CS150: Database & Datamining

Lecture 9: Transactions II: Intro to Concurrency & Locking

ShanghaiTech-SIST

Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

Announcements

1. Mid-term next lecture
    - Up to Relational Algebra (No questions on Transactions)

Today’s Lecture

1. Concurrency, scheduling & anomalies

2. Locking: 2PL, conflict serializability, deadlock detection

1. Concurrency, Scheduling & Anomalies

What you will learn about in this section

1. Interleaving & scheduling

2. Conflict & anomaly types

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that…

1. **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**
   - DBMS handles the details of *interleaving* various TXNs

ACID

2. **Consistency** is maintained: TXNs must leave the DB in a **consistent state**
   - DBMS handles the details of enforcing integrity constraints

ACID

Note the hard part…

…is the effect of *interleaving* transactions and *crashes*.
Maybe later for the gory details!

Example- consider two TXNs:

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = 'A'

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = 'B'
COMMIT
```

T1 transfers $100 from B's account to A's account

```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

We can look at the TXNs in a timeline view- serial execution:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

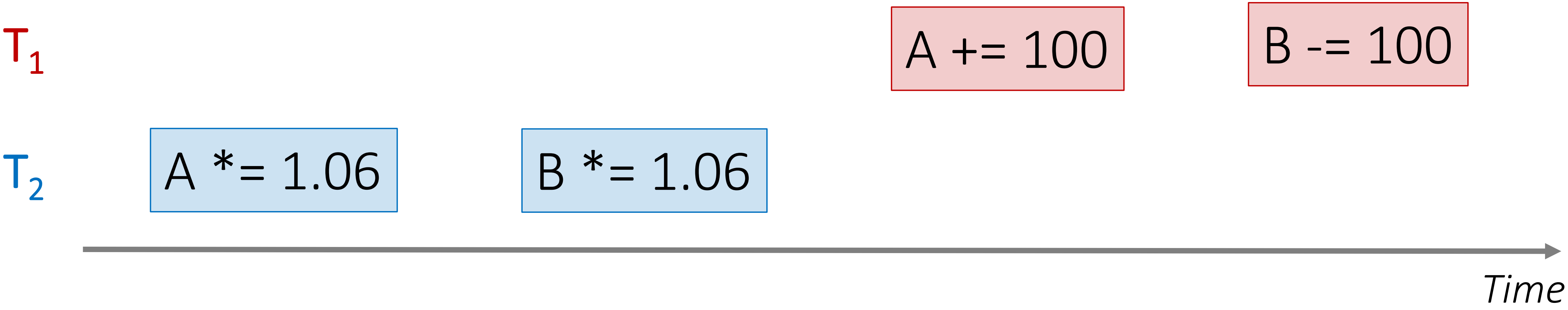
Time

T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!

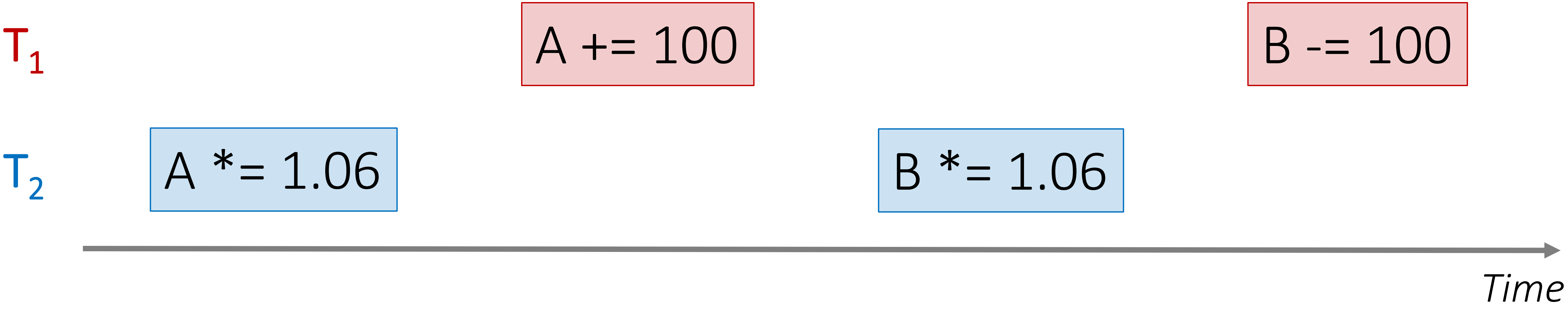


T2 credits both accounts with a 6% interest payment

T1 transfers $100 from B’s account to A’s account

Example- consider two TXNs:

The DBMS can also **interleave** the TXNs

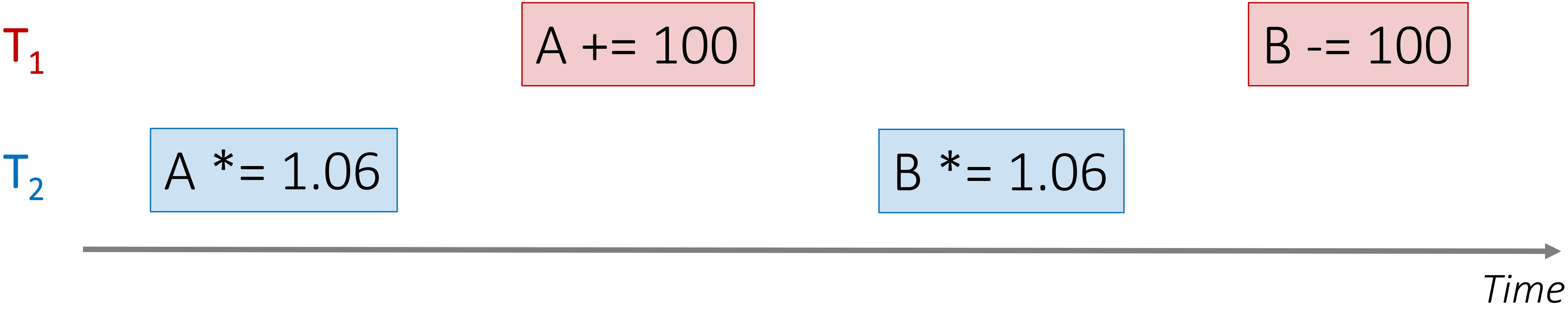


T2 credits A's account with 6% interest payment, then T1 transfers $100 to A's account…

T2 credits B's account with a 6% interest payment, then T1 transfers $100 from B's account…

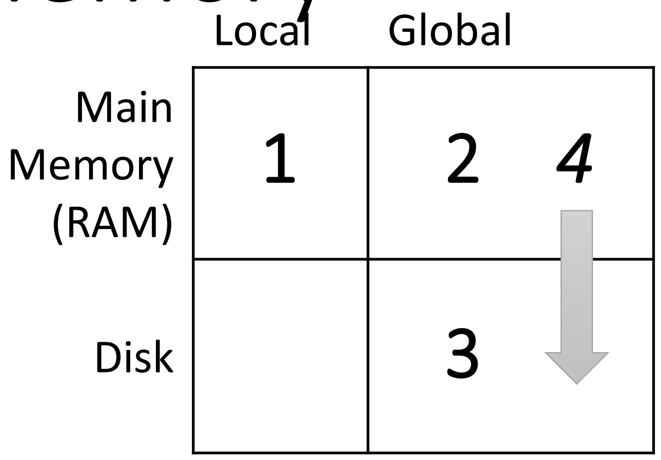
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



What goes wrong here??

Recall: Three Types of Regions of Memory



1. **Local:** In our model each process in a DBMS has its own local memory, where it stores values that only it "sees"

2. **Global:** Each process can read from / write to shared data in main memory

3. **Disk:** Global memory can read from / flush to disk

4. ***Log:*** *Assume on stable disk storage- spans both main memory and disk…*

Log is a *sequence* from main memory -> disk

"Flushing to disk" = writing to disk.

Why Interleave TXNs?

- Interleaving TXNs might lead to anomalous outcomes… why do it?

- Several important reasons:
  - Individual TXNs might be *slow*- don't want to block other users during!

  - Disk access may be *slow*- let some TXNs use CPUs while others accessing disk!

All concern large differences in ***performance***

Interleaving & Isolation

- The DBMS has freedom to interleave TXNs

- However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained

  - Must be *as if* the TXNs had executed serially!

"With great power comes great responsibility"

A<u>CI</u>D

DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule T_1,T_2:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

Interleaved schedule A:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

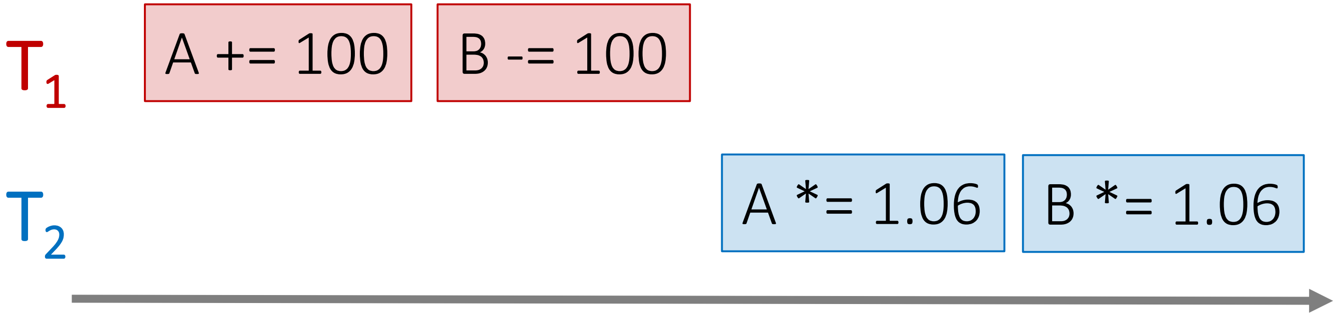
Same result!

Scheduling examples

Starting Balance

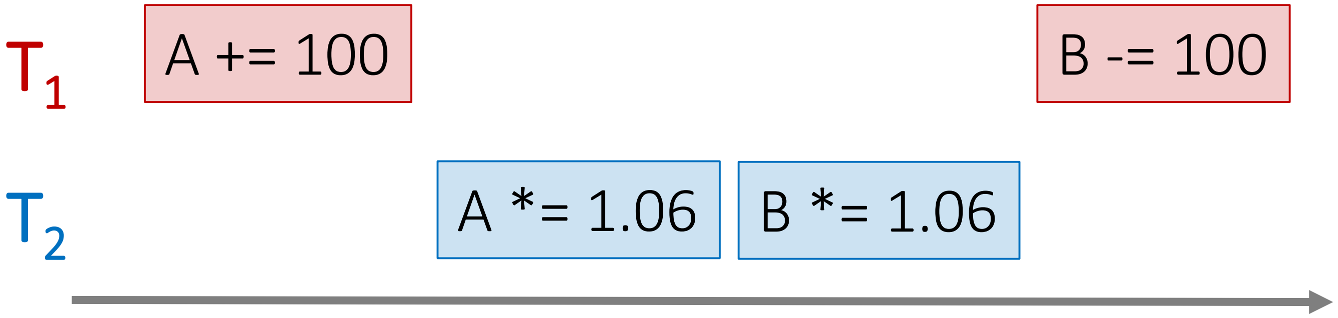
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule B:



A	B
$159	$112

Different result than serial T_1,T_2!

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule *T_2,T_1:*

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$153	$112

Interleaved schedule B:

T_1 A += 100 B -= 100

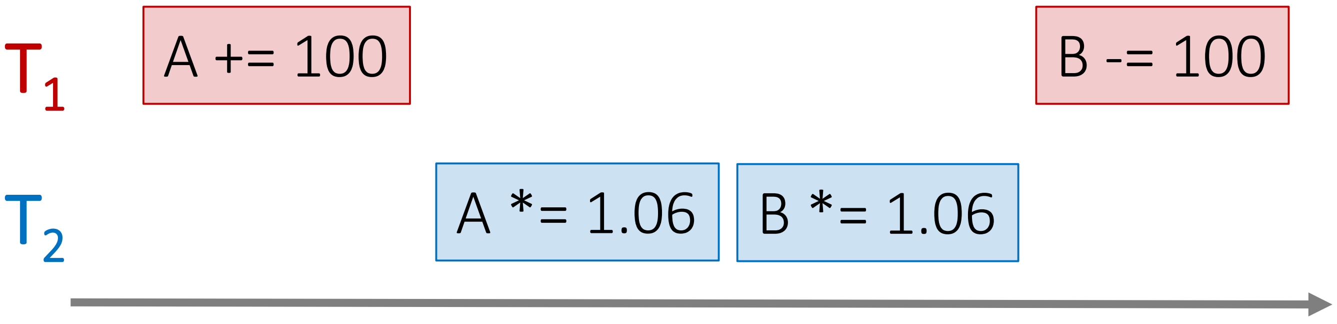
T_2 A *= 1.06 B *= 1.06

A	B
$159	$112

Different result than serial T_2,T_1 ALSO!

Scheduling examples

Interleaved schedule B:



This schedule is different than ***any serial order!*** We say that it is **not serializable**

Scheduling Definitions

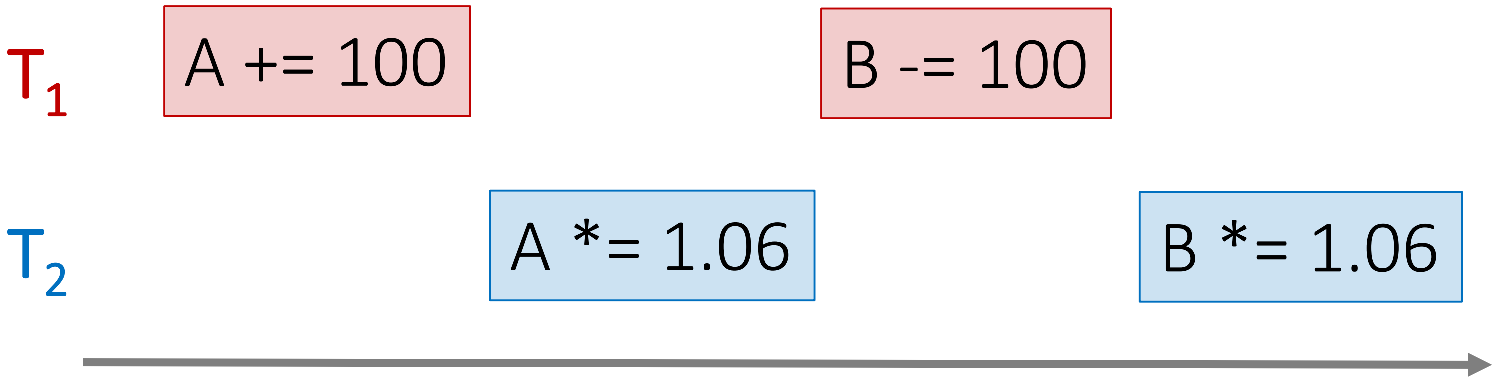
- A **serial schedule** is one that does not interleave the actions of different transactions

- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B

- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some**" makes this definition powerful & tricky!

Serializable?



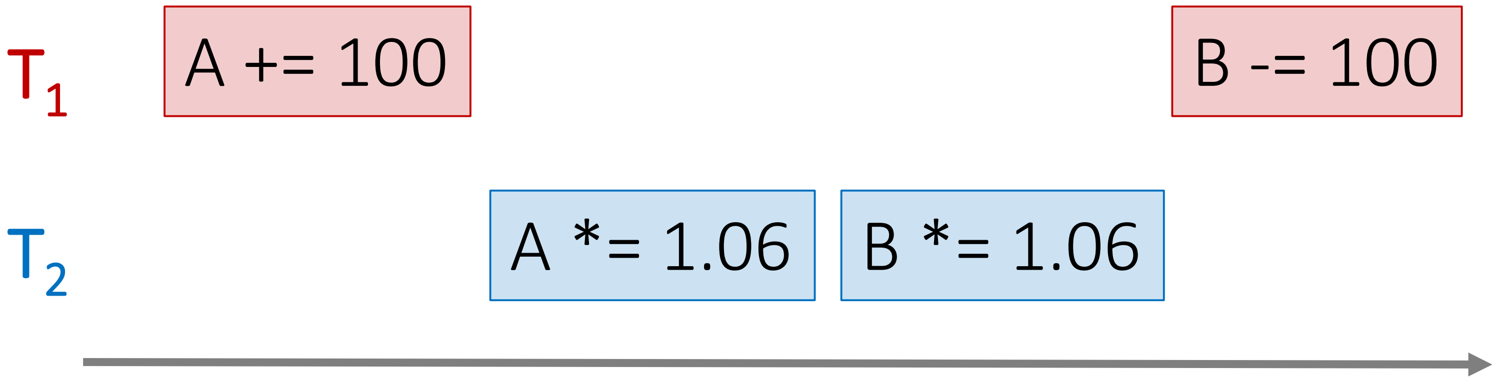
Serial schedules:

	A	B
T_1, T_2	1.06*(A+100)	1.06*(B-100)
T_2, T_1	1.06*A + 100	1.06*B - 100

A	B
1.06*(A+100)	1.06*(B-100)

Same as a serial schedule *for all possible values of A, B* = <u>serializable</u>

Serializable?



Serial schedules:

	A	B
T_1,T_2	1.06*(A+100)	1.06*(B-100)
T_2,T_1	1.06*A + 100	1.06*B - 100

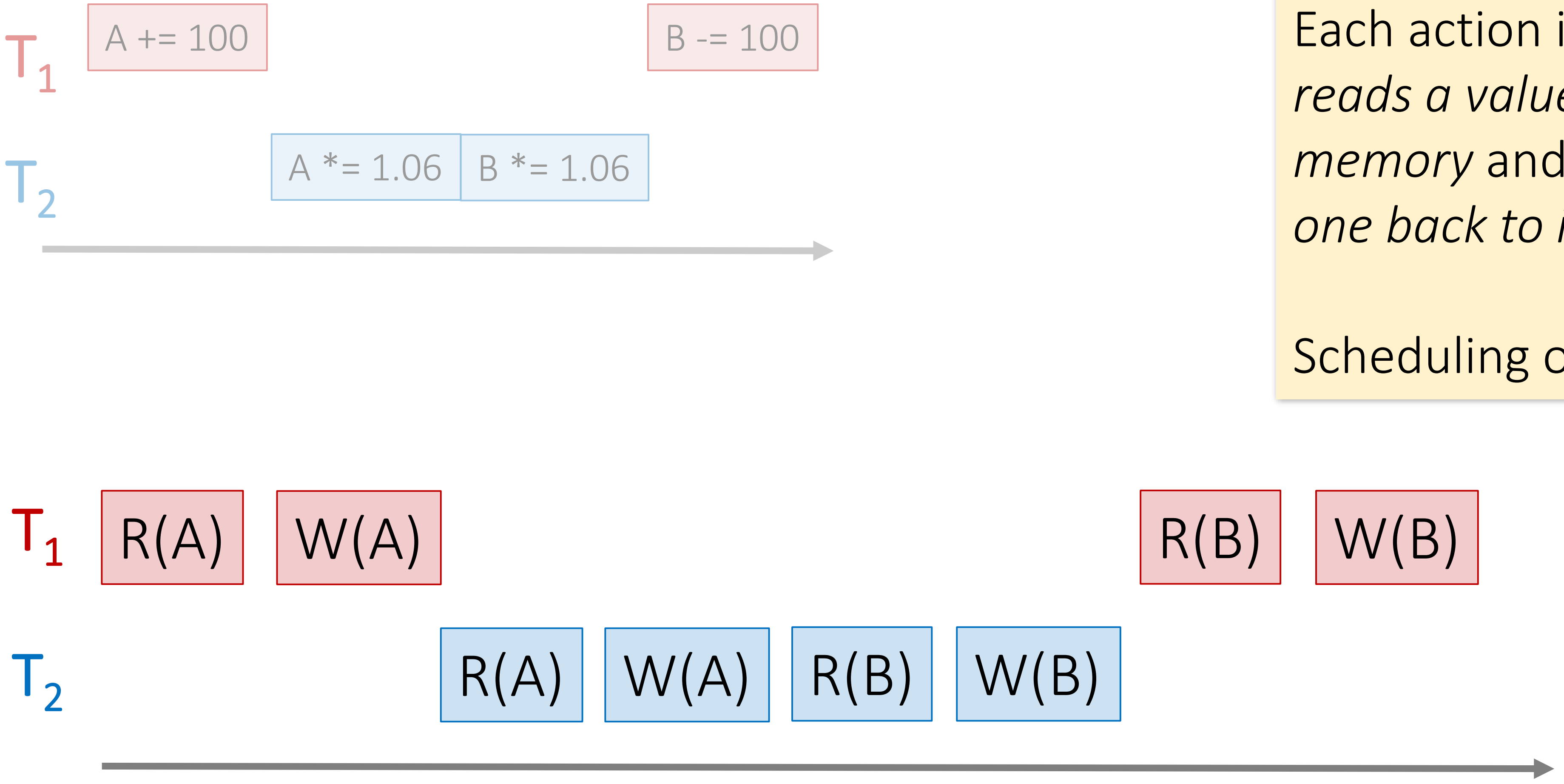
A	B
1.06*(A+100)	1.06*B - 100

Not *equivalent* to any serializable schedule = ***not serializable***

What else can go wrong with interleaving?

- Various anomalies which break isolation / serializability
    - Often referred to by name…
- Occur because of / with certain “conflicts” between interleaved TXNs

The DBMS's view of the schedule



Each action in the TXNs *reads a value from global memory* and then *writes one back to it*

Scheduling order matters!

Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

Why no "RR Conflict"?

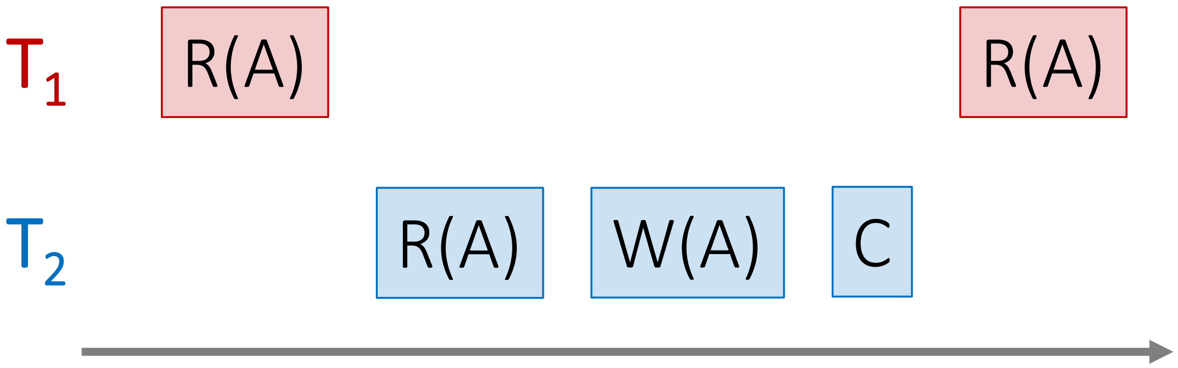
Interleaving anomalies occur with / because of these conflicts between TXNs *(but these conflicts can occur without causing anomalies!)*

See next section for more!

Classic Anomalies with Interleaved Execution

"Unrepeatable read":

Example:



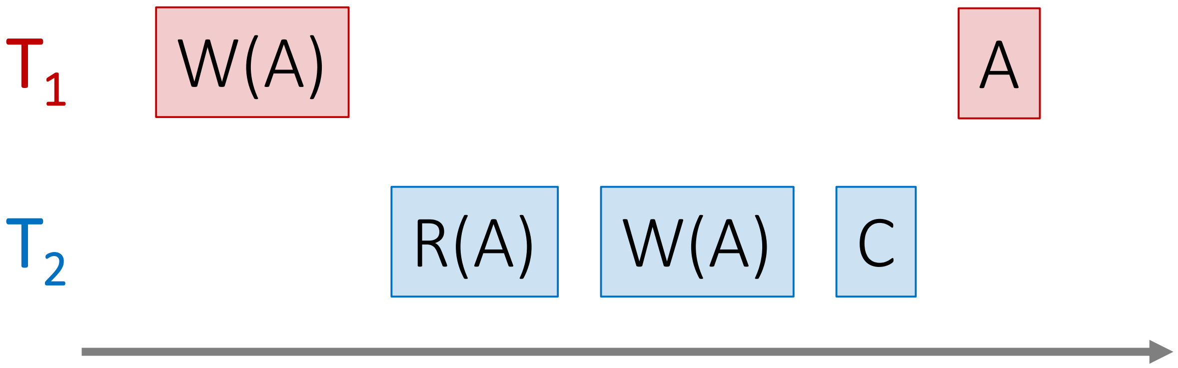
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

Occurring with / because of a ***RW conflict***

Classic Anomalies with Interleaved Execution

"Dirty read" / Reading uncommitted data:

Example:



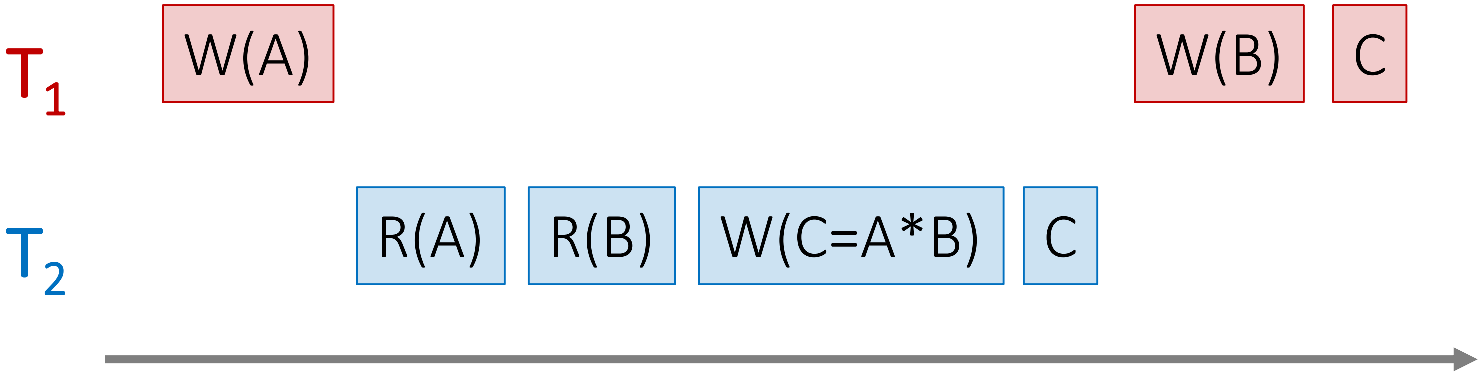
1. T_1 writes some data to A
2. T_2 reads from A, then writes back to A & commits
3. T_1 then aborts- *now T_2's result is based on an obsolete / inconsistent value*

Occurring with / because of a ***WR conflict***

Classic Anomalies with Interleaved Execution

"Inconsistent read" / Reading partial commits:

Example:



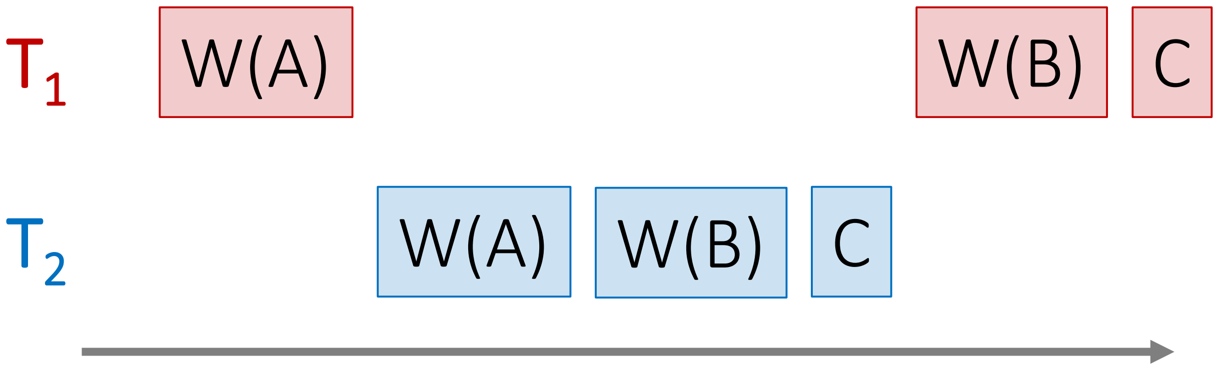
1. T_1 writes some data to A
2. T_2 reads from A *and B*, and then writes some value which depends on A & B
3. T_1 then writes to B- *now T_2's result is based on an incomplete commit*

*Again, occurring because of a **WR conflict***

Classic Anomalies with Interleaved Execution

Partially-lost update:

Example:



1. T_1 *blind* writes some data to A
2. T_2 *blind* writes to A *and B*
3. T_1 then *blind* writes to B; now we have T_2's value for B and T_1's value for A- ***not equivalent to any serial schedule!***

Occurring because of a ***WW conflict***

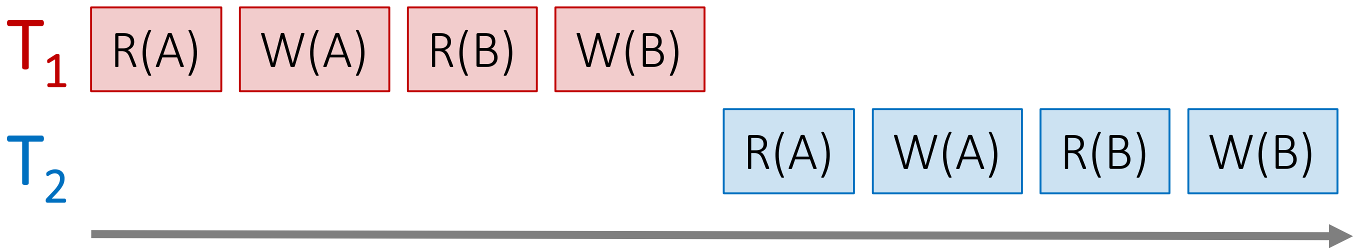
2. Conflict Serializability, Locking & Deadlock

What you will learn about in this section

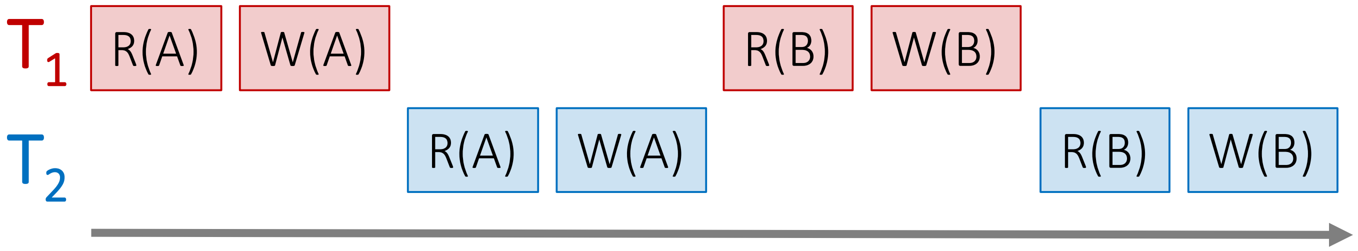
1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

Recall: Concurrency as Interleaving TXNs

Serial Schedule:



Interleaved Schedule:



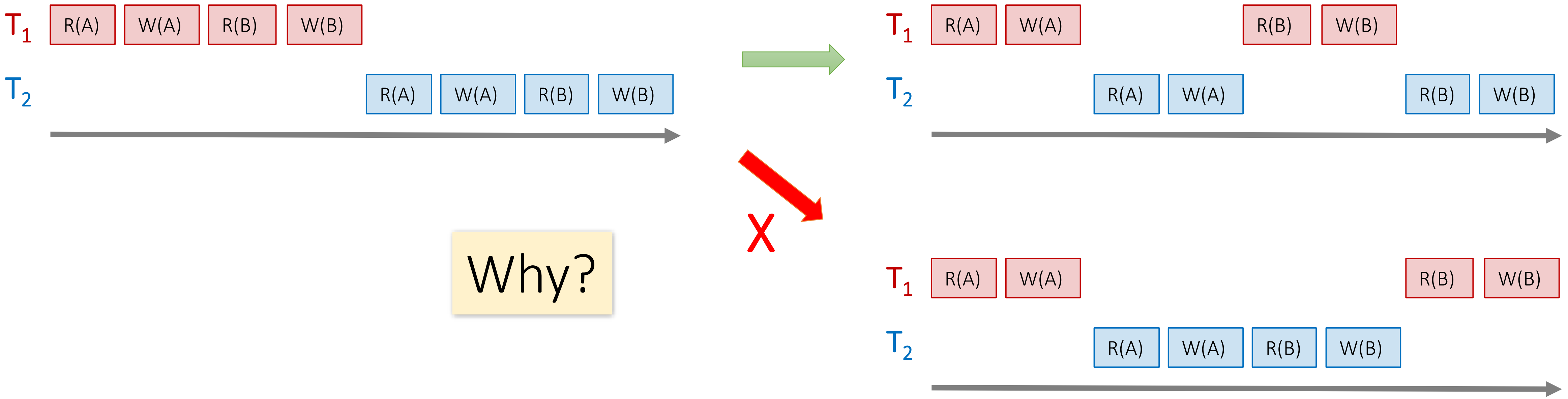
- For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

We call the particular order of interleaving a <u>schedule</u>

Recall: “Good” vs. “bad” schedules

Serial Schedule:

Interleaved Schedules:



We want to develop ways of discerning “good” vs. “bad” schedules

Ways of Defining "Good" vs. "Bad" Schedules

- Recall from last time: we call a schedule ***serializable*** if it is equivalent to *some* serial schedule

  - We used this as a notion of a "good" interleaved schedule, since **a serializable schedule will maintain isolation & consistency**

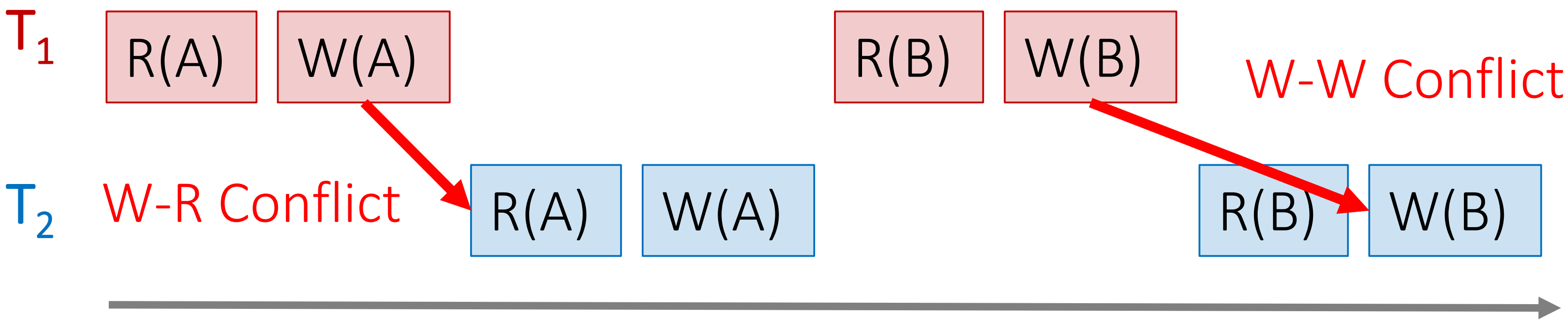
- Now, we'll define a stricter, but very useful variant:

  - ***<u>Conflict serializability</u>***

We'll need to define *conflicts* first..

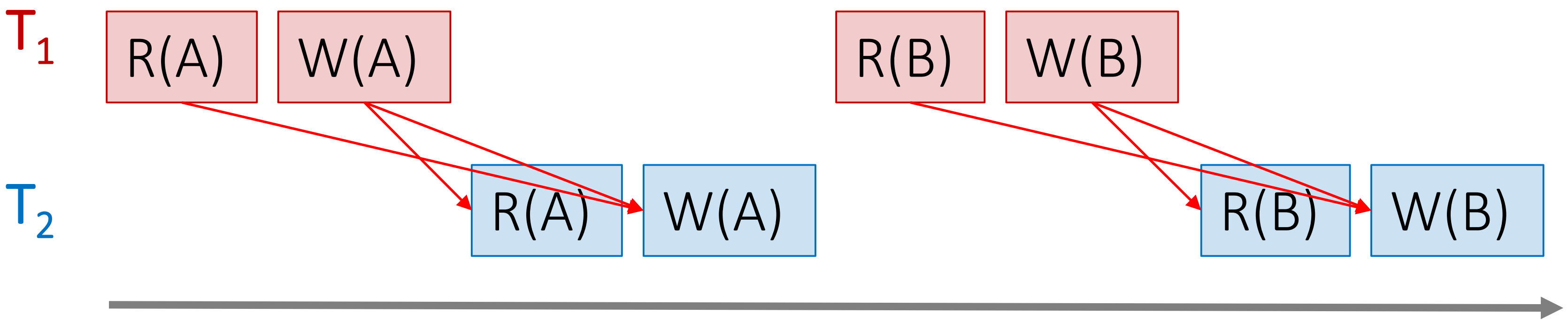
Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflicts

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write



All "conflicts"!

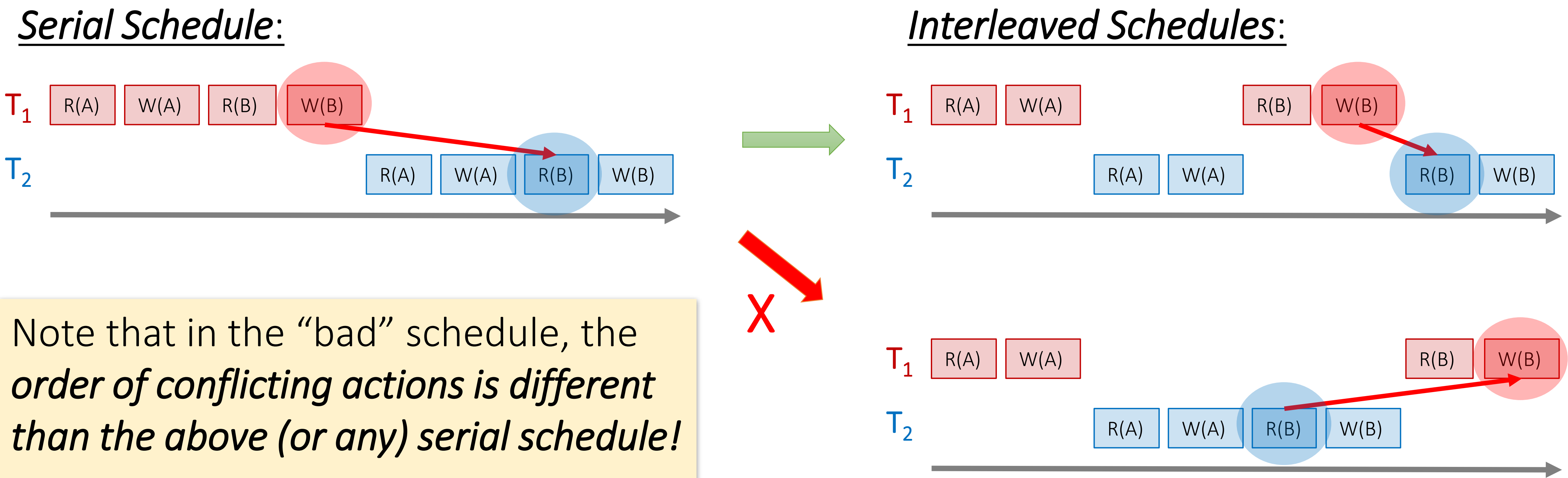
Conflict Serializability

- Two schedules are **conflict equivalent** if:
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

Conflict serializable ⇒ serializable

So if we have conflict serializable, we have consistency & isolation!

Recall: “Good” vs. “bad” schedules



Note that in the “bad” schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

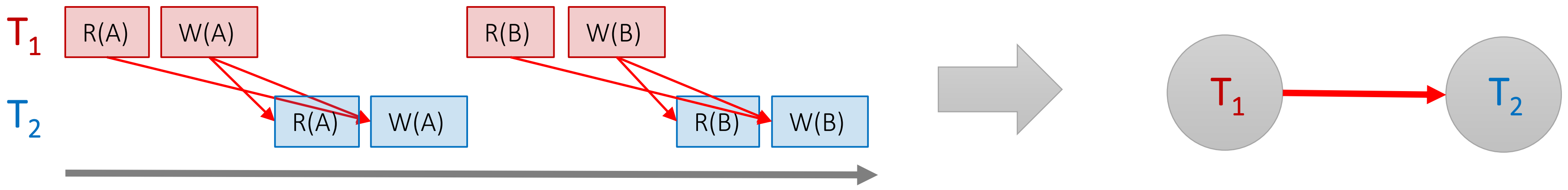
Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!

Note: Conflicts vs. Anomalies

- **Conflicts** are things we talk about to help us characterize different schedules
  - Present in both “good” and “bad” schedules
- **Anomalies** are instances where isolation and/or consistency is broken because of a “bad” schedule
  - We often characterize different anomaly types by what types of conflicts predicated them

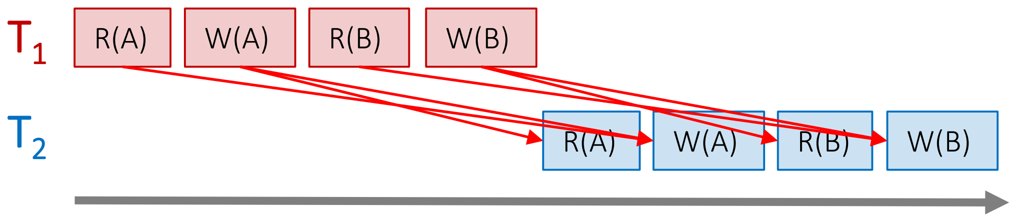
The Conflict Graph

- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if an action in T_i <u>precede and conflict with</u> an action in T_j**

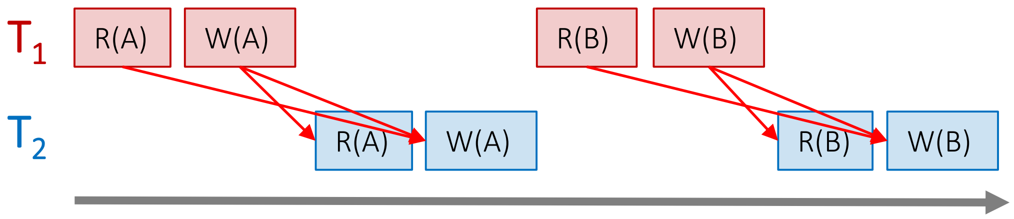


What can we say about "good" vs. "bad" conflict graphs?

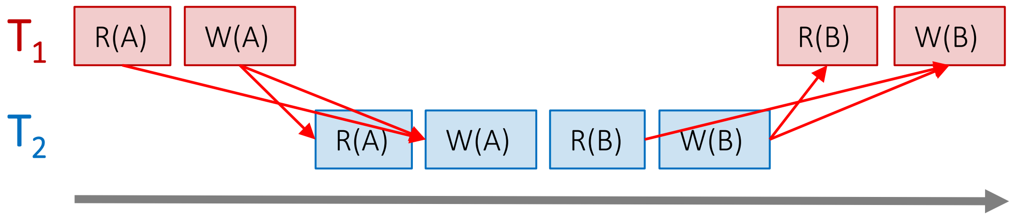
Serial Schedule:



Interleaved Schedules:



X

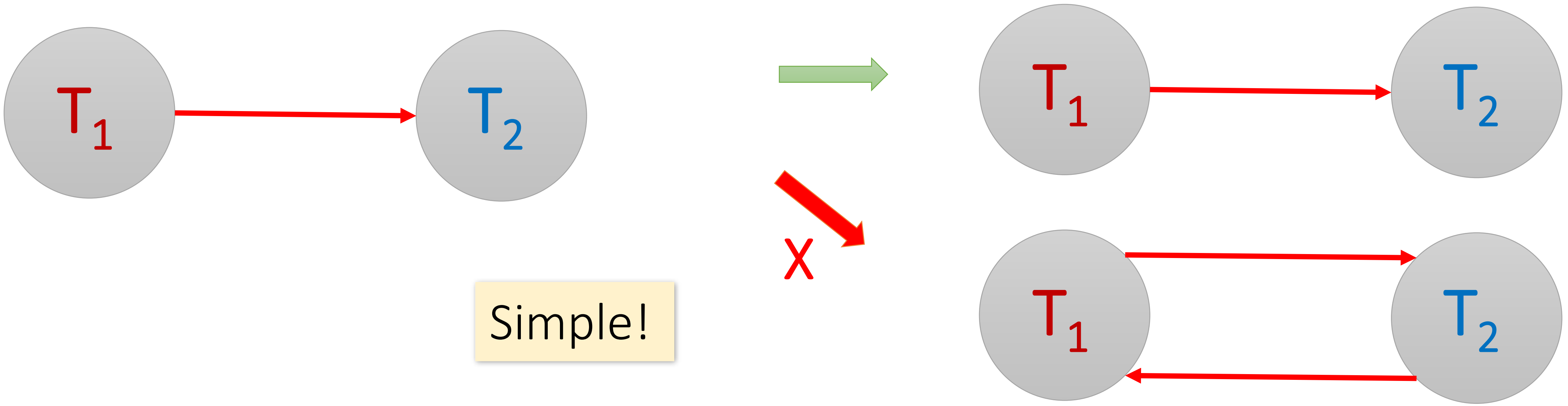


A bit complicated…

What can we say about “good” vs. “bad” conflict graphs?

__Serial Schedule__:

__Interleaved Schedules__:



Theorem: Schedule is **conflict serializable** if and only if its conflict graph is __acyclic__

Let’s unpack this notion of acyclic conflict graphs…

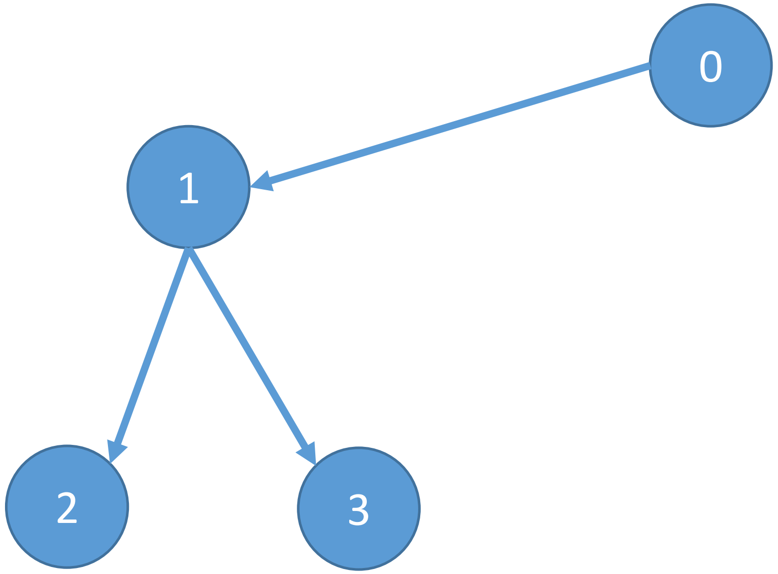
DAGs & Topological Orderings

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges

- A directed <u>**acyclic**</u> graph (DAG) always has one or more **topological orderings**
  - (And there exists a topological ordering *if and only if* there are no directed cycles)

DAGs & Topological Orderings

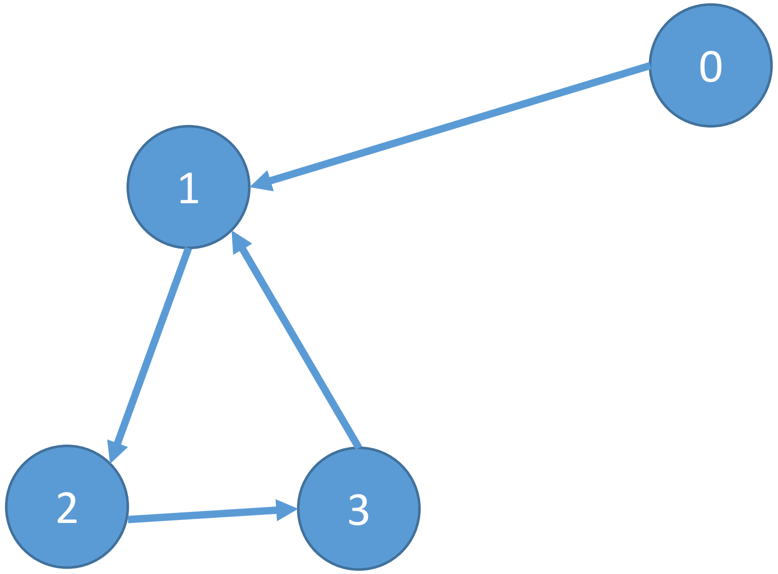
- Ex: What is one possible topological ordering here?



Ex: 0, 1, 2, 3 (or: 0, 1, 3, 2)

DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



There is none!

Connection to conflict serializability

- In the conflict graph, a topological ordering of nodes corresponds to **a serial ordering of TXNs**

- Thus an **acyclic** conflict graph → conflict serializable!

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is acyclic

Strict Two-Phase Locking

- We consider **locking**- specifically, *strict two-phase locking*- as a way to deal with concurrency, because it **guarantees conflict serializability (if it completes- see upcoming…)**

- Also (*conceptually*) straightforward to implement, and transparent to the user!

Two-Phase Locking (2PL)

- The most common scheme for enforcing conflict serializability
- A bit “pessimistic”
  - Sets locks for fear of conflict.. Some cost here.
  - Alternative schemes “optimistically” let transactions move forward, and aborting them when conflicts are detected.
    - Not today

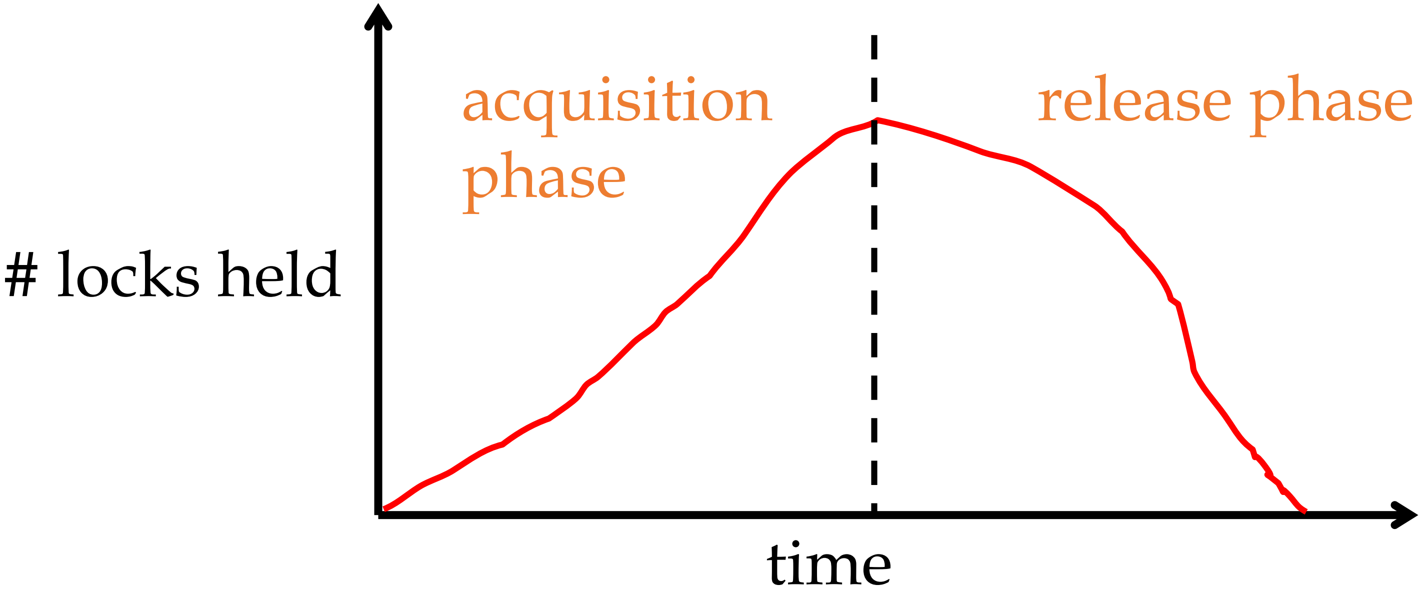
Two-phase Locking (2PL) Protocol:

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- TXNs cannot get new locks after releasing any locks.

Note: Terminology here- "exclusive", "shared"- meant to be intuitive- no tricks!

Two-Phase Locking (2PL)



Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

2PL guarantees conflict serializability ☺

But, does *not* prevent **Cascading Aborts**. ☹

Two-phase Locking (2PL)

- *Problem:* Cascading Aborts
- *Example:* rollback of T1 requires rollback of T2!

T1:	R(A), W(A)		Abort
T2:		R(A), W(A)	

What goes wrong here??

Strict Two-phase Locking (Strict 2PL) Protocol:

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

TXNs obtain:

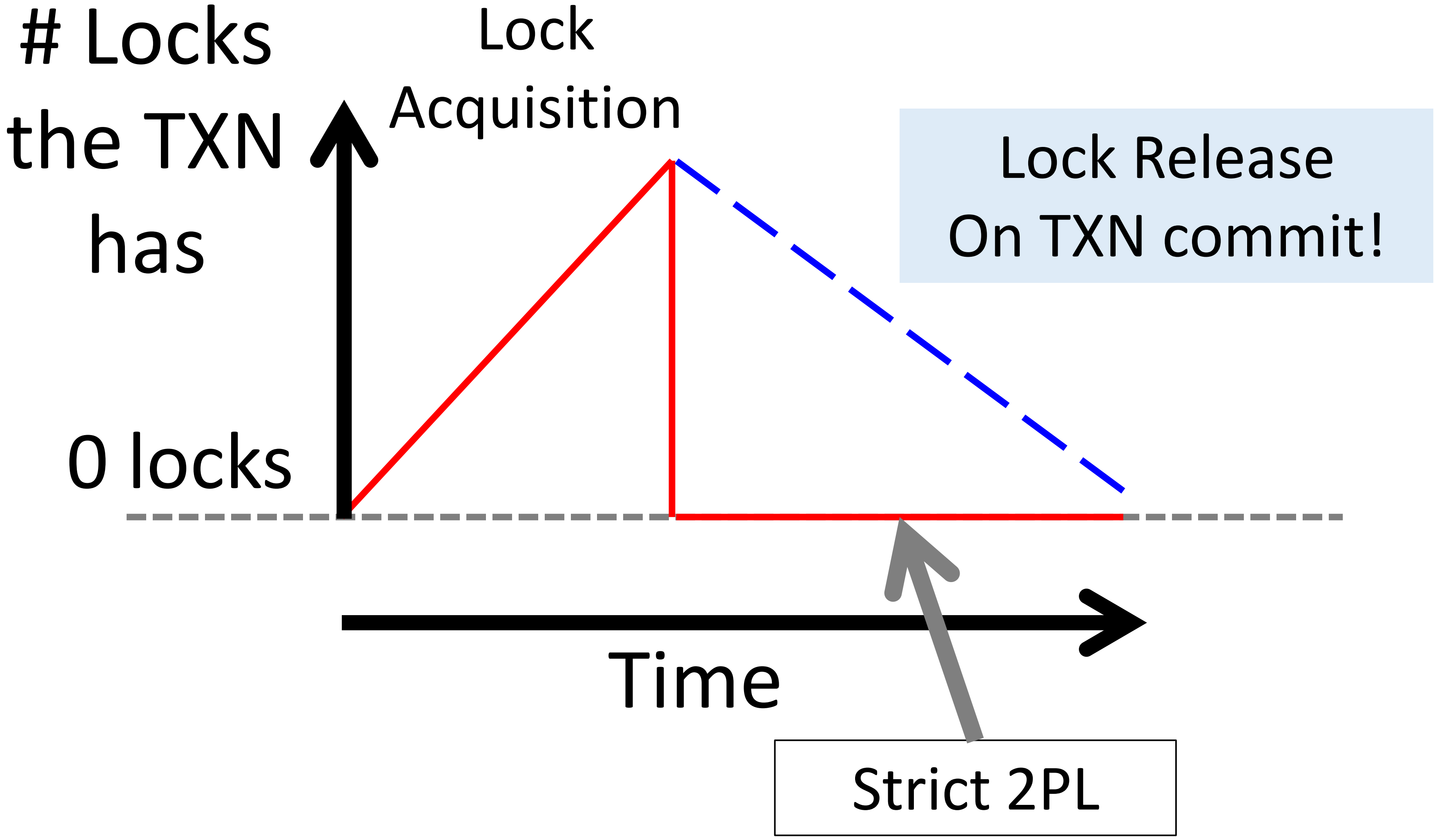
- Same as 2PL, except:
- All locks held by a TXN are released only when TXN completes. i.e., either:

  (a) transaction has committed (commit record on disk),

  or

  (b) transaction has aborted and rollback is complete.

Picture of 2-Phase Locking (2PL)



Strict 2PL

Theorem: Strict 2PL allows only schedules whose dependency graph is acyclic

Proof Intuition: In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable ⇒ serializable schedules

Non-2PL, A= 1000, B=2000, Output =?

T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	
	Read(A)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B)
	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

2PL, A= 1000, B=2000, Output =?

T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Strict 2PL, A= 1000, B=2000, Output =?

T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

Strict 2PL

- If a schedule follows strict 2PL and locking, it is conflict serializable…
    - …and thus serializable
    - …and thus maintains isolation & consistency!

- Not all serializable schedules are allowed by strict 2PL.

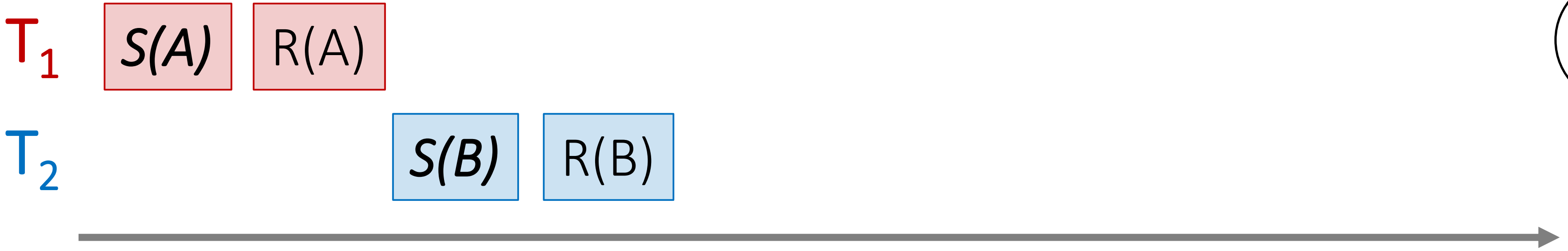
- So let’s use strict 2PL, what could go wrong?

Deadlock Detection: Example

Waits-for graph:

First, T_1 requests a shared lock on A to read from it

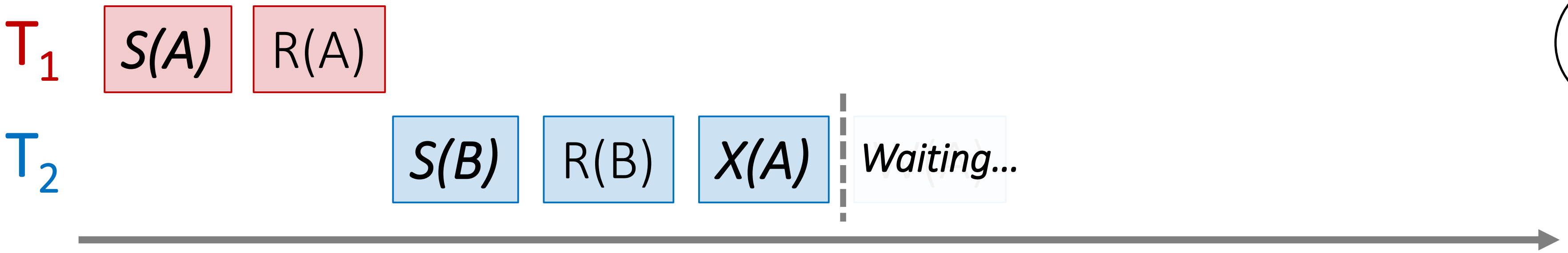
Deadlock Detection: Example



Waits-for graph:

Next, T_2 requests a shared lock on B to read from it

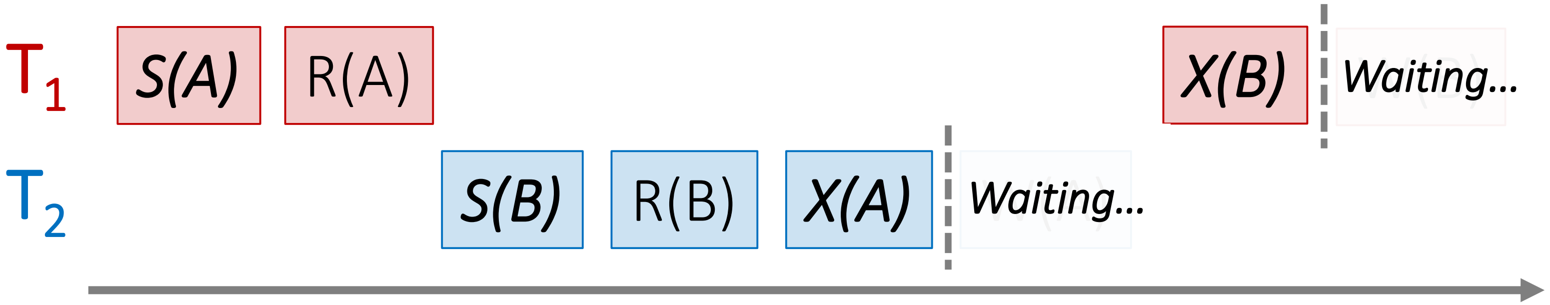
Deadlock Detection: Example



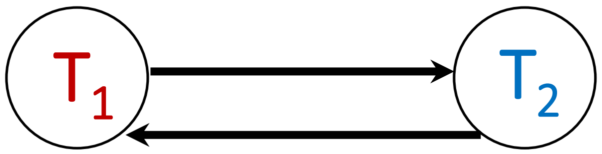
Waits-for graph:

T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Deadlock Detection: Example



Waits-for graph:



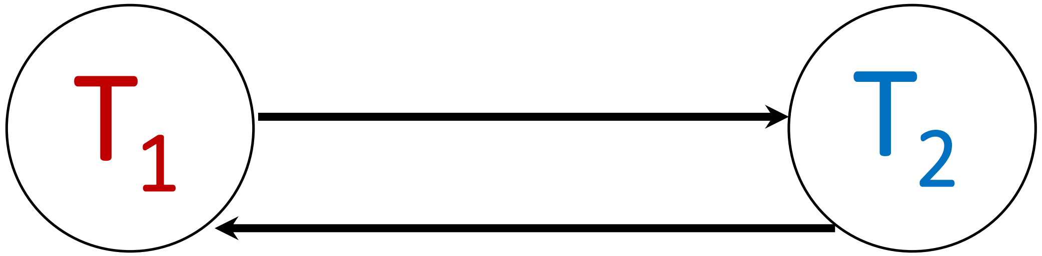
Cycle =
DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

```
sqlite3.OperationalError: database is locked
```

```
ERROR:  deadlock detected
DETAIL:  Process 321 waits for ExclusiveLock on tuple of
relation 20 of database 12002; blocked by process 4924.
Process 404 waits for ShareLock on transaction 689; blocked
by process 552.
HINT:  See server log for query details.
```

The problem?
Deadlock!??!



Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.

- Two ways of dealing with deadlocks:

  1. Deadlock prevention

  2. Deadlock detection

Deadlock Prevention

- Common technique in operating systems
- Standard approach: resource ordering
  - Screen < Network Card < Printer
- Why is this problematic for Xacts in a DBMS?

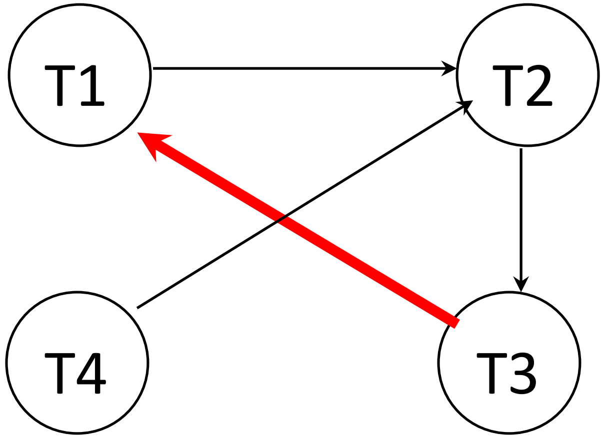
Deadlock Detection

- Create the **waits-for graph**:
  - Nodes are transactions
  - There is an edge from $T_i \rightarrow T_j$ if T_i is *waiting for T_j to release a lock*
- Periodically check for (***and break***) cycles in the waits-for graph

Deadlock Detection Example

Example:

T1:	S(A), S(D),		S(B)				
T2:		X(B)			X(C)		
T3:				S(D), S(C),			X(A)
T4:						X(B)	



Deadlock Avoidance

- Assign priorities based on timestamps.
- Say Ti wants a lock that Tj holds

Two possible policies:

Read the names like a “ternary predicate” on priorities, with the form:

(Ti > Tj) ? X : Y;

Wait-Die: if Ti has higher priority, Ti waits for Tj;
else Ti aborts

Wound-wait: if Ti has higher priority, Tj aborts;
else Ti waits

Summary

- Concurrency achieved by **interleaving TXNs** such that **isolation** & **consistency** are maintained
  - We formalized a notion of **serializability** that captured such a "good" interleaving schedule

- We defined **conflict serializability**, which implies serializability

- **Locking** allows only conflict serializable schedules
  - If the schedule completes… (it may deadlock!)