链滴

TiKV

作者：hupeng1984

原文链接：https://ld246.com/article/1681038892834

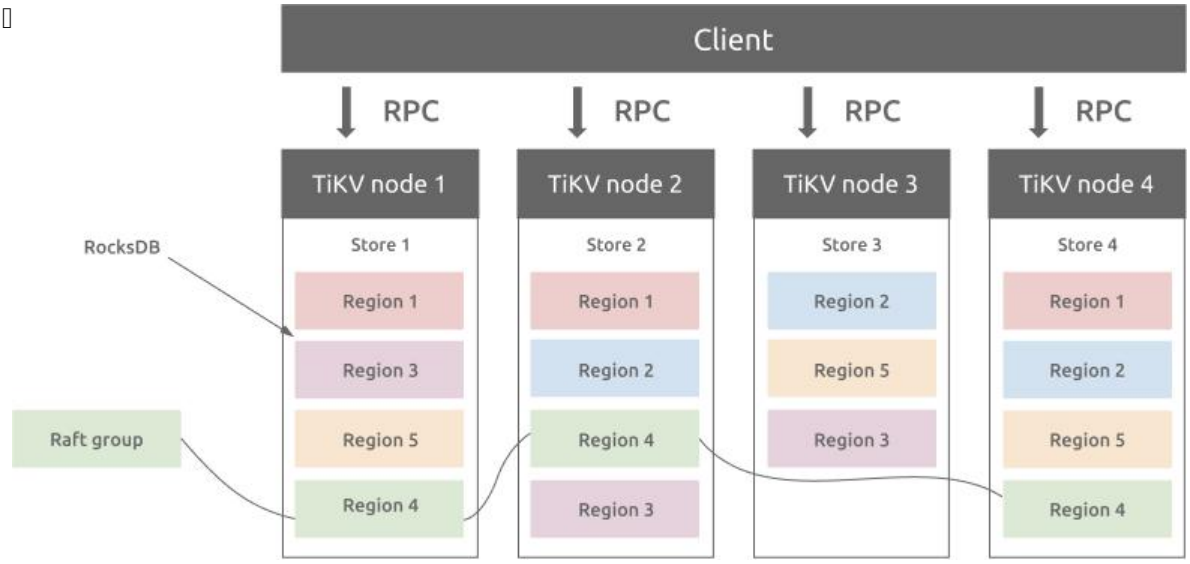
来源网站：链滴

TiKV 是一个分布式事务型的键值数据库，提供了满足 ACID 约束的分布式事务接口，并且通过RAFT 议保证了多副本数据一致性以及高可用。TiKV 作为 TiDB 的存储层，为用户写入 TiDB 的数据提供了久化以及读写服务，同时还存储了 TiDB 的统计信息数据。

整体架构

TiKV 参考 Spanner 设计了 multi-raft-group 的副本机制。

将数据按照 key 的范围划分成大致相等的切片，统称为 Region，每一个切片会有多个副本（通常是 3 个），其中一个副本是 Leader，提供读写服务。



虽然 TiKV 将数据按照范围切割成了多个 Region，但是同一个节点的所有 Region 数据仍然是不加分地存储于同一个 RocksDB 实例上，而用于 Raft 协议复制所需要的日志则存储于另一个 RocksDB 例。

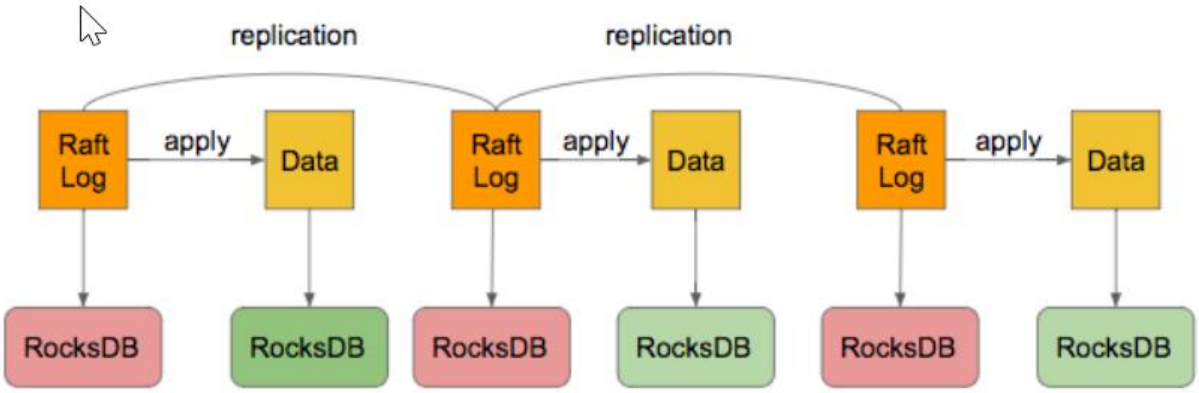
- 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 量差不多
- 以 Region 为单位做 Raft 的复制和成员管理

RocksDB

RocksDB 允许用户创建多个 ColumnFamily，这些 ColumnFamily 各自拥有独立的内存跳表以及 SST 文件，但是共享同一个 WAL 文件，这样的好处是可以根据应用特点为不同的 ColumnFamily 选择同的配置，但是又没有增加对 WAL 的写次数。

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）。

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，否则的话该数据会被存入到 default 列中。

- default 列：用于存储超过 255 字节长度的数据。

TiKV持久化[1]

分布式事务[3]

Raft与Multi Raft[4]

读写与 Coprocessor[5]

SQL执行流程[6]

1.

TiKV持久化

TiKV架构

RocksDB 作为 TiKV 的核心存储引擎，用于存储 Raft 日志以及用户数据。[2]

TiKV作用

- 数据持久化
- 分布式一致性
- MVCC
- 分布式事务
- Coprocessor 协同处理器

数据持久化和读取

Rocksdb

单机数据存储引擎

- 高性能key-value数据库
- 持久化机制，保证性能和安全行
- LSM存储引擎
- 良好的支持范围查询
- 为存储TB级别数据到本地FLASH或RAM的应用服务设计
- 针对中小键值优化-可以存储在FLASH或者直接存储在内存
- 性能随CPU数量线性提升，对多核系统友好

写入操作

1. 先写WAL（Write Ahead Log）预写日志，再写入内存MemTable

WAL关键参数：sync_log=true，避免先写操作系统缓存，直接写磁盘

2. 写入数据量到达write_buffer_size大小限制时，MemTable转变成immutable MemTable（不可）

immutable MemTable：写到磁盘SST文件的一个中间状态，避免直接写磁盘，减少IO等待

3. 重启开启一个MemTable

4. 将immutable MemTable刷到磁盘上形成SST文件，该immutable MemTable销毁

5. immutable MemTable达到5个就会触发rocksdb的流控write stall，限制写入MemTable的速度

6. 可以优化存储或者调高immutable数量来提高写入速度
7. WAL中数据都持久化到SST file之后，会被删除

□□

Level0：immutable Memtable的复制，默认达到4个后，向下一层做Compaction

为什么Compaction?

用于去除相同key的多条记录。当我们对一个key进行多次update/delete时，RocksDB会产生多条记
。

从Level0~LevelN：4个合并成1个，压缩并对key进行排序

immutable 数量达到 min-write-buffer-number-to-merge 之后就会触发 flush。

L0 层上包含的文件，是由内存中的memtable dump到磁盘上生成的，单个文件内部按key有序，文
之间无序。可能存在多个相同的key在 L0 层

L1~L6层上的文件都是按照key有序的。也就是每层只会存在一个key。

每一层都会切分成多个SST文件，每个SST文件都是键值对文件

对于每个文件使用二分法进行查找键值信息

删除时：不管数据在哪，直接写入delete key

查询操作

□□

查询性能不如B+树

Block Cache：最近最常读数据的缓存

依次按照写入最新时间查找MemTable，再按从磁盘中的Level 0依次往后查找到SST文件

根据查找的KEY判断是否在SST的min_key和max_key中间；

引入布隆过滤器bloom filter判断key在不在这个SST中，如果key不在，则查找下一个SST文件

如果数据在该SST文件，则二分法查找

□□

Column Family属于RocksDb的数据分片技术，可以将数据的键值对按照不同的属性分配给不同的C
，可以让某些内存和SST文件中存的都是相同类型的数据，可以极大地增加读写的效率、提升数据压
率；

落数的时候会自带CF1、CF2、default 来决定落入哪个分片中；

内存和SST文件都按照CF分了，但是WAL没有按照CF区分。

2. RocksDB 作为 TiKV 的核心存储引擎，用于存储 Raft 日志以及用户数据。

3.

分布式事务

TiDB采用Google Percolator事务模型来解决分布式事务的问题。

Google Percolator事务模型

Percolator事务分为两个阶段：预写（Pre-write）和提交（Commit），本质上相当于一个加强的2P
。

1. 从PD组件获取事务开始时间
2. 将要修改的数据读出到TiDB Server的内存中，进行修改，commit
3. 进行两阶段提交
4. 第一阶段：prewrite会将修改的数据和锁信息写入到TiKV节点中；
5. 第二阶段，在Write CF中写入数据；
6. 从PD获取TSO，作为事物结束时间；
7. 锁清理： 往Lock CF中落入一条数据清理锁。

什么是列族

RocksDB的每个键值对都与唯一一个列族（column family）结合。如果没有指定Column Family，值对将会结合到“default” 列族。

列族提供了一种从逻辑上给数据库分片的方法。他的一些有趣的特性包括：

- 支持跨列族原子写。意味着你可以原子执行Write({cf1, key1, value1}, {cf2, key2, value2})。
- 跨列族的一致性视图。
- 允许对不同的列族进行不同的配置
- 即时添加 / 删除列族。两个操作都是非常快的。

事务的过程

假设有这样一个事务：

```
begin;
update person set name = 'Frank' where id = 3;
commit;
```

事务开始

在事务开始时，begin时，TiDB会从PD中获取事务开始的时间戳TSO，假设TSO=100。

修改数据

然后TiDB将需要修改的数据读取到内存中，在内存中完成数据的修改。

TiDB在内存中修改数据的时候，不会将锁信息写入TiKV，此时其他会话无法感知锁的存在，是乐观务。

乐观事务与悲观事务

乐观事务模型就是直接提交，遇到冲突就回滚。

悲观事务模型就是在真正提交事务前，先尝试对需要修改的资源上锁，只有在确保事务一定能够执行功后，才开始提交。

对于乐观事务模型来说，比较适合冲突率不高的场景，因为直接提交大概率会成功，冲突是小概率事，但是一旦遇到事务冲突，回滚的代价会比较大。

悲观事务的好处是对于冲突率高的场景，提前上锁的代价小于事后回滚的代价，而且还能以比较低的价解决多个并发事务互相冲突导致谁也成功不了的场景。不过悲观事务在冲突率不高的场景并没有乐事务处理高效。

事务提交

在commit的时候进入两阶段提交。

预写Prewrite

在第一阶段，TiDB会写三个列族到TiKV中：

- Default列族：记录带有事务开始时间戳标记（100）的修改后的数据
- Lock列族：记录锁信息，在第一行数据加写锁（W），这是一把主锁（pk），并且记录下其他相关信息
- Write列族：预留用来存放提交信息

此时，其他会话会感知到写锁的存在，这样其他会话不会进行id=3的数据的读、写操作。

注意：

当用户写入的数据长度小于255字节时，数据会被存储在Write列族；

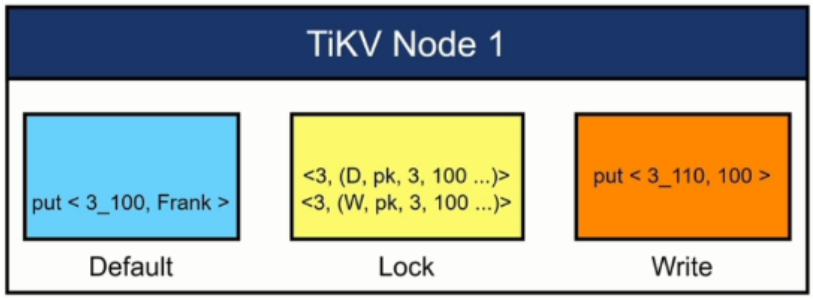
当用户写入的数据长度大于255字节时，数据会被存储在Default列族。

提交Commit

在第二阶段，TiDB会从PD中获取事务提交的时间戳TSO，假设TSO=110。

TiDB在Write列族中写入提交信息，包括事务提交的时间戳（110）和事务开始的时间戳（100）

完成后在Lock列族中记录一条锁清理的数据，表示写锁已经被释放。



此时，其他会话可进行id=3的数据的读、写操作。

如何处理分布式事务

假设有这样一个事务：

```
begin;
update person set name = 'Jack' where id = 1;
update person set name = 'Candy' where id = 2;
commit;
```

并且两条数据分布在TiKV中的两个实例上。

TiDB会按照事务的处理过程进行处理。

在预写阶段，在事务中的第一条数据（节点1，id=1）上加主锁（pk），在节点2上记录的是附加锁@1）表示的是主锁在id=1的那条记录那里。

此时，节点1和节点2上都有写锁，其他会话不能对这些数据进行读、写操作。

在提交阶段，节点1和节点2都写入提交信息和清理写锁。

当节点1提交成功，节点2提交失败，那么节点2上的写锁不会被清除。

后续在读取数据的时候，发现有写锁存在，并且是附加锁（@1），此时需要判断事务是否提交：

根据附加锁的指向找到主锁，发现主锁已成功提交，则可判断自己在提交阶段出现了问题。

因为主锁是成功提交的，所以附加锁这里只需要补充提交即可，继续写入提交信息，清理附加锁，Default列族中的数据变成最终数据。

MVCC

从前面的过程中知道，事务已预写、未提交的时候，数据不能进行读、写操作，这带来一个问题，就是**也会被阻塞。**

MVCC的引入是为了解决读操作被阻塞的，因为在修改中的、还未提交的数据，还不确定最后是否提，那么读取修改前的数据应该是被允许的。

MVCC机制下，读操作按最近一次提交记录读取，无需关心锁信息，写操作需要先检查当前是否已存其他写锁。

假设有两个事务：

```
--事务1
begin (start_ts = 100)
update person set name = 'Jack' where id = 1;
update person set name = 'Candy' where id = 2;
commit; (commit_ts = 110)

--事务2
begin (start_ts = 115)
update person set name = 'Tim' where id = 1;
update person set name = 'Jerry' where id = 4;
```

此时，事务1已成功提交，事务2未提交。

假设在时间戳TSO=120的时候，有会话读取数据：

```
select * from person where id in (1,2,4);
```

此时id=1和id=4的记录有锁信息但无提交信息，属于在事务中的数据，如果阻塞读操作，那么此时id=1和id=4的记录都是无法读取的。

引入MVCC后。

读id=1的数据时，从Write列族中发现最近一次提交是TSO=100，读操作可以读取TSO=100的数据。

写id=1的数据时，当前锁是TSO=115，因为TSO=115的锁还未提交，所以写被阻塞。

读id=2的数据时，从Write列族中发现最近一次提交是TSO=100，读操作可以读取TSO=100的数据。

写id=2的数据时，当前无锁，所以可以写。

读id=4的数据时，从Write列族中发现最近一次提交是TSO=80，读操作可以读取TSO=80的数据。

写id=4的数据时，当前锁是TSO=115的附加锁，因为主锁还未提交，所以附加锁也还在事务中，写阻塞。

4.
Raft与Multi Raft

raft group：region及其副本（上图以3副本为例）

Multi raft：由多个raft goup组成

名词解释

Leader

- 集群的管理者
- 所有读写流量都是走Leader
- Leader会周期性向follower发出心跳信息
- Leader会将写的数据以日志的方式传递给其他follower
- 当写入的数据成员过半，就认为写入成功

Follower

- 被管理者
- 对其他的服务作出响应
- 接受Leader的日志
- 如果长时间没收到Leader的通知信息，就会将自己角色转换为后选择candidate，发起投票，票多升级为Leader

Region

- 是按照Key排序的连续的有序集合
- 当Region插入达到96MB后会另起一个新Region
- 初始化时，Region内的数据是连续的，Region中间也是连续的，左闭右开区间

region1: [1,1000), region2:[1000-2000),region3:[2000,3000)

- 随着数据的修改(例如UPDATE等)，Region大小会发生变化，当数据涨到144M的时候会自动分裂；

当Region过小的时候会进行Region的合并；(分裂和合并的大小可以自定义)

- 一个Region构成一个Raft group

多个Region会形成多个Raft Group--Multi Raft

- 如果一个TiKV中的Region超过5W，会影响性能

Raft 日志复制

raft log利用region ID+日志的顺序ID来作为唯一标识

所有客户端的读写流量都通过leader

Leader日志写入的过程

1. Propose，Leader将写请求转化为Raft Log的形式；

2. Append：日志持久化， Leader在Propose后会将写入请求转换为写入日志，存到日志文件中；(志组成:region_id + 序号+数据组成，日志存储在本地的RocksDB实例中)

3. Replicate：Leader将日志分发给follower -> follower收到日志后写入到本地存储中（Append）- 返回消息给Leader确认；

4. Commited：当多数节点都返回了Append成功的消息后，Leader认为写入成功；此时可以保证Raf rocksdb的日志不丢失；（区别于用户的commit）

5. Apply：Leader将raft log（rocksdb raft中）的操作apply至rocksdb kv中的数据，写入完成（一TiKV中实际上有两个RocksDB，一个用于存储Raft Log，一个用于存储KV信息；）

ps：若大多数（超过一半的）follower没有持久化成功（Replicate），则日志不会被Apply

Raft- Leader选举

leader周期性向follower发出含有统治信息的心跳（参数**heartbeat time interval**）；若follower长间收不到统治信息（参数**election timeout**），某个region将会转换为candidate（候选者），并发投票重新选取leader

election timeout：控制收不到统治信息时，发起选举的阈值；一般用于初始化时进行选举（集群刚创建）

heartbeat time interval：控制发出心跳的周期，当收不到心跳的时间且超过**election timeout**的值，发起选举；一般用于集群运行中

任期的概念 term

● Raft 把时间分割成任意长度的任期（term），任期用连续的整数标记。

● 每一段任期从一次选举开始，一个或者多个 candidate 尝试成为 leader 。如果一个 candidate 赢选举，然后他就在该任期剩下的时间里充当 leader 。在某些情况下，一次选举无法选出 leader 。在种情况下，这一任期会以没有 leader 结束；一个新的任期（包含一次新的选举）会很快重新开始。

● Raft 保证了在任意一个任期内，最多只有一个 leader

任期的作用

● 不同的服务器节点观察到的任期转换的次数可能不同，在某些情况下，一个服务器节点可能没有看到 leader 选举过程或者甚至整个任期全程。

● 任期在 Raft 算法中充当逻辑时钟的作用，这使得服务器节点可以发现一些过期的信息比如过时的 le der 。

● 每一个服务器节点存储一个当前任期号，该编号随着时间单调递增。

● 服务器之间通信的时候会交换当前任期号；

● 如果一个服务器的当前任期号比其他的小，该服务器会将自己的任期号更新为较大的那个值。

● 如果一个 candidate 或者 leader 发现自己的任期号过期了，它会立即回到 follower 状态。（所以老leader如果发生了网络分区，后来接收到新leader的心跳的时候，比拼完任期之后，会自动变成foll wer。）

● 如果一个节点接收到一个包含过期的任期号的请求，它会直接拒绝这个请求。

选举的发起流程

● 每个Node启动的时候，初始化Role都是Follower，并且启动计时器，超时还没接收到消息（可以是eader的AppendEntries RPC，也可以是 Candidate的Vote RPC）

● Raft集群的启动选举

Raft 使用一种心跳机制来触发 leader 选举。当服务器程序启动时，他们都是 follower 。一个服务器点只要能从 leader 或 candidate 处接收到有效的 RPC 就一直保持 follower 状态。Leader 周期性向所有 follower 发送心跳（不包含日志条目的 AppendEntries RPC）来维持自己的地位。

如果一个 follower 在一段选举超时时间内没有接收到任何消息，它就假设系统中没有可用的 leader 然后开始进行选举以选出新的 leader。

● 选举过程

要开始一次选举过程，follower 先增加自己的当前任期号并且转换到 candidate 状态。然后投票给己并且并行地向集群中的其他服务器节点发送 RequestVote RPC（让其他服务器节点投票给它）。

Candidate 会一直保持当前状态直到以下三件事情之一发生：

(a) 它自己赢得了这次的选举（收到过半的投票）

(b) 其他的服务器节点成为 leader

(c) 一段时间之后没有任何获胜者。这些结果会在下面的章节里分别讨论。

当一个 candidate 获得集群中过半服务器节点针对同一个任期的投票，它就赢得了这次选举并成为 le der 。

对于同一个任期，每个服务器节点只会投给一个 candidate ，按照先来先服务（first-come-first-ser ed）的原则。

要求获得过半投票的规则确保了最多只有一个 candidate 赢得此次选举。

一旦 candidate 赢得选举，就立即成为 leader 。然后它会向其他的服务器节点发送心跳消息来确定己的地位并阻止新的选举。

图示说明

如图，当leader宕机，TiKV node 3长时间收不到心跳并率先达到阈值，自身转化为candidate，进下一个term并发起选举

向其他region发起请求，进行投票；其他follower接收请求并同意比自己term大的请求（term=2<te m=3）

如图，若碰巧所有Tikv node都转为candidate企图发起选取，election timeout参数将会初始为随机，减小每个TiKV node达到阈值的记录；若还是有多个candidate重复该过程，整个过程发起了多次取

election timeout

- raft-election-timeout-ticks：设置 election timeout有多少个相对时间单位（ticks，默认1s）
- raft-base-tick-interval：设置每个相对时间单位（ticks）有多长时间

heartbeat time interval

- raft-heartbeat-ticks：设置 heartbeat time interval有多少个相对时间单位（ticks，默认1s）
- raft-base-tick-interval：设置每个相对时间单位（ticks）有多长时间

假设raft-election-timeout-ticks=5，raft-base-tick-interval=1s，则election timeout=5*1=5s

ps：election timeout ticks不能小于heartbeat ticks

5.

读写与 Coprocessor

数据写入

1. 用户提交写请求
2. TiDB Server接收请求
3. TiDB Server向PD申请TSO，获得Region元数据信息（哪一个leader，在哪个TiKV上）
4. TiDB Server的写请求会由TiKV中的raftstore pool线程池接收进行处理
5. 执行Raft日志复制过程（Propose、Append、Replicate、Commited、Apply）

完成数据写入操作。

数据读取

Index Read

本质上属于一种判断操作是否apply至rocksdb的机制。

用户在读取数据的时候，读取请求会提交到TiDB Server，TiDB Server从PD获取数据的元数据信息包括所在的TiKV、Region、Leader等等信息。

问题一

TiDB Server拿到元数据后去TiKV节点找leader Region读取数据，此时不能完全保证在读取数据的时原来的leader还是leader。因为这个时间间隔内，原leader可能失效，重新选举了leader，也就是TiD Server从PD处拿到的leader信息在真正要读数据的这个时间间隔内失效了，变成follower了，就不读数据了。

TiDB采用了读取时进行心跳检测机制，TiDB Server拿到leader信息后，到leader region真正读数据时候，由该region发起一次心跳检测，检测当前region是否还是leader，如果是，就直接读取数据，果不是，就不能读取数据，重新获取leader region，从新的leader读取数据。

问题二

抛开MVCC机制，只考虑Raft协议，用户的读操作需要读取之前的请求已经提交的数据。在读取时，果有写操作存在，则读取被阻塞。什么时候可读?

TiDB引入了ReadIndex和ApplyIndex。当读取数据时，先获取到要读取的数据所在的位置（index）然后寻找Raft日志复制阶段中commited阶段的index，确保commited的index大于要读取的数据的inex后，记录commited的index作为ReadIndex值，寻找Raft日志复制阶段中的apply阶段的index作ApplyIndex值。只有当ApplyIndex的值等于ReadIndex的值时，可以确定当前index的数据已经提持久化，从而知道要读取的数据所在的index的值已确定持久化，此时读取的数据就是提交后的数据。

当某一会话提交操作（TSO=10:00），但其操作未apply至rocksdb；此时ReadIndex=1-95，ApplyIdex=1-92，rocksdb中的数据为1-92

当另一会话企图读取被修改的数据（TSO=10:05），此时raftstore pool将日志commit至1-97，但applypool将操作应用至1-93（该案例假设没有MVCC机制，若有则读取的是之前的数据即1-93）；即ReadIndex=1-97并阻塞commit，使其等待apply操作至1-97

当apply至1-95时，事务的commit完成

当apply至1-97时，即ApplyIndex等于ReadIndex，此时根据raft log的顺序性，事务1-95肯定已被久化至rocksdb

总结：利用日志的顺序性，通过判断后面的日志是否已apply推断出要读取的日志是否apply

ReadIndex一定大于要读的raft log，当ApplyIndex等于ReadIndex时判断

Lease Read

Lease Read也可以叫Local Read。

TiDB Server从PD获得TSO时的leader还是正常的，那么leader会发送心跳，时间间隔是heartbeat tie interval，而follower会等待election timeout，如果达到election timeout还没收到心跳，才会进重新选举leader。这意味着，即便TiDB Server从PD获取到TSO后leader出现问题，至少需要等待electon timeout这么长时间集群才会重新选举leader，也就是在election timeout这个时间范围内，还是以从原leader处读取数据的。这就是Lease Read。

Follower Read

follower的数据与leader的数据是一致的，所以读实际上可以从follower读的，也就是读写分离是可的，**前提是：**保证follower的数据与leader的数据是线性一致的。

要保证数据的线性一致，Follower Read是在follower节点上按照Index Read的方式读取数据的。先从eader节点获取到leader当前的commit index，然后需要follower节点的apply index等于获取到的cmmit index后才能进行数据的读取。

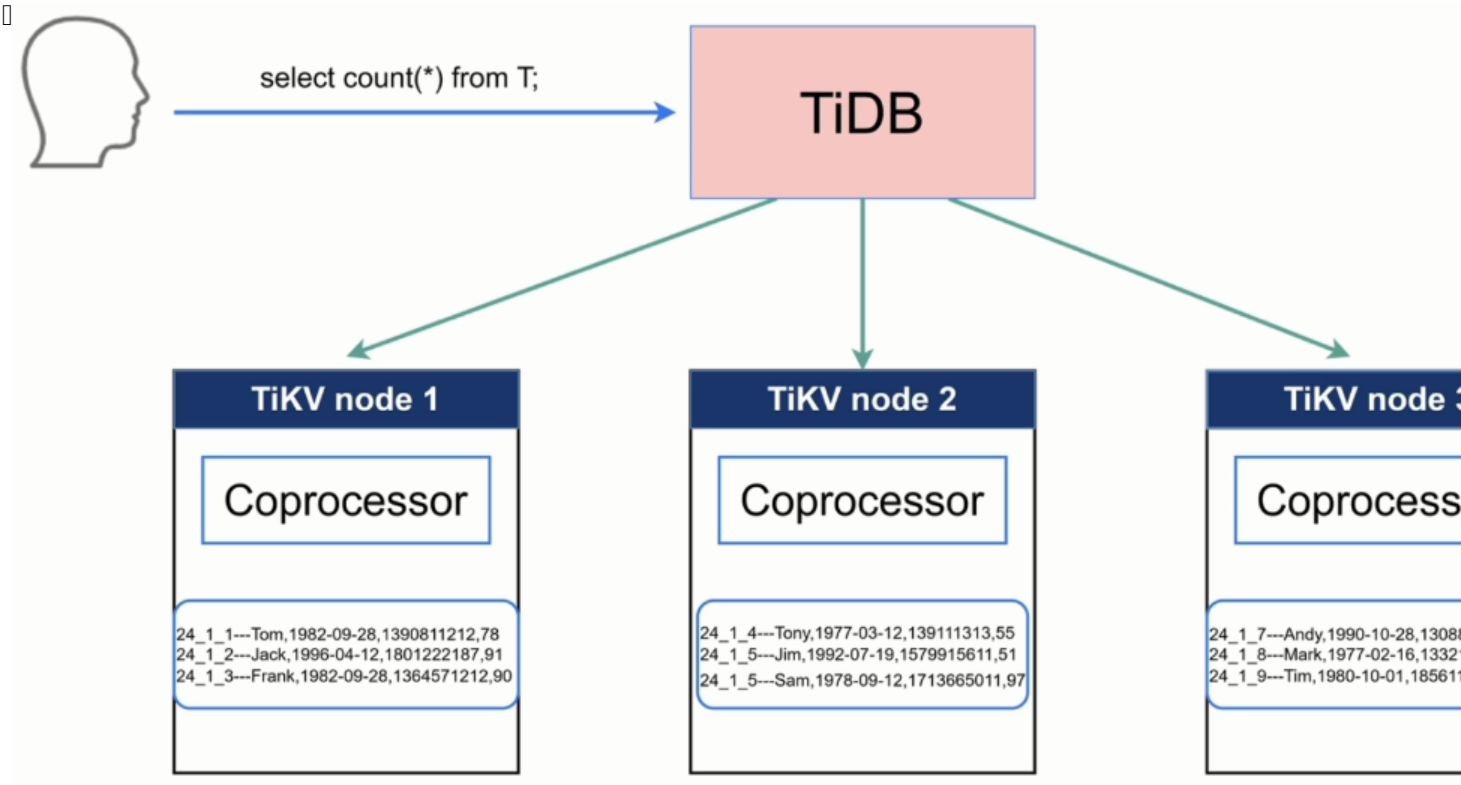
Coprocessor

协同处理器，可以实现：执行物理算子，算子下推：聚合、全表扫描、索引扫描等。

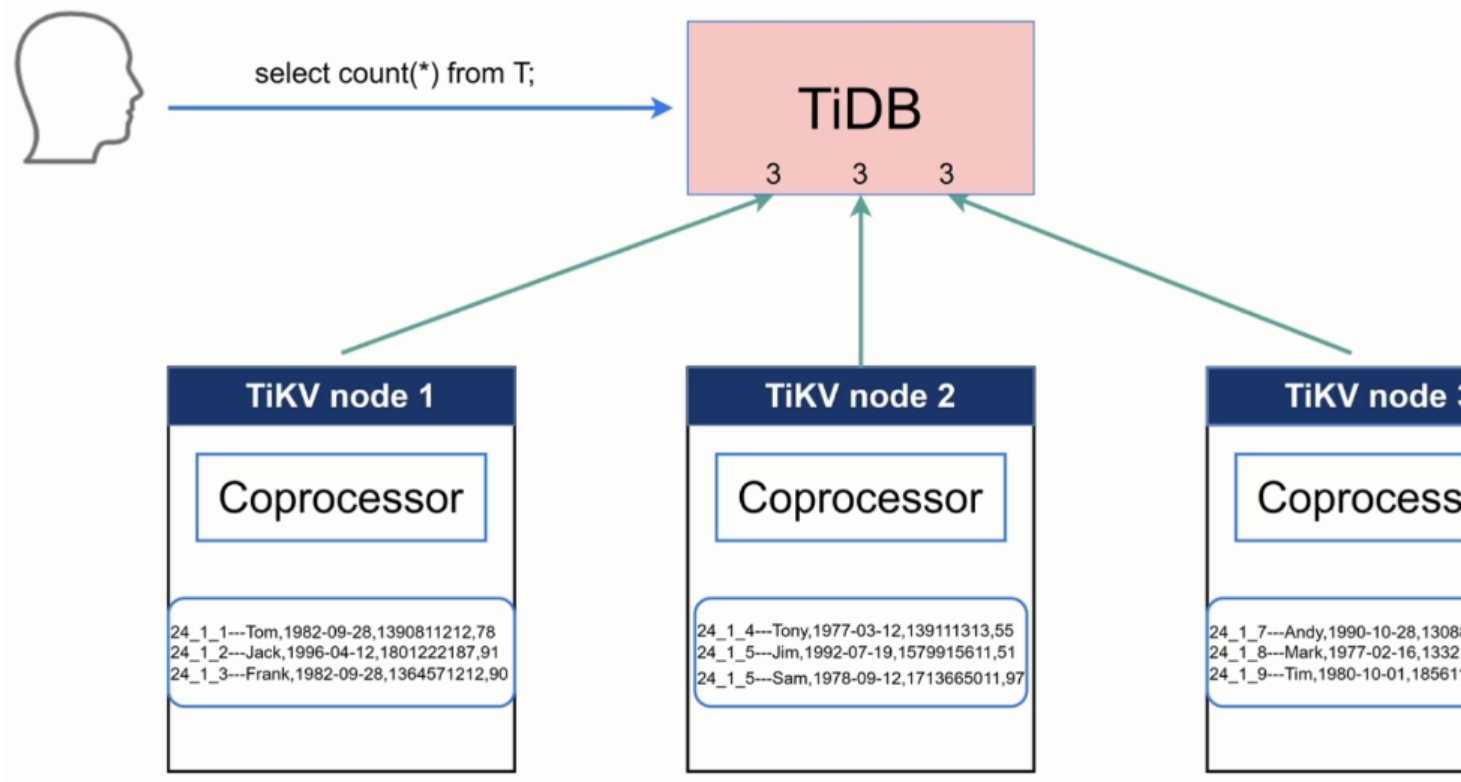
比如，TiDB Server接收到用户的请求。

如果不引入Coprocessor机制，那么TiKV的所有数据会发送到TiDB Server做运算，一方面增加网络宽，一方面TiDB Server的负载会很高。

引入Coprocessor后，可以实现count算子下推。



每个TiKV节点计算自己节点上的count，将值传到TiDB Server，TiDB Server只需要对各个节点的进一次处理就好了。



6.

SQL执行流程

DML语句读流程

TiDB Server中的Protocol Layer首先接收用户的SQL请求，TiDB Server会到PD获取TSO，同时经过析模块Parse对SQL进行词法分析和语法分析，再由Compile模块进行编译，最终进行Execute。

SQL的Parse与Compile

Protocol Layer接收到SQL请求后，会由PD Client与PD进行交互，获取TSO。

Parse模块会对SQL进行词法分析、语法分析，生成抽象语法树AST。

Compile分成几个阶段：

- 预处理preprocess：

检测SQL的合法性、绑定信息等，判断SQL是否是点查SQL（仅查询一条数据，比如按Key查询），果是点查语句则不需要执行optimize优化，直接执行即可。

- 优化optimize：
  - 逻辑优化，根据关系代数、等价交换等对SQL进行逻辑变换，比如外连接尝试转内连接等；
  - 物理优化，基于逻辑优化的结果和统计信息，选择最优的算子。

SQL的Execute

Compile的产物是执行计划，有了执行计划之后就由Executor进行执行。

Executor先获取information schema，information schema可以预先从TiKV加载到TiDB Server。后Executor需要从PD获取Region的元数据信息，为减少TiDB Server与PD交互带来的网络开销、延，TiKV Client的region Cache可以缓存Region的元数据信息，后续就可以直接使用。

Executor执行数据读取有两种类型，点查SQL，直接读取KV；复杂SQL，需要通过DistSQL将复杂SQ转换成对单表的简单查询SQL。

TiKV接收到读取请求后，会创建一个数据快照snapshot，所有查询SQL都会进入UnifyRead Pool线池，然后从RocksDB KV读取数据。

当数据读取完成后，数据通过TiKV Client返回给TiDB Server。

由于TiDB实现了算子下推，对于聚合操作，TiKV完成cop task，TiDB Server完成root task，也就是TiDB Server中还需要对下推算子的聚合结果进行汇总。

DML语句写流程

写流程，会先经过一次读流程，将数据读取到缓存MemBuffer中，然后再进行数据修改，最后进行阶段提交进行数据写入。

执行

Transaction执行两阶段提交。

Transaction按行读取memBuffer的数据进行数据写入，事务包含两个TSO，一个是事务开始TSO，个是事务提交TSO。

写请求发送给TiKV的Scheduler，Scheduler是接收并发处理的，需要负责协调冲突写入（两个会话行写相同的Key），冲突写入采用分配latch，谁获得latch谁执行写操作的方式解决冲突。无冲突的写，交Raftstore，完成Raft日志写入过程，涉及本地append、replicate、commited、apply等过程。

DDL语句流程

TiDB Server接收到用户的DDL请求，start job模块将操作写入队列，由TiDB Server的Owner角色的orkers模块负责执行队列中的DDL语句。

workers从job queue队列中获取待执行的DDL语句，执行，执行完成后将其放到history queue队列

。

执行

TiDB支持Online DDL，也就是DDL不锁表，不阻塞读、写。

同一时刻，只有一个TiDB Server的角色是Owner角色的，只有Owner角色的TiDB Server的workers可以执行DDL。schema load负责将最新的表结构信息加载到TiDB Server。

Compile生成的执行计划，交给start job，start job先检查本机节点的角色是否是Owner，如果是，可直接由本机workers直接执行，如果不是，则start job将DDL操作封装成job加入到**job queue**队列。如果DDL操作是对索引的操作，则job会加入到**add index queue**。

Owner的workers会定时扫描job queue，发现queue中有job就获取执行，执行完成后将job加入到history queue中。

Owner角色由PD协调，在TiDB Server之间是轮询切换，所以总体来说，每个TiDB Server都有机会为Owner。