

# 分布式数据库事务-Percolator

## 1. 简介

单机数据库事务是在一个节点上完成的，分布式数据库事务需要多个节点协调完成。为了满足 ACID，分布式数据库在实现事务时需要考虑更多复杂因素。

两阶段提交（two-phase commit）是一种用于实现跨多个节点的原子事务提交的算法，即确保所有节点提交或所有节点中止。


## 2. Percolator

这里介绍一个有代表性的 2PC 模型：Percolator。

### 2.1 背景

Percolator 是谷歌在 2010 发表的论文 [«Large-scale Incremental Processing Using Distributed Transactions and Notifications»](#) 介绍的分布式事务协议。

Percolator 构建在 Bigtable 之上，利用 Bigtable 的单行事务能力和 TSO 实现了 Snapshot Isolation 隔离级别的跨节点多行事务。

 从数据模型的角度，Bigtable 可以理解为是一个稀疏的，多维的持久化的键值对 Map，一个键值对的格式如下：  

```
(row:string, column:string, timestamp:int64) -> string
```

用户指定的 Percolator 中的一个 Column 在 Bigtable 中会映射到如下多个 Column：

Column	Use
<b>c:lock</b>	An uncommitted transaction is writing this cell; contains the location of primary lock
<b>c:write</b>	Committed data present; stores the Bigtable timestamp of the data
<b>c:data</b>	Stores the data itself

lock 列用于存储锁信息，write 列用于存储已提交的 start\_ts，data 列用于存储 value。

row	c1	c2
r1	123	abc

r2	321	cba
----	-----	-----

row	c1:data	c1:lock	c1:write	c2:data	c2:lock	c2:write
r1	6:	6:	6: data@5	6:	6:	6: data@5
r1	5: 123	5:	5:	5:abc	5:	5:
r2	6:	6:	6: data@5	6:	6:	6: data@5
r2	5: 321	5:	5:	5:cba	5:	5:

## 2.2 事务流程

### Begin

一个事务开启时会从 TSO 获取一个 timestamp 作为 start\_ts。

### Set

事务的所有 Write 在提交之前都会先缓存在内存，然后在提交阶段一次性写入，这种方式称为 Buffering Writes until Commit。

### Get

事务在读取时，需要先检查对应 lock 列在  $[0, \text{start\_ts})$  之间是否存在锁信息。如果锁存在，则检查锁是否过期，如果过期则 cleanup 锁，否则阻塞。如果锁不存在，先读 write 列在  $[0, \text{start\_ts})$  之间最新的 start\_ts 作为可见最大版本，再读 data 列的可见最大版本数据并返回。

### Commit

#### 1. Phase 1: Prewrite

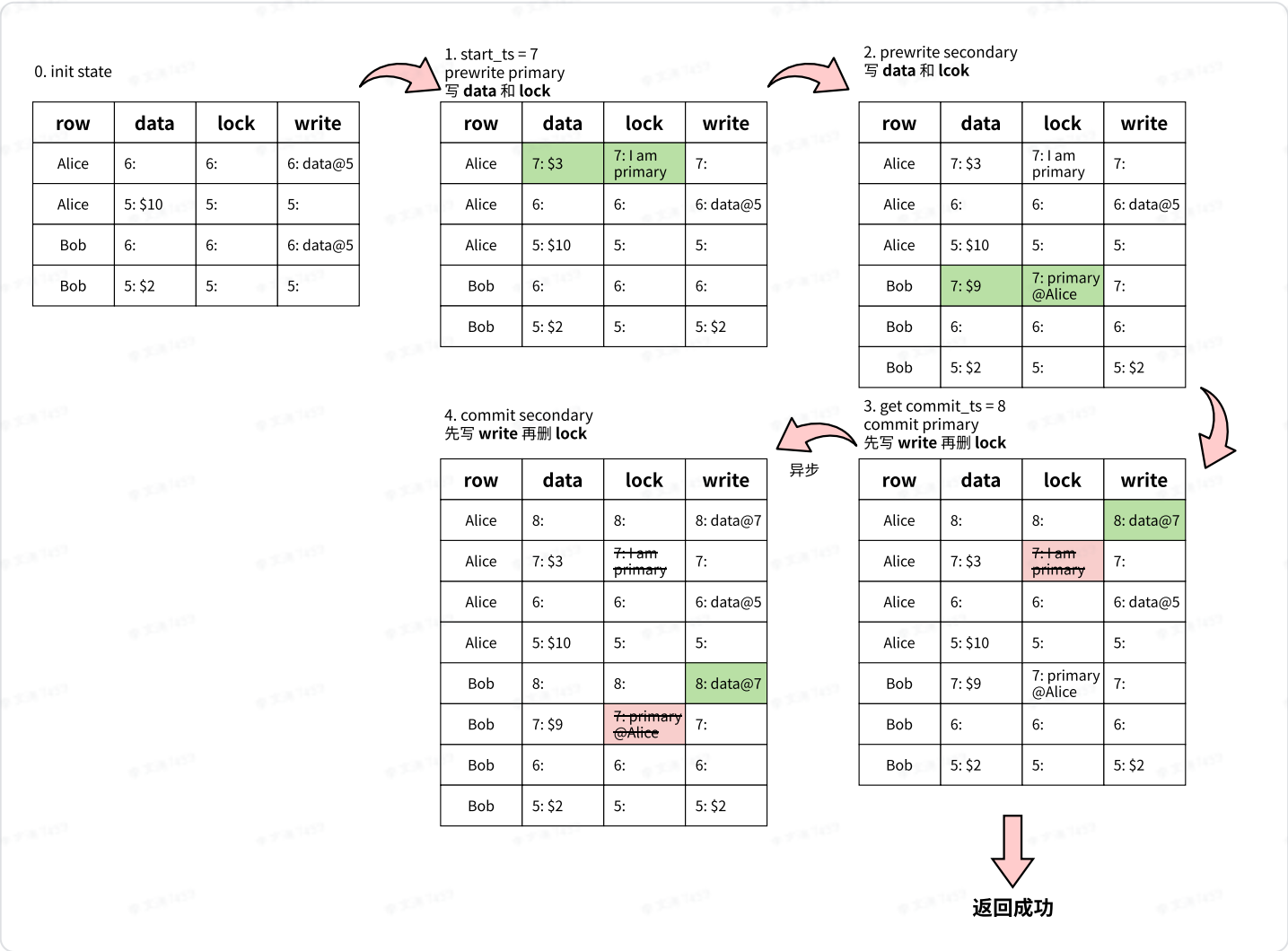
- a. 选择一个 Write 作为 primary，其它 Write 则是 secondaries；primary commit 作为事务提交的 commit point。
- b. 先 prewrite primary，成功后再 prewrite secondaries。对于每一个 Write：
  - i. 检查对应的 write 列在  $[\text{start\_ts}, \infty)$  之间是否存在数据。如果存在，则说明有 write-write conflict，直接 abort 整个事务。
  - ii. 检查对应的 lock 列在  $[0, \infty)$  之间是否存在锁信息。如果存在，直接 abort 整个事务。这种冲突处理方案避免了死锁的发生。
  - iii. 以 start\_ts 作为 timestamp，对 Write 上锁：以 {primary.row, primary.col} 作为 value，写入 lock 列。并将数据写入 data 列。由于此时 write 列尚未写入，因此数据对其它事务不可见。

#### 2. Phase 2: Commit

- a. 从 TSO 获取一个 timestamp 作为 commit\_ts。
- b. 先 commit primary , 如果失败则 abort 事务。
  - i. 检查 primary lock 是否存在，如果已经被其它事务清理（事务超时可能导致该情况），则 abort 整个事务。
  - ii. 以 commit\_ts 作为 timestamp，先以 start\_ts 作为 value 写 write 列，再删除 lock。
- c. 步骤 b 成功意味着事务已提交成功，此时可以返回事务提交成功，再异步 commit secondaries。

## 2.3 举例

下图表示 Alice 向 Bob 转 \$7 的过程：



## 2.4 一些问题

### 2.4.1 写热点行问题

### 1. prewrite 后 panic

t	T1	T2
1	begin	
2	prewrite a (lock a)	
3	panic	
4		begin
5		prewrite a (lock a failed)
6		abort

### 2. 活锁

t	T1	T2
1	begin	
2	prewrite a (lock a)	
3		begin
4		prewrite b (lock b)
5	prewrite b (lock b failed)	
6	abort	
7		prewrite a (lock a failed)
8		abort

1. 当客户端宕机后，他会遗留未开始提交的事务持有的锁，而这个锁都要到事务超时后，才能被其他事务的读操作清除，这就增加了其他并发执行的事务发生写冲突被回滚的几率。
2. 另外还存在活锁的情况，就是 T1 和 T2 相互冲突来回取消。

对于 google 的 web index 数据来说，可能出现热点行的几率较小，但是对于通用的 DBMS 来说，热点行是经常会出现的。

## 2.4.2 写阻塞读问题

初始时 a = 1, ts = 0

t	T1	T2
1	begin start_ts = 1	
2	prewrite a = 2	
3	commit_ts = 3	
4		begin start_ts = 4
5		read a = 1
6	commit a = 2, ts = 3	
7		read a = 2 ❌

假如写不阻塞读  
会破坏快照隔离

t	T1	T2
1	begin start_ts = 1	
2	prewrite a = 2 (lock a)	
3	commit_ts = 3	
4		begin start_ts = 4
5		read a (wait)
6	commit a = 2 (unlock a)	
7		read a = 2

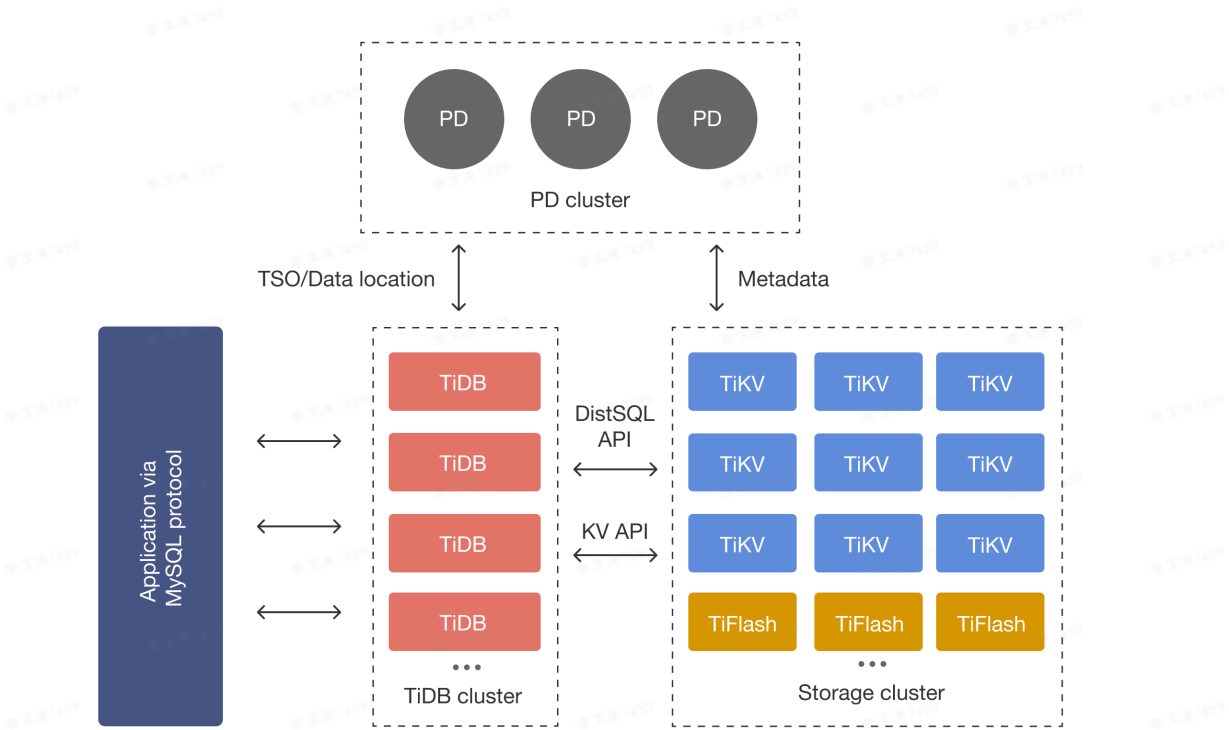
假如写阻塞读

t	T1	T2
1	begin start_ts = 1	
2	prewrite a = 2 (lock a)	
3	commit_ts = 3	
4		begin start_ts = 4
5		read a (wait)
6	abort	
...		wait until T1 timeout
10		read a = 1

Percolator 的设计下，当一个读事务遇到 lock 的时候需要等待 resolve lock 之后才能进行读取，否则可能会破坏快照隔离。因为有可能读的时候有其他已经获取了 commit\_ts 且小于这个读事务的 start\_ts，但 commit 操作还未完成。核心原因是：从 TSO 取 commit\_ts 和 commit 操作这两个步骤并不是原子的。

### 3. TiDB 分布式事务

#### 3.1 整体介绍



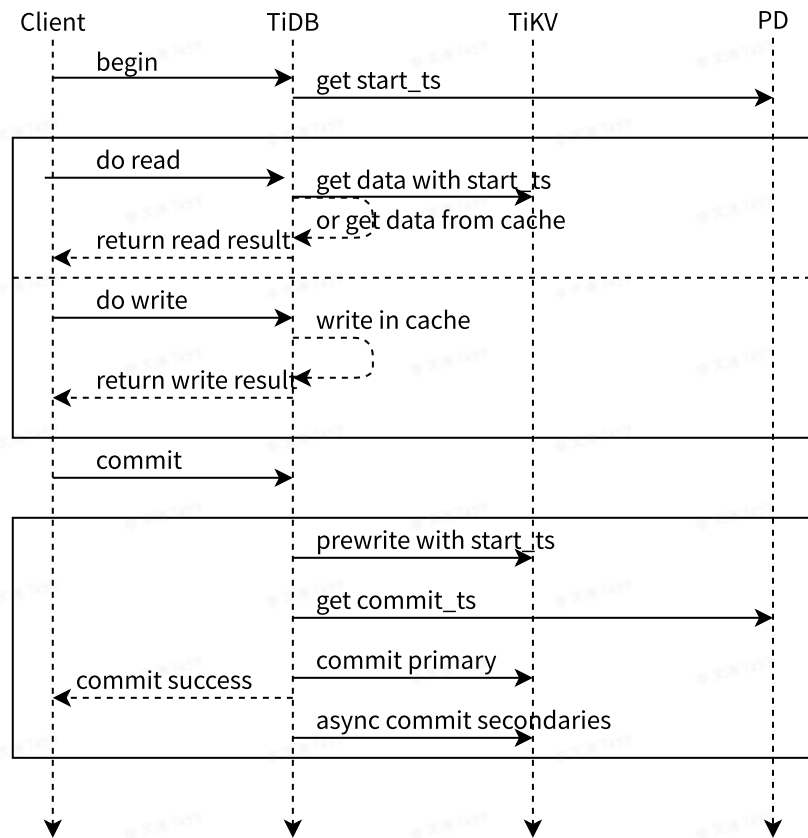
1. TiDB: SQL 层，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划，将实际的数据存取请求转发给底层的存储节点 TiKV。
2. PD(Placement Driver): 整个 TiDB 集群的元信息管理模块，负责存储每个 TiKV 节点实时的数据分布情况和集群的整体拓扑结构，为分布式事务分配事务 ID。
3. TiKV: KV 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。

#### 3.2 事务介绍

TiDB 在 Google Percolator 的基础上做了一些改动，实现了 SI 隔离级别的分布式事务，提供乐观事务与悲观事务两种事务模型。

##### 3.2.1 Optimistic Transaction

乐观事务的两阶段提交过程如下：



1. Client 开始一个事务。
2. TiDB 向 PD 获取 ts 作为当前事务的 start\_ts。
3. Client 发起读或写请求。
4. Client 发起 commit。
5. TiDB 开始**两阶段提交**:
  - a. prewrite:
    - i. 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，然后并发地向所有涉及的 TiKV 发起 prewrite 请求。
    - ii. TiKV 收到 prewrite 请求后:
      1. 检查 key 是否有新的已提交写入；
      2. 检查 key 是否已被其他事务加锁；
      3. 如果没有新的已提交写入且没有被其他事务加锁，就写入锁和数据，否则 prewrite 失败。
  - b. commit:
    - i. TiDB 收到所有 prewrite 响应且所有 prewrite 都成功后，向 PD 获取 ts 作为当前事务的 commit\_ts，向 primary key 所在 TiKV 发起 commit 请求。
    - ii. TiKV 收到 commit 操作后:

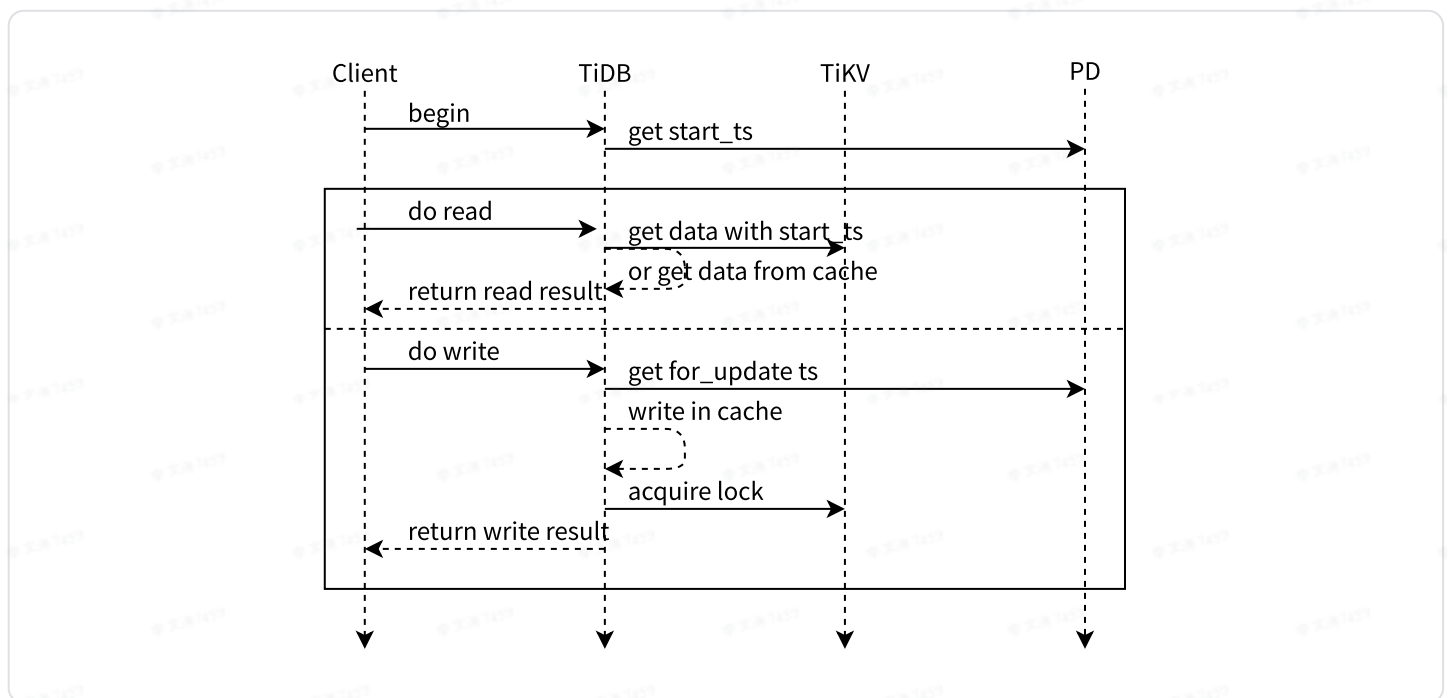
1. 检查 key 的锁是否还存在；
2. 检查 key 的锁的 owner 是否为当前事务；
3. 如果 key 的锁仍然存在且 owner 为当前事务，就提交数据并删除锁，否则 commit 失败。

6. TiDB 向 Client 返回事务提交成功。

7. TiDB 异步 commit secondaries。

### 3.2.2 Pessimistic Transaction

与乐观事务不同的是，悲观事务将上锁的时机提前到 DML 期间。



这里的悲观锁的格式和乐观事务中的锁几乎一致，但是这个锁只是一个占位符，等到 commit 的时候，再将这些悲观锁改写成标准 Percolator 模型的锁，后续流程和原来保持一致即可，唯一的改动是：对于读请求，遇到这类悲观锁的时候，不用像乐观事务那样等待解锁，直接返回可见数据即可。

## 3.3 Percolator 优化

### 3.3.1 Parallel Prewrite

TiDB 不会像 Percolator 要先 prewrite primary，再去 prewrite secondary。

当提交一个事务时，事务中涉及的 keys 会被分成多个 batches，每个 batch 在 prewrite 阶段会并行地执行。不需要关注 primary key 是否第一个 prewrite 成功。

### 3.3.2 Short Value in Write Column

当进行一次读操作时，需要先从 write 列找到 key 对应最新版本，然后从 data 列找到具体的数据。如果一个 value 比较小的话，那么查找两次开销相对来说比较大。



为了避免 short values 两次查找，TiDB 做了一个优化。如果 value 比较小，在 prewrite 阶段，不会将 value 放到 data 列中，而是将其放在 lock 列中。然后在 commit 阶段，这个 value 会从 lock 列移动到 write 列中。然后在访问这个 short value 时，就只需要访问 write 列就可以了，减少了一次查找。

### 3.3.3 Point Read Without Timestamp

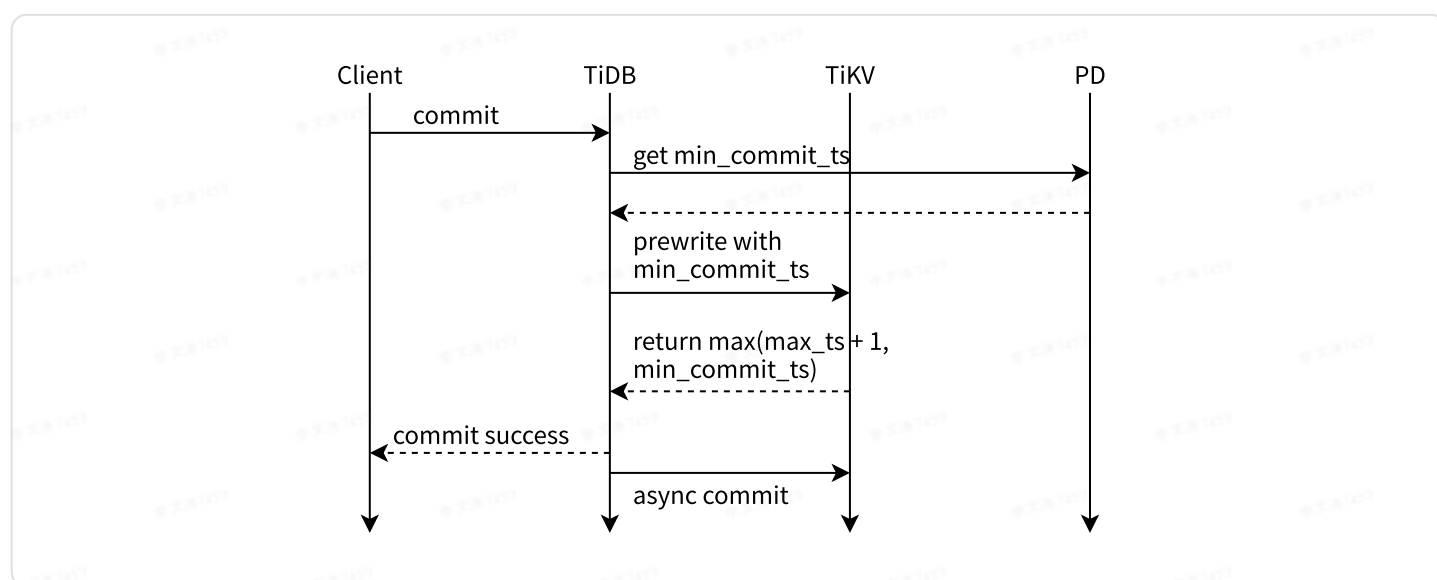
为了减少一次 RPC 调用和减轻 TSO 压力，对于单点读，并不需要获取 timestamp。

因为单点读不存在跨行一致性问题，所以直接可以读取最新的数据即可。

### 3.3.4 Calculated Commit Timestamp

TiDB 的 Async Commit 特性大大改善了事务提交的延迟，将 TiDB 与 TiKV 的 2 次交互减少到 1 次。

下图是 Async Commit 事务的提交流程：

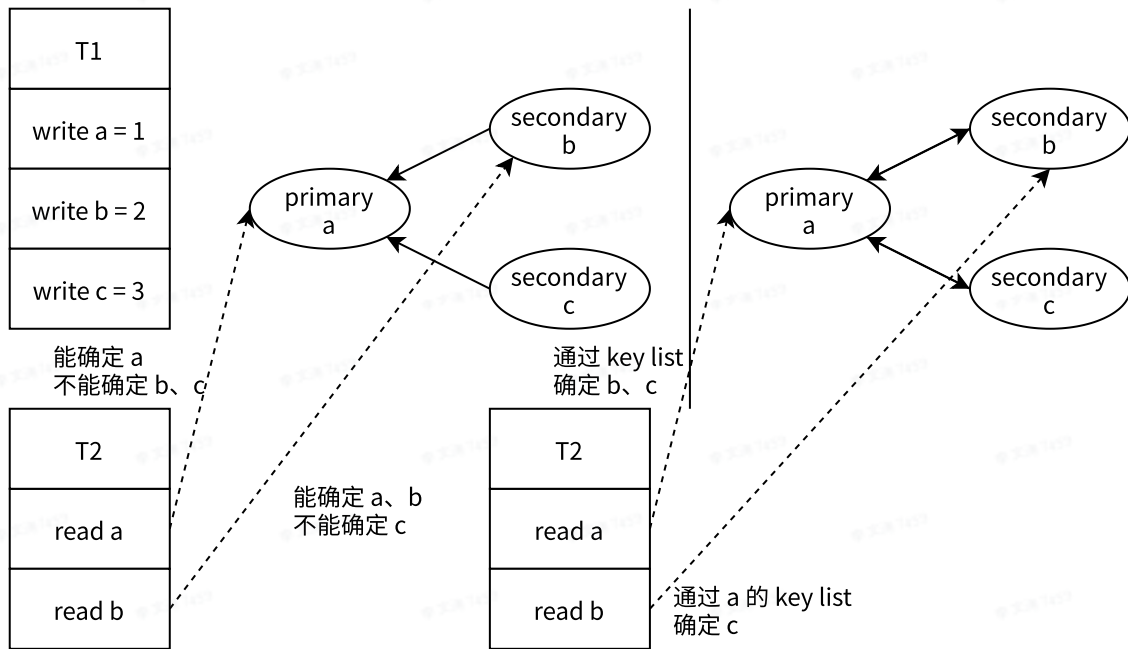


引入 Async Commit 之前，事务的 primary key 被提交才意味着这个事务被提交。Async Commit 就是把确定事务状态的时间提前到完成 prewrite 的时候，让整个提交的第二阶段都异步化进行。也就是说，对于 Async Commit 事务，**只要事务所有的 keys 都被成功 prewrite**，就意味着事务提交成功。并且为了实现 SI 的 RR 特性（repeatable read），commit\_ts 需要确保其他事务多次读取的值是一样的。那么 **commit\_ts** 就和其他事务的读取有相关性。

- 其他事务如何确定所有的 keys 都被成功 prewrite

TiDB 的做法是在 primary key 上存储所有 secondary keys 的列表，但显然，如果一个事务包含的 keys 的数量特别多，我们不可能把它们全部存到 primary key 上。所以 Async Commit 事务不能太大。只对包含不超过 256 个 keys 且所有 keys 的大小总和不超过 4096 字节的事务使用 Async Commit。过大的事务的提交时长本身较长，减少一次网络交互带来的延迟提升不明显，所以也不考虑让 Async Commit 支持更大的事务。





### 如何确定事务的 commit\_ts

假如只在 prewrite 前获取 commit\_ts，而不做推高处理，会影响 SI：

t	T1	T2
1	begin (start_ts = 1)	
2	get commit_ts = 2	
3		begin (start_ts = 3)
4		read a = 1
5	prewrite a = 2 (commit_ts = 2)	
6		read a = 2 ❌

不推高 commit\_ts

t	T1	T2
1	begin (start_ts = 1)	
2	get min_commit_ts = 2	
3		begin start_ts = 3
4		read a = 1 max_ts = 3
5	prewrite a = 2 (commit_ts = 4)	
6		read a = 1

推高 commit\_ts

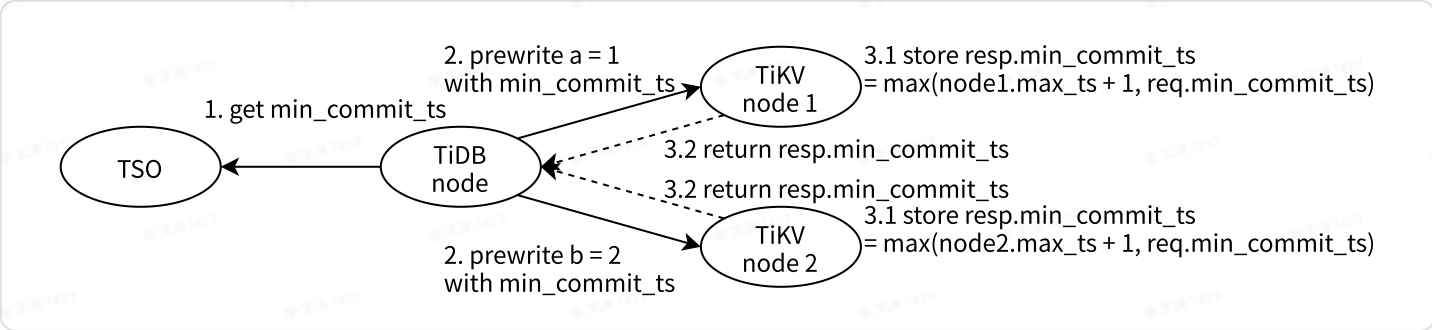
因此在 prewrite 前获取 commit\_ts 作为 min\_commit\_ts 后，还需要进行推高处理。下面公式可以计算出一个最终的 commit\_ts：

$$1 \max\{\min\_commit\_ts, \max\_read\_ts\_of\_written\_keys + 1\} \leq commit\_ts \leq now$$

TiDB 没有选择记录每个 key 的最大的读取时间，而是记录每个 region 的最大读取时间，所以公式转换为：

1

commit\_ts = max{min\_commit\_ts, max{region\_1\_max\_read\_ts, region\_2\_max\_read\_ts, .



思考：为什么在 prewrite 前要取一次新的 min\_commit\_ts？不能直接用 start\_ts + 1 作为 min\_commit\_ts 吗？

如果以 start\_ts + 1 作为 min\_commit\_ts，**线性一致性将降低为因果一致性**。如下图：

在物理时间上，T2 比 T1 先提交，但 T3 却能读到 T1 的修改。

初始状态 a = 1, b = 2  
物理时间上，T2 先把 b 改为 3，T1 才把 a 改为2

t	T1	T2	T3
1	begin (start_ts = 1)		
2			
3			begin (start_ts = 3)
4		begin (start_ts = 4)	
5		write b = 3 (commit_ts = 5)	
5	write a = 2 (commit_ts = 2)		
6			read a = 2
7			read b = 2

3.3.5 Single Region 1PC

对于事务只涉及到一个 Region，那其实是没有必要走 2PC 流程的，直接以 `commit_ts = region_max_read_ts` 提交事务即可。

## 4. CockroachDB 分布式事务

### 4.1 整体介绍

#### 4.1.1 基本概念

首先明确几个概念：

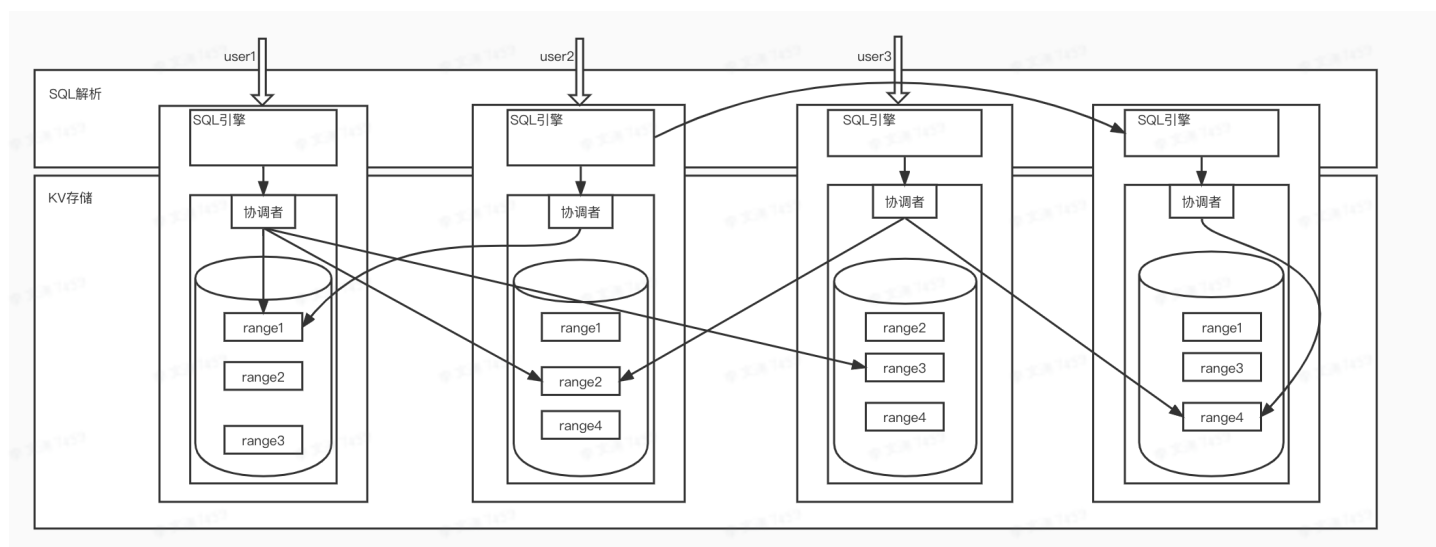
**Range:** CockroachDB 使用范围分区的方式，将全局递增有序的 K-V 数据，划分为 64MB 单位大小的 chunk，称为 K-V Range。Range 是路由、存储、复制的基本单位，具备自动分裂与合并的能力。

**Leaseholder:** CockroachDB 中最重要的概念之一，涉及到 lease 机制。简单来说，持有 lease 的副本可以对外提供 KV 的一致性读写。如果一个 raft group 中的某个副本持有 lease，那么该副本所在的节点称为 leaseholder 节点。持有 lease 的副本，一般也是 raft group 中的一个 leader。

**Gateway:** gateway 节点是和 leaseholder 节点相对的概念。gateway 节点负责解析 SQL 请求、充当事务的 coordinator，并将 KV 操作路由到正确的 leaseholder 节点上。

**Raft Leader / Log / Replica:** 至少 3 的 range 可以组成一个 raft group，对外保证高可用和一致性的读写。raft 中的 leader、log、replica 用于选举、日志复制等，属于 Raft 算法的范畴，这里不再赘述。

例如，一个 4 节点的 CockroachDB 集群，有如下的结构：



#### 4.1.2 基本架构

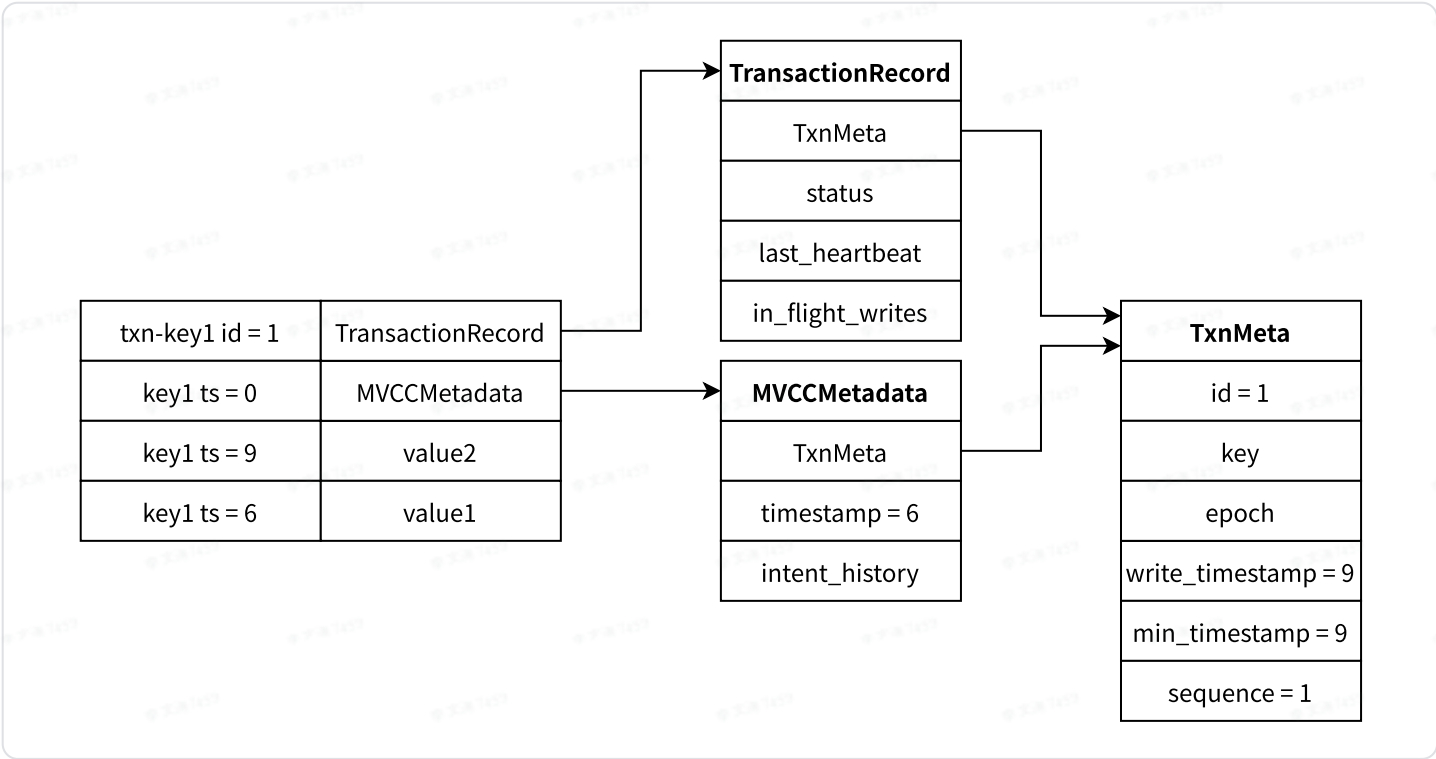
CockroachDB 在逻辑上分为 5 层，从上往下依次是：

1. SQL 层：解析 SQL，转化成 KV 操作
2. Transactional 层：允许对多个 KV 进行原子性变更
3. Distribution 层：负责路由，屏蔽跨节点的 KV 操作的细节，使得像在一个节点上操作所有 KV 一样
4. Replication 层：跨多节点一致性同步复制 KV Range 数据，并且提供一致性读的能力
5. Storage 层：在磁盘上写入和读取 KV 数据，目前支持 Pebble 和 RocksDB 两个 KV 存储引擎

## 4.2 事务介绍

CRDB 依赖 **HLC** 实现了 **Serializability** 隔离级别的分布式事务。此外，它还引入了较多优化减少了事务提交延迟。

先看 CRDB 的数据组织形式：



### MVCCMetadata

- TxnMeta：不为空时代表有 write intent。
- timestamp：key 的最新已提交版本。

### Transaction record

- TxnMeta
- status：表示事务状态，有以下取值：
  - PENDING：表示事务仍在进行中。
  - COMMITTED：表示事务已提交。
  - STAGING：启用 parallel commit 时使用，表示事务可能处于也可能不处于已提交状态。
  - ABORTED：表示事务已中止。
- last\_heartbeat：coordinator 对事务的心跳。
- in\_flight\_writes：启用 parallel commit 时使用，保存事务正在写入的 keys 列表。

### TxnMeta

- id：表示事务唯一 ID 标识。

- key: 表示事务第一次读取或写入的 range 的 start key。
- write\_timestamp: 表示事务的提交时间戳，可能会被后推。
- min\_timestamp: 表示事务的开始时间戳。

## 4.2.1 事务流程

### 4.2.1.1 Phase1 (writes and reads)

CRDB 在开始事务后每次执行写 DML 时就将数据写入存储层作为未提交数据，同时会在 MVCCMetadata 中创建 TxnMeta。在第一次写入数据时还会创建 Transaction Record。后续事务可以通过 `key -> MVCCMetadata -> TxnMeta` 找到事务的未提交数据。

每个事务会选取读取或写入的第一个 range 的 start key 作为 **Transaction Record** 的 **anchor key**。  
Transaction Record 的 key 的编码为 `txn-{anchor_key}{txn_id}`，具体实现在 **TransactionKey** 中（这样的好处是一个 range 中的所有 Transaction Record 都有序聚集在一起，方便找所有 records）。

### 4.2.1.2 Phase2 (commit or rollback)

CRDB 在接收到 commit 请求后，将 Transaction Record 的状态设置为 Committed，返回客户端事务提交成功，然后异步进行 **Cleanup** 操作。

### 4.2.1.3 Cleanup

更新 MVCCMetadata，将 TxnMeta 清除，并更新 timestamp。

## 4.2.2 事务并发控制

在并发控制方面，CRDB 在时间戳排序协议上做了修改，在保证 **Serializability** 隔离级别的情况下进一步追求性能。

首先引用一下《数据库系统概念》中对于经典多版本时间戳排序的描述：

- 1: 数据库系统会在每个事务  $T_i$  开启时为其分配一个时间戳  $TS(T_i)$ 。
- 2: 对于每个数据项  $Q$ ，都有一个版本序列  $\langle Q_1, Q_2, \dots, Q_m \rangle$  与之关联，对于数据项  $Q$  的一个数据版本  $Q_k$ ，它包含三部分内容，即：
  - a: 正常的数据内容 content
  - b: 此数据版本  $Q_k$  创建的时间  $W\text{-Timestamp}(Q)$ ,
  - c: 所有成功读取  $Q_k$  的事务的最大时间戳  $R\text{-Timestamp}(Q)$ 。

事务  $T_i$  执行写操作  $Write(Q)$ ，创建  $Q$  的一个新数据版本  $Q_k$ ，版本的 content 字段保存写入的值， $W\text{-Timestamp}$  和  $R\text{-Timestamp}$  初始化为  $TS(T_i)$ ，每当事务  $T_j$  读取  $Q_k$  并且  $TS(T_j) > R\text{-Timestamp}(Q_k)$  时，用  $TS(T_j)$  更新  $Q_k$  的  $R\text{-Timestamp}$ 。

- 3: 并发控制机制是，每当事务  $T_i$  发出  $Read(Q)$  或  $Write(Q)$ ，需要定位满足条件的数据项  $Q$  的版本  $Q_k$ ，即  $Q_k$  是数据项  $Q$  的所有版本中写时间戳  $W\text{-Timestamp}$  小于等于  $TS(T_i)$  的最大的那个

a: 如果  $T_i$  发出  $Read(Q)$ , 那么返回  $Q_k$ , 并用  $\max(R\text{-Timestamp}(Q_k), TS(T_i))$  更新  $Q_k$  的  $R\text{-Timestamp}$ 。

b: 如果  $T_i$  发出  $Write(Q)$ , 若  $TS(T_i) < R\text{-Timestamp}(Q_k)$ , 即有其它更新事务已经读过这个数据版本, 则  $T_i$  回滚; 否则, 若  $TS(T_i) == W\text{-Timestamp}(Q_k)$ , 则用要写入的 value 更新  $Q_k$  的 content; 否则, 也就是  $TS(T_i) > W\text{-Timestamp}(Q_k)$ , 那么以  $TS(T_i)$  为读写时间戳创建新的数据项  $Q$  的版本。

也就是说, 它的数据是如下的一种组织形式:

key1, rts=9, wts=9	value1
key1, rts=6, wts=6	value2
key2, rts=3, wts=3	value3
key2, rts=1, wts=1	vlaue5

与经典的时间戳排序协议相比, CRDB 的多版本时间戳排序做了一些改进:

#### 4.2.2.1 RW 冲突处理

经典的多版本时间戳排序, 通过简单粗暴的回滚方式处理 RW 冲突。然而事务回滚是一个代价比较大的操作, 尤其是对于下面的场景:

```
1 create table t (a int primary key, b int);
2 insert into t values (1,1),(2,2);
3 #T1                                     #T2
4 begin;
5 insert into t
6 values (3,3),(4,4),(5,5);
7                                     begin;
8                                     select * from t where a = 1;
9                                     commit;
10 update t set b = b + 1
11 where a = 1;
```

为了避免这种问题, 在 CRDB 中, 一个事务维护两个时间戳, 即 `read_timestamp` 和 `write_timestamp` (在事务开启时它们相等), 同时引入 **forward** 和 **refresh** 机制, 来挽救 T1 的回滚问题。

- **forward**

具体来说, 事务在运行过程中, 会记录下所有的读过的 key, 然后在遇到例如 T1 最后执行 update 的场景时, T1 会把自己的 `write_timestamp` 调整到 `a = 1` 这个记录项的最近一次 `read_timestamp` 之后, 然后继续用这个新的 `write_timestamp` 写入数据。具体实现在[这里](#)。



但是这样做会带来两个明显的问题：

1. 在事务执行过程中，可能会遇到多次 forward，那么它产生的未提交数据可能会是多个版本。
2. 如果事务的 read\_timestamp 和 write\_timestamp 不相等，而此时又有新的事务在 (read\_timestamp, write\_timestamp) 之间执行了写操作并提交，那就可能会破坏 Serializability。

对于第一个问题，事务需要在提交或回滚后，开启一个异步任务用于处理遗留下来的数据并调整 MVCCMetadata 的 timestamp，让事务之前写入的旧的 write\_timestamp 都收敛到事务最终的 write\_timestamp。

对于第二个问题，需要通过 refresh 机制尝试提交。

- **refresh**

如果在事务提交时发现事务的 read\_timestamp 和 write\_timestamp 不相等，那么就会触发 refresh 操作：以 write\_timestamp 作为新的 read\_timestamp，把之前读过的 key 重新读一遍，同时比较新读取到的数据和读过的数据是否一致。如果不一致，则说明有新的数据写入了，所以 refresh 失败，T1 只能回滚/重试；如果一致，说明在这个区间没有新的数据插入，refresh 成功，事务 T1 可以提交。具体实现在[这里](#)。

#### 4.2.2.2 WR 冲突处理

当事务读取一个 key 时，它先检查在 key 的 MVCCMetadata 是否存在 TxnMeta。

- 如果不存在，那就寻找小于自己 read\_timestamp 的最大数据版本
- 如果存在，那就确定这个 TxnMeta 是不是自己留下的
  - 如果是，就读取 TxnMeta 中的时间戳指示的那个版本
  - 如果不是，就需要判断 TxnMeta 的时间戳和自己的 read\_timestamp
    - 如果 TxnMeta 保存的时间戳大，那么读取之前的已提交版本即可
    - 如果 TxnMeta 保存的时间戳小，说明要读取的数据被修改过，需要去查看 Transaction Record 状态：
      - Pending：读请求进入队列等待
      - Committed：用 Transaction Record 中保存的 write\_timestamp 更新 MVCCMetadata 的 timestamp，同时删除掉 TxnMeta（roll forward），然后就可以按照 read\_timestamp 读取了
      - Aborted：删除 TxnMeta 和它指向的数据（roll back），再按照 read\_timestamp 读取对应版本

#### 4.2.2.3 WW 冲突处理

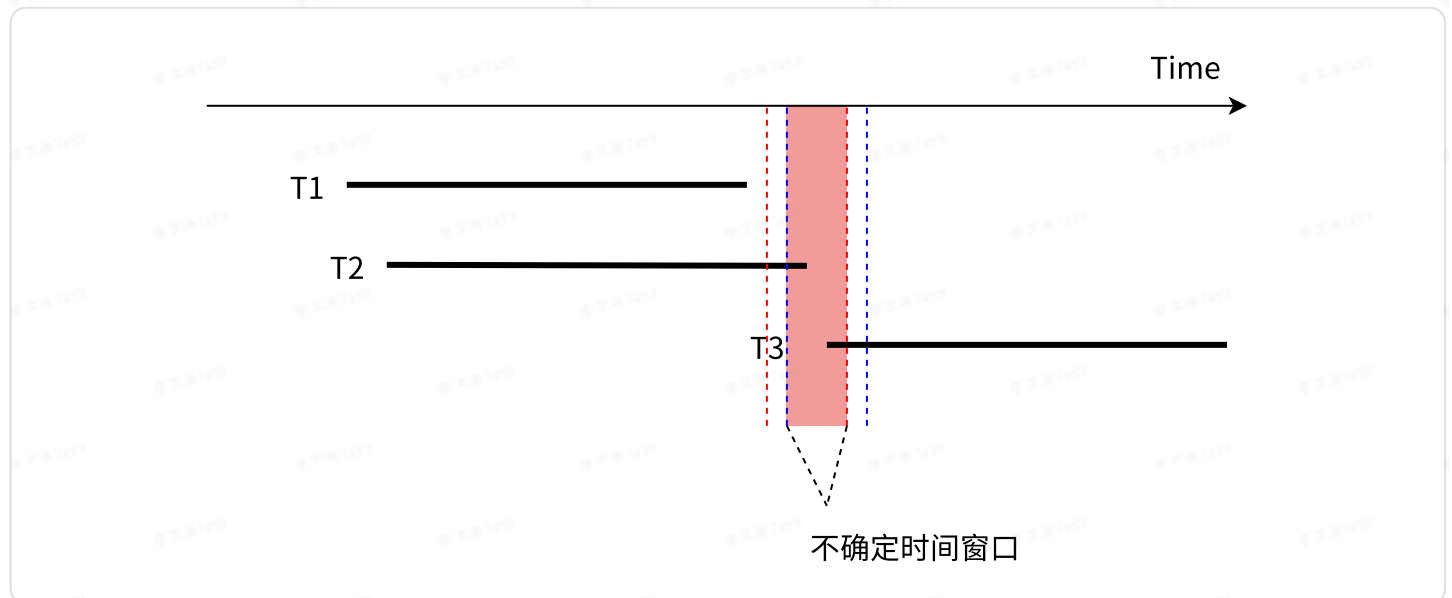
当事务写一个 key 时，和读操作类似，它首先还是需要检查 key 的 MVCCMetadata 是否存在 TxnMeta。



- 如果存在，无论时间戳大小，都需要去检查 TxnMeta 指向的 Transaction Record 的状态：
  - Pending：写请求进入队列等待
  - Committed：用 Transaction Record 中保存的 write\_timestamp 更新 MVCCMetadata 的 timestamp（roll forward），再写入新的 TxnMeta
  - Aborted：删除 TxnMeta 和它指向的数据（roll back），再写入新的 TxnMeta

#### 4.2.2.4 Uncertainty Interval

此外还有因为引入 HLC 后需要解决的 Read In Uncertainty Interval 问题。



HLC 需要考虑时间不同步误差，如下图，事务 T2 的提交时间点的误差范围与 T3 的开始时间点的误差范围重叠，就无法区分 T3 是否应该看到 T2 提交的数据。

CRDB 在事务开始时还会设置 [Transaction.GlobalUncertaintyLimit](#)（默认500ms）。在读的时候还要检查是否存在 `[read_timestamp, read_timestamp + GlobalUncertaintyLimit)` 的其他版本。如果存在，则会尝试后推 [read\\_timestamp](#)。

### 4.2.3 一些优化

#### 4.2.3.1 Transaction Pipelining

思路就是异步复制，事务写操作在复制时采用 pipeline 方式进行，从而减少执行多次写入的事务的延迟。如以下事务：

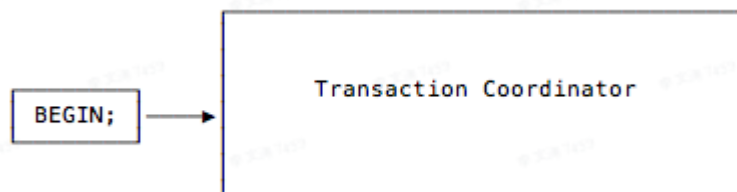
```
1 -- CREATE TABLE kv (id UUID PRIMARY KEY DEFAULT gen_random_uuid(), key VARCHAR,
2 BEGIN;
3 INSERT into kv (key, value) VALUES ('apple', 'red');
4 INSERT into kv (key, value) VALUES ('banana', 'yellow');
5 INSERT into kv (key, value) VALUES ('orange', 'orange');
6 COMMIT;
```

1. 对每个写操作，Gateway 向对应 Leaseholder 发送写请求。
2. Leaseholder 在本地写入数据后，返回写成功，并异步复制。未完成复制的 keys 都保存在 Gateway 的 `in_flight_writes` 中。
3. 在事务提交前，将 `in_flight_writes` 写入 Transaction Record，再等待剩余的写操作复制完成。

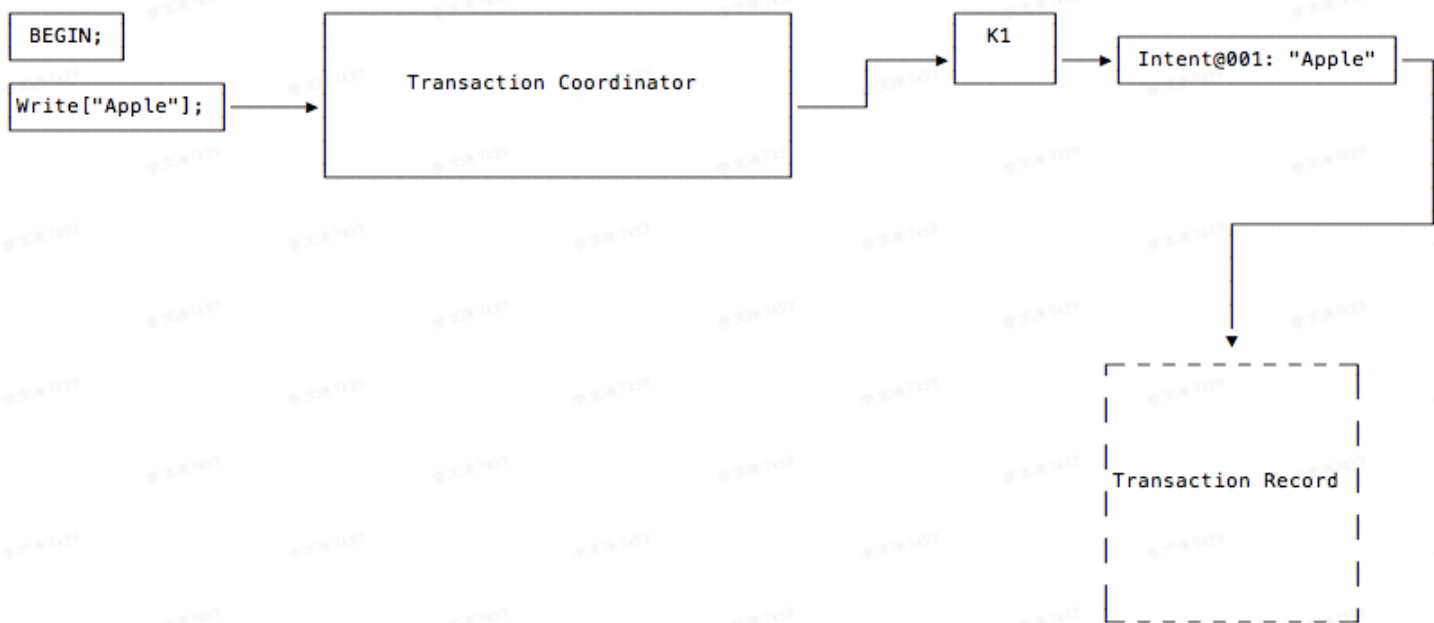
### 4.2.3.2 Parallel Commit

Parallel Commit 可以将事务的提交延迟从两轮 consensus 减少到一轮。结合 Transaction pipelining，一般情况下能使 OLTP 事务的延迟接近理论最小值：所有读延迟的总和 + 一轮 consensus 延迟。

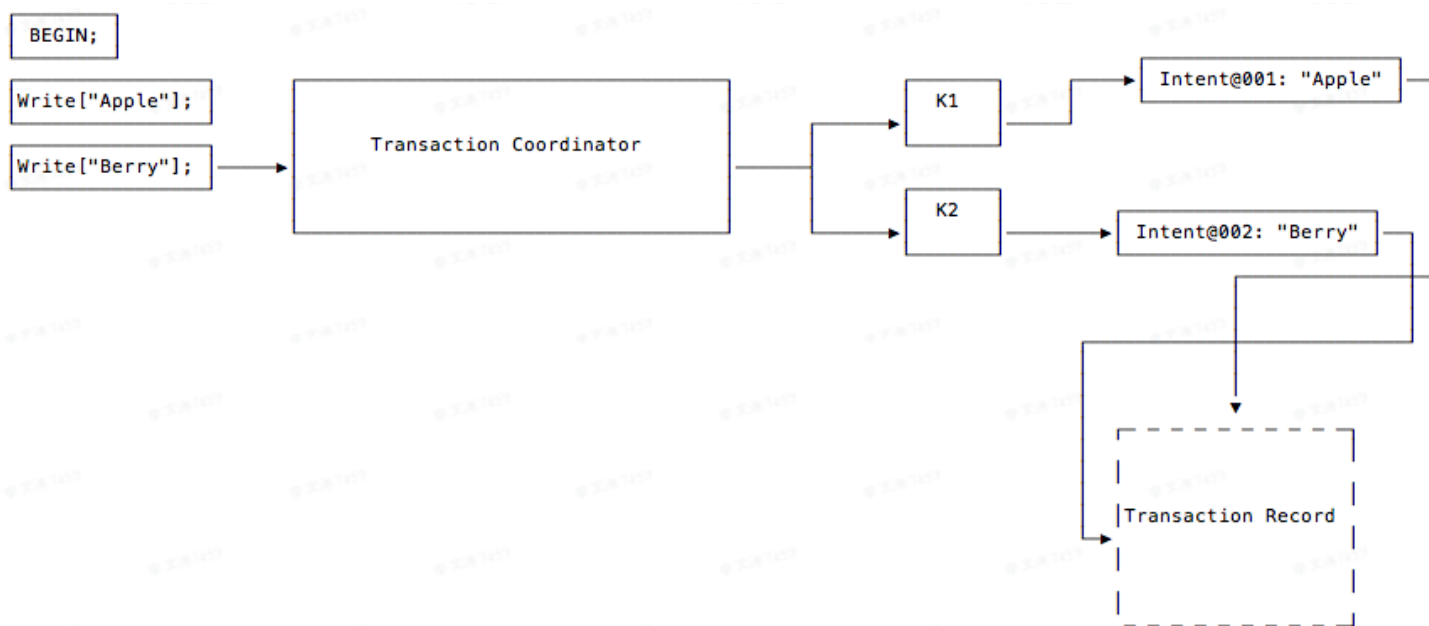
1. Client 开启事务。



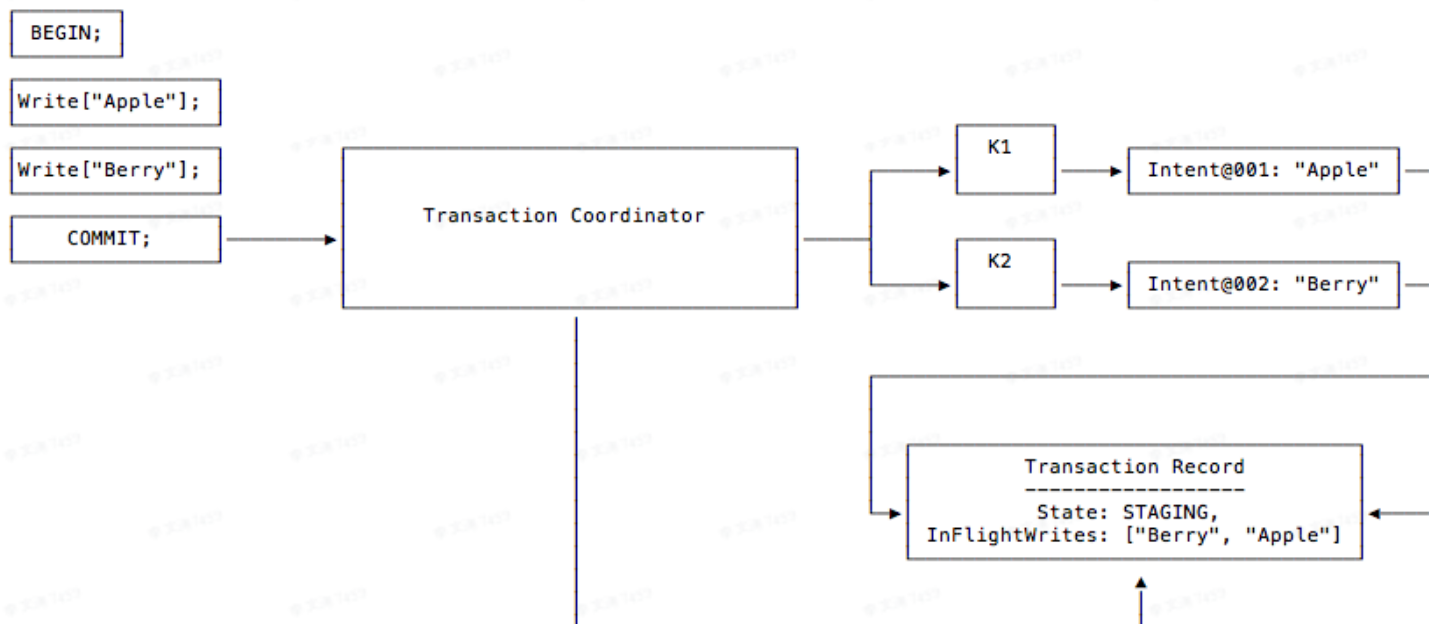
2. Client 发起第一个写请求 `key = Apple`，Transaction Coordinator 写入对应的 write intent 后就直接返回，然后异步复制 write intent。



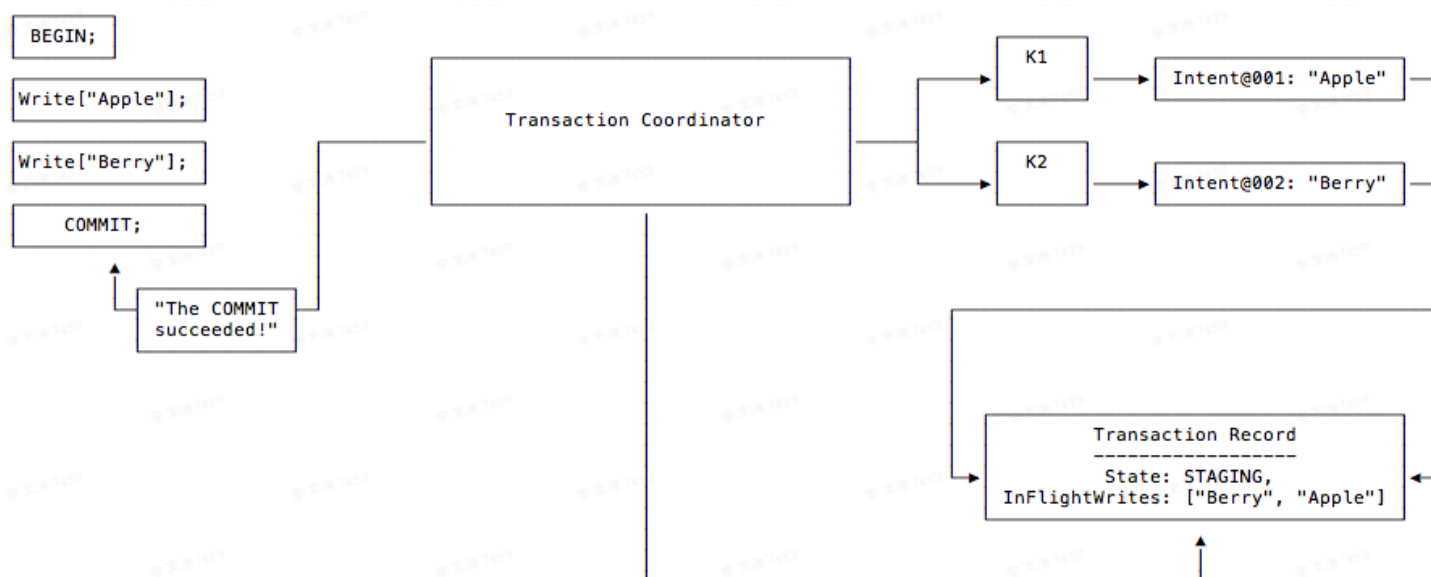
3. Client 发起第二个写请求 `key = Berry`，和第二步一样，Transaction Coordinator 写入对应的 write intent 后就直接返回，然后异步复制 write intent。



4. Client 发起 `COMMIT` 请求，Transaction Coordinator 创建 Transaction Record，且标记状态为 STAGING，并将每一个 write key 都写入 `in_flight_writes`。



5. Transaction Coordinator 等待 `in_flight_writes` 中的写操作复制完成后，再返回 Client 事务提交成功。



1. Transaction Record 的状态为 Staging，其 in\_flight\_writes 全部复制完成，表明事务已经隐式提交了。
2. Transaction Record 的状态为 Committed，表明事务已经显示提交了。

尽管 1 和 2 在逻辑上是等价的，但 Transaction Record 会尽快将 Transaction Record 从 Staging 状态切换到 Committed 状态，否则其他事务在遇到 Staging 状态的事务时，还需要查询 in\_flight\_writes 是否复制完成，然后再恢复事务状态，而这个查询操作也是很耗时的。

另外，当其他事务遇到处于 Staging 状态的事务时，它们会通过 Transaction Record 的 last\_heartbeat 来检查事务是否仍在进行中。如果还在进行中，则事务先进行等待。因为让 Coordinator 更新事务状态将比查询 in\_flight\_writes 是否复制完成要快。因此在一般情况下，事务状态的恢复操作在 Coordinator 宕机时才使用。

## 5. 附录

### 5.1 Percolator 伪代码

#### Transaction

```

1 class Transaction {
2     struct Write { Row row; Column col; string value; } // 写格式
3     vector<Write> writes_; // 一个事务可以包含多个 Write
4     int start_ts_; // 作为 Bigtable key 中的 timestamp
5     Transaction() : start_ts_(oracle.GetTimestamp()) {} // 开启事务时从 TSO 获取 sta
6     void Set(Write w);
7     void Get(Row row, Column col, string* value);
8     bool Prewrite(Write w, Write primary);
9     bool Commit();
10 };
  
```

## Set

```
1 void Transaction::Set(Write w) {
2     // 写入先做缓存, 实际写入在 Commit 时触发
3     writes_.push_back(w);
4 }
```

## Get

```
1 bool Transaction::Get(Row row, Column col, string* value) {
2     while (true) {
3         bigtable::Txn T = bigtable::StartRowTransaction(row);
4         // 1. 检查是否有 timestamp <= start_ts_ 的锁
5         if (T.Read(row, col+"lock", [0, start_ts_])) {
6             // Cleanup primary lock:
7             // 检查是否超时, 如果超时则删除 primary lock, 否则等待
8             // Cleanup secondary lock:
9             // 检查 primary 的状态
10            // 如果 primary 无锁, 表示已经过了 commit point, 删除 secondary lock
11            // 如果 primary 有锁且未超时, 等待
12            // 如果 primary 有锁且已超时, 删除 secondary lock
13            BackoffAndMaybeCleanupLock(row, col);
14            continue;
15        }
16
17        // 2. 然后读取并返回 commit timestamp <= start_ts_ 的最新的数据
18        latest_write = T.Read(row, col+"write", [0, start_ts_]);
19        if (!latest_write.found()) return false; // no data
20        int data_ts = latest_write.start_timestamp();
21        *value = T.Read(row, col+"data", [data_ts, data_ts]);
22        return true;
23    }
24 }
```

## Prewrite

```
1 // Prewrite tries to lock cell w, returning false in case of conflict.
2 bool Transaction::Prewrite(Write w, Write primary) {
3     bigtable::Txn T = bigtable::StartRowTransaction(w.row);
4
5     // 1. 检查 timestamp >= start_ts_ 的数据
6     // 如果已经有了说明已经有其他事务提交了数据, 本事务不能再写
```

```

7   if (T.Read(w.row, w.col+"write", [start_ts_, ∞])) return false;
8   // 2. 再看一下是否已经被上锁了, 如果有的话说明有写冲突
9   if (T.Read(w.row, w.col+"lock", [0, ∞])) return false;
10
11  // 3. 写入 data 和 lock 字段
12  T.Write(w.row, w.col+"data", start_ts_, w.value);
13  T.Write(w.row, w.col+"lock", start_ts_,
14          {primary.row, primary.col}); // 用于找 primary
15  return T.Commit();
16 }

```

## Commit

```

1  bool Transaction::Commit() {
2  // 1. 先选一行作为 primary, 其他的作为 secondaries
3  Write primary = writes_[0];
4  vector<Write> secondaries(writes_.begin()+1, writes_.end());
5
6  // 2. 先 Prewrite primary, 再 Prewrite secondaries
7  if (!Prewrite(primary, primary)) return false;
8  for (Write w : secondaries)
9      if (!Prewrite(w, primary) return false;
10
11  // 3. 从 TSO 获取 commit_ts
12  int commit_ts = oracle.GetTimestamp();
13
14  // 4. 先 Commit primary
15  Write p = primary;
16  bigtable::Txn T = bigtable::StartRowTransaction(p.row);
17  // 检查 primary 是否有锁, 如果没有说明被已经超时被清理, 直接退出
18  if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_])) return false;
19  // 对 primary 写入 write 列, 清除 lock 列
20  T.Write(p.row, p.col+"write", commit_ts, start_ts_);
21  T.Erase(p.row, p.col+"lock", start_ts_);
22  // commit point
23  if (!T.Commit()) return false;
24
25  // 5. 再 Commit secondaries
26  for (Write w : secondaries) {
27      bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
28      bigtable::Erase(w.row, w.col+"lock", start_ts_);
29  }
30  return true;
31 }

```

## 6. 总结

可以看出经典 Percolator 协议是针对查询优先的场景来设计的，而 TiDB 和 CRDB 就在写入优先的场景做了一些改动，根据不同的场景去做不同的设计也是十分必要的。

## 7. 参考

[https://www.usenix.org/legacy/event/osdi10/tech/full\\_papers/Peng.pdf](https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Peng.pdf)

<https://github.com/pingcap/tidb>

<https://github.com/tikv/tikv>

<https://cn.pingcap.com/blog/tikv-source-code-reading-12>

<https://cn.pingcap.com/blog/tikv-source-code-reading-13>

<https://cn.pingcap.com/blog/async-commit-principle>

<https://docs.pingcap.com/tidb/dev/optimistic-transaction>

<https://docs.pingcap.com/tidb/dev/pessimistic-transaction>

<https://tikv.org/deep-dive/distributed-transaction/optimized-percolator>

<https://github.com/cockroachdb/cockroach>

<https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer.html>