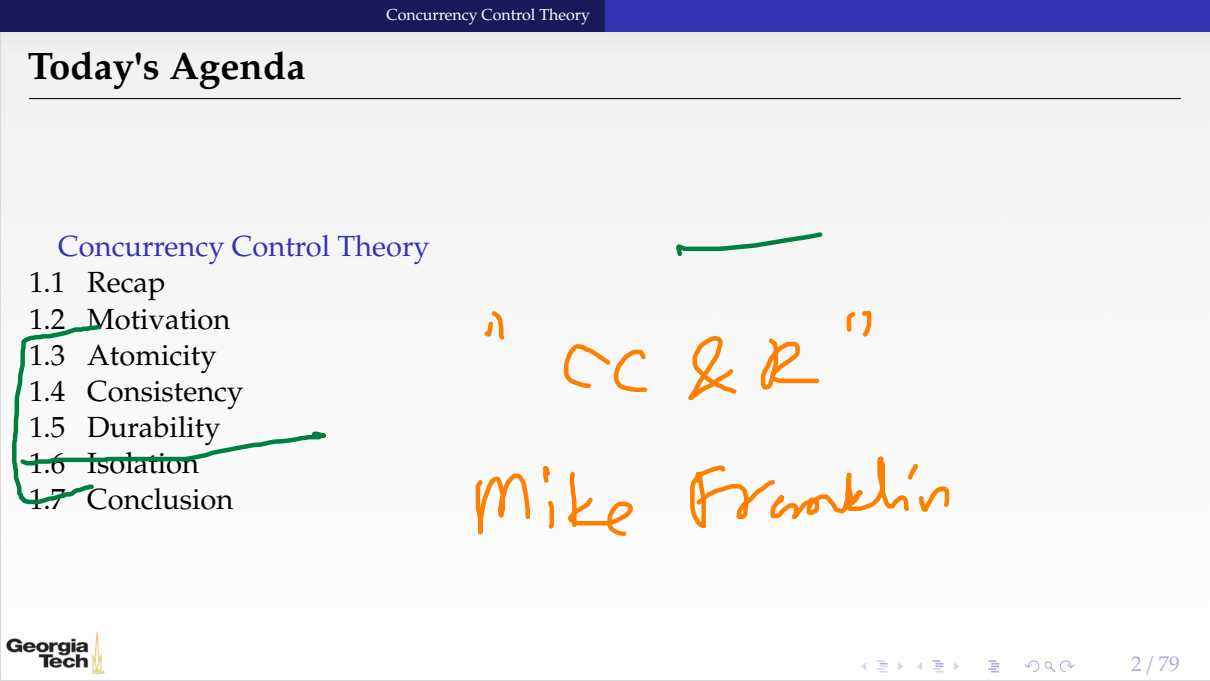
Georgia Tech

# Lecture 12: Concurrency Control Theory

CREATING THE NEXT®

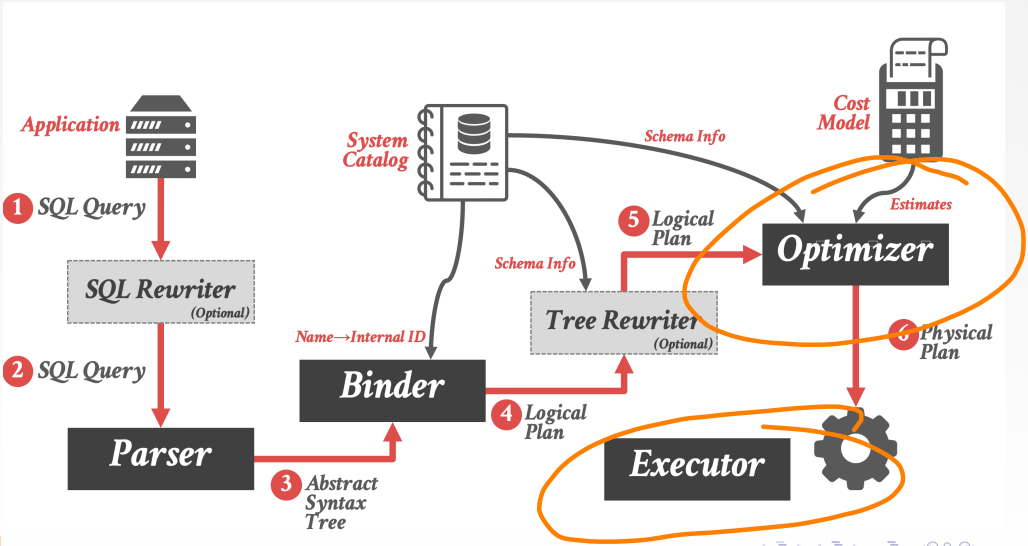
# Today's Agenda





# Recap

# Anatomy of a Database System [Monologue]



# Anatomy of a Database System [Monologue]

- Process Manager
  - Manages client connections
- Query Processor
  - Parse, plan and execute queries on top of storage manager
- Transactional Storage Manager
  - Knits together buffer management, concurrency control, logging and recovery
- Shared Utilities
  - Manage hardware resources across threads

# Anatomy of a Database System [Monologue]

- Process Manager
  - Connection Manager + Admission Control
- Query Processor
  - Query Parser
  - Query Optimizer (*a.k.a.*, Query Planner)
  - Query Executor
- Transactional Storage Manager
  - Lock Manager
  - Access Methods (*a.k.a.*, Indexes)
  - Buffer Pool Manager
  - Log Manager
- Shared Utilities
  - Memory, Disk, and Networking Manager

# Today's Agenda

- Motivation
- Atomicity,
- Consistency
- Durability
- Isolation

# Motivation

# Motivation

- **Lost Updates:**
  - We both change the same record in a table at the same time. How to avoid **race condition**?
  - **Concurrency Control** protocol
- **Durability:**
  - You transfer $100 between bank accounts but there is a power failure. What is the correct database state?
  - **Recovery** protocol

# Concurrency Control & Recovery

- Valuable properties of DBMSs.
- Based on concept of transactions with **ACID** properties.
- Let's talk about transactions . . .

# Transaction

- 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

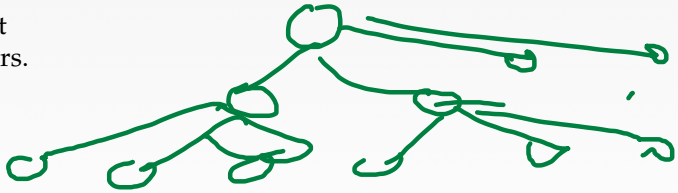
- Move $100 from A's bank account to B's account.
- Transaction:
  - Check whether A has $100.
  - Deduct $100 from A's account.
  - Add $100 to B's account.

# Strawman Solution

- Execute each txn one-by-one (*i.e.*, **serial order**) as they arrive at the DBMS.
  - One and only one txn can be running at the same time in the DBMS.
- Before a txn starts, copy the entire database to a new file and make all changes to that file.
  - If the txn completes successfully, overwrite the original file with the new one.
  - If the txn fails, just remove the dirty copy.

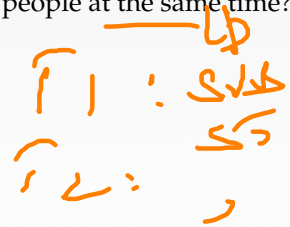
# Problem Statement

- A (potentially) better approach is to allow concurrent execution of independent transactions.
- **Why do we want that?**
  - Better utilization/throughput
  - Lower response times to users.
- But we also would like:
  - Correctness
  - Fairness



# Transactions

- Hard to ensure **correctness**?
  - What happens if A only has $100 and tries to pay off two people at the same time?

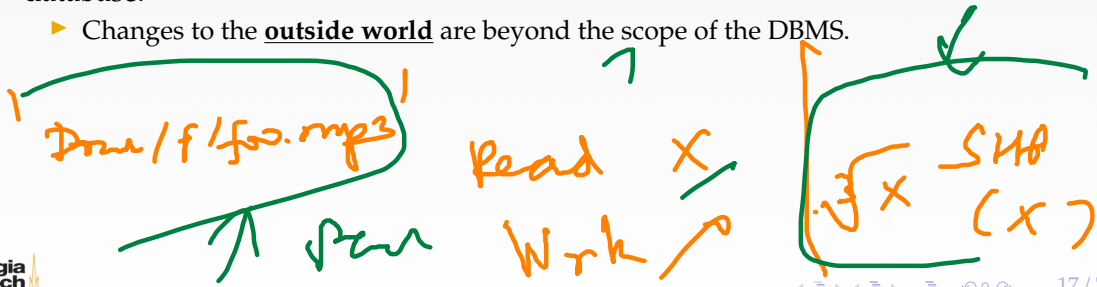


# Problem Statement

- 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
- However, 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”

# Atomicity

# 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 A's account but then the DBMS aborts the txn before we transfer it.
- **Scenario 2:**
  - We take $100 out of A's account but then there is a power failure before we transfer it.
- What should be the **correct state** of A'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)

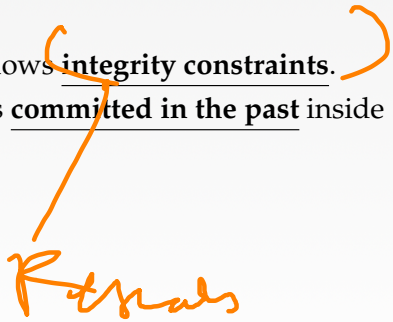
# Consistency

# 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

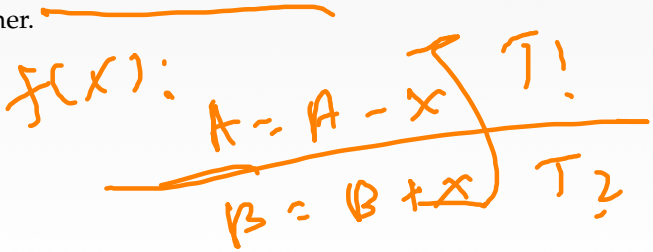
# Database Consistency

- 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.
  - We won't discuss this further.



# Durability

# Durability

- All of 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.

# Isolation

# Isolation of Transactions

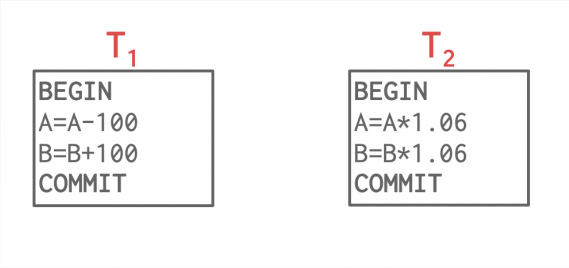
- Users submit txns, and each txn executes as if it was running by itself.
  - Easier programming model to reason about.
- But the DBMS achieves concurrency by interleaving the actions (reads/writes of DB objects) of txns.
- We need a way to interleave txns but still make it appear as if they ran one-at-a-time.

# 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.

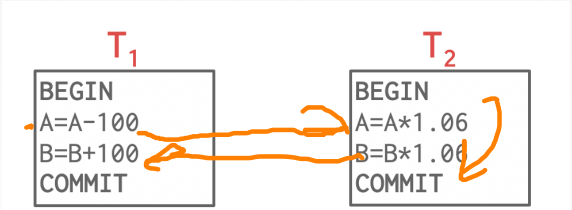
# Example

- Assume at first A and B each have $1000.
- T1 transfers $100 from A's account to B's
- T2 credits both accounts with 6% interest.



# Example

- Assume at first A and B each have $1000.
- **What are the possible outcomes of running T1 and T2?**



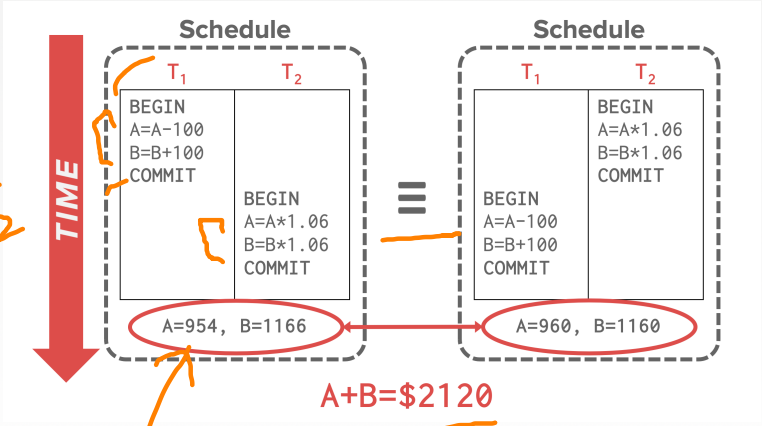
# Example

- Assume at first A and B each have $1000.
- What are the possible outcomes of running T1 and T2?
- Many! But A+B should be:
  - $2000 * 1.06 = 2120$
- There is no guarantee that T1 will execute before T2 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

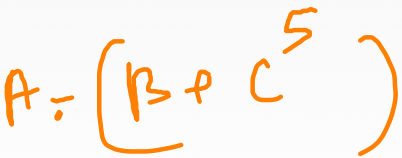
- Legal outcomes:
  - A=954, B=1166 → A+B=2120
  - A=960, B=1160 → A+B=2120
- The outcome depends on whether T1 executes before T2 or vice versa.

# Serial Execution Example

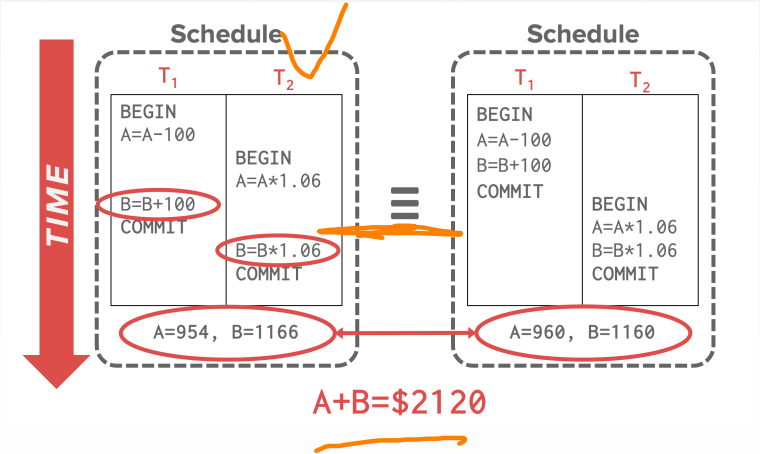


# Interleaving Transactions

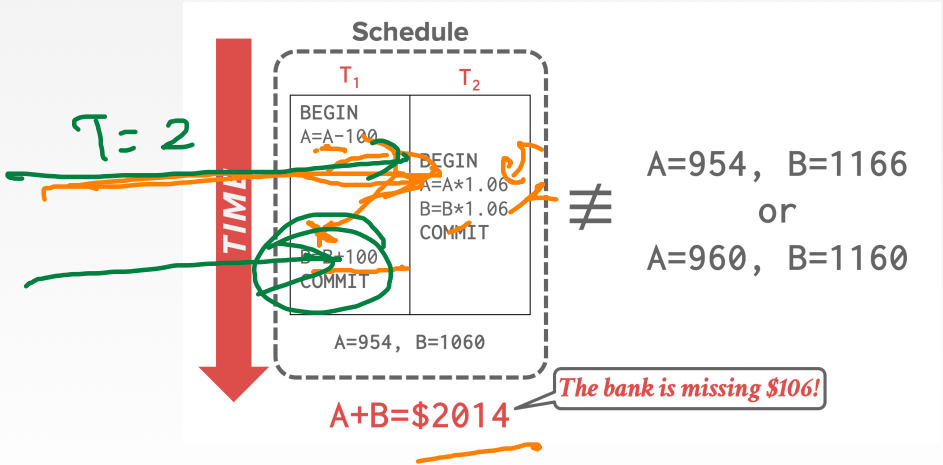
- We interleave txns to maximize concurrency.
  - Slow disk/network I/O.
  - Multi-core CPUs.
- When one txn stalls because of a resource (*e.g.*, page fault), another txn can continue executing and make forward progress.



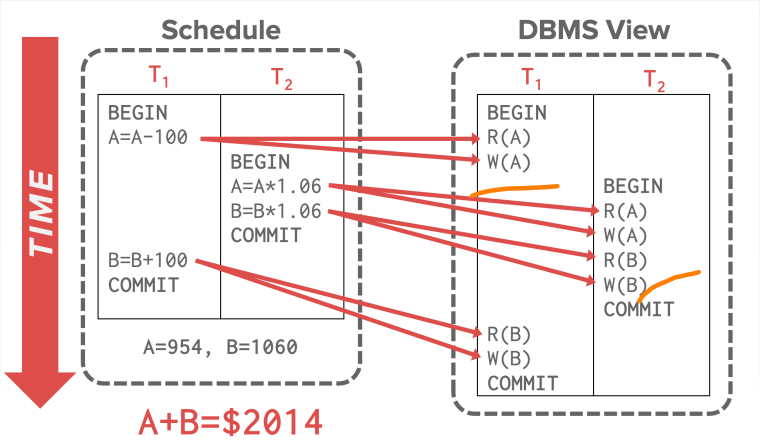
# Interleaving Example (Good)



# Interleaving Example (Bad)



# Interleaving Example (Bad)



# 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!

$S_1 \equiv S_2$

# 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

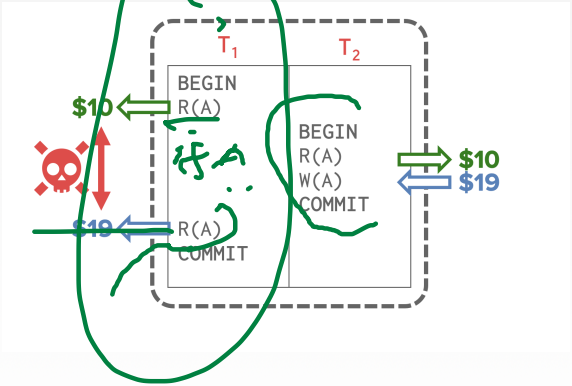
- We need a formal notion of equivalence that can be implemented efficiently based on the notion of **conflicting operations**
- Two operations **conflict** if:
  - They are by different transactions,
  - They are on the same object and at least one of them is a **write**.

# Interleaved Execution Anomalies

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

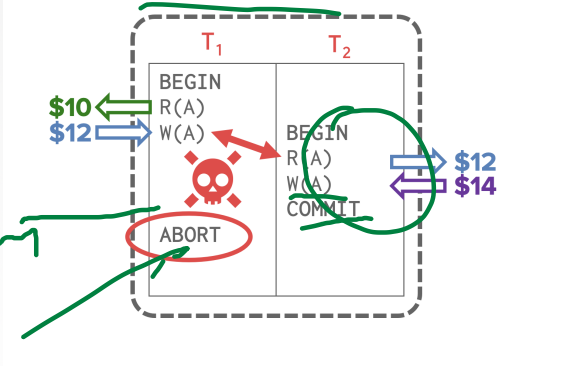
# Read-Write Conflicts

- Unrepeatable Reads



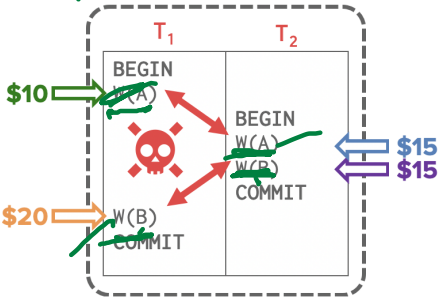
# Write-Read Conflicts

- Reading Uncommitted Data ("Dirty Reads")



# Write-Write Conflicts

- Overwriting Uncommitted Data



# 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.

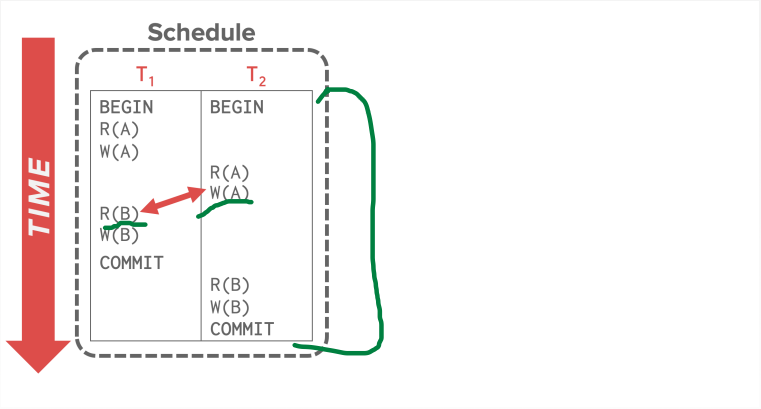
# Conflict Serializable Schedules

- 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.

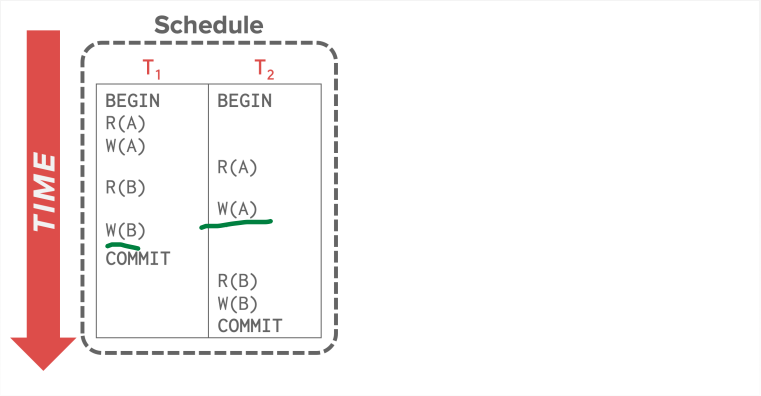
# Conflict Serializablity: Intuition

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

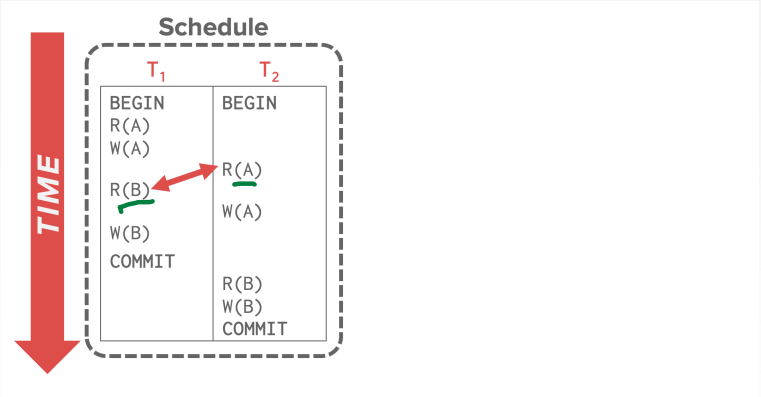
# Conflict Serializablity: Intuition



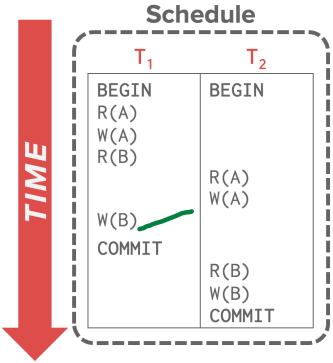
# Conflict Serializablity: Intuition



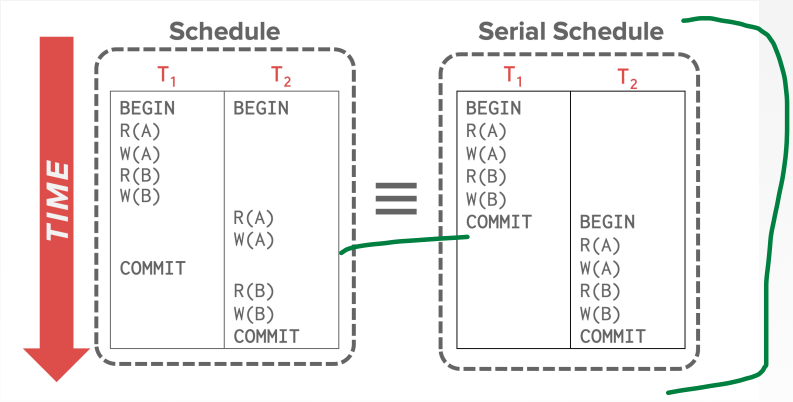
# Conflict Serializablity: Intuition



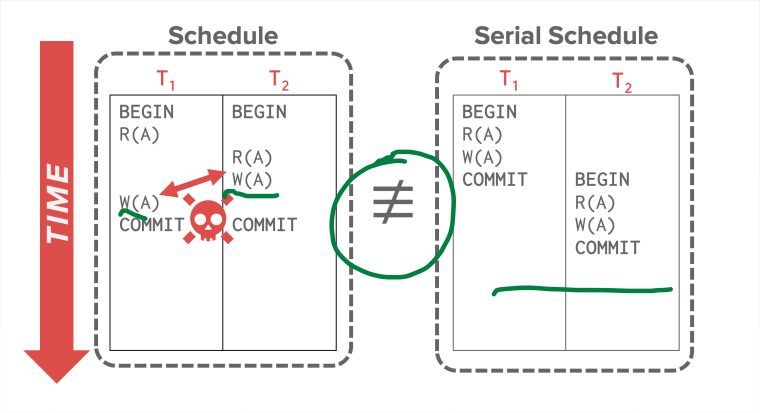
# Conflict Serializablity: Intuition



# Conflict Serializablity: Intuition



# Conflict Serializablity: Intuition

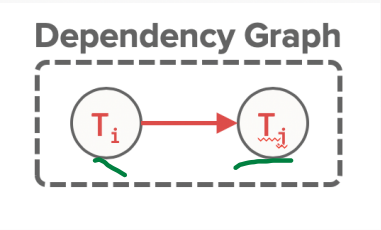


# Serializablity

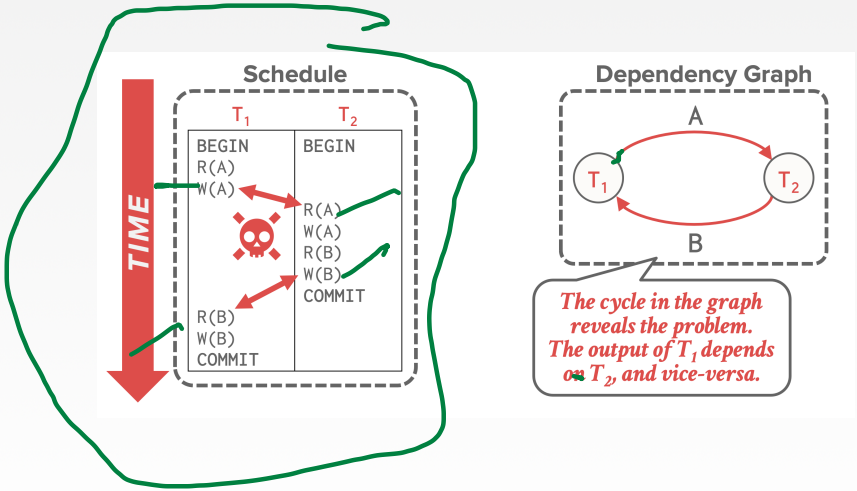
- Swapping operations is easy when there are only two txns in the schedule. It's cumbersome when there are many txns.
- Are there any **faster algorithms** to figure this out other than transposing operations?

# Dependency Graphs

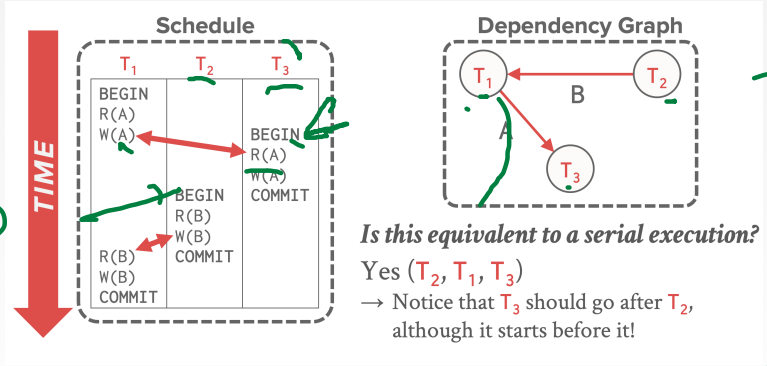
- One node per txn.
- Edge from $T_i$ to $T_j$ if:
  - An operation $O_i$ of $T_i$ conflicts with an operation $O_j$ of $T_j$ and
  - $O_i$ appears earlier in the schedule than $O_j$.
- Also known as a **precedence graph**. A schedule is conflict serializable iff its dependency graph is acyclic.



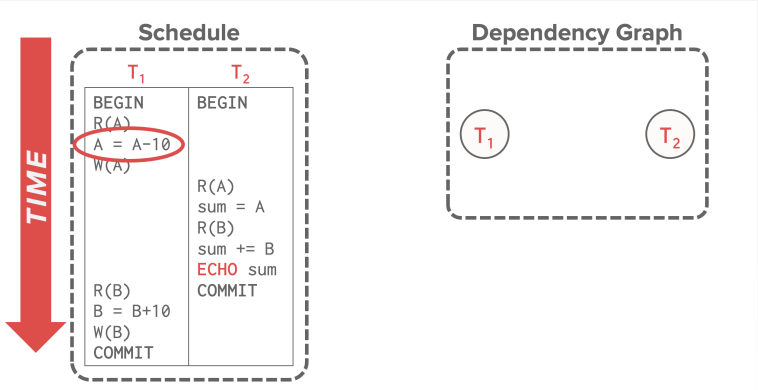

# Example 1


Schedule
T1
T2
BEGIN
R(A)
W(A)
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT
R(B)
W(B)
COMMIT
TIME
Dependency Graph
A
T1
T2
B
The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

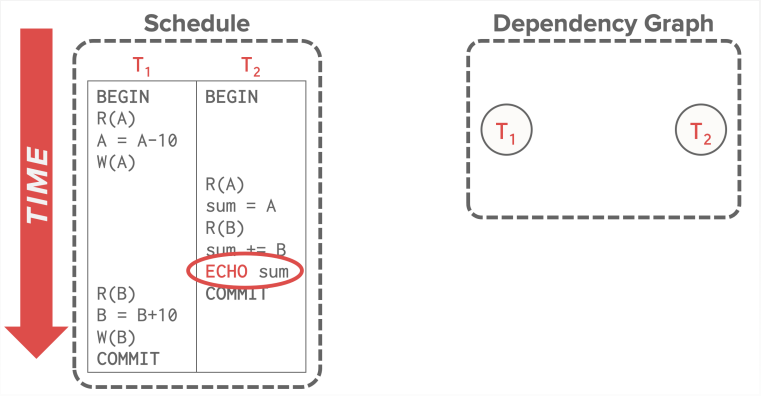
# Example 2


Schedule
TIME
T1
T2
T3
BEGIN
R(A)
W(A)
BEGIN
R(A)
W(A)
COMMIT
BEGIN
R(B)
W(B)
COMMIT
R(B)
W(B)
COMMIT
Dependency Graph
T1
T2
T3
B
A
Is this equivalent to a serial execution?
Yes (T2, T1, T3)
→ Notice that T3 should go after T2, although it starts before it!

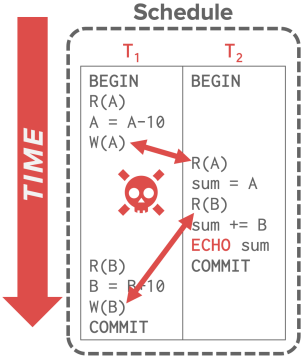
# Example 3 – Inconsistent Analysis

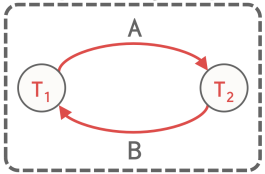


# Example 3 – Inconsistent Analysis



# Example 3 – Inconsistent Analysis



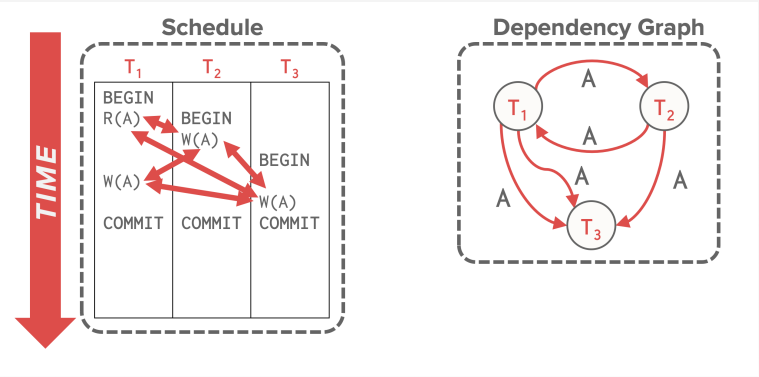

*Is it possible to modify only the application logic so that schedule produces a "correct" result but is still not conflict serializable?*

# View Serializability

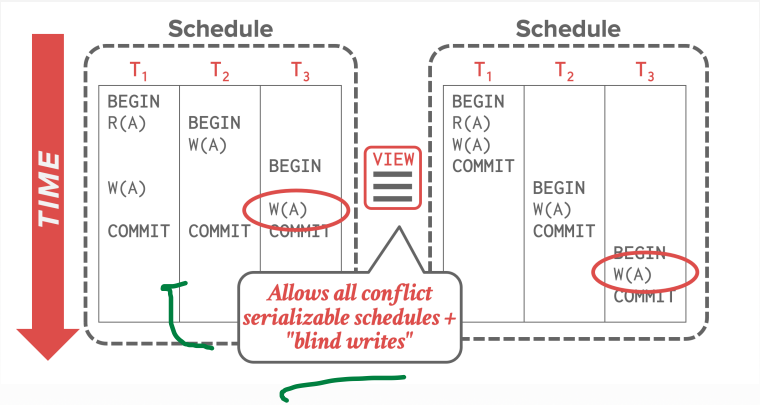
- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are view equivalent if:
  - If $T1$ reads initial value of A in S1, then $T1$ also reads initial value of A in S2.
  - If $T1$ reads value of A written by $T2$ in S1, then $T1$ also reads value of A written by $T2$ in S2.
  - If $T1$ writes final value of A in S1, then $T1$ also writes final value of A in S2.

Blind writes

# View Serializability



# View Serializability



# Serializability

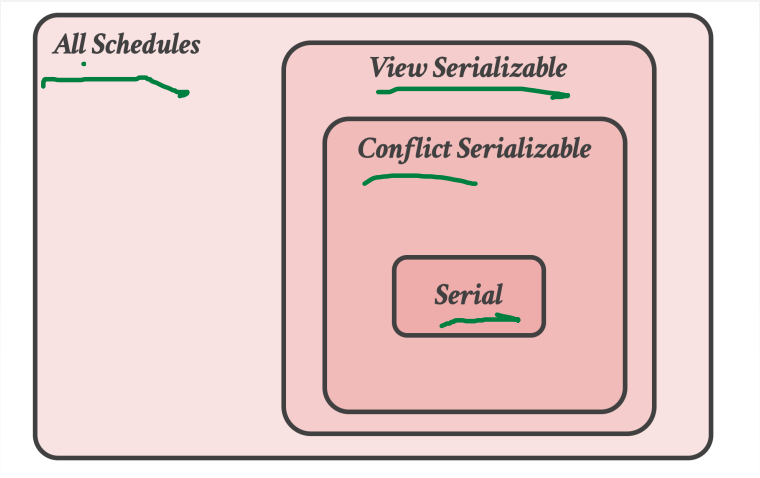
- **View Serializability** allows for (slightly) more schedules than **Conflict Serializability** does.
  - But is difficult to enforce efficiently.
- Neither definition allows all schedules that you would consider "serializable".
  - This is because they don't understand the meanings of the operations 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.

CRDTs

# Universe of Schedules



# Conclusion

# 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.

# Parting Thoughts

- 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