L16: Transactions & Concurrency Control
Part 1: Transactions & Logging
Part 2: Concurrency Control

CS3200 Database design (fa18 s2)

https://northeastern-datalab.github.io/cs3200/

Version 11/1/2018

Announcements!

- Exam 2 is on Monday
  - content is everything seen until today
  - setup like for Exam1: laptop SQL + paper database design + paper transactions
- HW10 = Past semester's final exam (ignore section 4 on internals / sort)
- Notice our changed OHs this week (no THU, but TUE and FRI)
- Today
  - Transactions
- CURN CCIS Undergraduate Research Night: Monday 6:30-8:30
- Jupyter exercises: let's try it once more together

CURN (CCIS Undergraduate Research Night): Mon, Nov 5

UNDERGRADUATE RESEARCH NIGHT

NOVEMBER 5 6:30 PM - 8:30 PM EST

Title: Undergraduate Research Night

Location: Northeastern University, 102 West Village H, 440 Huntington Avenue, Boston, Massachusetts 02115

Abstract

Learn about how you can participate in cutting-edge computer science research! Featuring a panel with fellow undergraduate students who have participated in research to discuss their experience, and academic advisors to discuss how you can integrate research into your curriculum (as a co-op and course credit). Afterward, attend the poster session to interact with researchers discussing ongoing research projects in the college and how you can get involved. This will be the fourth annual UG Research Night that CCIS has hosted.

Free food provided!

Link: https://www.ccis.northeastern.edu/event/undergraduate-research-night-2018/

ACID: Isolation

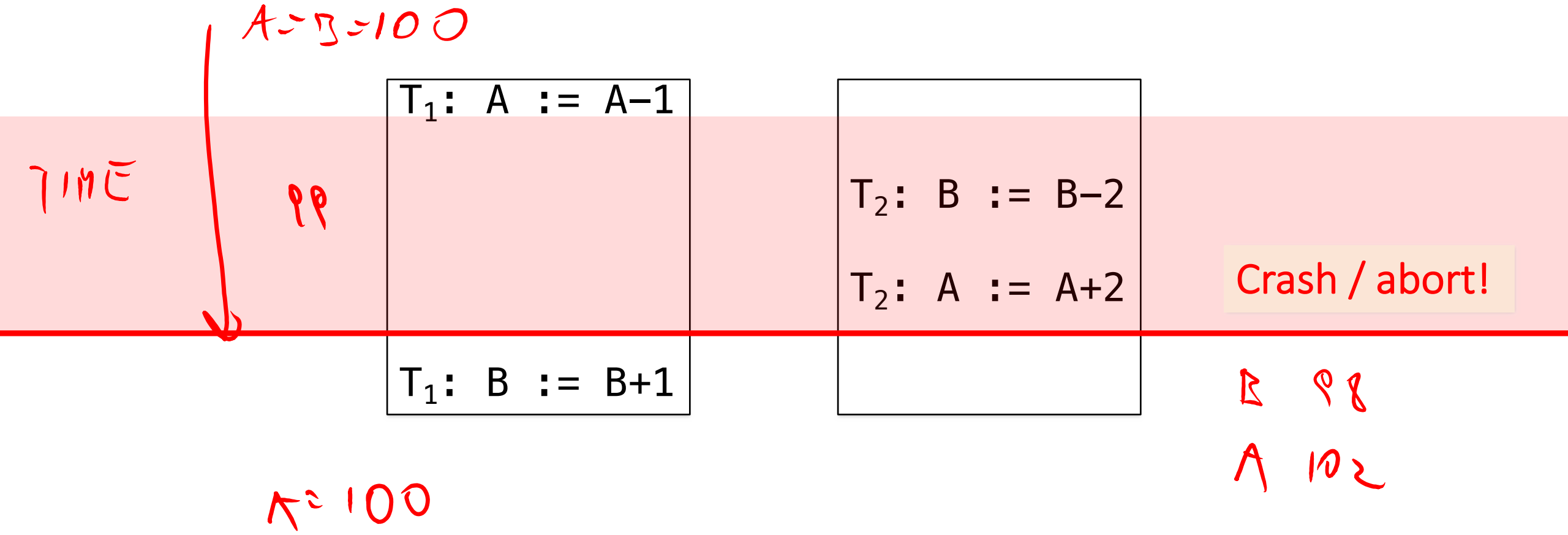
- A transaction executes concurrently with other transactions

- 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

Isolation failure

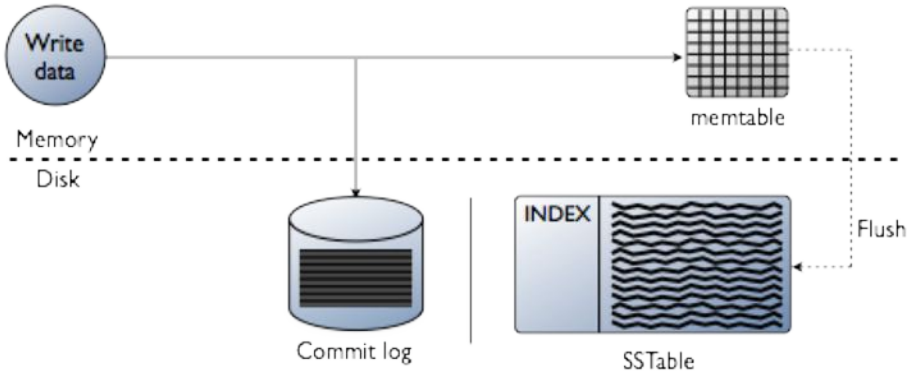
Write-Write Conflict



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

And all this with… Performance!!

A Note: ACID is contentious!

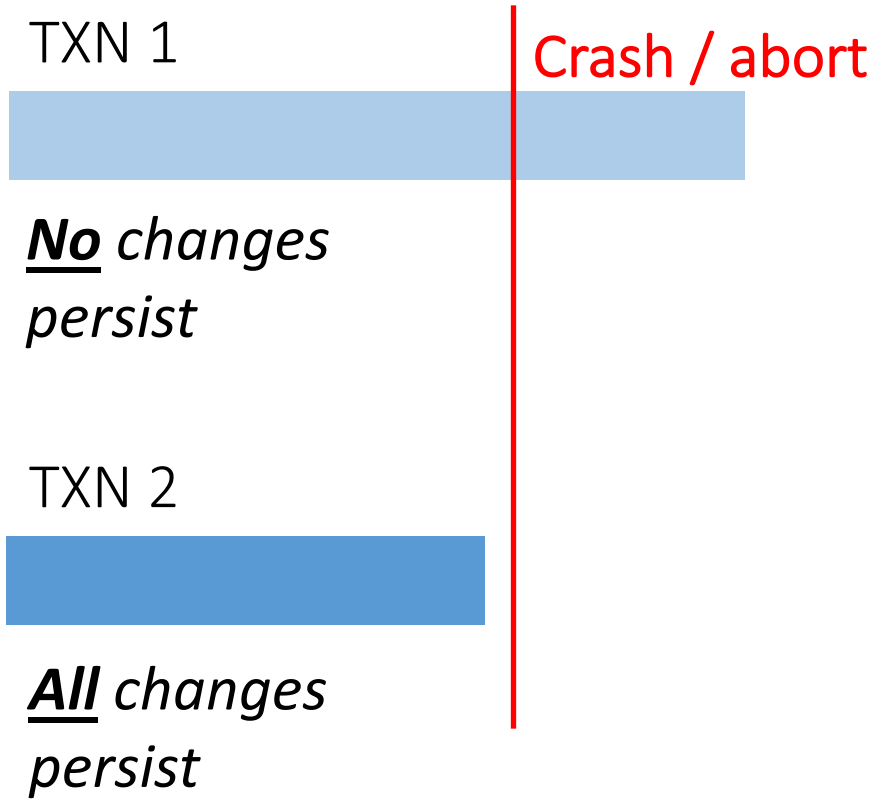
- Many debates over ACID, both historically and currently
- Many newer “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs…

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

Our first Goal: Ensuring Atomicity & Durability

ACID

- Atomicity:
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, no effects should be seen
- Durability:
  - If DBMS stops running, changes due to completed TXNs should all persist
  - Just store on stable disk



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 by the DBMS.
  - ("transparently" = without the user being aware)

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>

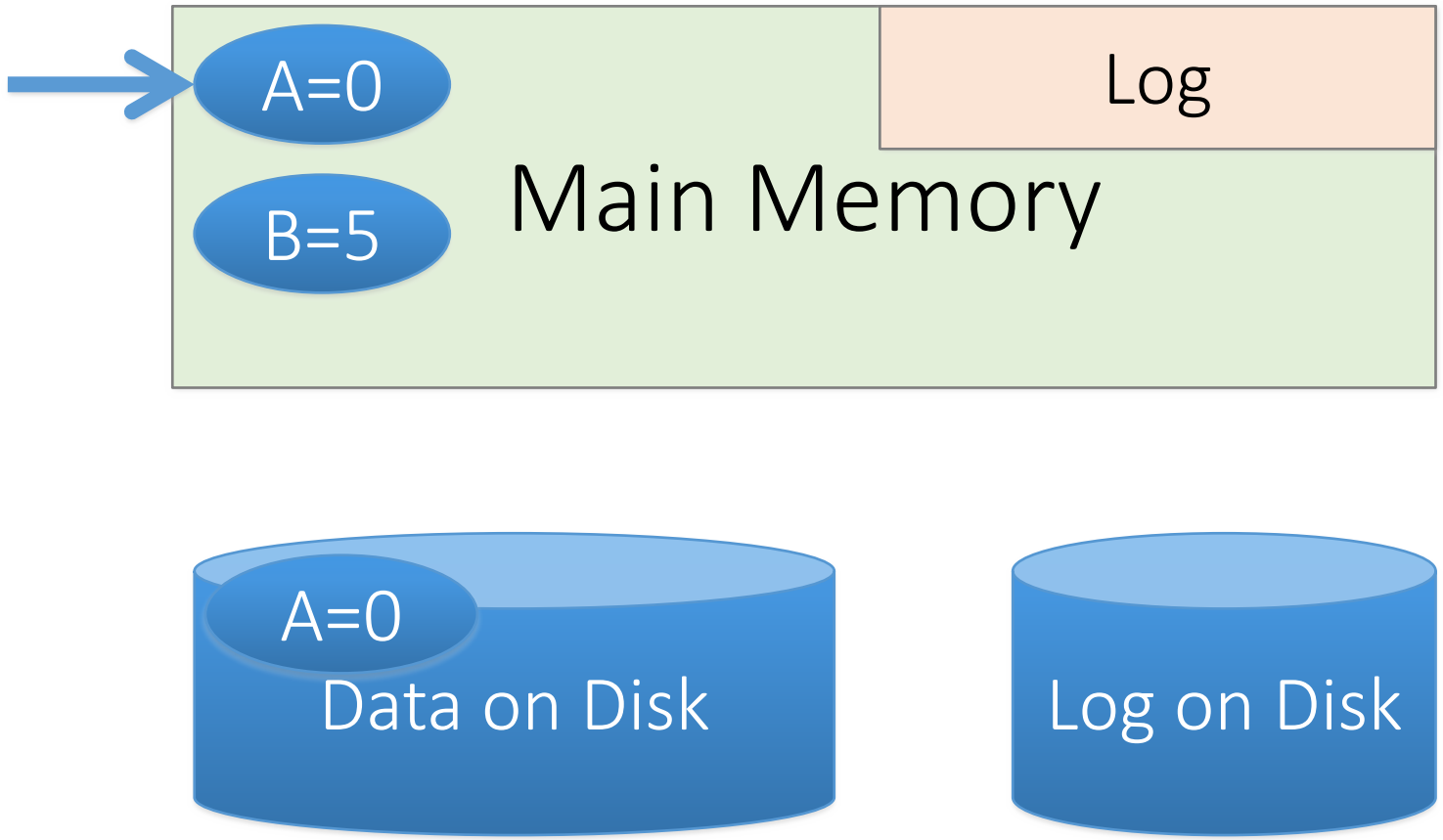
This is sufficient to UNDO any transaction!

3. Atomicity & Durability via Logging

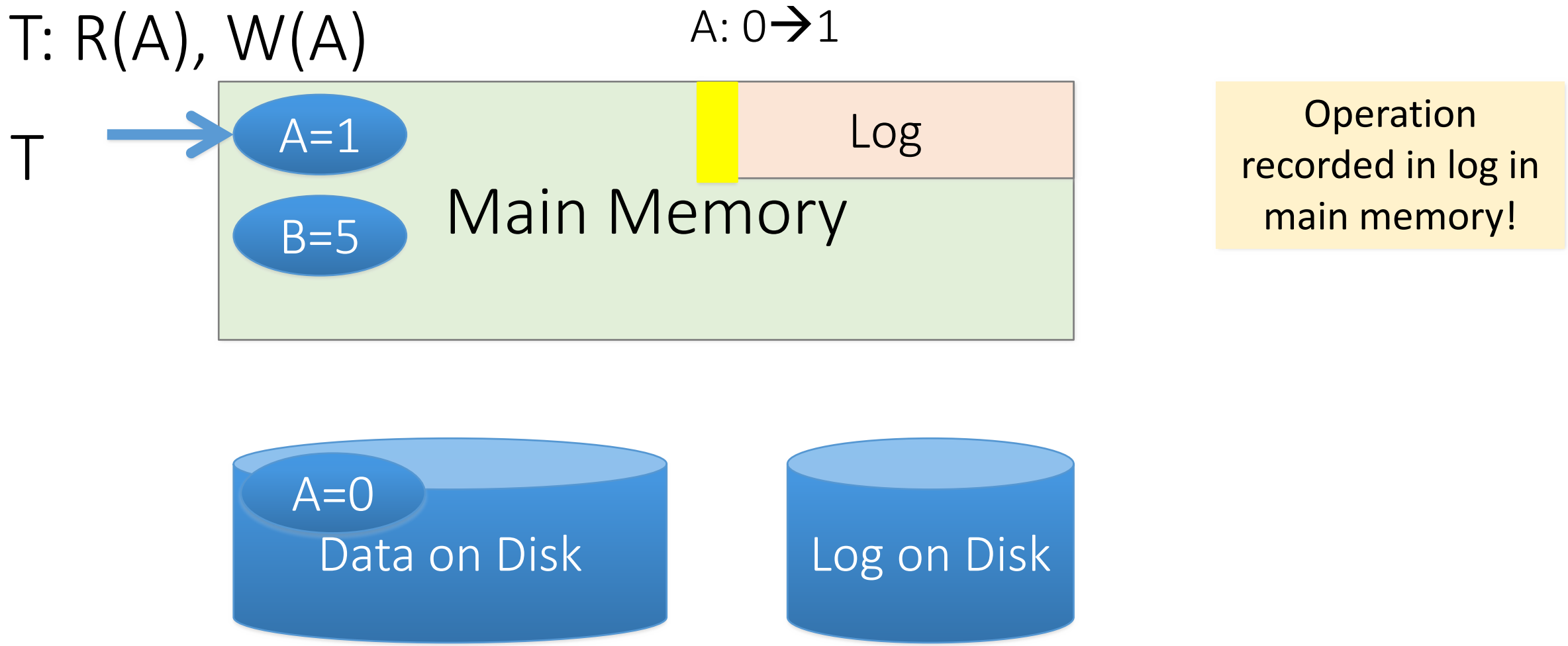
An animation of commit protocols

A picture of logging

T: R(A), W(A)



A picture of logging



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

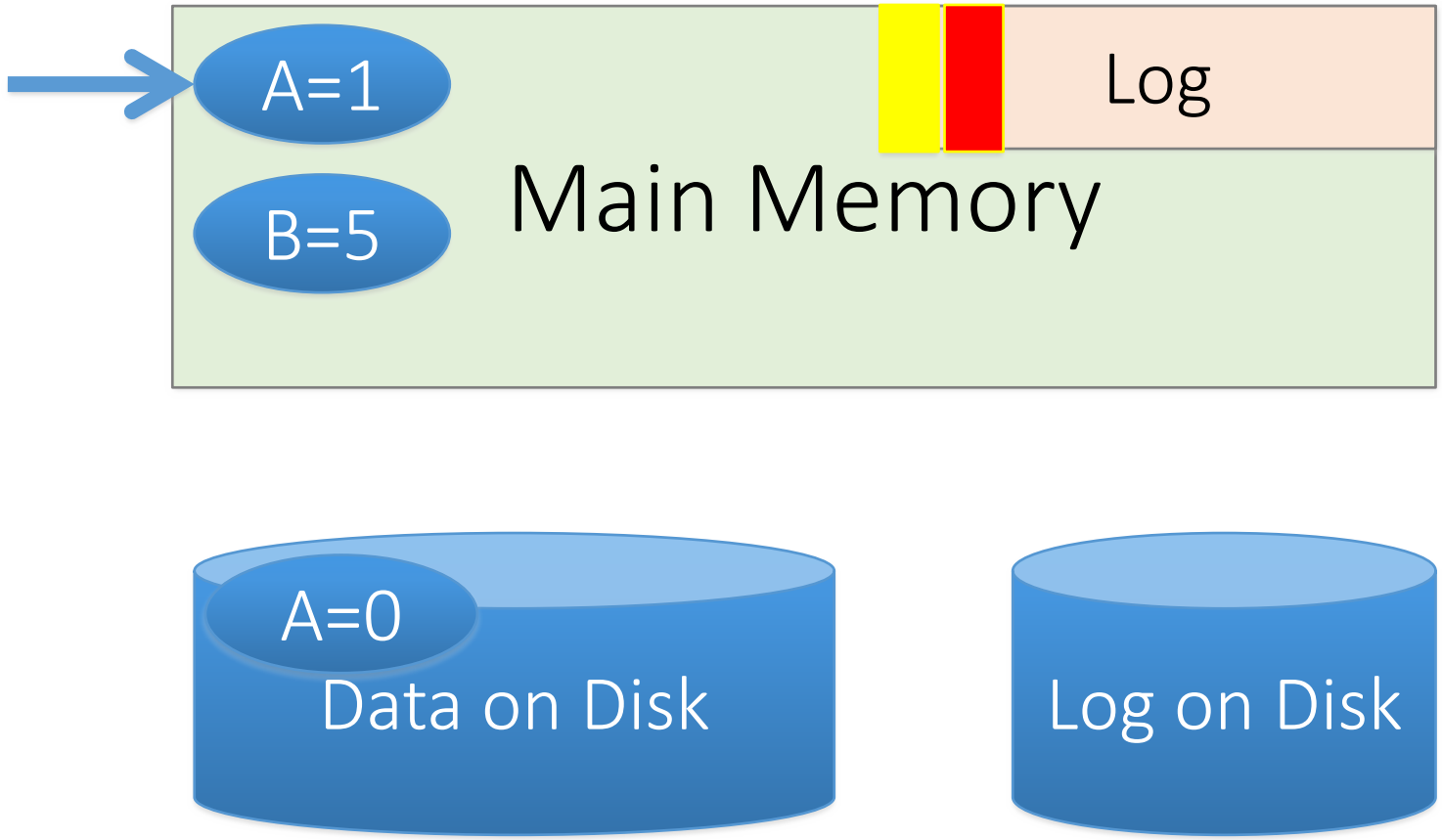
- FORCE Write commit record to log
- All log records up to last update from this TX are FORCED
- Commit() returns

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

Incorrect Commit Protocol #1

T: R(A), W(A)

A: 0→1



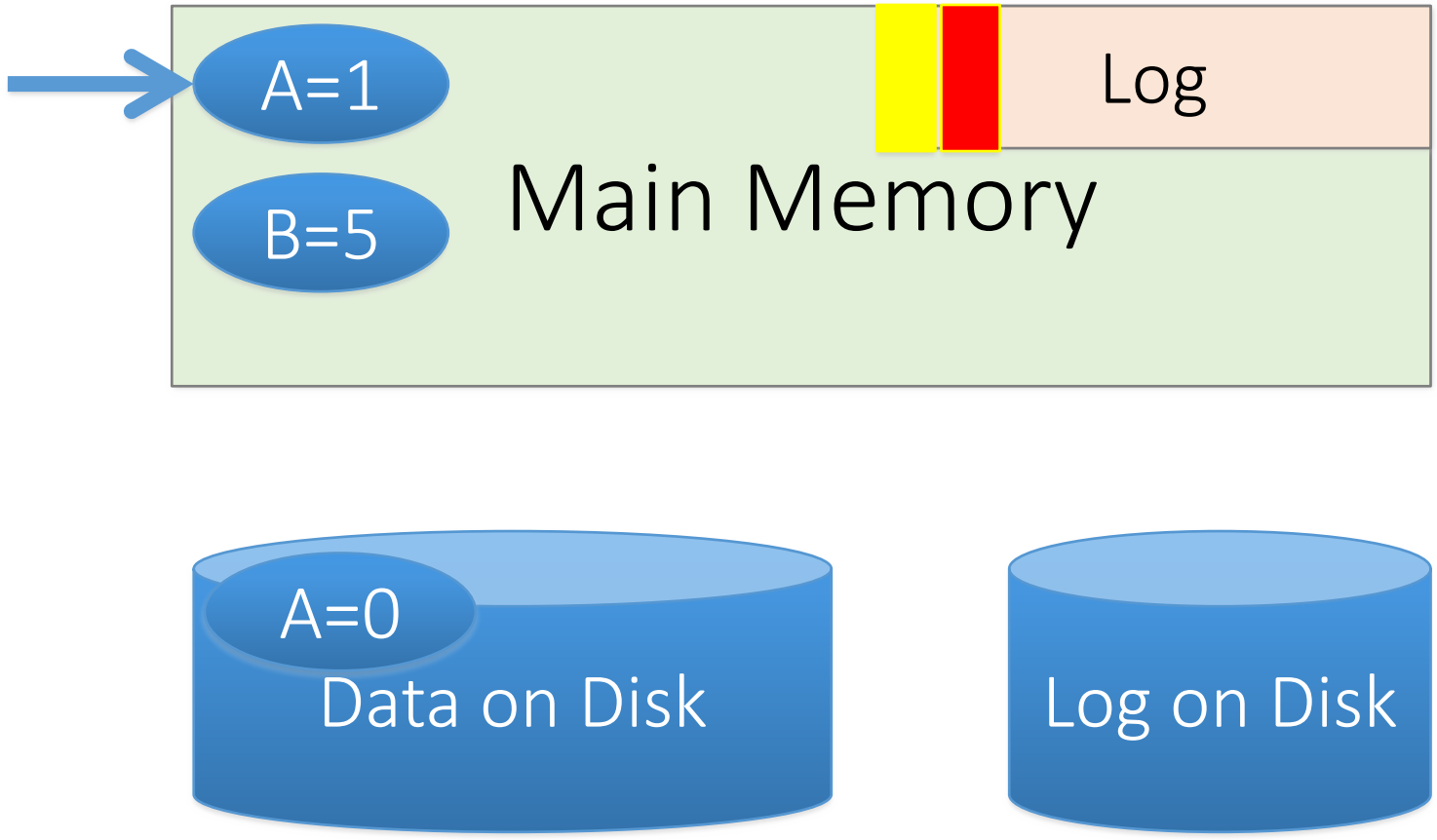
Let's try committing *before* we've written either data or log to disk…

OK, Commit!

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!

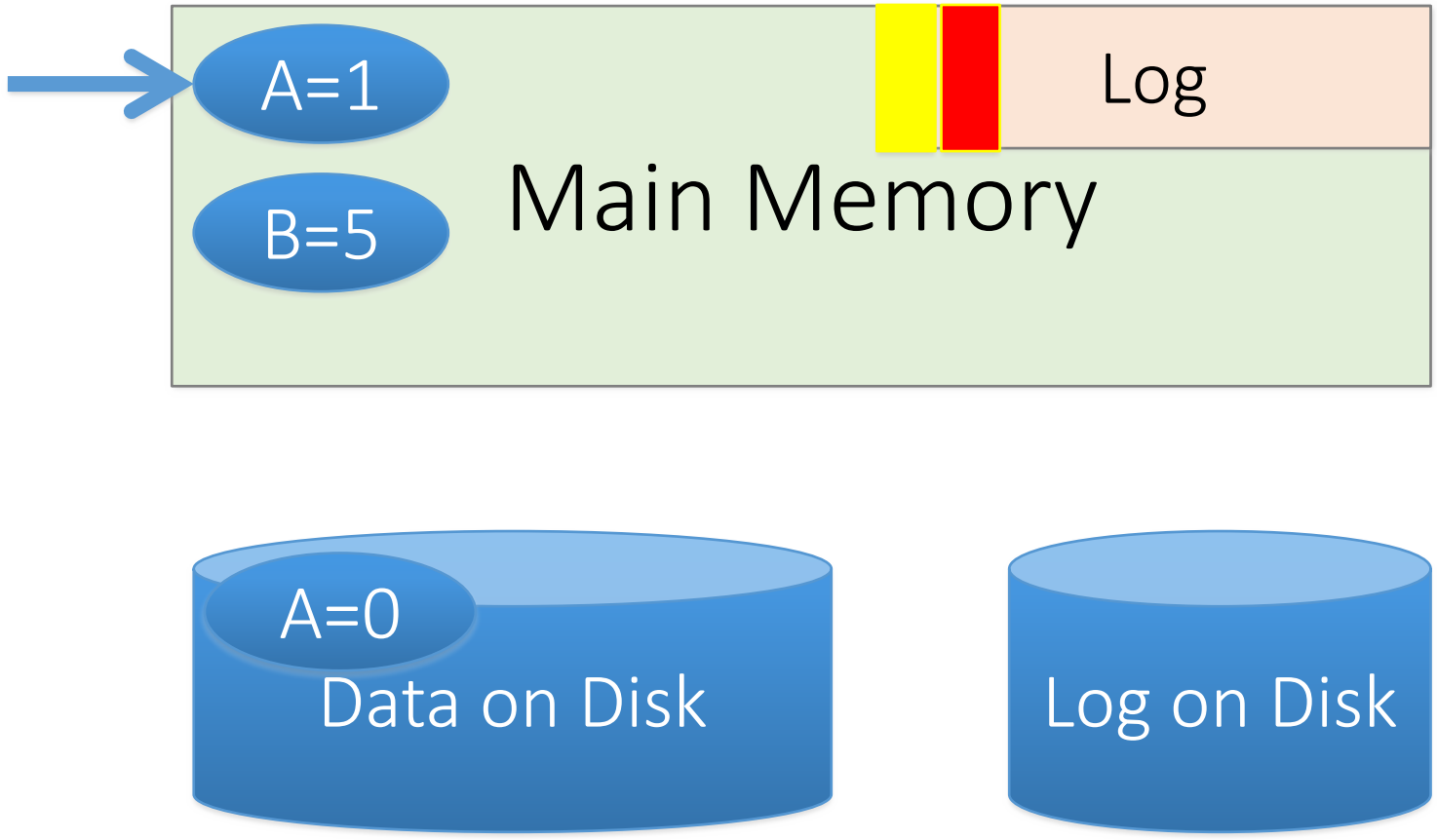
If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2

T: R(A), W(A)

A: 0→1



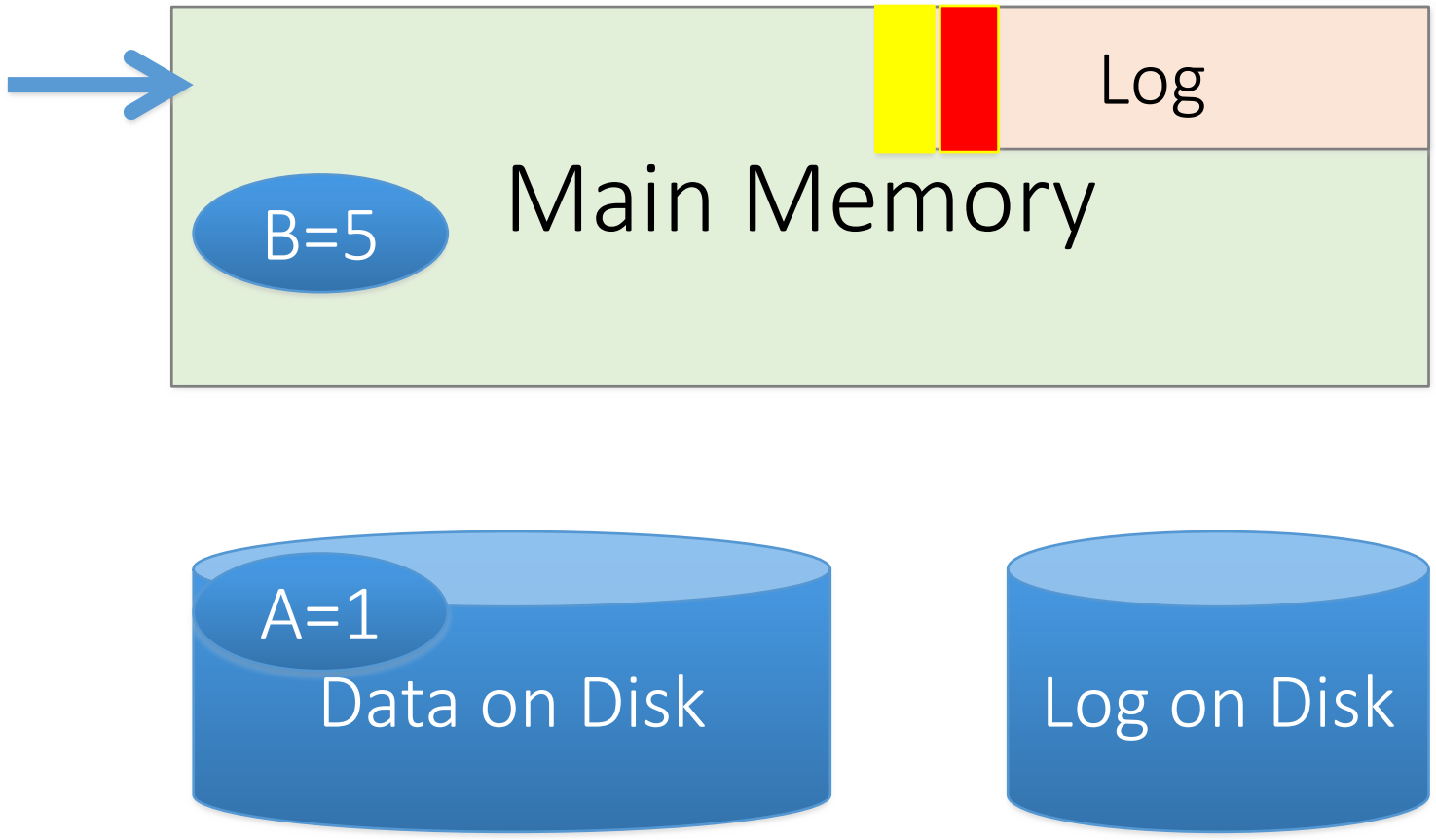
Let's try committing *after* we've written data but *before* we've written log to disk…

OK, Commit!

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…

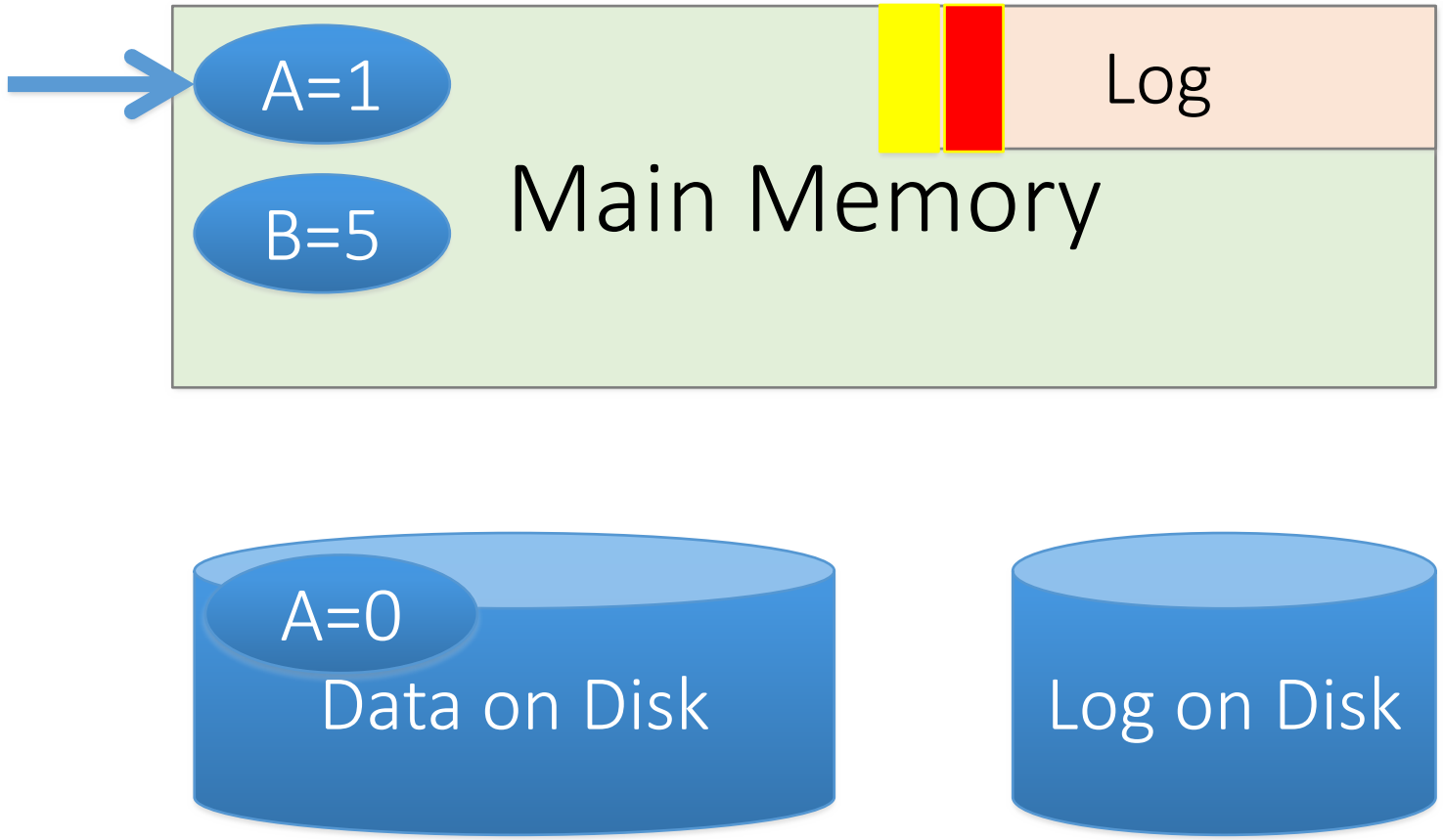
How do we know whether T was committed??

Improved Commit Protocol (WAL)

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

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!

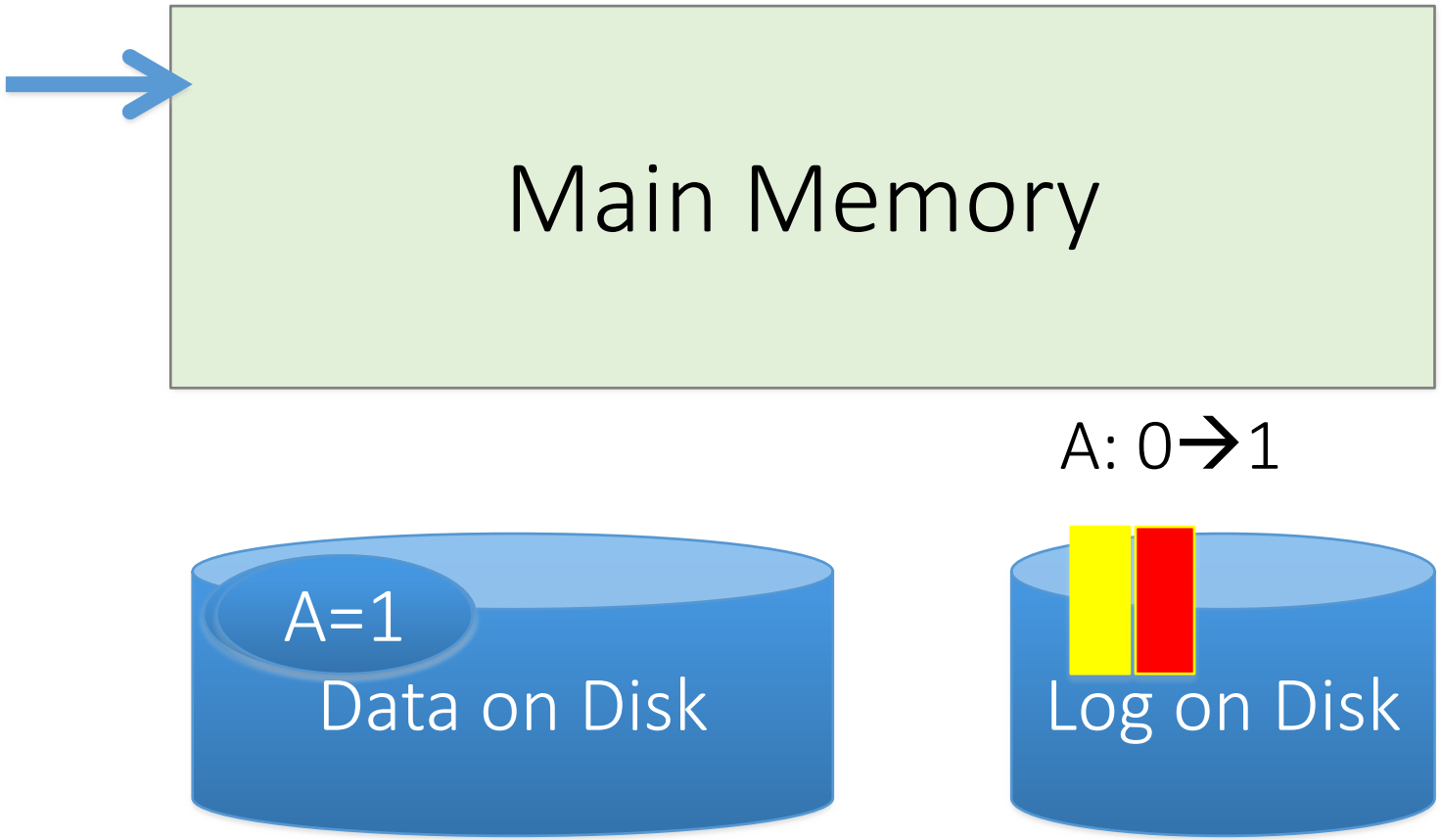
OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

T



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…

Concurrency & Locking

1) Concurrency, scheduling & anomalies
2) Locking: 2PL, conflict serializability, deadlock detection

1. Concurrency, Scheduling & Anomalies

What we will learn next

- Interleaving & scheduling
- Conflict & anomaly types
- ACTIVITY: TXN viewer

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency s.t. …
  - **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

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

ACID

The hard part is the effect of *interleaving* transactions and *crashes*.

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
```

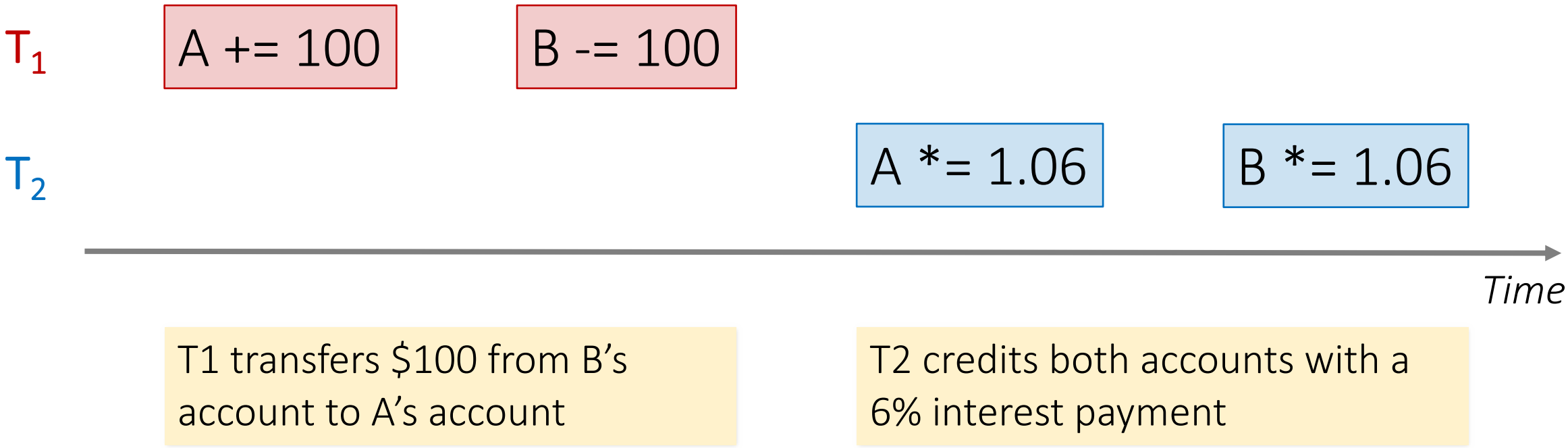
T1 transfers $100 from B's account to A's account

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

T2 credits both accounts with a 6% interest payment

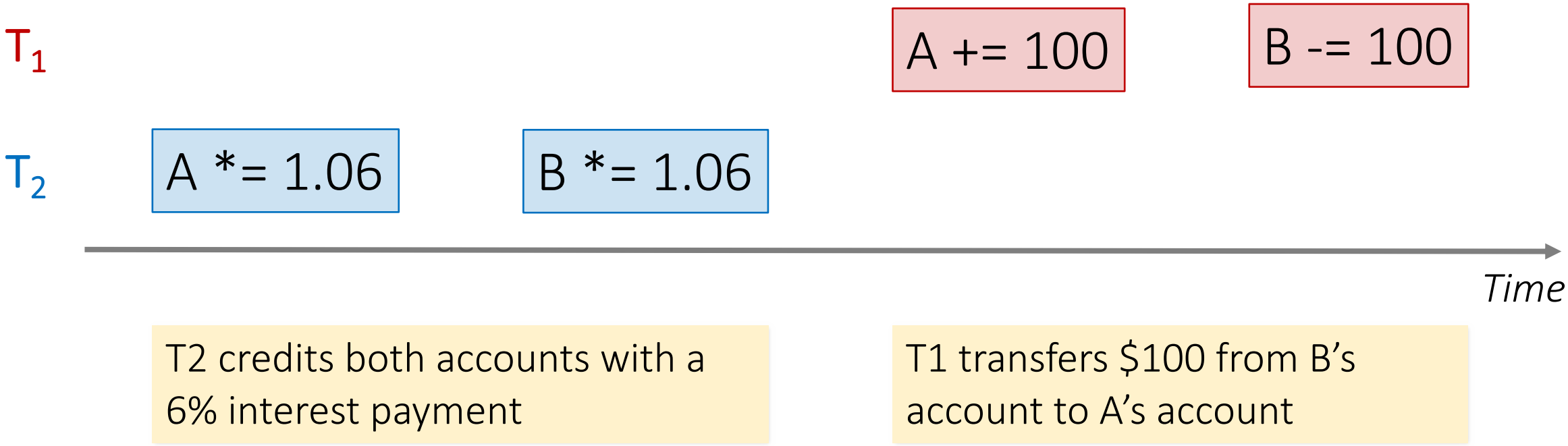
Example- consider two TXNs:

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



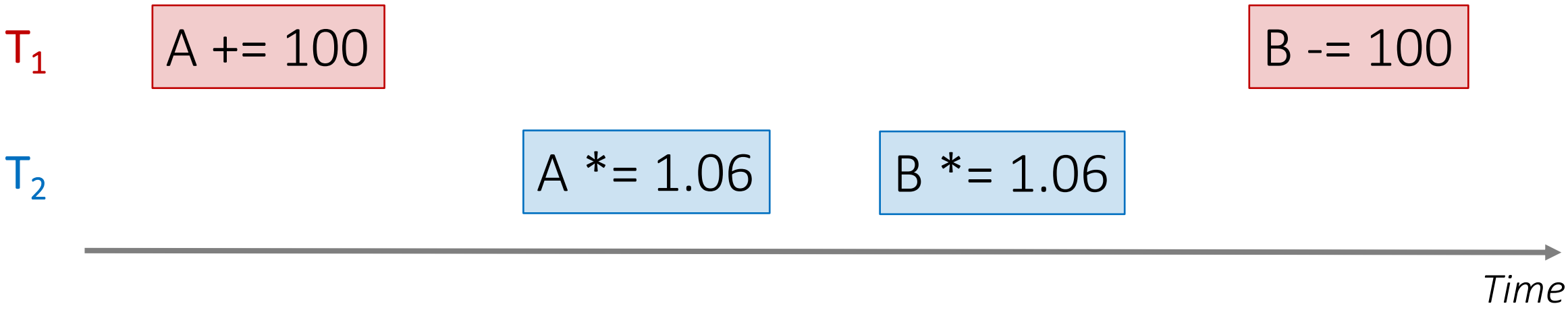
Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!



Example- consider two TXNs:

The DBMS can also **interleave** the TXNs

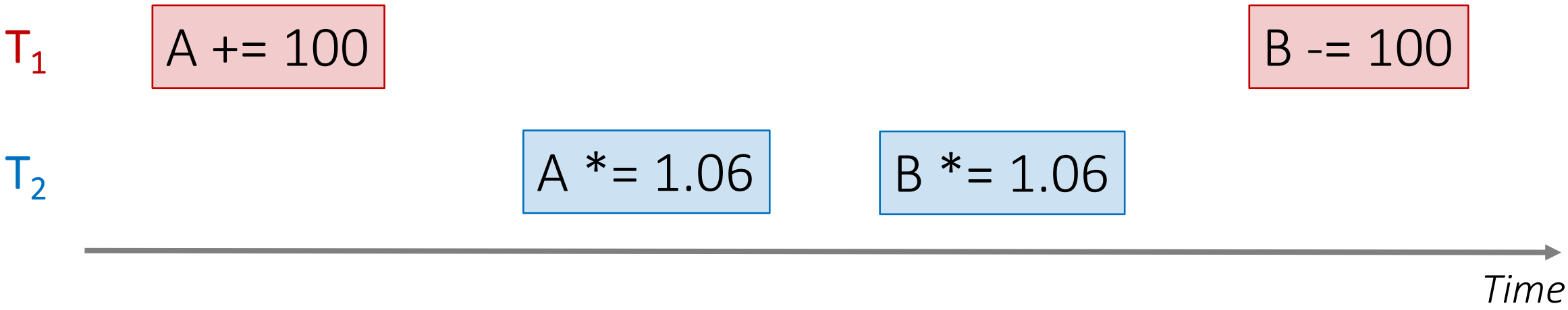


T1 transfers $100 to A's accout, then T2 credits A's account with 6% interest payment

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??

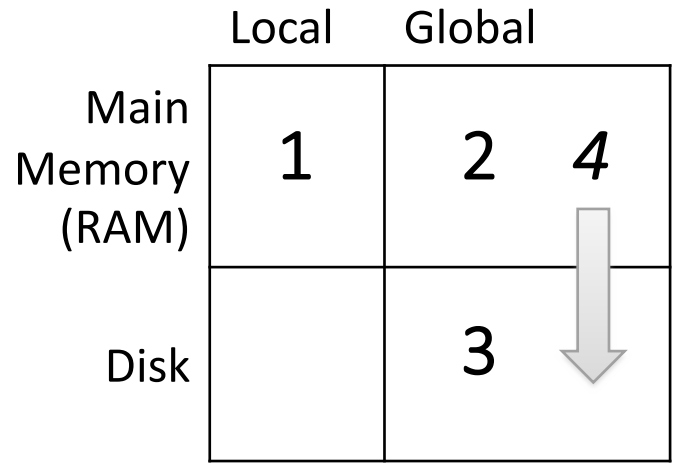
Recall: Three Types of Regions of Memory

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 from main memory

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

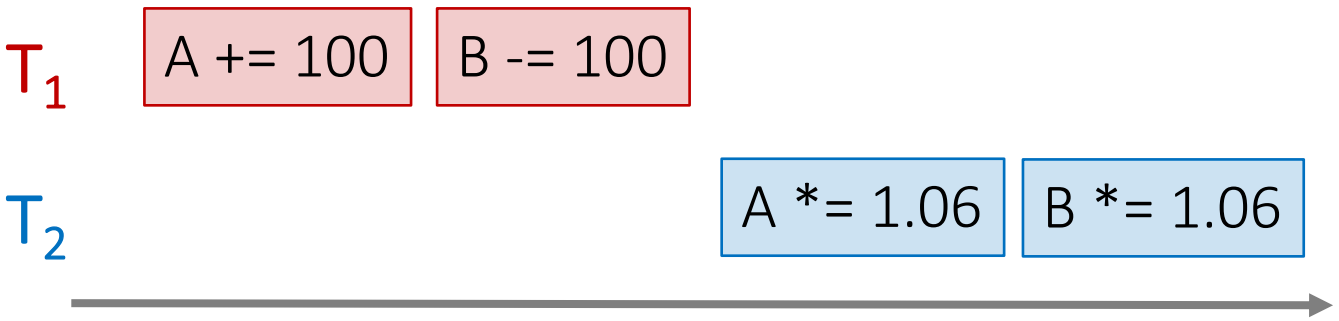
DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

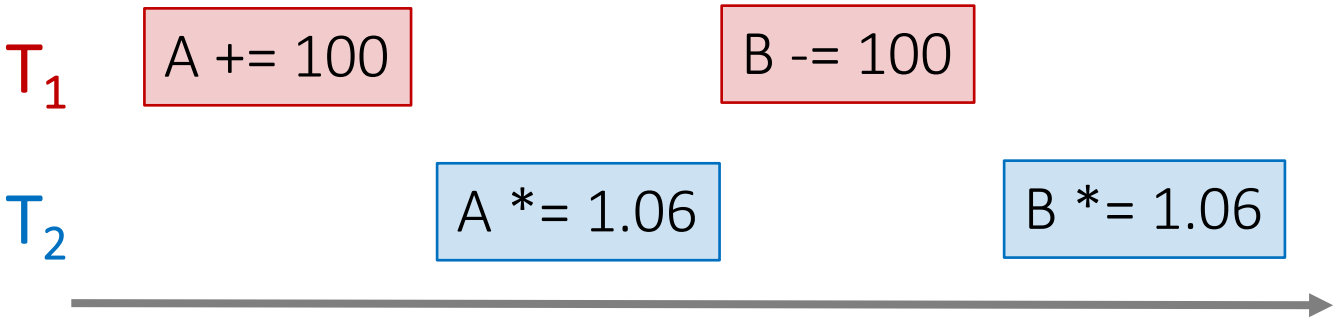
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule 1:



A	B
$159	$106

Same result!

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 2:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

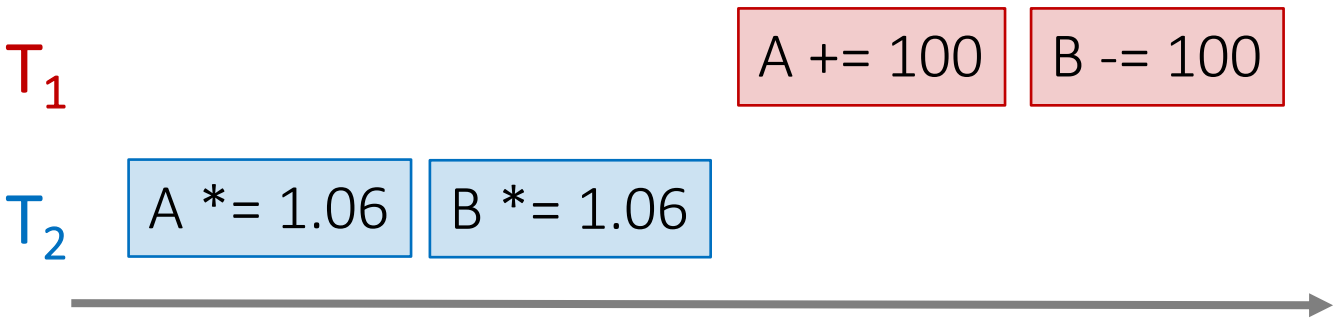
A	B
$159	$112

Different result than serial T_1,T_2

Scheduling examples

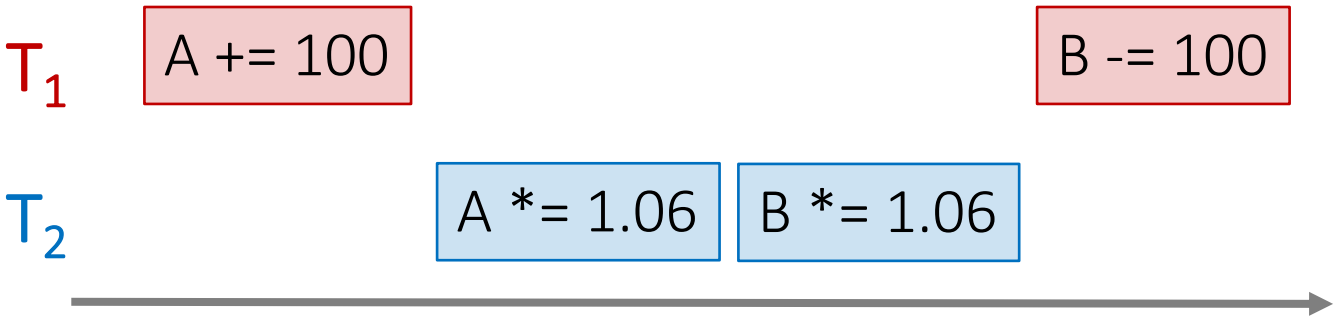
Starting Balance	A	B
	$50	$200

Serial schedule T_2,T_1:



A	B
$153	$112

Interleaved schedule 2:



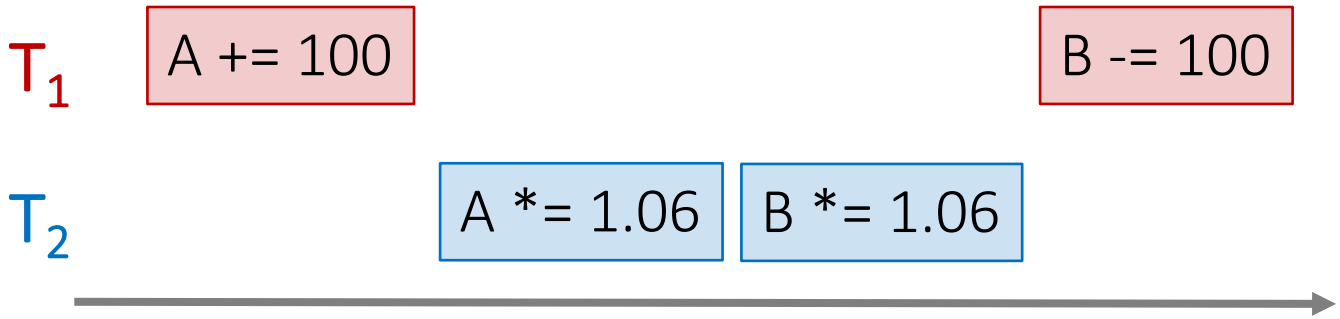
A	B
$159	$112

Different result than serial T_2,T_1 as well!

Scheduling examples

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

Interleaved schedule 2:



Scheduling Definitions

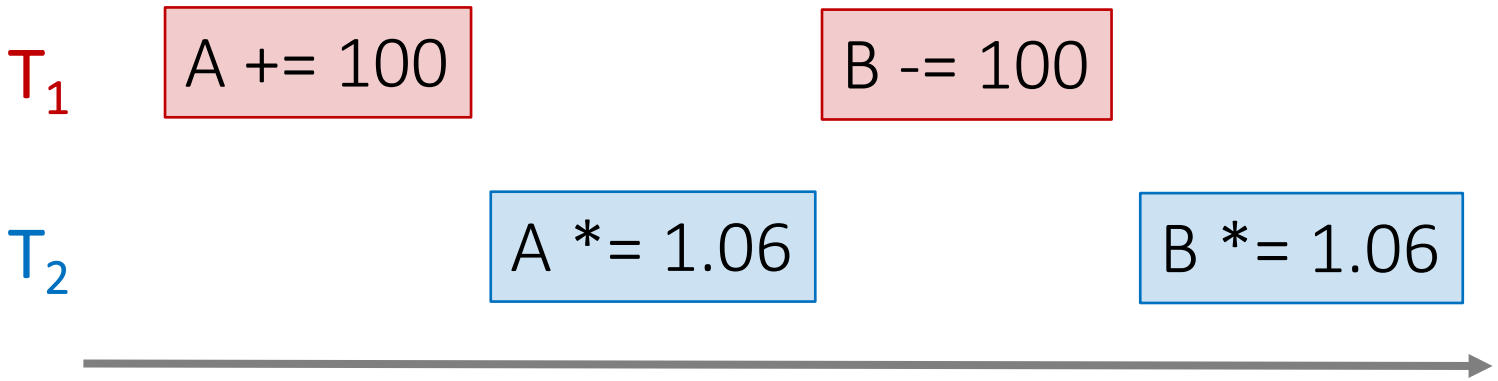
- A **serial schedule** is one that does not interleave the actions of different transactions

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

- 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?



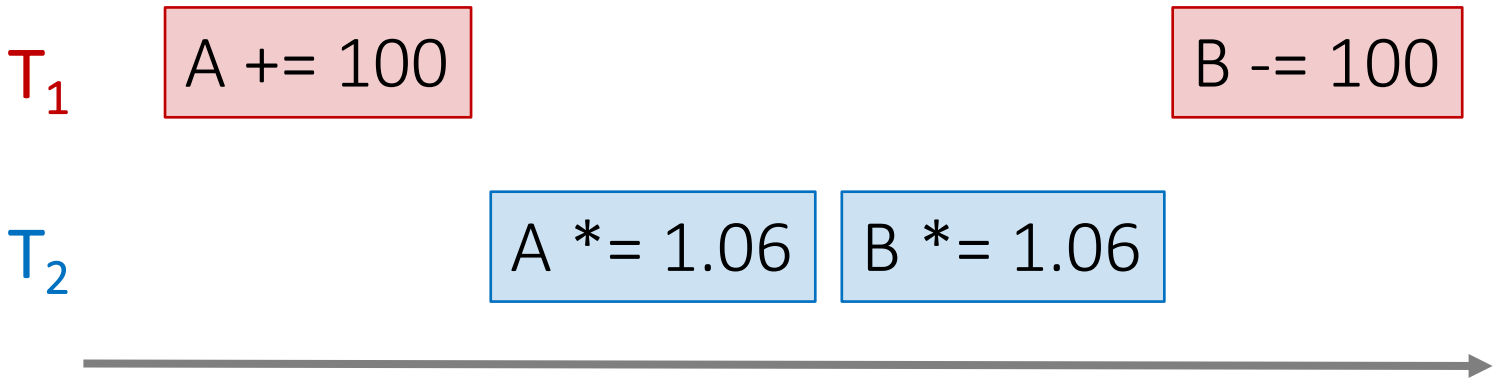
Serial schedules:

	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* = 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?

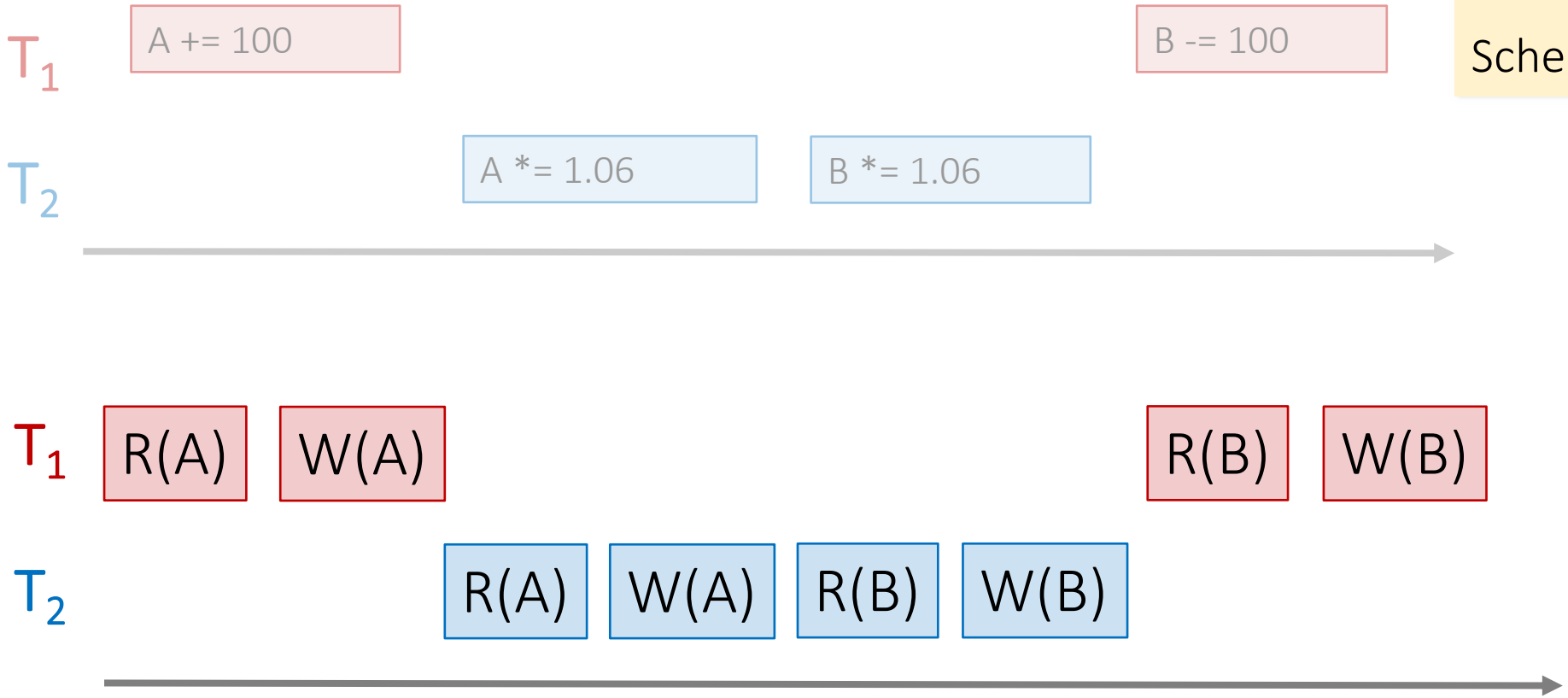
- Various anomalies which break isolation / serializability
  - Often referred to by name (recall last class: "lost updated", "unrepeateable reads", etc.)
  - We see them again in a bit

- 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)

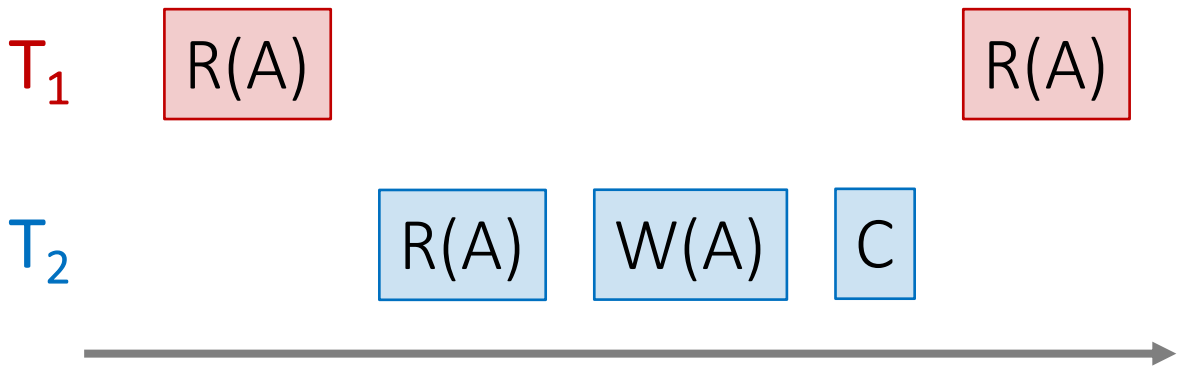
Why no "RR Conflict"?

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

Classic Anomalies with Interleaved Execution

“Unrepeatable read”

Example:

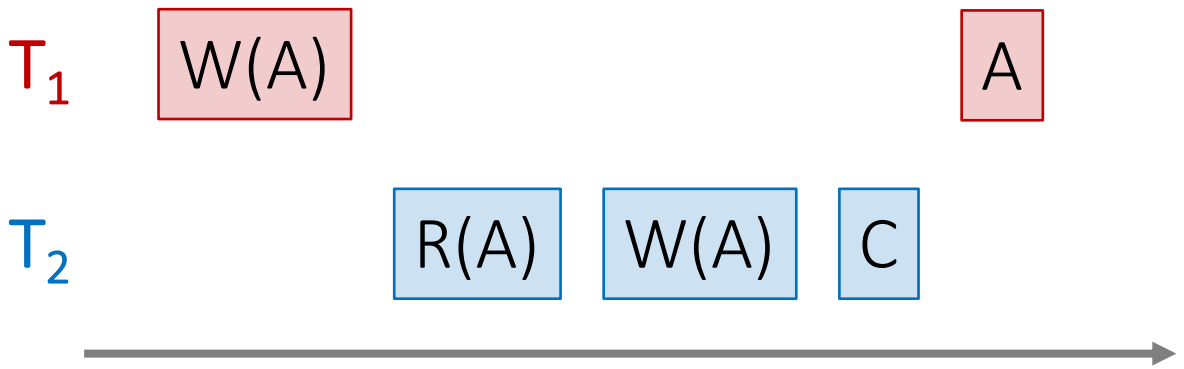


1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

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

“Dirty read” (Reading uncommitted data)

Example:

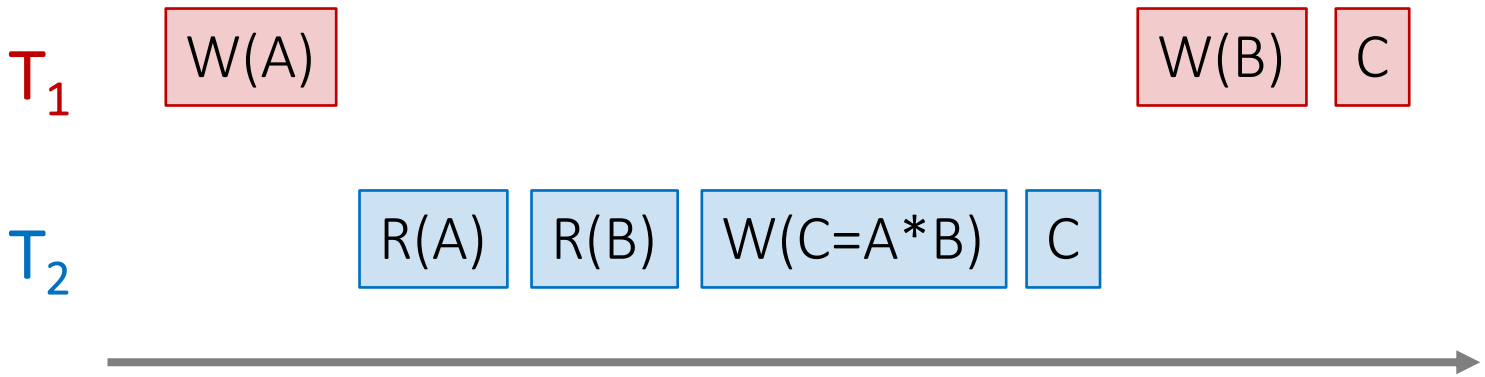


1. T_1 <u>writes</u> some data to A
2. T_2 <u>reads</u> from A, then writes back to A & commits
3. T_1 then aborts- *now T_2’s result is based on an obsolete / inconsistent value*

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

“Inconsistent read” (Reading partial commits)

Example:

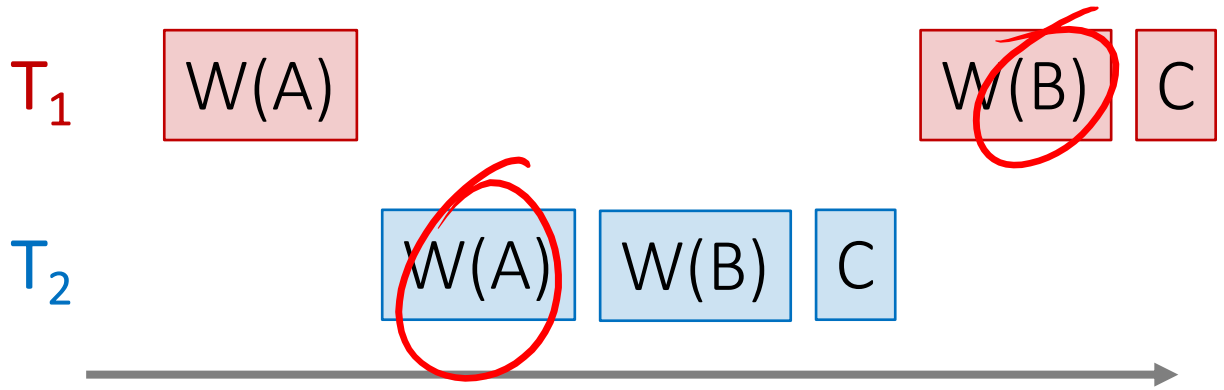


1. T_1 writes some data to A
2. T_2 reads from A *and B*, and then writes some value which depends on A & B
3. T_1 then writes to B- *now T_2’s result is based on an incomplete commit*

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

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-31.ipynb

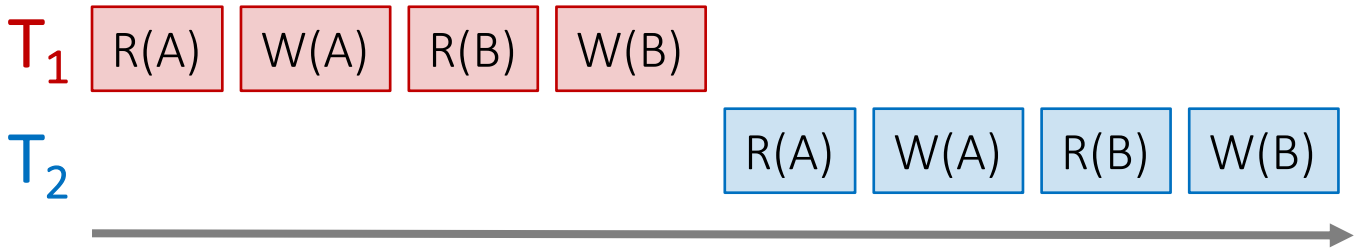
2. Conflict Serializability, Locking & Deadlock

What we will learn next

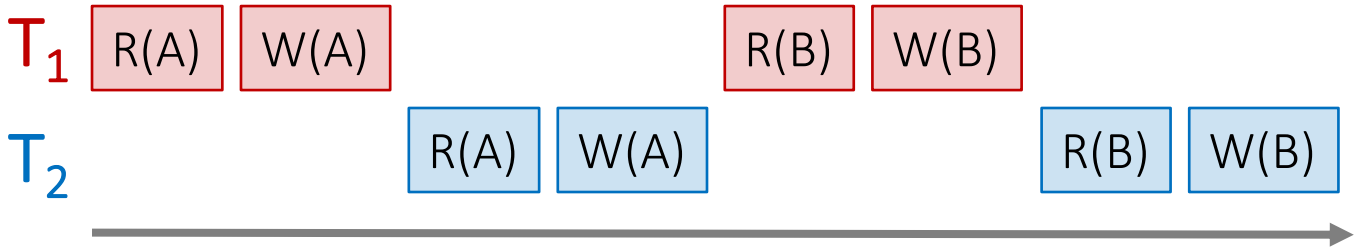
- RECAP: Concurrency
- Conflict Serializability
- DAGs & Topological Orderings
- Strict 2PL
- Deadlocks

Recall: Concurrency as Interleaving TXNs

Serial Schedule:



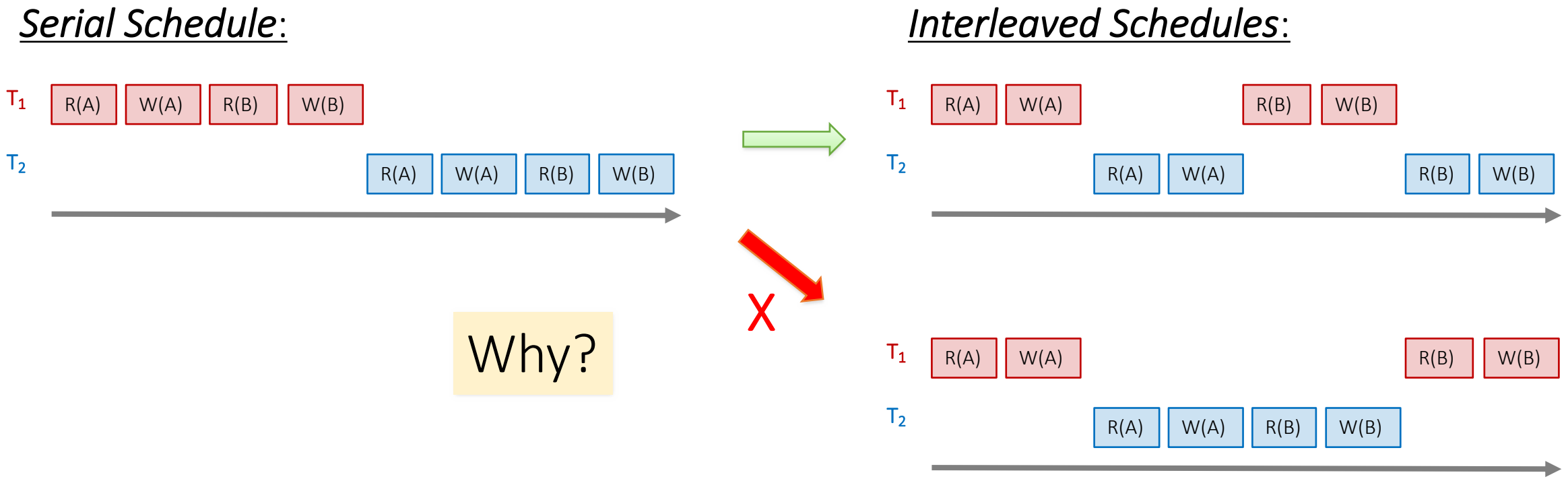
Interleaved Schedule:



- 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



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

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

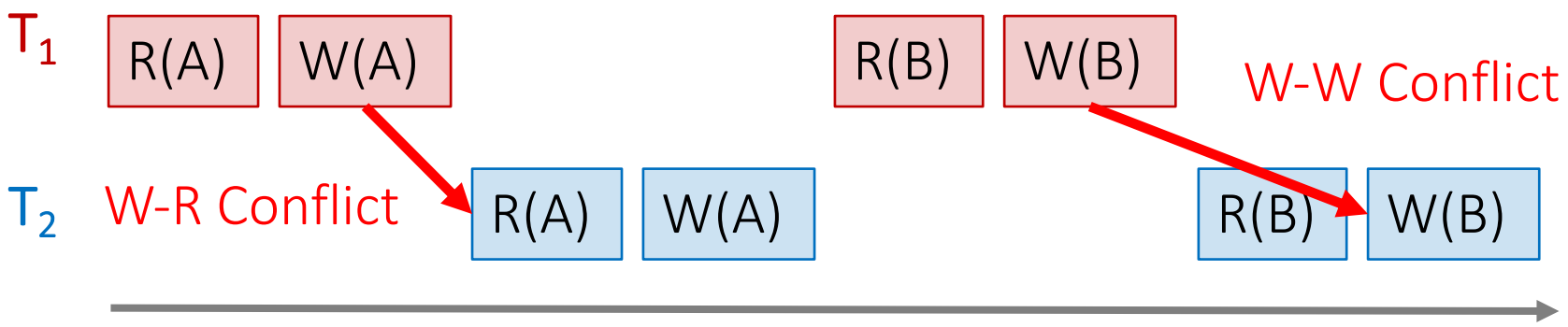
- Recall: we call a schedule serializable if it is equivalent to some serial schedule
  - We used this as a notion of a "good" interleaved schedule, since a serializable schedule will maintain isolation & consistency

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

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

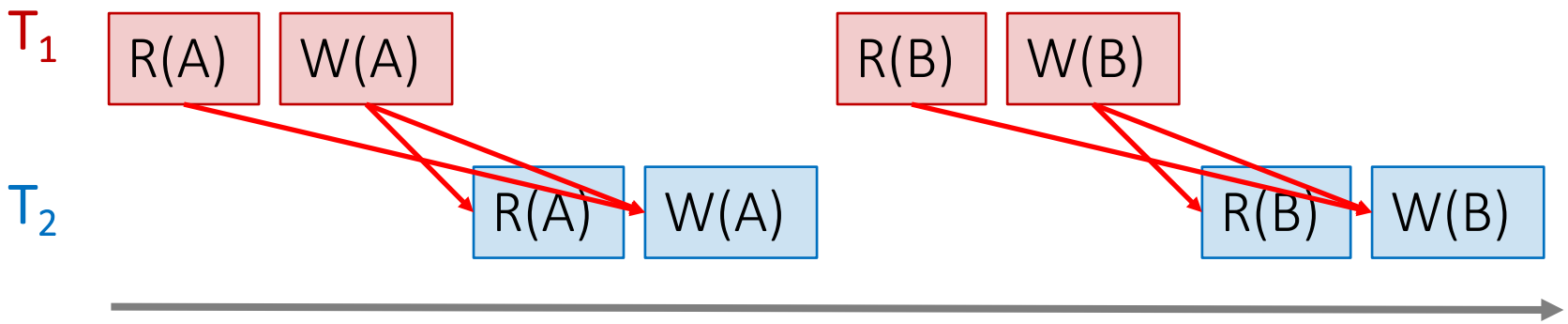
Conflicts

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

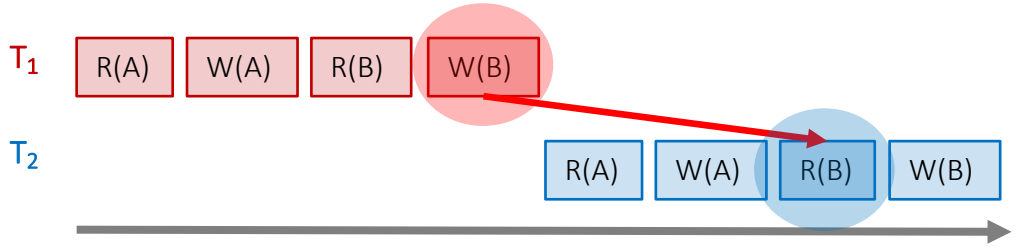
- Two schedules are conflict equivalent if:
  - They involve the same actions of the same TXNs
  - Every pair of conflicting actions of two TXNs are ordered in the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Conflict serializable ⇒ serializable

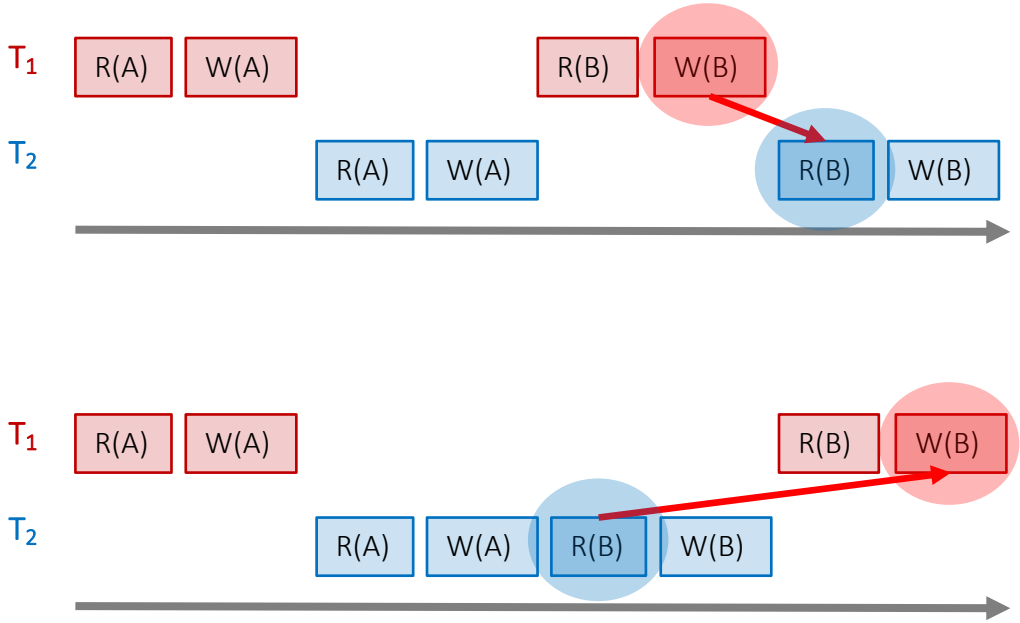
So if we have conflict serializable, we have consistency & isolation!

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

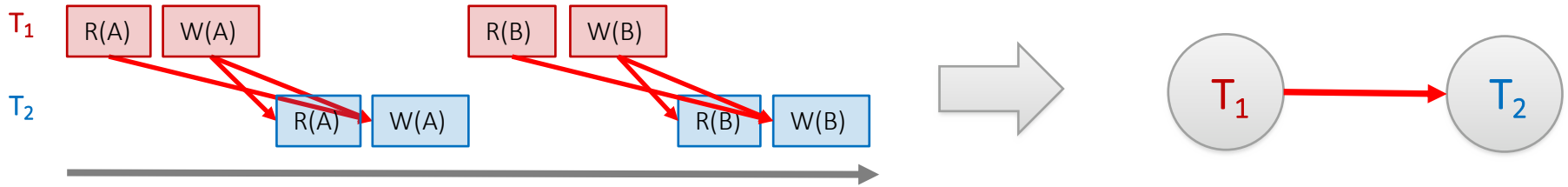
- Conflicts are things we talk about to help us characterize different schedules
  - Present in both “good” and “bad” schedules

- 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

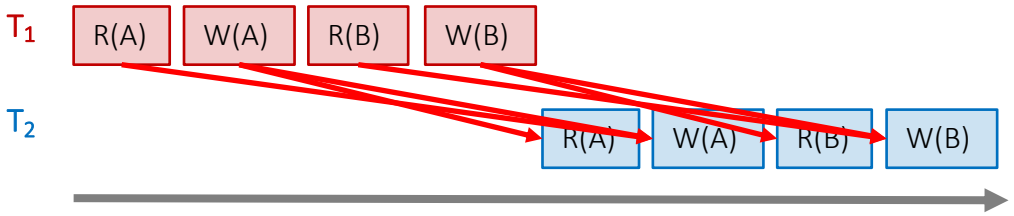
- Let's now consider looking at conflicts at the TXN level

- 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

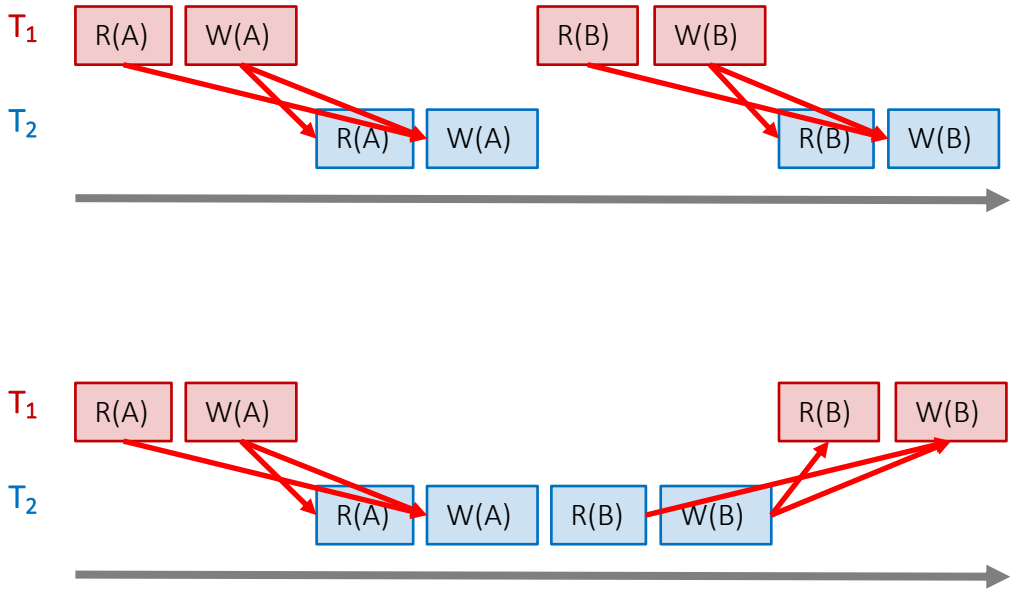


What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



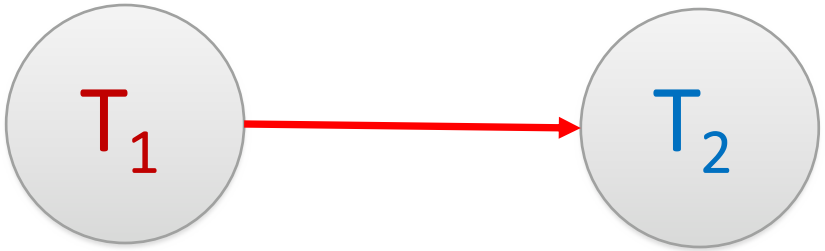
Interleaved Schedules:



A bit complicated…

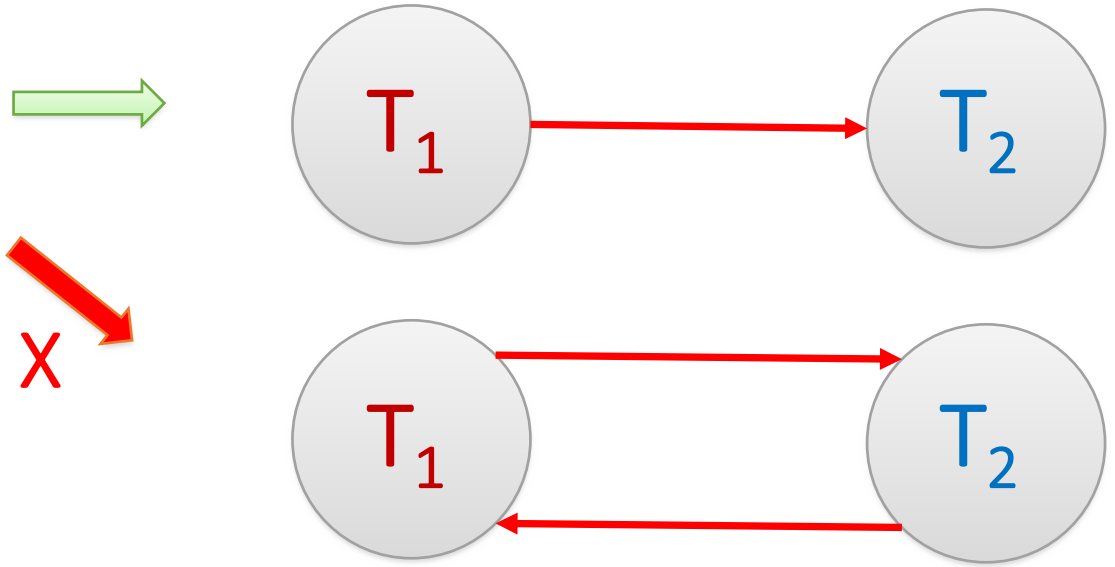
What can we say about "good" vs. "bad" conflict graphs?

Serial Schedule:



Simple!

Interleaved Schedules:



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

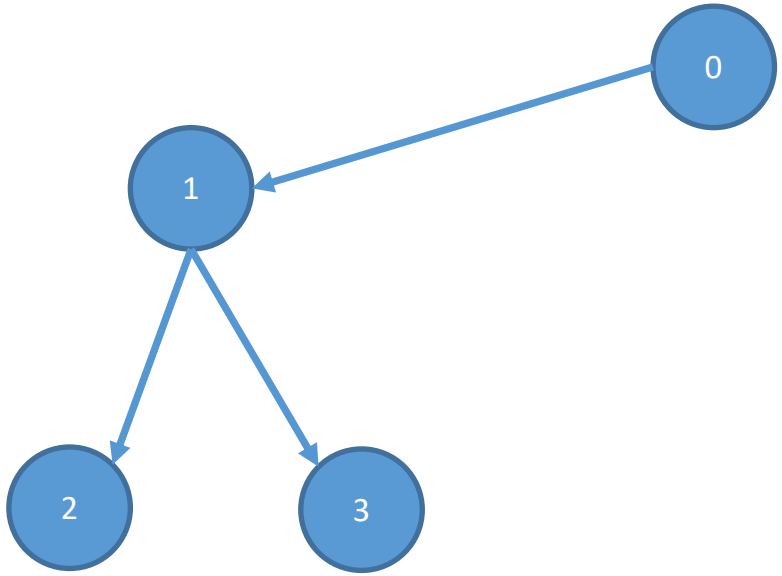
DAGs & Topological Orderings

- 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

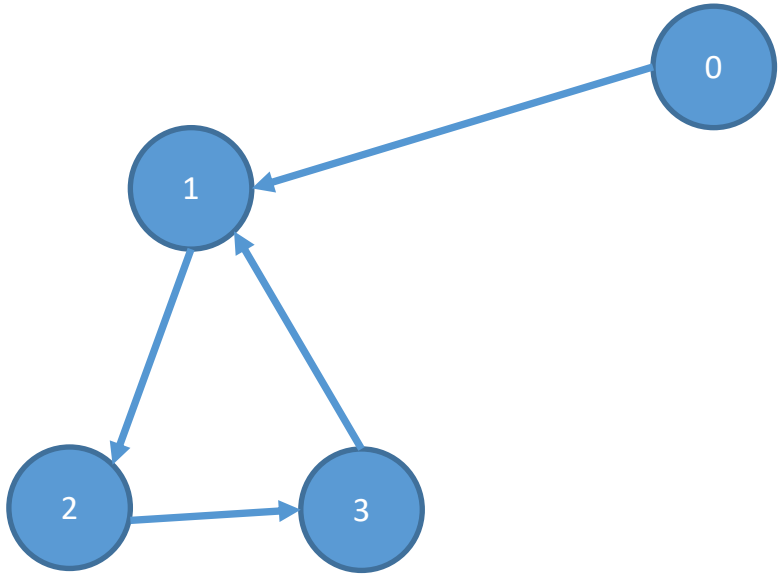
- Ex: What is one possible topological ordering here?



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 → conflict serializable!

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