COCO新的分布式OLTP数据库，支持基于epoch的提交和复制，把事务划分成epoch作为提交和恢复的基本单位，减少了2pc和同步复制的开销
一般来说用2pc保证一致性，用2PL OCC实现并发控制
在最后信息的传播的时候是异步的

contribute

1.coco
2.两种OCC算法的设计，性能优化，并且支持快照事务

Detail

deterministic datebase在多个节点上确定性的运行事务来避免使用2pc
Aria支持确定性事务执行无需任何输入，但是高争用负载性能差
deterministic datebase复制事务的输入，需要知道事务的读写集
Coco复制事务的输出
Coco系统用户可以自主选择同步复制的方式，例如primary-backup replication或者state machine replication，然后把同步复制和异步复制的优点结合

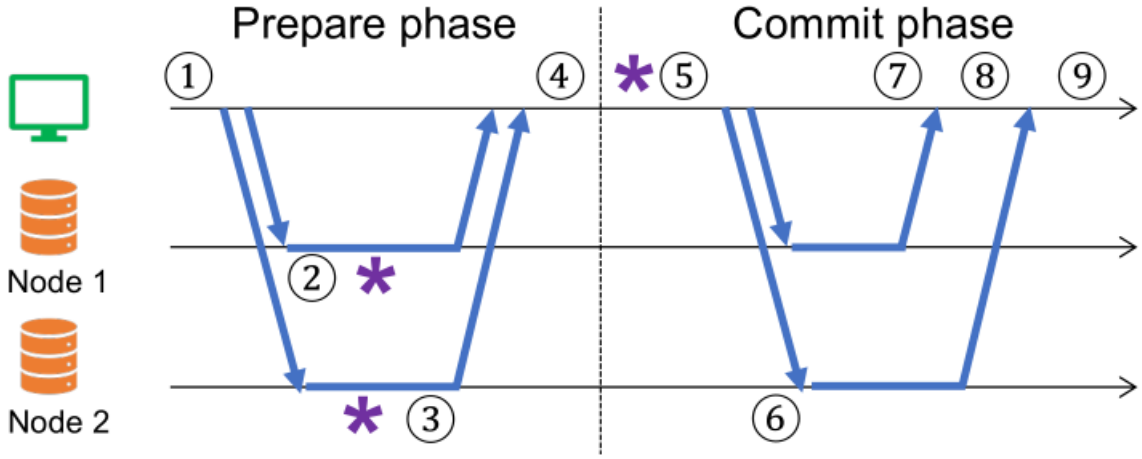

Figure 1: Failure scenarios in epoch-based commit

准备阶段：准备一下事务的写记录以及当前准备提交的事务ID
coordinator将准备消息发给每个participants node，coordinator可以是系统中任意node，也可以是外部独立节点，participants准备完成后给coordinator发送ack

提交阶段：coordinator确定这个epoch是不是可以提交，可以提交就写一个提交记录

coordinator确定epoch是否可以提交，不可以提交就将epoch内所有事务都abort，可以的话就写提交记录，然后发送给每个participants

容错

失败分为9类

1.coordinator发送准备请求之前(abort当前epoch)

2.部分participants接受到准备请求之后(abort当前epoch)

3.全部participants接受到准备请求之后(abort当前epoch)

4.coordinator接收到所有participants的ack之前(abort当前epoch)

5.在coordinator写提交记录之后(视为提交epoch)

6.在部分participants node收到提交请求前(视为提交epoch)

7.coordinator收到任何确认之前(视为提交epoch)

8.coordinator收到部分确认之后(视为提交epoch)

9.coordinator收到全部确认之后(此时已经进入下一个epoch)

局限性

1.基于epoch的提交和复制会增加每个事务的延迟，不适合需要极低延迟的workload

2.如果一个epoch有一个长事务，就影响其他事务的提交

3.发生故障全都得abort重新执行

事务生命周期

初始化事务的是协调节点其他是participants节点

执行阶段

从数据库读取记录，在读取集中维护记录的本地副本

提交阶段

1.锁定写集，只锁定数据的primary replica

2.验证读集以及TID的赋值

3.写入到数据库，可异步 先异步的写入primary replica然后epoch结束后再传给其他节点

1.OCC在2pc的commit阶段执行，COCO的2pc和普通2pc区别只在于普通的是准备和提交一个事务，而这个提交协议是准备和提交一个epoch的事务

2.initiating a transaction的是coordinator节点，在执行阶段时，所有数据已经到了各个节点了

代码应该执行在coordinator上，有事务的读写集,其中读到的locked是判断有没有被除自己以外的事务锁定，primary node和coordinator node不同，primary node是对应每个记

录有个主节点，coordinator node是对应事务发起的那个节点

将事务写集中所有记录的primary node给锁定上，这个tid是新生成的，如果这条记录在写集中，那么就遵循Thomas写规则，record的tid用最新的tid覆盖。然后ok的判断是判断这个锁有没有被其他的事务占用，如果占用了，他为了避免死锁，采用了no_wait的死锁预防策略，就是一旦有其他的事务占用锁了，那就直接abort就好。后面的TID不同就是可能这条记录被覆盖掉了，所以就abort，写写冲突。

Locking the write set

P T \| O C C

```
1  for record in T.WS:
2      ok, tid = call_n(lock, record.key)
3      if record not in T.RS:
4          record.tid = tid
5      if ok == false or tid != record.tid:
6          abort = true
```

再次确认此时的记录被当前事务lock并且没被其他并发事务修改

Validating the read set

```
for record in T.RS \ T.WS:
    # begin atomic section
    locked, tid = call_n(read_metadata, record.key)
    # end atomic section
    if locked or tid != record.tid:
        abort()
```

没有被abort就先提交到primary node的数据库，解锁这条记录，然后再异步的把这数据更改到其他participants node

Writing back to the database

```
for record in T.WS:
    call_n(db_write, record.key, record.value, record.tid)
    call_n(unlock, record.key)
    for i in get_replica_nodes(record.key) \ {n}:
        call_i(db_replicate, record.key, record.value, T.tid)
```

wts代表记录何时写入，rts代表在[wts, rts]区间都可以读

大部分相同，wts是记录写的时间，也相当于之前physical time的TID，rts是可以读取的时间，因为在当前时间有读写集，所以rts需要更新到最新的时间

locking the write set

LTI OCC

```
1  for record in T.WS:
2      ok, {wts, rts} = call_n(lock, record.key)
3      if record not in T.RS:
4          record.wts = wts
5      if ok == false or wts != record.wts:
6          abort()
7      record.rts = rts
8
```

TID>写集中的rts并且TID>=读集中的wts 中的最小的时间戳，TID之后就是读集中的记录已经全部写入，可读的时间，读集中的rts<Tid时需要尝试扩展,>Tid就无需验证，(TID>写集中的rts是否没有存在的意义)，事务会在两种情况abort 1.事务记录被其他并发事务修改也就是wts被更改 2.rts<T.tid 此时需要扩展rts，让该记录在TID时刻可读，但是如果被其他事务锁定，此时也需要abort。否则就把rts扩展到tid

validating the read set

```
for record in T.RS \ T.WS:
    if record.rts < T.tid:
        # begin atomic section
        locked, {wts, rts} = call_n(read_metadata, record.key)
        if wts != record.wts or (rts < T.tid and locked):
            abort()
        call_n(write_metadata, record.key, locked, {wts, T.tid})
        # end atomic section
```

OCC变体

将Silo的OCC和Tictoc的OCC在分布式环境使用

1.Silo和Tictoc中的TID符合基于epoch的提交和复制

2.使Silo和Tictoc中的snapshot transaction往返次数减少一次

PT-OCC

如何把silo的单节点并发控制用到coco中，不同的records可能有不同的primary node

事务先锁定写集中的记录，然后验证读集的记录，存在record被其他事务锁定或者被其他事务修改就abort

成功验证读取集后生成TID

TID>读写集的任何TID并且大于上一次worker thread选择的TID

保证可串行性，在验证读取record的TID之前，所有写入记录已经被锁定

LT-OCC

把Tictoc单节点并发控制算法用到coco

[wts, rts]wts代表记录何时写入，rts代表在[wts, rts]区间都可以读

首先将每个records锁定到写集

TID>写集中的rts并且TID>=读集中的wts然后找到最小的时间戳，这就是TID

读取集的records的rts进行验证，当rts<TID，此时需要验证，因为tictoc是在[wts, rts]区间都可以读，然后事务会尝试去延长rts当

1.wts已经更改

2.记录被其他事务锁定(一开始不都把record锁定了吗，也没有解锁，为什么会被其他事务锁定)

Snapshot transaction

对silo和tictoc进行更改，以支持快照隔离

PT-OCC

验证读集，无需为写集的每个记录hold锁，只要未检测到更改，就可以确保读取一致快照

LT-OCC

锁定写集的记录后会分配TID给可序列化事务，为了支持Snapshot transaction给每个事务分配一个最小的TIDsi>=读集的wts

Parallel locking and validation optimization

commit的前两个步骤同时进行，锁定写集和验证读集并行进行