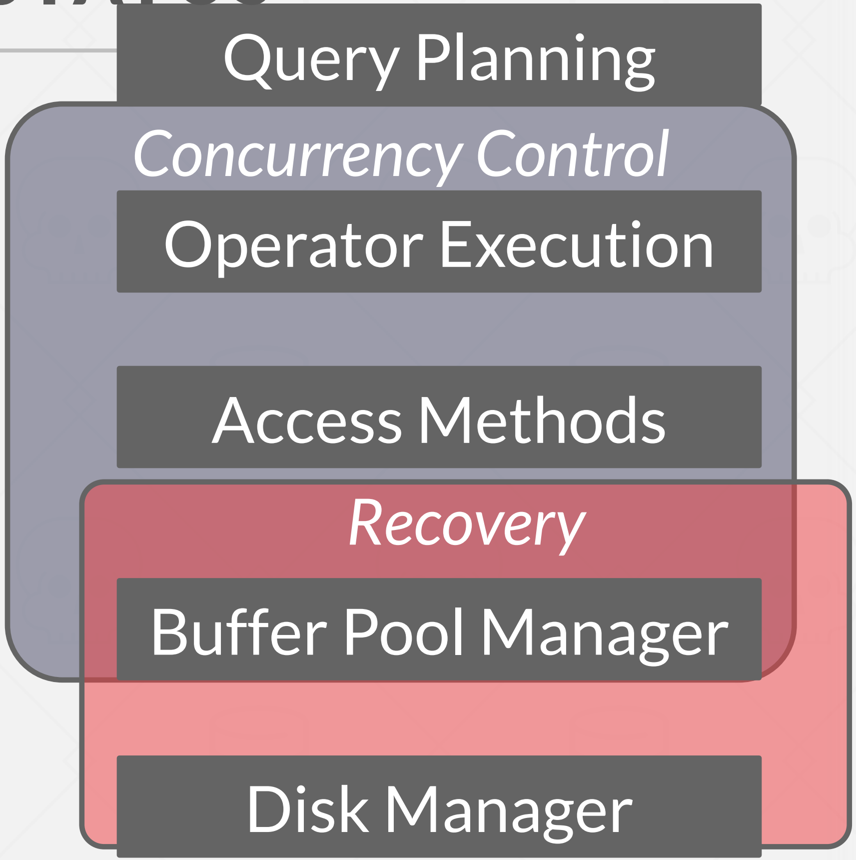
COURSE STATUS

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

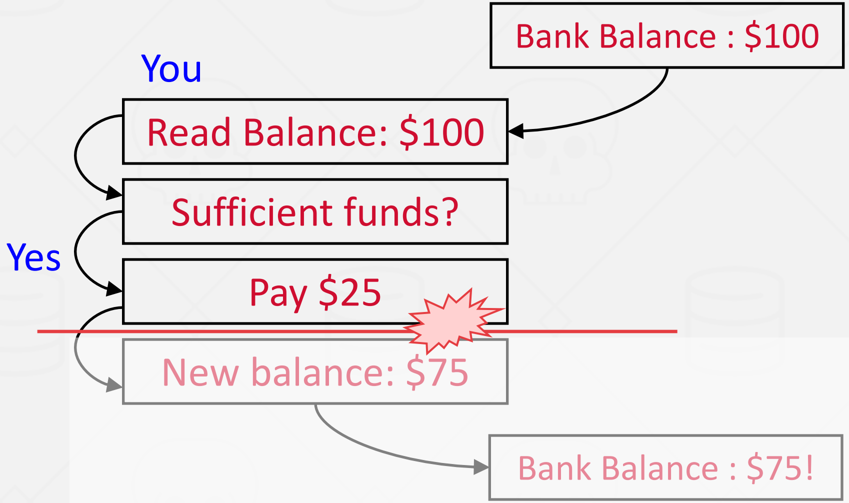
COURSE STATUS

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



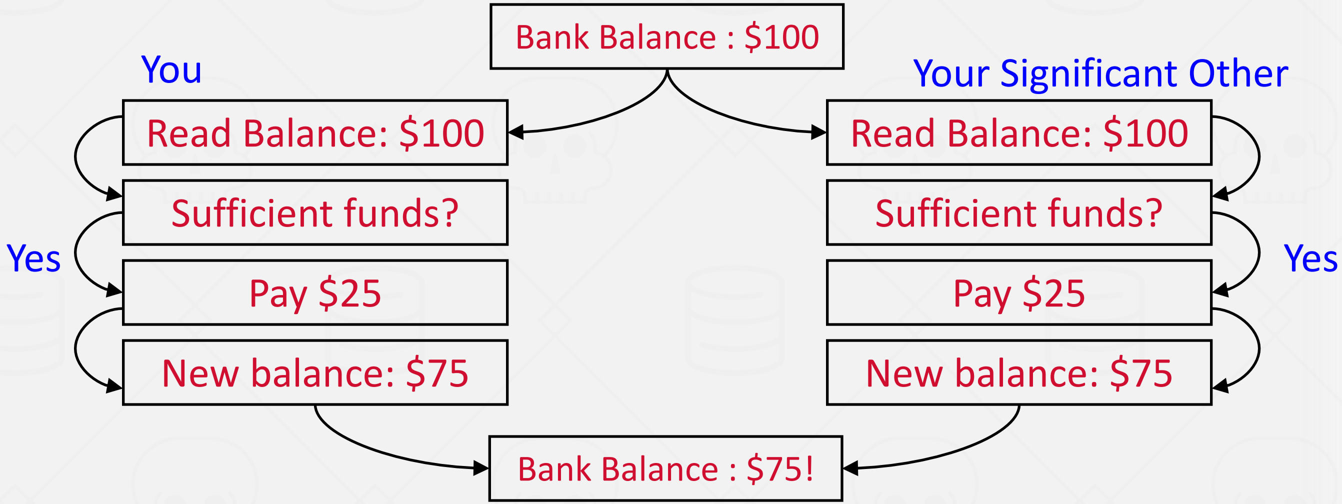
TRANSACTION MANAGEMENT

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



TRANSACTION MANAGEMENT

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



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 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!)

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

DEFINITIONS

A txn may carry out many operations on the data retrieved from the database

The DBMS is <u>only</u> concerned about what data is read/written from/to the database.

→ Changes to the "outside world" are beyond the scope of the DBMS.

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 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

Redo/Undo mechanism

<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	Execution of one txn is isolated from that of other txns. *"All by myself..."*
<u>D</u>urability	If a txn commits, its effects persist. *"I will survive..."*

Integrity Constraint

Concurrency Control

Redo/Undo mechanism

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.

ATOMICITY OF TRANSACTIONS

Scenario #1:

→ We take $100 out of an account, but then the DBMS aborts the txn before we transfer it.

Scenario #2:

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

What should be the correct state of the 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)

CONSISTENCY

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.

→ Responsibility of the Application to define these constraints.

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

A note on Eventual Consistency.

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

→ Difficult for application programmers to reason about such semantics.

→ The trend is to move away from such models.

ISOLATION OF TRANSACTIONS

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

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 **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.

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.

→ 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=960, B=1160

A+B=$2120

INTERLEAVING EXAMPLE (BAD)

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

TIME

≠

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

A+B=$2114

Off by $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	

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	

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

A+B=$2114

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.

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

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

→ Read-Write Conflicts (**R-W**)

→ Write-Read Conflicts (**W-R**)

→ Write-Write Conflicts (**W-W**)

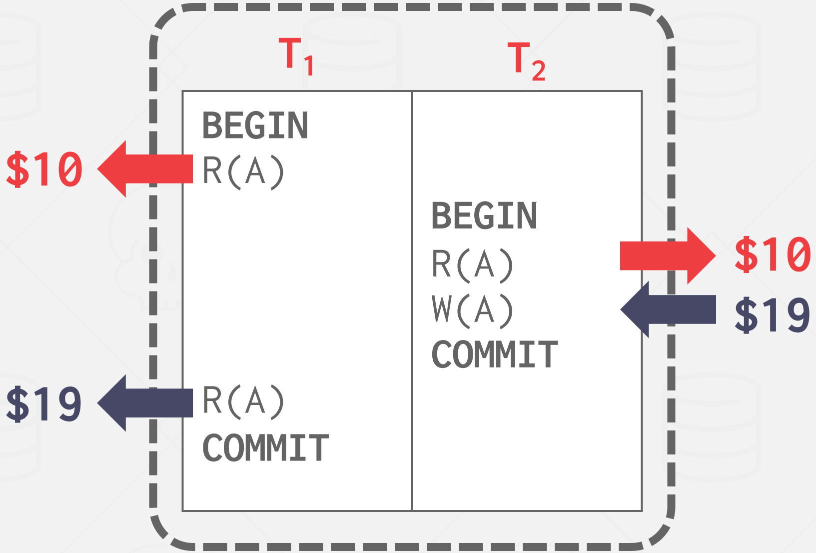
READ-WRITE CONFLICTS

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

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

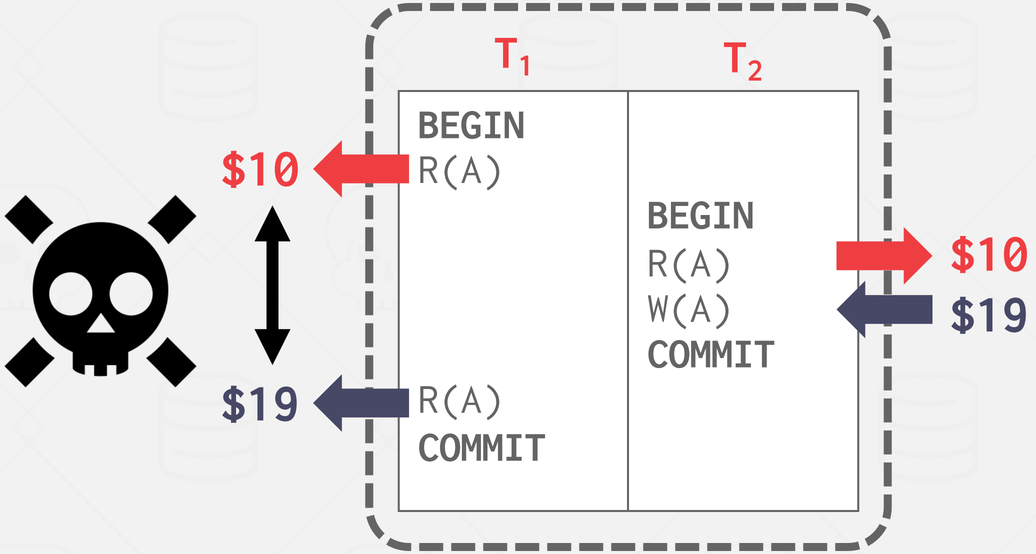
READ-WRITE CONFLICTS

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



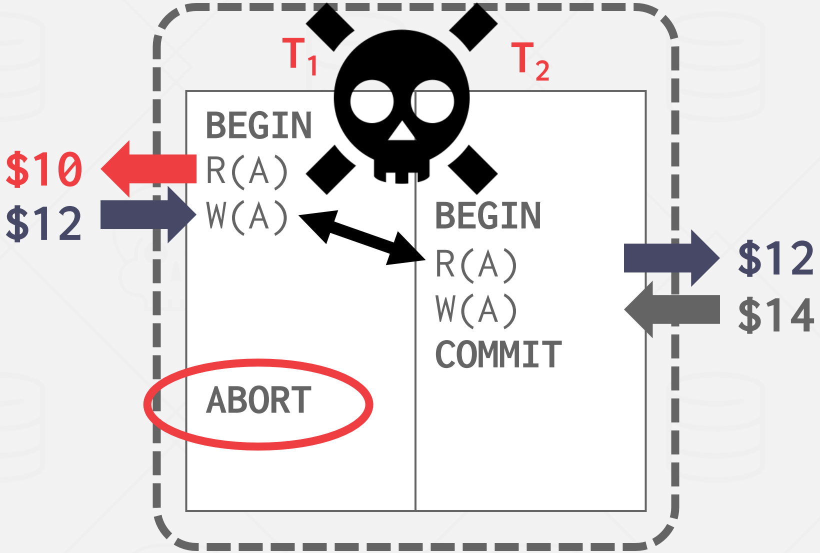
READ-WRITE CONFLICTS

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



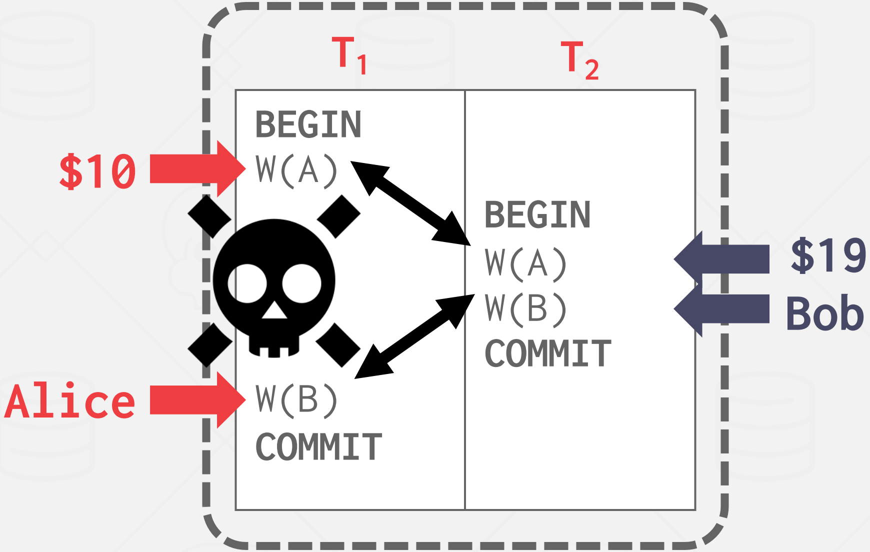
WRITE-READ CONFLICTS

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 not how to generate a correct schedule.

There are different levels of serializability:

→ **Conflict Serializability**

Most DBMSs try to support this.

→ **View Serializability**

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.

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)
W(B)	W(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

SERIALIZABILITY

Swapping operations is easy when there are only two txns in the schedule. It's cumbersome when there are many txns.

Are there faster algorithms to figure this out other than transposing operations?

DEPENDENCY GRAPHS

One node per txn.

Edge from T_i to T_j if:

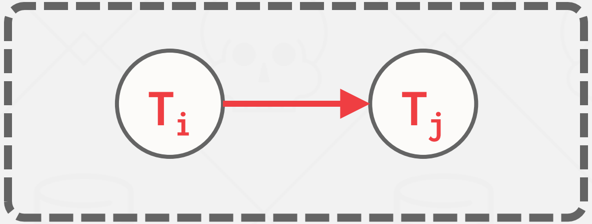
→ An operation O_i of T_i conflicts with an operation O_j of T_j and

→ 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



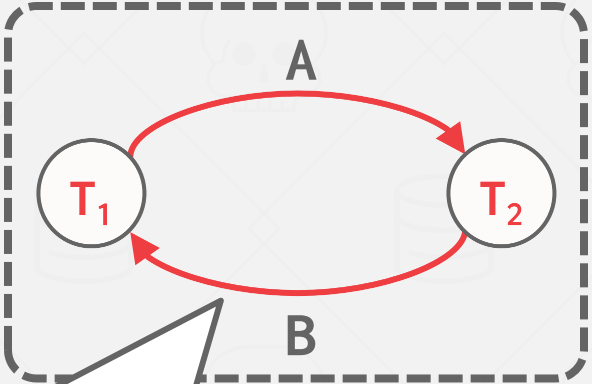
EXAMPLE #1

Schedule

TIME

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 the problem.
The output of T_1 depends on T_2, and vice-versa.

EXAMPLE #2 – THREE TRANSACTIONS

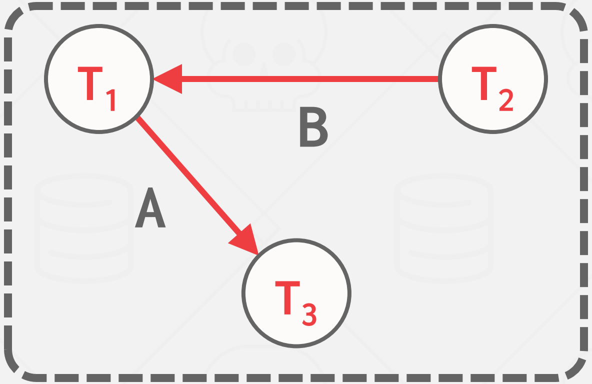
Is this equivalent to a serial execution?

Schedule

TIME

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

Dependency Graph



Yes (T_2, T_1, T_3)

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

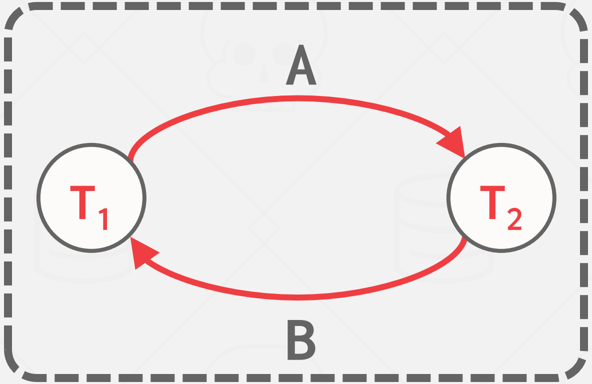
EXAMPLE #3 – INCONSISTENT ANALYSIS

TIME

Schedule

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
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



Is it possible to modify only 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 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.

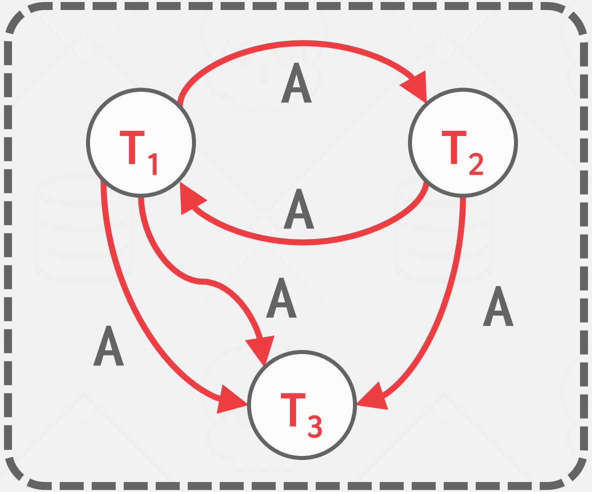
VIEW SERIALIZABILITY

TIME

Schedule

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

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

Schedule

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

Allows all conflict serializable schedules + "blind writes"

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."

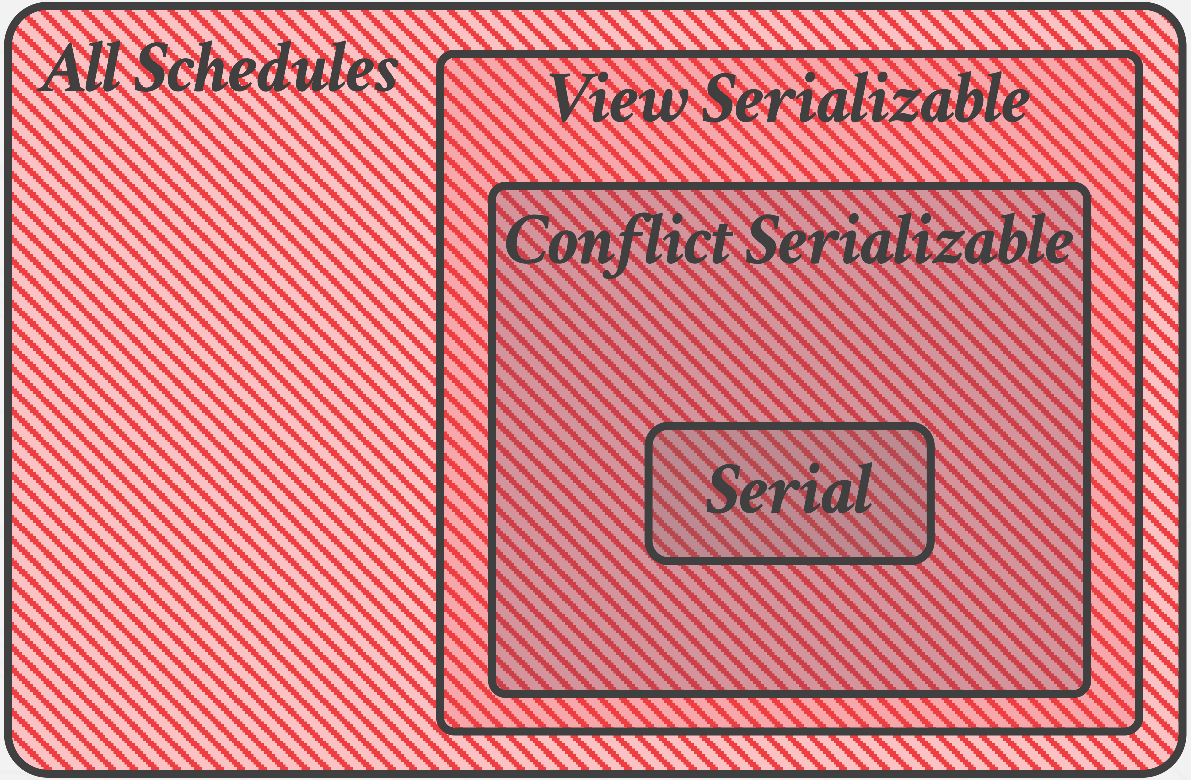
→ This is because they don't understand the meanings of the operations or the data (recall example #3)

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.

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…"

CONCLUSION

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 different txns.

→ Ensures that resulting execution is equivalent to executing the txns one after the other in some order.

CONCLUS

Concurrency control and recovery
important functions provided by

Concurrency control is automatic
→ System automatically inserts lock/un

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-
...pite its relatively poor write throughput. As a
...e, Spanner has evolved from a Bigtable-like
...ey-value store into a temporal multi-version
...Data is stored in schematized semi-relational
... is versioned, and each version is automati-
...amped with its commit time; old versions of
...ject to configurable garbage-collection poli-
...plications can read data at old timestamps.
...ports general-purpose transactions, and pro-
...-based query language.

...ally-distributed database, Spanner provides
...esting features. First, the replication con-
...r data can be dynamically controlled at a
...applications. Applications can specify con-
...trol which datacenters contain which data,
... is from its users (to control read latency),
...as are from each other (to control write la-
...ow many replicas are maintained (to con-
..., availability, and read performance). Data
...ynamically and transparently moved be-
...ters by the system to balance resource us-
...centers. Second, Spanner has two features
...lt to implement in a distributed database: it

1

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

CONCLUSION

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 different txns.

→ Ensures that resulting execution is equivalent to executing the txns one after the other in some order.

NEXT CLASS

Two-Phase Locking

Isolation Levels