Slide 1

THE UNIVERSITY of EDINBURGH

Advanced Database Systems
Spring 2026

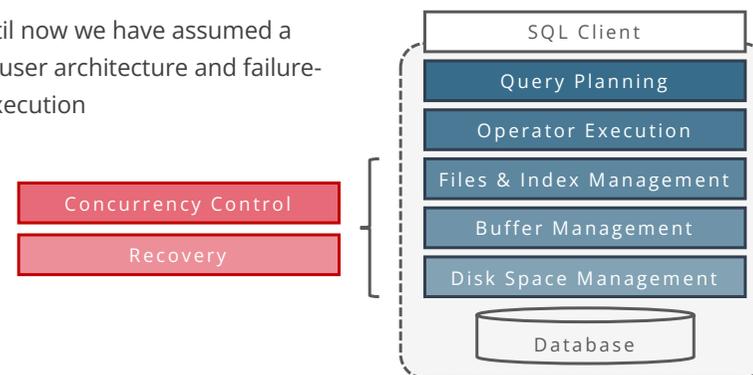Lecture #19:
Transactions

R&G: Chapters 16 & 17

1

Slide 2

ARCHITECTURE OF A DBMS

Up until now we have assumed a single-user architecture and failure-free execution

SQL Client

Query Planning

Operator Execution

Files & Index Management

Buffer Management

Disk Space Management

Database

Concurrency Control

Recovery

2

Slide 3

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
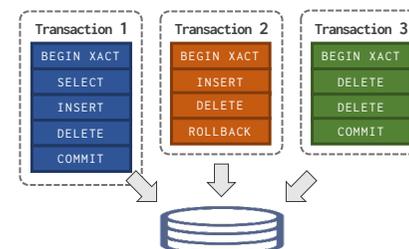
3

Slide 4

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!

Transaction 1
BEGIN XACT
SELECT
INSERT
DELETE
COMMIT

Transaction 2
BEGIN XACT
INSERT
DELETE
ROLLBACK

Transaction 3
BEGIN XACT
DELETE
DELETE
COMMIT

4

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

5

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

6

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 or T-SQL

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

7

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

8

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, page, 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 better approach is to allow **concurrent execution** of independent transactions

Why do we want that?

> Better resource utilization and throughput (txns/sec)
>
> > Use the CPU while another txn is waiting for the disk
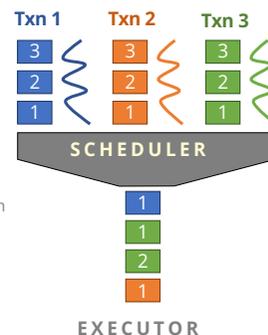> >
> > Multicore: Ideally, scale throughput in the # of CPUs
>
> Decreased response times to users
>
> > One txn's latency need not be dependent on another unrelated txn
> >
> > Or that's the hope

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

Txn 1   Txn 2   Txn 3

SCHEDULER

EXECUTOR

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 outcomes of executing a transaction:

Commit after completing all actions

Abort (or be aborted by the DBMS) after executing some actions

The 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

Efficiency reasons: random writes turned into sequential writes through a log

Audit trail: everything done by the app is recorded

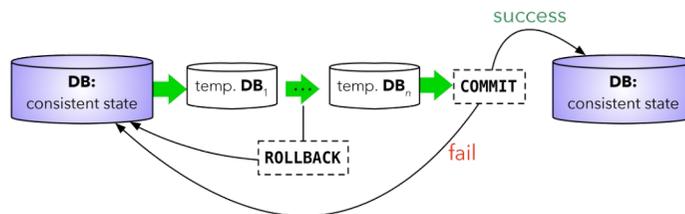**Approach #2: Shadow Paging** (copy-on-write)

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

The 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
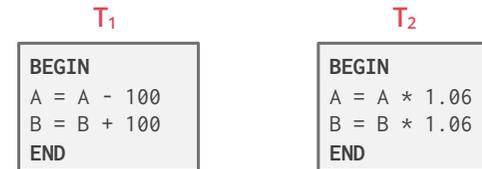
EXAMPLE

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 T_2

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

```
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**

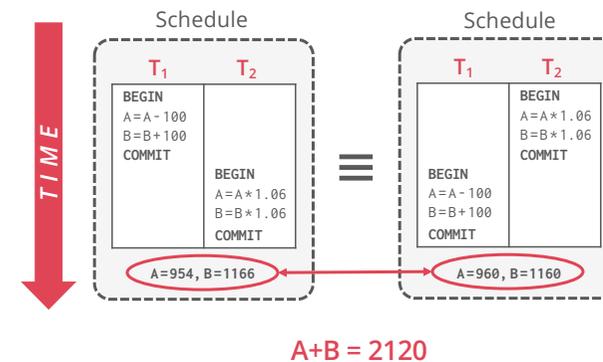There is no guarantee that T_1 will execute before T_2 or vice versa, if both are submitted together

But the net effect must be equivalent to these two transactions running **serially** in some order

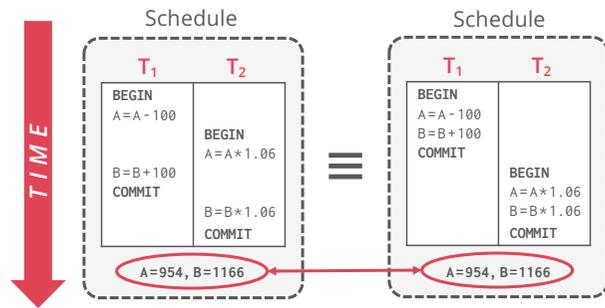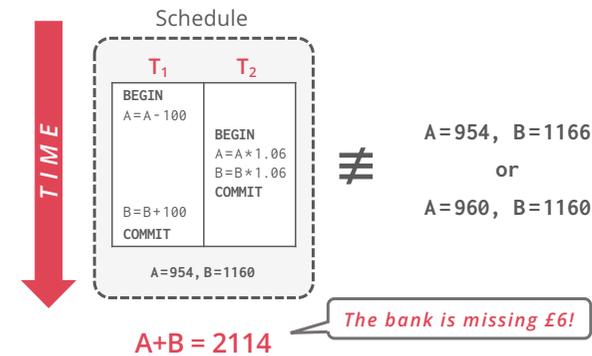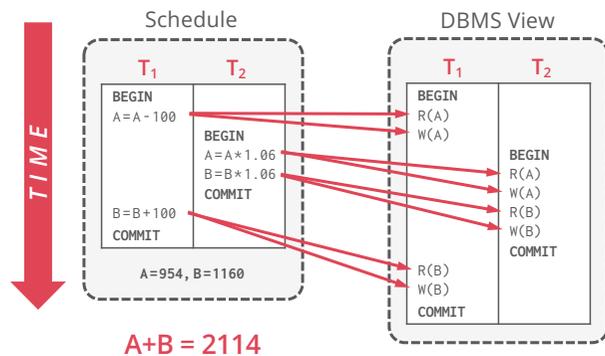## EXAMPLE: SERIAL EXECUTION

A+B = 2120

EXAMPLE: INTERLEAVING (GOOD)

Schedule

T₁   T₂
```
BEGIN
A=A-100
          BEGIN
          A=A*1.06


B=B+100
COMMIT
          B=B*1.06
          COMMIT
```
A=954, B=1166

$\equiv$

Schedule

T₁   T₂
```
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)

Schedule

T₁   T₂
```
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)

Schedule

T₁   T₂
```
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

T₁   T₂
```
BEGIN
R(A)
W(A)
          BEGIN
          R(A)
          W(A)
          R(B)
          W(B)
          COMMIT
R(B)
W(B)
COMMIT
```

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**) ⟩
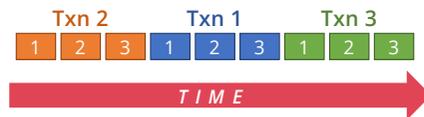
FORMAL PROPERTIES OF SCHEDULES

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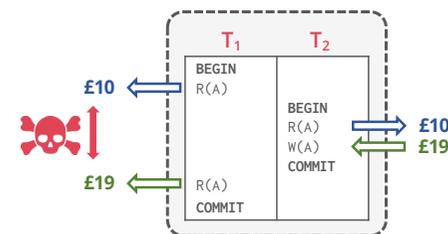
Read-Write conflicts (**R-W**)

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

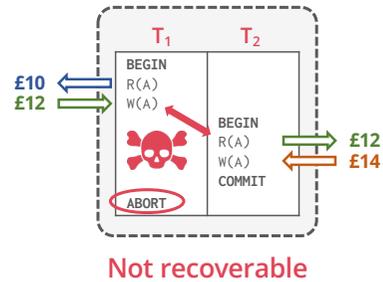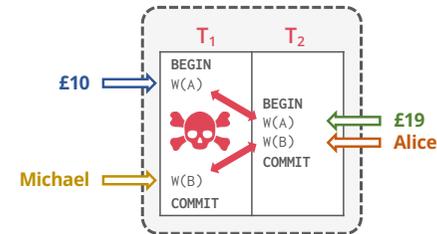Write-Write conflicts (**W-W**)

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 ← Most DBMS try to support this

View Serializability ← No DBMS supports this

CONFLICT SERIALIZABLE SCHEDULES

Two schedules are **conflict equivalent** iff
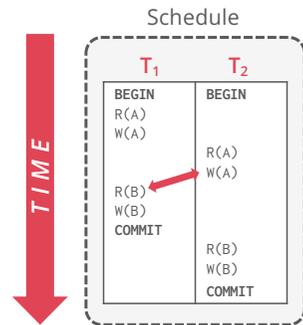
They involve the same actions of the same transactions

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

Slide 36

CONFLICT SERIALIZABILITY: INTUITION

Schedule

TIME

T₁	T₂
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

36

Slide 37

CONFLICT SERIALIZABILITY: INTUITION

Schedule

TIME

T₁	T₂
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

37

Slide 38

CONFLICT SERIALIZABILITY: INTUITION

Schedule

TIME

T₁	T₂
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

38

Slide 39

CONFLICT SERIALIZABILITY: INTUITION

Schedule

TIME

T₁	T₂
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

39

CONFLICT SERIALIZABILITY: INTUITION

Schedule

Serial schedule

TIME

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

≡

T1	T2
BEGIN	
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

Serializable

40

CONFLICT SERIALIZABILITY: INTUITION

Schedule

Serial schedule

TIME

T1	T2
BEGIN	BEGIN
R(A)	
	R(A)
	W(A)
W(A)	
COMMIT	COMMIT

≢

T1	T2
BEGIN	
R(A)	
W(A)	
COMMIT	BEGIN
	R(A)
	W(A)
	COMMIT
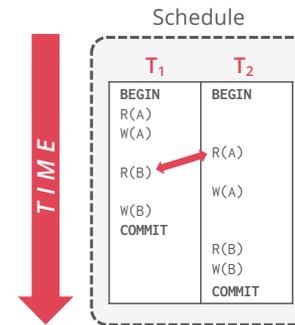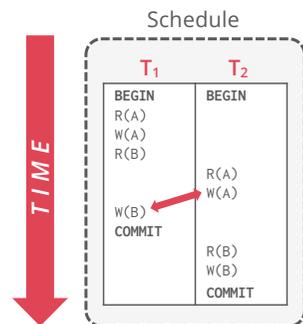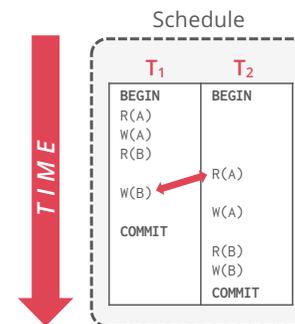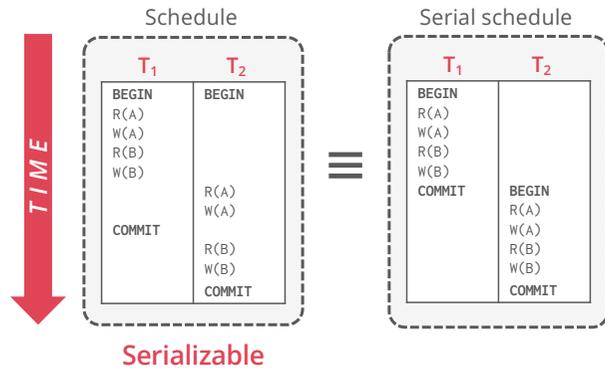
Not conflict-serializable
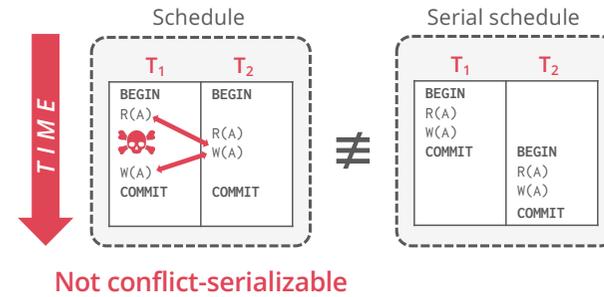
41

SERIALIZABILITY

Swapping operations is easy when there are only two txns in the schedule

But it's cumbersome when there are many txns

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

42

DEPENDENCY GRAPHS

Dependency Graph

Ti ──→ Tj

Dependency graph for a schedule

One node per transaction

Edge from **Ti** to **Tj** if:

Operation **Oi** of **Ti** conflicts with an operation **Oj** of **Tj** and

Oi appears earlier in the schedule than **Oj**

Also known as a **conflict graph** or **precedence graph**

> **A schedule is conflict-serializable if and only if its dependency graph is acyclic**
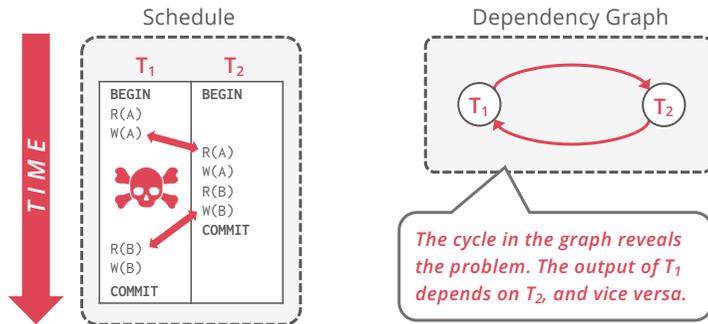
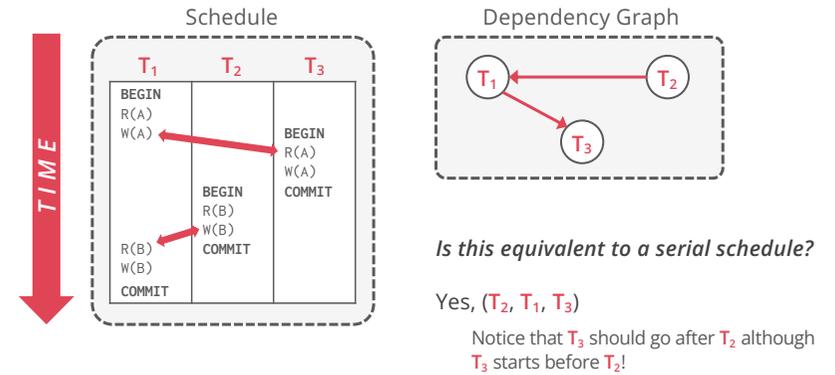Equivalent **serial schedule** can be obtained by sorting the graph **topologically**

43

EXAMPLE #1

Schedule

Dependency Graph

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

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

44

EXAMPLE #2 - THREESOME

Schedule

Dependency Graph

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		

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!

45

VIEW SERIALIZABILITY

Alternative (weaker) notion of serializability

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

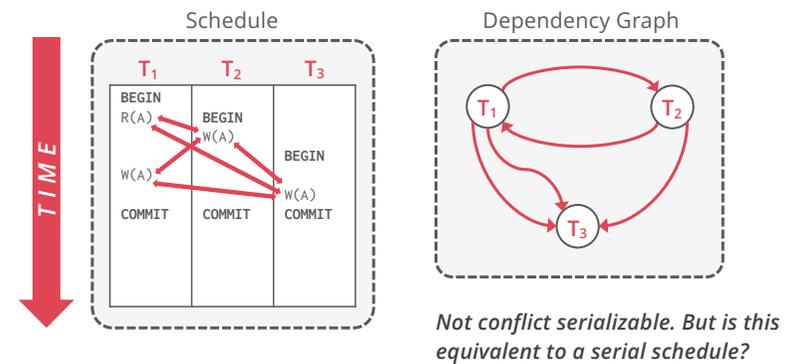If T_1 reads initial value of A in S_1, then T_1 also reads initial value of A in S_2

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

48

VIEW SERIALIZABILITY

Schedule

Dependency Graph

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

Not conflict serializable. But is this equivalent to a serial schedule?

49

VIEW SERIALIZABILITY

Schedule | Serial Schedule

TIME

T₁	T₂	T₃
BEGIN R(A) W(A) COMMIT	BEGIN W(A) COMMIT	BEGIN W(A) COMMIT

VIEW

T₁	T₂	T₃
BEGIN R(A) W(A) COMMIT	BEGIN W(A) COMMIT	BEGIN W(A) COMMIT

Allows all conflict serializable schedules + "blind writes"
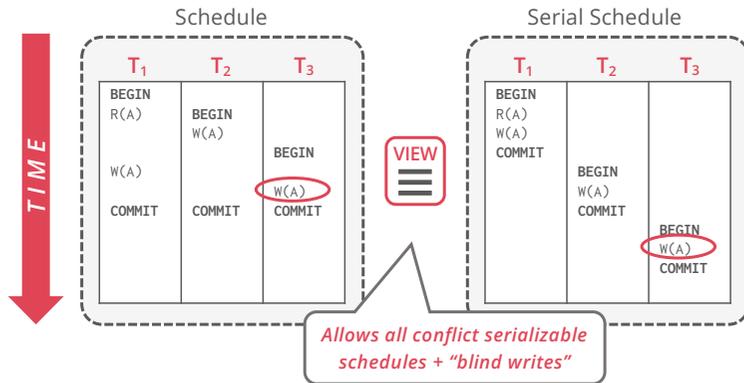
50

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

All Schedules
View Serializable
Conflict Serializable
Serial

51

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 in next lectures

52

SUMMARY

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

53