Concurrency Control Theory

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Motivation
2. A (Brief) Word About Atomicity
3. Concurrency Control: Schedules With Permissible Interleavings
4. Concurrency Control: Dependency Graphs
5. Concurrency Control: Dependency Graphs Are Insufficient!
6. View Serializability

Some material from lectures created by Professor Pavlo (CMU) (used with permission)

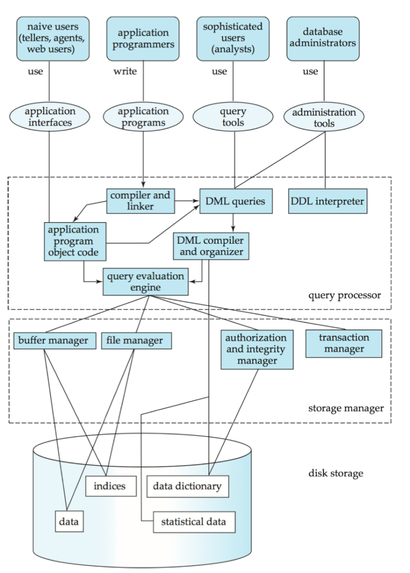
Context

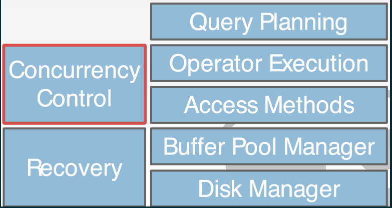
- We're in the middle of becoming skilled RDB users
- Definitely haven't yet started the "database implementation" part of the course ...
- Q: so, why are we doing a "concurrency control theory" lecture now?
- A: because the course project is about transactions
  - You will do a better job on the project with some knowledge of concurrency control theory (this lecture)
  - And knowledge of locking protocols (next lecture)
- If we wait until the "proper time", will be too late for the project

Motivation

Concurrency Control & Recovery

The Figure on the left is slightly misleading ...





- The Figure on the left doesn't convey the degree to which concurrency control and recovery components permeate the design of a DBMS's <u>entire architecture</u>

Motivation (At A Very High Level)

- Concurrency Control is about preventing the “lost updates” problem
  - We both change the same database record at the same time
  - How to avoid race conditions?
- Recovery is about prevent the “no durability” problem
  - You transfer $100 between bank accounts but there is a power failure …
  - What is the correct database state?
- A DBMS provides users the “concurrency control” and “recovery” properties as part of its (speaking loosely) “model of computation”
- These properties are exposed through the concept of ACID transactions

Transactions

- (Refer back to earlier lecture on ACID properties: speaking at a higher-level here ...)
- A <u>transaction</u> is the
  - Execution of a sequence of one or more operations (such as SQL DML statements)
  - On a shared database
  - To achieve something more complex than a "single statement"
  - (Note: even a "single statement" typically requires a transaction to be "atomic" ☺)
- Transactions are the fundamental unit-of-work in a DBMS
  - "Partial" transactions are simply forbidden!

Classic Example

- ("Classic", because it clearly demonstrates why an application cannot tolerate a "partially executed" transaction)
- Unit-of-work: *"Move $100 from Bob's bank account to Sue's bank account"*
    1. Check whether Bob's bank account has $100
    2. Deduct $100 from Bob's account
    3. Add $100 to Sue's account

Will This Implementation Work?

- Execute each transaction one at a time in some serial order as they arrive at the DBMS
- That is: your implementation throttles transaction execution such that "Only one tx can be running at the same time in the DBMS"
- Then:
  1. Before executing a tx: copy the entire database to a new file
  2. Execute the tx by applying its changes to the new file
  3. If the tx completes successfully, overwrite the original file with the new one
     - Overlooking for now the question of how to perform the overwrite in "atomic" fashion ☺
  4. If the tx fails, just remove the "dirty file", revert to the original database file

Yes, This Implementation Will Work, But …

- Consider the resulting terrible performance ☹
- The implementation forbids txs that access a different set of accounts from executing concurrently
- Result:
  - Terrible latency as clients needlessly wait for previously queued transactions to complete serial execution
  - Terrible throughput as DBMS resources are barely utilized when only a single transaction executes
- A (potentially) better approach is to allow concurrent execution of independent transactions
- But only if can provide the benefits of the "serial model"
  - Correctness
  - Fairness

Problem Statement

- We'll allow concurrent tx execution ...
- We're OK with temporary database inconsistency
  - Simply unavoidable given non-atomic hardware capabilities
  - But only OK if this inconsistency is only visible to the tx itself!
- We're not OK with permanent database inconsistency
  - Meaning: inconsistency cannot propagate beyond transaction boundaries
  - That is: beyond `TxMgr.commit` or `TxMgr.rollback`
  - In SQL: we use the BEGIN, COMMIT, ROLLBACK keywords
- Correctness criteria are specified by ACID properties
  - We need to make these criteria more formal than we've done so far
  - We need to say something about implementation approaches

A (Brief) Word About Atomicity

Transaction Atomicity

- As you know, there are two possible outcomes of executing a tx
  - Commit after completing all tx statements
  - Abort (or be aborted by the DBMS) after executing some tx statements
- DBMS guarantees that transaction are atomic from the client's viewpoint
  - A tx executes all statements or no statements
- There are essentially two mechanisms for implementing atomicity
  - Logging
  - Shadow paging
- Just a short discussion for now, much more in later "recovery" lectures

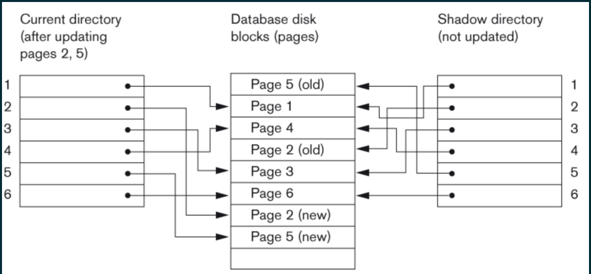
Logging For Atomicity (I)

- Key idea: the DBMS logs all tx actions so that it can undo the actions of aborted transactions
- The "log" is a list of modifications made to the database
- It's duplexed and archived on stable storage
  - This also provides the D in ACID ☺
- The DBMS can force write entries to disk so it "knows" when an action has actually been logged
  - Implication: a tx is committed once a "commit log record" has been written to stable storage
- Q: OK, I "get" the "happy path" implementation ☺
  - But how does this address the scenario of DBMS failure?
- A: See next slide ...

Logging For Atomicity (II)

- The DBMS records undo information in every record!
  - Logs information about what the transaction just did
  - And information about how to "undo" what the transaction just did
- The DBMS writes tx actions sequentially to the log
- Implication: as a DBMS recovers from failure, all it has to do is "execute" all "undo" records
  - Begins from the final log record, works its way backwards until it hit "commit" log records
- Note: almost all modern system use the logging approach (including POSTGRESQL)

Shadow Paging



- When beginning a tx, the DBMS copies the current directory to a shadow directory
- Writes are applied to new pages in the current directory
  - Shadow directory is unchanged, continues to point to old pages
- Tx commit ⇒ "discard the shadow directory"
- Tx rollback ⇒ "reinstate the shadow" directory
- The ground-breaking System R used this approach
  - But rare nowadays
  - Although I did see that it's used in the CouchDB implementation

Concurrency Control: Schedules With Permissible Interleavings

What Is A Concurrency Control Protocol?

- A concurrency control protocol is an algorithm through which the DBMS determines a "correct" interleaving of operations from multiple transactions
- Two broad classes of protocols:
  - Pessimistic: *"don't let problems arise in the first place"*
  - Optimistic: *"assume conflicts are rare, deal with problems if and when they do occur"*
- We'll be using this DBMS abstraction of a user's program:
  - A database is a fixed set of named data objects (e.g., $A, B, \ldots$
  - A transaction is a sequence of read and write operations $(R(A), W(B), \ldots)$
- A DBMS is only concerned about the data that are read/written from/to the database
  - Changes to the "outside world" (such as "send an email") are out of DBMS scope

EXAMPLE

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

T_1 transfers \$100 from **A**'s account to **B**'s

T_2 credits both accounts with 6% interest.

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?

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

TIME

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

≡

Schedule

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

A=960, B=1160

A+B=$2120

INTERLEAVING EXAMPLE (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=960, B=1160

INTERLEAVING EXAMPLE (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=960, B=1160

A+B=$2120

INTERLEAVING EXAMPLE (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=1060

≠

A=954, B=1166

or

A=960, B=1160

A+B=$2014

The bank is missing $106!

INTERLEAVING EXAMPLE (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	

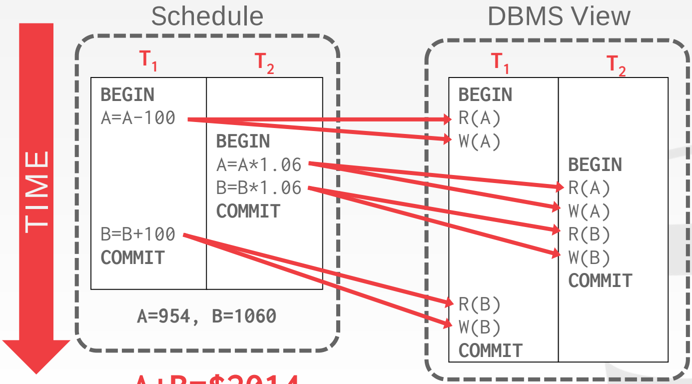
A=954, B=1060

A+B=$2014

DBMS View

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

INTERLEAVING EXAMPLE (BAD)



A+B=$2014

- We now have a correctness criterion that can tell us whether any DBMS execution schedule is correct ☺

> A schedule is correct *if and only if* it is equivalent to some serial execution

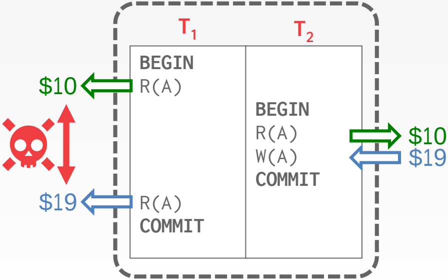
- Here “equivalent” means: for any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule
- Definition: a serializable schedule is *“a schedule that is equivalent to some serial execution of the transactions”*

Implementing A Non-Intuitive Criterion

- The previous examples should have made it clear that "serializability" is not an intuitive concept ☹
- But: our motivation for using it as the correctness criterion is that it provides the DBMS flexibility in scheduling operations
  - Meaning: the serializable schedule criterion gives us the benefits of concurrent execution
  - Without: giving up correctness
- Our problem at this point: how to implement an "equivalence detector" efficiently?
- We'll begin by introducing the concept of conflicting operations
- We say that two operations conflict if:
  1. Two different transactions have initiated the operations
  2. Both operations access the same object
  3. At least one of the operations is a write

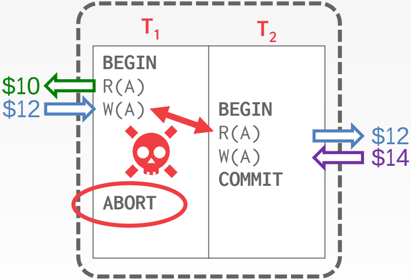
READ-WRITE CONFLICTS

Unrepeatable Reads



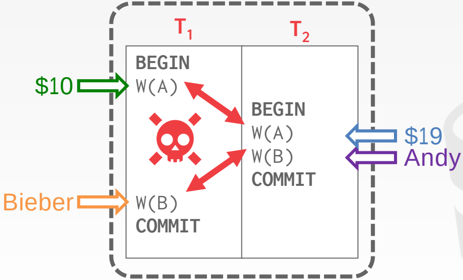
WRITE-READ CONFLICTS

Reading Uncommitted Data ("Dirty Reads")



WRITE-WRITE CONFLICTS

Overwriting Uncommitted Data



An “Equivalent Schedule” Detector

- We can now define an efficient algorithm that can check whether two schedules are equivalent
  - Remember: this will enable us to determine whether a schedule is serializable
- Two schedules are conflict equivalent *iff*
  - They involve the same actions by the same transactions
  - And: every pair of conflicting actions is ordered the same way
- Intuition: schedule S is equivalent to S' *iff* S can be transformed into S' by a series of swaps of non-conflicting instructions
- A schedule S is conflict serializable *iff* S is conflict equivalent to some serial schedule

CONFLICT SERIALIZABILITY INTUITION

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

CONFLICT SERIALIZABILITY INTUITION

TIME

Schedule

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

TIME

Schedule

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

TIME

Schedule

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

CONFLICT SERIALIZABILITY INTUITION

Schedule

TIME

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

CONFLICT SERIALIZABILITY INTUITION

TIME

Schedule

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

CONFLICT SERIALIZABILITY INTUITION

Schedule

TIME

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

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

≡

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

Concurrency Control: Dependency Graphs

“Are We Done Yet?”

- Review: we now have an algorithm that determines whether a proposed interleaved schedule is “correct”
  - It’s correct only if conflict equivalent to <u>some</u> serial schedule
  - Algorithm “swaps <u>non-conflicting</u> operations” to determine conflict equivalence
- Unfortunately: we still have a problem ☹
- Swapping operations is easy when there are only two txns in the schedule
  - In real-life™, a DBMS processes many, many concurrent transactions
- So: we want a (faster) algorithm that isn’t based on “swapping operations”
- The good news: we can use dependency graphs to construct such an algorithm ☺

Dependency Graphs

- Construct a graph as follows:
  - One node per transaction
  - Add an edge from $Tx_i \Rightarrow Tx_j$ *iff*
    - An operation O_i (in Tx_i) conflicts with an operation O_j (in Tx_j)
    - And: O_i appears earlier in the schedule than O_j
- Such graphs are called dependency or precedence graphs

> A schedule is conflict serializable *iff* its dependency graph is acyclic

Algorithm

- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph
  - A linear order consistent with the partial order of the graph
- Was shocked to see (previous edition) of Textbook state $O(V^2)$ algorithm
  - Ridiculous: from COM 2545 we know $O(V + E)$ algorithms exist ☺

Example: Is This Schedule Serializable?

T_1	T_2	T_3	T_4
read(A)			
	write(A)		
		read(A)	
		write(B)	
		read(C)	
			write(C)
read(B)			

Q: Is this schedule serializable?

$$\begin{tikzcd} T_1 \arrow[r] & T_2 \arrow[r] & T_3 \arrow[r] \arrow[ll, bend left] & T_4 \end{tikzcd}$$

A: A cycle, so not conflict serializable

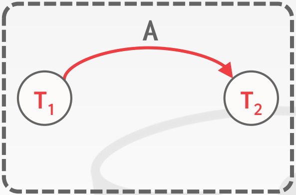
EXAMPLE #1

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	

Dependency Graph



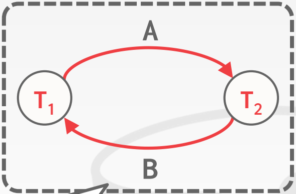
EXAMPLE #1

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	

Dependency Graph



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

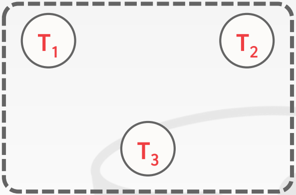
EXAMPLE #2 – THREESOME

Schedule

TIME

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

Dependency Graph



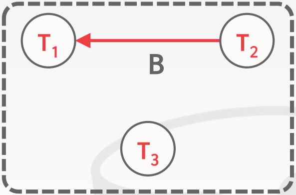
EXAMPLE #2 – THREESOME

TIME

Schedule

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		

Dependency Graph



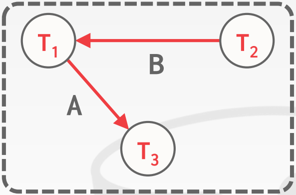
EXAMPLE #2 – THREESOME

Schedule

TIME

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

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!

Atomicity In the Presence of Failures

- Discussion of serializable schedules has (implicitly) assumed that transactions will never fail
  - We can crank up concurrency to the max, as long as the schedules remain serializable
- But transactions can fail (if only because internal business logic threw an exception)
- If T_j has already read data written by T_i and T_i does not commit
  - DBMS must also rollback T_j
- Such scenarios place further constraints on the set of valid transaction schedules

Recoverable Schedules

- Recoverable schedule: if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i must appear before the commit operation of T_j
- Q: Is the following schedule recoverable?

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- A: No ...
  - If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state
  - DBMS must not allow this schedule!

Cascading Rollbacks

- Does not suffice to have a *recoverable schedule*
- Even if a transaction commit occurs after a transaction that it depends on, DBMS still has to worry about a cascading rollback scenario
- Example:

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

- None of the transactions has yet committed so the schedule is recoverable …
- But if T_{10} fails, T_{11} and T_{12} must also be rolled back
- Can lead to the undoing of a significant amount of work ☹

Avoiding Cascading Rollbacks

DBMS can avoid cascading rollbacks by not allowing yet another set of schedules

> A cascadeless schedule is one where: for every T_i and T_j in which T_j depends on T_i (T_j reads a value that was written by T_i), the commit of T_i occurs before the read operation of T_j

Note: every cascadeless schedule is also recoverable

Summary (So Far): Concurrency Control Protocol Needs More

- Concurrency-control protocols allow concurrent schedules that are conflict serializable
  - Alternatively: view-serializable (see end of lecture)
- In addition: concurrency protocols must ensure that the schedules are recoverable and cascadeless
- Most important: concurrency control protocols (generally) do not examine the precedence graph as it is being created
- Instead: a protocol uses an algorithm that avoids non-serializable schedules
  - Next lecture ☺
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur
- We only use "serializability tests" to help us understand why a concurrency control protocol is correct

View Serializability

- View serializability allows for (slightly) more schedules than conflict serializability does
- View serializability is not used in practice, mostly because testing or rejecting this condition is NP-complete difficult ☹
- That's why I've deferred this discussion until now

> Note: you are responsible for this material but only at a high level

- Both conflict and view serializability disallow some schedules that are plausibly termed "serializable"
  - Because the only operation semantics they understand are "read" and "write"

- 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
- The effect of these rules is that *“As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results”*
- We say that *“read operations see the same view in both schedules*

Comparing The Two "Equivalence Definitions"

- The only difference (in practice) between these two definitions is that "view serializability" permits schedules with "blind writes"
  - ...and "conflict serializability" does not
- "Blind writes" are sometimes called "unconstrained writes"
  - Meaning: view serializability will allow a value written by an operation to be independent of its old value

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

> The precedence graph test for conflict serializability cannot be used directly to test for view serializability

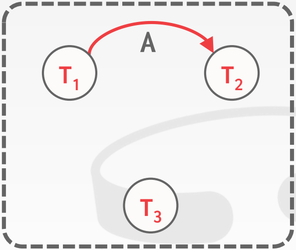
VIEW SERIALIZABILITY

Schedule

TIME

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

Dependency Graph



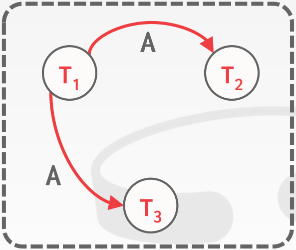
VIEW SERIALIZABILITY

Schedule

TIME

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

Dependency Graph



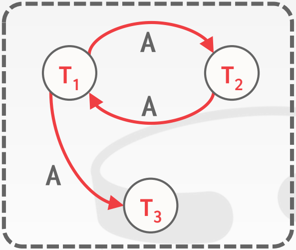
VIEW SERIALIZABILITY

TIME

Schedule

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

Dependency Graph



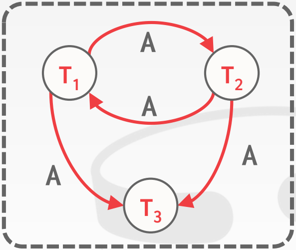
VIEW SERIALIZABILITY

TIME

Schedule

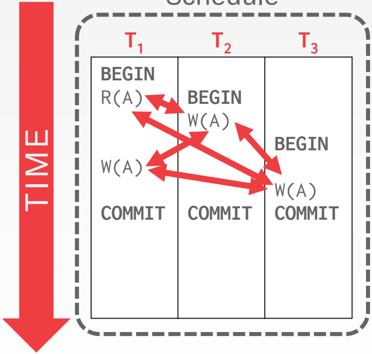
T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph

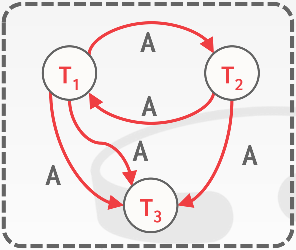


VIEW SERIALIZABILITY

Schedule



Dependency Graph



VIEW SERIALIZABILITY

TIME

Schedule

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

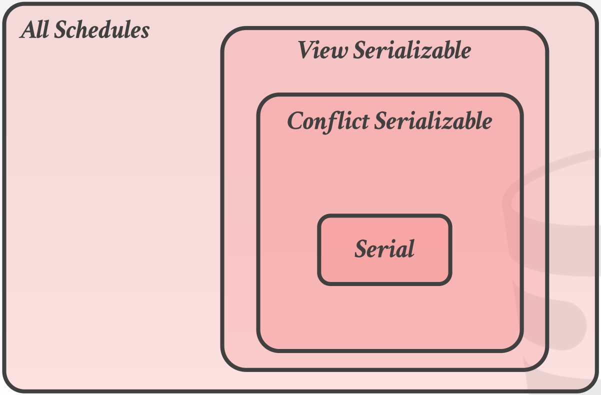
VIEW ≡

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"

UNIVERSE OF SCHEDULES



Motivation

A (Brief) Word About Atomicity

Concurrency Control: Schedules With Permissible Interleavings

Concurrency Control: Dependency Graphs

Concurrency Control: Dependency Graphs Are Insufficient!

View Serializability

Readings

- The textbook gives an overview of transactions, serializability, and isolation levels in Chapter 17
  - Rough overlap with today's lecture **except** that we did not yet discuss "isolation levels" (Chapter 17.8-9)
- The textbook drills down into concurrency control in Chapter 18
  - Rough overlap with next lecture