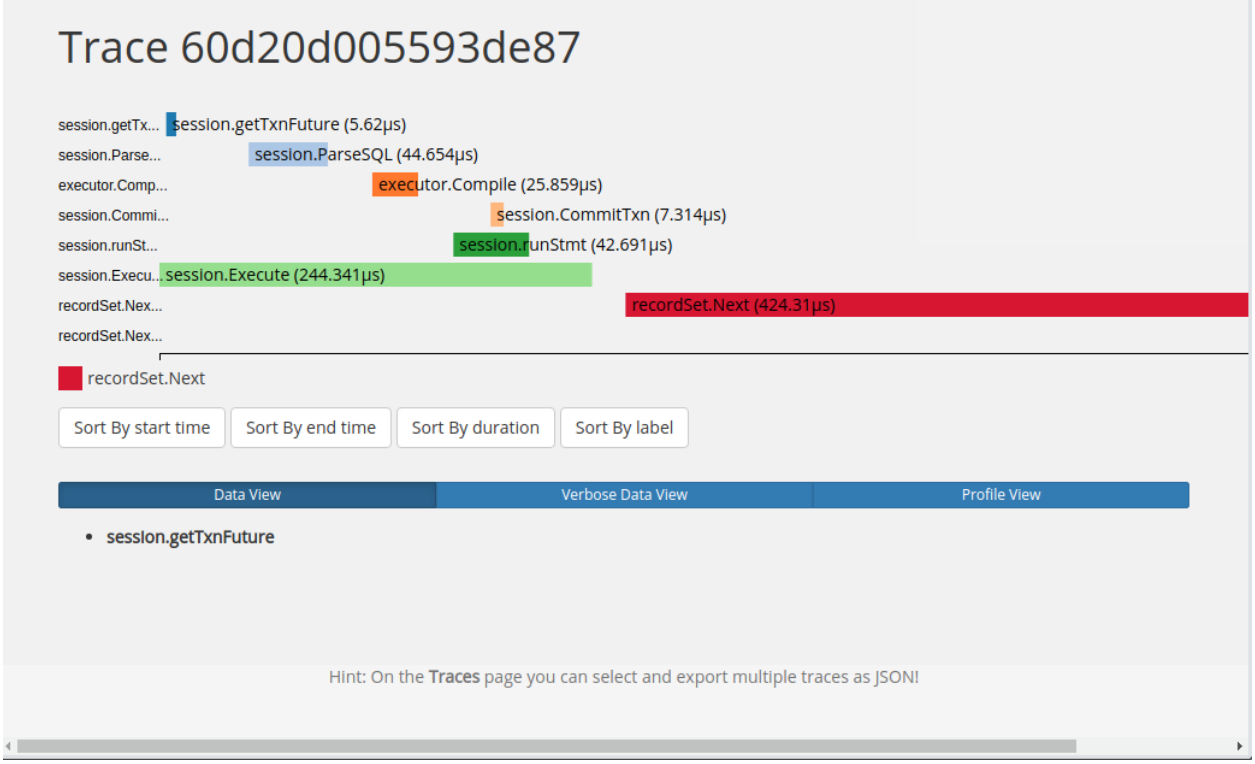


图 120: TiDB Trace Viewer-2

2.2.122.3 MySQL 兼容性

TRACE 语句是 TiDB 对 MySQL 语法的扩展。

2.2.122.4 另请参阅

- EXPLAIN ANALYZE

2.2.123 TRUNCATE

TRUNCATE 语句以非事务方式从表中删除所有数据。可认为 TRUNCATE 语句同 DROP TABLE + CREATE TABLE 组合在语义上相同，定义与 DROP TABLE 语句相同。

TRUNCATE TABLE tableName 和 TRUNCATE tableName 均为有效语法。

2.2.123.1 语法图

TruncateTableStmt:



图 121: TruncateTableStmt

OptTable:



图 122: OptTable

TableName:

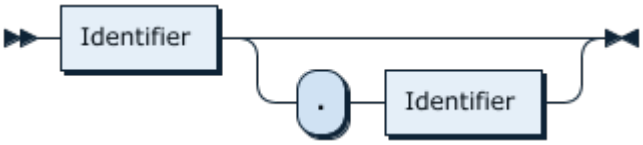


图 123: TableName

2.2.123.2 示例

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

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

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

```
TRUNCATE t1;
```

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

```
SELECT * FROM t1;
```

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

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
TRUNCATE TABLE t1;
```

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

2.2.123.3 MySQL 兼容性

TRUNCATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.123.4 另请参阅

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

2.2.124 UPDATE

UPDATE 语句用于修改指定表中的数据。

2.2.124.1 语法图

UpdateStmt:

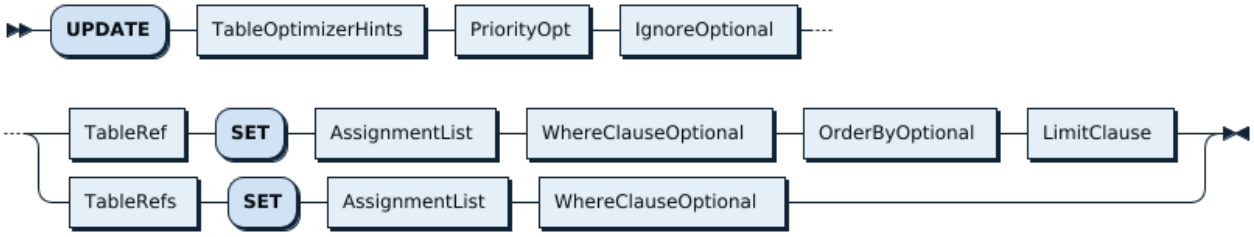


图 124: UpdateStmt

PriorityOpt:

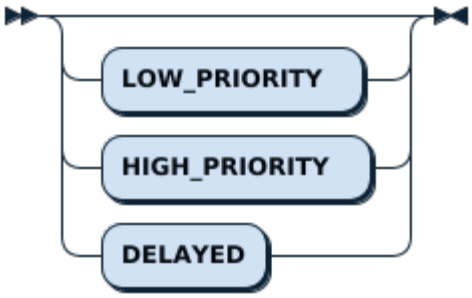


图 125: PriorityOpt

TableRef:

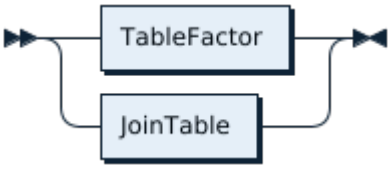


图 126: TableRef

TableRefs:



图 127: TableRefs

AssignmentList:

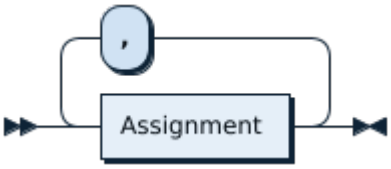


图 128: AssignmentList

WhereClauseOptional:

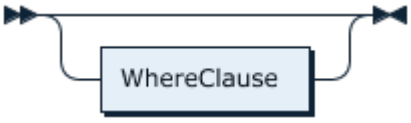


图 129: WhereClauseOptional

2.2.124.2 示例

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

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

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
+----+----+
3 rows in set (0.00 sec)
```

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

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

2.2.124.3 MySQL 兼容性

在计算表达式中的列时，TiDB 总使用原始的值。例如：

```
CREATE TABLE t (a int, b int);
INSERT INTO t VALUES (1,2);
UPDATE t SET a = a+1,b=a;
```

在 MySQL 中，b 列的值会被更新成 2，因为 b 列被设置为与 a 列相同，而 a（最初是 1）在同一条语句中被更新成了 2。

TiDB 遵守标准的 SQL 行为，这里将 b 列值更新成 1。

2.2.124.4 另请参阅

- INSERT
- SELECT
- DELETE
- REPLACE

2.2.125 USE

USE 语句可为用户会话选择当前数据库。

2.2.125.1 语法图

UseStmt:



图 130: UseStmt

DBName:



图 131: DBName

2.2.125.2 示例

```
USE mysql;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
SHOW TABLES;
```

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

```
| GLOBAL_VARIABLES        |
| bind_info               |
| columns_priv            |
| db                      |
| default_roles           |
| expr_pushdown_blacklist |
| gc_delete_range         |
| gc_delete_range_done    |
| global_priv             |
| help_topic              |
| opt_rule_blacklist      |
| role_edges              |
| stats_buckets           |
| stats_feedback          |
| stats_histograms        |
| stats_meta              |
| stats_top_n             |
| tables_priv             |
| tidb                    |
| user                    |
+-------------------------+
20 rows in set (0.01 sec)
```

```
CREATE DATABASE newtest;
```

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

```
USE newtest;
```

```
Database changed
```

```
SHOW TABLES;
```

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

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

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

```
SHOW TABLES;
```

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

```
| t1                |
+-------------------+
1 row in set (0.00 sec)
```

2.2.125.3 MySQL 兼容性

USE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

2.2.125.4 另请参阅

- CREATE DATABASE
- SHOW TABLES

2.2.126 WITH

公共表表达式 (CTE) 是一个临时的中间结果集，能够在 SQL 语句中引用多次，提高 SQL 语句的可读性与执行效率。在 TiDB 中可以通过 WITH 语句使用公共表表达式。

2.2.126.1 语法图

WithClause:

```
WithClause ::=
        "WITH" WithList
|       "WITH" recursive WithList
```

WithList:

```
WithList ::=
        WithList ',' CommonTableExpr
|       CommonTableExpr
```

CommonTableExpr:

```
CommonTableExpr ::=
        Identifier IdentListWithParenOpt "AS" SubSelect
```

IdentListWithParenOpt:

```
IdentListWithParenOpt ::=
( '(' IdentList ')' )?
```

2.2.126.2 示例

非递归的 CTE：

```
WITH CTE AS (SELECT 1, 2) SELECT * FROM cte t1, cte t2;
```

```
+---+---+---+---+
| 1 | 2 | 1 | 2 |
+---+---+---+---+
| 1 | 2 | 1 | 2 |
+---+---+---+---+
1 row in set (0.00 sec)
```

递归的 CTE：

```
WITH RECURSIVE cte(a) AS (SELECT 1 UNION SELECT a+1 FROM cte WHERE a < 5) SELECT * FROM cte;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

2.2.126.3 MySQL 兼容性

- 在严格模式下，当递归部分算出的数据长度超过初始部分的数据长度时，TiDB 会返回警告，而 MySQL 会返回错误。在非严格模式下，TiDB 与 MySQL 行为一致。
- 递归 CTE 所使用的数据类型由初始部分决定。初始部分的数据类型在某些情况（例如函数）下与 MySQL 并不完全一致。
- 多个 UNION / UNION ALL 情况下，MySQL 不允许 UNION 后面加 UNION ALL，TiDB 允许。
- 如果 CTE 的定义存在问题，TiDB 会报错，而 MySQL 在未引用的情况下不报错。

2.2.126.4 另请参阅

- SELECT
- INSERT
- DELETE
- UPDATE
- REPLACE

2.3 数据类型

2.3.1 数据类型概述

TiDB 支持除空间类型 (SPATIAL) 之外的所有 MySQL 数据类型，包括数值型类型、字符串类型、时间和日期类型、JSON 类型。

数据类型定义一般为 `T(M[, D])`，其中：

- `T` 表示具体的类型。
- `M` 在整数类型中表示最大显示长度；在浮点数或者定点数中表示精度；在字符类型中表示最大长度。`M` 的最大值取决于具体的类型。
- `D` 表示浮点数、定点数的小数位长度。
- `fsp` 在时间和日期类型里的 `TIME`、`DATETIME` 以及 `TIMESTAMP` 中表示秒的精度，其取值范围是 `0` 到 `6`。值为 `0` 表示没有小数部分。如果省略，则默认精度为 `0`。

2.3.2 数据类型的默认值

在一个数据类型描述中的 `DEFAULT value` 段描述了一个列的默认值。这个默认值必须是常量，不可以是一个函数或者是表达式。但是对于时间类型，可以例外的使用 `NOW`、`CURRENT_TIMESTAMP`、`LOCALTIME`、`LOCALTIMESTAMP` 等函数作为 `DATETIME` 或者 `TIMESTAMP` 的默认值。

`BLOB`、`TEXT` 以及 `JSON` 不可以设置默认值。

如果一个列的定义中没有 `DEFAULT` 的设置。TiDB 按照如下的规则决定：

- 如果该类型可以使用 `NULL` 作为值，那么这个列会在定义时添加隐式的默认值设置 `DEFAULT NULL`。
- 如果该类型无法使用 `NULL` 作为值，那么这个列在定义时不会添加隐式的默认值设置。

对于一个设置了 `NOT NULL` 但是没有显式设置 `DEFAULT` 的列，当 `INSERT`、`REPLACE` 没有涉及到该列的值时，TiDB 根据当时的 `SQL_MODE` 进行不同的行为：

- 如果此时是 `strict sql mode`，在事务中的语句会导致事务失败并回滚，非事务中的语句会直接报错。
- 如果此时不是 `strict sql mode`，TiDB 会为这列赋值为列数据类型的隐式默认值。

此时隐式默认值的设置按照如下规则：

- 对于数值类型，它们的默认值是 `0`。当有 `AUTO_INCREMENT` 参数时，默认值会按照增量情况赋予正确的值。
- 对于除了时间戳外的日期时间类型，默认值会是该类型的 “零值”。时间戳类型的默认值会是当前的时间。
- 对于除枚举以外的字符串类型，默认值会是空字符串。对于枚举类型，默认值是枚举中的第一个值。

2.3.3 数值类型

TiDB 支持 MySQL 所有的数值类型，按照精度可以分为：

- 整数类型（精确值）
- 浮点类型（近似值）
- 定点类型（精确值）

2.3.3.1 整数类型

TiDB 支持 MySQL 所有的整数类型，包括 INTEGER/INT、 TINYINT、 SMALLINT、 MEDIUMINT 以及 BIGINT，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	类型显示宽度，可选
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识，但是没有做补零的操作

2.3.3.1.1 类型定义

BIT 类型

比特值类型。M 表示比特位的长度，取值范围从 1 到 64，其默认值是 1。

```
BIT[(M)]
```

BOOLEAN 类型

布尔类型，别名为 BOOL，和 TINYINT(1) 等价。零值被认为是 False，非零值认为是 True。在 TiDB 内部，True 存储为 1，False 存储为 0。

```
BOOLEAN
```

TINYINT 类型

TINYINT 类型。有符号数的范围是 [-128，127]。无符号数的范围是 [0，255]。

```
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
```

SMALLINT 类型

SMALLINT 类型。有符号数的范围是 [-32768，32767]。无符号数的范围是 [0，65535]。

```
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
```

MEDIUMINT 类型

MEDIUMINT 类型。有符号数的范围是 [-8388608，8388607]。无符号数的范围是 [0，16777215]。

```
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
```

2.3.3.1.2 INTEGER 类型

INTEGER 类型，别名 INT。有符号数的范围是 [-2147483648，2147483647]。无符号数的范围是 [0，4294967295]。

```
INT[(M)] [UNSIGNED] [ZEROFILL]
```

或者：

```
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

2.3.3.1.3 BIGINT 类型

BIGINT 类型。有符号数的范围是 [-9223372036854775808，9223372036854775807]。无符号数的范围是 [0，18446744073709551615]。

```
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
```

2.3.3.1.4 存储空间以及取值范围

每种类型对存储空间的需求以及最大/最小值如下表所示：

类型	存储空间	最小值 (有符号/无符号)	最大值 (有符号/无符号)
TINYINT	1	-128 / 0	127 / 255
SMALLINT	2	-32768 / 0	32767 / 65535
MEDIUMINT	3	-8388608 / 0	8388607 / 16777215
INT	4	-2147483648 / 0	2147483647 / 4294967295
BIGINT	8	-9223372036854775808 / 0	9223372036854775807 / 18446744073709551615

2.3.3.2 浮点类型

TiDB 支持 MySQL 所有的浮点类型，包括 FLOAT、DOUBLE，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

2.3.3.2.1 类型定义

FLOAT 类型

单精度浮点数。允许的值范围为 -2^128 ~ +2^128，也即 -3.402823466E+38 到 -1.175494351E-38、0 和 1.175494351E-38 到 3.402823466E+38。这些是基于 IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

FLOAT(p) 类型中，p 表示精度（以位数表示）。只使用该值来确定是否结果列的数据类型为 FLOAT 或 DOUBLE。如果 p 为从 0 到 24，数据类型变为没有 M 或 D 值的 FLOAT。如果 p 为从 25 到 53，数据类型变为没有 M 或 D 值的 DOUBLE。结果列范围与本节前面描述的单精度 FLOAT 或双精度 DOUBLE 数据类型相同。

```
FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]
FLOAT(p) [UNSIGNED] [ZEROFILL]
```

> 注意：

- 与在 MySQL 中一样，FLOAT 数据类型用于存储近似值。对于类似货币的精确值，建议使用 DECIMAL 类型。
- 在 TiDB 中，FLOAT 数据类型默认的精度是 8 位，这与 MySQL 不同。在 MySQL 中，FLOAT 数据类型默认的精度是 6 位。例如，同时往 MySQL 和 TiDB 中类型为 FLOAT 的列中插入数据 123456789 和 1.23456789。在 MySQL 中查询对应的值，将会得到结果 123457000 和 1.23457。而在 TiDB 中查询对应的值，将会得到 123456790 和 1.2345679。

DOUBLE 类型

双精度浮点数，别名为 DOUBLE PRECISION。允许的值范围为：-2^1024 ~ +2^1024，也即是 -1.7976931348623157E+308 到 -2.2250738585072014E-308、0 和 2.2250738585072014E-308 到 1.7976931348623157E+308。这些是基于 IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

```
DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]
DOUBLE PRECISION [(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL]
```

警告：

与在 MySQL 中一样，DOUBLE 数据类型存储近似值。对于货币之类的精确值，建议使用 DECIMAL 类型。

2.3.3.2.2 存储空间

每种类型对存储空间的需求如下表所示：

类型	存储空间
FLOAT	4
FLOAT(p)	如果 0 <= p <= 24 为 4 个字节, 如果 25 <= p <= 53 为 8 个字节
DOUBLE	8

2.3.3.3 定点类型

TiDB 支持 MySQL 所有的定点类型，包括 DECIMAL、NUMERIC，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	数值总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

2.3.3.3.1 类型定义

DECIMAL 类型

定点数，别名为 NUMERIC。M 是数值位数（精度）的总数（整数位数 + 小数位数），D 是小数点（标度）后面的位数。小数点和 -（负数）符号不包括在 M 中。如果 D 是 0，则值没有小数点或分数部分。如果 D 被省略，默认是 0。如果 M 被省略，默认是 10。

```
DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL]
```

2.3.4 日期和时间类型

TiDB 支持 MySQL 所有的日期和时间类型，包括DATE、TIME、DATETIME、TIMESTAMP 以及YEAR。完整信息可以参考 MySQL 中的时间和日期类型。

每种类型都有有效值范围，值为 0 表示无效值。此外，TIMESTAMP 和 DATETIME 类型能自动生成新的时间值。

关于日期和时间值类型，需要注意：

- 日期部分必须是“年 -月 -日”的格式（例如 1998-09-04），而不是“月 -日 -年”或“日 -月 -年”的格式。
- 如果日期的年份部分是 2 位数，TiDB 会根据年份为两位数的具体规则进行转换。
- 如果格式必须是数值类型，TiDB 会自动将日期或时间值转换为数值类型。例如：

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------+----------------+--------------------+
| NOW()               | NOW()+0        | NOW(3)+0           |
+---------------------+----------------+--------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------+----------------+--------------------+
```

- TiDB 可以自动将无效值转换同一类型的零值。是否进行转换取决于 SQL 模式的设置。比如：

```
show create table t1;
```

```
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `a` time DEFAULT NULL
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+-----------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
select @@sql_mode;
```

```
+--
    ↪ -----------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
    ↪ -----------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+--
    ↪ -----------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
insert into t1 values (`2090-11-32:22:33:44`);
```

```
ERROR 1292 (22007): Truncated incorrect time value: `2090-11-32:22:33:44`
```

```
set @@sql_mode=``;
```

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

```
insert into t1 values (`2090-11-32:22:33:44`);
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
select * from t1;
```

```
+----------+
| a        |
+----------+
| 00:00:00 |
+----------+
1 row in set (0.01 sec)
```

- SQL 模式的不同设置，会改变 TiDB 对格式的要求。
- 如果 SQL 模式的 NO_ZERO_DATE 被禁用，TiDB 允许 DATE 和 DATETIME 列中的月份或日期为零。例如，2009-00-00 或 2009-01-00。如果使用函数计算这种日期类型，例如使用 DATE_SUB( ) 或 DATE_ADD( ) 函数，计算结果可能不正确。
- 默认情况下，TiDB 启用 NO_ZERO_DATE SQL 模式。该模式可以避免存储像 0000-00-00 这样的零值。

不同类型的零值如下表所示：

数据类型	零值
DATE	0000-00-00
TIME	00:00:00
DATETIME	0000-00-00 00:00:00
TIMESTAMP	0000-00-00 00:00:00
YEAR	0000

如果 SQL 模式允许使用无效的 DATE、DATETIME、TIMESTAMP 值，无效值会自动转换为相应的零值（0000-00-00 或 0000-00-00 00:00:00 ）。

2.3.4.1 类型定义

2.3.4.1.1 DATE 类型

DATE 类型只包含日期部分，不包含时间部分。DATE 类型的格式为 YYYY-MM-DD，支持的范围是 0000-01-01 到 9999-12-31。

```
DATE
```

2.3.4.1.2 TIME 类型

TIME 类型的格式为 HH:MM:SS[.fraction]，支持的范围是 -838:59:59.000000 到 838:59:59.000000。TIME 不仅可用于指示一天内的时间，还可用于指两个事件之间的时间间隔。fsp 参数表示秒精度，取值范围为：0 ~ 6，默认值为 0。

```
TIME[(fsp)]
```

> 注意：
>
> 注意 TIME 的缩写形式。例如，11:12 表示 11:12:00 而不是 00:11:12。但是，1112 表示 00:11:12。这些差异取决于：字符的存在与否。

2.3.4.1.3 DATETIME 类型

DATETIME 类型是日期和时间的组合，格式为 YYYY-MM-DD HH:MM:SS[.fraction]。支持的范围是 0000-01-01 00:00:00.000000 到 9999-12-31 23:59:59.999999。fsp 参数表示秒精度，取值范围为 0~6，默认值为 0。TiDB 支持字符串或数字转换为 DATETIME 类型。

```
DATETIME[(fsp)]
```

2.3.4.1.4 TIMESTAMP 类型

TIMESTAMP 类型是日期和时间的组合，支持的范围是 UTC 时间从 1970-01-01 00:00:01.000000 到 2038-01-19 03:14:07.999999。fsp 参数表示秒精度，取值范围为 0~6，默认值为 0。在 TIMESTAMP 中，不允许零出现在月份部分或日期部分，唯一的例外是零值本身 0000-00-00 00:00:00。

```
TIMESTAMP[(fsp)]
```

时区处理

当存储 TIMESTAMP 时，TiDB 会将当前时区的 TIMESTAMP 值转换为 UTC 时区。当读取 TIMESTAMP 时，TiDB 将存储的 TIMESTAMP 值从 UTC 时区转换为当前时区（注意：DATETIME 不会这样处理）。每次连接的默认时区是服务器的本地时区，可以通过环境变量 time_zone 进行修改。

> 警告：
>
> 和 MySQL 一样，TIMESTAMP 数据类型受 2038 年问题的影响。如果存储的值大于 2038，建议使用 DATETIME 类型。

2.3.4.1.5 YEAR 类型

YEAR 类型的格式为 YYYY，支持的值范围是 1901 到 2155，也支持零值 0000。

```
YEAR[(4)]
```

YEAR 类型遵循以下格式规则：

- 如果是四位数的数值，支持的范围是 1901 至 2155。
- 如果是四位数的字符串，支持的范围是 '1901' 到 '2155'。
- 如果是 1~99 之间的一位数或两位数的数字，1~69 换算成 2001_{2069}，$_{70}99$ 换算成 1970~1999。
- 支持 '0' 到 '99' 之间的一位数或两位数字符串的范围
- 将数值 0 转换为 0000，将字符串 '0' 或 '00' 转换为 '2000'。

无效的 YEAR 值会自动转换为 0000（如果用户没有使用 NO_ZERO_DATE SQL 模式）。

2.3.4.2 自动初始化和更新 TIMESTAMP 或 DATETIME

带有 TIMESTAMP 或 DATETIME 数据类型的列可以自动初始化为或更新为当前时间。

对于表中任何带有 TIMESTAMP 或 DATETIME 数据类型的列，你可以设置默认值，或自动更新为当前时间戳。

在定义列的时候，TIMESTAMP 和 DATETIME 可以通过 DEFAULT CURRENT_TIMESTAMP 和 ON UPDATE CURRENT_TIMESTAMP 来设置。DEFAULT 也可以设置为一个特定的值，例如 DEFAULT 0 或 DEFAULT '2000-01-01 00:00:00'。

```
CREATE TABLE t1 (
    ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    dt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

除非指定 DATETIME 的值为 NOT NULL，否则默认 DATETIME 的值为 NULL。指定 DATETIME 的值为 NOT NULL 时，如果没有设置默认值，则默认值为 0。

```
CREATE TABLE t1 (
    dt1 DATETIME ON UPDATE CURRENT_TIMESTAMP,          -- default NULL
    dt2 DATETIME NOT NULL ON UPDATE CURRENT_TIMESTAMP -- default 0
);
```

2.3.4.3 时间值的小数部分

DATETIME 和 TIMESTAMP 值最多可以有 6 位小数，精确到毫秒。如果包含小数部分，值的格式为 YYYY-MM-DD HH:MM:SS[.fraction]，小数部分的范围为 000000 到999999。必须使用小数点分隔小数部分与其他部分。

- 使用 type_name(fsp) 可以定义精确到小数的列，其中 type_name 可以是TIME、DATETIME 或 TIMESTAMP。例如：

  ```
  CREATE TABLE t1 (t TIME(3), dt DATETIME(6));
  ```

  fsp 范围是 0 到 6。

  0 表示没有小数部分。如果省略了 fsp ，默认为 0。

- 当插入包含小数部分的 TIME、DATETIME 或 TIMESTAMP 时，如果小数部分的位数过少或过多，可能需要进行四舍五入。例如：

  ```
  CREATE TABLE fractest( c1 TIME(2), c2 DATETIME(2), c3 TIMESTAMP(2) );
  ```

  ```
  Query OK, 0 rows affected (0.33 sec)
  ```

  ```
  INSERT INTO fractest VALUES
       > ('17:51:04.777', '2014-09-08 17:51:04.777',   '2014-09-08 17:51:04.777');
  ```

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

```
SELECT * FROM fractest;
```

```
+-------------|-------------------------|-------------------------+
| c1          | c2                      | c3                      |
+-------------|-------------------------|-------------------------+
| 17:51:04.78 | 2014-09-08 17:51:04.78 | 2014-09-08 17:51:04.78 |
+-------------|-------------------------|-------------------------+
1 row in set (0.00 sec)
```

2.3.4.4 日期和时间类型的转换

在日期和时间类型之间进行转换时，有些转换可能会导致信息丢失。例如，DATE、DATETIME 和 TIMESTAMP 都有各自的有效值范围。TIMESTAMP 不能早于 UTC 时间的 1970 年，也不能晚于 UTC 时间的 2038-01-19 03:14:07。根据这个规则，1968-01-01 对于 DATE 或 DATETIME 是有效的，但当 1968-01-01 转换为 TIMESTAMP 时，就会变成 0。

DATE 的转换：

- 当 DATE 转换为 DATETIME 或 TIMESTAMP 时，会添加时间部分 00:00:00，因为 DATE 不包含任何时间信息。
- 当 DATE 转换为 TIME 时，结果是 00:00:00。

DATETIME 或 TIMESTAMP 的转换：

- 当 DATETIME 或 TIMESTAMP 转换为 DATE 时，时间和小数部分将被舍弃。例如，1999-12-31 23:59:59.499 被转换为 1999-12-31。
- 当 DATETIME 或 TIMESTAMP 转换为 TIME 时，日期部分被舍弃，因为 TIME 不包含任何日期信息。

如果要将 TIME 转换为其他时间和日期格式，日期部分会自动指定为 CURRENT_DATE()。最终的转换结果是由 TIME 和 CURRENT_DATE() 组成的日期。也就是说，如果 TIME 的值超出了 00:00:00 到 23:59:59 的范围，那么转换后的日期部分并不表示当前的日期。

当 TIME 转换为 DATE 时，转换过程类似，时间部分被舍弃。

使用 CAST() 函数可以显式地将值转换为 DATE 类型。例如：

```
date_col = CAST(datetime_col AS DATE)
```

将 TIME 和 DATETIME 转换为数字格式。例如：

```
SELECT CURTIME(), CURTIME()+0, CURTIME(3)+0;
```

```
+-----------|-------------|--------------+
| CURTIME() | CURTIME()+0 | CURTIME(3)+0 |
+-----------|-------------|--------------+
| 09:28:00  |       92800 |    92800.887 |
+-----------|-------------|--------------+
```

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------|----------------|--------------------+
| NOW()               | NOW()+0        | NOW(3)+0           |
+---------------------|----------------|--------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------|----------------|--------------------+
```

2.3.4.5 年份为两位数

如果日期中包含年份为两位数，这个年份是有歧义的，并不显式地表示实际年份。

对于 DATETIME、DATE 和 TIMESTAMP 类型，TiDB 使用如下规则来消除歧义。

- 将 01 至 69 之间的值转换为 2001 至 2069 之间的值。
- 将 70 至 99 之间的值转化为 1970 至 1999 之间的值。

上述规则也适用于 YEAR 类型，但有一个例外。将数字 00 插入到 YEAR(4) 中时，结果是 0000 而不是 2000。

如果想让结果是 2000，需要指定值为 2000。

对于 MIN() 和 MAX() 等函数，年份为两位数时可能会得到错误的计算结果。建议年份为四位数时使用这类函数。

2.3.5 字符串类型

TiDB 支持 MySQL 所有的字符串类型，包括 CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM 以及 SET，完整信息参考这篇文档。

2.3.5.1 类型定义

2.3.5.1.1 CHAR 类型

定长字符串。CHAR 列的长度固定为创建表时声明的长度。M 表示列长度（字符的个数，不是字节的个数）。长度可以为从 0 到 255 的任何值。和 VARCHAR 类型不同，CHAR 列在写入时会对数据末尾的空格进行截断。

```
[NATIONAL] CHAR[(M)] [CHARACTER SET charset_name] [COLLATE collation_name]
```

2.3.5.1.2 VARCHAR 类型

变长字符串。M 表示最大列长度（字符的最大个数）。VARCHAR 的空间占用大小不得超过 65535 字节。在选择 VARCHAR 长度时，应当根据最长的行的大小和使用的字符集确定。

对于不同的字符集，单个字符所占用的空间可能有所不同。以下表格是各个字符集下单个字符占用的字节数，以及 VARCHAR 列长度的取值范围：

字符集	单个字符字节数	VARCHAR 最大列长度的取值范围
ascii	1	(0, 65535]
latin1	1	(0, 65535]
binary	1	(0, 65535]
utf8	3	(0, 21845]
utf8mb4	4	(0, 16383]

```
[NATIONAL] VARCHAR(M) [CHARACTER SET charset_name] [COLLATE collation_name]
```

2.3.5.1.3 TEXT 类型

文本串。M 表示最大列长度（字符的最大个数），范围是 0 到 65535。在选择 TEXT 长度时，应当根据最长的行的大小和使用的字符集确定。

```
TEXT[(M)] [CHARACTER SET charset_name] [COLLATE collation_name]
```

2.3.5.1.4 TINYTEXT 类型

类似于TEXT，区别在于最大列长度为 255。

```
TINYTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

2.3.5.1.5 MEDIUMTEXT 类型

类似于TEXT，区别在于最大列长度为 16,777,215。

```
MEDIUMTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

2.3.5.1.6 LONGTEXT 类型

类似于TEXT，区别在于最大列长度为 4,294,967,295。但由于TiDB 单列的限制，TiDB 中单列存储最大不超过 6 MB。

```
LONGTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

2.3.5.1.7 BINARY 类型

类似于CHAR，区别在于 BINARY 存储的是二进制字符串。

```
BINARY(M)
```

2.3.5.1.8 VARBINARY 类型

类似于VARCHAR，区别在于 VARBINARY 存储的是二进制字符串。

```
VARBINARY(M)
```

2.3.5.1.9 BLOB 类型

二进制大文件。M 表示最大列长度，单位是字节，范围是 0 到 65535。

```
BLOB[(M)]
```

2.3.5.1.10 TINYBLOB 类型

类似于BLOB，区别在于最大列长度为 255。

```
TINYBLOB
```

2.3.5.1.11 MEDIUMBLOB 类型

类似于BLOB，区别在于最大列长度为 16777215。

```
MEDIUMBLOB
```

2.3.5.1.12 LONGBLOB 类型

类似于BLOB，区别在于最大列长度为 4,294,967,295。但由于TiDB 单列的限制，TiDB 中单列存储最大不超过 6 MB。

```
LONGBLOB
```

2.3.5.1.13 ENUM 类型

枚举类型是一个字符串，它只能有一个值的字符串对象。其值必须是从一个固定集合中选取，这个固定集合在创建表的时候定义，语法是：

```
ENUM('value1','value2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

例如：

```
ENUM('apple', 'orange', 'pear')
```

枚举类型的值在 TiDB 内部使用数值来存储，每个值会按照定义的顺序转换为一个数字，比如上面的例子中，每个字符串值都会映射为一个数字：

值	数字
NULL	NULL
”	0
‘apple’	1
‘orange’	2
‘pear’	3

更多信息参考 MySQL 枚举文档。

2.3.5.1.14 SET 类型

集合类型是一个包含零个或多个值的字符串，其中每个值必须是从一个固定集合中选取，这个固定集合在创建表的时候定义，语法是：

```
SET('value1','value2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

例如：

```
SET('1', '2') NOT NULL
```

上面的例子中，这列的有效值可以是：

```
''
'1'
'2'
'1,2'
```

集合类型的值在 TiDB 内部会转换为一个 Int64 数值，每个元素是否存在用一个二进制位的 0/1 值来表示，比如这个例子 `SET('a','b','c','d')`，每一个元素都被映射为一个数字，且每个数字的二进制表示只会有一位是 1：

成员	十进制表示	二进制表示
‘a’	1	0001
‘b’	2	0010
‘c’	4	0100
‘d’	8	1000

这样对于值为 `('a', 'c')` 的元素，其二进制表示即为 0101。

更多信息参考 MySQL 集合文档。

2.4 函数与操作符

2.4.1 函数和操作符概述

TiDB 中函数和操作符使用方法与 MySQL 基本一致，详情参见：Functions and Operators。

在 SQL 语句中，表达式可用于诸如 SELECT 语句的 ORDER BY 或 HAVING 子句，SELECT/DELETE/UPDATE 语句的 WHERE 子句，或 SET 语句之类的地方。

可使用字面值，列名，NULL，内置函数，操作符等来书写表达式。其中有些表达式下推到 TiKV 上执行，详见下推到 TiKV 的表达式列表。

2.4.2 表达式求值的类型转换

TiDB 中表达式求值的类型转换与 MySQL 基本一致，详情参见 MySQL 表达式求值的类型转换。

2.4.3 操作符

操作符名	功能描述
AND, &&	逻辑与
=	赋值 (可用于 SET 语句中, 或用于 UPDATE 语句的 SET 中)
:=	赋值
BETWEEN ... AND ...	判断值满足范围
BINARY	将一个字符串转换为一个二进制字符串
&	位与
~	位非
|	位或
^	按位异或
CASE	case 操作符
DIV	整数除
/	除法
=	相等比较
<=>	空值安全型相等比较
>	大于
>=	大于或等于
IS	判断一个值是否等于一个布尔值
IS NOT	判断一个值是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
<<	左移
<	小于
<=	小于或等于
LIKE	简单模式匹配
-	减
lstinlineMOD!	求余
NOT, !	取反
NOT BETWEEN ... AND ...	判断值是否不在范围内
!=, <>	不等于
NOT LIKE	不符合简单模式匹配
NOT REGEXP	不符合正则表达式模式匹配
||, OR	逻辑或
+	加
REGEXP	使用正则表达式进行模式匹配
>>	右移
RLIKE	REGEXP 同义词
*	乘
-	取反符号
XOR	逻辑亦或

2.4.3.1 操作符优先级

操作符优先级显示在以下列表中，从最高优先级到最低优先级。同一行显示的操作符具有相同的优先级。

```
INTERVAL
BINARY
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<<, >>
&
|
= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
AND, &&
XOR
OR, ||
= (assignment), :=
```

详情参见这里。

2.4.3.2 比较方法和操作符

操作符名	功能描述
BETWEEN ... AND ...	判断值是否在范围内
COALESCE()	返回第一个非空值
=	相等比较
<=>	空值安全型相等比较
>	大于
>=	大于或等于
GREATEST()	返回最大值
IN()	判断值是否在一个值的集合内
INTERVAL()	返回一个小于第一个参数的参数的下标
IS	判断是否等于一个布尔值
IS NOT	判断是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
ISNULL()	判断参数是否为空
LEAST()	返回最小值
<	小于
<=	小于或等于
LIKE	简单模式匹配
NOT BETWEEN ... AND ...	判断值是否不在范围内

操作符名	功能描述
!= , <>	不等于
NOT IN()	判断值是否不在一个值的集合内
NOT LIKE	不满足简单模式匹配
STRCMP()	比较两个字符串

详情参见这里。

2.4.3.3 逻辑操作符

操作符名	功能描述
AND, &&	逻辑与
NOT, !	逻辑非
||, OR	逻辑或
XOR	逻辑异或

详情参见这里。

2.4.3.4 赋值操作符

操作符名	功能描述
=	赋值 (可用于 SET 语句中, 或用于 UPDATE 语句的 SET 中)
:=	赋值

详情参见这里。

2.4.4 控制流程函数

TiDB 支持使用 MySQL 5.7 中提供的控制流程函数。

函数名	功能描述
CASE	Case 操作符
IF()	构建 if/else
IFNULL()	构建 Null if/else
NULLIF()	如果 expr1 = expr2，返回 NULL

2.4.5 字符串函数

TiDB 支持使用大部分 MySQL 5.7 中提供的字符串函数和一部分 Oracle 21 所提供的函数。

2.4.5.1 支持的函数

函数名	功能描述
ASCII()	返回最左字符的数值
BIN()	返回一个数的二进制值的字符串表示
BIT_LENGTH()	返回字符串的位长度
CHAR()	返回由整数的代码值所给出的字符组成的字符串
CHAR_LENGTH()	返回字符串的字符长度
CHARACTER_LENGTH()	与 CHAR_LENGTH() 功能相同
CONCAT()	返回连接的字符串
CONCAT_WS()	返回由分隔符连接的字符串
ELT()	返回指定位置的字符串
EXPORT_SET()	返回一个字符串，其中值位中设置的每个位，可以得到一个 on 字符串，而每个未设置的位，可以得到
FIELD()	返回参数在后续参数中出现的第一个位置
FIND_IN_SET()	返回第一个参数在第二个参数中出现的位置
FORMAT()	返回指定小数位数格式的数字
FROM_BASE64()	解码 base-64 表示的字符串，并返回结果
HEX()	返回一个十进制数或字符串值的 16 进制表示
INSERT()	在指定位置插入一个子字符串，最多不超过指定字符数
INSTR()	返回第一次出现的子字符串的索引
LCASE()	与 LOWER() 功能相同
LEFT()	返回最左侧指定长度的字符
LENGTH()	返回字符串长度，单位为字节
LIKE	进行简单模式匹配
LOCATE()	返回第一次出现的子字符串的位置
LOWER()	返回全小写的参数
LPAD()	返回字符串参数，左侧添加指定字符串
LTRIM()	去掉前缀空格
MAKE_SET()	返回一组用逗号分隔的字符串，这些字符串的位数与给定的 bits 参数对应
MID()	返回一个以指定位置开始的子字符串
NOT LIKE	否定简单模式匹配
NOT REGEXP	REGEXP 的否定形式
OCT()	返回一个数值的八进制表示，形式为字符串
OCTET_LENGTH()	与 LENGTH() 功能相同
ORD()	返回该参数最左侧字符的字符编码
POSITION()	与 LOCATE() 功能相同
QUOTE()	使参数逃逸，为了在 SQL 语句中使用
REGEXP	使用正则表达式匹配模式
REPEAT()	以指定次数重复一个字符串
REPLACE()	替换所有出现的指定字符串
REVERSE()	反转字符串里的所有字符
RIGHT()	返回指定数量的最右侧的字符
RLIKE	与 REGEXP 功能相同
RPAD()	以指定次数添加字符串
RTRIM()	去掉后缀空格

函数名	功能描述
SPACE()	返回指定数量的空格，形式为字符串
STRCMP()	比较两个字符串
SUBSTR()	返回指定的子字符串
SUBSTRING()	返回指定的子字符串
SUBSTRING_INDEX()	从一个字符串中返回指定出现次数的定界符之前的子字符串
TO_BASE64()	返回转化为 base-64 表示的字符串参数
TRANSLATE()	将字符串中出现的所有指定字符替换为其它字符。这个函数不会像 Oracle 一样将空字符串视为NULL
TRIM()	去掉前缀和后缀空格
UCASE()	与 UPPER() 功能相同
UNHEX()	返回一个数的十六进制表示，形式为字符串
UPPER()	参数转换为大写形式

2.4.5.2 不支持的函数

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

2.4.6 数值函数与操作符

TiDB 支持使用 MySQL 5.7 中提供的所有数值函数与操作符。

2.4.6.1 算术操作符

操作符名	功能描述
+	加号
-	减号
*	乘号
/	除号
DIV	整数除法
lstinlineMOD!	模运算，取余
-	更改参数符号

2.4.6.2 数学函数

函数名	功能描述
POW()	返回参数的指定乘方的结果值
POWER()	返回参数的指定乘方的结果值

函数名	功能描述
EXP()	返回 e（自然对数的底）的指定乘方后的值
SQRT()	返回非负数的二次方根
LN()	返回参数的自然对数
LOG()	返回第一个参数的自然对数
LOG2()	返回参数以 2 为底的对数
LOG10()	返回参数以 10 为底的对数
PI()	返回 pi 的值
TAN()	返回参数的正切值
COT()	返回参数的余切值
SIN()	返回参数的正弦值
COS()	返回参数的余弦值
ATAN()	返回参数的反正切值
ATAN2(), ATAN()	返回两个参数的反正切值
ASIN()	返回参数的反正弦值
ACOS()	返回参数的反余弦值
RADIANS()	返回由度转化为弧度的参数
DEGREES()	返回由弧度转化为度的参数
MOD()	返回余数
ABS()	返回参数的绝对值
CEIL()	返回不小于参数的最小整数值
CEILING()	返回不小于参数的最小整数值
FLOOR()	返回不大于参数的最大整数值
ROUND()	返回参数最近似的整数或指定小数位数的数值
RAND()	返回一个随机浮点值
SIGN()	返回参数的符号
CONV()	不同数基间转换数字，返回数字的字符串表示
TRUNCATE()	返回被舍位至指定小数位数的数字
CRC32()	计算循环冗余码校验值并返回一个 32 位无符号值

2.4.7 日期和时间函数

TiDB 支持使用 MySQL 5.7 中提供的所有日期和时间函数。

2.4.7.1 日期时间函数表

函数名	功能描述
ADDDATE()	将时间间隔添加到日期上
ADDTIME()	时间数值相加
CONVERT_TZ()	转换时区
CURDATE()	返回当前日期
CURRENT_DATE(), CURRENT_DATE	与 CURDATE() 同义
CURRENT_TIME(), CURRENT_TIME	与 CURTIME() 同义

函数名	功能描述
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	与 NOW() 同义
CURTIME()	返回当前时间
DATE()	从日期或日期/时间表达式中提取日期部分
DATE_ADD()	将时间间隔添加到日期上
DATE_FORMAT()	返回满足指定格式的日期/时间
DATE_SUB()	从日期减去指定的时间间隔
DATEDIFF()	返回两个日期间隔的天数
DAY()	与 DAYOFMONTH() 同义
DAYNAME()	返回星期名称
DAYOFMONTH()	返回参数对应的天数部分 (1-31)
DAYOFWEEK()	返回参数对应的星期下标
DAYOFYEAR()	返回参数代表一年的哪一天 (1-366)
EXTRACT()	提取日期/时间中的单独部分
FROM_DAYS()	将天数转化为日期
FROM_UNIXTIME()	将 Unix 时间戳格式化为日期
GET_FORMAT()	返回满足日期格式的字符串
HOUR()	提取日期/时间表达式中的小时部分
LAST_DAY	返回参数中月份的最后一天
LOCALTIME(), LOCALTIME	与 NOW() 同义
LOCALTIMESTAMP, LOCALTIMESTAMP()	与 NOW() 同义
MAKEDATE()	根据给定的年份和一年中的天数生成一个日期
MAKETIME()	根据给定的时、分、秒生成一个时间
MICROSECOND()	返回参数的微秒部分
MINUTE()	返回参数的分钟部分
MONTH()	返回参数的月份部分
MONTHNAME()	返回参数的月份名称
NOW()	返回当前日期和时间
PERIOD_ADD()	在年 -月表达式上添加一段时间 (数个月)
PERIOD_DIFF()	返回间隔的月数
QUARTER()	返回参数对应的季度 (1-4)
SEC_TO_TIME()	将秒数转化为 ‘HH:MM:SS’ 的格式
SECOND()	返回秒数 (0-59)
STR_TO_DATE()	将字符串转化为日期
SUBDATE()	当传入三个参数时作为 DATE_SUB() 的同义
SUBTIME()	从一个时间中减去一段时间
SYSDATE()	返回该方法执行时的时间
TIME()	返回参数的时间表达式部分
TIME_FORMAT()	格式化时间
TIME_TO_SEC()	返回参数对应的秒数
TIMEDIFF()	返回时间间隔
TIMESTAMP()	传入一个参数时候, 该方法返回日期或日期/时间表达式, 传入两个参数时候, 返回参
TIMESTAMPADD()	在日期/时间表达式上增加一段时间间隔
TIMESTAMPDIFF()	从日期/时间表达式中减去一段时间间隔

函数名	功能描述
TO_DAYS()	将参数转化对应的天数 (从第 0 年开始)
TO_SECONDS()	将日期或日期/时间参数转化为秒数 (从第 0 年开始)
UNIX_TIMESTAMP()	返回一个 Unix 时间戳
UTC_DATE()	返回当前的 UTC 日期
UTC_TIME()	返回当前的 UTC 时间
UTC_TIMESTAMP()	返回当前的 UTC 日期和时间
WEEK()	返回参数所在的一年中的星期数
WEEKDAY()	返回星期下标
WEEKOFYEAR()	返回参数在日历中对应的一年中的星期数
YEAR()	返回参数对应的年数
YEARWEEK()	返回年数和星期数

2.4.8 位函数和操作符

TiDB 支持使用 MySQL 5.7 中提供的所有位函数和操作符。

位函数和操作符表

函数和操作符名	功能描述
BIT_COUNT()	返回参数二进制表示中为 1 的个数
&	位与
~	按位取反
|	位或
[^](https://dev.mysql.com/doc/refman/5.7/en/bit-functions.html#operator_bitwise-xor)	位异或
<<	左移
>>	右移

2.4.9 Cast 函数和操作符

Cast 函数和操作符用于将某种数据类型的值转换为另一种数据类型。TiDB 支持使用 MySQL 5.7 中提供的所有 Cast 函数和操作符。

2.4.9.1 Cast 函数和操作符表

函数和操作符名	功能描述
BINARY	将一个字符串转换成一个二进制字符串
CAST()	将一个值转换成一个确定类型
CONVERT()	将一个值转换成一个确定类型

2.4.10 加密和压缩函数

TiDB 支持使用 MySQL 5.7 中提供的大部分加密和压缩函数。

2.4.10.1 支持的函数

函数名	功能描述
MD5()	计算字符串的 MD5 校验和
PASSWORD()	计算并返回密码字符串
RANDOM_BYTES()	返回随机字节向量
SHA1(), SHA()	计算 SHA-1 160 位校验和
SHA2()	计算 SHA-2 校验和
AES_DECRYPT()	使用 AES 解密
AES_ENCRYPT()	使用 AES 加密
COMPRESS()	返回经过压缩的二进制字符串
UNCOMPRESS()	解压缩字符串
UNCOMPRESSED_LENGTH()	返回字符串解压后的长度
CREATE_ASYMMETRIC_PRIV_KEY()	创建私钥
CREATE_ASYMMETRIC_PUB_KEY()	创建公钥
CREATE_DH_PARAMETERS()	创建 DH 共享密钥
CREATE_DIGEST	从字符串创建摘要
ASYMMETRIC_DECRYPT()	使用公钥或私钥解密密文
ASYMMETRIC_DERIVE()	从非对称密钥导出对称密钥
ASYMMETRIC_ENCRYPT()	使用公钥或私钥加密明文
ASYMMETRIC_SIGN	从摘要创建签名
ASYMMETRIC_VERIFY()	验证签名字符串是否匹配摘要字符串

2.4.10.2 不支持的函数

- DES_DECRYPT()、DES_ENCRYPT()、OLD_PASSWORD() 和 ENCRYPT()：这些函数在 MySQL 5.7 中被废弃，并且

已在 MySQL 8.0 中移除。

- `VALIDATE_PASSWORD_STRENGTH()` 函数。
- 只在 MySQL 企业版中支持的函数。见 Issue #2632。

2.4.11 锁函数

TiDB 支持 MySQL 5.7 中的大部分用户级锁函数。

2.4.11.1 支持的函数

函数名	功能描述
`GET_LOCK(lockName, timeout)`	获取咨询锁。`lockName` 参数不得超过 64 个字符。在超时前，TiDB 最长等待 `timeout` 秒并返回失败。
`RELEASE_LOCK(lockName)`	释放先前获取的锁。`lockName` 参数不得超过 64 个字符。
`RELEASE_ALL_LOCKS()`	释放当前会话持有的所有锁。

2.4.11.2 MySQL 兼容性

- TiDB 允许的最短超时时间为 1 秒，最长超时时间为 1 小时（即 3600 秒）。而 MySQL 允许最短 0 秒和最长无限超时（`timeout=-1`）。TiDB 会自动将超出范围的值转换为最接近的允许值，`timeout=-1` 会被转换为 3600 秒。
- TiDB 不会自动检测用户级锁导致的死锁。死锁会话将在 1 小时内超时，但你也可以在任一受影响的会话上使用 `KILL` 语句手动终止死锁。你还可以通过始终用相同顺序获取用户级锁的方法来防止死锁。
- 在 TiDB 中，锁对集群中所有 TiDB 服务器生效。而在 MySQL Cluster 和 Group Replication 中，锁只对本地单个服务器生效。

2.4.11.3 不支持的函数

- `IS_FREE_LOCK()`
- `IS_USED_LOCK()`

2.4.12 信息函数

TiDB 支持使用 MySQL 5.7 中提供的大部分信息函数。

2.4.12.1 支持的函数

函数名	功能描述
`BENCHMARK()`	循环执行一个表达式
`CONNECTION_ID()`	返回当前连接的连接 ID (线程 ID)

函数名	功能描述
CURRENT_USER(), CURRENT_USER	返回当前用户的用户名和主机名
DATABASE()	返回默认 (当前) 的数据库名
FOUND_ROWS()	该函数返回对于一个包含 LIMIT 的 SELECT 查询语句，在不包含 LIMIT 的情况下回返回的记录数
LAST_INSERT_ID()	返回最后一条 INSERT 语句中自增列的值
ROW_COUNT()	影响的行数
SCHEMA()	与 DATABASE() 同义
SESSION_USER()	与 USER() 同义
SYSTEM_USER()	与 USER() 同义
USER()	返回客户端提供的用户名和主机名
VERSION()	返回当前 MySQL 服务器的版本信息

2.4.12.2 不支持的函数

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

2.4.13 GROUP BY 聚合函数

本文将详细介绍 TiDB 支持的聚合函数。

2.4.13.1 TiDB 支持的聚合函数

TiDB 支持的 MySQL GROUP BY 聚合函数如下所示：

函数名	功能描述
COUNT()	返回检索到的行的数目
COUNT(DISTINCT)	返回不同值的数目
SUM()	返回和
AVG()	返回平均值
MAX()	返回最大值
MIN()	返回最小值
GROUP_CONCAT()	返回连接的字符串
VARIANCE(), VAR_POP()	返回总体标准方差
STD(), STDDEV(), STDDEV_POP	返回总体标准差
VAR_SAMP()	返回采样方差
STDDEV_SAMP()	返回采样标准方差
JSON_OBJECTAGG(key, value)	将结果集返回为单个含 (key, value) 键值对的 JSON object

注意：

- 除非另有说明，否则聚合函数默认忽略 NULL 值。
- 如果在不包含 GROUP BY 子句的语句中使用聚合函数，则相当于对所有行进行分组。

另外，TiDB 还支持以下聚合函数：

- APPROX_PERCENTILE(expr, constant_integer_expr)

  该函数用于计算 expr 值的百分位数。参数 constant_integer_expr 是一个取值为区间 [1,100] 内整数的常量表达式，表示百分数。一个百分位数 Pk（k为百分数）表示数据集中至少有 k% 的数据小于等于 Pk。

  该函数中，表达式的返回结果必须为数值类型或日期与时间类型。函数不支持计算其他类型的返回结果，并直接返回 NULL。

  以下是一个计算第 50 百分位数的例子：

```
drop table if exists t;
create table t(a int);
insert into t values(1), (2), (3);
```

```
select approx_percentile(a, 50) from t;
```

```
+--------------------------+
| approx_percentile(a, 50) |
+--------------------------+
|                        2 |
+--------------------------+
1 row in set (0.00 sec)
```

上述聚合函数除 GROUP_CONCAT() 和 APPROX_PERCENTILE() 以外，均可作为窗口函数使用。

2.4.13.2 GROUP BY 修饰符

TiDB 目前不支持 GROUP BY 修饰符，例如 WITH ROLLUP，将来会提供支持。详情参阅 #4250。

2.4.13.3 对 SQL 模式的支持

TiDB 支持 SQL 模式 ONLY_FULL_GROUP_BY，当启用该模式时，TiDB 拒绝不明确的非聚合列的查询。例如，以下查询在启用 ONLY_FULL_GROUP_BY 时是不合规的，因为 SELECT 列表中的非聚合列 “b” 在 GROUP BY 语句中不显示：

```
drop table if exists t;
create table t(a bigint, b bigint, c bigint);
insert into t values(1, 2, 3), (2, 2, 3), (3, 2, 3);
```

```
select a, b, sum(c) from t group by a;
```

```
+------+------+--------+
| a    | b    | sum(c) |
+------+------+--------+
|    1 |    2 |      3 |
|    2 |    2 |      3 |
|    3 |    2 |      3 |
+------+------+--------+
3 rows in set (0.01 sec)
```

```
set sql_mode = 'ONLY_FULL_GROUP_BY';
```

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

```
select a, b, sum(c) from t group by a;
```

```
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP BY clause and contains
    ↪ nonaggregated column 'b' which is not functionally dependent on columns in GROUP BY
    ↪ clause; this is incompatible with sql_mode=only_full_group_by
```

目前，TiDB 默认开启 SQL 模式ONLY_FULL_GROUP_BY。

2.4.13.3.1 与 MySQL 的区别

TiDB 目前实现的 ONLY_FULL_GROUP_BY 没有 MySQL 5.7 严格。例如，假设我们执行以下查询，希望结果按 “c” 排序：

```
drop table if exists t;
create table t(a bigint, b bigint, c bigint);
insert into t values(1, 2, 1), (1, 2, 2), (1, 3, 1), (1, 3, 2);
select distinct a, b from t order by c;
```

要对结果进行排序，必须先清除重复。但选择保留哪一行会影响 c 的保留值，也会影响排序，并使其具有任意性。

在 MySQL 中，ORDER BY 表达式需至少满足以下条件之一，否则 DISTINCT 和 ORDER BY 查询将因不合规而被拒绝：

- 表达式等同于 SELECT 列表中的一个。
- 表达式引用并属于查询选择表的所有列都是 SELECT 列表的元素。

但是在 TiDB 中，上述查询是合规的，详情参阅 #4254。

TiDB 中另一个标准 SQL 的扩展允许 HAVING 子句中的引用使用 SELECT 列表中的别名表达式。例如：以下查询返回在 orders 中只出现一次的 name 值

```
select name, count(name) from orders
group by name
having count(name) = 1;
```

这个 TiDB 扩展允许在聚合列的 HAVING 子句中使用别名：

```
select name, count(name) as c from orders
group by name
having c = 1;
```

标准 SQL 只支持 GROUP BY 子句中的列表达式，以下语句不合规，因为 FLOOR(value/100) 是一个非列表达式：

```
select id, floor(value/100)
from tbl_name
group by id, floor(value/100);
```

TiDB 对标准 SQL 的扩展支持 GROUP BY 子句中非列表达式，认为上述语句合规。

标准 SQL 也不支持 GROUP BY 子句中使用别名。TiDB 对标准 SQL 的扩展支持使用别名，查询的另一种写法如下：

```
select id, floor(value/100) as val
from tbl_name
group by id, val;
```

2.4.13.4 TiDB 不支持的聚合函数

TiDB 目前不支持的聚合函数如下所示，相关进展参阅 TiDB #7623。

- JSON_ARRAYAGG

2.4.14 窗口函数

TiDB 中窗口函数的使用方法与 MySQL 8.0 基本一致，详情可参见 MySQL 窗口函数。由于窗口函数会使用一些保留关键字，可能导致原先可以正常执行的 SQL 语句在升级 TiDB 后无法被解析语法，此时可以将 tidb_enable_window_function 设置为 0，该参数的默认值为 1。

TiDB 支持除 GROUP_CONCAT() 和 APPROX_PERCENTILE() 以外的所有GROUP BY 聚合函数。此外，TiDB 支持的其他窗口函数如下：

函数名	功能描述
CUME_DIST()	返回一组值中的累积分布
DENSE_RANK()	返回分区中当前行的排名，并且排名是连续的
FIRST_VALUE()	当前窗口中第一行的表达式值
LAG()	分区中当前行前面第 N 行的表达式值
LAST_VALUE()	当前窗口中最后一行的表达式值
LEAD()	分区中当前行后面第 N 行的表达式值

函数名	功能描述
NTH_VALUE()	当前窗口中第 N 行的表达式值
NTILE()	将分区划分为 N 桶，为分区中的每一行分配桶号
PERCENT_RANK()	返回分区中小于当前行的百分比
RANK()	返回分区中当前行的排名，排名可能不连续
ROW_NUMBER()	返回分区中当前行的编号

2.4.15 其他函数

TiDB 支持使用 MySQL 5.7 中提供的大部分其他函数。

2.4.15.1 支持的函数

函数名	功能描述
ANY_VALUE()	在 ONLY_FULL_GROUP_BY 模式下，防止带有 GROUP BY 的语句报错
DEFAULT()	返回表的某一列的默认值
INET_ATON()	将 IP 地址转换为数值
INET_NTOA()	将数值转换为 IP 地址
INET6_ATON()	将 IPv6 地址转换为数值
INET6_NTOA()	将数值转换为 IPv6 地址
IS_IPV4()	判断参数是否为 IPv4 地址
IS_IPV4_COMPAT()	判断参数是否为兼容 IPv4 的地址
IS_IPV4_MAPPED()	判断参数是否为 IPv4 映射的地址
IS_IPV6()	判断参数是否为 IPv6 地址
NAME_CONST()	可以用于重命名列名
SLEEP()	让语句暂停执行几秒时间
UUID()	返回一个通用唯一识别码 (UUID)
VALUES()	定义 INSERT 语句使用的值

2.4.15.2 不支持的函数

函数名	功能描述
GET_LOCK()	获取命名锁，详见 TiDB #10929
RELEASE_LOCK()	释放命名锁，详见 TiDB #10929
UUID_SHORT()	基于特定假设提供唯一的 UUID，目前这些假设在 TiDB 中不存在，详见 TiDB #4620
MASTER_WAIT_POS()	与 MySQL 同步相关

2.4.16 精度数学

TiDB 中精度数学计算与 MySQL 中基本一致。详情请参见：Precision Math

- 数值类型
- DECIMAL 数据类型的特性

2.4.16.1 数值类型

精确数值运算的范围包括精确值数据类型（整型和 DECIMAL 类型），以及精确值数字字面量。近似值数据类型和近似值数字字面量被作为浮点数来处理。

精确值数字字面量包含整数部分或小数部分，或二者都包含。精确值数字字面量可以包含符号位。例如：1, .2, 3.4, -5, -6.78, +9.10。

近似值数字字面量以一个包含尾数和指数的科学计数法表示（基数为 10）。其中尾数和指数可以分别或同时带有符号位。例如：1.2E3, 1.2E-3, -1.2E3, -1.2E-3。

两个看起来相似的数字可能会被以不同的方式进行处理。例如：2.34 是精确值（定点数），而 2.3E0 是近似值（浮点数）。

DECIMAL 数据类型是定点数类型，其运算是精确计算。FLOAT 和 DOUBLE 数据类型是浮点类型，其运算是近似计算。

2.4.16.2 DECIMAL 数据类型的特性

本节讨论 DECIMAL 数据类型的特性，主要涉及以下几点：

1. 最大位数
2. 存储格式
3. 存储要求

DECIMAL 列的声明语法为 DECIMAL(M, D)。其中参数值意义及其范围如下：

- M 表示最大的数字位数（精度），1<= M <= 65。
- D 表示小数点右边数字的位数（标度）。1 <= D <= 30 且不大于 M。

M 的最大值 65 表示 DECIMAL 值的计算精确到 65 位数字。该精度同样适用于其精确值字面量。

DECIMAL 列的值采用二进制进行存储，其将每 9 位十进制数字包装成 4 个字节。其中整数和小数部分分别确定所需的存储空间。如果数字位数为 9 的倍数，则每 9 位十进制数字各采用 4 个字节进行存储，对于剩余不足 9 位的数字，所需的存储空间如下表所示

剩余数字位数	存储所需字节数
0	0
1–2	1
3–4	2
5–6	3
7–9	4

例如：

- 定义类型为 DECIMAL(18, 9) 的列，其小数点两侧均各包含 9 位十进制数字。因此，分别需要 4 个字节的存储空间。
- 定义类型为 DECIMAL(20, 6) 的列，其小数部分包含 6 位十进制数字，整数部分包含 14 位十进制数字。整数部分中 9 位数字需要 4 个字节进行存储，其余 5 位数字需要 3 个字节进行存储。小数部分 6 位数字需要 3 个字节进行存储。

DECIMAL 列不存储前导的字符 + 或字符 - 或数字 0。如果将 +0003.1 插入到 DECIMAL(5, 1) 列中，则将其存储为 3.1。对于负数，不存储字符 - 的字面值。

DECIMAL 列不允许插入大于列定义的隐含范围的值。例如：DECIMAL(3, 0) 列范围为 -999 到 999。DECIMAL(M, D) 列小数点左边部分最多支持 M-D 位数字。

有关 DECIMAL 值的内部格式完整说明，请参阅 TiDB 源码文件 types/mydecimal.go。

2.4.16.3 表达式计算

在涉及精度数学计算的表达式中，TiDB 会尽可能不做任何修改的使用每个输入的数值。比如：在计算比较函数时，参与运算的数字将不做任何改变。在严格 SQL 模式下，向一个数据列插入一个值时，如果该值处于这一列的值域范围内，这个值将直接不做任何修改的直接插入进去，提取这个值的时候，取得的值和插入的值将会是同一个值。当处于非严格 SQL 模式时，TiDB 会允许数据插入过程中发生的数据截断。

处理数值类型表达式取决于这个表达式参数的具体值：

- 当表达式参数中包含近似值时，这个表达式的结果也是近似值，TiDB 会使用浮点数对应的计算逻辑返回一个浮点数的结果。
- 当表达式参数中不包含任何近似值时（也就是说表达式的参数全部是精确值），如果某个精确值包含小数部分，TiDB 会对这个表达式使用 DECIMAL 对应的计算逻辑，返回一个 DECIMAL 的结果，精确到 65 位数字。
- 其他情况下，表达式只会包含整数参数，这个表达式的结果也是精确的，TiDB 会使用整数对应的计算逻辑返回一个整数结果，精度和 BIGINT 保持一致（64 位）。

如果数值类型表达式中包含字符串参数，这些字符串参数将被转换成双精度浮点数，这个表达式的计算结果将是个近似值。

向一个数值类型列插入数据的具体行为会受到 SQL 模式的影响。接下来的讨论将围绕严格模式以及 ERROR_FOR_DIVISION_BY_ZERO 模式展开，如果要打开所有的限制，可以简单的使用 TRADITIONAL 模式，这个模式将同时使用严格模式以及 ERROR_FOR_DIVISION_BY_ZERO 模式：

```
SET sql_mode = 'TRADITIONAL';
```

向一个具有精确值类型（DECIMAL 或者整数类型）的列插入数据时，如果插入的数据位于该列的值域范围内将使用该数据的精确值。如果该数据的小数部分太长，将会发生数值修约，这时会有 warning 产生，具体内容可以看“数值修约”。

如果该数据整数部分太长：

- 如果没有开启严格模式，这个值会被截断并产生一个 warning。
- 如果开启了严格模式，将会产生一个数据溢出的 error。

如果向一个数值类型列插入字符串，如果该字符串中包含非数值部分，TiDB 将这样做类型转换：

- 在严格模式下，没有以数字开头的字符串（即使是一个空字符串）不能被被用作数字值并会返回一个 error 或者是 warning。
- 以数字开头的字符串可以被转换，不过末尾的非数字部分会被截断。如果被截断的部分包含的不全是空格，在严格模式下这回产生一个 error 或者 warning。

默认情况下，如果计算的过程中发生了除数是 0 的现象将会得到一个 NULL 结果，并且不会有 warning 产生。通过设置适当的 SQL 模式，除以 0 的操作可以被限制。当设置 ERROR_FOR_DIVISION_BY_ZERO SQL 模式时，TiDB 的行为是：

- 如果设置了严格 SQL 模式，INSERT 和 UPDATE 的过程中如果发生了除以 0 的操作，正在进行的 INSERT 或者 UPDATE 操作会被禁止，并且会返回一个 error。
- 如果没有设置严格 SQL 模式，除以 0 的操作仅会返回一个 warning。

假设我们有如下的 SQL 语句：

```
INSERT INTO t SET i = 1/0;
```

不同的 SQL 模式将会导致不同的结果如下：

sql_mode 的值	结果
''	没有 warning，没有 error，i 被设为 NULL
strict	没有 warning，没有 error，i 被设为 NULL
ERROR_FOR_DIVISION_BY_ZERO	有 warning，没有 error，i 被设为 NULL
strict, ERROR_FOR_DIVISION_BY_ZERO	有 error，插入失败

2.4.16.4 数值修约

round() 函数的结果取决于他的参数是否是精确值：

- 如果参数是精确值，round() 函数将使用四舍五入的规则。
- 如果参数是一个近似值，round() 表达式的结果可能和 MySQL 不太一样。

```
SELECT ROUND(2.5), ROUND(25E-1);
```

```
+------------+--------------+
| ROUND(2.5) | ROUND(25E-1) |
+------------+--------------+
|          3 |            3 |
+------------+--------------+
1 row in set (0.00 sec)
```

向一个 DECIMAL 或者整数类型列插入数据时，round 的规则将采用 round half away from zero 的方式：

```
CREATE TABLE t (d DECIMAL(10,0));
```

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

```
INSERT INTO t VALUES(2.5),(2.5E0);
```

```
Query OK, 2 rows affected, 2 warnings (0.00 sec)
```

```
SELECT d FROM t;
```

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

2.4.17 集合运算

TiDB 支持三种集合运算：并集 (UNION)，差集 (EXCEPT) 和交集 (INTERSECT)。最小的集合单位是一个SELECT 语句。

2.4.17.1 并集 (UNION)

数学上，两个集合 A 和 B 的并集是含有所有属于 A 或属于 B 的元素。下面是一个 UNION 的例子：

```
select 1 union select 2;
+---+
| 1 |
+---+
| 2 |
| 1 |
+---+
2 rows in set (0.00 sec)
```

TiDB 支持 UNION ALL 和 UNION DISTINCT 并集，两者区别在于 UNION DISTINCT 会对并集结果去重复，而 UNION ↪ ALL 不会。TiDB 中默认使用 UNION DISTINCT。

```
create table t1 (a int);
create table t2 (a int);
insert into t1 values (1),(2);
insert into t2 values (1),(3);
```

UNION DISTINCT与 UNION ALL 的结果分别如下：

```
select * from t1 union distinct select * from t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows in set (0.00 sec)

select * from t1 union all select * from t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 1 |
| 3 |
+---+
4 rows in set (0.00 sec)
```

2.4.17.2 差集 (EXCEPT)

若 A 和 B 是集合，则 A 与 B 的差集是由所有属于 A 但不属于 B 的元素组成的集合。

```
select * from t1 except select * from t2;
+---+
| a |
+---+
| 2 |
+---+
1 rows in set (0.00 sec)
```

差集 (EXCEPT) 暂时不支持 EXCEPT ALL。

2.4.17.3 交集 (INTERSECT)

数学上，两个集合 A 和 B 的交集是含有所有既属于 A 又属于 B 的元素，而且没有其他元素的集合。

```
select * from t1 intersect select * from t2;
+---+
| a |
+---+
| 1 |
+---+
```

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

交集 (INTERSECT) 暂时不支持 INTERSECT ALL。交集 (INTERSECT) 的计算优先级大于差集 (EXCEPT) 和并集 (UNION)。

```
select * from t1 union all select * from t1 intersect select * from t2;
+---+
| a |
+---+
| 1 |
| 1 |
| 2 |
+---+
3 rows in set (0.00 sec)
```

2.4.17.4 括号优先

TiDB 支持使用括号修改集合运算的优先级，如同四则运算中先计算括号部分，集合运算也先计算括号内的部分。

```
(select * from t1 union all select * from t1) intersect select * from t2;
+---+
| a |
+---+
| 1 |
+---+
1 rows in set (0.00 sec)
```

2.4.17.5 与 Order By 和 Limit 结合

TiDB 支持单独为整个集合运算进行 ORDER BY 或者 LIMIT。

```
(select * from t1 union all select * from t1 intersect select * from t2) order by a limit 2;
+---+
| a |
+---+
| 1 |
| 1 |
+---+
2 rows in set (0.00 sec)
```

2.4.18 下推到 TiKV 的表达式列表

当 TiDB 从 TiKV 中读取数据的时候，TiDB 会尽量下推一些表达式运算到 TiKV 中，从而减少数据传输量以及 TiDB 单一节点的计算压力。本文将介绍 TiDB 已支持下推的表达式，以及如何禁止下推特定表达式。

2.4.18.1 已支持下推的表达式列表

表达式分类	具体操作
逻辑运算	AND (&&), OR (||), NOT (!)
比较运算	<, <=, =, != (<>), >, >=, <=>, IN(), IS NULL, LIKE, IS TRUE, IS FALSE, COALESCE()
数值运算	+, -, *, /, ABS(), CEIL(), CEILING(), FLOOR(), MOD()
控制流运算	CASE, IF(), IFNULL()
JSON 运算	JSON_TYPE(json_val), JSON_EXTRACT(json_doc, path[, path] ⋯), JSON_OBJECT(key, val[, key, val] ⋯), JSON_ARRAY([val[, val] ⋯]), JSON_MERGE(json_doc, json_doc[, json_doc] ⋯), JSON_SET(json_doc, path, val[, path, val] ⋯), JSON_INSERT(json_doc, path, val[, path, val] ⋯), JSON_REPLACE(json_doc, path, val[, path, val] ⋯), JSON_REMOVE(json_doc, path[, path] ⋯)
日期运算	DATE_FORMAT(), SYSDATE()
字符串函数	RIGHT()

2.4.18.2 禁止特定表达式下推

当已支持下推的表达式列表中的函数和运算符，或特定的数据类型（仅限ENUM 类型 和BIT 类型）的计算过程因下推而出现异常时，你可以使用黑名单功能禁止其下推，从而快速恢复 TiDB 业务。具体而言，你可以将函数名、运算符名，或数据列类型加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。具体方法，请参阅加入黑名单。

`mysql.expr_pushdown_blacklist` 的 schema 如下：

```
tidb> desc mysql.expr_pushdown_blacklist;
+------------+--------------+------+------+-------------------+-------+
| Field      | Type         | Null | Key  | Default           | Extra |
+------------+--------------+------+------+-------------------+-------+
| name       | char(100)    | NO   |      | NULL              |       |
| store_type | char(100)    | NO   |      | tikv,tiflash,tidb |       |
| reason     | varchar(200) | YES  |      | NULL              |       |
+------------+--------------+------+------+-------------------+-------+
3 rows in set (0.00 sec)
```

以上结果字段解释如下：

- `name`：禁止下推的函数名、运算符名或数据类型。
- `store_type`：用于指定希望禁止该函数、运算符或数据类型下推到哪些组件进行计算。组件可选 `tidb`、`tikv` 和 `tiflash`。`store_type` 不区分大小写，如果需要禁止向多个存储引擎下推，各个存储之间需用逗号隔开。
  - `store_type` 为 `tidb` 时表示在读取 TiDB 内存表时，是否允许该函数在其他 TiDB Server 上执行。
  - `store_type` 为 `tikv` 时表示是否允许该函数在 TiKV Server 的 Coprocessor 模块中执行。
  - `store_type` 为 `tiflash` 时表示是否允许该函数在 TiFlash Server 的 Coprocessor 模块中执行。
- `reason`：用于记录该函数被加入黑名单的原因。

> **注意：**
>
> tidb 是一种特殊的 store_type，其含义是 TiDB 内存表，比如：PERFORMANCE_SCHEMA.events_statements_summary_by_digest，属于系统表的一种，非特殊情况不用考虑这种存储引擎。

2.4.18.2.1 加入黑名单

执行以下步骤，可将一个或多函数名、运算符名或数据类型（仅限ENUM 类型 和BIT 类型）加入黑名单：

1. 向 mysql.expr_pushdown_blacklist 插入以下内容：
    - 希望禁止下推的函数名、运算符名或数据类型
    - 希望禁止下推的存储引擎
2. 执行 admin reload expr_pushdown_blacklist;。

2.4.18.2.2 移出黑名单

执行以下步骤，可将一个或多个函数名、运算符名或数据类型移出黑名单：

1. 从 mysql.expr_pushdown_blacklist 表中删除对应的函数名、运算符名或数据类型。
2. 执行 admin reload expr_pushdown_blacklist;。

2.4.18.2.3 黑名单使用示例

以下示例首先将函数 DATE_FORMAT()、运算符 > 及数据类型 BIT 加入黑名单，然后再将运算符 > 从黑名单中移出。

黑名单是否生效可以从 explain 结果中进行观察（参见如何理解 explain 结果）。

```
tidb> create table t(a int);
Query OK, 0 rows affected (0.06 sec)

tidb> explain select * from t where a < 2 and a > 2;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                      |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7         | 0.00     | root      |               | data:Selection_6
    ↪                    |
```

```
| └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2), lt(ssb_1.t.
    ↪ a, 2) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo     |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)

tidb> insert into mysql.expr_pushdown_blacklist values('date_format()', 'tikv',''), ('>','tikv','
    ↪ '), ('bit','tikv','');
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

tidb> admin reload expr_pushdown_blacklist;
Query OK, 0 rows affected (0.00 sec)

tidb> explain select * from t where a < 2 and a > 2;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                       |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| Selection_7             | 10000.00 | root      |               | gt(ssb_1.t.a, 2), lt(ssb_1.t.a
    ↪ , 2) |
| └─TableReader_6         | 10000.00 | root      |               | data:TableFullScan_5
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo     |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)

tidb> delete from mysql.expr_pushdown_blacklist where name = '>';
Query OK, 1 row affected (0.01 sec)

tidb> admin reload expr_pushdown_blacklist;
Query OK, 0 rows affected (0.00 sec)

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

```
    ↪ ------------------------+----------+-----------+--------------+---------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ------------------------+----------+-----------+--------------+---------------------------------+
    ↪
| Selection_8               | 0.00     | root      |               | lt(ssb_1.t.a, 2)
    ↪               |
| └─TableReader_7           | 0.00     | root      |               | data:Selection_6
    ↪               |
|   └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2)
    ↪               |
|     └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ------------------------+----------+-----------+--------------+---------------------------------+
    ↪
4 rows in set (0.00 sec)
```

注意：

- `admin reload expr_pushdown_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。
- 表达式黑名单功能在 v3.0.0 及以上版本中支持。
- 在 v3.0.3 及以下版本中，不支持将某些运算符的原始名称文本（如 “>”、“+” 和 “is null”）加入黑名单中，部分运算符在黑名单中需使用别名。已支持下推的表达式中，别名与原始名不同的运算符见下表（区分大小写）。

运算符原始名称	运算符别名
<	lt
>	gt
<=	le
>=	ge
=	eq
!=	ne
<>	ne
<=>	nulleq
|	bitor
&&	bitand
||	or
!	not

运算符原始名称	运算符别名
in	in
+	plus
-	minus
	mul
/	div
DIV	intdiv
IS NULL	isnull
IS TRUE	istrue
IS FALSE	isfalse

2.4.19 TiDB 特有的函数

本文档介绍 TiDB 特有的函数。

2.4.19.1 TIDB_BOUNDED_STALENESS

TIDB_BOUNDED_STALENESS 是 TiDB 的内部函数，用于指定一个时间范围。用法为 TIDB_BOUNDED_STALENESS(t1, t2)，其中 t1 和 t2 为时间范围的两端，支持使用日期时间和时间函数。

使用该函数，TiDB 会在指定的时间范围内选择一个合适的时间戳，该时间戳能保证所访问的副本上不存在开始于这个时间戳之前且还没有提交的相关事务，即能保证所访问的可用副本上执行读取操作而且不会被阻塞。

2.4.19.2 TIDB_DECODE_KEY

TIDB_DECODE_KEY 函数用于将 TiDB 编码的键输入解码为包含 _tidb_rowid 和 table_id 的 JSON 结构。你可以在一些系统表和日志输出中找到 TiDB 的编码键。

2.4.19.2.1 语法图

```
TableStmt ::=
    "TIDB_DECODE_KEY(" STR ")"
```

2.4.19.2.2 示例

以下示例中，表 t1 有一个隐藏的 rowid，该 rowid 由 TiDB 生成。语句中使用了 TIDB_DECODE_KEY 函数。结果显示，隐藏的 rowid 被解码后并输出，这是典型的非聚簇主键结果。

```
SELECT START_KEY, TIDB_DECODE_KEY(START_KEY) FROM information_schema.tikv_region_status WHERE
    ↪ table_name='t1' AND REGION_ID=2\G
```

```
*************************** 1. row ***************************
                START_KEY: 7480000000000000FF3B5F728000000000FF1DE3F10000000000FA
TIDB_DECODE_KEY(START_KEY): {"_tidb_rowid":1958897,"table_id":"59"}
1 row in set (0.00 sec)
```

以下示例中，表 t2 有一个复合聚簇主键。由 JSON 输出可知，输出结果的 handle 项中包含了主键部分两列的信息，即两列的名称和对应的值。

```
show create table t2\G
```

```
*************************** 1. row ***************************
       Table: t2
Create Table: CREATE TABLE `t2` (
  `id` binary(36) NOT NULL,
  `a` tinyint(3) unsigned NOT NULL,
  `v` varchar(512) DEFAULT NULL,
  PRIMARY KEY (`a`,`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.001 sec)
```

```
select * from information_schema.tikv_region_status where table_name='t2' limit 1\G
```

```
*************************** 1. row ***************************
               REGION_ID: 48
               START_KEY: 7480000000000000
                    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DF
                    ↪
                 END_KEY:
                TABLE_ID: 62
                 DB_NAME: test
              TABLE_NAME: t2
                IS_INDEX: 0
                INDEX_ID: NULL
              INDEX_NAME: NULL
           EPOCH_CONF_VER: 1
            EPOCH_VERSION: 38
            WRITTEN_BYTES: 0
               READ_BYTES: 0
         APPROXIMATE_SIZE: 136
         APPROXIMATE_KEYS: 479905
  REPLICATIONSTATUS_STATE: NULL
REPLICATIONSTATUS_STATEID: NULL
1 row in set (0.005 sec)
```

```
select tidb_decode_key('7480000000000000
    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DFFFF3830653635303
    ↪ ');
```

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

```
| tidb_decode_key('7480000000000000
    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DFFFF3830653635303
    ↪ ') |
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
| {"handle":{"a":"6","id":"c4038db2-d51c-11eb-8c75-80e65018a9be"},"table_id":62}
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.001 sec)
```

2.4.19.2.3 MySQL 兼容性

TIDB_DECODE_KEY 是 TiDB 特有的函数，和 MySQL 不兼容。

2.4.19.3 TIDB_DECODE_PLAN

TIDB_DECODE_PLAN 函数用于解码 TiDB 执行计划。你可以在慢查询日志中找到 TiDB 执行计划。

2.4.19.3.1 语法图

```
TableStmt ::=
    "TIDB_DECODE_PLAN(" STR ")"
```

2.4.19.3.2 示例

```
SELECT tidb_decode_plan('8
    ↪ QIYMAkzMV83CQEH8E85LjAOCWRhdGE6U2VsZWN0aW9uXzYJOTYwCXRpbWU6NzEzLjHCtXMsIGxvb3BzOjIsIGNvcF90YXNrOiB7bnVtOi
    ↪ ')\G
```

```
*************************** 1. row ***************************
  tidb_decode_plan('8
      ↪ QIYMAkzMV83CQEH8E85LjAOCWRhdGE6U2VsZWN0aW9uXzYJOTYwCXRpbWU6NzEzLjHCtXMsIGxvb3BzOjIsIGNvcF90YXNrOiB7bnVt
      ↪ :     id                    task         estRows    operator info
      ↪                            actRows    execution info
      ↪
      ↪ memory     disk
    TableReader_7          root         319.04     data:Selection_6                          960
        ↪         time:713.1µs, loops:2, cop_task: {num: 1, max: 568.5µs, proc_keys: 0, rpc_num
        ↪ : 1, rpc_time: 549.1µs, copr_cache_hit_ratio: 0.00}    3.99 KB    N/A
```

```
    └─Selection_6          cop[tikv]    319.04     lt(test.t.a, 10000)
      ↪ 960        tikv_task:{time:313.8µs, loops:960}
      ↪
      ↪ N/A        N/A
     └─TableFullScan_5     cop[tikv]    960        table:t, keep order:false, stats:pseudo
       ↪ 960        tikv_task:{time:153µs, loops:960}
       ↪
       ↪ N/A        N/A
```

2.4.19.3.3 MySQL 兼容性

TIDB_DECODE_PLAN 是 TiDB 特有的函数，和 MySQL 不兼容。

2.4.19.4 TIDB_IS_DDL_OWNER

TIDB_IS_DDL_OWNER 函数用于检查你连接的 TiDB 实例是否是 DDL Owner。DDL Owner 代表集群中所有其他节点执行 DDL 语句的 TiDB 实例。

2.4.19.4.1 语法图

```
TableStmt ::=
    "TIDB_IS_DDL_OWNER())"
```

2.4.19.4.2 示例

```
SELECT tidb_is_ddl_owner();
```

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

2.4.19.4.3 MySQL 兼容性

TIDB_IS_DDL_OWNER 是 TiDB 特有的函数，和 MySQL 不兼容。

2.4.19.4.4 另请参阅

- ADMIN SHOW DDL
- ADMIN CANCEL DDL

2.4.19.5 TIDB_PARSE_TSO

TIDB_PARSE_TSO 函数用于从 TiDB TSO 时间戳中提取物理时间戳。

TSO 指 Time Stamp Oracle，是 PD (Placement Driver) 为每个事务提供的单调递增的时间戳。

TSO 是一串数字，包含以下两部分：

- 一个物理时间戳
- 一个逻辑计数器

2.4.19.5.1 语法图

```
TableStmt ::=
    "TIDB_PARSE_TSO(" NUM ")"
```

2.4.19.5.2 示例

```
BEGIN;
SELECT TIDB_PARSE_TSO(@@tidb_current_ts);
ROLLBACK;
```

```
+-----------------------------------+
| TIDB_PARSE_TSO(@@tidb_current_ts) |
+-----------------------------------+
| 2021-05-26 11:33:37.776000        |
+-----------------------------------+
1 row in set (0.0012 sec)
```

以上示例使用 TIDB_PARSE_TSO 函数从 tidb_current_ts 会话变量提供的可用时间戳编号中提取物理时间戳。因为每个事务都会分配到时间戳，所以此函数在事务中运行。

2.4.19.5.3 MySQL 兼容性

TIDB_PARSE_TSO 是 TiDB 特有的函数，和 MySQL 不兼容。

2.4.19.5.4 另请参阅

- tidb_current_ts

2.4.19.6 TIDB_VERSION

TIDB_VERSION 函数用于获取当前连接的 TiDB 服务器版本和构建详细信息。向 GitHub 上提交 issue 时，你可使用此函数获取相关信息。

2.4.19.6.1 语法图

```
TableStmt ::=
    "TIDB_VERSION()"
```

2.4.19.6.2 示例

```
SELECT TIDB_VERSION()\G
```

```
*************************** 1. row ***************************
TIDB_VERSION(): Release Version: v5.1.0-alpha-13-gd5e0ed0aa-dirty
Edition: Community
Git Commit Hash: d5e0ed0aaed72d2f2dfe24e9deec31cb6cb5fdf0
Git Branch: master
UTC Build Time: 2021-05-24 14:39:20
GoVersion: go1.13
Race Enabled: false
TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306
Check Table Before Drop: false
1 row in set (0.00 sec)
```

2.4.19.6.3 MySQL 兼容性

TIDB_VERSION 是 TiDB 特有的函数，和 MySQL 不兼容。如果要求兼容 MySQL，可以使用 VERSION 获取版本信息，但结果不包含详细的构建信息。

2.4.19.7 TIDB_DECODE_SQL_DIGESTS

TIDB_DECODE_SQL_DIGESTS 函数用于在集群中查询一组 SQL Digest 所对应的 SQL 语句的归一化形式（即去除格式和参数后的形式）。函数接受 1 个或 2 个参数：

- digests：字符串类型，该参数应符合 JSON 字符串数组的格式，数组中的每个字符串应为一个 SQL Digest。
- stmtTruncateLength：可选参数，整数类型，用来限制返回结果中每条 SQL 语句的长度，超过指定的长度会被截断。0 表示不限制长度。

返回一个字符串，符合 JSON 字符串数组的格式，数组中的第 i 项为参数 digests 中的第 i 个元素所对应的语句。如果参数 digests 中的某一项不是一个有效的 SQL Digest 或系统无法查询到其对应的 SQL 语句，则返回结果中对应项为 null。如果指定了截断长度（stmtTruncateLength > 0），则返回结果中每条超过该长度的语句，保留前 stmtTruncateLength 个字符，并在尾部增加 "..." 后缀表示发生了截断。如果参数 digests 为 NULL，则函数的返回值为 NULL。

> 注意：

- 仅持有 PROCESS 权限的用户可以使用该函数。
- TIDB_DECODE_SQL_DIGESTS 执行时，TiDB 内部从 Statement Summary 一系列表中查询每个 SQL Digest 所对应的语句，因而并不能保证对任意 SQL Digest 都总是能查询到对应的语句，只有在集群中执行过的语句才有可能被查询到，且是否能查询到受 Statement Summary 表相关配置的影响。有关 Statement Summary 表的详细说明，参见 Statement Summary Tables。
- 该函数开销较大，在行数很多的查询中（比如在规模较大、比较繁忙的集群上查询 information_schema.cluster_tidb_trx 全表时）直接使用该函数可能导致查询运行时间较长。请谨慎使用。
  - 该函数开销大的原因是，其每次被调用时，都会在内部发起对 STATEMENTS_SUMMARY、STATEMENTS_SUMMARY_HISTORY、CLUSTER_STATEMENTS_SUMMARY 和 CLUSTER_STATEMENTS_SUMMARY_HISTORY 这几张表的查询，且其中涉及 UNION 操作。且该函数目前不支持向量化，即对于多行数据调用该函数时，对每行都会独立进行一次上述的查询。

2.4.19.7.1 语法图

```
DecodeSQLDigestsExpr ::=
    "TIDB_DECODE_SQL_DIGESTS" "(" digests ( "," stmtTruncateLength )? ")"
```

2.4.19.7.2 示例

```
set @digests = '["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
    ↪ b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821","
    ↪ e5796985ccafe2f71126ed6c0ac939ffa015a8c0744a24b7aee6d587103fd2f7"]';

select tidb_decode_sql_digests(@digests);
```

```
+------------------------------------+
| tidb_decode_sql_digests(@digests)  |
+------------------------------------+
| ["begin",null,"select * from `t`"] |
+------------------------------------+
1 row in set (0.00 sec)
```

上面的例子中，参数是一个包含 3 个 SQL Digest 的 JSON 数组，其对应的 SQL 语句分别为查询结果中给出的三项。但是其中第二条 SQL Digest 所对应的 SQL 语句未能从集群中找到，因而结果中的第二项为 null。

```
select tidb_decode_sql_digests(@digests, 10);
```

```
+---------------------------------------+
| tidb_decode_sql_digests(@digests, 10) |
+---------------------------------------+
| ["begin",null,"select * f..."]        |
```

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

上述调用指定了第二个参数（即截断长度）为 10，而查询结果中的第三条语句的长度大于 10，因而仅保留了前 10 个字符，并在尾部添加了 "..." 表示发生了截断。

2.4.19.7.3 MySQL 兼容性

TIDB_DECODE_SQL_DIGESTS 是 TiDB 特有的函数，和 MySQL 不兼容。

2.4.19.7.4 另请参阅

- Statement Summary Tables
- INFORMATION_SCHEMA.TIDB_TRX

2.4.19.8 TIDB_SHARD

TIDB_SHARD 函数用于创建一个 SHARD INDEX 来打散热点索引。SHARD INDEX 是一种以 TIDB_SHARD 函数为前缀的表达式索引。

2.4.19.8.1 SHARD INDEX

- 创建方式：

  使用 uk((tidb_shard(a)), a)) 为字段 a 创建一个 SHARD INDEX。当二级唯一索引 uk((tidb_shard(a)), a)) 的索引字段 a 上存在因单调递增或递减而产生的热点时，索引的前缀 tidb_shard(a) 会打散热点，从而提升集群可扩展性。

- 适用场景：

  - 二级唯一索引上 key 值存在单调递增或递减导致的写入热点，且该索引包含的列是整型。
  - 业务中 SQL 语句根据该二级索引的全部字段做等值查询，查询可以是单独的 SELECT，也可以是 UPDATE，DELETE 等产生的内部查询，等值查询包括 a = 1 或 a IN (1, 2, ...) 两种方式。

- 使用限制：

  - 非等值查询无法使用索引。
  - 查询条件中 AND 和 OR 混合且最外层是 AND 算子时无法使用 SHARD INDEX。
  - GROUP BY 无法使用 SHARD INDEX。
  - ORDER BY 无法使用 SHARD INDEX。
  - ON 子句无法使用 SHARD INDEX。
  - WHERE 子查询无法使用 SHARD INDEX。
  - SHARD INDEX 只能打散整型字段的唯一索引。
  - SHARD INDEX 联合索引可能失效。
  - SHARD INDEX 无法走 FastPlan 流程，影响优化器性能。
  - SHARD INDEX 无法使用执行计划缓存。

2.4.19.8.2 语法图

```
TIDBShardExpr ::=
    "TIDB_SHARD" "(" expr ")"
```

2.4.19.8.3 示例

- 使用 TIDB_SHARD 函数计算 SHARD 值

  以下示例说明如何使用 TIDB_SHARD 函数计算 12373743746 的 SHARD 值。

```
SELECT TIDB_SHARD(12373743746);
```

- 计算得出 SHARD 值为：

```
+-------------------------+
| TIDB_SHARD(12373743746) |
+-------------------------+
|                     184 |
+-------------------------+
1 row in set (0.00 sec)
```

- 使用 TIDB_SHARD 函数创建 SHARD INDEX

```
CREATE TABLE test(id INT PRIMARY KEY CLUSTERED, a INT, b INT, UNIQUE KEY uk((tidb_shard(a)),
    ↪  a));
```

2.4.19.8.4 MySQL 兼容性

TIDB_SHARD 是 TiDB 特有的函数，和 MySQL 不兼容。

2.5 聚簇索引

聚簇索引 (clustered index) 是 TiDB 从 v5.0 开始支持的特性，用于控制含有主键的表数据的存储方式。通过使用聚簇索引，TiDB 可以更好地组织数据表，从而提高某些查询的性能。有些数据库管理系统也将聚簇索引称为“索引组织表” (index-organized tables)。

目前 TiDB 中含有主键的表分为以下两类：

- NONCLUSTERED，表示该表的主键为非聚簇索引。在非聚簇索引表中，行数据的键由 TiDB 内部隐式分配的 _tidb_rowid 构成，而主键本质上是唯一索引，因此非聚簇索引表存储一行至少需要两个键值对，分别为
  - _tidb_rowid（键）- 行数据（值）
  - 主键列数据（键）- _tidb_rowid（值）
- CLUSTERED，表示该表的主键为聚簇索引。在聚簇索引表中，行数据的键由用户给定的主键列数据构成，因此聚簇索引表存储一行至少只要一个键值对，即

- 主键列数据（键）- 行数据（值）

> **注意：**
>
> TiDB 仅支持根据表的主键来进行聚簇操作。聚簇索引启用时，“主键”和“聚簇索引”两个术语在一些情况下可互换使用。主键指的是约束（一种逻辑属性），而聚簇索引描述的是数据存储的物理实现。

2.5.1 使用场景

相较于非聚簇索引表，聚簇索引表在以下几个场景中，性能和吞吐量都有较大优势：

- 插入数据时会减少一次从网络写入索引数据。
- 等值条件查询仅涉及主键时会减少一次从网络读取数据。
- 范围条件查询仅涉及主键时会减少多次从网络读取数据。
- 等值或范围条件查询仅涉及主键的前缀时会减少多次从网络读取数据。

另一方面，聚簇索引表也存在一定的劣势：

- 批量插入大量取值相邻的主键时，可能会产生较大的写热点问题。
- 当使用大于 64 位的数据类型作为主键时，可能导致表数据需要占用更多的存储空间。该现象在存在多个二级索引时尤为明显。

2.5.2 使用方法

2.5.2.1 创建聚簇索引表

从 TiDB 版本 5.0 开始，要指定一个表的主键是否使用聚簇索引，可以在 `CREATE TABLE` 语句中将 `CLUSTERED` 或者 `NONCLUSTERED` 非保留关键字标注在 `PRIMARY KEY` 后面，例如：

```
CREATE TABLE t (a BIGINT PRIMARY KEY CLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT PRIMARY KEY NONCLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT KEY CLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT KEY NONCLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) CLUSTERED);
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) NONCLUSTERED);
```

注意，列定义中的 `KEY` 和 `PRIMARY KEY` 含义相同。

此外，TiDB 支持使用可执行的注释语法指定聚簇索引属性：

```
CREATE TABLE t (a BIGINT PRIMARY KEY /*T![clustered_index] CLUSTERED */, b VARCHAR(255));
CREATE TABLE t (a BIGINT PRIMARY KEY /*T![clustered_index] NONCLUSTERED */, b VARCHAR(255));
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /*T![clustered_index] CLUSTERED */);
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /*T![clustered_index] NONCLUSTERED */
    ↪ );
```

对于未显式指定该关键字的语句，默认行为受系统变量 @@global.tidb_enable_clustered_index 影响。该变量有三个取值：

- OFF 表示所有主键默认使用非聚簇索引。
- ON 表示所有主键默认使用聚簇索引。
- INT_ONLY 此时的行为受配置项 alter-primary-key 控制。如果该配置项取值为 true，则所有主键默认使用非聚簇索引；如果该配置项取值为 false，则由单个整数类型的列构成的主键默认使用聚簇索引，其他类型的主键默认使用非聚簇索引。

系统变量 @@global.tidb_enable_clustered_index 本身的默认值为 INT_ONLY。

2.5.2.2 添加、删除聚簇索引

目前 TiDB 不支持在建表之后添加或删除聚簇索引，也不支持聚簇索引和非聚簇索引的互相转换。例如：

```
ALTER TABLE t ADD PRIMARY KEY(b, a) CLUSTERED; -- 暂不支持
ALTER TABLE t DROP PRIMARY KEY;      -- 如果主键为聚簇索引，则不支持
ALTER TABLE t DROP INDEX `PRIMARY`; -- 如果主键为聚簇索引，则不支持
```

2.5.2.3 添加、删除非聚簇索引

TiDB 支持在建表之后添加或删除非聚簇索引。此时可以选择显式指定 NONCLUSTERED 关键字或省略关键字：

```
ALTER TABLE t ADD PRIMARY KEY(b, a) NONCLUSTERED;
ALTER TABLE t ADD PRIMARY KEY(b, a); -- 不指定关键字，则为非聚簇索引
ALTER TABLE t DROP PRIMARY KEY;
ALTER TABLE t DROP INDEX `PRIMARY`;
```

2.5.2.4 查询主键是否为聚簇索引

可通过以下方式来确定一张表的主键是否使用了聚簇索引：

- 执行语句 SHOW CREATE TABLE。
- 执行语句 SHOW INDEX FROM。
- 查询系统表 information_schema.tables 中的 TIDB_PK_TYPE 列。

通过 SHOW CREATE TABLE 查看，PRIMARY KEY 的属性可能为 CLUSTERED 或 NONCLUSTERED：

```
mysql> SHOW CREATE TABLE t;
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
```

```
+--
    ↪ -----+-----------------------------------------------------------------------------------------------
    ↪
| t     | CREATE TABLE `t` (
  `a` bigint(20) NOT NULL,
  `b` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+-----------------------------------------------------------------------------------------------
    ↪
1 row in set (0.01 sec)
```

通过 `SHOW INDEX FROM` 查看，`Clustered` 一列可能的结果为 `Yes` 或 `No`：

```
mysql> SHOW INDEX FROM t;
+--
    ↪ -----+------------+----------+--------------+-------------+-----------+-------------+----------+--------+
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
    ↪ Clustered |
+--
    ↪ -----+------------+----------+--------------+-------------+-----------+-------------+----------+--------+
    ↪
| t     |          0 | PRIMARY  |            1 | a           | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES     | NULL       | YES
    ↪       |
+--
    ↪ -----+------------+----------+--------------+-------------+-----------+-------------+----------+--------+
    ↪
1 row in set (0.01 sec)
```

查询 `information_schema.tables` 系统表中的 `TIDB_PK_TYPE` 列，可能的结果为 `CLUSTERED` 或 `NONCLUSTERED`：

```
mysql> SELECT TIDB_PK_TYPE FROM information_schema.tables WHERE table_schema = 'test' AND
    ↪ table_name = 't';
+--------------+
| TIDB_PK_TYPE |
+--------------+
| CLUSTERED    |
+--------------+
1 row in set (0.03 sec)
```

2.5.3 限制

目前 TiDB 的聚簇索引具有以下几类限制：

- 明确不支持且没有支持计划的使用限制：
  - 不支持与SHARD_ROW_ID_BITS 一起使用；PRE_SPLIT_REGIONS 在聚簇索引表上不生效。
  - 不支持对聚簇索引表进行降级。如需降级，请使用逻辑备份工具迁移数据。
- 尚未支持，但未来有计划支持的使用限制：
  - 尚未支持通过 ALTER TABLE 语句增加、删除、修改聚簇索引。
- 特定版本的限制：
  - 在 v5.0 版本中，聚簇索引不支持与 TiDB Binlog 一起使用。开启 TiDB Binlog 后，TiDB 只允许创建单个整数列作为主键的聚簇索引；已创建的聚簇索引表的数据插入、删除和更新动作不会通过 TiDB Binlog 同步到下游。如需同步聚簇索引表，请升级至 v5.1 版本或使用 TiCDC。

开启 TiDB Binlog 之后，要创建的聚簇索引如果不是由单个整数列构成，会报以下错误：

```
mysql> CREATE TABLE t (a VARCHAR(255) PRIMARY KEY CLUSTERED);
ERROR 8200 (HY000): Cannot create clustered index table when the binlog is ON
```

与 SHARD_ROW_ID_BITS 一起使用时会报以下错误：

```
mysql> CREATE TABLE t (a VARCHAR(255) PRIMARY KEY CLUSTERED) SHARD_ROW_ID_BITS = 3;
ERROR 8200 (HY000): Unsupported shard_row_id_bits for table with primary key as row id
```

2.5.4 兼容性

2.5.4.1 升降级兼容性

TiDB 支持对聚簇索引表的升级兼容，但不支持降级兼容，即高版本 TiDB 聚簇索引表的数据在低版本 TiDB 上不可用。

聚簇索引在 TiDB v3.0 和 v4.0 中已完成部分支持，当表中存在单个整数列作为主键时默认启用，即：

- 表设置了主键
- 主键只有一列
- 主键的数据类型为整数类型

TiDB v5.0 完成了所有类型主键的支持，但默认行为与 TiDB v3.0 和 v4.0 保持一致。要修改默认行为，请设置系统变量 @@tidb_enable_clustered_index 为 ON 或 OFF。

2.5.4.2 MySQL 兼容性

TiDB 支持使用可执行注释的语法来包裹 CLUSTERED 或 NONCLUSTERED 关键字，且 SHOW CREATE TABLE 的结果均包含 TiDB 特有的可执行注释，这些注释在 MySQL 或低版本的 TiDB 中会被忽略。

2.5.4.3 TiDB 数据迁移工具兼容性

聚簇索引仅与 v5.0 及以后版本的以下数据迁移工具兼容：

- 备份与恢复工具 BR、 Dumpling、 TiDB Lightning。
- 数据迁移和同步工具 DM、 TiCDC。

v5.0 的 BR 不能通过备份恢复将非聚簇索引表转换成聚簇索引表，反之亦然。

2.5.4.4 与 TiDB 其他特性的兼容性

在非单整数列作为主键的表中，从非聚簇索引变为聚簇索引之后，在 v5.0 之前版本的 TiDB 能够执行的 SPLIT TABLE BY/BETWEEN 语句在 v5.0 及以后版本的 TiDB 上不再可用，原因是行数据键的构成发生了变化。在聚簇索引表上执行 SPLIT TABLE BY/BETWEEN 时需要依据主键列指定值，而不是指定一个整数值。例如：

```
mysql> create table t (a int, b varchar(255), primary key(a, b) clustered);
Query OK, 0 rows affected (0.01 sec)

mysql> split table t between (0) and (1000000) regions 5;
ERROR 1105 (HY000): Split table region lower value count should be 2

mysql> split table t by (0), (50000), (100000);
ERROR 1136 (21S01): Column count doesn't match value count at row 0

mysql> split table t between (0, 'aaa') and (1000000, 'zzz') regions 5;
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  4 |                    1 |
+--------------------+----------------------+
1 row in set (0.00 sec)

mysql> split table t by (0, ''), (50000, ''), (100000, '');
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  3 |                    1 |
+--------------------+----------------------+
1 row in set (0.01 sec)
```

AUTO_RANDOM 属性只能在聚簇索引表上使用。在非聚簇索引上使用 AUTO_RANDOM 会报以下错误：

```
mysql> create table t (a bigint primary key nonclustered auto_random);
ERROR 8216 (HY000): Invalid auto random: column a is not the integer primary key, or the primary
    ↪ key is nonclustered
```

2.6 约束

TiDB 支持的约束与 MySQL 的基本相同。

2.6.1 非空约束

TiDB 支持的非空约束规则与 MySQL 支持的一致。例如：

```
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 age INT NOT NULL,
 last_login TIMESTAMP
);
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,123,NOW());
```

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

```
INSERT INTO users (id,age,last_login) VALUES (NULL,NULL,NOW());
```

```
ERROR 1048 (23000): Column 'age' cannot be null
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,123,NULL);
```

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

- 第一条 INSERT 语句成功，因为对于定义为 AUTO_INCREMENT 的列，允许 NULL 作为其特殊值。TiDB 将为其分配下一个自动值。
- 第二条 INSERT 语句失败，因为 age 列被定义为 NOT NULL。
- 第三条 INSERT 语句成功，因为 last_login 列没有被明确地指定为 NOT NULL。默认允许 NULL 值。

2.6.2 CHECK 约束

TiDB 会解析并忽略 CHECK 约束。该行为与 MySQL 5.7 的相兼容。

示例如下：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username),
 CONSTRAINT min_username_length CHECK (CHARACTER_LENGTH(username) >=4)
);
INSERT INTO users (username) VALUES ('a');
SELECT * FROM users;
```

2.6.3 唯一约束

在 TiDB 的乐观事务中，默认会对唯一约束进行惰性检查。通过在事务提交时再进行批量检查，TiDB 能够减少网络开销、提升性能。例如：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');
```

默认的悲观事务模式下：

```
BEGIN;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'username'
```

乐观事务模式下且 tidb_constraint_check_in_place=0：

```
BEGIN OPTIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
Query OK, 0 rows affected (0.00 sec)
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
INSERT INTO users (username) VALUES ('steve'),('elizabeth');
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
COMMIT;
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'username'
```

在乐观事务的示例中，唯一约束的检查推迟到事务提交时才进行。由于 bill 值已经存在，这一行为导致了重复键错误。

你可通过设置 tidb_constraint_check_in_place 为 1 停用此行为（该变量设置对悲观事务无效，悲观事务始终在语句执行时检查约束）。当 tidb_constraint_check_in_place 设置为 1 时，则会在执行语句时就对唯一约束进行检查。例如：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
```

```
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');
```

```
SET tidb_constraint_check_in_place = 1;
```

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

```
BEGIN OPTIMISTIC;
```

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

```
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'username'
..
```

第一条 INSERT 语句导致了重复键错误。这会造成额外的网络通信开销，并可能降低插入操作的吞吐量。

2.6.4 主键约束

与 MySQL 行为一样，主键约束包含了唯一约束，即创建了主键约束相当于拥有了唯一约束。此外，TiDB 其他的主键约束规则也与 MySQL 相似。例如：

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
CREATE TABLE t2 (a INT NULL PRIMARY KEY);
```

```
ERROR 1171 (42000): All parts of a PRIMARY KEY must be NOT NULL; if you need NULL in a key, use
    ↪ UNIQUE instead
```

```
CREATE TABLE t3 (a INT NOT NULL PRIMARY KEY, b INT NOT NULL PRIMARY KEY);
```

```
ERROR 1068 (42000): Multiple primary key defined
```

```
CREATE TABLE t4 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b));
```

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

分析：

- 表 t2 创建失败，因为定义为主键的列 a 不能允许 NULL 值。
- 表 t3 创建失败，因为一张表只能有一个主键。
- 表 t4 创建成功，因为虽然只能有一个主键，但 TiDB 支持定义一个多列组合作为复合主键。

除上述规则外，TiDB 目前仅支持对 NONCLUSTERED 的主键进行添加和删除操作。例如：

```
CREATE TABLE t5 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b) CLUSTERED);
ALTER TABLE t5 DROP PRIMARY KEY;
```

```
ERROR 8200 (HY000): Unsupported drop primary key when the table is using clustered index
```

```
CREATE TABLE t5 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b) NONCLUSTERED);
ALTER TABLE t5 DROP PRIMARY KEY;
```

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

要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

2.6.5 外键约束

> **注意：**
>
> TiDB 仅部分支持外键约束功能。

TiDB 支持创建外键约束。例如：

```
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 doc JSON
);
CREATE TABLE orders (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 user_id INT NOT NULL,
 doc JSON,
 FOREIGN KEY fk_user_id (user_id) REFERENCES users(id)
);
```

```
SELECT table_name, column_name, constraint_name, referenced_table_name, referenced_column_name
FROM information_schema.key_column_usage WHERE table_name IN ('users', 'orders');
```

```
+------------+-------------+-----------------+-----------------------+------------------------+
| table_name | column_name | constraint_name | referenced_table_name | referenced_column_name |
+------------+-------------+-----------------+-----------------------+------------------------+
```

```
| users      | id          | PRIMARY         | NULL                 | NULL                   |
| orders     | id          | PRIMARY         | NULL                 | NULL                   |
| orders     | user_id     | fk_user_id      | users                | id                     |
+------------+-------------+-----------------+----------------------+------------------------+
3 rows in set (0.00 sec)
```

TiDB 也支持使用 ALTER TABLE 命令来删除外键 (DROP FOREIGN KEY) 和添加外键 (ADD FOREIGN KEY)：

```
ALTER TABLE orders DROP FOREIGN KEY fk_user_id;
ALTER TABLE orders ADD FOREIGN KEY fk_user_id (user_id) REFERENCES users(id);
```

2.6.5.1 注意

- TiDB 支持外键是为了在将其他数据库迁移到 TiDB 时，不会因为此语法报错。但是，TiDB 不会在 DML 语句中对外键进行约束检查。例如，即使 users 表中不存在 id=123 的记录，下列事务也能提交成功：

  ```
  START TRANSACTION;
  INSERT INTO orders (user_id, doc) VALUES (123, NULL);
  COMMIT;
  ```

2.7 SQL 模式

TiDB 服务器采用不同 SQL 模式来操作，且不同客户端可以应用不同模式。SQL 模式定义 TiDB 支持哪些 SQL 语法及执行哪种数据验证检查。

TiDB 启动之后采用 SET [SESSION | GLOBAL] sql_mode='modes'设置 SQL 模式。设置 GLOBAL 级别的 SQL 模式时用户需要有 SUPER 权限，并且只会影响到从设置 SQL 模式开始后续新建立的连接 (注：老连接不受影响)。SESSION 级别的 SQL 模式的变化只会影响当前的客户端。

Modes 是用逗号（‘,’）间隔开的一系列不同的模式。使用 SELECT @@sql_mode 语句查询当前 SQL 模式，SQL 模式默认值：ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION。

2.7.1 重要的 sql_mode 值

- ANSI: 符合标准 SQL，对数据进行校验，如果不符合定义类型或长度，对数据类型调整或截断保存，且返回 warning 警告。
- STRICT_TRANS_TABLES: 严格模式，对数据进行严格校验，当数据出现错误时，插入到表中，并且返回错误。
- TRADITIONAL: 采用此模式使 TiDB 的行为像“传统” SQL 数据库系统，当在列中插入不正确的值时“给出错误而不是警告”，一旦发现错误立即放弃 INSERT/UPDATE。

2.7.2 SQL mode 列表，如下

名称	含义
PIPES_AS_CONCAT	将“||”视为字符串连接操作符(+)(同CONCAT())，而不视为OR(支持)

名称	含义
ANSI_QUOTES	将"视为识别符，如果启用ANSI_QUOTES，只单引号内的会被认为是String Liter-als，双引号被解释为识别符，因此不能用双引号来引用字符串（支持）

名称	含义
IGNORE_SPACE	若开启该模式，系统忽略空格。例如：“user”和“user “是相同的（支持）

名称	含义
ONLY_FULL_GROUP_BY	如果 GROUP BY 出现的列并没有在 SELECT，HAVING，ORDER BY 中出现，此 SQL 不合法，因为不在 GROUP BY 中的列被查询展示出来不符合正常现象 (支持)

名称	含义
NO_UNSIGNED_SUBTRACTION	在减运算中，如果某个操作数没有符号，不要将结果标记为 UNSIGNED（支持）
NO_DIR_IN_CREATE	创建表时，忽视所有 INDEX DIRECTORY 和 DATA DIRECTORY 指令，该选项仅对从复制服务器有用（仅语法支持）

名称	含义
NO_KEY_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_FIELD_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_TABLE_OPTIONS	使用 SHOW CRE-ATE TABLE 时不会输出 MySQL 特有的语法部分，如 EN-GINE，使用 mysql-dump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_AUTO_VALUE_ON_ZERO	若启用该模式，在 AUTO_INCREMENT 列的处理传入的值是 0 或者具体数值时系统直接将该值写入此列，传入 NULL 时系统自动生成下一个序列号（支持）
NO_BACKSLASH_ESCAPES	若启用该模式，\ 反斜杠符号仅代表它自己（支持）

名称	含义
STRICT_TRANS_TABLES	对于事务存储引擎启用严格模式，insert非法值之后，回滚整条语句（支持）
STRICT_ALL_TABLES	对于事务型表，写入非法值之后，回滚整个事务语句（支持）

名称	含义
NO_ZERO_IN_DATE	在严格模式，不接受月或日部分为 0 的日期。如果使用 IGNORE 选项，我们为类似的日期插入’0000-00-00’。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
NO_ZERO_DATE	在严格模式，不要将 '0000-00-00' 做为合法日期。你仍然可以用 IGNORE 选项插入零日期。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
ALLOW_INVALID_DATES	不检查全部日期的合法性，仅检查月份值在 1 到 12 及日期值在 1 到 31 之间，仅适用于 DATE 和 DATA-TIME 列，TIMES-TAMP 列需要全部检查其合法性（支持）

名称	含义
ERROR_FOR_DIVISION_BY_ZERO	若启用该模式，在 INSERT 或 UPDATE 过程中，被除数为 0 值时，系统产生错误若未启用该模式，被除数为 0 值时，系统产生警告，并用 NULL 代替（支持）

名称	含义
NO_AUTO_CREATE_USER	防止 GRANT 自动创建新用户，但指定密码除外（支持）

名称	含义
HIGH_NOT_PRECEDENCE	NOT 操作符的优先级是表达式。例如：NOT a BETWEEN b AND c 被解释为 NOT (a BETWEEN b AND c)。在部份旧版本 MySQL 中，表达式被解释为 (NOT a) BETWEEN b AND c (支持)

名称	含义
NO_ENGINE_SUBSTITUTION	如果需要的存储引擎被禁用或未编译，可以防止自动替换存储引擎（仅语法支持）
PAD_CHAR_TO_FULL_LENGTH	若启用该模式，系统对于 CHAR 类型不会截断尾部空格（仅语法支持。该模式在 MySQL 8.0 中已废弃。）

名称	含义
REAL_AS_FLOAT	将 REAL 视为 FLOAT 的同义词，而不是 DOUBLE 的同义词（支持）
POSTGRESQL	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）
MSSQL	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）

名称	含义
DB2	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）
MAXDB	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER（支持）
MySQL32	等同于 NO_FIELD_OPTIONS、HIGH_NOT_PRECEDENCE（仅语法支持）
MYSQL40	等同于 NO_FIELD_OPTIONS、HIGH_NOT_PRECEDENCE（仅语法支持）

名称	含义
ANSI	等同于 REAL_AS_FLOAT、PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE (仅语法支持)
TRADITIONAL	等同于 STRICT_TRANS_TABLES、STRICT_ALL_TABLES、NO_ZERO_IN_DATE、NO_ZERO_DATE、ERROR_FOR_DIVISION_BY_ZERO、NO_AUTO_CREATE_USER (仅语法支持)
ORACLE	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER (仅语法支持)

2.8 表属性

表属性是 TiDB 从 5.3.0 版本开始引入的新特性，用于为表或分区添加特定的属性，以对表或分区执行相应属性对应的操作，例如可以利用表属性控制 Region 的合并。

注意：

- 目前 TiDB 仅支持为表或分区添加 merge_option 属性，用于控制 Region 合并。该属性仅能处理部分热点问题。如需了解更多的热点问题处理相关内容，请参阅 TiDB 热点问题处理。
- 当使用 TiDB Binlog 或 TiCDC 进行同步或者使用 BR 进行增量备份时，同步和备份会跳过设置表属性的 DDL 语句。如需在下游或者备份集群使用表属性，需要在下游或者备份集群手动执行该 DDL 语句以设置表属性。

2.8.1 使用方法

表属性为 key=value 的形式，多个属性需要用逗号分隔。具体示例如下，其中 t 为所要修改的表名，p 为所要修改的分区名，[]内部为可选项。

- 设置表或分区属性

```
ALTER TABLE t [PARTITION p] ATTRIBUTES [=] 'key=value[, key1=value1...]';
```

- 重置表或分区属性

```
ALTER TABLE t [PARTITION p] ATTRIBUTES [=] DEFAULT;
```

- 查看全部表及分区属性

```
SELECT * FROM information_schema.attributes;
```

- 查看某一张表或分区配置的属性

```
SELECT * FROM information_schema.attributes WHERE id='schema/t[/p]';
```

- 查看拥有某属性的所有表及分区

```
SELECT * FROM information_schema.attributes WHERE attributes LIKE '%key%';
```

2.8.2 覆盖关系

为分区表配置的属性会对表的所有分区生效。一种例外情况是，如果分区表和分区都配置了相同属性但属性值不同，分区属性将覆盖分区表属性。例如，当分区表 t 配置属性 key=value，同时分区 p 配置属性 key=value1 时：

```
ALTER TABLE t ATTRIBUTES[=]'key=value';
ALTER TABLE t PARTITION p ATTRIBUTES[=]'key=value1';
```

分区 p 实际生效的属性为 key=value1。

2.8.3 使用表属性控制 Region 合并

2.8.3.1 使用场景

当写入或读取数据存在热点时，可以使用表属性控制 Region 合并，通过为表或分区添加 `merge_option` 属性，将其设置为 deny 来解决。以下介绍了两种使用场景。

2.8.3.1.1 新建表或分区的写入热点问题

在对某张新建表或某个新建分区写入数据存在热点问题时，通常需要使用分裂打散 Region 的操作避免写入热点，但由于新建表或分区的分裂操作实际产生的是空 Region，如果分裂打散操作距离写入存在一定时间间隔，则 Region 可能会被合并，从而导致无法真正规避写入热点问题。此时可以为表或分区添加 `merge_option` 属性，设置为 deny 来解决问题。

2.8.3.1.2 只读场景下周期性读热点问题

在只读场景下，如果是通过手动分裂 Region 缓解某张表或分区的周期性读热点问题，且不希望热点消失后手动分裂的 Region 被合并。此时可以为表或分区添加 `merge_option` 属性，设置为 deny 来解决问题。

2.8.3.2 使用方法

使用方法如下，其中 `t` 为所要修改的表名，`p` 为所要修改的分区名。

- 禁止属于某个表的 Region 被合并

```
ALTER TABLE t ATTRIBUTES 'merge_option=deny';
```

- 允许属于某个表的 Region 被合并

```
ALTER TABLE t ATTRIBUTES 'merge_option=allow';
```

- 重置某个表的属性

```
ALTER TABLE t ATTRIBUTES DEFAULT;
```

- 禁止属于某个分区的 Region 被合并

```
ALTER TABLE t PARTITION p ATTRIBUTES 'merge_option=deny';
```

- 允许属于某个分区的 Region 被合并

```
ALTER TABLE t PARTITION p attributes 'merge_option=allow';
```

- 查看所有配置了 `merge_option` 属性的表或分区

```
SELECT * FROM information_schema.attributes WHERE attributes LIKE '%merge_option%';
```

2.8.3.3 覆盖关系

```
ALTER TABLE t ATTRIBUTES 'merge_option=deny';
ALTER TABLE t PARTITION p ATTRIBUTES 'merge_option=allow';
```

同时配置上述两个属性时，实际分区 p 的 Region 可以被合并。当分区的属性被重置时，分区 p 则会继承表 t 的属性，Region 无法被合并。

> 注意：
>
> - 如果目前只存在分区表的属性，即使配置 merge_option=allow，分区也会默认按照实际分区数量切分成多个 Region。如需合并所有 Region，则需要重置该分区表的属性。
> - 使用该属性需要注意 PD 的参数split-merge-interval 的配置。如果没有配置 merge_option，Region 在超过 split-merge-interval 指定的时间后满足条件即可合并。如果配置了 merge_option，则超过指定时间后会根据 merge_option 的配置情况再决定是否可以合并。

2.9 事务

2.9.1 TiDB 事务概览

TiDB 支持分布式事务，提供乐观事务与悲观事务两种事务模式。TiDB 3.0.8 及以后版本，TiDB 默认采用悲观事务模式。

本文主要介绍涉及事务的常用语句、显式/隐式事务、事务的隔离级别和惰性检查，以及事务大小的限制。

常用的变量包括autocommit、 tidb_disable_txn_auto_retry、 tidb_retry_limit 以及tidb_txn_mode。

> 注意：
>
> 变量tidb_disable_txn_auto_retry 和tidb_retry_limit 仅适用于乐观事务，不适用于悲观事务。

2.9.1.1 常用事务语句

2.9.1.1.1 开启事务

要显式地开启一个新事务，既可以使用BEGIN 语句，也可以使用START TRANSACTION 语句，两者效果相同。

语法：

```
BEGIN;
```

```
START TRANSACTION;
```

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

```
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY;
```

如果执行以上语句时，当前 Session 正处于一个事务的中间过程，那么系统会先自动提交当前事务，再开启一个新的事务。

> **注意：**
>
> 与 MySQL 不同的是，TiDB 在执行完上述语句后即会获取当前数据库快照，而 MySQL 的 `BEGIN` 和 `START TRANSACTION` 是在开启事务后的第一个从 InnoDB 读数据的 `SELECT` 语句（非 `SELECT FOR UPDATE`）后获取快照，`START TRANSACTION WITH CONSISTENT SNAPSHOT` 是语句执行时获取快照。因此，TiDB 中的 `BEGIN`、`START TRANSACTION` 和 `START TRANSACTION WITH CONSISTENT SNAPSHOT` 都等效为 MySQL 中的 `START TRANSACTION WITH CONSISTENT SNAPSHOT`。

2.9.1.1.2 提交事务

`COMMIT` 语句用于提交 TiDB 在当前事务中进行的所有修改。

语法：

```
COMMIT;
```

> **建议：**
>
> 启用乐观事务前，请确保应用程序可正确处理 `COMMIT` 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

2.9.1.1.3 回滚事务

`ROLLBACK` 语句用于回滚并撤销当前事务的所有修改。

语法：

```
ROLLBACK;
```

如果客户端连接中止或关闭，也会自动回滚该事务。

2.9.1.2 自动提交

为满足 MySQL 兼容性的要求，在默认情况下，TiDB 将在执行语句后立即进行 autocommit（自动提交）。

举例：

```
mysql> CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  pad1 VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
| 1            |
+--------------+
1 row in set (0.00 sec)

mysql> INSERT INTO t1 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM t1;
+----+------+
| id | pad1 |
+----+------+
|  1 | test |
+----+------+
1 row in set (0.00 sec)
```

以上示例中，ROLLBACK 语句没产生任何效果。由于 INSERT 语句是在自动提交的情况下执行的，等同于以下单语句事务：

```
START TRANSACTION;
INSERT INTO t1 VALUES (1, 'test');
COMMIT;
```

如果已显式地启动事务，则不适用自动提交。以下示例，ROLLBACK 语句成功撤回了 INSERT 语句：

```
mysql> CREATE TABLE t2 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  pad1 VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.10 sec)
```

```
mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
| 1            |
+--------------+
1 row in set (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t2 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

autocommit 是一个系统变量，可以基于 Session 或 Global 进行修改。

举例：

```
SET autocommit = 0;
```

```
SET GLOBAL autocommit = 0;
```

2.9.1.3 显式事务和隐式事务

> **注意：**
>
> 有些语句是隐式提交的。例如，执行 [BEGIN|START TRANCATION] 语句时，TiDB 会隐式提交上一个事务，并开启一个新的事务以满足 MySQL 兼容性的需求。详情参见 implicit commit。

TiDB 可以显式地使用事务（通过 [BEGIN|START TRANSACTION]/COMMIT 语句定义事务的开始和结束）或者隐式地使用事务 (SET autocommit = 1)。

在自动提交状态下，使用 [BEGIN|START TRANSACTION] 语句会显式地开启一个事务，同时也会禁用自动提交，使隐式事务变成显式事务。直到执行 COMMIT 或 ROLLBACK 语句时才会恢复到此前默认的自动提交状态。

对于 DDL 语句，会自动提交并且不能回滚。如果运行 DDL 的时候，正在一个事务的中间过程中，会先自动提交当前事务，再执行 DDL。

2.9.1.4 惰性检查

执行 DML 语句时，乐观事务默认不会检查主键约束或唯一约束，而是在 COMMIT 事务时进行这些检查。

举例：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
INSERT INTO t1 VALUES (1);
BEGIN OPTIMISTIC;
INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
INSERT INTO t1 VALUES (2);
COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
```

```
mysql> CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.10 sec)

mysql> INSERT INTO t1 VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> BEGIN OPTIMISTIC;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (2);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
mysql> SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

惰性检查优化通过批处理约束检查并减少网络通信来提升性能。可以通过设置 tidb_constraint_check_in_place = TRUE 禁用该行为。

> 注意：
>
> - 本优化仅适用于乐观事务。

- 本优化仅对普通的 INSERT 语句生效，对 INSERT IGNORE 和 INSERT ON DUPLICATE KEY UPDATE 不会生效。

2.9.1.5 语句回滚

TiDB 支持语句执行失败后的原子性回滚。如果语句报错，则所做的修改将不会生效。该事务将保持打开状态，并且在发出 COMMIT 或 ROLLBACK 语句之前可以进行其他修改。

```
CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
BEGIN;
INSERT INTO test VALUES (1);
INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
INSERT INTO test VALUES (3);
COMMIT;
SELECT * FROM test;
```

```
mysql> CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.09 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO test VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
ERROR 1146 (42S02): Table 'test.tset' doesn't exist
mysql> INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
mysql> INSERT INTO test VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM test;
+----+
| id |
+----+
|  1 |
|  3 |
+----+
2 rows in set (0.00 sec)
```

以上例子中，INSERT 语句执行失败之后，事务保持打开状态。最后的 INSERT 语句执行成功，并且提交了修改。

2.9.1.6 事务限制

由于底层存储引擎的限制，TiDB 要求单行不超过 6 MB。可以将一行的所有列根据类型转换为字节数并加和来估算单行大小。

TiDB 同时支持乐观事务与悲观事务，其中乐观事务是悲观事务的基础。由于乐观事务是先将修改缓存在私有内存中，因此，TiDB 对于单个事务的容量做了限制。

TiDB 中，单个事务的总大小默认不超过 100 MB，这个默认值可以通过配置文件中的配置项 txn-total-size-limit 进行修改，最大支持 1 TB。单个事务的实际大小限制还取决于服务器剩余可用内存的大小，执行事务时 TiDB 进程的内存消耗相对于事务大小会存在一定程度的放大，最大可能达到提交事务大小的 2 到 3 倍以上。

在 4.0 以前的版本，TiDB 限制了单个事务的键值对的总数量不超过 30 万条，从 4.0 版本起 TiDB 取消了这项限制。

> 注意：
>
> 通常，用户会开启 TiDB Binlog 将数据向下游进行同步。某些场景下，用户会使用消息中间件来消费同步到下游的 binlog，例如 Kafka。
>
> 以 Kafka 为例，Kafka 的单条消息处理能力的上限是 1 GB。因此，当把 txn-total-size-limit 设置为 1 GB 以上时，可能出现事务在 TiDB 中执行成功，但下游 Kafka 报错的情况。为避免这种情况出现，请用户根据最终消费者的限制来决定 txn-total-size-limit 的实际大小。例如：下游使用了 Kafka，则 txn-total-size-limit 不应超过 1 GB。

2.9.1.7 因果一致性事务

> 注意：
>
> 因果一致性事务只在启用 Async Commit 特性和一阶段提交特性时生效。关于这两个特性的启用情况，请参见tidb_enable_async_commit 系统变量介绍和tidb_enable_1pc 系统变量介绍。

TiDB 支持开启因果一致性的事务。因果一致性的事务在提交时无需向 PD 获取时间戳，所以提交延迟更低。开启因果一致性事务的语法为：

```
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY;
```

默认情况下，TiDB 保证线性一致性。在线性一致性的情况下，如果事务 2 在事务 1 提交完成后提交，逻辑上事务 2 就应该在事务 1 后发生。

因果一致性弱于线性一致性。在因果一致性的情况下，只有事务 1 和事务 2 加锁或写入的数据有交集时（即事务 1 和事务 2 存在数据库可知的因果关系时），才能保证事务的提交顺序与事务的发生顺序保持一致。目前暂不支持传入数据库外部的因果关系。

采用因果一致性的两个事务有以下特性：

- 有潜在因果关系的事务之间的逻辑顺序与物理提交顺序一致
- 无因果关系的事务之间的逻辑顺序与物理提交顺序不保证一致
- 不加锁的读取不产生因果关系

2.9.1.7.1 有潜在因果关系的事务之间的逻辑顺序与物理提交顺序一致

假设事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY	START TRANSACTION WITH CAUSAL CONSISTENCY ONLY
x = SELECT v FROM t WHERE id = 1 FOR UPDATE	
UPDATE t set v = $(x + 1) WHERE id = 2	
COMMIT	
	UPDATE t SET v = 2 WHERE id = 1
	COMMIT

上面的例子中，事务 1 对 `id = 1` 的记录加了锁，事务 2 的事务对 `id = 1` 的记录进行了修改，所以事务 1 和事务 2 有潜在的因果关系。所以即使用因果一致性开启事务，只要事务 2 在事务 1 提交成功后才提交，逻辑上事务 2 就必定比事务 1 晚发生。因此，不存在某个事务读到了事务 2 对 `id = 1` 记录的修改，但却没有读到事务 1 对 `id = 2` 记录的修改的情况。

2.9.1.7.2 无因果关系的事务之间的逻辑顺序与物理提交顺序不保证一致

假设 `id = 1` 和 `id = 2` 的记录最初值都为 0，事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2	事务 3
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY	START TRANSACTION WITH CAUSAL CONSISTENCY ONLY	
UPDATE t set v = 3 WHERE id = 2		
	UPDATE t SET v = 2 WHERE id = 1	
		BEGIN
COMMIT		
	COMMIT	
		SELECT v FROM t WHERE id IN (1, 2)

在本例中，事务 1 不读取 `id = 1` 的记录。此时事务 1 和事务 2 没有数据库可知的因果关系。如果使用因果一致性开启事务，即使物理时间上事务 2 在事务 1 提交完成后才开始提交，TiDB 也不保证逻辑上事务 2 比事务 1 晚发生。

此时如果有一个事务 3 在事务 1 提交前开启，并在事务 2 提交后读取 `id = 1` 和 `id = 2` 的记录，事务 3 可能读到 `id = 1` 的值为 2 但是 `id = 2` 的值为 0。

2.9.1.7.3 不加锁的读取不产生因果关系

假设事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY	START TRANSACTION WITH CAUSAL CONSISTENCY ONLY

事务 1	事务 2
	UPDATE t SET v = 2 WHERE id = 1
SELECT v FROM t WHERE id = 1	
UPDATE t set v = 3 WHERE id = 2	
	COMMIT
COMMIT	

如本例所示，不加锁的读取不产生因果关系。事务 1 和事务 2 产生了写偏斜的异常，如果他们有业务上的因果关系，则是不合理的。所以本例中，使用因果一致性的事务 1 和事务 2 没有确定的逻辑顺序。

2.9.2 TiDB 事务隔离级别

事务隔离级别是数据库事务处理的基础，ACID 中的 “I”，即 Isolation，指的就是事务的隔离性。

SQL-92 标准定义了 4 种隔离级别：读未提交 (READ UNCOMMITTED)、读已提交 (READ COMMITTED)、可重复读 (REPEATABLE READ)、串行化 (SERIALIZABLE)。详见下表：

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not possible	Possible	Possible
REPEATABLE READ	Not Possible	Not possible	Not possible	Possible
SERIALIZABLE	Not Possible	Not possible	Not possible	Not possible

TiDB 实现了快照隔离 (Snapshot Isolation, SI) 级别的一致性。为与 MySQL 保持一致，又称其为 “可重复读”。该隔离级别不同于ANSI 可重复读隔离级别和MySQL 可重复读隔离级别。

> 注意：
>
> 在 TiDB v3.0 中，事务的自动重试功能默认为禁用状态。不建议开启自动重试功能，因为可能导致事务隔离级别遭到破坏。更多关于事务自动重试的文档说明，请参考事务重试。
>
> 从 TiDB v3.0.8 版本开始，新创建的 TiDB 集群会默认使用悲观事务模式，悲观事务中的当前读（for update 读）为不可重复读，关于悲观事务使用注意事项，请参考悲观事务模式

2.9.2.1 可重复读隔离级别 (Repeatable Read)

当事务隔离级别为可重复读时，只能读到该事务启动时已经提交的其他事务修改的数据，未提交的数据或在事务启动后其他事务提交的数据是不可见的。对于本事务而言，事务语句可以看到之前的语句做出的修改。

对于运行于不同节点的事务而言，不同事务启动和提交的顺序取决于从 PD 获取时间戳的顺序。

处于可重复读隔离级别的事务不能并发的更新同一行，当事务提交时发现该行在该事务启动后，已经被另一个已提交的事务更新过，那么该事务会回滚。示例如下：

```
create table t1(id int);
insert into t1 values(0);

start transaction;              |               start transaction;
select * from t1;               |               select * from t1;
update t1 set id=id+1;          |               update t1 set id=id+1; -- 如果使用悲观事务，
    ↪ 则后一个执行的 update 语句会等锁，直到持有锁的事务提交或者回滚释放行锁。
commit;                         |
                                |               commit; -- 事务提交失败，回滚。如果使用悲观事务，
                                    ↪ 可以提交成功。
```

2.9.2.1.1 与 ANSI 可重复读隔离级别的区别

尽管名称是可重复读隔离级别，但是 TiDB 中可重复读隔离级别和 ANSI 可重复隔离级别是不同的。按照 A Critique of ANSI SQL Isolation Levels 论文中的标准，TiDB 实现的是论文中的快照隔离级别。该隔离级别不会出现狭义上的幻读 (A3)，但不会阻止广义上的幻读 (P3)，同时，SI 还会出现写偏斜，而 ANSI 可重复读隔离级别不会出现写偏斜，会出现幻读。

2.9.2.1.2 与 MySQL 可重复读隔离级别的区别

MySQL 可重复读隔离级别在更新时并不检验当前版本是否可见，也就是说，即使该行在事务启动后被更新过，同样可以继续更新。这种情况在 TiDB 使用乐观事务时会导致事务回滚，导致事务最终失败，而 TiDB 默认的悲观事务和 MySQL 是可以更新成功的。

2.9.2.2 读已提交隔离级别 (Read Committed)

从 TiDB v4.0.0-beta 版本开始，TiDB 支持使用 Read Committed 隔离级别。由于历史原因，当前主流数据库的 Read Committed 隔离级别本质上都是 Oracle 定义的一致性读隔离级别。TiDB 为了适应这一历史原因，悲观事务中的 Read Committed 隔离级别的实质行为也是一致性读。

> **注意：**
>
> Read Committed 隔离级别仅在悲观事务模式下生效。在乐观事务模式下设置事务隔离级别为 Read Committed 将不会生效，事务将仍旧使用可重复读隔离级别。

从 v6.0.0 版本开始，TiDB 支持使用系统变量tidb_rc_read_check_ts 对读写冲突较少情况下优化时间戳的获取。开启此变量后，SELECT 语句会尝试使用前一个有效的时间戳进行数据读取，初始值为事务的 start_ts。

- 如果整个读取过程没有遇到更新的数据版本，则返回结果给客户端且 SELECT 语句执行成功。
- 如果读取过程中遇到更新的数据版本：
  - 如果当前 TiDB 尚未向客户端回复数据，则尝试重新获取一个新的时间戳重试此语句。
  - 如果 TiDB 已经向客户端返回部分数据，则 TiDB 会向客户端报错。每次向客户端回复的数据量受 tidb_init_chunk_size 和 tidb_max_chunk_size 控制。

在使用 READ-COMMITTED 隔离级别且单个事务中 SELECT 语句较多、读写冲突较少的场景，可通过开启此变量来避免获取全局 timestamp 带来的延迟和开销。

2.9.2.2.1 与 MySQL Read Committed 隔离级别的区别

MySQL 的 Read Committed 隔离级别大部分符合一致性读特性，但其中存在某些特例，如半一致性读 (semi-consistent read)，TiDB 没有兼容这个特殊行为。

2.9.2.3 更多阅读

- TiDB 的乐观事务模型
- TiDB 新特性漫谈 -悲观事务
- TiDB 新特性 -白话悲观锁
- TiKV 的 MVCC (Multi-Version Concurrency Control) 机制

2.9.3 TiDB 乐观事务模型

乐观事务模型下，将修改冲突视为事务提交的一部分。因此并发事务不常修改同一行时，可以跳过获取行锁的过程进而提升性能。但是并发事务频繁修改同一行（冲突）时，乐观事务的性能可能低于悲观事务。

启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

> **注意：**
>
> 自 v3.0.8 开始，TiDB 集群默认使用悲观事务模式。但如果从 3.0.7 及之前版本创建的集群升级到 3.0.8 及之后的版本，不会改变默认事务模式，即只有新创建的集群才会默认使用悲观事务模式。

2.9.3.1 乐观事务原理

为支持分布式事务，TiDB 中乐观事务使用两阶段提交协议，流程如下：

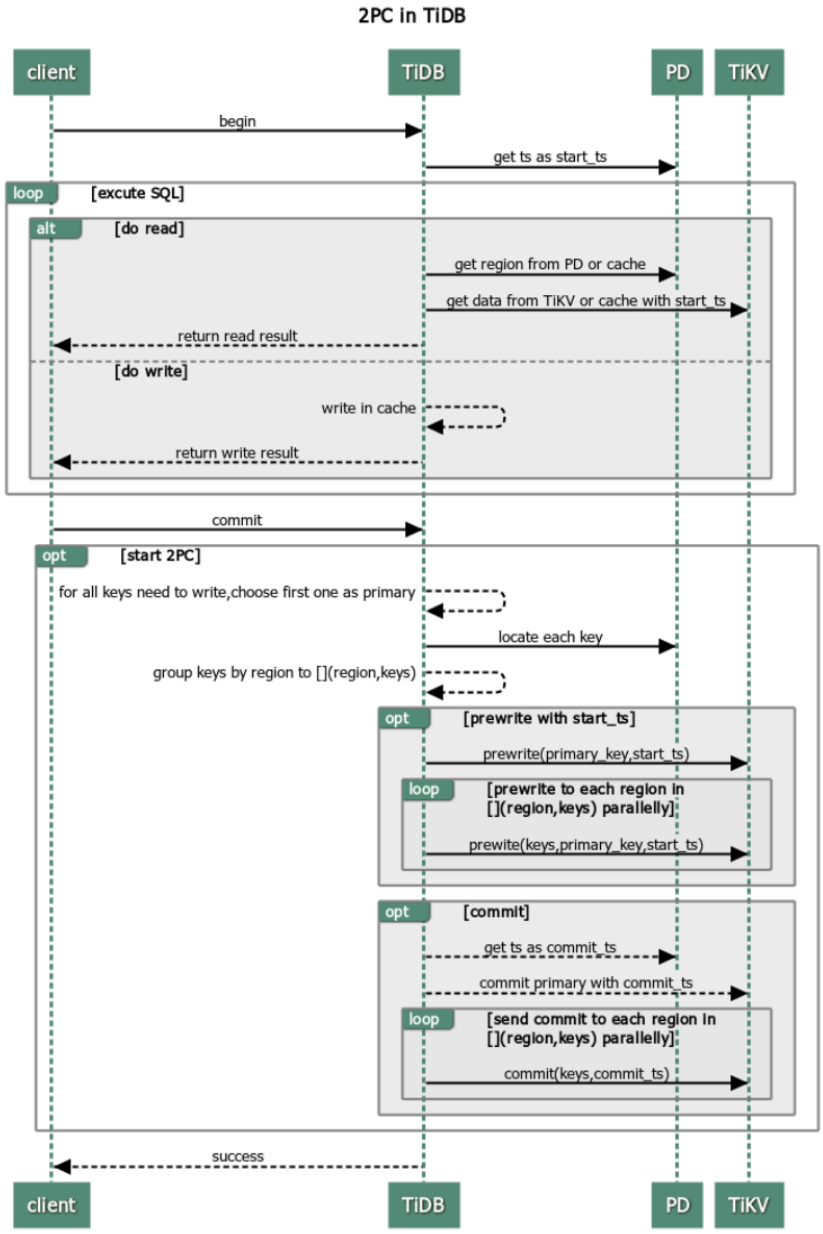


图 132: TiDB 中的两阶段提交

1. 客户端开始一个事务。

   TiDB 从 PD 获取一个全局唯一递增的时间戳作为当前事务的唯一事务 ID，这里称为该事务的 `start_ts`。TiDB 实现了多版本并发控制 (MVCC)，因此 `start_ts` 同时也作为该事务获取的数据库快照版本。该事务只能读到此 `start_ts` 版本可以读到的数据。

2. 客户端发起读请求。

   1. TiDB 从 PD 获取数据路由信息，即数据具体存在哪个 TiKV 节点上。
   2. TiDB 从 TiKV 获取 `start_ts` 版本下对应的数据。

3. 客户端发起写请求。

   TiDB 校验写入数据是否符合约束（如数据类型是否正确、是否符合非空约束等）。校验通过的数据将存放在 TiDB 中该事务的私有内存里。

4. 客户端发起 commit。

5. TiDB 开始两阶段提交，在保证事务原子性的前提下，进行数据持久化。

   1. TiDB 从当前要写入的数据中选择一个 Key 作为当前事务的 Primary Key。
   2. TiDB 从 PD 获取所有数据的写入路由信息，并将所有的 Key 按照所有的路由进行分类。
   3. TiDB 并发地向所有涉及的 TiKV 发起 prewrite 请求。TiKV 收到 prewrite 数据后，检查数据版本信息是否存在冲突或已过期。符合条件的数据会被加锁。
   4. TiDB 收到所有 prewrite 响应且所有 prewrite 都成功。
   5. TiDB 向 PD 获取第二个全局唯一递增版本号，定义为本次事务的 `commit_ts`。
   6. TiDB 向 Primary Key 所在 TiKV 发起第二阶段提交。TiKV 收到 commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁。
   7. TiDB 收到两阶段提交成功的信息。

6. TiDB 向客户端返回事务提交成功的信息。

7. TiDB 异步清理本次事务遗留的锁信息。

2.9.3.2 优缺点分析

通过分析 TiDB 中事务的处理流程，可以发现 TiDB 事务有如下优点：

- 实现原理简单，易于理解。
- 基于单实例事务实现了跨节点事务。
- 锁管理实现了去中心化。

但 TiDB 事务也存在以下缺点：

- 两阶段提交使网络交互增多。
- 需要一个中心化的分配时间戳服务。
- 事务数据量过大时易导致内存暴涨。

2.9.3.3 事务的重试

使用乐观事务模型时，在高冲突率的场景中，事务容易发生写写冲突而导致提交失败。MySQL 使用悲观事务模型，在执行写入类型的 SQL 语句的过程中进行加锁并且在 Repeatable Read 隔离级别下使用了当前读的机制，能够读取到最新的数据，所以提交时一般不会出现异常。为了降低应用改造难度，TiDB 提供了数据库内部自动重试机制。

2.9.3.3.1 重试机制

当事务提交时，如果发现写写冲突，TiDB 内部重新执行包含写操作的 SQL 语句。你可以通过设置 `tidb_disable_txn_auto_retry = OFF` 开启自动重试，并通过 `tidb_retry_limit` 设置重试次数：

```
### 设置是否禁用自动重试，默认为 “on”，即不重试。
tidb_disable_txn_auto_retry = OFF
### 控制重试次数，默认为 “10”。只有自动重试启用时该参数才会生效。
### 当 “tidb_retry_limit= 0” 时，也会禁用自动重试。
tidb_retry_limit = 10
```

你也可以修改当前 Session 或 Global 的值：

- Session 级别设置：

```
SET tidb_disable_txn_auto_retry = OFF;
```

```
SET tidb_retry_limit = 10;
```

- Global 级别设置：

```
SET GLOBAL tidb_disable_txn_auto_retry = OFF;
```

```
SET GLOBAL tidb_retry_limit = 10;
```

> 注意：
>
> `tidb_retry_limit` 变量决定了事务重试的最大次数。当它被设置为 0 时，所有事务都不会自动重试，包括自动提交的单语句隐式事务。这是彻底禁用 TiDB 中自动重试机制的方法。禁用自动重试后，所有冲突的事务都会以最快的方式上报失败信息（包含 `try again later`）给应用层。

2.9.3.3.2 重试的局限性

TiDB 默认不进行事务重试，因为重试事务可能会导致更新丢失，从而破坏可重复读的隔离级别。

事务重试的局限性与其原理有关。事务重试可概括为以下三个步骤：

1. 重新获取 `start_ts`。
2. 重新执行包含写操作的 SQL 语句。
3. 再次进行两阶段提交。

第二步中，重试时仅重新执行包含写操作的 SQL 语句，并不涉及读操作的 SQL 语句。但是当前事务中读到数据的时间与事务真正开始的时间发生了变化，写入的版本变成了重试时获取的 `start_ts` 而非事务一开始时

获取的 start_ts。因此，当事务中存在依赖查询结果来更新的语句时，重试将无法保证事务原本可重复读的隔离级别，最终可能导致结果与预期出现不一致。

如果业务可以容忍事务重试导致的异常，或并不关注事务是否以可重复读的隔离级别来执行，则可以开启自动重试。

2.9.3.4 冲突检测

作为一个分布式系统，TiDB 在内存中的冲突检测是在 TiKV 中进行，主要发生在 prewrite 阶段。因为 TiDB 集群是一个分布式系统，TiDB 实例本身无状态，实例之间无法感知到彼此的存在，也就无法确认自己的写入与别的 TiDB 实例是否存在冲突，所以会在 TiKV 这一层检测具体的数据是否有冲突。

具体配置如下：

```
### scheduler 内置一个内存锁机制，防止同时对一个 Key 进行操作。
### 每个 Key hash 到不同的 slot。（默认为 2048000）
scheduler-concurrency = 2048000
```

此外，TiKV 支持监控等待 latch 的时间：

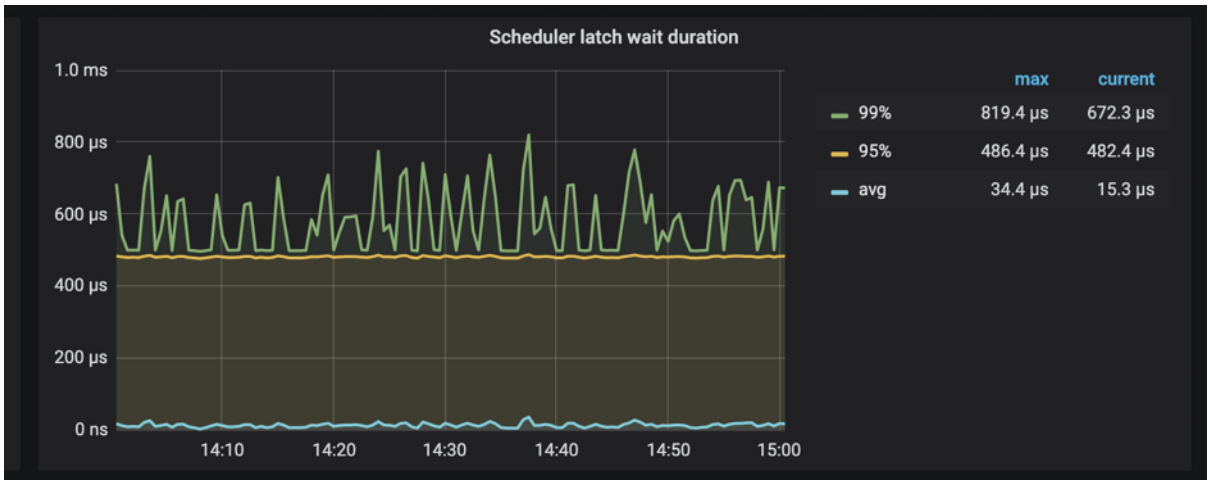


图 133: Scheduler latch wait duration

当 Scheduler latch wait duration 的值特别高时，说明大量时间消耗在等待锁的请求上。如果不存在底层写入慢的问题，基本上可以判断该段时间内冲突比较多。

2.9.3.5 更多阅读

- Percolator 和 TiDB 事务算法

2.9.4 TiDB 悲观事务模式

为了使 TiDB 的使用方式更加贴近传统数据库，降低用户迁移的成本，TiDB 自 v3.0 版本开始在乐观事务模型的基础上支持了悲观事务模式。本文将介绍 TiDB 悲观事务的相关特性。

> **注意：**
>
> 自 v3.0.8 开始，新创建的 TiDB 集群默认使用悲观事务模式。但如果从 v3.0.7 版本及之前创建的集群升级到 >= v3.0.8 的版本，则不会改变默认的事务模式，即只有新创建的集群才会默认使用悲观事务模式。

2.9.4.1 事务模式的修改方法

你可以使用tidb_txn_mode 系统变量设置事务模式。执行以下命令，即可使整个集群中所有新创建 session 执行的所有显示事务（即非 autocommit 的事务）进入悲观事务模式：

```
SET GLOBAL tidb_txn_mode = 'pessimistic';
```

除此之外，还可以执行以下 SQL 语句显式地开启悲观事务：

```
BEGIN PESSIMISTIC;
```

```
BEGIN /*T! PESSIMISTIC */;
```

`BEGIN PESSIMISTIC;` 和 `BEGIN OPTIMISTIC;` 等语句的优先级高于 `tidb_txn_mode` 系统变量。使用这两个语句开启的事务，会忽略系统变量，从而支持悲观、乐观事务混合使用。

2.9.4.2 悲观事务模式的行为

悲观事务的行为和 MySQL 基本一致（不一致之处详见和 MySQL InnoDB 的差异）：

- 悲观事务中引入快照读和当前读的概念：
  - 快照读是一种不加锁读，读的是该事务开始时刻前已提交的版本。`SELECT` 语句中的读是快照读。
  - 当前读是一种加锁读，读取的是最新已提交的版本，`UPDATE`、`DELETE`、`INSERT`、`SELECT FOR UPDATE` 语句中的读是当前读。

以下示例是对快照读和当前读的详细说明：

session 1	session 2
CREATE TABLE t (a INT);	
INSERT INTO T VALUES(1);	
BEGIN PESSIMISTIC;	
UPDATE t SET a = a + 1;	
	BEGIN PESSIMISTIC;
	SELECT * FROM t; – 使用
COMMIT; – 释放锁，session 3 的 SELECT FOR UPDATE 操作获得锁，使用当前读，读到最新已提交的版本 (a=2)	
	SELECT * FROM t; – 使用

- 悲观锁会在事务提交或回滚时释放。其他尝试修改这一行的写事务会被阻塞，等待悲观锁的释放。其他尝试读取这一行的事务不会被阻塞，因为 TiDB 采用多版本并发控制机制 (MVCC)。
- 如果多个事务尝试获取各自的锁，会出现死锁，并被检测器自动检测到。其中一个事务会被随机终止掉并返回兼容 MySQL 的错误码 1213。
- 通过 `innodb_lock_wait_timeout` 变量，设置事务等锁的超时时间（默认值为 50，单位为秒）。等锁超时后返回兼容 MySQL 的错误码 1205。如果多个事务同时等待同一个锁释放，会大致按照事务 `start ts` 顺序获取锁。
- 乐观事务和悲观事务可以共存，事务可以任意指定使用乐观模式或悲观模式来执行。
- 支持 `FOR UPDATE NOWAIT` 语法，遇到锁时不会阻塞等锁，而是返回兼容 MySQL 的错误码 3572。
- 如果 `Point Get` 和 `Batch Point Get` 算子没有读到数据，依然会对给定的主键或者唯一键加锁，阻塞其他事务对相同主键唯一键加锁或者进行写入操作。
- 支持 `FOR UPDATE OF TABLES` 语法，对于存在多表 join 的语句，只对 `OF TABLES` 中包含的表关联的行进行悲观锁加锁操作。

2.9.4.3 和 MySQL InnoDB 的差异

1. 有些 `WHERE` 子句中使用了 range，TiDB 在执行这类 DML 语句和 `SELECT FOR UPDATE` 语句时，不会阻塞 range 内并发的 DML 语句的执行。

   举例：

   ```
   CREATE TABLE t1 (
    id INT NOT NULL PRIMARY KEY,
    pad1 VARCHAR(100)
   );
   INSERT INTO t1 (id) VALUES (1),(5),(10);
   ```

   ```
   BEGIN /*T! PESSIMISTIC */;
   SELECT * FROM t1 WHERE id BETWEEN 1 AND 10 FOR UPDATE;
   ```

   ```
   BEGIN /*T! PESSIMISTIC */;
   INSERT INTO t1 (id) VALUES (6); -- 仅 MySQL 中出现阻塞。
   UPDATE t1 SET pad1='new value' WHERE id = 5; -- MySQL 和 TiDB 处于等待阻塞状态。
   ```

   产生这一行为是因为 TiDB 当前不支持 gap locking（间隙锁）。

2. TiDB 不支持 `SELECT LOCK IN SHARE MODE`。

   使用这个语句执行的时候，效果和没有加锁是一样的，不会阻塞其他事务的读写。

3. DDL 可能会导致悲观事务提交失败。

   MySQL 在执行 DDL 语句时，会被正在执行的事务阻塞住，而在 TiDB 中 DDL 操作会成功，造成悲观事务提交失败：`ERROR 1105 (HY000): Information schema is changed. [try again later]`。TiDB 事务执行过程中并发执行 `TRUNCATE TABLE` 语句，可能会导致事务报错 `table doesn't exist`。

4. START TRANSACTION WITH CONSISTENT SNAPSHOT 之后，MySQL 仍然可以读取到之后在其他事务创建的表，而 TiDB 不能。
5. autocommit 事务优先采用乐观事务提交。

   使用悲观事务模式时，autocommit 事务首先尝试使用开销更小的乐观事务模式提交。如果发生了写冲突，重试时才会使用悲观事务提交。所以 tidb_retry_limit = 0 时，autocommit 事务遇到写冲突仍会报 Write Conflict 错误。

   自动提交的 SELECT FOR UPDATE 语句不会等锁。
6. 对语句中 EMBEDDED SELECT 读到的相关数据不会加锁。
7. 垃圾回收 (GC) 不会影响到正在执行的事务，但悲观事务的执行时间仍有上限，默认为 1 小时，可通过 TiDB 配置文件 [performance] 类别下的 max-txn-ttl 修改。

2.9.4.4 隔离级别

TiDB 在悲观事务模式下支持了 2 种隔离级别：

1. 默认使用与 MySQL 行为相同的可重复读隔离级别 (Repeatable Read)。

> 注意：
> 在这种隔离级别下，DML 操作会基于已提交的最新数据来执行，行为与 MySQL 相同，但与 TiDB 乐观事务不同，请参考与 MySQL 可重复读隔离级别的区别。

2. 使用SET TRANSACTION 语句可将隔离级别设置为读已提交隔离级别 (Read Committed)。

2.9.4.5 悲观事务提交流程

TiDB 悲观锁复用了乐观锁的两阶段提交逻辑，重点在 DML 执行时做了改造。

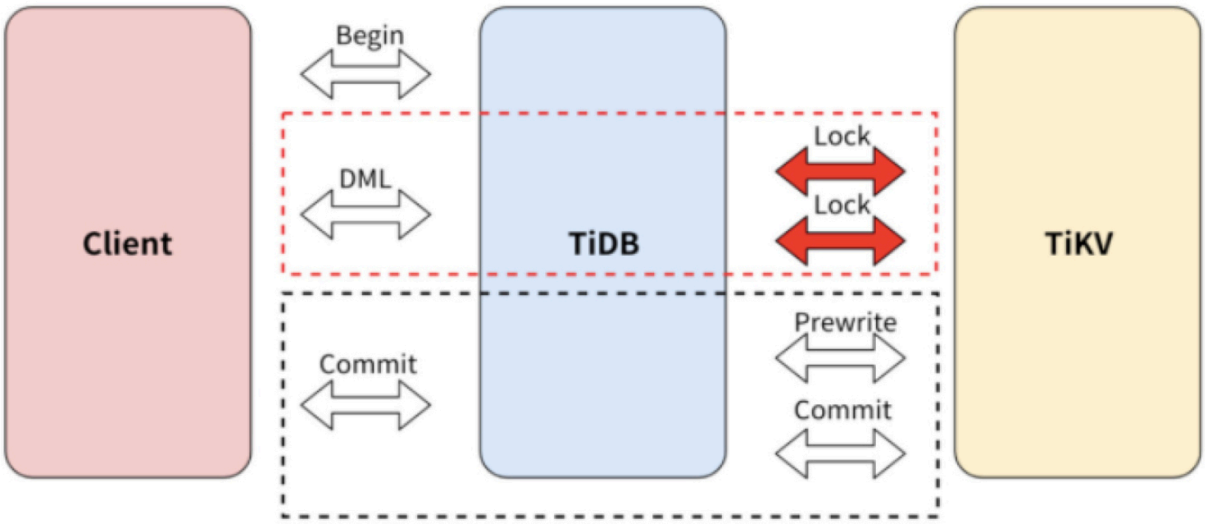


图 134: TiDB 悲观事务的提交流程

在两阶段提交之前增加了 Acquire Pessimistic Lock 阶段，简要步骤如下。

1. （同乐观锁）TiDB 收到来自客户端的 begin 请求，获取当前版本号作为本事务的 StartTS。
2. TiDB 收到来自客户端的更新数据的请求：TiDB 向 TiKV 发起加悲观锁请求，该锁持久化到 TiKV。
3. （同乐观锁）客户端发起 commit，TiDB 开始执行与乐观锁一样的两阶段提交。

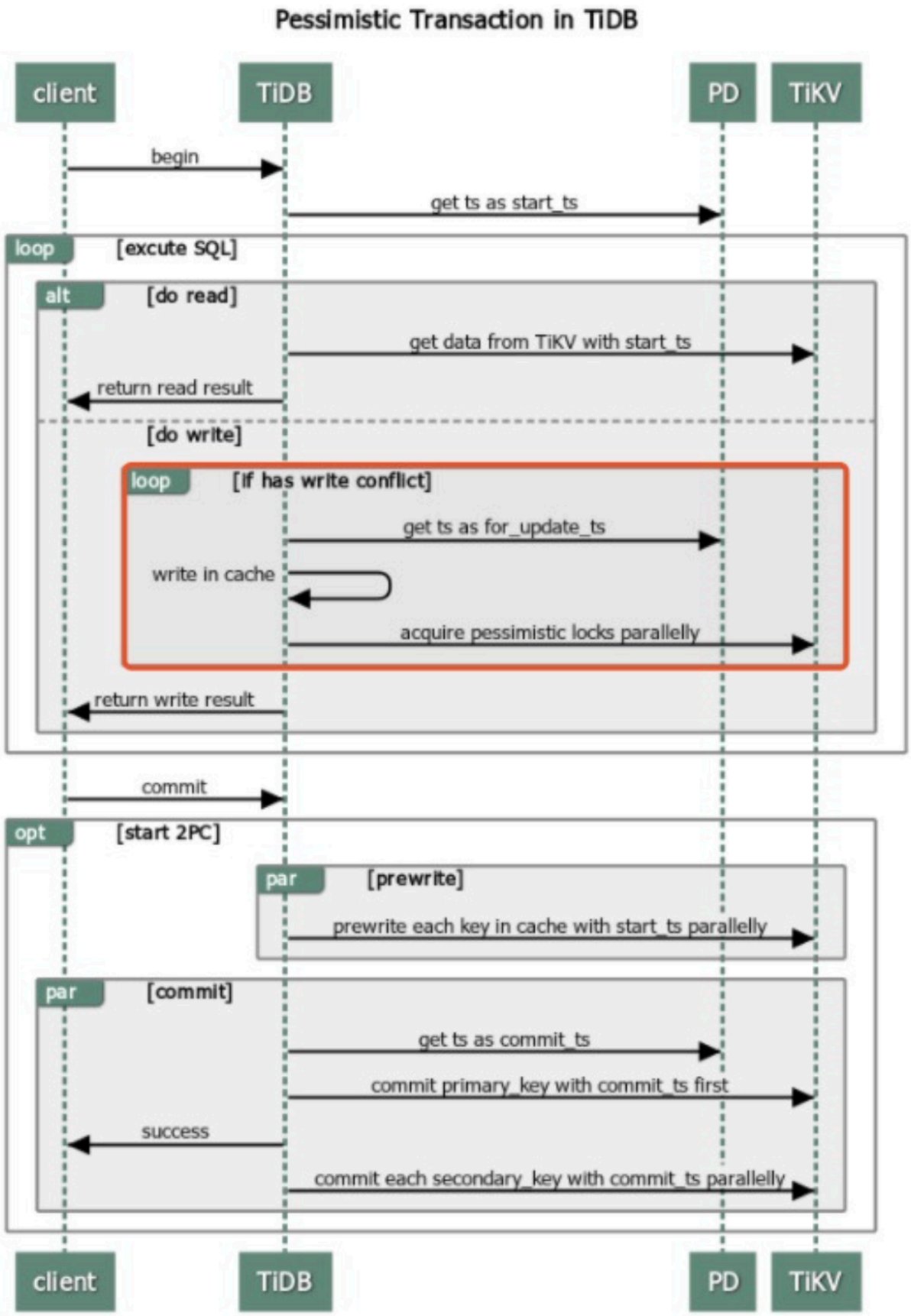


图 135: TiDB 中的悲观事务

相关细节本节不再赘述，详情可阅读 TiDB 悲观锁实现原理。

2.9.4.6 Pipelined 加锁流程

加悲观锁需要向 TiKV 写入数据，要经过 Raft 提交并 apply 后才能返回，相比于乐观事务，不可避免的会增加部分延迟。为了降低加锁的开销，TiKV 实现了 pipelined 加锁流程：当数据满足加锁要求时，TiKV 立刻通知 TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但当 TiKV 出现网络隔离或者节点宕机时，悲观锁异步写入有可能失败，从而产生以下影响：

- 无法阻塞修改相同数据的其他事务。如果业务逻辑依赖加锁或等锁机制，业务逻辑的正确性将受到影响。
- 有较低概率导致事务提交失败，但不会影响事务正确性。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭 pipelined 加锁功能。

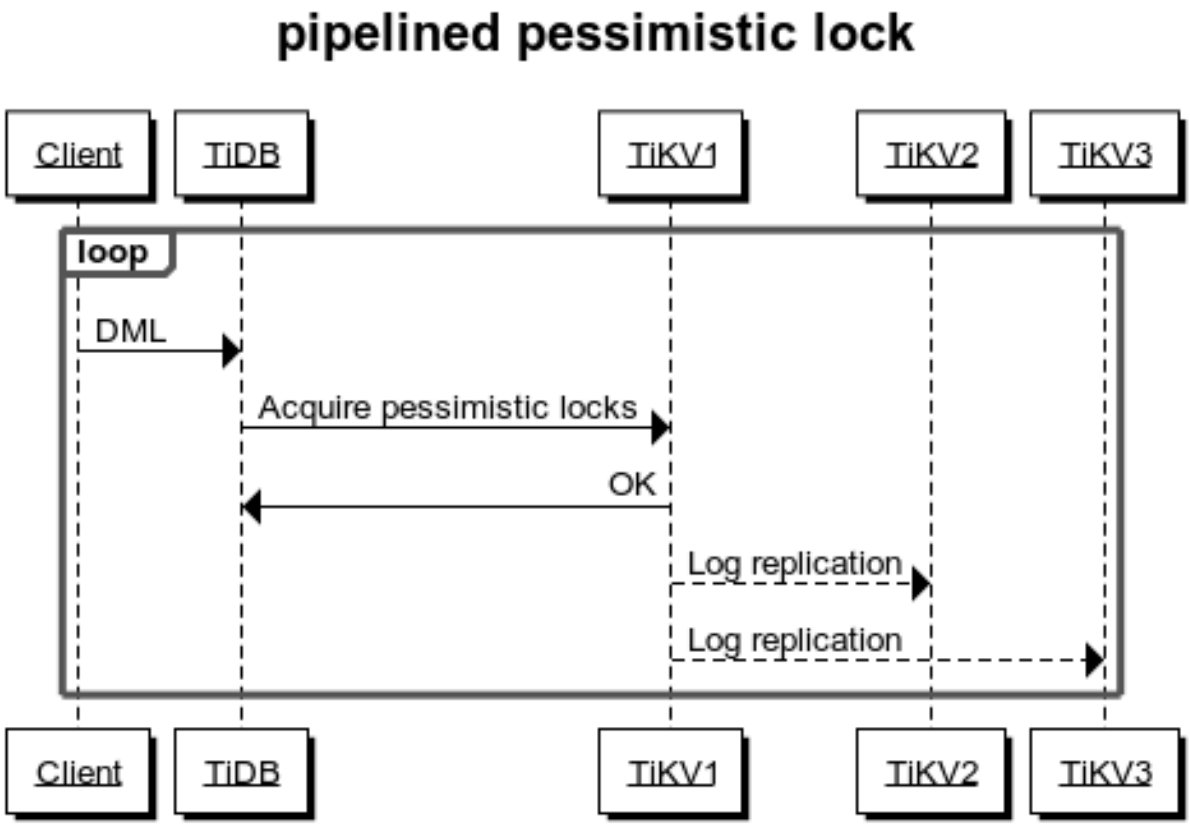


图 136: Pipelined pessimistic lock

该功能默认开启，可修改 TiKV 配置关闭：

```
[pessimistic-txn]
pipelined = false
```

若集群是 v4.0.9 及以上版本，也可通过在线修改 TiKV 配置功能动态关闭该功能：

```
set config tikv pessimistic-txn.pipelined='false';
```

2.9.4.7 内存悲观锁

TiKV 在 v6.0.0 中引入了内存悲观锁功能。开启内存悲观锁功能后，悲观锁通常只会被存储在 Region leader 的内存中，而不会将锁持久化到磁盘，也不会通过 Raft 协议将锁同步到其他副本，因此可以大大降低悲观事务加锁的开销，提升悲观事务的吞吐并降低延迟。

当内存悲观锁占用的内存达到 Region 或节点的阈值时，加悲观锁会回退为使用pipelined 加锁流程。当 Region 发生合并或 leader 迁移时，为避免悲观锁丢失，TiKV 会将内存悲观锁写入磁盘并同步到其他副本。

内存悲观锁实现了和pipelined 加锁类似的表现，即集群无异常时不影响加锁表现，但当 TiKV 出现网络隔离或者节点宕机时，事务加的悲观锁可能丢失。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭内存悲观锁功能。

该功能默认开启。如要关闭，可修改 TiKV 配置：

```
[pessimistic-txn]
in-memory = false
```

也可通过在线修改 TiKV 配置功能动态关闭该功能：

```
set config tikv pessimistic-txn.in-memory='false';
```

2.9.5 非事务 DML 语句

本文档介绍非事务 DML 语句的使用场景、使用方法、使用限制和使用该功能的常见问题。

非事务 DML 语句是将一个普通 DML 语句拆成多个 SQL 语句（即多个 batch）执行，以牺牲事务的原子性和隔离性为代价，增强批量数据处理场景下的性能和易用性。

非事务 DML 语句包括 INSERT、UPDATE 和 DELETE，TiDB 目前只支持非事务 DELETE 语句，详细的语法介绍见BATCH。

> 注意：
>
> 非事务 DML 语句不保证该语句的原子性和隔离性，不能认为它和原始 DML 语句等价。

2.9.5.1 使用场景

在大批量的数据处理场景，用户经常需要对一大批数据执行相同的操作。如果直接使用一个 SQL 语句执行操作，很可能导致事务大小超过限制，而大事务会明显影响执行性能。

批量数据处理操作往往和在线业务操作不具有时间或数据上的交集。没有并发操作时，隔离性是不必要的。如果将批量数据操作设计成幂等的，或者易于重试的，那么原子性也是不必要的。如果你的业务满足这两个条件，那么可以考虑使用非事务 DML 语句。

非事务 DML 语句用于在特定场景下绕过大事务的事务大小限制，用一条语句完成原本需要拆分成多个事务完成的任务，且执行效率更高，占用资源更少。

例如，对于一个删除过期数据的需求，在确保没有任何业务会访问过期数据时，适合使用非事务 DML 语句来提升删除性能。

2.9.5.2 前提条件

要使用非事务 DML 语句，必须满足以下条件：

- 确保该语句不需要原子性，即允许执行结果中，一部分行被修改，而一部分行没有被修改。
- 确保该语句具有幂等性，或是做好准备根据错误信息对部分数据重试。如果系统变量 `tidb_redact_log` `= 1` 且 `tidb_nontransactional_ignore_error = 1`，则该语句必须是幂等的。否则语句部分失败时，无法准确定位失败的部分。
- 确保该语句将要操作的数据没有其它并发的写入，即不被其它语句同时更新。否则可能出现漏删、多删等非预期的现象。
- 确保该语句不会修改语句自身会读取的内容，否则后续的 batch 读到之前 batch 写入的内容，容易引起非预期的情况。
- 确认该语句满足使用限制。
- 不建议在该 DML 语句将要读写的表上同时进行并发的 DDL 操作。

> **警告：**
>
> 如果同时开启了 `tidb_redact_log` 和 `tidb_nontransactional_ignore_error`，你可能无法完整得知每个 batch 的错误信息，无法只对失败的 batch 进行重试。因此，如果同时开启了这两个系统变量，该非事务 DML 语句必须是幂等的。

2.9.5.3 使用示例

2.9.5.3.1 使用非事务 DML 语句

以下部分通过示例说明非事务 DML 语句的使用方法：

创建一张表 `t`，表结构如下：

```
CREATE TABLE t(id int, v int, key(id));
```

```
Query OK, 0 rows affected
```

向表 t 中插入一些数据。

```
INSERT INTO t VALUES (1,2),(2,3),(3,4),(4,5),(5,6);
```

```
Query OK, 5 rows affected
```

以下操作使用非事务 DML 语句，删除表 t 的 v 列上小于整数 6 的行。该语句将以 2 为 batch size，以 id 列为划分列，拆分为两个 SQL 语句执行。

```
BATCH ON id LIMIT 2 DELETE FROM t where v < 6;
```

```
+----------------+---------------+
| number of jobs | job status    |
+----------------+---------------+
| 2              | all succeeded |
+----------------+---------------+
1 row in set
```

查看非事务 DML 语句的删除结果。

```
SELECT * FROM t;
```

```
+----+---+
| id | v |
+----+---+
| 5  | 6 |
+----+---+
1 row in set
```

2.9.5.3.2 查看非事务 DML 语句的执行进度

非事务 DML 语句执行过程中，可以通过 SHOW PROCESSLIST 查看执行进度，返回结果中的 Time 表示当前 batch 执行的耗时。日志、慢日志等也会记录每个拆分后的语句在整个非事务 DML 语句中的进度。例如：

```
show processlist;
```

```
+--
    ↪ ----+------+--------------------+--------+---------+------+------------+-------------------------------
    ↪
| Id   | User | Host               | db     | Command | Time | State      | Info
    ↪
    ↪ |
```

```
+--
    ↪ ----+------+--------------------+--------+---------+------+------------+-----------------------------------
    ↪
| 1203 | root | 100.64.10.62:52711 | test   | Query   | 0    | autocommit | /* job 506/500000 */
    ↪ DELETE FROM `test`.`t1` WHERE `test`.`t1`.`_tidb_rowid` BETWEEN 2271 AND 2273 |
| 1209 | root | 100.64.10.62:52735 | <null> | Query   | 0    | autocommit | show full processlist
    ↪                                                                                |
+--
    ↪ ----+------+--------------------+--------+---------+------+------------+-----------------------------------
    ↪
```

2.9.5.3.3 终止一个非事务 DML 语句

通过 `KILL TIDB` 终止一个非事务语句时，TiDB 会取消当前正在执行的 batch 之后的所有 batch。执行结果信息需要从日志里获得。

2.9.5.3.4 查询非事务 DML 语句中划分 batch 的语句

要查询非事务 DML 语句中用于划分 batch 的语句，你可在非事务 DML 语句中添加 `DRY RUN QUERY`。添加后，TiDB 不实际执行这个查询和后续的 DML 操作。

下面这条语句查询 `BATCH ON id LIMIT 2 DELETE FROM t WHERE v < 6` 这条非事务 DML 语句内用于划分 batch 的查询语句。

```
BATCH ON id LIMIT 2 DRY RUN QUERY DELETE FROM t WHERE v < 6;
```

```
+-------------------------------------------------------------------------------+
| query statement                                                               |
+-------------------------------------------------------------------------------+
| SELECT `id` FROM `test`.`t` WHERE (`v` < 6) ORDER BY IF(ISNULL(`id`),0,1),`id` |
+-------------------------------------------------------------------------------+
1 row in set
```

2.9.5.3.5 查询非事务 DML 语句中首末 batch 对应的语句

要查询非事务 DML 语句中第一个和最后一个 batch 对应的实际 DML 语句，你可在语句中添加 `DRY RUN`。添加后，TiDB 只划分 batch，不执行这些 SQL 语句。因为 batch 数量可能很多，不显示全部 batch，只显示第一个和最后一个 batch。

```
BATCH ON id LIMIT 2 DRY RUN DELETE FROM t where v < 6;
```

```
+-----------------------------------------------------------------+
| split statement examples                                        |
+-----------------------------------------------------------------+
| DELETE FROM `test`.`t` WHERE (`id` BETWEEN 1 AND 2 AND (`v` < 6)) |
| DELETE FROM `test`.`t` WHERE (`id` BETWEEN 3 AND 4 AND (`v` < 6)) |
```

```
+-------------------------------------------------------------------+
2 rows in set
```

2.9.5.3.6 在非事务 DML 语句中使用 Optimizer Hint

对于 DELETE 语句原本支持的 Optimizer Hint，非事务 DELETE 语句也同样支持，hint 位置与普通 DELETE 语句中的位置相同：

```
BATCH ON id LIMIT 2 DELETE /*+ USE_INDEX(t)*/ FROM t where v < 6;
```

2.9.5.4 最佳实践

建议按照以下步骤执行非事务 DML 语句：

1. 选择合适的划分列。建议使用整数或字符串类型。
2. （可选）在非事务 DML 语句中添加 DRY RUN QUERY，手动执行查询，确认 DML 语句影响的数据范围是否大体正确。
3. （可选）在非事务 DML 语句中添加 DRY RUN，手动执行查询，检查拆分后的语句和执行计划。需要关注索引选择效率。
4. 执行非事务 DML 语句。
5. 如果报错，从报错信息或日志中获取具体失败的数据范围，进行重试或手动处理。

2.9.5.5 参数说明

参数	说明	默认值	是否必填	建议值

参数	说明	默认值	是否必填	建议值

划分列	用于划分 batch 的列，例如以上非事务 DML 语句	TiDB 尝试自动选择	否	选择可以最高效地满足 WHERE 条件的列

```
BATCH
ON
id
LIMIT
2
DELETE
FROM
t
WHERE
v
<
6
```

参数	说明	默认值	是否必填	建议值
Batch size	用于控制每个 batch 的大小，batch 即 DML 操作拆分成的 SQL 语句个数，例如以上非事务 DML 语句 BATCH ↪ ↪ ON ↪ ↪ id ↪ ↪ LIMIT ↪ ↪ 2 493 ↪ ↪ DELETE ↪	N/A	是	1000 ~ 1000000，过小和过大都会导致性能下降

参数	说明	默认值	是否必填	建议值

2.9.5.5.1 划分列的选择

非事务 DML 语句需要用一个列作为数据分批的标准，该列即为划分列。为获得更高的执行效率，划分列必须能够利用索引。不同的索引和划分列所导致的执行效率可能有数十倍的差别。选择划分列时，可以考虑以下建议：

- 当你对业务数据分布有一定了解时，根据 `WHERE` 条件，选择划分 batch 后，划分范围较小的列。
  - 在理想情况下，`WHERE` 条件可以利用划分列的索引，降低每个 batch 需要扫描的数据量。例如有一个交易表，记录了每一笔交易的开始和结束时间，你希望删除结束时间在一个月之前的所有交易记录。如果在交易的开始时间有索引，且已知交易的开始和结束时间通常相差不大，那么可以选择开始时间作为划分列。
  - 在不太理想的情况下：划分列的数据分布与 `WHERE` 条件完全无关，无法利用划分列的索引来减少数据扫描的范围。
- 有聚簇索引时，建议用主键作为划分列，这样语句执行效率更高（包括 `INT` 主键和 `_tidb_rowid`）。
- 选择重复值较少的列。

你可以不指定划分列，TiDB 默认会使用 handle 的第一列作为划分列。但如果聚簇索引主键的第一列是非事务 DML 语句不支持的数据类型（即 `ENUM`，`BIT`，`SET`，`JSON`），TiDB 会报错。你可根据业务需要选择合适的划分列。

2.9.5.5.2 Batch size 的选择

非事务 DML 语句中，batch size 越大，拆分出来的 SQL 语句越少，每个 SQL 语句执行起来越慢。最优的 batch size 依据 workload 而定。根据经验值，推荐从 50000 开始尝试。过小和过大的 batch size 都会导致执行效率下降。

每个 batch 的信息存储在内存里，因此 batch 数量过多会显著增加内存消耗。这也是 batch size 不能过小的原因之一。非事务语句用于存储 batch 信息消耗的内存上限与`tidb_mem_quota_query` 相同，超出这个限制时触发的操作由系统变量`tidb_mem_oom_action` 控制。

2.9.5.6 使用限制

非事务 DML 语句的硬性限制，不满足这些条件时 TiDB 会报错。

- 只可对单表进行操作，暂不支持多表连接。
- DML 语句不能包含 `ORDER BY` 或 `LIMIT` 字句。
- 用于拆分的列必须被索引。该索引可以是单列的索引，或是一个联合索引的第一列。
- 必须在`autocommit` 模式中使用。
- 不能在开启了 batch-dml 时使用。
- 不能在设置了`tidb_snapshot` 时使用。
- 不能与 `prepare` 语句一起使用。

- 划分列不支持 ENUM，BIT，SET，JSON 类型。
- 不支持用在临时表上。
- 不支持 [公共表表达式](/develop/dev-guide-use-common-table-expression.md)。

2.9.5.7 控制 batch 执行失败

非事务 DML 语句不满足原子性，可能存在一些 batch 成功，一些 batch 失败的情况。系统变量tidb_nontransactional_ignore_error
↪ 控制非事务 DML 语句处理错误的行为。

一个例外是，如果第一个 batch 就执行失败，有很大概率是语句本身有错，此时整个非事务语句会直接返回一个错误。

2.9.5.8 实现原理

非事务 DML 语句的实现原理，是将原本需要在用户侧手动执行的 SQL 语句拆分工作内置为 TiDB 的一个功能，简化用户操作。要理解非事务 DML 语句的行为，可以将其想象成一个用户脚本进行了如下操作：

对于非事务 DML `BATCH ON $C$ LIMIT $N$ DELETE FROM ... WHERE $P$`，其中 C 为用于拆分的列，N 为 batch size，P 为筛选条件。

1. TiDB 根据原始语句的筛选条件 P，和指定的用于拆分的列 C，查询出所有满足 P 的 C。对这些 C 排序后按 N 分成多个分组 $B_1 \dots B_k$。对所有 B_i，保留它的第一个和最后一个 C，记为 S_i 和 E_i。这一步所执行的查询语句，可以通过DRY RUN QUERY 查看。
2. B_i 所涉及的数据就是满足 P_i: C BETWEEN S_i AND E_i 的一个子集。可以通过 P_i 来缩小每个 batch 需要处理的数据范围。
3. 对 B_i，将上面这个条件嵌入原始语句的 WHERE 条件，使其变为 WHERE (P_i) AND (P)。这一步的执行结果，可以通过DRY RUN 查看。
4. 对所有 batch，依次执行新的语句。收集每个分组的错误并合并，在所有分组结束后作为整个非事务 DML 语句的结果返回。

2.9.5.9 与 batch-dml 的异同

batch-dml 是一种在 DML 语句执行期间将一个事务拆成多个事务提交的机制。

> 注意：
>
> batch-dml 功能使用不当时，存在数据索引不一致风险。该功能将在 TiDB 后续版本中被废弃，因此不建议使用。

非事务 DML 语句尚不能替代所有的 batch-dml 使用场景。它们的主要区别有：

- 性能：在划分效率较高的情况下，非事务 DML 语句和 batch-dml 性能接近。在划分效率较低的情况下，非事务 DML 语句可能会明显慢于 batch-dml。
- 稳定性：batch-dml 极易因为使用不当导致数据索引不一致问题。而非事务 DML 语句不会导致数据索引不一致问题。但使用不当时，非事务 DML 语句与原始语句不等价，应用可能观察到和预期不符的现象。详见常见问题。

2.9.5.10 常见问题

2.9.5.10.1 实际的 batch 大小和指定的 batch size 不一样

在非事务 DML 语句的执行过程中，最后一个 batch 处理的数据量可能会小于 batch size。

在划分列有重复值时，每个 batch 会将当前 batch 中划分列的最后一个元素的所有重复值全部加入当前 batch 中，因此这个 batch 的行数可能会多于 batch size。

另外，在有其它并发的数据写入时，也可能导致每个 batch 实际处理的行数和 batch size 不一致。

2.9.5.10.2 执行时出现报错 `Failed to restore the delete statement, probably because of unsupported type of the shard column`

划分列的类型暂时不支持 ENUM、BIT、SET、JSON 类型，请尝试重新指定一个划分列。推荐使用整数或字符串类型的列。如果划分列不是这些类型，请联系 PingCAP 技术支持。

2.9.5.10.3 非事务 DELETE 出现和普通的 DELETE 不等价的“异常”行为

非事务 DML 语句和这个 DML 语句的原始形式并不等价，这可能是由以下原因导致的：

- 有并发的其它写入。
- 非事务 DML 语句修改了语句自身会读取的值。
- 在每个 batch 上实际执行的 SQL 语句由于改变了 WHERE 条件，可能会导致执行计划以及表达式计算顺序不同，由此导致了执行结果不一样。
- DML 语句中含有非确定性的操作。

2.9.5.11 兼容信息

非事务语句是 TiDB 独有的功能，与 MySQL 不兼容。

2.9.5.12 探索更多

- BATCH 语法
- `tidb_nontransactional_ignore_error`

2.10 垃圾回收 (GC)

2.10.1 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。

2.10.1.1 整体流程

一个 TiDB 集群中会有一个 TiDB 实例被选举为 GC leader，GC 的运行由 GC leader 来控制。

GC 会被定期触发。每次 GC 时，首先，TiDB 会计算一个称为 safe point 的时间戳，接下来 TiDB 会在保证 safe point 之后的快照全部拥有正确数据的前提下，删除更早的过期数据。每一轮 GC 分为以下三个步骤：

1. “Resolve Locks” 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)。

2.10.1.2 实现细节

2.10.1.2.1 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。

2.10.1.2.2 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。

2.10.1.2.3 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。

2.10.2 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

2.10.2.1 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
```

2.10.2.2 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。

2.10.2.3 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。

2.10.2.4 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  |
+------+------------------+-----------------------------+-------+
```

2.11 视图

TiDB 支持视图，视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义。使用视图一方面可以对用户只暴露安全的字段及数据，进而保证底层表的敏感字段及数据的安全。另一方面，将频繁出现的复杂查询定义为视图，可以使复杂查询更加简单便捷。

2.11.1 查询视图

查询一个视图和查询一张普通表类似。但是 TiDB 在真正执行查询视图时，会将视图展开成创建视图时定义的 SELECT 语句，进而执行展开后的查询语句。

2.11.2 查看视图的相关信息

通过以下方式，可以查看 view 相关的信息。

2.11.2.1 使用 SHOW CREATE TABLE view_name 或 SHOW CREATE VIEW view_name 语句

示例：

```
show create view v;
```

使用该语句可以查看 view 对应的创建语句，及创建 view 时对应的 character_set_client 及 collation_connection 系统变量值。

```
+--
    ↪ ----+----------------------------------------------------------------------------------------------
    ↪
| View | Create View
    ↪
    ↪ | character_set_client | collation_connection |
+--
    ↪ ----+----------------------------------------------------------------------------------------------
    ↪
| v    | CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`127.0.0.1` SQL SECURITY DEFINER VIEW `v` (`a
    ↪ `) AS SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s` ON `t`.`a`=`s`.`a` | utf8
    ↪                  | utf8_general_ci      |
+--
    ↪ ----+----------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

2.11.2.2 查询 INFORMATION_SCHEMA.VIEWS 表

示例：

```
select * from information_schema.views;
```

通过查询该表可以查看 view 的相关元信息，如 TABLE_CATALOG、TABLE_SCHEMA、TABLE_NAME、VIEW_DEFINITION、CHECK_OPTION、IS_UPDATABLE、DEFINER、SECURITY_TYPE、CHARACTER_SET_CLIENT、COLLATION_CONNECTION 等。

```
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
| TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME | VIEW_DEFINITION
    ↪                                                                | CHECK_OPTION | IS_UPDATABLE |
    ↪ DEFINER        | SECURITY_TYPE | CHARACTER_SET_CLIENT | COLLATION_CONNECTION |
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
| def           | test         | v          | SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s`
    ↪  ON `t`.`a`=`s`.`a` | CASCADED     | NO           | root@127.0.0.1 | DEFINER       | utf8
    ↪                  | utf8_general_ci      |
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

2.11.2.3 查询 HTTP API

示例：

```
curl http://127.0.0.1:10080/schema/test/v
```

通过访问 http://{TiDBIP}:10080/schema/{db}/{view} 可以得到对应 view 的所有元信息。

```
{
 "id": 122,
 "name": {
  "O": "v",
  "L": "v"
 },
 "charset": "utf8",
 "collate": "utf8_general_ci",
 "cols": [
  {
   "id": 1,
   "name": {
    "O": "a",
    "L": "a"
   },
   "offset": 0,
   "origin_default": null,
   "default": null,
```

```
  "default_bit": null,
  "default_is_expr": false,
  "generated_expr_string": "",
  "generated_stored": false,
  "dependences": null,
  "type": {
   "Tp": 0,
   "Flag": 0,
   "Flen": 0,
   "Decimal": 0,
   "Charset": "",
   "Collate": "",
   "Elems": null
  },
  "state": 5,
  "comment": "",
  "hidden": false,
  "version": 0
 }
],
"index_info": null,
"fk_info": null,
"state": 5,
"pk_is_handle": false,
"is_common_handle": false,
"comment": "",
"auto_inc_id": 0,
"auto_id_cache": 0,
"auto_rand_id": 0,
"max_col_id": 1,
"max_idx_id": 0,
"update_timestamp": 416801600091455490,
"ShardRowIDBits": 0,
"max_shard_row_id_bits": 0,
"auto_random_bits": 0,
"pre_split_regions": 0,
"partition": null,
"compression": "",
"view": {
 "view_algorithm": 0,
 "view_definer": {
  "Username": "root",
  "Hostname": "127.0.0.1",
  "CurrentUser": false,
  "AuthUsername": "root",
```

```
   "AuthHostname": "%"
  },
  "view_security": 0,
  "view_select": "SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s` ON `t`.`a`=`s`.`a`",
  "view_checkoption": 1,
  "view_cols": null
 },
 "sequence": null,
 "Lock": null,
 "version": 3,
 "tiflash_replica": null
}
```

2.11.3 示例

以下例子将创建一个视图，并在该视图上进行查询，最后删除该视图。

```
create table t(a int, b int);
```

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

```
insert into t values(1, 1),(2,2),(3,3);
```

```
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
create table s(a int);
```

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

```
insert into s values(2),(3);
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
create view v as select s.a from t left join s on t.a = s.a;
```

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

```
select * from v;
```

```
+------+
| a    |
+------+
| NULL |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

```
drop view v;
```

```
Query OK, 0 rows affected (0.02 sec)
```

2.11.4 局限性

目前 TiDB 中的视图有以下局限性：

- 不支持物化视图。
- TiDB 中视图为只读视图，不支持对视图进行 UPDATE、INSERT、DELETE、TRUNCATE 等写入操作。
- 对已创建的视图仅支持 DROP 的 DDL 操作，即 DROP [VIEW | TABLE]。

2.11.5 扩展阅读

- 创建视图
- 删除视图

2.12 分区表

本文介绍 TiDB 的分区表。

2.12.1 分区类型

本节介绍 TiDB 中的分区类型。当前支持的类型包括Range 分区、List 分区、List COLUMNS 分区和Hash 分区。Range 分区，List 分区和 List COLUMNS 分区可以用于解决业务中大量删除带来的性能问题，支持快速删除分区。Hash 分区则可以用于大量写入场景下的数据打散。

2.12.1.1 Range 分区

一个表按 Range 分区是指，对于表的每个分区中包含的所有行，按分区表达式计算的值都落在给定的范围内。Range 必须是连续的，并且不能有重叠，通过使用 VALUES LESS THAN 进行定义。

下列场景中，假设你要创建一个人事记录的表：

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

你可以根据需求按各种方式进行 Range 分区。其中一种方式是按 store_id 列进行分区：

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

PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

在这个分区模式中，所有 store_id 为 1 到 5 的员工，都存储在分区 p0 里面，store_id 为 6 到 10 的员工则存储在分区 p1 里面。Range 分区要求，分区的定义必须是有序的，按从小到大递增。

新插入一行数据 (72, 'Tom', 'John', '2015-06-25', NULL, NULL, 15) 将会落到分区 p2 里面。但如果你插入一条 store_id 大于 20 的记录，则会报错，因为 TiDB 无法知晓应该将它插入到哪个分区。这种情况下，可以在建表时使用最大值：

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

```
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

MAXVALUE 表示一个比所有整数都大的整数。现在，所有 store_id 列大于等于 16 的记录都会存储在 p3 分区中。

你也可以按员工的职位编号进行分区，也就是使用 job_code 列的值进行分区。假设两位数字编号是用于普通员工，三位数字编号是用于办公室以及客户支持，四位数字编号是管理层职位，那么你可以这样建表：

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

PARTITION BY RANGE (job_code) (
    PARTITION p0 VALUES LESS THAN (100),
    PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (10000)
);
```

在这个例子中，所有普通员工存储在 p0 分区，办公室以及支持人员在 p1 分区，管理者在 p2 分区。

除了可以按 store_id 切分，你还可以按日期切分。例如，假设按员工离职的年份进行分区：

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

PARTITION BY RANGE ( YEAR(separated) ) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1996),
    PARTITION p2 VALUES LESS THAN (2001),
    PARTITION p3 VALUES LESS THAN MAXVALUE
```

```
);
```

在 Range 分区中，可以基于 timestamp 列的值分区，并使用 unix_timestamp() 函数，例如：

```
CREATE TABLE quarterly_report_status (
    report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)

PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

对于 timestamp 列，使用其它的分区表达式是不允许的。

Range 分区在下列条件之一或者多个都满足时，尤其有效：

- 删除旧数据。如果你使用之前的 employees 表的例子，你可以简单使用 ALTER TABLE employees DROP PARTITION p0; 删除所有在 1991 年以前停止继续在这家公司工作的员工记录。这会比使用 DELETE FROM employees WHERE YEAR(separated)<= 1990; 执行快得多。
- 使用包含时间或者日期的列，或者是其它按序生成的数据。
- 频繁查询分区使用的列。例如执行这样的查询 EXPLAIN SELECT COUNT(*)FROM employees WHERE separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id; 时，TiDB 可以迅速确定，只需要扫描 p2 分区的数据，因为其它的分区不满足 where 条件。

2.12.1.2 List 分区

在创建 List 分区表之前，需要先将 session 变量 tidb_enable_list_partition 的值设置为 ON。

```
set @@session.tidb_enable_list_partition = ON
```

此外，还需保证 tidb_enable_table_partition 变量已开启（默认开启）。

List 分区和 Range 分区有很多相似的地方。不同之处主要在于 List 分区中，对于表的每个分区中包含的所有行，按分区表达式计算的值属于给定的数据集合。每个分区定义的数据集合有任意个值，但不能有重复的值，可通过 PARTITION ... VALUES IN (...) 子句对值进行定义。

假设你要创建一张人事记录表，示例如下：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
);
```

假如一共有 20 个商店分布在 4 个地区，如下表所示：

```
| Region  | Store ID Numbers     |
| ------- | -------------------- |
| North   | 1, 2, 3, 4, 5        |
| East    | 6, 7, 8, 9, 10       |
| West    | 11, 12, 13, 14, 15   |
| Central | 16, 17, 18, 19, 20   |
```

如果想把同一个地区商店员工的人事数据都存储在同一个分区中，你可以根据 `store_id` 来创建 List 分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);
```

这样就能方便地在表中添加或删除与特定区域相关的记录。例如，假设东部地区 (East) 所有的商店都卖给了另一家公司，所有该地区商店员工相关的行数据都可以通过 `ALTER TABLE employees TRUNCATE PARTITION pEast` 删除，这比等效的 `DELETE` 语句 `DELETE FROM employees WHERE store_id IN (6, 7, 8, 9, 10)` 执行起来更加高效。

使用 `ALTER TABLE employees DROP PARTITION pEast` 也能删除所有这些行，但同时也会从表的定义中删除分区 `pEast`。那样你还需要使用 `ALTER TABLE ... ADD PARTITION` 语句来还原表的原始分区方案。

与 Range 分区的情况不同，List 分区没有类似的 `MAXVALUE` 分区来存储所有不属于其他 partition 的值。分区表达式的所有期望值都应包含在 `PARTITION ... VALUES IN (...)` 子句中。如果 `INSERT` 语句要插入的值不匹配分区的列值，该语句将执行失败并报错，如下例所示：

```
test> CREATE TABLE t (
    ->   a INT,
    ->   b INT
    -> )
    -> PARTITION BY LIST (a) (
    ->   PARTITION p0 VALUES IN (1, 2, 3),
```

```
    ->   PARTITION p1 VALUES IN (4, 5, 6)
    -> );
Query OK, 0 rows affected (0.11 sec)

test> INSERT INTO t VALUES (7, 7);
ERROR 1525 (HY000): Table has no partition for value 7
```

要忽略以上类型的错误，可以通过使用 IGNORE 关键字。使用该关键字后，就不会插入包含不匹配分区列值的行，但是会插入任何具有匹配值的行，并且不会报错：

```
test> TRUNCATE t;
Query OK, 1 row affected (0.00 sec)

test> INSERT IGNORE INTO t VALUES (1, 1), (7, 7), (8, 8), (3, 3), (5, 5);
Query OK, 3 rows affected, 2 warnings (0.01 sec)
Records: 5  Duplicates: 2  Warnings: 2

test> select * from t;
+------+------+
| a    | b    |
+------+------+
|    5 |    5 |
|    1 |    1 |
|    3 |    3 |
+------+------+
3 rows in set (0.01 sec)
```

2.12.1.3 List COLUMNS 分区

List COLUMNS 分区是 List 分区的一种变体，可以将多个列用作分区键，并且可以将整数类型以外的数据类型的列用作分区列。你还可以使用字符串类型、DATE 和 DATETIME 类型的列。

假设商店员工分别来自以下 12 个城市，想要根据相关规定分成 4 个区域，如下表所示：

```
| Region | Cities                          |
| :----- | ------------------------------- |
| 1      | LosAngeles,Seattle, Houston     |
| 2      | Chicago, Columbus, Boston       |
| 3      | NewYork, LongIsland, Baltimore  |
| 4      | Atlanta, Raleigh, Cincinnati    |
```

使用列表列分区，你可以为员工数据创建一张表，将每行数据存储在员工所在城市对应的分区中，如下所示：

```
CREATE TABLE employees_1 (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
```

```
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT,
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(city) (
    PARTITION pRegion_1 VALUES IN('LosAngeles', 'Seattle', 'Houston'),
    PARTITION pRegion_2 VALUES IN('Chicago', 'Columbus', 'Boston'),
    PARTITION pRegion_3 VALUES IN('NewYork', 'LongIsland', 'Baltimore'),
    PARTITION pRegion_4 VALUES IN('Atlanta', 'Raleigh', 'Cincinnati')
);
```

与 List 分区不同的是，你不需要在 `COLUMNS()` 子句中使用表达式来将列值转换为整数。

List COLUMNS 分区也可以使用 `DATE` 和 `DATETIME` 类型的列进行分区，如以下示例中所示，该示例使用与先前的 `employees_1` 表相同的名称和列，但根据 `hired` 列采用 List COLUMNS 分区：

```
CREATE TABLE employees_2 (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT,
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(hired) (
    PARTITION pWeek_1 VALUES IN('2020-02-01', '2020-02-02', '2020-02-03',
        '2020-02-04', '2020-02-05', '2020-02-06', '2020-02-07'),
    PARTITION pWeek_2 VALUES IN('2020-02-08', '2020-02-09', '2020-02-10',
        '2020-02-11', '2020-02-12', '2020-02-13', '2020-02-14'),
    PARTITION pWeek_3 VALUES IN('2020-02-15', '2020-02-16', '2020-02-17',
        '2020-02-18', '2020-02-19', '2020-02-20', '2020-02-21'),
    PARTITION pWeek_4 VALUES IN('2020-02-22', '2020-02-23', '2020-02-24',
        '2020-02-25', '2020-02-26', '2020-02-27', '2020-02-28')
);
```

另外，你也可以在 `COLUMNS()` 子句中添加多个列，例如：

```
CREATE TABLE t (
    id int,
    name varchar(10)
)
PARTITION BY LIST COLUMNS(id,name) (
     partition p0 values IN ((1,'a'),(2,'b')),
```

```
    partition p1 values IN ((3,'c'),(4,'d')),
    partition p3 values IN ((5,'e'),(null,null))
);
```

2.12.1.4 Hash 分区

Hash 分区主要用于保证数据均匀地分散到一定数量的分区里面。在 Range 分区中你必须为每个分区指定值的范围；在 Hash 分区中，你只需要指定分区的数量。

使用 Hash 分区时，需要在 `CREATE TABLE` 后面添加 `PARTITION BY HASH (expr)`，其中 `expr` 是一个返回整数的表达式。当这一列的类型是整数类型时，它可以是一个列名。此外，你很可能还需要加上 `PARTITIONS num`，其中 `num` 是一个正整数，表示将表划分多少分区。

下面的语句将创建一个 Hash 分区表，按 `store_id` 分成 4 个分区：

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

PARTITION BY HASH(store_id)
PARTITIONS 4;
```

如果不指定 `PARTITIONS num`，默认的分区数量为 1。

你也可以使用一个返回整数的 SQL 表达式。例如，你可以按入职年份分区：

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

PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

最高效的 Hash 函数是作用在单列上，并且函数的单调性是跟列的值是一样递增或者递减的。

例如，date_col 是类型为 DATE 的列，表达式 TO_DAYS(date_col) 的值是直接随 date_col 的值变化的。YEAR(date_col) 跟 TO_DAYS(date_col) 就不太一样，因为不是每次 date_col 变化时 YEAR(date_col) 都会得到不同的值。

作为对比，假设我们有一个类型是 INT 的 int_col 的列。考虑一下表达式 POW(5-int_col,3)+ 6，这并不是一个比较好的 Hash 函数，因为随着 int_col 的值的变化，表达式的结果不会成比例地变化。改变 int_col 的值会使表达式的结果的值变化巨大。例如，int_col 从 5 变到 6 表达式的结果变化是 -1，但是从 6 变到 7 的时候表达式的值的变化是 -7。

总而言之，表达式越接近 y = cx 的形式，它越是适合作为 Hash 函数。因为表达式越是非线性的，在各个分区上面的数据的分布越是倾向于不均匀。

理论上，Hash 分区也是可以做分区裁剪的。而实际上对于多列的情况，实现很难并且计算很耗时。因此，不推荐 Hash 分区在表达式中涉及多列。

使用 PARTITIION BY HASH 的时候，TiDB 通过表达式的结果做“取余”运算，决定数据落在哪个分区。换句话说，如果分区表达式是 expr，分区数是 num，则由 MOD(expr, num) 决定存储的分区。假设 t1 定义如下：

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY HASH( YEAR(col3) )
    PARTITIONS 4;
```

向 t1 插入一行数据，其中 col3 列的值是 ‘2005-09-15’，这条数据会被插入到分区 1 中：

```
MOD(YEAR('2005-09-01'),4)
=  MOD(2005,4)
=  1
```

2.12.1.5 分区对 NULL 值的处理

TiDB 允许计算结果为 NULL 的分区表达式。注意，NULL 不是一个整数类型，NULL 小于所有的整数类型值，正如 ORDER BY 的规则一样。

2.12.1.5.1 Range 分区对 NULL 的处理

如果插入一行到 Range 分区表，它的分区列的计算结果是 NULL，那么这一行会被插入到最小的那个分区。

```
CREATE TABLE t1 (
    c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY RANGE(c1) (
    PARTITION p0 VALUES LESS THAN (0),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
select * from t1 partition(p0);
```

```
+------|--------+
| c1   | c2     |
+------|--------+
| NULL | mothra |
+------|--------+
1 row in set (0.00 sec)
```

```
select * from t1 partition(p1);
```

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

```
select * from t1 partition(p2);
```

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

删除 p0 后验证：

```
alter table t1 drop partition p0;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
select * from t1;
```

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

2.12.1.5.2 Hash 分区对 NULL 的处理

在 Hash 分区中 NULL 值的处理有所不同，如果分区表达式的计算结果为 NULL，它会被当作 0 值处理。

```
CREATE TABLE th (
    c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY HASH(c1)
PARTITIONS 2;
```

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

```
INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
```

```
Query OK, 2 rows affected (0.04 sec)
```

```
select * from th partition (p0);
```

```
+------|--------+
| c1   | c2     |
+------|--------+
| NULL | mothra |
|    0 | gigan  |
+------|--------+
2 rows in set (0.00 sec)
```

```
select * from th partition (p1);
```

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

可以看到，插入的记录 `(NULL, 'mothra')` 跟 `(0, 'gigan')` 落在了同一个分区。

> 注意：
>
> 这里 Hash 分区对 NULL 的处理跟 MySQL 的文档描述一致，但是跟 MySQL 的实际行为并不一致。也就是说，MySQL 的文档跟它的实现并不一致。
>
> TiDB 的最终行为以本文档描述为准。

2.12.2 分区管理

对于 `LIST` 和 `RANGE` 分区表，通过 `ALTER TABLE <表名> ADD PARTITION (<分区说明>)` 或 `ALTER TABLE <表名> DROP PARTITION <分区列表>` 语句，可以执行添加和删除分区的操作。

对于 `LIST` 和 `RANGE` 分区表，暂不支持 `REORGANIZE PARTITION` 语句。

对于 `HASH` 分区表，暂不支持 `COALESCE PARTITION` 和 `ADD PARTITION` 语句。

`EXCHANGE PARTITION` 语句用来交换分区和非分区表，类似于重命名表如 `RENAME TABLE t1 TO t1_tmp, t2 TO t1, t1_tmp TO t2` 的操作。

例如，`ALTER TABLE partitioned_table EXCHANGE PARTITION p1 WITH TABLE non_partitioned_table` 交换的是 `p1` 分区的 `non_partitioned_table` 表和 `partitioned_table` 表。

确保要交换入分区中的所有行与分区定义匹配；否则，你将无法找到这些行，并导致意外情况出现。

> 警告：
>
> `EXCHANGE PARTITION` 是实验性功能，不建议在生产环境中使用。要启用 `EXCHANGE PARTITION`，将系统变量 `tidb_enable_exchange_partition` 设置为 `ON`。

2.12.2.1 Range 分区管理

创建分区表：

```
CREATE TABLE members (
    id INT,
    fname VARCHAR(25),
    lname VARCHAR(25),
    dob DATE
)

PARTITION BY RANGE( YEAR(dob) ) (
    PARTITION p0 VALUES LESS THAN (1980),
    PARTITION p1 VALUES LESS THAN (1990),
    PARTITION p2 VALUES LESS THAN (2000)
);
```

删除分区：

```
ALTER TABLE members DROP PARTITION p2;
```

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

清空分区：

```
ALTER TABLE members TRUNCATE PARTITION p1;
```

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

> **注意：**
>
> `ALTER TABLE ... REORGANIZE PARTITION` 在 TiDB 中暂不支持。

添加分区：

```
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2010));
```

Range 分区中，`ADD PARTITION` 只能在分区列表的最后面添加，如果是添加到已存在的分区范围则会报错：

```
ALTER TABLE members
    ADD PARTITION (
    PARTITION n VALUES LESS THAN (1970));
```

```
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly »
   increasing for each partition
```

2.12.2.2 Hash 分区管理

跟 Range 分区不同，Hash 分区不能够 DROP PARTITION。

目前 TiDB 的实现暂时不支持 ALTER TABLE ... COALESCE PARTITION。对于暂不支持的分区管理语句，TiDB 会返回错误。

```
alter table members optimize partition p0;
```

```
ERROR 8200 (HY000): Unsupported optimize partition
```

2.12.3 分区裁剪

有一个优化叫做“分区裁剪”，它基于一个非常简单的概念：不需要扫描那些匹配不上的分区。

假设创建一个分区表 t1：

```
CREATE TABLE t1 (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    region_code TINYINT UNSIGNED NOT NULL,
    dob DATE NOT NULL
)

PARTITION BY RANGE( region_code ) (
    PARTITION p0 VALUES LESS THAN (64),
    PARTITION p1 VALUES LESS THAN (128),
    PARTITION p2 VALUES LESS THAN (192),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

如果你想获得这个 select 语句的结果：

```
SELECT fname, lname, region_code, dob
    FROM t1
    WHERE region_code > 125 AND region_code < 130;
```

很显然，结果必然是在分区 p1 或者 p2 里面，也就是说，我们只需要在 p1 和 p2 里面去搜索匹配的行。去掉不必要的分区就是所谓的裁剪。优化器如果能裁剪掉一部分的分区，则执行会快于处理整个不做分区的表的相同查询。

优化器可以通过 where 条件裁剪的两个场景：

- partition_column = constant
- partition_column IN (constant1, constant2, ⋯, constantN)

分区裁剪暂不支持 LIKE 语句。

2.12.3.1 分区裁剪生效的场景

1. 分区裁剪需要使用分区表上面的查询条件，所以根据优化器的优化规则，如果查询条件不能下推到分区表，则相应的查询语句无法执行分区裁剪。

   例如：

   ```
   create table t1 (x int) partition by range (x) (
       partition p0 values less than (5),
       partition p1 values less than (10));
   create table t2 (x int);
   ```

   ```
   explain select * from t1 left join t2 on t1.x = t2.x where t2.x > 5;
   ```

   在这个查询中，外连接可以简化成内连接，然后由 `t1.x = t2.x` 和 `t2.x > 5` 可以推出条件 `t1.x > 5`，于是可以分区裁剪并且只使用 `p1` 分区。

   ```
   explain select * from t1 left join t2 on t1.x = t2.x and t2.x > 5;
   ```

   这个查询中的 `t2.x > 5` 条件不能下推到 `t1` 分区表上面，因此 `t1` 无法分区裁剪。

2. 由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

   例如：

   ```
   create table t1 (x int) partition by range (x) (
       partition p0 values less than (5),
       partition p1 values less than (10));
   ```

   ```
   explain select * from t2 where x < (select * from t1 where t2.x < t1.x and t2.x < 2);
   ```

   这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x > val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

3. 由于当前实现中的一处限制，对于查询条件无法下推到 TiKV 的表达式，不支持分区裁剪。

   对于一个函数表达式 `fn(col)`，如果 TiKV 支持这个函数 `fn`，则在查询优化做谓词下推的时候，`fn(col)` 会被推到叶子节点（也就是分区），因而能够执行分区裁剪。

   如果 TiKV 不支持 `fn`，则优化阶段不会把 `fn(col)` 推到叶子节点，而是在叶子上面连接一个 Selection 节点，分区裁剪的实现没有处理这种父节点的 Selection 中的条件，因此对不能下推到 TiKV 的表达式不支持分区裁剪。

4. 对于 Hash 分区类型，只有等值比较的查询条件能够支持分区裁剪。

5. 对于 Range 分区类型，分区表达式必须是 `col` 或者 `fn(col)` 的简单形式，查询条件是 `>`、`<`、`=`、`>=`、`<=` 时才能支持分区裁剪。如果分区表达式是 `fn(col)` 形式，还要求 `fn` 必须是单调函数，才有可能分区裁剪。

   这里单调函数是指某个函数 `fn` 满足条件：对于任意 `x` `y`，如果 `x > y`，则 `fn(x)> fn(y)`。

   这种是严格递增的单调函数，非严格递增的单调函数也可以符合分区裁剪要求，只要函数 `fn` 满足：对于任意 `x` `y`，如果 `x > y`，则 `fn(x)>= fn(y)`。

   理论上所有满足单调条件（严格或者非严格）的函数都是可以支持分区裁剪。实际上，目前 TiDB 已经支持的单调函数只有：

```
unix_timestamp
to_days
```

例如，分区表达式是简单列的情况：

```
create table t (id int) partition by range (id) (
    partition p0 values less than (5),
    partition p1 values less than (10));
select * from t where t > 6;
```

分区表达式是 `fn(col)` 的形式，`fn` 是我们支持的单调函数 `to_days`：

```
create table t (dt datetime) partition by range (to_days(id)) (
    partition p0 values less than (to_days('2020-04-01')),
    partition p1 values less than (to_days('2020-05-01')));
select * from t where t > '2020-04-18';
```

有一处例外是 `floor(unix_timestamp(ts))` 作为分区表达式，TiDB 针对这个场景做了特殊处理，可以支持分区裁剪。

```
create table t (ts timestamp(3) not null default current_timestamp(3))
partition by range (floor(unix_timestamp(ts))) (
    partition p0 values less than (unix_timestamp('2020-04-01 00:00:00')),
    partition p1 values less than (unix_timestamp('2020-05-01 00:00:00')));
select * from t where t > '2020-04-18 02:00:42.123';
```

2.12.4 分区选择

SELECT 语句中支持分区选择。实现通过使用一个 PARTITION 选项实现。

```
CREATE TABLE employees  (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    fname VARCHAR(25) NOT NULL,
    lname VARCHAR(25) NOT NULL,
    store_id INT NOT NULL,
    department_id INT NOT NULL
)

PARTITION BY RANGE(id)  (
    PARTITION p0 VALUES LESS THAN (5),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN (15),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);

INSERT INTO employees VALUES
```

```
    ('', 'Bob', 'Taylor', 3, 2), ('', 'Frank', 'Williams', 1, 2),
    ('', 'Ellen', 'Johnson', 3, 4), ('', 'Jim', 'Smith', 2, 4),
    ('', 'Mary', 'Jones', 1, 1), ('', 'Linda', 'Black', 2, 3),
    ('', 'Ed', 'Jones', 2, 1), ('', 'June', 'Wilson', 3, 1),
    ('', 'Andy', 'Smith', 1, 3), ('', 'Lou', 'Waters', 2, 4),
    ('', 'Jill', 'Stone', 1, 4), ('', 'Roger', 'White', 3, 2),
    ('', 'Howard', 'Andrews', 1, 2), ('', 'Fred', 'Goldberg', 3, 3),
    ('', 'Barbara', 'Brown', 2, 3), ('', 'Alice', 'Rogers', 2, 2),
    ('', 'Mark', 'Morgan', 3, 3), ('', 'Karen', 'Cole', 3, 2);
```

你可以查看存储在分区 p1 中的行：

```
SELECT * FROM employees PARTITION (p1);
```

```
+----|-------|--------|----------|---------------+
| id | fname | lname  | store_id | department_id |
+----|-------|--------|----------|---------------+
|  5 | Mary  | Jones  |        1 |             1 |
|  6 | Linda | Black  |        2 |             3 |
|  7 | Ed    | Jones  |        2 |             1 |
|  8 | June  | Wilson |        3 |             1 |
|  9 | Andy  | Smith  |        1 |             3 |
+----|-------|--------|----------|---------------+
5 rows in set (0.00 sec)
```

如果希望获得多个分区中的行，可以提供分区名的列表，用逗号隔开。例如，`SELECT * FROM employees PARTITION (p1, p2)` 返回分区 p1 和 p2 的所有行。

使用分区选择时，仍然可以使用 where 条件，以及 ORDER BY 和 LIMIT 等选项。使用 HAVING 和 GROUP BY 等聚合选项也是支持的。

```
SELECT * FROM employees PARTITION (p0, p2)
    WHERE lname LIKE 'S%';
```

```
+----|-------|-------|----------|---------------+
| id | fname | lname | store_id | department_id |
+----|-------|-------|----------|---------------+
|  4 | Jim   | Smith |        2 |             4 |
| 11 | Jill  | Stone |        1 |             4 |
+----|-------|-------|----------|---------------+
2 rows in set (0.00 sec)
```

```
SELECT id, CONCAT(fname, ' ', lname) AS name
    FROM employees PARTITION (p0) ORDER BY lname;
```

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

```
| id | name           |
+----|----------------+
|  3 | Ellen Johnson  |
|  4 | Jim Smith      |
|  1 | Bob Taylor     |
|  2 | Frank Williams |
+----|----------------+
4 rows in set (0.06 sec)
```

```
SELECT store_id, COUNT(department_id) AS c
    FROM employees PARTITION (p1,p2,p3)
    GROUP BY store_id HAVING c > 4;
```

```
+---|----------+
| c | store_id |
+---|----------+
| 5 |        2 |
| 5 |        3 |
+---|----------+
2 rows in set (0.00 sec)
```

分支选择支持所有类型的分区表，无论是 Range 分区或是 Hash 分区等。对于 Hash 分区，如果没有指定分区名，会自动使用 p0、p1、p2、……、或 pN-1 作为分区名。

在 INSERT ... SELECT 的 SELECT 中也是可以使用分区选择的。

2.12.5 分区的约束和限制

本节介绍当前 TiDB 分区表的一些约束和限制。

2.12.5.1 分区键，主键和唯一键

本节讨论分区键，主键和唯一键之间的关系。一句话总结它们之间的关系要满足的规则：分区表的每个唯一键，必须包含分区表达式中用到的所有列。

> every unique key on the table must use every column in the table’s partitioning expression.

这里所指的唯一也包含了主键，因为根据主键的定义，主键必须是唯一的。例如，下面这些建表语句就是无效的：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
```

```
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

它们都是有唯一键但没有包含所有分区键的。

下面是一些合法的语句的例子：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

下例中会产生一个报错：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

```
ERROR 1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

原因是 `col1` 和 `col3` 出现在分区键中，但是几个唯一键定义并没有完全包含它们，做如下修改后语句即为合法：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3),
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

下面这个表就没法做分区了，因为无论如何都不可能找到满足条件的分区键：

```
CREATE TABLE t4 (
    col1 INT NOT NULL,
    col2 INT NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3),
    UNIQUE KEY (col2, col4)
);
```

根据定义，主键也是唯一键，下面两个建表语句是无效的：

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
```

```
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

以上两个例子中，主键都没有包含分区表达式中的全部的列，在主键中补充缺失列后语句即为合法：

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

如果既没有主键，也没有唯一键，则不存在这个限制。

DDL 变更时，添加唯一索引也需要考虑到这个限制。比如创建了这样一个表：

```
CREATE TABLE t_no_pk (c1 INT, c2 INT)
    PARTITION BY RANGE(c1) (
        PARTITION p0 VALUES LESS THAN (10),
        PARTITION p1 VALUES LESS THAN (20),
        PARTITION p2 VALUES LESS THAN (30),
        PARTITION p3 VALUES LESS THAN (40)
    );
```

```
Query OK, 0 rows affected (0.12 sec)
```

通过 ALTER TABLE 添加非唯一索引是可以的。但是添加唯一索引时，唯一索引里面必须包含 c1 列。

使用分区表时，前缀索引是不能指定为唯一属性的：

```
CREATE TABLE t (a varchar(20), b blob,
    UNIQUE INDEX (a(5)))
    PARTITION by range columns (a) (
    PARTITION p0 values less than ('aaaaa'),
    PARTITION p1 values less than ('bbbbb'),
    PARTITION p2 values less than ('ccccc'));
```

```
ERROR 1503 (HY000): A UNIQUE INDEX must include all columns in the table's partitioning function
```

2.12.5.2 关于函数的分区限制

只有以下函数可以用于分区表达式：

```
ABS()
CEILING()
DATEDIFF()
DAY()
DAYOFMONTH()
DAYOFWEEK()
DAYOFYEAR()
EXTRACT() (see EXTRACT() function with WEEK specifier)
FLOOR()
HOUR()
MICROSECOND()
MINUTE()
MOD()
MONTH()
QUARTER()
SECOND()
TIME_TO_SEC()
TO_DAYS()
```

```
TO_SECONDS()
UNIX_TIMESTAMP() (with TIMESTAMP columns)
WEEKDAY()
YEAR()
YEARWEEK()
```

2.12.5.3 兼容性

目前 TiDB 支持 Range 分区、List 分区、List COLUMNS 分区和 Hash 分区，其它的 MySQL 分区类型（例如 Key 分区）尚不支持。

对于 Range Columns 类型的分区表，目前只支持单列的场景。

分区管理方面，只要底层实现可能会涉及数据挪动的操作，目前都暂不支持。包括且不限于：调整 Hash 分区表的分区数量，修改 Range 分区表的范围，合并分区，交换分区等。

对于暂不支持的分区类型，在 TiDB 中建表时会忽略分区信息，以普通表的形式创建，并且会报 Warning。

Load Data 暂时不支持分区选择。

```
create table t (id int, val int) partition by hash(id) partitions 4;
```

普通的 Load Data 操作在 TiDB 中是支持的，如下：

```
load local data infile "xxx" into t ...
```

但 Load Data 不支持分区选择操作：

```
load local data infile "xxx" into t partition (p1)...
```

对于分区表，`select * from t` 的返回结果是分区之间无序的。这跟 MySQL 不同，MySQL 的返回结果是分区之间有序，分区内部无序。

```
create table t (id int, val int) partition by range (id) (
    partition p0 values less than (3),
    partition p1 values less than (7),
    partition p2 values less than (11));
```

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

```
insert into t values (1, 2), (3, 4),(5, 6),(7,8),(9,10);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

TiDB 每次返回结果会不同，例如：

```
select * from t;
```

```
+------|------+
| id   | val  |
+------|------+
|    7 |    8 |
|    9 |   10 |
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
+------|------+
5 rows in set (0.00 sec)
```

MySQL 的返回结果：

```
select * from t;
```

```
+------|------+
| id   | val  |
+------|------+
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
|    7 |    8 |
|    9 |   10 |
+------|------+
5 rows in set (0.00 sec)
```

环境变量 tidb_enable_list_partition 可以控制是否启用分区表功能。如果该变量设置为 OFF，则建表时会忽略分区信息，以普通表的方式建表。

该变量仅作用于建表，已经建表之后再修改该变量无效。详见系统变量和语法。

2.12.5.4 动态裁剪模式

TiDB 访问分区表有两种模式，dynamic 和 static，目前默认使用 static 模式。如果想开启 dynamic 模式，需要手动将 tidb_partition_prune_mode 设置为 dynamic。

```
set @@session.tidb_partition_prune_mode = 'dynamic'
```

普通查询和手动 analyze 使用的是 session 级别的 tidb_partition_prune_mode 设置，后台的 auto-analyze 使用的是 global 级别的 tidb_partition_prune_mode 设置。

静态裁剪模式下，分区表使用的是分区级别的统计信息，而动态裁剪模式下，分区表用的是表级别的汇总统计信息，即 GlobalStats。详见动态裁剪模式下的分区表统计信息。

从 static 静态裁剪模式切到 dynamic 动态裁剪模式时，需要手动检查和收集统计信息。在刚切换到 dynamic 时，分区表上仍然只有分区的统计信息，需要等到全局 dynamic 动态裁剪模式开启后的下一次 auto-analyze 周期，才会更新生成汇总统计信息。

```
set session tidb_partition_prune_mode = 'dynamic';
show stats_meta where table_name like "t";
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          | p0             | 2022-05-27 20:23:34 |            1 |         2 |
| test    | t          | p1             | 2022-05-27 20:23:34 |            2 |         4 |
| test    | t          | p2             | 2022-05-27 20:23:34 |            2 |         4 |
+---------+------------+----------------+---------------------+--------------+-----------+
3 rows in set (0.01 sec)
```

为保证开启全局 dynamic 动态裁剪模式时，SQL 可以用上正确的统计信息，此时需要手动触发一次 analyze 来更新汇总统计信息，可以通过 analyze 表或者单个分区来更新。

```
analyze table t partition p1;
show stats_meta where table_name like "t";
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          | global         | 2022-05-27 20:50:53 |            0 |         5 |
| test    | t          | p0             | 2022-05-27 20:23:34 |            1 |         2 |
| test    | t          | p1             | 2022-05-27 20:50:52 |            0 |         2 |
| test    | t          | p2             | 2022-05-27 20:50:08 |            0 |         2 |
+---------+------------+----------------+---------------------+--------------+-----------+
4 rows in set (0.00 sec)
```

若 analyze 过程中提示如下 warning，说明分区的统计信息之间存在不一致，需要重新收集分区或整个表统计信息。

```
| Warning | 8244 | Build table: `t` column: `a` global-level stats failed due to missing
    ↪ partition-level column stats, please run analyze table to refresh columns of all
    ↪ partitions
```

也可以使用脚本来统一更新所有的分区表统计信息，详见为动态裁剪模式更新所有分区表的统计信息。

表级别统计信息准备好后，即可开启全局的动态裁剪模式。全局动态裁剪模式，对全局所有的 SQL 和对后台的统计信息自动收集（即 auto analyze）起作用。

```
set global tidb_partition_prune_mode = dynamic
```

在 static 模式下，TiDB 用多个算子单独访问每个分区，然后通过 Union 将结果合并起来。下面例子进行了一个简单的读取操作，可以发现 TiDB 用 Union 合并了对应两个分区的结果：

```
mysql> create table t1(id int, age int, key(id)) partition by range(id) (
    ->     partition p0 values less than (100),
```

```
    ->     partition p1 values less than (200),
    ->     partition p2 values less than (300),
    ->     partition p3 values less than (400));
Query OK, 0 rows affected (0.01 sec)

mysql> explain select * from t1 where id < 150;
```

```
+-----------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
| id                          | estRows  | task      | access object         | operator info
    ↪                  |
+-----------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
| PartitionUnion_9            | 6646.67  | root      |                       |
    ↪                                |
| ├─TableReader_12             | 3323.33  | root      |                        | data:
    ↪ Selection_11              |
| │ └─Selection_11             | 3323.33  | cop[tikv] |                        | lt(test.t1.id,
    ↪ 150)            |
| │   └─TableFullScan_10       | 10000.00 | cop[tikv] | table:t1, partition:p0 | keep order:
    ↪ false, stats:pseudo |
| └─TableReader_18             | 3323.33  | root      |                        | data:
    ↪ Selection_17              |
|   └─Selection_17             | 3323.33  | cop[tikv] |                        | lt(test.t1.id,
    ↪ 150)            |
|     └─TableFullScan_16       | 10000.00 | cop[tikv] | table:t1, partition:p1 | keep order:
    ↪ false, stats:pseudo |
+-----------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
7 rows in set (0.00 sec)
```

在 dynamic 模式下，每个算子都支持直接访问多个分区，所以 TiDB 不再使用 Union。

```
mysql> set @@session.tidb_partition_prune_mode = 'dynamic';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select * from t1 where id < 150;
+--
    ↪ ----------------------+----------+-----------+-----------------+-----------------------------+
    ↪
| id                      | estRows  | task      | access object   | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+-----------------+-----------------------------+
    ↪
```

```
| TableReader_7           | 3323.33  | root      | partition:p0,p1 | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |                 | lt(test.t1.id, 150)
    ↪            |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1        | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+-----------------+----------------------------+
    ↪
3 rows in set (0.00 sec)
```

从以上查询结果可知，执行计划中的 Union 消失了，分区裁剪依然生效，且执行计划只访问了 p0 和 p1 两个分区。

dynamic 模式让执行计划更简单清晰，省略 Union 操作可提高执行效率，还可避免 Union 并发管理的问题。此外 dynamic 模式下，执行计划可以使用 IndexJoin 的方式，这在 static 模式下是无法实现的。请看下面的例子：

示例一：以下示例在 static 模式下执行计划带 IndexJoin 的查询。

```
mysql> create table t1 (id int, age int, key(id)) partition by range(id)
    -> (partition p0 values less than (100),
    ->  partition p1 values less than (200),
    ->  partition p2 values less than (300),
    ->  partition p3 values less than (400));
Query OK, 0 rows affected (0,08 sec)
mysql> create table t2 (id int, code int);

Query OK, 0 rows affected (0.01 sec)

mysql> set @@tidb_partition_prune_mode = 'static';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select /*+ TIDB_INLJ(t1, t2) */ t1.* from t1, t2 where t2.code = 0 and t2.id = t1.
    ↪ id;
+--
    ↪ ------------------------------+----------+-----------+-----------------------+-------------------------
    ↪
| id                            | estRows  | task      | access object         | operator info
    ↪                                |
+--
    ↪ ------------------------------+----------+-----------+-----------------------+-------------------------
    ↪
| HashJoin_13                   | 12.49    | root      |                       | inner join,
    ↪ equal:[eq(test.t1.id, test.t2.id)] |
| ├─TableReader_42(Build)       | 9.99     | root      |                       | data:
    ↪ Selection_41                           |
| │ └─Selection_41              | 9.99     | cop[tikv] |                       | eq(test.t2.
```

```
    ↪ code, 0), not(isnull(test.t2.id))   |
| |   └─TableFullScan_40         | 10000.00 | cop[tikv] | table:t2               | keep order:
    ↪ false, stats:pseudo                 |
| └─PartitionUnion_15(Probe)    | 39960.00 | root      |                        |
    ↪                                                 |
|   ├─TableReader_18            | 9990.00  | root      |                        | data:
    ↪ Selection_17                           |
|   | └─Selection_17            | 9990.00  | cop[tikv] |                        | not(isnull(
    ↪ test.t1.id))                        |
|   |   └─TableFullScan_16      | 10000.00 | cop[tikv] | table:t1, partition:p0 | keep order:
    ↪ false, stats:pseudo                 |
|   ├─TableReader_24            | 9990.00  | root      |                        | data:
    ↪ Selection_23                           |
|   | └─Selection_23            | 9990.00  | cop[tikv] |                        | not(isnull(
    ↪ test.t1.id))                        |
|   |   └─TableFullScan_22      | 10000.00 | cop[tikv] | table:t1, partition:p1 | keep order:
    ↪ false, stats:pseudo                 |
|   ├─TableReader_30            | 9990.00  | root      |                        | data:
    ↪ Selection_29                           |
|   | └─Selection_29            | 9990.00  | cop[tikv] |                        | not(isnull(
    ↪ test.t1.id))                        |
|   |   └─TableFullScan_28      | 10000.00 | cop[tikv] | table:t1, partition:p2 | keep order:
    ↪ false, stats:pseudo                 |
|   └─TableReader_36            | 9990.00  | root      |                        | data:
    ↪ Selection_35                           |
|     └─Selection_35            | 9990.00  | cop[tikv] |                        | not(isnull(
    ↪ test.t1.id))                        |
|       └─TableFullScan_34      | 10000.00 | cop[tikv] | table:t1, partition:p3 | keep order:
    ↪ false, stats:pseudo                 |
+--
    ↪ ------------------------------+----------+-----------+------------------------+----------------------------
    ↪
17 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+--
    ↪ -------+------+-----------------------------------------------------------------------------------+
    ↪
| Level   | Code | Message
    ↪                                                                                 |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------+
    ↪
| Warning | 1815 | Optimizer Hint /*+ INL_JOIN(t1, t2) */ or /*+ TIDB_INLJ(t1, t2) */ is
    ↪ inapplicable |
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------+
    ↪
1 row in set (0,00 sec)
```

从以上示例一结果可知，即使使用了 TIDB_INLJ 的 hint，也无法使得带分区表的查询选上带 IndexJoin 的执行计划。

示例二：以下示例在 dynamic 模式下尝试执行计划带 IndexJoin 的查询。

```
mysql> set @@tidb_partition_prune_mode = 'dynamic';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select /*+ TIDB_INLJ(t1, t2) */ t1.* from t1, t2 where t2.code = 0 and t2.id = t1.
    ↪ id;
+--
    ↪ ------------------------------+----------+-----------+----------------------+----------------------------
    ↪
| id                            | estRows  | task      | access object          | operator info
    ↪
    ↪
    ↪ |
+--
    ↪ ------------------------------+----------+-----------+----------------------+----------------------------
    ↪
| IndexJoin_11                  | 12.49    | root      |                        | inner join,
    ↪ inner:IndexLookUp_10, outer key:test.t2.id, inner key:test.t1.id, equal cond:eq(test.t2.
    ↪ id, test.t1.id) |
| ├─TableReader_16(Build)       | 9.99     | root      |                        | data:
    ↪ Selection_15
    ↪
    ↪ |
| │ └─Selection_15              | 9.99     | cop[tikv] |                        | eq(test.t2.
    ↪ code, 0), not(isnull(test.t2.id))
    ↪                                                                                  |
| │   └─TableFullScan_14        | 10000.00 | cop[tikv] | table:t2               | keep order:
    ↪ false, stats:pseudo
    ↪                                                                                       |
| └─IndexLookUp_10(Probe)       | 1.25     | root      | partition:all          |
    ↪
    ↪ |
|   ├─Selection_9(Build)        | 1.25     | cop[tikv] |                        | not(isnull(
    ↪ test.t1.id))
    ↪
    ↪ |
|   │ └─IndexRangeScan_7        | 1.25     | cop[tikv] | table:t1, index:id(id) | range:
```

```
    ↪ decided by [eq(test.t1.id, test.t2.id)], keep order:false, stats:pseudo
    ↪                                       |
|   └─TableRowIDScan_8(Probe)     | 1.25     | cop[tikv] | table:t1               | keep order:
    ↪ false, stats:pseudo
    ↪                                                                                 |
+--
    ↪ ------------------------------+----------+-----------+-----------------------+-----------------------------
    ↪
8 rows in set (0.00 sec)
```

从示例二结果可知，开启 dynamic 模式后，带 IndexJoin 的计划在执行查询时被选上。

目前，静态和动态裁剪模式都不支持执行计划缓存。

2.12.5.4.1 为动态裁剪模式更新所有分区表的统计信息

1. 找到所有的分区表：

```
select distinct concat(TABLE_SCHEMA,'.',TABLE_NAME)
    from information_schema.PARTITIONS
    where TABLE_SCHEMA not in ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        ↪ METRICS_SCHEMA');
```

```
+-------------------------------------+
| concat(TABLE_SCHEMA,'.',TABLE_NAME) |
+-------------------------------------+
| test.t                              |
+-------------------------------------+
1 row in set (0.02 sec)
```

2. 生成所有分区表的更新统计信息的语句：

```
select distinct concat('ANALYZE TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;')
    from information_schema.PARTITIONS
    where TABLE_SCHEMA not in ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        ↪ METRICS_SCHEMA');
```

```
+-----------------------------------------------------------------------+
| concat('ANALYZE TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;') |
+-----------------------------------------------------------------------+
| ANALYZE TABLE test.t ALL COLUMNS;                                     |
+-----------------------------------------------------------------------+
1 row in set (0.01 sec)
```

   可以按需将 `ALL COLUMNS` 改为实际需要的列。

3. 将批量更新语句导出到文件：

```
mysql --host xxxx --port xxxx -u root -p -e "select distinct concat('ANALYZE TABLE ',
    ↪ TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;') \
    from information_schema.PARTITIONS \
    where TABLE_SCHEMA not in ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        ↪ METRICS_SCHEMA');" | tee gatherGlobalStats.sql
```

4. 执行批量更新：

   在运行 source 命令之前处理 SQL 文件：

```
sed -i "" '1d' gatherGlobalStats.sql --- mac
sed -i '1d' gatherGlobalStats.sql --- linux
```

```
SET session tidb_partition_prune_mode = dynamic;
source gatherGlobalStats.sql
```

2.13 临时表

TiDB 在 v5.3.0 版本中引入了临时表功能。该功能针对业务中间计算结果的临时存储问题，让用户免于频繁地建表和删表等操作。用户可将业务上的中间计算数据存入临时表，用完数据后 TiDB 自动清理回收临时表。这避免了用户业务过于复杂，减少了表管理开销，并提升了性能。

本文介绍了 TiDB 临时表的使用场景、临时表类型、使用示例、限制临时表内存占用的方法、与其他 TiDB 功能的兼容性限制。

2.13.1 使用场景

TiDB 临时表主要应用于以下业务场景：

- 缓存业务的中间临时数据，计算完成后将数据转储至普通表，临时表会自动释放。
- 短期内对同一数据进行多次 DML 操作。例如在电商购物车应用中，添加、修改、删除商品及完成结算，并移除购物车信息。
- 快速批量导入中间临时数据，提升导入临时数据的性能。
- 批量更新数据。将数据批量导入到数据库的临时表，修改完成后再导出到文件。

2.13.2 临时表类型

TiDB 的临时表分为本地临时表和全局临时表：

- 本地临时表的表定义和表内数据只对当前会话可见，适用于暂存会话内的中间数据。
- 全局临时表的表定义对整个 TiDB 集群可见，表内数据只对当前事务可见，适用于暂存事务内的中间数据。

2.13.3 本地临时表

本地临时表的语义与 MySQL 临时表一致，它有以下特性：

- 本地临时表的表定义不持久化，只在创建该表的会话内可见，其他会话无法访问该本地临时表
- 不同会话可以创建同名的本地临时表，各会话只会读写该会话内创建的本地临时表
- 本地临时表的数据对会话内的所有事务可见
- 在会话结束后，该会话创建的本地临时表会被自动删除
- 本地临时表可以与普通表同名，此时在 DDL 和 DML 语句中，普通表被隐藏，直到本地临时表被删除

用户可通过 `CREATE TEMPORARY TABLE` 语句创建本地临时表，通过 `DROP TABLE` 或 `DROP TEMPORARY TABLE` 语句删除本地临时表。

不同于 MySQL，TiDB 本地临时表都是外部表，SQL 语句不会创建内部临时表。

2.13.3.1 本地临时表使用示例

> 注意：
>
> - 使用 TiDB 中的临时表前，注意临时表与其他 TiDB 功能的兼容性限制以及与 MySQL 临时表的兼容性。
> - 如果在 v5.3.0 升级前创建了本地临时表，这些临时表实际为普通表，在升级后也会被 TiDB 当成普通表处理。

假设已存在普通表 `users`:

```
CREATE TABLE users (
    id BIGINT,
    name VARCHAR(100),
    PRIMARY KEY(id)
);
```

在会话 A 中创建本地临时表 `users`，不会有名字冲突。会话 A 访问 `users` 时，访问的是本地临时表 `users`。

```
CREATE TEMPORARY TABLE users (
    id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
);
```

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

此时将数据插入 `users`，插入到的是会话 A 中的本地临时表 `users`。

```
INSERT INTO users(id, name, city) VALUES(1001, 'Davis', 'LosAngeles');
```

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

```
SELECT * FROM users;
```

```
+------+-------+------------+
| id   | name  | city       |
+------+-------+------------+
| 1001 | Davis | LosAngeles |
+------+-------+------------+
1 row in set (0.00 sec)
```

在会话 B 中创建本地临时表 users，不会与普通表 users 冲突，也不会与会话 A 中的本地临时表 users 冲突。会话 B 内访问 users 时，访问的是会话 B 内创建的本地临时表 users 数据。

```
CREATE TEMPORARY TABLE users (
    id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
);
```

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

此时将数据插入 users，插入到的是会话 B 中的本地临时表 users。

```
INSERT INTO users(id, name, city) VALUES(1001, 'James', 'NewYork');
```

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

```
SELECT * FROM users;
```

```
+------+-------+---------+
| id   | name  | city    |
+------+-------+---------+
| 1001 | James | NewYork |
+------+-------+---------+
1 row in set (0.00 sec)
```

2.13.3.2 与 MySQL 临时表的兼容性

TiDB 本地临时表的以下特性与限制与 MySQL 一致：

- 创建、删除本地临时表时，不会自动提交当前事务

- 删除本地临时表所在的 schema 后，临时表不会被删除，仍然可以读写
- 创建本地临时表需要 `CREATE TEMPORARY TABLES` 权限，随后对该表的所有操作不需要权限
- 本地临时表不支持外键和分区表
- 不支持基于本地临时表创建视图
- `SHOW [FULL] TABLES` 不显示本地临时表

TiDB 本地临时表与 MySQL 临时表有以下方面不兼容：

- TiDB 本地临时表不支持 `ALTER TABLE`
- TiDB 本地临时表忽略 `ENGINE` 表选项，始终在 TiDB 内存中暂存临时表数据，并且有内存限制
- 当声明存储引擎为 `MEMORY` 时，TiDB 本地临时表没有 `MEMORY` 存储引擎的限制
- 当声明存储引擎为 `INNODB` 或 `MYISAM` 时，TiDB 本地临时表忽略 InnoDB 临时表特有的系统变量
- MySQL 不允许在同一条 SQL 中多次引用同一张临时表，而 TiDB 本地临时表没有该限制
- MySQL 中用于显示临时表的 `information_schema.INNODB_TEMP_TABLE_INFO` 表在 TiDB 中不存在。TiDB 暂无用于显示本地临时表的系统表。
- TiDB 没有内部临时表，MySQL 针对内部临时表的系统变量对 TiDB 不生效

2.13.4 全局临时表

全局临时表是 TiDB 的扩展功能，它有以下特性：

- 全局临时表的表定义会持久化，对所有会话可见
- 全局临时表的数据只对当前的事务内可见，事务结束后数据自动清空
- 全局临时表不能与普通表同名

用户可通过 `CREATE GLOBAL TEMPORARY TABLE` 语句创建全局临时表，语句末尾要加上 `ON COMMIT DELETE ROWS`。可通过 `DROP TABLE` 或 `DROP GLOBAL TEMPORARY TABLE` 语句删除全局临时表。

2.13.4.1 全局临时表使用示例

> 注意：
>
> - 使用 TiDB 中的临时表前，注意临时表与其他 TiDB 功能的兼容性限制。
> - 如果在 v5.3.0 或以上版本中创建了全局临时表，这些临时表在降级后会被当作普通表处理，导致数据错误。

在会话 A 中创建全局临时表 `users`：

```
CREATE GLOBAL TEMPORARY TABLE users (
    id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
) ON COMMIT DELETE ROWS;
```

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

写入 users 的数据对当前事务可见：

```
BEGIN;
```

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

```
INSERT INTO users(id, name, city) VALUES(1001, 'Davis', 'LosAngeles');
```

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

```
SELECT * FROM users;
```

```
+------+-------+------------+
| id   | name  | city       |
+------+-------+------------+
| 1001 | Davis | LosAngeles |
+------+-------+------------+
1 row in set (0.00 sec)
```

事务结束后数据自动被清空：

```
COMMIT;
```

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

```
SELECT * FROM users;
```

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

会话 A 创建了 users 后，会话 B 也可以读写该表：

```
SELECT * FROM users;
```

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

> 注意：
>
> 如果事务是自动提交的，插入的数据在 SQL 语句执行结束后会被自动清空，导致后续 SQL 执行查找不到结果。因此应该使用非自动提交的事务读写全局临时表。

2.13.5 限制临时表的内存占用

无论定义表时声明的 ENGINE 是哪种存储引擎，本地临时表和全局临时表的数据都只暂存在 TiDB 实例的内存中，不持久化。

为了避免内存溢出，用户可通过系统变量tidb_tmp_table_max_size 限制每张临时表的大小。当临时表大小超过限制后 TiDB 会报错。tidb_tmp_table_max_size 的默认值是 64MB。

例如，将每张临时表的大小限制为 256MB：

```
SET GLOBAL tidb_tmp_table_max_size=268435456;
```

2.13.6 与其他 TiDB 功能的兼容性限制

以下是本地临时表和全局临时表都不支持的功能：

- 不支持 AUTO_RANDOM 列
- 不支持 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 表选项
- 不支持分区表
- 不支持 SPLIT REGION 语句
- 不支持 ADMIN CHECK TABLE 和 ADMIN CHECKSUM TABLE 语句
- 不支持 FLASHBACK TABLE 和 RECOVER TABLE 语句
- 不支持以临时表为源表执行 CREATE TABLE LIKE 语句
- 不支持 Stale Read
- 不支持外键
- 不支持 SQL binding
- 不支持添加 TiFlash 副本
- 不支持在临时表上创建视图
- 不支持 Placement Rules
- 包含临时表的执行计划不会被 prepared plan cache 缓存

以下是只有本地临时表不支持的功能：

- 不支持通过系统变量 tidb_snapshot 读取历史数据

2.13.7 TiDB 数据迁移工具支持

本地临时表只对当前会话可见，因此本地临时表不会被 TiDB 数据迁移工具导出、备份、同步。

全局临时表的表定义全局可见，因此全局临时表的表定义会被 TiDB 数据迁移工具导出、备份、同步，但不导出数据。

> 注意：
>
> - TiCDC 必须使用 v5.3.0 及以上版本同步，否则下游集群的表定义错误

- BR 必须使用 v5.3.0 及以上版本备份，否则备份后的表定义错误
- 导入的集群、恢复后的集群、同步的下游集群需要支持全局临时表，否则报错

2.13.8 另请参阅

- CREATE TABLE
- CREATE TABLE LIKE
- DROP TABLE

2.14 缓存表

TiDB 在 v6.0.0 版本中引入了缓存表功能。该功能适用于频繁被访问且很少被修改的热点小表，即把整张表的数据加载到 TiDB 服务器的内存中，直接从内存中获取表数据，避免从 TiKV 获取表数据，从而提升读性能。

本文介绍了 TiDB 缓存表的使用场景、使用示例、与其他 TiDB 功能的兼容性限制。

2.14.1 使用场景

TiDB 缓存表功能适用于以下特点的表：

- 表的数据量不大
- 只读表，或者几乎很少修改
- 表的访问很频繁，期望有更好的读性能

当表的数据量不大，访问又特别频繁的情况下，数据会集中在 TiKV 一个 Region 上，形成热点，从而影响性能。因此，TiDB 缓存表的典型使用场景如下：

- 配置表，业务通过该表读取配置信息
- 金融场景中的存储汇率的表，该表不会实时更新，每天只更新一次
- 银行分行或者网点信息表，该表很少新增记录项

以配置表为例，当业务重启的瞬间，全部连接一起加载配置，会造成较高的数据库读延迟。如果使用了缓存表，则可以解决这样的问题。

2.14.2 使用示例

本节通过示例介绍缓存表的使用方法。

2.14.2.1 将普通表设为缓存表

假设已存在普通表 users:

```
CREATE TABLE users (
    id BIGINT,
    name VARCHAR(100),
    PRIMARY KEY(id)
);
```

通过 ALTER TABLE 语句，可以将这张表设置成缓存表：

```
ALTER TABLE users CACHE;
```

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

2.14.2.2 验证是否为缓存表

要验证一张表是否为缓存表，使用 SHOW CREATE TABLE 语句。如果为缓存表，返回结果中会带有 CACHED ON 属性：

```
SHOW CREATE TABLE users;
```

```
+-------+-----------------------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-----------------------------------------------------------------------------------------------------------------------------
    ↪
| users | CREATE TABLE `users` (
  `id` bigint(20) NOT NULL,
  `name` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /* CACHED ON */ |
+-------+-----------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

从缓存表读取数据后，TiDB 会将数据加载到内存中。你可使用 trace 语句查看 TiDB 是否已将数据加载到内存中。当缓存还未加载时，语句的返回结果会出现 regionRequest.SendReqCtx，表示 TiDB 从 TiKV 读取了数据。

```
TRACE SELECT * FROM users;
```

```
+-----------------------------------------------+-----------------+------------+
| operation                                     | startTS         | duration   |
+-----------------------------------------------+-----------------+------------+
```

```
| trace                                          | 17:47:39.969980 | 827.73µs   |
|   ├─session.ExecuteStmt                         | 17:47:39.969986 | 413.31µs   |
|   │ ├─executor.Compile                          | 17:47:39.969993 | 198.29µs   |
|   │ └─session.runStmt                           | 17:47:39.970221 | 157.252µs  |
|   │   └─TableReaderExecutor.Open                | 17:47:39.970294 | 47.068µs   |
|   │     └─distsql.Select                        | 17:47:39.970312 | 24.729µs   |
|   │       └─regionRequest.SendReqCtx            | 17:47:39.970454 | 189.601µs  |
|   ├─*executor.UnionScanExec.Next                | 17:47:39.970407 | 353.073µs  |
|   │ ├─*executor.TableReaderExecutor.Next        | 17:47:39.970411 | 301.106µs  |
|   │ └─*executor.TableReaderExecutor.Next        | 17:47:39.970746 | 6.57µs     |
|   └─*executor.UnionScanExec.Next                | 17:47:39.970772 | 17.589µs   |
|     └─*executor.TableReaderExecutor.Next        | 17:47:39.970776 | 6.59µs     |
+-------------------------------------------------+-----------------+------------+
12 rows in set (0.01 sec)
```

而再次执行 `trace`，返回结果中不再有 `regionRequest.SendReqCtx`，表示 TiDB 已经不再从 TiKV 读取数据，而是直接从内存中读取：

```
+---------------------------------------+-----------------+------------+
| operation                             | startTS         | duration   |
+---------------------------------------+-----------------+------------+
| trace                                 | 17:47:40.533888 | 453.547µs  |
|   ├─session.ExecuteStmt               | 17:47:40.533894 | 402.341µs  |
|   │ ├─executor.Compile                | 17:47:40.533903 | 205.54µs   |
|   │ └─session.runStmt                 | 17:47:40.534141 | 132.084µs  |
|   │   └─TableReaderExecutor.Open      | 17:47:40.534202 | 14.749µs   |
|   ├─*executor.UnionScanExec.Next      | 17:47:40.534306 | 3.21µs     |
|   └─*executor.UnionScanExec.Next      | 17:47:40.534316 | 1.219µs    |
+---------------------------------------+-----------------+------------+
7 rows in set (0.00 sec)
```

注意，读取缓存表会使用 `UnionScan` 算子，所以通过 `explain` 查看缓存表的执行计划时，可能会在结果中看到 `UnionScan`：

```
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                    | estRows | task      | access object | operator info
    ↪ |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
| UnionScan_5           | 1.00    | root      |               |
    ↪ |
| └─TableReader_7       | 1.00    | root      |               | data:TableFullScan_6
    ↪           |
```

```
|   └─TableFullScan_6     | 1.00    | cop[tikv] | table:users   | keep order:false, stats:pseudo
    ↪  |
+--
    ↪ ----------------------+---------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

2.14.2.3 往缓存表写入数据

缓存表支持写入数据。例如，往 users 表中插入一条记录：

```
INSERT INTO users(id, name) VALUES(1001, 'Davis');
```

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

```
SELECT * FROM users;
```

```
+------+-------+
| id   | name  |
+------+-------+
| 1001 | Davis |
+------+-------+
1 row in set (0.00 sec)
```

> 注意：
>
> 往缓存表写入数据时，有可能出现秒级别的写入延迟。延迟的时长由全局环境变量tidb_table_cache_lease 控制。你可根据实际业务能否承受此限制带来的延迟，决定是否适合使用缓存表功能。例如，对于完全只读的场景，可以将 tidb_table_cache_lease 调大：
>
> ```
> set @@global.tidb_table_cache_lease = 10;
> ```
>
> 缓存表的写入延时高是受到实现的限制。存在多个 TiDB 实例时，一个 TiDB 实例并不知道其它的 TiDB 实例是否缓存了数据，如果该实例直接修改了表数据，而其它 TiDB 实例依然读取旧的缓存数据，就会读到错误的结果。为了保证数据正确性，缓存表的实现使用了一套基于 lease 的复杂机制：读操作在缓存数据同时，还会对于缓存设置一个有效期，也就是 lease。在 lease 过期之前，无法对数据执行修改操作。因为修改操作必须等待 lease 过期，所以会出现写入延迟。

2.14.2.4 将缓存表恢复为普通表

> **注意：**
>
> 对缓存表执行 DDL 语句会失败。若要对缓存表执行 DDL 语句，需要先去掉缓存属性，将缓存表设回普通表后，才能对其执行 DDL 语句。

```
TRUNCATE TABLE users;
```

```
ERROR 8242 (HY000): 'Truncate Table' is unsupported on cache tables.
```

```
mysql> ALTER TABLE users ADD INDEX k_id(id);
```

```
ERROR 8242 (HY000): 'Alter Table' is unsupported on cache tables.
```

使用 `ALTER TABLE t NOCACHE` 语句可以将缓存表恢复成普通表：

```
ALTER TABLE users NOCACHE
```

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

2.14.3 缓存表大小限制

由于 TiDB 将整张缓存表的数据加载到 TiDB 进程的内存中，并且执行修改操作后缓存会失效，需要重新加载，所以 TiDB 缓存表只适用于表比较小的场景。

目前 TiDB 对于每张缓存表的大小限制为 64 MB。如果表的数据超过了 64 MB，执行 `ALTER TABLE t CACHE` 会失败。

2.14.4 与其他 TiDB 功能的兼容性限制

以下是缓存表不支持的功能：

- 不支持对分区表执行 `ALTER TABLE t CACHE` 操作
- 不支持对临时表执行 `ALTER TABLE t CACHE` 操作
- 不支持对视图执行 `ALTER TABLE t CACHE` 操作
- 不支持 Stale Read 功能
- 不支持对缓存表直接做 DDL 操作，需要先通过 `ALTER TABLE t NOCACHE` 将缓存表改回普通表后再进行 DDL 操作。

以下是缓存表无法使用缓存的场景：

- 设置系统变量 `tidb_snapshot` 读取历史数据
- 执行修改操作期间，已有缓存会失效，直到数据被再次加载

2.14.5 TiDB 数据迁移工具兼容性

缓存表并不是标准的 MySQL 功能，而是 TiDB 扩展。只有 TiDB 能识别 `ALTER TABLE ... CACHE` 语句。所有的 TiDB 数据迁移工具均不支持缓存表功能，包括 Backup & Restore (BR)、TiCDC、Dumpling 等组件，它们会将缓存表当作普通表处理。

这意味着，备份恢复一张缓存表时，它会变成一张普通表。如果下游集群是另一套 TiDB 集群并且你希望继续使用缓存表功能，可以对下游集群中的表执行 `ALTER TABLE ... CACHE` 手动开启缓存表功能。

2.14.6 另请参阅

- ALTER TABLE
- System Variables

2.15 字符集和排序

2.15.1 字符集和排序规则

本文介绍了 TiDB 中支持的字符集和排序规则。

2.15.1.1 字符集和排序规则的概念

字符集 (character set) 是符号与编码的集合。TiDB 中的默认字符集是 utf8mb4，与 MySQL 8.0 及更高版本中的默认字符集匹配。

排序规则 (collation) 是在字符集中比较字符以及字符排序顺序的规则。例如，在二进制排序规则中，比较 A 和 a 的结果是不一样的：

```
SET NAMES utf8mb4 COLLATE utf8mb4_bin;
SELECT 'A' = 'a';
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
SELECT 'A' = 'a';
```

```
mysql> SELECT 'A' = 'a';
+-----------+
| 'A' = 'a' |
+-----------+
|         0 |
+-----------+
1 row in set (0.00 sec)

mysql> SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT 'A' = 'a';
+-----------+
| 'A' = 'a' |
```

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

TiDB 默认使用二进制排序规则。这一点与 MySQL 不同，MySQL 默认使用不区分大小写的排序规则。

2.15.1.2 支持的字符集和排序规则

目前 TiDB 支持以下字符集：

```
SHOW CHARACTER SET;
```

```
+---------+-------------------------------------+--------------------+--------+
| Charset | Description                         | Default collation | Maxlen |
+---------+-------------------------------------+--------------------+--------+
| ascii   | US ASCII                            | ascii_bin          |      1 |
| binary  | binary                              | binary             |      1 |
| gbk     | Chinese Internal Code Specification | gbk_bin            |      2 |
| latin1  | Latin1                              | latin1_bin         |      1 |
| utf8    | UTF-8 Unicode                       | utf8_bin           |      3 |
| utf8mb4 | UTF-8 Unicode                       | utf8mb4_bin        |      4 |
+---------+-------------------------------------+--------------------+--------+
6 rows in set (0.00 sec)
```

TiDB 支持以下排序规则：

```
mysql> show collation;
+-------------+---------+------+---------+----------+---------+
| Collation   | Charset | Id   | Default | Compiled | Sortlen |
+-------------+---------+------+---------+----------+---------+
| utf8mb4_bin | utf8mb4 |   46 | Yes     | Yes      |       1 |
| latin1_bin  | latin1  |   47 | Yes     | Yes      |       1 |
| binary      | binary  |   63 | Yes     | Yes      |       1 |
| ascii_bin   | ascii   |   65 | Yes     | Yes      |       1 |
| utf8_bin    | utf8    |   83 | Yes     | Yes      |       1 |
| gbk_bin     | gbk     |   87 | Yes     | Yes      |       1 |
+-------------+---------+------+---------+----------+---------+
6 rows in set (0.00 sec)
```

> **警告：**
>
> TiDB 会错误地将 `latin1` 视为 `utf8` 的子集。当用户存储不同于 `latin1` 和 `utf8` 编码的字符时，可能会导致意外情况出现。因此强烈建议使用 `utf8mb4` 字符集。详情参阅 TiDB #18955。

> **注意：**
>
> TiDB 中的默认排序规则（后缀为 _bin 的二进制排序规则）与 MySQL 中的默认排序规则不同，后者通常是一般排序规则，后缀为 _general_ci。当用户指定了显式字符集，但依赖于待选的隐式默认排序规则时，这个差异可能导致兼容性问题。

利用以下的语句可以查看字符集对应的排序规则（以下是新的排序规则框架）下的结果：

```
SHOW COLLATION WHERE Charset = 'utf8mb4';
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci | utf8mb4 |  224 |         | Yes      |       1 |
+--------------------+---------+------+---------+----------+---------+
3 rows in set (0.00 sec)
```

TiDB 对 GBK 字符集的支持详情见GBK。

2.15.1.3 TiDB 中的 utf8 和 utf8mb4

MySQL 限制字符集 utf8 为最多 3 个字节。这足以存储在基本多语言平面 (BMP) 中的字符，但不足以存储表情符号 (emoji) 等字符。因此，建议改用字符集utf8mb4。

默认情况下，TiDB 同样限制字符集 utf8 为最多 3 个字节，以确保 TiDB 中创建的数据可以在 MySQL 中顺利恢复。你可以禁用此功能，方法是在 TiDB 配置文件中将 check-mb4-value-in-utf8 的值更改为 FALSE。

以下示例演示了在表中插入 4 字节的表情符号字符（emoji 字符）时的默认行为。utf8 字符集下 INSERT 语句不能执行，utf8mb4 字符集下可以执行 INSERT 语句：

```
mysql> CREATE TABLE utf8_test (
    ->  c char(1) NOT NULL
    -> ) CHARACTER SET utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE utf8m4_test (
    ->  c char(1) NOT NULL
    -> ) CHARACTER SET utf8mb4;
Query OK, 0 rows affected (0.09 sec)

mysql> INSERT INTO utf8_test VALUES ('');
ERROR 1366 (HY000): incorrect utf8 value f09f9889( ) for column c
mysql> INSERT INTO utf8m4_test VALUES ('');
```

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

mysql> SELECT char_length(c), length(c), c FROM utf8_test;
Empty set (0.01 sec)

mysql> SELECT char_length(c), length(c), c FROM utf8m4_test;
+----------------+-----------+------+
| char_length(c) | length(c) | c    |
+----------------+-----------+------+
|              1 |         4 | 𐐷    |
+----------------+-----------+------+
1 row in set (0.00 sec)
```

2.15.1.4 不同范围的字符集和排序规则

字符集和排序规则可以在设置在不同的层次。

2.15.1.4.1 数据库的字符集和排序规则

每个数据库都有相应的字符集和排序规则。数据库的字符集和排序规则可以通过以下语句来设置：

```
CREATE DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]

ALTER DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

在这里 DATABASE 可以跟 SCHEMA 互换使用。

不同的数据库之间可以使用不一样的字符集和排序规则。

通过系统变量 character_set_database 和 collation_database 可以查看到当前数据库的字符集以及排序规则：

```
CREATE SCHEMA test1 CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test1;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| utf8mb4                  | utf8mb4_general_ci   |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

```
CREATE SCHEMA test2 CHARACTER SET latin1 COLLATE latin1_bin;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test2;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| latin1                   | latin1_bin           |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

在 INFORMATION_SCHEMA 中也可以查看到这两个值：

```
SELECT DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME
FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'db_name';
```

2.15.1.4.2 表的字符集和排序规则

表的字符集和排序规则可以通过以下语句来设置：

```
CREATE TABLE tbl_name (column_list)
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]]

ALTER TABLE tbl_name
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]
```

例如：

```
CREATE TABLE t1(a int) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.08 sec)
```

如果表的字符集和排序规则没有设置，那么数据库的字符集和排序规则就作为其默认值。

2.15.1.4.3 列的字符集和排序规则

列的字符集和排序规则的语法如下：

```
col_name {CHAR | VARCHAR | TEXT} (col_length)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]

col_name {ENUM | SET} (val_list)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]
```

如果列的字符集和排序规则没有设置，那么表的字符集和排序规则就作为其默认值。

2.15.1.4.4 字符串的字符集和排序规则

每一个字符串都对应一个字符集和一个排序规则，在使用字符串时指此选项可选，如下：

```
[_charset_name]'string' [COLLATE collation_name]
```

示例如下：

```
SELECT 'string';
SELECT _utf8mb4'string';
SELECT _utf8mb4'string' COLLATE utf8mb4_general_ci;
```

规则如下：

- 规则 1：如果指定了 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 charset_name 字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用 character_set_connection
  ↪ 和 collation_connection 系统变量给出的字符集和排序规则。

2.15.1.4.5 客户端连接的字符集和排序规则

- 服务器的字符集和排序规则可以通过系统变量 character_set_server 和 collation_server 获取。
- 数据库的字符集和排序规则可以通过环境变量 character_set_database 和 collation_database 获取。

对于每一个客户端的连接，也有相应的变量表示字符集和排序规则：character_set_connection 和 collation_connection。

character_set_client 代表客户端的字符集。在返回结果前，服务端会把结果根据 character_set_results 转换成对应的字符集，包括结果的元信息等。

可以用以下的语句来影响这些跟客户端相关的字符集变量：

- SET NAMES 'charset_name' [COLLATE 'collation_name']

  SET NAMES 用来设定客户端会在之后的请求中使用的字符集。SET NAMES utf8mb4 表示客户端会在接下来的请求中，都使用 utf8mb4 字符集。服务端也会在之后返回结果的时候使用 utf8mb4 字符集。SET NAMES 'charset_name' 语句其实等于下面语句的组合：

```
SET character_set_client = charset_name;
SET character_set_results = charset_name;
SET character_set_connection = charset_name;
```

  COLLATE 是可选的，如果没有提供，将会用 charset_name 对应的默认排序规则设置 collation_connection 。

- SET CHARACTER SET 'charset_name'

  跟 SET NAMES 类似，等价于下面语句的组合：

```
SET character_set_client = charset_name;
SET character_set_results = charset_name;
SET charset_connection = @@charset_database;
SET collation_connection = @@collation_database;
```

2.15.1.5 服务器、数据库、表、列、字符串的字符集和排序规则的优先级

优先级从高到低排列顺序为：

字符串 > 列 > 表 > 数据库 > 服务器

2.15.1.6 字符集和排序规则的通用选择规则

- 规则 1：如果指定了 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 charset_name 字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用更高优先级的字符集和排序规则。

2.15.1.7 字符合法性检查

当指定的字符集为 utf8 或 utf8mb4 时，TiDB 仅支持合法的 utf8 字符。对于不合法的字符，会报错：incorrect utf8 value。该字符合法性检查与 MySQL 8.0 兼容，与 MySQL 5.7 及以下版本不兼容。

如果不希望报错，可以通过 set @@tidb_skip_utf8_check=1; 跳过字符检查。

2.15.1.8 排序规则支持

排序规则的语法支持和语义支持受到配置项new_collations_enabled_on_first_bootstrap 的影响。这里语法支持和语义支持有所区别。语法支持是指 TiDB 能够解析和设置排序规则；而语义支持是指 TiDB 能够在比较字符串时正确地使用排序规则。

在 4.0 版本之前，TiDB 只提供了旧的排序规则框架，能够在语法上支持的绝大部分 MySQL 排序规则，但语义上所有的排序规则都当成二进制排序规则。

4.0 版本中，TiDB 增加了新的排序规则框架用于在语义上支持不同的排序规则，保证字符串比较时严格遵循对应的排序规则，详情请见下文。

2.15.1.8.1 旧框架下的排序规则支持

在 4.0 版本之前，TiDB 中可以指定大部分 MySQL 中的排序规则，并把这些排序规则按照默认排序规则处理，即以编码字节序为字符定序。和 MySQL 不同的是，TiDB 在比较字符前按照排序规则的 PADDING 属性将字符末尾的空格删除，因此会造成以下的行为区别：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
Query OK, 0 rows affected
INSERT INTO t VALUES ('A');
Query OK, 1 row affected
INSERT INTO t VALUES ('a');
Query OK, 1 row affected # TiDB 会执行成功，而在 MySQL 中，则由于 utf8mb4_general_ci 大小写不敏感
    ↪ ，报错 Duplicate entry 'a'。
INSERT INTO t VALUES ('a ');
Query OK, 1 row affected # TiDB 会执行成功，而在 MySQL 中，则由于补齐空格比较，报错 Duplicate
    ↪ entry 'a '。
```

2.15.1.8.2 新框架下的排序规则支持

TiDB 4.0 新增了完整的排序规则支持框架，从语义上支持了排序规则，并新增了配置开关 new_collations_enabled_on_first_boot
↪ ，在集群初次初始化时决定是否启用新排序规则框架。如需启用新排序规则框架，可将 new_collations_enabled_on_first_bootstrap 的值设为 true，详情参见new_collations_enabled_on_first_bootstrap
↪ 。要在该配置开关打开之后初始化集群，可以通过 mysql.tidb 表中的 new_collation_enabled 变量确认是否启用了新排序规则框架：

```
SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME='new_collation_enabled';
```

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

在新的排序规则框架下，TiDB 能够支持 utf8_general_ci、utf8mb4_general_ci、utf8_unicode_ci、utf8mb4_unicode_ci、gbk_chinese_ci 和 gbk_bin 这几种排序规则，与 MySQL 兼容。

使用 utf8_general_ci、utf8mb4_general_ci、utf8_unicode_ci 、utf8mb4_unicode_ci 和 gbk_chinese_ci 中任一种时，字符串之间的比较是大小写不敏感 (case-insensitive) 和口音不敏感 (accent-insensitive) 的。同时，TiDB 还修正了排序规则的 PADDING 行为：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
Query OK, 0 rows affected (0.00 sec)
INSERT INTO t VALUES ('A');
Query OK, 1 row affected (0.00 sec)
INSERT INTO t VALUES ('a');
ERROR 1062 (23000): Duplicate entry 'a' for key 'PRIMARY' # TiDB 兼容了 MySQL 的 case insensitive
    ↪  collation。
INSERT INTO t VALUES ('a ');
ERROR 1062 (23000): Duplicate entry 'a ' for key 'PRIMARY' # TiDB 修正了 `PADDING` 行为，与 MySQL
    ↪  兼容。
```

> 注意：
>
> TiDB 中 padding 的实现方式与 MySQL 的不同。在 MySQL 中，padding 是通过补齐空格实现的。而在 TiDB 中 padding 是通过裁剪掉末尾的空格来实现的。两种做法在绝大多数情况下是一致的，唯一的例外是字符串尾部包含小于空格 (0x20) 的字符时，例如 `'a' < 'a\t'` 在 TiDB 中的结果为 1，而在 MySQL 中，其等价于 `'a ' < 'a\t'`，结果为 0。

2.15.1.9 表达式中排序规则的 Coercibility 值

如果一个表达式涉及多个不同排序规则的子表达式时，需要对计算时用的排序规则进行推断，规则如下：

- 显式 COLLATE 子句的 coercibility 值为 0。
- 如果两个字符串的排序规则不兼容，这两个字符串 concat 结果的 coercibility 值为 1。
- 列或者 CAST()、CONVERT() 和 BINARY() 的排序规则的 coercibility 值为 2。
- 系统常量（USER() 或者 VERSION() 返回的字符串）的 coercibility 值为 3。
- 常量的 coercibility 值为 4。
- 数字或者中间变量的 coercibility 值为 5。
- NULL 或者由 NULL 派生出的表达式的 coercibility 值为 6。

在推断排序规则时，TiDB 优先使用 coercibility 值较低的表达式的排序规则。如果 coercibility 值相同，则按以下优先级确定排序规则：

binary > utf8mb4_bin > (utf8mb4_general_ci = utf8mb4_unicode_ci) > utf8_bin > (utf8_general_ci = utf8_unicode_ci) > latin1_bin > ascii_bin

以下情况 TiDB 无法推断排序规则并报错：

- 如果两个子表达式的排序规则不相同，而且表达式的 coercibility 值都为 0。
- 如果两个子表达式的排序规则不兼容，而且表达式的返回类型为 String 类。

2.15.1.10 COLLATE 子句

TiDB 支持使用 COLLATE 子句来指定一个表达式的排序规则，该表达式的 coercibility 值为 0，具有最高的优先级。示例如下：

```
SELECT 'a' = _utf8mb4 'A' collate utf8mb4_general_ci;
```

```
+-----------------------------------------------+
| 'a' = _utf8mb4 'A' collate utf8mb4_general_ci |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

更多细节，参考 Connection Character Sets and Collations。

2.15.2 GBK

TiDB 从 v5.4.0 开始支持 GBK 字符集。本文档介绍 TiDB 对 GBK 字符集的支持和兼容情况。

```
SHOW CHARACTER SET WHERE CHARSET = 'gbk';
+---------+-----------------------------------+--------------------+--------+
| Charset | Description                       | Default collation | Maxlen |
+---------+-----------------------------------+--------------------+--------+
| gbk     | Chinese Internal Code Specification | gbk_bin           |      2 |
+---------+-----------------------------------+--------------------+--------+
1 row in set (0.00 sec)

SHOW COLLATION WHERE CHARSET = 'gbk';
+----------------+---------+------+---------+----------+---------+
| Collation      | Charset | Id   | Default | Compiled | Sortlen |
+----------------+---------+------+---------+----------+---------+
| gbk_bin        | gbk     |   87 |         | Yes      |       1 |
+----------------+---------+------+---------+----------+---------+
1 rows in set (0.00 sec)
```

2.15.2.1 与 MySQL 的兼容性

本节介绍 TiDB 中 GBK 字符集与 MySQL 的兼容情况。

2.15.2.1.1 排序规则兼容性

MySQL 的字符集默认排序规则是 gbk_chinese_ci。与 MySQL 不同，TiDB GBK 字符集的默认排序规则为 gbk_bin。另外，TiDB 支持的 gbk_bin 与 MySQL 支持的 gbk_bin 排序规则也不一致，TiDB 是将 GBK 转换成 UTF8MB4 然后做二进制排序。

如果要使 TiDB 兼容 MySQL 的 GBK 字符集排序规则，你需要在初次初始化 TiDB 集群时设置 TiDB 配置项new_collations_enabled_on_first_bootstrap 为 true 来开启新的排序规则框架。

开启新的排序规则框架后，如果查看 GBK 字符集对应的排序规则，你可以看到 TiDB GBK 默认排序规则已经切换为 gbk_chinese_ci。

```
SHOW CHARACTER SET WHERE CHARSET = 'gbk';
+---------+-----------------------------------+--------------------+--------+
| Charset | Description                       | Default collation | Maxlen |
+---------+-----------------------------------+--------------------+--------+
| gbk     | Chinese Internal Code Specification | gbk_chinese_ci    |      2 |
+---------+-----------------------------------+--------------------+--------+
1 row in set (0.00 sec)

SHOW COLLATION WHERE CHARSET = 'gbk';
+----------------+---------+------+---------+----------+---------+
| Collation      | Charset | Id   | Default | Compiled | Sortlen |
+----------------+---------+------+---------+----------+---------+
| gbk_bin        | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci | gbk     |   28 | Yes     | Yes      |       1 |
+----------------+---------+------+---------+----------+---------+
2 rows in set (0.00 sec)
```

2.15.2.1.2 非法字符兼容性

- 在系统变量character_set_client 和character_set_connection 不同时设置为 gbk 的情况下，TiDB 处理非法字符的方式与 MySQL 一致。
- 在 character_set_client 和 character_set_connection 同时为 gbk 的情况下，TiDB 处理非法字符的方式与 MySQL 有所区别。
  - MySQL 处理非法 GBK 字符集时，对读和写操作的处理方式不同。
  - TiDB 处理非法 GBK 字符集时，对读和写操作的处理方式相同。TiDB 在严格模式下读写非法 GBK 字符都会报错，在非严格模式下，读写非法 GBK 字符都会用 ? 替换。

例如，当 SET NAMES gbk 时，如果分别在 MySQL 和 TiDB 上通过 CREATE TABLE gbk_table(a VARCHAR(32) CHARACTER SET gbk) 语句建表，然后按照下表中的 SQL 语句进行操作，就能看到具体的区别。

数据库	如果设置的 SQL 模式包含 STRICT_ALL_TABLES 或 STRICT_TRANS_TABLES	如果设置的 SQL 模式不包含 STRICT_ALL_TABLES 和 STRICT_TRANS_TABLES
MySQL	SELECT HEX('一a'); e4b88061INSERT INTO gbk_table values('一a'); Incorrect Error	SELECT HEX('一a'); e4b88061INSERT INTO gbk_table VALUES('一a'); SELECT HEX(a)FROM gbk_table; e4b8

数据库	如果设置的 SQL 模式包含 STRICT_ALL_TABLES 或 STRICT_TRANS_TABLES	如果设置的 SQL 模式不包含 STRICT_ALL_TABLES 和 STRICT_TRANS_TABLES
TiDB	SELECT HEX('一a'); Incorrect ErrorINSERT ↪ INTO gbk_table VALUES('一a'); Incorrect Error	SELECT HEX('一a'); e4b83fINSERT INTO gbk_table VALUES('一a'); ↪ SELECT HEX(a)FROM gbk_table; e4b83f

说明：该表中 SELECT HEX('一a'); 在 utf8mb4 字节集下的结果为 e4b88061。

2.15.2.1.3 其它

- 目前 TiDB 不支持通过 ALTER TABLE 语句将其它字符集类型改成 gbk 或者从 gbk 转成其它字符集类型。
- TiDB 不支持使用 _gbk，比如：

```
sql CREATE TABLE t(a CHAR(10)CHARSET BINARY); Query OK, 0 rows affected (0.00 sec)INSERT INTO t
↪ VALUES (_gbk'啊'); ERROR 1115 (42000): Unsupported character introducer: 'gbk'
```

- 对于 ENUM 和 SET 类型中的二进制字符，TiDB 目前都会将其作为 utf8mb4 字符集处理。

2.15.2.2 组件兼容性

- TiFlash 目前不支持 GBK 字符集。
- TiDB Data Migration (DM) 在 v5.4.0 之前不支持将 charset=GBK 的表迁移到 TiDB。
- TiDB Lightning 在 v5.4.0 之前不支持导入 charset=GBK 的表。
- TiCDC 在 v6.1.0 之前不支持同步 charset=GBK 的表。另外，任何版本的 TiCDC 都不支持同步 charset=GBK 的表到 6.1.0 之前的 TiDB 集群。
- TiDB Backup & Restore（BR）在 v5.4.0 之前不支持恢复 charset=GBK 的表。另外，任何版本的 BR 都不支持恢复 charset=GBK 的表到 5.4.0 之前的 TiDB 集群。

2.16 Placement Rules in SQL

Placement Rules in SQL 特性用于通过 SQL 接口配置数据在 TiKV 集群中的放置位置。通过该功能，用户可以将表和分区指定部署至不同的地域、机房、机柜、主机。适用场景包括低成本优化数据高可用策略、保证本地的数据副本可用于本地 Stale Read 读取、遵守数据本地要求等。

> **注意：**
>
> Placement Rules in SQL 底层的实现依赖 PD 提供的放置规则 (placement rules) 功能，参考Placement Rules 使用文档。在 Placement Rules in SQL 语境下，放置规则既可以代指绑定对象的放置策略 (placement policy)，也可以代指 TiDB 发给 PD 的放置规则。

该功能可以实现以下业务场景：

- 合并多个不同业务的数据库，大幅减少数据库常规运维管理的成本
- 增加重要数据的副本数，提高业务可用性和数据可靠性
- 将最新数据存入 SSD，历史数据存入 HDD，降低归档数据存储成本
- 把热点数据的 leader 放到高性能的 TiKV 实例上
- 将冷数据分离到不同的存储中以提高可用性
- 支持物理隔离不同用户之间的计算资源，满足实例内部不同用户的隔离需求，以及不同混合负载 CPU、I/O、内存等资源隔离的需求

2.16.1 指定放置规则

指定放置规则，首先需要通过CREATE PLACEMENT POLICY 语句创建放置策略 (placement policy)。

```
CREATE PLACEMENT POLICY myplacementpolicy PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1
    ↪ ";
```

然后可以使用 CREATE TABLE 或者 ALTER TABLE 将规则绑定至表或分区表，这样就在表或分区上指定了放置规则：

```
CREATE TABLE t1 (a INT) PLACEMENT POLICY=myplacementpolicy;
CREATE TABLE t2 (a INT);
ALTER TABLE t2 PLACEMENT POLICY=myplacementpolicy;
```

PLACEMENT POLICY 为全局作用域，不与任何数据库表结构相关联。因此，通过 CREATE TABLE 指定放置规则时，无需任何额外的权限。

要修改放置策略，你可以使用ALTER PLACEMENT POLICY 语句。修改将传播到所有绑定此放置策略的对象。

```
ALTER PLACEMENT POLICY myplacementpolicy FOLLOWERS=5;
```

要删除没有绑定任何分区或表的放置策略，你可以使用DROP PLACEMENT POLICY：

```
DROP PLACEMENT POLICY myplacementpolicy;
```

2.16.2 查看放置规则

如果一张表绑定了放置规则，你可以用SHOW CREATE TABLE 来查看。还可以用SHOW CREATE PLACEMENT POLICY 来查看已经创建的放置策略。

```
tidb> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /*T![placement] PLACEMENT POLICY=`
    ↪ myplacementpolicy` */
```

```
1 row in set (0.00 sec)
tidb> SHOW CREATE PLACEMENT POLICY myplacementpolicy\G
*************************** 1. row ***************************
       Policy: myplacementpolicy
Create Policy: CREATE PLACEMENT POLICY myplacementpolicy PRIMARY_REGION="us-east-1" REGIONS="us-
    ↪ east-1,us-west-1"
1 row in set (0.00 sec)
```

你也可以用`INFORMATION_SCHEMA.PLACEMENT_POLICIES`系统表查看所有放置策略的定义。

```
tidb> select * from information_schema.placement_policies\G
***************************[ 1. row ]***************************
POLICY_ID            | 1
CATALOG_NAME         | def
POLICY_NAME          | p1
PRIMARY_REGION       | us-east-1
REGIONS              | us-east-1,us-west-1
CONSTRAINTS          |
LEADER_CONSTRAINTS   |
FOLLOWER_CONSTRAINTS |
LEARNER_CONSTRAINTS  |
SCHEDULE             |
FOLLOWERS            | 4
LEARNERS             | 0
1 row in set
```

`information_schema.tables`表和`information_schema.partitions`表也有一列`tidb_placement_policy_name`，用于展示所有绑定了放置规则的对象：

```
SELECT * FROM information_schema.tables WHERE tidb_placement_policy_name IS NOT NULL;
SELECT * FROM information_schema.partitions WHERE tidb_placement_policy_name IS NOT NULL;
```

所有绑定规则的对象都是异步调度的。可以用`SHOW PLACEMENT`来查看放置规则的调度进度。

2.16.3 放置选项参考

注意：

- 放置选项依赖于正确地指定在每个 TiKV 节点配置中的标签 (label)。例如，`PRIMARY_REGION` 选项依赖 TiKV 中的 `region` 标签。若要查看当前 TiKV 集群中所有可用的标签，可执行`SHOW PLACEMENT LABELS`语句。

```
mysql> show placement labels;
+--------+----------------+
```

```
| Key    | Values         |
+--------+----------------+
| disk   | ["ssd"]        |
| region | ["us-east-1"]  |
| zone   | ["us-east-1a"] |
+--------+----------------+
3 rows in set (0.00 sec)
```

- 使用 CREATE PLACEMENT POLICY 创建放置规则时，TiDB 不会检查标签是否存在，而是在绑定表的时候进行检查。

选项名	描述
PRIMARY_REGION	Raft leader 被放置在有 region 标签的节点上，且这些 region 标签匹配本选项的值。
REGIONS	Raft followers 被放置在有 region 标签的节点上，且这些 region 标签匹配本选项的值。
SCHEDULE	用于调度 follower 放置位置的策略。可选值为 EVEN（默认值）或 MAJORITY_IN_PRIMARY。
FOLLOWERS	Follower 的数量。例如 FOLLOWERS=2 表示数据有 3 个副本（2 个 follower 和 1 个 leader）。

除以上配置选项外，你还可以使用高级配置，详细介绍见高级放置选项。

选项名	描述
CONSTRAINTS	适用于所有角色 (role) 的约束列表。例如，CONSTRAINTS="[+disk=ssd]"。
LEADER_CONSTRAINTS	仅适用于 leader 的约束列表。
FOLLOWER_CONSTRAINTS	仅适用于 follower 的约束列表。
LEARNER_CONSTRAINTS	仅适用于 learner 的约束列表。
LEARNERS	指定 learner 的数量。

2.16.4 示例

2.16.4.1 增加副本数

max-replicas 配置项的默认值为 3。如要为特定的表调大该值，可使用配置策略，示例如下：

```
CREATE PLACEMENT POLICY fivereplicas FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=fivereplicas;
```

注意，PD 配置中包含了 leader 数和 follower 数。因此，5 个副本为 4 个 follower + 1 个 leader。

对于以上示例，你还可使用 PRIMARY_REGION 和 REGIONS 选项来描述 follower 的放置规则：

```
CREATE PLACEMENT POLICY eastandwest PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-east-2,us-
    ↪ west-1" SCHEDULE="MAJORITY_IN_PRIMARY" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=eastandwest;
```

SCHEDULE 选项指示 TiDB 如何平衡 follower。该选项默认的 EVEN 调度规则确保 follower 在所有区域内分布平衡。

如要保证在主区域内 (us-east-1) 放置足够多的 follower 副本，你可以使用 MAJORITY_IN_PRIMARY 调度规则来使该区域的 follower 达到指定数量。该调度牺牲一些可用性来换取更低的事务延迟。如果主区域宕机，MAJORITY_IN_PRIMARY 无法提供自动故障转移。

2.16.4.2 为分区表指定放置规则

除了给表绑定放置策略之外，你还可以给表分区绑定放置策略。示例如下：

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=5;
CREATE PLACEMENT POLICY europe PRIMARY_REGION="eu-central-1" REGIONS="eu-central-1,eu-west-1";
CREATE PLACEMENT POLICY northamerica PRIMARY_REGION="us-east-1" REGIONS="us-east-1";

SET tidb_enable_list_partition = 1;
CREATE TABLE t1 (
  country VARCHAR(10) NOT NULL,
  userdata VARCHAR(100) NOT NULL
) PLACEMENT POLICY=p1 PARTITION BY LIST COLUMNS (country) (
  PARTITION pEurope VALUES IN ('DE', 'FR', 'GB') PLACEMENT POLICY=europe,
  PARTITION pNorthAmerica VALUES IN ('US', 'CA', 'MX') PLACEMENT POLICY=northamerica,
  PARTITION pAsia VALUES IN ('CN', 'KR', 'JP')
);
```

如果分区没有绑定任何放置策略，分区将尝试继承表上可能存在的策略。比如，pEurope 分区将会应用 europe 策略，而 pAsia 分区将会应用表 t1 的放置策略 p1。如果 t1 没有绑定任何策略，pAsia 就不会应用任何策略。

给分区绑定放置策略后，你可以更改指定分区的放置策略。示例如下：

```
ALTER TABLE t1 PARTITION pEurope PLACEMENT POLICY=p1;
```

2.16.4.3 为数据库配置默认的放置规则

你可以为某个数据库指定默认的放置策略，类似于为数据库设置默认字符集或排序规则。如果没有指定其他选项，就会使用数据库上指定的配置。示例如下：

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-east-2";  --
    ↪ 创建放置策略

CREATE PLACEMENT POLICY p2 FOLLOWERS=4;

CREATE PLACEMENT POLICY p3 FOLLOWERS=2;

CREATE TABLE t1 (a INT);  -- 创建表 t1，且未指定放置规则。

ALTER DATABASE test PLACEMENT POLICY=p2;  -- 更改默认的放置规则，但更改不影响已有的表 t1。

CREATE TABLE t2 (a INT);  -- 创建表 t2，默认的放置策略 p2 在 t2 上生效。
```

```
CREATE TABLE t3 (a INT) PLACEMENT POLICY=p1;  -- 创建表 t3。因为语句中已经指定了其他放置规则，
    ↪ 默认的 p2 策略在 t3 上不生效。

ALTER DATABASE test PLACEMENT POLICY=p3;  -- 再次更改默认的放置规则，此更改不影响已有的表。

CREATE TABLE t4 (a INT);  -- 创建表 t4，默认的放置策略 p3 生效。

ALTER PLACEMENT POLICY p3 FOLLOWERS=3; -- 绑定策略 p3 的表，也就是 t4，会采用 FOLLOWERS=3。
```

注意分区与表之间的继承和这里的继承不同。改变表的放置策略，也会让分区应用新的策略。但是只有建表时没有指定放置策略的时候，表才会从数据库继承放置策略，且之后再改变数据库也不影响已经继承的表。

2.16.4.4 高级放置选项

放置选项 PRIMARY_REGION、REGIONS 和 SCHEDULE 可满足数据放置的基本需求，但会缺乏一些灵活性。在较复杂的场景下，若需要更灵活地放置数据，可以使用高级放置选项 CONSTRAINTS 和 FOLLOWER_CONSTRAINTS。PRIMARY_REGION、REGIONS 和 SCHEDULE 选项不可与 CONSTRAINTS 选项同时指定，否则会报错。

以下示例设置了一个约束，要求数据必须位于某个 TiKV 节点，且该节点的 disk 标签必须匹配特定的值：

```
CREATE PLACEMENT POLICY storeonfastssd CONSTRAINTS="[+disk=ssd]";
CREATE PLACEMENT POLICY storeonhdd CONSTRAINTS="[+disk=hdd]";
CREATE PLACEMENT POLICY companystandardpolicy CONSTRAINTS="";

CREATE TABLE t1 (id INT, name VARCHAR(50), purchased DATE)
PLACEMENT POLICY=companystandardpolicy
PARTITION BY RANGE( YEAR(purchased) ) (
  PARTITION p0 VALUES LESS THAN (2000) PLACEMENT POLICY=storeonhdd,
  PARTITION p1 VALUES LESS THAN (2005),
  PARTITION p2 VALUES LESS THAN (2010),
  PARTITION p3 VALUES LESS THAN (2015),
  PARTITION p4 VALUES LESS THAN MAXVALUE PLACEMENT POLICY=storeonfastssd
);
```

该约束可通过列表格式 ([+disk=ssd]) 或字典格式 ({+disk=ssd: 1,+disk=hdd: 2}) 指定。

在列表格式中，约束以键值对列表格式。键以 + 或 - 开头。+disk=ssd 表示 disk 标签必须设为 ssd，-disk=hdd 表示 disk 标签值不能为 hdd。

在字典格式中，约束还指定了适用于该规则的多个实例。例如，FOLLOWER_CONSTRAINTS="{+region=us-east ↪ -1: 1,+region=us-east-2: 1,+region=us-west-1: 1}"; 表示 1 个 follower 位于 us-east-1，1 个 follower 位于 us-east-2，1 个 follower 位于 us-west-1。再例如，FOLLOWER_CONSTRAINTS='{"+region=us-east-1,+disk=hdd": ↪ 1,"+region=us-west-1": 1}'; 表示 1 个 follower 位于 us-east-1 区域中有 hdd 硬盘的机器上，1 个 follower 位于 us-west-1。

> **注意：**
>
> 字典和列表格式都基于 YAML 解析，但 YAML 语法有些时候不能被正常解析。例如 YAML 会把“{+disk=ssd:1,+disk=hdd:2}”错误地解析成‘{“+disk=ssd:1”: null, “+disk=hdd:1”: null}’，不符合预期。但“{+disk=ssd: 1,+disk=hdd: 1}”能被正确解析成‘{“+disk=ssd”: 1, “+disk=hdd”: 1}’。

2.16.5 工具兼容性

工具名称	最低兼容版本	说明
Backup & Restore (BR)	6.0	支持放置规则的导入与导出，见 BR 兼容性
TiDB Lightning	暂时不兼容	导入包含放置策略的数据时会报错
TiCDC	6.0	忽略放置规则，不同步规则到下游集群
TiDB Binlog	6.0	忽略放置规则，不同步规则到下游集群

2.16.6 使用限制

目前已知 Placement Rules in SQL 特性存在以下限制：

- 临时表不支持放置规则。
- 设置 `PRIMARY_REGION` 和 `REGIONS` 时允许存在语法糖。但在未来版本中，我们计划为 `PRIMARY_RACK`、`PRIMARY_ZONE` 和 `PRIMARY_HOST` 添加变体支持，见 issue #18030。
- 不能通过放置规则语法配置 TiFlash 副本。
- 放置规则仅保证静态数据被放置在正确的 TiKV 节点上。该规则不保证传输中的数据（通过用户查询或内部操作）只出现在特定区域内。

2.17 系统表

2.17.1 `mysql` Schema

`mysql` 库里存储的是 TiDB 系统表。该设计类似于 MySQL 中的 `mysql` 库，其中 `mysql.user` 之类的表可以直接编辑。该库还包含许多 MySQL 的扩展表。

2.17.1.1 权限系统表

这些系统表里面包含了用户账户以及相应的授权信息：

- `user` 用户账户，全局权限，以及其它一些非权限的列
- `db` 数据库级别的权限
- `tables_priv` 表级的权限
- `columns_priv` 列级的权限

2.17.1.2 服务端帮助信息系统表

- help_topic 目前为空

2.17.1.3 统计信息相关系统表

- stats_buckets 统计信息的桶
- stats_histograms 统计信息的直方图
- stats_meta 表的元信息，比如总行数和修改数
- analyze_jobs 正在执行的统计信息收集任务以及过去 7 天内的历史任务记录

2.17.1.4 GC Worker 相关系统表

- gc_delete_range

2.17.1.5 其它系统表

- GLOBAL_VARIABLES 全局系统变量表
- tidb 用于 TiDB 在 bootstrap 的时候记录相关版本信息

2.17.2 INFORMATION_SCHEMA

2.17.2.1 Information Schema

Information Schema 提供了一种查看系统元数据的 ANSI 标准方法。除了包含与 MySQL 兼容的表外，TiDB 还提供了许多自定义的 INFORMATION_SCHEMA 表。

许多 INFORMATION_SCHEMA 表都有相应的 SHOW 命令。查询 INFORMATION_SCHEMA 的好处是可以在表之间进行 join 操作。

2.17.2.1.1 与 MySQL 兼容的表

表名	描述
CHARACTER_SETS	提供 TiDB 支持的字符集列表。
COLLATIONS	提供 TiDB 支持的排序规则列表。
COLLATION_CHARACTER_SET_APPLICABILITY	说明哪些排序规则适用于哪些字符集。
COLUMNS	提供所有表中列的列表。
COLUMN_PRIVILEGES	TiDB 未实现，返回零行。
COLUMN_STATISTICS	TiDB 未实现，返回零行。
ENGINES	提供支持的存储引擎列表。
EVENTS	TiDB 未实现，返回零行。
FILES	TiDB 未实现，返回零行。
GLOBAL_STATUS	TiDB 未实现，返回零行。
GLOBAL_VARIABLES	TiDB 未实现，返回零行。

表名	描述
KEY_COLUMN_USAGE	描述列的键约束，例如主键约束。
OPTIMIZER_TRACE	TiDB 未实现，返回零行。
PARAMETERS	TiDB 未实现，返回零行。
PARTITIONS	提供表分区的列表。
PLUGINS	TiDB 未实现，返回零行。
PROCESSLIST	提供与 SHOW PROCESSLIST 命令类似的信息。
PROFILING	TiDB 未实现，返回零行。
REFERENTIAL_CONSTRAINTS	提供有关 FOREIGN KEY 约束的信息。
ROUTINES	TiDB 未实现，返回零行。
SCHEMATA	提供与 SHOW DATABASES 命令类似的信息。
SCHEMA_PRIVILEGES	TiDB 未实现，返回零行。
SESSION_STATUS	TiDB 未实现，返回零行。
SESSION_VARIABLES	提供与 SHOW SESSION VARIABLES 命令类似的功能。
STATISTICS	提供有关表索引的信息。
TABLES	提供当前用户可见的表的列表。类似于 SHOW TABLES。
TABLESPACES	TiDB 未实现，返回零行。
TABLE_CONSTRAINTS	提供有关主键、唯一索引和外键的信息。
TABLE_PRIVILEGES	TiDB 未实现，返回零行。
TRIGGERS	TiDB 未实现，返回零行。
USER_PRIVILEGES	汇总与当前用户相关的权限。
VIEWS	提供当前用户可见的视图列表。类似于 SHOW FULL TABLES WHERE table_type = '

2.17.2.1.2 TiDB 中的扩展表

表名	描述
ANALYZE_STATUS	提供有关收集统计信息的任务的信息。
CLIENT_ERRORS_SUMMARY_BY_HOST	汇总由客户端请求生成并返回给客户端的错误和警告。
CLIENT_ERRORS_SUMMARY_BY_USER	汇总由客户端产生的错误和警告。
CLIENT_ERRORS_SUMMARY_GLOBAL	汇总由客户端产生的错误和警告。
CLUSTER_CONFIG	提供有关整个 TiDB 集群的配置设置的详细信息。
CLUSTER_DEADLOCKS	提供 DEADLOCKS 表的集群级别的视图。
CLUSTER_HARDWARE	提供在每个 TiDB 组件上发现的底层物理硬件的详细信息。
CLUSTER_INFO	提供当前集群拓扑的详细信息。
CLUSTER_LOAD	提供集群中 TiDB 服务器的当前负载信息。
CLUSTER_LOG	提供整个 TiDB 集群的日志。
CLUSTER_PROCESSLIST	提供 PROCESSLIST 表的集群级别的视图。
CLUSTER_SLOW_QUERY	提供 SLOW_QUERY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY	提供 STATEMENTS_SUMMARY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY_HISTORY	提供 STATEMENTS_SUMMARY_HISTORY 表的集群级别的视图。
CLUSTER_TIDB_TRX	提供 TIDB_TRX 表的集群级别的视图。
CLUSTER_SYSTEMINFO	提供集群中服务器的内核参数配置的详细信息。
DATA_LOCK_WAITS	提供 TiKV 服务器上的等锁信息。

表名	描述
DDL_JOBS	提供与 ADMIN SHOW DDL JOBS 类似的输出。
DEADLOCKS	提供 TiDB 节点上最近发生的数次死锁错误的信息。
INSPECTION_RESULT	触发内部诊断检查。
INSPECTION_RULES	进行的内部诊断检查的列表。
INSPECTION_SUMMARY	重要监视指标的摘要报告。
METRICS_SUMMARY	从 Prometheus 获取的指标的摘要。
METRICS_SUMMARY_BY_LABEL	参见 METRICS_SUMMARY 表。
METRICS_TABLES	为 METRICS_SCHEMA 中的表提供 PromQL 定义。
PLACEMENT_POLICIES	提供所有放置策略的定义信息。
SEQUENCES	描述了基于 MariaDB 实现的 TiDB 序列。
SLOW_QUERY	提供当前 TiDB 服务器上慢查询的信息。
STATEMENTS_SUMMARY	类似于 MySQL 中的 performance_schema 语句摘要。
STATEMENTS_SUMMARY_HISTORY	类似于 MySQL 中的 performance_schema 语句摘要历史。
TABLE_STORAGE_STATS	提供存储的表的大小的详细信息。
TIDB_HOT_REGIONS	提供有关哪些 Region 访问次数最多的统计信息。
TIDB_HOT_REGIONS_HISTORY	提供有关哪些 Region 访问次数最多的历史统计信息。
TIDB_INDEXES	提供有关 TiDB 表的索引信息。
TIDB_SERVERS_INFO	提供 TiDB 服务器的列表
TIDB_TRX	提供 TiDB 节点上正在执行的事务的信息。
TIFLASH_REPLICA	提供有关 TiFlash 副本的详细信息。
TIKV_REGION_PEERS	提供 Region 存储位置的详细信息。
TIKV_REGION_STATUS	提供 Region 的统计信息。
TIKV_STORE_STATUS	提供 TiKV 服务器的基本信息。

2.17.2.2 ANALYZE_STATUS

ANALYZE_STATUS 表提供正在执行的收集统计信息的任务以及有限条历史任务记录。

从 TiDB v6.1.0 起，ANALYZE_STATUS 表显示集群级别的任务，且 TiDB 重启后仍能看到重启之前的任务记录。在 TiDB v6.1.0 之前，ANALYZE_STATUS 仅显示实例级别的任务，且 TiDB 重启后任务记录会被清空。

从 TiDB v6.1.0 起，可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史任务记录。

```
USE information_schema;
DESC analyze_status;
```

```
+----------------+---------------------+------+------+---------+-------+
| Field          | Type                | Null | Key  | Default | Extra |
+----------------+---------------------+------+------+---------+-------+
| TABLE_SCHEMA   | varchar(64)         | YES  |      | NULL    |       |
| TABLE_NAME     | varchar(64)         | YES  |      | NULL    |       |
| PARTITION_NAME | varchar(64)         | YES  |      | NULL    |       |
| JOB_INFO       | longtext            | YES  |      | NULL    |       |
| PROCESSED_ROWS | bigint(64) unsigned | YES  |      | NULL    |       |
| START_TIME     | datetime            | YES  |      | NULL    |       |
```

```
| END_TIME       | datetime             | YES  |      | NULL    |       |
| STATE          | varchar(64)          | YES  |      | NULL    |       |
| FAIL_REASON    | longtext             | YES  |      | NULL    |       |
| INSTANCE       | varchar(512)         | YES  |      | NULL    |       |
| PROCESS_ID     | bigint(64) unsigned  | YES  |      | NULL    |       |
+----------------+----------------------+------+------+---------+-------+
11 rows in set (0.00 sec)
```

```
SELECT * FROM information_schema.analyze_status;
```

```
+--
    ↪ ------------+------------+----------------+-------------------------------------------------------------
    ↪
| TABLE_SCHEMA | TABLE_NAME | PARTITION_NAME | JOB_INFO
    ↪                                                                | PROCESSED_ROWS | START_TIME
    ↪          | END_TIME            | STATE    | FAIL_REASON | INSTANCE       | PROCESS_ID |
+--
    ↪ ------------+------------+----------------+-------------------------------------------------------------
    ↪
| test         | t          | p1             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze index idx
    ↪                                                  |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx
    ↪                                                  |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns
    ↪                                                     |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns
    ↪                                                     |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--
    ↪ ------------+------------+----------------+-------------------------------------------------------------
    ↪
6 rows in set (0.00 sec)
```

ANALYZE_STATUS 表中列的含义如下：

- TABLE_SCHEMA：表所属的数据库的名称。

- TABLE_NAME：表的名称。
- PARTITION_NAME：分区表的名称。
- JOB_INFO：ANALYZE 任务的信息。如果分析索引，该信息会包含索引名。当 tidb_analyze_version =2 时，该信息会包含采样率等配置项。
- PROCESSED_ROWS：已经处理的行数。
- START_TIME：ANALYZE 任务的开始时间。
- END_TIME：ANALYZE 任务的结束时间。
- STATE：ANALYZE 任务的执行状态。其值可以是 pending、 running、 finished 或者 failed。
- FAIL_REASON：任务失败的原因。如果执行成功则为 NULL。
- INSTANCE：执行任务的 TiDB 实例。
- PROCESS_ID：执行任务的 process ID。

2.17.2.3 CLIENT_ERRORS_SUMMARY_BY_HOST

CLIENT_ERRORS_SUMMARY_BY_HOST 表汇总了已返回给连接到 TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过 MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。information_schema ↪ .CLIENT_ERRORS_SUMMARY_BY_HOST 表提供了一种有效方法，能够在应用程序没有正确处理（或记录）TiDB 服务器返回的错误的情况下检查错误。

由于 CLIENT_ERRORS_SUMMARY_BY_HOST 会基于每个远程主机汇总错误，在诊断其中一台应用程序服务器比其他服务器生成更多错误的情况时很有用。可能的情况包括：

- 过时的 MySQL 客户端库。
- 过时的应用程序（可能是在推出新部署时遗漏了此服务器）。
- 用户权限中“主机”部分的使用不正确。
- 网络连接不可靠，导致更多超时或连接断开。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个 TiDB 服务器的本地数据，并且只保留在内存中。如果 TiDB 服务器重新启动，会丢失汇总信息。

```
USE information_schema;
DESC CLIENT_ERRORS_SUMMARY_BY_HOST;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| HOST          | varchar(255)  | NO   |      | NULL    |       |
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
```

```
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段说明如下：

- HOST：客户端的远程主机。
- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：此错误返回到客户端主机的次数。
- WARNING_COUNT：此警告返回到客户端主机的次数。
- FIRST_SEEN：首次从客户端主机看到此错误（或警告）的时间。
- LAST_SEEN：最近一次从客户端主机看到此错误（或警告）的时间。

以下示例显示了客户端连接到本地 TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_HOST;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_HOST;
```

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    ↪ ---------+--------------+---------------+-------------+---------------+---------------------+-----------
    ↪
| HOST      | ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          |
    ↪ LAST_SEEN           |
+--
    ↪ ---------+--------------+---------------+-------------+---------------+---------------------+-----------
    ↪
| 127.0.0.1 |         1365 | Division by 0 |           0 |             1 | 2021-03-18 12:51:54 |
    ↪ 2021-03-18 12:51:54 |
+--
    ↪ ---------+--------------+---------------+-------------+---------------+---------------------+-----------
    ↪
```

```
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

2.17.2.4 CLIENT_ERRORS_SUMMARY_BY_USER

CLIENT_ERRORS_SUMMARY_BY_USER 表汇总了已返回给连接到 TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过 MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。information_schema ↪ .CLIENT_ERRORS_SUMMARY_BY_USER 表提供了一种有效方法，能够在应用程序没有正确处理（或记录）TiDB 服务器返回的错误的情况下检查错误。

因为 CLIENT_ERRORS_SUMMARY_BY_USER 会基于每个用户汇总错误，所以在诊断一个用户服务器比其他服务器产生更多错误的方案时很有用。可能的情况包括：

- 权限错误。
- 表或关系对象丢失。
- SQL 语法不正确，或应用程序和 TiDB 版本之间不兼容。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个 TiDB 服务器的本地数据，并且只保留在内存中。如果 TiDB 服务器重新启动，会丢失汇总信息。

```
USE information_schema;
DESC CLIENT_ERRORS_SUMMARY_BY_USER;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| USER          | varchar(64)   | NO   |      | NULL    |       |
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段说明如下：

- USER：已认证的用户。
- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：返回此错误给用户的次数。
- WARNING_COUNT：返回此警告给用户的次数。
- FIRST_SEEN：首次向用户发送此错误（或警告）的时间。
- LAST_SEEN：最近一次向用户发送此错误（或警告）的时间。

以下示例显示了客户端连接到本地 TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_USER;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_USER;
```

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    ↪ ----+--------------+---------------+-------------+---------------+---------------------+----------------
    ↪
| USER | ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          |
    ↪ LAST_SEEN           |
+--
    ↪ ----+--------------+---------------+-------------+---------------+---------------------+----------------
    ↪
| root |         1365 | Division by 0 |           0 |             1 | 2021-03-18 13:05:36 |
    ↪ 2021-03-18 13:05:36 |
+--
    ↪ ----+--------------+---------------+-------------+---------------+---------------------+----------------
    ↪
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

2.17.2.5 CLIENT_ERRORS_SUMMARY_GLOBAL

CLIENT_ERRORS_SUMMARY_GLOBAL 表汇总了已返回给连接到 TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过 MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。information_schema ↪ .CLIENT_ERRORS_SUMMARY_BY_GLOBAL 表提供了高级概述，在应用程序无法正确处理（或记录）TiDB 服务器返回的错误的情况下很有用。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个 TiDB 服务器的本地数据，并且只保留在内存中。如果 TiDB 服务器重新启动，会丢失汇总信息。

```
USE information_schema;
DESC CLIENT_ERRORS_SUMMARY_GLOBAL;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段说明如下：

- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：返回此错误的次数。
- WARNING_COUNT：返回此警告的次数。
- FIRST_SEEN：首次返回此错误（或警告）的时间。
- LAST_SEEN：最近一次发送此错误（或警告）的时间。

以下示例显示了客户端连接到本地 TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_GLOBAL;
```

```
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_GLOBAL;
```

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    ↪ ------------+---------------+-------------+---------------+---------------------+---------------------+
    ↪
| ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          | LAST_SEEN
    ↪            |
+--
    ↪ ------------+---------------+-------------+---------------+---------------------+---------------------+
    ↪
|         1365 | Division by 0 |           0 |             1 | 2021-03-18 13:10:51 | 2021-03-18
    ↪ 13:10:51 |
+--
    ↪ ------------+---------------+-------------+---------------+---------------------+---------------------+
    ↪
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

2.17.2.6 CHARACTER_SETS

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

```
USE information_schema;
DESC character_sets;
```

```
+----------------------+-------------+------+------+---------+-------+
| Field                | Type        | Null | Key  | Default | Extra |
+----------------------+-------------+------+------+---------+-------+
| CHARACTER_SET_NAME   | varchar(32) | YES  |      | NULL    |       |
| DEFAULT_COLLATE_NAME | varchar(32) | YES  |      | NULL    |       |
| DESCRIPTION          | varchar(60) | YES  |      | NULL    |       |
| MAXLEN               | bigint(3)   | YES  |      | NULL    |       |
+----------------------+-------------+------+------+---------+-------+
4 rows in set (0.00 sec)
```

```
SELECT * FROM `character_sets`;
```

```
+--------------------+----------------------+---------------+--------+
| CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME | DESCRIPTION   | MAXLEN |
+--------------------+----------------------+---------------+--------+
| utf8               | utf8_bin             | UTF-8 Unicode |      3 |
| utf8mb4            | utf8mb4_bin          | UTF-8 Unicode |      4 |
| ascii              | ascii_bin            | US ASCII      |      1 |
| latin1             | latin1_bin           | Latin1        |      1 |
| binary             | binary               | binary        |      1 |
+--------------------+----------------------+---------------+--------+
5 rows in set (0.00 sec)
```

CHARACTER_SETS 表中列的含义如下：

- CHARACTER_SET_NAME：字符集名称
- DEFAULT_COLLATE_NAME：字符集的默认排序规则名称
- DESCRIPTION：字符集的描述信息
- MAXLEN：该字符集存储一个字符所需要的最大字节数

2.17.2.7 CLUSTER_CONFIG

CLUSTER_CONFIG 表用于获取集群当前所有组件实例的配置。在 TiDB 早期的版本，用户需要逐个访问各个实例的 HTTP API 才能收集到所有组件配置。TiDB v4.0 后，该表的引入提高了易用性。

```
USE information_schema;
DESC cluster_config;
```

```
+----------+--------------+------+------+---------+-------+
| Field    | Type         | Null | Key  | Default | Extra |
+----------+--------------+------+------+---------+-------+
| TYPE     | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE | varchar(64)  | YES  |      | NULL    |       |
| KEY      | varchar(256) | YES  |      | NULL    |       |
| VALUE    | varchar(128) | YES  |      | NULL    |       |
+----------+--------------+------+------+---------+-------+
```

字段解释：

- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- KEY：配置项名。
- VALUE：配置项值。

以下示例查询 TiKV 实例的 coprocessor 相关配置：

```
SELECT * FROM cluster_config WHERE type='tikv' AND `key` LIKE 'coprocessor%';
```

```
+------+-----------------+-----------------------------------+---------+
| TYPE | INSTANCE        | KEY                               | VALUE   |
+------+-----------------+-----------------------------------+---------+
| tikv | 127.0.0.1:20165 | coprocessor.batch-split-limit     | 10      |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-keys       | 1440000 |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-size       | 144MiB  |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-keys     | 960000  |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-size     | 96MiB   |
| tikv | 127.0.0.1:20165 | coprocessor.split-region-on-table | false   |
+------+-----------------+-----------------------------------+---------+
6 rows in set (0.00 sec)
```

2.17.2.8 CLUSTER_HARDWARE

集群硬件表 CLUSTER_HARDWARE 提供了集群各实例所在服务器的硬件信息。

```
USE information_schema;
DESC cluster_hardware;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应集群信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于集群信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型。目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 的取值不同。
  - cpu：硬件名为 cpu。
  - memory：硬件名为 memory。
  - disk：磁盘名。
  - net：网卡名。

- NAME：硬件不同的信息名，比如 cpu 有 cpu-logical-cores 和 cpu-physical-cores 两个信息名，表示逻辑核心数量和物理核心数量。
- VALUE：对应硬件信息的值。例如磁盘容量和 CPU 核数。

查询集群 CPU 信息的示例如下：

```
SELECT * FROM cluster_hardware WHERE device_type='cpu' AND device_name='cpu' AND name LIKE '%
    ↪ cores';
```

```
+------+-----------------+-------------+-------------+--------------------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME               | VALUE |
+------+-----------------+-------------+-------------+--------------------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-logical-cores  | 16    |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-physical-cores | 8     |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-logical-cores  | 16    |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-physical-cores | 8     |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-logical-cores  | 16    |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-physical-cores | 8     |
+------+-----------------+-------------+-------------+--------------------+-------+
6 rows in set (0.03 sec)
```

2.17.2.9 CLUSTER_INFO

集群拓扑表 CLUSTER_INFO 提供集群当前的拓扑信息，以及各个节点的版本信息、版本对应的 Git Hash、各节点的启动时间、各实例的运行时间。

```
USE information_schema;
desc cluster_info;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| INSTANCE       | varchar(64) | YES  |      | NULL    |       |
| STATUS_ADDRESS | varchar(64) | YES  |      | NULL    |       |
| VERSION        | varchar(64) | YES  |      | NULL    |       |
| GIT_HASH       | varchar(64) | YES  |      | NULL    |       |
| START_TIME     | varchar(32) | YES  |      | NULL    |       |
| UPTIME         | varchar(32) | YES  |      | NULL    |       |
+----------------+-------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段解释：

- TYPE：节点类型，目前节点的可取值为 tidb，pd 和 tikv。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。

- STATUS_ADDRESS：HTTP API 的服务地址。部分 tikv-ctl、pd-ctl 或 tidb-ctl 命令会使用到 HTTP API 和该地址。用户也可以通过该地址获取一些额外的集群信息，详情可参考 HTTP API 文档。
- VERSION：对应节点的语义版本号。TiDB 版本为了兼容 MySQL 的版本号，以 ${mysql-version}-${tidb-version} 的格式展示版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。

```
SELECT * FROM cluster_info;
```

```
+--
    ↪ ----+-----------------+-----------------+--------------+----------------------------------------+------
    ↪
| TYPE | INSTANCE        | STATUS_ADDRESS   | VERSION      | GIT_HASH
    ↪                                  | START_TIME                | UPTIME              |
+--
    ↪ ----+-----------------+-----------------+--------------+----------------------------------------+------
    ↪
| tidb | 0.0.0.0:4000    | 0.0.0.0:10080    | 4.0.0-beta.2 | 0
    ↪ df3b74f55f8f8fbde39bbd5d471783f49dc10f7 | 2020-07-05T09:25:53-06:00 | 26h39m4.352862693s
    ↪  |
| pd   | 127.0.0.1:2379  | 127.0.0.1:2379   | 4.1.0-alpha  | 1
    ↪ ad59bcbf36d87082c79a1fffa3b0895234ac862 | 2020-07-05T09:25:47-06:00 | 26h39m10.352868103s
    ↪  |
| tikv | 127.0.0.1:20165 | 127.0.0.1:20180 | 4.1.0-alpha  |
    ↪ b45e052df8fb5d66aa8b3a77b5c992ddbfbb79df | 2020-07-05T09:25:50-06:00 | 26h39m7.352869963s
    ↪   |
+--
    ↪ ----+-----------------+-----------------+--------------+----------------------------------------+------
    ↪
3 rows in set (0.00 sec)
```

2.17.2.10 CLUSTER_LOAD

集群负载表 CLUSTER_LOAD 提供集群各个实例所在服务器的当前负载信息。

```
USE information_schema;
DESC cluster_load;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
```

```
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型，目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 取值不同。
  - cpu：硬件名为 cpu。
  - disk：磁盘名。
  - net：网卡名。
  - memory：硬件名为 memory。
- NAME：不同的负载类型。例如 cpu 有 load1/load5/load15 三个负载类型，分别表示 cpu 在 1min/5min/15min 内的平均负载。
- VALUE：硬件负载的值，例如 cpu 在 1min/5min/15min 内的平均负载。

查询集群当前的 CPU 负载信息示例如下：

```
SELECT * FROM cluster_load WHERE device_type='cpu' AND device_name='cpu';
```

```
+------+-----------------+-------------+-------------+--------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME   | VALUE |
+------+-----------------+-------------+-------------+--------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load1  | 0.13  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load5  | 0.25  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load15 | 0.31  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load1  | 0.13  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load5  | 0.25  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load15 | 0.31  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load1  | 0.13  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load5  | 0.25  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load15 | 0.31  |
+------+-----------------+-------------+-------------+--------+-------+
9 rows in set (1.50 sec)
```

2.17.2.11 CLUSTER_LOG

集群日志表 CLUSTER_LOG 表用于查询集群当前所有 TiDB/PD/TiKV 节点日志。它通过将查询条件下推到各个节点，降低了日志查询对集群的影响。该表的查询性能优于 grep 命令。

TiDB 4.0 版本之前，要获取集群的日志，用户需要逐个登录各个节点汇总日志。TiDB 4.0 的集群日志表提供了一个全局且时间有序的日志搜索结果，为跟踪全链路事件提供了便利的手段。例如按照某一个 region id 搜索日志，可以查询该 Region 生命周期内的所有日志；类似地，通过慢日志的 txn id 搜索全链路日志，可以查询该事务在各个节点扫描的 key 数量以及流量等信息。

```
USE information_schema;
DESC cluster_log;
```

```
+----------+------------------+------+------+---------+-------+
| Field    | Type             | Null | Key  | Default | Extra |
+----------+------------------+------+------+---------+-------+
| TIME     | varchar(32)      | YES  |      | NULL    |       |
| TYPE     | varchar(64)      | YES  |      | NULL    |       |
| INSTANCE | varchar(64)      | YES  |      | NULL    |       |
| LEVEL    | varchar(8)       | YES  |      | NULL    |       |
| MESSAGE  | var_string(1024) | YES  |      | NULL    |       |
+----------+------------------+------+------+---------+-------+
5 rows in set (0.00 sec)
```

字段解释：

- TIME：日志打印时间。
- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- LEVEL：日志级别。
- MESSAGE：日志内容。

> 注意：
>
> - 日志表的所有字段都会下推到对应节点执行，所以为了降低使用集群日志表的开销，必须指定搜索关键字以及时间范围，然后尽可能地指定更多的条件。例如 select * from cluster_log where message like '%ddl%' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00' and type='tidb'。
> - message 字段支持 like 和 regexp 正则表达式，对应的 pattern 会编译为 regexp。同时指定多个 message 条件，相当于 grep 命令的 pipeline 形式，例如：select * from cluster_log where message like 'coprocessor%' and message regexp '.*slow.*' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00' 相当于在集群所有节点执行 grep 'coprocessor' xxx.log | grep -E '.*slow.*'。

查询某个 DDL 的执行过程示例如下：

```
SELECT time,instance,left(message,150) FROM cluster_log WHERE message LIKE '%ddl%job%ID.80%' AND
    ↪ type='tidb' AND time > '2020-05-18 20:40:00' AND time < '2020-05-18 21:40:00'
```

```
+--
    ↪ ----------------------+---------------+-----------------------------------------------------------------
    ↪
| time                     | instance       | left(message,150)
    ↪
    ↪ |
+--
    ↪ ----------------------+---------------+-----------------------------------------------------------------
    ↪
| 2020/05/18 21:37:54.784 | 127.0.0.1:4002 | [ddl_worker.go:261] ["[ddl] add DDL jobs"] ["batch
    ↪ count"=1] [jobs="ID:80, Type:create table, State:none, SchemaState:none, SchemaID:1,
    ↪ TableID:79, Ro |
| 2020/05/18 21:37:54.784 | 127.0.0.1:4002 | [ddl.go:477] ["[ddl] start DDL job"] [job="ID:80,
    ↪ Type:create table, State:none, SchemaState:none, SchemaID:1, TableID:79, RowCount:0,
    ↪ ArgLen:1, start |
| 2020/05/18 21:37:55.327 | 127.0.0.1:4000 | [ddl_worker.go:568] ["[ddl] run DDL job"] [worker="
    ↪ worker 1, tp general"] [job="ID:80, Type:create table, State:none, SchemaState:none,
    ↪ SchemaID:1, Ta |
| 2020/05/18 21:37:55.381 | 127.0.0.1:4000 | [ddl_worker.go:763] ["[ddl] wait latest schema
    ↪ version changed"] [worker="worker 1, tp general"] [ver=70] ["take time"=50.809848ms] [job
    ↪ ="ID:80, Type: |
| 2020/05/18 21:37:55.382 | 127.0.0.1:4000 | [ddl_worker.go:359] ["[ddl] finish DDL job"] [worker
    ↪ ="worker 1, tp general"] [job="ID:80, Type:create table, State:synced, SchemaState:public
    ↪ , SchemaI |
| 2020/05/18 21:37:55.786 | 127.0.0.1:4002 | [ddl.go:509] ["[ddl] DDL job is finished"] [jobID
    ↪ =80]
    ↪
    ↪ |
+-------------------------+---------------+-----------------------------------------------------------------
    ↪
```

上面查询结果记录了一个 DDL 执行的过程：

- 用户将 DDL JOB ID 为 80 的请求发给 127.0.0.1:4002 TiDB 节点。
- 127.0.0.1:4000 TiDB 节点处理这个 DDL 请求，说明此时 127.0.0.1:4000 节点是 DDL owner。
- DDL JOB ID 为 80 的请求处理完成。

2.17.2.12 CLUSTER_SYSTEMINFO

内核参数表 CLUSTER_SYSTEMINFO 用于查询集群所有实例所在服务器的内核配置信息。目前支持查询 sysctl 的信息。

```
USE information_schema;
DESC cluster_systeminfo;
```

```
+-------------+-------------+------+------+---------+-------+
| Field       | Type        | Null | Key  | Default | Extra |
+-------------+-------------+------+------+---------+-------+
| TYPE        | varchar(64) | YES  |      | NULL    |       |
| INSTANCE    | varchar(64) | YES  |      | NULL    |       |
| SYSTEM_TYPE | varchar(64) | YES  |      | NULL    |       |
| SYSTEM_NAME | varchar(64) | YES  |      | NULL    |       |
| NAME        | varchar(256)| YES  |      | NULL    |       |
| VALUE       | varchar(128)| YES  |      | NULL    |       |
+-------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- SYSTEM_TYPE：系统类型，目前可以查询的系统类型有 system。
- SYSTEM_NAME：目前可以查询的 SYSTEM_NAME 为 sysctl。
- NAME：sysctl 对应的配置名。
- VALUE：sysctl 对应配置项的值。

查询集群所有服务器的内核版本示例如下：

```
SELECT * FROM cluster_systeminfo WHERE name LIKE '%kernel.osrelease%'
```

```
+--
    ↪ ----+------------------+-------------+-------------+------------------+--------------------------+
    ↪
| TYPE | INSTANCE          | SYSTEM_TYPE | SYSTEM_NAME | NAME             | VALUE
    ↪                       |
+--
    ↪ ----+------------------+-------------+-------------+------------------+--------------------------+
    ↪
| tidb | 172.16.5.40:4008  | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
| pd   | 172.16.5.40:20379 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
| tikv | 172.16.5.40:21150 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
+--
    ↪ ----+------------------+-------------+-------------+------------------+--------------------------+
    ↪
```

2.17.2.13 COLLATIONS

COLLATIONS 表提供了 CHARACTER_SETS 表中字符集对应的排序规则列表。目前 TiDB 包含该表仅为兼容 MySQL。

```
USE information_schema;
DESC collations;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| COLLATION_NAME     | varchar(32) | YES  |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | YES  |      | NULL    |       |
| ID                 | bigint(11)  | YES  |      | NULL    |       |
| IS_DEFAULT         | varchar(3)  | YES  |      | NULL    |       |
| IS_COMPILED        | varchar(3)  | YES  |      | NULL    |       |
| SORTLEN            | bigint(3)   | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

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

```
+--------------------+--------------------+------+------------+-------------+---------+
| COLLATION_NAME     | CHARACTER_SET_NAME | ID   | IS_DEFAULT | IS_COMPILED | SORTLEN |
+--------------------+--------------------+------+------------+-------------+---------+
| utf8mb4_bin        | utf8mb4            |   46 | Yes        | Yes         |       1 |
| utf8mb4_general_ci | utf8mb4            |   45 |            | Yes         |       1 |
| utf8mb4_unicode_ci | utf8mb4            |  224 |            | Yes         |       1 |
+--------------------+--------------------+------+------------+-------------+---------+
3 rows in set (0.001 sec)
```

COLLATIONS 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称
- ID：排序规则的 ID
- IS_DEFAULT：该排序规则是否是所属字符集的默认排序规则
- IS_COMPILED：字符集是否编译到服务器中
- SORTLEN：排序规则在对字符进行排序时，所分配内存的最小长度

2.17.2.14 COLLATION_CHARACTER_SET_APPLICABILITY

COLLATION_CHARACTER_SET_APPLICABILITY 表将排序规则映射至适用的字符集名称。和 COLLATIONS 表一样，包含此表只是为了兼容 MySQL。

```
USE information_schema;
DESC collation_character_set_applicability;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| COLLATION_NAME     | varchar(32) | NO   |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | NO   |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
2 rows in set (0.00 sec)
```

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

```
+----------------+--------------------+
| COLLATION_NAME | CHARACTER_SET_NAME |
+----------------+--------------------+
| utf8mb4_bin    | utf8mb4            |
+----------------+--------------------+
1 row in set (0.00 sec)
```

COLLATION_CHARACTER_SET_APPLICABILITY 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称

2.17.2.15 COLUMNS

COLUMNS 表提供了表的所有列的信息。

```
USE information_schema;
DESC columns;
```

```
+-------------------------+---------------+------+------+---------+-------+
| Field                   | Type          | Null | Key  | Default | Extra |
+-------------------------+---------------+------+------+---------+-------+
| TABLE_CATALOG           | varchar(512)  | YES  |      | NULL    |       |
| TABLE_SCHEMA            | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME              | varchar(64)   | YES  |      | NULL    |       |
| COLUMN_NAME             | varchar(64)   | YES  |      | NULL    |       |
| ORDINAL_POSITION        | bigint(64)    | YES  |      | NULL    |       |
| COLUMN_DEFAULT          | text          | YES  |      | NULL    |       |
| IS_NULLABLE             | varchar(3)    | YES  |      | NULL    |       |
| DATA_TYPE               | varchar(64)   | YES  |      | NULL    |       |
| CHARACTER_MAXIMUM_LENGTH | bigint(21)   | YES  |      | NULL    |       |
| CHARACTER_OCTET_LENGTH  | bigint(21)    | YES  |      | NULL    |       |
| NUMERIC_PRECISION       | bigint(21)    | YES  |      | NULL    |       |
| NUMERIC_SCALE           | bigint(21)    | YES  |      | NULL    |       |
| DATETIME_PRECISION      | bigint(21)    | YES  |      | NULL    |       |
```

```
| CHARACTER_SET_NAME       | varchar(32)   | YES  |     | NULL    |       |
| COLLATION_NAME           | varchar(32)   | YES  |     | NULL    |       |
| COLUMN_TYPE              | text          | YES  |     | NULL    |       |
| COLUMN_KEY               | varchar(3)    | YES  |     | NULL    |       |
| EXTRA                    | varchar(30)   | YES  |     | NULL    |       |
| PRIVILEGES               | varchar(80)   | YES  |     | NULL    |       |
| COLUMN_COMMENT           | varchar(1024) | YES  |     | NULL    |       |
| GENERATION_EXPRESSION    | text          | NO   |     | NULL    |       |
+--------------------------+---------------+------+-----+---------+-------+
21 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (a int);
SELECT * FROM columns WHERE table_schema='test' AND TABLE_NAME='t1'\G
```

```
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: test
              TABLE_NAME: t1
             COLUMN_NAME: a
        ORDINAL_POSITION: 1
          COLUMN_DEFAULT: NULL
             IS_NULLABLE: YES
               DATA_TYPE: int
CHARACTER_MAXIMUM_LENGTH: NULL
  CHARACTER_OCTET_LENGTH: NULL
       NUMERIC_PRECISION: 11
           NUMERIC_SCALE: 0
      DATETIME_PRECISION: NULL
      CHARACTER_SET_NAME: NULL
          COLLATION_NAME: NULL
             COLUMN_TYPE: int(11)
              COLUMN_KEY:
                   EXTRA:
              PRIVILEGES: select,insert,update,references
          COLUMN_COMMENT:
   GENERATION_EXPRESSION:
1 row in set (0.02 sec)
```

COLUMNS 表中列的含义如下：

- TABLE_CATALOG：包含列的表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：包含列的表所属的数据库的名称。
- TABLE_NAME：包含列的表的名称。
- COLUMN_NAME：列的名称。
- ORDINAL_POSITION：表中列的位置。

- COLUMN_DEFAULT：列的默认值。如果列的显式默认值为 NULL，或者列定义中不包含 default 子句，则此值为 NULL。
- IS_NULLABLE：列的可空性。如果列中可以存储空值，则该值为 YES，否则为 NO。
- DATA_TYPE：列的数据类型。
- CHARACTER_MAXIMUM_LENGTH：对于字符串列，以字符为单位的最大长度。
- CHARACTER_OCTET_LENGTH：对于字符串列，以字节为单位的最大长度。
- NUMERIC_PRECISION：对于数字列，为数字精度。
- NUMERIC_SCALE：对于数字列，为数字刻度。
- DATETIME_PRECISION：对于时间列，小数秒精度。
- CHARACTER_SET_NAME：对于字符串列，字符集名称。
- COLLATION_NAME：对于字符串列，排序规则名称。
- COLUMN_TYPE：列类型。
- COLUMN_KEY：该列是否被索引。具体显示如下：
  - 如果此值为空，则该列要么未被索引，要么被索引且是多列非唯一索引中的第二列。
  - 如果此值是 PRI，则该列是主键，或者是多列主键中的一列。
  - 如果此值是 UNI，则该列是唯一索引的第一列。
  - 如果此值是 MUL，则该列是非唯一索引的第一列，在该列中允许给定值的多次出现。
- EXTRA：关于给定列的任何附加信息。
- PRIVILEGES：当前用户对该列拥有的权限。目前在 TiDB 中，此值为定值，一直为 select,insert,update,references。
- COLUMN_COMMENT：列定义中包含的注释。
- GENERATION_EXPRESSION：对于生成的列，显示用于计算列值的表达式。对于未生成的列为空。

对应的 SHOW 语句如下：

```
SHOW COLUMNS FROM t1 FROM test;
```

```
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
1 row in set (0.00 sec)
```

2.17.2.16 DATA_LOCK_WAITS

DATA_LOCK_WAITS 表展示了集群中所有 TiKV 节点上当前正在发生的悲观锁等锁的情况。

```
USE information_schema;
DESC data_lock_waits;
```

```
+-----------------------+---------------------+------+------+---------+-------+
| Field                 | Type                | Null | Key  | Default | Extra |
+-----------------------+---------------------+------+------+---------+-------+
| KEY                   | text                | NO   |      | NULL    |       |
```

```
| KEY_INFO               | text                | YES  |     | NULL    |       |
| TRX_ID                 | bigint(21) unsigned | NO   |     | NULL    |       |
| CURRENT_HOLDING_TRX_ID | bigint(21) unsigned | NO   |     | NULL    |       |
| SQL_DIGEST             | varchar(64)         | YES  |     | NULL    |       |
| SQL_DIGEST_TEXT        | text                | YES  |     | NULL    |       |
+------------------------+---------------------+------+-----+---------+-------+
```

DATA_LOCK_WAITS 表中各列的字段含义如下：

- KEY：正在发生等锁的 key，以十六进制编码的形式显示。
- KEY_INFO：对 KEY 进行解读得出的一些详细信息，见KEY_INFO。
- TRX_ID：正在等锁的事务 ID，即 start_ts。
- CURRENT_HOLDING_TRX_ID：当前持有锁的事务 ID，即 start_ts。
- SQL_DIGEST：当前正在等锁的事务中被阻塞的 SQL 语句的 Digest。
- SQL_DIGEST_TEXT：当前正在等锁的事务中被阻塞的 SQL 语句的归一化形式，即去除了参数和格式的 SQL 语句。与 SQL_DIGEST 对应。

> 警告：
>
> - 仅拥有 PROCESS 权限的用户可以查询该表。
> - 目前该表仅能记录悲观锁等锁的情况，如果乐观事务（如 autocommit 的事务）被悲观锁阻塞，这一表中不会显示相应等锁信息。
> - DATA_LOCK_WAITS 表中的信息是在查询时，从所有 TiKV 节点实时获取的。目前，即使加上了 WHERE 查询条件，也无法避免对所有 TiKV 节点都进行信息收集。如果集群规模很大、负载很高，查询该表有造成性能抖动的潜在风险，因此请根据实际情况使用。
> - 来自不同 TiKV 节点的信息不能保证是同一时间点的快照。
> - SQL_DIGEST 列中的信息（SQL Digest）为 SQL 语句进行归一化后计算得到的哈希值。SQL_DIGEST_TEXT 列中的信息为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅 Statement Summary Tables。

2.17.2.16.1 KEY_INFO

KEY_INFO 列中展示了对 KEY 列中所给出的 key 的详细信息，以 JSON 格式给出。其包含的信息如下：

- "db_id"：该 key 所属的数据库（schema）的 ID。
- "db_name"：该 key 所属的数据库（schema）的名称。
- "table_id"：该 key 所属的表的 ID。
- "table_name"：该 key 所属的表的名称。
- "partition_id"：该 key 所在的分区（partition）的 ID。
- "partition_name"：该 key 所在的分区（partition）的名称。
- "handle_type"：该 row key（即储存一行数据的 key）的 handle 类型，其可能的值有：

  - "int"：handle 为 int 类型，即 handle 为 row ID
  - "common"：非 int64 类型的 handle，在启用 clustered index 时非 int 类型的主键会显示为此类型
  - "unknown"：当前暂不支持的 handle 类型

- "handle_value"：handle 的值。
- "index_id"：该 index key （即储存索引的 key ）所属的 index ID。
- "index_name"：该 index key 所属的 index 名称。
- "index_values"：该 index key 中的 index value。

其中，不适用或当前无法查询到的信息会被省略。比如，row key 的信息中不会包含 index_id、index_name 和 index_values；index key 不会包含 handle_type 和 handle_value；非分区表不会显示 partition_id 和 partition_name；已经被删除掉的表中的 key 的信息无法获取 table_name、db_id、db_name、index_name 等 schema 信息，且无法区分是否为分区表。

> 注意：
>
> 如果一个 key 来自一张启用了分区的表，而在查询时，由于某些原因（例如，其所属的表已经被删除）导致无法查询其所属的 schema 信息，则其所属的分区的 ID 可能会出现在 table_id 字段中。这是因为，TiDB 对不同分区的 key 的编码方式与对几张独立的表的 key 的编码方式一致，因而在缺失 schema 信息时无法确认该 key 属于一张未分区的表还是某张表的一个分区。

2.17.2.16.2 示例

```
select * from information_schema.data_lock_waits\G
```

```
*************************** 1. row ***************************
                  KEY: 748000000000000355F728000000000000001
             KEY_INFO: {"db_id":1,"db_name":"test","table_id":53,"table_name":"t","handle_type":
                 ↪ "int","handle_value":"1"}
               TRX_ID: 426790594290122753
CURRENT_HOLDING_TRX_ID: 426790590082449409
            SQL_DIGEST: 38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821
       SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ?
1 row in set (0.01 sec)
```

以上查询结果显示，ID 为 426790594290122753 的事务在执行 Digest 为 "38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e7
↪ "、形如 update `t` set `v` = `v` + ? where `id` = ? 的语句的过程中，试图在 "748000000000000355
↪ F728000000000000001" 这个 key 上获取悲观锁，但是该 key 上的锁目前被 ID 为 426790590082449409 的事务持有。

2.17.2.17 DDL_JOBS

DDL_JOBS 表为 ADMIN SHOW DDL JOBS 命令提供了一个 INFORMATION_SCHEMA 接口。它提供了 TiDB 集群中 DDL 操作的当前状态和简短历史记录。

```
USE information_schema;
DESC ddl_jobs;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| JOB_ID       | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME   | varchar(64) | YES  |      | NULL    |       |
| JOB_TYPE     | varchar(64) | YES  |      | NULL    |       |
| SCHEMA_STATE | varchar(64) | YES  |      | NULL    |       |
| SCHEMA_ID    | bigint(21)  | YES  |      | NULL    |       |
| TABLE_ID     | bigint(21)  | YES  |      | NULL    |       |
| ROW_COUNT    | bigint(21)  | YES  |      | NULL    |       |
| START_TIME   | datetime    | YES  |      | NULL    |       |
| END_TIME     | datetime    | YES  |      | NULL    |       |
| STATE        | varchar(64) | YES  |      | NULL    |       |
| QUERY        | varchar(64) | YES  |      | NULL    |       |
+--------------+-------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM ddl_jobs LIMIT 3\G
```

```
*************************** 1. row ***************************
      JOB_ID: 44
     DB_NAME: mysql
  TABLE_NAME: opt_rule_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 43
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.opt_rule_blacklist (
        name char(100) NOT NULL
    );
*************************** 2. row ***************************
      JOB_ID: 42
     DB_NAME: mysql
  TABLE_NAME: expr_pushdown_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
```

```
   SCHEMA_ID: 3
    TABLE_ID: 41
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.expr_pushdown_blacklist (
        name char(100) NOT NULL,
        store_type char(100) NOT NULL DEFAULT 'tikv,tiflash,tidb',
        reason varchar(200)
    );
*************************** 3. row ***************************
      JOB_ID: 40
     DB_NAME: mysql
  TABLE_NAME: stats_top_n
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 39
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:26
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE if not exists mysql.stats_top_n (
        table_id bigint(64) NOT NULL,
        is_index tinyint(2) NOT NULL,
        hist_id bigint(64) NOT NULL,
        value longblob,
        count bigint(64) UNSIGNED NOT NULL,
        index tbl(table_id, is_index, hist_id)
    );
3 rows in set (0.01 sec)
```

2.17.2.18 DEADLOCKS

DEADLOCKS 表提供当前 TiDB 节点上最近发生的若干次死锁错误的信息。

```
USE information_schema;
DESC deadlocks;
```

```
+-------------------------+---------------------+------+------+---------+-------+
| Field                   | Type                | Null | Key  | Default | Extra |
+-------------------------+---------------------+------+------+---------+-------+
| DEADLOCK_ID             | bigint(21)          | NO   |      | NULL    |       |
| OCCUR_TIME              | timestamp(6)        | YES  |      | NULL    |       |
```

```
| RETRYABLE               | tinyint(1)          | NO   |     | NULL    |       |
| TRY_LOCK_TRX_ID         | bigint(21) unsigned | NO   |     | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)         | YES  |     | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                | YES  |     | NULL    |       |
| KEY                     | text                | YES  |     | NULL    |       |
| KEY_INFO                | text                | YES  |     | NULL    |       |
| TRX_HOLDING_LOCK        | bigint(21) unsigned | NO   |     | NULL    |       |
+-------------------------+---------------------+------+-----+---------+-------+
```

DEADLOCKS 表中需要用多行来表示同一个死锁事件，每行显示参与死锁的其中一个事务的信息。当该 TiDB 节点记录了多次死锁错误时，需要按照 DEADLOCK_ID 列来区分，相同的 DEADLOCK_ID 表示同一个死锁事件。需要注意，DEADLOCK_ID 并不保证全局唯一，也不会持久化，因而其只能在同一个结果集里表示同一个死锁事件。

DEADLOCKS 表中各列的字段含义如下：

- DEADLOCK_ID：死锁事件的 ID。当表内存在多次死锁错误的信息时，需要使用该列来区分属于不同死锁错误的行。
- OCCUR_TIME：发生该次死锁错误的时间。
- RETRYABLE：该次死锁错误是否可重试。关于可重试的死锁错误的说明，参见可重试的死锁错误小节。
- TRY_LOCK_TRX_ID：试图上锁的事务 ID，即事务的 start_ts。
- CURRENT_SQL_DIGEST：试图上锁的事务中当前正在执行的 SQL 语句的 Digest。
- CURRENT_SQL_DIGEST_TEXT：试图上锁的事务中当前正在执行的 SQL 语句的归一化形式。
- KEY：该事务试图上锁、但是被阻塞的 key，以十六进制编码的形式显示。
- KEY_INFO：对 KEY 进行解读得出的一些详细信息，详见KEY_INFO。。
- TRX_HOLDING_LOCK：该 key 上当前持锁并导致阻塞的事务 ID，即事务的 start_ts。

要调整 DEADLOCKS 表中可以容纳的死锁事件数量，可通过 TiDB 配置文件中的pessimistic-txn.deadlock-history-capacity 配置项进行调整，默认容纳最近 10 次死锁错误的信息。

> 注意：
>
> - 仅拥有 PROCESS 权限的用户可以查询该表。
> - CURRENT_SQL_DIGEST 列中的信息（SQL Digest）为 SQL 语句进行归一化后计算得到的哈希值。CURRENT_SQL_DIGEST_TEXT 列中的信息为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅 Statement Summary Tables。

2.17.2.18.1 KEY_INFO

KEY_INFO 列中展示了对 KEY 列中所给出的 key 的详细信息，以 JSON 格式给出。其包含的信息如下：

- "db_id"：该 key 所属的数据库（schema）的 ID。
- "db_name"：该 key 所属的数据库（schema）的名称。

- "table_id"：该 key 所属的表的 ID。
- "table_name"：该 key 所属的表的名称。
- "partition_id"：该 key 所在的分区（partition）的 ID。
- "partition_name"：该 key 所在的分区（partition）的名称。
- "handle_type"：该 row key（即储存一行数据的 key）的 handle 类型，其可能的值有：
  - "int"：handle 为 int 类型，即 handle 为 row ID
  - "common"：非 int64 类型的 handle，在启用 clustered index 时非 int 类型的主键会显示为此类型
  - "unknown"：当前暂不支持的 handle 类型
- "handle_value"：handle 的值。
- "index_id"：该 index key（即储存索引的 key）所属的 index ID。
- "index_name"：该 index key 所属的 index 名称。
- "index_values"：该 index key 中的 index value。

其中，不适用或当前无法查询到的信息会被省略。比如，row key 的信息中不会包含 index_id、index_name 和 index_values；index key 不会包含 handle_type 和 handle_value；非分区表不会显示 partition_id 和 partition_name；已经被删除掉的表中的 key 的信息无法获取 table_name、db_id、db_name、index_name 等 schema 信息，且无法区分是否为分区表。

> 注意：
>
> 如果一个 key 来自一张启用了分区的表，而在查询时，由于某些原因（例如，其所属的表已经被删除）导致无法查询其所属的 schema 信息，则其所属的分区的 ID 可能会出现在 table_id 字段中。这是因为，TiDB 对不同分区的 key 的编码方式与对几张独立的表的 key 的编码方式一致，因而在缺失 schema 信息时无法确认该 key 属于一张未分区的表还是某张表的一个分区。

2.17.2.18.2 可重试的死锁错误

> 注意：
>
> DEADLOCKS 表中默认不收集可重试的死锁错误的信息。如果需要收集，可通过 TiDB 配置文件中的pessimistic-txn.deadlock-history-collect-retryable 配置项进行调整。

当事务 A 被另一个事务 B 已经持有的锁阻塞，而事务 B 直接或间接地被当前事务 A 持有的锁阻塞，将会引发一个死锁错误。这里：

- 情况一：事务 B 可能（直接或间接地）被事务 A 开始后到被阻塞前这段时间内已经执行完成的语句产生的锁阻塞
- 情况二：事务 B 也可能被事务 A 目前正在执行的语句阻塞

对于情况一，TiDB 将会向事务 A 的客户端报告死锁错误，并终止该事务；而对于情况二，事务 A 当前正在执行的语句将在 TiDB 内部被自动重试。例如，假设事务 A 执行了如下语句：

```
update t set v = v + 1 where id = 1 or id = 2;
```

事务 B 则先后执行如下两条语句：

```
update t set v = 4 where id = 2;
update t set v = 2 where id = 1;
```

那么如果事务 A 先后对 `id = 1` 和 `id = 2` 的两行分别上锁，且两个事务以如下时序运行：

1. 事务 A 对 `id = 1` 的行上锁
2. 事务 B 执行第一条语句并对 `id = 2` 的行上锁
3. 事务 B 执行第二条语句试图对 `id = 1` 的行上锁，被事务 A 阻塞
4. 事务 A 试图对 `id = 2` 的行上锁，被 B 阻塞，形成死锁

对于情况二，由于事务 A 阻塞其它事务的语句也是当前正在执行的语句，因而可以解除当前语句所上的悲观锁（使得事务 B 可以继续运行），并重试当前语句。TiDB 内部使用 key 的哈希值来判断是否属于这种情况。

当可重试的死锁发生时，内部自动重试并不会引起事务报错，因而对客户端透明，但是这种情况的频繁发生可能影响性能。当这种情况发生时，在 TiDB 的日志中可以观察到 `single statement deadlock, retry statement` 字样的日志。

2.17.2.18.3 示例 1

假设有如下表定义和初始数据：

```
create table t (id int primary key, v int);
insert into t values (1, 10), (2, 20);
```

使两个事务按如下顺序执行：

事务 1	事务 2	说明
`update t set v = 11 where id = 1;`		
	`update t set v = 21 where id = 2;`	
`update t set v = 12 where id = 2;`		事务 1 阻塞
	`update t set v = 22 where id = 1;`	事务 2 报出死锁错误

接下来，事务 2 将报出死锁错误。此时，查询 DEADLOCKS 表，将得到如下结果：

```
select * from information_schema.deadlocks;
```

```
+--
    ↪ -----------+----------------------------+-----------+--------------------+-------------------------------
    ↪
| DEADLOCK_ID | OCCUR_TIME                 | RETRYABLE | TRY_LOCK_TRX_ID    | CURRENT_SQL_DIGEST
    ↪                                              | CURRENT_SQL_DIGEST_TEXT                 |
    ↪  KEY                                  | KEY_INFO
```

```
    ↪
    ↪ | TRX_HOLDING_LOCK    |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
```

该表中产生了两行数据，两行的 DEADLOCK_ID 字段皆为 1，表示这两行数据包含同一次死锁错误的信息。第一行显示 ID 为 426812829645406216 的事务，在 "74800000000000000355F728000000000000002" 这个 key 上，被 ID 为 "426812829645406217" 的事务阻塞了；第二行则显示 ID 为 "426812829645406217" 的事务在 "74800000000000000355F728000000000000001" 这个 key 上被 ID 为 426812829645406216 的事务阻塞了，构成了相互阻塞的状态，形成了死锁。

2.17.2.18.4 示例 2

假设查询 DEADLOCKS 表得到了如下结果集：

```
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
| DEADLOCK_ID | OCCUR_TIME                 | RETRYABLE | TRY_LOCK_TRX_ID    | CURRENT_SQL_DIGEST
    ↪                                                | CURRENT_SQL_DIGEST_TEXT                |
    ↪  KEY                                    | KEY_INFO
    ↪
    ↪ | TRX_HOLDING_LOCK    |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
```

```
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809412 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 748000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812832017809413 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809413 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 748000000000000355F728000000000000003 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"3"} | 426812832017809414 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809414 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 748000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812832017809412 |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
```

以上查询结果中的 DEADLOCK_ID 列表明，前两行共同表示一次死锁错误的信息，两条事务相互等待构成了死锁；而后三行共同表示另一次死锁信息，三个事务循环等待构成了死锁。

2.17.2.18.5 CLUSTER_DEADLOCKS

CLUSTER_DEADLOCKS 表返回整个集群上每个 TiDB 节点中最近发生的数次死锁错误的信息，即将每个节点上的 DEADLOCKS 表内的信息合并在一起。CLUSTER_DEADLOCKS 还包含额外的 INSTANCE 列展示所属节点的 IP 地址和端口，用以区分不同的 TiDB 节点。

需要注意的是，由于 DEADLOCK_ID 并不保证全局唯一，所以在 CLUSTER_DEADLOCKS 表的查询结果中，需要 INSTANCE 和 DEADLOCK_ID 两个字段共同区分结果集中的不同死锁错误的信息。

```
USE information_schema;
DESC cluster_deadlocks;
```

```
+-------------------------+---------------------+------+------+---------+-------+
| Field                   | Type                | Null | Key  | Default | Extra |
+-------------------------+---------------------+------+------+---------+-------+
| INSTANCE                | varchar(64)         | YES  |      | NULL    |       |
| DEADLOCK_ID             | bigint(21)          | NO   |      | NULL    |       |
| OCCUR_TIME              | timestamp(6)        | YES  |      | NULL    |       |
| RETRYABLE               | tinyint(1)          | NO   |      | NULL    |       |
| TRY_LOCK_TRX_ID         | bigint(21) unsigned | NO   |      | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)         | YES  |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                | YES  |      | NULL    |       |
| KEY                     | text                | YES  |      | NULL    |       |
| KEY_INFO                | text                | YES  |      | NULL    |       |
| TRX_HOLDING_LOCK        | bigint(21) unsigned | NO   |      | NULL    |       |
+-------------------------+---------------------+------+------+---------+-------+
```

2.17.2.19 ENGINES

ENGINES 表提供了关于存储引擎的信息。从和 MySQL 兼容性上考虑，TiDB 会一直将 InnoDB 描述为唯一支持的引擎。此外，ENGINES 表中其它列值也都是定值。

```
USE information_schema;
DESC engines;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| ENGINE       | varchar(64) | YES  |      | NULL    |       |
| SUPPORT      | varchar(8)  | YES  |      | NULL    |       |
| COMMENT      | varchar(80) | YES  |      | NULL    |       |
| TRANSACTIONS | varchar(3)  | YES  |      | NULL    |       |
| XA           | varchar(3)  | YES  |      | NULL    |       |
| SAVEPOINTS   | varchar(3)  | YES  |      | NULL    |       |
+--------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM engines;
```

```
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
| ENGINE | SUPPORT | COMMENT                                                        | TRANSACTIONS |
    ↪ XA   | SAVEPOINTS |
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES          |
    ↪ YES  | YES        |
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
1 row in set (0.01 sec)
```

ENGINES 表中列的含义如下：

- ENGINE：存储引擎的名称。
- SUPPORT：服务器对存储引擎的支持级别，在 TiDB 中此值一直是 DEFAULT。
- COMMENT：存储引擎的简要描述。
- TRANSACTIONS：存储引擎是否支持事务。
- XA：存储引擎是否支持 XA 事务。
- SAVEPOINTS：存储引擎是否支持 savepoints。

2.17.2.20 INSPECTION_RESULT

TiDB 内置了一些诊断规则，用于检测系统中的故障以及隐患。

该诊断功能可以帮助用户快速发现问题，减少用户的重复性手动工作。可使用 `select * from information_schema.inspection_result` 语句来触发内部诊断。

诊断结果表 `information_schema.inspection_result` 的表结构如下：

```
USE information_schema;
DESC inspection_result;
```

```
+----------------+--------------+------+------+---------+-------+
| Field          | Type         | Null | Key  | Default | Extra |
+----------------+--------------+------+------+---------+-------+
| RULE           | varchar(64)  | YES  |      | NULL    |       |
| ITEM           | varchar(64)  | YES  |      | NULL    |       |
| TYPE           | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE       | varchar(64)  | YES  |      | NULL    |       |
| STATUS_ADDRESS | varchar(64)  | YES  |      | NULL    |       |
| VALUE          | varchar(64)  | YES  |      | NULL    |       |
| REFERENCE      | varchar(64)  | YES  |      | NULL    |       |
| SEVERITY       | varchar(64)  | YES  |      | NULL    |       |
| DETAILS        | varchar(256) | YES  |      | NULL    |       |
+----------------+--------------+------+------+---------+-------+
9 rows in set (0.00 sec)
```

字段解释：

- RULE：诊断规则名称，目前实现了以下规则：
  - config：配置一致性以及合理性检测。如果同一个配置在不同实例不一致，会生成 warning 诊断结果。
  - version：版本一致性检测。如果同一类型的实例版本不同，会生成 critical 诊断结果。
  - node-load：服务器负载检测。如果当前系统负载太高，会生成对应的 warning 诊断结果。
  - critical-error：系统各个模块定义了严重的错误，如果某一个严重错误在对应时间段内超过阈值，会生成 warning 诊断结果。
  - threshold-check：诊断系统会对一些关键指标进行阈值判断，如果超过阈值会生成对应的诊断信息。
- ITEM：每一个规则会对不同的项进行诊断，该字段表示对应规则下面的具体诊断项。
- TYPE：诊断的实例类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：诊断的具体实例地址。
- STATUS_ADDRESS：实例的 HTTP API 服务地址。
- VALUE：针对这个诊断项得到的值。
- REFERENCE：针对这个诊断项的参考值（阈值）。如果 VALUE 超过阈值，就会产生对应的诊断信息。
- SEVERITY：严重程度，取值为 warning 或 critical。
- DETAILS：诊断的详细信息，可能包含进一步调查的 SQL 或文档链接。

2.17.2.20.1 诊断示例

对当前时间的集群进行诊断。

```
SELECT * FROM information_schema.inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE      | config
ITEM      | log.slow-threshold
TYPE      | tidb
INSTANCE  | 172.16.5.40:4000
VALUE     | 0
REFERENCE | not 0
SEVERITY  | warning
DETAILS   | slow-threshold = 0 will record every query to slow log, it may affect performance
***************************[ 2. row ]***************************
RULE      | version
ITEM      | git_hash
TYPE      | tidb
INSTANCE  |
VALUE     | inconsistent
REFERENCE | consistent
SEVERITY  | critical
DETAILS   | the cluster has 2 different tidb version, execute the sql to see more detail: select
    ↪ * from information_schema.cluster_info where type='tidb'
***************************[ 3. row ]***************************
RULE      | threshold-check
ITEM      | storage-write-duration
TYPE      | tikv
INSTANCE  | 172.16.5.40:23151
VALUE     | 130.417
REFERENCE | < 0.100
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:23151 tikv storage-write-duration was too slow
***************************[ 4. row ]***************************
RULE      | threshold-check
ITEM      | rocksdb-write-duration
TYPE      | tikv
INSTANCE  | 172.16.5.40:20151
VALUE     | 108.105
REFERENCE | < 0.100
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:20151 tikv rocksdb-write-duration was too slow
```

上述诊断结果发现了以下几个问题：

- 第一行表示 TiDB 的 `log.slow-threshold` 配置值为 `0`，可能会影响性能。

- 第二行表示集群中有 2 个不同的 TiDB 版本
- 第三、四行表示 TiKV 的写入延迟太大，期望时间是不超过 0.1s, 但实际值远超预期。

诊断集群在时间段“2020-03-26 00:03:00”，“2020-03-26 00:08:00”的问题。指定时间范围需要使用 `/*+ time_range ↪ ()*/` 的 SQL Hint，参考下面的查询示例：

```
select /*+ time_range("2020-03-26 00:03:00", "2020-03-26 00:08:00") */ * from information_schema.
    ↪ inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE      | critical-error
ITEM      | server-down
TYPE      | tidb
INSTANCE  | 172.16.5.40:4009
VALUE     |
REFERENCE |
SEVERITY  | critical
DETAILS   | tidb 172.16.5.40:4009 restarted at time '2020/03/26 00:05:45.670'
***************************[ 2. row ]***************************
RULE      | threshold-check
ITEM      | get-token-duration
TYPE      | tidb
INSTANCE  | 172.16.5.40:10089
VALUE     | 0.234
REFERENCE | < 0.001
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:10089 tidb get-token-duration is too slow
```

上面的诊断结果发现了以下问题：

- 第一行表示 172.16.5.40:4009 TiDB 实例在 2020/03/26 00:05:45.670 发生了重启。
- 第二行表示 172.16.5.40:10089 TiDB 实例的最大的 get-token-duration 时间为 0.234s, 期望时间是小于 0.001s。

也可以指定条件，比如只查询 critical 严重级别的诊断结果：

```
select * from information_schema.inspection_result where severity='critical';
```

只查询 critical-error 规则的诊断结果:

```
select * from information_schema.inspection_result where rule='critical-error';
```

2.17.2.20.2 诊断规则介绍

诊断模块内部包含一系列的规则，这些规则会通过查询已有的监控表和集群信息表，对结果和阈值进行对比。如果结果超过阈值将生成 warning 或 critical 的结果，并在 details 列中提供相应信息。

可以通过查询 inspection_rules 系统表查询已有的诊断规则:

```
select * from information_schema.inspection_rules where type='inspection';
```

```
+-----------------+------------+---------+
| NAME            | TYPE       | COMMENT |
+-----------------+------------+---------+
| config          | inspection |         |
| version         | inspection |         |
| node-load       | inspection |         |
| critical-error  | inspection |         |
| threshold-check | inspection |         |
+-----------------+------------+---------+
```

config 诊断规则

config 诊断规则通过查询 CLUSTER_CONFIG 系统表，执行以下两个诊断规则：

- 检测相同组件的配置值是否一致，并非所有配置项都会有一致性检查，下面是一致性检查的白名单：

```
// TiDB 配置一致性检查白名单
port
status.status-port
host
path
advertise-address
status.status-port
log.file.filename
log.slow-query-file
tmp-storage-path

// PD 配置一致性检查白名单
advertise-client-urls
advertise-peer-urls
client-urls
data-dir
log-file
log.file.filename
metric.job
name
peer-urls

// TiKV 配置一致性检查白名单
server.addr
server.advertise-addr
server.status-addr
log-file
raftstore.raftdb-path
```

```
storage.data-dir
storage.block-cache.capacity
```

- 检测以下配置项的值是否符合预期。

组件	配置项	预期值
TiDB	log.slow-threshold	大于 0
TiKV	raftstore.sync-log	true

version 诊断规则

version 诊断规则通过查询 CLUSTER_INFO 系统表，检测相同组件的版本 hash 是否一致。示例如下：

```
SELECT * FROM information_schema.inspection_result WHERE rule='version'\G
```

```
***************************[ 1. row ]***************************
RULE      | version
ITEM      | git_hash
TYPE      | tidb
INSTANCE  |
VALUE     | inconsistent
REFERENCE | consistent
SEVERITY  | critical
DETAILS   | the cluster has 2 different tidb versions, execute the sql to see more detail: SELECT
    ↪  * FROM information_schema.cluster_info WHERE type='tidb'
```

critical-error 诊断规则

critical-error 诊断规则执行以下两个诊断规则：

- 通过查询metrics schema 数据库中相关的监控系统表，检测集群是否有出现以下比较严重的错误：

组件	错误名字	相关监控表	错误说明
TiDB	panic-count	tidb_panic_total_count	TiDB 出现 panic 错误
TiDB	binlog-error	tidb_binlog_error_total_count	TiDB 写 binlog 时出现的错误

组件	错误名字	相关监控表	错误说明
TiKV	critical-error	tikv_critical_error_total_count	TiKV 的 critical error
TiKV	scheduler-is-busy	tikv_scheduler_is_busy_total_count	TiKV 的 scheduler 太忙，会导致 TiKV 临时不可用
TiKV	coprocessor-is-busy	tikv_coprocessor_is_busy_total_count	TiKV 的 coprocessor 太忙
TiKV	channel-is-full	tikv_channel_full_total_count	TiKV 出现 channel full 的错误
TiKV	tikv_engine_write_stall	tikv_engine_write_stall	TiKV 出现写入 stall 的错误

- 通过查询 `metrics_schema.up` 监控表和 `CLUSTER_LOG` 系统表，检查是否有组件发生重启。

`threshold-check` 诊断规则

`threshold-check` 诊断规则通过查询metrics schema 数据库中相关的监控系统表，检测集群中以下指标是否超出阈值：

组件	监控指标	相关监控表	预期值	说明
TiDB	tso-duration	pd_tso_wait_duration	小于 50 ms	获取事务 TSO 时间戳的等待耗时
TiDB	get-token-duration	tidb_get_token_duration	小于 1 ms	查询获取 token 的耗时，相关的 TiDB 配置参数是 token-limit
TiDB	load-schema-duration	tidb_load_schema_duration	小于 1 s	TiDB 更新表元信息的耗时
TiKV	scheduler-cmd-duration	tikv_scheduler_command_duration	小于 0.1 s	TiKV 执行 KV cmd 请求的耗时
TiKV	handle-snapshot-duration	tikv_handle_snapshot_duration	小于 30 s	TiKV 处理 snap-shot 的耗时
TiKV	storage-write-duration	tikv_storage_async_request_duration	小于 0.1 s	TiKV 写入的延迟

组件	监控指标	相关监控表	预期值	说明
TiKV	storage-snapshot-duration	tikv_storage_async_request_duration	小于 50 ms	TiKV 获取 snapshot 的耗时
TiKV	rocksdb-write-duration	tikv_engine_write_duration	小于 100 ms	TiKV RocksDB 的写入延迟
TiKV	rocksdb-get-duration	tikv_engine_max_get_duration	小于 50 ms	TiKV RocksDB 的读取延迟
TiKV	rocksdb-seek-duration	tikv_engine_max_seek_duration	小于 50 ms	TiKV RocksDB 执行 seek 的延迟
TiKV	scheduler-pending-cmd-coun	tikv_scheduler_pending_commands	小于 1000	TiKV 中被阻塞的命令数量
TiKV	index-block-cache-hit	tikv_block_index_cache_hit	大于 0.95	TiKV 中 index block 缓存的命中率
TiKV	filter-block-cache-hit	tikv_block_filter_cache_hit	大于 0.95	TiKV 中 filter block 缓存的命中率

组件	监控指标	相关监控表	预期值	说明
TiKV	data-block-cache-hit	tikv_block_data_cache_hit	大于 0.80	TiKV 中 data block 缓存的命中率
TiKV	leader-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 leader score 是否均衡，期望实例间的差异小于 5%
TiKV	region-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 Region score 是否均衡，期望实例间的差异小于 5%

组件	监控指标	相关监控表	预期值	说明
TiKV	store-available-balance	pd_scheduler_store_status	小于 0.2	检测各个 TiKV 实例的存储可用空间大小是否均衡，期望实例间的差异小于 20%
TiKV	region-count	pd_scheduler_store_status	小于 20000	检测各个 TiKV 实例的 Region 数量，期望单个实例的 Region 数量小于 20000

组件	监控指标	相关监控表	预期值	说明
PD	region-health	pd_region_health	小于 100	检测集群中处于调度中间状态的 Region 数量，期望总数小于 100

另外还会检测 TiKV 实例的以下 thread cpu usage 是否过高:

- scheduler-worker-cpu
- coprocessor-normal-cpu
- coprocessor-high-cpu
- coprocessor-low-cpu
- grpc-cpu
- raftstore-cpu
- apply-cpu
- storage-readpool-normal-cpu
- storage-readpool-high-cpu
- storage-readpool-low-cpu
- split-check-cpu

TiDB 内置的诊断规则还在不断的完善改进中，如果你也想到了一些诊断规则，非常欢迎在 tidb repository 下提 PR 或 Issue。

2.17.2.21 INSPECTION_RULES

INSPECTION_RULES 表提供在检查结果中运行哪些诊断测试的信息，示例用法参见inspection-result 表。

```
USE information_schema;
DESC inspection_rules;
```

```
+---------+--------------+------+------+---------+-------+
| Field   | Type         | Null | Key  | Default | Extra |
+---------+--------------+------+------+---------+-------+
| NAME    | varchar(64)  | YES  |      | NULL    |       |
```

```
| TYPE    | varchar(64)  | YES  |      | NULL    |       |
| COMMENT | varchar(256) | YES  |      | NULL    |       |
+---------+--------------+------+------+---------+-------+
3 rows in set (0.00 sec)
```

```
SELECT * FROM inspection_rules;
```

```
+-----------------+------------+---------+
| NAME            | TYPE       | COMMENT |
+-----------------+------------+---------+
| config          | inspection |         |
| version         | inspection |         |
| node-load       | inspection |         |
| critical-error  | inspection |         |
| threshold-check | inspection |         |
| ddl             | summary    |         |
| gc              | summary    |         |
| pd              | summary    |         |
| query-summary   | summary    |         |
| raftstore       | summary    |         |
| read-link       | summary    |         |
| rocksdb         | summary    |         |
| stats           | summary    |         |
| wait-events     | summary    |         |
| write-link      | summary    |         |
+-----------------+------------+---------+
15 rows in set (0.00 sec)
```

2.17.2.22 INSPECTION_SUMMARY

在部分场景下，用户只需要关注特定链路或模块的监控汇总。例如当前 Coprocessor 配置的线程池为 8，如果 Coprocessor 的 CPU 使用率达到了 750%，就可以确定存在风险，或者可能提前成为瓶颈。但是部分监控会因为用户的 workload 不同而差异较大，所以难以定义确定的阈值。排查这部分场景的问题也非常重要，所以 TiDB 提供了 `inspection_summary` 来进行链路汇总。

诊断汇总表 `information_schema.inspection_summary` 的表结构如下：

```
USE information_schema;
DESC inspection_summary;
```

```
+--------------+--------------+------+------+---------+-------+
| Field        | Type         | Null | Key  | Default | Extra |
+--------------+--------------+------+------+---------+-------+
| RULE         | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE     | varchar(64)  | YES  |      | NULL    |       |
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |       |
```

```
| LABEL        | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE     | double       | YES  |      | NULL    |       |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |       |
| COMMENT      | varchar(256) | YES  |      | NULL    |       |
+--------------+--------------+------+------+---------+-------+
9 rows in set (0.00 sec)
```

字段解释：

- RULE：汇总规则。由于规则在持续添加，最新的规则列表可以通过 `select * from inspection_rules where type='summary'` 查询。
- INSTANCE：监控的具体实例。
- METRICS_NAME：监控表的名字。
- QUANTILE：对于包含 QUANTILE 的监控表有效，可以通过谓词下推指定多个百分位，例如 `select * from inspection_summary where rule='ddl' and quantile in (0.80, 0.90, 0.99, 0.999)` 来汇总 DDL 相关监控，查询百分位为 80/90/99/999 的结果。AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示聚合的平均值、最小值、最大值。
- COMMENT：对应监控的解释。

> 注意：
>
> 由于汇总所有结果有一定开销，建议在 SQL 的谓词中显示指定的 rule 以减小开销。例如 `select * from inspection_summary where rule in ('read-link', 'ddl')` 会汇总读链路和 DDL 相关的监控。

使用示例:

诊断结果表和诊断监控汇总表都可以通过 hint 的方式指定诊断的时间范围，例如 `select /*+ time_range ('2020-03-07 12:00:00','2020-03-07 13:00:00')*/* from inspection_summary` 是对 2020-03-07 12:00:00 - 2020-03-07 13:00:00 时间段的监控汇总。和监控汇总表一样，inspection_summary 系统表也可以通过对比两个不同时间段的数据，快速发现差异较大的监控项。

以下为一个例子，对比以下两个时间段，读系统链路的监控项:

- (2020-01-16 16:00:54.933, 2020-01-16 16:10:54.933)
- (2020-01-16 16:10:54.933, 2020-01-16 16:20:54.933)

```
SELECT
  t1.avg_value / t2.avg_value AS ratio,
  t1.*,
  t2.*
FROM
```

```
  (
    SELECT
      /*+ time_range("2020-01-16 16:00:54.933", "2020-01-16 16:10:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t1
  JOIN
  (
    SELECT
      /*+ time_range("2020-01-16 16:10:54.933", "2020-01-16 16:20:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t2
  ON t1.metrics_name = t2.metrics_name
  and t1.instance = t2.instance
  and t1.label = t2.label
ORDER BY
  ratio DESC;
```

2.17.2.23 KEY_COLUMN_USAGE

KEY_COLUMN_USAGE 表描述了列的键约束，比如主键约束。

```
USE information_schema;
DESC key_column_usage;
```

```
+------------------------------+--------------+------+------+---------+-------+
| Field                        | Type         | Null | Key  | Default | Extra |
+------------------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG           | varchar(512) | NO   |      | NULL    |       |
| CONSTRAINT_SCHEMA            | varchar(64)  | NO   |      | NULL    |       |
| CONSTRAINT_NAME              | varchar(64)  | NO   |      | NULL    |       |
| TABLE_CATALOG                | varchar(512) | NO   |      | NULL    |       |
| TABLE_SCHEMA                 | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME                   | varchar(64)  | NO   |      | NULL    |       |
| COLUMN_NAME                  | varchar(64)  | NO   |      | NULL    |       |
| ORDINAL_POSITION             | bigint(10)   | NO   |      | NULL    |       |
| POSITION_IN_UNIQUE_CONSTRAINT | bigint(10)   | YES  |      | NULL    |       |
| REFERENCED_TABLE_SCHEMA      | varchar(64)  | YES  |      | NULL    |       |
| REFERENCED_TABLE_NAME        | varchar(64)  | YES  |      | NULL    |       |
| REFERENCED_COLUMN_NAME       | varchar(64)  | YES  |      | NULL    |       |
+------------------------------+--------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM key_column_usage WHERE table_schema='mysql' and table_name='user';
```

```
*************************** 1. row ***************************
```

```
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: Host
             ORDINAL_POSITION: 1
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
        REFERENCED_TABLE_NAME: NULL
       REFERENCED_COLUMN_NAME: NULL
*************************** 2. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: User
             ORDINAL_POSITION: 2
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
        REFERENCED_TABLE_NAME: NULL
       REFERENCED_COLUMN_NAME: NULL
2 rows in set (0.00 sec)
```

KEY_COLUMN_USAGE 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。该值始终为 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束名称。
- TABLE_CATALOG：表所属目录的名称。该值始终为 def。
- TABLE_SCHEMA：表所属的架构数据库的名称。
- TABLE_NAME：具有约束的表的名称。
- COLUMN_NAME：具有约束的列的名称。
- ORDINAL_POSITION：列在约束中的位置，而不是列在表中的位置。列位置从 1 开始编号。
- POSITION_IN_UNIQUE_CONSTRAINT：唯一约束和主键约束为空。对于外键约束，此列是被引用的表的键的序号位置。
- REFERENCED_TABLE_SCHEMA：约束引用的数据库的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_TABLE_NAME：约束引用的表的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_COLUMN_NAME：约束引用的列的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。

2.17.2.24 METRICS_SUMMARY

由于 TiDB 集群的监控指标数量较多，为了方便用户从众多监控中找出异常的监控项，TiDB 4.0 提供了以下监控汇总表：

- information_schema.metrics_summary
- information_schema.metrics_summary_by_label

这两张表用于汇总所有监控数据，用户排查各个监控指标会更有效率。其中 information_schema.metrics_summary_by_label 会对不同的 label 进行区分统计。

```
USE information_schema;
DESC metrics_summary;
```

```
+--------------+--------------+------+------+---------+-------+
| Field        | Type         | Null | Key  | Default | Extra |
+--------------+--------------+------+------+---------+-------+
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE     | double       | YES  |      | NULL    |       |
| SUM_VALUE    | double(22,6) | YES  |      | NULL    |       |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |       |
| COMMENT      | varchar(256) | YES  |      | NULL    |       |
+--------------+--------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段解释：

- METRICS_NAME：监控表名。
- QUANTILE：百分位。可以通过 SQL 语句指定 QUANTILE，例如：
  - select * from metrics_summary where quantile=0.99 指定查看百分位为 0.99 的数据。
  - select * from metrics_summary where quantile in (0.80, 0.90, 0.99, 0.999) 同时查看百分位为 0.80, 0.90, 0.99, 0.999 的数据。
- SUM_VALUE、AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示总和、平均值、最小值、最大值。
- COMMENT：对应监控的解释。

具体查询示例：

查询 '2020-03-08 13:23:00', '2020-03-08 13:33:00' 时间范围内 TiDB 集群中平均耗时最高的三组监控项。可直接查询 information_schema.metrics_summary 表，并通过 /*+ time_range()*/ 这个 hint 来指定时间范围，构造的 SQL 语句如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary
WHERE metrics_name LIKE 'tidb%duration'
```

```
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 3\G
```

```
***************************[ 1. row ]***************************
METRICS_NAME | tidb_get_token_duration
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    ↪ concurrency limit is reached(second)
***************************[ 2. row ]***************************
METRICS_NAME | tidb_query_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.269079
AVG_VALUE    | 0.007272
MIN_VALUE    | 0.000667
MAX_VALUE    | 0.01554
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
METRICS_NAME | tidb_kv_request_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.170232
AVG_VALUE    | 0.004601
MIN_VALUE    | 0.000975
MAX_VALUE    | 0.013
COMMENT      | The quantile of kv requests durations by store
```

类似的，查询 `metrics_summary_by_label` 监控汇总表示例如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary_by_label
WHERE metrics_name LIKE 'tidb%duration'
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 10\G
```

```
***************************[ 1. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_get_token_duration
LABEL        |
QUANTILE     | 0.99
```

```
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    ↪ concurrency limit is reached(second)
***************************[ 2. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Select
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Rollback
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
```

前文提到 `metrics_summary_by_label` 表结构相对于 `metrics_summary` 多了一列 `LABEL`。以上面查询结果的第 2、3 行分别表示 `tidb_query_duration` 的 `Select` 和 `Rollback` 类型的语句平均耗时非常高。

除以上示例之外，监控汇总表可以通过对比两个时间段的全链路监控，迅速找出监控数据中变化最大的模块，快速定位瓶颈。以下示例对比两个时间段的所有监控（其中 t1 为 baseline），并按照差别最大的监控排序：

- 时间段 t1：`("2020-03-03 17:08:00", "2020-03-03 17:11:00")`
- 时间段 t2：`("2020-03-03 17:18:00", "2020-03-03 17:21:00")`

对两个时间段的监控按照 `METRICS_NAME` 进行 join，并按照差异值大小排序。其中 `TIME_RANGE` 是用于指定查询时间的 hint。

```
SELECT GREATEST(t1.avg_value,t2.avg_value)/LEAST(t1.avg_value,
         t2.avg_value) AS ratio,
         t1.metrics_name,
         t1.avg_value as t1_avg_value,
         t2.avg_value as t2_avg_value,
         t2.comment
FROM
```

```
    (SELECT /*+ time_range("2020-03-03 17:08:00", "2020-03-03 17:11:00")*/ *
    FROM information_schema.metrics_summary ) t1
JOIN
    (SELECT /*+ time_range("2020-03-03 17:18:00", "2020-03-03 17:21:00")*/ *
    FROM information_schema.metrics_summary ) t2
    ON t1.metrics_name = t2.metrics_name
ORDER BY ratio DESC LIMIT 10;
```

```
+--
    ↪ -------------+----------------------------------------+---------------+-----------------+----------
    ↪
| ratio          | metrics_name                           | t1_avg_value   | t2_avg_value     |
    ↪  comment
    ↪                                                                                          |
+--
    ↪ -------------+----------------------------------------+---------------+-----------------+----------
    ↪
| 5865.59537065  | tidb_slow_query_cop_process_total_time   |       0.016333 |        95.804724 |
    ↪  The total time of TiDB slow query statistics with slow query total cop process time(
    ↪ second) |
| 3648.74109023  | tidb_distsql_partial_scan_key_total_num  |   10865.666667 |  39646004.4394   |
    ↪  The total num of distsql partial scan key numbers
    ↪                                          |
|  267.002351165 | tidb_slow_query_cop_wait_total_time      |       0.003333 |         0.890008 |
    ↪  The total time of TiDB slow query statistics with slow query total cop wait time(second)
    ↪     |
|  192.43267836  | tikv_cop_total_response_total_size       | 2515333.66667  | 484032394.445    |
    ↪
    ↪
    ↪ |
|  192.43267836  | tikv_cop_total_response_size_per_seconds |   41922.227778 |   8067206.57408  |
    ↪
    ↪
    ↪ |
|  152.780296296 | tidb_distsql_scan_key_total_num          |    5304.333333 |    810397.618317 |
    ↪  The total num of distsql scan numbers
    ↪                                                    |
|  126.042290167 | tidb_distsql_execution_total_time        |       0.421622 |        53.142143 |
    ↪  The total time of distsql execution(second)
    ↪                                                  |
|  105.164020657 | tikv_cop_scan_details                    |     134.450733 |     14139.379665 |
    ↪
    ↪
    ↪ |
|  105.164020657 | tikv_cop_scan_details_total              |    8067.043981 |    848362.77991  |
```

```
    ↪
    ↪
    ↪ |
|  101.635495394 | tikv_cop_scan_keys_num                    |    1070.875    |    108838.91113  |
    ↪
    ↪
    ↪ |
+--
    ↪ --------------+-------------------------------------------+----------------+-----------------+----------
    ↪
```

上面查询结果表示：

- t2 时间段内的 tidb_slow_query_cop_process_total_time（TiDB 慢查询中的 cop process 耗时）比 t1 时间段高了 5865 倍。
- t2 时间段内的 tidb_distsql_partial_scan_key_total_num（TiDB 的 distsql 请求扫描 key 的数量）比 t1 时间段高了 3648 倍。
- t2 时间段内的 tidb_slow_query_cop_wait_total_time（TiDB 慢查询中的 cop 请求排队等待的耗时）比 t1 时间段高了 267 倍。
- t2 时间段内的 tikv_cop_total_response_size（TiKV 的 cop 请求结果的大小）比 t1 时间段高了 192 倍。
- t2 时间段内的 tikv_cop_scan_details（TiKV 的 cop 请求的 scan）比 t1 时间段高了 105 倍。

综上，可以马上知道 t2 时间段的 cop 请求要比 t2 时间段高很多，导致 TiKV 的 Coprocessor 过载，出现了 cop task 等待，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 ~ t2 整个时间段内都在跑 go-ycsb 的压测，然后在 t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了出现很多的 cop 请求。

2.17.2.25 METRICS_TABLES

METRICS_TABLES 表为metrics_schema 数据库中的每个视图提供 PromQL（Prometheus 查询语言）定义。

```
USE information_schema;
DESC metrics_tables;
```

```
+------------+--------------+------+------+---------+-------+
| Field      | Type         | Null | Key  | Default | Extra |
+------------+--------------+------+------+---------+-------+
| TABLE_NAME | varchar(64)  | YES  |      | NULL    |       |
| PROMQL     | varchar(64)  | YES  |      | NULL    |       |
| LABELS     | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE   | double       | YES  |      | NULL    |       |
| COMMENT    | varchar(256) | YES  |      | NULL    |       |
+------------+--------------+------+------+---------+-------+
```

表 metrics_tables 的字段解释：

- TABLE_NAME：对应于 metrics_schema 中的表名。
- PROMQL：监控表的主要原理是将 SQL 映射成 PromQL，并将 Prometheus 结果转换成 SQL 查询结果。这个字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控定义的 label，每一个 label 对应监控表中的一列。SQL 中如果包含对应列的过滤，对应的 PromQL 也会改变。
- QUANTILE：百分位。对于直方图类型的监控数据，指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。
- COMMENT：对这个监控表的注释。

```
SELECT * FROM metrics_tables LIMIT 5\G
```

```
*************************** 1. row ***************************
TABLE_NAME: abnormal_stores
    PROMQL: sum(pd_cluster_status{ type=~"store_disconnected_count|store_unhealth_count|
        ↪ store_low_space_count|store_down_count|store_offline_count|store_tombstone_count"})
    LABELS: instance,type
  QUANTILE: 0
   COMMENT:
*************************** 2. row ***************************
TABLE_NAME: etcd_disk_wal_fsync_rate
    PROMQL: delta(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[$RANGE_DURATION])
    LABELS: instance
  QUANTILE: 0
   COMMENT: The rate of writing WAL into the persistent storage
*************************** 3. row ***************************
TABLE_NAME: etcd_wal_fsync_duration
    PROMQL: histogram_quantile($QUANTILE, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket{
        ↪ $LABEL_CONDITIONS}[$RANGE_DURATION])) by (le,instance))
    LABELS: instance
  QUANTILE: 0.99
   COMMENT: The quantile time consumed of writing WAL into the persistent storage
*************************** 4. row ***************************
TABLE_NAME: etcd_wal_fsync_total_count
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[
        ↪ $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
   COMMENT: The total count of writing WAL into the persistent storage
*************************** 5. row ***************************
TABLE_NAME: etcd_wal_fsync_total_time
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_sum{$LABEL_CONDITIONS}[
        ↪ $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
```

```
   COMMENT: The total time of writing WAL into the persistent storage
5 rows in set (0.00 sec)
```

2.17.2.26 PARTITIONS

PARTITIONS 表提供有关分区表的信息。

```
USE information_schema;
DESC partitions;
```

```
+-------------------------------+--------------+------+------+---------+-------+
| Field                         | Type         | Null | Key  | Default | Extra |
+-------------------------------+--------------+------+------+---------+-------+
| TABLE_CATALOG                 | varchar(512) | YES  |      | NULL    |       |
| TABLE_SCHEMA                  | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME                    | varchar(64)  | YES  |      | NULL    |       |
| PARTITION_NAME                | varchar(64)  | YES  |      | NULL    |       |
| SUBPARTITION_NAME             | varchar(64)  | YES  |      | NULL    |       |
| PARTITION_ORDINAL_POSITION    | bigint(21)   | YES  |      | NULL    |       |
| SUBPARTITION_ORDINAL_POSITION | bigint(21)   | YES  |      | NULL    |       |
| PARTITION_METHOD              | varchar(18)  | YES  |      | NULL    |       |
| SUBPARTITION_METHOD           | varchar(12)  | YES  |      | NULL    |       |
| PARTITION_EXPRESSION          | longblob     | YES  |      | NULL    |       |
| SUBPARTITION_EXPRESSION       | longblob     | YES  |      | NULL    |       |
| PARTITION_DESCRIPTION         | longblob     | YES  |      | NULL    |       |
| TABLE_ROWS                    | bigint(21)   | YES  |      | NULL    |       |
| AVG_ROW_LENGTH                | bigint(21)   | YES  |      | NULL    |       |
| DATA_LENGTH                   | bigint(21)   | YES  |      | NULL    |       |
| MAX_DATA_LENGTH               | bigint(21)   | YES  |      | NULL    |       |
| INDEX_LENGTH                  | bigint(21)   | YES  |      | NULL    |       |
| DATA_FREE                     | bigint(21)   | YES  |      | NULL    |       |
| CREATE_TIME                   | datetime     | YES  |      | NULL    |       |
| UPDATE_TIME                   | datetime     | YES  |      | NULL    |       |
| CHECK_TIME                    | datetime     | YES  |      | NULL    |       |
| CHECKSUM                      | bigint(21)   | YES  |      | NULL    |       |
| PARTITION_COMMENT             | varchar(80)  | YES  |      | NULL    |       |
| NODEGROUP                     | varchar(12)  | YES  |      | NULL    |       |
| TABLESPACE_NAME               | varchar(64)  | YES  |      | NULL    |       |
+-------------------------------+--------------+------+------+---------+-------+
25 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT NOT NULL PRIMARY KEY) PARTITION BY HASH (id) PARTITIONS 2;
SELECT * FROM partitions WHERE table_schema='test' AND table_name='t1'\G
```

```
*************************** 1. row ***************************
```

```
               TABLE_CATALOG: def
                TABLE_SCHEMA: test
                  TABLE_NAME: t1
              PARTITION_NAME: p0
           SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 1
SUBPARTITION_ORDINAL_POSITION: NULL
            PARTITION_METHOD: HASH
         SUBPARTITION_METHOD: NULL
        PARTITION_EXPRESSION: `id`
     SUBPARTITION_EXPRESSION: NULL
       PARTITION_DESCRIPTION:
                  TABLE_ROWS: 0
              AVG_ROW_LENGTH: 0
                 DATA_LENGTH: 0
             MAX_DATA_LENGTH: 0
                INDEX_LENGTH: 0
                   DATA_FREE: 0
                 CREATE_TIME: 2020-07-06 16:35:28
                 UPDATE_TIME: NULL
                  CHECK_TIME: NULL
                    CHECKSUM: NULL
           PARTITION_COMMENT:
                   NODEGROUP: NULL
              TABLESPACE_NAME: NULL
*************************** 2. row ***************************
               TABLE_CATALOG: def
                TABLE_SCHEMA: test
                  TABLE_NAME: t1
              PARTITION_NAME: p1
           SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 2
SUBPARTITION_ORDINAL_POSITION: NULL
            PARTITION_METHOD: HASH
         SUBPARTITION_METHOD: NULL
        PARTITION_EXPRESSION: `id`
     SUBPARTITION_EXPRESSION: NULL
       PARTITION_DESCRIPTION:
                  TABLE_ROWS: 0
              AVG_ROW_LENGTH: 0
                 DATA_LENGTH: 0
             MAX_DATA_LENGTH: 0
                INDEX_LENGTH: 0
                   DATA_FREE: 0
                 CREATE_TIME: 2020-07-06 16:35:28
```

```
                   UPDATE_TIME: NULL
                    CHECK_TIME: NULL
                      CHECKSUM: NULL
              PARTITION_COMMENT:
                     NODEGROUP: NULL
                TABLESPACE_NAME: NULL
2 rows in set (0.00 sec)
```

2.17.2.27 PLACEMENT_POLICIES

PLACEMENT_POLICIES 表展示所有放置策略 (placement policy) 的信息，见Placement Rules in SQL。

```
USE information_schema;
DESC placement_policies;
```

```
+----------------------+---------------+------+-----+---------+-------+
| Field                | Type          | Null | Key | Default | Extra |
+----------------------+---------------+------+-----+---------+-------+
| POLICY_ID            | bigint(64)    | NO   |     | <null>  |       |
| CATALOG_NAME         | varchar(512)  | NO   |     | <null>  |       |
| POLICY_NAME          | varchar(64)   | NO   |     | <null>  |       |
| PRIMARY_REGION       | varchar(1024) | YES  |     | <null>  |       |
| REGIONS              | varchar(1024) | YES  |     | <null>  |       |
| CONSTRAINTS          | varchar(1024) | YES  |     | <null>  |       |
| LEADER_CONSTRAINTS   | varchar(1024) | YES  |     | <null>  |       |
| FOLLOWER_CONSTRAINTS | varchar(1024) | YES  |     | <null>  |       |
| LEARNER_CONSTRAINTS  | varchar(1024) | YES  |     | <null>  |       |
| SCHEDULE             | varchar(20)   | YES  |     | <null>  |       |
| FOLLOWERS            | bigint(64)    | YES  |     | <null>  |       |
| LEARNERS             | bigint(64)    | YES  |     | <null>  |       |
+----------------------+---------------+------+-----+---------+-------+
12 rows in set (0.00 sec)
```

2.17.2.27.1 示例

PLACEMENT_POLICIES 表只展示放置策略 (placement policy) 的信息。如需查看所有信息的规范版本（放置策略，以及绑定放置策略的对象），请改用 SHOW PLACEMENT 语句：

```
CREATE TABLE t1 (a INT);
CREATE PLACEMENT POLICY p1 primary_region="us-east-1" regions="us-east-1";
CREATE TABLE t3 (a INT) PLACEMENT POLICY=p1;
SHOW PLACEMENT; -- 显示所有信息，包含 t3。
SELECT * FROM information_schema.placement_policies; -- 只显示放置策略，不包含 t3。
```

```
Query OK, 0 rows affected (0.09 sec)

Query OK, 0 rows affected (0.11 sec)

Query OK, 0 rows affected (0.08 sec)

+---------------+-----------------------------------------------+------------------+
| Target        | Placement                                     | Scheduling_State |
+---------------+-----------------------------------------------+------------------+
| POLICY p1     | PRIMARY_REGION="us-east-1" REGIONS="us-east-1" | NULL             |
| TABLE test.t3 | PRIMARY_REGION="us-east-1" REGIONS="us-east-1" | SCHEDULED        |
+---------------+-----------------------------------------------+------------------+
2 rows in set (0.00 sec)

+--
    ↪ ---------+--------------+-------------+---------------+-----------+-------------+----------------------+--
    ↪
| POLICY_ID | CATALOG_NAME | POLICY_NAME | PRIMARY_REGION | REGIONS    | CONSTRAINTS |
    ↪ LEADER_CONSTRAINTS | FOLLOWER_CONSTRAINTS | LEARNER_CONSTRAINTS | SCHEDULE | FOLLOWERS |
    ↪ LEARNERS |
+--
    ↪ ---------+--------------+-------------+---------------+-----------+-------------+----------------------+--
    ↪
| 1         | def          | p1          | us-east-1      | us-east-1 |             |
    ↪                     |                       |                     |           | 2         |
    ↪ 0         |
+--
    ↪ ---------+--------------+-------------+---------------+-----------+-------------+----------------------+--
    ↪
1 rows in set (0.00 sec)
```

2.17.2.28 PROCESSLIST

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

PROCESSLIST 表比 SHOW PROCESSLIST 的结果多出下面几列：

- MEM 列：MEM 显示正在处理的请求已使用的内存，单位是 byte。
- TxnStart列：显示事务的开始时间

```
USE information_schema;
DESC processlist;
```

```
+----------+---------------------+------+------+----------+--------+
```

```
| Field    | Type                | Null | Key  | Default | Extra |
+----------+---------------------+------+------+---------+-------+
| ID       | bigint(21) unsigned | NO   |      | 0       |       |
| USER     | varchar(16)         | NO   |      |         |       |
| HOST     | varchar(64)         | NO   |      |         |       |
| DB       | varchar(64)         | YES  |      | NULL    |       |
| COMMAND  | varchar(16)         | NO   |      |         |       |
| TIME     | int(7)              | NO   |      | 0       |       |
| STATE    | varchar(7)          | YES  |      | NULL    |       |
| INFO     | binary(512)         | YES  |      | NULL    |       |
| MEM      | bigint(21) unsigned | YES  |      | NULL    |       |
| TxnStart | varchar(64)         | NO   |      |         |       |
+----------+---------------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

```
SELECT * FROM processlist\G
```

```
*************************** 1. row ***************************
      ID: 16
    USER: root
    HOST: 127.0.0.1
      DB: information_schema
 COMMAND: Query
    TIME: 0
   STATE: autocommit
    INFO: SELECT * FROM processlist
     MEM: 0
TxnStart:
1 row in set (0.00 sec)
```

PROCESSLIST 表各列的含义如下：

- ID：客户连接 ID。
- USER：执行当前 PROCESS 的用户名。
- HOST：客户连接的地址。
- DB：当前连接的默认数据库名。
- COMMAND：当前 PROCESS 执行的命令类型。
- TIME：当前 PROCESS 的已经执行的时间，单位是秒。
- STATE：当前连接的状态。
- INFO：正在处理的请求语句。
- MEM：正在处理的请求已使用的内存，单位是 byte。
- TxnStart列：显示事务的开始时间

2.17.2.28.1 CLUSTER_PROCESSLIST

CLUSTER_PROCESSLIST 是 PROCESSLIST 对应的集群系统表，用于查询集群中所有 TiDB 节点的 PROCESSLIST 信息。CLUSTER_PROCESSLIST 表结构上比 PROCESSLIST 多一列 INSTANCE，表示该行数据来自的 TiDB 节点地址。

```
SELECT * FROM cluster_processlist;
```

```
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    ↪
| INSTANCE         | ID  | USER | HOST     | DB   | COMMAND | TIME | STATE      | INFO
    ↪                                                  | MEM | TxnStart
    ↪                                |
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    ↪
| 10.0.1.22:10080 | 150 | u1   | 10.0.1.1 | test | Query   | 0    | autocommit | select count(*)
    ↪ from usertable                        | 372 | 05-28 03:54:21.230(416976223923077223) |
| 10.0.1.22:10080 | 138 | root | 10.0.1.1 | test | Query   | 0    | autocommit | SELECT * FROM
    ↪ information_schema.cluster_processlist | 0   | 05-28 03:54:21.230(416976223923077220) |
| 10.0.1.22:10080 | 151 | u1   | 10.0.1.1 | test | Query   | 0    | autocommit | select count(*)
    ↪ from usertable                        | 372 | 05-28 03:54:21.230(416976223923077224) |
| 10.0.1.21:10080 | 15  | u2   | 10.0.1.1 | test | Query   | 0    | autocommit | select max(
    ↪ field0) from usertable                     | 496 | 05-28 03:54:21.230(416976223923077222)
    ↪ |
| 10.0.1.21:10080 | 14  | u2   | 10.0.1.1 | test | Query   | 0    | autocommit | select max(
    ↪ field0) from usertable                     | 496 | 05-28 03:54:21.230(416976223923077225)
    ↪ |
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    ↪
```

2.17.2.29 REFERENTIAL_CONSTRAINTS

REFERENTIAL_CONSTRAINTS 表提供 TiDB 表之间 FOREIGN KEY 关系的信息。请注意，目前 TiDB 不强制进行 FOREIGN
↪ KEY 约束，也不执行 ON DELETE CASCADE 等操作。

```
USE information_schema;
DESC referential_constraints;
```

```
+--------------------------+--------------+------+------+---------+-------+
| Field                    | Type         | Null | Key  | Default | Extra |
+--------------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG       | varchar(512) | NO   |      | NULL    |       |
| CONSTRAINT_SCHEMA        | varchar(64)  | NO   |      | NULL    |       |
| CONSTRAINT_NAME          | varchar(64)  | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_CATALOG | varchar(512) | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_SCHEMA | varchar(64)  | NO   |      | NULL    |       |
```

```
| UNIQUE_CONSTRAINT_NAME    | varchar(64)  | YES  |     | NULL    |       |
| MATCH_OPTION              | varchar(64)  | NO   |     | NULL    |       |
| UPDATE_RULE               | varchar(64)  | NO   |     | NULL    |       |
| DELETE_RULE               | varchar(64)  | NO   |     | NULL    |       |
| TABLE_NAME                | varchar(64)  | NO   |     | NULL    |       |
| REFERENCED_TABLE_NAME     | varchar(64)  | NO   |     | NULL    |       |
+---------------------------+--------------+------+-----+---------+-------+
11 rows in set (0.00 sec)
```

```
CREATE TABLE test.parent (
 id INT NOT NULL AUTO_INCREMENT,
 PRIMARY KEY (id)
);

CREATE TABLE test.child (
 id INT NOT NULL AUTO_INCREMENT,
 name varchar(255) NOT NULL,
 parent_id INT DEFAULT NULL,
 PRIMARY KEY (id),
 CONSTRAINT fk_parent FOREIGN KEY (parent_id) REFERENCES parent (id) ON UPDATE CASCADE ON DELETE
     ↪ RESTRICT
);

SELECT * FROM referential_constraints\G
```

```
*************************** 1. row ***************************
       CONSTRAINT_CATALOG: def
        CONSTRAINT_SCHEMA: test
          CONSTRAINT_NAME: fk_parent
UNIQUE_CONSTRAINT_CATALOG: def
 UNIQUE_CONSTRAINT_SCHEMA: test
   UNIQUE_CONSTRAINT_NAME: PRIMARY
             MATCH_OPTION: NONE
              UPDATE_RULE: CASCADE
              DELETE_RULE: RESTRICT
               TABLE_NAME: child
    REFERENCED_TABLE_NAME: parent
1 row in set (0.00 sec)
```

2.17.2.30 SCHEMATA

SCHEMATA 表提供了关于数据库的信息。表中的数据与 SHOW DATABASES 语句的执行结果等价。

```
USE information_schema;
desc SCHEMATA;
```

```
+---------------------------+--------------+------+------+---------+-------+
| Field                     | Type         | Null | Key  | Default | Extra |
+---------------------------+--------------+------+------+---------+-------+
| CATALOG_NAME              | varchar(512) | YES  |      | NULL    |       |
| SCHEMA_NAME               | varchar(64)  | YES  |      | NULL    |       |
| DEFAULT_CHARACTER_SET_NAME | varchar(64)  | YES  |      | NULL    |       |
| DEFAULT_COLLATION_NAME    | varchar(32)  | YES  |      | NULL    |       |
| SQL_PATH                  | varchar(512) | YES  |      | NULL    |       |
+---------------------------+--------------+------+------+---------+-------+
5 rows in set (0.00 sec)
```

```
SELECT * FROM SCHEMATA;
```

```
+--
    ↪ ------------+--------------------+----------------------------+------------------------+----------+
    ↪
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME |
    ↪ SQL_PATH |
+--
    ↪ ------------+--------------------+----------------------------+------------------------+----------+
    ↪
| def          | INFORMATION_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | METRICS_SCHEMA     | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | mysql              | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | PERFORMANCE_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | test               | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
+--
    ↪ ------------+--------------------+----------------------------+------------------------+----------+
    ↪
5 rows in set (0.00 sec)
```

SCHEMATA 表各列字段含义如下：

- CATALOG_NAME：数据库归属的目录名，该列值永远为 def。
- SCHEMA_NAME：数据库的名字。
- DEFAULT_CHARACTER_SET_NAME：数据库的默认字符集。
- DEFAULT_COLLATION_NAME：数据库的默认 collation。
- SQL_PATH：该项值永远为 NULL。

2.17.2.31 SEQUENCES

SEQUENCES 表提供了有关序列的信息。TiDB 中序列的功能是参照 MariaDB 中的类似功能来实现的。

```
USE information_schema;
DESC sequences;
```

```
+-----------------+--------------+------+------+---------+-------+
| Field           | Type         | Null | Key  | Default | Extra |
+-----------------+--------------+------+------+---------+-------+
| TABLE_CATALOG   | varchar(512) | NO   |      | NULL    |       |
| SEQUENCE_SCHEMA | varchar(64)  | NO   |      | NULL    |       |
| SEQUENCE_NAME   | varchar(64)  | NO   |      | NULL    |       |
| CACHE           | tinyint(4)   | NO   |      | NULL    |       |
| CACHE_VALUE     | bigint(21)   | YES  |      | NULL    |       |
| CYCLE           | tinyint(4)   | NO   |      | NULL    |       |
| INCREMENT       | bigint(21)   | NO   |      | NULL    |       |
| MAX_VALUE       | bigint(21)   | YES  |      | NULL    |       |
| MIN_VALUE       | bigint(21)   | YES  |      | NULL    |       |
| START           | bigint(21)   | YES  |      | NULL    |       |
| COMMENT         | varchar(64)  | YES  |      | NULL    |       |
+-----------------+--------------+------+------+---------+-------+
11 rows in set (0.00 sec)
```

```
CREATE SEQUENCE test.seq;
SELECT nextval(test.seq);
SELECT * FROM sequences\G
```

```
+-------------------+
| nextval(test.seq) |
+-------------------+
|                 1 |
+-------------------+
1 row in set (0.01 sec)
*************************** 1. row ***************************
  TABLE_CATALOG: def
SEQUENCE_SCHEMA: test
  SEQUENCE_NAME: seq
          CACHE: 1
    CACHE_VALUE: 1000
          CYCLE: 0
      INCREMENT: 1
      MAX_VALUE: 9223372036854775806
      MIN_VALUE: 1
          START: 1
        COMMENT:
```

```
1 row in set (0.00 sec)
```

2.17.2.32 SESSION_VARIABLES

SESSION_VARIABLES 表提供了关于 session 变量的信息。表中的数据跟 SHOW SESSION VARIABLES 语句执行结果类似。

```
USE information_schema;
DESC session_variables;
```

```
+----------------+---------------+------+------+---------+-------+
| Field          | Type          | Null | Key  | Default | Extra |
+----------------+---------------+------+------+---------+-------+
| VARIABLE_NAME  | varchar(64)   | YES  |      | NULL    |       |
| VARIABLE_VALUE | varchar(1024) | YES  |      | NULL    |       |
+----------------+---------------+------+------+---------+-------+
2 rows in set (0.00 sec)
```

```
SELECT * FROM session_variables ORDER BY variable_name LIMIT 10;
```

```
+-----------------------------------+------------------+
| VARIABLE_NAME                     | VARIABLE_VALUE   |
+-----------------------------------+------------------+
| allow_auto_random_explicit_insert | off              |
| auto_increment_increment          | 1                |
| auto_increment_offset             | 1                |
| autocommit                        | 1                |
| automatic_sp_privileges           | 1                |
| avoid_temporal_upgrade            | 0                |
| back_log                          | 80               |
| basedir                           | /usr/local/mysql |
| big_tables                        | 0                |
| bind_address                      | *                |
+-----------------------------------+------------------+
10 rows in set (0.00 sec)
```

SESSION_VARIABLES 表各列字段含义如下：

- VARIABLE_NAME：数据库中 session 级变量的名称。
- VARIABLE_VALUE：数据库中对应该 session 变量名的具体值。

2.17.2.33 SLOW_QUERY

SLOW_QUERY 表中提供了当前节点的慢查询相关的信息，其内容通过解析当前节点的 TiDB 慢查询日志而来，列名和慢日志中的字段名是一一对应。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
USE information_schema;
DESC slow_query;
```

```
+---------------------------+---------------------+------+------+---------+-------+
| Field                     | Type                | Null | Key  | Default | Extra |
+---------------------------+---------------------+------+------+---------+-------+
| Time                      | timestamp(6)        | YES  |      | NULL    |       |
| Txn_start_ts              | bigint(20) unsigned | YES  |      | NULL    |       |
| User                      | varchar(64)         | YES  |      | NULL    |       |
| Host                      | varchar(64)         | YES  |      | NULL    |       |
| Conn_ID                   | bigint(20) unsigned | YES  |      | NULL    |       |
| Query_time                | double              | YES  |      | NULL    |       |
| Parse_time                | double              | YES  |      | NULL    |       |
| Compile_time              | double              | YES  |      | NULL    |       |
| Rewrite_time              | double              | YES  |      | NULL    |       |
| Preproc_subqueries        | bigint(20) unsigned | YES  |      | NULL    |       |
| Preproc_subqueries_time   | double              | YES  |      | NULL    |       |
| Optimize_time             | double              | YES  |      | NULL    |       |
| Wait_TS                   | double              | YES  |      | NULL    |       |
| Prewrite_time             | double              | YES  |      | NULL    |       |
| Wait_prewrite_binlog_time | double              | YES  |      | NULL    |       |
| Commit_time               | double              | YES  |      | NULL    |       |
| Get_commit_ts_time        | double              | YES  |      | NULL    |       |
| Commit_backoff_time       | double              | YES  |      | NULL    |       |
| Backoff_types             | varchar(64)         | YES  |      | NULL    |       |
| Resolve_lock_time         | double              | YES  |      | NULL    |       |
| Local_latch_wait_time     | double              | YES  |      | NULL    |       |
| Write_keys                | bigint(22)          | YES  |      | NULL    |       |
| Write_size                | bigint(22)          | YES  |      | NULL    |       |
| Prewrite_region           | bigint(22)          | YES  |      | NULL    |       |
| Txn_retry                 | bigint(22)          | YES  |      | NULL    |       |
| Cop_time                  | double              | YES  |      | NULL    |       |
| Process_time              | double              | YES  |      | NULL    |       |
| Wait_time                 | double              | YES  |      | NULL    |       |
| Backoff_time              | double              | YES  |      | NULL    |       |
| LockKeys_time             | double              | YES  |      | NULL    |       |
| Request_count             | bigint(20) unsigned | YES  |      | NULL    |       |
| Total_keys                | bigint(20) unsigned | YES  |      | NULL    |       |
| Process_keys              | bigint(20) unsigned | YES  |      | NULL    |       |
| DB                        | varchar(64)         | YES  |      | NULL    |       |
| Index_names               | varchar(100)        | YES  |      | NULL    |       |
| Is_internal               | tinyint(1)          | YES  |      | NULL    |       |
| Digest                    | varchar(64)         | YES  |      | NULL    |       |
| Stats                     | varchar(512)        | YES  |      | NULL    |       |
```

```
| Cop_proc_avg              | double              | YES  |      | NULL    |       |
| Cop_proc_p90              | double              | YES  |      | NULL    |       |
| Cop_proc_max              | double              | YES  |      | NULL    |       |
| Cop_proc_addr             | varchar(64)         | YES  |      | NULL    |       |
| Cop_wait_avg              | double              | YES  |      | NULL    |       |
| Cop_wait_p90              | double              | YES  |      | NULL    |       |
| Cop_wait_max              | double              | YES  |      | NULL    |       |
| Cop_wait_addr             | varchar(64)         | YES  |      | NULL    |       |
| Mem_max                   | bigint(20)          | YES  |      | NULL    |       |
| Disk_max                  | bigint(20)          | YES  |      | NULL    |       |
| Succ                      | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache           | tinyint(1)          | YES  |      | NULL    |       |
| Plan                      | longblob            | YES  |      | NULL    |       |
| Plan_digest               | varchar(128)        | YES  |      | NULL    |       |
| Prev_stmt                 | longblob            | YES  |      | NULL    |       |
| Query                     | longblob            | YES  |      | NULL    |       |
+---------------------------+---------------------+------+------+---------+-------+
54 rows in set (0.00 sec)
```

2.17.2.33.1 CLUSTER_SLOW_QUERY table

CLUSTER_SLOW_QUERY 表中提供了集群所有节点的慢查询相关的信息，其内容通过解析 TiDB 慢查询日志而来，该表使用上和 SLOW_QUERY 表一样。CLUSTER_SLOW_QUERY 表结构上比 SLOW_QUERY 多一列 INSTANCE，表示该行慢查询信息来自的 TiDB 节点地址。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
desc cluster_slow_query;
```

```
+---------------------------+---------------------+------+------+---------+-------+
| Field                     | Type                | Null | Key  | Default | Extra |
+---------------------------+---------------------+------+------+---------+-------+
| INSTANCE                  | varchar(64)         | YES  |      | NULL    |       |
| Time                      | timestamp(6)        | YES  |      | NULL    |       |
| Txn_start_ts              | bigint(20) unsigned | YES  |      | NULL    |       |
| User                      | varchar(64)         | YES  |      | NULL    |       |
| Host                      | varchar(64)         | YES  |      | NULL    |       |
| Conn_ID                   | bigint(20) unsigned | YES  |      | NULL    |       |
| Query_time                | double              | YES  |      | NULL    |       |
| Parse_time                | double              | YES  |      | NULL    |       |
| Compile_time              | double              | YES  |      | NULL    |       |
| Rewrite_time              | double              | YES  |      | NULL    |       |
| Preproc_subqueries        | bigint(20) unsigned | YES  |      | NULL    |       |
| Preproc_subqueries_time   | double              | YES  |      | NULL    |       |
| Optimize_time             | double              | YES  |      | NULL    |       |
| Wait_TS                   | double              | YES  |      | NULL    |       |
| Prewrite_time             | double              | YES  |      | NULL    |       |
```

```
| Wait_prewrite_binlog_time | double              | YES  |      | NULL    |       |
| Commit_time               | double              | YES  |      | NULL    |       |
| Get_commit_ts_time        | double              | YES  |      | NULL    |       |
| Commit_backoff_time       | double              | YES  |      | NULL    |       |
| Backoff_types             | varchar(64)         | YES  |      | NULL    |       |
| Resolve_lock_time         | double              | YES  |      | NULL    |       |
| Local_latch_wait_time     | double              | YES  |      | NULL    |       |
| Write_keys                | bigint(22)          | YES  |      | NULL    |       |
| Write_size                | bigint(22)          | YES  |      | NULL    |       |
| Prewrite_region           | bigint(22)          | YES  |      | NULL    |       |
| Txn_retry                 | bigint(22)          | YES  |      | NULL    |       |
| Cop_time                  | double              | YES  |      | NULL    |       |
| Process_time              | double              | YES  |      | NULL    |       |
| Wait_time                 | double              | YES  |      | NULL    |       |
| Backoff_time              | double              | YES  |      | NULL    |       |
| LockKeys_time             | double              | YES  |      | NULL    |       |
| Request_count             | bigint(20) unsigned | YES  |      | NULL    |       |
| Total_keys                | bigint(20) unsigned | YES  |      | NULL    |       |
| Process_keys              | bigint(20) unsigned | YES  |      | NULL    |       |
| DB                        | varchar(64)         | YES  |      | NULL    |       |
| Index_names               | varchar(100)        | YES  |      | NULL    |       |
| Is_internal               | tinyint(1)          | YES  |      | NULL    |       |
| Digest                    | varchar(64)         | YES  |      | NULL    |       |
| Stats                     | varchar(512)        | YES  |      | NULL    |       |
| Cop_proc_avg              | double              | YES  |      | NULL    |       |
| Cop_proc_p90              | double              | YES  |      | NULL    |       |
| Cop_proc_max              | double              | YES  |      | NULL    |       |
| Cop_proc_addr             | varchar(64)         | YES  |      | NULL    |       |
| Cop_wait_avg              | double              | YES  |      | NULL    |       |
| Cop_wait_p90              | double              | YES  |      | NULL    |       |
| Cop_wait_max              | double              | YES  |      | NULL    |       |
| Cop_wait_addr             | varchar(64)         | YES  |      | NULL    |       |
| Mem_max                   | bigint(20)          | YES  |      | NULL    |       |
| Disk_max                  | bigint(20)          | YES  |      | NULL    |       |
| Succ                      | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache           | tinyint(1)          | YES  |      | NULL    |       |
| Plan                      | longblob            | YES  |      | NULL    |       |
| Plan_digest               | varchar(128)        | YES  |      | NULL    |       |
| Prev_stmt                 | longblob            | YES  |      | NULL    |       |
| Query                     | longblob            | YES  |      | NULL    |       |
+---------------------------+---------------------+------+------+---------+-------+
55 rows in set (0.00 sec)
```

查询集群系统表时，TiDB 也会将相关计算下推给其他节点执行，而不是把所有节点的数据都取回来，可以查看执行计划，如下：

```
desc SELECT count(*) FROM cluster_slow_query WHERE user = 'u1';
```

```
+--
    ↪ ----------------------+----------+-----------+-------------------------+-------------------------------
    ↪
| id                        | estRows  | task      | access object           | operator info
    ↪                                        |
+--
    ↪ ----------------------+----------+-----------+-------------------------+-------------------------------
    ↪
| StreamAgg_20              | 1.00     | root      |                         | funcs:count(Column
    ↪ #53)->Column#51                      |
| └─TableReader_21          | 1.00     | root      |                         | data:StreamAgg_9
    ↪                                        |
|   └─StreamAgg_9           | 1.00     | cop[tidb] |                         | funcs:count(1)->
    ↪ Column#53                              |
|     └─Selection_19        | 10.00    | cop[tidb] |                         | eq(
    ↪ information_schema.cluster_slow_query.user, "u1") |
|       └─TableFullScan_18 | 10000.00 | cop[tidb] | table:CLUSTER_SLOW_QUERY | keep order:false,
    ↪  stats:pseudo                          |
+--
    ↪ ----------------------+----------+-----------+-------------------------+-------------------------------
    ↪
```

上面执行计划表示，会将 `user = u1` 条件下推给其他的 (cop) TiDB 节点执行，也会把聚合算子（即图中的 StreamAgg 算子）下推。

目前由于没有对系统表收集统计信息，所以有时会导致某些聚合算子不能下推，导致执行较慢，用户可以通过手动指定聚合下推的 SQL HINT 来将聚合算子下推，示例如下：

```
SELECT /*+ AGG_TO_COP() */ count(*) FROM cluster_slow_query GROUP BY user;
```

2.17.2.34 STATISTICS

STATISTICS 表提供了关于表索引的信息。

```
USE information_schema;
DESC statistics;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| TABLE_CATALOG | varchar(512)  | YES  |      | NULL    |       |
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |       |
| NON_UNIQUE    | varchar(1)    | YES  |      | NULL    |       |
```

```
| INDEX_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| INDEX_NAME    | varchar(64)   | YES  |      | NULL    |       |
| SEQ_IN_INDEX  | bigint(2)     | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(21)   | YES  |      | NULL    |       |
| COLLATION     | varchar(1)    | YES  |      | NULL    |       |
| CARDINALITY   | bigint(21)    | YES  |      | NULL    |       |
| SUB_PART      | bigint(3)     | YES  |      | NULL    |       |
| PACKED        | varchar(10)   | YES  |      | NULL    |       |
| NULLABLE      | varchar(3)    | YES  |      | NULL    |       |
| INDEX_TYPE    | varchar(16)   | YES  |      | NULL    |       |
| COMMENT       | varchar(16)   | YES  |      | NULL    |       |
| INDEX_COMMENT | varchar(1024) | YES  |      | NULL    |       |
| IS_VISIBLE    | varchar(3)    | YES  |      | NULL    |       |
| Expression    | varchar(64)   | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
18 rows in set (0.00 sec)
```

STATISTICS 表中列的含义如下：

- TABLE_CATALOG：包含索引的表所属的目录的名称。这个值总是 def。
- TABLE_SCHEMA：包含索引的表所属的数据库的名称。
- TABLE_NAME：包含索引的表的名称。
- NON_UNIQUE：如果索引不能包含重复项，则为 0；如果可以，则为 1。
- INDEX_SCHEMA：索引所属的数据库的名称。
- INDEX_NAME：索引的名称。如果索引是主键，那么名称总是 PRIMARY。
- SEQ_IN_INDEX：索引中的列序号，从 1 开始。
- COLUMN_NAME：列名。请参见表达式列的说明。
- COLLATION：列在索引中的排序方式。取值可以是 A（升序）、D（降序）或 NULL（未排序）。
- CARDINALITY：TiDB 未使用该字段。该字段的值总是 0。
- SUB_PART：索引的前缀。如果只对列的部分前缀进行索引，则为索引字符的数量；如果对整个列进行索引，则为 NULL。
- PACKED：TiDB 未使用该字段。这个值总是 NULL。
- NULLABLE：如果列可能包含 NULL 值，则值为 YES；如果不包含，则值为 ''。
- INDEX_TYPE：索引的类型。
- COMMENT：其他与索引有关的信息。
- INDEX_COMMENT：在创建索引时为索引提供的带有注释属性的任何注释。
- IS_VISIBLE：优化器能否使用该索引。
- Expression 对于非表达式部分的索引键，这个值为 NULL；对于表达式部分的索引键，这个值为表达式本身。可参考表达式索引

下列语句是等价的：

```
SELECT * FROM INFORMATION_SCHEMA.STATISTICS
  WHERE table_name = 'tbl_name'
  AND table_schema = 'db_name'
```

```
SHOW INDEX
  FROM tbl_name
  FROM db_name
```

2.17.2.35 TABLES

TABLES 表提供了数据库里关于表的信息。

```
USE information_schema;
DESC tables;
```

```
+---------------------------+---------------+------+------+----------+-------+
| Field                     | Type          | Null | Key  | Default  | Extra |
+---------------------------+---------------+------+------+----------+-------+
| TABLE_CATALOG             | varchar(512)  | YES  |      | NULL     |       |
| TABLE_SCHEMA              | varchar(64)   | YES  |      | NULL     |       |
| TABLE_NAME                | varchar(64)   | YES  |      | NULL     |       |
| TABLE_TYPE                | varchar(64)   | YES  |      | NULL     |       |
| ENGINE                    | varchar(64)   | YES  |      | NULL     |       |
| VERSION                   | bigint(21)    | YES  |      | NULL     |       |
| ROW_FORMAT                | varchar(10)   | YES  |      | NULL     |       |
| TABLE_ROWS                | bigint(21)    | YES  |      | NULL     |       |
| AVG_ROW_LENGTH            | bigint(21)    | YES  |      | NULL     |       |
| DATA_LENGTH               | bigint(21)    | YES  |      | NULL     |       |
| MAX_DATA_LENGTH           | bigint(21)    | YES  |      | NULL     |       |
| INDEX_LENGTH              | bigint(21)    | YES  |      | NULL     |       |
| DATA_FREE                 | bigint(21)    | YES  |      | NULL     |       |
| AUTO_INCREMENT            | bigint(21)    | YES  |      | NULL     |       |
| CREATE_TIME               | datetime      | YES  |      | NULL     |       |
| UPDATE_TIME               | datetime      | YES  |      | NULL     |       |
| CHECK_TIME                | datetime      | YES  |      | NULL     |       |
| TABLE_COLLATION           | varchar(32)   | NO   |      | utf8_bin |       |
| CHECKSUM                  | bigint(21)    | YES  |      | NULL     |       |
| CREATE_OPTIONS            | varchar(255)  | YES  |      | NULL     |       |
| TABLE_COMMENT             | varchar(2048) | YES  |      | NULL     |       |
| TIDB_TABLE_ID             | bigint(21)    | YES  |      | NULL     |       |
| TIDB_ROW_ID_SHARDING_INFO | varchar(255)  | YES  |      | NULL     |       |
+---------------------------+---------------+------+------+----------+-------+
23 rows in set (0.00 sec)
```

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

```
*************************** 1. row ***************************
            TABLE_CATALOG: def
             TABLE_SCHEMA: mysql
```

```
               TABLE_NAME: user
               TABLE_TYPE: BASE TABLE
                   ENGINE: InnoDB
                  VERSION: 10
               ROW_FORMAT: Compact
               TABLE_ROWS: 0
           AVG_ROW_LENGTH: 0
              DATA_LENGTH: 0
          MAX_DATA_LENGTH: 0
             INDEX_LENGTH: 0
                DATA_FREE: 0
           AUTO_INCREMENT: NULL
              CREATE_TIME: 2020-07-05 09:25:51
              UPDATE_TIME: NULL
               CHECK_TIME: NULL
          TABLE_COLLATION: utf8mb4_bin
                 CHECKSUM: NULL
           CREATE_OPTIONS:
            TABLE_COMMENT:
            TIDB_TABLE_ID: 5
TIDB_ROW_ID_SHARDING_INFO: NULL
1 row in set (0.00 sec)
```

下列语句是等价的：

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

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

TABLES 表各列字段含义如下：

- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：表所属数据库的名称。
- TABLE_NAME：表的名称。
- TABLE_TYPE：表的类型。
- ENGINE：存储引擎类型。该值暂为 ‘InnoDB’。
- VERSION：版本，默认值为 10。
- ROW_FORMAT：行格式。该值暂为 ‘Compact’。
- TABLE_ROWS：统计信息中该表所存的行数。
- AVG_ROW_LENGTH：该表中所存数据的平均行长度。平均行长度 = DATA_LENGTH / 统计信息中的行数。
- DATA_LENGTH：数据长度。数据长度 = 统计信息中的行数 × 元组各列存储长度和，这里尚未考虑 TiKV 的副本数。

- MAX_DATA_LENGTH：最大数据长度。该值暂为 0，表示没有最大数据长度的限制。
- INDEX_LENGTH：索引长度。索引长度 = 统计信息中的行数 × 索引元组各列长度和，这里尚未考虑 TiKV 的副本数。
- DATA_FREE：空间碎片。该值暂为 0。
- AUTO_INCREMENT：该表中自增主键自动增量的当前值。
- CREATE_TIME：该表的创建时间。
- UPDATE_TIME：该表的更新时间。
- CHECK_TIME：该表的检查时间。
- TABLE_COLLATION：该表的字符校验编码集。
- CHECKSUM：校验和。
- CREATE_OPTIONS：创建选项。
- TABLE_COMMENT：表的注释、备注。

表中的信息大部分定义自 MySQL，只有两列是 TiDB 新增的：

- TIDB_TABLE_ID：标识表的内部 ID，该 ID 在一个 TiDB 集群内部唯一。
- TIDB_ROW_ID_SHARDING_INFO：标识表的 Sharding 类型，可能的值为：
  - "NOT_SHARDED"：表未被 Shard。
  - "NOT_SHARDED(PK_IS_HANDLE)"：一个定义了整型主键的表未被 Shard。
  - "PK_AUTO_RANDOM_BITS={bit_number}"：一个定义了整型主键的表由于定义了 AUTO_RANDOM 而被 Shard。
  - "SHARD_BITS={bit_number}"：表使用 SHARD_ROW_ID_BITS={bit_number} 进行了 Shard。
  - NULL：表属于系统表或 View，无法被 Shard。

2.17.2.36 TABLE_CONSTRAINTS

TABLE_CONSTRAINTS 表记录了表的约束信息。

```
USE information_schema;
DESC table_constraints;
```

```
+--------------------+--------------+------+------+---------+-------+
| Field              | Type         | Null | Key  | Default | Extra |
+--------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG | varchar(512) | YES  |      | NULL    |       |
| CONSTRAINT_SCHEMA  | varchar(64)  | YES  |      | NULL    |       |
| CONSTRAINT_NAME    | varchar(64)  | YES  |      | NULL    |       |
| TABLE_SCHEMA       | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME         | varchar(64)  | YES  |      | NULL    |       |
| CONSTRAINT_TYPE    | varchar(64)  | YES  |      | NULL    |       |
+--------------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

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

```
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
| CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA   | CONSTRAINT_NAME         | TABLE_SCHEMA       |
    ↪ TABLE_NAME                           | CONSTRAINT_TYPE |
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
| def                | mysql               | name                    | mysql              |
    ↪ help_topic                           | UNIQUE          |
| def                | mysql               | tbl                     | mysql              |
    ↪ stats_meta                           | UNIQUE          |
| def                | mysql               | tbl                     | mysql              |
    ↪ stats_histograms                     | UNIQUE          |
| def                | mysql               | tbl                     | mysql              |
    ↪ stats_buckets                        | UNIQUE          |
| def                | mysql               | delete_range_index      | mysql              |
    ↪ gc_delete_range                      | UNIQUE          |
| def                | mysql               | delete_range_done_index | mysql              |
    ↪ gc_delete_range_done                 | UNIQUE          |
| def                | PERFORMANCE_SCHEMA  | SCHEMA_NAME             | PERFORMANCE_SCHEMA |
    ↪ events_statements_summary_by_digest  | UNIQUE          |
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
7 rows in set (0.01 sec)
```

TABLE_CONSTRAINTS 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。这个值总是 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束的名称。
- TABLE_NAME：表的名称。
- CONSTRAINT_TYPE：约束的类型。取值可以是 UNIQUE、PRIMARY KEY 或者 FOREIGN KEY。UNIQUE 和 PRIMARY KEY 信息与 SHOW INDEX 语句的执行结果类似。

2.17.2.37 TABLE_STORAGE_STATS

TABLE_STORAGE_STATS 表提供有关由存储引擎 (TiKV) 存储的表大小的信息。

```
USE information_schema;
DESC table_storage_stats;
```

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

```
+--------------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA       | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME         | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID           | bigint(21)  | YES  |      | NULL    |       |
| PEER_COUNT         | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT       | bigint(21)  | YES  |      | NULL    |       |
| EMPTY_REGION_COUNT | bigint(21)  | YES  |      | NULL    |       |
| TABLE_SIZE         | bigint(64)  | YES  |      | NULL    |       |
| TABLE_KEYS         | bigint(64)  | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
8 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT);
INSERT INTO test.t1 VALUES (1);
SELECT * FROM table_storage_stats WHERE table_schema = 'test' AND table_name = 't1'\G
```

```
*************************** 1. row ***************************
      TABLE_SCHEMA: test
        TABLE_NAME: t1
          TABLE_ID: 56
        PEER_COUNT: 1
      REGION_COUNT: 1
EMPTY_REGION_COUNT: 1
        TABLE_SIZE: 1
        TABLE_KEYS: 0
1 row in set (0.00 sec)
```

2.17.2.38 TIDB_HOT_REGIONS

TIDB_HOT_REGIONS 表提供了关于当前热点 Region 的相关信息。历史热点信息可以在TIDB_HOT_REGIONS_HISTORY 表查看。

```
USE information_schema;
DESC tidb_hot_regions;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TABLE_ID       | bigint(21)  | YES  |      | NULL    |       |
| INDEX_ID       | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME        | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME     | varchar(64) | YES  |      | NULL    |       |
| INDEX_NAME     | varchar(64) | YES  |      | NULL    |       |
| REGION_ID      | bigint(21)  | YES  |      | NULL    |       |
| TYPE           | varchar(64) | YES  |      | NULL    |       |
```

```
| MAX_HOT_DEGREE | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT   | bigint(21)  | YES  |      | NULL    |       |
| FLOW_BYTES     | bigint(21)  | YES  |      | NULL    |       |
+----------------+-------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS 表各列字段含义如下：

- TABLE_ID：热点 Region 所在表的 ID。
- INDEX_ID：热点 Region 所在索引的 ID。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_NAME：热点 Region 所在表的名称。
- INDEX_NAME：热点 Region 所在索引的名称。
- REGION_ID：热点 Region 的 ID。
- TYPE：热点 Region 的类型。
- MAX_HOT_DEGREE：该 Region 的最大热度。
- REGION_COUNT：所在实例的热点 Region 数量。
- FLOW_BYTES：该 Region 内读写的字节数量。

2.17.2.39 TIDB_HOT_REGIONS_HISTORY

TIDB_HOT_REGIONS_HISTORY 表提供了关于历史热点 Region 的相关信息，这些信息由 PD 定期存储在本地，存储间隔为hot-regions-write-interval 的值（默认值为 10m），历史热点信息保持的期限为hot-regions-reserved-days 的值（默认值为 7），详情参见PD 配置文件描述。

```
USE information_schema;
DESC tidb_hot_regions_history;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| UPDATE_TIME | timestamp(6) | YES  |      | NULL    |       |
| DB_NAME     | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME  | varchar(64)  | YES  |      | NULL    |       |
| TABLE_ID    | bigint(21)   | YES  |      | NULL    |       |
| INDEX_NAME  | varchar(64)  | YES  |      | NULL    |       |
| INDEX_ID    | bigint(21)   | YES  |      | NULL    |       |
| REGION_ID   | bigint(21)   | YES  |      | NULL    |       |
| STORE_ID    | bigint(21)   | YES  |      | NULL    |       |
| PEER_ID     | bigint(21)   | YES  |      | NULL    |       |
| IS_LEARNER  | tinyint(1)   | NO   |      | 0       |       |
| IS_LEADER   | tinyint(1)   | NO   |      | 0       |       |
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| HOT_DEGREE  | bigint(21)   | YES  |      | NULL    |       |
| FLOW_BYTES  | double       | YES  |      | NULL    |       |
| KEY_RATE    | double       | YES  |      | NULL    |       |
```

```
| QUERY_RATE   | double        | YES  |      | NULL    |       |
+--------------+---------------+------+------+---------+-------+
16 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS_HISTORY 表各列字段含义如下：

- UPDATE_TIME：热点 Region 更新时间。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_ID：热点 Region 所在表的 ID。
- TABLE_NAME：热点 Region 所在表的名称。
- INDEX_NAME：热点 Region 所在索引的名称。
- INDEX_ID：热点 Region 所在索引的 ID。
- REGION_ID：热点 Region 的 ID。
- STORE_ID：热点 Region 所在 TiKV Store 的 ID。
- PEER_ID：热点 Region 对应的副本 Peer 的 ID。
- IS_LEARNER：PEER 是否是 LEARNER。
- IS_LEADER：PEER 是否是 LEADER。
- TYPE：热点 Region 的类型。
- HOT_DEGREE：该热点 Region 的热度。
- FLOW_BYTES：该 Region 内读写的字节数量。
- KEY_RATE：该 Region 内读写的 Key 数量。
- QUERY_RATE：该 Region 内读写的 Query 数量。

> 注意：
>
> TIDB_HOT_REGIONS_HISTORY 表的 UPDATE_TIME、REGION_ID、STORE_ID、PEER_ID、IS_LEARNER、IS_LEADER、TYPE 字段会下推到 PD 服务器执行。所以为了降低使用该表的开销，必须指定搜索时间范围，然后尽可能地指定更多的条件。例如 `select * from tidb_hot_regions_history where store_id = 11 and update_time > '2020-05-18 20:40:00' and update_time < '2020-05-18 21:40:00' and type='write'`。

2.17.2.39.1 常见使用场景

- 查询一段时间内的所有热点 Regions。将以下语句中的 update_time 替换为实际所需的时间即可。

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18
    -> 21:40:00' and update_time <'2021-09-19 00:00:00';
```

> 注意：
>
> UPDATE_TIME 同样支持 Unix 时间戳。例如：update_time >TIMESTAMP('2012-08-18 21:40:00') 或 update_time > FROM_UNIXTIME(1629294000.000)。

- 查询某张表指定时间内的热点 Regions，替换 update_time、 table_name 即可。

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18
    ↪ 21:40:00' and update_time <'2021-09-19 00:00:00' and TABLE_NAME = 'table_name';
```

- 查询某张表指定时间内热点 Regions 的分布，替换 update_time、 table_name 即可。

```
SELECT count(region_id) cnt, store_id FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE
    ↪  update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and
    ↪ table_name = 'table_name' GROUP BY STORE_ID ORDER BY cnt DESC;
```

- 查询某张表指定时间内热点 Leader Regions 的分布，替换 update_time、 table_name 即可。

```
SELECT count(region_id) cnt, store_id FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE
    ↪  update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and
    ↪ table_name = 'table_name' and is_leader=1 GROUP BY STORE_ID ORDER BY cnt DESC;
```

- 查询某张表指定时间内热点 Index Regions 的分布，替换 update_time、 table_name 即可。

```
SELECT count(region_id) cnt, index_name, store_id FROM INFORMATION_SCHEMA.
    ↪ TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'
    ↪ 2021-09-19 00:00:00' and table_name = 'table_name' group by index_name, store_id
    ↪ order by index_name,cnt desc;
```

- 查询某张表指定时间内热点 Index Leader Regions 的分布，替换 update_time、 table_name 即可。

```
SELECT count(region_id) cnt, index_name, store_id FROM INFORMATION_SCHEMA.
    ↪ TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'
    ↪ 2022-09-19 00:00:00' and table_name = 'table_name' and is_leader=1 group by
    ↪ index_name, store_id order by index_name,cnt desc;
```

2.17.2.40 TIDB_INDEXES

TIDB_INDEXES 记录了所有表中的 INDEX 信息。

```
USE information_schema;
DESC tidb_indexes;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |       |
| NON_UNIQUE    | bigint(21)    | YES  |      | NULL    |       |
| KEY_NAME      | varchar(64)   | YES  |      | NULL    |       |
| SEQ_IN_INDEX  | bigint(21)    | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(64)   | YES  |      | NULL    |       |
```

```
| SUB_PART      | bigint(21)    | YES  |      | NULL    |       |
| INDEX_COMMENT | varchar(2048) | YES  |      | NULL    |       |
| Expression    | varchar(64)   | YES  |      | NULL    |       |
| INDEX_ID      | bigint(21)    | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

INDEX_ID 是 TiDB 为每个索引分配的唯一 ID。它可以与从另一个表或 API 获得的 INDEX_ID 一起执行 join 操作。

例如，你可以在SLOW_QUERY 表 中获取某些慢查询所涉及的 TABLE_ID 和 INDEX_ID，然后使用以下 SQL 语句获取特定索引信息：

```
SELECT
 tidb_indexes.*
FROM
 tidb_indexes,
 tables
WHERE
  tidb_indexes.table_schema = tables.table_schema
 AND tidb_indexes.table_name = tidb_indexes.table_name
 AND tables.tidb_table_id = ?
 AND index_id = ?
```

TIDB_INDEXES 表中列的含义如下：

- TABLE_SCHEMA：索引所在表的所属数据库的名称。
- TABLE_NAME：索引所在表的名称。
- NON_UNIQUE：如果索引是唯一的，则为 0，否则为 1。
- KEY_NAME：索引的名称。如果索引是主键，则名称为 PRIMARY。
- SEQ_IN_INDEX：索引中列的顺序编号，从 1 开始。
- COLUMN_NAME：索引所在的列名。
- SUB_PART：索引前缀长度。如果列是部分被索引，则该值为被索引的字符数量，否则为 NULL。
- INDEX_COMMENT：创建索引时以 COMMENT 标注的注释。
- INDEX_ID：索引的 ID。

2.17.2.41 TIDB_SERVERS_INFO

TIDB_SERVERS_INFO 表提供了 TiDB 集群中 TiDB 服务器的信息（即 tidb-server 进程）。

```
USE information_schema;
DESC tidb_servers_info;
```

```
+---------------+-------------+------+------+---------+-------+
| Field         | Type        | Null | Key  | Default | Extra |
+---------------+-------------+------+------+---------+-------+
| DDL_ID        | varchar(64) | YES  |      | NULL    |       |
| IP            | varchar(64) | YES  |      | NULL    |       |
```

```
| PORT          | bigint(21)  | YES  |      | NULL    |       |
| STATUS_PORT   | bigint(21)  | YES  |      | NULL    |       |
| LEASE         | varchar(64) | YES  |      | NULL    |       |
| VERSION       | varchar(64) | YES  |      | NULL    |       |
| GIT_HASH      | varchar(64) | YES  |      | NULL    |       |
| BINLOG_STATUS | varchar(64) | YES  |      | NULL    |       |
+---------------+-------------+------+------+---------+-------+
8 rows in set (0.00 sec)
```

```
SELECT * FROM tidb_servers_info\G
```

```
*************************** 1. row ***************************
       DDL_ID: 771c169d-0a3a-48ea-a93c-a4d6751d3674
           IP: 0.0.0.0
         PORT: 4000
  STATUS_PORT: 10080
        LEASE: 45s
      VERSION: 5.7.25-TiDB-v4.0.0-beta.2-720-g0df3b74f5
     GIT_HASH: 0df3b74f55f8f8fbde39bbd5d471783f49dc10f7
BINLOG_STATUS: Off
1 row in set (0.00 sec)
```

2.17.2.42 TIDB_TRX

TIDB_TRX 表提供了当前 TiDB 节点上正在执行的事务的信息。

```
USE information_schema;
DESC tidb_trx;
```

```
+--
    ↪ ----------------------+----------------------------------------------------------------------+------+------+-
    ↪
| Field                  | Type                                                                 |
    ↪ Null | Key  | Default | Extra |
+--
    ↪ ----------------------+----------------------------------------------------------------------+------+------+-
    ↪
| ID                     | bigint(21) unsigned                                                  | NO
    ↪   | PRI  | NULL    |       |
| START_TIME             | timestamp(6)                                                         | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST     | varchar(64)                                                          | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                                                                | YES
    ↪   |      | NULL    |       |
```

```
| STATE                 | enum('Idle','Running','LockWaiting','Committing','RollingBack') | YES
    ↪   |       | NULL    |       |
| WAITING_START_TIME    | timestamp(6)                                                    | YES
    ↪   |       | NULL    |       |
| MEM_BUFFER_KEYS       | bigint(64)                                                      | YES
    ↪   |       | NULL    |       |
| MEM_BUFFER_BYTES      | bigint(64)                                                      | YES
    ↪   |       | NULL    |       |
| SESSION_ID            | bigint(21) unsigned                                             | YES
    ↪   |       | NULL    |       |
| USER                  | varchar(16)                                                     | YES
    ↪   |       | NULL    |       |
| DB                    | varchar(64)                                                     | YES
    ↪   |       | NULL    |       |
| ALL_SQL_DIGESTS       | text                                                            | YES
    ↪   |       | NULL    |       |
+--
    ↪ ----------------------+-----------------------------------------------------------------+------+------+-
    ↪
```

TIDB_TRX 表中各列的字段含义如下：

- ID：事务 ID，即事务的开始时间戳 start_ts。
- START_TIME：事务的开始时间，即事务的 start_ts 所对应的物理时间。
- CURRENT_SQL_DIGEST：该事务当前正在执行的 SQL 语句的 Digest。
- CURRENT_SQL_DIGEST_TEXT：该事务当前正在执行的 SQL 语句的归一化形式，即去除了参数和格式的 SQL 语句。与 CURRENT_SQL_DIGEST 对应。
- STATE：该事务当前所处的状态。其可能的值包括:
  - Idle：事务处于闲置状态，即正在等待用户输入查询。
  - Running：事务正在正常执行一个查询。
  - LockWaiting：事务处于正在等待悲观锁上锁完成的状态。需要注意的是，事务刚开始进行悲观锁上锁操作时即进入该状态，无论是否有被其它事务阻塞。
  - Committing：事务正在提交过程中。
  - RollingBack：事务正在回滚过程中。
- WAITING_START_TIME：当 STATE 值为 LockWaiting 时，该列显示等待的开始时间。
- MEM_BUFFER_KEYS：当前事务写入内存缓冲区的 key 的个数。
- MEM_BUFFER_BYTES：当前事务写入内存缓冲区的 key 和 value 的总字节数。
- SESSION_ID：该事务所属的 session 的 ID。
- USER：执行该事务的用户名。
- DB：执行该事务的 session 当前的默认数据库名。
- ALL_SQL_DIGESTS：该事务已经执行过的语句的 Digest 的列表，表示为一个 JSON 格式的字符串数组。每个事务最多记录前 50 条语句。通过TIDB_DECODE_SQL_DIGESTS 函数可以将该列的信息变换为对应的归一化 SQL 语句的列表。

注意：

- 仅拥有 PROCESS 权限的用户可以获取该表中的完整信息。没有 PROCESS 权限的用户则只能查询到当前用户所执行的事务的信息。
- CURRENT_SQL_DIGEST 和 ALL_SQL_DIGESTS 列中的信息（SQL Digest）为 SQL 语句进行归一化后计算得到的哈希值。CURRENT_SQL_DIGEST_TEXT 列中的信息和函数 TIDB_DECODE_SQL_DIGESTS 所得到的结果均为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅 Statement Summary Tables。
- TIDB_DECODE_SQL_DIGESTS 函数调用开销较大，如果对大量事务的信息调用该函数查询历史 SQL，可能查询耗时较长。如果集群规模较大、同一时刻并发运行的事务较多，请避免直接在查询 TIDB_TRX 全表的同时直接将该函数用于 ALL_SQL_DIGEST 列（即尽量避免 `select *, tidb_decode_sql_digests(all_sql_digests)from tidb_trx` 这样的用法）。
- 目前 TIDB_TRX 表暂不支持显示 TiDB 内部事务相关的信息。

2.17.2.42.1 示例

```
select * from information_schema.tidb_trx\G
```

```
*************************** 1. row ***************************
                   ID: 426789913200689153
           START_TIME: 2021-08-04 10:51:54.883000
    CURRENT_SQL_DIGEST: NULL
CURRENT_SQL_DIGEST_TEXT: NULL
                STATE: Idle
    WAITING_START_TIME: NULL
       MEM_BUFFER_KEYS: 1
      MEM_BUFFER_BYTES: 29
            SESSION_ID: 7
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","04
           ↪ fa858fa491c62d194faec2ab427261cc7998b3f1ccf8f6844febca504cb5e9","
           ↪ b83710fa8ab7df8504920e8569e48654f621cf828afbe7527fd003b79f48da9e"]
*************************** 2. row ***************************
                   ID: 426789921471332353
           START_TIME: 2021-08-04 10:52:26.433000
    CURRENT_SQL_DIGEST: 38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ?
                STATE: LockWaiting
    WAITING_START_TIME: 2021-08-04 10:52:35.106568
       MEM_BUFFER_KEYS: 0
```

```
       MEM_BUFFER_BYTES: 0
            SESSION_ID: 9
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
           ↪ b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821"]
2 rows in set (0.01 sec)
```

此示例的查询结果表示：当前节点有两个运行中的事务，第一个事务正在闲置状态（STATE 为 Idle ↪ ，CURRENT_SQL_DIGEST 为 NULL），该事务已经执行过 3 条语句（ALL_SQL_DIGESTS 列表中有三条记录，分别为执行过的 3 条语句的 Digest）；第二个事务正在执行一条语句并正在等锁（STATE 为 LockWaiting，WAITING_START_TIME 显示了等锁开始的时间），该事务已经执行过两条语句，当前正在执行的语句形如 "update `t` set `v` = `v` + ? where `id` = ?"。

```
select id, all_sql_digests, tidb_decode_sql_digests(all_sql_digests) as all_sqls from
    ↪ information_schema.tidb_trx\G
```

```
*************************** 1. row ***************************
             id: 426789913200689153
all_sql_digests: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","04
    ↪ fa858fa491c62d194faec2ab427261cc7998b3f1ccf8f6844febca504cb5e9","
    ↪ b83710fa8ab7df8504920e8569e48654f621cf828afbe7527fd003b79f48da9e"]
       all_sqls: ["begin","insert into `t` values ( ... )","update `t` set `v` = `v` + ?"]
*************************** 2. row ***************************
             id: 426789921471332353
all_sql_digests: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
    ↪ b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821"]
       all_sqls: ["begin","update `t` set `v` = `v` + ? where `id` = ?"]
```

此查询对 TIDB_TRX 表的 ALL_SQL_DIGESTS 列调用了TIDB_DECODE_SQL_DIGESTS 函数，将 SQL Digest 的数组通过系统内部的查询转换成归一化 SQL 语句的数组，以便于直观地获取事务历史执行过的语句的信息。但是需要注意上述查询扫描了 TIDB_TRX 全表，并对每一行都调用了 TIDB_DECODE_SQL_DIGESTS 函数；而 TIDB_DECODE_SQL_DIGESTS 函数调用的开销很大，所以如果集群中并发事务数量较多，请尽量避免这种查询。

2.17.2.42.2 CLUSTER_TIDB_TRX

TIDB_TRX 表仅提供单个 TiDB 节点中正在执行的事务信息。如果要查看整个集群上所有 TiDB 节点中正在执行的事务信息，需要查询 CLUSTER_TIDB_TRX 表。与 TIDB_TRX 表的查询结果相比，CLUSTER_TIDB_TRX 表的查询结果额外包含了 INSTANCE 字段。INSTANCE 字段展示了集群中各节点的 IP 地址和端口，用于区分事务所在的 TiDB 节点。

```
USE information_schema;
DESC cluster_tidb_trx;
```

```
mysql> desc cluster_tidb_trx;
```

```
+--
    ↪ ----------------------+------------------------------------------------------------------------+------+------+-
    ↪
| Field                   | Type                                                                   |
    ↪ Null | Key  | Default | Extra |
+--
    ↪ ----------------------+------------------------------------------------------------------------+------+------+-
    ↪
| INSTANCE                | varchar(64)                                                            | YES
    ↪   |      | NULL    |       |
| ID                      | bigint(21) unsigned                                                    | NO
    ↪   | PRI  | NULL    |       |
| START_TIME              | timestamp(6)                                                           | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)                                                            | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                                                                   | YES
    ↪   |      | NULL    |       |
| STATE                   | enum('Idle','Running','LockWaiting','Committing','RollingBack')        | YES
    ↪   |      | NULL    |       |
| WAITING_START_TIME      | timestamp(6)                                                           | YES
    ↪   |      | NULL    |       |
| MEM_BUFFER_KEYS         | bigint(64)                                                             | YES
    ↪   |      | NULL    |       |
| MEM_BUFFER_BYTES        | bigint(64)                                                             | YES
    ↪   |      | NULL    |       |
| SESSION_ID              | bigint(21) unsigned                                                    | YES
    ↪   |      | NULL    |       |
| USER                    | varchar(16)                                                            | YES
    ↪   |      | NULL    |       |
| DB                      | varchar(64)                                                            | YES
    ↪   |      | NULL    |       |
| ALL_SQL_DIGESTS         | text                                                                   | YES
    ↪   |      | NULL    |       |
+--
    ↪ ----------------------+------------------------------------------------------------------------+------+------+-
    ↪
```

2.17.2.43 TIFLASH_REPLICA

TIFLASH_REPLICA 表提供了有关可用的 TiFlash 副本的信息。

```
USE information_schema;
DESC tiflash_replica;
```

```
+-----------------+-------------+------+------+---------+-------+
| Field           | Type        | Null | Key  | Default | Extra |
+-----------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA    | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID        | bigint(21)  | YES  |      | NULL    |       |
| REPLICA_COUNT   | bigint(64)  | YES  |      | NULL    |       |
| LOCATION_LABELS | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE       | tinyint(1)  | YES  |      | NULL    |       |
| PROGRESS        | double      | YES  |      | NULL    |       |
+-----------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

2.17.2.44 TIKV_REGION_PEERS

TIKV_REGION_PEERS 表提供了 TiKV 中单个 Region 节点的详细信息，比如它是一个 learner 还是一个 leader。

```
USE information_schema;
DESC tikv_region_peers;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| REGION_ID    | bigint(21)  | YES  |      | NULL    |       |
| PEER_ID      | bigint(21)  | YES  |      | NULL    |       |
| STORE_ID     | bigint(21)  | YES  |      | NULL    |       |
| IS_LEARNER   | tinyint(1)  | NO   |      | 0       |       |
| IS_LEADER    | tinyint(1)  | NO   |      | 0       |       |
| STATUS       | varchar(10) | YES  |      | 0       |       |
| DOWN_SECONDS | bigint(21)  | YES  |      | 0       |       |
+--------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

例如，使用以下 SQL 语句，你可以查询 WRITTEN_BYTES 最大的前 3 个 Region 所在的 TiKV 地址：

```
SELECT
 address,
 tikv.address,
 region.region_id
FROM
 tikv_store_status tikv,
 tikv_region_peers peer,
 (SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3) region
WHERE
  region.region_id = peer.region_id
```

```
AND peer.is_leader = 1
AND peer.store_id = tikv.store_id;
```

TIKV_REGION_PEERS 表各列含义如下：

- REGION_ID：REGION 的 ID。
- PEER_ID：REGION 中对应的副本 PEER 的 ID。
- STORE_ID：REGION 所在 TiKV Store 的 ID。
- IS_LEARNER：PEER 是否是 LEARNER。
- IS_LEADER：PEER 是否是 LEADER。
- STATUS：PEER 的状态，一共有 3 种状态：
  - PENDING：暂时不可用状态。
  - DOWN：下线转态，该 PEER 不再提供服务。
  - NORMAL: 正常状态。
- DOWN_SECONDS：处于下线状态的时间，单位是秒。

2.17.2.45 TIKV_REGION_STATUS

TIKV_REGION_STATUS 表通过 PD 的 API 展示 TiKV Region 的基本信息，比如 Region ID、开始和结束键值以及读写流量。

```
USE information_schema;
DESC tikv_region_status;
```

```
+--------------------------+-------------+------+------+---------+-------+
| Field                    | Type        | Null | Key  | Default | Extra |
+--------------------------+-------------+------+------+---------+-------+
| REGION_ID                | bigint(21)  | YES  |      | NULL    |       |
| START_KEY                | text        | YES  |      | NULL    |       |
| END_KEY                  | text        | YES  |      | NULL    |       |
| TABLE_ID                 | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME                  | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME               | varchar(64) | YES  |      | NULL    |       |
| IS_INDEX                 | tinyint(1)  | NO   |      | 0       |       |
| INDEX_ID                 | bigint(21)  | YES  |      | NULL    |       |
| INDEX_NAME               | varchar(64) | YES  |      | NULL    |       |
| EPOCH_CONF_VER           | bigint(21)  | YES  |      | NULL    |       |
| EPOCH_VERSION            | bigint(21)  | YES  |      | NULL    |       |
| WRITTEN_BYTES            | bigint(21)  | YES  |      | NULL    |       |
| READ_BYTES               | bigint(21)  | YES  |      | NULL    |       |
| APPROXIMATE_SIZE         | bigint(21)  | YES  |      | NULL    |       |
| APPROXIMATE_KEYS         | bigint(21)  | YES  |      | NULL    |       |
| REPLICATIONSTATUS_STATE  | varchar(64) | YES  |      | NULL    |       |
| REPLICATIONSTATUS_STATEID| bigint(21)  | YES  |      | NULL    |       |
+--------------------------+-------------+------+------+---------+-------+
```

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

TIKV_REGION_STATUS 表中列的含义如下：

- REGION_ID：Region 的 ID。
- START_KEY：Region 的起始 key 的值。
- END_KEY：Region 的末尾 key 的值。
- TABLE_ID：Region 所属的表的 ID。
- DB_NAME：TABLE_ID 所属的数据库的名称。
- TABLE_NAME：Region 所属的表的名称。
- IS_INDEX：Region 数据是否是索引，0 代表不是索引，1 代表是索引。如果当前 Region 同时包含表数据和索引数据，会有多行记录，IS_INDEX 分别是 0 和 1。
- INDEX_ID：Region 所属的索引的 ID。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- INDEX_NAME：Region 所属的索引的名称。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- EPOCH_CONF_VER：Region 的配置的版本号，在增加或减少 peer 时版本号会递增。
- EPOCH_VERSION：Region 的当前版本号，在分裂或合并时版本号会递增。
- WRITTEN_BYTES：已经往 Region 写入的数据量 (bytes)。
- READ_BYTES：已经从 Region 读取的数据量 (bytes)。
- APPROXIMATE_SIZE：Region 的近似数据量 (MB)。
- APPROXIMATE_KEYS：Region 中 key 的近似数量。
- REPLICATIONSTATUS_STATE:Region 当前的同步状态,可能为 UNKNOWN/SIMPLE_MAJORITY/INTEGRITY_OVER_LABEL ↪ 其中一种状态。
- REPLICATIONSTATUS_STATEID：REPLICATIONSTATUS_STATE 对应的标识符。

通过在 EPOCH_CONF_VER、WRITTEN_BYTES 和 READ_BYTES 列上执行 ORDER BY X LIMIT Y 操作，你可以在 pd-ctl 中实现 top confver、top read 和 top write 操作。

你可以使用以下 SQL 语句查询写入数据最多的前 3 个 Region：

```
SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3;
```

2.17.2.46 TIKV_STORE_STATUS

TIKV_STORE_STATUS 表通过 PD 的 API 显示了 TiKV 节点的一些基本信息，例如在集群中分配的 ID，地址和端口，状态，容量以及当前节点的 Region leader 的数量。

```
USE information_schema;
DESC tikv_store_status;
```

```
+-------------------+-------------+------+------+---------+-------+
| Field             | Type        | Null | Key  | Default | Extra |
+-------------------+-------------+------+------+---------+-------+
| STORE_ID          | bigint(21)  | YES  |      | NULL    |       |
| ADDRESS           | varchar(64) | YES  |      | NULL    |       |
| STORE_STATE       | bigint(21)  | YES  |      | NULL    |       |
| STORE_STATE_NAME  | varchar(64) | YES  |      | NULL    |       |
```

```
| LABEL             | json        | YES  |      | NULL    |       |
| VERSION           | varchar(64) | YES  |      | NULL    |       |
| CAPACITY          | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE         | varchar(64) | YES  |      | NULL    |       |
| LEADER_COUNT      | bigint(21)  | YES  |      | NULL    |       |
| LEADER_WEIGHT     | double      | YES  |      | NULL    |       |
| LEADER_SCORE      | double      | YES  |      | NULL    |       |
| LEADER_SIZE       | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT      | bigint(21)  | YES  |      | NULL    |       |
| REGION_WEIGHT     | double      | YES  |      | NULL    |       |
| REGION_SCORE      | double      | YES  |      | NULL    |       |
| REGION_SIZE       | bigint(21)  | YES  |      | NULL    |       |
| START_TS          | datetime    | YES  |      | NULL    |       |
| LAST_HEARTBEAT_TS | datetime    | YES  |      | NULL    |       |
| UPTIME            | varchar(64) | YES  |      | NULL    |       |
+-------------------+-------------+------+------+---------+-------+
19 rows in set (0.00 sec)
```

TIKV_STORE_STATUS 表中列的含义如下：

- STORE_ID：Store 的 ID。
- ADDRESS：Store 的地址。
- STORE_STATE：Store 状态的标识符，与 STORE_STATE_NAME 相对应。
- STORE_STATE_NAME：Store 状态的名字，为 Up / Offline / Tombstone 中的一种。
- LABEL：给 Store 设置的标签。
- VERSION：Store 的版本号。
- CAPACITY：Store 的存储容量。
- AVAILABLE：Store 的剩余存储空间。
- LEADER_COUNT：Store 上的 leader 的数量。
- LEADER_WEIGHT：Store 的 leader 权重。
- LEADER_SCORE：Store 的 leader 评分。
- LEADER_SIZE：Store 上的所有 leader 的近似总数据量 (MB)。
- REGION_COUNT：Store 上的 Region 总数。
- REGION_WEIGHT：Store 的 Region 权重。
- REGION_SCORE：Store 的 Region 评分。
- REGION_SIZE：Store 上的所有 Region 的近似总数据量 (MB)。
- START_TS：Store 启动时的时间戳。
- LAST_HEARTBEAT_TS：Store 上次发出心跳的时间戳。
- UPTIME：Store 启动以来的总时间。

2.17.2.47 USER_PRIVILEGES

USER_PRIVILEGES 表提供了关于全局权限的信息。该表的数据根据 mysql.user 系统表生成。

```
USE information_schema;
DESC user_privileges;
```

```
+----------------+--------------+------+------+---------+-------+
| Field          | Type         | Null | Key  | Default | Extra |
+----------------+--------------+------+------+---------+-------+
| GRANTEE        | varchar(81)  | YES  |      | NULL    |       |
| TABLE_CATALOG  | varchar(512) | YES  |      | NULL    |       |
| PRIVILEGE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| IS_GRANTABLE   | varchar(3)   | YES  |      | NULL    |       |
+----------------+--------------+------+------+---------+-------+
4 rows in set (0.00 sec)
```

```
SELECT * FROM user_privileges;
```

```
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | Select                  | YES          |
| 'root'@'%' | def           | Insert                  | YES          |
| 'root'@'%' | def           | Update                  | YES          |
| 'root'@'%' | def           | Delete                  | YES          |
| 'root'@'%' | def           | Create                  | YES          |
| 'root'@'%' | def           | Drop                    | YES          |
| 'root'@'%' | def           | Process                 | YES          |
| 'root'@'%' | def           | References              | YES          |
| 'root'@'%' | def           | Alter                   | YES          |
| 'root'@'%' | def           | Show Databases          | YES          |
| 'root'@'%' | def           | Super                   | YES          |
| 'root'@'%' | def           | Execute                 | YES          |
| 'root'@'%' | def           | Index                   | YES          |
| 'root'@'%' | def           | Create User             | YES          |
| 'root'@'%' | def           | Trigger                 | YES          |
| 'root'@'%' | def           | Create View             | YES          |
| 'root'@'%' | def           | Show View               | YES          |
| 'root'@'%' | def           | Create Role             | YES          |
| 'root'@'%' | def           | Drop Role               | YES          |
| 'root'@'%' | def           | CREATE TEMPORARY TABLES | YES          |
| 'root'@'%' | def           | LOCK TABLES             | YES          |
| 'root'@'%' | def           | CREATE ROUTINE          | YES          |
| 'root'@'%' | def           | ALTER ROUTINE           | YES          |
| 'root'@'%' | def           | EVENT                   | YES          |
| 'root'@'%' | def           | SHUTDOWN                | YES          |
| 'root'@'%' | def           | RELOAD                  | YES          |
| 'root'@'%' | def           | FILE                    | YES          |
| 'root'@'%' | def           | CONFIG                  | YES          |
+------------+---------------+-------------------------+--------------+
```

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

USER_PRIVILEGES 表中列的含义如下：

- GRANTEE：被授权的用户名称，格式为 'user_name'@'host_name'。
- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- PRIVILEGE_TYPE：被授权的权限类型，每行只列一个权限。
- IS_GRANTABLE：如果用户有 GRANT OPTION 的权限，则为 YES，否则为 NO。

2.17.3 Metrics Schema

METRICS_SCHEMA 是基于 Prometheus 中 TiDB 监控指标的一组视图。每个表的 PromQL（Prometheus 查询语言）的源均可在INFORMATION_SCHEMA.METRICS_TABLES 表中找到。

```
use metrics_schema;
SELECT * FROM uptime;
SELECT * FROM information_schema.metrics_tables WHERE table_name='uptime'\G
```

```
+----------------------------+-----------------+------------+--------------------+
| time                       | instance        | job        | value              |
+----------------------------+-----------------+------------+--------------------+
| 2020-07-06 15:26:26.203000 | 127.0.0.1:10080 | tidb       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:10080 | tidb       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:20180 | tikv       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:20180 | tikv       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:2379  | pd         | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:2379  | pd         | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:9090  | prometheus | 123.72300004959106 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:9090  | prometheus | 183.72300004959106 |
+----------------------------+-----------------+------------+--------------------+
8 rows in set (0.00 sec)
*************************** 1. row ***************************
TABLE_NAME: uptime
    PROMQL: (time() - process_start_time_seconds{$LABEL_CONDITIONS})
    LABELS: instance,job
  QUANTILE: 0
   COMMENT: TiDB uptime since last restart(second)
1 row in set (0.00 sec)
```

```
show tables;
```

```
+------------------------------------------------+
| Tables_in_metrics_schema                       |
+------------------------------------------------+
| abnormal_stores                                |
```

```
| etcd_disk_wal_fsync_rate                          |
| etcd_wal_fsync_duration                           |
| etcd_wal_fsync_total_count                        |
| etcd_wal_fsync_total_time                         |
| go_gc_count                                       |
| go_gc_cpu_usage                                   |
| go_gc_duration                                    |
| go_heap_mem_usage                                 |
| go_threads                                        |
| goroutines_count                                  |
| node_cpu_usage                                    |
| node_disk_available_size                          |
| node_disk_io_util                                 |
| node_disk_iops                                    |
| node_disk_read_latency                            |
| node_disk_size                                    |
..
| tikv_storage_async_request_total_time             |
| tikv_storage_async_requests                       |
| tikv_storage_async_requests_total_count           |
| tikv_storage_command_ops                          |
| tikv_store_size                                   |
| tikv_thread_cpu                                   |
| tikv_thread_nonvoluntary_context_switches         |
| tikv_thread_voluntary_context_switches            |
| tikv_threads_io                                   |
| tikv_threads_state                                |
| tikv_total_keys                                   |
| tikv_wal_sync_duration                            |
| tikv_wal_sync_max_duration                        |
| tikv_worker_handled_tasks                         |
| tikv_worker_handled_tasks_total_num               |
| tikv_worker_pending_tasks                         |
| tikv_worker_pending_tasks_total_num               |
| tikv_write_stall_avg_duration                     |
| tikv_write_stall_max_duration                     |
| tikv_write_stall_reason                           |
| up                                                |
| uptime                                            |
+---------------------------------------------------+
626 rows in set (0.00 sec)
```

METRICS_SCHEMA 是监控相关的 summary 表的数据源，例如metrics_summary、metrics_summary_by_label 和inspection_summary。

2.17.3.1 更多例子

下面以 metrics_schema 中的 tidb_query_duration 监控表为例，介绍监控表相关的使用和原理，其他的监控表原理均类似。

查询 information_schema.metrics_tables 中关于 tidb_query_duration 表相关的信息如下：

```
select * from information_schema.metrics_tables where table_name='tidb_query_duration';
```

```
+--
    ↪ -------------------+-------------------------------------------------------------------------------------------
    ↪
| TABLE_NAME           | PROMQL
    ↪
    ↪ | LABELS             | QUANTILE | COMMENT                                                |
+--
    ↪ -------------------+-------------------------------------------------------------------------------------------
    ↪
| tidb_query_duration | histogram_quantile($QUANTILE, sum(rate(
    ↪ tidb_server_handle_query_duration_seconds_bucket{$LABEL_CONDITIONS}[$RANGE_DURATION])) by
    ↪  (le,sql_type,instance)) | instance,sql_type | 0.9      | The quantile of TiDB query
    ↪ durations(second) |
+--
    ↪ -------------------+-------------------------------------------------------------------------------------------
    ↪
```

- TABLE_NAME：对应于 metrics_schema 中的表名，这里表名是 tidb_query_duration。
- PROMQL：因为监控表的原理是将 SQL 映射成 PromQL 后向 Prometheus 请求数据，并将 Prometheus 返回的结果转换成 SQL 查询结果。该字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控项定义的 label，tidb_query_duration 有两个 label，分别是 instance 和 sql_type。
- QUANTILE：百分位。直方图类型的监控数据会指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。tidb_query_duration 默认查询 0.9 ，也就是 P90 的监控值。
- COMMENT：对这个监控表的解释。可以看出 tidb_query_duration 表是用来查询 TiDB query 执行的百分位时间，如 P999/P99/P90 的查询耗时，单位是秒。

再来看 tidb_query_duration 的表结构：

```
show create table metrics_schema.tidb_query_duration;
```

```
+--
    ↪ -------------------+-------------------------------------------------------------------------------------------
    ↪
| Table                | Create Table
    ↪
    ↪ |
```

```
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| tidb_query_duration | CREATE TABLE `tidb_query_duration` (
    ↪                                                                                 |
|                     |   `time` datetime unsigned DEFAULT CURRENT_TIMESTAMP,
    ↪                                                                 |
|                     |   `instance` varchar(512) DEFAULT NULL,
    ↪                                                                              |
|                     |   `sql_type` varchar(512) DEFAULT NULL,
    ↪                                                                              |
|                     |   `quantile` double unsigned DEFAULT '0.9',
    ↪                                                                          |
|                     |   `value` double unsigned DEFAULT NULL
    ↪                                                                           |
|                     | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin COMMENT='The
    ↪ quantile of TiDB query durations(second)' |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
```

- `time`：监控项的时间。
- `instance` 和 `sql_type`：是 `tidb_query_duration` 这个监控项的 label。`instance` 表示监控的地址，`sql_type` 表示执行 SQL 的类似。
- `quantile`，百分位，直方图类型的监控都会有该列，表示查询的百分位时间，如 `quantile=0.9` 就是查询 P90 的时间。
- `value`：监控项的值。

下面是查询时间 [2020-03-25 23:40:00, 2020-03-25 23:42:00] 范围内的 P99 的 TiDB Query 耗时：

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    ↪ 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+-------------------+----------+----------+----------------+
| time                | instance          | sql_type | quantile | value          |
+---------------------+-------------------+----------+----------+----------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.509929485256 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.494690793986 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.493460506934 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.152058493415 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.152193879678 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.140498483232 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99     | 0.47104        |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99     | 0.11776        |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99     | 0.11776        |
```

```
+--------------------+------------------+----------+----------+----------------+
```

以上查询结果的第一行意思是，在 2020-03-25 23:40:00 时，在 TiDB 实例 172.16.5.40:10089 上，Insert 类型的语句的 P99 执行时间是 0.509929485256 秒。其他各行的含义类似，sql_type 列的其他值含义如下：

- Select：表示执行的 select 类型的语句。
- internal：表示 TiDB 的内部 SQL 语句，一般是统计信息更新，获取全局变量相关的内部语句。

进一步再查看上面语句的执行计划如下：

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    ↪ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------------
    ↪
| id                | estRows  | task | access object            | operator info
    ↪
    ↪ |
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------------
    ↪
| Selection_5       | 8000.00  | root |                          | not(isnull(Column#5))
    ↪
    ↪ |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    ↪ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[60s])) by (le,sql_type
    ↪ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:1m0s |
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------------
    ↪
```

可以发现执行计划中有一个 PromQL, 以及查询监控的 start_time 和 end_time，还有 step 值，在实际执行时，TiDB 会调用 Prometheus 的 query_range HTTP API 接口来查询监控数据。

从以上结果可知，在 [2020-03-25 23:40:00, 2020-03-25 23:42:00] 时间范围内，每个 label 只有三个时间的值，间隔时间是 1 分钟，即执行计划中的 step 值。该间隔时间由以下两个 session 变量决定：

- tidb_metric_query_step：查询的分辨率步长。从 Prometheus 的 query_range 接口查询数据时需要指定 start_time，end_time 和 step，其中 step 会使用该变量的值。
- tidb_metric_query_range_duration：查询监控时，会将 PROMQL 中的 $RANGE_DURATION 替换成该变量的值，默认值是 60 秒。

如果想要查看不同时间粒度的监控项的值，用户可以修改上面两个 session 变量后查询监控表，示例如下：

首先修改两个 session 变量的值，将时间粒度设置为 30 秒。

> **注意：**
>
> Prometheus 支持查询的最小粒度为 30 秒。

```
set @@tidb_metric_query_step=30;
set @@tidb_metric_query_range_duration=30;
```

再查询 tidb_query_duration 监控如下，可以发现在三分钟时间范围内，每个 label 有六个时间的值，每个值时间间隔是 30 秒。

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    ↪ 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+-------------------+----------+----------+-----------------+
| time                | instance          | sql_type | quantile | value           |
+---------------------+-------------------+----------+----------+-----------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.483285651924  |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | Insert   | 0.99     | 0.484151462113  |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.504576        |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | Insert   | 0.99     | 0.493577384561  |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.49482474311   |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.189253402185  |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | Select   | 0.99     | 0.184224951851  |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.151673410553  |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | Select   | 0.99     | 0.127953838989  |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.127455434547  |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0624          |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | internal | 0.99     | 0.12416         |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0304          |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | internal | 0.99     | 0.06272         |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0629333333333 |
+---------------------+-------------------+----------+----------+-----------------+
```

最后查看执行计划，也会发现执行计划中的 PromQL 以及 step 的值都已经变成了 30 秒。

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    ↪ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------------
    ↪
| id                | estRows  | task | access object            | operator info
    ↪
    ↪ |
```

```
+--
    ↪ ----------------+----------+------+--------------------------+-------------------------------------------
    ↪
| Selection_5       | 8000.00  | root |                          | not(isnull(Column#5))
    ↪
    ↪ |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    ↪ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[30s])) by (le,sql_type
    ↪ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:30s |
+--
    ↪ ----------------+----------+------+--------------------------+-------------------------------------------
    ↪
```

3 账户与安全

3.1 权限管理

3.1.1 与 MySQL 安全特性差异

除以下功能外，TiDB 支持与 MySQL 5.7 类似的安全特性。

- 不支持列级别权限设置。
- 不支持密码过期，最后一次密码变更记录以及密码生存期。#9709
- 不支持权限属性 max_questions，max_updated，max_connections 以及 max_user_connections。
- 不支持密码验证。#9741

3.1.1.1 可用的身份验证插件

TiDB 支持多种身份验证方式。通过使用CREATE USER 语句和ALTER USER 语句，即可创建新用户或更改 TiDB 权限系统内的已有用户。TiDB 身份验证方式与 MySQL 兼容，其名称与 MySQL 保持一致。

TiDB 目前支持的身份验证方式可在以下的表格中查找到。服务器和客户端建立连接时，如要指定服务器对外通告的默认验证方式，可通过default_authentication_plugin 变量进行设置。

针对 TLS 身份验证，TiDB 目前采用不同的配置方案。具体情况请参见为 TiDB 客户端服务端间通信开启加密传输。

身份验证方式	支持
mysql_native_password	是
sha256_password	否
caching_sha2_password	是（5.2.0 版本起）
auth_socket	是（5.3.0 版本起）
[TLS Certificates]	是
LDAP	否

身份验证方式	支持
PAM	否
ed25519 (MariaDB)	否
GSSAPI (MariaDB)	否
FIDO	否

3.1.2 权限管理

TiDB 支持 MySQL 5.7 的权限管理系统，包括 MySQL 的语法和权限类型。同时 TiDB 还支持 MySQL 8.0 的以下特性：

- 从 TiDB 3.0 开始，支持 SQL 角色。
- 从 TiDB 5.1 开始，支持动态权限。

本文档主要介绍 TiDB 权限相关操作、各项操作需要的权限以及权限系统的实现。

3.1.2.1 权限相关操作

3.1.2.1.1 授予权限

授予 xxx 用户对数据库 test 的读权限：

```
GRANT SELECT ON test.* TO 'xxx'@'%';
```

为 xxx 用户授予所有数据库，全部权限：

```
GRANT ALL PRIVILEGES ON *.* TO 'xxx'@'%';
```

默认情况下，如果指定的用户不存在，GRANT 语句将报错。该行为受 SQL Mode 中的 NO_AUTO_CREATE_USER 控制。

```
SET sql_mode=DEFAULT;
Query OK, 0 rows affected (0.00 sec)

SELECT @@sql_mode;
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
    ↪ NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
```

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

SELECT * FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)

GRANT ALL PRIVILEGES ON test.* TO 'idontexist';
ERROR 1105 (HY000): You are not allowed to create a user with GRANT

SELECT user,host,authentication_string FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)
```

在下面的例子中，由于没有将 SQL Mode 设置为 NO_AUTO_CREATE_USER，用户 idontexist 会被自动创建且密码为空。不推荐使用这种方式，因为会带来安全风险：如果用户名拼写错误，会导致新用户被创建且密码为空。

```
SET @@sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
Query OK, 0 rows affected (0.00 sec)

SELECT @@sql_mode;
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
    ↪ NO_ENGINE_SUBSTITUTION |
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

SELECT * FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)

GRANT ALL PRIVILEGES ON test.* TO 'idontexist';
Query OK, 1 row affected (0.05 sec)

SELECT user,host,authentication_string FROM mysql.user WHERE user='idontexist';
+------------+------+-----------------------+
```

```
| user       | host | authentication_string |
+------------+------+-----------------------+
| idontexist | %    |                       |
+------------+------+-----------------------+
1 row in set (0.01 sec)
```

GRANT 还可以模糊匹配地授予用户数据库的权限：

```
GRANT ALL PRIVILEGES ON `te%`.* TO genius;
```

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

```
SELECT user,host,db FROM mysql.db WHERE user='genius';
```

```
+--------|------|-----+
| user   | host | db  |
+--------|------|-----+
| genius | %    | te% |
+--------|------|-----+
1 row in set (0.00 sec)
```

这个例子中通过 % 模糊匹配，所有 te 开头的数据库，都被授予了权限。

3.1.2.1.2 收回权限

REVOKE 语句与 GRANT 对应：

```
REVOKE ALL PRIVILEGES ON `test`.* FROM 'genius'@'localhost';
```

> 注意：
>
> REVOKE 收回权限时只做精确匹配，若找不到记录则报错。而 GRANT 授予权限时可以使用模糊匹配。

```
REVOKE ALL PRIVILEGES ON `te%`.* FROM 'genius'@'%';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'genius' on host '%'
```

关于模糊匹配和转义，字符串和 identifier：

```
GRANT ALL PRIVILEGES ON `te\%`.* TO 'genius'@'localhost';
```

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

上述例子是精确匹配名为 te% 的数据库，注意使用 \ 转义字符。

以单引号包含的部分，是一个字符串。以反引号包含的部分，是一个 identifier。注意下面的区别：

```
GRANT ALL PRIVILEGES ON 'test'.* TO 'genius'@'localhost';
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near ''test'.* to 'genius'@'localhost'' at line 1
```

```
GRANT ALL PRIVILEGES ON `test`.* TO 'genius'@'localhost';
```

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

如果想将一些特殊的关键字做为表名，可以用反引号包含起来。比如：

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

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

3.1.2.1.3 查看为用户分配的权限

SHOW GRANTS 语句可以查看为用户分配了哪些权限。例如：

查看当前用户的权限：

```
SHOW GRANTS;
```

```
+---------------------------------------------------------------+
| Grants for User                                               |
+---------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION |
+---------------------------------------------------------------+
```

或者：

```
SHOW GRANTS FOR CURRENT_USER();
```

查看某个特定用户的权限：

```
SHOW GRANTS FOR 'user'@'host';
```

例如，创建一个用户 rw_user@192.168.% 并为其授予 test.write_table 表的写权限，和全局读权限。

```
CREATE USER `rw_user`@`192.168.%`;
GRANT SELECT ON *.* TO `rw_user`@`192.168.%`;
GRANT INSERT, UPDATE ON `test`.`write_table` TO `rw_user`@`192.168.%`;
```

查看用户 rw_user@192.168.% 的权限。

```
SHOW GRANTS FOR `rw_user`@`192.168.%`;
```

```
+-------------------------------------------------------------------+
| Grants for rw_user@192.168.%                                      |
+-------------------------------------------------------------------+
| GRANT Select ON *.* TO 'rw_user'@'192.168.%'                      |
| GRANT Insert,Update ON test.write_table TO 'rw_user'@'192.168.%' |
+-------------------------------------------------------------------+
```

3.1.2.1.4 动态权限

从 v5.1 开始，TiDB 支持 MySQL 8.0 中的动态权限特性。动态权限用于限制 SUPER 权限，实现对某些操作更细粒度的访问。例如，系统管理员可以使用动态权限来创建一个只能执行 BACKUP 和 RESTORE 操作的用户帐户。

动态权限包括：

- BACKUP_ADMIN
- RESTORE_ADMIN
- ROLE_ADMIN
- CONNECTION_ADMIN
- SYSTEM_VARIABLES_ADMIN
- RESTRICTED_REPLICA_WRITER_ADMIN 允许权限拥有者在 TiDB 集群开启了只读模式的情况下不受影响地执行写入或更新操作，详见tidb_restricted_read_only 配置项。

若要查看全部的动态权限，请执行 SHOW PRIVILEGES 语句。由于用户可使用插件来添加新的权限，因此可分配的权限列表可能因用户的 TiDB 安装情况而异。

3.1.2.2 TiDB 各操作需要的权限

TiDB 用户目前拥有的权限可以在 INFORMATION_SCHEMA.USER_PRIVILEGES 表中查找到。例如：

```
SELECT * FROM INFORMATION_SCHEMA.USER_PRIVILEGES WHERE grantee = "'root'@'%'";
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | Select                  | YES          |
| 'root'@'%' | def           | Insert                  | YES          |
| 'root'@'%' | def           | Update                  | YES          |
| 'root'@'%' | def           | Delete                  | YES          |
| 'root'@'%' | def           | Create                  | YES          |
| 'root'@'%' | def           | Drop                    | YES          |
| 'root'@'%' | def           | Process                 | YES          |
| 'root'@'%' | def           | References              | YES          |
| 'root'@'%' | def           | Alter                   | YES          |
| 'root'@'%' | def           | Show Databases          | YES          |
```

```
| 'root'@'%' | def          | Super                    | YES          |
| 'root'@'%' | def          | Execute                  | YES          |
| 'root'@'%' | def          | Index                    | YES          |
| 'root'@'%' | def          | Create User              | YES          |
| 'root'@'%' | def          | Create Tablespace        | YES          |
| 'root'@'%' | def          | Trigger                  | YES          |
| 'root'@'%' | def          | Create View              | YES          |
| 'root'@'%' | def          | Show View                | YES          |
| 'root'@'%' | def          | Create Role              | YES          |
| 'root'@'%' | def          | Drop Role                | YES          |
| 'root'@'%' | def          | CREATE TEMPORARY TABLES  | YES          |
| 'root'@'%' | def          | LOCK TABLES              | YES          |
| 'root'@'%' | def          | CREATE ROUTINE           | YES          |
| 'root'@'%' | def          | ALTER ROUTINE            | YES          |
| 'root'@'%' | def          | EVENT                    | YES          |
| 'root'@'%' | def          | SHUTDOWN                 | YES          |
| 'root'@'%' | def          | RELOAD                   | YES          |
| 'root'@'%' | def          | FILE                     | YES          |
| 'root'@'%' | def          | CONFIG                   | YES          |
| 'root'@'%' | def          | REPLICATION CLIENT       | YES          |
| 'root'@'%' | def          | REPLICATION SLAVE        | YES          |
+------------+--------------+--------------------------+--------------+
31 rows in set (0.00 sec)
```

3.1.2.2.1 ALTER

- 对于所有的 ALTER 语句，均需要用户对所操作的表拥有 ALTER 权限。
- 除 ALTER...DROP 和 ALTER...RENAME TO 外，均需要对所操作表拥有 INSERT 和 CREATE 权限。
- 对于 ALTER...DROP 语句，需要对表拥有 DROP 权限。
- 对于 ALTER...RENAME TO 语句，需要对重命名前的表拥有 DROP 权限，对重命名后的表拥有 CREATE 和 INSERT 权限。

> **注意：**
>
> 根据 MySQL 5.7 文档中的说明，对表进行 ALTER 操作需要 INSERT 和 CREATE 权限，但在 MySQL 5.7.25 版本实际情况中，该操作仅需要 ALTER 权限。目前，TiDB 中的 ALTER 权限与 MySQL 实际行为保持一致。

3.1.2.2.2 BACKUP

需要拥有 SUPER 或者 BACKUP_ADMIN 权限。

3.1.2.2.3 CREATE DATABASE

需要拥有全局 CREATE 权限。

3.1.2.2.4 CREATE INDEX

需要对所操作的表拥有 INDEX 权限。

3.1.2.2.5 CREATE TABLE

需要对要创建的表所在的数据库拥有 CREATE 权限；若使用 CREATE TABLE...LIKE... 需要对相关的表拥有 SELECT 权限。

3.1.2.2.6 CREATE VIEW

需要拥有 CREATE VIEW 权限。

> 注意：
> 如果当前登录用户与创建视图的用户不同，除需要 CREATE VIEW 权限外，还需要 SUPER 权限。

3.1.2.2.7 DROP DATABASE

需要对数据库拥有 DROP 权限。

3.1.2.2.8 DROP INDEX

需要对所操作的表拥有 INDEX 权限。

3.1.2.2.9 DROP TABLES

需要对所操作的表拥有 DROP 权限。

3.1.2.2.10 LOAD DATA

LOAD DATA 需要对所操作的表拥有 INSERT 权限。执行 REPLACE INTO 语句还需要对所操作的表拥有 DELETE 权限。

3.1.2.2.11 TRUNCATE TABLE

需要对所操作的表拥有 DROP 权限。

3.1.2.2.12 RENAME TABLE

需要对重命名前的表拥有 ALTER 和 DROP 权限，对重命名后的表拥有 CREATE 和 INSERT 权限。

3.1.2.2.13 ANALYZE TABLE

需要对所操作的表拥有 INSERT 和 SELECT 权限。

3.1.2.2.14 SHOW

SHOW CREATE TABLE 需要任意一种权限。

SHOW CREATE VIEW 需要 SHOW VIEW 权限。

SHOW GRANTS 需要拥有对 mysql 数据库的 SELECT 权限。如果是使用 SHOW GRANTS 查看当前用户权限，则不需要任何权限。

SHOW PROCESSLIST 需要 SUPER 权限来显示属于其他用户的连接。

3.1.2.2.15 CREATE ROLE/USER

CREATE ROLE 需要 CREATE ROLE 权限。

CREATE USER 需要 CREATE USER 权限

3.1.2.2.16 DROP ROLE/USER

DROP ROLE 需要 DROP ROLE 权限。

DROP USER 需要 CREATE USER 权限

3.1.2.2.17 ALTER USER

ALTER USER 需要 CREATE USER 权限。

3.1.2.2.18 GRANT

GRANT 需要 GRANT 权限并且拥有 GRANT 所赋予的权限。

如果在 GRANTS 语句中创建用户，需要有 CREATE USER 权限。

GRANT ROLE 操作需要拥有 SUPER 或者 ROLE_ADMIN 权限。

3.1.2.2.19 REVOKE

REVOKE 需要 GRANT 权限并且拥有 REVOKE 所指定要撤销的权限。

REVOKE ROLE 操作需要拥有 SUPER 或者 ROLE_ADMIN 权限。

3.1.2.2.20 SET GLOBAL

使用 SET GLOBAL 设置全局变量需要拥有 SUPER 或者 SYSTEM_VARIABLES_ADMIN 权限。

3.1.2.2.21 ADMIN

需要拥有 SUPER 权限。

3.1.2.2.22 SET DEFAULT ROLE

需要拥有 SUPER 权限。

3.1.2.2.23 KILL

使用 KILL 终止其他用户的会话需要拥有 SUPER 或者 CONNECTION_ADMIN 权限。

3.1.2.3 权限系统的实现

3.1.2.3.1 授权表

以下几张系统表是非常特殊的表，权限相关的数据全部存储在这几张表内。

- mysql.user：用户账户，全局权限
- mysql.db：数据库级别的权限
- mysql.tables_priv：表级别的权限
- mysql.columns_priv：列级别的权限，当前暂不支持

这几张表包含了数据的生效范围和权限信息。例如，mysql.user 表的部分数据：

```
SELECT User,Host,Select_priv,Insert_priv FROM mysql.user LIMIT 1;
```

```
+------|------|-------------|-------------+
| User | Host | Select_priv | Insert_priv |
+------|------|-------------|-------------+
| root | %    | Y           | Y           |
+------|------|-------------|-------------+
1 row in set (0.00 sec)
```

这条记录中，Host 和 User 决定了 root 用户从任意主机 (%) 发送过来的连接请求可以被接受，而 Select_priv 和 Insert_priv 表示用户拥有全局的 Select 和 Insert 权限。mysql.user 这张表里面的生效范围是全局的。

mysql.db 表里面包含的 Host 和 User 决定了用户可以访问哪些数据库，权限列的生效范围是数据库。

理论上，所有权限管理相关的操作，都可以通过直接对授权表的 CRUD 操作完成。

实现层面其实也只是包装了一层语法糖。例如删除用户会执行：

```
DELETE FROM mysql.user WHERE user='test';
```

但是，不推荐手动修改授权表，建议使用 DROP USER 语句：

```
DROP USER 'test';
```

3.1.2.3.2 连接验证

当客户端发送连接请求时，TiDB 服务器会对登录操作进行验证。验证过程先检查 mysql.user 表，当某条记录的 User 和 Host 和连接请求匹配上了，再去验证 authentication_string。用户身份基于两部分信息，发起连接的客户端的 Host，以及用户名 User。如果 User 不为空，则用户名必须精确匹配。

User+Host 可能会匹配 user 表里面多行，为了处理这种情况，user 表的行是排序过的，客户端连接时会依次去匹配，并使用首次匹配到的那一行做权限验证。排序是按 Host 在前，User 在后。

3.1.2.3.3 请求验证

连接成功之后，请求验证会检测执行操作是否拥有足够的权限。

对于数据库相关请求 (INSERT, UPDATE)，先检查 mysql.user 表里面的用户全局权限，如果权限够，则直接可以访问。如果全局权限不足，则再检查 mysql.db 表。

user 表的权限是全局的，并且不管默认数据库是哪一个。比如 user 里面有 DELETE 权限，任何一行，任何的表，任何的数据库。

db表里面，User 为空是匹配匿名用户，User 里面不能有通配符。Host 和 Db 列里面可以有 % 和 _，可以模式匹配。

user 和 db 读到内存也是排序的。

tables_priv 和 columns_priv 中使用 % 是类似的，但是在Db, Table_name, Column_name 这些列不能包含 %。加载进来时排序也是类似的。

3.1.2.3.4 生效时机

TiDB 启动时，将一些权限检查的表加载到内存，之后使用缓存的数据来验证权限。系统会周期性的将授权表从数据库同步到缓存，生效则是由同步的周期决定，目前这个值设定的是 5 分钟。

修改了授权表，如果需要立即生效，可以手动调用：

```
FLUSH PRIVILEGES;
```

3.1.3 TiDB 用户账户管理

本文档主要介绍如何管理 TiDB 用户账户。

要快速了解 TiDB 如何进行认证与赋权并创建与管理用户账户，建议先观看下面的培训视频（时长 22 分钟）。注意本视频只作为学习参考，如需了解具体的用户账户管理方法，请参考本文档的内容。

3.1.3.1 用户名和密码

TiDB 将用户账户存储在 mysql.user 系统表里面。每个账户由用户名和 host 作为标识。每个账户可以设置一个密码。

通过 MySQL 客户端连接到 TiDB 服务器，通过指定的账户和密码登录：

```
mysql --port 4000 --user xxx --password
```

使用缩写的命令行参数则是：

```
mysql -P 4000 -u xxx -p
```

3.1.3.2 添加用户

添加用户有两种方式：

- 通过标准的用户管理的 SQL 语句创建用户以及授予权限，比如 CREATE USER 和 GRANT。
- 直接通过 INSERT、UPDATE 和 DELETE 操作授权表。

推荐使用第一种方式。第二种方式修改容易导致一些不完整的修改，因此不推荐。还有另一种可选方式是使用第三方工具的图形化界面工具。

```
CREATE USER [IF NOT EXISTS] user [IDENTIFIED BY 'auth_string'];
```

设置登录密码后，auth_string 会被 TiDB 经过加密存储在 mysql.user 表中。

```
CREATE USER 'test'@'127.0.0.1' IDENTIFIED BY 'xxx';
```

TiDB 的用户账户名由一个用户名和一个主机名组成。账户名的语法为 'user_name'@'host_name'。

- user_name 大小写敏感。
- host_name 可以是一个主机名或 IP 地址。主机名或 IP 地址中允许使用通配符 % 和 _。例如，名为 '%' 的主机名可以匹配所有主机，'192.168.1.%' 可以匹配子网中的所有主机。

host 支持模糊匹配，比如：

```
CREATE USER 'test'@'192.168.10.%';
```

允许 test 用户从 192.168.10 子网的任何一个主机登录。

如果没有指定 host，则默认是所有 IP 均可登录。如果没有指定密码，默认为空：

```
CREATE USER 'test';
```

等价于：

```
CREATE USER 'test'@'%' IDENTIFIED BY '';
```

为一个不存在的用户授权时，是否会自动创建用户的行为受 sql_mode 影响。如果 sql_mode 中包含 NO_AUTO_CREATE_USER，则 GRANT 不会自动创建用户并报错。

假设 sql_mode 不包含 NO_AUTO_CREATE_USER，下面的例子用 CREATE USER 和 GRANT 语句创建了四个账户：

```
CREATE USER 'finley'@'localhost' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'localhost' WITH GRANT OPTION;
```

```
CREATE USER 'finley'@'%' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'%' WITH GRANT OPTION;
```

```
CREATE USER 'admin'@'localhost' IDENTIFIED BY 'admin_pass';
```

```
GRANT RELOAD,PROCESS ON *.* TO 'admin'@'localhost';
```

```
CREATE USER 'dummy'@'localhost';
```

使用 SHOW GRANTS 可以看到为一个用户授予的权限：

```
SHOW GRANTS FOR 'admin'@'localhost';
```

```
+-------------------------------------------------------+
| Grants for admin@localhost                            |
+-------------------------------------------------------+
| GRANT RELOAD, PROCESS ON *.* TO 'admin'@'localhost' |
+-------------------------------------------------------+
```

3.1.3.3 删除用户

使用 DROP USER 语句可以删除用户，例如：

```
DROP USER 'test'@'localhost';
```

这个操作会清除用户在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录。

3.1.3.4 保留用户账户

TiDB 在数据库初始化时会生成一个 'root'@'%' 的默认账户。

3.1.3.5 设置资源限制

暂不支持。

3.1.3.6 设置密码

TiDB 将密码存在 mysql.user 系统数据库里面。只有拥有 CREATE USER 权限，或者拥有 mysql 数据库权限（INSERT 权限用于创建，UPDATE 权限用于更新）的用户才能够设置或修改密码。

- 在 CREATE USER 创建用户时通过 IDENTIFIED BY 指定密码：

  ```
  CREATE USER 'test'@'localhost' IDENTIFIED BY 'mypass';
  ```

- 为一个已存在的账户修改密码，可以通过 SET PASSWORD FOR 或者 ALTER USER 语句完成：

```
SET PASSWORD FOR 'root'@'%' = 'xxx';
```

或者：

```
ALTER USER 'test'@'localhost' IDENTIFIED BY 'mypass';
```

3.1.3.7 忘记 root 密码

1. 修改配置文件，在 security 部分添加 skip-grant-table：

```
[security]
skip-grant-table = true
```

2. 使用修改之后的配置启动 TiDB，然后使用 root 登录后修改密码：

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

设置 skip-grant-table 之后，启动 TiDB 进程会增加操作系统用户检查，只有操作系统的 root 用户才能启动 TiDB 进程。

3.1.3.8 FLUSH PRIVILEGES

用户以及权限相关的信息都存储在 TiKV 服务器中，TiDB 在进程内部会缓存这些信息。一般通过 CREATE USER，GRANT 等语句来修改相关信息时，可在整个集群迅速生效。如果遇到网络或者其它因素影响，由于 TiDB 会周期性地更新缓存信息，正常情况下，最多 15 分钟左右生效。

如果授权表已被直接修改，则不会通知 TiDB 节点更新缓存，运行如下命令可使改动立即生效：

```
FLUSH PRIVILEGES;
```

详情参见权限管理。

3.1.4 基于角色的访问控制

TiDB 的基于角色的访问控制 (RBAC) 系统的实现类似于 MySQL 8.0 的 RBAC 系统。TiDB 兼容大部分 MySQL RBAC 系统的语法。

本文档主要介绍 TiDB 基于角色的访问控制相关操作及实现。

3.1.4.1 角色访问控制相关操作

角色是一系列权限的集合。用户可以创建角色、删除角色、将权限赋予角色；也可以将角色授予给其他用户，被授予的用户在启用角色后，可以得到角色所包含的权限。

3.1.4.1.1 创建角色

创建角色 app_developer，app_read 和 app_write：

```
CREATE ROLE 'app_developer', 'app_read', 'app_write';
```

角色名的格式和规范可以参考TiDB 用户账户管理。

角色会被保存在 mysql.user 表中，角色名称的主机名部分（如果省略）默认为 '%'。如果表中有同名角色或用户，角色会创建失败并报错。创建角色的用户需要拥有 CREATE ROLE 或 CREATE USER 权限。

3.1.4.1.2 授予角色权限

为角色授予权限和为用户授予权限操作相同，可参考TiDB 权限管理。

为 app_read 角色授予数据库 app_db 的读权限：

```
GRANT SELECT ON app_db.* TO 'app_read'@'%';
```

为 app_write 角色授予数据库 app_db 的写权限：

```
GRANT INSERT, UPDATE, DELETE ON app_db.* TO 'app_write'@'%';
```

为 app_developer 角色授予 app_db 数据库的全部权限：

```
GRANT ALL ON app_db.* TO 'app_developer';
```

3.1.4.1.3 将角色授予给用户

假设有一个用户拥有开发者角色，可以对 app_db 的所有操作权限；另外有两个用户拥有 app_db 的只读权限；还有一个用户拥有 app_db 的读写权限。

首先用 CREATE USER 来创建用户。

```
CREATE USER 'dev1'@'localhost' IDENTIFIED BY 'dev1pass';
CREATE USER 'read_user1'@'localhost' IDENTIFIED BY 'read_user1pass';
CREATE USER 'read_user2'@'localhost' IDENTIFIED BY 'read_user2pass';
CREATE USER 'rw_user1'@'localhost' IDENTIFIED BY 'rw_user1pass';
```

然后使用 GRANT 授予用户对应的角色。

```
GRANT 'app_developer' TO 'dev1'@'localhost';
GRANT 'app_read' TO 'read_user1'@'localhost', 'read_user2'@'localhost';
GRANT 'app_read', 'app_write' TO 'rw_user1'@'localhost';
```

用户执行将角色授予给其他用户或者收回角色的命令，需要用户拥有 SUPER 权限。将角色授予给用户时并不会启用该角色，启用角色需要额外的操作。

以下操作可能会形成一个“关系环”：

```
CREATE USER 'u1', 'u2';
CREATE ROLE 'r1', 'r2';

GRANT 'u1' TO 'u1';
GRANT 'r1' TO 'r1';

GRANT 'r2' TO 'u2';
GRANT 'u2' TO 'r2';
```

TiDB 允许这种多层授权关系存在，可以使用多层授权关系实现权限继承。

3.1.4.1.4 查看角色拥有的权限

可以通过 `SHOW GRANTS` 语句查看用户被授予了哪些角色。当用户查看其他用户权限相关信息时，需要对 `mysql` 数据库拥有 `SELECT` 权限。

```
SHOW GRANTS FOR 'dev1'@'localhost';
```

```
+-----------------------------------------------+
| Grants for dev1@localhost                     |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO `dev1`@`localhost`      |
| GRANT `app_developer`@`%` TO `dev1`@`localhost` |
+-----------------------------------------------+
```

可以通过使用 `SHOW GRANTS` 的 `USING` 选项来查看角色对应的权限。

```
SHOW GRANTS FOR 'dev1'@'localhost' USING 'app_developer';
```

```
+-------------------------------------------------------+
| Grants for dev1@localhost                             |
+-------------------------------------------------------+
| GRANT USAGE ON *.* TO `dev1`@`localhost`              |
| GRANT ALL PRIVILEGES ON `app_db`.* TO `dev1`@`localhost` |
| GRANT `app_developer`@`%` TO `dev1`@`localhost`       |
+-------------------------------------------------------+
```

```
SHOW GRANTS FOR 'rw_user1'@'localhost' USING 'app_read', 'app_write';
```

```
+-----------------------------------------------------------------------------+
| Grants for rw_user1@localhost                                               |
+-----------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO `rw_user1`@`localhost`                                |
| GRANT SELECT, INSERT, UPDATE, DELETE ON `app_db`.* TO `rw_user1`@`localhost` |
| GRANT `app_read`@`%`,`app_write`@`%` TO `rw_user1`@`localhost`              |
+-----------------------------------------------------------------------------+
```

```
SHOW GRANTS FOR 'read_user1'@'localhost' USING 'app_read';
```

```
+-----------------------------------------------------------+
| Grants for read_user1@localhost                           |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO `read_user1`@`localhost`            |
| GRANT SELECT ON `app_db`.* TO `read_user1`@`localhost`    |
| GRANT `app_read`@`%` TO `read_user1`@`localhost`          |
+-----------------------------------------------------------+
```

可以使用 SHOW GRANTS 或 SHOW GRANTS FOR CURRENT_USER() 查看当前用户的权限。这两个语句有细微的差异，SHOW GRANTS 会显示当前用户的启用角色的权限，而 SHOW GRANTS FOR CURRENT_USER() 则不会显示启用角色的权限。

3.1.4.1.5 设置默认启用角色

角色在授予给用户之后，并不会生效；只有在用户启用了某些角色之后，才可以使用角色拥有的权限。

可以对用户设置默认启用的角色；用户在登录时，默认启用的角色会被自动启用。

```
SET DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}
    TO user [, user ]
```

比如将 app_read 和 app_wirte 设置为 rw_user1@localhost 的默认启用角色：

```
SET DEFAULT ROLE app_read, app_write TO 'rw_user1'@'localhost';
```

将 dev1@localhost 的所有角色，设为其默认启用角色：

```
SET DEFAULT ROLE ALL TO 'dev1'@'localhost';
```

关闭 dev1@localhost 的所有默认启用角色：

```
SET DEFAULT ROLE NONE TO 'dev1'@'localhost';
```

需要注意的是，设置为默认启用角色的角色必须已经授予给那个用户。

3.1.4.1.6 在当前 session 启用角色

除了使用 SET DEFAULT ROLE 启用角色外，TiDB 还提供让用户在当前 session 启用某些角色的功能。

```
SET ROLE {
    DEFAULT
  | NONE
  | ALL
  | ALL EXCEPT role [, role ] ...
  | role [, role ] ...
}
```

例如，登录 rw_user1 后，为当前用户启用角色 app_read 和 app_write ，仅在当前 session 有效：

```
SET ROLE 'app_read', 'app_write';
```

启用当前用户的默认角色：

```
SET ROLE DEFAULT
```

启用授予给当前用户的所有角色：

```
SET ROLE ALL
```

不启用任何角色：

```
SET ROLE NONE
```

启用除 app_read 外的角色：

```
SET ROLE ALL EXCEPT 'app_read'
```

> 注意：
>
> 使用 SET ROLE 启用的角色只有在当前 session 才会有效。

3.1.4.1.7 查看当前启用角色

当前用户可以通过 CURRENT_ROLE() 函数查看当前用户启用了哪些角色。

例如，先对 rw_user1'@'localhost 设置默认角色：

```
SET DEFAULT ROLE ALL TO 'rw_user1'@'localhost';
```

用 rw_user1@localhost 登录后：

```
SELECT CURRENT_ROLE();
```

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

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

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

3.1.4.1.8 收回角色

解除角色 app_read 与用户 read_user1@localhost、read_user2@localhost 的授权关系。

```
REVOKE 'app_read' FROM 'read_user1'@'localhost', 'read_user2'@'localhost';
```

解除角色 app_read、app_write 与用户 rw_user1@localhost 的授权关系。

```
REVOKE 'app_read', 'app_write' FROM 'rw_user1'@'localhost';
```

解除角色授权具有原子性，如果在撤销授权操作中失败会回滚。

3.1.4.1.9 收回权限

REVOKE 语句与 GRANT 对应，可以使用 REVOKE 来撤销 app_write 的权限。

```
REVOKE INSERT, UPDATE, DELETE ON app_db.* FROM 'app_write';
```

具体可参考TiDB 权限管理。

3.1.4.1.10 删除角色

删除角色 app_read 和 app_write：

```
DROP ROLE 'app_read', 'app_write';
```

这个操作会清除角色在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录，解除和其相关的授权关系。执行删除角色的用户需要拥有 DROP ROLE 或 DROP USER 权限。

3.1.4.1.11 授权表

在原有的四张系统权限表的基础上，角色访问控制引入了两张新的系统表：

- mysql.role_edges：记录角色与用户的授权关系
- mysql.default_roles：记录每个用户默认启用的角色

以下是 mysql.role_edges 所包含的数据。

```
select * from mysql.role_edges;
```

```
+-----------+-----------+---------+---------+-------------------+
| FROM_HOST | FROM_USER | TO_HOST | TO_USER | WITH_ADMIN_OPTION |
+-----------+-----------+---------+---------+-------------------+
| %         | r_1       | %       | u_1     | N                 |
+-----------+-----------+---------+---------+-------------------+
1 row in set (0.00 sec)
```

其中 FROM_HOST 和 FROM_USER 分别表示角色的主机名和用户名，TO_HOST 和 TO_USER 分别表示被授予角色的用户的主机名和用户名。

mysql.default_roles 中包含了每个用户默认启用了哪些角色。

```
select * from mysql.default_roles;
```

```
+------+------+-------------------+-------------------+
| HOST | USER | DEFAULT_ROLE_HOST | DEFAULT_ROLE_USER |
+------+------+-------------------+-------------------+
| %    | u_1  | %                 | r_1               |
| %    | u_1  | %                 | r_2               |
+------+------+-------------------+-------------------+
2 rows in set (0.00 sec)
```

HOST 和 USER 分别表示用户的主机名和用户名，DEFAULT_ROLE_HOST 和 DEFAULT_ROLE_USER 分别表示默认启用的角色的主机名和用户名。

3.1.4.1.12 其他

由于基于角色的访问控制模块和用户管理以及权限管理结合十分紧密，因此需要参考一些操作的细节：

- TiDB 权限管理
- TiDB 用户账户管理

3.1.5 TiDB 证书鉴权使用指南

TiDB 支持基于证书鉴权的登录方式。采用这种方式，TiDB 对不同用户签发证书，使用加密连接来传输数据，并在用户登录时验证证书。相比 MySQL 用户常用的用户名密码验证方式，与 MySQL 相兼容的证书鉴权方式更安全，因此越来越多的用户使用证书鉴权来代替用户名密码验证。

在 TiDB 上使用证书鉴权的登录方法，可能需要进行以下操作：

- 创建安全密钥和证书
- 配置 TiDB 和客户端使用的证书
- 配置登录时需要校验的用户证书信息
- 更新和替换证书

本文介绍了如何进行证书鉴权的上述几个操作。

3.1.5.1 创建安全密钥和证书

目前推荐使用 OpenSSL 来生成密钥和证书，生成证书的过程和为 TiDB 客户端服务端间通信开启加密传输过程类似，下面更多演示如何在证书中配置更多需校验的属性字段。

3.1.5.1.1 生成 CA 密钥和证书

1. 执行以下命令生成 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

命令执行后输出以下结果：

```
Generating RSA private key, 2048 bit long modulus (2 primes)
...................+++++
.............................................+++++
e is 65537 (0x010001)
```

2. 执行以下命令生成该密钥对应的证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.pem
```

3. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:TiDB admin
Email Address []:s@pingcap.com
```

> 注意：
> 以上信息中，: 后的文字为用户输入的信息。

3.1.5.1.2 生成服务端密钥和证书

1. 执行以下命令生成服务端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.pem -out server-req
    ↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiKV
Common Name (e.g. server FQDN or YOUR name) []:TiKV Test Server
```

```
Email Address []:k@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成服务端的 RSA 密钥：

```
sudo openssl rsa -in server-key.pem -out server-key.pem
```

输出结果如下：

```
writing RSA key
```

4. 使用 CA 证书签名来生成服务端的证书：

```
sudo openssl x509 -req -in server-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    ↪ set_serial 01 -out server-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiKV, CN = TiKV
    ↪ Test Server, emailAddress = k@pingcap.com
Getting CA Private Key
```

> 注意：
> 以上结果中，用户登录时 TiDB 将强制检查 `subject` 部分的信息是否一致。

3.1.5.1.3 生成客户端密钥和证书

生成服务端密钥和证书后，需要生成客户端使用的密钥和证书。通常需要为不同的用户生成不同的密钥和证书。

1. 执行以下命令生成客户端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.pem -out client-req
    ↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
```

```
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:tpch-user1
Email Address []:zz@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成客户端 RSA 证书：

```
sudo openssl rsa -in client-key.pem -out client-key.pem
```

以上命令的输出结果如下：

```
writing RSA key
```

4. 执行以下命令，使用 CA 证书签名来生成客户端证书：

```
sudo openssl x509 -req -in client-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    ↪ set_serial 01 -out client-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiDB, CN = tpch-
    ↪ user1, emailAddress = zz@pingcap.com
Getting CA Private Key
```

> 注意：
> 以上结果中，`subject` 部分后的信息会被用来在 `require` 中配置和要求验证。

3.1.5.1.4 验证证书

执行以下命令验证证书：

```
openssl verify -CAfile ca-cert.pem server-cert.pem client-cert.pem
```

如果验证通过，会显示以下信息：

```
server-cert.pem: OK
client-cert.pem: OK
```

3.1.5.2 配置 TiDB 和客户端使用证书

在生成证书后，需要在 TiDB 中配置服务端所使用的证书，同时让客户端程序使用客户端证书。

3.1.5.2.1 配置 TiDB 服务端

修改 TiDB 配置文件中的 `[security]` 段。这一步指定 CA 证书、服务端密钥和服务端证书存放的路径。可将 `path/to/server-cert.pem`、`path/to/server-key.pem` 和 `path/to/ca-cert.pem` 替换成实际的路径。

```
[security]
ssl-cert ="path/to/server-cert.pem"
ssl-key ="path/to/server-key.pem"
ssl-ca="path/to/ca-cert.pem"
```

启动 TiDB 日志。如果日志中有以下内容，即代表配置生效：

```
[INFO] [server.go:264] ["secure connection is enabled"] ["client verification enabled"=true]
```

3.1.5.2.2 配置客户端程序

配置客户端程序，让客户端使用客户端密钥和证书来登录 TiDB。

以 MySQL 客户端为例，可以通过指定 `ssl-cert`、`ssl-key`、`ssl-ca` 来使用新的 CA 证书、客户端密钥和证书：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/client-
    ↪ key.new.pem --ssl-ca /path/to/ca-cert.pem
```

> 注意：
>
> `/path/to/client-cert.new.pem`、`/path/to/client-key.new.pem` 和 `/path/to/ca-cert.pem` 是 CA 证书、客户端密钥和客户端存放的路径。可将以上命令中的这些部分替换为实际的路径。

3.1.5.3 配置登录时需要校验的用户证书信息

使用客户端连接 TiDB 进行授权配置。先获取需要验证的用户证书信息，再对这些信息进行配置。

3.1.5.3.1 获取用户证书信息

用户证书信息可由 `require subject`、`require issuer`、`require san` 和 `require cipher` 来指定，用于检查 X.509 certificate attributes。

- `require subject`：指定用户在连接时需要提供客户端证书的 `subject` 内容。指定该选项后，不需要再配置 `require ssl` 或 x509。配置内容对应生成客户端密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

  ```
  openssl x509 -noout -subject -in client-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed
      ↪  's/ = /=/g' | sed 's/^/\//'
  ```

- `require issuer`：指定签发用户证书的 CA 证书的 `subject` 内容。配置内容对应生成 CA 密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

  ```
  openssl x509 -noout -subject -in ca-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed 's/
      ↪  = /=/g' | sed 's/^/\//'
  ```

- `require san`：指定签发用户证书的 CA 证书的 `Subject Alternative Name` 内容。配置内容对应生成客户端证书使用的openssl.cnf 配置文件的 `alt_names` 信息。

  - 可以执行以下命令来获取已生成证书中的 `require san` 项的信息：

    ```
    openssl x509 -noout -extensions subjectAltName -in client.crt
    ```

  - `require san` 目前支持以下 `Subject Alternative Name` 检查项：

    * URI
    * IP
    * DNS

  - 多个检查项可通过逗号连接后进行配置。例如，对用户 u1 进行以下配置：

    ```
    create user 'u1'@'%' require san 'DNS:d1,URI:spiffe://example.org/myservice1,URI:spiffe
        ↪ ://example.org/myservice2'
    ```

    以上配置只允许用户 u1 使用 URI 项为 `spiffe://example.org/myservice1` 或 `spiffe://example.org` ↪ `/myservice2`、DNS 项为 d1 的证书登录 TiDB。

- `require cipher`：配置该项检查客户端支持的 `cipher method`。可以使用以下语句来查看支持的列表：

  ```
  SHOW SESSION STATUS LIKE 'Ssl_cipher_list';
  ```

3.1.5.3.2 配置用户证书信息

获取用户证书信息（`require subject`, `require issuer`、`require san` 和 `require cipher`）后，可在创建用户、赋予权限或更改用户时配置用户证书信息。将以下命令中的 `<replaceable>` 替换为对应的信息。可以选择配置其中一项或多项，使用空格或 and 分隔。

- 可以在创建用户 (`create user`) 时配置登录时需要校验的证书信息：

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

- 可以在赋予权限 (grant) 时配置登录时需要校验的证书信息：

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

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

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

配置完成后，用户在登录时 TiDB 会验证以下内容：

- 使用 SSL 登录，且证书为服务器配置的 CA 证书所签发
- 证书的 Issuer 信息和权限配置里的信息相匹配
- 证书的 Subject 信息和权限配置里的信息相匹配
- 证书的 Subject Alternative Name 信息和权限配置里的信息相匹配

全部验证通过后用户才能登录，否则会报 ERROR 1045 (28000): Access denied 错误。登录后，可以通过以下命令来查看当前链接是否使用证书登录、TLS 版本和 Cipher 算法。

连接 MySQL 客户端并执行：

```
\s
```

返回结果如下：

```
--------------
mysql  Ver 15.1 Distrib 10.4.10-MariaDB, for Linux (x86_64) using readline 5.1

Connection id:       1
Current database:    test
Current user:        root@127.0.0.1
SSL:                 Cipher in use is TLS_AES_256_GCM_SHA384
```

然后执行：

```
show variables like '%ssl%';
```

返回结果如下：

```
+---------------+---------------------------------+
| Variable_name | Value                           |
+---------------+---------------------------------+
| ssl_cert      | /path/to/server-cert.pem        |
| ssl_ca        | /path/to/ca-cert.pem            |
| have_ssl      | YES                             |
| have_openssl  | YES                             |
| ssl_key       | /path/to/server-key.pem         |
+---------------+---------------------------------+
6 rows in set (0.067 sec)
```

3.1.5.4 更新和替换证书

证书和密钥通常会周期性更新。下文介绍更新密钥和证书的流程。

CA 证书是客户端和服务端相互校验的依据，所以如果需要替换 CA 证书，则需要生成一个组合证书来在替换期间同时支持客户端和服务器上新旧证书的验证，并优先替换客户端和服务端的 CA 证书，再替换客户端和服务端的密钥和证书。

3.1.5.4.1 更新 CA 密钥和证书

1. 以替换 CA 密钥为例（假设 `ca-key.pem` 被盗），将旧的 CA 密钥和证书进行备份：

```
mv ca-key.pem ca-key.old.pem && \
mv ca-cert.pem ca-cert.old.pem
```

2. 生成新的 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

3. 用新的密钥生成新的 CA 证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.new.pem
```

> **注意：**
>
> 生成新的 CA 证书是为了替换密钥和证书，保证在线用户不受影响。所以以上命令中填写的附加信息必须与已配置的 `require issuer` 信息一致。

4. 生成组合 CA 证书：

```
cat ca-cert.new.pem ca-cert.old.pem > ca-cert.pem
```

之后使用新生成的组合 CA 证书并重启 TiDB Server，此时服务端可以同时接受和使用新旧 CA 证书。

之后先将所有客户端用的 CA 证书也替换为新生成的组合 CA 证书，使客户端能同时和使用新旧 CA 证书。

3.1.5.4.2 更新客户端密钥和证书

> **注意：**
>
> 需要将集群中所有服务端和客户端使用的 CA 证书都替换为新生成的组合 CA 证书后才能开始进行以下步骤。

1. 生成新的客户端 RSA 密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.new.pem -out client
    ↪ -req.new.pem && \
sudo openssl rsa -in client-key.new.pem -out client-key.new.pem
```

> 注意：
> 以上命令是为了替换密钥和证书，保证在线用户不受影响，所以以上命令中填写的附加信息必须与已配置的 `require subject` 信息一致。

2. 使用新的组合 CA 证书和新 CA 密钥生成新客户端证书：

```
sudo openssl x509 -req -in client-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem
    ↪  -set_serial 01 -out client-cert.new.pem
```

3. 让客户端使用新的客户端密钥和证书来连接 TiDB（以 MySQL 客户端为例）：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/
    ↪ client-key.new.pem --ssl-ca /path/to/ca-cert.pem
```

> 注意：
> `/path/to/client-cert.new.pem`、`/path/to/client-key.new.pem` 和 `/path/to/ca-cert.pem` 是 CA 证书、客户端密钥和客户端存放的路径。可将以上命令中的这些部分替换为实际的路径。

3.1.5.4.3 更新服务端密钥和证书

1. 生成新的服务端 RSA 密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.new.pem -out server
    ↪ -req.new.pem && \
sudo openssl rsa -in server-key.new.pem -out server-key.new.pem
```

2. 使用新的组合 CA 证书和新 CA 密钥生成新服务端证书：

```
sudo openssl x509 -req -in server-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem
    ↪  -set_serial 01 -out server-cert.new.pem
```

3. 配置 TiDB 使用上面新生成的服务端密钥和证书并重启。参见配置 TiDB 服务端。

3.2 安全加固

3.2.1 为 TiDB 客户端服务端间通信开启加密传输

TiDB 服务端与客户端之间默认采用非加密连接，因而具备监视信道流量能力的第三方可以知悉 TiDB 服务端与客户端之间发送和接受的数据，包括但不限于查询语句内容、查询结果等。若信道是不可信的，例如客户端是通过公网连接到 TiDB 服务端的，则非加密连接容易造成信息泄露，建议使用加密连接确保安全性。

TiDB 服务端支持启用基于 TLS（传输层安全）协议的加密连接，协议与 MySQL 加密连接一致，现有 MySQL Client 如 MySQL Shell 和 MySQL 驱动等能直接支持。TLS 的前身是 SSL，因而 TLS 有时也被称为 SSL，但由于 SSL 协议有已知安全漏洞，TiDB 实际上并未支持。TiDB 支持的 TLS/SSL 协议版本为 TLSv1.0、TLSv1.1、TLSv1.2 和 TLSv1.3。

使用加密连接后，连接将具有以下安全性质：

- 保密性：流量明文被加密，无法被窃听；
- 完整性：流量明文无法被篡改；
- 身份验证（可选）：客户端和服务端能验证双方身份，避免中间人攻击。

要为 TiDB 客户端与服务端间的通信开启 TLS 加密传输，首先需要在 TiDB 服务端通过配置开启 TLS 加密连接的支持，然后通过配置客户端应用程序使用 TLS 加密连接。一般情况下，如果服务端正确配置了 TLS 加密连接支持，客户端库都会自动启用 TLS 加密传输。

另外，与 MySQL 相同，TiDB 也支持在同一 TCP 端口上开启 TLS 连接或非 TLS 连接。对于开启了 TLS 连接支持的 TiDB 服务端，客户端既可以选择通过加密连接安全地连接到该 TiDB 服务端，也可以选择使用普通的非加密连接。如需使用加密连接，你可以通过以下方式进行配置：

- 通过配置系统变量 `require_secure_transport` 要求所有用户必须使用加密连接来连接到 TiDB。
- 通过在创建用户 (`create user`)，或修改已有用户 (`alter user`) 时指定 `REQUIRE SSL` 要求指定用户必须使用加密连接来连接 TiDB。以创建用户为例：

```
CREATE USER 'u1'@'%' IDENTIFIED BY 'my_random_password' REQUIRE SSL;
```

> 注意：
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户必须使用加密连接连接 TiDB。

3.2.1.1 配置 TiDB 服务端启用安全连接

要启用安全连接，请参考以下相关参数说明：

- `auto-tls`: 启用证书自动生成功能（从 v5.2.0 开始）
- `ssl-cert`：指定 SSL 证书文件路径
- `ssl-key`：指定证书文件对应的私钥

- ssl-ca：可选，指定受信任的 CA 证书文件路径
- tls-version：可选，指定最低 TLS 版本，例如 TLSv1.2

auto tls 支持安全连接，但不提供客户端证书验证。有关证书验证和控制证书生成方式的说明，请参考下面配置 ssl-cert，ssl-key 和 ssl-ca 变量的建议：

- 在启动 TiDB 时，至少需要在配置文件中同时指定 ssl-cert 和 ssl-key 参数，才能在 TiDB 服务端开启安全连接。还可以指定 ssl-ca 参数进行客户端身份验证（请参见配置启用身份验证章节）。
- 参数指定的文件都为 PEM 格式。另外目前 TiDB 尚不支持加载有密码保护的私钥，因此必须提供一个没有密码的私钥文件。若提供的证书或私钥无效，则 TiDB 服务端将照常启动，但并不支持客户端加密连接到 TiDB 服务端。
- 若证书参数无误，则 TiDB 在启动时将会输出 secure connection is enabled，否则 TiDB 会输出 secure connection is NOT ENABLED。

> 注意：
>
> v5.2.0 版本之前，你可以使用 mysql_ssl_rsa_setup --datadir=./certs 生成证书。mysql_ssl_rsa_setup 工具是 MySQL Server 的一部分。

3.2.1.2 配置 MySQL Client 使用安全连接

MySQL 5.7 及以上版本自带的客户端默认尝试使用安全连接，若服务端不支持安全连接则自动退回到使用非安全连接；MySQL 5.7 以下版本自带的客户端默认采用非安全连接。

可以通过命令行参数修改客户端的连接行为，包括：

- 强制使用安全连接，若服务端不支持安全连接则连接失败 (--ssl-mode=REQUIRED)
- 尝试使用安全连接，若服务端不支持安全连接则退回使用不安全连接
- 使用不安全连接 (--ssl-mode=DISABLED)

除此参数外，MySQL 8.0 客户端有两种 SSL 模式：

- --ssl-mode=VERIFY_CA: 根据 --ssl-ca 签发的 CA 验证来自服务器的证书。
- --ssl-mode=VERIFY_IDENTITY: 与 VERIFY_CA 相同，但也验证所连接的主机名是否与证书匹配。

详细信息请参阅 MySQL 文档中关于客户端配置安全连接的部分。

3.2.1.3 配置启用身份验证

若在 TiDB 服务端或 MySQL Client 中未指定 ssl-ca 参数，则默认不会进行客户端或服务端身份验证，无法抵御中间人攻击，例如客户端可能会“安全地”连接到了一个伪装的服务端。可以在服务端和客户端中配置 ssl-ca 参数进行身份验证。一般情况下只需验证服务端身份，但也可以验证客户端身份进一步增强安全性。

- 若要使 MySQL Client 验证 TiDB 服务端身份，TiDB 服务端需至少配置 `ssl-cert` 和 `ssl-key` 参数，客户端需至少指定 `--ssl-ca` 参数，且 `--ssl-mode` 至少为 `VERIFY_CA`。必须确保 TiDB 服务端配置的证书 (`ssl-cert`) 是由客户端 `--ssl-ca` 参数所指定的 CA 所签发的，否则身份验证失败。
- 若要使 TiDB 服务端验证 MySQL Client 身份，TiDB 服务端需配置 `ssl-cert`、`ssl-key`、`ssl-ca` 参数，客户端需至少指定 `--ssl-cert`、`--ssl-key` 参数。必须确保服务端配置的证书和客户端配置的证书都是由服务端配置指定的 `ssl-ca` 签发的。
- 若要进行双向身份验证，请同时满足上述要求。

默认情况，服务端对客户端的身份验证是可选的。若客户端在 TLS 握手时未出示自己的身份证书，也能正常建立 TLS 连接。但也可以通过在创建用户 (`create user`)，赋予权限 (`grant`) 或修改已有用户 (`alter user`) 时指定 `require 509` 要求客户端需进行身份验证，以创建用户为例：

```
create user 'u1'@'%'  require x509;
```

> **注意：**
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户需进行身份验证。

3.2.1.4 检查当前连接是否是加密连接

可以通过 `SHOW STATUS LIKE "%Ssl%";` 了解当前连接的详细情况，包括是否使用了安全连接、安全连接采用的加密协议、TLS 版本号等。

以下是一个安全连接中执行该语句的结果。由于客户端支持的 TLS 版本号和加密协议会有所不同，执行结果相应地也会有所变化。

```
SHOW STATUS LIKE "%Ssl%";
```

```
......
| Ssl_verify_mode | 5                          |
| Ssl_version     | TLSv1.2                    |
| Ssl_cipher      | ECDHE-RSA-AES128-GCM-SHA256  |
......
```

除此以外，对于 MySQL 自带客户端，还可以使用 `STATUS` 或 `\s` 语句查看连接情况：

```
\s
```

```
...
SSL: Cipher in use is ECDHE-RSA-AES128-GCM-SHA256
...
```

3.2.1.5 支持的 TLS 版本及密钥交换协议和加密算法

TiDB 支持的 TLS 版本及密钥交换协议和加密算法由 Golang 官方库决定。

你使用的客户端库和操作系统加密策略也可能会影响到支持的协议版本和密码套件。

3.2.1.5.1 支持的 TLS 版本

- TLSv1.0（默认禁用）
- TLSv1.1
- TLSv1.2
- TLSv1.3

可以使用配置项 `tls-version` 来限制 TLS 版本。

实际可用的 TLS 版本取决于操作系统的加密策略、MySQL 客户端版本和客户端的 SSL/TLS 库。

3.2.1.5.2 支持的密钥交换协议及加密算法

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

3.2.1.6 重加载证书、密钥和 CA

在需要替换证书、密钥或 CA 时，可以在完成对应文件替换后，对运行中的 TiDB 实例执行`ALTER INSTANCE RELOAD TLS` 语句从原配置的证书 (`ssl-cert`)、密钥 (`ssl-key`) 和 CA (`ssl-ca`) 的路径重新加证书、密钥和 CA，而无需重启 TiDB 实例。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

3.2.1.7 监控

自 TiDB v5.2.0 版本起，你可以使用 `Ssl_server_not_after` 和 `Ssl_server_not_before` 状态变量监控证书有效期的起止时间。

```
SHOW GLOBAL STATUS LIKE 'Ssl\_server\_not\_%';
```

```
+----------------------+--------------------------+
| Variable_name        | Value                    |
+----------------------+--------------------------+
| Ssl_server_not_after  | Nov 28 06:42:32 2021 UTC |
| Ssl_server_not_before | Aug 30 06:42:32 2021 UTC |
+----------------------+--------------------------+
2 rows in set (0.0076 sec)
```

3.2.1.8 另请参阅

- 为 TiDB 组件间通信开启加密传输

3.2.2 为 TiDB 组件间通信开启加密传输

本部分介绍如何为 TiDB 集群内各部组件间开启加密传输，一旦开启以下组件间均将使用加密传输：

- TiDB 与 TiKV、 PD
- TiKV 与 PD
- TiDB Control 与 TiDB，TiKV Control 与 TiKV，PD Control 与 PD
- TiKV、 PD、 TiDB 各自集群内内部通讯

目前暂不支持只开启其中部分组件的加密传输。

3.2.2.1 配置开启加密传输

1. 准备证书。

   推荐为 TiDB、 TiKV、 PD 分别准备一个 Server 证书，并保证可以相互验证，而它们的 Control 工具则可选择共用 Client 证书。

   有多种工具可以生成自签名证书，如 `openssl`，`easy-rsa`，`cfssl`。

   这里提供一个使用 `openssl` 生成证书的示例：生成自签名证书。

2. 配置证书。

   - TiDB

     在 `config` 文件或命令行参数中设置：

```
[security]
# Path of file that contains list of trusted SSL CAs for connection with cluster
    ↪ components.
cluster-ssl-ca = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format for connection with cluster
    ↪  components.
cluster-ssl-cert = "/path/to/tidb-server.pem"
# Path of file that contains X509 key in PEM format for connection with cluster
    ↪ components.
cluster-ssl-key = "/path/to/tidb-server-key.pem"
```

- TiKV

  在 config 文件或命令行参数中设置，并设置相应的 URL 为 https：

```
[security]
# set the path for certificates. Empty string means disabling secure connectoins.
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/tikv-server.pem"
key-path = "/path/to/tikv-server-key.pem"
```

- PD

  在 config 文件或命令行参数中设置，并设置相应的 URL 为 https：

```
[security]
# Path of file that contains list of trusted SSL CAs. if set, following four settings
    ↪ shouldn't be empty
cacert-path = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format.
cert-path = "/path/to/pd-server.pem"
# Path of file that contains X509 key in PEM format.
key-path = "/path/to/pd-server-key.pem"
```

- TiFlash（从 v4.0.5 版本开始引入）

  在 tiflash.toml 文件中设置，将 http_port 项改为 https_port:

```
[security]
# Path of file that contains list of trusted SSL CAs. if set, following four settings
    ↪ shouldn't be empty
ca_path = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format.
cert_path = "/path/to/tiflash-server.pem"
# Path of file that contains X509 key in PEM format.
key_path = "/path/to/tiflash-server-key.pem"
```

  在 tiflash-learner.toml 文件中设置，

```
[security]
# Sets the path for certificates. The empty string means that secure connections are
    ↪ disabled.
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/tiflash-server.pem"
key-path = "/path/to/tiflash-server-key.pem"
```

- TiCDC

  在启动命令行中设置，并设置相应的 URL 为 `https`：

```
cdc server --pd=https://127.0.0.1:2379 --log-file=ticdc.log --addr=0.0.0.0:8301 --
    ↪ advertise-addr=127.0.0.1:8301 --ca=/path/to/ca.pem --cert=/path/to/ticdc-cert.
    ↪ pem --key=/path/to/ticdc-key.pem
```

此时 TiDB 集群各个组件间已开启加密传输。

> 注意：
>
> 若 TiDB 集群各个组件间开启加密传输后，在使用 tidb-ctl、tikv-ctl 或 pd-ctl 工具连接集群时，需要指定 client 证书，示例：

```
./tidb-ctl -u https://127.0.0.1:10080 --ca /path/to/ca.pem --ssl-cert /path/to/client.pem --
    ↪ ssl-key /path/to/client-key.pem
```

```
tiup ctl pd -u https://127.0.0.1:2379 --cacert /path/to/ca.pem --cert /path/to/client.pem --
    ↪ key /path/to/client-key.pem
```

```
./tikv-ctl --host="127.0.0.1:20160" --ca-path="/path/to/ca.pem" --cert-path="/path/to/client
    ↪ .pem" --key-path="/path/to/clinet-key.pem"
```

3.2.2.2 认证组件调用者身份

通常被调用者除了校验调用者提供的密钥、证书和 CA 有效性外，还需要校验调用方身份以防止拥有有效证书的非法访问者进行访问（例如：TiKV 只能被 TiDB 访问，需阻止拥有合法证书但非 TiDB 的其他访问者访问 TiKV）。

如希望进行组件调用者身份认证，需要在生证书时通过 Common Name 标识证书使用者身份，并在被调用者配置检查证书 Common Name 列表来检查调用者身份。

- TiDB

  在 `config` 文件或命令行参数中设置：

```
[security]
cluster-verify-cn = [
  "TiDB-Server",
  "TiKV-Control",
]
```

- TiKV

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = [
    "TiDB-Server", "PD-Server", "TiKV-Control", "RawKvClient1",
]
```

- PD

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = ["TiKV-Server", "TiDB-Server", "PD-Control"]
```

- TiCDC

  在启动命令行中设置：

```
cdc server --pd=https://127.0.0.1:2379 --log-file=ticdc.log --addr=0.0.0.0:8301 --advertise-
    ↪ addr=127.0.0.1:8301 --ca=/path/to/ca.pem --cert=/path/to/ticdc-cert.pem --key=/path/
    ↪ to/ticdc-key.pem --cert-allowed-cn="client1,client2"
```

- TiFlash（从 v4.0.5 版本开始引入）

  在 tiflash.toml 文件中设置：

```
[security]
cert_allowed_cn = ["TiKV-Server", "TiDB-Server"]
```

  在 tiflash-learner.toml 文件中设置：

```
[security]
cert-allowed-cn = ["PD-Server", "TiKV-Server", "TiFlash-Server"]
```

3.2.2.3 证书重加载

TiDB、PD 和 TiKV 和各种 Client 都会在每次新建相互通讯的连接时重新读取当前的证书和密钥文件内容，实现证书和密钥的重加载。目前暂不支持 CA 的重加载。

3.2.2.4 另请参阅

- 为 TiDB 客户端服务端间通信开启加密传输

3.2.3 生成自签名证书

> **注意：**
>
> 要在 TiDB 客户端与服务端间通信开启加密传输，你只需配置 `auto-tls` 参数。

本文档提供使用 `openssl` 生成自签名证书的一个示例，用户也可以根据自己的需求生成符合要求的证书和密钥。

假设实例集群拓扑如下：

Name	Host IP	Services
node1	172.16.10.11	PD1, TiDB1
node2	172.16.10.12	PD2
node3	172.16.10.13	PD3
node4	172.16.10.14	TiKV1
node5	172.16.10.15	TiKV2
node6	172.16.10.16	TiKV3

3.2.3.1 安装 OpenSSL

对于 Debian 或 Ubuntu 操作系统：

```
apt install openssl
```

对于 RedHat 或 CentOS 操作系统：

```
yum install openssl
```

也可以参考 OpenSSL 官方的下载文档进行安装。

3.2.3.2 生成 CA 证书

CA 的作用是签发证书。实际情况中，请联系你的管理员签发证书或者使用信任的 CA 机构。CA 会管理多个证书对，这里只需生成原始的一对证书，步骤如下：

1. 生成 root 密钥：

    ```
    openssl genrsa -out root.key 4096
    ```

2. 生成 root 证书：

    ```
    openssl req -new -x509 -days 1000 -key root.key -out root.crt
    ```

3. 验证 root 证书：

    ```
    openssl x509 -text -in root.crt -noout
    ```

3.2.3.3 签发各个组件的证书

3.2.3.3.1 集群中可能使用到的证书

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

3.2.3.3.2 给 TiKV 实例签发证书

给 TiKV 实例签发证书的步骤如下：

1. 生成该证书对应的私钥：

```
openssl genrsa -out tikv.key 2048
```

2. 拷贝一份 OpenSSL 的配置模板文件。

   模板文件可能存在多个位置，请以实际位置为准：

```
cp /usr/lib/ssl/openssl.cnf .
```

   如果不知道实际位置，请在根目录下查找：

```
find / -name openssl.cnf
```

3. 编辑 `openssl.cnf`，在 `[ req ]` 字段下加入 `req_extensions = v3_req`，然后在 `[ v3_req ]` 字段下加入 `subjectAltName = @alt_names`。最后新建一个字段，并编辑 SAN 的信息：

```
[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 172.16.10.14
IP.3 = 172.16.10.15
IP.4 = 172.16.10.16
```

4. 保存 `openssl.cnf` 文件后，生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，各个组件默认不会开启验证，需要在配置文件中启用该功能才生效）：

```
openssl req -new -key tikv.key -out tikv.csr -config openssl.cnf
```

5. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in tikv.csr -out
    ↪ tikv.crt -extensions v3_req -extfile openssl.cnf
```

6. 验证证书携带 SAN 字段信息（可选）：

```
openssl x509 -text -in tikv.crt -noout
```

7. 确认在当前目录下得到如下文件：

```
root.crt
tikv.crt
tikv.key
```

为其它 TiDB 组件签发证书的过程类似，此文档不再赘述。

3.2.3.3.3 为客户端签发证书

为客户端签发证书的步骤如下。

1. 生成该证书对应的私钥：

```
openssl genrsa -out client.key 2048
```

2. 生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，默认不会开启对各个组件的验证，需要在配置文件中启用该功能才生效）

```
openssl req -new -key client.key -out client.csr
```

3. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in client.csr -out
    ↪  client.crt
```

3.2.4 静态加密

> **注意：**
>
> 如果集群部署在 AWS 上并在使用 EBS (Elastic Block Store) 存储数据，建议使用 EBS 加密，详细信息请参考 AWS 文档 - EBS 加密。如果集群部署在 AWS 上，但未使用 EBS 存储（例如使用本地 NVMe 存储），则建议使用本文中介绍的静态加密方式。

静态加密 (encryption at rest) 即在存储数据时进行数据加密。对于数据库，静态加密功能也叫透明数据加密 (TDE)，区别于传输数据加密 (TLS) 或使用数据加密（很少使用）。SSD 驱动器、文件系统、云供应商等都可进行静态加密。但区别于这些加密方式，若 TiKV 在存储数据前就进行数据加密，攻击者则必须通过数据库的身份验证才能访问数据。例如，即使攻击者获得物理机的访问权限时，也无法通过复制磁盘上的文件来访问数据。

3.2.4.1 各 TiDB 组件支持的加密方式

在一个 TiDB 集群中，不同的组件使用不同的加密方式。本节介绍 TiKV、TiFlash、PD 和 Backup & Restore (BR) 等不同 TiDB 组件支持的加密方式。

部署好一个 TiDB 集群后，大部分用户数据会存储在 TiKV 和 TiFlash 节点上。此外，一些元数据会存储在 PD 节点上，例如用作为 TiKV Region 边界的二级索引密钥。为了能够充分发挥静态加密的优势，所有组件都需要启用加密功能。此外，进行加密时，也需要对备份、日志文件和通过网络传输的数据进行加密。

3.2.4.1.1 TiKV

TiKV 支持静态加密，即在 CTR 模式下使用 AES 对数据文件进行透明加密。要启用静态加密，用户需提供一个加密密钥，即主密钥。TiKV 自动轮换 (rotate) 用于加密实际数据文件的密钥，主密钥则可以由用户手动轮换。请注意，静态加密仅加密静态数据（即磁盘上的数据），而不加密网络传输中的数据。建议 TLS 与静态加密一起使用。

可以选择将 AWS KMS (Key Management Service) 用于云上部署或本地部署，也可以指定将密钥以明文形式存储在文件中。

TiKV 当前不从核心转储 (core dumps) 中排除加密密钥和用户数据。建议在使用静态加密时禁用 TiKV 进程的核心转储，该功能目前无法由 TiKV 独立处理。

TiKV 使用文件的绝对路径来跟踪已加密的数据文件。一旦 TiKV 节点开启了加密功能，用户就不应更改数据文件的路径配置，例如 `storage.data-dir`，`raftstore.raftdb-path`，`rocksdb.wal-dir` 和 `raftdb.wal-dir`。

3.2.4.1.2 TiFlash

TiFlash 支持静态加密。数据密钥由 TiFlash 生成。TiFlash（包括 TiFlash Proxy）写入的所有文件，包括数据文件、Schema 文件、临时文件等，均由当前数据密钥加密。TiFlash 支持的加密算法、加密配置方法（配置项在 `tiflash-learner.toml` 中）和监控项含义等均与 TiKV 一致。

如果 TiFlash 中部署了 Grafana 组件，可以查看 TiFlash-Proxy-Details -> Encryption。

3.2.4.1.3 PD

PD 的静态加密为实验特性，其配置方式与 TiKV 相同。

3.2.4.1.4 BR 备份

BR 支持对备份到 S3 的数据进行 S3 服务端加密 (SSE)。BR S3 服务端加密也支持使用用户自行创建的 AWS KMS 密钥进行加密，详细信息请参考BR S3 服务端加密。

3.2.4.1.5 日志

TiKV，TiDB 和 PD 信息日志中可能包含用于调试的用户数据。信息日志不会被加密，建议开启日志脱敏功能。

3.2.4.2 TiKV 静态加密

TiKV 当前支持的加密算法包括 AES128-CTR、AES192-CTR 和 AES256-CTR。TiKV 使用信封加密 (envelop encryption)，所以启用加密后，TiKV 使用以下两种类型的密钥：

- 主密钥 (master key)：主密钥由用户提供，用于加密 TiKV 生成的数据密钥。用户在 TiKV 外部进行主密钥的管理。
- 数据密钥 (data key)：数据密钥由 TiKV 生成，是实际用于加密的密钥。

多个 TiKV 实例可共用一个主密钥。在生产环境中，推荐通过 AWS KMS 提供主密钥。首先通过 AWS KMS 创建用户主密钥 (CMK)，然后在配置文件中将 CMK 密钥的 ID 提供给 TiKV。TiKV 进程在运行时可以通过 IAM 角色访问 KMS CMK。如果 TiKV 无法访问 KMS CMK，TiKV 就无法启动或重新启动。详情参阅 AWS 文档中的 KMS and IAM。

用户也可以通过文件形式提供主密钥。该文件须包含一个用十六进制字符串编码的 256 位（32 字节）密钥，并以换行符结尾（即 `\n`），且不包含其他任何内容。将密钥存储在磁盘上会泄漏密钥，因此密钥文件仅适合存储在 RAM 内存的 `tempfs` 中。

数据密钥由 TiKV 传递给底层存储引擎（即 RocksDB）。RocksDB 写入的所有文件，包括 SST 文件，WAL 文件和 MANIFEST 文件，均由当前数据密钥加密。TiKV 使用的其他临时文件（可能包括用户数据）也由相同的数据密钥加密。默认情况下，TiKV 每周自动轮换数据密钥，但是该时间段是可配置的。密钥轮换时，TiKV 不会重写全部现有文件来替换密钥，但如果集群的写入量恒定，则 RocksDB compaction 会将使用最新的数据密钥对数据重新加密。TiKV 跟踪密钥和加密方法，并使用密钥信息对读取的内容进行解密。

无论用户配置了哪种数据加密方法，数据密钥都使用 AES256-GCM 算法进行加密，以方便对主密钥进行验证。所以当使用文件而不是 KMS 方式指定主密钥时，主密钥必须为 256 位（32 字节）。

3.2.4.2.1 创建密钥

如需在 AWS 上创建一个密钥，请执行以下步骤：

1. 进入 AWS 控制台的 AWS KMS。
2. 确保在控制台的右上角选择正确的区域。
3. 点击创建密钥，选择对称 (Symmetric) 作为密钥类型。
4. 为钥匙设置一个别名。

你也可以使用 AWS CLI 执行该操作：

```
aws --region us-west-2 kms create-key
aws --region us-west-2 kms create-alias --alias-name "alias/tidb-tde" --target-key-id 0987dcba-09
    ↪ fe-87dc-65ba-ab0987654321
```

需要在第二条命令中输入的 `--target-key-id` 是第一条命令的结果。

3.2.4.2.2 配置加密

要启用加密，你可以在 TiKV 和 PD 的配置文件中添加加密部分：

```
[security.encryption]
data-encryption-method = "aes128-ctr"
data-key-rotation-period = "168h" # 7 days
```

data-encryption-method 的可选值为 "aes128-ctr"、"aes192-ctr"、"aes256-ctr" 和 "plaintext"。默认值为 "plaintext"，即默认不开启加密功能。data-key-rotation-period 指定 TiKV 轮换密钥的频率。可以为新 TiKV 群集或现有 TiKV 群集开启加密，但只有启用后写入的数据才保证被加密。要禁用加密，请在配置文件中删除 data-encryption-method，或将该参数值为 "plaintext"，然后重启 TiKV。

如果启用了加密（即 data-encryption-method 的值不是 "plaintext"），则必须指定主密钥。要使用 AWS KMS 方式指定为主密钥，请在 [security.encryption] 部分之后添加 [security.encryption.master-key] 部分：

```
[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
endpoint = "https://kms.us-west-2.amazonaws.com"
```

key-id 指定 KMS CMK 的密钥 ID。region 为 KMS CMK 的 AWS 区域名。endpoint 通常无需指定，除非你在使用非 AWS 提供的 AWS KMS 兼容服务或需要使用 KMS VPC endpoint。

你也可以使用 AWS 多区域键。为此，你需要在一个特定的区域设置一个主键，并在需要的区域中添加副本密钥。

若要使用文件方式指定主密钥，主密钥配置应如下所示：

```
[security.encryption.master-key]
type = "file"
path = "/path/to/key/file"
```

以上示例中，path 为密钥文件的路径。该文件须包含一个 256 位（32 字节）的十六进制字符串，并以换行符结尾（即 \n），且不包含其他任何内容。密钥文件示例如下：

```
3b5896b5be691006e0f71c3040a29495ddcad20b14aff61806940ebd780d3c62
```

3.2.4.2.3 轮换主密钥

要轮换主密钥，你必须在配置中同时指定新的主密钥和旧的主密钥，然后重启 TiKV。用 security.encryption.master-key 指定新的主密钥，用 security.encryption.previous-master-key 指定旧的主密钥。security.encryption.previous-master-key 配置的格式与 encryption.master-key 相同。重启时，TiKV 必须能同时访问旧主密钥和新主密钥，但一旦 TiKV 启动并运行，就只需访问新密钥。此后，可以将 encryption.previous-master-key 项保留在配置文件中。即使重启时，TiKV 也只会在无法使用新的主密钥解密现有数据时尝试使用旧密钥。

TiKV 当前不支持在线轮换主密钥，因此你需要重启 TiKV 进行主密钥轮换。建议对运行中的、提供在线查询的 TiKV 集群进行滚动重启。

轮换 KMS CMK 的配置示例如下：

```
[security.encryption.master-key]
type = "kms"
key-id = "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
region = "us-west-2"
```

```
[security.encryption.previous-master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

3.2.4.2.4 监控和调试

要监控静态加密（如果 TiKV 中部署了 Grafana 组件），可以查看 TiKV-Details -> Encryption 面板中的监控项：

- Encryption initialized：如果在 TiKV 启动期间初始化了加密，则为 1，否则为 0。进行主密钥轮换时可通过该监控项确认主密钥轮换是否已完成。
- Encryption data keys：现有数据密钥的数量。每次轮换数据密钥后，该数字都会增加 1。通过此监控指标可以监测数据密钥是否按预期轮换。
- Encrypted files：当前的加密数据文件数量。为先前未加密的群集启用加密时，将此数量与数据目录中的当前数据文件进行比较，可通过此监控指标估计已经被加密的数据量。
- Encryption meta file size：加密元数据文件的大小。
- Read/Write encryption meta duration：对用于加密的元数据进行操作带来的额外开销。

在调试方面，可使用 `tikv-ctl` 命令查看加密元数据（例如使用的加密方法和数据密钥列表）。该操作可能会暴露密钥，因此不推荐在生产环境中使用。详情参阅TiKV Control。

3.2.4.2.5 TiKV 版本间兼容性

为了减少 TiKV 管理加密元数据造成的 I/O 操作和互斥锁竞争引发的开销，TiKV v4.0.9 对此进行了优化。此优化可以通过 TiKV 配置文件中的 `security.encryption.enable-file-dictionary-log` 参数启用或关闭。此配置参数仅在 TiKV v4.0.9 或更高版本中生效。

该配置项默认开启，此时 TiKV v4.0.8 或更早期的版本无法识别加密元数据的数据格式。例如，假设你正在使用 TiKV v4.0.9 或更高版本，开启了静态加密和默认开启了 `enable-file-dictionary-log` 配置。如果将集群降级到 TiKV v4.0.8 或更早版本，TiKV 将无法启动，并且信息日志中会有类似如下的报错：

```
[2020/12/07 07:26:31.106 +08:00] [ERROR] [mod.rs:110] ["encryption: failed to load file
    ↪ dictionary."]
[2020/12/07 07:26:33.598 +08:00] [FATAL] [lib.rs:483] ["called `Result::unwrap()` on an `Err`
    ↪ value: Other(\"[components/encryption/src/encrypted_file/header.rs:18]: unknown version
    ↪ 2\")"]
```

为了避免上面所示错误，你可以首先将 `security.encryption.enable-file-dictionary-log` 设置为 `false`，然后启动 TiKV v4.0.9 或更高版本。TiKV 成功启动后，加密元数据的数据格式将降级为 TiKV 早期版本可以识别的格式。此时，你可再将 TiKV 集群降级到较早的版本。

3.2.4.3 BR S3 服务端加密

使用 BR 备份数据到 S3 时，若要启用 S3 服务端加密，需要传递 `--s3.sse` 参数并将参数值设置为 `aws:kms`。S3 将使用自己的 KMS 密钥进行加密。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.sse aws:kms
```

若要使用用户创建和拥有的自定义 AWS KMS CMK，需另外传递 `--s3.sse-kms-key-id` 参数。此时，BR 进程和集群中的所有 TiKV 节点都需访问该 KMS CMK（例如，通过 AWS IAM），并且该 KMS CMK 必须与存储备份的 S3 bucket 位于同一 AWS 区域。建议通过 AWS IAM 向 BR 进程和 TiKV 节点授予对 KMS CMK 的访问权限。参见 AWS 文档中的 IAM。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.region <region> --s3.
    ↪ sse aws:kms --s3.sse-kms-key-id 0987dcba-09fe-87dc-65ba-ab0987654321
```

恢复备份时，不需要也不可指定 `--s3.sse` 和 `--s3.sse-kms-key-id` 参数。S3 将自动相应进行解密。用于恢复备份数据的 BR 进程和集群中的 TiKV 节点也需要访问 KMS CMK，否则恢复将失败。示例如下：

```
./br restore full --pd <pd-address> --storage "s3://<bucket>/<prefix> --s3.region <region>"
```

3.2.5 为 TiDB 落盘文件开启加密

当配置项 `oom-use-tmp-storage` 为 `true` 时，如果单条 SQL 语句的内存使用超出系统变量`tidb_mem_quota_query` 的限制，某些算子可以将执行时的中间结果作为临时文件落盘保存，直到查询执行完成之后将它们删除。

用户可以开启落盘文件加密功能，防止攻击者通过读取临时文件来访问数据。

3.2.5.1 配置方法

要启用落盘文件加密功能，可以在 TiDB 配置文件中的 `[security]` 部分，配置`spilled-file-encryption-method` 选项：

```
[security]
spilled-file-encryption-method = "aes128-ctr"
```

`spilled-file-encryption-method` 的可选值为 `aes128-ctr` 和 `plaintext`。默认值为 `plaintext`，表示不启用加密。

3.2.6 日志脱敏

TiDB 在提供详细的日志信息时，可能会把数据库敏感的数据（例如用户数据）打印出来，造成数据安全方面的风险。因此 TiDB、TiKV、PD 等组件各提供了一个配置项开关，开关打开后，会隐藏日志中包含的用户数据值。

3.2.6.1 TiDB 组件日志脱敏

TiDB 侧的日志脱敏需要将`global.tidb_redact_log` 的值设为 1。该变量值默认为 0，即关闭脱敏。

可以通过 `set` 语法，设置全局系统变量 `tidb_redact_log`，示例如下：

```
set @@global.tidb_redact_log=1;
```

设置后，所有新 session 产生的日志都会脱敏：

```
create table t (a int, unique key (a));
Query OK, 0 rows affected (0.00 sec)

insert into t values (1),(1);
ERROR 1062 (23000): Duplicate entry '1' for key 'a'
```

打印出的错误日志如下：

```
[2020/10/20 11:45:49.539 +08:00] [INFO] [conn.go:800] ["command dispatched failed"] [conn=5] [
    ↪ connInfo="id:5, addr:127.0.0.1:57222 status:10, collation:utf8_general_ci,  user:root"] [
    ↪ command=Query] [status="inTxn:0, autocommit:1"] [sql="insert into t values ( ? ) , ( ? )
    ↪ "] [txn_mode=OPTIMISTIC] [err="[kv:1062]Duplicate entry '?' for key 'a'"]
```

从以上报错日志可以看到，开启 tidb_redact_log 后，报错信息里的敏感内容被隐藏掉了（目前是用问号替代）。TiDB 日志中会把敏感信息隐藏掉，以此规避数据安全风险。

3.2.6.2 TiKV 组件日志脱敏

TiKV 侧的日志脱敏需要将security.redact-info-log 的值设为 true。该配置项值默认为 false，即关闭脱敏。

3.2.6.3 PD 组件日志脱敏

PD 侧的日志脱敏需要将security.redact-info-log 的值设为 true。该配置项值默认为 false，即关闭脱敏。

3.2.6.4 TiFlash 组件日志脱敏

TiFlash 侧的日志脱敏需要将 tiflash-server 中security.redact_info_log 配置项的值以及 tiflash-learner 中security ↪ .redact-info-log 配置项的值均设为 true。两配置项默认值均为 false，即关闭脱敏。

4 使用插件

4.1 TiDB 审计插件

TiDB 审计插件 (TiDB Audit Plugin) 记录 TiDB 服务器上的活动，这些活动应遵守企业的审计规范。对于每个用户 session，该插件记录用户访问服务器的信息（即用户名和 IP 地址）、执行的 SQL 查询、访问的数据库和表、环境变量变更、执行的 DDL 操作等信息。审计信息记录在一个日志文件中以备将来使用。

本文介绍了如何部署和使用 TiDB 审计插件。

4.1.1 获取插件

请联系 PingCAP 销售获取插件安装包文件。

4.1.2 部署审计插件

你可以使用 TiDB Operator 或者 TiUP 来部署审计插件。

4.1.2.1 使用 TiDB Operator 部署审计插件

配置 TidbCluster CR。

```
tidb:
  additionalContainers:
  - command:
    - sh
    - -c
    - touch /var/log/tidb/tidb-audit.log; tail -n0 -F /var/log/tidb/tidb-audit.log;
    image: busybox:1.26.2
    imagePullPolicy: IfNotPresent
    name: auditlog
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 20m
        memory: 5Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/log/tidb
      name: slowlog
  baseImage: &lt;your_private_docker_hub_username>/tidb-ent:v5.4.0
  plugins:
  - audit-1
```

4.1.2.2 使用 TiUP 部署审计插件

本节介绍了几种使用 TiUP 部署审计插件的场景：

- 在已部署集群上追加插件
- 安装时部署
- 扩容
- 升级集群

4.1.2.2.1 在已部署集群上追加插件

1. 在所有目标服务器上建立文件夹。

```
tiup cluster exec cluster_name --command "mkdir {{.DeployDir}}/plugin" -R tidb
```

2. 传输插件到所有目标服务器。

```
tiup cluster push cluster_name ~/Download/audit-1.so {{.DeployDir}}/plugin/audit-1.so -R
    ↪ tidb
```

3. 修改 TiDB 配置。

```
tiup cluster edit-config cluster_name
```

在每个 TiDB 节点下的 config 字段中，配置 plugin.dir 为对应的绝对路径，plugin.load 为插件名。

```
tidb_servers:
-  host: 172.16.7.213
   ssh_port: 22
   port: 4000
   status_port: 10080
   deploy_dir: /tidb-deploy/tidb-4000
   log_dir: /tidb-deploy/tidb-4000/log
   arch: amd64
   os: linux
   config:
     plugin.dir: /tidb-deploy/tidb-4000/plugin
     plugin.load: audit-1
```

4. 滚动重启所有 TiDB 节点。

```
tiup cluster reload cluster_name -R tidb
```

建议：

- 在步骤 3 中，如果所有节点部署路径完全相同，可直接在 server_configs.tidb 中进行全局配置，避免手动修改每一个节点。如果路径不完全相同，可以修改步骤 1、2 中 plugin 的路径为某一相同路径。
- 如果不是所有 TiDB 节点都要部署插件，则通过 -N 参数来指定节点（tiup cluster display 显示的 ID）。

4.1.2.2.2 安装时部署

安装时部署与“在已部署集群上追加插件”的操作步骤相似：在部署前的拓扑文件中配置好 plugin.load 和 plugin.dir，在启动集群前执行 `tiup cluster exec` 和 `tiup cluster push`。

4.1.2.2.3 扩容

按照常规操作正常扩容节点，然后参考“在已部署集群上追加插件”的操作步骤，通过 -N 参数指定刚扩容的节点。

请注意，你需要在扩容进行期间将插件安装文件（如上面例子中的 audit-1.so ）上传到目标服务器，而且要确保在扩容结束之前传完，否则扩容会报错。

如果扩容报错，你可以执行以下操作重新安装插件：

1. 重新进行扩容。
2. 在扩容完成后上传插件安装文件。
3. 执行 tiup cluster start cluster_name 命令重启。

4.1.2.2.4 升级集群

1. 删除老插件。

```
tiup cluster exec cluster_name --command "rm {{.DeployDir}}/plugin/audit-1.so" -R tidb
```

2. 传输新插件到所有目标服务器。为了避免修改配置，使用相同名称。

```
tiup cluster push cluster_name ~/Download/audit-1_new.so {{.DeployDir}}/plugin/audit-1.so -R
    ↪ tidb
```

说明：域名不能包含 “-” （如 tidb-server1 ），否则该命令会执行失败。

3. 升级集群。

```
tiup cluster upgrade cluster_name version
```

4.1.2.3 插件的 Events

TiDB 审计插件有以下几种 Events。

4.1.2.3.1 CONNECTION: Connection events and authentication events.

CONNECTION logs 示例：

- A successful authentication

  [2022/04/18 21:50:29.669 +08:00] [INFO] [logger.go:70] [ID=16502898290] [TIMESTAMP=2022/04/18 21:50:29.669 +08:00] [EVENT_CLASS=CONNECTION] [EVENT_SUBCLASS=Disconnect] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=] [ROWS=0] [CLIENT_PORT=39290] [CONNECTION_ID=11] [CONNECTION_TYPE=Socket] [SERVER_ID=1] [SERVER_PORT=4000] [DURATION=4522.55501] [SERVER_OS_LOGIN_USER=] [OS_VERSION=“Centos 6”] [CLIENT_VERSION=] [SERVER_VERSION=v5.2.0] [AUDIT_VERSION=] [SSL_VERSION=v1.2.0] [PID=32638] [Reason=]

- A failed authentication

  [2022/04/18 21:50:40.198 +08:00] [INFO] [logger.go:70] [ID=16502898400] [TIMESTAMP=2022/04/18 21:50:40.198 +08:00] [EVENT_CLASS=CONNECTION] [EVENT_SUBCLASS=Reject] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=ccd] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=] [ROWS=0] [CLIENT_PORT=39296] [CONNECTION_ID=15] [CONNECTION_TYPE=Socket] [SERVER_ID=1] [SERVER_PORT=4000] [DURATION=0] [SERVER_OS_LOGIN_USER=] [OS_VERSION=”Centos 6“] [CLIENT_VERSION=] [SERVER_VERSION=v5.2.0] [AUDIT_VERSION=] [SSL_VERSION=v1.2.0] [PID=32638] [Reason=“[server:1045]Access denied for user ‘ccd’@‘127.0.0.1’ (using password: NO)”]

4.1.2.3.2 TABLE_ACCESS: SQL queries related to tables.

TABLE_ACCESS logs 示例：

- Queries on a table (`select * from t`)

  [2022/04/18 22:00:47.085 +08:00] [INFO] [logger.go:70] [ID=16502904470] [TIMESTAMP=2022/04/18 22:00:47.085 +08:00] [EVENT_CLASS=TABLE_ACCESS] [EVENT_SUBCLASS=Select] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[test]”] [TABLES=“[t]”] [SQL_TEXT=“select * from t”] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=Select]

- Drop a table (`drop table t`)

  [2022/04/18 22:02:40.775 +08:00] [INFO] [logger.go:70] [ID=16502905600] [TIMESTAMP=2022/04/18 22:02:40.775 +08:00] [EVENT_CLASS=TABLE_ACCESS] [EVENT_SUBCLASS=DropTable] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[test]”] [TABLES=“[t,t1]”] [SQL_TEXT=“drop table t , t1”] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=DropTable]

> 注意：
>
> - 默认情况下，TABLE_ACCESS events 不会被记录在日志中。请参考本文档中 “修改 TABLE_ACCESS 相关配置” 了解如何配置。
> - 失败的 statements 不会被记录到日志中。

4.1.2.3.3 GENERAL: SQL queries that don’t access tables.

GENERAL logs 示例：

- `show tables` statement:

  [2022/04/18 22:10:34.845 +08:00] [INFO] [logger.go:70] [ID=16502910340] [TIMESTAMP=2022/04/18 22:10:34.845 +08:00] [EVENT_CLASS=GENERAL] [EVENT_SUBCLASS=] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=“show tables”] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=Show]

- `begin` statement:

  [2022/04/18 22:06:53.198 +08:00] [INFO] [logger.go:70] [ID=16502908130] [TIMESTAMP=2022/04/18 22:06:53.198 +08:00] [EVENT_CLASS=GENERAL] [EVENT_SUBCLASS=] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=begin] [ROWS=0] [CONNECTION_ID=19] [CLIENT_PORT=39522] [PID=32638] [COMMAND=Query] [SQL_STATEMENTS=Begin]

4.1.3 查看状态

通过 MySQL 的 `show plugins` 命令可以查看当前插件加载和启用状态信息：

- Name：插件名称

- Status：插件状态，由加载状态 (uninitialized, ready) 和启用状态 (enable, disable) 组成
- Type：插件类型
- Library：插件二进制文件位置
- Version：插件版本

```
mysql> show plugins;
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| Name  | Status       | Type  | Library                                                                           | License | Version |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| audit | Ready-enable | Audit | /home/robi/code/go/src/github.com/pingcap/enterprise-plugin/audit/audit-1.so |         | 1       |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 137: Show Plugins

4.1.4 启用和禁用插件

审计插件加载后，默认会对 CONNECTION 和 GENERAL 事件进行审计。启用或禁用命令在任意一个 TiDB 节点执行成功后，会作用于集群中所有 TiDB 节点。

执行以下命令启用审计插件：

```
mysql> admin plugins enable audit;
```

如果不想对这两种事件做审计，执行如下命令禁用审计插件：

```
mysql> admin plugins disable audit;
```

```
mysql> admin plugins disable audit;
Query OK, 0 rows affected (0.03 sec)

mysql> show plugins;
+-------+---------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| Name  | Status        | Type  | Library                                                                           | License | Version |
+-------+---------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| audit | Ready-disable | Audit | /home/robi/code/go/src/github.com/pingcap/enterprise-plugin/audit/audit-1.so |         | 1       |
+-------+---------------+-------+-----------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)

mysql> admin plugins enable audit;
Query OK, 0 rows affected (0.00 sec)

mysql> show plugins;
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| Name  | Status       | Type  | Library                                                                           | License | Version |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
| audit | Ready-enable | Audit | /home/robi/code/go/src/github.com/pingcap/enterprise-plugin/audit/audit-1.so |         | 1       |
+-------+--------------+-------+-----------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 138: Disable Plugins

4.1.5 修改 TABLE_ACCESS 相关配置

对于 TABLE_ACCESS 事件，默认不会进行审计。如需使用，需要进行相关配置。

TABLE_ACCESS 支持通过用户名 + 数据库 + 表名 + 语句类型这 4 个维度的组合对审计进行细粒度控制。修改配置需要运行 flush，使其在所有 TiDB 节点生效。

4.1.5.1 修改配置表 tidb_audit_table_access

在首次使用审计插件时，会在 TiDB 的 MySQL 库中创建一个名为 tidb_audit_table_access 的插件配置表。你可以通过修改配置表来控制审计插件对 table_access 的审计。

```
mysql> show create table mysql.tidb_audit_table_access;
CREATE TABLE `tidb_audit_table_access` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user` varchar(32) NOT NULL COMMENT 'username need be audit(regex)',
  `db` varchar(32) NOT NULL COMMENT 'db name need be audit(regex)',
  `tbl` varchar(32) NOT NULL COMMENT 'table name need be audit(regex)',
  `access_type` varchar(128) NOT NULL COMMENT 'table access types split by comma',
  `create_time` timestamp NULL DEFAULT NULL COMMENT 'record create time',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT
      ↪ 'record last update time',
  PRIMARY KEY (`id`),
  UNIQUE KEY `user` (`user`,`db`,`tbl`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

相关参数解释如下：

- user：开启 table_access 事件审计的用户。
- db：开启 table_access 事件审计的数据库。
- tbl：开启 table_access 事件审计的表名。
- access_type：开启 table_access 事件审计语句类型，类型同上面提到的 subClass，支持多个逗号分割，空字符串则代表所有都审计。

注意：user、db、tbl 使用正则且区分大小写，.* 代表 table-access 对所有 user、db 或 tbl 都开启。

例如，如果希望对任意用户访问任意库和表的所有 SQL 进行审计，可以插入以下数据：

```
mysql> insert into mysql.tidb_audit_table_access (user, db, tbl, access_type) values ('.*', '.*',
    ↪  '.*', '');
```

如果希望对任意用户访问 ac 库 account 表的所有 SQL 进行审计，可以插入以下数据：

```
mysql> insert into mysql.tidb_audit_table_access (user, db, tbl, access_type) values (".*", 'ac',
    ↪  ^account$, '');
```

如果希望对以 op_ 开头的用户名访问全部数据库 account 表的所有 SQL 进行审计，可以插入以下数据：

```
mysql> insert into mysql.tidb_audit_table_access (user, db, tbl, access_type) values ('op_.*',
    ↪ '.*', '.*', '');
```

注意匹配的规则不是通配符风格的。例如，如果 user 设置为 abc*，会匹配到 “ab”，“abc”，“abcc”，“abccc” 等。

支持的 access_type 请参考下面列表中的 SubEvent，支持多个逗号分割。空字符串则代表所有都审计。

- Insert
- Select
- Update
- Replace
- Delete
- LoadData
- Alter
- DropTable
- CreateTable

4.1.5.2 刷新配置

对 mysql.tidb_audit_table_access 表进行修改后，需要执行 flush tidb plugins audit; 命令进行刷新，以通知集群中所有 TiDB 节点从表中 load 最新配置。如该命令执行失败，则需要进行重试执行 flush 操作。

```
mysql> flush tidb plugins audit;
Query OK, 0 rows affected (0.00 sec)
```

4.1.6 使用插件输出审计日志

审计插件加载后，会根据配置把不同类型的审计日志输出到和 TiDB 配置项 slow-query-file 相同的目录下，日志文件名称为 `tidb-audit.log`。

4.1.6.1 支持的审计事件类型

下表列出了目前支持的审计事件类型。

Event Class

SubEvent

开关

备注

CONNECTION

Connected

Disconnect

ChangeUser

PreAuth

Reject

默认开启。

只能通过禁用插件关闭。

连接事件和鉴权事件。

TABLE_ACCESS

Insert

Select

Update

Replace

Delete

LoadData

Alter

DropTable

CreateTable

默认关闭。

需要通过修改配置表指定 db - user - table - 事件来开启。

带 table 的 SQL。

会对 SQL sql 中的参数脱敏处理替换为 “?”。

GENERAL

- 默认开启。
  只能通过禁用插件关闭。
  没有 table 访问的其他 SQL。

4.1.7 日志输出

所有事件类型的日志输出中，头部分是通用的字段, 在跳过前 3 个由日志库输出的字段后，有以下字段：

- ID：事件 ID
- TIMESTAMP：事件时间
- EVENT_CLASS：事件类型
- EVENT_SUBCLASS：事件子类型
- STATUS_CODE：语句响应状态
- COST_TIME：语句耗时
- HOST：服务器 IP
- CLIENT_IP：客户端 IP
- USER：登录用户名
- DATABASE：事件相关 db
- TABLES：事件相关的表名
- SQL_TEXT：脱敏后的 SQL
- ROWS：影响行数，无则为 0

日志输出中，会出现冗余或没数据也固定输出的情况，主要目的是方便当前日志分析系统处理。

CONNECTION、 TABLE_ACCESS、 GENERAL 还提供了补充输出的内容。

4.1.7.1 CONNECTION 相关日志

CONNECTION 连接事件补充输出的内容如下:

- CLIENT_PORT：客户端端口号
- CONNECTION_ID：连接 ID
- CONNECTION_TYPE：是通过 socket 还是 unix-socket 连接
- SERVER_ID：TiDB Server ID
- SERVER_PORT：TiDB server 监听 MySQL 协议端口
- SERVER_OS_LOGIN_USER：TiDB 进程启动系统用户名
- OS_VERSION：TiDB Server 所在操作系统版本
- SSL_VERSION：TiDB 当前 SSL 版本
- PID：TiDB 进程 PID

示例：

[2019/07/24 22:13:30.099 +08:00] [INFO] [logger.go:71] [ID=“15640224010”] [TIMESTAMP=2019/07/24 22:13:30.099 +08:00] [EVENT_CLASS=CONNECTION] [EVENT_SUBCLASS=Disconnect] [STATUS_CODE=0] [COST_TIME=0] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[test]”] [TABLES=“[]”] [SQL_TEXT=] [ROWS=0] [CLIENT_PORT=45814] [CONNECTION_ID=1] [CONNECTION_TYPE=Socket] [SERVER_ID=1] [SERVER_PORT=4000] [DURATION=1233372.167907] [SERVER_OS_LOGIN_USER=robi] [OS_VERSION=“Linux 5.0.0-20-generic.x86_64”] [CLIENT_VERSION=] [SERVER_VERSION=v2.1.15-16-g136e334df-dirty] [AUDIT_VERSION=] [SSL_VERSION=v1.2.0] [PID=24108]

4.1.7.1.1 TABLE_ACCESS 相关日志

TABLE_ACCESS 事件补充输出的内容如下：

- CONNECTION_ID：连接 ID
- COMMAND：MySQL 协议的 command 类型
- SQL_STATEMENTS：SQL 语句类型
- PID：TiDB 进程所在 PID

示例：

[2019/07/24 22:35:01.640 +08:00] [INFO] [logger.go:71] [ID=“15640226010”] [TIMESTAMP=2019/07/24 22:35:01.640 +08:00] [EVENT_CLASS=TABLE_ACCESS] [EVENT_SUBCLASS=Insert] [STATUS_CODE=0] [COST_TIME=31581.623] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[mysql]”] [TABLES=“[t1]”] [SQL_TEXT=“insert into t1 (a) values (?)”] [ROWS=1] [CONNECTION_ID=2] [CLIENT_PORT=47430] [PID=24108] [COMMAND=Query] [SQL_STATEMENTS=Insert]

4.1.7.1.2 GENERAL 相关日志

GENERAL 事件补充输出的内容如下：

- CONNECTION_ID：连接 ID
- COMMAND：MySQL 协议的 command 类型
- SQL_STATEMENTS：SQL 语句类型
- PID：TiDB 进程所在 PID

示例：

[2019/07/24 22:31:22.612 +08:00] [INFO] [logger.go:71] [ID=“15640334010”] [TIMESTAMP=2019/07/24 22:31:22.612 +08:00] [EVENT_CLASS=GENERAL] [EVENT_SUBCLASS=] [STATUS_CODE=0] [COST_TIME=67.6] [HOST=127.0.0.1] [CLIENT_IP=127.0.0.1] [USER=root] [DATABASES=“[]”] [TABLES=“[]”] [SQL_TEXT=“set @@tidb_max_chunk_size = ?”] [ROWS=0] [CONNECTION_ID=2] [CLIENT_PORT=47430] [PID=24108] [COMMAND=Query] [SQL_STATEMENTS=Set]

4.1.8 清除审计日志

TiUP 对于审计插件产生的审计日志不会自动清除。如果需要清除审计日志，执行以下命令：

```
tiup cluster clean &lt;cluster-name> --audit-log
```

> 注意：
>
> - 执行以上命令默认删除所有审计日志。
> - 清除操作不可逆，请先备份数据后使用。

4.2 TiDB 白名单插件

TiDB 提供了白名单插件。使用该插件，你不需要在自己的业务层开发管理白名单功能，也不需要使用 MySQL 的用户功能逐个添加用户、管理权限，而是可以直接通过配置插件管控白名单，提高管理效率。

本文档介绍了如何使用白名单插件。

4.2.1 获取插件

请联系 PingCAP 销售获取插件安装包。

4.2.2 部署插件

你可以使用 TiDB Operator 或者 TiUP 部署白名单插件。

4.2.2.1 使用 TiDB Operator 部署插件

配置 TidbCluster CR。

```
tidb:
  additionalContainers:
  - command:
    - sh
    - -c
    - touch /var/log/tidb/tidb-whitelist.log; tail -n0 -F /var/log/tidb/tidb-whitelist.log;
```

```
    image: busybox:1.26.2
    imagePullPolicy: IfNotPresent
    name: whitelistlog
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 20m
        memory: 5Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/log/tidb
      name: slowlog
  baseImage: &lt;your_private_docker_hub_username>/tidb-ent:v5.4.0
  plugins:
  - whitelist-1
```

4.2.2.2 使用 TiUP 部署插件

4.2.2.2.1 在已部署集群上追加插件

1. 在所有目标服务器建立文件夹。

```
tiup cluster exec cluster_name --command "mkdir {{.DeployDir}}/plugin" -R tidb
```

2. 传输插件到所有目标服务器。

```
tiup cluster push cluster_name ~/Download/whitelist-1.so {{.DeployDir}}/plugin/whitelist-1.
    ↪ so -R tidb
```

3. 修改 TiDB 配置。

```
tiup cluster edit-config cluster_name
```

在每个 TiDB 节点下的 config 字段中，配置 plugin.dir 为对应绝对路径，plugin.load 为插件名。

“‘yaml tidb_servers:

- host: 172.16.7.213 ssh_port: 22 port: 4000 status_port: 10080 deploy_dir: /tidb-deploy/tidb-4000 log_dir: /tidb-deploy/tidb-4000/log arch: amd64 os: linux config: plugin.dir: /tidb-deploy/tidb-4000/plugin plugin.load: whitelist-1 “‘

4. 滚动重启所有 TiDB 节点。

```
tiup cluster reload cluster_name -R tidb
```

建议：

- 步骤 3 中，如果所有节点部署路径完全相同，可直接在 server_configs.tidb 中进行全局配置，避免手动修改每一个节点。如果路径不完全相同，可以修改步骤 1、2 中 plugin 的路径为某一相同路径。
- 如果不是所有 TiDB 节点都要部署插件，通过 -N 参数来指定节点（tiup cluster display 显示的 ID）。

4.2.2.2.2 安装时部署

安装时部署与“在已部署集群上追加插件”的操作步骤相似：在部署前的拓扑文件中配置好 plugin.load 和 plugin.dir，在启动集群前执行 `tiup cluster exec` 和 `tiup cluster push`。

4.2.2.2.3 扩容

按照常规操作正常扩容节点，然后参考“在已部署集群上追加插件”的操作步骤，通过 -N 参数指定刚扩容的节点。

请注意，你需要在扩容进行期间将插件安装文件（如上面例子中的 whitelist-1.so）上传到目标服务器，而且要确保在扩容结束之前传完，否则扩容会报错。

如果扩容报错，你可以执行以下操作重新安装插件：

1. 重新进行扩容。
2. 在扩容完成后上传插件安装文件。
3. 执行 `tiup cluster start cluster_name` 命令重启。

4.2.2.2.4 升级集群

1. 删除老插件。

```
tiup cluster exec cluster_name --command "rm {{.DeployDir}}/plugin/whitelist-1.so" -R tidb
```

2. 传输新插件到所有目标服务器。为了避免修改配置，使用相同名称。

```
tiup cluster push cluster_name ~/Download/whitelist-1_new.so {{.DeployDir}}/plugin/whitelist
    ↪ -1.so -R tidb
```

   说明：域名不能包含“-”（如 tidb-server1），否则该命令会执行失败。

3. 升级集群。

```
tiup cluster upgrade cluster_name version
```

4.2.3 查看状态

使用 MySQL 的 `show plugins` 命令，可以查看当前插件加载和启用状态信息：

- Name: 插件名称

- Status: 插件状态, 由加载状态 (uninitialized, ready) 和启用状态 (enable, disable) 组成
- Type: 插件类型
- Library: 插件二进制文件位置
- Version: 插件版本

```
mysql> show plugins:
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
| Name      | Status       | Type  | Library                                                                                                         | License | Version |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
| whitelist | Ready-enable | Audit | /media/genius/OS/project/src/github.com/pingcap/enterprise-plugin/whitelist/whitelist-1.so |         | 1       |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 139: Show Plugins

4.2.4 启用和禁用插件

开启白名单以后，TiDB 会根据用户 IP 进行过滤，只有属于白名单的 IP 才允许登录。启用或禁用命令在可以在任意一个 TiDB 节点执行。执行成功后，会作用于集群中所有 TiDB 节点。

如不希望过滤 IP，可以禁用白名单插件。

要禁用和重启用白名单插件，分别执行以下相应的 admin plugins 命令即可：

```
mysql> admin plugins disable whitelist;
mysql> admin plugins enable whitelist;
```

```
mysql> admin plugins enable whitelist:
Query OK, 0 rows affected (0.00 sec)

mysql> show plugins:
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
| Name      | Status       | Type  | Library                                                                                                         | License | Version |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
| whitelist | Ready-enable | Audit | /media/genius/OS/project/src/github.com/pingcap/enterprise-plugin/whitelist/whitelist-1.so |         | 1       |
+-----------+--------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
1 row in set (0.01 sec)

mysql> admin plugins disable whitelist:
Query OK, 0 rows affected (0.00 sec)

mysql> show plugins:
+-----------+---------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
| Name      | Status        | Type  | Library                                                                                                         | License | Version |
+-----------+---------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
| whitelist | Ready-disable | Audit | /media/genius/OS/project/src/github.com/pingcap/enterprise-plugin/whitelist/whitelist-1.so |         | 1       |
+-----------+---------------+-------+-----------------------------------------------------------------------------------------------------------------+---------+---------+
1 row in set (0.00 sec)
```

图 140: Disable Plugins

你可以使用 MySQL 的 `show plugins` 命令查看插件的启用状态信息。

4.2.5 使用和验证插件

白名单插件启用后，TiDB 会从 `mysql.whitelist` 系统表读取白名单信息。

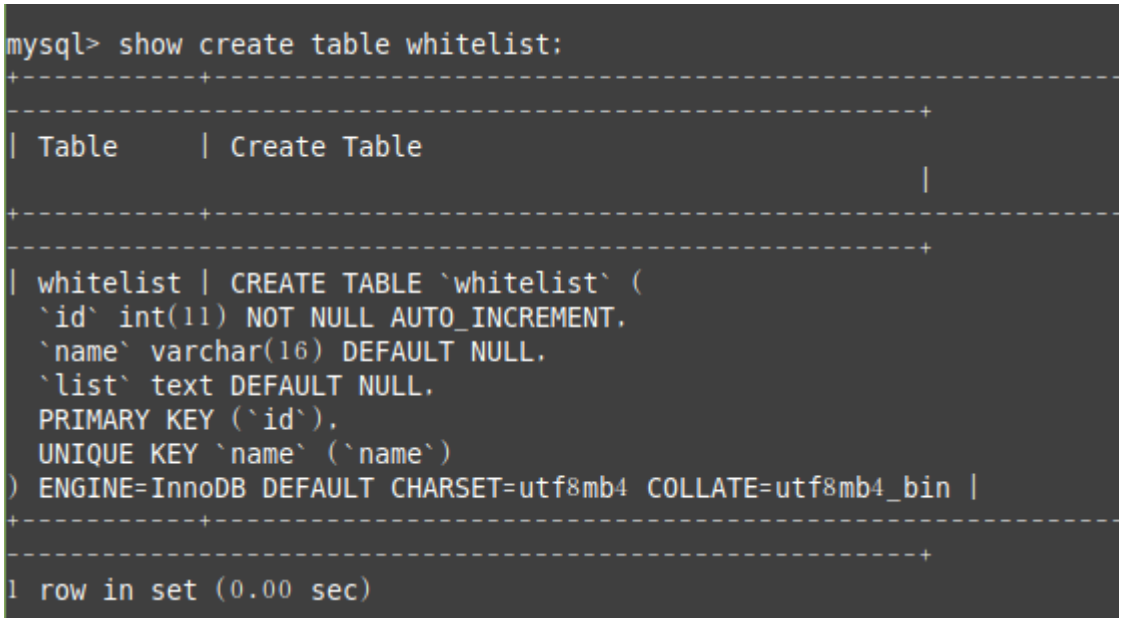


图 141: Plugin Allowlist

白名单表的 list 字段格式为一个字符串，以单个英文逗号 “,” 为分隔符，连接一串的 IP 网段。IP 网段以 CIDR 格式表示，比如 192.0.2.1/24。具体使用方法，请参考 Go 的 ParseCIDR 函数。

白名单插件一般配合前端 Web 使用。手动的验证操作如下：

1. 添加一条白名单记录，允许 192.0.2.0 网段和 127.0.0.0 网段的 IP 访问服务器：

```
mysql> insert into mysql.whitelist (name, list) values ('default',
    ↪ '192.0.2.1/24,127.0.0.1/24');
Query OK, 1 row affected (0.04 sec)
```

2. 配置生效后，从非白名单的 IP 机器去连接 TiDB 的时候会报错：

```
bin git:(master) mysql -h zenlife.tk -P 4000 -u root
ERROR 2013 (HY000): Lost connection to MySQL server at 'reading initial communication packet
    ↪ ', system error: 0
```

5 命令行工具

5.1 TiKV Control 使用说明

TiKV Control（以下简称 tikv-ctl）是 TiKV 的命令行工具，用于管理 TiKV 集群。它的安装目录如下：

- 如果是使用 TiUP 部署的集群，在 `~/.tiup/components/ctl/{VERSION}/` 目录下。

5.1.1 通过 TiUP 使用 TiKV Control

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

`tikv-ctl` 也集成在了 `tiup` 命令中。执行以下命令，即可调用 `tikv-ctl` 工具：

```
tiup ctl tikv
```

```
Starting component `ctl`: /home/tidb/.tiup/components/ctl/v4.0.8/ctl tikv
TiKV Control (tikv-ctl)
Release Version:   4.0.8
Edition:           Community
Git Commit Hash:   83091173e960e5a0f5f417e921a0801d2f6635ae
Git Commit Branch: heads/refs/tags/v4.0.8
UTC Build Time:    2020-10-30 08:40:33
Rust Version:      rustc 1.42.0-nightly (0de96d37f 2019-12-19)
Enable Features:   jemalloc mem-profiling portable sse protobuf-codec
Profile:           dist_release

A tool for interacting with TiKV deployments.

USAGE:
    TiKV Control (tikv-ctl) [FLAGS] [OPTIONS] [SUBCOMMAND]

FLAGS:
    -h, --help                    Prints help information
        --skip-paranoid-checks    Skip paranoid checks when open rocksdb
    -V, --version                 Prints version information

OPTIONS:
        --ca-path <ca_path>              Set the CA certificate path
        --cert-path <cert_path>          Set the certificate path
        --config <config>                Set the config for rocksdb
        --db <db>                        Set the rocksdb path
        --decode <decode>                Decode a key in escaped format
        --encode <encode>                Encode a key in escaped format
        --to-hex <escaped-to-hex>        Convert an escaped key to hex key
        --to-escaped <hex-to-escaped>    Convert a hex key to escaped key
        --host <host>                    Set the remote host
        --key-path <key_path>            Set the private key path
        --pd <pd>                        Set the address of pd
```

```
        --raftdb <raftdb>                Set the raft rocksdb path

SUBCOMMANDS:
    bad-regions           Get all regions with corrupt raft
    cluster               Print the cluster id
    compact               Compact a column family in a specified range
    compact-cluster       Compact the whole cluster in a specified range in one or more column
        ↪ families
    consistency-check     Force a consistency-check for a specified region
    decrypt-file          Decrypt an encrypted file
    diff                  Calculate difference of region keys from different dbs
    dump-snap-meta        Dump snapshot meta file
    encryption-meta       Dump encryption metadata
    fail                  Inject failures to TiKV and recovery
    help                  Prints this message or the help of the given subcommand(s)
    metrics               Print the metrics
    modify-tikv-config    Modify tikv config, eg. tikv-ctl --host ip:port modify-tikv-config -n
                          rocksdb.defaultcf.disable-auto-compactions -v true
    mvcc                  Print the mvcc value
    print                 Print the raw value
    raft                  Print a raft log entry
    raw-scan              Print all raw keys in the range
    recover-mvcc          Recover mvcc data on one node by deleting corrupted keys
    recreate-region       Recreate a region with given metadata, but alloc new id for it
    region-properties     Show region properties
    scan                  Print the range db range
    size                  Print region size
    split-region          Split the region
    store                 Print the store id
    tombstone             Set some regions on the node to tombstone by manual
    unsafe-recover        Unsafely recover the cluster when the majority replicas are failed
```

你可以在 `tiup ctl tikv` 后面再接上相应的参数与子命令。

5.1.2 通用参数

tikv-ctl 提供以下两种运行模式：

- 远程模式。通过 `--host` 选项接受 TiKV 的服务地址作为参数。在此模式下，如果 TiKV 启用了 SSL，则 tikv-ctl 也需要指定相关的证书文件，例如：

```
tikv-ctl --ca-path ca.pem --cert-path client.pem --key-path client-key.pem --host
    ↪ 127.0.0.1:20160 <subcommands>
```

  某些情况下，tikv-ctl 与 PD 进行通信，而不与 TiKV 通信。此时你需要使用 `--pd` 选项而非 `--host` 选项，例如：

```
tikv-ctl --pd 127.0.0.1:2379 compact-cluster
```

```
store:"127.0.0.1:20160" compact db:KV cf:default range:([], []) success!
```

- 本地模式。通过 --db 选项来指定本地 TiKV 数据的目录路径。在此模式下，需要停止正在运行的 TiKV 实例。

以下如无特殊说明，所有命令都同时支持这两种模式。

除此之外，tikv-ctl 还有两个简单的命令 --to-hex 和 --to-escaped，用于对 key 的形式作简单的变换。一般使用 escaped 形式，示例如下：

```
tikv-ctl --to-escaped 0xaaff
```

```
\252\377
```

```
tikv-ctl --to-hex "\252\377"
```

```
AAFF
```

> 注意：
>
> 在命令行上指定 escaped 形式的 key 时，需要用双引号引起来，否则 bash 会将反斜杠吃掉，导致结果错误。

5.1.3 各项子命令及部分参数、选项

下面逐一对 tikv-ctl 支持的子命令进行举例说明。有的子命令支持很多可选参数，要查看全部细节，可运行 tikv-ctl --help <subcommand>。

5.1.3.1 查看 Raft 状态机的信息

raft 子命令可以查看 Raft 状态机在某一时刻的状态。状态信息包括 RegionLocalState、 RaftLocalState 和 RegionApplyState 三个结构体，及某一条 log 对应的 Entries。

你可以使用 region 和 log 两个子命令分别查询以上信息。两条子命令都同时支持远程模式和本地模式。其用法及输出内容如下所示：

```
tikv-ctl --host 127.0.0.1:20160 raft region -r 2
```

```
region id: 2
region state key: \001\003\000\000\000\000\000\000\000\002\001
region state: Some(region {id: 2 region_epoch {conf_ver: 3 version: 1} peers {id: 3 store_id: 1}
    ↪ peers {id: 5 store_id: 4} peers {id: 7 store_id: 6}})
```

```
raft state key: \001\002\000\000\000\000\000\000\000\002\002
raft state: Some(hard_state {term: 307 vote: 5 commit: 314617} last_index: 314617)
apply state key: \001\002\000\000\000\000\000\000\000\002\003
apply state: Some(applied_index: 314617 truncated_state {index: 313474 term: 151})
```

5.1.3.2 查看 Region 的大小

使用 `size` 命令可以查看 Region 的大小：

```
tikv-ctl --data-dir /path/to/tikv size -r 2
```

```
region id: 2
cf default region size: 799.703 MB
cf write region size: 41.250 MB
cf lock region size: 27616
```

5.1.3.3 扫描查看给定范围的 MVCC

`scan` 命令的 `--from` 和 `--to` 参数接受两个 escaped 形式的 raw key，并用 `--show-cf` 参数指定只需要查看哪些列族。

```
tikv-ctl --data-dir /path/to/tikv scan --from 'zm' --limit 2 --show-cf lock,default,write
```

```
key: zmBootstr\377a\377pKey\000\000\377\000\000\373\000\000\000\000\000\377\000\000s
    ↪ \000\000\000\000\000\372
        write cf value: start_ts: 399650102814441473 commit_ts: 399650102814441475 short_value:
            ↪ "20"
key: zmDB:29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371
        write cf value: start_ts: 399650105239273474 commit_ts: 399650105239273475 short_value:
            ↪ "\000\000\000\000\000\000\000\002"
        write cf value: start_ts: 399650105199951882 commit_ts: 399650105213059076 short_value:
            ↪ "\000\000\000\000\000\000\000\001"
```

5.1.3.4 查看给定 key 的 MVCC

与 `scan` 命令类似，`mvcc` 命令可以查看给定 key 的 MVCC：

```
tikv-ctl --data-dir /path/to/tikv mvcc -k "zmDB
    ↪ :29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371" --
    ↪ show-cf=lock,write,default
```

```
key: zmDB:29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371
        write cf value: start_ts: 399650105239273474 commit_ts: 399650105239273475 short_value:
            ↪ "\000\000\000\000\000\000\000\002"
        write cf value: start_ts: 399650105199951882 commit_ts: 399650105213059076 short_value:
            ↪ "\000\000\000\000\000\000\000\001"
```

> **注意：**
>
> 该命令中，key 同样需要是 escaped 形式的 raw key。

5.1.3.5 扫描 raw key

使用 `raw-scan` 命令，TiKV 可直接在 RocksDB 中扫描 raw key。

> **注意：**
>
> 如果要扫描数据 key，需要在 key 前添加 `'z'` 前缀。

- 要指定扫描范围，可在 `raw-scan` 命令中使用 `--from` 和 `--to` 参数（默认不限范围）
- 要限制能够打印出的 key 的数量（默认为 30），可在命令中使用 `--limit` 参数
- 要指定扫描的 CF，可在命令中使用 `--cf` 参数（可选值为 `default`，`write`，`lock`）

```
$ ./tikv-ctl --data-dir /var/lib/tikv raw-scan --from 'zt' --limit 2 --cf default
```

```
key: "zt\200\000\000\000\000\000\000\377\005_r
    ↪ \200\000\000\000\000\377\000\000\001\000\000\000\000\000\000\372\372b2,^\033\377\364", value:
    ↪  "\010\002\002\002%\010\004\002\010root\010\006\002\000\010\010\t\002\010\n\t
    ↪ \002\010\014\t\002\010\016\t\002\010\020\t\002\010\022\t\002\010\024\t\002\010\026\t
    ↪ \002\010\030\t\002\010\032\t\002\010\034\t\002\010\036\t\002\010 \t\002\010\"\t\002\010s\
    ↪ t\002\010&\t\002\010(\t\002\010*\t\002\010,\t\002\010.\t\002\0100\t\002\0102\t\002\0104\t
    ↪ \002"
key: "zt\200\000\000\000\000\000\000\377\025_r
    ↪ \200\000\000\000\000\377\000\000\023\000\000\000\000\000\000\372\372b2,^\033\377\364", value:
    ↪  "\010\002\002&slow_query_log_file\010\004\002P/usr/local/mysql/data/localhost-slow.log"

Total scanned keys: 2
```

5.1.3.6 打印某个 key 的值

打印某个 key 的值需要用到 `print` 命令。示例从略。

5.1.3.7 打印 Region 的 properties 信息

为了记录 Region 的状态信息，TiKV 将一些数据写入 Region 的 SST 文件中。你可以用子命令 `region-properties` 运行 tikv-ctl 来查看这些 properties 信息。例如：

```
tikv-ctl --host localhost:20160 region-properties -r 2
```

```
num_files: 0
num_entries: 0
num_deletes: 0
mvcc.min_ts: 18446744073709551615
mvcc.max_ts: 0
mvcc.num_rows: 0
mvcc.num_puts: 0
mvcc.num_versions: 0
mvcc.max_row_versions: 0
middle_key_by_approximate_size:
```

这些 properties 信息可以用于检查某个 Region 是否健康或者修复不健康的 Region。例如，使用 `middle_key_approximate_size` 可以手动分裂 Region。

5.1.3.8 手动 compact 单个 TiKV 的数据

`compact` 命令可以对单个 TiKV 进行手动 compact。如果指定 `--from` 和 `--to` 选项，那么它们的参数也是 escaped raw key 形式的。

- `--host` 参数可以指定要 compact 的 TiKV。
- `-d` 参数可以指定要 compact 的 RocksDB，有 `kv` 和 `raft` 参数值可以选。
- `--threads` 参数可以指定 compact 的并发数，默认值是 8。一般来说，并发数越大，compact 的速度越快，但是也会对服务造成影响，所以需要根据情况选择合适的并发数。
- `--bottommost` 参数可以指定 compact 是否包括最下层的文件。可选值为 `default`、`skip` 和 `force`，默认为 `default`。
  - `default` 表示只有开启了 Compaction Filter 时 compact 才会包括最下层文件。
  - `skip` 表示 compact 不包括最下层文件。
  - `force` 表示 compact 总是包括最下层文件。

```
tikv-ctl --data-dir /path/to/tikv compact -d kv
```

```
success!
```

5.1.3.9 手动 compact 整个 TiKV 集群的数据

`compact-cluster` 命令可以对整个 TiKV 集群进行手动 compact。该命令参数的含义和使用与 `compact` 命令一样。

5.1.3.10 设置一个 Region 副本为 tombstone 状态

`tombstone` 命令常用于没有开启 sync-log，因为机器掉电导致 Raft 状态机丢失部分写入的情况。它可以在一个 TiKV 实例上将一些 Region 的副本设置为 Tombstone 状态，从而在重启时跳过这些 Region，避免因为这些 Region 的副本的 Raft 状态机损坏而无法启动服务。这些 Region 应该在其他 TiKV 上有足够多的健康的副本以便能够继续通过 Raft 机制进行读写。

一般情况下，可以先在 PD 上将 Region 的副本通过 `remove-peer` 命令删除掉：

```
pd-ctl>> operator add remove-peer <region_id> <store_id>
```

然后再用 tikv-ctl 在那个 TiKV 实例上将 Region 的副本标记为 tombstone 以便跳过启动时对他的健康检查：

```
tikv-ctl --data-dir /path/to/tikv tombstone -p 127.0.0.1:2379 -r <region_id>
```

```
success!
```

但是有些情况下，当不能方便地从 PD 上移除这个副本时，可以指定 tikv-ctl 的 `--force` 选项来强制设置它为 tombstone：

```
tikv-ctl --data-dir /path/to/tikv tombstone -p 127.0.0.1:2379 -r <region_id>,<region_id> --force
```

```
success!
```

注意：

- 该命令只支持本地模式
- -p 选项的参数指定 PD 的 endpoints，无需 `http` 前缀。指定 PD 的 endpoints 是为了询问 PD 是否可以安全切换至 Tombstone 状态。

5.1.3.11 向 TiKV 发出 consistency-check 请求

`consistency-check` 命令用于在某个 Region 对应的 Raft 副本之间进行一致性检查。如果检查失败，TiKV 自身会 panic。如果 `--host` 指定的 TiKV 不是这个 Region 的 Leader，则会报告错误。

```
tikv-ctl --host 127.0.0.1:20160 consistency-check -r 2
```

```
success!
```

```
tikv-ctl --host 127.0.0.1:20161 consistency-check -r 2
```

```
DebugClient::check_region_consistency: RpcFailure(RpcStatus { status: Unknown, details: Some("
    ↪ StringError(\"Leader is on store 1\")") })
```

注意：

- 目前 consistency-check 与 TiDB GC 操作不兼容，存在误报错误的可能，因此不建议使用该命令。
- 该命令只支持远程模式。
- 即使该命令返回了成功信息，也需要检查是否有 TiKV panic 了。因为该命令只是向 Leader 请求进行一致性检查，但整个检查流程是否成功并不能在客户端知道。

5.1.3.12 Dump snapshot 元文件

这条子命令可以用于解析指定路径下的 Snapshot 元文件并打印结果。

5.1.3.13 打印 Raft 状态机出错的 Region

前面 `tombstone` 命令可以将 Raft 状态机出错的 Region 设置为 Tombstone 状态，避免 TiKV 启动时对它们进行检查。在运行 `tombstone` 命令之前，可使用 `bad-regions` 命令找到出错的 Region，以便将多个工具组合起来进行自动化的处理。

```
tikv-ctl --data-dir /path/to/tikv bad-regions
```

```
all regions are healthy
```

命令执行成功后会打印以上信息，否则会打印出有错误的 Region 列表。目前可以检出的错误包括 `last index`、`commit index` 和 `apply index` 之间的不匹配，以及 Raft log 的丢失。其他一些情况，比如 Snapshot 文件损坏等仍然需要后续的支持。

5.1.3.14 查看 Region 属性

- 本地查看部署在 `/path/to/tikv` 的 TiKV 上面 Region 2 的 properties 信息：

  ```
  tikv-ctl --data-dir /path/to/tikv/data region-properties -r 2
  ```

- 在线查看运行在 `127.0.0.1:20160` 的 TiKV 上面 Region 2 的 properties 信息：

  ```
  tikv-ctl --host 127.0.0.1:20160 region-properties -r 2
  ```

5.1.3.15 动态修改 TiKV 的配置

使用 `modify-tikv-config` 命令可以动态修改配置参数。目前可动态修改的 TiKV 配置与具体的修改行为与 SQL 动态修改配置功能相同，可参考在线修改 TiKV 配置。

- `-n` 用于指定完整的配置名。支持动态修改的配置名可以参考在线修改 TiKV 配置中支持的配置项列表。
- `-v` 用于指定配置值。

设置 `shared block cache` 的大小：

```
tikv-ctl --host ip:port modify-tikv-config -n storage.block-cache.capacity -v 10GB
```

```
success
```

当禁用 `shared block cache` 时，为 `write` CF 设置 `block cache size`：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.writecf.block-cache-size -v 256MB
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftdb.defaultcf.disable-auto-compactions -v true
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftstore.sync-log -v false
```

```
success
```

如果 compaction 的流量控制导致待 compact 数据量 (compaction pending bytes) 堆积，可以禁用 `rate-limiter-auto-tuned` 配置项或调高 compaction 相关的流量阈值。示例如下：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.rate-limiter-auto-tuned -v false
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.rate-bytes-per-sec -v "1GB"
```

```
success
```

5.1.3.16 强制 Region 从多副本失败状态恢复服务（弃用）

> 警告：
>
> 不推荐使用该功能，恢复需求可通过 `pd-ctl` 的 Online Unsafe Recovery 功能实现。它提供了一键式自动恢复的能力，无需停止服务等额外操作，具体使用方式请参考 Online Unsafe Recovery 使用文档。

`unsafe-recover remove-fail-stores` 命令可以将故障机器从指定 Region 的 peer 列表中移除。运行命令之前，需要目标 TiKV 先停掉服务以便释放文件锁。

`-s` 选项接受多个以逗号分隔的 `store_id`，并使用 `-r` 参数来指定包含的 Region。如果要对某一个 store 上的全部 Region 都执行这个操作，可简单指定 `--all-regions`。

> 警告：
>
> - 此功能使用不当可能导致集群难以恢复，存在风险。请悉知潜在的风险，尽量避免在生产环境中使用。
> - 如果使用 `--all-regions`，必须在剩余所有连入集群的 store 上执行此命令。需要保证这些健康的 store 都停掉服务后再进行恢复，否则期间 Region 副本之间的 peer 列表不一致会导致执行 `split-region` 或者 `remove-peer` 时报错进而引起其他元数据的不一致，最终引发 Region 不可用。

- 一旦执行了 remove-fail-stores，不可再重新启动被移除的节点并将其加入集群，否则会导致元数据的不一致，最终引发 Region 不可用。

```
tikv-ctl --data-dir /path/to/tikv unsafe-recover remove-fail-stores -s 3 -r 1001,1002
```

```
success!
```

```
tikv-ctl --data-dir /path/to/tikv unsafe-recover remove-fail-stores -s 4,5 --all-regions
```

之后启动 TiKV，这些 Region 便可以使用剩下的健康副本继续提供服务了。此命令常用于多个 TiKV store 损坏或被删除的情况。

注意：

- 一般来说，您需要为指定 Region 的 peers 所在的每个 store 运行此命令。
- 该命令只支持本地模式。在运行成功后，会打印 success!。

5.1.3.17 恢复损坏的 MVCC 数据

recover-mvcc 命令用于 MVCC 数据损坏导致 TiKV 无法正常运行的情况。为了从不同种类的不一致情况中恢复，该命令会交叉检查 3 个 CF（“default”，“write”，“lock”）。

- -r 选项可以通过 region_id 指定包含的 Region。
- -p 选项可以指定 PD 的 endpoints。

```
tikv-ctl --data-dir /path/to/tikv recover-mvcc -r 1001,1002 -p 127.0.0.1:2379
```

```
success!
```

注意：

- 该命令只支持本地模式。在运行成功后，会打印 success!。
- -p 选项指定 PD 的 endpoint，不使用 http 前缀，用于查询指定的 region_id 是否有效。
- 对于指定 Region 的 peers 所在的每个 store，均须执行该命令。

5.1.3.18 Ldb 命令

ldb 命令行工具提供多种数据访问以及数据库管理命令。下方列出了一些示例用法。详细信息请在运行 tikv-ctl ldb 命令时查看帮助消息或查阅 RocksDB 文档。

数据访问序列的示例如下：

用 HEX 格式 dump 现有 RocksDB 数据：

```
tikv-ctl ldb --hex --db=/tmp/db dump
```

Dump 现有 RocksDB 的声明：

```
tikv-ctl ldb --hex manifest_dump --path=/tmp/db/MANIFEST-000001
```

您可以通过 --column_family=<string> 指定查询的目标列族。

通过 --try_load_options 命令加载数据库选项文件以打开数据库。在数据库运行时，建议您保持该命令为开启的状态。如果您使用默认配置打开数据库，LSM-tree 存储组织可能会出现混乱，且无法自动恢复。

5.1.3.19 打印加密元数据

encryption-meta 命令用于打印加密元数据。该子命令可以打印两种加密元数据：数据文件的加密信息，以及所有的数据加密密钥。

使用 encryption-meta dump-file 子命令打印数据文件的加密信息。你需要创建一个 TiKV 配置文件用以指定 TiKV 的数据目录：

```
## conf.toml
[storage]
data-dir = "/path/to/tikv/data"
```

--path 选项可以指定数据文件的绝对或者相对路径。如果指定的文件是明文存储的，本命令有可能没有输出。如果不指定 --path 选项，本命令打印所有数据文件的加密信息。

```
$ tikv-ctl --config=./conf.toml encryption-meta dump-file --path=/path/to/tikv/data/db/CURRENT
/path/to/tikv/data/db/CURRENT: key_id: 9291156302549018620 iv: E3C2FDBF63FC03BFC28F265D7E78283F
    ↪ method: Aes128Ctr
```

使用 encryption-meta dump-key 打印数据加密密钥。使用本命令的时候，除了在 TiKV 配置文件中指定 TiKV 的数据目录以外，还需要指定当前的主加密密钥。请参阅静态加密文档关于配置 TiKV 主加密密钥的说明。使用本命令时 security.encryption.previous-master-key 配置项不生效，即使配置文件中使用了该配置，本命令也不会触发更换主加密密钥。

```
## conf.toml
[storage]
data-dir = "/path/to/tikv/data"

[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

注意如果使用了 AWS KMS 作为主加密密钥，使用本命令时 `tikv-ctl` 需要该 KMS 密钥的访问权限。KMS 访问权限可以通过环境变量、AWS 默认配置文件或 IAM 的方式传递给 `tikv-ctl`。详情请参阅相关 AWS 文档。

`--ids` 选项可以指定以逗号分隔的数据加密密钥 id 列表。如果不指定 `--ids` 选项，本命令打印所有的数据加密密钥，以及最新的数据加密密钥的 id。

本命令会输出一个警告，提示本命令会泄漏敏感数据。根据提示输入 “I consent” 即可。

```
$ ./tikv-ctl --config=./conf.toml encryption-meta dump-key
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
current key id: 9291156302549018620
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    ↪ 1592938357
```

```
$ ./tikv-ctl --config=./conf.toml encryption-meta dump-key --ids=9291156302549018620
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    ↪ 1592938357
```

> **注意：**
>
> 本命令会以明文方式打印数据加密密钥。在生产环境中，请勿将本命令的输出重定向到磁盘文件中。即使使用以后删除该文件也不能保证文件内容从磁盘中干净清除。

5.1.3.20 打印损坏的 SST 文件信息

TiKV 中损坏的 SST 文件会导致 TiKV 进程崩溃。在 TiDB v6.1.0 之前，损坏的 SST 文件会导致 TiKV 进程立即崩溃。从 TiDB v6.1.0 起，TiKV 进程会在 SST 文件损坏 1 小时后崩溃。

为了方便清理掉这些 SST 文件，你可以先使用 `bad-ssts` 命令打印出损坏的 SST 文件信息。

> **注意：**
>
> 执行此命令前，请保证关闭当前运行的 TiKV 实例。

```
$ tikv-ctl bad-ssts --data-dir </path/to/tikv> --pd <endpoint>
```

```
--------------------------------------------------------
corruption info:
data/tikv-21107/db/000014.sst: Corruption: Bad table magic number: expected 9863518390377041911,
    ↪ found 759105309091689679 in data/tikv-21107/db/000014.sst
```

```
sst meta:
14:552997[1 .. 5520]['0101' seq:1, type:1 .. '7
    ↪ A7480000000000000FF0F5F728000000000FF000216000000000FAFA13AB33020BFFFA' seq:2032, type
    ↪ :1] at level 0 for Column family "default"  (ID 0)
it isn't easy to handle local data, start key:0101
overlap region:
RegionInfo { region: id: 4 end_key: 7480000000000000FF0500000000000000F8 region_epoch { conf_ver:
    ↪  1 version: 2 } peers { id: 5 store_id: 1 }, leader: Some(id: 5 store_id: 1) }
suggested operations:
tikv-ctl ldb --db=data/tikv-21107/db unsafe_remove_sst_file "data/tikv-21107/db/000014.sst"
tikv-ctl --db=data/tikv-21107/db tombstone -r 4 --pd <endpoint>
-------------------------------------------------------
corruption analysis has completed
```

通过上面的输出，你可以看到损坏的 SST 文件和损坏原因等信息先被打印出，然后是相关的元信息。

- 在 `sst meta` 输出部分，14 表示 SST 文件号，552997 表示文件大小，紧随其后的是最小和最大的序列号 (seq) 等其它元信息。
- `overlap region` 部分为损坏 SST 文件所在 Region 的信息。该信息是从 PD 组件获取的。
- `suggested operations` 部分为你清理损坏的 SST 文件提供建议操作。你可以参考这些建议的命令，清理文件，并重新启动该 TiKV 实例。

5.2 PD Control 使用说明

PD Control 是 PD 的命令行工具，用于获取集群状态信息和调整集群。

5.2.1 安装方式

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

5.2.1.1 使用 TiUP

可直接通过 `tiup ctl:<cluster-version> pd -u http://<pd_ip>:<pd_port> [-i]` 使用。

5.2.1.2 下载安装包

PD Control 的安装包 `etcdctl` 位于 TiDB 离线工具包中。下载方式，请参考 TiDB 工具下载。

5.2.1.3 源码编译

1. Go Version 1.13 以上
2. 在 PD 项目根目录使用 `make` 或者 `make pd-ctl` 命令进行编译，生成 bin/pd-ctl

5.2.2 简单例子

单命令模式：

```
tiup ctl pd store -u http://127.0.0.1:2379
```

交互模式：

```
tiup ctl pd -i -u http://127.0.0.1:2379
```

使用环境变量：

```
export PD_ADDR=http://127.0.0.1:2379 &&
tiup ctl pd
```

使用 TLS 加密：

```
tiup ctl pd -u https://127.0.0.1:2379 --cacert="path/to/ca" --cert="path/to/cert" --key="path/to/
    ↪ key"
```

5.2.3 命令行参数 (flags)

5.2.3.1 `--cacert`

- 指定 PEM 格式的受信任 CA 证书的文件路径
- 默认值： “”

5.2.3.2 `--cert`

- 指定 PEM 格式的 SSL 证书的文件路径
- 默认值： “”

5.2.3.3 `--detach / -d`

- 使用单命令行模式（不进入 readline）
- 默认值：true

5.2.3.4 `--help / -h`

- 输出帮助信息
- 默认值：false

5.2.3.5 `--interact/-i`

- 使用交互模式（进入 readline）
- 默认值：false

5.2.3.6 --key

- 指定 PEM 格式的 SSL 证书密钥文件路径，即 --cert 所指定的证书的私钥
- 默认值：“”

5.2.3.7 --pd/-u

- 指定 PD 的地址
- 默认地址：http://127.0.0.1:2379
- 环境变量：PD_ADDR

5.2.3.8 --version/-V

- 打印版本信息并退出
- 默认值：false

5.2.4 命令 (command)

5.2.4.1 cluster

用于显示集群基本信息。

示例：

```
>> cluster
```

```
{
  "id": 6493707687106161130,
  "max_peer_count": 3
}
```

5.2.4.2 config [show | set <option> <value> | placement-rules]

用于显示或调整配置信息。示例如下。

显示 scheduling 的相关 config 信息：

```
>> config show
```

```
{
  "replication": {
    "enable-placement-rules": "true",
    "isolation-level": "",
    "location-labels": "",
    "max-replicas": 3,
    "strictly-match-label": "false"
```

```
  },
  "schedule": {
    "enable-cross-table-merge": "true",
    "high-space-ratio": 0.7,
    "hot-region-cache-hits-threshold": 3,
    "hot-region-schedule-limit": 4,
    "leader-schedule-limit": 4,
    "leader-schedule-policy": "count",
    "low-space-ratio": 0.8,
    "max-merge-region-keys": 200000,
    "max-merge-region-size": 20,
    "max-pending-peer-count": 64,
    "max-snapshot-count": 64,
    "max-store-down-time": "30m0s",
    "merge-schedule-limit": 8,
    "patrol-region-interval": "10ms",
    "region-schedule-limit": 2048,
    "region-score-formula-version": "v2",
    "replica-schedule-limit": 64,
    "scheduler-max-waiting-operator": 5,
    "split-merge-interval": "1h0m0s",
    "tolerant-size-ratio": 0
  }
}
```

显示所有的 config 信息：

```
>> config show all
```

显示 replication 的相关 config 信息：

```
>> config show replication
```

```
{
  "max-replicas": 3,
  "isolation-level": "",
  "location-labels": "",
  "strictly-match-label": "false",
  "enable-placement-rules": "true"
}
```

显示目前集群版本，是目前集群 TiKV 节点的最低版本，并不对应 binary 的版本：

```
>> config show cluster-version
```

```
"5.2.2"
```

- `max-snapshot-count` 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。当需要加快补副本或 balance 速度时可以调大这个值。

  设置最大 snapshot 为 64：

```
>> config set max-snapshot-count 64
```

- `max-pending-peer-count` 控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。需要加快补副本或 balance 速度可以适当调大这个值，设置为 0 则表示不限制。

  设置最大 pending peer 数量为 64：

```
>> config set max-pending-peer-count 64
```

- `max-merge-region-size` 控制 Region Merge 的 size 上限（单位是 M）。当 Region Size 大于指定值时 PD 不会将其与相邻的 Region 合并。设置为 0 表示不开启 Region Merge 功能。

  设置 Region Merge 的 size 上限为 16 M：

```
>> config set max-merge-region-size 16
```

- `max-merge-region-keys` 控制 Region Merge 的 keyCount 上限。当 Region KeyCount 大于指定值时 PD 不会将其与相邻的 Region 合并。

  设置 Region Merge 的 keyCount 上限为 50000：

```
>> config set max-merge-region-keys 50000
```

- `split-merge-interval` 控制对同一个 Region 做 `split` 和 `merge` 操作的间隔，即对于新 `split` 的 Region 一段时间内不会被 `merge`。

  设置 `split` 和 `merge` 的间隔为 1 天：

```
>> config set split-merge-interval 24h
```

- `enable-one-way-merge` 用于控制是否只允许和相邻的后一个 Region 进行合并。当设置为 `false` 时，PD 允许与相邻的前后 Region 进行合并。

  设置只允许和相邻的后一个 Region 合并：

```
>> config set enable-one-way-merge true
```

- `enable-cross-table-merge` 用于开启跨表 Region 的合并。当设置为 `false` 时，PD 不会合并不同表的 Region。该选项只在键类型为“table”时生效。

  设置允许跨表合并：

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

- `key-type` 用于指定集群的键编码类型。支持的类型有 `["table", "raw", "txn"]`，默认值为“table”。

  - 如果集群中不存在 TiDB 实例，`key-type` 的值为“raw”或“txn”。此时，无论 `enable-cross-table-merge` 设置为何，PD 均可以跨表合并 Region。

  - 如果集群中存在 TiDB 实例，key-type 的值应当为 “table”。此时，enable-cross-table-merge 的设置决定了 PD 是否能跨表合并 Region。如果 key-type 的值为 “raw”，placement rules 不生效。

  启用跨表合并：

```
>> config set key-type raw
```

- region-score-formula-version 用于设置 Region 算分公式的版本，支持的值有 ["v1", "v2"]。v2 版本公式有助于减少上下线等场景下冗余的 balance Region 调度。

  开启 v2 版本 Region 算分公式：

```
>> config set region-score-formula-version v2
```

- patrol-region-interval 控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整。

  设置 replicaChecker 的运行频率为 10 毫秒：

```
>> config set patrol-region-interval 10ms
```

- max-store-down-time 为 PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。

  设置 store 心跳丢失 30 分钟开始补副本：

```
>> config set max-store-down-time 30m
```

- max-store-preparing-time 控制 store 上线阶段的最长等待时间。在 store 的上线阶段，PD 可以查询该 store 的上线进度。当超过该配置项指定的时间后，PD 会认为该 store 已完成上线，无法再次查询这个 store 的上线进度，但是不影响 Region 向这个新上线 store 的迁移。通常用户无需修改该配置项。

  设置 store 上线阶段最多等待 4 小时：

```
>> config set max-store-preparing-time 4h
```

- 通过调整 leader-schedule-limit 可以控制同时进行 leader 调度的任务个数。这个值主要影响 leader balance 的速度，值越大调度得越快，设置为 0 则关闭调度。Leader 调度的开销较小，需要的时候可以适当调大。

  最多同时进行 4 个 leader 调度：

```
>> config set leader-schedule-limit 4
```

- 通过调整 region-schedule-limit 可以控制同时进行 Region 调度的任务个数。这个值可以避免创建过多的 Region balance operator。默认值为 2048，对所有大小的集群都足够。设置为 0 则关闭调度。Region 调度的速度通常受到 store-limit 的限制，但除非你熟悉该设置，否则不推荐自定义该参数。

  最多同时进行 2 个 Region 调度：

```
>> config set region-schedule-limit 2
```

- 通过调整 `replica-schedule-limit` 可以控制同时进行 replica 调度的任务个数。这个值主要影响节点挂掉或者下线的时候进行调度的速度，值越大调度得越快，设置为 0 则关闭调度。Replica 调度的开销较大，所以这个值不宜调得太大。

  最多同时进行 4 个 replica 调度：

```
>> config set replica-schedule-limit 4
```

- `merge-schedule-limit` 控制同时进行的 Region Merge 调度的任务，设置为 0 则关闭 Region Merge。Merge 调度的开销较大，所以这个值不宜调得过大。

  最多同时进行 16 个 merge 调度：

```
>> config set merge-schedule-limit 16
```

- `hot-region-schedule-limit` 控制同时进行的 Hot Region 调度的任务，设置为 0 则关闭调度。这个值不宜调得过大，否则可能对系统性能造成影响。

  最多同时进行 4 个 Hot Region 调度：

```
>> config set hot-region-schedule-limit 4
```

- `hot-region-cache-hits-threshold` 用于设置识别热点 Region 所需的分钟数，只有 Region 处于热点状态持续时间超过该分钟数后，才能参与热点调度。

- `tolerant-size-ratio` 控制 balance 缓冲区大小。当两个 store 的 leader 或 Region 的得分差距小于指定倍数的 Region size 时，PD 会认为此时 balance 达到均衡状态。

  设置缓冲区为约 20 倍平均 RegionSize：

```
>> config set tolerant-size-ratio 20
```

- `low-space-ratio` 用于设置 store 空间不足的阈值。当节点的空间占用比例超过指定值时，PD 会尽可能避免往对应节点迁移数据，同时主要针对剩余空间大小进行调度，避免对应节点磁盘空间被耗尽。

  设置空间不足阈值为 0.9：

```
config set low-space-ratio 0.9
```

- `high-space-ratio` 用于设置 store 空间充裕的阈值，此配置仅的在 `region-score-formula-version = v1` 时生效。当节点的空间占用比例小于指定值时，PD 调度时会忽略剩余空间这个指标，主要针对实际数据量进行均衡。

  设置空间充裕阈值为 0.5：

```
config set high-space-ratio 0.5
```

- `cluster-version` 集群的版本，用于控制某些 Feature 是否开启，处理兼容性问题。通常是集群正常运行的所有 TiKV 节点中的最低版本，需要回滚到更低的版本时才进行手动设置。

  设置 cluster version 为 1.0.8：

```
config set cluster-version 1.0.8
```

- `leader-schedule-policy` 用于选择 Leader 的调度策略，可以选择按照 `size` 或者 `count` 来进行调度。
- `scheduler-max-waiting-operator` 用于控制每个调度器同时存在的 operator 的个数。
- `enable-remove-down-replica` 用于开启自动删除 DownReplica 的特性。当设置为 false 时，PD 不会自动清理宕机状态的副本。
- `enable-replace-offline-replica` 用于开启迁移 OfflineReplica 的特性。当设置为 false 时，PD 不会迁移下线状态的副本。
- `enable-make-up-replica` 用于开启补充副本的特性。当设置为 false 时，PD 不会为副本数不足的 Region 补充副本。
- `enable-remove-extra-replica` 用于开启删除多余副本的特性。当设置为 false 时，PD 不会为副本数过多的 Region 删除多余副本。
- `enable-location-replacement` 用于开启隔离级别检查。当设置为 false 时，PD 不会通过调度来提升 Region 副本的隔离级别。
- `enable-debug-metrics` 用于开启 debug 的 metrics。当设置为 true 时，PD 会开启一些 metrics，比如 `balance-tolerant-size` 等。
- `enable-placement-rules` 用于开启 placement rules，在 v5.0 及以上的版本默认开启。
- `store-limit-mode` 用于控制 store 限速机制的模式。主要有两种模式：`auto` 和 `manual`。`auto` 模式下会根据 load 自动进行平衡调整（实验性功能）。
- PD 会对流量信息的末尾数字进行四舍五入处理，减少 Region 流量信息变化引起的统计信息更新。该配置项用于指定对 Region 流量信息的末尾进行四舍五入的位数。例如流量 100512 会归约到 101000。默认值为 3。该配置替换了 `trace-region-flow`。

  示例：将 `flow-round-by-digit` 的值设为 4：

```
config set flow-round-by-digit 4
```

5.2.4.3 config placement-rules [disable | enable | load | save | show | rule-group]

关于 `config placement-rules` 的具体用法，参考Placement Rules 使用文档。

5.2.4.4 health

用于显示集群健康信息。示例如下。

显示健康信息：

```
>> health
```

```
[
  {
    "name": "pd",
    "member_id": 13195394291058371180,
    "client_urls": [
```

```
      "http://127.0.0.1:2379"
      ......
    ],
    "health": true
  }
  ......
]
```

5.2.4.5 hot [read | write | store| history <start_time> <end_time> [<key> <value>]]

用于显示集群热点信息。示例如下。

显示读热点信息：

```
>> hot read
```

显示写热点信息：

```
>> hot write
```

显示所有 store 的读写信息：

```
>> hot store
```

显示历史读写热点信息:

```
>> hot history startTime endTime [ <name> <value> ]
```

例如查询时间 1629294000000 到 1631980800000 （毫秒）之间的历史热点 Region 信息:

```
>> hot history 1629294000000 1631980800000
```

```
{
  "history_hot_region": [
    {
      "update_time": 1630864801948,
      "region_id": 103,
      "peer_id": 1369002,
      "store_id": 3,
      "is_leader": true,
      "is_learner": false,
      "hot_region_type": "read",
      "hot_degree": 152,
      "flow_bytes": 0,
      "key_rate": 0,
      "query_rate": 305,
      "start_key": "7480000000000000FF5300000000000000F8",
      "end_key": "7480000000000000FF5600000000000000F8"
```

```
    },
    ...
  ]
}
```

对于参数的值为数组的请用 `x, y, ...` 的形式进行参数值的设置，所有支持的参数如下所示:

```
>> hot history 1629294000000 1631980800000 hot_region_type read region_id 1,2,3 store_id 1,2,3
    ↪ peer_id 1,2,3 is_leader true is_learner true
```


```
{
  "history_hot_region": [
    {
      "update_time": 1630864801948,
      "region_id": 103,
      "peer_id": 1369002,
      "store_id": 3,
      "is_leader": true,
      "is_learner": false,
      "hot_region_type": "read",
      "hot_degree": 152,
      "flow_bytes": 0,
      "key_rate": 0,
      "query_rate": 305,
      "start_key": "7480000000000000FF5300000000000000F8",
      "end_key": "7480000000000000FF5600000000000000F8"
    },
    ...
  ]
}
```


5.2.4.6 label [store <name> <value>]

用于显示集群标签信息。示例如下。

显示所有 label：

```
>> label
```

显示所有包含 label 为 “zone”：“cn” 的 store：

```
>> label store zone cn
```

5.2.4.7 member [delete | leader_priority | leader [show | resign | transfer <member_name>]]

用于显示 PD 成员信息，删除指定成员，设置成员的 leader 优先级。示例如下。

显示所有成员的信息：

```
>> member
```

```
{
  "header": {......},
  "members": [......],
  "leader": {......},
  "etcd_leader": {......},
}
```

下线 “pd2”：

```
>> member delete name pd2
```

```
Success!
```

使用 id 下线节点：

```
>> member delete id 1319539429105371180
```

```
Success!
```

显示 leader 的信息：

```
>> member leader show
```

```
{
  "name": "pd",
  "member_id": 13155432540099656863,
  "peer_urls": [......],
  "client_urls": [......]
}
```

将 leader 从当前成员移走：

```
>> member leader resign
```

```
......
```

将 leader 迁移至指定成员：

```
>> member leader transfer pd3
```

```
......
```

5.2.4.8 operator [check | show | add | remove]

用于显示和控制调度操作。

示例：

```
>> operator show                                    // 显示所有的 operators
>> operator show admin                              // 显示所有的 admin operators
>> operator show leader                             // 显示所有的 leader operators
>> operator show region                             // 显示所有的 Region operators
>> operator add add-peer 1 2                        // 在 store 2 上新增 Region 1 的一个副本
>> operator add add-learner 1 2                     // 在 store 2 上新增 Region 1 的一个
    ↪ learner 副本
>> operator add remove-peer 1 2                     // 移除 store 2 上的 Region 1 的一个副本
>> operator add transfer-leader 1 2                 // 把 Region 1 的 leader 调度到 store 2
>> operator add transfer-region 1 2 3 4             // 把 Region 1 调度到 store 2,3,4
>> operator add transfer-peer 1 2 3                 // 把 Region 1 在 store 2 上的副本调度到
    ↪ store 3
>> operator add merge-region 1 2                    // 将 Region 1 与 Region 2 合并
>> operator add split-region 1 --policy=approximate // 将 Region 1 对半拆分成两个 Region，
    ↪ 基于粗略估计值
>> operator add split-region 1 --policy=scan        // 将 Region 1 对半拆分成两个 Region，
    ↪ 基于精确扫描值
>> operator remove 1                                // 把 Region 1 的调度操作删掉
>> operator check 1                                 // 查看 Region 1 相关 operator 的状态
```

其中，Region 的分裂都是尽可能地从靠近中间的位置开始。对这个位置的选择支持两种策略，即 scan 和 approximate。它们之间的区别是，前者通过扫描这个 Region 的方式来确定中间的 key，而后者是通过查看 SST 文件中记录的统计信息，来得到近似的位置。一般来说，前者更加精确，而后者消耗更少的 I/O，可以更快地完成。

5.2.4.9 ping

用于显示ping PD 所需要花费的时间

示例：

```
>> ping
```

```
time: 43.12698ms
```

5.2.4.10 region <region_id> [--jq="<query string>"]

用于显示 Region 信息。使用 jq 格式化输出请参考jq 格式化 json 输出示例。示例如下。

显示所有 Region 信息：

```
>> region
```

```
{
  "count": 1,
  "regions": [......]
}
```

显示 Region id 为 2 的信息：

```
>> region 2
```

```
{
  "id": 2,
  "start_key": "7480000000000000FF1D00000000000000F8",
  "end_key": "7480000000000000FF1F00000000000000F8",
  "epoch": {
    "conf_ver": 1,
    "version": 15
  },
  "peers": [
    {
      "id": 40,
      "store_id": 3
    }
  ],
  "leader": {
    "id": 40,
    "store_id": 3
  },
  "written_bytes": 0,
  "read_bytes": 0,
  "written_keys": 0,
  "read_keys": 0,
  "approximate_size": 1,
  "approximate_keys": 0
}
```

5.2.4.11 region key [--format=raw|encode|hex] <key>

用于查询某个 key 位于哪一个 Region 上，支持 raw、encoding 和 hex 格式。使用 encoding 格式时，key 需要使用单引号。

Hex 格式（默认）示例：

```
>> region key 7480000000000000FF1300000000000000F8
{
  "region": {
    "id": 2,
```

```
    ......
  }
}
```

Raw 格式示例：

```
>> region key --format=raw abc
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

Encoding 格式示例：

```
>> region key --format=encode 't\200\000\000\000\000\000\000\377\035_r
    ↪ \200\000\000\000\000\377\017U\320\000\000\000\000\000\000\372'
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

5.2.4.12 region scan

用于获取所有 Region。

示例：

```
>> region scan
```

```
{
  "count": 20,
  "regions": [......],
}
```

5.2.4.13 region sibling <region_id>

用于查询某个 Region 相邻的 Region。

示例：

```
>> region sibling 2
```

```
{
  "count": 2,
  "regions": [......],
}
```

5.2.4.14 region keys [--format=raw|encode|hex] <start_key> <end_key> <limit>

用于查询某个 key 范围内的所有 Region。支持不带 endKey 的范围。`limit` 的默认值是 16，设为 -1 则表示无数量限制。示例如下：

显示从 a 开始的所有 Region 信息，数量上限为 16：

```
>> region keys --format=raw a
```

```
{
  "count": 16,
  "regions": [......],
}
```

显示 [a, z) 范围内的所有 Region 信息，数量上限为 16：

```
>> region keys --format=raw a z
```

```
{
  "count": 16,
  "regions": [......],
}
```

显示 [a, z) 范围内的所有 Region 信息，无数量上限：

```
>> region keys --format=raw a z -1
```

```
{
  "count": ...,
  "regions": [......],
}
```

显示从 a 开始的所有 Region 信息，数量上限为 20：

```
>> region keys --format=raw a "" 20
```

```
{
  "count": 20,
  "regions": [......],
}
```

5.2.4.15 region store <store_id>

用于查询某个 store 上面所有的 Region。

示例：

```
>> region store 2
```

```
{
  "count": 10,
  "regions": [......],
}
```

5.2.4.16 region topread [limit]

用于查询读流量最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topread
```

```
{
  "count": 16,
  "regions": [......],
}
```

5.2.4.17 region topwrite [limit]

用于查询写流量最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topwrite
```

```
{
  "count": 16,
  "regions": [......],
}
```

5.2.4.18 region topconfver [limit]

用于查询 conf version 最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topconfver
```

```
{
  "count": 16,
  "regions": [......],
}
```

5.2.4.19 region topversion [limit]

用于查询 version 最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topversion
```

```
{
  "count": 16,
  "regions": [......],
}
```

5.2.4.20 region topsize [limit]

用于查询 approximate size 最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topsize
```

```
{
  "count": 16,
  "regions": [......],
}
```

5.2.4.21 region check [miss-peer | extra-peer | down-peer | pending-peer | offline-peer | empty-region | hist-size | hist-keys]

用于查询处于异常状态的 Region，各类型的意义如下

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

示例：

```
>> region check miss-peer
```

```
{
  "count": 2,
  "regions": [......],
}
```

5.2.4.22 scheduler [show | add | remove | pause | resume | config]

用于显示和控制调度策略。

示例：

```
>> scheduler show                                    // 显示所有已经创建的 schedulers
>> scheduler add grant-leader-scheduler 1            // 把 store 1 上的所有 Region 的 leader
    ↪ 调度到 store 1
>> scheduler add evict-leader-scheduler 1            // 把 store 1 上的所有 Region 的 leader
    ↪ 从 store 1 调度出去
>> scheduler config evict-leader-scheduler           // v4.0.0 起，展示该调度器具体在哪些
    ↪ store 上
>> scheduler add shuffle-leader-scheduler            // 随机交换不同 store 上的 leader
>> scheduler add shuffle-region-scheduler            // 随机调度不同 store 上的 Region
>> scheduler add evict-slow-store-scheduler          // 当有且仅有一个 slow store 时将该
    ↪ store 上的所有 Region 的 leader 驱逐出去
>> scheduler remove grant-leader-scheduler-1         // 把对应的调度器删掉，`-1` 对应 store
    ↪ ID
>> scheduler pause balance-region-scheduler 10       // 暂停运行 balance-region 调度器 10 秒
>> scheduler pause all 10                            // 暂停运行所有的调度器 10 秒
>> scheduler resume balance-region-scheduler         // 继续运行 balance-region 调度器
>> scheduler resume all                              // 继续运行所有的调度器
>> scheduler config balance-hot-region-scheduler     // 显示 balance-hot-region 调度器的配置
```

5.2.4.23 scheduler config balance-leader-scheduler

用于查看和控制 balance-leader-scheduler 策略。

从 TiDB v6.0.0 起，PD 为 balance-leader-scheduler 引入了 Batch 参数，用于控制 balance-leader 执行任务的速度。你可以通过 pd-ctl 修改 balance-leader batch 配置项设置该功能。

在 v6.0.0 前，PD 不带有该配置（即 balance-leader batch=1）。在 v6.0.0 或更高版本中，balance-leader batch 的默认值为 4。如果你想为该配置项设置大于 4 的值，你需要同时调大scheduler-max-waiting-operator（默认值 5）。同时调大两个配置项后，你才能体验预期的加速效果。

```
>> scheduler config balance-leader-scheduler set batch 3  // 将 balance-leader
    ↪ 调度器可以批量执行的算子大小设置为 3
```

5.2.4.24 scheduler config balance-hot-region-scheduler

用于查看和控制 balance-hot-region-scheduler 策略。

示例：

```
>> scheduler config balance-hot-region-scheduler  // 显示 balance-hot-region 调度器的所有配置
{
  "min-hot-byte-rate": 100,
  "min-hot-key-rate": 10,
  "min-hot-query-rate": 10,
  "max-zombie-rounds": 3,
  "max-peer-number": 1000,
  "byte-rate-rank-step-ratio": 0.05,
  "key-rate-rank-step-ratio": 0.05,
  "query-rate-rank-step-ratio": 0.05,
  "count-rank-step-ratio": 0.01,
  "great-dec-ratio": 0.95,
  "minor-dec-ratio": 0.99,
  "src-tolerance-ratio": 1.05,
  "dst-tolerance-ratio": 1.05,
  "read-priorities": [
    "query",
    "byte"
  ],
  "write-leader-priorities": [
    "key",
    "byte"
  ],
  "write-peer-priorities": [
    "byte",
    "key"
  ],
  "strict-picking-store": "true",
  "enable-for-tiflash": "true"
}
```

- min-hot-byte-rate 指计数的最小字节数，通常为 100。

  ```
  >> scheduler config balance-hot-region-scheduler set min-hot-byte-rate 100
  ```

- min-hot-key-rate 指计数的最小 key 数，通常为 10。

  ```
  >> scheduler config balance-hot-region-scheduler set min-hot-key-rate 10
  ```

- min-hot-query-rate 指计数的最小 query 数，通常为 10。

  ```
  >> scheduler config balance-hot-region-scheduler set min-hot-query-rate 10
  ```

- `max-zombie-rounds` 指一个 operator 可被纳入 pending influence 所允许的最大心跳次数。如果将它设置为更大的值，更多的 operator 可能会被纳入 pending influence。通常用户不需要修改这个值。pending influence 指的是在调度中产生的、但仍生效的影响。

```
>> scheduler config balance-hot-region-scheduler set max-zombie-rounds 3
```

- `max-peer-number` 指最多要被解决的 peer 数量。这个配置可避免调度器处理速度过慢。

```
>> scheduler config balance-hot-region-scheduler set max-peer-number 1000
```

- `byte-rate-rank-step-ratio`、`key-rate-rank-step-ratio`、`query-rate-rank-step-ratio` 和 `count-rank-step-ratio` 分别控制 byte、key、query 和 count 的 step ranks。rank-step-ratio 决定了计算 rank 时的 step 值。`great-dec-ratio` 和 `minor-dec-ratio` 控制 dec 的 rank。通常用户不需要修改这些配置项。

```
>> scheduler config balance-hot-region-scheduler set byte-rate-rank-step-ratio 0.05
```

- `src-tolerance-ratio` 和 `dst-tolerance-ratio` 是期望调度器的配置项。`tolerance-ratio` 的值越小，调度就越容易。当出现冗余调度时，你可以适当调大这个值。

```
>> scheduler config balance-hot-region-scheduler set src-tolerance-ratio 1.1
```

- `read-priorities`、`write-leader-priorities` 和 `write-peer-priorities` 用于控制调度器优先从哪些维度进行热点均衡，支持配置两个维度。
  - `read-priorities` 和 `write-leader-priorities` 用于控制调度器在处理 read 和 write-leader 类型的热点时优先均衡的维度，可选的维度有 query、byte 和 key。
  - `write-peer-priorities` 用于控制调度器在处理 write-peer 类型的热点时优先均衡的维度，支持配置 byte 和 key 维度。

> **注意：**
>
> 若集群的所有组件未全部升级到 v5.2 及以上版本，query 维度的配置不生效，部分组件升级完成后调度器仍默认优先从 byte 和 key 维度进行热点均衡，集群的所有组件全部升级完成后，也会继续保持这样的兼容配置，可通过 pd-ctl 查看实时配置。通常用户不需要修改这些配置项。

```
>> scheduler config balance-hot-region-scheduler set read-priorities query,byte
```

- `strict-picking-store` 是控制热点调度搜索空间的开关，通常为打开。当打开时，热点调度的目标是保证所配置的两个维度的热点均衡。当关闭后，热点调度只保证处于第一优先级的维度的热点均衡表现更好，但可能会导致其他维度的热点不再那么均衡。通常用户不需要修改这个配置项。

```
>> scheduler config balance-hot-region-scheduler set strict-picking-store true
```

- `enable-for-tiflash` 是控制热点调度是否对 TiFlash 生效的开关。通常为打开，关闭后将不会产生 TiFlash 实例之间的热点调度。

```
>> scheduler config balance-hot-region-scheduler set enable-for-tiflash true
```

5.2.4.25 `store [delete | cancel-delete | label | weight | remove-tombstone | limit ] <store_id> [--jq="<query string>"]`

用于显示 store 信息或者删除指定 store。使用 jq 格式化输出请参考jq 格式化 json 输出示例。示例如下。

显示所有 store 信息：

```
>> store
```

```
{
  "count": 3,
  "stores": [...]
}
```

获取 store id 为 1 的 store：

```
>> store 1
```

```
......
```

下线 store id 为 1 的 store：

```
>> store delete 1
```

撤销已使用 store delete 下线并处于 Offline 状态的 store。撤销后，该 store 会从 Offline 状态变为 Up 状态。注意，该命令无法使 Tombstone 状态的 store 变回 Up 状态。以下示例撤销已使用 store delete 下线的 store，其 store id 为 1：

```
>> store cancel-delete 1
```

> 注意：
>
> 若下线过程中切换了 PD leader，需要手动修改 store limit。

设置 store id 为 1 的 store 的键为 “zone” 的 label 的值为 “cn”：

```
>> store label 1 zone cn
```

设置 store id 为 1 的 store 的 leader weight 为 5，Region weight 为 10：

```
>> store weight 1 5 10
```

```
>> store remove-tombstone              // 删除所有 tombstone 状态的 store
>> store limit                         // 显示所有 store 添加和删除 peer 的速度上限
>> store limit add-peer                // 显示所有 store 添加 peer 的速度上限
>> store limit remove-peer             // 显示所有 store 删除 peer 的速度上限
```

```
>> store limit all 5                    // 设置所有 store 添加和删除 peer 的速度上限为每分钟 5 个
>> store limit 1 5                      // 设置 store 1 添加和删除 peer 的速度上限为每分钟 5 个
>> store limit all 5 add-peer           // 设置所有 store 添加 peer 的速度上限为每分钟 5 个
>> store limit 1 5 add-peer             // 设置 store 1 添加 peer 的速度上限为每分钟 5 个
>> store limit 1 5 remove-peer          // 设置 store 1 删除 peer 的速度上限为每分钟 5 个
>> store limit all 5 remove-peer        // 设置所有 store 删除 peer 的速度上限为每分钟 5 个
```

注意：

- store limit 命令原有的 region-add 和 region-remove 子命令已废弃，请使用 add-peer 和 remove-peer 来替代。
- 使用 pd-ctl 可以查看 TiKV 节点的状态信息，即 Up，Disconnect，Offline，Down，或 Tombstone。如需查看各个状态之间的关系，请参考 TiKV Store 状态之间的关系。

5.2.4.26 log [fatal | error | warn | info | debug]

用于设置 PD leader 的日志级别。

```
>> log warn
```

5.2.4.27 tso

用于解析 TSO 到物理时间和逻辑时间。示例如下。

解析 TSO：

```
>> tso 395181938313123110
```

```
system:  2017-10-09 05:50:59 +0800 CST
logic:  120102
```

5.2.4.28 unsafe remove-failed-stores [store-ids | show]

警告：

- 此功能为有损恢复，无法保证数据和数据索引完整性。
- 建议在 TiDB 团队支持下进行相关操作，操作不当可能导致集群难以恢复。

用于在多数副本永久损坏造成数据不可用时进行有损恢复。示例如下。详见 Online Unsafe Recovery。

执行 Online Unsafe Recovery，移除永久损坏的节点 (Store):

```
>> unsafe remove-failed-stores 101,102,103
```

```
Success!
```

显示正在运行的 Online Unsafe Recovery 的当前状态或历史状态。

```
>> unsafe remove-failed-stores show
```

```
[
  "Collecting cluster info from all alive stores, 10/12.",
  "Stores that have reports to PD: 1, 2, 3, ...",
  "Stores that have not reported to PD: 11, 12",
]
```

5.2.5 jq 格式化 JSON 输出示例

5.2.5.1 简化 store 的输出

```
>> store --jq=".stores[].store | { id, address, state_name}"
```

```
{"id":1,"address":"127.0.0.1:20161","state_name":"Up"}
{"id":30,"address":"127.0.0.1:20162","state_name":"Up"}
...
```

5.2.5.2 查询节点剩余空间

```
>> store --jq=".stores[] | {id: .store.id, available: .status.available}"
```

```
{"id":1,"available":"10 GiB"}
{"id":30,"available":"10 GiB"}
...
```

5.2.5.3 查询状态不为 Up 的所有节点

```
>> store --jq='.stores[].store | select(.state_name!="Up") | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Offline"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Offline"}
...
```

5.2.5.4 查询所有的 TiFlash 节点

```
>> store --jq='.stores[].store | select(.labels | length>0 and contains([{"key":"engine","value":
    ↪ "tiflash"}])) | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Up"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Up"}
...
```

5.2.5.5 查询 Region 副本的分布情况

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id]}"
```

```
{"id":2,"peer_stores":[1,30,31]}
{"id":4,"peer_stores":[1,31,34]}
...
```

5.2.5.6 根据副本数过滤 Region

例如副本数不为 3 的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length != 3)}"
```

```
{"id":12,"peer_stores":[30,32]}
{"id":2,"peer_stores":[1,30,31,32]}
```

5.2.5.7 根据副本 store ID 过滤 Region

例如在 store30 上有副本的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==30))}"
```

```
{"id":6,"peer_stores":[1,30,31]}
{"id":22,"peer_stores":[1,30,32]}
...
```

还可以像这样找出在 store30 或 store31 上有副本的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==(30,31)))
    ↪ }"
```

```
{"id":16,"peer_stores":[1,30,34]}
{"id":28,"peer_stores":[1,30,32]}
{"id":12,"peer_stores":[30,32]}
...
```

5.2.5.8 恢复数据时寻找相关 Region

例如当 [store1, store30, store31] 宕机时不可用时，我们可以通过查找所有 Down 副本数量大于正常副本数量的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length as $total
    ↪  | map(if .==(1,30,31) then . else empty end) | length>=$total-length) }"
```

```
{"id":2,"peer_stores":[1,30,31,32]}
{"id":12,"peer_stores":[30,32]}
{"id":14,"peer_stores":[1,30,32]}
...
```

或者在 [store1, store30, store31] 无法启动时，找出 store1 上可以安全手动移除数据的 Region。我们可以这样过滤出所有在 store1 上有副本并且没有其他 DownPeer 的 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length>1 and any
    ↪ (.==1) and all(.!=(30,31)))}"
```

```
{"id":24,"peer_stores":[1,32,33]}
```

在 [store30, store31] 宕机时，找出能安全地通过创建 `remove-peer` Operator 进行处理的所有 Region，即有且仅有一个 DownPeer 的 Region：

```
>> region --jq=".regions[] | {id: .id, remove_peer: [.peers[].store_id] | select(length>1) | map(
    ↪ if .==(30,31) then . else empty end) | select(length==1)}"
```

```
{"id":12,"remove_peer":[30]}
{"id":4,"remove_peer":[31]}
{"id":22,"remove_peer":[30]}
...
```

5.3 TiDB Control 使用说明

TiDB Control 是 TiDB 的命令行工具，用于获取 TiDB 状态信息，多用于调试。本文介绍了 TiDB Control 的主要功能和各个功能的使用方法。

5.3.1 获取 TiDB Control

本节提供了两种方式获取 TiDB Control 工具。

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

5.3.1.1 通过 TiUP 安装

在安装 TiUP 之后，可以使用 `tiup ctl tidb` 命令来获取 TiDB Control 的二进制程序以及运行 TiDB Control。

5.3.1.2 从源代码编译安装

编译环境要求：Go Version 1.13 以上

编译步骤：在 TiDB Control 项目根目录，使用 `make` 命令进行编译，生成 tidb-ctl。

编译文档：帮助文档在 doc 文件夹下，如丢失或需要更新，可通过 `make doc` 命令生成帮助文档。

5.3.2 使用介绍

`tidb-ctl` 的使用由命令（包括子命令）、选项和参数组成。命令即不带 - 或者 -- 的字符，选项即带有 - 或者 -- 的字符，参数即命令或选项字符后紧跟的传递给命令和选项的字符。

如：`tidb-ctl schema in mysql -n db`

- schema: 命令
- in: schema 的子命令
- mysql: in 的参数
- -n: 选项
- db: -n 的参数

目前，TiDB Control 包含以下子命令。

- `tidb-ctl base64decode` 用于 BASE64 解码
- `tidb-ctl decoder` 用于 KEY 解码
- `tidb-ctl etcd` 用于操作 etcd
- `tidb-ctl log` 用于格式化日志文件，将单行的堆栈信息展开
- `tidb-ctl mvcc` 用于获取 MVCC 信息
- `tidb-ctl region` 用于获取 Region 信息
- `tidb-ctl schema` 用于获取 Schema 信息
- `tidb-ctl table` 用于获取 Table 信息

5.3.2.1 获取帮助

`tidb-ctl -h/--help` 用于获取帮助信息。tidb-ctl 由多层命令组成，tidb-ctl 及其所有子命令都可以通过 `-h/--help` 来获取使用帮助。

以获取 Schema 信息为例：

通过 `tidb-ctl schema -h` 可以获取这个子命令的使用帮助。schema 有两个子命令 ——in 和 tid。in 用来通过数据库名获取数据库中所有表的表结构，tid 用来通过全数据库唯一的 table_id 获取表的表结构。

5.3.2.2 全局参数

tidb-ctl 有 4 个与连接相关的全局参数，分别为：

- --host TiDB 服务地址
- --port TiDB status 端口
- --pdhost PD 服务地址
- --pdport PD 服务端口
- --ca 连接使用的 TLS CA 文件路径
- --ssl-key 连接使用的 TLS 密钥文件路径
- --ssl-cert 连接使用的 TLS 证书文件路径

其中 --pdhost 和 --pdport 主要是用于 etcd 子命令，例如：tidb-ctl etcd ddlinfo。如不添加地址和端口将使用默认值，TiDB/PD 服务默认的地址是 127.0.0.1（服务地址只能使用 IP 地址），TiDB 服务端口默认的端口是 10080，PD 服务端口默认的端口是 2379 连接选项是全局选项，适用于以下所有命令。

5.3.2.3 schema 命令

5.3.2.3.1 in 子命令

in 子命令用来通过数据库名获取数据库中所有表的表结构。

tidb-ctl schema in {数据库名}

如：tidb-ctl schema in mysql 将得到以下结果

```
[
    {
        "id": 13,
        "name": {
            "O": "columns_priv",
            "L": "columns_priv"
        },
              ...
        "update_timestamp": 399494726837600268,
        "ShardRowIDBits": 0,
        "Partition": null
    }
]
```

结果将以 json 形式展示，内容较长，这里做了截断。

如希望指定表名，可以使用 tidb-ctl schema in {数据库名} -n {表名} 进行过滤。

如：tidb-ctl schema in mysql -n db 将得到 mysql 库中 db 表的表结构。结果如下：

```
{
    "id": 9,
    "name": {
```

```
        "O": "db",
        "L": "db"
    },
    ...
    "Partition": null
}
```

这里同样做了截断。

如使用的 TiDB 地址不为默认地址和端口，可以使用命令行参数 --host, --port 选项，如：tidb-ctl --host ↪ 172.16.55.88 --port 8898 schema in mysql -n db。

5.3.2.3.2 tid 子命令

tid 子命令用来通过表的 id 获取数据库中表的表结构。

通过使用 in 子命令查询到数据库中表的 id，之后可以通过 tid 子命令查看表的详细信息。

例如，查询到 mysql.stat_meta 表的 id 是 21，可以通过 tidb-ctl schema tid -i 21 查看表的详细信息。

```
{
 "id": 21,
 "name": {
  "O": "stats_meta",
  "L": "stats_meta"
 },
 "charset": "utf8mb4",
 "collate": "utf8mb4_bin",
  ...
}
```

同 in 子命令一样，如果使用的 TiDB 地址不是默认的地址和端口，需要通过 --host 和 --port 参数指定 TiDB 的地址和 status 端口。

5.3.2.4 base64decode 命令

base64decode 用来解码 base64 数据。

基本用法：

```
tidb-ctl base64decode [base64_data]
tidb-ctl base64decode [db_name.table_name] [base64_data]
tidb-ctl base64decode [table_id] [base64_data]
```

1. 准备环境，执行以下 SQL

```
use test;
create table t (a int, b varchar(20),c datetime default current_timestamp , d timestamp
    ↪ default current_timestamp, unique index(a));
```

```
insert into t (a,b,c) values(1,"哈哈 hello",NULL);
alter table t add column e varchar(20);
```

2. 用 HTTP API 接口获取 MVCC 数据

```
curl "http://$IP:10080/mvcc/index/test/t/a/1?a=1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306449994645510,
    "commit_ts": 407306449994645513,
    "short_value": "AAAAAAAAAAE="    # unique index a 存的值是对应行的 handle id.
   }
  ]
 }
}%
```

```
curl "http://$IP:10080/mvcc/key/test/t/1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306588892692486,
    "commit_ts": 407306588892692489,
    "short_value": "CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk="  # handle id 为 1
        ↪ 的行数据。
   }
  ]
 }
}%
```

3. 用 base64decode 解码 handle id (uint64).

```
tidb-ctl base64decode AAAAAAAAAAE=
```

```
hex: 0000000000000001
uint64: 1
```

4. 用 base64decode 解码行数据。

```
./tidb-ctl base64decode test.t CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk=
```

```
a:      1
b:      哈哈 hello
c is NULL
d:      2019-03-28 05:35:30
e not found in data
```

如果 test.t 的 table id 是 60，你也可以使用下列命令获得同样结果：

```
./tidb-ctl base64decode 60 CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk=
```

```
a:      1
b:      哈哈 hello
c is NULL
d:      2019-03-28 05:35:30
e not found in data
```

5.3.2.5 decoder 命令

- 以下示例解码 row key，index key 类似。

```
./tidb-ctl decoder "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\x00\x00\xfa"
```

```
format: table_row
table_id: -9223372036854775780
row_id: -9223372036854775558
```

- 以下示例解码 value

```
./tidb-ctl decoder AhZoZWxsbyB3b3JsZAiAEA==
```

```
format: index_value
index_value[0]: {type: bytes, value: hello world}
index_value[1]: {type: bigint, value: 1024}
```

5.3.2.6 etcd 命令

- tidb-ctl etcd ddlinfo 获取 DDL 信息。
- tidb-ctl etcd putkey KEY VALUE 添加 KEY VALUE 到 etcd（所有的 KEY 会添加到 /tidb/ddl/all_schema_versions/ 之下）。

```
tidb-ctl etcd putkey "foo" "bar"
```

实际是添加 KEY 为 /tidb/ddl/all_schema_versions/foo，VALUE 为 bar 的键值对到 etcd 中。

- `tidb-ctl etcd delkey` 删除 etcd 中的 KEY，只有前缀以 `/tidb/ddl/fg/owner/` 和 `/tidb/ddl/all_schema_versions/` 开头才允许被删除。

```
tidb-ctl etcd delkey "/tidb/ddl/fg/owner/foo" &&
tidb-ctl etcd delkey "/tidb/ddl/all_schema_versions/bar"
```

5.3.2.7 log 命令

TiDB 错误日志的堆栈信息是一行的格式，可以使用 `tidb-ctl log` 将堆栈信息格式化成多行形式。

5.3.2.8 keyrange 命令

keyrange 子命令用于查询全局或表相关的关键 key range 信息，以十六进制形式输出。

- 使用 `tidb-ctl keyrange` 命令查看全局的关键 key range。

```
tidb-ctl keyrange
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
```

- 添加 `--encode` 选项可以显示 encode 过的 key（与 TiKV 及 PD 中的格式相同）。

```
tidb-ctl keyrange --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
```

- 使用 `tidb-ctl keyrange --database={db} --table={tbl}` 命令查看全局和表相关的关键 key range。

```
tidb-ctl keyrange --database test --table ttt
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (74800000000000002f, 748000000000000030)
  table indexes: (7480000000000002f5f69, 7480000000000002f5f72)
    index c2: (7480000000000002f5f698000000000000001, 748000000000000002
        ↪ f5f698000000000000002)
    index c3: (7480000000000002f5f698000000000000002, 748000000000000002
        ↪ f5f698000000000000003)
    index c4: (7480000000000002f5f698000000000000003, 748000000000000002
        ↪ f5f698000000000000004)
  table rows: (7480000000000002f5f72, 748000000000000030)
```

5.4 PD Recover 使用文档

PD Recover 是对 PD 进行灾难性恢复的工具，用于恢复无法正常启动或服务的 PD 集群。

5.4.1 安装 PD Recover

要使用 PD Recover，你可以从源代码编译，也可以直接下载 TiDB 工具包。

5.4.1.1 从源代码编译

- Go：PD Recover 使用了 Go 模块，请安装 Go v1.13 或更新版本。
- 在 PD 根目录下，运行 `make pd-recover` 命令来编译源代码并生成 `bin/pd-recover`。

> **注意：**
>
> 一般来说，用户不需要编译源代码，因为发布的二进制文件或 Docker 中已包含 PD Recover 工具。开发者可以参考以上步骤来编译源代码。

5.4.1.2 下载 TiDB 工具包

PD Recover 的安装包位于 TiDB 离线工具包中。下载方式，请参考 TiDB 工具下载。

5.4.2 快速开始

5.4.2.1 获取 Cluster ID

一般在 PD、TiKV 或 TiDB 的日志中都可以获取 Cluster ID。你可以直接在服务器上查看日志以获取 Cluster ID。

5.4.2.1.1 从 PD 日志获取 Cluster ID（推荐）

使用以下命令，从 PD 日志中获取 Cluster ID：

```
cat {{/path/to}}/pd.log | grep "init cluster id"
```

```
[2019/10/14 10:35:38.880 +00:00] [INFO] [server.go:212] ["init cluster id"] [cluster-id
    ↪ =6747551640615446306]
...
```

或者也可以从 TiDB 或 TiKV 的日志中获取。

5.4.2.1.2 从 TiDB 日志获取 Cluster ID

使用以下命令，从 TiDB 日志中获取 Cluster ID：

```
cat {{/path/to}}/tidb.log | grep "init cluster id"
```

```
2019/10/14 19:23:04.688 client.go:161: [info] [pd] init cluster id 6747551640615446306
...
```

5.4.2.1.3 从 TiKV 日志获取 Cluster ID

使用以下命令，从 TiKV 日志中获取 Cluster ID：

```
cat {{/path/to}}/tikv.log | grep "connect to PD cluster"
```

```
[2019/10/14 07:06:35.278 +00:00] [INFO] [tikv-server.rs:464] ["connect to PD cluster
    ↪ 6747551640615446306"]
...
```

5.4.2.2 获取已分配 ID

在指定已分配 ID 时，需指定一个比当前最大的已分配 ID 更大的值。可以从监控中获取已分配 ID，也可以直接在服务器上查看日志。

5.4.2.2.1 从监控中获取已分配 ID（推荐）

要从监控中获取已分配的 ID，需要确保你所查看的监控指标是上一任 PD Leader 的指标。可从 PD Dashboard 中 Current ID allocation 面板获取最大的已分配 ID。

5.4.2.2.2 从 PD 日志获取已分配 ID

要从 PD 日志中获取分配的 ID，需要确保你所查看的日志是上一任 PD Leader 的日志。运行以下命令获取最大的已分配 ID：

```
cat {{/path/to}}/pd*.log | grep "idAllocator allocates a new id" |  awk -F'=' '{print $2}' | awk
    ↪ -F']' '{print $1}' | sort -r -n | head -n 1
```

```
4000
...
```

你也可以在所有 PD server 中运行上述命令，找到最大的值。

5.4.2.3 部署一套新的 PD 集群

部署新的 PD 集群之前，需要停止当前的 PD 集群，然后删除旧的数据目录（或者用 `--data-dir` 指定新的数据目录）。

5.4.2.4 使用 pd-recover

只需在一个 PD 节点上执行 pd-recover 即可。

```
./pd-recover -endpoints http://10.0.1.13:2379 -cluster-id 6747551640615446306 -alloc-id 10000
```

5.4.2.5 重启整个集群

当出现 recovery is successful 的提示信息时，重启整个集群。

5.4.3 常见问题

5.4.3.1 获取 Cluster ID 时发现有多个 Cluster ID

新建 PD 集群时，会生成新的 Cluster ID。可以通过日志判断旧集群的 Cluster ID。

5.4.3.2 执行 pd-recover 时返回错误 dial tcp 10.0.1.13:2379: connect: connection refused

执行 pd-recover 时需要 PD 提供服务，请先部署并启动 PD 集群。

6 命令行参数

6.1 TiDB 配置参数

在启动 TiDB 时，你可以使用命令行参数或环境变量来配置 TiDB。

要快速了解 TiDB 的参数体系与参数作用域，建议先观看下面的培训视频（时长 17 分钟）。

本文将详细介绍 TiDB 的命令行启动参数。TiDB 的默认端口为 4000（客户端请求）与 10080（状态报告）。

6.1.1 --advertise-address

- 登录 TiDB 的 IP 地址
- 默认：“”
- 必须确保用户和集群中的其他机器都能够访问到该 IP 地址

6.1.2 --config

- 配置文件
- 默认：“”
- 如果你指定了配置文件，TiDB 会首先读取配置文件的配置。如果对应的配置在命令行参数里面也存在，TiDB 就会使用命令行参数的配置来覆盖配置文件中的配置。详细的配置项请参阅TiDB 配置文件描述。

6.1.3 --config-check

- 检查配置文件的有效性并退出
- 默认：false

6.1.4 --config-strict

- 增强配置文件的有效性
- 默认：false

6.1.5 --cors

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

6.1.6 --host

- TiDB 服务监听的 host
- 默认：“0.0.0.0”
- 0.0.0.0 默认会监听所有的网卡地址。如果有多块网卡，可以指定对外提供服务的网卡，如 192.168.100.113

6.1.7 --enable-binlog

- 是否产生 TiDB Binlog
- 默认：false

6.1.8 -L

- Log 级别
- 默认：“info”
- 可选：“debug”，“info”，“warn”，“error”，“fatal”

6.1.9 --lease

- Schema lease 的持续时间。除非你知道更改该值带来的后果，否则你的更改操作是危险的。
- 默认：45s

6.1.10 --log-file

- Log 文件
- 默认：“”
- 如果未设置该参数，log 会默认输出到“stderr”；如果设置了该参数，log 会输出到对应的文件中。

6.1.11 --log-slow-query

- 慢查询日志文件路径
- 默认：“”
- 如果未设置该参数，log 会默认输出到 `--log-file` 指定的文件中

6.1.12 --metrics-addr

- Prometheus Pushgateway 地址
- 默认：“”
- 如果该参数为空，TiDB 不会将统计信息推送给 Pushgateway。参数格式示例：`--metrics-addr =192.168.100.115:9091`

6.1.13 --metrics-interval

- 推送统计信息到 Prometheus Pushgateway 的时间间隔
- 默认：15s
- 设置为 0 表示不推送统计信息给 Pushgateway。示例：`--metrics-interval=2` 指每两秒推送到 Pushgateway

6.1.14 -P

- TiDB 服务监听端口
- 默认：“4000”
- TiDB 服务会使用该端口接受 MySQL 客户端发来的请求

6.1.15 --path

- 对于本地存储引擎“unistore”来说，path 指定的是实际的数据存放路径
- 当 `--store = tikv` 时，必须指定 path；当 `--store = unistore` 时，如果不指定 path，会使用默认值。
- 对于“TiKV”存储引擎来说，path 指定的是实际的 PD 地址。假如在 192.168.100.113:2379、192.168.100.114:2379 和 192.168.100.115:2379 上面部署了 PD，那么 path 为“192.168.100.113:2379, 192.168.100.114:2379, 192.168.100.115:2379”
- 默认：“/tmp/tidb”
- 可以通过 `tidb-server --store=unistore --path=""` 来启动一个纯内存引擎的 TiDB

6.1.16 --proxy-protocol-networks

- 允许使用 PROXY 协议连接 TiDB 的代理服务器地址列表。
- 默认：“”
- 通常情况下，通过反向代理使用 TiDB 时，TiDB 会将反向代理服务器的 IP 地址视为客户端 IP 地址。对于支持 PROXY 协议的反向代理（如 HAProxy），开启 PROXY 协议后能让反向代理透传客户端真实的 IP 地址给 TiDB。
- 配置该参数后，TiDB 将允许配置的源 IP 地址使用 PROXY 协议连接到 TiDB，且拒绝这些源 IP 地址使用非 PROXY 协议连接。若该参数为空，则任何源 IP 地址都不能使用 PROXY 协议连接到 TiDB。地址可以使用 IP 地址格式 (192.168.1.50) 或者 CIDR 格式 (192.168.1.0/24)，并可用 , 分隔多个地址，或用 * 代表所有 IP 地址。

> **警告：**
>
> 需谨慎使用 * 符号，因为它可能引入安全风险，允许来自任何 IP 的客户端自行汇报其 IP 地址。另外，它可能会导致部分直接连接 TiDB 的内部组件无法使用，例如 TiDB Dashboard。

6.1.17 --proxy-protocol-header-timeout

- PROXY Protocol 请求头读取超时时间
- 默认：5
- 单位：秒

> **注意：**
>
> 不要将该参数配置为 0。除非特殊情况，一般使用默认值即可。

6.1.18 --report-status

- 用于打开或者关闭服务状态监听端口
- 默认：true
- 将参数值设置为 true 表明开启状态监听端口；设置为 false 表明关闭状态监听端口

6.1.19 --run-ddl

- tidb-server 是否运行 DDL 语句，集群内至少需要有一台 tidb-server 设置该参数
- 默认：true
- 值可以为 true 或者 false。设置为 true 表明自身会运行 DDL；设置为 false 表明自身不会运行 DDL

6.1.20 --socket string

- TiDB 服务使用 unix socket file 方式接受外部连接
- 默认：“”
- 例如可以使用 “/tmp/tidb.sock” 来打开 unix socket file

6.1.21 --status

- TiDB 服务状态监听端口
- 默认：“10080”
- 该端口用于展示 TiDB 内部数据，包括 prometheus 统计和 pprof
- Prometheus 统计可以通过 http://host:status_port/metrics 访问
- pprof 数据可以通过 http://host:status_port/debug/pprof 访问

6.1.22 --status-host

- TiDB 服务状态监听 host
- 默认：“0.0.0.0”

6.1.23 --store

- 用来指定 TiDB 底层使用的存储引擎
- 默认：“unistore”
- 可以选择 “unistore”（本地存储引擎）或者 “tikv”（分布式存储引擎）

6.1.24 --token-limit

- TiDB 中同时允许运行的 Session 数量，用于流量控制
- 默认：1000
- 如果当前运行的连接多于该 token-limit，那么请求会阻塞，等待已经完成的操作释放 Token

6.1.25 -V

- 输出 TiDB 的版本
- 默认：“”

6.1.26 --plugin-dir

- plugin 存放目录
- 默认：“/data/deploy/plugin”

6.1.27 --plugin-load

- 需要加载的 plugin 名称，多个 plugin 以 “,” 逗号分隔
- 默认：“”

6.1.28 --affinity-cpus

- 设置 TiDB server CPU 亲和性，以 “,” 逗号分隔，例如 “1,2,3”
- 默认：“”

6.1.29 --repair-mode

- 是否开启修复模式，仅用于数据修复场景
- 默认：false

6.1.30 --repair-list

- 修复模式下需要修复的表名
- 默认：“”

6.2 TiKV 配置参数

TiKV 的命令行参数支持一些可读性好的单位转换。

- 文件大小（以 bytes 为单位）：KB, MB, GB, TB, PB（也可以全小写）
- 时间（以毫秒为单位）：ms, s, m, h

6.2.1 -A, --addr

- TiKV 监听地址。
- 默认：“127.0.0.1:20160”
- 如果部署一个集群，--addr 必须指定当前主机的 IP 地址，例如“192.168.100.113:20160”，如果是运行在 docker 则需要指定为“0.0.0.0:20160”。

6.2.2 --advertise-addr

- TiKV 对外访问地址。
- 默认：${addr}
- 在某些情况下，比如 Docker 或者 NAT 网络环境，客户端并不能通过 --addr 的地址来访问到 TiKV。这时候，你可以设置 --advertise-addr 来让客户端访问 TiKV。
- 例如，docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20160:20160，那么可以设置为 --advertise-addr=“192.168.100.113:20160”，客户端可以通过 192.168.100.113:20160 来找到这个服务。

6.2.3 --status-addr

- TiKV 服务状态监听端口。
- 默认：“20180”
- Prometheus 统计可以通过 http://host:status_port/metrics 访问。
- Profile 数据可以通过 http://host:status_port/debug/pprof/profile 访问。

6.2.4 --advertise-status-addr

- TiKV 对外访问服务状态地址。
- 默认：使用 --status-addr
- 在某些情况下，例如 docker 或者 NAT 网络环境，客户端并不能通过 --status-addr 的地址来访问到 TiKV。此时，你可以设置 --advertise-status-addr 来让客户端访问 TiKV。
- 例如，Docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20180:20180，那么可以设置 --advertise-status-addr="192.168.100.113:20180"，客户端可以通过 192.168.100.113:20180 来找到这个服务。

6.2.5 -C, --config

- 配置文件。

- 默认：“”
- 如果你指定了配置文件，TiKV 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，TiKV 就会使用命令行参数的配置来覆盖配置文件里面的。

6.2.6 --capacity

- TiKV 存储数据的容量。
- 默认：0（无限）
- PD 需要使用这个值来对整个集群做 balance 操作。（提示：你可以使用 10GB 来替代 10737418240，从而简化参数的传递）。

6.2.7 --data-dir

- TiKV 数据存储路径。
- 默认：“/tmp/tikv/store”

6.2.8 -L

- Log 级别。
- 默认：“info”
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”

6.2.9 --log-file

- Log 文件。
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面。

6.2.10 --pd

- PD 地址列表。
- 默认：“”
- TiKV 必须使用这个值连接 PD，才能正常工作。使用逗号来分隔多个 PD 地址，例如：192.168.100.113:2379, 192.168.100.114:2379, 192.168.100.115:2379。

6.3 TiFlash 命令行参数

本文介绍了 TiFlash 的命令行启动参数。

6.3.1 server --config-file

- 指定 TiFlash 的配置文件路径
- 默认：“”
- 必须指定配置文件，详细的配置项请参阅TiFlash 配置参数

6.3.2 dttool migrate

- 迁移 DTFile 的文件格式（用于测试和原地降级）。数据迁移的单位为单个 DTFile。如果想进行整表迁移，通常需要定位到所有形如 `<data dir>/t_<table id>/stable/dmf_<file id>` 的路径，逐一进行迁移。可以结合脚本来自动进行这一操作。
- 使用场景：
  - 当需要从开启了数据校验功能 (`storage.format_version` >= 3) 的 TiFlash v5.4 及以上版本降级回以前的版本时，可以使用此工具完成数据格式降级。
  - 当升级到 TiFlash v5.4 及以上，并希望对存量数据也加上数据校验功能以加固数据检验时，可以使用此工具完成数据格式升级。
  - 测试不同配置的 DTFile 空间占用和读取速度。
- 参数：
  - `--imitative`：当不使用 DTFile 的加密功能时，可以使用本选项避免使用配置文件和连接 PD。
  - `--version`：DTFile 的版本，可选值为 1 和 2，默认为 2。1 为传统格式，2 为新版 checksum 对应的 DTFile 格式。
  - `--algorithm`：检验哈希算法，可选值为 xxh3，city128，crc32，crc64，none，默认为 xxh3，仅在 version=2 时有用。
  - `--frame`：校验帧大小，默认为 1048576，仅在 version=2 时有用。
  - `--compression`：目标压缩算法，可选值为 LZ4（默认）、LZ4HC、zstd 和 none。
  - `--level`：目标压缩等级，不指定则根据压缩算法默认使用推荐的压缩级别。如果 `compression` 设置为 LZ4 或 `zstd`，则默认设置为 1；如果 `compression` 设置为 LZ4HC，则默认设置为 9。
  - `--config-file`：dttool migrate 的配置文件应当与 server 模式下的配置文件保持一致；使用配置文件时，需要退出本地的 TiFlash 服务器实例。见 `--imitative` 选项。
  - `--file-id`：对应 DTFile 的 ID，如 `dmf_123` 对应的 ID 是 123。
  - `--workdir`：指向 `dmf_xxx` 的父级目录。
  - `--dry`：空跑模式，只输出迁移过程。
  - `--nokeep`：不保留原数据。不开启该选项时，会产生 `dmf_xxx.old` 文件。

> **警告：**
>
> 虽然 TiFlash 可以读取自定义压缩算法和压缩等级的 DTFile，但目前正式支持的只有默认压缩等级的 LZ4 算法。自定义压缩参数并未经过大量测试，仅作实验。

> **注意：**
>
> 为保证安全 DTTool 在迁移模式下会尝试对工作目录进行加锁，因此同一工作目录下同一时间只能有一个 DTTool 执行迁移工作。如果您在中途强制停止 DTTool，可能会因锁未释放导致后面在运行 DTTool 时工具拒绝进行迁移工作。如果您遇到这种情况，在保证安全的前提下，可以手动删除工作目录下的 LOCK 文件来释放锁。

6.3.3 dttool bench

- 提供 DTFile 的简单 IO 速度测试。
- 参数：
  - `--version`：DTFile 的版本，见dttool migrate 对应参数。
  - `--algorithm`：检验哈希算法，见dttool migrate 对应参数。
  - `--frame`：校验帧大小，见dttool migrate 对应参数。
  - `--column`：测试表宽度，默认为 100。
  - `--size`：测试表长度，默认为 1000。
  - `--field`：测试表字段长度上限，默认为 1024。
  - `--random`：随机数种子。如未提供，该值从系统熵池抽取。
  - `--encryption`：启用加密功能。
  - `--repeat`：性能测试采样次数，默认为 5。
  - `--workdir`：临时数据文件夹，应指向需要测试的文件系统下的路径，默认为 /tmp/test。

6.3.4 dttool inspect

- 检查 DTFile 的完整性。数据校验的单位为单个 DTFile。如果想进行整表校验，通常需要定位到所有形如 `<data dir>/t_<table id>/stable/dmf_<file id>` 的路径，逐一进行校验。可以结合脚本来自动进行这一操作。
- 使用场景：
  - 完成格式升降级后进行完整性检测。
  - 将原有数据文件搬迁至新环境后进行完整性检测。
- 参数：
  - `--config-file`：dttool bench 的配置文件，见dttool migrate 对应参数。
  - `--check`：进行哈希校验。
  - `--file-id`：对应 DTFile 的 ID，见dttool migrate 对应参数。
  - `--imitative`：模拟数据库上下文，见dttool migrate 对应参数。
  - `--workdir`：数据文件夹，见dttool migrate 对应参数。

6.4 PD 配置参数

PD 可以通过命令行参数或环境变量配置。

6.4.1 --advertise-client-urls

- 用于外部访问 PD 的 URL 列表。
- 默认：`${client-urls}`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的 client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2379:2379`，那么可以设置为 `--advertise-client-urls="http://192.168.100.113:2379"`，客户端可以通过 `http://192.168.100.113:2379` 来找到这个服务。

6.4.2 --advertise-peer-urls

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认：${peer-urls}
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的 peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问。
- 例如，Docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 2380:2380，那么可以设置为 advertise-peer-urls="http://192.168.100.113:2380"，其他 PD 节点可以通过 http://192.168.100.113:2380 来找到这个服务。

6.4.3 --client-urls

- PD 监听的客户端 URL 列表。
- 默认：http://127.0.0.1:2379
- 如果部署一个集群，--client-urls 必须指定当前主机的 IP 地址，例如 http://192.168.100.113:2379"，如果是运行在 Docker 则需要指定为 http://0.0.0.0:2379。

6.4.4 --peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认：http://127.0.0.1:2380
- 如果部署一个集群，--peer-urls 必须指定当前主机的 IP 地址，例如 http://192.168.100.113:2380，如果是运行在 Docker 则需要指定为 http://0.0.0.0:2380。

6.4.5 --config

- 配置文件。
- 默认：“”
- 如果你指定了配置文件，PD 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，PD 就会使用命令行参数的配置来覆盖配置文件里面的。

6.4.6 --data-dir

- PD 存储数据路径。
- 默认：default.${name}

6.4.7 --initial-cluster

- 初始化 PD 集群配置。
- 默认："{name}=http://{advertise-peer-url}"
- 例如，如果 name 是 “pd”, 并且 advertise-peer-urls 是 http://192.168.100.113:2380，那么 initial-cluster 就是 pd=http://192.168.100.113:2380。
- 如果你需要启动三台 PD，那么 initial-cluster 可能就是 pd1=http://192.168.100.113:2380, pd2=http://192.168.100.114:2380, pd3=192.168.100.115:2380。

6.4.8 --join

- 动态加入 PD 集群。
- 默认：“”
- 如果你想动态将一台 PD 加入集群，你可以使用 --join="${advertise-client-urls}"，advertise-client-url 是当前集群里面任意 PD 的 advertise-client-url，你也可以使用多个 PD 的，需要用逗号分隔。

6.4.9 -L

- Log 级别。
- 默认：“info”
- 可选：“debug”，“info”，“warn”，“error”，“fatal”

6.4.10 --log-file

- Log 文件。
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面。

6.4.11 --log-rotate

- 是否开启日志切割。
- 默认：true
- 当值为 true 时，按照 PD 配置文件中 [log.file] 信息执行。

6.4.12 --name

- 当前 PD 的名字。
- 默认：“pd”
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字

6.4.13 --cacert

- CA 文件路径，用于开启 TLS。
- 默认：“”

6.4.14 --cert

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

6.4.15 --key

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

6.4.16 --metrics-addr

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

6.4.17 --force-new-cluster

- 强制使用当前节点创建新的集群。
- 默认：false
- 仅用于在 PD 丢失多数副本的情况下恢复服务，可能会产生部分数据丢失。

6.4.18 -V, --version

- 输出版本信息并退出。

7 配置文件参数

7.1 TiDB 配置文件描述

TiDB 配置文件比命令行参数支持更多的选项。你可以在 config/config.toml.example 找到默认值的配置文件，重命名为 config.toml 即可。本文档只介绍未包含在命令行参数中的参数。

7.1.0.1 split-table

- 为每个 table 建立单独的 Region。
- 默认值：true
- 如果需要创建大量的表，建议将此参数设置为 false。

7.1.0.2 token-limit

- 可以同时执行请求的 session 个数
- 类型：Integer
- 默认值：1000
- 最小值：1
- 最大值（64 位平台）：18446744073709551615
- 最大值（32 位平台）：4294967295

7.1.0.3 oom-use-tmp-storage

- 设置是否在单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时为某些算子启用临时磁盘。
- 默认值：true

7.1.0.4 tmp-storage-path

- 单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时，某些算子的临时磁盘存储位置。
- 默 认 值：<操作系统临时文件夹>/<操作系统用户ID>_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-storage。 其 中 MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA= 是 对 <host>:<port>/<statusHost>:<statusPort> 进行 Base64 编码的输出结果。
- 此配置仅在 oom-use-tmp-storage 为 true 时有效。

7.1.0.5 tmp-storage-quota

- tmp-storage-path 存储使用的限额。
- 单位：Byte
- 当单条 SQL 语句使用临时磁盘，导致 TiDB server 的总体临时磁盘总量超过 tmp-storage-quota 时，当前 SQL 操作会被取消，并返回 Out Of Global Storage Quota! 错误。
- 当 tmp-storage-quota 小于 0 时则没有上述检查与限制。
- 默认值：-1
- 当 tmp-storage-path 的剩余可用容量低于 tmp-storage-quota 所定义的值时，TiDB server 启动时将会报出错误并退出。

7.1.0.6 lease

- DDL 租约超时时间。
- 默认值：45s
- 单位：秒

7.1.0.7 compatible-kill-query

- 设置 KILL 语句的兼容性。
- 默认值：false
- TiDB 中 KILL xxx 的行为和 MySQL 中的行为不相同。为杀死一条查询，在 TiDB 里需要加上 TIDB 关键词，即 KILL TIDB xxx。但如果把 compatible-kill-query 设置为 true，则不需要加上 TIDB 关键词。
- 这种区别很重要，因为当用户按下 Ctrl+C 时，MySQL 命令行客户端的默认行为是：创建与后台的新连接，并在该新连接中执行 KILL 语句。如果负载均衡器或代理已将该新连接发送到与原始会话不同的 TiDB 服务器实例，则该错误会话可能被终止，从而导致使用 TiDB 集群的业务中断。只有当您确定在 KILL 语句中引用的连接正好位于 KILL 语句发送到的服务器上时，才可以启用 compatible-kill-query。

7.1.0.8 check-mb4-value-in-utf8

- 开启检查 utf8mb4 字符的开关，如果开启此功能，字符集是 utf8，且在 utf8 插入 mb4 字符，系统将会报错。
- 默认值：true

7.1.0.9 treat-old-version-utf8-as-utf8mb4

- 将旧表中的 utf8 字符集当成 utf8mb4 的开关。
- 默认值：true

7.1.0.10 alter-primary-key（已废弃）

- 用于控制添加或者删除主键功能。
- 默认值：false
- 默认情况下，不支持增删主键。将此变量被设置为 true 后，支持增删主键功能。不过对在此开关开启前已经存在的表，且主键是整型类型时，即使之后开启此开关也不支持对此列表删除主键。

> 注意：
>
> 该配置项已被废弃，目前仅在 `@@tidb_enable_clustered_index` 取值为 `INT_ONLY` 时生效。如果需要增删主键，请在建表时使用 `NONCLUSTERED` 关键字代替。要了解关于 `CLUSTERED` 主键的详细信息，请参考聚簇索引。

7.1.0.11 server-version

- 用来修改 TiDB 在以下情况下返回的版本号：
  - 当使用内置函数 `VERSION()` 时。
  - 当与客户端初始连接，TiDB 返回带有服务端版本号的初始握手包时。具体可以查看 MySQL 初始握手包的描述。
- 默认值：“”
- 默认情况下，TiDB 版本号格式为：`5.7.${mysql_latest_minor_version}-TiDB-${tidb_version}`。

7.1.0.12 repair-mode

- 用于开启非可信修复模式，启动该模式后，可以过滤 `repair-table-list` 名单中坏表的加载。
- 默认值：false
- 默认情况下，不支持修复语法，默认启动时会加载所有表信息。

7.1.0.13 repair-table-list

- 配合 repair-mode 为 true 时使用，用于列出实例中需要修复的坏表的名单，该名单的写法为 [“db.table1”, “db.table2”, ……]。
- 默认值：[]
- 默认情况下，该 list 名单为空，表示没有所需修复的坏表信息。

7.1.0.14 new_collations_enabled_on_first_bootstrap

- 用于开启新的 collation 支持
- 默认值：true
- 注意：该配置项只有在初次初始化集群时生效，初始化集群后，无法通过更改该配置项打开或关闭新的 collation 框架；4.0 版本之前的 TiDB 集群升级到 4.0 或更高版本时，由于集群已经初始化过，该参数无论如何配置，都作为 false 处理。

7.1.0.15 max-server-connections

- TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认值：0
- 默认情况下，TiDB 不限制客户端连接数。当本配置项的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。

7.1.0.16 max-index-length

- 用于设置新建索引的长度限制。
- 默认值：3072
- 单位：Byte
- 目前的合法值范围 [3072，3072*4]。MySQL 和 TiDB v3.0.11 之前版本（不包含 v3.0.11）没有此配置项，不过都对新建索引的长度做了限制。MySQL 对此的长度限制为 3072，TiDB 在 v3.0.7 以及之前版本该值为 3072*4，在 v3.0.7 之后版本（包含 v3.0.8、v3.0.9 和 v3.0.10）的该值为 3072。为了与 MySQL 和 TiDB 之前版本的兼容，添加了此配置项。

7.1.0.17 table-column-count-limit 从 v5.0 版本开始引入

- 用于设置单个表中列的数量限制
- 默认值：1017
- 目前的合法值范围 [1017，4096]。

7.1.0.18 index-limit 从 v5.0 版本开始引入

- 用于设置单个表中索引的数量限制
- 默认值：64
- 目前的合法值范围 [64，512]。

7.1.0.19 enable-telemetry 从 v4.0.2 版本开始引入

- 是否开启 TiDB 遥测功能。
- 默认值：true
- 如果所有 TiDB 实例上该选项都设置为 false，那么将完全禁用 TiDB 遥测功能，且忽略tidb_enable_telemetry 系统变量。参阅遥测了解该功能详情。

7.1.0.20 enable-tcp4-only 从 v5.0 版本开始引入

- 控制是否只监听 TCP4。
- 默认值：false
- 当使用 LVS 为 TiDB 做负载均衡时，可开启此配置项。这是因为 LVS 的 TOA 模块可以通过 TCP4 协议从 TCP 头部信息中解析出客户端的真实 IP。

7.1.0.21 enable-enum-length-limit 从 v5.0 版本开始引入

- 是否限制单个 ENUM 元素和单个 SET 元素的最大长度
- 默认值：true
- 当该配置项值为 true 时，ENUM 和 SET 单个元素的最大长度为 255 个字符，与 MySQL 8 兼容；当该配置项值为 false 时，不对单个元素的长度进行限制，与 TiDB v5.0 之前的版本兼容。

7.1.0.22 graceful-wait-before-shutdown 从 v5.0 版本开始引入

- 指定关闭服务器时 TiDB 等待的秒数，使得客户端有时间断开连接。
- 默认值：0
- 在 TiDB 等待服务器关闭期间，HTTP 状态会显示失败，使得负载均衡器可以重新路由流量。

7.1.0.23 enable-global-kill 从 v6.1.0 版本开始引入

- 用于开启 Global Kill（跨节点终止查询或连接）功能。
- 默认值：true
- 当该配置项值为 true 时，KILL 语句和 KILL TIDB 语句均能跨节点终止查询或连接，无需担心错误地终止其他查询或连接。当你使用客户端连接到任何一个 TiDB 节点执行 KILL 语句或 KILL TIDB 语句时，该语句会被转发给对应的 TiDB 节点。当客户端和 TiDB 中间有代理时，KILL 语句或 KILL TIDB 语句也会被转发给对应的 TiDB 节点执行。目前暂时不支持在 enable-global-kill 为 true 时用 MySQL 命令行 ctrl+c 终止查询或连接。关于 KILL 语句的更多信息，请参考KILL [TIDB]。

7.1.1 log

日志相关的配置项。

7.1.1.1 level

- 指定日志的输出级别，可选项为 [debug, info, warn, error, fatal]
- 默认值：“info”

7.1.1.2 format

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

7.1.1.3 enable-timestamp

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

> 注意：
>
> - 考虑后向兼容性，原来的配置项 disable-timestamp 仍然有效，但如果和 enable-timestamp 配置的值在语义上冲突（例如在配置中把 enable-timestamp 和 disable-timestamp 同时设置为 true），则 TiDB 会忽略 disable-timestamp 的值。
> - 当前 TiDB 默认使用 disable-timestamp 来决定是否在日志中输出时间戳，此时 enable-timestamp 的值为 null。
> - 在未来的版本中，disable-timestamp 配置项将被彻底移除，请废弃 disable-timestamp 的用法，使用语义上更易于理解的 enable-timestamp。

7.1.1.4 enable-slow-log

- 是否开启慢查询日志
- 默认值：true
- 可以设置成 true 或 false 来启用或或禁用慢查询日志。

7.1.1.5 slow-query-file

- 慢查询日志的文件名。
- 默认值：“tidb-slow.log”。注：由于 TiDB V2.1.8 更新了慢日志格式，所以将慢日志单独输出到了慢日志文件。V2.1.8 之前的版本，该变量的默认值是 “”。
- 设置后，慢查询日志会单独输出到该文件。

7.1.1.6 slow-threshold

- 输出慢日志的耗时阈值。
- 默认值：300ms
- 当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。

7.1.1.7 record-plan-in-slow-log

- 在慢日志中记录执行计划
- 默认值：1
- 0 表示关闭，1 表示开启，默认开启，该值作为系统变量tidb_record_plan_in_slow_log 的初始值。

7.1.1.8 expensive-threshold

- 输出 expensive 操作的行数阈值。
- 默认值：10000
- 当查询的行数（包括中间结果，基于统计信息）大于这个值，该操作会被认为是 expensive 查询，并输出一个前缀带有 [EXPENSIVE_QUERY] 的日志。

7.1.2 log.file

日志文件相关的配置项。

7.1.2.0.1 filename

- 一般日志文件名字。
- 默认值：“”
- 如果设置，会输出一般日志到这个文件。

7.1.2.0.2 max-size

- 日志文件的大小限制。
- 默认值：300
- 单位：MB
- 最大设置上限为 4096。

7.1.2.0.3 max-days

- 日志最大保留的天数。
- 默认值：0
- 默认不清理；如果设置了参数值，在 max-days 之后 TiDB 会清理过期的日志文件。

7.1.2.0.4 max-backups

- 保留的日志的最大数量。
- 默认值：0
- 默认全部保存；如果设置为 7，会最多保留 7 个老的日志文件。

7.1.3 security

安全相关配置。

7.1.3.1 enable-sem

- 启用安全增强模式 (SEM)。
- 默认值：`false`
- 可以通过系统变量`tidb_enable_enhanced_security`获取安全增强模式的状态。

7.1.3.2 ssl-ca

- PEM 格式的受信任 CA 的证书文件路径。
- 默认值：“”
- 当同时设置了该选项和 `--ssl-cert`、`--ssl-key` 选项时，TiDB 将在客户端出示证书的情况下根据该选项指定的受信任的 CA 列表验证客户端证书。若验证失败，则连接会被终止。
- 即使设置了该选项，若客户端没有出示证书，则安全连接仍然继续，不会进行客户端证书验证。

7.1.3.3 ssl-cert

- PEM 格式的 SSL 证书文件路径。
- 默认值：“”
- 当同时设置了该选项和 `--ssl-key` 选项时，TiDB 将接受（但不强制）客户端使用 TLS 安全地连接到 TiDB。
- 若指定的证书或私钥无效，则 TiDB 会照常启动，但无法接受安全连接。

7.1.3.4 ssl-key

- PEM 格式的 SSL 证书密钥文件路径，即 `--ssl-cert` 所指定的证书的私钥。
- 默认值：“”
- 目前 TiDB 不支持加载由密码保护的私钥。

7.1.3.5 cluster-ssl-ca

- CA 根证书，用于用 tls 连接 TiKV/PD
- 默认值：“”

7.1.3.6 cluster-ssl-cert

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

7.1.3.7 cluster-ssl-key

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

7.1.3.8 spilled-file-encryption-method

- 内存落盘文件的加密方式。
- 默认值：`"plaintext"`，表示不进行加密。
- 可选值：`"plaintext"`、`"aes128-ctr"`。

7.1.3.9 auto-tls

- 控制 TiDB 启动时是否自动生成 TLS 证书。
- 默认值：`false`

7.1.3.10 tls-version

- 设置用于连接 MySQL 协议的最低 TLS 版本。
- 默认值：“”，支持 TLSv1.1 及以上版本。
- 可选值：`"TLSv1.0"`、`"TLSv1.1"`、`"TLSv1.2"` 和 `"TLSv1.3"`

7.1.4 performance

性能相关配置。

7.1.4.1 max-procs

- TiDB 的 CPU 使用数量。
- 默认值：0
- 默认值为 0 表示使用机器上所有的 CPU；如果设置成 n，那么 TiDB 会使用 n 个 CPU 数量。

7.1.4.2 server-memory-quota 从 v4.0.9 版本开始引入

> **警告：**
>
> `server-memory-quota` 目前为实验性特性，不建议在生产环境中使用。

- 设置 tidb-server 实例的最大内存用量，单位为字节。
- 默认值：0
- 默认值为 0 表示无内存限制。

7.1.4.3 memory-usage-alarm-ratio 从 v4.0.9 版本开始引入

- tidb-server 实例内存使用占总内存的比例超过一定阈值时会报警。该配置项的有效范围为 0 到 1。如果配置该选项为 0 或 1，则表示关闭内存阈值报警功能。
- 默认值：0.8
- 当内存阈值报警功能开启时，如果配置项server-memory-quota 未设置，则内存报警阈值为 memory-usage-alarm-ratio * 系统内存大小；如果 server-memory-quota 被设置且大于 0，则内存报警阈值为 memory-usage-alarm-ratio * server-memory-quota。
- 当 TiDB 检测到 tidb-server 的内存使用超过了阈值，则会认为存在内存溢出的风险，会将当前正在执行的所有 SQL 语句中内存使用最高的 10 条语句和运行时间最长的 10 条语句以及 heap profile 记录到目录tmp-storage-path/record 中，并输出一条包含关键字 tidb-server has the risk of OOM 的日志。
- 该值作为系统变量tidb_memory_usage_alarm_ratio 的初始值。

7.1.4.4 txn-entry-size-limit 从 v5.0 版本开始引入

- TiDB 单行数据的大小限制
- 默认值：6291456
- 单位：Byte
- 事务中单个 key-value 记录的大小限制。若超出该限制，TiDB 将会返回 entry too large 错误。该配置项的最大值不超过 125829120（表示 120MB）。
- 注意，TiKV 有类似的限制。若单个写入请求的数据量大小超出raft-entry-max-size，默认为 8MB，TiKV 会拒绝处理该请求。当表的一行记录较大时，需要同时修改这两个配置。

7.1.4.5 txn-total-size-limit

- TiDB 单个事务大小限制
- 默认值：104857600
- 单位：Byte
- 单个事务中，所有 key-value 记录的总大小不能超过该限制。该配置项的最大值不超过 1099511627776（表示 1TB）。注意，如果使用了以 Kafka 为下游消费者的 binlog，如：arbiter 集群，该配置项的值不能超过 1073741824（表示 1GB），因为这是 Kafka 的处理单条消息的最大限制，超过该限制 Kafka 将会报错。

7.1.4.6 max-txn-ttl

- 单个事务持锁的最长时间，超过该时间，该事务的锁可能会被其他事务清除，导致该事务无法成功提交。
- 默认值：3600000
- 单位：毫秒
- 超过此时间的事务只能执行提交或者回滚，提交不一定能够成功。

7.1.4.7 stmt-count-limit

- TiDB 单个事务允许的最大语句条数限制。

- 默认值：5000
- 在一个事务中，超过 `stmt-count-limit` 条语句后还没有 rollback 或者 commit，TiDB 将会返回 `statement count 5001 exceeds the transaction limitation, autocommit = false` 错误。该限制只在可重试的乐观事务中生效，如果使用悲观事务或者关闭了事务重试，事务中的语句数将不受此限制。

7.1.4.8 tcp-keep-alive

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

7.1.4.9 tcp-no-delay

- 控制 TiDB 是否在 TCP 层开启 TCP_NODELAY。开启后，TiDB 将禁用 TCP/IP 协议中的 Nagle 算法，允许小数据包的发送，可以降低网络延时，适用于延时敏感型且数据传输量比较小的应用。
- 默认值：true

7.1.4.10 cross-join

- 默认值：true
- 默认可以执行在做 join 时两边表没有任何条件（where 字段）的语句；如果设置为 false，则有这样的 join 语句出现时，server 会拒绝执行

7.1.4.11 stats-lease

- TiDB 重载统计信息，更新表行数，检查是否需要自动 analyze，利用 feedback 更新统计信息以及加载列的统计信息的时间间隔。
- 默认值：3s
  - 每隔 `stats-lease` 时间，TiDB 会检查统计信息是否有更新，如果有会将其更新到内存中
  - 每隔 `20 * stats-lease` 时间，TiDB 会将 DML 产生的总行数以及修改的行数变化更新到系统表中
  - 每隔 `stats-lease` 时间，TiDB 会检查是否有表或者索引需要自动 analyze
  - 每隔 `stats-lease` 时间，TiDB 会检查是否有列的统计信息需要被加载到内存中
  - 每隔 `200 * stats-lease` 时间，TiDB 会将内存中缓存的 feedback 写入系统表中
  - 每隔 `5 * stats-lease` 时间，TiDB 会读取系统表中的 feedback，更新内存中缓存的统计信息
- 当 `stats-lease` 为 0s 时，TiDB 会以 3s 的时间间隔周期性的读取系统表中的统计信息并更新内存中缓存的统计信息。但不会自动修改统计信息相关系统表，具体来说，TiDB 不再自动修改这些表：
  - `mysql.stats_meta`：TiDB 不再自动记录事务中对某张表的修改行数，也不会更新到这个系统表中
  - `mysql.stats_histograms`/`mysql.stats_buckets` 和 `mysql.stats_top_n`：TiDB 不再自动 analyze 和主动更新统计信息
  - `mysql.stats_feedback`：TiDB 不再根据被查询的数据反馈的部分统计信息更新表和索引的统计信息

7.1.4.12 feedback-probability

- TiDB 对查询收集统计信息反馈的概率。
- 默认值：0
- 此功能默认关闭，暂不建议开启。如果开启此功能，对于每一个查询，TiDB 会以 feedback-probability 的概率收集查询的反馈，用于更新统计信息。

7.1.4.13 query-feedback-limit

- 在内存中缓存的最大 Query Feedback 数量，超过这个数量的 Feedback 会被丢弃。
- 默认值：512

7.1.4.14 pseudo-estimate-ratio

- 修改过的行数/表的总行数的比值，超过该值时系统会认为统计信息已经过期，会采用 pseudo 的统计信息。
- 默认值：0.8
- 最小值：0
- 最大值：1

7.1.4.15 force-priority

- 把所有的语句优先级设置为 force-priority 的值。
- 默认值：NO_PRIORITY
- 可选值：NO_PRIORITY, LOW_PRIORITY, HIGH_PRIORITY, DELAYED。

7.1.4.16 distinct-agg-push-down

- 设置优化器是否执行将带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。
- 默认值：false
- 该变量作为系统变量tidb_opt_distinct_agg_push_down 的初始值。

7.1.4.17 enforce-mpp

- 用于控制是否忽略优化器代价估算，强制使用 TiFlash 的 MPP 模式执行查询。
- 默认值：false
- 该配置项可以控制系统变量tidb_enforce_mpp 的初始值。例如，当设置该配置项为 true 时，tidb_enforce_mpp 的默认值为 ON。

7.1.4.18 stats-load-concurrency 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载功能目前为实验性特性，不建议在生产环境中使用。

- TiDB 统计信息同步加载功能可以并发处理的最大列数
- 默认值：5
- 目前的合法值范围：[1, 128]

7.1.4.19 stats-load-queue-size 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载功能目前为实验性特性，不建议在生产环境中使用。

- 用于设置 TiDB 统计信息同步加载功能最多可以缓存多少列的请求
- 默认值：1000
- 目前的合法值范围：[1, 100000]

7.1.4.20 enable-stats-cache-mem-quota 从 v6.1.0 版本开始引入

> **警告：**
>
> 该变量为实验特性，不推荐在生产环境中使用。

- 用于控制 TiDB 是否开启统计信息缓存的内存上限。
- 默认值：false

7.1.5 opentracing

opentracing 的相关的设置。

7.1.5.1 enable

- 开启 opentracing 跟踪 TiDB 部分组件的调用开销。注意开启后会有一定的性能损失。
- 默认值：false

7.1.5.2 rpc-metrics

- 开启 rpc metrics。
- 默认值：false

7.1.6 opentracing.sampler

opentracing.sampler 相关的设置。

7.1.6.1 type

- opentracing 采样器的类型。
- 默认值：“const”
- 可选值：“const”，“probabilistic”，“rateLimiting”，remote”

7.1.6.2 param

- 采样器参数。
  - 对于 const 类型，可选值为 0 或 1，表示是否开启。
  - 对于 probabilistic 类型，参数为采样概率，可选值为 0 到 1 之间的浮点数。
  - 对于 rateLimiting 类型，参数为每秒采样 span 的个数。
  - 对于 remote 类型，参数为采样概率，可选值为 0 到 1 之间的浮点数。
- 默认值：1.0

7.1.6.3 sampling-server-url

- jaeger-agent 采样服务器的 HTTP URL 地址。
- 默认值：“”

7.1.6.4 max-operations

- 采样器可追踪的最大操作数。如果一个操作没有被追踪，会启用默认的 probabilistic 采样器。
- 默认值：0

7.1.6.5 sampling-refresh-interval

- 控制远程轮询 jaeger-agent 采样策略的频率。
- 默认值：0

7.1.7 opentracing.reporter

opentracing.reporter 相关的设置。

7.1.7.1 queue-size

- reporter 在内存中记录 spans 个数的队列容量。
- 默认值：0

7.1.7.2 buffer-flush-interval

- reporter 缓冲区的刷新频率。
- 默认值：0

7.1.7.3 log-spans

- 是否为所有提交的 span 打印日志。
- 默认值：false

7.1.7.4 local-agent-host-port

- reporter 向 jaeger-agent 发送 span 的地址。
- 默认值：“”

7.1.8 tikv-client

7.1.8.1 grpc-connection-count

- 跟每个 TiKV 之间建立的最大连接数。
- 默认值：4

7.1.8.2 grpc-keepalive-time

- TiDB 与 TiKV 节点之间 rpc 连接 keepalive 时间间隔，如果超过该值没有网络包，grpc client 会 ping 一下 TiKV 查看是否存活。
- 默认值：10
- 单位：秒

7.1.8.3 grpc-keepalive-timeout

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

7.1.8.4 grpc-compression-type

- 控制 TiDB 向 TiKV 节点传输数据使用的压缩算法类型。默认值为 “none” 即不压缩。修改为 “gzip” 可以使用 gzip 算法压缩数据。
- 默认值：“none”
- 可选值：“none”，“gzip”

7.1.8.5 commit-timeout

- 执行事务提交时，最大的超时时间。
- 默认值：41s
- 这个值必须是大于两倍 Raft 选举的超时时间。

7.1.8.6 max-batch-size

- 批量发送 rpc 封包的最大数量，如果不为 0，将使用 BatchCommands api 发送请求到 TiKV，可以在并发度高的情况降低 rpc 的延迟，推荐不修改该值。
- 默认值：128

7.1.8.7 max-batch-wait-time

- 等待 max-batch-wait-time 纳秒批量将此期间的数据包封装成一个大包发送给 TiKV 节点，仅在 tikv-client.max-batch-size 值大于 0 时有效，不推荐修改该值。
- 默认值：0
- 单位：纳秒

7.1.8.8 batch-wait-size

- 批量向 TiKV 发送的封包最大数量，不推荐修改该值。
- 默认值：8
- 若此值为 0 表示关闭此功能。

7.1.8.9 overload-threshold

- TiKV 的负载阈值，如果超过此阈值，会收集更多的 batch 封包，来减轻 TiKV 的压力。仅在 tikv-client.max-batch-size 值大于 0 时有效，不推荐修改该值。
- 默认值：200

7.1.9 tikv-client.copr-cache 从 v4.0.0 版本开始引入

本部分介绍 Coprocessor Cache 相关的配置项。

7.1.9.1 capacity-mb

- 缓存的总数据量大小。当缓存空间满时，旧缓存条目将被逐出。值为 0.0 时表示关闭 Coprocessor Cache。
- 默认值：1000.0
- 单位：MB
- 类型：Float

7.1.10 binlog

TiDB Binlog 相关配置。

7.1.10.1 enable

- binlog 开关。
- 默认值：false

7.1.10.2 write-timeout

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

7.1.10.3 ignore-error

- 忽略写 binlog 发生的错误时处理开关，推荐不修改该值。
- 默认值：false
- 如果设置为 `true`，发生错误时，TiDB 会停止写入 binlog，并且在监控项 `tidb_server_critical_error_total` 上计数加 1；如果设置为 `false`，写入 binlog 失败，会停止整个 TiDB 的服务。

7.1.10.4 binlog-socket

- binlog 输出网络地址。
- 默认值：“”

7.1.10.5 strategy

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

7.1.11 status

TiDB 服务状态相关配置。

7.1.11.1 report-status

- 开启 HTTP API 服务的开关。
- 默认值：true

7.1.11.2 record-db-qps

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

7.1.12 pessimistic-txn

悲观事务使用方法请参考TiDB 悲观事务模式。

7.1.12.1 max-retry-count

- 悲观事务中单个语句最大重试次数，重试次数超过该限制，语句执行将会报错。
- 默认值：256

7.1.12.2 deadlock-history-capacity

- 单个 TiDB 节点的INFORMATION_SCHEMA.DEADLOCKS 表最多可记录的死锁事件个数。当表的容量已满时，如果再次发生死锁错误，最早的一次死锁错误的信息将从表中移除。
- 默认值：10
- 最小值：0
- 最大值：10000

7.1.12.3 deadlock-history-collect-retryable

- 控制INFORMATION_SCHEMA.DEADLOCKS 表中是否收集可重试的死锁错误信息。详见 DEADLOCKS 表文档的可重试的死锁错误小节。
- 默认值：false

7.1.12.4 pessimistic-auto-commit

- 用来控制开启全局悲观事务模式下 (tidb_txn_mode='pessimistic') 时，自动提交的事务使用的事务模式。默认情况下，即使开启全局悲观事务模式，自动提交事务依然使用乐观事务模式来执行。当开启该配置项后 (设置为 true)，在全局悲观事务模式下，自动提交事务将也使用悲观事务模式执行。行为与其他显式提交的悲观事务相同。
- 对于存在冲突的场景，开启本开关可以将自动提交事务纳入全局等锁管理中，从而避免死锁，改善冲突造成死锁带来的时延尖刺。
- 对于不存在冲突的场景，如果有大量自动提交事务且单个事务操作数据量较大的情况下，开启该配置项会造成性能回退。例如，自动提交的 INSERT INTO SELECT 语句。
- 默认值：false

7.1.13 experimental

experimental 部分为 TiDB 实验功能相关的配置。该部分从 v3.1.0 开始引入。

7.1.13.1 allow-expression-index 从 v4.0.0 版本开始引入

- 用于控制是否能创建表达式索引。自 v5.2.0 版本起，如果表达式中的函数是安全的，你可以直接基于该函数创建表达式索引，不需要打开该配置项。如果要创建基于其他函数的表达式索引，可以打开该配置项，但可能存在正确性问题。通过查询 `tidb_allow_function_for_expression_index` 变量可得到能直接用于创建表达式的安全函数。
- 默认值：false

7.2 TiKV 配置文件描述

TiKV 配置文件比命令行参数支持更多的选项。你可以在 etc/config-template.toml 找到默认值的配置文件，重命名为 config.toml 即可。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见TiKV 配置参数。

7.2.1 全局配置

7.2.1.1 abort-on-panic

- 设置 TiKV panic 时是否调用 `abort()` 退出进程。此选项影响 TiKV 是否允许系统生成 core dump 文件。
  - 如果此配置项值为 false，当 TiKV panic 时，TiKV 调用 `exit()` 退出进程。
  - 如果此配置项值为 true，当 TiKV panic 时，TiKV 调用 `abort()` 退出进程。此时 TiKV 允许系统在退出时生成 core dump 文件。要生成 core dump 文件，你还需要进行 core dump 相关的系统配置（比如打开 `ulimit -c` 和配置 core dump 路径，不同操作系统配置方式不同）。建议将 core dump 生成路径设置在 TiKV 数据的不同磁盘分区，避免 core dump 文件占用磁盘空间过大，造成 TiKV 磁盘空间不足。
- 默认值：false

7.2.1.2 slow-log-file

- 存储慢日志的文件。
- 如果未设置本项但设置了 `log.file.filename`，慢日志将输出至 `log.file.filename` 指定的日志文件中。
- 如果本项和 `log.file.filename` 均未设置，所有日志默认输出到 `"stderr"`。
- 如果同时设置了两项，普通日志会输出至 `log.file.filename` 指定的日志文件中，而慢日志则会输出至本配置项指定的日志文件中。
- 默认值：“”

7.2.1.3 slow-log-threshold

- 输出慢日志的阈值。处理时间超过该阈值后会输出慢日志。
- 默认值：“1s”

7.2.2 log 从 v5.4.0 版本开始引入

日志相关的配置项。

自 v5.4.0 版本起，废弃原 log 参数 `log-rotation-timespan`，并将 `log-level`、`log-format`、`log-file`、`log-rotation-size` 变更为下列参数，与 TiDB 的 log 参数保持一致。如果只设置了原参数、且把其值设为非默认值，原参数与新参数会保持兼容；如果同时设置了原参数和新参数，则会使用新参数。

7.2.2.1 `level` 从 v5.4.0 版本开始引入

- 日志等级。
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”
- 默认值：“info”

7.2.2.2 `format` 从 v5.4.0 版本开始引入

- 日志的格式。
- 可选值：“json”，“text”
- 默认值：“text”

7.2.2.3 `enable-timestamp` 从 v5.4.0 版本开始引入

- 是否开启日志中的时间戳。
- 可选值：“true”，“false”
- 默认值：“true”

7.2.3 log.file 从 v5.4.0 版本开始引入

日志文件相关的配置项。

7.2.3.1 `filename` 从 v5.4.0 版本开始引入

- log 文件。如果未设置该参数，日志会默认输出到 `"stderr"`；如果设置了该参数，log 会输出到对应的文件中。
- 默认值：“”

7.2.3.2 `max-size` 从 v5.4.0 版本开始引入

- 单个 log 文件最大大小，超过设定的参数值后，系统自动切分成多个文件。
- 默认值：300
- 最大值：4096
- 单位：MiB

7.2.3.3 max-days 从 v5.4.0 版本开始引入

- 保留 log 文件的最长天数。
  - 如果未设置本参数或把此参数设置为默认值 0，TiKV 不清理 log 文件。
  - 如果把此参数设置为非 0 的值，在 max-days 之后，TiKV 会清理过期的日志文件。
- 默认值：0

7.2.3.4 max-backups 从 v5.4.0 版本开始引入

- 可保留的 log 文件的最大数量。
  - 如果未设置本参数或把此参数设置为默认值 0，TiKV 会保存所有的 log 文件；
  - 如果把此参数设置为非 0 的值，TiKV 最多会保留 max-backups 中指定的数量的旧日志文件。比如，如果该值设置为 7，TiKV 最多会保留 7 个旧的日志文件。
- 默认值：0

7.2.4 server

服务器相关的配置项。

7.2.4.1 status-thread-pool-size

- HTTP API 服务的工作线程数量。
- 默认值：1
- 最小值：1

7.2.4.2 grpc-compression-type

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

> 注意：
>
> 取值为 gzip 时，部分 TiDB Dashboard 可能无法完成对应的压缩运算，会显示异常。调整回默认值 none 后，TiDB Dashboard 可正常显示。

7.2.4.3 grpc-concurrency

- gRPC 工作线程的数量。调整 gRPC 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：5
- 最小值：1

7.2.4.4 grpc-concurrent-stream

- 一个 gRPC 链接中最多允许的并发请求数量。
- 默认值：1024
- 最小值：1

7.2.4.5 grpc-memory-pool-quota

- gRPC 可使用的内存大小限制。
- 默认值：无限制
- 建议仅在出现内存不足 (OOM) 的情况下限制内存使用。需要注意，限制内存使用可能会导致卡顿。

7.2.4.6 grpc-raft-conn-num

- TiKV 节点之间用于 Raft 通讯的链接最大数量。
- 默认值：1
- 最小值：1

7.2.4.7 max-grpc-send-msg-len

- 设置可发送的最大 gRPC 消息长度。
- 默认值：10485760
- 单位：Bytes
- 最大值：2147483647

7.2.4.8 grpc-stream-initial-window-size

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

7.2.4.9 grpc-keepalive-time

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

7.2.4.10 grpc-keepalive-timeout

- 关闭 gRPC 链接的超时时长。
- 默认值：3s
- 最小值：1s

7.2.4.11 concurrent-send-snap-limit

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

7.2.4.12 concurrent-recv-snap-limit

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

7.2.4.13 end-point-recursion-limit

- endpoint 下推查询请求解码消息时，最多允许的递归层数。
- 默认值：1000
- 最小值：1

7.2.4.14 end-point-request-max-handle-duration

- endpoint 下推查询请求处理任务最长允许的时长。
- 默认值：60s
- 最小值：1s

7.2.4.15 snap-max-write-bytes-per-sec

- 处理 snapshot 时最大允许使用的磁盘带宽。
- 默认值：100MB
- 单位：KB|MB|GB
- 最小值：1KB

7.2.4.16 end-point-slow-log-threshold

- endpoint 下推查询请求输出慢日志的阈值，处理时间超过阈值后会输出慢日志。
- 默认值：1s
- 最小值：0

7.2.4.17 raft-client-queue-size

- 该配置项指定 TiKV 中发送 Raft 消息的缓冲区大小。如果存在消息发送不及时导致缓冲区满、消息被丢弃的情况，可以适当调大该配置项值以提升系统运行的稳定性。
- 默认值：8192

7.2.5 readpool.unified

统一处理读请求的线程池相关的配置项。该线程池自 4.0 版本起取代原有的 storage 和 coprocessor 线程池。

7.2.5.1 min-thread-count

- 统一处理读请求的线程池最少的线程数量。
- 默认值：1

7.2.5.2 max-thread-count

- 统一处理读请求的线程池最多的线程数量，即 UnifyReadPool 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 可调整范围：`[min-thread-count, MAX(4, CPU)]`。其中，`MAX(4, CPU)` 表示：如果 CPU 核心数量小于 4，取 4；如果 CPU 核心数量大于 4，则取 CPU 核心数量。
- 默认值：MAX(4, CPU * 0.8)

7.2.5.3 stack-size

- 统一处理读请求的线程池中线程的栈大小。
- 类型：整数 + 单位
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

7.2.5.4 max-tasks-per-worker

- 统一处理读请求的线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.6 readpool.storage

存储线程池相关的配置项。

7.2.6.1 use-unified-pool

- 是否使用统一的读取线程池（在`readpool.unified` 中配置）处理存储请求。该选项值为 false 时，使用单独的存储线程池。通过本节 (`readpool.storage`) 中的其余配置项配置单独的线程池。
- 默认值：如果本节 (`readpool.storage`) 中没有其他配置，默认为 true。否则，为了升级兼容性，默认为 false，请根据需要更改`readpool.unified` 中的配置后再启用该选项。

7.2.6.2 high-concurrency

- 处理高优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

7.2.6.3 normal-concurrency

- 处理普通优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

7.2.6.4 low-concurrency

- 处理低优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

7.2.6.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.6.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.6.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.6.8 stack-size

- Storage 读线程池中线程的栈大小。
- 类型：整数 + 单位
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

7.2.7 readpool.coprocessor

协处理器线程池相关的配置项。

7.2.7.1 use-unified-pool

- 是否使用统一的读取线程池（在`readpool.unified` 中配置）处理协处理器请求。该选项值为 false 时，使用单独的协处理器线程池。通过本节 (`readpool.coprocessor`) 中的其余配置项配置单独的线程池。
- 默认值：如果本节 (`readpool.coprocessor`) 中没有其他配置，默认为 true。否则，为了升级兼容性，默认为 false，请根据需要更改`readpool.unified` 中的配置后再启用该选项。

7.2.7.2 high-concurrency

- 处理高优先级 Coprocessor 请求（如点查）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

7.2.7.3 normal-concurrency

- 处理普通优先级 Coprocessor 请求的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

7.2.7.4 low-concurrency

- 处理低优先级 Coprocessor 请求（如扫表）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

7.2.7.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.7.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.7.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

7.2.7.8 stack-size

- Coprocessor 线程池中线程的栈大小。
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

7.2.8 storage

存储相关的配置项。

7.2.8.1 scheduler-concurrency

- scheduler 内置一个内存锁机制，防止同时对一个 key 进行操作。每个 key hash 到不同的槽。
- 默认值：524288
- 最小值：1

7.2.8.2 scheduler-worker-pool-size

- Scheduler 线程池中线程的数量。Scheduler 线程主要负责写入之前的事务一致性检查工作。如果 CPU 核心数量大于等于 16，默认为 8；否则默认为 4。调整 scheduler 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：4
- 可调整范围：`[1, MAX(4, CPU)]`。其中，`MAX(4, CPU)` 表示：如果 CPU 核心数量小于 4，取 4；如果 CPU 核心数量大于 4，则取 CPU 核心数量。

7.2.8.3 scheduler-pending-write-threshold

- 写入数据队列的最大值，超过该值之后对于新的写入 TiKV 会返回 Server Is Busy 错误。
- 默认值：100MB
- 单位：MB|GB

7.2.8.4 reserve-space

- TiKV 启动时会预留一块空间用于保护磁盘空间。当磁盘剩余空间小于该预留空间时，TiKV 会限制部分写操作。预留空间形式上分为两个部分：预留空间的 80% 用作磁盘空间不足时的运维操作所需要的额外磁盘空间，剩余的 20% 为磁盘临时文件。在回收空间的过程中，如果额外使用的磁盘空间过多，导致存储耗尽时，该临时文件会成为恢复服务的最后一道防御。
- 临时文件名为 space_placeholder_file，位于 storage.data-dir 目录下。当 TiKV 因磁盘空间耗尽而下线时，重启 TiKV 会自动删除该临时文件，并自动尝试回收空间。
- 当剩余空间不足时，TiKV 不会创建该临时文件。防御的有效性与预留空间的大小有关。预留空间大小的计算方式为磁盘容量的 5% 与该配置项之间的最大值。当该配置项的值为 0MB 时，TiKV 会关闭磁盘防护功能。
- 默认值：5GB
- 单位：MB|GB

7.2.8.5 enable-ttl

> 警告：
>
> - 你只能在部署新的 TiKV 集群时将 enable-ttl 的值设置为 true 或 false，不能在已有的 TiKV 集群中修改该配置项的值。由于该配置项为 true 和 false 的 TiKV 集群所存储的数据格式不相同，如果你在已有的 TiKV 集群中修改该配置项的值，会造成不同格式的数据存储在同一个集群，导致重启对应的 TiKV 集群时 TiKV 报“can’t enable ttl on a non-ttl instance”错误。
> - 你只能在 TiKV 集群中使用 enable-ttl，不能在有 TiDB 节点的集群中使用该配置项（即在此类集群中把 enable-ttl 设置为 true），否则会导致数据损坏、TiDB 集群升级失败等严重后果。

- TTL 即 Time to live。数据超过 TTL 时间后会被自动删除。用户需在客户端写入请求中指定 TTL。不指定 TTL 即表明相应数据不会被自动删除。
- 默认值：false

7.2.8.6 ttl-check-poll-interval

- 回收数据物理空间的检查周期。如果数据超过了 TTL 时间，数据的物理空间会在检查时被强制回收。
- 默认值：12h
- 最小值：0s

7.2.8.7 background-error-recovery-window 从 v6.1.0 版本开始引入

- RocksDB 检测到可恢复的后台错误后，所允许的最长恢复时间。如果后台 SST 文件出现损坏，RocksDB 在检测到故障 SST 文件所属的 Peer 后，会通过心跳上报到 PD。PD 随后会进行调度操作移除该 Peer。最后故障 SST 文件将会被直接删除，随后 TiKV 后台恢复正常。

- 在恢复操作完成之前，损坏的 SST 文件将一直存在。此时 RocksDB 可以继续写入新的内容，但读到损坏的数据范围时会返回错误。
- 如果恢复操作未能在该时间窗口内完成，TiKV 会崩溃。
- 默认值：1h

7.2.8.8 api-version 从 v6.1.0 版本开始引入

- TiKV 作为 Raw Key Value 存储数据时使用的存储格式与接口版本。
- 可选值：
  - 1：使用 API V1。不对客户端传入的数据进行编码，而是原样存储。在 v6.1.0 之前的版本，TiKV 都使用 API V1。
  - 2：使用 API V2：
    * 数据采用 MVCC（Multi Version Concurrency Control）方式存储，其中时间戳由 tikv-server 从 PD 获取（即 TSO）。
    * 需要同时设置 `storage.enable-ttl = true`。由于 API V2 支持 TTL 特性，因此强制要求打开 `enable-ttl` 以避免这个参数出现歧义。
    * 启用 API V2 后需要在集群中额外部署至少一个 tidb-server 以回收过期数据。注意该 tidb-server 不可提供读写服务。可以部署多个 tidb-server 以保证高可用。
    * 需要客户端的支持。请参考对应客户端的 API V2 使用说明。
- 默认值：1

> **警告：**
>
> - API V2 是 TiKV 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。
> - 只能在部署新的 TiKV 集群时将 `api-version` 的值设置为 2，不能在已有的 TiKV 集群中修改该配置项的值。由于 API V1 和 API V2 存储的数据格式不相同，如果在已有的 TiKV 集群中修改该配置项，会造成不同格式的数据存储在同一个集群，导致数据损坏。这种情况下，启动 TiKV 集群时会报“unable to switch storage.api_version”错误。
> - 启用 API V2 后，不能将 TiKV 集群回退到 v6.1.0 之前的版本，否则可能导致数据损坏。

7.2.9 storage.block-cache

RocksDB 多个 CF 之间共享 block cache 的配置选项。当开启时，为每个 CF 单独配置的 block cache 将无效。

7.2.9.1 shared

- 是否开启共享 block cache。
- 默认值：true

7.2.9.2 capacity

- 共享 block cache 的大小。
- 默认值：系统总内存大小的 45%
- 单位：KB|MB|GB

7.2.10 storage.flow-control

在 scheduler 层进行流量控制代替 RocksDB 的 write stall 机制，可以避免 write stall 机制卡住 Raftstore 或 Apply 线程导致的次生问题。本节介绍 TiKV 流量控制机制相关的配置项。

7.2.10.1 enable

- 是否开启流量控制机制。开启后，TiKV 会自动关闭 KvDB 的 write stall 机制，还会关闭 RaftDB 中除 memtable 以外的 write stall 机制。
- 默认值：true

7.2.10.2 memtables-threshold

- 当 KvDB 的 memtable 的个数达到该阈值时，流控机制开始工作。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).max-write-buffer-number 的配置。
- 默认值：5

7.2.10.3 l0-files-threshold

- 当 KvDB 的 L0 文件个数达到该阈值时，流控机制开始工作。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).level0-slowdown-writes-trigger的配置。
- 默认值：20

7.2.10.4 soft-pending-compaction-bytes-limit

- 当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制开始拒绝部分写入请求，报错 ServerIsBusy。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).soft-pending-compaction-bytes-limit 的配置。
- 默认值：“192GB”

7.2.10.5 hard-pending-compaction-bytes-limit

- 当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制拒绝所有写入请求，报错 ServerIsBusy。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).hard-pending-compaction-bytes-limit 的配置。
- 默认值：“1024GB”

7.2.11 storage.io-rate-limit

I/O rate limiter 相关的配置项。

7.2.11.1 max-bytes-per-sec

- 限制服务器每秒从磁盘读取数据或写入数据的最大 I/O 字节数，I/O 类型由下面的 mode 配置项决定。达到该限制后，TiKV 倾向于放缓后台操作为前台操作节流。该配置项值应设为磁盘的最佳 I/O 带宽，例如云盘厂商指定的最大 I/O 带宽。
- 默认值：“0MB”

7.2.11.2 mode

- 确定哪些类型的 I/O 操作被计数并受 max-bytes-per-sec 阈值的限流。当前 TiKV 只支持 write-only 只写模式。
- 可选值：write-only
- 默认值：write-only

7.2.12 raftstore

raftstore 相关的配置项。

7.2.12.1 prevote

- 开启 Prevote 的开关，开启有助于减少隔离恢复后对系统造成的抖动。
- 默认值：true

7.2.12.2 capacity

- 存储容量，即允许的最大数据存储大小。如果没有设置，则使用当前磁盘容量。如果要将多个 TiKV 实例部署在同一块物理磁盘上，需要在 TiKV 配置中添加该参数，参见混合部署的关键参数介绍。
- 默认值：0

7.2.12.3 raftdb-path

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

7.2.12.4 raft-base-tick-interval

> **注意：**
>
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 状态机 tick 一次的间隔时间。
- 默认值：1s
- 最小值：大于 0

7.2.12.5 raft-heartbeat-ticks

> **注意：**
>
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发送心跳时经过的 tick 个数，即每隔 raft-base-tick-interval * raft-heartbeat-ticks 时间发送一次心跳。
- 默认值：2
- 最小值：大于 0

7.2.12.6 raft-election-timeout-ticks

> **注意：**
>
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发起选举时经过的 tick 个数，即如果处于无主状态，大约经过 raft-base-tick-interval * raft-election-timeout-ticks 时间以后发起选举。
- 默认值：10
- 最小值：raft-heartbeat-ticks

7.2.12.7 raft-min-election-timeout-ticks

> **注意：**
>
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发起选举时至少经过的 tick 个数，如果为 0，则表示使用 raft-election-timeout-ticks，不能比 raft-election-timeout-ticks 小。
- 默认值：0
- 最小值：0

7.2.12.8 raft-max-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发起选举时最多经过的 tick 个数，如果为 0，则表示使用 raft-election-timeout-ticks * 2。
- 默认值：0
- 最小值：0

7.2.12.9 raft-max-size-per-msg

- 产生的单个消息包的大小限制，软限制。
- 默认值：1MB
- 最小值：大于 0
- 最大值: 3GB
- 单位：KB|MB|GB

7.2.12.10 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。
- 默认值：256
- 最小值：大于 0
- 最大值: 16384

7.2.12.11 raft-entry-max-size

- 单个日志最大大小，硬限制。
- 默认值：8MB
- 最小值：0
- 单位：MB|GB

7.2.12.12 raft-log-compact-sync-interval 从 v5.3 版本开始引入

- 压缩非必要 Raft 日志的时间间隔
- 默认值：“2s”
- 最小值：“0s”

7.2.12.13 raft-log-gc-tick-interval

- 删除 Raft 日志的轮询任务调度间隔时间，0 表示不启用。
- 默认值：“3s”
- 最小值：“0s”

7.2.12.14 raft-log-gc-threshold

- 允许残余的 Raft 日志个数，这是一个软限制。
- 默认值：50
- 最小值：1

7.2.12.15 raft-log-gc-count-limit

- 允许残余的 Raft 日志个数，这是一个硬限制。默认值为按照每个日志 1MB 而计算出来的 3/4 region 大小所能容纳的日志个数。
- 最小值：0

7.2.12.16 raft-log-gc-size-limit

- 允许残余的 Raft 日志大小，这是一个硬限制，默认为 region 大小的 3/4。
- 最小值：大于 0

7.2.12.17 raft-log-reserve-max-ticks 从 v5.3 版本开始引入

- 超过本配置项设置的的 tick 数后，即使剩余 Raft 日志的数量没有达到 raft-log-gc-threshold 设置的值，TiKV 也会进行 GC 操作。
- 默认值：6
- 最小值：大于 0

7.2.12.18 raft-entry-cache-life-time

- 内存中日志 cache 允许的最长残留时间。
- 默认值：30s
- 最小值：0

7.2.12.19 hibernate-regions

- 打开或关闭静默 Region。打开后，如果 Region 长时间处于非活跃状态，即被自动设置为静默状态。静默状态的 Region 可以降低 Leader 和 Follower 之间心跳信息的系统开销。可以通过 peer-stale-state-check-interval 调整 Leader 和 Follower 之间的心跳间隔。
- 默认值：v5.0.2 及以后版本默认值为 true，v5.0.2 以前的版本默认值为 false

7.2.12.20 split-region-check-tick-interval

- 检查 region 是否需要分裂的时间间隔，0 表示不启用。
- 默认值：10s
- 最小值：0

7.2.12.21 region-split-check-diff

- 允许 region 数据超过指定大小的最大值，默认为 region 大小的 1/16。
- 最小值：0

7.2.12.22 region-compact-check-interval

- 检查是否需要人工触发 rocksdb compaction 的时间间隔，0 表示不启用。
- 默认值：5m
- 最小值：0

7.2.12.23 region-compact-check-step

- 每轮校验人工 compaction 时，一次性检查的 region 个数。
- 默认值：100
- 最小值：0

7.2.12.24 region-compact-min-tombstones

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

7.2.12.25 region-compact-tombstones-percent

- 触发 rocksdb compaction 需要的 tombstone 所占比例。
- 默认值：30
- 最小值：1
- 最大值：100

7.2.12.26 pd-heartbeat-tick-interval

- 触发 region 对 PD 心跳的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

7.2.12.27 pd-store-heartbeat-tick-interval

- 触发 store 对 PD 心跳的时间间隔，0 表示不启用。
- 默认值：10s
- 最小值：0

7.2.12.28 snap-mgr-gc-tick-interval

- 触发回收过期 snapshot 文件的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

7.2.12.29 snap-gc-timeout

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

7.2.12.30 snap-generator-pool-size 从 v5.4.0 版本开始引入

- 用于配置 snap-generator 线程池的大小。
- 为了让 TiKV 在恢复场景下加快 Region 生成 Snapshot 的速度，需要调大对应 Worker 的 snap-generator 线程数量。可通过本配置项调大对应线程的数量。
- 默认值：2
- 最小值：1

7.2.12.31 lock-cf-compact-interval

- 触发对 lock CF compact 检查的时间间隔。
- 默认值：10m
- 最小值：0

7.2.12.32 lock-cf-compact-bytes-threshold

- 触发对 lock CF 进行 compact 的大小。
- 默认值：256MB
- 最小值：0
- 单位：MB

7.2.12.33 notify-capacity

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

7.2.12.34 messages-per-tick

- 每轮处理的消息最大个数。
- 默认值：4096
- 最小值：0

7.2.12.35 max-peer-down-duration

- 副本允许的最长未响应时间，超过将被标记为 down，后续 PD 会尝试将其删掉。
- 默认值：10m
- 最小值：当 Hibernate Region 功能启用时，为 peer-stale-state-check-interval * 2；Hibernate Region 功能关闭时，为 0。

7.2.12.36 max-leader-missing-duration

- 允许副本处于无主状态的最长时间，超过将会向 PD 校验自己是否已经被删除。
- 默认值：2h
- 最小值：> abnormal-leader-missing-duration

7.2.12.37 abnormal-leader-missing-duration

- 允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中。
- 默认值：10m
- 最小值：> peer-stale-state-check-interval

7.2.12.38 peer-stale-state-check-interval

- 触发检验副本是否处于无主状态的时间间隔。
- 默认值：5m
- 最小值：> 2 * election-timeout

7.2.12.39 leader-transfer-max-log-lag

- 尝试转移领导权时被转移者允许的最大日志缺失个数。
- 默认值：128
- 最小值：10

7.2.12.40 max-snapshot-file-raw-size 从 v6.1.0 版本开始引入

- 当 snapshot 文件大于该配置项指定的大小时，snapshot 文件会被切割为多个文件。
- 默认值：100MiB
- 最小值：100MiB

7.2.12.41 snap-apply-batch-size

- 当导入 snapshot 文件需要写数据时，内存写缓存的大小
- 默认值：10MB
- 最小值：0
- 单位：MB

7.2.12.42 consistency-check-interval

> **警告：**
>
> 开启一致性检查对集群性能有影响，并且和 TiDB GC 操作不兼容，不建议在生产环境中使用。

- 触发一致性检查的时间间隔，0 表示不启用。
- 默认值：0s
- 最小值：0

7.2.12.43 raft-store-max-leader-lease

- region 主可信任期的最长时间。
- 默认值：9s
- 最小值：0

7.2.12.44 right-derive-when-split

- 为 true 时，以最大分裂 key 为起点的 region 复用原 region 的 key；否则以原 region 起点 key 作为起点的 region 复用原 region 的 key。
- 默认值：true

7.2.12.45 merge-max-log-gap

- 进行 merge 时，允许的最大日志缺失个数。
- 默认值：10
- 最小值：> raft-log-gc-count-limit

7.2.12.46 merge-check-tick-interval

- 触发 merge 完成检查的时间间隔。
- 默认值：2s
- 最小值：大于 0

7.2.12.47 use-delete-range

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

7.2.12.48 cleanup-import-sst-interval

- 触发检查过期 SST 文件的时间间隔，0 表示不启用。
- 默认值：10m
- 最小值：0

7.2.12.49 local-read-batch-size

- 一轮处理读请求的最大个数。
- 默认值：1024
- 最小值：大于 0

7.2.12.50 apply-max-batch-size

- Raft 状态机由 BatchSystem 批量执行数据写入请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。
- 默认值：256
- 最小值：大于 0
- 最大值: 10240

7.2.12.51 apply-pool-size

- Apply 线程池负责把数据落盘至磁盘。该配置项为 Apply 线程池中线程的数量，即 Apply 线程池的大小。调整 Apply 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：2
- 可调整范围：[1, CPU * 10]

7.2.12.52 store-max-batch-size

- Raft 状态机由 BatchSystem 批量执行把日志落盘至磁盘的请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。
- 如果开启 hibernate-regions，默认值为 256；如果关闭 hibernate-regions，默认值为 1024
- 最小值：大于 0
- 最大值: 10240

7.2.12.53 store-pool-size

- 表示处理 Raft 的线程池中线程的数量，即 Raftstore 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 默认值：2
- 可调整范围：[1, CPU * 10]

7.2.12.54 store-io-pool-size 从 v5.3.0 版本开始引入

- 表示处理 Raft I/O 任务的线程池中线程的数量，即 StoreWriter 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 默认值：0
- 最小值：0

7.2.12.55 future-poll-size

- 驱动 future 的线程池中线程的数量。
- 默认值：1
- 最小值：大于 0

7.2.12.56 cmd-batch

- 对请求进行攒批的控制开关，开启后可显著提升写入性能。
- 默认值：true

7.2.12.57 inspect-interval

- TiKV 每隔一段时间会检测 Raftstore 组件的延迟情况，该配置项设置检测的时间间隔。当检测的延迟超过该时间，该检测会被记为超时。
- 根据超时的检测延迟的比例计算判断 TiKV 是否为慢节点。
- 默认值：500ms
- 最小值：1ms

7.2.12.58 raft-write-size-limit 从 v5.3.0 版本开始引入

- 触发 Raft 数据写入的阈值。当数据大小超过该配置项值，数据会被写入磁盘。当 store-io-pool-size 的值为 0 时，该配置项不生效。
- 默认值：1MB
- 最小值：0

7.2.13 coprocessor

coprocessor 相关的配置项。

7.2.13.1 split-region-on-table

- 开启按 table 分裂 Region 的开关，建议仅在 TiDB 模式下使用。
- 默认值：false

7.2.13.2 batch-split-limit

- 批量分裂 Region 的阈值，调大该值可加速分裂 Region。
- 默认值：10
- 最小值：1

7.2.13.3 region-max-size

- Region 容量空间最大值，超过时系统分裂成多个 Region。
- 默认值：`region-split-size / 2 * 3`
- 单位：KiB|MiB|GiB

7.2.13.4 region-split-size

- 分裂后新 Region 的大小，此值属于估算值。
- 默认值：96MiB
- 单位：KiB|MiB|GiB

7.2.13.5 region-max-keys

- Region 最多允许的 key 的个数，超过时系统分裂成多个 Region。
- 默认值：`region-split-keys / 2 * 3`

7.2.13.6 region-split-keys

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

7.2.13.7 enable-region-bucket 从 v6.1.0 版本开始引入

- 是否将 Region 划分为更小的区间 bucket，并且以 bucket 作为并发查询单位，以提高扫描数据的并发度。bucket 的详细设计可见 Dynamic size Region。
- 默认值：false

> **警告：**

- enable-region-bucket 是 TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。
- 这个参数仅在 region-split-size 调到两倍 region-bucket-size 及以上时才有意义，否则不会真正生成 bucket。
- 将 region-split-size 调大可能会有潜在的性能回退、数据调度缓慢的风险。

7.2.13.8 region-bucket-size 从 v6.1.0 版本开始引入

- 设置 enable-region-bucket 启用时 bucket 的预期大小。
- 默认值：96MiB

> **警告：**
>
> region-bucket-size 是 TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。

7.2.14 rocksdb

rocksdb 相关的配置项。

7.2.14.1 max-background-jobs

- RocksDB 后台线程个数。调整 RocksDB 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：
  - CPU 核数为 10 时，默认值为 9
  - CPU 核数为 8 时，默认值为 7
  - CPU 核数为 N 时，默认值为 max(2, min(N - 1, 9))
- 最小值：2

7.2.14.2 max-background-flushes

- RocksDB 用于刷写 memtable 的最大后台线程数量。
- 默认值：
  - CPU 核数为 10 时，默认值为 3
  - CPU 核数为 8 时，默认值为 2
  - CPU 核数为 N 时，默认值为 [(max-background-jobs + 3)/ 4]
- 最小值：1

7.2.14.3 max-sub-compactions

- RocksDB 进行 subcompaction 的并发个数。
- 默认值：3
- 最小值：1

7.2.14.4 max-open-files

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

7.2.14.5 max-manifest-file-size

- RocksDB Manifest 文件最大大小。
- 默认值：128MB
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.6 create-if-missing

- 自动创建 DB 开关。
- 默认值：true

7.2.14.7 wal-recovery-mode

- 预写式日志 (WAL, Write Ahead Log) 的恢复模式。
- 可选值：
  - "tolerate-corrupted-tail-records"：容忍并丢弃位于日志尾部的不完整的数据 (trailing data)。
  - "absolute-consistency"：当发现待恢复的日志中有被损坏的日志时，放弃恢复所有日志。
  - "point-in-time"：按顺序恢复日志。遇到第一个损坏的日志时，停止恢复剩余的日志。
  - "skip-any-corrupted-records"：灾难后恢复。跳过日志中的损坏记录，尽可能多地恢复数据。
- 默认值："point-in-time"

7.2.14.8 wal-dir

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

7.2.14.9 wal-ttl-seconds

- 归档 WAL 生存周期，超过该值时，系统会删除相关 WAL。
- 默认值：0
- 最小值：0
- 单位：秒

7.2.14.10 wal-size-limit

- 归档 WAL 大小限制，超过该值时，系统会删除相关 WAL。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.11 enable-statistics

- 开启 RocksDB 的统计信息。
- 默认值：true

7.2.14.12 stats-dump-period

- 将统计信息输出到日志中的间隔时间。
- 默认值：10m

7.2.14.13 compaction-readahead-size

- 开启 RocksDB compaction 过程中的预读功能，该项指定预读数据的大小。如果使用的是机械磁盘，建议该值至少为 2MB。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.14 writable-file-max-buffer-size

- WritableFileWrite 所使用的最大的 buffer 大小。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

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

- 决定后台 flush 或者 compaction 的读写是否设置 O_DIRECT 的标志。该选项对性能的影响：开启 O_DIRECT 可以绕过并防止污染操作系统 buffer cache，但后续文件读取需要把内容重新读到 buffer cache。
- 默认值：false

7.2.14.16 rate-bytes-per-sec

- RocksDB compaction rate limiter 的限制速率。
- 默认值：10GB
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.17 rate-limiter-mode

- RocksDB 的 compaction rate limiter 模式。
- 可选值：“read-only”，“write-only”，“all-io”
- 默认值：“write-only”

7.2.14.18 rate-limiter-auto-tuned 从 v5.0 版本开始引入

- 控制是否依据最近的负载量自动优化 RocksDB 的 compaction rate limiter 配置。此配置项开启后，compaction pending bytes 监控指标值会比一般情况下稍微高些。
- 默认值：true

7.2.14.19 enable-pipelined-write

- 控制是否开启 Pipelined Write。开启时会使用旧的 Pipelined Write，关闭时会使用新的 Pipelined Commit 机制。
- 默认值：false

7.2.14.20 bytes-per-sync

- 异步 Sync 限速速率。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.21 wal-bytes-per-sync

- WAL Sync 限速速率，默认：512KB。
- 默认值：512KB
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.22 info-log-max-size

- Info 日志的最大大小。
- 默认值：1GB
- 最小值：0
- 单位：B|KB|MB|GB

7.2.14.23 info-log-roll-time

- 日志截断间隔时间，如果为 0s 则不截断。
- 默认值：0s

7.2.14.24 info-log-keep-log-file-num

- 保留日志文件最大个数。
- 默认值：10
- 最小值：0

7.2.14.25 info-log-dir

- 日志存储目录。
- 默认值：“”

7.2.15 rocksdb.titan

Titan 相关的配置项。

7.2.15.1 enabled

- 开启 Titan 开关。
- 默认值：false

7.2.15.2 dirname

- Titan Blob 文件存储目录。
- 默认值：titandb

7.2.15.3 disable-gc

- 关闭 Titan 对 Blob 文件的 GC 的开关。
- 默认值：false

7.2.15.4 max-background-gc

- Titan 后台 GC 的线程个数。
- 默认值：4
- 最小值：1

7.2.16 rocksdb.defaultcf | rocksdb.writecf | rocksdb.lockcf

rocksdb defaultcf、 rocksdb writecf 和 rocksdb lockcf 相关的配置项。

7.2.16.1 block-size

- 一个 RocksDB block 的默认大小。
- defaultcf 默认值：64KB
- writecf 默认值：64KB
- lockcf 默认值：16KB
- 最小值：1KB
- 单位：KB|MB|GB

7.2.16.2 block-cache-size

- 一个 RocksDB block 的默认缓存大小。
- defaultcf 默认值：机器总内存 * 25%
- writecf 默认值：机器总内存 * 15%
- lockcf 默认值：机器总内存 * 2%
- 最小值：0
- 单位：KB|MB|GB

7.2.16.3 disable-block-cache

- 开启 block cache 开关。
- 默认值：false

7.2.16.4 cache-index-and-filter-blocks

- 开启缓存 index 和 filter 的开关。
- 默认值：true

7.2.16.5 pin-l0-filter-and-index-blocks

- 控制第 0 层 SST 文件的 index block 和 filter block 是否常驻在内存中的开关。
- 默认值：true

7.2.16.6 use-bloom-filter

- 开启 bloom filter 的开关。
- 默认值：true

7.2.16.7 optimize-filters-for-hits

- 开启优化 filter 的命中率的开关。
- defaultcf 默认值：true
- writecf 默认值：false
- lockcf 默认值：false

7.2.16.8 whole-key-filtering

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

7.2.16.9 bloom-filter-bits-per-key

bloom filter 为每个 key 预留的长度。

- 默认值：10
- 单位：字节

7.2.16.10 block-based-bloom-filter

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

7.2.16.11 read-amp-bytes-per-bit

- 开启读放大统计的开关，0：不开启，> 0 开启。
- 默认值：0
- 最小值：0

7.2.16.12 compression-per-level

- 每一层默认压缩算法。
- defaultcf 的默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]
- writecf 的默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]
- lockcf 的默认值：[“no”，“no”，“no”，“no”，“no”，“no”，“no”]

7.2.16.13 bottommost-level-compression

- 设置最底层的压缩算法。该设置将覆盖 compression-per-level 的设置。
- 因为最底层并非从数据开始写入 LSM-tree 起就直接采用 compression-per-level 数组中的最后一个压缩算法，使用 bottommost-level-compression 可以让最底层从一开始就使用压缩效果最好的压缩算法。
- 如果不想设置最底层的压缩算法，可以将该配置项的值设为 disable。
- 默认值：“zstd”

7.2.16.14 write-buffer-size

- memtable 大小。
- defaultcf 默认值：`"128MB"`
- writecf 默认值：`"128MB"`
- lockcf 默认值：`"32MB"`
- 最小值：0
- 单位：KB|MB|GB

7.2.16.15 max-write-buffer-number

- 最大 memtable 个数。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.memtables-threshold` 会覆盖此配置。
- 默认值：5
- 最小值：0

7.2.16.16 min-write-buffer-number-to-merge

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

7.2.16.17 max-bytes-for-level-base

- base level (L1) 最大字节数，一般设置为 memtable 大小 4 倍。
- defaultcf 默认值：`"512MB"`
- writecf 默认值：`"512MB"`
- lockcf 默认值：`"128MB"`
- 最小值：0
- 单位：KB|MB|GB

7.2.16.18 target-file-size-base

- base level 的目标文件大小。当 `enable-compaction-guard` 的值为 `true` 时，`compaction-guard-max-output-file-size` 会覆盖此配置。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

7.2.16.19 level0-file-num-compaction-trigger

- 触发 compaction 的 L0 文件最大个数。
- defaultcf 默认值：4
- writecf 默认值：4
- lockcf 默认值：1
- 最小值：0

7.2.16.20 level0-slowdown-writes-trigger

- 触发 write stall 的 L0 文件最大个数。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.l0-files-threshold` 会覆盖此配置。
- 默认值：20
- 最小值：0

7.2.16.21 level0-stop-writes-trigger

- 完全阻停写入的 L0 文件最大个数。
- 默认值：36
- 最小值：0

7.2.16.22 max-compaction-bytes

- 一次 compaction 最大写入字节数，默认 2GB。
- 默认值：2GB
- 最小值：0
- 单位：KB|MB|GB

7.2.16.23 compaction-pri

- 优先处理 compaction 的类型
- 可选值：
  - `"by-compensated-size"`：根据大小顺序，优先对大文件进行 compaction。
  - `"oldest-largest-seq-first"`：根据时间顺序，优先对数据更新时间晚的文件进行 compaction。当你只在小范围内更新部分热点键 (hot keys) 时，可以使用此配置。
  - `"oldest-smallest-seq-first"`：根据时间顺序，优先对长时间没有被 compact 到下一级的文件进行 compaction。如果你在大范围内随机更新了部分热点键，使用该配置可以轻微缓解写放大。
  - `"min-overlapping-ratio"`：根据重叠比例，优先对在不同层之间文件重叠比例高的文件进行 compaction，即一个文件在 下一层的大小/本层的大小 的值越小，compaction 的优先级越高。在诸多场景下，该配置可以有效缓解写放大。
- 默认值：
  - `defaultcf` 和 `writecf` 的默认值：`"min-overlapping-ratio"`
  - `lockcf` 的默认值：`"by-compensated-size"`

7.2.16.24 dynamic-level-bytes

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

7.2.16.25 num-levels

- RocksDB 文件最大层数。
- 默认值：7

7.2.16.26 max-bytes-for-level-multiplier

- 每一层的默认放大倍数。
- 默认值：10

7.2.16.27 compaction-style

- compaction 方法。
- 可选值：“level”，“universal”，“fifo”
- 默认值：“level”

7.2.16.28 disable-auto-compactions

- 是否关闭自动 compaction。
- 默认值：false

7.2.16.29 soft-pending-compaction-bytes-limit

- pending compaction bytes 的软限制。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.soft-pending-compaction-bytes-limit` 会覆盖此配置。
- 默认值：192GB
- 单位：KB|MB|GB

7.2.16.30 hard-pending-compaction-bytes-limit

- pending compaction bytes 的硬限制。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.hard-pending-compaction-bytes-limit` 会覆盖此配置。
- 默认值：256GB
- 单位：KB|MB|GB

7.2.16.31 enable-compaction-guard

- 设置 compaction guard 的启用状态。compaction guard 优化通过使用 TiKV Region 边界分割 SST 文件，帮助降低 compaction I/O，让 TiKV 能够输出较大的 SST 文件，并且在迁移 Region 时及时清理过期数据。
- `defaultcf` 默认值：`true`
- `writecf` 默认值：`true`
- `lockcf` 默认值：`false`

7.2.16.32 compaction-guard-min-output-file-size

- 设置 compaction guard 启用时 SST 文件大小的最小值，防止 SST 文件过小。
- 默认值：8MB
- 单位：KB|MB|GB

7.2.16.33 compaction-guard-max-output-file-size

- 设置 compaction guard 启用时 SST 文件大小的最大值，防止 SST 文件过大。对于同一列族，此配置项的值会覆盖 target-file-size-base。
- 默认值：128MB
- 单位：KB|MB|GB

7.2.17 rocksdb.defaultcf.titan

rocksdb defaultcf titan 相关的配置项。

7.2.17.1 min-blob-size

- 最小存储在 Blob 文件中 value 大小，低于该值的 value 还是存在 LSM-Tree 中。
- 默认值：1KB
- 最小值：0
- 单位：KB|MB|GB

7.2.17.2 blob-file-compression

- Blob 文件所使用的压缩算法，可选值：no、 snappy、 zlib、 bz2、 lz4、 lz4hc、 zstd。
- 默认值：lz4

7.2.17.3 blob-cache-size

- Blob 文件的 cache 大小。
- 默认值：0GB
- 最小值：0
- 单位：KB|MB|GB

7.2.17.4 min-gc-batch-size

- 做一次 GC 所要求的最低 Blob 文件大小总和。
- 默认值：16MB
- 最小值：0
- 单位：KB|MB|GB

7.2.17.5 max-gc-batch-size

- 做一次 GC 所要求的最高 Blob 文件大小总和。
- 默认值：64MB
- 最小值：0
- 单位：KB|MB|GB

7.2.17.6 discardable-ratio

- Blob 文件 GC 的触发比例，如果某 Blob 文件中的失效 value 的比例高于该值才可能被 GC 选中。
- 默认值：0.5
- 最小值：0
- 最大值：1

7.2.17.7 sample-ratio

- 进行 GC 时，对 Blob 文件进行采样时读取数据占整个文件的比例。
- 默认值：0.1
- 最小值：0
- 最大值：1

7.2.17.8 merge-small-file-threshold

- Blob 文件的大小小于该值时，无视 discardable-ratio 仍可能被 GC 选中。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

7.2.17.9 blob-run-mode

- Titan 的运行模式选择，可选值：
  - “normal”：value size 超过 min-blob-size 的数据会写入到 blob 文件。
  - “read_only”：不再写入新数据到 blob，原有 blob 内的数据仍然可以读取。
  - “fallback”：将 blob 内的数据写回 LSM。
- 默认值：“normal”

7.2.17.10 level-merge

- 是否通过开启 level-merge 来提升读性能，副作用是写放大会比不开启更大。
- 默认值：false

7.2.17.11 gc-merge-rewrite

- 是否开启使用 merge operator 来进行 Titan GC 写回操作，减少 Titan GC 对于前台写入的影响。
- 默认值：false

7.2.18 raftdb

raftdb 相关配置项。

7.2.18.1 max-background-jobs

- RocksDB 后台线程个数。调整 RocksDB 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：4
- 最小值：2

7.2.18.2 max-sub-compactions

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

7.2.18.3 wal-dir

- WAL 存储目录。
- 默认值：/tmp/tikv/store

7.2.19 raft-engine

Raft Engine 相关的配置项。

> 注意：
>
> - 第一次开启 Raft Engine 时，TiKV 会将原有的 RocksDB 数据转移至 Raft Engine 中。因此，TiKV 的启动时间会比较长，你需要额外等待几十秒。
> - 如果你要将 TiDB 集群降级至 v5.4.0 以前的版本（不含 v5.4.0），你需要在降级之前先关闭 Raft Engine（即把 enable 配置项设置为 false，并重启 TiKV 使配置生效），否则会导致集群降级后无法正常开启。

7.2.19.1 enable

- 决定是否使用 Raft Engine 来存储 Raft 日志。开启该配置项后，raftdb 的配置不再生效
- 默认值：true

7.2.19.2 dir

- 存储 Raft 日志文件的目录。如果该目录不存在，则在启动 TiKV 时创建该目录。
- 如果未设置此配置，则使用 `{data-dir}/raft-engine`。
- 如果你的机器上有多个磁盘，建议将 Raft Engine 的数据存储在单独的磁盘上，以提高 TiKV 性能。
- 默认值：`""`

7.2.19.3 batch-compression-threshold

- 指定日志批处理的阈值大小。大于此配置的日志批次将被压缩。如果将此配置项设置为 `0`，则禁用压缩。
- 默认值：`"8KB"`

7.2.19.4 bytes-per-sync

- 指定缓存写入的最大累积大小。当超过此配置值时，缓存的写入将被刷写到磁盘。
- 如果将此配置项设置为 `0`，则禁用增量同步。
- 默认值：`"4MB"`

7.2.19.5 target-file-size

- 指定日志文件的最大大小。当日志文件大于此值时，将对其进行轮转。
- 默认值：`"128MB"`

7.2.19.6 purge-threshold

- 指定主日志队列的阈值大小。当超过此配置值时，将对主日志队列执行垃圾回收。
- 此参数可用于调整 Raft Engine 的空间占用大小。
- 默认值：`"10GB"`

7.2.19.7 recovery-mode

- 确定在日志恢复过程中如何处理文件损坏。
- 可选值：`"absolute-consistency"`, `"tolerate-tail-corruption"`, `"tolerate-any-corruption"`
- 默认值：`"tolerate-tail-corruption"`

7.2.19.8 recovery-read-block-size

- 恢复期间读取日志文件的最小 I/O 大小。
- 默认值：`"16KB"`
- 最小值：`"512B"`

7.2.19.9 recovery-threads

- 用于扫描和恢复日志文件的线程数。
- 默认值：4
- 最小值：1

7.2.19.10 memory-limit

- 指定 Raft Engine 使用内存的上限。
- 当该配置项未设置时，Raft Engine 默认使用系统总内存的 15%。
- 默认值：系统总内存 * 15%

7.2.20 security

安全相关配置项。

7.2.20.1 ca-path

- CA 文件路径
- 默认值：“”

7.2.20.2 cert-path

- 包含 X.509 证书的 PEM 文件路径
- 默认值：“”

7.2.20.3 key-path

- 包含 X.509 key 的 PEM 文件路径
- 默认值：“”

7.2.20.4 cert-allowed-cn

- 客户端提供的证书中，可接受的 X.509 通用名称列表。仅当提供的通用名称与列表中的条目之一完全匹配时，才会允许其请求。
- 默认值：[]。这意味着默认情况下禁用客户端证书 CN 检查。

7.2.20.5 redact-info-log 从 v4.0.8 版本开始引入

- 若开启该选项，日志中的用户数据会以 ? 代替。
- 默认值：false

7.2.21 security.encryption

静态加密 (TDE) 有关的配置项。

7.2.21.1 data-encryption-method

- 数据文件的加密方法。
- 可选值：`"plaintext"`，`"aes128-ctr"`，`"aes192-ctr"`，`"aes256-ctr"`
- 选择 `"plaintext"` 以外的值则表示启用加密功能。此时必须指定主密钥。
- 默认值：`"plaintext"`

7.2.21.2 data-key-rotation-period

- 指定 TiKV 轮换数据密钥的频率。
- 默认值：`7d`

7.2.21.3 enable-file-dictionary-log

- 启用优化，以减少 TiKV 管理加密元数据时的 I/O 操作和互斥锁竞争。
- 此配置参数默认启用，为避免可能出现的兼容性问题，请参考静态加密 - TiKV 版本间兼容性。
- 默认值：`true`

7.2.21.4 master-key

- 指定启用加密时的主密钥。若要了解如何配置主密钥，可以参考静态加密 - 配置加密。

7.2.21.5 previous-master-key

- 指定轮换新主密钥时的旧主密钥。旧主密钥的配置格式与主密钥相同。若要了解如何配置主密钥，可以参考静态加密 - 配置加密。

7.2.22 import

用于 TiDB Lightning 导入及 BR 恢复相关的配置项。

7.2.22.1 num-threads

- 处理 RPC 请求的线程数量。
- 默认值：8
- 最小值：1

7.2.23 gc

7.2.23.1 `enable-compaction-filter` 从 v5.0 版本开始引入

- 是否开启 GC in Compaction Filter 特性
- 默认值：true

7.2.24 backup

用于 BR 备份相关的配置项。

7.2.24.1 `num-threads`

- 处理备份的工作线程数量。
- 默认值：CPU * 0.5，但最大为 8
- 可调整范围：[1, CPU]
- 最小值：1

7.2.24.2 `enable-auto-tune` 从 v5.4 版本开始引入

- 在集群资源占用率较高的情况下，是否允许 BR 自动限制备份使用的资源，减少对集群的影响。详情见自动调节。
- 默认值：true

7.2.25 cdc

用于 TiCDC 捕捉变更数据相关的配置项。

7.2.25.1 `min-ts-interval`

- 定期推进 Resolved TS 的时间间隔。
- 默认值：1s

7.2.25.2 `old-value-cache-memory-quota`

- 缓存在内存中的 TiCDC Old Value 的条目占用内存的上限。
- 默认值：512MB

7.2.25.3 `sink-memory-quota`

- 缓存在内存中的 TiCDC 数据变更事件占用内存的上限。
- 默认值：512MB

7.2.25.4 incremental-scan-speed-limit

- 增量扫描历史数据的速度上限。
- 默认值：128MB，即 128MB 每秒。

7.2.25.5 incremental-scan-threads

- 增量扫描历史数据任务的线程个数。
- 默认值：4，即 4 个线程

7.2.25.6 incremental-scan-concurrency

- 增量扫描历史数据任务的最大并发执行个数。
- 默认值：6，即最多并发执行 6 个任务
- 注意：incremental-scan-concurrency 需要大于等于 incremental-scan-threads，否则 TiKV 启动会报错。

7.2.26 resolved-ts

用于维护 Resolved TS 以服务 Stale Read 请求的相关配置项。

7.2.26.1 enable

- 是否为所有 Region 维护 Resolved TS
- 默认值：true

7.2.26.2 advance-ts-interval

- 定期推进 Resolved TS 的时间间隔。
- 默认值：1s

7.2.26.3 scan-lock-pool-size

- 初始化 Resolved TS 时 TiKV 扫描 MVCC（多版本并发控制）锁数据的线程个数。
- 默认值：2，即 2 个线程

7.2.27 pessimistic-txn

悲观事务使用方法请参考TiDB 悲观事务模式。

7.2.27.1 wait-for-lock-timeout

- 悲观事务在 TiKV 中等待其他事务释放锁的最长时间。若超时则会返回错误给 TiDB 并由 TiDB 重试加锁，语句最长等锁时间由 innodb_lock_wait_timeout 控制。
- 默认值：1s
- 最小值：1ms

7.2.27.2 wake-up-delay-duration

- 悲观事务释放锁时，只会唤醒等锁事务中 start_ts 最小的事务，其他事务将会延迟 wake-up-delay-duration 之后被唤醒。
- 默认值：20ms

7.2.27.3 pipelined

- 开启流水线式加悲观锁流程。开启该功能后，TiKV 在检测数据满足加锁要求后，立刻通知 TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但有较低概率出现悲观锁异步写入失败的情况，可能会导致悲观事务提交失败。
- 默认值：true

7.2.27.4 in-memory（从 v6.0.0 版本开始引入）

- 开启内存悲观锁功能。开启该功能后，悲观事务会尽可能在 TiKV 内存中存储悲观锁，而不将悲观锁写入磁盘，也不将悲观锁同步给其他副本，从而提升悲观事务的性能。但有较低概率出现悲观锁丢失的情况，可能会导致悲观事务提交失败。
- 默认值：true
- 注意：in-memory 仅在 pipelined 为 true 时生效。

7.2.28 quota

用于前台限流 (Quota Limiter) 相关的配置项。

当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果 TiKV 前台处理的读写请求量过大，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，你可以使用前台限流相关的 quota 配置项以限制前台各类请求占用的 CPU 资源。触发该限制的请求会被强制等待一段时间以让出 CPU 资源。具体等待时间与新增请求量相关，最多不超过max-delay-duration 的值。

> 警告：
>
> - 前台限流是 TiDB 在 v6.0.0 中引入的实验特性，不建议在生产环境中使用。
> - 该功能仅适合在资源有限的环境中使用，以保证 TiKV 在该环境下可以长期稳定地运行。如果在资源丰富的机型环境中开启该功能，可能会导致读写请求量达到峰值时 TiKV 的性能下降的问题。

7.2.28.1 foreground-cpu-time（从 v6.0.0 版本开始引入）

- 限制处理 TiKV 前台读写请求所使用的 CPU 资源使用量，这是一个软限制。
- 默认值：0（即无限制）
- 单位：millicpu（当该参数值为 1500 时，前端请求会消耗 1.5v CPU）。

7.2.28.2 foreground-write-bandwidth（从 v6.0.0 版本开始引入）

- 限制事务写入的带宽，这是一个软限制。
- 默认值：0KB（即无限制）

7.2.28.3 foreground-read-bandwidth（从 v6.0.0 版本开始引入）

- 限制事务读取数据和 Coprocessor 读取数据的带宽，这是一个软限制。
- 默认值：0KB（即无限制）

7.2.28.4 max-delay-duration（从 v6.0.0 版本开始引入）

- 单次前台读写请求被强制等待的最大时间。
- 默认值：500ms

7.2.29 causal-ts 从 v6.1.0 版本开始引入

用于 TiKV API V2（`storage.api-version = 2`）中时间戳获取相关的配置项。

为了降低写请求延迟，TiKV 会定期获取一批时间戳缓存在本地，避免频繁访问 PD。当本地缓存的时间戳用完，会立即发起一次时间戳请求。这种情况下，部分写请求的延迟会增大。TiKV 会根据负载情况动态调整时间戳缓存的大小，以减少这种情况的发生，大部分情况下不需要做参数调整。

> **警告：**
>
> - API V2 是 TiKV 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。

7.2.29.1 renew-interval

- 刷新本地缓存时间戳的周期。
- TiKV 会每隔 `renew-interval` 发起一次时间戳更新，并根据前一周期的使用情况，来调整时间戳的缓存数量。这个参数配置过大会导致不能及时反映最新的 TiKV 负载变化。而配置过小则会增加 PD 的负载。如果写流量剧烈变化、频繁出现时间戳耗尽、写延迟增加，可以适当调小这个参数，但需要同时关注 PD 的负载情况。
- 默认值：100ms

7.2.29.2 renew-batch-min-size

- 时间戳缓存的最小数量。

- TiKV 会根据前一周期的使用情况，来调整时间戳的缓存数量。如果本地缓存使用率偏低，TiKV 会逐步降低缓存数量，直至等于 `renew-batch-min-size`。如果业务中经常出现突发的大流量写入，可以适当调大这个参数。注意这个参数是单个 tikv-server 的缓存大小，如果配置过大、而同时集群中 tikv-server 较多，会导致 TSO 消耗过快。
- Grafana TiKV-Raw 面板下 Causal timestamp 中的 TSO batch size 是根据业务负载动态调整后的本地缓存数量。可以参考该监控指标值调整这个参数的大小。
- 默认值：100

7.3 TiFlash 配置参数

本文介绍了与部署使用 TiFlash 相关的配置参数。

7.3.1 PD 调度参数

可通过pd-ctl 调整参数。如果你使用 TiUP 部署，可以用 `tiup ctl pd` 代替 `pd-ctl -u <pd_ip:pd_port>` 命令。

- `replica-schedule-limit`：用来控制 replica 相关 operator 的产生速度（涉及到下线、补副本的操作都与该参数有关）

> 注意：
>
> 不要超过 `region-schedule-limit`，否则会影响正常 TiKV 之间的 Region 调度。

- `store-balance-rate`：用于限制每个 TiKV store 或 TiFlash store 的 Region 调度速度。注意这个参数只对新加入集群的 store 有效，如果想立刻生效请用下面的方式。

> 注意：
>
> 4.0.2 版本之后（包括 4.0.2 版本）废弃了 `store-balance-rate` 参数且 `store limit` 命令有部分变化。该命令变化的细节请参考 store-limit 文档。

```
- 使用 `pd-ctl -u <pd_ip:pd_port> store limit <store_id> <value>` 命令单独设置某个 store 的
    ↪ Region 调度速度。（`store_id` 可通过 `pd-ctl -u <pd_ip:pd_port> store` 命令获得）
    ↪ 如果没有单独设置，则继承 `store-balance-rate` 的设置。你也可以使用 `pd-ctl -u <pd_ip:
    ↪ pd_port> store limit` 命令查看当前设置值。
```

- `replication.location-labels`：用来表示 TiKV 实例的拓扑关系，其中 key 的顺序代表了不同标签的层次关系。在 TiFlash 开启的情况下需要使用`pd-ctl config placement-rules` 来设置默认值，详细可参考 geo-distributed-deployment-topology。

7.3.2 TiFlash 配置参数

7.3.2.1 配置文件 tiflash.toml

```
### TiFlash TCP/HTTP 等辅助服务的监听 host。建议配置成 0.0.0.0，即监听本机所有 IP 地址。
listen_host = "0.0.0.0"
### TiFlash TCP 服务的端口
tcp_port = 9000
### TiFlash HTTP 服务的端口
http_port = 8123
### 数据块元信息的内存 cache 大小限制，通常不需要修改
mark_cache_size = 5368709120
### 数据块 min-max 索引的内存 cache 大小限制，通常不需要修改
minmax_index_cache_size = 5368709120
### DeltaIndex 内存 cache 大小限制，默认为 0，代表没有限制
delta_index_cache_size = 0

### TiFlash 数据的存储路径。如果有多个目录，以英文逗号分隔。
### 从 v4.0.9 版本开始，不推荐使用 path 及 path_realtime_mode 参数。推荐使用 [storage]
    ↪ 下的配置项代替，这样在多盘部署的场景下能更好地利用节点性能。
### 从 v5.2.0 版本开始，如果要使用配置项 storage.io_rate_limit，需要同时将 TiFlash
    ↪ 的数据存储路径设置为 storage.main.dir。
### 当 [storage] 配置项存在的情况下，path 和 path_realtime_mode 两个配置会被忽略。
## path = "/tidb-data/tiflash-9000"
### 或
## path = "/ssd0/tidb-data/tiflash,/ssd1/tidb-data/tiflash,/ssd2/tidb-data/tiflash"
### 默认为 false。如果设为 true，且 path 配置了多个目录，表示在第一个目录存放最新数据，
    ↪ 在其他目录存放较旧的数据。
## path_realtime_mode = false

### TiFlash 临时文件的存放路径。默认使用 [`path` 或者 `storage.latest.dir` 的第一个目录] + "/tmp"
## tmp_path = "/tidb-data/tiflash-9000/tmp"

### 存储路径相关配置，从 v4.0.9 开始生效
[storage]
    ## 该参数从 v5.2.0 开始废弃，请使用 `[storage.io_rate_limit]` 相关配置
    # bg_task_io_rate_limit = 0

    ## DTFile 储存文件格式
    ## * format_version = 1 老旧文件格式，已废弃
    ## * format_version = 2 v6.0.0 以前版本的默认文件格式
    ## * format_version = 3 v6.0.0 及以后版本的默认文件格式，具有更完善的检验功能
    # format_version = 3

    [storage.main]
    ## 用于存储主要的数据，该目录列表中的数据占总数据的 90% 以上。
```

```
dir = [ "/tidb-data/tiflash-9000" ]
## 或
# dir = [ "/ssd0/tidb-data/tiflash", "/ssd1/tidb-data/tiflash" ]

## storage.main.dir 存储目录列表中每个目录的最大可用容量。
## * 在未定义配置项，或者列表中全填 0 时，会使用目录所在的硬盘容量
## * 以 byte 为单位。目前不支持如 "10GB" 的设置
## * capacity 列表的长度应当与 dir 列表长度保持一致
## 例如:
# capacity = [ 10737418240, 10737418240 ]

[storage.latest]
## 用于存储最新的数据，大约占总数据量的 10% 以内，需要较高的 IOPS。
## 默认情况该项可留空。在未配置或者为空列表的情况下，会使用 storage.main.dir 的值。
# dir = [ ]
## storage.latest.dir 存储目录列表中，每个目录的最大可用容量。
# capacity = [ 10737418240, 10737418240 ]

## [storage.io_rate_limit] 相关配置从 v5.2.0 开始引入。
[storage.io_rate_limit]
## 该配置项是 I/O 限流功能的开关，默认关闭。TiFlash 的 I/O
    ↪ 限流功能适用于磁盘带宽较小且磁盘带宽大小明确的云盘场景。
## I/O 限流功能限制下的读写流量总带宽，单位为 Byte，默认值为 0，即默认关闭 I/O 限流功能。
# max_bytes_per_sec = 0
## max_read_bytes_per_sec 和 max_write_bytes_per_sec 的含义和 max_bytes_per_sec 类似，分别指
    ↪ I/O 限流功能限制下的读流量总带宽和写流量总带宽。
## 分别用两个配置项控制读写带宽限制，适用于一些读写带宽限制分开计算的云盘，例如 GCP 上的
    ↪ persistent disk。
## 当 max_bytes_per_sec 配置不为 0 时，优先使用 max_bytes_per_sec。
# max_read_bytes_per_sec = 0
# max_write_bytes_per_sec = 0

## 下面的参数用于控制不同 I/O 流量类型分配到的带宽权重，一般不需要调整。
## TiFlash 内部将 I/O 请求分成 4 种类型：前台写、后台写、前台读、后台读。
## I/O 限流初始化时，TiFlash 会根据下面的权重 (weight) 比例分配带宽。
## 以下默认配置表示每一种流量将获得 25 / (25 + 25 + 25 + 25) = 25% 的权重。
## 如果将 weight 配置为 0，则对应的 I/O 操作不会被限流。
# foreground_write_weight = 25
# background_write_weight = 25
# foreground_read_weight = 25
# background_read_weight = 25
## TiFlash 支持根据当前的 I/O 负载情况自动调整各种 I/O 类型的限流带宽，有可能会超过设置的权重
    ↪ 。
## auto_tune_sec 表示自动调整的执行间隔，单位为秒。设为 0 表示关闭自动调整。
# auto_tune_sec = 5
```

```
[flash]
    tidb_status_addr = tidb status 端口地址 # 多个地址以逗号分割
    service_addr =  TiFlash raft 服务 和 coprocessor 服务监听地址

## 多个 TiFlash 节点会选一个 master 来负责往 PD 增删 placement rule，通过 flash.flash_cluster
    ↪ 中的参数控制。
[flash.flash_cluster]
    refresh_interval = master 定时刷新有效期
    update_rule_interval = master 定时向 tidb 获取 TiFlash 副本状态并与 pd 交互
    master_ttl = master 选出后的有效期
    cluster_manager_path = pd buddy 所在目录的绝对路径
    log = pd buddy log 路径

[flash.proxy]
    addr = proxy 监听地址，不填则默认是 127.0.0.1:20170
    advertise-addr = 外部访问 addr 的地址，不填则默认是 "addr"
    data-dir = proxy 数据存储路径
    config = proxy 配置文件路径
    log-file = proxy log 路径
    log-level = proxy log 级别，默认是 "info"
    status-addr = 拉取 proxy metrics | status 信息的监听地址，不填则默认是 127.0.0.1:20292
    advertise-status-addr = 外部访问 status-addr 的地址，不填则默认是 "status-addr"

[logger]
    ## log 级别（支持 trace、debug、information、warning、error），默认是 "debug"
    level = debug
    log = TiFlash log 路径
    errorlog = TiFlash error log 路径
    ## 单个日志文件的大小，默认是 "100M"
    size = "100M"
    ## 最多保留日志文件个数，默认是 10
    count = 10

[raft]
    pd_addr = pd 服务地址 # 多个地址以逗号隔开

[status]
    ## Prometheus 拉取 metrics 信息的端口，默认是 8234
    metrics_port = 8234

[profiles]

[profiles.default]
    ## 存储引擎的 segment 分裂是否使用逻辑分裂。使用逻辑分裂可以减小写放大，
```

```
    ↪ 但是会造成一定程度的硬盘空间回收不及时。默认为 false。不建议修改默认选项。
  # dt_enable_logical_split = false

  ## 单次 coprocessor 查询过程中，对中间数据的内存限制，单位为 byte，默认为 0，表示不限制
  max_memory_usage = 0

  ## 所有查询过程中，对中间数据的内存限制，单位为 byte，默认为 0，表示不限制
  max_memory_usage_for_all_queries = 0

  ## 从 v5.0 引入，表示 TiFlash Coprocessor 最多同时执行的 cop 请求数量。
    ↪ 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
    ↪ 则使用默认值，即物理核数的两倍。
  cop_pool_size = 0

  ## 从 v5.0 引入，表示 TiFlash Coprocessor 最多同时执行的 batch 请求数量。
    ↪ 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
    ↪ 则使用默认值，即物理核数的两倍。
  batch_cop_pool_size = 0

  ## 从 v6.1 引入，指定 TiFlash 执行来自 TiDB 的 ALTER TABLE ... COMPACT 请求时，
    ↪ 能同时并行处理的请求数量。
  ## 如果这个值没有设置或设为了 0，则会采用默认值（1）。
  manual_compact_pool_size = 1

  ## 从 v5.4.0 引入，表示是否启用弹性线程池，这项功能可以显著提高 TiFlash 在高并发场景的 CPU
    ↪ 利用率。默认为 true。
  # enable_elastic_threadpool = true
  # TiFlash 存储引擎的压缩算法，支持 LZ4、zstd 和 LZ4HC，大小写不敏感。默认使用 LZ4 算法。
  dt_compression_method = "LZ4"

  # TiFlash 存储引擎的压缩级别，默认为 1。如果 dt_compression_method 设置为 LZ4，推荐将该值设为
    ↪  1；如果 dt_compression_method 设置为 zstd ，推荐将该值设为 -1 或 1，设置为 -1
    ↪ 的压缩率更小，但是读性能会更好；如果 dt_compression_method 设置为 LZ4HC，
    ↪ 推荐将该值设为 9。
  dt_compression_level = 1


### 安全相关配置，从 v4.0.5 开始生效
[security]
  ## 从 v5.0 引入，控制是否开启日志脱敏
  ## 若开启该选项，日志中的用户数据会以 `?` 代替显示
  ## 注意，tiflash-learner 对应的安全配置选项为 `security.redact-info-log`，需要在 tiflash-
    ↪ learner.toml 中另外开启
  # redact_info_log = false
```

```
    ## 包含可信 SSL CA 列表的文件路径。如果你设置了该值，`cert_path` 和 `key_path`
        ↪ 中的路径也需要填写
    # ca_path = "/path/to/ca.pem"
    ## 包含 PEM 格式的 X509 certificate 文件路径
    # cert_path = "/path/to/tiflash-server.pem"
    ## 包含 PEM 格式的 X509 key 文件路径
    # key_path = "/path/to/tiflash-server-key.pem"
```

7.3.2.2 配置文件 tiflash-learner.toml

```
[server]
    engine-addr = 外部访问 TiFlash coprocessor 服务的地址

[raftstore]
    ## 处理 Raft 数据落盘的线程池中线程的数量
    apply-pool-size = 4
    ## 处理 Raft 的线程池中线程的数量，即 Raftstore 线程池的大小。
    store-pool-size = 4
    ## 控制处理 snapshot 的线程数，默认为 2。设为 0 则关闭多线程优化
    snap-handle-pool-size = 2
    ## 控制 raft store 持久化 WAL 的最小间隔。通过适当增大延迟以减少 IOPS 占用，默认为 "4ms"，
        ↪ 设为 "0ms" 则关闭该优化。
    store-batch-retry-recv-timeout = "4ms"

[security]
    ## 从 v5.0 引入，控制是否开启日志脱敏
    ## 若开启该选项，日志中的用户数据会以 `?` 代替显示
    ## 默认值为 false
    redact-info-log = false
```

除以上几项外，其余功能参数和 TiKV 的配置相同。需要注意的是：key 为 engine 的 label 是保留项，不可手动配置。

7.3.2.3 多盘部署

TiFlash 支持单节点多盘部署。如果你的部署节点上有多块硬盘，可以通过以下的方式配置参数，提高节点的硬盘 I/O 利用率。TiUP 中参数配置格式参照详细 TiFlash 配置模版。

7.3.2.3.1 TiDB 集群版本低于 v4.0.9

TiDB v4.0.9 之前的版本中，TiFlash 只支持将存储引擎中的主要数据分布在多盘上。通过 path（TiUP 中为 data_dir）和 path_realtime_mode 这两个参数配置多盘部署。

多个数据存储目录在 path 中以英文逗号分隔，比如 /nvme_ssd_a/data/tiflash,/sata_ssd_b/data/tiflash,/sata_ssd_c/data/tiflash。如果你的节点上有多块硬盘，推荐把性能最好的硬盘目录放在最前面，以更好地利用节点性能。

如果节点上有多块相同规格的硬盘，可以把 path_realtime_mode 参数留空（或者把该值明确地设为 false）。这表示数据会在所有的存储目录之间进行均衡。但由于最新的数据仍然只会被写入到第一个目录，因此该目录所在的硬盘会较其他硬盘繁忙。

如果节点上有多块规格不一致的硬盘，推荐把 path_relatime_mode 参数设置为 true，并且把性能最好的硬盘目录放在 path 参数内的最前面。这表示第一个目录只会存放最新数据，较旧的数据会在其他目录之间进行均衡。注意此情况下，第一个目录规划的容量大小需要占总容量的约 10%。

7.3.2.3.2 TiDB 集群版本为 v4.0.9 及以上

TiDB v4.0.9 及之后的版本中，TiFlash 支持将存储引擎的主要数据和新数据都分布在多盘上。多盘部署时，推荐使用 [storage] 中的参数，以更好地利用节点的 I/O 性能。但 TiFlash 仍然支持TiDB 集群版本低于 v4.0.9 中的参数。

如果节点上有多块相同规格的硬盘，推荐把硬盘目录填到 storage.main.dir 列表中，storage.latest.dir 列表留空。TiFlash 会在所有存储目录之间分摊 I/O 压力以及进行数据均衡。

如果节点上有多块规格不同的硬盘，推荐把 I/O 性能较好的硬盘目录配置在 storage.latest.dir 中，把 I/O 性能较一般的硬盘目录配置在 storage.main.dir 中。例如节点上有一块 NVMe-SSD 硬盘加上两块 SATA-SSD 硬盘，你可以把 storage.latest.dir 设为 ["/nvme_ssd_a/data/tiflash"] 以及把 storage.main.dir 设为 ["/sata_ssd_b/data/tiflash", "/sata_ssd_c/data/tiflash"]。TiFlash 会根据两个目录列表分别进行 I/O 压力分摊及数据均衡。需要注意此情况下，storage.latest.dir 中规划的容量大小需要占总规划容量的约 10%。

> 警告：
>
> - [storage] 参数从 TiUP v1.2.5 版本开始支持。如果你的 TiDB 版本为 v4.0.9 及以上，请确保你的 TiUP 版本不低于 v1.2.5，否则 [storage] 中定义的数据目录不会被 TiUP 纳入管理。
> - 在 TiFlash 节点改为使用 [storage] 配置后，如果将集群版本降级到低于 v4.0.9，可能导致 TiFlash 部分数据丢失。

7.4 PD 配置文件描述

PD 配置文件比命令行参数支持更多的选项。你可以在 conf/config.toml 找到默认的配置文件。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见PD 配置参数。

7.4.0.1 name

- PD 节点名称。
- 默认值：`"pd"`
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字。

7.4.0.2 data-dir

- PD 存储数据路径。
- 默认值：`"default.${name}"`

7.4.0.3 client-urls

- PD 监听的客户端 URL 列表。
- 默认值：`"http://127.0.0.1:2379"`
- 如果部署一个集群，client URLs 必须指定当前主机的 IP 地址，例如 `"http://192.168.100.113:2379"`，如果是运行在 Docker 则需要指定为 `"http://0.0.0.0:2379"`。

7.4.0.4 advertise-client-urls

- 用于外部访问 PD 的 URL 列表。
- 默认值：`"${client-urls}"`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的 client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2379:2379`，那么可以设置为 `advertise-client-urls="http://192.168.100.113:2379"`，客户端可以通过 `http://192.168.100.113:2379` 来找到这个服务。

7.4.0.5 peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认：`"http://127.0.0.1:2380"`
- 如果部署一个集群，peer URLs 必须指定当前主机的 IP 地址，例如 `"http://192.168.100.113:2380"`，如果是运行在 Docker 则需要指定为 `"http://0.0.0.0:2380"`。

7.4.0.6 advertise-peer-urls

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认值：`"${peer-urls}"`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的 peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2380:2380`，那么可以设置为 `advertise-peer-urls="http://192.168.100.113:2380"`，其他 PD 节点可以通过 `http://192.168.100.113:2380` 来找到这个服务。

7.4.0.7 initial-cluster

- 初始化 PD 集群配置。
- 默认值：`"{name}=http://{advertise-peer-url}"`
- 例如，如果 name 是“pd”，并且 advertise-peer-urls 是 `"http://192.168.100.113:2380"`，那么 initial-cluster 就是 `"pd=http://192.168.100.113:2380"`。
- 如果启动三台 PD，那么 initial-cluster 可能就是 `pd1=http://192.168.100.113:2380, pd2=http://192.168.100.114:2380, pd3=192.168.100.115:2380`。

7.4.0.8 initial-cluster-state

- 集群初始状态
- 默认值：“new”

7.4.0.9 initial-cluster-token

- 用于在集群初始化阶段标识不同的集群。
- 默认值：“pd-cluster”
- 如果先后部署多个集群，且多个集群有相同配置的节点，应指定不同的 token 来隔离不同的集群。

7.4.0.10 lease

- PD Leader Key 租约超时时间，超时系统重新选举 Leader。
- 默认值：3
- 单位：秒

7.4.0.11 quota-backend-bytes

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

7.4.0.12 auto-compaction-mod

- 元信息数据库自动压缩的模式，可选项为 periodic（按周期），revision（按版本数）。
- 默认值：periodic

7.4.0.13 auto-compaction-retention

- compaction-mode 为 periodic 时为元信息数据库自动压缩的间隔时间；compaction-mode 设置为 revision 时为自动压缩的版本数。
- 默认值：1h

7.4.0.14 force-new-cluster

- 强制让该 PD 以一个新集群启动，且修改 raft 成员数为 1。
- 默认值：false

7.4.1 security

安全相关配置项。

7.4.1.1 cacert-path

- CA 文件路径
- 默认值：“”

7.4.1.2 cert-path

- 包含 X509 证书的 PEM 文件路径
- 默认值：“”

7.4.1.3 key-path

- 包含 X509 key 的 PEM 文件路径
- 默认值：“”

7.4.1.4 redact-info-log 从 v5.0 版本开始引入

- 控制 PD 日志脱敏的开关
- 该配置项值设为 true 时将对 PD 日志脱敏，遮蔽日志中的用户信息。
- 默认值：false

7.4.2 log

日志相关的配置项。

7.4.2.1 level

- 指定日志的输出级别。
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”
- 默认值：“info”

7.4.2.2 format

- 日志格式。
- 可选值：“text”，“json”
- 默认值：“text”

7.4.2.3 disable-timestamp

- 是否禁用日志中自动生成的时间戳。
- 默认值：false

7.4.3 log.file

日志文件相关的配置项。

7.4.3.1 max-size

- 单个日志文件最大大小，超过该值系统自动切分成多个文件。
- 默认值：300
- 单位：MiB
- 最小值为 1

7.4.3.2 max-days

- 日志保留的最长天数。
- 默认：0

7.4.3.3 max-backups

- 日志文件保留的最大个数。
- 默认：0

7.4.4 metric

监控相关的配置项。

7.4.4.1 interval

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

7.4.5 schedule

调度相关的配置项。

7.4.5.1 max-merge-region-size

- 控制 Region Merge 的 size 上限，当 Region Size 大于指定值时 PD 不会将其与相邻的 Region 合并。
- 默认：20

7.4.5.2 max-merge-region-keys

- 控制 Region Merge 的 key 上限，当 Region key 大于指定值时 PD 不会将其与相邻的 Region 合并。
- 默认：200000

7.4.5.3 patrol-region-interval

- 控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整
- 默认：10ms

7.4.5.4 split-merge-interval

- 控制对同一个 Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被 merge。
- 默认：1h

7.4.5.5 max-snapshot-count

- 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。
- 默认：64

7.4.5.6 max-pending-peer-count

- 控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。
- 默认值：64

7.4.5.7 max-store-down-time

- PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。
- 默认值：30m

7.4.5.8 max-store-preparing-time 从 v6.1.0 版本开始引入

- 控制 store 上线阶段的最长等待时间。在 store 的上线阶段，PD 可以查询该 store 的上线进度。当超过该配置项指定的时间后，PD 会认为该 store 已完成上线，无法再次查询这个 store 的上线进度，但是不影响 Region 向这个新上线 store 的迁移。通常用户无需修改该配置项。
- 默认值：48h

7.4.5.9 leader-schedule-limit

- 同时进行 leader 调度的任务个数。
- 默认值：4

7.4.5.10 region-schedule-limit

- 同时进行 Region 调度的任务个数
- 默认值：2048

7.4.5.11 hot-region-schedule-limit

- 控制同时进行的 hot Region 任务。该配置项独立于 Region 调度。
- 默认值：4

7.4.5.12 hot-region-cache-hits-threshold

- 设置识别热点 Region 所需的分钟数。只有当 Region 处于热点状态持续时间超过此分钟数时，PD 才会参与热点调度。
- 默认值：3

7.4.5.13 replica-schedule-limit

- 同时进行 replica 调度的任务个数。
- 默认值：64

7.4.5.14 merge-schedule-limit

- 同时进行的 Region Merge 调度的任务，设置为 0 则关闭 Region Merge。
- 默认值：8

7.4.5.15 high-space-ratio

- 设置 store 空间充裕的阈值。当节点的空间占用比例小于该阈值时，PD 调度时会忽略节点的剩余空间，主要根据实际数据量进行均衡。此配置仅在 `region-score-formula-version = v1` 时生效。
- 默认值：0.7
- 最小值：大于 0
- 最大值：小于 1

7.4.5.16 low-space-ratio

- 设置 store 空间不足的阈值。当某个节点的空间占用比例超过该阈值时，PD 会尽可能避免往该节点迁移数据，同时主要根据节点剩余空间大小进行调度，避免对应节点的磁盘空间被耗尽。
- 默认值：0.8
- 最小值：大于 0
- 最大值：小于 1

7.4.5.17 tolerant-size-ratio

- 控制 balance 缓冲区大小。
- 默认值：0（为 0 为自动调整缓冲区大小）
- 最小值：0

7.4.5.18 enable-cross-table-merge

- 设置是否开启跨表 merge。
- 默认值：true

7.4.5.19 region-score-formula-version 从 v5.0 版本开始引入

- 设置 Region 算分公式版本。
- 默认值：v2
- 可选值：v1，v2。v2 相比于 v1，变化会更平滑，空间回收引起的调度抖动情况会得到改善。

> **注意：**
>
> 如果是从 v4.0 升级至当前版本，默认不自动开启该算分公式新版本，以保证升级前后 PD 行为一致。若想切换算分公式的版本，使用需要手动通过 pd-ctl 设置切换，详见PD Control 文档。

7.4.5.20 enable-joint-consensus 从 v5.0 版本开始引入

- 是否使用 Joint Consensus 进行副本调度。关闭该特性时，PD 将采用一次调度一个副本的方式进行调度。
- 默认值：true

7.4.5.21 hot-regions-write-interval 从 v5.4.0 版本开始引入

- 设置 PD 存储 Hot Region 信息时间间隔。
- 默认值：10m

> **注意：**
>
> Hot Region 的信息一般 3 分钟更新一次。如果设置时间间隔小于 3 分钟，中间部分的更新可能没有意义。

7.4.5.22 hot-regions-reserved-days 从 v5.4.0 版本开始引入

- 设置 PD 保留的 Hot Region 信息的最长时间。单位为天。
- 默认值: 7

7.4.6 replication

副本相关的配置项。

7.4.6.1 max-replicas

- 所有副本数量，即 leader 与 follower 数量之和。默认为 3，即 1 个 leader 和 2 个 follower。当此配置被在线修改后，PD 会在后台通过调度使得 Region 的副本数量符合配置。
- 默认值：3

7.4.6.2 location-labels

- TiKV 集群的拓扑信息。
- 默认值：[]
- 配置集群拓扑

7.4.6.3 isolation-level

- TiKV 集群的最小强制拓扑隔离级别。
- 默认值：“”
- 配置集群拓扑

7.4.6.4 strictly-match-label

- 打开强制 TiKV Label 和 PD 的 location-labels 是否匹配的检查
- 默认值：false

7.4.6.5 enable-placement-rules

- 打开 placement-rules
- 默认值：false
- 参考Placement Rules 使用文档
- 4.0 实验性特性

7.4.6.6 flow-round-by-digit 从 v5.1 版本开始引入

- 默认值：3
- PD 会对流量信息的末尾数字进行四舍五入处理，减少 Region 流量信息变化引起的统计信息更新。该配置项用于指定对 Region 流量信息的末尾进行四舍五入的位数。例如流量 100512 会归约到 101000。默认值为 3。该配置替换了 trace-region-flow。

> 注意：
>
> 如果是从 v4.0 升级至当前版本，升级后的 flow-round-by-digit 行为和升级前的 trace-region-flow 行为默认保持一致：如果升级前 trace-region-flow 为 false，则升级后 flow-round-by-digit 为 127；如果升级前 trace-region-flow 为 true，则升级后 flow-round-by-digit 为 3。

7.4.7 label-property

标签相关的配置项。

7.4.7.1 key

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

7.4.7.2 value

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

7.4.8 dashboard

PD 中内置的TiDB Dashboard 相关配置项。

7.4.8.1 tidb-cacert-path

- CA 根证书文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值： “”

7.4.8.2 tidb-cert-path

- SSL 证书文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值： “”

7.4.8.3 tidb-key-path

- SSL 私钥文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值： “”

7.4.8.4 public-path-prefix

- 通过反向代理访问 TiDB Dashboard 时，配置反向代理提供服务的路径前缀。
- 默认值： “/dashboard”
- 若不通过反向代理访问 TiDB Dashboard，请勿配置该项，否则可能导致 TiDB Dashboard 无法正常访问。关于该配置的详细使用场景，参见通过反向代理使用 TiDB Dashboard。

7.4.8.5 enable-telemetry

- 是否启用 TiDB Dashboard 遥测功能。
- 默认值：true
- 参阅遥测了解该功能详情。

7.5 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。

7.5.1 常用操作

7.5.1.1 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'
```

7.5.1.2 在线修改 TiKV 配置

> **注意：**
>
> 在线修改 TiKV 配置项后，同时会自动修改 TiKV 的配置文件。但还需要使用 `tiup edit-config` 命令来修改对应的配置项，否则 `upgrade` 和 `reload` 等运维操作会将在线修改配置后的结果覆盖。修改配置的操作请参考：使用 TiUP 修改配置。执行 `tiup edit-config` 后不需要执行 `tiup reload` 操作。

执行 SQL 语句 `set config`，可以结合实例地址或组件类型来修改单个实例配置或全部实例配置，如：

修改全部 TiKV 实例配置：

> **注意：**
>
> 建议使用反引号包裹变量名称。

```
set config tikv `split.qps-threshold`=1000
```

修改单个 TiKV 实例配置：

```
set config "127.0.0.1:20180" `split.qps-threshold`=1000
```

设置成功会返回 `Query OK`：

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

在批量修改时如果有错误发生，会以 warning 的形式返回：

```
set config tikv `log-level`='warn';
```

```
Query OK, 0 rows affected, 1 warning (0.04 sec)
```

```
show warnings;
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------------
    ↪
| Warning | 1105 | bad request to http://127.0.0.1:20180/config: fail to update, error: "config
    ↪ log-level can not be changed" |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

批量修改配置不保证原子性，可能出现某些实例成功，而某些失败的情况。如使用 `set tikv key=val` 命令修改整个 TiKV 集群配置时，可能有部分实例失败，请执行 `show warnings` 进行查看。

如遇到部分修改失败的情况，需要重新执行对应的修改语句，或通过修改单个实例的方式完成修改。如果因网络或者机器故障等原因无法访问到的 TiKV，需要等到恢复后再次进行修改。

针对 TiKV 可在线修改的配置项，如果成功修改后，修改的结果会被持久化到配置文件中，后续以配置文件中的配置为准。某些配置项名称可能和 TiDB 预留关键字冲突，如 `limit`、`key` 等，对于此类配置项，需要用反引号 ` 包裹起来，如 `` `raftstore.raft-log-gc-size-limit` ``。

支持的配置项列表如下：

配置项	简介
raftstore.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检查
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-tombstones	触发 RocksDB compaction 需要的 tombstone 个数

配置项	简介
raftstore.region-compact-tombstones-percent	触发 RocksDB compaction 需要的 tombstone 所占比例
raftstore.pd-heartbeat-tick-interval	触发 Region 对 PD 心跳的时间间隔
raftstore.pd-store-heartbeat-tick-interval	触发 store 对 PD 心跳的时间间隔
raftstore.snap-mgr-gc-tick-interval	触发回收过期 snapshot 文件的时间间隔
raftstore.snap-gc-timeout	snapshot 文件的最长保存时间

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.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，超过将会向 PD 校验自己是否已经被删除
raftstore.abnormal-leader-missing-duration	允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中

配置项	简介
raftstore.peer-stale-state-check-interval	触发检验副本是否处于无主状态的时间间隔
raftstore.consistency-check-interval	触发一致性检查的时间间隔（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.backup-import-sst-interval	触发检查过期 SST 文件的时间间隔
raftstore.local-read-batch-size	一轮处理读请求的最大个数
raftstore.hibernate-timeout	启动后进入静默状态前需要等待的最短时间，在该时间段内不会进入静默状态（未 release）

配置项	简介
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
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

配置项	简介
raftstore.apply-max-batch-size	Apply 状态机由 BatchSystem 批量执行数据写入请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。

配置项	简介
raftstore.store-max-batch-size	Raft 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。
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

配置项	简介
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
coprocessor.region-max-size	Region 容量空间的最大值
coprocessor.region-split-size	分裂后新 Region 的大小
coprocessor.region-max-keys	Region 最多允许的 key 的个数
coprocessor.region-split-keys	分裂后新 Region 的 key 的个数

配置项	简介
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
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

配置项	简介
quota.foreground-write-bandwidth	限制事务写入的带宽，软限制
quota.foreground-read-bandwidth	限制事务读取数据和 Coprocessor 读取数据的带宽，软限制
quota.max-delay-duration	单次前台读写请求被强制等待的最大时间
gc.ratio-threshold	跳过 Region GC 的阈值（GC 版本个数/key 个数）

配置项	简介
gc.batch-keys	一轮处理 key 的个数
gc.max-write-bytes-per-sec	一秒可写入 RocksDB 的最大字节数
gc.enable-compaction-filter	是否使用 compaction filter
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

配置项	简介
{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
{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

配置项	简介
{db-name}.{cf-name}.write-buffer-size	memtable 大小
{db-name}.{cf-name}.max-write-buffer-number	最大 memtable 个数
{db-name}.{cf-name}.max-bytes-for-level-base	base level (L1) 最大字节数
{db-name}.{cf-name}.target-file-size-base	base level 的目标文件大小
{db-name}.{cf-name}.level0-file-num-compaction-trigger	触发 compaction 的 L0 文件最大个数
{db-name}.{cf-name}.level0-slowdown-writes-trigger	触发 write stall 的 L0 文件最大个数

配置项	简介
{db-name}.{cf-name}.level0-stop-writes-trigger	完全阻停写入的 L0 文件最大个数
{db-name}.{cf-name}.max-compaction-bytes	一次 compaction 最大写入字节数
{db-name}.{cf-name}.max-bytes-for-level-multiplier	每一层的默认放大倍数
{db-name}.{cf-name}.disable-auto-compactions	自动 compaction 的开关
{db-name}.{cf-name}.soft-pending-compaction-bytes-limit	pending compaction bytes 的软限制
{db-name}.{cf-name}.hard-pending-compaction-bytes-limit	pending compaction bytes 的硬限制

配置项	简介
{db-name}.{cf-name}.titan.blob-run-mode	处理 blob 文件的模式
server.grpc-memory-pool-quota	gRPC 可使用的内存大小限制
server.max-grpc-send-msg-len	gRPC 可发送的最大消息长度
server.raft-msg-max-batch-size	单个 gRPC 消息可包含的最大 Raft 消息个数
storage.block-cache.capacity	共享 block cache 的大小（自 v4.0.3 起支持）
storage.scheduler-worker-pool-size	Scheduler 线程池中线程的数量

配置项	简介
backup.num-threads	backup 线程的数量（自 v4.0.3 起支持）
split.qps-threshold	对 Region 执行 load-base-split 的阈值。如果连续一段时间内，某个 Region 的读请求的 QPS 超过 qps-threshold，则切分该 Region

配置项	简介
split.byte-threshold	对 Region 执行 load-base-split 的阈值。如果连续一段时间内，某个 Region 的读请求的流量超过 byte-threshold，则切分该 Region

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

7.5.1.3 在线修改 PD 配置

PD 暂不支持单个实例拥有独立配置。所有实例共享一份配置，可以通过下列方式修改 PD 的配置项：

```
set config pd `log.level`='info'
```

设置成功会返回 Query OK：

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

针对 PD 可在线修改的配置项，成功修改后则会持久化到 etcd 中，不会对配置文件进行持久化，后续以 etcd 中的配置为准。同上，若和 TiDB 预留关键字冲突，需要用反引号 ` 包裹此类配置项，例如 `schedule.leader-schedule-limit`。

支持配置项列表如下：

配置项	简介
log.level	日志级别
cluster-version	集群的版本
schedule.max-merge-region-size	控制 Region Merge 的 size 上限（单位是 MB）
schedule.max-merge-region-keys	控制 Region Merge 的 key 数量上限
schedule.patrol-region-interval	控制 replicaChecker 检查 Region 健康状态的运行频率
schedule.split-merge-interval	控制对同一个 Region 做 split 和 merge 操作的间隔
schedule.max-snapshot-count	控制单个 store 最多同时接收或发送的 snapshot 数量

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。

7.5.1.4 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

7.5.1.5 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)
```

8 系统变量

TiDB 系统变量的行为与 MySQL 相似，变量的作用范围可以是会话级别有效 (Session Scope) 或全局范围有效 (Global Scope)。其中：

- 对 SESSION 作用域变量的更改，设置后只影响当前会话。
- 对 GLOBAL 作用域变量的更改，设置后立即生效。如果该变量也有 SESSION 作用域，已经连接的所有会话 (包括当前会话) 将继续使用会话当前的 SESSION 变量值。

- 要设置变量值，可使用SET 语句。

```
# 以下两个语句等价地改变一个 Session 变量
SET tidb_distsql_scan_concurrency = 10;
SET SESSION tidb_distsql_scan_concurrency = 10;

# 以下两个语句等价地改变一个 Global 变量
SET @@global.tidb_distsql_scan_concurrency = 10;
SET  GLOBAL tidb_distsql_scan_concurrency = 10;
```

注意：

部分 GLOBAL 作用域的变量会持久化到 TiDB 集群中。文档中的变量有一个“是否持久化到集群”的说明，可以为“是”或者“否”。

- 对于持久化到集群的变量，当该全局变量被修改后，会通知所有 TiDB 服务器刷新其系统变量缓存。在集群中增加一个新的 TiDB 服务器时，或者重启现存的 TiDB 服务器时，都将自动使用该持久化变量。
- 对于不持久化到集群的变量，对变量的修改只对当前连接的 TiDB 实例生效。如果需要保留设置过的值，需要在 `tidb.toml` 配置文件中声明。

此外，由于应用和连接器通常需要读取 MySQL 变量，为了兼容这一需求，在 TiDB 中，部分 MySQL 的变量既可读取也可设置。例如，尽管 JDBC 连接器不依赖于查询缓存 (query cache) 的行为，但仍然可以读取和设置查询缓存。

注意：

变量取较大值并不总会带来更好的性能。由于大部分变量对单个连接生效，设置变量时，还应考虑正在执行语句的并发连接数量。

确定安全值时，应考虑变量的单位：

- 如果单位为线程，安全值通常取决于 CPU 核的数量。
- 如果单位为字节，安全值通常小于系统内存的总量。
- 如果单位为时间，单位可能为秒或毫秒。

单位相同的多个变量可能会争夺同一组资源。

8.1 变量参考

8.1.1 allow_auto_random_explicit_insert 从 v4.0.3 版本开始引入

- 作用域：SESSION | GLOBAL

- 是否持久化到集群：是
- 默认值：OFF
- 是否允许在 INSERT 语句中显式指定含有 AUTO_RANDOM 属性的列的值。

8.1.2 auto_increment_increment

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 65535]
- 控制 AUTO_INCREMENT 自增值字段的自增步长。该变量常与 auto_increment_offset 一起使用。

8.1.3 auto_increment_offset

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 65535]
- 控制 AUTO_INCREMENT 自增值字段的初始值。该变量常与 auto_increment_increment 一起使用。示例如下：

```
mysql> CREATE TABLE t1 (a int not null primary key auto_increment);
Query OK, 0 rows affected (0.10 sec)

mysql> set auto_increment_offset=1;
Query OK, 0 rows affected (0.00 sec)

mysql> set auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (),(),(),();
Query OK, 4 rows affected (0.04 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t1;
+----+
| a  |
+----+
|  1 |
|  4 |
|  7 |
| 10 |
+----+
4 rows in set (0.00 sec)
```

8.1.4 autocommit

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 用于设置在非显式事务时是否自动提交事务。更多信息，请参见事务概述。

8.1.5 character_set_client

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 这个变量表示从客户端发出的数据所用的字符集。有关更多 TiDB 支持的字符集和排序规则，参阅字符集和排序规则文档。如果需要更改字符集，建议使用SET NAMES 语句。

8.1.6 character_set_connection

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 若没有为字符串常量指定字符集，该变量表示这些字符串常量所使用的字符集。

8.1.7 character_set_database

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 该变量表示当前默认在用数据库的字符集，不建议设置该变量。选择新的默认数据库后，服务器会更改该变量的值。

8.1.8 character_set_results

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 该变量表示数据发送至客户端时所使用的字符集。

8.1.9 character_set_server

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 当 CREATE SCHEMA 中没有指定字符集时，该变量表示这些新建的表结构所使用的字符集。

8.1.10 cte_max_recursion_depth

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1000
- 范围：[0, 4294967295]
- 这个变量用于控制公共表表达式的最大递归深度。

8.1.11 datadir

- 作用域：NONE
- 默认值：/tmp/tidb
- 这个变量表示数据存储的位置，位置可以是本地路径。如果数据存储在 TiKV 上，则可以是指向 PD 服务器的路径。
- 如果变量值的格式为 ip_address:port，表示 TiDB 在启动时连接到的 PD 服务器。

8.1.12 ddl_slow_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：300
- 单位：毫秒
- 耗时超过该阈值的 DDL 操作会被输出到日志。

8.1.13 default_authentication_plugin

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：mysql_native_password
- 可选值：mysql_native_password，caching_sha2_password
- 服务器和客户端建立连接时，这个变量用于设置服务器对外通告的默认身份验证方式。如要了解该变量的其他可选值，参见可用的身份验证插件。

8.1.14 foreign_key_checks

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 为保持兼容，TiDB 对外键检查返回 OFF。

8.1.15 hostname

- 作用域：NONE
- 默认值：(系统主机名)
- 这个变量一个只读变量，表示 TiDB server 的主机名。

8.1.16 init_connect

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 用户首次连接到 TiDB 服务器时，init_connect 特性允许 TiDB 自动执行一条或多条 SQL 语句。如果你有 CONNECTION_ADMIN 或者 SUPER 权限，这些 SQL 语句将不会被自动执行。如果这些语句执行报错，你的用户连接将被终止。

8.1.17 innodb_lock_wait_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：50
- 范围：[1, 3600]
- 单位：秒
- 悲观事务语句等锁时间。

8.1.18 interactive_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：28800
- 范围：[1, 31536000]
- 单位：秒
- 该变量表示交互式用户会话的空闲超时。交互式用户会话是指使用 CLIENT_INTERACTIVE 选项调用 mysql_real_connect() API 建立的会话（例如：MySQL shell 客户端）。该变量与 MySQL 完全兼容。

8.1.19 last_plan_from_binding 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 该变量用来显示上一条执行的语句所使用的执行计划是否来自 binding 的执行计划。

8.1.20 last_plan_from_cache 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来显示上一个 execute 语句所使用的执行计划是不是直接从 plan cache 中取出来的。

8.1.21 license

- 作用域：NONE
- 默认值：Apache License 2.0
- 这个变量表示 TiDB 服务器的安装许可证。

8.1.22 max_execution_time

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 2147483647]
- 单位：毫秒
- 语句最长执行时间。默认值 (0) 表示无限制。

> **注意：**
>
> max_execution_time 目前对所有类型的语句生效，并非只对 SELECT 语句生效，与 MySQL 不同（只对SELECT 语句生效）。实际精度在 100ms 级别，而非更准确的毫秒级别。

8.1.23 plugin_dir

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：“”
- 指定加载插件的目录。

8.1.24 plugin_load

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：“”
- 指定 TiDB 启动时加载的插件，多个插件之间用逗号（,）分隔。

8.1.25 port

- 作用域：NONE
- 默认值：4000
- 范围：[0, 65535]
- 使用 MySQL 协议时 tidb-server 监听的端口。

8.1.26 rand_seed1

- 作用域：SESSION
- 默认值：0
- 范围：[0, 2147483647]
- 该变量用于为 SQL 函数 RAND() 中使用的随机值生成器添加种子。
- 该变量的行为与 MySQL 兼容。

8.1.27 rand_seed2

- 作用域：SESSION
- 默认值：0
- 范围：[0, 2147483647]
- 该变量用于为 SQL 函数 RAND() 中使用的随机值生成器添加种子。
- 该变量的行为与 MySQL 兼容。

8.1.28 require_secure_transport 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量控制是否所有 TiDB 的连接都在本地 socket 上进行通信，或使用 TLS。详情见为 TiDB 客户端服务端间通信开启加密传输。
- 该变量设置为 ON 时，必须使用开启 TLS 的会话连接到 TiDB，防止在 TLS 配置不正确时出现锁定的情况。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (security.require-secure-transport) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.29 skip_name_resolve 从 v5.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量控制 tidb-server 实例是否将主机名作为连接握手的一部分来解析。
- 当 DNS 不可靠时，可以启用该变量来提高网络性能。

> 注意：
>
> 当 skip_name_resolve 设置为 ON 时，身份信息中包含主机名的用户将无法登录服务器。例如：
>
> ```
> CREATE USER 'appuser'@'apphost' IDENTIFIED BY 'app-password';
> ```
>
> 该示例中，建议将 apphost 替换为 IP 地址或通配符 (%)。

8.1.30 socket

- 作用域：NONE
- 默认值：“”
- 使用 MySQL 协议时，tidb-server 所监听的本地 unix 套接字文件。

8.1.31 sql_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO ↪ ,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
- 这个变量控制许多 MySQL 兼容行为。详情见SQL 模式。

8.1.32 sql_select_limit 从 v4.0.2 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：18446744073709551615
- 范围：[0, 18446744073709551615]
- SELECT 语句返回的最大行数。

8.1.33 system_time_zone

- 作用域：NONE
- 默认值：(随系统)
- 该变量显示首次引导启动 TiDB 时的系统时区。另请参阅time_zone。

8.1.34 tidb_allow_batch_cop 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[0, 2]
- 这个变量用于控制 TiDB 向 TiFlash 发送 coprocessor 请求的方式，有以下几种取值：
  - 0：从不批量发送请求
  - 1：aggregation 和 join 的请求会进行批量发送
  - 2：所有的 cop 请求都会批量发送

8.1.35 tidb_allow_fallback_to_tikv 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 这个变量表示将 TiKV 作为备用存储引擎的存储引擎列表。当该列表中的存储引擎发生故障导致 SQL 语句执行失败时，TiDB 会使用 TiKV 作为存储引擎再次执行该 SQL 语句。目前支持设置该变量为 “” 或者 “tiflash”。如果设置该变量为 “tiflash”，当 TiFlash 返回超时错误 (对应的错误码为 ErrTiFlashServerTimeout) 时，TiDB 会使用 TiKV 作为存储引擎再次执行该 SQL 语句。

8.1.36 tidb_allow_function_for_expression_index 从 v5.2.0 版本开始引入

- 作用域：NONE
- 默认值：lower, md5, reverse, tidb_shard, upper, vitess_hash
- 这个变量用于显示创建表达式索引所允许使用的函数。

8.1.37 tidb_allow_mpp 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否使用 TiFlash 的 MPP 模式执行查询，可以设置的值包括：
  - 0 或 OFF，代表从不使用 MPP 模式
  - 1 或 ON，代表由优化器根据代价估算选择是否使用 MPP 模式（默认）

MPP 是 TiFlash 引擎提供的分布式计算框架，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法。MPP 模式选择的详细说明参见控制是否选择 MPP 模式。

8.1.38 tidb_allow_remove_auto_inc 从 v2.1.18 和 v3.0.4 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除某个列的 AUTO_INCREMENT 属性。默认 (OFF) 为不允许。

8.1.39 tidb_analyze_version 从 v5.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：2
- 范围：[1, 2]
- 这个变量用于控制 TiDB 收集统计信息的行为。
- 在 v5.3.0 及之后的版本中，该变量的默认值为 2，作为实验特性启用，具体可参照统计信息简介文档。如果从 v5.3.0 之前版本的集群升级至 v5.3.0 及之后的版本，tidb_analyze_version 的默认值不发生变化。

8.1.40 tidb_auto_analyze_end_time

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动 ANALYZE 更新统计信息的结束时间。例如，只允许在凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

```
- tidb_auto_analyze_start_time='01:00 +0000'
- tidb_auto_analyze_end_time='03:00 +0000'
```

8.1.41 tidb_auto_analyze_ratio

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.5
- 这个变量用来设置 TiDB 在后台自动执行ANALYZE TABLE 更新统计信息的阈值。0.5 指的是当表中超过 50% 的行被修改时，触发自动 ANALYZE 更新。可以指定 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 来限制自动 ANALYZE 的时间

> 注意：
>
> 当系统变量 tidb_enable_auto_analyze 设置为 ON 时，TiDB 才会触发 auto_analyze。

8.1.42 tidb_auto_analyze_start_time

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动 ANALYZE 更新统计信息的开始时间。例如，只允许在凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

```
- tidb_auto_analyze_start_time='01:00 +0000'
- tidb_auto_analyze_end_time='03:00 +0000'
```

8.1.43 tidb_max_auto_analyze_time 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：43200
- 范围：[0, 2147483647]
- 单位：秒
- 这个变量用于指定自动 ANALYZE 的最大执行时间。当执行时间超出指定的时间时，自动 ANALYZE 会被终止。当该变量值为 0 时，自动 ANALYZE 没有最大执行时间的限制。

8.1.44 tidb_backoff_lock_fast

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 默认值：10
- 范围：[1，2147483647]
- 这个变量用来设置读请求遇到锁的 backoff 时间。

8.1.45 tidb_backoff_weight

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：2
- 范围：[0，2147483647]
- 这个变量用来给 TiDB 的 backoff 最大时间增加权重，即内部遇到网络或其他组件 (TiKV, PD) 故障时，发送重试请求的最大重试时间。可以通过这个变量来调整最大重试时间，最小值为 1。

  例如，TiDB 向 PD 取 TSO 的基础超时时间是 15 秒，当 tidb_backoff_weight = 2 时，取 TSO 的最大超时时间为：基础时间 * 2 等于 30 秒。

  在网络环境较差的情况下，适当增大该变量值可以有效缓解因为超时而向应用端报错的情况；而如果应用端希望更快地接到报错信息，则应该尽量减小该变量的值。

8.1.46 tidb_batch_insert

- 作用域：SESSION
- 默认值：OFF
- 该变量允许 tidb_dml_batch_size 变量作用于 INSERT 语句。
- 只有该变量值为 OFF 才符合 ACID 原则。将该变量修改为任何其他值，会破坏 TiDB 的原子性和隔离性保证，因为每个 INSERT 语句都会被分割成较小的事务。

8.1.47 tidb_broadcast_join_threshold_count 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：10240
- 范围：[0，9223372036854775807]
- 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。

8.1.48 tidb_broadcast_join_threshold_size 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：104857600 (100 MiB)
- 范围：[0，9223372036854775807]
- 单位：字节
- 如果表大小（字节数）小于该值，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。

8.1.49 tidb_build_stats_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 这个变量用来设置 ANALYZE 语句执行时并发度。
- 当这个变量被设置得更大时，会对其它的查询语句执行性能产生一定影响。

8.1.50 tidb_capture_plan_baselines 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启自动捕获绑定功能。该功能依赖 Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。
- 开启该功能后会定期遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动创建绑定。

8.1.51 tidb_check_mb4_value_in_utf8

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：ON
- 设置该变量为 ON 可强制只存储基本多文种平面 (BMP) 编码区段内的 utf8 字符值。若要存储 BMP 区段外的 utf8 值，推荐使用 utf8mb4 字符集。
- 早期版本的 TiDB 中 (v2.1.x)，utf8 检查更为宽松。如果你的 TiDB 集群是从早期版本升级的，推荐关闭该变量，详情参阅升级与升级后常见问题。

8.1.52 tidb_committer_concurrency 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：128
- 范围：[1, 10000]
- 在单个事务的提交阶段，用于执行提交操作相关请求的 goroutine 数量。
- 若提交的事务过大，事务提交时的流控队列等待耗时可能会过长。此时，可以通过调大该配置项来加速提交。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (performance.committer-concurrency) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.53 tidb_checksum_table_concurrency

- 作用域：SESSION
- 默认值：4

- 这个变量用来设置 ADMIN CHECKSUM TABLE 语句执行时扫描索引的并发度。当这个变量被设置得更大时，会对其它的查询语句执行性能产生一定影响。

8.1.54 tidb_config

- 作用域：SESSION
- 默认值：“”
- 这个变量是一个只读变量，用来获取当前 TiDB Server 的配置信息。

8.1.55 tidb_constraint_check_in_place

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量仅适用于乐观事务模型。当这个变量设置为 OFF 时，唯一索引的重复值检查会被推迟到事务提交时才进行。这有助于提高性能，但对于某些应用，可能导致非预期的行为。详情见约束。
  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 0：

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

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

```
tidb> commit; -- 事务提交时才检查
```

```
ERROR 1062 : Duplicate entry '1' for key 'PRIMARY'
```

  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 1：

```
set @@tidb_constraint_check_in_place=1;
begin optimistic;
insert into t values (1);
```

```
ERROR 1062 : Duplicate entry '1' for key 'PRIMARY'
```

悲观事务模式中，始终默认执行约束检查。

8.1.56 tidb_current_ts

- 作用域：SESSION
- 默认值：0
- 这个变量是一个只读变量，用来获取当前事务的时间戳。

8.1.57 tidb_ddl_error_count_limit

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：512
- 范围：[0, 9223372036854775807]
- 这个变量用来控制 DDL 操作失败重试的次数。失败重试次数超过该参数的值后，会取消出错的 DDL 操作。

8.1.58 tidb_ddl_reorg_batch_size

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：256
- 范围：[32, 10240]
- 这个变量用来设置 DDL 操作 re-organize 阶段的 batch size。比如 ADD INDEX 操作，需要回填索引数据，通过并发 tidb_ddl_reorg_worker_cnt 个 worker 一起回填数据，每个 worker 以 batch 为单位进行回填。
  - 如果 ADD INDEX 操作时有较多 UPDATE 操作或者 REPLACE 等更新操作，batch size 越大，事务冲突的概率也会越大，此时建议调小 batch size 的值，最小值是 32。
  - 在没有事务冲突的情况下，batch size 可设为较大值，最大值是 10240，这样回填数据的速度更快，但是 TiKV 的写入压力也会变大。

8.1.59 tidb_ddl_reorg_priority

- 作用域：SESSION
- 默认值：PRIORITY_LOW
- 这个变量用来设置 ADD INDEX 操作 re-organize 阶段的执行优先级，可设置为 PRIORITY_LOW/PRIORITY_NORMAL/PRIORITY_HIGH。

8.1.60 tidb_ddl_reorg_worker_cnt

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 范围：[1, 256]
- 这个变量用来设置 DDL 操作 re-organize 阶段的并发度。

8.1.61 tidb_disable_txn_auto_retry

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 默认值：ON
- 这个变量用来设置是否禁用显式的乐观事务自动重试，设置为 ON 时，不会自动重试，如果遇到事务冲突需要在应用层重试。

  如果将该变量的值设为 OFF，TiDB 将会自动重试事务，这样在事务提交时遇到的错误更少。需要注意的是，这样可能会导致数据更新丢失。

  这个变量不会影响自动提交的隐式事务和 TiDB 内部执行的事务，它们依旧会根据 tidb_retry_limit 的值来决定最大重试次数。

  关于是否需要禁用自动重试，请参考重试的局限性。

  该变量只适用于乐观事务，不适用于悲观事务。悲观事务的重试次数由max_retry_count 控制。

8.1.62 tidb_distsql_scan_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：15
- 范围：[1, 256]
- 这个变量用来设置 scan 操作的并发度。
- AP 类应用适合较大的值，TP 类应用适合较小的值。对于 AP 类应用，最大值建议不要超过所有 TiKV 节点的 CPU 核数。
- 若表的分区较多可以适当调小该参数，避免 TiKV 内存溢出 (OOM)。

8.1.63 tidb_dml_batch_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 2147483647]
- 这个变量的值大于 0 时，TiDB 会将 INSERT 或 LOAD DATA 等语句在更小的事务中批量提交。这样可减少内存使用，确保大批量修改时事务大小不会达到 txn-total-size-limit 限制。
- 要使 tidb_dml_batch_size 作用于 INSERT 语句，你还需要将系统变量tidb_batch_insert 设置为 ON。
- 只有变量值为 0 时才符合 ACID 要求。否则无法保证 TiDB 的原子性和隔离性要求。

8.1.64 tidb_enable_1pc 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 指定是否在只涉及一个 Region 的事务上启用一阶段提交特性。比起传统两阶段提交，一阶段提交能大幅降低事务提交延迟并提升吞吐。

> 注意：

- 对于新创建的集群，默认值为 ON。对于升级版本的集群，如果升级前是 v5.0 以下版本，升级后默认值为 OFF。
- 启用 TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用 TiCDC 替代 TiDB Binlog。
- 启用该参数仅意味着一阶段提交成为可选的事务提交模式，实际由 TiDB 自行判断选择最合适的提交模式进行事务提交。

8.1.65 tidb_enable_amend_pessimistic_txn 从 v4.0.7 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启 AMEND TRANSACTION 特性。在悲观事务模式下开启该特性后，如果该事务相关的表存在并发 DDL 操作和 SCHEMA VERSION 变更，TiDB 会尝试对该事务进行 amend 操作，修正该事务的提交内容，使其和最新的有效 SCHEMA VERSION 保持一致，从而成功提交该事务而不返回 Information schema is changed 报错。该特性对以下并发 DDL 变更生效：
  - ADD COLUMN 或 DROP COLUMN 类型的 DDL 操作。
  - MODIFY COLUMN 或 CHANGE COLUMN 类型的 DDL 操作，且只对增大字段长度的操作生效。
  - ADD INDEX 或 DROP INDEX 类型的 DDL 操作，且操作的索引列须在事务开启之前创建。

> 注意：
>
> 目前该特性可能造成事务语义的变化，且与 TiDB Binlog 存在部分不兼容的场景，可以参考事务语义行为区别和与 TiDB Binlog 兼容问题汇总了解更多关于该特性的使用注意事项。

8.1.66 tidb_enable_async_commit 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 该变量控制是否启用 Async Commit 特性，使事务两阶段提交的第二阶段于后台异步进行。开启本特性能降低事务提交的延迟。

> 注意：
>
> - 对于新创建的集群，默认值为 ON。对于升级版本的集群，如果升级前是 v5.0 以下版本，升级后默认值为 OFF。

- 启用 TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用 TiCDC 替代 TiDB Binlog。
- 启用该参数仅意味着 Async Commit 成为可选的事务提交模式，实际由 TiDB 自行判断选择最合适的提交模式进行事务提交。

8.1.67 tidb_enable_auto_analyze 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 该变量控制 TiDB 是否以后台操作自动更新表的统计信息。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (performance.run-auto-analyze) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.68 tidb_enable_auto_increment_in_generated

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否允许在创建生成列或者表达式索引时引用自增列。

8.1.69 tidb_enable_batch_dml 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量控制 TiDB 是否允许非事务的“批处理”语句。由于 batch DML 不提供 ACID 保证，只有该变量设置为 OFF 时才是安全的。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (enable-batch-dml) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.70 tidb_enable_cascades_planner

> **警告：**
>
> 目前 cascades planner 为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启 cascades planner。

8.1.71 tidb_enable_chunk_rpc 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：ON
- 这个变量用来设置是否启用 Coprocessor 的 Chunk 数据编码格式。

8.1.72 tidb_enable_clustered_index 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：INT_ONLY
- 可选值：OFF，ON，INT_ONLY
- 这个变量用于控制默认情况下表的主键是否使用聚簇索引。“默认情况”即不显式指定 CLUSTERED /NONCLUSTERED 关键字的情况。可设置为 OFF/ON/INT_ONLY。
  - OFF 表示所有主键默认使用非聚簇索引。
  - ON 表示所有主键默认使用聚簇索引。
  - INT_ONLY 此时的行为受配置项 alter-primary-key 控制。如果该配置项取值为 true，则所有主键默认使用非聚簇索引；如果该配置项取值为 false，则由单个整数类型的列构成的主键默认使用聚簇索引，其他类型的主键默认使用非聚簇索引。

8.1.73 tidb_enable_collect_execution_info

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：ON
- 这个变量用于控制是否同时将各个执行算子的执行信息记录入 slow query log 中。

8.1.74 tidb_enable_column_tracking 从 v5.4.0 版本开始引入

> **警告：**
>
> 收集 PREDICATE COLUMNS 的统计信息目前为实验特性，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启 TiDB 对 PREDICATE COLUMNS 的收集。关闭该变量后，之前收集的 PREDICATE COLUMNS 会被清除。详情见收集部分列的统计信息。

8.1.75 tidb_enable_enhanced_security

- 作用域：NONE

- 默认值：OFF
- 这个变量表示所连接的 TiDB 服务器是否启用了安全增强模式 (SEM)。若要改变该变量值，你需要在 TiDB 服务器的配置文件中修改 enable-sem 项的值，并重启 TiDB 服务器。
- 安全增强模式受安全增强式 Linux 等系统设计的启发，削减拥有 MySQL SUPER 权限的用户能力，转而使用细粒度的 RESTRICTED 权限作为替代。这些细粒度的 RESTRICTED 权限如下：
  - RESTRICTED_TABLES_ADMIN：能够写入 mysql 库中的系统表，能查看 information_schema 表上的敏感列。
  - RESTRICTED_STATUS_ADMIN：能够在 SHOW STATUS 命令中查看敏感内容。
  - RESTRICTED_VARIABLES_ADMIN：能够在 SHOW [GLOBAL] VARIABLES 和 SET 命令中查看和设置包含敏感内容的变量。
  - RESTRICTED_USER_ADMIN：能够阻止其他用户更改或删除用户帐户。

8.1.76 tidb_restricted_read_only 从 v5.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 可选值：0 和 1
- 该变量可以控制整个集群的只读状态。开启后（即该值为 1），整个集群中的 TiDB 服务器都将进入只读状态，只有 SELECT、USE、SHOW 等不会修改数据的语句才能被执行，其他如 INSERT、UPDATE 等语句会被拒绝执行。
- 该变量开启只读模式只保证整个集群最终进入只读模式，当变量修改状态还没被同步到其他 TiDB 服务器时，尚未同步的 TiDB 仍然停留在非只读模式。
- 在变量开启时，正在执行的 SQL 语句不会受影响，只对新执行的 SQL 语句进行是否只读的检查。
- 在变量开启时，对于尚未提交的事务：
  - 如果有尚未提交的只读事务，可正常提交该事务。
  - 如果尚未提交的事务为非只读事务，在事务内执行写入的 SQL 语句会被拒绝。
  - 如果尚提交的事务已经有数据改动，其提交也会被拒绝。
- 当集群开启只读模式后，所有用户（包括 SUPER 用户）都无法执行可能写入数据的 SQL 语句，除非该用户被显式地授予了 RESTRICTED_REPLICA_WRITER_ADMIN 权限。
- 拥有 RESTRICTED_VARIABLES_ADMIN 或 SUPER 权限的用户可以修改该变量。如果用户开启了安全增强模式 (Security Enhanced Mode)，则只有 RESTRICTED_VARIABLES_ADMIN 权限的用户才能修改该变量。

8.1.77 tidb_enable_fast_analyze

> **警告：**
>
> 目前快速分析功能为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 默认值：OFF
- 这个变量用来控制是否启用统计信息快速分析功能。默认值 0 表示不开启。
- 快速分析功能开启后，TiDB 会随机采样约 10000 行的数据来构建统计信息。因此在数据分布不均匀或者数据量比较少的情况下，统计信息的准确度会比较低。这可能导致执行计划不优，比如选错索引。如果可以接受普通 ANALYZE 语句的执行时间，则推荐关闭快速分析功能。

8.1.78 tidb_enable_index_merge 从 v4.0 版本开始引入

> 注意：
>
> - 当集群从 v4.0.0 以下版本升级到 v5.4.0 及以上版本时，该变量开关默认关闭，防止升级后计划发生变化导致回退。
> - 当集群从 v4.0.0 及以上版本升级到 v5.4.0 及以上版本时，该变量开关保持升级前的状态。
> - 对于 v5.4.0 及以上版本的新建集群，该变量开关默认开启。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启 index merge 功能。

8.1.79 tidb_enable_legacy_instance_scope 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于允许使用 SET SESSION 对 INSTANCE 作用域的变量进行设置，用法同 SET GLOBAL。
- 为了兼容之前的 TiDB 版本，该变量值默认为 ON。

8.1.80 tidb_enable_list_partition 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来设置是否开启 LIST (COLUMNS)TABLE PARTITION 特性。

8.1.81 tidb_enable_mutation_checker (从 v6.0.0 版本开始引入)

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON

- 这个变量用于设置是否开启 mutation checker。mutation checker 是一项在 DML 语句执行过程中进行的数据索引一致性校验，校验报错会回滚当前语句。开启该校验会导致 CPU 使用轻微上升。详见数据索引一致性报错。
- 对于新创建的 v6.0.0 及以上的集群，默认值为 ON。对于升级版本的集群，如果升级前是低于 v6.0.0 的版本，升级后默认值为 OFF。

8.1.82 tidb_enable_noop_functions 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 默认情况下，用户尝试将某些语法用于尚未实现的功能时，TiDB 会报错。若将该变量值设为 ON，TiDB 则自动忽略此类功能不可用的情况，即不会报错。若用户无法更改 SQL 代码，可考虑将变量值设为 ON。
- 启用 noop 函数可以控制以下行为：
  - LOCK IN SHARE MODE 语法
  - SQL_CALC_FOUND_ROWS 语法
  - START TRANSACTION READ ONLY 和 SET TRANSACTION READ ONLY 语法
  - tx_read_only、transaction_read_only、offline_mode、super_read_only、read_only 以及 sql_auto_is_null 系统变量

> **警告：**
>
> 该变量只有在默认值 OFF 时，才算是安全的。因为设置 tidb_enable_noop_functions=1 后，TiDB 会自动忽略某些语法而不报错，这可能会导致应用程序出现异常行为。例如，允许使用语法 START TRANSACTION READ ONLY 时，事务仍会处于读写模式。

8.1.83 tidb_enable_outer_join_reorder 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 自 v6.1.0 起，TiDB 的Join Reorder 算法 开始支持 Outer Join。该变量用于控制这个支持行为，默认开启。
- 对于从旧版本升级上来的集群，该变量的默认值也会是 TRUE。

8.1.84 tidb_enable_paging 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制 IndexLookUp 算子是否使用分页 (paging) 方式发送 Coprocessor 请求。
- 适用场景：对于使用 IndexLookUp 和 Limit 并且 Limit 无法下推到 IndexScan 上的读请求，可能会出现读请求的延迟高、TiKV 的 Unified read pool CPU 使用率高的情况。在这种情况下，由于 Limit 算子只需要少部分数据，开启 tidb_enable_paging，能够减少处理数据的数量，从而降低延迟、减少资源消耗。

- 开启 tidb_enable_paging 后，Limit 无法下推且数量小于 960 的 IndexLookUp 请求会使用 paging 方式发送 Coprocessor 请求。Limit 的值越小，优化效果会越明显。

8.1.85 tidb_enable_parallel_apply 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 这个变量用于控制是否开启 Apply 算子并发，并发数由 tidb_executor_concurrency 变量控制。Apply 算子用来处理关联子查询且默认无并发，所以执行速度较慢。打开 Apply 并发开关可增加并发度，提高执行速度。目前默认关闭。

8.1.86 tidb_enable_prepared_plan_cache 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启Prepared Plan Cache。开启后，对 Prepare、Execute 请求的执行计划会进行缓存，以便在后续执行时跳过查询计划优化这个步骤，获得性能上的提升。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (prepared-plan-cache.enabled) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.87 tidb_enable_pseudo_for_outdated_stats 从 v5.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制优化器在一张表上的统计信息过期时的行为。
- 统计信息过期的判断标准：最近一次对某张表执行 ANALYZE 获得统计信息后，该表数据被修改的行数大于该表总行数的 80%，便可判定该表的统计信息已过期。该比例可通过pseudo-estimate-ratio 配置参数调整。
- 默认情况下（即该变量值为 ON 时），某张表上的统计信息过期后，优化器认为该表上除总行数以外的统计信息不再可靠，转而使用 pseudo 统计信息。将该变量值设为 OFF 后，即使统计信息过期，优化器也仍会使用该表上的统计信息。
- 如果表数据修改较频繁，没有及时对表执行 ANALYZE，但又希望执行计划保持稳定，可以将该变量值设为 OFF。

8.1.88 tidb_enable_rate_limit_action

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量控制是否为读数据的算子开启动态内存控制功能。读数据的算子默认启用tidb_distsql_scan_concurrency 所允许的最大线程数来读取数据。当单条 SQL 语句的内存使用每超过tidb_mem_quota_query 一次，读数据的算子会停止一个线程。

- 当读数据的算子只剩 1 个线程且当单条 SQL 语句的内存使用继续超过tidb_mem_quota_query 时，该 SQL 语句会触发其它的内存控制行为，例如落盘。

8.1.89 tidb_enable_slow_log

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：ON
- 这个变量用于控制是否开启 slow log 功能。

8.1.90 tidb_enable_stmt_summary 从 v3.0.4 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启 statement summary 功能。如果开启，SQL 的耗时等执行信息将被记录到系统表 information_schema.STATEMENTS_SUMMARY 中，用于定位和排查 SQL 性能问题。

8.1.91 tidb_enable_strict_double_type_check 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否可以用 DOUBLE 类型的无效定义创建表。该设置的目的是提供一个从 TiDB 早期版本升级的方法，因为早期版本在验证类型方面不太严格。
- 该变量的默认值 ON 与 MySQL 兼容。

例如，由于无法保证浮点类型的精度，现在将 DOUBLE(10) 类型视为无效。将 tidb_enable_strict_double_type_check ↪ 更改为 OFF 后，将会创建表。如下所示：

```
CREATE TABLE t1 (id int, c double(10));
ERROR 1149 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your MySQL server version for the right syntax to use
SET tidb_enable_strict_double_type_check = 'OFF';
Query OK, 0 rows affected (0.00 sec)
CREATE TABLE t1 (id int, c double(10));
Query OK, 0 rows affected (0.09 sec)
```

> 注意：
>
> 该设置仅适用于 DOUBLE 类型，因为 MySQL 允许为 FLOAT 类型指定精度。从 MySQL 8.0.17 开始已弃用此行为，不建议为 FLOAT 或 DOUBLE 类型指定精度。

8.1.92 tidb_enable_table_partition

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 可选值：OFF，ON，AUTO
- 这个变量用来设置是否开启 TABLE PARTITION 特性。目前变量支持以下三种值：
  - 默认值 ON 表示开启 TiDB 当前已实现了的分区表类型，目前 Range partition、 Hash partition 以及 Range column 单列的场景会生效。
  - AUTO 目前作用和 ON 一样。
  - OFF 表示关闭 TABLE PARTITION 特性，此时语法还是保持兼容，只是创建的表并不是真正的分区表，而是普通的表。

8.1.93 tidb_enable_telemetry 从 v4.0.2 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于动态地控制 TiDB 遥测功能是否开启。设置为 OFF 可以关闭 TiDB 遥测功能。当所有 TiDB 实例都设置enable-telemetry 为 false 时将忽略该系统变量并总是关闭 TiDB 遥测功能。参阅遥测了解该功能详情。

8.1.94 tidb_enable_top_sql 从 v5.4.0 版本开始引入

> **警告：**
>
> Top SQL 目前是实验性功能，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用控制是否开启Top SQL 特性。

8.1.95 tidb_enable_tso_follower_proxy 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来开启 TSO Follower Proxy 特性。当该值为 OFF 时，TiDB 仅会从 PD leader 获取 TSO。开启该特性之后，TiDB 在获取 TSO 时会将请求均匀地发送到所有 PD 节点上，通过 PD follower 转发 TSO 请求，从而降低 PD leader 的 CPU 压力。
- 适合开启 TSO Follower Proxy 的场景：

  - PD leader 因高压力的 TSO 请求而达到 CPU 瓶颈，导致 TSO RPC 请求的延迟较高。
  - 集群中的 TiDB 实例数量较多，且调高 `tidb_tso_client_batch_max_wait_time` 并不能缓解 TSO RPC 请求延迟高的问题。

> 注意：
>
> 如果 PD leader 的 TSO RPC 延迟升高，但其现象并非由 CPU 使用率达到瓶颈而导致（可能存在网络等问题），此时，打开 TSO Follower Proxy 可能会导致 TiDB 的语句执行延迟上升，从而影响集群的 QPS 表现。

8.1.96 `tidb_enable_vectorized_expression` 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启向量化执行。

8.1.97 `tidb_enable_window_function`

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启窗口函数的支持。默认值 1 代表开启窗口函数的功能。
- 由于窗口函数会使用一些保留关键字，可能导致原先可以正常执行的 SQL 语句在升级 TiDB 后无法被解析语法，此时可以将 `tidb_enable_window_function` 设置为 OFF。

8.1.98 `tidb_enforce_mpp` 从 v5.1 版本开始引入

- 作用域：SESSION
- 默认值：OFF（表示关闭）。如需修改此变量的默认值，请配置 `performance.enforce-mpp` 参数。
- 这个变量用于控制是否忽略优化器代价估算，强制使用 TiFlash 的 MPP 模式执行查询，可以设置的值包括：
  - 0 或 OFF，代表不强制使用 MPP 模式（默认）
  - 1 或 ON，代表将忽略代价估算，强制使用 MPP 模式。注意：只有当 `tidb_allow_mpp=true` 时该设置才生效。

MPP 是 TiFlash 引擎提供的分布式计算框架，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法。MPP 模式选择的详细说明参见控制是否选择 MPP 模式。

8.1.99 `tidb_evolve_plan_baselines` 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL

- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否启用自动演进绑定功能。该功能的详细介绍和使用方法可以参考自动演进绑定。
- 为了减少自动演进对集群的影响，可以进行以下配置：
  - 设置 tidb_evolve_plan_task_max_time，限制每个执行计划运行的最长时间，其默认值为 600s；
  - 设置tidb_evolve_plan_task_start_time 和 tidb_evolve_plan_task_end_time，限制运行演进任务的时间窗口，默认值分别为 00:00 +0000 和 23:59 +0000。

8.1.100 tidb_evolve_plan_task_end_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动演进的结束时间。

8.1.101 tidb_evolve_plan_task_max_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：600
- 范围：[-1, 9223372036854775807]
- 单位：秒
- 该变量用于限制自动演进功能中，每个执行计划运行的最长时间。

8.1.102 tidb_evolve_plan_task_start_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动演进的开始时间。

8.1.103 tidb_executor_concurrency 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：5
- 范围：[1, 256]

变量用来统一设置各个 SQL 算子的并发度，包括：

- index lookup
- index lookup join

- hash join
- hash aggregation（partial 和 final 阶段）
- window
- projection

tidb_executor_concurrency 整合了已有的系统变量，方便管理。这些变量所列如下：

- tidb_index_lookup_concurrency
- tidb_index_lookup_join_concurrency
- tidb_hash_join_concurrency
- tidb_hashagg_partial_concurrency
- tidb_hashagg_final_concurrency
- tidb_projection_concurrency
- tidb_window_concurrency

v5.0 后，用户仍可以单独修改以上系统变量（会有废弃警告），且修改只影响单个算子。后续通过 tidb_executor_concurrency 的修改也不会影响该算子。若要通过 tidb_executor_concurrency 来管理所有算子的并发度，需要将以上所列变量的值设置为 -1。

对于从 v5.0 之前的版本升级到 v5.0 的系统，如果用户对上述所列变量的值没有做过改动（即 tidb_hash_join_concurrency 值为 5，其他值为 4），则会自动转为使用 tidb_executor_concurrency 来统一管理算子并发度。如果用户对上述变量的值做过改动，则沿用之前的变量对相应的算子做并发控制。

8.1.104 tidb_expensive_query_time_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：60
- 范围：[10, 2147483647]
- 单位：秒
- 这个变量用来控制打印 expensive query 日志的阈值时间，默认值是 60 秒。expensive query 日志和慢日志的差别是，慢日志是在语句执行完后才打印，expensive query 日志可以把正在执行中的语句且执行时间超过阈值的语句及其相关信息打印出来。

8.1.105 tidb_force_priority

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：NO_PRIORITY
- 这个变量用于改变 TiDB server 上执行的语句的默认优先级。例如，你可以通过设置该变量来确保正在执行 OLAP 查询的用户优先级低于正在执行 OLTP 查询的用户。
- 可设置为 NO_PRIORITY、LOW_PRIORITY、DELAYED 或 HIGH_PRIORITY。

8.1.106 tidb_gc_concurrency 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 这个变量用于指定 GC 在Resolve Locks（清理锁）步骤中线程的数量。默认值 -1 表示由 TiDB 自主判断运行 GC 要使用的线程的数量。

8.1.107 tidb_gc_enable 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否启用 TiKV 的垃圾回收 (GC) 机制。如果不启用 GC 机制，系统将不再清理旧版本的数据，因此会有损系统性能。

8.1.108 tidb_gc_life_time 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：10m0s
- 范围：[10m0s, 8760h0m0s]
- 这个变量用于指定每次进行垃圾回收 (GC) 时保留数据的时限。变量值为 Go 的 Duration 字符串格式。每次进行 GC 时，将以当前时间减去该变量的值作为 safe point。

> Note:
>
> - 在数据频繁更新的场景下，将 tidb_gc_life_time 的值设置得过大（如数天甚至数月）可能会导致一些潜在的问题，如：
>   - 占用更多的存储空间。
>   - 大量的历史数据可能会在一定程度上影响系统性能，尤其是范围的查询（如 select count(*)from t）。
> - 如果一个事务的运行时长超过了 tidb_gc_life_time 配置的值，在 GC 时，为了使这个事务可以继续正常运行，系统会保留从这个事务开始时间 start_ts 以来的数据。例如，如果 tidb_gc_life_time 的值配置为 10 分钟，且在一次 GC 时，集群正在运行的事务中最早开始的那个事务已经运行了 15 分钟，那么本次 GC 将保留最近 15 分钟的数据。

8.1.109 tidb_gc_max_wait_time 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是

- 默认值：86400
- 范围：[600, 31536000]
- 单位：秒
- 这个变量用于指定活跃事务阻碍 GC safe point 推进的最大时间。每次进行 GC 时，默认 GC safe point 不会超过正在执行中的事务的开始时间。如果活跃事务运行时间未超过该值，GC safe point 会一直被阻塞不更新，直到活跃事务运行时间超过该值 safe point 才会正常推进。

8.1.110 tidb_gc_run_interval 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：10m0s
- 范围：[10m0s, 8760h0m0s]
- 这个变量用于指定垃圾回收 (GC) 运行的时间间隔。变量值为 Go 的 Duration 字符串格式，如"1h30m"、"15m"等。

8.1.111 tidb_gc_scan_lock_mode 从 v5.0 版本开始引入

> **警告：**
>
> Green GC 目前是实验性功能，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：LEGACY
- 可设置为：PHYSICAL，LEGACY
  - LEGACY：使用旧的扫描方式，即禁用 Green GC。
  - PHYSICAL：使用物理扫描方式，即启用 Green GC。
- 这个变量用于指定垃圾回收 (GC) 的 Resolve Locks（清理锁）步骤中扫描锁的方式。当变量值设置为 LEGACY 时，TiDB 以 Region 为单位进行扫描。当变量值设置为 PHYSICAL 时，每个 TiKV 节点分别绕过 Raft 层直接扫描数据，可以有效地缓解在启用Hibernate Region 功能时，GC 唤醒全部 Region 的影响，从而提升 Resolve Locks（清理锁）这个步骤的执行速度。

8.1.112 tidb_general_log

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：OFF
- 这个变量用来设置是否在日志里记录所有的 SQL 语句。该功能默认关闭。如果系统运维人员在定位问题过程中需要追踪所有 SQL 记录，可考虑开启该功能。
- 通过查询 "GENERAL_LOG" 字符串可以定位到该功能在日志中的所有记录。日志会记录以下内容：

- conn：当前会话对应的 ID
- user：当前会话用户
- schemaVersion：当前 schema 版本
- txnStartTS：当前事务的开始时间戳
- forUpdateTS：事务模式为悲观事务时，SQL 语句的当前时间戳。悲观事务内发生写冲突时，会重试当前执行语句，该时间戳会被更新。重试次数由max-retry-count 配置。事务模式为乐观事务时，该条目与 txnStartTS 等价。
- isReadConsistency：当前事务隔离级别是否是读已提交 (RC)
- current_db：当前数据库名
- txn_mode：事务模式。可选值：OPTIMISTIC（乐观事务模式），或 PESSIMISTIC（悲观事务模式）
- sql：当前查询对应的 SQL 语句

8.1.113 tidb_hash_join_concurrency

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 这个变量用来设置 hash join 算法的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.114 tidb_hashagg_final_concurrency

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 这个变量用来设置并行 hash aggregation 算法 final 阶段的执行并发度。对于聚合函数参数不为 distinct 的情况，HashAgg 分为 partial 和 final 阶段分别并行执行。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.115 tidb_hashagg_partial_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 这个变量用来设置并行 hash aggregation 算法 partial 阶段的执行并发度。对于聚合函数参数不为 distinct 的情况，HashAgg 分为 partial 和 final 阶段分别并行执行。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.116 tidb_ignore_prepared_cache_close_stmt（从 v6.0 版本开始引入）

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否忽略关闭 Prepared Statement 的指令。
- 如果变量值设为 ON，Binary 协议的 COM_STMT_CLOSE 信号和文本协议的DEALLOCATE PREPARE 语句都会被忽略。

8.1.117 tidb_index_join_batch_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：25000
- 范围：[1, 2147483647]
- 这个变量用来设置 index lookup join 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值。

8.1.118 tidb_index_lookup_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1

- 范围：[1，256]
- 这个变量用来设置 index lookup 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.119 tidb_index_lookup_join_concurrency

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1，256]
- 这个变量用来设置 index lookup join 算法的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.120 tidb_index_lookup_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：20000
- 范围：[1，2147483647]
- 这个变量用来设置 index lookup 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值。

8.1.121 tidb_index_serial_scan_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1，256]
- 这个变量用来设置顺序 scan 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。

8.1.122 tidb_init_chunk_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：32
- 范围：[1，32]
- 这个变量用来设置执行过程中初始 chunk 的行数。默认值是 32，可设置的范围是 1 ~ 32。

8.1.123 tidb_isolation_read_engines 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：tikv,tiflash,tidb
- 这个变量用于设置 TiDB 在读取数据时可以使用的存储引擎列表。

8.1.124 tidb_log_file_max_days 从 v5.3.0 版本开始引入

- 作用域：SESSION
- 默认值：0
- 范围：[0, 2147483647]
- 这个变量可以调整当前 TiDB 实例上日志的最大保留天数。默认值是实例配置文件中指定的值，见配置项max-days。此变量只影响当前 TiDB 实例上的配置，重启后丢失，且配置文件不受影响。

8.1.125 tidb_low_resolution_tso

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来设置是否启用低精度 TSO 特性。开启该功能之后，新事务会使用一个每 2s 更新一次的 TS 来读取数据。
- 主要场景是在可以容忍读到旧数据的情况下，降低小的只读事务获取 TSO 的开销。

8.1.126 tidb_nontransactional_ignore_error 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否在非事务语句中立刻返回错误。当设为 OFF 时，在碰到第一个报错的 batch 时，非事务 DML 语句即中止，取消其后的所有 batch，返回错误。当设为 ON 时，当某个 batch 执行报错时，其后的 batch 会继续执行，直到所有 batch 执行完毕，返回结果时把这些错误合并后返回。

8.1.127 tidb_max_chunk_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1024
- 范围：[32, 2147483647]
- 这个变量用来设置执行过程中一个 chunk 最大的行数，设置过大可能引起缓存局部性的问题。

8.1.128 tidb_max_delta_schema_count

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1024

- 范围：[100，16384]
- 这个变量用来设置缓存 schema 版本信息（对应版本修改的相关 table IDs）的个数限制，可设置的范围 100 - 16384。此变量在 2.1.18 及之后版本支持。

8.1.129 tidb_mem_oom_action 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：CANCEL
- 可选值：CANCEL，LOG
- 该变量控制当单个查询使用的内存超过限制 (tidb_mem_quota_query) 且不能再利用临时磁盘时，TiDB 所采取的操作。详情见TiDB 内存控制。
- 该变量默认值为 CANCEL，但在 TiDB v4.0.2 及之前的版本中，默认值为 LOG。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (oom-action) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.130 tidb_mem_quota_analyze 从 v6.1.0 版本开始引入

> **警告：**
>
> 目前限制 ANALYZE 的内存使用量为实验特性，在生产环境中使用时可能存在内存统计有误差的情况。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 单位：字节
- 这个变量用来控制 TiDB 更新统计信息时的最大总内存占用，包括用户执行的ANALYZE TABLE 和 TiDB 后台自动执行的统计信息更新任务。当总的内存占用超过这个阈值时，用户执行的 ANALYZE 会被终止退出，并通过错误信息提示用户尝试更小的采样率或稍后重试。如果 TiDB 后台自动执行的统计信息更新任务因内存超限而退出，且使用的采样率高于默认值，则会使用默认采样率重试一次。当该变量值为负数或零时，TiDB 不对更新统计信息的前后台任务进行内存限制。

> **注意：**
>
> 只有在 TiDB 的启动配置文件中开启了 run-auto-analyze 选项，该 TiDB 集群才会触发 auto_analyze。

8.1.131 tidb_mem_quota_apply_cache 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL

- 是否持久化到集群：是
- 默认值：33554432 (32 MiB)
- 范围：[0, 9223372036854775807]
- 单位：字节
- 这个变量用来设置 Apply 算子中局部 Cache 的内存使用阈值。
- Apply 算子中局部 Cache 用来加速 Apply 算子的计算，该变量可以设置 Apply Cache 的内存使用阈值。设置变量值为 0 可以关闭 Apply Cache 功能。

8.1.132 tidb_mem_quota_binding_cache（从 v6.0 版本开始引入）

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：67108864 (64 MiB)
- 范围：[0, 2147483647]
- 单位：字节
- 这个变量用来设置存放 binding 的缓存的内存使用阈值。
- 如果一个系统创建或者捕获了过多的绑定，导致绑定所使用的内存空间超过该阈值，TiDB 会在日志中增加警告日志进行提示。这种情况下，缓存无法存放所有可用的绑定，并且无法保证哪些绑定存在于缓存中，因此，可能存在一些查询无法使用可用绑定的情况。此时，可以调大该变量的值，从而保证所有可用绑定都能正常使用。修改变量值以后，需要执行命令 admin reload bindings 重新加载绑定，确保变更生效。

8.1.133 tidb_mem_quota_query

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1073741824 (1 GiB)
- 范围：[-1, 9223372036854775807]
- 单位：字节
- 这个变量用来设置一条查询语句的内存使用阈值。
- 如果一条查询语句执行过程中使用的内存空间超过该阈值，会触发系统变量tidb_mem_oom_action 中指定的行为。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (mem-quota-query) 进行配置，且作用域为 SESSION。升级到 v6.1.0 时会自动继承原有设置，作用域变更为 SESSION | GLOBAL。

8.1.134 tidb_memory_usage_alarm_ratio

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：0.8
- TiDB 内存使用占总内存的比例超过一定阈值时会报警。该功能的详细介绍和使用方法可以参考memory-usage-alarm-ratio。
- 该变量的初始值可通过memory-usage-alarm-ratio 进行配置。

8.1.135 tidb_metric_query_range_duration 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：60
- 范围：[10, 216000]
- 单位：秒
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 range duration。

8.1.136 tidb_metric_query_step 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：60
- 范围：[10, 216000]
- 单位：秒
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 step。

8.1.137 tidb_multi_statement_mode 从 v4.0.11 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 可选值：OFF，ON，WARN
- 该变量用于控制是否在同一个 COM_QUERY 调用中执行多个查询。
- 为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。该变量可用作早期 TiDB 版本的升级路径选项。该变量值与是否允许多语句行为的对照表如下：

客户端设置	tidb_multi_statement_mode 值	是否允许多语句
Multiple Statements = ON	OFF	允许
Multiple Statements = ON	ON	允许
Multiple Statements = ON	WARN	允许
Multiple Statements = OFF	OFF	不允许
Multiple Statements = OFF	ON	允许
Multiple Statements = OFF	WARN	允许 + 警告提示

> 注意：
>
> 只有默认值 OFF 才是安全的。如果用户业务是专为早期 TiDB 版本而设计的，那么需要将该变量值设为 ON。如果用户业务需要多语句支持，建议用户使用客户端提供的设置，不要使用 tidb_multi_statement_mode 变量进行设置。

- go-sql-driver (multiStatements)
- Connector/J (allowMultiQueries)
- PHP mysqli (mysqli_multi_query)

8.1.138 tidb_enable_new_only_full_group_by_check 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 可选值：OFF，ON
- 该变量用于控制用于控制 TiDB 执行 ONLY_FULL_GOUP_BY 检查时的行为。有关 ONLY_FULL_GROUP_BY 的信息可以参考 MySQL 文档。在 v6.1 中 TiDB 对该项检查做了更严格正确的处理。
- 由于可能存在版本升级造成的兼容性问题，在 v6.1 中该变量默认值是 OFF，即默认关闭。

8.1.139 tidb_opt_agg_push_down

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来设置优化器是否执行聚合函数下推到 Join，Projection 和 UnionAll 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 ON。

8.1.140 tidb_opt_correlation_exp_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[0, 2147483647]
- 当交叉估算方法不可用时，会采用启发式估算方法。这个变量用来控制启发式方法的行为。当值为 0 时不用启发式估算方法，大于 0 时，该变量值越大，启发式估算方法越倾向 index scan，越小越倾向 table scan。

8.1.141 tidb_opt_correlation_threshold

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0.9
- 这个变量用来设置优化器启用交叉估算 row count 方法的阈值。如果列和 handle 列之间的顺序相关性超过这个阈值，就会启用交叉估算方法。
- 交叉估算方法可以简单理解为，利用这个列的直方图来估算 handle 列需要扫的行数。

8.1.142 tidb_opt_distinct_agg_push_down

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来设置优化器是否执行带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。当查询中带有 Distinct 的聚合操作执行很慢时，可以尝试设置该变量为 1。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从 TiKV 读取所有数据，并在 TiDB 侧执行 distinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增里一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                          |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_6             | 1.00     | root      |               | funcs:count(distinct test.t.a)
    ↪ ->Column#4 |
| └─TableReader_10        | 10000.00 | root      |               | data:TableFullScan_9
    ↪                       |
|   └─TableFullScan_9     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc select count(distinct a) from test.t;
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                           |
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashAgg_8               | 1.00     | root      |               | funcs:count(distinct test.t.
    ↪ a)->Column#3 |
```

```
| └─TableReader_9           | 1.00     | root      |               | data:HashAgg_5
    ↪                            |
|   └─HashAgg_5             | 1.00     | cop[tikv] |               | group by:test.t.a,
    ↪                             |
|     └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
4 rows in set (0.00 sec)
```

8.1.143 tidb_opt_enable_correlation_adjustment

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制优化器是否开启交叉估算。

8.1.144 tidb_opt_insubq_to_join_and_agg

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来设置是否开启优化规则：将子查询转成 join 和 aggregation。

  例如，打开这个优化规则后，会将下面子查询做如下变化：

  ```
  select * from t where t.a in (select aa from t1);
  ```

  将子查询转成如下 join：

  ```
  select t.* from t, (select aa from t1 group by aa) tmp_t where t.a = tmp_t.aa;
  ```

  如果 t1 在列 aa 上有 unique 且 not null 的限制，可以直接改写为如下，不需要添加 aggregation。

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

8.1.145 tidb_opt_limit_push_down_threshold

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[0, 2147483647]
- 这个变量用来设置将 Limit 和 TopN 算子下推到 TiKV 的阈值。
- 如果 Limit 或者 TopN 的取值小于等于这个阈值，则 Limit 和 TopN 算子会被强制下推到 TiKV。该变量可以解决部分由于估算误差导致 Limit 或者 TopN 无法被下推的问题。

8.1.146 tidb_opt_prefer_range_scan 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 将该变量值设为 ON 后，优化器总是偏好区间扫描而不是全表扫描。
- 在以下示例中，tidb_opt_prefer_range_scan 开启前，TiDB 优化器需要执行全表扫描。tidb_opt_prefer_range_scan 开启后，优化器选择了索引区间扫描。

```
explain select * from t where age=5;
+-------------------------+------------+-----------+---------------+-------------------+
| id                      | estRows    | task      | access object | operator info     |
+-------------------------+------------+-----------+---------------+-------------------+
| TableReader_7           | 1048576.00 | root      |               | data:Selection_6  |
| └─Selection_6           | 1048576.00 | cop[tikv] |               | eq(test.t.age, 5) |
|   └─TableFullScan_5     | 1048576.00 | cop[tikv] | table:t       | keep order:false  |
+-------------------------+------------+-----------+---------------+-------------------+
3 rows in set (0.00 sec)

set session tidb_opt_prefer_range_scan = 1;

explain select * from t where age=5;
+--
    ↪ ---------------------------+------------+-----------+----------------------------+-------------------
    ↪
| id                           | estRows    | task      | access object              | operator
    ↪  info                 |
+--
    ↪ ---------------------------+------------+-----------+----------------------------+-------------------
    ↪
| IndexLookUp_7                | 1048576.00 | root      |                            |
    ↪                              |
| ├─IndexRangeScan_5(Build)    | 1048576.00 | cop[tikv] | table:t, index:idx_age(age) | range
    ↪ :[5,5], keep order:false |
| └─TableRowIDScan_6(Probe)    | 1048576.00 | cop[tikv] | table:t                    | keep
    ↪ order:false              |
+--
    ↪ ---------------------------+------------+-----------+----------------------------+-------------------
    ↪
3 rows in set (0.00 sec)
```

8.1.147 tidb_opt_write_row_id

- 作用域：SESSION

- 默认值：OFF
- 这个变量用来设置是否允许 INSERT、REPLACE 和 UPDATE 操作 _tidb_rowid 列，默认是不允许操作。该选项仅用于 TiDB 工具导数据时使用。

8.1.148 tidb_partition_prune_mode 从 v5.1 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：static
- 这个变量用来设置是否开启分区表动态裁剪模式。关于动态裁剪模式的详细说明请参阅分区表动态裁剪模式。

8.1.149 tidb_persist_analyze_options 从 v5.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启ANALYZE 配置持久化特性。

8.1.150 tidb_placement_mode（从 v6.0.0 版本开始引入）

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：STRICT
- 可选值：STRICT，IGNORE
- 该变量用于控制 DDL 语句是否忽略Placement Rules in SQL 指定的放置规则。变量值为 IGNORE 时将忽略所有放置规则选项。
- 该变量可由逻辑转储或逻辑恢复工具使用，确保即使绑定了不合适的放置规则，也始终可以成功创建表。这类似于 mysqldump 将 SET FOREIGN_KEY_CHECKS=0; 写入每个转储文件的开头部分。

8.1.151 tidb_pprof_sql_cpu 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：0
- 范围：[0, 1]
- 这个变量用来控制是否在 profile 输出中标记出对应的 SQL 语句，用于定位和排查性能问题。

8.1.152 tidb_prepared_plan_cache_memory_guard_ratio 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.1
- 范围：[0, 1]

- 这个变量用来控制 Prepared Plan Cache 触发内存保护机制的阈值，具体可见Prepared Plan Cache 的内存管理。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (prepared-plan-cache.memory-guard-ratio) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.153 tidb_prepared_plan_cache_size 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[1, 100000]
- 这个变量用来控制单个 SESSION 的 Prepared Plan Cache 最多能够缓存的计划数量，具体可见Prepared Plan Cache 的内存管理。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (prepared-plan-cache.capacity) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

8.1.154 tidb_projection_concurrency

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[-1, 256]
- 这个变量用来设置 Projection 算子的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.155 tidb_query_log_max_len

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4096 (4 KiB)
- 范围：[0, 1073741824]
- 单位：字节
- 该变量控制 SQL 语句输出的最大长度。当一条 SQL 语句的输出长度大于 tidb_query_log_max_len 时，输出将会被截断。
- 在 v6.1.0 之前这个开关也可以通过 TiDB 配置文件 (log.query-log-max-len) 进行配置，升级到 v6.1.0 后仅可通过系统变量配置。

示例：

```
SET tidb_query_log_max_len = 20;
```

8.1.156 tidb_rc_read_check_ts（从 v6.0.0 版本开始引入）

> **警告：**
>
> - 该特性与 replica-read 尚不兼容，开启 tidb_rc_read_check_ts 的读请求无法使用 replica-read，请勿同时开启两项特性。
> - 如果客户端使用游标操作，建议不开启 tidb_rc_read_check_ts 这一特性，避免前一批返回数据已经被客户端使用而语句最终会报错的情况。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量用于优化时间戳的获取，适用于悲观事务 READ-COMMITTED 隔离级别下读写冲突较少的场景，开启此变量可以避免获取全局 timestamp 带来的延迟和开销，并优化事务内读语句延迟。
- 如果读写冲突较为严重，开启此功能会增加额外开销和延迟，造成性能回退。更详细的说明，请参考读已提交隔离级别 (Read Committed) 文档。

8.1.157 tidb_read_staleness 从 v5.4.0 版本开始引入

- 作用域：SESSION
- 默认值：0
- 范围 [-2147483648, 0]
- 这个变量用于设置当前会话允许读取的历史数据范围。设置后，TiDB 会从参数允许的范围内选出一个尽可能新的时间戳，并影响后继的所有读操作。比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证 TiKV 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

8.1.158 tidb_record_plan_in_slow_log

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：ON
- 这个变量用于控制是否在 slow log 里包含慢查询的执行计划。

8.1.159 tidb_redact_log

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF

- 这个变量用于控制在记录 TiDB 日志和慢日志时，是否将 SQL 中的用户信息遮蔽。
- 将该变量设置为 1 即开启后，假设执行的 SQL 为 `insert into t values (1,2)`，在日志中记录的 SQL 会是 `insert into t values (?,?)`，即用户输入的信息被遮蔽。

8.1.160 tidb_regard_null_as_point 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制优化器是否可以将包含 null 的等值条件作为前缀条件来访问索引。
- 该变量默认开启。开启后，该变量可以使优化器减少需要访问的索引数据量，从而提高查询的执行速度。例如，在有多列索引 `index(a, b)` 且查询条件为 `a<=>null and b=1` 的情况下，优化器可以同时使用查询条件中的 `a<=>null` 和 `b=1` 进行索引访问。如果关闭该变量，因为 `a<=>null and b=1` 包含 null 的等值条件，优化器不会使用 `b=1` 进行索引访问。

8.1.161 tidb_replica_read 从 v4.0 版本开始引入

- 作用域：SESSION
- 是否持久化到集群：是
- 默认值：`leader`
- 可选值：`leader`，`follower`，`leader-and-follower`
- 这个变量用于控制 TiDB 读取数据的位置，有以下三个选择：
  - leader：只从 leader 节点读取
  - follower：只从 follower 节点读取
  - leader-and-follower：从 leader 或 follower 节点读取

更多细节，见Follower Read。

8.1.162 tidb_retry_limit

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：10
- 范围：`[-1, 9223372036854775807]`
- 这个变量用来设置乐观事务的最大重试次数。一个事务执行中遇到可重试的错误（例如事务冲突、事务提交过慢或表结构变更）时，会根据该变量的设置进行重试。注意当 `tidb_retry_limit = 0` 时，也会禁用自动重试。该变量仅适用于乐观事务，不适用于悲观事务。

8.1.163 tidb_row_format_version

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：2
- 范围：[1, 2]
- 控制新保存数据的表数据格式版本。TiDB v4.0 中默认使用版本号为 2 的新表数据格式保存新数据。
- 但如果从 4.0.0 之前的版本升级到 4.0.0，不会改变表数据格式版本，TiDB 会继续使用版本为 1 的旧格式写入表中，即只有新创建的集群才会默认使用新表数据格式。
- 需要注意的是修改该变量不会对已保存的老数据产生影响，只会对修改变量后的新写入数据使用对应版本格式保存。

8.1.164 tidb_scatter_region

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- TiDB 默认会在建表时为新表分裂 Region。开启该变量后，会在建表语句执行时，同步打散刚分裂出的 Region。适用于批量建表后紧接着批量写入数据，能让刚分裂出的 Region 先在 TiKV 分散而不用等待 PD 进行调度。为了保证后续批量写入数据的稳定性，建表语句会等待打散 Region 完成后再返回建表成功，建表语句执行时间会是该变量关闭时的数倍。
- 如果建表时设置了 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS，建表成功后会均匀切分出指定数量的 Region。

8.1.165 tidb_skip_ascii_check 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否校验 ASCII 字符的合法性。
- 校验 ASCII 字符会损耗些许性能。当你确认输入的字符串为有效的 ASCII 字符时，可以将其设置为 ON。

8.1.166 tidb_skip_isolation_level_check

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 开启这个开关之后，如果对 tx_isolation 赋值一个 TiDB 不支持的隔离级别，不会报错，有助于兼容其他设置了（但不依赖于）不同隔离级别的应用。

```
tidb> set tx_isolation='serializable';
ERROR 8048 (HY000): The isolation level 'serializable' is not supported. Set
    ↪ tidb_skip_isolation_level_check=1 to skip this error
tidb> set tidb_skip_isolation_level_check=1;
Query OK, 0 rows affected (0.00 sec)

tidb> set tx_isolation='serializable';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

8.1.167 tidb_skip_utf8_check

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否校验 UTF-8 字符的合法性。
- 校验 UTF-8 字符会损耗些许性能。当你确认输入的字符串为有效的 UTF-8 字符时，可以将其设置为 ON。

8.1.168 tidb_slow_log_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否
- 默认值：300
- 范围：[-1, 9223372036854775807]
- 单位：毫秒
- 输出慢日志的耗时阈值。当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。默认为 300 ms。

示例：

```
set tidb_slow_log_threshold = 200;
```

8.1.169 tidb_max_tiflash_threads 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[-1, 256]
- TiFlash 中 request 执行的最大并发度。默认值为 -1，表示该系统变量无效。0 表示由 TiFlash 系统自动设置该值。

8.1.170 tidb_slow_query_file

- 作用域：SESSION

- 默认值：“”
- 查询 INFORMATION_SCHEMA.SLOW_QUERY 只会解析配置文件中 slow-query-file 设置的慢日志文件名，默认是“tidb-slow.log”。但如果想要解析其他的日志文件，可以通过设置 session 变量 tidb_slow_query_file 为具体的文件路径，然后查询 INFORMATION_SCHEMA.SLOW_QUERY 就会按照设置的路径去解析慢日志文件。更多详情可以参考 SLOW_QUERY 文档。

8.1.171 tidb_snapshot

- 作用域：SESSION
- 默认值：“”
- 这个变量用来设置当前会话期待读取的历史数据所处时刻。比如当设置为 "2017-11-11 20:20:20" 时或者一个 TSO 数字“400036290571534337”，当前会话将能读取到该时刻的数据。

8.1.172 tidb_stats_cache_mem_quota 从 v6.1.0 版本开始引入

> **警告：**
>
> 该变量为实验特性，不推荐在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 这个变量用于控制 TiDB 内部统计信息缓存使用内存的上限。

8.1.173 tidb_stats_load_sync_wait 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载目前为实验性特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 单位：毫秒
- 范围：[0, 2147483647]
- 这个变量用于控制是否开启统计信息的同步加载模式（默认为 0 代表不开启，即为异步加载模式），以及开启的情况下，SQL 执行同步加载完整统计信息等待多久后会超时。更多信息，请参考统计信息的加载。

8.1.174 tidb_stats_load_pseudo_timeout 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载目前为实验性特性，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制统计信息同步加载超时后，SQL 是执行失败（OFF），还是退回使用 pseudo 的统计信息（ON）。

8.1.175 tidb_stmt_summary_history_size 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：24
- 范围：[0, 255]
- 这个变量设置了 statement summary tables 的历史记录容量。

8.1.176 tidb_stmt_summary_internal_query 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来控制是否在 statement summary tables 中包含 TiDB 内部 SQL 的信息。

8.1.177 tidb_stmt_summary_max_sql_length 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4096
- 范围：[0, 2147483647]
- 这个变量控制 statement summary tables 显示的 SQL 字符串长度。

8.1.178 tidb_stmt_summary_max_stmt_count 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：3000
- 范围：[1, 32767]
- 这个变量设置了 statement summary tables 在内存中保存的语句的最大数量。

8.1.179 tidb_stmt_summary_refresh_interval 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1800
- 范围：[1, 2147483647]
- 单位：秒
- 这个变量设置了 statement summary tables 的刷新时间。

8.1.180 tidb_top_sql_max_meta_count 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：5000
- 范围：[1, 10000]
- 这个变量用于控制Top SQL 每分钟最多收集 SQL 语句类型的数量。

8.1.181 tidb_top_sql_max_time_series_count 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[1, 5000]
- 这个变量用于控制Top SQL 每分钟保留消耗负载最大的前多少条 SQL（即 Top N) 的数据。

> **注意：**
>
> TiDB Dashboard 中的 Top SQL 页面目前只显示消耗负载最多的 5 类 SQL 查询，这与 tidb_top_sql_max_time_series_count 的配置无关。

8.1.182 tidb_store_limit 从 v3.0.4 和 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 9223372036854775807]
- 这个变量用于限制 TiDB 同时向 TiKV 发送的请求的最大数量，0 表示没有限制。

8.1.183 tidb_sysdate_is_now（从 v6.0.0 版本开始引入）

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 默认值：OFF
- 这个变量用于控制 SYSDATE 函数能否替换为 NOW 函数，其效果与 MYSQL 中的 sysdate-is-now 一致。

8.1.184 tidb_table_cache_lease（从 v6.0.0 版本开始引入）

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：3
- 范围：[1, 10]
- 单位：秒
- 这个变量用来控制缓存表的 lease 时间，默认值是 3 秒。该变量值的大小会影响缓存表的修改。在缓存表上执行修改操作后，最长可能出现 tidb_table_cache_lease 变量值时长的等待。如果业务表为只读表，或者能接受很高的写入延迟，则可以将该变量值调大，从而增加缓存的有效时间，减少 lease 续租的频率。

8.1.185 tidb_tmp_table_max_size 从 v5.3 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：67108864
- 范围：[1048576, 137438953472]
- 单位：字节
- 这个变量用于限制单个临时表的最大大小，临时表超出该大小后报错。

8.1.186 tidb_tso_client_batch_max_wait_time 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 10]
- 单位：毫秒
- 这个变量用来设置 TiDB 向 PD 请求 TSO 时进行一次攒批操作的最大等待时长。默认值为 0，即不进行额外的等待。
- 在向 PD 获取 TSO 请求时，TiDB 使用的 PD Client 会一次尽可能多地收集同一时刻的 TSO 请求，将其攒批合并成一个 RPC 请求后再发送给 PD，从而减轻 PD 的压力。
- 将这个变量值设置为非 0 后，TiDB 会在每一次攒批结束前进行一个最大时长为其值的等待，目的是为了收集到更多的 TSO 请求，从而提高攒批效果。
- 适合调高这个变量值的场景：
  - PD leader 因高压力的 TSO 请求而达到 CPU 瓶颈，导致 TSO RPC 请求的延迟较高。
  - 集群中 TiDB 实例的数量不多，但每一台 TiDB 实例上的并发量较高。
- 在实际使用中，推荐将该变量尽可能设置为一个较小的值。

> 注意：
>
> 如果 PD leader 的 TSO RPC 延迟升高，但其现象并非由 CPU 使用率达到瓶颈而导致（可能存在网络等问题），此时，调高 tidb_tso_client_batch_max_wait_time 可能会导致 TiDB 的语句执行延迟上升，影响集群的 QPS 表现。

8.1.187 tidb_txn_assertion_level（从 v6.0.0 版本开始引入）

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：FAST
- 可选值：OFF，FAST，STRICT
- 这个变量用于设置 assertion 级别。assertion 是一项在事务提交过程中进行的数据索引一致性校验，它对正在写入的 key 是否存在进行检查。如果不符则说明数据索引不一致，会导致事务 abort。详见数据索引一致性报错。
- 对于新创建的 v6.0.0 及以上的集群，默认值为 FAST。对于升级版本的集群，如果升级前是低于 v6.0.0 的版本，升级后默认值为 OFF。
  - OFF: 关闭该检查。
  - FAST: 开启大多数检查项，对性能几乎无影响。
  - STRICT: 开启全部检查项，当系统负载较高时，对悲观事务的性能有较小影响。

8.1.188 tidb_txn_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：pessimistic
- 可选值：pessimistic，optimistic
- 这个变量用于设置事务模式。TiDB v3.0 支持了悲观事务，自 v3.0.8 开始，默认使用悲观事务模式。
- 但如果从 3.0.7 及之前的版本升级到 >= 3.0.8 的版本，不会改变默认事务模式，即只有新创建的集群才会默认使用悲观事务模式。
- 将该变量设置为“optimistic”或“”时，将会使用乐观事务模式。

8.1.189 tidb_use_plan_baselines（从 v4.0 版本开始引入）

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启执行计划绑定功能，默认打开，可通过赋值 OFF 来关闭。关于执行计划绑定功能的使用可以参考执行计划绑定文档。

8.1.190 tidb_wait_split_region_finish

- 作用域：SESSION
- 默认值：ON
- 由于打散 Region 的时间可能比较长，主要由 PD 调度以及 TiKV 的负载情况所决定。这个变量用来设置在执行 SPLIT REGION 语句时，是否同步等待所有 Region 都打散完成后再返回结果给客户端。
  - 默认 ON 代表等待打散完成后再返回结果
  - OFF 代表不等待 Region 打散完成就返回。
- 需要注意的是，在 Region 打散期间，对正在打散 Region 上的写入和读取的性能会有一定影响，对于批量写入、导数据等场景，还是建议等待 Region 打散完成后再开始导数据。

8.1.191 tidb_wait_split_region_timeout

- 作用域：SESSION
- 默认值：300
- 范围：[1, 2147483647]
- 单位：秒
- 这个变量用来设置 SPLIT REGION 语句的执行超时时间，默认值是 300 秒，如果超时还未完成，就返回一个超时错误。

8.1.192 tidb_window_concurrency 从 v4.0 版本开始引入

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 这个变量用于设置 window 算子的并行度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

8.1.193 time_zone

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：SYSTEM
- 数据库所使用的时区。这个变量值可以写成时区偏移的形式，如 '-8:00'，也可以写成一个命名时区，如 'America/Los_Angeles'。
- 默认值 SYSTEM 表示时区应当与系统主机的时区相同。系统的时区可通过system_time_zone 获取。

8.1.194 timestamp

- 作用域：SESSION
- 默认值：0
- 一个 Unix 时间戳。变量值非空时，表示 CURRENT_TIMESTAMP()、NOW() 等函数的时间戳。该变量通常用于数据恢复或数据复制。

8.1.195 transaction_isolation

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：REPEATABLE-READ
- 可选值：READ-UNCOMMITTED，READ-COMMITTED，REPEATABLE-READ，SERIALIZABLE
- 这个变量用于设置事务隔离级别。TiDB 为了兼容 MySQL，支持可重复读 (REPEATABLE-READ)，但实际的隔离级别是快照隔离。详情见事务隔离级别。

8.1.196 tx_isolation

这个变量是 transaction_isolation 的别名。

8.1.197 version

- 作用域：NONE
- 默认值：5.7.25-TiDB-(tidb version)
- 这个变量的值是 MySQL 的版本和 TiDB 的版本，例如 ‘5.7.25-TiDB-v4.0.0-beta.2-716-g25e003253’。

8.1.198 version_comment

- 作用域：NONE
- 默认值：(string)
- 这个变量的值是 TiDB 版本号的其他信息，例如 ‘TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7 compatible’。

8.1.199 version_compile_os

- 作用域：NONE
- 默认值：(string)
- 这个变量值是 TiDB 所在操作系统的名称。

8.1.200 version_compile_machine

- 作用域：NONE
- 默认值：(string)
- 这个变量值是运行 TiDB 的 CPU 架构的名称。

8.1.201 wait_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：28800
- 范围：[0, 31536000]
- 单位：秒
- 这个变量表示用户会话的空闲超时。0 代表没有时间限制。

8.1.202 warning_count

- 作用域：SESSION
- 默认值：0
- 这个只读变量表示之前执行语句中出现的警告数。

8.1.203 windowing_use_high_precision

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制计算窗口函数时是否采用高精度模式。

9 Placement Rules 使用文档

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

Placement Rules 是 PD 在 4.0 版本引入的一套副本规则系统，用于指导 PD 针对不同类型的数据生成对应的调度。通过组合不同的调度规则，用户可以精细地控制任何一段连续数据的副本数量、存放位置、主机类型、是否参与 Raft 投票、是否可以担任 Raft leader 等属性。

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

9.1 规则系统介绍

整个规则系统的配置由多条规则即 Rule 组成。每条 Rule 可以指定不同的副本数量、Raft 角色、放置位置等属性，以及这条规则生效的 key range。PD 在进行调度时，会先根据 Region 的 key range 在规则系统中查到该 Region 对应的规则，然后再生成对应的调度，来使得 Region 副本的分布情况符合 Rule。

多条规则的 key range 可以有重叠部分的，即一个 Region 能匹配到多条规则。这种情况下 PD 根据 Rule 的属性来决定规则是相互覆盖还是同时生效。如果有多条规则同时生效，PD 会按照规则的堆叠次序依次去生成调度进行规则匹配。

此外，为了满足不同来源的规则相互隔离的需求，支持更灵活的方式来组织规则，还引入了分组 (Group) 的概念。通常情况下，用户可根据规则的不同来源把规则放置在不同的 Group。

Placement Rules 示意图如下所示：

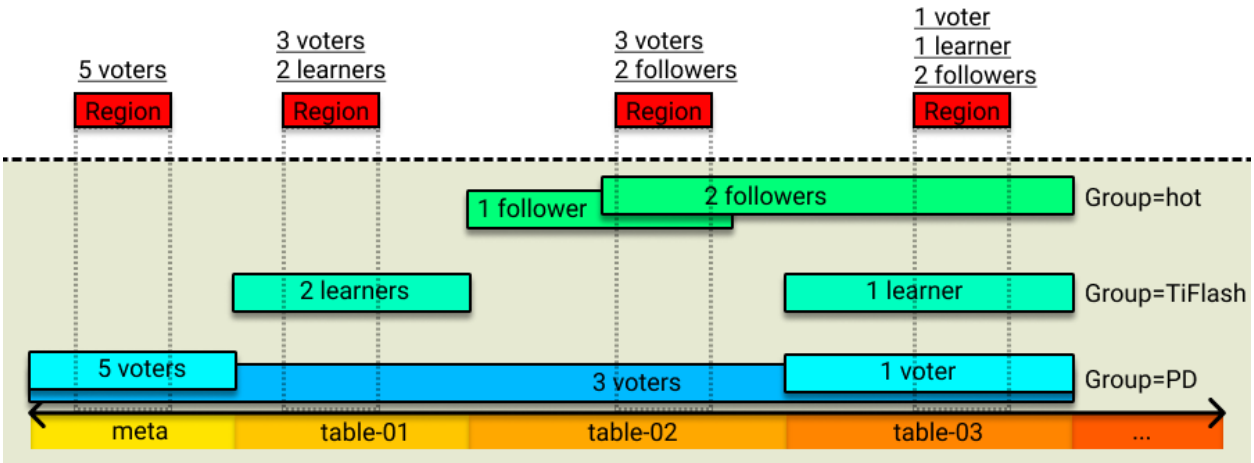


图 142: Placement rules overview

9.1.1 规则字段

以下是每条规则中各个字段的具体含义：

字段名	类型及约束	说明
GroupID	string	分组 ID，标识规则的来源
ID	string	分组内唯一 ID
Index	int	分组内堆叠次序
Override	true/false	是否覆盖 index 的更小 Rule（限分组内）
StartKey	string，十六进制编码	适用 Range 起始 key
EndKey	string，十六进制编码	适用 Range 终止 key
Role	string	副本角色，包括 leader/follower/learner
Count	int，正整数	副本数量
LabelConstraint	[]Constraint	用于按 label 筛选节点
LocationLabels	[]string	用于物理隔离
IsolationLevel	string	用于设置最小强制物理隔离级别

LabelConstraint 与 Kubernetes 中的功能类似，支持通过 in、notIn、exists 和 notExists 四种原语来筛选 label。这四种原语的意义如下：

- `in`：给定 key 的 label value 包含在给定列表中。
- `notIn`：给定 key 的 label value 不包含在给定列表中。
- `exists`：包含给定的 label key。
- `notExists`：不包含给定的 label key。

`LocationLabels` 的意义和作用与 PD v4.0 之前的版本相同。比如配置 `[zone,rack,host]` 定义了三层的拓扑结构：集群分为多个 zone（可用区），每个 zone 下有多个 rack（机架），每个 rack 下有多个 host（主机）。PD 在调度时首先会尝试将 Region 的 Peer 放置在不同的 zone，假如无法满足（比如配置 3 副本但总共只有 2 个 zone）则保证放置在不同的 rack；假如 rack 的数量也不足以保证隔离，那么再尝试 host 级别的隔离，以此类推。

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

9.1.2 规则分组字段

以下是规则分组字段的含义：

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

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

9.2 配置规则操作步骤

本节的操作步骤以使用pd-ctl 工具为例，涉及到的命令也支持通过 HTTP API 进行调用。

9.2.1 开启 Placement Rules 特性

Placement Rules 特性在 TiDB v5.0 及以上的版本中默认开启。如需关闭 Placement Rules 特性，请参考关闭 Placement Rules。如需在关闭后重新开启该特性，可以集群初始化以前设置 PD 配置文件：

```
[replication]
enable-placement-rules = true
```

这样，PD 在初始化成功后会开启这个特性，并根据 `max-replicas` 及 `location-labels` 配置生成对应的规则：

```
{
  "group_id": "pd",
  "id": "default",
  "start_key": "",
```

```
  "end_key": "",
  "role": "voter",
  "count": 3,
  "location_labels": ["zone", "rack", "host"],
  "isolation_level": ""
}
```

如果是已经初始化过的集群，也可以通过 pd-ctl 进行在线开启：

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

PD 同样将根据系统的 `max-replicas` 及 `location-labels` 生成默认的规则。

> 注意：
>
> 开启 Placement Rules 后，原先的 `max-replicas` 及 `location-labels` 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

9.2.2 关闭 Placement Rules 特性

使用 pd-ctl 可以关闭 Placement Rules 特性，切换为之前的调度策略。

```
pd-ctl config placement-rules disable
```

> 注意：
>
> 关闭 Placement Rules 后，PD 将使用原先的 `max-replicas` 及 `location-labels` 配置。在 Placement Rules 开启期间对 Rule 的修改不会导致这两项配置的同步更新。此外，设置好的所有 Rule 都会保留在系统中，会在下次开启 Placement Rules 时被使用。

9.2.3 使用 pd-ctl 设置规则

> 注意：
>
> 规则的变更将实时地影响 PD 调度，不恰当的规则设置可能导致副本数较少，影响系统的高可用。

pd-ctl 支持使用多种方式查看系统中的 Rule，输出是 json 格式的 Rule 或 Rule 列表：

- 查看所有规则列表

```
pd-ctl config placement-rules show
```

- 查看 PD Group 的所有规则列表

```
pd-ctl config placement-rules show --group=pd
```

- 查看对应 Group 和 ID 的某条规则

```
pd-ctl config placement-rules show --group=pd --id=default
```

- 查看 Region 所匹配的规则列表

```
pd-ctl config placement-rules show --region=2
```

  上面的例子中 2 为 Region ID。

新增和编辑规则是类似的，需要把对应的规则写进文件，然后使用 save 命令保存至 PD：

```
cat > rules.json <<EOF
[
    {
        "group_id": "pd",
        "id": "rule1",
        "role": "voter",
        "count": 3,
        "location_labels": ["zone", "rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "rule2",
        "role": "voter",
        "count": 2,
        "location_labels": ["zone", "rack", "host"]
    }
]
EOF
pd-ctl config placement save --in=rules.json
```

以上操作会将 rule1、rule2 两条规则写入 PD，如果系统中已经存在 GroupID+ID 相同的规则，则会覆盖该规则。

如果需要删除某条规则，只需要将规则的 count 置为 0 即可，对应 GroupID+ID 相同的规则会被删除。以下命令将删除 pd/rule2 这条规则：

```
cat > rules.json <<EOF
[
    {
```

```
        "group_id": "pd",
        "id": "rule2"
    }
]
EOF
pd-ctl config placement save --in=rules.json
```

9.2.4 使用 pd-ctl 设置规则分组

- 查看所有的规则分组列表

```
pd-ctl config placement-rules rule-group show
```

- 查看指定 ID 的规则分组

```
pd-ctl config placement-rules rule-group show pd
```

- 设置规则分组的 index 和 override 属性

```
pd-ctl config placement-rules rule-group set pd 100 true
```

- 删除规则分组配置（如组内还有规则，则使用默认分组配置）

```
pd-ctl config placement-rules rule-group delete pd
```

9.2.5 使用 pd-ctl 批量更新分组及组内规则

使用 `rule-bundle` 子命令，可以方便地同时查看和修改规则分组及组内的所有规则。

该子命令中 `get {group_id}` 用来查询一个分组，输出结果为嵌套形式的规则分组和组内规则：

```
pd-ctl config placement-rules rule-bundle get pd
```

输出示例：

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

```
    }
  ]
}
```

rule-bundle get 子命令中可以添加 -out 参数来将输出写入文件，方便后续修改保存。

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

修改完成后，使用 rule-bundle set 子命令将文件中的配置保存至 PD 服务器。与前面介绍的 save 不同，此命令会替换服务器端该分组内的所有规则。

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

9.2.6 使用 pd-ctl 查看和修改所有配置

用户还可以使用 pd-ctl 查看和修改所有配置，即把全部配置保存至文件，修改后再覆盖保存。该操作同样使用 rule-bundle 子命令。

下面的命令将所有配置保存至 rules.json 文件：

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

编辑完文件后，使用下面的命令将配置保存至 PD 服务器：

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

9.2.7 使用 tidb-ctl 查询表相关的 key range

若需要针对元数据或某个特定的表进行特殊配置，可以通过 tidb-ctl 的 keyrange 命令来查询相关的 key。注意要添加 --encode 返回 PD 中的表示形式。

```
tidb-ctl keyrange --database test --table ttt --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (7480000000000000ff2d00000000000000f8, 7480000000000000ff2e00000000000000f8)
  table indexes: (7480000000000000ff2d5f690000000000fa, 7480000000000000ff2d5f720000000000fa)
    index c2: (7480000000000000ff2d5f698000000000ff0000010000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000020000000000fa)
    index c3: (7480000000000000ff2d5f698000000000ff0000020000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000030000000000fa)
    index c4: (7480000000000000ff2d5f698000000000ff0000030000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000040000000000fa)
  table rows: (7480000000000000ff2d5f720000000000fa, 7480000000000000ff2e00000000000000f8)
```

> **注意：**
>
> DDL 等操作会导致 table ID 发生变化，需要同步更新对应的规则。

9.3 典型场景示例

本部分介绍 Placement Rules 的使用场景示例。

9.3.1 场景一：普通的表使用 3 副本，元数据使用 5 副本提升集群容灾能力

只需要增加一条规则，将 key range 限定在 meta 数据的范围，并把 `count` 值设为 5。添加规则示例如下：

```
{
  "group_id": "pd",
  "id": "meta",
  "index": 1,
  "override": true,
  "start_key": "6d00000000000000f8",
  "end_key": "6e00000000000000f8",
  "role": "voter",
  "count": 5,
  "location_labels": ["zone", "rack", "host"]
}
```

9.3.2 场景二：5 副本按 2-2-1 的比例放置在 3 个数据中心，且第 3 个中心不产生 Leader

创建三条规则，分别设置副本数为 2、2、1，并且在每个规则内通过 `label_constraints` 将副本限定在对应的数据中心内。另外，不需要 leader 的数据中心将 `role` 改为 `follower`。

```
[
    {
        "group_id": "pd",
        "id": "zone1",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone1"]}
        ],
        "location_labels": ["rack", "host"]
    },
```

```
    {
        "group_id": "pd",
        "id": "zone2",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone2"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone3",
        "start_key": "",
        "end_key": "",
        "role": "follower",
        "count": 1,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone3"]}
        ],
        "location_labels": ["rack", "host"]
    }
]
```

9.3.3 场景三：为某张表添加 2 个 TiFlash Learner 副本

为表的 row key 单独添加一条规则，限定数量为 2，并且通过 `label_constraints` 保证副本产生在 `engine=tiflash` 的节点。注意这里使用了单独的 `group_id`，保证这条规则不会与系统中其他来源的规则互相覆盖或产生冲突。

```
{
  "group_id": "tiflash",
  "id": "learner-replica-table-ttt",
  "start_key": "7480000000000000ff2d5f720000000000fa",
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "learner",
  "count": 2,
  "label_constraints": [
    {"key": "engine", "op": "in", "values": ["tiflash"]}
  ],
  "location_labels": ["host"]
}
```

9.3.4 场景四：为某张表在有高性能磁盘的北京节点添加 2 个 Follower 副本

这个例子展示了比较复杂的 label_constraints 配置，下面的例子限定了副本放置在 bj1 或 bj2 机房，且磁盘类型不能为 hdd。

```
{
  "group_id": "follower-read",
  "id": "follower-read-table-ttt",
  "start_key": "748000000000000ff2d00000000000000f8",
  "end_key": "748000000000000ff2e00000000000000f8",
  "role": "follower",
  "count": 2,
  "label_constraints": [
    {"key": "zone", "op": "in", "values": ["bj1", "bj2"]},
    {"key": "disk", "op": "notIn", "values": ["hdd"]}
  ],
  "location_labels": ["host"]
}
```

9.3.5 场景五：将某张表迁移至 TiFlash 集群

与场景三不同，这个场景不是要在原有配置的基础上增加新副本，而是要强制覆盖一段数据的其它配置，因此需要通过配置规则分组来指定一个足够大的 index 以及设置 override 来覆盖原有规则。

规则：

```
{
  "group_id": "tiflash-override",
  "id": "learner-replica-table-ttt",
  "start_key": "748000000000000ff2d5f720000000000fa",
  "end_key": "748000000000000ff2e00000000000000f8",
  "role": "voter",
  "count": 3,
  "label_constraints": [
    {"key": "engine", "op": "in", "values": ["tiflash"]}
  ],
  "location_labels": ["host"]
}
```

规则分组：

```
{
  "id": "tiflash-override",
  "index": 1024,
  "override": true,
}
```

10 读取历史数据

10.1 使用 Stale Read 功能读取历史数据（推荐）

10.1.1 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。

10.1.1.1 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。

10.1.1.2 使用方法

TiDB 提供语句级别与会话级别的 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。

10.1.2 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。

> **警告：**
>
> 目前 Stale Read 特性无法和 TiFlash 一起使用。如果你的查询中带有 AS OF TIMESTAMP 并且 TiDB 可能从 TiFlash 副本读取数据，你可能会遇到 ERROR 1105 (HY000): stale requests require tikv backend 报错信息。
>
> 要解决该问题，你需要为使用 Stale Read 特性的查询禁用 TiFlash 副本。要禁用 TiFlash 副本，你可以使用以下任一方法：
>
> - 通过设置变量来禁用 TiFlash 副本 set session tidb_isolation_read_engines='tidb, tikv'。
> - 使用hint 强制 TiDB 从 TiKV 读取数据。

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。

10.1.2.1 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。

10.1.2.2 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。

10.1.2.2.1 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)
```

10.1.2.2.2 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。

10.1.2.2.3 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。

10.1.2.2.4 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。

10.1.3 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。

10.1.3.1 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 ""。

10.1.3.2 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)
```

10.2 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。

10.2.1 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。

10.2.2 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。

10.2.3 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。

10.2.4 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。

10.2.5 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。

11 监控与告警

11.1 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。

11.1.1 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：

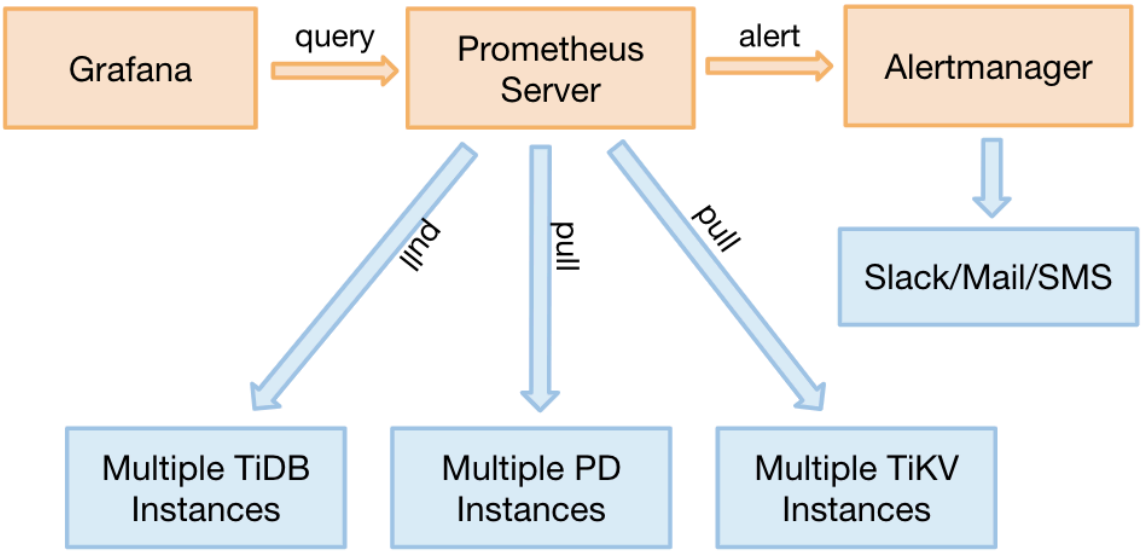


图 143: Prometheus in TiDB

11.1.2 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：

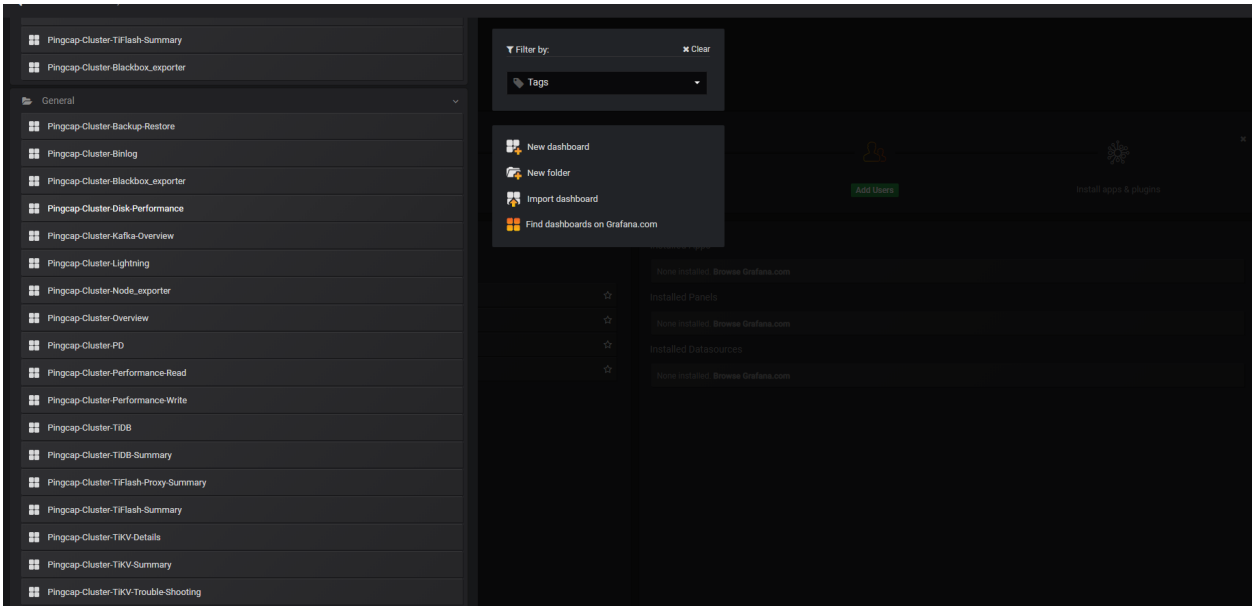


图 144: Grafana monitored_groups

- {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 145: 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。

11.2 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。

11.2.1 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。

11.2.1.1 TiDB Server

- TiDB API 地址：`http://${host}:${port}`
- 默认端口：10080

11.2.1.1.1 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
}
```

11.2.1.1.2 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
    }
]
```

11.2.1.2 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"
      }
    }
]
```

11.2.2 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。

11.3 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。

11.3.1 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

11.3.1.1 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-0.17.0.linux-amd64.tar.gz
wget https://download.pingcap.org/grafana-6.1.6.linux-amd64.tar.gz
```

解压二进制包：

```
tar -xzf prometheus-2.27.1.linux-amd64.tar.gz
tar -xzf node_exporter-0.17.0.linux-amd64.tar.gz
tar -xzf grafana-6.1.6.linux-amd64.tar.gz
```

11.3.1.2 第 2 步：在 Node1，Node2，Node3，Node4 上启动 node_exporter

```
cd node_exporter-0.17.0.linux-amd64
```

启动 node_exporter 服务：

```
./node_exporter --web.listen-address=":9100" \
    --log.level="info" &
```

11.3.1.3 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" &
```

11.3.1.4 第 4 步：在 Node1 上启动 Grafana

编辑 Grafana 的配置文件：

```
cd grafana-6.1.6 &&
vi conf/grafana.ini
```

```
...

[paths]
data = ./data
logs = ./data/log
plugins = ./data/plugins
[server]
http_port = 3000
domain = 192.168.199.113
[database]
[session]
[analytics]
```

```
check_for_updates = true
[security]
admin_user = admin
admin_password = admin
[snapshots]
[users]
[auth.anonymous]
[auth.basic]
[auth.ldap]
[smtp]
[emails]
[log]
mode = file
[log.console]
[log.file]
level = info
format = text
[log.syslog]
[event_publisher]
[dashboards.json]
enabled = false
path = ./data/dashboards
[metrics]
[grafana_net]
url = https://grafana.net

...
```

启动 Grafana 服务：

```
./bin/grafana-server \
    --config="./conf/grafana.ini" &
```

11.3.2 配置 Grafana

本小节介绍如何配置 Grafana。

11.3.2.1 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。

11.3.2.2 第 2 步：导入 Grafana 面板

执行以下步骤，为 PD Server、 TiKV Server 和 TiDB Server 分别导入 Grafana 面板：

1. 点击侧边栏的 Grafana 图标。
2. 在侧边栏菜单中，依次点击 Dashboards > Import 打开 Import Dashboard 窗口。
3. 点击 Upload .json File 上传对应的 JSON 文件（下载 TiDB Grafana 配置文件。

> **注意：**
>
> TiKV、 PD 和 TiDB 面板对应的 JSON 文件分别为 `tikv_summary.json`，`tikv_details.json`，`tikv_trouble_shooting.json`，`pd.json`，`tidb.json`，`tidb_summary.json`。

4. 点击 Load。
5. 选择一个 Prometheus 数据源。
6. 点击 Import，Prometheus 面板即导入成功。

11.3.3 查看组件 metrics

在顶部菜单中，点击 New dashboard，选择要查看的面板。

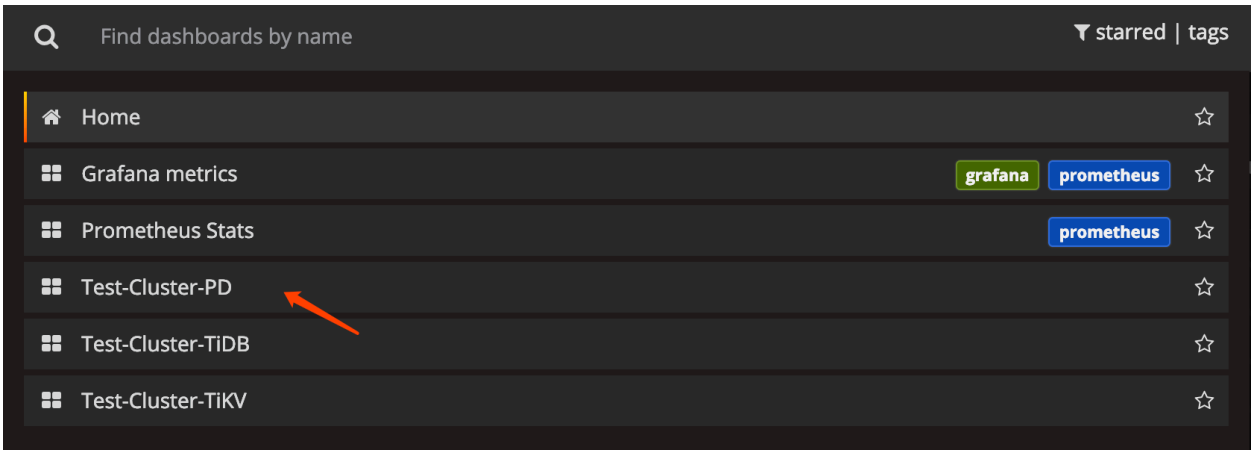


图 146: view dashboard

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。

11.4 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。

11.4.1 使用方法

可以通过访问 <https://metricstool.pingcap.com> 来使用 MetricsTool。它主要提供以下三种功能：

- 导出快照：提供一段在浏览器开发者工具上运行的用户脚本。你可以使用这个脚本在任意 Grafana v6.x.x 服务器上下载当前 Dashboard 中所有可见面板的快照。

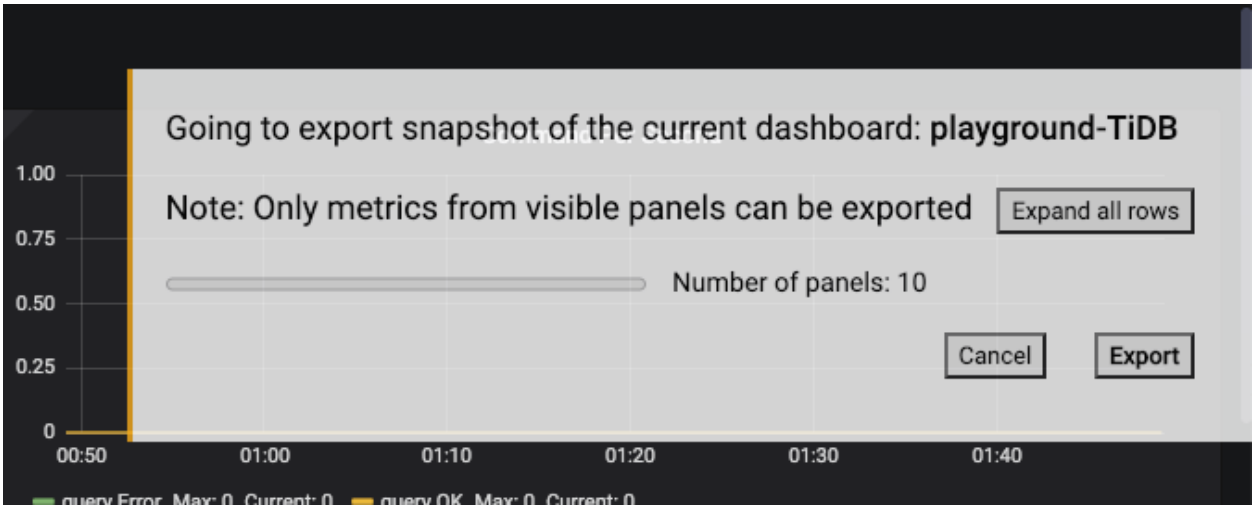


图 147: 运行用户脚本后的 MetricsTool Exporter 截图

- 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。

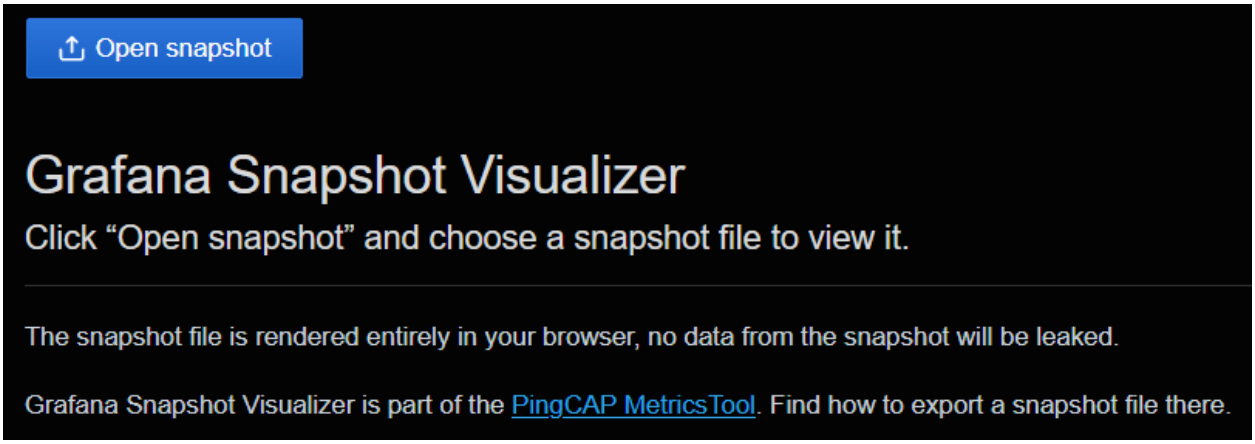


图 148: MetricsTool Visualizer 截图

- 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。

11.4.2 FAQs

11.4.2.1 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。

11.4.2.2 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。

11.4.2.3 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。

11.4.2.4 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。

11.4.2.5 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。

11.4.2.6 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。

11.5 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

11.5.1 TiDB 报警规则

本节介绍了 TiDB 组件的报警项。

11.5.1.1 紧急级别报警项

11.5.1.1.1 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。

11.5.1.1.2 TiDB_tikvclient_region_err_total

- 报警规则：

  ```
  increase(tidb_tikvclient_region_err_total[10m])> 6000
  ```

- 规则描述：

  TiDB 访问 TiKV 时发生了 Region 错误。如果在 10 分钟之内该错误多于 6000 次，则报警。

- 处理方法：

  查看 TiKV 的监控状态。

11.5.1.1.3 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。

11.5.1.1.4 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。

11.5.1.2 严重级别报警项

11.5.1.2.1 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。

11.5.1.3 警告级别报警项

11.5.1.3.1 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。

11.5.1.3.2 TiDB_query_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 1
```

- 规则描述：

  TiDB 处理请求的延时。如果.99 的延迟大于 1 秒，则报警。

- 处理方法：

  查看 TiDB 的日志，搜索 SLOW_QUERY 和 TIME_COP_PROCESS 关键字，查找慢 SQL。

11.5.1.3.3 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。

11.5.1.3.4 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。

11.5.1.3.5 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。

11.5.1.3.6 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。

11.5.2 PD 报警规则

本节介绍了 PD 组件的报警项。

11.5.2.1 紧急级别报警项

11.5.2.1.1 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。

11.5.2.2 严重级别报警项

11.5.2.2.1 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。

11.5.2.2.2 PD_miss_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="miss_peer_region_count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

Region 的副本数小于 `max-replicas` 配置的值。这通常是由于 TiKV 宕机等问题导致一段时间内一些 Region 缺副本。

- 处理方法：

  - 查看是否有 TiKV 宕机或在做下线操作，尝试定位问题产生的原因。
  - 观察 region health 面板，查看 `miss_peer_region_count` 是否在不断减少。

11.5.2.3 警告级别报警项

11.5.2.3.1 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。

11.5.2.3.2 PD_cluster_low_space

- 报警规则：

```
(sum(pd_cluster_status{type="store_low_space_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

表示 TiKV/TiFlash 节点空间不足。

- 处理方法：

  - 检查集群中的空间是否普遍不足。如果是，则需要扩容。
  - 检查 Region balance 调度是否有问题。如果有问题，会导致数据分布不均衡。
  - 检查是否有文件占用了大量磁盘空间，比如日志、快照、core dump 等文件。
  - 降低该节点的 Region weight 来减少数据量。
  - 无法释放空间时，可以考虑主动下线该节点，防止由于磁盘空间不足而宕机。

11.5.2.3.3 PD_etcd_network_peer_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_network_peer_round_trip_time_seconds_bucket[1m]))by (
↪ To, instance, job, le))> 1
```

- 规则描述：

PD 节点之间网络延迟高，严重情况下会导致 leader 超时和 TSO 存盘超时，从而影响集群服务。

- 处理方法：

  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

11.5.2.3.4 PD_tidb_handle_requests_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(pd_client_request_handle_requests_duration_seconds_bucket{
↪ type="tso"}[1m]))by (instance, job, le))> 0.1
```

- 规则描述：

  PD 处理 TSO 请求耗时过长，一般是由于负载过高。

- 处理方法：

  - 检查服务器负载状况。
  - 使用 pprof 抓取 PD 的 CPU profile 进行分析。
  - 手动切换 PD leader。
  - 如果是环境问题，则将有问题的 PD 下线替换。

11.5.2.3.5 PD_down_peer_region_nums

- 报警规则：

```
(sum(pd_regions_status{type="down-peer-region-count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft leader 上报有不响应 peer 的 Region 数量。

- 处理方法：

  - 检查是否有 TiKV 宕机，或刚发生重启，或者繁忙。
  - 观察 region health 面板，检查 down_peer_region_count 是否在不断减少。
  - 检查是否有 TiKV 之间网络不通。

11.5.2.3.6 PD_pending_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="pending-peer-region-count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft log 落后的 Region 过多。由于调度产生少量的 pending peer 是正常的，但是如果持续很高，就可能有问题。

- 处理方法：

  - 观察 region health 面板，检查 pending_peer_region_count 是否在不断减少。
  - 检查 TiKV 之间的网络状况，特别是带宽是否足够。

11.5.2.3.7 PD_leader_change

- 报警规则：

```
count(changes(pd_tso_events{type="save"}[10m])> 0)>= 2
```

- 规则描述：

  近期发生了 PD leader 切换。

- 处理方法：

  - 排除人为因素，比如重启 PD、手动 transfer leader 或调整 leader 优先级等。
  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

11.5.2.3.8 TiKV_space_used_more_than_80%

- 报警规则：

```
sum(pd_cluster_status{type="storage_size"})/ sum(pd_cluster_status{type="storage_capacity"})
↪ * 100 > 80
```

- 规则描述：

  集群空间占用超过 80%。

- 处理方法：

  - 确认是否需要扩容。
  - 排查是否有文件占用了大量磁盘空间，比如日志、快照或 core dump 等文件。

11.5.2.3.9 PD_system_time_slow

- 报警规则：

```
changes(pd_tso_events{type="system_time_slow"}[10m])>= 1
```

- 规则描述：

  系统时间可能发生回退。

- 处理方法：

  检查系统时间设置是否正确。

11.5.2.3.10 PD_no_store_for_making_replica

- 报警规则：

```
increase(pd_checker_event_count{type="replica_checker", name="no_target_store"}[1m])> 0
```

- 规则描述：

  没有合适的 store 用来补副本。

- 处理方法：

  - 检查 store 是否空间不足。
  - 根据 label 配置（如果有这个配置的话）来检查是否有可以补副本的 store。

11.5.3 TiKV 报警规则

本节介绍了 TiKV 组件的报警项。

11.5.3.1 紧急级别报警项

11.5.3.1.1 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。

11.5.3.1.2 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。否则需要报备开发人员 support@pingcap.com 进行处理。

11.5.3.2 严重级别报警项

11.5.3.2.1 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。

11.5.3.2.2 TiKV_channel_full_total

- 报警规则：

```
sum(rate(tikv_channel_full_total[10m]))BY (type, instance)> 0
```

- 规则描述：

  该错误通常是因为 Raftstore 线程卡死，TiKV 的压力已经非常大了。

- 处理方法：

  1. 观察 Raft Propose 监控，看这个报警的 TiKV 节点是否明显有比其他 TiKV 高很多。如果是，表明这个 TiKV 上有热点，需要检查热点调度是否能正常工作。
  2. 观察 Raft IO 监控，看延迟是否升高。如果延迟很高，表明磁盘可能有瓶颈。一个能缓解但不怎么安全的办法是将 sync-log 改成 false。
  3. 观察 Raft Process 监控，看 tick duration 是否很高。如果是，需要在 [raftstore] 配置下加上 raft-base-tick-interval = “2s”。

11.5.3.2.3 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。

11.5.3.2.4 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。

11.5.3.2.5 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。

11.5.3.2.6 TiKV_async_request_write_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
↪ type="write"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明 Raft write 耗时很长。

- 处理方法：

  1. 检查 Raftstore 上的压力，参考TiKV_channel_full_total 的处理方法。
  2. 检查 apply worker 线程的压力。

11.5.3.2.7 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。

11.5.3.2.8 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。

11.5.3.2.9 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。

11.5.3.2.10 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。

11.5.3.2.11 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。

11.5.3.2.12 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。

11.5.3.3 警告级别报警项

11.5.3.3.1 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。

11.5.3.3.2 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。

11.5.3.3.3 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”。

11.5.3.3.4 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。

11.5.3.3.5 TiKV_scheduler_command_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_scheduler_command_duration_seconds_bucket[1m]))by (le
↪ , instance, type)/ 1000)> 1
```

- 规则描述：

表明 Scheduler 执行命令的耗时。

- 处理方法：

参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

11.5.3.3.6 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。

11.5.3.3.7 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。

11.5.3.3.8 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%。

11.5.3.3.9 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：

查看是哪一类任务的值偏高，通常 Coprocessor、apply worker 这类任务都可以在其他指标里找到解决办法。

11.5.3.3.10 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。

11.5.3.3.11 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。

11.5.4 TiFlash 报警规则

关于 TiFlash 报警规则的详细描述，参见TiFlash 报警规则。

11.5.5 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。

11.5.6 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。

11.5.7 Node_exporter 主机报警规则

本节介绍了 Node_exporter 主机的报警项。

11.5.7.1 紧急级别报警项

11.5.7.1.1 NODE_disk_used_more_than_80%

- 报警规则：

```
node_filesystem_avail{fstype=~"(ext.|xfs)", mountpoint!~"/boot"} / node_filesystem_size{
↪ fstype=~"(ext.|xfs)", mountpoint!~"/boot"} * 100 <= 20
```

- 规则描述：

  机器磁盘空间使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -h` 命令，查看磁盘空间使用率，做好扩容计划。

11.5.7.1.2 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。

11.5.7.1.3 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。

11.5.7.2 严重级别报警项

11.5.7.2.1 NODE_memory_used_more_than_80%

- 报警规则：

```
(((node_memory_MemTotal-node_memory_MemFree-node_memory_Cached)/(node_memory_MemTotal)*100))
↪ >= 80
```

- 规则描述：

  机器内存使用率超过 80%。

- 处理方法：

  - 在 Grafana Node Exporter 页面查看该主机的 Memory 面板，检查 `Used` 是否过高，`Available` 内存是否过低。
  - 登录机器，执行 `free -m` 命令查看内存使用情况，执行 `top` 看是否有异常进程的内存使用率过高。

11.5.7.3 警告级别报警项

11.5.7.3.1 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。

11.5.7.3.2 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。

11.5.7.3.3 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。

11.5.7.3.4 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。

11.5.7.3.5 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。

11.5.8 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。

11.5.8.1 紧急级别报警项

11.5.8.1.1 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。

11.5.8.1.2 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。

11.5.8.1.3 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。

11.5.8.1.4 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。

11.5.8.1.5 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。

11.5.8.1.6 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。

11.5.8.1.7 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。

11.5.8.1.8 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。

11.5.8.1.9 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。

11.5.8.1.10 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。

11.5.8.1.11 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。

11.5.8.1.12 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。

11.5.8.2 警告级别报警项

11.5.8.2.1 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。

11.6 TiFlash 报警规则

本文介绍了 TiFlash 集群的报警规则。

11.6.1 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。

11.6.2 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。

11.6.3 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。

11.6.4 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。

11.7 错误码与故障诊断

本篇文档描述在使用 TiDB 过程中会遇到的问题以及解决方法。

11.7.1 错误码

TiDB 兼容 MySQL 的错误码，在大多数情况下，返回和 MySQL 一样的错误码。关于 MySQL 的错误码列表，详见 [MySQL 5.7 Error Message Reference。另外还有一些 TiDB 特有的错误码：

> 注意：
>
> 有一部分错误码属于内部错误，正常情况下 TiDB 会自行处理不会直接返回给用户，故没有在此列出。
>
> 如果您遇到了这里没有列出的错误码，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8001

  请求使用的内存超过 TiDB 内存使用的阈值限制。出现这种错误，可以通过调整系统变量`tidb_mem_quota_query`来增大单个 SQL 使用的内存上限。

- Error Number: 8002

  带有 `SELECT FOR UPDATE` 语句的事务，在遇到写入冲突时，为保证一致性无法进行重试，事务将进行回滚并返回该错误。出现这种错误，应用程序可以安全地重新执行整个事务。

- Error Number: 8003

  `ADMIN CHECK TABLE` 命令在遇到行数据跟索引不一致的时候返回该错误，在检查表中数据是否有损坏时常出现。出现该错误时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8004

  单个事务过大，原因及解决方法请参考这里

- Error Number: 8005

  事务在 TiDB 中遇到了写入冲突，原因及解决方法请参考这里

- Error Number: 8018

  当执行重新载入插件时，如果之前插件没有载入过，则会出现该错误。出现该错误，进行插件首次载入即可。

- Error Number: 8019

  重新载入的插件版本与之前的插件版本不一致，无法重新载入插件并报告该错误。可重新载入插件，确保插件的版本与之前载入的插件版本一致。

- Error Number: 8020

  当表被加锁时，如果对该表执行写入操作，将出现该错误。请将表解锁后，再进行尝试写入。

- Error Number: 8021

  当向 TiKV 读取的 key 不存在时将出现该错误，该错误用于内部使用，对外表现为读到的结果为空。

- Error Number: 8022

  事务提交失败，已经回滚，应用程序可以安全的重新执行整个事务。

- Error Number: 8023

  在事务内，写入事务缓存时，设置了空值，将返回该错误。这是一个内部使用的错误，将由内部进行处理，不会返回给应用程序。

- Error Number: 8024

  非法的事务。当事务执行时，发现没有获取事务的 ID (Start Timestamp)，代表正在执行的事务是一个非法的事务，将返回该错误。通常情况下不会出现该问题，当发生时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8025

  写入的单条键值对过大。TiDB 默认支持最大 6MB 的单个键值对，超过该限制可适当调整`txn-entry-size-limit` 配置项以放宽限制。

- Error Number: 8026

  使用了没有实现的接口函数。该错误仅用于数据库内部，应用程序不会收到这个错误。

- Error Number: 8027

  表结构版本过期。TiDB 采用在线变更表结构的方法。当 TiDB server 表结构版本落后于整个系统的时，执行 SQL 将遇到该错误。遇到该错误，请检查该 TiDB server 与 PD leader 之间的网络。

- Error Number: 8028

  TiDB 没有表锁（在 MySQL 中称为元数据锁，在其他数据库中可能称为意向锁）。当事务执行时，TiDB 表结构发生了变化是无法被事务感知到的。因此，TiDB 在事务提交时，会对事务涉及表的结构进行检查。如果事务执行中表结构发生了变化，则事务将提交失败，并返回该错误。遇到该错误，应用程序可以安全地重新执行整个事务。

- Error Number: 8029

  当数据库内部进行数值转换发生错误时，将会出现该错误，该错误仅在内部使用，对外部应用将转换为具体类型的错误。

- Error Number: 8030

  将值转变为带符号正整数时发生了越界，将结果显示为负数。多在告警信息里出现。

- Error Number: 8031

  将负数转变为无符号数时，将负数转变为了正数。多在告警信息里出现。

- Error Number: 8032

  使用了非法的 year 格式。year 只允许 1 位、2 位和 4 位数。

- Error Number: 8033

  使用了非法的 year 值。year 的合法范围是 (1901, 2155)。

- Error Number: 8037

  week 函数中使用了非法的 mode 格式。mode 必须是一位数字，范围 [0, 7]。

- Error Number: 8038

  字段无法获取到默认值。一般作为内部错误使用，转换成其他具体错误类型后，返回给应用程序。

- Error Number: 8040

  尝试进行不支持的操作，比如在 View 和 Sequence 上进行 lock table。

- Error Number: 8047

  设置了不支持的系统变量值，通常在用户设置了数据库不支持的变量值后的告警信息里出现。

- Error Number: 8048

  设置了不支持的隔离级别，如果是使用第三方工具或框架等无法修改代码进行适配的情况，可以考虑通过tidb_skip_isolation_level_check 来绕过这一检查。

```
set @@tidb_skip_isolation_level_check = 1;
```

- Error Number: 8050

  设置了不支持的权限类型，遇到该错误请参考TiDB 权限说明进行调整。

- Error Number: 8051

  TiDB 在解析客户端发送的 Exec 参数列表时遇到了未知的数据类型。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8052

  来自客户端的数据包的序列号错误。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8055

  当前快照过旧，数据可能已经被 GC。可以调大`tidb_gc_life_time` 的值来避免该问题。从 TiDB v4.0.8 版本起，TiDB 会自动为长时间运行的事务保留数据，一般不会遇到该错误。

  有关 GC 的介绍和配置可以参考GC 机制简介和GC 配置文档。

- Error Number: 8059

  自动随机量可用次数用尽无法进行分配。当前没有恢复这类错误的方法。建议在使用 auto random 功能时使用 bigint 以获取最大的可分配次数，并尽量避免手动给 auto random 列赋值。相关的介绍和使用建议可以参考auto random 功能文档。

- Error Number: 8060

  非法的自增列偏移量。请检查 `auto_increment_increment` 和 `auto_increment_offset` 的取值是否符合要求。

- Error Number: 8061

  不支持的 SQL Hint。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8062

  SQL Hint 中使用了非法的 token，与 Hint 的保留字冲突。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8063

  SQL Hint 中限制内存使用量超过系统设置的上限，设置被忽略。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8064

  解析 SQL Hint 失败。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8065

  SQL Hint 中使用了非法的整数。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8066

  JSON_OBJECTAGG 函数的第二个参数是非法参数。

- Error Number: 8101

  插件 ID 格式错误，正确的格式是 `[name]-[version]` 并且 name 和 version 中不能带有 ‘-’ 。

- Error Number: 8102

  无法读取插件定义信息。请检查插件相关的配置。

- Error Number: 8103

  插件名称错误，请检查插件的配置。

- Error Number: 8104

  插件版本不匹配，请检查插件的配置。

- Error Number: 8105

  插件被重复载入。

- Error Number: 8106

  插件定义的系统变量名称没有以插件名作为开头，请联系插件的开发者进行修复。

- Error Number: 8107

  载入的插件未指定版本或指定的版本过低，请检查插件的配置。

- Error Number: 8108

  不支持的执行计划类型。该错误为内部处理的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8109

  analyze 索引时找不到指定的索引。

- Error Number: 8110

  不能进行笛卡尔积运算，需要将配置文件里的 `cross-join` 设置为 `true`。

- Error Number: 8111

  execute 语句执行时找不到对应的 prepare 语句。

- Error Number: 8112

  execute 语句的参数个数与 prepare 语句不符合。

- Error Number: 8113

  execute 语句涉及的表结构在 prepare 语句执行后发生了变化。

- Error Number: 8115

  不支持 prepare 多行语句。

- Error Number: 8116

  不支持 prepare DDL 语句。

- Error Number: 8120

  获取不到事务的 start tso，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

- Error Number: 8121

  权限检查失败，请检查数据库的权限配置。

- Error Number: 8122

  指定了通配符，但是找不到对应的表名。

- Error Number: 8123

  带聚合函数的 SQL 中返回非聚合的列，违反了 `only_full_group_by` 模式。请修改 SQL 或者考虑关闭 `only_full_group_by` 模式。

- Error Number: 8129

  TiDB 尚不支持键长度 >= 65536 的 JSON 对象。

- Error Number: 8138

  事务试图写入的行值有误，请参考数据索引不一致报错。

- Error Number: 8139

  事务试图写入的行和索引的 handle 值不一致，请参考数据索引不一致报错。

- Error Number: 8140

  事务试图写入的行和索引的值不一致，请参考数据索引不一致报错。

- Error Number: 8141

  事务写入时，对 key 的存在性断言报错，请参考数据索引不一致报错。

- Error Number: 8143

  非事务 DML 语句的一个 batch 报错，语句中止，请参考非事务 DML 语句

- Error Number: 8200

  尚不支持的 DDL 语法。请参考与 MySQL DDL 的兼容性。

- Error Number: 8214

  DDL 操作被 admin cancel 操作终止。

- Error Number: 8215

  Admin Repair 表失败，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8216

  自动随机列使用的方法不正确，请参考auto random 功能文档进行修改。

- Error Number: 8223

  检测出数据与索引不一致的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8224

  找不到 DDL job，请检查 restore 操作指定的 job id 是否存在。

- Error Number: 8225

  DDL 已经完成，无法被取消。

- Error Number: 8226

  DDL 几乎要完成了，无法被取消。

- Error Number: 8227

  创建 Sequence 时使用了不支持的选项，支持的选项的列表可以参考Sequence 使用文档。

- Error Number: 8228

  在 Sequence 上使用 `setval` 时指定了不支持的类型，该函数的示例可以在Sequence 使用文档中找到。

- Error Number: 8229

  事务超过存活时间，遇到该问题可以提交或者回滚当前事务，开启一个新事务。

- Error Number: 8230

  TiDB 目前不支持在新添加的列上使用 Sequence 作为默认值，如果尝试进行这类操作会返回该错误。

- Error Number: 9001

  请求 PD 超时，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

- Error Number: 9002

  请求 TiKV 超时，请检查 TiKV Server 状态/监控/日志以及 TiDB Server 与 TiKV Server 之间的网络。

- Error Number: 9003

  TiKV 操作繁忙，一般出现在数据库负载比较高时，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9004

  当数据库上承载的业务存在大量的事务冲突时，会遇到这种错误，请检查业务代码。

- Error Number: 9005

  某个 Raft Group 不可用，如副本数目不足，出现在 TiKV 比较繁忙或者是 TiKV 节点停机的时候，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9006

  GC Life Time 间隔时间过短，长事务本应读到的数据可能被清理了，应增加 GC Life Time。

- Error Number: 9007

  事务在 TiKV 中遇到了写入冲突，原因及解决方法请参考这里。

- Error Number: 9008

  同时向 TiKV 发送的请求过多，超过了限制。请调大 `tidb_store_limit` 或将其设置为 0 来取消对请求流量的限制。

- Error Number: 9010

  TiKV 无法处理这条 raft log，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9012

  请求 TiFlash 超时。请检查 TiFlash Server 状态/监控/日志以及 TiDB Server 与 TiFlash Server 之间的网络。

- Error Number: 9013

  TiFlash 操作繁忙。该错误一般出现在数据库负载比较高时。请检查 TiFlash Server 的状态/监控/日志。

11.7.2 故障诊断

参见故障诊断文档以及FAQ。

11.8 监控指标

11.8.1 Overview 面板重要监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

对于日常运维，我们单独挑选出重要的 Metrics 放在 Overview 页面，方便日常运维人员观察集群组件 (PD, TiDB, TiKV) 使用状态以及集群使用状态。

以下为 Overview Dashboard 监控说明：

11.8.1.1 Services Port Status

- Services Up：各服务在线节点数量

11.8.1.2 PD

- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小，TiKV 多副本的空间占用也会包含在内
- Normal stores：处于正常状态的节点数目
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Number of Regions：当前集群的 Region 总量，请注意 Region 数量与副本数无关
- 99% completed_cmds_duration_seconds：单位时间内，99% 的 pd-server 请求执行时间小于监控曲线的值，一般 <= 5ms
- Handle_requests_duration_seconds：PD 发送请求的网络耗时
- Region health：每个 Region 的状态，通常情况下，pending 的 peer 应该少于 100，miss 的 peer 不能一直大于 0
- Hot write Region’s leader distribution：每个 TiKV 实例上是写入热点的 leader 的数量
- Hot read Region’s leader distribution：每个 TiKV 实例上是读取热点的 leader 的数量
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

11.8.1.3 TiDB

- Statement OPS：不同类型 SQL 语句每秒执行的数量。按 SELECT、 INSERT、 UPDATE 等来统计
- Duration：执行的时间
  - 客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行的时间
  - 由于 TiDB 支持 Multi-Query，因此，可以接受客户端一次性发送的多条 SQL 语句，如：`select 1; select 1;`。此时，统计的执行时间是所有 SQL 执行完之后的总时间
- CPS By Instance：每个 TiDB 实例上的命令统计。按照命令和执行结果成功或失败来统计

- Failed Query OPM：每个 TiDB 实例上，每秒钟执行 SQL 语句发生错误按照错误类型的统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码
- Connection count：每个 TiDB 的连接数
- Memory Usage：每个 TiDB 实例的内存使用统计，分为进程占用内存和 Golang 在堆上申请的内存
- Transaction OPS：每秒事务执行数量统计
- Transaction Duration：事务执行的时间
- KV Cmd OPS：KV 命令执行数量统计
- KV Cmd Duration 99：KV 命令执行的时间
- PD TSO OPS：TiDB 每秒从 PD 获取 TSO 的数量
- PD TSO Wait Duration：TiDB 等待从 PD 获取 TS 的时间
- TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
- Lock Resolve OPS：TiDB 清理锁操作的数量。当 TiDB 的读写请求遇到锁时，会尝试进行锁清理
- Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
- KV Backoff OPS：TiKV 返回错误信息的数量

11.8.1.4 TiKV

- leader：各个 TiKV 节点上 Leader 的数量分布
- region：各个 TiKV 节点上 Region 的数量分布
- CPU：各个 TiKV 节点的 CPU 使用率
- Memory：各个 TiKV 节点的内存使用量
- store size：每个 TiKV 实例的使用的存储空间的大小
- cf size：每个列族的大小
- channel full：每个 TiKV 实例上 channel full 错误的数量，正常情况下应当为 0
- server report failures：每个 TiKV 实例上报错的消息个数，正常情况下应当为 0
- scheduler pending commands：每个 TiKV 实例上 pending 命令的个数
- coprocessor executor count：TiKV 每秒收到的 coprocessor 操作数量，按照 coprocessor 类型统计
- coprocessor request duration：处理 coprocessor 读请求所花费的时间
- raft store CPU：raftstore 线程的 CPU 使用率，线程数量默认为 2（通过 `raftstore.store-pool-size` 配置）。如果单个线程使用率超过 80%，说明使用率很高
- Coprocessor CPU：coprocessor 线程的 CPU 使用率

11.8.1.5 System Info

- Vcores：CPU 核心数量
- Memory：内存总大小
- CPU Usage：CPU 使用率，最大为 100%
- Load [1m]：1 分钟的负载情况
- Memory Available：剩余内存大小
- Network Traffic：网卡流量统计
- TCP Retrans：TCP 重传数量统计
- IO Util：磁盘使用率，最高为 100%，一般到 80% - 90% 就需要考虑加节点

11.8.1.6 图例



图 149: overview

11.8.2 Performance Overview 面板重要监控指标详解

使用 TiUP 部署 TiDB 集群时，你可以一键部署监控系统 (Prometheus & Grafana)。监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview、Performance_overview 等。

Performance Overview Dashboard 按总分结构对 TiDB、TiKV、PD 的性能指标进行编排组织，包含了以下三部分内容：

- 总的概览：数据库时间和 SQL 执行时间概览。通过颜色优化法，你可以快速识别数据库负载特征和性能瓶颈。
- 资源负载：关键指标和资源利用率，包含数据库 QPS、应用和数据库的连接信息和请求命令类型、数据库内部 TSO 和 KV 请求 OPS、TiDB 和 TiKV 的资源使用概况。
- 自上而下的延迟分解：Query 延迟和连接空闲时间对比、Query 延迟分解、execute 阶段 TSO 请求和 KV 请求的延迟、TiKV 内部写延迟的分解等。

借助 Performance Overview Dashboard，你可以高效地进行性能分析，确认用户响应时间的瓶颈是否在数据库中。如果数据库是整个系统的瓶颈，通过数据库时间概览和 SQL 延迟的分解，定位数据库内部的瓶颈点，并进行针对性的优化。详情请参考TiDB 性能分析和优化方法。

以下为 Performance Overview Dashboard 监控说明：

11.8.2.1 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

11.8.2.2 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。

11.8.2.3 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。

11.8.2.4 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

11.8.2.5 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）

11.8.2.6 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

11.8.2.7 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。

11.8.2.8 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

11.8.2.9 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

11.8.2.10 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

11.8.2.11 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

11.8.2.12 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

11.8.2.13 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。

11.8.2.14 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。

11.8.2.15 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。

11.8.2.16 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

11.8.2.17 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

11.8.2.18 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。

11.8.2.19 图例

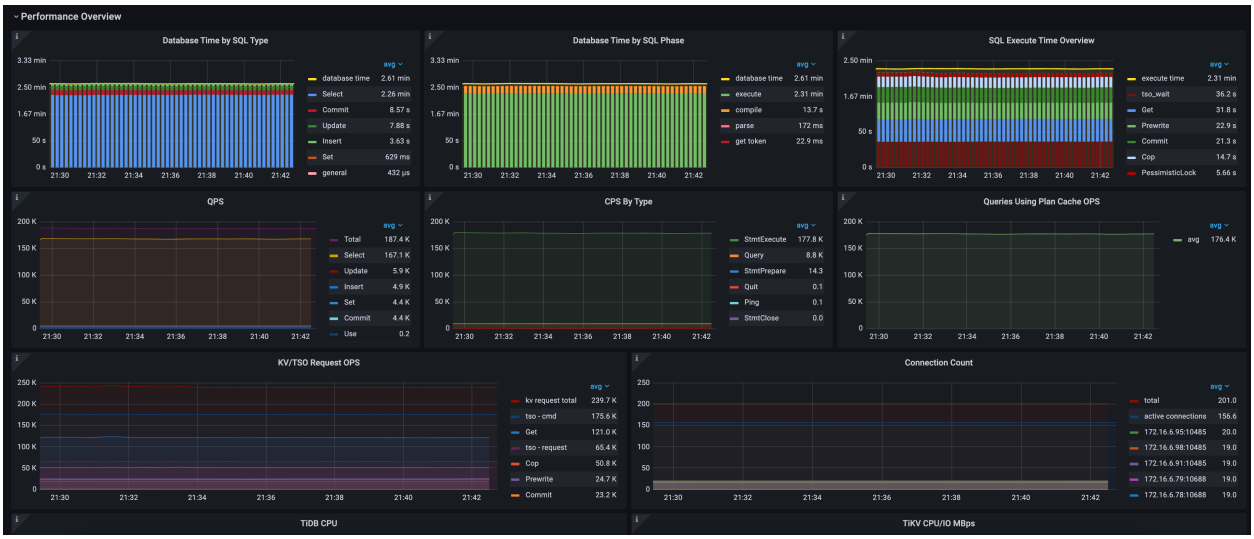


图 150: performance overview

11.8.3 TiDB 重要监控指标详解

使用 TiUP 部署 TiDB 集群时，你可以一键部署监控系统 (Prometheus & Grafana)，参考监控架构TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview、Performance_overview 等。TiDB 分为 TiDB 和 TiDB Summary 面板，两个面板的区别如下：

- TiDB 面板：提供尽可能全面的信息，供排查集群异常。
- TiDB Summary 面板：将 TiDB 面板中用户最为关心的部分抽取出来，并做了些许修改。主要用于提供数据库日常运行中用户关心的数据，如 QPS、TPS、响应延迟等，以便作为外部展示、汇报用的监控信息。

以下为 TiDB Dashboard 关键监控指标的说明：

11.8.3.1 关键指标说明

- Query Summary
  - Duration：执行时间
    * 客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行时间。
    * 由于 TiDB 支持 Multi-Query，因此，客户端可以一次性发送多条 SQL 语句，如 `select 1; select 1; select 1;`。此时的执行时间是所有 SQL 语句执行完之后的总时间。
  - Command Per Second：TiDB 按照执行结果成功或失败来统计每秒处理的命令数。
  - QPS：按 SELECT、INSERT、UPDATE 类型统计所有 TiDB 实例上每秒执行的 SQL 语句数量。
  - CPS By Instance：按照命令和执行结果成功或失败来统计每个 TiDB 实例上的命令。

  - Failed Query OPM：每个 TiDB 实例上，对每分钟执行 SQL 语句发生的错误按照错误类型进行统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码。
  - Slow query：慢查询的处理时间（整个慢查询耗时、Coprocessor 耗时、Coprocessor 调度等待时间），慢查询分为 internal 和 general SQL 语句。
  - Connection Idle Duration：空闲连接的持续时间。
  - 999/99/95/80 Duration：不同类型的 SQL 语句执行耗时（不同百分位）。

- Query Detail
  - Duration 80/95/99/999 By Instance：每个 TiDB 实例执行 SQL 语句的耗时（不同百分位）。
  - Failed Query OPM Detail：每个 TiDB 实例上，对每分钟执行 SQL 语句发生的错误按照错误类型进行统计（例如语法错误、主键冲突等）。
  - Internal SQL OPS：整个 TiDB 集群内部 SQL 语句执行的 QPS。内部 SQL 语句是指 TiDB 内部自动执行的 SQL 语句，一般由用户 SQL 语句来触发或者内部定时任务触发。

- Server
  - Uptime：每个 TiDB 实例的运行时间。
  - Memory Usage：每个 TiDB 实例的内存使用，分为进程占用内存和 Golang 在堆上申请的内存。
  - CPU Usage：每个 TiDB 实例的 CPU 使用。
  - Connection Count：每个 TiDB 的连接数。
  - Open FD Count：每个 TiDB 实例的打开的文件描述符数量。
  - Disconnection Count：每个 TiDB 实例断开连接的数量。
  - Event OPM：每个 TiDB 实例关键事件，例如 start，close，graceful-shutdown，kill，hang 等。
  - Goroutine Count：每个 TiDB 实例的 Goroutine 数量。
  - Prepare Statement Count：每个 TiDB 实例现存的 `Prepare` 语句数以及总数。
  - Keep Alive OPM：每个 TiDB 实例每分钟刷新监控的次数，通常不需要关注。
  - Panic And Critical Error：TiDB 中出现的 Panic、Critical Error 数量。
  - Time Jump Back OPS：每个 TiDB 实例上每秒操作系统时间回跳的次数。
  - Get Token Duration：每个连接获取 Token 的耗时。
  - Skip Binlog Count：TiDB 写入 Binlog 失败的数量。
  - Client Data Traffic：TiDB 和客户端的数据流量。

- Transaction
  - Transaction OPS：每秒事务的执行数量
  - Duration：事务执行时间
  - Transaction Statement Num：事务中的 SQL 语句数量
  - Transaction Retry Num：事务重试次数
  - Session Retry Error OPS：事务重试时每秒遇到的错误数量，分为重试失败和超过最大重试次数两种类型
  - Commit Token Wait Duration：事务提交时的流控队列等待时间。当出现较长等待时，代表提交事务过大，正在限流。如果系统还有资源可以使用，可以通过增大系统变量 `tidb_committer_concurrency` 的值来加速提交
  - KV Transaction OPS：每个 TiDB 内部每秒执行的事务数量
    * 一个用户的事务，在 TiDB 内部可能会触发多次事务执行，其中包含，内部元数据的读取，用户事务原子性地多次重试执行等
    * TiDB 内部的定时任务也会通过事务来操作数据库，这部分也包含在这个面板里
  - KV Transaction Duration：每个 TiDB 内部执行事务的耗时

  - Transaction Regions Num：事务操作的 Region 数量
  - Transaction Write KV Num Rate and Sum：事务写入 KV 的速率总和
  - Transaction Write KV Num：事务操作的 KV 数量
  - Statement Lock Keys：单个语句的加锁个数
  - Send HeartBeat Duration：事务发送心跳的时间间隔
  - Transaction Write Size Bytes Rate and sum：事务写入字节数的速率总和
  - Transaction Write Size Bytes：事务写入的数据大小
  - Acquire Pessimistic Locks Duration：加锁所消耗的时间
  - TTL Lifetime Reach Counter：事务的 TTL 寿命上限。TTL 上限默认值 1 小时，它的含义是从悲观事务第一次加锁，或者乐观事务的第一个 prewrite 开始，超过了 1 小时。可以通过修改 TiDB 配置文件中 `max-txn-ttl` 来改变 TTL 寿命上限
  - Load Safepoint OPS：加载 Safepoint 的次数。Safepoint 作用是在事务读数据时，保证不读到 Safepoint 之前的数据，保证数据安全。因为，Safepoint 之前的数据有可能被 GC 清理掉
  - Pessimistic Statement Retry OPS：悲观语句重试次数。当语句尝试加锁时，可能遇到写入冲突，此时，语句会重新获取新的 snapshot 并再次加锁
  - Transaction Types Per Seconds：每秒采用两阶段提交 (2PC)、异步提交（Async Commit) 和一阶段提交 (1PC) 机制的事务数量，提供成功和失败两种数量

- Executor

  - Parse Duration：SQL 语句解析耗时统计。
  - Compile Duration：将解析后的 SQL AST 编译成执行计划的耗时。
  - Execution Duration：执行 SQL 语句执行计划耗时。
  - Expensive Executor OPS：每秒消耗系统资源比较多的算子。包括 Merge Join、Hash Join、Index Look Up Join、Hash Agg、 Stream Agg、 Sort、 TopN 等。
  - Queries Using Plan Cache OPS：每秒使用 Plan Cache 的查询数量。
  - Plan Cache Miss OPS：每秒出现 Plan Cache Miss 的数量。

- Distsql

  - Distsql Duration： Distsql 处理的时长
  - Distsql QPS：每秒 Distsql 的数量
  - Distsql Partial QPS：每秒 Partial Results 的数量
  - Scan Keys Num：每个 Query 扫描的 Key 的数量
  - Scan Keys Partial Num：每一个 Partial Result 扫描的 Key 的数量
  - Partial Num：每个 SQL 语句 Partial Results 的数量

- KV Errors

  - KV Backoff Duration：KV 每个请求重试的总时间。TiDB 向 TiKV 的请求都有重试机制，这里统计的是向 TiKV 发送请求时遇到错误重试的总时间
  - TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
  - KV Backoff OPS：TiKV 返回错误信息的数量
  - Lock Resolve OPS：TiDB 清理锁操作的数量。当 TiDB 的读写请求遇到锁时，会尝试进行锁清理
  - Other Errors OPS：其他类型的错误数量，包括清锁和更新 SafePoint

- KV Request

  - KV Request OPS：KV Request 根据 TiKV 显示执行次数
  - KV Request Duration 99 by store：根据 TiKV 显示 KV Request 执行时间

  - KV Request Duration 99 by type：根据类型显示 KV Request 的执行时间

- PD Client
  - PD Client CMD OPS：PD Client 每秒执行命令的数量
  - PD Client CMD Duration：PD Client 执行命令耗时
  - PD Client CMD Fail OPS：PD Client 每秒执行命令失败的数量
  - PD TSO OPS：TiDB 每秒从 PD 获取 TSO 的数量
  - PD TSO Wait Duration：TiDB 等待从 PD 返回 TSO 的时间
  - PD TSO RPC Duration：TiDB 从向 PD 发送获取 TSO 的请求到接收到 TSO 的耗时
  - Start TSO Wait Duration：TiDB 从向 PD 发送获取 start tso 请求开始到开始等待 tso 返回的时间

- Schema Load
  - Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
  - Load Schema OPS：TiDB 从 TiKV 每秒获取 Schema 的数量
  - Schema Lease Error OPM：Schema Lease 出错统计，包括 change 和 outdate 两种，change 代表 schema 发生了变化，outdate 代表无法更新 schema，属于较严重错误，出现 outdate 错误时会报警
  - Load Privilege OPS：TiDB 从 TiKV 每秒获取权限信息的数量

- DDL
  - DDL Duration 95：DDL 语句处理时间的 95% 分位
  - Batch Add Index Duration 100：创建索引时每个 Batch 所花费的最大时间
  - DDL Waiting Jobs Count：等待的 DDL 任务数量
  - DDL META OPM：DDL 每分钟获取 META 的次数
  - DDL Worker Duration 99：每个 DDL worker 执行时间的 99% 分位
  - Deploy Syncer Duration：Schema Version Syncer 初始化，重启，清空等操作耗时
  - Owner Handle Syncer Duration：DDL Owner 在执行更新，获取以及检查 Schema Version 的耗时
  - Update Self Version Duration：Schema Version Syncer 更新版本信息耗时
  - DDL OPM：DDL 语句的每秒执行次数
  - DDL add index progress in percentage：添加索引的进度展示

- Statistics
  - Auto Analyze Duration 95：自动 ANALYZE 耗时
  - Auto Analyze QPS：自动 ANALYZE 数量
  - Stats Inaccuracy Rate：统计信息不准确度
  - Pseudo Estimation OPS：使用假的统计信息优化 SQL 的数量
  - Dump Feedback OPS：存储统计信息 Feedback 的数量
  - Store Query Feedback QPS：存储合并查询的 Feedback 信息的每秒操作数量，该操作在 TiDB 内存中进行
  - Significant Feedback：重要的 Feedback 更新统计信息的数量
  - Update Stats OPS：利用 Feedback 更新统计信息的数量
  - Fast Analyze Status 100：快速收集统计信息的状态

- Owner
  - New ETCD Session Duration 95：创建一个新的 etcd 会话花费的时间。TiDB 通过 etcd client 连接 PD 中的 etcd 保存/读取部分元数据信息。这里记录了创建会话花费的时间
  - Owner Watcher OPS：DDL owner watch PD 的 etcd 的元数据的 goroutine 的每秒操作次数

- Meta

  - AutoID QPS：AutoID 相关操作的数量统计，包括全局 ID 分配、单个 Table AutoID 分配、单个 Table AutoID Rebase 三种操作
  - AutoID Duration：AutoID 相关操作的耗时
  - Region Cache Error OPS：TiDB 缓存的 region 信息每秒遇到的错误次数
  - Meta Operations Duration 99：元数据操作延迟

- GC
  - Worker Action OPM：GC 相关操作的数量，包括 run_job，resolve_lock，delete_range 等操作
  - Duration 99：GC 相关操作的耗时
  - Config：GC 的数据保存时长 (life time) 和 GC 运行间隔 (run interval) 配置
  - GC Failure OPM：GC 相关操作失败的数量
  - Delete Range Failure OPM：Delete range 失败的次数
  - Too Many Locks Error OPM：GC 清锁过多错误的数量
  - Action Result OPM：GC 相关操作结果数量
  - Delete Range Task Status：Delete range 的任务状态，包含完成和失败状态
  - Push Task Duration 95：将 GC 子任务推送给 GC worker 的耗时

- Batch Client
  - Pending Request Count by TiKV：TiKV 批量消息处理的等待数量
  - Wait Duration 95: 批量消息处理的等待时间。
  - Batch Client Unavailable Duration 95：批处理客户端的不可用时长。
  - No Available Connection Counter：批处理客户端不可用的连接数。

11.8.4 PD 重要监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

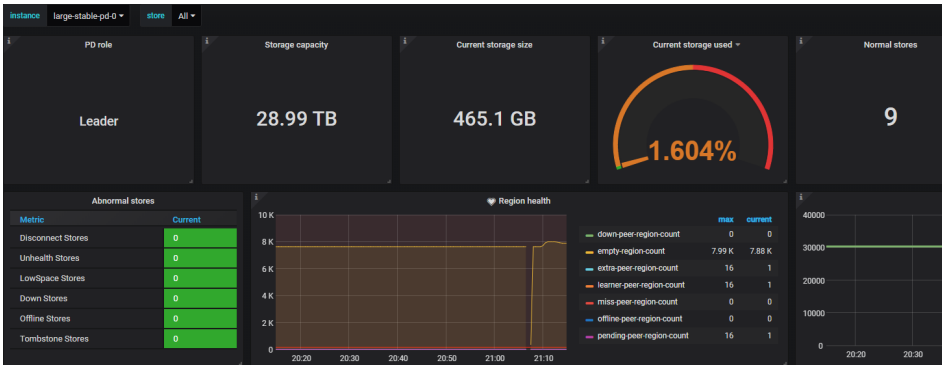
目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

对于日常运维，我们通过观察 PD 面板上的 Metrics，可以了解 PD 当前的状态。

以下为 PD Dashboard 监控说明：

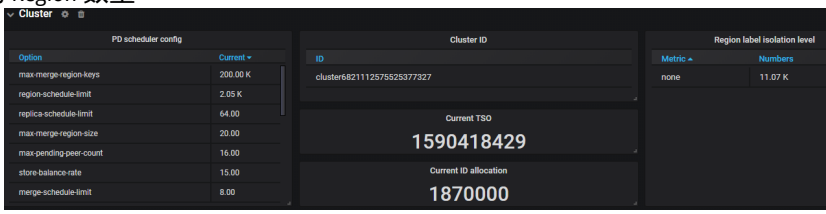
- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小
- Current storage usage：TiDB 集群存储空间的使用率
- Normal stores：处于正常状态的节点数目
- Number of Regions：当前集群的 Region 总量
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Region health：集群所有 Region 的状态。通常情况下，pending 或 down 的 peer 应该少于 100，miss 的 peer 不能一直大于 0，empty Region 过多需及时打开 Region Merge

- Current peer count：当前集群 peer 的总量



11.8.4.1 Cluster

- PD scheduler config：PD 调度配置列表
- Cluster ID：集群的 cluster id，唯一标识
- Current TSO：当前分配 TSO 的物理时间戳部分
- Current ID allocation：当前可分配 ID 的最大值
- Region label isolation level：不同 label 所在的 level 的 Region 数量
- Label distribution：集群中 TiKV 节点的 label 分布情况



11.8.4.2 Operator

- Schedule operator create：新创建的不同 operator 的数量，单位 opm 代表一分钟内创建的个数
- Schedule operator check：已检查的 operator 的次数，主要检查是否当前步骤已经执行完成，如果是，则执行下一个步骤
- Schedule operator finish：已完成调度的 operator 的数量
- Schedule operator timeout：已超时的 operator 的数量
- Schedule operator replaced or canceled：已取消或者被替换的 operator 的数量
- Schedule operators count by state：不同状态的 operator 的数量
- Operator finish duration：已完成的 operator 所花费的最长时间
- Operator step duration：已完成的 operator 的步骤所花费的最长时间

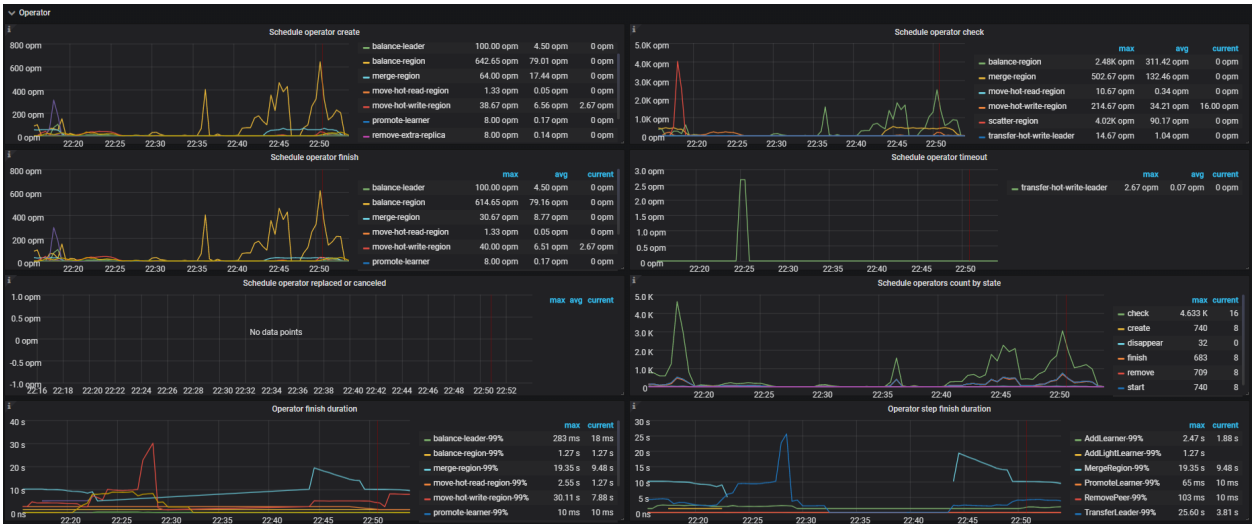


图 151: PD Dashboard - Operator metrics

11.8.4.3 Statistics - Balance

- Store capacity：每个 TiKV 实例的总的空间大小
- Store available：每个 TiKV 实例的可用空间大小
- Store used：每个 TiKV 实例的已使用空间大小
- Size amplification：每个 TiKV 实例的空间放大比率
- Size available ratio：每个 TiKV 实例的可用空间比率
- Store leader score：每个 TiKV 实例的 leader 分数
- Store Region score：每个 TiKV 实例的 Region 分数
- Store leader size：每个 TiKV 实例上所有 leader 的大小
- Store Region size：每个 TiKV 实例上所有 Region 的大小
- Store leader count：每个 TiKV 实例上所有 leader 的数量
- Store Region count：每个 TiKV 实例上所有 Region 的数量



图 152: PD Dashboard - Balance metrics

11.8.4.4 Statistics - hot write

- Hot Region’s leader distribution：每个 TiKV 实例上成为写入热点的 leader 的数量
- Total written bytes on hot leader Regions：每个 TiKV 实例上所有成为写入热点的 leader 的总的写入流量大小
- Hot write Region’s peer distribution：每个 TiKV 实例上成为写入热点的 peer 的数量
- Total written bytes on hot peer Regions：每个 TiKV 实例上所有成为写入热点的 peer 的写入流量大小
- Store Write rate bytes：每个 TiKV 实例总的写入的流量
- Store Write rate keys：每个 TiKV 实例总的写入 keys
- Hot cache write entry number：每个 TiKV 实例进入热点统计模块的 peer 的数量
- Selector events：热点调度中选择器的事件发生次数
- Direction of hotspot move leader：热点调度中 leader 的调度方向，正数代表调入，负数代表调出
- Direction of hotspot move peer：热点调度中 peer 的调度方向，正数代表调入，负数代表调出



图 153: PD Dashboard - Hot write metrics

11.8.4.5 Statistics - hot read

- Hot Region’s peer distribution：每个 TiKV 实例上成为读取热点的 peer 的数量
- Total read bytes on hot peer Regions：每个 TiKV 实例上所有成为读取热点的 peer 的总的读取流量大小
- Store read rate bytes：每个 TiKV 实例总的读取的流量
- Store read rate keys：每个 TiKV 实例总的读取 keys
- Hot cache read entry number：每个 TiKV 实例进入热点统计模块的 peer 的数量



图 154: PD Dashboard - Hot read metrics

11.8.4.6 Scheduler

- Scheduler is running：所有正在运行的 scheduler
- Balance leader movement：leader 移动的详细情况
- Balance Region movement：Region 移动的详细情况
- Balance leader event：balance leader 的事件数量
- Balance Region event：balance Region 的事件数量
- Balance leader scheduler：balance-leader scheduler 的状态
- Balance Region scheduler：balance-region scheduler 的状态
- Replica checker：replica checker 的状态
- Rule checker：rule checker 的状态
- Region merge checker：merge checker 的状态
- Filter target：尝试选择 Store 作为调度 taget 时没有通过 Filter 的计数
- Filter source：尝试选择 Store 作为调度 source 时没有通过 Filter 的计数
- Balance Direction：Store 被选作调度 target 或 source 的次数
- Store Limit：Store 的调度限流状态

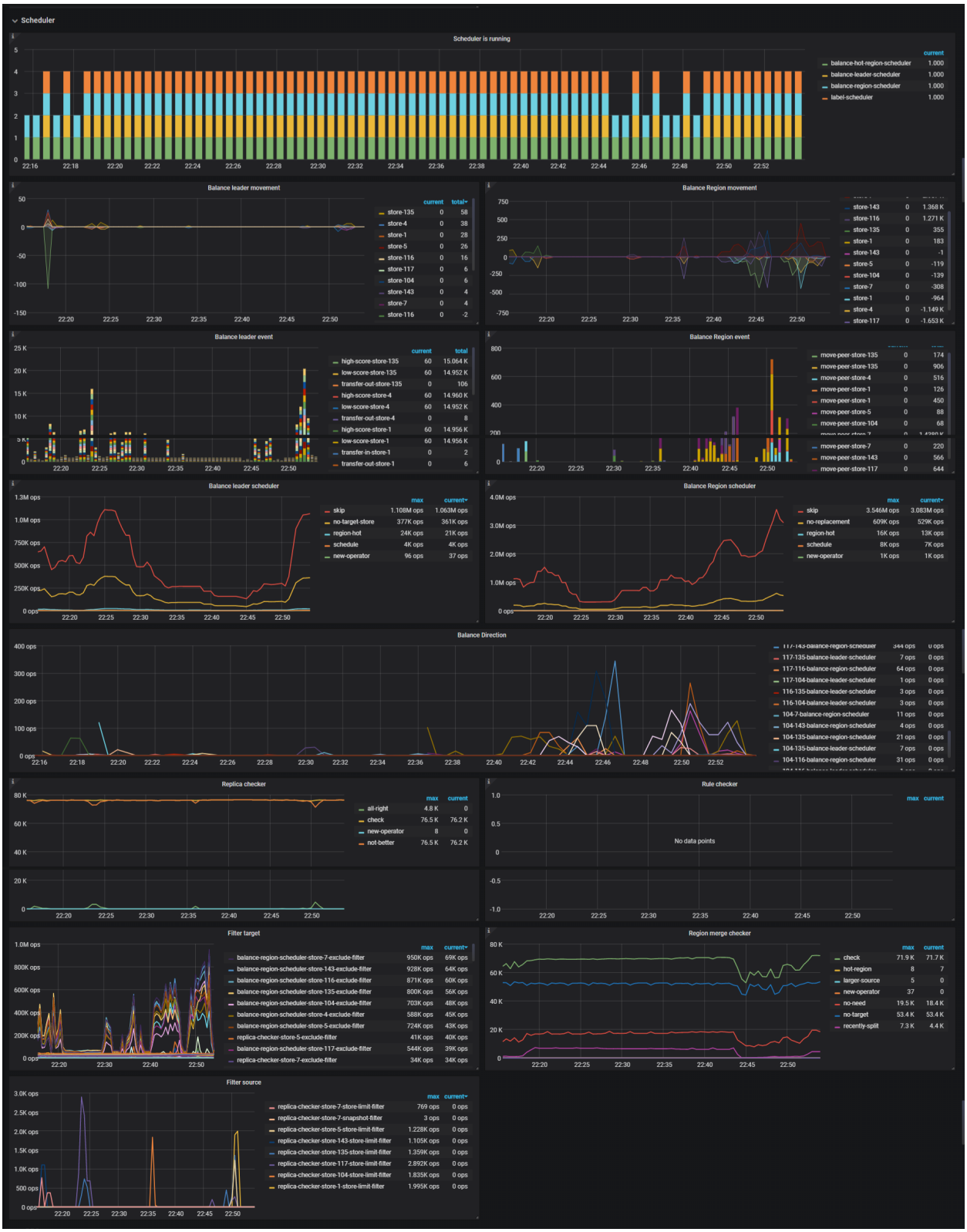


图 155: PD Dashboard - Scheduler metrics

11.8.4.7 gRPC

- Completed commands rate：gRPC 命令的完成速率
- 99% Completed commands duration：99% 命令的最长消耗时间



图 156: PD Dashboard - gRPC metrics

11.8.4.8 etcd

- Handle transactions count：etcd 的事务个数
- 99% Handle transactions duration：99% 的情况下，处理 etcd 事务所需花费的时间
- 99% WAL fsync duration：99% 的情况下，持久化 WAL 所需花费的时间，这个值通常应该小于 1s
- 99% Peer round trip time seconds：99% 的情况下，etcd 的网络延时，这个值通常应该小于 1s
- etcd disk WAL fsync rate：etcd 持久化 WAL 的速率
- Raft term：当前 Raft 的 term
- Raft committed index：最后一次 commit 的 Raft index
- Raft applied index：最后一次 apply 的 Raft index



图 157: PD Dashboard - etcd metrics

11.8.4.9 TiDB

- PD Server TSO handle time and Client recv time：从 PD 开始处理 TSO 请求到 client 端接收到 TSO 的总耗时
- Handle requests count：TiDB 的请求数量
- Handle requests duration：每个请求所花费的时间，99% 的情况下，应该小于 100ms

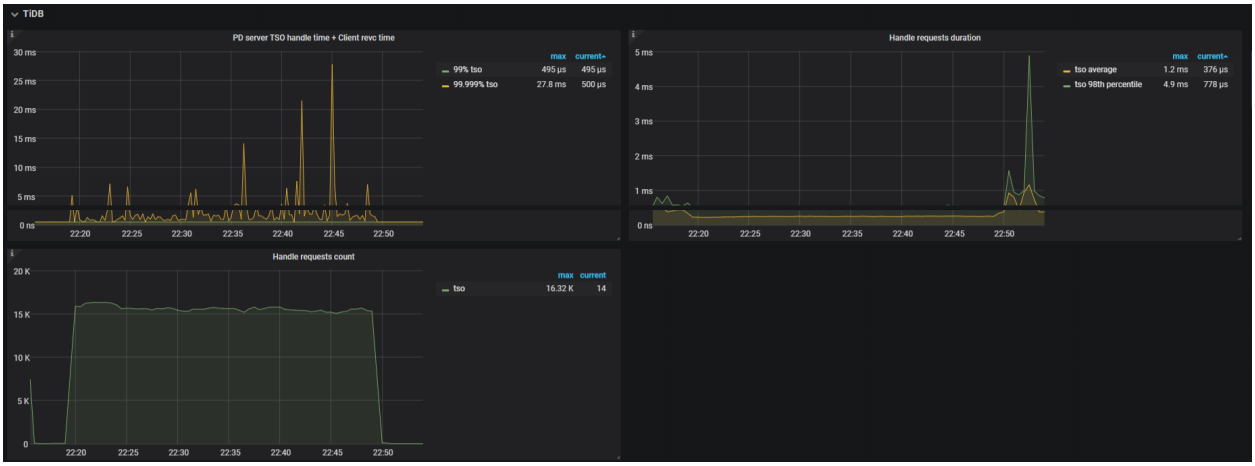


图 158: PD Dashboard - TiDB metrics

11.8.4.10 Heartbeat

- Heartbeat region event QPS：心跳处理 region 的 QPS，包括更新缓存和持久化
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- Region heartbeat report error：TiKV 向 PD 发送的异常的心跳个数
- Region heartbeat report active：TiKV 向 PD 发送的正常的心跳个数
- Region schedule push：PD 向 TiKV 发送的调度命令的个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

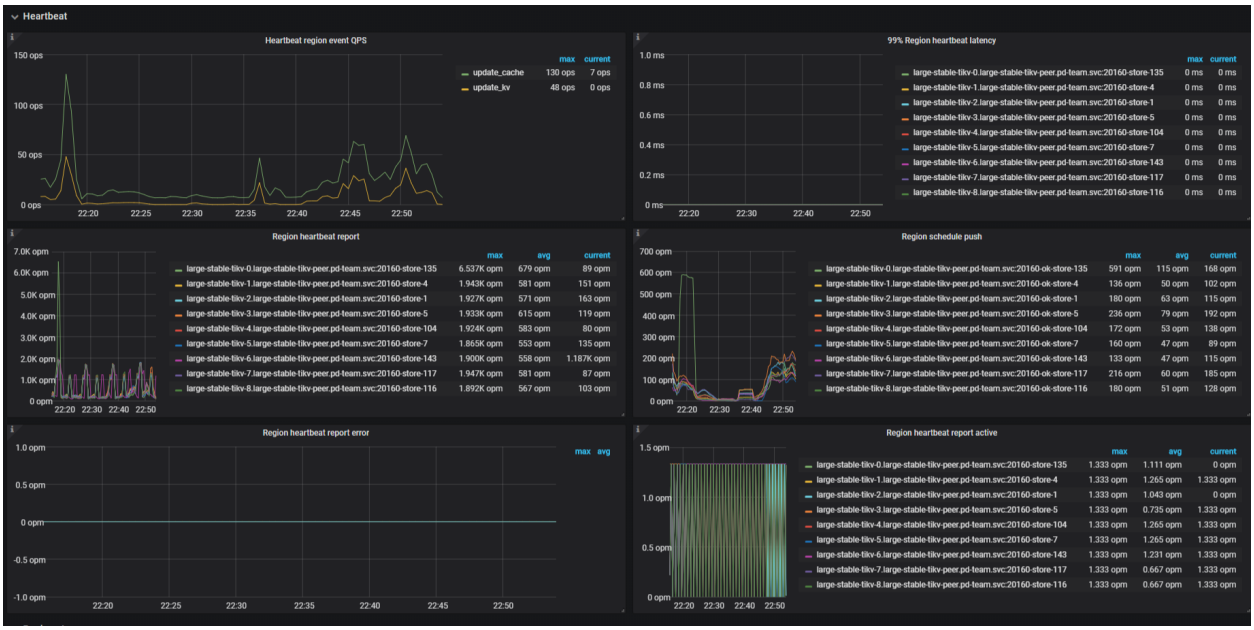


图 159: PD Dashboard - Heartbeat metrics

11.8.4.11 Region storage

- Syncer Index：Leader 记录 Region 变更历史的最大 index
- history last index：Follower 成功同步的 Region 变更历史的 index



图 160: PD Dashboard - Region storage

11.8.5 TiKV 监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

对于日常运维，我们通过观察 TiKV-Details 面板上的指标，可以了解 TiKV 当前的状态。根据性能地图可以检查集群的状态是否符合预期。

以下为 TiKV-Details 默认的监控信息：

11.8.5.1 Cluster

- Store size：每个 TiKV 实例的使用的存储空间的大小
- Available size：每个 TiKV 实例的可用的存储空间的大小
- Capacity size：每个 TiKV 实例的存储容量的大小
- CPU：每个 TiKV 实例 CPU 的使用率
- Memory：每个 TiKV 实例内存的使用情况
- IO utilization：每个 TiKV 实例 IO 的使用率
- MBps：每个 TiKV 实例写入和读取的数据量大小
- QPS：每个 TiKV 实例上各种命令的 QPS
- Errps：每个 TiKV 实例上 gRPC 消息失败的速率
- leader：每个 TiKV 实例 leader 的个数
- Region：每个 TiKV 实例 Region 的个数
- Uptime：自上次重启以来 TiKV 正常运行的时间

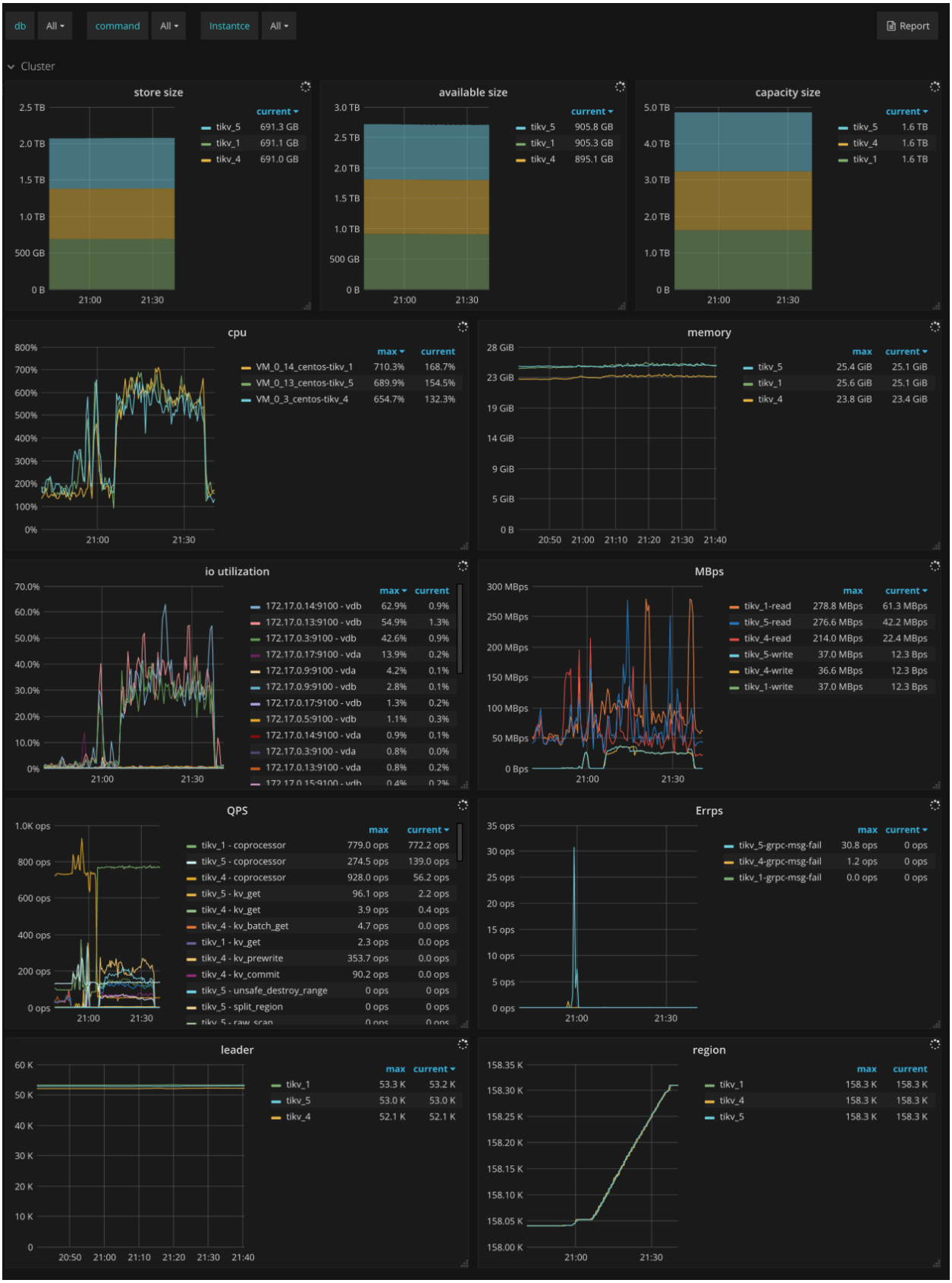


图 161: TiKV Dashboard - Cluster metrics

11.8.5.2 Errors

- Critical error：严重错误的数量
- Server is busy：各种会导致 TiKV 实例暂时不可用的事件个数，如 write stall，channel full 等，正常情况下应当为 0
- Server report failures：server 报错的消息个数，正常情况下应当为 0
- Raftstore error：每个 TiKV 实例上 raftstore 发生错误的个数
- Scheduler error：每个 TiKV 实例上 scheduler 发生错误的个数
- Coprocessor error：每个 TiKV 实例上 coprocessor 发生错误的个数
- gRPC message error：每个 TiKV 实例上 gRPC 消息发生错误的个数
- Leader drop：每个 TiKV 实例上 drop leader 的个数
- Leader missing：每个 TiKV 实例上 missing leader 的个数
- Log Replication Rejected：每个 TiKV 实例上由于内存不足而拒绝 logappend 消息的个数



图 162: TiKV Dashboard - Errors metrics

11.8.5.3 Server

- CF size：每个列族的大小
- Store size：每个 TiKV 实例的使用的存储空间的大小
- Channel full：每个 TiKV 实例上 channel full 错误的数量，正常情况下应当为 0
- Active written leaders：各个 TiKV 实例中正在被写入的 Leader 的数量
- Approximate Region size：每个 Region 近似的大小
- Approximate Region size Histogram：每个 Region 近似大小的直方图
- Region average written keys：每个 TiKV 实例上所有 Region 的平均 key 写入个数
- Region average written bytes：每个 TiKV 实例上所有 Region 的平均写入大小

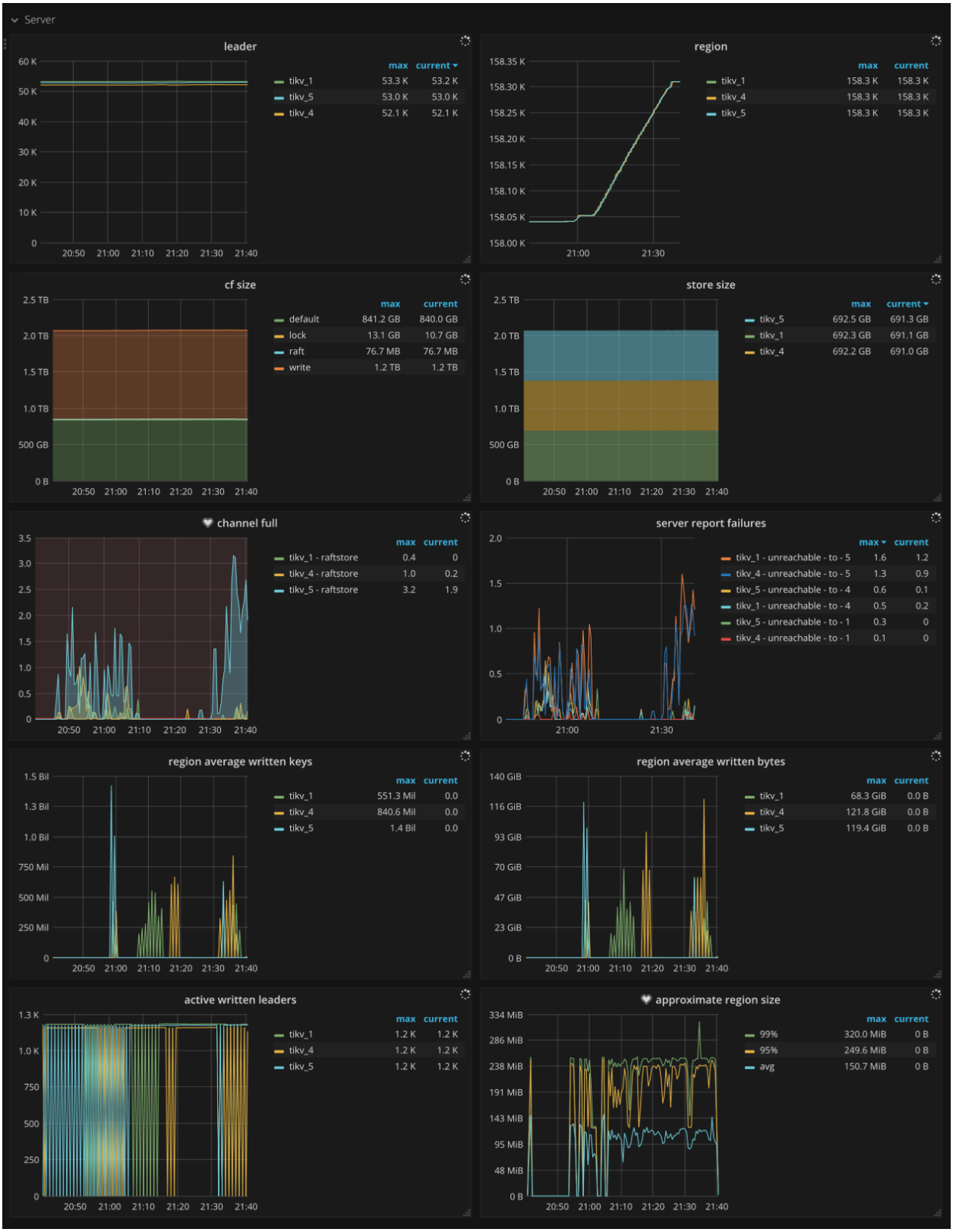


图 163: TiKV Dashboard - Server metrics

11.8.5.4 gRPC

- gRPC message count：每种 gRPC 请求的速度
- gRPC message failed：失败的 gRPC 请求的速度
- 99% gRPC message duration：99% gRPC 请求的执行时间小于该值
- Average gRPC message duration：gRPC 请求平均的执行时间
- gRPC batch size：TiDB 与 TiKV 之间 grpc 请求的 batch 大小
- raft message batch size：TiKV 与 TiKV 之间 raft 消息的 batch 大小

11.8.5.5 Thread CPU

- Raft store CPU：raftstore 线程的 CPU 使用率，通常应低于 80% * `raftstore.store-pool-size`
- Async apply CPU：async apply 线程的 CPU 使用率，通常应低于 90% * `raftstore.apply-pool-size`
- Scheduler worker CPU：scheduler worker 线程的 CPU 使用率，通常应低于 90% * `storage.scheduler-worker-pool-size`
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80% * `server.grpc-concurrency`
- Unified read pool CPU：unified read pool 线程的 CPU 使用率
- Storage ReadPool CPU：storage read pool 线程的 CPU 使用率
- Coprocessor CPU：coprocessor 线程的 CPU 使用率
- RocksDB CPU：RocksDB 线程的 CPU 使用率
- GC worker CPU：GC worker 线程的 CPU 使用率
- BackGround worker CPU：background worker 线程的 CPU 使用率

11.8.5.6 PD

- PD requests：TiKV 发送给 PD 的请求速度
- PD request duration (average)：TiKV 发送给 PD 的请求处理的平均时间
- PD heartbeats：发送给 PD 的心跳的速度
- PD validate peers：TiKV 发送给 PD 用于验证 TiKV 的 peer 有效的消息的速度

11.8.5.7 Raft IO

- Apply log duration：Raft apply 日志所花费的时间
- Apply log duration per server：每个 TiKV 实例上 Raft apply 日志所花费的时间
- Append log duration：Raft append 日志所花费的时间
- Append log duration per server：每个 TiKV 实例上 Raft append 日志所花费的时间
- Commit log duration：Raft commit 日志所花费的时间
- Commit log duration per server：每个 TiKV 实例上 Raft commit 日志所花费的时间

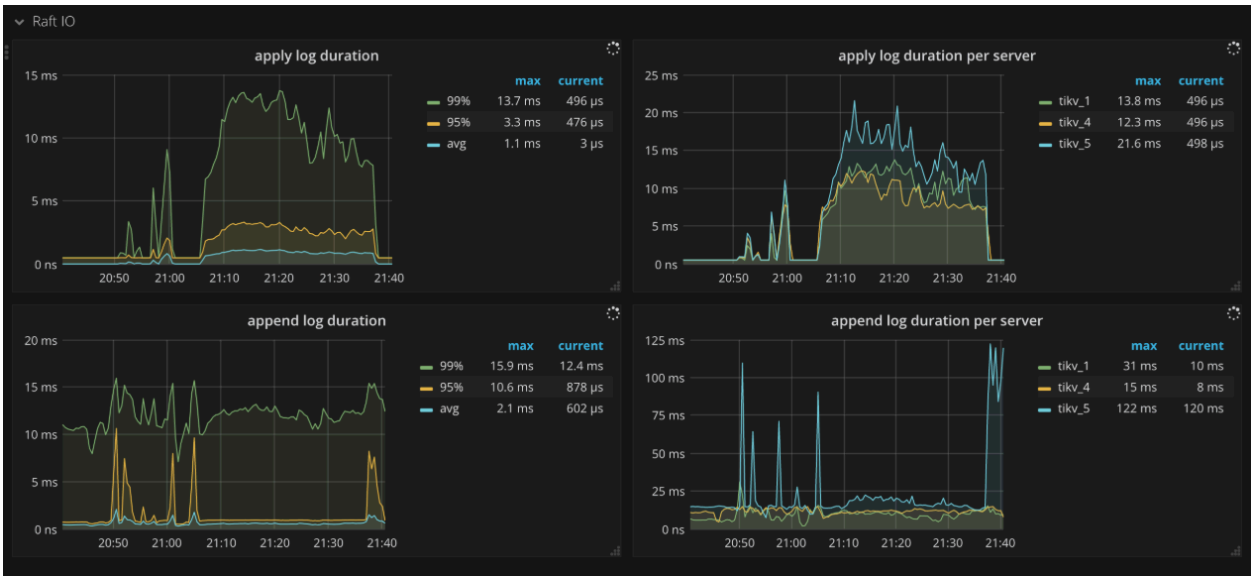


图 164: TiKV Dashboard - Raft IO metrics

11.8.5.8 Raft process

- Ready handled：Raft 中不同 ready 类型的 ops
  - count: 批量处理 ready 的 ops
  - has_ready_region: 获得 ready 的 Region 的 ops
  - pending_region: 被检查是否获得 ready 的 Region 的 ops，v3.0.0 后废弃
  - message: ready 内待发送 message 的 ops
  - append: ready 内 Raft log entry 的 ops
  - commit: ready 内 committed Raft log entry 的 ops
  - snapshot: 携带 snapshot 的 ready 的 ops
- 0.99 Duration of Raft store events：99% 的 raftstore 事件所花费的时间
- Process ready duration：处理 ready 所花费的时间
- Process ready duration per server：每个 TiKV 实例处理 ready 所花费的时间，99.99% 的情况下，应该小于 2s



图 165: TiKV Dashboard - Raft process metrics

11.8.5.9 Raft message

- Sent messages per server：每个 TiKV 实例发送 Raft 消息的 ops
- Flush messages per server：每个 TiKV 实例中 raft client 往外 flush Raft 消息的 ops
- Receive messages per server：每个 TiKV 实例接受 Raft 消息的 ops
- Messages：发送不同类型的 Raft 消息的 ops
- Vote：Raft 投票消息发送的 ops
- Raft dropped messages：每秒钟丢弃不同类型的 Raft 消息的个数

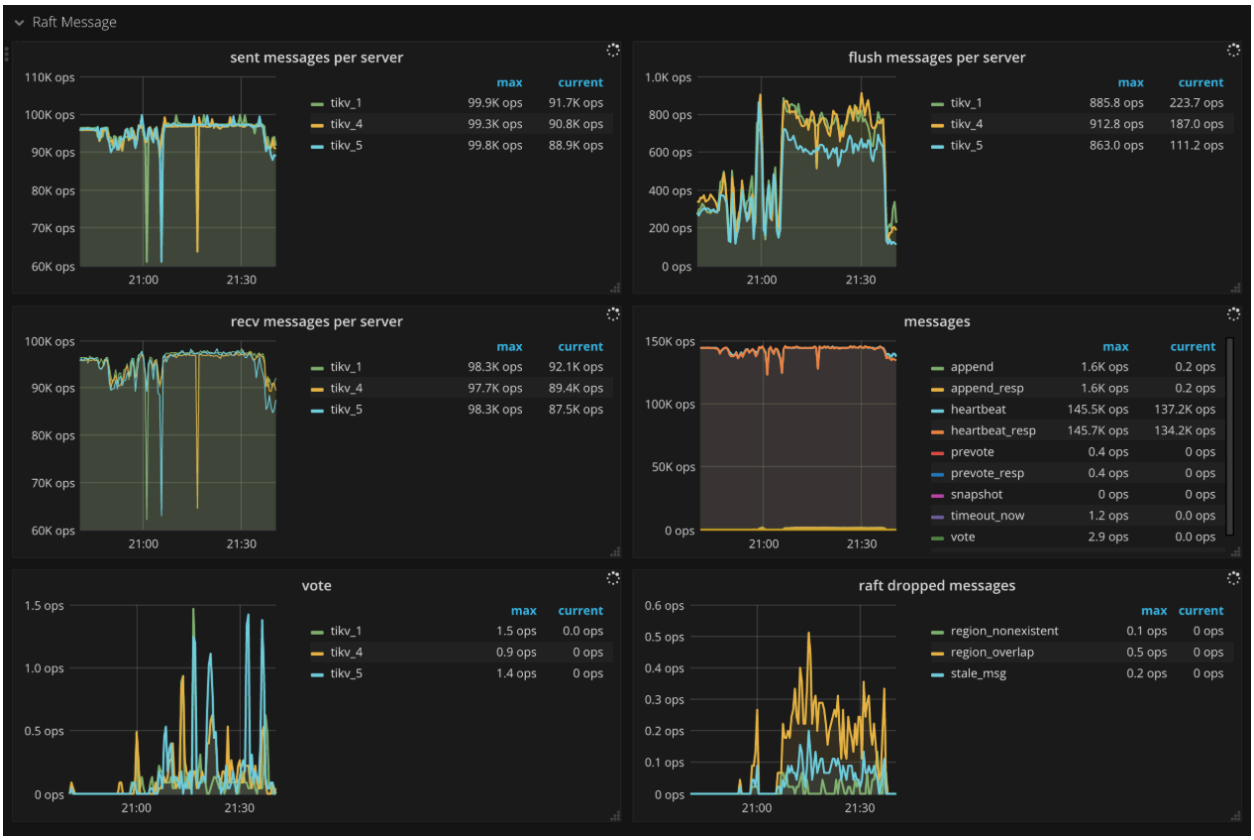


图 166: TiKV Dashboard - Raft message metrics

11.8.5.10 Raft propose

- Raft apply proposals per ready：在一个 batch 内，apply proposal 时每个 ready 中包含 proposal 的个数的直方图
- Raft read/write proposals：不同类型的 proposal 的 ops
- Raft read proposals per server：每个 TiKV 实例发起读 proposal 的 ops
- Raft write proposals per server：每个 TiKV 实例发起写 proposal 的 ops
- Propose wait duration：proposal 的等待时间的直方图
- Propose wait duration per server：每个 TiKV 实例上每个 proposal 的等待时间的直方图
- Apply wait duration：apply 的等待时间的直方图
- Apply wait duration per server：每个 TiKV 实例上每个 apply 的等待时间的直方图
- Raft log speed：peer propose 日志的平均速度

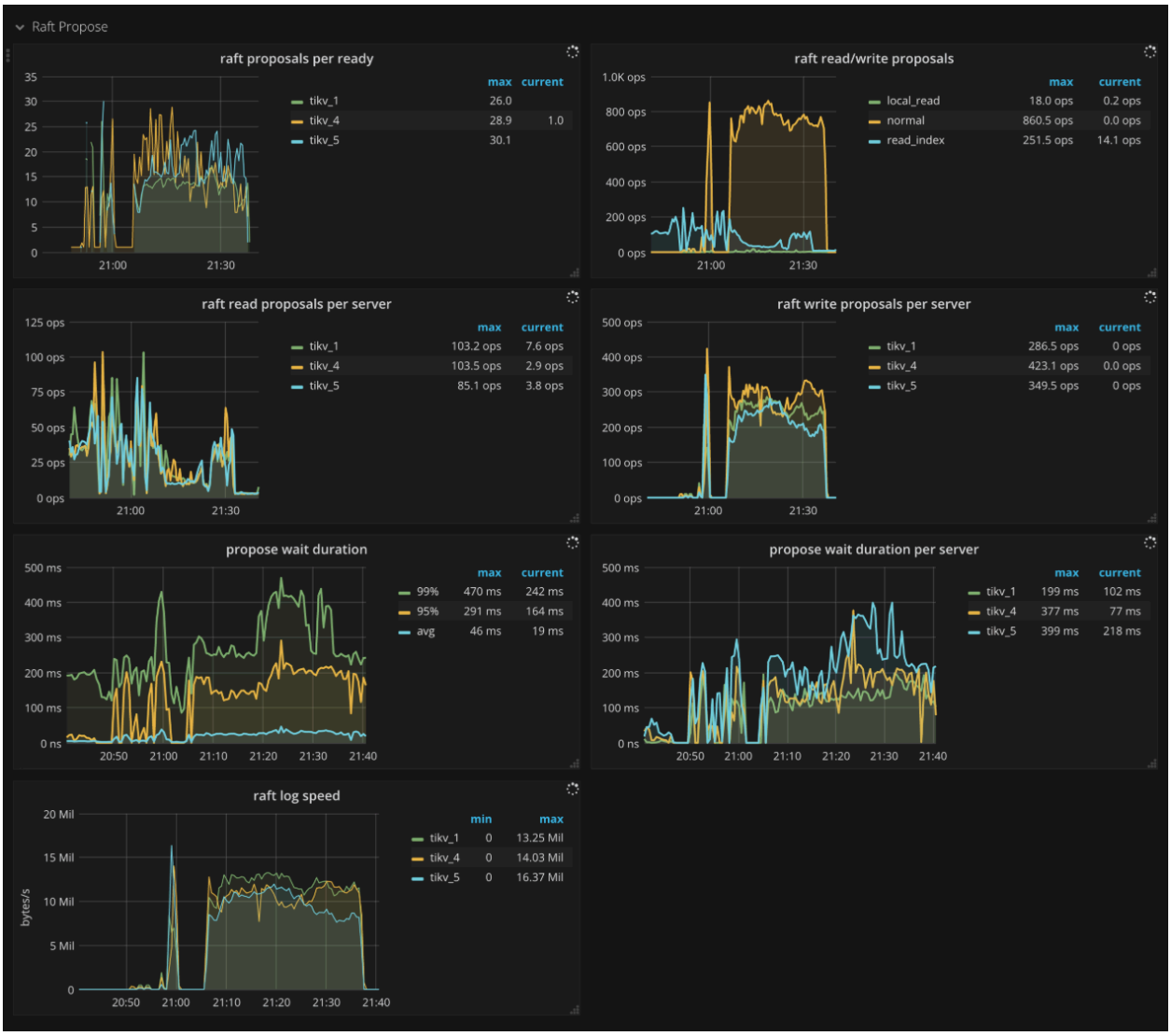


图 167: TiKV Dashboard - Raft propose metrics

11.8.5.11 Raft admin

- Admin proposals：admin proposal 的 ops
- Admin apply：apply 命令的 ops
- Check split：split check 命令的 ops
- 99.99% Check split duration：99.99% 的情况下，split check 所需花费的时间

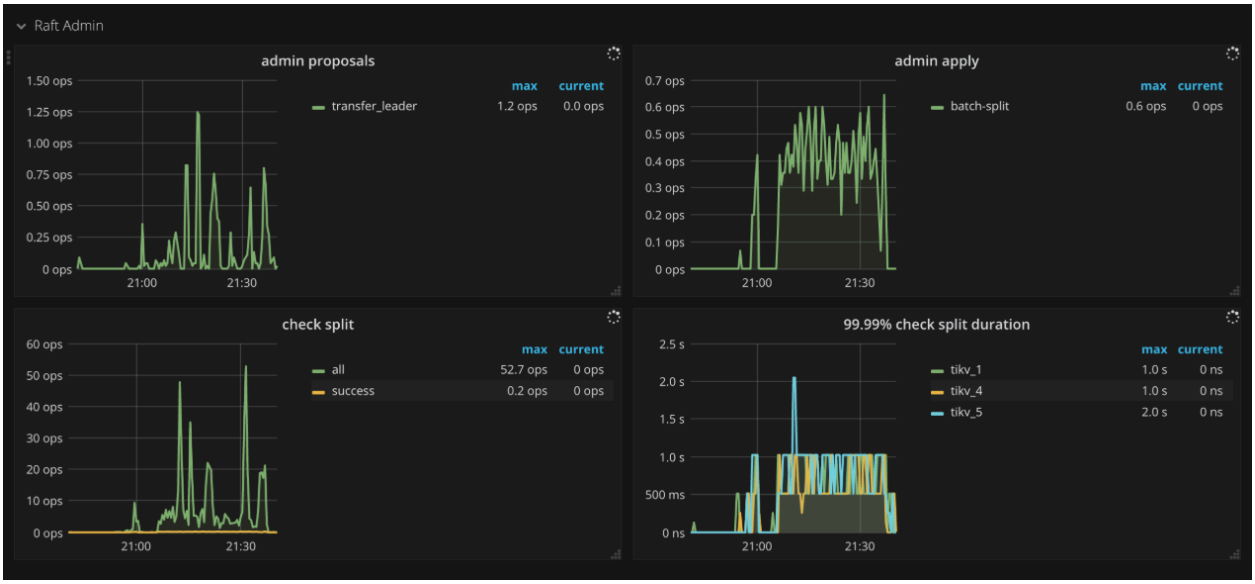


图 168: TiKV Dashboard - Raft admin metrics

11.8.5.12 Local reader

- Local reader requests：所有请求的总数以及 local read 线程拒绝的请求数量



图 169: TiKV Dashboard - Local reader metrics

11.8.5.13 Unified Read Pool

- Time used by level：在 unified read pool 中每个级别使用的时间，级别 0 指小查询

- Level 0 chance：在 unified read pool 中调度的 level 0 任务的比例
- Running tasks：在 unified read pool 中并发运行的任务数量

11.8.5.14 Storage

- Storage command total：收到不同命令的 ops
- Storage async request error：异步请求出错的 ops
- Storage async snapshot duration：异步处理 snapshot 所花费的时间，99% 的情况下，应该小于 1s
- Storage async write duration：异步写所花费的时间，99% 的情况下，应该小于 1s

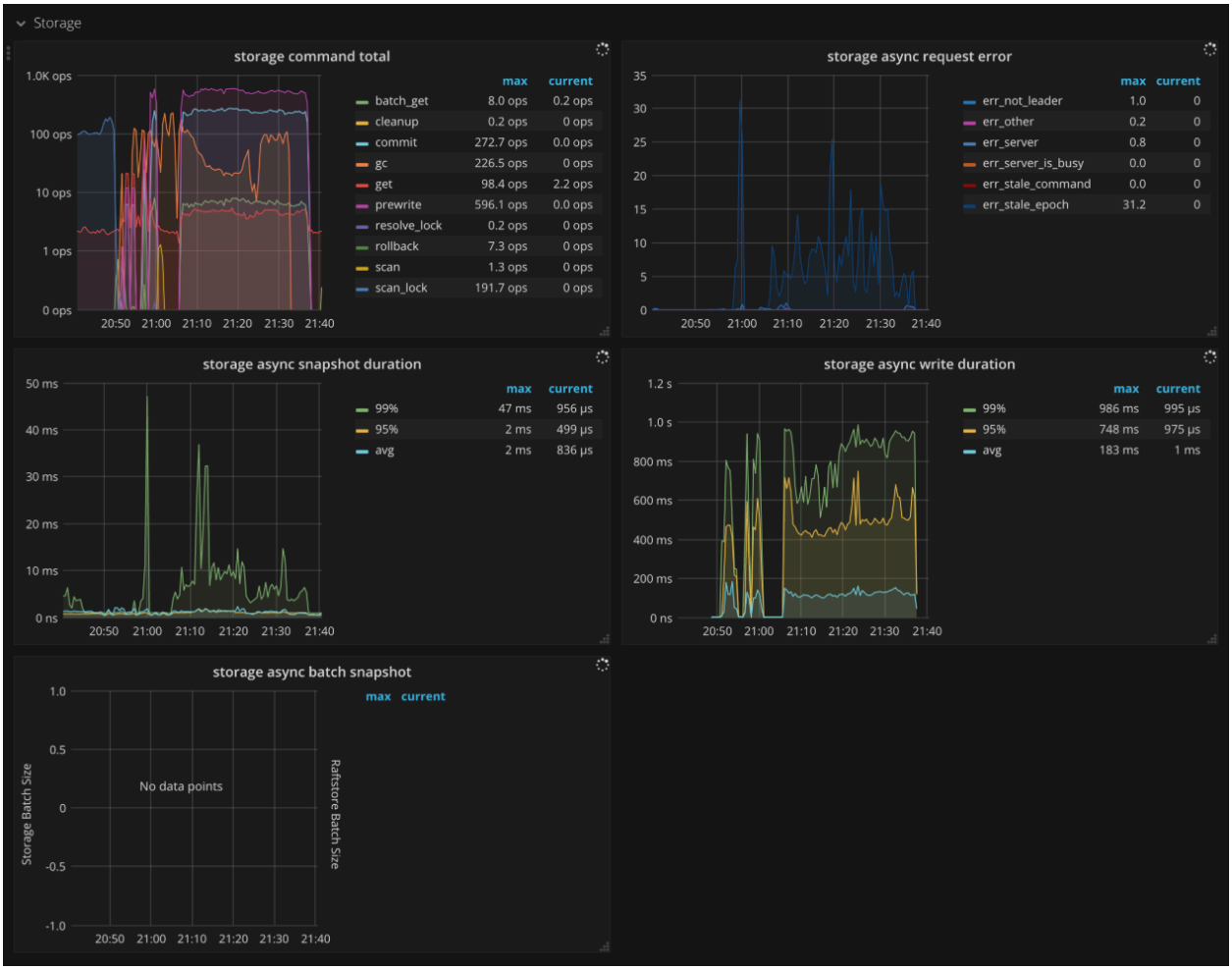


图 170: TiKV Dashboard - Storage metrics

11.8.5.15 Scheduler

- Scheduler stage total：每种命令不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler writing bytes：每个 TiKV 实例正在处理的命令的写入字节数量
- Scheduler priority commands：不同优先级命令的 ops

- Scheduler pending commands：每个 TiKV 实例上 pending 命令的 ops

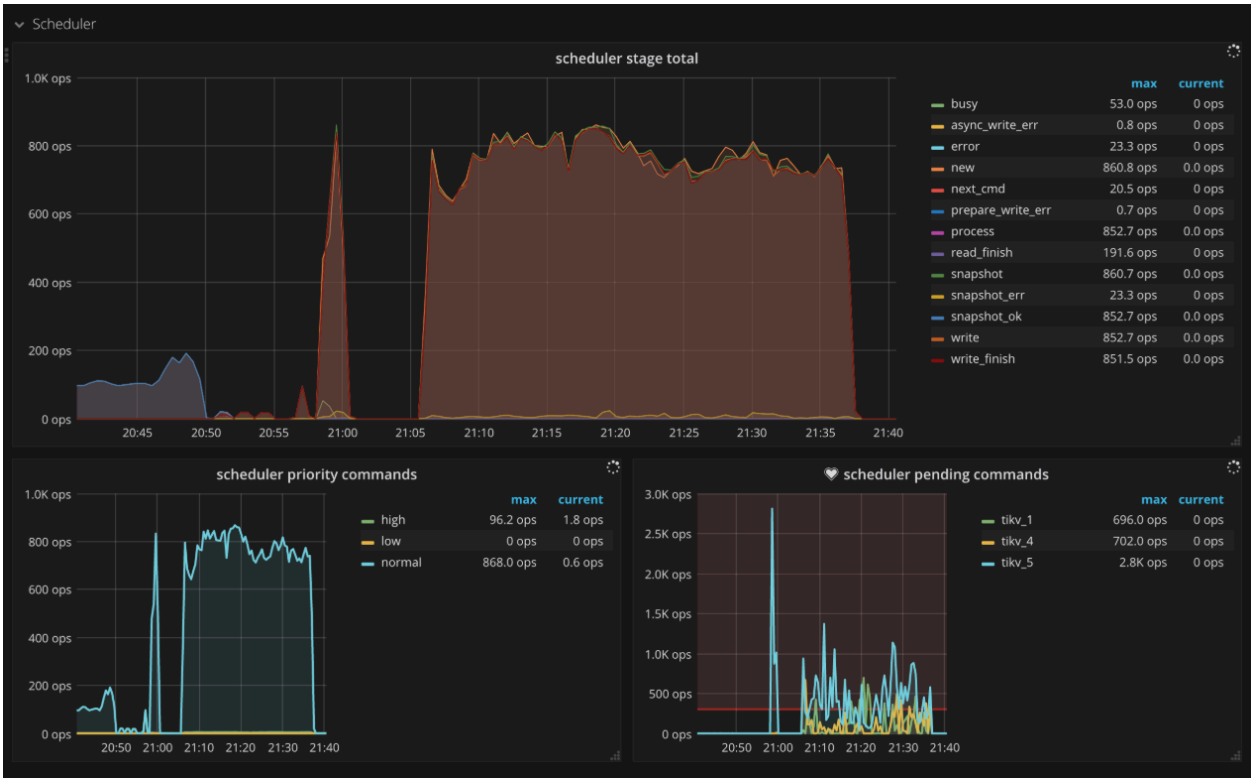


图 171: TiKV Dashboard - Scheduler metrics

11.8.5.16 Scheduler - commit

- Scheduler stage total：commit 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 commit 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：commit 命令读取 key 的个数
- Scheduler keys written：commit 命令写入 key 的个数
- Scheduler scan details：执行 commit 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 commit 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 commit 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 commit 命令时，扫描每个 default CF 中 key 的详细情况



图 172: TiKV Dashboard - Scheduler commit metrics

11.8.5.17 Scheduler - pessimistic_rollback

- Scheduler stage total：pessimistic_rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 pessimistic_rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：pessimistic_rollback 命令读取 key 的个数
- Scheduler keys written：pessimistic_rollback 命令写入 key 的个数

- Scheduler scan details：执行 pessimistic_rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 pessimistic_rollback 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 pessimistic_rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 pessimistic_rollback 命令时，扫描每个 default CF 中 key 的详细情况

11.8.5.18 Scheduler - prewrite

- Scheduler stage total：prewrite 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 prewrite 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：prewrite 命令读取 key 的个数
- Scheduler keys written：prewrite 命令写入 key 的个数
- Scheduler scan details：执行 prewrite 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 prewrite 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 prewrite 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 prewrite 命令时，扫描每个 default CF 中 key 的详细情况

11.8.5.19 Scheduler - rollback

- Scheduler stage total：rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：rollback 命令读取 key 的个数
- Scheduler keys written：rollback 命令写入 key 的个数
- Scheduler scan details：执行 rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 rollback 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 rollback 命令时，扫描每个 default CF 中 key 的详细情况

11.8.5.20 GC

- GC tasks：由 gc_worker 处理的 GC 任务的个数
- GC tasks Duration：执行 GC 任务时所花费的时间
- TiDB GC seconds：TiDB 执行 GC 花费的时间
- TiDB GC worker actions：TiDB GC worker 的不同 action 的个数
- TiKV AutoGC Working：Auto GC 管理器的工作状态
- ResolveLocks Progress：GC 第一阶段 (ResolveLocks) 的进度
- TiKV Auto GC Progress：GC 第二阶段的进度
- GC speed：GC 每秒删除的 key 的数量
- TiKV Auto GC SafePoint：TiKV GC 的 safe point 的数值，safe point 为当前 GC 的时间戳
- GC lifetime：TiDB 设置的 GC lifetime
- GC interval：TiDB 设置的 GC 间隔
- GC in Compaction Filter：write CF 的 Compaction Filter 中已过滤版本的数量

11.8.5.21 Snapshot

- Rate snapshot message：发送 Raft snapshot 消息的速率
- 99% Handle snapshot duration：99% 的情况下，处理 snapshot 所需花费的时间
- Snapshot state count：不同状态的 snapshot 的个数
- 99.99% Snapshot size：99.99% 的 snapshot 的大小
- 99.99% Snapshot KV count：99.99% 的 snapshot 包含的 key 的个数

11.8.5.22 Task

- Worker handled tasks：worker 每秒钟处理的任务的数量
- Worker pending tasks：当前 worker 中，每秒钟 pending 和 running 的任务的数量，正常情况下，应该小于 1000
- FuturePool handled tasks：future pool 每秒钟处理的任务的数量
- FuturePool pending tasks：当前 future pool 中，每秒钟 pending 和 running 的任务的数量

11.8.5.23 Coprocessor Overview

- Request duration：从收到 coprocessor 请求到处理结束所消耗的总时间
- Total Requests：每种类型的总请求的 ops
- Handle duration：每分钟实际处理 coprocessor 请求所消耗的时间的直方图
- Total Request Errors：Coprocessor 每秒请求错误的数量，正常情况下，短时间内不应该有大量的错误
- Total KV Cursor Operations：各种类型的 KV cursor 操作的总数量的 ops，例如 select、index、analyze_table、analyze_index、checksum_table、checksum_index 等
- KV Cursor Operations：每秒各种类型的 KV cursor 操作的数量，以直方图形式显示
- Total RocksDB Perf Statistics：RocksDB 性能统计数据
- Total Response Size：coprocessor 回应的数据大小

11.8.5.24 Coprocessor Detail

- Handle duration：每秒钟实际处理 coprocessor 请求所消耗的时间的直方图
- 95% Handle duration by store：每秒钟中 95% 的情况下，每个 TiKV 实例处理 coprocessor 请求所花费的时间
- Wait duration：coprocessor 每秒钟内请求的等待时间，99.99% 的情况下，应该小于 10s
- 95% Wait duration by store：每秒钟 95% 的情况下，每个 TiKV 实例上 coprocessor 请求的等待时间
- Total DAG Requests：DAG 请求的总数量的 ops
- Total DAG Executors：DAG executor 的总数量的 ops
- Total Ops Details (Table Scan)：coprocessor 中请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details (Index Scan)：coprocessor 中请求为 index 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Table Scan)：coprocessor 中对于每个 CF 请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Index Scan)：coprocessor 中对于每个 CF 请求为 index 的 scan 过程中每秒钟各种事件发生的次数

11.8.5.25 Threads

- Threads state：TiKV 线程的状态
- Threads IO：TiKV 各个线程的 I/O 流量
- Thread Voluntary Context Switches：TiKV 线程自主切换的次数
- Thread Nonvoluntary Context Switches：TiKV 线程被动切换的次数

11.8.5.26 RocksDB - kv/raft

- Get operations：get 操作的 ops
- Get duration：get 操作的耗时
- Seek operations：seek 操作的 ops
- Seek duration：seek 操作的耗时
- Write operations：write 操作的 ops
- Write duration：write 操作的耗时
- WAL sync operations：sync WAL 操作的 ops
- Write WAL duration：write 操作中写 WAL 的耗时
- WAL sync duration：sync WAL 操作的耗时
- Compaction operations：compaction 和 flush 操作的 ops
- Compaction duration：compaction 和 flush 操作的耗时
- SST read duration：读取 SST 所需的时间
- Write stall duration：由于 write stall 造成的时间开销，正常情况下应为 0
- Memtable size：每个 CF 的 memtable 的大小
- Memtable hit：memtable 的命中率
- Block cache size：block cache 的大小。如果将 `shared block cache` 禁用，即为每个 CF 的 block cache 的大小
- Block cache hit：block cache 的命中率
- Block cache flow：不同 block cache 操作的流量
- Block cache operations 不同 block cache 操作的个数
- Keys flow：不同操作造成的 key 的流量
- Total keys：每个 CF 中 key 的个数
- Read flow：不同读操作的流量
- Bytes/Read：每次读的大小
- Write flow：不同写操作的流量
- Bytes/Write：每次写的大小
- Compaction flow：compaction 相关的流量
- Compaction pending bytes：等待 compaction 的大小
- Read amplification：每个 TiKV 实例的读放大
- Compression ratio：每一层的压缩比
- Number of snapshots：每个 TiKV 的 snapshot 的数量
- Oldest snapshots duration：最旧的 snapshot 保留的时间
- Number files at each level：每一层的文件个数
- Ingest SST duration seconds：ingest SST 所花费的时间
- Stall conditions changed of each CF：每个 CF stall 的原因

11.8.5.27 Titan - All

- Blob file count：Titan blob 文件的数量
- Blob file size：Titan blob 文件总大小
- Live blob size：有效 blob record 的总大小
- Blob cache hit：Titan 的 blob cache 命中率
- Iter touched blob file count：单个 Iterator 所涉及到 blob 文件的数量
- Blob file discardable ratio distribution：blob 文件的失效 blob record 比例的分布情况
- Blob key size：Titan 中 blob key 的大小
- Blob value size：Titan 中 blob value 的大小
- Blob get operations：blob 的 get 操作的数量
- Blob get duration：blob 的 get 操作的耗时
- Blob iter operations：blob 的 iter 操作的耗时
- Blob seek duration：blob 的 seek 操作的耗时
- Blob next duration：blob 的 next 操作的耗时
- Blob prev duration：blob 的 prev 操作的耗时
- Blob keys flow：Titan blob 读写的 key 数量
- Blob bytes flow：Titan blob 读写的 bytes 数量
- Blob file read duration：blob 文件的读取耗时
- Blob file write duration：blob 文件的写入耗时
- Blob file sync operations：blob 文件 sync 次数
- Blob file sync duration：blob 文件 sync 耗时
- Blob GC action：Titan GC 细分动作的次数
- Blob GC duration：Titan GC 的耗时
- Blob GC keys flow：Titan GC 读写的 key 数量
- Blob GC bytes flow：Titan GC 读写的 bytes 数量
- Blob GC input file size：Titan GC 输入文件的大小
- Blob GC output file size：Titan GC 输出文件的大小
- Blob GC file count：Titan GC 涉及的 blob 文件数量

11.8.5.28 Pessimistic Locking

- Lock Manager Thread CPU：lock manager 的线程 CPU 使用率
- Lock Manager Handled tasks：lock manager 处理的任务数量
- Waiter lifetime duration：事务等待锁释放的时间
- Wait table：wait table 的状态信息，包括锁的数量和等锁事务的数量
- Deadlock detect duration：处理死锁检测请求的耗时
- Detect error：死锁检测遇到的错误数量，包含死锁的数量
- Deadlock detector leader：死锁检测器 leader 所在节点的信息
- Total pessimistic locks memory size：内存悲观锁占用内存的总大小
- In-memory pessimistic locking result：将悲观锁仅保存到内存的结果，其中 full 表示因为超过内存限制而无法将悲观锁保存至内存的次数

11.8.5.29 Memory

- Allocator Stats：内存分配器的统计信息

11.8.5.30 Backup

- Backup CPU：backup 的线程 CPU 使用率
- Range Size：backup range 的大小直方图
- Backup Duration：backup 的耗时
- Backup Flow：backup 总的字节大小
- Disk Throughput：实例磁盘的吞吐量
- Backup Range Duration：backup range 的耗时
- Backup Errors：backup 中发生的错误数量

11.8.5.31 Encryption

- Encryption data keys：正在使用的加密 data key 的总数量
- Encrypted files：被加密的文件数量
- Encryption initialized：显示加密是否被启用，1 代表已经启用
- Encryption meta files size：加密相关的元数据文件的大小
- Encrypt/decrypt data nanos：每次加密/解密数据的耗时的直方图
- Read/write encryption meta duration：每秒钟读写加密文件所耗费的时间

11.8.5.32 面板常见参数的解释

11.8.5.32.1 gRPC 消息类型

1. 使用事务型接口的命令：

    - kv_get：事务型的 get 命令，获取指定 ts 能读到的最新版本数据
    - kv_scan：扫描连续的一段数据
    - kv_prewrite：2PC 的第一阶段，预写入事务要提交的数据
    - kv_pessimistic_lock：对 key 加悲观锁，防止其他事务修改
    - kv_pessimistic_rollback：删除 key 上的悲观锁
    - kv_txn_heart_beat：更新悲观事务或大事务的 `lock_ttl` 以防止其被回滚
    - kv_check_txn_status：检查事务的状态
    - kv_commit：2PC 的第二阶段，提交 prewrite 阶段写入的数据
    - kv_cleanup：回滚一个事务（此命令将会在 4.0 中废除）
    - kv_batch_get：与 `kv_get` 类似，一次性获取批量 key 的 value
    - kv_batch_rollback：批量回滚多个预写的事务
    - kv_scan_lock：扫描所有版本号在 `max_version` 之前的锁，用于清理过期的事务
    - kv_resolve_lock：根据事务状态，提交或回滚事务的锁
    - kv_gc：触发垃圾回收
    - kv_delete_range：从 TiKV 中删除连续的一段数据

2. 非事务型的裸命令：

    - raw_get：获取 key 所对应的 value
    - raw_batch_get：获取一批 key 所对应的 value

- raw_scan：扫描一段连续的数据
- raw_batch_scan：扫描多段连续的数据
- raw_put：写入一个 key/value 对
- raw_batch_put：直接写入一批 key/value 对
- raw_delete：删除一个 key/value 对
- raw_batch_delete：删除一批 key/value 对
- raw_delete_range：删除连续的一段区间

11.8.6 TiFlash 集群监控

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

TiFlash 面板一共包括 TiFlash-Summary、 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details。通过面板上的指标，可以了解 TiFlash 当前的状态。其中 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details 主要为 TiFlash 的 Raft 层信息，其监控指标信息可参考TiKV 监控指标详解。

> 注意：
>
> 低版本的 TiFlash 监控信息较不完善，如有需要推荐使用 v4.0.5 或更高版本的 TiDB 集群。

以下为 TiFlash-Summary 默认的监控信息：

11.8.6.1 Server

- Store size：每个 TiFlash 实例的使用的存储空间的大小。
- Available size：每个 TiFlash 实例的可用的存储空间的大小。
- Capacity size：每个 TiFlash 实例的存储容量的大小。
- Uptime：自上次重启以来 TiFlash 正常运行的时间。
- Memory：每个 TiFlash 实例内存的使用情况。
- CPU Usage：每个 TiFlash 实例 CPU 的使用率。
- FSync OPS：每个 TiFlash 实例每秒进行 fsync 操作的次数。
- File Open OPS：每个 TiFlash 实例每秒进行 open 操作的次数。
- Opened File Count：当前每个 TiFlash 实例打开的文件句柄数。

> 注意：
>
> Store size、FSync OPS、File Open OPS、Opened File Count 目前仅包含了 TiFlash 存储层的统计指标，未包括 TiFlash-Proxy 内的信息。

11.8.6.2 Coprocessor

- Request QPS：所有 TiFlash 实例收到的 coprocessor 请求数量。其中 batch 是 batch 请求数量，batch_cop 是 batch 请求中的 coprocessor 请求数量，cop 是直接通过 coprocessor 接口发送的 coprocessor 请求数量，cop_dag 是所有 coprocessor 请求中 dag 请求数量，super_batch 是开启 super batch 特性的请求数量。
- Executor QPS：所有 TiFlash 实例收到的请求中，每种 dag 算子的数量，其中 table_scan 是扫表算子，selection 是过滤算子，aggregation 是聚合算子，top_n 是 TopN 算子，limit 是 limit 算子。
- Request Duration：所有 TiFlash 实例处理 coprocessor request 总时间，总时间为接收到该 coprocessor 请求至请求应答完毕的时间。
- Error QPS：所有 TiFlash 实例处理 coprocessor 请求的错误数量。其中 meet_lock 为读取的数据有锁，region_not_found 为 Region 不存在，epoch_not_match 为读取的 Region epoch 与本地不一致，kv_client_error 为与 TiKV 通信产生的错误，internal_error 为 TiFlash 内部系统错误，other 为其他错误。
- Request Handle Duration：所有 TiFlash 实例处理 coprocessor 请求处理时间，处理时间为该 coprocessor 请求开始执行到执行结束的时间。
- Response Bytes/Seconds：所有 TiFlash 实例应答总字节数。
- Cop task memory usage：所有 TiFlash 实例处理 coprocessor 请求占用的总内存。
- Handling Request Number：所有 TiFlash 实例正在处理的 coprocessor 请求数量之和。请求的分类与 Request QPS 中的分类相同。
- Threads of RPC：每个 TiFlash 实例使用的实时 RPC 线程数。
- Max Threads of RPC：最近一段时间每个 TiFlash 实例使用的 RPC 线程数峰值。
- Threads：每个 TiFlash 实例使用的实时线程数。
- Max Threads：最近一段时间每个 TiFlash 实例使用的线程数峰值。

11.8.6.3 Task Scheduler

- Min TSO：每个 TiFlash 实例上正在运行的查询语句中的最小 TSO，该值确保具有最小 TSO 的查询可以被调度。如果当前没有正在运行的查询，则该值为 `uint64` 整数型最大值。
- Estimated Thread Usage and Limit：每个 TiFlash 实例上正在运行的所有任务占用的线程估值，以及该实例上任务调度器设置的估算线程用量的软限制和硬限制。
- Active and Waiting Queries Count：每个 TiFlash 实例上正在运行的查询数量和正在等待的查询数量。
- Active and Waiting Tasks Count：每个 TiFlash 实例上正在运行的任务数量和正在等待的任务数量。
- Hard Limit Exceeded Count：每个 TiFlash 实例上运行中任务的估算线程用量超过了设置的硬限制的次数。
- Task Waiting Duration：每个 TiFlash 实例上任务从初始化到被调度的等待时长。

11.8.6.4 DDL

- Schema Version：每个 TiFlash 实例目前缓存的 schema 版本。
- Schema Apply OPM：所有 TiFlash 实例每分钟 apply 同步 TiDB schema diff 的次数。diff apply 是正常的单次 apply 过程，如果 diff apply 失败，则 failed apply +1，并回退到 full apply，拉取最新的 schema 信息以更新 TiFlash 的 schema 版本。
- Schema Internal DDL OPM：所有 TiFlash 实例每分钟执行的内部 DDL 次数。
- Schema Apply Duration：所有 TiFlash 实例 apply schema 消耗的时间。

11.8.6.5 Storage

- Write Command OPS：所有 TiFlash 实例存储层每秒收到的写请求数量。
- Write Amplification：每个 TiFlash 实例写放大倍数（实际磁盘写入量/逻辑数据写入量）。total 为自此次启动以来的写放大倍数，5min 为最近 5 分钟内的写放大倍数。
- Read Tasks OPS：每个 TiFlash 实例每秒存储层内部读取任务的数量。
- Rough Set Filter Rate：每个 TiFlash 实例最近 1 分钟内读取的 packet 数被存储层粗糙索引过滤的比例。
- Internal Tasks OPS：所有 TiFlash 实例每秒进行内部数据整理任务的次数。
- Internal Tasks Duration：所有 TiFlash 实例进行内部数据整理任务消耗的时间。
- Page GC Tasks OPM：所有 TiFlash 实例每分钟进行 Delta 部分数据整理任务的次数。
- Page GC Tasks Duration：所有 TiFlash 实例进行 Delta 部分数据整理任务消耗的时间分布。
- Disk Write OPS：所有 TiFlash 实例每秒进行磁盘写入的次数。
- Disk Read OPS：所有 TiFlash 实例每秒进行磁盘读取的次数。
- Write flow：所有 TiFlash 实例磁盘写操作的流量。
- Read flow：所有 TiFlash 实例磁盘读操作的流量。

> 注意：
>
> 目前这部分监控指标仅包含了 TiFlash 存储层的统计指标，未包括 TiFlash-Proxy 内的信息。

11.8.6.6 Storage Write Stall

- Write & Delta Management Throughput：所有实例写入及数据整理的吞吐量。
  - `throughput_write` 表示通过 Raft 进行数据同步的吞吐量。
  - `throughput_delta-management` 表示数据整理的吞吐量。
  - `total_write` 表示自上次启动以来的总写入字节数。
  - `total_delta-management` 表示自上次启动以来数据整理的总字节数。
- Write Stall Duration：每个实例写入和移除 Region 数据产生的卡顿时长。
- Write Throughput By Instance：每个实例写入数据的吞吐量，包括 apply Raft 数据日志以及 Raft 快照的写入吞吐量。
- Write Command OPS By Instance：每个实例收到各种命令的总计数。
  - `write block` 表示通过 Raft 同步数据日志。
  - `delete_range` 表示从该实例中删除一些 Region 或移动一些 Region 到该实例中。
  - `ingest` 表示这些 Region 的快照被应用到这个实例中。

11.8.6.7 Raft

- Read Index OPS：每个 TiFlash 实例每秒触发 read_index 请求的次数，等于请求触发的 Region 总数。
- Read Index Duration：所有 TiFlash 实例在进行 read_index 消耗的时间，主要消耗在于和 Region leader 的交互和重试时间。
- Wait Index Duration：所有 TiFlash 实例在进行 wait_index 消耗的时间，即拿到 read_index 请求后，等待本地的 Region index >= read_index 所花费的时间。

11.8.7 TiCDC 重要监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署的监控系统面板包含 TiCDC 面板。本文档对 TiCDC 监控面板上的各项指标进行详细说明。在日常运维中，运维人员可通过观察 TiCDC 面板上的指标了解 TiCDC 当前的状态。

本文档的对指标的介绍基于以下同步任务，即使用默认配置同步数据到 MySQL。

```
cdc cli changefeed create --pd=http://10.0.10.25:2379 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

下图显示了 TiCDC Dashboard 各监控面板：



图 173: TiCDC Dashboard - Overview

各监控面板说明如下：

- Server：TiDB 集群中 TiKV 节点和 TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务的详细信息
- Events：TiCDC 内部数据流转的详细信息
- TiKV：TiKV 中和 TiCDC 相关的详细信息

11.8.7.1 Server 面板

Server 面板示例如下：



图 174: TiCDC Dashboard - Server metrics

Server 面板的各指标说明如下：

- Uptime：TiKV 节点和 TiCDC 节点已经运行的时间
- Goroutine count：TiCDC 节点 Goroutine 的个数
- Open FD count：TiCDC 节点打开的文件句柄个数
- Ownership：TiCDC 集群中节点的当前状态
- Ownership history：TiCDC 集群中 Owner 节点的历史记录
- CPU usage：TiCDC 节点使用的 CPU
- Memory usage：TiCDC 节点使用的内存

11.8.7.2 Changefeed 面板

Changefeed 面板示例如下：



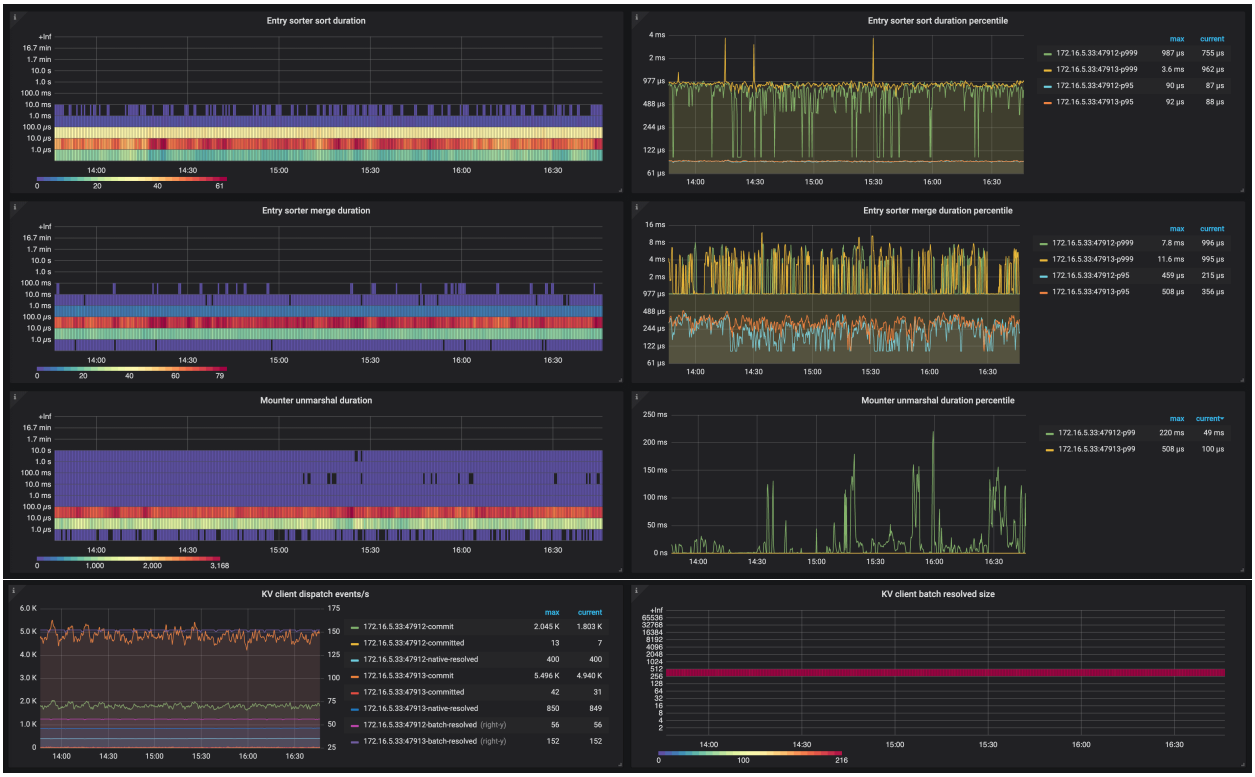
Changefeed 面板的各指标说明如下：

- Changefeed table count：一个同步任务中分配到各个 TiCDC 节点同步的数据表个数。
- Processor resolved ts：TiCDC 节点内部状态中已同步的时间点。
- Table resolved ts：同步任务中各数据表的同步进度。
- Changefeed checkpoint：同步任务同步到下游的进度，正常情况下绿柱应和黄线相接。
- PD etcd requests/s：TiCDC 节点每秒向 PD 读写数据的次数。
- Exit error count：每分钟内导致同步中断的错误发生次数。
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间计算）。
- Changefeed resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间计算）。
- Flush sink duration：TiCDC 异步刷写数据入下游的耗时直方图。
- Flush sink duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 异步刷写数据入下游所花费的时间。
- Sink write duration：TiCDC 将一个事务的更改写到下游的耗时直方图。
- Sink write duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 将一个事务的更改写到下游所花费的时间。
- MySQL sink conflict detect duration：MySQL 写入冲突检测耗时直方图。
- MySQL sink conflict detect duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，MySQL 写入冲突检测耗时。
- MySQL sink worker load：TiCDC 节点中写 MySQL 线程的负载情况。

11.8.7.3 Events 面板

Events 面板示例如下：





Events 面板的各指标说明如下：

- Eventfeed count：TiCDC 节点中 Eventfeed RPC 的个数。
- Event size percentile：每秒钟中 95% 和 99.9% 的情况下，TiCDC 收到的来自 TiKV 的数据变更消息大小。
- Eventfeed error/m：TiCDC 节点中每分钟 Eventfeed RPC 遇到的错误个数。
- KV client receive events/s：TiCDC 节点中 KV client 模块每秒收到来自 TiKV 的数据变更个数。
- Puller receive events/s：TiCDC 节点中 Puller 模块每秒收到来自 KV client 模块的数据变更个数。
- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更个数。
- Sink flush rows/s：TiCDC 节点每秒写到下游的数据变更的个数。
- Puller buffer size：TiCDC 节点中缓存在 Puller 模块中的数据变更个数。
- Entry sorter buffer size：TiCDC 节点中缓存在 Sorter 模块中的数据变更个数。
- Processor/Mounter buffer size：TiCDC 节点中缓存在 Processor 模块和 Mounter 模块中的数据变更个数。
- Sink row buffer size：TiCDC 节点中缓存在 Sink 模块中的数据变更个数。
- Entry sorter sort duration：TiCDC 节点对数据变更进行排序的耗时直方图。
- Entry sorter sort duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 排序数据变更所花费的时间。
- Entry sorter merge duration：TiCDC 节点合并排序后的数据变更的耗时直方图。
- Entry sorter merge duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 合并排序后的数据变更所花费的时间。
- Mounter unmarshal duration：TiCDC 节点解码数据变更的耗时直方图。
- Mounter unmarshal duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 解码数据变更所花费的时间。
- KV client dispatch events/s：TiCDC 节点内部 KV client 模块每秒分发数据变更的个数。
- KV client batch resolved size：TiKV 批量发给 TiCDC 的 resolved ts 消息的大小。

11.8.7.4 TiKV 面板

TiKV 面板示例如下：



TiKV 面板的各指标说明如下：

- CDC endpoint CPU：TiKV 节点上 CDC endpoint 线程使用的 CPU
- CDC worker CPU：TiKV 节点上 CDC worker 线程使用的 CPU
- Min resolved ts：TiKV 节点上最小的 resolved ts
- Min resolved region：TiKV 节点上最小的 resolved ts 的 Region ID
- Resolved ts lag duration percentile：TiKV 节点上最小的 resolved ts 与当前时间的差距
- Initial scan duration：TiKV 节点与 TiCDC 建立链接时增量扫的耗时直方图
- Initial scan duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiKV 节点增量扫的耗时
- Memory without block cache：TiKV 节点在减去 RocksDB block cache 后使用的内存
- CDC pending bytes in memory：TiKV 节点中 CDC 模块使用的内存
- Captured region count：TiKV 节点上捕获数据变更的 Region 个数

11.9 TiDB Dashboard

11.9.1 TiDB Dashboard 介绍

TiDB Dashboard 是 TiDB 自 4.0 版本起提供的图形化界面，可用于监控及诊断 TiDB 集群。TiDB Dashboard 内置于 TiDB 的 PD 组件中，无需独立部署。

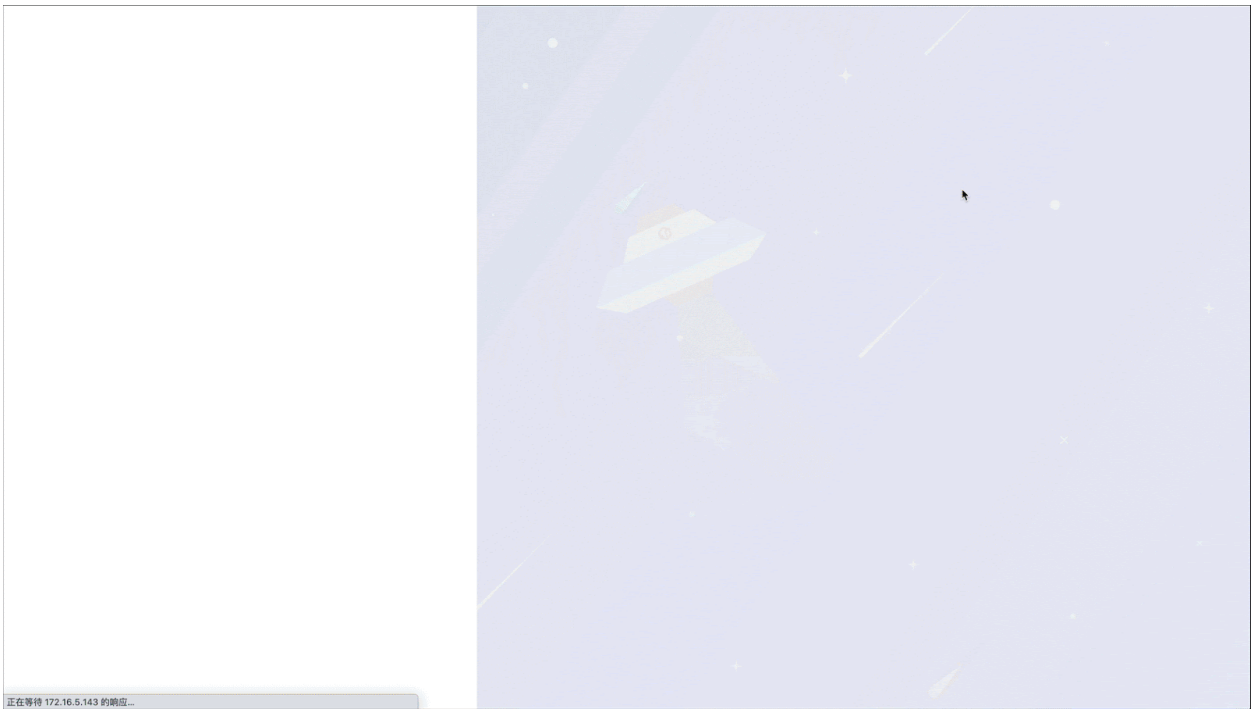


图 175: 界面

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。

11.9.1.1 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。

11.9.1.2 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。

11.9.1.3 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。

11.9.1.4 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。

11.9.1.5 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。

11.9.1.6 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。

11.9.1.7 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。

11.9.1.8 收集分析各个组件的性能数据

高级调试功能：无需第三方工具，在线地对各个组件进行性能分析，剖析组件实例在分析时间段内执行的各种内部操作及比例。

参阅实例性能分析页面了解详情。

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

11.9.2 运维

11.9.2.1 部署 TiDB Dashboard

TiDB Dashboard 界面内置于 TiDB 4.0 或更高版本的 PD 组件中，无需额外部署。只需部署标准 TiDB 集群，TiDB Dashboard 就会原生集成。

请参阅下列文档了解如何部署标准 TiDB 集群：

- 快速试用 TiDB 集群
- 生产环境部署
- Kubernetes 环境部署

> 注意：
>
> TiDB Dashboard 目前不能在低于 4.0 版本的集群中部署或使用。

11.9.2.1.1 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。

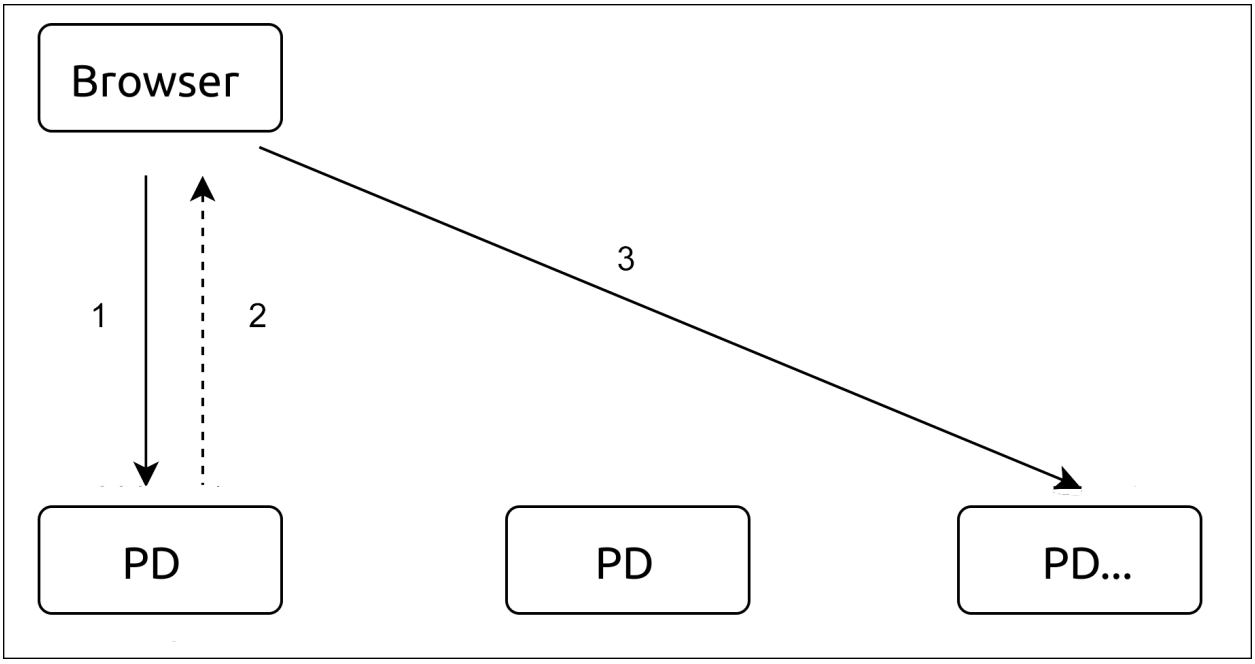


图 176: 流程示意

> **注意：**
>
> 提供 TiDB Dashboard 服务的 PD 实例不一定与 PD leader 一致。

查询实际运行 TiDB Dashboard 服务的 PD 实例

使用 TiUP 部署时，对于已启动的集群，可通过 `tiup cluster display` 命令查看哪个 PD 节点提供了 TiDB Dashboard 服务（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> **注意：**
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

切换其他 PD 实例提供 TiDB Dashboard 服务

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl 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 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。

11.9.2.1.2 禁用 TiDB Dashboard

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl 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 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.
```

11.9.2.1.3 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 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 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。

11.9.2.1.4 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。

11.9.2.2 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。

11.9.2.2.1 操作步骤

第 1 步：获取实际 TiDB Dashboard 地址

当集群中部署有多个 PD 实例时，其中仅有一个 PD 实例会真正运行 TiDB Dashboard，因此需要确保反向代理的上游 (Upstream) 指向了正确的地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署章节。

使用 TiUP 部署工具时，操作命令如下（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际 TiDB Dashboard 地址。样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> 注意：
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

第 2 步：配置反向代理

使用 HAProxy 反向代理

HAProxy 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理 TiDB Dashboard 为例，在 HAProxy 配置文件中，新增如下配置：

“‘haproxy frontend tidb_dashboard_front bind *:8033 use_backend tidb_dashboard_back if { path /dashboard } or { path_beg /dashboard/ }

backend tidb_dashboard_back mode http server tidb_dashboard 192.168.0.123:2379 “‘

其中 192.168.0.123:2379 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址中的 IP 及端口部分。

> 警告：
>
> 请务必保留 use_backend 指令中的 if 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

2. 重启 HAProxy，以使配置生效。

3. 测试反向代理是否生效：访问 HAProxy 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com:8033/dashboard/ ，即可访问 TiDB Dashboard。

使用 NGINX 反向代理

NGINX 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理 TiDB Dashboard 为例，在 NGINX 配置文件中，新增如下配置：

```
nginx  server {  listen 8033; location /dashboard/ { proxy_pass http://192.168.0.123:2379/dashboard
↪ /; } }
```

其中 http://192.168.0.123:2379/dashboard/ 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> 警告：
>
> 请务必保留 proxy_pass 指令中的 /dashboard/ 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

2. 重新载入 NGINX 以使配置生效：

```
shell  sudo nginx -s reload
```

3. 测试反向代理是否生效：访问 NGINX 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com:8033/dashboard/ ，即可访问 TiDB Dashboard。

11.9.2.2.2 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。

第 1 步：修改 PD 配置指定 TiDB Dashboard 服务路径前缀

修改 PD 配置中 [dashboard] 类别的 public-path-prefix 配置项，可指定服务路径前缀。该配置修改后需要重启 PD 实例生效。

以 TiUP 部署且希望运行在 http://example.com:8033/foo/ 为例，可指定以下配置：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /foo
```

使用 TiUP 部署全新集群时修改配置

若要全新部署集群，可在 TiUP 拓扑文件 topology.yaml 中加入上述配置项后进行部署，具体步骤参阅 TiUP 部署文档。

使用 TiUP 修改已部署集群的配置

1. 以编辑模式打开该集群的配置文件（将 CLUSTER_NAME 替换为集群名称）

```
shell  tiup cluster edit-config CLUSTER_NAME
```

2. 在 server_configs 的 pd 配置下修改或新增配置项，若没有 server_configs 请在最顶层新增：

```
yaml  server_configs: pd:     dashboard.public-path-prefix: /foo
```

修改完成后的配置文件类似于：

```
yaml  server_configs: pd:     dashboard.public-path-prefix: /foo global: user: tidb ...
```

或

```
yaml  monitored:  ...  server_configs: tidb: ...  tikv: ...  pd:     dashboard.public-path-
↪ prefix: /foo ...
```

3. 滚动重启所有 PD 实例生效配置（将 CLUSTER_NAME 替换为集群名称）

```
shell  tiup cluster reload CLUSTER_NAME -R pd
```

详情请参阅TiUP 常见运维操作 - 修改配置参数。

若希望运行在根路径（如 http://example.com:8033/）下，相应的配置为：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /
```

> 警告：
>
> 修改自定义路径前缀生效后，直接访问将不能正常使用 TiDB Dashboard，您只能通过和路径前缀匹配的反向代理访问。

第 2 步：修改反向代理配置

使用 HAProxy 反向代理

以 http://example.com:8033/foo/ 为例，HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back if { path /foo } or { path_beg /foo/ }

backend tidb_dashboard_back
  mode http
  http-request set-path %[path,regsub(^/foo/?,/dashboard/)]
  server tidb_dashboard 192.168.0.123:2379
```

其中 192.168.0.123:2379 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址中的 IP 及端口部分。

> **警告：**
>
> 请务必保留 use_backend 指令中的 if 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 http://example.com:8033/），HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back

backend tidb_dashboard_back
  mode http
  http-request set-path /dashboard%[path]
  server tidb_dashboard 192.168.0.123:2379
```

修改配置并重启 HAProxy 后即可生效。

使用 NGINX 反向代理

以 http://example.com:8033/foo/ 为例，相应的 NGINX 配置为：

```
server {
  listen 8033;
  location /foo/ {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

其中 http://192.168.0.123:2379/dashboard/ 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> **警告：**
>
> 请务必保留 proxy_pass 指令中的 /dashboard/ 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 http://example.com:8033/），NGINX 配置为：

```
server {
  listen 8033;
```

```
  location / {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

修改配置并重启 NGINX 后即可生效：

```
sudo nginx -s reload
```

11.9.2.2.3 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。

11.9.2.3 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。

11.9.2.3.1 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

> **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：

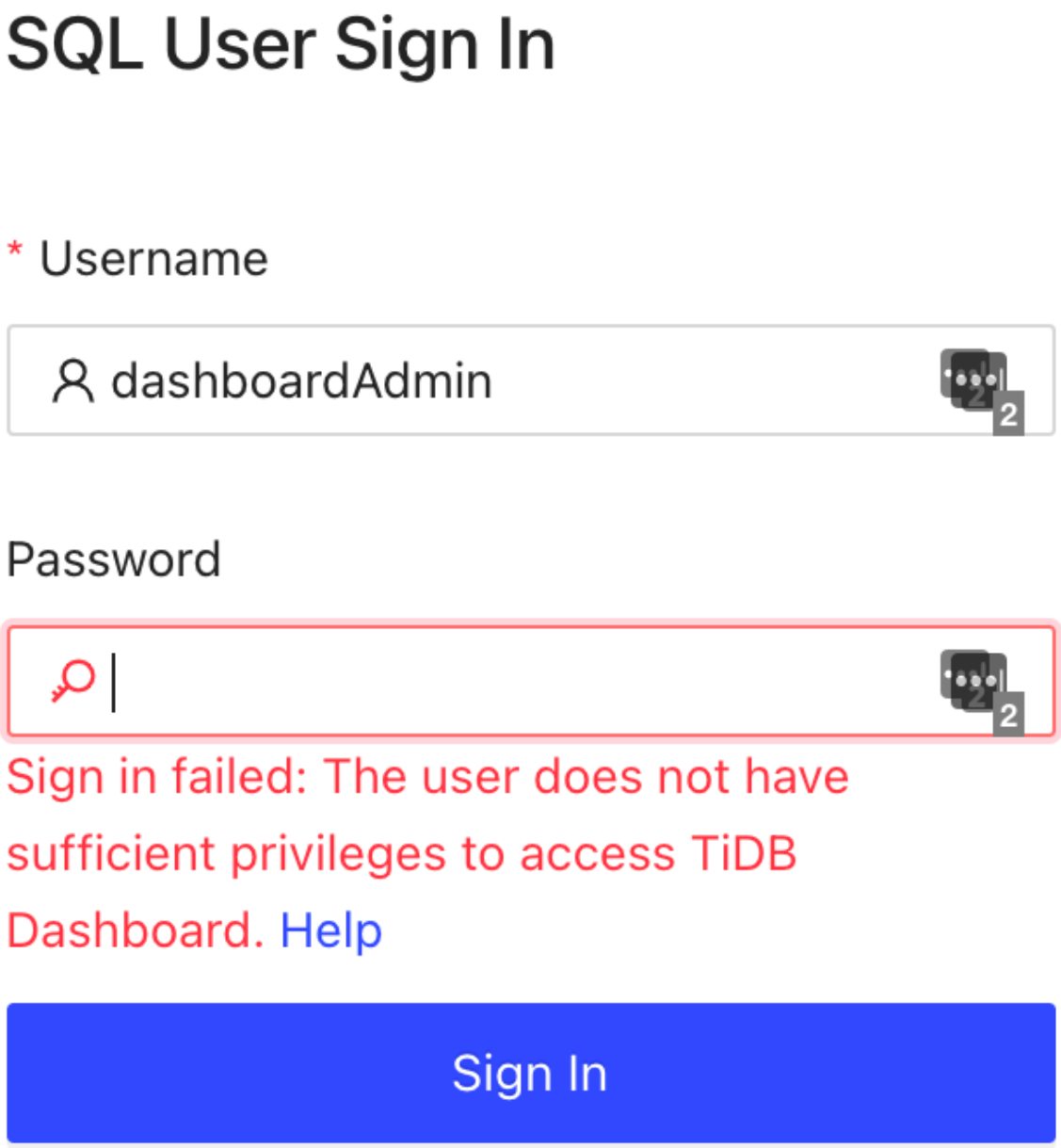


图 177: insufficient-privileges

11.9.2.3.2 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'@'%';
```

- 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 SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
```

11.9.2.3.3 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. 创建一个包含登录 TiDB Dashboard 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'@'%';
```

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。

11.9.2.3.4 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。

11.9.2.4 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。

11.9.2.4.1 提高 TiDB 用户安全性

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。

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。

11.9.2.4.2 使用防火墙阻止不可信访问

TiDB Dashboard 通过 PD Client 端口提供服务，默认为 http://IP:2379/dashboard/。尽管 TiDB Dashboard 需要验证身份，但 PD Client 端口上承载的其他 PD 内部特权接口不需要验证身份，且能进行特权操作，例如 http://IP:2379/pd/api/v1/members。因此，将 PD Client 端口直接暴露给外部网络具有极大的风险。

建议的措施：

1. 使用防火墙禁止组件外部网络或不可信网络访问任何 PD 组件的 Client 端口。

注意，TiDB、TiKV 等组件需要通过 PD Client 端口与 PD 组件进行通信，因此请勿对组件内部网络阻止访问，这将导致集群不可用。

2. 参见通过反向代理使用 TiDB Dashboard 了解如何配置反向代理将 TiDB Dashboard 服务在另一个端口上安全地提供给外部网络。

如何在多 PD 实例部署时开放 TiDB Dashboard 端口访问

> 警告：
>
> 本章节描述了一个不安全的访问方案，仅供测试环境使用。在生产环境中，不要使用本方案。请参见本文的其余章节。

在测试环境中，您可能需要配置防火墙开放 TiDB Dashboard 端口供外部访问。

当部署了多个 PD 实例时，其中仅有一个 PD 实例会真正运行 TiDB Dashboard，访问其他 PD 实例时会发生浏览器重定向，因此需要确保防火墙配置了正确的 IP 地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署章节。

使用 TiUP 部署工具时，可使用以下命令查看实际运行 TiDB Dashboard 的 PD 实例地址（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际 TiDB Dashboard 地址。

> 注意：
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

以下是一个样例输出：

```
http://192.168.0.123:2379/dashboard/
```

在这个样例中，需要为防火墙配置开放 IP 192.168.0.123 的 2379 端口入站访问，并通过 http://192.168.0.123:2379/dashboard/ 访问 TiDB Dashboard。

11.9.2.4.3 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。

11.9.2.4.4 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。

11.9.2.4.5 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

11.9.3 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。

11.9.3.1 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。

11.9.3.2 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。

11.9.3.3 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。

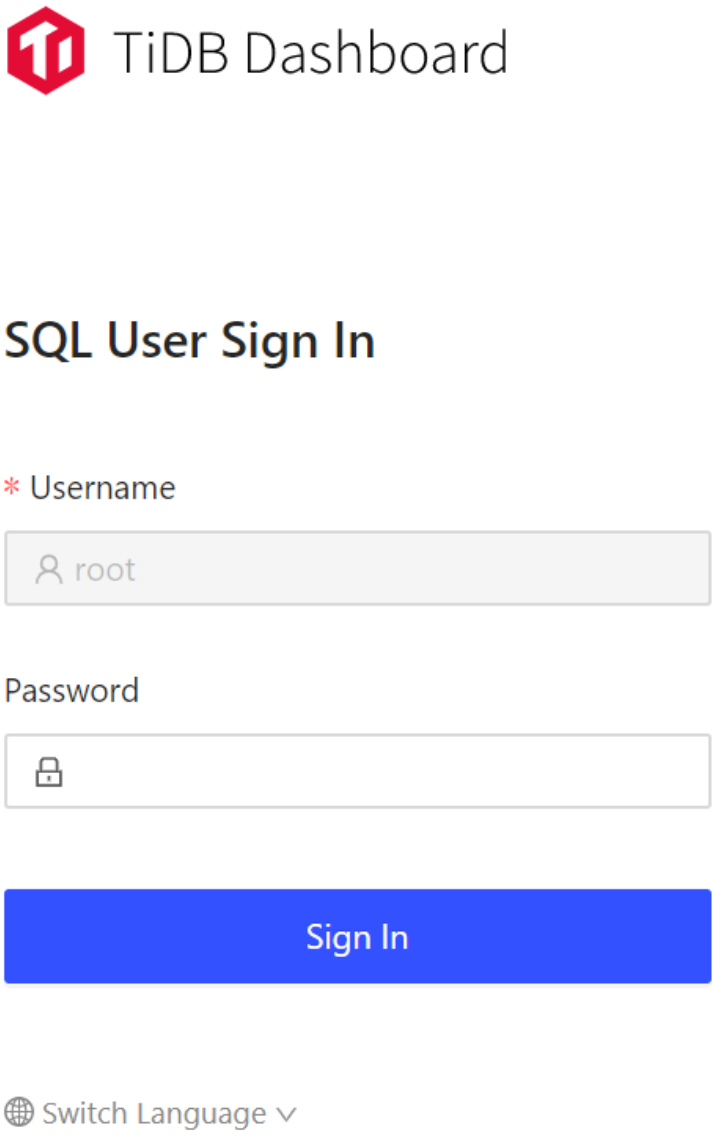


图 178: 登录界面

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。

11.9.3.4 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：

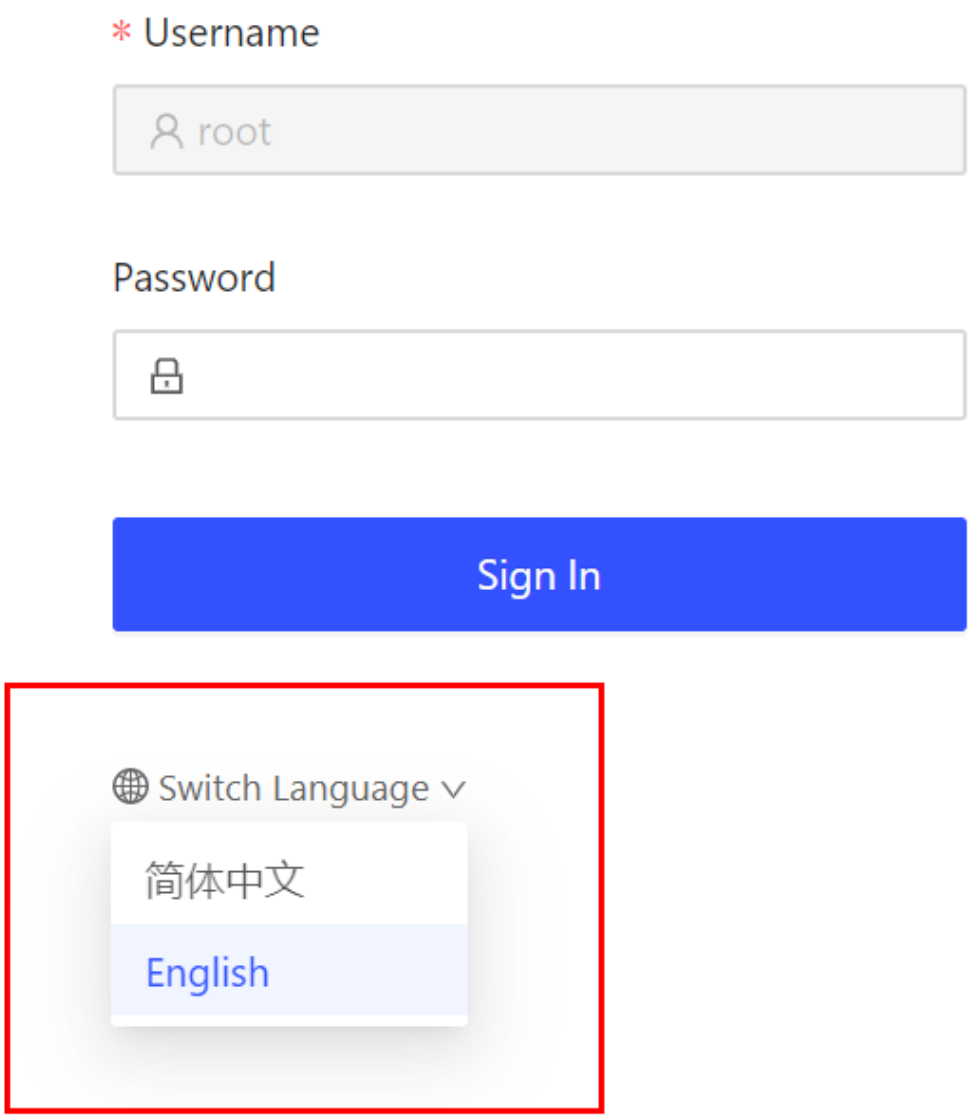


图 179: 切换语言

11.9.3.5 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。

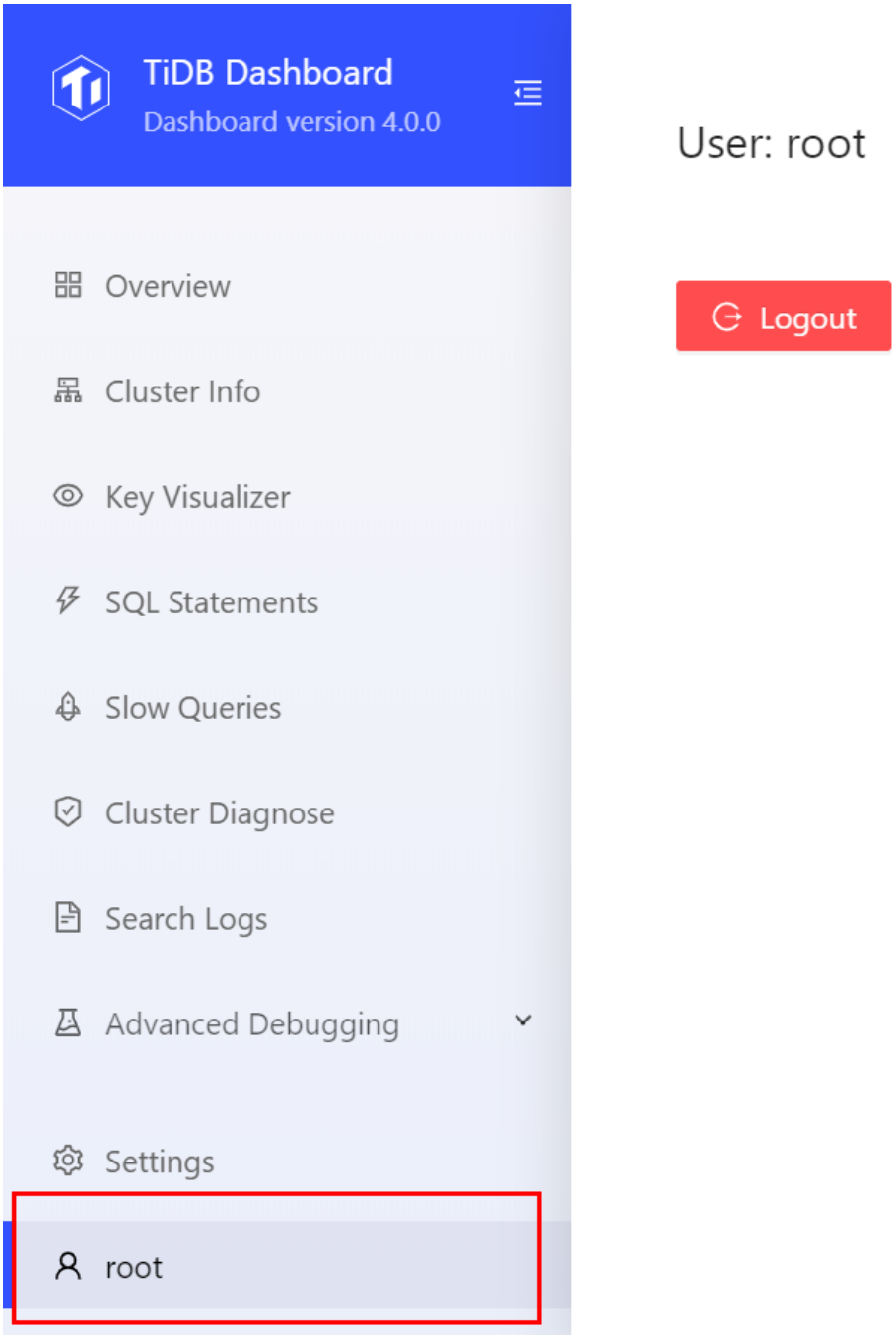


图 180: 登出

11.9.4 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

11.9.4.1 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 181: 访问

11.9.4.2 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：

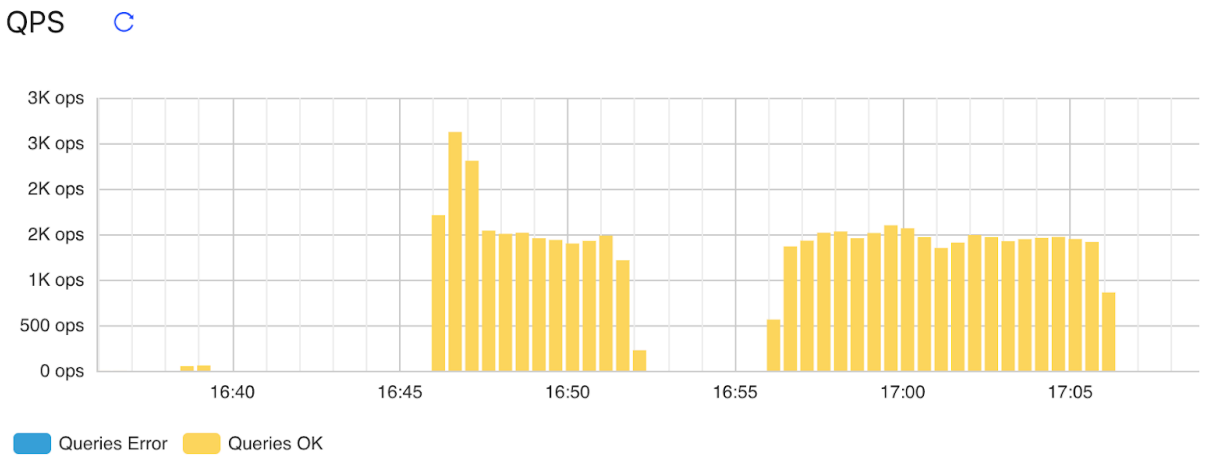


图 182: 界面

> **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。

11.9.4.3 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：

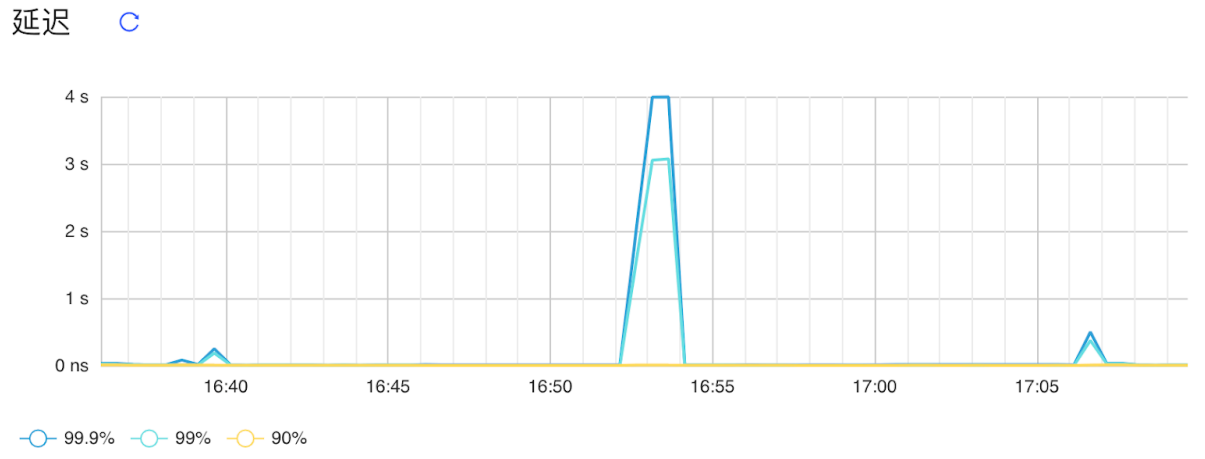


图 183: 界面

> **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。

11.9.4.4 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：

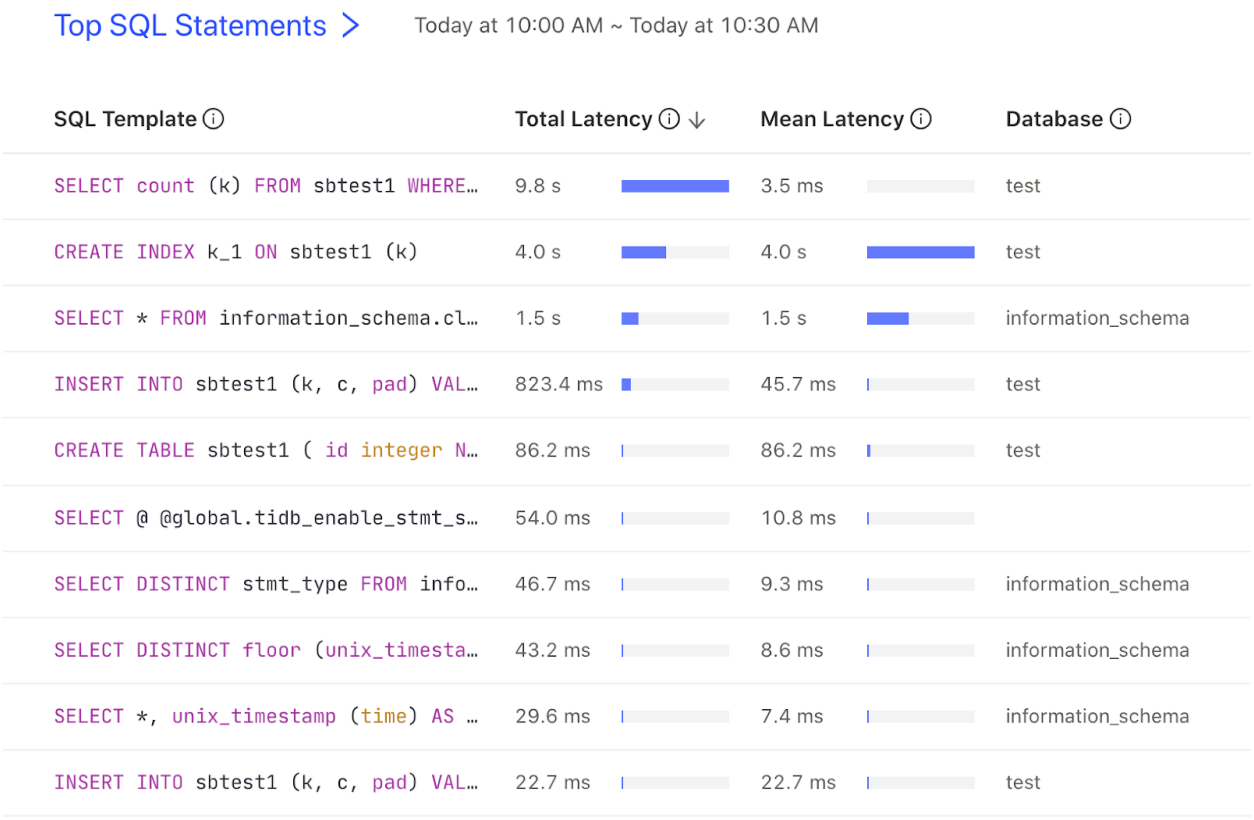


图 184: 界面

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。

11.9.4.5 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 185: 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。

11.9.4.6 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：

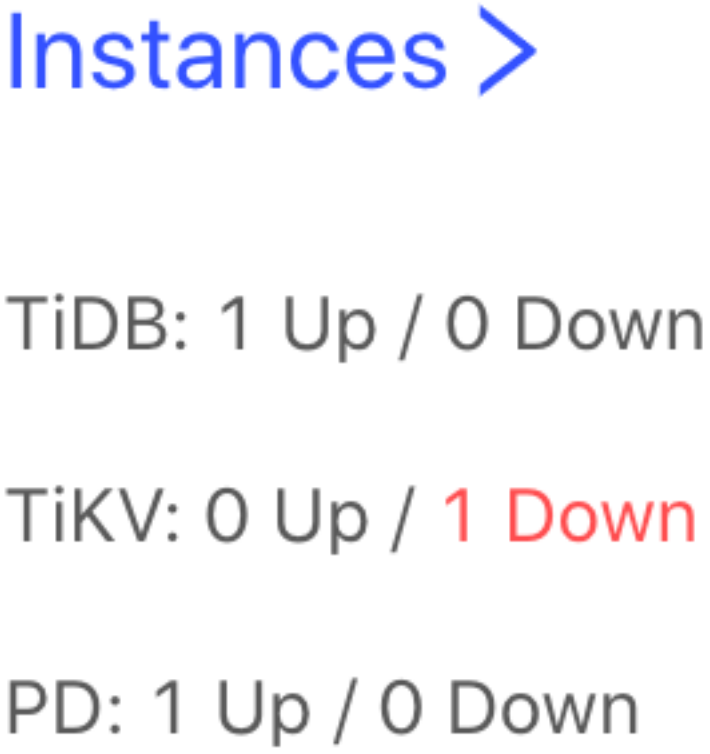


图 186: 界面

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。

11.9.4.7 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：

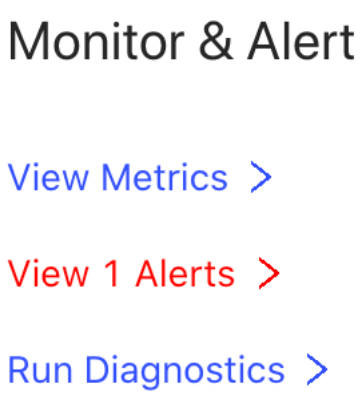


图 187: 界面

- 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。

11.9.5 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。

11.9.5.1 访问

可以通过以下两种方法访问集群信息页面：

- 登录后，左侧导航条点击集群信息：

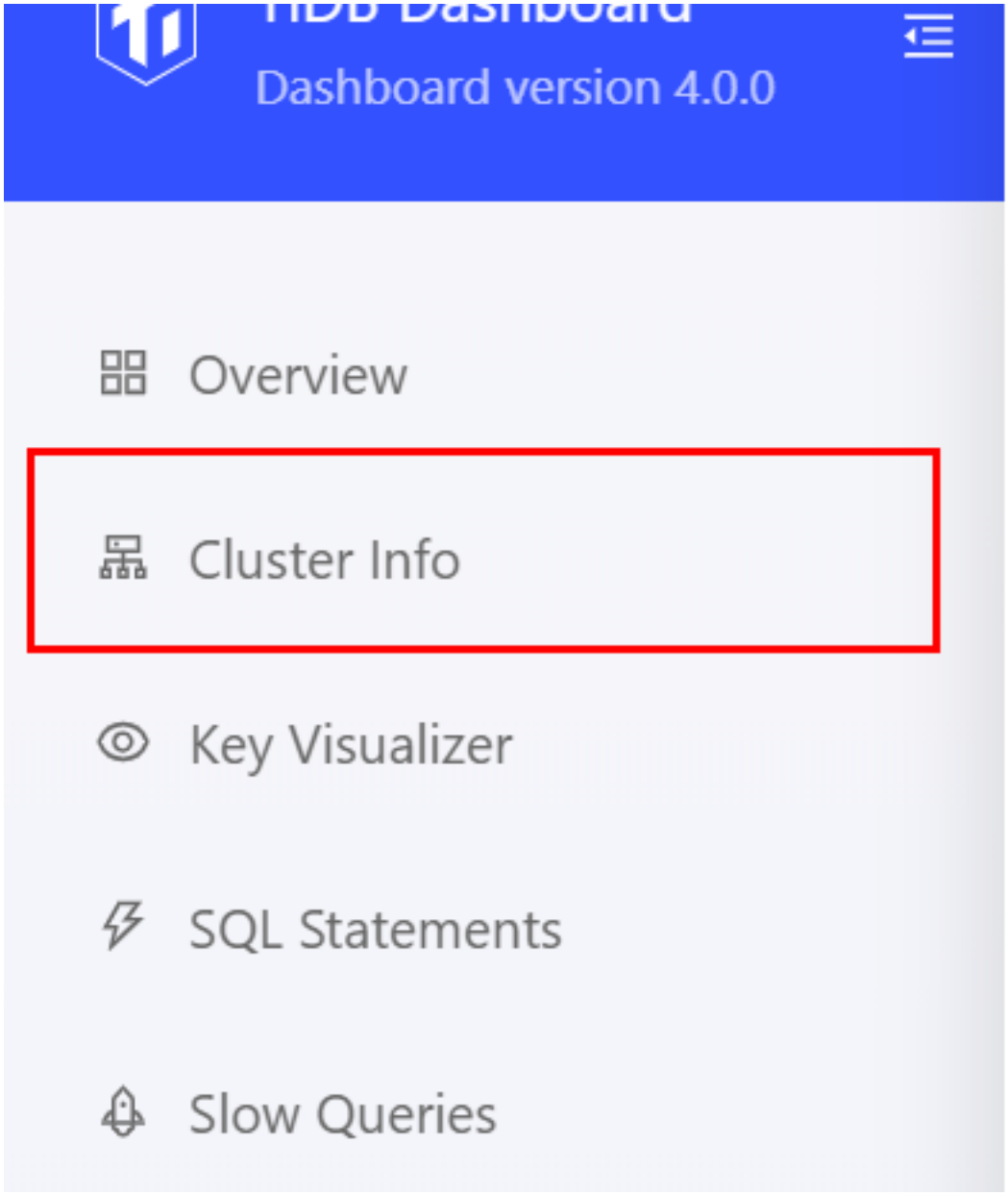


图 188: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/cluster_info/instance（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

11.9.5.2 实例列表

点击实例可查看实例列表：

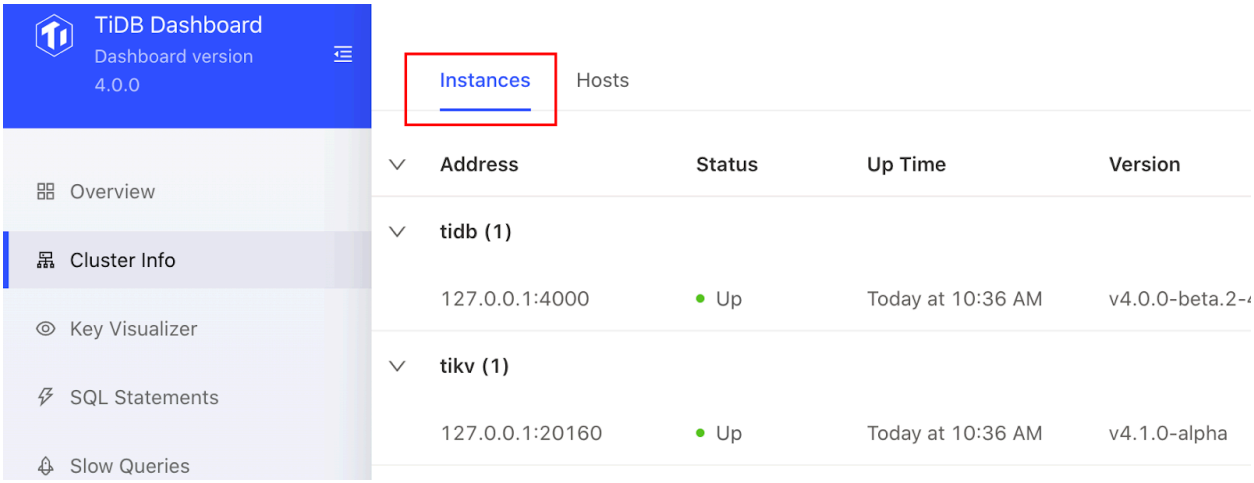


图 189: 实例

实例列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例的概况信息。

表格包含以下列：

- 地址：实例地址
- 状态：实例的运行状态
- 启动时间：实例的启动时间
- 版本：实例版本号
- 部署路径：实例二进制文件所在目录路径
- Git 哈希值：实例二进制对应的 Git 哈希值

实例的运行状态有：

- 在线 (Up)：实例正常运行。
- 离线 (Down) 或无法访问 (Unreachable)：实例未启动或对应主机存在网络问题。
- 已缩容下线 (Tombstone)：实例上的数据已被完整迁出并缩容完毕。仅 TiKV 或 TiFlash 实例存在该状态。
- 下线中 (Offline)：实例上的数据正在被迁出并缩容。仅 TiKV 或 TiFlash 实例存在该状态。
- 未知 (Unknown)：未知的实例运行状态。

> 注意：
>
> 表格中部分列仅在实例处于在线状态时能显示。

11.9.5.3 主机列表

点击主机可查看主机列表：

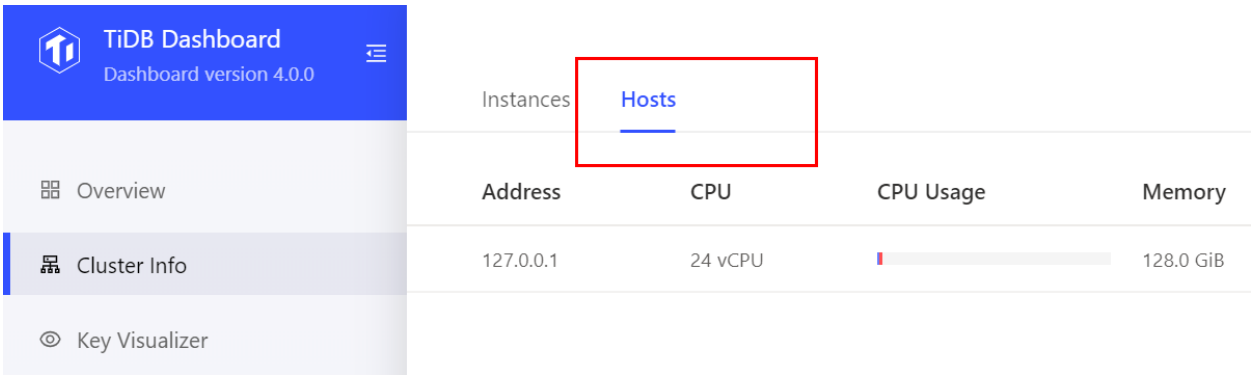


图 190: 主机

主机列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例对应主机的运行情况。

表格包含以下列：

- 地址：主机 IP 地址
- CPU：主机 CPU 逻辑核心数
- CPU 使用率：主机当前 1 秒的用户态和内核态 CPU 使用率
- 物理内存：主机总计的物理内存大小
- 内存使用率：主机当前内存使用率
- 部署磁盘：主机上运行实例所在磁盘的文件系统和磁盘挂载路径
- 磁盘使用率：主机上运行实例所在磁盘的空间使用率

> 注意：
>
> 主机列表信息由各个实例进程给出，因此当主机上所有实例都处于离线状态时，该主机信息将无法显示。

11.9.6 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。

11.9.6.1 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。

11.9.6.2 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

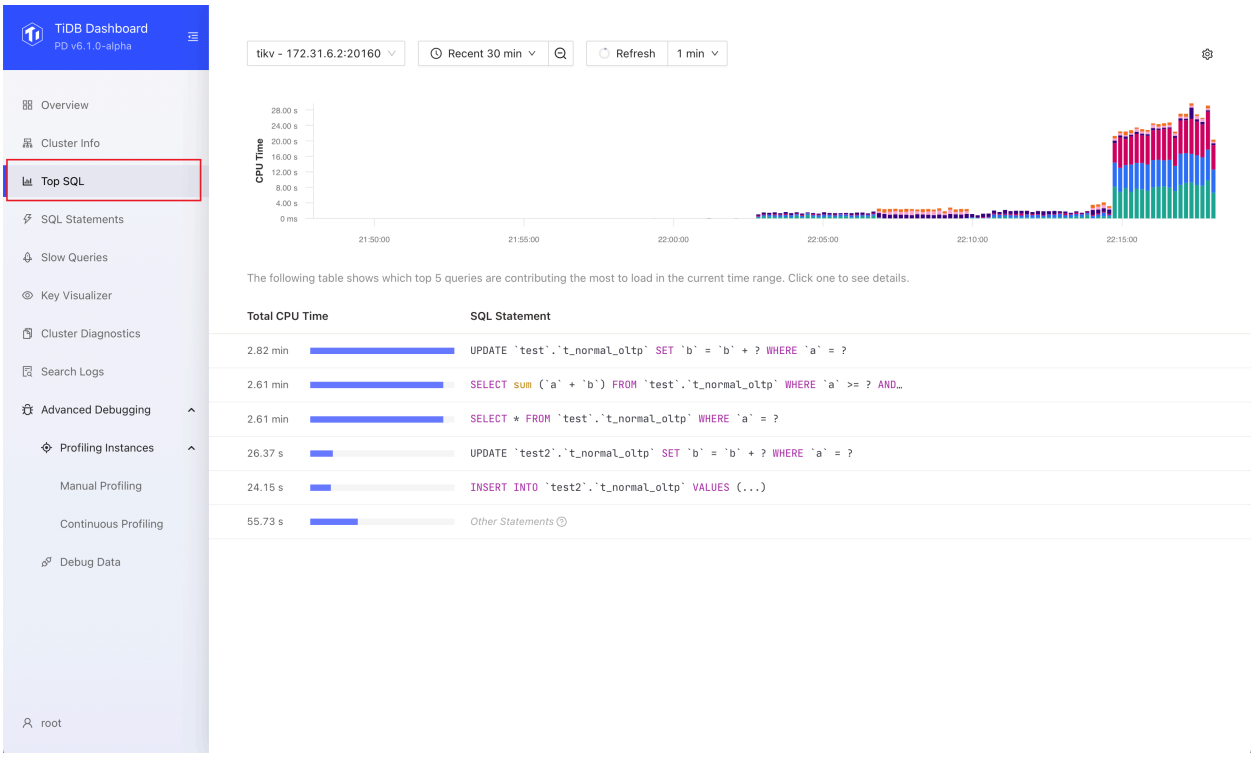


图 191: Top SQL

- 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）。

11.9.6.3 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;
```

11.9.6.4 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。

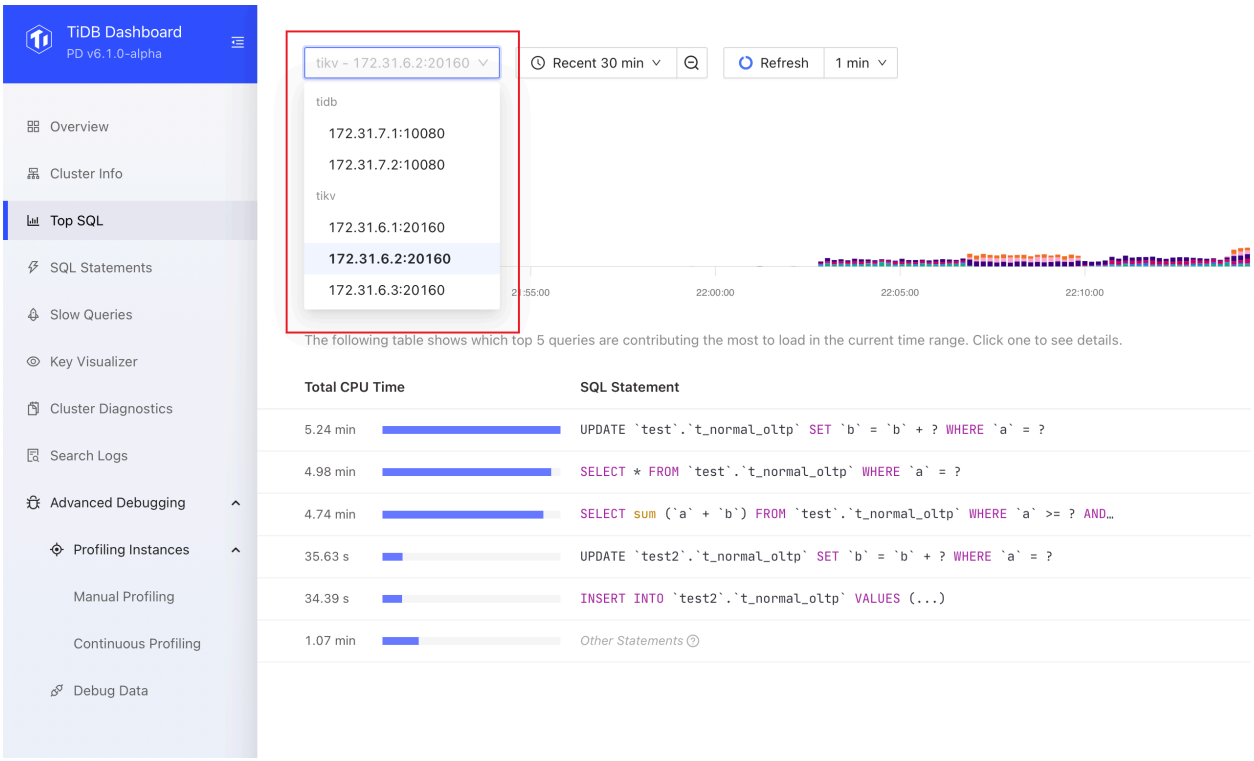


图 192: 选择实例

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。

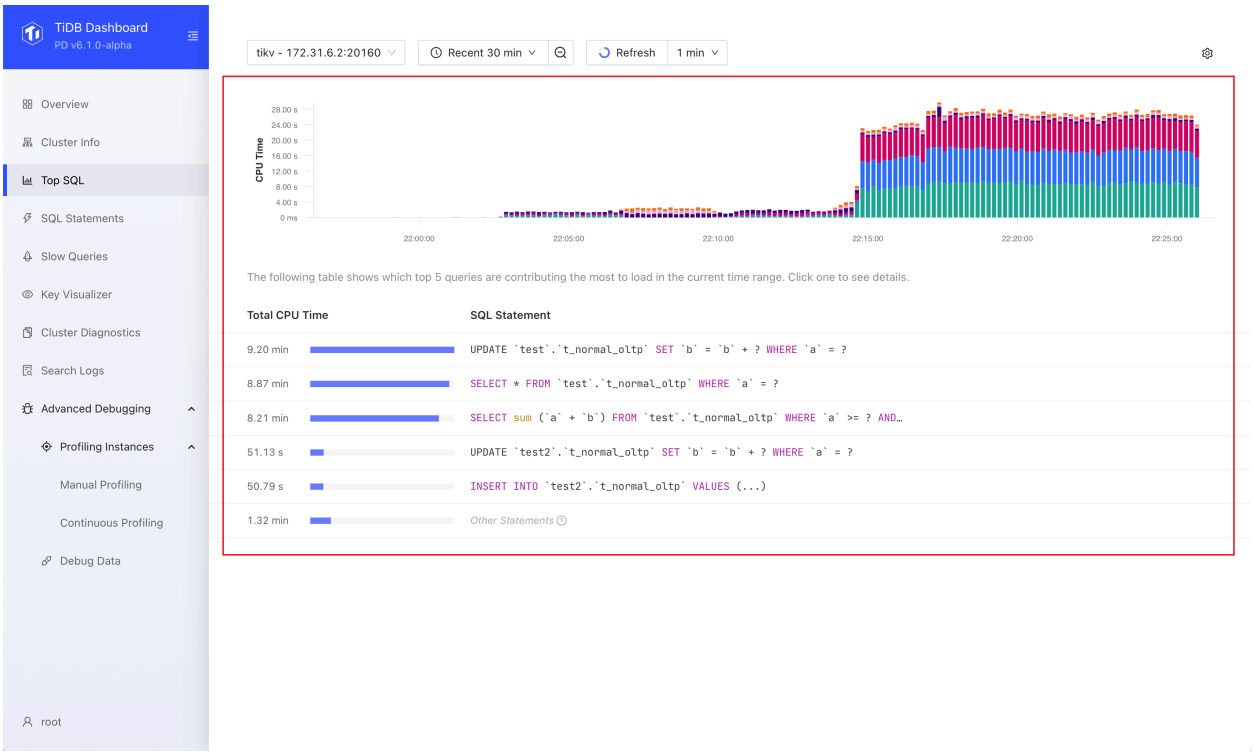


图 193: 图表表格

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。

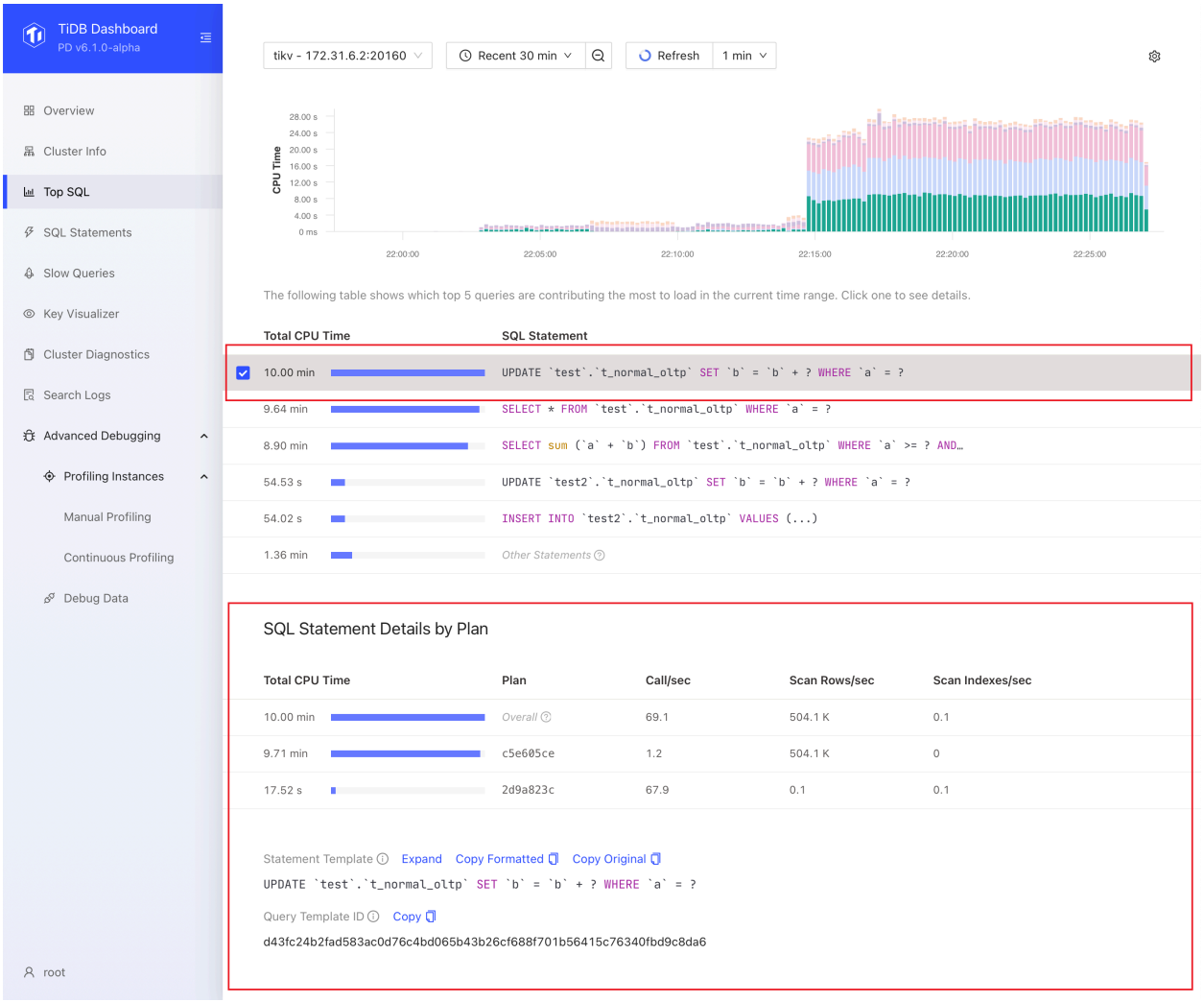


图 194: 详情

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 195: 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。

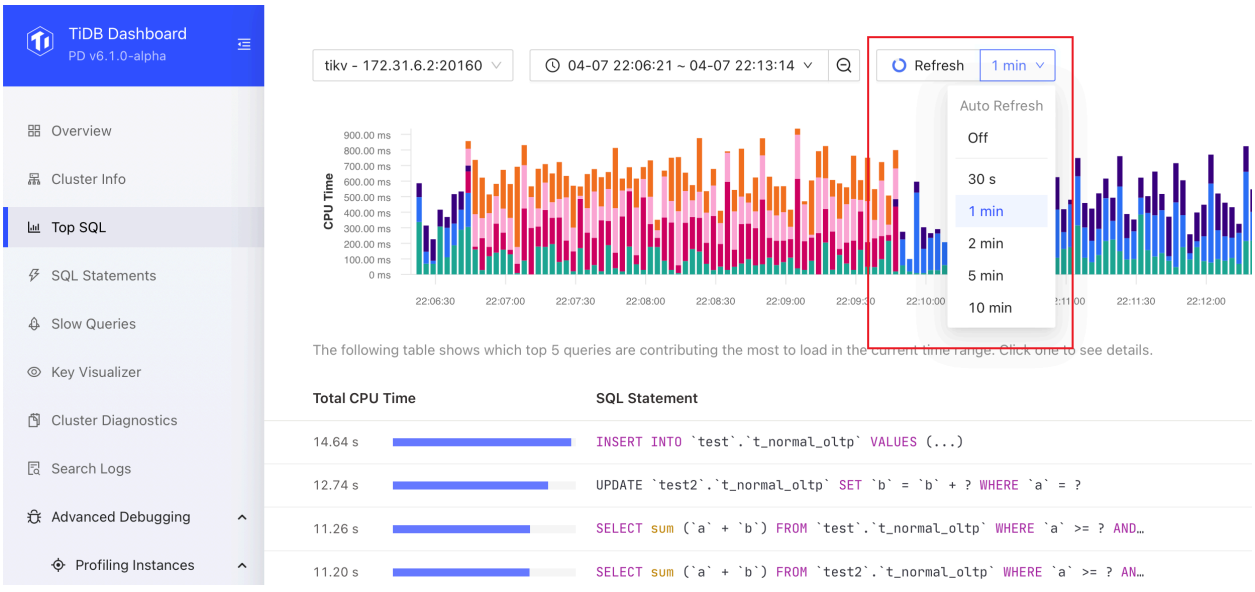


图 196: 刷新

11.9.6.5 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;
```

11.9.6.6 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。

11.9.7 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。

11.9.7.1 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 Key Visualizer（流量可视化）：

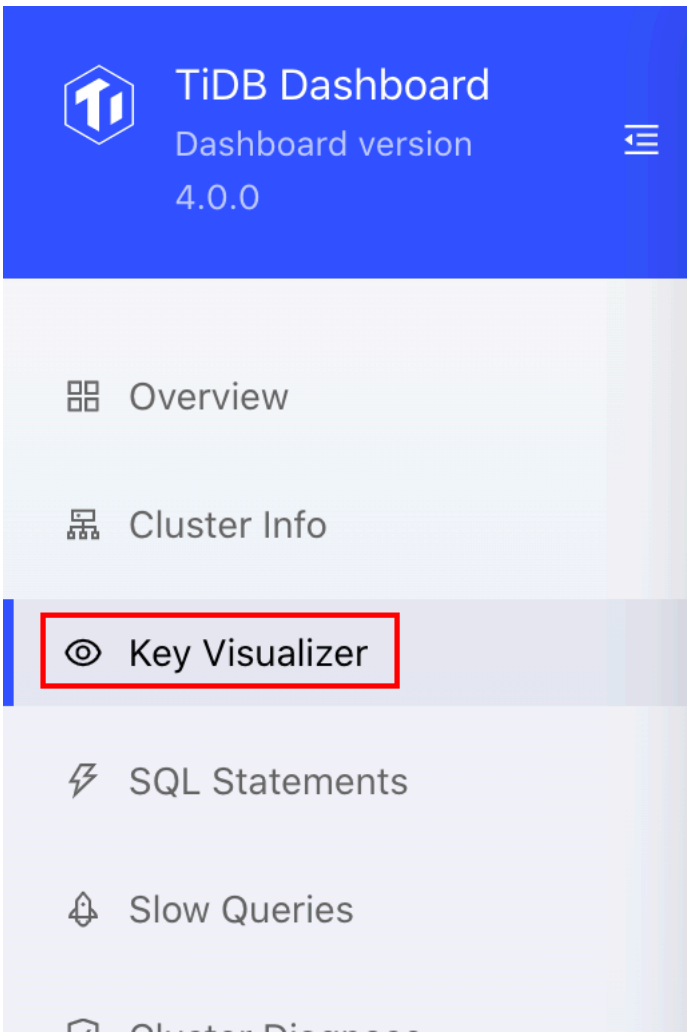


图 197: 访问

- 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）。

11.9.7.2 界面示例

流量可视化页面示例如下：

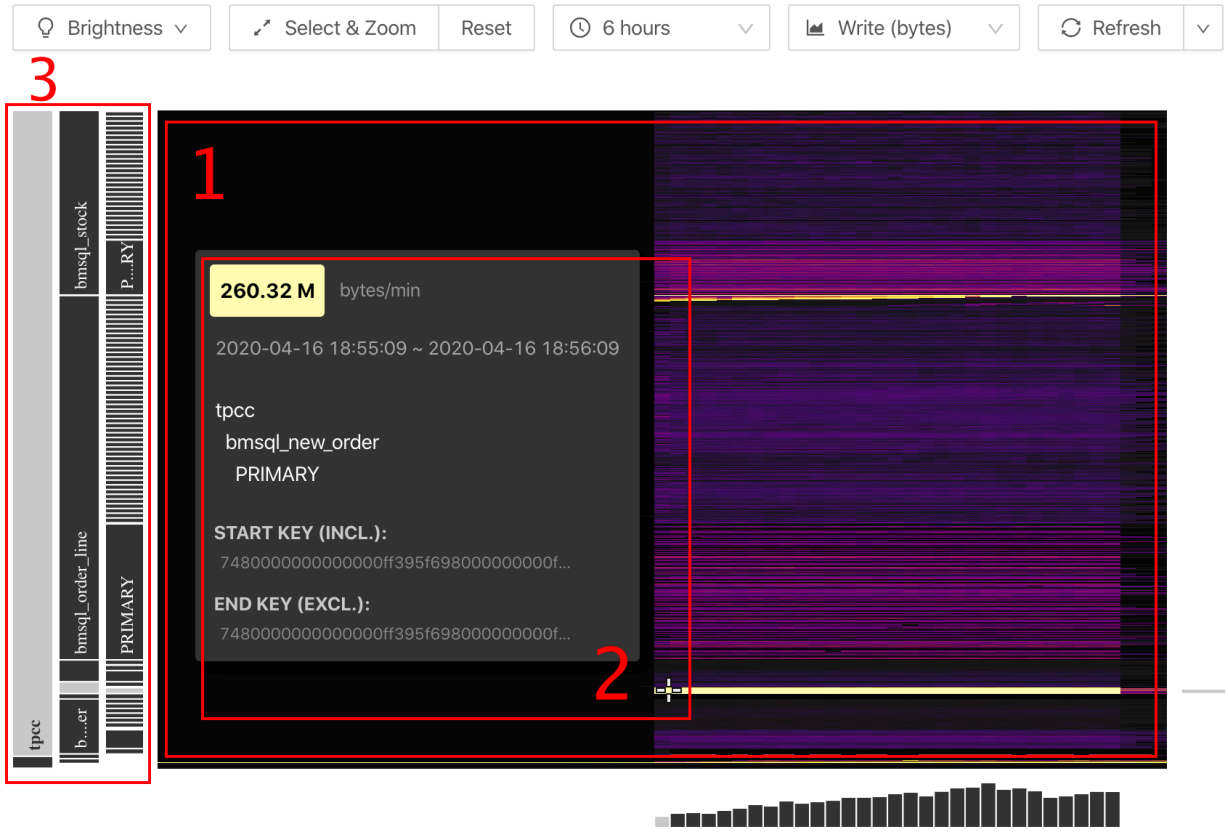


图 198: Key Visualizer 示例图

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。

11.9.7.3 基本概念

本节介绍流量可视化涉及的一些基本概念。

11.9.7.3.1 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

11.9.7.3.2 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。

11.9.7.3.3 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。

11.9.7.3.4 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。

11.9.7.4 使用介绍

本节介绍如何使用流量可视化页面。

11.9.7.4.1 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 Open Settings（打开设置）即可打开设置页面：

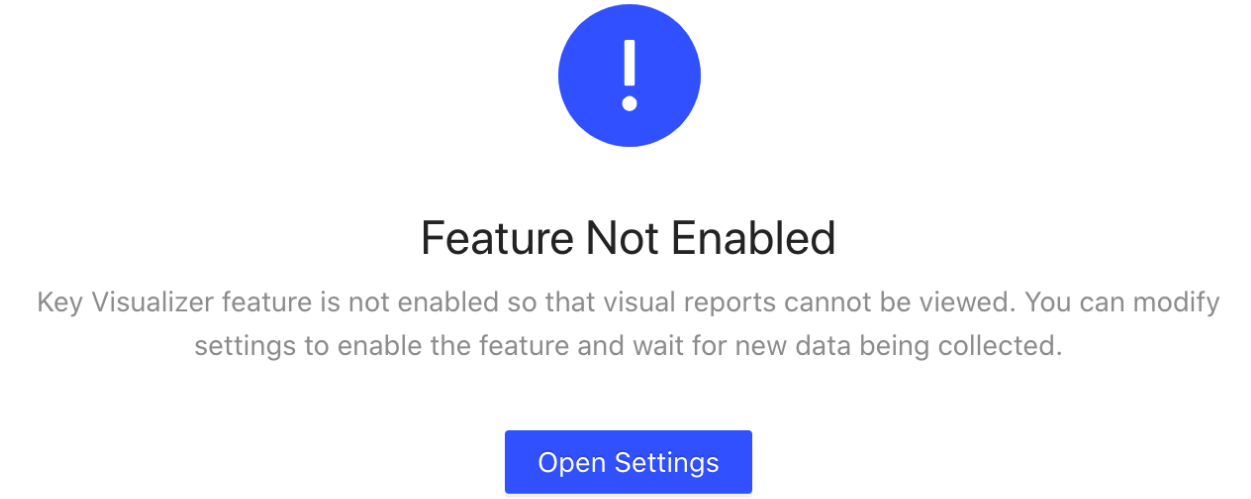


图 199: 功能未开启

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：

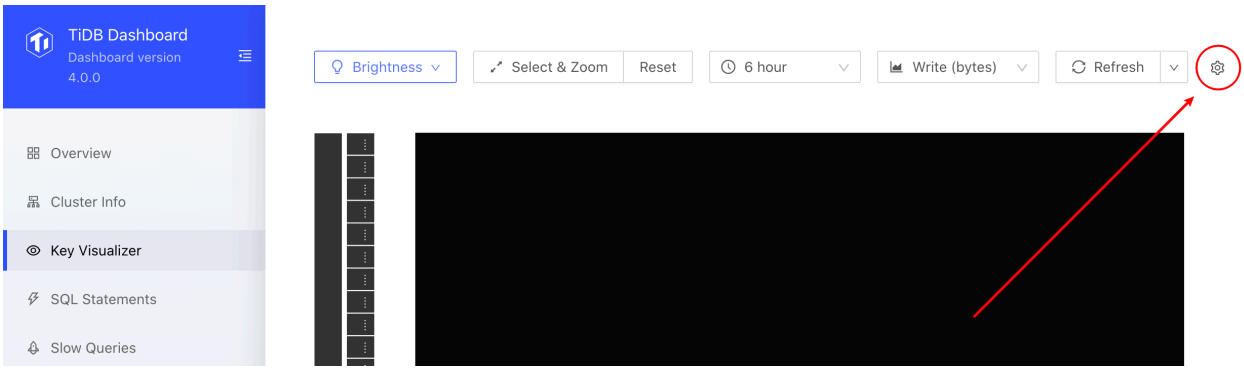


图 200: 设置按钮

设置页面如下图所示：

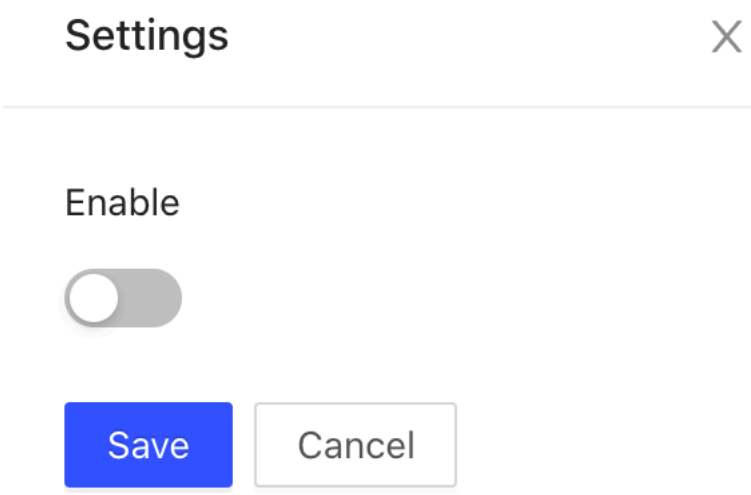


图 201: 设置页面

通过开关设定好是否开启收集，并点击 Save（保存）后生效。开启后，在界面上观察到工具栏已经可以使用：

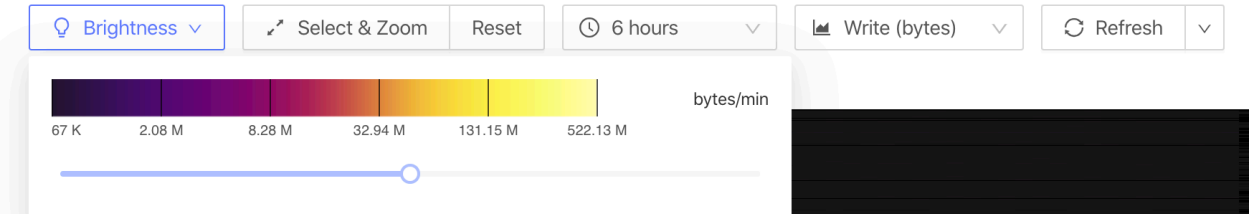


图 202: 工具栏

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。

11.9.7.4.2 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 Select & Zoom（框选）按钮，然后点击并拖动以选择要放大的区域。

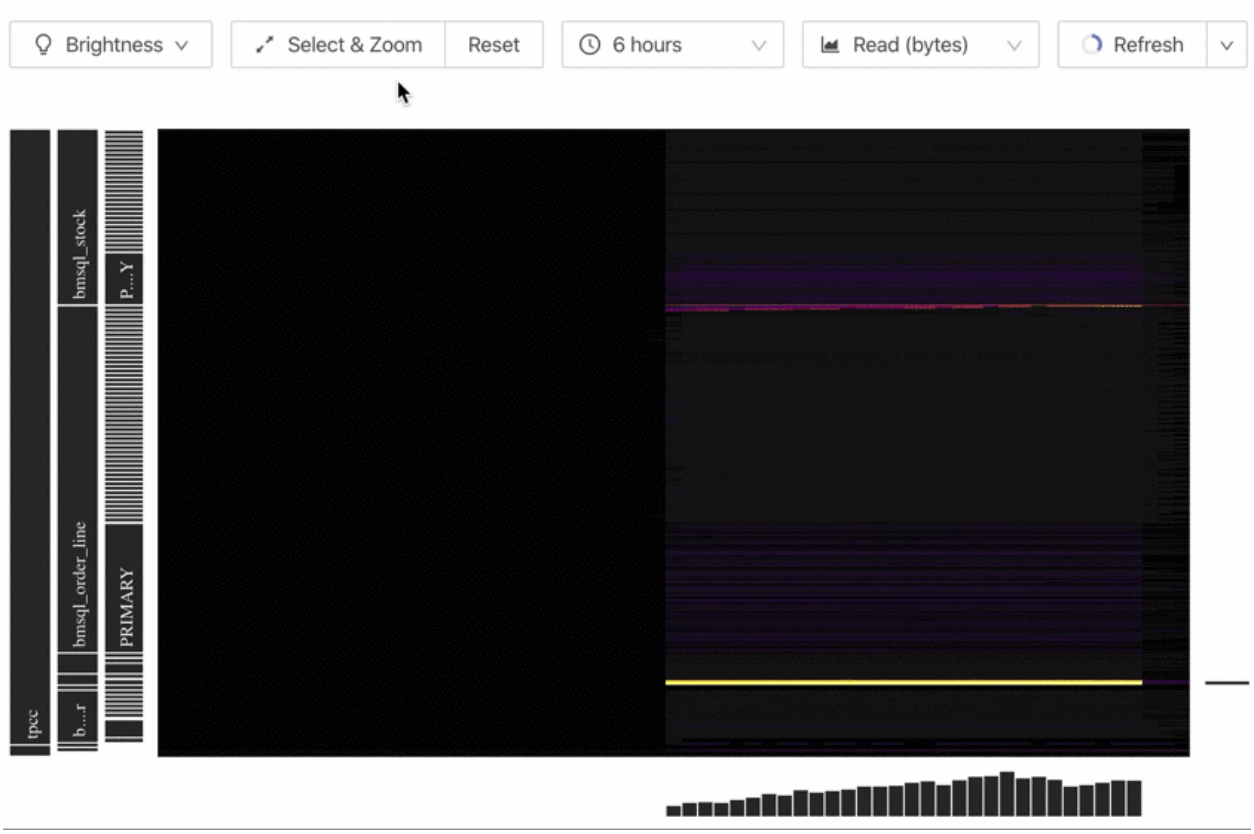


图 203: 框选

- 点击 Reset（重置）按钮，将 Region 范围重置为整个数据库。
- 点击「时间选择框」(界面的 6 hours 处)，重新选择观察时间段。

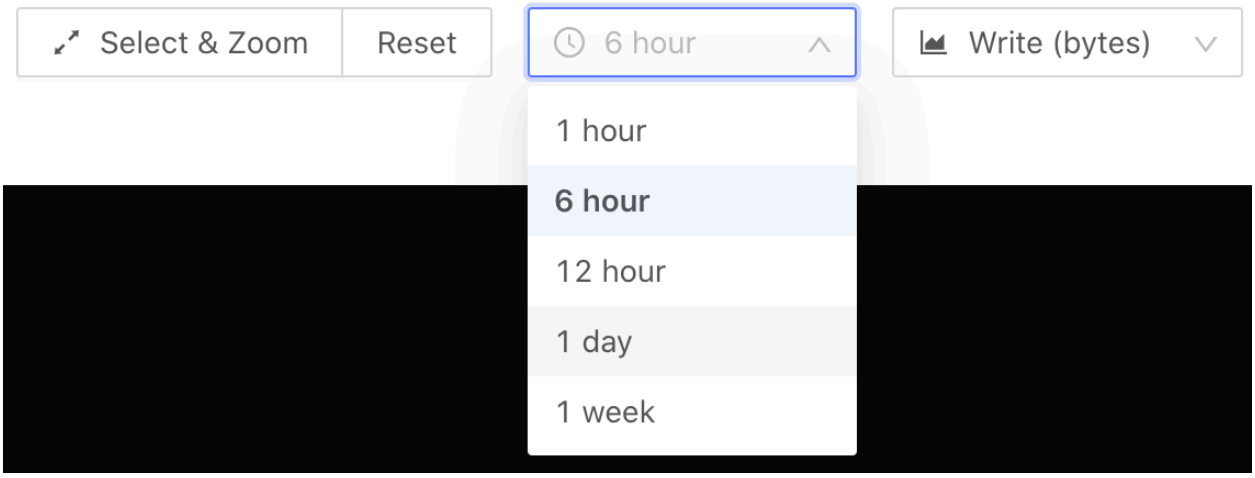


图 204: 时间选择

> 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。

11.9.7.4.3 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 Brightness（调整亮度）按钮，然后通过滑块来调节页面的亮度。

> 注意：
>
> 在显示一个区域内的热力图时，会根据区域内的流量情况来界定冷热。当整个区域流量较为平均时，即使整体流量在数值上很低，你依然有可能观察到较大的亮色区域。请注意一定要结合数值一起分析。

11.9.7.4.4 选择指标

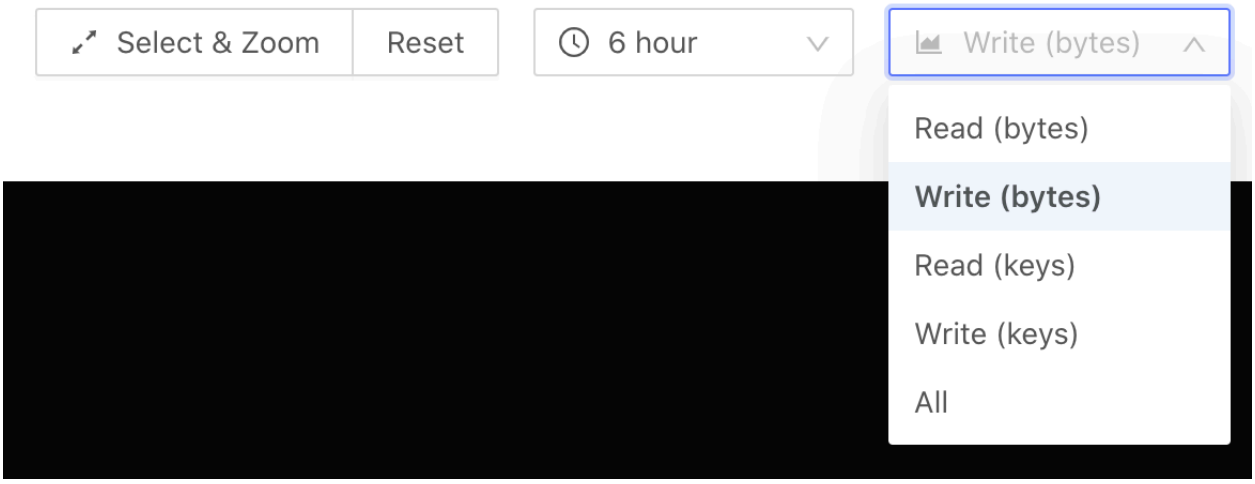


图 205: 指标选择

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）

11.9.7.4.5 刷新与自动刷新

可以通过点击 Refresh（刷新）按钮来重新获得基于当前时间的热力图。当需要实时观察数据库的流量分布情况时，可以点击按钮右侧的向下箭头，选择一个固定的时间间隔让热力图按此间隔自动刷新。

> 注意：
>
> 如果进行了时间范围或者 Region 范围的调整，自动刷新会被关闭。

11.9.7.4.6 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：

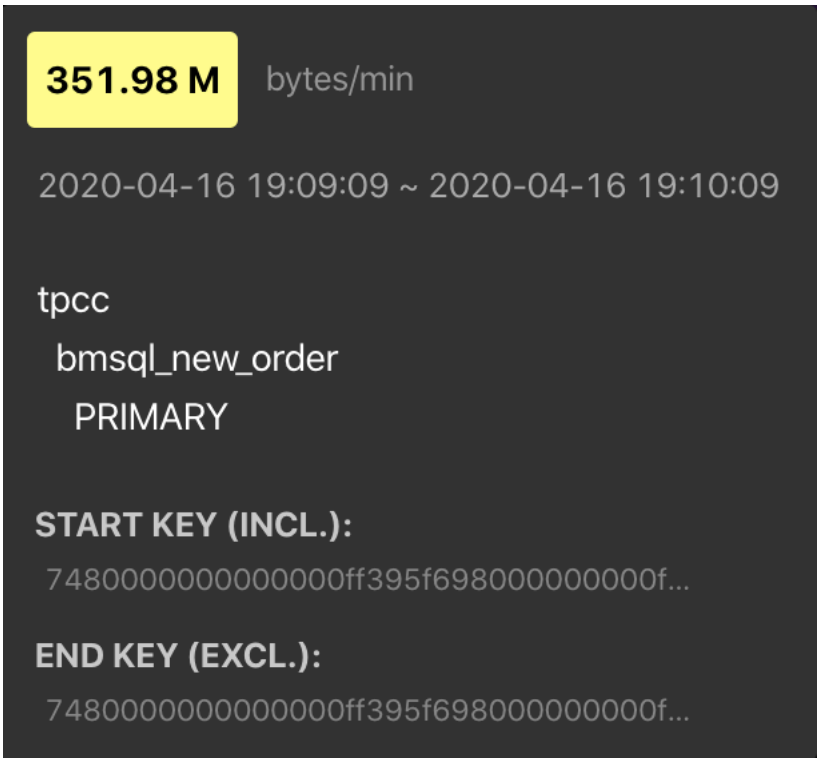


图 206: Bucket 详细信息

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：

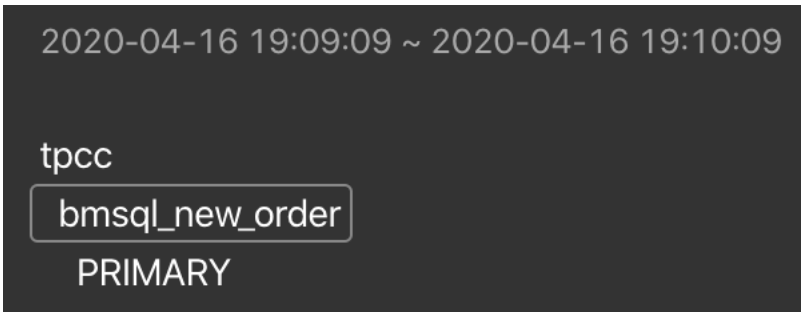


图 207: 复制 Bucket 详细信息

11.9.7.5 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。

11.9.7.5.1 均衡：期望结果

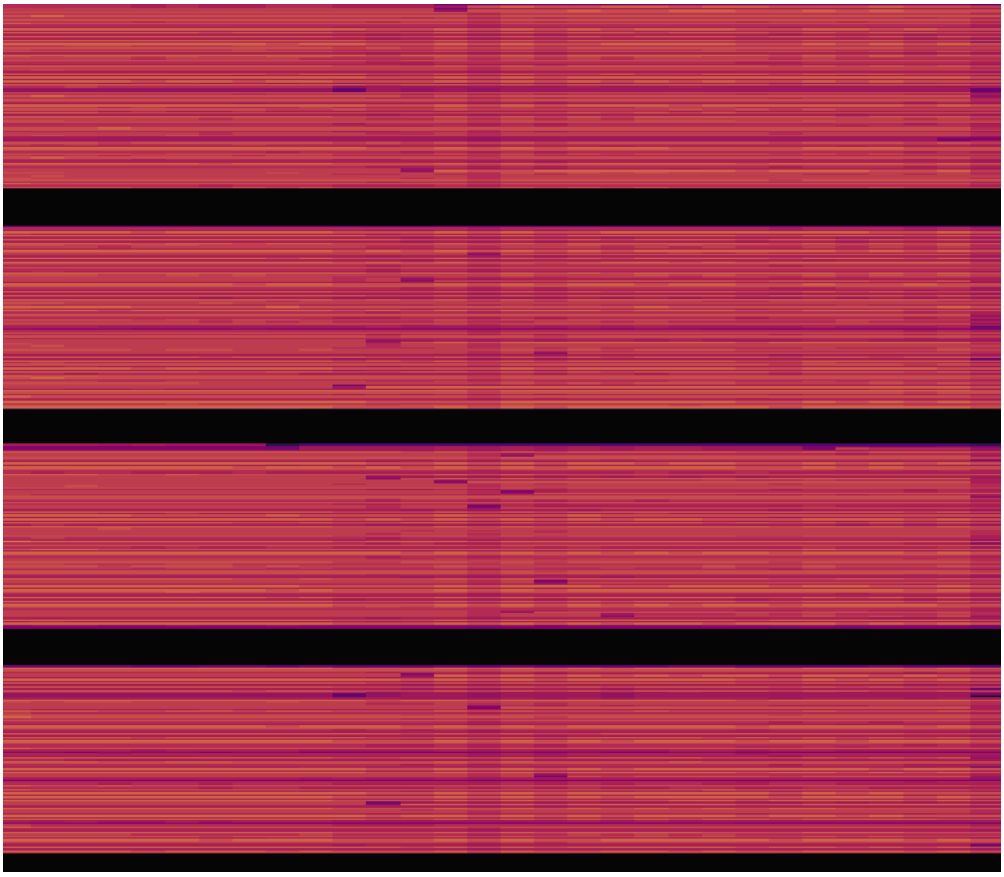


图 208: 均衡结果图

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。

11.9.7.5.2 X 轴明暗交替：需要关注高峰期的资源情况

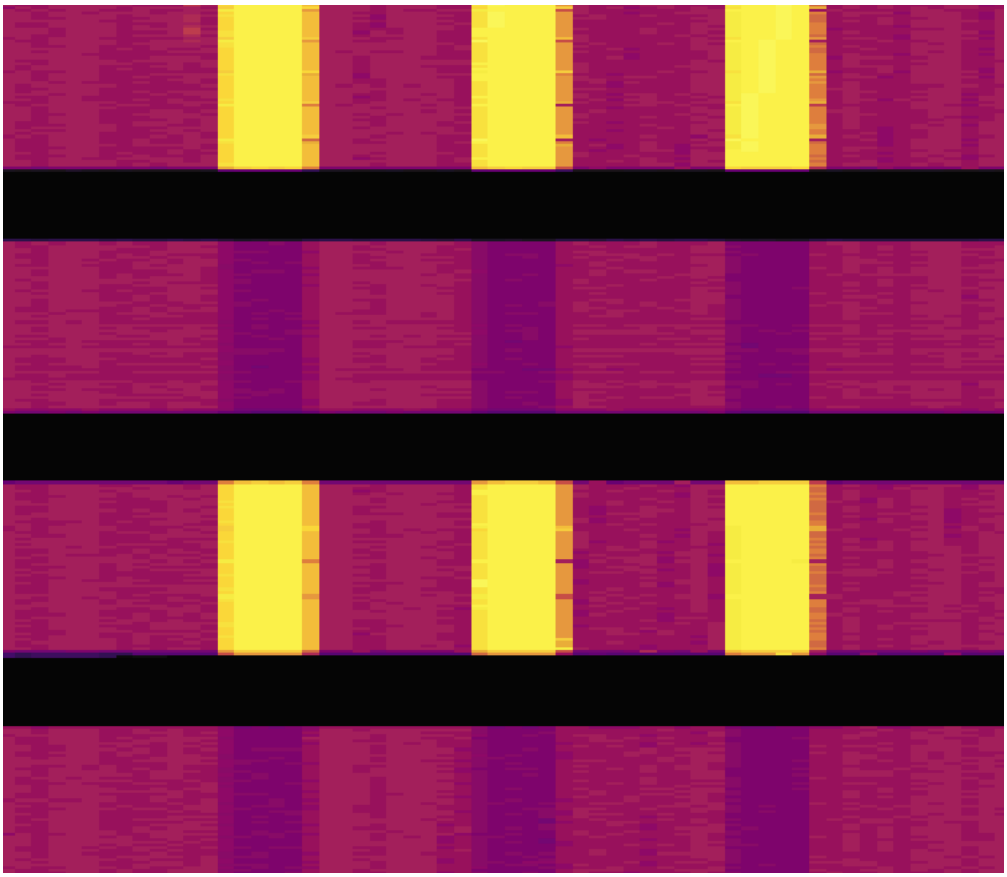


图 209: X 轴明暗交替

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。

11.9.7.5.3 Y 轴明暗交替：需要关注产生的热点聚集程度

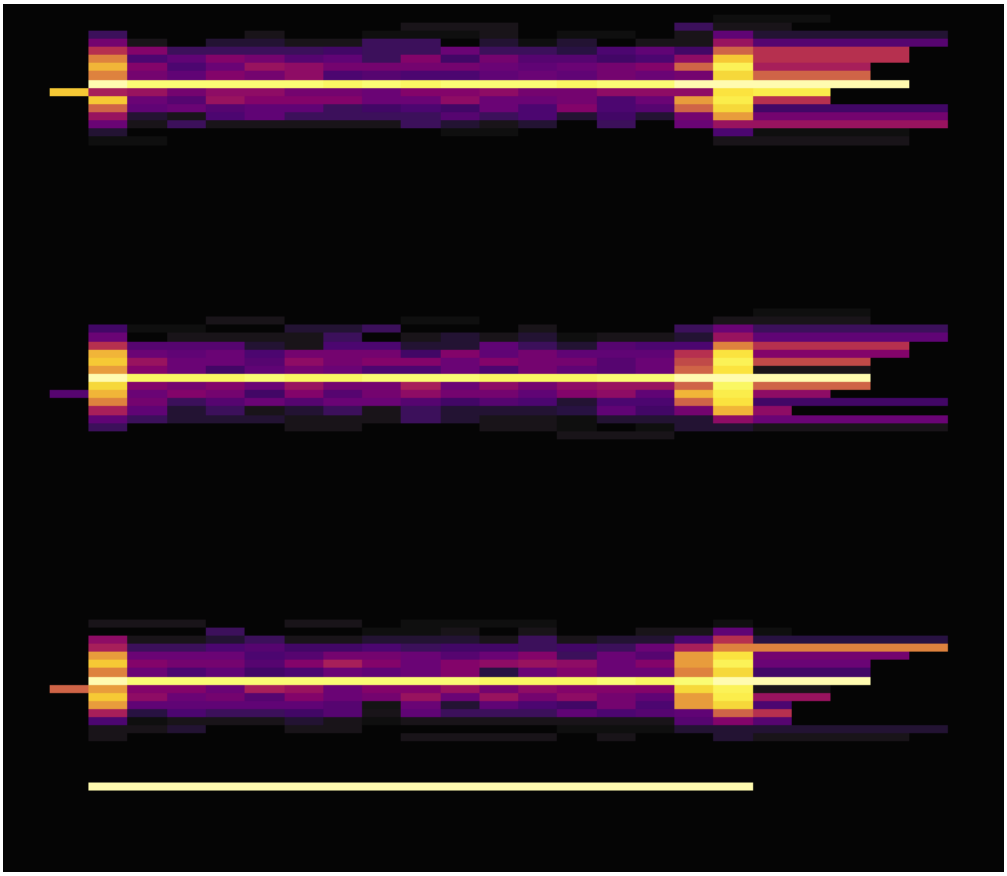


图 210: Y 轴明暗交替

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。

11.9.7.5.4 明亮斜线：需要关注业务模式

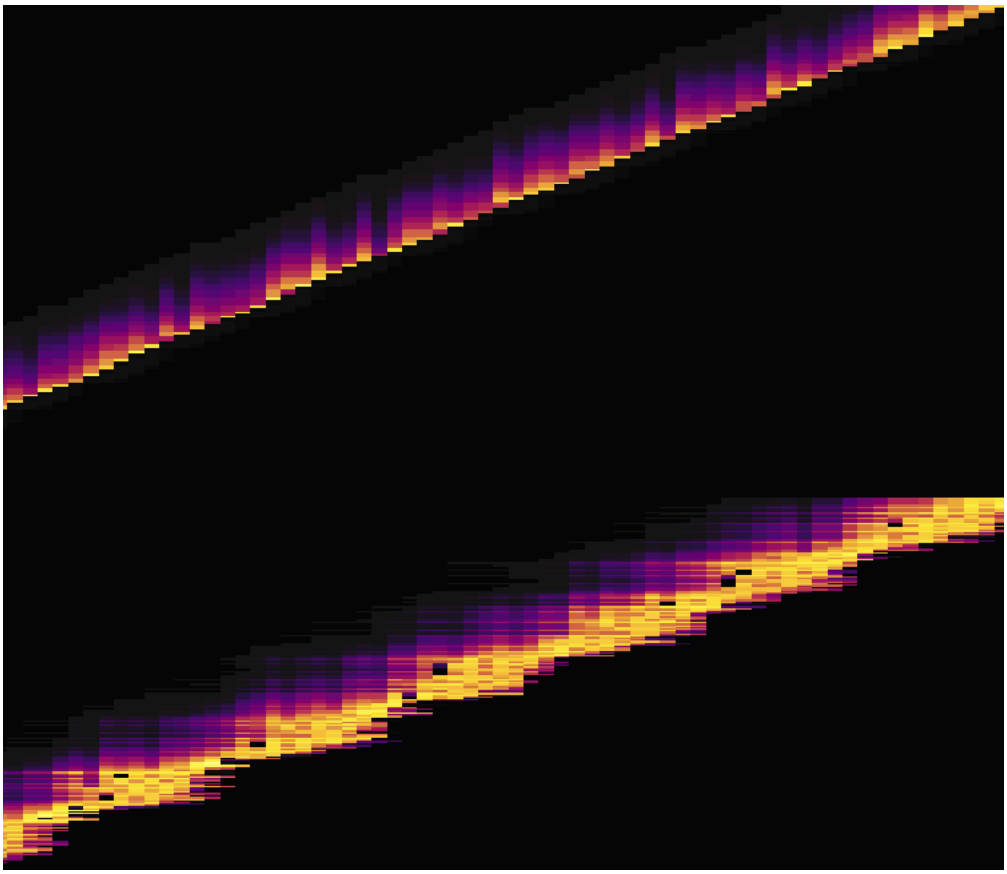


图 211: 明亮斜线

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。

11.9.7.6 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。

11.9.8 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。

11.9.8.1 访问关系图

登录 TiDB Dashboard 后点击左侧导航的集群诊断可以进入此功能页面：



图 212: 生成监控关系图首页

设置区间起始时间和区间长度参数后，点击生成监控关系图按钮后，会进入监控关系图页面。

11.9.8.2 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。

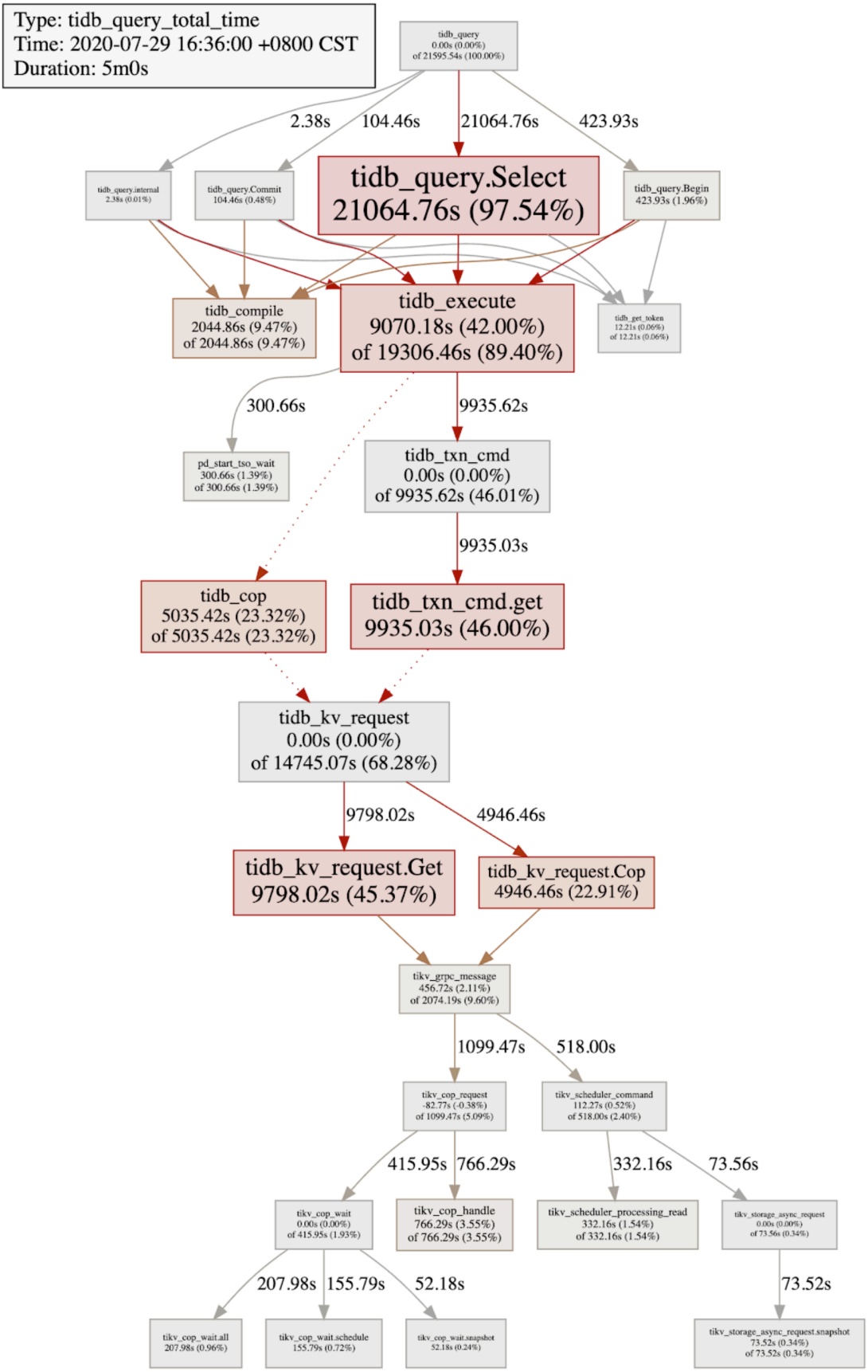


图 213: 监控关系图示例

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。

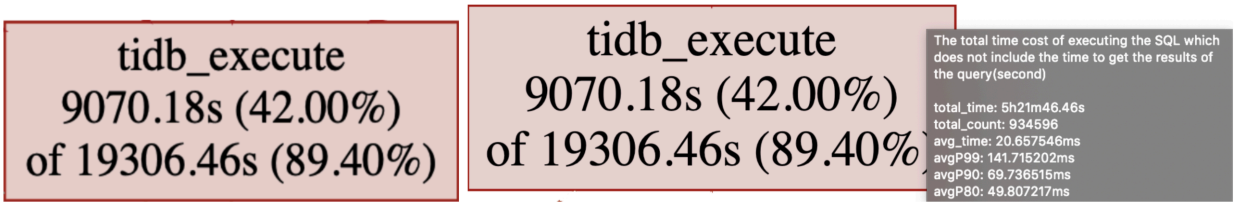


图 214: 监控关系图 tidb_execute 节点示例

11.9.8.2.1 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：

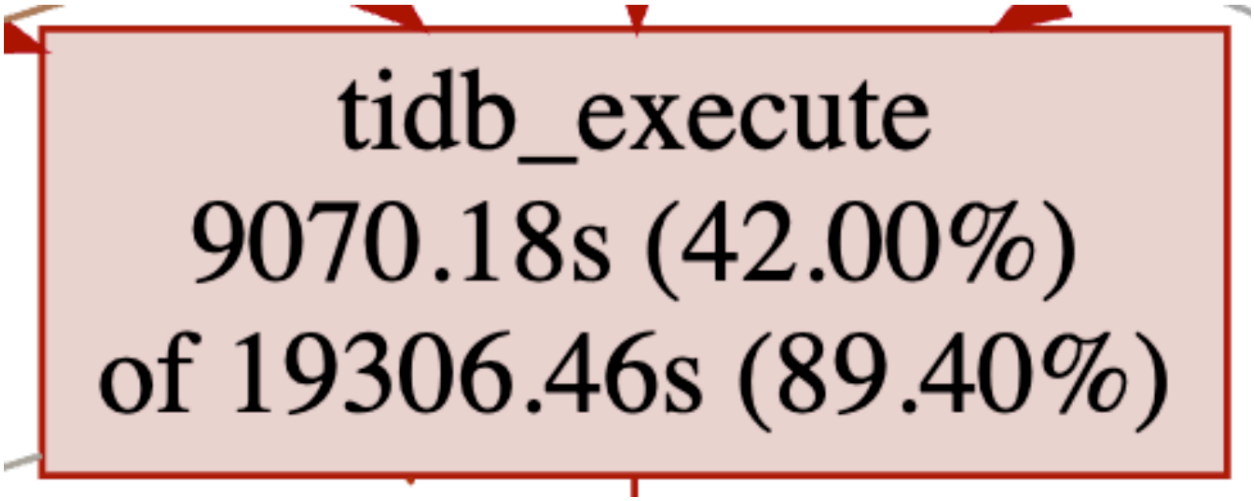


图 215: 监控关系图 tidb_execute 节点示例

- 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：

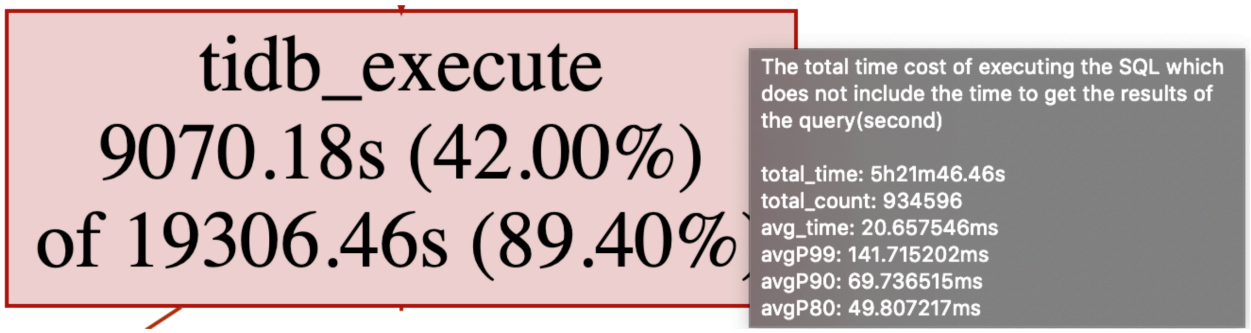


图 216: 监控关系图 tidb_execute 节点注释

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。

11.9.8.2.2 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：

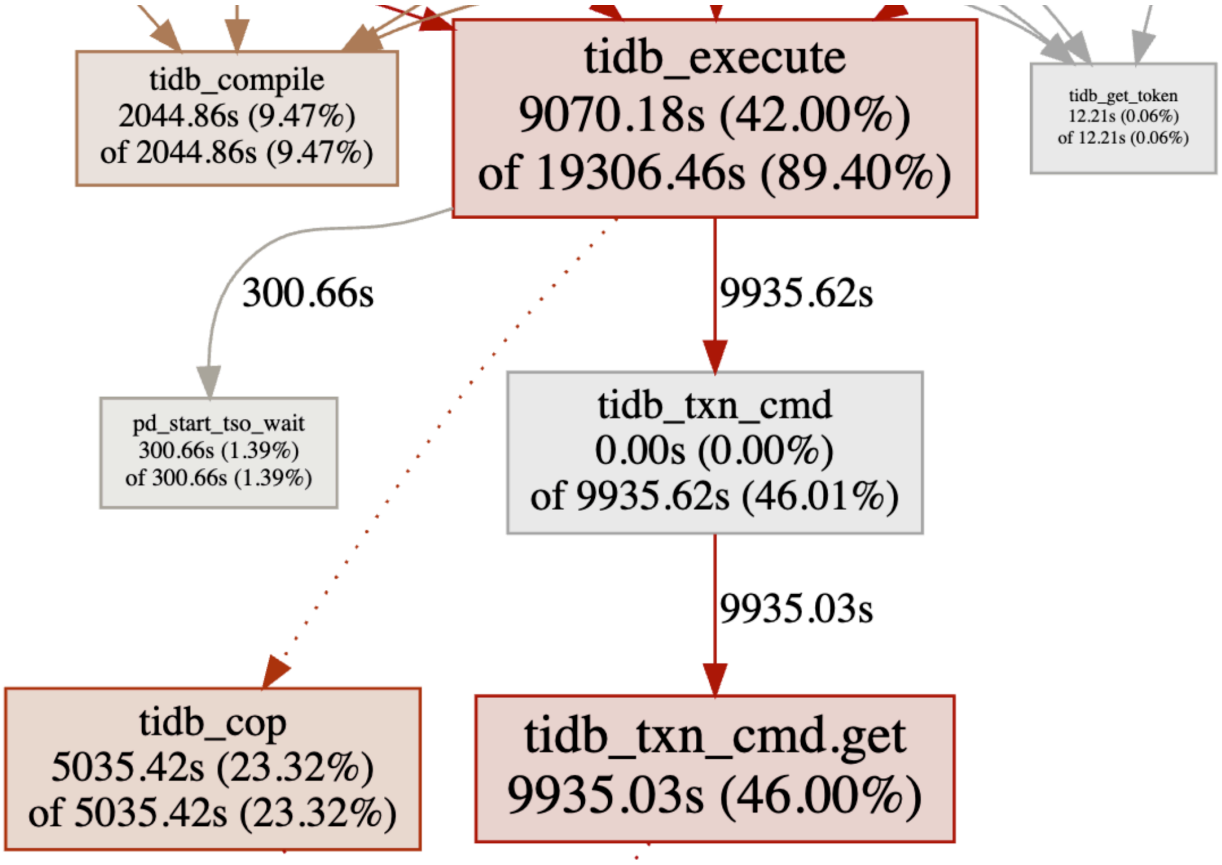


图 217: 监控关系图 tidb_execute 节点注释

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。

11.9.8.2.3 tidb_kv_request 及其父节点

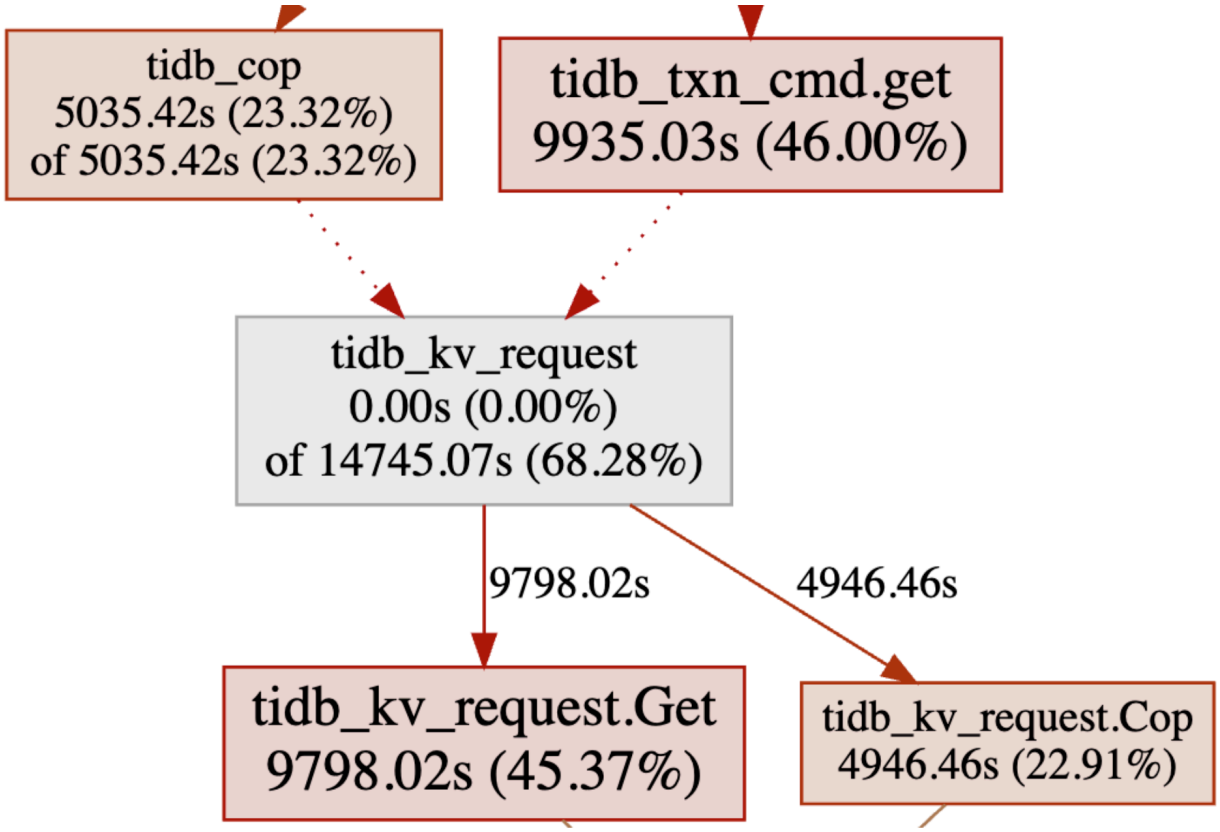


图 218: 监控关系图虚线节点关系

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。

11.9.9 SQL 语句分析

11.9.9.1 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 (...)`。

11.9.9.1.1 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：

- 登录后，左侧导航条点击“SQL 语句分析” (SQL Statements)。

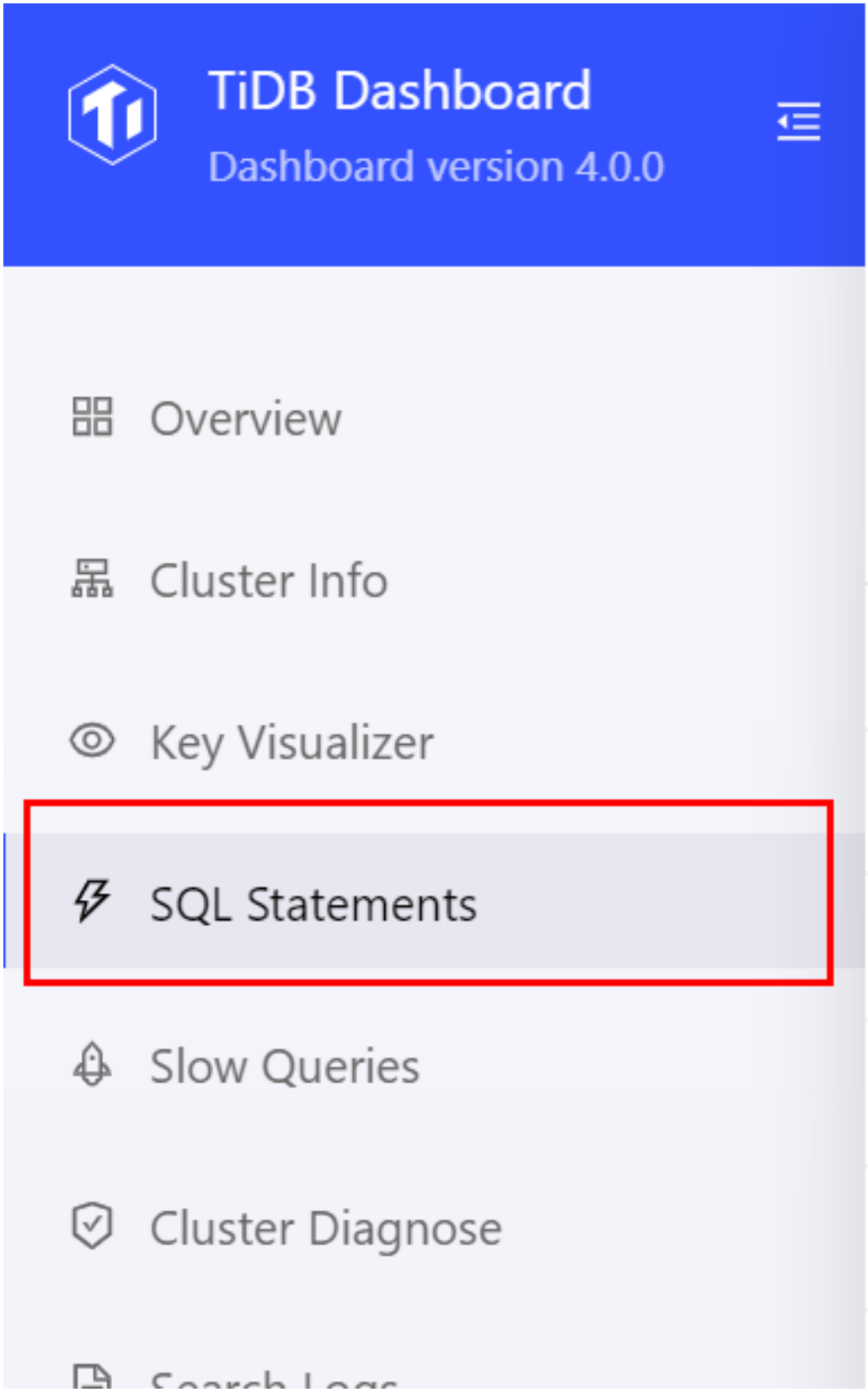

TiDB Dashboard
Dashboard version 4.0.0
Overview
Cluster Info
Key Visualizer
SQL Statements
Slow Queries
Cluster Diagnose

图 219: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/statement（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

SQL 语句分析页面所展示的所有数据都来自于 TiDB Statement 系统表，参见 Statement Summary Tables 文档了解该系统表的详细情况。

> **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。

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。

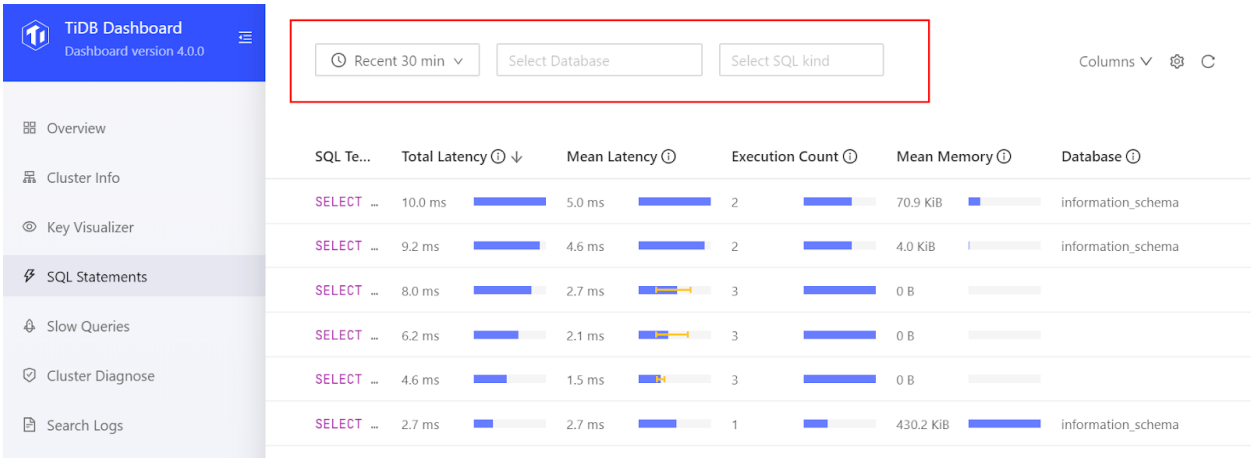


图 220: 修改过滤条件

显示其他字段

页面顶部选择列 (Columns) 选项中可选择显示更多列，可将鼠标移动到列名右侧的 (i) 图标处查看列的说明：

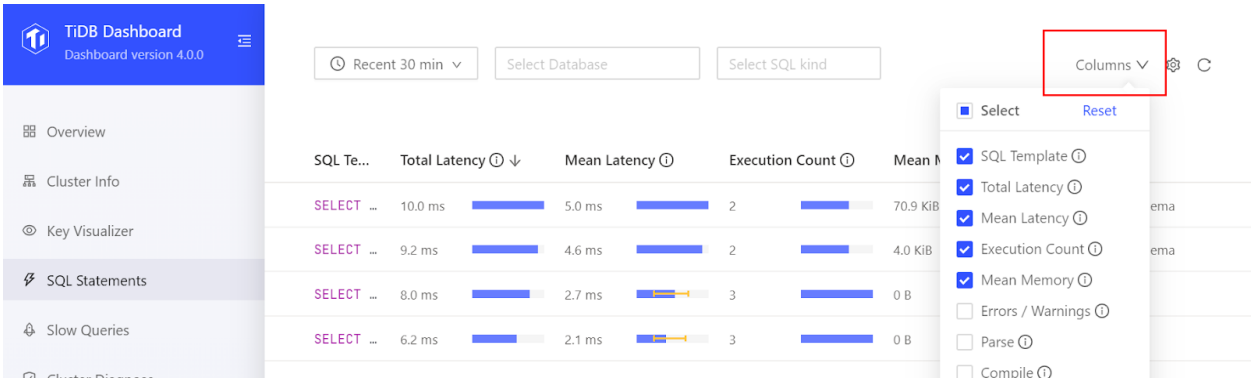


图 221: 选择列

修改列表排序依据

列表默认以累计耗时 (Total Latency) 从高到低进行排序，点击不同的列标题可以修改排序依据或切换排序顺序：

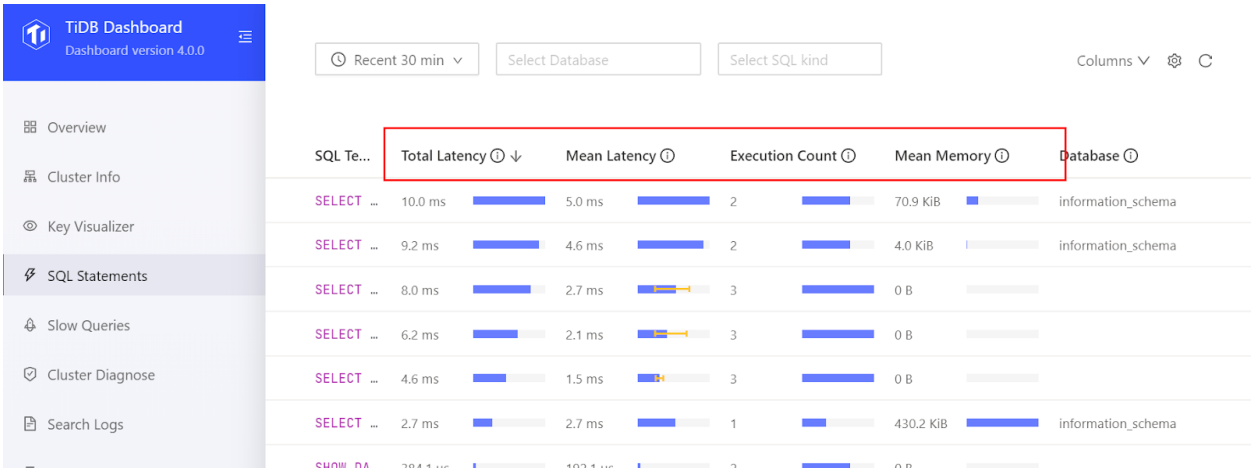


图 222: 修改列排序

修改数据收集设置

在列表页面，点击顶部右侧的设置 (Settings) 按钮，即可对 SQL 语句分析功能进行设置：

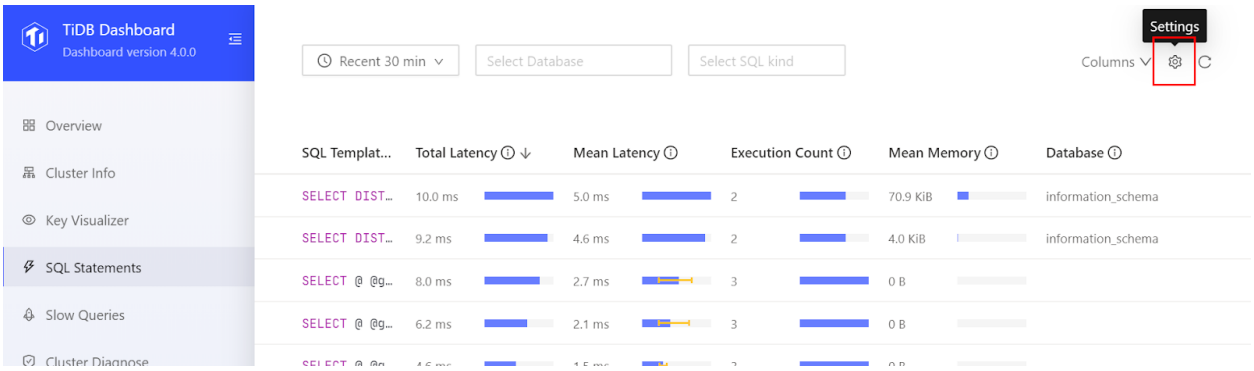


图 223: 设置入口

点击后设置界面如下图所示：

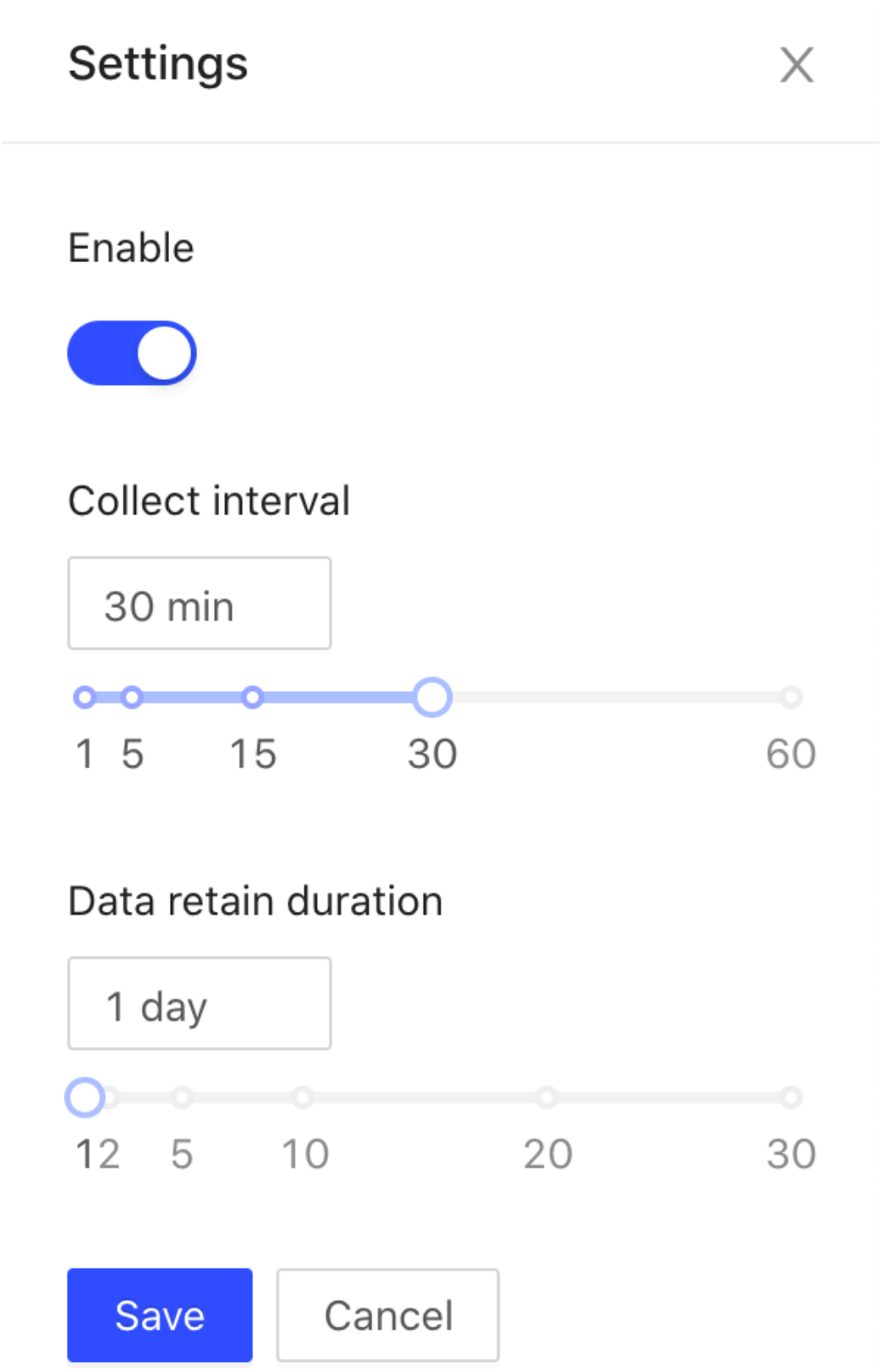

Settings
Enable
Collect interval
30 min
1 5 15 30 60
Data retain duration
1 day
12 5 10 20 30
Save
Cancel

图 224: 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。

11.9.9.1.2 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。

11.9.9.2 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，这此信息包括三大部分：

- SQL 语句概况：包括 SQL 模板，SQL 模板 ID，当前查看的时间范围，执行计划个数以及执行所在的数据库（下图区域 1）
- 执行计划列表：如果该 SQL 语句有多个执行计划，则显示该列表，可以选择不同的执行计划，在列表下方会显示选中的执行计划详情；如果只有一个执行计划，则该列表不显示（下图区域 2）
- 执行计划详情：显示选中的执行计划的详细信息，具体见下一小节（下图区域 3）

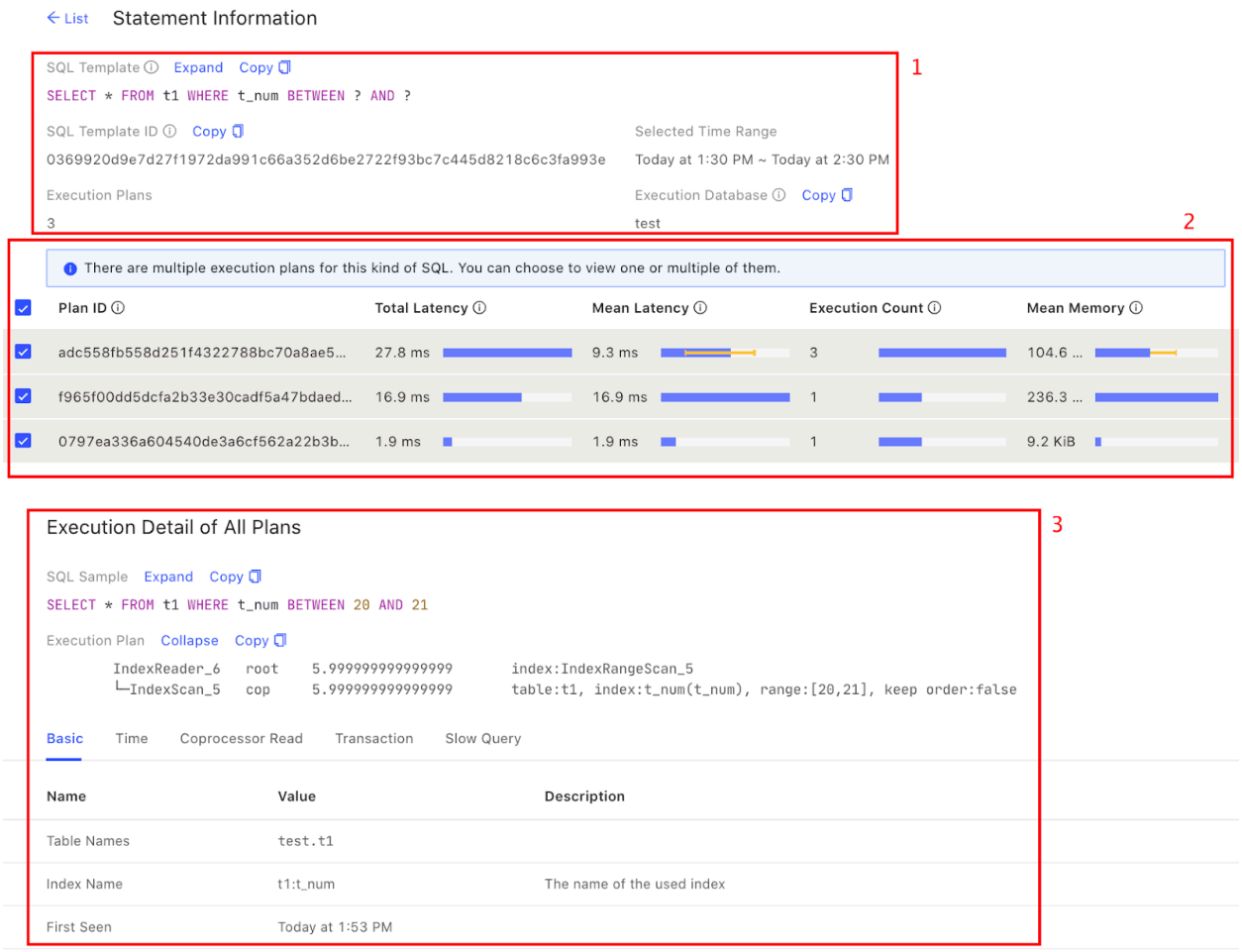


图 225: 详情

11.9.9.2.1 执行计划详情

执行计划详情包括以下内容：

- SQL 样本：该计划对应的实际执行的某一条 SQL 语句文本。时间范围内任何出现过的 SQL 都可能作为 SQL 样本。
- 执行计划：执行计划的完整内容，参阅理解 TiDB 执行计划文档了解如何解读执行计划。如果选择了多个执行计划，则显示的是其中任意一个。
- 其他关于该 SQL 的基本信息、执行时间、Coprocessor 读取、事务、慢查询等信息，可点击相应标签页标题切换。

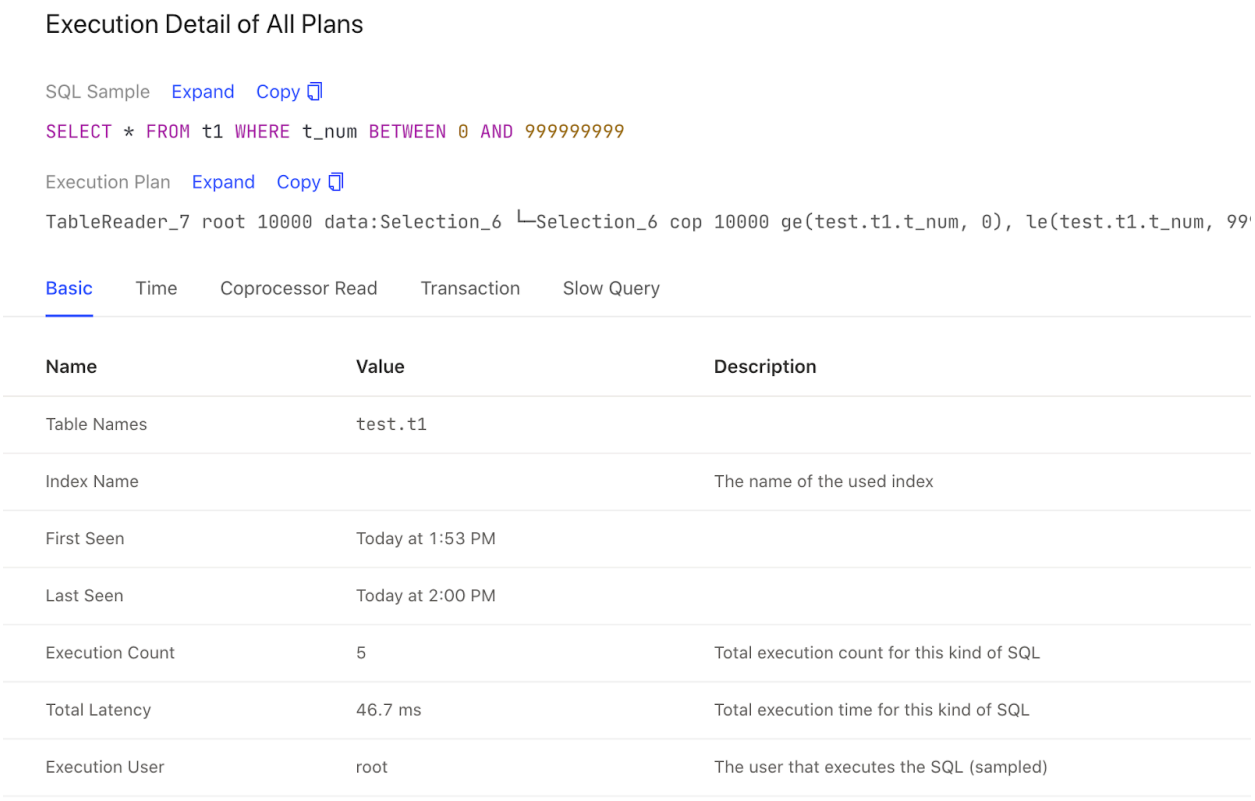


图 226: 执行计划详情

基本信息

包含关于表名、索引名、执行次数、累计耗时等信息。描述 (Description) 列对各个字段进行了具体描述。

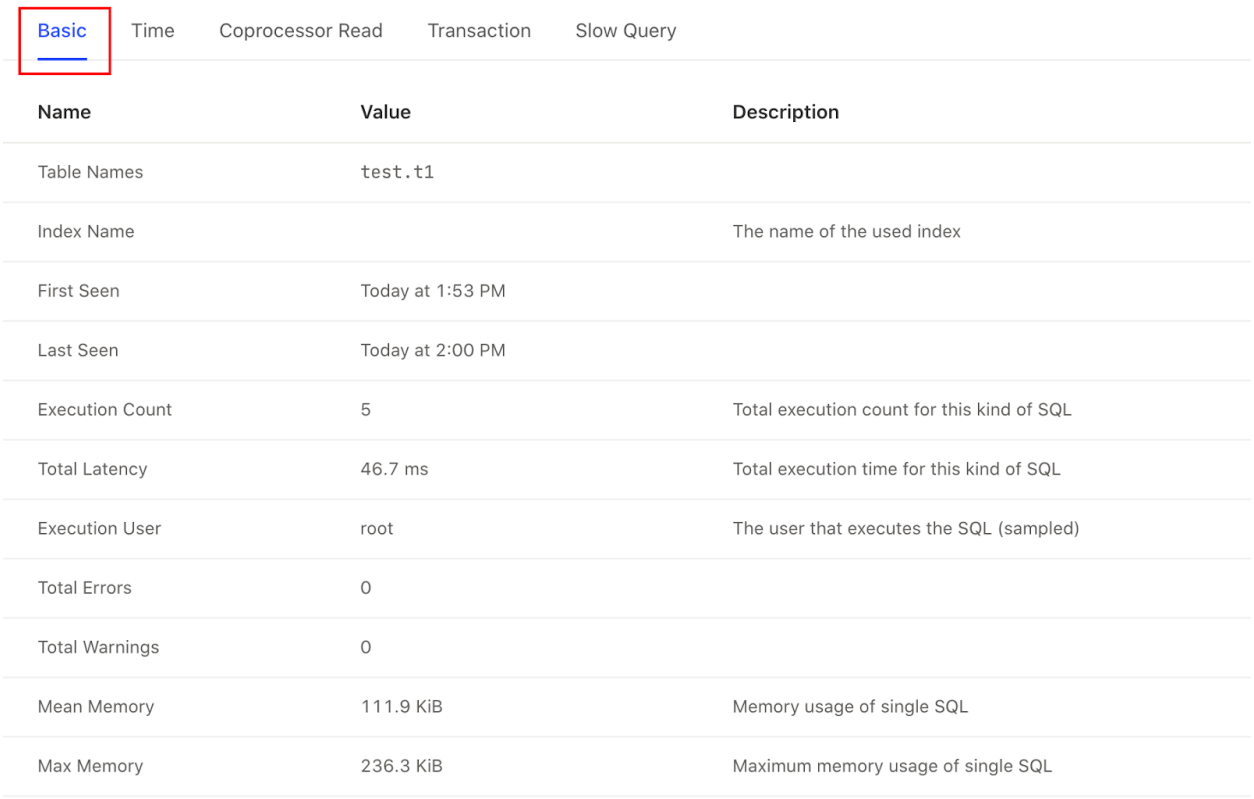


图 227: 基本信息

执行时间

显示执行计划执行的各阶段所耗费时间。

> **注意：**
>
> 由于单个 SQL 语句内部可能有并行执行的操作，因此各阶段累加时间可能超出该 SQL 语句的实际执行时间。

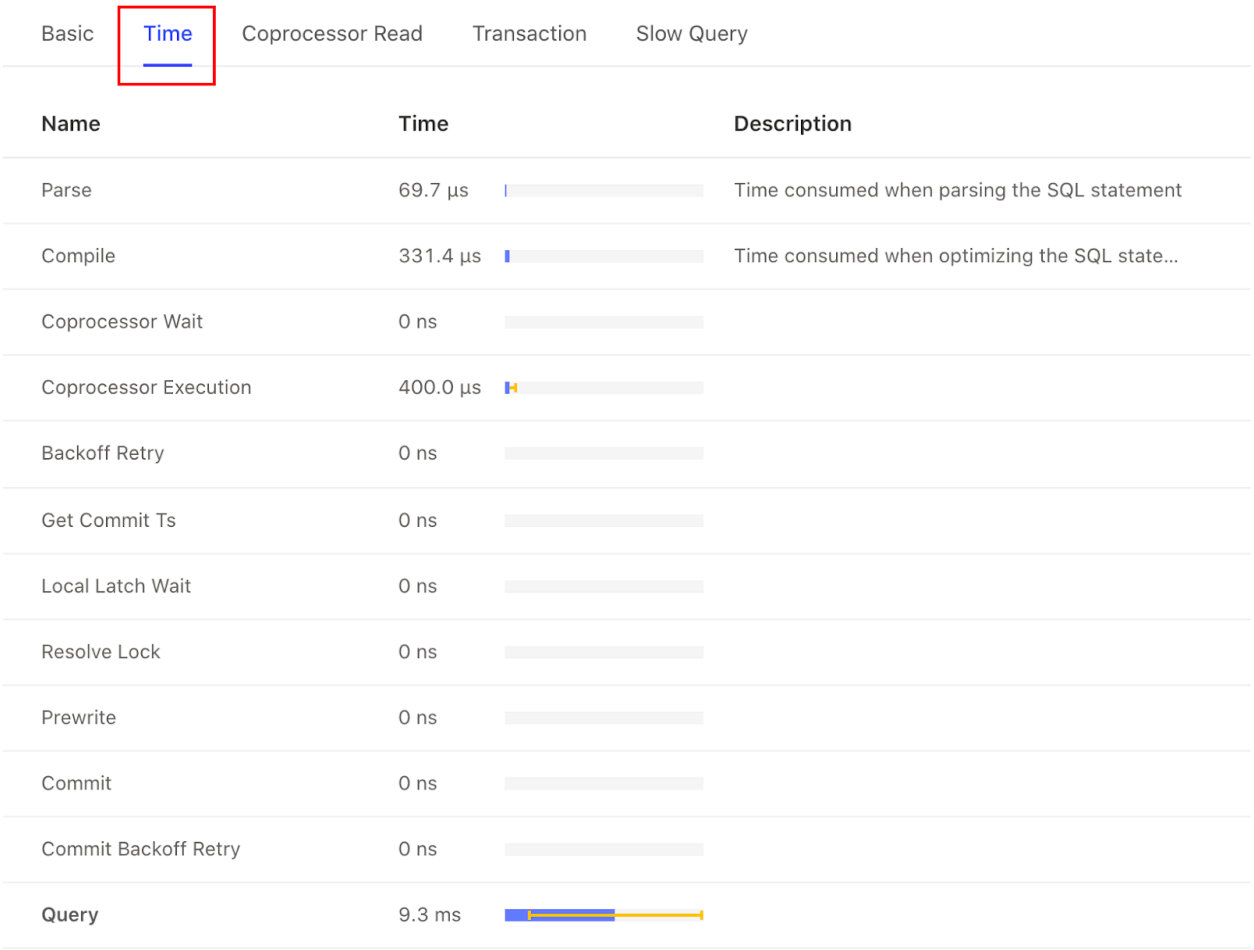


图 228: 执行时间

Coprocessor 读取

显示 Coprocessor 读取的相关信息。

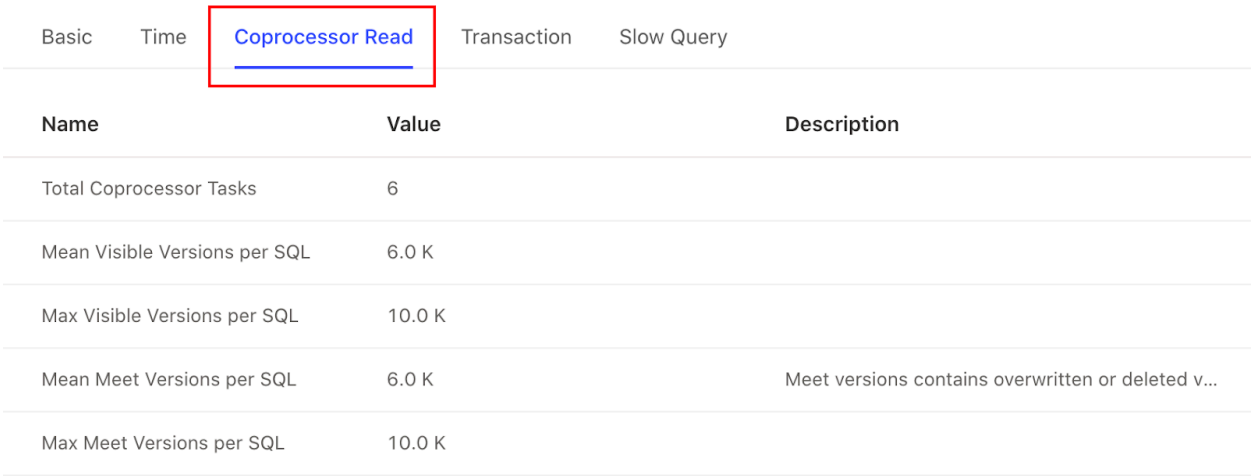


图 229: Coprocessor 读取

事务

显示执行计划与事务相关的信息，比如平均写入 key 个数，最大写入 key 个数等。

Basic Time Coprocessor Read **Transaction** Slow Query

Name	Value	Description
Mean Affected Rows	0	
Total Backoff Count	0	
Mean Written Keys	0	
Max Written Keys	0	
Mean Written Data Size	0 B	
Max Written Data Size	0 B	
Mean Prewrite Regions	0	
Max Prewrite Regions	0	
Mean Transaction Retries	0	
Max Transaction Retries	0	

图 230: 事务

慢查询

如果该执行计划执行过慢，则在慢查询标签页下可以看到其关联的慢查询记录。

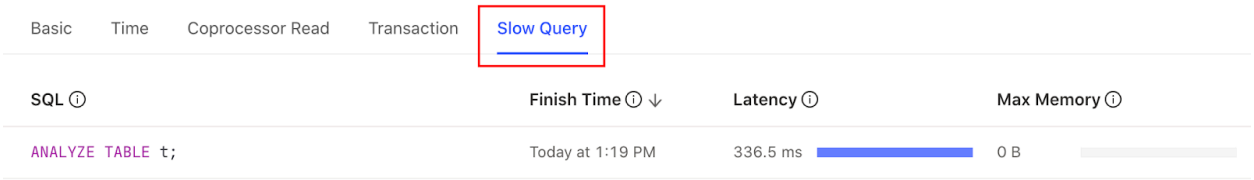


图 231: 慢查询

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。

11.9.10 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 配置`enable-slow-log` 开启或禁用。

11.9.10.1 访问列表页面

可以通过以下两种方法访问慢查询页面：

- 登录后，左侧导航条点击慢查询 (Slow Queries)：

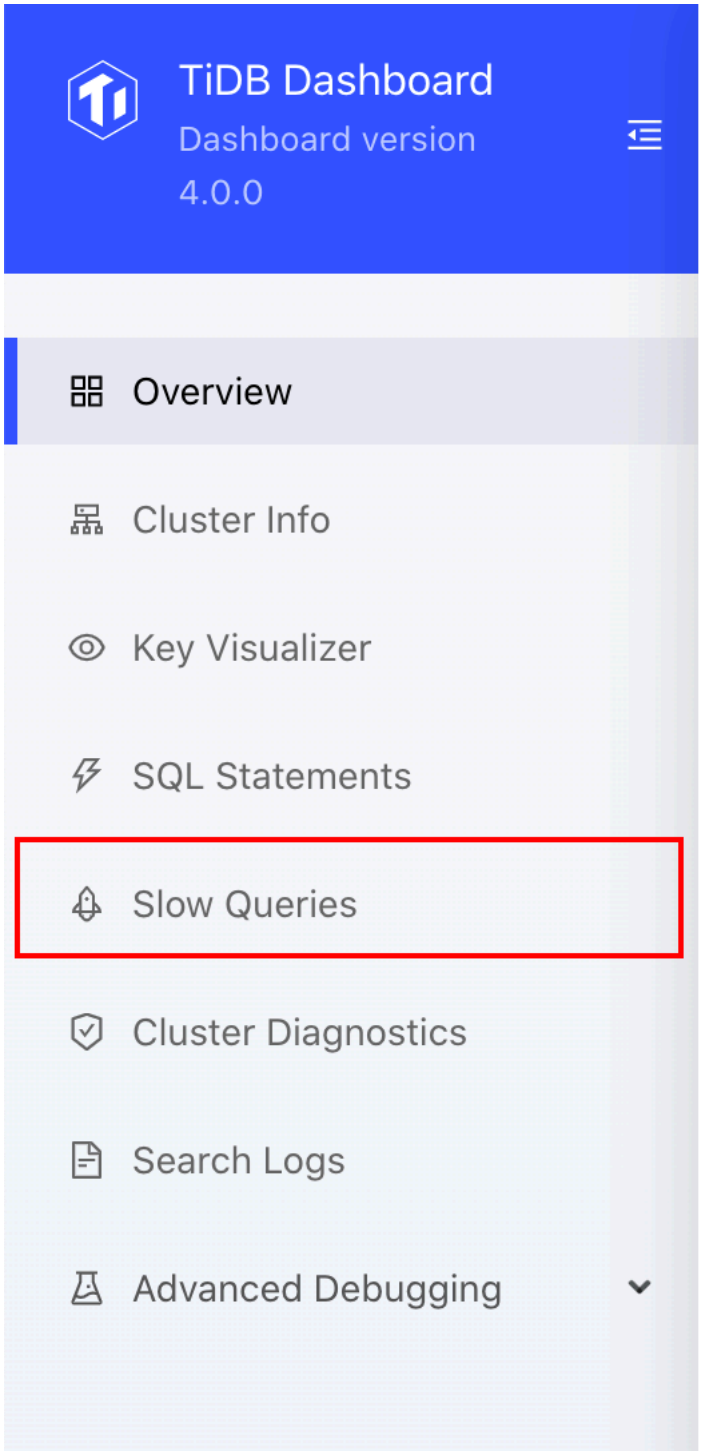


图 232: access 访问页面

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/slow_query（将 127.0.0.1:2379 替换为任意实际 PD 地址和端口）。

慢查询页面所展示的所有数据都来自于 TiDB 慢查询系统表及慢查询日志，参见慢查询日志文档了解详细情况。

11.9.10.1.1 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。

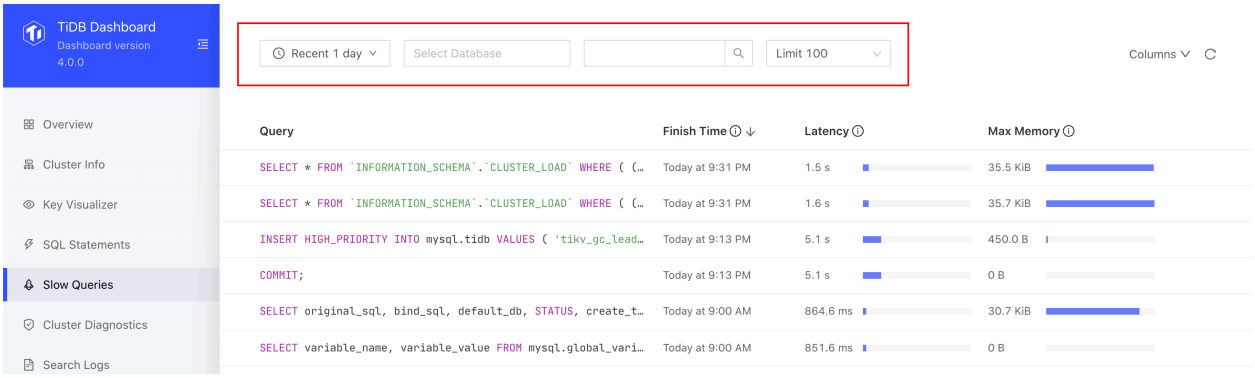


图 233: 修改列表过滤条件

11.9.10.1.2 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：

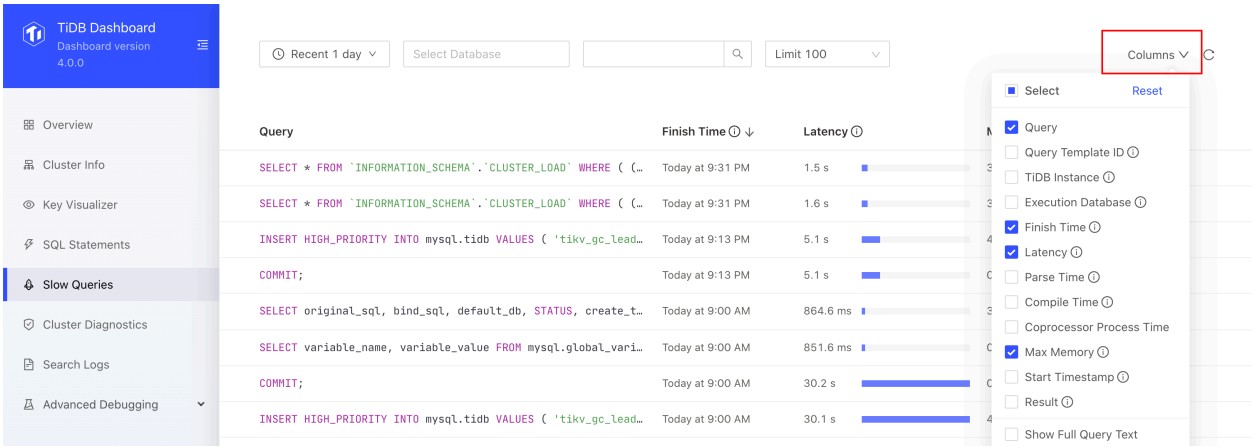


图 234: 显示更多列信息

11.9.10.1.3 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：

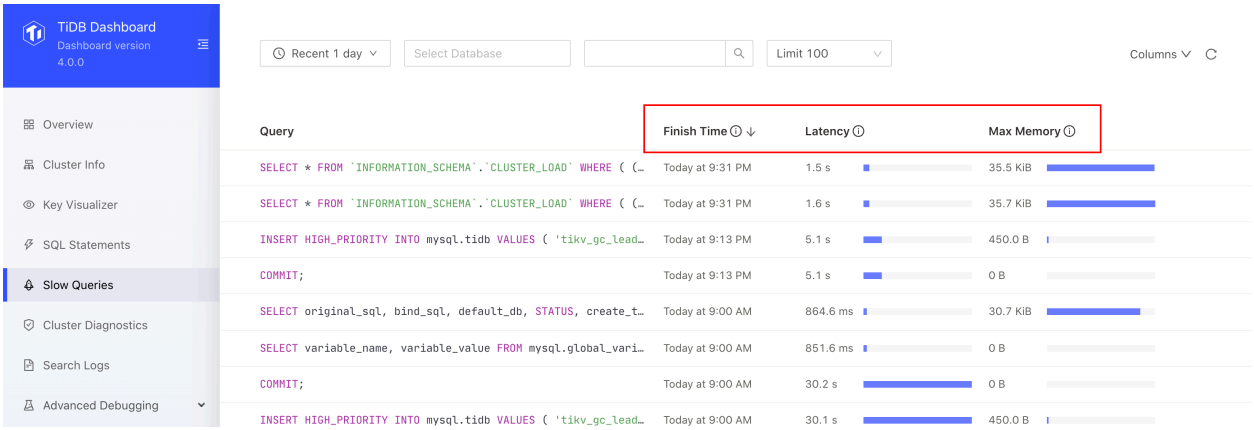


图 235: 修改列表排序依据

11.9.10.2 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，参阅理解 TiDB 执行计划文档了解如何解读执行计划（下图中区域 2）
- 其他分类好的 SQL 执行信息（下图中区域 3）

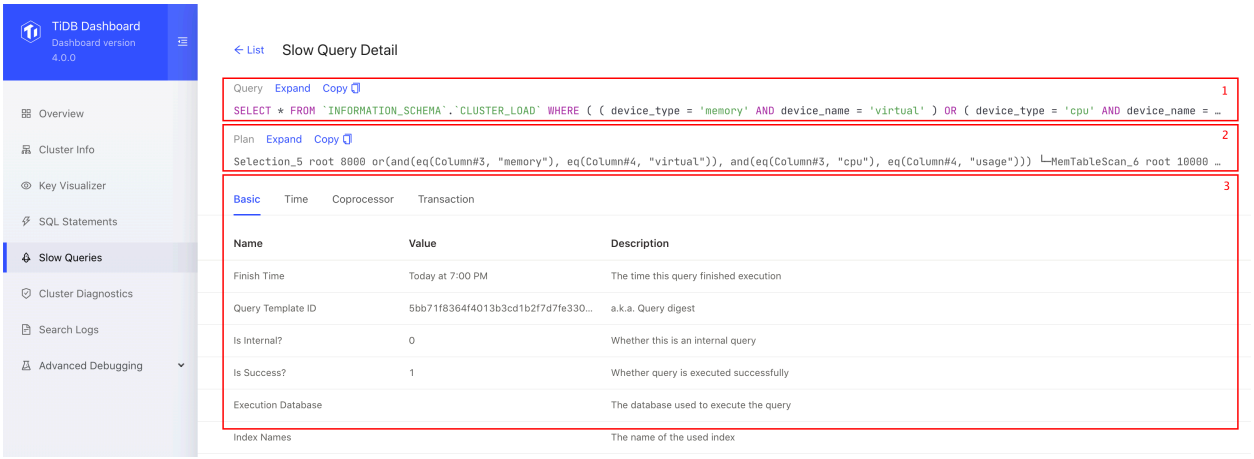


图 236: 查看执行详情

点击展开 (Expand) 链接可以展开相应项的完整内容，点击复制 (Copy) 链接可以复制完整内容到剪贴板。

点击标签页标题可切换显示不同分类的 SQL 执行信息：

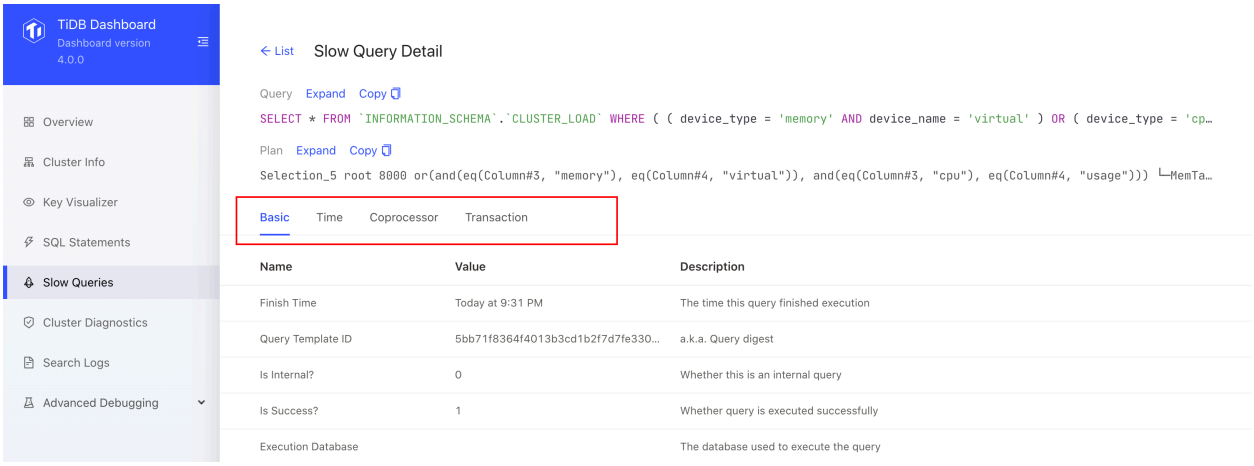


图 237: 显示不同分类执行信息

11.9.11 集群诊断页面

11.9.11.1 TiDB Dashboard 集群诊断页面

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

集群诊断是在指定的时间范围内，对集群可能存在的问题进行诊断，并将诊断结果和一些集群相关的负载监控信息汇总成一个诊断报告。诊断报告是网页形式，通过浏览器保存后可离线浏览和传阅。

> **注意：**
>
> 集群诊断功能依赖于集群中部署有 Prometheus 监控组件，参见 TiUP 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。

11.9.11.1.1 访问

可以通过以下两种方法访问集群诊断页面：

- 登录后，左侧导航条点击集群诊断 (Cluster Diagnose)：

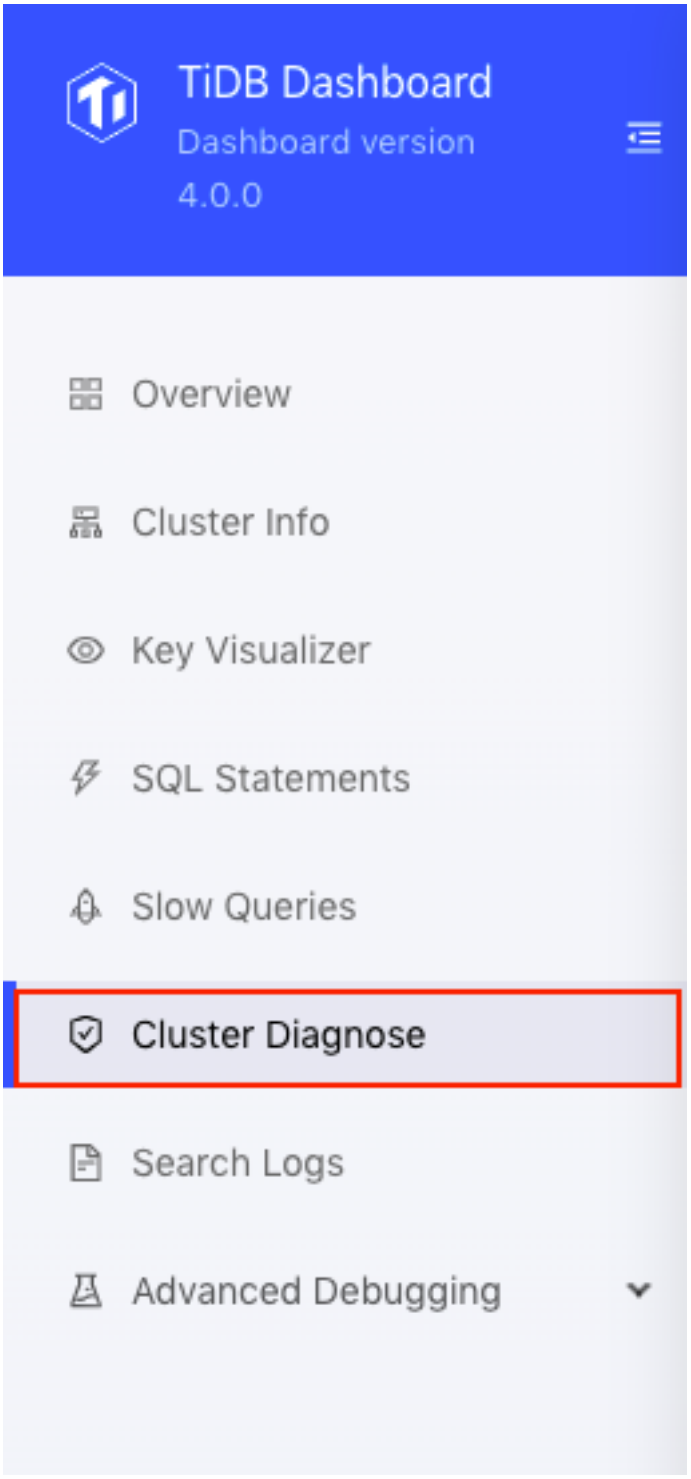


图 238: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/diagnose（将 127.0.0.1:2379 替换为任意实际 PD 地址和端口）。

11.9.11.1.2 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 2020-05-21 14:40:00。
2. 设置区间长度。例如 10 min 。
3. 点击开始。

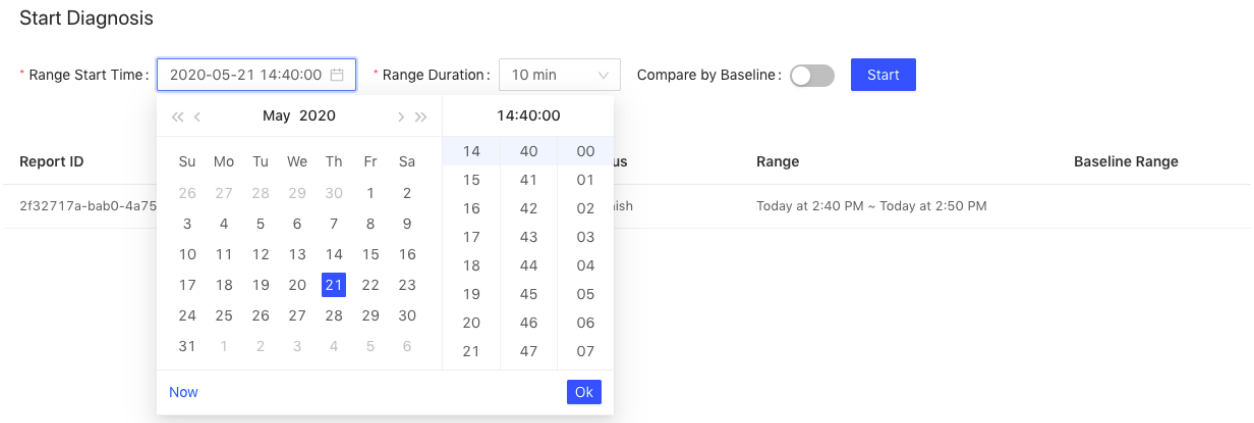


图 239: 生成单个时间段的诊断报告

> **注意：**
>
> 建议生成报告的时间范围在 1 min ~ 60 min 内，目前不建议生成超过 1 小时范围的报告。

以上操作会生成 2020-05-21 14:40:00 至 2020-05-21 14:50:00 时间范围的诊断报告。点击开始 (start) 后，会看到以下界面，生成进度 (progress) 是生成报告的进度条，生成报告完成后，点击查看报告 (View Full Report) 即可。

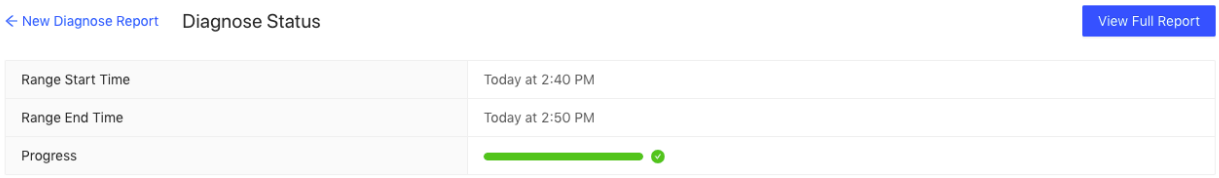


图 240: 生成报告的进度

11.9.11.1.3 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：2020-05-21 14:40:00 ~ 2020-05-21 14:45:00，系统异常时间。
- 系统正常时间段：2020-05-21 14:30:00 ~ 2020-05-21 14:35:00，系统正常时间。

生成以上两个时间范围的对比报告的步骤如下：

1. 设置区间的开始时间，即异常时间段的开始时间，如 2020-05-21 14:40:00。
2. 设置区间长度。一般只系统异常的持续时间，例如 5 min。
3. 开启与基线区间对比开关。
4. 设置基线开始时间，即想要对比的系统正常时段的开始时间，如 2020-05-21 14:30:00。
5. 点击开始。

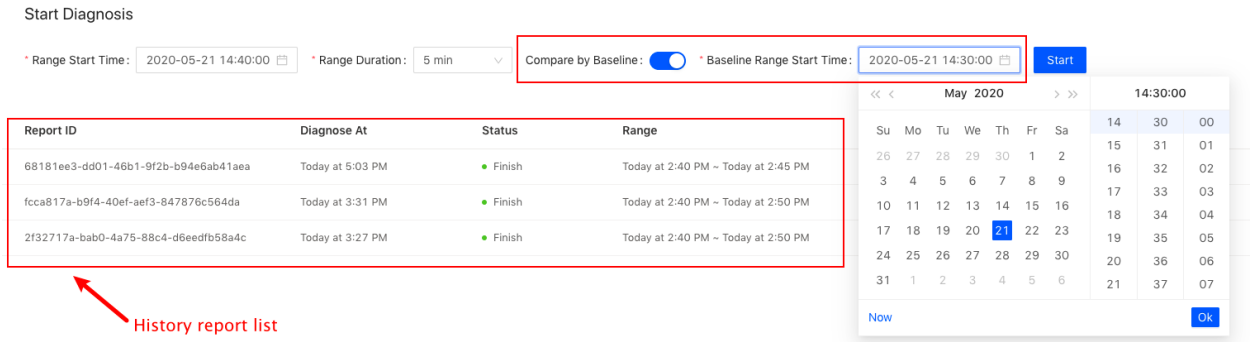


图 241: 生成对比报告

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。

11.9.11.2 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。

11.9.11.2.1 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 242: 示例报表

上图中，最上面蓝框内的 Total Time Consume 是报表名。下方红框内的内容是对该报表意义的解释，以及报表中各个字段的含义。

报表中小按钮图标的解释如下：

- i 图标：鼠标移动到 i 图标处会显示该行的说明注释。
- expand：点击 expand 会看到这项监控更加详细的信息。如是上图中 `tidb_get_token` 的详细信息包括各个 TiDB 实例的延迟监控信息。
- fold：和 expand 相反，用于把监控的详细信息折叠起来。

所有监控基本上和 TiDB Grafna 监控面板上的监控内容相对应，发现某个模块异常后，可以在 TiDB Grafna 监控面板上查看更多详细的监控信息。

另外，报表中统计的 `TOTAL_TIME` 和 `TOTAL_COUNT` 由于是从 Prometheus 读取的监控数据，其统计会有一些计算上的精度误差。

以下介绍诊断报告的各部分内容。

基本信息

Report Time Range

Report Time Range 表显示生成报告的时间范围，包括开始时间和结束时间。

Report Time Range

START_TIME	END_TIME
2020-05-21 06:40:00	2020-05-21 06:50:00

图 243: 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 244: 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 245: Cluster Info 报表

上表中各个字段含义如下：

- TYPE：节点类型。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。
- VERSION：对应节点的语义版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。

诊断信息

TiDB 内置自动诊断的结果，具体各字段含义以及介绍可以参考information_schema.inspection_result 系统表的内容。

负载信息

Node Load Info

Node Load Info 表显示服务器节点的负载信息，包括时间范围内，服务器以下指标的平均值 (AVG)、最大值 (MAX)、最小值 (MIN)：

- CPU 使用率，最大值是 100%
- 内存使用率
- 磁盘 I/O 使用率
- 磁盘写延迟
- 磁盘读延迟
- 磁盘每秒的读取字节数
- 磁盘每秒的写入字节数
- 节点网络每分钟收到的字节数
- 节点网络每分钟发送的字节数
- 节点正在使用的 TCP 连接数
- 节点所有的 TCP 连接数

Node Load Info

METRIC_NAME	instance	AVG	MAX	MIN
node_cpu_usage expand		40.42%	99.6%	6.98%
node_mem_usage expand		66.9%	96.28%	33.65%
node_disk_io_utilization expand		16%	97%	0%
Disk write latency expand		4000 us	50 ms	0 us
Disk read latency expand		2000 us	6000 us	0 us
tikv_disk_read_bytes expand		605.980 KB	15.842 MB	0.000 KB
tikv_disk_write_bytes expand		1.129 MB	12.235 MB	0.000 KB
node_network_in_traffic expand		225.706 KB	1.427 MB	0.000 KB
node_network_out_traffic expand		165.807 KB	1.659 MB	0.000 KB
node_tcp_in_use expand		19	47	5
node_tcp_connections expand		30	61	6

图 246: 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 247: 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 248: 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 249: 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 250: TiDB/PD goroutines count 报表

概览信息

Time Consumed by Each Component

Time Consumed by Each Component 显示包括集群中 TiDB、PD、TiKV 各个模块的监控耗时以及各项耗时的占比。默认时间单位是秒。用户可以用该表快速定位哪些模块的耗时较多。

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token expand		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
tidb_parse expand		0.00001	0.29	4603	0.0006	0.0004	0.0002	0.0001
tidb_compile expand		0.0008	17.82	463488	0.001	0.0009	0.0003	0.0002
tidb_execute expand		1	23237.39	463491	90.81	0.35	0.1	0.08
tidb_distsql_execution expand		0.0002	4.53	1064	0.13	0.1	0.02	0.01
tidb_cop expand		0.0005	12.39	1075	2.03	1.84	0.02	0.01
Transaction expand		1	23158.73	460023	8.16	7.91	5.38	0.06
tidb_transaction_local_latch_wait		0	0	0				
tidb_kv_backoff expand		0.00003	0.58	291	0.002	0.002	0.002	0.002
KV request expand		1.65	38325.58	2300949	2.03	1.84	0.04	0.03
Slow query expand		0.01	323.65	280	65.5	65.21	62.26	58.98
tidb_slow_query_cop_process		0	0	280	0.001	0.001	0.0009	0.0008
tidb_slow_query_cop_wait		0	0	280	0.001	0.001	0.0009	0.0008
DDL job		0	0	0				
tidb_ddl_worker		0	0	0				
tidb_ddl_update_self_version		0	0	0				
tidb_owner_handle_syncer		0	0	0				
Batch add index								
tidb_ddl_deploy_syncer		0	0	0				
Schema load expand		0.000008	0.19	27	0.06	0.06	0.06	0.05
TiDB meta operation expand		0.0001	3.21	1200	0.13	0.1	0.01	0.005
TiDB auto id request		0	0	0				
TiDB auto analyze expand		0.003	78.35	1	81.88	81.51	77.82	73.73
TiDB GC expand		0.000000001	0.00002	3	1	0.99	0.9	0.8
tidb_gc_push_task		0	0	0				
tidb_batch_client_unavailable		0	0	0				
tidb_batch_client_wait		0	0	0				
pd_tso_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_tso_wait expand		0.07	1662.29	920545	0.03	0.01	0.004	0.003
PD client cmd expand		0.15	3370.87	2761712	0.03	0.01	0.004	0.003
pd_client_request_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_grpc_completed_commands expand		0.00005	1.24	5255	0.01	0.01	0.006	0.005
pd_operator_finish								
pd_operator_step_finish								
Etcd transactions expand		0.000008	0.17	207	0.008	0.008	0.006	0.002
pd_region_heartbeat expand		0.00006	1.33	195	0	0	0	0
etcd_wal_fsync expand		0.00001	0.33	288	0.02	0.01	0.006	0.004
pd_peer_round_trip								

图 251: 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：

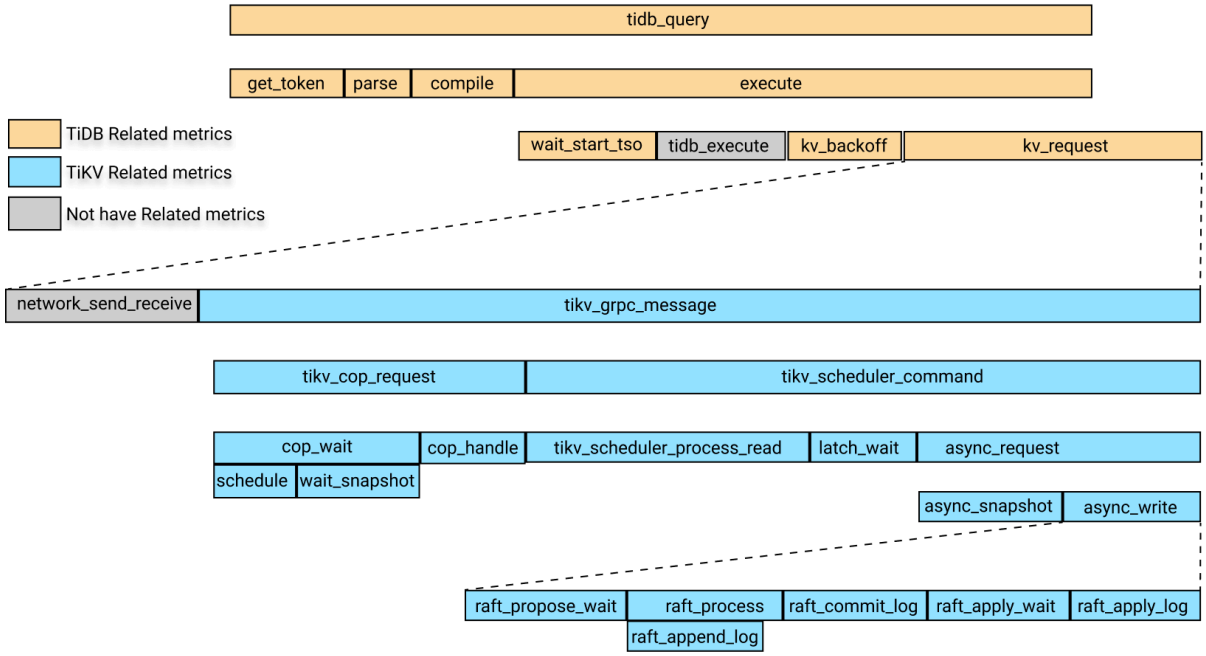


图 252: 各个模块耗时关系图

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 253: Errors Occurred in Each Component 报表

TiDB/PD/TiKV 的具体监控信息

这部分包括了 TiDB/PD/TiKV 更多的具体的监控信息。

TiDB 相关监控信息

Time Consumed by TiDB Component

Time Consumed by TiDB Component 表显示 TiDB 的各项监控耗时以及各项耗时的占比。和Time Consumed by Each Component 表类似，但是这个表的 label 信息会更丰富，细节更多。

TiDB Server Connections

TiDB Server Connections 表显示 TiDB 各个实例的客户端连接数。

TiDB Transaction

TiDB Transaction 表显示 TiDB 事务相关的监控。

METRIC_NAME	LABEL	TOTAL_VALUE	TOTAL_COUNT	P999	P99	P90	P80
tidb_transaction_retry_num		0	0	0	0	0	0
tidb_transaction_statement_num expand		6854588	6852371	4	4	3	2
tidb_txn_region_num expand		1070770	706887	3	2	2	2
tidb_txn_kv_write_num expand		513386	181312	234	2	2	2
tidb_txn_kv_write_size expand		266.772 MB	181296	116.913 KB	1.996 KB	1.905 KB	1.805 KB
tidb_load_safepoint_total_num expand		16					
tidb_lock_resolver_total_num expand		420514					

图 254: 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 255: TiDB DDL Owner 报表

上表表示从 2020-05-21 14:40:00 开始，集群的 DDL owner 在 10.0.1.13:10080 节点。如果 owner 发生变更，上表会有多行数据，其中 MinTime 列表示已知对应 Owner 的最小时间。

> 注意：
>
> 如果 owner 信息为空，不代表这个时间段内一定没有 owner，因为这里是依靠 ddl_worker 的监控信息来判断 DDL owner 的。也可能是这个时间段内 ddl_worker 没有做任何 DDL job 导致 owner 信息为空。

TiDB 中其他部分监控表如下：

- Statistics Info：TiDB 统计信息的相关监控。
- Top 10 Slow Query：报表时间范围内 Top 10 的慢查询信息。
- Top 10 Slow Query Group By Digest：报表时间范围内 Top 10 的慢查询信息，并按照 SQL 指纹聚合。
- Slow Query With Diff Plan：报表时间范围内执行计划发生变更的 SQL 语句。

PD 相关监控信息

PD 模块相关监控的报表如下：

- Time Consumed by PD Component：PD 中相关模块的耗时监控
- Scheduled Leader/Region：报表时间范围内集群发生的 `balance-region` 和 `balance leader` 监控，比如从 `tikv_note_1` 上调度走了多少个 leader，调度进了多少个 leader。
- Cluster Status：集群的状态信息，包括总 TiKV 数量、总集群存储容量、Region 数量、离线 TiKV 的数量等信息。
- Store Status：记录各个 TiKV 节点的状态信息，包括 Region score、 leader score、 Region/leader 的数量。
- etcd Status：PD 内部的 etcd 相关信息。

TiKV 相关监控信息

TIKV 模块的相关监控报表如下：

- Time Consumed by TiKV Component：TiKV 中相关模块的耗时监控。
- Time Consumed by RocksDB：TiKV 中 RocksDB 的耗时监控。
- TiKV Error：TiKV 中各个模块相关的 error 信息。
- TiKV Engine Size：TiKV 中各个节点 column family 的存储数据大小。
- Coprocessor Info：TiKV 中 Coprocessor 模块相关的监控。
- Raft Info：TiKV 中 Raft 模块的相关监控信息。
- Snapshot Info：TiKV 中 snapshot 相关监控信息。
- GC Info：TiKV 中 GC 相关的监控信息。
- Cache Hit：TiKV 中 Rocksdb 的各个缓存的命中率监控信息。

配置信息

配置信息中，部分模块的配置信息可以显示报告时间范围内的配置值，有部分配置则因为无法获取到历史的配置值，所以是生成报告时刻的当前配置值。

在报告时间范围内，以下表包括部分配置的在报告时间范围的开始时间的值：

- Scheduler Initial Config：PD 调度相关配置在报告开始时间的初始值。
- TiDB GC Initial Config：TiDB GC 相关配置在报告开始时间的初始值。
- TiKV RocksDB Initial Config：TiKV RocksDB 相关配置在报告开始时间的初始值。
- TiKV RaftStore Initial Config：TiKV RaftStore 相关配置在报告开始时间的初始值。

在报表时间范围内，如若有些配置被修改过，以下表包括部分配置被修改的记录：

- Scheduler Config Change History
- TiDB GC Config Change History

- TiKV RocksDB Config Change History
- TiKV RaftStore Config Change History

示例：

APPROXIMATE_CHANGE_TIME	CONFIG_ITEM	VALUE
2020-05-22T20:00:00+08:00	leader-schedule-limit	4
2020-05-22T20:07:00+08:00	leader-schedule-limit	8

图 256: 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

11.9.11.2.2 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 257: Compare Report Time Range 报表

其中 `t1` 是正常时间段，或者叫参考时间段，`t2` 是异常时间段。

下面是一些慢查询相关的报表：

- Slow Queries In Time Range t2：仅出现在 `t2` 时间段但没有出现在 `t1` 时间段的慢查询。
- Top 10 slow query in time range t1： `t1` 时间段的 Top10 慢查询。
- Top 10 slow query in time range t2： `t2` 时间段的 Top10 慢查询。

DIFF_RATIO 介绍

本部分以 Instance CPU Usage 为例介绍 DIFF_RATIO。

Instance CPU Usage

INSTANCE	JOB	t1.AVG	t1.MAX	t1.MIN	t2.AVG	t2.MAX	t2.MIN	AVG_DIFF_RATIO	MAX_DIFF_RATIO	MIN_DIFF_RATIO
172.16.5.40:10089	tidb	410%	410%	410%	1240%	1240%	1240%	2.02	2.02	2.02
172.16.5.40:20151	tikv	221%	221%	221%	617%	617%	617%	1.79	1.79	1.79
172.16.5.40:20379	pd	82%	82%	82%	78%	78%	78%	-0.05	-0.05	-0.05

图 258: Compare Instance CPU Usage 报表

- t1.AVG、t1.MAX、t1.Min 分别是 t1 时间段内 CPU 使用率的平均值、最大值、最小值。
- t2.AVG、t2.MAX、t2.Min 分别是 t2 时间段内 CPU 使用率的平均值、最大值、最小值。
- AVG_DIFF_RATIO 表示 t1 和 t2 时间段平均值的 DIFF_RATIO。
- MAX_DIFF_RATIO 表示 t1 和 t2 时间段最大值的 DIFF_RATIO。
- MIN_DIFF_RATIO 表示 t1 和 t2 时间段最小值的 DIFF_RATIO。

DIFF_RATIO 表示两个时间段的差异大小，有以下几个取值方式：

- 如果该监控仅在 t2 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 1。
- 如果监控项仅在 t1 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 -1。
- 如果 t2 时间段的值比 t1 时间段的值大，则 DIFF_RATIO = (t2.value / t1.value)- 1。
- 如果 t2 时间段的值比 t1 时间段的值小，则 DIFF_RATIO = 1 - (t1.value / t2.value)。

例如上表中，tidb 节点的平均 CPU 使用率在 t2 时间段比 t1 时间段高 2.02 倍，2.02 = 1240/410 - 1。

Maximum Different Item 报表介绍

Maximum Different Item 的报表是对比两个时间段的监控项后，按照监控项的差异大小排序，通过这个表可以很快发现两个时间段哪些监控的差异最大。示例如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	28916.85	61.33	1773532	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	7219.00		7219	TOTAL_COUNT
overview, total_time_consume	tidb_parse	general	1201.00		1201	TOTAL_COUNT
TiDB, tidb_time_consume	tidb_slow_query_cop_wait	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	tidb_slow_query_cop_process	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	Slow query	172.16.5.40:10089	799.00	1	800	TOTAL_COUNT
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	534.43	15.822 MB	8.273 GB	TOTAL_VALUE
overview, total_time_consume	tidb_distsql_execution	dag	470.06	0.68	320.32	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	412.29	0.07%	28.93%	MIN
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	198.39	1067055.38	212755381.33	TOTAL_VALUE
overview, total_time_consume Expand	tidb_ddl_worker	drop view	139.00		139	TOTAL_COUNT
overview, total_error Expand	tikv_storage_async_request_error	snapshot	89.00		89	TOTAL_COUNT
overview, total_time_consume Expand	Coprocessor handling request	index	78.99	0.67	53.59	TOTAL_TIME
overview, total_time_consume	tidb_cop	172.16.5.40:10089	75.11	5.93	451.31	TOTAL_TIME
overview, total_time_consume	KV request	Cop	75.10	5.93	451.26	TOTAL_TIME
overview, total_time_consume Expand	Coprocessor request	select	73.12	4.73	350.6	TOTAL_TIME
PD, pd_time_consume Expand	pd_region_heartbeat	172.16.5.40:20150,1	-73.00	73		TOTAL_COUNT
overview, total_time_consume Expand	tikv_grpc_messge	coprocessor	71.75	5.56	404.49	TOTAL_TIME
TiDB, tidb_time_consume	tidb_ddl_update_self_version	172.16.5.40:10089,ok	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	TiDB meta operation	update_ddl_job	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	tidb_owner_handle_syncer	update_global_version	71.00		71	TOTAL_COUNT
TiDB, tidb_time_consume Expand	tidb_query	172.16.5.40:10089,internal	-62.84	15.96	0.25	P999
TiKV, scheduler_info Expand	Scheduler stage	172.16.5.40:20151,scan_lock,snapshot_ok	55.45		55.45	TOTAL_VALUE
overview, total_time_consume Expand	tikv_grpc_messge	kv_scan_lock	55.00		55	TOTAL_COUNT
TiKV, tikv_time_consume Expand	tikv_scheduler_command	172.16.5.40:20151,scan_lock	55.00		55	TOTAL_COUNT
overview, total_time_consume Expand	PD client cmd	get_region	44.00		44	TOTAL_COUNT
overview, total_time_consume Expand	tikv_cop_wait	select	39.40	171	6908	TOTAL_COUNT
TiDB, tidb_time_consume	Schema load	172.16.5.40:10089	37.50	0.02	0.77	TOTAL_TIME
overview, total_time_consume	Schema load	172.16.5.40:10089	30.25	0.008	0.25	P999

图 259: 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。

11.9.11.3 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。

11.9.11.3.1 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。

大查询/写入导致 QPS 抖动或延迟上升诊断

示例 1

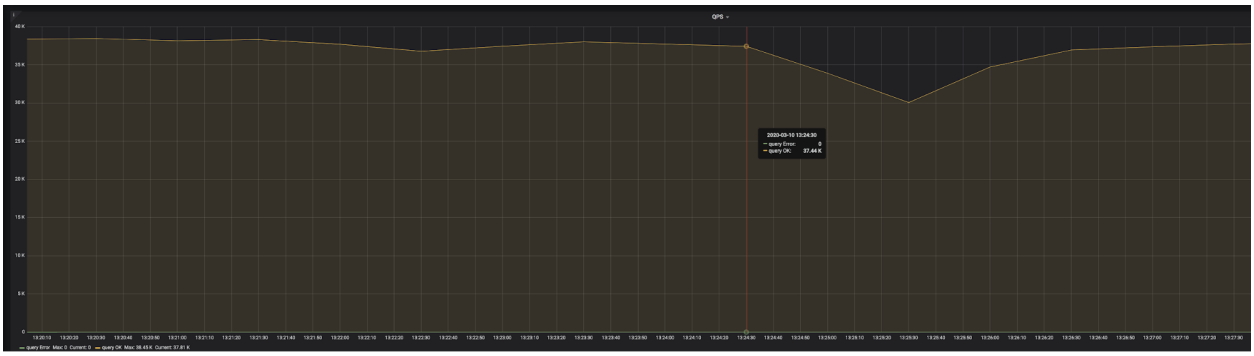


图 260: QPS 图

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 261: 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 262: 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 263: 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。

11.9.11.3.2 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。

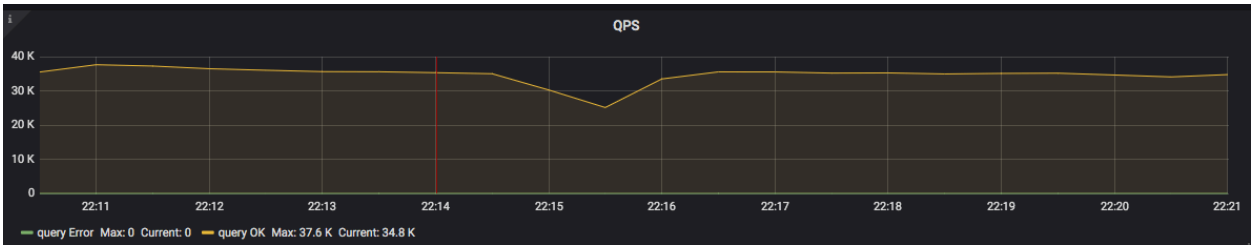


图 264: QPS 图

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 265: 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。

11.9.12 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。

11.9.12.1 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：

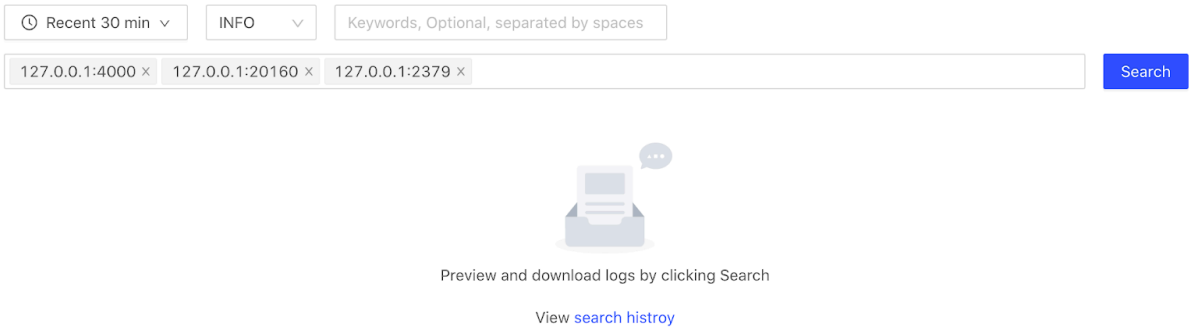


图 266: 日志搜索首页

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。

11.9.12.2 搜索结果详情

搜索结果详情页面如下图所示：



图 267: 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。

11.9.12.2.1 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

11.9.12.3 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：

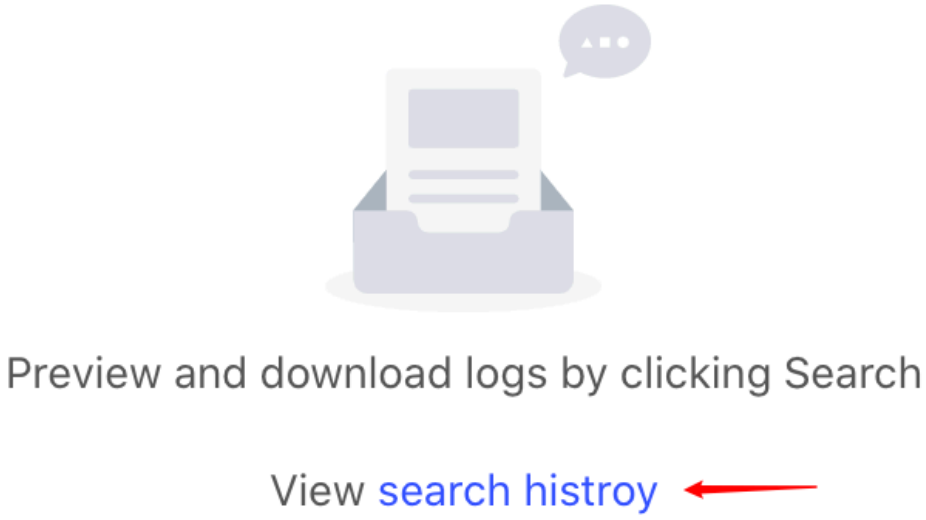


图 268: 搜索历史入口

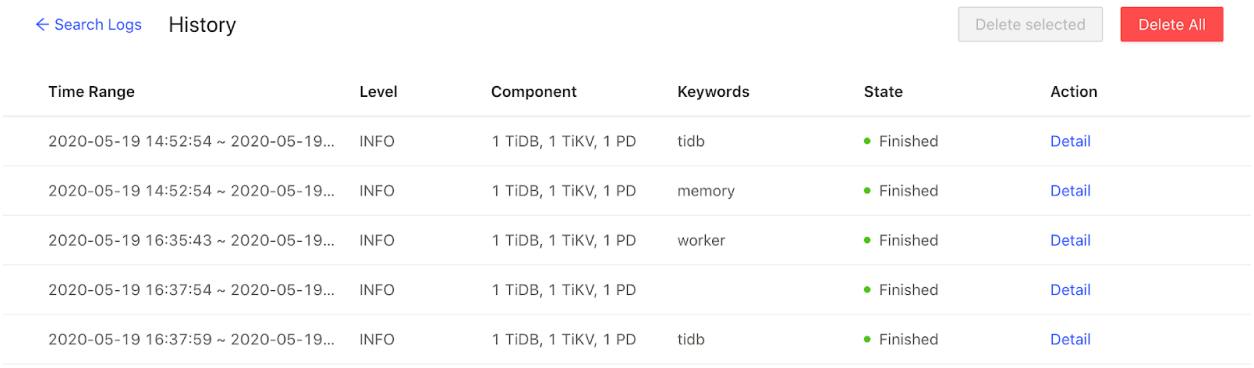


图 269: 搜索历史列表

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：

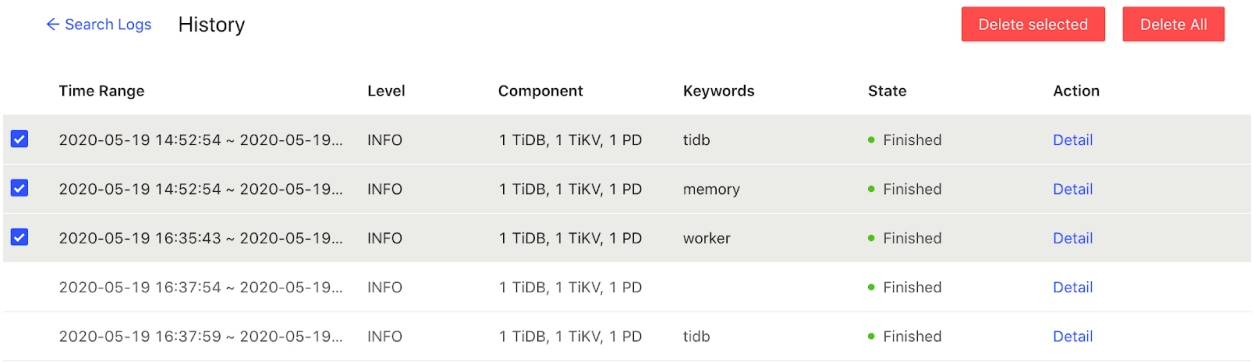


图 270: 删除搜索历史

11.9.13 实例性能分析

11.9.13.1 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。

11.9.13.1.1 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

11.9.13.1.2 访问页面

可以通过以下两种方法访问实例性能分析页面：

- 登录后，左侧导航条点击高级调试 (Advanced Debugging) > 实例性能分析 (Profile Instances) > 手动分析 (Manual Profiling)：

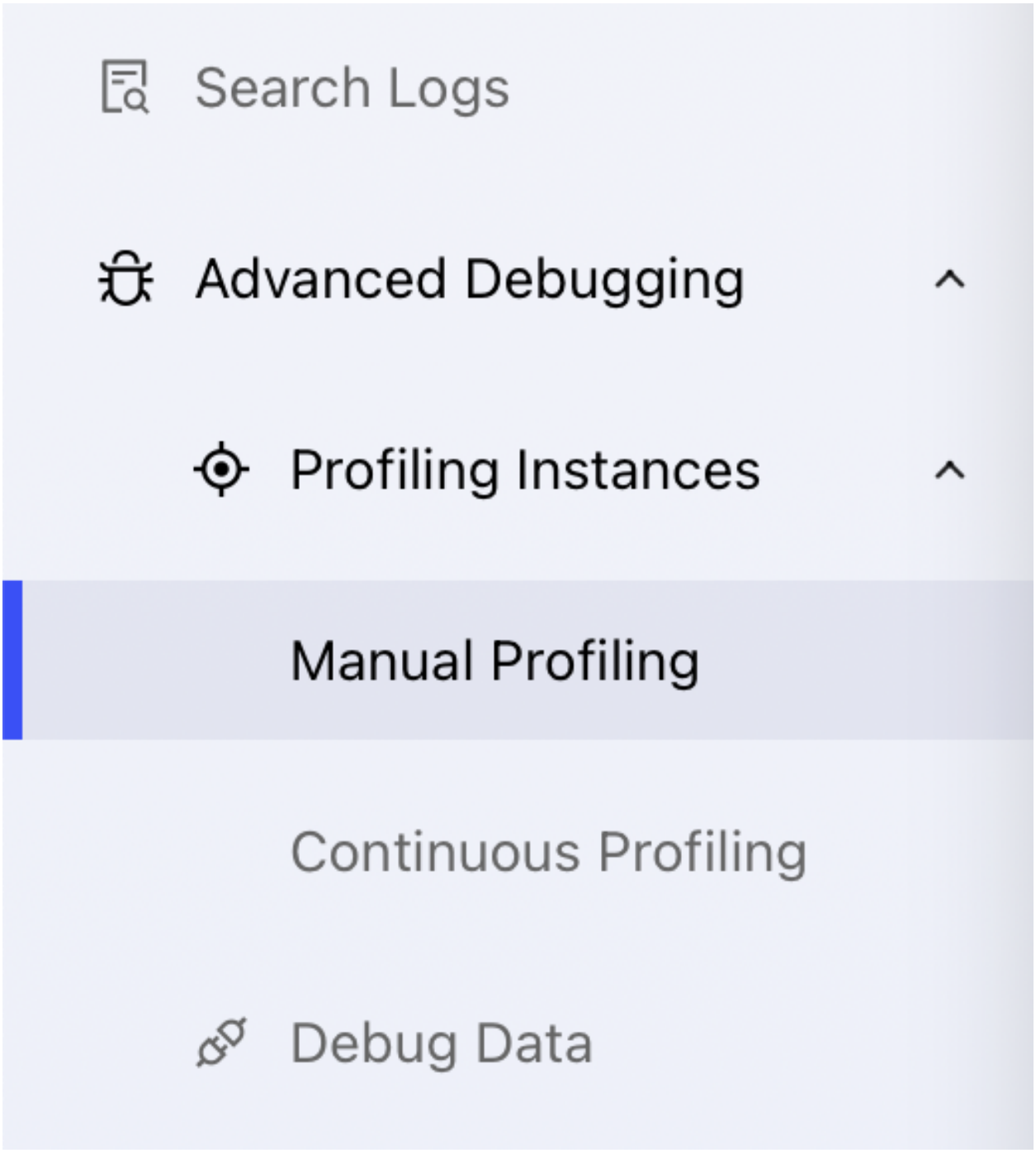


图 271: 访问页面

- 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)。

11.9.13.1.3 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)：

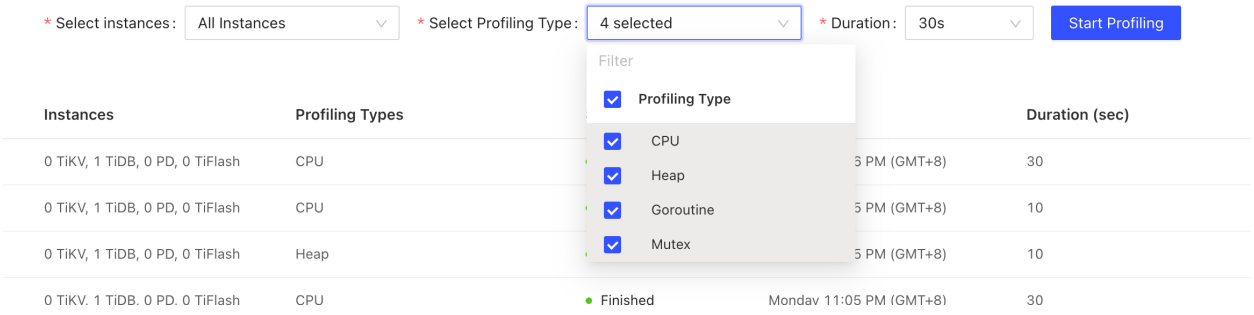


图 272: 开始分析

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。

11.9.13.1.4 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：

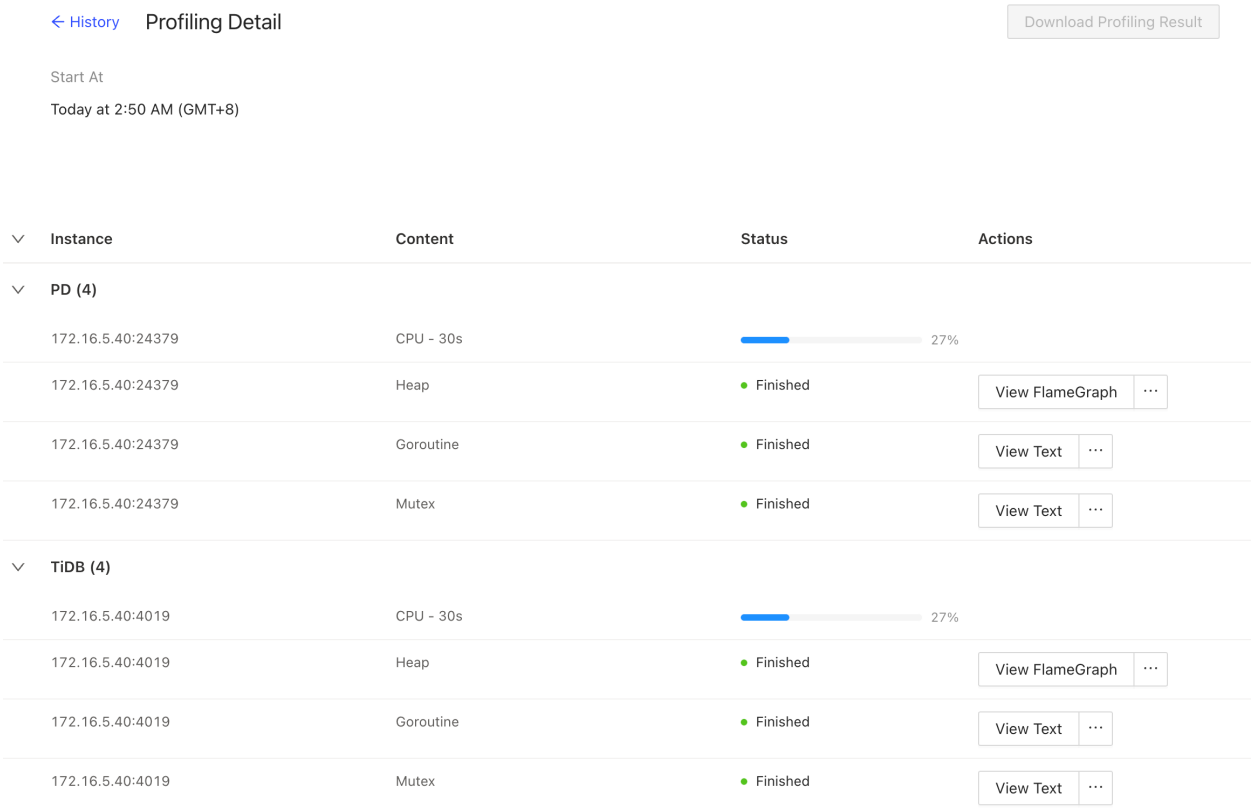


图 273: 实时状态

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。

11.9.13.1.5 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：

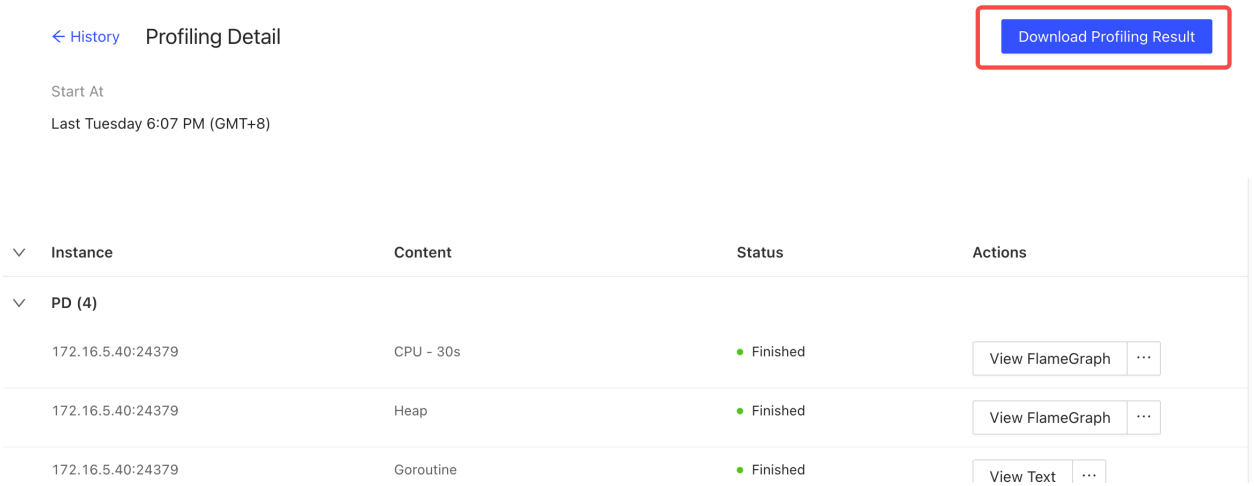


图 274: 下载分析结果

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：

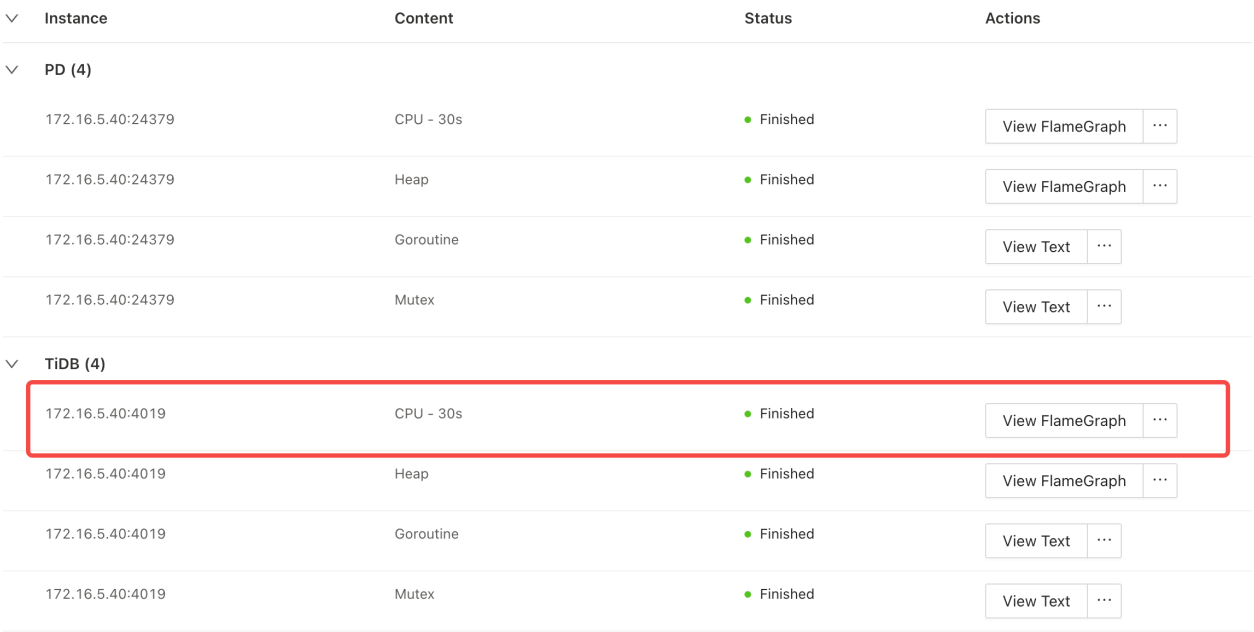


图 275: 在线查看分析结果

11.9.13.1.6 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：

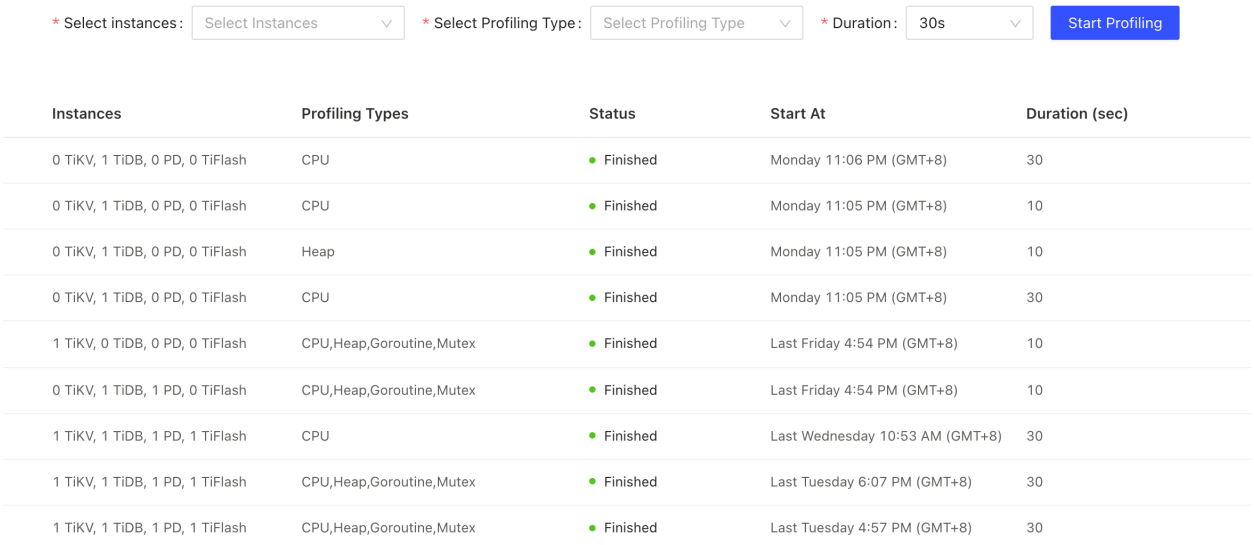


图 276: 历史列表

关于状态详情页的操作，参见查看性能分析状态章节。

11.9.13.2 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。

11.9.13.2.1 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。

11.9.13.2.2 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

11.9.13.2.3 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)。

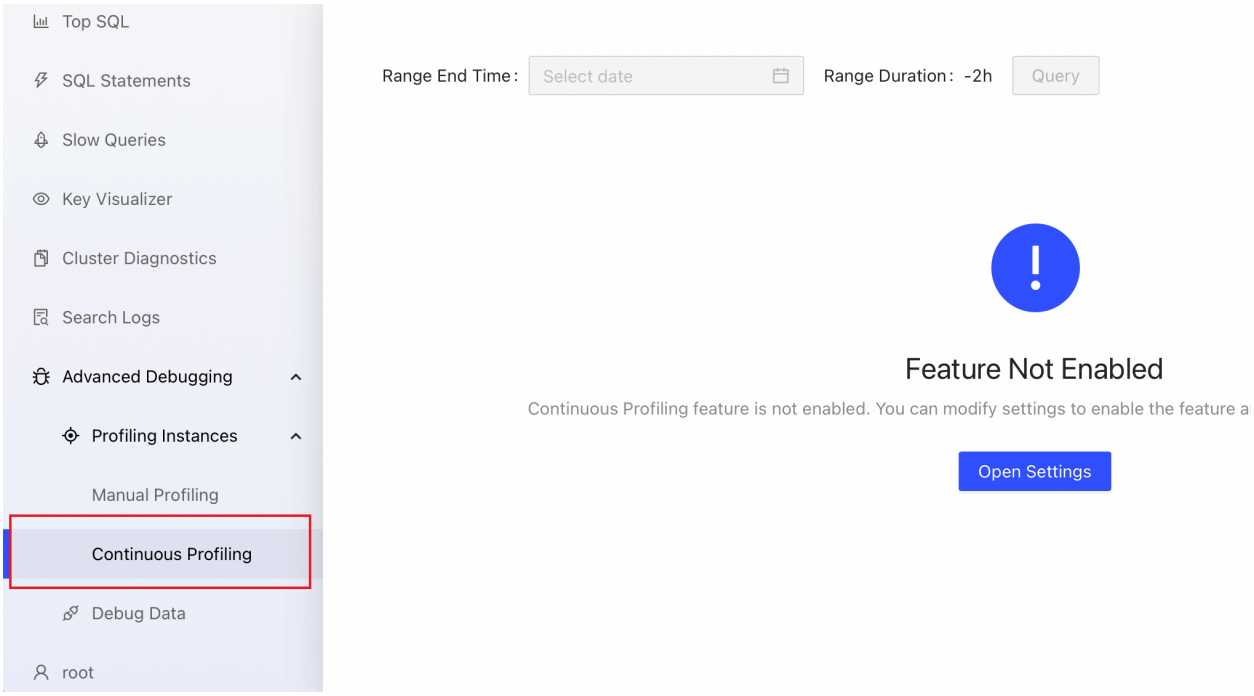


图 277: 访问页面

- 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）。

11.9.13.2.4 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。

持续性能分析默认处于关闭状态。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)。

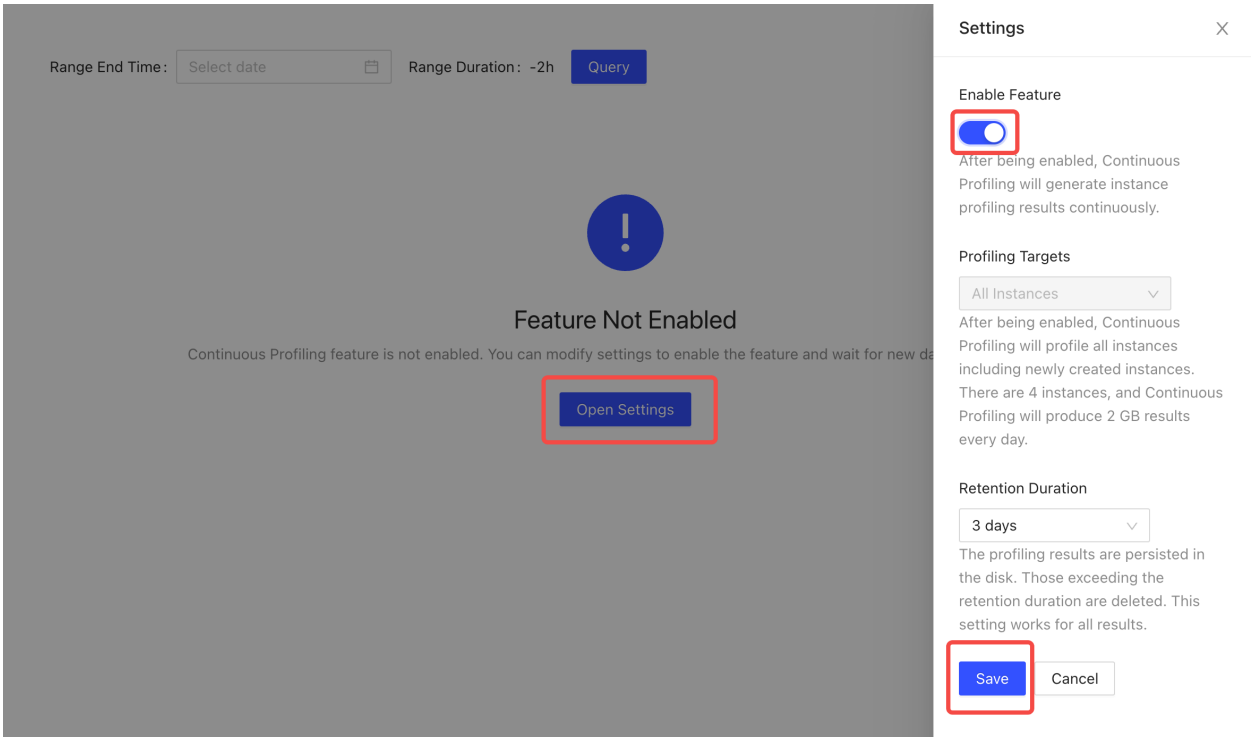


图 278: 启用功能

11.9.13.2.5 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。

11.9.13.2.6 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：

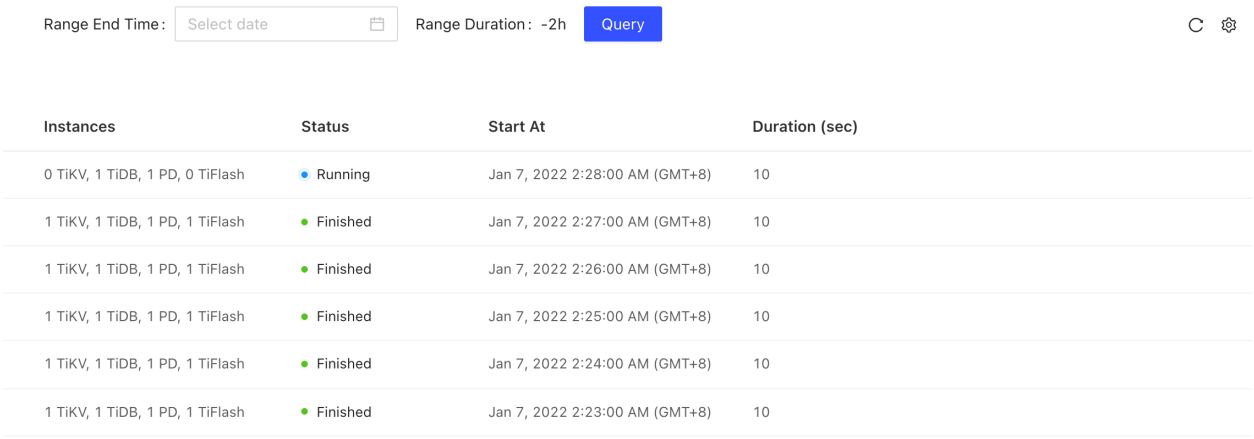


图 279: 历史结果

11.9.13.2.7 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：

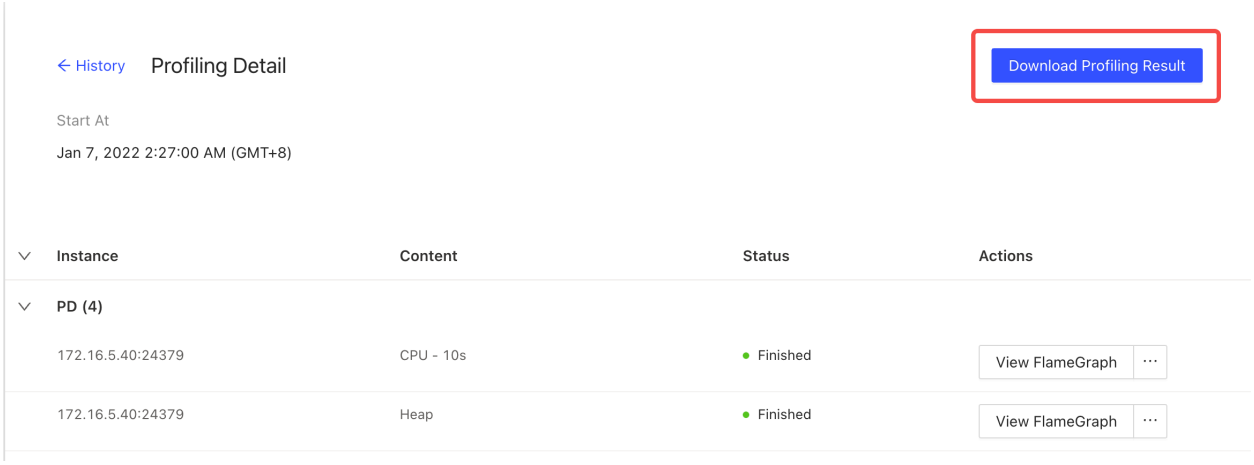


图 280: 下载某次分析结果

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：

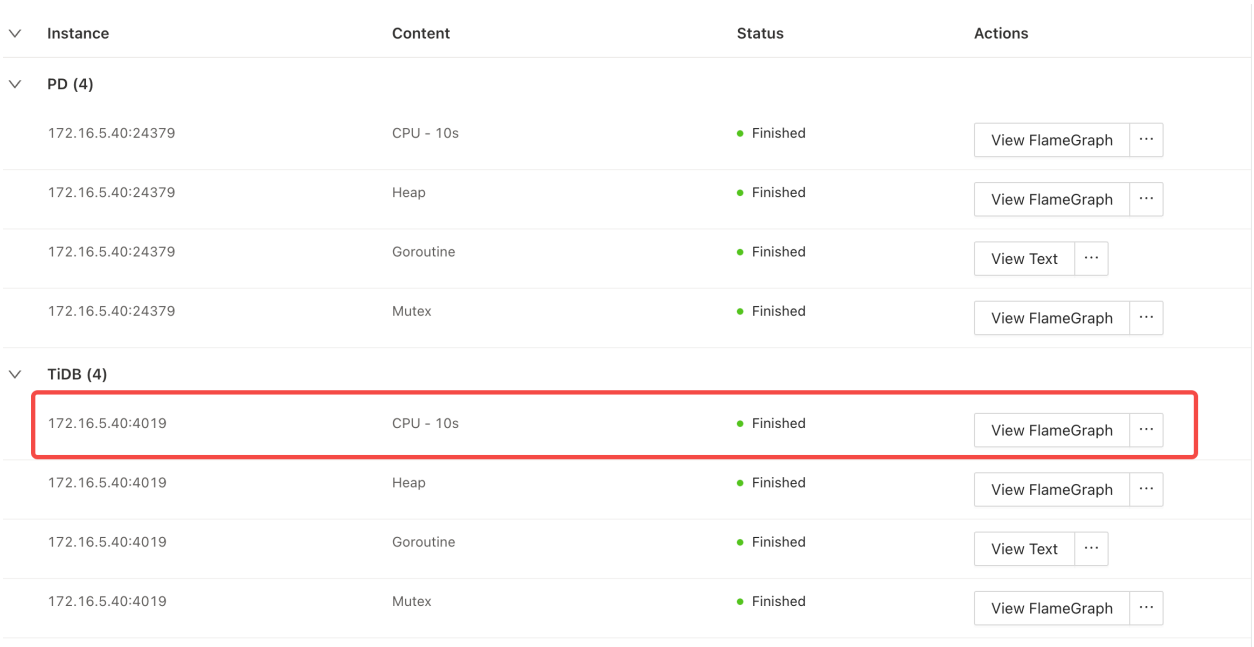


图 281: 查看单个实例分析结果

11.9.13.2.8 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)。

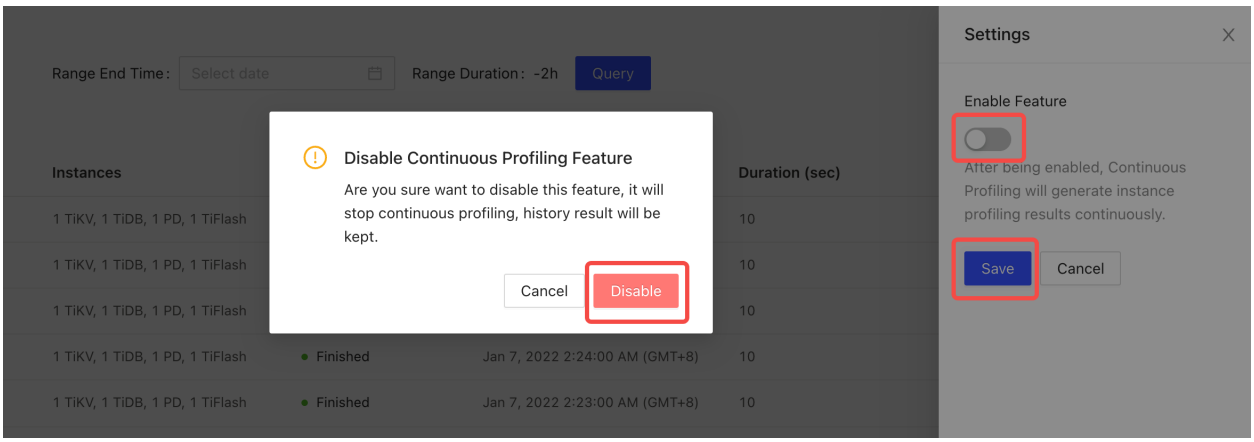


图 282: 停用功能

11.9.13.2.9 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。

11.9.14 会话管理与配置

11.9.14.1 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。

11.9.14.1.1 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)。

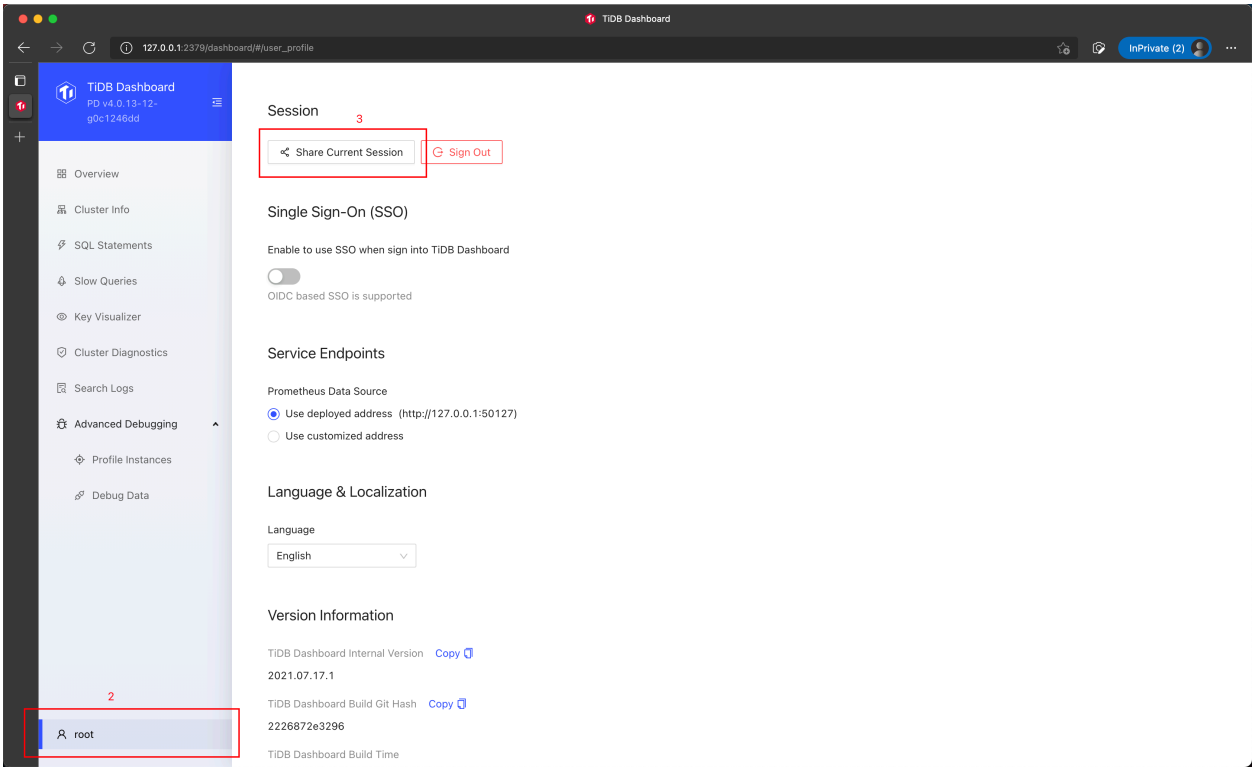


图 283: 操作示例

> **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)。

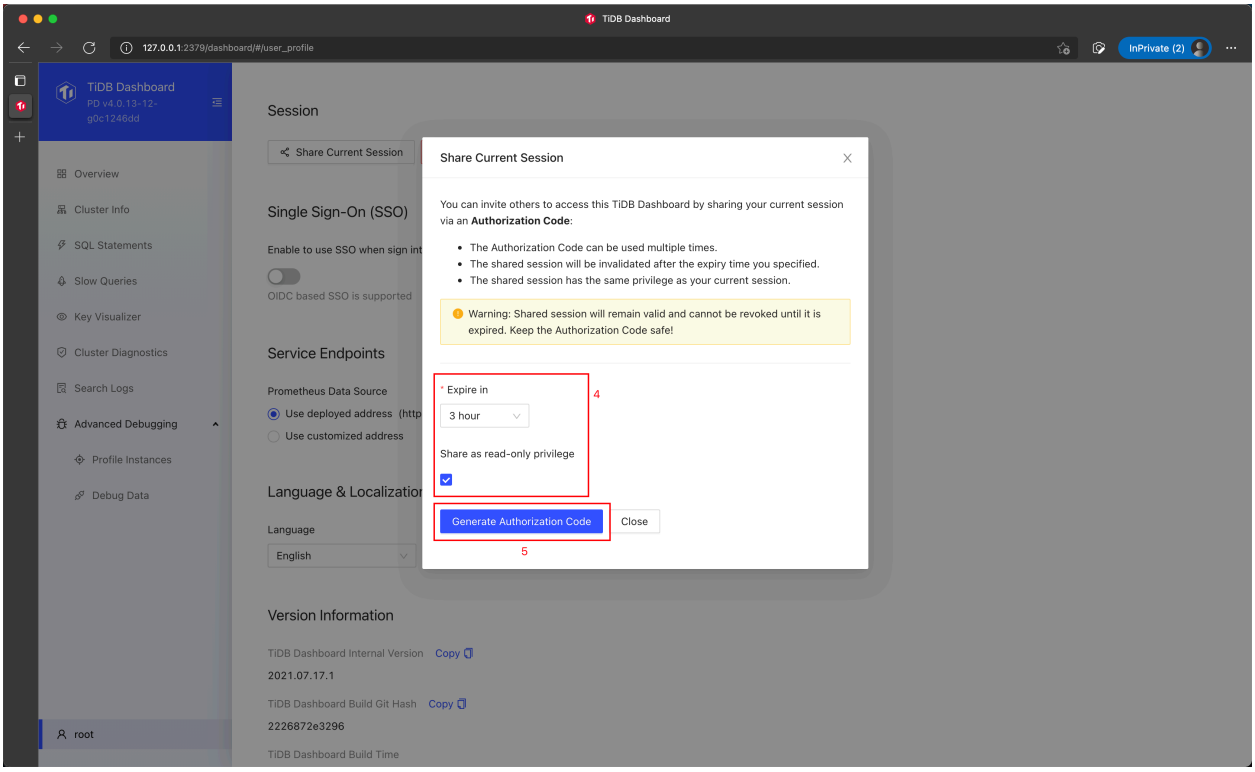


图 284: 操作示例

6. 将生成出来的授权码提供给要分享的用户。

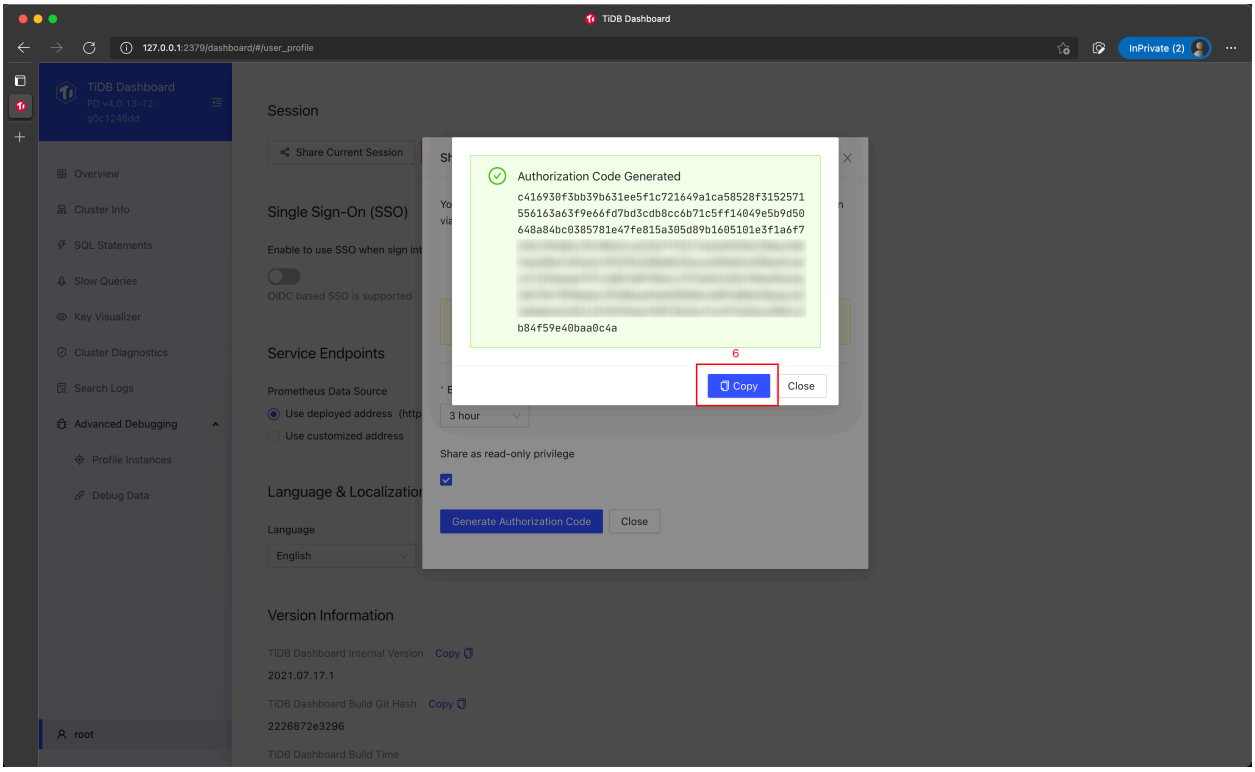


图 285: 操作示例

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。

11.9.14.1.2 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)。

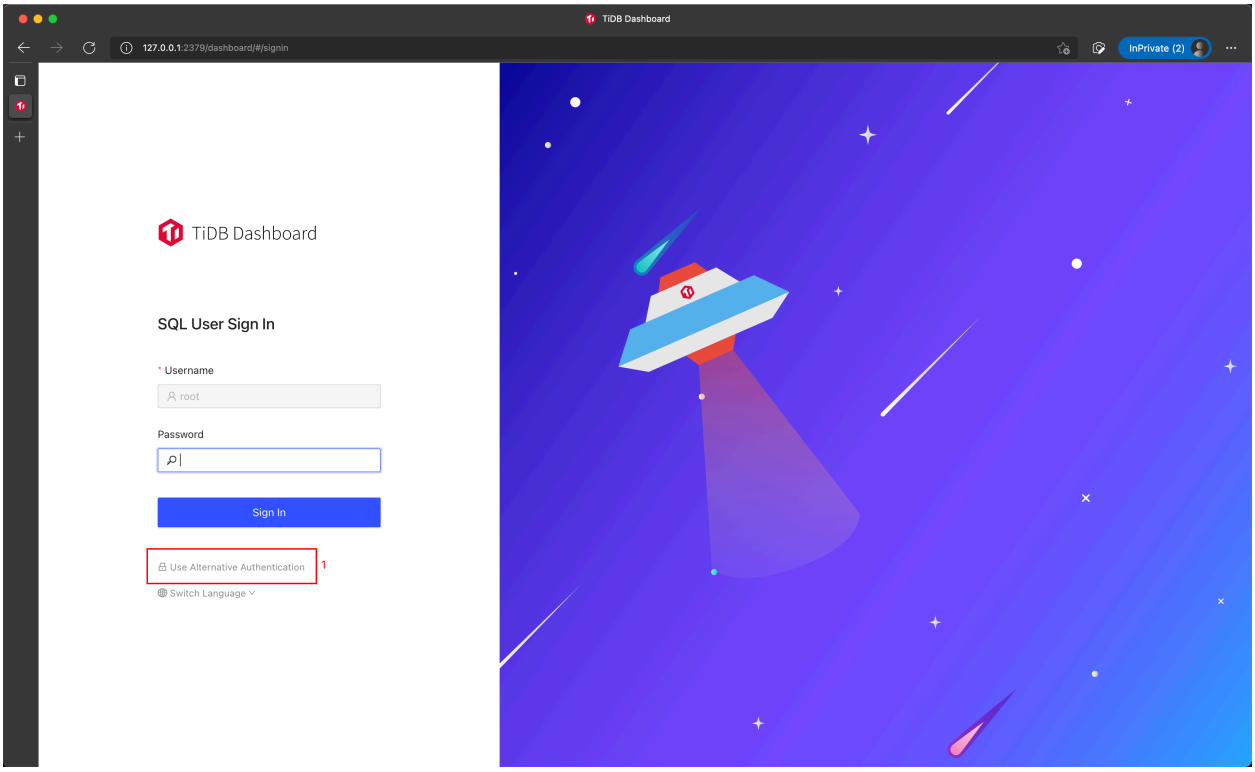


图 286: 操作示例

2. 选择使用授权码 (Authorization Code) 登录。

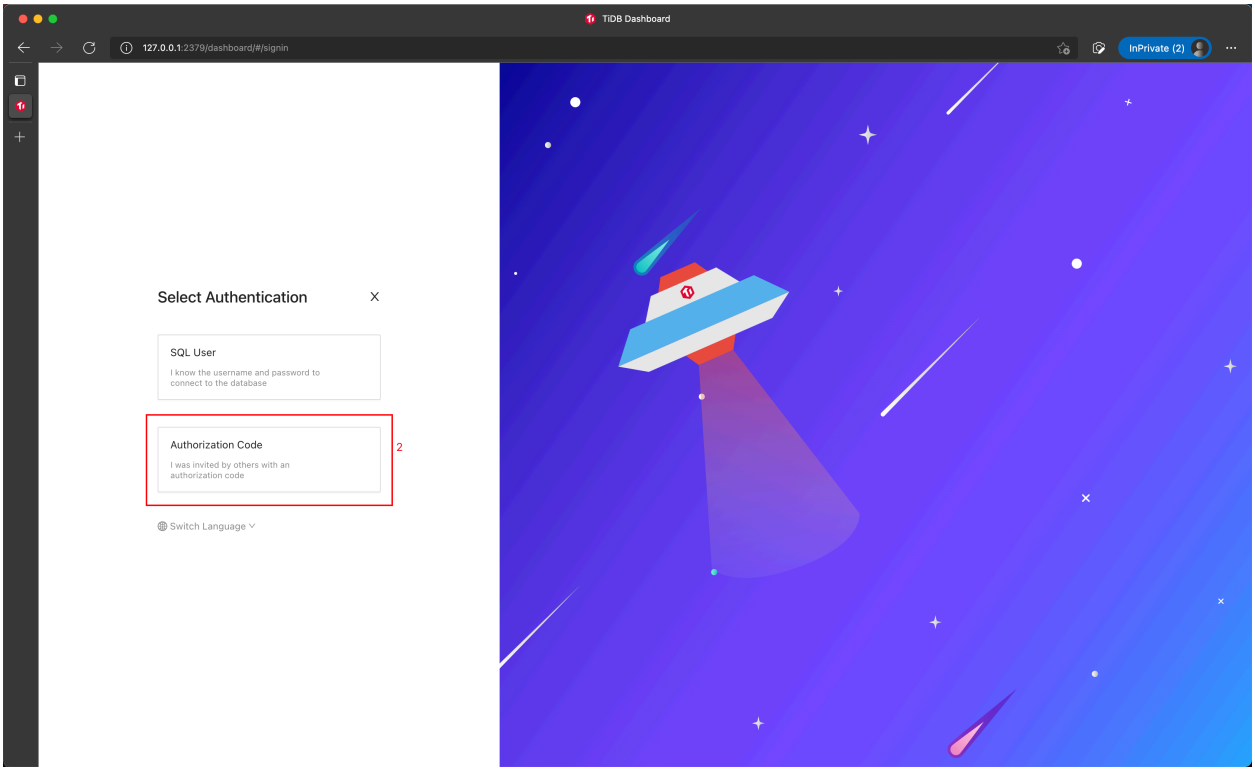


图 287: 操作示例

3. 输入从分享者取得的授权码。
4. 点击登录 (Sign In)。

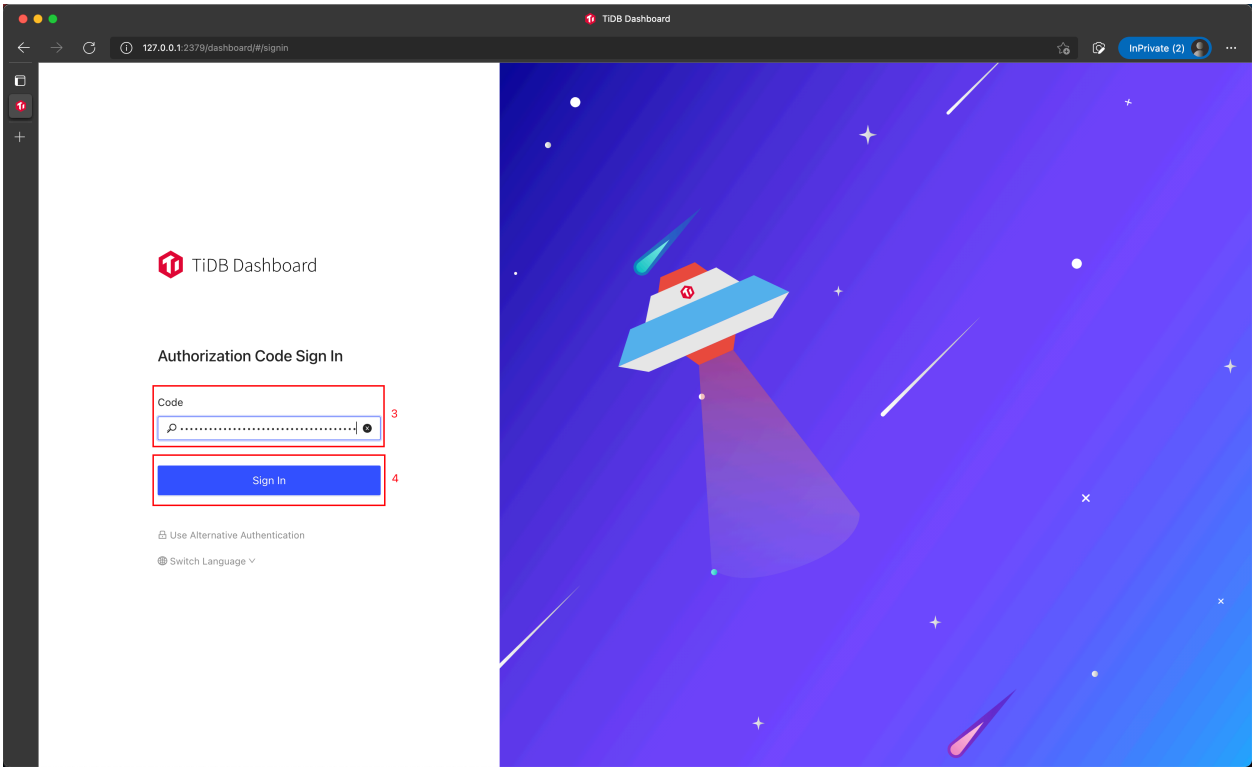


图 288: 操作示例

11.9.14.2 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。

11.9.14.2.1 配置 OIDC SSO

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。

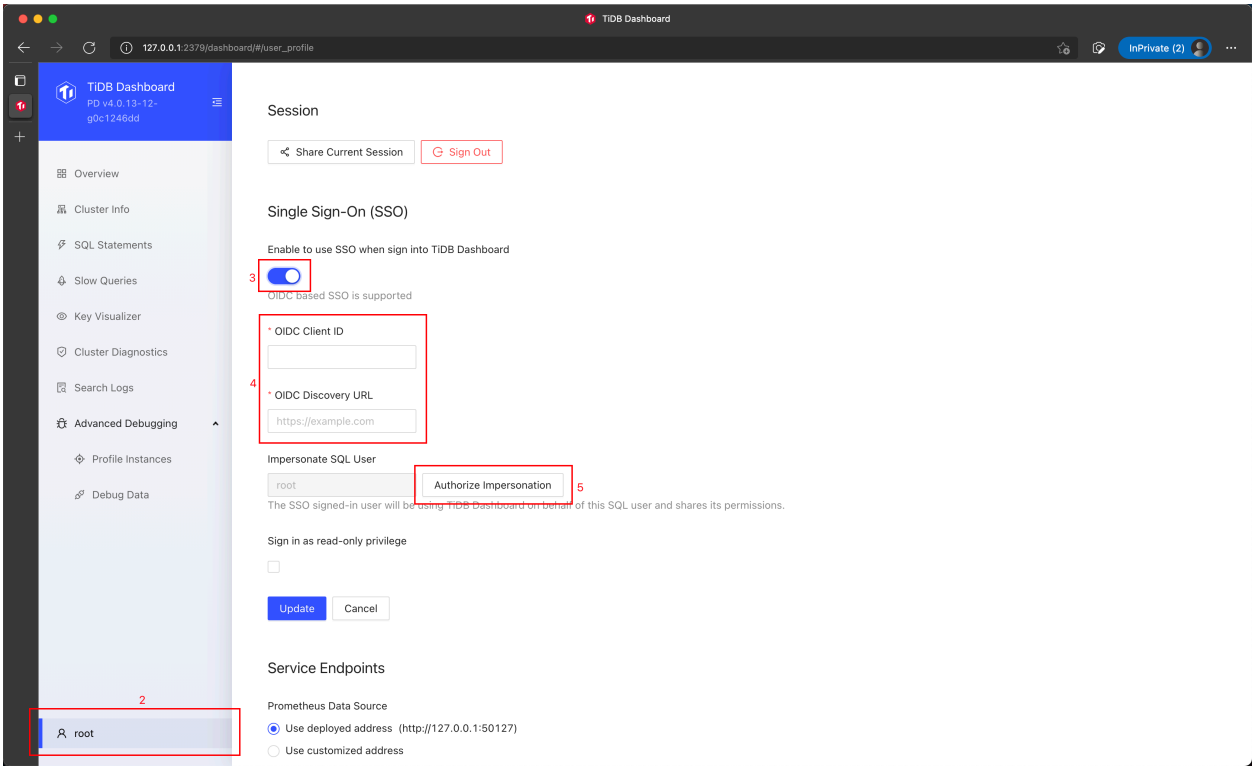


图 289: 操作示例

> **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)。

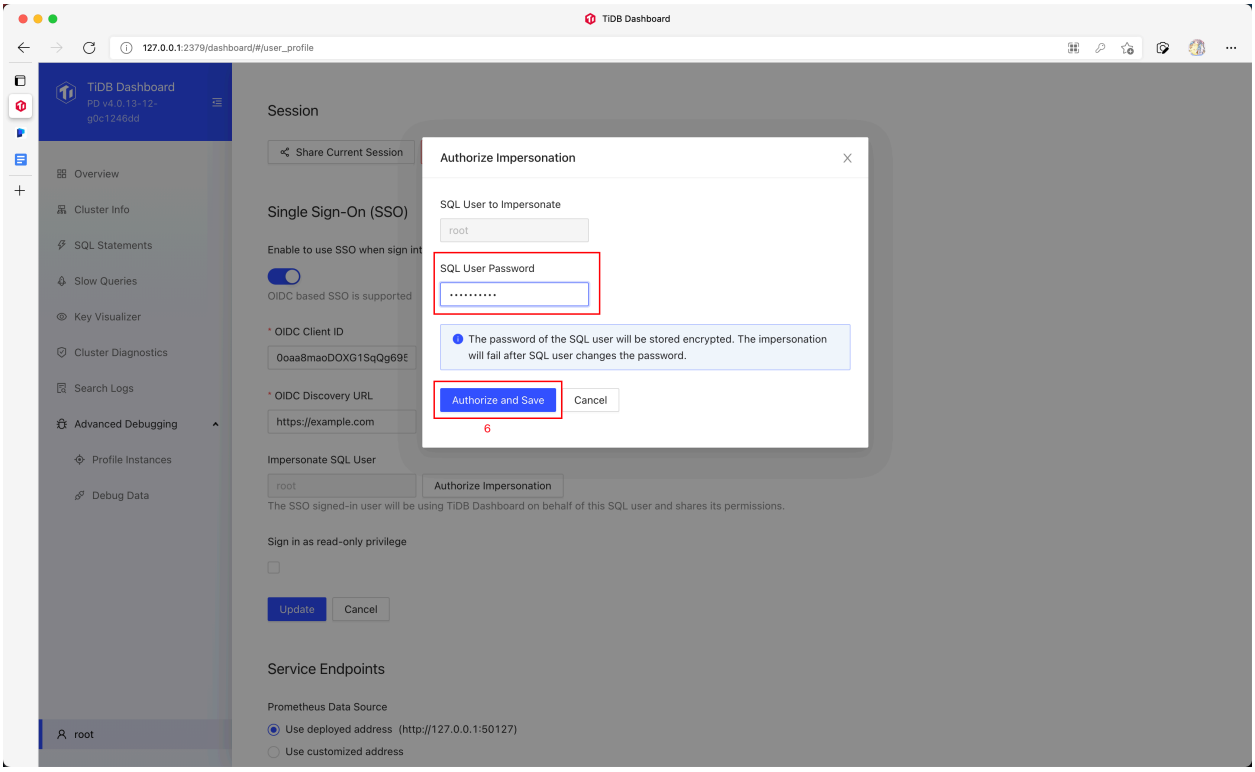


图 290: 操作示例

7. 点击更新 (Update) 保存配置。

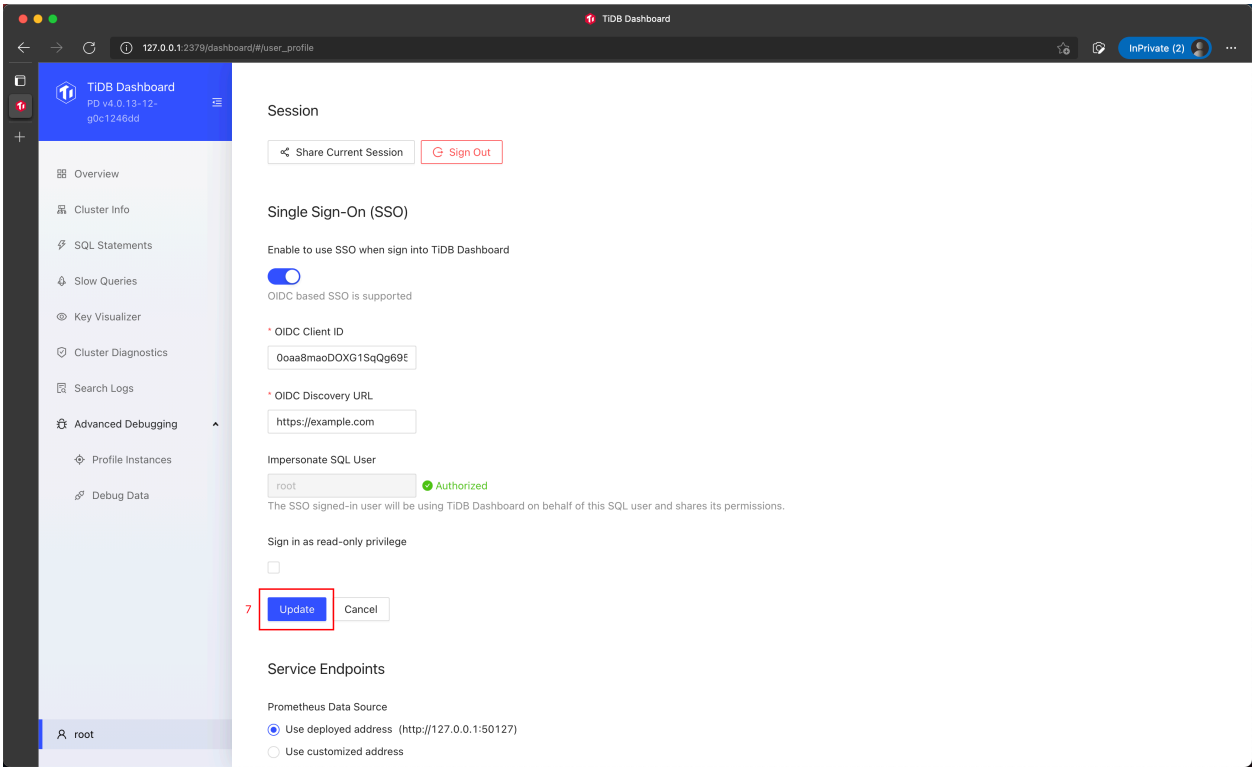


图 291: 操作示例

至此 TiDB Dashboard 中已经成功开启了 SSO 登录。

> **注意：**
>
> 出于安全原因，部分 SSO 服务还需要你进一步在 SSO 服务中配置受信任的登录和登出跳转地址，请参见 SSO 服务的具体帮助完成配置。

禁用 SSO

你可以随时禁用 SSO。禁用后，之前已录入并存储在本地的替代登录 SQL 密码将被彻底清除。禁用步骤如下：

1. 登录 TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，关闭允许使用 SSO 登录到 TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 点击更新 (Update) 保存配置。

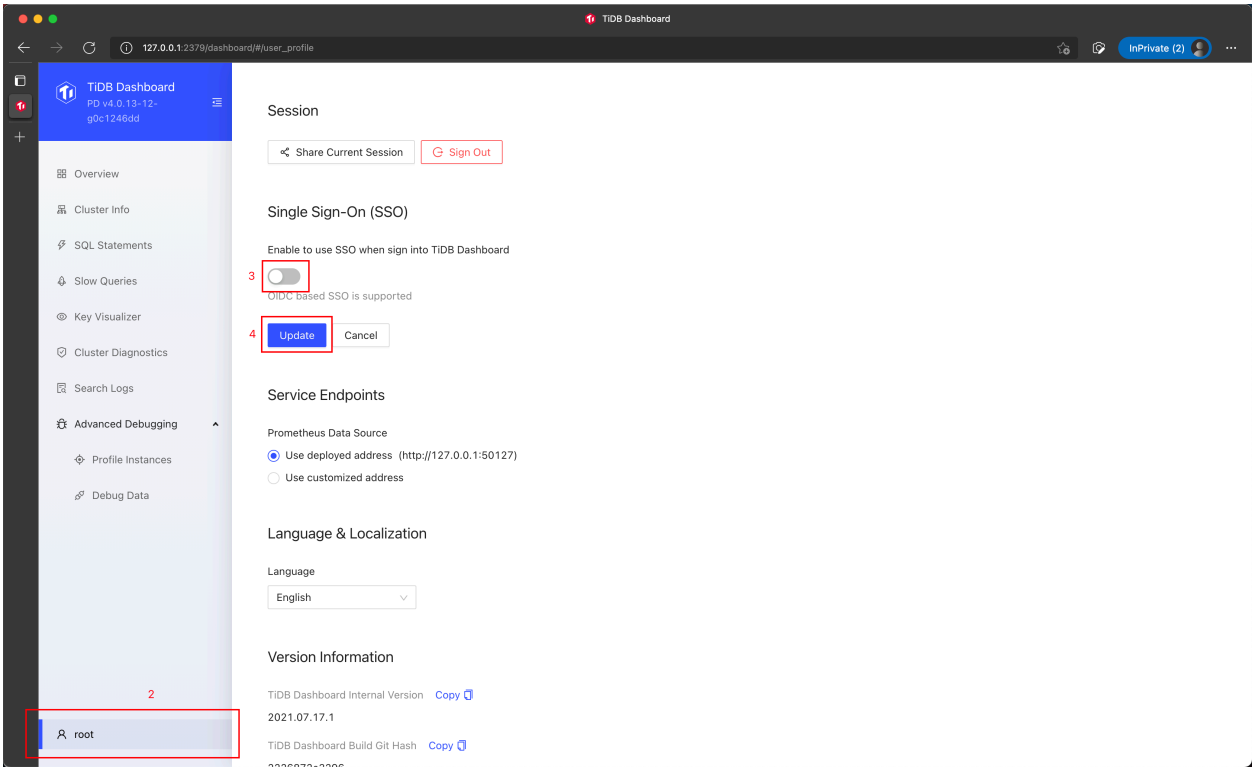


图 292: 操作示例

密码发生变更后重新录入密码

若替代登录的 SQL 用户密码发生了变更，则 SSO 登录将会失败。此时，你可以将新的登录密码录入到 TiDB Dashboard 中以恢复正常 SSO 登录功能，步骤如下：

1. 登录 TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，点击授权登录为该用户 (Authorize Impersonation) 来录入新的密码。

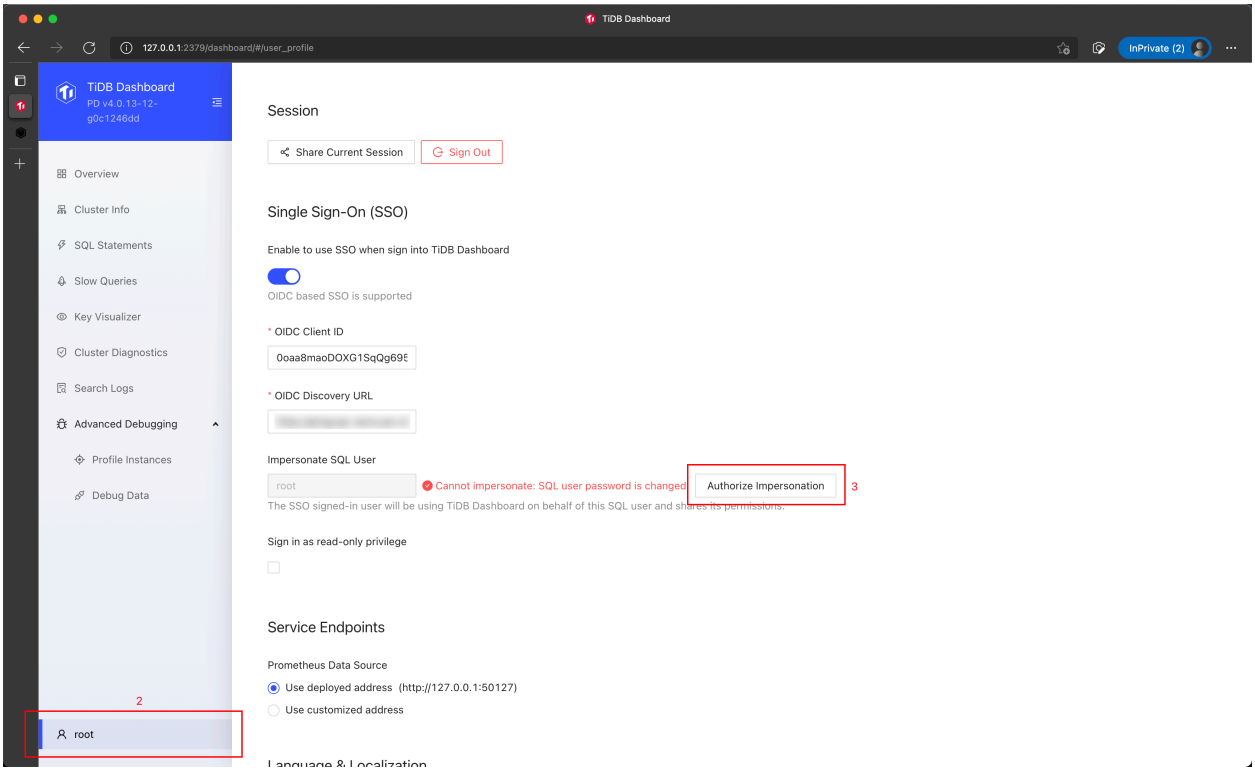


图 293: 操作示例

4. 在对话框中填写完毕密码后，点击授权并保存 (Authorize and Save)。

11.9.14.2.2 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)。

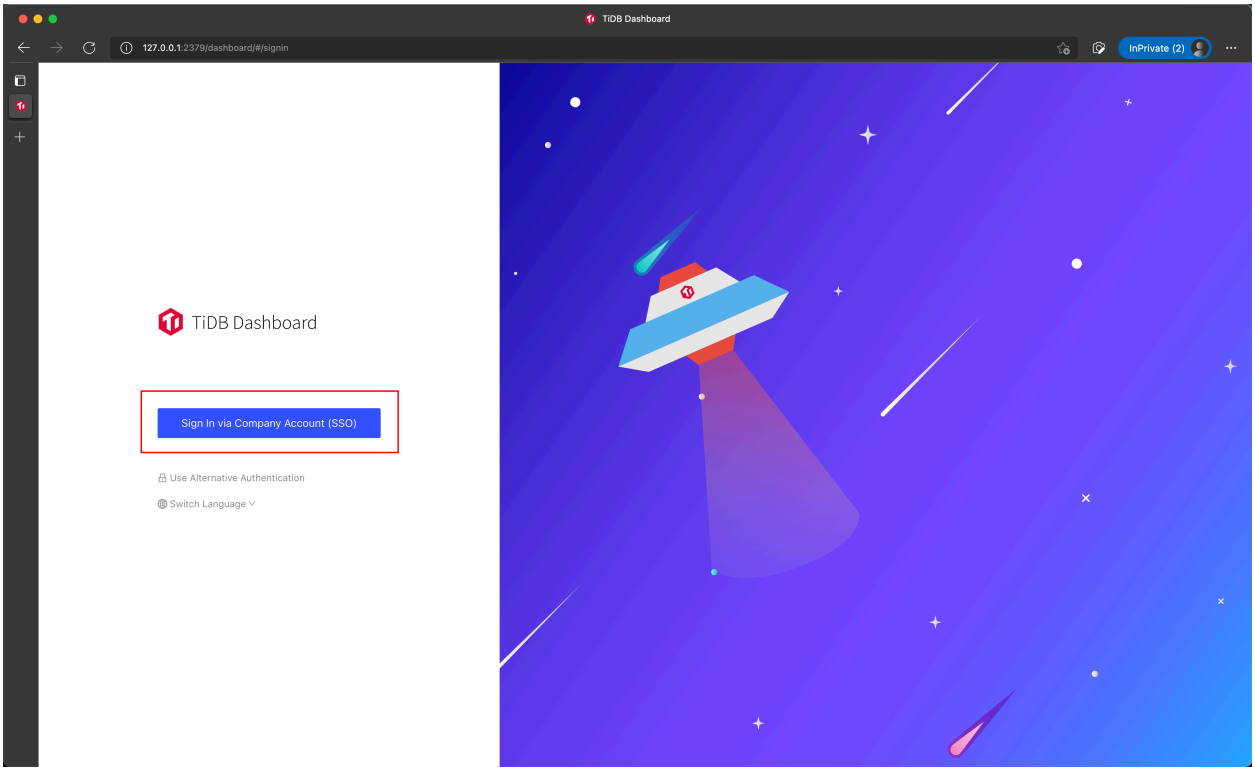


图 294: 操作示例

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。

11.9.14.2.3 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。

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。

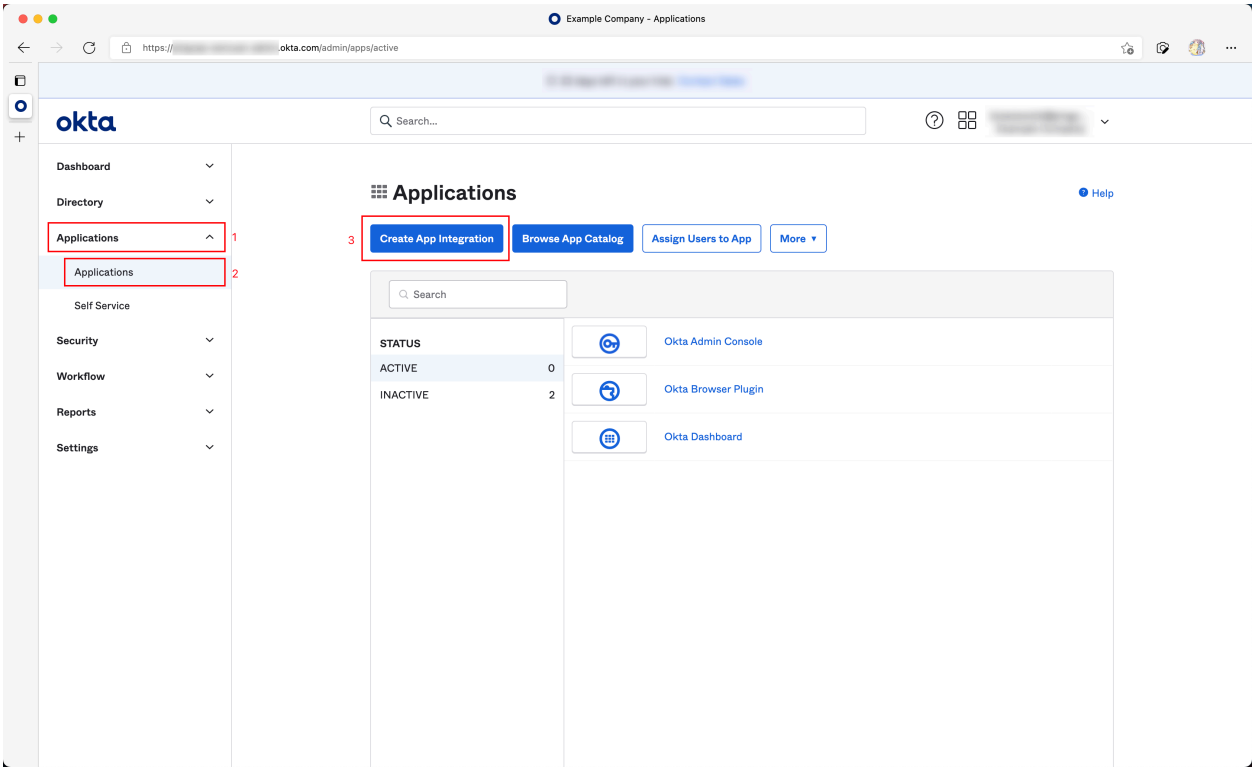


图 295: 操作示例

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。

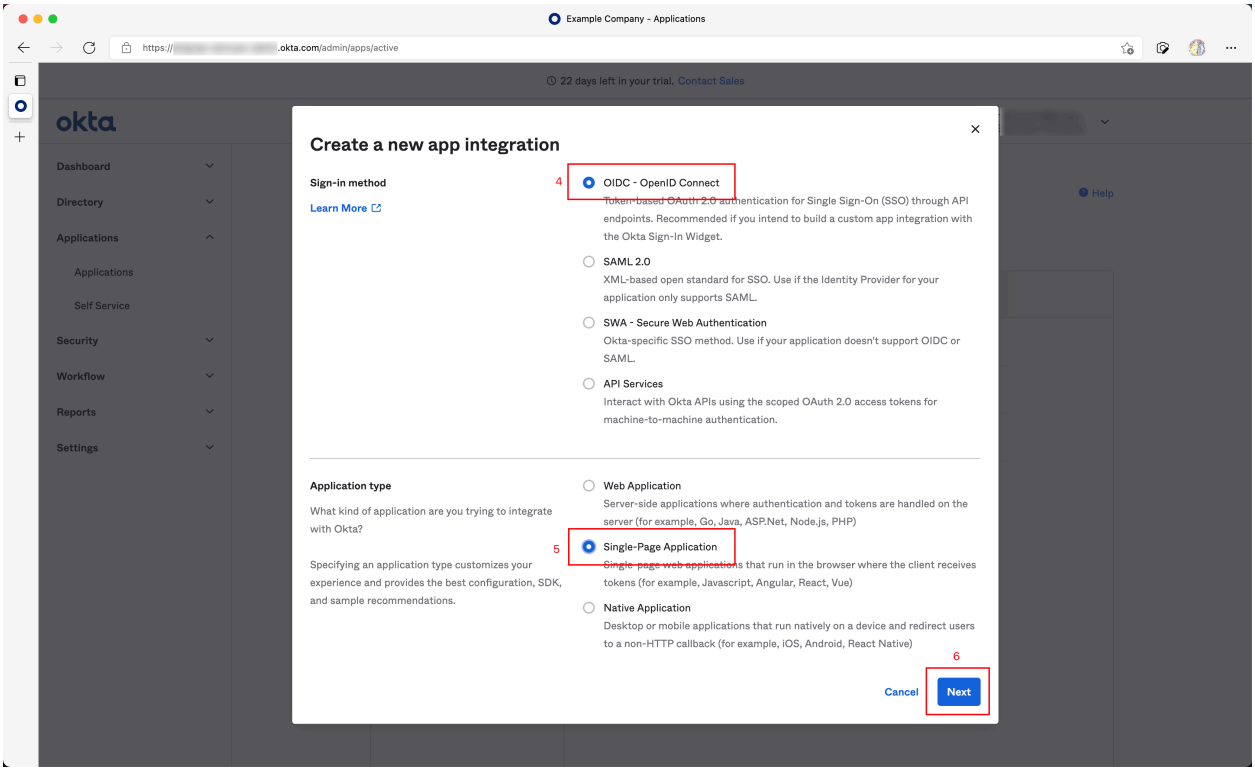


图 296: 操作示例

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。

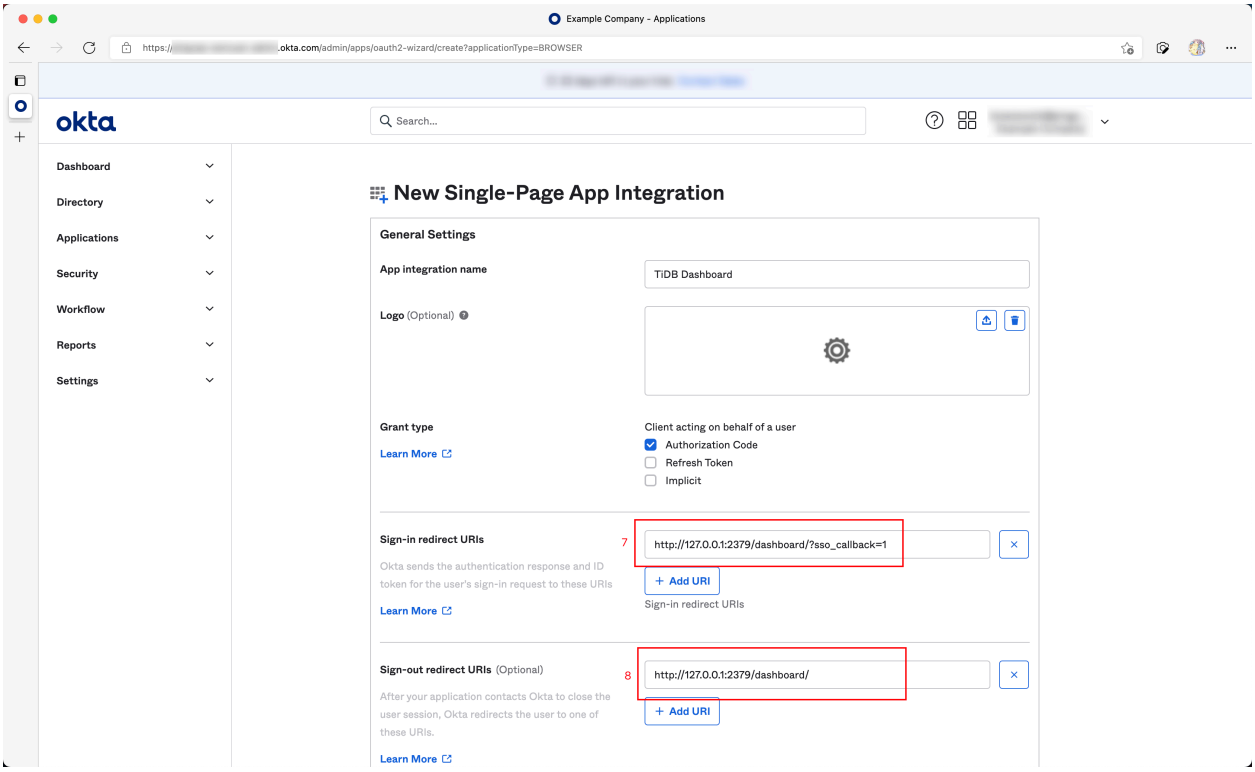


图 297: 操作示例

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。

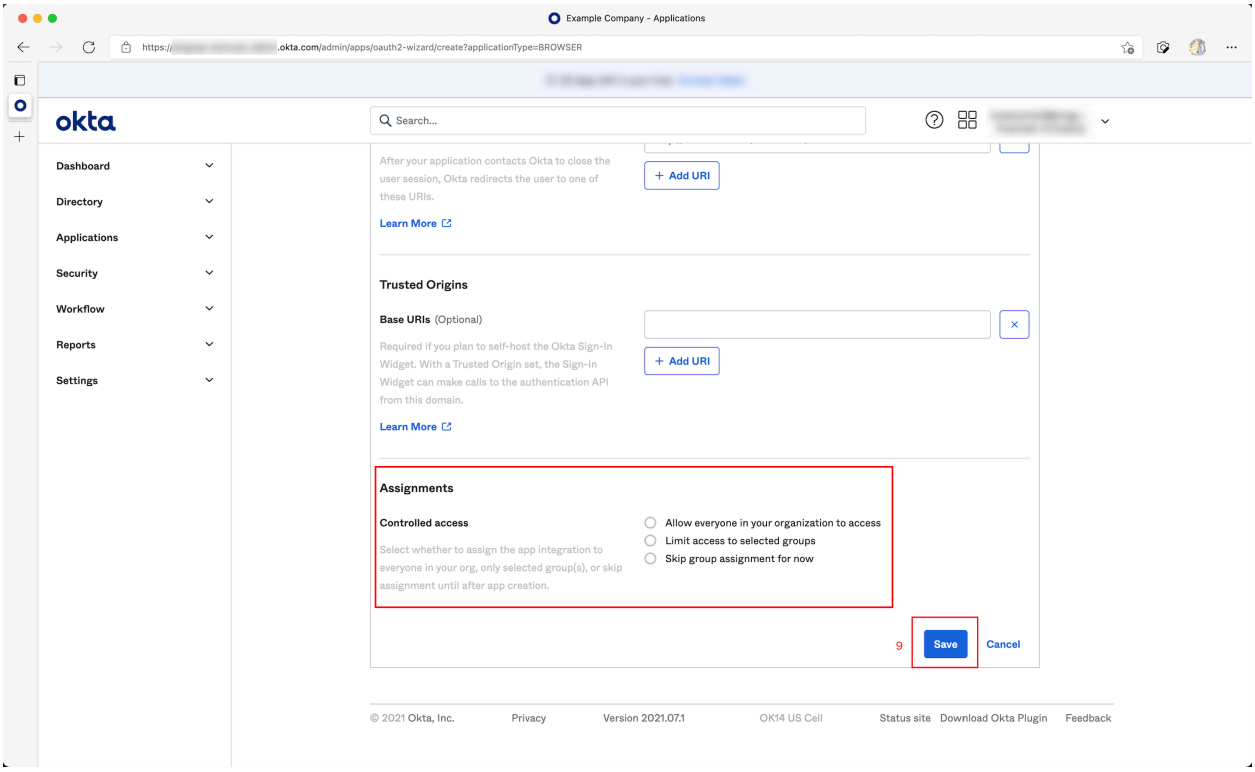


图 298: 操作示例

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 在 Okta 创建的 App Integration 中，点击 Sign On。

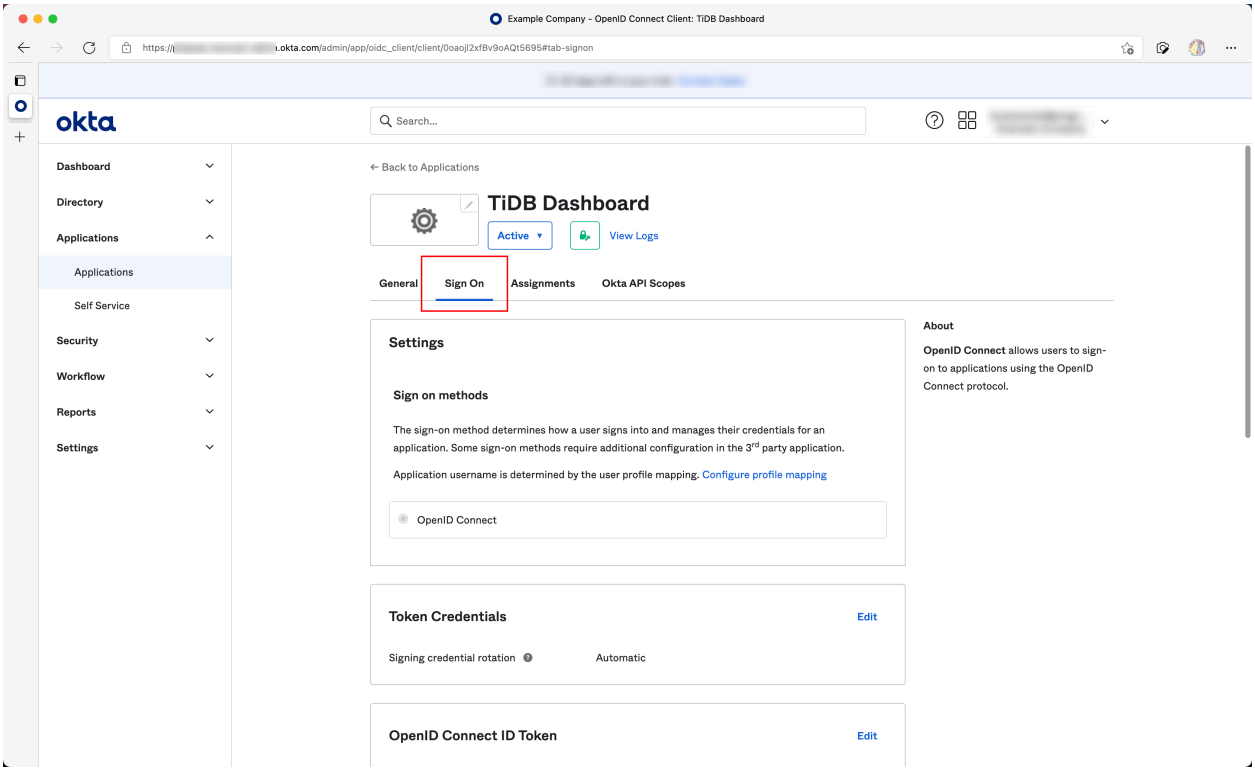


图 299: 操作示例 1

2. OpenID Connect ID Token 区域中有 Issuer 和 Audience 字段，复制这两个字段的值。

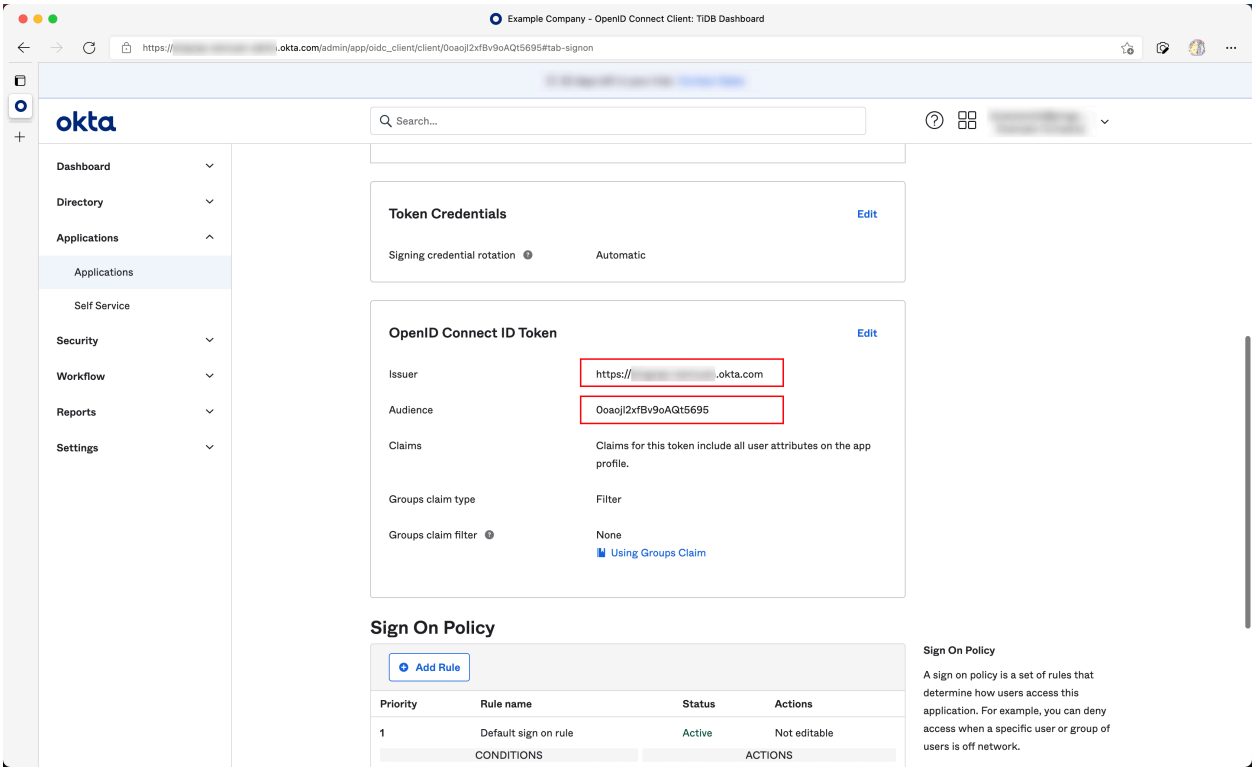


图 300: 操作示例 2

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：

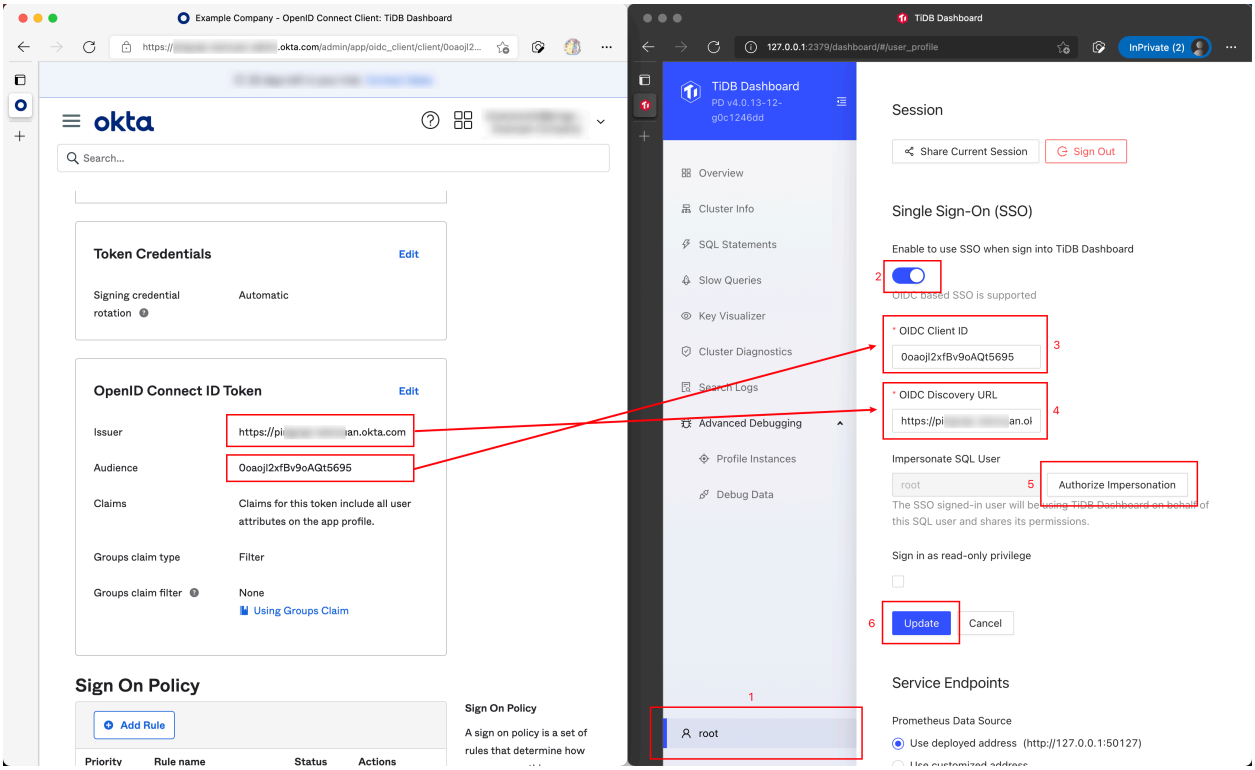


图 301: 操作示例 3

至此，TiDB Dashboard 已被配置为使用 Okta 进行 SSO 登录。

11.9.14.2.4 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。

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”。

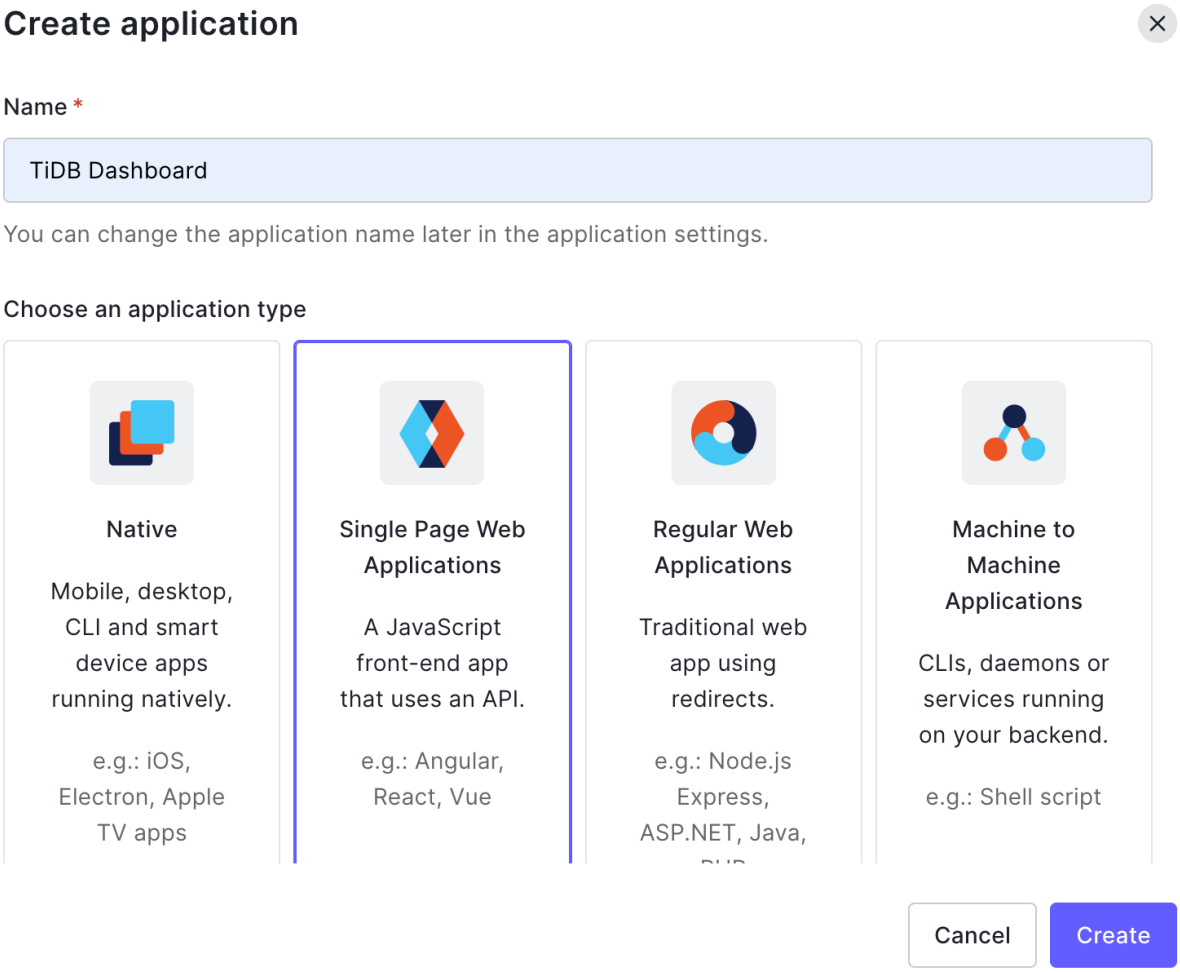


图 302: Create Application

4. 点击 Settings 栏。

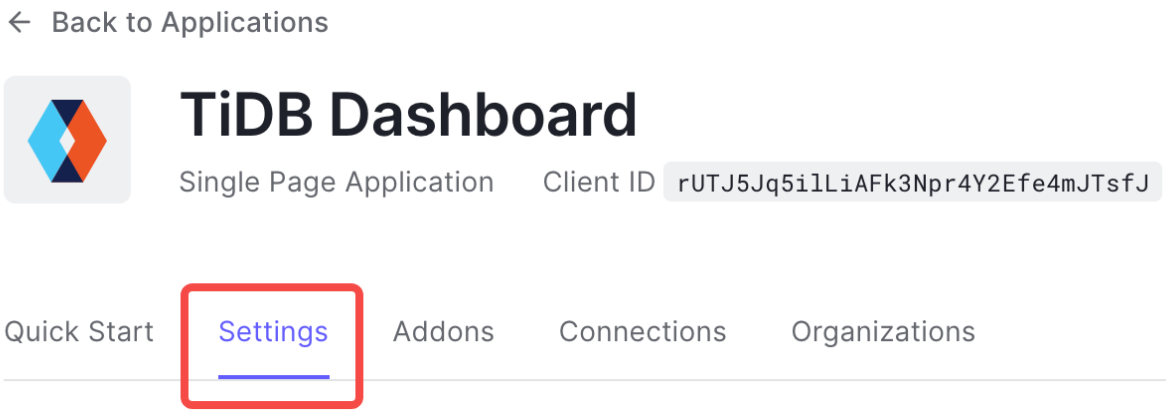


图 303: Settings

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 304: Settings

7. 其它设置保持默认，点击 Save Changes 保存。

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 将 Auth0 Settings 栏 Basic Information 项的 Client ID 字段的值填入 TiDB Dashboard 的 OIDC Client ID，将 Domain 字段的值，加上 `https://` 前缀和 `/` 后缀后填入 OIDC Discovery URL 中，例如 `https://example.us.auth0.com/`。完成授权并保存配置即可。

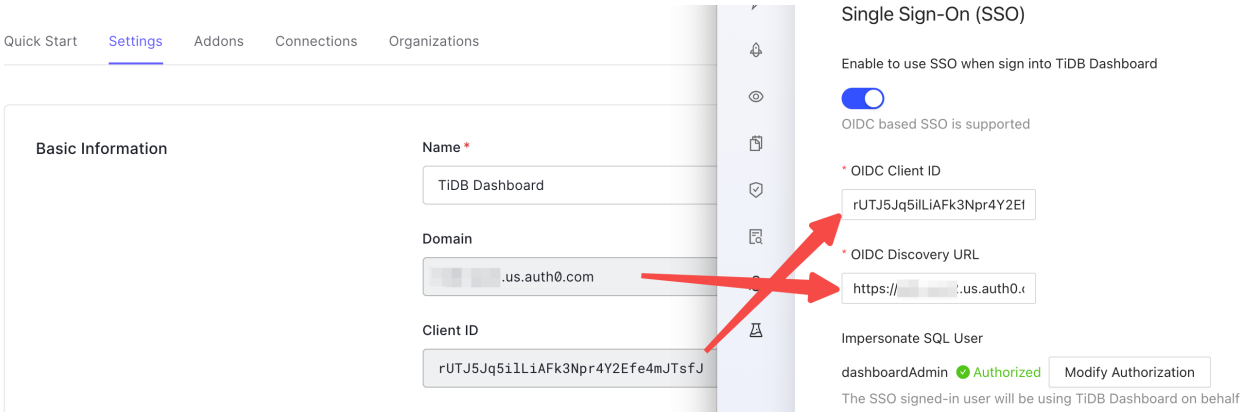


图 305: Settings

至此，TiDB Dashboard 已被配置为使用 Auth0 进行 SSO 登录。

11.9.14.2.5 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。

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。

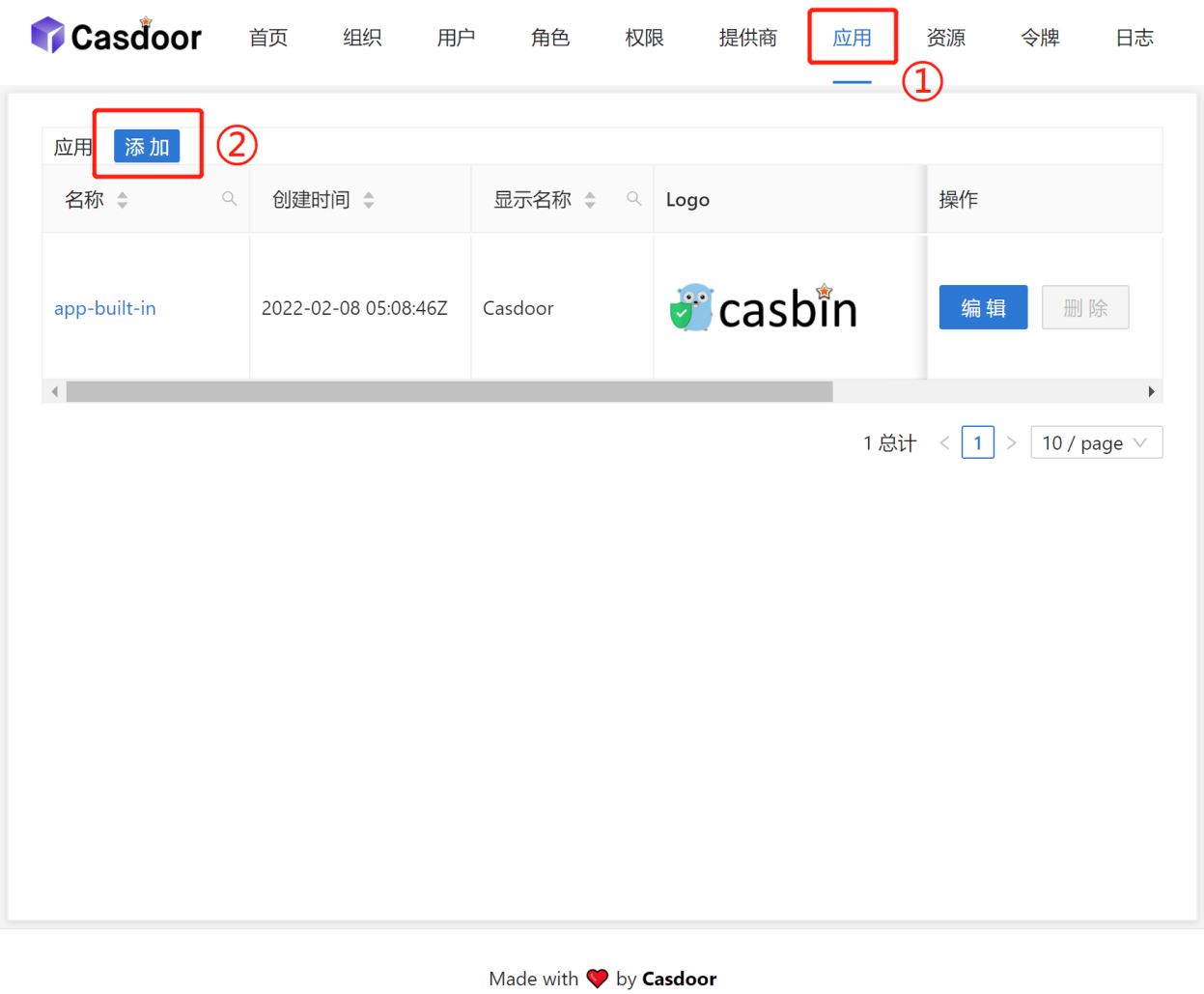


图 306: Settings

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 ⓘ：

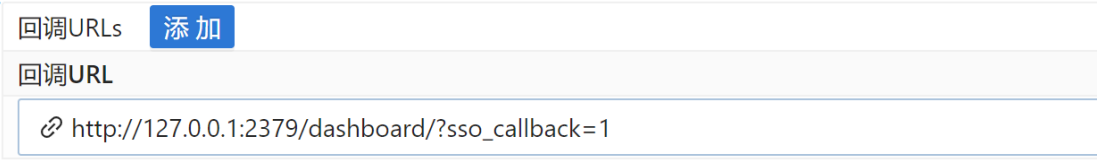


图 307: Settings

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。

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。

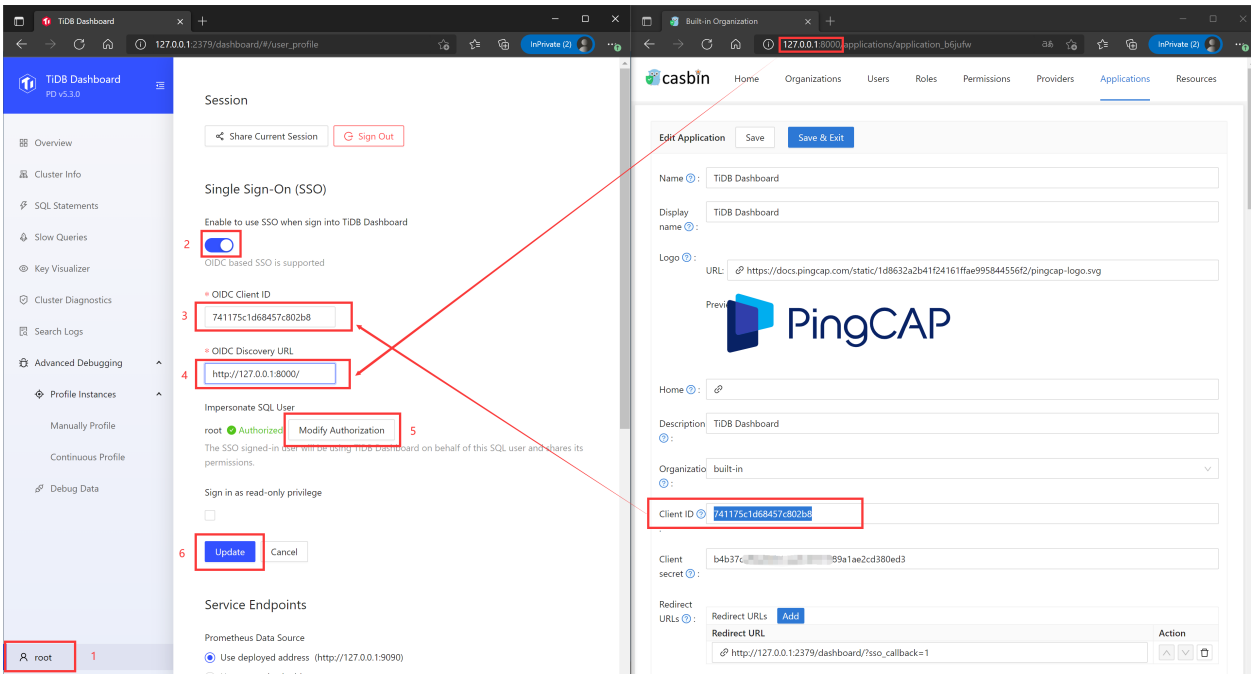


图 308: Settings

至此，TiDB Dashboard 已被配置为使用 Casdoor 进行 SSO 登录。

11.9.15 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，请联系 PingCAP 技术支持获取帮助。

11.9.15.1 访问

11.9.15.1.1 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 Dashbaord 服务，访问其他 PD 实例时会发生浏览器端重定向。若防火墙或反向代理没有为此进行正确配置，就可能出现访问后被重定向到一个被防火墙或反向代理保护的内部地址的情况。

- 参阅TiDB Dashboard 多 PD 实例部署章节了解多 PD 实例下 TiDB Dashboard 的工作原理。
- 参阅通过反向代理使用 TiDB Dashboard 章节了解如何正确配置反向代理。
- 参阅提高 TiDB Dashboard 安全性章节了解如何正确配置防火墙。

11.9.15.1.2 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。

11.9.15.2 界面功能

11.9.15.2.1 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。

11.9.15.2.2 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。

11.9.15.2.3 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，请联系 PingCAP 技术支持获取帮助。

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
```

11.9.15.2.4 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：

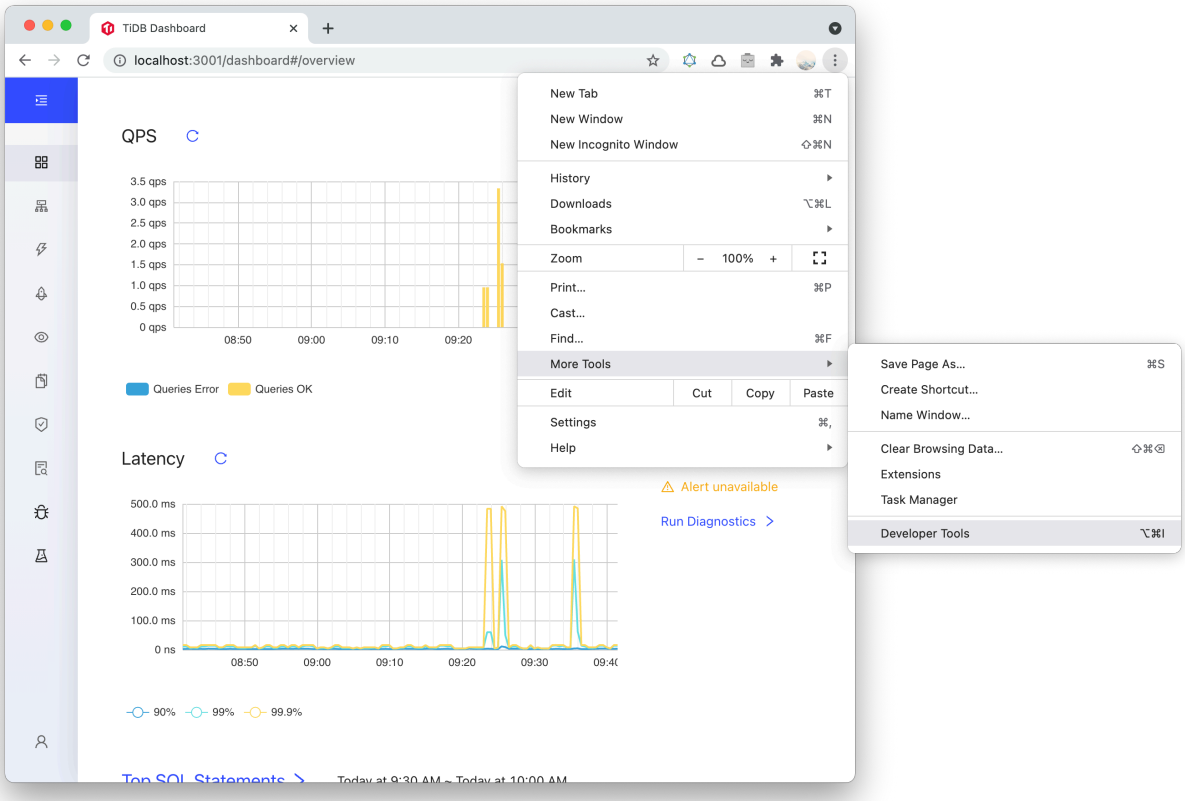


图 309: 打开开发者工具

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。

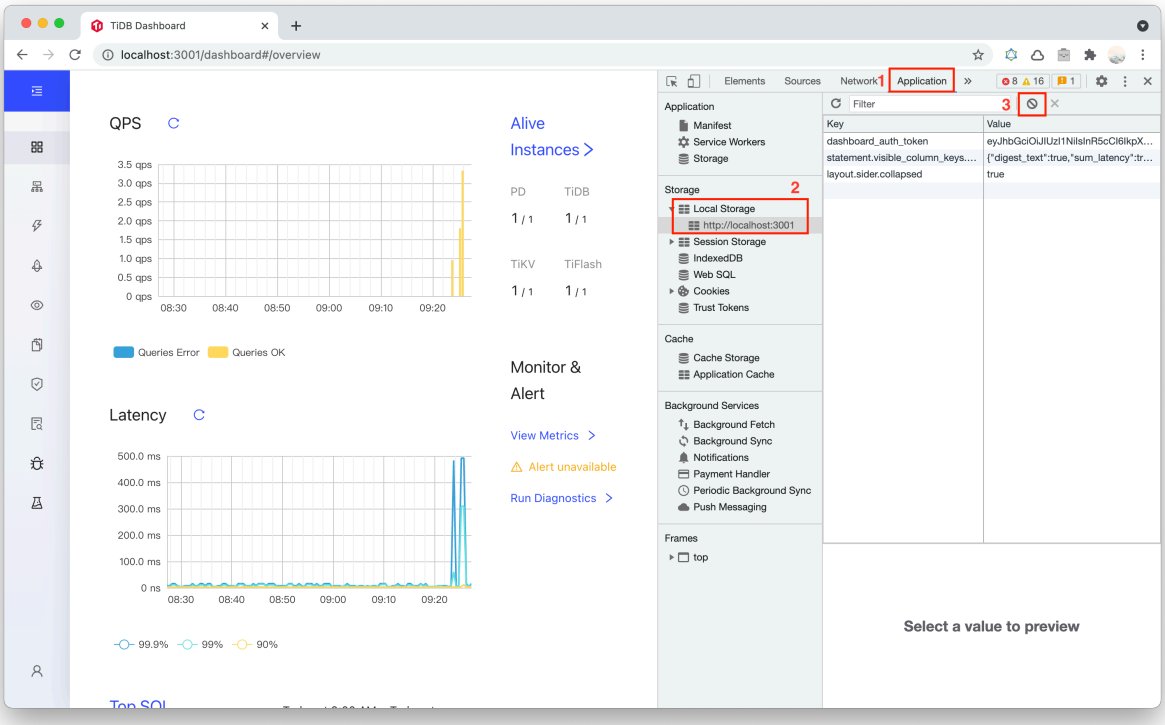


图 310: 清理 Local Storage

12 最佳实践

12.1 TiDB 最佳实践

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

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

12.1.1 前言

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

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

12.1.2 基本概念

TiDB 的最佳实践与其实现原理密切相关，建议读者先了解一些基本的实现机制，包括 Raft、分布式事务、数据分片、负载均衡、SQL 到 KV 的映射方案、二级索引的实现方法、分布式执行引擎。下面会做一点简单的介

绍，更详细的信息可以参考 PingCAP 公众号以及知乎专栏的一些文章。

12.1.2.1 Raft

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

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

12.1.2.2 分布式事务

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

- 乐观锁

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

- 悲观锁

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

- 事务大小限制

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

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

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

12.1.2.3 数据分片

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

12.1.2.4 负载均衡

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

12.1.2.5 SQL on KV

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

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

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

12.1.2.6 二级索引

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

- 二级索引是否越多越好

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

- 对哪些列建索引比较合适

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

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

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

- 通过索引查询和直接扫描 Table 的区别

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

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

  以下情况不会涉及到两次访问的问题：

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

- 查询并发度

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

  - tidb_distsql_scan_concurrency

  在进行扫描数据的时候的并发度，这里包括扫描 Table 以及索引数据。

  - tidb_index_lookup_size

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

  - tidb_index_lookup_concurrency

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

- 通过索引保证结果顺序

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

- 逆序索引

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

12.1.3 场景与实践

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

12.1.3.1 部署

在部署之前请务必阅读TiDB 部署建议以及对硬件的需求。

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

12.1.3.2 导入数据

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

12.1.3.3 写入

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

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

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

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

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

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

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

12.1.3.4 查询

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

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

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

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

12.1.3.5 监控和日志

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

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

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

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

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

12.1.3.6 文档

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

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

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

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

12.1.4 TiDB 的最佳适用场景

简单来说，TiDB 适合具备下面这些特点的场景：

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

12.2 开发 Java 应用使用 TiDB 的最佳实践

本文主要介绍如何开发 Java 应用程序以更好地使用 TiDB，包括开发中的常见问题与最佳实践。

12.2.1 Java 应用中的数据库相关组件

通常 Java 应用中和数据库相关的常用组件有：

- 网络协议：客户端通过标准 MySQL 协议和 TiDB 进行网络交互。
- JDBC API 及实现：Java 应用通常使用 JDBC (Java Database Connectivity) 来访问数据库。JDBC 定义了访问数据库 API，而 JDBC 实现完成标准 API 到 MySQL 协议的转换，常见的 JDBC 实现是 MySQL Connector/J，此外有些用户可能使用 MariaDB Connector/J。
- 数据库连接池：为了避免每次创建连接，通常应用会选择使用数据库连接池来复用连接，JDBC DataSource 定义了连接池 API，开发者可根据实际需求选择使用某种开源连接池实现。
- 数据访问框架：应用通常选择通过数据访问框架 (MyBatis, Hibernate) 的封装来进一步简化和管理数据库访问操作。
- 业务实现：业务逻辑控制着何时发送和发送什么指令到数据库，其中有些业务会使用 Spring Transaction 切面来控制管理事务的开始和提交逻辑。

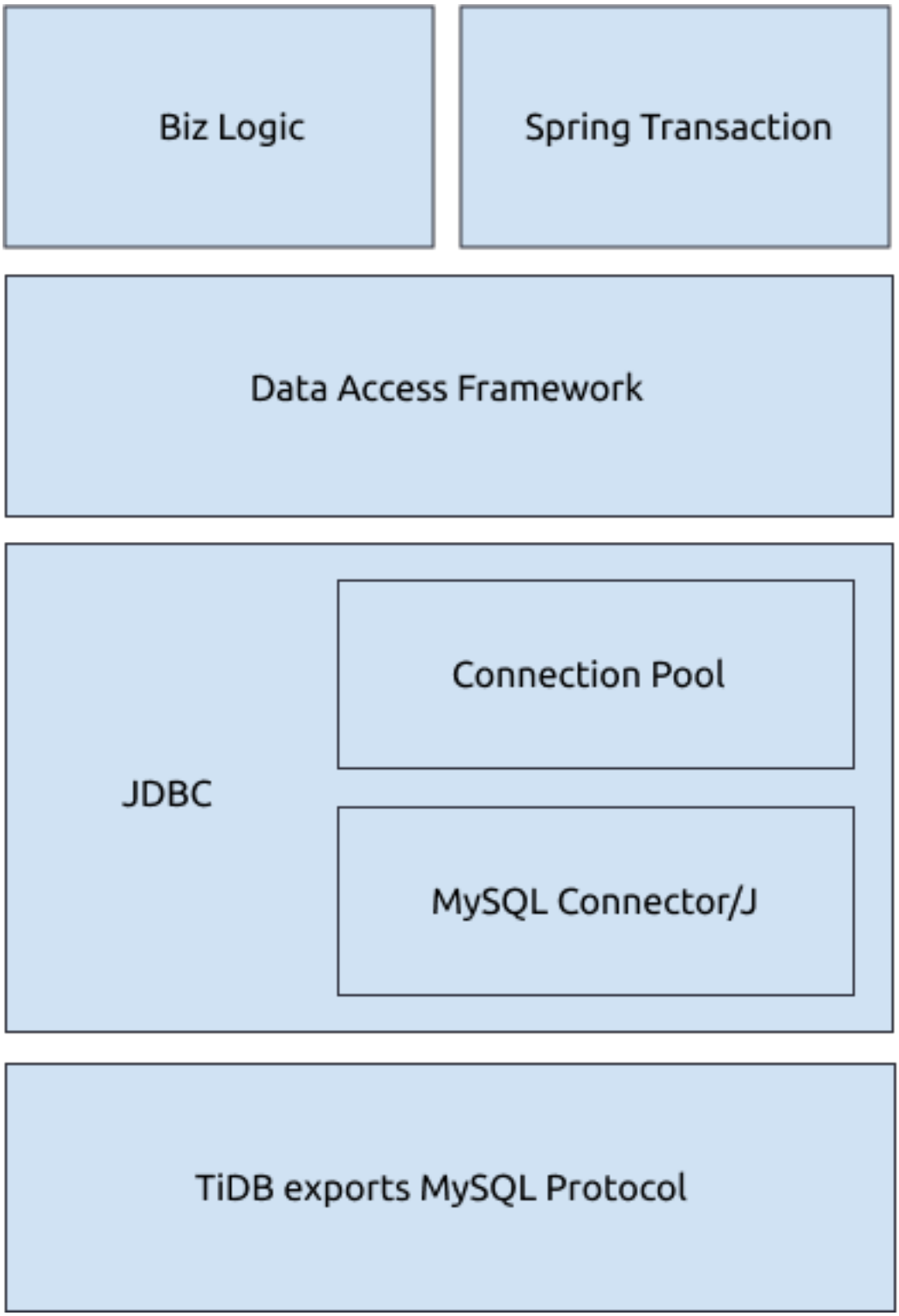


图 311: Java Component

如上图所示，应用可能使用 Spring Transaction 来管理控制事务非手工启停，通过类似 MyBatis 的数据访问框架管理生成和执行 SQL，通过连接池获取已池化的长连接，最后通过 JDBC 接口调用实现通过 MySQL 协议和 TiDB

完成交互。

接下来将分别介绍使用各个组件时可能需要关注的问题。

12.2.2 JDBC

Java 应用尽管可以选择在不同的框架中封装，但在最底层一般会通过调用 JDBC 来与数据库服务器进行交互。对于 JDBC，需要关注的主要有：API 的使用选择和 API Implementer 的参数配置。

12.2.2.1 JDBC API

对于基本的 JDBC API 使用可以参考 JDBC 官方教程，本文主要强调几个比较重要的 API 选择。

12.2.2.1.1 使用 Prepare API

对于 OLTP 场景，程序发送给数据库的 SQL 语句在去除参数变化后都是可穷举的某几类，因此建议使用预处理语句 (Prepared Statements) 代替普通的文本执行，并复用预处理语句来直接执行，从而避免 TiDB 重复解析和生成 SQL 执行计划的开销。

目前多数上层框架都会调用 Prepare API 进行 SQL 执行，如果直接使用 JDBC API 进行开发，注意选择使用 Prepare API。

另外需要注意 MySQL Connector/J 实现中默认只会做客户端的语句预处理，会将 `?` 在客户端替换后以文本形式发送到服务端，所以除了要使用 Prepare API，还需要在 JDBC 连接参数中配置 `useServerPrepStmts = true`，才能在 TiDB 服务器端进行语句预处理（下面参数配置章节有详细介绍）。

12.2.2.1.2 使用 Batch 批量插入更新

对于批量插入更新，如果插入记录较多，可以选择使用 addBatch/executeBatch API。通过 addBatch 的方式将多条 SQL 的插入更新记录先缓存在客户端，然后在 executeBatch 时一起发送到数据库服务器。

> **注意：**
>
> 对于 MySQL Connector/J 实现，默认 Batch 只是将多次 addBatch 的 SQL 发送时机延迟到调用 executeBatch 的时候，但实际网络发送还是会一条条的发送，通常不会降低与数据库服务器的网络交互次数。
>
> 如果希望 Batch 网络发送，需要在 JDBC 连接参数中配置 `rewriteBatchedStatements = true`（下面参数配置章节有详细介绍）。

12.2.2.1.3 使用 StreamingResult 流式获取执行结果

一般情况下，为提升执行效率，JDBC 会默认提前获取查询结果并将其保存在客户端内存中。但在查询返回超大结果集的场景中，客户端会希望数据库服务器减少向客户端一次返回的记录数，等客户端在有限内存处理完一部分后再去向服务器要下一批。

在 JDBC 中通常有以下两种处理方式：

- 设置 FetchSize 为 Integer.MIN_VALUE 让客户端不缓存，客户端通过 StreamingResult 的方式从网络连接上流式读取执行结果。

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

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

- 使用 Cursor Fetch，首先需设置 FetchSize 为正整数，且在 JDBC URL 中配置 useCursorFetch = true。

TiDB 中同时支持两种方式，但更推荐使用第一种将 FetchSize 设置为 Integer.MIN_VALUE 的方式，比第二种功能实现更简单且执行效率更高。

12.2.2.2 MySQL JDBC 参数

JDBC 实现通常通过 JDBC URL 参数的形式来提供实现相关的配置。这里以 MySQL 官方的 Connector/J 来介绍参数配置（如果使用的是 MariaDB，可以参考 MariaDB 的类似配置）。因为配置项较多，这里主要关注几个可能影响到性能的参数。

12.2.2.2.1 Prepare 相关参数

useServerPrepStmts

默认情况下，useServerPrepStmts 的值为 false，即尽管使用了 Prepare API，也只会在客户端做“prepare”。因此为了避免服务器重复解析的开销，如果同一条 SQL 语句需要多次使用 Prepare API，则建议设置该选项为 true。

在 TiDB 监控中可以通过 Query Summary > QPS By Instance 查看请求命令类型，如果请求中 COM_QUERY 被 COM_STMT_EXECUTE 或 COM_STMT_PREPARE 代替即生效。

cachePrepStmts

虽然 useServerPrepStmts = true 能让服务端执行预处理语句，但默认情况下客户端每次执行完后会 close 预处理语句，并不会复用，这样预处理的效率甚至不如文本执行。所以建议开启 useServerPrepStmts = true 后同时配置 cachePrepStmts = true，这会让客户端缓存预处理语句。

在 TiDB 监控中可以通过 Query Summary > QPS By Instance 查看请求命令类型，如果类似下图，请求中 COM_STMT_EXECUTE 数目远远多于 COM_STMT_PREPARE 即生效。

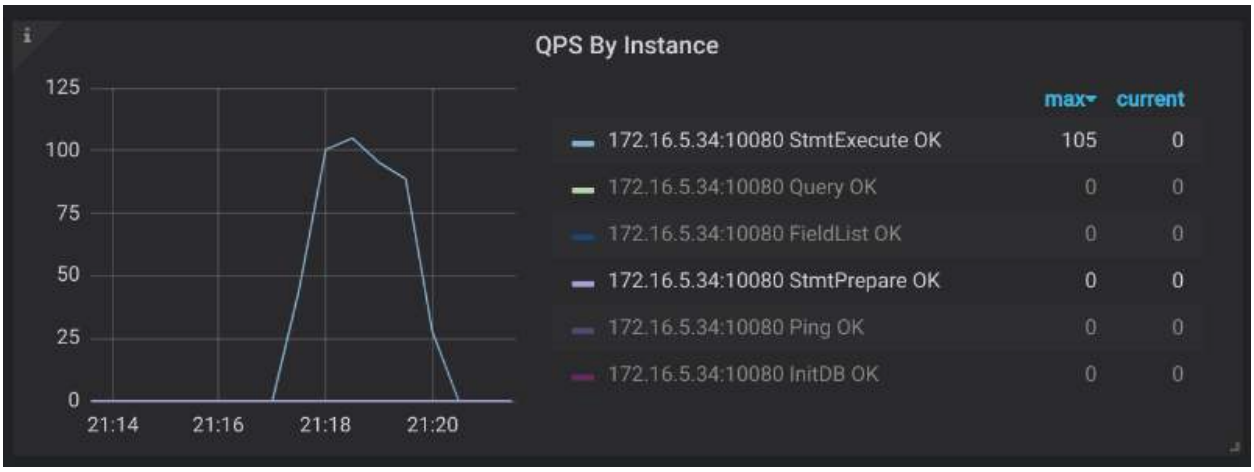


图 312: QPS By Instance

另外，通过 `useConfigs = maxPerformance` 配置会同时配置多个参数，其中也包括 `cachePrepStmts = true`。

`prepStmtCacheSqlLimit`

在配置 `cachePrepStmts` 后还需要注意 `prepStmtCacheSqlLimit` 配置（默认为 256），该配置控制客户端缓存预处理语句的最大长度，超过该长度将不会被缓存。

在一些场景 SQL 的长度可能超过该配置，导致预处理 SQL 不能复用，建议根据应用 SQL 长度情况决定是否需要调大该值。

在 TiDB 监控中通过 Query Summary > QPS by Instance 查看请求命令类型，如果已经配置了 `cachePrepStmts = true`，但 `COM_STMT_PREPARE` 还是和 `COM_STMT_EXECUTE` 基本相等且有 `COM_STMT_CLOSE`，需要检查这个配置项是否设置得太小。

`prepStmtCacheSize`

`prepStmtCacheSize` 控制缓存的预处理语句数目（默认为 25），如果应用需要预处理的 SQL 种类很多且希望复用预处理语句，可以调大该值。

和上一条类似，在监控中通过 Query Summary > QPS by Instance 查看请求中 `COM_STMT_EXECUTE` 数目是否远远多于 `COM_STMT_PREPARE` 来确认是否正常。

12.2.2.2.2 Batch 相关参数

在进行 batch 写入处理时推荐配置 `rewriteBatchedStatements = true`，在已经使用 `addBatch` 或 `executeBatch` 后默认 JDBC 还是会一条条 SQL 发送，例如：

```
pstmt = prepare(“insert into t (a) values(?)”);
pstmt.setInt(1, 10);
pstmt.addBatch();
pstmt.setInt(1, 11);
pstmt.addBatch();
pstmt.setInt(1, 12);
pstmt.executeBatch();
```

虽然使用了 batch 但发送到 TiDB 语句还是单独的多条 insert：

```
insert into t(a) values(10);
insert into t(a) values(11);
insert into t(a) values(12);
```

如果设置 rewriteBatchedStatements = true，发送到 TiDB 的 SQL 将是：

```
insert into t(a) values(10),(11),(12);
```

需要注意的是，insert 语句的改写，只能将多个 values 后的值拼接成一整条 SQL，insert 语句如果有其他差异将无法被改写。例如：

```
insert into t (a) values (10) on duplicate key update a = 10;
insert into t (a) values (11) on duplicate key update a = 11;
insert into t (a) values (12) on duplicate key update a = 12;
```

上述 insert 语句将无法被改写成一条语句。该例子中，如果将 SQL 改写成如下形式：

```
insert into t (a) values (10) on duplicate key update a = values(a);
insert into t (a) values (11) on duplicate key update a = values(a);
insert into t (a) values (12) on duplicate key update a = values(a);
```

即可满足改写条件，最终被改写成：

```
insert into t (a) values (10), (11), (12) on duplicate key update a = values(a);
```

批量更新时如果有 3 处或 3 处以上更新，则 SQL 语句会改写为 multiple-queries 的形式并发送，这样可以有效减少客户端到服务器的请求开销，但副作用是会产生较大的 SQL 语句，例如这样：

```
update t set a = 10 where id = 1; update t set a = 11 where id = 2; update t set a = 12 where id
    ↪ = 3;
```

另外，因为一个客户端 bug，批量更新时如果要配置 rewriteBatchedStatements = true 和 useServerPrepStmts ↪ = true，推荐同时配置 allowMultiQueries = true 参数来避免这个 bug。

12.2.2.2.3 集成参数

通过监控可能会发现，虽然业务只向集群进行 insert 操作，却看到有很多多余的 select 语句。通常这是因为 JDBC 发送了一些查询设置类的 SQL 语句（例如 select @@session.transaction_read_only）。这些 SQL 对 TiDB 无用，推荐配置 useConfigs = maxPerformance 来避免额外开销。

useConfigs = maxPerformance 会包含一组配置：

```
cacheServerConfiguration = true
useLocalSessionState = true
elideSetAutoCommits = true
alwaysSendSetIsolation = false
enableQueryTimeouts = false
```

配置后查看监控，可以看到多余语句减少。

12.2.2.2.4 超时参数

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

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

12.2.3 连接池

TiDB (MySQL) 连接建立是比较昂贵的操作（至少对于 OLTP），除了建立 TCP 连接外还需要进行连接鉴权操作，所以客户端通常会把 TiDB (MySQL) 连接保存到连接池中进行复用。

Java 的连接池实现很多 (HikariCP, tomcat-jdbc, druid, c3p0, dbcp)，TiDB 不会限定使用的连接池，应用可以根据业务特点自行选择连接池实现。

12.2.3.1 连接数配置

比较常见的是应用需要根据自身情况配置合适的连接池大小，以 HikariCP 为例：

- `maximumPoolSize`：连接池最大连接数，配置过大会导致 TiDB 消耗资源维护无用连接，配置过小则会导致应用获取连接变慢，所以需根据应用自身特点配置合适的值，可参考这篇文章。
- `minimumIdle`：连接池最小空闲连接数，主要用于在应用空闲时存留一些连接以应对突发请求，同样是需要根据业务情况进行配置。

应用在使用连接池时，需要注意连接使用完成后归还连接，推荐应用使用对应的连接池相关监控（如 `metricRegistry`），通过监控能及时定位连接池问题。

12.2.3.2 探活配置

连接池维护到 TiDB 的长连接，TiDB 默认不会主动关闭客户端连接（除非报错），但一般客户端到 TiDB 之间还会有 LVS 或 HAProxy 之类的网络代理，它们通常会在连接空闲一定时间后主动清理连接。除了注意代理的 idle 配置外，连接池还需要进行保活或探测连接。

如果常在 Java 应用中看到以下错误：

```
The last packet sent successfully to the server was 3600000 milliseconds ago. The driver has not
    ↪ received any packets from the server. com.mysql.jdbc.exceptions.jdbc4.
    ↪ CommunicationsException: Communications link failure
```

如果 `n milliseconds ago` 中的 n 如果是 0 或很小的值，则通常是执行的 SQL 导致 TiDB 异常退出引起的报错，推荐查看 TiDB stderr 日志；如果 n 是一个非常大的值（比如这里的 3600000），很可能是因为这个连接空闲太久然后被中间 proxy 关闭了，通常解决方式除了调大 proxy 的 idle 配置，还可以让连接池执行以下操作：

- 每次使用连接前检查连接是否可用。

- 使用单独线程定期检查连接是否可用。
- 定期发送 test query 保活连接。

不同的连接池实现可能会支持其中一种或多种方式，可以查看所使用的连接池文档来寻找对应配置。

12.2.4 数据访问框架

业务应用通常会使用某种数据访问框架来简化数据库的访问。

12.2.4.1 MyBatis

MyBatis 是目前比较流行的 Java 数据访问框架，主要用于管理 SQL 并完成结果集和 Java 对象的来回映射工作。MyBatis 和 TiDB 兼容性很好，从历史 issue 可以看出 MyBatis 很少出现问题。这里主要关注如下几个配置。

12.2.4.1.1 Mapper 参数

MyBatis 的 Mapper 中支持两种参数：

- `select 1 from t where id = #{param1}` 会作为预处理语句，被转换为 `select 1 from t where id = ?` 进行预处理，并使用实际参数来复用执行，通过配合前面的 Prepare 连接参数能获得最佳性能。
- `select 1 from t where id = ${param2}` 会做文本替换为 `select 1 from t where id = 1` 执行，如果这条语句被预处理为不同参数，可能会导致 TiDB 缓存大量的预处理语句，并且以这种方式执行 SQL 有注入安全风险。

12.2.4.1.2 动态 SQL Batch

动态 SQL - foreach

要支持将多条 insert 语句自动重写为 `insert ... values(...), (...), ...` 的形式，除了前面所说的在 JDBC 配置 `rewriteBatchedStatements = true` 外，MyBatis 还可以使用动态 SQL 来半自动生成 batch insert。比如下面的 mapper:

```
<insert id="insertTestBatch" parameterType="java.util.List" fetchSize="1">
  insert into test
   (id, v1, v2)
  values
  <foreach item="item" index="index" collection="list" separator=",">
  (
   #{item.id}, #{item.v1}, #{item.v2}
  )
  </foreach>
  on duplicate key update v2 = v1 + values(v1)
</insert>
```

会生成一个 `insert on duplicate key update` 语句，values 后面的 `(?, ?, ?)` 数目是根据传入的 list 个数决定，最终效果和使用 `rewriteBatchStatements = true` 类似，可以有效减少客户端和 TiDB 的网络交互次数，同样需要注意预处理后超过 `prepStmtCacheSqlLimit` 限制导致不缓存预处理语句的问题。

12.2.4.1.3 Streaming 结果

前面介绍了在 JDBC 中如何使用流式读取结果，除了 JDBC 相应的配置外，在 MyBatis 中如果希望读取超大结果集合也需要注意：

- 可以通过在 mapper 配置中对单独一条 SQL 设置 `fetchSize`（见上一段代码段），效果等同于调用 JDBC `setFetchSize`
- 可以使用带 `ResultHandler` 的查询接口来避免一次获取整个结果集
- 可以使用 `Cursor` 类来进行流式读取

对于使用 xml 配置映射，可以通过在映射 `<select>` 部分配置 `fetchSize="-2147483648"(Integer.MIN_VALUE)` 来流式读取结果。

```
<select id="getAll" resultMap="postResultMap" fetchSize="-2147483648">
  select * from post;
</select>
```

而使用代码配置映射，则可以使用 `@Options(fetchSize = Integer.MIN_VALUE)` 并返回 `Cursor` 从而让 SQL 结果能被流式读取。

```
@Select("select * from post")
@Options(fetchSize = Integer.MIN_VALUE)
Cursor<Post> queryAllPost();
```

12.2.4.2 ExecutorType

在 openSession 的时候可以选择 `ExecutorType`，MyBatis 支持三种 executor：

- Simple：每次执行都会向 JDBC 进行预处理语句的调用（如果 JDBC 配置有开启 `cachePrepStmts`，重复的预处理语句会复用）。
- Reuse：在 `executor` 中缓存预处理语句，这样不用 JDBC 的 `cachePrepStmts` 也能减少重复预处理语句的调用。
- Batch：每次更新只有在 addBatch 到 query 或 commit 时才会调用 `executeBatch` 执行，如果 JDBC 层开启了 `rewriteBatchStatements`，则会尝试改写，没有开启则会一条条发送。

通常默认值是 `Simple`，需要在调用 `openSession` 时改变 `ExecutorType`。如果是 Batch 执行，会遇到事务中前面的 update 或 insert 都非常快，而在读数据或 commit 事务时比较慢的情况，这实际上是正常的，在排查慢 SQL 时需要注意。

12.2.5 Spring Transaction

在应用代码中业务可能会通过使用 Spring Transaction 和 AOP 切面的方式来启停事务。

通过在方法定义上添加 `@Transactional` 注解标记方法，AOP 将会在方法前开启事务，方法返回结果前 commit 事务。如果遇到类似业务，可以通过查找代码 `@Transactional` 来确定事务的开启和关闭时机。需要特别注意有内嵌的情况，如果发生内嵌，Spring 会根据 Propagation 配置使用不同的行为，因为 TiDB 未支持 savepoint，所以不支持嵌套事务。

12.2.6 其他

12.2.6.1 排查工具

在 Java 应用发生问题并且不知道业务逻辑情况下，使用 JVM 强大的排查工具会比较有用。这里简单介绍几个常用工具：

12.2.6.1.1 jstack

jstack 对应于 Go 中的 pprof/goroutine，可以比较方便地排查进程卡死的问题。

通过执行 `jstack pid`，即可输出目标进程中所有线程的线程 id 和堆栈信息。输出中默认只有 Java 堆栈，如果希望同时输出 JVM 中的 C++ 堆栈，需要加 `-m` 选项。

通过多次 jstack 可以方便地发现卡死问题（比如：都通过 Mybatis BatchExecutor flush 调用 update）或死锁问题（比如：测试程序都在抢占应用中某把锁导致没发送 SQL）

另外，`top -p $PID -H` 或者 Java swiss knife 都是常用的查看线程 ID 的方法。通过 `printf "%x\n" pid` 把线程 ID 转换成 16 进制，然后去 jstack 输出结果中找对应线程的栈信息，可以定位”某个线程占用 CPU 比较高，不知道它在执行什么”的问题。

12.2.6.1.2 jmap & mat

和 Go 中的 pprof/heap 不同，jmap 会将整个进程的内存快照 dump 下来（go 是分配器的采样），然后可以通过另一个工具 mat 做分析。

通过 mat 可以看到进程中所有对象的关联信息和属性，还可以观察线程运行的状态。比如：我们可以通过 mat 找到当前应用中有多少 MySQL 连接对象，每个连接对象的地址和状态信息是什么。

需要注意 mat 默认只会处理 reachable objects，如果要排查 young gc 问题可以在 mat 配置中设置查看 unreachable objects。另外对于调查 young gc 问题（或者大量生命周期较短的对象）的内存分配，用 Java Flight Recorder 比较方便。

12.2.6.1.3 trace

线上应用通常无法修改代码，又希望在 Java 中做动态插桩来定位问题，推荐使用 btrace 或 arthas trace。它们可以在不重启进程的情况下动态插入 trace 代码。

12.2.6.1.4 火焰图

Java 应用中获取火焰图较繁琐，可参阅 Java Flame Graphs Introduction: Fire For Everyone! 来手动获取。

12.2.7 总结

本文从常用的和数据库交互的 Java 组件的角度，阐述了开发 Java 应用程序使用 TiDB 的常见问题与解决办法。TiDB 是高度兼容 MySQL 协议的数据库，基于 MySQL 开发的 Java 应用的最佳实践也多适用于 TiDB。

欢迎大家在 ASK TUG 踊跃发言，和我们一起分享讨论 Java 应用使用 TiDB 的实践技巧或遇到的问题。

12.3 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。

12.3.1 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。

12.3.1.1 60 秒分析法

此分析法由《性能之巅》的作者 Brendan Gregg 及其所在的 Netflix 性能工程团队公布。所用到的工具均可从发行版的官方源获取，通过分析以下清单中的输出，可定位大部分常见的性能问题。

- uptime
- dmesg | tail
- vmstat 1
- mpstat -P ALL 1
- pidstat 1
- iostat -xz 1
- free -m
- sar -n DEV 1
- sar -n TCP,ETCP 1
- top

具体用法可查询相应 man 手册。

12.3.1.2 perf

Perf 是 Linux 内核提供的一个重要的性能分析工具，它涵盖硬件级别（CPU/PMU 和性能监视单元）功能和软件功能（软件计数器和跟踪点）。详细用法请参考 perf Examples。

12.3.1.3 BCC/bpftrace

CentOS 从 7.6 版本起，内核已实现对 bpf 的支持，因此可根据上述清单的结果，选取适当的工具进行深入分析。相比 perf/ftrace，bpf 提供了可编程能力和更小的性能开销。相比 kprobe，bpf 提供了更高的安全性，更适合在生产环境上使用。关于 BCC 工具集的使用请参考 BPF Compiler Collection (BCC)。

12.3.2 性能调优

性能调优将根据内核子系统进行分类描述。

12.3.2.1 处理器 ——动态节能技术

cpufreq 是一个动态调整 CPU 频率的模块，可支持五种模式。为保证服务性能应选用 performance 模式，将 CPU 频率固定工作在其支持的最高运行频率上，不进行动态调节，操作命令为 `cpupower frequency-set --governor performance`。

12.3.2.2 处理器 ——中断亲和性

- 自动平衡：可通过 `irqbalance` 服务实现。
- 手动平衡：
  - 确定需要平衡中断的设备，从 CentOS 7.5 开始，系统会自动为某些设备及其驱动程序配置最佳的中断关联性。不能再手动配置其亲和性。目前已知的有使用 `be2iscsi` 驱动的设备，以及 NVMe 设置；
  - 对于其他设备，可查询其芯片手册，是否支持分发中断，若不支持，则该设备的所有中断会路由到同一个 CPU 上，无法对其进行修改。若支持，则计算 `smp_affinity` 掩码并设置对应的配置文件，具体请参考内核文档。

12.3.2.3 NUMA 绑核

为尽可能的避免跨 NUMA 访问内存，可以通过设置线程的 CPU 亲和性来实现 NUMA 绑核。对于普通程序，可使用 `numactl` 命令来绑定，具体用法请查询 `man` 手册。对于网卡中断，请参考下文网络章节。

12.3.2.4 内存 ——透明大页

对于数据库应用，不推荐使用 THP，因为数据库往往具有稀疏而不是连续的内存访问模式，且当高阶内存碎片化比较严重时，分配 THP 页面会出现较大的延迟。若开启针对 THP 的直接内存规整功能，也会出现系统 CPU 使用率激增的现象，因此建议关闭 THP。

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

12.3.2.5 内存 ——虚拟内存参数

- `dirty_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统将开始使用 pdflush 操作将脏的 page cache 写入磁盘。默认值为 20%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，降低其值有利于提高内存回收时的效率。
- `dirty_background_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统开始在后台将脏的 page cache 写入磁盘。默认值为 10%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，设置较低的值有利于提高内存回收时的效率。

12.3.2.6 存储及文件系统

内核 I/O 栈链路较长，包含了文件系统层、块设备层和驱动层。

12.3.2.6.1 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
```

12.3.2.6.2 格式化参数 ——块大小

块是文件系统的工作单元。块大小决定了单个块中可以存储多少数据，因此决定了一次写入或读取的最小数据量。

默认块大小适用于大多数使用情况。但是，如果块大小（或多个块的大小）与通常一次读取或写入的数据量相同或稍大，则文件系统将性能更好，数据存储效率更高。小文件仍将使用整个块。文件可以分布在多个块中，但这会增加运行时开销。

使用 mkfs 命令格式化设备时，将块大小指定为文件系统选项的一部分。指定块大小的参数随文件系统的不同而不同。有关详细信息，请查询对应文件系统的 `mkfs` 手册页，比如 `man mkfs.ext4`。

12.3.2.6.3 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。

12.3.2.7 网络

网络子系统由具有敏感连接的许多不同部分组成。因此，CentOS 7 网络子系统旨在为大多数工作负载提供最佳性能，并自动优化其性能。因此，通常无需手动调整网络性能。

网络问题通常由硬件或相关设施出现问题导致的，因此在调优协议栈前，请先排除硬件问题。

尽管网络堆栈在很大程度上是自我优化的。但是在网络数据包处理过程中，以下方面可能会成为瓶颈并降低性能：

- 网卡硬件缓存：正确观察硬件层面的丢包方法是使用 `ethtool -S ${NIC_DEV_NAME}` 命令观察 drops 字段。当出现丢包现象时，主要考虑是硬/软中断的处理速度跟不上网卡接收速度。若接收缓存小于最大限制时，也可尝试增加 RX 缓存来防止丢包。查询命令为：`ethtool -g ${NIC_DEV_NAME}`，修改命令为 `ethtool -G ${NIC_DEV_NAME}`。
- 硬中断：若网卡支持 Receive-Side Scaling（RSS 也称为多网卡接收）功能，则观察 `/proc/interrputs` 网卡中断，如果出现了中断不均衡的情况，请参考处理器调优章节。若不支持 RSS 或 RSS 数量远小于物理 CPU 核数，则可配置 Receive Packet Steering（RPS，可以看作 RSS 的软件实现），及 RPS 的扩展 Receive Flow Steering (RFS)。具体设置请参考内核文档。
- 软中断：观察 `/proc/net/softnet\_stat` 监控。如果除第三列的其他列的数值在增长，则应适度调大 `net.core.netdev\_budget` 或 `net.core.dev\_weight` 值，使 softirq 可以获得更多的 CPU 时间。除此之外，也需要检查 CPU 使用情况，确定哪些任务在频繁占用 CPU，能否优化。
- 应用的套接字接收队列：监控 `ss -nmp` 的 Recv-q 列，若队列已满，则应考虑增大应用程序套接字的缓存大小或使用自动调整缓存的方式。除此之外，也要考虑能否优化应用层的架构，降低读取套接字的间隔。

- 以太网流控：若网卡和交换机支持流控功能，可通过使能此功能，给内核一些时间来处理网卡队列中的数据，来规避网卡缓存溢出的问题。对于网卡测，可通过 `ethtool -a ${NIC_DEV_NAME}` 命令检查是否支持/使能，并通过 `ethtool -A ${NIC_DEV_NAME}` 命令开启。对于交换机，请查询其手册。
- 中断合并：过于频繁的硬件中断会降低系统性能，而过晚的硬件中断会导致丢包。对于较新的网卡支持中断合并功能，并允许驱动自动调节硬件中断数。可通过 `ethtool -c ${NIC_DEV_NAME}` 命令检查，`ethtool -C ${NIC_DEV_NAME}` 命令开启。自适应模式使网卡可以自动调节中断合并。在自适应模式下，驱动程序将检查流量模式和内核接收模式，并实时评估合并设置，以防止数据包丢失。不同品牌的网卡具有不同的功能和默认配置，具体请参考网卡手册。
- 适配器队列：在协议栈处理之前，内核利用此队列缓存网卡接收的数据，每个 CPU 都有各自的 backlog 队列。此队列可缓存的最大 packets 数量为 `netdev\_max\_backlog`。观察 `/proc/net/softnet\_stat` 第二列，当某行的第二列持续增加，则意味着 CPU [行 -1] 队列已满，数据包被丢失，可通过持续加倍 `net.core.netdev\_max\_backlog` 值来解决。
- 发送队列：发送队列长度值确定在发送之前可以排队的数据包数量。默认值是 1000，对于 10 Gbps 足够。但若从 `ip -s link` 的输出中观察到 `TX errors` 值时，可尝试加倍该数据包数量：`ip link set dev ${NIC_DEV_NAME} txqueuelen 2000`。
- 驱动：网卡驱动通常也会提供调优参数，请查询设备硬件手册及其驱动文档。

12.4 HAProxy 在 TiDB 中的最佳实践

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

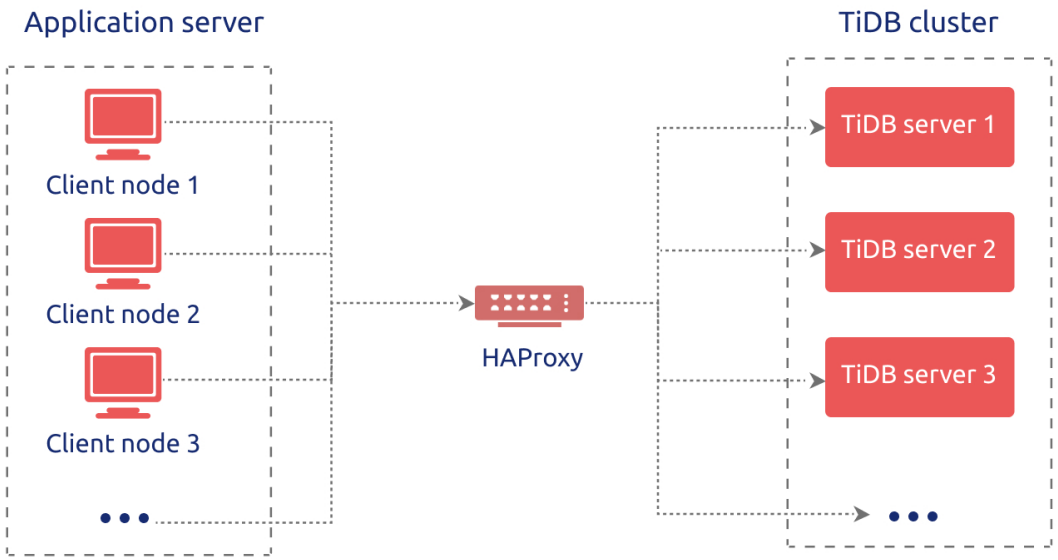


图 313: HAProxy 在 TiDB 中的最佳实践

12.4.1 HAProxy 简介

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

HAProxy 由 Linux 内核的核心贡献者 Willy Tarreau 于 2000 年编写，他现在仍然负责该项目的维护，并在开源社区免费提供版本迭代。本文示例使用 HAProxy 2.5.0。推荐使用最新稳定版的 HAProxy，详情见已发布的 HAProxy 版本。

12.4.2 HAProxy 部分核心功能介绍

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

12.4.3 准备环境

在部署 HAProxy 之前，需准备好以下环境。

12.4.3.1 硬件要求

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

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

12.4.3.2 依赖软件

根据 HAProxy 官方文档，对操作系统和依赖包有以下建议，如果通过 yum 源部署安装 HAProxy 软件，依赖包无需单独安装。

12.4.3.2.1 操作系统

Linux 操作系统	版本
Red Hat Enterprise Linux	7 或者 8

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

注意：

- 其他操作系统支持情况，详见 HAProxy 文档。

12.4.3.2.2 依赖包

- epel-release
- gcc
- systemd-devel

执行如下命令安装依赖包：

```
yum -y install epel-release gcc systemd-devel
```

12.4.4 部署 HAProxy

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

12.4.4.1 安装 HAProxy

1. 下载 HAProxy 2.5.0 的源码包：

   ```
   wget https://github.com/haproxy/haproxy/archive/refs/tags/v2.5.0.zip
   ```

2. 解压源码包：

   ```
   unzip v2.5.0.zip
   ```

3. 从源码编译 HAProxy 应用：

   ```
   cd haproxy-2.5.0
   make clean
   make -j 8 TARGET=linux-glibc USE_THREAD=1
   make PREFIX=${/app/haproxy} SBINDIR=${/app/haproxy/bin} install  # 将 `${/app/haproxy}` 和 `
       ↪ ${/app/haproxy/bin}` 替换为自定义的实际路径。
   ```

4. 重新配置 profile 文件：

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

5. 检查 HAProxy 是否安装成功：

```
which haproxy
```

12.4.4.1.1 HAProxy 命令介绍

执行如下命令查看命令行参数及基本用法：

```
haproxy --help
```

参数	说明
-v	显示简略的版本信息。
-vv	显示详细的版本信息。
-d	开启 debug 模式。
-db	禁用后台模式和多进程模式。
-dM [<byte >]	执行分配内存。
-V	启动过程显示配置和轮询信息。
-D	开启守护进程模式。
-C <dir>	在加载配置文件之前更改目录位置至 <dir>。
-W	主从模式。
-q	静默模式，不输出信息。
-c	只检查配置文件并在尝试绑定之前退出。
-n <limit>	设置每个进程的最大总连接数为 <limit>。

参数	说明
-m <limit>	设置所有进程的最大可用内存为 <limit>（单位：MB）。
-N <limit>	设置单点最大连接数为 <limit>，默认为 2000。
-L <name>	将本地实例对等名称改为 <name>，默认为本地主机名。
-p <file>	将 HAProxy 所有子进程的 PID 信息写入 <file>。
-de	禁止使用 epoll(7)，epoll(7) 仅在 Linux 2.6 和某些定制的 Linux 2.4 系统上可用。
-dp	禁止使用 epoll(2)，可改用 select(2)。
-dS	禁止使用 splice(2)，splice(2) 在一些旧版 Linux 内核上不可用。
-dR	禁止使用 SO_REUSEPORT。
-dr	忽略服务器地址解析失败。
-dV	禁止在服务器端使用 SSL。

参数	说明
`-sf <pidlist>`	启动后，向 pidlist 中的 PID 发送 `finish` 信号，收到此信号的进程在退出之前将等待所有会话完成，即优雅停止服务。此选项必须最后指定，后跟任意数量的 PID。从技术上讲，SIGTTOU 和 SIGUSR1 都被发送。
`-st <pidlist>`	启动后，向 pidlist 中的 PID 发送 `terminate` 信号，收到此信号的进程将立即终止，关闭所有活动会话。此选项必须最后指定，后跟任意数量的 PID。从技术上讲，SIGTTOU 和 SIGTERM 都被发送。
`-x <unix_socket>`	连接指定的 socket 并从旧进程中获取所有 listening socket，然后，使用这些 socket 而不是绑定新的。

参数	说明
-S <bind >[,< bind_option >...]	主从模式下，创建绑定到主进程的 socket，此 socket 可访问每个子进程的 socket。

更多有关 HAProxy 命令参数的信息，可参阅 Management Guide of HAProxy 和 General Commands Manual of HAProxy。

12.4.4.2 配置 HAProxy

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

```
global                                  # 全局配置。
   log         127.0.0.1 local2         # 定义全局的 syslog 服务器，最多可以定义两个。
   chroot      /var/lib/haproxy         # 更改当前目录并为启动进程设置超级用户权限，
       ↪ 从而提高安全性。
   pidfile     /var/run/haproxy.pid     # 将 HAProxy 进程的 PID 写入 pidfile。
   maxconn     4096                     # 单个 HAProxy 进程可接受的最大并发连接数，
       ↪ 等价于命令行参数 "-n"。
   nbthread    48                       # 最大线程数。线程数的上限与 CPU 数量相同。
   user        haproxy                  # 同 UID 参数。
   group       haproxy                  # 同 GID 参数，建议使用专用用户组。
   daemon                               # 让 HAProxy 以守护进程的方式工作于后台，
       ↪ 等同于命令行参数“-D”的功能。当然，也可以在命令行中用“-db”参数将其禁用。
   stats socket /var/lib/haproxy/stats  # 统计信息保存位置。

defaults                                # 默认配置。
   log global                           # 日志继承全局配置段的设置。
   retries 2                            # 向上游服务器尝试连接的最大次数，
       ↪ 超过此值便认为后端服务器不可用。
   timeout connect  2s                  # HAProxy 与后端服务器连接超时时间。
       ↪ 如果在同一个局域网内，可设置成较短的时间。
   timeout client 30000s                # 客户端与 HAProxy 连接后，数据传输完毕，
       ↪ 即非活动连接的超时时间。
   timeout server 30000s                # 服务器端非活动连接的超时时间。

listen admin_stats                      # frontend 和 backend 的组合体，
    ↪ 此监控组的名称可按需进行自定义。
   bind 0.0.0.0:8080                    # 监听端口。
   mode http                            # 监控运行的模式，此处为 `http` 模式。
   option httplog                       # 开始启用记录 HTTP 请求的日志功能。
   maxconn 10                           # 最大并发连接数。
   stats refresh 30s                    # 每隔 30 秒自动刷新监控页面。
```

```
   stats uri /haproxy                     # 监控页面的 URL。
   stats realm HAProxy                    # 监控页面的提示信息。
   stats auth admin:pingcap123            # 监控页面的用户和密码，可设置多个用户名。
   stats hide-version                     # 隐藏监控页面上的 HAProxy 版本信息。
   stats  admin if TRUE                   # 手工启用或禁用后端服务器（HAProxy 1.4.9
       ↪ 及之后版本开始支持）。

listen tidb-cluster                       # 配置 database 负载均衡。
   bind 0.0.0.0:3390                      # 浮动 IP 和 监听端口。
   mode tcp                               # HAProxy 要使用第 4 层的传输层。
   balance leastconn                      # 连接数最少的服务器优先接收连接。`leastconn`
       ↪ 建议用于长会话服务，例如 LDAP、SQL、TSE 等，而不是短会话协议，如 HTTP。该算法是动态的
       ↪ ，对于启动慢的服务器，服务器权重会在运行中作调整。
   server tidb-1 10.9.18.229:4000 check inter 2000 rise 2 fall 3       # 检测 4000 端口，
       ↪ 检测频率为每 2000 毫秒一次。如果 2 次检测为成功，则认为服务器可用；如果 3 次检测为失败
       ↪ ，则认为服务器不可用。
   server tidb-2 10.9.39.208:4000 check inter 2000 rise 2 fall 3
   server tidb-3 10.9.64.166:4000 check inter 2000 rise 2 fall 3
```

12.4.4.3 启动 HAProxy

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

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

12.4.4.4 停止 HAProxy

要停止 HAProxy，使用 kill -9 命令。

1. 执行如下命令：

   ```
   ps -ef | grep haproxy
   ```

2. 终止 HAProxy 相关的 PID 进程：

   ```
   kill -9 ${haproxy.pid}
   ```

12.5 性能调优

12.5.1 优化手册

12.5.1.1 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。

12.5.1.1.1 用户响应时间和数据库时间

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。

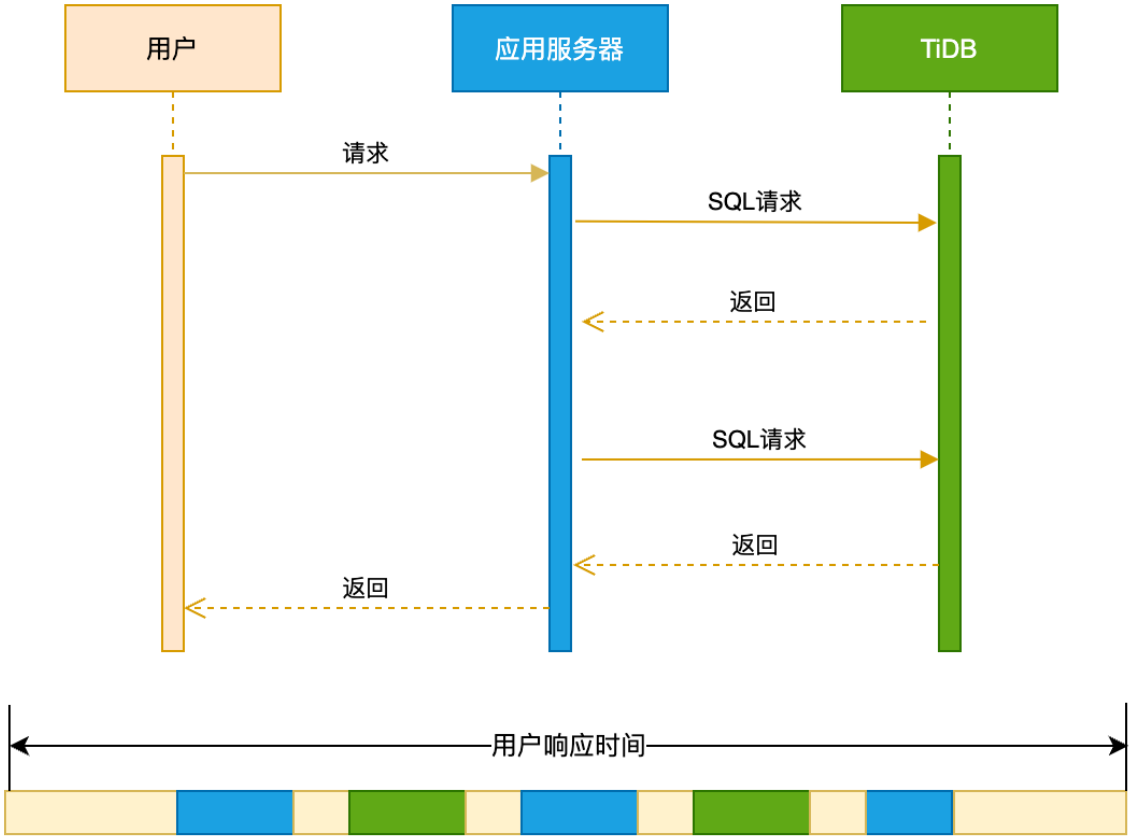


图 314: 用户响应时间

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_handle_query_duration_seconds_sum 计算，即 `ΔT DB Time = rate(TiDB_server_handle_query_duration_seconds_sum)× ΔT`

12.5.1.1.2 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。

12.5.1.1.3 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。

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。

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

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。

实现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。

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。

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。

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。

12.5.1.2 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。

12.5.1.2.1 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。

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。

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。

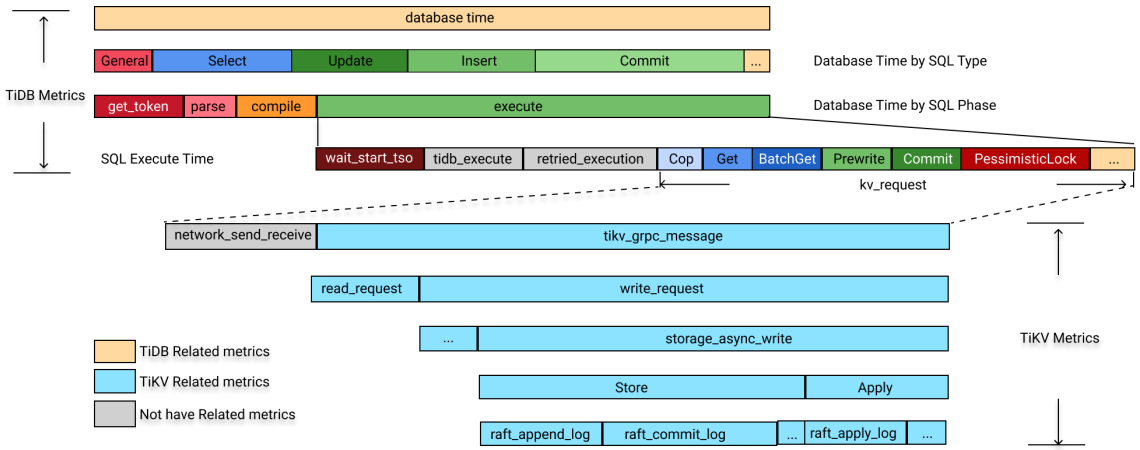


图 315: 数据库时间分解图

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
  ```

12.5.1.2.2 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。

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），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

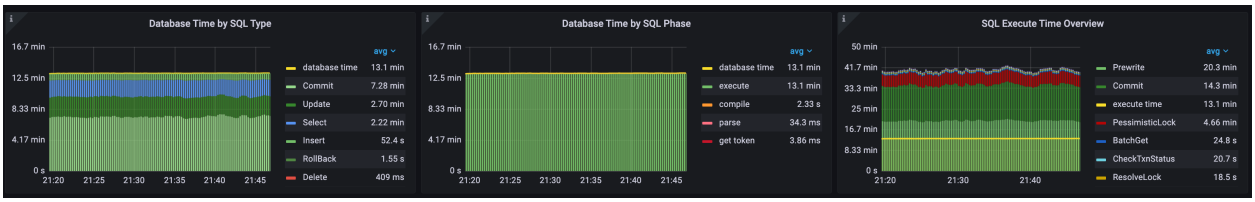


图 316: TPC-C

- 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 PessimisticLock 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

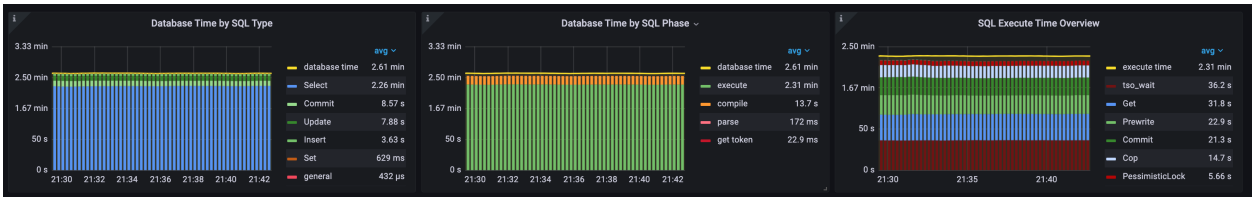


图 317: OLTP

- 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

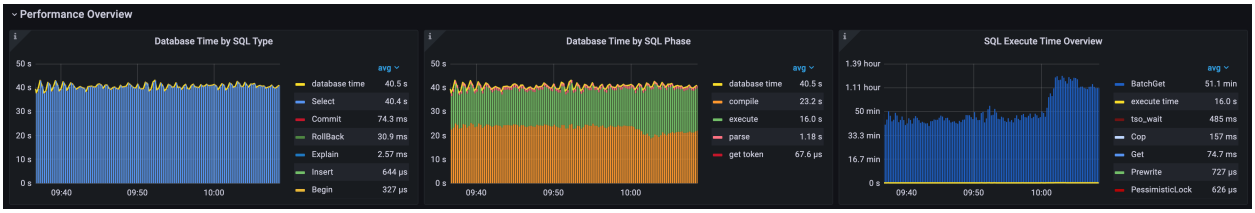


图 318: OLTP

- 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

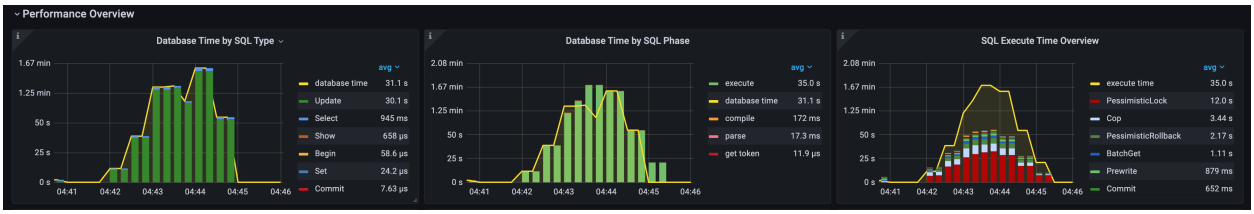


图 319: OLTP

- 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。

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 集群每秒命中执行计划缓存的次数。执行计划缓存只支持 prepared statement 命令。TiDB 开启执行计划缓存的情况下，存在三种使用情况：
  - 完全无法命中执行计划缓存：每秒命中次数为 0，因为应用使用 query 命令，或者每次 StmtExecute 执行之后调用 StmtClose 命令，导致缓存的执行计划被清理。
  - 完全命中执行计划缓存：每秒命中次数等于 StmtExecute 命令每秒执行次数。
  - 部分命中执行计划缓存：每秒命中次数小于 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。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。

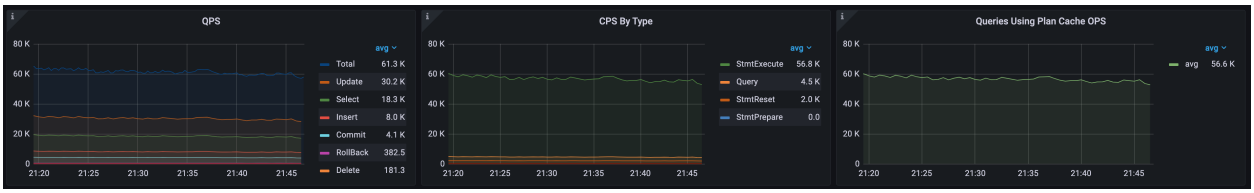


图 320: TPC-C

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。

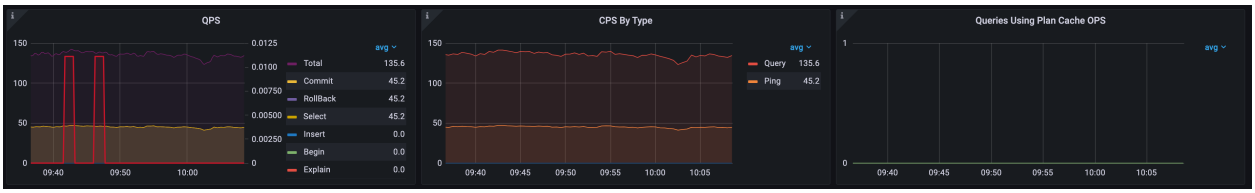


图 321: OLTP-Query

- 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。

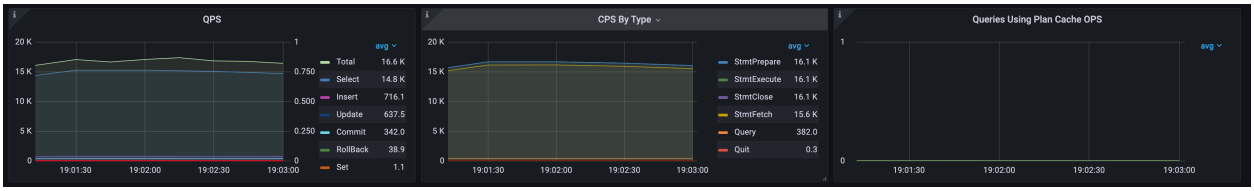


图 322: OLTP-Prepared

KV/TSO Request OPS 和连接信息

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 Connection Count （连接信息）面板中，你可以查看总的连接数和每个 TiDB 的连接数，并由此判断连接总数是否正常，每个 TiDB 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。

示例 1：繁忙的负载



图 323: TPC-C

在此 TPC-C 负载中：

- 每秒总的 KV 请求的数量为 104.2k。按请求数量排序，最高的请求类型为 `PessimisticLock`、`Prewrite`、`Commit` 和 `BatchGet` 等。
- 总的连接数为 810，三个 TiDB 实例的连接数大体均衡。活跃的连接数为 787.1。对比活跃连接数和总连接数，97% 的连接都是活跃的，通常意味着数据库是这个应用系统的性能瓶颈。

示例 2：空闲的负载



图 324: OLTP

在此负载中：

- 每秒总的 KV 请求数据是 2.6 K，TSO 请求次数是每秒 1.1 K。
- 总的连接数为 410，连接数在三个 TiDB 中相对均衡。活跃连接数只有 2.5，说明数据库系统非常空闲。

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 325: 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。

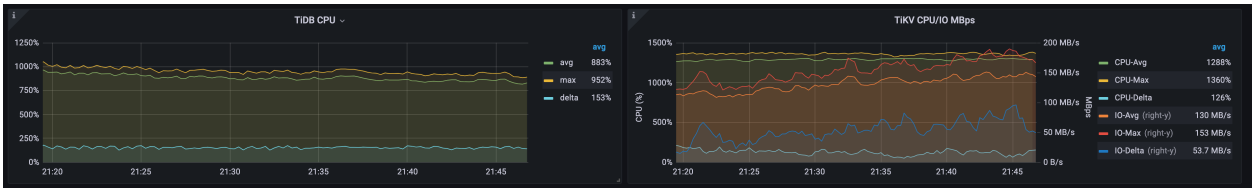


图 326: TPC-C

- 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。

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 和 Connection Idle Duration

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。

示例 1：用户响应时间的瓶颈在 TiDB 中

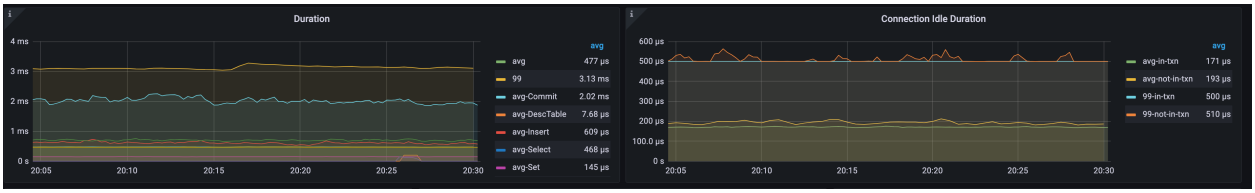


图 327: TiDB is the Bottleneck

在此 TPC-C 负载中：

- 所有 SQL 语句的平均延迟 477 us，99 延迟 3.13ms。平均 commit 语句 2.02ms，平均 insert 语句 609ms，平均查询语句 468us。
- 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 2：用户响应时间的瓶颈不在 TiDB 中

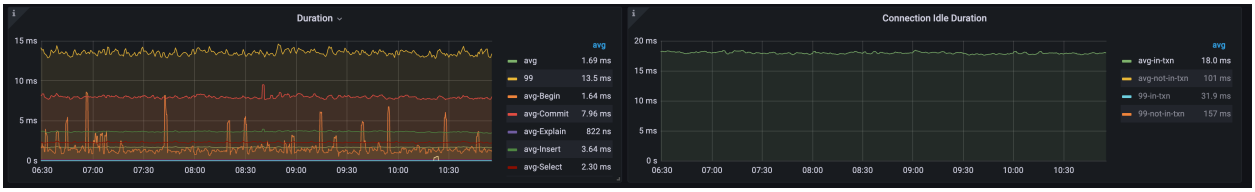


图 328: TiDB is the Bottleneck

在此负载中，平均 query 延迟为 1.69ms，事务中连接空闲时间 `avg-in-txn` 为 18ms。说明事务中，TiDB 平均花了 1.69ms 处理完一个 SQL 语句之后，需要等待 18ms 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，执行计划缓扮演了重要的角色。

示例 1：数据库瓶颈在 compile 阶段

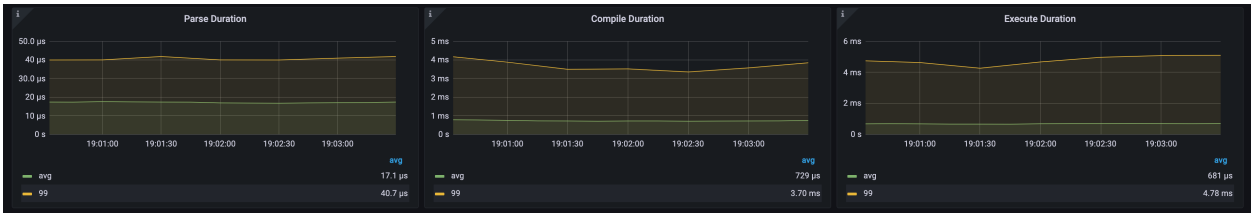


图 329: Compile

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

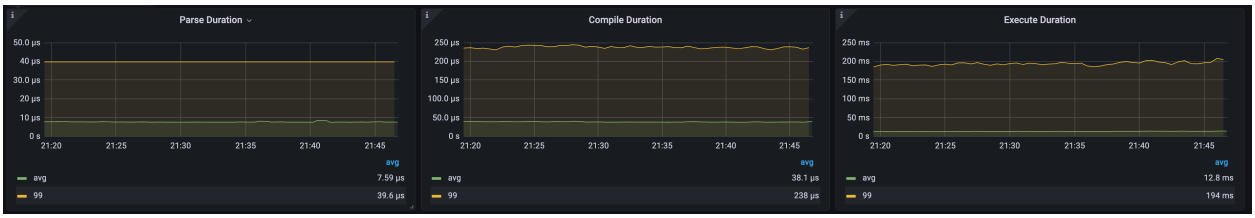


图 330: Execute

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

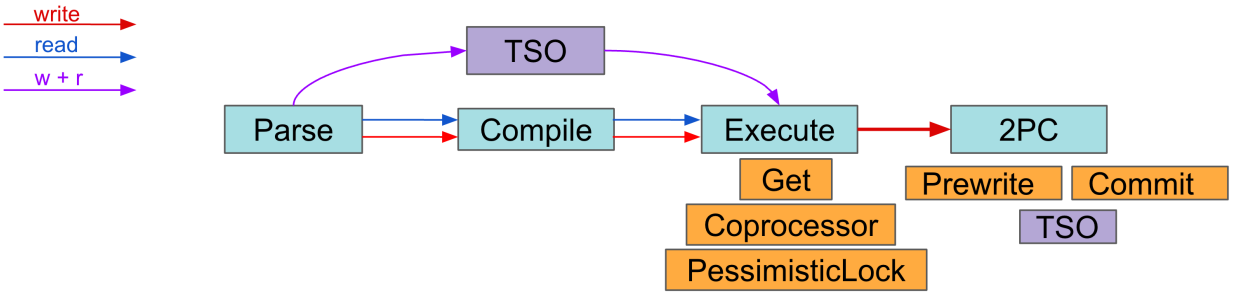


图 331: Execute

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

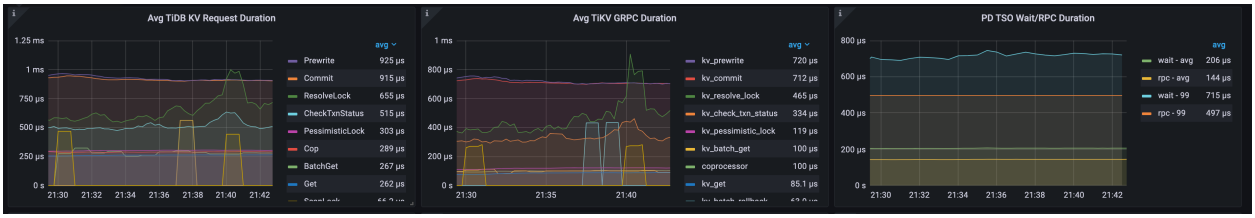


图 332: Same Data Center

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

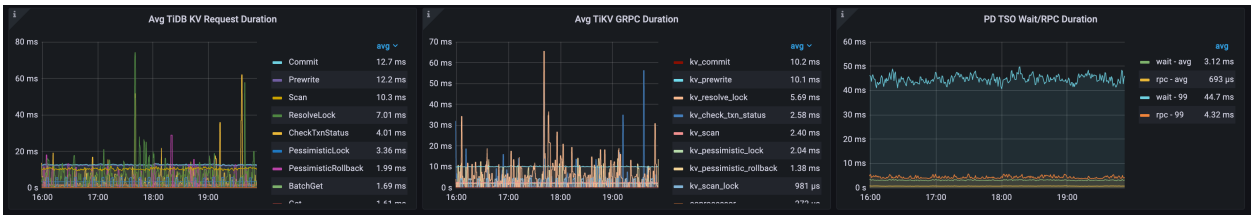


图 333: Cloud Env

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 10.2ms，相差 2.5ms 左右。TSO wait 平均延迟为 3.12ms，rpc 时间为 693us。

示例 3：公有云集群，资源严重过载

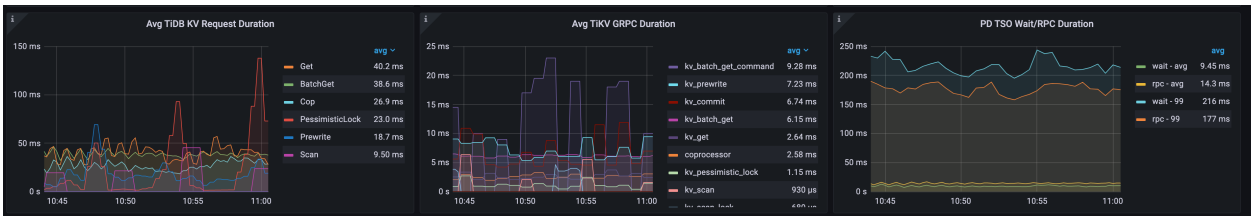


图 334: Cloud Env, TiDB Overloaded

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 9.45ms，rpc 时间为 14.3ms。

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。

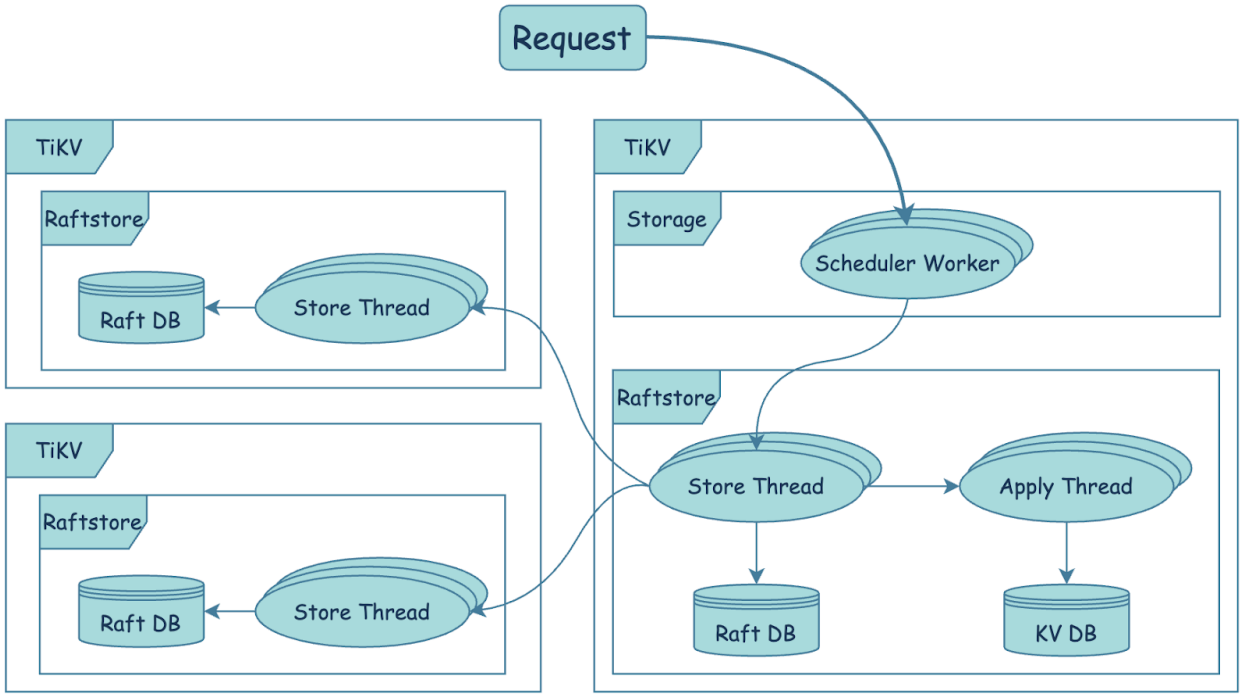


图 335: TiKV Write

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：

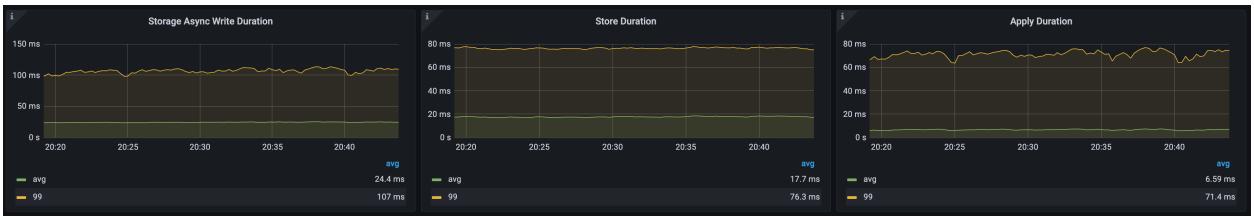


图 336: v5.3.0

v5.4.0：



图 337: 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。

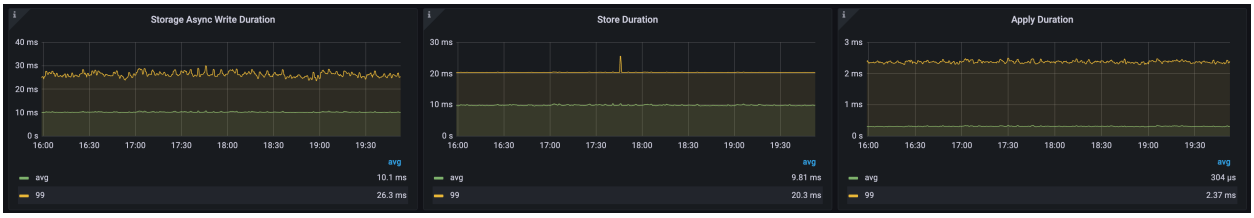


图 338: Store

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：

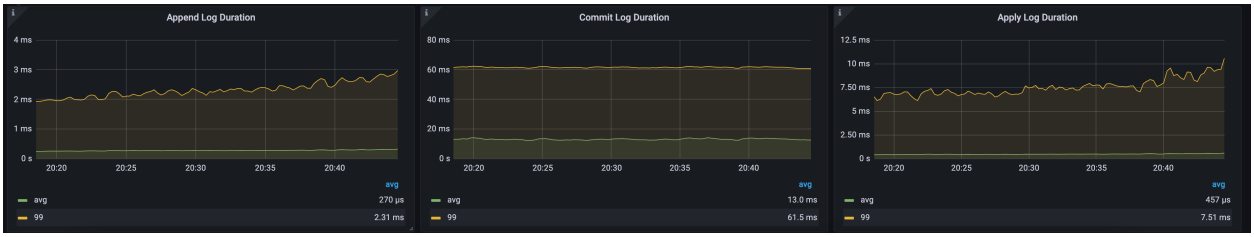


图 339: v5.3.0

v5.4.0：

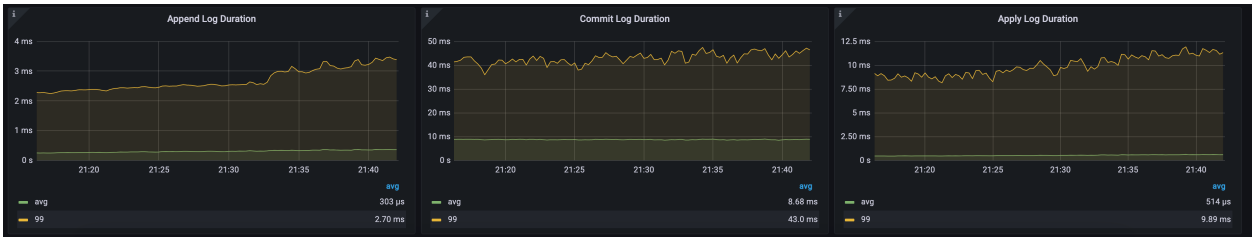


图 340: v5.4.0

示例 2：Commit Log Duration 瓶颈明显的例子

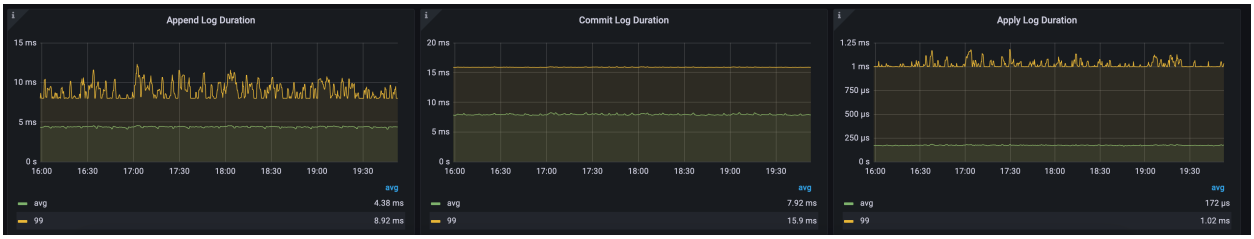


图 341: Store

- 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`。

12.5.1.2.3 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：

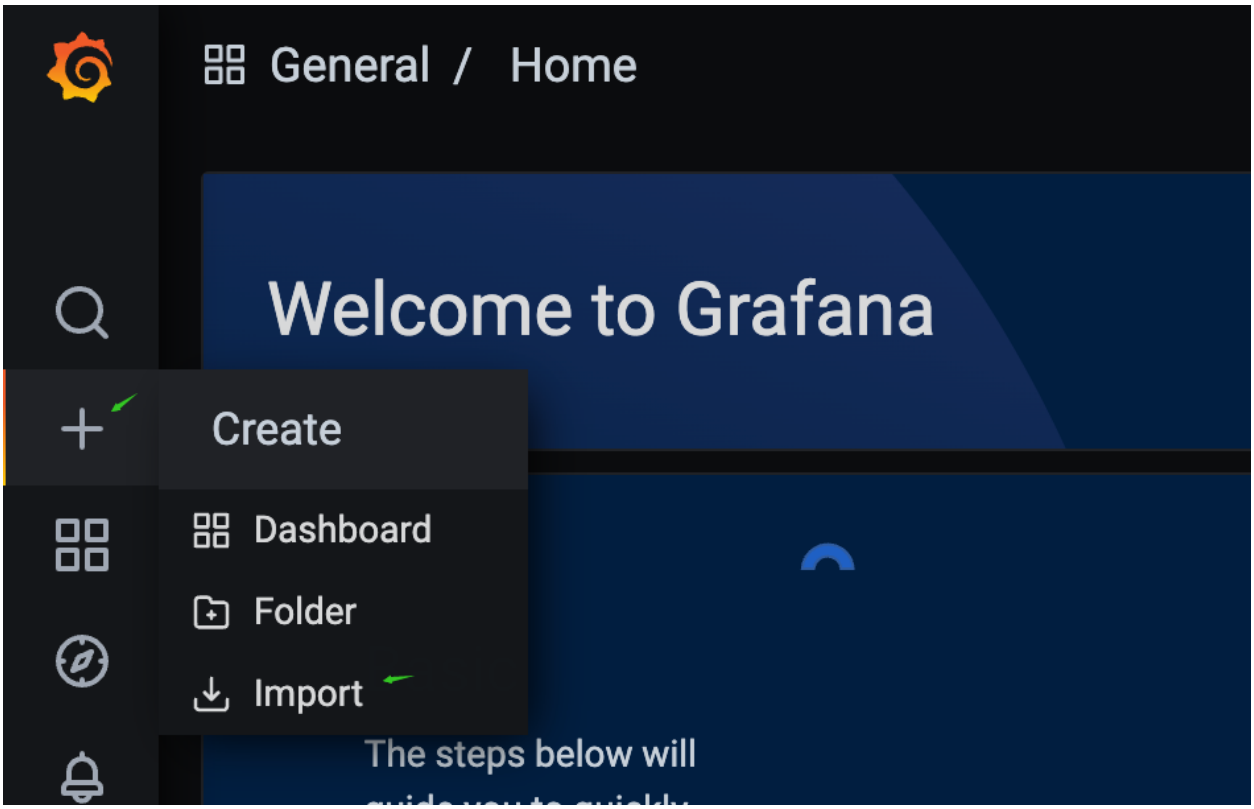


图 342: Store

12.5.1.3 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。

12.5.1.3.1 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。

12.5.1.3.2 场景 1：使用 Query 接口

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
```

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 343: 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。

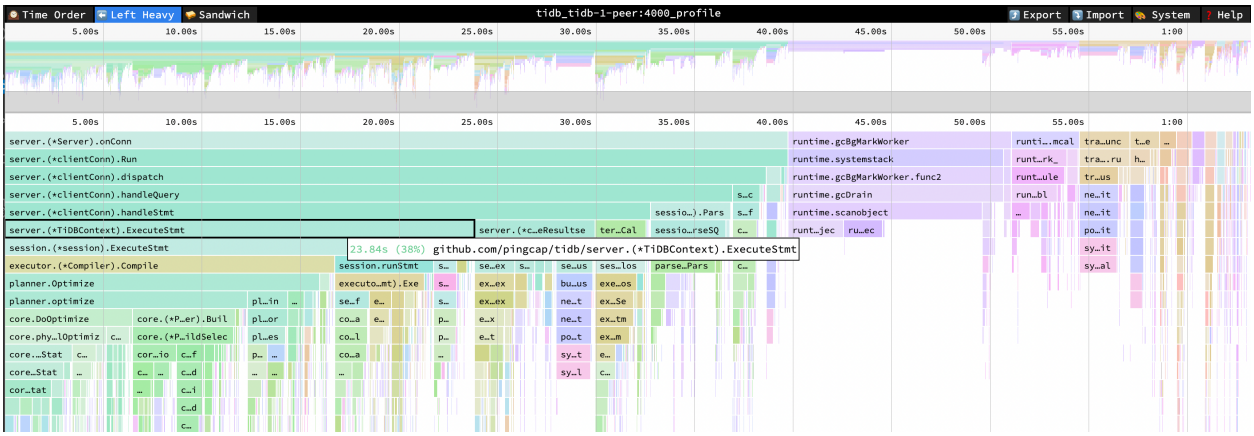


图 344: flame-graph-for-query-interface

- 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：

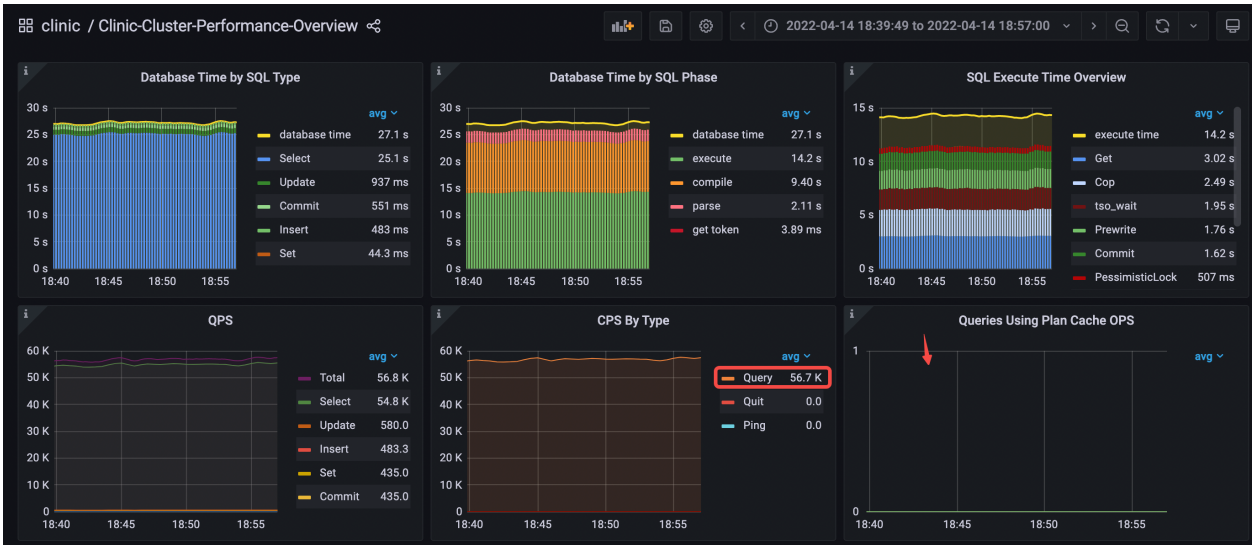


图 345: performance-overview-1-for-query-interface

- 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。

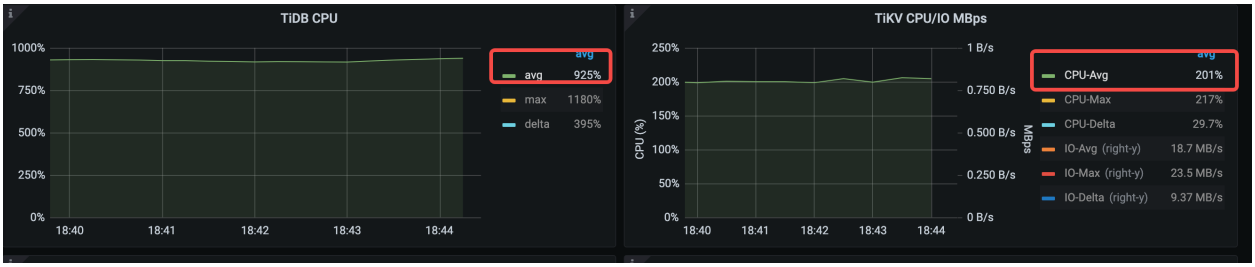


图 346: performance-overview-2-for-query-interface

分析结论

需要屏蔽这些大量无用的非业务 SQL 语句。

12.5.1.3.3 场景 2：使用 maxPerformance 配置

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
```

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。

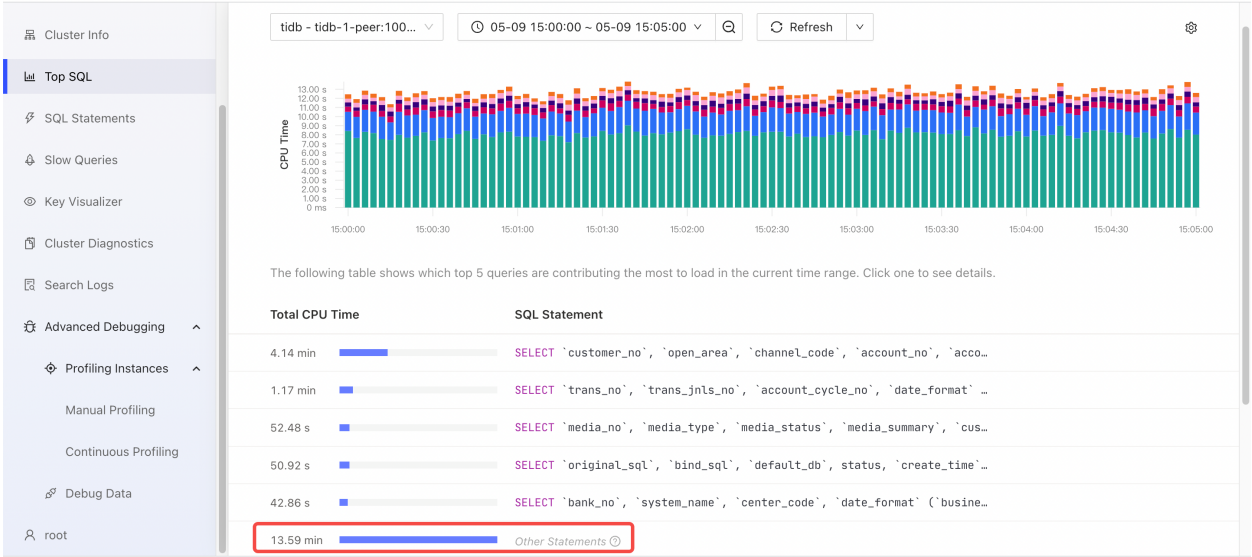


图 347: dashboard-for-maxPerformance

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：

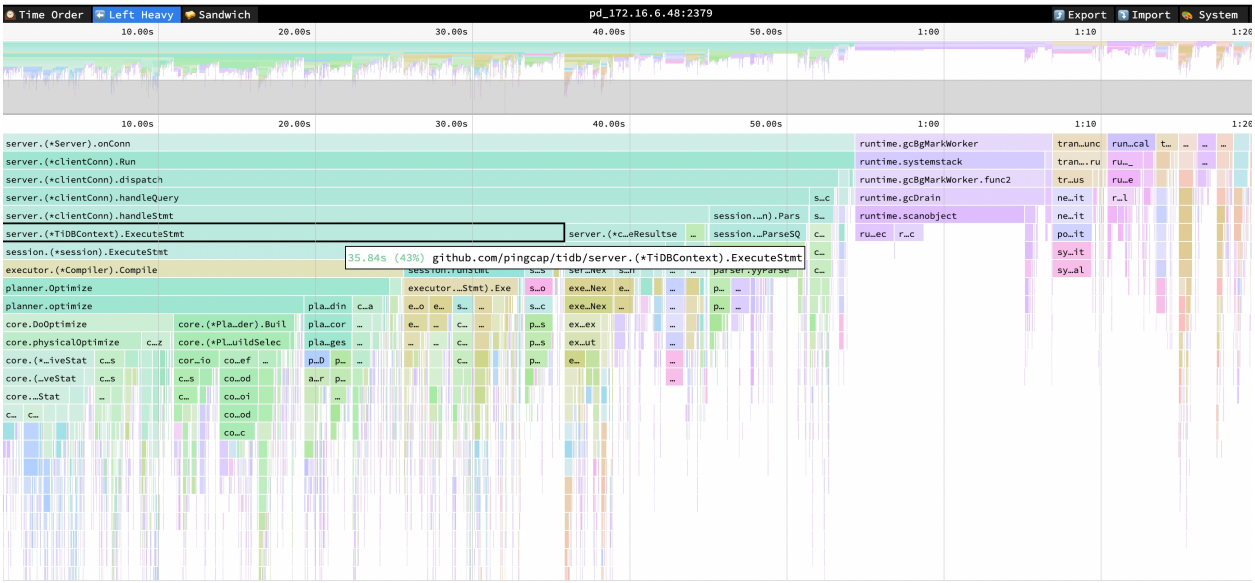


图 348: flame-graph-for-maxPerformance

- 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：

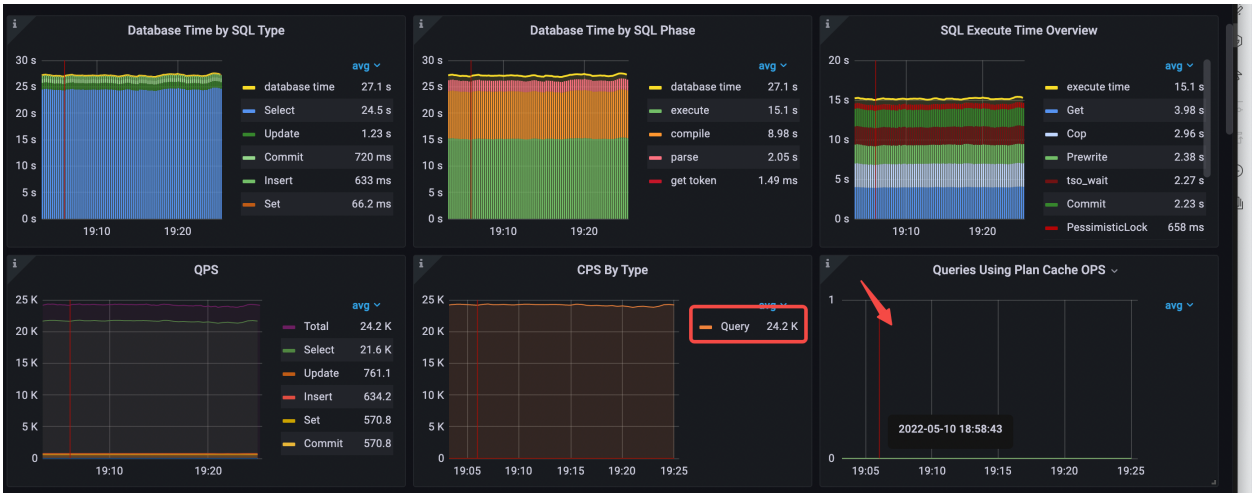


图 349: performance-overview-1-for-maxPerformance

- 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。

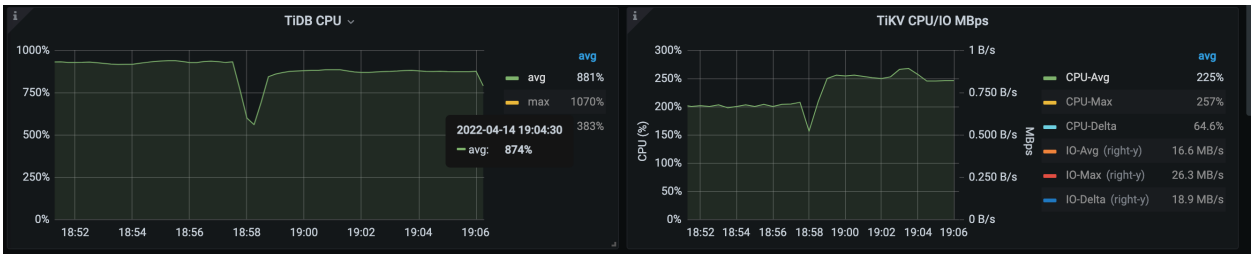


图 350: performance-overview-2-for-maxPerformance

关键延迟指标变化如下：



图 351: 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)

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 TiDB CPU 消耗，降低延迟。

12.5.1.3.4 场景 3：使用 Prepared Statement 接口，未开启执行计划缓存

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"
```

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。

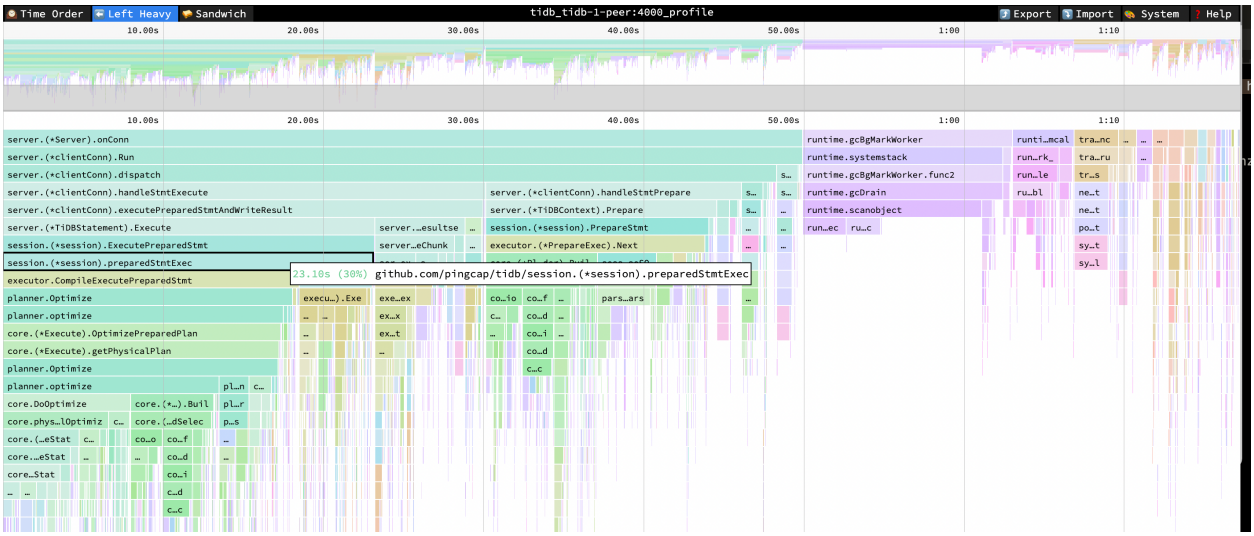


图 352: flame-graph-for-PrepStmts

- 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：

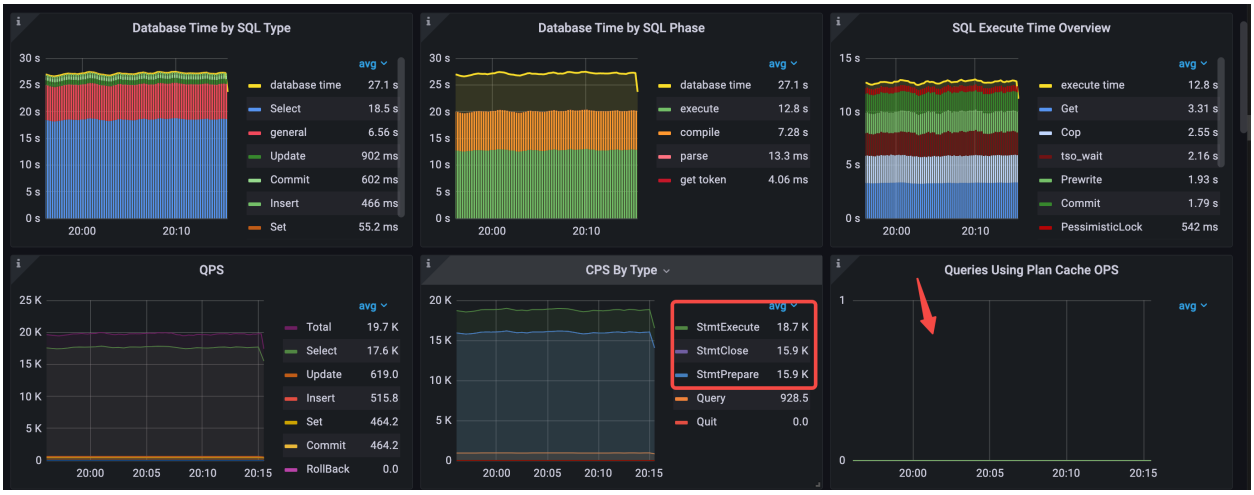


图 353: performance-overview-1-for-PrepStmts

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 354: performance-overview-1-for-PrepStmts

主要延迟数据如下：

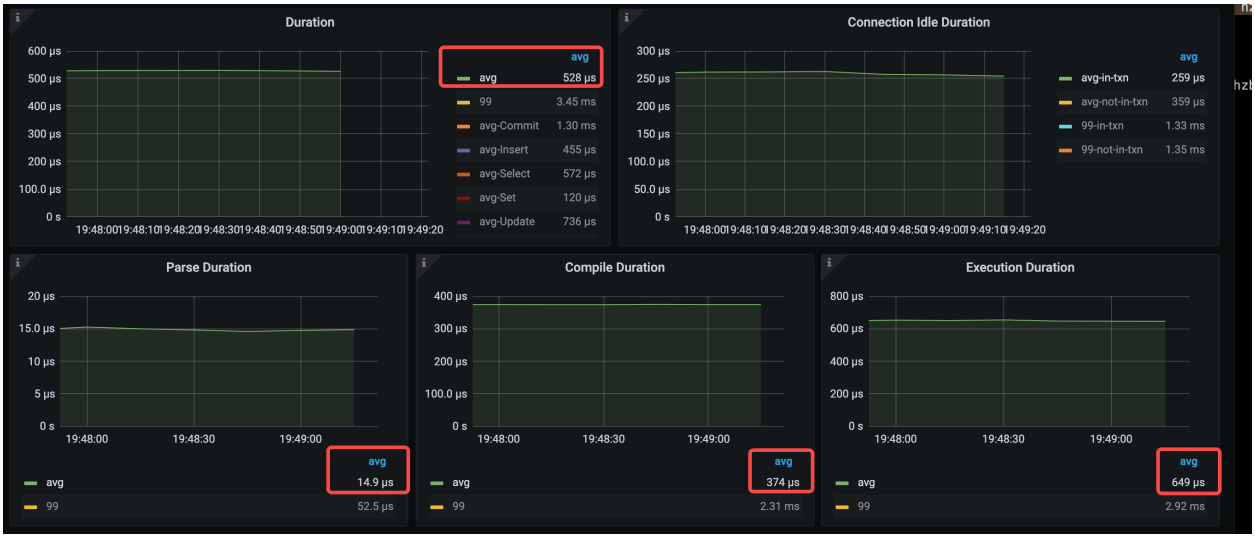


图 355: performance-overview-2-for-PrepStmts

- 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)

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。

12.5.1.3.5 场景 4：使用 Prepared Statement 接口，开启执行计划缓存

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

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 356: 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。

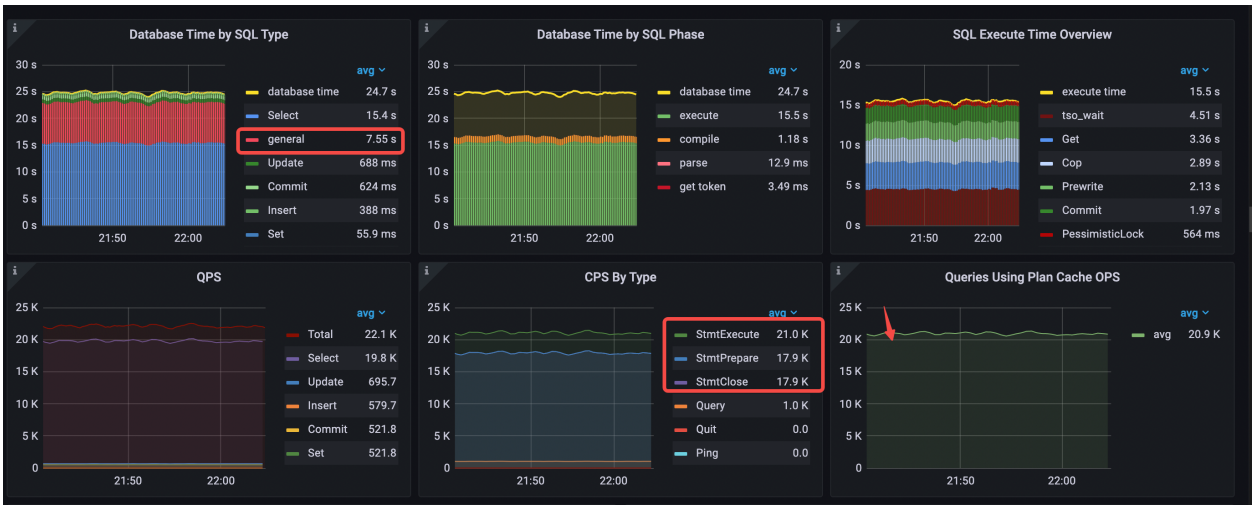


图 357: performance-overview-1-for-3-commands

- 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%。

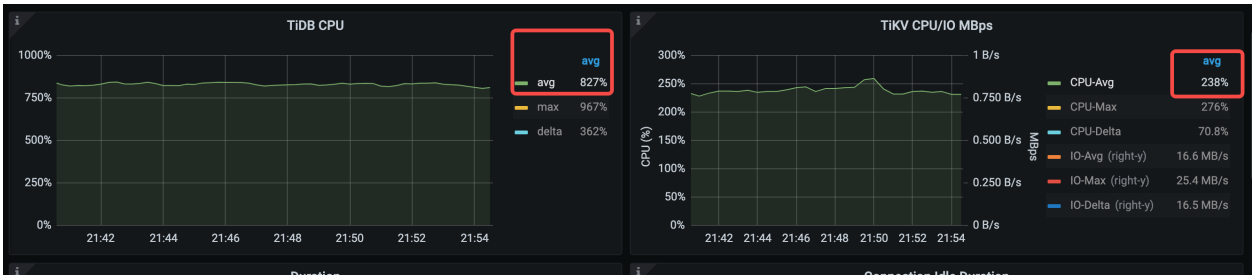


图 358: performance-overview-2-for-3-commands

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。

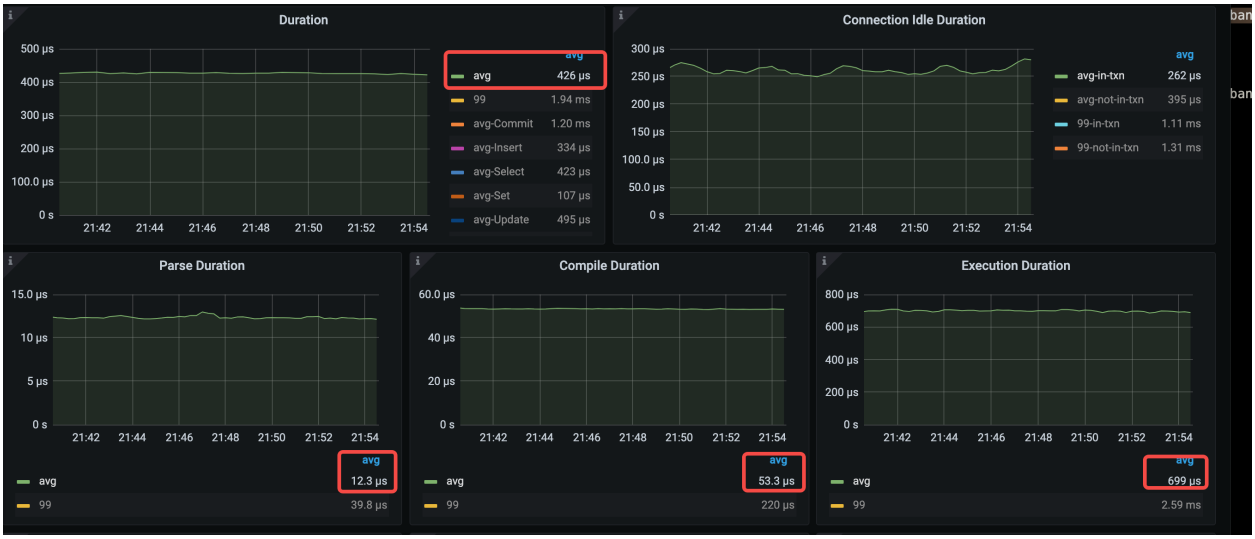


图 359: performance-overview-3-for-3-commands

- 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)

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。

12.5.1.3.6 场景 5：客户端缓存 prepared 对象

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
```

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 360: 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，QSP 面板中 QPS 上升到了 30.9k。

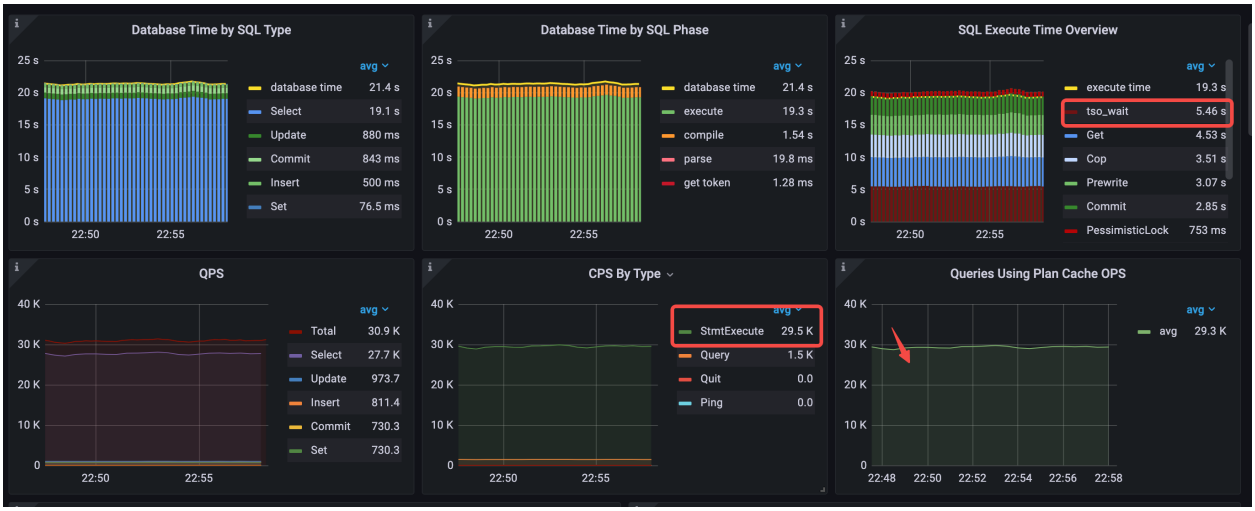


图 361: performance-overview-for-1-command

- 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 362: performance-overview-for-2-command

关键的延迟指标如下：

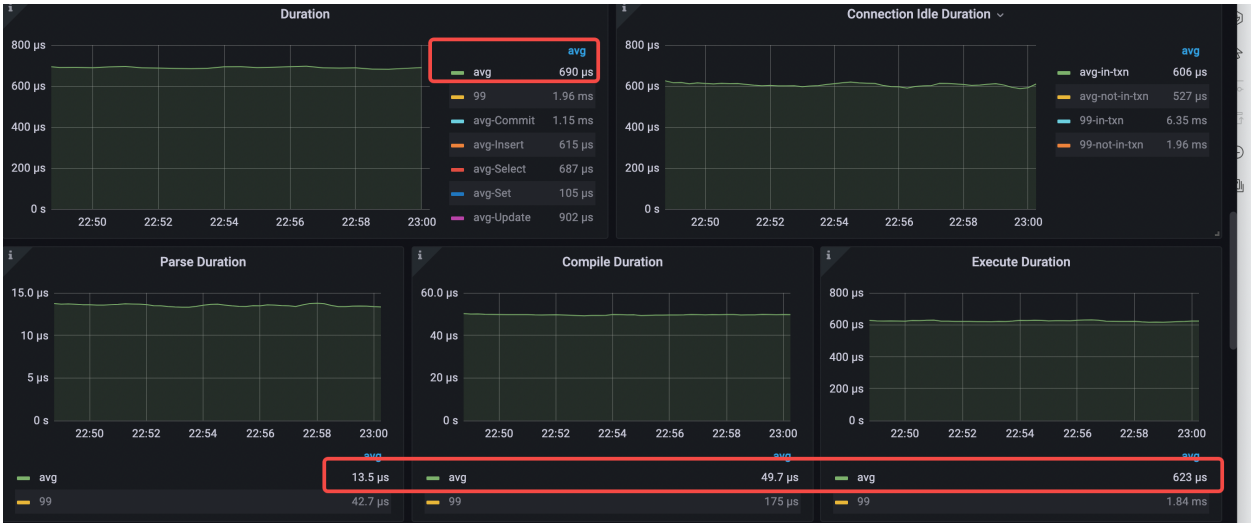


图 363: performance-overview-for-3-command

- 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)

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。

12.5.1.3.7 场景 6：开启 tidb_rc_read_check_ts 变量降低 TSO 请求

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。

性能分析

Dashboard

TiDB 的 CPU 火焰图没有明显变化。

- ExecutePreparedStmt cpu = 22% cpu time = 8.4s

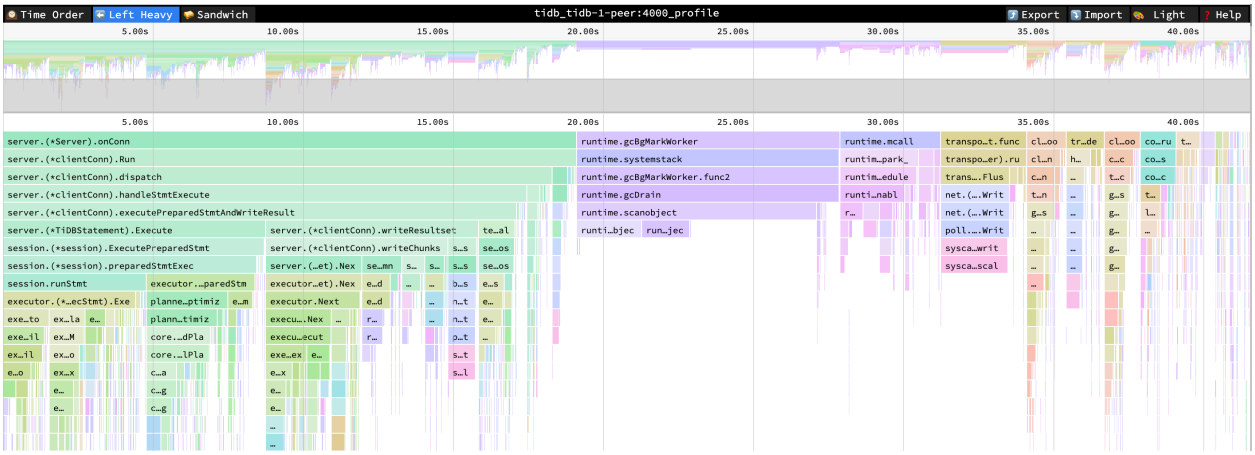


图 364: flame-graph-for-rc-read

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。

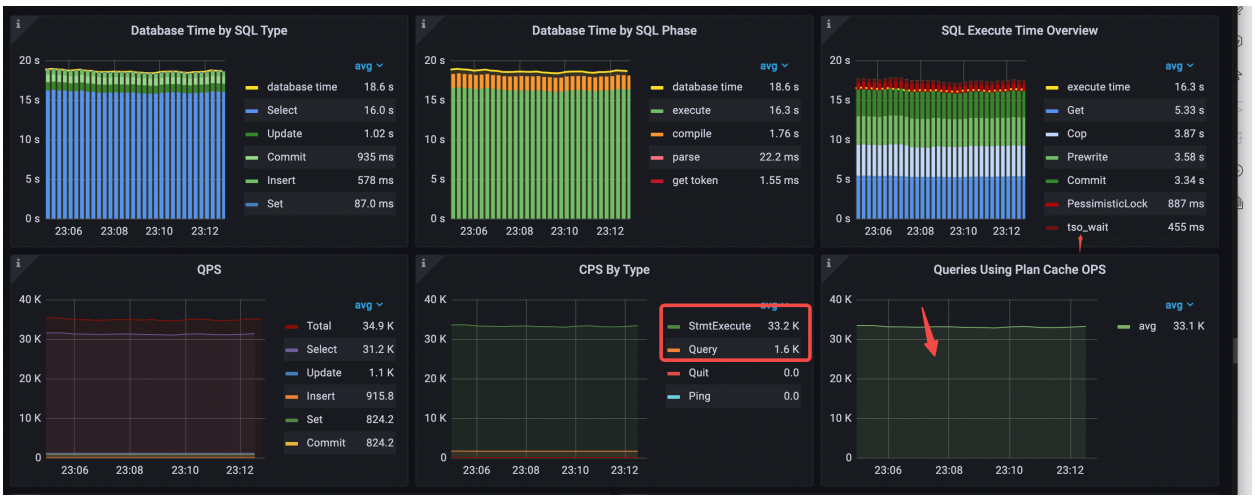


图 365: performance-overview-1-for-rc-read

- 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。

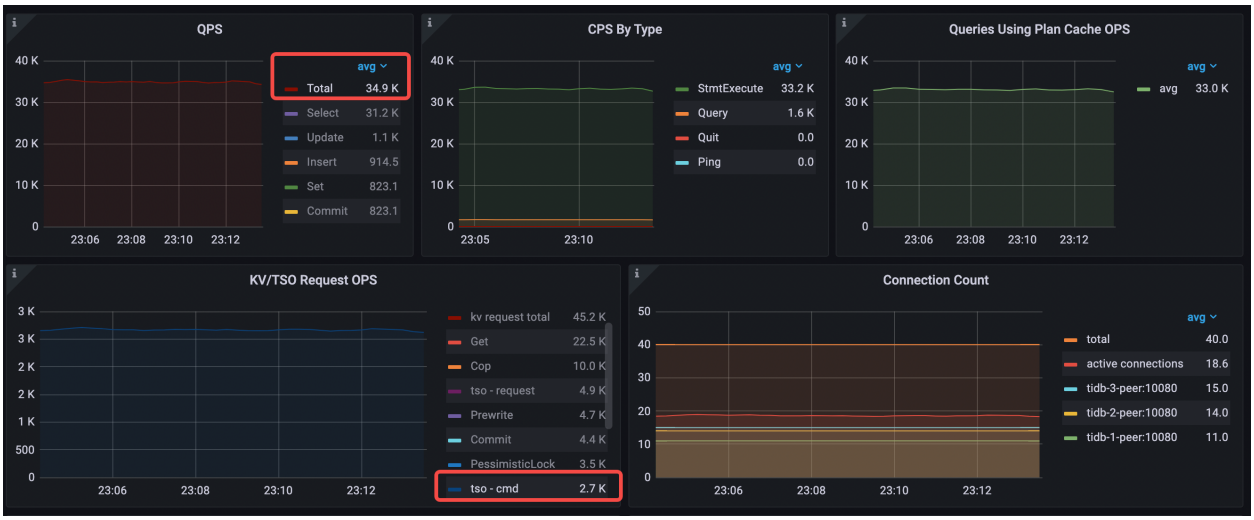


图 366: performance-overview-2-for-rc-read

平均 TiDB CPU 上升为 603% (577%->603%)。



图 367: performance-overview-3-for-rc-read

关键延迟指标如下：

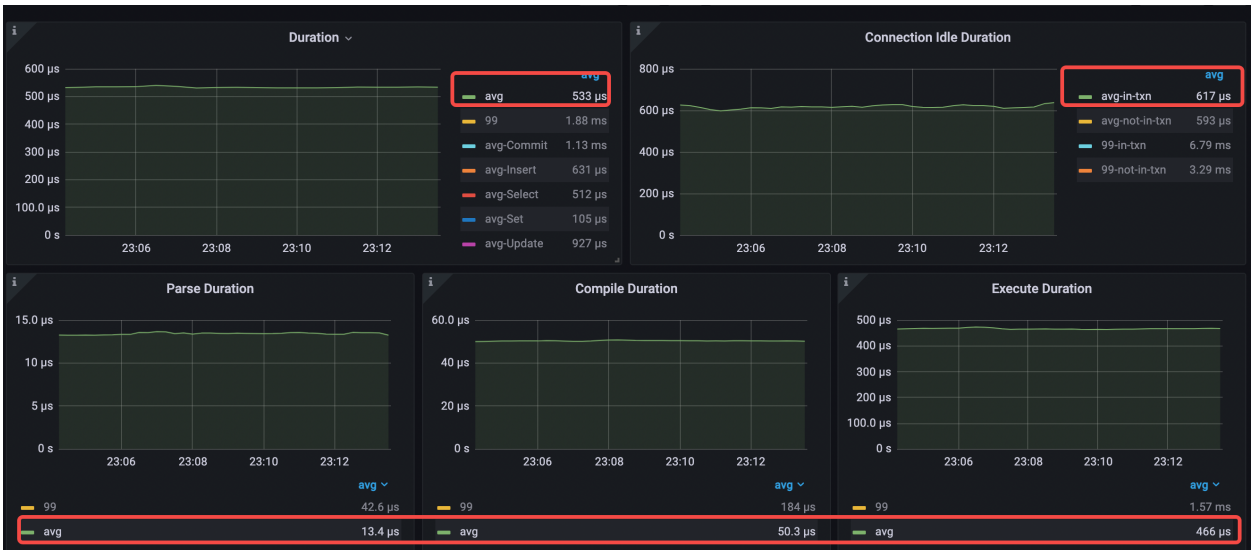


图 368: performance-overview-4-for-rc-read

- 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)

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。

12.5.1.3.8 场景 7：使用小表缓存

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;`。

性能分析

Dashboard

TiDB CPU 火焰图没有明显变化。

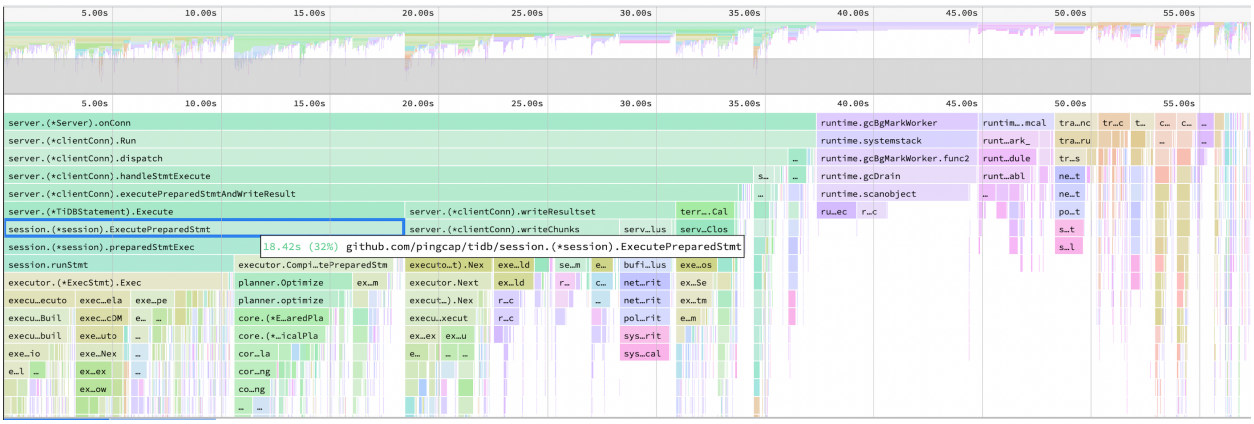


图 369: flame-graph-for-table-cache

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。

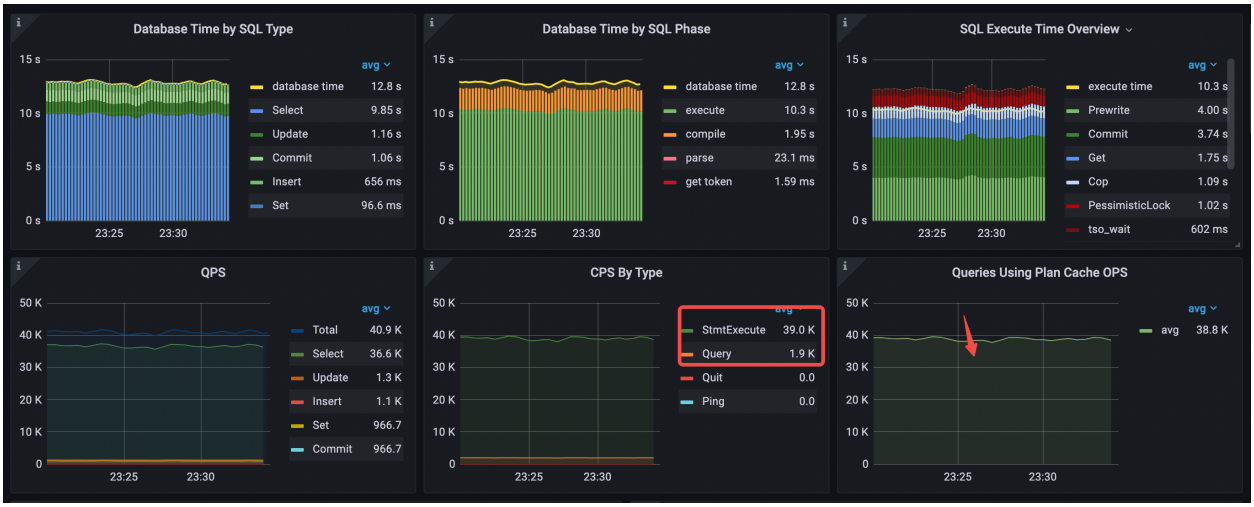


图 370: performance-overview-1-for-table-cache

- 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 371: 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。

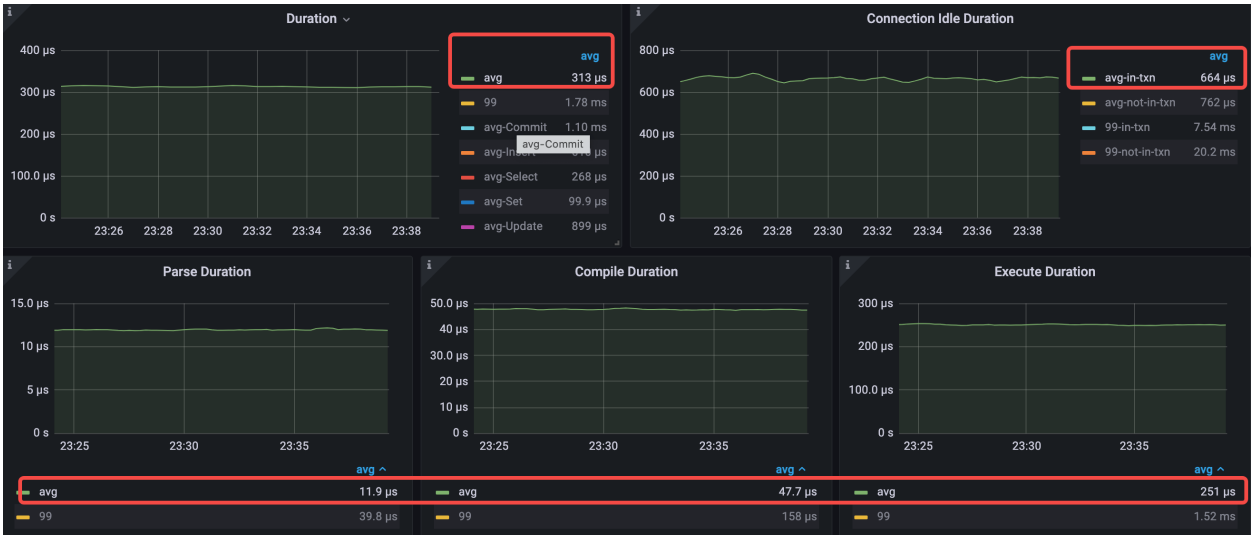


图 372: performance-overview-3-for-table-cache

- 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)

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。

12.5.1.3.9 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。

12.5.2 软件调优

12.5.2.1 TiDB 内存控制文档

目前 TiDB 已经能够做到追踪单条 SQL 查询过程中的内存使用情况，当内存使用超过一定阈值后也能采取一些操作来预防 OOM 或者排查 OOM 原因。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。

12.5.2.1.1 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;
```

12.5.2.1.2 如何配置 tidb-server 实例使用内存的阈值

可以在配置文件中设置 tidb-server 实例的内存使用阈值。相关配置项为server-memory-quota 。

例如，配置 tidb-server 实例的内存使用总量，将其设置成为 32 GB：

```
[performance]
server-memory-quota = 34359738368
```

在该配置下，当 tidb-server 实例内存使用到达 32 GB 时，正在执行的 SQL 语句会被随机强制终止，直至 tidb-server 实例内存使用下降到 32 GB 以下。被强制终止的 SQL 操作会向客户端返回 Out Of Global Memory Limit! 错误信息。

> 警告：
>
> - server-memory-quota 目前为实验性特性，不建议在生产环境中使用。
> - server-memory-quota 默认值为 0，表示无内存限制。

12.5.2.1.3 tidb-server 内存占用过高时的报警

默认配置下，tidb-server 实例会在机器内存使用达到总内存量的 80% 时打印报警日志，并记录相关状态文件。该内存使用率可以通过配置项memory-usage-alarm-ratio 进行设置。具体报警规则请参考该配置项的说明部分。

注意，当触发一次报警后，只有在内存使用率连续低于阈值超过 10 秒并再次达到阈值时，才会再次触发报警。此外，为避免报警时产生的状态文件积累过多，目前只会保留最近 5 次报警时所生成的状态文件。

下例通过构造一个占用大量内存的 SQL 语句触发报警，对该报警功能进行演示：

1. 配置报警比例为 0.8：

```
mem-quota-query = 34359738368
[performance]
memory-usage-alarm-ratio = 0.8
```

2. 创建单表 CREATE TABLE t(a int); 并插入 1000 行数据。

3. 执行 select * from t t1 join t 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。

```
[2020/11/30 15:25:17.252 +08:00] [WARN] [memory_usage_alarm.go:141] ["tidb-server has the
    ↪ risk of OOM. Running SQLs and heap profile will be recorded in record path"] ["is
    ↪ server-memory-quota set"=false] ["system memory total"=33682427904] ["system memory
    ↪ usage"=27142864896] ["tidb-server memory usage"=22417922896] [memory-usage-alarm-
    ↪ ratio=0.8] ["record path"="/tmp/1000_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-
    ↪ storage/record"]
```

   以上 Log 字段的含义如下：

   - is server-memory-quota set：表示配置项server-memory-quota 是否被设置
   - system memory total：表示当前系统的总内存
   - system memory usage：表示当前系统的内存使用量
   - tidb-server memory usage：表示 tidb-server 实例的内存使用量
   - memory-usage-alarm-ratio：表示配置项memory-usage-alarm-ratio 的值
   - record path：表示状态文件存放的目录

5. 通过访问状态文件所在目录 (该示例中的目录为 /tmp/1000_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA
   ↪ =/tmp-storage/record)，可以得到一组文件，其中包括 goroutinue、heap、running_sql 3 个文件，文件以记录状态文件的时间为后缀。这 3 个文件分别用来记录报警时的 goroutine 栈信息，堆内存使用状态，及正在运行的 SQL 信息。其中 running_sql 文件内的日志格式请参考 expensive-queries。

12.5.2.1.4 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`、`oom-use-tmp-storage`、`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)
```

12.5.2.2 TiKV 线程池性能调优

本文主要介绍 TiKV 线程池性能调优的主要手段，以及 TiKV 内部线程池的主要用途。

12.5.2.2.1 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。

12.5.2.2.2 TiKV 的只读请求

TiKV 的读取请求分为两类：

- 一类是指定查询某一行或者某几行的简单查询，这类查询会运行在 Storage Read Pool 中。
- 另一类是复杂的聚合计算、范围查询，这类请求会运行在 Coprocessor Read Pool 中。

从 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。

12.5.2.2.3 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）。

- 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
```

12.5.2.3 TiKV 内存参数性能调优

本文档用于描述如何根据机器配置情况来调整 TiKV 的参数，使 TiKV 的性能达到最优。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。如果要为每个 CF 使用单独的 block cache 实例，需要在 [storage.block-cache] 下设置 shared=false，并为每个 CF 配置单独的 block cache 大小。例如，可以在 [rocksdb.writecf] 下设置 block-cache-size 参数来配置 write CF 的大小。

> **注意：**
>
> 在 TiKV 3.0 之前的版本中，不支持使用 shared block cache，需要为每个 CF 单独配置 block cache。

每个 CF 有各自的 write-buffer，大小通过 write-buffer-size 控制。

12.5.2.3.1 参数说明

```
#### 日志级别，可选值为：trace，debug，warn，error，info，off
log-level = "info"

[server]
#### 监听地址
#### addr = "127.0.0.1:20160"

#### gRPC 线程池大小
#### grpc-concurrency = 4
#### TiKV 每个实例之间的 gRPC 连接数
#### grpc-raft-conn-num = 10

#### TiDB 过来的大部分读请求都会发送到 TiKV 的 Coprocessor 进行处理，该参数用于设置
#### coprocessor 线程的个数，如果业务是读请求比较多，增加 coprocessor 的线程数，但应比系统的
```

```
#### CPU 核数小。例如：TiKV 所在的机器有 32 core，在重读的场景下甚至可以将该参数设置为 30。在没有
#### 设置该参数的情况下，TiKV 会自动将该值设置为 CPU 总核数乘以 0.8。
#### end-point-concurrency = 8

#### 可以给 TiKV 实例打标签，用于副本的调度
#### labels = {zone = "cn-east-1", host = "118", disk = "ssd"}

[storage]
#### 数据目录
#### data-dir = "/tmp/tikv/store"

#### 通常情况下使用默认值就可以了。在导数据的情况下建议将该参数设置为 1024000。
#### scheduler-concurrency = 102400
#### 该参数控制写入线程的个数，当写入操作比较频繁的时候，需要把该参数调大。使用 top -H -p tikv-
    ↪ pid
#### 发现名称为 sched-worker-pool 的线程都特别忙，这个时候就需要将 scheduler-worker-pool-size
#### 参数调大，增加写线程的个数。
#### scheduler-worker-pool-size = 4

[storage.block-cache]
##### 是否为 RocksDB 的所有 CF 都创建一个 `shared block cache`。
##### ## RocksDB 使用 block cache 来缓存未压缩的数据块。较大的 block cache 可以加快读取速度。
##### 推荐开启 `shared block cache` 参数。这样只需要设置全部缓存大小，使配置过程更加方便。
##### 在大多数情况下，可以通过 LRU 算法在各 CF 间自动平衡缓存用量。
##### ## `storage.block-cache` 会话中的其余配置仅在开启 `shared block cache` 时起作用。
#### shared = true
##### `shared block cache` 的大小。正常情况下应设置为系统全部内存的 30%-50%。
##### 如果未设置该参数，则由以下字段或其默认值的总和决定。
##### ##   * rocksdb.defaultcf.block-cache-size 或系统全部内存的 25%
#####   * rocksdb.writecf.block-cache-size 或系统全部内存的 15%
#####   * rocksdb.lockcf.block-cache-size 或系统全部内存的 2%
#####   * raftdb.defaultcf.block-cache-size 或系统全部内存的 2%
##### ## 要在单个物理机上部署多个 TiKV 节点，需要显式配置该参数。
##### 否则，TiKV 中可能会出现 OOM 错误。
#### capacity = "1GB"

[pd]
#### pd 的地址
#### endpoints = ["127.0.0.1:2379","127.0.0.2:2379","127.0.0.3:2379"]

[metric]
#### 将 metrics 推送给 Prometheus pushgateway 的时间间隔
interval = "15s"
#### Prometheus pushgateway 的地址
address = ""
```

```
job = "tikv"

[raftstore]
#### 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"

#### 可以通过该参数打开或者关闭 RocksDB 的统计信息。
#### enable-statistics = true

#### 开启 RocksDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[rocksdb.defaultcf]
#### 数据块大小。RocksDB 是按照 block 为单元对数据进行压缩的，同时 block 也是缓存在 block-cache
#### 中的最小单元（类似其他数据库的 page 概念）。
block-size = "64KB"

#### RocksDB 每一层数据的压缩方式，可选的值为：no,snappy,zlib,bzip2,lz4,lz4hc,zstd。
#### no:no:lz4:lz4:lz4:zstd:zstd 表示 level0 和 level1 不压缩，level2 到 level4 采用 lz4 压缩算法
    ↪ ,
#### level5 和 level6 采用 zstd 压缩算法,。
#### no 表示没有压缩，lz4 是速度和压缩比较为中庸的压缩算法，zlib 的压缩比很高，对存储空间比较友
#### 好，但是压缩速度比较慢，压缩的时候需要占用较多的 CPU 资源。不同的机器需要根据 CPU 以及 I/O
    ↪ 资
#### 源情况来配置怎样的压缩方式。例如：如果采用的压缩方式为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量
#### 写入数据的情况下（导数据），发现系统的 I/O 压力很大（使用 iostat 发现 %util 持续 100% 或者使
#### 用 top 命令发现 iowait 特别多），而 CPU 的资源还比较充裕，这个时候可以考虑将 level0 和
#### level1 开启压缩，用 CPU 资源换取 I/O 资源。如果采用的压缩方式
#### 为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量写入数据的情况下，发现系统的 I/O 压力不大，但是 CPU
#### 资源已经吃光了，top -H 发现有大量的 bg 开头的线程（RocksDB 的 compaction 线程）在运行，这
#### 个时候可以考虑用 I/O 资源换取 CPU 资源，将压缩方式改成"no:no:no:lz4:lz4:zstd:zstd"。总之，目
#### 的是为了最大限度地利用系统的现有资源，使 TiKV 的性能在现有的资源情况下充分发挥。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### RocksDB memtable 的大小。
write-buffer-size = "128MB"

#### 最多允许几个 memtable 存在。写入到 RocksDB 的数据首先会记录到 WAL 日志里面，然后会插入到
#### memtable 里面，当 memtable 的大小到达了 write-buffer-size 限定的大小的时候，当前的
#### memtable 会变成只读的，然后生成一个新的 memtable 接收新的写入。只读的 memtable 会被
#### RocksDB 的 flush 线程（max-background-flushes 参数能够控制 flush 线程的最大个数）
#### flush 到磁盘，成为 level0 的一个 sst 文件。当 flush 线程忙不过来，导致等待 flush 到磁盘的
#### memtable 的数量到达 max-write-buffer-number 限定的个数的时候，RocksDB 会将新的写入
#### stall 住，stall 是 RocksDB 的一种流控机制。在导数据的时候可以将 max-write-buffer-number
#### 的值设置的更大一点，例如 10。
max-write-buffer-number = 5

#### 当 level0 的 sst 文件个数到达 level0-slowdown-writes-trigger 指定的限度的时候，
#### RocksDB 会尝试减慢写入的速度。因为 level0 的 sst 太多会导致 RocksDB 的读放大上升。
```

```
#### level0-slowdown-writes-trigger 和 level0-stop-writes-trigger 是 RocksDB 进行流控的
#### 另一个表现。当 level0 的 sst 的文件个数到达 4（默认值），level0 的 sst 文件会和 level1 中
#### 有 overlap 的 sst 文件进行 compaction，缓解读放大的问题。
level0-slowdown-writes-trigger = 20

#### 当 level0 的 sst 文件个数到达 level0-stop-writes-trigger 指定的限度的时候，RocksDB 会
#### stall 住新的写入。
level0-stop-writes-trigger = 36

#### 当 level1 的数据量大小达到 max-bytes-for-level-base 限定的值的时候，会触发 level1 的
#### sst 和 level2 种有 overlap 的 sst 进行 compaction。
#### 黄金定律：max-bytes-for-level-base 的设置的第一参考原则就是保证和 level0 的数据量大致相
#### 等，这样能够减少不必要的 compaction。例如压缩方式为"no:no:lz4:lz4:lz4:lz4:lz4"，那么
#### max-bytes-for-level-base 的值应该是 write-buffer-size 的大小乘以 4，因为 level0 和
#### level1 都没有压缩，而且 level0 触发 compaction 的条件是 sst 的个数到达 4（默认值）。在
#### level0 和 level1 都采取了压缩的情况下，就需要分析下 RocksDB 的日志，看一个 memtable 的压
#### 缩成一个 sst 文件的大小大概是多少，例如 32MB，那么 max-bytes-for-level-base 的建议值就应
#### 该是 32MB * 4 = 128MB。
max-bytes-for-level-base = "512MB"

#### sst 文件的大小。level0 的 sst 文件的大小受 write-buffer-size 和 level0 采用的压缩算法的
#### 影响，target-file-size-base 参数用于控制 level1-level6 单个 sst 文件的大小。
target-file-size-base = "32MB"

[rocksdb.writecf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1

#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"

[raftdb]
#### RaftDB 能够打开的最大文件句柄数。
#### max-open-files = 40960

#### 可以通过该参数打开或者关闭 RaftDB 的统计信息。
#### enable-statistics = true

#### 开启 RaftDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
```

```
#### compaction-readahead-size = "2MB"

[raftdb.defaultcf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1

#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"
```

12.5.2.3.2 TiKV 内存使用情况

除了以上列出的 block-cache 以及 write-buffer 会占用系统内存外：

1. 需预留一些内存作为系统的 page cache
2. TiKV 在处理大的查询的时候（例如 select * from ...）会读取数据然后在内存中生成对应的数据结构返回给 TiDB，这个过程中 TiKV 会占用一部分内存

12.5.2.3.3 TiKV 机器配置推荐

1. 生产环境中，不建议将 TiKV 部署在 CPU 核数小于 8 或内存低于 32GB 的机器上
2. 如果对写入吞吐要求比较高，建议使用吞吐能力比较好的磁盘
3. 如果对读写的延迟要求非常高，建议使用 IOPS 比较高的 SSD 盘

12.5.2.4 Follower Read

当系统中存在读取热点 Region 导致 leader 资源紧张成为整个系统读取瓶颈时，启用 Follower Read 功能可明显降低 leader 的负担，并且通过在多个 follower 之间均衡负载，显著地提升整体系统的吞吐能力。本文主要介绍 Follower Read 的使用方法与实现机制。

12.5.2.4.1 概述

Follower Read 功能是指在强一致性读的前提下使用 Region 的 follower 副本来承载数据读取的任务，从而提升 TiDB 集群的吞吐能力并降低 leader 负载。Follower Read 包含一系列将 TiKV 读取负载从 Region 的 leader 副本上 offload 到 follower 副本的负载均衡机制。TiKV 的 Follower Read 可以保证数据读取的一致性，可以为用户提供强一致的数据读取能力。

> 注意：
>
> 为了获得强一致读取的能力，在当前的实现中，follower 节点需要向 leader 节点询问当前的执行进度（即 ReadIndex），这会产生一次额外的网络请求开销，因此目前 Follower Read 的主要优势是处理隔离集群的读写请求以及提升整体读取吞吐。

12.5.2.4.2 使用方式

要开启 TiDB 的 Follower Read 功能，将变量 `tidb_replica_read` 的值设置为 `follower` 或 `leader-and-follower` 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。

- 当设定为默认值 `leader` 或者空字符串时，TiDB 会维持原有行为方式，将所有的读取操作都发送给 leader 副本处理。
- 当设置为 `follower` 时，TiDB 会选择 Region 的 follower 副本完成所有的数据读取操作。
- 当 `tidb_replica_read` 的值设为 `leader-and-follower` 时，TiDB 可以选择任意副本来执行读取操作。
- 当设置为 `leader-and-follower` 时，读请求会在 leader 和 follower 之间负载均衡。

12.5.2.4.3 实现机制

在 Follower Read 功能出现之前，TiDB 采用 strong leader 策略将所有的读写操作全部提交到 Region 的 leader 节点上完成。虽然 TiKV 能够很均匀地将 Region 分散到多个物理节点上，但是对于每一个 Region 来说，只有 leader 副本能够对外提供服务，另外的 follower 除了时刻同步数据准备着 failover 时投票切换成为 leader 外，没有办法对 TiDB 的请求提供任何帮助。

为了允许在 TiKV 的 follower 节点进行数据读取，同时又不破坏线性一致性和 Snapshot Isolation 的事务隔离，Region 的 follower 节点需要使用 Raft ReadIndex 协议确保当前读请求可以读到当前 leader 上已经 commit 的最新数据。在 TiDB 层面，Follower Read 只需根据负载均衡策略将某个 Region 的读取请求发送到 follower 节点。

Follower 强一致读

TiKV follower 节点处理读取请求时，首先使用 Raft ReadIndex 协议与 Region 当前的 leader 进行一次交互，来获取当前 Raft group 最新的 commit index。本地 apply 到所获取的 leader 最新 commit index 后，便可以开始正常的读取请求处理流程。

Follower 副本选择策略

由于 TiKV 的 Follower Read 不会破坏 TiDB 的 Snapshot Isolation 事务隔离级别，因此 TiDB 选择 follower 的策略可以采用 round robin 的方式。目前，对于 Coprocessor 请求，Follower Read 负载均衡策略粒度是连接级别的，对于一个 TiDB 的客户端连接在某个具体的 Region 上会固定使用同一个 follower，只有在选中的 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。而对于非 Coprocessor 请求（点查等），Follower Read 负载均衡策略粒度是事务级别的，对于一个 TiDB 的事务在某个具体的 Region 上会固定使用同一个 follower，同样在 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。

12.5.2.5 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。

12.5.2.5.1 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。

12.5.2.5.2 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。

12.5.2.5.3 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`。

12.5.2.6 TiFlash 性能调优

本文介绍了使 TiFlash 性能达到最优的几种方式，包括规划机器资源、TiDB 参数调优、配置 TiKV Region 大小等。

12.5.2.6.1 资源规划

对于希望节省机器资源，并且完全没有隔离要求的场景，可以使用 TiKV 和 TiFlash 联合部署。建议为 TiKV 与 TiFlash 分别留够资源，并且不要共享磁盘。

12.5.2.6.2 TiDB 相关参数调优

1. 对于 OLAP/TiFlash 专属的 TiDB 节点，建议调大读取并发数`tidb_distsql_scan_concurrency` 到 80：

```
set @@tidb_distsql_scan_concurrency = 80;
```

2. 开启 Super batch 功能：

   `tidb_allow_batch_cop` 变量用来设置从 TiFlash 读取时，是否把 Region 的请求进行合并。当查询中涉及的 Region 数量比较大，可以尝试设置该变量为 1（对带 `aggregation` 下推到 TiFlash Coprocessor 的请求生效），或设置该变量为 2（对全部下推到 TiFlash Coprocessor 请求生效）。

```
set @@tidb_allow_batch_cop = 1;
```

3. 尝试开启聚合推过 `Join` / `Union` 等 TiDB 算子的优化：

   `tidb_opt_agg_push_down` 变量用来设置优化器是否执行聚合函数下推到 Join 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 1。

```
set @@tidb_opt_agg_push_down = 1;
```

4. 尝试开启 `Distinct` 推过 `Join` / `Union` 等 TiDB 算子的优化：

   `tidb_opt_distinct_agg_push_down` 变量用来设置优化器是否执行带有 `Distinct` 的聚合函数（比如 `select count(distinct a)from t`）下推到 Coprocessor 的优化操作。当查询中带有 `Distinct` 的聚合操作执行很慢时，可以尝试设置该变量为 1。

```
set @@tidb_opt_distinct_agg_push_down = 1;
```

12.5.2.7 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。

12.5.2.7.1 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。

12.5.2.7.2 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。

12.5.2.7.3 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。

12.5.3 SQL 性能优化

12.5.4 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。

12.5.5 理解 TiDB 执行计划

12.5.5.1 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

12.5.5.1.1 解读 EXPLAIN 的返回结果

EXPLAIN 的返回结果包含以下字段：

- id 为算子名，或执行 SQL 语句需要执行的子任务。详见算子简介。
- estRows 为显示 TiDB 预计会处理的行数。该预估数可能基于字典信息（例如访问方法基于主键或唯一键），或基于 CMSketch 或直方图等统计信息估算而来。
- task 显示算子在执行语句时的所在位置。详见Task 简介。
- access-object 显示被访问的表、分区和索引。显示的索引为部分索引。以上示例中 TiDB 使用了 a 列的索引。尤其是在有组合索引的情况下，该字段显示的信息很有参考意义。
- operator info 显示访问表、分区和索引的其他信息。详见operator info 结果。

算子简介

算子是为返回查询结果而执行的特定步骤。真正执行扫表（读盘或者读 TiKV Block Cache）操作的算子有如下几类：

- TableFullScan：全表扫描。
- TableRangeScan：带有范围的表数据扫描。
- TableRowIDScan：根据上层传递下来的 RowID 扫描表数据。时常在索引读操作后检索符合条件的行。
- IndexFullScan：另一种“全表扫描”，扫的是索引数据，不是表数据。
- IndexRangeScan：带有范围的索引数据扫描操作。

TiDB 会汇聚 TiKV/TiFlash 上扫描的数据或者计算结果，这种“数据汇聚”算子目前有如下几类：

- TableReader：将 TiKV 上底层扫表算子 TableFullScan 或 TableRangeScan 得到的数据进行汇总。
- IndexReader：将 TiKV 上底层扫表算子 IndexFullScan 或 IndexRangeScan 得到的数据进行汇总。
- IndexLookUp：先汇总 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确地读取 TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。
- IndexMerge：和 IndexLookupReader 类似，可以看做是它的扩展，可以同时读取多个索引的数据，有多个 Build 端，一个 Probe 端。执行过程也很类似，先汇总所有 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确地读取 TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。

算子的执行顺序

算子的结构是树状的，但在查询执行过程中，并不严格要求子节点任务在父节点之前完成。TiDB 支持同一查询内的并行处理，即子节点“流入”父节点。父节点、子节点和同级节点可能并行执行查询的一部分。

在以上示例中，├─IndexRangeScan_8(Build) 算子为 a(a) 索引所匹配的行查找内部 RowID。└─TableRowIDScan_9 ↪ (Probe) 算子随后从表中检索这些行。

Build 总是先于 Probe 执行，并且 Build 总是出现在 Probe 前面。即如果一个算子有多个子节点，子节点 ID 后面有 Build 关键字的算子总是先于有 Probe 关键字的算子执行。TiDB 在展现执行计划的时候，Build 端总是第一个出现，接着才是 Probe 端。

范围查询

在 WHERE/HAVING/ON 条件中，TiDB 优化器会分析主键或索引键的查询返回。如数字、日期类型的比较符，如大于、小于、等于以及大于等于、小于等于，字符类型的 LIKE 符号等。

若要使用索引，条件必须是 “Sargable” (Search ARGument ABLE) 的。例如条件 YEAR(date_column)< 1992 不能使用索引，但 date_column < '1992-01-01 就可以使用索引。

推荐使用同一类型的数据以及同一类型的字符串和排序规则进行比较，以避免引入额外的 cast 操作而导致不能利用索引。

可以在范围查询条件中使用 AND（求交集）和 OR（求并集）进行组合。对于多维组合索引，可以对多个列使用条件。例如对组合索引 (a, b, c)：

- 当 a 为等值查询时，可以继续求 b 的查询范围。
- 当 b 也为等值查询时，可以继续求 c 的查询范围。
- 反之，如果 a 为非等值查询，则只能求 a 的范围。

Task 简介

目前 TiDB 的计算任务分为两种不同的 task：cop task 和 root task。Cop task 是指使用 TiKV 中的 Coprocessor 执行的计算任务，root task 是指在 TiDB 中执行的计算任务。

SQL 优化的目标之一是将计算尽可能地下推到 TiKV 中执行。TiKV 中的 Coprocessor 能支持大部分 SQL 内建函数（包括聚合函数和标量函数）、SQL LIMIT 操作、索引扫描和表扫描。但是，所有的 Join 操作都只能作为 root task 在 TiDB 上执行。

operator info 结果

EXPLAIN 返回结果中 operator info 列可显示诸如条件下推等信息。本文以上示例中，operator info 结果各字段解释如下：

- range: [1,1] 表示查询的 WHERE 字句 (a = 1) 被下推到了 TiKV，对应的 task 为 cop[tikv]。
- keep order:false 表示该查询的语义不需要 TiKV 按顺序返回结果。如果查询指定了排序（例如 SELECT * FROM t WHERE a = 1 ORDER BY id），该字段的返回结果为 keep order:true。
- stats:pseudo 表示 estRows 显示的预估数可能不准确。TiDB 定期在后台更新统计信息。也可以通过执行 ANALYZE TABLE t 来手动更新统计信息。

EXPLAIN 执行后，不同算子返回不同的信息。你可以使用 Optimizer Hints 来控制优化器的行为，以此控制物理算子的选择。例如 /*+ HASH_JOIN(t1, t2)*/ 表示优化器将使用 Hash Join 算法。详细内容见Optimizer Hints。

12.5.5.1.2 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。

12.5.5.1.3 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE
- ANALYZE TABLE

- TRACE
- TiDB in Action
- 系统变量

12.5.5.2 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。

12.5.5.2.1 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)
```

12.5.5.2.2 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。

12.5.5.3 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;
```

12.5.5.3.1 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。

12.5.5.3.2 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。

12.5.5.3.3 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}
    ↪                                                 | 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)。

12.5.5.3.4 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

12.5.5.4 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;
```

12.5.5.4.1 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                | 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。

12.5.5.4.2 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。

12.5.5.4.3 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)
```

12.5.5.4.4 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)
```

12.5.5.4.5 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

12.5.5.5 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;
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
SELECT SLEEP(1);
ANALYZE TABLE t1, t2;
```

12.5.5.5.1 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：

```
EXPLAIN SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id;
```

```
+--
    ↪ -------------------------------+-----------+-----------+-----------------------------+-----------------
    ↪
| id                              | estRows   | task      | access object               |
    ↪ operator info                                                                       |
+--
    ↪ -------------------------------+-----------+-----------+-----------------------------+-----------------
    ↪
| IndexJoin_10                    | 180000.00 | root      |                             | inner
    ↪ join, inner:IndexLookUp_9, outer key:test.t1.id, inner key:test.t2.t1_id |
| ├─TableReader_28(Build)         | 142020.00 | root      |                             | data:
    ↪ TableFullScan_27                                                                    |
```

```
| │ └─TableFullScan_27            | 142020.00 | cop[tikv] | table:t1                      | keep
    ↪ order:false                                                                          |
| └─IndexLookUp_9(Probe)          | 1.27      | root      |                               |
    ↪                                                                                      |
|   ├─IndexRangeScan_7(Build)     | 1.27      | cop[tikv] | table:t2, index:t1_id(t1_id) | range
    ↪ : decided by [eq(test.t2.t1_id, test.t1.id)], keep order:false                       |
|   └─TableRowIDScan_8(Probe)     | 1.27      | cop[tikv] | table:t2                      | keep
    ↪ order:false                                                                          |
+--
    ↪ ------------------------------+-----------+-----------+------------------------------+-----------------
    ↪
6 rows in set (0.00 sec)
```

Index Join 算法对内存消耗较小，但如果需要执行大量探查操作，运行速度可能会慢于其他 Join 算法。以下面这条查询语句为例：

```
SELECT * FROM t1 INNER JOIN t2 ON t1.id=t2.t1_id WHERE t1.pad1 = 'value' and t2.pad1='value';
```

在 Inner Join 操作中，TiDB 会先执行 Join Reorder 算法，所以不能确定会先读取 t1 还是 t2。假设 TiDB 先读取了 t1 来构建 Build 端，那么 TiDB 会在探查 t2 前先根据谓词 t1.col = 'value' 筛选数据，但接下来每次探查 t2 时都要应用谓词 t2.col='value'。所以对于这条语句，Index Join 算法可能不如其他 Join 算法高效。

但如果 Build 端的数据量比 Probe 端小，且 Probe 端的数据已预先建立了索引，那么这种情况下 Index Join 算法效率更高。在下面这段查询语句中，因为 Index Join 比 Hash Join 效率低，所以 SQL 优化器选择了 Hash Join 算法：

```
-- 删除已有索引
ALTER TABLE t2 DROP INDEX t1_id;

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ↪  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ↪ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
Query OK, 0 rows affected (0.29 sec)

+--
    ↪ --------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                                                  | memory
    ↪                 | disk |
+--
    ↪ --------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
```

```
| IndexJoin_13                | 90000.00 | 20000   | root      |               | time:613.19955ms
    ↪ , loops:21, inner:{total:42.494047ms, concurrency:5, task:12, construct:33.149671ms,
    ↪ fetch:9.322956ms, build:8.66µs}, probe:32.435355ms
    ↪
    ↪ | inner join, inner:TableReader_9, outer key:test.t2.t1_id, inner key:test.t1.id |
    ↪ 269.63341903686523 MB | N/A  |
| ├─TableReader_19(Build)     | 90000.00 | 90000   | root      |               | time:586.613252
    ↪ ms, loops:95, cop_task: {num: 3, max: 205.893949ms, min: 185.051354ms, avg: 194.878702ms,
    ↪  p95: 205.893949ms, max_proc_keys: 31715, p95_proc_keys: 31715, tot_proc: 332ms, tot_wait
    ↪ : 4ms, rpc_num: 4, rpc_time: 584.907774ms, copr_cache_hit_ratio: 0.00}, backoff{
    ↪ regionMiss: 2ms} | data:TableFullScan_18
    ↪                                                          | 182.624906539917 MB   | N/A  |
| │ └─TableFullScan_18        | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns, loops
    ↪ :0, tikv_task:{proc max:72ms, min:64ms, p80:72ms, p95:72ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                                          | N/A
    ↪                   | N/A  |
| └─TableReader_9(Probe)      | 0.00     | 5       | root      |               | time:8.432051ms
    ↪ , loops:14, cop_task: {num: 14, max: 629.805µs, min: 226.129µs, avg: 420.979µs, p95:
    ↪ 629.805µs, max_proc_keys: 4, p95_proc_keys: 4, rpc_num: 15, rpc_time: 5.953229ms,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪                                                                        | data:
    ↪ Selection_8                                                  | N/A
    ↪                   | N/A  |
|   └─Selection_8             | 0.00     | 5       | cop[tikv] |               | time:0ns, loops
    ↪ :0, tikv_task:{proc max:0s, min:0s, p80:0s, p95:0s, iters:14, tasks:14}
    ↪
    ↪ | eq(test.t1.int_col, 1)                                                    | N/A
    ↪                   | N/A  |
|     └─TableRangeScan_7      | 1.00     | 25      | cop[tikv] | table:t1      | time:0ns, loops
    ↪ :0, tikv_task:{proc max:0s, min:0s, p80:0s, p95:0s, iters:14, tasks:14}
    ↪
    ↪ | range: decided by [test.t2.t1_id], keep order:false                       | N/A
    ↪                   | N/A  |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
6 rows in set (0.61 sec)

+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                | memory                | disk    |
```

```
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
| HashJoin_19                  | 90000.00 | 20000   | root      |               | time:406.098528
    ↪ ms, loops:22, build_hash_table:{total:148.574644ms, fetch:146.843636ms, build:1.731008ms
    ↪ }, probe:{concurrency:5, total:2.026547436s, max:406.039309ms, probe:205.337813ms, fetch
    ↪ :1.821209623s}                                          | inner join, equal:[eq(test.t1.
    ↪ id, test.t2.t1_id)] | 30.00731658935547 MB  | 0 Bytes |
| ├─TableReader_22(Build)      | 71.01    | 10000   | root      |               | time
    ↪ :147.072725ms, loops:12, cop_task: {num: 3, max: 145.847743ms, min: 50.932527ms, avg:
    ↪ 113.009029ms, p95: 145.847743ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc:
    ↪ 284ms, rpc_num: 3, rpc_time: 338.950488ms, copr_cache_hit_ratio: 0.00} | data:
    ↪ Selection_21                                 | 29.679713249206543 MB | N/A     |
| │ └─Selection_21              | 71.01    | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:132ms, min:48ms, p80:132ms, p95:132ms, iters:83, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                           | N/A                   | N/A     |
| │   └─TableFullScan_20       | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:128ms, min:48ms, p80:128ms, p95:128ms, iters:83, tasks:3}
    ↪
    ↪ | keep order:false                                 | N/A                   | N/A     |
| └─TableReader_24(Probe)      | 90000.00 | 90000   | root      |               | time
    ↪ :365.918504ms, loops:91, cop_task: {num: 3, max: 398.62145ms, min: 338.460345ms, avg:
    ↪ 358.732721ms, p95: 398.62145ms, max_proc_keys: 31715, p95_proc_keys: 31715, tot_proc: 536
    ↪ ms, rpc_num: 3, rpc_time: 1.076128895s, copr_cache_hit_ratio: 0.00}  | data:
    ↪ TableFullScan_23                             | 182.62489891052246 MB | N/A     |
|   └─TableFullScan_23         | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:100ms, min:40ms, p80:100ms, p95:100ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                 | N/A                   | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
6 rows in set (0.41 sec)

+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
| id                           | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                    | memory                | disk    |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
| HashJoin_20                  | 90000.00 | 20000   | root      |               | time:441.897092
```

```
    ↪ ms, loops:21, build_hash_table:{total:138.600864ms, fetch:136.353899ms, build:2.246965ms
    ↪ }, probe:{concurrency:5, total:2.207403854s, max:441.850032ms, probe:148.01937ms, fetch
    ↪ :2.059384484s}                                                | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.t1_id)] | 30.00731658935547 MB  | 0 Bytes |
| ├─TableReader_25(Build)      | 71.01    | 10000   | root      |               | time
    ↪ :138.081807ms, loops:12, cop_task: {num: 3, max: 134.702901ms, min: 53.356202ms, avg:
    ↪ 93.372186ms, p95: 134.702901ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc: 236
    ↪ ms, rpc_num: 3, rpc_time: 280.017658ms, copr_cache_hit_ratio: 0.00}   | data:Selection_24
    ↪                                  | 29.680171966552734 MB | N/A     |
| │ └─Selection_24             | 71.01    | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:52ms, p80:80ms, p95:80ms, iters:83, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                            | N/A                   | N/A     |
| │   └─TableFullScan_23       | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:52ms, p80:80ms, p95:80ms, iters:83, tasks:3}
    ↪
    ↪ | keep order:false                                  | N/A                   | N/A     |
| └─TableReader_22(Probe)      | 90000.00 | 90000   | root      |               | time
    ↪ :413.560548ms, loops:91, cop_task: {num: 3, max: 432.938474ms, min: 231.263355ms, avg:
    ↪ 365.710741ms, p95: 432.938474ms, max_proc_keys: 31715, p95_proc_keys: 31715, tot_proc:
    ↪ 488ms, rpc_num: 3, rpc_time: 1.097021983s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_21                              | 182.62489891052246 MB | N/A     |
|   └─TableFullScan_21         | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:80ms, p80:80ms, p95:80ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                  | N/A                   | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------
    ↪
6 rows in set (0.44 sec)
```

在上面所示的 Index Join 操作中，`t1.int_col` 一列的索引被删除了。如果加上这个索引，操作执行速度可以从 `0.61` 秒 提高到 `0.14` 秒，如下表所示：

```
-- 重新添加索引
ALTER TABLE t2 ADD INDEX (t1_id);

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ↪  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ↪ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
Query OK, 0 rows affected (3.65 sec)
```

```
+--
    ↪ ------------------------------+----------+---------+-----------+---------------------------+---------
    ↪
| id                            | estRows  | actRows | task      | access object
    ↪                | execution info
    ↪
    ↪ | operator info                                                                 |
    ↪ memory               | disk |
+--
    ↪ ------------------------------+----------+---------+-----------+---------------------------+---------
    ↪
| IndexJoin_11                  | 90000.00 | 0       | root      |
    ↪                              | time:136.876686ms, loops:1, inner:{total:114.948158ms,
    ↪ concurrency:5, task:7, construct:5.329114ms, fetch:109.610054ms, build:2.38µs}, probe
    ↪ :1.699799ms
    ↪
    ↪ | inner join, inner:IndexLookUp_10, outer key:test.t1.id, inner key:test.t2.t1_id |
    ↪ 29.864535331726074 MB | N/A  |
| ├─TableReader_32(Build)       | 10000.00 | 10000   | root      |
    ↪                              | time:95.755212ms, loops:12, cop_task: {num: 3, max:
    ↪ 95.652443ms, min: 30.758712ms, avg: 57.545129ms, p95: 95.652443ms, max_proc_keys: 31724,
    ↪ p95_proc_keys: 31724, tot_proc: 124ms, rpc_num: 3, rpc_time: 172.528417ms,
    ↪ copr_cache_hit_ratio: 0.00}                     | data:Selection_31
    ↪                                                                  | 29.679298400878906 MB | N
    ↪ /A  |
| │ └─Selection_31             | 10000.00 | 10000   | cop[tikv] |
    ↪                              | time:0ns, loops:0, tikv_task:{proc max:44ms, min:28ms, p80
    ↪ :44ms, p95:44ms, iters:84, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                                                        | N/A
    ↪                   | N/A  |
| │   └─TableFullScan_30        | 71010.00 | 71010   | cop[tikv] | table:t1
    ↪                     | time:0ns, loops:0, tikv_task:{proc max:44ms, min:28ms, p80:44ms,
    ↪ p95:44ms, iters:84, tasks:3}
    ↪
    ↪ | keep order:false                                                              | N/A
    ↪                   | N/A  |
| └─IndexLookUp_10(Probe)       | 9.00     | 0       | root      |
    ↪                              | time:103.93801ms, loops:7
    ↪
    ↪ |                                                                             |
    ↪ 2.1787109375 KB       | N/A  |
|   ├─IndexRangeScan_8(Build)   | 9.00     | 0       | cop[tikv] | table:t2, index:t1_id(t1_id
    ↪ ) | time:0s, loops:0, cop_task: {num: 7, max: 23.969244ms, min: 12.003682ms, avg:
    ↪ 14.659066ms, p95: 23.969244ms, tot_proc: 100ms, rpc_num: 7, rpc_time: 102.435966ms,
```

```
    ↪ copr_cache_hit_ratio: 0.00}, tikv_task:{proc max:24ms, min:12ms, p80:16ms, p95:24ms,
    ↪ iters:7, tasks:7} | range: decided by [eq(test.t2.t1_id, test.t1.id)], keep order:false
    ↪             | N/A                   | N/A  |
|   └─TableRowIDScan_9(Probe)     | 9.00     | 0       | cop[tikv] | table:t2
    ↪                     | time:0ns, loops:0
    ↪
    ↪ | keep order:false                                                          | N/A
    ↪                   | N/A  |
+--
    ↪ ------------------------------+----------+---------+-----------+---------------------------+---------
    ↪
7 rows in set (0.14 sec)

+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory                | disk    |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
| HashJoin_31                 | 90000.00 | 0       | root      |               | time:402.263795
    ↪ ms, loops:1, build_hash_table:{total:128.467151ms, fetch:126.871282ms, build:1.595869ms},
    ↪  probe:{concurrency:5, total:2.010969815s, max:402.212295ms, probe:8.924769ms, fetch
    ↪ :2.002045046s}                                          | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.t1_id)] | 29.689788818359375 MB | 0 Bytes |
| ├─TableReader_34(Build)     | 10000.00 | 10000   | root      |               | time
    ↪ :126.765972ms, loops:11, cop_task: {num: 3, max: 126.721293ms, min: 54.375481ms, avg:
    ↪ 84.518849ms, p95: 126.721293ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc: 208
    ↪ ms, rpc_num: 3, rpc_time: 253.478218ms, copr_cache_hit_ratio: 0.00} | data:Selection_33
    ↪                                 | 29.679292678833008 MB | N/A     |
| │ └─Selection_33            | 10000.00 | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:72ms, min:56ms, p80:72ms, p95:72ms, iters:84, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                          | N/A                   | N/A     |
| │   └─TableFullScan_32      | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:72ms, min:56ms, p80:72ms, p95:72ms, iters:84, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_36(Probe)     | 90000.00 | 90000   | root      |               | time
    ↪ :400.447175ms, loops:90, cop_task: {num: 3, max: 400.838264ms, min: 309.474053ms, avg:
    ↪ 341.01943ms, p95: 400.838264ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc: 528
    ↪ ms, rpc_num: 3, rpc_time: 1.02298055s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_35                          | 182.62786674499512 MB | N/A     |
```

```
|   └─TableFullScan_35          | 90000.00 | 90000    | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:108ms, min:72ms, p80:108ms, p95:108ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
+--
    ↪ ----------------------------+----------+---------+-----------+--------------+------------------------------
    ↪
6 rows in set (0.40 sec)

+--
    ↪ ----------------------------+----------+---------+-----------+--------------+------------------------------
    ↪
| id                            | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                   | memory                | disk    |
+--
    ↪ ----------------------------+----------+---------+-----------+--------------+------------------------------
    ↪
| HashJoin_32                   | 90000.00 | 0       | root      |               | time:356.282882
    ↪ ms, loops:1, build_hash_table:{total:154.187155ms, fetch:151.259305ms, build:2.92785ms},
    ↪ probe:{concurrency:5, total:1.781087041s, max:356.238312ms, probe:7.406146ms, fetch
    ↪ :1.773680895s}                                                | inner join, equal:[eq(test
    ↪ .t1.id, test.t2.t1_id)] | 29.689788818359375 MB | 0 Bytes |
| ├─TableReader_41(Build)       | 10000.00 | 10000    | root      |               | time
    ↪ :151.190175ms, loops:11, cop_task: {num: 3, max: 151.055697ms, min: 56.214348ms, avg:
    ↪ 96.70463ms, p95: 151.055697ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc: 240
    ↪ ms, rpc_num: 3, rpc_time: 290.019942ms, copr_cache_hit_ratio: 0.00}   | data:Selection_40
    ↪                                    | 29.679292678833008 MB | N/A     |
| │ └─Selection_40              | 10000.00 | 10000    | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:56ms, p80:80ms, p95:80ms, iters:84, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                          | N/A                   | N/A     |
| │   └─TableFullScan_39        | 71010.00 | 71010    | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:56ms, p80:80ms, p95:80ms, iters:84, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_43(Probe)       | 90000.00 | 90000    | root      |               | time:354.68523
    ↪ ms, loops:90, cop_task: {num: 3, max: 354.313475ms, min: 328.460762ms, avg: 345.530558ms,
    ↪  p95: 354.313475ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc: 508ms, rpc_num:
    ↪  3, rpc_time: 1.036492374s, copr_cache_hit_ratio: 0.00} | data:TableFullScan_42
    ↪                              | 182.62786102294922 MB | N/A     |
|   └─TableFullScan_42          | 90000.00 | 90000    | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:84ms, min:64ms, p80:84ms, p95:84ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
```

```
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
6 rows in set (0.36 sec)
```

> 注意：
>
> 在上方示例中，SQL 优化器之所以选择了性能较差的 Hash Join 算法，而不是 Index Join 算法，原因在于查询优化是一个 NP 完全问题，可能会选择不太理想的计划。如果需要频繁调用这个查询，建议通过执行计划管理的方式将 Hint 与 SQL 语句绑定，这样要比在发送给 TiDB 的 SQL 语句中插入 Hint 更容易管理。

Index Join 相关算法

如果使用 Hint INL_JOIN 进行 Index Join 操作，TiDB 会在连接外表之前创建一个中间结果的 Hash Table。TiDB 同样也支持使用 Hint INL_HASH_JOIN 在外表上建 Hash Table。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。

12.5.5.5.2 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 1GB），且 oom-use-tmp-storage 的值为 true （默认为 true），那么 TiDB 会尝试使用临时存储，在磁盘上创建 Hash Join 的 Build 端。EXPLAIN ↪ ANALYZE 返回结果中的 execution info 一栏记录了有关内存使用情况等运行数据。下面的例子展示了 tidb_mem_quota_query 的值分别设为 1GB（默认）及 500MB 时，EXPLAIN ANALYZE 的返回结果（当内存配额设为 500MB 时，磁盘用作临时存储区）：

```
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
SET tidb_mem_quota_query=500 * 1024 * 1024;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+----------+---------+----------+--------------+--------------------------
    ↪
| id                          | estRows   | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory                | disk    |
+--
    ↪ --------------------------+----------+---------+----------+--------------+--------------------------
    ↪
```

```
| HashJoin_27                 | 142020.00 | 71010   | root      |               | time:647.508572
    ↪ ms, loops:72, build_hash_table:{total:579.254415ms, fetch:566.91012ms, build:12.344295ms
    ↪ }, probe:{concurrency:5, total:3.23315006s, max:647.520113ms, probe:330.884716ms, fetch
    ↪ :2.902265344s}                                          | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.id)] | 209.61642456054688 MB | 0 Bytes |
| ├─TableReader_29(Build)     | 142020.00 | 71010   | root      |               | time
    ↪ :567.088247ms, loops:72, cop_task: {num: 2, max: 569.809411ms, min: 369.67451ms, avg:
    ↪ 469.74196ms, p95: 569.809411ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc: 400
    ↪ ms, rpc_num: 2, rpc_time: 939.447231ms, copr_cache_hit_ratio: 0.00}   | data:
    ↪ TableFullScan_28                          | 210.2100534439087 MB  | N/A     |
| │ └─TableFullScan_28        | 142020.00 | 71010   | cop[tikv] | table:t1      | proc max:64ms,
    ↪  min:48ms, p80:64ms, p95:64ms, iters:79, tasks:2
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_31(Probe)     | 180000.00 | 90000   | root      |               | time
    ↪ :337.233636ms, loops:91, cop_task: {num: 3, max: 569.790741ms, min: 332.758911ms, avg:
    ↪ 421.543165ms, p95: 569.790741ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ↪ 500ms, rpc_num: 3, rpc_time: 1.264570696s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_30                          | 267.1126985549927 MB  | N/A     |
|   └─TableFullScan_30        | 180000.00 | 90000   | cop[tikv] | table:t2      | proc max:84ms,
    ↪  min:72ms, p80:84ms, p95:84ms, iters:102, tasks:3
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
+--
    ↪ ---------------------------+----------+---------+----------+--------------+-----------------------------
    ↪
5 rows in set (0.65 sec)

Query OK, 0 rows affected (0.00 sec)

+--
    ↪ ---------------------------+----------+---------+----------+--------------+-----------------------------
    ↪
| id                          | estRows   | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                   | memory                | disk
    ↪                 |
+--
    ↪ ---------------------------+----------+---------+----------+--------------+-----------------------------
    ↪
| HashJoin_27                 | 142020.00 | 71010   | root      |               | time:963.983353
    ↪ ms, loops:72, build_hash_table:{total:775.961447ms, fetch:503.789677ms, build:272.17177ms
    ↪ }, probe:{concurrency:5, total:4.805454793s, max:963.973133ms, probe:922.156835ms, fetch
    ↪ :3.883297958s}                                          | inner join, equal:[eq(test.t1.
    ↪ id, test.t2.id)] | 93.53974533081055 MB  | 210.7459259033203 MB |
```

```
| ├─TableReader_29(Build)     | 142020.00 | 71010   | root      |               | time
    ↪ :504.062018ms, loops:72, cop_task: {num: 2, max: 509.276857ms, min: 402.66386ms, avg:
    ↪ 455.970358ms, p95: 509.276857ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc:
    ↪ 384ms, rpc_num: 2, rpc_time: 911.893237ms, copr_cache_hit_ratio: 0.00}  | data:
    ↪ TableFullScan_28                          | 210.20934200286865 MB | N/A
    ↪ |
| │ └─TableFullScan_28        | 142020.00 | 71010   | cop[tikv] | table:t1      | proc max:88ms,
    ↪  min:72ms, p80:88ms, p95:88ms, iters:79, tasks:2
    ↪
    ↪ | keep order:false                                | N/A                   | N/A
    ↪                   |
| └─TableReader_31(Probe)     | 180000.00 | 90000   | root      |               | time
    ↪ :363.058382ms, loops:91, cop_task: {num: 3, max: 412.659191ms, min: 358.489688ms, avg:
    ↪ 391.463008ms, p95: 412.659191ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ↪ 484ms, rpc_num: 3, rpc_time: 1.174326746s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_30                          | 267.11340618133545 MB | N/A
    ↪ |
|   └─TableFullScan_30        | 180000.00 | 90000   | cop[tikv] | table:t2      | proc max:92ms,
    ↪  min:64ms, p80:92ms, p95:92ms, iters:102, tasks:3
    ↪
    ↪ | keep order:false                                | N/A                   | N/A
    ↪                   |
+--
    ↪ --------------------------+----------+---------+----------+---------------+-------------------------
    ↪
5 rows in set (0.98 sec)
```

配置

Hash Join 算法的性能受以下系统变量影响：

- tidb_mem_quota_query（默认值：1GB）- 如果某条查询的内存消耗超出了配额，TiDB 会尝试将 Hash Join 的 Build 端移到磁盘上以节省内存。
- tidb_hash_join_concurrency（默认值：5）- 可以并发执行的 Hash Join 任务数量。

12.5.5.5.3 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。

12.5.5.5.4 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

12.5.5.6 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;
```

12.5.5.6.1 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_14                      | 5.00     | root      |                              | inner
    ↪ join, inner:IndexLookUp_13, outer key:test.t2.t1_id, inner key:test.t1.id, equal cond:eq(
    ↪ test.t2.t1_id, test.t1.id) |
| ├─StreamAgg_49(Build)             | 5.00     | root      |                              | group
    ↪  by:test.t2.t1_id, funcs:firstrow(test.t2.t1_id)->test.t2.t1_id
    ↪                                                        |
| │ └─IndexReader_50                | 5.00     | root      |                              | index
    ↪ :StreamAgg_39
    ↪
    ↪ |
| │   └─StreamAgg_39                | 5.00     | cop[tikv] |                              | group
    ↪  by:test.t2.t1_id,
    ↪
    ↪ |
| │     └─IndexFullScan_31          | 50000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep
    ↪ order:true
```

```
    ↪
    ↪ |
| └─IndexLookUp_13(Probe)          | 1.00     | root      |                                |
    ↪
    ↪ |
|   ├─IndexRangeScan_11(Build)     | 1.00     | cop[tikv] | table:t1, index:PRIMARY(id)  | range
    ↪ : decided by [eq(test.t1.id, test.t2.t1_id)], keep order:false
    ↪                                                      |
|   └─TableRowIDScan_12(Probe)     | 1.00     | cop[tikv] | table:t1                     | keep
    ↪ order:false
    ↪
    ↪ |
+--
    ↪ --------------------------------+----------+-----------+------------------------------+----------------
    ↪
8 rows in set (0.00 sec)
```

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)。

12.5.5.6.2 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_17                     | 1978.13 | root      |                              | inner
    ↪ join, inner:IndexLookUp_16, outer key:test.t3.t1_id, inner key:test.t1.id, equal cond:eq(
    ↪ test.t3.t1_id, test.t1.id) |
| ├─TableReader_44(Build)          | 1978.00 | root      |                               | data:
    ↪ TableFullScan_43
```

```
    ↪
    ↪ |
| | └─TableFullScan_43             | 1978.00 | cop[tikv] | table:t3                    | keep
    ↪ order:false
    ↪
    ↪ |
| └─IndexLookUp_16(Probe)          | 1.00    | root      |                             |
    ↪
    ↪ |
|   ├─IndexRangeScan_14(Build)     | 1.00    | cop[tikv] | table:t1, index:PRIMARY(id) | range:
    ↪ decided by [eq(test.t1.id, test.t3.t1_id)], keep order:false
    ↪                                                     |
|   └─TableRowIDScan_15(Probe)     | 1.00    | cop[tikv] | table:t1                    | keep
    ↪ order:false
    ↪
    ↪ |
+--
    ↪ --------------------------------+---------+-----------+-----------------------------+-----------------
    ↪
6 rows in set (0.01 sec)
```

从语义上看，因为约束保证了 `t3.t1_id` 列值的唯一性，TiDB 可以直接执行 `INNER JOIN` 查询。

12.5.5.6.3 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)     | 180000.00 | root      |                                 | index:
    ↪ IndexFullScan_23
    ↪                                                                                          |
| │ └─IndexFullScan_23        | 180000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep
    ↪ order:true
    ↪                                                                                                |
| └─TableReader_22(Probe)     | 56808.00  | root      |                                 | data:
    ↪ Selection_21
    ↪                                                                                                     |
|   └─Selection_21            | 56808.00  | cop[tikv] |                                 | ne(test.
    ↪ t1.id, test.t1.int_col)
    ↪                                                                                  |
|     └─TableFullScan_20      | 71010.00  | cop[tikv] | table:t1                        | keep
    ↪ order:true
    ↪                                                                                                |
+--
    ↪ --------------------------+----------+----------+-----------------------------+-------------------
    ↪
6 rows in set (0.00 sec)
```

由上述查询结果可知，TiDB 执行了 Semi Join。不同于 Inner Join，Semi Join 仅允许右键 (t2.t1_id) 上的第一个值，也就是该操作将去除 Join 算子任务中的重复数据。Join 算法也包含 Merge Join，会按照排序顺序同时从左侧和右侧读取数据，这是一种高效的 Zipper Merge。

可以将原语句视为关联子查询，因为它引入了子查询外的 t1.int_col 列。然而，EXPLAIN 语句的返回结果显示的是关联子查询去关联后的执行计划。条件 t1_id != t1.int_col 会被重写为 t1.id != t1.int_col。TiDB 可以从表 t1 中读取数据并且在 └─Selection_21 中执行此操作，因此这种去关联和重写操作会极大提高执行效率。

12.5.5.6.4 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_14                    | 1582.40 | root      |                              | anti
    ↪ semi join, inner:IndexLookUp_13, outer key:test.t3.t1_id, inner key:test.t1.id, equal
    ↪ cond:eq(test.t3.t1_id, test.t1.id) |
| ├─TableReader_35(Build)         | 1978.00 | root      |                              | data:
    ↪ TableFullScan_34
    ↪
    ↪ |
| │ └─TableFullScan_34            | 1978.00 | cop[tikv] | table:t3                     | keep
    ↪ order:false
    ↪
    ↪ |
| └─IndexLookUp_13(Probe)         | 1.00    | root      |                              |
    ↪
    ↪ |
|   ├─IndexRangeScan_10(Build)    | 1.00    | cop[tikv] | table:t1, index:PRIMARY(id) | range:
    ↪ decided by [eq(test.t1.id, test.t3.t1_id)], keep order:false
    ↪                                                          |
|   └─Selection_12(Probe)         | 1.00    | cop[tikv] |                              | lt(test
    ↪ .t1.int_col, 100)
    ↪
    ↪ |
|     └─TableRowIDScan_11         | 1.00    | cop[tikv] | table:t1                     | keep
    ↪ order:false
    ↪
    ↪ |
+--
    ↪ --------------------------------+---------+-----------+------------------------------+-------------------
    ↪
7 rows in set (0.00 sec)
```

上述查询首先读取了表 t3，然后根据主键开始探测 (probe) 表 t1。连接类型是 anti semi join，即反半连接：之所以使用 anti，是因为上述示例有不存在匹配值（即 NOT IN）的情况；使用 Semi Join 则是因为仅需要匹配第一行后就可以停止查询。

12.5.5.6.5 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

12.5.5.7 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;
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)
```

12.5.5.7.1 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`。

12.5.5.7.2 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)
```

12.5.5.7.3 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

12.5.5.8 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。

12.5.5.8.1 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

12.5.5.9 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)        |
|   │       └─TableFullScan_53        | 17740.00 | cop[tikv] | table:t1, partition:p2017 | keep
    ↪ order:false                 |
|   ├─StreamAgg_74                    | 1.00     | root      |                           | funcs:
    ↪ count(Column#13)->Column#7 |
|   │ └─TableReader_75                | 1.00     | root      |                           | data:
    ↪ StreamAgg_63                |
|   │   └─StreamAgg_63                | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#13        |
|   │     └─Selection_73              | 3977.60  | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)        |
|   │       └─TableFullScan_72        | 4972.00  | cop[tikv] | table:t1, partition:p2018 | keep
    ↪ order:false                 |
|   ├─StreamAgg_93                    | 1.00     | root      |                           | funcs:
    ↪ count(Column#15)->Column#7 |
```

```
|   |  └─TableReader_94               | 1.00     | root      |                              | data:
    ↪ StreamAgg_82                |
|   |    └─StreamAgg_82               | 1.00     | cop[tikv] |                              | funcs:
    ↪ count(1)->Column#15         |
|   |      └─Selection_92             | 20361.60 | cop[tikv] |                              | eq(
    ↪ year(test.t1.d), 2017)        |
|   |        └─TableFullScan_91       | 25452.00 | cop[tikv] | table:t1, partition:p2019    | keep
    ↪ order:false                 |
|   └─StreamAgg_112                   | 1.00     | root      |                              | funcs:
    ↪ count(Column#17)->Column#7  |
|     └─TableReader_113               | 1.00     | root      |                              | data:
    ↪ StreamAgg_101               |
|       └─StreamAgg_101               | 1.00     | cop[tikv] |                              | funcs:
    ↪ count(1)->Column#17         |
|         └─Selection_111             | 8892.80  | cop[tikv] |                              | eq(
    ↪ year(test.t1.d), 2017)        |
|           └─TableFullScan_110       | 11116.00 | cop[tikv] | table:t1, partition:pmax     | keep
    ↪ order:false                 |
+--
    ↪ ---------------------------------+----------+-----------+---------------------------+-----------------
    ↪
27 rows in set (0.00 sec)
```

由上述 EXPLAIN 结果可知：

- TiDB 认为需要访问所有分区 (p2016..pMax)。这是因为 TiDB 将谓词 YEAR (d) = 2017 视为 non-sargable。这个问题并非是 TiDB 特有的。
- 在扫描每个分区时，Selection 算子将筛选出年份不为 2017 的行。
- 在每个分区上会执行流式聚合，以计算匹配的行数。
- └─PartitionUnion_21 算子会合并访问每个分区后的结果。

12.5.5.9.1 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

12.5.5.10 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。

12.5.5.10.1 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;
```

12.5.5.10.2 示例

```
EXPLAIN SELECT * FROM t WHERE a = 1 OR b = 1;
+--
    ↪ ----------------------+----------+-----------+---------------+---------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                         |
+--
    ↪ ----------------------+----------+-----------+---------------+---------------------------------+
    ↪
| TableReader_7           | 8000.00  | root      |               | data:Selection_6
    ↪                       |
| └─Selection_6           | 8000.00  | cop[tikv] |               | or(eq(test.t.a, 1), eq(test.t
    ↪ .b, 1)) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo       |
+--
    ↪ ----------------------+----------+-----------+---------------+---------------------------------+
    ↪
EXPLAIN SELECT /*+ USE_INDEX_MERGE(t) */ * FROM t WHERE a > 1 OR b > 1;
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------
    ↪
| id                            | estRows | task      | access object           | operator info
    ↪                                 |
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------
    ↪
| IndexMerge_16                 | 6666.67 | root      |                         |
    ↪                                                   |
| ├─IndexRangeScan_13(Build)    | 3333.33 | cop[tikv] | table:t, index:idx_a(a) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_14(Build)    | 3333.33 | cop[tikv] | table:t, index:idx_b(b) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| └─TableRowIDScan_15(Probe)    | 6666.67 | cop[tikv] | table:t                 | keep order:
    ↪ false, stats:pseudo               |
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------
```

```
↪
```

例如，在上述示例中，过滤条件是使用 OR 条件连接的 WHERE 子句。在启用索引合并前，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。针对这类场景，TiDB 引入了对表的新访问方式：索引合并。

在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进行合并，生成以上示例中后一个执行计划。此时的 IndexMerge_16 算子有三个子节点，其中 IndexRangeScan_13 和 IndexRangeScan_14 根据范围扫描得到符合条件的所有 RowID，再由 TableRowIDScan_15 算子根据这些 RowID 精确地读取所有满足条件的数据。

其中对于 IndexRangeScan/TableRangeScan 一类按范围进行的扫表操作，EXPLAIN 表中 operator info 列相比于其他扫表操作，多了被扫描数据的范围这一信息。比如上面的例子中，IndexRangeScan_13 算子中的 range:(1,+inf] 这一信息表示该算子扫描了从 1 到正无穷这个范围的数据。

> 注意：
>
> - 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。
> - 索引合并目前仅支持析取范式（or 连接的表达式），不支持合取范式（and 连接的表达式）。
> - 索引合并目前无法在临时表上使用。

12.5.5.10.3 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

12.5.6 SQL 优化流程

12.5.6.1 SQL 优化流程简介

在 TiDB 中，从输入的查询文本到最终的执行计划执行结果的过程可以见下图。

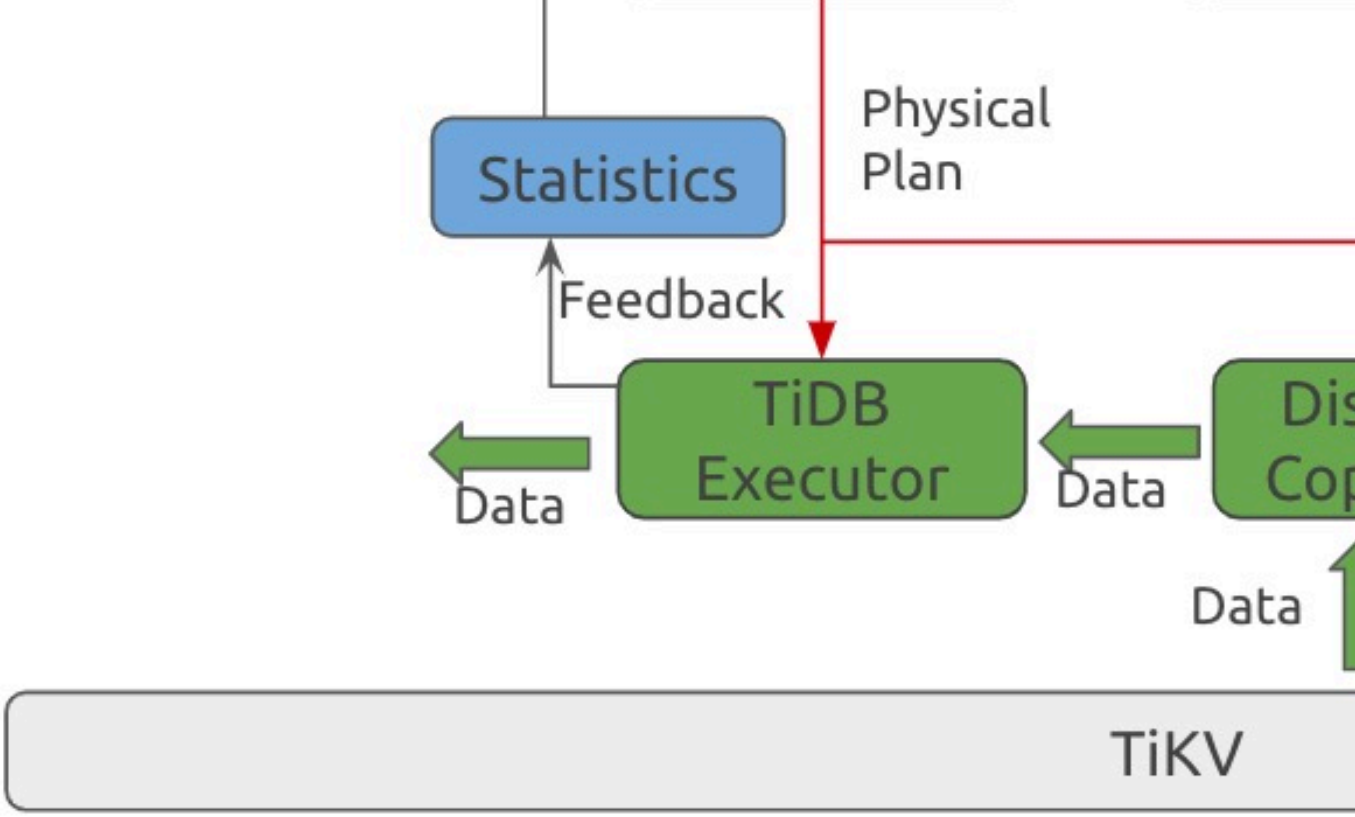


图 373: SQL Optimization

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。

12.5.6.2 逻辑优化

12.5.6.2.1 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

12.5.6.2.2 子查询相关的优化

本文主要介绍子查询相关的优化。

通常会遇到如下情况的子查询：

- `NOT IN (SELECT ... FROM ...)`
- `NOT EXISTS (SELECT ... FROM ...)`
- `IN (SELECT ... FROM ..)`
- `EXISTS (SELECT ... FROM ...)`
- `... >/>=/</<=/=/!= (SELECT ... FROM ...)`

有时，子查询中包含了非子查询中的列，如 `select * from t where t.a in (select * from t2 where t.b =t2.b)` 中，子查询中的 `t.b` 不是子查询中的列，而是从子查询外面引入的列。这种子查询通常会被称为关联子查询，外部引入的列会被称为关联列，关联子查询相关的优化参见关联子查询去关联。本文主要关注不涉及关联列的子查询。

子查询默认会以理解 TiDB 执行计划中提到的 `semi join` 作为默认的执行方式，同时对于一些特殊的子查询，TiDB 会做一些逻辑上的替换使得查询可以获得更好的执行性能。

`... < ALL (SELECT ... FROM ...)` 或者 `... > ANY (SELECT ... FROM ...)`

对于这种情况，可以将 `ALL` 或者 `ANY` 用 `MAX` 以及 `MIN` 来代替。不过由于在表为空时，`MAX(EXPR)` 以及 `MIN(EXPR)` 的结果会为 `NULL`，其表现形式和 `EXPR` 是有 `NULL` 值的结果一样。以及外部表达式结果为 `NULL` 时也会影响表达式的最终结果，因此这里完整的改写会是如下的形式：

- `t.id < all(select s.id from s)` 会被改写为 `t.id < min(s.id)and if(sum(s.id is null)!= 0, null, true)`。
- `t.id < any (select s.id from s)` 会被改写为 `t.id < max(s.id)or if(sum(s.id is null)!= 0, null, false)`。

`... != ANY (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数只有一种的话，那只要和这个值对比就即可。如果子查询中不同值的个数多于 1 个，那么必然会有不相等的情况出现。因此这样的子查询可以采取如下的改写手段：

- `select * from t where t.id != any (select s.id from s)` 会被改写为 `select t.* from t, (select s.id, count(distinct s.id)as cnt_distinct from s)where (t.id != s.id or cnt_distinct > 1)`

`... = ALL (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数多于一种的话，那么这个表达式的结果必然为假。因此这样的子查询在 TiDB 中会改写为如下的形式：

- select * from t where t.id = all (select s.id from s) 会被改写为 select t.* from t, (select s
  ↪ .id, count(distinct s.id)as cnt_distinct from s)where (t.id = s.id and cnt_distinct <= 1)

... IN (SELECT ... FROM ...)

对于这种情况，会将其 IN 的子查询改写为 SELECT ... FROM ... GROUP ... 的形式，然后将 IN 改写为普通的 JOIN 的形式。如 select * from t1 where t1.a in (select t2.a from t2) 会被改写为 select t1.* from t1
↪ , (select distinct(a)a from t2)t2 where t1.a = t2.a 的形式。同时这里的 DISTINCT 可以在 t2.a 具有 UNIQUE 属性时被自动消去。

```
explain select * from t1 where t1.a in (select t2.a from t2);
```

```
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
| id                           | estRows | task      | access object         | operator info
    ↪                                                                  |
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
| IndexJoin_12                 | 9990.00 | root      |                       | inner join, inner
    ↪ :TableReader_11, outer key:test.t2.a, inner key:test.t1.a |
| ├─HashAgg_21(Build)          | 7992.00 | root      |                       | group by:test.t2
    ↪ .a, funcs:firstrow(test.t2.a)->test.t2.a                  |
| │ └─IndexReader_28           | 9990.00 | root      |                       | index:
    ↪ IndexFullScan_27                                                 |
| │   └─IndexFullScan_27       | 9990.00 | cop[tikv] | table:t2, index:idx(a) | keep order:false
    ↪ , stats:pseudo                                            |
| └─TableReader_11(Probe)      | 1.00    | root      |                       | data:
    ↪ TableRangeScan_10                                                |
|   └─TableRangeScan_10        | 1.00    | cop[tikv] | table:t1              | range: decided
    ↪ by [test.t2.a], keep order:false, stats:pseudo             |
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
```

这个改写会在 IN 子查询相对较小，而外部查询相对较大时产生更好的执行性能。因为不经过改写的情况下，我们无法使用以 t2 为驱动表的 index join。同时这里的弊端便是，当改写删成的聚合无法被自动消去且 t2 表比较大时，反而会影响查询的性能。目前 TiDB 中使用tidb_opt_insubq_to_join_and_agg 变量来控制这个优化的打开与否。当遇到不合适这个优化的情况可以手动关闭。

EXISTS 子查询以及 ... >/>=/</<=/=/!= (SELECT ... FROM ...)

当前对于这种场景的子查询，当它不是关联子查询时，TiDB 会在优化阶段提前展开它，将其直接替换为一个结果集直接判断结果。如下图中，EXISTS 会提前在优化阶段被执行为 TRUE，从而不会在最终的执行结果中看到它。

```
create table t1(a int);
```

```
create table t2(a int);
insert into t2 values(1);
explain select * from t where exists (select * from t2);
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_12          | 10000.00 | root      |               | data:TableFullScan_11
    ↪ |
| └─TableFullScan_11      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:pseudo
    ↪  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
```

12.5.6.2.3 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。

12.5.6.2.4 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。这时可以通过优化规则及表达式下推的黑名单中关闭子查询去关联优化规则的方式来关闭这个优化。

样例

```
create table t1(a int, b int);
create table t2(a int, b int, index idx(b));
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    ↪ ---------------------------------+----------+-----------+---------------+-------------------------------
    ↪
| id                                | estRows  | task      | access object | operator info
    ↪                                                                                 |
+--
    ↪ ---------------------------------+----------+-----------+---------------+-------------------------------
    ↪
| HashJoin_11                       | 9990.00  | root      |               | inner join, equal:[eq
    ↪ (test.t1.b, test.t2.b)], other cond:lt(cast(test.t1.a), Column#7)  |
| ├─HashAgg_23(Build)               | 7992.00  | root      |               | group by:test.t2.b,
    ↪ funcs:sum(Column#8)->Column#7, funcs:firstrow(test.t2.b)->test.t2.b |
| │ └─TableReader_24                | 7992.00  | root      |               | data:HashAgg_16
    ↪                                                                        |
| │   └─HashAgg_16                  | 7992.00  | cop[tikv] |               | group by:test.t2.b,
    ↪ funcs:sum(test.t2.a)->Column#8                                        |
| │     └─Selection_22              | 9990.00  | cop[tikv] |               | not(isnull(test.t2.b
    ↪ ))                                                                    |
| │       └─TableFullScan_21        | 10000.00 | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo                                                            |
| └─TableReader_15(Probe)           | 9990.00  | root      |               | data:Selection_14
    ↪                                                                         |
|   └─Selection_14                  | 9990.00  | cop[tikv] |               | not(isnull(test.t1.b
    ↪ ))                                                                    |
```

```
|     └─TableFullScan_13           | 10000.00 | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo                                                          |
+--
    ↪ ------------------------------+----------+-----------+---------------+--------------------------------
    ↪
```

上面是优化生效的情况，可以看到 HashJoin_11 是一个普通的 inner join。

接下来，关闭关联规则：

```
insert into mysql.opt_rule_blacklist values("decorrelate");
admin reload opt_rule_blacklist;
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    ↪ -------------------------------------+----------+-----------+-----------------------+----------------
    ↪
| id                                  | estRows  | task      | access object           |
    ↪ operator info                                                        |
+--
    ↪ -------------------------------------+----------+-----------+-----------------------+----------------
    ↪
| Projection_10                       | 10000.00 | root      |                         | test.
    ↪ t1.a, test.t1.b                                                      |
| └─Apply_12                           | 10000.00 | root      |                         |
    ↪ CARTESIAN inner join, other cond:lt(cast(test.t1.a), Column#7)       |
|   ├─TableReader_14(Build)            | 10000.00 | root      |                         | data:
    ↪ TableFullScan_13                                                     |
|   │ └─TableFullScan_13               | 10000.00 | cop[tikv] | table:t1                | keep
    ↪ order:false, stats:pseudo                                            |
|   └─MaxOneRow_15(Probe)              | 1.00     | root      |                         |
    ↪                                                                      |
|     └─HashAgg_27                     | 1.00     | root      |                         | funcs
    ↪ :sum(Column#10)->Column#7                                            |
|       └─IndexLookUp_28               | 1.00     | root      |                         |
    ↪                                                                      |
|         ├─IndexRangeScan_25(Build)   | 10.00    | cop[tikv] | table:t2, index:idx(b) | range
    ↪ : decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo |
|         └─HashAgg_17(Probe)          | 1.00     | cop[tikv] |                         | funcs
    ↪ :sum(test.t2.a)->Column#10                                           |
|           └─TableRowIDScan_26        | 10.00    | cop[tikv] | table:t2                | keep
    ↪ order:false, stats:pseudo                                            |
+--
    ↪ -------------------------------------+----------+-----------+-----------------------+----------------
    ↪
```

在执行了关闭关联规则的语句后，可以在 IndexRangeScan_25(Build) 的 operator info 中看到 range: decided by [eq(test.t2.b, test.t1.b)]。这部分信息就是关联依赖未被解除时，TiDB 使用关联条件进行索引范围查询的显示结果。

12.5.6.2.5 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)
```

12.5.6.2.6 谓词下推

本文档介绍 TiDB 逻辑优化规则中的谓词下推规则，旨在让读者对谓词下推形成理解，并了解常见的谓词下推适用及不适用的场景。

谓词下推将查询语句中的过滤表达式计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

示例

以下通过一些例子对谓词下推优化进行说明，其中示例 1、2、3 为谓词下推适用的案例，示例 4、5、6 为谓词下推不适用的案例。

示例 1: 谓词下推到存储层

```
create table t(id int primary key, a int);
explain select * from t where a < 1;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
```

```
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，将谓词 `a < 1` 下推到 TiKV 上对数据进行过滤，可以减少由于网络传输带来的开销。

示例 2: 谓词下推到存储层

```
create table t(id int primary key, a int not null);
explain select * from t where a < substring('123', 1, 1);
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
```

该查询与示例 1 中的查询生成了完成一样的执行计划，这是因为谓词 `a < substring('123', 1, 1)` 的 `substring` 的入参均为常量，因此可以提前计算，进而简化得到等价的谓词 `a < 1`。进一步的，可以将 `a < 1` 下推至 TiKV 上。

示例 3: 谓词下推到 join 下方

```
create table t(id int primary key, a int not null);
```

```
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a where t.a < 1;
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                           | estRows  | task      | access object | operator info
    ↪                              |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashJoin_8                   | 4154.17  | root      |               | inner join, equal:[eq(
    ↪ test.t.a, test.s.a)] |
| ├─TableReader_15(Build)      | 3323.33  | root      |               | data:Selection_14
    ↪                            |
| │ └─Selection_14             | 3323.33  | cop[tikv] |               | lt(test.s.a, 1)
    ↪                              |
| │   └─TableFullScan_13       | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo             |
| └─TableReader_12(Probe)      | 3323.33  | root      |               | data:Selection_11
    ↪                            |
|   └─Selection_11             | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                              |
|     └─TableFullScan_10       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo             |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
7 rows in set (0.00 sec)
```

在该查询中，将谓词 `t.a < 1` 下推到 join 前进行过滤，可以减少 join 时的计算开销。

此外，这条 SQL 执行的是内连接，且 ON 条件是 `t.a = s.a`，可以由 `t.a < 1` 推导出谓词 `s.a < 1`，并将其下推至 join 运算前对 s 表进行过滤，可以进一步减少 join 时的计算开销。

示例 4: 存储层不支持的谓词无法下推

```
create table t(id int primary key, a int not null);
desc select * from t where substring('123', a, 1) = '1';
+--
    ↪ ----------------------+---------+-----------+---------------+----------------------------------+
    ↪
| id                      | estRows | task      | access object | operator info
    ↪                         |
+--
    ↪ ----------------------+---------+-----------+---------------+----------------------------------+
    ↪
| Selection_7             | 2.00    | root      |               | eq(substring("123", test.t.a,
```

```
    ↪ 1), "1") |
| └─TableReader_6         | 2.00    | root      |               | data:TableFullScan_5
    ↪                    |
|   └─TableFullScan_5     | 2.00    | cop[tikv] | table:t       | keep order:false, stats:pseudo
    ↪          |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
```

在该查询中，存在谓词 `substring('123', a, 1)= '1'`。

从 explain 结果中可以看到，该谓词没有被下推到 TiKV 上进行计算，这是因为 TiKV coprocessor 中没有对 `substring` 内置函数进行支持，因此无法将其下推到 TiKV 上。

示例 5: 外连接中内表上的谓词不能下推

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a where s.a is null;
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                            | estRows  | task      | access object | operator info
    ↪                                    |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| Selection_7                   | 10000.00 | root      |               | isnull(test.s.a)
    ↪                                  |
| └─HashJoin_8                   | 12500.00 | root      |               | left outer join, equal
    ↪ :[eq(test.t.a, test.s.a)] |
|   ├─TableReader_13(Build)     | 10000.00 | root      |               | data:TableFullScan_12
    ↪                           |
|   │ └─TableFullScan_12        | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo                  |
|   └─TableReader_11(Probe)     | 10000.00 | root      |               | data:TableFullScan_10
    ↪                           |
|     └─TableFullScan_10        | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo                  |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
6 rows in set (0.00 sec)
```

在该查询中，内表 s 上存在谓词 `s.a is null`。

从 explain 中可以看到，该谓词没有被下推到 join 前进行计算，这是因为外连接在不满足 on 条件时会对内表填充 NULL，而在该查询中 `s.a is null` 用来对 join 后的结果进行过滤，如果将其下推到 join 前在内表上进行过

滤，则下推前后不等价，因此不可进行下推。

示例 6: 谓词中包含用户变量时不能下推

```
create table t(id int primary key, a char);
set @a = 1;
explain select * from t where a < @a;
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| Selection_5             | 8000.00  | root      |               | lt(test.t.a, getvar("a"))
    ↪      |
| └─TableReader_7         | 10000.00 | root      |               | data:TableFullScan_6
    ↪           |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，表 t 上存在谓词 a < @a，其中 @a 为值为 1 的用户变量。

从 explain 中可以看到，该谓词没有像示例 2 中一样，将谓词简化为 a < 1 并下推到 TiKV 上进行计算。这是因为，用户变量 @a 的值可能会某些场景下在查询过程中发生改变，且 TiKV 对于用户变量 @a 的值不可知，因此 TiDB 不会将 @a 替换为 1，且不会下推至 TiKV 上进行计算。

一个帮助理解的例子如下：

```
create table t(id int primary key, a int);
insert into t values(1, 1), (2,2);
set @a = 1;
select id, a, @a:=@a+1 from t where a = @a;
+----+------+----------+
| id | a    | @a:=@a+1 |
+----+------+----------+
|  1 |    1 | 2        |
|  2 |    2 | 3        |
+----+------+----------+
2 rows in set (0.00 sec)
```

可以从在该查询中看到，@a 的值会在查询过程中发生改变，因此如果将 a = @a 替换为 a = 1 并下推至 TiKV，则优化前后不等价。

12.5.6.2.7 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。

12.5.6.2.8 TopN 和 Limit 下推

SQL 中的 LIMIT 子句在 TiDB 查询计划树中对应 Limit 算子节点，ORDER BY 子句在查询计划树中对应 Sort 算子节点，此外，我们会将相邻的 Limit 和 Sort 算子组合成 TopN 算子节点，表示按某个排序规则提取记录的前 N 项。从另一方面来说，Limit 节点等价于一个排序规则为空的 TopN 节点。

和谓词下推类似，TopN（及 Limit，下同）下推将查询计划树中的 TopN 计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

如果要关闭这个规则，可参照优化规则及表达式下推的黑名单中的关闭方法。

示例

以下通过一些例子对 TopN 下推进行说明。

示例 1：下推到存储层 Coprocessor

```
create table t(id int primary key, a int not null);
explain select * from t order by a limit 10;
```

```
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TopN_7                    | 10.00    | root      |               | test.t.a, offset:0, count
    ↪ :10   |
| └─TableReader_15           | 10.00    | root      |               | data:TopN_14
    ↪                   |
|   └─TopN_14                | 10.00    | cop[tikv] |               | test.t.a, offset:0, count
    ↪ :10   |
|     └─TableFullScan_13     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

在该查询中，将 TopN 算子节点下推到 TiKV 上对数据进行过滤，每个 Coprocessor 只向 TiDB 传输 10 条记录。在 TiDB 将数据整合后，再进行最终的过滤。

示例 2：TopN 下推过 Join 的情况（排序规则仅依赖于外表中的列）

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.a limit 10;
```

```
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                    |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| TopN_12                         | 10.00    | root      |               | test.t.a, offset:0,
    ↪ count:10                     |
| └─HashJoin_17                    | 12.50    | root      |               | left outer join,
    ↪ equal:[eq(test.t.a, test.s.a)] |
|   ├─TopN_18(Build)               | 10.00    | root      |               | test.t.a, offset:0,
    ↪ count:10                     |
|   │ └─TableReader_26             | 10.00    | root      |               | data:TopN_25
    ↪                                      |
|   │   └─TopN_25                  | 10.00    | cop[tikv] |               | test.t.a, offset:0,
    ↪ count:10                     |
|   │     └─TableFullScan_24       | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                   |
|   └─TableReader_30(Probe)        | 10000.00 | root      |               | data:
    ↪ TableFullScan_29                            |
|     └─TableFullScan_29           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    ↪ stats:pseudo                   |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
8 rows in set (0.01 sec)
```

在该查询中，TopN 算子的排序规则仅依赖于外表 t 中的列，可以将 TopN 下推到 Join 之前进行一次计算，以减少 Join 时的计算开销。除此之外，TiDB 同样将 TopN 下推到了存储层中。

示例 3：TopN 不能下推过 Join 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a order by t.id limit 10;
```

```
+-------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                            | estRows  | task      | access object | operator info
    ↪                               |
+-------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| TopN_12                       | 10.00    | root      |               | test.t.id, offset:0,
    ↪ count:10              |
| └─HashJoin_16                  | 12500.00 | root      |               | inner join, equal:[eq(
    ↪ test.t.a, test.s.a)] |
```

```
|   ├─TableReader_21(Build)     | 10000.00 | root      |               | data:TableFullScan_20
    ↪                       |
|   |  └─TableFullScan_20       | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo             |
|   └─TableReader_19(Probe)     | 10000.00 | root      |               | data:TableFullScan_18
    ↪                       |
|     └─TableFullScan_18        | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo             |
+-------------------------------+----------+-----------+---------------+-----------------------------------
    ↪
6 rows in set (0.00 sec)
```

TopN 无法下推过 Inner Join。以上面的查询为例，如果先 Join 得到 100 条记录，再做 TopN 可以剩余 10 条记录。而如果在 TopN 之前就过滤到剩余 10 条记录，做完 Join 之后可能就剩下 5 条了，导致了结果的差异。

同理，TopN 无法下推到 Outer Join 的内表上。在 TopN 的排序规则涉及多张表上的列时，也无法下推，如 `t.a+s.a`。只有当 TopN 的排序规则仅依赖于外表上的列时，才可以下推。

示例 4：TopN 转换成 Limit 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.id limit 10;
```

```
+---------------------------------+----------+-----------+---------------+-----------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                    |
+---------------------------------+----------+-----------+---------------+-----------------------------------
    ↪
| TopN_12                         | 10.00    | root      |               | test.t.id, offset:0,
    ↪ count:10                   |
| └─HashJoin_17                    | 12.50    | root      |               | left outer join,
    ↪ equal:[eq(test.t.a, test.s.a)] |
|   ├─Limit_21(Build)              | 10.00    | root      |               | offset:0, count:10
    ↪                               |
|   |  └─TableReader_31            | 10.00    | root      |               | data:Limit_30
    ↪                                  |
|   |    └─Limit_30                | 10.00    | cop[tikv] |               | offset:0, count:10
    ↪                               |
|   |      └─TableFullScan_29      | 10.00    | cop[tikv] | table:t       | keep order:true,
    ↪ stats:pseudo                    |
|   └─TableReader_35(Probe)        | 10000.00 | root      |               | data:
    ↪ TableFullScan_34                          |
|     └─TableFullScan_34           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    ↪ stats:pseudo                   |
```

```
+---------------------------------+----------+-----------+--------------+-------------------------------------------------
    ↪
8 rows in set (0.00 sec)
```

在上面的查询中，TopN 首先推到了外表 t 上。然后因为它要对 t.id 进行排序，而 t.id 是表 t 的主键，可以直接按顺序读出 (keep order:true)，从而省略了 TopN 中的排序，将其简化为 Limit。

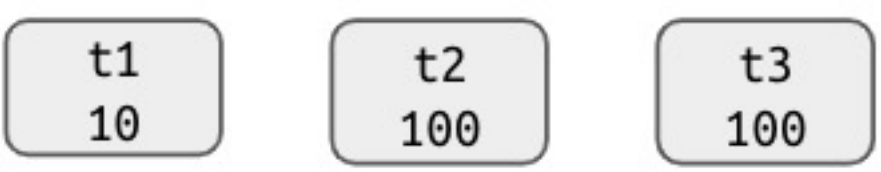
12.5.6.2.9 Join Reorder 算法简介

在实际的业务场景中，多个表的 Join 语句是很常见的，而 Join 的执行效率和各个表参与 Join 的顺序有关系。如 select * from t1, t2, t3 where t1.a=t2.a and t3.a=t2.a，这个 SQL 中可能的执行顺序有“t1 和 t2 先做 Join，然后再和 t3 做 Join”以及“t2 和 t3 先做 Join，然后再和 t1 做 Join”两种情况。根据 t1 和 t3 的数据量及数据分布，这两种执行顺序会有不同的性能表现。

因此优化器需要实现一种决定 Join 顺序的算法。目前 TiDB 中使用的算法是 Join Reorder 算法，又称贪心算法。

Join Reorder 算法实例

以三个表 t1、t2、t3 的 Join 为例。
序。



之后选定其中最小
一对。

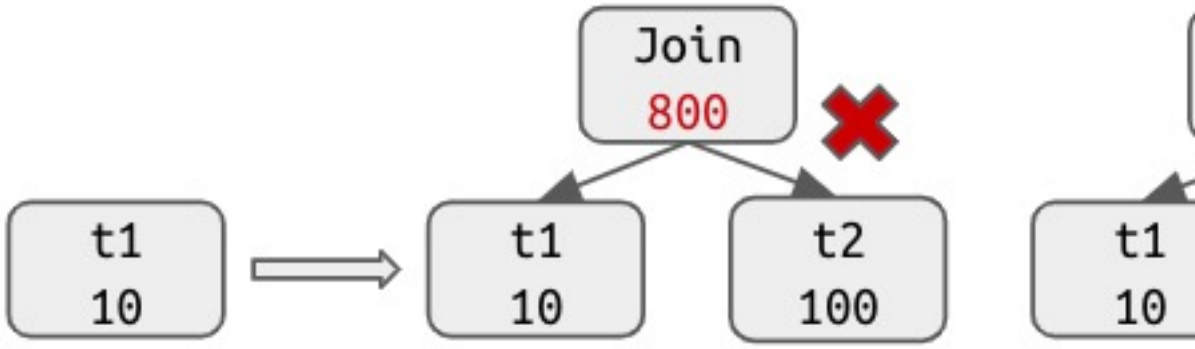


图 375: join-reorder-2

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。

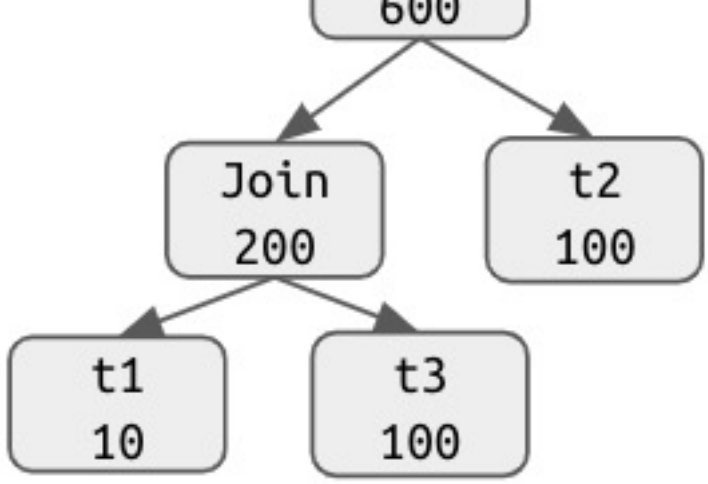


图 376: join-reorder-3

以上就是当前 TiDB 中使用的 Join reorder 算法。

Join reorder 算法限制

当前的 Join Reorder 算法存在如下限制：

- 受结果集的计算算法所限并不会保证一定会选到合适的 Join order

目前 TiDB 中支持使用 STRAIGHT_JOIN 语法来强制指定一种 Join 顺序，参见语法元素说明。

12.5.6.3 物理优化

12.5.6.3.1 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。

12.5.6.3.2 索引的选择

从存储层读取数据是 SQL 计算过程中最为耗时的部分之一，TiDB 目前支持从不同的存储和不同的索引中读取数据，索引选择得是否合理将很大程度上决定一个查询的运行速度。

本章节将介绍 TiDB 如何选择索引去读入数据，以及相关的一些控制索引选择的方式。

读表

在介绍索引的选择之前，首先要了解 TiDB 有哪些读表的方式，这些方式的触发条件是什么，不同方式有什么区别，各有什么优劣。

读表算子

读表算子	触发条件	适用场景	说明
PointGet/BatchPoint	读表的范围是一个或多个单点范围	任何场景	如果能被触发，通常被认为是最快的算子，因为其直接调用 kvget 的接口进行计算，不走 copro-cessor
TableReader	无	任何场景	从 TiKV 端直接扫描表数据，一般被认为是效率最低的算子，除非在 `_tidb_rowid` ↪ 这一列上存在范围查询，或者无其他可以选择的读表算子时，才会选择这个算子

读表算子	触发条件	适用场景	说明
TableReader	表在 TiFlash 节点上存在副本	需要读取的列比较少，但是需要计算的行很多	TiFlash 是列式存储，如果需要对少量的列和大量的行进行计算，一般会选择这个算子
IndexReader	表有一个或多个索引，且计算所需的列被包含在索引里	存在较小的索引上的范围查询，或者对索引列有顺序需求的时候	当存在多个索引的时候，会根据估算代价选择合理的索引
IndexLookupReader	表有一个或多个索引，且计算所需的列不完全被包含在索引里	同 IndexReader	因为计算列不完全被包含在索引里，所以读完索引后需要回表，这里会比 IndexReader 多一些开销

注意：

TableReader 是基于 `_tidb_rowid` 的索引，TiFlash 是列存索引，所以索引的选择即是读表算子的选择。

索引的选择

TiDB 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。

12.5.6.3.3 统计信息简介

TiDB 使用统计信息来决定索引的选择。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，作为实验特性启用。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。

> 注意：
>
> 当 tidb_analyze_version = 2 时，如果执行 ANALYZE 语句后发生 OOM，请设置全局变量 tidb_analyze_version = 1，然后进行以下操作之一：
>
> - 如果 ANALYZE 语句是手动执行的，请手动 analyze 每张需要的表：

```
sql  select distinct(concat('ANALYZE ',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，请使用以下 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 ;
```

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	√	√

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。

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。

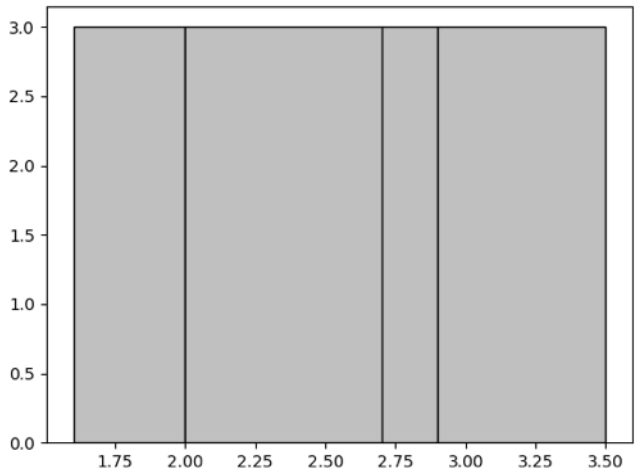


图 377: 等深直方图示例

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，更新的周期是 20 * stats-lease，stats-lease 的默认值是 3s，如果将其指定为 0，那么将不会自动更新。

和统计信息自动更新相关的三个系统变量如下：

系统变量名	默认值	功能
tidb_auto_analyze_ratio	0.5	自动更新阈值
tidb_auto_analyze_start_time	00:00 +0000	一天中能够进行自动更新的开始时间
tidb_auto_analyze_end_time	23:59 +0000	一天中能够进行自动更新的结束时间

当某个表 tbl 的修改行数与总行数的比值大于 tidb_auto_analyze_ratio，并且当前时间在 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 之间时，TiDB 会在后台执行 ANALYZE TABLE tbl 语句自动更新这个表的统计信息。

在 TiDB v5.0 之前，执行查询语句时，TiDB 会以feedback-probability 的概率收集反馈信息，并将其用于更新直方图和 Count-Min Sketch。从 v5.0 起，该功能默认关闭，暂不建议开启此功能。

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

配置项	对应的 ANALYZE 语法
采样率	WITH FLOAT_NUM SAMPLERATE
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：

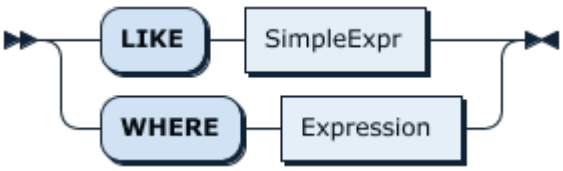


图 378: ShowLikeOrWhereOpt

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 379: 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 380: 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。

> **警告：**
>
> 统计信息同步加载目前为实验性特性，不建议在生产环境中使用。

统计信息同步加载特性默认关闭。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 0，代表未开启。

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 OFF，代表超时后 SQL 执行失败。当设置该变量为 ON 时，整个 SQL 优化过程不会使用任何列上的直方图、TopN 或 CMSketch，而是退回使用 pseudo 的统计信息。
- 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。

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。

另请参阅

- LOAD STATS
- DROP STATS

12.5.6.3.4 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。

12.5.6.3.5 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)
```

12.5.6.4 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。

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；
- 包含子查询的查询，如 `select * from t where a > (select ...)`；
- 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；
- ? 在 Limit 后的查询，如 `Limit ?` 或者 `Limit 10, ?`，此时 ? 的具体值对查询性能影响较大，故不缓存；
- ? 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，或修改 SQL Mode / time_zone 的值），或 LRU 淘汰机制触发都会导致 Execute 时无法命中执行计划缓存。

成功从缓存中获取到执行计划后，TiDB 会先检查执行计划是否依然合法，如果当前 Execute 在显式事务里执行，并且引用的表在事务前序语句中被修改，而缓存的执行计划对该表访问不包含 UnionScan 算子，则它不能被执行。

在通过合法性检测后，会根据当前最新参数值，对执行计划的扫描范围做相应调整，再用它执行获取数据。

关于执行计划缓存和查询性能有几点值得注意：

- 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)
```

12.5.6.4.1 Prepared Plan Cache 的内存管理

使用 Prepared Plan Cache 会有一定的内存开销，在内部测试中，平均每个缓存计划会消耗 100 KiB 内存，且目前 Plan Cache 是 SESSION 级别的，因此总内存消耗大致为 SESSION 个数 * SESSION 平均缓存计划个数 * 100KiB。

比如目前 TiDB 实例的 SESSION 并发数是 50，平均每个 SESSION 大致缓存 100 个计划，则总内存开销为 50 * 100 * 100KiB 约等于 512MB。

目前可以通过变量 tidb_prepared_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：

- TiDB Server 实例内存阈值 <= 64 GiB 时，tidb_prepared_plan_cache_size = 50
- TiDB Server 实例内存阈值 > 64 GiB 时，tidb_prepared_plan_cache_size = 100

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。

12.5.6.4.2 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.'
```

12.5.6.4.3 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)
```

12.5.7 控制执行计划

12.5.7.1 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。

12.5.7.2 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`。

12.5.7.2.1 语法

Optimizer Hints 不区分大小写，通过 `/*+ ... */` 注释的形式跟在 `SELECT`、`UPDATE` 或 `DELETE` 关键字的后面。`INSERT` 关键字后不支持 Optimizer Hints。

多个不同的 Hint 之间需用逗号隔开，例如：

```
SELECT /*+ USE_INDEX(t1, idx1), HASH_AGG(), HASH_JOIN(t1) */ count(*) FROM t t1, t t2 WHERE t1.a
    ↪ = t2.b;
```

可以通过Explain/Explain Analyze 语句的输出，来查看 Optimizer Hints 对查询执行计划的影响。

如果 Optimizer Hints 包含语法错误或不完整，查询语句不会报错，而是按照没有 Optimizer Hints 的情况执行。如果 Hint 不适用于当前语句，TiDB 会返回 Warning，用户可以在查询结束后通过 `Show Warnings` 命令查看具体信息。

> 注意：
>
> 如果注释不是跟在指定的关键字后，会被当作是普通的 MySQL comment，注释不会生效，且不会上报 warning。

TiDB 目前支持的 Optimizer Hints 根据生效范围的不同可以划分为两类：第一类是在查询块范围生效的 Hint，例如`/*+ HASH_AGG()*/`；第二类是在整个查询范围生效的 Hint，例如`/*+ MEMORY_QUOTA(1024 MB)*/`。

每条语句中每一个查询和子查询都对应着一个不同的查询块，每个查询块有自己对应的名字。以下面这条语句为例：

```
SELECT * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

该查询语句有 3 个查询块，最外面一层 SELECT 所在的查询块的名字为 sel_1，两个 SELECT 子查询的名字依次为 sel_2 和 sel_3。其中数字序号根据 SELECT 出现的位置从左到右计数。如果分别用 DELETE 和 UPDATE 查询替代第一个 SELECT 查询，则对应的查询块名字分别为 del_1 和 upd_1。

12.5.7.2.2 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_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;
```

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;
```

AGG_TO_COP()

AGG_TO_COP() 提示优化器将指定查询块中的聚合函数下推到 coprocessor。如果优化器没有下推某些适合下推的聚合函数，建议尝试使用。例如：

```
SELECT /*+ AGG_TO_COP() */ sum(t1.a) FROM t t1;
```

LIMIT_TO_COP()

LIMIT_TO_COP() 提示优化器将指定查询块中的 Limit 和 TopN 算子下推到 coprocessor。优化器没有下推 Limit 或者 TopN 算子时建议尝试使用该提示。例如：

```
SELECT /*+ LIMIT_TO_COP() */ * FROM t WHERE a = 1 AND b > 10 ORDER BY c LIMIT 1;
```

READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name …]], TIKV[t2_name [, tl_name …]])

READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name ...]], TIKV[t2_name [, tl_name ...]]) 提示优化器从指定的存储引擎来读取指定的表，目前支持的存储引擎参数有 TIKV 和 TIFLASH。如果为表指定了别名，就只能使用表的别名作为 READ_FROM_STORAGE() 的参数；如果没有指定别名，则用表的本名作为其参数。例如：

```
SELECT /*+ READ_FROM_STORAGE(TIFLASH[t1], TIKV[t2]) */ t1.a FROM t t1, t t2 WHERE t1.a = t2.a;
```

> 注意：
>
> 如果需要提示优化器使用的表不在同一个数据库内，需要显式指定数据库名。例如 SELECT /*+ READ_FROM_STORAGE(TIFLASH[test1.t1,test2.t2])*/ t1.a FROM test1.t t1, test2.t t2 WHERE t1.a = t2.a;。

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name …])

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name ...]) 提示优化器通过 index merge 的方式来访问指定的表，其中索引列表为可选参数。若显式地指出索引列表，会尝试在索引列表中选取索引来构建 index merge。若不给出索引列表，会尝试在所有可用的索引中选取索引来构建 index merge。例如：

```
SELECT /*+ USE_INDEX_MERGE(t1, idx_a, idx_b, idx_c) */ * FROM t1 WHERE t1.a > 10 OR t1.b > 10;
```

当对同一张表有多个 USE_INDEX_MERGE Hint 时，优化器会从这些 Hint 指定的索引列表的并集中尝试选取索引。

> 注意：
>
> USE_INDEX_MERGE 的参数是索引名，而不是列名。对于主键索引，索引名为 primary。

目前该 Hint 生效的条件较为苛刻，包括：

- 如果查询有除了全表扫以外的单索引扫描方式可以选择，优化器不会选择 index merge；

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

12.5.7.2.3 查询范围生效的 Hint

这类 Hint 只能跟在语句中第一个 SELECT、UPDATE 或 DELETE 关键字的后面，等同于在当前这条查询运行时对指定的系统变量进行修改，其优先级高于现有系统变量的值。

> 注意：
>
> 这类 Hint 虽然也有隐藏的可选变量 @QB_NAME，但就算指定了该值，Hint 还是会在整个查询范围生效。

NO_INDEX_MERGE()

NO_INDEX_MERGE() 会关闭优化器的 index merge 功能。

下面的例子不会使用 index merge：

```
SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE t.a > 0 or t.b > 0;
```

除了 Hint 外，系统变量 tidb_enable_index_merge 也能决定是否开启该功能。

> 注意：
>
> - NO_INDEX_MERGE 优先级高于 USE_INDEX_MERGE，当这两类 Hint 同时存在时，USE_INDEX_MERGE 不会生效。
> - 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。

12.5.7.3 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。

12.5.7.3.1 执行计划绑定 (SQL Binding)

执行计划绑定是 SPM 的基础。在优化器 Hints 中介绍了可以通过 Hint 的方式选择指定的执行计划，但有时需要在不修改 SQL 语句的情况下干预执行计划的选择。执行计划绑定功能使得可以在不修改 SQL 语句的情况下选择指定的执行计划。

创建绑定

```
CREATE [GLOBAL | SESSION] BINDING FOR BindableStmt USING BindableStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内为 SQL 绑定执行计划。目前，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。

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。

> 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

```
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。

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 8 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL
default_db	默认数据库名
status	状态，包括 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（由 `create [global] binding` 生成）、capture（由 tidb 自动创建生成）和 evolve（由 tidb 自动演进生成）

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)
```

12.5.7.3.2 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。

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一段时间。

> 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。

12.5.7.3.3 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，请与 PingCAP 的技术支持联系获取相关支持。

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。

12.5.7.3.4 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。
```

12.5.7.4 优化规则与表达式下推的黑名单

本文主要介绍优化规则的黑名单与表达式下推的黑名单。

12.5.7.4.1 优化规则黑名单

优化规则黑名单是针对优化规则的调优手段之一，主要用于手动禁用一些优化规则。

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

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;
```

12.5.7.4.2 表达式下推黑名单

表达式下推黑名单是针对表达式下推的调优手段之一，主要用于对于某些存储类型手动禁用一些表达式。

已支持下推的表达式

表达式分类	具体操作
逻辑运算	AND (&&), OR (||), NOT (!)
比较运算	<, <=, =, != (<>), >, >=, <=>, IN(), IS NULL, LIKE, IS TRUE, IS FALSE, COALESCE()
数值运算	+, -, *, /, ABS(), CEIL(), CEILING(), FLOOR()
控制流运算	CASE, IF(), IFNULL()
JSON 运算	JSON_TYPE(json_val), JSON_EXTRACT(json_doc, path[, path] ⋯), JSON_UNQUOTE(json_val), JSON_OBJECT(key, val[, key, val] ⋯), JSON_ARRAY([val[, val] ⋯]), JSON_MERGE(json_doc, json_doc[, json_doc] ⋯), JSON_SET(json_doc, path, val[, path, val] ⋯), JSON_INSERT(json_doc, path, val[, path, val] ⋯), JSON_REPLACE(json_doc, path, val[, path, val] ⋯), JSON_REMOVE(json_doc, path[, path] ⋯)

表达式分类	具体操作
日期运算	DATE_FORMAT()

禁止特定表达式下推

当函数的计算过程由于下推而出现异常时，可通过黑名单功能禁止其下推来快速恢复业务。具体而言，你可以将上述支持的函数或运算符名加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。

`mysql.expr_pushdown_blacklist` 的 schema 如下：

```
DESC mysql.expr_pushdown_blacklist;
```

```
+------------+--------------+------+------+-------------------+-------+
| Field      | Type         | Null | Key  | Default           | Extra |
+------------+--------------+------+------+-------------------+-------+
| name       | char(100)    | NO   |      | NULL              |       |
| store_type | char(100)    | NO   |      | tikv,tiflash,tidb |       |
| reason     | varchar(200) | YES  |      | NULL              |       |
+------------+--------------+------+------+-------------------+-------+
3 rows in set (0.00 sec)
```

以上结果字段解释如下：

- `name`：禁止下推的函数名。
- `store_type`：用于指明希望禁止该函数下推到哪些组件进行计算。组件可选 `tidb`、`tikv` 和 `tiflash`。`store_type` 不区分大小写，如果需要禁止向多个存储引擎下推，各个存储之间需用逗号隔开。
  - `store_type` 为 `tidb` 时表示在读取 TiDB 内存表时，是否允许该函数在其他 TiDB Server 上执行。
  - `store_type` 为 `tikv` 时表示是否允许该函数在 TiKV Server 的 Coprocessor 模块中执行。
  - `store_type` 为 `tiflash` 时表示是否允许该函数在 TiFlash Server 的 Coprocessor 模块中执行。
- `reason`：用于记录该函数被加入黑名单的原因。

使用方法

加入黑名单

如果要将一个或多个函数或运算符加入黑名单，执行以下步骤：

1. 向 `mysql.expr_pushdown_blacklist` 插入对应的函数名或运算符名以及希望禁止下推的存储引擎集合。
2. 执行 `admin reload expr_pushdown_blacklist;`。

移出黑名单

如果要将一个或多个函数及运算符移出黑名单，执行以下步骤：

1. 从 `mysql.expr_pushdown_blacklist` 表中删除对应的函数名或运算符名。

2. 执行 `admin reload expr_pushdown_blacklist;`。

> **注意：**
>
> `admin reload expr_pushdown_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。

表达式黑名单用法示例

以下示例首先将运算符 < 及 > 加入黑名单，然后将运算符 > 从黑名单中移出。

黑名单是否生效可以从 `explain` 结果中进行观察（参见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)
```

13 常见问题

13.1 FAQ

13.1.1 一、TiDB 介绍、架构、原理

13.1.1.1 1.1 TiDB 介绍及整体架构

13.1.1.1.1 1.1.1 TiDB 整体架构

TiDB 简介

13.1.1.1.2 1.1.2 TiDB 是什么？

TiDB 是一个分布式 NewSQL 数据库。它支持水平弹性扩展、ACID 事务、标准 SQL、MySQL 语法和 MySQL 协议，具有数据强一致的高可用特性，是一个不仅适合 OLTP 场景还适合 OLAP 场景的混合数据库。

13.1.1.1.3 1.1.3 TiDB 是基于 MySQL 开发的吗？

不是，虽然 TiDB 支持 MySQL 语法和协议，但是 TiDB 是由 PingCAP 团队完全自主开发的产品。

13.1.1.1.4 1.1.4 TiDB、 TiKV、 Placement Driver (PD) 主要作用？

- TiDB 是 Server 计算层，主要负责 SQL 的解析、制定查询计划、生成执行器。
- TiKV 是分布式 Key-Value 存储引擎，用来存储真正的数据，简而言之，TiKV 是 TiDB 的存储引擎。
- PD 是 TiDB 集群的管理组件，负责存储 TiKV 的元数据，同时也负责分配时间戳以及对 TiKV 做负载均衡调度。

13.1.1.1.5 1.1.5 TiDB 易用性如何？

TiDB 使用起来很简单，可以将 TiDB 集群当成 MySQL 来用，你可以将 TiDB 用在任何以 MySQL 作为后台存储服务的应用中，并且基本上不需要修改应用代码，同时你可以用大部分流行的 MySQL 管理工具来管理 TiDB。

13.1.1.1.6 1.1.6 TiDB 和 MySQL 兼容性如何？

TiDB 目前还不支持触发器、存储过程、自定义函数、外键，除此之外，TiDB 支持绝大部分 MySQL 5.7 的语法。

详情参见与 MySQL 兼容性对比。

13.1.1.1.7 1.1.7 TiDB 支持分布式事务吗？

支持。无论是一个地方的几个节点，还是跨多个数据中心的多个节点，TiDB 均支持 ACID 分布式事务。

TiDB 事务模型灵感源自 Google Percolator 模型，主体是一个两阶段提交协议，并进行了一些实用的优化。该模型依赖于一个时间戳分配器，为每个事务分配单调递增的时间戳，这样就检测到事务冲突。在 TiDB 集群中，PD 承担时间戳分配器的角色。

13.1.1.1.8 1.1.8 TiDB 支持哪些编程语言？

只要支持 MySQL Client/Driver 的编程语言，都可以直接使用 TiDB。

13.1.1.1.9 1.1.9 TiDB 是否支持其他存储引擎？

是的，除了 TiKV 之外，TiDB 还支持一些单机存储引擎，比如 UniStore 和 MockTiKV。注意，TiDB 后续版本可能不再支持 MockTiKV。

要查看 TiDB 支持的存储引擎，可使用以下命令：

```
./bin/tidb-server -h
```

返回结果如下：

```
Usage of ./bin/tidb-server:
  -L string
        log level: info, debug, warn, error, fatal (default "info")
  -P string
        tidb server port (default "4000")
  -V    print version information and exit (default false)
.........
  -store string
        registered store name, [tikv, mocktikv, unistore] (default "unistore")
  ......
```

13.1.1.1.10 1.1.10 除了官方文档，有没有其他 TiDB 知识获取途径？

目前官方文档是获取 TiDB 相关知识最主要、最及时的发布途径。除此之外，我们也有一些技术沟通群，如有需求可发邮件至 info@pingcap.com 获取，以及 AskTUG 网站与技术专家互动交流。

13.1.1.1.11 1.1.11 TiDB 用户名长度限制？

在 TiDB 中用户名最长为 32 字符。

13.1.1.1.12 1.1.12 TiDB 是否支持 XA？

虽然 TiDB 的 JDBC 驱动用的就是 MySQL JDBC (Connector / J)，但是当使用 Atomikos 的时候，数据源要配置成类似这样的配置：`type="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"`。MySQL JDBC XADataSource 连接 TiDB 的模式目前是不支持的。MySQL JDBC 中配置好的 XADataSource 模式，只对 MySQL 数据库起作用（DML 去修改 redo 等）。

Atomikos 配好两个数据源后，JDBC 驱动都要设置成 XA 模式，然后 Atomikos 在操作 TM 和 RM (DB) 的时候，会通过数据源的配置，发起带有 XA 指令到 JDBC 层。JDBC 层 XA 模式启用的情况下，会对 InnoDB（如果是 MySQL 的话）下发操作一连串 XA 逻辑的动作，包括 DML 去变更 redo log 等，就是两阶段递交的那些操作。TiDB 目前的引擎版本中，没有对上层应用层 JTA / XA 的支持，不解析这些 Atomikos 发过来的 XA 类型的操作。

MySQL 是单机数据库，只能通过 XA 来满足跨数据库事务，而 TiDB 本身就通过 Google 的 Percolator 事务模型支持分布式事务，性能稳定性比 XA 要高出很多，所以不会也不需要支持 XA。

13.1.1.2 1.2 TiDB 原理

13.1.1.2.1 1.2.1 存储 TiKV

1.2.1.1 TiKV 详细解读

三篇文章了解 TiDB 技术内幕 - 说存储

13.1.1.2.2 1.2.2 计算 TiDB

1.2.2.1 TiDB 详细解读

三篇文章了解 TiDB 技术内幕 - 说计算

13.1.1.2.3 1.2.3 调度 PD

1.2.3.1 PD 详细解读

三篇文章了解 TiDB 技术内幕 - 谈调度

13.1.2 二、云上部署

13.1.2.1 2.1 公有云

13.1.2.1.1 2.1.1 目前 TiDB 云上部署都支持哪些云厂商？

关于云上部署，TiDB 支持在 Google GKE、AWS EKS 和阿里云 ACK 上部署使用。

此外，TiDB 云上部署也已在京东云、UCloud 上线，均为数据库一级入口，欢迎大家使用。

13.1.3 三、故障排除

13.1.3.1 3.1 TiDB 自定义报错汇总

13.1.3.1.1 3.1.1 ERROR 8005 (HY000) : Write Conflict, txnStartTS is stale

可以检查 `tidb_disable_txn_auto_retry` 是否为 on。如是，将其设置为 off；如已经是 off，将 `tidb_retry_limit` 调大到不再发生该错误。

13.1.3.1.2 3.1.2 ERROR 9001 (HY000) : PD Server Timeout

请求 PD 超时，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

13.1.3.1.3 3.1.3 ERROR 9002 (HY000) : TiKV Server Timeout

请求 TiKV 超时，请检查 TiKV Server 状态/监控/日志以及 TiDB Server 与 TiKV Server 之间的网络。

13.1.3.1.4 3.1.4 ERROR 9003 (HY000) : TiKV Server is Busy

TiKV 操作繁忙，一般出现在数据库负载比较高时，请检查 TiKV Server 状态/监控/日志。

13.1.3.1.5 3.1.5 ERROR 9004 (HY000) : Resolve Lock Timeout

清理锁超时，当数据库上承载的业务存在大量的事务冲突时，会遇到这种错误，请检查业务代码是否有锁争用。

13.1.3.1.6 3.1.6 ERROR 9005 (HY000) : Region is unavailable

访问的 Region 不可用，某个 Raft Group 不可用，如副本数目不足，出现在 TiKV 比较繁忙或者是 TiKV 节点停机的时候，请检查 TiKV Server 状态/监控/日志。

13.1.3.1.7 3.1.7 ERROR 9006 (HY000) : GC life time is shorter than transaction duration

`GC Life Time` 间隔时间过短，长事务本应读到的数据可能被清理了。你可以使用如下命令修改`tidb_gc_life_time` 的值：

```
SET GLOBAL tidb_gc_life_time = '30m';
```

其中 30m 代表仅清理 30 分钟前的数据，这可能会额外占用一定的存储空间。

13.1.3.1.8 3.1.8 ERROR 9007 (HY000) : Write Conflict

可以检查 `tidb_disable_txn_auto_retry` 是否为 on。如是，将其设置为 off；如已经是 off，将 `tidb_retry_limit` 调大到不再发生该错误。

13.1.3.1.9 3.1.9 ERROR 8130 (HY000): client has multi-statement capability disabled

从早期版本的 TiDB 升级后，可能会出现该问题。为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。

可通过系统变量`tidb_multi_statement_mode`控制是否在同一 COM_QUERY 调用中执行多个查询。

13.1.3.2 3.2 MySQL 原生报错汇总

13.1.3.2.1 3.2.1 ERROR 2013 (HY000): Lost connection to MySQL server during query 问题的排查方法？

- log 中是否有 panic
- dmesg 中是否有 oom，命令：`dmesg -T | grep -i oom`
- 长时间没有访问，也会收到这个报错，一般是 tcp 超时导致的，tcp 长时间不用，会被操作系统 kill。

13.1.3.2.2 3.2.2 ERROR 1105 (HY000): other error: unknown error Wire Error(InvalidEnumValue(4004)) 是什么意思？

这类问题一般是 TiDB 和 TiKV 版本不匹配，在升级过程尽量一起升级，避免版本 mismatch。

13.1.3.2.3 3.2.3 ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法？

这个问题是因为在执行 `LOAD DATA LOCAL` 语句的时候，MySQL 客户端不允许执行此语句（即 `local_infile` 选项为 0）。解决方法是在启动 MySQL 客户端时，用 `--local-infile=1` 选项。具体启动指令类似：`mysql --local-infile=1 -u root -h 127.0.0.1 -P 4000`。有些 MySQL 客户端需要设置而有些不需要设置，原因是不同版本的 MySQL 客户端对 `local-infile` 的默认值不同。

13.1.3.2.4 3.2.4 ERROR 9001 (HY000): PD server timeout start timestamp may fall behind safe point

这个报错一般是 TiDB 访问 PD 出了问题，TiDB 后台有个 worker 会不断地从 PD 查询 safepoint，如果超过 100s 查不成功就会报这个错。一般是因为 PD 磁盘操作过忙、反应过慢，或者 TiDB 和 PD 之间的网络有问题。TiDB 常见错误码请参考错误码与故障诊断。

13.1.3.3 3.3 TiDB 日志中的报错信息

13.1.3.3.1 3.3.1 EOF

当客户端或者 proxy 断开连接时，TiDB 不会立刻察觉连接已断开，而是等到开始往连接返回数据时，才发现连接已断开，此时日志会打印 EOF 错误。

13.2 SQL 操作常见问题

本文档介绍 TiDB 中常见的 SQL 操作问题。

13.2.1 TiDB 对哪些 MySQL variables 兼容？

详细可参考系统变量。

13.2.2 省略 ORDER BY 条件时 TiDB 中返回结果的顺序与 MySQL 中的不一致

这不是 bug。返回结果的顺序视不同情况而定，不保证顺序统一。

MySQL 中，返回结果的顺序可能较为固定，因为查询是通过单线程执行的。但升级到新版本后，查询计划也可能变化。无论是否期待返回结果有序，都推荐使用 ORDER BY 条件。

ISO/IEC 9075:1992, Database Language SQL- July 30, 1992 对此有如下表述：

> If an <order by clause> is not specified, then the table specified by the <cursor specification> is T and the ordering of rows in T is implementation-dependent. (如果未指定 <order by 条件>，通过 <cursor specification> 指定的表为 T，那么 T 表中的行顺序视执行情况而定。)

以下两条查询的结果都是合法的：

```
> select * from t;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
+------+------+
2 rows in set (0.00 sec)
```

```
> select * from t; -- 不确定返回结果的顺序
+------+------+
| a    | b    |
+------+------+
|    2 |    2 |
|    1 |    1 |
+------+------+
2 rows in set (0.00 sec)
```

如果 ORDER BY 中使用的列不是唯一列，该语句就不确定返回结果的顺序。以下示例中，a 列有重复值，因此只有 ORDER BY a, b 能确定返回结果的顺序。

```
> select * from t order by a;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    1 |
|    2 |    2 |
+------+------+
3 rows in set (0.00 sec)
```

```
> select * from t order by a; -- 能确定 a 列的顺序，不确定 b 列的顺序
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    1 |
+------+------+
3 rows in set (0.00 sec)
```

13.2.3 TiDB 是否支持 SELECT FOR UPDATE？

支持。当 TiDB 使用悲观锁（自 TiDB v3.0 起默认使用）时，TiDB 中 SELECT FOR UPDATE 的行为与 MySQL 中的基本一致。

当 TiDB 使用乐观锁时，SELECT FOR UPDATE 不会在事务启动时对数据加锁，而是在提交事务时检查冲突。如果检查出冲突，会回滚待提交的事务。

13.2.4 TiDB 的 codec 能保证 UTF8 的字符串是 memcomparable 的吗？我们的 key 需要支持 UTF8，有什么编码建议吗？

TiDB 字符集默认就是 UTF8 而且目前只支持 UTF8，字符串就是 memcomparable 格式的。

13.2.5 一个事务中的语句数量最大是多少？

一个事务中的语句数量，默认限制最大为 5000 条。

13.2.6 TiDB 中，为什么出现后插入数据的自增 ID 反而小？

TiDB 的自增 ID (AUTO_INCREMENT) 只保证自增且唯一，并不保证连续分配。TiDB 目前采用批量分配的方式，所以如果在多台 TiDB 上同时插入数据，分配的自增 ID 会不连续。当多个线程并发往不同的 TiDB-server 插入数据的时候，有可能会出现后插入的数据自增 ID 小的情况。此外，TiDB 允许给整型类型的字段指定 AUTO_INCREMENT，且一个表只允许一个属性为 AUTO_INCREMENT 的字段。详情可参考CREATE TABLE 语法。

13.2.7 如何在 TiDB 中修改 sql_mode？

TiDB 支持在会话或全局作用域上修改sql_mode 系统变量。对全局作用域变量的更改将作用于集群中的其它服务器，并且重启后更改依然有效。因此，你无需在每台 TiDB 服务器上都更改 sql_mode 的值。

13.2.8 用 Sqoop 批量写入 TiDB 数据，虽然配置了 `--batch` 选项，但还是会遇到 `java.sql.BatchUpdateExecption:statement count 5001 exceeds the transaction limitation` 的错误，该如何解决？

- 在 Sqoop 中，`--batch` 是指每个批次提交 100 条 statement，但是默认每个 statement 包含 100 条 SQL 语句，所以此时 100 * 100 = 10000 条 SQL 语句，超出了 TiDB 的事务限制 5000 条，可以增加选项 `-Dsqoop.export.records.per.statement=10` 来解决这个问题，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10 \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大 TiDB 的单个事物语句数量限制，不过此操作会导致内存增加。

13.2.9 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考TiDB 历史数据回溯。

13.2.10 TiDB 中删除数据后会立即释放空间吗？

DELETE，TRUNCATE 和 DROP 都不会立即释放空间。对于 TRUNCATE 和 DROP 操作，在达到 TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的 GC 机制会删除数据并释放空间。对于 DELETE 操作，TiDB 的 GC 机制会删除数据，但不会立即释放空间，而是等到后续进行 compaction 时释放空间。

13.2.11 数据删除后查询速度为何会变慢？

大量删除数据后，会有很多无用的 key 存在，影响查询效率。可以尝试开启 Region Merge 功能，具体看参考最佳实践中的删除数据部分。

13.2.12 对数据做删除操作之后，空间回收比较慢，如何处理？

TiDB 采用了多版本并发控制 (MVCC)，为了使并发事务能查看到早期版本的数据，删除数据不会立即回收空间，而是推迟一段时间后再进行垃圾回收 (GC)。你可以通过修改系统变量`tidb_gc_life_time`的值（默认值为 `10m0s`）配置历史数据的保留时限。

13.2.13 `SHOW PROCESSLIST` 是否显示系统进程号？

TiDB 的 `SHOW PROCESSLIST` 与 MySQL 的 `SHOW PROCESSLIST` 显示内容基本一样，不会显示系统进程号，而 ID 表示当前的 session ID。其中 TiDB 的 `show processlist` 和 MySQL 的 `show processlist` 区别如下：

- 由于 TiDB 是分布式数据库，TiDB server 实例是无状态的 SQL 解析和执行引擎（详情可参考 TiDB 整体架构），用户使用 MySQL 客户端登录的是哪个 TiDB server，`show processlist` 就会显示当前连接的这个 TiDB server 中执行的 session 列表，不是整个集群中运行的全部 session 列表；而 MySQL 是单机数据库，`show processlist` 列出的是当前整个 MySQL 数据库的全部执行 SQL 列表。
- 在查询执行期间，TiDB 中的 `State` 列不会持续更新。由于 TiDB 支持并行查询，每个语句可能同时处于多个状态，因此很难显示为某一种状态。

13.2.14 在 TiDB 中如何控制或改变 SQL 提交的执行优先级？

TiDB 支持改变per-session、全局或单个语句的优先级。优先级包括：

- HIGH_PRIORITY：该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
- LOW_PRIORITY：该语句为低优先级语句，TiDB 在执行阶段会降低这条语句的优先级

以上两种参数可以结合 TiDB 的 DML 语言进行使用，使用方法举例如下：

1. 通过在数据库中写 SQL 的方式来调整优先级：

```
select HIGH_PRIORITY | LOW_PRIORITY count(*) from table_name;
insert HIGH_PRIORITY | LOW_PRIORITY into table_name insert_values;
delete HIGH_PRIORITY | LOW_PRIORITY from table_name;
update HIGH_PRIORITY | LOW_PRIORITY table_reference set assignment_list where
    ↪ where_condition;
replace HIGH_PRIORITY | LOW_PRIORITY into table_name;
```

2. 全表扫会自动调整为低优先级，analyze 也是默认低优先级。

13.2.15 在 TiDB 中 auto analyze 的触发策略是怎样的？

触发策略：新表达到 1000 条，并且在 1 分钟内没有写入，会自动触发。

当表的（修改数/当前总行数）大于 `tidb_auto_analyze_ratio` 的时候，会自动触发 `analyze` 语句。`tidb_auto_analyze_ratio` 的默认值为 0.5，即默认开启此功能。为了保险起见，在开启此功能的时候，保证了其最小值为 0.3。但是不能大于等于 `pseudo-estimate-ratio`（默认值为 0.8），否则会有一段时间使用 pseudo 统计信息，建议设置值为 0.5。

你可以用系统变量`tidb_enable_auto_analyze` 关闭 auto analyze。

13.2.16 可以使用 Hints 控制优化器行为吗？

在 TiDB 中，你可以用多种方法控制查询优化器的默认行为，包括使用Optimizer Hints 和SQL 执行计划管理 (SPM)。基本用法同 MySQL 中的一致，还包含若干 TiDB 特有的用法，示例如下：`select column_name from table_name ↪ use index(index_name) where where_condition;`

13.2.17 触发 Information schema is changed 错误的原因？

TiDB 在执行 SQL 语句时，会使用当时的 schema 来处理该 SQL 语句，而且 TiDB 支持在线异步变更 DDL。那么，在执行 DML 的时候可能有 DDL 语句也在执行，而你需要确保每个 SQL 语句在同一个 schema 上执行。所以当执行 DML 时，遇到正在执行中的 DDL 操作就可能会报 `Information schema is changed` 的错误。为了避免太多的 DML 语句报错，已做了一些优化。

现在会报此错的可能原因如下（只有第一个报错原因与表有关）：

- 执行的 DML 语句中涉及的表和集群中正在执行的 DDL 的表有相同的，那么这个 DML 语句就会报此错。
- 这个 DML 执行时间很久，而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 1024（此为默认值，可以通过 `tidb_max_delta_schema_count` 变量修改）。
- 接受 DML 请求的 TiDB 长时间不能加载到 `schema information`（TiDB 与 PD 或 TiKV 之间的网络连接故障等会导致此问题），而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 100。
- TiDB 重启后执行第一个 DDL 操作前，执行 DML 操作，并且在执行过程中遇到了第 1 个 DDL 操作（即在执行第 1 个 DDL 操作前，启动该 DML 对应的事务，且在该 DDL 变更第一个 schema 版本后，提交该 DML 对应的事务），那么这个 DML 会报此错。

> 注意：
>
> - 目前 TiDB 未缓存所有的 schema 版本信息。
> - 对于每个 DDL 操作，schema 版本变更的数量与对应 `schema state` 变更的次数一致。
> - 不同的 DDL 操作版本变更次数不一样。例如，`create table` 操作会有 1 次 schema 版本变更；`add column` 操作有 4 次 schema 版本变更。

13.2.18 触发 Information schema is out of date 错误的原因？

当执行 DML 时，TiDB 超过一个 DDL lease 时间（默认 45s）没能加载到最新的 schema 就可能会报 `Information schema is out of date` 的错误。遇到此错的可能原因如下：

- 执行此 DML 的 TiDB 被 kill 后准备退出，且此 DML 对应的事务执行时间超过一个 DDL lease，在事务提交时会报这个错误。
- TiDB 在执行此 DML 时，有一段时间内连不上 PD 或者 TiKV，导致 TiDB 超过一个 DDL lease 时间没有 load schema，或者导致 TiDB 断开与 PD 之间带 keep alive 设置的连接。

13.2.19 高并发情况下执行 DDL 时报错的原因？

高并发情况下执行 DDL（比如批量建表）时，极少部分 DDL 可能会由于并发执行时 key 冲突而执行失败。

并发执行 DDL 时，建议将 DDL 数量保持在 20 以下，否则你需要在应用端重试失败的 DDL 语句。

13.2.20 SQL 优化

13.2.20.1 TiDB 执行计划解读

详细解读理解 TiDB 执行计划。

13.2.20.2 统计信息收集

详细解读统计信息。

13.2.20.3 Count 如何加速？

Count 就是暴力扫表，提高并发度能显著的提升速度，修改并发度可以参考 `tidb_distsql_scan_concurrency` 变量，但是也要看 CPU 和 I/O 资源。TiDB 每次查询都要访问 TiKV，在数据量小的情况下，MySQL 都在内存里，TiDB 还需要进行一次网络访问。

提升建议：

- 建议提升硬件配置，可以参考部署建议。
- 提升并发度，默认是 10，可以提升到 50 试试，但是一般提升在 2-4 倍之间。
- 测试大数据量的 count。
- 调优 TiKV 配置，可以参考性能调优。
- 可以参考下推计算结果缓存。

13.2.20.4 查看当前 DDL 的进度？

通过 `admin show ddl` 查看当前 job 进度。操作如下：

```
admin show ddl;
```

```
*************************** 1. row ***************************
  SCHEMA_VER: 140
       OWNER: 1a1c4174-0fcd-4ba0-add9-12d08c4077dc
RUNNING_JOBS: ID:121, Type:add index, State:running, SchemaState:write reorganization, SchemaID
    ↪ :1, TableID:118, RowCount:77312, ArgLen:0, start time: 2018-12-05 16:26:10.652 +0800 CST,
    ↪  Err:<nil>, ErrCount:0, SnapshotVersion:404749908941733890
     SELF_ID: 1a1c4174-0fcd-4ba0-add9-12d08c4077dc
```

从上面操作结果可知，当前正在处理的是 `add index` 操作。且从 `RUNNING_JOBS` 列的 `RowCount` 字段可以知道当前 `add index` 操作已经添加了 77312 行索引。

13.2.20.5 如何查看 DDL job？

可以使用 `admin show ddl` 语句查看正在运行的 DDL 作业。

- `admin show ddl jobs`：用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。
- `admin show ddl job queries 'job_id' [, 'job_id'] ...`：用于显示 `job_id` 对应的 DDL 任务的原始 SQL 语句。此 `job_id` 只搜索正在执行中的任务以及 DDL 历史作业队列中的最近十条结果。

13.2.20.6 TiDB 是否支持基于 COST 的优化 (CBO)，如果支持，实现到什么程度？

是的，TiDB 使用的是基于成本的优化器 (CBO)，会对代价模型、统计信息持续优化。除此之外，TiDB 还支持 hash join、 soft-merge join 等 join 算法。

13.2.20.7 如何确定某张表是否需要做 analyze ？

可以通过 `show stats_healthy` 来查看 Healthy 字段，一般小于等于 60 的表需要做 analyze。

13.2.20.8 SQL 的执行计划展开成了树，ID 的序号有什么规律吗？这棵树的执行顺序会是怎么样的？

ID 没什么规律，只要是唯一就行。不过生成的时候，是有一个计数器，生成一个 plan 就加一，执行的顺序和序号无关。整个执行计划是一颗树，执行时从根节点开始，不断地向上返回数据。执行计划的理解，请参考理解 TiDB 执行计划。

13.2.20.9 TiDB 执行计划中，task cop 在一个 root 下，这个是并行的么？

目前 TiDB 的计算任务隶属于两种不同的 task：cop task 和 root task。 cop task 是指被下推到 KV 端分布式执行的计算任务，root task 是指在 TiDB 端单点执行的计算任务。

一般来讲 root task 的输入数据是来自于 cop task 的，但是 root task 在处理数据的时候，TiKV 上的 cop task 也可以同时处理数据，等待 TiDB 的 root task 拉取。所以从上述观点来看，他们是并行的，同时存在数据上下游关系。

在执行的过程中，某些时间段其实也是并行的，第一个 cop task 在处理 [100, 200] 的数据，第二个 cop task 在处理 [1, 100] 的数据。执行计划的理解，请参考理解 TiDB 执行计划。

13.2.21 数据库优化

13.2.21.1 TiDB 参数及调整

详情参考TiDB 配置参数。

13.2.21.2 如何打散热点

TiDB 中以 Region 分片来管理数据库，通常来讲，TiDB 的热点指的是 Region 的读写访问热点。而 TiDB 中对于非整数主键或没有主键的表，可以通过设置 SHARD_ROW_ID_BITS 来适度分解 Region 分片，以达到打散 Region 热点的效果。详情可参考官网SHARD_ROW_ID_BITS 中的介绍。

13.2.21.3 TiKV 性能参数调优

详情参考TiKV 性能参数调优。

13.3 高可用常见问题

本文档介绍高可用相关的常见问题。

13.3.1 TiDB 数据是强一致的吗？

通过使用 Raft 一致性算法，数据在各 TiKV 节点间复制为多副本，以确保某个节点挂掉时数据的安全性。

在底层，TiKV 使用复制日志 + 状态机 (State Machine) 的模型来复制数据。对于写入请求，数据被写入 Leader，然后 Leader 以日志的形式将命令复制到它的 Follower 中。当集群中的大多数节点收到此日志时，日志会被提交，状态机会相应作出变更。

13.3.2 官方有没有三中心跨机房多活部署的推荐方案？

从 TiDB 架构来讲，支持真正意义上的跨中心异地多活，从操作层面讲，依赖数据中心之间的网络延迟和稳定性，一般建议延迟在 5ms 以下，目前我们已经有相似客户方案，具体请咨询官方 info@pingcap.com。

13.4 高可靠常见问题

本文档介绍高可靠相关的常见问题。

13.4.1 我们的安全漏洞扫描工具对 MySQL version 有要求，TiDB 是否支持修改 server 版本号呢？

TiDB 在 v3.0.8 后支持修改 server 版本号，可以通过配置文件中的`server-version`配置项进行修改。在使用 TiUP 部署集群时，可以通过 `tiup cluster edit-config <cluster-name>` 修改配置文件中以下部分来设置合适的版本号：

```
server_configs:
  tidb:
    server-version: 'YOUR_VERSION_STRING'
```

并通过 `tiup cluster reload <cluster-name> -R tidb` 命令使得以上修改生效，以避免出现安全漏洞扫描不通过的问题。

13.4.2 TiDB 支持哪些认证协议，过程是怎样的？

这一层跟 MySQL 一样，走的 SASL 认证协议，用于用户登录认证，对密码的处理流程。

客户端连接 TiDB 的时候，走的是 challenge-response（挑战 -应答）的认证模式，过程如下：

1. 客户端连接服务器；
2. 服务器发送随机字符串 `challenge` 给客户端；
3. 客户端发送 `username` + `response` 给服务器；
4. 服务器验证 `response`。

13.4.3 如何修改用户名密码和权限？

TiDB 作为分布式数据库，在 TiDB 中修改用户密码建议使用 `set password for 'root'@'%' = '0101001';` 或 `alter` 方法，不推荐使用 `update mysql.user` 的方法进行，这种方法可能会造成其它节点刷新不及时的情况。修改权限也一样，都建议采用官方的标准语法。详情可参考TiDB 用户账户管理。

©2022 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。