Database Management Systems

Transaction Management

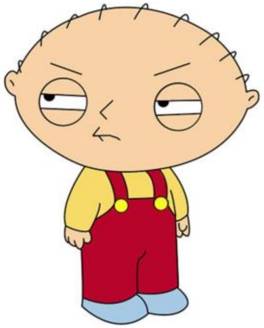
M. Emre Gürsoy

Assistant Professor
Department of Computer Engineering
www.memregursoy.com

Motivation

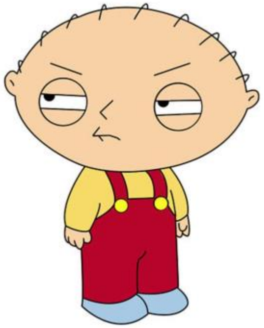
- Consider the database of a banking application
  - Multiple concurrent users and processes: withdrawals from ATM, mobile payments, money transfers...

- Consider that Peter wants to send 100 TL to Stewie
  - Read money in Peter's account (a)
  - a = a – 100; write it to the database
  - Read money in Stewie's account (b)
  - b = b + 100; write it to the database
- Simultaneously, the Griffin Family Fund wants to calculate the total money Peter Griffin and Stewie Griffin have



400 TL

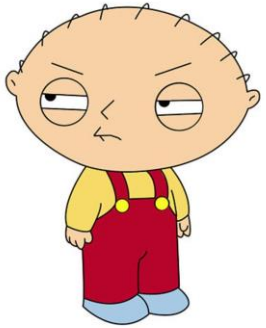
100 TL



100 TL

300 TL

100 TL



100 TL

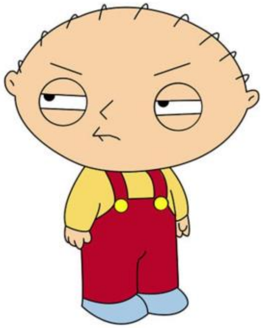
300 TL → 200 TL

Griffin Family Fund

500 TL

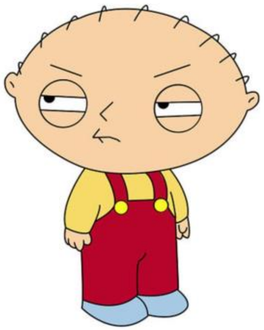
Things are fine if:

- Money is transferred
- Afterwards, the sum is calculated



400 TL

100 TL



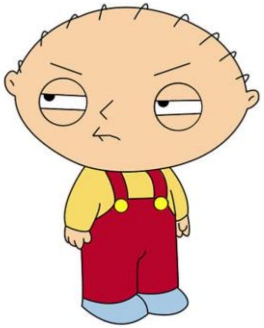
100 TL

300 TL → 100 TL

Griffin
Family
Fund

400 TL

But what if the concurrent transaction is scheduled right before the money is credited to Stewie's account?



100 TL

300 TL

100 TL

How much money should be in their account after the system is restored?

Transactions

- A **transaction** is the execution of a sequence of one or more operations (e.g., SQL queries) on a DB to perform some higher-level function.
  - Money transfer is a higher-level function.
  - Summation of funds is a higher-level function.
- Transaction = one logical unit of work
- Technically, a transaction contains a sequence of reads and writes: R(A), W(A), R(B), W(C), ...
  - A,B,C: database objects
  - DBMS's abstract view of a high-level program/function
- A transaction starts with BEGIN

- A transaction ends with COMMIT or ABORT (aka ROLLBACK)

Strawman Approach

- Say that we have two transactions:
  - **T1:** Peter -> Stewie money transfer
  - **T2:** Griffin Family Fund summation
  - Execute T1 first, then T2
- The strawman approach:
  - In general, when you have N transactions, execute them in **serial order** (one-by-one).
  - Before each transaction BEGINs, copy the whole DB to a new file and make all changes on that file.
    - If transaction finishes successfully and COMMITs, copy the file's contents to the DB.
    - If transactions ABORT, just delete the dirty copy.

Strawman Approach

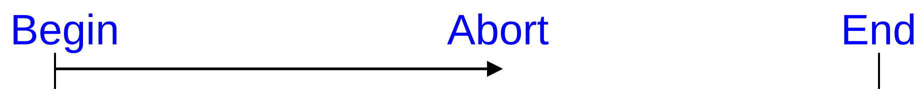
- What's bad about the strawman approach?
  - No parallelism, no multi-threading
    - Some transactions can be parallelized
    - **T1**: R(A), W(A)   **T2**: R(B), W(B)
  - Low throughput and increased response times to users
  - Copying large volumes of data back and forth
- We certainly want to do things smarter than the strawman approach, but...
  - We want our results to be correct
  - We want our data to be safe

ACID Properties

- **Atomicity**
  - "All or nothing": Either all actions in a transaction happen, or none happen.
- **Consistency**
  - If the DB is initially consistent, after a valid transaction, the DB should again be consistent.
- **Isolation**
  - "As if alone": Execution of one transaction is isolated from the executions of other transactions.
- **Durability**
  - "The results stay": If a transaction commits, its effects persist (despite OS crashes, electricity outages, ...).

Atomicity

- From the user's point of view: transactions always execute either all actions, or execute no actions at all.



- Approach #1: **Logging**
  - Log all actions, undo the actions of aborted txns
  - Used in almost every DBMS
  - Also good for auditing
- Approach #2: **Shadow Paging**
  - DBMS makes copies of pages and txns make changes to those copies. Only when the txn commits, the page is made visible to others.
  - Very few DBMS systems do this (it's quite slow)

Isolation

- Users submit transactions, and each transaction executes as if it was running by itself.
- We can achieve this using the strawman approach: run transactions one-by-one, in serial order.
  - But that's inefficient
- The DBMS achieves concurrency by interleaving the actions (reads/writes of DB objects) of transactions.
  - We need a way to interleave transactions but still make it appear as if they ran one at a time.
- The task of finding a proper interleaving of operations from multiple transactions is achieved by **scheduling** and **concurrency control**.

Scheduling Example

- Assume at first A and B each have $1000.
- Say that T_1 and T_2 are submitted together (there's no guarantee which one executes first).
- What are the legal outcomes?
  - A = 954, B = 1166 → A+B = 2120
  - A = 960, B = 1160 → A+B = 2120
- Regardless of which executes first, A+B should be 2120.
  - **"The net effect is equal to some serial execution of T1 and T2".**

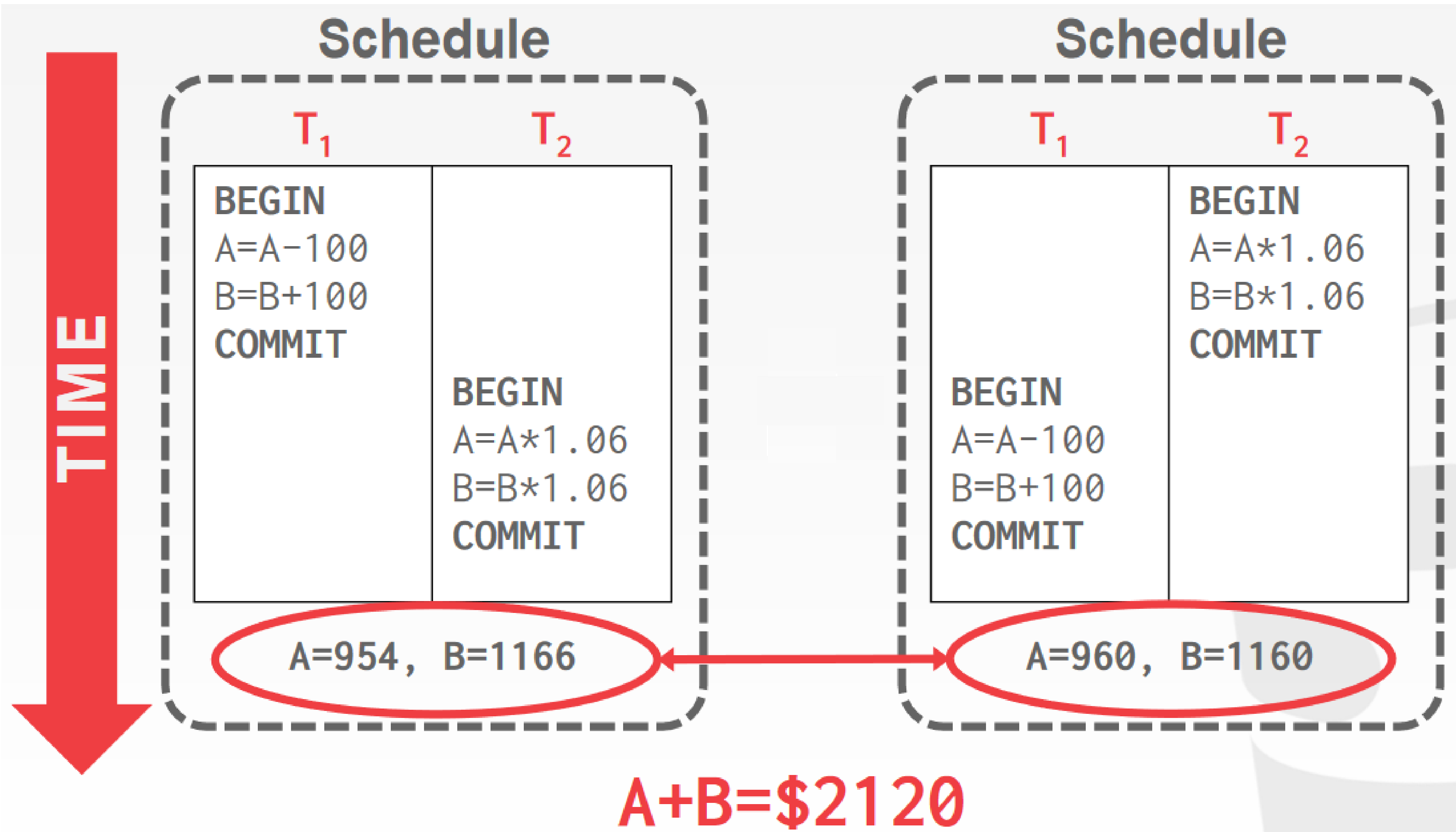
T_1

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

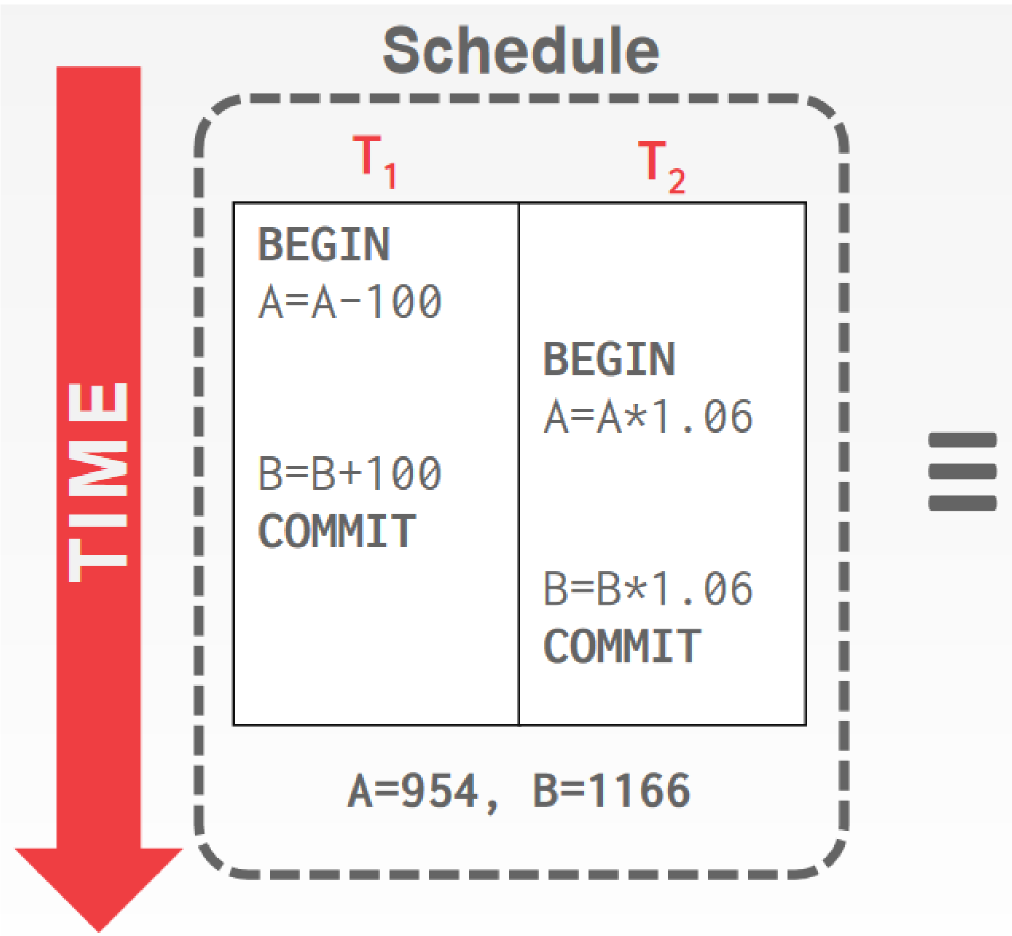
T_2

```
BEGIN
A=A*1.06
B=B*1.06
COMMIT
```

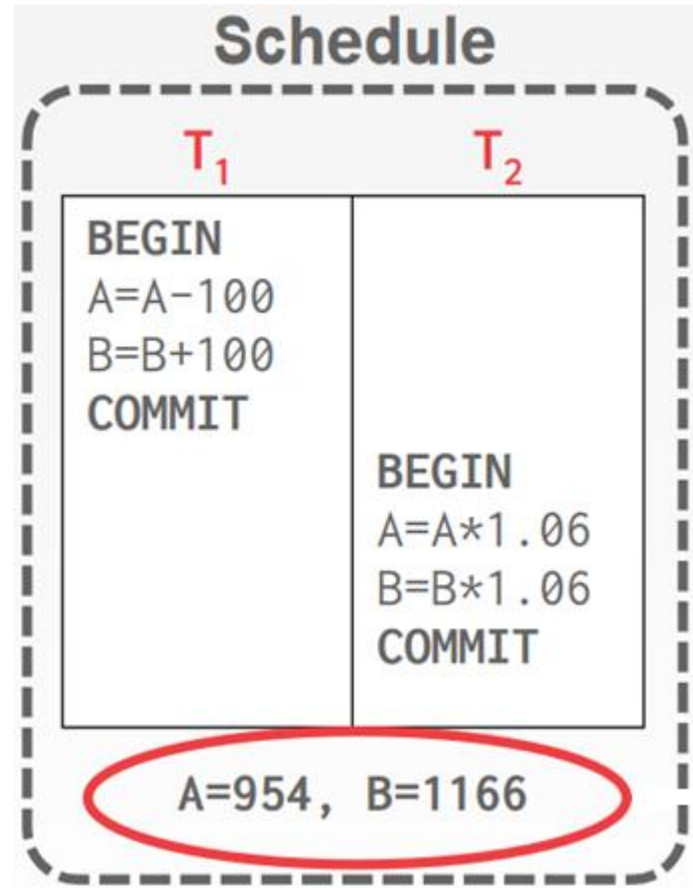
Serial Execution



Interleaving Execution (Good)

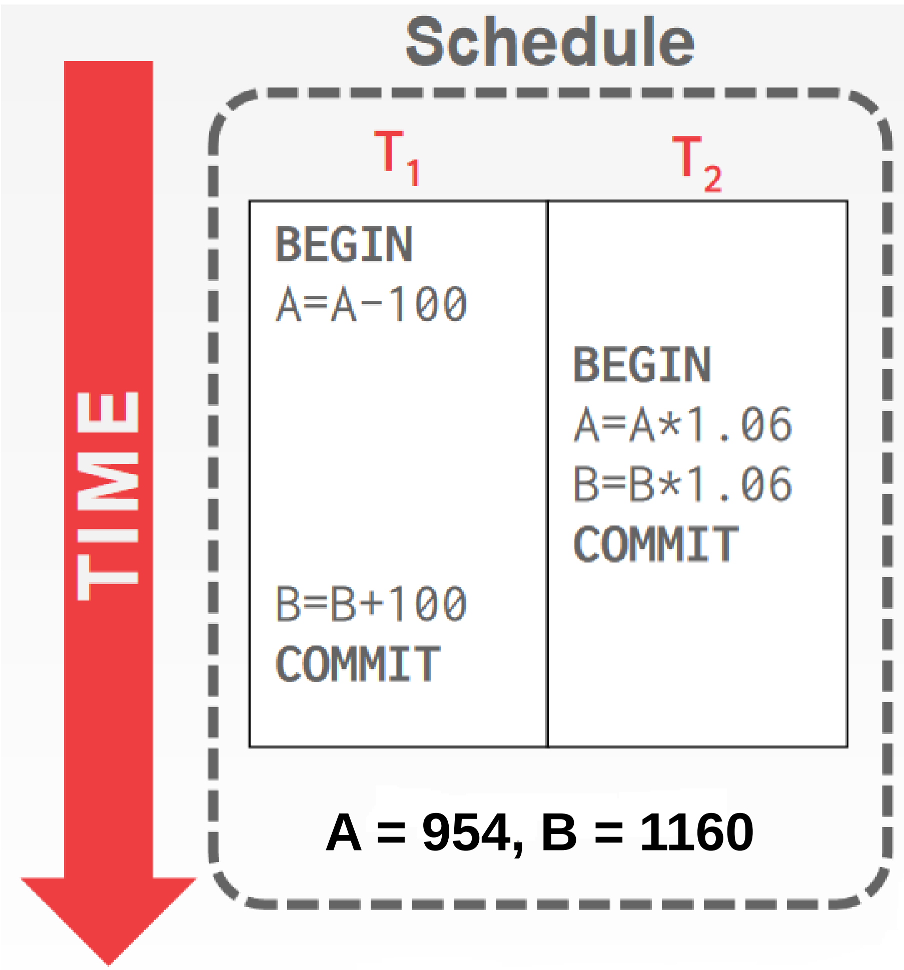


Interleaving Execution



Serial Execution

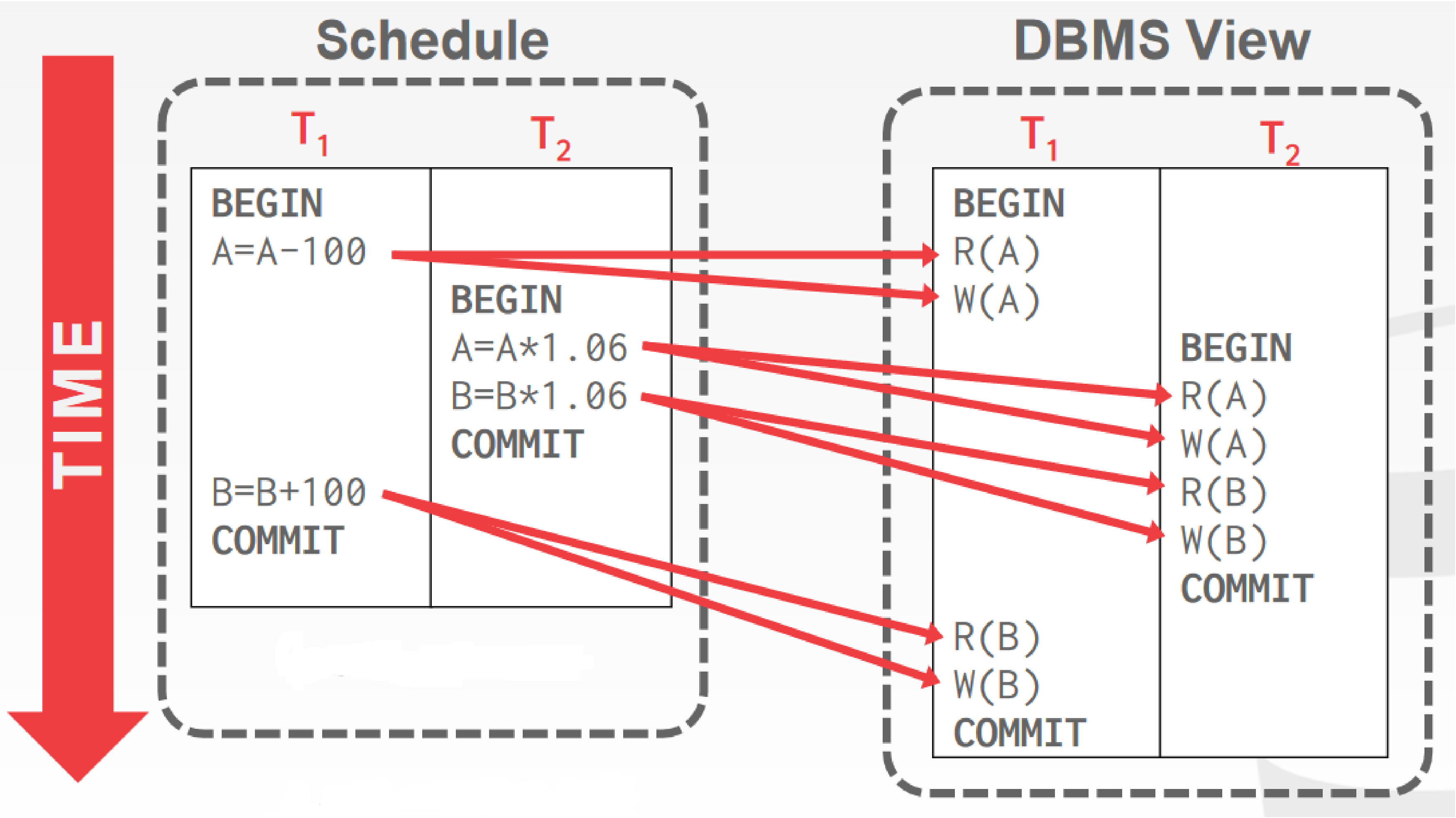
Interleaving Execution (Bad)



≠ A=954, B=1166
or
A=960, B=1160

A+B = $2114, the bank is missing $6!

DBMS Perspective



Schedules

- **Serial schedule:** A schedule that does not interleave the actions of different transactions.

- **Equivalent schedules:** Two schedules are equivalent if 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.

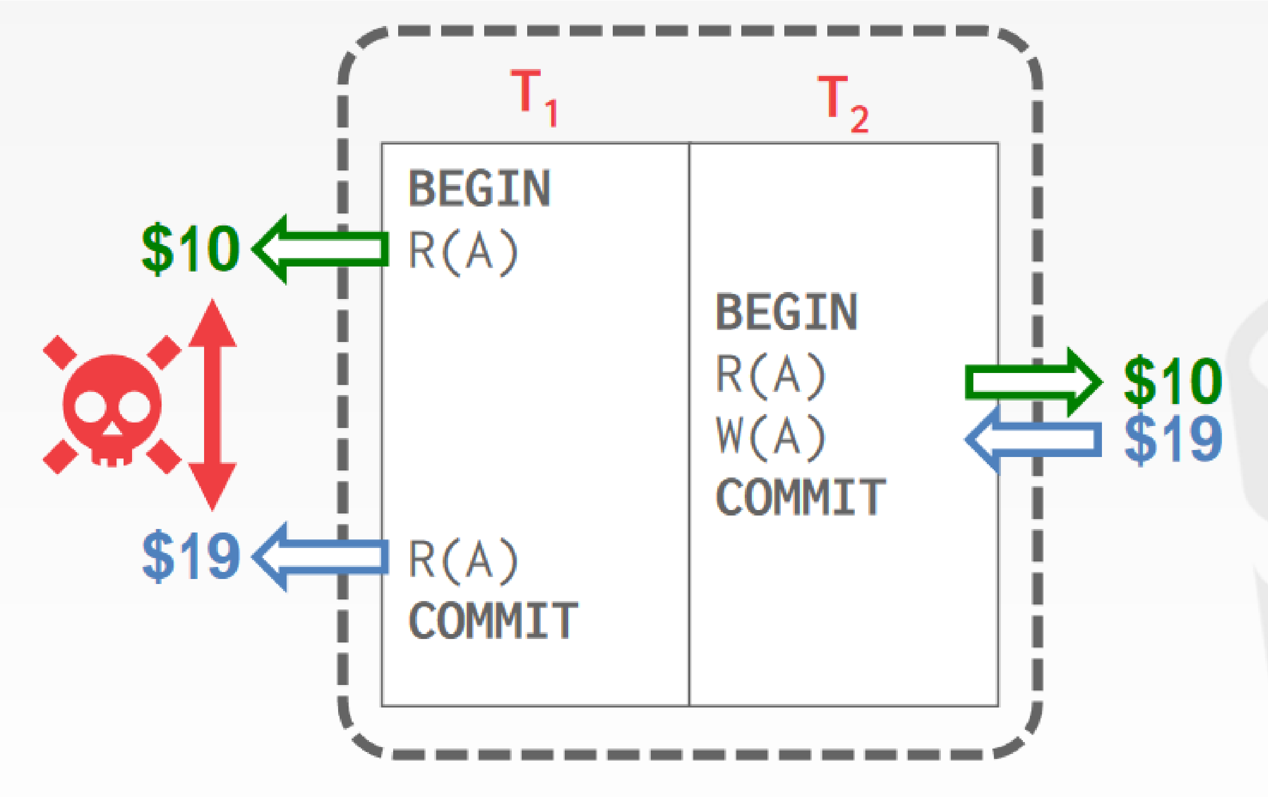
Conflicting Operations

- Two operations **conflict** if:
  - They are by different transactions
  - They are on the same object and at least one of them is a write operation
    - **Why do multiple reads not cause a problem?**

- Types of **problems** that conflicts can lead to:
  - Read-Write Problems **(R-W)**
  - Write-Read Problems **(W-R)**
  - Write-Write Problems **(W-W)**

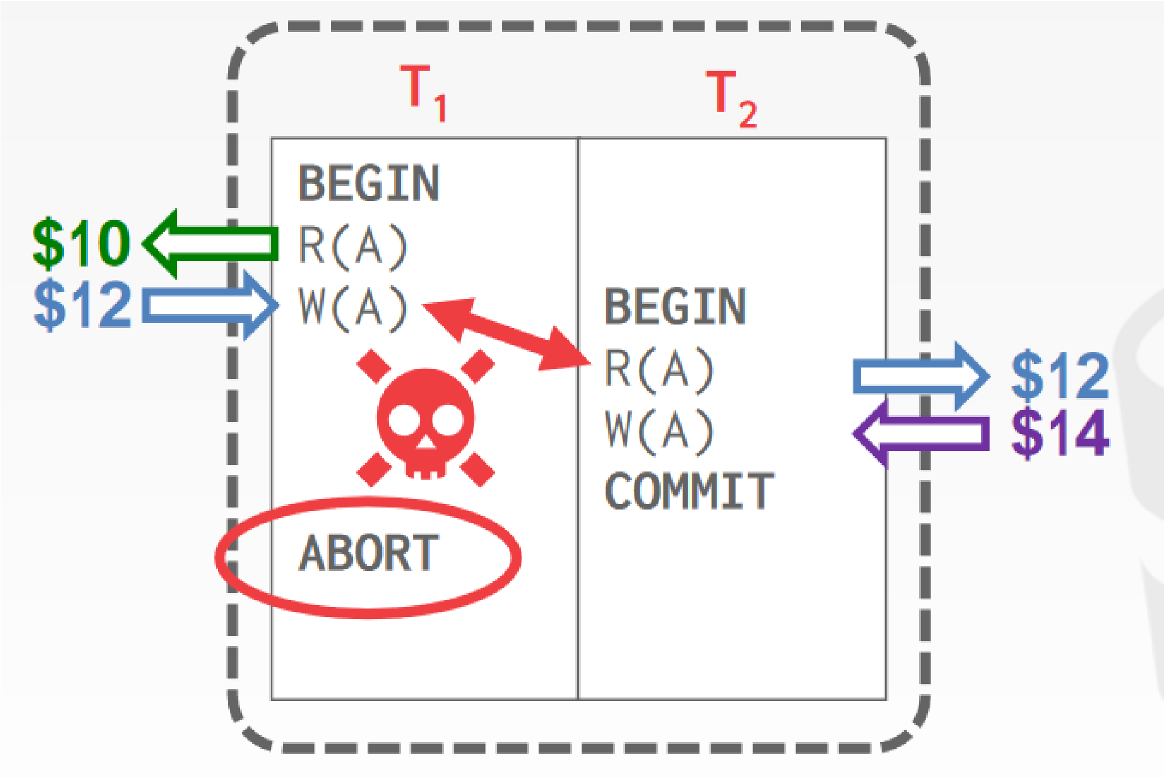
R-W Problems

- "Unrepeatable reads"



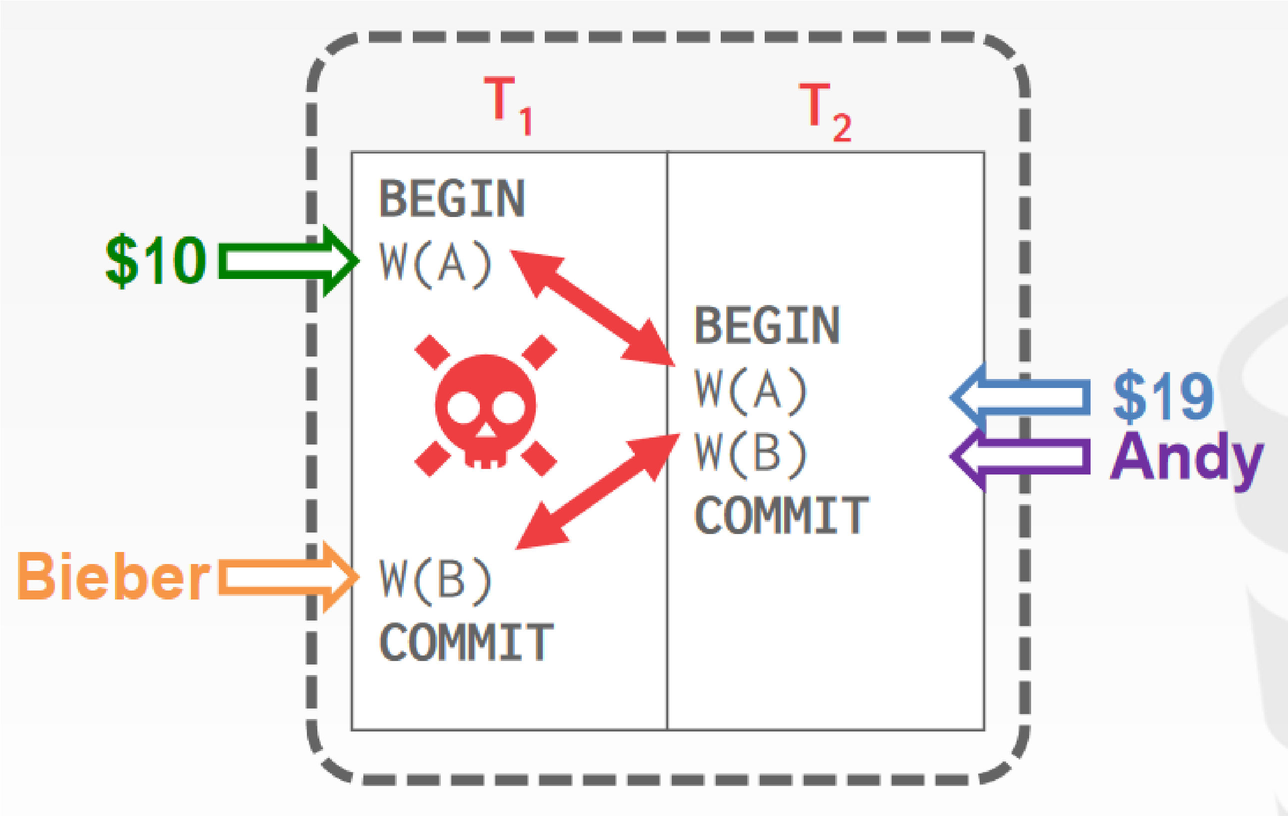
W-R Problems

- "Dirty reads": reading uncommitted data



W-W Problems

- Overwriting uncommitted data



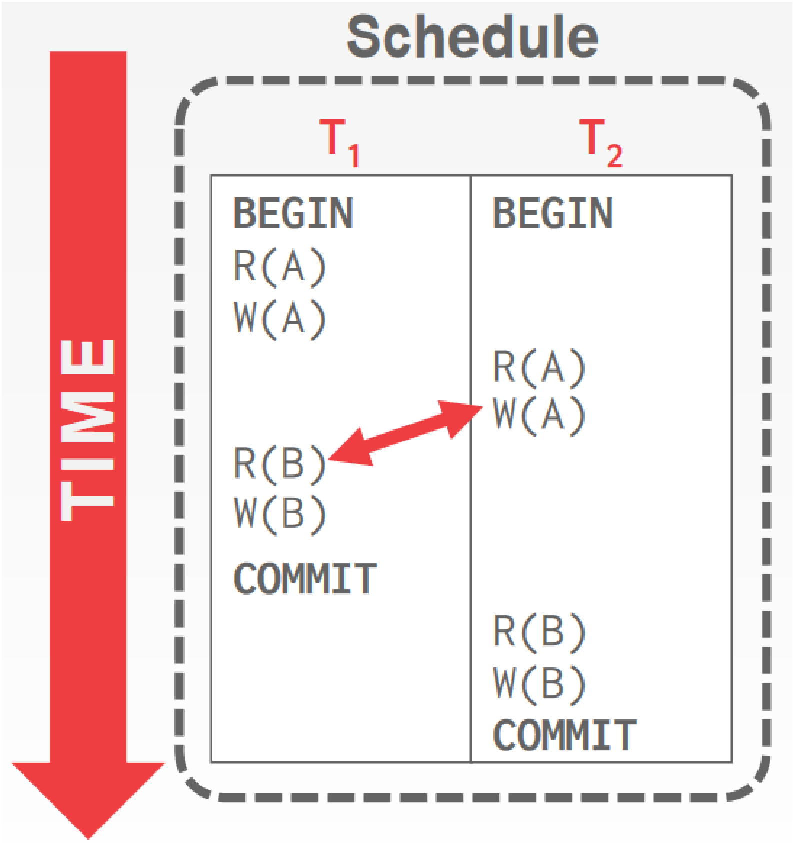
Conflict Serializability

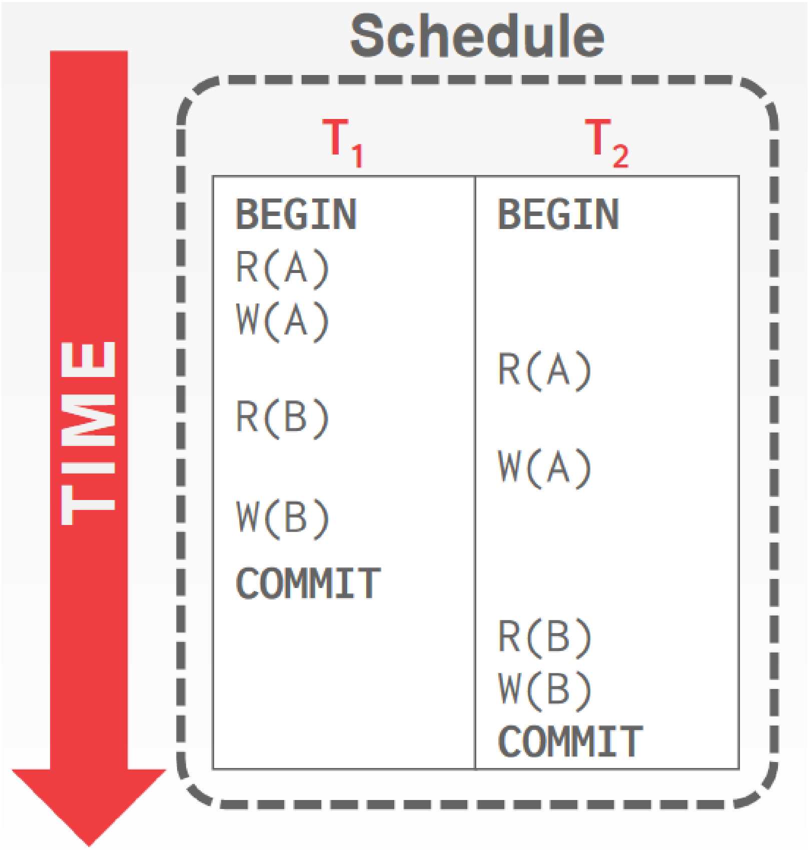
- Two schedules are **conflict equivalent** iff:
  - They involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Schedule S is **conflict serializable** if:
  - S is conflict equivalent to some serial schedule

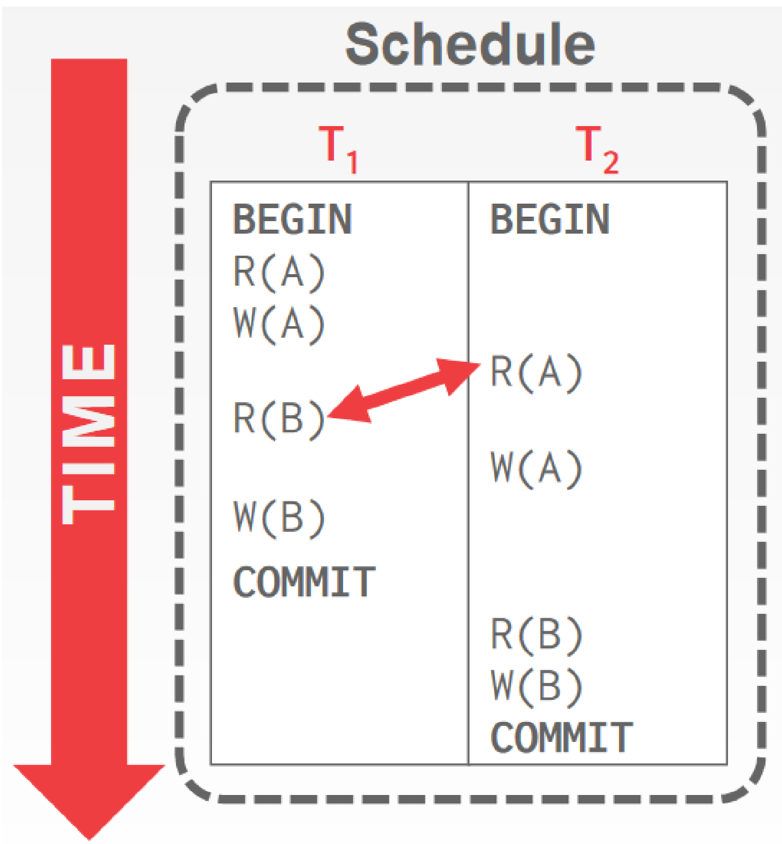
- **Swapping method**: to check conflict serializability
  - Schedule S is conflict serializable if you can transform S into a serial schedule by swapping the order of consecutive non-conflicting operations of different transactions.

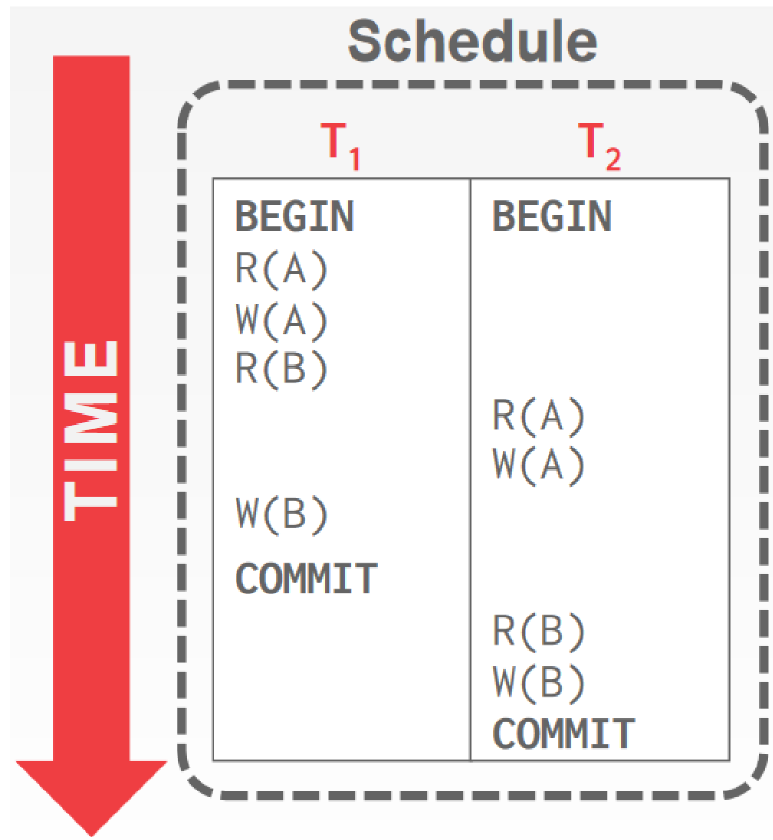
Swapping Method



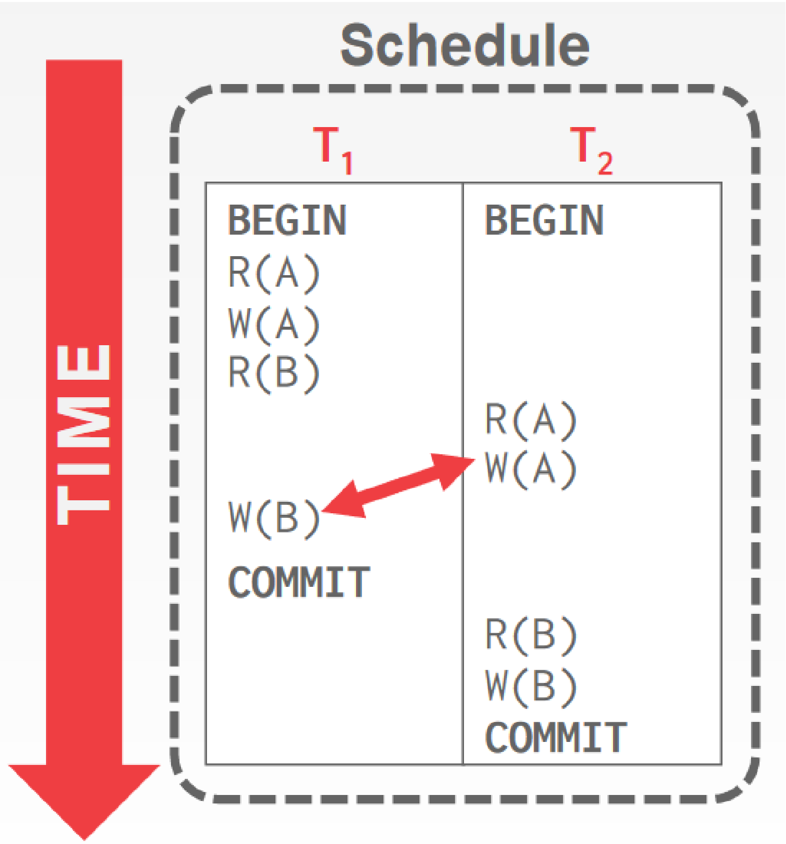


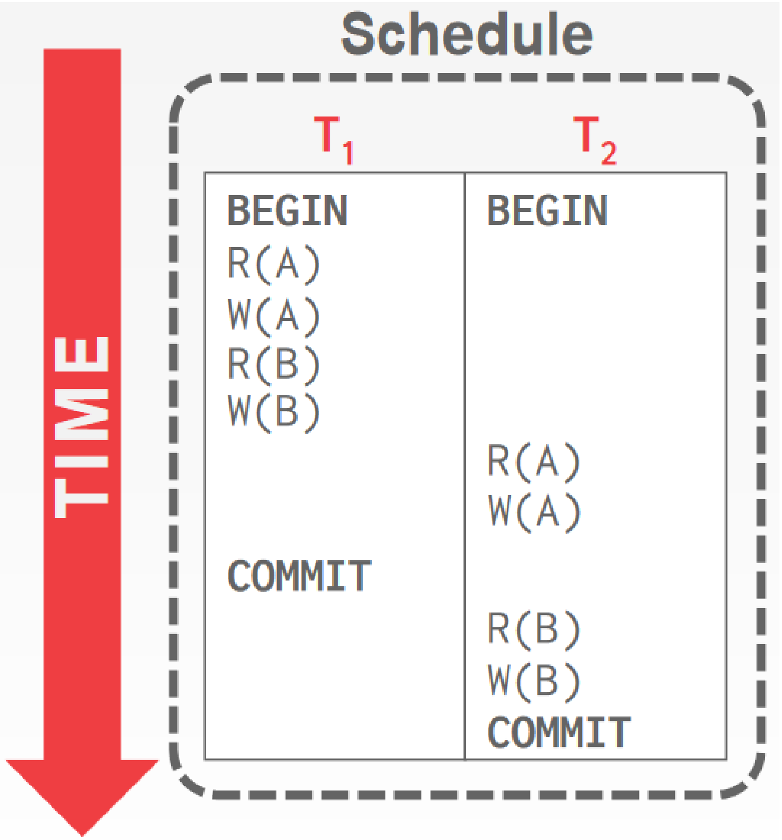
Swapping Method





Swapping Method





Fast-forward one more step: comparing W(B) and R(A)

Swapping Method

Original Schedule

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

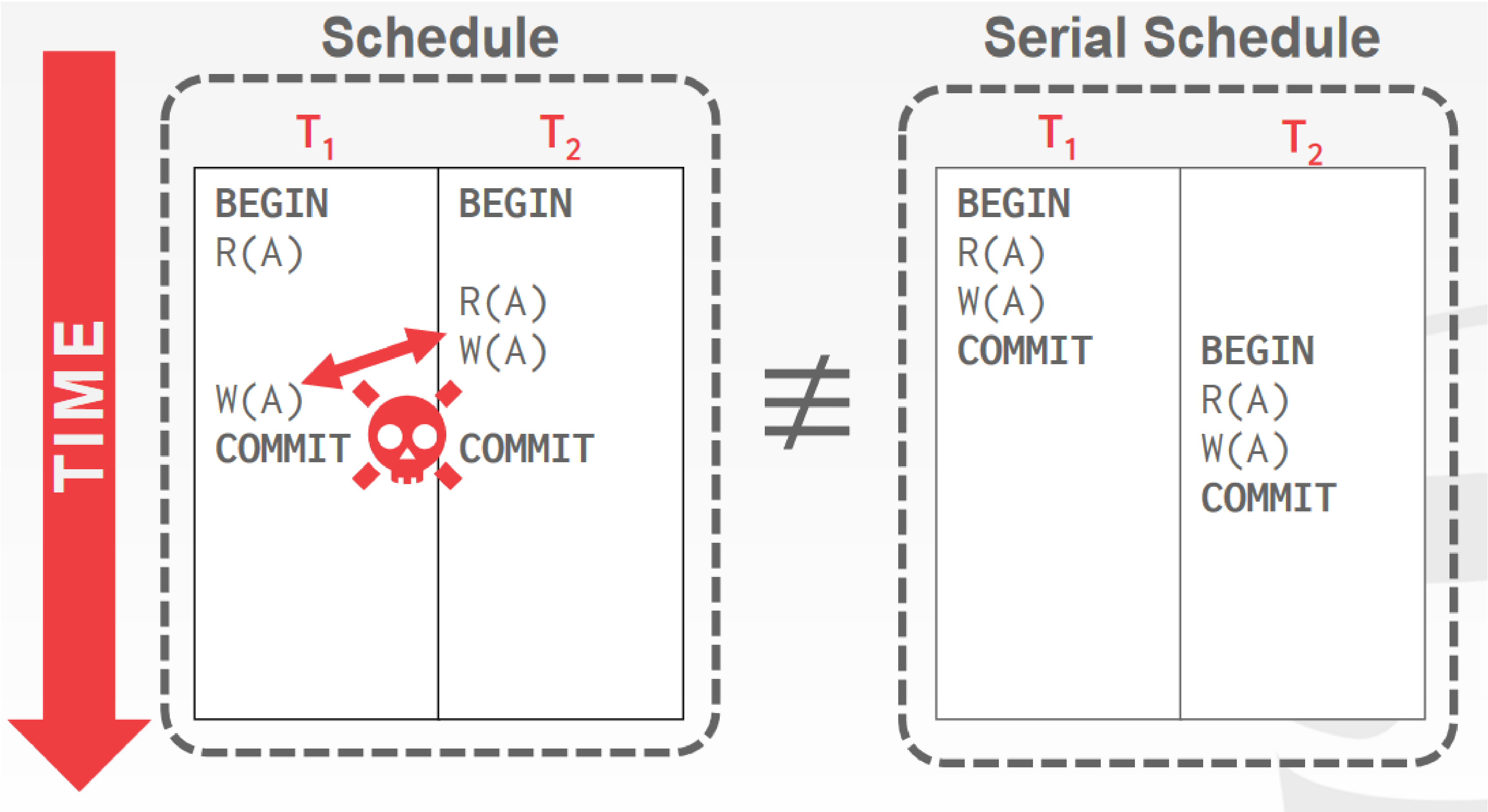
Transformed into:

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 serializable!

Swapping Method



Not conflict serializable!

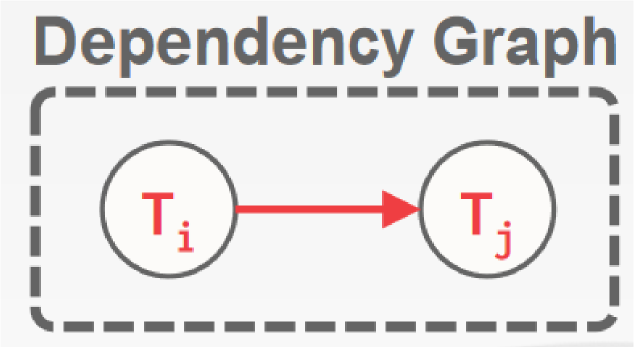
Conflict Serializability

- Swapping method (i.e., checking conflict serializability by moving actions up and down) is feasible when:
  - There are few transactions in the schedule
  - Each transaction contains few actions
- But very cumbersome in complex scenarios!
  - Tens of transactions, hundreds of objects, thousands of reads & writes

- How to solve this problem better?
  - **DEPENDENCY GRAPH**

Dependency Graphs

- One node per transaction
- Edge from T_i to T_j if:
  - An action A_i of T_i conflicts with an action A_j of T_j
  - A_i appears earlier than A_j in the schedule



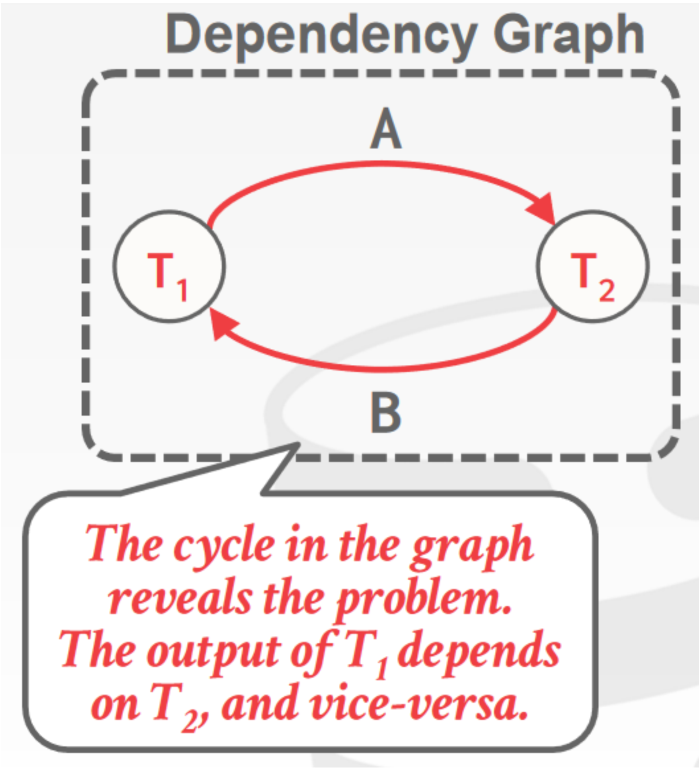
- <u>Theorem:</u> **A schedule is conflict serializable if and only if its dependency graph is acyclic.**

Example #1

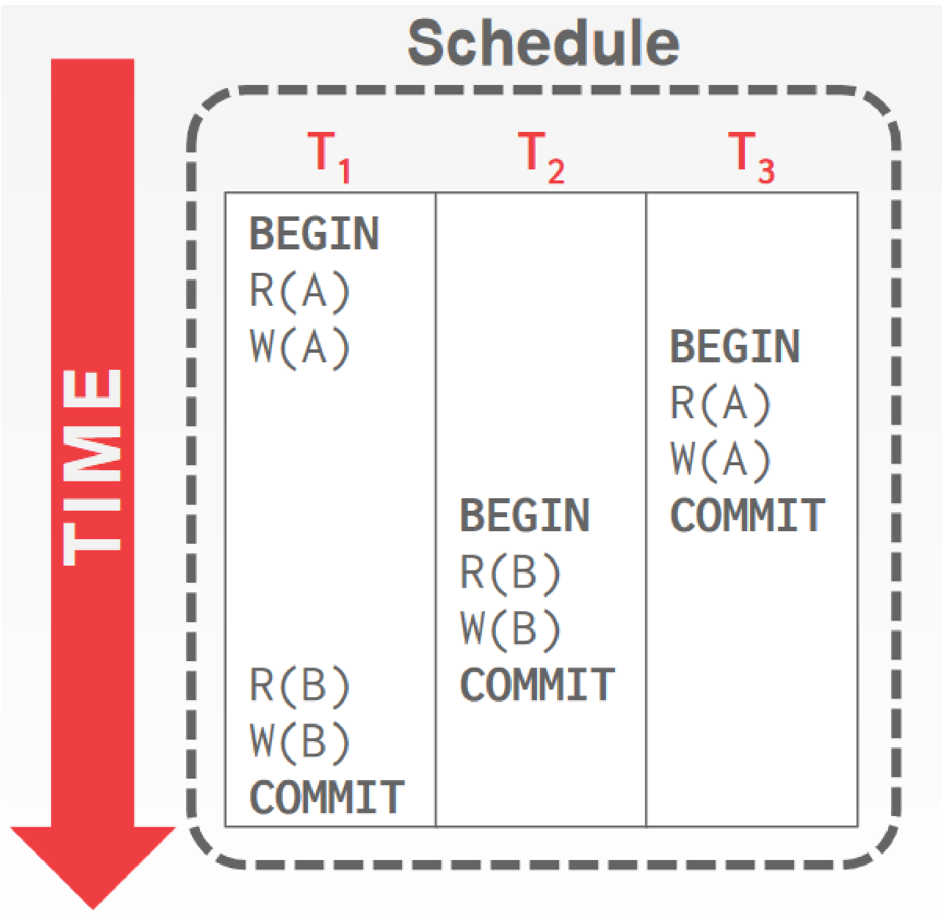
Schedule

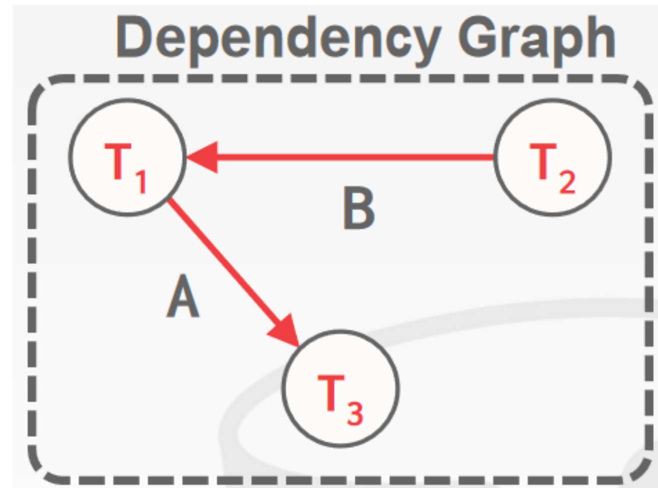
TIME

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



Example #2





Which serial execution is this equivalent to?

T_2, T_1, T_3

Scheduling – Informal Summary

- Serial schedule: transactions executed one-by-one
- Two schedules are equivalent if their net effect is the same
- A schedule (consisting of multiple transactions) is serializable if it is equivalent to <u>some</u> serial execution of the transactions
- Schedules may contain conflicts (conflicting actions)
- Two schedules are conflict equivalent if their conflicting actions are ordered the same way
- A schedule is conflict serializable if it is conflict equivalent to some serial execution of the transactions
- We check conflict serializability using the swapping method and/or the dependency graph