事务并发控制算法(二)

胡卉芪
华东师范大学
数据科学与工程学院
hqhu@dase.ecnu.edu.cn

Revisit并发控制算法的分类

- 悲观/乐观
  - 悲观： 2PL
  - 乐观：乐观并发控制(OCC)
- 基于锁/基于时间戳
  - 基于锁： 2PL
  - 基于时间戳： TO, OCC
- 并发控制的关键
  - 如何检测冲突
  - 如何处理冲突

Optimistic Concurrency Control (OCC)

乐观并发控制(OCC)

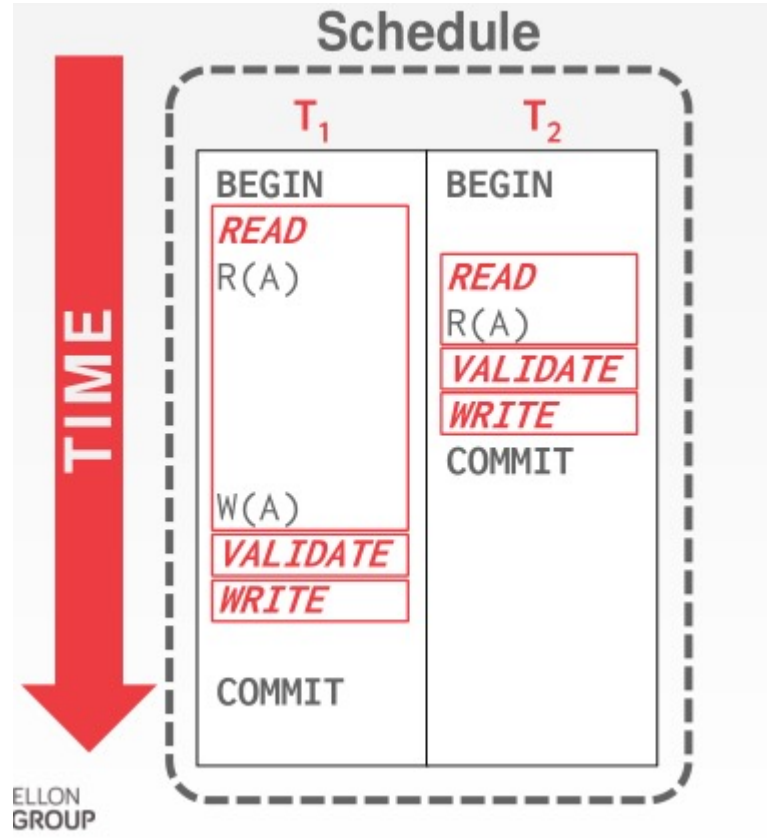
- 每个事务产生一个私有空间
  - 维护事务中的读写集合
- 事务提交时，验证读写集合是否与其他事务冲突，如果没有将写集合应用于数据
- 另外， OCC为每个事务分配了一个时间戳，方便验证
- 我们参照OCC的原文
  - On Optimistic Methods for Concurrency Control

OCC的三个阶段

- 读阶段
  - 执行事务，并在事务的私有空间生成事务的读写集合
- 验证阶段
  - 提交之前，通过读写集合验证是否有冲突
- 写阶段
  - 将写集合应用到数据，commit or abort

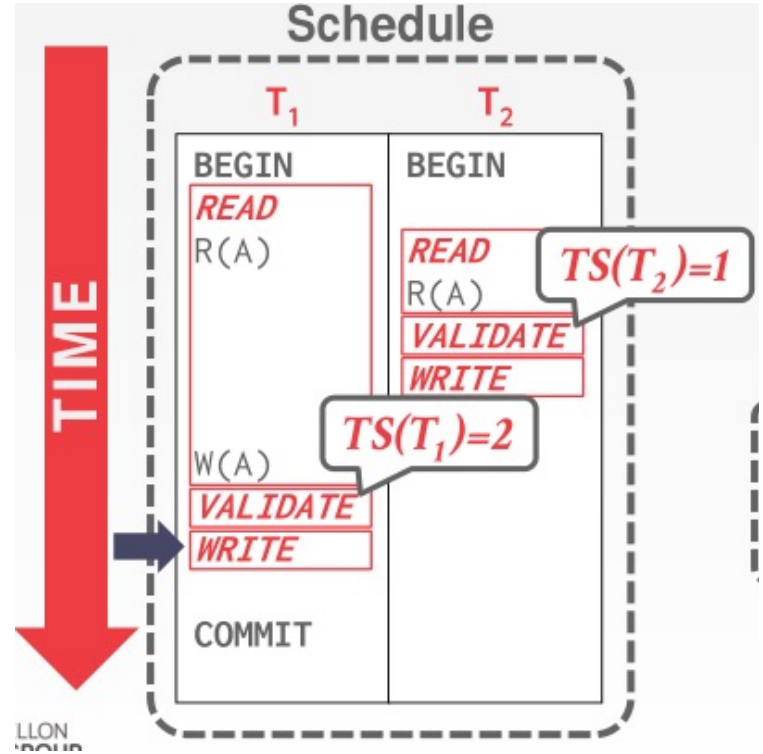
OCC例子

- 读阶段生成集合
  - T1.ReadSet={A}，T1.WriteSet={A}
  - T2.ReadSet={A}



OCC例子

- 算法中验证成功的事务分配事务时间戳
  - 怎么判断验证成功？

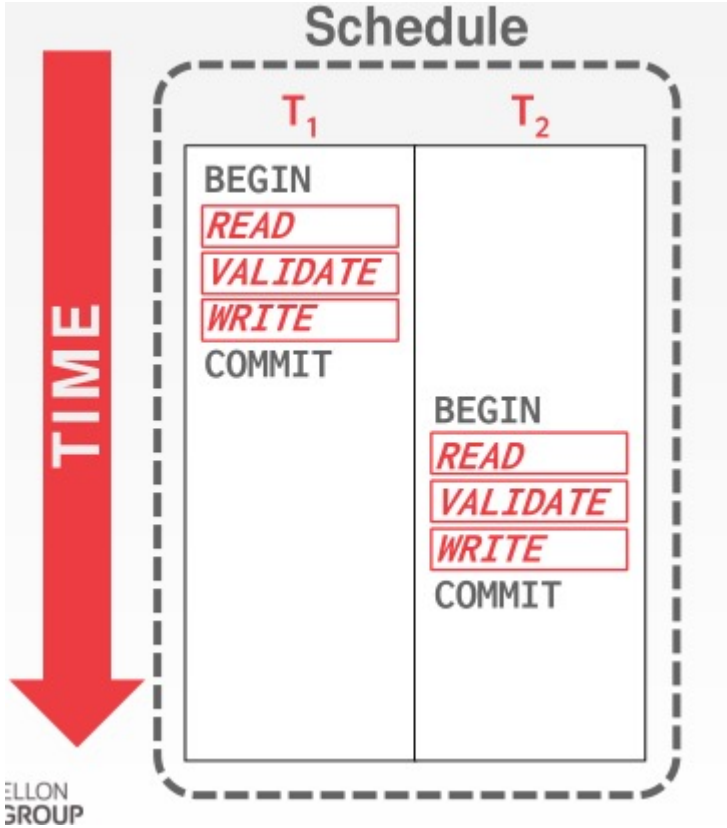


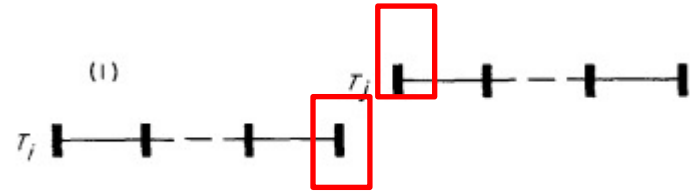
验证阶段

- 保证可串行化
  - Ti 需要和其他事务是否有读写冲突与写写冲突
- 如果Ti <Tj 那么下面三种情况Ti是可以提交的
- 要注意看的是三个时间点
  - 事务开始（读阶段）时间
  - 事务结束(写阶段)时间
  - 验证阶段开始时间

Case 1

- Ti 在 Tj事务开始前完成写阶段



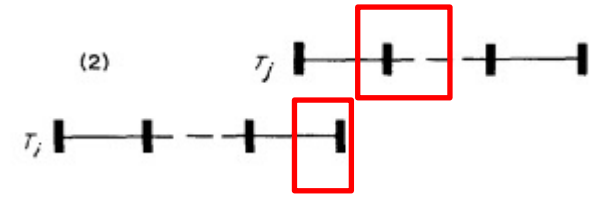


Case 2

- Ti 在 Tj的验证阶段开始前完成写阶段， 并且

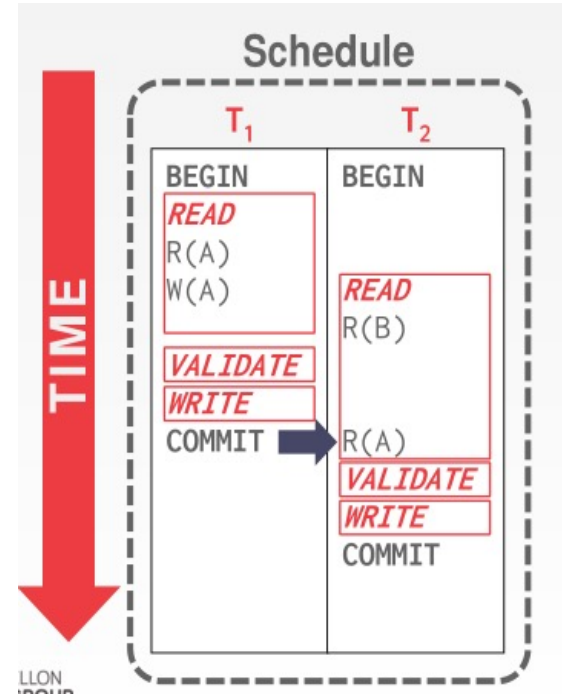
$$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$$

- 如果交集不为空，那么T2没有读到T1的写



Tj.readset={A,B}
Ti.writeset={A}

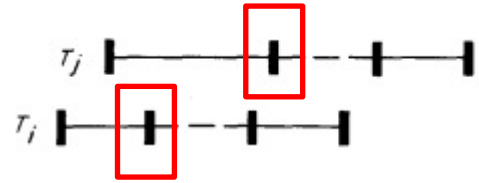
- Tj.readset和Ti的writeset有交集
  Abort T2

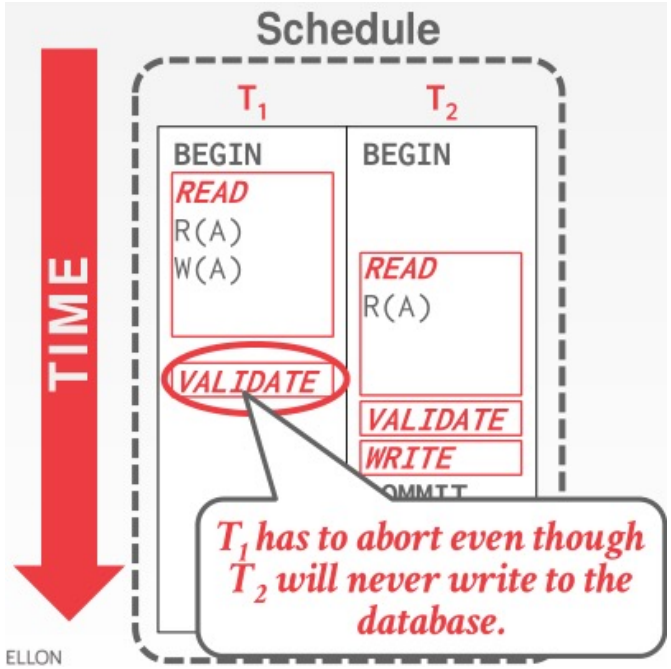


Case 3

- Ti 在 Tj的读阶段结束前开始验证阶段， 并且

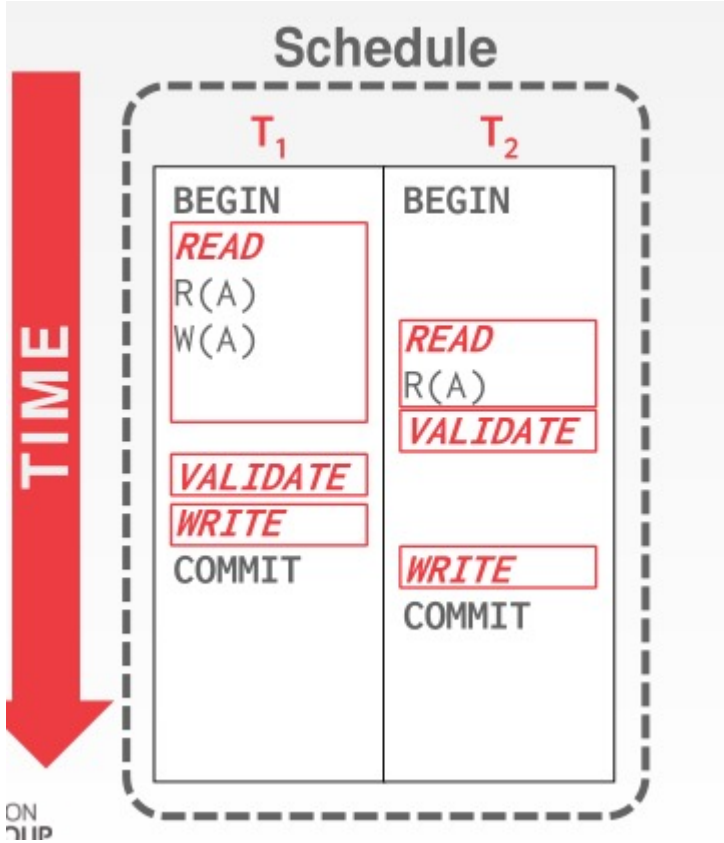
$$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$$
$$WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$$





Case 3

- 这种情况如何理解？
  - T1或者T2会abort么？



OCC基本实现

- 验证阶段+写阶段组成临界区
  - 所有事务的验证+写阶段互斥
  - Case3不会实际出现
    - 所以有些描述中只关心Case1和Case2
  - 总结Case1与Case2
    - **finish**(T_i) < **start**(T_j)
    - **start**(T_j) < **finish**(T_i)<**validation**(T_j) 对Tj的读集合，验证所有满足上面范围的Ti的写集合是否有交集

OCC验证与写阶段算法

```
tbegin = (
   create set := empty;
   read set := empty;
   write set := empty;
   delete set := empty;
   start tn := tnc)

tend = (
   ⟨finish tn := tnc;
    valid := true;
    for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
          then valid := false;
    if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
    if valid
        then (cleanup)
        else (backup)).
```

tnc是一个单调递增公共时间戳，事务一开始时就获取

临界区

对满足Case2的事务t进行验证

验证成功后产生事务时间戳

OCC验证与写阶段算法

- 思考算法是如何巧妙利用时间戳实现Case1和Case2的？
  - Case1， 事务开始获得当前时间戳 start tn，在比较时从start tn+1开始的事务去验证，因为start tn之前的事务已经完成。
  - Case2， 事务再次获取当前时间戳 finish tn, 因此所有在start tn与finish tn之间的事务都是满足case2的。

多版本并发控制

多版本

- 对某一个数据项，数据库中存储其多个不同时间点上的状态(一次修改看做一次状态)
  - 每个写操作，创造一个新的版本
  - 每个读操作，通常在事务开始时获取一个版本，读取小于等于该版本最新数据
- 读写分离
  - 读不阻塞写
  - 写不阻塞读

多版本

- 对某一个数据项，数据库中存储其多个不同时间点上的状态(一次修改看做一次状态)
  - 每个写操作，创造一个新的版本
  - 每个读操作，通常在事务开始时获取一个版本，读取小于等于该版本最新数据
- 读写分离
  - 读不阻塞写
  - 写不阻塞读

SnapShot Isolation

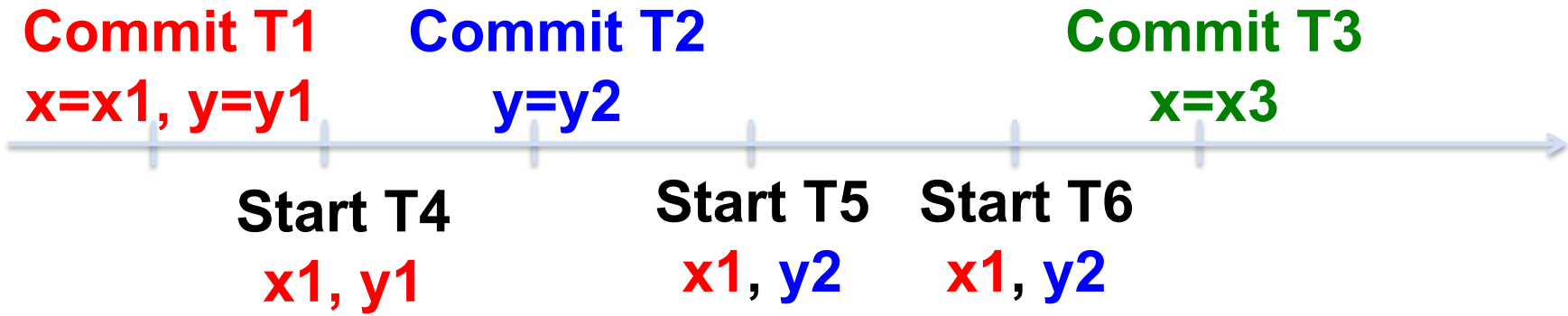
- 事务开始:
  - Get snapshot
- 事务执行:
  - Reads from snapshot
  - Writes to private workspace
- 事务提交:
  - Check for write-write conflicts
    - 如果有abort到只有一个事务提交
  - Install updates

SI的更多理解

- SI既是实现也是一种隔离级别
- 事务之间的写有一个统一顺序
  - 事务之间没有写写冲突，为什么只有一个写可以成功？
    - 假设银行取钱的场景，当我有100块钱时我可以取，但如果我的家人已经取了100，我再去取50块就取不出了。
- 事务读一个consistent的snapshot。

举例：SI的版本变化

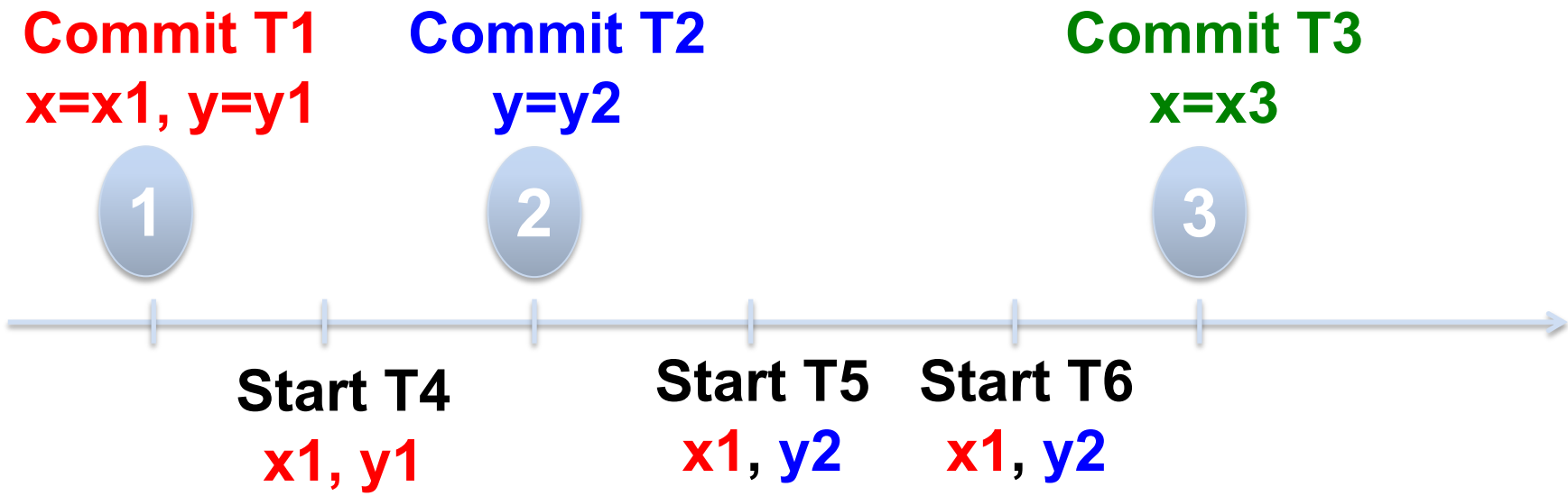
举例：SI的版本变化



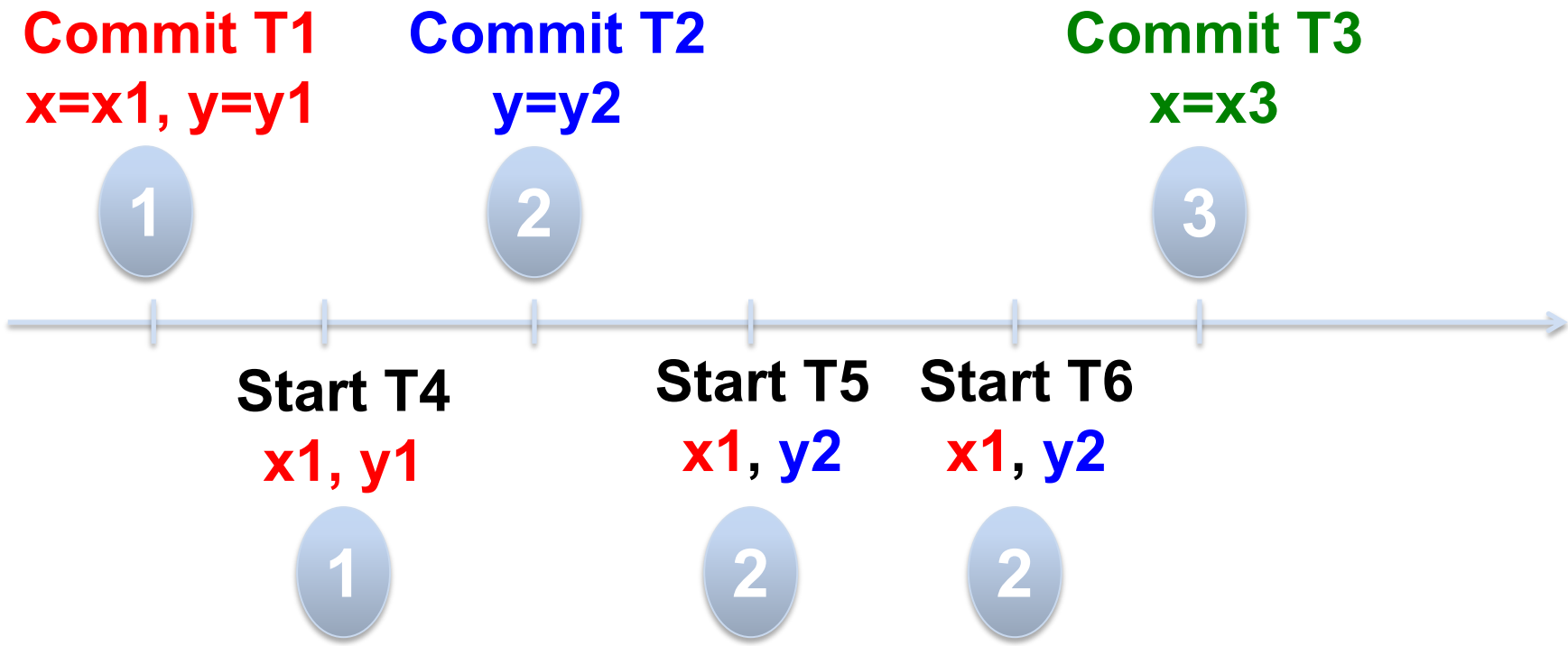
写操作：
　生成一个时间戳

读操作：
　事务开始时安排一个时间戳

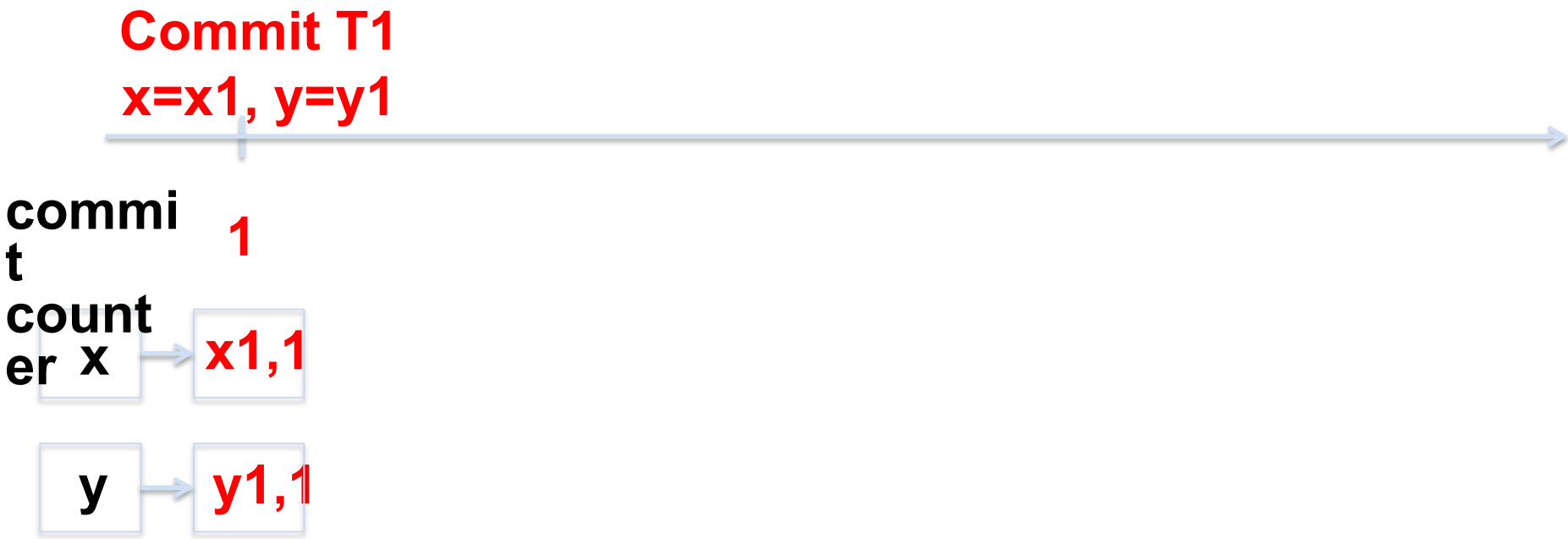
举例：SI的版本变化


Commit T1
x=x1, y=y1
Commit T2
y=y2
Commit T3
x=x3
1
2
3
Start T4
x1, y1
Start T5
x1, y2
Start T6
x1, y2

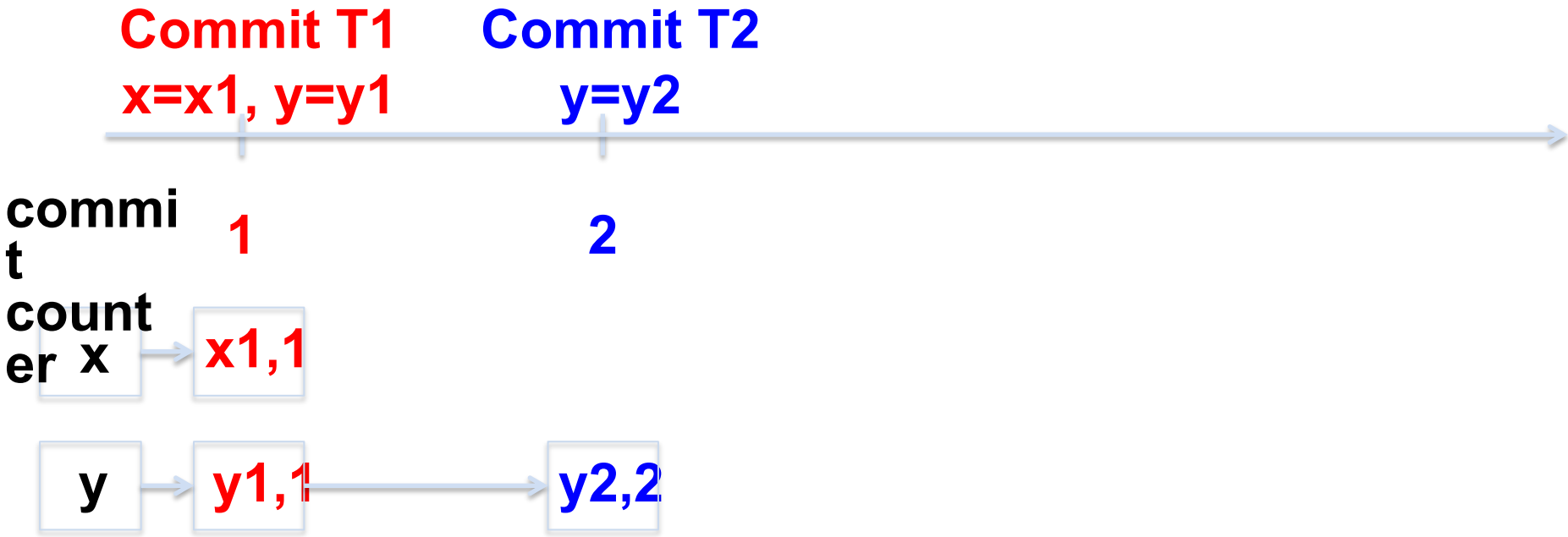
举例：SI的版本变化


Commit T1
x=x1, y=y1
Commit T2
y=y2
Commit T3
x=x3
1
2
3
Start T4
x1, y1
Start T5
x1, y2
Start T6
x1, y2
1
2
2

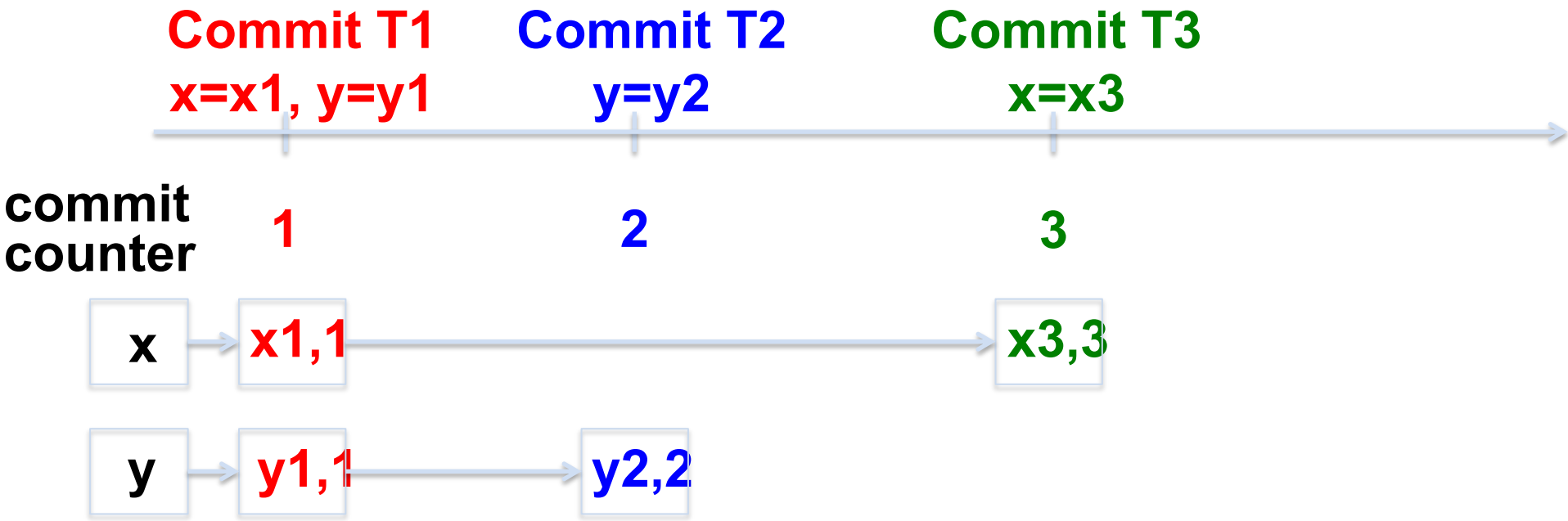
举例：SI的版本变化



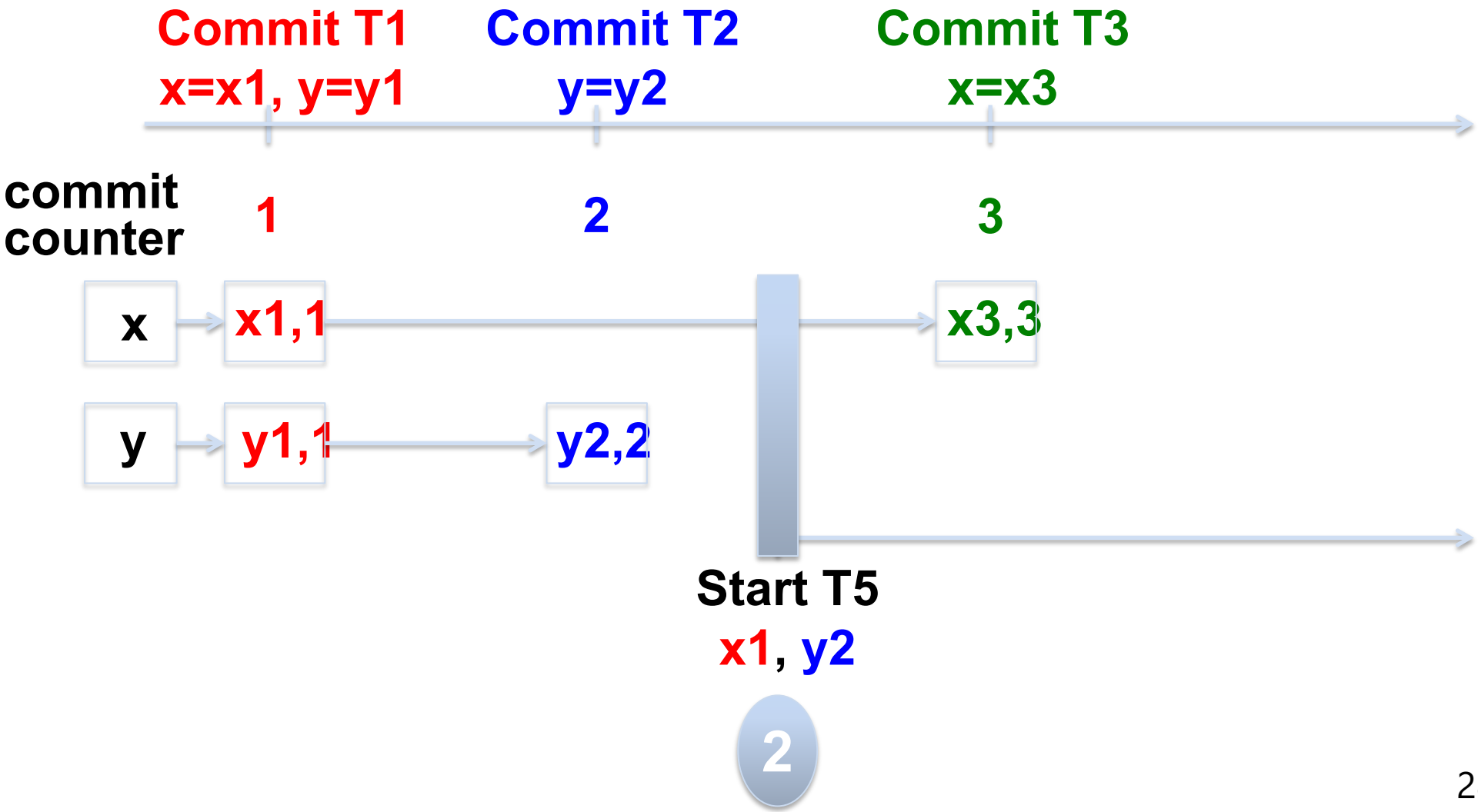
举例：SI的版本变化


Commit T1
x=x1, y=y1
Commit T2
y=y2
commit counter
1
2
x
x1,1
y
y1,1
y2,2

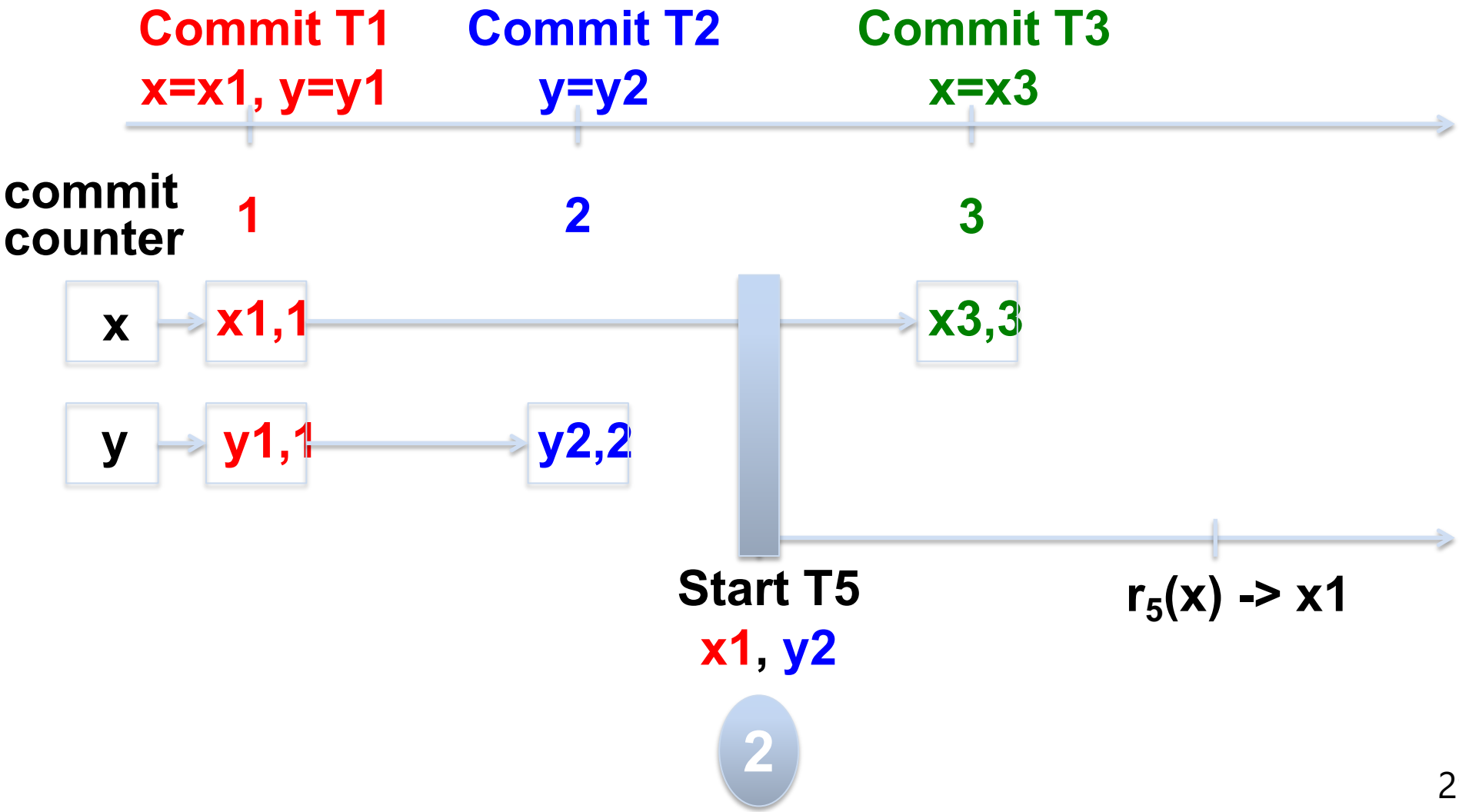
举例：SI的版本变化


Commit T1
x=x1, y=y1
Commit T2
y=y2
Commit T3
x=x3
commit counter
1
2
3
x
x1,1
x3,3
y
y1,1
y2,2

举例：SI的版本变化


Commit T1
x=x1, y=y1
Commit T2
y=y2
Commit T3
x=x3
commit counter
1
2
3
x
x1,1
x3,3
y
y1,1
y2,2
Start T5
x1, y2
2

举例：SI的版本变化


Commit T1
x=x1, y=y1
Commit T2
y=y2
Commit T3
x=x3
commit counter
1
2
3
x
x1,1
x3,3
y
y1,1
y2,2
Start T5
x1, y2
r5(x) -> x1
2

多版本其他

- 多版本做到可串行化
  - 结合OCC， 2PL： High-Performance Concurrency Control Mechanisms for Main-Memory Databases
  - SSI：serializable isolation for snapshot databases
- 多版本的实现形式
  - An Empirical Evaluation of In-Memory Multi-Version Concurrency Contro
- 旧版本回收
  - 多版本方式中需要去回收一些不再使用的旧的数据版本，释放存储空间， 如何找到旧的数据版本？
    - 基于后台寻找
    - 基于事务执行

浅谈分布式事务中并发控制问题

理解并发控制问题

- 假设100，101事务都要写两个数据500和1500。如果将他们发到两个partition上，每个partition分别使用单点的并发控制协议，会有什么问题？
  - 如果是并发控制是S2PL呢？