CMPT 354: Database System I

Lecture 11. Transaction Management

Why this lecture

- **DB application developer**
  - What if crash occurs, power goes out, etc?
  - Single user → Multiple users

Turing Awards in Data Management

Charles Bachman, 1973
IDS and CODASYL

Ted Codd, 1981
Relational model

Jim Gray, 1998
Transaction processing

Michael Stonebraker, 2014
INGRES and Postgres

Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - Anomaly Types
  - Conflict Serializability

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
```

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

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!

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 Employee
              SET Salary = Salary + 1000


Client 2:     UPDATE Employee
              SET Salary = Salary * 2
```

Two managers attempt to increase employee salary *concurrently*- What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
              UPDATE Employee
              SET Salary = Salary + 1000
            COMMIT

Client 2: START TRANSACTION
              UPDATE Employee
              SET Salary = Salary * 2
             COMMIT
```

Now works like a charm- we'll see how / why later…

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: Atomic

- 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: Consistent

- The tables must always satisfy user-specified ***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: Isolated

- 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: Durable

- 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**

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**

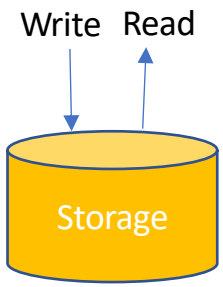
- Many newer "NoSQL" DBMSs relax ACID

- In turn, now "NewSQL" reintroduces ACID compliance to NoSQL-style DBMSs…

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

Transaction Management (Big Picture)

Definition



Transaction = A list of writes and reads

For example: {Read, Read, Write}

Two Big Problems

1. Support multiple transaction at the same time

2. Make sure the data stored is reliable

Techniques

1. Concurrency Control

2. Database Recovery

Properties

Atomicity

Consistency

Isolation

Durability

Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties

- **Concurrency Control**
  - **Scheduling**
  - **Anomaly types**
  - **Conflict serializability**

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
```

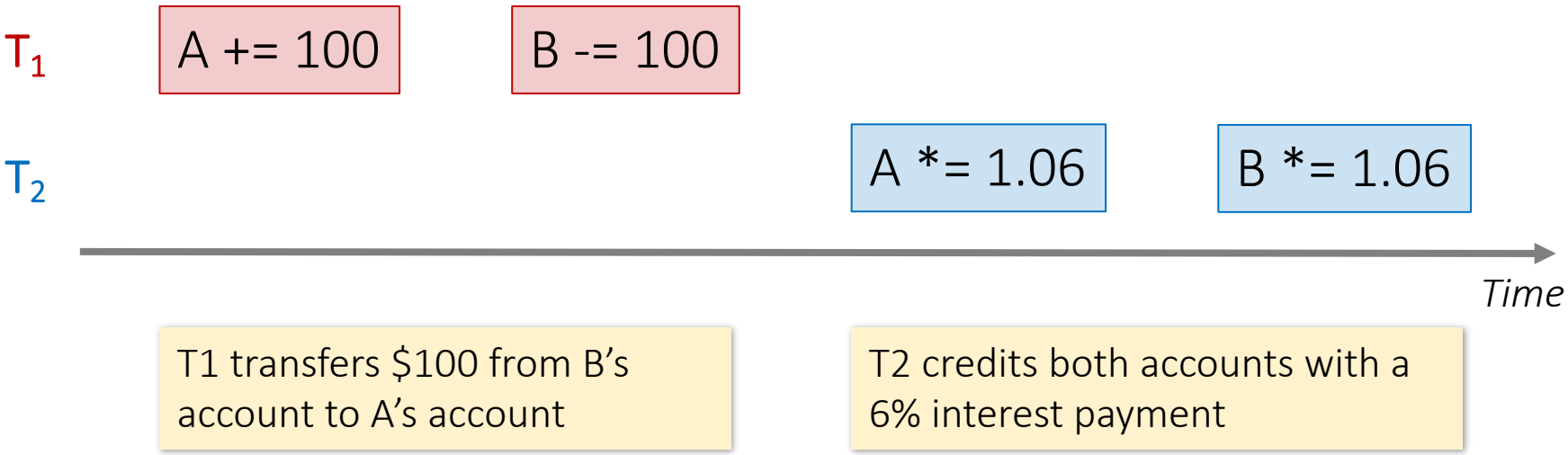
```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

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

T2 credits both accounts with a 6% interest payment

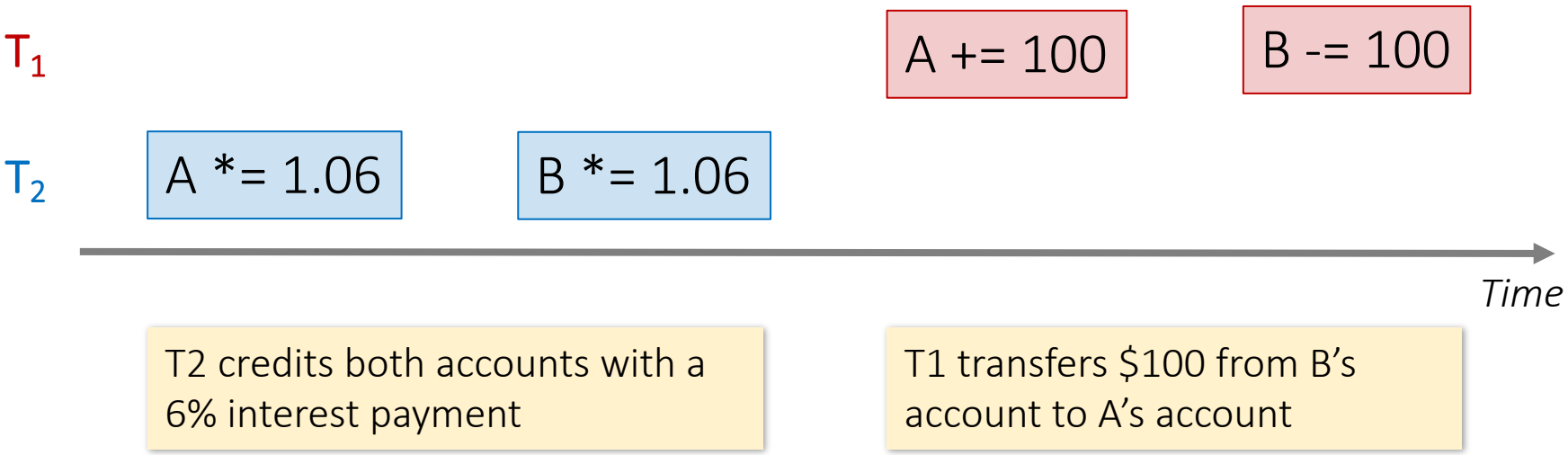
Example- consider two TXNs:

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



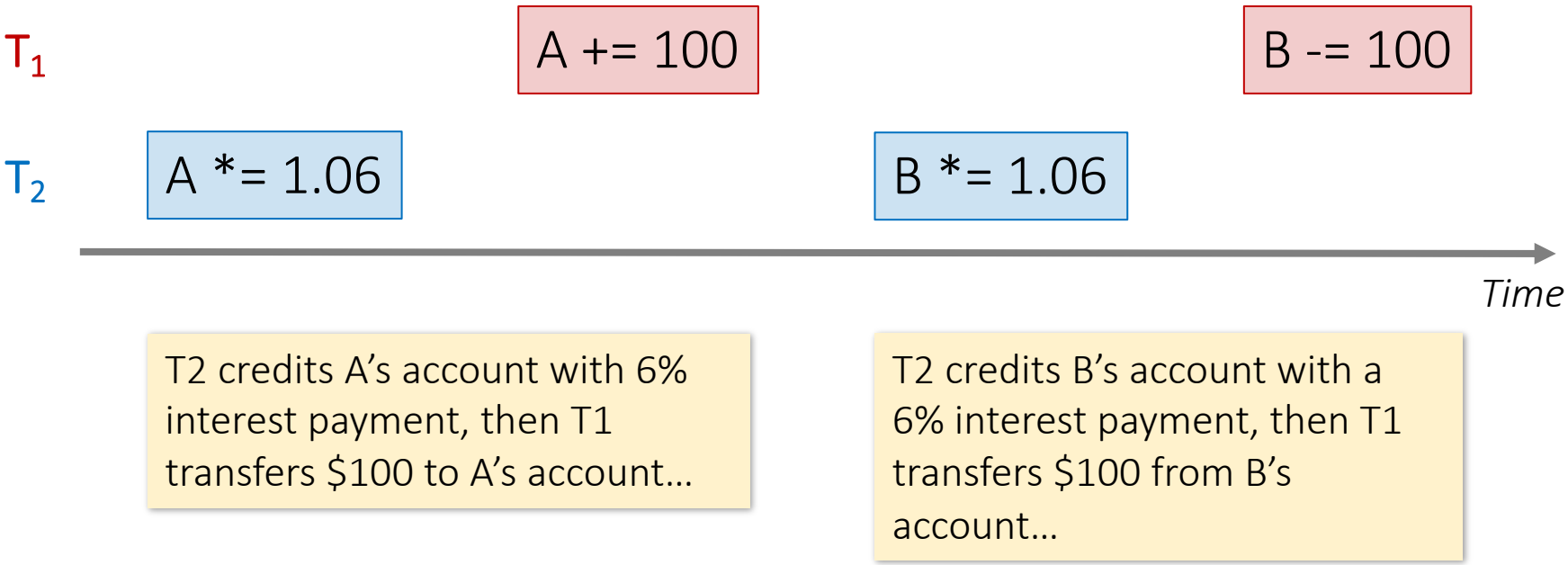
Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!



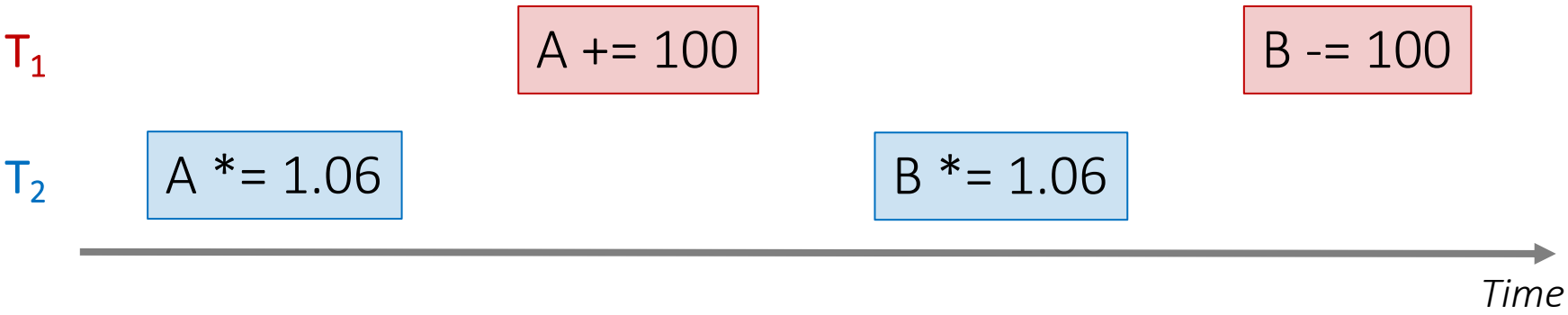
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



Is it correct?

Why Interleave TXNs?

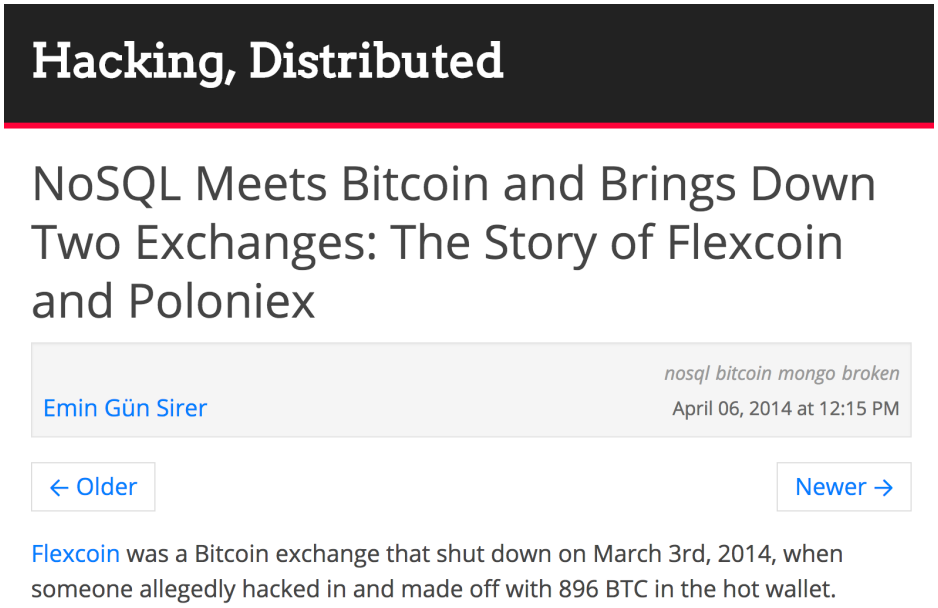
- Interleaving TXNs might lead to anomalous outcomes… why do it?

- 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***

Ignore all issues?

- At Facebook, only 0.0004% of results returned are inconsistent
- But,



Hacking, Distributed

NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex

nosql bitcoin mongo broken

Emin Gün Sirer — April 06, 2014 at 12:15 PM

← Older Newer →

Flexcoin was a Bitcoin exchange that shut down on March 3rd, 2014, when someone allegedly hacked in and made off with 896 BTC in the hot wallet.

Interleaving & Isolation

- The DBMS has freedom to interleave TXNs

"With great power comes great responsibility"

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

ACID

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

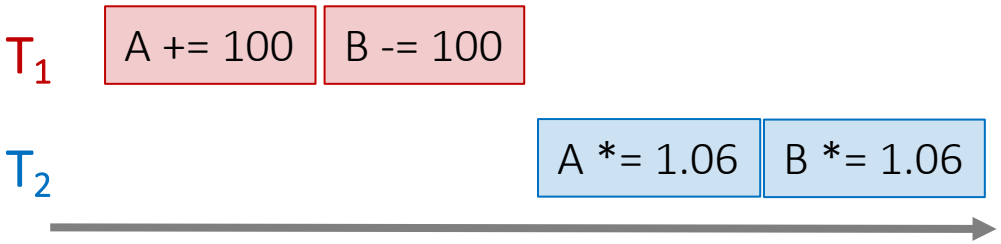
DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance

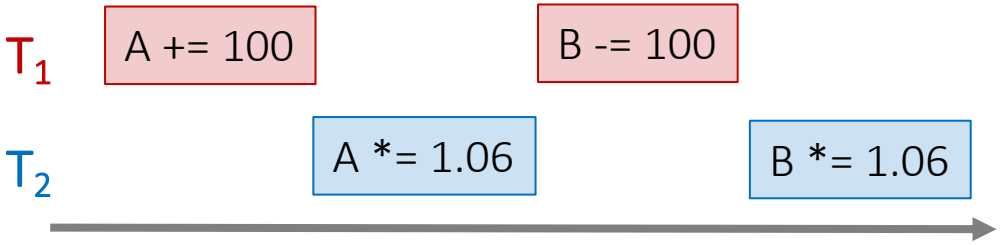
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule 1:



A	B
$159	$106

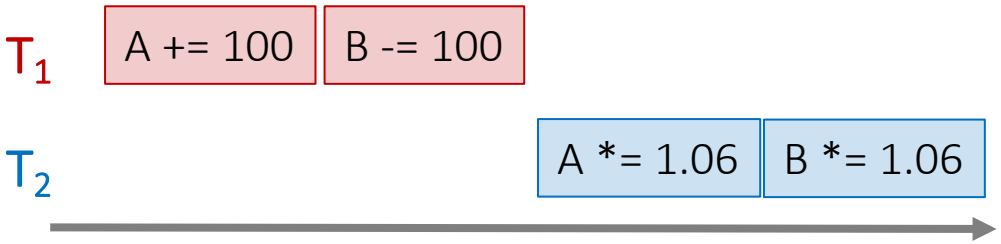
Same result!

Scheduling examples

Starting Balance

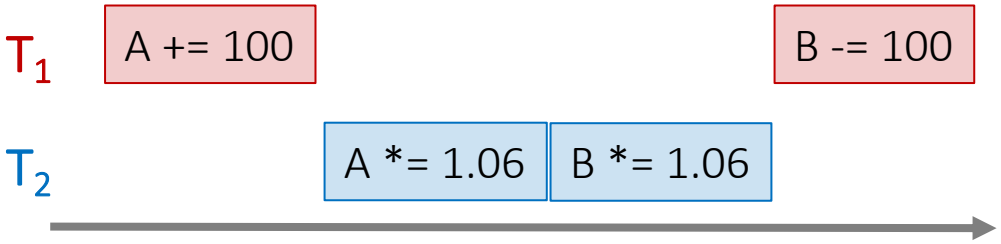
A	B
$50	$200

Serial schedule T_1,T_2:



A	B
$159	$106

Interleaved schedule 2:



A	B
$159	$112

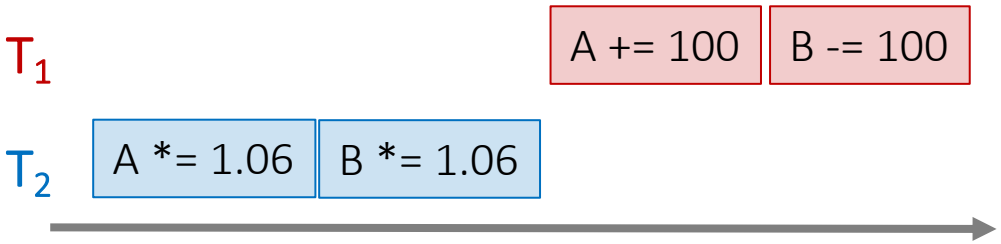
Different result than serial T_1,T_2!

Scheduling examples

Starting Balance

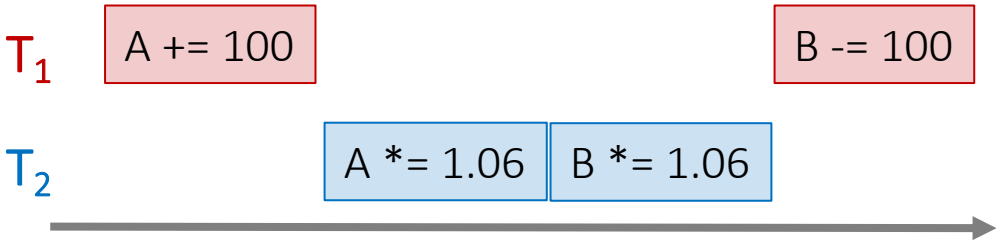
A	B
$50	$200

Serial schedule T_2,T_1:



A	B
$153	$112

Interleaved schedule 2:

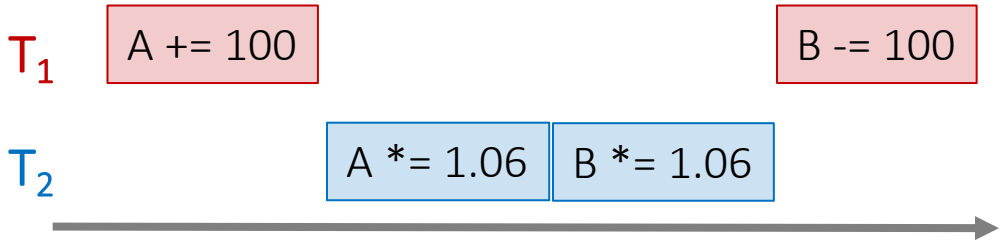


A	B
$159	$112

Different result than serial T_2,T_1 ALSO!

Scheduling examples

Interleaved schedule B:



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

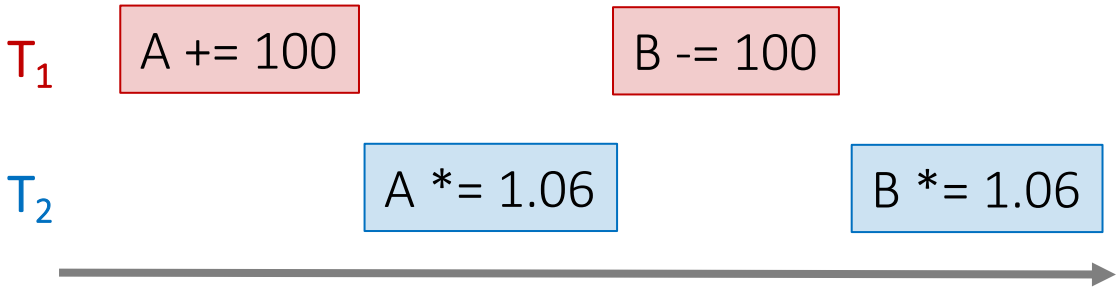
Scheduling Definitions

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

- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial schedule.

- Schedule 1 and Schedule 2 are **equivalent** if, ***for any database state***, the effect on DB of executing Schedule 1 **is identical to** the effect of Schedule 2

Serializable?



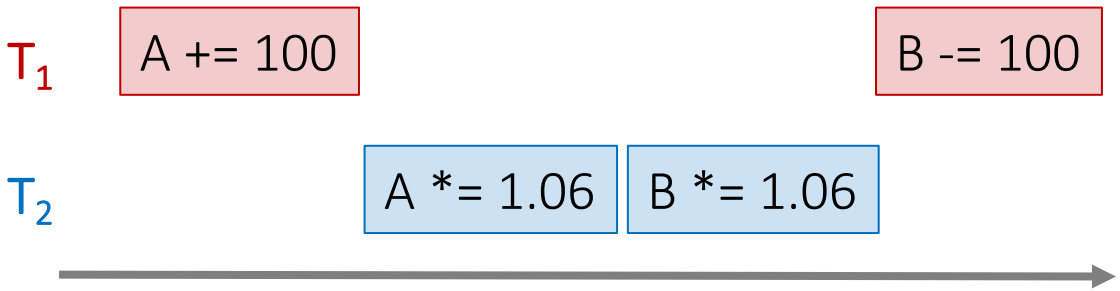
Serial schedules:

	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

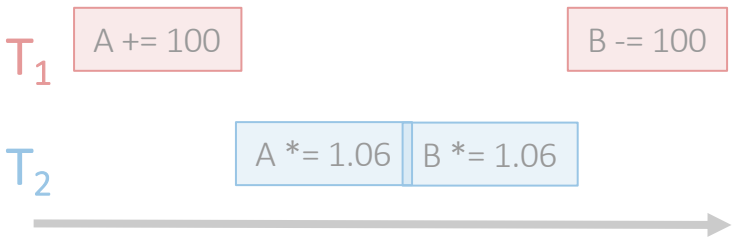
Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - **Anomaly types**
  - Conflict Serializability

What else can go wrong with interleaving?

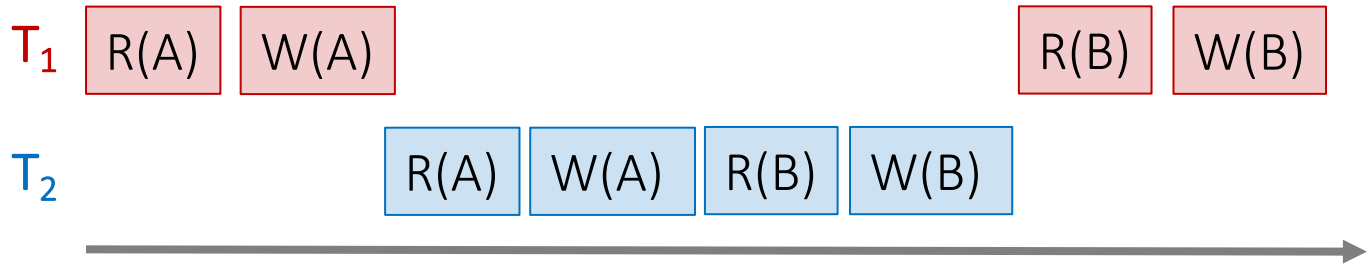
- Various anomalies which break isolation / serializability
    - Often referred to by name…
- Occur because of / with certain “conflicts” between interleaved TXNs

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)

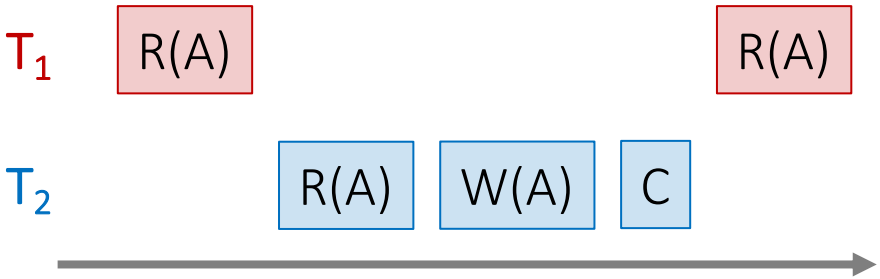
Why no "RR Conflict"?

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:



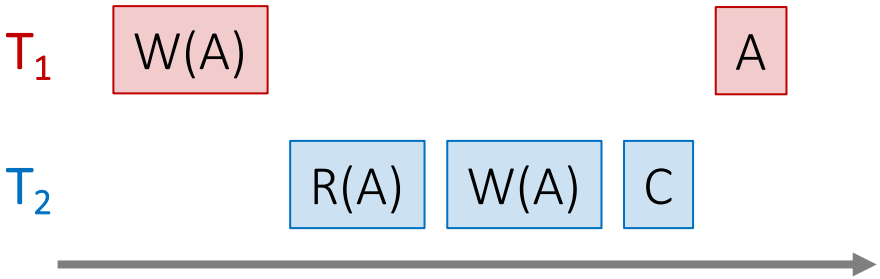
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

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

Classic Anomalies with Interleaved Execution

"Dirty read" / Reading uncommitted data:

Example:



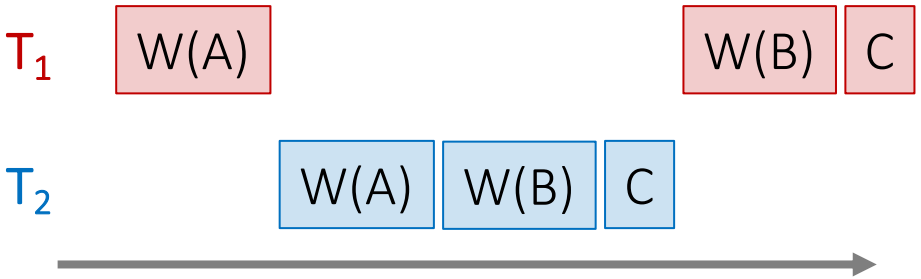
1. T_1 writes some data to A
2. T_2 reads from A, then writes back to A & commits
3. T_1 then aborts- *now T_2's result is based on an obsolete / inconsistent value*

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

Classic Anomalies with Interleaved Execution

"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!***

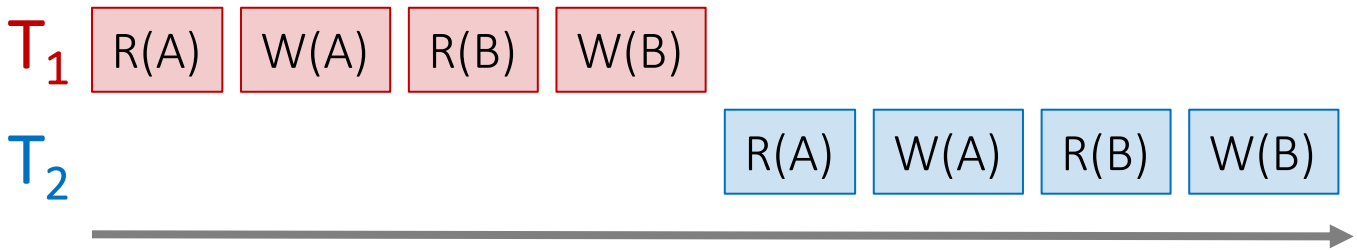
*Occurring because of a **WW conflict***

Outline

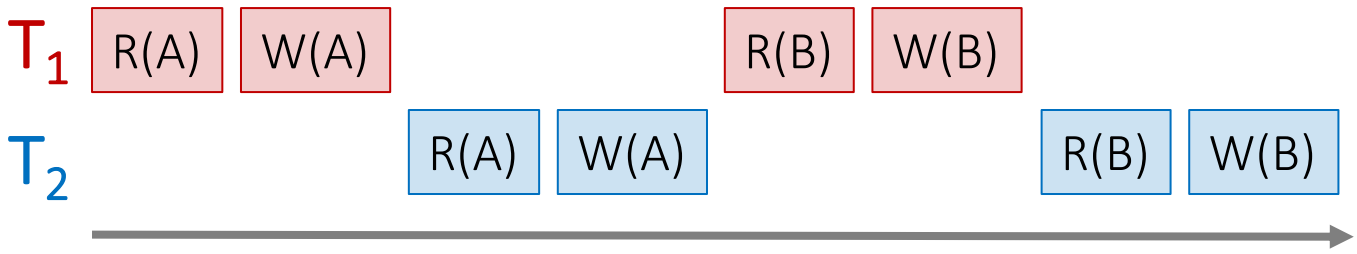
- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - Anomaly Types
  - **Conflict Serializability**

Schedules

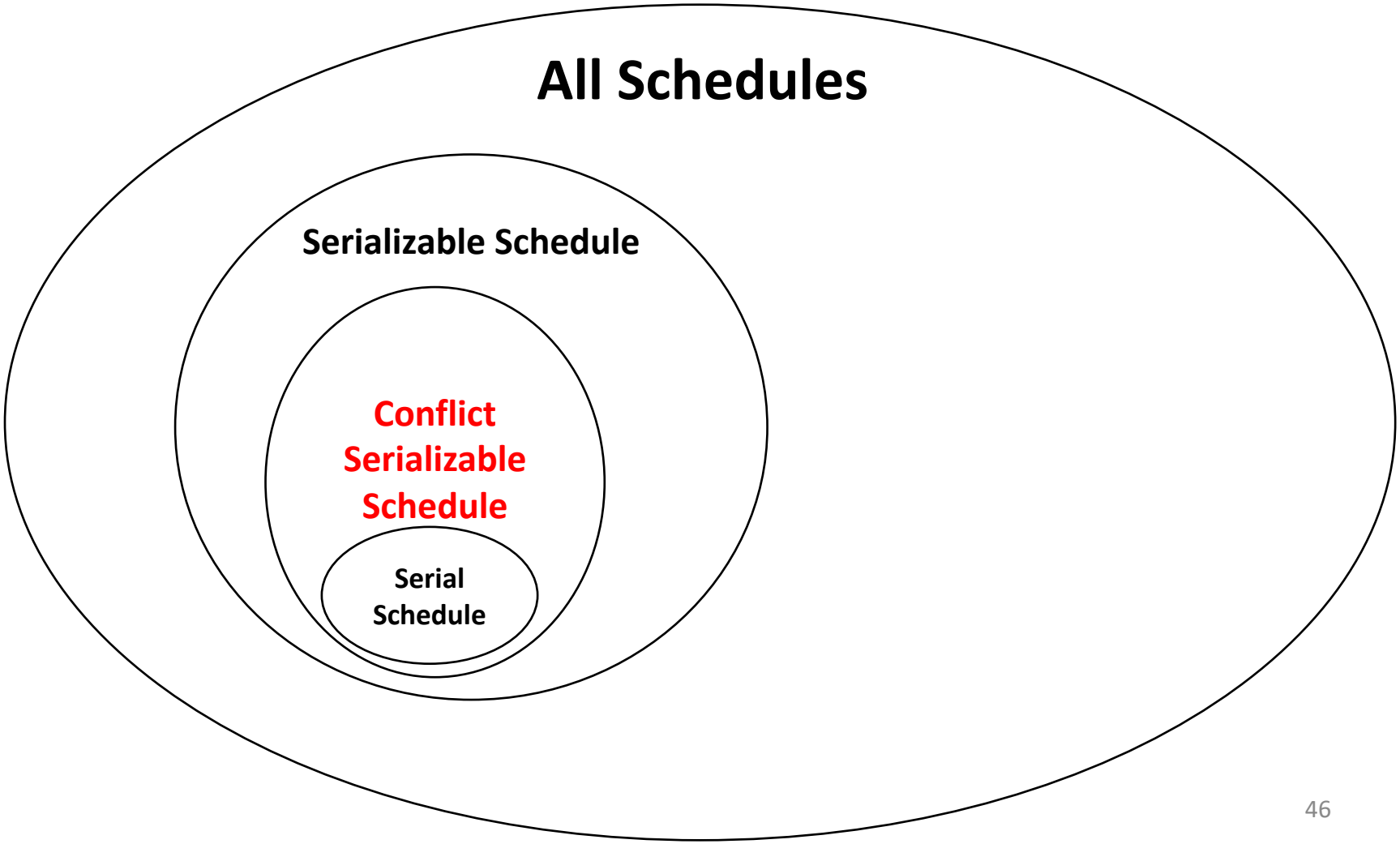
Serial Schedule:



Serializable Schedule:

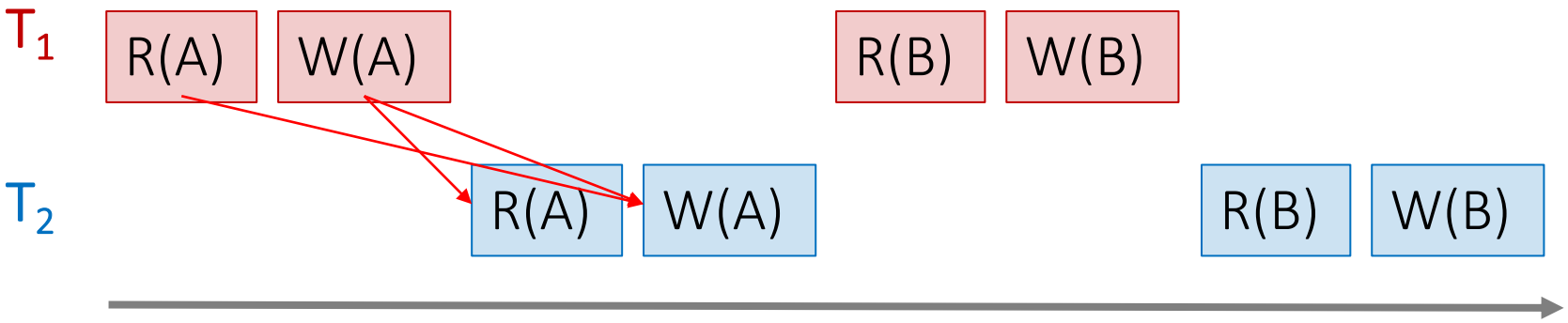


Conflict Serializable Schedule



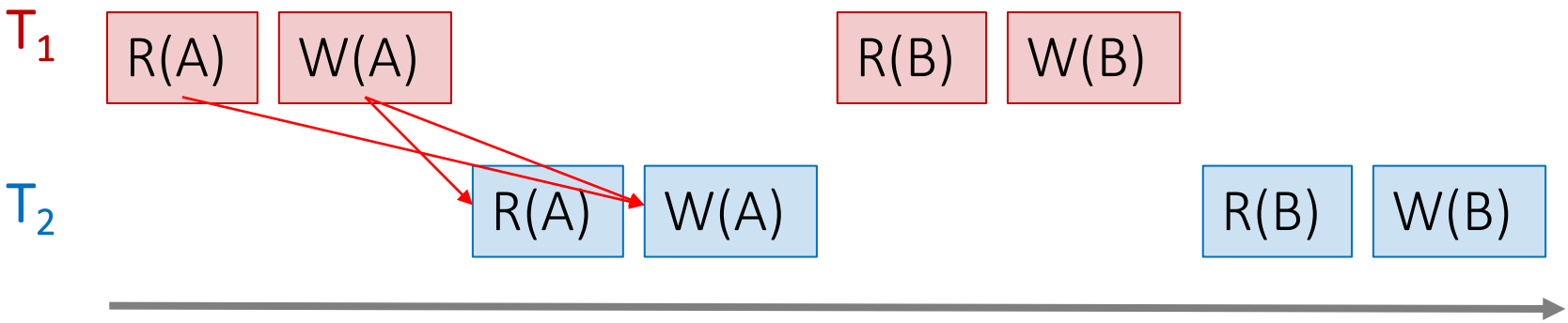
Conflicts

- Two actions **conflict** if all the conditions hold
  - i) they are part of different TXNs,
  - ii) they involve the same variable,
  - iii) at least one of them is a write



Exercise

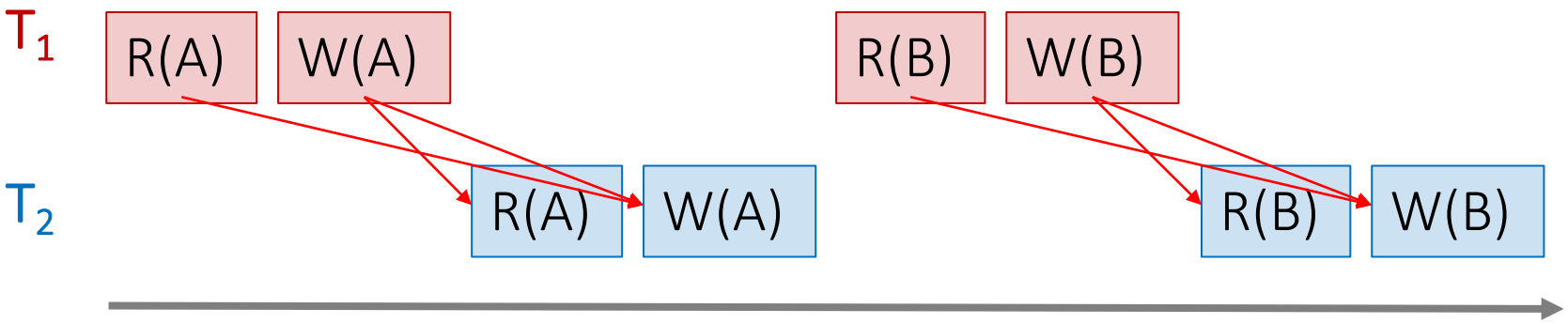
- Two actions **conflict** if all the conditions hold
  - i) they are part of different TXNs,
  - ii) they involve the same variable,
  - iii) at least one of them is a write



Find all the other conflicts

Exercise: Answer

- Two actions **conflict** if all the conditions hold
  - i) they are part of different TXNs,
  - ii) they involve the same variable,
  - iii) at least one of them is a write

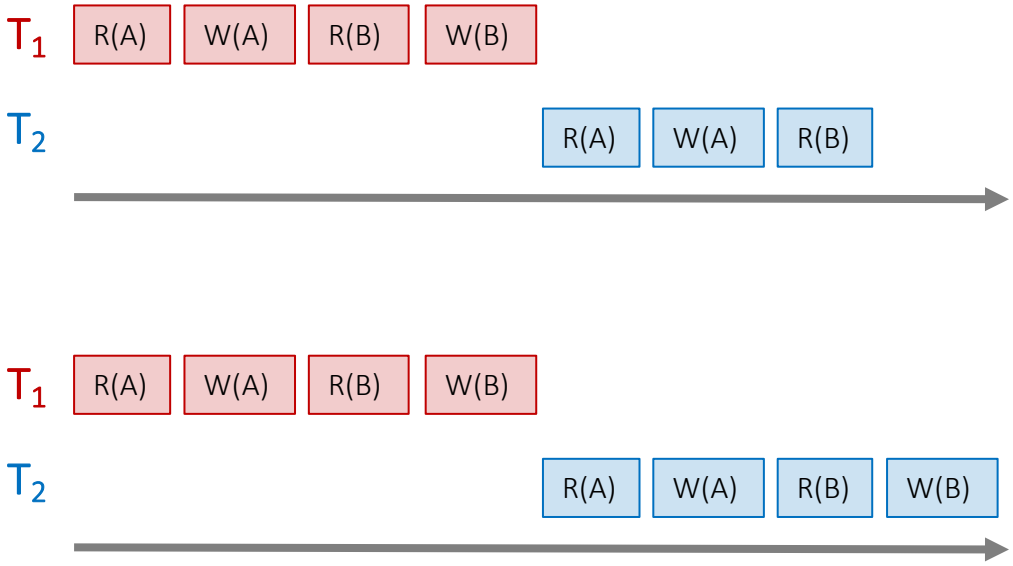


Conflict serializable

- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule

- 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*

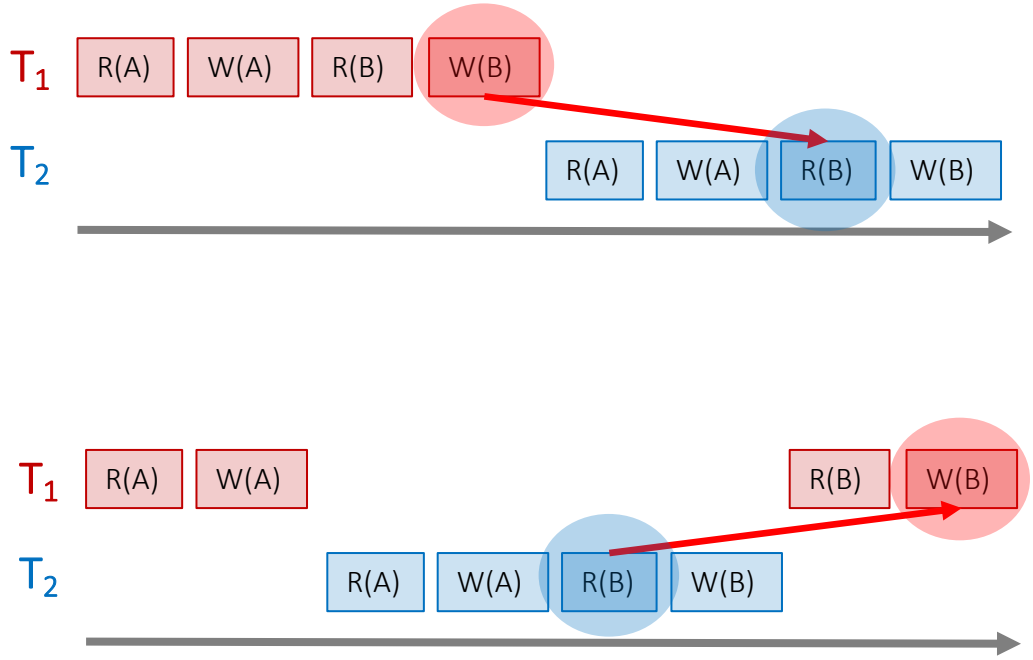
Are they conflict equivalent?



NO

- "They involve *the same actions of the same TXNs"* does not hold

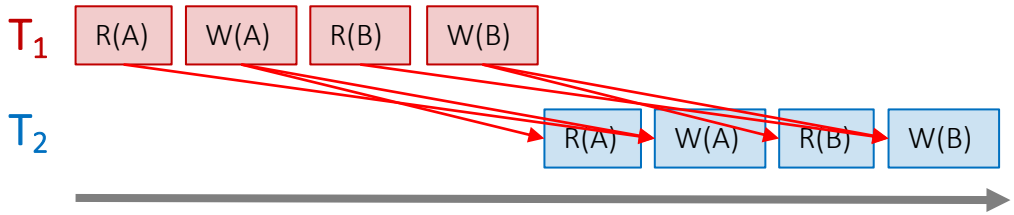
Are they conflict equivalent?



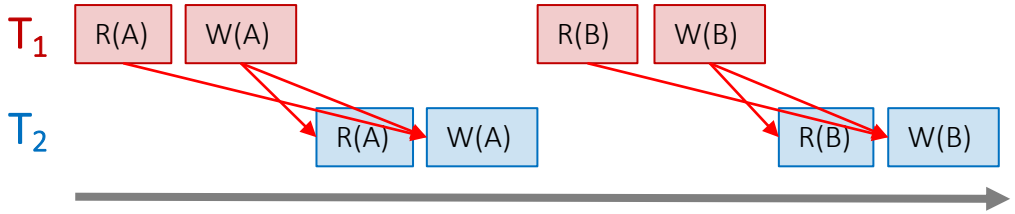
NO

- "Every *pair of conflicting actions* of two TXNs are *ordered in the same way*" does not hold

Are they conflict equivalent?



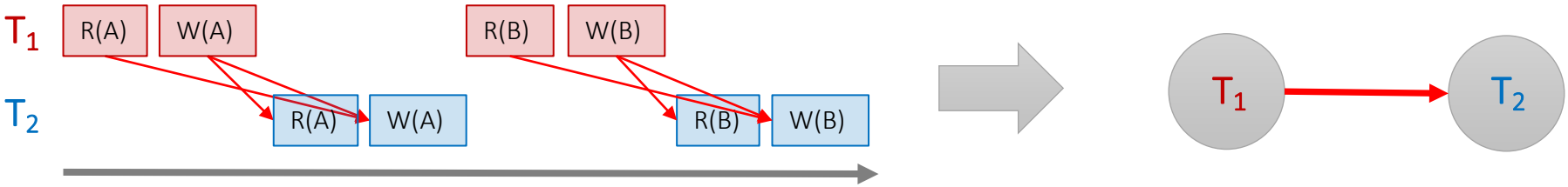
YES



- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

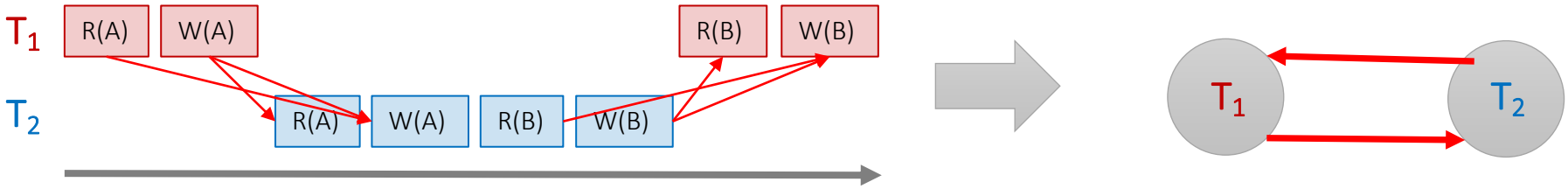
The Conflict Graph

- 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**



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

Is this schedule conflict serializable ?



1. Find all conflicts
2. Model the schedule as a conflict graph
3. Check whether the graph has a cycle
   - Yes → not conflict serializable
   - No → conflict serializable

NO

Isolation Levels

- Transactions in SQLite are serializable (https://www.sqlite.org/isolation.html)
- Isolation Levels in SQL Server (https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-2017)

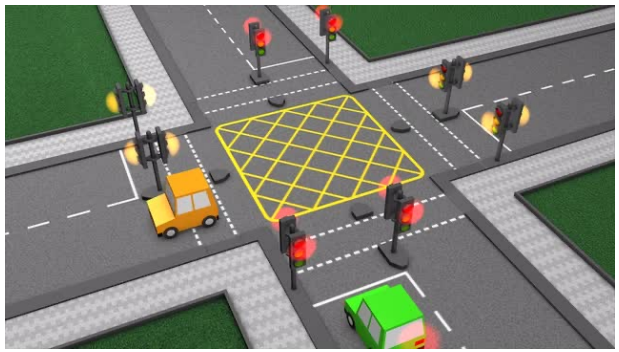
```
-- Syntax for SQL Server and Azure SQL Database

SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

ACID

Check out CMPT 454

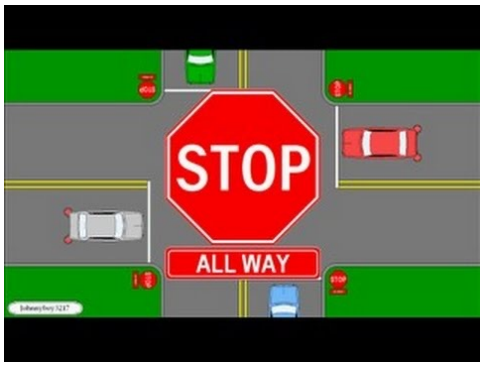
Concurrency Control Algorithms

- Locking



2-phase locking

- Timestamp Ordering



Multi-version concurrency control (MVCC)

Check out CMPT 454

Summary

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - Anomaly Types
  - Conflict Serializability

58

Acknowledge

- Some lecture slides were copied from or inspired by the following course materials
  - “W4111: Introduction to databases” by Eugene Wu at Columbia University
  - “CSE344: Introduction to Data Management” by Dan Suciu at University of Washington
  - “CMPT354: Database System I” by John Edgar at Simon Fraser University
  - “CS186: Introduction to Database Systems” by Joe Hellerstein at UC Berkeley
  - “CS145: Introduction to Databases” by Peter Bailis at Stanford
  - “CS 348: Introduction to Database Management” by Grant Weddell at University of Waterloo