Lectures 8 & 9: 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

If you want to build a TXN engine, CS245 is needed.

Some Comments

- **Omg, you're Piazza'ing!**
  - This is great! Keep it up!
  - Prizes have been given… more coming.

- I'm worried that some of you started so late on a coding project.
  - Messy data is frustrating. This is real data.
  - Please start early, it's stressful for us ☺

- **Late days.** You have three late days to unexpected issues—they shouldn't really be the default (especially not this early).

- MVDWe'll read MVDs on our own.
  - There is an activity as well.
  - If you have questions, ask on Piazza or we can use class!

Lecture 8: Intro to Transactions & Logging

Today's Lecture

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

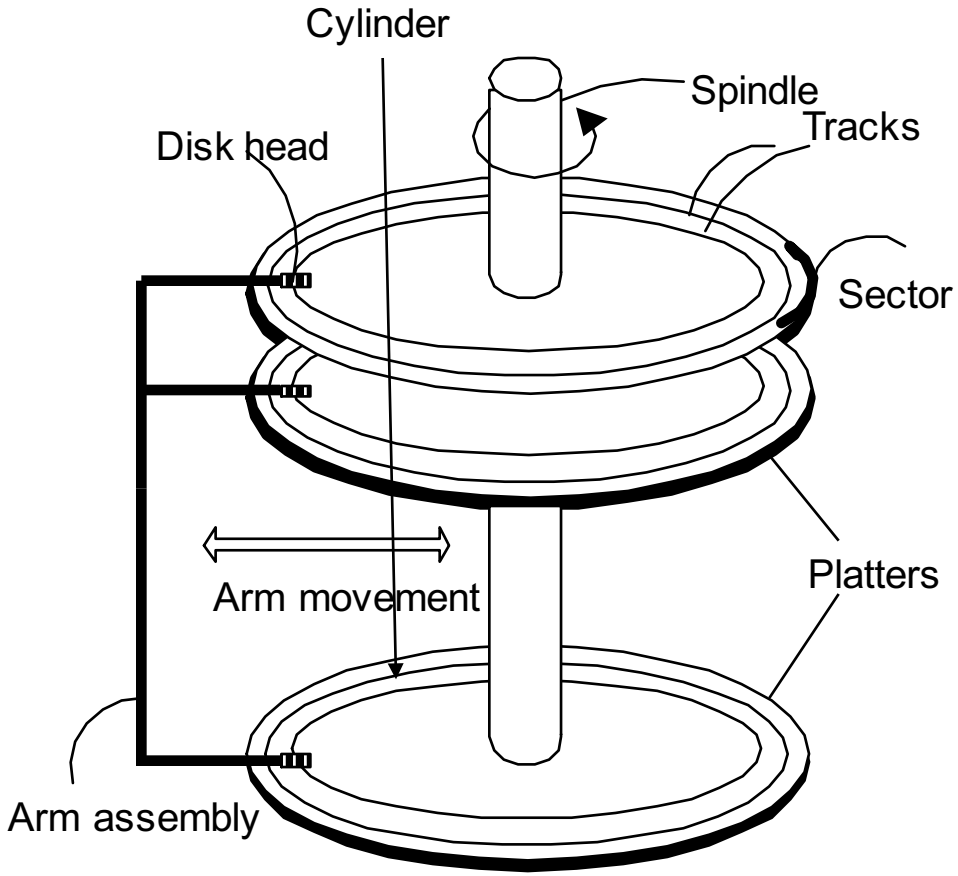
1. Transactions

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]*
5. ACTIVITY: ABORT!!!

High-level: Disk vs. Main Memory

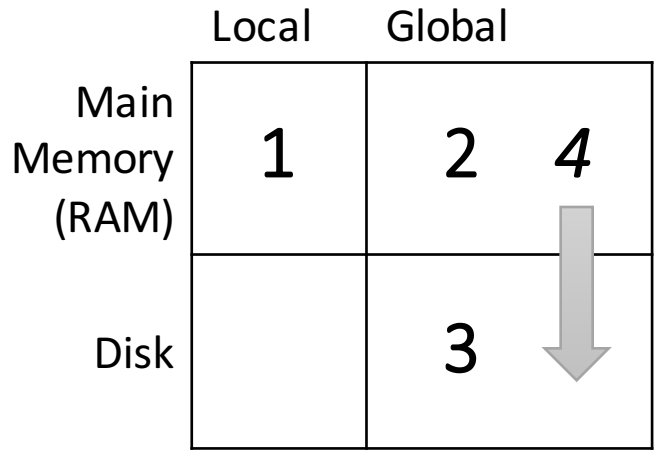
- **Disk**:
  - *Slow*
    - Sequential access
      - (although fast sequential reads)
  - *Durable*
    - We will assume that once on disk, data is safe!
  - Cheap



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 + erasing ("evicting") 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 coherent *transactions* helps with two goals:

1. **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

This lecture!

2. **Concurrency:** Achieving better performance by parallelizing TXNs *without* creating anomalies

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

        DELETE Product
            WHERE price <=0.99
```

Crash / abort!

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…

ACTIVITY: Aborts & TXNs in SQLite

- Instructions: In this activity we'll use SQLite **directly** (rather than via Ipython Notebooks) to demonstrate TXNs

1. Download the file abort.sql & take a look- what do you think is *supposed to* happen? What do you think *will* happen?

2. Run it: "sqlite3 < abort.sql"

3. View the *accounts* table in sqlite- what happened?
   1. Run "sqlite3"
   2. Type ".open bank.db", then "SELECT * FROM accounts"

4. Can you use the "BEGIN TRANSACTION" and "END TRANSACTION" commands to fix this scenario??

Note: on some computers you might need to use a semicolon: "BEGIN TRANSACTION:"

2. Properties of Transactions

What you will learn about in this section

1. Atomicity
2. Consistency
3. Isolation
4. Durability
5. ACTIVITY?

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 is/was 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

This lecture

- 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!)

Next lecture

And all this with… Performance!!

A Note: ACID is contentious!

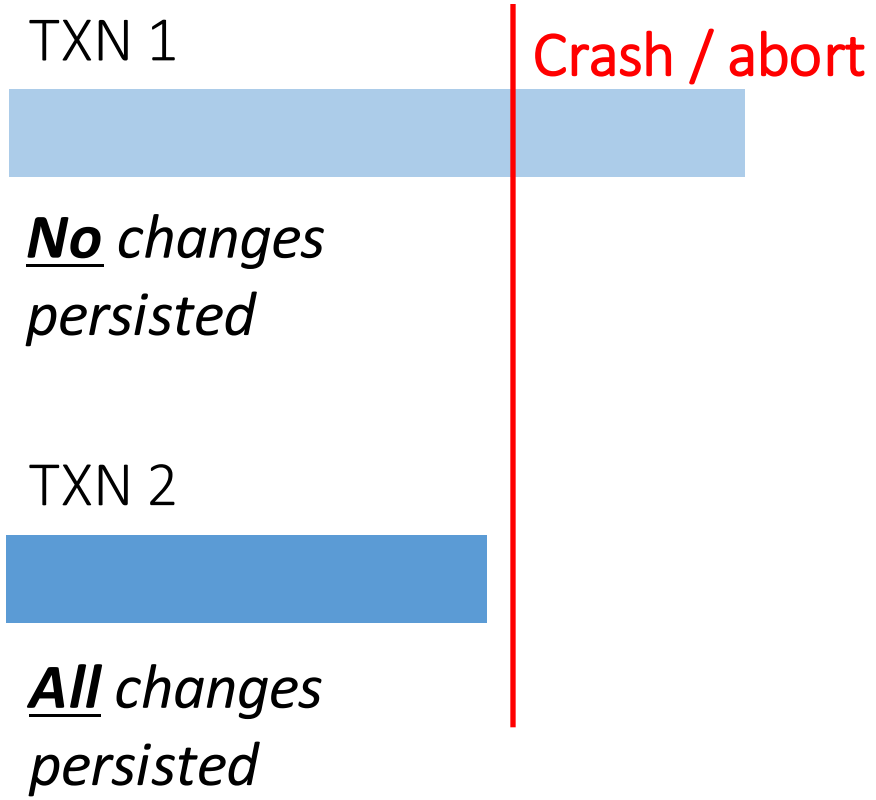
- Many debates over ACID, both **historically** and **currently**
- Many newer "NoSQL" DBMSs relax ACID
- In turn, now "NewSQL" reintroduces ACID compliance to NoSQL-style DBMSs…

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

Goal for this lecture: Ensuring Atomicity & Durability

ACID

- **A**tomicity:
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen
- **D**urability:
  - 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*** 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!

READ ABOUT Bitcoins & TXNs (or lack thereof…):
http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/

and/or time to ask CAs questions!

3. Atomicity & Durability via Logging

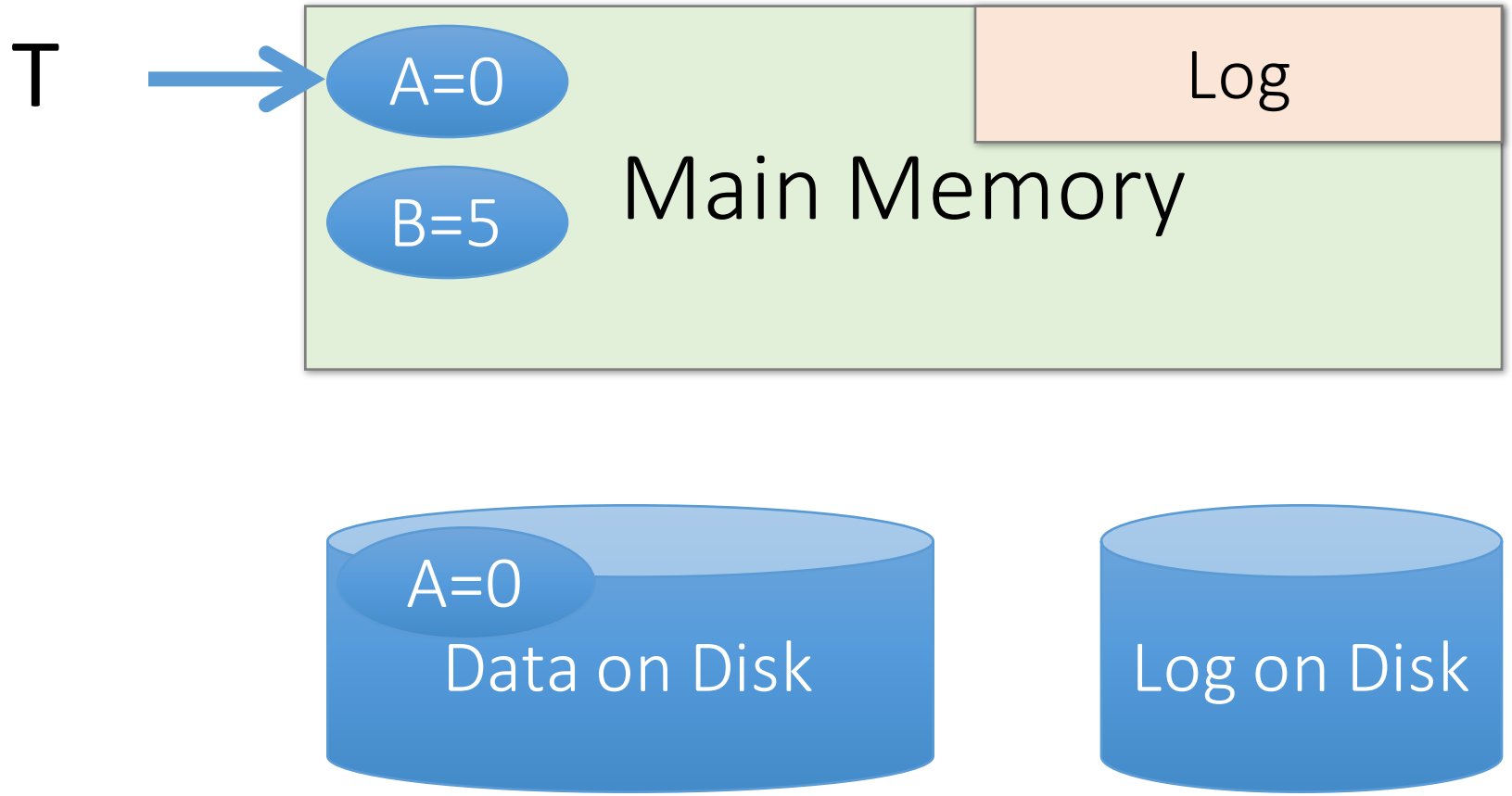
What you will learn about in this section

1. Logging: An animation of commit protocols

A Picture of Logging

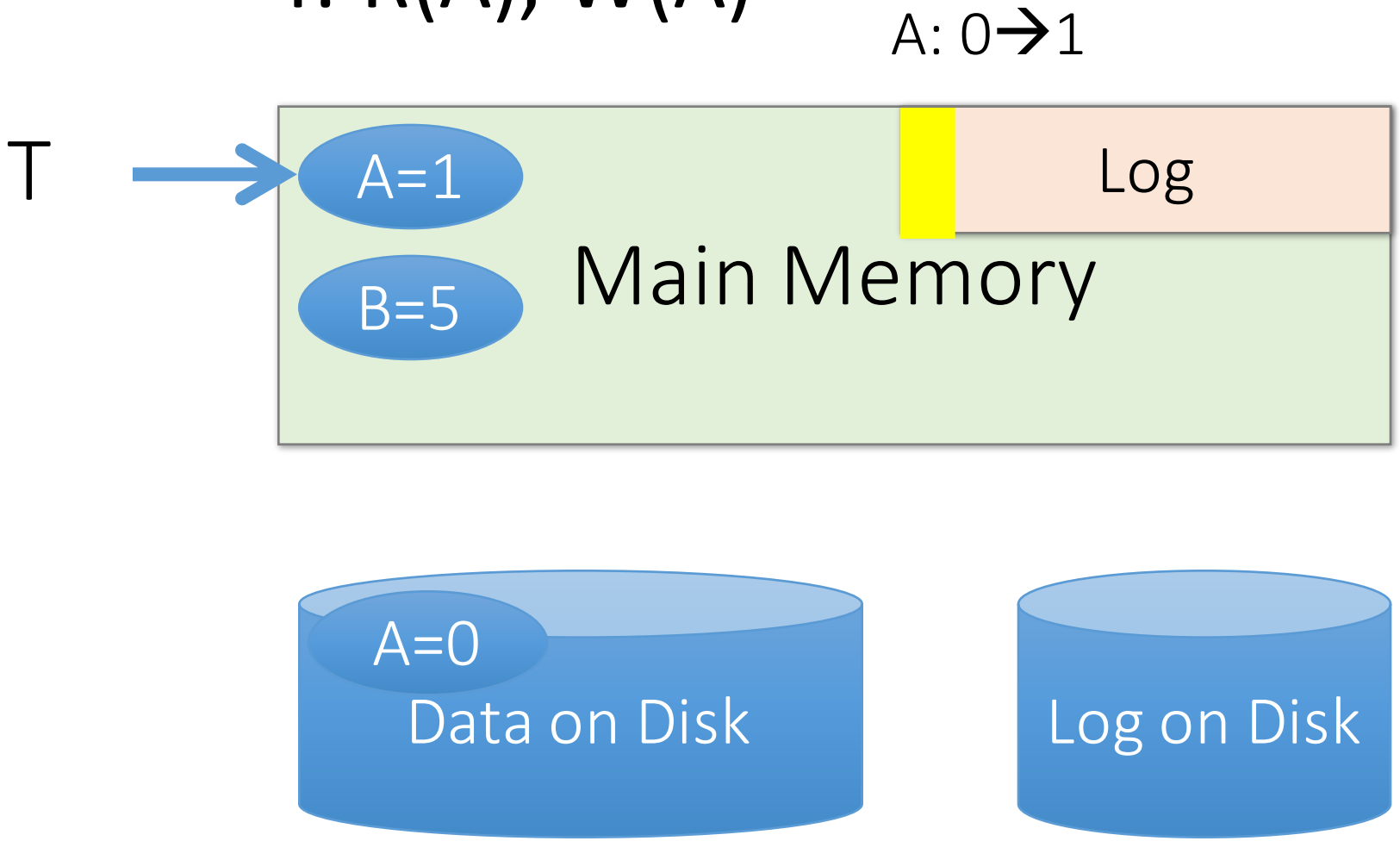
A picture of logging

T: R(A), W(A)



A picture of logging

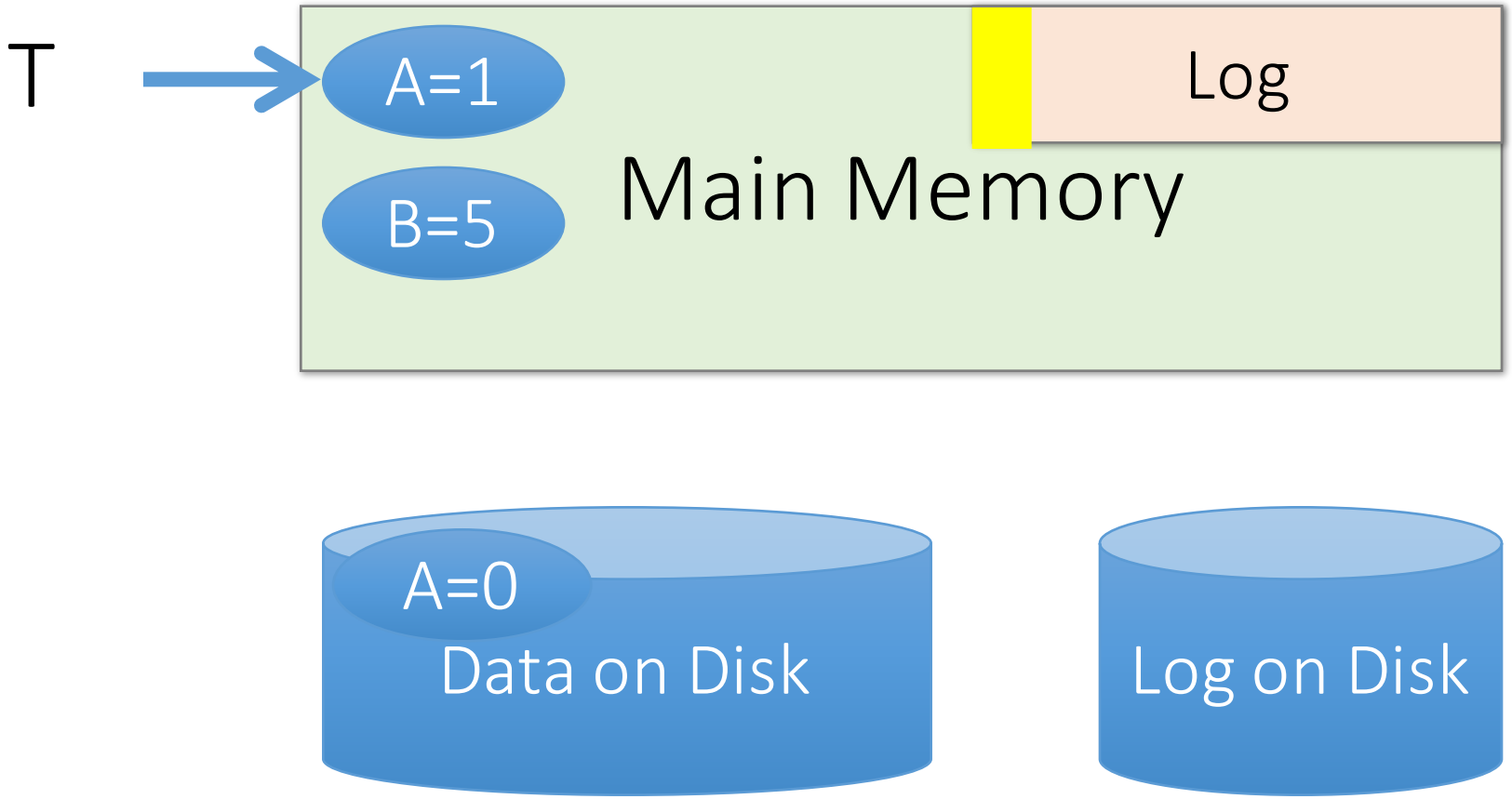
T: R(A), W(A)



A picture of logging

T: R(A), W(A)

A: 0→1



Operation recorded in log in main memory!

NB: Logging can happen after modification, but not before disk!

Let’s figure out WAL by making a bunch of mistakes without it!

(What can go wrong…)

Faulty scenario #1:

DBMS Writes A to disk without WAL…

A picture of logging, without WAL…

T: R(A), W(A)

A: 0→1

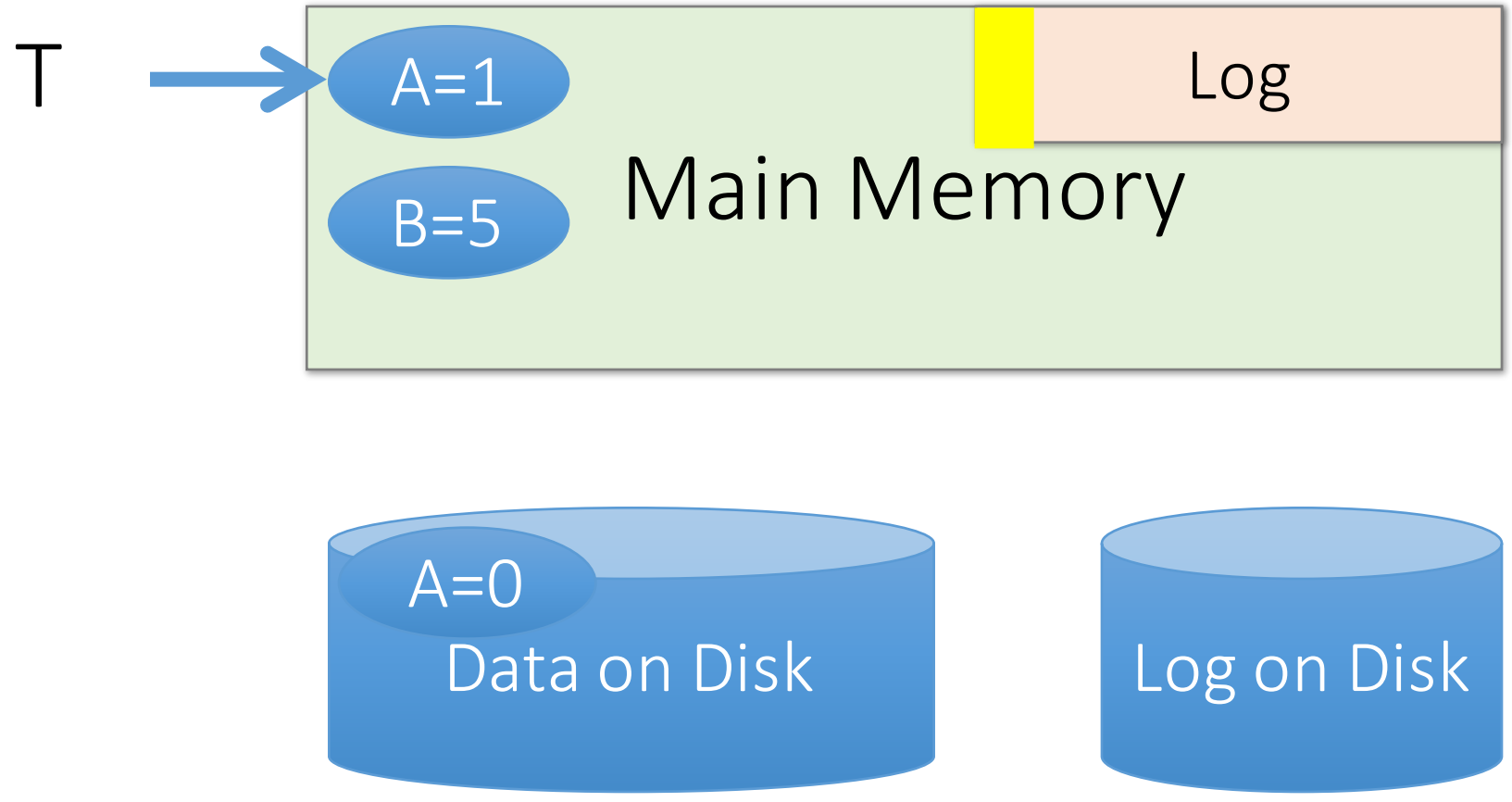
What happens if we crash or abort now, in the middle of T??

How do we "undo" T?

With WAL!

T: R(A), W(A)

A: 0→1



Now if we crash, we have the info to recover A…

However, what is the correct value?! Depends on commit!

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

T

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

T

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

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?

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 9: 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

Note the hard part…

…is the effect of *interleaving* transactions and *crashes*.
See 245 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
```

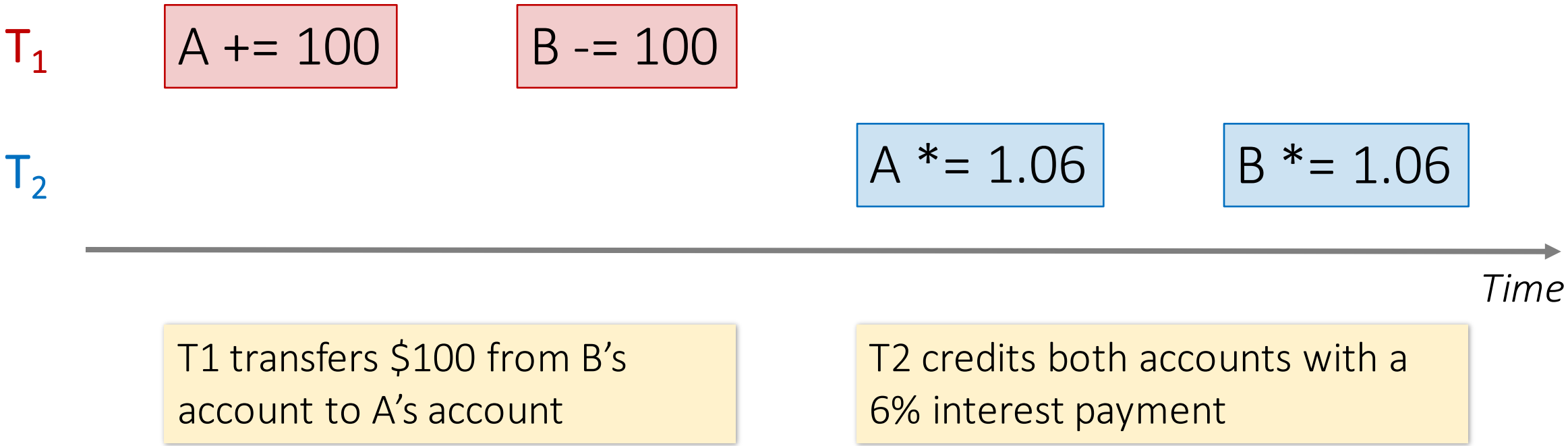
T1 transfers $100 from B's account to A's account

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

T2 credits both accounts with a 6% interest payment

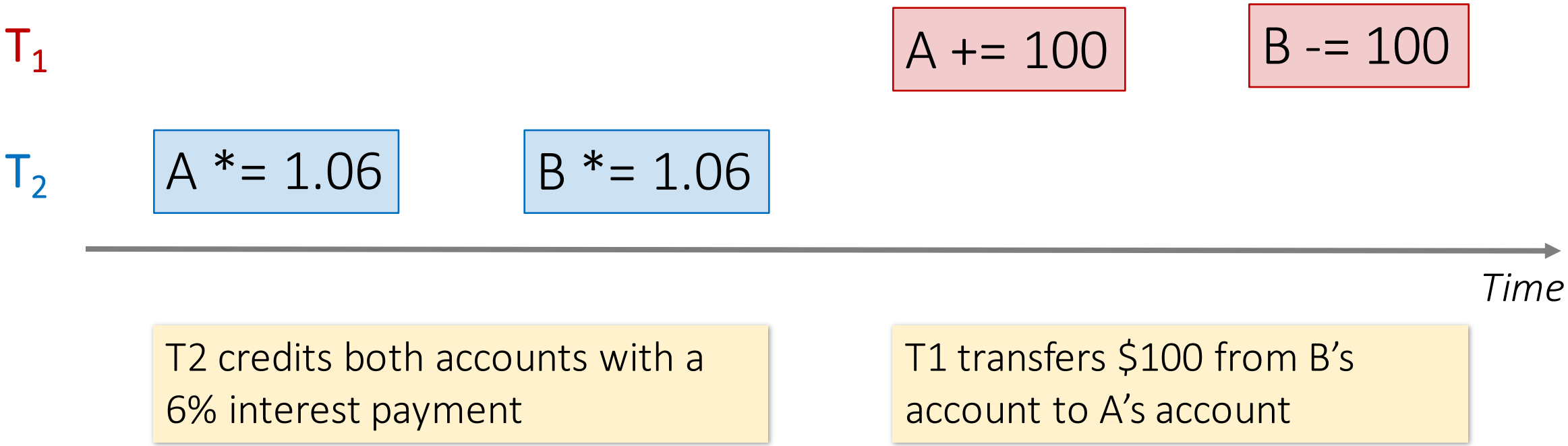
Example- consider two TXNs:

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



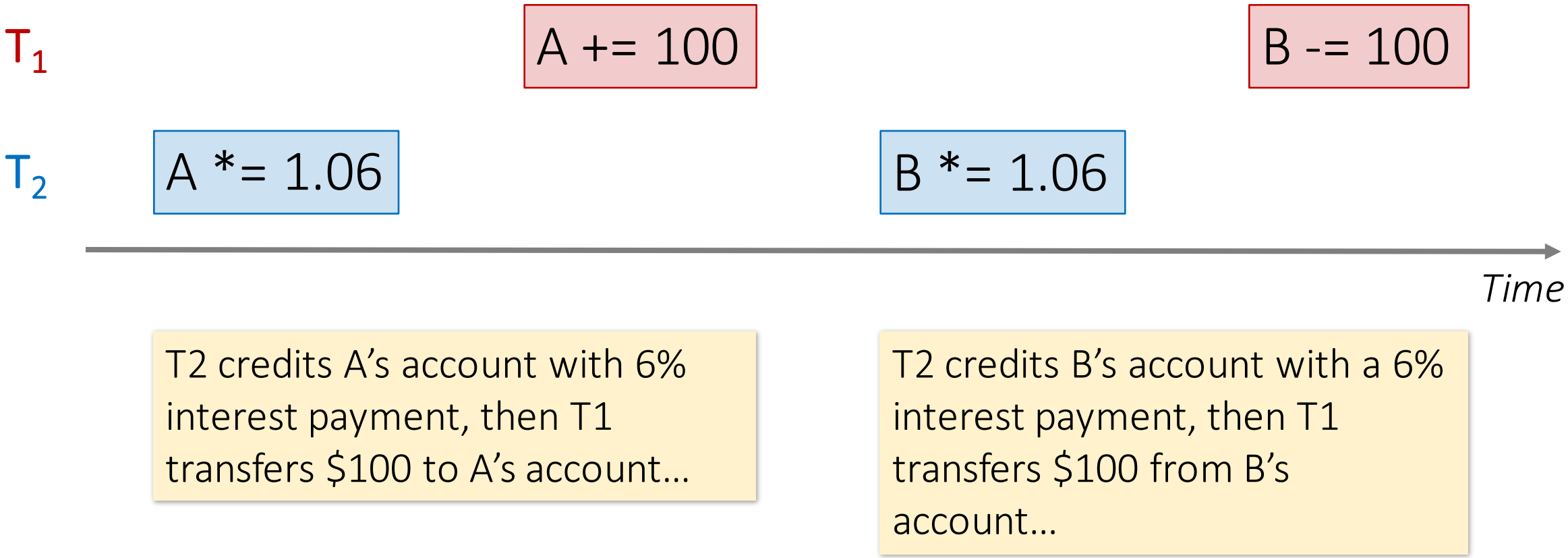
Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!



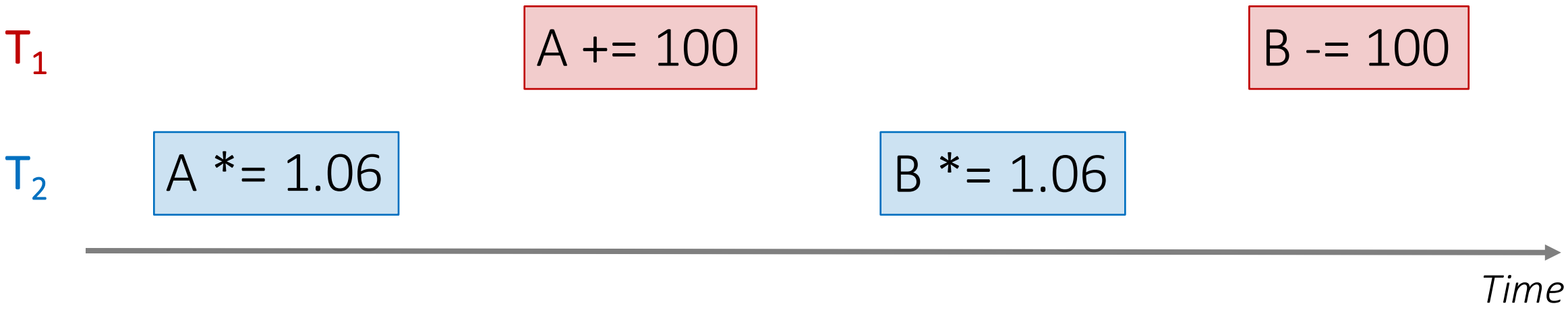
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



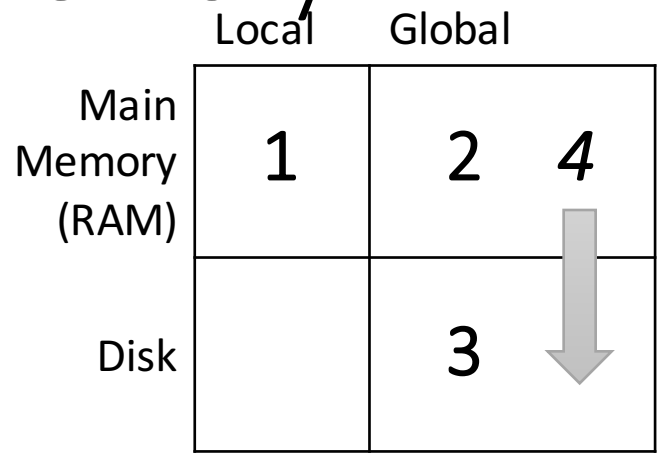
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



What goes / could go 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 + erasing ("evicting") 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

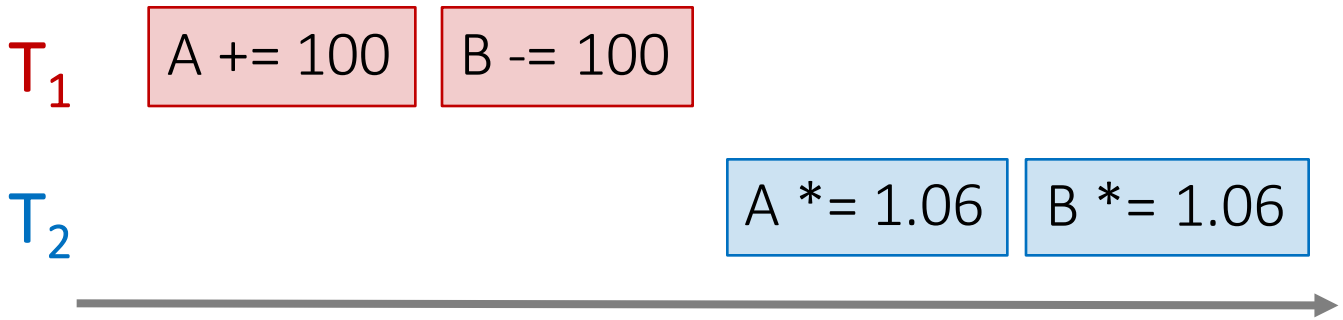
DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

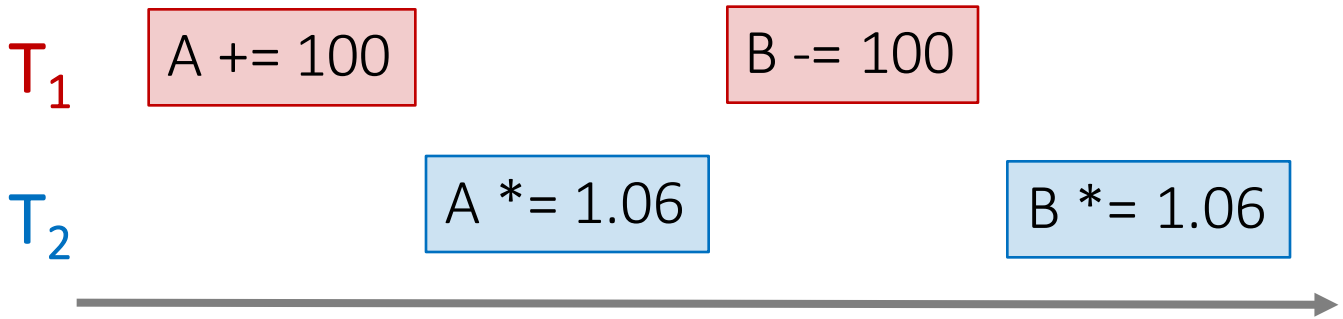
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule A:



A	B
$159	$106

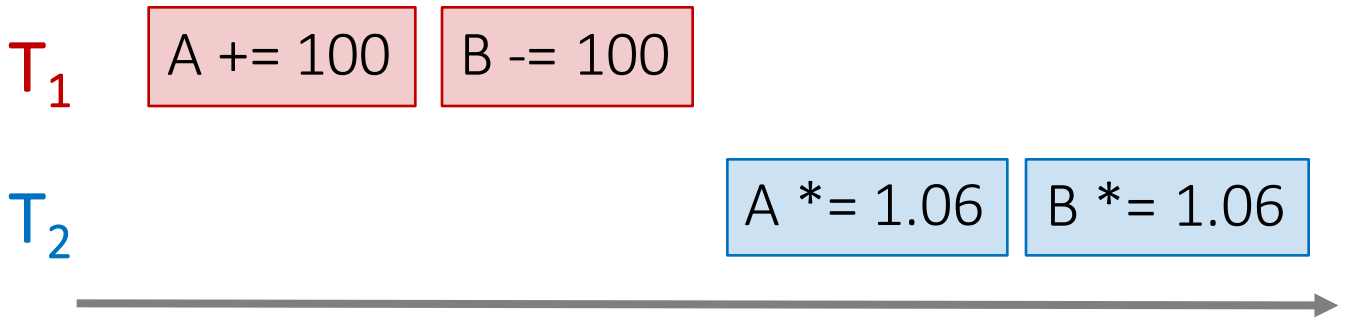
Same result!

Scheduling examples

Starting Balance

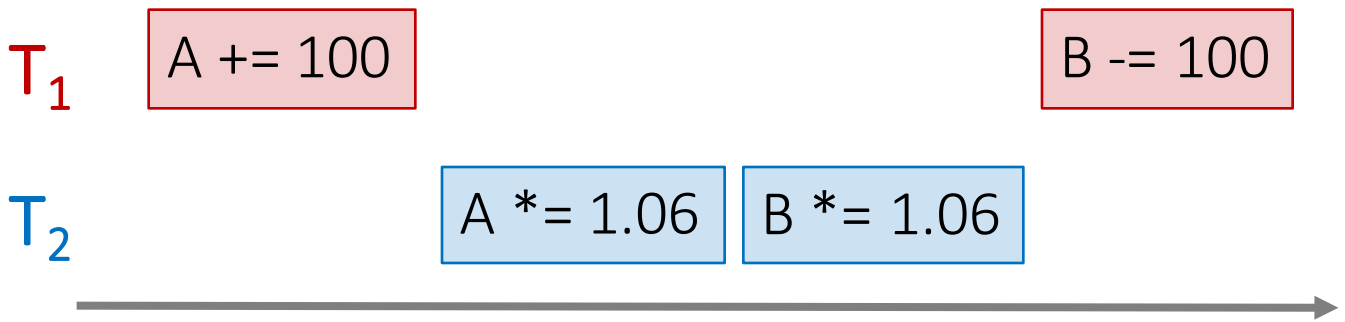
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule B:



A	B
$159	$112

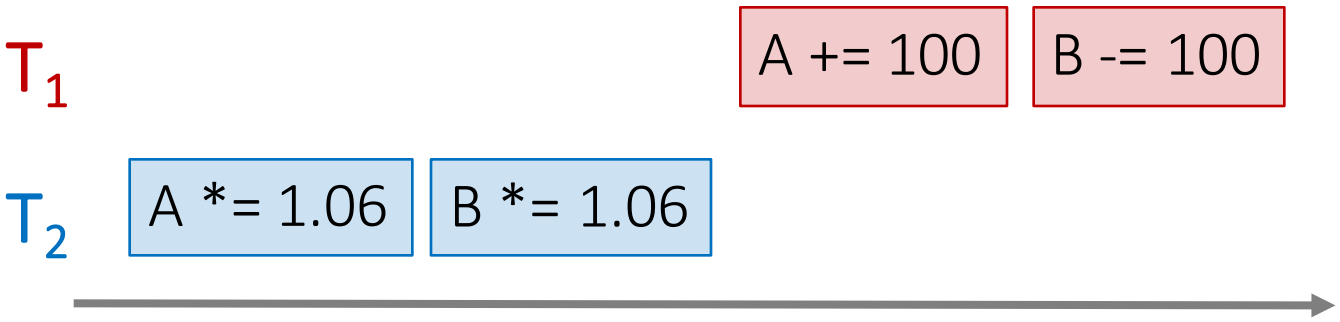
Different result than serial T_1,T_2!

Scheduling examples

Starting Balance

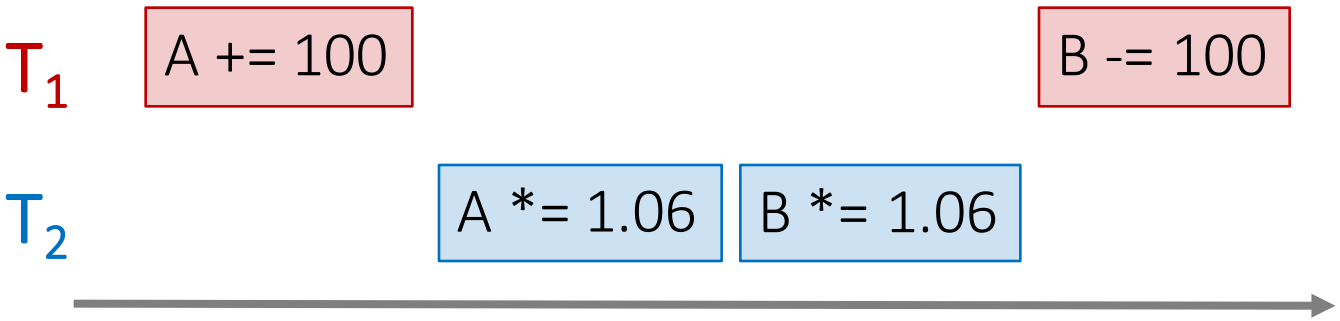
A	B
$50	$200

Serial schedule T_2,T_1:



A	B
$153	$112

Interleaved schedule B:

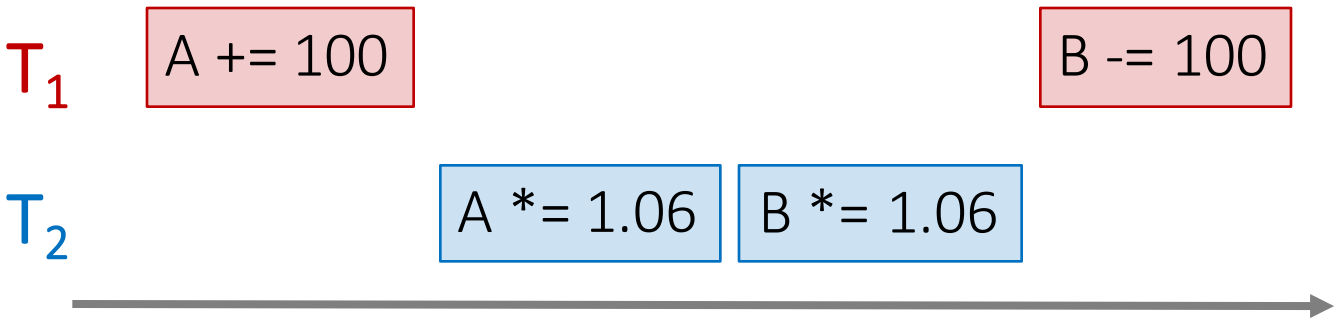


A	B
$159	$112

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

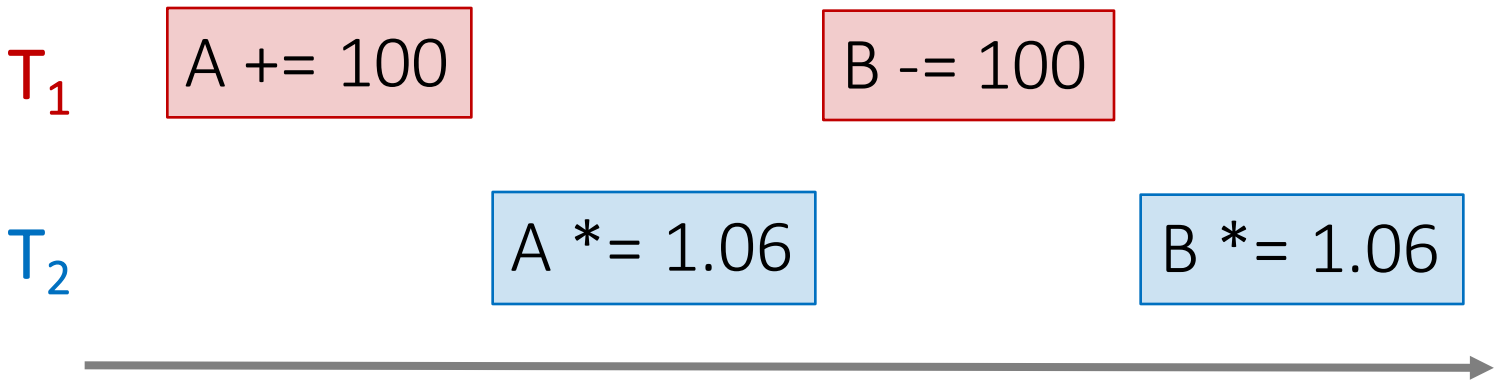
- A **serial schedule** is one that does not interleave the actions of different transactions

- 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 def powerful and tricky!

Serializable?



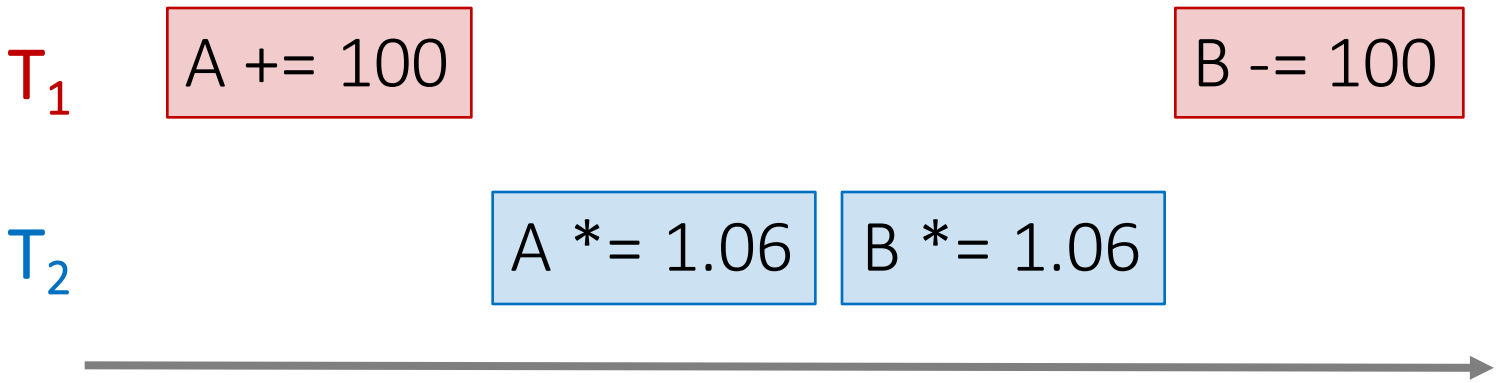
Serial schedules:

	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

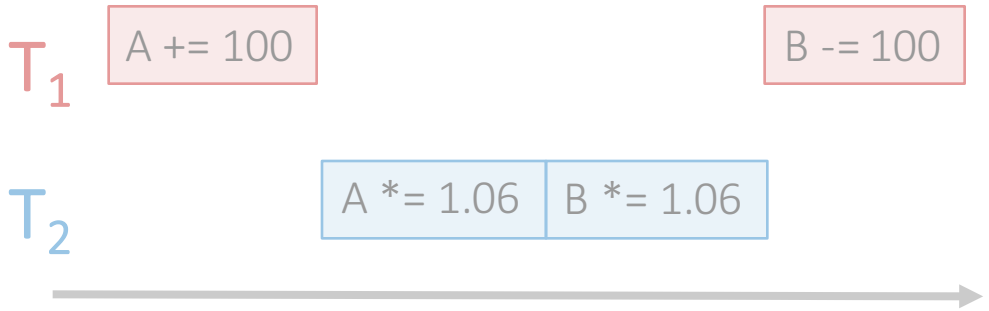
A	B
1.06*(A+100)	1.06*B - 100

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

What else can go wrong with interleaving?

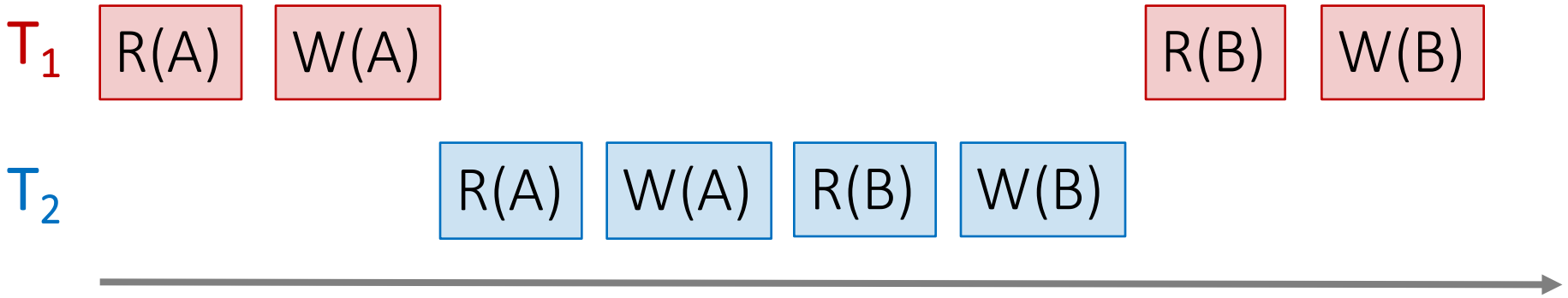
- Various anomalies which break isolation / serializability
  - Often referred to by name…
- Occur because of / with certain “conflicts” between interleaved TXNs

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)

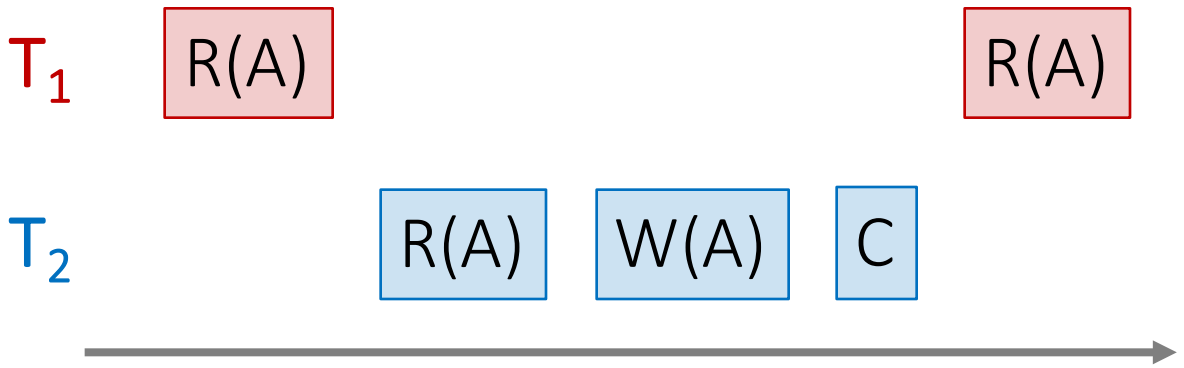
Why no "RR Conflict"?

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:



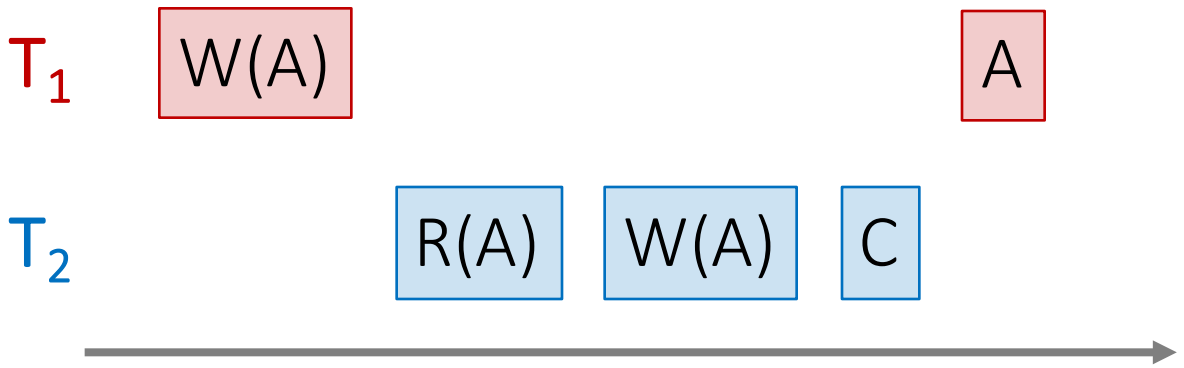
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

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

Classic Anomalies with Interleaved Execution

"Dirty read" / Reading uncommitted data:

Example:



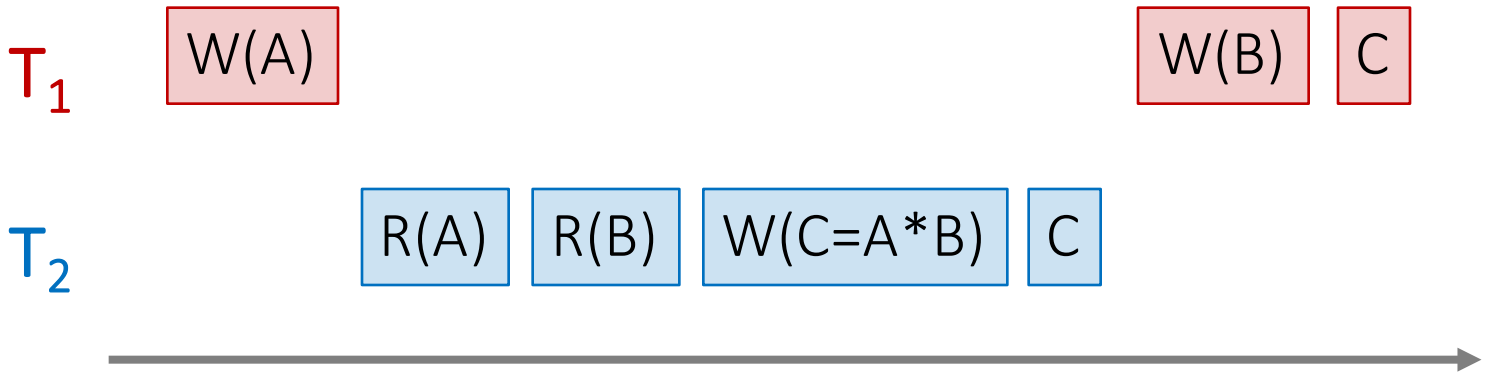
1. T_1 writes some data to A
2. T_2 reads from A, then writes back to A & commits
3. T_1 then aborts- *now T_2's result is based on an obsolete / inconsistent value*

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

Classic Anomalies with Interleaved Execution

"Inconsistent read" / Reading partial commits:

Example:



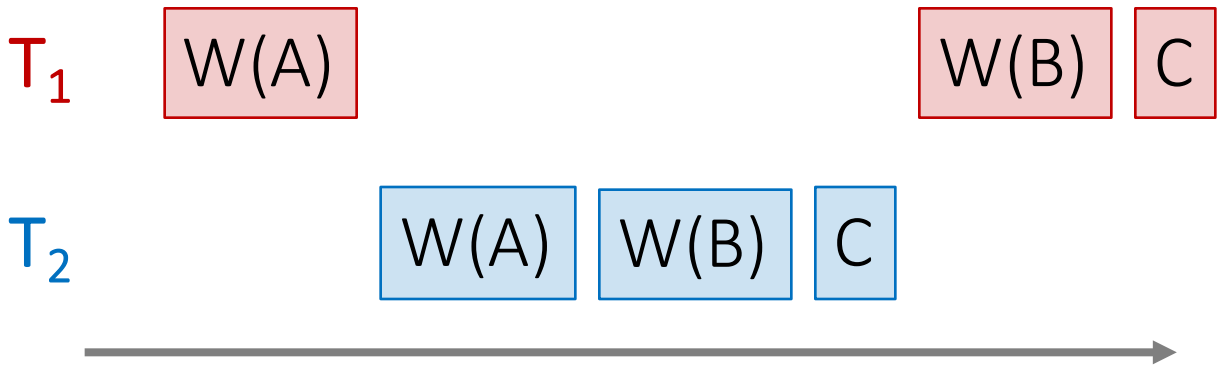
1. T_1 writes some data to A
2. T_2 reads from A *and B*, and then writes some value which depends on A & B
3. T_1 then writes to B- *now T_2's result is based on an incomplete commit*

*Again, occurring with / 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 with / because of a **WW conflict***

Activity-9-1.ipynb

2. Locking

What you will learn about in this section

1. Locking: basics & 2PL
2. Conflict serializability
3. Deadlock detection
4. ACTIVITY: TXN viewer

Motivation

- Ensure that TXNs remain **isolated** i.e. that they follow serializable schedules
  - So that we don't encounter any of the types of anomalies just covered!
- One method: **Locking**
  - We will cover a specific locking strategy, *strict two-phase locking (2PL)*

Locking to Avoid Conflicts

- We saw that all data anomalies due to concurrency involve **conflicts**

- We can avoid conflicts by making sure that **two or more TXNs never access the same variable at the same time**, unless they are all **reads**

Recall: Two actions conflict if they are:

- part of **different TXNs**,
- involve the **same variable**,
- ≥ one of them is a **write**

This is what ***locking*** is!

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

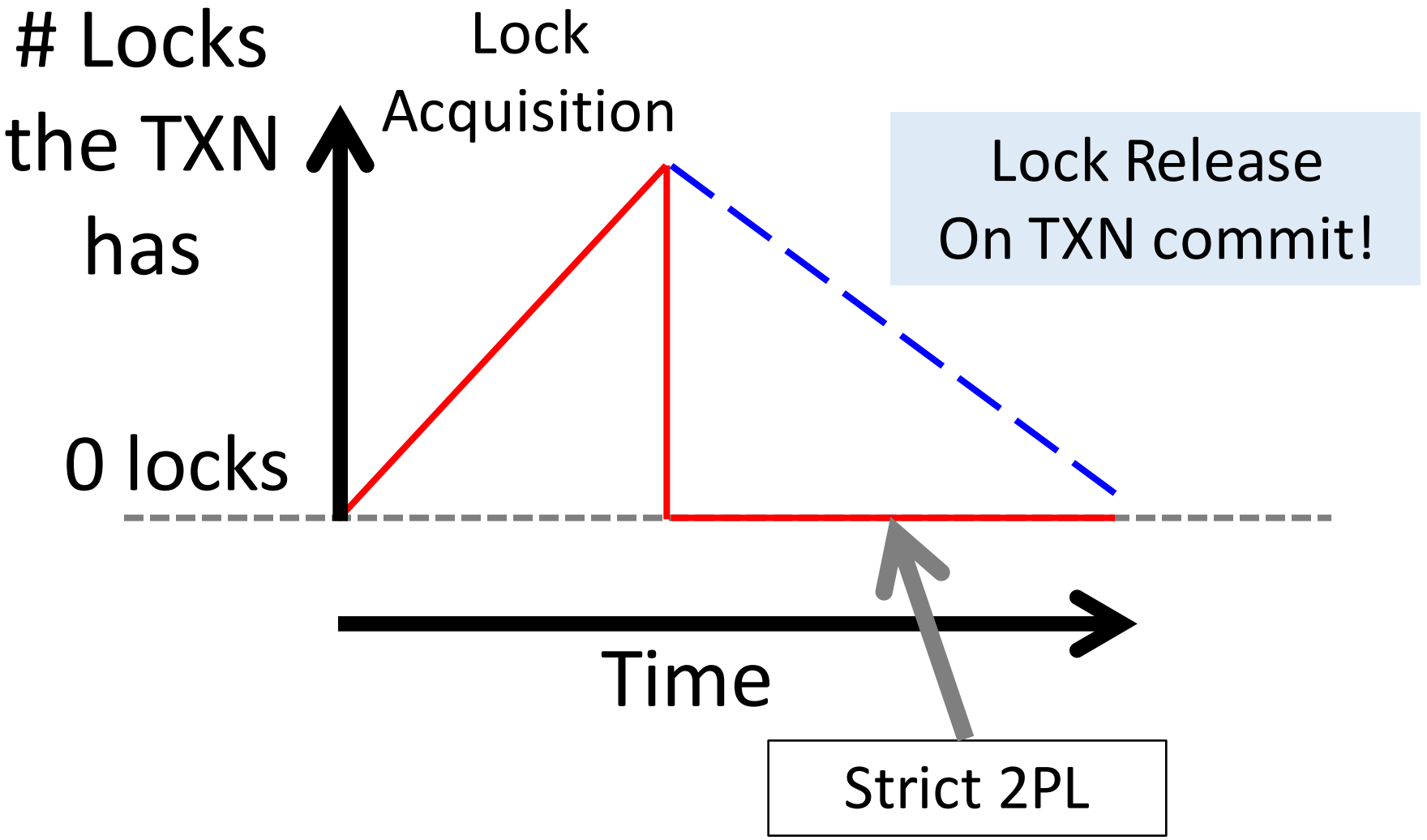
TXNs obtain:

- 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!

These policies ensure that no conflicts (RW/WR/WW) occur!

Picture of 2-Phase Locking (2PL)



Using Strict 2PL Locking & Serializabilty

Motivation

- You can't understand how your application works without understanding TXNs.
  - Serializability is a slippery notion!

- We'll study lock-based, which is the easiest to understand & essentially what the SQL standard is based on.
  - There are fancier things too (see 245)

Conflict Serializable Schedules

- Two schedules are **conflict equivalent** if:
  - 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

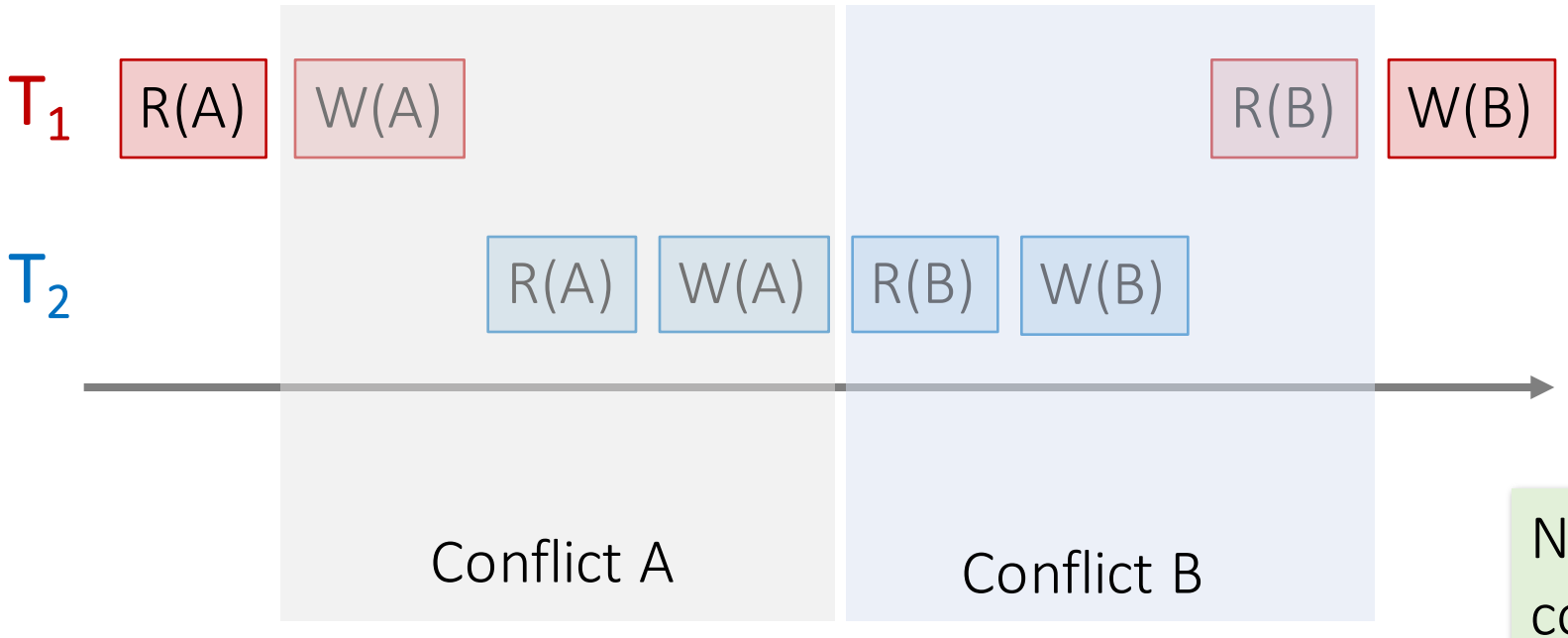
Recall: Two actions **conflict** if they are:

- part of **different TXNs**,
- involve the **same variable**,
- ≥ one of them is a **write**

If a schedule is conflict serializable, then it maintains isolation & consistency- why we care about!

Example

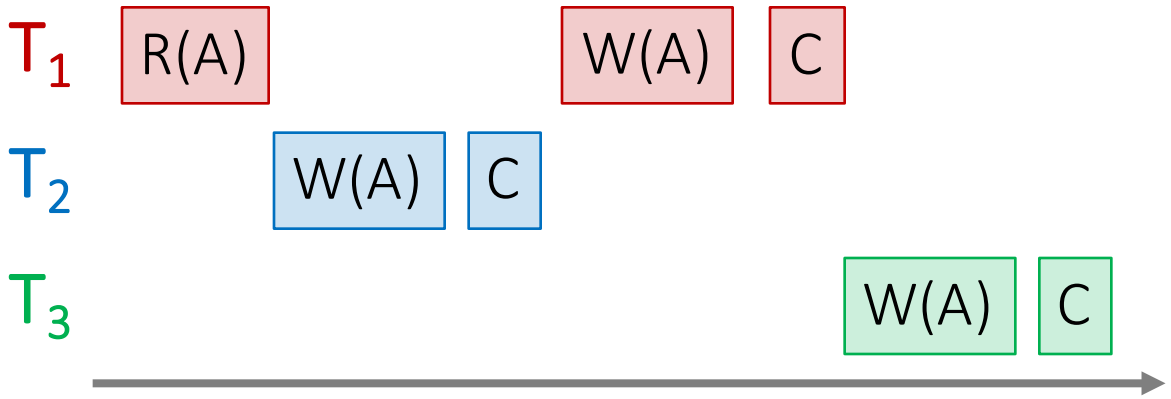
A schedule that is **not** conflict serializable:



No way for the actions of conflicts A & B to ***both*** happen in this order in a serial schedule!

Serializable vs. Conflict Serializable

Example of serializable but ***not*** conflict serializable



This is ***equivalent*** to T_1,T_2,T_3, so serializable

But not conflict ***equivalent*** to T_1,T_2,T_3 (or any other serial schedule) so not conflict serializable!

Conflict serializable ⇒ serializable, but not the other way around

Conflict Dependency Graph

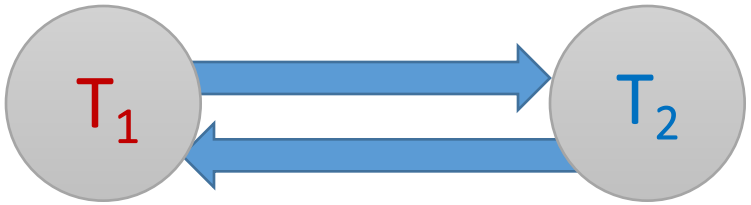
- Node for each committed TXN $T_1 \ldots T_N$
- Edge from $T_i \rightarrow T_j$ if an actions in T_i **precedes** and **conflicts** with an action in T_j

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

Conflict Dependency Graph

Example:

Conflict dependency graph:



A non-conflict serializable schedule has a **cyclic** conflict dependency graph!

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 $T_j \rightarrow T_i$

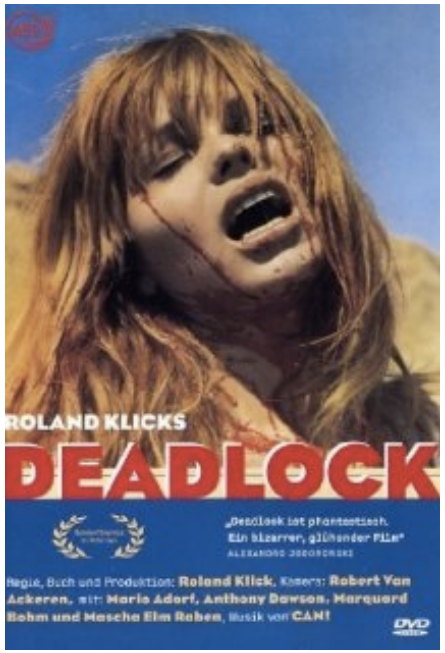
Therefore, Strict 2PL only allows conflict serializable ⇒ serializable schedules

Summary So far

- If a schedule follows strict 2PL and locking, it is serializable. Yes!
- Not all serializable schedules are allowed by strict 2PL.
- So let's use strict 2PL, what could go wrong?

```
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 Prevention

- Assign priorities based on timestamps. Assume T_i wants a lock that T_j holds. Two policies are possible:
  - *Wait-Die*: If T_i has higher priority, T_i waits for T_j; otherwise T_i aborts
  - *Wound-wait*: If T_i has higher priority, T_j aborts; otherwise T_i waits
- Note: If a transaction re-starts, make sure it has its original timestamp

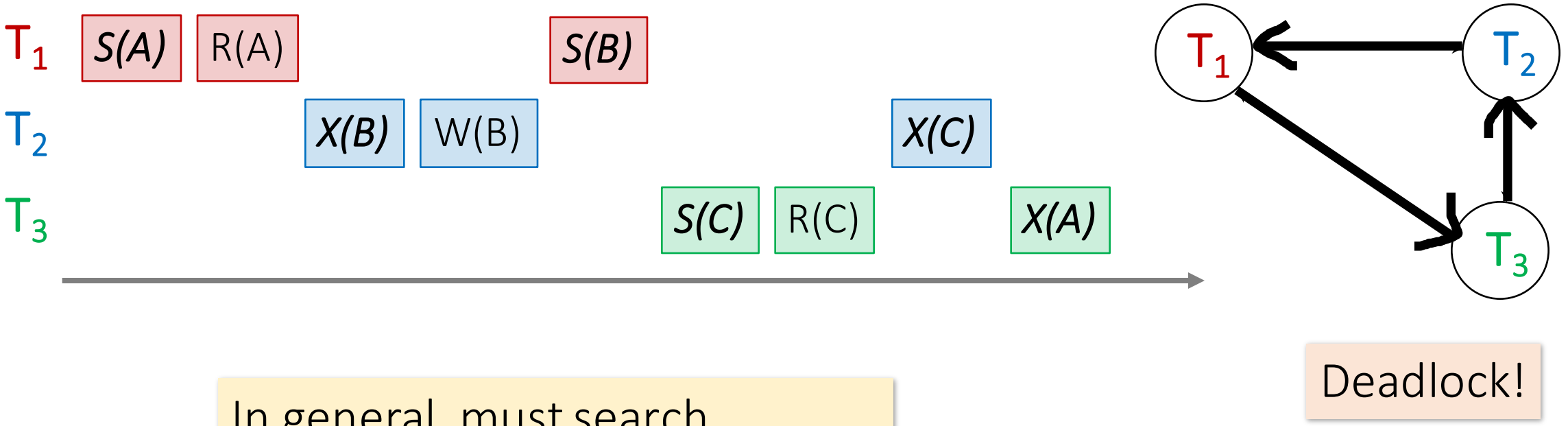
Issue: What if a transaction never makes progress?

Deadlock Detection

- Create a **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:



Deadlock!

In general, must search through this big graph. Sounds expensive! Is it?

Activity-9-2.ipynb

Locking Summary

- Locks must be atomic, primitive operation
- 2PL does not avoid deadlock
- Deadlock detection sounds more expensive than it is….