

Carnegie Mellon University

# Database Systems

## Concurrency Control Theory



15-445/645 FALL 2024 » PROF. ANDY PAVLO

# ADMINISTRIVIA

---

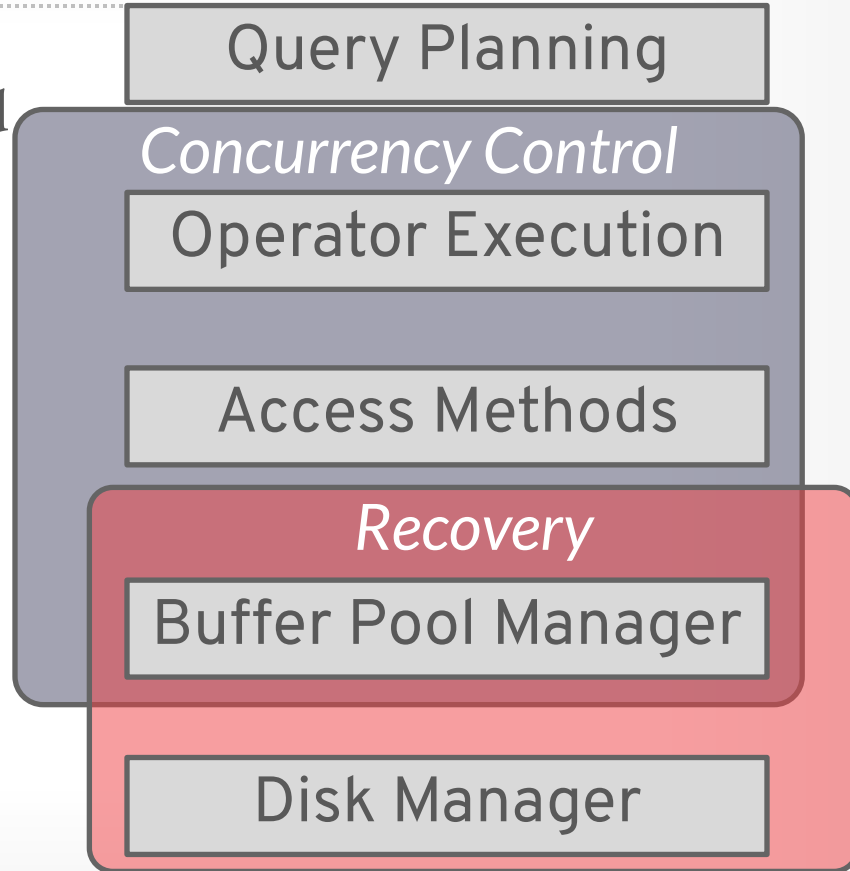
**Project #3** is due Sunday Nov 17th @ 11:59pm

→ **Recitation: Monday Nov 4<sup>th</sup> @ 8:00pm (Zoom)**

**Homework #4** is due Sunday Nov 3<sup>rd</sup> @ 11:59pm

# COURSE STATUS

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



# MOTIVATION EXAMPLE #1

## *Application Logic*



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

Read Balance: \$100



Bank Balance : \$100

# MOTIVATION EXAMPLE #1

## *Application Logic*

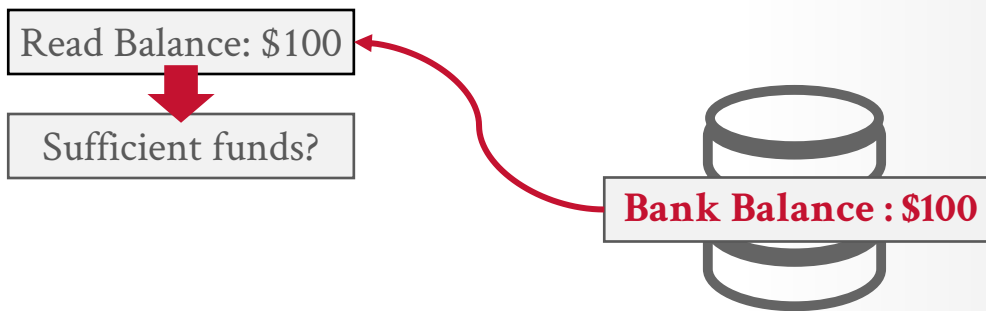
Read(A);

➔ Check(A > \$25);

Pay(\$25);


A = A - \$25;

Write(A);

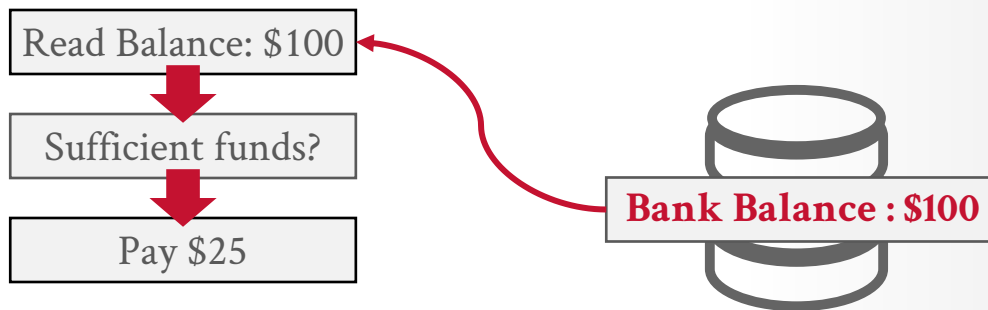


# MOTIVATION EXAMPLE #1

## *Application Logic*



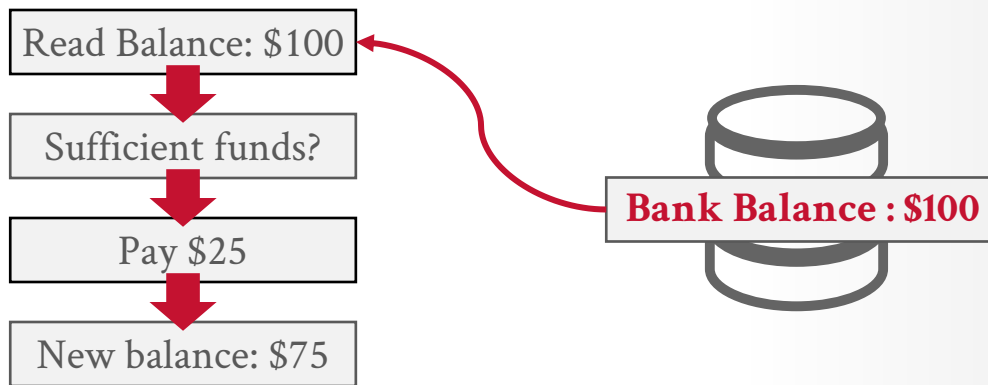
```
Read(A);  
Check(A > $25);  
Pay($25);  
A = A - $25;  
Write(A);
```



# MOTIVATION EXAMPLE #1

## *Application Logic*

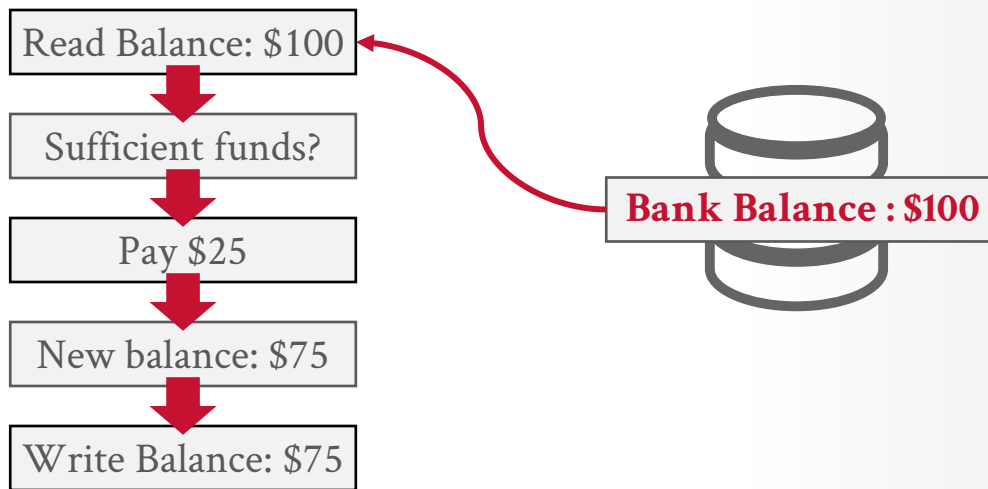
```
Read(A);  
Check(A > $25);  
Pay($25);  
➔ A = A - $25;  
Write(A);
```



# MOTIVATION EXAMPLE #1

## *Application Logic*

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

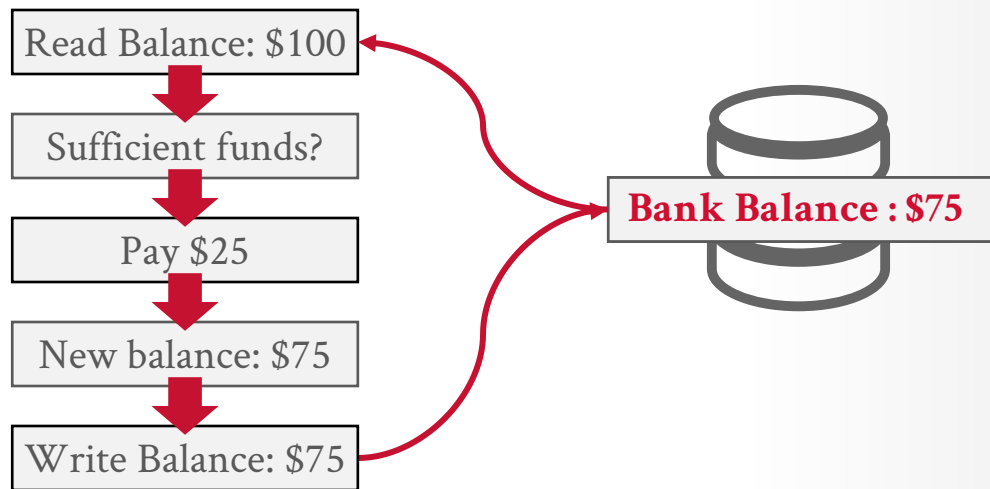




# MOTIVATION EXAMPLE #1

## *Application Logic*

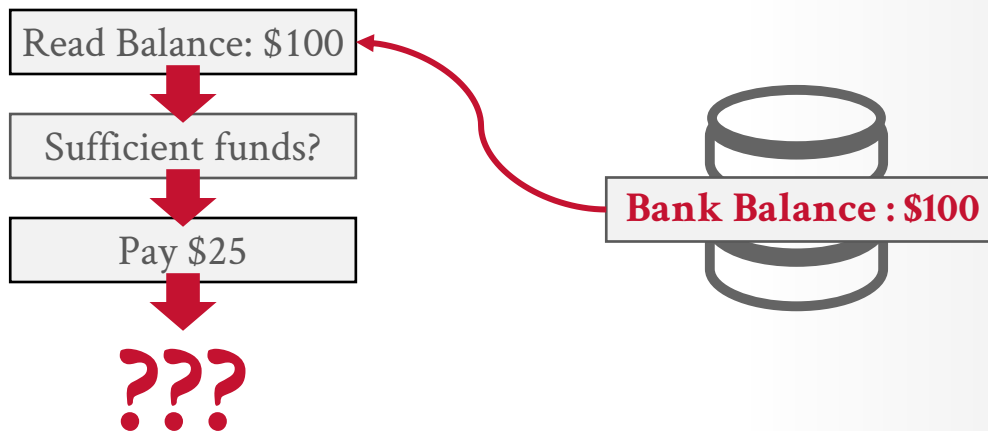
```
Read(A);  
Check(A > $25);  
Pay($25);  
A = A - $25;  
➔ Write(A);
```



# MOTIVATION EXAMPLE #1

## *Application Logic*

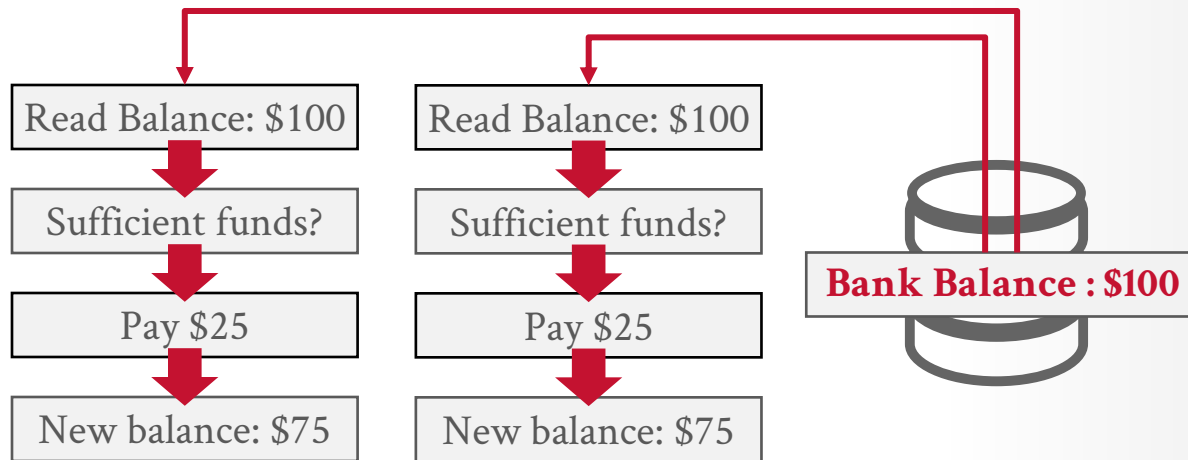
```
Read(A);  
Check(A > $25);  
Pay($25);
```



# MOTIVATION EXAMPLE #2

## Application Logic

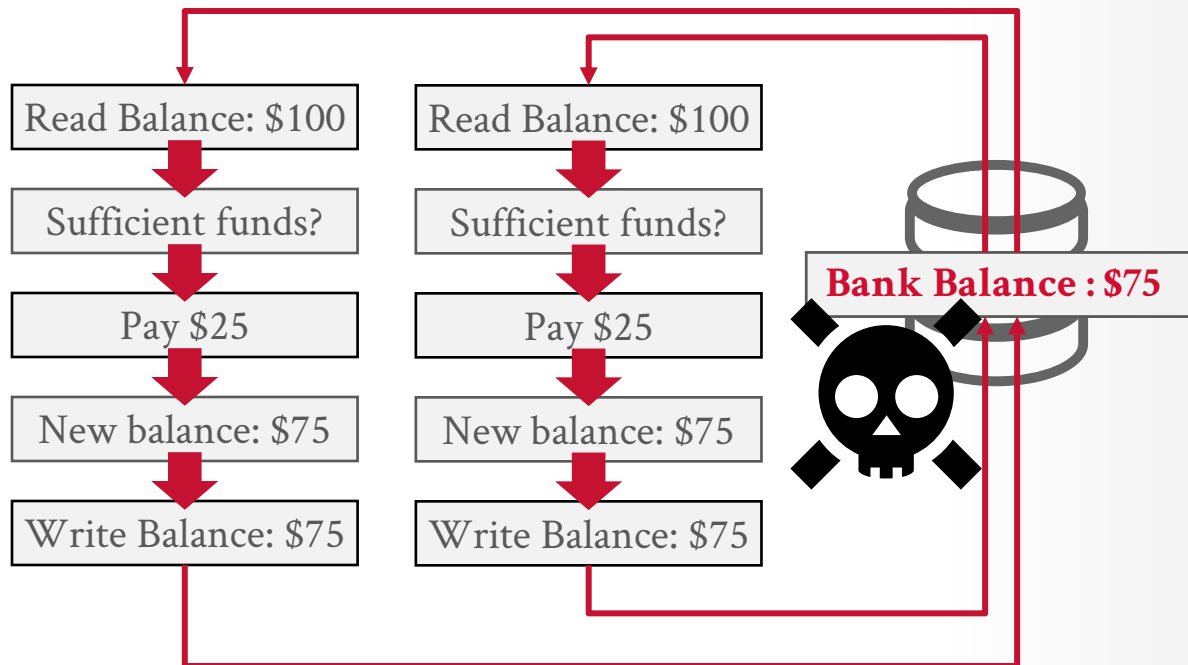
```
Read(A);  
Check(A > $25);  
Pay($25);  
➔ A = A - $25;  
Write(A);
```



# MOTIVATION EXAMPLE #2

## Application Logic

**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. shadow page

→ If the txn completes successfully, overwrite the original file with the new one.

→ If the txn fails, just remove the dirty copy.

# PROBLEM STATEMENT

---

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

## **Why do we want that?**

- Better utilization/throughput
- Increased response times to users.

## **But we also would like:**

- Correctness
- Fairness

# PROBLEM STATEMENT

---

Arbitrary interleaving of operations can lead to:

- Temporary Inconsistency (ok, unavoidable)
- Permanent Inconsistency (bad!)

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

- Changes to the “outside world” are beyond the scope of the DBMS.

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

# FORMAL DEFINITIONS

---

**Database:** A 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 next week.

**Transaction:** A sequence of read and write operations (e.g., **R(A)**, **W(B)**, ...)

- DBMS's abstract view of a user program.
- A new txn starts with the **BEGIN** command.
- The txn stops with either **COMMIT** or **ROLLBACK**



# CORRECTNESS CRITERIA: ACID

---

<b><u>A</u>tomicity</b>	All actions in txn happen, or none happen. <i>“All or nothing...”</i>
<b><u>C</u>onsistency</b>	If each txn is consistent and the DB starts consistent, then it ends up consistent. <i>“It looks correct to me...”</i>
<b><u>I</u>solation</b>	Execution of one txn is isolated from that of other txns. <i>“All by myself...”</i>
<b><u>D</u>urability</b>	If a txn commits, its effects persist. <i>“I will survive...”</i>

# CORRECTNESS CRITERIA: ACID

---

<b><u>A</u>tomicity</b>	All actions in txn happen, or none happen. <i>“All or nothing...”</i>
<b><u>C</u>onsistency</b>	If each txn is consistent and the DB starts consistent, then it ends up consistent. <i>“It looks correct to me...”</i>
<b><u>I</u>solation</b>	Execution of one txn is isolated from that of other txns. <i>“All by myself...”</i>
<b><u>D</u>urability</b>	If a txn commits, its effects persist. <i>“I will survive...”</i>

# TODAY'S AGENDA

---

Atomicity

Consistency

Isolation

Durability

DB Flash Talk: **Firebolt**

# ATOMICITY OF TRANSACTIONS

---

Two possible outcomes of executing a txn:

- Commit after completing all its actions.
- Abort (or be aborted by the DBMS) after executing some actions.

DBMS guarantees that txns are **atomic**.

- From user's point of view: txn always either executes all its actions or executes no actions at all.

# MECHANISMS FOR ENSURING ATOMICITY

---

## Approach #1: Logging

- DBMS logs all actions so that it can undo the actions of aborted transactions.
- Maintain undo records both in memory and on disk.
- Think of this like the black box in airplanes...

Logging is used by almost every DBMS.

- Audit Trail
- Efficiency Reasons

# MECHANISMS FOR ENSURING ATOMICITY



*Don't  
Do This!*

## Approach #2: Shadow Paging

→ DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits is the page made visible to others.

→ Originally from IBM System R.

容易产生页面碎片，需要重新整理，这会严重影响顺序扫描的性能。

但有较快的数据恢复速度。

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.
- Application must define these constraints.
- DBMS ensures that all ICs are true before and after the transaction ends.

*Lecture #23*

A note on **Eventual Consistency**.

- A committed transaction may see inconsistent results (e.g., may not see the updates of an older committed txn).
- Difficult for developers to reason about such semantics.
- The trend is to move away from such models.

# 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. 悲观锁
- **Optimistic:** Assume conflicts are rare; deal with them 乐观锁 after they happen.

# EXAMPLE

Assume at first **A** and **B** each have \$1000.

**T<sub>1</sub>** transfers \$100 from **A**'s account to **B**'s

**T<sub>2</sub>** credits both accounts with 6% interest.

**T<sub>1</sub>**

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

**T<sub>2</sub>**

```
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

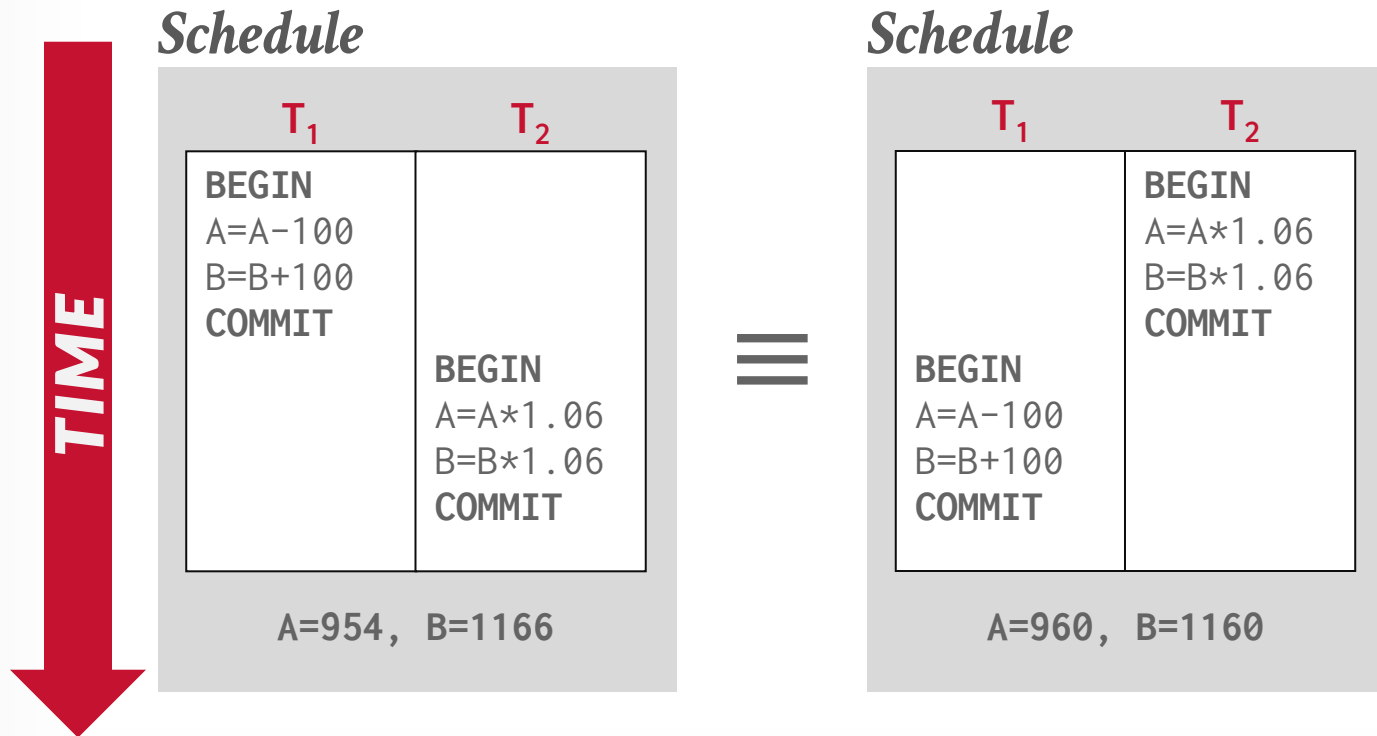
Legal outcomes:

→  $A=954$ ,  $B=1166$  →  $A+B=\$2120$

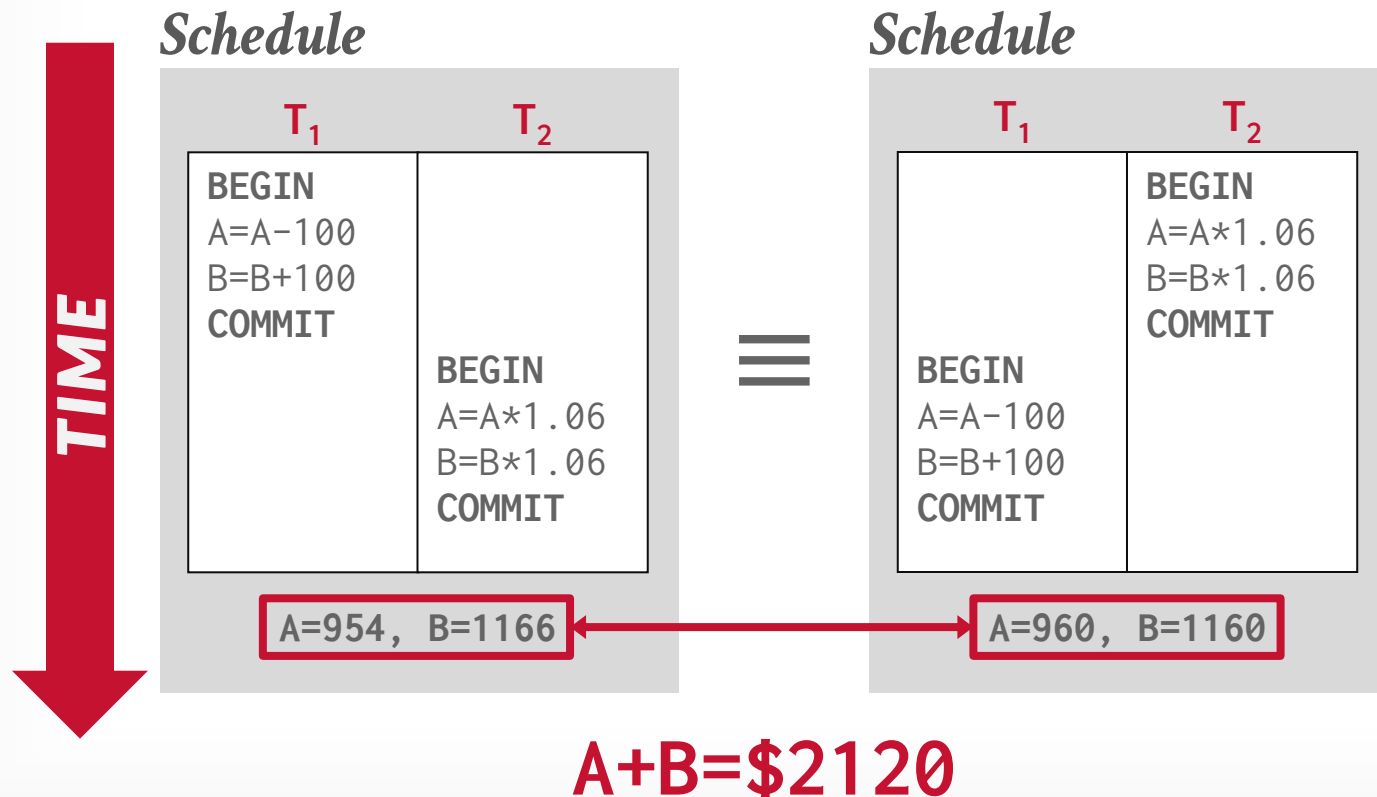
→  $A=960$ ,  $B=1160$  →  $A+B=\$2120$

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

# SERIAL EXECUTION EXAMPLE



# SERIAL EXECUTION EXAMPLE



# INTERLEAVING TRANSACTIONS

---

We interleave txns to maximize concurrency.

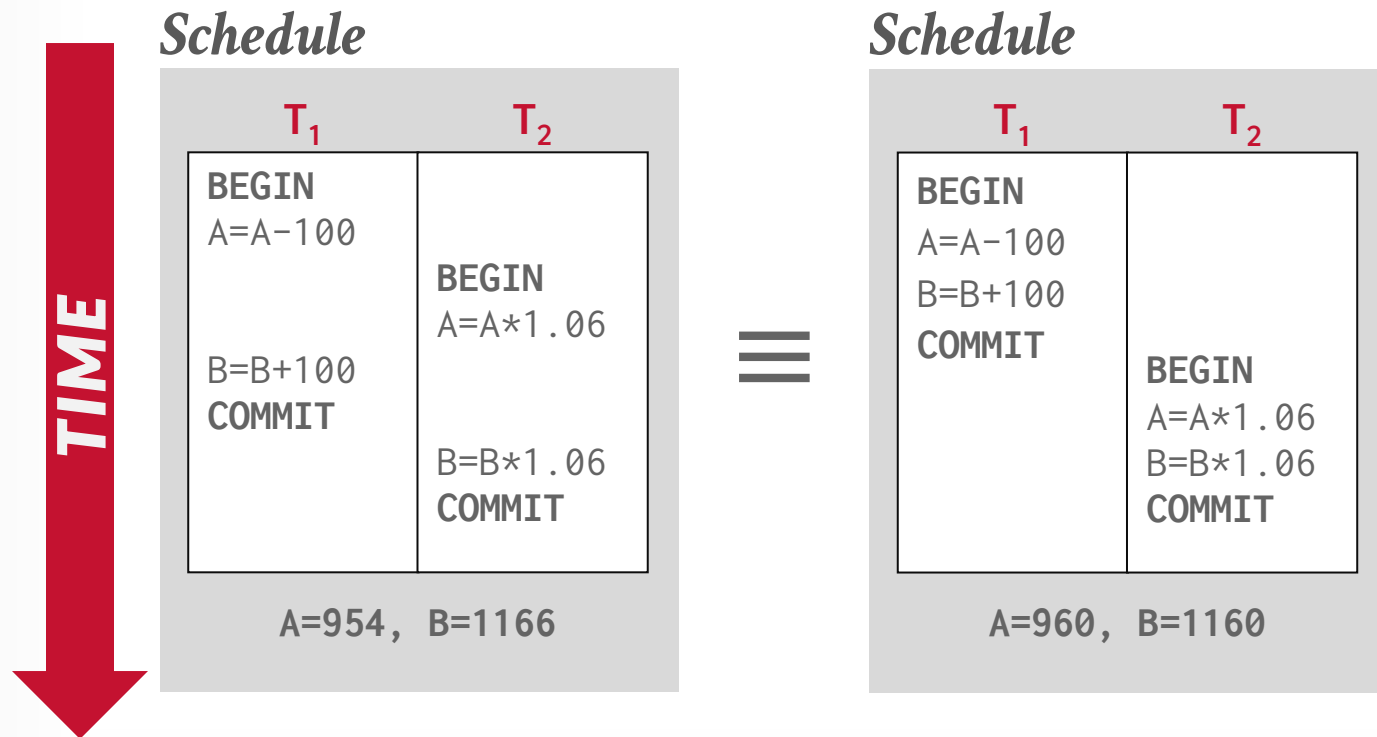
→ Slow disk/network I/O.

→ Multi-core CPUs.

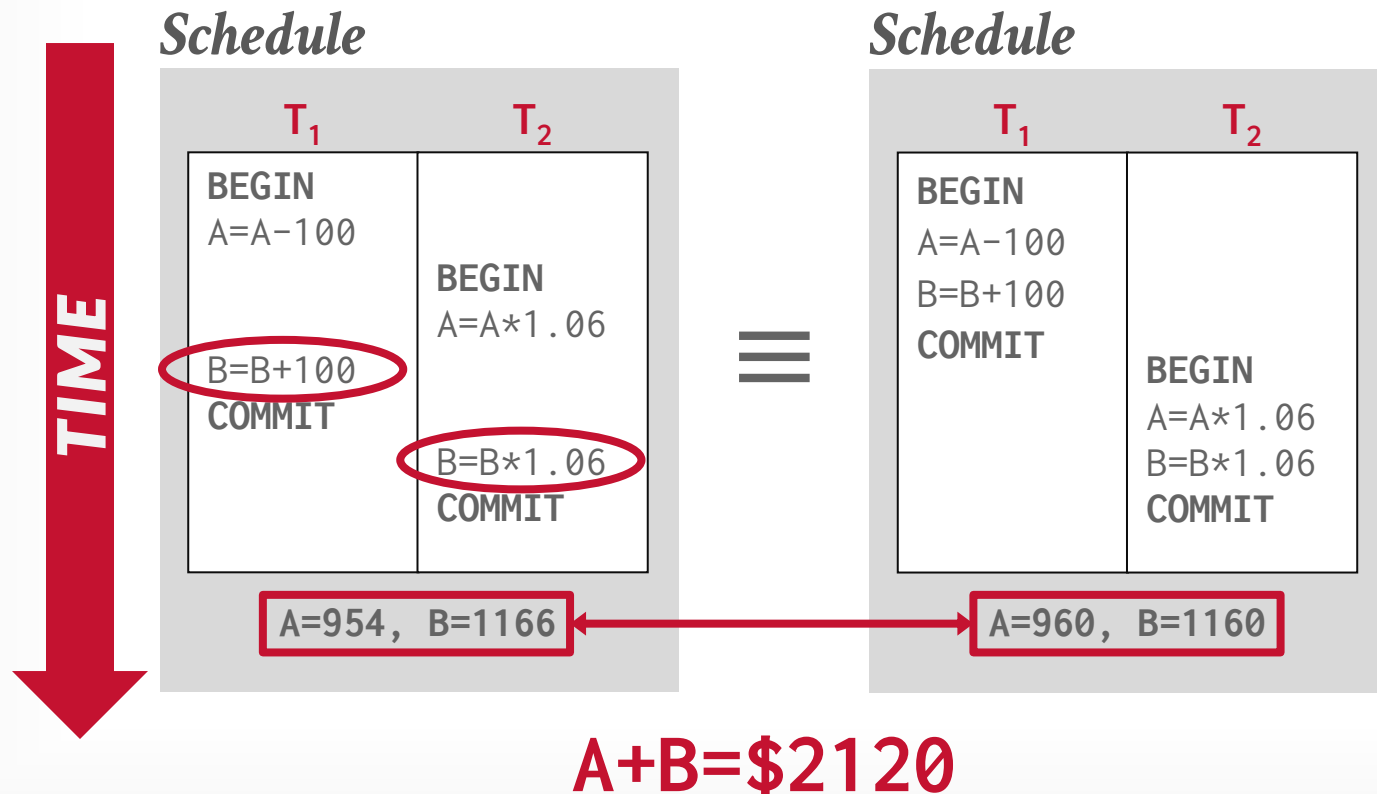
When one txn stalls because of a resource (e.g., page fault), another txn can continue executing and make forward progress.



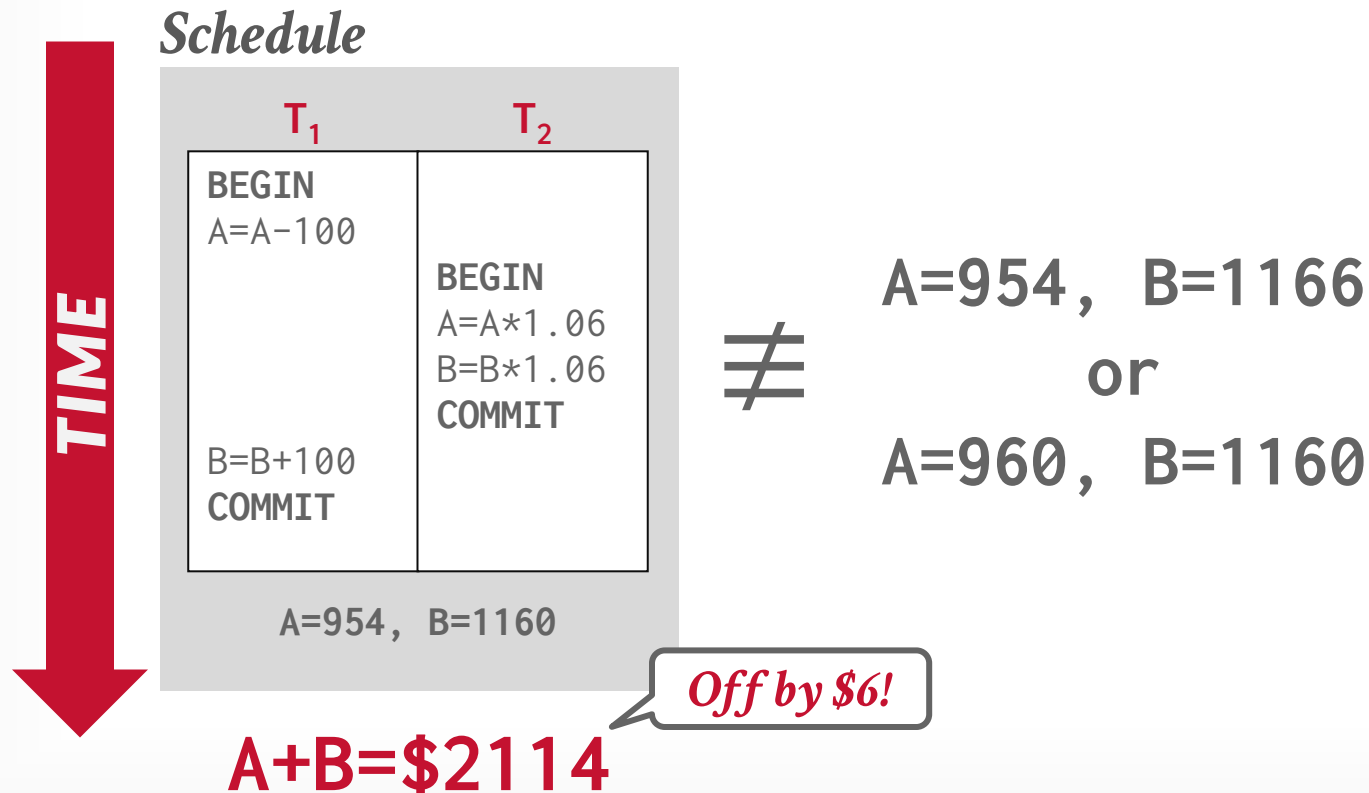
# INTERLEAVING EXAMPLE (GOOD)



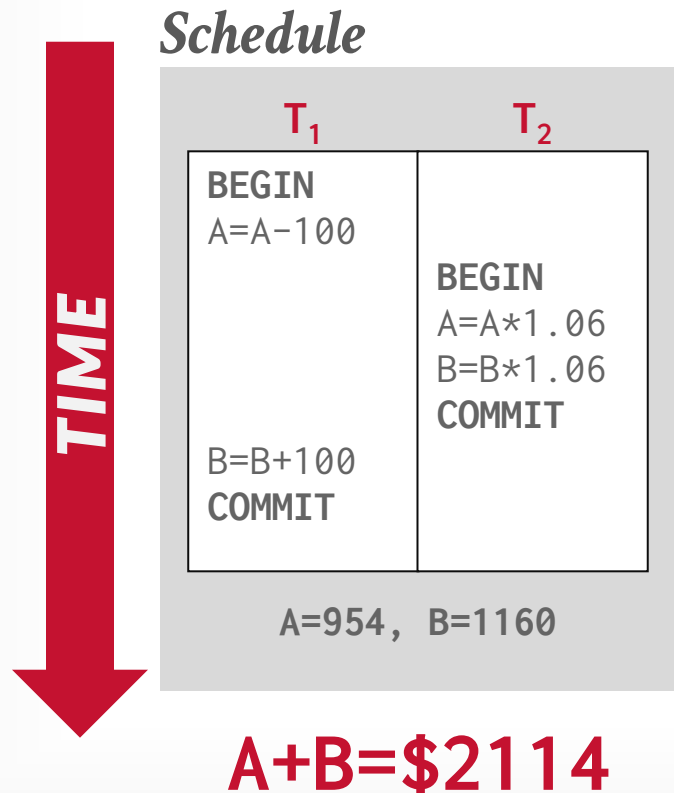
# INTERLEAVING EXAMPLE (GOOD)



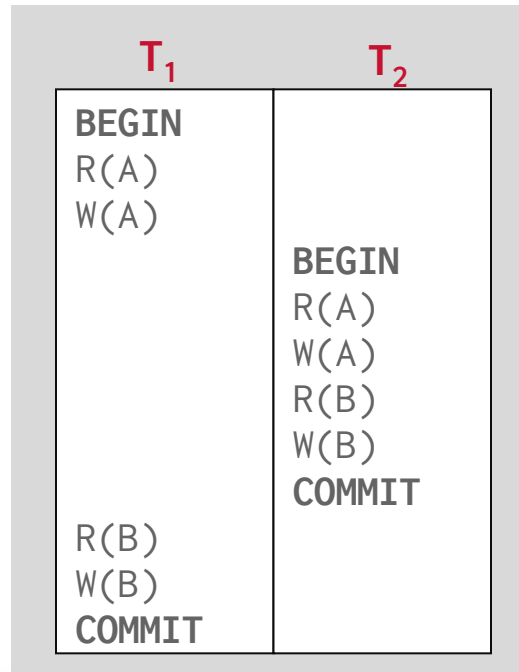
# INTERLEAVING EXAMPLE (BAD)



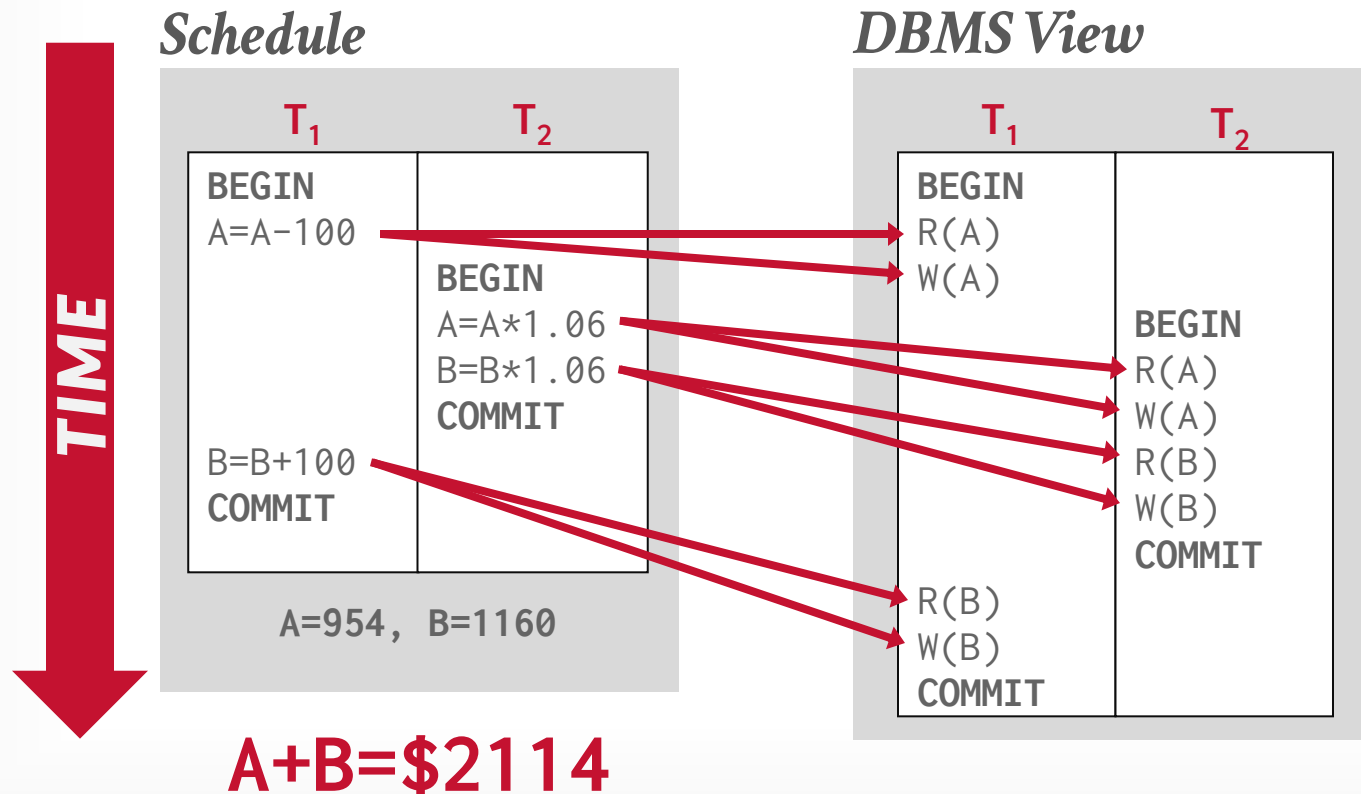
# INTERLEAVING EXAMPLE (BAD)



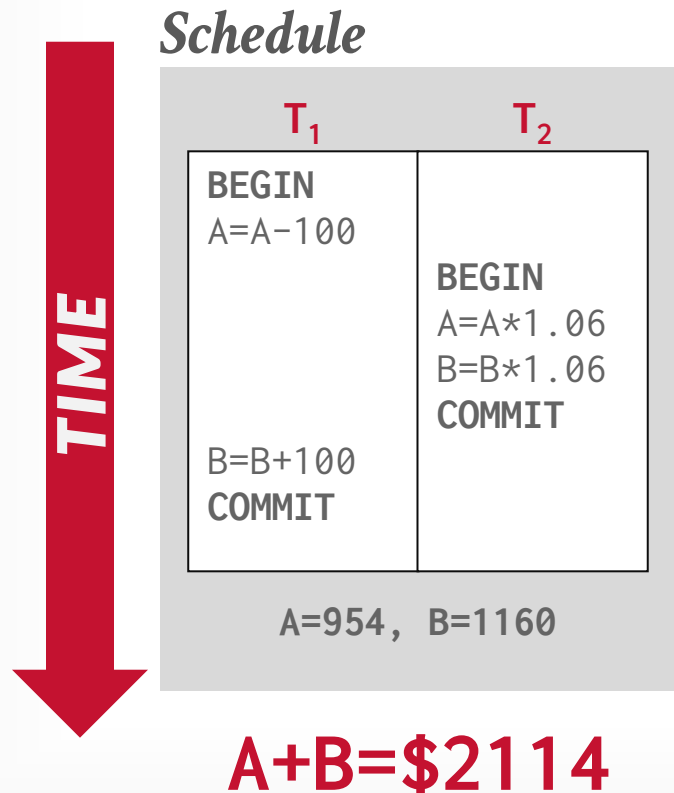
## DBMS View



# INTERLEAVING EXAMPLE (BAD)



# INTERLEAVING EXAMPLE (BAD)



*How do we judge whether a schedule is correct?*

If the schedule is equivalent to some serial execution.

# FORMAL PROPERTIES OF SCHEDULES

---

## Serial Schedule

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

## Equivalent Schedules

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

## Serializable Schedule

- A schedule that is equivalent to some serial execution of the transactions.
- If each transaction preserves consistency, every serializable schedule preserves consistency.

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

- More flexibility means better parallelism.



# CONFLICTING OPERATIONS

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

Two operations conflict if:

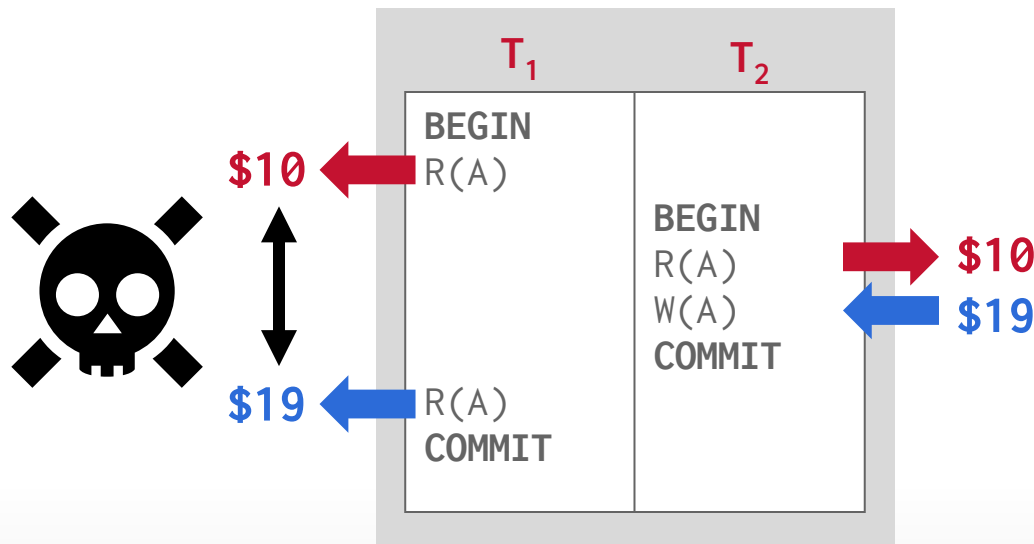
- They are by different transactions,
- They are on the same object and one of them is a **write**.

## Interleaved Execution Anomalies

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

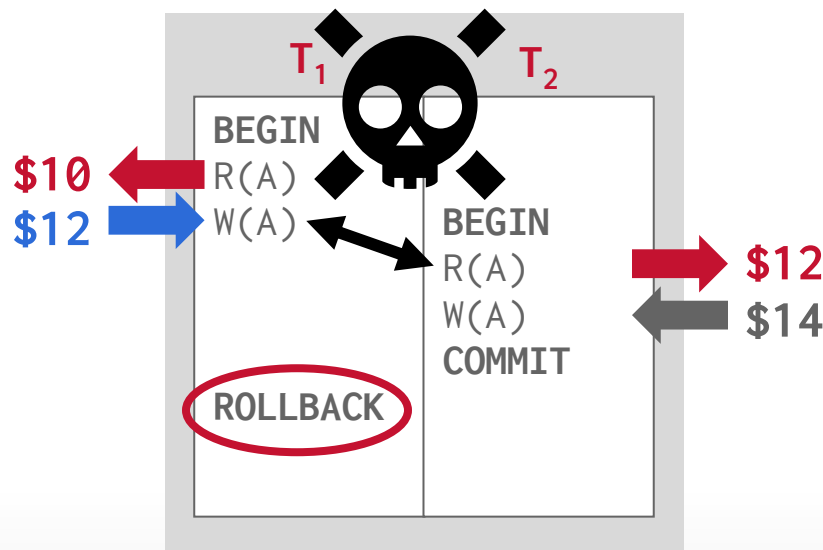
# READ-WRITE CONFLICTS

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



# WRITE-READ CONFLICTS

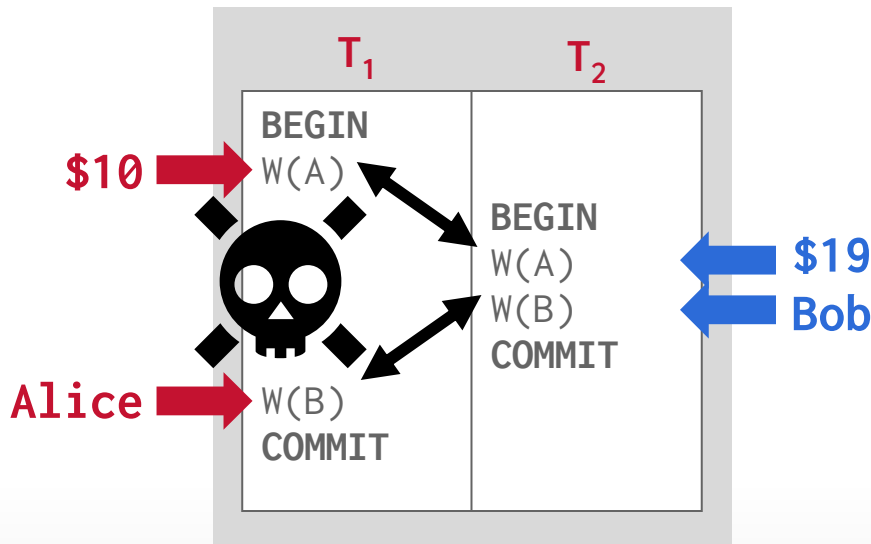
**Dirty Read:** One txn reads data written by another txn that has not committed yet.



# WRITE-WRITE CONFLICTS

丢失更新

**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**
- **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.*

*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.

# DEPENDENCY GRAPHS

One node per txn.

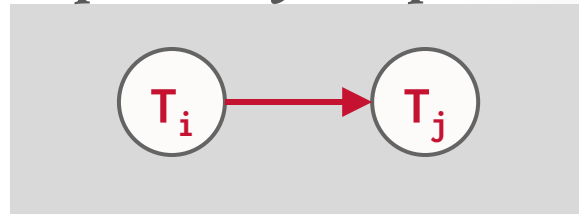
Edge from  $T_i$  to  $T_j$  if:

- 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. 先行图

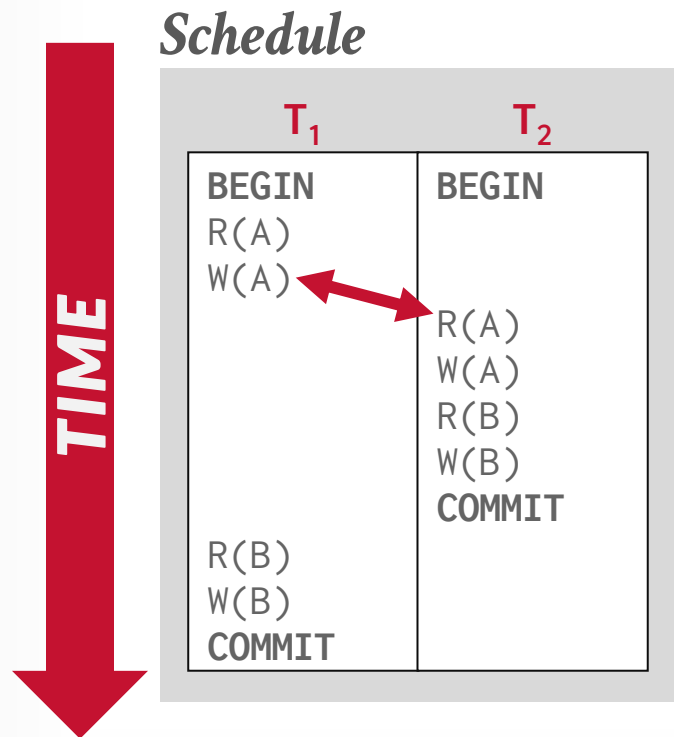
A schedule is conflict serializable iff its dependency graph is acyclic. 无环

*Dependency Graph*

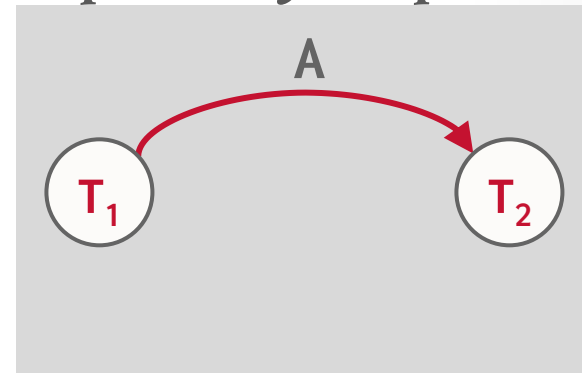




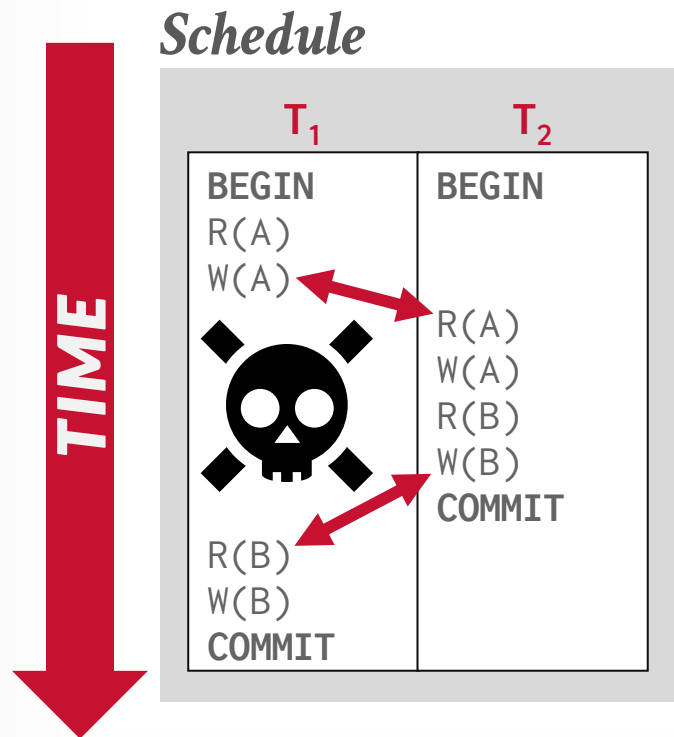
# EXAMPLE #1



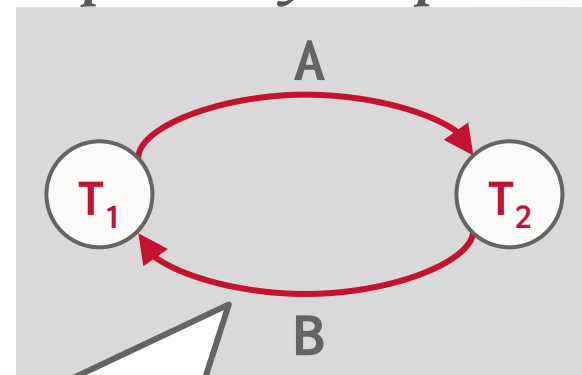
**Dependency Graph**



# EXAMPLE #1

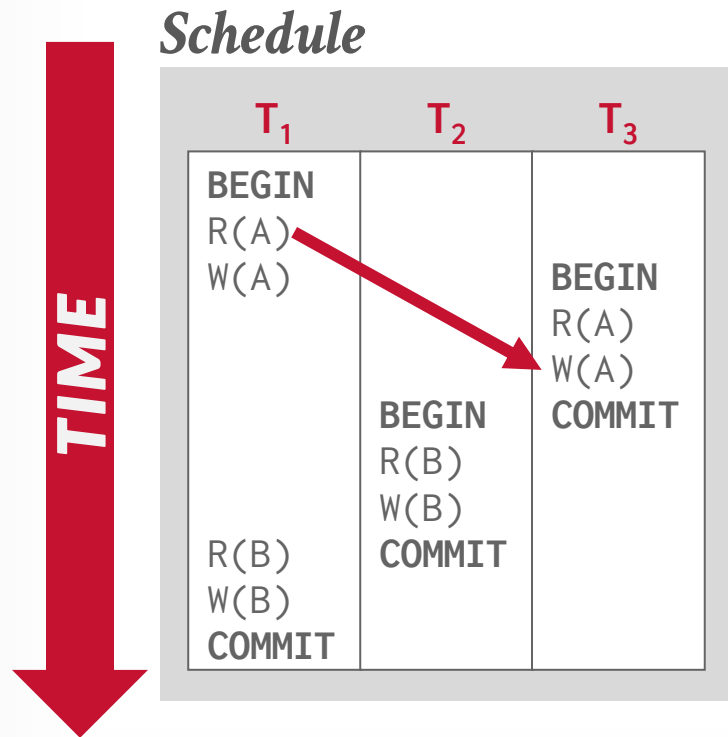


## Dependency Graph

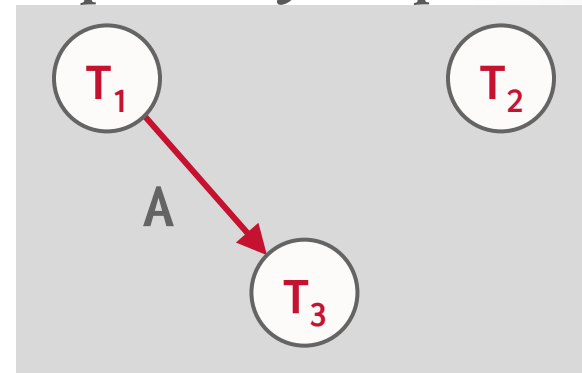


*The cycle in the graph reveals the problem.  
The output of  $T_1$  depends on  $T_2$ ,  
and vice-versa.*

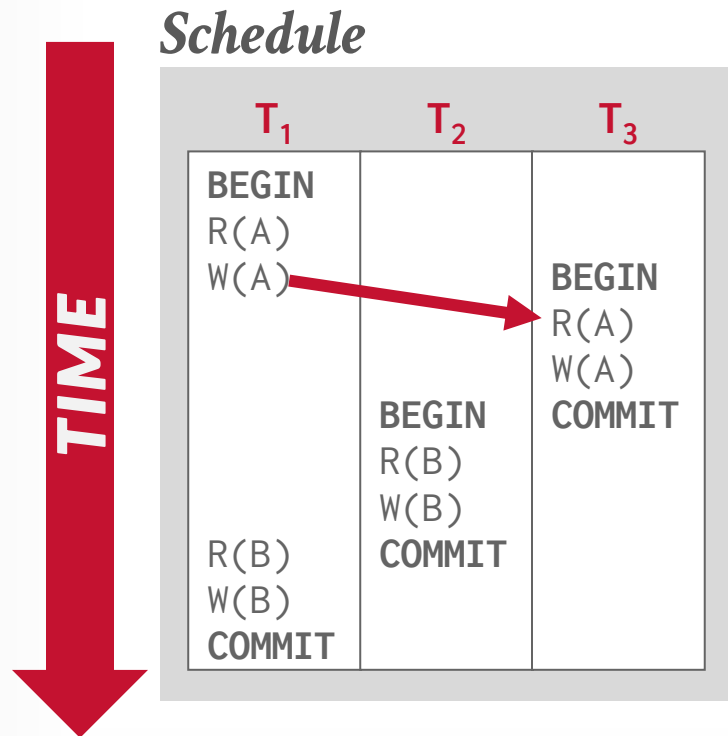
# EXAMPLE #2 – THREE TRANSACTIONS



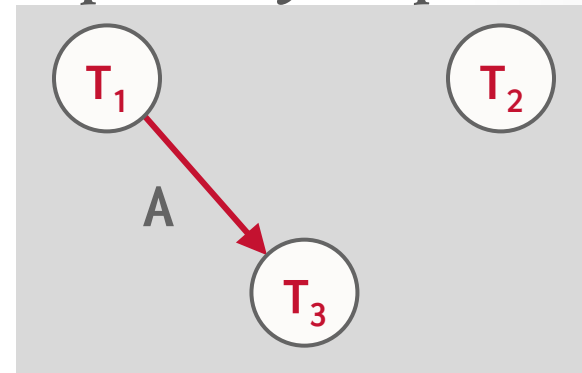
**Dependency Graph**



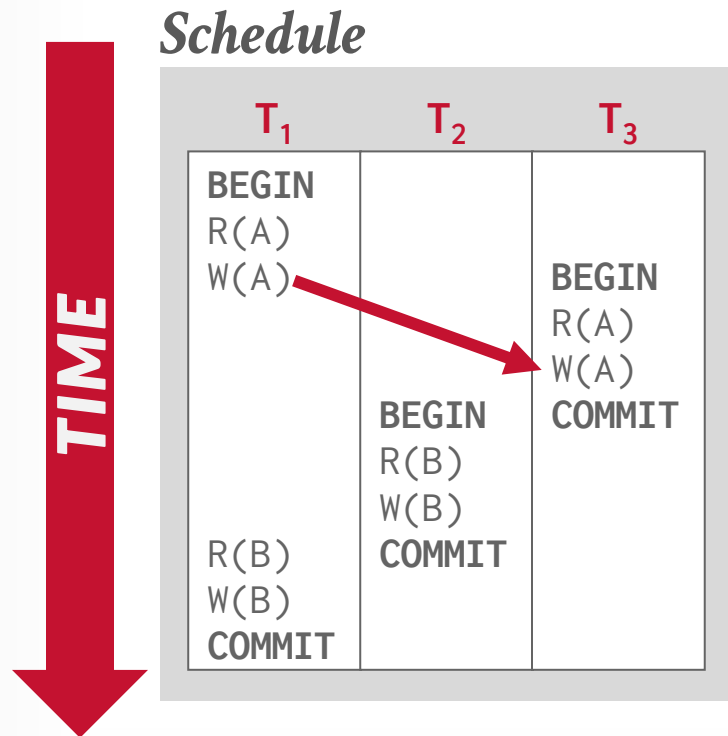
# EXAMPLE #2 – THREE TRANSACTIONS



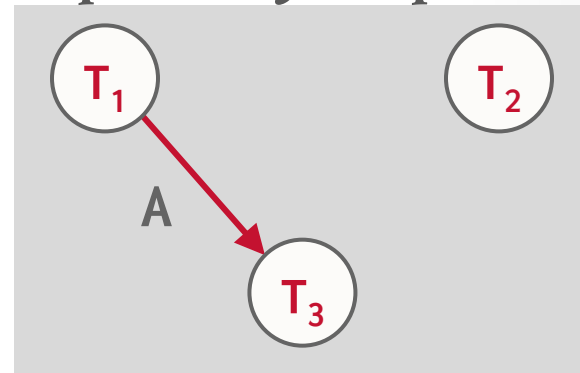
**Dependency Graph**



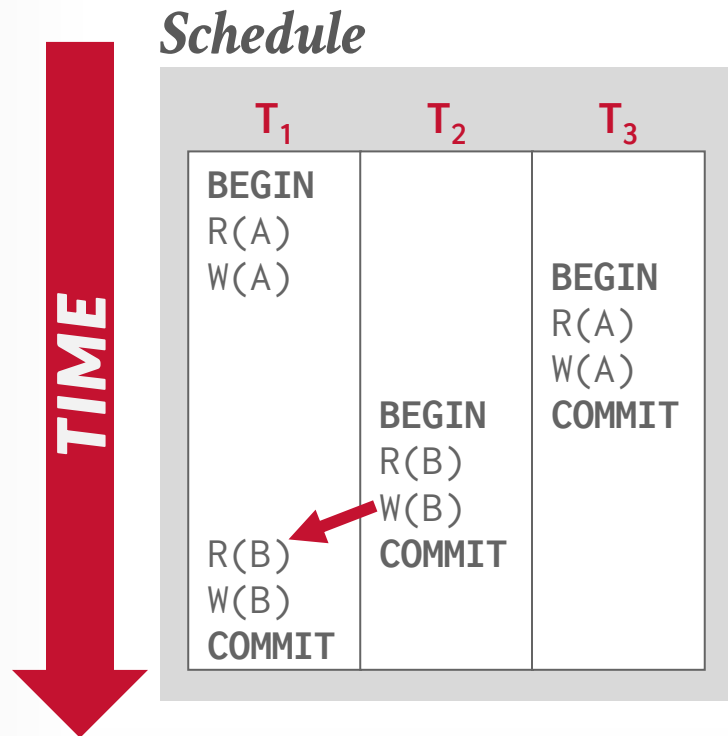
# EXAMPLE #2 – THREE TRANSACTIONS



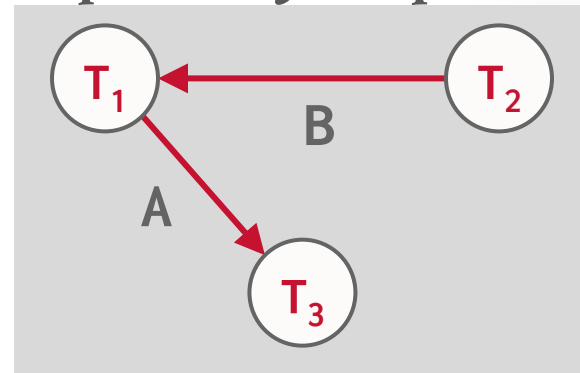
**Dependency Graph**



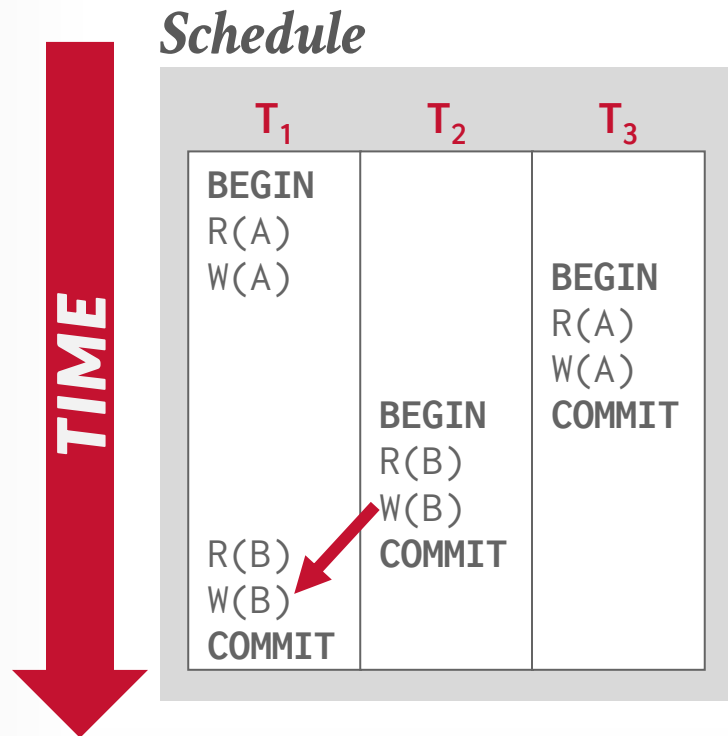
# EXAMPLE #2 – THREE TRANSACTIONS



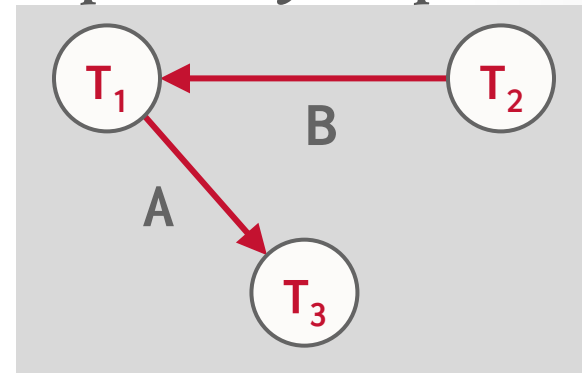
**Dependency Graph**



# EXAMPLE #2 – THREE TRANSACTIONS



**Dependency Graph**

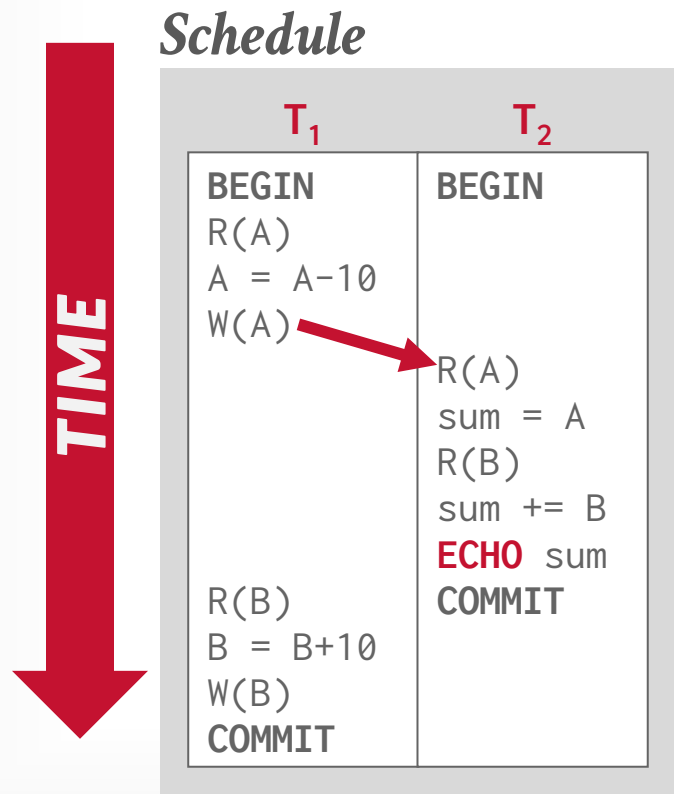


*Is this equivalent to a serial execution?*

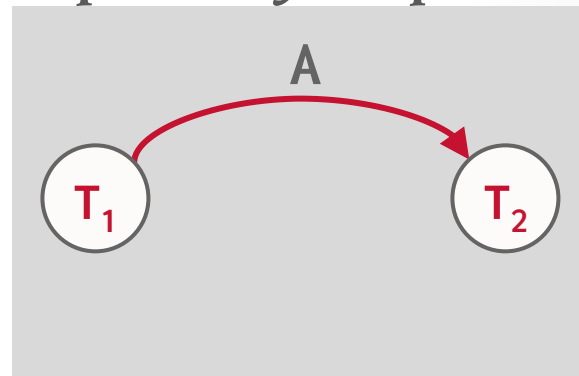
Yes ( $T_2, T_1, T_3$ )

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

# EXAMPLE #3 – INCONSISTENT ANALYSIS

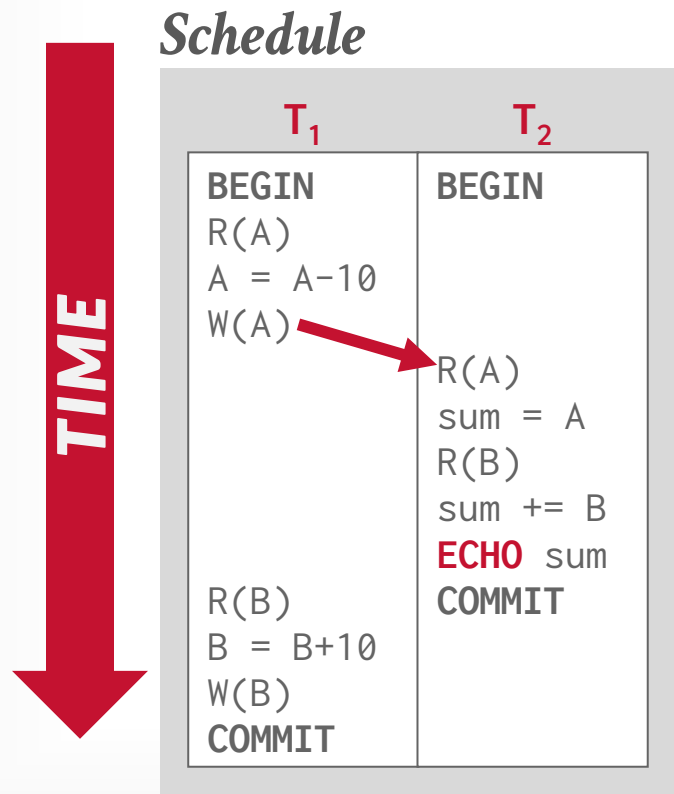


**Dependency Graph**

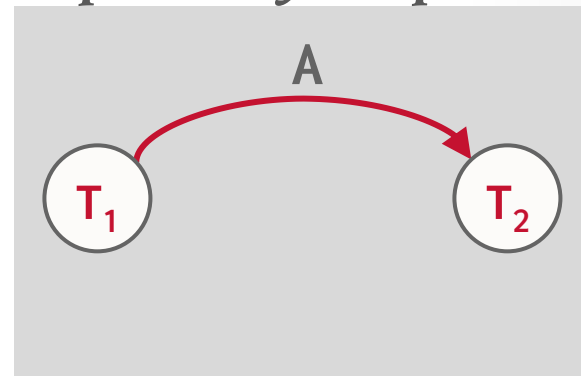




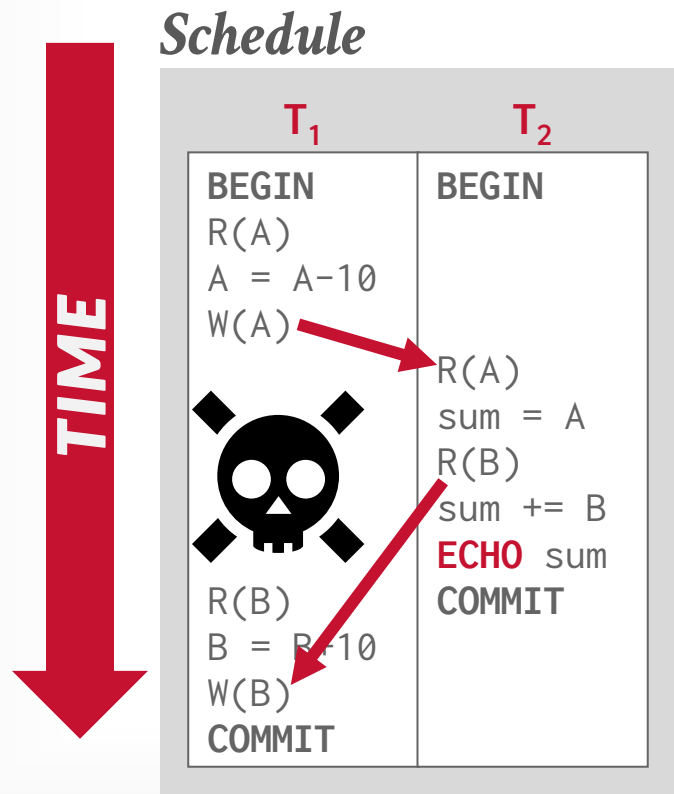
# EXAMPLE #3 – INCONSISTENT ANALYSIS



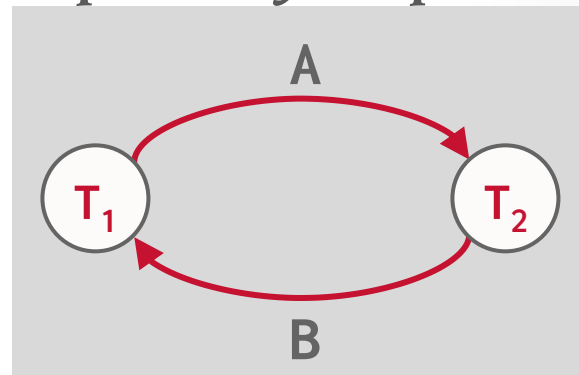
**Dependency Graph**



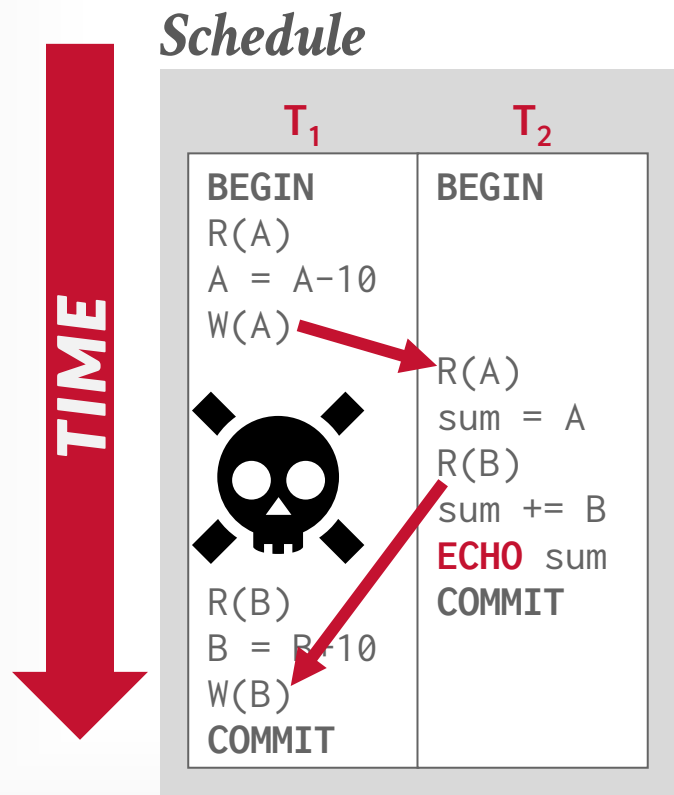
# EXAMPLE #3 – INCONSISTENT ANALYSIS



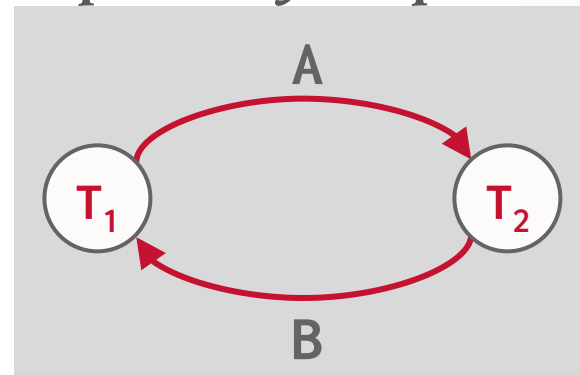
**Dependency Graph**



# EXAMPLE #3 – INCONSISTENT ANALYSIS

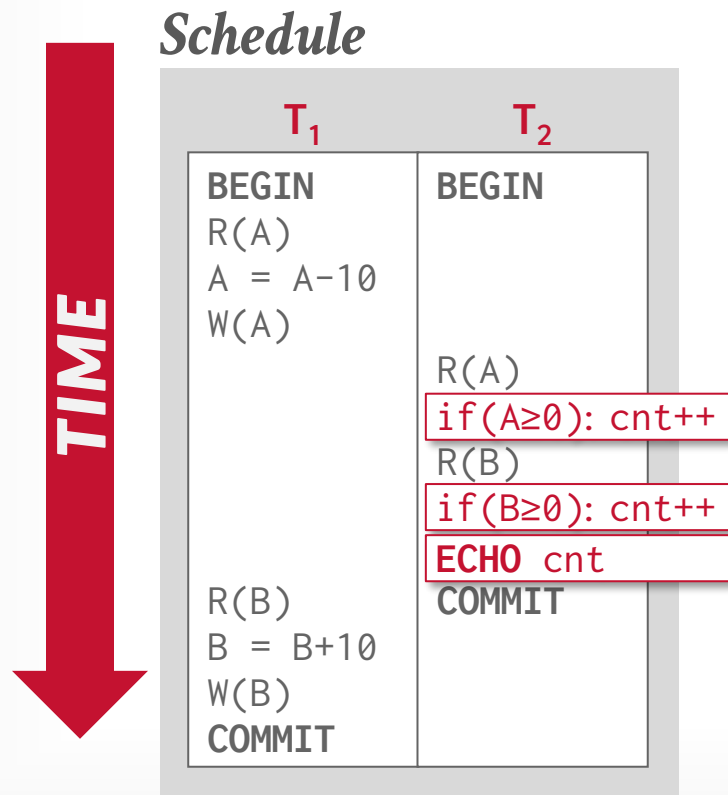


**Dependency Graph**

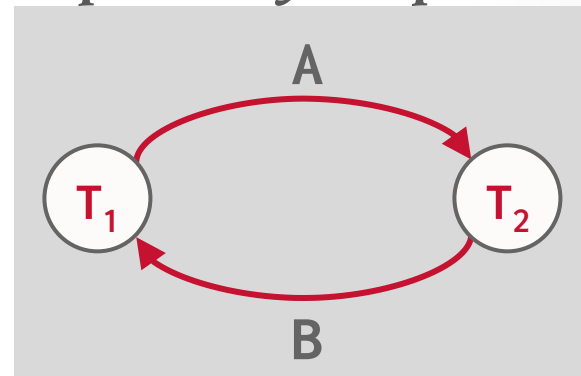


Is it possible to modify only the application logic so that schedule produces a “correct” result but is still not conflict serializable?

# EXAMPLE #3 – INCONSISTENT ANALYSIS



**Dependency Graph**



Is it possible to modify only the application logic so that schedule produces a “correct” result but is still not conflict serializable?

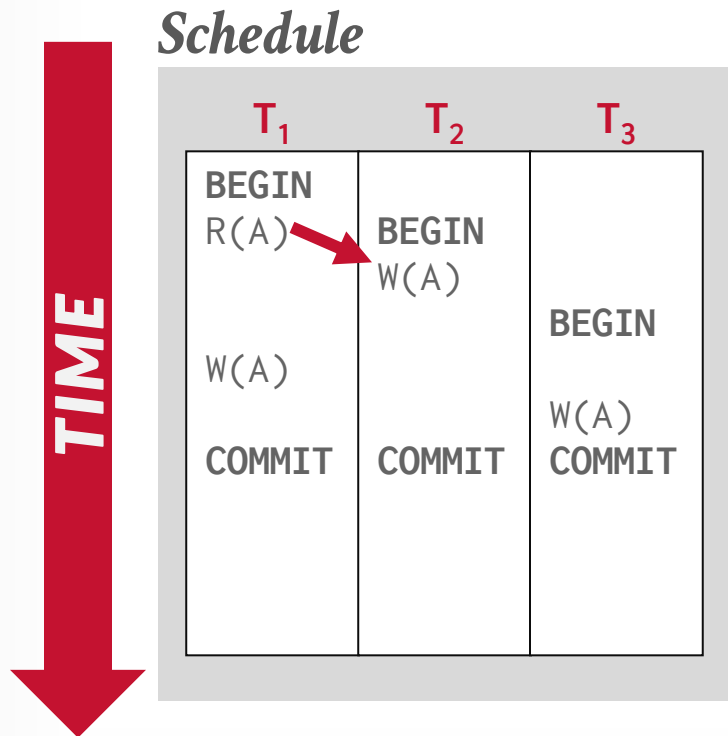
# VIEW SERIALIZABILITY

Alternative (broader) notion of serializability.

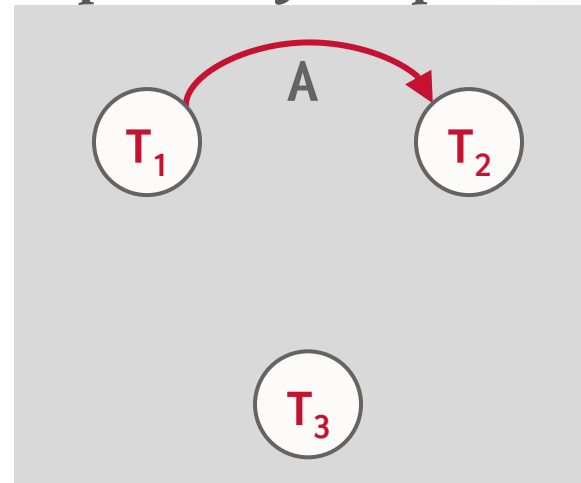
Schedules  $S_1$  and  $S_2$  are view equivalent if:

- 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  $S_2$ .

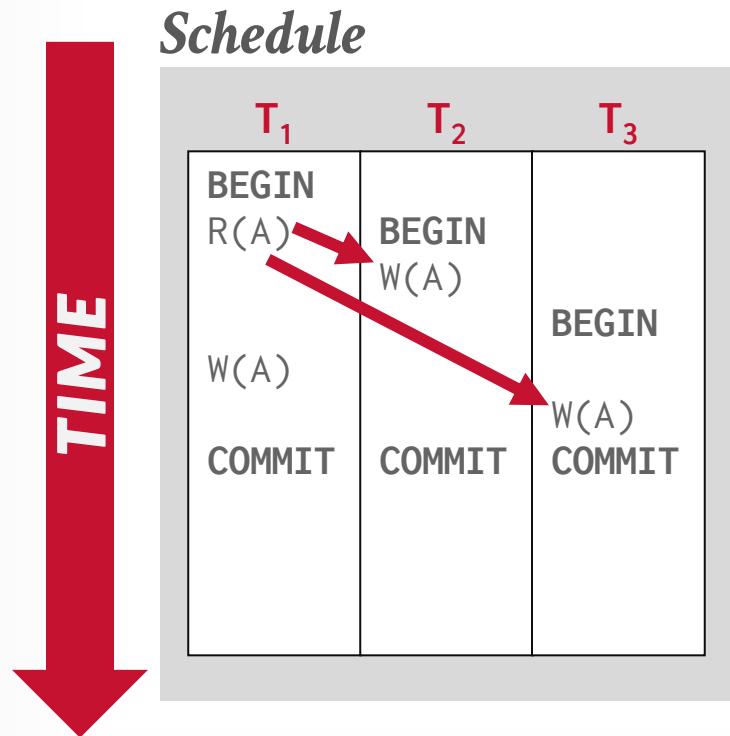
# VIEW SERIALIZABILITY



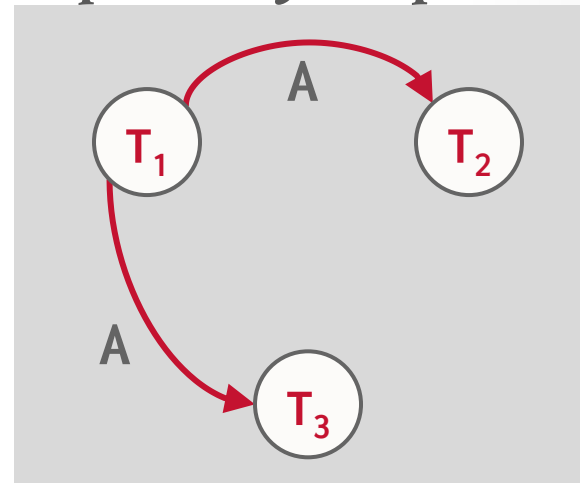
**Dependency Graph**



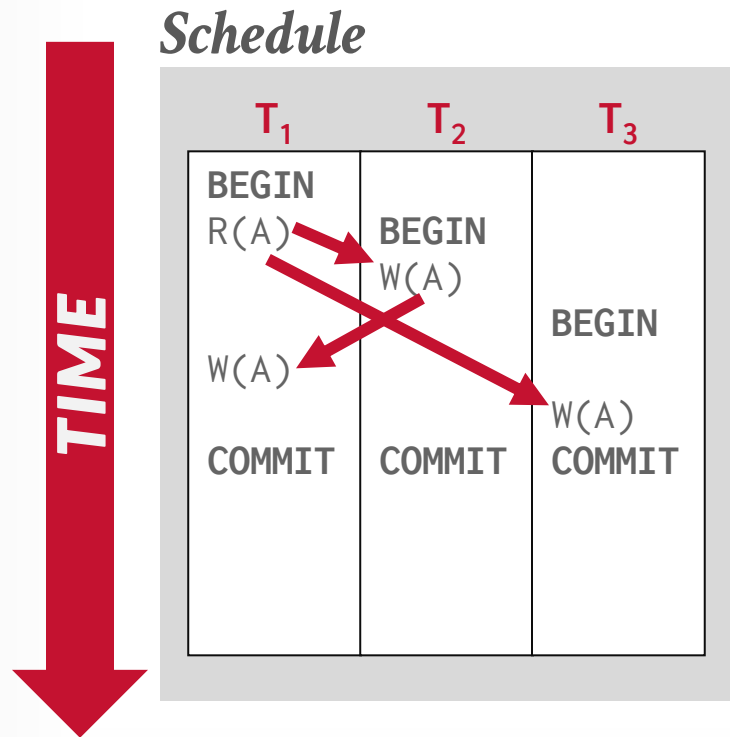
# VIEW SERIALIZABILITY



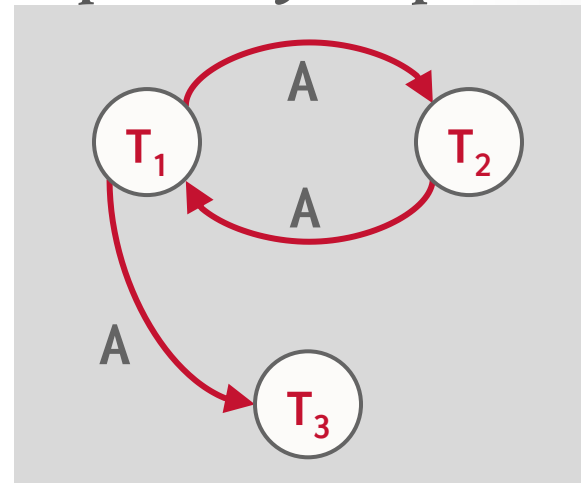
**Dependency Graph**



# VIEW SERIALIZABILITY



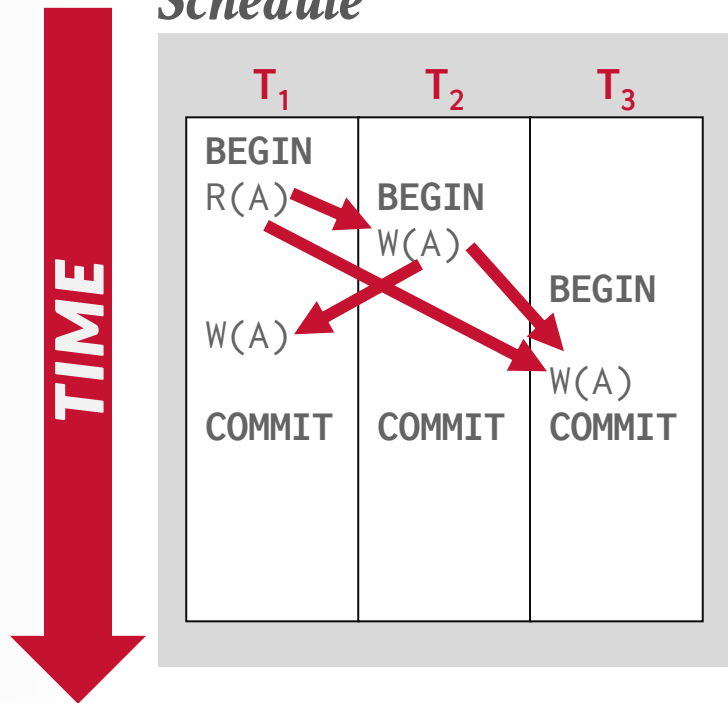
**Dependency Graph**



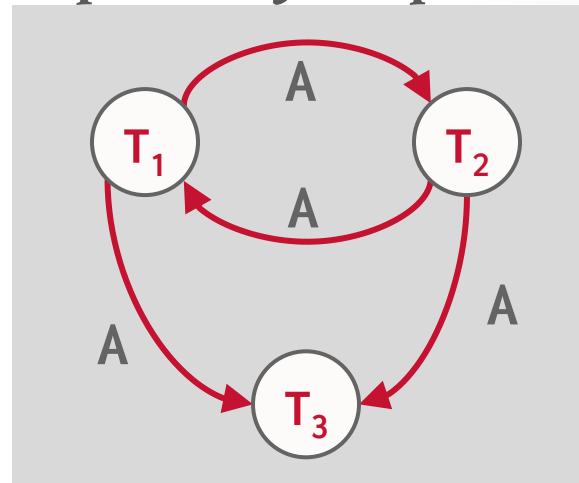


# VIEW SERIALIZABILITY

*Schedule*

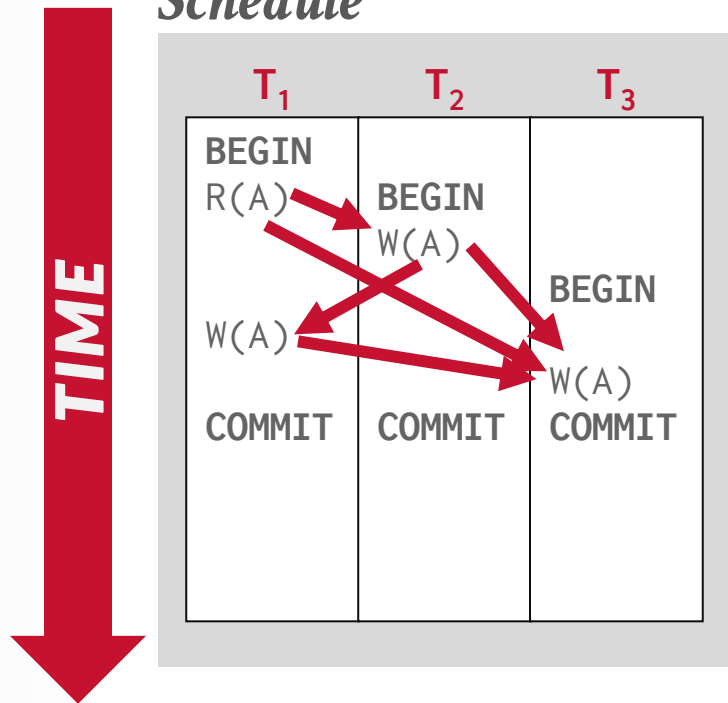


*Dependency Graph*

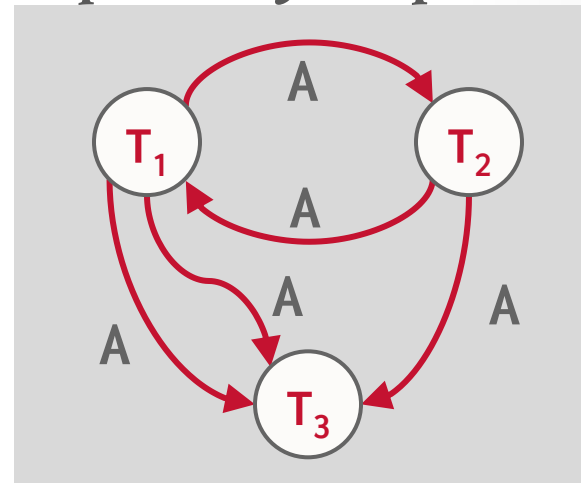


# VIEW SERIALIZABILITY

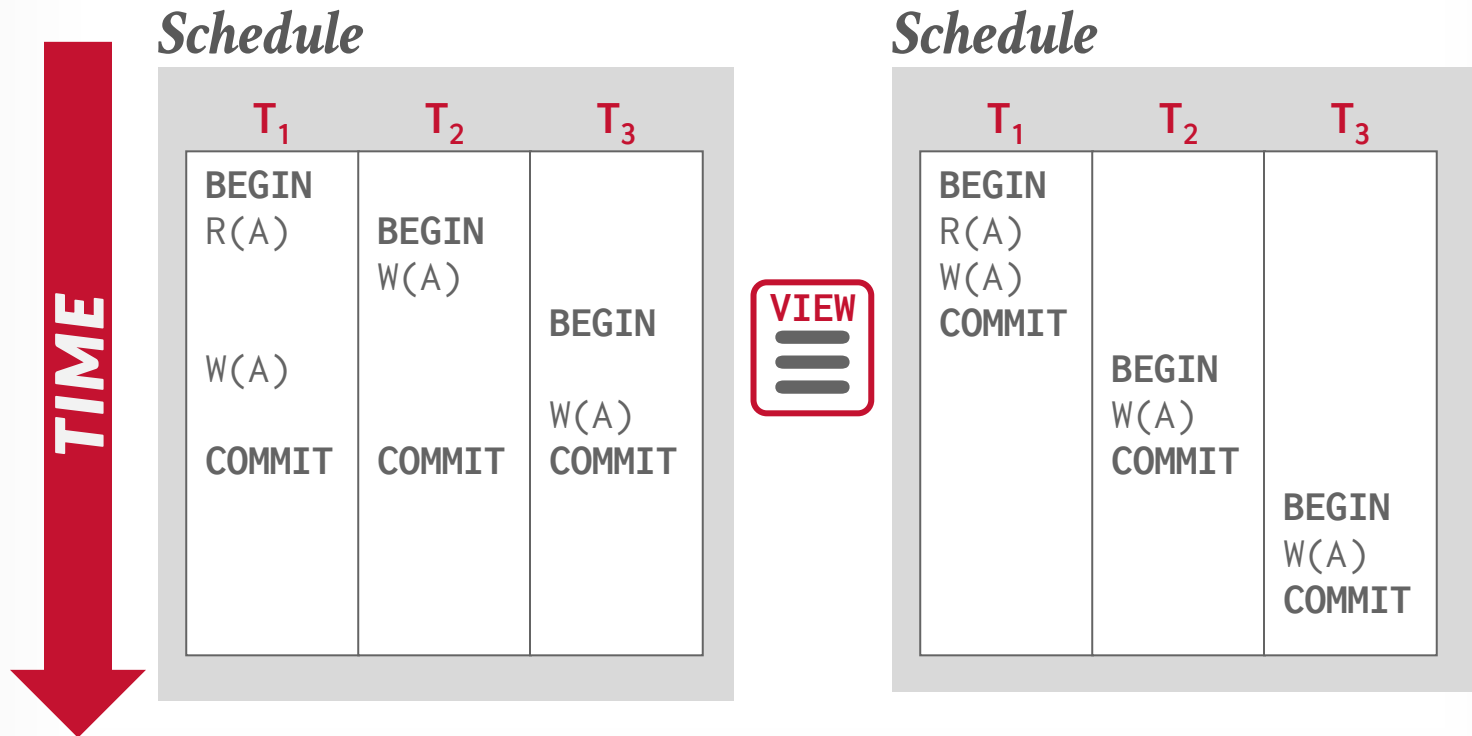
*Schedule*



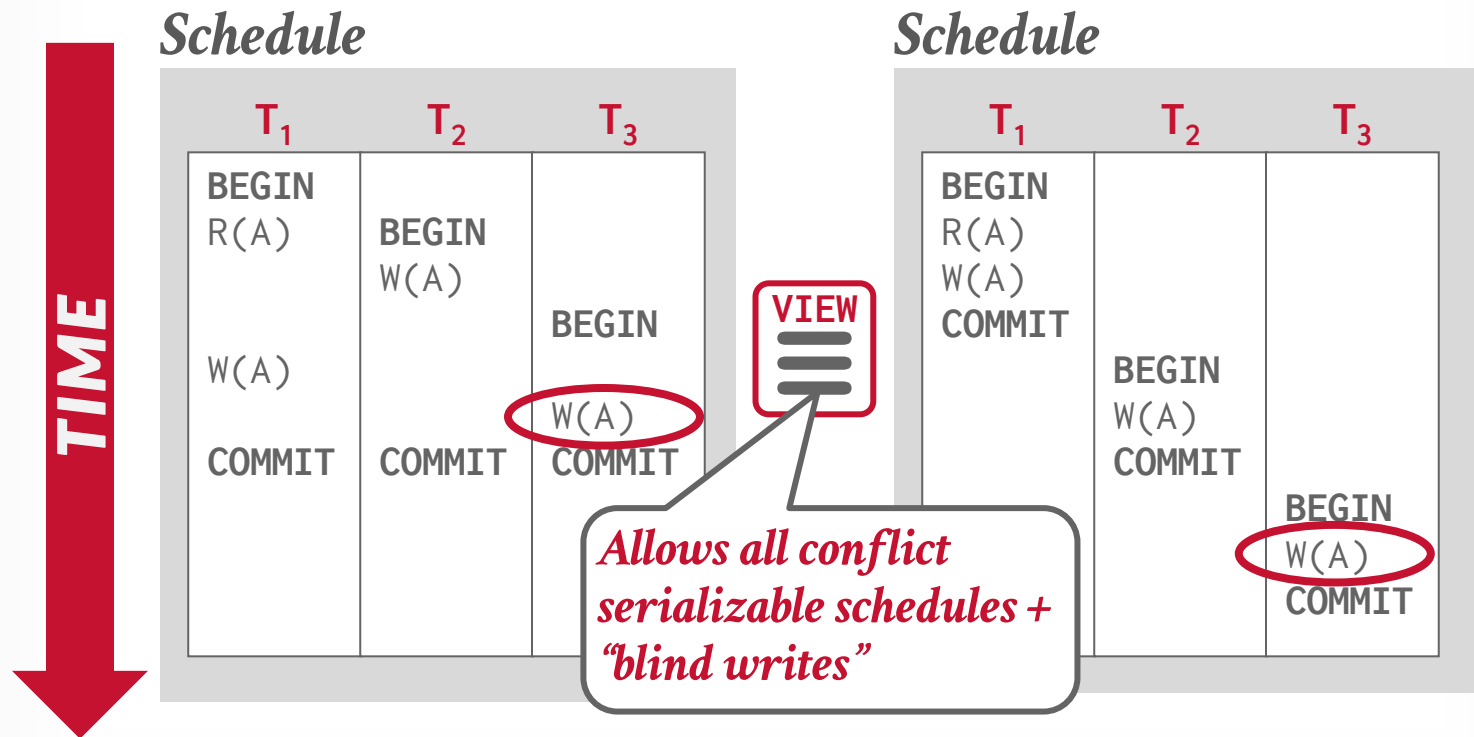
*Dependency Graph*



# VIEW SERIALIZABILITY



# VIEW SERIALIZABILITY



# SERIALIZABILITY

---

**View Serializability** allows for (slightly) more schedules than **Conflict Serializability** does.

→ But it is difficult to enforce efficiently.

Neither definition allows all schedules that you would consider “serializable.”

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

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

# UNIVERSE OF SCHEDULES

*All Schedules*

*View Serializable*

*Conflict Serializable*

*Serial*

# TRANSACTION DURABILITY

---

All the changes of committed transactions should be persistent.

→ No torn updates.

→ No changes from failed transactions.

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

# CORRECTNESS CRITERIA: ACID

---

*Redo/Undo  
Mechanism*

## Atomicity

All actions in txn happen, or none happen.  
*“All or nothing...”*

*Integrity  
Constraints*

## Consistency

If each txn is consistent and the DB starts consistent, then it ends up consistent.  
*“It looks correct to me...”*

*Concurrency  
Control*

## Isolation

Execution of one txn is isolated from that of other txns.  
*“All by myself...”*

*Redo/Undo  
Mechanism*

## Durability

If a txn commits, its effects persist.  
*“I will survive...”*



# 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.

# CONCLUSI

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

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

## 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 Szmaniak, 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.

tenacy 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's relatively poor write throughput. As a result, Spanner has evolved from a Bigtable-like key-value store into a temporal multi-version relational database. Data is stored in a schematized semi-relational format, is versioned, and each version is automatically garbage-collected. Applications can read data at old timestamps, subject to configurable garbage-collection policies. Spanner supports general-purpose transactions, and provides a SQL-like query language.

Spanner is a globally-distributed database. Spanner provides a rich set of features. First, the replication control algorithm can be dynamically controlled at the application level. Applications can specify control which datacenters contain which data, and which users (to control read latency), and which datacenters are from each other (to control write latency). Second, Spanner has two features to implement in a distributed database: it

### 1 Introduction

# NEXT CLASS

---

Two-Phase Locking  
Isolation Levels