零声教育出品 Mark 老师 QQ：2548898954

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。

分布式系统

> 分布式系统是一种其组件位于不同的联网计算机上的系统，然后通过互相传递消息来进行通讯和协调，为了达到共同的目标，这些组件会相互作用；换句话说，分布式系统把需要进行大量计算的工程数据分割成若干个小块，由多台计算机分别进行计算和存储，然后将结果统一合并到数据结论的科学；本质上就是**进行**数据存储与计算的**分治**；

CAP 理论

一致性

> 指所有的节点在同一时间的数据一致性；
>
> all nodes see the same data at the same time;

可用性

> 服务在正常响应时间内的可用；
>
> reads and writes always succeed;

分区容错性

> 分布式系统在遇到某节点或网络分区故障的时候仍然能够对外提供满足一致性或可用性的服务；

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。

  > 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。

  > iDB 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

  > 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

> 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。（ETL 是将业务系统的数据经过抽取、清洗转换之后加载到数据仓库的过程，目的是将企业中的分散、零乱、标准不统一的数据整合到一起，为企业的决策提供分析依据）

关系型模型

> 在传统的在线交易场景里，关系型模型仍然是标准；关系型数据库的关键在于一定要具备事务；

事务

> 事务的本质是：并发控制的单元，是用户定义的一个操作序列；这些操作要么都做，要么都不做，是一个不可分割的工作单位；
>
> 为了保证系统始终处于一个完整且正确的状态；

ACID 特性

- 原子性

  > 事务包含的全部操作时一个不可分割的整体；要么全部执行，要么全部不执行；

- 一致性

  > 事务的前后，所有的数据都保持一个一致的状态；不能违反数据的一致性检测；

- 隔离性

> 各个并发事务之间互相影响的程度；主要规定多个并发事务访问同一个数据资源，各个并发事务对该数据资源访问的行为；不同的隔离性是应对不同的现象（脏读、可重复读、幻读等）；

- 持久性

> 事务一旦完成要将数据所做的变更记录下来；包括数据存储和多副本的网络备份；

TiDB 部署本地测试集群

```
# 下载并安装 TiUP
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
# 声明全局环境变量
source .bash_profile
# 运行最新版本的 TiDB 集群，其中 TiDB、TiKV、PD 和 TiFlash 实例各 1 个
tiup playground
# 如果想指定版本并运行多个
tiup playground v4.0.16 --db 3 --pd 3 --kv 3 --monitor
# 注意：按照上面的部署，在结束部署测试后 TiUP 会清理掉原集群数据，重新执行该命令后会得到一个全新的集群。

# 如果希望持久化数据，并指定存储目录为 /tmp/tidb
tiup playground -T /tmp/tidb
```

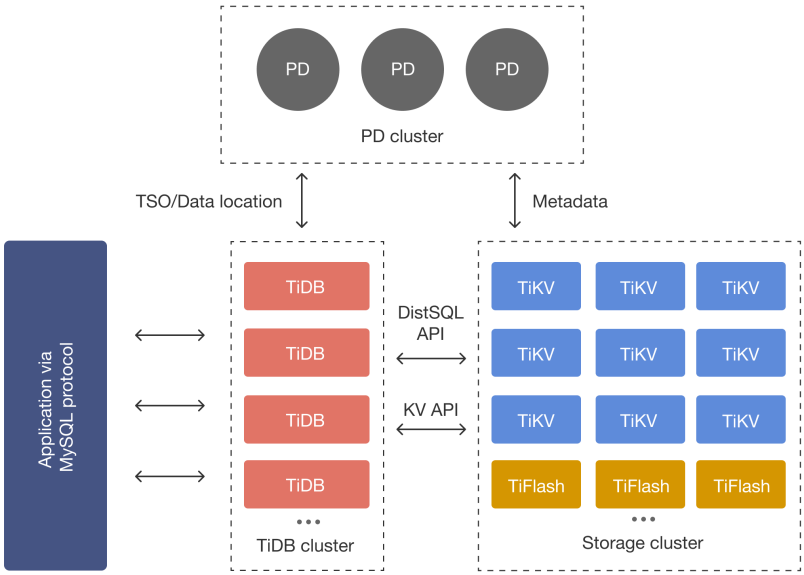
与传统非分布式数据库架构对比

- 两者都支持 ACID、事务强一致性；
- 分布式架构，组件解耦，拥有良好的扩展性，支持弹性的扩缩容；

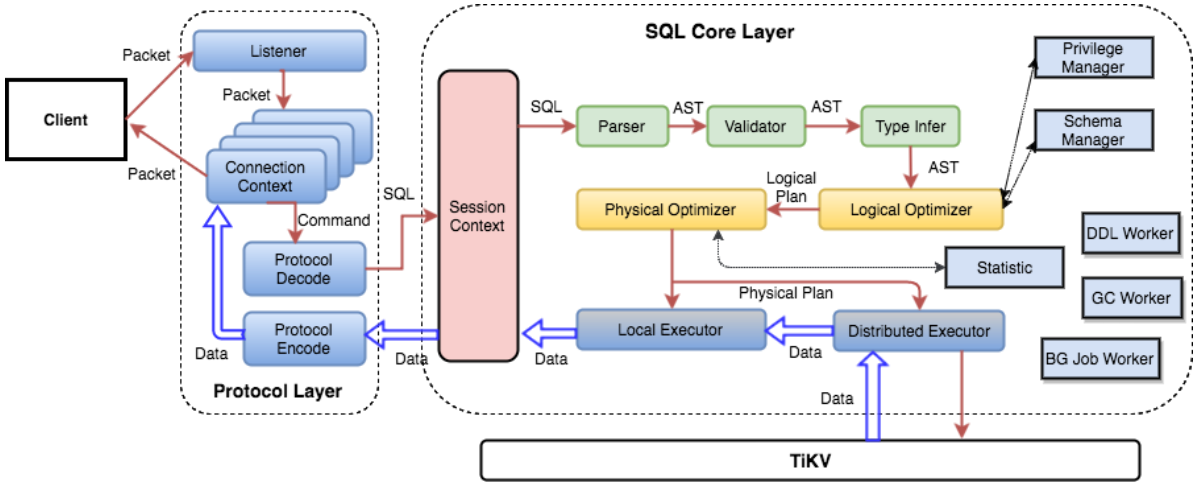
- 默认支持高可用，在少数副本失效的情况下，数据库能够自动进行故障转移，对业务透明；
- 采用水平扩展，在大数据量、高吞吐的业务场景中具有先天优势；
- 强项不在于轻量的简单 SQL 的响应速度，而在于大量高并发 SQL 的吞吐；

TiDB 分布式数据库整体架构

- 由多模块组成，各模块互相通信，组成完整的 TiDB 系统；
- 前端 `stateless`、后端 `stateful (Raft)`；
- 兼容 MySQL；



TiDB Server 的模块



> SQL 层，对外暴露 MySQL 协议的连接 endpoint，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划。TiDB 层本身是无状态的，实践中可以启动多个 TiDB 实例，通过负载均衡组件（如 LVS、HAProxy 或 F5）对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个 TiDB 实例上以达到负载均衡的效果。TiDB Server 本身并不存储数据，只是解析 SQL，将实际的数据读取请求转发给底层的存储节点 TiKV（或 TiFlash）。

数据映射关系

> 数据与 KV 的映射关系中定义如下：

```
tablePrefix     = []byte{'t'}
recordPrefixSep = []byte{'r'}
indexPrefixSep  = []byte{'i'}
```

> 假设表结构如下：

```
CREATE TABLE User (
    ID int,
    Name varchar(20),
    Role varchar(20),
    Age int,
    UID int,
    PRIMARY KEY (ID),
    KEY idxAge (Age),
    UNIQUE KEY idxUID (UID)
);
```

> 假设表数据如下：

```
1, "TiDB", "SQL Layer", 10, 10001
2, "TiKV", "KV Engine", 20, 10002
3, "PD", "Manager", 30, 10003
```

表数据与 KV 的映射关系

- Key 的形式：`tablePrefix{TableID}_recordPrefixSep{RowID}`；
- Value 的形式：`[col1, col2, col3, col4]`
- 映射示例：

```
t10_r1 --> ["TiDB", "SQL Layer", 10, 10001]
t10_r2 --> ["TiKV", "KV Engine", 20, 10002]
t10_r3 --> ["PD",   "Manager",   30, 10003]
```

索引数据和 KV 的映射关系

对于唯一索引：

- Key 的形式：`tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue`
- Value 的形式：`RowID`
- 映射示例：

```
t10_i1_10001 --> 1
t10_i2_10002 --> 2
t10_i3_10003 --> 3
```

非唯一索引：

- Key 的形式：`tablePrefix{TableID}_indexPrefixSep{IndexID}_indexedColumnsValue_{RowID}`
- Value 的形式：`null`
- 映射示例：

```
# 假设 IndexID 为 1
t10_i1_10_1 --> null
t10_i1_20_2 --> null
t10_i1_30_3 --> null
```

PD（Placement Driver）Server

> 整个 TiDB 集群的元信息管理模块，负责存储每个 TiKV 节点实时的数据分布情况和集群的整体拓扑结构，提供 TiDB Dashboard 管控界面，并为分布式事务分配事务 ID。PD 不仅存储元信息，同时还会根据 TiKV 节点实时上报的数据分布状态，下发数据调度命令给具体的 TiKV 节点，可以说是整个集群的“大脑”。此外，PD 本身也是由至少 3 个节点构成，拥有高可用的能力。建议部署奇数个 PD 节点。

调度需求

1. 作为一个分布式高可用存储系统，必须满足的需求，包括几种：

- 副本数量不能多也不能少
- 副本需要根据拓扑结构分布在不同属性的机器上
- 节点宕机或异常能够自动合理快速地进行容灾

2. 作为一个良好的分布式系统，需要考虑的地方包括：

- 维持整个集群的 Leader 分布均匀
- 维持每个节点的储存容量均匀
- 维持访问热点分布均匀
- 控制负载均衡的速度，避免影响在线服务
- 管理节点状态，包括手动上线/下线节点

> 满足第一类需求后，整个系统将具备强大的容灾功能。满足第二类需求后，可以使得系统整体的资源利用率更高且合理，具备良好的扩展性。

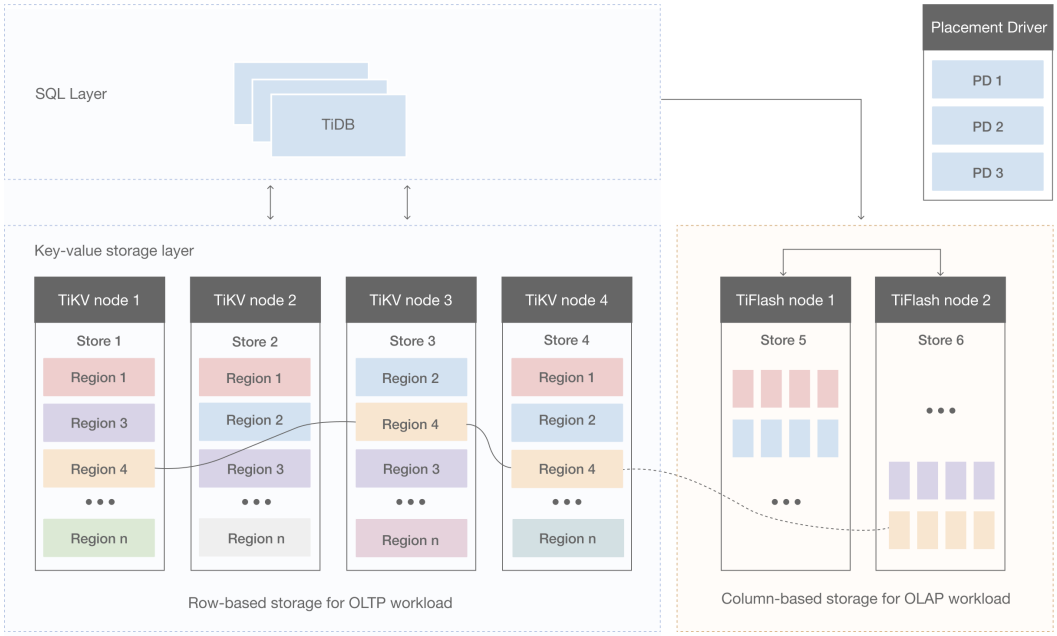
调度操作

- 增加一个副本
- 删除一个副本
- 将 Leader 角色在一个 Raft Group 的不同副本之间 transfer（迁移）。

信息收集

- 每个 TiKV 节点会定期向 PD 汇报节点的状态信息
- 每个 Raft Group 的 Leader 会定期向 PD 汇报 Region 的状态信息

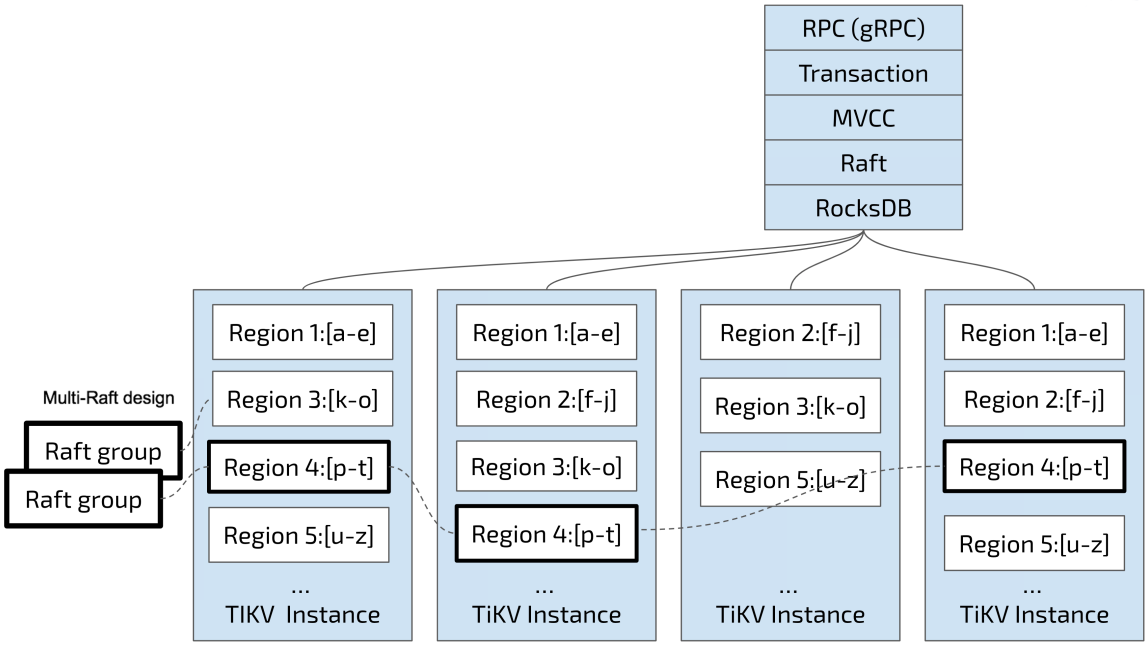
存储节点



TiKV Server

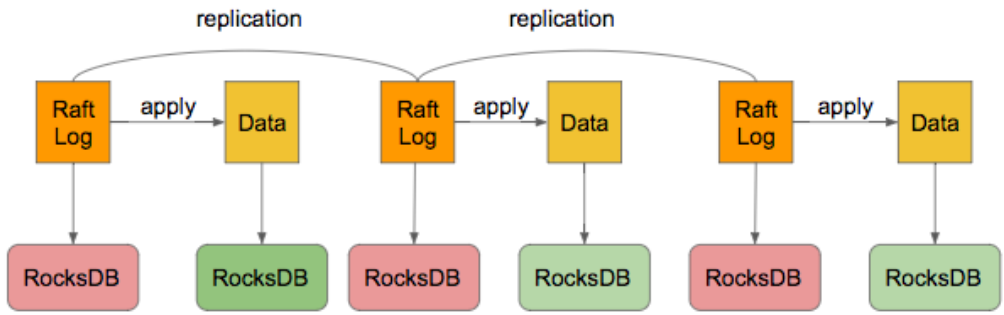
> 负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。TiKV 的 API 在 KV 键值对层面提供对分布式事务的原生支持，默认提供

> 了 SI (Snapshot Isolation) 的隔离级别，这也是 TiDB 在 SQL 层面支持分布式事务的核心。TiDB 的 SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为对 TiKV API 的实际调用。所以，数据都存储在 TiKV 中。另外，TiKV 中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。



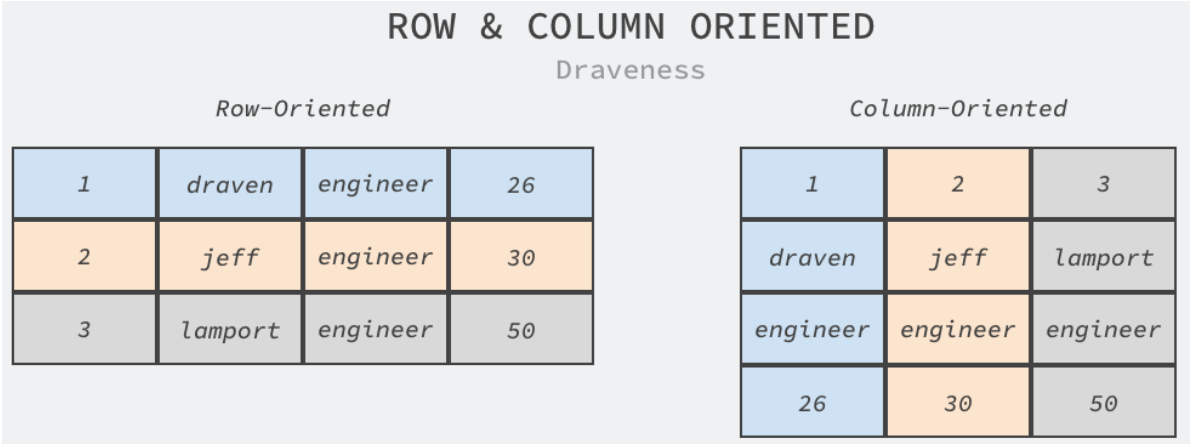
PingCAP

TiKV Architecture



TiFlash

> TiFlash 是一类特殊的存储节点。和普通 TiKV 节点不一样的是，在 TiFlash 内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。



- 列式存储可以满足快速读取特定列的需求，在线分析处理往往需要在上百列的宽表中读取指定列分析；
- 列式存储就近存储同一列的数据，使用压缩算法可以得到更高的压缩率，减少存储占用的磁盘空间；

RocksDB

> RocksDB 作为 TiKV 的核心存储引擎，用于存储 Raft 日志以及用户数据。每个 TiKV 实例中有两个 RocksDB 实例，一个用于存储 Raft 日志（通常被称为 raftdb），另一个用于存储用户数据以及 MVCC 信息（通常被称为 kvdb）。kvdb 中有四个 ColumnFamily：raft、lock、default 和 write：

- raft 列：用于存储各个 Region 的元信息。仅占极少量空间，用户可以不必关注。
- lock 列：用于存储悲观事务的悲观锁以及分布式事务的一阶段 Prewrite 锁。当用户的事务提交之后，lock cf 中对应的数据会很快删除掉，因此大部分情况下 lock cf 中的数据也很少（少于 1GB）。如果 lock cf 中的数据大量增加，说明有大量事务等待提交，系统出现了 bug 或者故障。
- write 列：用于存储用户真实的写入数据以及 MVCC 信息（该数据所属事务的开始时间以及提交时间）。当用户写入了一行数据时，如果该行数据长度小于 255 字节，那么会被存储 write 列中，否则的话该行数据会被存入到 default 列中。由于 TiDB 的非 unique 索引存储的 value 为空，unique 索引存储的 value 为主键索引，因此二级索引只会占用 writecf 的空间。

- default 列：用于存储超过 255 字节长度的数据。

内存占用

> 为了提高读取性能以及减少对磁盘的读取，RocksDB 将存储在磁盘上的文件都按照一定大小切分成 block（默认是 64KB），读取 block 时先去内存中的 BlockCache 中查看该块数据是否存在，存在的话则可以直接从内存中读取而不必访问磁盘。
>
> BlockCache 按照 LRU 算法淘汰低频访问的数据，TiKV 默认将系统总内存大小的 45% 用于 BlockCache，用户也可以自行修改 `storage.block-cache.capacity` 配置设置为合适的值，但是不建议超过系统总内存的 60%。
>
> 写入 RocksDB 中的数据会写入 MemTable，当一个 MemTable 的大小超过 128MB 时，会切换到一个新的 MemTable 来提供写入。TiKV 中一共有 2 个 RocksDB 实例，合计 4 个 ColumnFamily，每个 ColumnFamily 的单个 MemTable 大小限制是 128MB，最多允许 5 个 MemTable 存在，否则会阻塞前台写入，因此这部分占用的内存最多为 4 x 5 x 128MB = 2.5GB。这部分占用内存较少，不建议用户自行更改。

空间占用

- 多版本：RocksDB 作为一个 LSM-tree 结构的键值存储引擎，MemTable 中的数据会首先被刷到 L0。L0 层的 SST 之间的范围可能存在重叠（因为文件顺序是按照生成的顺序排列），因此同一个 key 在 L0 中可能存在多个版本。当文件从 L0 合并到 L1 的时候，会按照一定大小（默认是 8MB）切割为多个文件，同一层的文件的范围互不重叠，所以 L1 及其以后的层每一层的 key 都只有一个版本。
- 空间放大：RocksDB 的每一层文件总大小都是上一层的 x 倍，在 TiKV 中这个配置默认是 10，因此 90% 的数据存储在最后一

层，这也意味着 RocksDB 的空间放大不超过 1.11（L0 层的数据较少，可以忽略不计）。

- TiKV 的空间放大：TiKV 在 RocksDB 之上还有一层自己的 MVCC，当用户写入一个 key 的时候，实际上写入到 RocksDB 的是 key + commit_ts，也就是说，用户的更新和删除都是会写入新的 key 到 RocksDB。TiKV 每隔一段时间会删除旧版本的数据（通过 RocksDB 的 Delete 接口），因此可以认为用户存储在 TiKV 上的数据的实际空间放大为，1.11 加最近 10 分钟内写入的数据（假设 TiKV 回收旧版本数据足够及时）。

compact

> RocksDB 中，将内存中的 MemTable 转化为磁盘上的 SST 文件，以及合并各个层级的 SST 文件等操作都是在后台线程池中执行的。后台线程池的默认大小是 8，当机器 CPU 数量小于等于 8 时，则后台线程池默认大小为 CPU 数量减一。通常来说，用户不需要更改这个配置。如果用户在一个机器上部署了多个 TiKV 实例，或者机器的读负载比较高而写负载比较低，那么可以适当调低 `rocksdb/max-background-jobs` 至 3 或者 4。

WriteStall（写停顿）

> RocksDB 的 L0 与其他层不同，L0 的各个 SST 是按照生成顺序排列，各个 SST 之间的 key 范围存在重叠，因此查询的时候必须依次查询 L0 中的每一个 SST。为了不影响查询性能，当 L0 中的文件数量过多时，会触发 WriteStall 阻塞写入。
>
> 如果用户遇到了写延迟突然大幅度上涨，可以先查看 Grafana RocksDB KV 面板 WriteStall Reason 指标，如果是 L0 文件数量过多引起的 WriteStall，可以调整下面几个配置到 64。

```
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
```

数据测试（Bookshop）

> Bookshop 是一个虚拟的在线书店应用，你可以在 Bookshop 当中便捷地购买到各种类别的书，也可以对你看过的书进行点评。

```
tiup demo bookshop prepare
```

参数	简写	默认值	解释
`--host`	`-H`	`127.0.0.1`	数据库地址
`--port`	`-P`	`4000`	数据库端口
`--user`	`-U`	`root`	数据库用户
`--password`	`-p`	无	数据库用户密码
`--db`	`-D`	`bookshop`	数据库名称

参数	默认值	解释
`--users`	`10000`	指定在 `users` 表生成的数据行数
`--authors`	`20000`	指定在 `authors` 表生成的数据行数
`--books`	`20000`	指定在 `books` 表生成的数据行数
`--orders`	`300000`	指定在 `orders` 表生成的数据行数
`--ratings`	`300000`	指定在 `ratings` 表生成的数据行数

`books` 表

> 该表用于存储书籍的基本信息。

字段名	类型	含义
id	bigint(20)	书籍的唯一标识
title	varchar(100)	书籍名称
type	enum	书籍类型（如：杂志、动漫、教辅等）
stock	bigint(20)	库存
price	decimal(15,2)	价格
published_at	datetime	出版时间

`authors` 表

> 该表用于存储作者的基本信息。

字段名	类型	含义
id	bigint(20)	作者的唯一标识
name	varchar(100)	姓名
gender	tinyint(1)	生理性别 (0: 女, 1: 男，NULL: 未知)
birth_year	smallint(6)	生年
death_year	smallint(6)	卒年

`users` 表

> 该表用于存储使用 Bookshop 应用程序的用户。

字段名	类型	含义
id	bigint(20)	用户的唯一标识
balance	decimal(15,2)	余额
nickname	varchar(100)	昵称

`ratings` 表

> 该表用于存储用户对书籍的评分记录。

字段名	类型	含义
book_id	bigint	书籍的唯一标识（关联至 [books]）
user_id	bigint	用户的唯一标识（关联至 [users]）
score	tinyint	用户评分 (1-5)
rated_at	datetime	评分时间

`book_authors` 表

> 一个作者可能会编写多本书，一本书可能需要多个作者同时编写，该表用于存储书籍与作者之间的对应关系。

字段名	类型	含义
book_id	bigint(20)	书籍的唯一标识（关联至 [books]）
author_id	bigint(20)	作者的唯一标识（关联至 [authors]）

`orders` 表

> 该表用于存储用户购买书籍的订单信息。

字段名	类型	含义
id	bigint(20)	订单的唯一标识
book_id	bigint(20)	书籍的唯一标识（关联至 [books]）
user_id	bigint(20)	用户唯一标识（关联至 [users]）
quantity	tinyint(4)	购买数量
ordered_at	datetime	购买时间

数据库初始化

如果你希望手动创建 Bookshop 应用的数据库表结构，你可以运行以下 SQL 语句：

```sql
CREATE DATABASE IF NOT EXISTS `bookshop`;

DROP TABLE IF EXISTS `bookshop`.`books`;
CREATE TABLE `bookshop`.`books` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int(11) DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`authors`;
```

```sql
CREATE TABLE `bookshop`.`authors` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `name` varchar(100) NOT NULL,
  `gender` tinyint(1) DEFAULT NULL,
  `birth_year` smallint(6) DEFAULT NULL,
  `death_year` smallint(6) DEFAULT NULL,
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`book_authors`;
CREATE TABLE `bookshop`.`book_authors` (
  `book_id` bigint(20) NOT NULL,
  `author_id` bigint(20) NOT NULL,
  PRIMARY KEY (`book_id`,`author_id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`ratings`;
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint NOT NULL,
  `user_id` bigint NOT NULL,
  `score` tinyint NOT NULL,
  `rated_at` datetime NOT NULL DEFAULT NOW() ON
UPDATE NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED,
  UNIQUE KEY `uniq_book_user_idx`
(`book_id`,`user_id`)
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
ALTER TABLE `bookshop`.`ratings` SET TIFLASH
REPLICA 1;

DROP TABLE IF EXISTS `bookshop`.`users`;
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `balance` decimal(15,2) DEFAULT '0.0',
  `nickname` varchar(100) UNIQUE NOT NULL,
  PRIMARY KEY (`id`)
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

```
49
50 DROP TABLE IF EXISTS `bookshop`.`orders`;
51 CREATE TABLE `bookshop`.`orders` (
52   `id` bigint(20) AUTO_RANDOM NOT NULL,
53   `book_id` bigint(20) NOT NULL,
54   `user_id` bigint(20) NOT NULL,
55   `quality` tinyint(4) NOT NULL,
56   `ordered_at` datetime NOT NULL DEFAULT
   CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
57   PRIMARY KEY (`id`) CLUSTERED,
58   KEY `orders_book_id_idx` (`book_id`)
59 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
   COLLATE=utf8mb4_bin
```