# TiDB学习笔记

[TiDB学习笔记 1](#_Toc55608084)

[1. 概述 1](#_Toc449681343)

[2. TiKV存储 3](#_Toc799378008)

[3. TiDB计算 4](#_Toc488484824)

[3.1 关系模型到TiKV的Key-Value模型的映射 4](#_Toc134454487)

[3.2 SQL查询和运算 5](#_Toc623766365)

[4. TiDB分布式调度 6](#_Toc1773615548)

[4.1 调度的基本操作及集群信息收集 6](#_Toc2083494876)

[4.2 调度的策略 7](#_Toc430032950)

[4.3 调度的实现 7](#_Toc1281318495)

[5. Range分区 7](#_Toc153933349)

[5.1 分区和分表概念和区别 7](#_Toc1587485655)

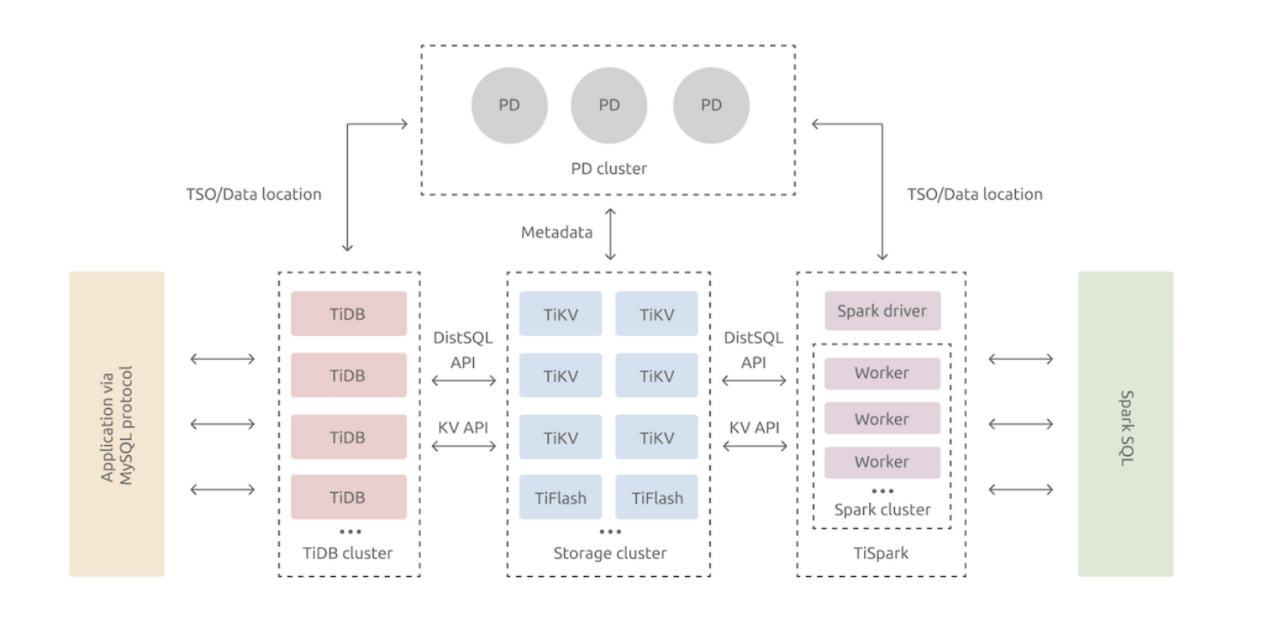
[5.2 TiDB的Range分区 8](#_Toc534573257)

## 1. 概述

TiDB作为一个高性能的分布式数据库，具有以下特点：

* 高度兼容 MySQL：大多数情况下，无需修改代码即可从 MySQL 轻松迁移至 TiDB，分库分表后的 MySQL 集群亦可通过 TiDB 工具进行实时迁移
* 分布式事务：100% 支持标准的 ACID 事务
* 一站式 HTAP 解决方案：作为典型的 OLTP 行存数据库，同时兼具强大的 OLAP 性能
* 云原生 SQL 数据库：“为云而设计的数据库，支持公有云、私有云和混合云
* 水平弹性扩展：可实现 TiDB 的水平扩展，按需扩展吞吐或存储，轻松应对高并发、海量数据场景
* 金融级高可用：基于 Raft 的多数派选举协议可以提供金融级的 100% 数据强一致性保证，且在不丢失大多数副本的前提下，可以实现故障的自动恢复，无需人工介入

TiDB的整体架构图如下：



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

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

TiKV Server：负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。TiKV 的 API 在 KV 键值对层面提供对分布式事务的原生支持，默认提供了 SI (Snapshot Isolation) 的隔离级别，这也是 TiDB 在 SQL 层面支持分布式事务的核心。TiDB 的 SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为对 TiKV API 的实际调用。所以，数据都存储在 TiKV 中。另外，TiKV 中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。

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

在进行测试设计之前，需要充分理解TiDB如何工作，主要从以下几个方面入手：

* TiKV如何存储KV数据（TiKV存储）
* TiDB如何将关系数据保存到TiKV，以及如何查询数据（TiDB计算）
* TiDB分布式系统如何合理的调度（TiDB分布式调度）
* TiDB的Range分区的概念、存储原理及如何使用

## 2. TiKV存储

<https://pingcap.com/blog-cn/tidb-internal-1/>

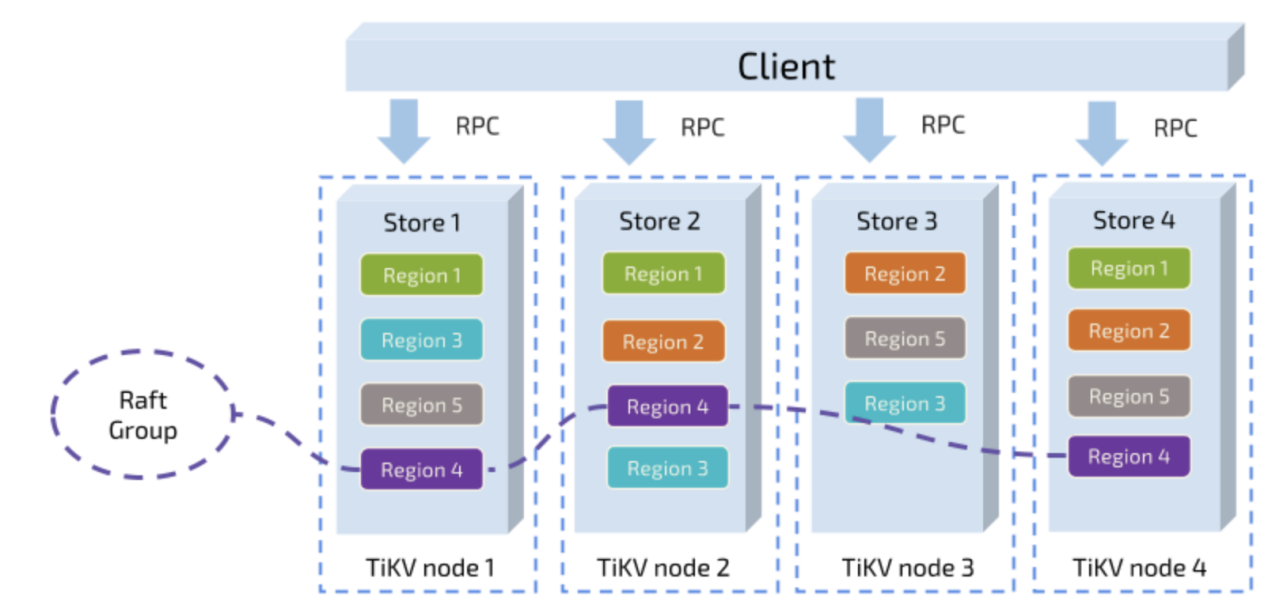
TiKV的存储结构：

* 一个巨大的 Map，也就是存储的是 Key-Value pair
* 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是我们可以 Seek 到某一个 Key 的位置，然后不断的调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value
* 数据持久化依赖RocksDB

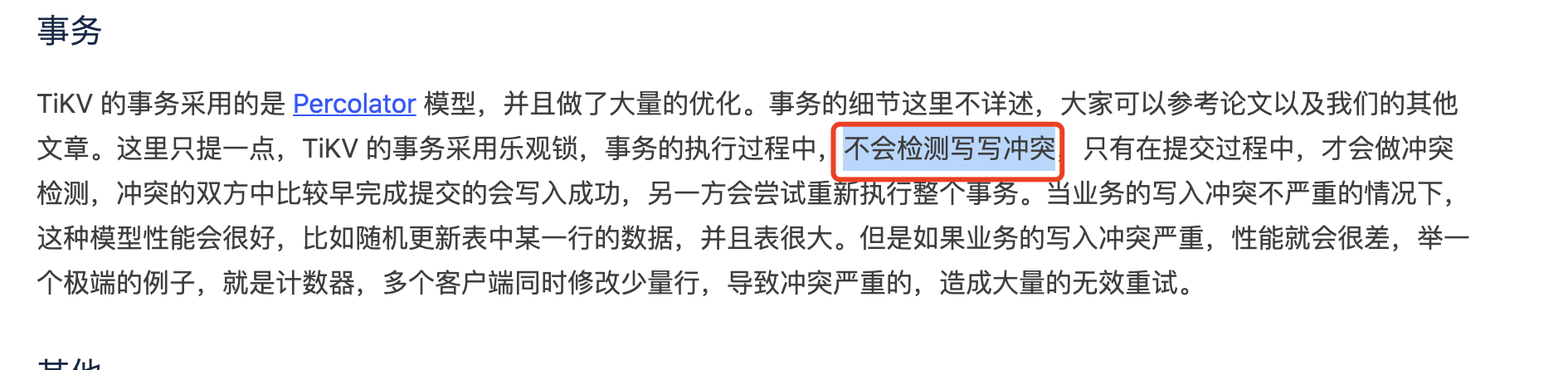
TiKV的分布式实现：

* 基于RAFT协议的改进版本
* 以Region为单位，利用RAFT来做数据复制（可以类比Kafka以Topic的分区为单位来做数据复制）；TiKV将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，我们将每一段叫做一个 Region
* 所有的读和写都是通过 Leader 进行，再由 Leader 复制到其他的 Follower副本（跟Kafka也类似）
* 为实现存储的水平扩展和负载均衡，以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多
* 为提高性能和规避可能的死锁，引入多版本控制机制（MVCC）；当用户通过一个 Key + Version 来获取 Value 的时候，可以将 Key 和 Version 构造出 MVCC 的 Key，也就是 Key-Version。然后可以直接 Seek(Key-Version)，定位到第一个大于等于这个 Key-Version 的位置
* 分布式事务：支持乐观事务模型和悲观事务模型；其中乐观事务模型只在提交过程中，才会做冲突检测，冲突的双方中比较早完成提交的会写入成功，另一方会尝试重新执行整个事务

以下是多个副本的存储和查询示例图：



另外，这里发现一处参考文档的Typo：



## 3. TiDB计算

<https://pingcap.com/blog-cn/tidb-internal-2/>

### 3.1 关系模型到TiKV的Key-Value模型的映射

一个关系型数据库，对外体现的功能主要就是存储和操作。

对于一个关系模型的Table 来说，需要存储的数据包括三部分：

* 表的元信息：每个 Database/Table 都被分配了一个唯一的 ID，这个 ID 作为唯一标识，并且在编码为 Key-Value 时，这个 ID 都会编码到 Key 中，再加上 m\_ 前缀。这样可以构造出一个 Key，Value 中存储的是序列化后的元信息
* Table 中的 Row：每行数据按照如下规则进行编码成 Key-Value pair，Key: tablePrefix{tableID}\_recordPrefixSep{rowID}、Value: [col1, col2, col3, col4]；每一行分配一个 RowID（如果表有整数型的 Primary Key，那么会用 Primary Key 的值当做 RowID）
* 索引数据：对于Unique Index，会按照如下规则编码成 Key-Value pair，Key: tablePrefix{tableID}\_indexPrefixSep{indexID}\_indexedColumnsValue、Value: rowID；对于非Unique Index，会按照如下规则编码成 Key-Value pair，Key: tablePrefix{tableID}\_indexPrefixSep{indexID}\_indexedColumnsValue\_rowID、Value: null

而对于数据的操作需求，主要考虑 Insert/Update/Delete/Select 这四种语句：

* 对于 Insert 语句，需要将 Row 写入 KV，并且建立好索引数据。
* 对于 Update 语句，需要将 Row 更新的同时，更新索引数据（如果有必要）。
* 对于 Delete 语句，需要在删除 Row 的同时，将索引也删除。
* 对于Select语句，对于快速获取一行数据，通过构造出某一个或者某几个 Key，定位到这一行，就能利用 TiKV 提供的 Seek 方法快速定位到这一行数据所在位置。对于扫描全表的需求，够映射为一个 Key 的 Range，从 StartKey 扫描到 EndKey，就可以简单的通过这种方式获得全表数据

### 3.2 SQL查询和运算

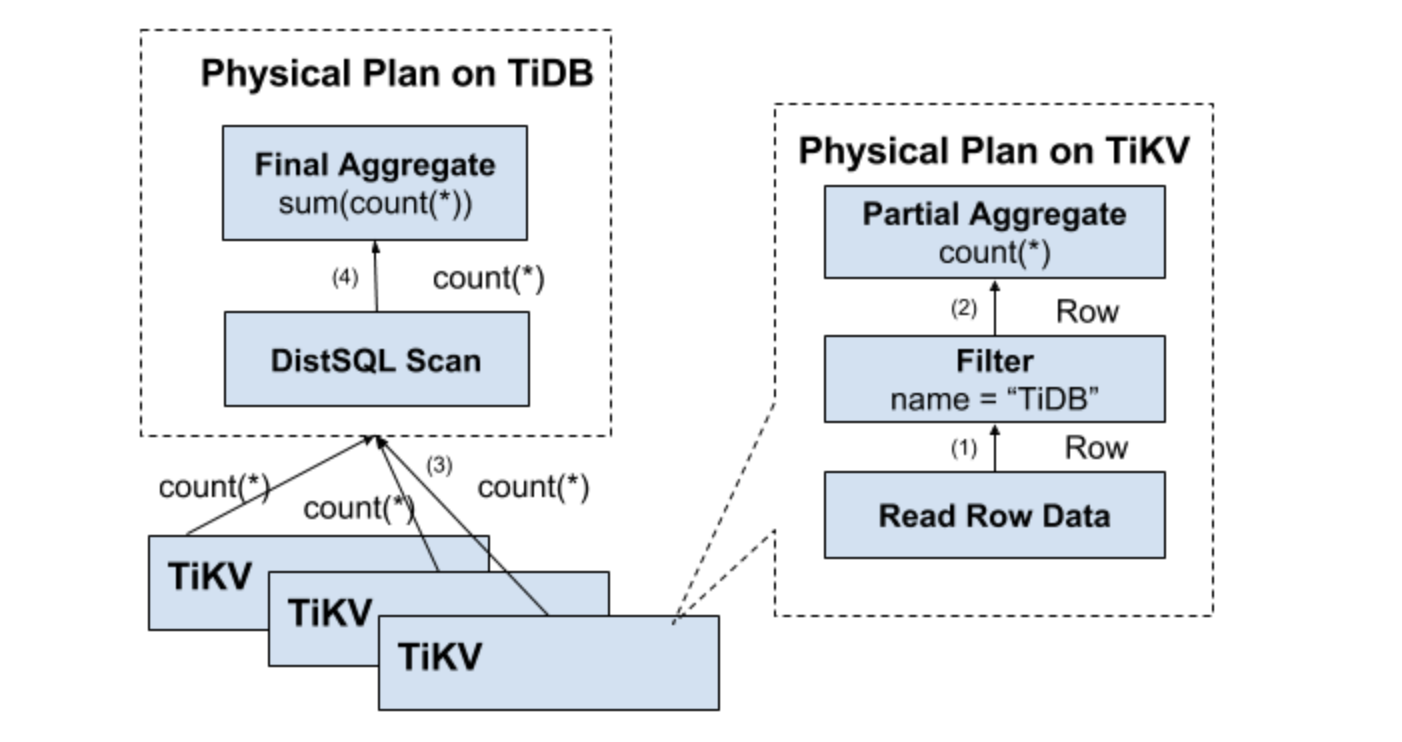
假设要执行Select count(\*) from user where name="TiDB" 这条语句，可以将该操作转换为 KV 操作流程：

* 构造出 Key Range：一个表中所有的 RowID 都在 [0, MaxInt64) 这个范围内，那么我们用 0 和 MaxInt64 根据 Row 的 Key 编码规则，就能构造出一个 [StartKey, EndKey) 的左闭右开区间
* 扫描 Key Range：根据上面构造出的 Key Range，读取 TiKV 中的数据
* 过滤数据：对于读到的每一行数据，计算 name="TiDB" 这个表达式，如果为真，则向上返回这一行，否则丢弃这一行数据
* 计算 Count：对符合要求的每一行，累计到 Count 值上面

但上述方面性能开销很大，TiDB做了以下优化：

* 将count计算下沉到存储节点，以避免大量的 RPC 调用
* 将 Filter 也下推到存储节点进行计算，这样只需要返回有效的行，避免无意义的网络传输
* 将聚合函数、GroupBy 也下推到存储节点，进行预聚合，每个节点只需要返回一个 Count 值即可，再由 tidb-server 将 Count 值 Sum 起来

分布式SQL运算的示意图如下：



## 4. TiDB分布式调度

PD是TiDB集群的大脑，用于对分布式系统的整体状况进行把控和调整。

### 4.1 调度的基本操作及集群信息收集

通过Raft协议的AddReplica、RemoveReplica、TransferLeader 这三个命令实现以下操作：

* 增加一个 Replica
* 删除一个 Replica
* 将 Leader 角色在一个 Raft Group 的不同 Replica 之间 transfer

而调度依赖于整个集群信息的收集，调度需要知道每个 TiKV 节点的状态以及每个 Region 的状态。TiKV 集群会向 PD 汇报两类消息：

* 每个 TiKV 节点会定期向 PD 汇报节点的整体信息，一方面 PD 通过心跳包检测每个 Store 是否存活，以及是否有新加入的 Store，主要包括：磁盘容量、承载的 Region 数量、数据写入速度、 标签信息和是否过载等
* 每个 Raft Group 的 Leader 会定期向 PD 汇报信息，用于汇报这个 Region 的状态，主要包括：Leader 的位置、Followers 的位置、掉线 Replica 的个数和数据写入/读取的速度等

除此之外，PD 还可以通过管理接口接受额外的信息，用来做更准确的决策。比如运维人员主动通过 PD 的管理接口通知 PD 某个 Store 不可用，PD 就可以马上判断需要将这个 Store 上面的 Region 都调度走。

### 4.2 调度的策略

调度策略主要考虑以下方面：

* Region 的 Replica 数量的数量是否正确：当 PD 通过某个 Region Leader 的心跳包发现这个 Region 的 Replica 数量不满足要求时，需要通过 Add/Remove Replica 操作调整 Replica 数量
* 一个 Raft Group 中的多个 Replica 不在同一个位置：给节点配置 lables 并且通过在 PD 上配置 location-labels 来指明哪些 lable 是位置标识，需要在 Replica 分配的时候尽量保证不会有一个 Region 的多个 Replica 所在结点有相同的位置标识
* 副本在 Store 之间的分布均匀分配
* Leader 数量在 Store 之间均匀分配
* 访问热点数量在 Store 之间均匀分配
* 各个 Store 的存储空间占用大致相等
* 控制调度速度，避免影响在线服务：避免调度对线上服务造成太大影响
* 支持手动下线节点

### 4.3 调度的实现

PD 不断的通过 Store 或者 Leader 的心跳包收集信息，获得整个集群的详细数据，并且根据这些信息以及调度策略生成调度操作序列。

每次收到 Region Leader 发来的心跳包时，PD 都会检查是否有对这个 Region 待进行的操作，通过心跳包的回复消息，将需要进行的操作返回给 Region Leader，并在后面的心跳包中监测执行结果。注意这里的操作只是给 Region Leader 的建议，并不保证一定能得到执行，具体是否会执行以及什么时候执行，由 Region Leader 自己根据当前自身状态来定。

## 5. Range分区

### 5.1 分区和分表概念和区别

分表就是把一张表分成N多个小表，而分区就是把一张表的数据分成N多个区块，每个分区进行连续的存储。

分表和分区的区别：

* 分区和分表的测重点不同：分表重点是存取数据时，如何提高数据库并发能力上；而分区侧重如何突破磁盘的读写能力，从而达到提高数据库性能的目的。
* 在数据处理上：分表后数据都是存放在分表里，总表只是一个外壳；而分区只不过把存放数据的文件分成了许多小块，分区后的表呢，还是一张表。

### 5.2 TiDB的Range分区

TiDB支持的分区类型包括 Range 分区和 Hash 分区。Range 分区可以用于解决业务中大量删除带来的性能问题，支持快速删除分区。Hash 分区则可以用于大量写入场景下的数据打散。

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。

Range分区可以带来的显著性能提升点：

* 可以快速删除分区及分区中的数据
* 频繁查询分区使用的列，通过分区裁剪可提升Select性能
* 使用包含时间或者日期的列，或者是其它按序生成的数据

5.2.1 Range分区管理

对于Range分区的支持管理操作有：

* 创建分区表
* 删除分区
* 清空分区
* 添加分区

对于Range分区的不支持管理操作有：

* REORGANIZE PARTITION
* 修改 Range 分区表的范围，合并分区，交换分区 都不支持

Range分区的创建或添加，可支持的场景：

* 不同数据类型上使用 Range分区：数值型类型、字符串类型、时间和日期类型、JSON类型等
* 使用特定的分区表达式或者直接使用INT类型的列来划分Range

Range分区创建或添加，约束和限制有：

* 不支持的分区类型：List 分区和 Key 分区等MySQL支持但是TiDB不支持的分区类型
* 仅能使用单列用于Range划分
* 分区表的每个组件和唯一键，必须包含分区表达式中用到的所有列；如果既没有主键，也没有唯一键，可以创建Range分区
* 添加新分区时，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
* 环境变量 tidb\_enable\_table\_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。

关于Range分区的删除，删除分区后该分区的数据也会一同被删除。测试用例设计时可以关注一下分区删除之后TiKV的GC回收是否正常。

关于Range分区的清空操作，该操作仅删除分区上的数据。

5.2.2 Range分区的数据插入、修改和删除

对异常数据插入的处理：

* 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。
* 插入一行不属于任何Range分区的数据，会报错

思考点：分区对数据插入性能是否有影响？

对于数据修改，需要考虑数据所在Range分区发生变化的情况。

对于数据删除分区中的数据，DELETE FROM PARTITION p0 应该会报错。

5.2.3 基于Range分区的数据查询和分区裁剪

分区选择示例：

SELECT \* FROM employees PARTITION (p0, p2)

说明：

* 可与where 条件，以及 ORDER BY 和 LIMIT 等选项。使用 HAVING 和 GROUP BY 等聚合选项的组合
* 也可用于INSERT INTO ... SELECT FROM 语句中

在SELECT查询时，分区裁剪的几点说明：

* 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
* 查询条件无法下推到 TiKV 的表达式，不支持分区裁剪
* 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。目前只有：unix\_timestamp、to\_days和floor(unix\_timestamp(ts))

测试用例设计的时候，可以关注一下EXPLAIN执行计划对于Range分区的分析是否合理。

5.2.4 Range分区在TiKV中的存储

在3.1节中，学习了数据库Table在没有分区的情况下如何映射到TiKV中。如果使用Range分区，确保同一个表的不同分区使用不同的tableID即可实现关系模型映射到KV模型。