TiDB

文档地址：https://www.pingcap.com/docs-cn/

TiDB 简介

TiDB 是 PingCAP 公司受 Google Spanner / F1 论文启发而设计的开源分布式 HTAP (Hybrid Transactional and Analytical Processing) 数据库，结合了传统的 RDBMS 和 NoSQL 的最佳特性。TiDB 兼容 MySQL，支持无限的水平扩展，具备强一致性和高可用性。

TiDB 特性：

- 高度兼容 MySQL （https://www.pingcap.com/docs-cn/sql/mysql-compatibility/）

  不支持的特性

  - 存储过程
  - 视图
  - 触发器
  - 自定义函数
  - 外键约束
  - 全文索引
  - 空间索引
  - 非 UTF8 字符集
  - 事务

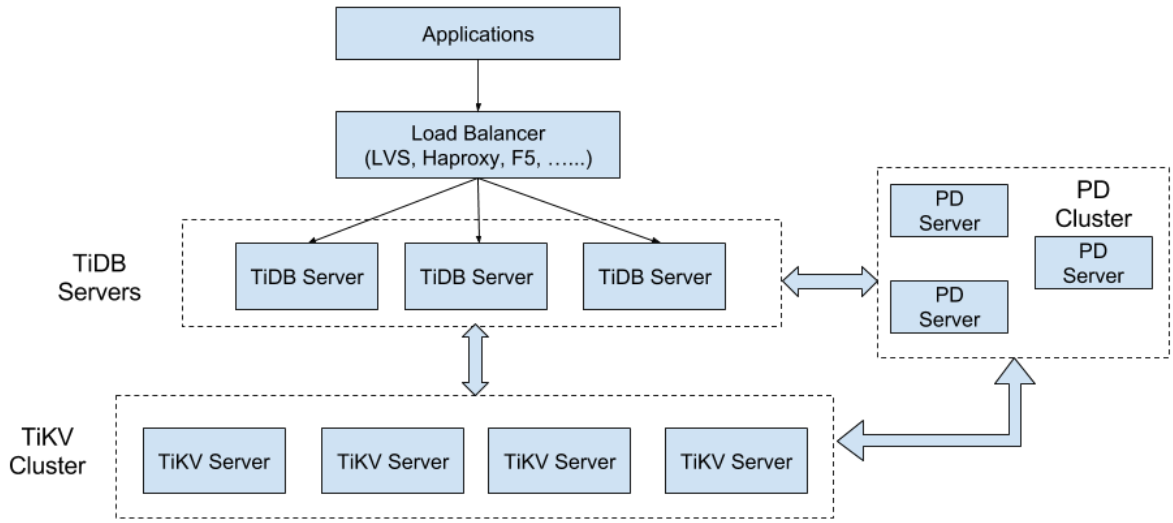
    TiDB 使用乐观事务模型，在执行 Update、Insert、Delete 等语句时，只有在提交过程中才会检查写写冲突，而不是像 MySQL 一样使用行锁来避免写写冲突。所以业务端在执行 SQL 语句后，需要注意检查 commit 的返回值，即使执行时没有出错，commit的时候也可能会出错。
- 水平弹性扩展
- 分布式事务

  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

TiDB 整体架构图



TiDB Server

TiDB Server 负责接收 SQL 请求，处理 SQL 相关的逻辑，并通过 PD 找到存储计算所需数据的 TiKV 地址，与 TiKV 交互获取数据，最终返回结果。

PD Server

Placement Driver (简称 PD) 是整个集群的管理模块，其主要工作有三个： 一是存储集群的元信息（某个 Key 存储在哪个 TiKV 节点）；二是对 TiKV 集群进行调度和负载均衡（如数据的迁移、Raft group leader 的迁移等）；三是分配全局唯一且递增的事务 ID。

TiKV Server

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 使用 Raft 协议做复制，保持数据的一致性和容灾。副本以 Region 为单位进行管理，不同节点上的多个 Region 构成一个 Raft Group，互为副本。数据在多个 TiKV 之间的负载均衡由 PD 调度，这里也是以 Region 为单位进行调度。

TiKV 存储解析

TiKV 记住两点：

1. 这是一个巨大的 Map，也就是存储的是 Key-Value pair
2. 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

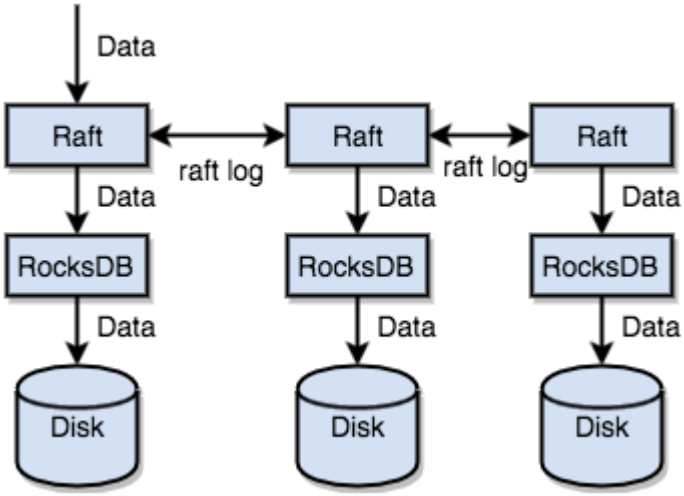
TiKV 的底层的存储引擎使用的是RocksDB ， 具体的数据的落地的工作是 由 RocksDB 实现的。

分布式的一致性协议：Raft

保证数据在多个节点的同步复制，数据一致性。 TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到 Group 的多数节点中。

Raft 是一个一致性协议，提供几个重要的功能：

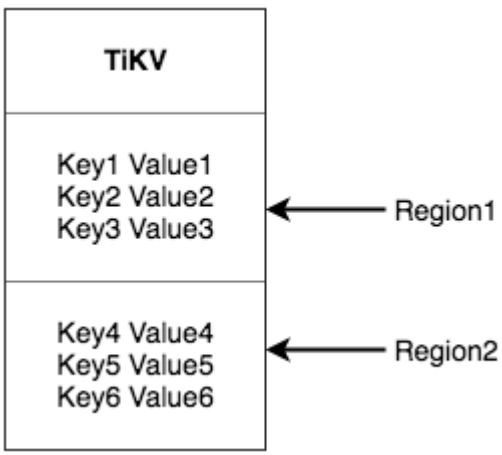
1. Leader 选举
2. 成员变更
3. 日志复制



到这里我们总结一下，通过单机的 RocksDB，我们可以将数据快速地存储在磁盘上；通过 Raft，我们可以将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，我们拥有了一个分布式的 KV，现在再也不用担心某台机器挂掉了。

Region

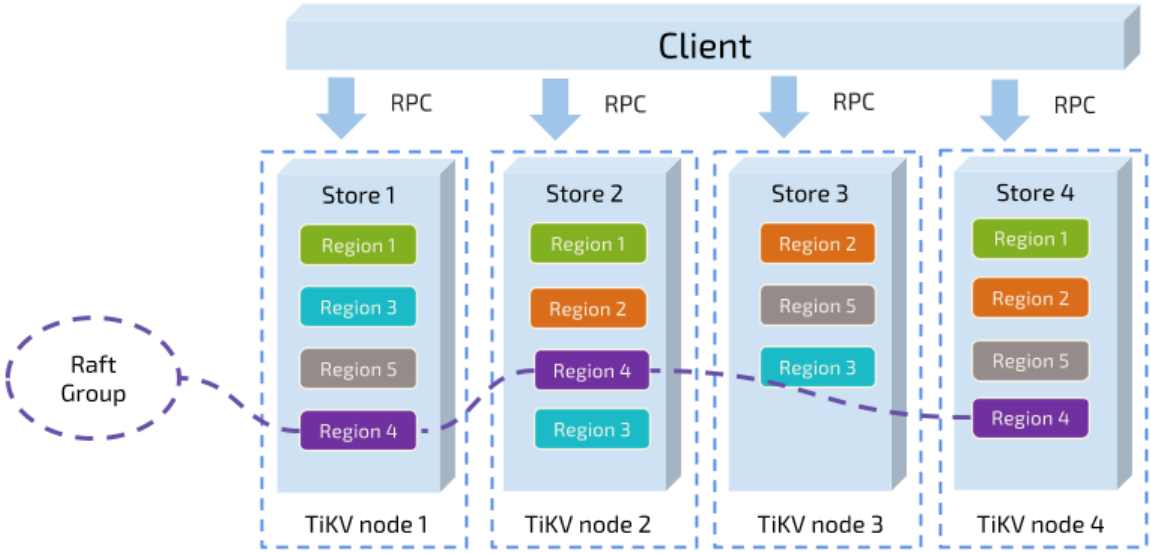
TiKV 看做一个巨大的有序的 KV Map，对于一个 KV 系统，将数据**分散在多台机器**上有两种比较典型的方案：一种是按照 Key 做 Hash，根据 **Hash 值选择对应的存储节点**；另一种是分 Range，某一段**连续的 Key 都保存在一个存储节点上**，每一段是一系列连续的 Key，我们将每一段叫做一个 **Region**，并且我们会尽量保持每个 Region 中保存的数据不超过一定的大小(这个大小可以配置，目前默认是 64mb)。每一个 Region 都可以用 StartKey 到 EndKey 这样一个左闭右开区间来描述。



TiKV 以region 为单位操作

- 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
    - pd 会来管理是region 尽可能的均匀分散在各个节点，存储容量不足时，增加节点进行region 的调度，进行水平的扩展
    - 负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）

    - 上层客户端能够访问，pd 会通过任意一个 Key 就能查询到这个 Key 在哪个 Region 中，以及这个 Region 目前在哪个节点上
- 以 Region 为单位做 Raft 的复制和成员管理
    - TiKV 是以 Region 为单位做数据的复制，也就是一个 Region 的数据会保存多个副本，Repica 之间是通过 Raft 来保持数据的一致
    - 一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。



MVCC

很多数据库都会实现多版本控制（MVCC），TiKV 也不例外。设想这样的场景，两个 Client 同时去修改一个 Key 的 Value，如果没有 MVCC，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。 TiKV 的 MVCC 实现是通过在 Key 后面添加 Version 来实现，简单来说，没有 MVCC 之前，可以把 TiKV 看做这样的：

```
Key1 -> Value
Key2 -> Value
……
KeyN -> Value
```

有了 MVCC 之后，TiKV 的 Key 排列是这样的：

```
Key1-Version3 -> Value
Key1-Version2 -> Value
Key1-Version1 -> Value
……
Key2-Version4 -> Value
Key2-Version3 -> Value
Key2-Version2 -> Value
Key2-Version1 -> Value
……
KeyN-Version2 -> Value
KeyN-Version1 -> Value
……
```

注意，对于同一个 Key 的多个版本，我们把版本号较大的放在前面，版本号小的放在后面（回忆一下 Key-Value 一节我们介绍过的 Key 是有序的排列），这样当用户通过一个 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。

事务

TiKV 的事务采用的是 Percolator 模型，并且做了大量的优化。事务的细节这里不详述，大家可以参考论文以及我们的其他文章。这里只提一点，TiKV 的事务采用乐观锁，**事务的执行过程中，不会检测写写冲突**，只有在提交过程中，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。当业务的写入冲突不严重的情况下，这种模型性能会很好，比如随机更新表中某一行的数据，并且表很大。但是如果业务的写入冲突严重，性能就会很差，举一个极端的例子，就是计数器，多个客户端同时修改少量行，导致冲突严重的，造成大量的无效重试。

TiDB 计算解析

关系模型到 Key-Value 模型的映射

假设表中有 3 行数据：

1. “TiDB”, “SQL Layer”, 10
2. “TiKV”, “KV Engine”, 20
3. “PD”, “Manager”, 30

那么首先每行数据都会映射为一个 Key-Value pair，注意这个表有一个 Int 类型的 Primary Key，所以 RowID 的值即为这个 Primary Key 的值。假设这个表的 Table ID 为 10，其 Row 的数据为：

```
t_r_10_1 --> ["TiDB", "SQL Layer", 10]
t_r_10_2 --> ["TiKV", "KV Engine", 20]
t_r_10_3 --> ["PD", "Manager", 30]
```

除了 Primary Key 之外，这个表还有一个 Index，假设这个 Index 的 ID 为 1，则其数据为：

```
t_i_10_1_10_1 --> null
t_i_10_1_20_2 --> null
t_i_10_1_30_3 --> null
```

PD-server 调度

信息收集

调度依赖于整个集群信息的收集，简单来说，我们需要知道每个 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

TiKV 节点（Store）与 PD 之间存在心跳包，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；另一方面，心跳包中也会携带这个 Store 的状态信息，主要包括：

- 总磁盘容量
- 可用磁盘容量
- 承载的 Region 数量
- 数据写入速度
- 发送/接受的 Snapshot 数量（Replica 之间可能会通过 Snapshot 同步数据）
- 是否过载
- 标签信息（标签是具备层级关系的一系列 Tag）

每个 Raft Group 的 Leader 会定期向 PD 汇报信息

每个 Raft Group 的 Leader 和 PD 之间存在心跳包，用于汇报这个 Region 的状态，主要包括下面几点信息：

- Leader 的位置
- Followers 的位置
- 掉线 Replica 的个数
- 数据写入/读取的速度