

CSC 261/461 – Database Systems

Lecture 24

Fall 2017

TRANSACTIONS

Announcement

- Poster:
 - You should have sent us the poster by yesterday. If you have not done so, please send us asap.
 - Make sure to send it for printing by tomorrow
 - Read the guidelines for printing the poster on Piazza.
- <https://piazza.com/class/j6it7h5r4gy3ti?cid=123>

Poster Suggestions

1. If you are using any figure that's not your own, you must cite the source.

2. Ideally, we would like you to recreate the images. It's okay if you use images from the paper itself. But make sure the resolution is good (300 dpi). You can do that using Acrobat Reader.

Go to preference->General->Fixed Resolution for Snapshot too images. Select 300 dpi. Now take snapshots. The resolution would be way better.

What we covered last time

- Transactions
- Properties of Transactions: *ACID*
- Logging:
 - Atomicity & Durability
 - Write-Ahead Logging (WAL) protocol

Today's Lecture

1. Concurrency, scheduling & anomalies
2. Locking: Strict 2PL, conflict serializability, deadlock detection
3. Recovery

Concurrency & Locking

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

ACCID

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
```

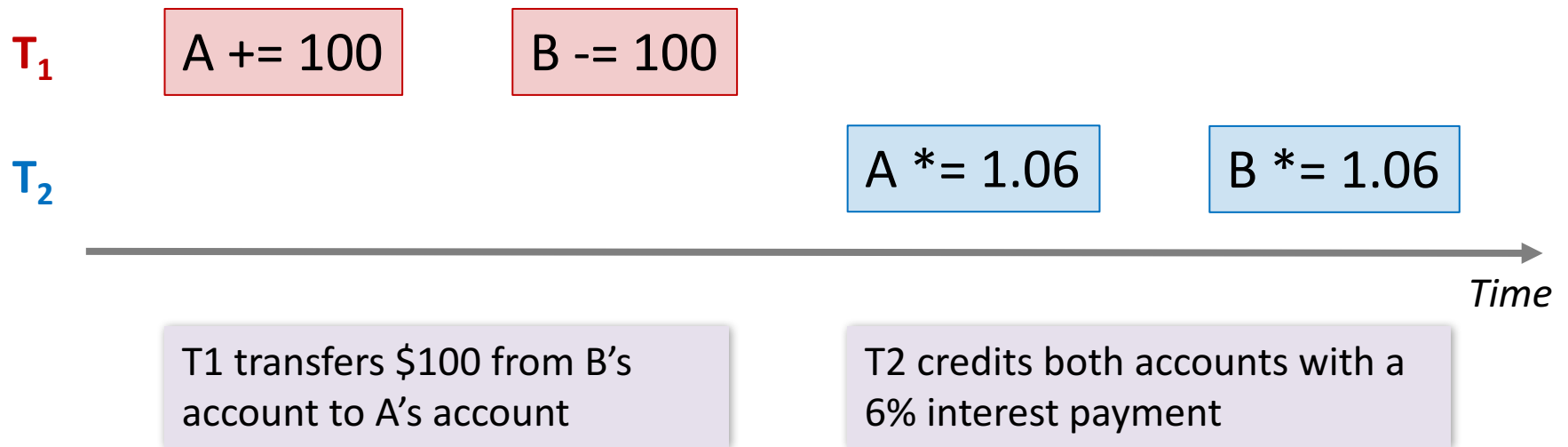
T1 transfers \$100 from B's account to A's account

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

T2 credits both accounts with a 6% interest payment

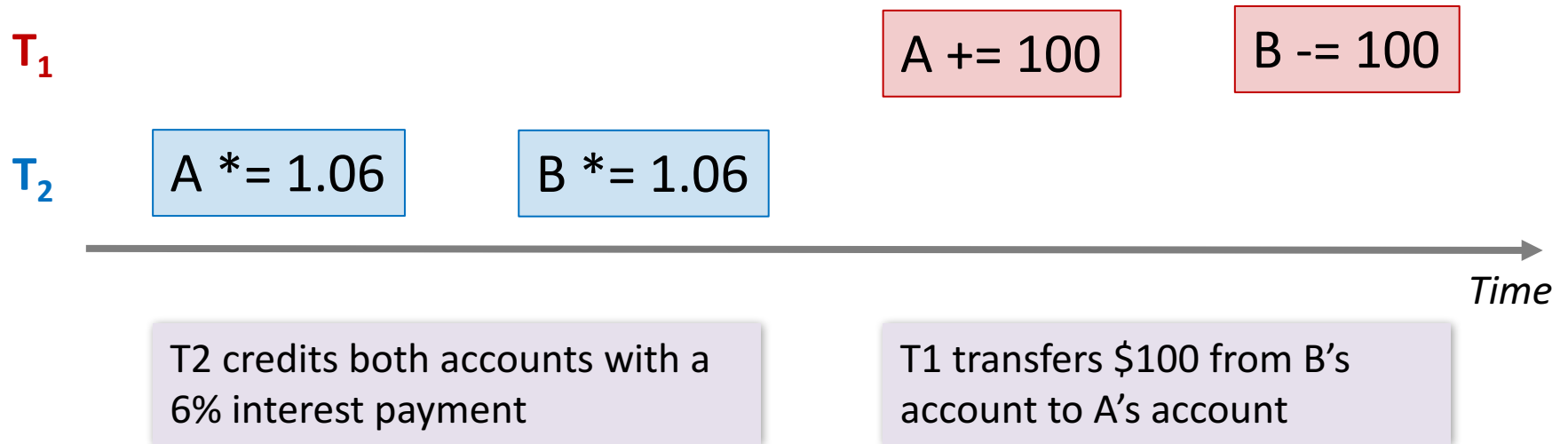
Example- consider two TXNs:

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



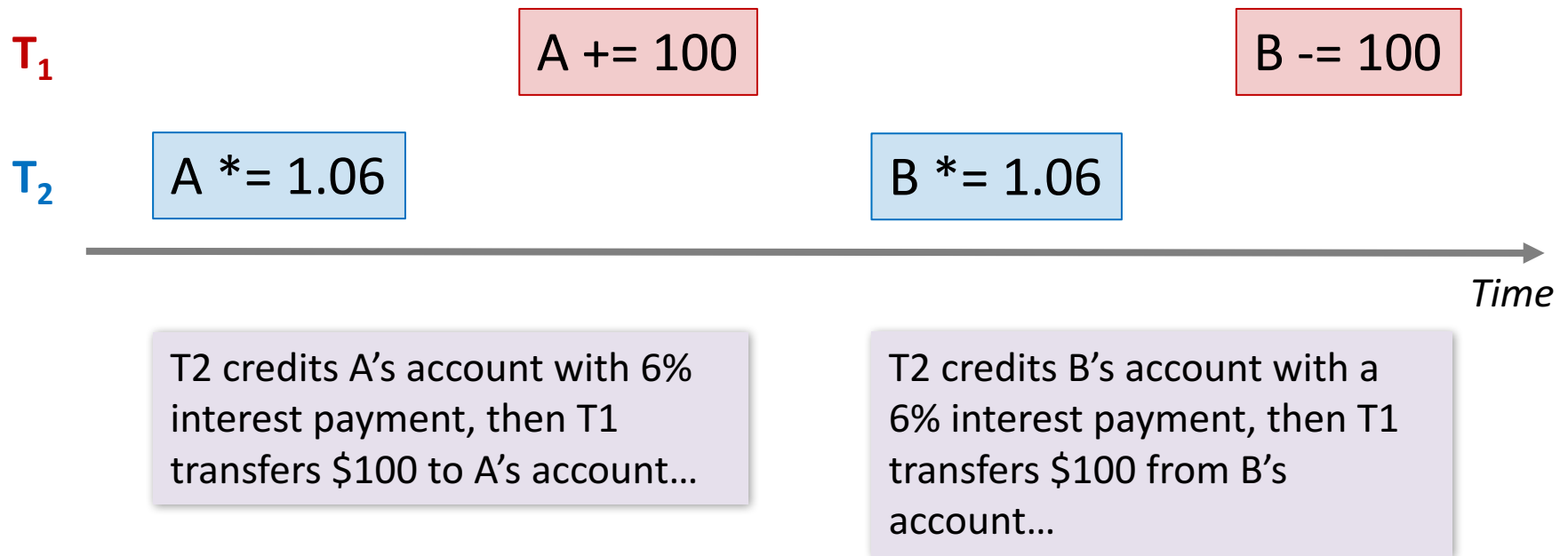
Example- consider two TXNs:

The TXNs could occur in either order... DBMS allows!



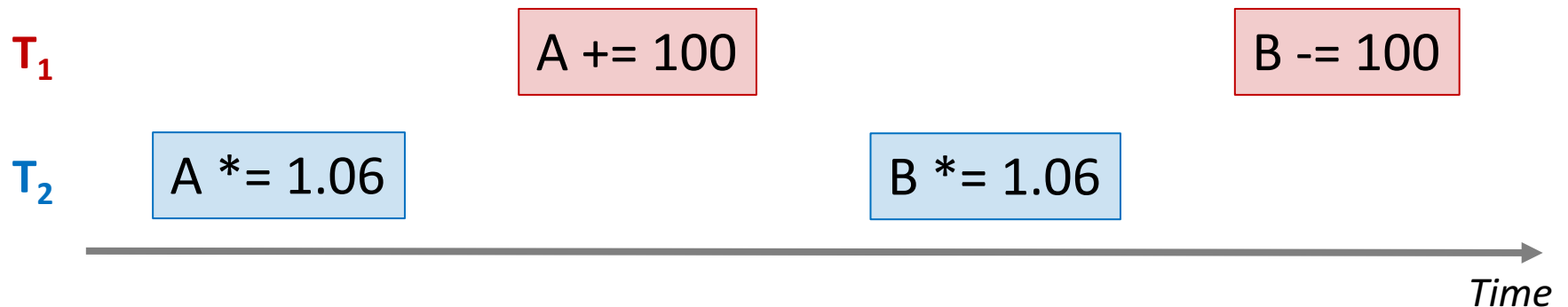
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



Example- consider two TXNs:

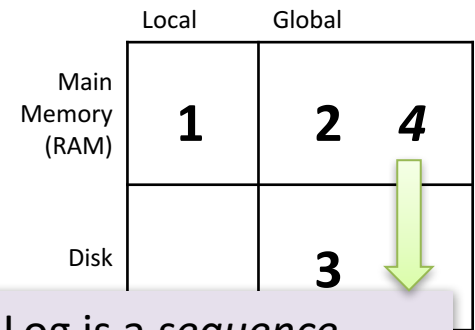
The DBMS can also **interleave** the TXNs



What goes wrong here??

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”

ACID

DBMS must pick a **schedule** which maintains isolation & consistency

Schedule

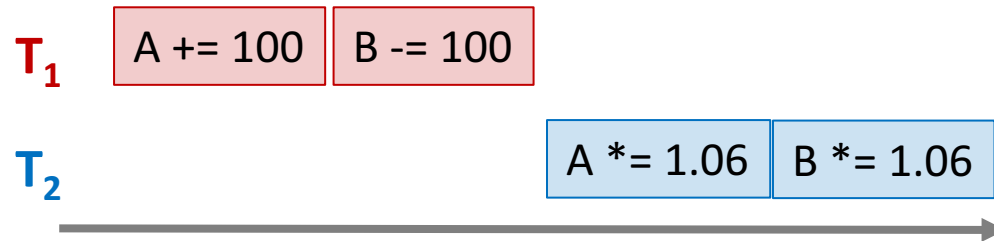
- A schedule is a list of actions
 - Reading (R)
 - Writing (W)
 - Aborting (A)
 - Committing (C)
- A schedule represents actual or potential execution sequence.

Scheduling examples

*Starting
Balance*

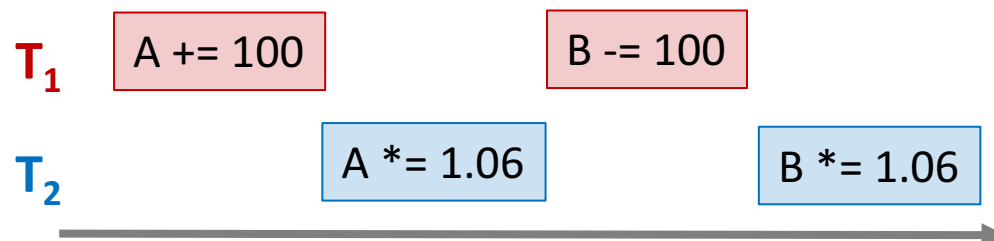
A	B
\$50	\$200

Serial schedule T_1, T_2 :



A	B
\$159	\$106

Interleaved schedule A:



A	B
\$159	\$106

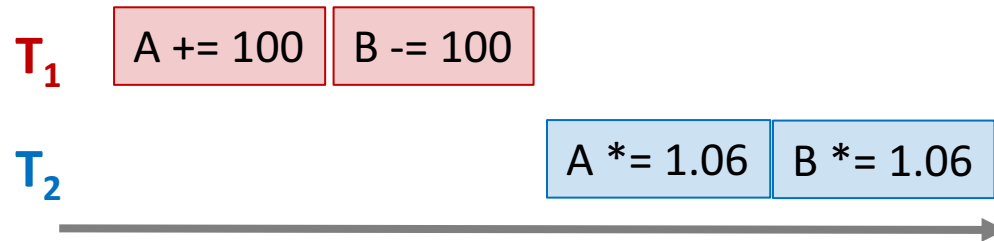
Same
result!

Scheduling examples

*Starting
Balance*

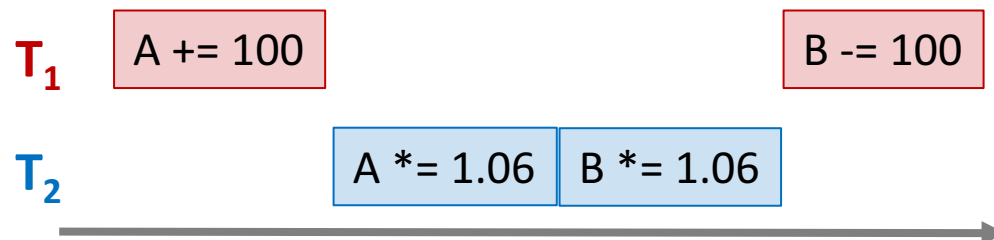
A	B
\$50	\$200

Serial schedule T_1, T_2 :



A	B
\$159	\$106

Interleaved schedule B:



A	B
\$159	\$112

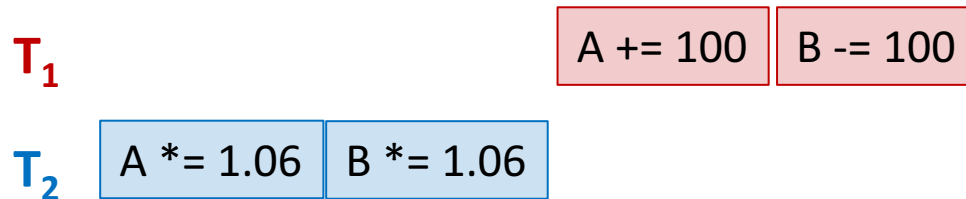
Different
result
than
serial
 T_1, T_2 !

Scheduling examples

Starting
Balance

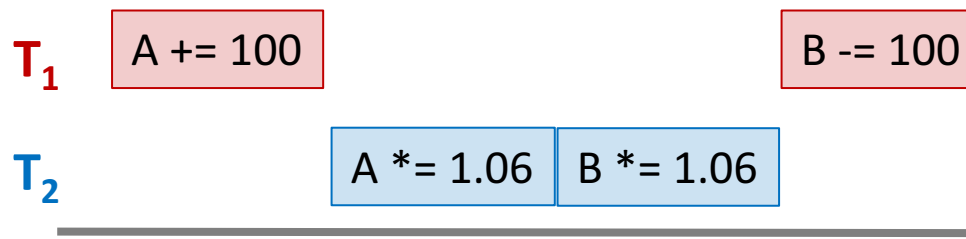
A	B
\$50	\$200

Serial schedule T_2, T_1 :



A	B
\$153	\$112

Interleaved schedule B:

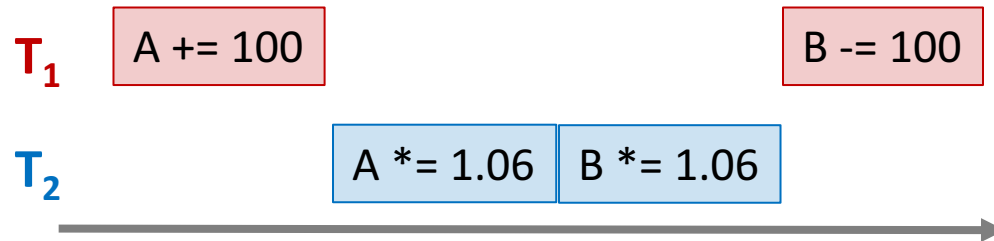


A	B
\$159	\$112

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!

Order of Execution

- Executing transactions in different order may produce different results
 - But all are presumed to be acceptable.
 - DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.

Serial schedule T_1T_2 :

T_1 A += 100 B -= 100


T_2 A *= 1.06 B *= 1.06



Serial schedule T_2T_1 :

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06



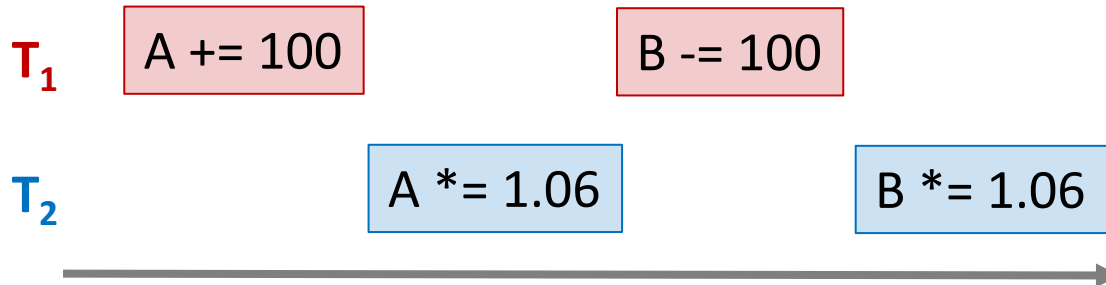
*Starting
Balance*

A	B
\$50	\$200

A	B
\$159	\$106

A	B
\$153	\$112

Serializable?



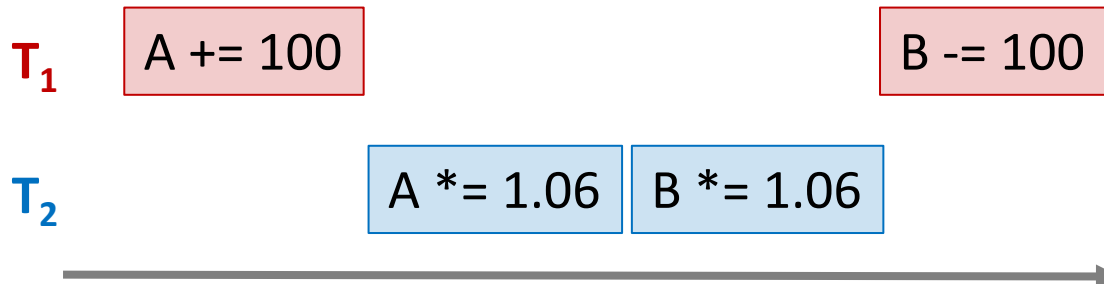
Serial schedules:

	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 = serializable***

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

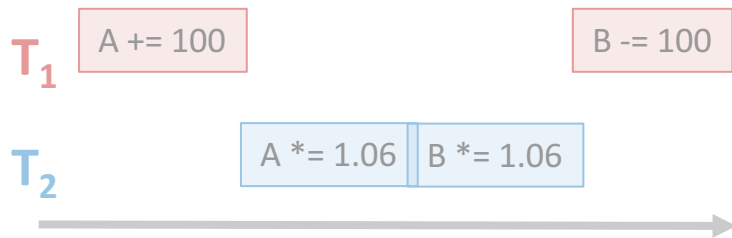
What else can go wrong with interleaving?

- Various anomalies which break isolation / serializability
 - Often referred to by name...

conflicts

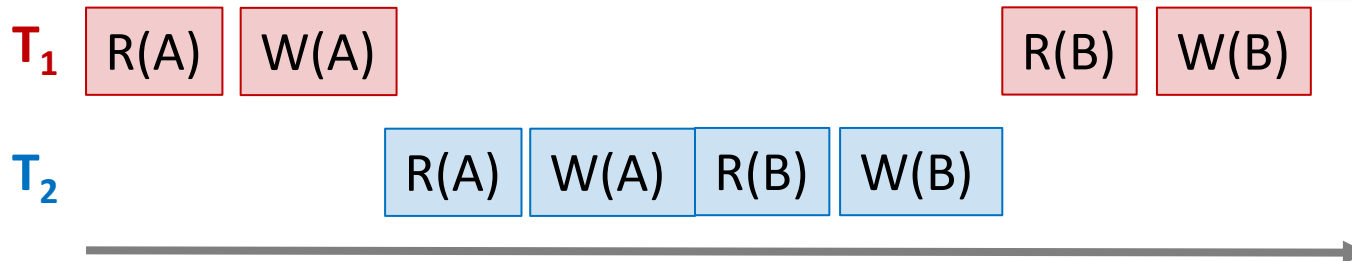
- Occur because of / with certain “conflicts” between interleaved TXNs

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 / object, 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)

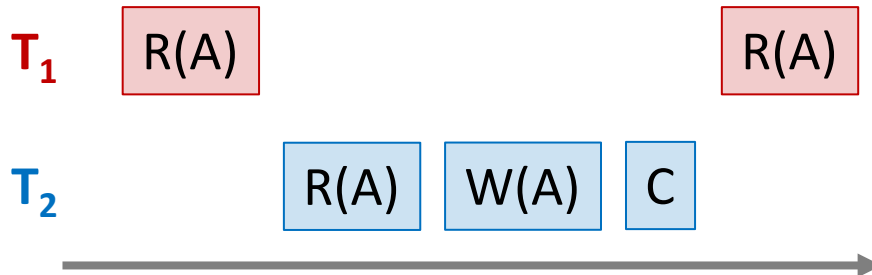
Why no “RR Conflict”?

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

Classic Anomalies with Interleaved Execution

“Unrepeatable read”:

Example:



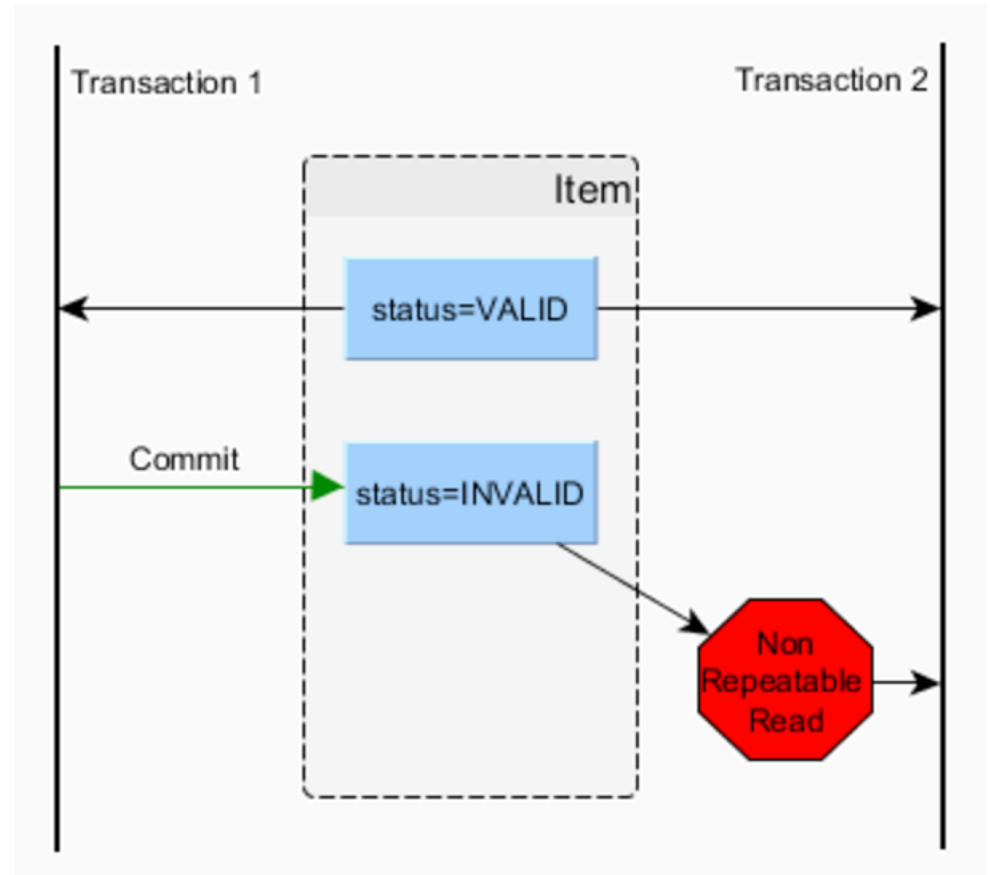
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again
and now gets a different / inconsistent value

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

Possible issue: Error due to integrity constraint

Unrepeatable Read (RW Conflicts)

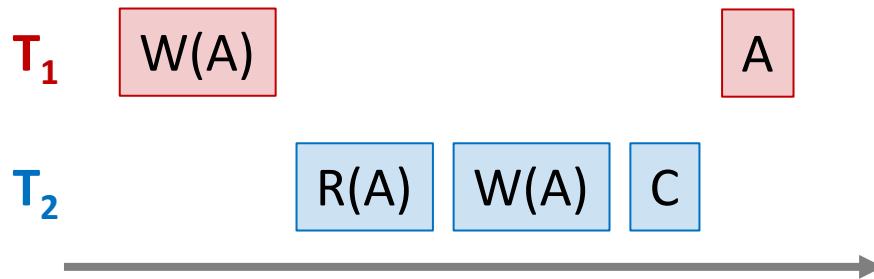
- A unrepeatable read manifests when consecutive reads yield different results due to a concurring transaction that has just updated the record we're reading.
- This is undesirable since we end up using stale data.
- This is prevented by holding a shared lock (read lock) on the read record for the whole duration of the current transaction.



Classic Anomalies with Interleaved Execution

“Dirty read” / Reading uncommitted data:

Example:



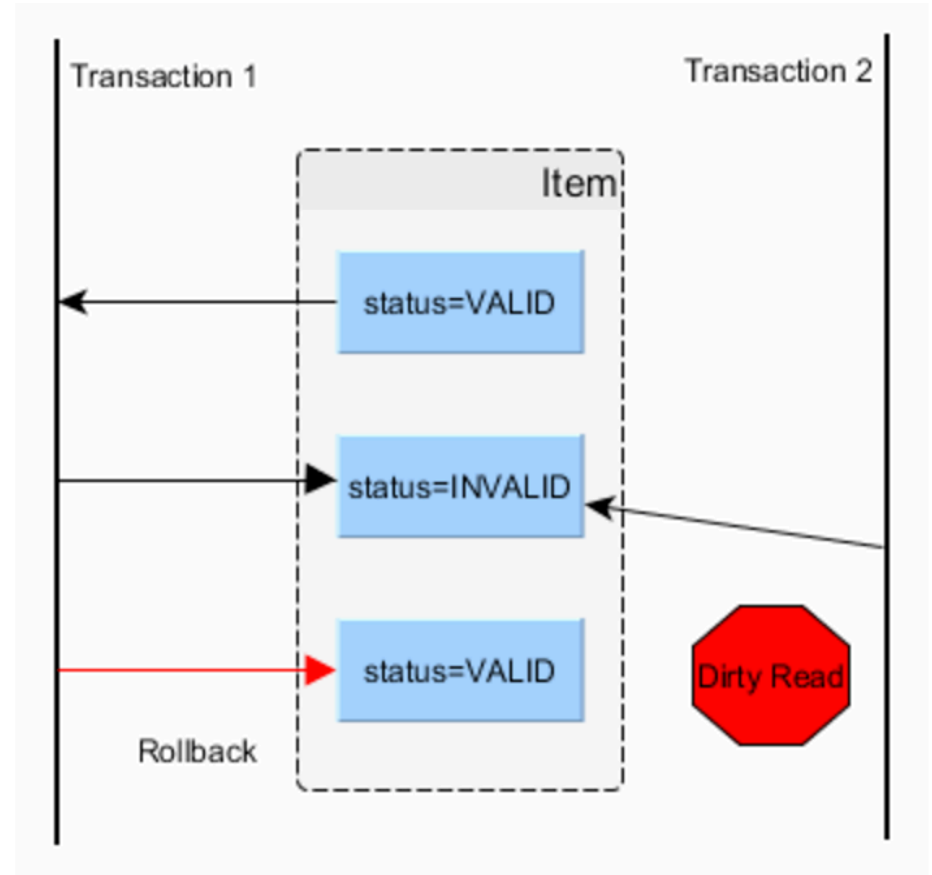
1. T_1 writes some data to A
2. T_2 reads from A , then writes back to A & commits
3. T_1 then aborts- *now T_2 's result is based on an obsolete / inconsistent value*

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

Problem: The value of A written by T_1 is read by T_2 before T_1 has completed all its changes.

Dirty Read (Reading Uncommitted Data) (WR Conflicts)

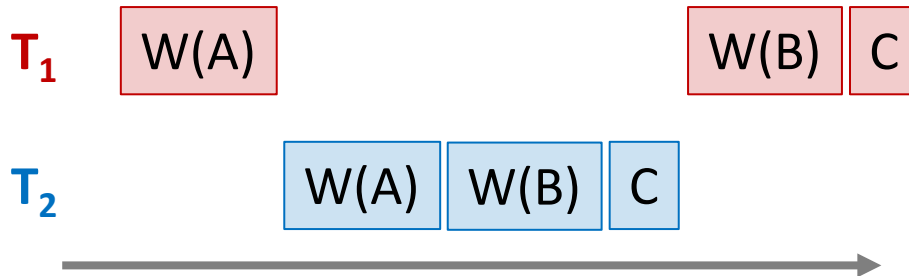
- A dirty read happens when a transaction is allowed to read uncommitted changes of some other running transaction.
- This happens because there is no locking preventing it.
- In the picture, you can see that the second transaction uses an inconsistent value as the first transaction is aborted.



Classic Anomalies with Interleaved Execution

Partially-lost update:

Example:

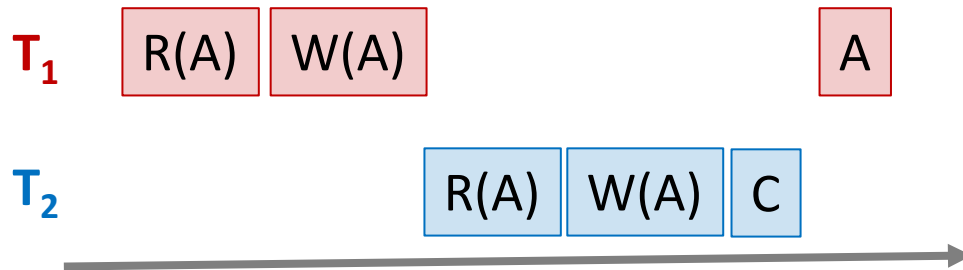


1. T_1 blind writes some data to A
2. T_2 blind writes to A and B
3. T_1 then blind writes to B; now we have T_2 's value for A and T_1 's value for B- **not equivalent to any serial schedule!**

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

Problem: T_1 's update ($W(A)$) is lost. T_2 's update ($W(B)$) is lost

Unrecoverable Schedule



1. T_1 reads and writes data to A
2. T_2 reads and writes data to A
3. T_2 commits
4. T_1 aborts.

In a recoverable schedule, transactions commit only after all transactions whose changes they read commit.

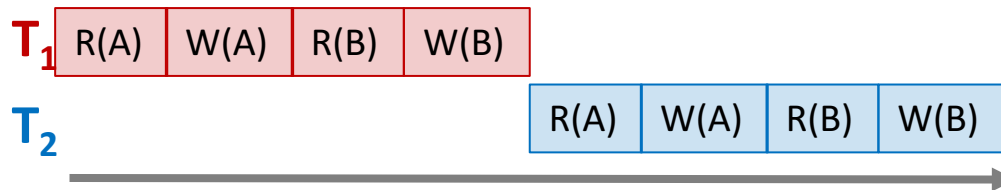
2. CONFLICT SERIALIZABILITY, LOCKING & DEADLOCK

What you will learn about in this section

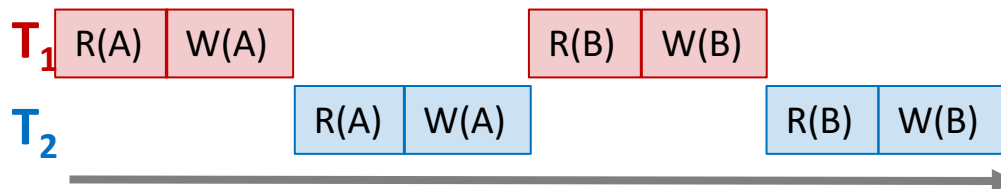
1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

Recall: Concurrency as Interleaving TXNs

Serial Schedule:



Interleaved Schedule:

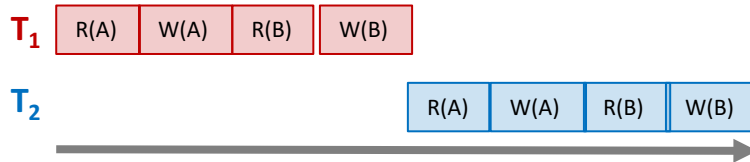


- For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

We call the particular order of interleaving a **schedule**

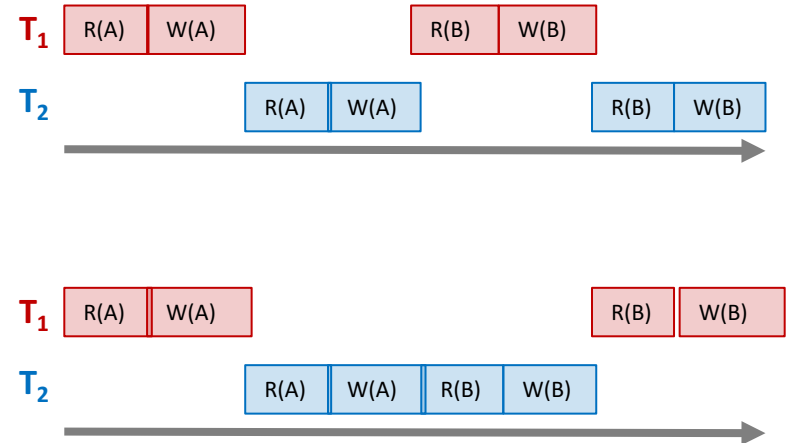
Recall: “Good” vs. “bad” schedules

Serial Schedule:



Why?

Interleaved Schedules:



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

Ways of Defining “Good” vs. “Bad” Schedules

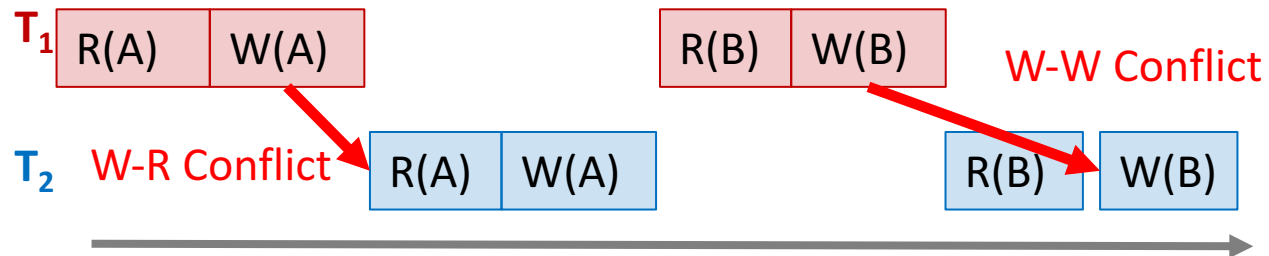
- Recall from last time: we call a schedule **serializable** if it is equivalent to *some* serial schedule
 - We used this as a notion of a “good” interleaved schedule, since **a serializable schedule will maintain isolation & consistency**
- Now, we’ll define a stricter, but very useful variant:

– ***Conflict serializability***

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

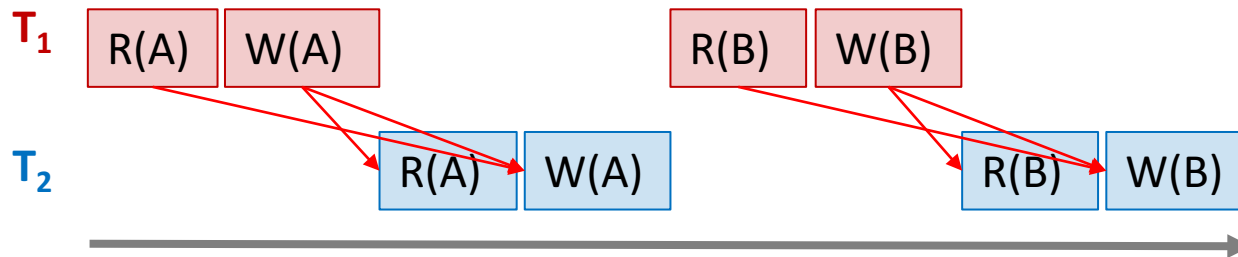
Conflicts

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”!

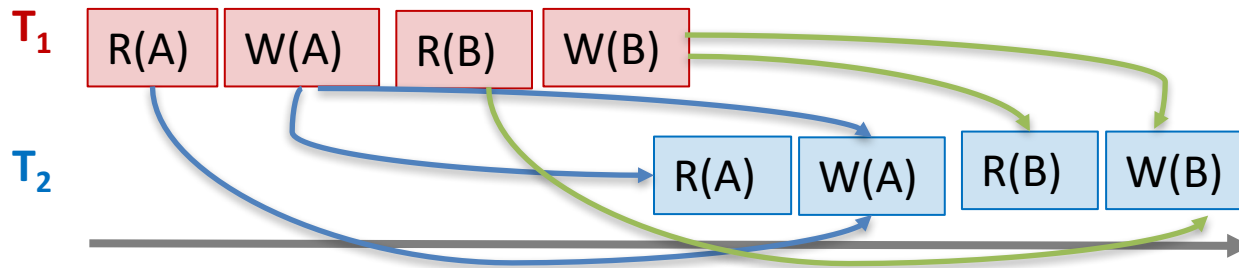
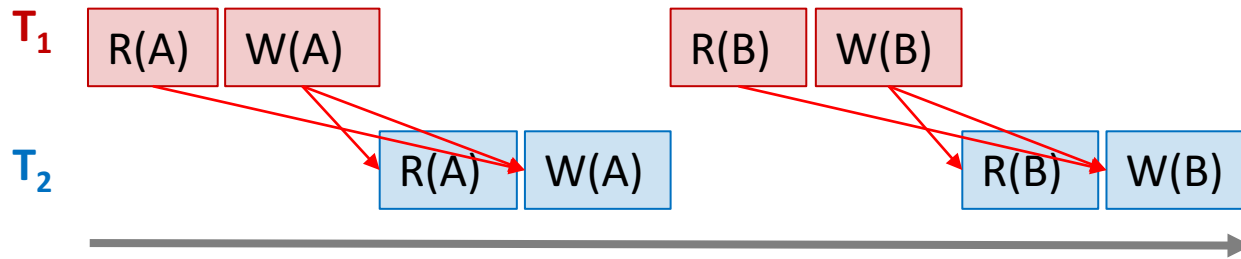
Conflict Serializability

- 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

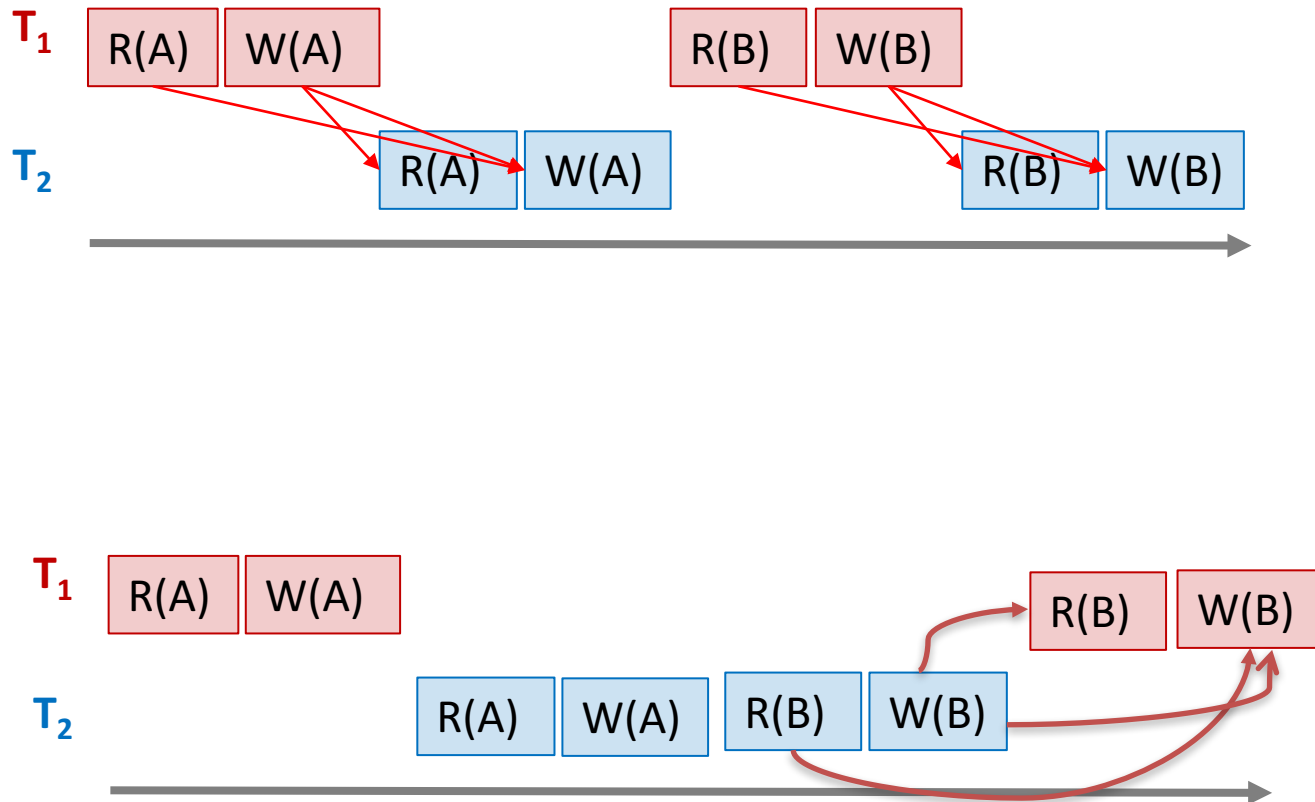
Conflict serializable \Rightarrow serializable

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

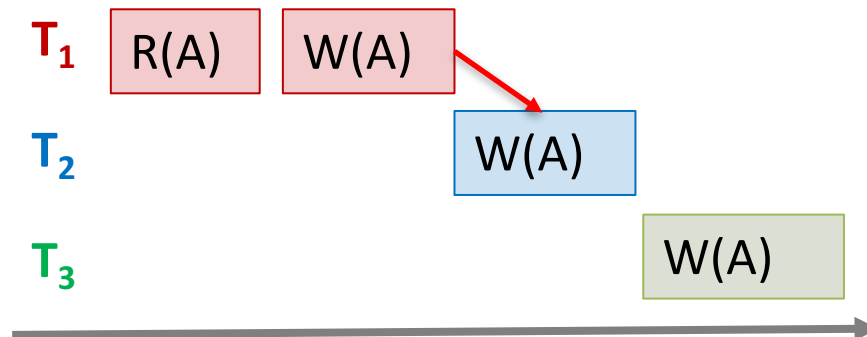
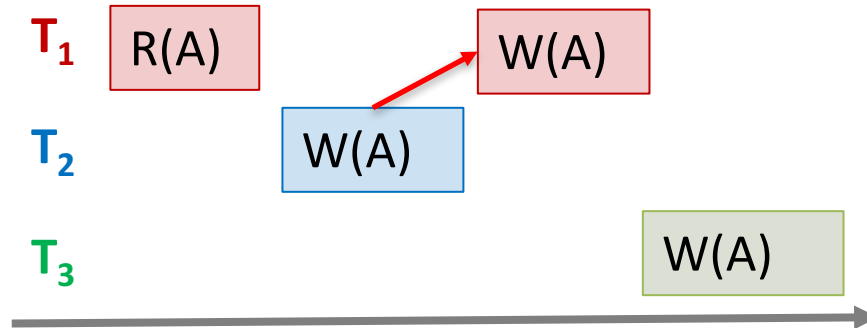
Conflict serializable



Not Conflict serializable



Example of Serializable Schedule that is not Conflict Serializable

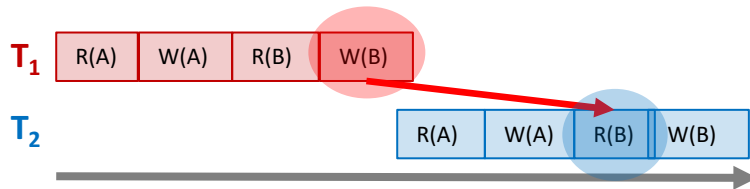


Serializable

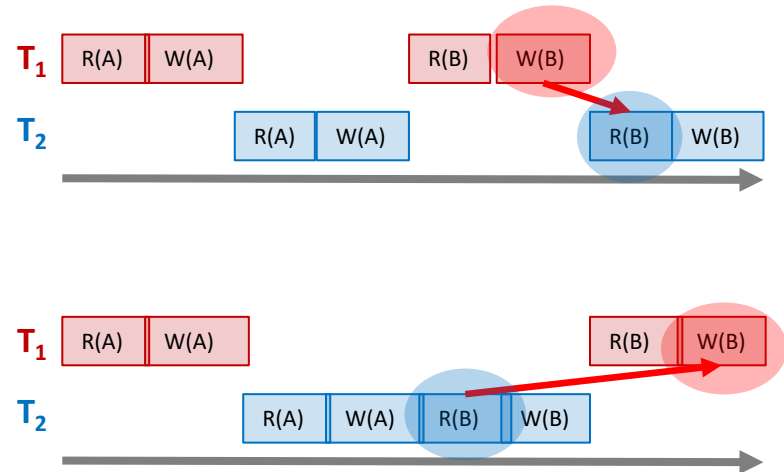
But
Not Conflict
Serializable

Recall: “Good” vs. “bad” schedules

Serial Schedule:



Interleaved Schedules:



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

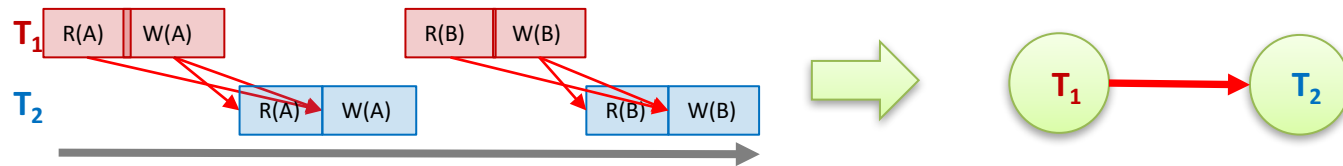
Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!

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

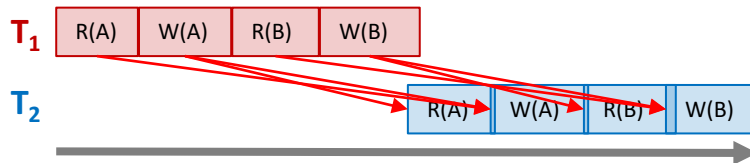
The Conflict / Precedence / Serializability Graph

- 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 any actions in T_i precede and conflict with any actions in T_j



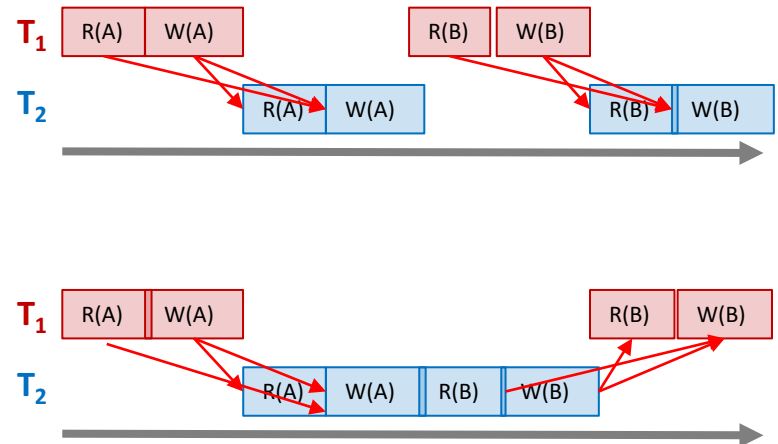
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



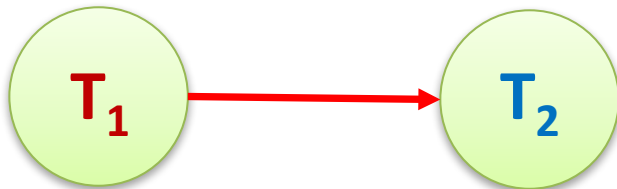
A bit complicated...

Interleaved Schedules:



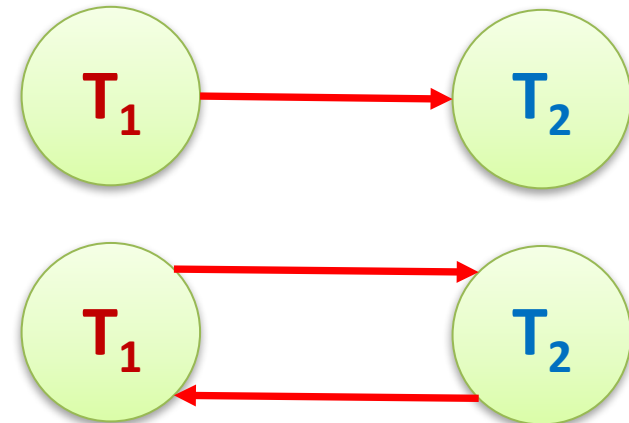
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



Simple!

Interleaved Schedules:



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

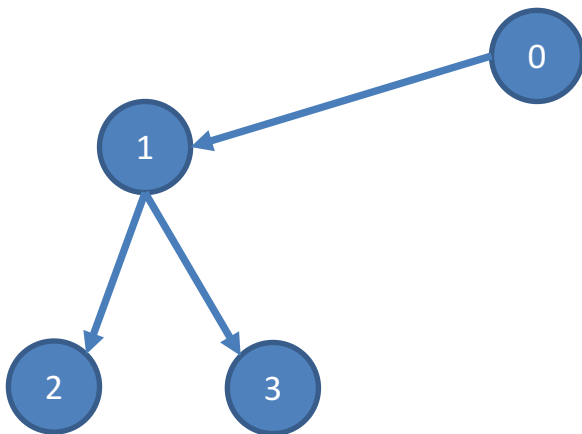
Let's unpack this notion of acyclic conflict graphs...

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 **acyclic** 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

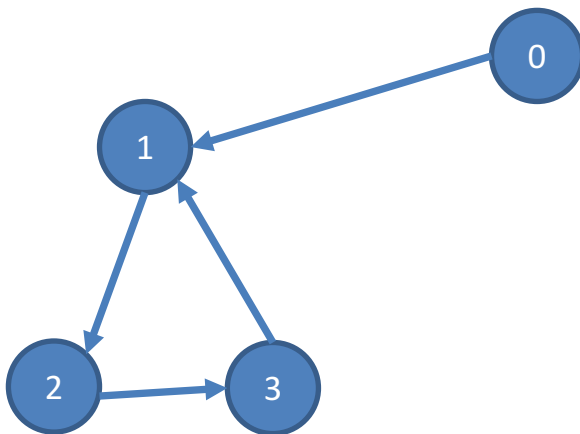
- Ex: What is one possible topological ordering here?



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 \rightarrow conflict serializable!

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

How to deal with concurrency



Locking

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!

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

- Rule 1:
 - If a transaction T wants to:
 - **Read** an object, it obtains a **shared** (S) lock on the object
 - **Write** an object, it obtains an **exclusive** (X) lock on the object
- Rule 2:
 - All locks held by a transaction are released when transaction is completed.

If a TXN holds a lock **S** , no other TXN can get a lock **X** on that object.

If a TXN holds a lock **X** , no other TXN can get a lock (**S** or **X**) on that object.

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 \Rightarrow serializable schedules

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

Deadlock Detection: Example



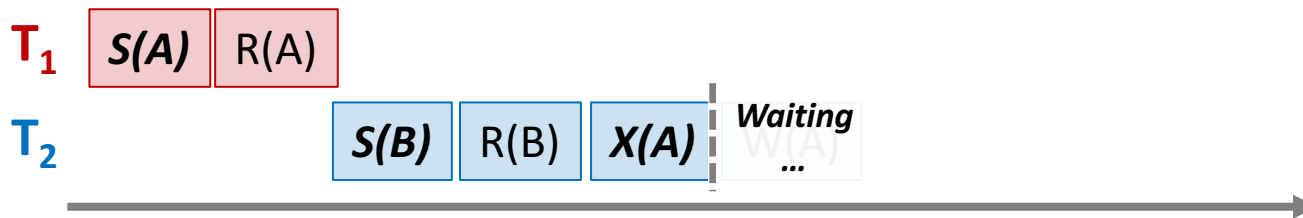
First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example



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

Deadlock Detection: Example

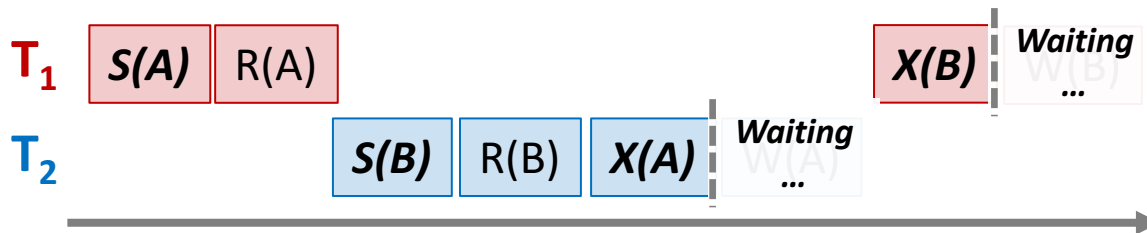


Waits-for graph:



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!

Performance of Locking

- Resolve conflicts between transactions and use two basic mechanisms:
 - Blocking
 - Aborting
- Both incurs performance penalty.
 - Blocking (Other transactions need to wait)
 - Aborting (Wastes the work done thus far)
- **Deadlock:**
 - Extreme instance of blocking
 - A set of transactions are forever blocked unless one of the deadlocked transactions is **aborted** by the DBMS

Deadlocks

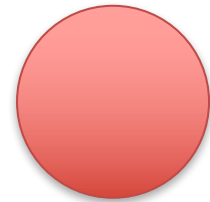
- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
 1. Deadlock prevention
 2. Deadlock avoidance

Deadlock Prevention

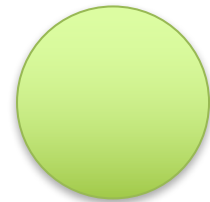
- Use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.
- Wait-Die Scheme
- Wound-Wait Scheme

Timestamp Ordering

- Each transaction is assigned a *unique* increasing timestamp
- *Earlier* transactions receives a *smaller* timestamp
- T_1 (**old**), T_2 , T_3 (**new**), ...
- Notation: Old Transaction T_{old} New Transaction T_{new}



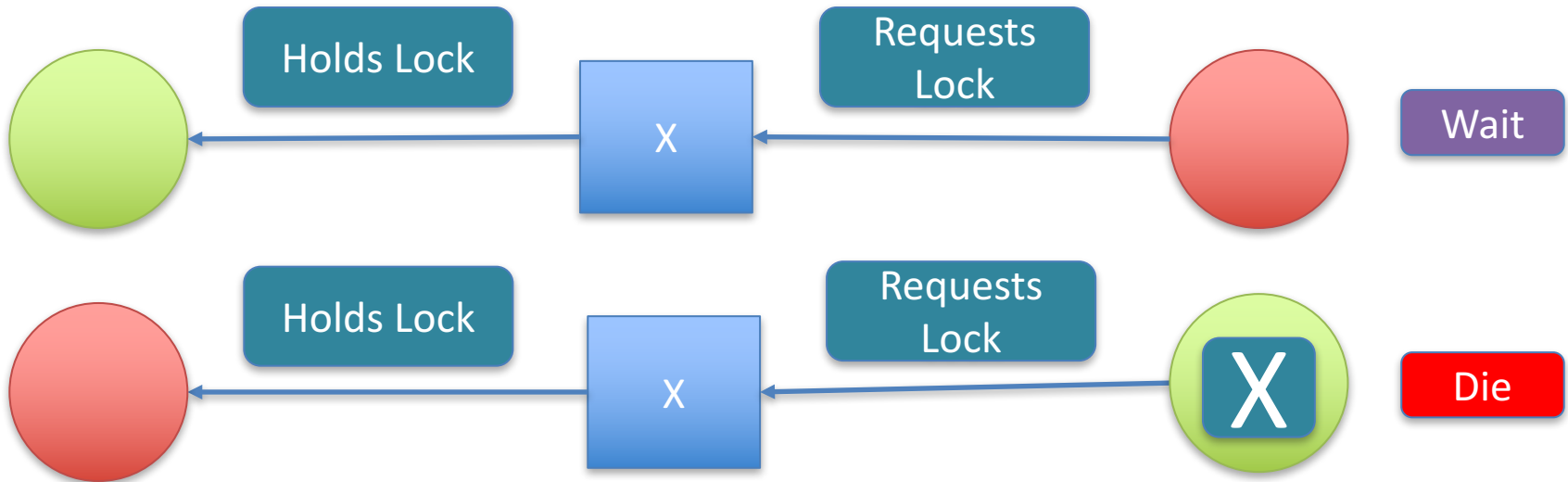
T_{old}



T_{new}

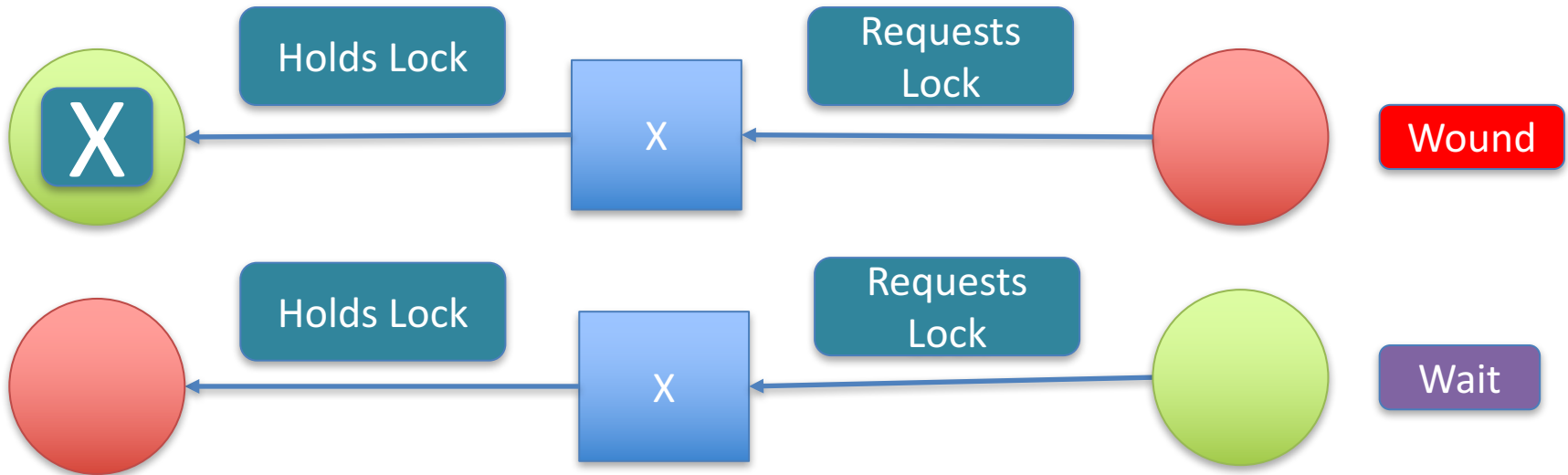
Wait-Die

T_{old} is allowed to *wait* for T_{new}
 T_{new} will *die* when it waits for T_{old}



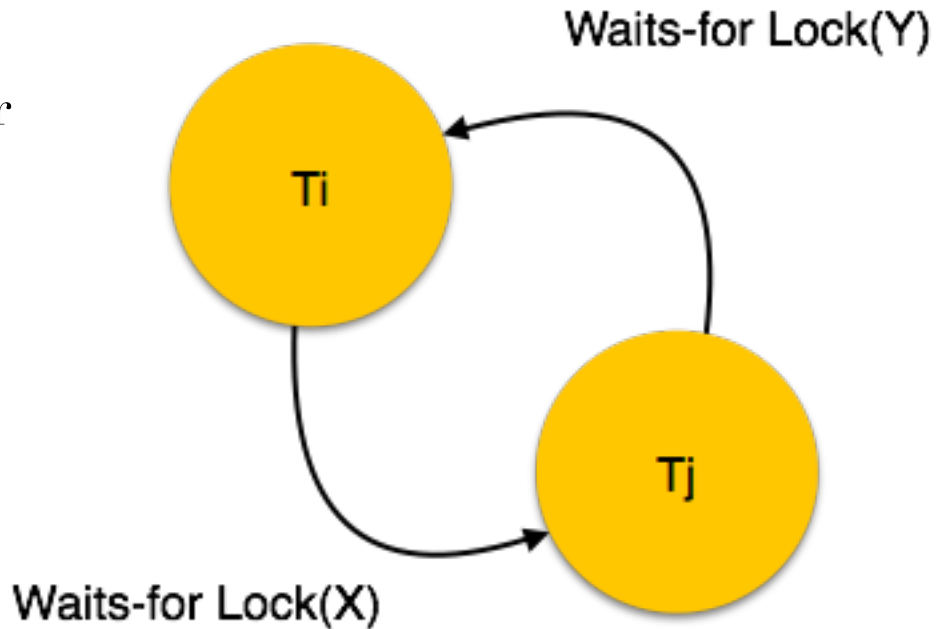
Wound Wait

T_{old} will wound T_{new}
 T_{new} waits for T_{old}



Deadlock Avoidance

- **Waits-for graph:**
 - For each transaction entering into the system, a node is created.
 - When a transaction T_i requests for a lock on an item, say X , which is held by some other transaction T_j , a directed edge is created from T_i to T_j .
 - If T_j releases item X , the edge between them is dropped and T_i locks the data item.
- The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.
- <https://vladmihalcea.com/2014/01/05/a-beginners-guide-to-acid-and-database-transactions/>