THE UNIVERSITY of EDINBURGH

Advanced Databases

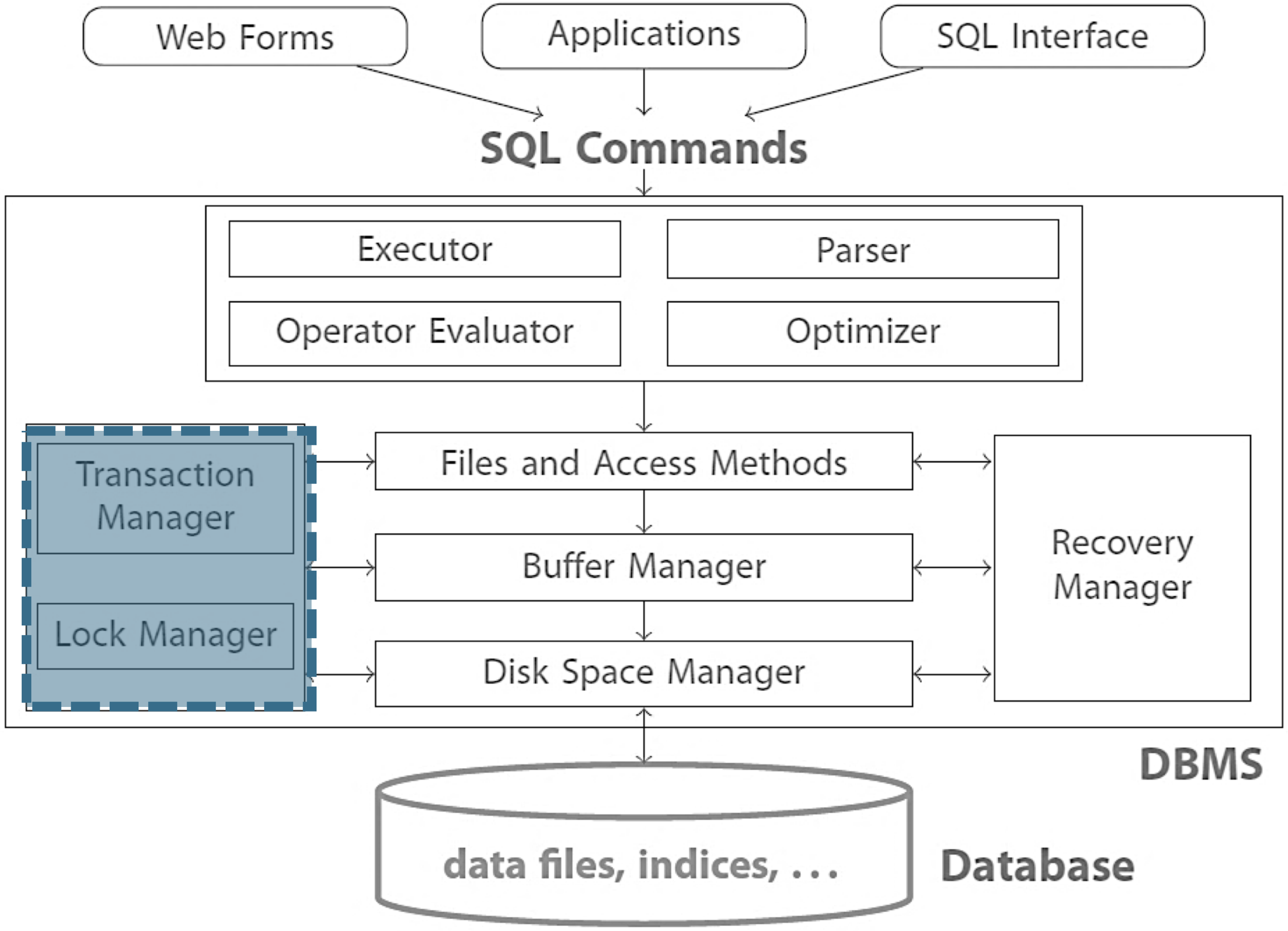
Spring 2020

Lecture #12:

Transactions

Milos Nikolic

Database Architecture



Motivation

We both change the same record in a table at the same time.

How to avoid race condition?

Concurrency Control

You transfer £100 between bank accounts but there is a power failure.

What is the correct database state?

Recovery

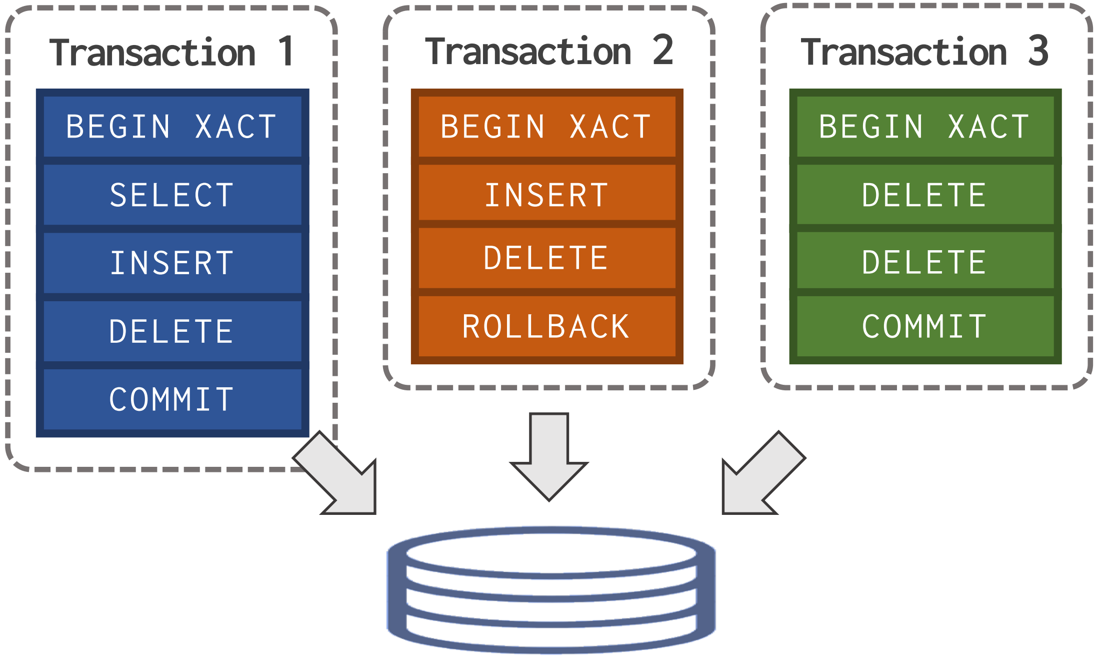
Both concurrency control and recovery are based on a concept of transactions with ACID properties

TRANSACTIONS

A **transaction** is the execution of a sequence of operations (e.g., SQL queries) on a shared database to perform some higher-level function

Basic unit of change in a DBMS

Partial transactions are not allowed!



User Perspective: Transactions

Transaction (abbr. txn) = **group of operations** the user wants the DBMS to treat “as one”

A new transaction starts with the **BEGIN** command

The transaction stops with either **COMMIT** or **ABORT** (**ROLLBACK**)

- If commit, all changes are saved
- If abort, all changes are undone (as if the txn never executed at all)
- Abort can be either self-inflicted or caused by DBMS

Transaction Example

Transfer £100 from Checking to Savings account of user 1904

```
BEGIN
  // check if Checking balance > 100
  UPDATE Accounts
     SET balance = balance - 100
   WHERE customer_id = 1904
     AND account_type = ‘Checking’;
  UPDATE Accounts
     SET balance = balance + 100
   WHERE customer_id = 1904
     AND account_type = ‘Savings’;
COMMIT
```

Consistent DB

Temporary inconsistent DB

Consistent DB

Transaction Example

Transfer £100 from Checking to Savings account of user 1904

```
BEGIN
  // check if Checking balance > 100
  UPDATE Accounts
     SET balance = balance - 100
   WHERE customer_id = 1904
     AND account_type = ‘Checking’;
  UPDATE Accounts
     SET balance = balance + 100
   WHERE customer_id = 1904
     AND account_type = ‘Savings’;
COMMIT
```

How to check if balance > 100?

Outside DBMS using another language

E.g., in Java or PHP code

Inside DBMS using **stored procedures** expressed in PL/SQL

PL/SQL = SQL + procedural constructs such as if-then-else, loops, variables, functions…

Database Perspective

A transaction 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 scope of the DBMS

Transactions: Formal Definition

Database = fixed set of named data objects (A, B, C, …)

- Transactions access object A using read A and write A, for short R(A) and W(A)
- In a relational DBMS, an object can be an attribute, record, or table

Transaction = sequence of read and write operations

- T = ⟨ R(A), W(A), W(B), … ⟩
- DBMS's abstract view of a user program

Strawman Execution

Execute each txn **one-by-one** (serial order) as they arrive in 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

SQLite executes transactions in serial order

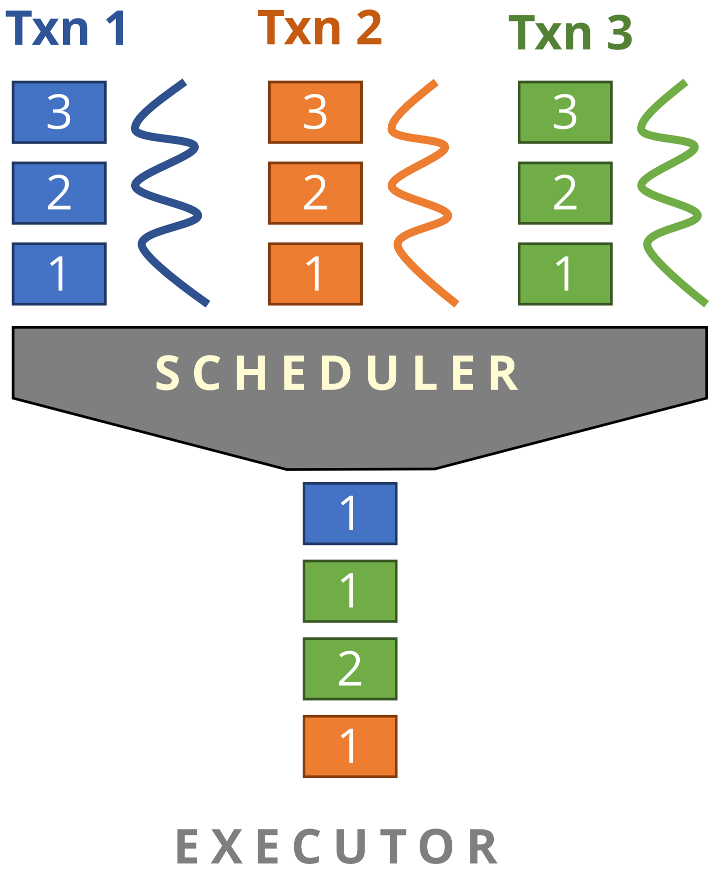
Concurrent Execution

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

Why do we want that?

- Better resource utilization (CPU) and throughput
- Decreased response times to users

But we also would like **correctness** and **fairness**



Transaction Guarantees: ACID

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

"all or nothing"

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

"it looks correct to me"

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

"as if alone"

Durability: If a txn commits, its effects persist

"survive failures"

ACID Properties: Atomicity

Two possible outcome of executing a transaction:

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

DBMS guarantees that transactions are **atomic**

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

Example:

- Take £100 from account A, but then a power failure happens before crediting account B
- ***When the DBMS comes back online, what should be the correct state of the database***?

Mechanisms for Ensuring Atomicity

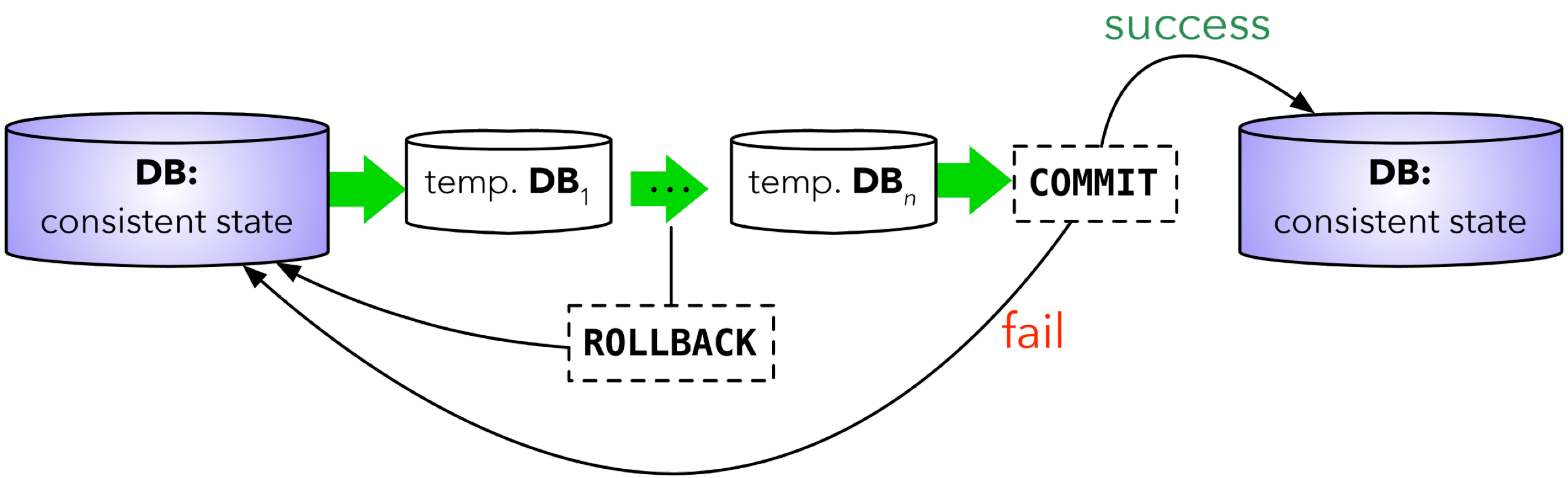
Approach #1: Logging

- DBMS logs all actions so that it can undo the actions of aborted transactions
- Write-ahead logging is used by almost all modern database systems
  - Audit trail & efficiency reasons

Approach #2: Shadow Paging (copy-on-write)

- DBMS makes copies of pages and transactions make changes to those copies
- Only when the transaction commits is the page made visible to others
- Few database systems do this (CouchDB, LMDB)

ACID Properties: 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

Transaction consistency

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

Transaction consistency is the application's responsibility!

ACID Properties: Isolation

Users submit transactions, and each transaction executes as if it was running alone

DBMS achieves concurrency by interleaving actions (read/writes of database objects) of various transactions

How do we achieve this?

Mechanisms for Ensuring Isolation

A concurrency control protocol is how the DBMS decides the proper interleaving of operations from multiple transactions

Two main categories:

- **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 accounts A and B each have £1000

T_1 transfers £100 from A to B

T_2 credits both accounts with 6% interest

T_1

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

T_2

```
BEGIN
A = A * 1.06
B = B * 1.06
END
```

Example

Assume at first accounts **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
END
```

T_2

```
BEGIN
A = A * 1.06
B = B * 1.06
END
```

Example

Assume at first accounts 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

Assume at first accounts A and B each have £1000

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

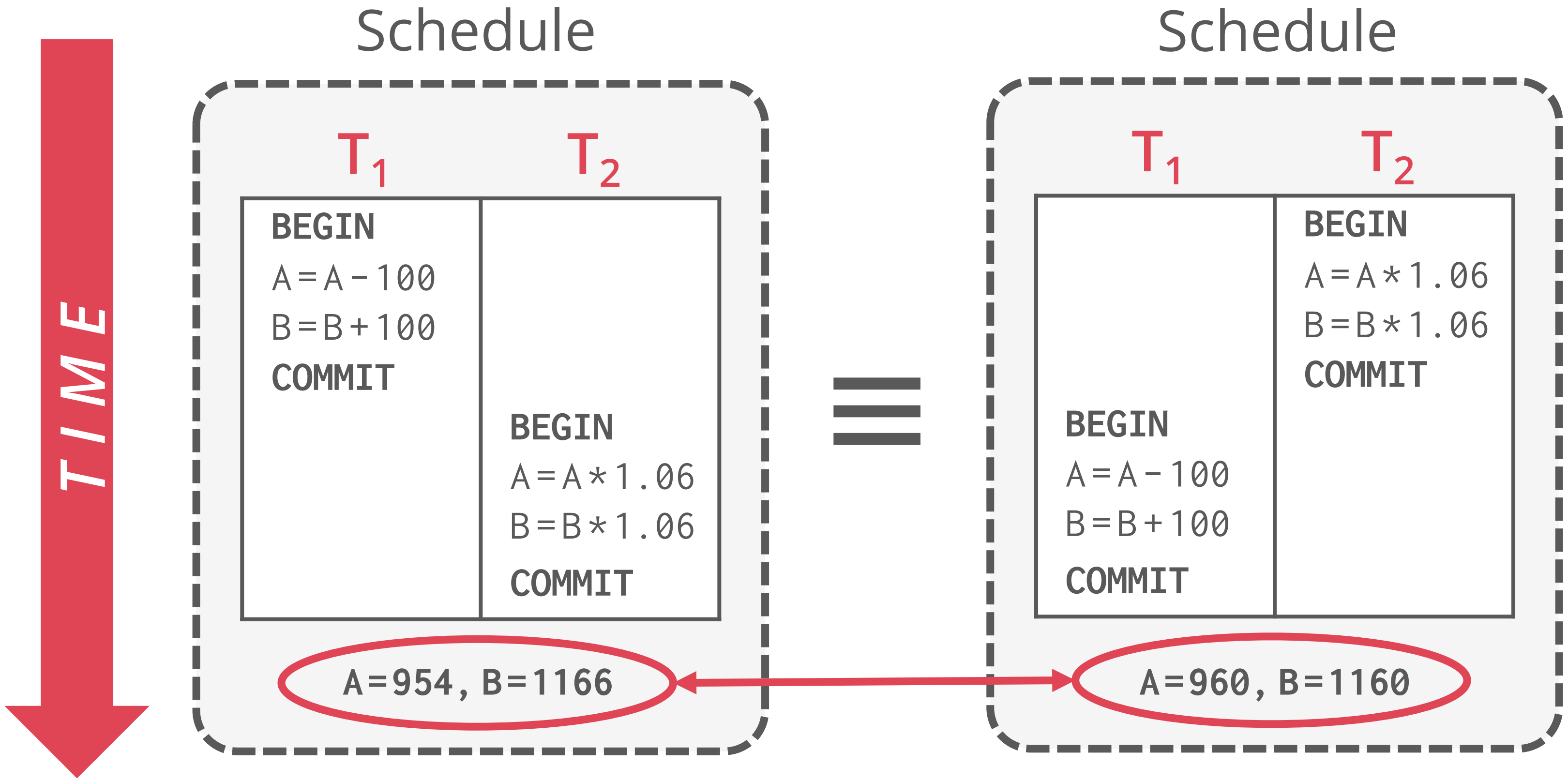
T_1

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

T_2

```
BEGIN
A = A * 1.06
B = B * 1.06
END
```

Example: Serial Execution



A+B = 2120

Interleaving Transactions

We interleave transactions to maximise concurrency

- Slow disk I/O or network
- Multi-core CPU

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

Example: Interleaving (Good)

TIME

Schedule

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

A=954, B=1166

≡

Schedule

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

A=954, B=1166

Example: Interleaving (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

$\neq$

A=954, B=1166

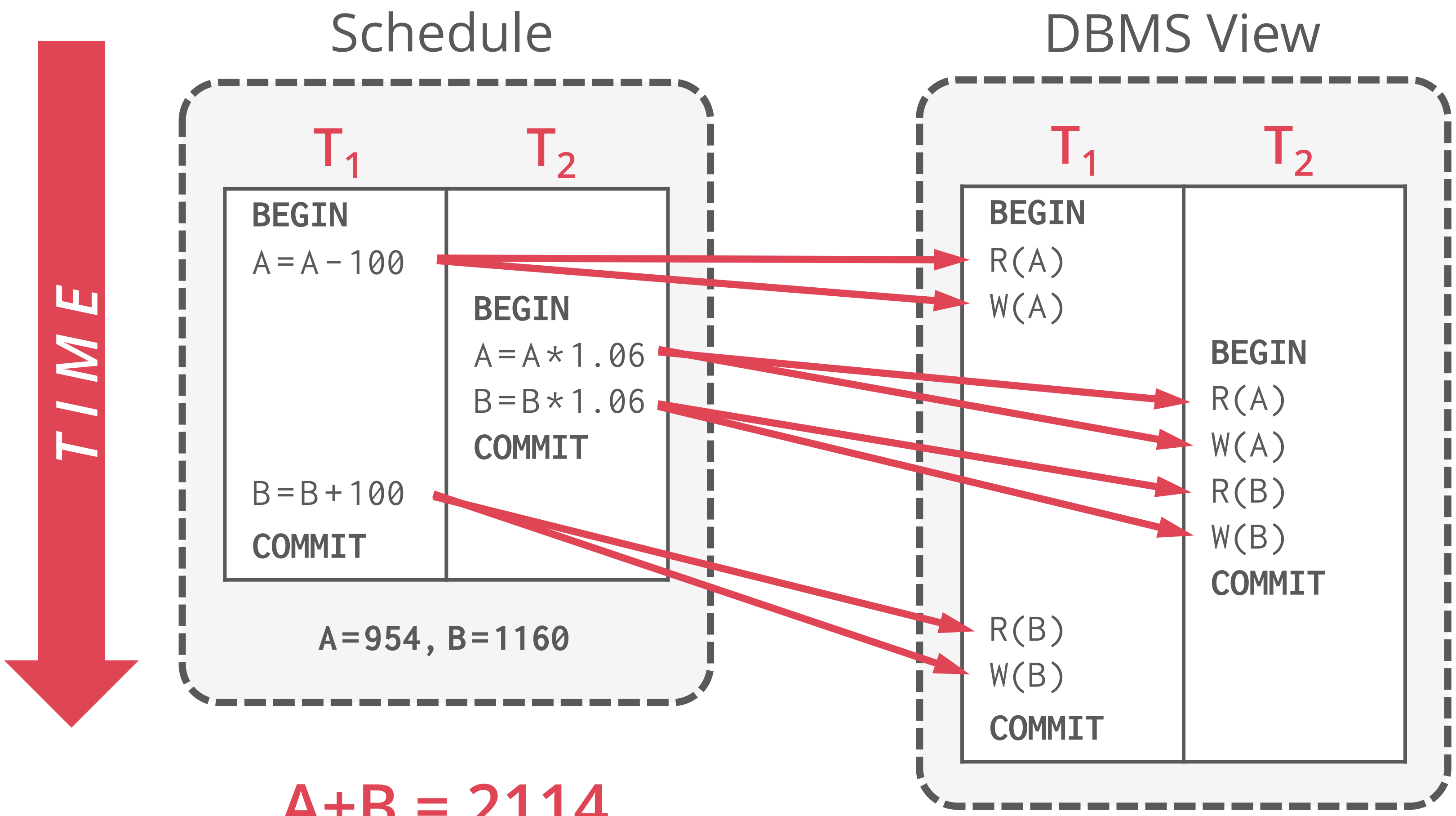
or

A=960, B=1160

A+B = 2114

The bank is missing £6!

EXAMPLE: INTERLEAVING (BAD)



A+B = 2114

CORRECTNESS

How do we judge whether a schedule is correct?

If the schedule is **equivalent** to some **serial execution**

Schedule S for a set of transactions { T_1, … , T_n }

S contains ***all*** steps of all transactions and order among steps in each T_i is ***preserved***

S = ⟨ (T_1, read B), (T_2, read A), (T_2, write B), (T_1, write A) ⟩

for short, **S** = ⟨ R_1(B), R_2(A), W_2(B), W_1(A) ⟩

Formal Properties of Schedules

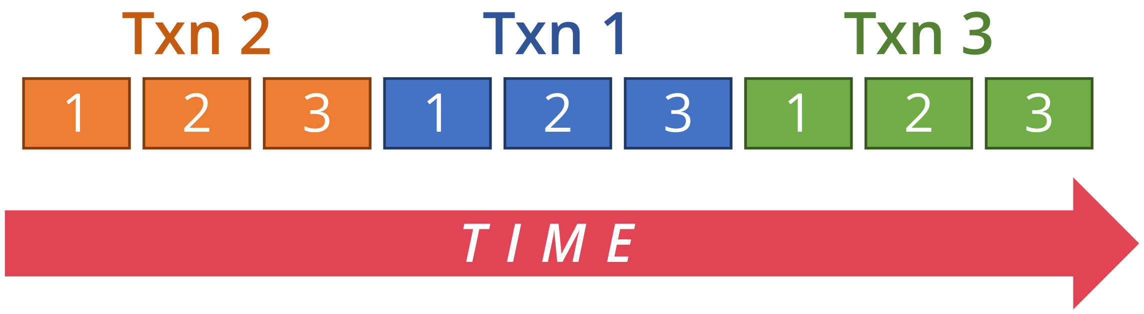
Equivalent schedules

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

Does not matter what the higher-level operations are!

Serial schedule (no concurrency)

A schedule that does not interleave the actions of different transactions



Formal Properties of Schedules

Serializable schedule

A schedule that is equivalent to some serial execution of the transactions

If each transaction preserves consistency, every serializable schedule preserves consistency

Serializability

Less intuitive notion of correctness compared to transaction initiation time or commit order

But it provides the DBMS with 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

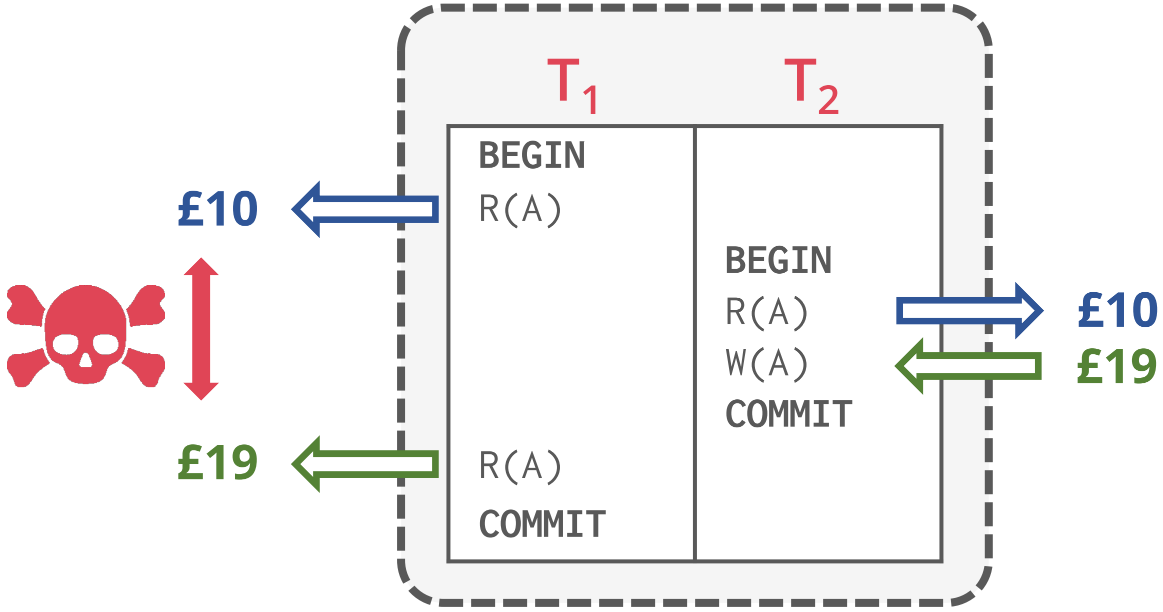
- They are by different transactions
- They are on the same object and at least one of them is a write

Interleaved execution anomalies:

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

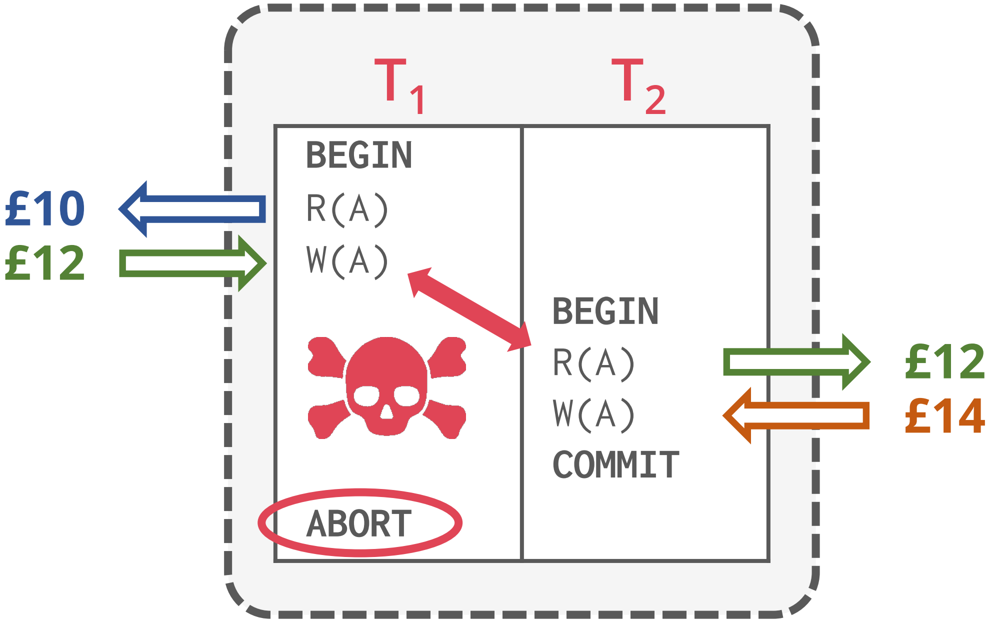
Read-Write Conflicts

Unrepeatable Reads



Write-Read Conflicts

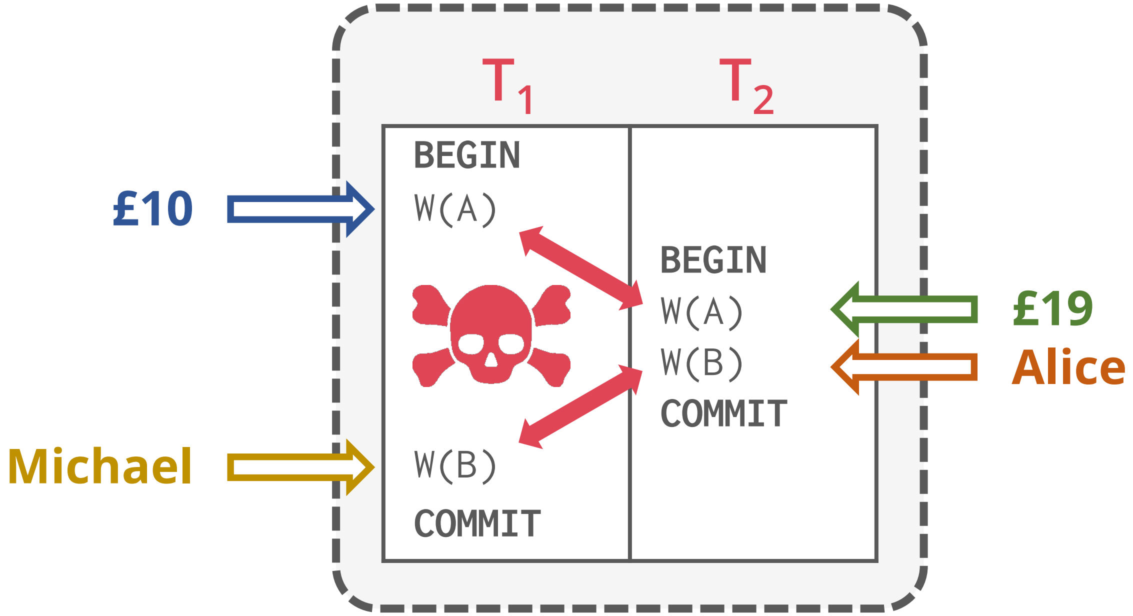
Reading Uncommitted Data ("Dirty Reads")



Not recoverable

WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data ("Lost Update")



Formal Properties of Schedules

Given these conflicts, we can now 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 levels of serializability

- Conflict Serializability
- View Serializability

Most DBMS try to support this

No DBMS supports this

Conflict Serializable Schedules

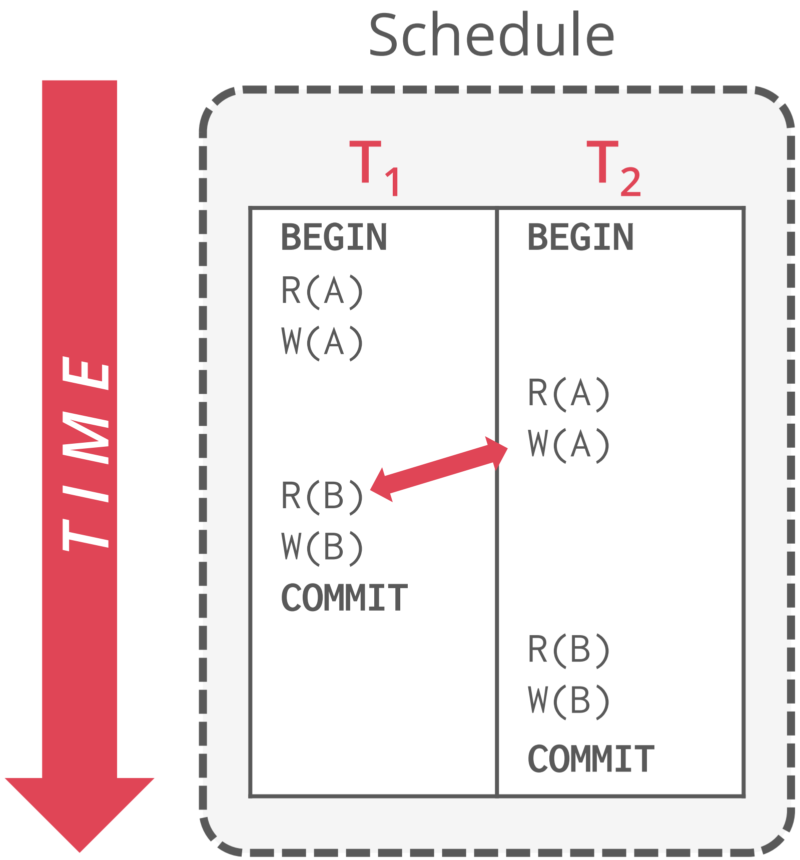
Two schedules are **conflict equivalent** iff

- They involve the same actions of the same transactions
- Every pair of conflicting actions is ordered in the same way

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

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

Conflict Serializability: Intuition



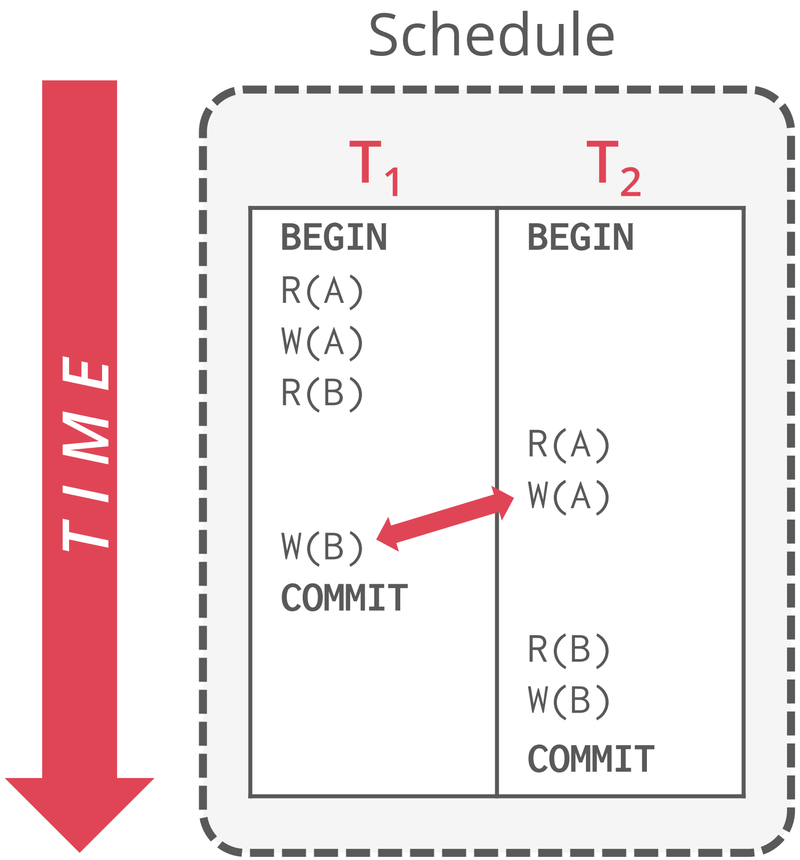
Conflict Serializability: Intuition

Schedule

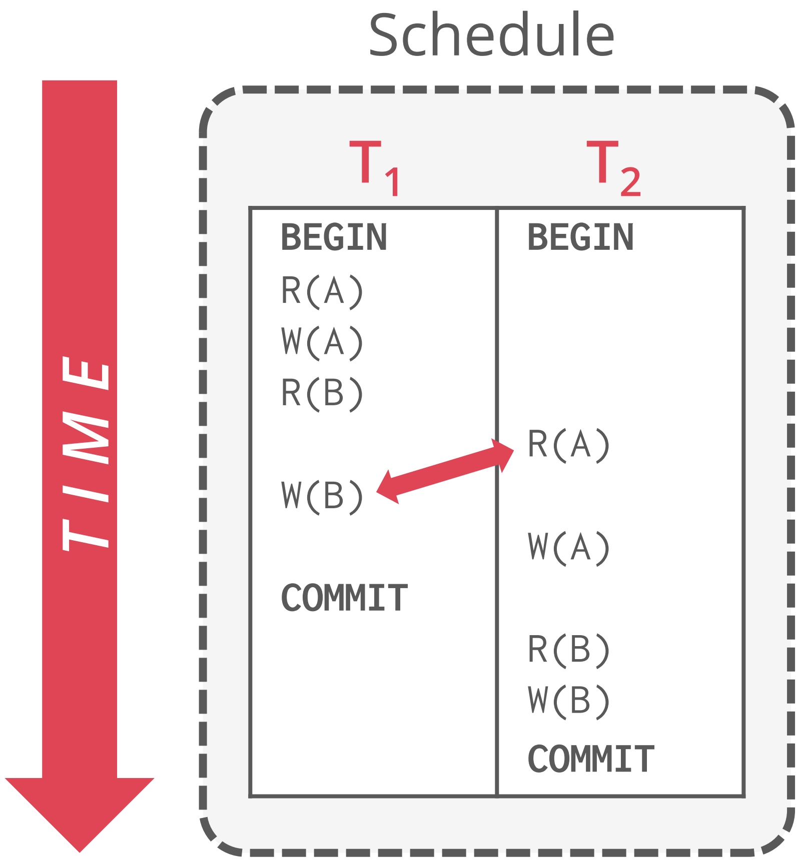
TIME

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



Conflict Serializability: Intuition



Conflict Serializability: Intuition

TIME

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

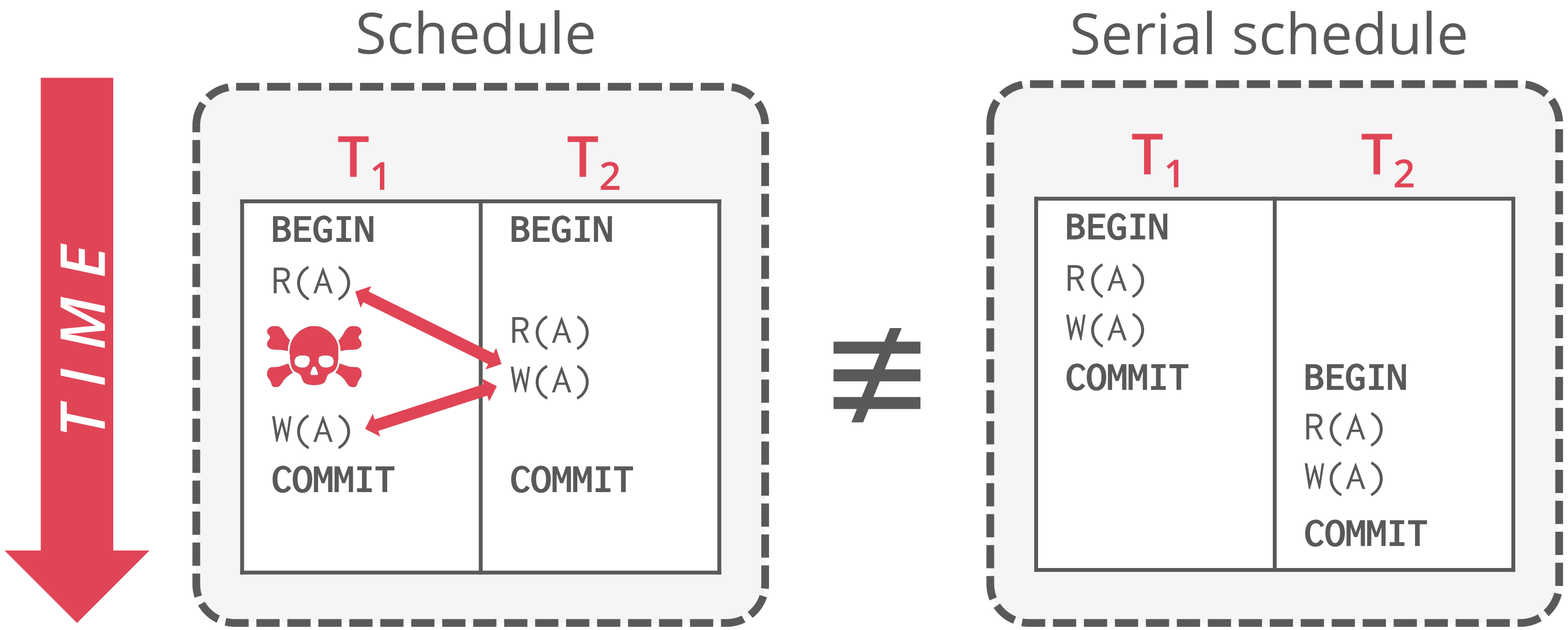
Serializable

$\equiv$

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



Not conflict-serializable

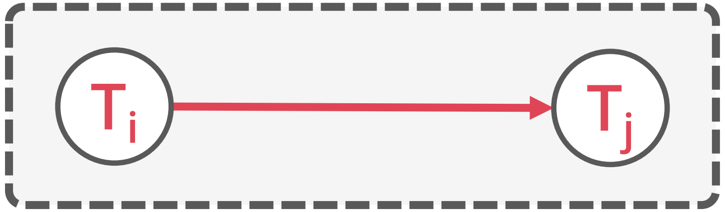
Serializability

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

Dependency Graph



Dependency graph for a schedule

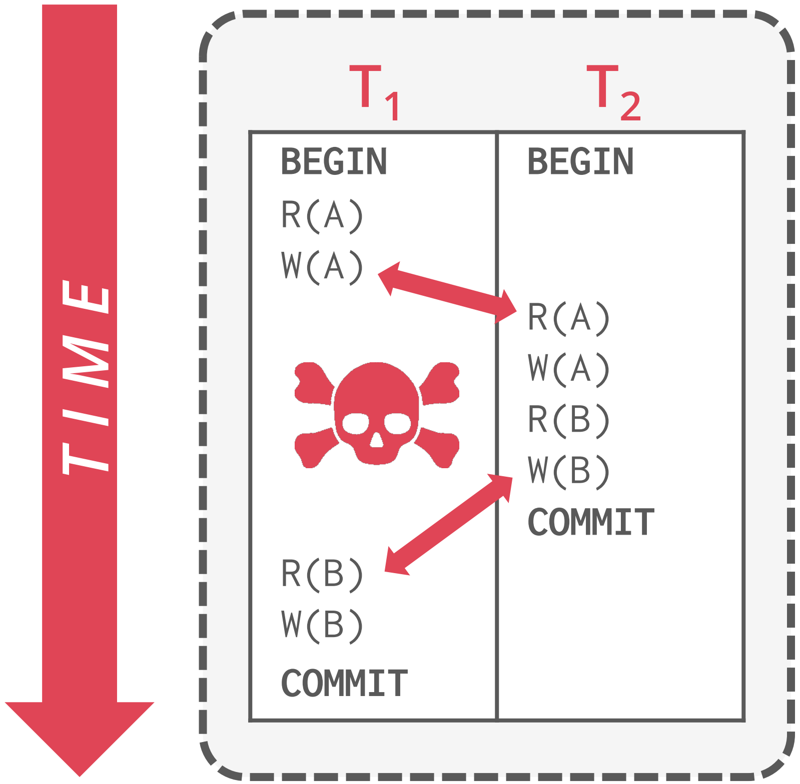
- One node per transaction
- Edge from T_i to T_j if:
  - Operation O_i of T_i conflicts with an operation O_j of T_j and
  - O_i appears earlier in the schedule than O_j
- Also known as a **conflict graph** or **precedence graph**

A schedule is conflict-serializable if and only if its dependency graph is acyclic.

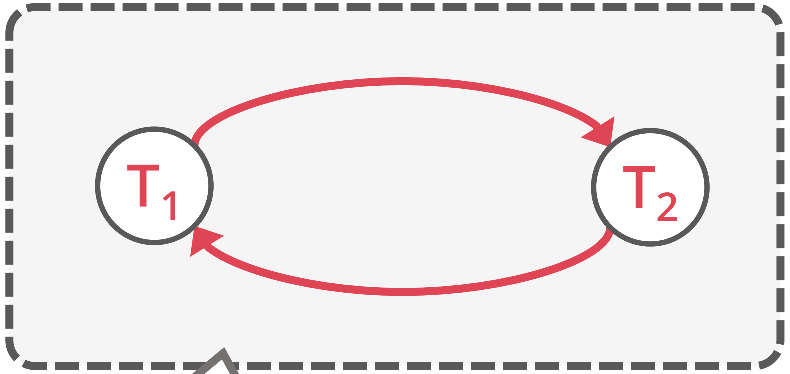
Equivalent **serial schedule** can be obtained by sorting the graph **topologically**

Example #1

Schedule



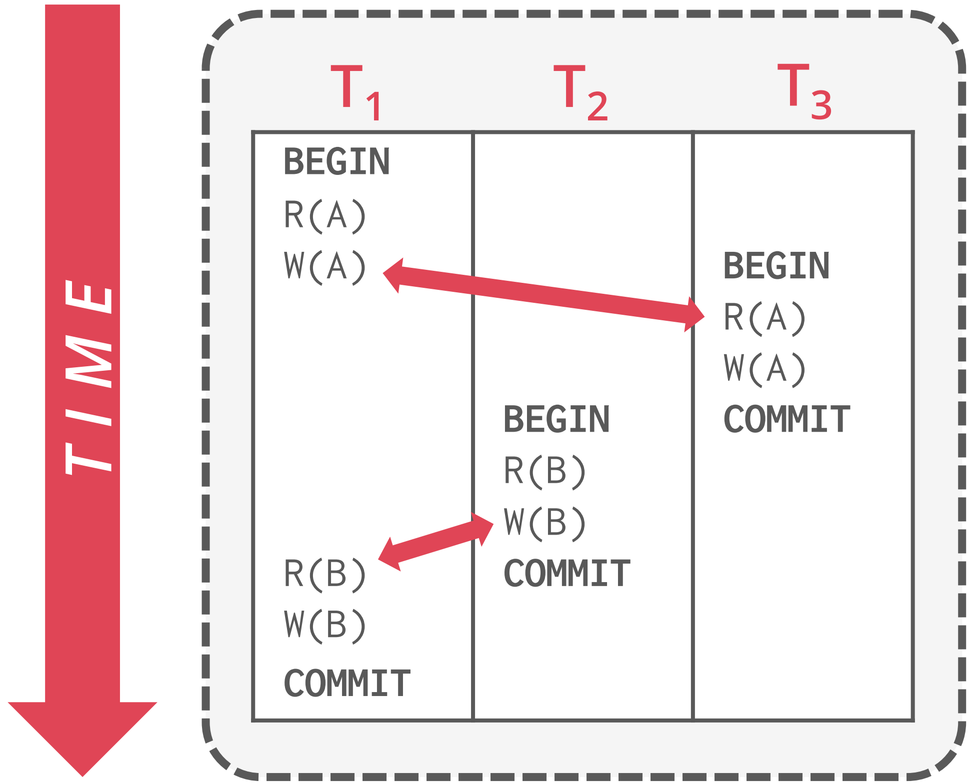
Dependency Graph



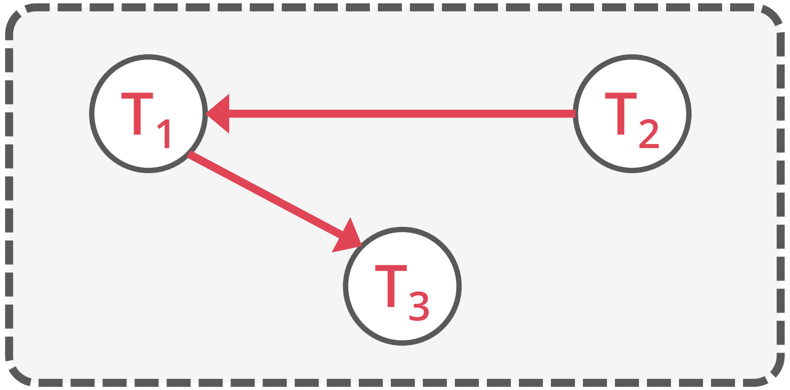
The cycle in the graph reveals the problem. The output of T_1 depends on T_2, and vice versa.

Example #2 - Threesome

Schedule



Dependency Graph



Is this equivalent to a serial schedule?

Yes, (T_2, T_1, T_3)

Notice that T_3 should go after T_2 although T_3 starts before T_2!

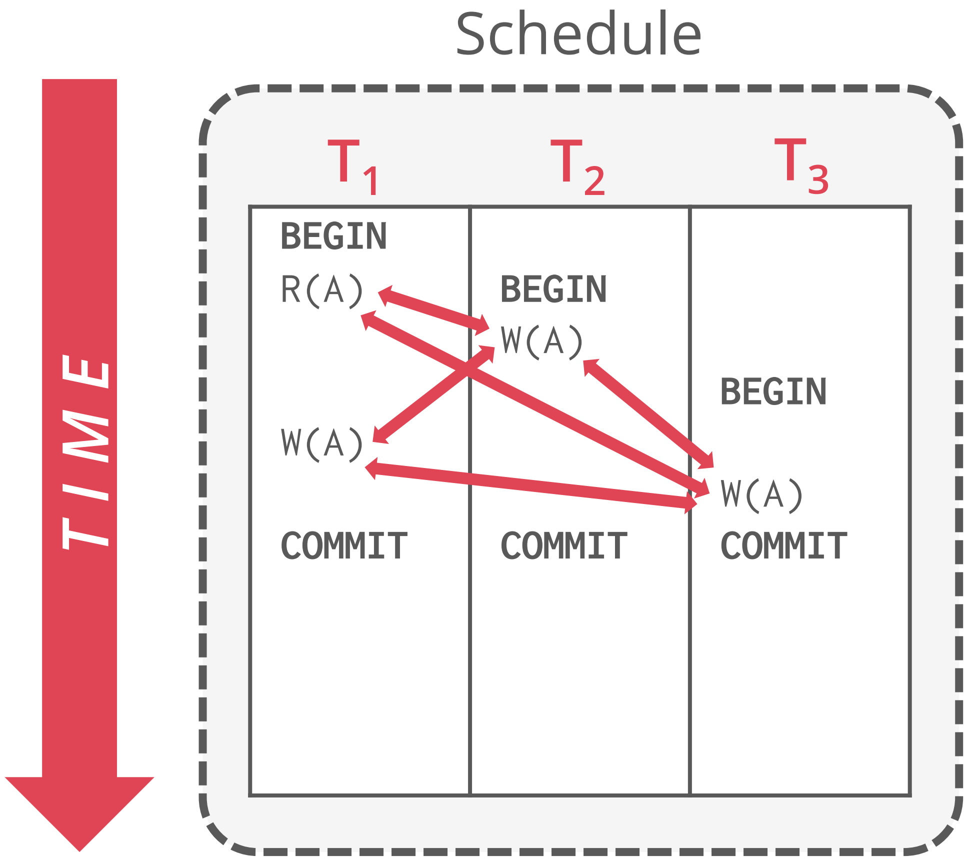
View Serializability

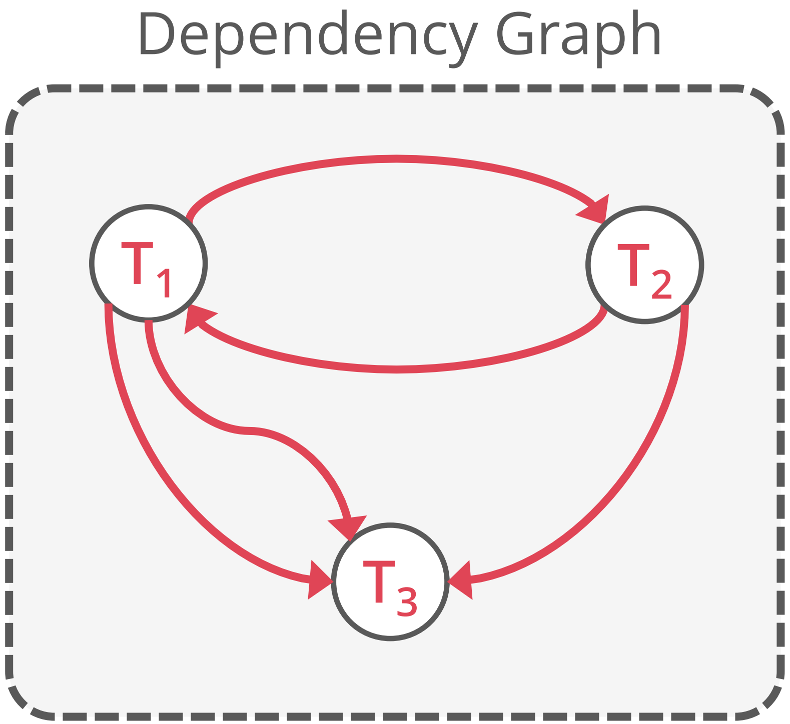
Alternative (weaker) notion of serializability

Schedule S_1 and S_2 are **view equivalent** iff

- 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





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 ≡

Serial 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

Conflict serializability

Can enforced efficiently

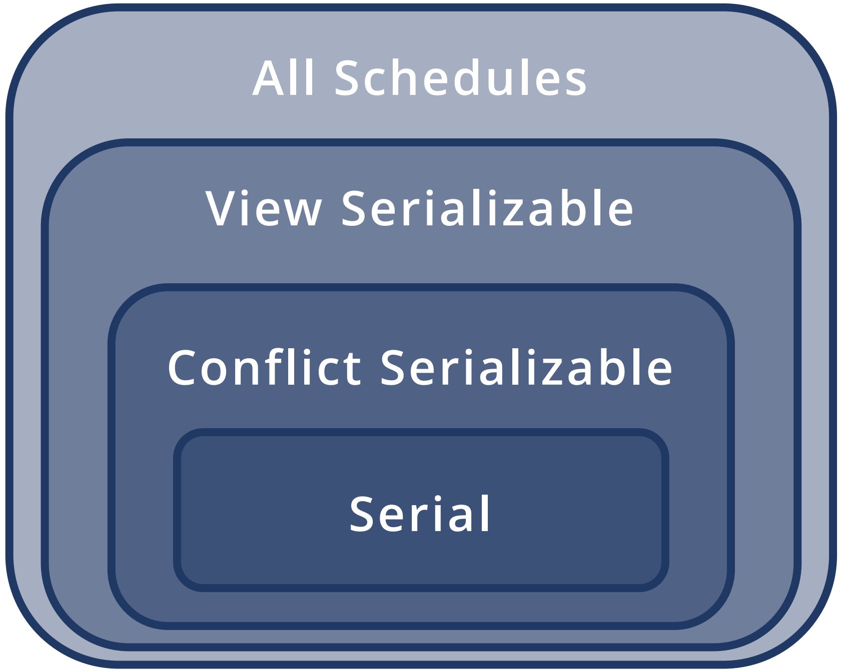
All DBMSs support it

View serializability

Admits (slightly) more schedules than CS

But it is difficult to enforce efficiently

No DBMS supports it



Neither definition allows all "serializable" schedules

They do not understand the meaning of the operations or the data

ACID Properties: Durability

All of the changes of committed transactions must be persistent

- No torn updates
- No changes from failed transactions

The DBMS uses either logging or shadow paging to ensure that all changes are durable

More about logging next week

Conclusion

ACID Transactions

- **Atomicity**: All or nothing
- **Consistency**: Only valid data
- **Isolation**: No interference
- **Durability**: Committed data persists

Serializability

- Serializable schedules
- Conflict & view serializability
- Checking for conflict serializability

Concurrency control and recovery are among the most important functions provided by a DBMS