【计算机系统工程】事务2: Consistency+Isolation

@credits Xingda Wei, IPADS

这里我们主要关注事务的ACID中的consistency和isolation。

1. Before or after atomicity

1.1 What?

我们其实要做的就是避免说我一个操作还没有做完，然后中间结果被别人看见了；所以就要定义说我们的system要满足before or after。

> 定义：
>
> **Concurrent** actions have the **before-or-after property** if their effect from the point of view of their invokers is as **if the actions occurred either completely before or completely after one another**
>
> **几个操作等价于线性执行。**

1.2 How?

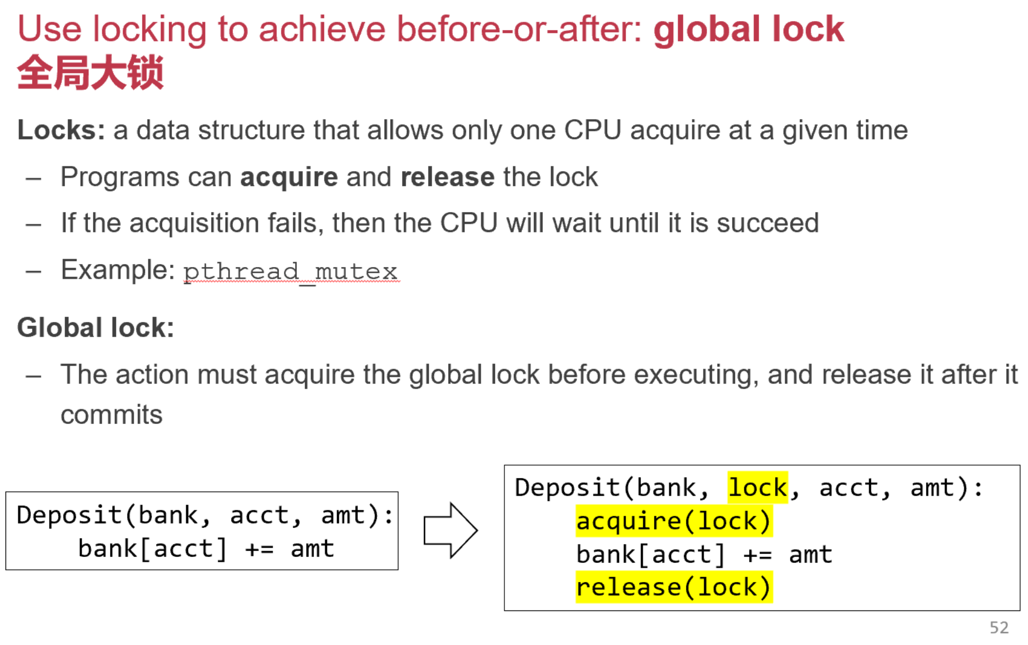
主要有三种方法：

- 2PL
- OCC
- MVCC

2. Locking

2.1 Global Lock

怎么不让别人访问我的中间状态？最简单的方法，上锁啊！



Use locking to achieve before-or-after: **global lock**
全局大锁

Locks: a data structure that allows only one CPU acquire at a given time

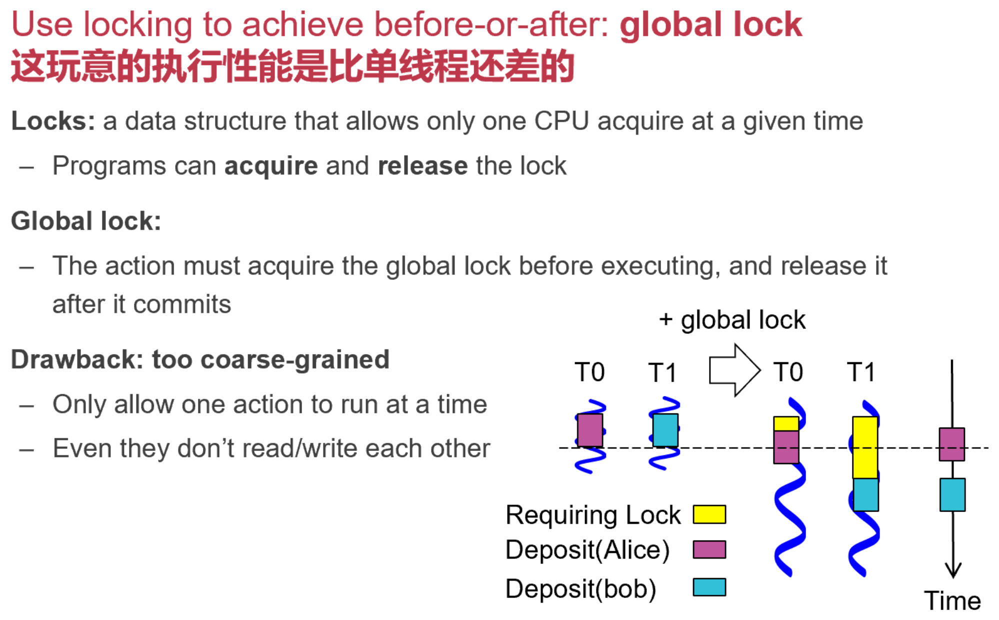
- Programs can **acquire** and **release** the lock
- If the acquisition fails, then the CPU will wait until it is succeed
- Example: `pthread_mutex`

Global lock:

- The action must acquire the global lock before executing, and release it after it commits

```
Deposit(bank, acct, amt):
    bank[acct] += amt
```

```
Deposit(bank, lock, acct, amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```



Use locking to achieve before-or-after: **global lock**
这玩意的执行性能是比单线程还差的

Locks: a data structure that allows only one CPU acquire at a given time

- Programs can **acquire** and **release** the lock

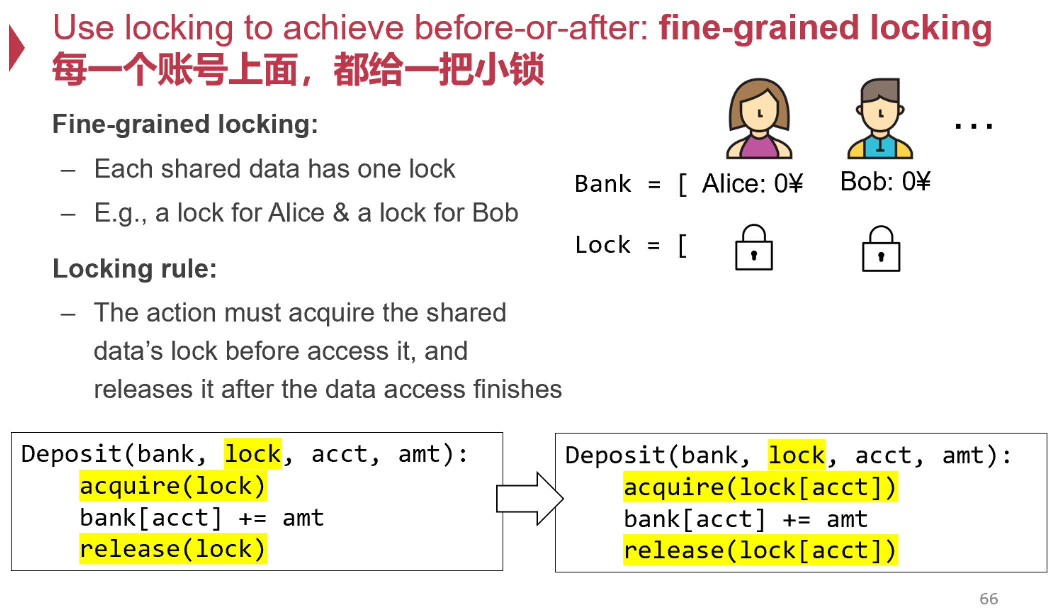
Global lock:

- The action must acquire the global lock before executing, and release it after it commits

Drawback: too coarse-grained

- Only allow one action to run at a time
- Even they don't read/write each other

2.2 2PL



Use locking to achieve before-or-after: **fine-grained locking**
每一个账号上面，都给一把小锁

Fine-grained locking:

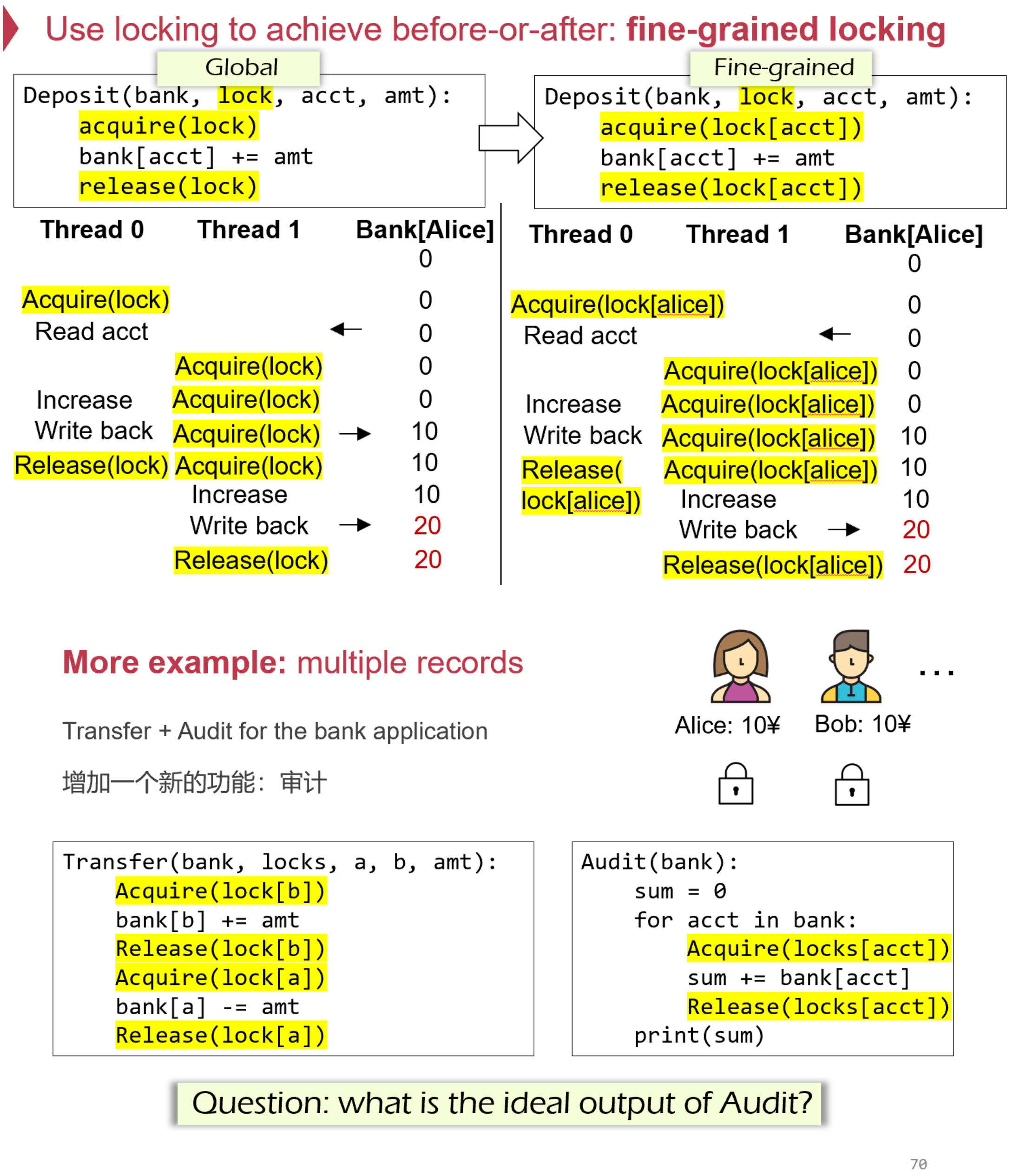
- Each shared data has one lock
- E.g., a lock for Alice & a lock for Bob

Locking rule:

- The action must acquire the shared data's lock before access it, and releases it after the data access finishes

```
Deposit(bank, lock, acct, amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

```
Deposit(bank, lock, acct, amt):
    acquire(lock[acct])
    bank[acct] += amt
    release(lock[acct])
```



Use locking to achieve before-or-after: **fine-grained locking**

Global

```
Deposit(bank, lock, acct, amt):
    acquire(lock)
    bank[acct] += amt
    release(lock)
```

Fine-grained

```
Deposit(bank, lock, acct, amt):
    acquire(lock[acct])
    bank[acct] += amt
    release(lock[acct])
```

Thread 0	Thread 1	Bank[Alice]
		0
Acquire(lock)		0
Read acct	←	0
	Acquire(lock)	0
Increase	Acquire(lock)	0
Write back	Acquire(lock) →	10
Release(lock)	Acquire(lock)	10
	Increase	10
	Write back →	20
	Release(lock)	20

Thread 0	Thread 1	Bank[Alice]
		0
Acquire(lock[alice])		0
Read acct	←	0
	Acquire(lock[alice])	0
Increase	Acquire(lock[alice])	0
Write back	Acquire(lock[alice])	10
Release(lock[alice])	Acquire(lock[alice])	10
	Increase	10
	Write back →	20
	Release(lock[alice])	20

More example: multiple records

Transfer + Audit for the bank application

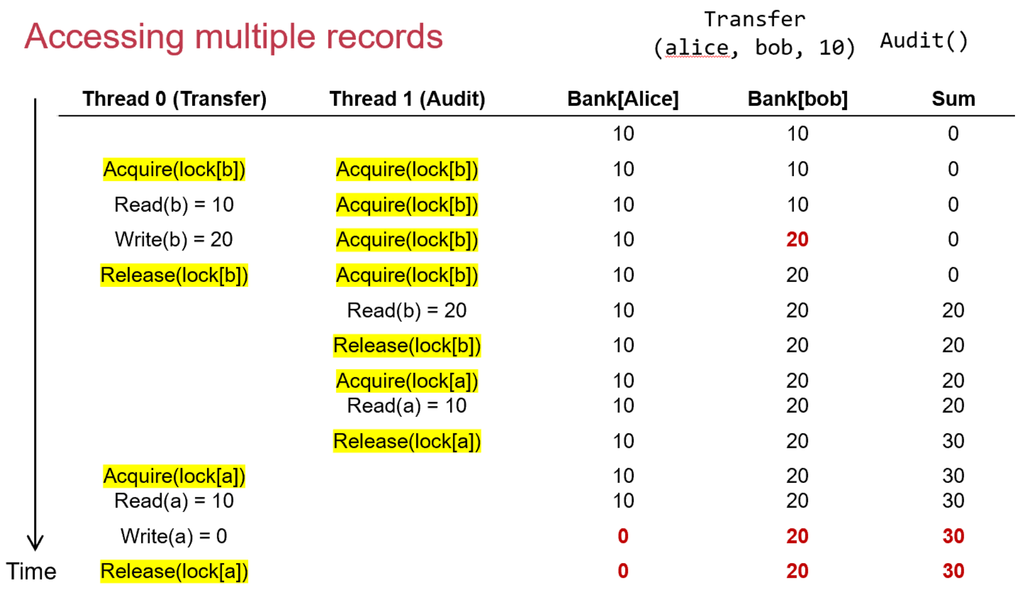
增加一个新的功能：审计

Alice: 10¥ Bob: 10¥ …

```
Transfer(bank, locks, a, b, amt):
    Acquire(lock[b])
    bank[b] += amt
    Release(lock[b])
    Acquire(lock[a])
    bank[a] -= amt
    Release(lock[a])
```

```
Audit(bank):
    sum = 0
    for acct in bank:
        Acquire(locks[acct])
        sum += bank[acct]
        Release(locks[acct])
    print(sum)
```

Question: what is the ideal output of Audit?



Accessing multiple records

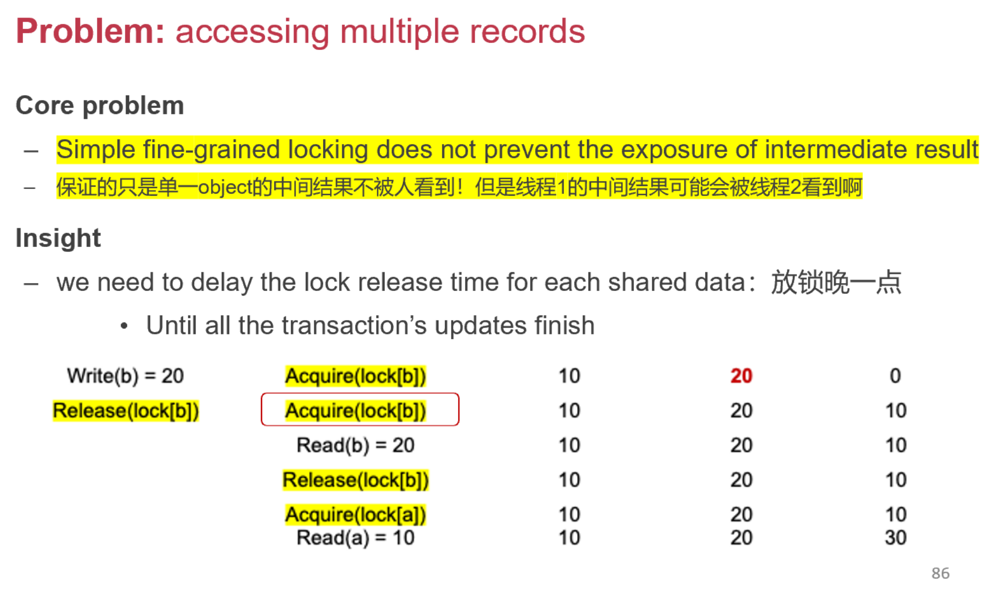
Transfer (alice, bob, 10) Audit()

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = 20	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a])	10	20	20
	Read(a) = 10	10	20	20
	Release(lock[a])	10	20	30
Acquire(lock[a])		10	20	30
Read(a) = 10		10	20	30
Write(a) = 0		0	20	30
Release(lock[a])		0	20	30

Time

还是对“你不能看见我的中间状态”的定义的解释。

之前加一把锁能解决问题，本质是我们只有一个变量。现在变量多了，我可能要改两个变量，如果某个时刻一个变量改了，另一个没有改，也可以称之为中间状态。



Problem: accessing multiple records

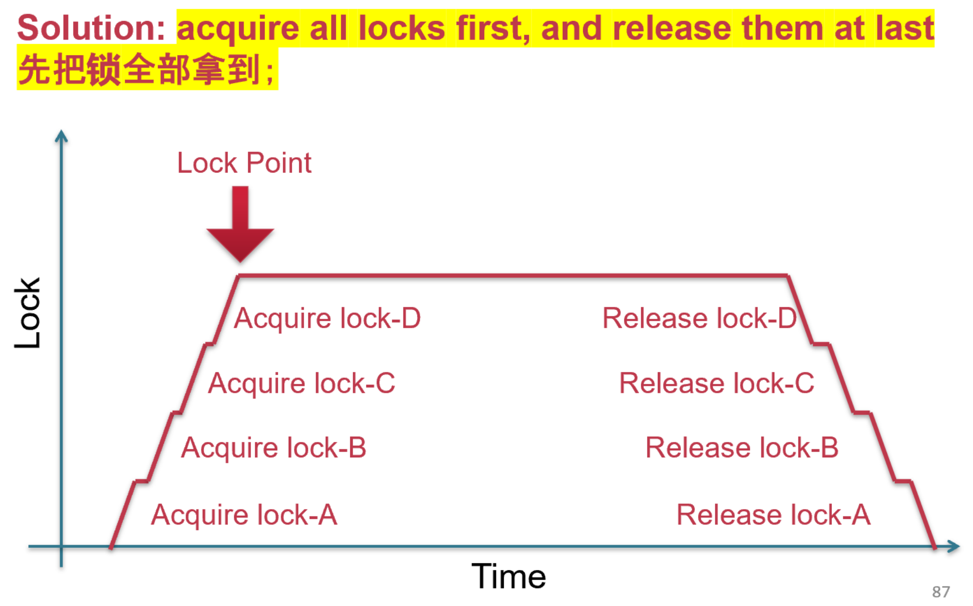
Core problem

- Simple fine-grained locking does not prevent the exposure of intermediate result
- 保证的只是单一object的中间结果不被人看到！但是线程1的中间结果可能会被线程2看到啊

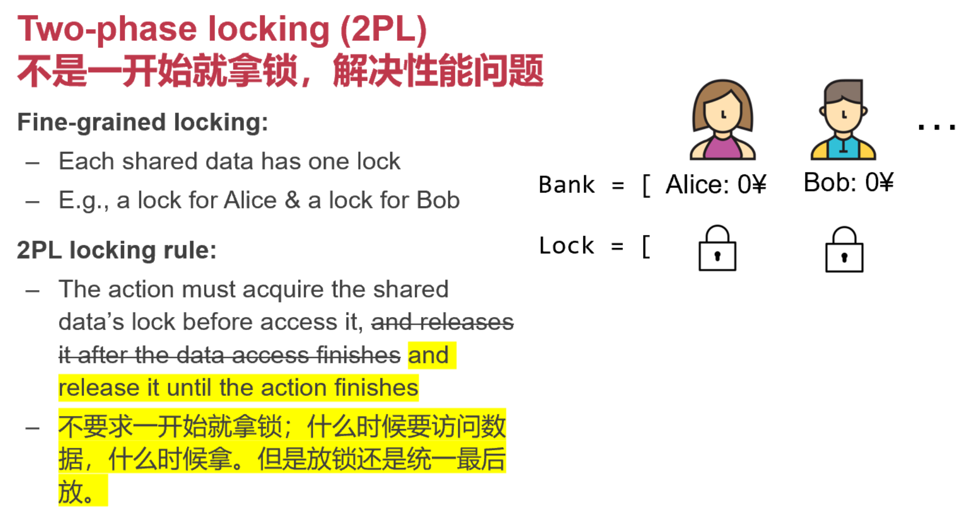
Insight

- we need to delay the lock release time for each shared data：放锁晚一点
  - Until all the transaction’s updates finish

Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	10
	Read(b) = 20	10	20	10
	Release(lock[b])	10	20	10
	Acquire(lock[a])	10	20	10
	Read(a) = 10	10	20	30



Solution: acquire all locks first, and release them at last 先把锁全部拿到；



Two-phase locking (2PL) 不是一开始就拿锁，解决性能问题

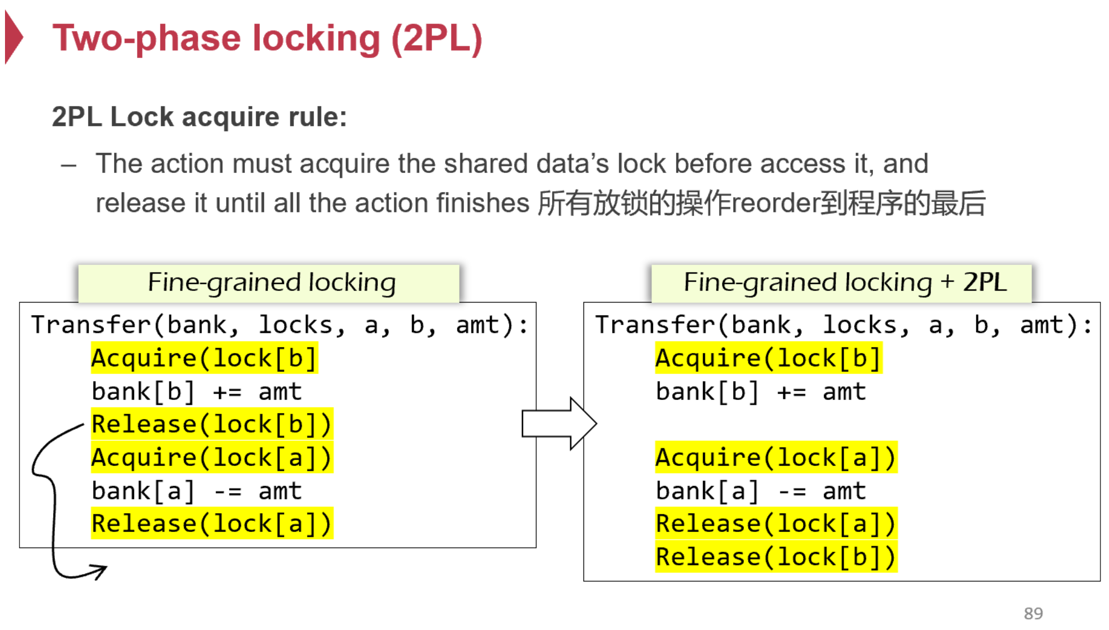
Fine-grained locking:

- Each shared data has one lock
- E.g., a lock for Alice & a lock for Bob

2PL locking rule:

- The action must acquire the shared data’s lock before access it, ~~and releases it after the data access finishes~~ and release it until the action finishes
- 不要求一开始就拿锁；什么时候要访问数据，什么时候拿。但是放锁还是统一最后放。

和单纯的fine-grained locking的最大的区别就是把放锁的步骤放到了最后：



Two-phase locking (2PL)

2PL Lock acquire rule:

– The action must acquire the shared data’s lock before access it, and release it until all the action finishes 所有放锁的操作reorder到程序的最后

Fine-grained locking

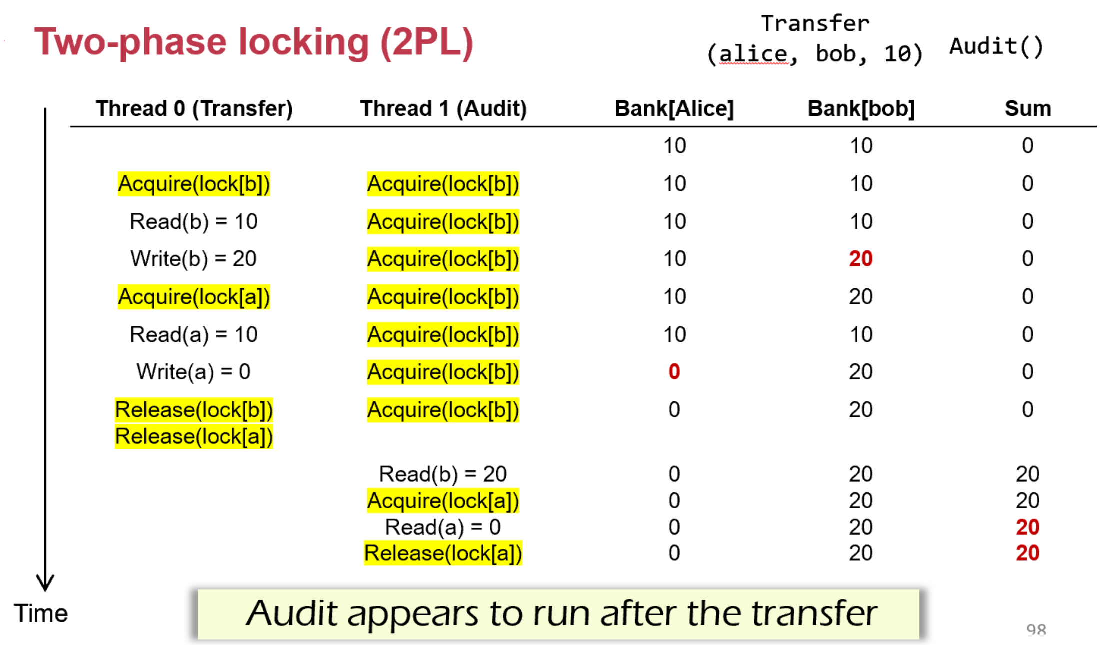
```
Transfer(bank, locks, a, b, amt):
    Acquire(lock[b]
    bank[b] += amt
    Release(lock[b])
    Acquire(lock[a])
    bank[a] -= amt
    Release(lock[a])
```

Fine-grained locking + 2PL

```
Transfer(bank, locks, a, b, amt):
    Acquire(lock[b]
    bank[b] += amt

    Acquire(lock[a])
    bank[a] -= amt
    Release(lock[a])
    Release(lock[b])
```

89



Two-phase locking (2PL)

```
Transfer
(alice, bob, 10)   Audit()
```

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	**20**	0
Acquire(lock[a])	Acquire(lock[b])	10	20	0
Read(a) = 10	Acquire(lock[b])	10	10	0
Write(a) = 0	Acquire(lock[b])	**0**	20	0

Release(lock[b])
Release(lock[a])
	Acquire(lock[b])	0	20	0

	Read(b) = 20	0	20	20
	Acquire(lock[a])	0	20	20
	Read(a) = 0	0	20	**20**
	Release(lock[a])	0	20	**20**

Time

Audit appears to run after the transfer

98

2.3 Dead Lock issue of 2PL

说穿了就是两个TX互相等待。

Resolving deadlock

1. Acquire locks in a pre-defined order

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

2. Detect deadlock by calculating the conflict graph

- 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

- May have false positive, or live locks

我们仔细来分析一下这几种解决方案：

1. 所有人按照顺序去拿资源——悲观，认为死锁是经常发生的，要设计一种机制来避免这种问题
2. 探测死锁，如果发现了死锁，就让某个人去放锁——乐观，认为死锁是较为罕见的，发现再说

3. OCC：乐观并发控制

3.1 Why OCC

Problem of 2PL for before-or-after: locking overhead + deadlock

- Pessimistically execute the TX to avoid race conditions

Executing TXs optimistically w/o acquiring the lock

- Checks the results of TX before it commits
- If violate serializability, then **aborts & retries**

3.2 What OCC

Phase 1: Concurrent local processing 不去拿锁，直接执行事务

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

Phase 2: Validation serializability in critical section 判断执行过程中是否遇到race condition

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

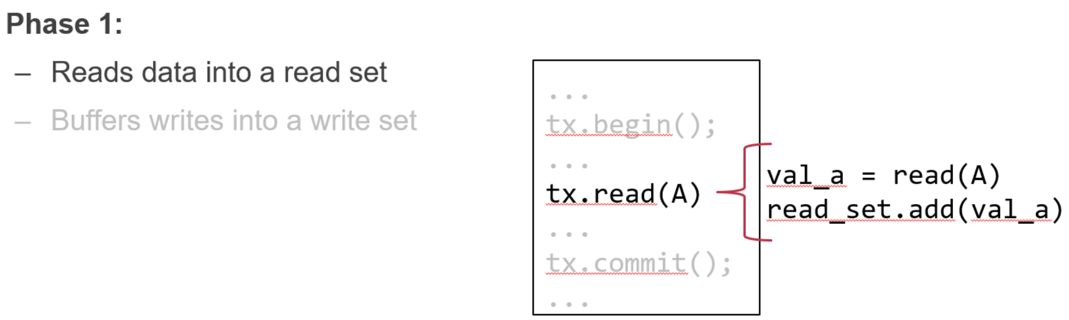
Phase 3: Commit the results in critical section or abort

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

3.3 How OCC

核心就是维护一个Read_set和一个Write_set：

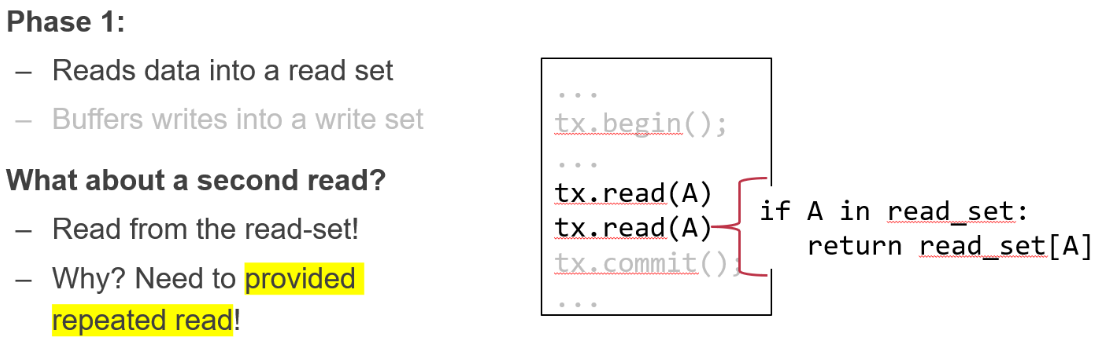
- 然后读数据都从读集合里面读；
- 写数据先不在数据库里面源数据上操作，先buffer到写集合里面



Phase 1:

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

```
...
tx.begin();
...
tx.read(A)     val_a = read(A)
               read_set.add(val_a)
...
tx.commit();
...
```



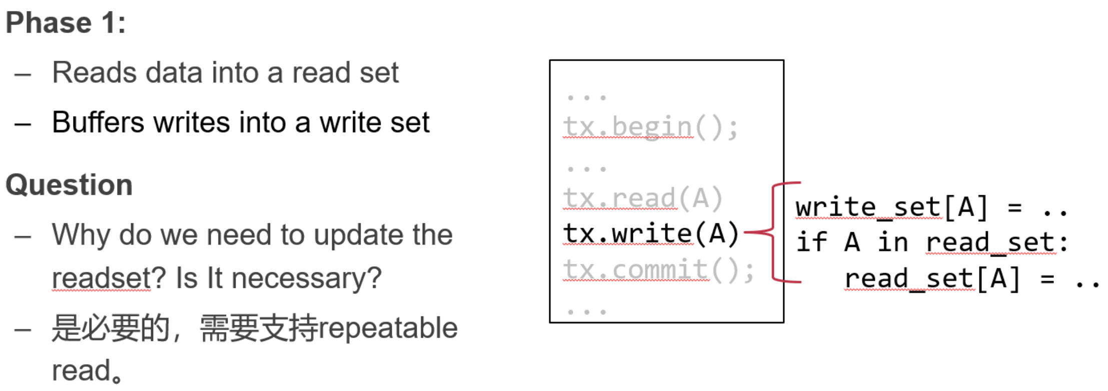
Phase 1:

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

What about a second read?

- Read from the read-set!
- Why? Need to provided repeated read!

```
...
tx.begin();
...
tx.read(A)
tx.read(A)     if A in read_set:
                  return read_set[A]
tx.commit();
...
```



Phase 1:

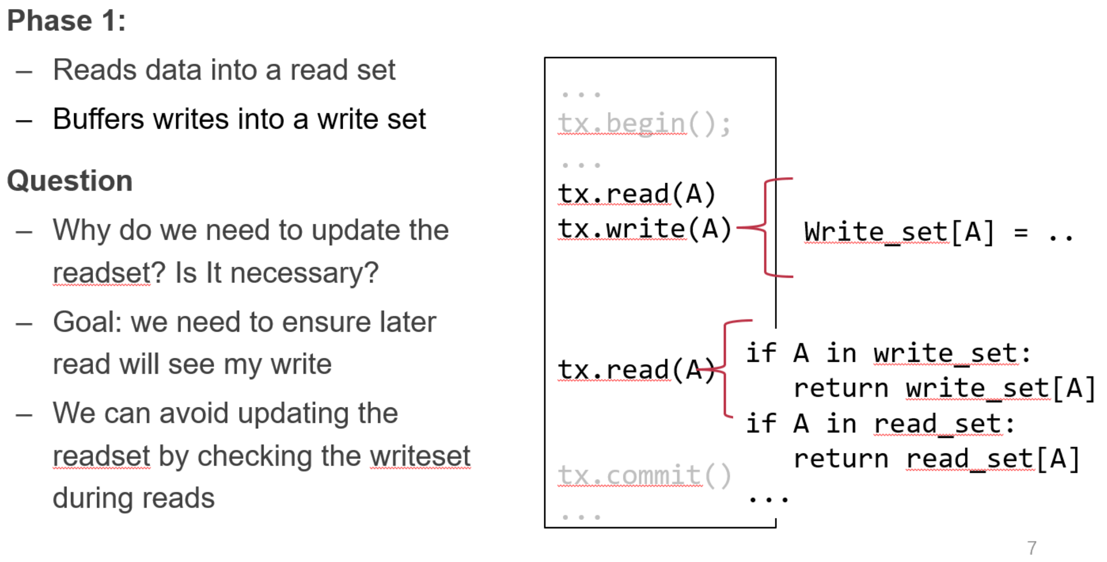
- Reads data into a read set
- Buffers writes into a write set

Question

- Why do we need to update the readset? Is It necessary?
- 是必要的，需要支持repeatable read。

```
...
tx.begin();
...
tx.read(A)
tx.write(A)    write_set[A] = ..
               if A in read_set:
                  read_set[A] = ..
tx.commit();
...
```

当然，其实我们还有另一种实现，也可以达到相同的效果：



Phase 1:

– Reads data into a read set

– Buffers writes into a write set

Question

– Why do we need to update the readset? Is It necessary?

– Goal: we need to ensure later read will see my write

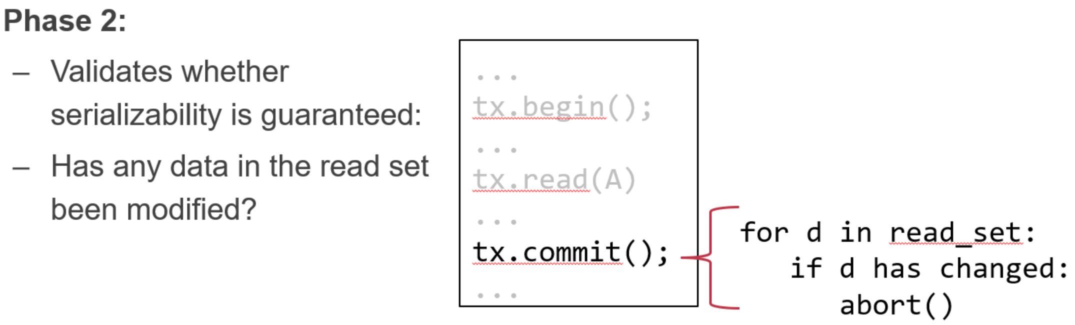
– We can avoid updating the readset by checking the writeset during reads

```
...
tx.begin();
...
tx.read(A)
tx.write(A)      Write_set[A] = ..


tx.read(A)    if A in write_set:
                 return write_set[A]
              if A in read_set:
                 return read_set[A]
tx.commit()   ...
...
```

7

然后就是第二阶段：准备commit当前事务



Phase 2:

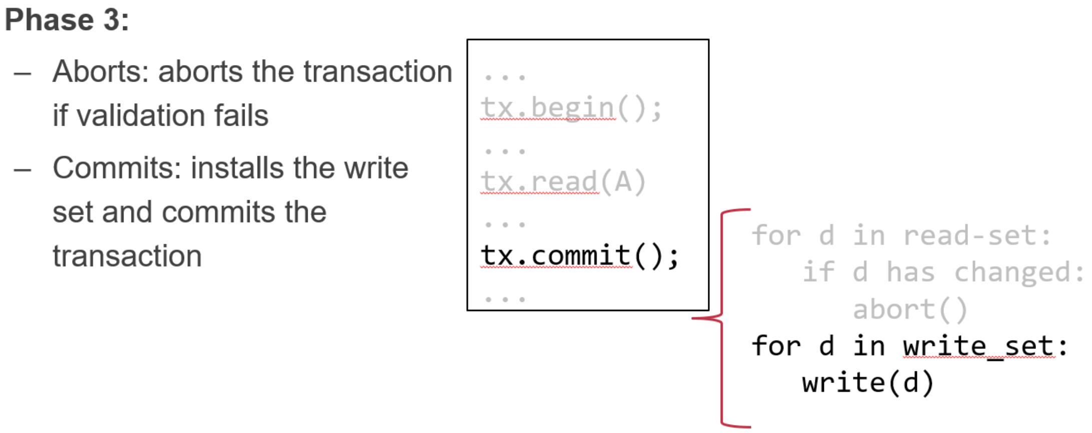
– Validates whether serializability is guaranteed:

– 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()
...
```

也就是验证说各个事务之间是否保证了serializability，如果是，那么提交当前的事务；如果不是，则abort

如果事务成功commit，将之前buffer在write_set中的写落实到数据库中



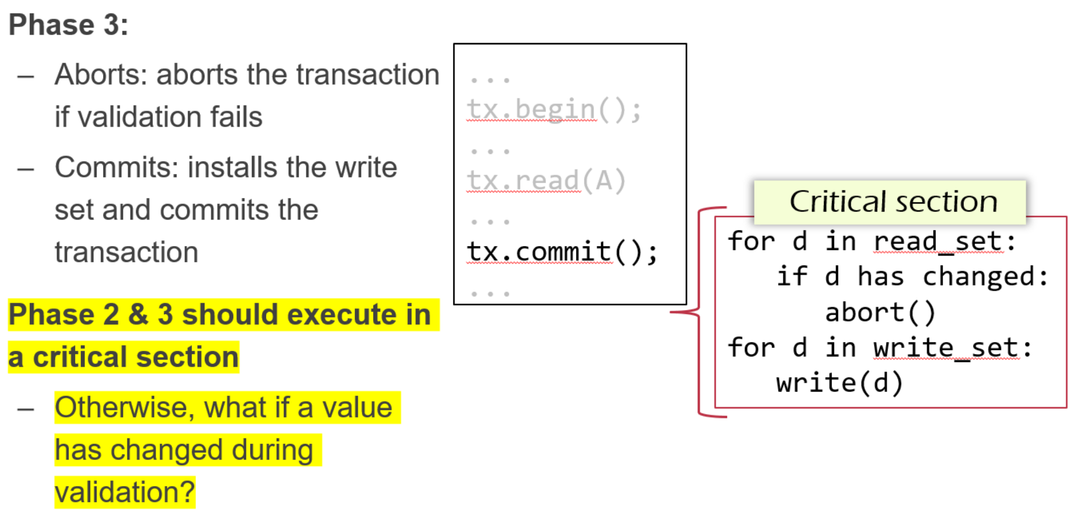
Phase 3:

– 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)
```

当然，请注意，我们要求上面的两步操作是原子性的：



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

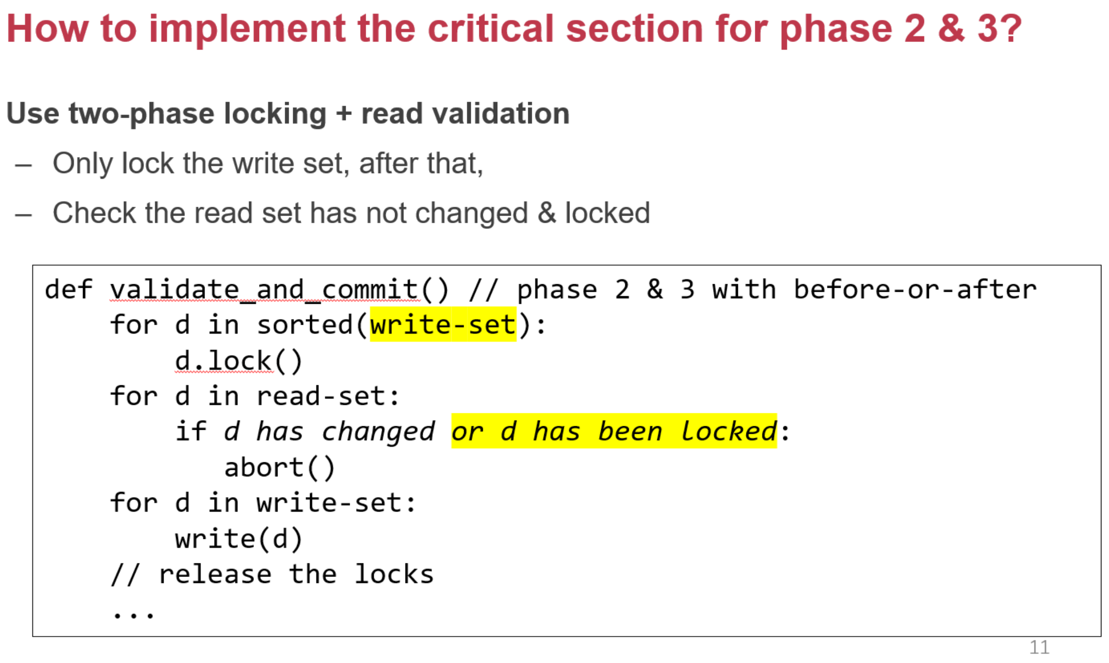
- Otherwise, what if a value has changed during validation?

```
...
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)
```

那么如何实现OCC中的critical section？——上锁。



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

Use two-phase locking + read validation

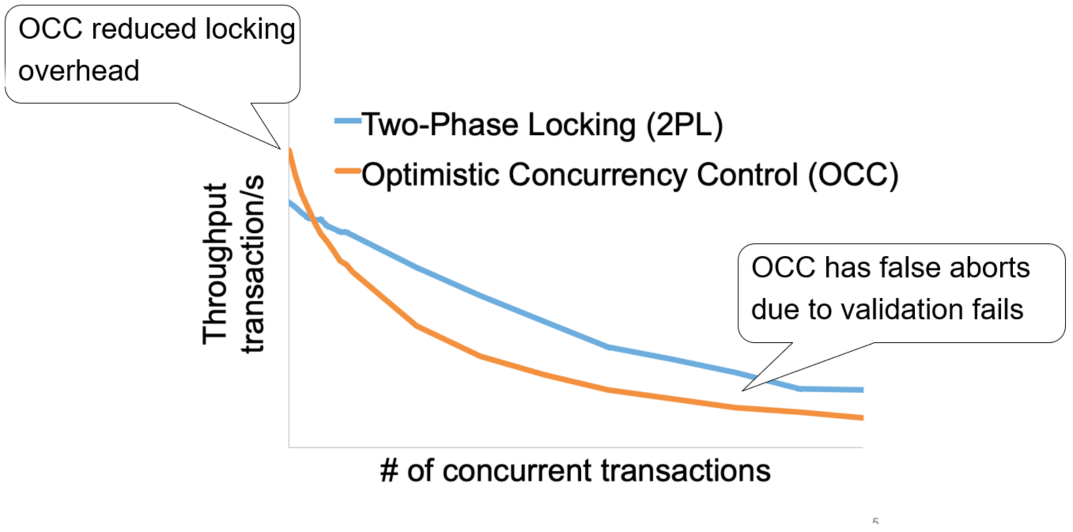
- Only lock the write set, after that,
- Check the read set has not changed & locked

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

11

- 首先，我们可以对write_set进行排序，因为我们知道有哪些数据，因此可以通过这种方式避免死锁。
- 然后就是去检查Read_set，这里稍微有些歧义，这里所谓的 `or d has been locked` ，应该是指被其他事务锁住，当前事务无法获得lock，那么当前事务需要abort掉。

3.4 OCC v.c. 2PL

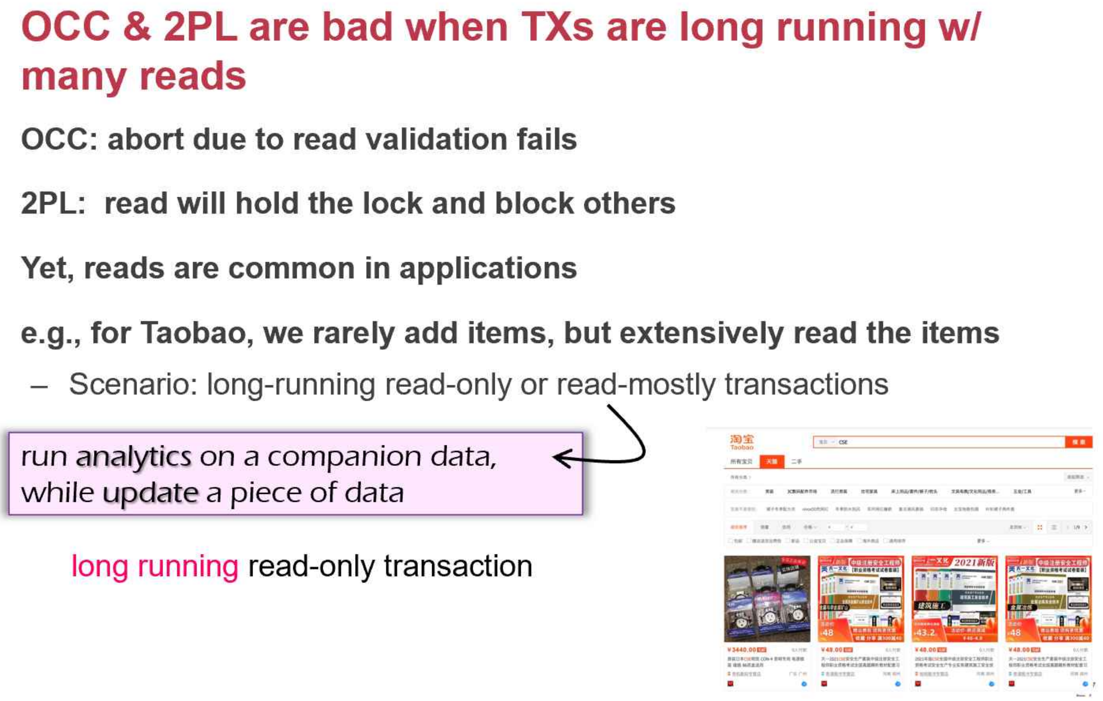


Source: Extracting More Concurrency from Distributed Transactions [OSDI'14] 12

当数据的race condition发生得非常频繁的时候，不适合用OCC。因为会经常发生abort+retry，这样带来的损失比拿锁更大。

4. MVCC：多版本并发控制

4.1 Why MVCC



OCC & 2PL are bad when TXs are long running w/ many reads

OCC: abort due to read validation fails

2PL: read will hold the lock and block others

Yet, reads are common in applications

e.g., for Taobao, we rarely add items, but extensively read the items

– Scenario: long-running read-only or read-mostly transactions

run analytics on a companion data, while update a piece of data

long running read-only transaction

有很多读的时候，场景很烂；比如逛淘宝你只是读啊！你下订单才是写；真实场景中读操作远远大于写；occ来做这个，读的过程中，另一个人一旦写了，我就要abort。

4.2 What MVCC

From CMU-15445:

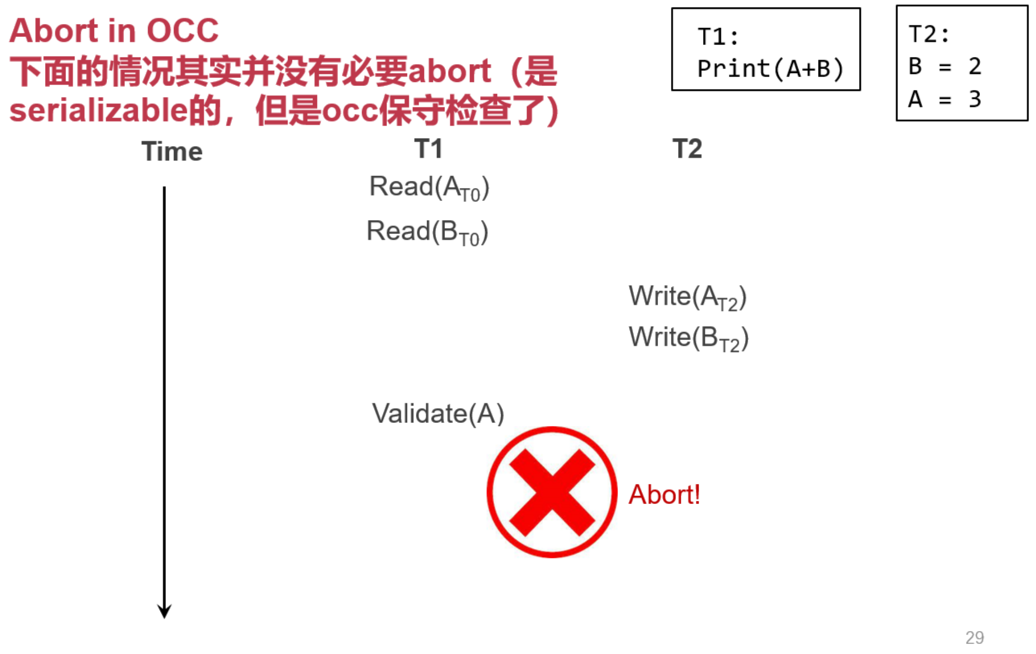
Multi-Version Concurrency Control (MVCC) is a larger concept than just a concurrency control protocol. It involves all aspects of the DBMS's design and implementation. MVCC is the most widely used scheme in DBMSs. It is now used in almost every new DBMS implemented in last 10 years. Even some systems (e.g., NoSQL) that do not support multi-statement transactions use it.

With MVCC, the DBMS maintains multiple physical versions of a single logical object in the database. When a transaction writes to an object, the DBMS creates a new version of that object. When a transaction reads an object, it reads the newest version that existed when the transaction started.

The fundamental concept/benefit of MVCC is that writers do not block readers and readers do not block writers. This means that one transaction can modify an object while other transactions read old versions. Writers may still block other writers if they are writing the same object, since there is still a lock on the versions related to the database object.

One advantage of using MVCC is that read-only transactions can read a consistent **snapshot** of the database without using locks of any kind. Additionally, multi-versioned DBMSs can easily support *time-travel queries*, which are queries based on the state of the database at some other point in time (e.g. performing a query on the database as it was 3 hours ago).

1. Can we avoid validation for reads?

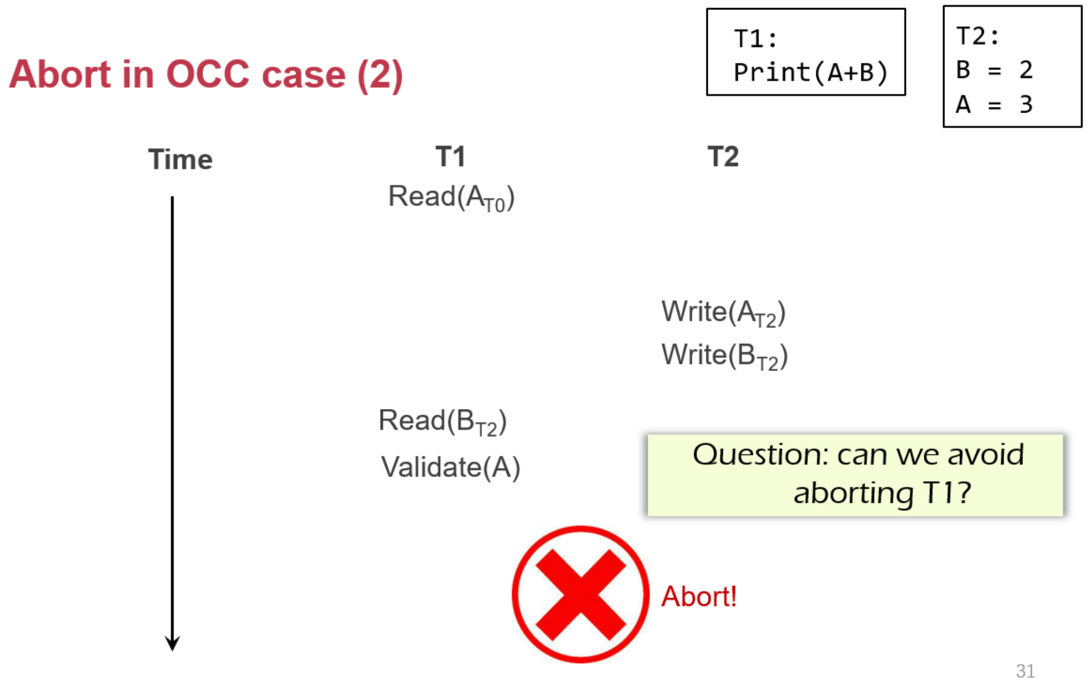


Abort in OCC
下面的情况其实并没有必要abort（是serializable的，但是occ保守检查了）

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

在下面这个case里面，occ其实是正确地abort掉了：



Abort in OCC case (2)

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

Time

T1

T2

Read(A_{T0})

Write(A_{T2})

Write(B_{T2})

Read(B_{T2})

Validate(A)

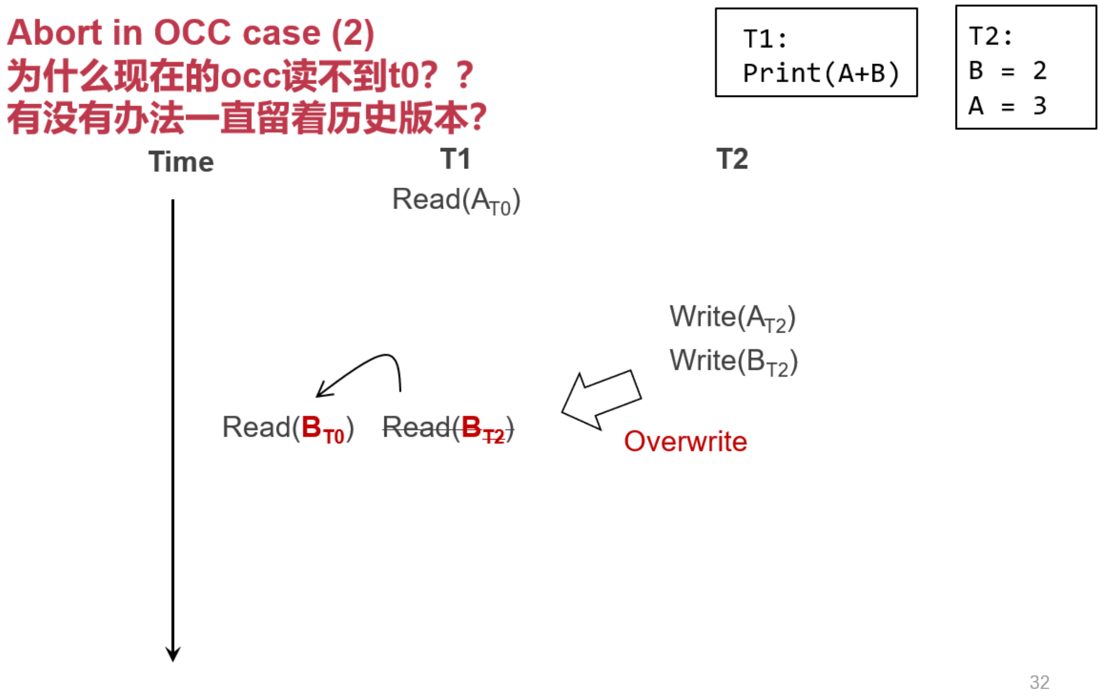
Question: can we avoid aborting T1?

Abort!

31

但我们有没有一些方法，做一些小的trick，来让T1也避免这个abort?

本质上就是T1在T2执行完之后，还能读到T2执行之前的状态就可以了！



Abort in OCC case (2)
为什么现在的occ读不到t0？？
有没有办法一直留着历史版本?

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

Time

T1

T2

Read(A_{T0})

Write(A_{T2})

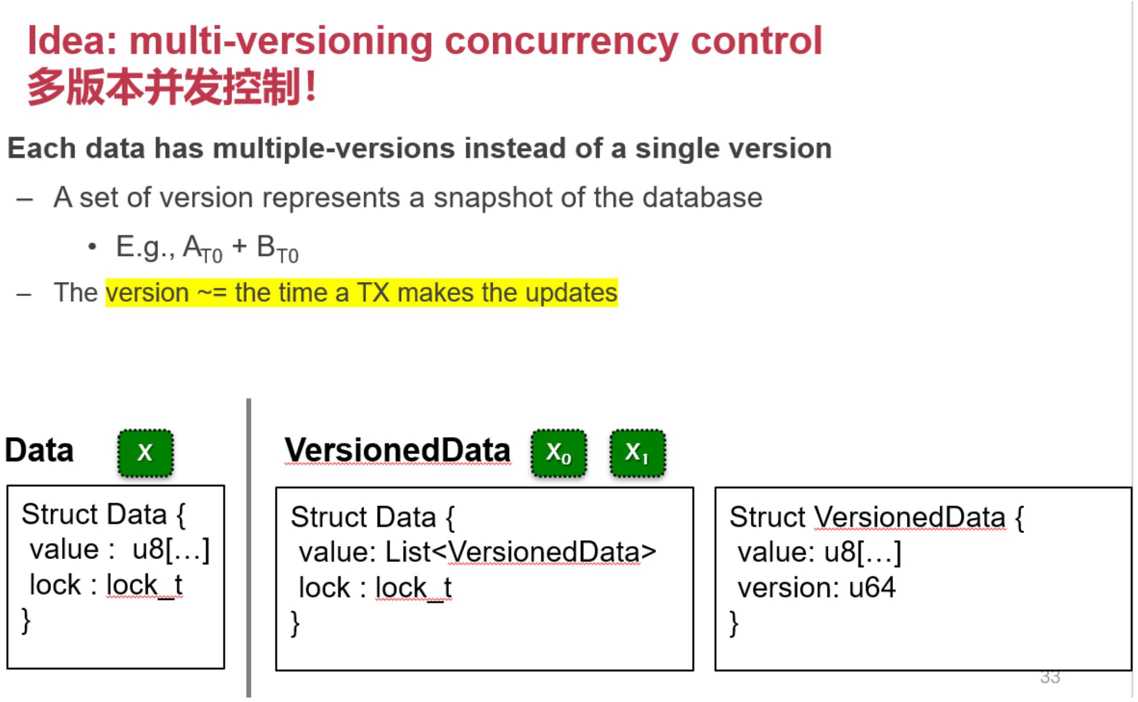
Write(B_{T2})

Read(B_{T0}) ~~Read(B_{T2})~~

Overwrite

32

2. 多版本并发控制：idea



Idea: multi-versioning concurrency control 多版本并发控制!

Each data has multiple-versions instead of a single version

- A set of version represents a snapshot of the database
  - E.g., A_{T0} + B_{T0}
- The version ~= the time a TX makes the updates

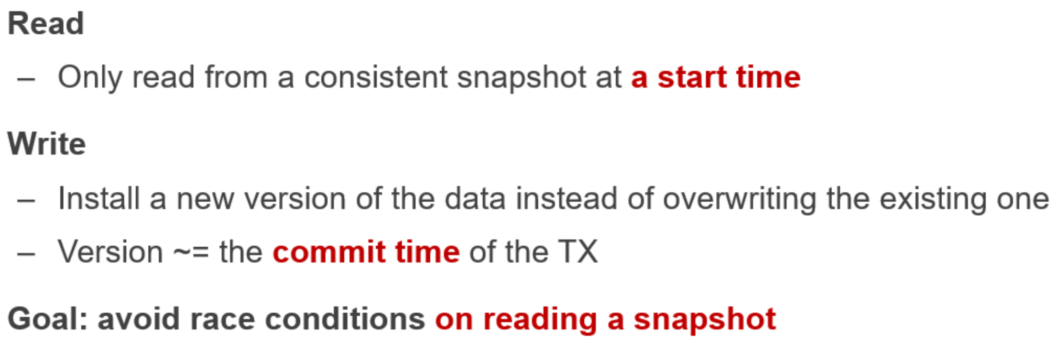
Data X

```
Struct Data {
 value :  u8[…]
 lock : lock_t
}
```

VersionedData X_0 X_1

```
Struct Data {
 value: List<VersionedData>
 lock : lock_t
}
```

```
Struct VersionedData {
 value: u8[…]
 version: u64
}
```



Read

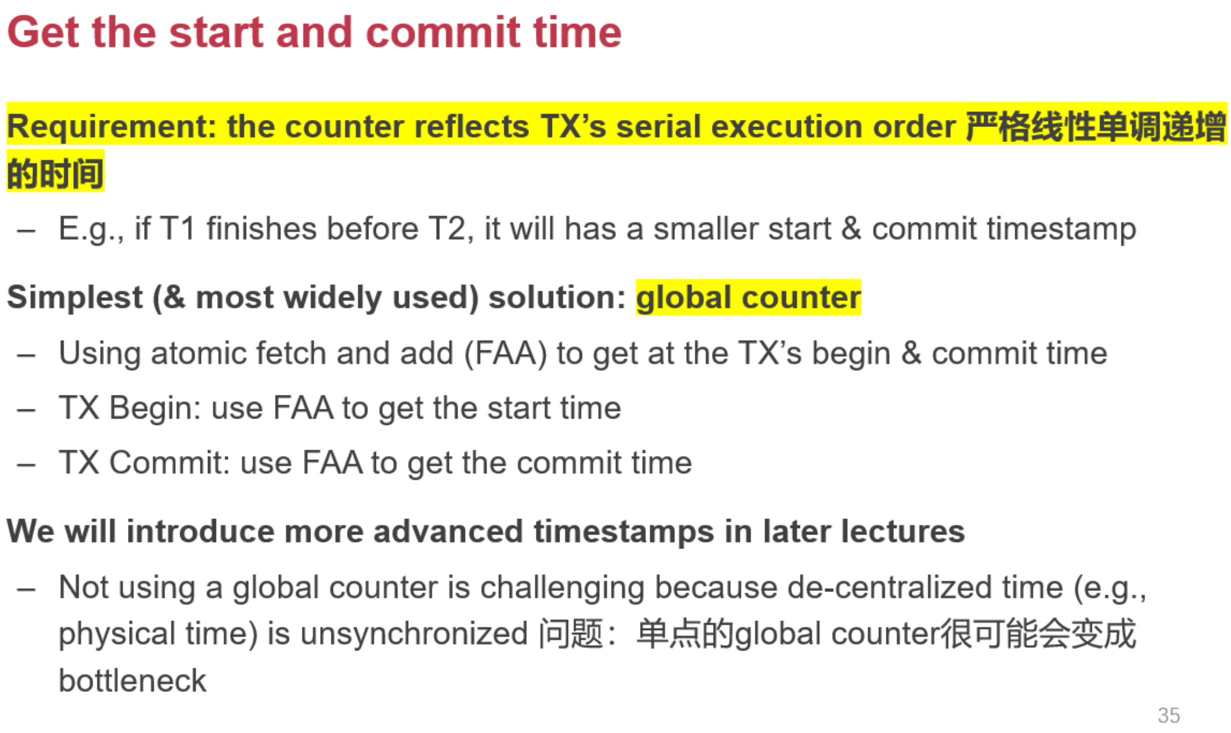
- Only read from a consistent snapshot at **a start time**

Write

- Install a new version of the data instead of overwriting the existing one
- Version ~= the **commit time** of the TX

Goal: avoid race conditions on reading a snapshot

CPU里面放一个global counter，来实现单调递增的时间。这样的问题就是我们只有一个单点，就会变成性能瓶颈。（当然不用glocal counter是非常难的，后面分布式的时候我们会讲一些高级的时间戳）



Get the start and commit time

Requirement: the counter reflects TX's serial execution order 严格线性单调递增的时间

- E.g., if T1 finishes before T2, it will has a smaller start & commit timestamp

Simplest (& most widely used) solution: global counter

- Using atomic fetch and add (FAA) to get at the TX's begin & commit time
- TX Begin: use FAA to get the start time
- TX Commit: use FAA to get the commit time

We will introduce more advanced timestamps in later lectures

- Not using a global counter is challenging because de-centralized time (e.g., physical time) is unsynchronized 问题：单点的global counter很可能会变成 bottleneck

3. 多版本并发控制：1.0

Try #1: Optimize OCC w/ MV (incomplete)
这个版本不太对，有corner case没有处理!

Acquire the start time 事务开始的时候获得【开始时间】

Phase 1: Concurrent local processing

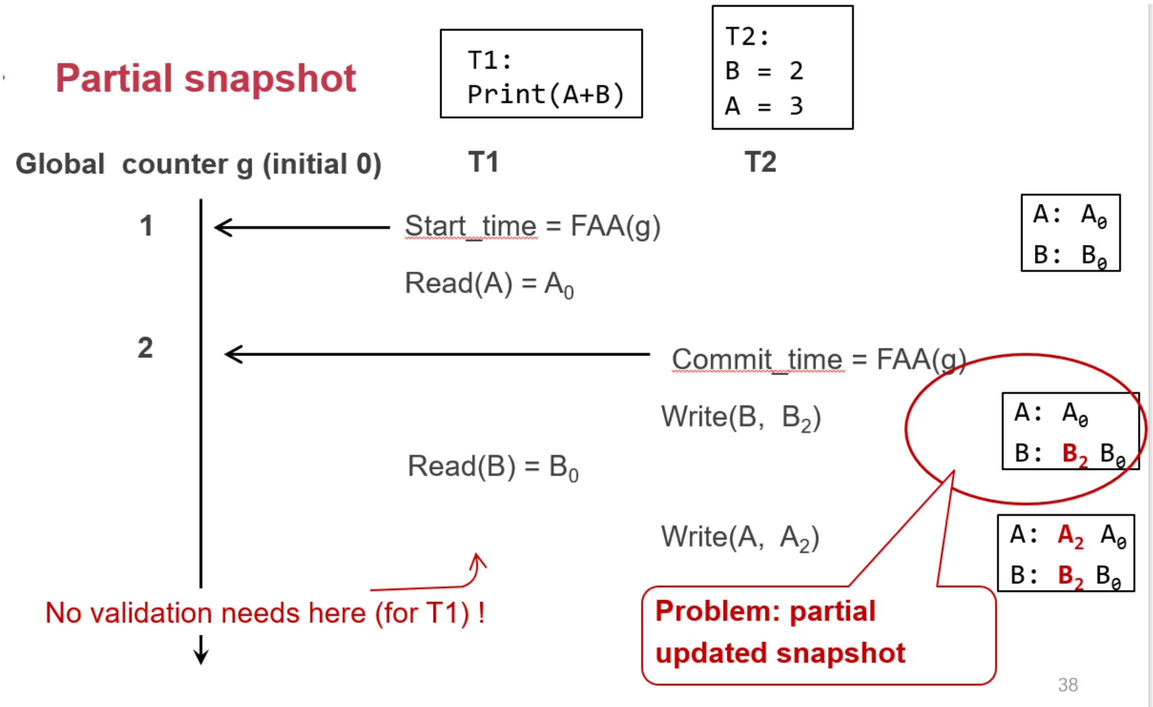
- Reads data belongs to the snapshot closest to the start time 【这里反正snapshot是不动的，直接读就可以了】
- Buffers writes into a write set【写还是缓存到写集】

Acquire the commit time 以【commit time】作为新版本，加到原来数据list的最后

Phase 2: Commit the results in critical section

- Commits: installs the write set with the commit time

Compared to the OCC, no validation is need!



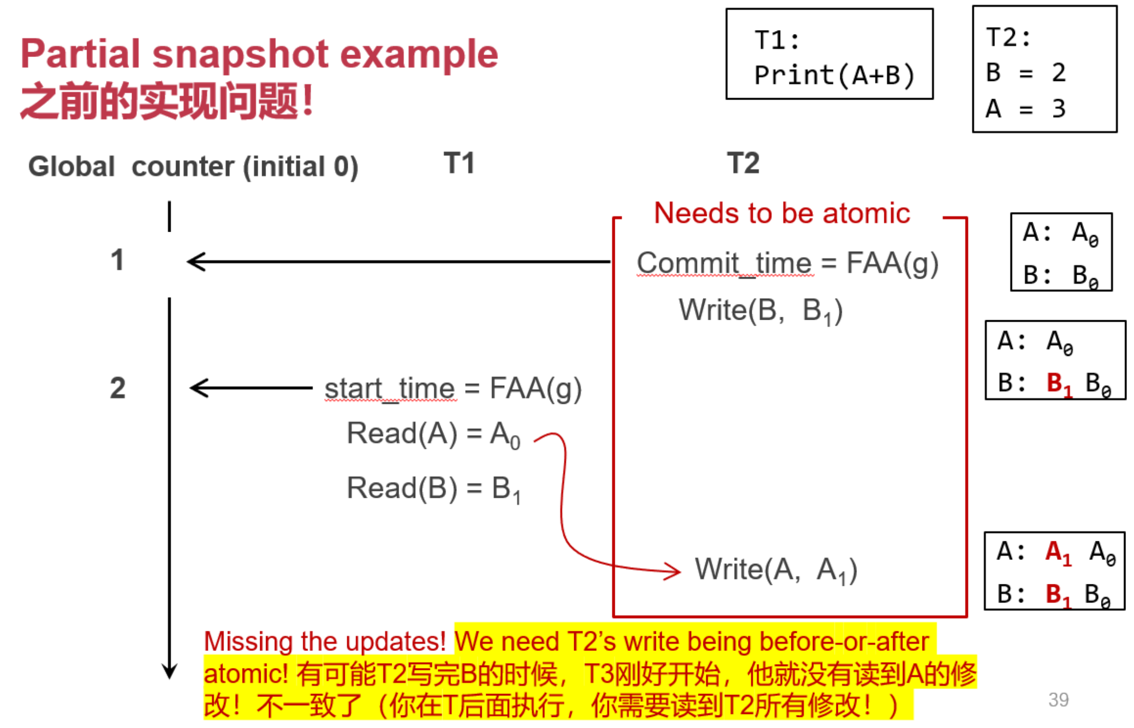
Partial snapshot

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

No validation needs here (for T1) !

Problem: partial updated snapshot



Partial snapshot example
之前的实现问题!

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

Global counter (initial 0) — T1 — T2

1 ← Commit_time = FAA(g)

Write(B, B_1)

2 ← start_time = FAA(g)

Read(A) = A_0

Read(B) = B_1

Write(A, A_1)

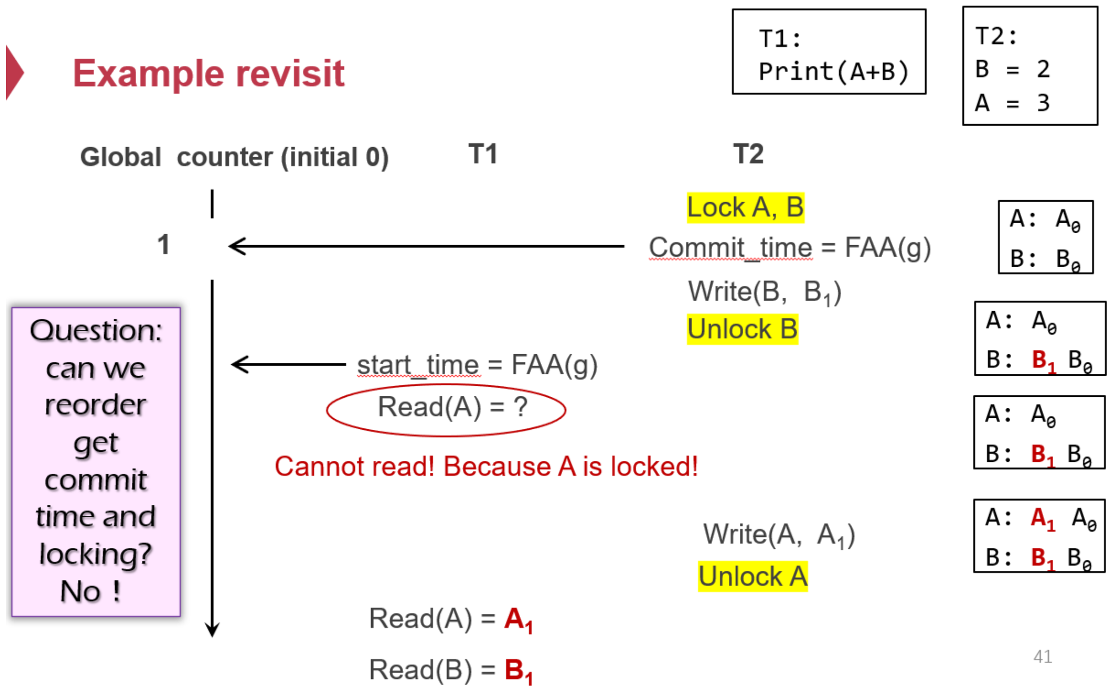
Needs to be atomic

Missing the updates! We need T2's write being before-or-after atomic! 有可能T2写完B的时候，T3刚好开始，他就没有读到A的修改！不一致了（你在T后面执行，你需要读到T2所有修改！）

4. 多版本并发控制：2.0

所以我们发现拿锁这件事似乎是不可避免的，我们OCC就在写，直到写完了我们才可以放掉，让别人来读。

read之前先check一下是否有人在写。



Example revisit

```
T1:
Print(A+B)
```

```
T2:
B = 2
A = 3
```

Global counter (initial 0) — T1 — T2

Lock A, B

1 ← Commit_time = FAA(g)

Write(B, B_1)

Unlock B

← start_time = FAA(g)

Read(A) = ?

Cannot read! Because A is locked!

Write(A, A_1)

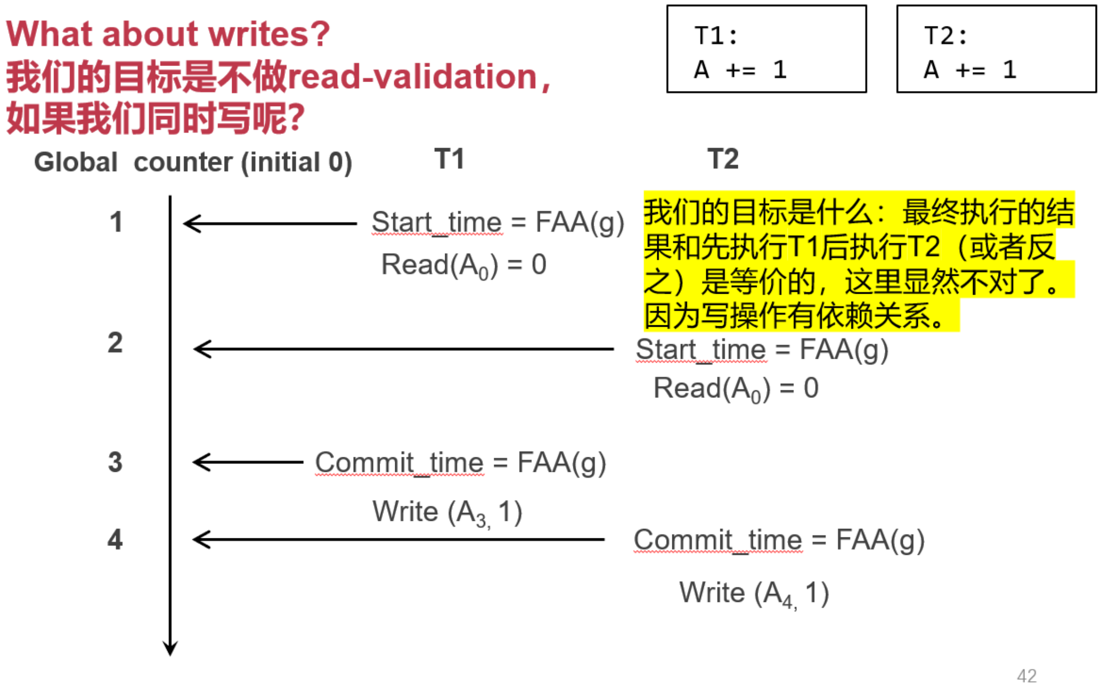
Unlock A

Read(A) = A_1

Read(B) = B_1

Question: can we reorder get commit time and locking? No !

我们之前这一套MVCC的目标是不做read_validation，目前已经可以保证了，但是如果我们同时要做多个write呢？



What about writes?
我们的目标是不做read-validation，
如果我们同时写呢？

```
T1:
A += 1
```

```
T2:
A += 1
```

Start_time = FAA(g)
Read(A_0) = 0

我们的目标是什么：最终执行的结果和先执行T1后执行T2（或者反之）是等价的，这里显然不对了。因为写操作有依赖关系。

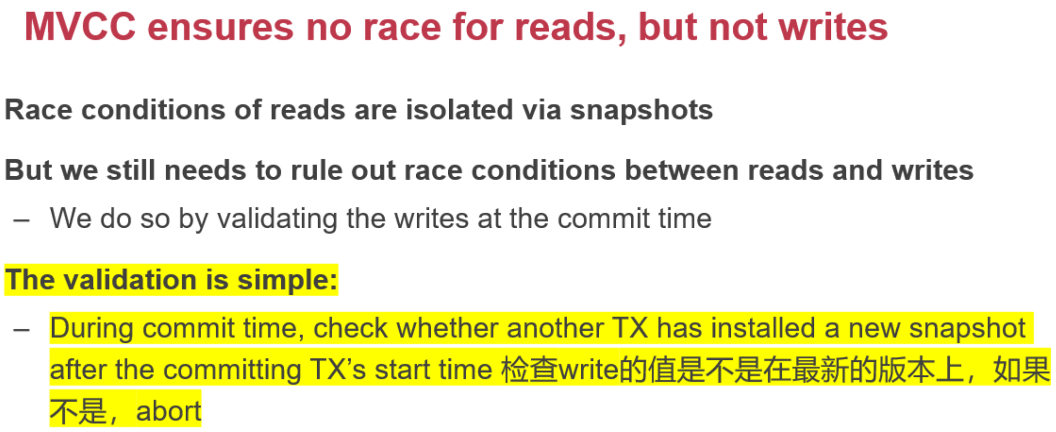
Start_time = FAA(g)
Read(A_0) = 0

Commit_time = FAA(g)
Write (A_3, 1)

Commit_time = FAA(g)
Write (A_4, 1)

42

当前我们并没有检查write-write的race condition。如果有多个写者，只能有第一个人成功。



MVCC ensures no race for reads, but not writes

Race conditions of reads are isolated via snapshots

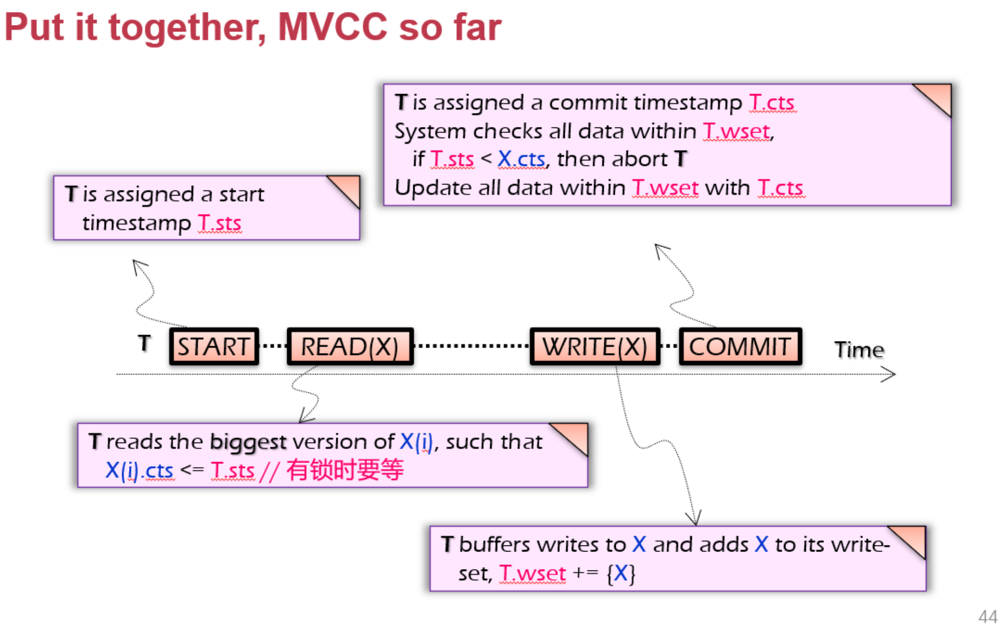
But we still needs to rule out race conditions between reads and writes

– We do so by validating the writes at the commit time

The validation is simple:

– During commit time, check whether another TX has installed a new snapshot after the committing TX's start time 检查write的值是不是在最新的版本上，如果不是，abort

4.3 How MVCC：完整的MVCC协议



Put it together, MVCC so far

T is assigned a commit timestamp T.cts
System checks all data within T.wset,
if T.sts < X.cts, then abort **T**
Update all data within T.wset with T.cts

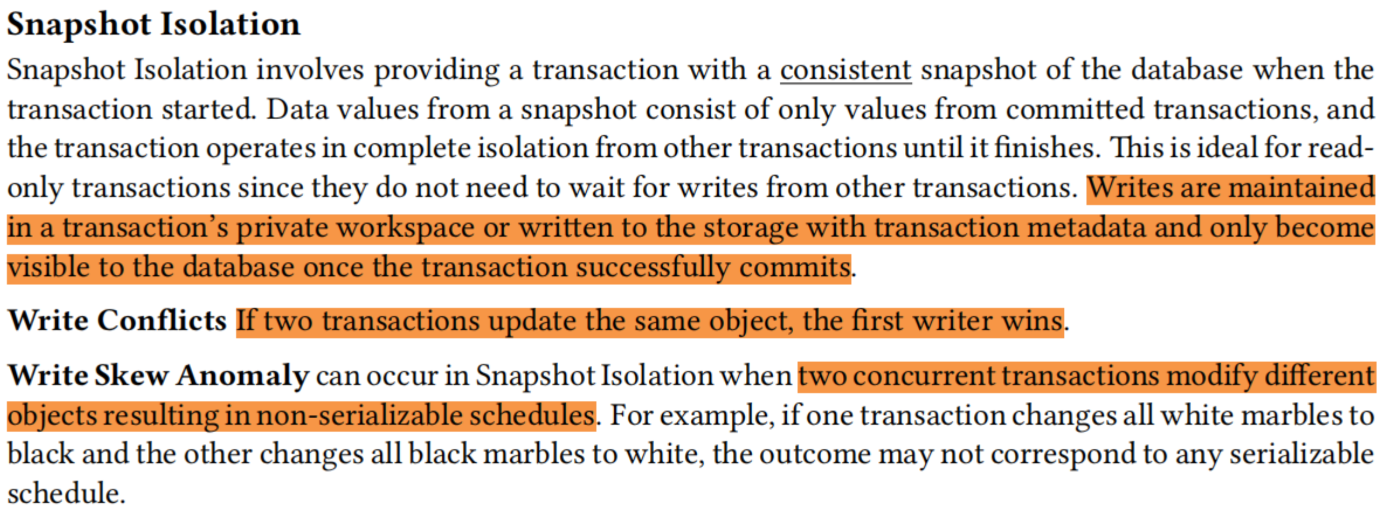
T is assigned a start timestamp T.sts

T reads the biggest version of X(i), such that X(i).cts <= T.sts // 有锁时要等

T buffers writes to X and adds X to its write-set, T.wset += {X}

44

1. 如果读的时候，去找当前版本最大的时间戳来读；如果当前版本时间戳被上锁了，需要等待
    a. 上锁需要等待的原因？因为可能有别的TX正在commit写数据库呢，如果你这个时候去读了，就有可能读到别人写了一半的东西，就没有办法做到before or after atomicity。
2. 写操作：
    a. 写操作就是把所有的write给buffer到write set里面，最后commit的时候统一处理。
3. Commit：
    a. Commit的时候，要去拿锁，来保证我们所有的写是atomic的，拿到锁之后，统一将所有的写落实到db中。【这里的锁是为了保证reader和writer之间的竞争】
    b. 然后我们其实还需要保证【writer和writer之间的竞争】。我们需要保证writer写的时候是基于最新版本的snapshot的。如果我们发现写者基于的snapshot并不是最新版本的，我们需要abort当前的事务。



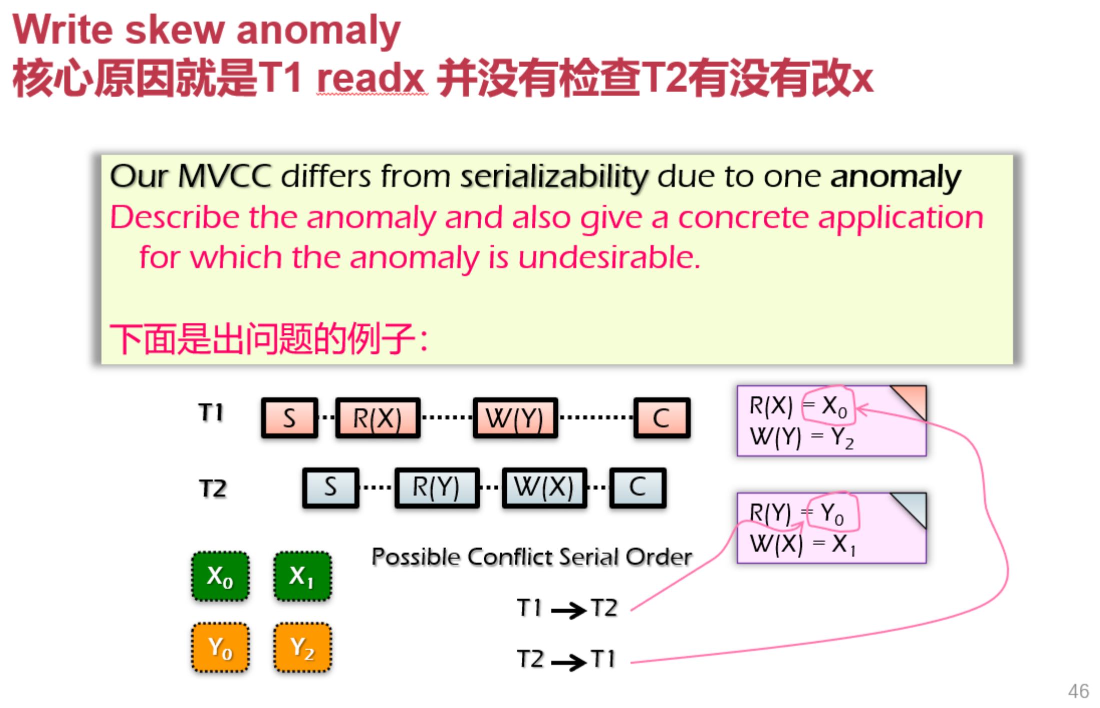
Snapshot Isolation

Snapshot Isolation involves providing a transaction with a consistent snapshot of the database when the transaction started. Data values from a snapshot consist of only values from committed transactions, and the transaction operates in complete isolation from other transactions until it finishes. This is ideal for read-only transactions since they do not need to wait for writes from other transactions. Writes are maintained in a transaction's private workspace or written to the storage with transaction metadata and only become visible to the database once the transaction successfully commits.

Write Conflicts If two transactions update the same object, the first writer wins.

Write Skew Anomaly can occur in Snapshot Isolation when two concurrent transactions modify different objects resulting in non-serializable schedules. For example, if one transaction changes all white marbles to black and the other changes all black marbles to white, the outcome may not correspond to any serializable schedule.

但是其实MVCC并不能保证before or after atomicity。下面就是反例：

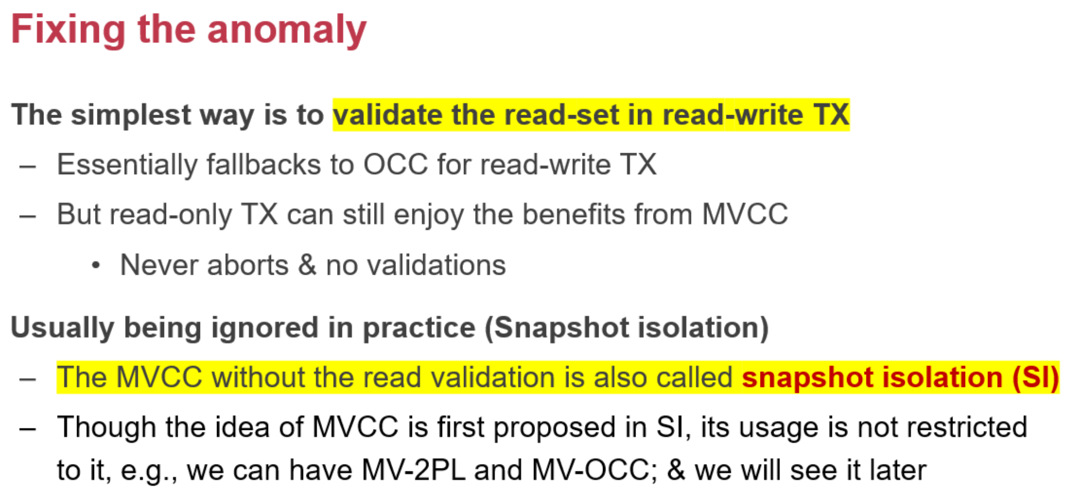


Write skew anomaly
核心原因就是T1 readx 并没有检查T2有没有改x

Our MVCC differs from serializability due to one anomaly
Describe the anomaly and also give a concrete application for which the anomaly is undesirable.

下面是出问题的例子：

fix的方法就是像occ一样，在read-write事务中，再去检查一下read set（在这种情况下，又回到occ了，但是在只读的事务上还是要好一些的）



Fixing the anomaly

The simplest way is to validate the read-set in read-write TX

- Essentially fallbacks to OCC for read-write TX
- But read-only TX can still enjoy the benefits from MVCC
    - Never aborts & no validations

Usually being ignored in practice (Snapshot isolation)

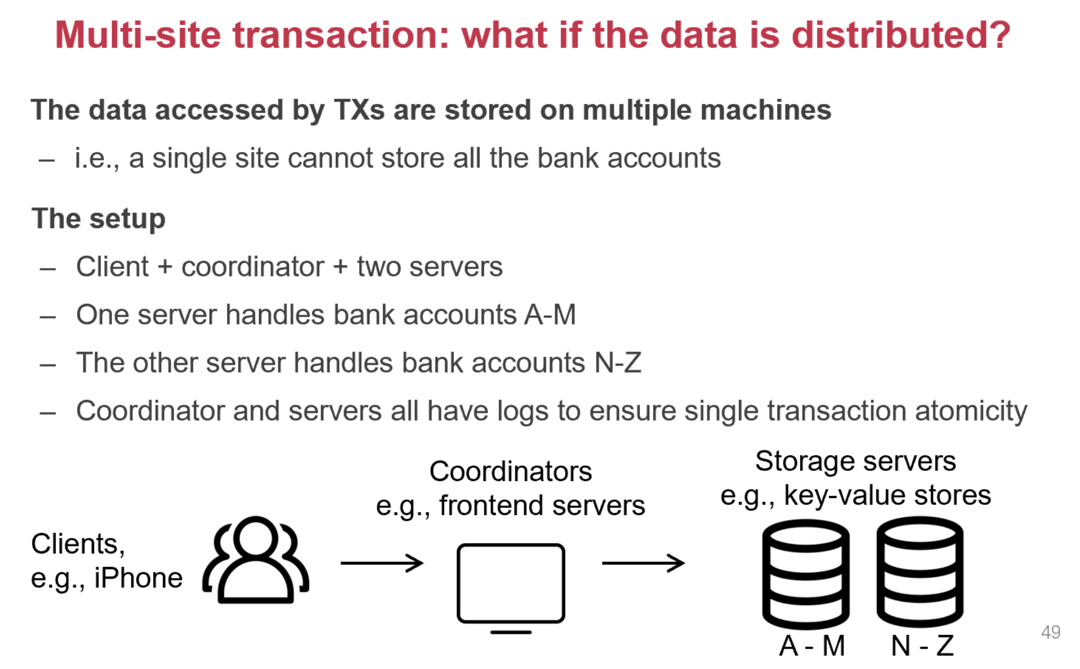
- The MVCC without the read validation is also called **snapshot isolation (SI)**
- Though the idea of MVCC is first proposed in SI, its usage is not restricted to it, e.g., we can have MV-2PL and MV-OCC; & we will see it later

但是实际上我们不太管这件事情，因为这事情发生得本来就不算很多。

现在主流的数据库声称它们做了2PL或者OCC，但是后面实际上基本都叠了MVCC来做进一步的优化。

5. Multi-site transaction && Multi-site atomicity 多机事务

我们假设下面的场景：比如我们有一个银行，一部分用户的信息存放在一台服务器上，另一部分用户的数据存放在另一台服务器上：



Multi-site transaction: what if the data is distributed?

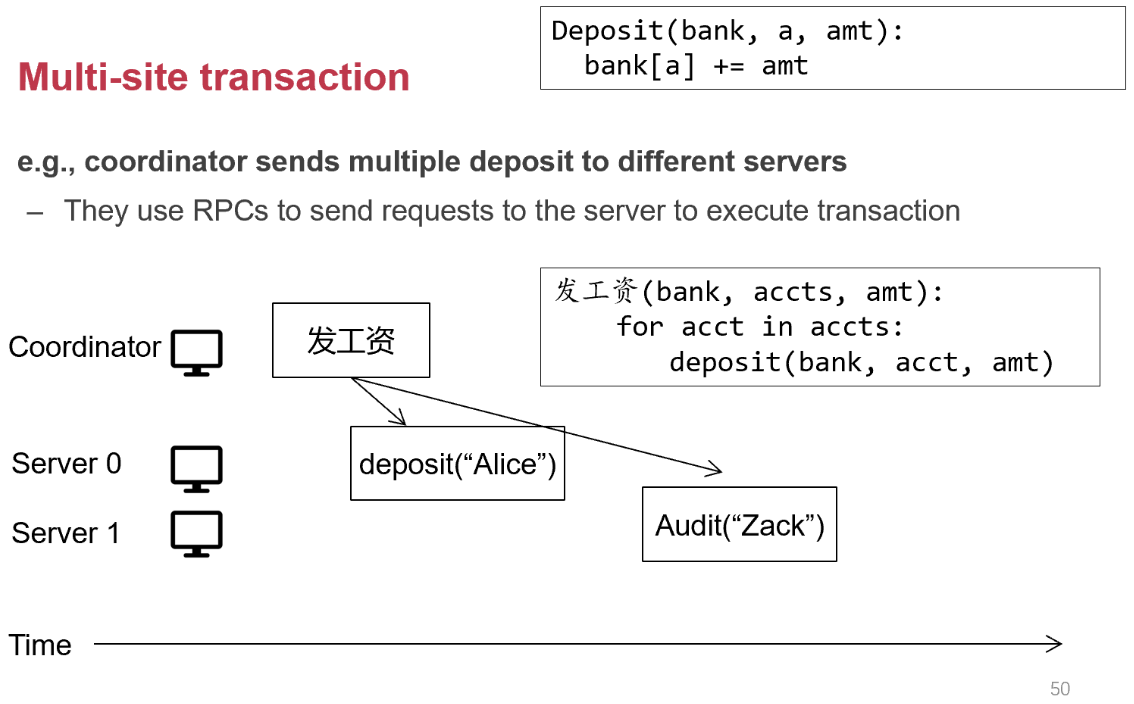
The data accessed by TXs are stored on multiple machines

- i.e., a single site cannot store all the bank accounts

The setup

- Client + coordinator + two servers
- One server handles bank accounts A-M
- The other server handles bank accounts N-Z
- Coordinator and servers all have logs to ensure single transaction atomicity

然后我们在中间叠了一个coordinator，来给不同的服务器发讯息：



Multi-site transaction

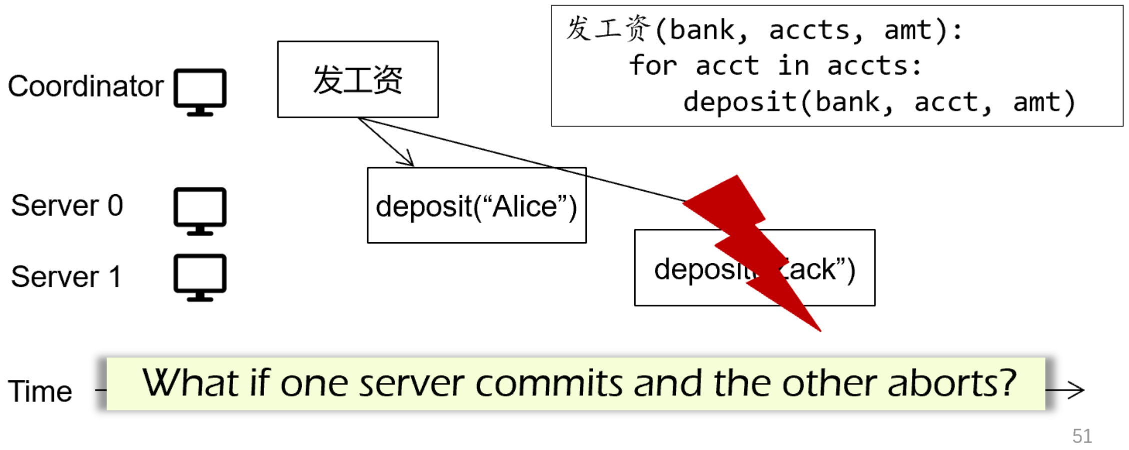
```
Deposit(bank, a, amt):
  bank[a] += amt
```

e.g., coordinator sends multiple deposit to different servers

- They use RPCs to send requests to the server to execute transaction

```
发工资(bank, accts, amt):
    for acct in accts:
        deposit(bank, acct, amt)
```

每台服务器上跑的各自给Alice和Zack发工资是事务，但这些小事务整体也组成了一个大的事务！整个大的事务也是需要保证all-or-nothing atomicity的：



```
发工资(bank, accts, amt):
    for acct in accts:
        deposit(bank, acct, amt)
```

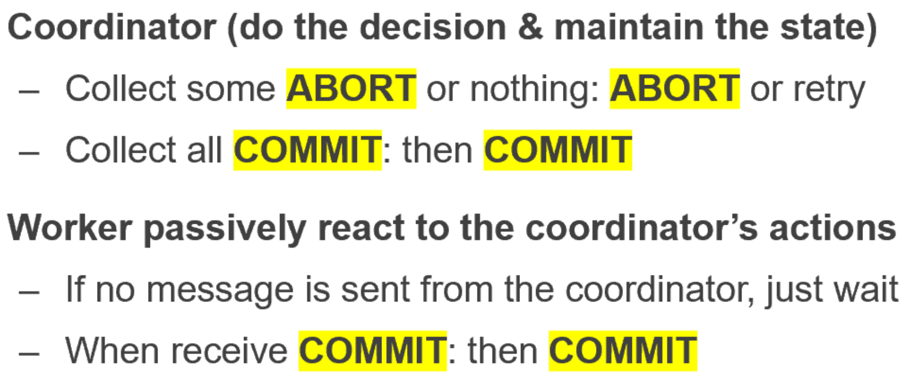
What if one server commits and the other aborts?

也就是不能出现有一个节点的事务commit了，另一个节点的事务却没有的情况！因此我们引入了两阶段提交协议。

5.1 两阶段提交协议

这里我们首先有两个原则：

- worker本身不做任何决定，完全follow coordinator
- 需要对重要的信息打log来做容错



Coordinator (do the decision & maintain the state)

– Collect some **ABORT** or nothing: **ABORT** or retry

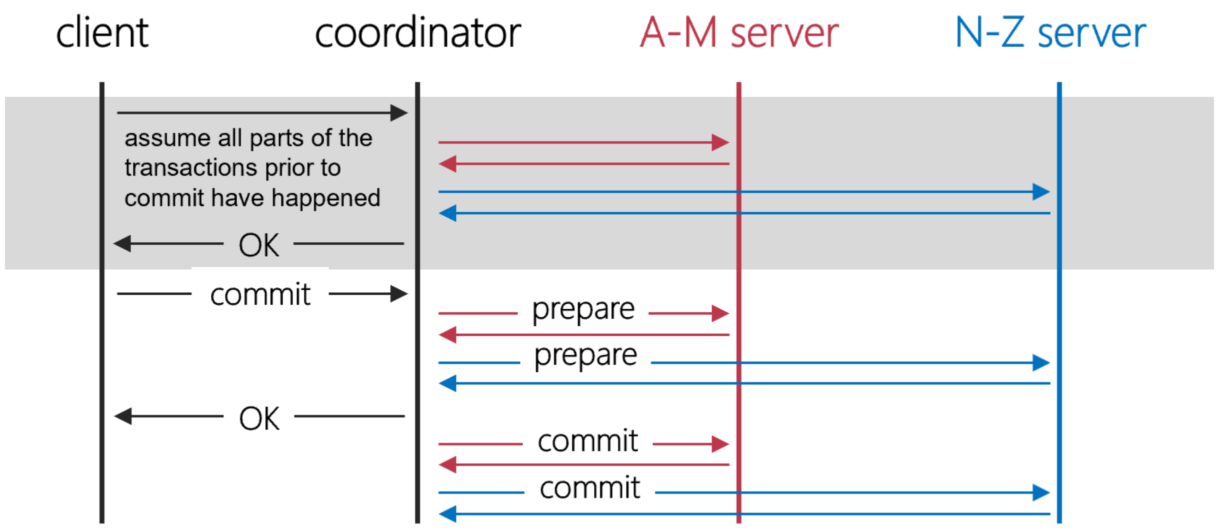
– Collect all **COMMIT**: then **COMMIT**

Worker passively react to the coordinator’s actions

– If no message is sent from the coordinator, just wait

– When receive **COMMIT**: then **COMMIT**

两阶段提交协议的核心就是，coordinator先去问一遍说大家是不是都同意提交这个事务？如果大家都同意了，那么coordinator就发出指令，大家提交整个事务：

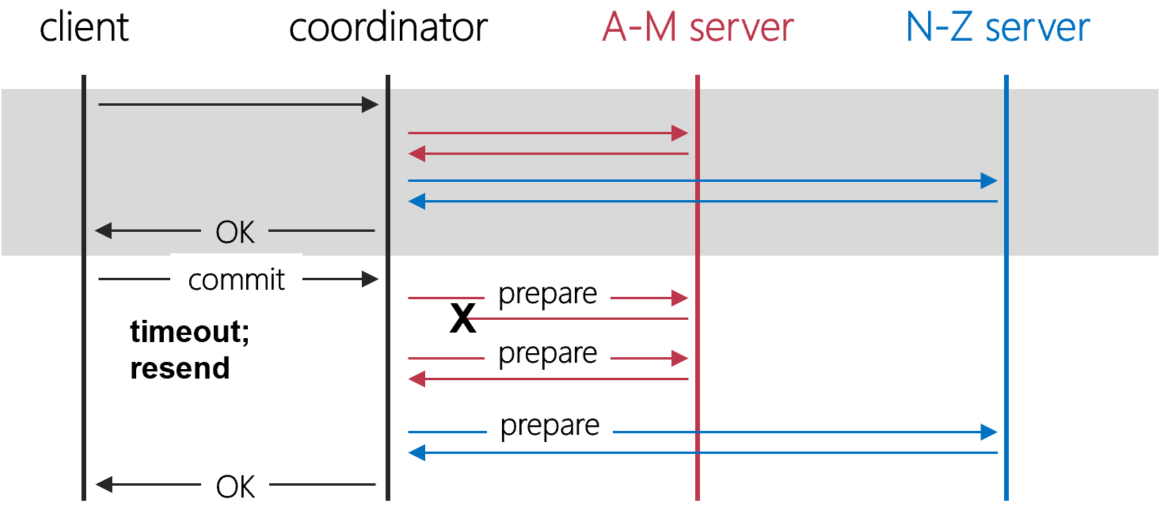


two-phase commit: nodes agree that they are ready to commit before committing

62

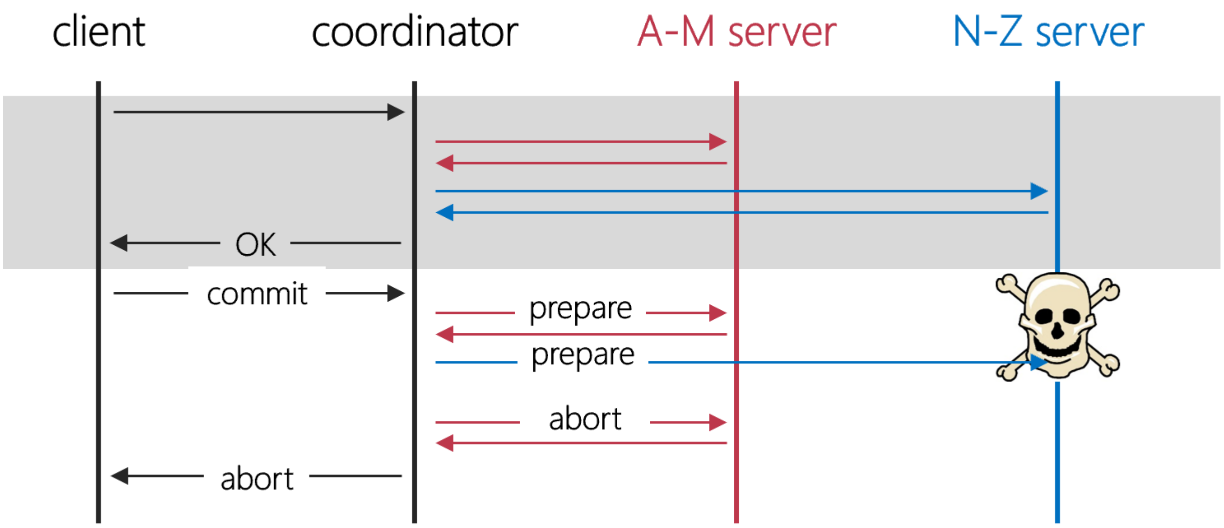
5.2 Case Study

coordinator和worker通信是通过RPC的，我们知道网络本身是不稳定的，同时服务器也有挂掉的可能。我们来看一些出错的情况。



failure: lost ACK for prepare

64



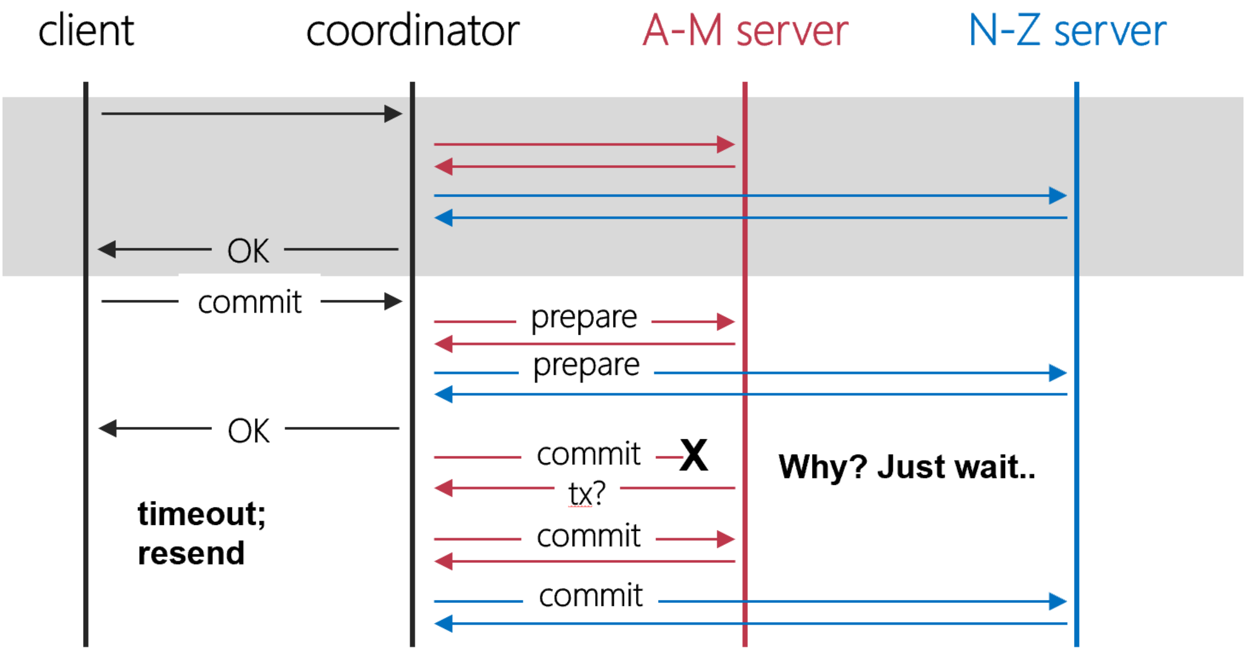
failure: worker failure during prepare

coordinator can safely abort transaction, will send explicit abort messages to live workers

65

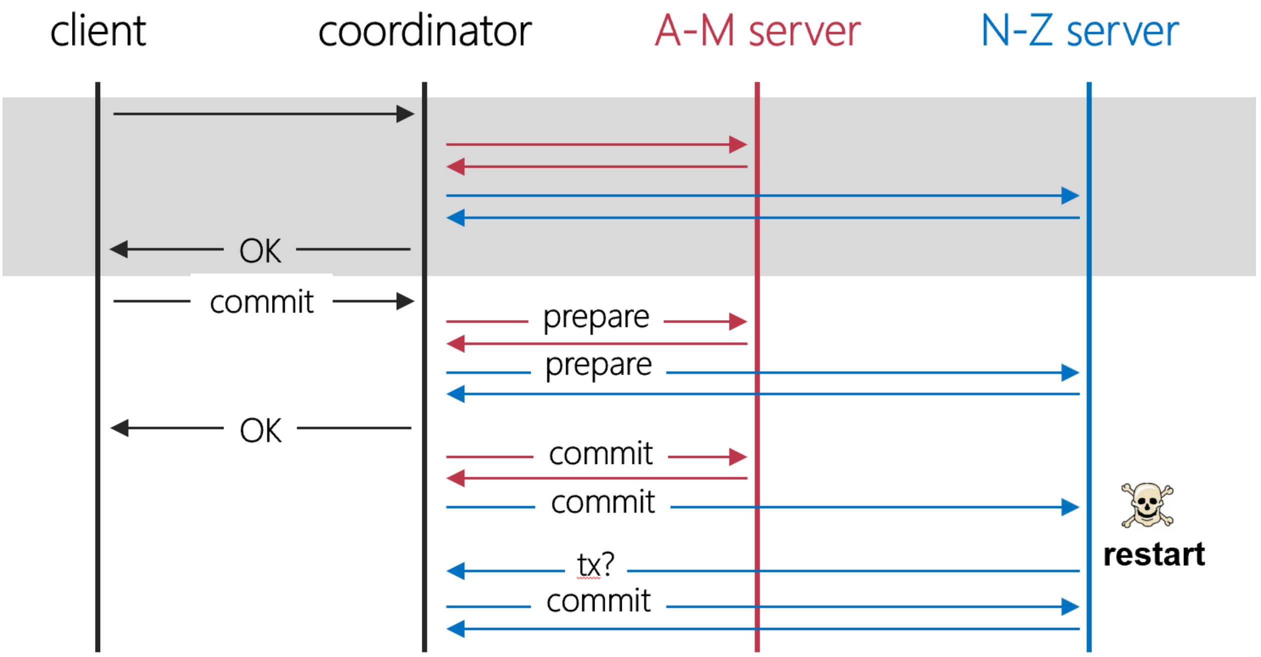
其实这里我认为，你站在coordinator的角度是完全无法判断究竟是worker死了还是网络故障没收到ACK，所以我觉得究竟是retry还是abort，就是一个可能根据时间来做的一个选择吧（更多基于经验，可能设置一个阈值时间？……）

当然，当coordinator一旦决定整个大事务提交，如果有worker死了或者一直没收到worker的ACK，就必须一直retry了！



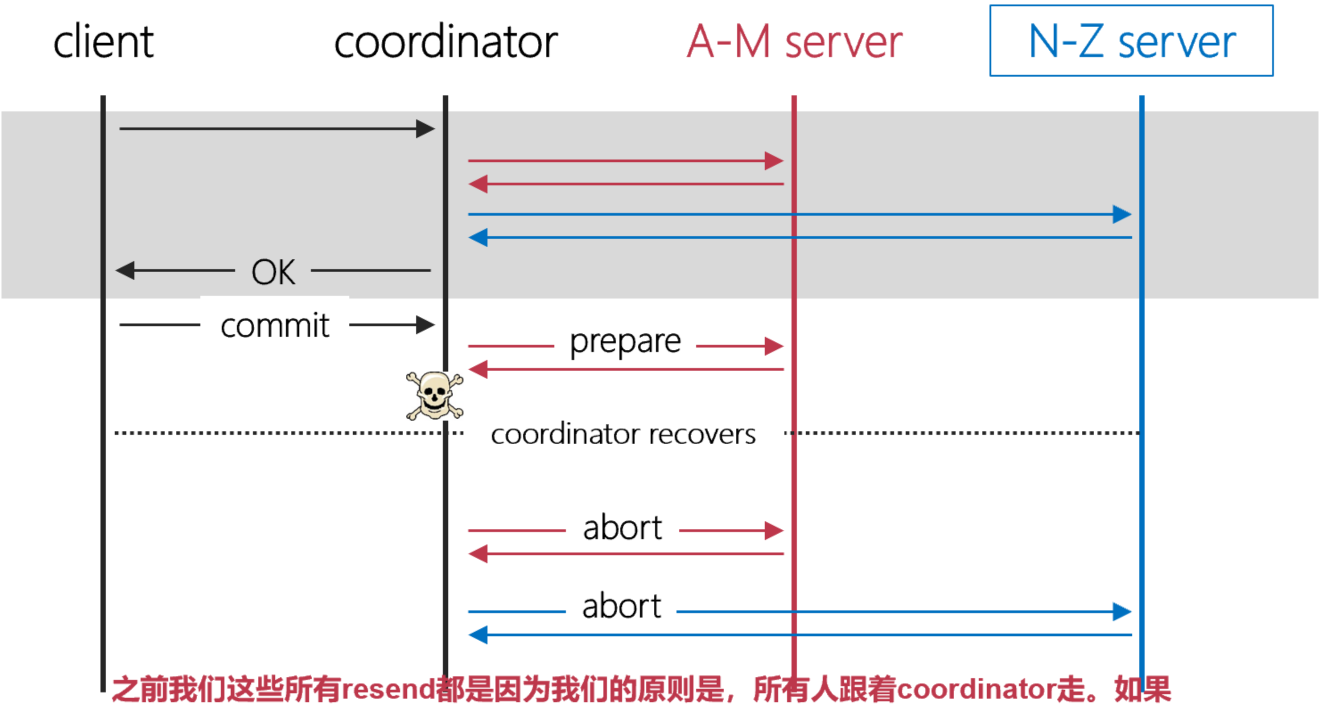
failure: lost commit message

66



failure: **worker failure during commit**

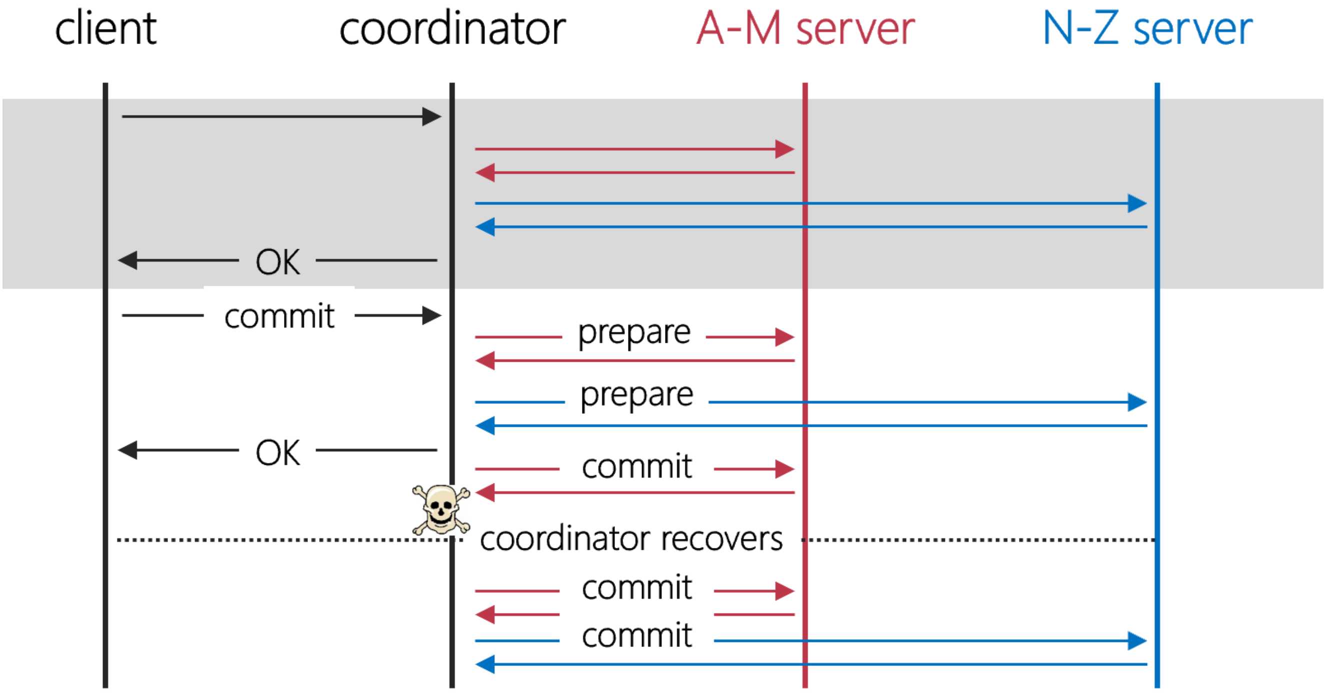
那如果coordinator挂了呢？我们就只能等它重启了！



之前我们这些所有resend都是因为我们的原则是，所有人跟着coordinator走。如果coordinator挂了呢？我必须等他重启！

failure: **coordinator failure during prepare**

所以coordintor必须做好logging！



failure: **coordinator failure during commit**

Must log the decision before sending the commit messages

5.3 【TODO】容错?

这里需要对我们之前提出的undo-redo logging做一点小小的维修。

6. Take away messages: Transactions in All

所以，说了这么多，到底什么是事务?

- 事务其实是一种**管理数据的方式**。

我们需要一个start和一个end来标识一个事务。

所以事务到底provide了什么特性?

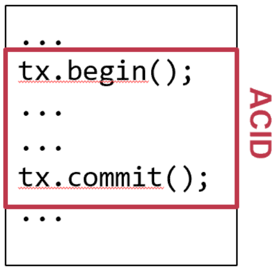
The program btw { `tx.begin()` & `tx.commit()` } has the following properties:

A: Atomicity

C: Consistency

I: Isolation

D: Durability



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

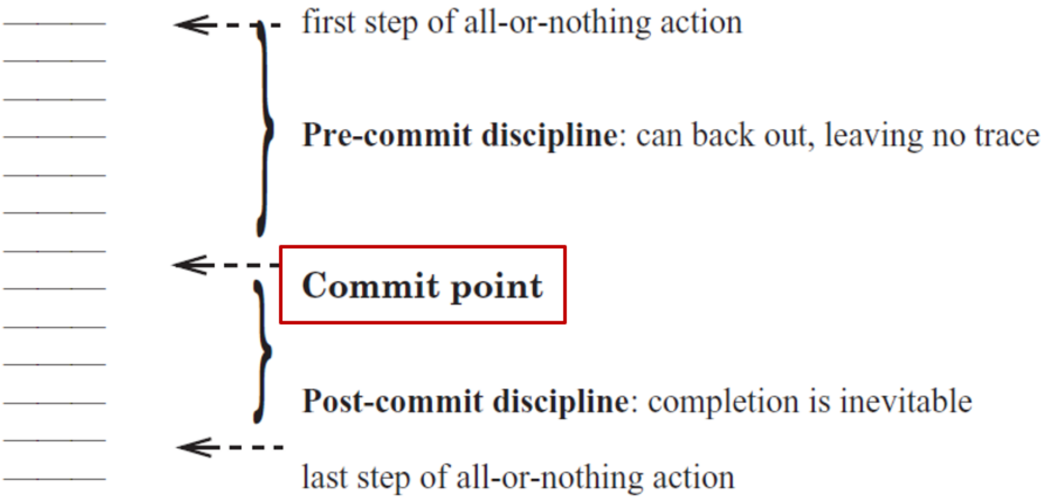
Application program

RELIABLE RELIABLE RELIABLE RELIABLE

RELIABLE

6.1 Atomicity

All-or-nothing atomicity。整个事务要么做完，要么不做。



first step of all-or-nothing action

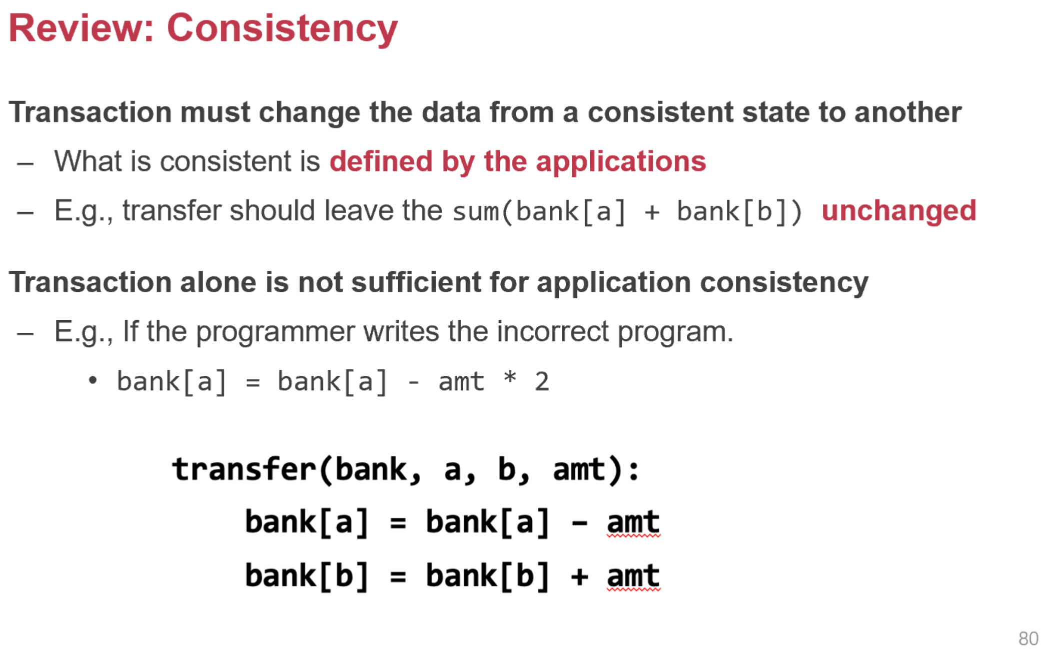
Pre-commit discipline: can back out, leaving no trace

Commit point

Post-commit discipline: completion is inevitable

last step of all-or-nothing action

6.2 Consistency



Review: Consistency

Transaction must change the data from a consistent state to another

– What is consistent is **defined by the applications**
– E.g., transfer should leave the `sum(bank[a] + bank[b])` **unchanged**

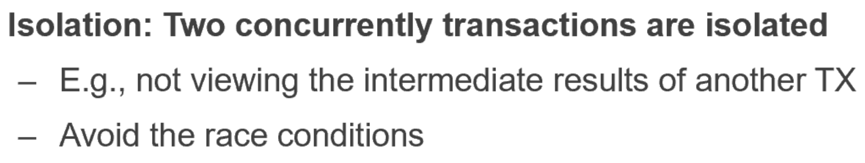
Transaction alone is not sufficient for application consistency

– E.g., If the programmer writes the incorrect program.
  - `bank[a] = bank[a] - amt * 2`

```
transfer(bank, a, b, amt):
    bank[a] = bank[a] - amt
    bank[b] = bank[b] + amt
```

6.3 Isolation

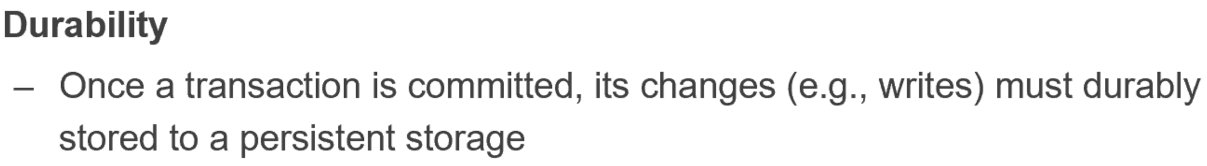
这里就涉及到before-or-after atomicity的内容了：



Isolation: Two concurrently transactions are isolated

– E.g., not viewing the intermediate results of another TX
– Avoid the race conditions

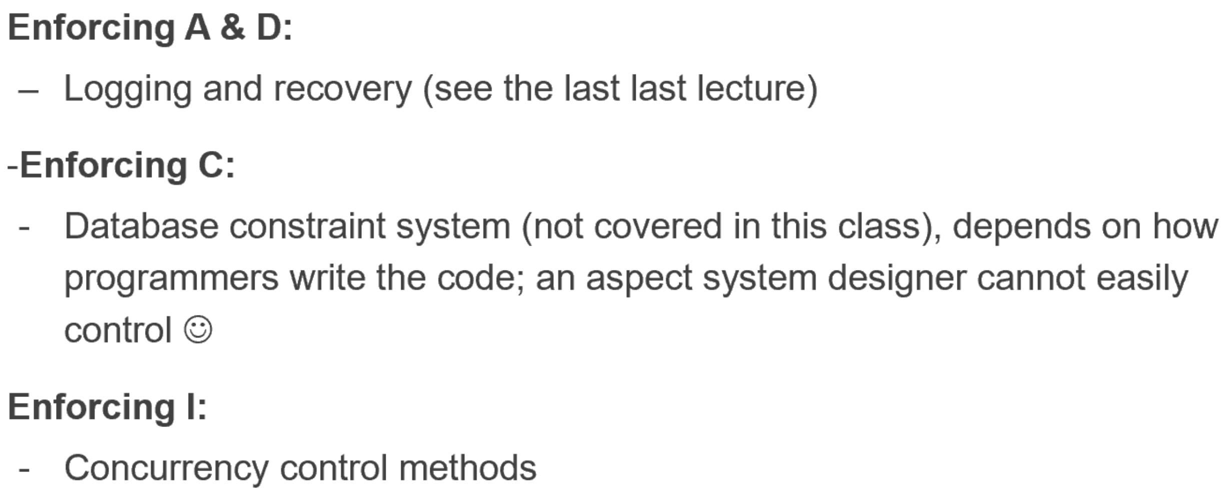
6.4 Durability



Durability

– Once a transaction is committed, its changes (e.g., writes) must durably stored to a persistent storage

6.5 实现



Enforcing A & D:

– Logging and recovery (see the last last lecture)

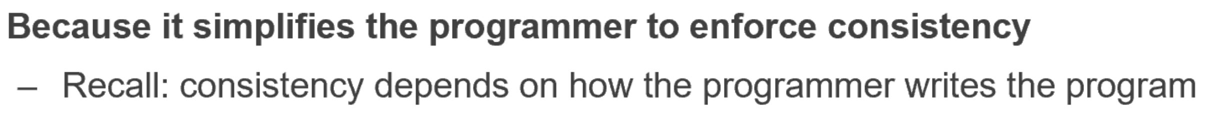
-**Enforcing C:**

- Database constraint system (not covered in this class), depends on how programmers write the code; an aspect system designer cannot easily control ☺

Enforcing I:

- Concurrency control methods

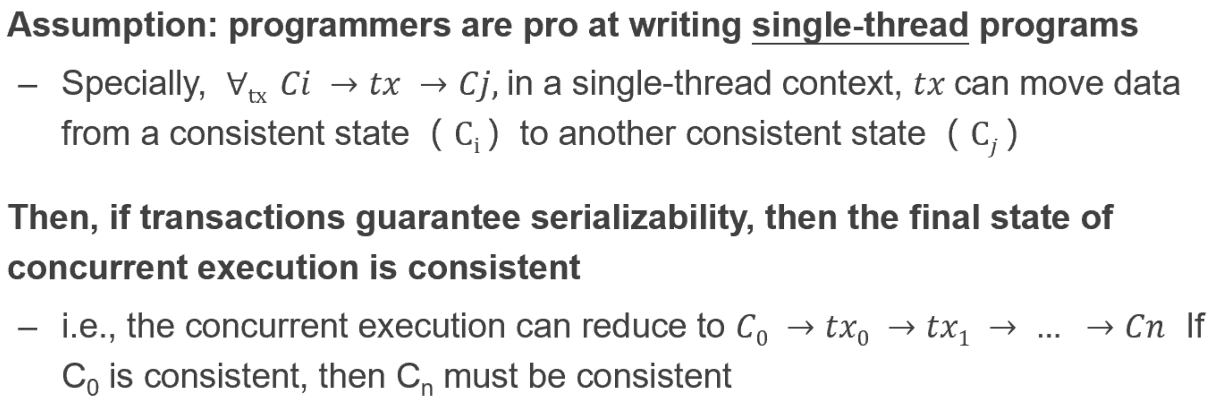
这里的C是值得说道说道的。consistency的定义本身就有点模糊，而且是否达到consistency一定程度上也取决于程序员怎么写代码。那么为什么我们一直在强调serializability呢？



Because it simplifies the programmer to enforce consistency

– Recall: consistency depends on how the programmer writes the program

程序员如果写出了一个巨烂的并发代码，这就很不好说了。所以我们假设程序员是可以写出没有错误的单线程代码的！



Assumption: programmers are pro at writing single-thread programs

– Specially, $\forall_{tx}\ Ci \rightarrow tx \rightarrow Cj$, in a single-thread context, tx can move data from a consistent state（C_i）to another consistent state（C_j）

Then, if transactions guarantee serializability, then the final state of concurrent execution is consistent

– i.e., the concurrent execution can reduce to $C_0 \rightarrow tx_0 \rightarrow tx_1 \rightarrow \ldots \rightarrow Cn$ If C_0 is consistent, then C_n must be consistent

所以只要并发事务执行是可以被serialize的，整个问题就迎刃而解了！