Carnegie Mellon University

15 Concurrency Control Theory

Intro to Database Systems
15-445/15-645
Fall 2021

Lin Ma
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #4 will be released on Wednesday. It is due Sun Nov 7th @ 11:59pm.

Project #3 is due Sun Nov 14th @ 11:59pm

Project #2 practice submission available on Gradescope

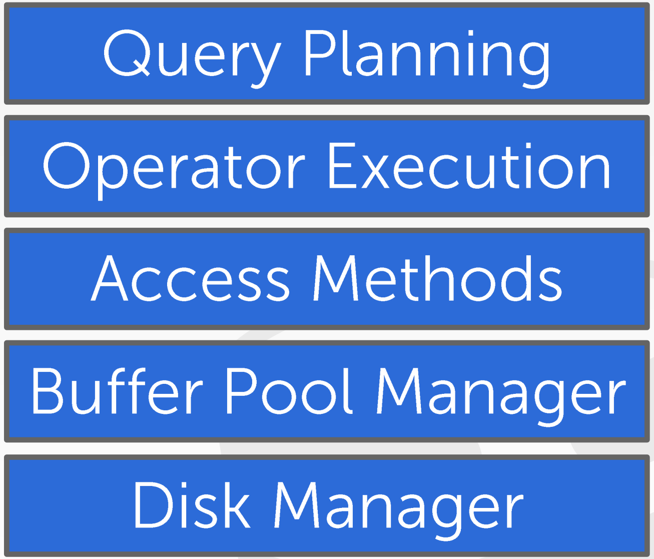
UPCOMING DATABASE TALK

An Overview of the Starburst Trino Query Optimizer

→ Today Oct 25th @ 4:30pm ET

COURSE STATUS

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



Query Planning

Operator Execution

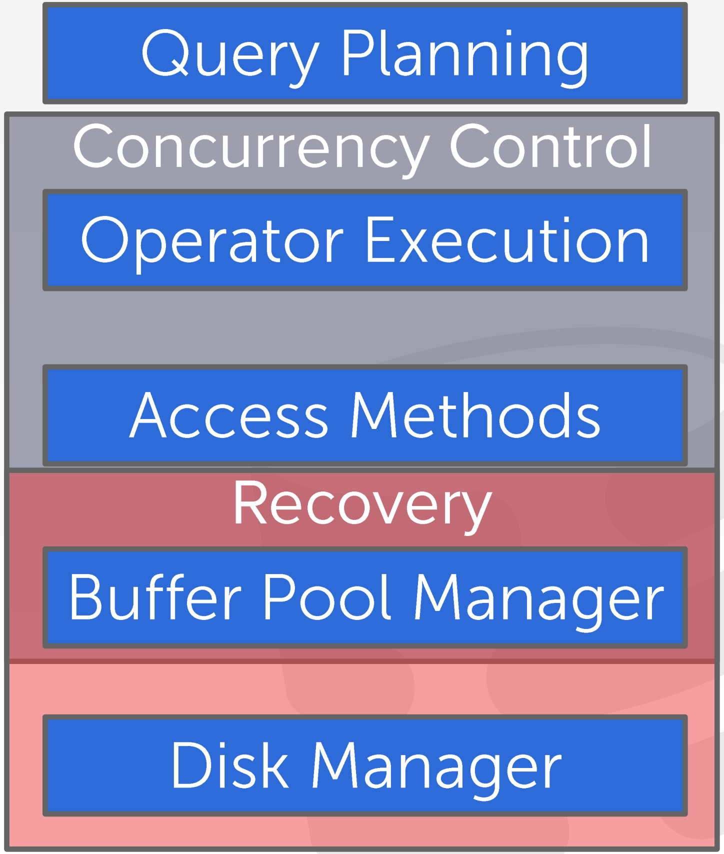
Access Methods

Buffer Pool Manager

Disk Manager

COURSE STATUS

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



MOTIVATION

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

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

MOTIVATION

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

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** 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:
→ Partial transactions are not allowed!

TRANSACTION EXAMPLE

Move $100 from Lin' bank account to his promotor's account.

Transaction:

→ Check whether Lin has $100.
→ Deduct $100 from his account.
→ Add $100 to his promotor 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?

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.

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 only 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 fixed set of named data objects (e.g., A, B, C, …).

→ We do not need to define what these objects are now.

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 the txn happen, or none happen.

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one txn is isolated from that of other txns.

Durability: If a txn commits, its effects persist.

CORRECTNESS CRITERIA: ACID

Atomicity: “all or nothing”

Consistency: “it looks correct to me”

Isolation: “as if alone”

Durability: “survive failures”

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 Lin's account but then the DBMS aborts the txn before we transfer it.

Scenario #2:

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

What should be the correct state of Lin'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 System R.

Few systems do this:

→ CouchDB

→ LMDB (OpenLDAP)

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 System R.

Few systems do this:

→ CouchDB

→ LMDB (OpenLDAP)

CONSISTENCY

The "world" represented by the database is logically correct. All questions asked about the data are given logically correct answers.

Database Consistency

Transaction Consistency

DATABASE CONSISTENCY

The database accurately models the real world and follows integrity constraints.

Transactions in the future see the effects of transactions committed in the past inside of the database.

TRANSACTION CONSISTENCY

If the database is consistent before the transaction starts (running alone), it will also be consistent after.

Transaction consistency is the application's responsibility. DBMS cannot control this.

→ We won't discuss this issue further…

ISOLATION OF TRANSACTIONS

Users submit txns, and each txn executes as if it was running by itself.

→ Easier programming model to reason about.

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
```

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=960, B=1160

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

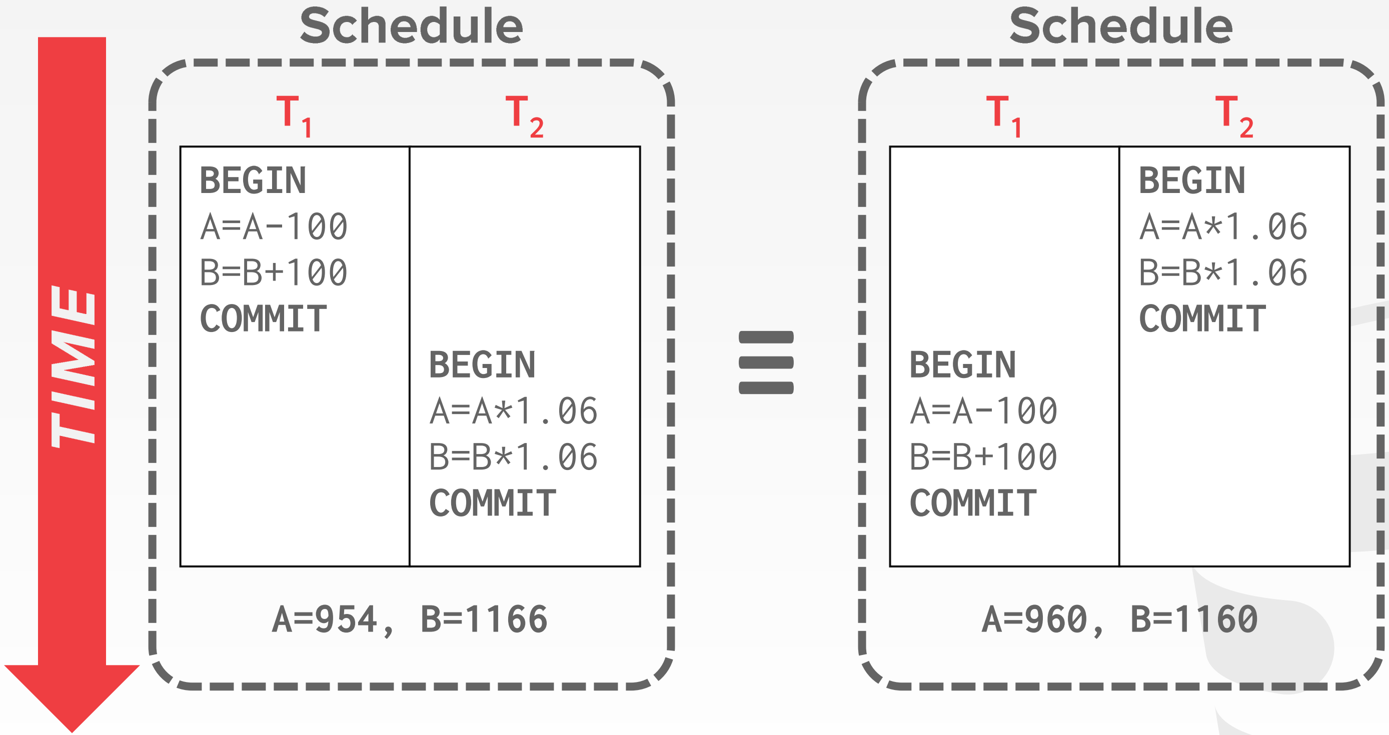
EXAMPLE

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



Schedule

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

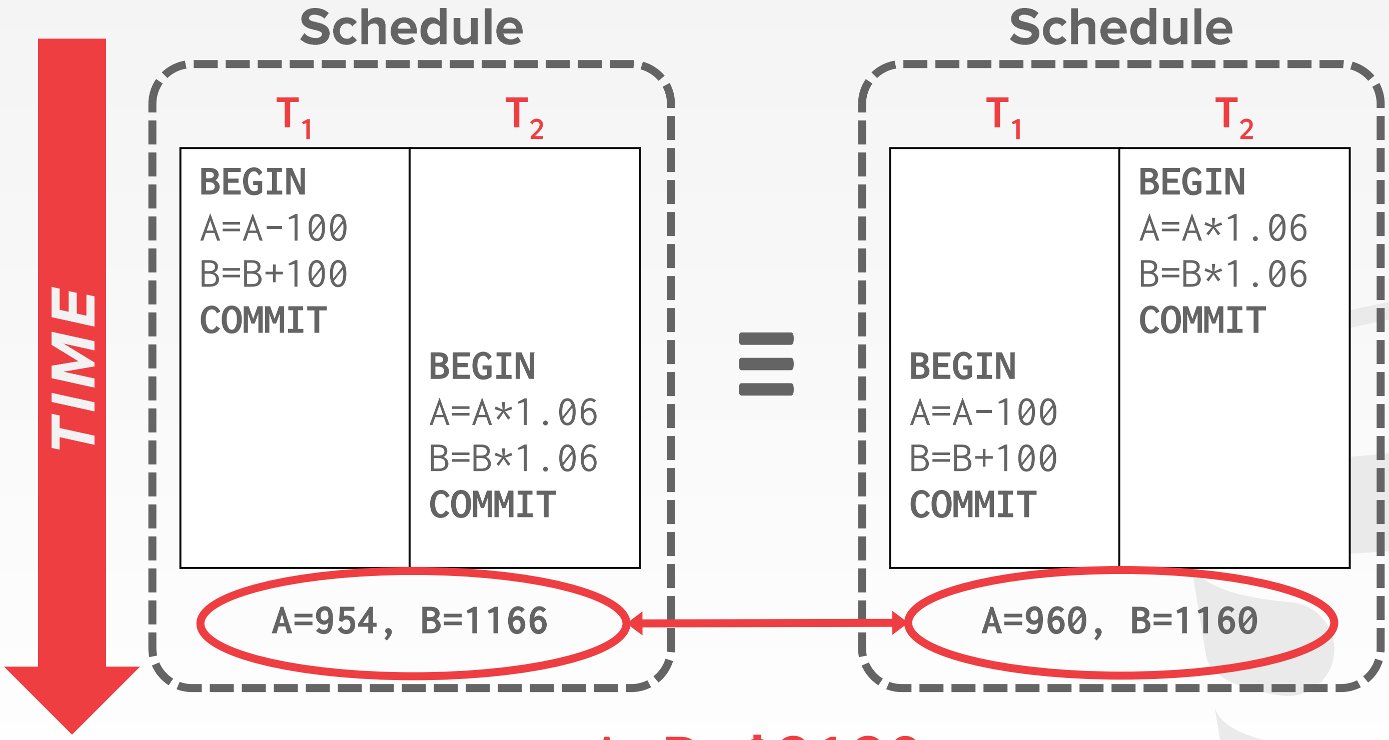
A=954, B=1166

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

SERIAL EXECUTION EXAMPLE



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

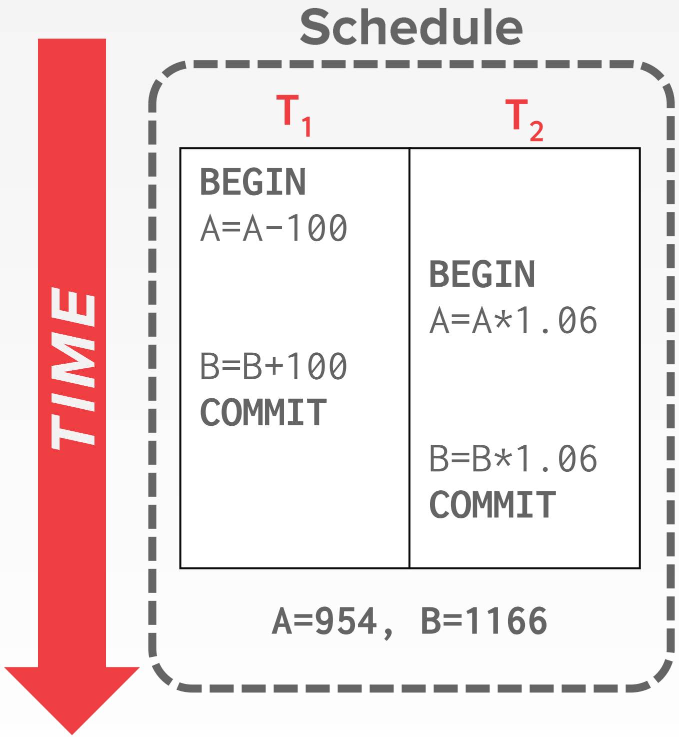
INTERLEAVING TRANSACTIONS

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)

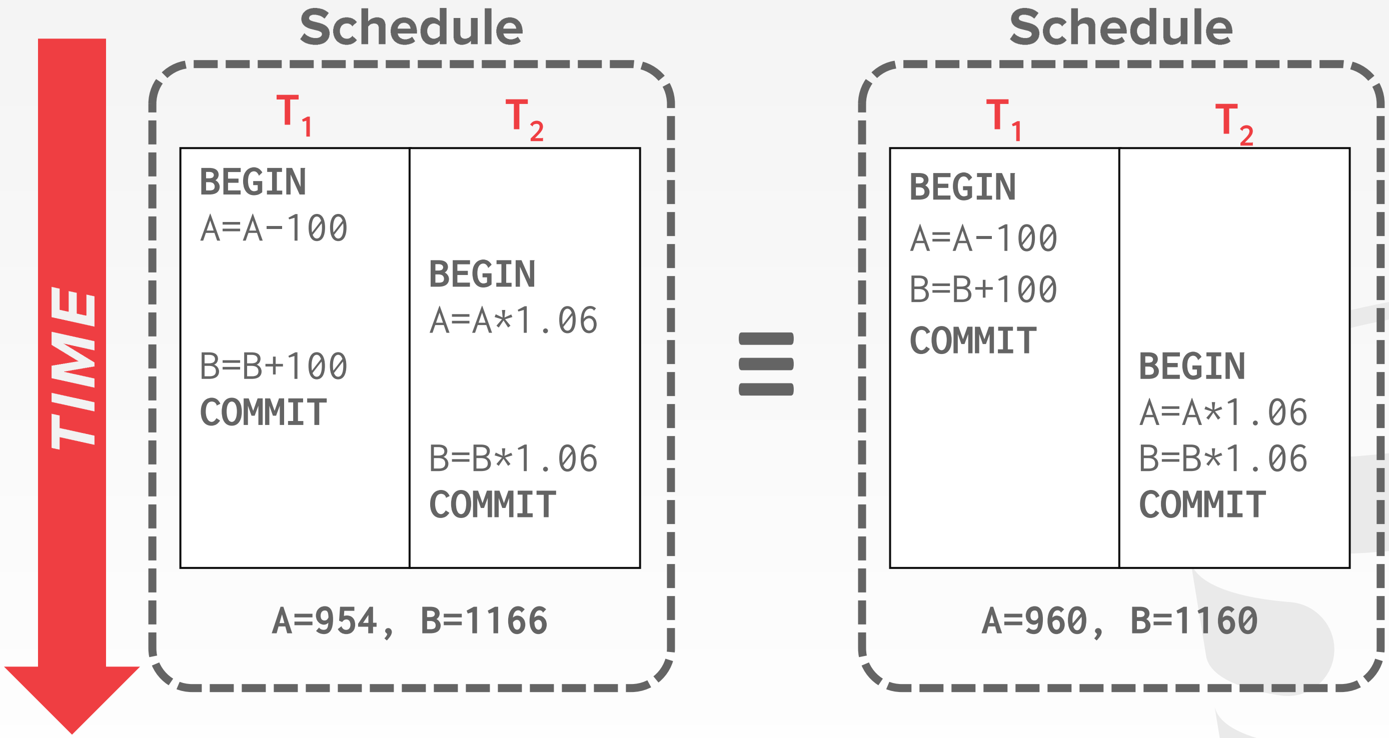


Schedule

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

INTERLEAVING EXAMPLE (GOOD)



Schedule

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

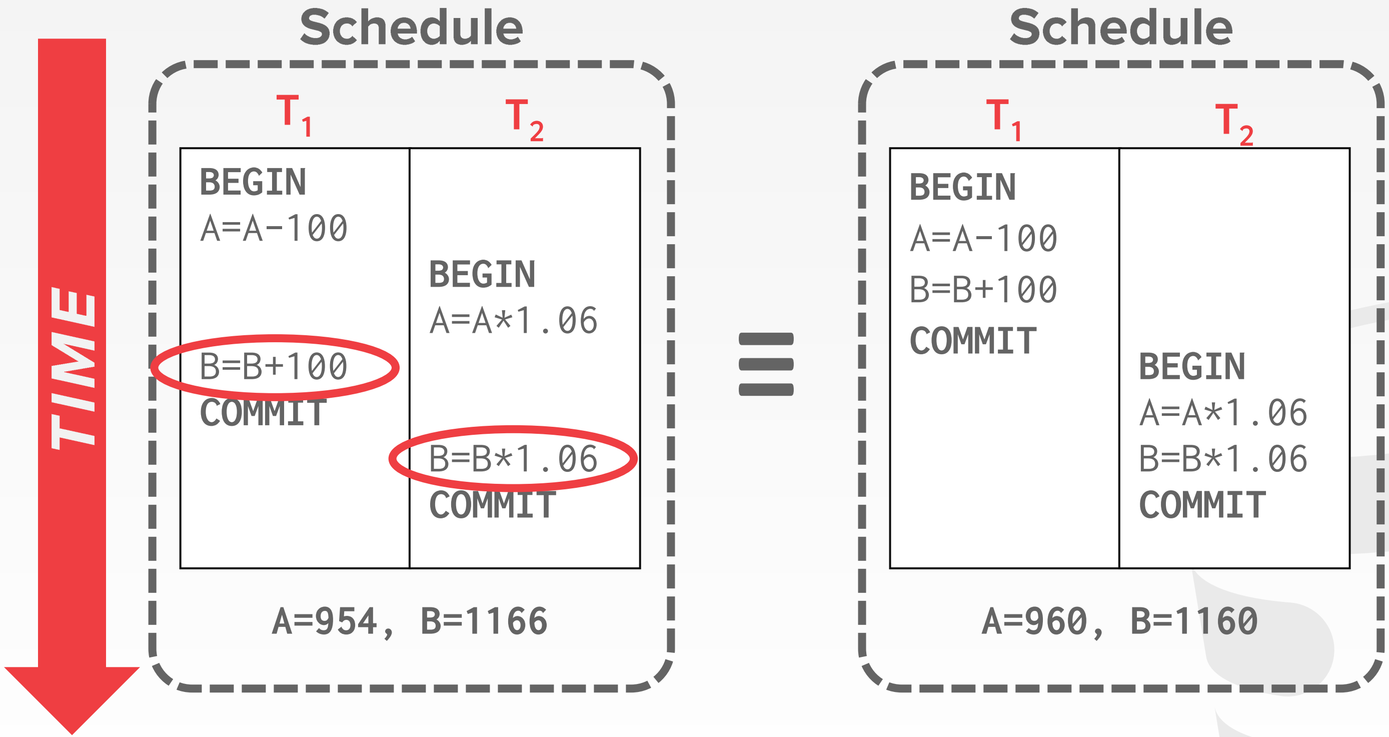
A=954, B=1166

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

INTERLEAVING EXAMPLE (GOOD)



Schedule

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

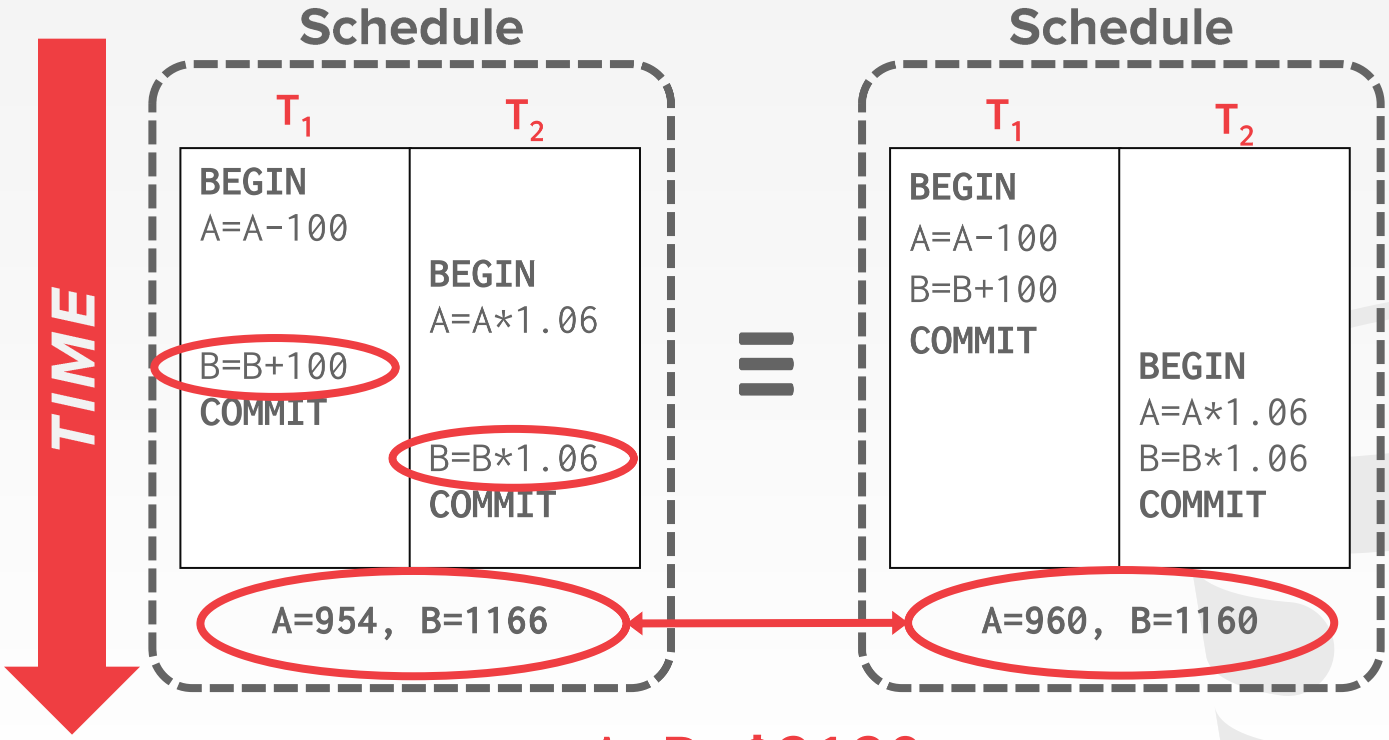
A=954, B=1166

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

INTERLEAVING EXAMPLE (GOOD)



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

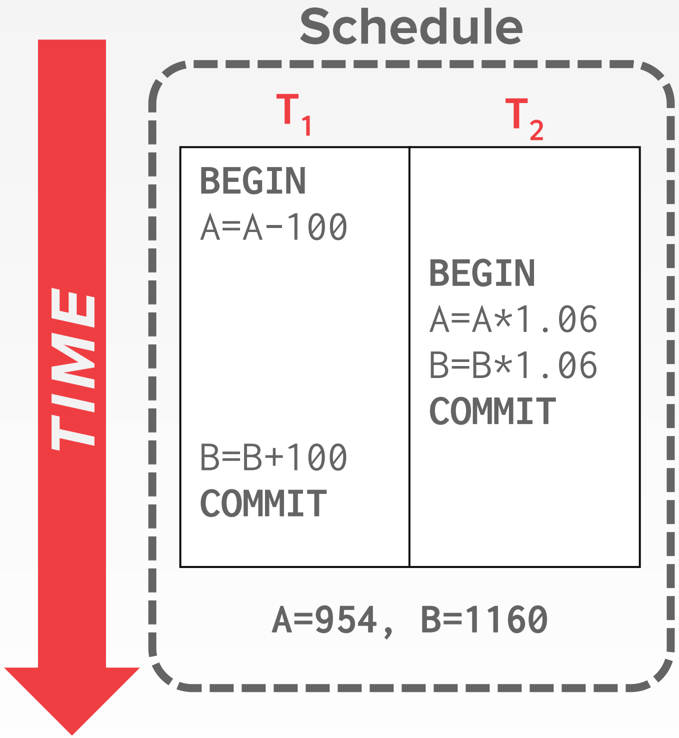
Schedule

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

≠

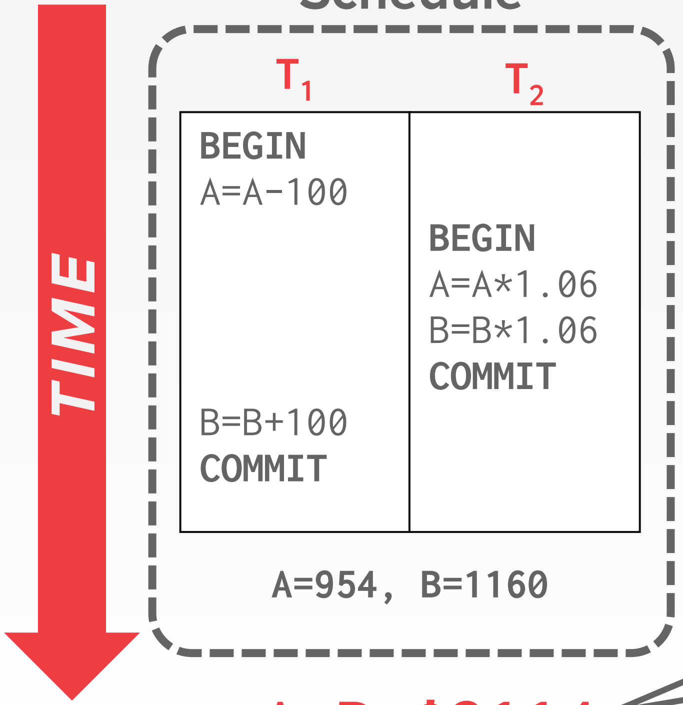
A=954, B=1166

or

A=960, B=1160

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

≠

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	

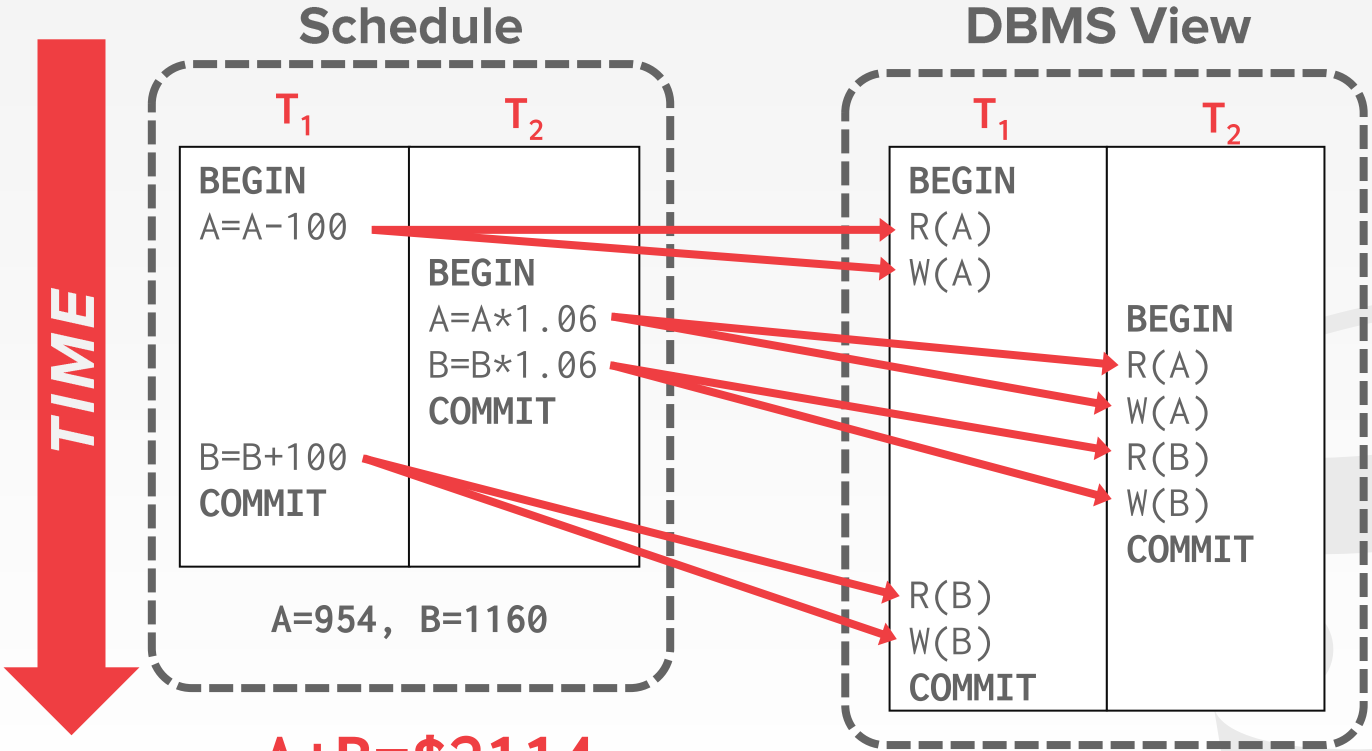
A=954, B=1160

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	

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

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+B=$2114

CORRECTNESS

How do we judge whether a schedule is correct?

CORRECTNESS

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.

→ Doesn't matter what the arithmetic operations are!

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.

FORMAL PROPERTIES OF SCHEDULES

Serializability is a less intuitive notion of correctness compared to txn initiation time or commit order, but it provides the DBMS with additional 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 at least one of them is a write.

INTERLEAVED EXECUTION ANOMALIES

Read-Write Conflicts (**R-W**)

Write-Read Conflicts (**W-R**)

Write-Write Conflicts (**W-W**)

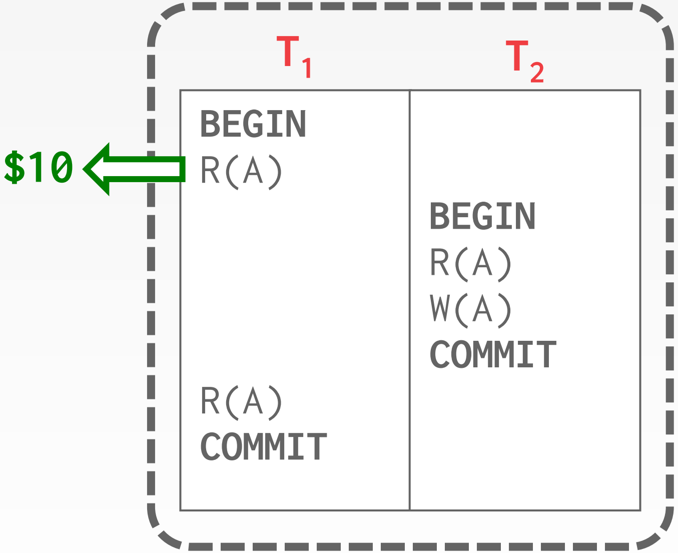
READ-WRITE CONFLICTS

Unrepeatable Reads

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

READ-WRITE CONFLICTS

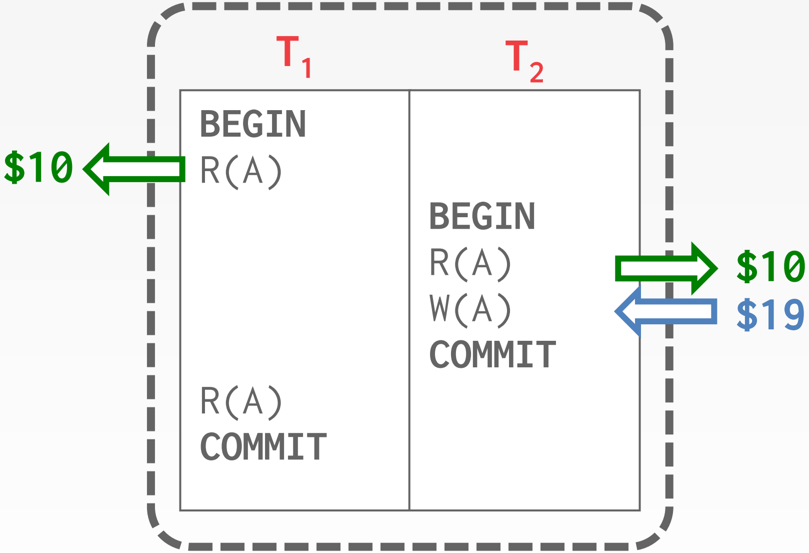
Unrepeatable Reads



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

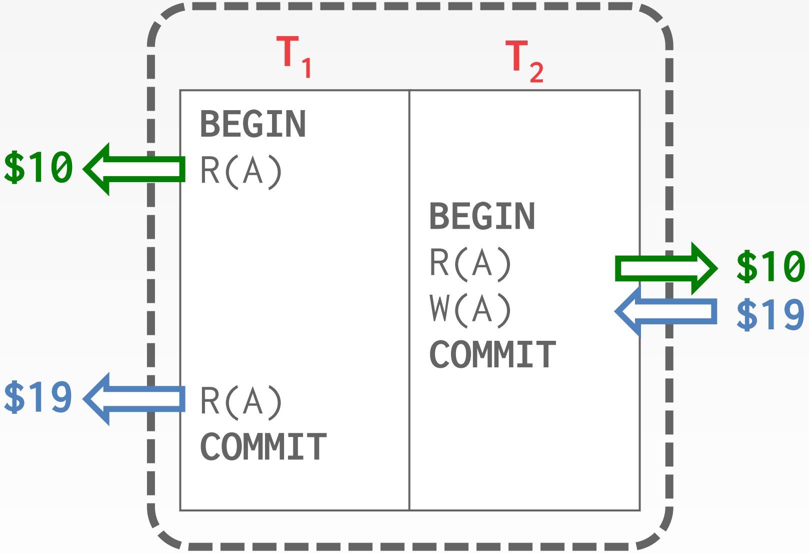
READ-WRITE CONFLICTS

Unrepeatable Reads



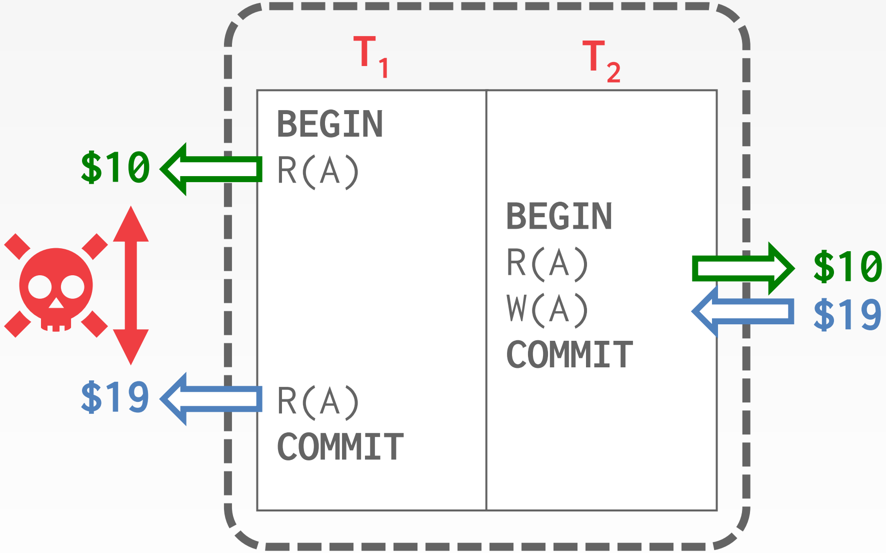
READ-WRITE CONFLICTS

Unrepeatable Reads



READ-WRITE CONFLICTS

Unrepeatable Reads



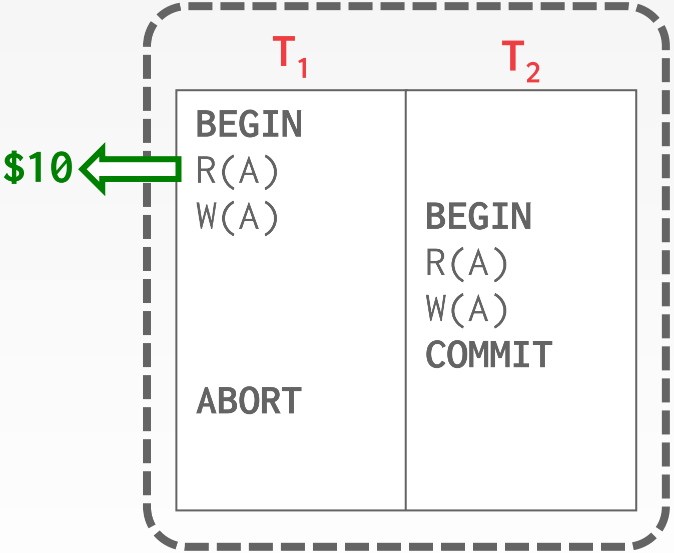
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")

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

WRITE-READ CONFLICTS

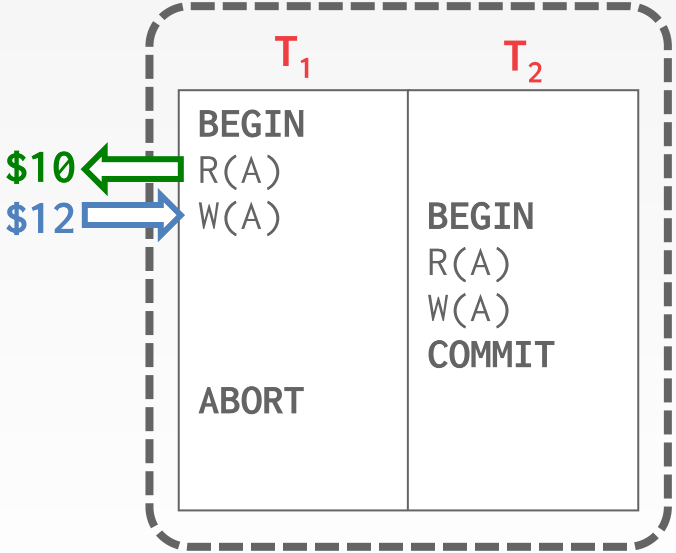
Reading Uncommitted Data ("Dirty Reads")



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

WRITE-READ CONFLICTS

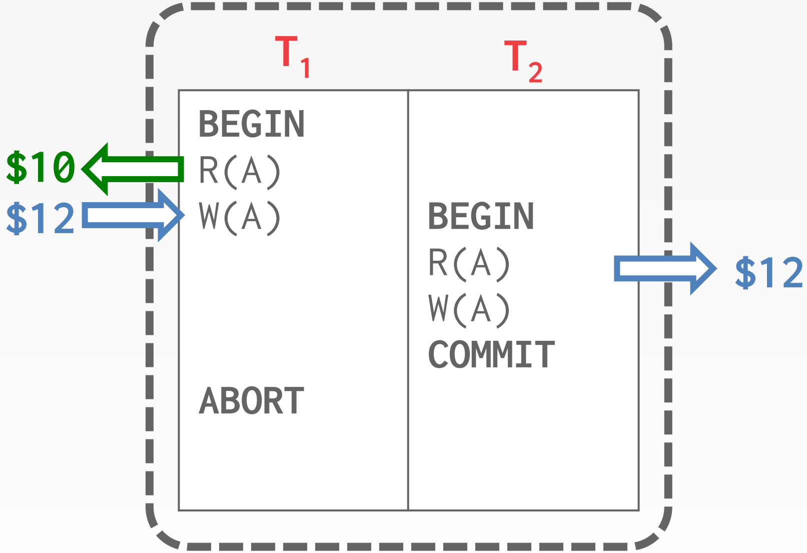
Reading Uncommitted Data ("Dirty Reads")



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

WRITE-READ CONFLICTS

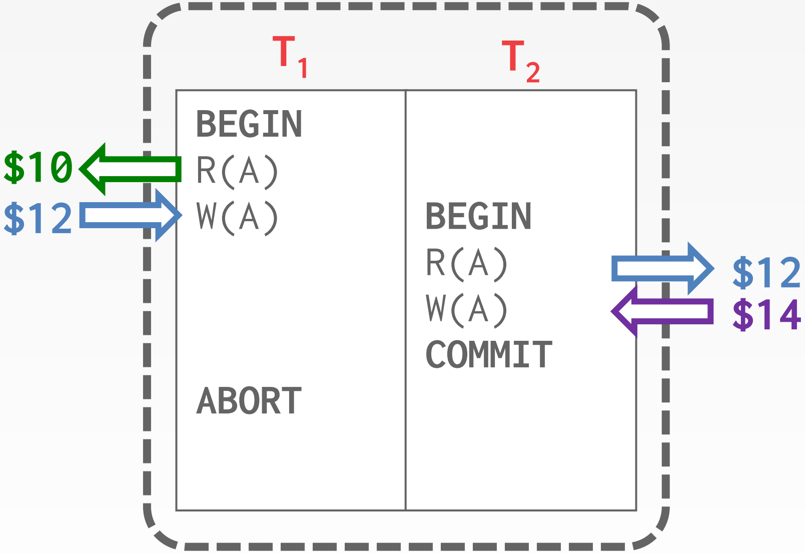
Reading Uncommitted Data ("Dirty Reads")



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

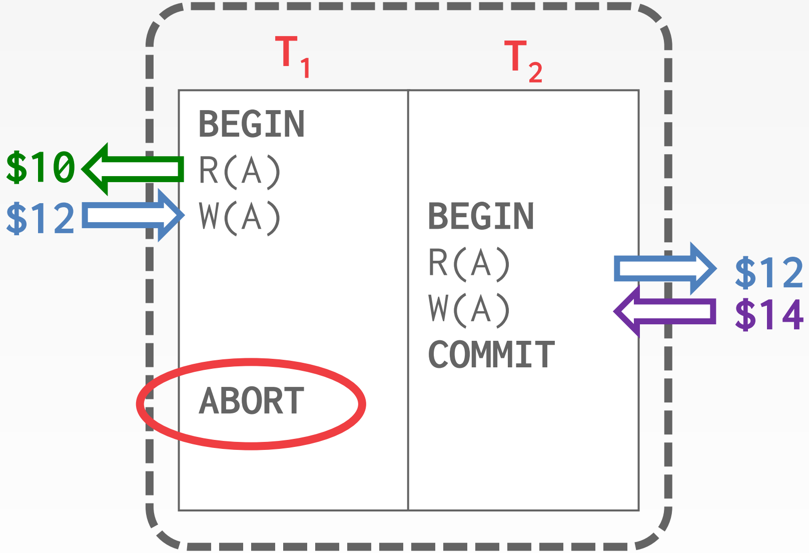
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")



WRITE-READ CONFLICTS

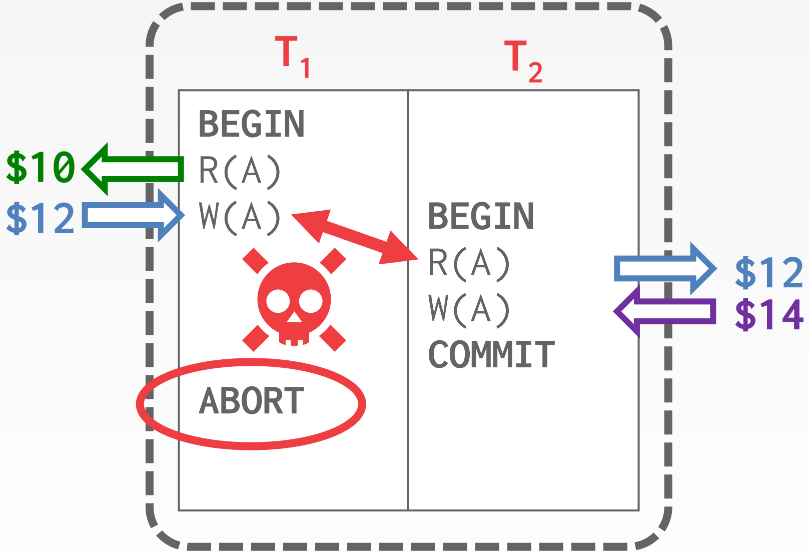
Reading Uncommitted Data ("Dirty Reads")



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

WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")



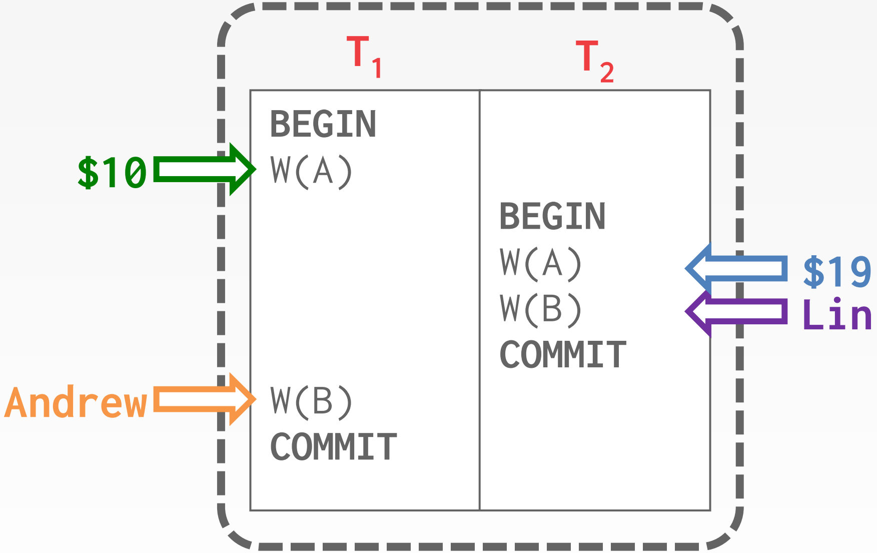
WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data

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

WRITE-WRITE CONFLICTS

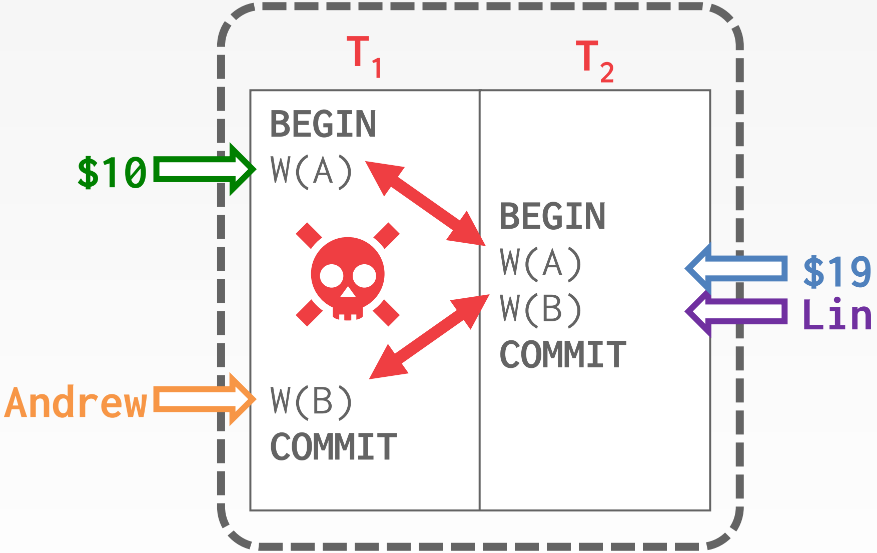
Overwriting Uncommitted Data



	T_1	T_2	
	BEGIN		
$10	W(A)		
		BEGIN	
		W(A)	$19
		W(B)	Lin
		COMMIT	
Andrew	W(B)		
	COMMIT		

WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data



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

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**

→ **View Serializability**

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**

→ **View Serializability**

Most DBMSs try to support this.

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, and

→ Every pair of conflicting actions is ordered the same way.

Schedule S is **conflict serializable** if:

→ S is conflict equivalent to some serial schedule.

CONFLICT SERIALIZABILITY INTUITION

Schedule **S** is conflict serializable if you can transform **S** into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

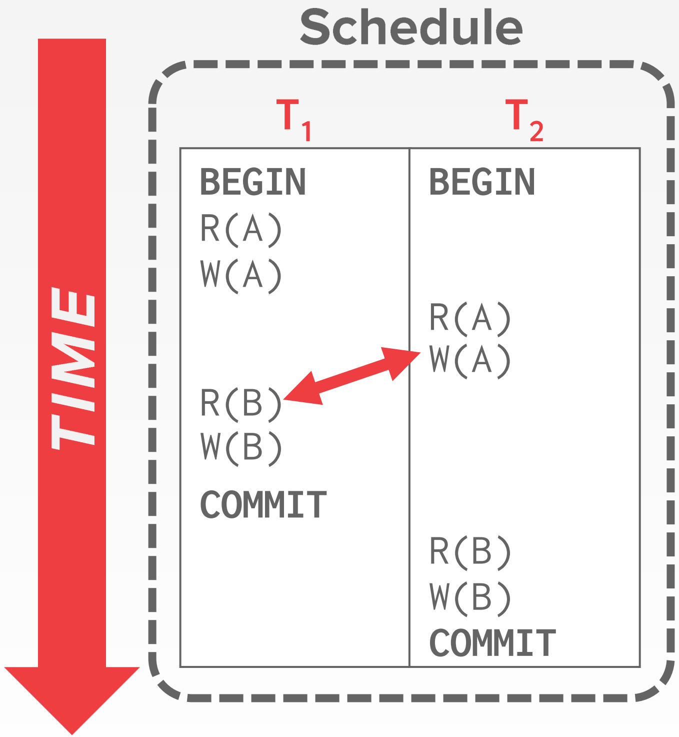
CONFLICT SERIALIZABILITY INTUITION

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

CONFLICT SERIALIZABILITY INTUITION

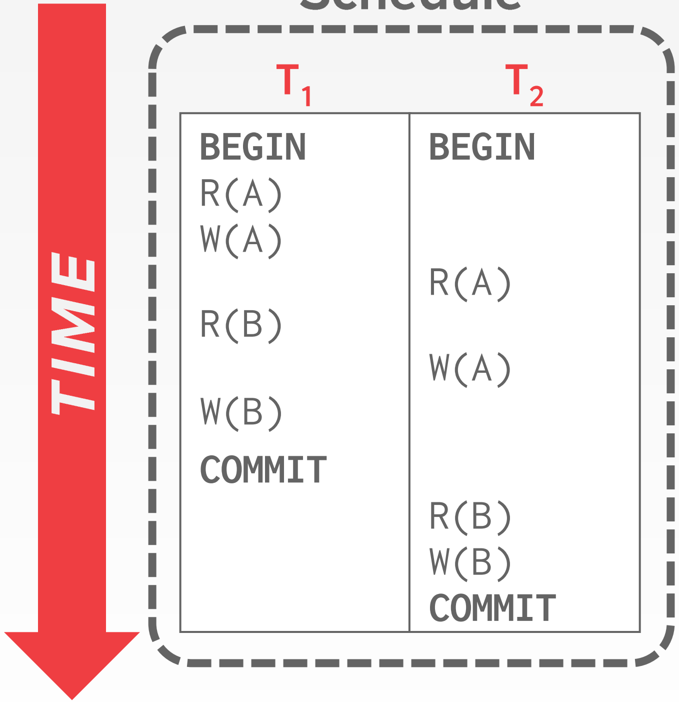


Schedule

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

CONFLICT SERIALIZABILITY INTUITION

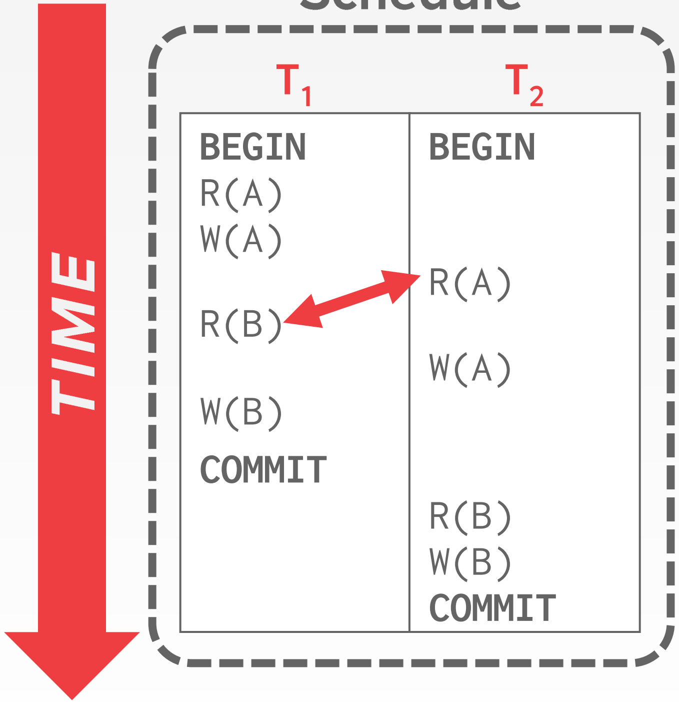
Schedule



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

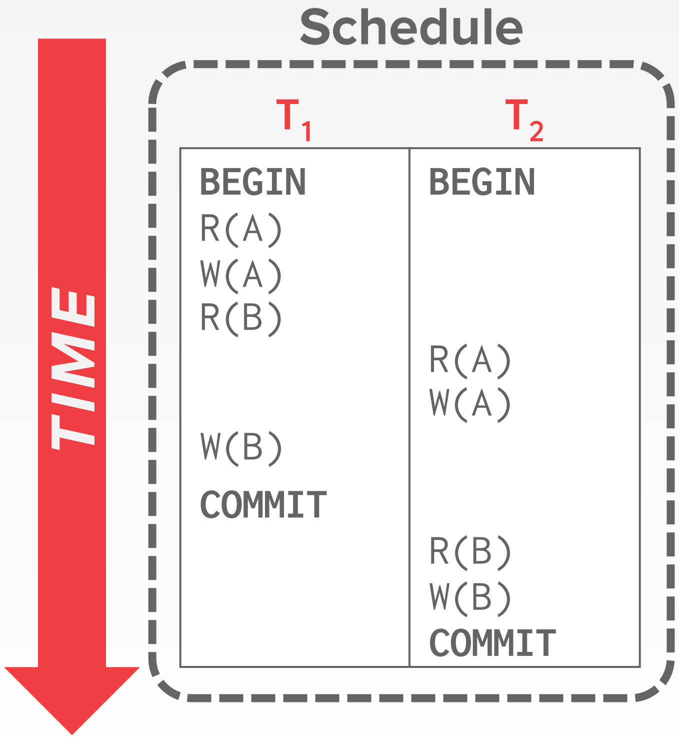
CONFLICT SERIALIZABILITY INTUITION

Schedule



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

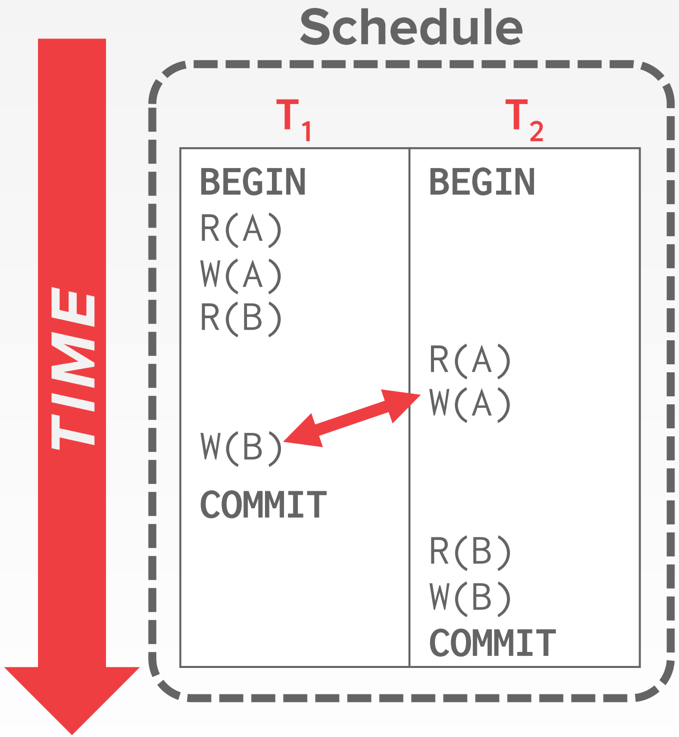
CONFLICT SERIALIZABILITY INTUITION



Schedule

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

CONFLICT SERIALIZABILITY INTUITION



Schedule

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

CONFLICT SERIALIZABILITY INTUITION

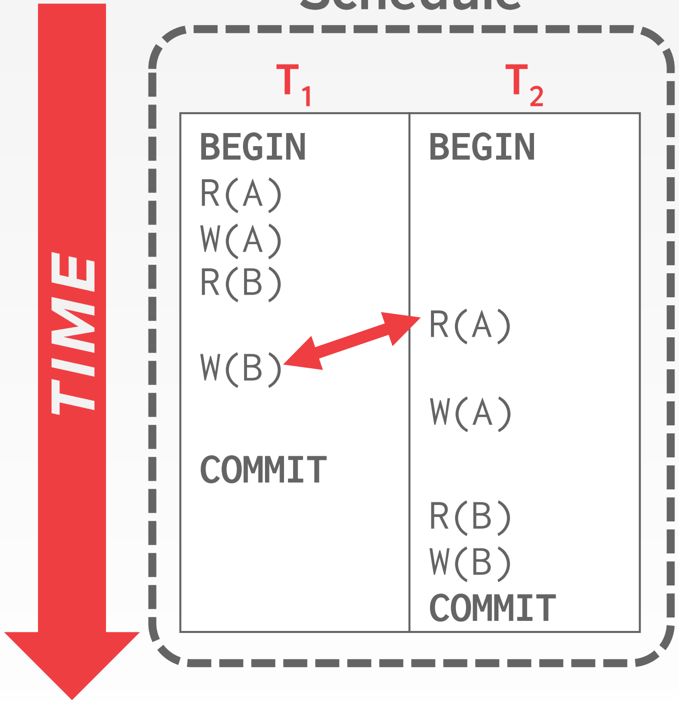
Schedule

TIME

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

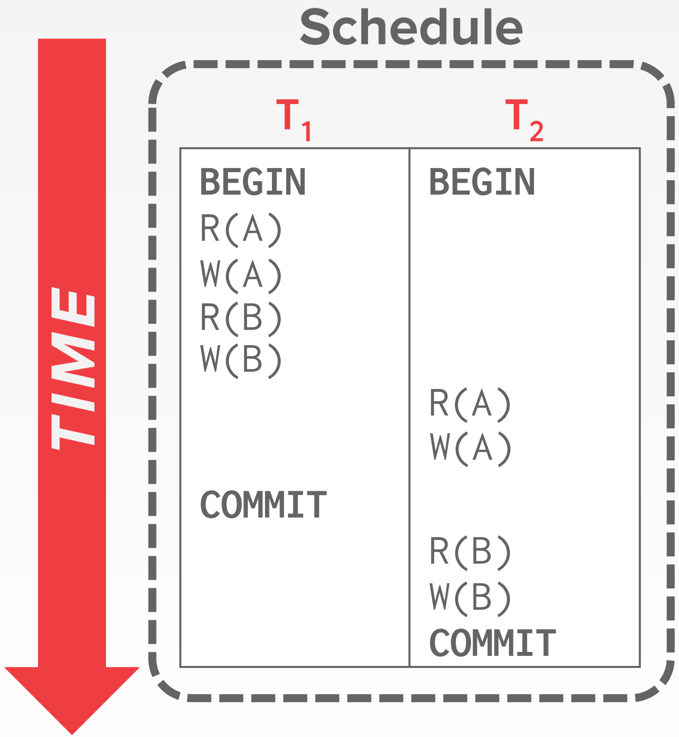
CONFLICT SERIALIZABILITY INTUITION

Schedule



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

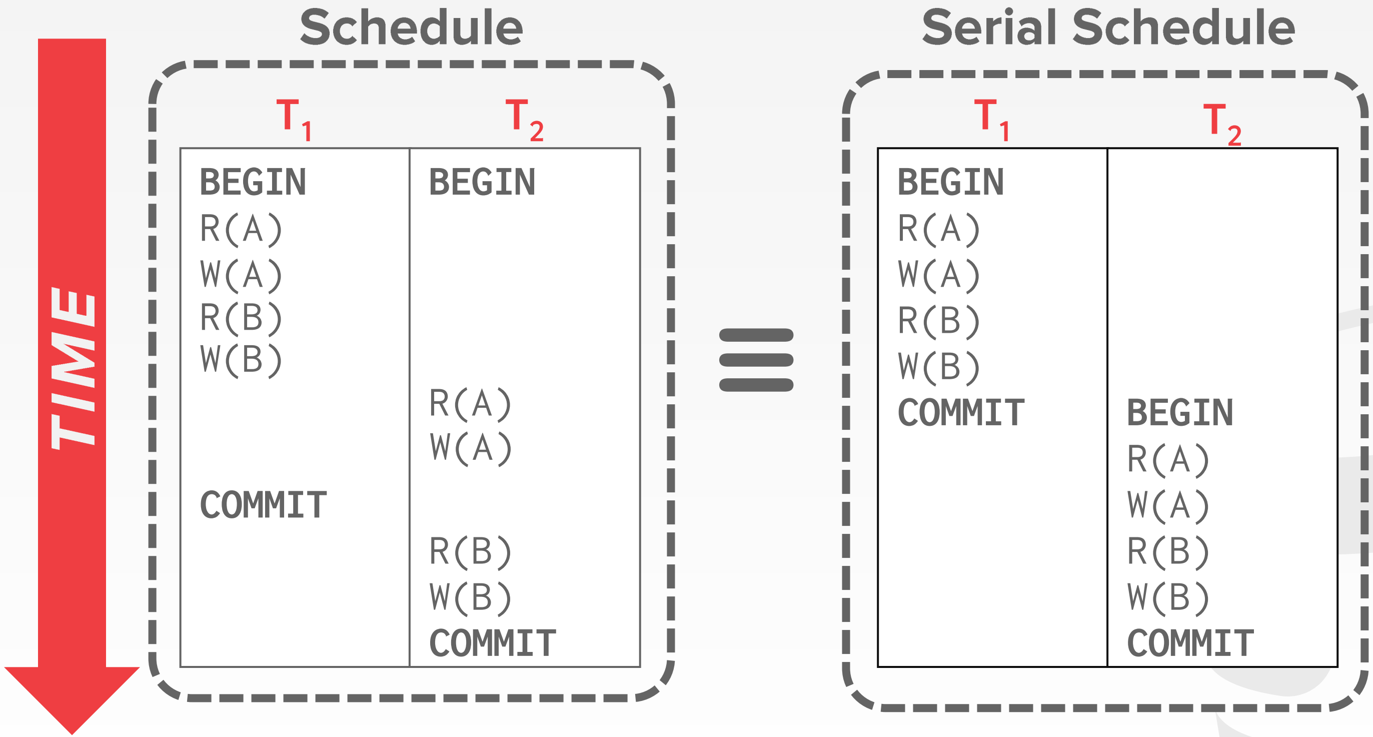
CONFLICT SERIALIZABILITY INTUITION



Schedule

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

CONFLICT SERIALIZABILITY INTUITION



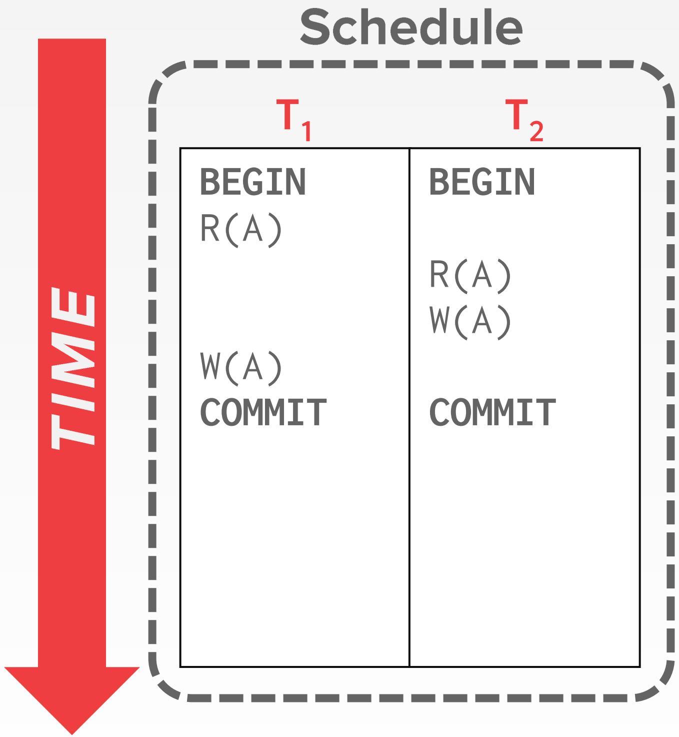
Schedule

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

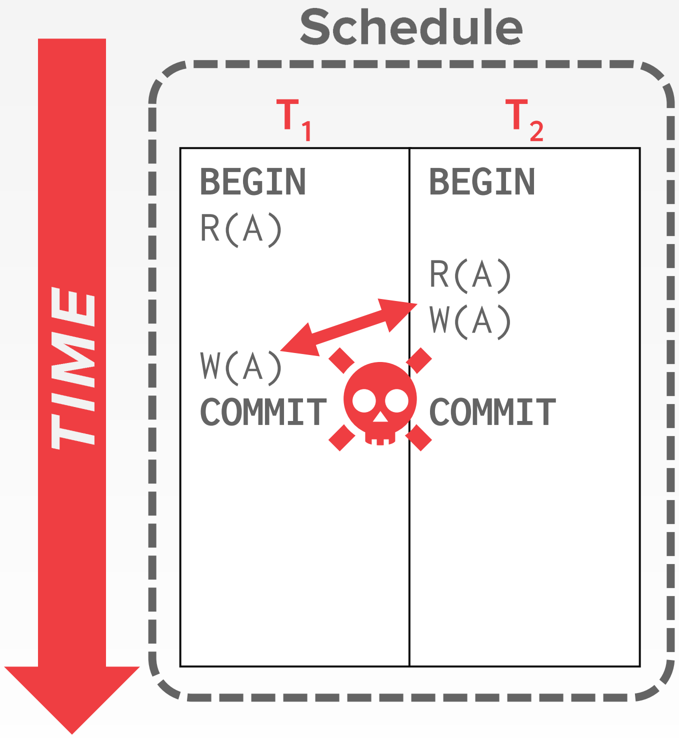
CONFLICT SERIALIZABILITY INTUITION



Schedule

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

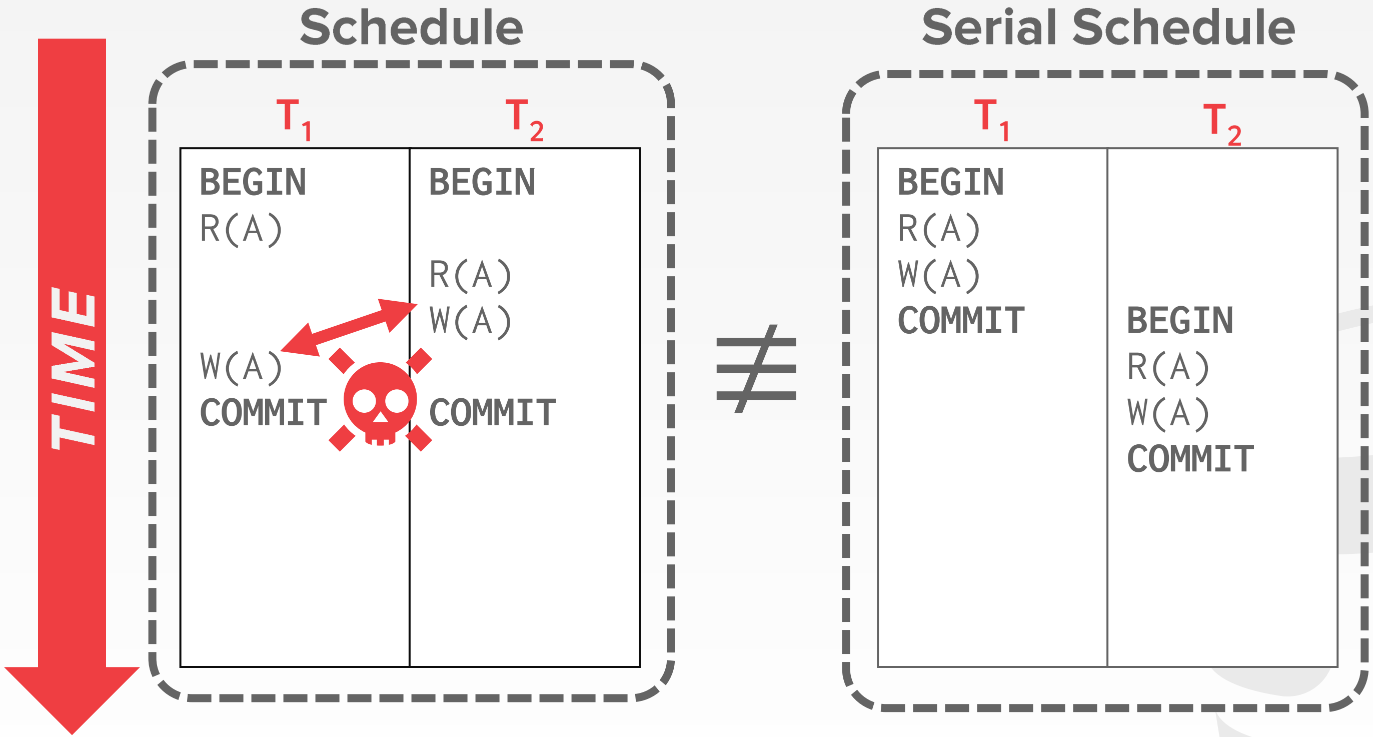
CONFLICT SERIALIZABILITY INTUITION



Schedule

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

CONFLICT SERIALIZABILITY INTUITION



Schedule

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

≠

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 any faster algorithms to figure this out other than transposing operations?

DEPENDENCY GRAPHS

One node per txn.

Edge from T_i to T_j if:

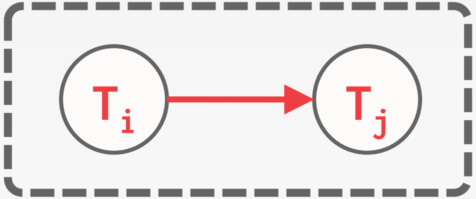
→ An operation O_i of T_i conflicts with an operation O_j of T_j and

→ 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



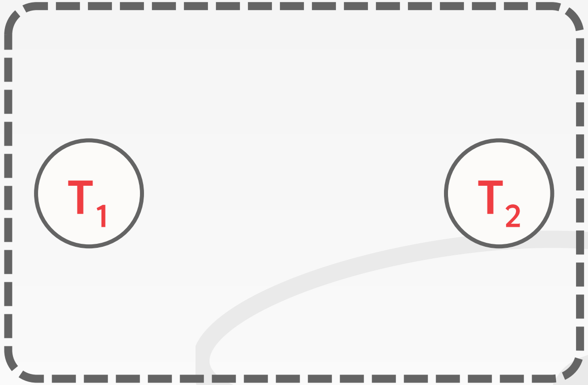
EXAMPLE #1

Schedule

TIME

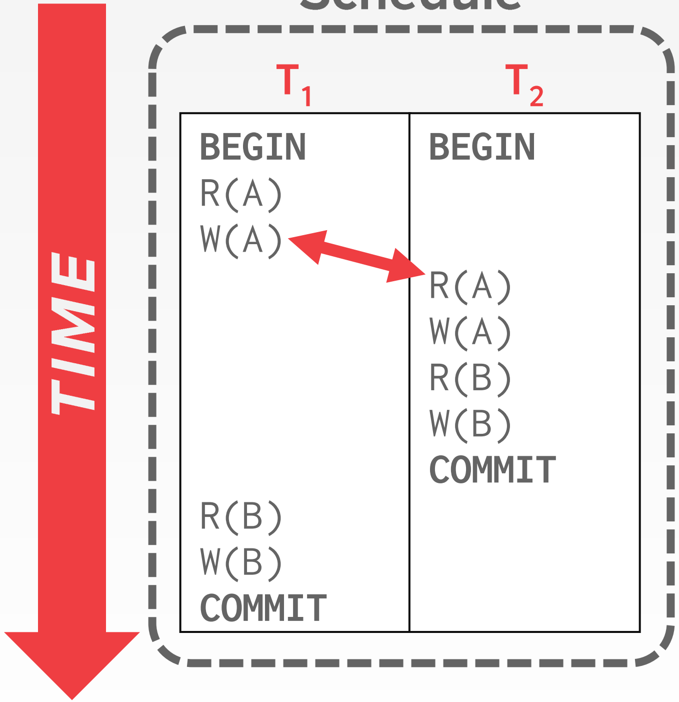
T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph



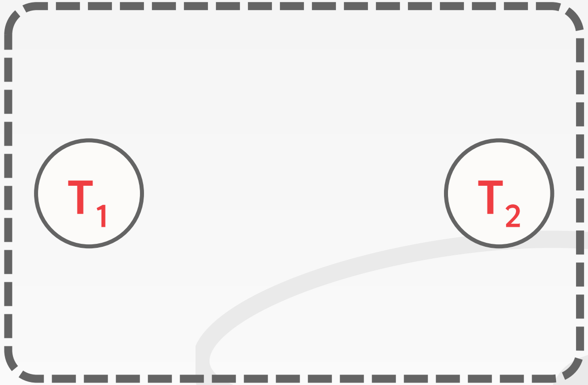
EXAMPLE #1

Schedule



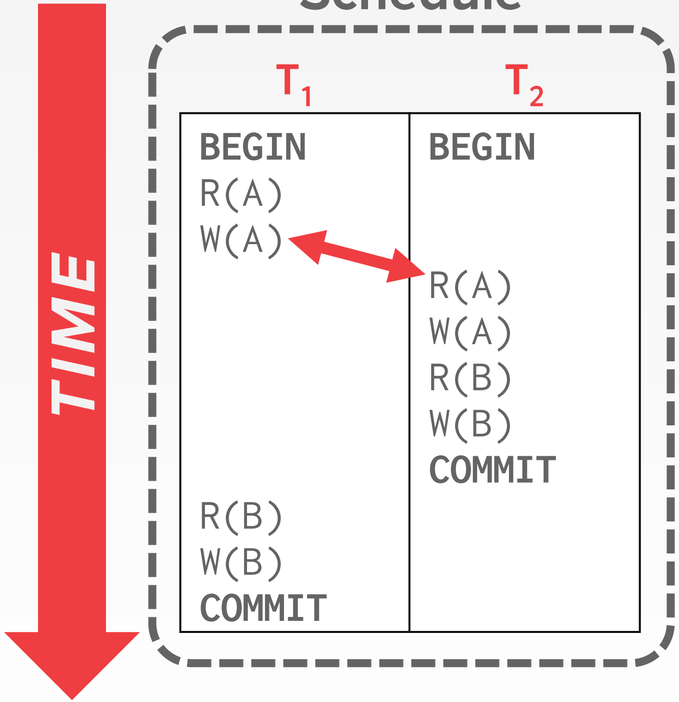
T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph



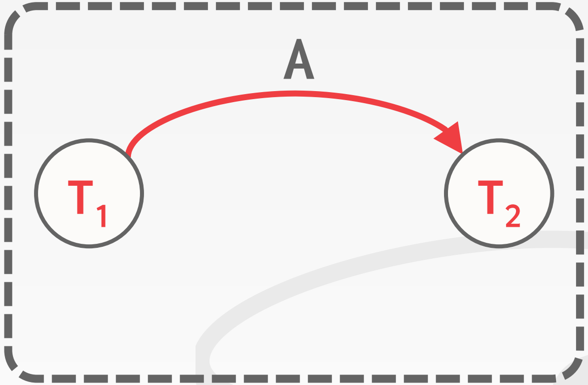
EXAMPLE #1

Schedule



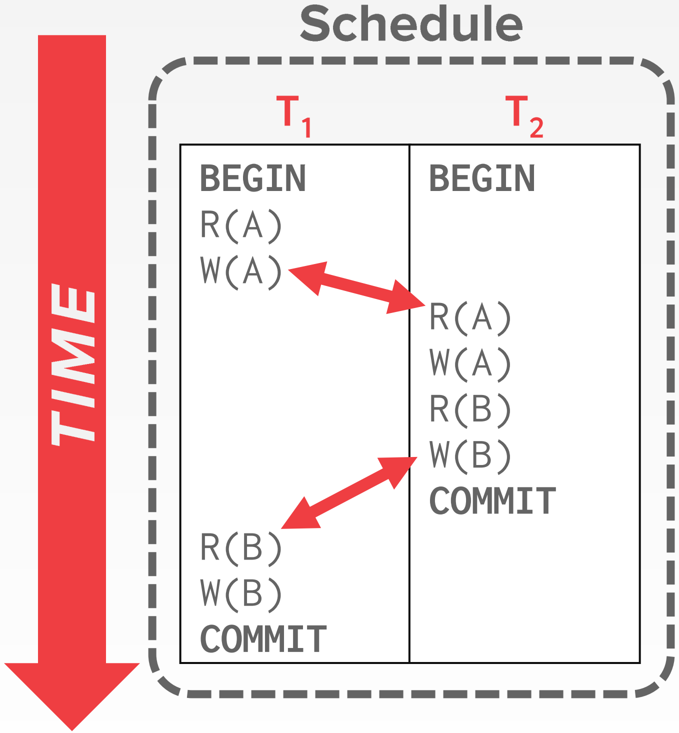
T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph



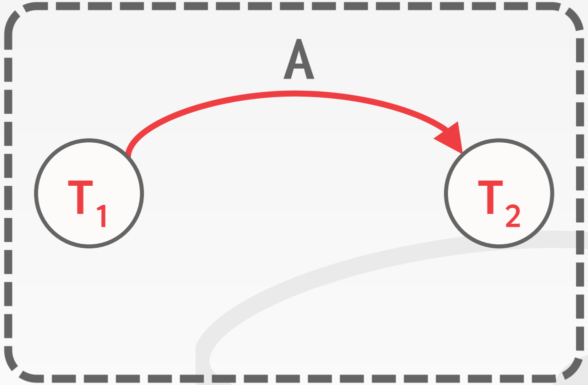
EXAMPLE #1

Schedule



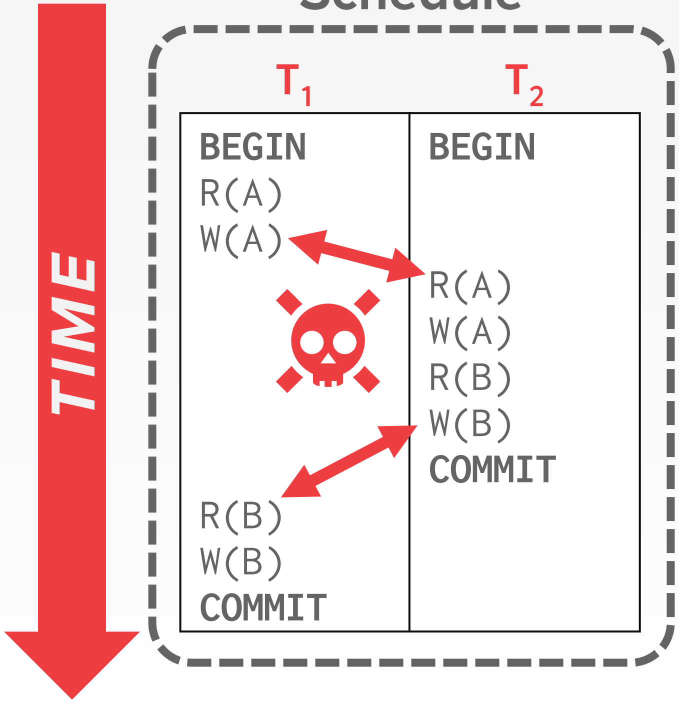
T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph



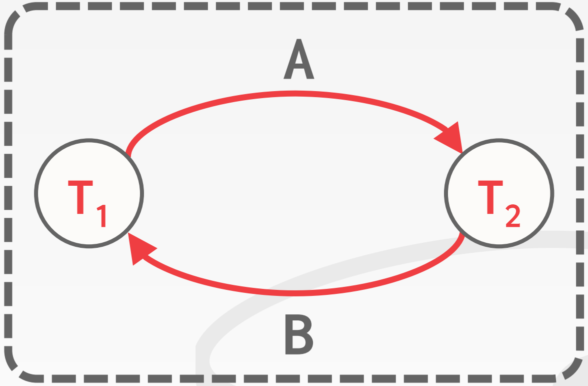
EXAMPLE #1

Schedule



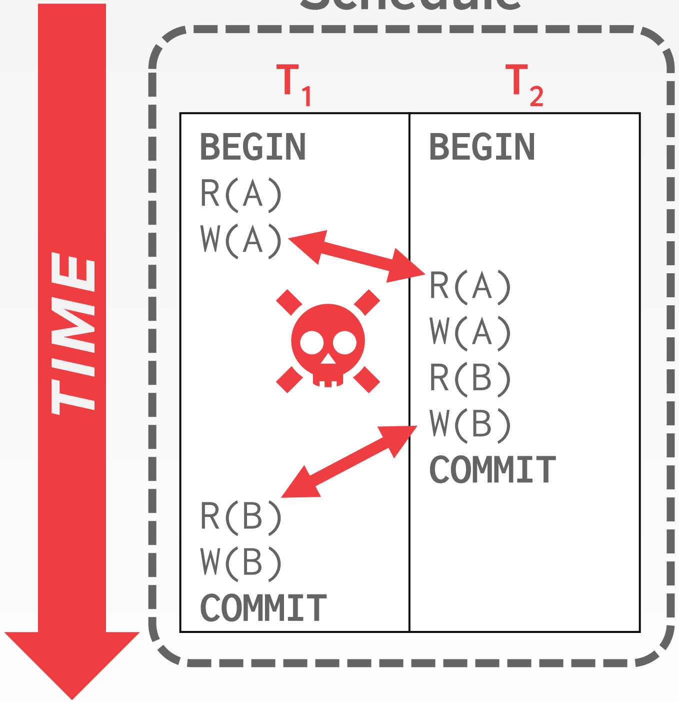
T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph



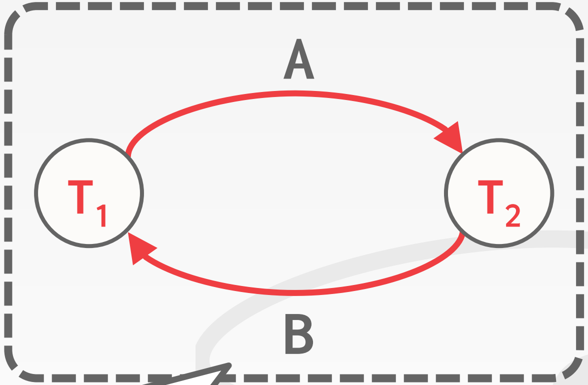
EXAMPLE #1

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

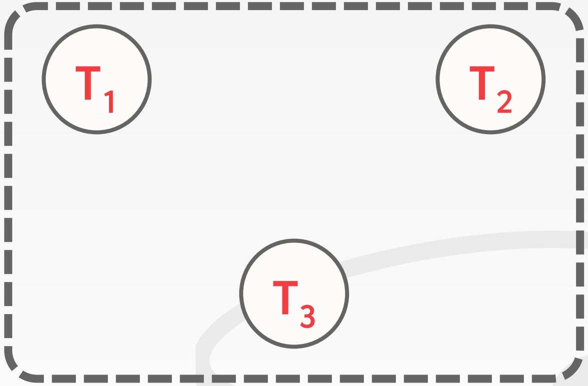
EXAMPLE #2 – THREESOME

Schedule

TIME

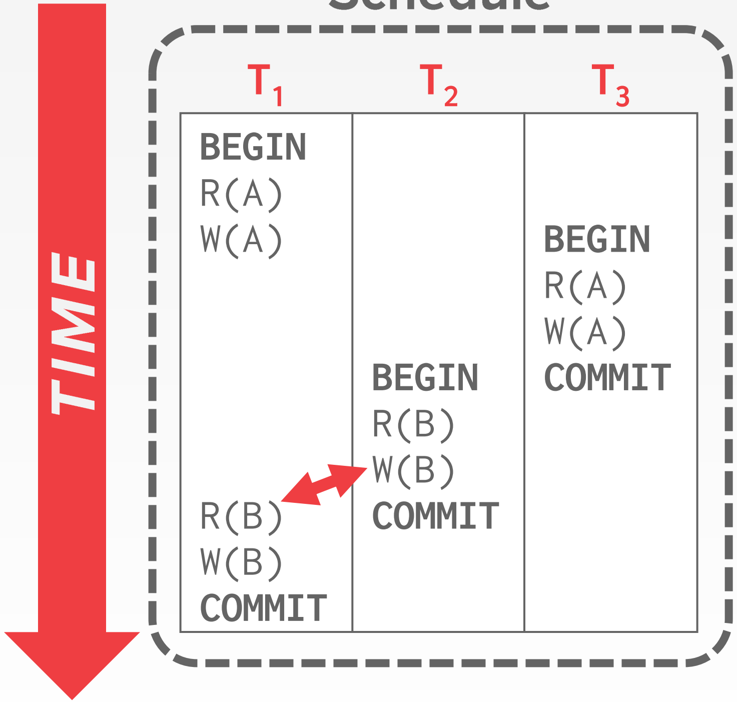
T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		

Dependency Graph



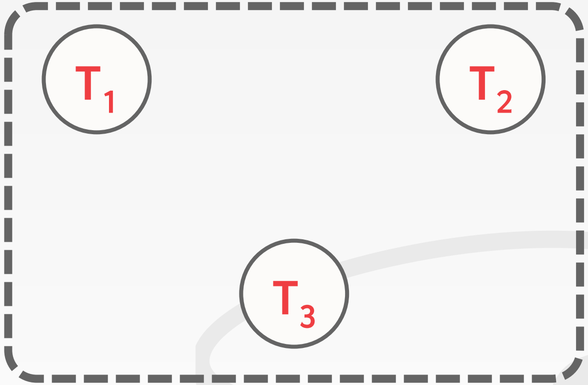
EXAMPLE #2 – THREESOME

Schedule



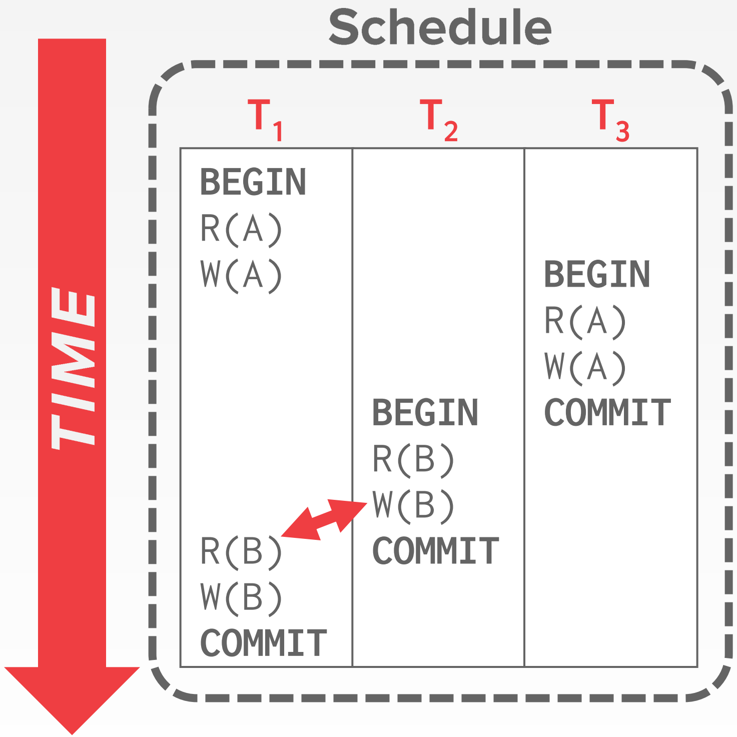
T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		
		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		

Dependency Graph



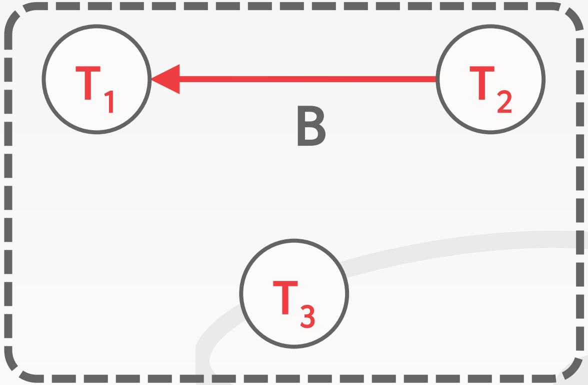
EXAMPLE #2 – THREESOME

Schedule



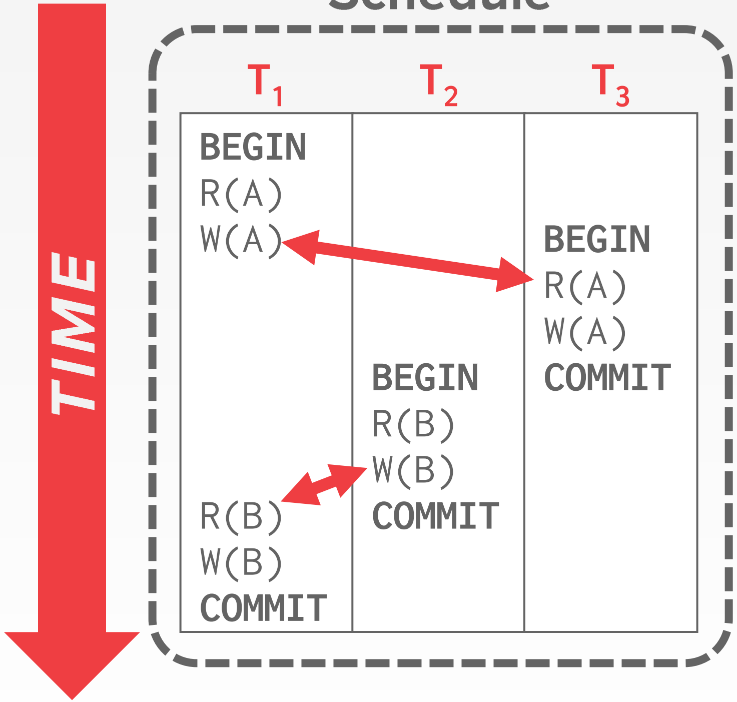
T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		

Dependency Graph



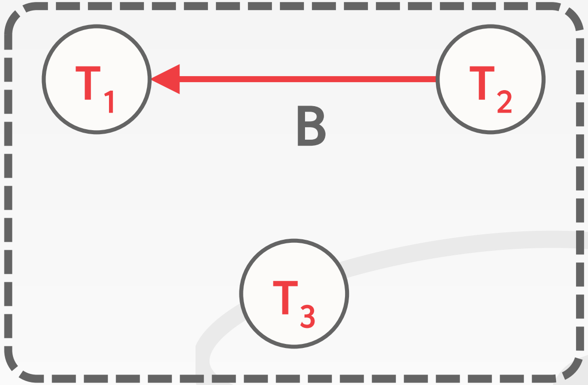
EXAMPLE #2 – THREESOME

Schedule



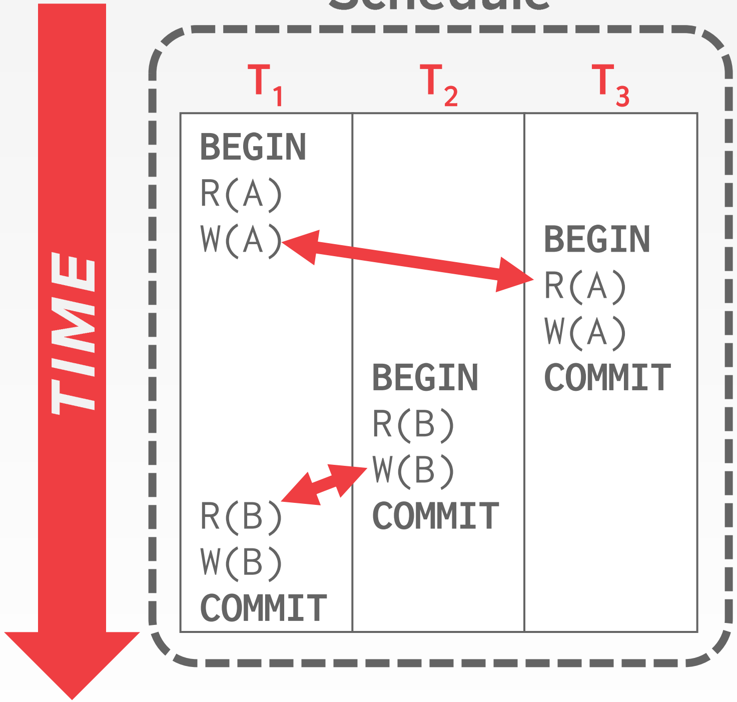
T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		

Dependency Graph



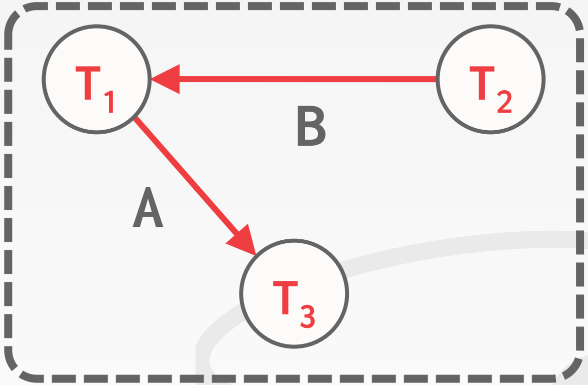
EXAMPLE #2 – THREESOME

Schedule



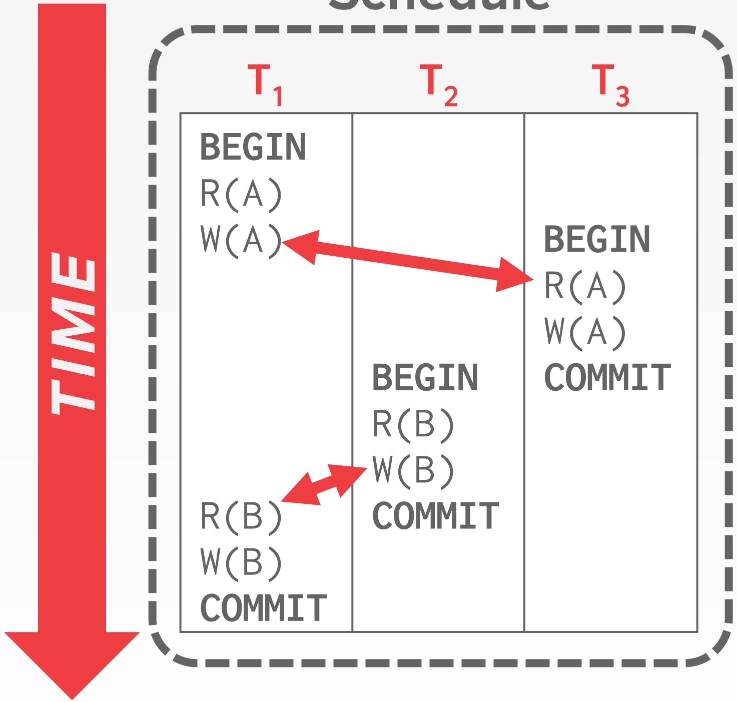
T_1	T_2	T_3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		

Dependency Graph



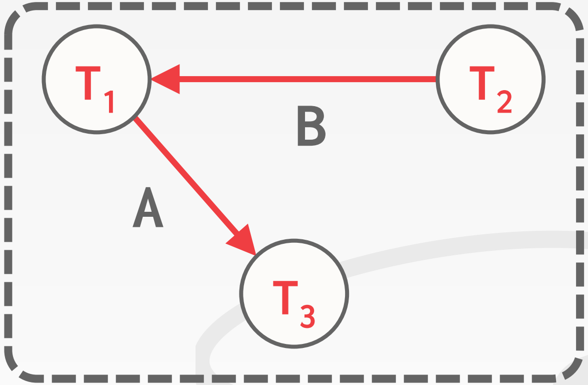
EXAMPLE #2 – THREESOME

Schedule



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		

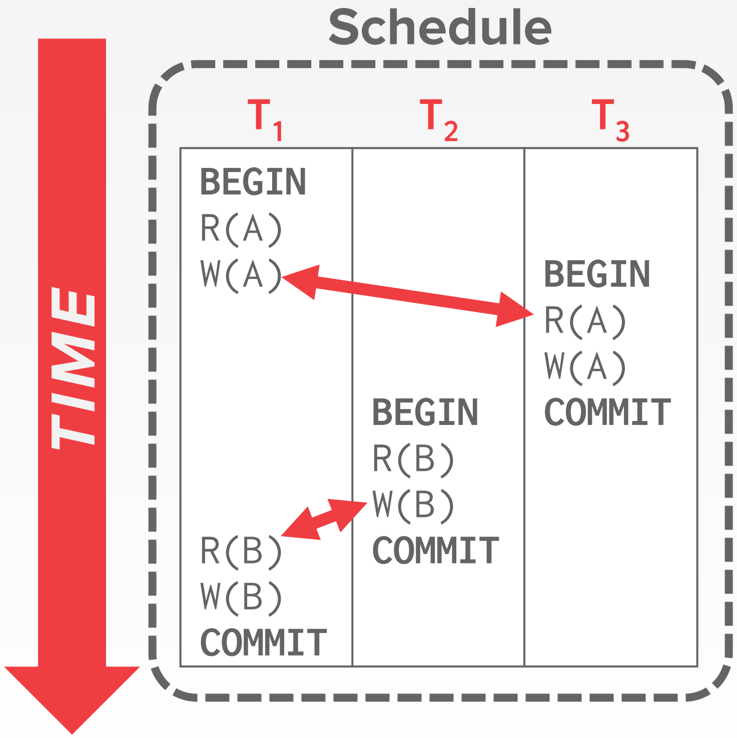
Dependency Graph



Is this equivalent to a serial execution?

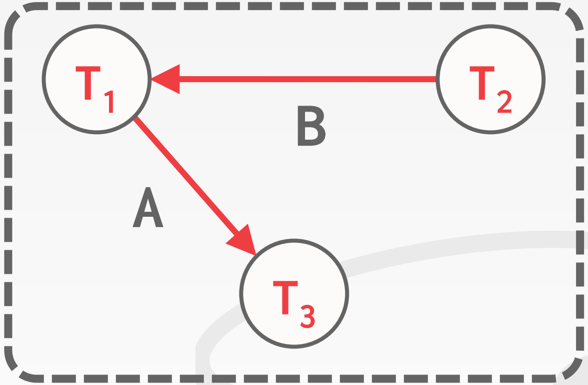
EXAMPLE #2 – THREESOME

Schedule



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)

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

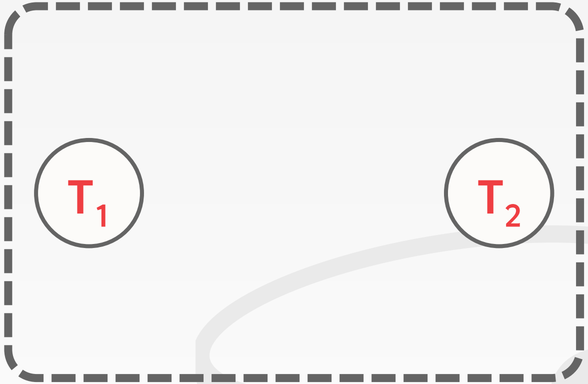
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

T_1	T_2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
R(B)	**COMMIT**
B = B+10	
W(B)	
COMMIT	

Dependency Graph



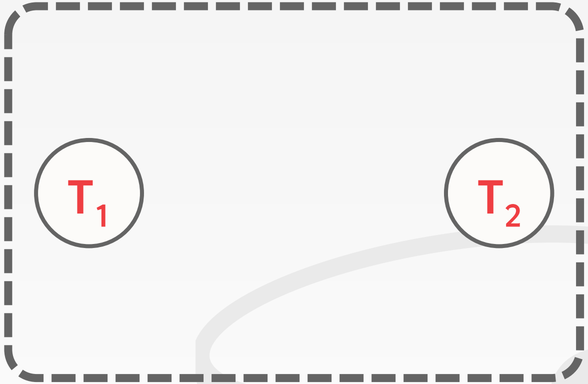
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



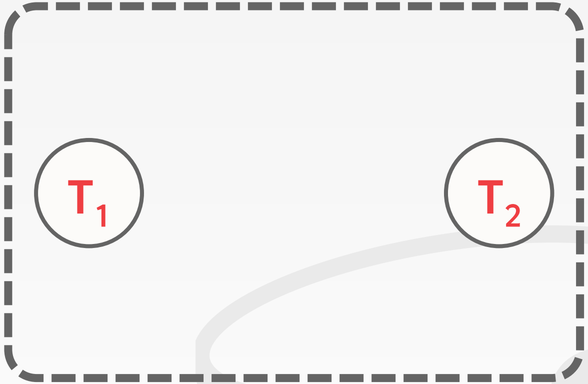
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



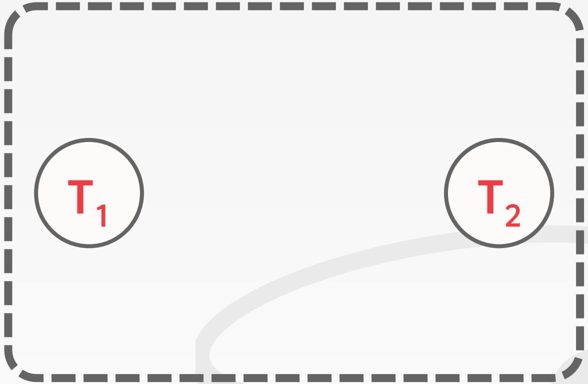
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
R(B)	COMMIT
B = B+10	
W(B)	
COMMIT	

Dependency Graph



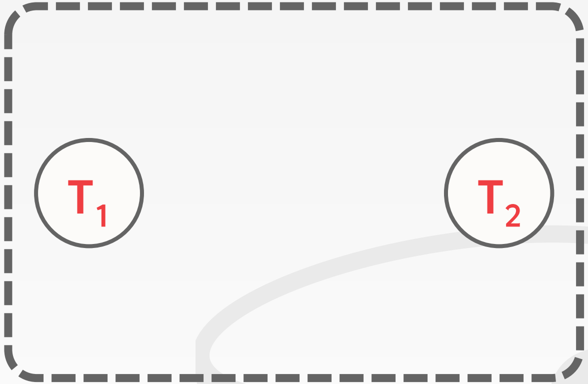
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

T_1	T_2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
R(B)	**COMMIT**
B = B+10	
W(B)	
COMMIT	

Dependency Graph



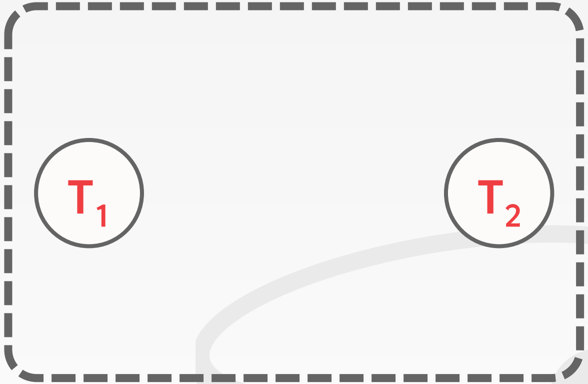
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

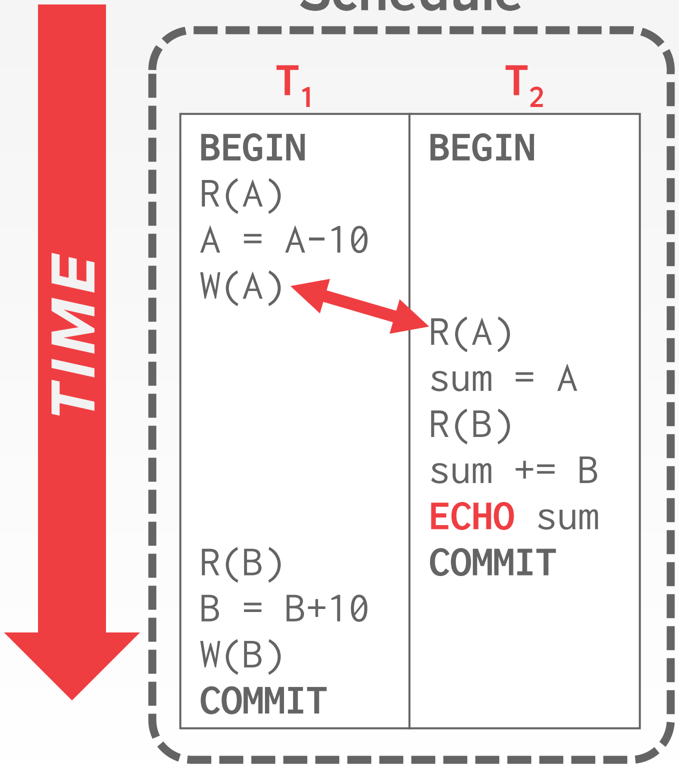
T_1	T_2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



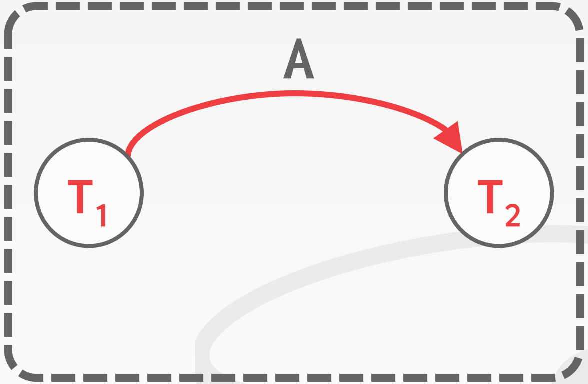
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule



T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



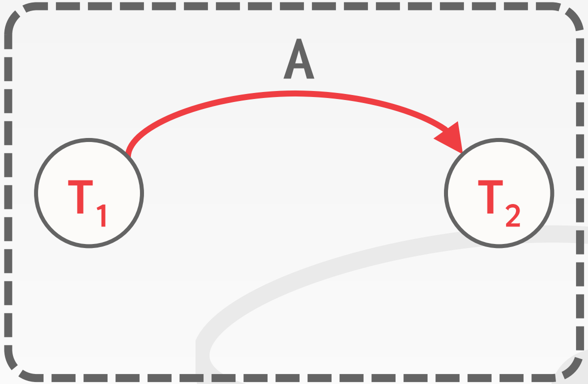
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule

TIME

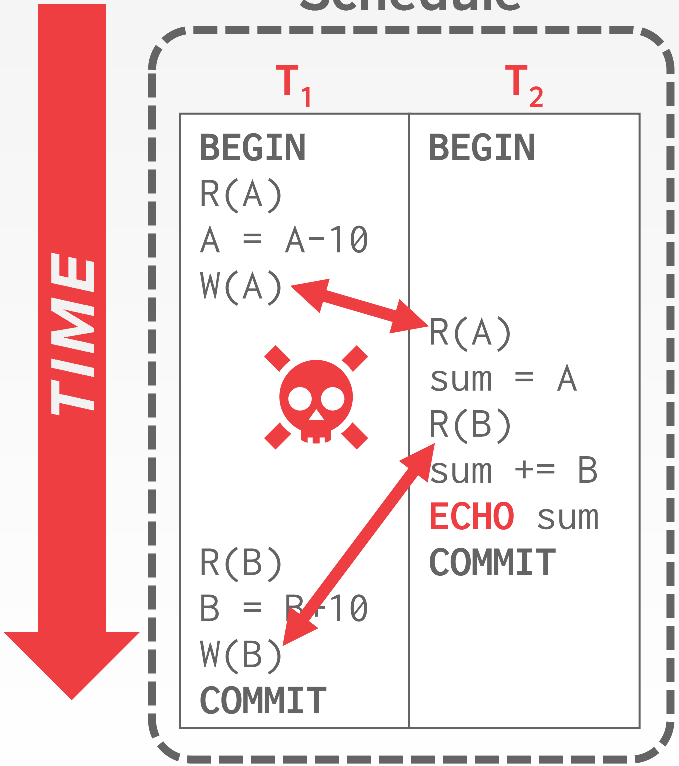
T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
R(B)	COMMIT
B = B+10	
W(B)	
COMMIT	

Dependency Graph



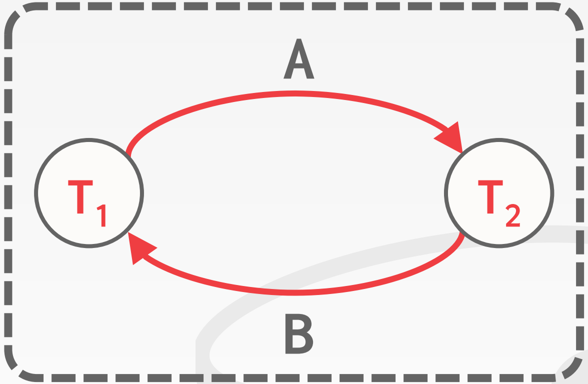
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule



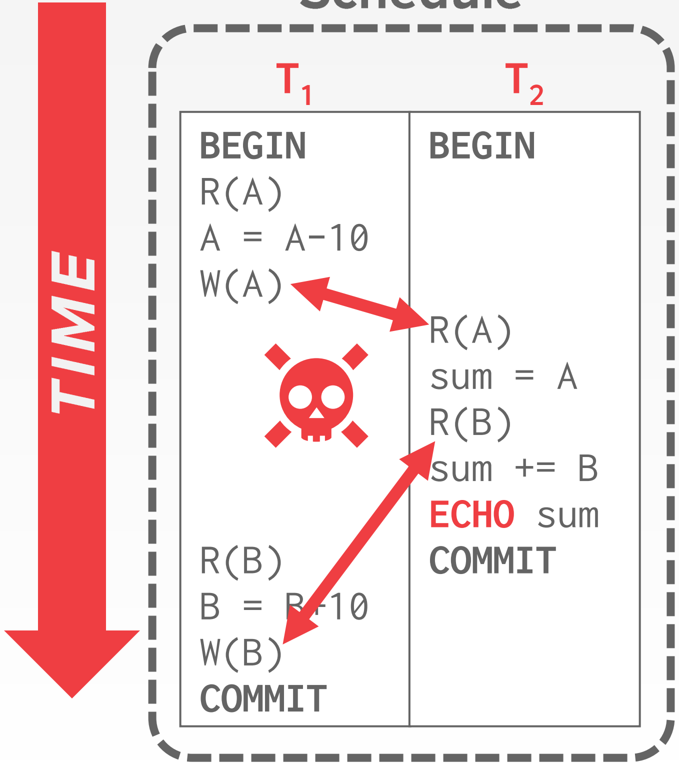
T1	T2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



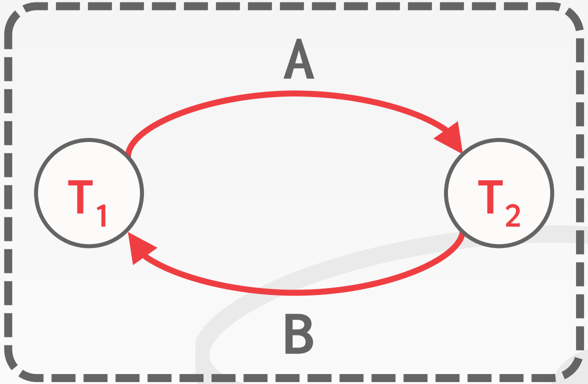
EXAMPLE #3 – INCONSISTENT ANALYSIS

Schedule



T_1	T_2
BEGIN	BEGIN
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO sum
	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?

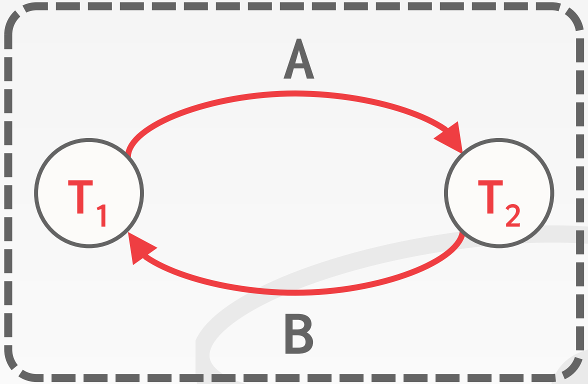
EXAMPLE #3 – INCONSISTENT ANALYSIS

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



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 (weaker) 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.

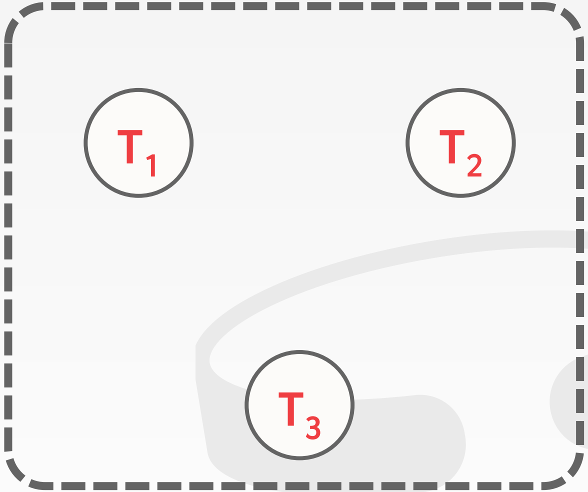
VIEW SERIALIZABILITY

Schedule

TIME

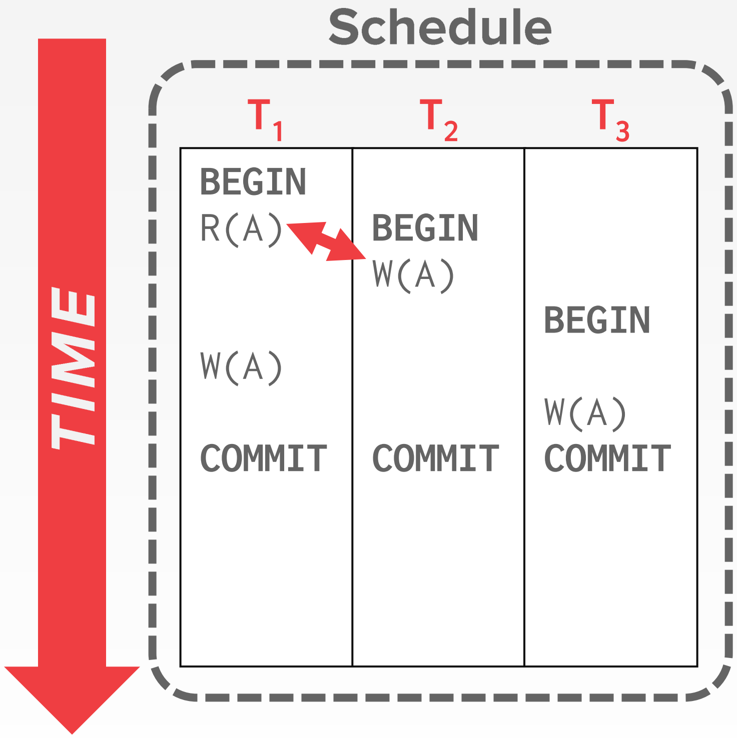
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph



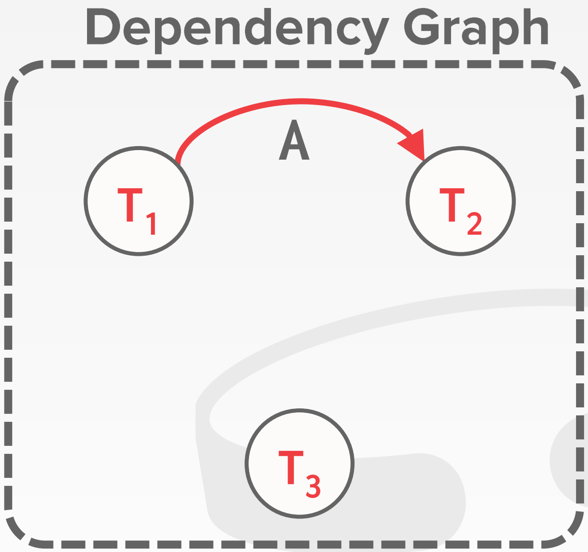
VIEW SERIALIZABILITY

Schedule



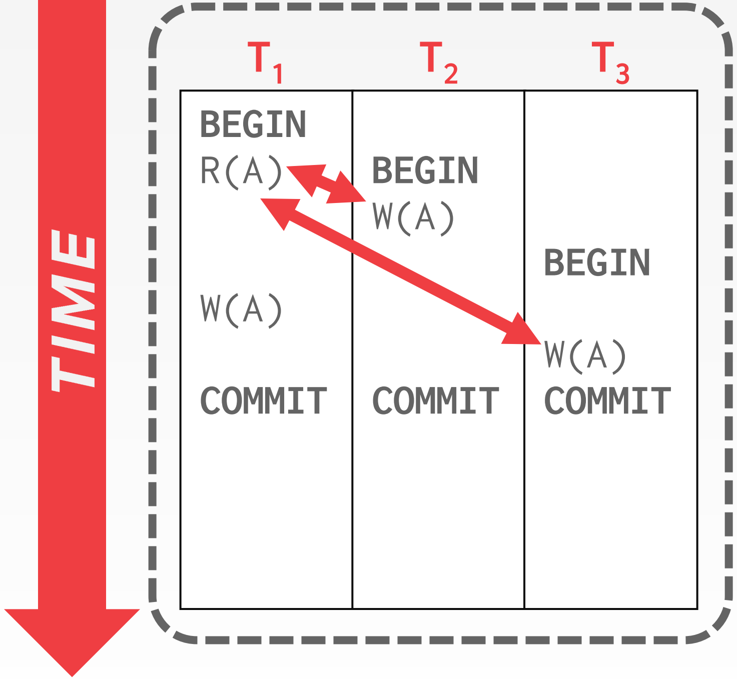
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph



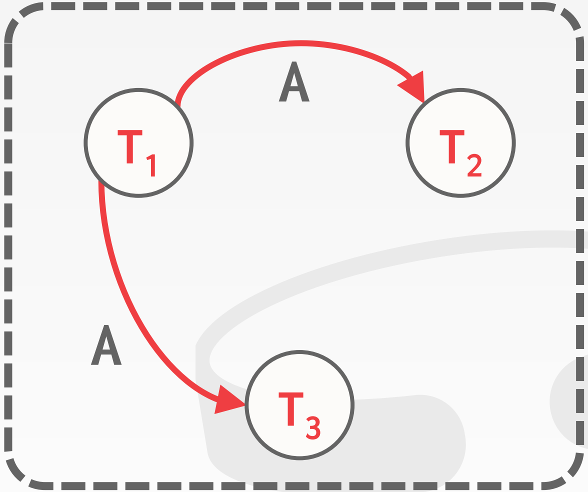
VIEW SERIALIZABILITY

Schedule



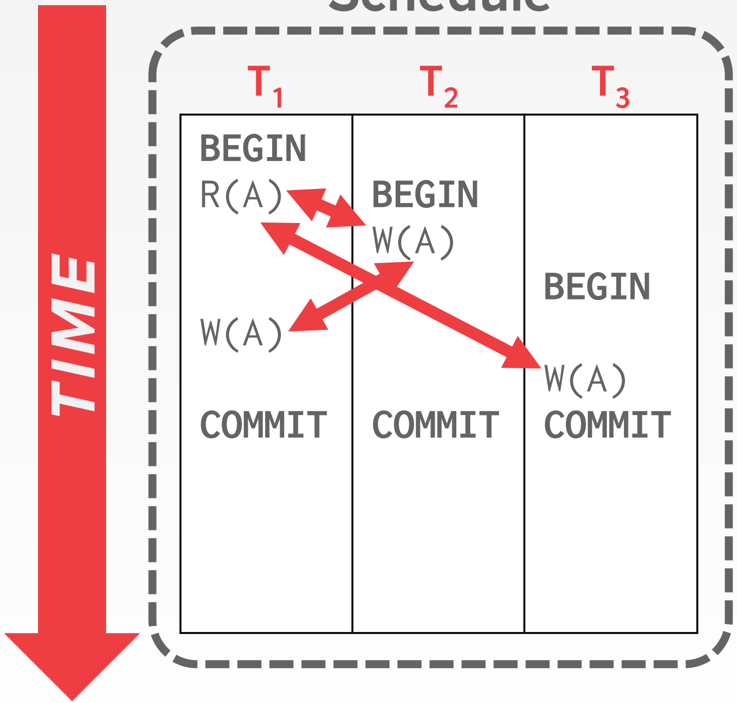
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph



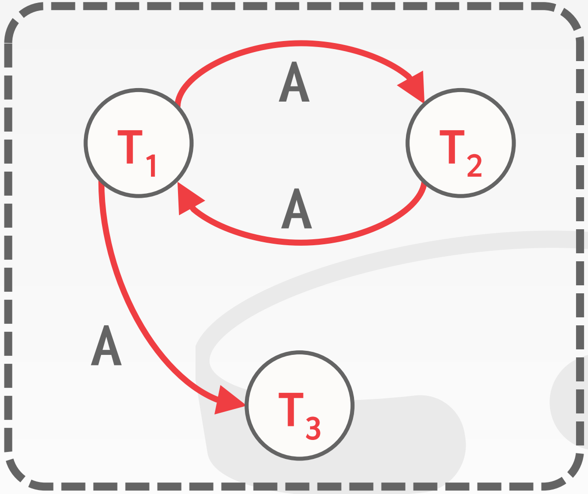
VIEW SERIALIZABILITY

Schedule



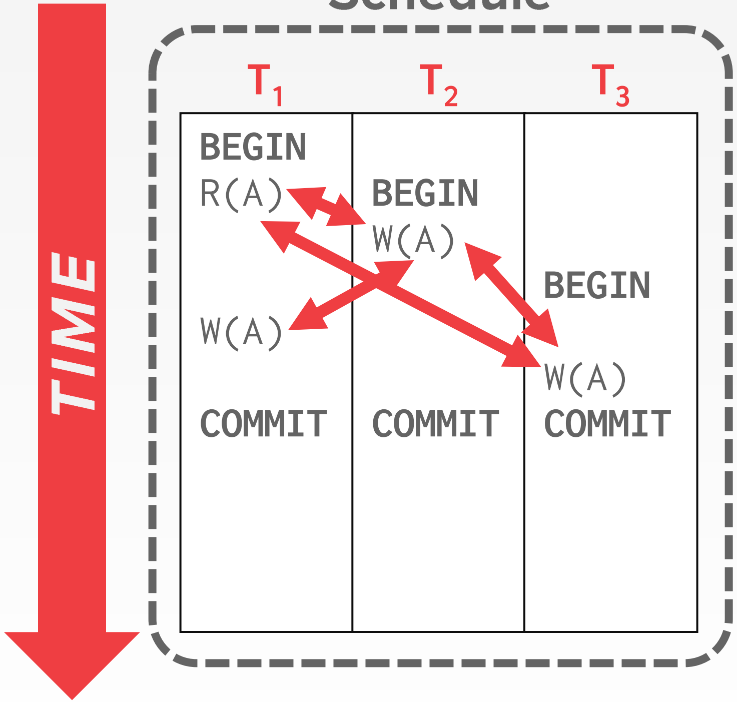
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph



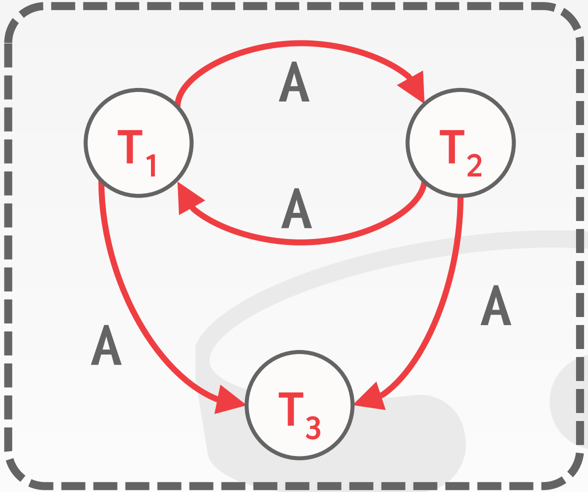
VIEW SERIALIZABILITY

Schedule



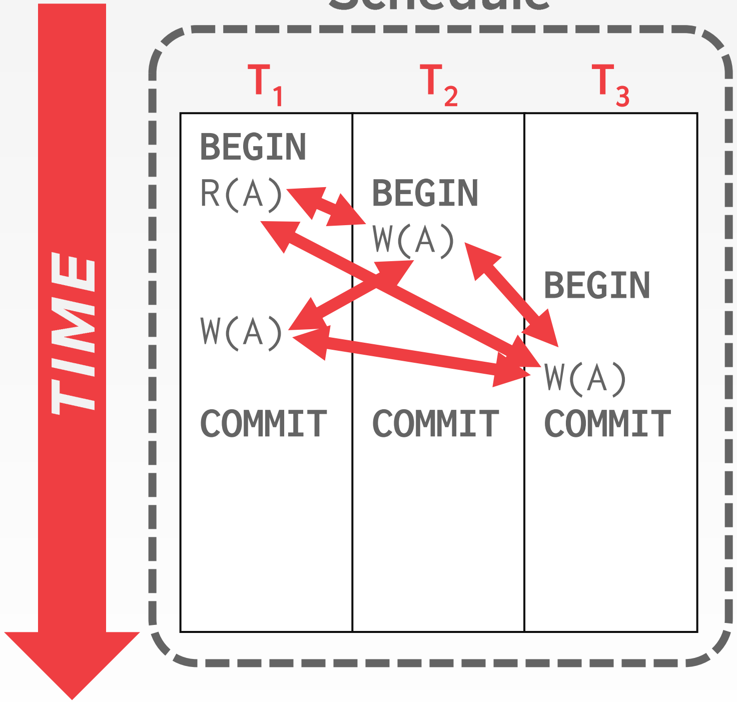
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph



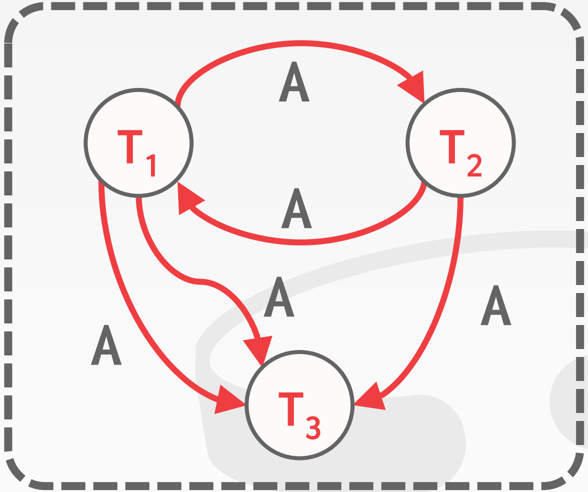
VIEW SERIALIZABILITY

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

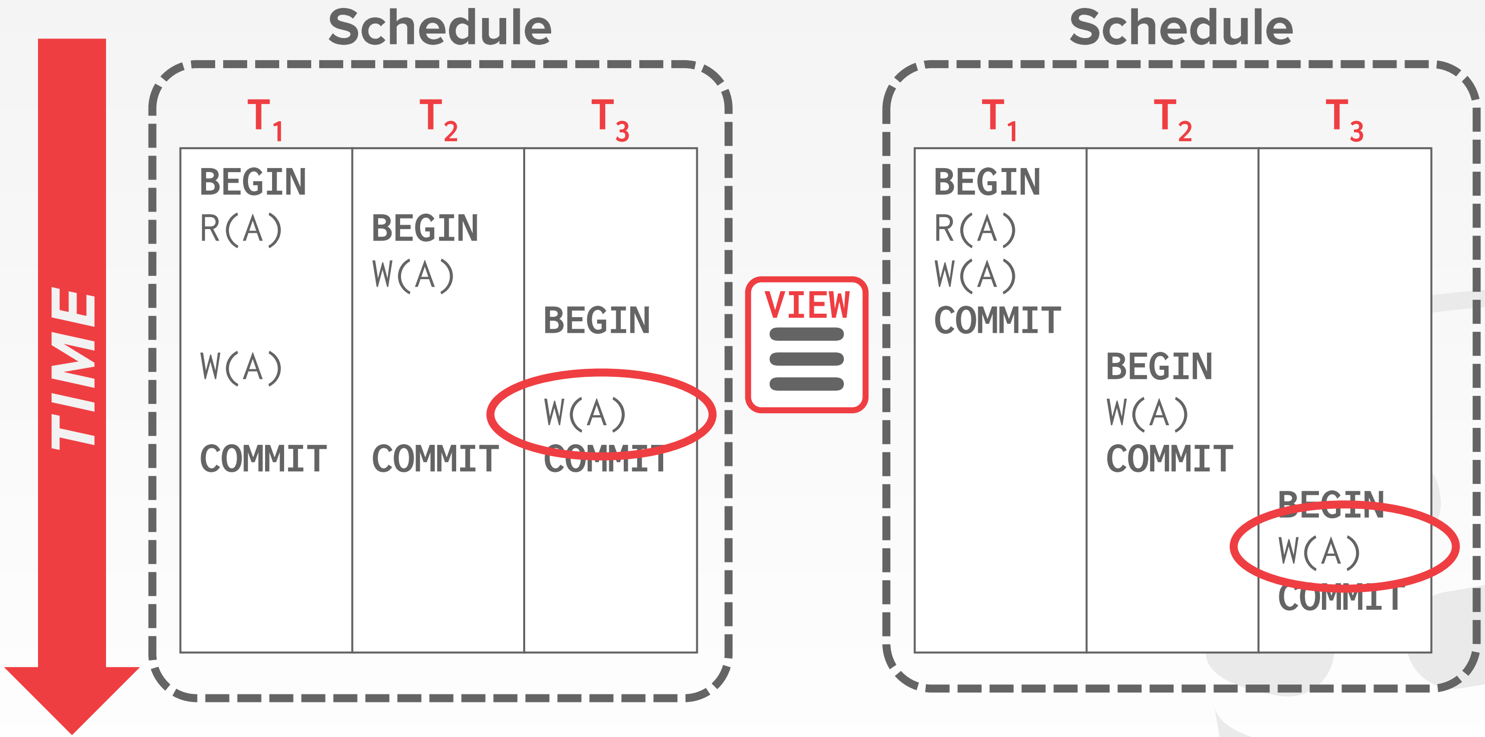
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

VIEW ≡

Schedule

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

VIEW SERIALIZABILITY



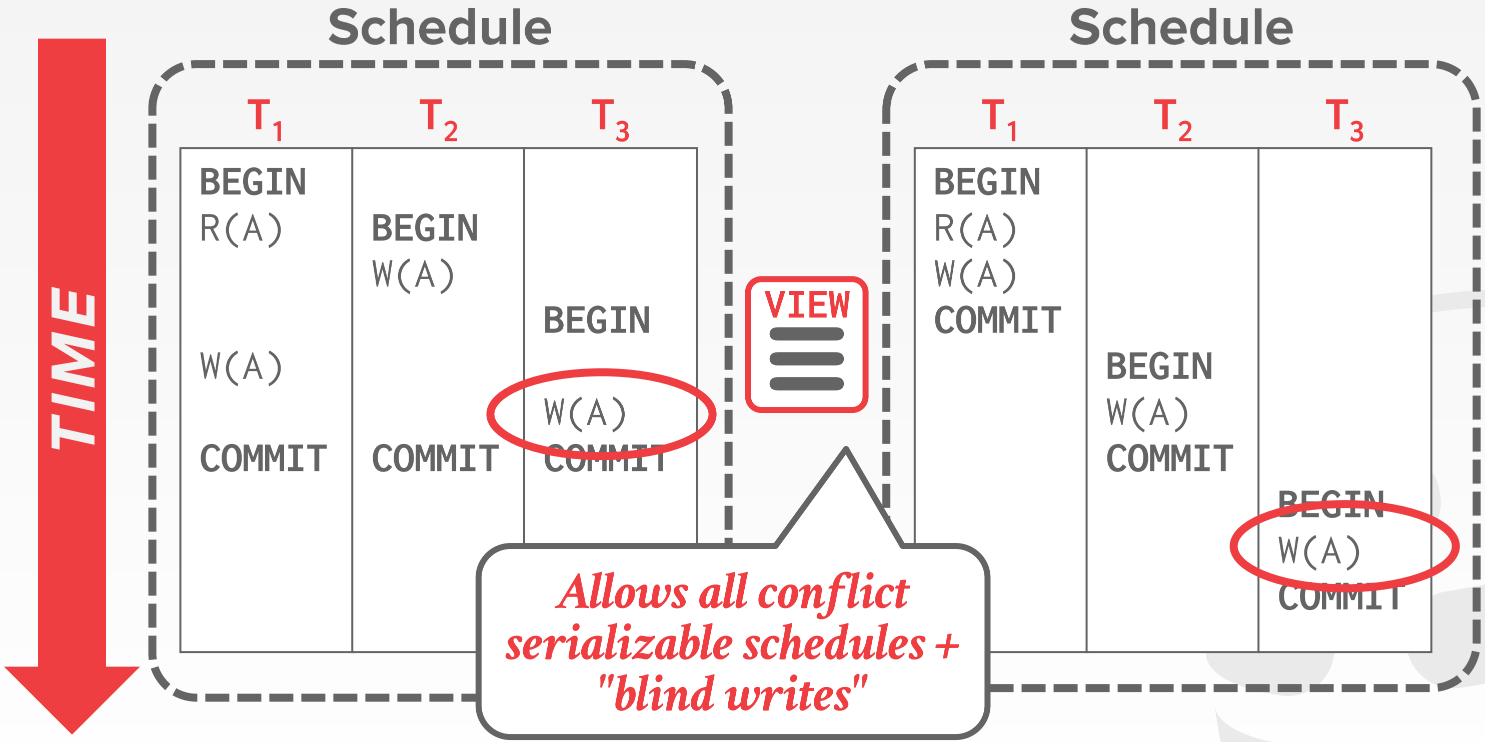
Schedule

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

Schedule

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

VIEW SERIALIZABILITY



Schedule

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

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 is difficult to enforce efficiently.

Neither definition allows all schedules that you would consider "serializable".

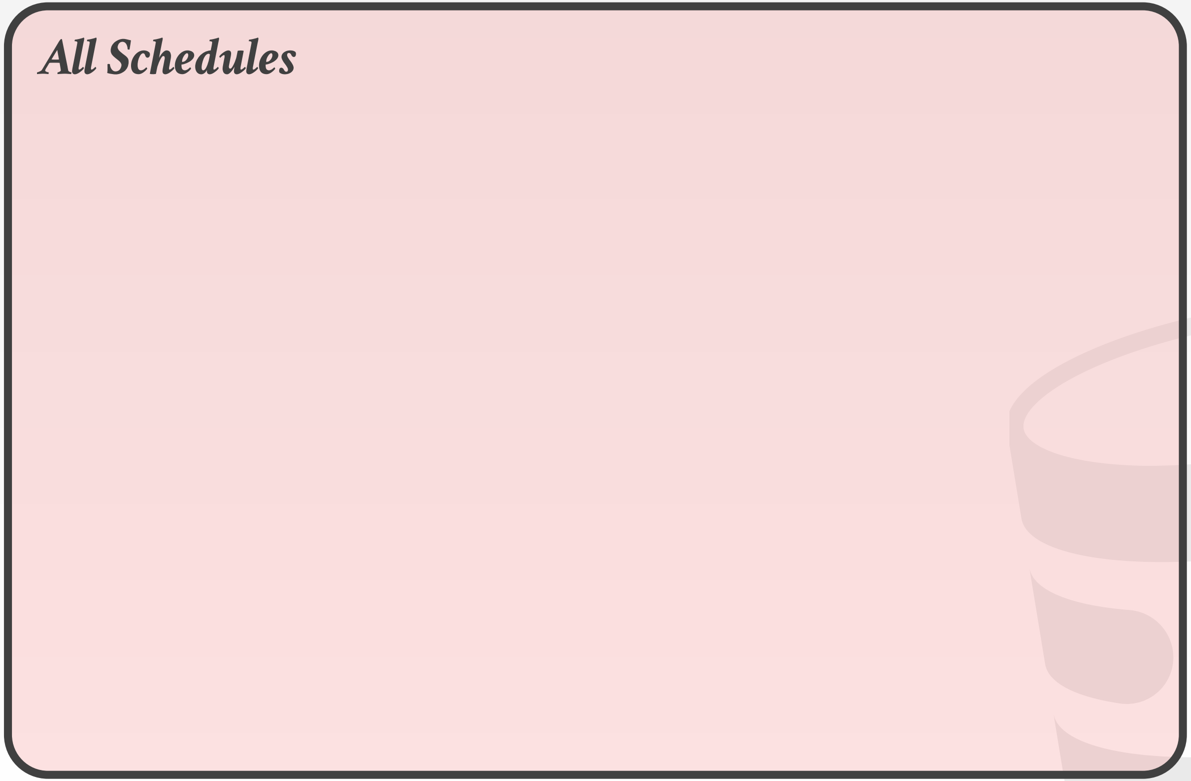
→ This is because they don't understand the meanings of the operations or the data (recall example #3)

SERIALIZABILITY

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

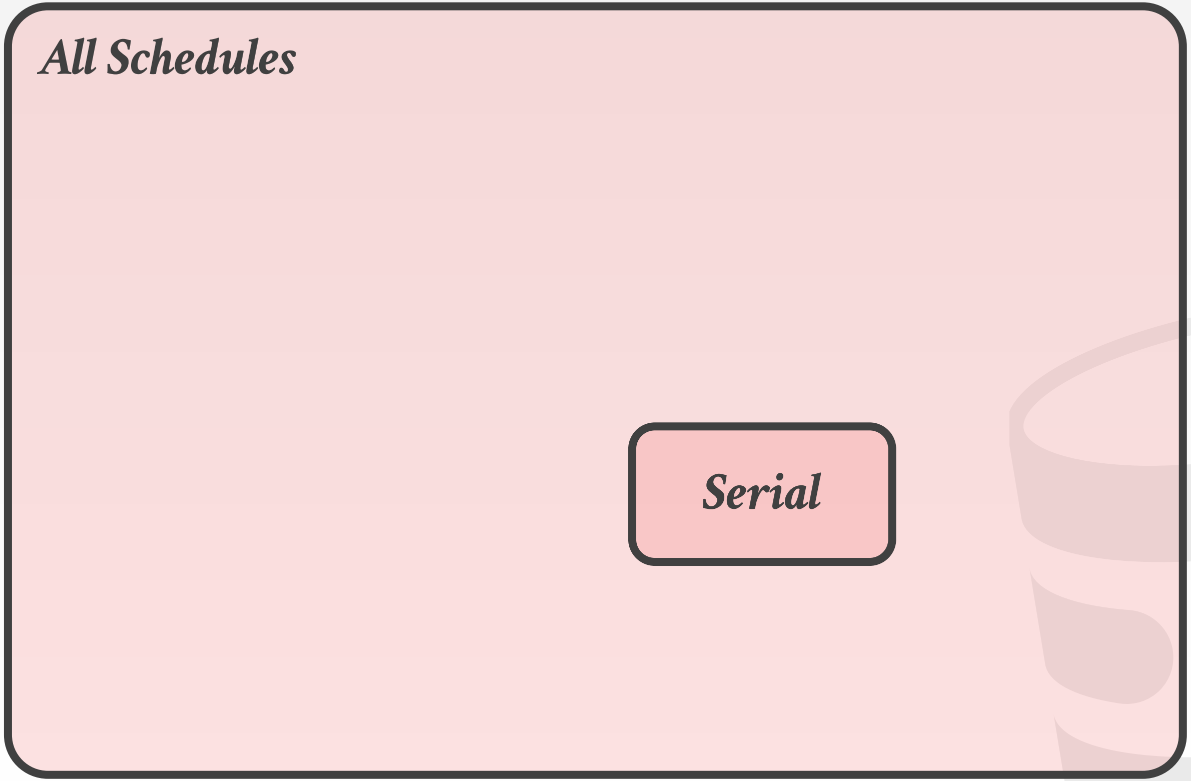
To allow more concurrency, some special cases get handled separately at the application level.

UNIVERSE OF SCHEDULES

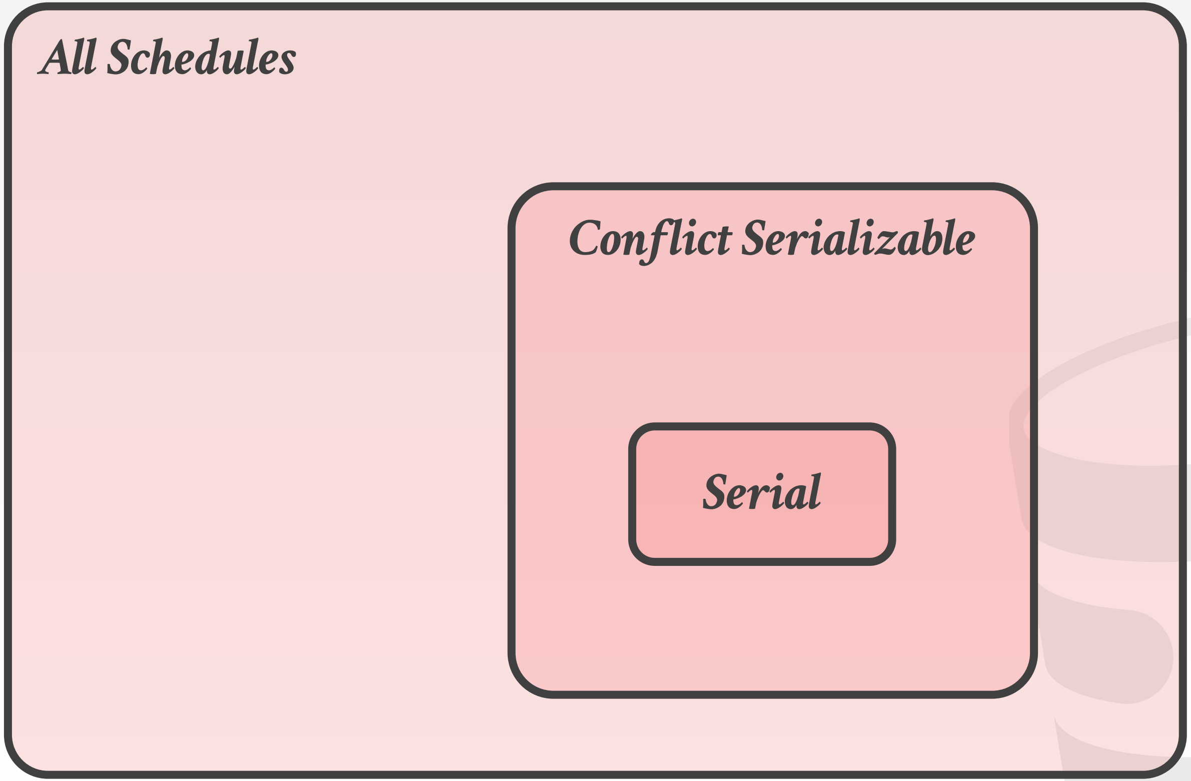


All Schedules

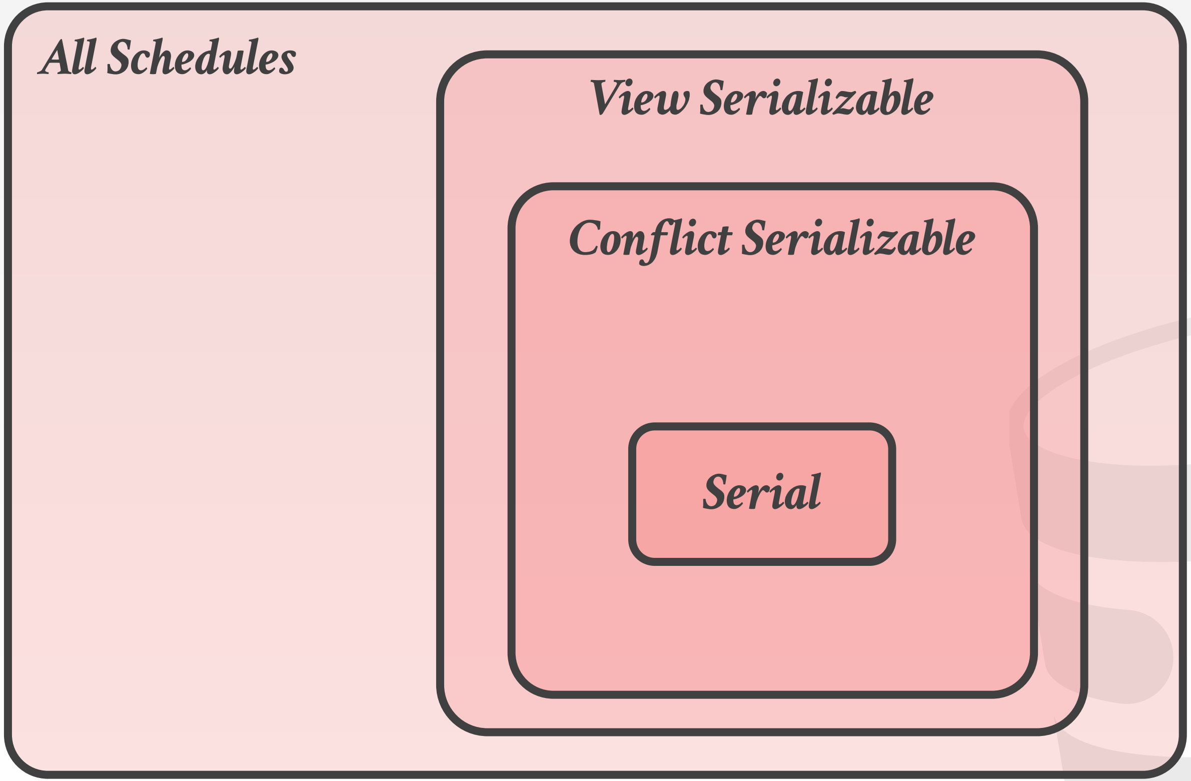
UNIVERSE OF SCHEDULES



UNIVERSE OF SCHEDULES



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.

ACID PROPERTIES

Atomicity: All actions in the txn happen, or none happen.

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent.

Isolation: Execution of one txn is isolated from that of other txns.

Durability: If a txn commits, its effects persist.

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.

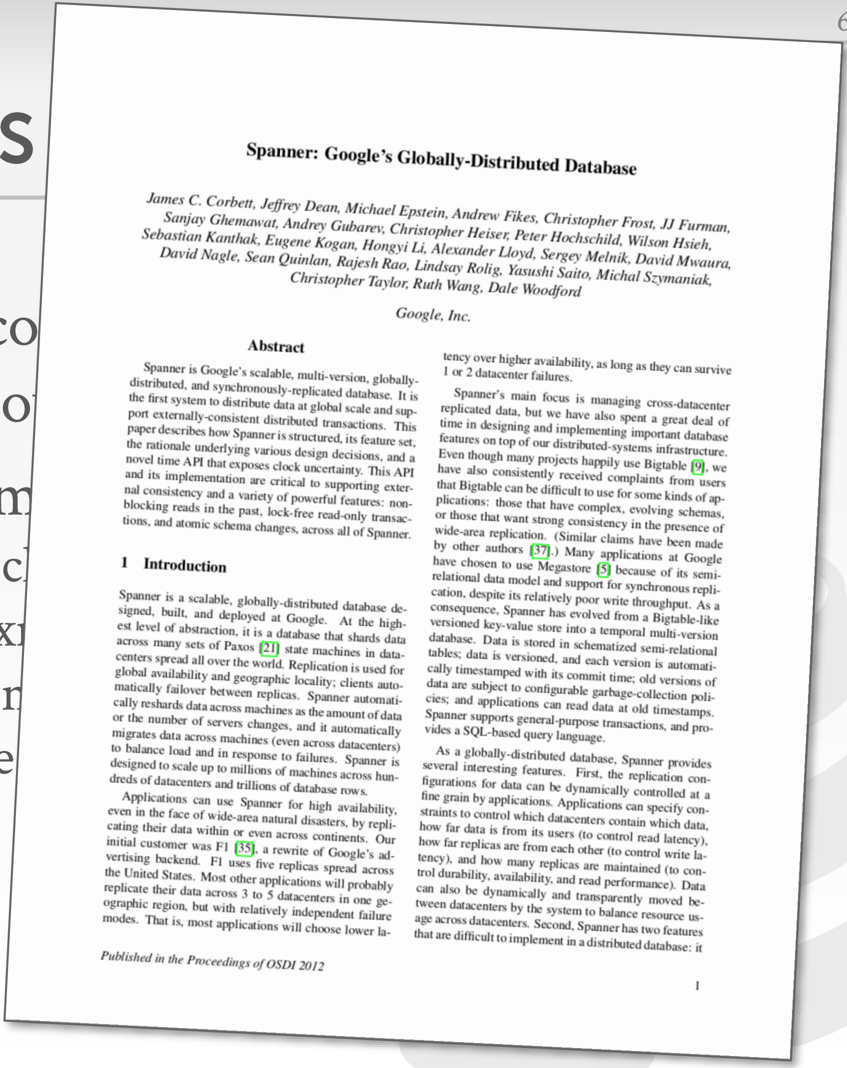
CONCLUS

Concurrency control and reco
most important functions pro

Concurrency control is autom

→ System automatically inserts loc
schedules actions of different tx

→ Ensures that resulting execution
executing the txns one after the



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: nonblocking reads in the past, lock-free read-only transactions, and atomic schema changes, across all of Spanner.

1 Introduction

Spanner is a scalable, globally-distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [21] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [35], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency 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 replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in schematized semi-relational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides a SQL-based query language.

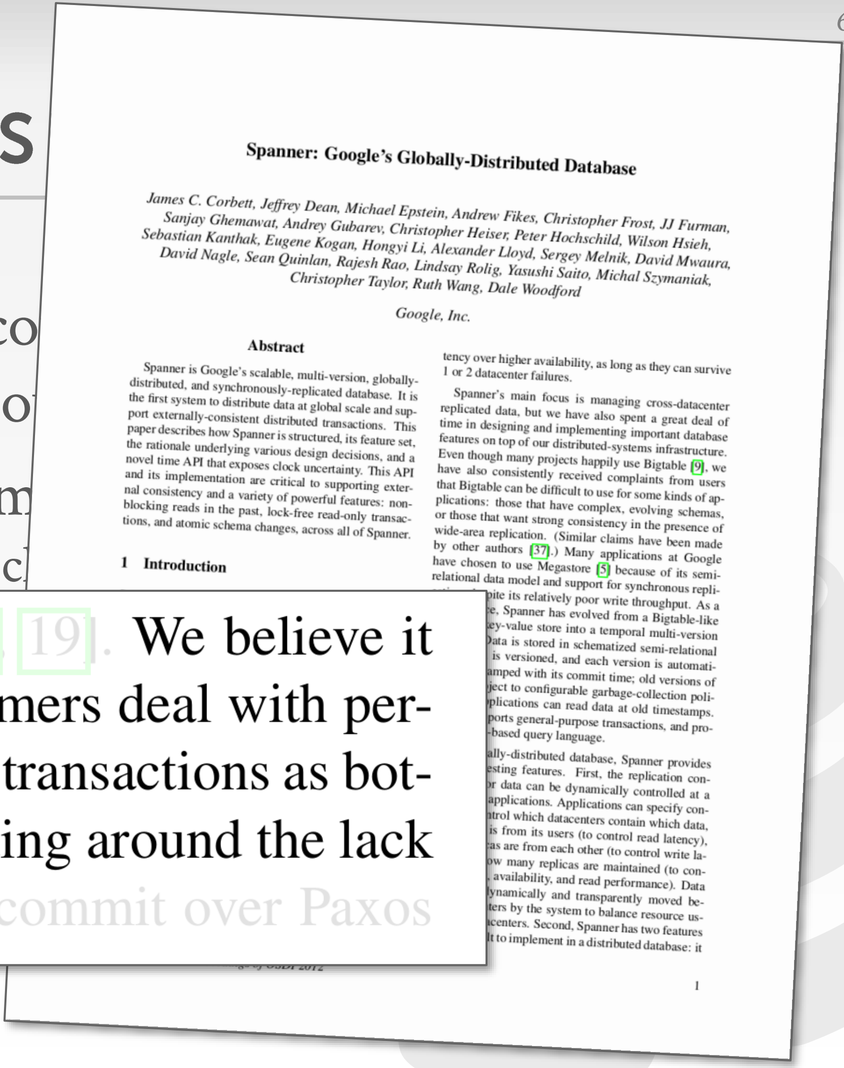
As a globally-distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it

Published in the Proceedings of OSDI 2012

CONCLUS

Concurrency control and reco
most important functions pro

Concurrency control is autom
→ System automatically inserts loc



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: nonblocking 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

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