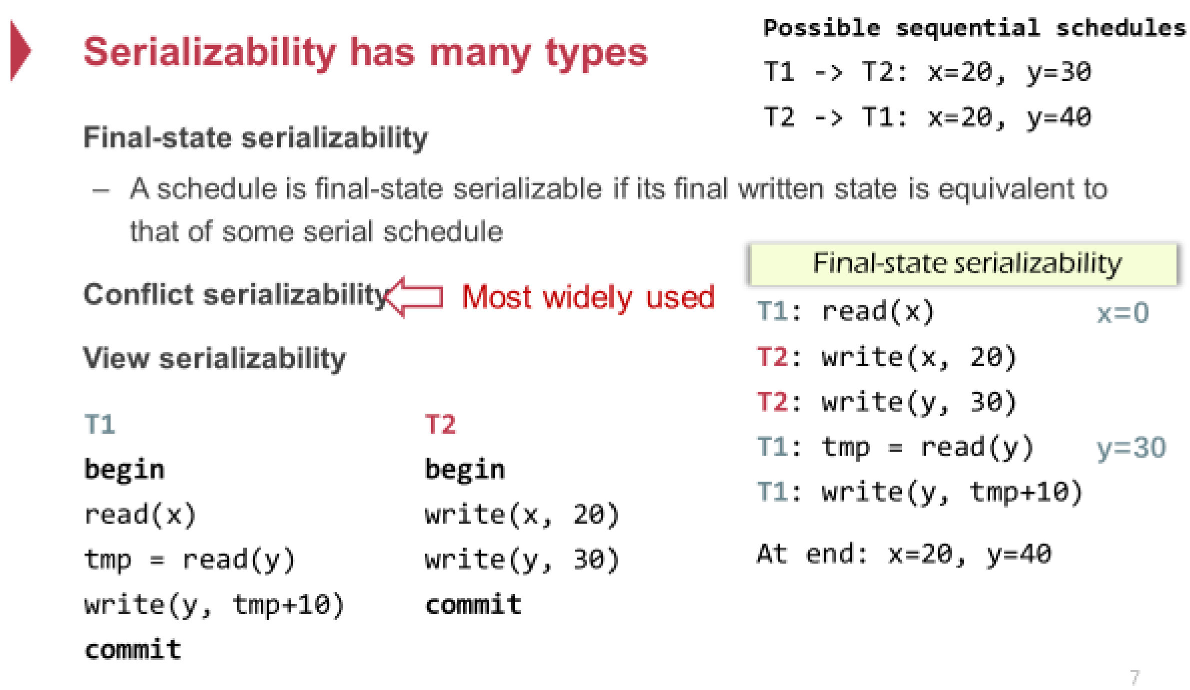
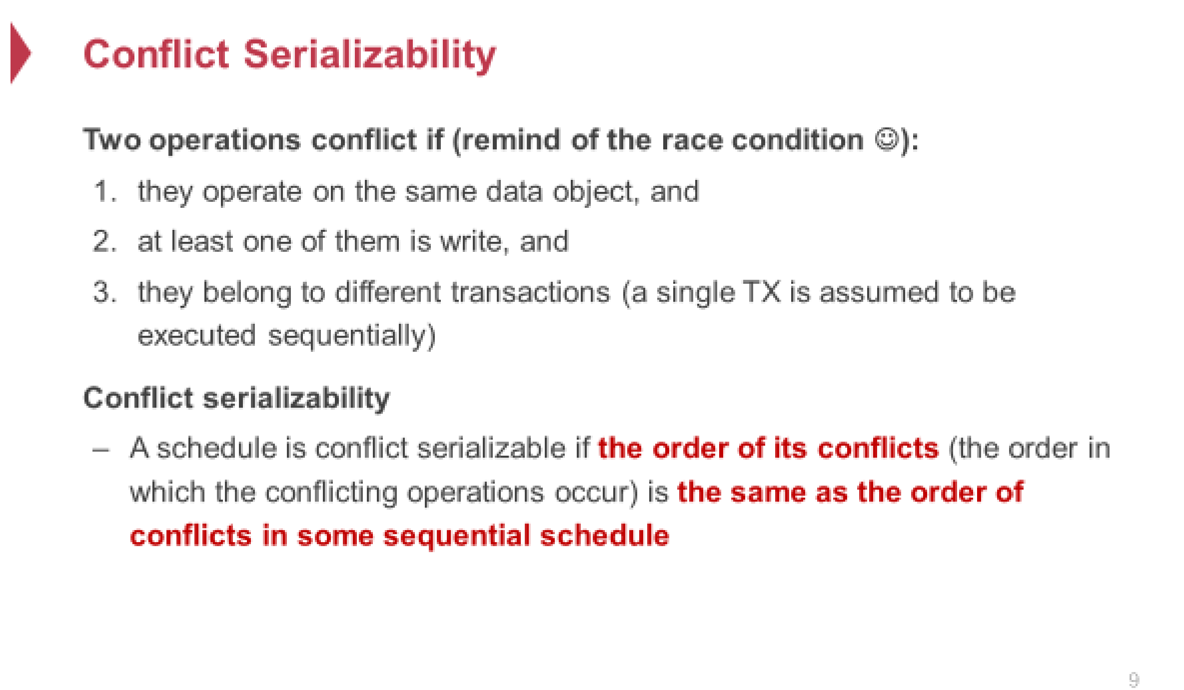
Serializability, OCC&Transaction

Serializability

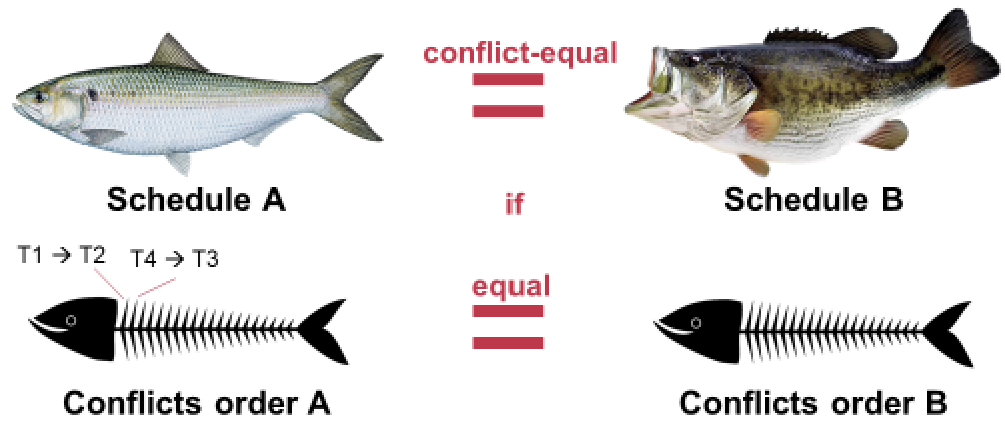


final state serializabliity 只要结果正确即可，但并不serializable T1,T2交替执行，任何一个线性执行是不会有这样的情况。



conflict 可以证明符合serializability

Conflict Equivalence



If conflicts-order-A equals to conflicts-order-B, then schedule-A conflict-equals to schedule-B

构建conflict graph来判断是否一样

怎么构建？如下：

Use Conflict Graph to determine the sequential schedule

Conflict Graph

– Nodes are transactions, edges are **directed**

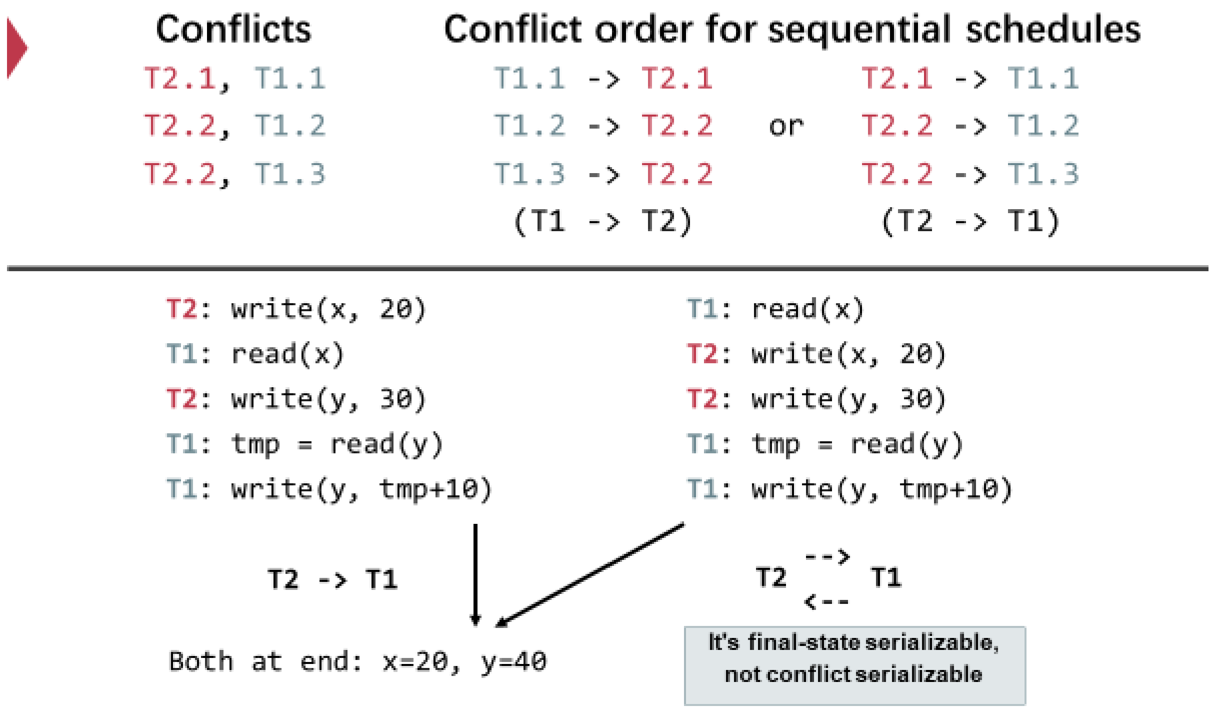
– Edge between T_i and T_j if and only if:

1. T_i and T_j have a conflict between them, and
2. the first step in the conflict occurs in T_i

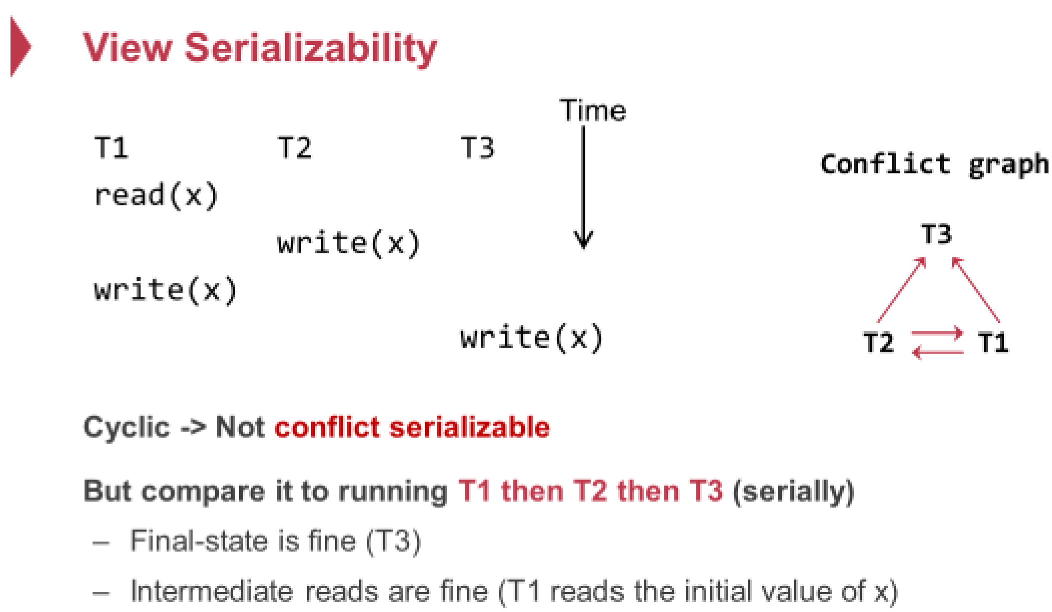
A schedule is conflict serializable if and only if:

– It has an **acyclic** conflict graph

例子：



无环一定是serializable，但有环不一定不是serializable：



先是T1->T2，再是T2->T1，再是T1,T2->T3，存在环

但这个结果和线性跑T1 T2 T3结果是一致的。因为最终结果只用考虑T3写的值就够了。初始状态只用考虑T1读的是初始状态。

因此使用conflict graph是一个充分条件，并不是必要条件，因此需要对这个条件进行放松，即view serializability

View Serializability

Informal definition

– *A schedule is view serializable if the final written state as well as intermediate reads are the same as in some serial schedule*

Formally, for those interested

– Two schedules **S** and **S'** are **view equivalent** if:

- If T_i in S reads an initial value for X, so does T_i in S'
- If T_i in S reads the value written by T_j in S for some X, so does T_i in S'
- If T_i in S does the final write to X, so does T_i in S'

A schedule is **view serializable** if it is **view equivalent** to some serial schedule

18

总结

Question

Why conflict serializability, given that it seems too strict?

Why not focus on view serializability? (Allow more interleaving, & is still correct)

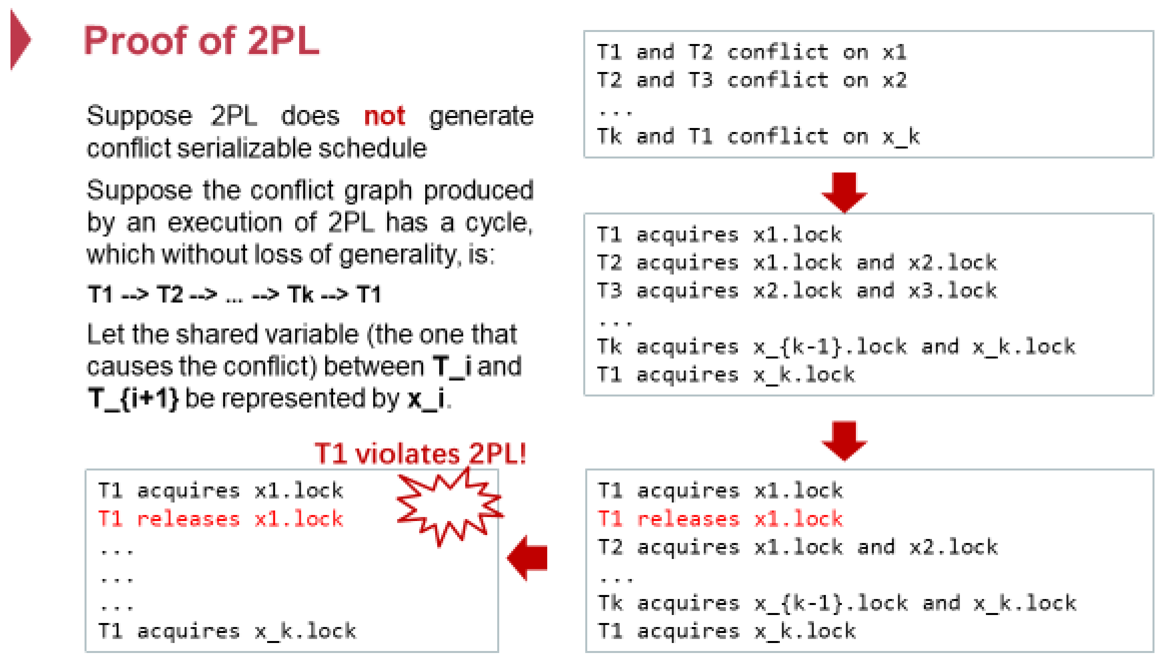
Final-state Serializability	$\supset$	**View Serializability**	$\supset$	**Conflict Serializability**
Care the final state only		Care the final state as well as intermediate read		Care the final state as well as all the **data dependency**

19

final state没有考虑过程中读的问题。view虽然好，但非常难判断，是np难问题。conflict很好判断，虽然很严格，但其实用的最多的还是conflict，并且conflict可以实现，比如2PL。

目前数据库中的serializability就是指conflict

2PL即conflict serializability：

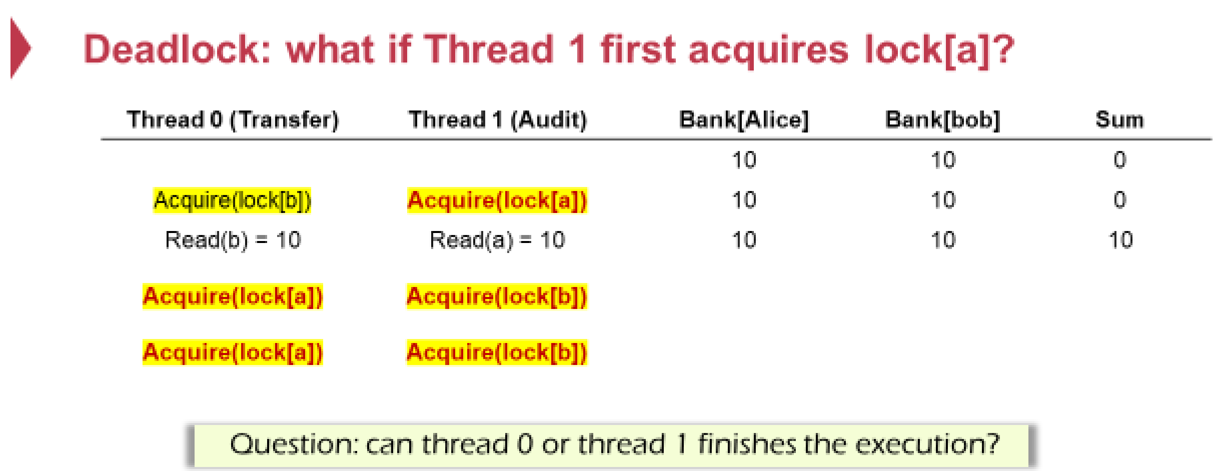


即证无环。用反证法，若有环T1,T2,...,Tn,T1 ，然后可以把这些conflict通过锁展开。如T2要拿T1的锁，T3要拿T2的锁... 而T1必须要release，T2才能拿到锁，而T1最后还需要拿Tn的锁，不符合2PL的定义（2PL拿完锁后就不能放锁）因此矛盾

但2PL不一定能全部保证serializability。比如一个list，中间有两个元素，两个元素自己拿锁修改，但list并没有锁，现在依然可以在list后面插入元素。但真实场景往往不会考虑这种麻烦。

Deadlock

例子：



Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[a])	10	10	0
Read(b) = 10	Read(a) = 10	10	10	10
Acquire(lock[a])	Acquire(lock[b])			
Acquire(lock[a])	Acquire(lock[b])			

Methods for resolving the deadlock

1. Acquire locks in a pre-defined order Prevention

– Not support general TX: TX must know the read/write sets before execution

2. Detect deadlock by calculating the conflict graph Detection

– If there is a cycle, then there must be a deadlock

– Abort one TX to break the cycle

– High cost for detection

3. Using heuristics (e.g., timestamp) to pre-abort the TXs Retry

– May have false positive, or live locks

36

1. 拿锁顺序如果是一样，则不会出现死锁问题。但不一定所有transaction都可以满足这个条件
2. 拿锁的时候可以把等待的环给拿出来，然后abort一个transaction。但开销太大，要去构建一个环
3. 用时间来判断是否死锁，若超时则直接abort。但可能会出现误判。

拿锁是一种悲观的判断，那么也有乐观的控制方法

Optimistic concurrency control

Executing TXs optimistically w/o acquiring the lock

Checks the results of TX before it commits

– If violate serializability, then **aborts & retries**

First proposed in 1981, widely used today

HEKATON Microsoft SQL Server　App Engine　CouchDB

39

先不拿锁，如果满足，则继续，如果不满足则直接abort

OCC Executes a Transaction in 3 Phases

Phase 1: Concurrent local processing

- Reads data into a read set
- Buffers writes into a write set

Phase 2: Validation serializability in critical section

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

Phase 3: Commit the results in critical section or abort

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

1. 把读写都放入一个缓存集合
2. 验证是否序列化
3. 成功则提交，失败则abort重新来一次

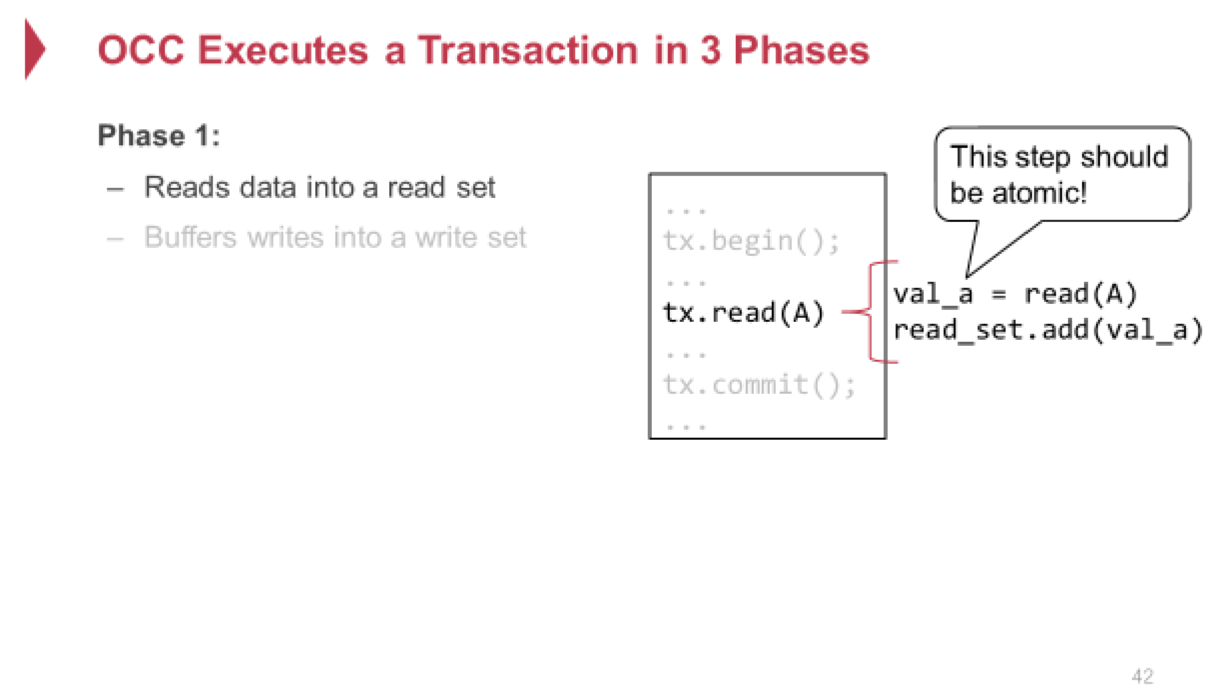
OCC Executes a Transaction in 3 Phases

Before phase one, TX needs to allocate the execution context

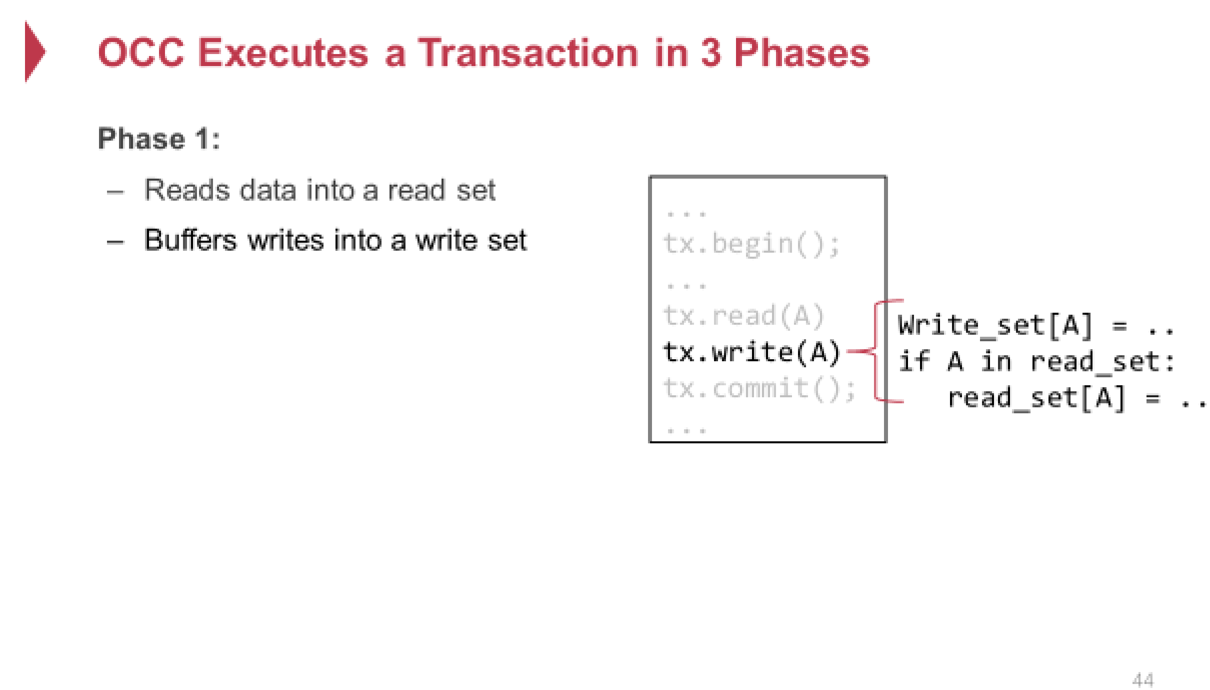
- Read-set & write set

```
...
tx.begin();      Init read_set
                 Init write_set
...
tx.read(A)
...
tx.commit();
...
```


先初始化read/write set



直接读，放入读集合。若后面直接接了一个相同的read（可重复读）则直接到set中去寻找，不然可能被改动。



在write中，同时需要去改变read set。

OCC Executes a Transaction in 3 Phases

Phase 2:

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

```
for d in read_set:
    if d has changed:
        abort()
```

45

OCC Executes a Transaction in 3 Phases

Phase 3:

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

```
for d in read-set:
    if d has changed:
        abort()
for d in write_set:
    write(d)
```

46

phase2，phase3如上

OCC Executes a Transaction in 3 Phases

Phase 3:

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

Phase 2 & 3 should execute in a critical section

- Otherwise, what if a value has changed during validation?

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
...
```

Critical section

```
for d in read_set:
    if d has changed:
        abort()
for d in write_set:
    write(d)
```

47

验证和写入必须要原子操作

How to implement the critical section for phase 2 & 3?

Global lock may satisfy

- The phase 2 & 3 are typically short
  - No TX logic is executed, only the validation

```
def validate_and_commit() // phase 2 & 3
    global_lock.lock()
    for d in read-set:
        if d has changed:
            abort()
    for d in write-set:
        write(d)
    global_lock.unlock()
```

50

最简单方式：直接拿全局锁

效果比2PL好，因为phase1可以直接并发执行

同样可以加更加细粒度的锁

How to implement the critical section for phase 2 & 3?

Using two-phase locking

- Allow more concurrency
- What about the problem of deadlock?

```
def validate_and_commit() // phase 2 & 3
    for d in read-set:
        d.lock()
        if d has changed:
           abort() // abort will release all the held lock
    for d in write-set:
        d.lock()
        write(d)
    // release the locks
    ...
```

在这里可以很好解决死锁问题。可以直接按照顺序拿锁

How to implement the critical section for phase 2 & 3?

Using two-phase locking

- Allow more concurrency
- Use sort to avoid deadlock; b king overhead as 2PL

Question: do we need to lock the read-set?

```
def validate_and_commit() // phase 2 & 3
    for d in sorted(read-set + write-set):
        d.lock()
    for d in read-set:
        if d has changed:
           abort()
    for d in write-set:
        write(d)
    // release the locks
    ...
```

但如果read和write都要拿锁，其实和2PL的效率是一样的。其实可以直接不拿read的锁

How to implement the critical section for phase 2 & 3?

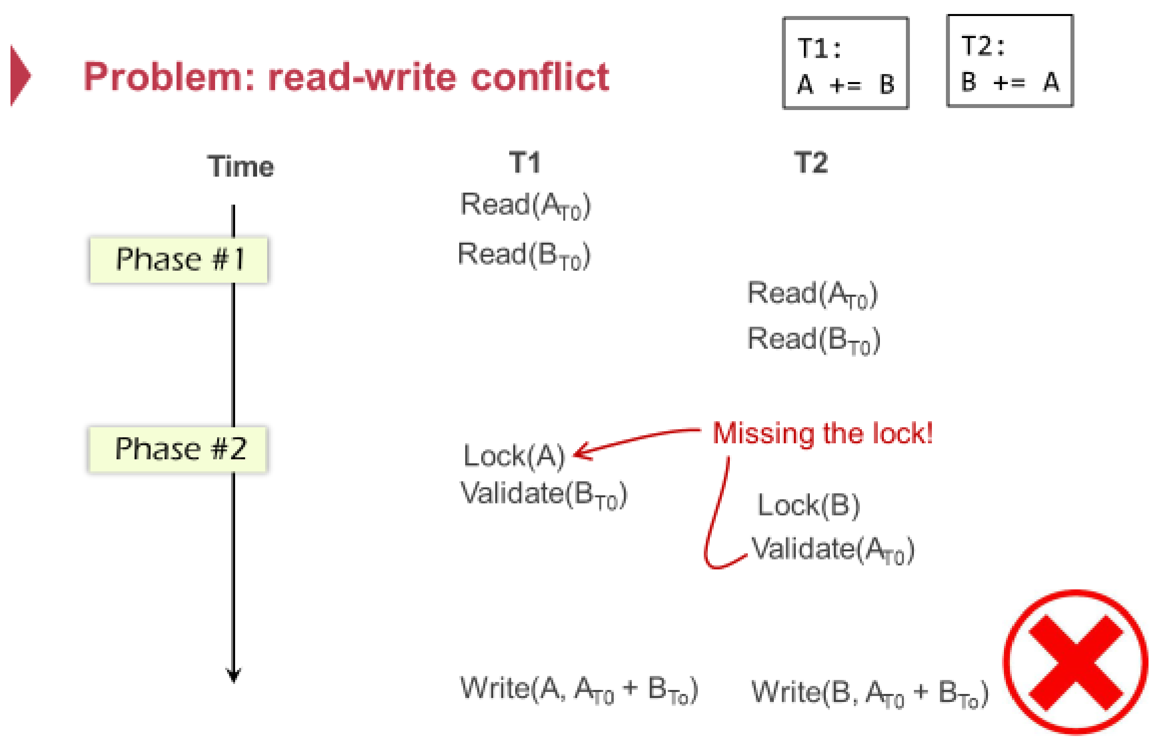
Observation (for the read-set):

- If the validation passes, then it ”appears“ that a lock is held during
- Question: is the following implementation correct?

```
def validate_and_commit() // phase 2 & 3
    for d in sorted(write-set):
        d.lock()
    for d in read-set:
        if d has changed:
           abort()
    for d in write-set:
        write(d)
    // release the locks
    ...
```

54

但这个依然有问题。如果d不在这里被拿了锁，而在其他地方被上锁，那么读d时可能不是最新的结果。如下：



OCC advantage

OCC Advantages

Phase 1: Concurrent local processing

- Reads data into a read set
- Buffers writes into a write set

Phase 2: Validation in critical section

- Validates whether serializability is guaranteed:
- Has any data in the read set been modified?

Phase 3: Commit the results in critical section or abort

- Aborts: aborts the transaction if validation fails
- Commits: installs the write set and commits the transaction

Needs synchronization (e.g., with lock), but usually very short at low contention

59

OCC Advantages

Use reads more than writes (for each read-only data access)

- OCC (in the optimal case, i.e., no abort)
  - 1 read to read the data value
  - 1 read to validate whether the value has been changed or not (as well as locked)
- 2PL
  - 1 operation to acquire the lock (typically an atomic CAS)
  - 1 read to read the data value
  - 1 write to release the lock
    - A single CPU write is atomic, no need to do the atomic CAS

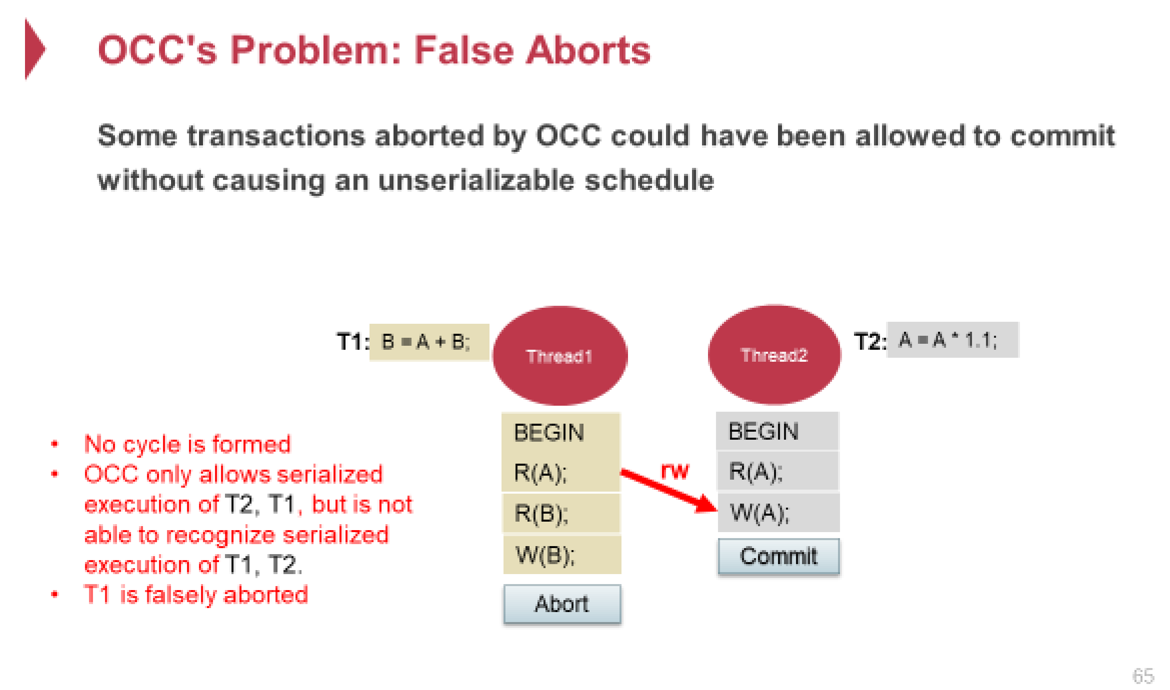
Locking is costly especially compared to reads

60

OCC problem

OCC中可能会出现误判的情况

首先是误以为出现矛盾的误判，如下

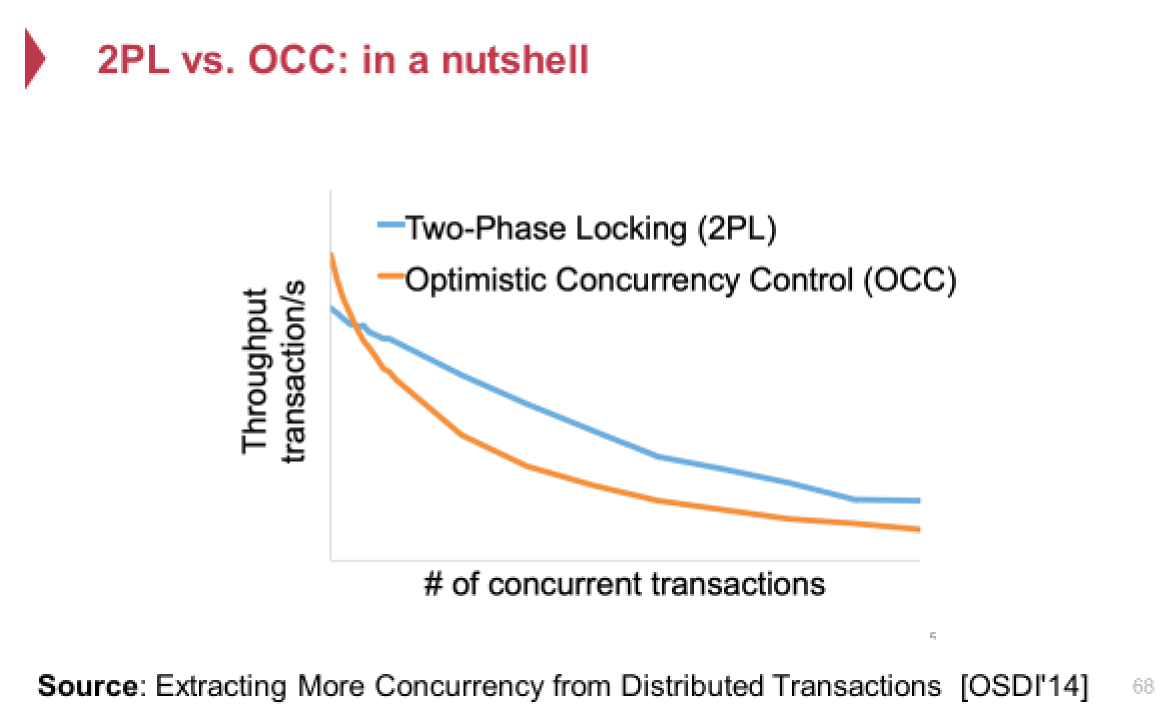


在这个情境中，两个事务实际上并没有冲突。尽管T1读取了A的值，但它并没有修改A；而T2只修改了A的值，没有读取或修改B的值。

但是，当T2先于T1提交时，OCC可能会认为T1读取的A的值已经过时，因为T2已经修改了A的值。因此，T1可能会被中止，即使它和T2实际上是可以并发执行的，不会导致任何数据不一致。

同时，OCC在执行一个耗时很长的操作时，非常容易出现错误。

2PL v.s. OCC



可以看出，在小数量事务时OCC快于2PL，但随着事务变多，OCC效率不如2PL

Lock preliminary

Lock是由计算机原子操作实现的，而lock的原子性操作往往比其余正常操作更加费时，不仅本身操作时间长，还需要阻隔其余cpu的管线。因此使用lock会降低性能