

# Lectures 7 & 8: Transactions

# Goals for this pair of lectures

- **Transactions** are a programming abstraction that enables the DBMS to handle *recovery* and *concurrency* for users.
- **Application:** Transactions are critical for users
  - Even casual users of data processing systems!
- **Fundamentals:** The basics of **how TXNs work**
  - Transaction processing is part of the debate around new data processing systems
  - Give you enough information to understand how TXNs work, and the main concerns with using them

# Lecture 7: Intro to Transactions & Logging

# Today's Lecture

1. Transactions
2. Properties of Transactions: ACID
3. Logging

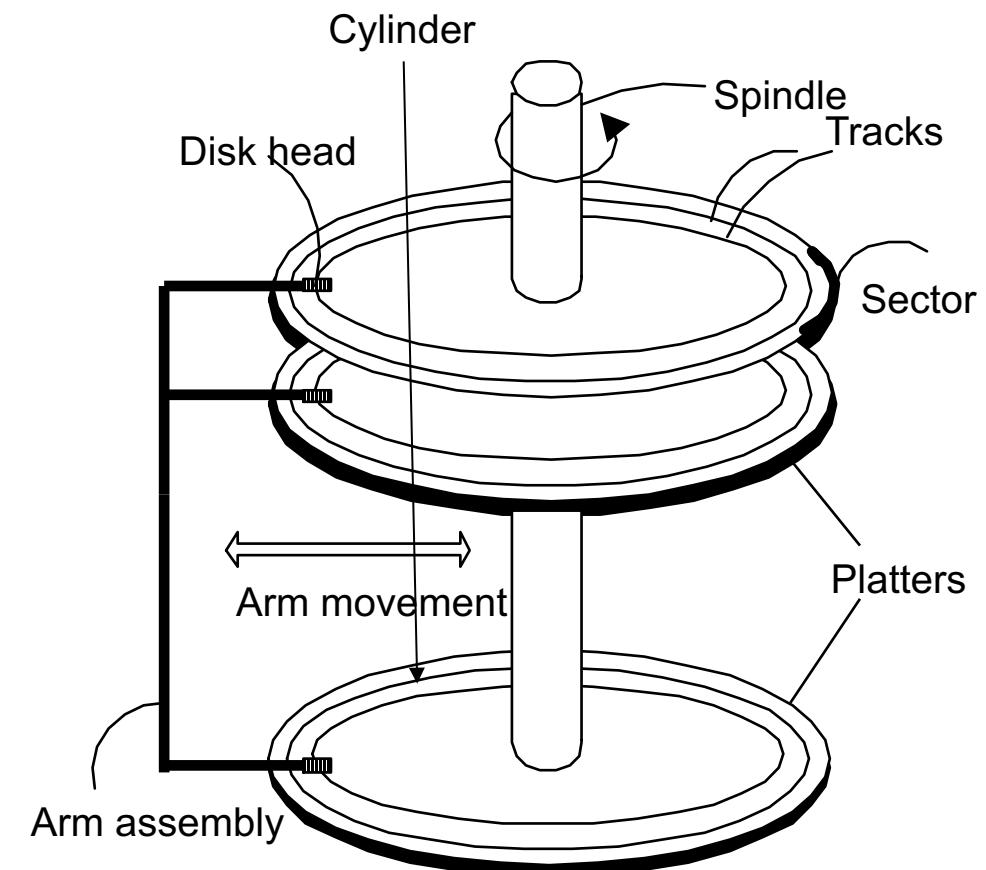
# 1. Transactions

# What you will learn about in this section

1. Our “model” of the DBMS / computer
2. Transactions basics
3. Motivation: Recovery & Durability
4. Motivation: Concurrency [*next lecture*]

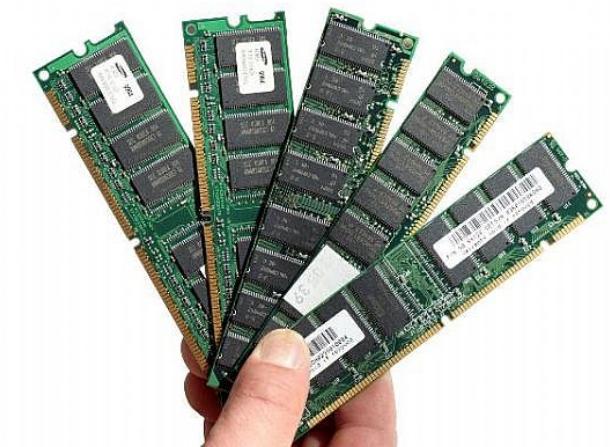
# High-level: Disk vs. Main Memory

- **Disk:**
  - *Slow*
    - Sequential access
      - (although fast sequential reads)
  - *Durable*
    - We will assume that once on disk, data is safe!
  - *Cheap*



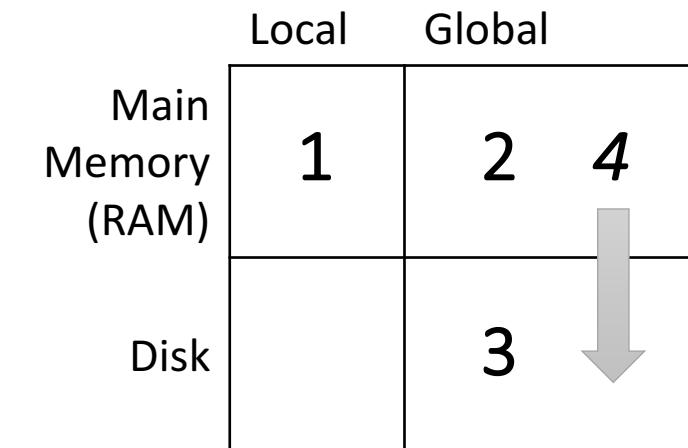
# High-level: Disk vs. Main Memory

- Random Access Memory (RAM) or **Main Memory**:
  - *Fast*
    - Random access, byte addressable
      - ~10x faster for sequential access
      - ~100,000x faster for random access!
  - *Volatile*
    - Data can be lost if e.g. crash occurs, power goes out, etc!
  - *Expensive*
    - For \$100, get 16GB of RAM vs. 2TB of disk!



# Our model: 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 from main memory

# High-level: Disk vs. Main Memory

- Keep in mind the tradeoffs here as motivation for the mechanisms we introduce
  - Main memory: fast but limited capacity, volatile
  - Vs. Disk: slow but large capacity, durable

How do we effectively utilize *both* ensuring certain critical guarantees?

# Transactions

# Transactions: Basic Definition

A transaction (“TXN”) is a sequence of one or more *operations* (reads or writes) which reflects *a single real-world transition*.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION  
    UPDATE Product  
        SET Price = Price - 1.99  
        WHERE pname = 'Gizmo'  
    COMMIT
```

# Transactions: Basic Definition

A transaction (“TXN”) is a sequence of one or more *operations* (reads or writes) which reflects a *single real-world transition*.

In the real world, a TXN either happened completely or not at all

## Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

# Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
    UPDATE Bank SET amount = amount - 100
    WHERE name = 'Bob'
    UPDATE Bank SET amount = amount + 100
    WHERE name = 'Joe'
COMMIT
```

# Model of Transaction for CS 145

*Note:* For 145, we assume that the DBMS *only* sees  
reads and writes to data

- User may do much more
- In real systems, databases do have more info...

# Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.
2. **Concurrency**: Achieving better performance by parallelizing TXNs *without* creating anomalies

This lecture!

Next lecture

# Motivation

**1. Recovery & Durability** of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either durably stored in full, or not at all; keep log to be able to “roll-back” TXNs

# Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)  
    SELECT pname, price  
        FROM Product  
    WHERE price <= 0.99
```

Crash / abort!

```
DELETE Product  
    WHERE price <=0.99
```

What goes wrong?

# Protection against crashes / aborts

Client 1:

```
START TRANSACTION
    INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

    DELETE Product
        WHERE price <=0.99
COMMIT OR ROLLBACK
```

Now we'd be fine! We'll see how / why this lecture

# Motivation

**2. Concurrent** execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming...

# Multiple users: single statements

```
Client 1: UPDATE Product  
          SET Price = Price - 1.99  
          WHERE pname = 'Gizmo'
```

```
Client 2: UPDATE Product  
          SET Price = Price*0.5  
          WHERE pname='Gizmo'
```

Two managers attempt to discount products *concurrently*-  
What could go wrong?

# Multiple users: single statements

```
Client 1: START TRANSACTION
          UPDATE Product
          SET Price = Price - 1.99
          WHERE pname = 'Gizmo'
          COMMIT

Client 2: START TRANSACTION
          UPDATE Product
          SET Price = Price*0.5
          WHERE pname='Gizmo'
          COMMIT
```

Now works like a charm- we'll see how / why next lecture...

## 2. Properties of Transactions

# What you will learn about in this section

1. Atomicity

2. Consistency

3. Isolation

4. Durability

# Transaction Properties: ACID

- **Atomic**
  - State shows either all the effects of txn, or none of them
- **Consistent**
  - Txn moves from a state where integrity holds, to another where integrity holds
- **Isolated**
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **Durable**
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

# ACID: Atomicity

- TXN's activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

# ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can't be negative
    - Sum of *debits* and of *credits* is 0
- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

# ACID: Isolation

- A transaction executes concurrently with other transactions
- **Isolation:** the effect is as if each transaction executes in *isolation* of the others.
  - E.g. Should not be able to observe changes from other transactions during the run

# ACID: Durability

- The effect of a TXN must continue to exist (“*persist*”) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc...
- Means: Write data to **disk**

Change on the horizon?  
Non-Volatile Ram (NVRam).  
Byte addressable.

# Challenges for ACID properties

- In spite of failures: Power failures, but not media failures
- Users may abort the program: need to “rollback the changes”
  - Need to *log* what happened
- Many users executing concurrently
  - Can be solved via locking (we’ll see this next lecture!)

This lecture

Next lecture

And all this with... Performance!!

# A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**
- Many newer “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs...



ACID is an extremely important & successful paradigm, but still debated!

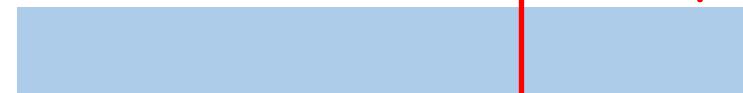
# Goal for this lecture: Ensuring Atomicity & Durability

ACID

- Atomicity:

- TXNs should either happen completely or not at all
- If abort / crash during TXN, *no* effects should be seen

TXN 1



*No changes persisted*

- Durability:

- If DBMS stops running, changes due to completed TXNs should all persist
- *Just store on stable disk*

TXN 2



*All changes persisted*

We'll focus on how to accomplish atomicity (via logging)

# The Log

- Is a list of modifications
- Log is *duplexed* and *archived* on stable storage.
- Can **force write** entries to disk
  - A page goes to disk.
- All log activities *handled transparently* the DBMS.

Assume we  
don't lose it!

# Basic Idea: (Physical) Logging

- Record UNDO information for every update!
  - Sequential writes to log
  - Minimal info (diff) written to log
- The **log** consists of an **ordered list of actions**
  - Log record contains:  
**<XID, location, old data, new data>**

This is sufficient to UNDO any transaction!

# Why do we need logging for atomicity?

- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work...*
- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!

...And so we need a **log** to be able to *undo* these partial results!

# 3. Atomicity & Durability via Logging

# What you will learn about in this section

1. Logging: An animation of commit protocols

# A Picture of Logging

# A picture of logging

$T: R(A), W(A)$



# A picture of logging

$T: R(A), W(A)$

$A: 0 \rightarrow 1$



# A picture of logging

$T: R(A), W(A)$

$A: 0 \rightarrow 1$



Operation recorded in log in main memory!



# What is the correct way to write this all to disk?

- We'll look at the *Write-Ahead Logging (WAL)* protocol
- We'll see why it works by looking at other protocols which are incorrect!

Remember: Key idea is to ensure durability  
while maintaining our ability to “undo”!

# Write-Ahead Logging (WAL) TXN Commit Protocol

# Transaction Commit Process

1. FORCE Write **commit** record to log
2. All log records up to last update from this TX are FORCED
3. Commit() returns

Transaction is committed *once commit log record is on stable storage*

# Incorrect Commit Protocol #1

T: R(A), W(A)

A: 0 → 1



Let's try committing  
before we've written  
either data or log to  
disk...

**OK, Commit!**

If we crash now, is T  
durable?

**Lost T's update!**

# Incorrect Commit Protocol #2

T: R(A), W(A)

A: 0 → 1



Let's try committing  
*after* we've written  
data but *before* we've  
written log to disk...

**OK, Commit!**

If we crash now, is T  
durable? Yes! Except...

**How do we know  
whether T was  
committed??**

# Improved Commit Protocol (WAL)

# Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



This time, let's try committing after we've written log to disk but before we've written data to disk... this is WAL!

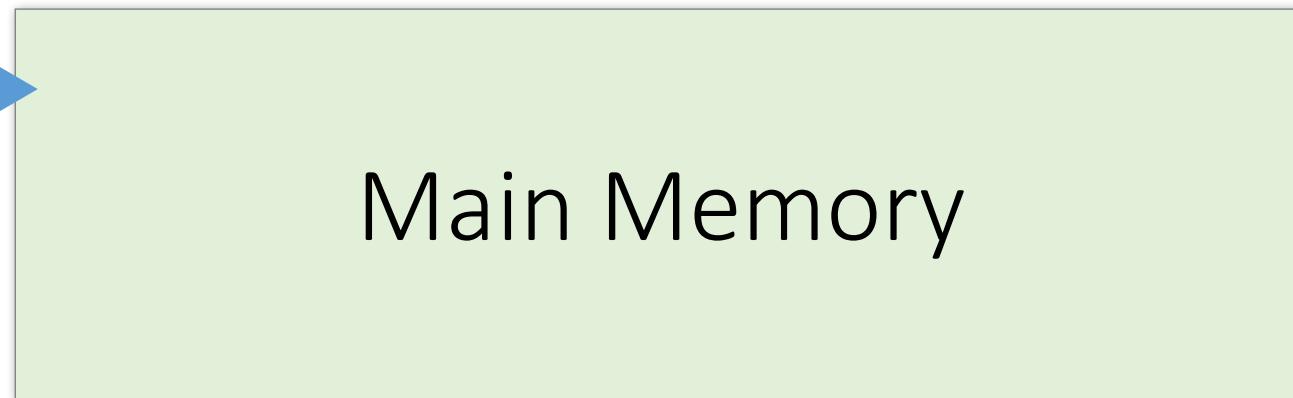
**OK, Commit!**

If we crash now, is T durable?

# Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

T →



A: 0 → 1



This time, let's try committing after we've written log to disk but before we've written data to disk... this is WAL!

**OK, Commit!**

If we crash now, is T durable?

**USE THE LOG!**

# Write-Ahead Logging (WAL)

- DB uses **Write-Ahead Logging (WAL)** Protocol:

Each update is logged! Why not reads?

1. Must *force log record* for an update *before* the corresponding data page goes to storage

→ Atomicity

2. Must *write all log records* for a TX *before commit*

→ Durability

# Logging Summary

- If DB says TX **commits**, TX effect **remains** after database crash
- DB can **undo actions** and help us with **atomicity**
- This is only half the story...

# Lecture 8: Concurrency & Locking

# 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
3. ACTIVITY: TXN viewer

# 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

## 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

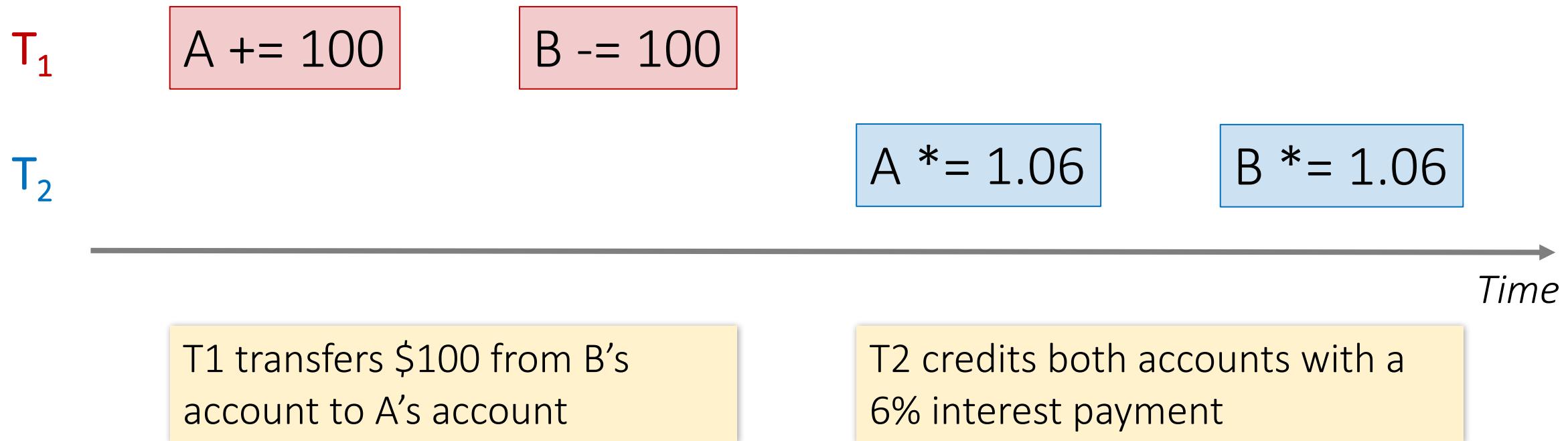
    SET Amt = Amt \* 1.06

    COMMIT

T2 credits both accounts with a 6%  
interest payment

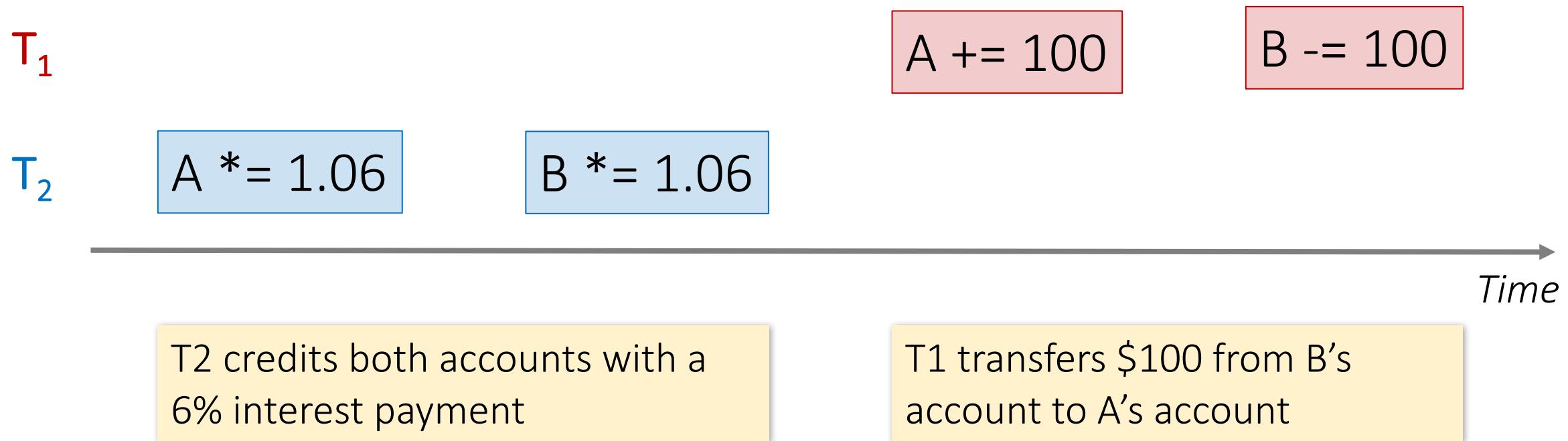
# Example- consider two TXNs:

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



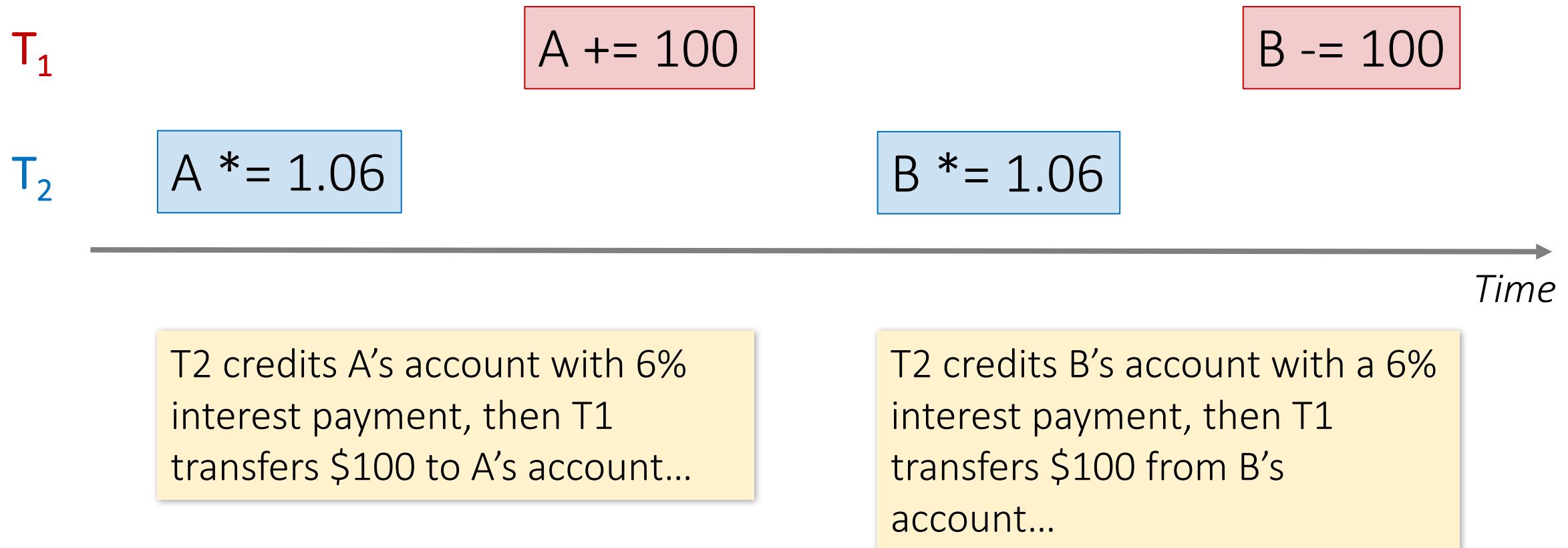
# Example- consider two TXNs:

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



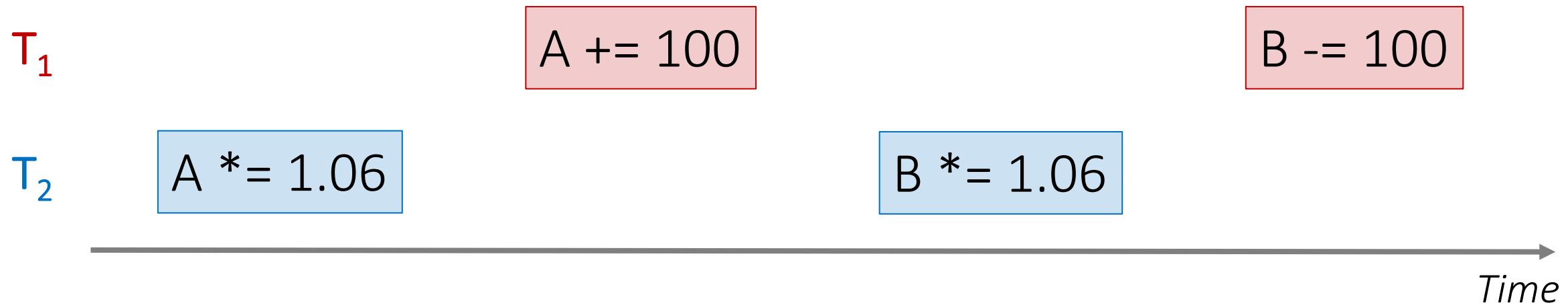
# Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



# Example- consider two TXNs:

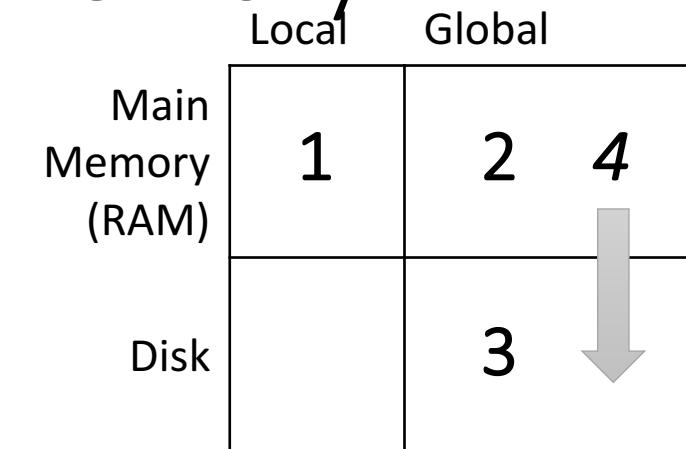
The DBMS can also **interleave** the TXNs



What goes wrong here??

# 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”

**ACID**

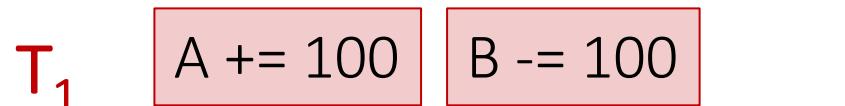
DBMS must pick a schedule which maintains isolation & consistency

# Scheduling examples

*Starting  
Balance*

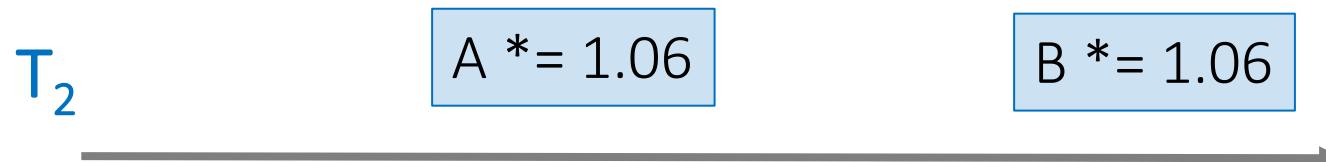
A	B
\$50	\$200

Serial schedule  $T_1, T_2$ :



A	B
\$159	\$106

Interleaved schedule A:



A	B
\$159	\$106

Same  
result!

# Scheduling examples

*Starting  
Balance*

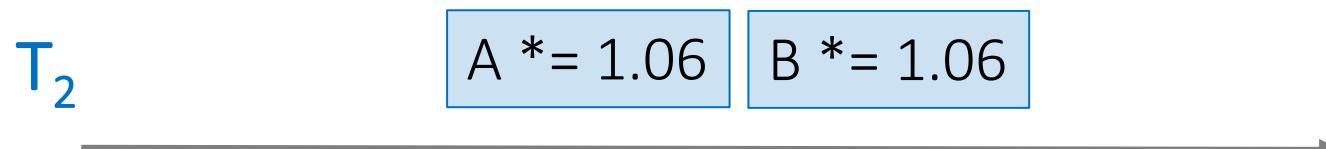
A	B
\$50	\$200

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  $T_2, T_1$ :

$T_1$

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

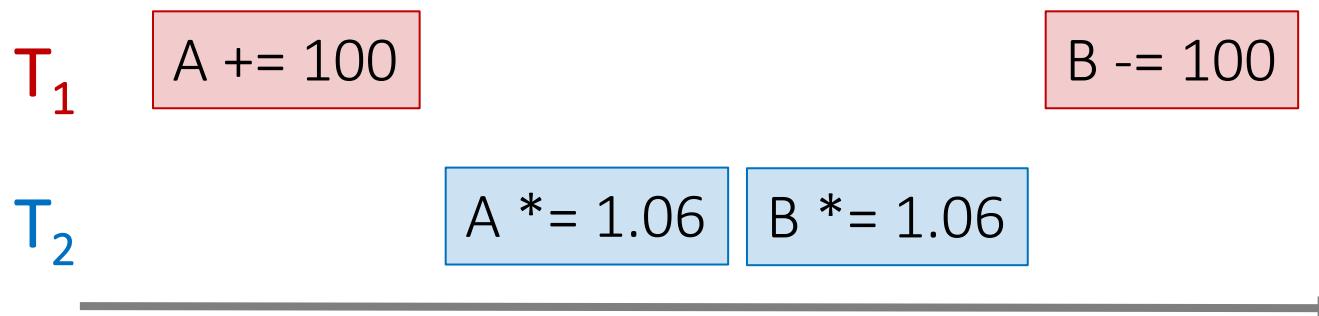
B

\$159      \$112

Different result than serial  $T_2, T_1$   
ALSO!

# Scheduling examples

Interleaved schedule B:



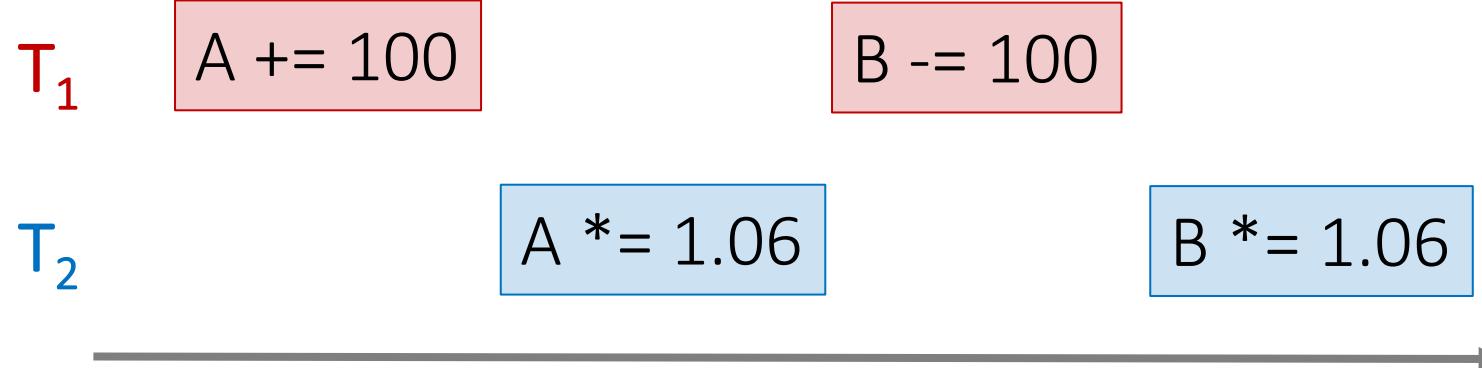
This schedule is different than *any serial order!* We say that it is not serializable

# 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?



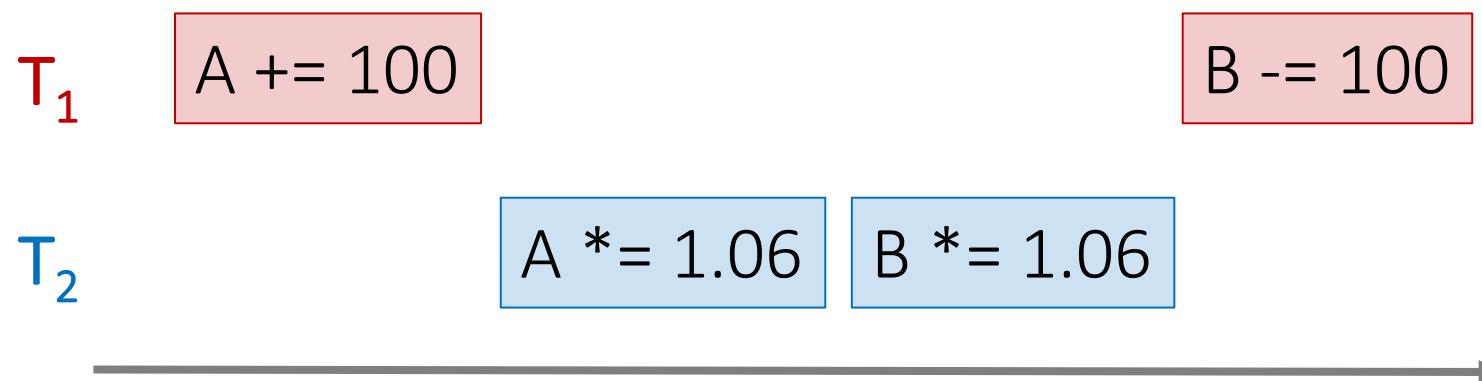
Serial schedules:

	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*  
 $A, B = \underline{\text{Serializable}}$

# Serializable?



Serial schedules:

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

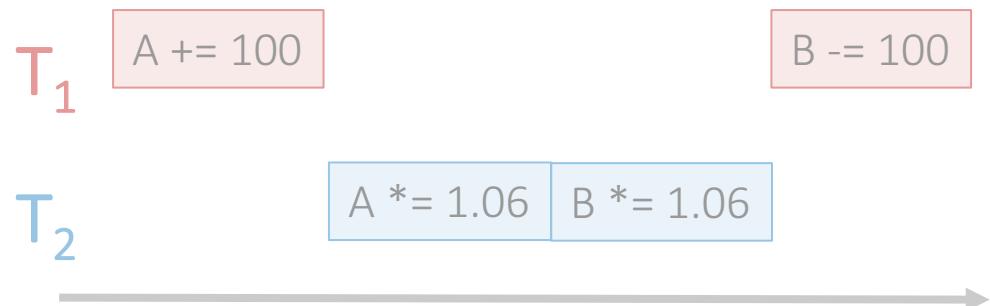
$A$	$B$
$1.06*(A+100)$	$1.06*B - 100$

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

# What else can go wrong with interleaving?

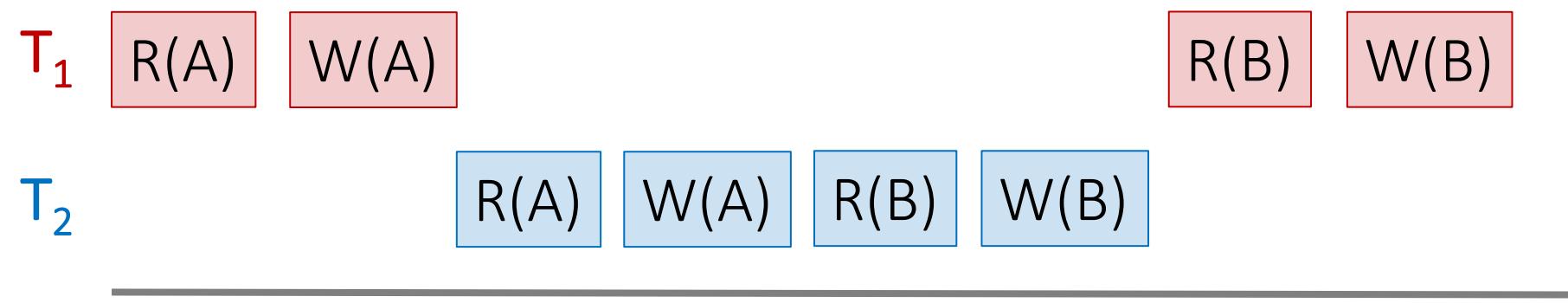
- Various anomalies which break isolation / serializability
  - Often referred to by name...
- Occur because of / with certain “conflicts” between interleaved TXNs

# 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”?

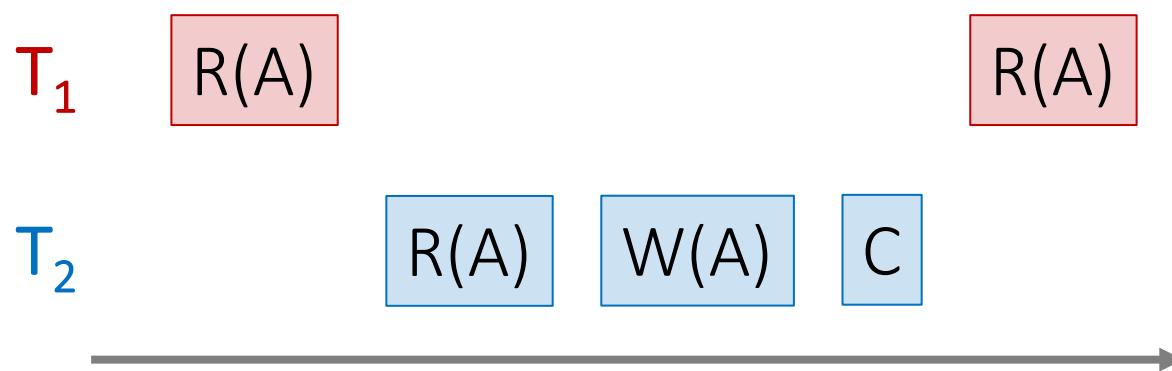
Interleaving anomalies occur with / because of these conflicts between TXNs (*but these conflicts can occur without causing anomalies!*)

See next section for more!

# Classic Anomalies with Interleaved Execution

“Unrepeatable read”:

Example:



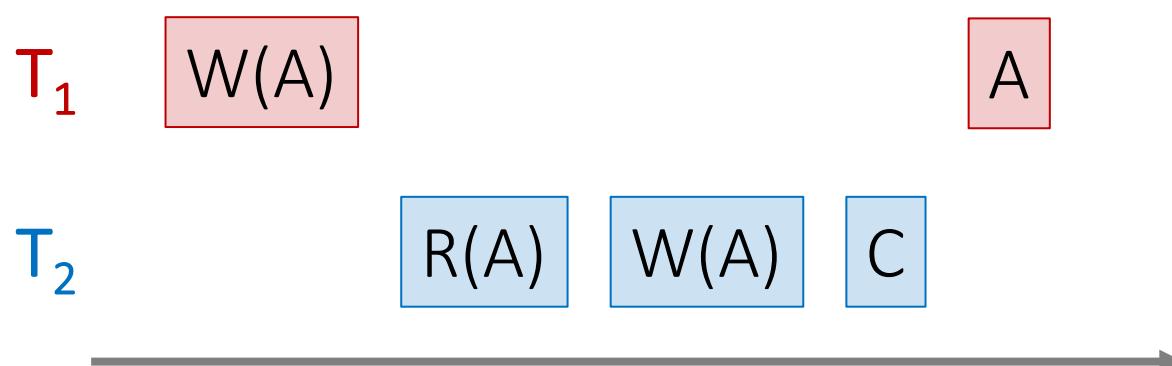
1.  $T_1$  reads some data from A
2.  $T_2$  writes to A
3. Then,  $T_1$  reads from A again  
*and now gets a different / inconsistent value*

*Occurring with / because of a RW conflict*

# Classic Anomalies with Interleaved Execution

“Dirty read” / Reading uncommitted data:

Example:



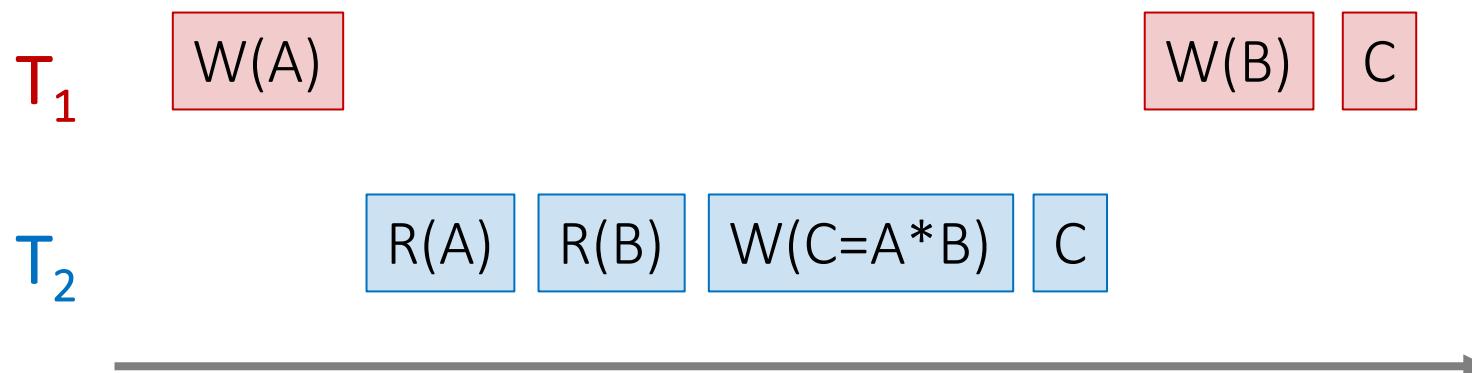
1.  $T_1$  writes some data to  $A$
2.  $T_2$  reads from  $A$ , then writes back to  $A$  & commits
3.  $T_1$  then aborts- *now  $T_2$ 's result is based on an obsolete / inconsistent value*

*Occurring with / because of a WR conflict*

# Classic Anomalies with Interleaved Execution

“Inconsistent read” / Reading partial commits:

Example:



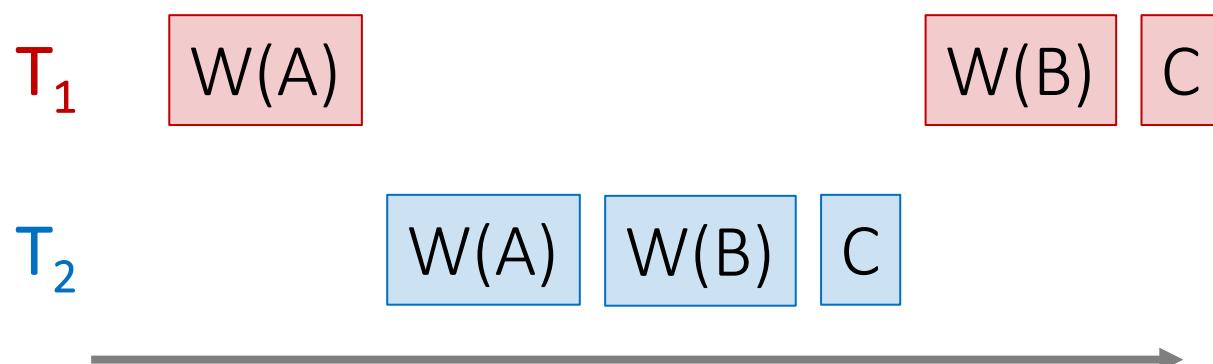
1.  $T_1$  writes some data to A
2.  $T_2$  reads from A and B, and then writes some value which depends on A & B
3.  $T_1$  then writes to B- now  $T_2$ 's result is based on an incomplete commit

*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*

[Activity-8-1.ipynb](#)

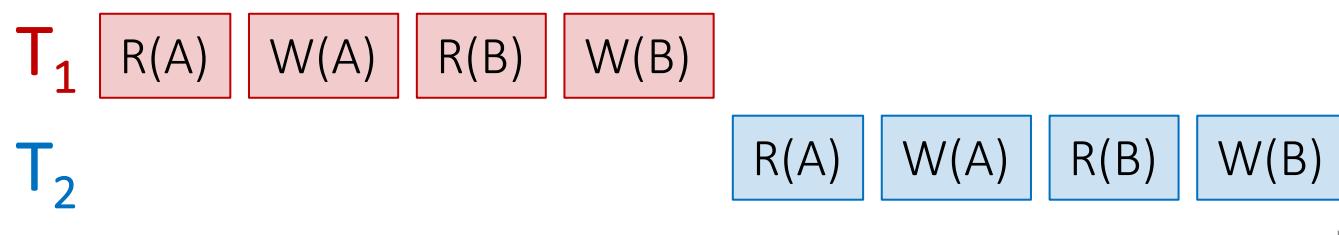
## 2. Conflict Serializability, Locking & Deadlock

# What you will learn about in this section

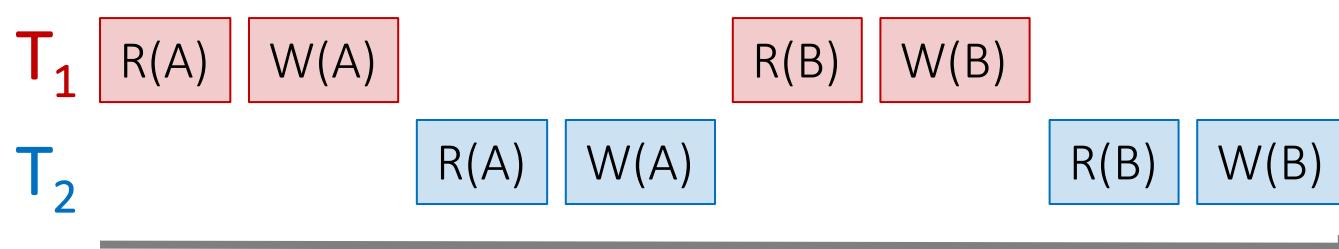
1. RECAP: Concurrency
2. Conflict Serializability
3. DAGs & Topological Orderings
4. Strict 2PL
5. Deadlocks

# Recall: Concurrency as Interleaving TXNs

## Serial Schedule:



## Interleaved Schedule:

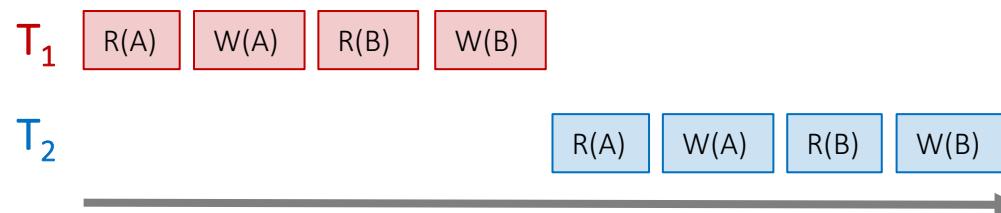


- For our purposes, having TXNs occur concurrently means **interleaving their component actions (R/W)**

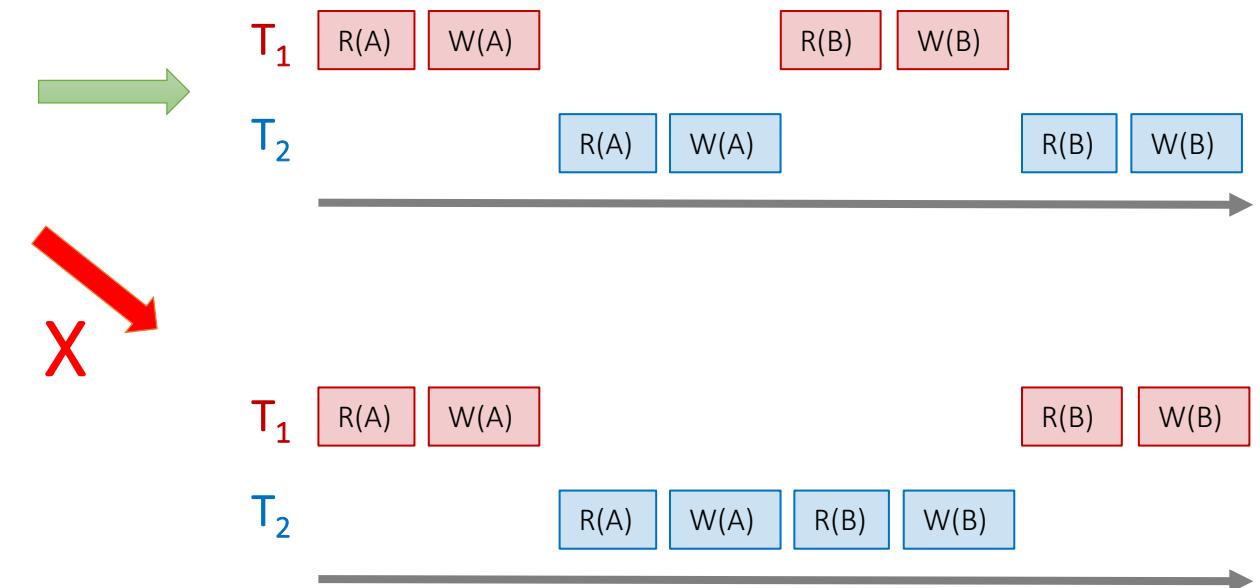
We call the particular order of interleaving a **schedule**

# Recall: “Good” vs. “bad” schedules

## Serial Schedule:



## Interleaved Schedules:



Why?

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

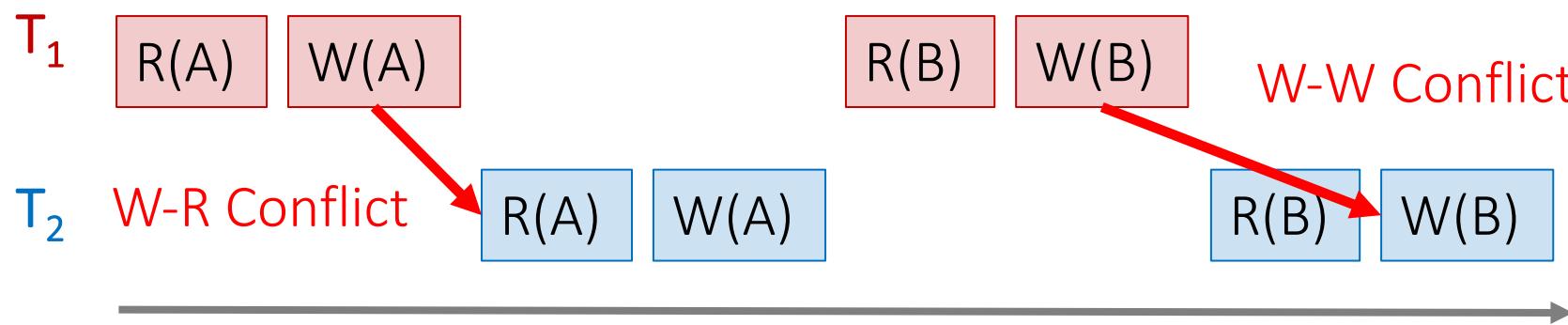
# Ways of Defining “Good” vs. “Bad” Schedules

- 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..

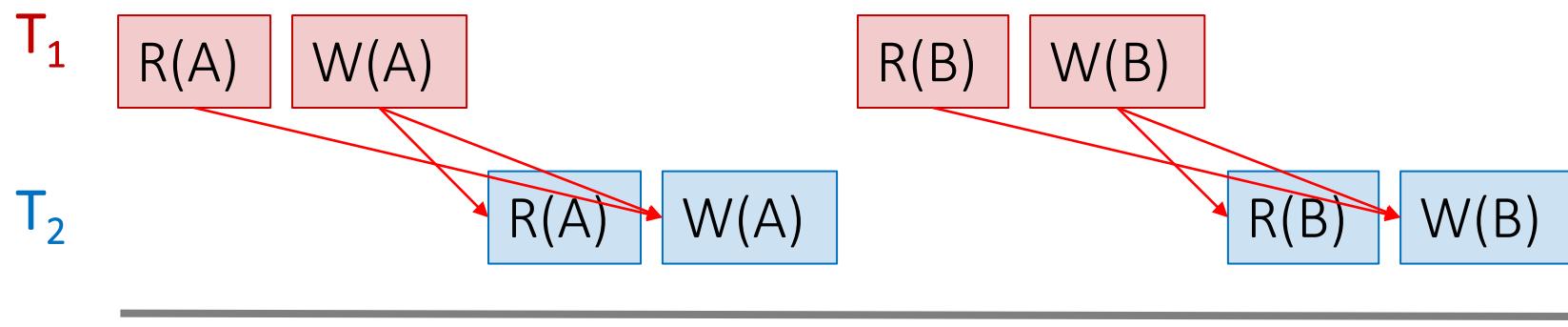
# Conflicts

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

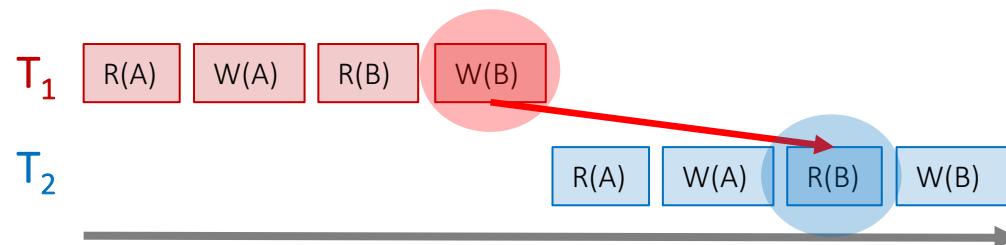
- Two schedules are **conflict equivalent** if:
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

Conflict serializable  $\Rightarrow$  serializable

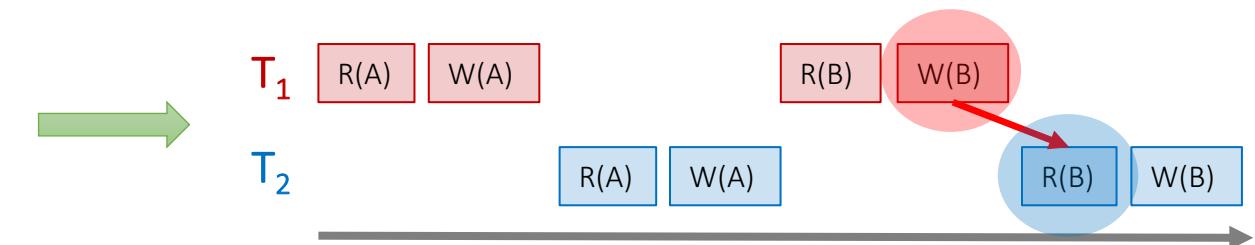
So if we have conflict serializable, we have consistency & isolation!

# 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!*



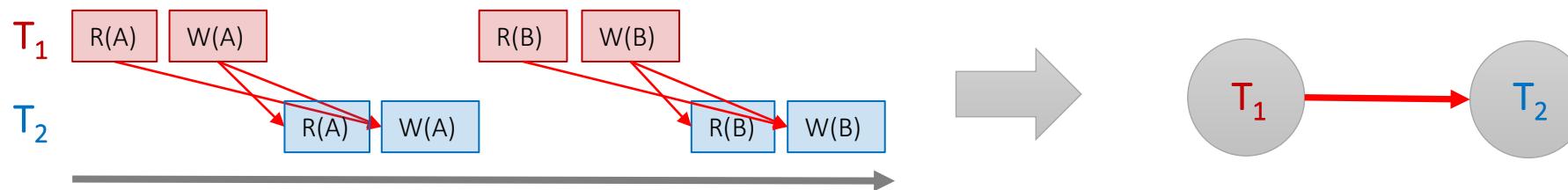
Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!

# 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

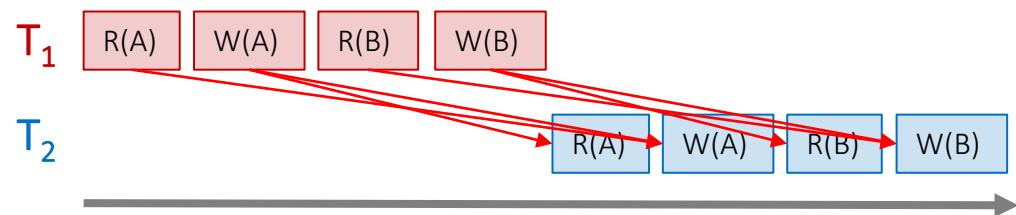
# The Conflict Graph

- 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  $T_j$**



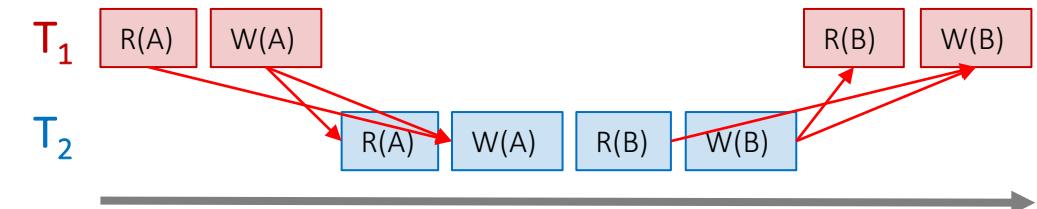
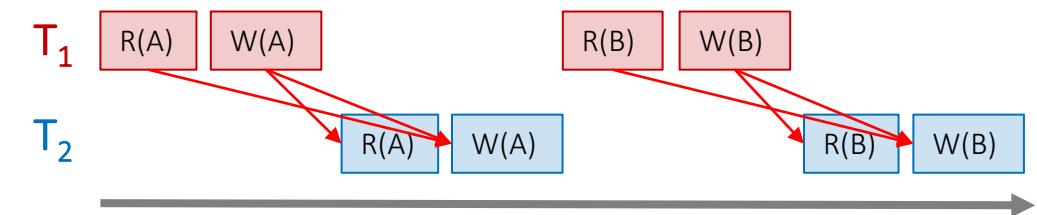
# What can we say about “good” vs. “bad” conflict graphs?

## Serial Schedule:



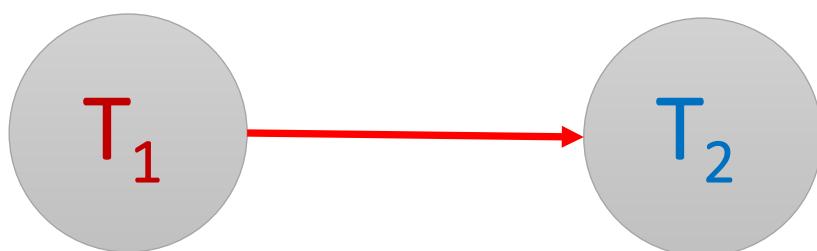
A bit complicated...

## Interleaved Schedules:



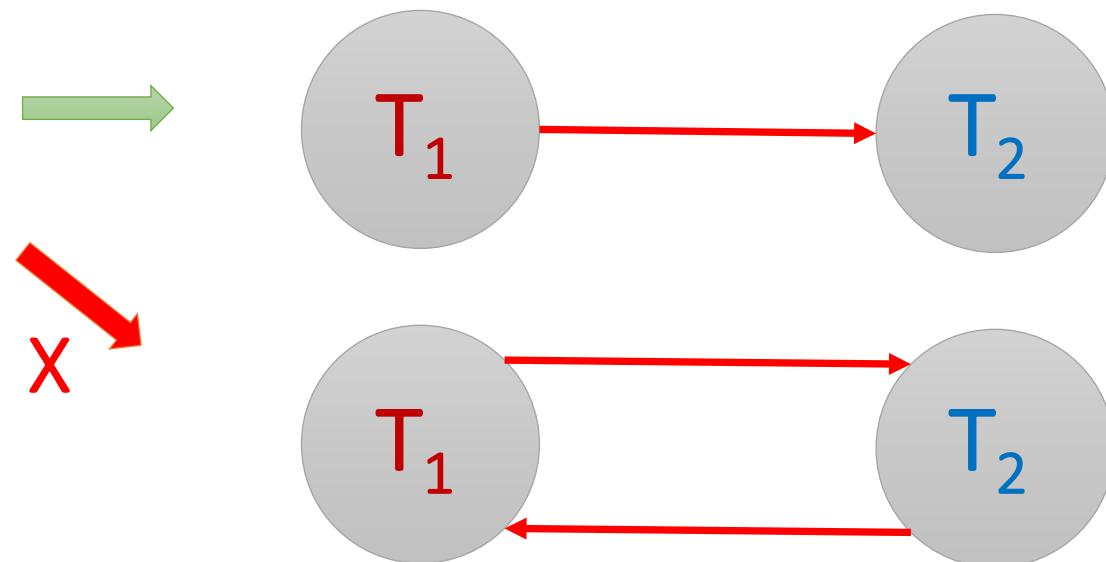
# What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



Simple!

Interleaved Schedules:



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

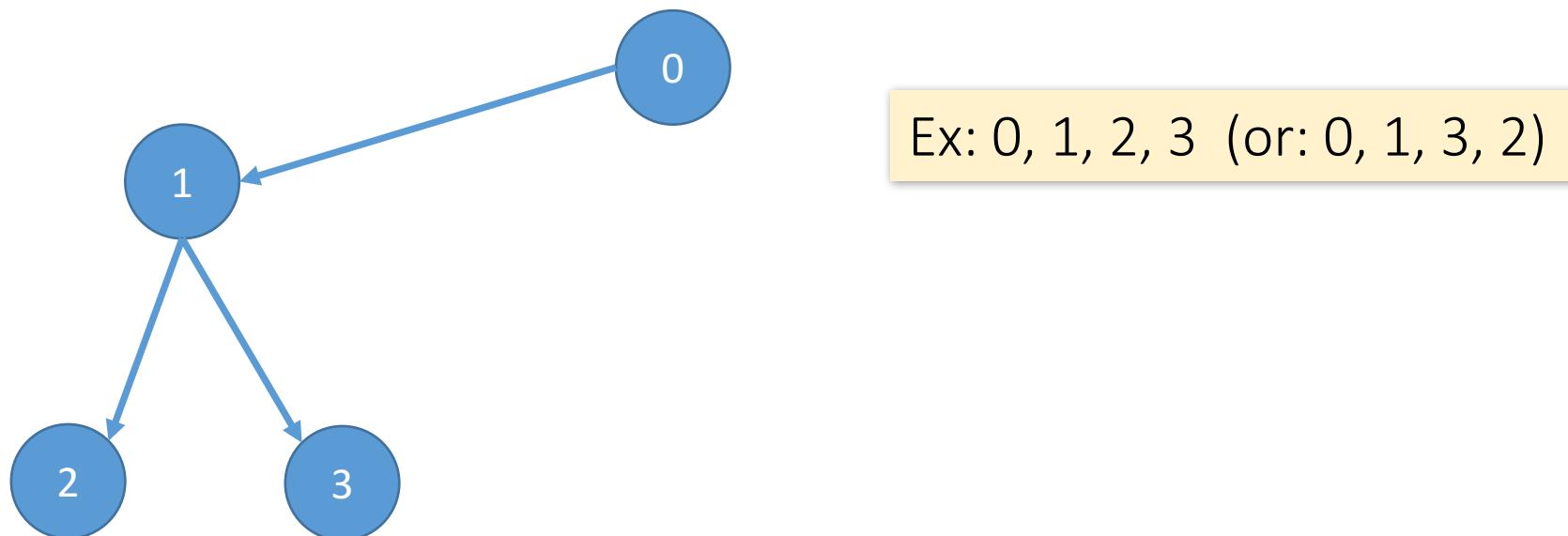
Let's unpack this notion of acyclic  
conflict graphs...

# 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)

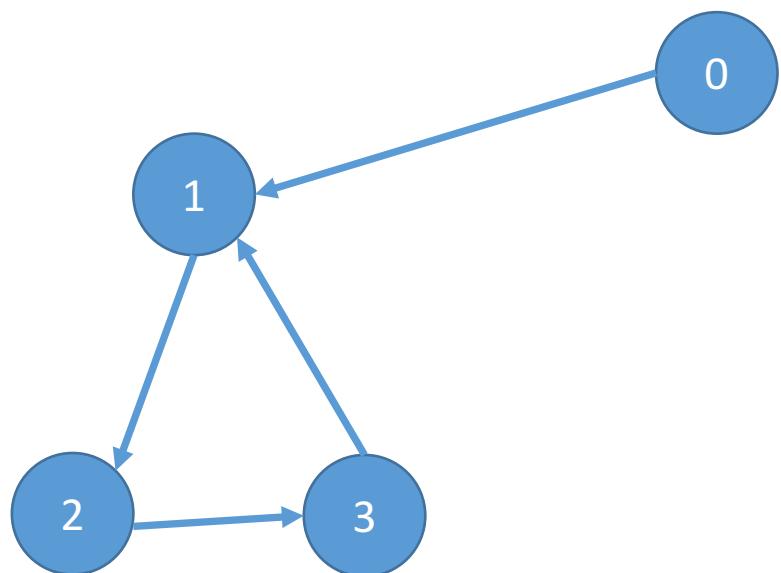
# DAGs & Topological Orderings

- Ex: What is one possible topological ordering here?



# 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!

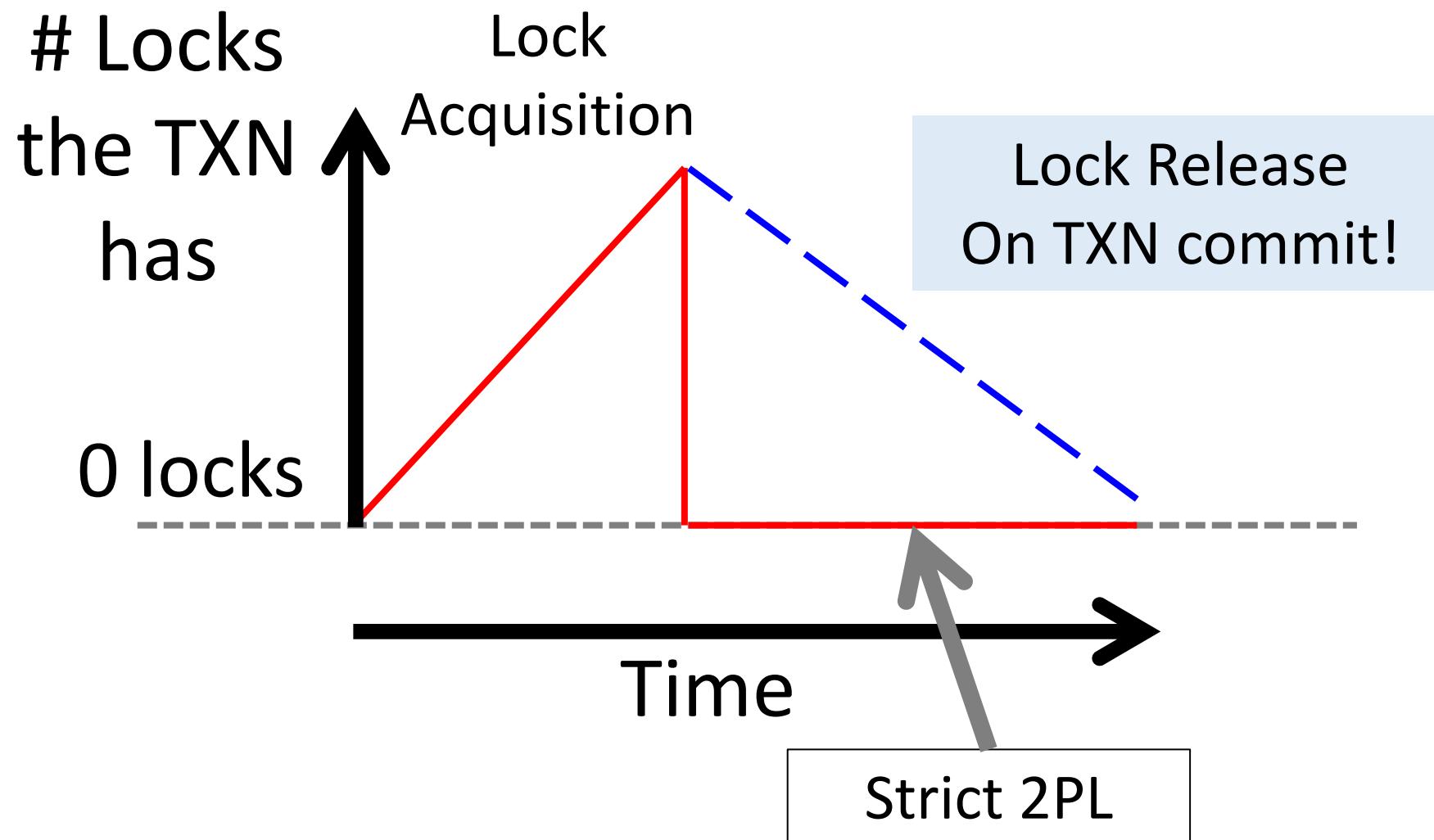
# Strict Two-phase Locking (Strict 2PL) Protocol:

## 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
- All locks held by a TXN are released when TXN completes.

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

# 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  $\Rightarrow$  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 Detection: Example

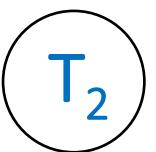
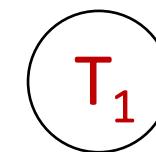


$T_2$



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

Waits-for graph:

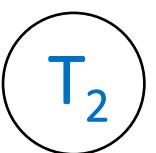
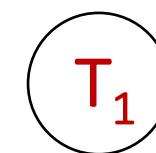


# Deadlock Detection: Example

$T_1$      $S(A)$      $R(A)$

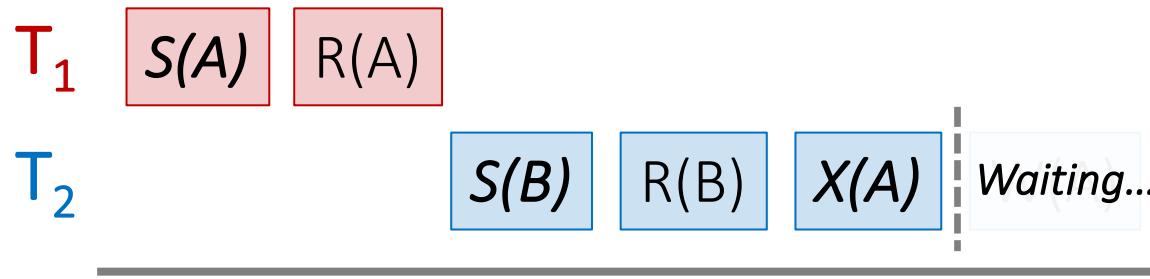
$T_2$                  $S(B)$      $R(B)$

Waits-for graph:



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

# Deadlock Detection: Example

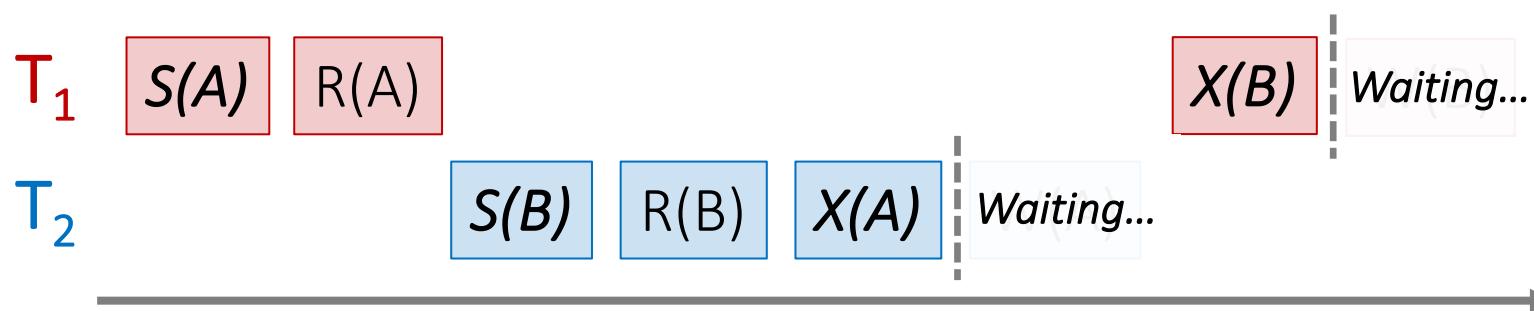


Waits-for graph:

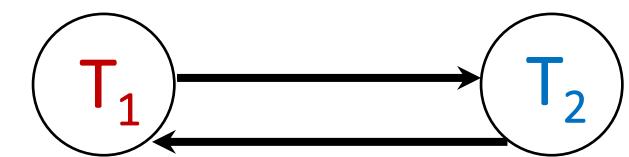


$T_2$  then requests an exclusive lock on A to write to it- now  $T_2$  is waiting on  $T_1$ ...

# Deadlock Detection: Example



Waits-for graph:

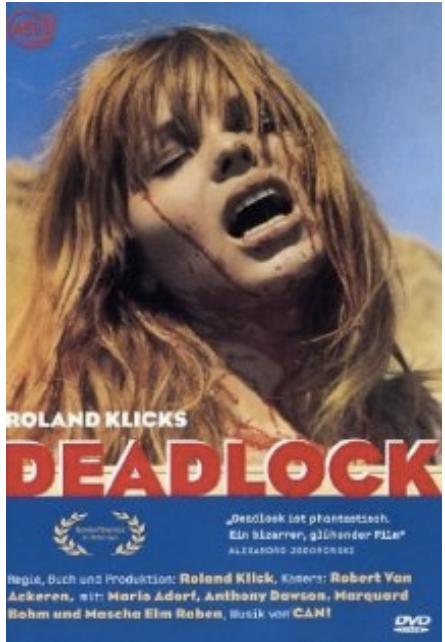


Cycle =  
DEADLOCK

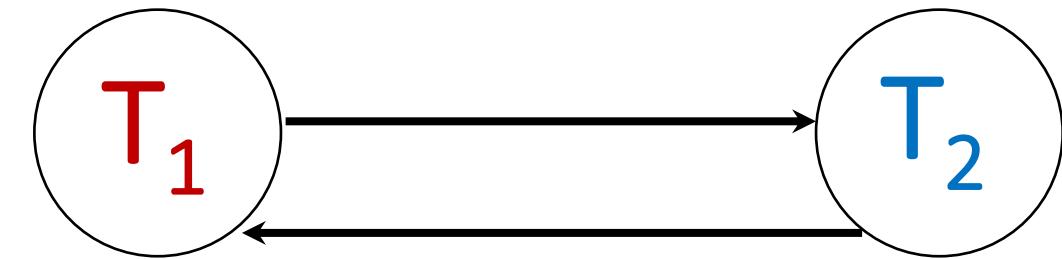
Finally,  $T_1$  requests an exclusive lock on B to write to it- now  $T_1$  is waiting on  $T_2$ ... DEADLOCK!

## 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!??!



NB: Also movie called wedlock  
(deadlock) set in a futuristic prison...  
I haven't seen either of them...

# 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 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

# 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!)