

Concurrency Control

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Transaction so far

All-or-nothing atomicity in transactions

- Failures do not result in (undesirable) intermediate state

How to realize all-or-nothing atomicity?

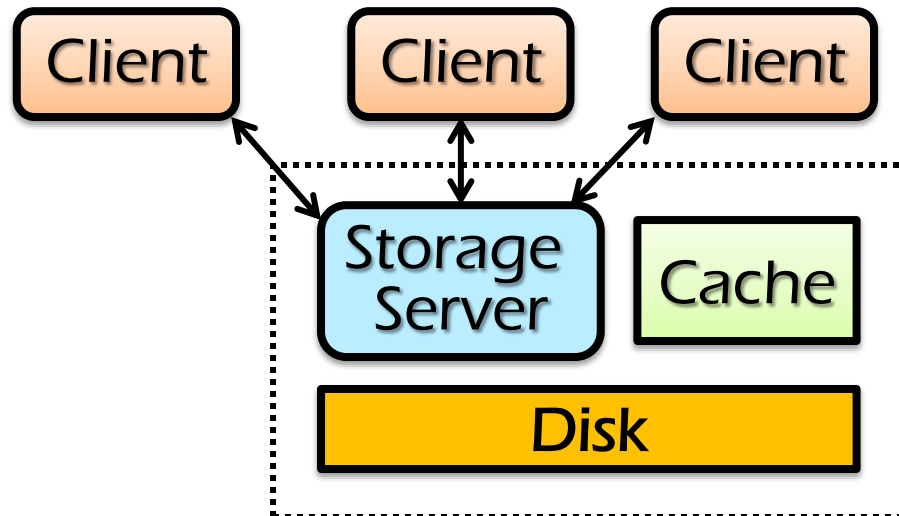
- Logging and recovery
- **DO-REDO-UNDO** protocol
- **REDO/UNDO** vs. **REDO-only**

How about concurrent transactions?

Concurrency Control

Concurrent transactions

- Multiple application clients are concurrently accessing a storage system
- Transactions might interfere
- Similar to **data race** in parallel programs



Example

Client Transactions1

Transfer \$1000 from A to B

```
A_bal = READ(A)
if (A_bal > 1000) {
  B_bal = READ(B)
  B_bal += 1000
  A_bal -= 1000
  WRITE(A, A_bal)
  WRITE(B, B_bal)
}
```

Client Transactions2

Report SUM of A and B

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
```

Storage Server

```
graph TD
    TS1[Client Transactions1] --> T1[Transfer $1000 from A to B]
    T1 --> C1[A_bal = READ(A)  
if (A_bal > 1000) {  
  B_bal = READ(B)  
  B_bal += 1000  
  A_bal -= 1000  
  WRITE(A, A_bal)  
  WRITE(B, B_bal)  
}]
    TS2[Client Transactions2] --> T2[Report SUM of A and B]
    T2 --> C2[A_bal = READ(A)  
B_bal = READ(B)  
Print(A_bal + B_bal)]
    C1 -.-> SS[Storage Server]
    C2 -.-> SS
```

Example

Client Transactions1

Transfer \$1000 from A to B

```
A_bal = READ(A)
if (A_bal > 1000) {
  B_bal = READ(B)
  B_bal += 1000
  A_bal -= 1000
  WRITE(A, A_bal)
  WRITE(B, B_bal)
}
```

Client Transactions2

Withdraw \$500 from A

```
A_bal = READ(A)
If (A_bal > 500) {
  A_bal -= 500
  WRITE(A_bal)
  dispense $500 to user
}
```

Storage Server

```
graph TD
    T1[Client Transactions1] --> S[Storage Server]
    T2[Client Transactions2] --> S
```

Potential Solutions

Leave it to **application** programmers

- Application programmers are responsible for protecting (e.g., lock) data items

Storage system performs automatic concurrency control

- Concurrent transactions execute as if **serially**

ACID Transactions

A (Atomicity)

All-or-nothing with respect to failures

C (Consistency)

Transactions maintain any internal state **invariants**

I (Isolation)

Concurrently executing transactions don't **interfere**

D (Durability)

Effect to transactions **survive** failures

ACID Transactions

A (Atomicity)

All-or-nothing with respect to failures

C (Consistency)

Transactions maintain any internal state **invariants**

I (Isolation)

Concurrently executing transactions don't **interfere**

D (Durability)

Effect to transactions **survive** failures

What is ACID ?

T1 Transfer \$1000 from A to B

T2 Transfer \$500 from B to A

A

T1 **completes** or **nothing** (ditto for T2)

C

Preserves **invariants**, e.g. account balance > 0

I

No **race**, as if T1 happens either before or after T2

D

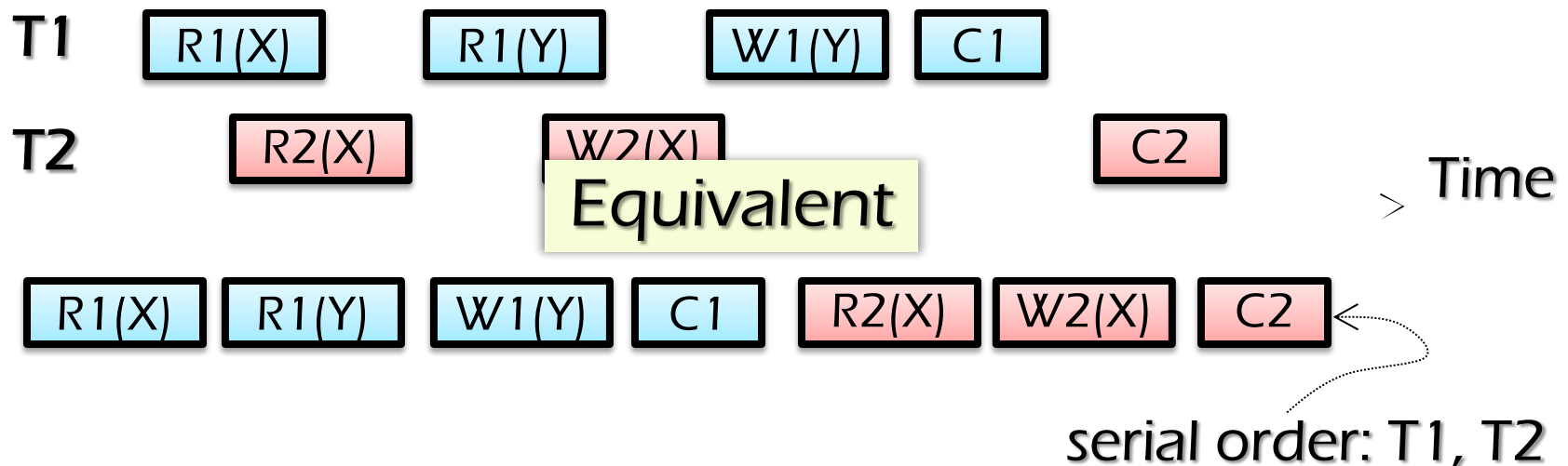
Once T1/T2 commits, stays done, no updates **lost**

Ideal Semantics: Serializability

Definition: execution of a set of transactions is **equivalent** to some **serial** order

□ Two executions are **equivalent**

→ if they have the same effect on database and produce same output



Execution Schedule

An execution schedule is the ordering of **READ/WRITE/COMMIT/ABORT** ops

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 1000
A_bal -= 1000
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
Commit
```

A (serial) schedule



Conflict Serializability

Two schedules are **equivalent** →

1. Contain **same** operations
2. Order **conflicting** operations the same way

Two operations **conflict** → they access the same data item and at least one is a write

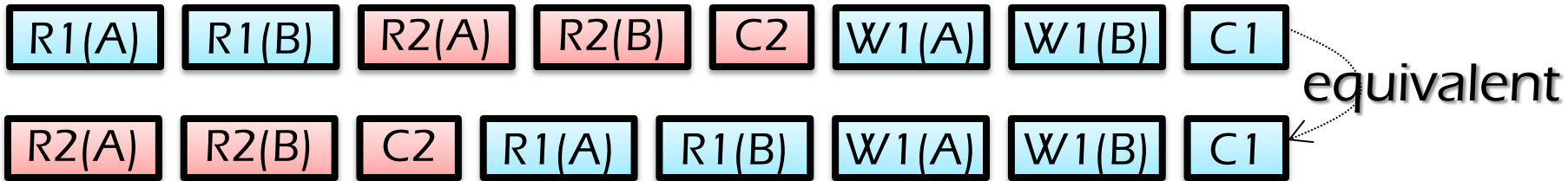
A schedule is **serializable** if it's equivalent to **some** serial schedule

Examples

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 1000
A_bal -= 1000
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
Commit
```

Serializable?

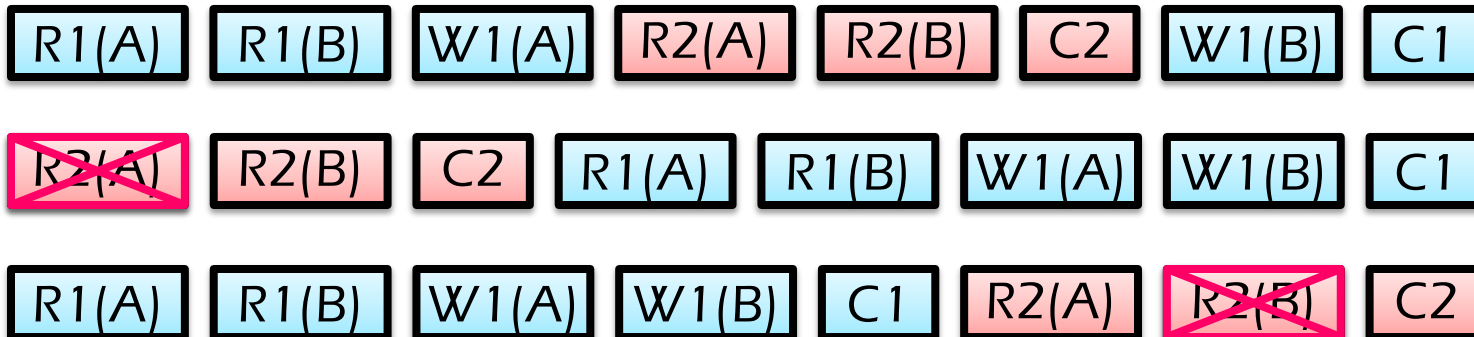


Examples

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 1000
A_bal -= 1000
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
Commit
```

Serializable?



Serializable Schedule

Locking based approach

Strawman solution 1:

- Grab global lock before transaction **START**
- Release global lock after transaction **COMMIT**

A **serial** schedule → Too Slow!!

Strawman solution 2:

Serializable or not ?

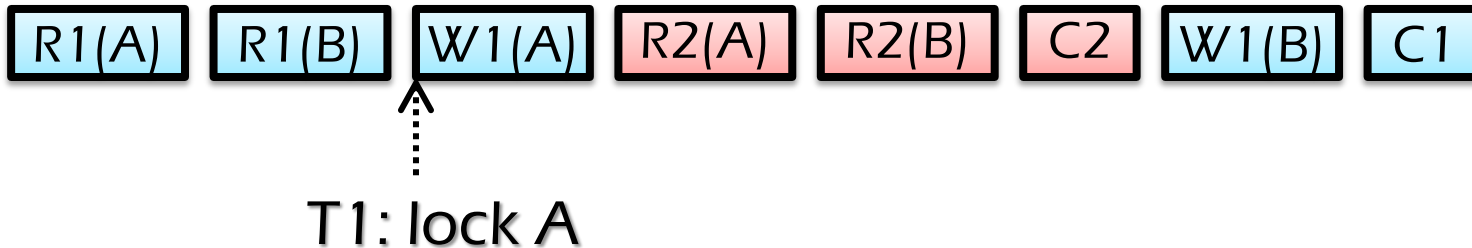
- Grab lock on item X before **READ/WRITE** X
- Release lock on X after **READ/WRITE** X

Strawman 2's Example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 1000
A_bal -= 1000
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
Commit
```

Is it possible?

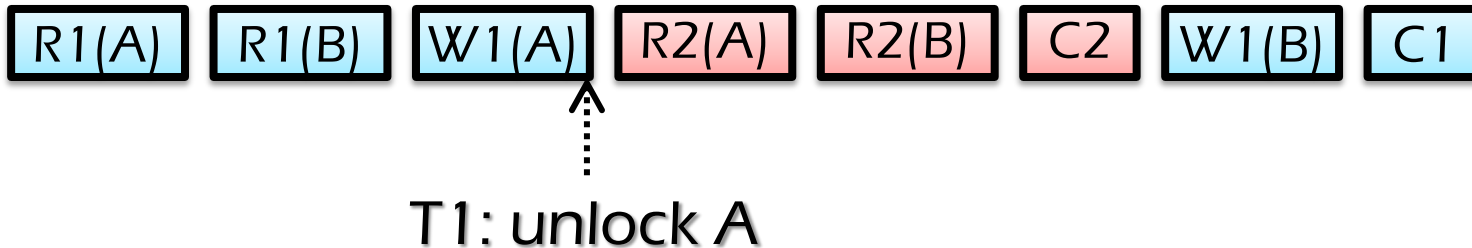


Strawman 2's Example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 1000
A_bal -= 1000
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
Commit
```

Short-duration locks on **WRITES**

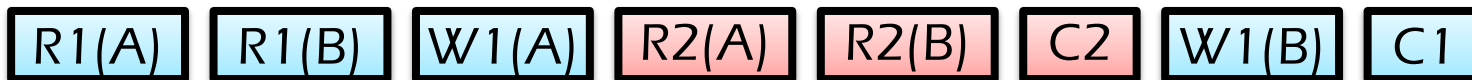


Strawman 2's Example

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 1000
A_bal -= 1000
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal + B_bal)
Commit
```

Short-duration locks on **WRITES**



T2: lock A

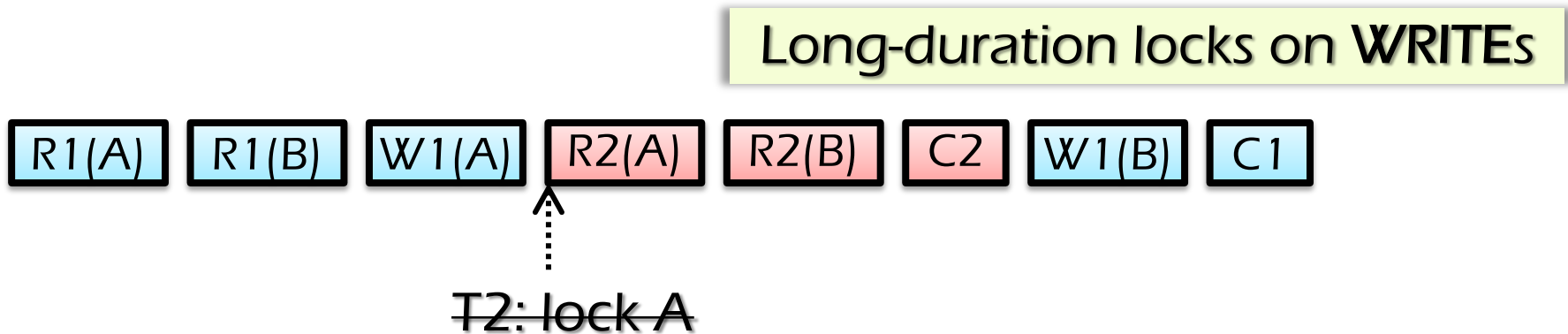
Read an **uncommitted** value?

Locks on **WRITES** should be held till end of transaction

Serializable Schedule

Strawman solution 3:

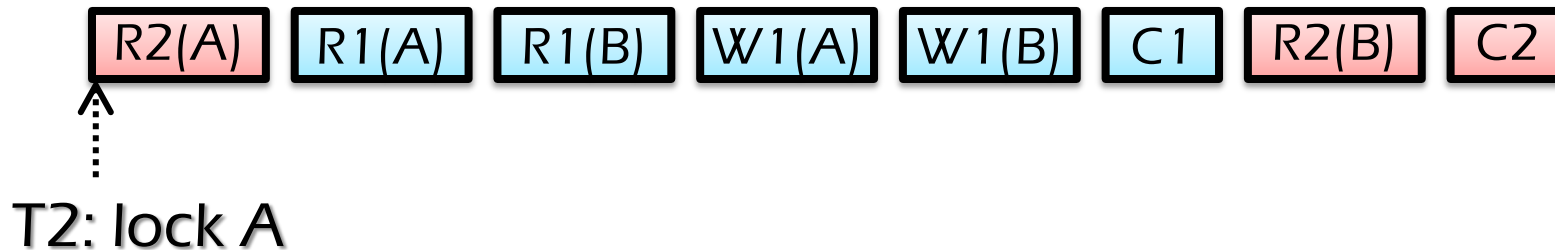
- Grab write lock on item X before **WRITE**
- Release write lock after transaction **COMMIT**
- Grab read lock on item X before **READ**
- Release read lock on X immediately **READ**



Serializable Schedule

Strawman solution 3:

- Grab write lock on item X before **WRITE**
- Release write lock after transaction **COMMIT**
- Grab read lock on item X before **READ**
- Release read lock on X immediately **READ**



Serializable Schedule

Strawman solution 3:

- Grab write lock on item X before **WRITE**
- Release write lock after transaction **COMMIT**
- Grab read lock on item X before **READ**
- Release read lock on X immediately **READ**

Short-duration locks on READs



T2: unlock A

Serializable Schedule

Strawman solution 3:

- Grab write lock on item X before **WRITE**
- Release write lock after transaction **COMMIT**
- Grab read lock on item X before **READ**
- Release read lock on X immediately **READ**

Short-duration locks on READs



T1: lock A

Non-repeatable read?

Locks on **READ** should be held till end of transaction

Serializable Schedule

Two-phase locking (**2PL**)

- A **growing** phase in which the transaction is acquiring locks
- A **shrinking** phase in which locks are released

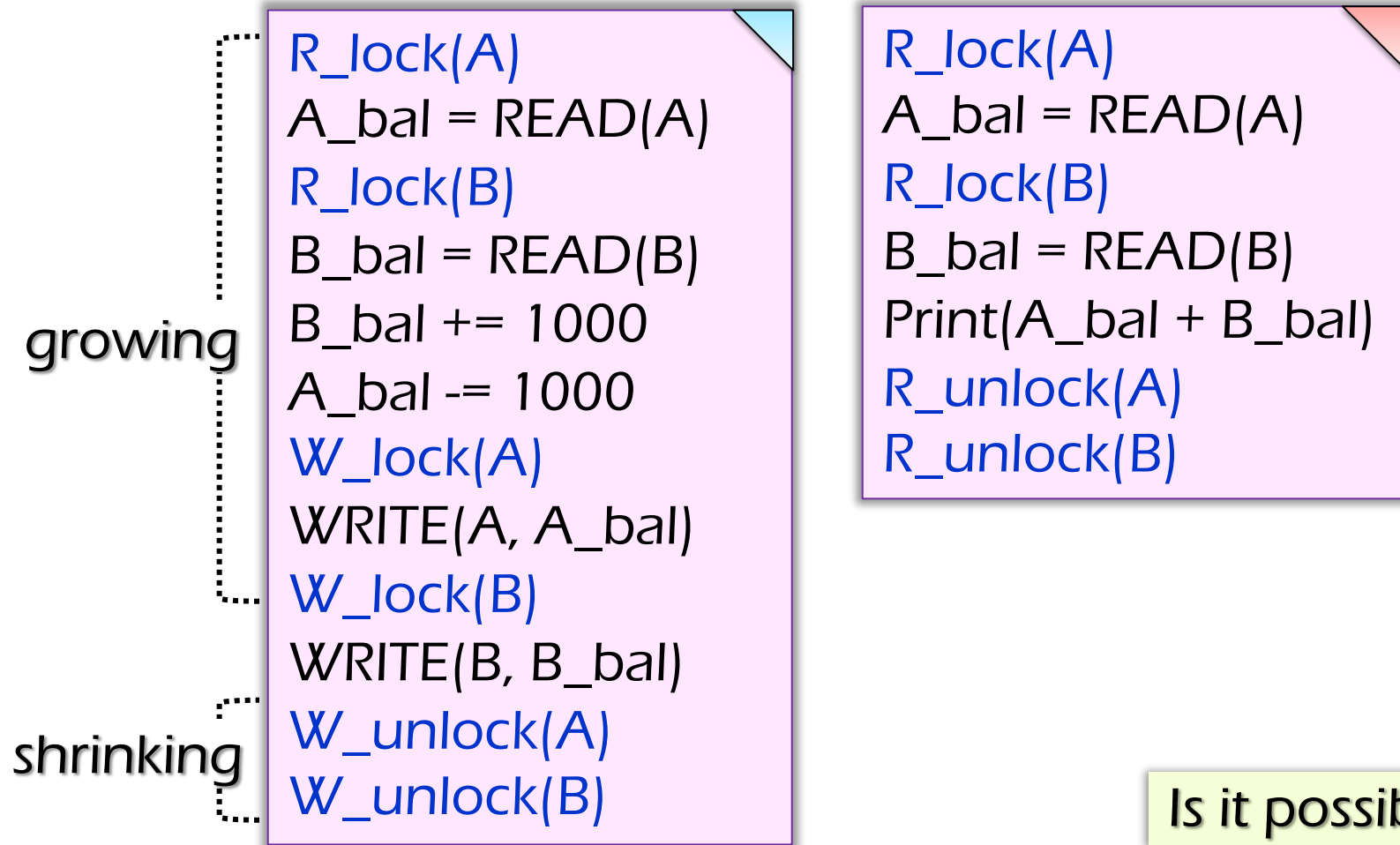
In practice

- The **growing** phase is the **entire** transaction
- The **shrinking** phase is at the **COMMIT** time

Optimization

- Use read/write locks instead of exclusive locks

Two-phase Locking's Example



R1(A) R1(B) W1(A) R2(A) R2(B) C2 W1(B) C1

Two-phase Locking

Q1: What if a lock is **unavailable**?

Q2: **Deadlocks** possible?

How to cope with **deadlock**?

- **Avoidance** → grab locks in order

Not always possible

- **Detection** → detects deadlock cycles (timeout) and **aborts** affected transactions (self)



Crash recovery mechanism
(e.g., UNDO) to clean up

Phantom Problem

T1 Update emp (set salary=1.1 * salary) where dept='SS'

Update Binyu
Update Haibo
Update Xvjia
Update Yubin
Commit

T2 Update emp (set dept='SS') where emp.name='Rong' or "Zhaoguo"

Move Rong
Move Zhaoguo
Commit (release all locks)

R1(P) W1(S.B) W1(S.H) W2(P.R) W2(P.X) C2 W1(S.S) W1(S.Y) C1

Phantom Problem

Issue: only lock things you touch

- But must guarantee non-existence of any new ones!

Two operations conflict → they access the same **data item** and at least one is a write

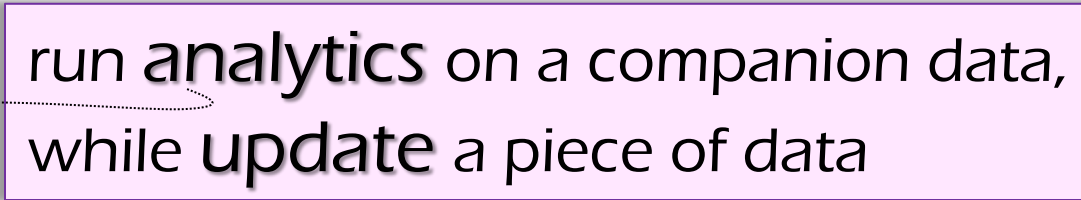
Solutions

→ e.g., object, row, list, entire table, ...

- Predicate locking
- Range locks in a B-tree index (e.g., assumes an index on **dept**), or lock entire table
- Often ignore in practice

Why **less** than Serializability?

Performance

- Long-duration locks on all **READ/WRITE**
- Scenarios: 
long running read-only transaction

Disadvantages of locking-based approach

- Need distributed **deadlock** detection
- Read-only transactions (e.g., *data mining*) can block update transactions

Multi-Version Concurrency Control

No locking during transaction execution

Each data item has **multiple** versions

Multi-version transactions:

- Buffer **WRITEs** during execution
- **READs** choose the appropriate version
- At **COMMIT** time, system validates if OK to make writes visible (generating **new** versions)

Snapshot Isolation

A popular multi-version concurrency control (MVCC) scheme

Transactions:

- **WRITEs** a local buffer
- **READs** a “snapshot” of entire data image
- **COMMIT** only if no write-write conflict

Implementation

T is assigned a start timestamp **T.sts**

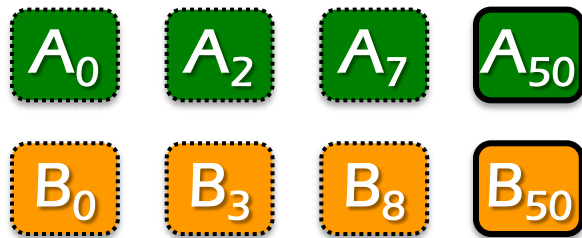
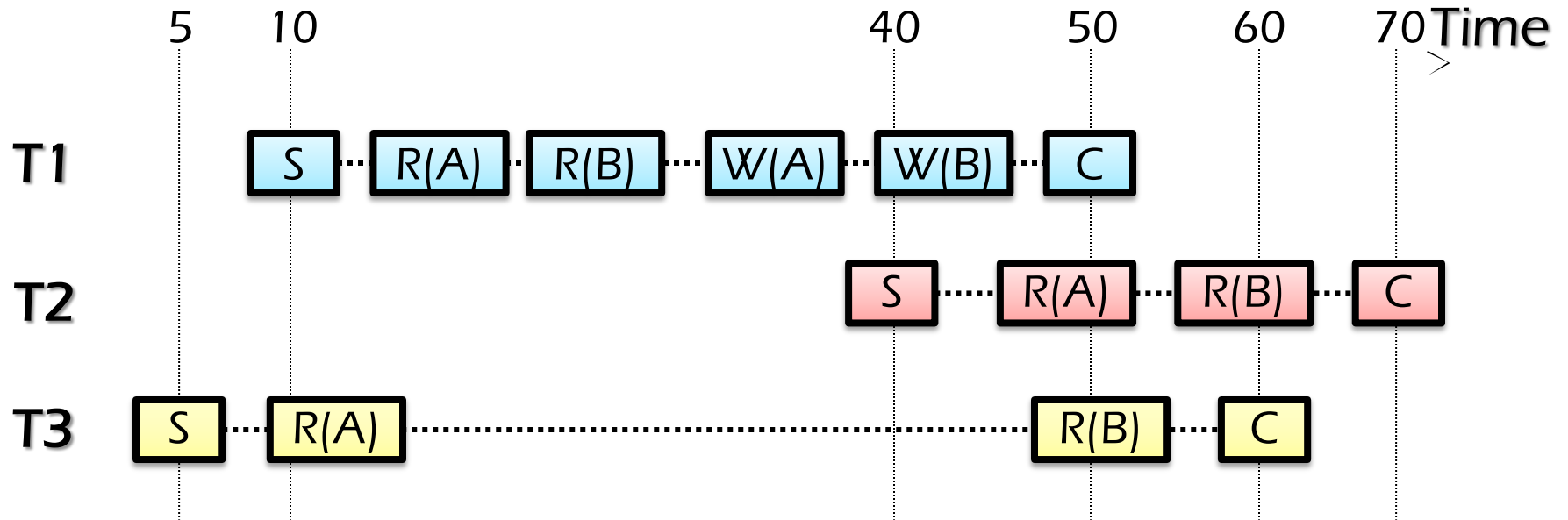
T is assigned a commit timestamp **T.cts**
System checks all data within **T.wset**,
if **T.sts** < **X.cts** < **T.cts**, then abort **T**
Update all data within **T.wset** with **T.cts**



T reads the biggest version of **X(i)**,
such that **X(i).cts** ≤ **T.sts**

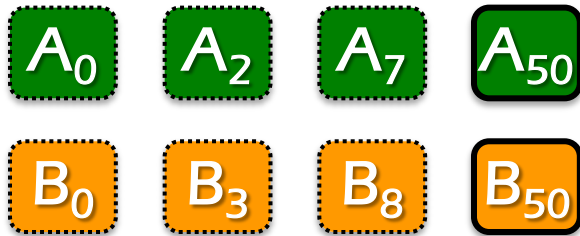
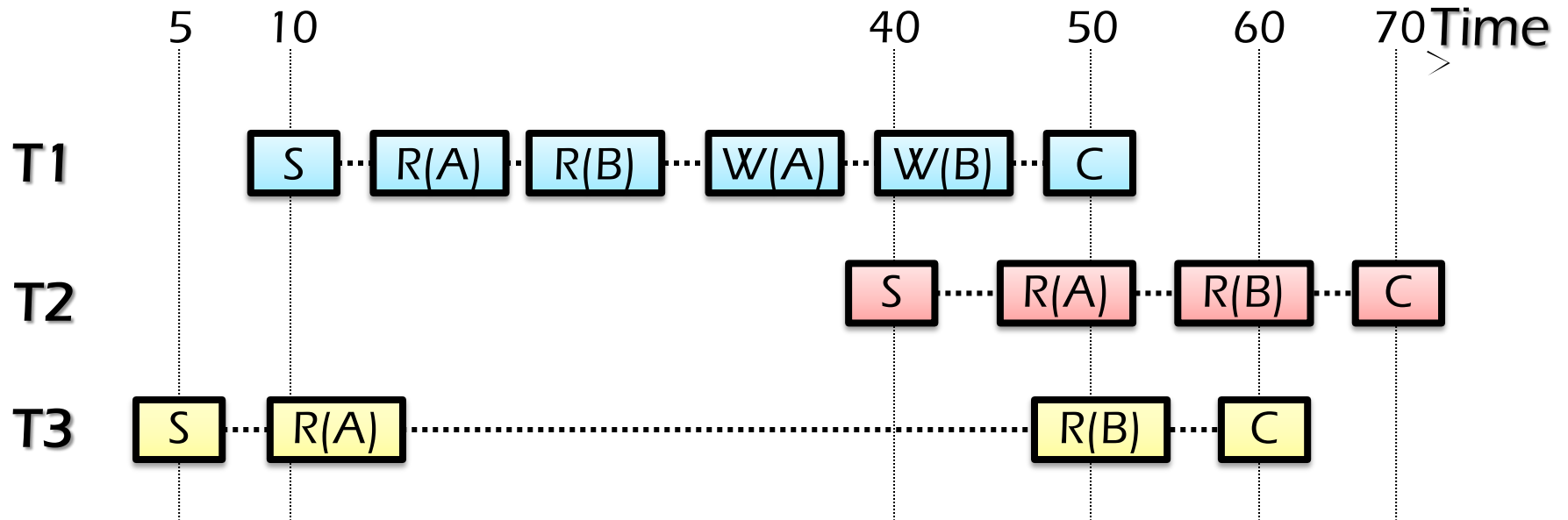
T buffers writes to **X** and adds **X**
to its write-set, **T.wset** += {**X**}

Example



T.sts = 10
R(A) = A₇
R(B) = B₈
T.cts = 50
W(A) = A₅₀
W(B) = B₅₀

Example

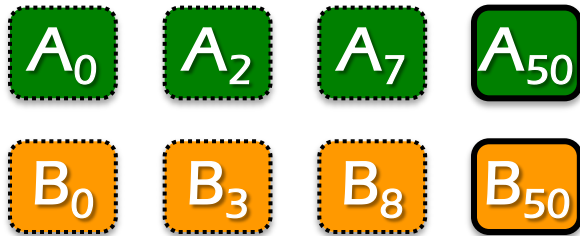
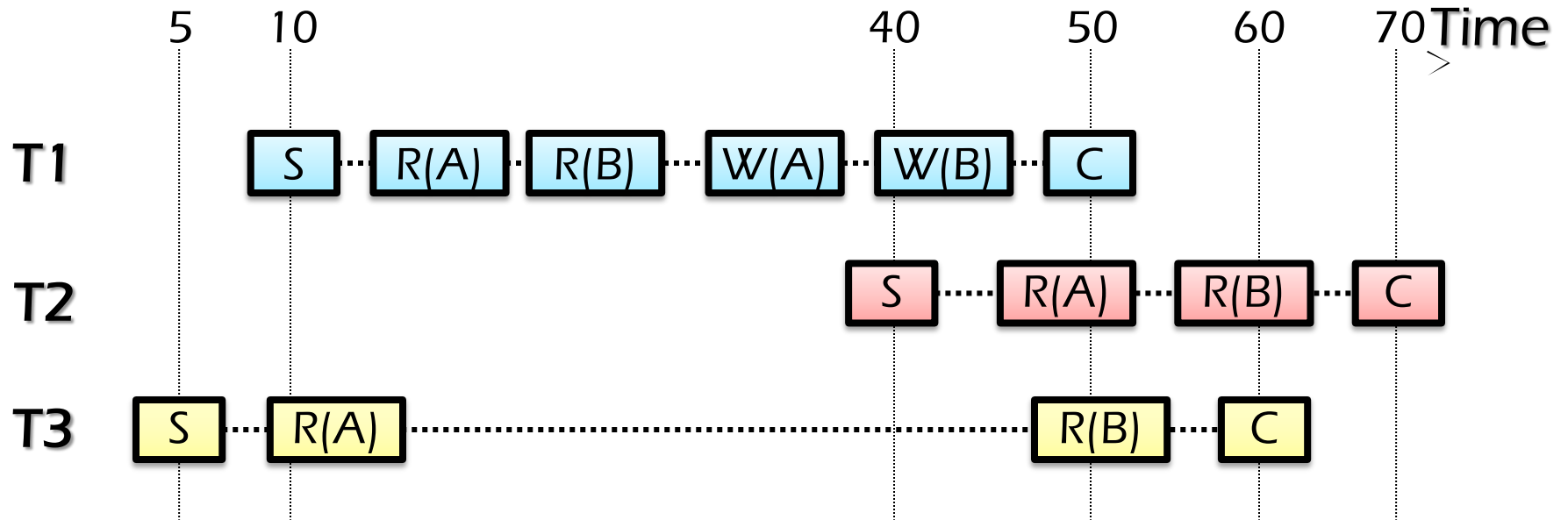


T.sts = 10
R(A) = A_7
R(B) = B_8
T.cts = 50
W(A) = A_{50}
W(B) = B_{50}

T.sts = 40
R(A) = A_7
R(B) = B_8
T.cts = 70

No read-uncommitted

Example



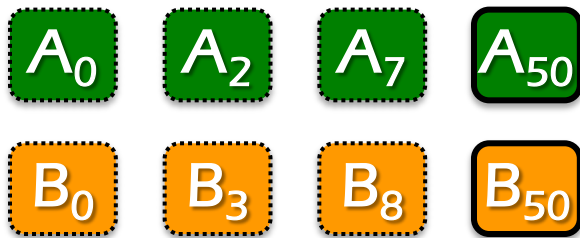
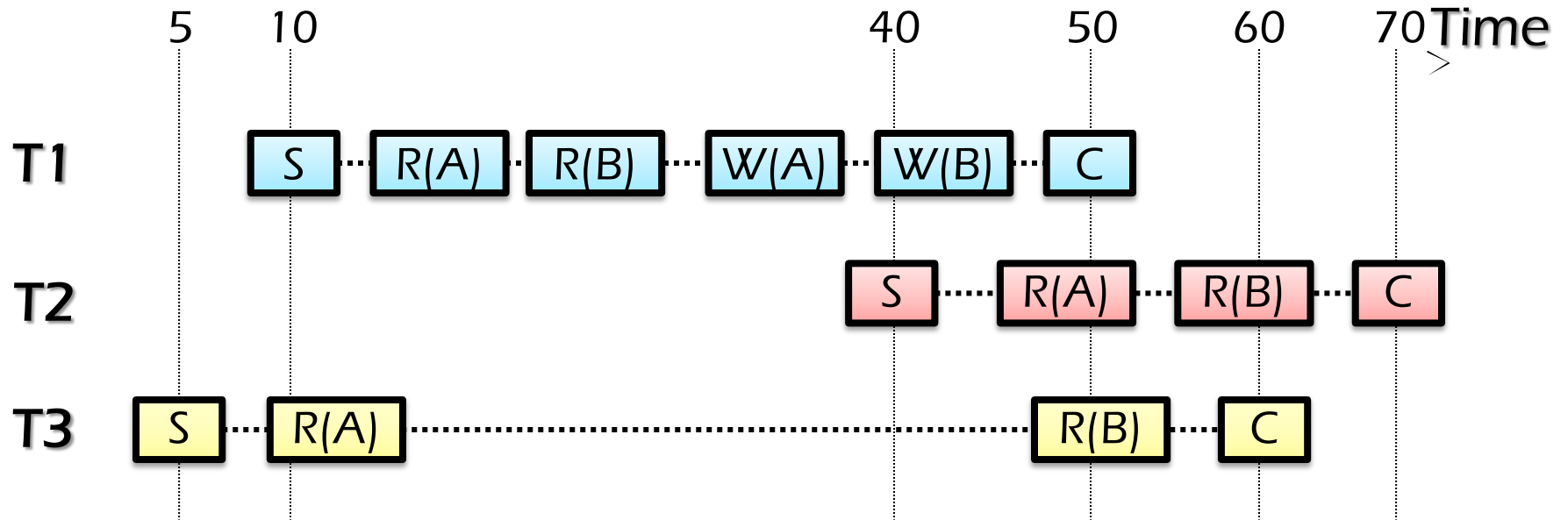
$T.sts = 10$
 $R(A) = A_7$
 $R(B) = B_8$
 $T.cts = 50$
 $W(A) = A_{50}$
 $W(B) = B_{50}$

$T.sts = 40$
 $R(A) = A_7$
 $R(B) = B_8$
 $T.cts = 70$

$T.sts = 5$
 $R(A) = A_2$
 $R(B) = B_3$
 $T.cts = 60$

No unrepeatable-read

Example



Equivalent Serial Order

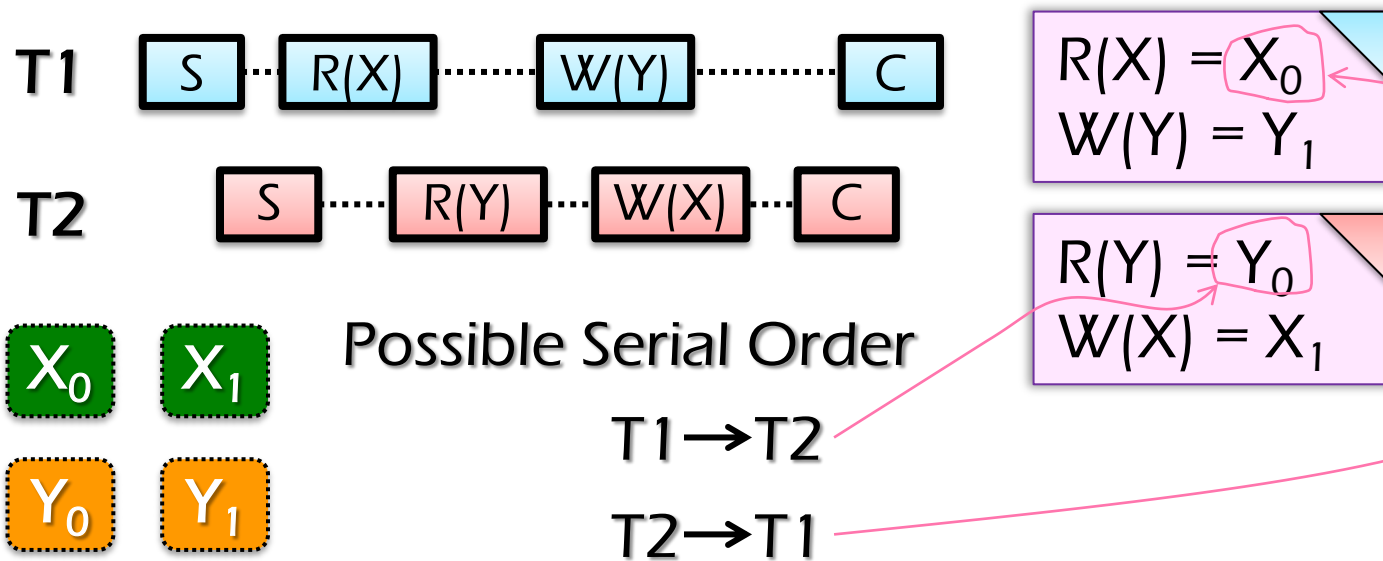
$T3 \rightarrow A_7 \rightarrow B_8 \rightarrow T2 \rightarrow T1$

Snapshot Isolation is serializable or not ?

Question

Snapshot isolation (SI) differs from serializability due to one **anomaly** that is possible under SI but not under serializability.

Describe the anomaly and also give a concrete application for which the anomaly is undesirable.



Next Lecture

Topic: Two-phase Commit (2PC)

THANKS