Intro to Database Systems (15-445/645)

15 Concurrency Control Theory

Carnegie Mellon University

SPRING 2023

Charlie Garrod

ADMINISTRIVIA

Project 2 still ongoing
→ Due Wednesday, March 22nd
→ Special office hours today and tomorrow 5 – 7 p.m.

Project 3 released late this week

Final exam Monday, May 1st, 8:30 – 11:30 a.m.

LAST TIME: QUERY OPTIMIZATION

Heuristics / Rules

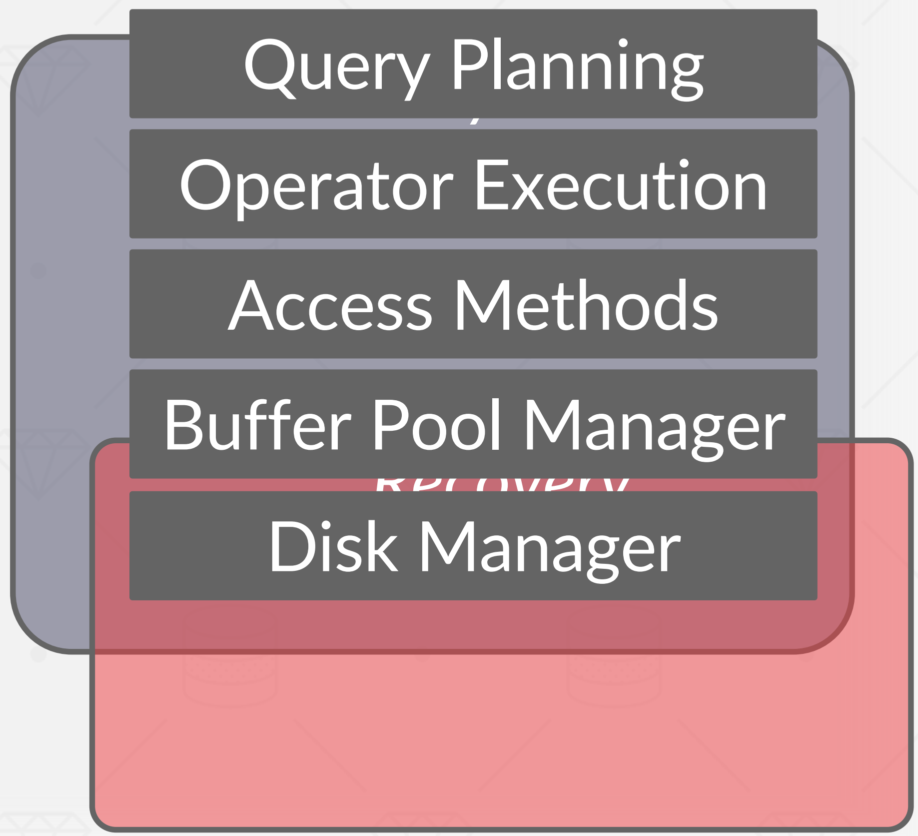
→ Rewrite the query to remove stupid / inefficient things.
→ These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

→ Use a model to estimate the cost of executing a plan.
→ Enumerate multiple equivalent plans for a query and pick the one with the lowest cost.

COURSE STATUS

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



MOTIVATION

We both change the same record in a table at the same time.
How to avoid race conditions?

Lost Updates
Concurrency Control

You transfer $100 between bank accounts but there is a power failure.
What is the correct database state?

Durability
Recovery

CONCURRENCY CONTROL & RECOVERY

Valuable properties of DBMSs.

Based on concept of transactions with **ACID** properties.

Let's talk about transactions…

TRANSACTIONS

A **transaction** (txn) is the execution of a sequence of one or more operations (e.g., SQL queries) on a database to perform some higher-level function.

It is the basic unit of change in a DBMS.

TRANSACTION EXAMPLE

Move $100 from Andy's bank account to his bookie's account.

Transaction:
- → Check whether Andy has $100.
- → Deduct $100 from his account.
- → Add $100 to his bookie's account.

STRAWMAN SYSTEM

Execute each txn one-by-one (i.e., serial order) as they arrive at the DBMS.

→ One and only one txn can be running at the same time 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 internal inconsistency (ok, unavoidable)
→ Permanent inconsistency (bad!)

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

Caveat: We're only concerned with what's happening inside the database: reads, writes, etc.

FORMAL DEFINITIONS

Database: A fixed 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 later.

Transaction: A sequence of read and write operations (`R(A)`, `W(B)`, …)
→ DBMS's abstract view of a user program

TRANSACTIONS IN SQL

A new txn starts with the **BEGIN** command.

The txn stops with either **COMMIT** or **ABORT**:
- → If commit, the DBMS either saves all the txn's changes **or** aborts it.
- → If abort, all changes are undone so that it's like as if the txn never executed at all.

Abort can be either self-inflicted or caused by the DBMS.

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	Each txn sees the DB as if it's running alone in the DB. *"All by myself…"*
Durability	If a txn commits, its effects persist. *"I will survive…"*

TODAY'S AGENDA

Atomicity
Isolation
Durability
Consistency

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.

ATOMICITY OF TRANSACTIONS

Scenario #1:

→ We take $100 out of Andy's account but then the DBMS aborts the txn before we transfer it.

Scenario #2:

→ We take $100 out of Andy's account but then there is a power failure before we transfer it.

What should be the correct state of Andy's account after both txns abort?

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

MECHANISMS FOR ENSURING ATOMICITY

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)

ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it was 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

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
```

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

EXAMPLE

Assume at first A and B each have $1000.

What are the possible outcomes of running T_1 and T_2?

→ More than one! 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 **serially** 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.

INTERLEAVING TRANSACTIONS

We interleave txns to maximize concurrency.
→ Slow disk/network I/O.
→ 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=954, B=1166

A+B=$2120

INTERLEAVING EXAMPLE (BAD)

TIME

Schedule

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

≢

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

A+B=$2114

The bank is missing $6!

INTERLEAVING EXAMPLE (BAD)

TIME

Schedule

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

A+B=$2114

DBMS View

T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

A schedule is correct if it 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.

FORMAL PROPERTIES OF SCHEDULES

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

Serializability can be enforced efficiently based on the notion of conflicting operations.

Two operations **conflict** if:

→ They are by different transactions,

→ They are on the same object and ≥1 of them is a write.

Interleaved Execution Anomalies

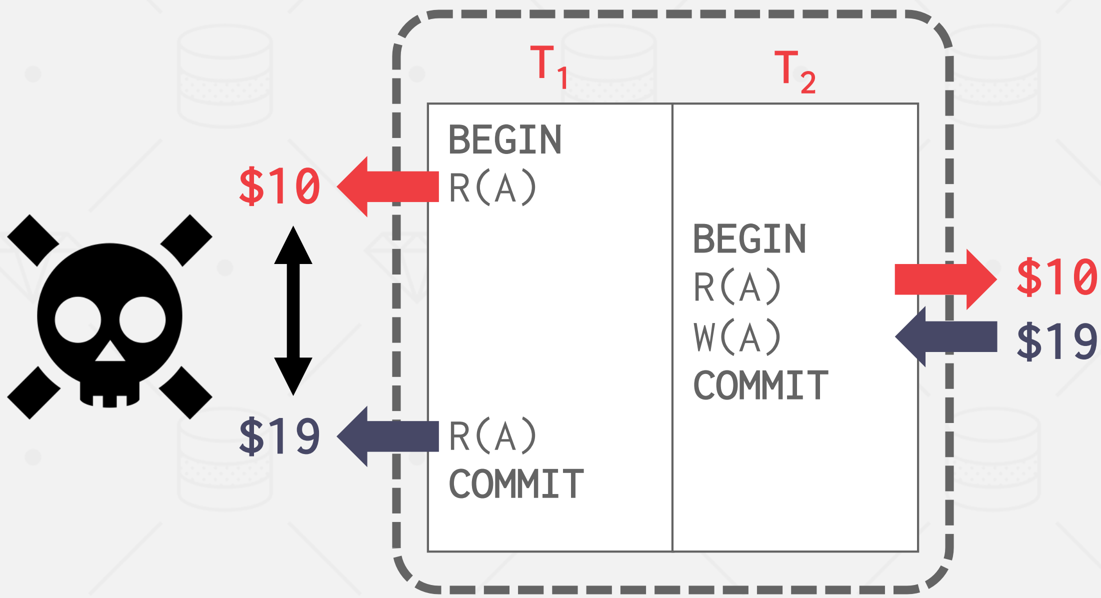
→ Read-Write Conflicts (**R-W**)

→ Write-Read Conflicts (**W-R**)

→ Write-Write Conflicts (**W-W**)

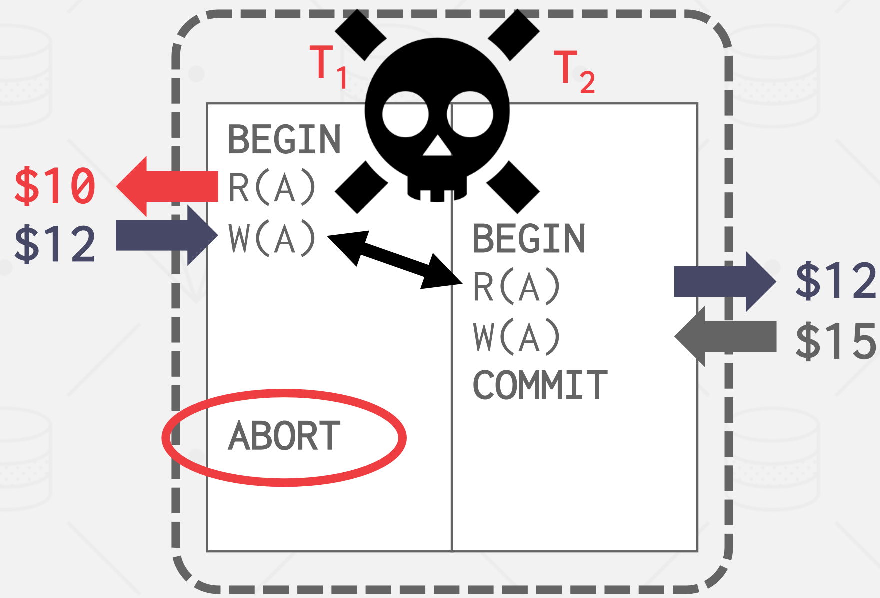
READ-WRITE CONFLICTS

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



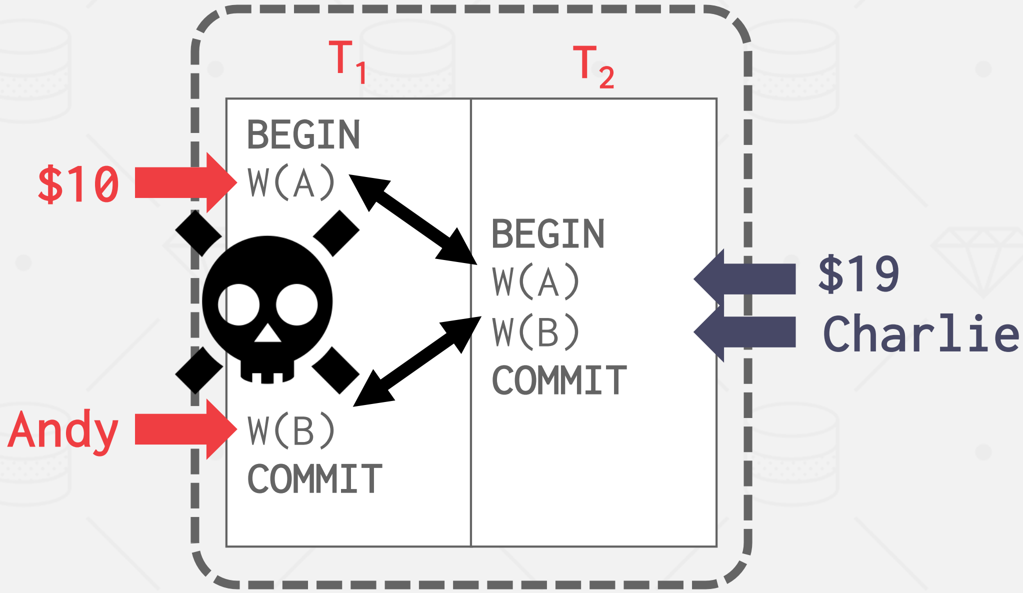
WRITE-READ CONFLICTS

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

We can use these conflicts to prove that a schedule of operations is serializable.

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

Most DBMSs support this (or something like this).

No DBMS does 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.

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.

CONFLICT SERIALIZABILITY INTUITION

TIME

Schedule

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	R(A)
R(B)	R(A)
R(B)	R(A)
W(B)	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

≡

Serial Schedule

T_1	T_2
BEGIN	
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

CONFLICT SERIALIZABILITY INTUITION

TIME

Schedule

T_1	T_2
BEGIN	BEGIN
R(A)	
	R(A)
	W(A)
W(A)	
COMMIT	COMMIT

$\neq$

Serial Schedule

T_1	T_2
BEGIN	
R(A)	
W(A)	
COMMIT	BEGIN
	R(A)
	W(A)
	COMMIT

DEPENDENCY GRAPHS

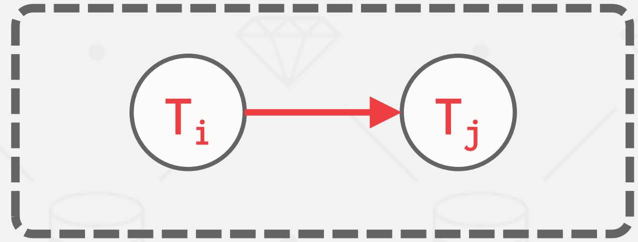
One node per txn.

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



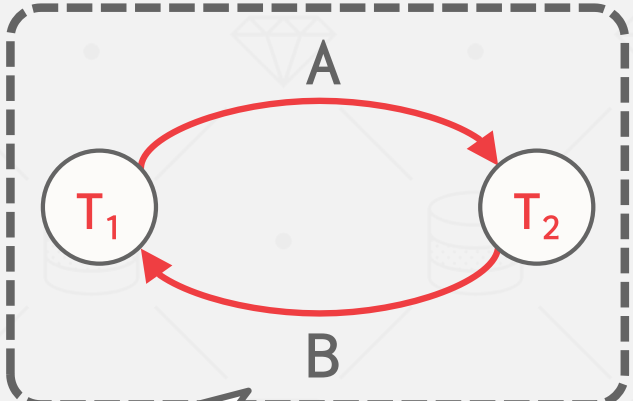
EXAMPLE #1

TIME

Schedule

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph



The cycle in the graph reveals that the output of T_1 depends on T_2, and vice-versa.

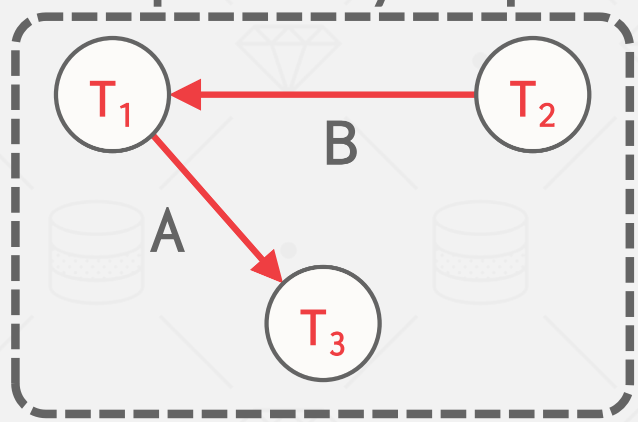
EXAMPLE #2 – THREE TRANSACTIONS

Schedule

TIME

T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		

Dependency Graph



Is this equivalent to a serial execution?

Yes: T_2, T_1, T_3

→ T_3 is after T_2 in the equivalent serial schedule, although it starts before it!

VIEW SERIALIZABILITY

Alternative (broader) notion of serializability.

Schedules S_1 and S_2 are view equivalent if:

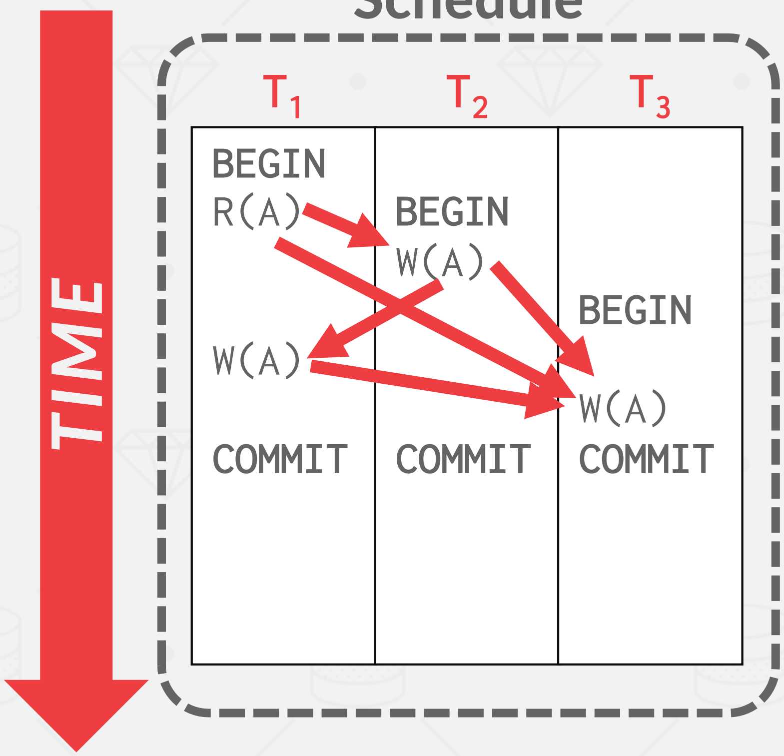
→ If T_1 reads initial value of A in S_1, then T_1 also reads initial value of A in S_2.

→ 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.

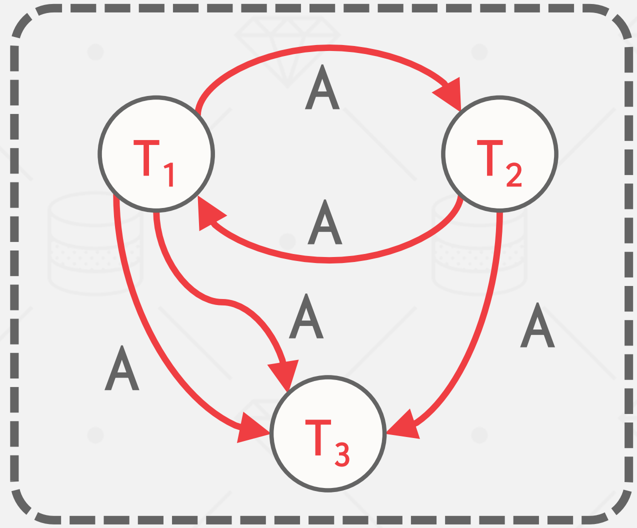
→ If T_1 writes final value of A in S_1, then T_1 also writes final value of A in S_2.

VIEW SERIALIZABILITY

Schedule



Dependency Graph



VIEW SERIALIZABILITY

TIME

Schedule

T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

VIEW

Allows all conflict serializable schedules + "blind writes"

Schedule

T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		
COMMIT		
	BEGIN	
	W(A)	
	COMMIT	
		BEGIN
		W(A)
		COMMIT

SERIALIZABILITY

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

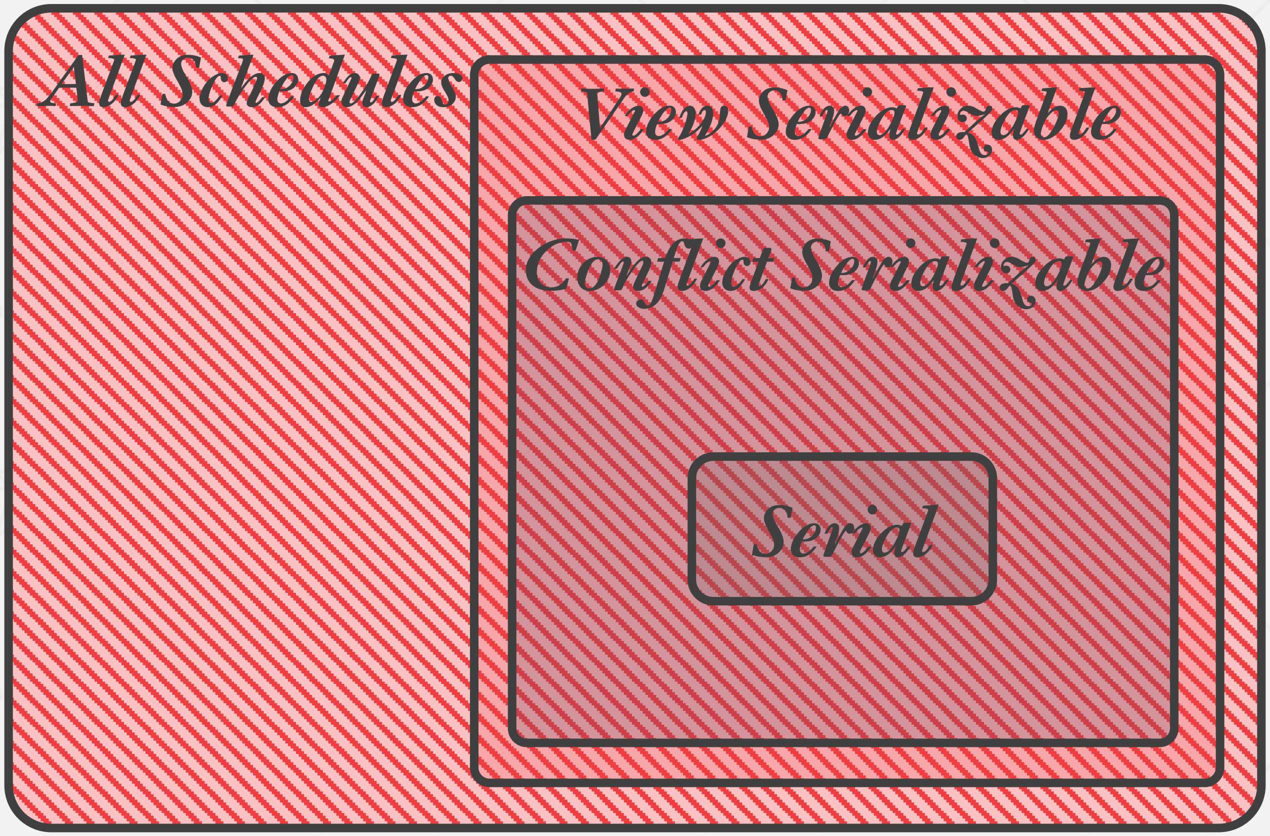
Neither definition allows all serializable schedules.

SERIALIZABILITY

In practice, **Conflict Serializability** is what systems support because it can be enforced efficiently.

To allow more concurrency, some special cases get handled separately at the application level.

UNIVERSE OF SCHEDULES



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.

CONSISTENCY

The database is *consistent* if it satisfies application-specific correctness constraints.
→ Implicit: Informally specified real-world constraints
→ Explicit: DBMS-enforced integrity constraints

Future transactions see the effects of past committed transactions.

A transaction is *consistent* if it takes the database from a consistent state to a consistent state.

CORRECTNESS CRITERIA: ACID

<u>A</u>tomicity All actions in txn happen, or none happen.
"All or nothing…"

<u>C</u>onsistency If each txn is consistent and the DB starts consistent, then it ends up consistent.
"It looks correct to me…"

<u>I</u>solation Each txn sees the DB as if it's running alone in the DB.
"All by myself…"

<u>D</u>urability If a txn commits, its effects persist.
"I will survive…"

CONCLUS

Concurrency control and reco
most important functions pro

Concurrency control is autom
→ System automatically inserts lock

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

tency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [9], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [37].) Many applications at Google have chosen to use Megastore [5] because of its semi-relational data model and support for synchronous repli-

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