COMP 421: Files & Databases

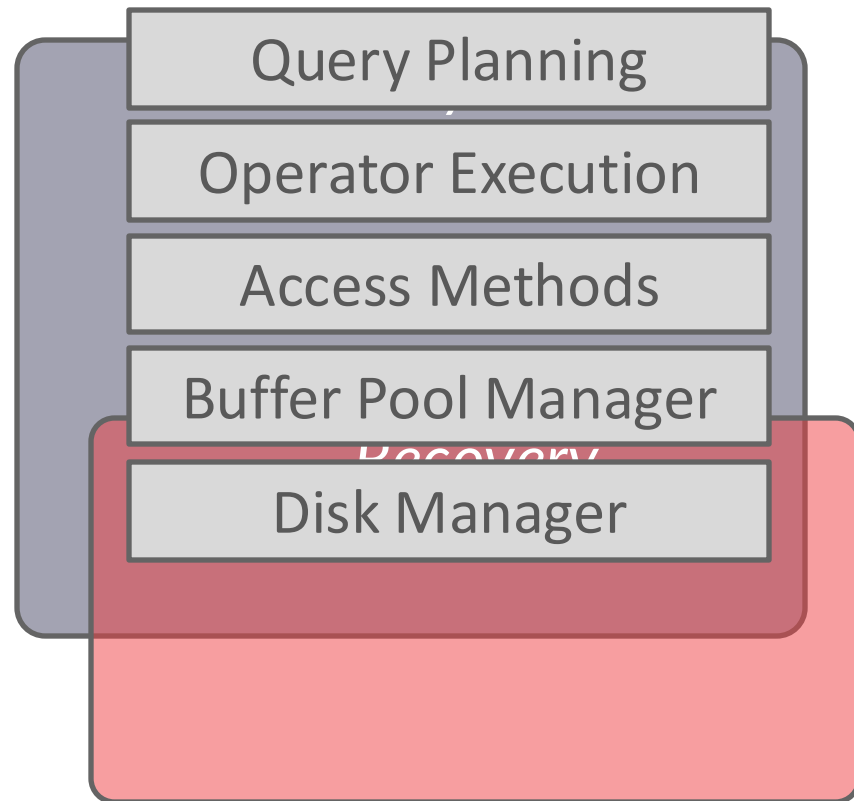Lecture 16: Concurrency Control

Project 3 released on Wednesday

Due to recent adjustments, working on schedule for last third of the class, stay tuned.

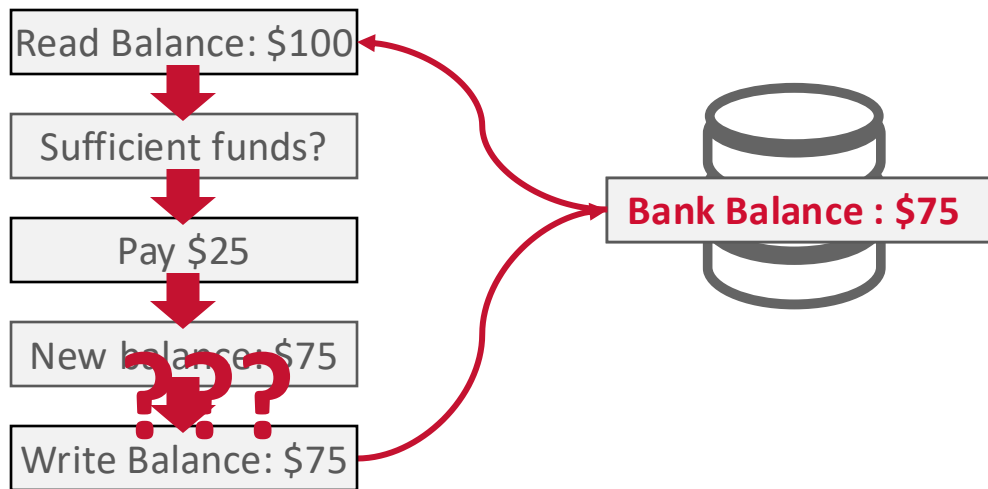We'll handle Project 2 on Wednesday once all late days are over

Course Status

A DBMS's concurrency control and recovery components permeate throughout the design of its entire architecture.

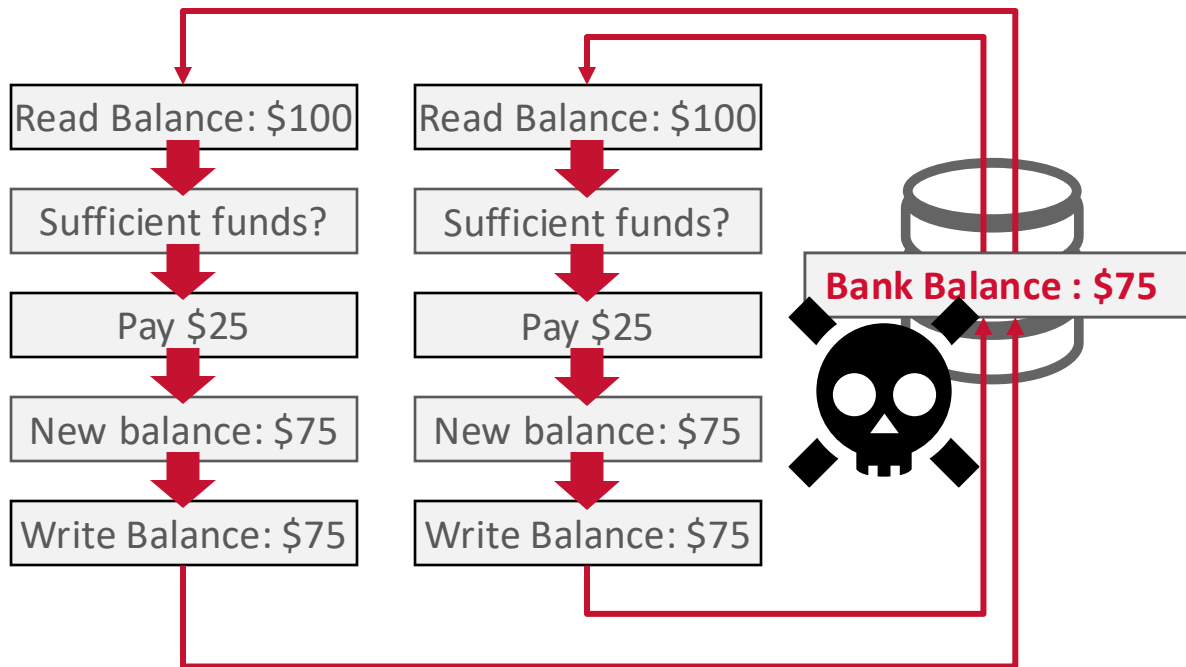| Query Planning |
|---|
| Operator Execution |
| Access Methods |
| Buffer Pool Manager |
| Disk Manager |

Motivation Example #1

Application Logic

Read(A);

Check($A > \$25$);

Pay($\$25$);

Read Balance: $100

Sufficient funds?

Pay $25

New balance: $75

???

Write Balance: $75

Bank Balance : $75

Motivation Example #2

Application Logic

```
Read(A);
Check(A > $25);
Pay($25);
A = A - $25;
Write(A);
```

Read Balance: $100	Read Balance: $100
Sufficient funds?	Sufficient funds?
Pay $25	Pay $25
New balance: $75	New balance: $75
Write Balance: $75	Write Balance: $75

Bank Balance : $75

Execute each txn one-by-one (i.e., serial order) as they arrive at the DBMS.
→ One and only one txn can be running simultaneously in the DBMS.

Before a txn starts, copy the entire database to a new file and make all changes to that file.
→ If the txn completes successfully, overwrite the original file with the new one.
→ If the txn fails, just remove the dirty copy.

Problem Statement

A (potentially) better approach is to allow concurrent execution of independent transactions.

Why do we want that?
→ Better utilization/throughput
→ Increased response times to users.

But we also would like:
→ Correctness
→ Fairness

Problem Statement

Arbitrary interleaving of operations can lead to:
→ Temporary Inconsistency (ok, unavoidable)
→ Permanent Inconsistency (bad!)

The DBMS is only concerned about what data is read/written from/to the database.
→ Changes to the "outside world" are beyond the scope of the DBMS.

We need formal correctness criteria to determine whether an interleaving is valid.

Formal Definitions

Database: A <u>fixed</u> set of named data objects (e.g., A, B, C, …).
→ We do not need to define what these objects are now.
→ We will discuss how to handle inserts/deletes next week.

Transaction: A sequence of <u>read</u> and <u>write</u> operations (e.g., R(A), W(B), …)
→ DBMS's abstract view of a user program.
→ A new txn starts with the BEGIN command.
→ The txn stops with either COMMIT or ROLLBACK

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Correctness Criteria: ACID

Atomicity	All actions in txn happen, or none happen. *"All or nothing…"*
Consistency	If each txn is consistent and the DB starts consistent, then it ends up consistent. *"It looks correct to me…"*
Isolation	Execution of one txn is isolated from that of other txns. *"All by myself…"*
Durability	If a txn commits, its effects persist. *"I will survive…"*

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Today's Agenda

Atomicity

Consistency

Isolation

Durability

Atomicity Of Transactions

Two possible outcomes of executing a txn:
→ Commit after completing all its actions.
→ Abort (or be aborted by the DBMS) after executing some actions.

DBMS guarantees that txns are **atomic**.
→ From user's point of view: txn always either executes all its actions or executes no actions at all.

Mechanisms For Ensuring Atomicity

Approach #1: Logging
→ DBMS logs all actions so that it can undo the actions of aborted transactions.
→ Maintain undo records both in memory and on disk.
→ Think of this like the black box in airplanes…

Logging is used by almost every DBMS.
→ Audit Trail
→ Efficiency Reasons

Don't Do This!

Approach #2: Shadow Paging
→ DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.
→ Originally from IBM System R.

Few systems do this:
→ CouchDB
→ Tokyo Cabinet
→ LMDB (OpenLDAP)

The database accurately models the real world.

→ SQL has methods to specify integrity constraints (e.g., key definitions, CHECK and ADD CONSTRAINT) and the DBMS will enforce them.

→ Application must define these constraints.

→ DBMS ensures that all ICs are true before and after the transaction ends.

Lecture #23

A note on **Eventual Consistency**.

→ A committed transaction may see inconsistent results (e.g., may not see the updates of an older committed txn).

→ Difficult for developers to reason about such semantics.

→ The trend is to move away from such models.

UNC DEPARTMENT OF COMPUTER SCIENCE

Users submit txns, and each txn executes as if it were running by itself.
→ Easier programming model to reason about.

But the DBMS achieves concurrency by interleaving the actions (reads/writes of DB objects) of txns.

We need a way to interleave txns but still make it appear as if they ran **one-at-a-time**.

Mechanisms For Ensuring Isolation

A **concurrency control** protocol is how the DBMS decides the proper interleaving of operations from multiple transactions.

Two categories of protocols:
→ **Pessimistic:** Don't let problems arise in the first place.
→ **Optimistic:** Assume conflicts are rare; deal with them after they happen.

Example

18

Assume at first A and B each have $1000.

T$_1$ transfers $100 from A's account to B's

T$_2$ credits both accounts with 6% interest.

T$_1$

```
BEGIN
A=A-100
B=B+100
COMMIT
```

T$_2$

```
BEGIN
A=A*1.06
B=B*1.06
COMMIT
```

Example

Assume at first A and B each have $1000.

What are the possible outcomes of running T_1 and T_2?

T_1

```
BEGIN
A=A-100
B=B+100
COMMIT
```

T_2

```
BEGIN
A=A*1.06
B=B*1.06
COMMIT
```

Example

Assume at first A and B each have $1000.

What are the possible outcomes of running T_1 and T_2?

Many! But A+B should be:
→ $2000*1.06=$2120

There is no guarantee that T_1 will execute before T_2 or vice-versa, if both are submitted together.

But the net effect must be equivalent to these two transactions running **<u>serially</u>** in some order.

Example

Legal outcomes:
→ A=954, B=1166 → A+B=$2120
→ A=960, B=1160 → A+B=$2120

The outcome depends on whether T_1 executes before T_2 or vice versa.

Serial Execution Example

TIME

Schedule

T_1	T_2
BEGIN A=A-100 B=B+100 COMMIT	
	BEGIN A=A*1.06 B=B*1.06 COMMIT

A=954, B=1166

=

Schedule

T_1	T_2
	BEGIN A=A*1.06 B=B*1.06 COMMIT
BEGIN A=A-100 B=B+100 COMMIT	

A=960, B=1160

A+B=$2120
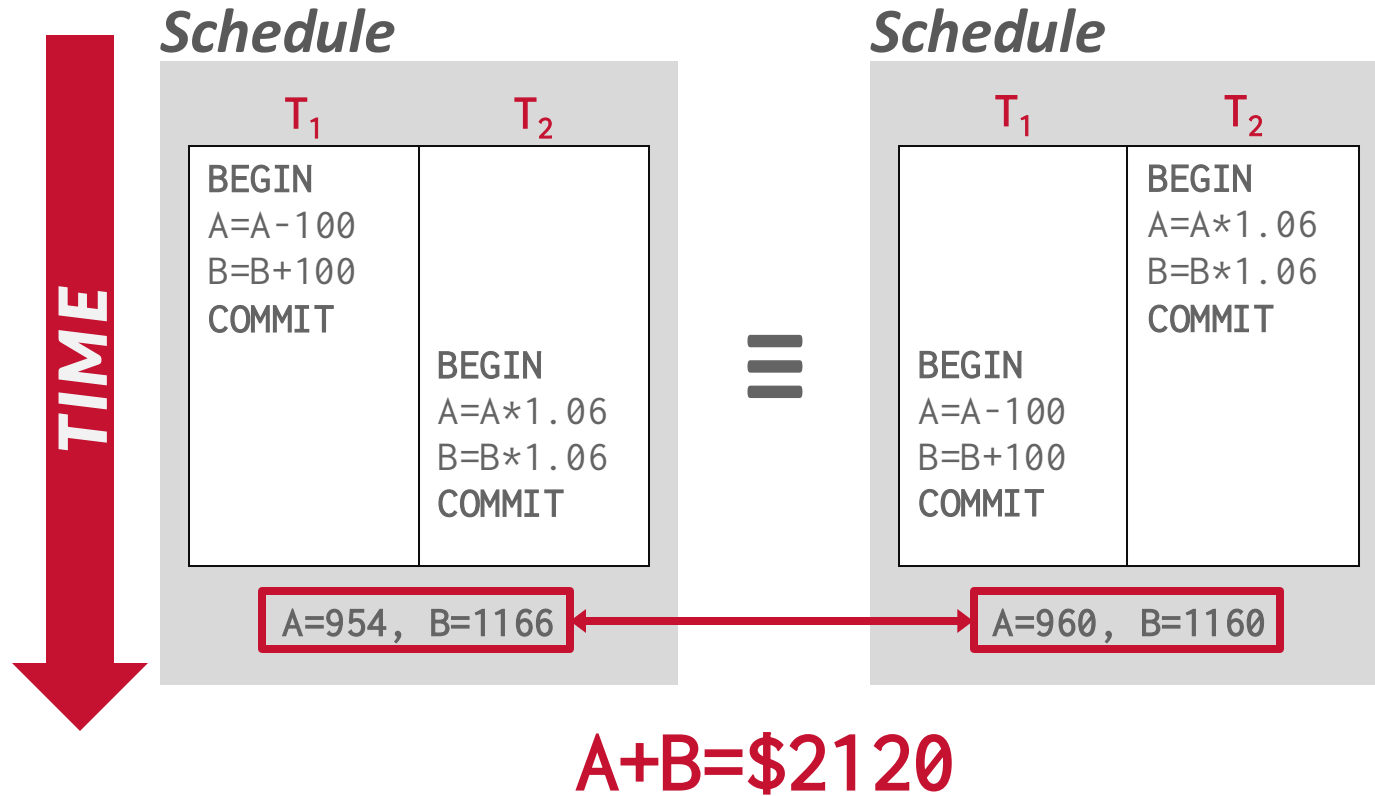
Interleaving Transactions

We interleave txns to maximize concurrency.
$\rightarrow$ Slow disk/network I/O.
$\rightarrow$ Multi-core CPUs.

When one txn stalls because of a resource (e.g., page fault), another txn can continue executing and make forward progress.

Interleaving Example (Good)

TIME

Schedule

T_1	T_2
BEGIN	
A=A-100	
	BEGIN
	A=A*1.06
B=B+100	
COMMIT	
	B=B*1.06
	COMMIT

A=954, B=1166

=

Schedule

T_1	T_2
BEGIN	
A=A-100	
B=B+100	
COMMIT	
	BEGIN
	A=A*1.06
	B=B*1.06
	COMMIT

A=960, B=1160

A+B=$2120

Interleaving Example (Bad)

Schedule

TIME

T_1	T_2
BEGIN A=A-100	
	BEGIN A=A*1.06 B=B*1.06 COMMIT
B=B+100 COMMIT	

A=954, B=1160

$$\ne$$

A=954, B=1166
or
A=960, B=1160

Off by \$6!

A+B=\$2114

Interleaving Example (Bad)

Schedule

TIME

	T₁	T₂
	BEGIN A=A-100	
		BEGIN A=A*1.06 B=B*1.06 COMMIT
	B=B+100 COMMIT	

A=954, B=1160

A+B=$2114

DBMS View

T₁	T₂
BEGIN R(A) W(A)	
	BEGIN R(A) W(A) R(B) W(B) COMMIT
R(B) W(B) COMMIT	

How do we judge whether a schedule is correct?

If the schedule is **equivalent** to some **serial execution**.

Formal Properties Of Schedules

Serial Schedule
→ A schedule that does not interleave the actions of different transactions.

Equivalent Schedules
→ For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.

Serializable Schedule

→ A schedule that is equivalent to some serial execution of the transactions.

→ If each transaction preserves consistency, every serializable schedule preserves consistency.

Serializability is a less intuitive notion of correctness compared to txn initiation time or commit order, but it provides the DBMS with more flexibility in scheduling operations.

→ More flexibility means better parallelism.

Conflicting Operations

We need a formal notion of equivalence that can be implemented efficiently based on the notion of "conflicting" operations.

Two operations **conflict** if:
→ They are by different transactions,
→ They are on the same object and one of them is a write.

Interleaved Execution Anomalies
→ Unrepeatable Read (**Read-Write**)
→ Dirty Read (**Write-Read**)
→ Lost Update (**Write-Write**)
→ **Phantom Reads (Scan-Write)** *Lecture #17*
→ **Write-Skew (Read-Write)** *Lecture #19*

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Read-Write Conflicts

Unrepeatable Read: Txn gets different values when reading the same object multiple times.
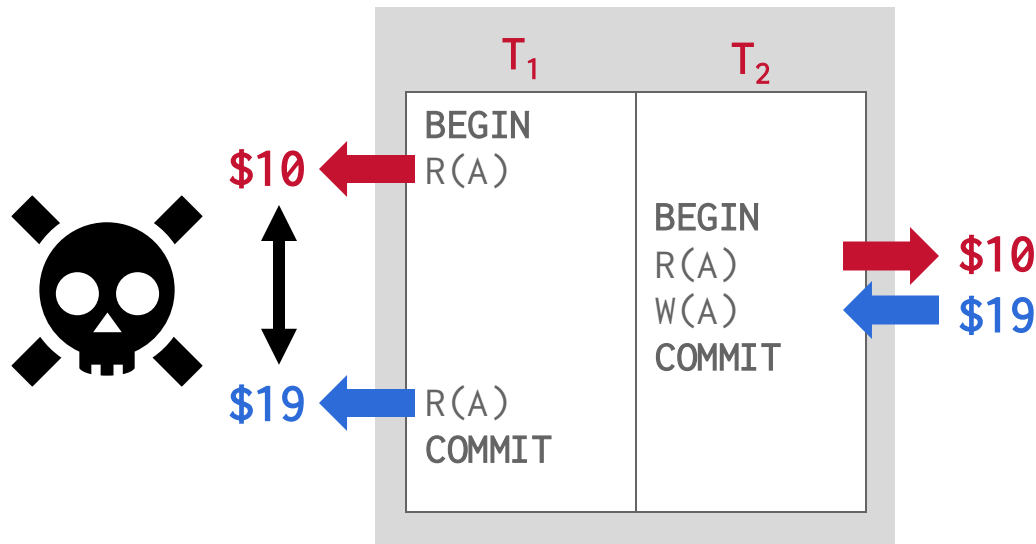
Write-Read Conflicts

Dirty Read: One txn reads data written by another txn that has not committed yet.
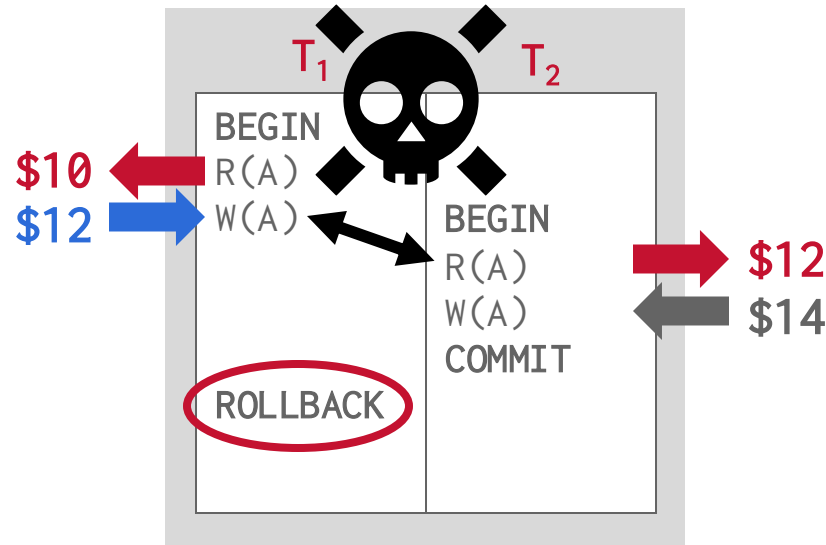
Write-Write Conflicts

Lost Update: One txn overwrites uncommitted data from another uncommitted txn.
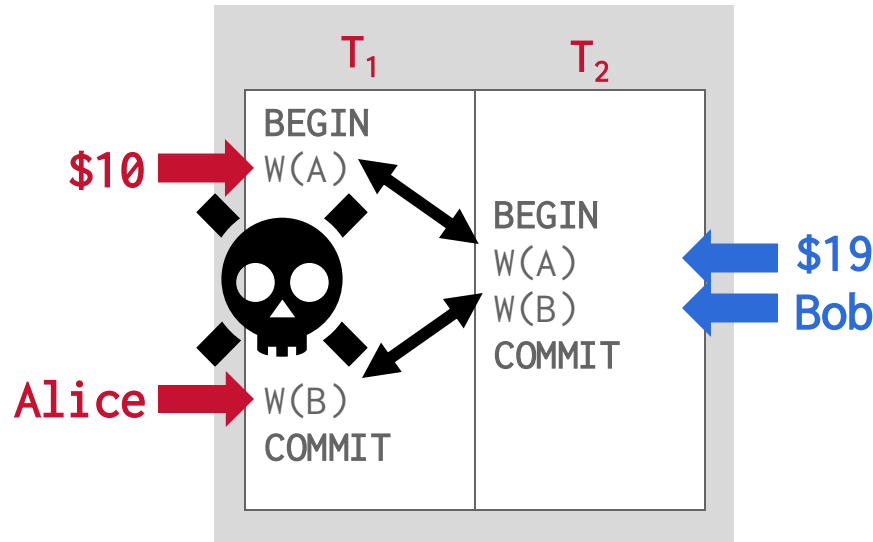
Formal Properties Of Schedules

Given these conflicts, we now can understand what it means for a schedule to be serializable.
→ This is to check whether schedules are correct.
→ This is <u>not</u> how to generate a correct schedule.

There are different levels of serializability:
→ **Conflict Serializability**
→ **View Serializability**

Most DBMSs try to support this.

No DBMS can do this.

Conflict Serializable Schedules

Two schedules are **conflict equivalent** iff:
→ They involve the same actions of the same transactions.
→ Every pair of conflicting actions is ordered the same way in time.

Schedule S is **conflict serializable** if:
→ S is conflict equivalent to some serial schedule.
→ Intuition: You can transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

Dependency Graphs

One node per txn.

Edge from T_i to T_j if:

→ An operation O_i of T_i conflicts with an operation O_j of T_j and

→ O_i appears earlier in the schedule than O_j.

Also known as a **precedence graph**. A schedule is conflict serializable iff its dependency graph is acyclic.

Dependency Graph

Example #1

Schedule

TIME

T₁	T₂
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph

A

T₁ → T₂

B
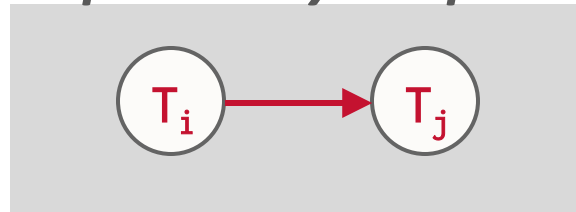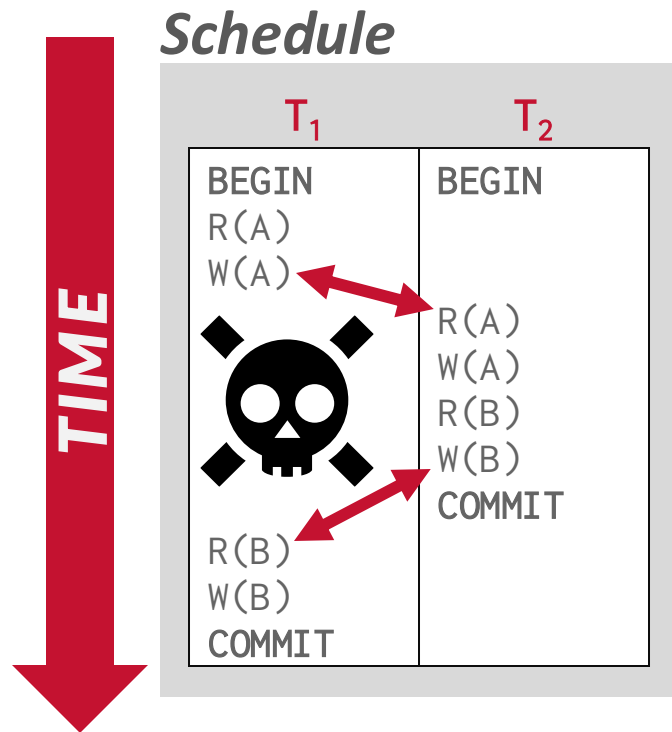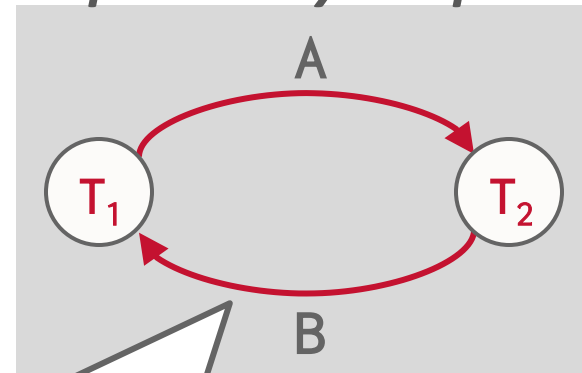
The cycle in the graph reveals the problem.
The output of T₁ depends on T₂, and vice-versa.

Example #2 – Three Transactions

Schedule
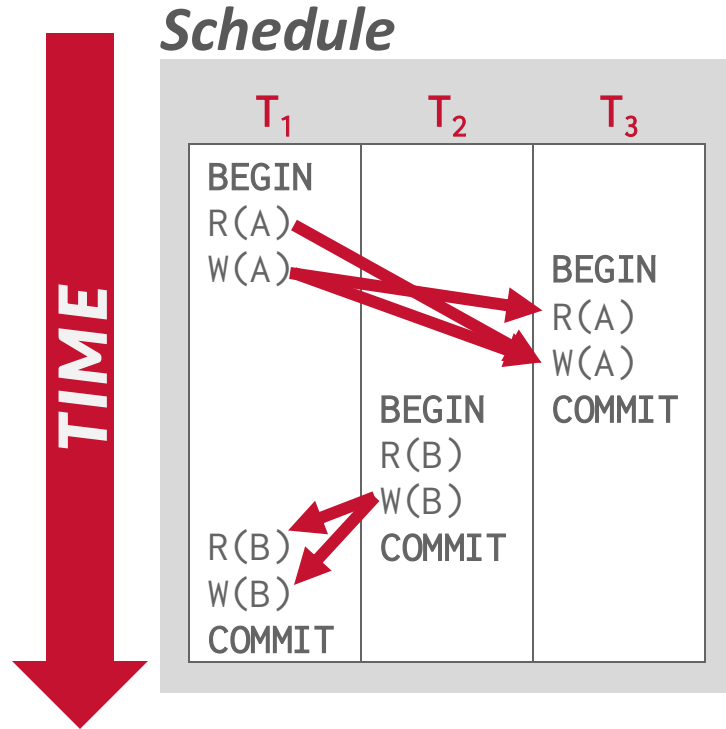
Dependency Graph

Is this equivalent to a serial execution?

Yes (T_2, T_1, T_3)

→ Notice that T_3 should go after T_2, although it starts before it!

Example #3 – Inconsistent Analysis

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	if(A≥0): cnt++
	R(B)
	if(B≥0): cnt++
	ECHO cnt
R(B)	COMMIT
B = B+10	
W(B)	
COMMIT	

Dependency Graph

A

T_1 T_2

B
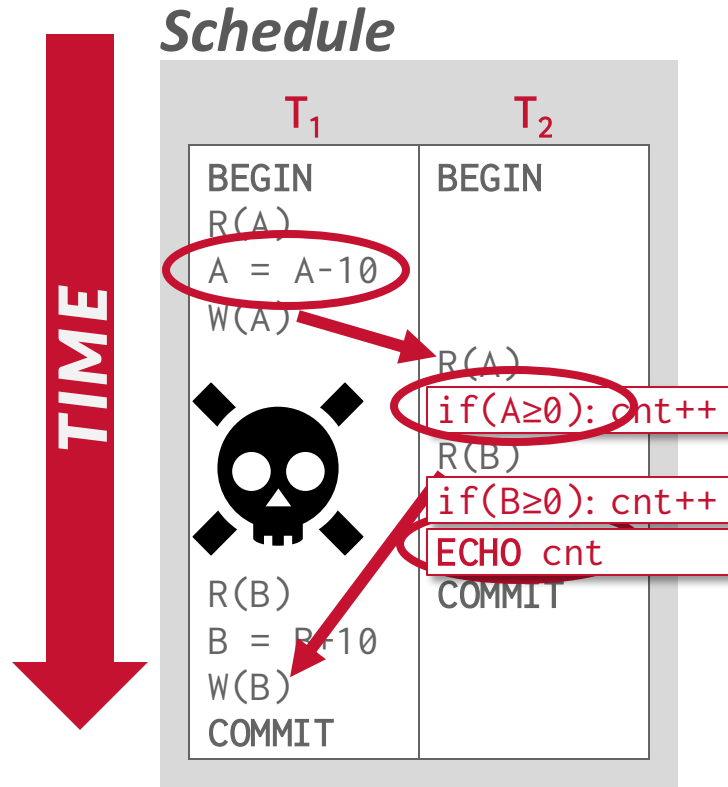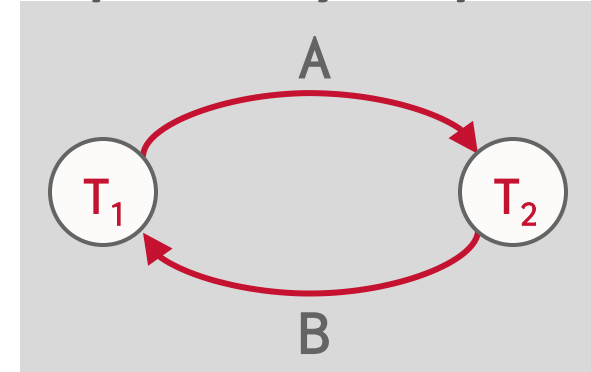
Is it possible to modify <u>only</u> the application logic so that schedule produces a "correct" result but is still not conflict serializable?

View Serializability

Alternative (broader) notion of serializability.

Schedules S_1 and S_2 are <u>view equivalent</u> if:

→ If T_1 reads initial value of A in S_1, then T_1 also reads initial value of A in S_2.

→ If T_1 reads value of A written by T_2 in S_1, then T_1 also reads value of A written by T_2 in S_2.

→ If T_1 writes final value of A in S_1, then T_1 also writes final value of A in S_2.

View Serializability
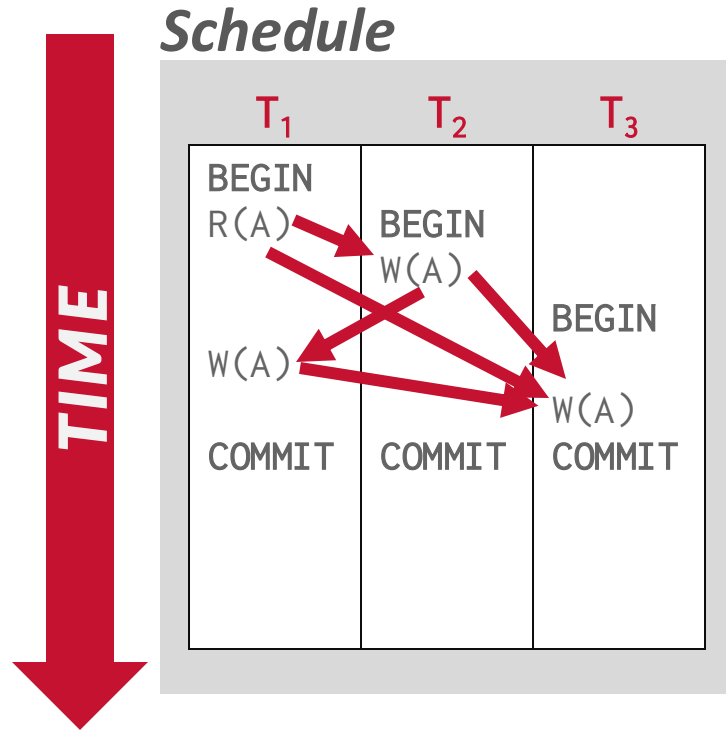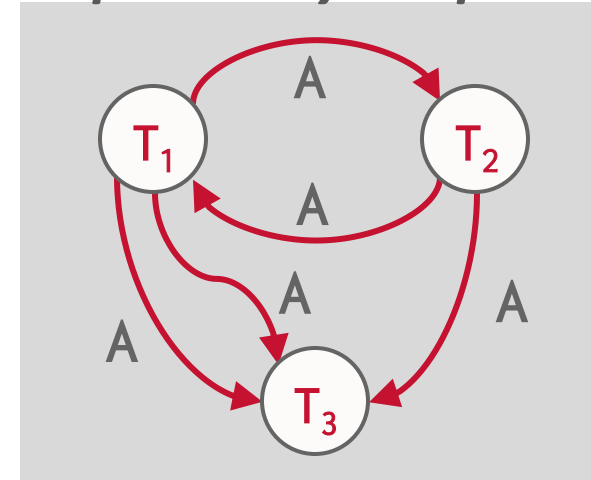
View Serializability
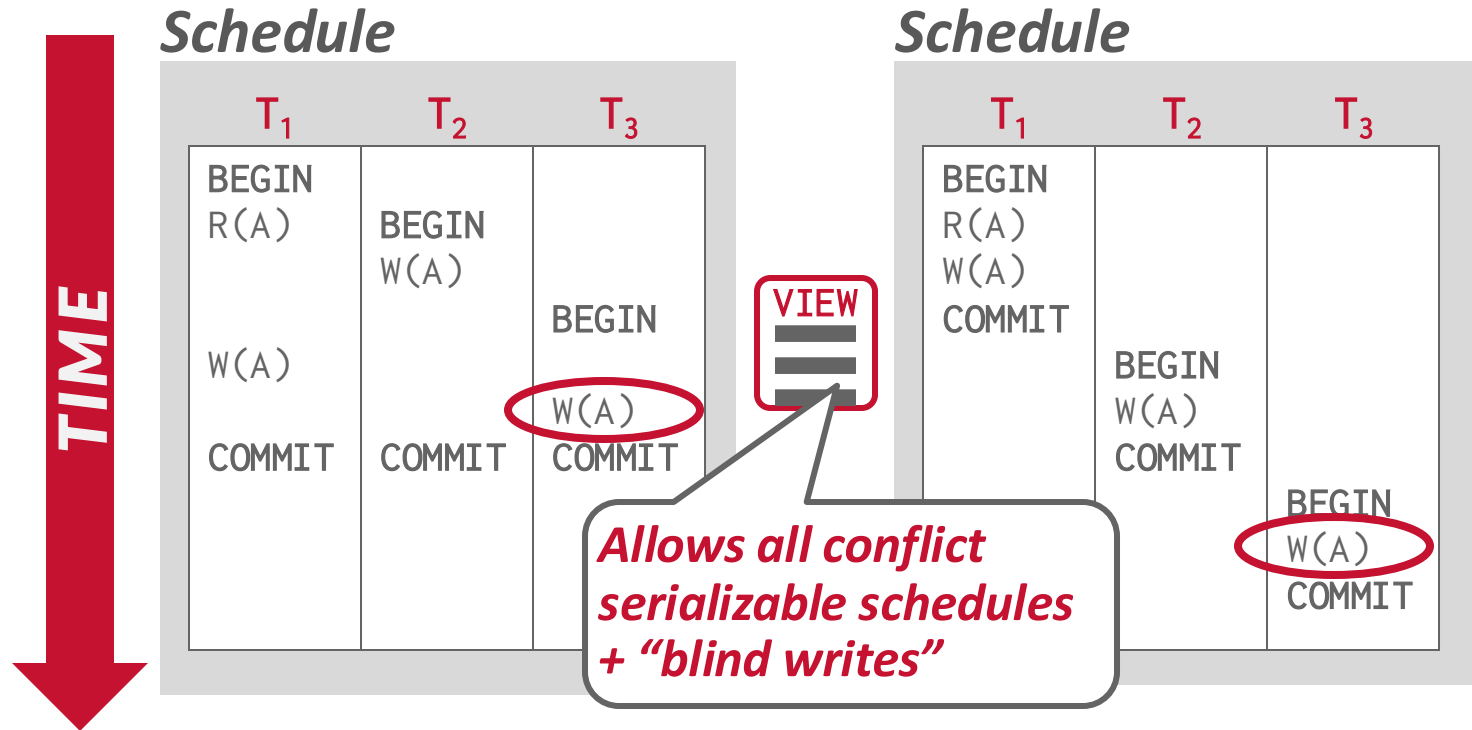
Serializability

View Serializability allows for (slightly) more schedules than **Conflict Serializability** does.
→ But it is difficult to enforce efficiently.

Neither definition allows all schedules that you would consider "serializable."
→ This is because they don't understand the meanings of the operations or the data (recall example #3)
→ In practice, Conflict Serializability is what systems support because it can be enforced efficiently.

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Universe of Schedules

All Schedules

View Serializable

Conflict Serializable

Serial

Transaction Durability

All the changes of committed transactions should be persistent.
→ No torn updates.
→ No changes from failed transactions.

The DBMS can use either logging or shadow paging to ensure that all changes are durable.

Correctness Criteria: ACID

Redo/Undo Mechanism

Atomicity — All actions in txn happen, or none happen.
"All or nothing…"

Integrity Constraints

Consistency — If each txn is consistent and the DB starts consistent, then it ends up consistent.
"It looks correct to me…"

Concurrency Control

Isolation — Execution of one txn is isolated from that of other txns.
"All by myself…"

Redo/Undo Mechanism

Durability — If a txn commits, its effects persist.
"I will survive…"

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Conclusio

Concurrency control and recovery are among the most important functions provided by a DBMS.

Concurrency control is automatic
→ System automatically inserts lock/unlock requests and schedules actions of

→

Spanner: Google's Globally-Distributed Database

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman,
Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh,
Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura,
David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak,
Christopher Taylor, Ruth Wang, Dale Woodford

Google, Inc.

Abstract

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This paper describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

ability problems that it brings [9, 10, 19]. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos

Next Class

Two-Phase Locking

Isolation Levels