


确定性数据库简述

2021.11.26

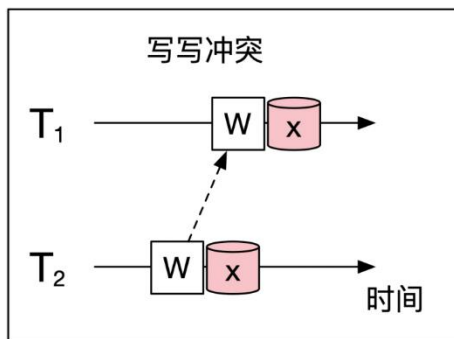
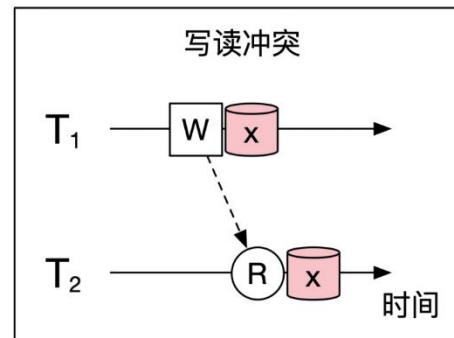
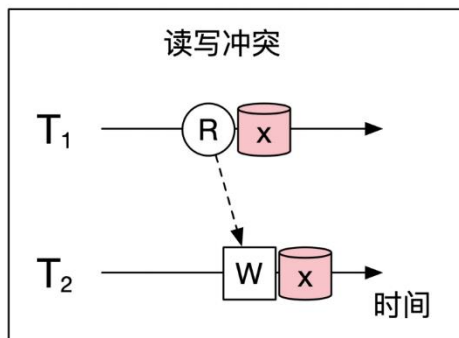


目录

- 
- A horizontal bar consisting of two segments: a blue segment on the left and an orange segment on the right.
- 确定性数据库
 - Calvin
 - PWV
 - Aria
 - 总结



确定性数据库

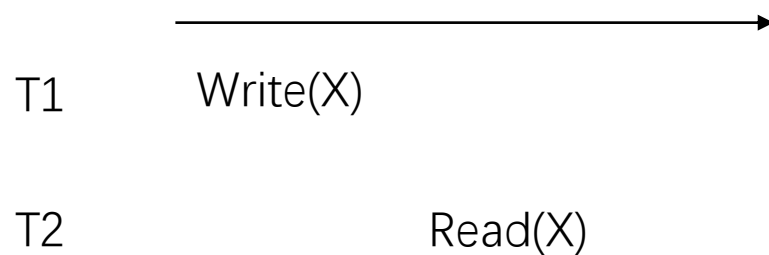


https://blog.csdn.net/m0_68620448

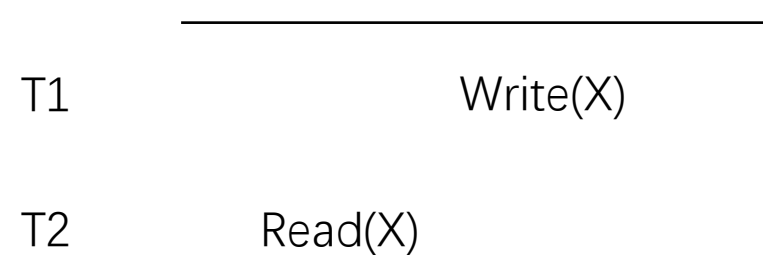
事务冲突可以分为三种冲突读写，写读，写写冲突



确定性数据库



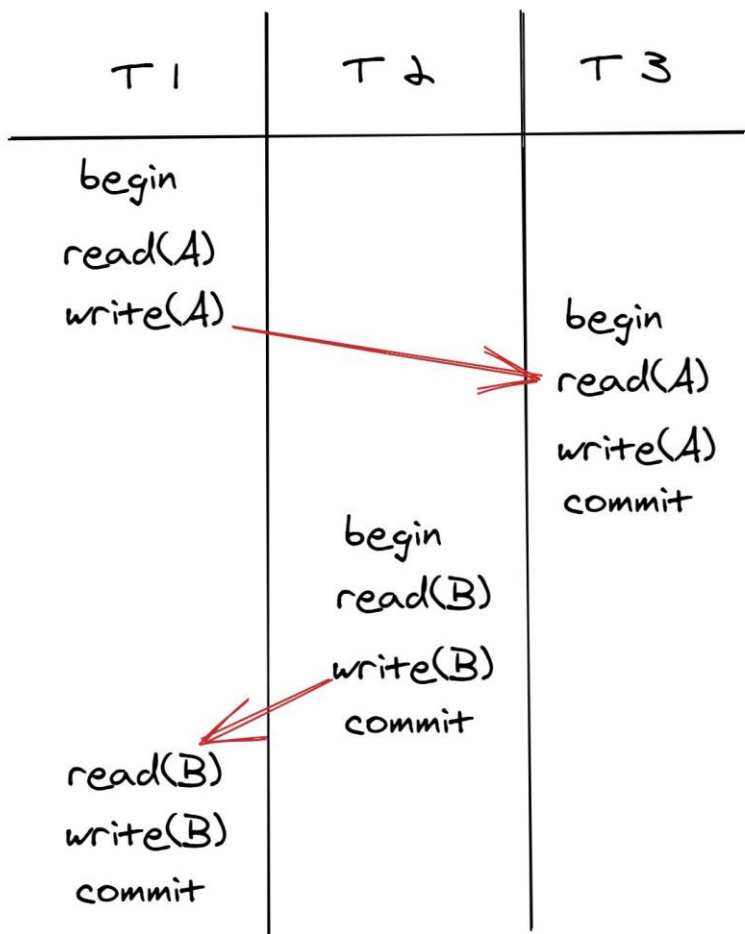
$Write_1(X) < Read_2(X)$
T1在T2之前执行



$Read_2(X) < Write_1(X)$
T2在T1前执行

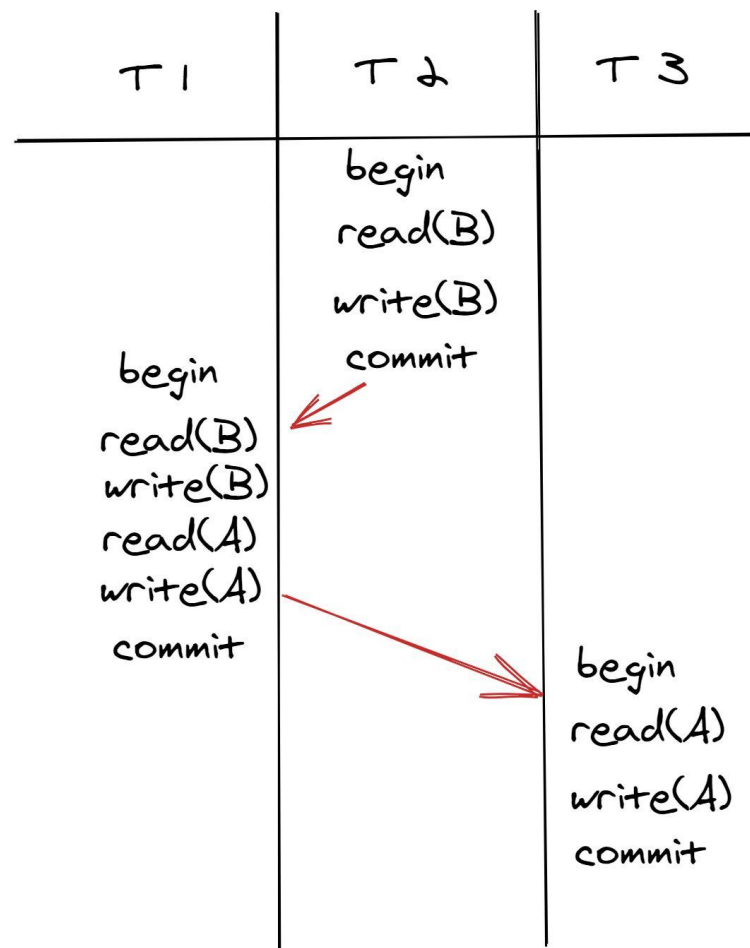


确定性数据库



$Write_1(A) < Read_3(A)$

T1在T3之前执行

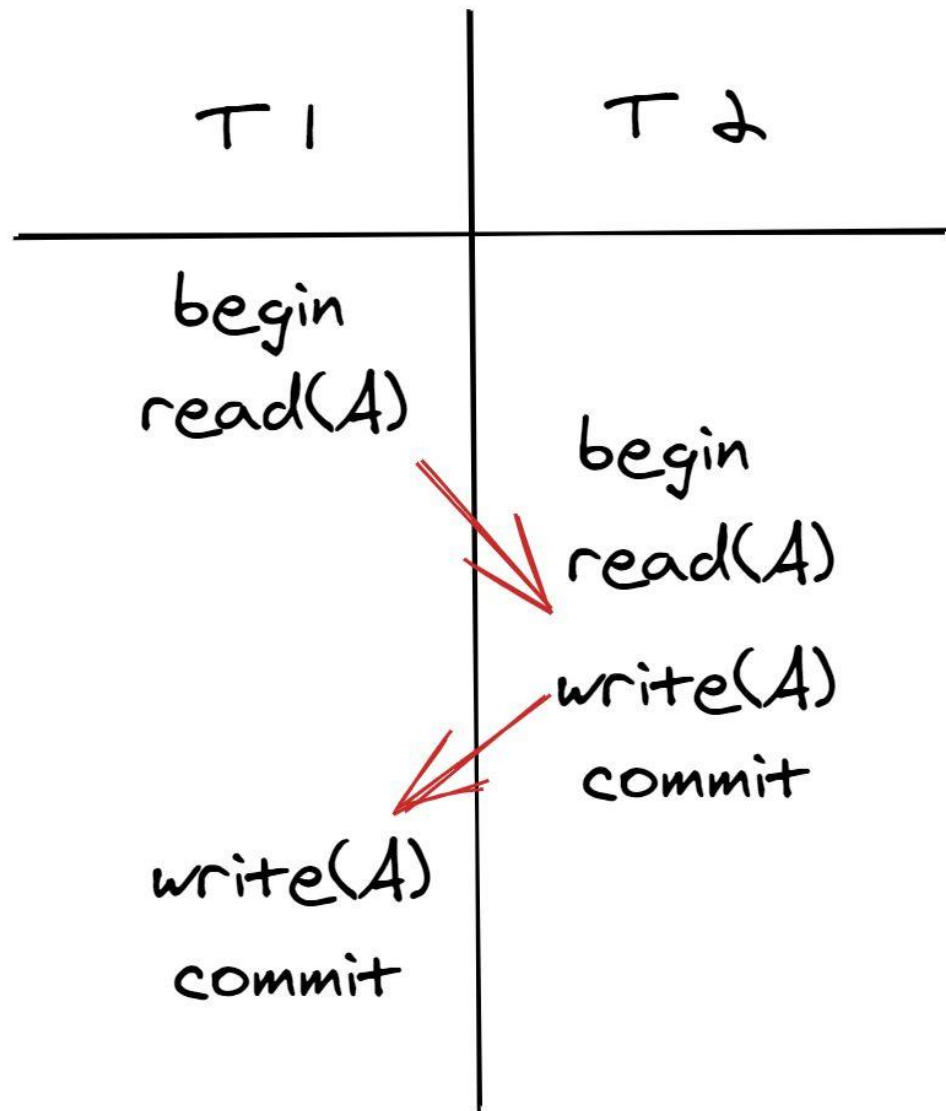


$Read_2(B) < Write_1(B)$

T2在T1之前执行



确定性数据库



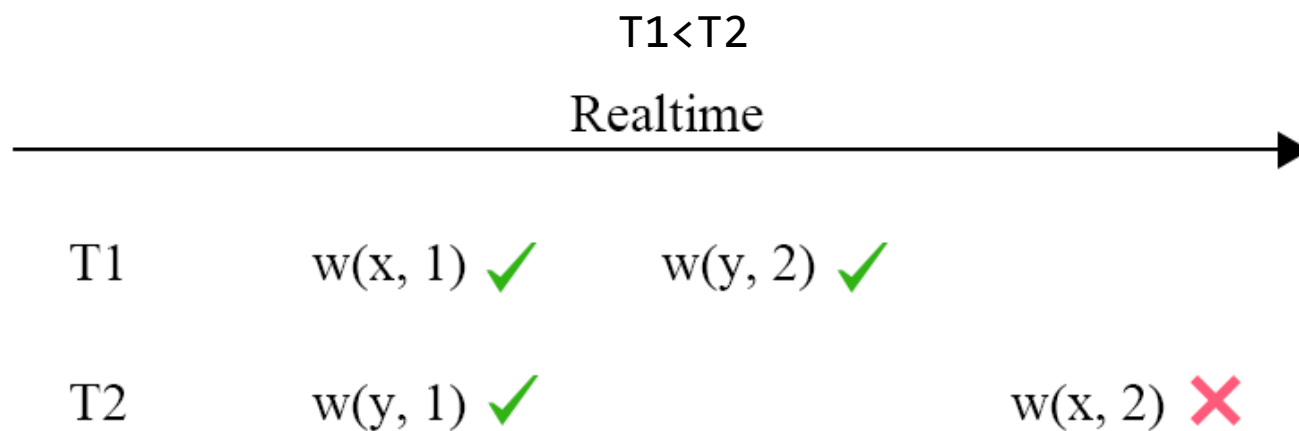
$Read_1(A) < Write_2(A)$
T₁在T₂之前执行

$Write_2(A) < Write_1(A)$
T₂在T₁之前执行



确定性数据库

传统数据库中为了保证事务执行序列满足操作的偏序关系，实现串行等价，往往通过加锁方式限制事务执行。



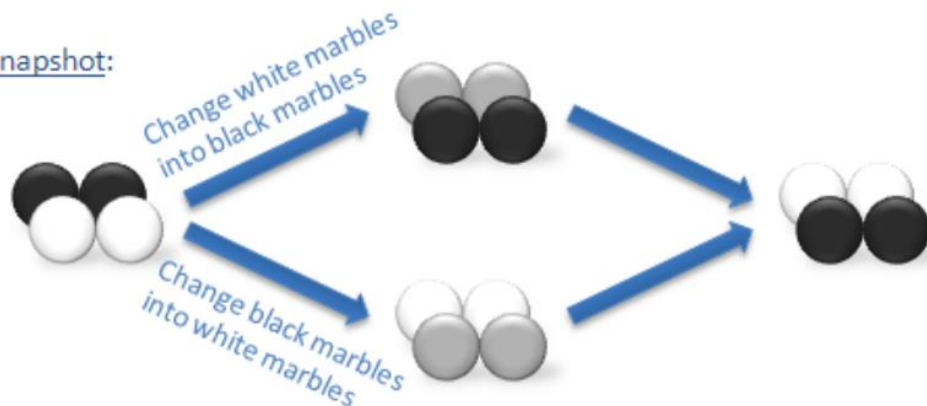
但是加锁方式容易出现死锁的问题，导致事务回滚的发生。

实现事务的可串行化隔离级别，是非常困难的

Serializable:



Snapshot:



T1:

读取所有白色棋子
将读取到的白色棋子变为黑色

T2:

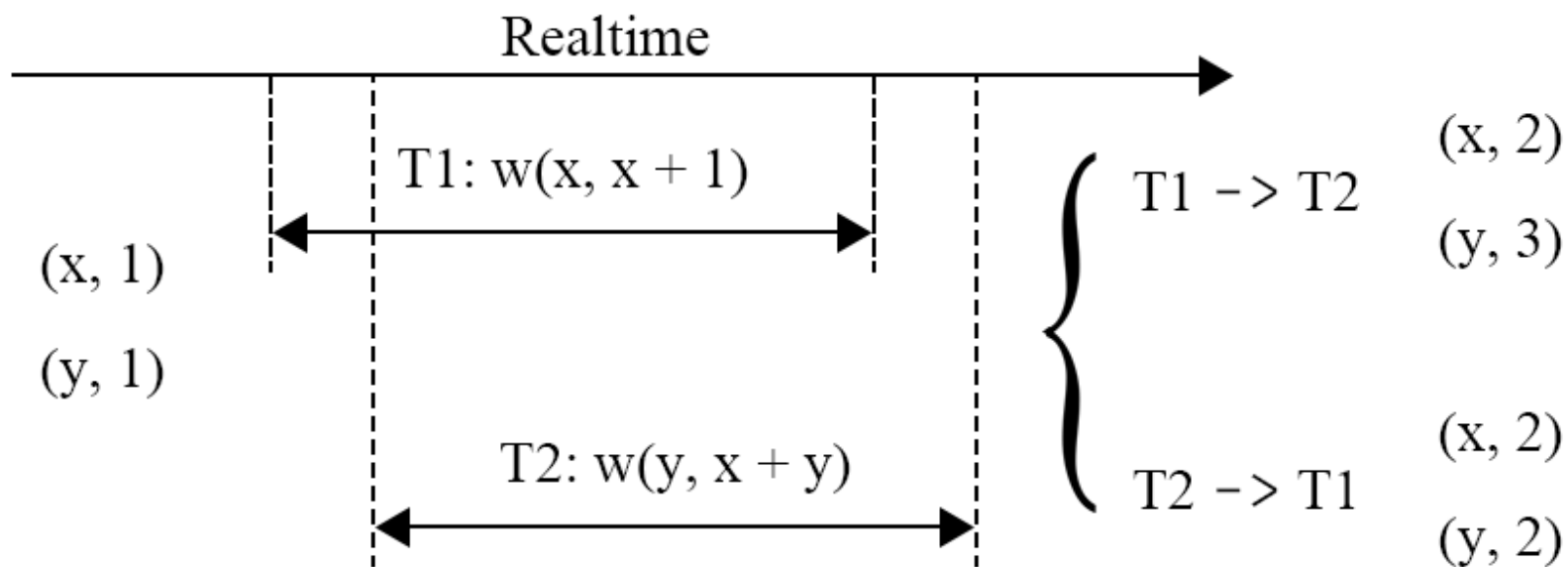
读取所有黑色棋子
将读取到的黑色棋子变为白色

T1和T2之间实际存在着写读冲突，但是
Snapshot的隔离级别下无法阻止这种冲突发生



确定性数据库

什么是确定性？

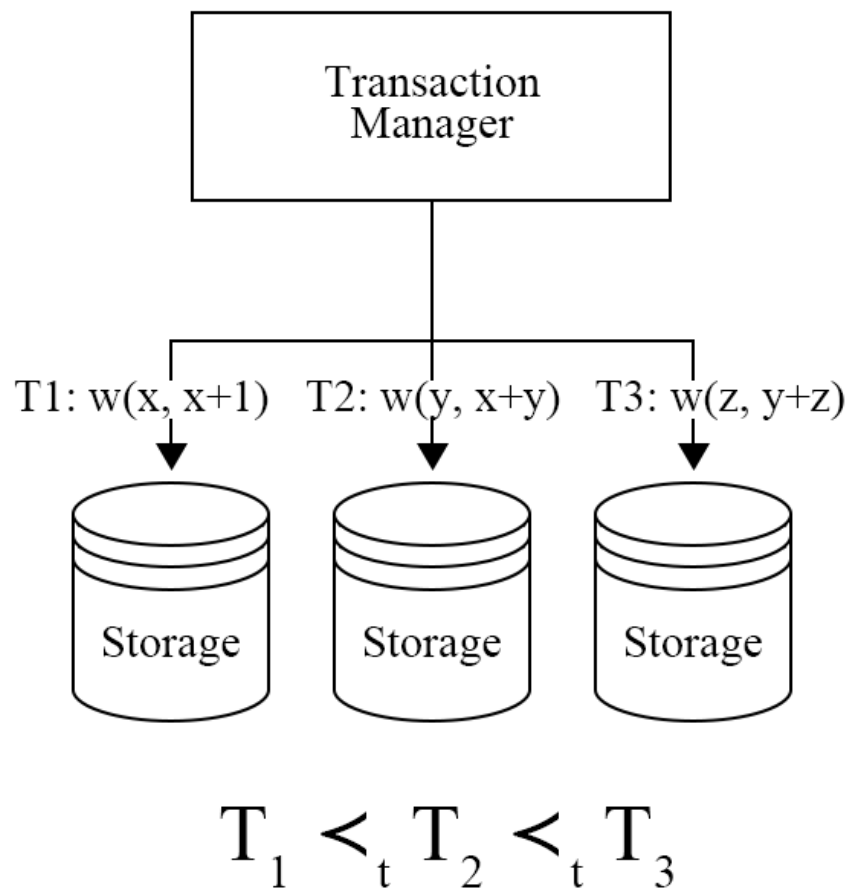


确定性数据库的确定性指的是执行结果的确定性，
这里执行结果的确定性不是用户视角的确定性。




确定性数据库

什么是确定性？



对于一组经过事务管理器确定了执行顺序的事务，
在确定型数据库中它们执行的结果确定的。

目录

- 
- 确定性数据库
 - Calvin
 - PWV
 - Aria
 - 总结



Calvin

Calvin发表于2012年,
Calvin试图将事务执行前对于事务进行排序的方式分布式事务的难题。

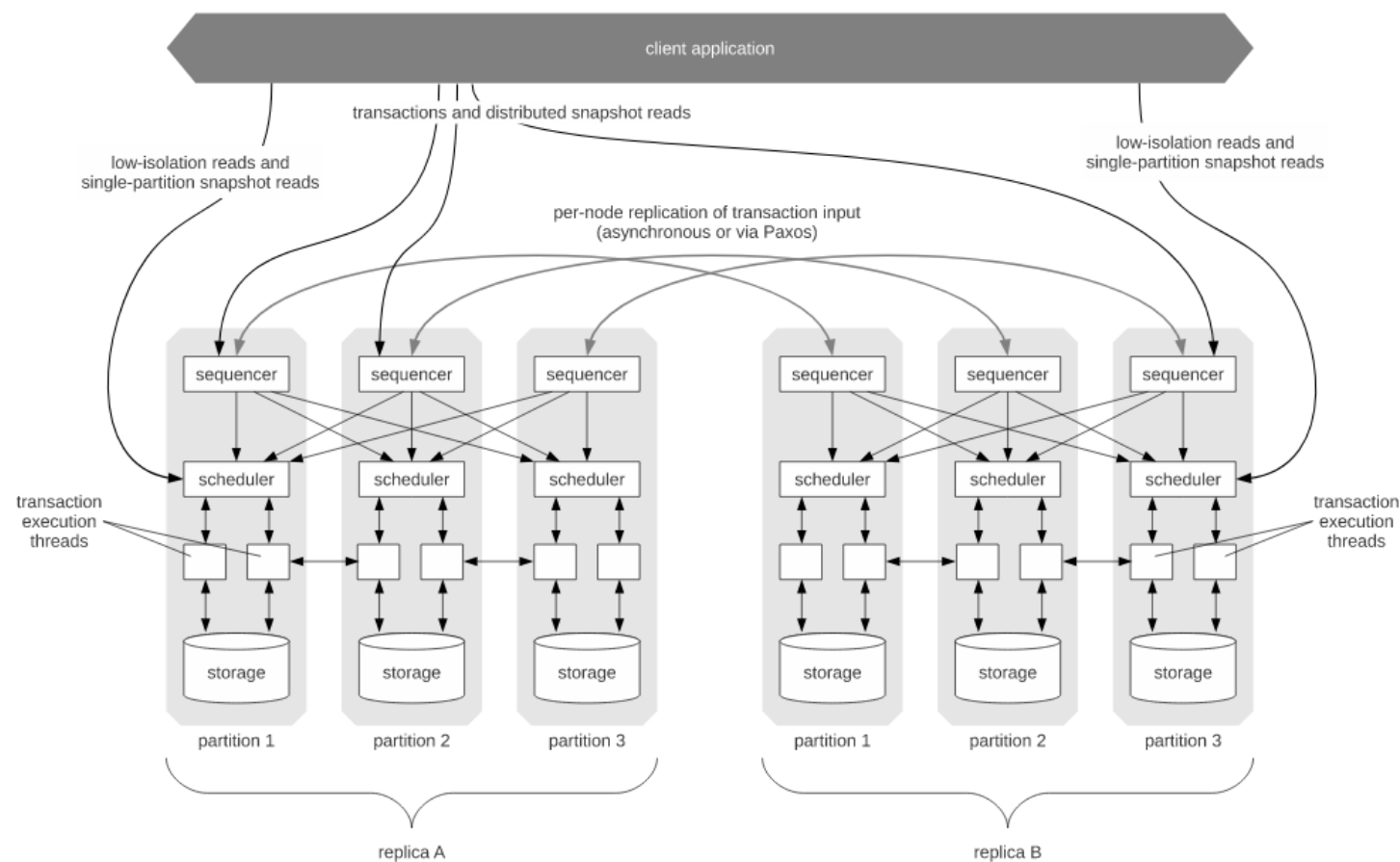


Figure 1: System Architecture of Calvin

- **Sequencer**
每10ms收集来自用户的请求，并将请求发送给对应的scheduler
- **Scheduler**
执行事务，并保证确定性的结果
- **Storage**
一个单机的存储数据库，只需要支持 KV 的 CRUD 接口即可

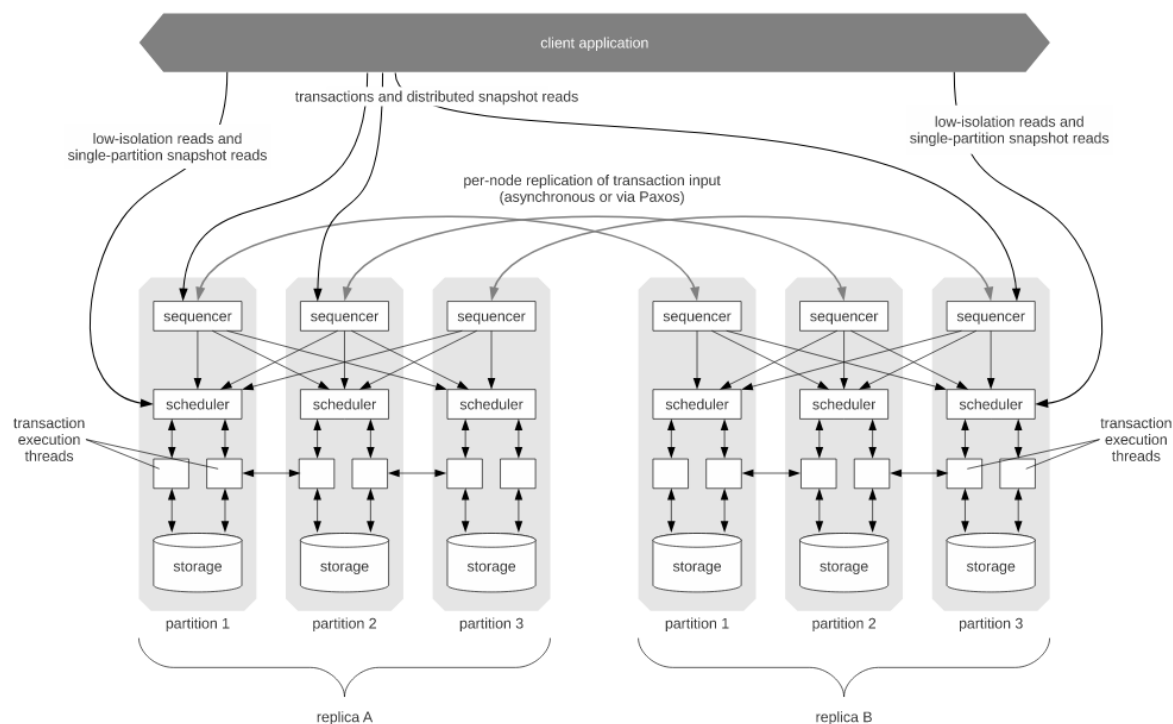
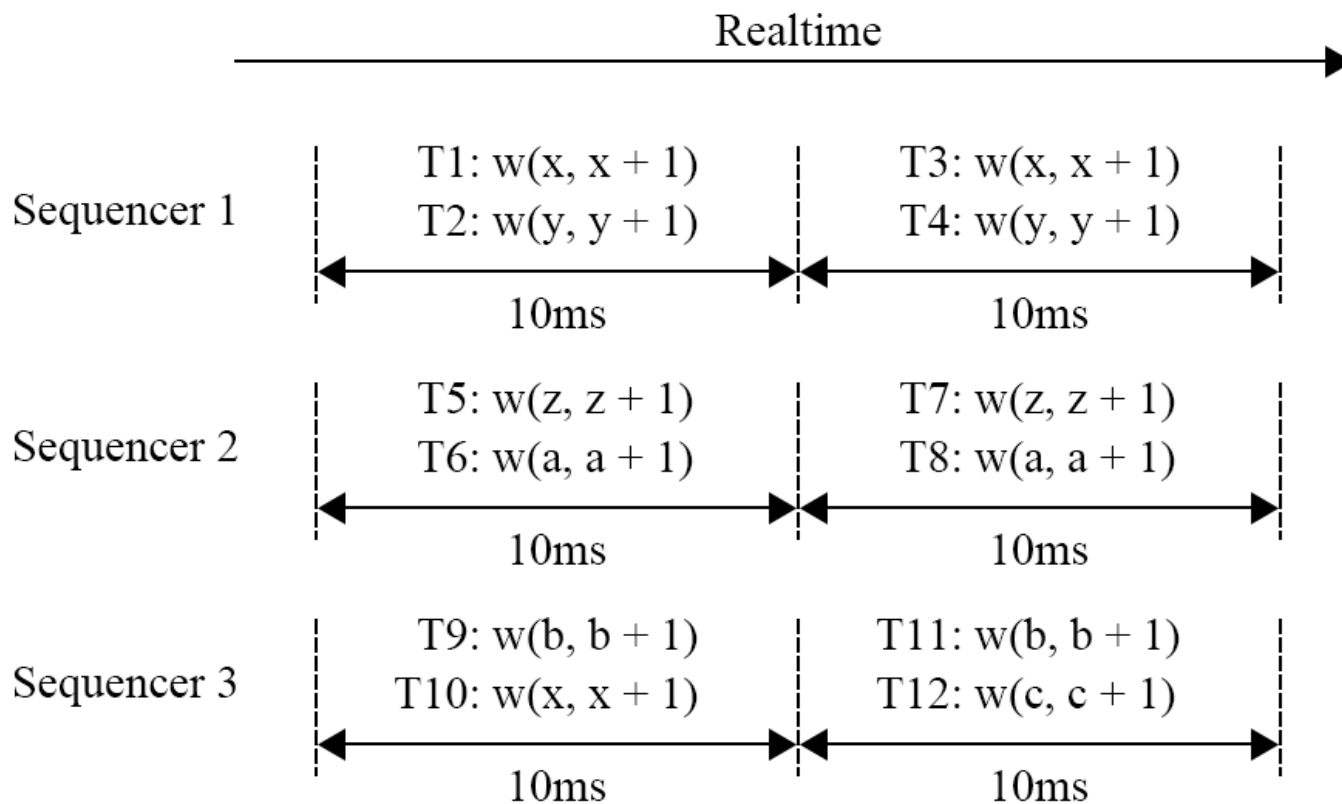


Figure 1: System Architecture of Calvin

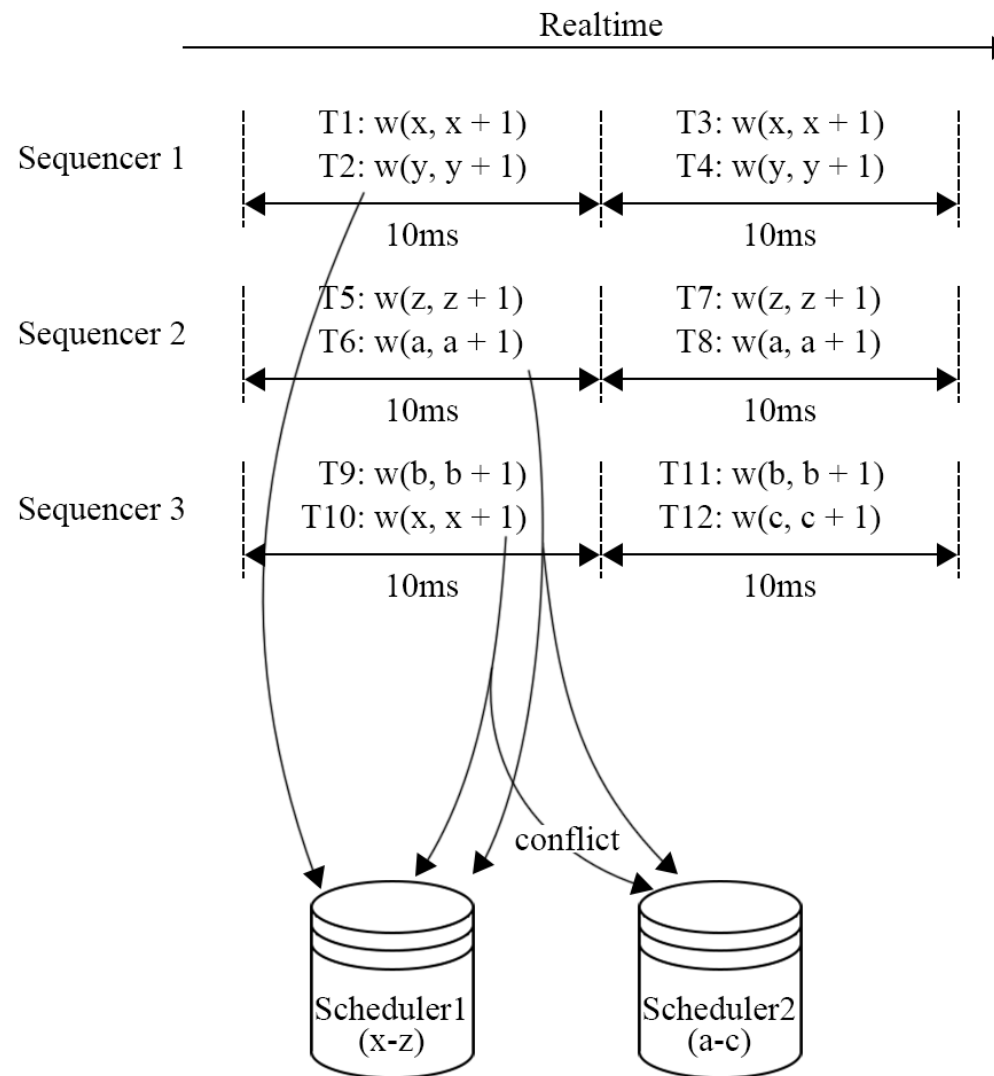
在Sequencer层每10ms。
将客户端发送来的请求打包
成一个batch，在这时
Sequencer的replica之间
会进行副本的复制。



Sequencer在复制结束后将数据发送给Scheduler进行执行。

发送的数据有：

1. Sequencer unique node ID
2. Epoch number
3. 所有与该recipient有关的事务



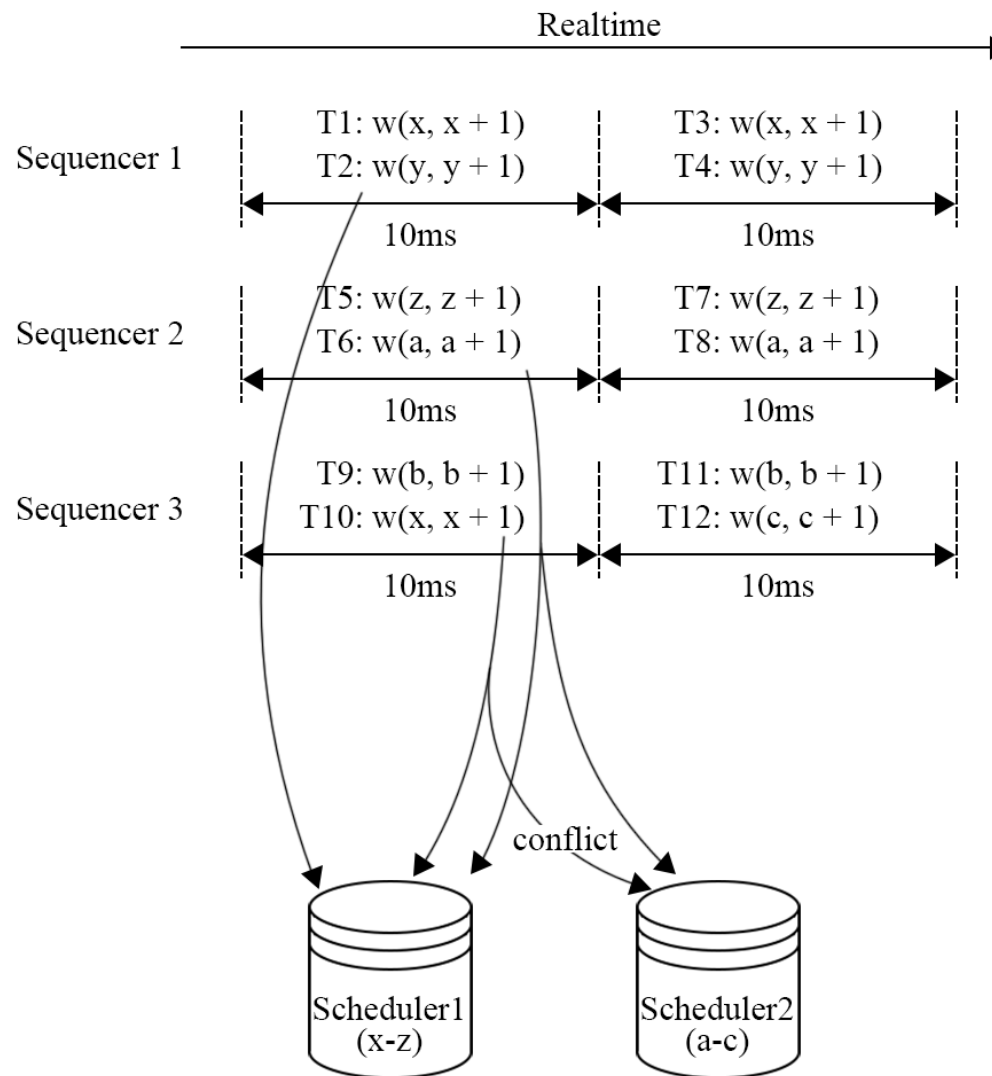
Scheduler排序

Sequencer unique node ID+Epoch number+recipient中顺序

对于 T_1 和 T_{10} 而言, T_1 和 T_{10} 同时需要对于变量 x 进行加锁, 为了避免死锁, Calvin在原有强两阶段锁中加入新的限制。

- 对于任意一个事务对A和B, 同时对于某一个记录R申请排他锁, 如果事务A的事务号小于B, 则A必须先加锁。
- 锁管理器必须保证所有的锁都按照事务的顺序进行赋予锁

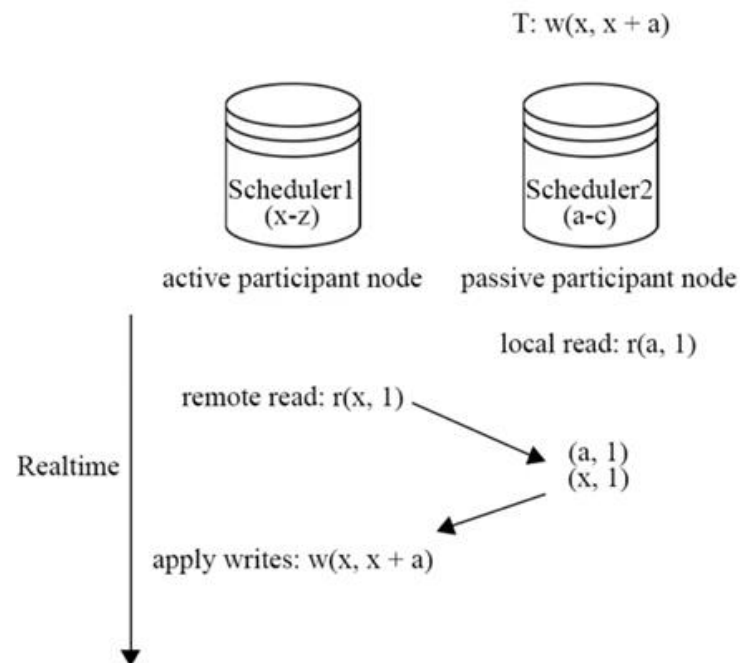
当一个事务获取了所有的锁便可以开始执行





Calvin

1. 读/写集合分析
2. 进行本地读取
3. 服务远程读取
4. 收集远程读取结果
5. 执行事务进行本地写
(在这一阶段各个partition无需交互)





Calvin

为什么是本地写?

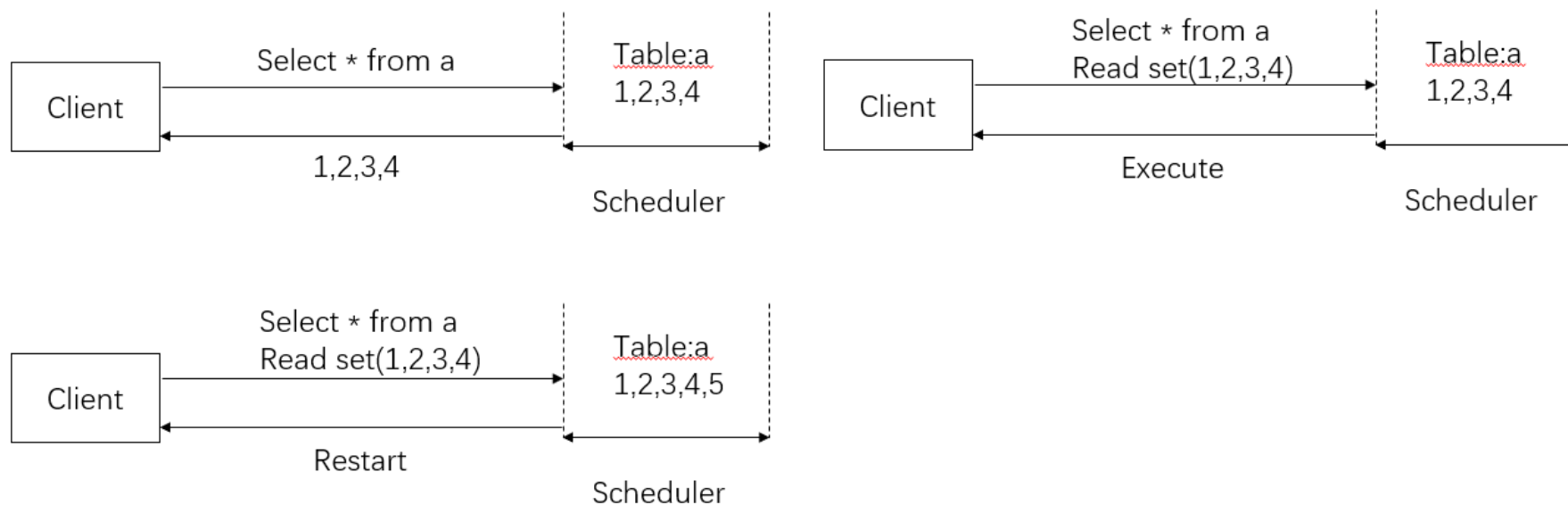
```
//例如这是下订单的存储过程;  
if(inventory num > 0){  
    inventory num = inventory num - 1;  
}else{  
    throw exception;  
}  
insert new order;
```

```
//库存partition执行过程;  
if(inventory num > 0){  
    inventory_num = inventory_num - 1;  
    insert inventory_num  
}else{  
    throw exception;  
}  
insert new order;  
flush inventory_num;
```

```
//订单partition执行过程;  
if(inventory num > 0){  
    inventory_num = inventory_num - 1;  
}else{  
    throw exception;  
}  
insert new order;  
flush order;
```



Calvin



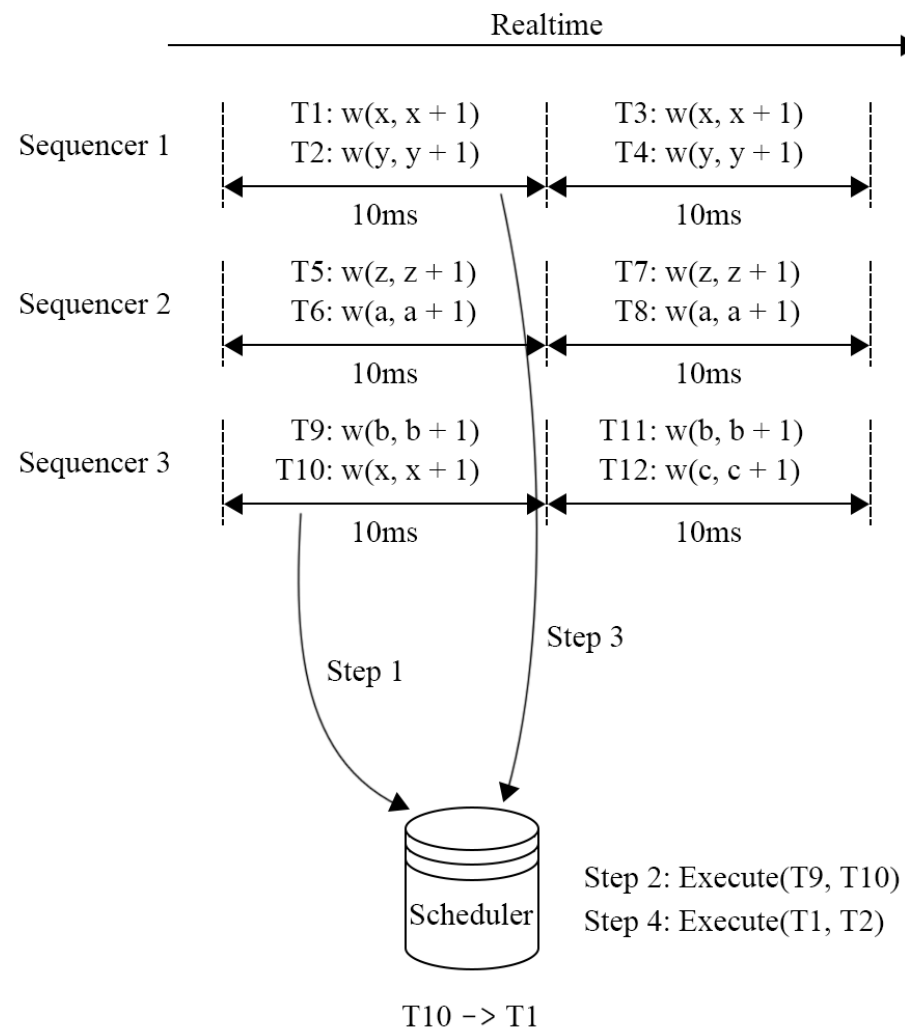
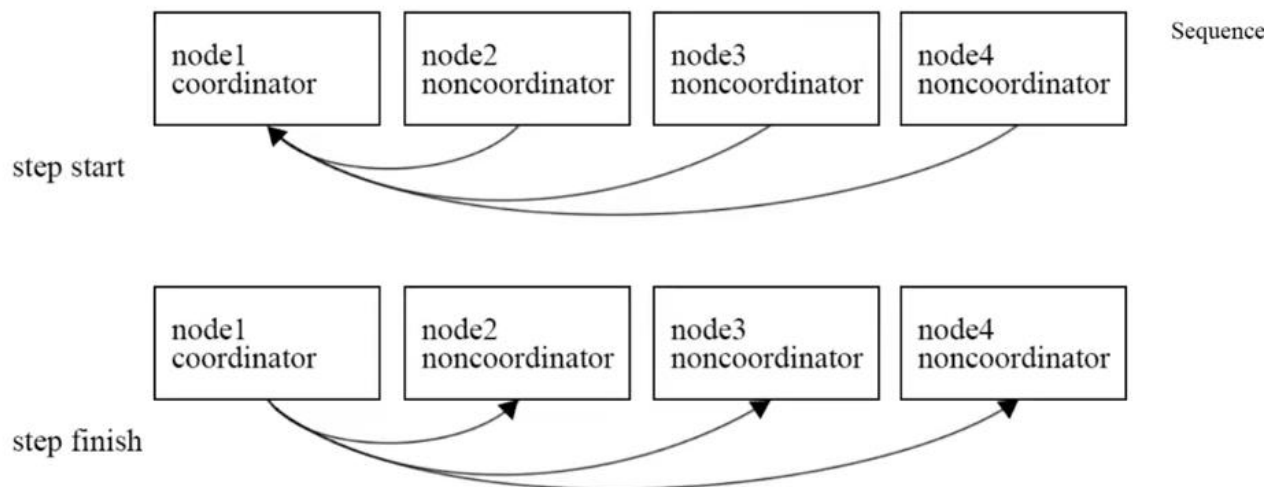
Optimistic Lock Location Prediction (OLLP)

事务被添加sequence,并被执行之前,就使用读请求,来探测这次事务所有的读写记录集,并在执行之前,再次检查,如果读写集变化了,则这个过程要重新执行。



Calvin

如何避免 T_{10} 执行结束, T_1 才到达?
加入coordinator在所有事务未到达scheduler之前不能开始执行。




The deterministic protocol is broken!



Calvin

- 优点：
 - 避免了死锁机制。
 - 避免了写数据在副本间的复制。
 - 实现了SERIALIZABLE的一致性。
- 缺点：
 - coordinator的存在限制了数据库的性能。
 - 在读写集错误以后需要重新进行执行。

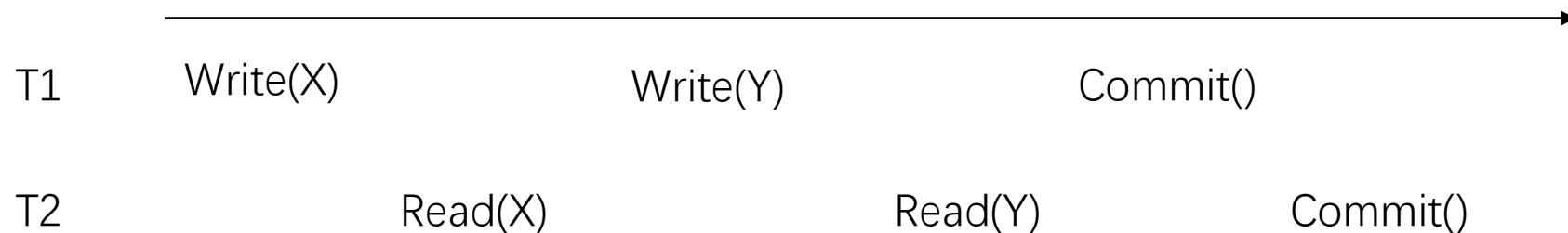
目录

- 
- 确定性数据库
 - Calvin
 - PWV
 - Aria
 - 总结



PWV

PWV



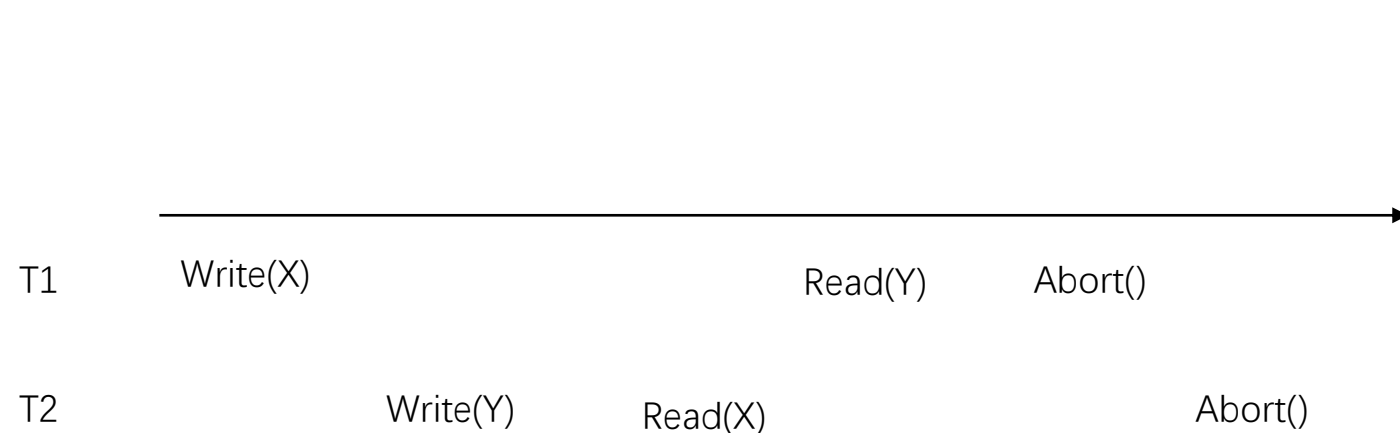
PWV对于事务写入可见性进行了重新的思考，将事务写入可见性进行了提前。

现有数据库一直要等到事务Commit了，事务的写入才可见。



PWV

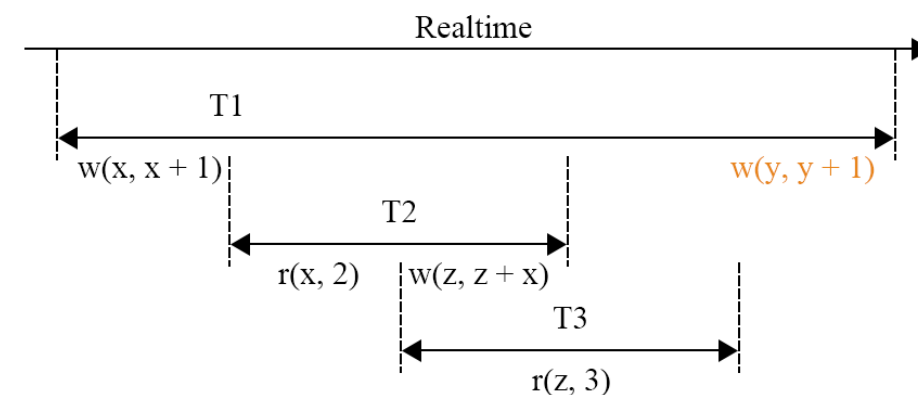
PWV



系统引起的abort
(由于并发控制算法引起的abort)

(x, 1)
(y, 9)
(z, 1)
constraint: value < 10

T1: w(x, x + 1), w(y, y + 1)
T2: r(x, ?), w(z, z + x)
T3: r(z, ?)



逻辑引起的abort
(由于事务逻辑限制引起的abort)

提前写入的可见性，在事务abort时，容易引起级联的回滚



PWV

PWV



在确定性数据库中，存在偏序关系的事务都是串行执行的，也就不存在因为系统原因引起abort。

但是确定性数据库依然存在逻辑引起的abort

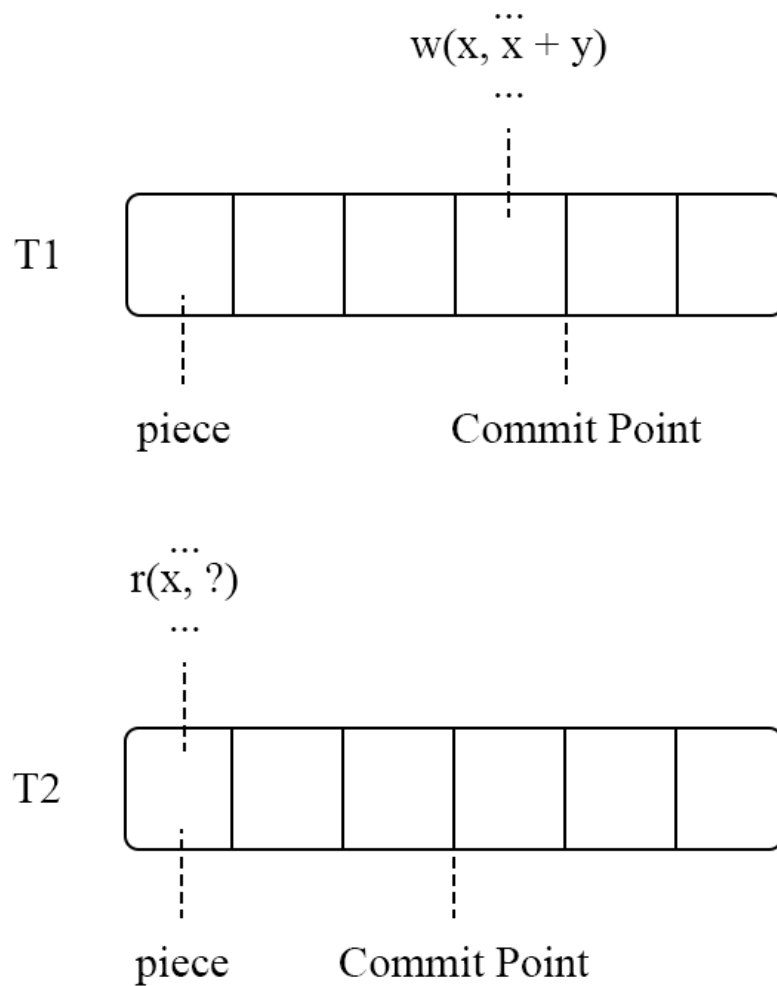


PWV

PWV

为了解决逻辑引起的abort，
PWV引入了提交点的概念，只要事务执行
到达了提交点，我们就认为它写入就可以
被其他事务所读取。

constraint: $x < 10$





PWV

PWV

为了充分发掘事务的并行性，PWV将一个事务分为了多个Piece。

所有在commit point之前的Piece被称为abortable，在commit point之后的Piece称为non-abortable

如果一个Piece的数据来自于其他Piece的执行结果那么该Piece就与其他Piece之间存在data dependencies

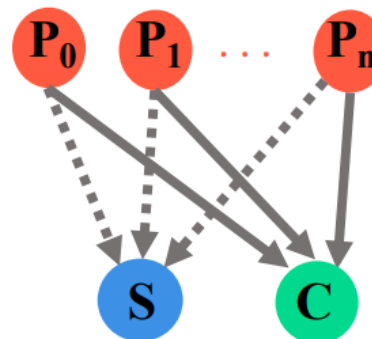
如果一个Piece在提交点之后执行，且有写入操作，则该Piece与所有abortable的Piece存在commit dependencies

```
1 price = 0
2 for p_id in p_id_list:
3     prod = DB.write_ref(p_id, "products")
4     if prod.count == 0:
5         ABORT()
6     else:
7         prod.count -= 1
8         price += prod.price
9
10 stats = DB.write_ref("statistics")
11 stats.num_purchases += p_id_list.size()
12
13 cust = DB.write_ref(c_id, "customer")
14 cust.bill += price
```

P

S

C





PWV

PWV

RVP counter:

一个piece的RVP counter的值parent piece的数目相同。每有一个parent piece完成, RVP counter减一。

Commit RVP:

一个事务的Commit RVP与其abortable的事务数目相同, 每当有一个abortable事务完成, Commit RVP减一, 当RVP Counter为0时表示事务可以提交。

```
1 price = 0
2 for p_id in p_id_list:
3     prod = DB.write_ref(p_id, "products")
4     if prod.count == 0:
5         ABORT()
6     else:
7         prod.count -= 1
8         price += prod.price

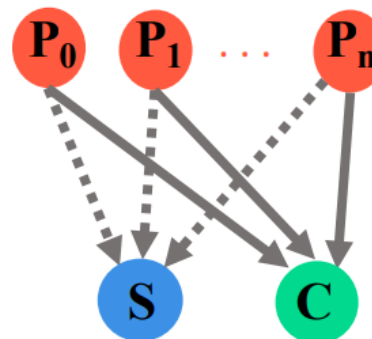
9 stats = DB.write_ref("statistics")
10 stats.num_purchases += p_id_list.size()

11 cust = DB.write_ref(c_id, "customer")
12 cust.bill += price
```

P

S

C





PWV

PWV

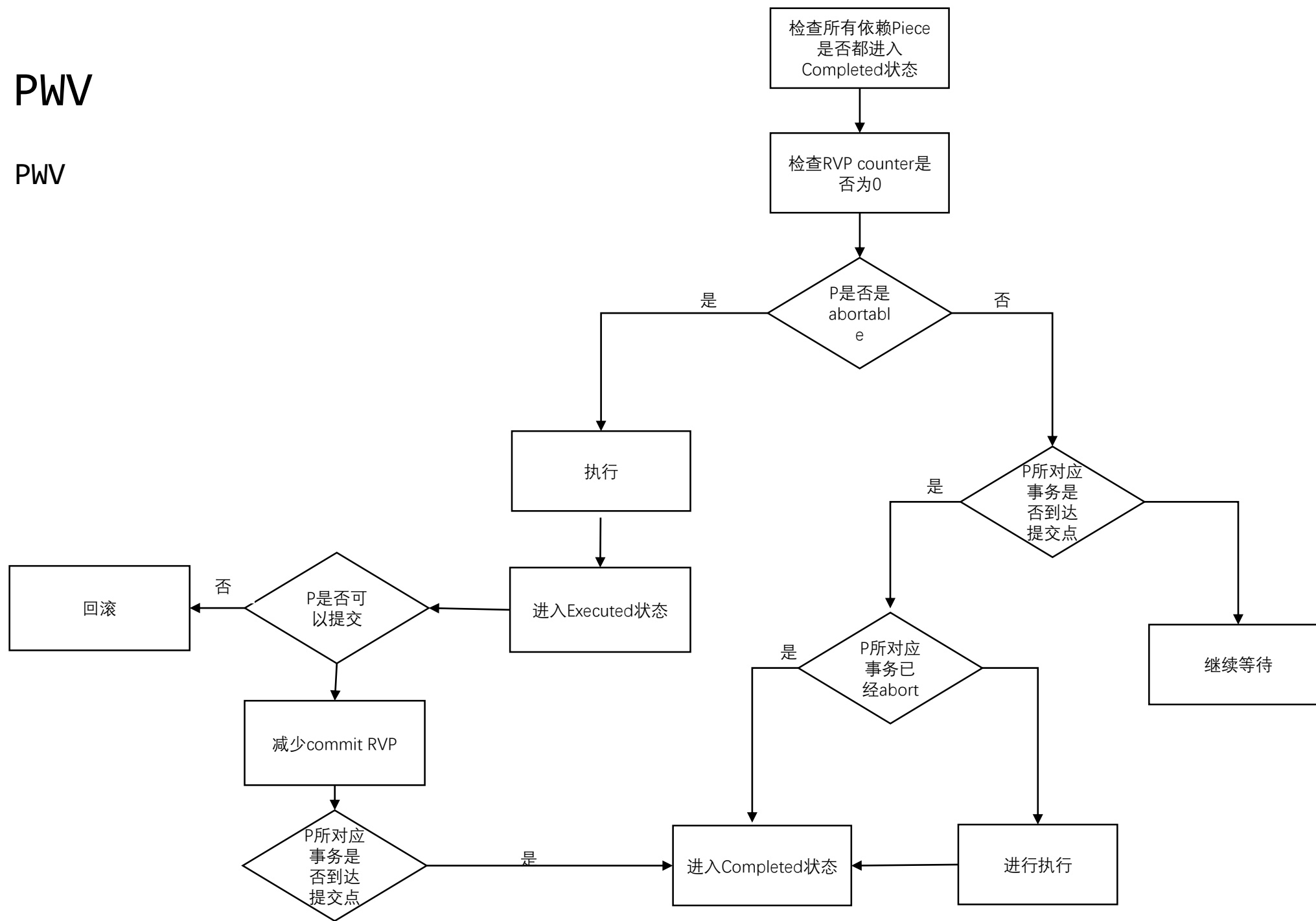
假设 $T_1 < T_2$; $P_1 \in T_1; P_2 \in T_2$

- 限制1: 如果 P_1 和 P_2 之间存在着写读, 写写冲突
 - 如果 P_1 是 abortable, 那么所有属于 T_1 的 abortable pieces 都必须在 P_2 之前执行。
 - 如果 P_1 non - abortable, 那么因为 P_1 一定在 commit point 之后, 所以 P_1 必须在 P_2 之前执行。
- 限制2: 如果 P_1 和 P_2 之间存在着读写冲突, 那么 P_1 必须在 P_2 之前执行。



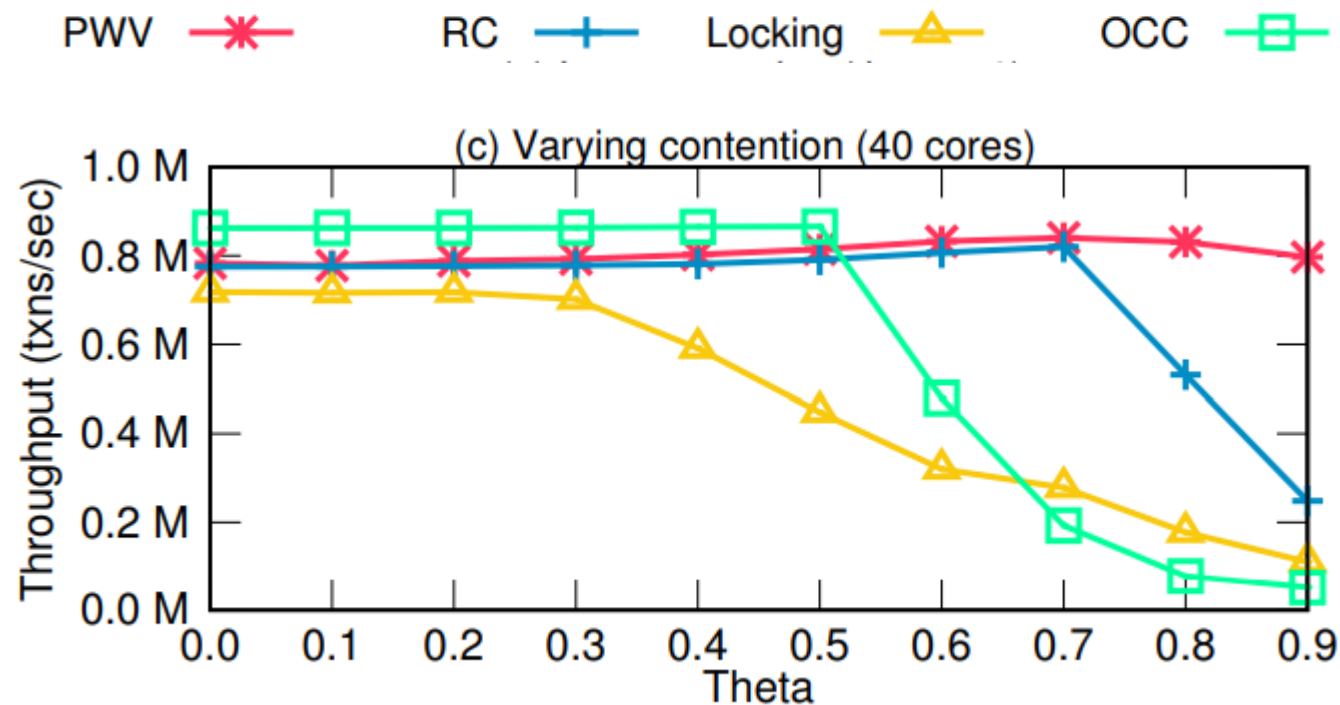
PWV

PWV






在确定性事务的大前提下：
PWV都提升了事务执行的效率，将事务写的可见性提前，但是需要获得事务的全局状态，所以只适合单机数据库。

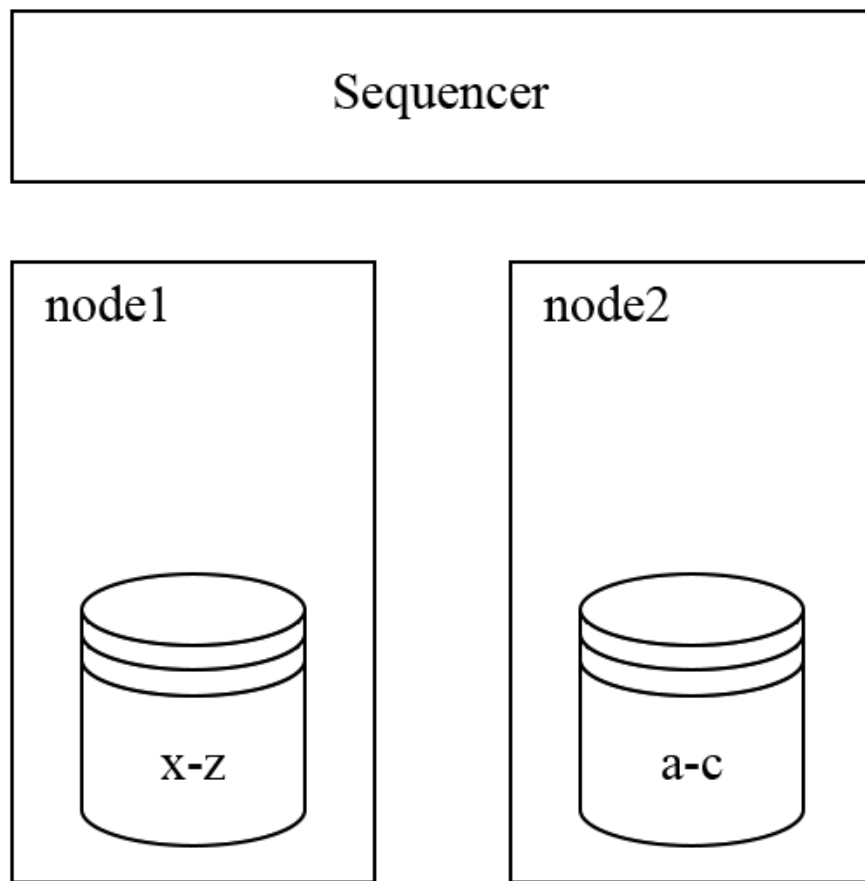


目录

- 
- 确定性数据库
 - Calvin
 - PWV
 - **Aria**
 - 总结



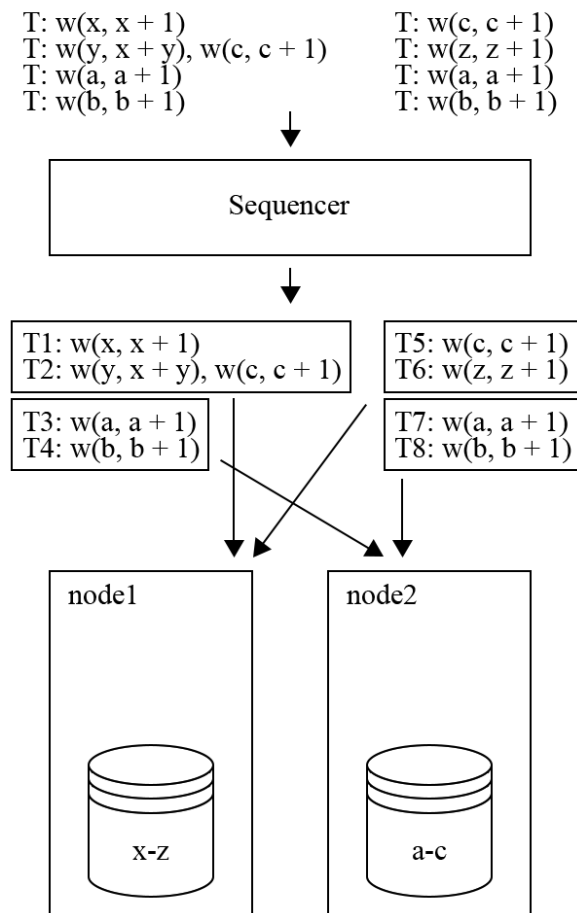
Aria与Calvin一样是分布式数据库，但是Aria的并发控制是在Sequencer之下而不是在Sequencer之上，且Aria无需提前确定读写集。

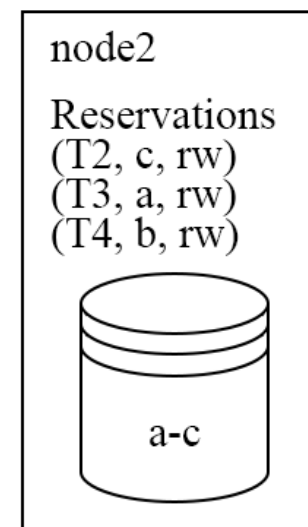
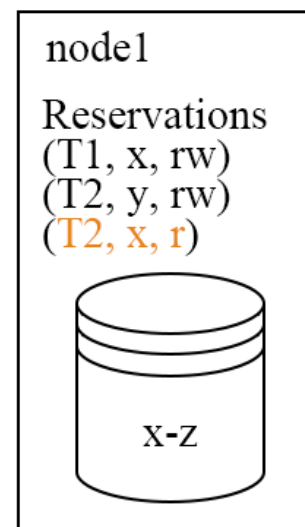
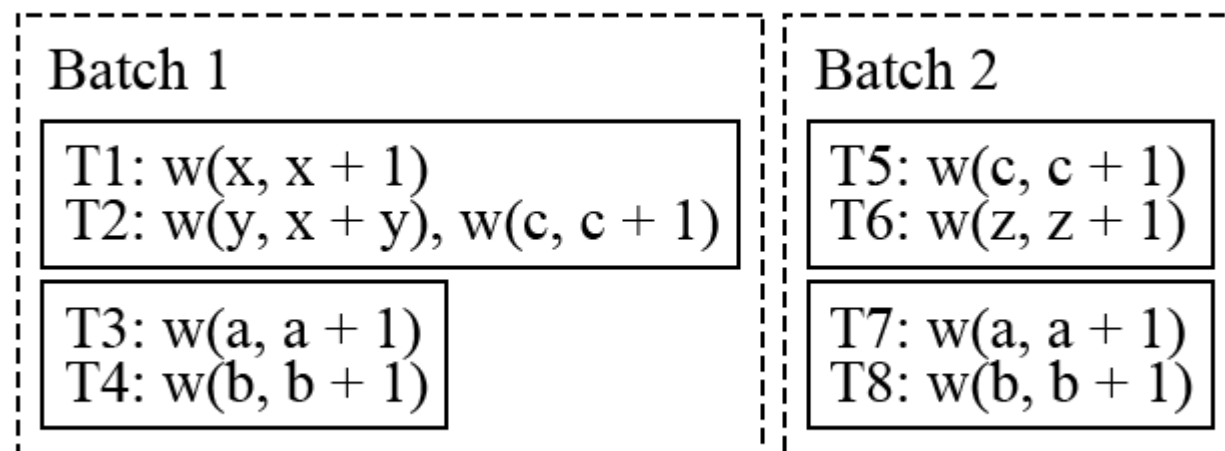


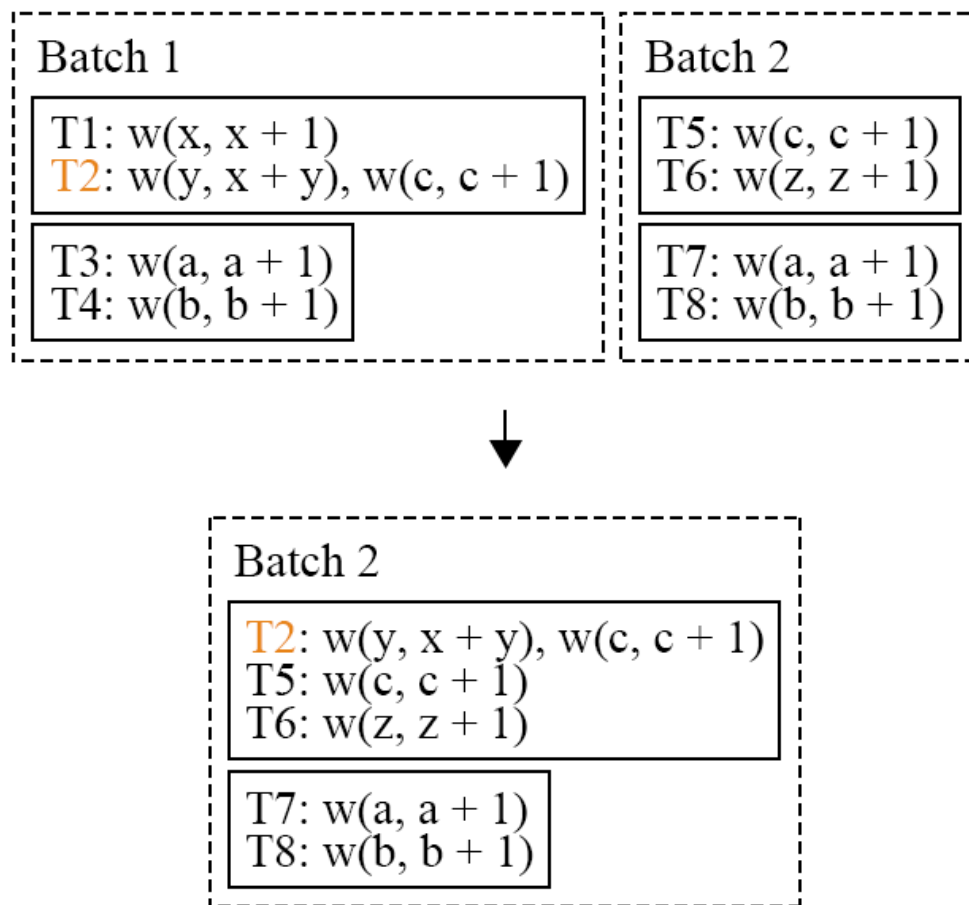
Sequencer Level Replication

Storage Level Replication

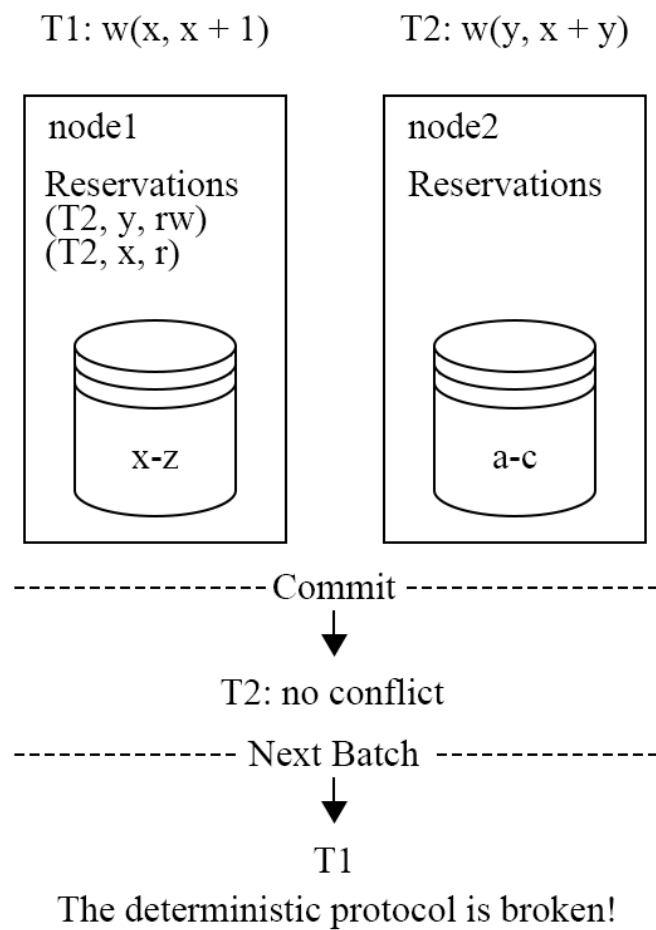
- 一个 sequence 层为事务分配全局递增的 id;
- 将输入的事务持久化;
- 执行事务, 将 mutation 存在执行节点的内存中;
- 对持有这个 key 的节点进行 reservation;
- 在 commit 阶段进行冲突检测, 是否允许 commit, 没有发生冲突的事务则返回执行成功;
- 异步的写入数据



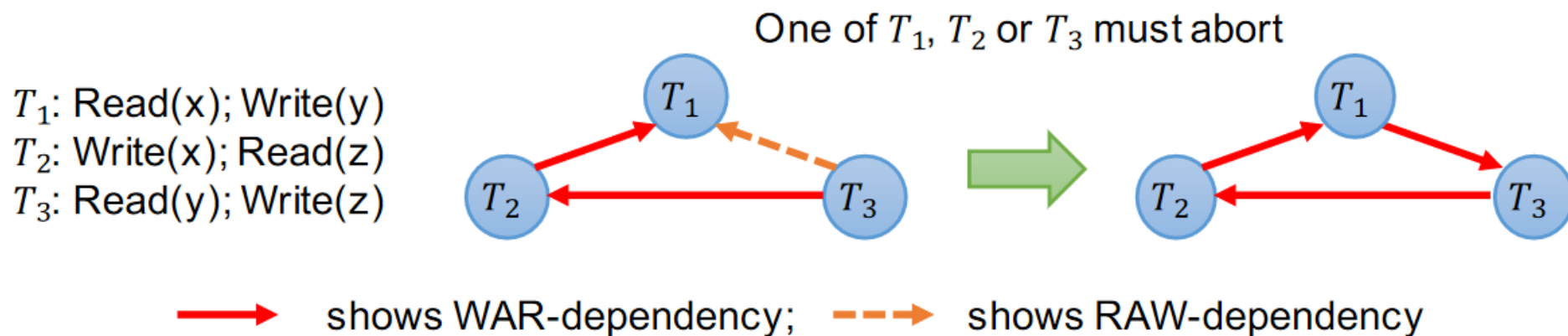
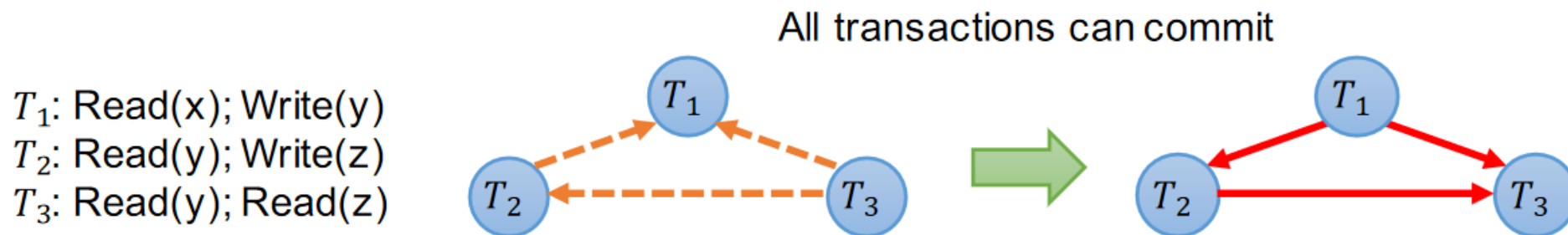




由于T2是第二个batch中享有最高优先级所以不会无限推迟



与Calvin一样, Aria也需要加入coordinator




进行了重排序, 减少事务冲突



优点

- 在于执行和 reservation 的策略拥有更高的并行度,
- 不需要额外的 OLLP 策略进行试探性读

目录

- 
- 确定性数据库
 - Calvin
 - BOHM&PWV
 - Aria
 - 总结



总结

确定性数据库相比与传统数据而言

- 无需存储多版本的数据
- 减少了死锁的问题
- 对长事务支持较差
- 每次一个batch的机制，导致必须要有全局的coordinator存在。

Barrier
Step Start

Barrier
Step Finish

