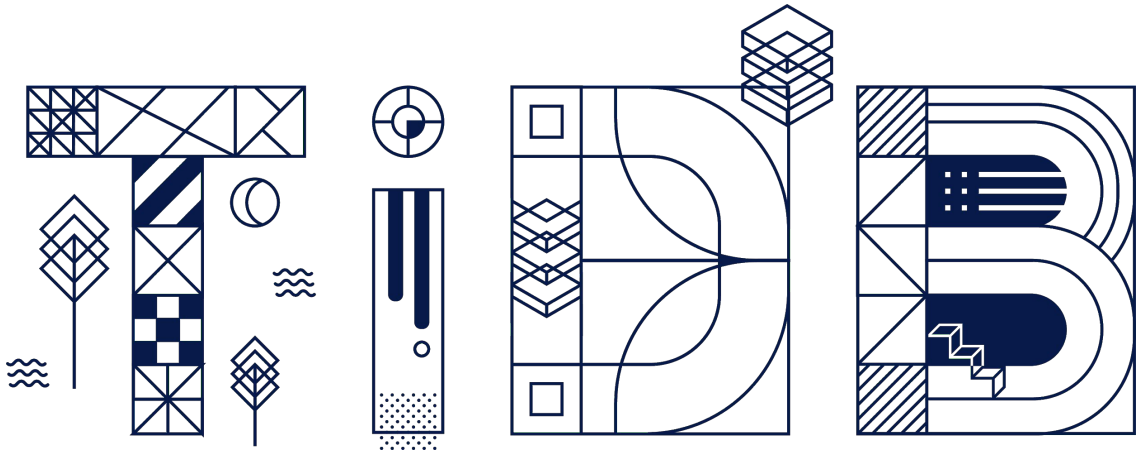
PingCAP TiDB

TiDB 悲观锁

Presented by: zhaolei



Agenda

- 悲观锁要解决的问题
- MySQL 悲观锁行为
- TiDB 悲观事务
    - 等锁
    - 死锁
- 测试
- 使用

PingCAP

Part I - 悲观锁解决的问题

乐观事务

- 提交时检测冲突。
- 非常适合分布式数据 库，性能高。

乐观事务存在的问题

- 冲突需要重试。
- 同一语句重试可能会得到不同的结果，如 affected rows。
- 冲突严重且重试代价大的场景性能差，失败率高。
- 用户习惯于悲观事务模型，应用侧一般不会重试。

悲观锁解决的问题

- 通过支持悲观事务，降低用户修改代码的难度甚至不用修改代码。
- 乐观事务模型在冲突严重的场景和重试代价大的场景无法满足用户需求，支持悲观事务可以弥补这方面的缺陷，拓展 TiDB 的应用场景。

PingCAP

Part II - MySQL 悲观锁行为

PostgreSQL

Session A	*Session B*
> begin; > set transaction isolation level repeatable read;	> begin; > set transaction isolation level repeatable read;
> update test set value = value + 1 where id = 1; UPDATE 1	
	> update test set value = value + 1 where id = 1; block...
> commit; COMMIT	ERROR: could not serialize access due to concurrent update

MySQL

Session A	*Session B*
> begin;	> begin;
> update test set value = value + 1 where id = 1; Query OK, 1 row affected	
	> update test set value = value + 1 where id = 1; block...
> commit; Query OK, 0 rows affected	Query OK, 1 row affected
	> commit; Query OK, 0 rows affected

MySQL

Session A	*Session B*
> create table t (i int);	
> begin; > select * from t; Empty set	
	> insert into t values (1); Query OK, 1 row affected
> select * from t for update; 1 row in set	
> select * from t; Empty set	

MySQL 悲观锁行为

- LOCKING READS、UPDATE 和 DELETE 基于锁。
- 写写冲突不会 abort，基于锁的行为会读到最新已提交的数据。
- 写写冲突不一定会导致异常，写操作基于最新已提交的数据能避免大部分异常。

PingCAP

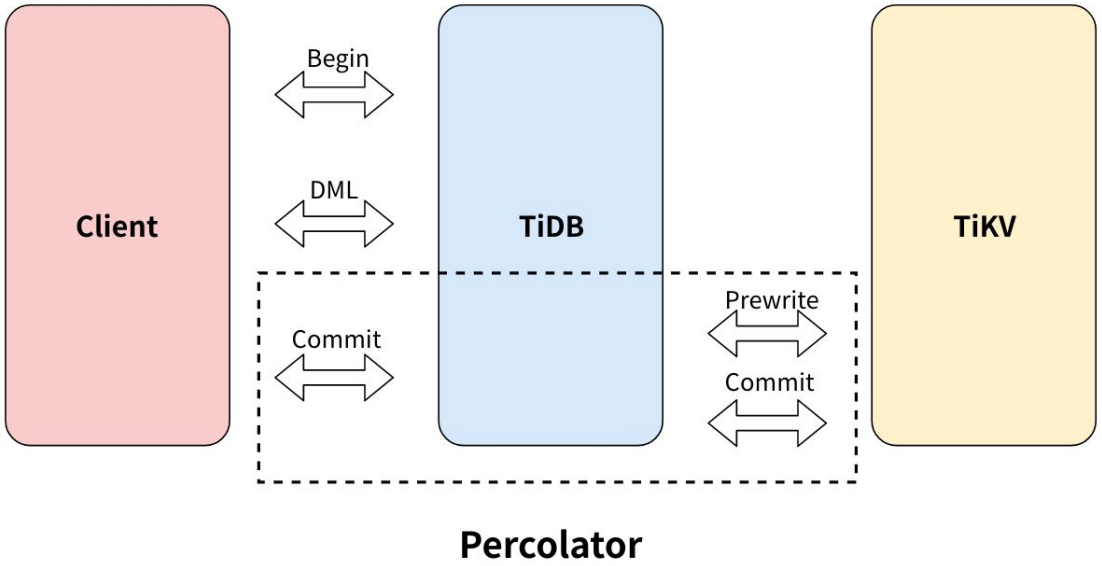
Part III - TiDB 悲观事务

实现上的考虑

- 在现有实现基础上进行改造，减少改动。
- 兼容现有乐观事务模型，支持乐观、悲观混合使用。
- 兼容 MySQL 行为。

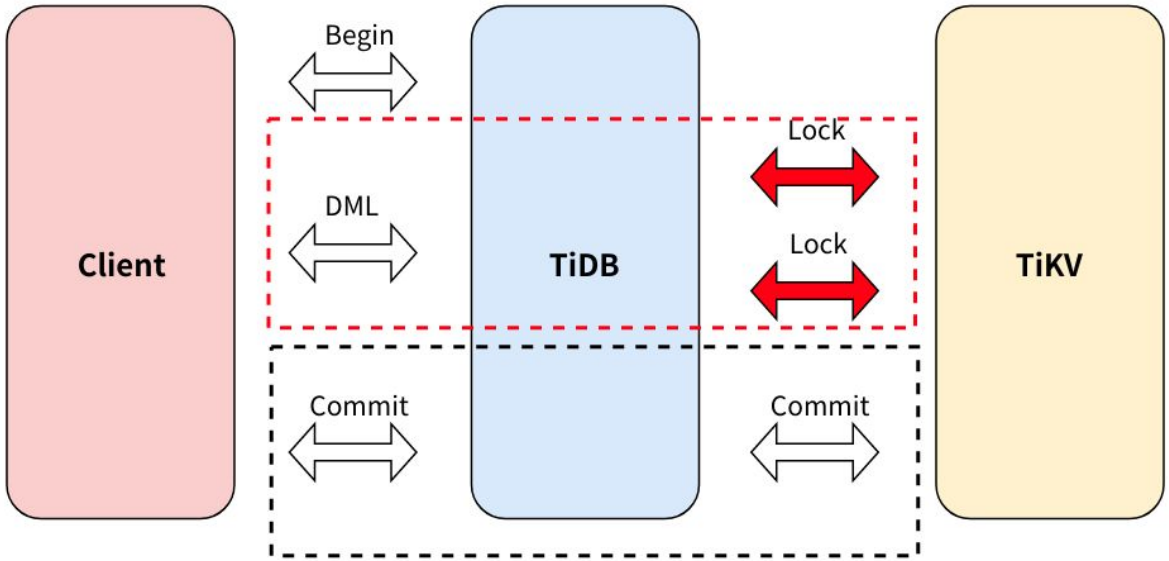
Percolator

- 支持 Snapshot Isolation 的分布式事务。
- 乐观事务模型，在提交时检测冲突，发生冲突需要重试整个事务。



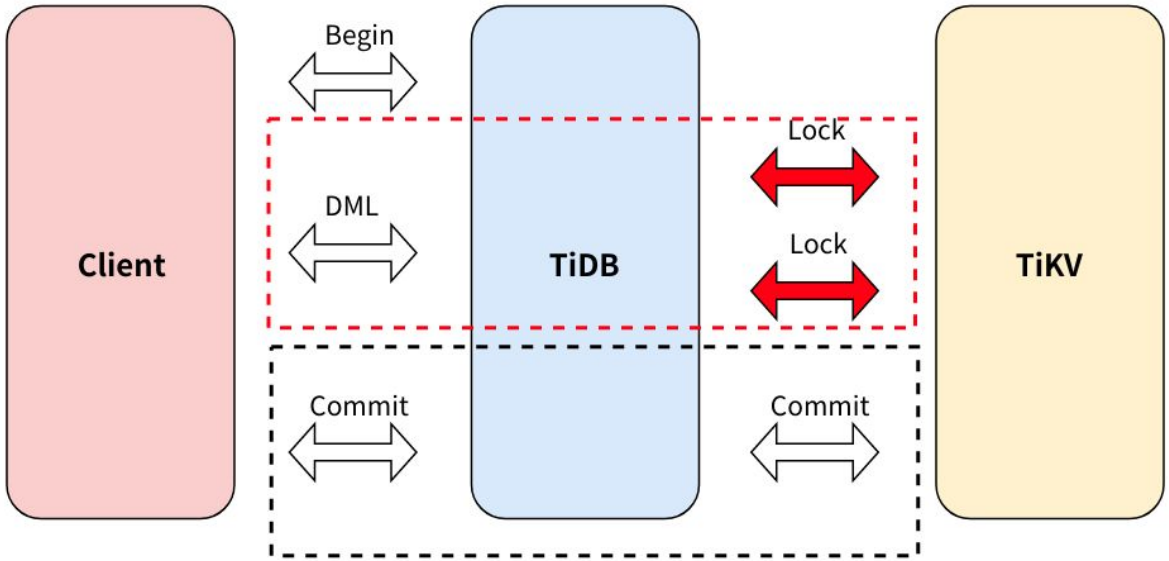
初步想法

- 把 Prewrite 阶段一次性加锁操作提前到执行 DML 阶段。
- 悲观锁遇到写冲突不会 abort，所以可以逐步加上所有 锁。



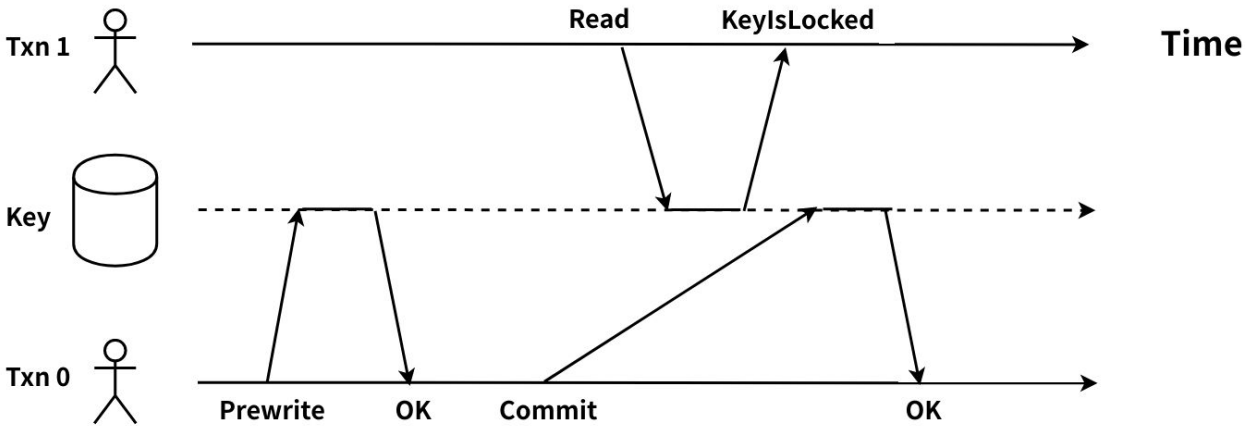
问题

- 锁的存在会阻塞读，悲观锁存在时间长。



为什么 Percolator 中锁会阻塞读？

- 事务要读到最新的 commit ts 小于事务 start ts 的数据。
- 可能该 lock 所属事务已提交，且 commit ts 小于读事务的 start ts。

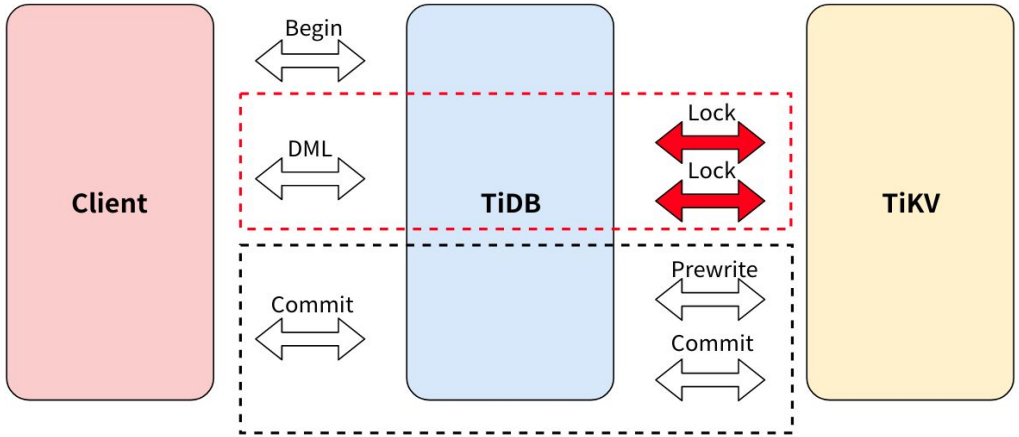


为什么 Percolator 中锁会阻塞读？

- 问题的关键是，遇到 Percolator 的 lock 无法知道这个事务是在 Commit 阶段还是 Prewrite 阶段。

基于 Percolator 的悲观事务

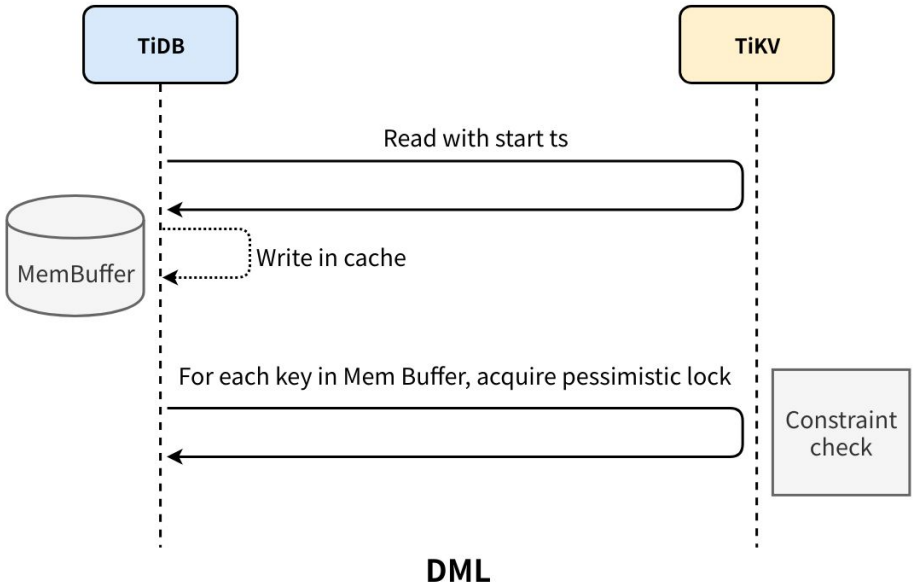
- 执行 DML 时，对修改的 key 加上悲观锁。
- 事务提交同 Percolator：
  - Prewrite 将悲观锁改写为 Percolator 的乐观锁，悲观锁的存在保证了 Prewrite 必定成功。
  - 遇到悲观锁时可以保证该锁的事务未到 Commit 阶段，从而不会阻塞 读。
- 基于 Percolator，实现分布式事务的原子提交、Snapshot Isolation，同时保证了与乐观事务的兼容性，支持混合使用。



基于 Percolator 的悲观事务

Pessimistic Lock

1. 执行 DML 获取到需要修改的 key。
2. 对要修改的 key 加上悲观锁。
3. Prewrite 时的约束检查提前到 Pessimistic Lock 阶段。



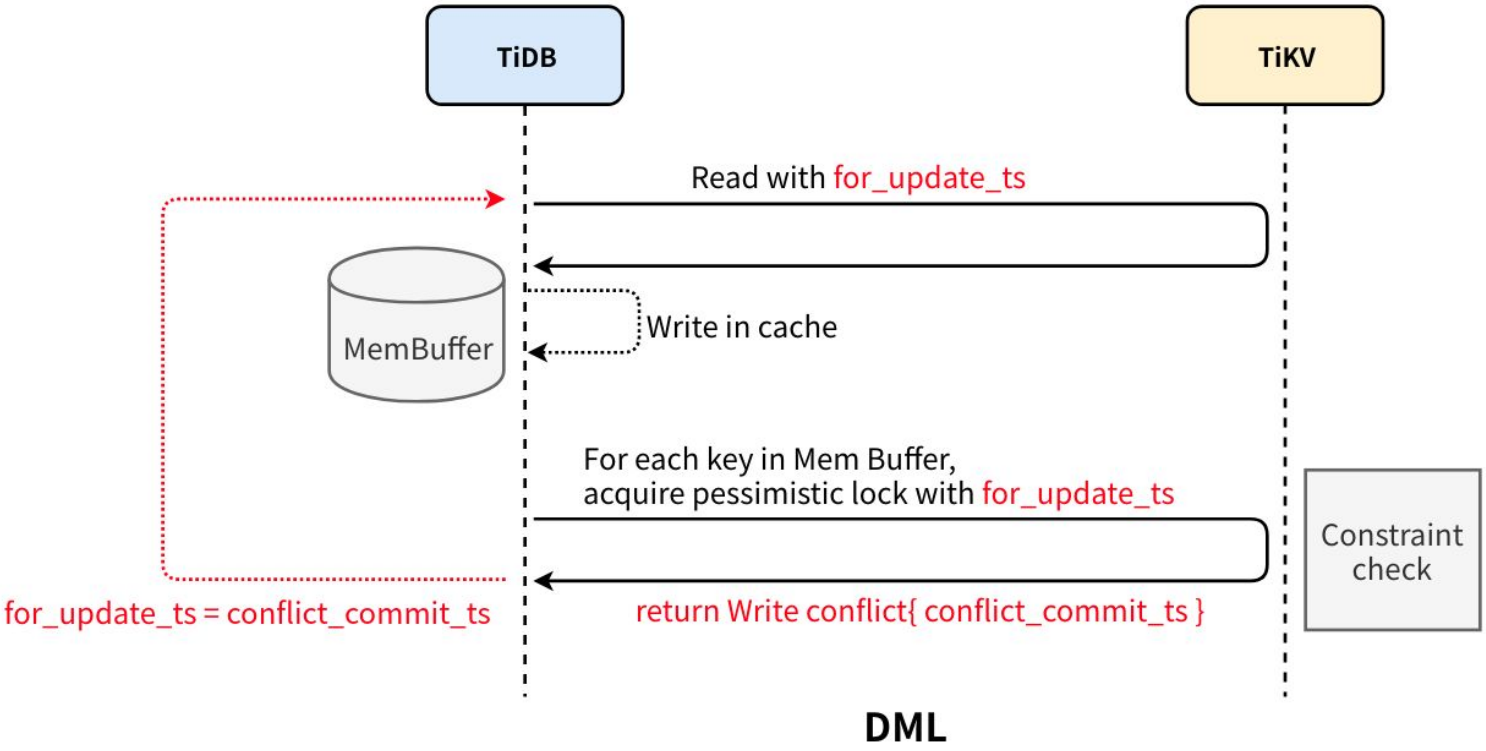
Pessimistic Lock

- 格式和乐观锁相同。
- 写悲观锁时不会写入数据，数据仍在 Prewrite 中写入。

Pessimistic Lock —— 遇到更新的数据

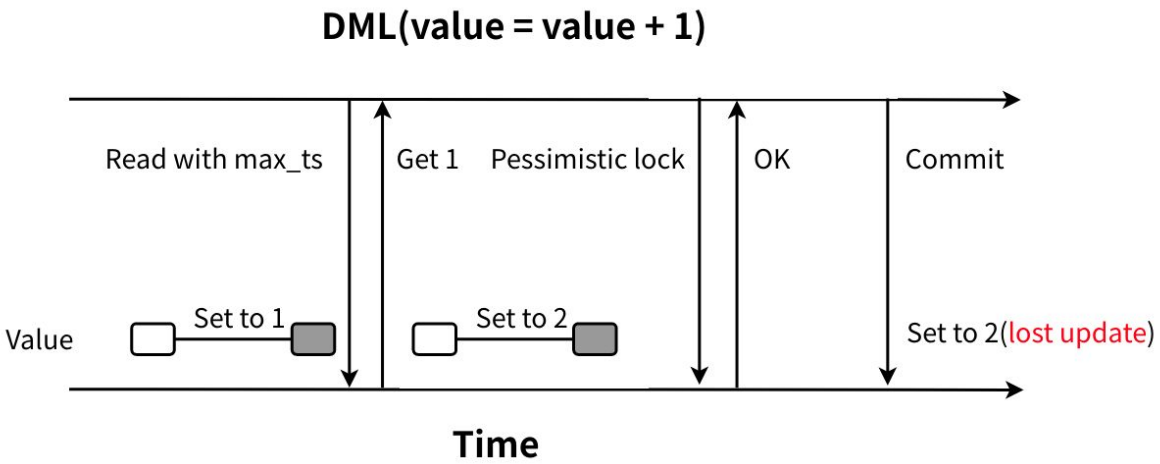
- DML 执行要基于最新已提交的数据，需要重新 执行 DML。
- 除了 start ts 和 commit ts，增加了 for update ts 用于悲观事务读取最新已提交的数据。

Pessimistic Lock —— 遇到更新的数据



为什么不用 max 来读最新的数据？

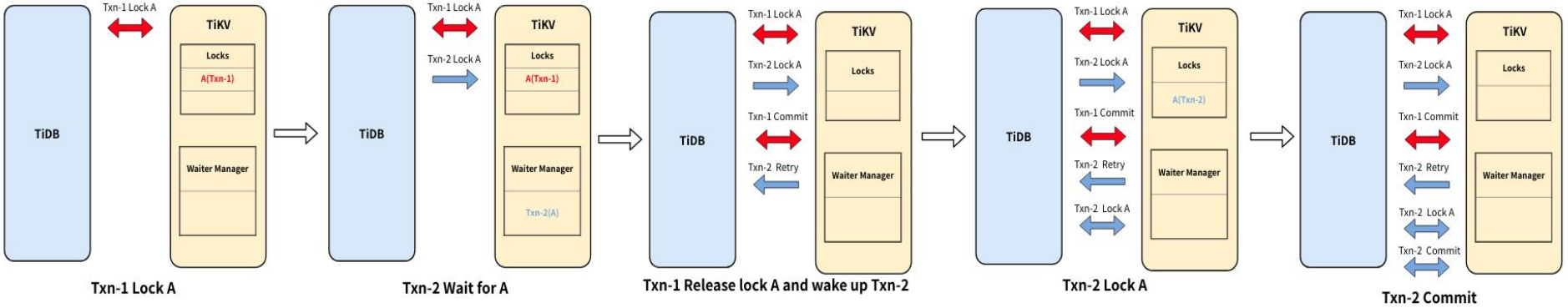
- 要保证加悲观锁时的数据与执行 DML 时的数据一致。
- for update ts 起到了单个语句的 start ts 的作用：保证了读、加锁的 snapshot isolation。



PingCAP

Pessimistic Lock —— 遇到其他事务的锁

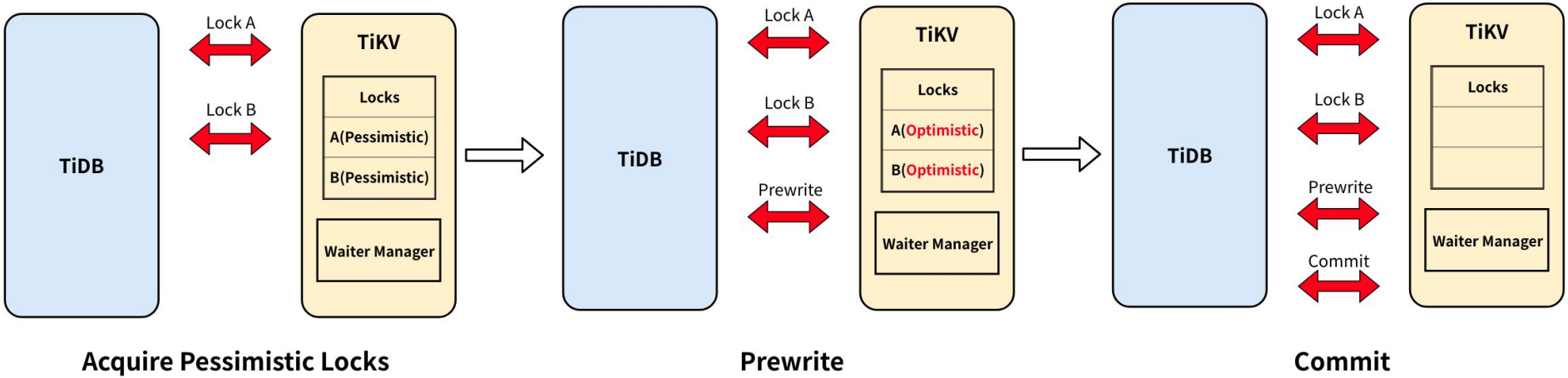
- 先在 TiKV 中等锁(Waiter Manager)，然后再让 TiDB 重试。
- 锁超时处理和乐观锁相同。



PingCAP

Commit

- Prewrite 将悲观锁改写为乐观锁，并写入数据：只需检查悲观锁是否存在且属于该事务。
- 后续同 Percolator。



PingCAP

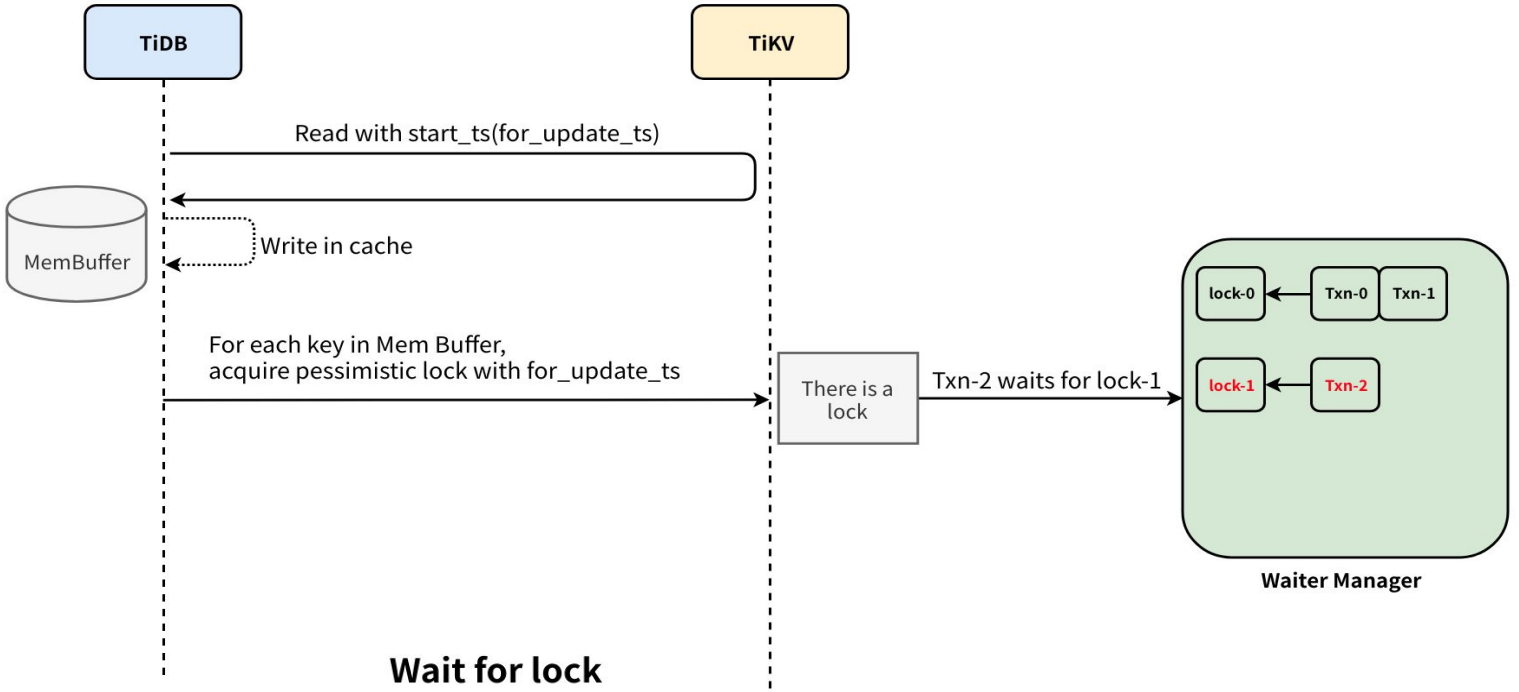
Part III.I - 等锁

在哪里等锁？

- 若在 TiDB 实现，要定期重试，不及时且代价大。
- 在 TiKV 中实现等锁，能够及时得知锁被释放。

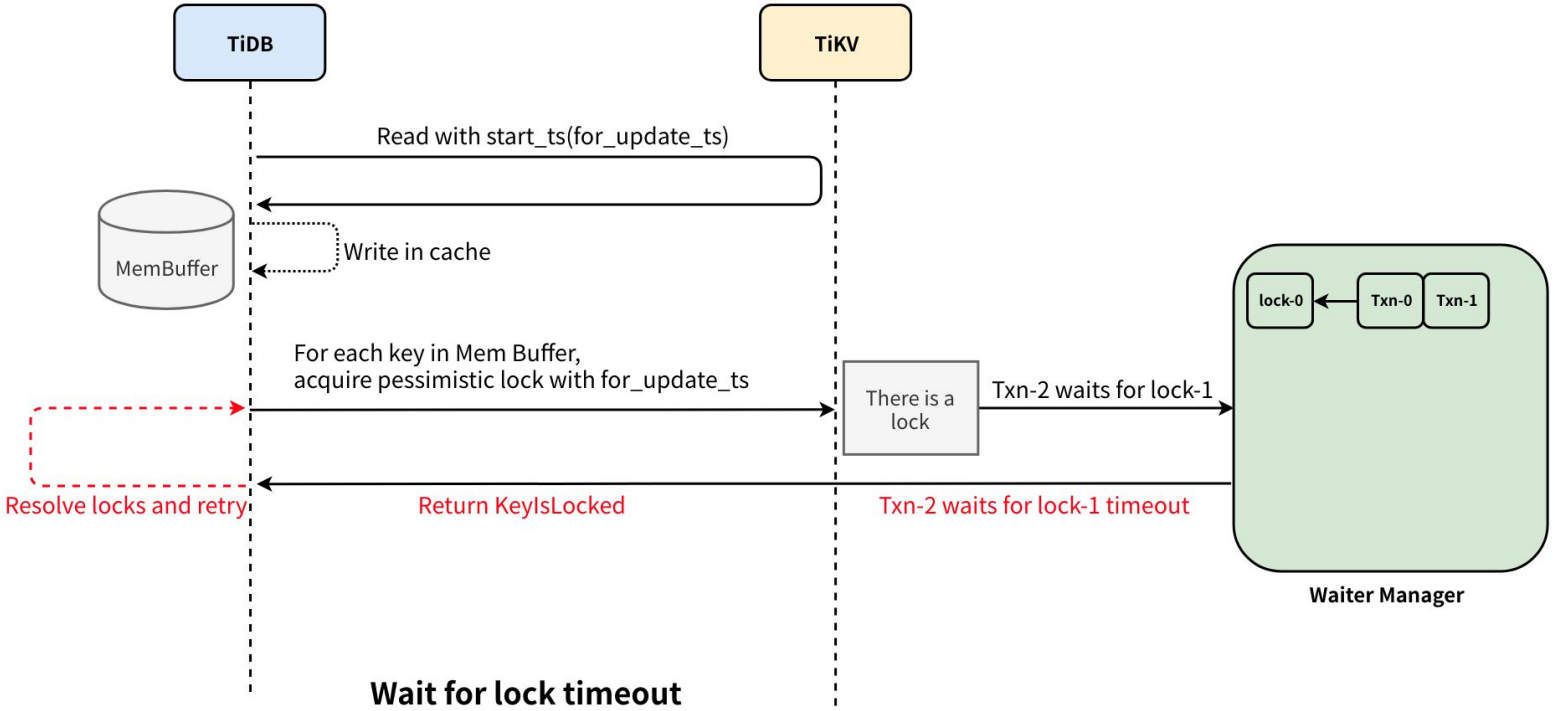
Waiter Manager

- 当遇到锁时，会在 TiKV 中等锁。



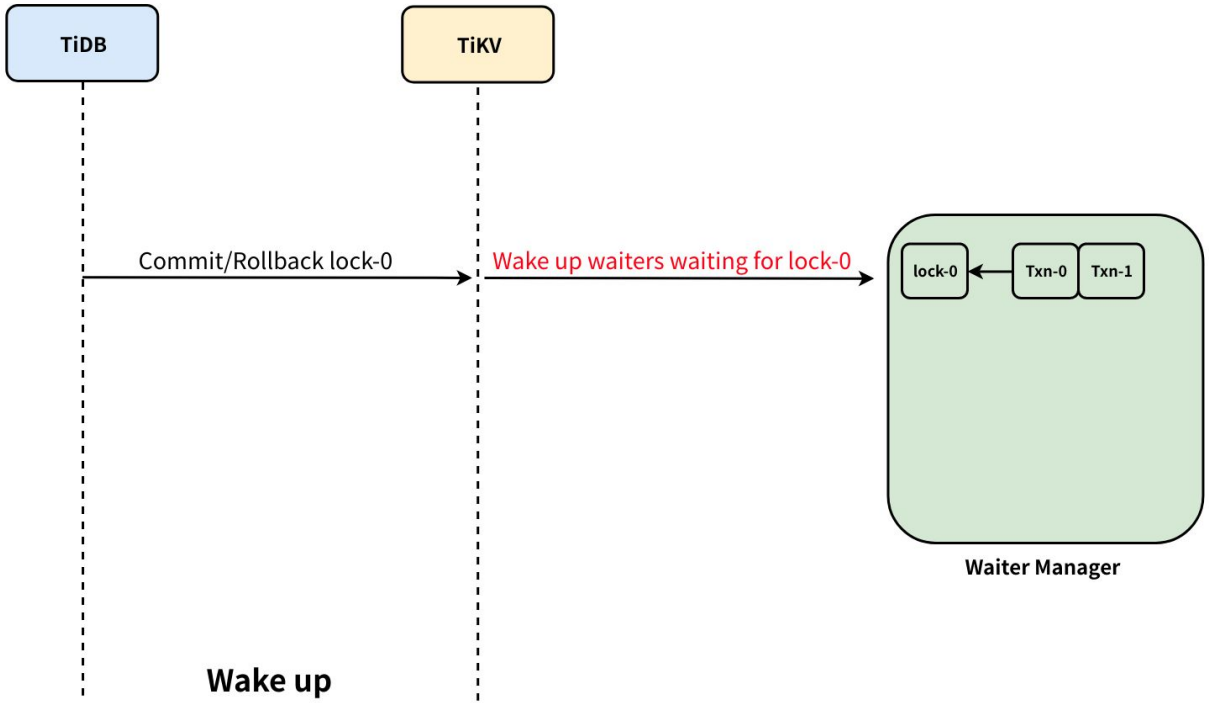
Waiter Manager

- 等锁有超时时间，默认为 3s，可配置。



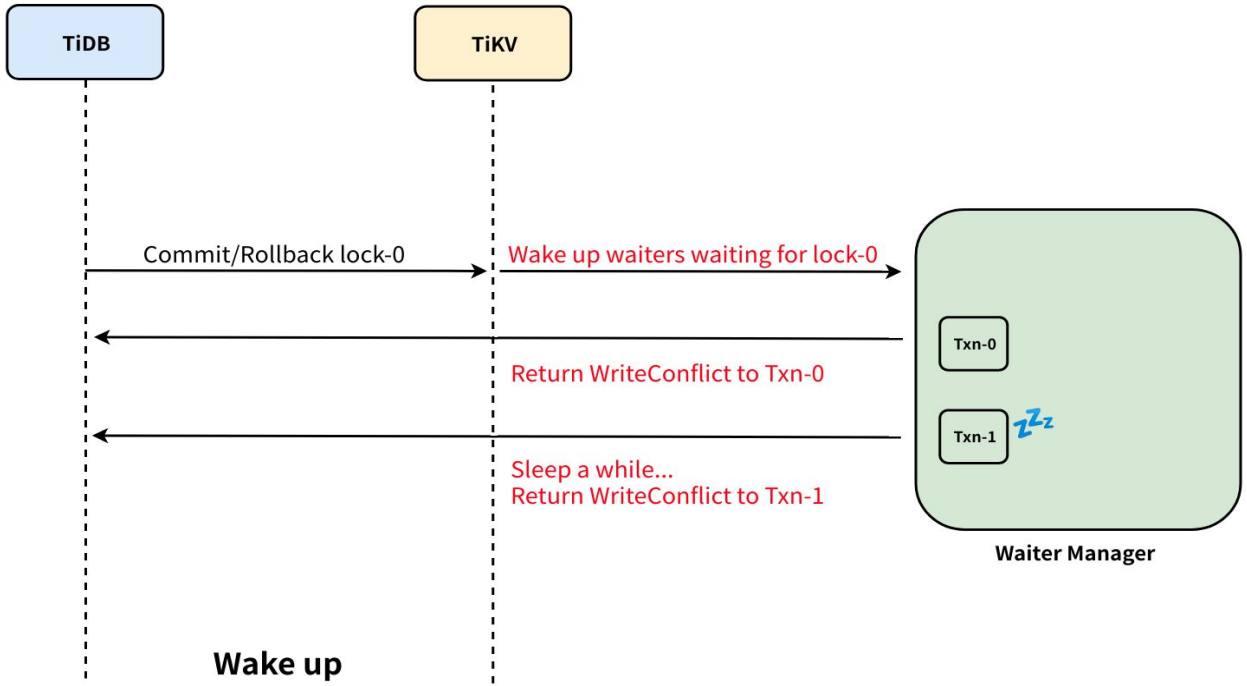
Waiter Manager

- 当锁被释放时(提交或回滚)，会唤醒等待的事务。



Waiter Manager

- 当有多个事务等待同一个锁时，会按照事务的 start_ts 排序来返回响应，让 start_ts 小的更有可能获取到锁。



Waiter Manager

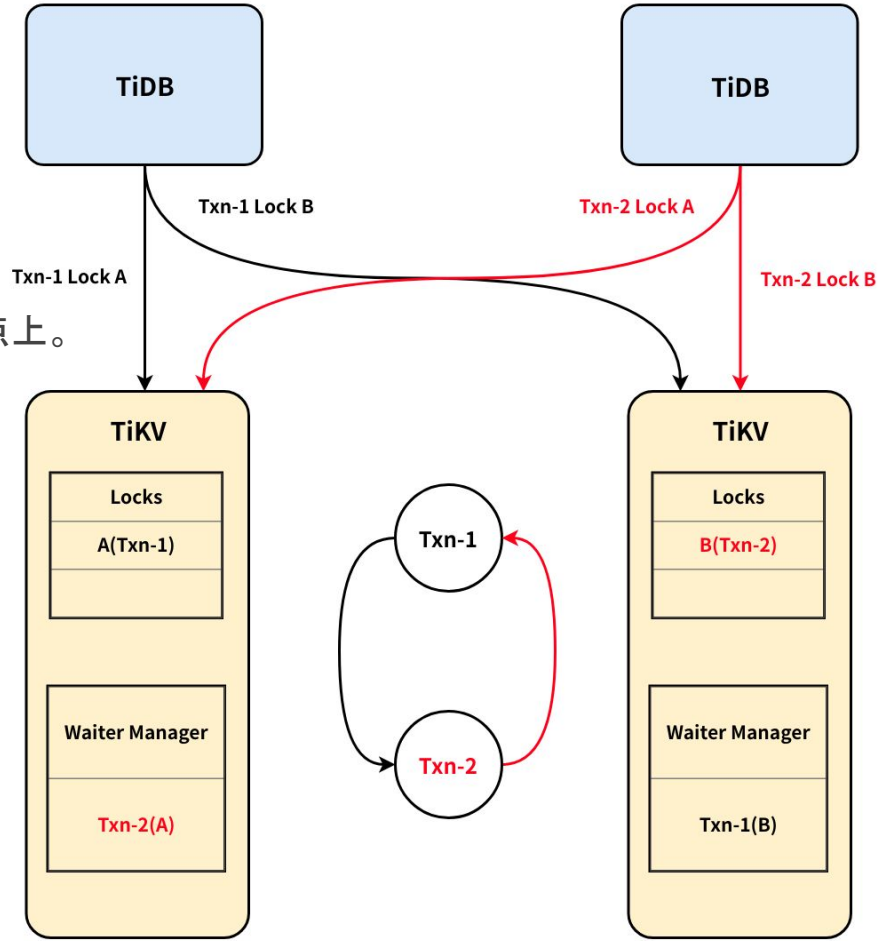
- 在纯乐观事务场景，不会有多余计算和唤醒。

PingCAP

Part III.II - 死锁

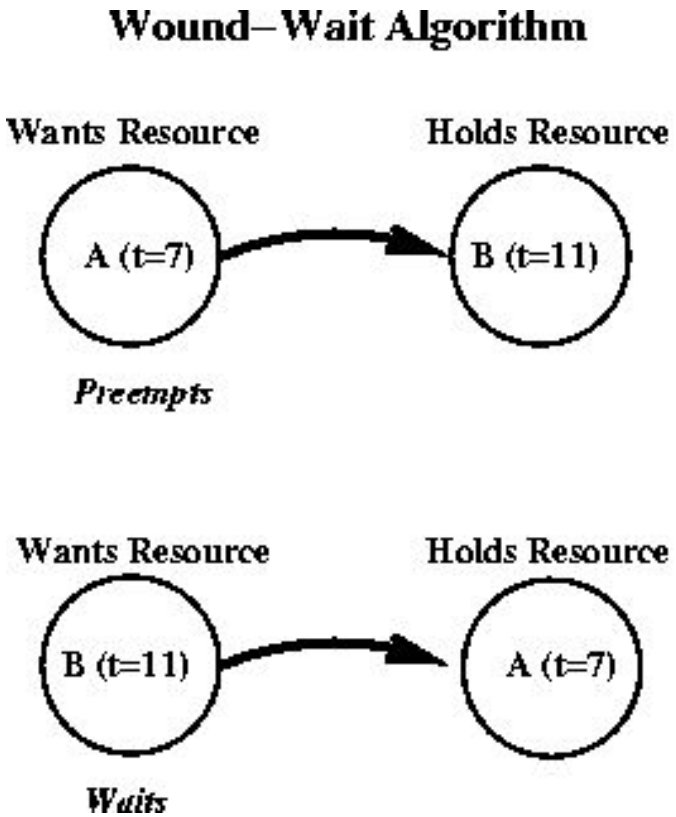
死锁

- 加悲观锁遇到其他事务的锁时，要等待锁被释放。
- 事务间互相等待会导致死锁，且会发生在不同节点上。



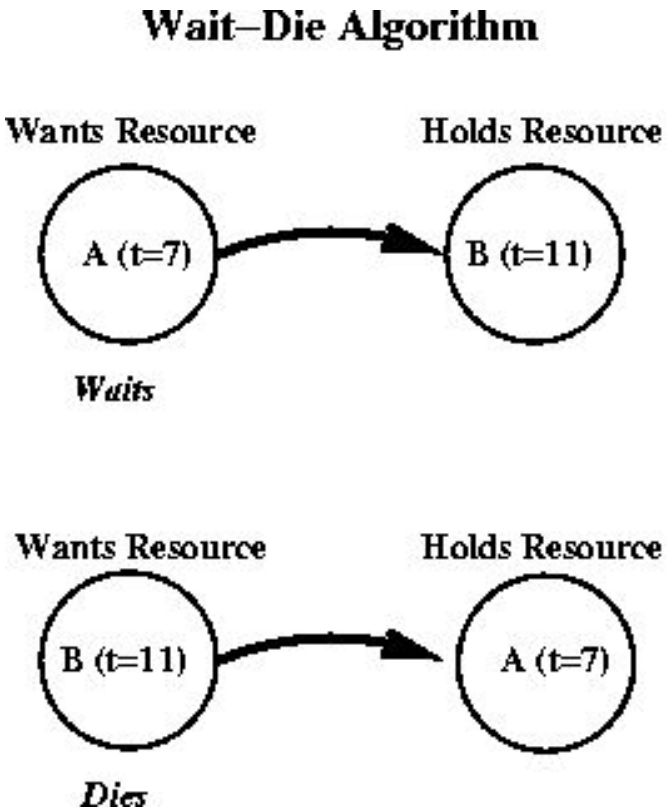
死锁解决方案

- 死锁避免：
  - Wound-Wait



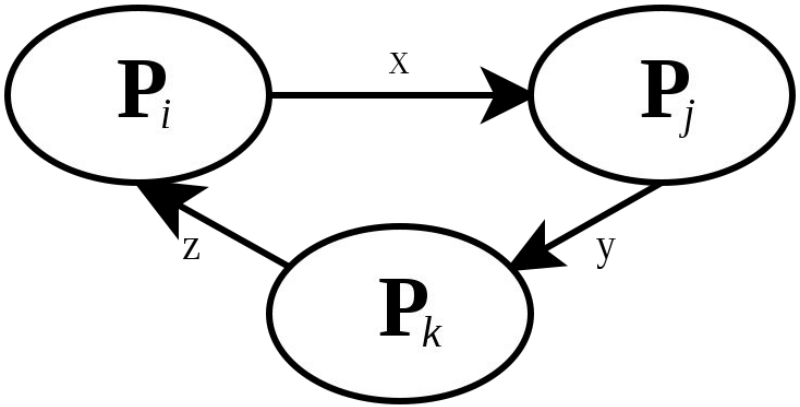
死锁解决方案

- 死锁避免：
  - Wait-Die



死锁解决方案

- 死锁检测：维护全局的等待图，等锁时添加等待路径，拒绝在图中产生环的动作。



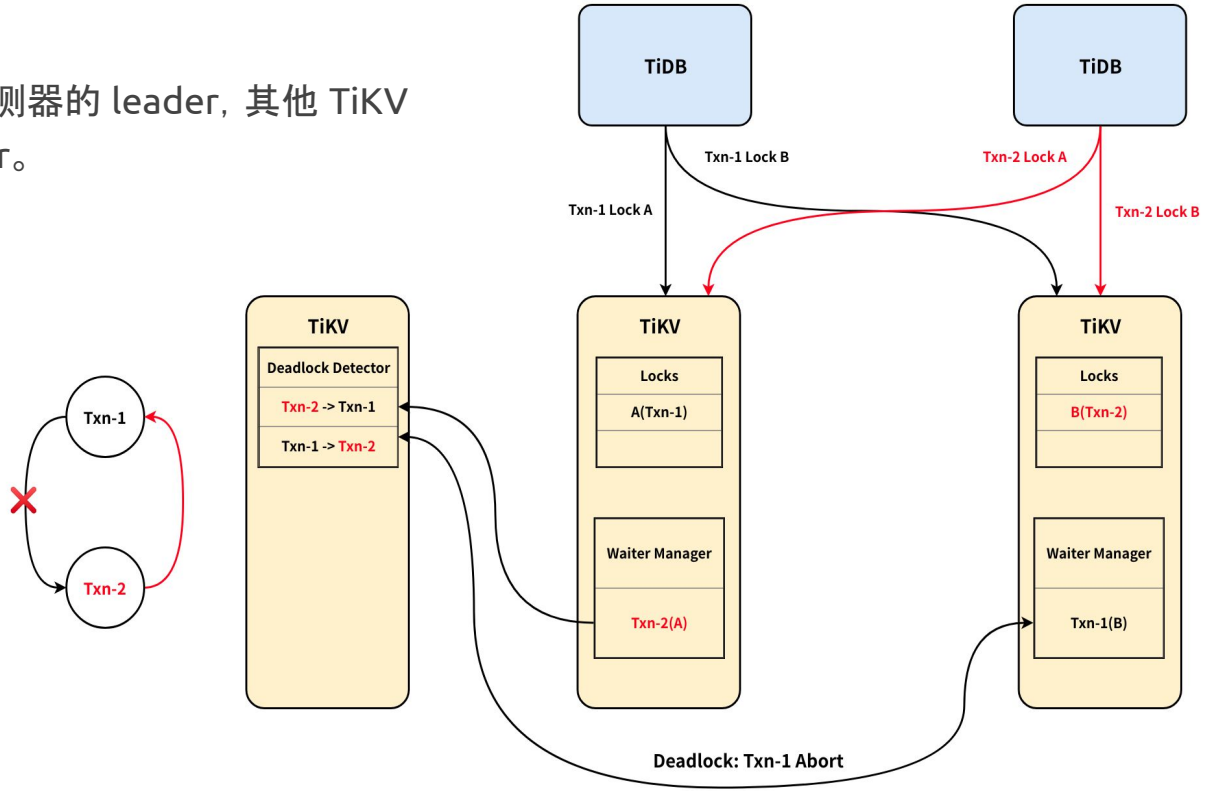
死锁解决方案

- 死锁避免：
  - 实现简单，无中心节点。
  - 只允许一种顺序的等待，事务回滚较多。

- 死锁检测：
  - 准确检测死锁，最小化事务回滚。
  - 需要有中心节点，实现复杂。

TiDB 的实现

- 在 TiKV 中实现死锁检测
  - 一台 TiKV 作为死锁检测器的 leader，其他 TiKV 发送等锁信息到 leader。



Leader 选举

- Region 1 的 leader 作为死锁检测器的 leader。
- 拥有 Region 1 的 TiKV 通过 Raft 得知 leader 变更。
- 其他 TiKV 通过 PD 得知 leader 变更。

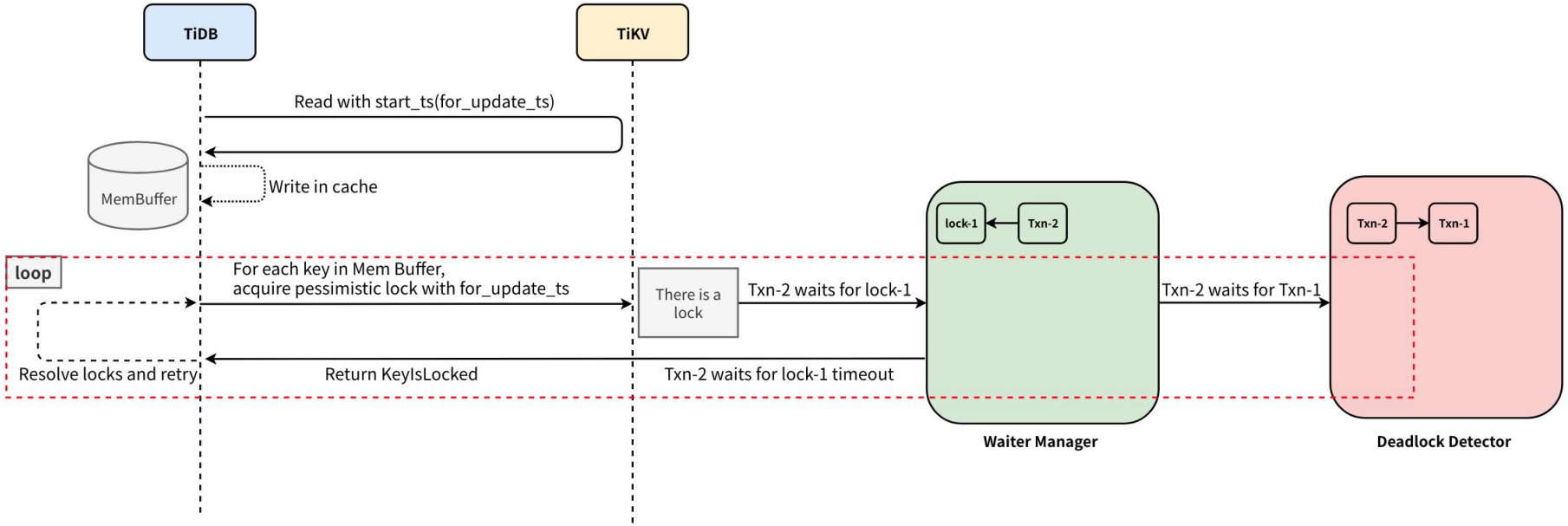
问题

- 等锁请求发送失败，如何处理？
- leader 还没选举出来，怎么处理？
- leader 变更了，新的 leader 没有之前的 wait-for graph，如何处理？

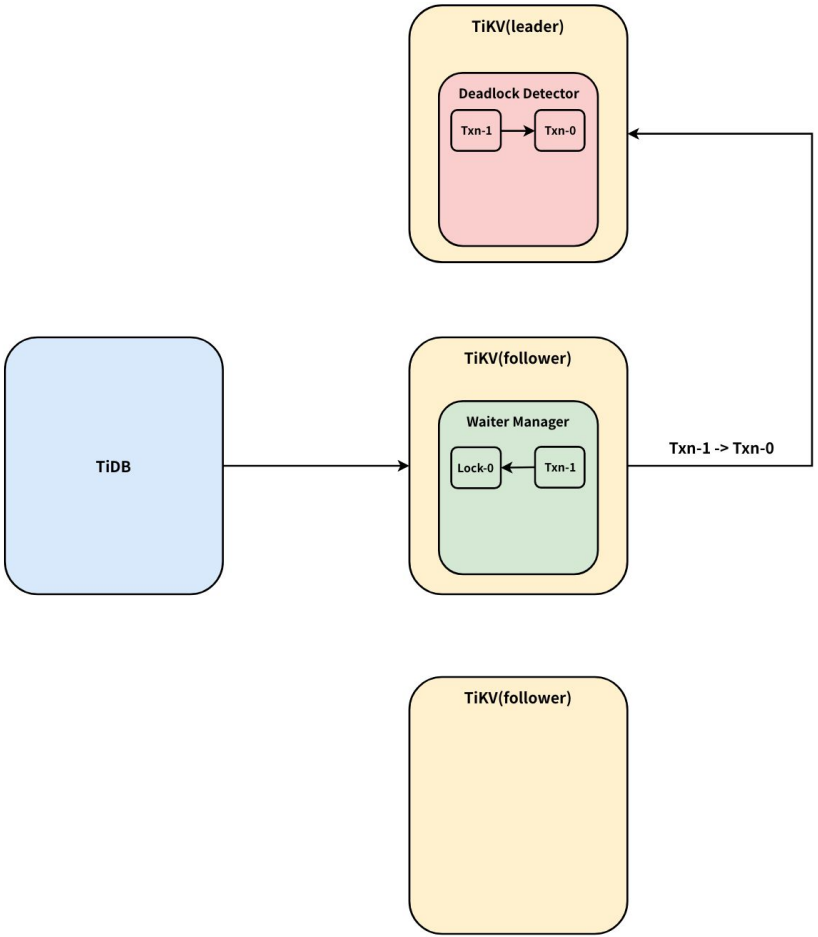
问题

- 等锁请求发送失败，如何处理？
- leader 还没选举出来，怎么处理？
- leader 变更了，新的 leader 没有之前的 wait-for graph，如何处理？
- 重试！

等锁发送失败

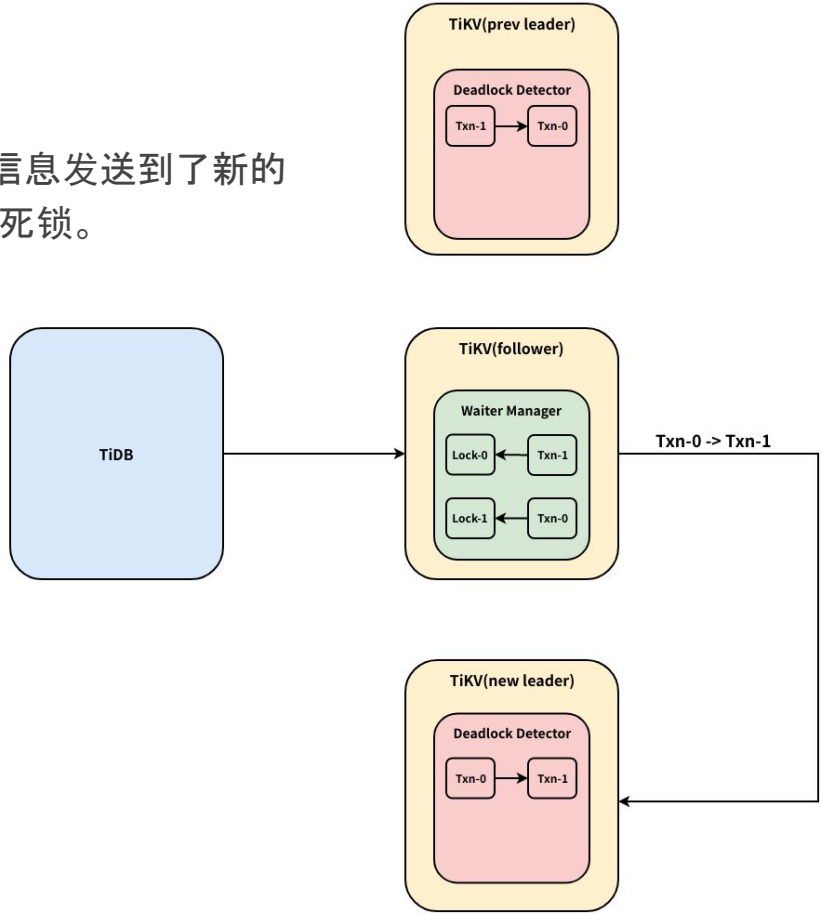

TiDB
TiKV
Read with start_ts(for_update_ts)
MemBuffer
Write in cache
loop
For each key in Mem Buffer,
acquire pessimistic lock with for_update_ts
There is a lock
Txn-2 waits for lock-1
lock-1
Txn-2
Txn-2 waits for Txn-1
Txn-2
Txn-1
Resolve locks and retry
Return KeyIsLocked
Txn-2 waits for lock-1 timeout
Waiter Manager
Deadlock Detector

Leader 变更


TiKV(leader)
Deadlock Detector
Txn-1
Txn-0
TiDB
TiKV(follower)
Waiter Manager
Lock-0
Txn-1
Txn-1 -> Txn-0
TiKV(follower)

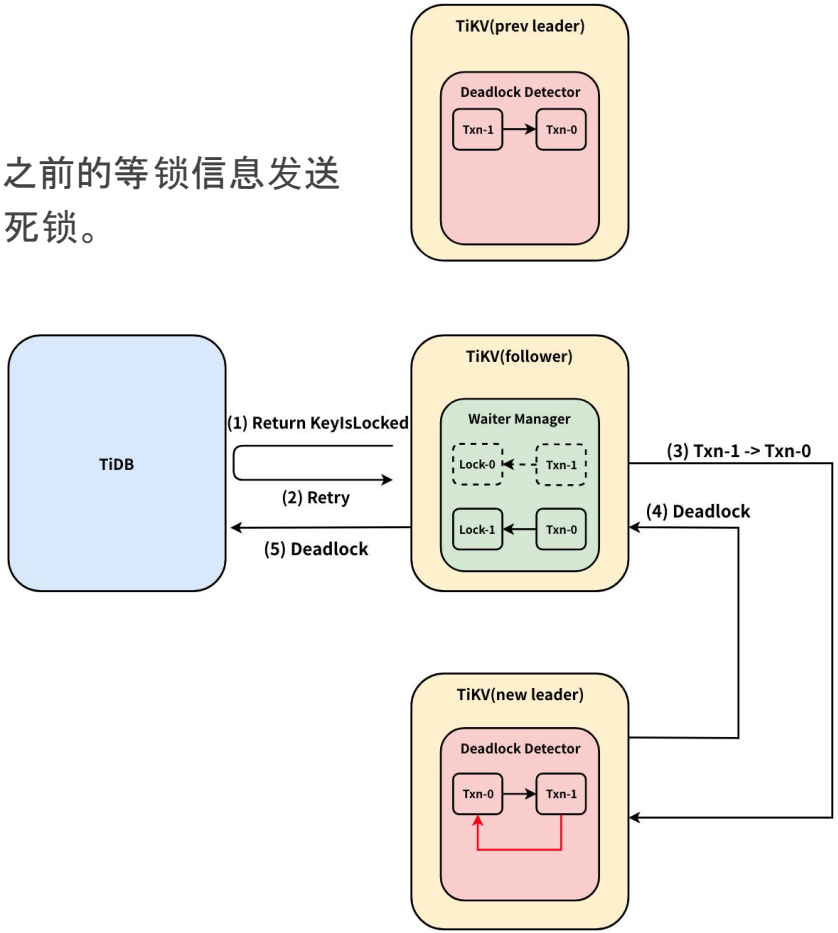
Leader 变更

- leader 变化了，等锁信息发送到了新的 leader 导致未检测出死锁。



Leader 变更

- 等锁超时，TiDB 重试，之前的等锁信息发送到新的 leader，检测到死锁。



单语句 rollback

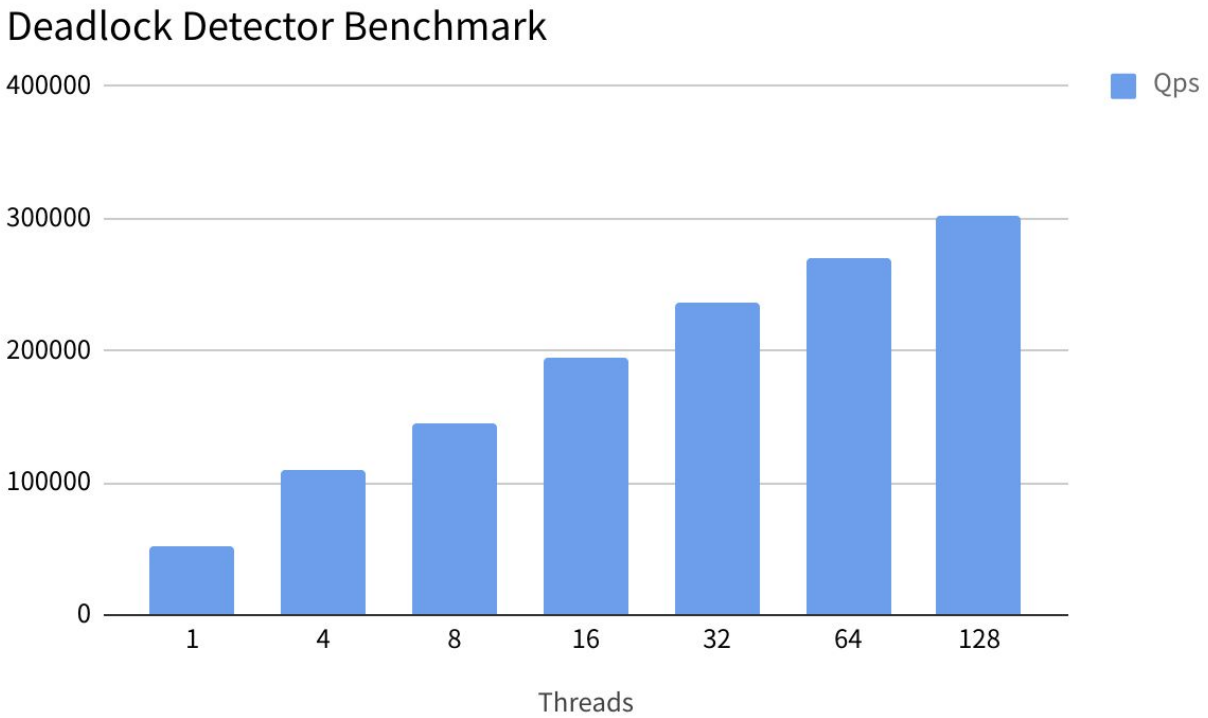
- 两条语句间可能发生死锁：
  - TiDB 按照 region 划分请求，并行加锁，无法控制加锁顺序。

- Percolator 的 Prewrite 有同样的问题。

单语句 rollback

- TiKV 返回给 TiDB 的 deadlock 错误中包含导致死锁的事务正在等待的 key 信息。
- 若 TiDB 发现该 key 为当前语句持有，会删除该语句持有的悲观锁，等待并重试。

死锁检测器性能


Deadlock Detector Benchmark
400000
300000
200000
100000
0
Qps
1
4
8
16
32
64
128
Threads

PingCAP

Part IV - 测试

测试

- TiDB 所有已有的集成测试
- TiDB 所有已有的薛定谔测试
- Jepsen 测试：纯乐观、纯悲观、乐观悲观混合定期测试。
- 新增的薛定谔测试与集成测试。

PingCAP

Part V - 使用

悲观事务使用方法

进入悲观事务模式有以下三种方式:

- 执行 `BEGIN PESSIMISTIC;` 语句开启的事务，会进入悲观事务模式。 可以通过写成注释的形式 `BEGIN /*!90000 PESSIMISTIC */;` 来兼容 MySQL 语法。
- 执行 `set @@tidb_txn_mode = 'pessimistic';` ，使这个 session 执行的所有显式事务（即非 autocommit 的事务）都会进入悲观事务模式。
- 执行 `set @@global.tidb_txn_mode = 'pessimistic';` ，使之后整个集群所有新创建的 session 都会进入悲观事务模式执行显式事务。

乐观事务

Session A	*Session B(no auto retry)*
> begin **optimistic**;	> begin optimistic;
> update test set v = v + 1 where k = 1;	**> update test set v = v + 1 where k = 1;**
> commit;	
	> commit; **ERROR 8005 (HY000): Write conflict, txnStartTS 410234613249343489 is stale [try again later]**

悲观事务

Session A	*Session B*
> begin pessimistic;	> begin pessimistic;
> update test set v = v + 1 where k = 1;	
	> update test set v = v + 1 where k = 1; **block...**
> commit;	**Query OK, 1 row affected (0.00 sec)**
	> commit; **Query OK, 0 row affected (0.00 sec)**

SELECT … FOR UPDATE

Session A	*Session B*
> begin pessimistic;	
> select * from test;(SI) +------+------+ | k | v | +------+------+ | 1 | 1 | +------+------+	
	> update test set **v = v + 1** where k = 1;
> select * from test for update;(RC) +------+------+ | k | v | +------+------+ | 1 | 2 | +------+------+	

DEADLOCK

Session A	*Session B*
> begin pessimistic;	> begin pessimistic;
> update test set v = 2 where k = 1;	**>** update test set v = 1 where k = 2;
> update test set v = 1 where k = 2;	
	> update test set v = 2 where k = 1; **ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction**
> commit;	

PingCAP

Thank You!