Timestamp-Based Concurrency Control & Isolation Levels

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Timestamp Ordering: The Basic Idea

2. Optimistic Concurrency Control: Validation-Based Protocols

3. Isolation Levels

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

Context

- We're coming to the end of our series of concurrency control lectures
- Although today's lecture is the final one on this topic for this semester, we have definitely <u>not</u> covered even the broad contours of this important topic
- Example: could easily devote a lecture to each of
  - Validation-based protocols (Chapter 18.6)
  - Multiversion protocols (Chapter 18.7)
- So: continue to walk humbly, study this material if you're interested in the topic, and all will be well ...

Timestamp Ordering: The Basic Idea

Concurrency Control Based on Timestamp Ordering

- Until now, we've only considered lock-based protocols as a concurrency control implementation technique
- Specifically: we examined the 2PL protocol
  - 2PL determines the serializability order of conflicting operations at runtime
  - Meaning: while txs execute
- Today: we'll examine timestamp ordering (or TSO) as a concurrency control implementation technique
- TSO schemes determine serializability order of txs before they execute
- As we shall see, we can characterize 2PL as a pessimistic approach, in contrast to the optimistic approach used by TSO
- TSO techniques do not use locks!
  - Implication: we don't have to worry about deadlocks ☺

Timestamps & TSO

- In the concurrency-control context, a timestamp is simply a *unique identifier assigned by the DBMS to transactions*
- Each transaction tx_i is assigned a timestamp $TS(tx_i)$ when it enters the system
- Newer transactions have timestamps that are strictly greater than earlier ones
- Timestamp values can be based on a logical counter
  - Note: "real time" values may not be unique
  - Can use a combination of "wall-clock" time concatenated to a "logical counter"
- Key idea: TSO protocols manage concurrent execution such that timestamp order equals serializability order
- Meaning: *"if $TS(tx_i) < TS(tx_j)$, then the DBMS must ensure that the execution schedule is equivalent to a serial schedule where tx_i appears before tx_j"*
- This basic idea underlies several alternative timestamp-based protocols

“Basic” TSO Protocol

- Transactions read and write objects without locks
- Every datum X is tagged with timestamp of the last tx that successfully did a read or write
  - Define W-TS (X): the write timestamp on X
  - Define R-TS (X): the read timestamp on X
- So: transactions are assigned timestamps and data are also assigned read and write timestamps
- Now we imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in transaction timestamp order
  - Out of order operations cause transaction rollback

“Basic” TSO Protocol: tx_i Reads(X)

- If $\text{TS}(tx_i) < \text{W-TS}(X)$, we have a violation of the timestamp order of tx_i with regard to the writer of X
  - Meaning: the transaction needs to read data that was already overwritten
  - Therefore: abort tx_i and restart it with new timestamp value
- Else ($\text{TS}(tx_i) \geq \text{W-TS}(X)$):
  1. Allow tx_i to read X
  2. Update R-TS (X) to $max(\text{R-TS}(X), \text{TS}(tx_i))$
- Note: the DBMS must make a private copy of X to ensure repeatable reads for tx_i

"Basic" TSO Protocol: tx_i Writes(X)

- If $\text{TS}(tx_i) < \text{R-TS}(X)$, the value of X that tx_i is creating was previously assumed by the DBMS to be "something that will not be produced"
  - After all: the DBMS let some other transaction read it
  - So: abort tx_i, and restart with new timestamp value
- If $\text{TS}(tx_i) < \text{W-TS}(X)$, then tx_i is attempting to write an obsolete value of X
  - So: abort tx_i, and restart with new timestamp value
- Else:
  - Allow tx_i to write X
  - Set W-TS (X) to the value of $\text{TS}(tx_i)$
  - As before, the DBMS must make a private copy to X to ensure repeatable reads for tx_i

BASIC T/O – EXAMPLE #1

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

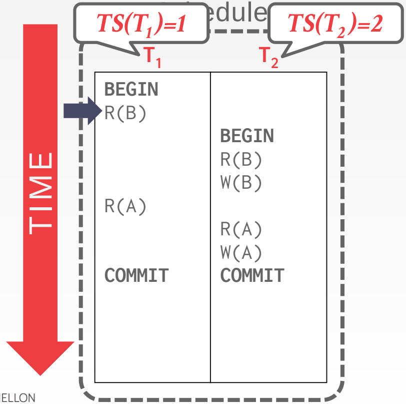
TIME

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

Database

Object	R-TS	W-TS
A	0	0
B	0	0

BASIC T/O – EXAMPLE #1



Database

Object	R-TS	W-TS
A	0	0
B	1	0

BASIC T/O – EXAMPLE #1

edule

$TS(T_1)=1$ $TS(T_2)=2$

TIME

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

Database

Object	R-TS	W-TS
A	0	0
B	2	0

BASIC T/O – EXAMPLE #1

TIME

$TS(T_1)=1$ $TS(T_2)=2$

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

Database

Object	R-TS	W-TS
A	0	0
B	2	2

BASIC T/O – EXAMPLE #1

TIME

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

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

Database

Object	R-TS	W-TS
A	1	0
B	2	2

BASIC T/O – EXAMPLE #1

TIME

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

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

Database

Object	R-TS	W-TS
A	2	0
B	2	2

BASIC T/O – EXAMPLE #1

$TS(T_1)=1$ **$TS(T_2)=2$**

Schedule

TIME

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

Database

Object	R-TS	W-TS
A	2	2
B	2	2

No violations so both txns are safe to commit.

BASIC T/O – EXAMPLE #2

Schedule

TIME

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

Database

Object	R-TS	W-TS
A	1	0
B	0	0

BASIC T/O – EXAMPLE #2

Schedule

TIME

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

Database

Object	R-TS	W-TS
A	1	2
B	0	0

BASIC T/O – EXAMPLE #2

TIME

Schedule

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

T_1 cannot overwrite update by T_2, so the DBMS has to abort it!

Database

Object	R-TS	W-TS
A	1	2
B	0	0

Violation:
$TS(T_1) < W\text{-}TS(A)$

Correctness of Timestamp-Ordering Protocol

In a nutshell: the TSO protocol ensures that if a tx tries to access an object “from the future”, it aborts and restarts

- The TSO protocol guarantees serializability!
- Conflicting operations are ordered in timestamp order
- All edges in the dependency graph represent a dependency from a transaction with a larger timestamp on a transaction with a smaller timestamp
  - Implication: no cycles in the dependency graph ☺
- Note: TSO protocol ensures “no deadlocks”
  - No transaction ever waits ☺
- The protocol detects potential conflicts and avoids them
- The protocol produces a conflict equivalent schedule that orders transactions by their timestamps

Problems With The Timestamp-Ordering Protocol (I)

- The TSO is vulnerable to starvation
  - Doesn't have built-in protection that ensures that some (e.g., "long") transaction will not get repeatedly restarted ☹
- Bigger problem: TSO produces schedules that are not recoverable
- Reminder: a schedule is recoverable if txs commit only after all txs whose changes they read, commit
  - TSO is only checking the "relative timestamp order": it's not looking at "commit" and "rollback" events!
- "Recoverable schedules" are a serious issue!
  - After all, if a schedule is not recoverable, what can the DBMS do to rollback already committed txs when the tx on which they depend subsequently fail?

Problems With The Timestamp-Ordering Protocol (II)

- TSO may even generate schedules that are not cascade-free
  - Meaning: a single tx failure cause multiple tx rollbacks because the other txs depend (whether directly or indirectly on that first transaction
- Reminder: a cascadeless schedule is one where for each pair of transactions tx_i and tx_j in which tx_j reads data previously written by tx_i . . .
  - tx_i commits before tx_j reads the data written by tx_i
- Note: every cascadeless schedule is also a recoverable schedule
- See Textbook for “fixes” to the TSO algorithms: frankly, they seem like a “hack” to me ☹

RECOVERABLE SCHEDULES

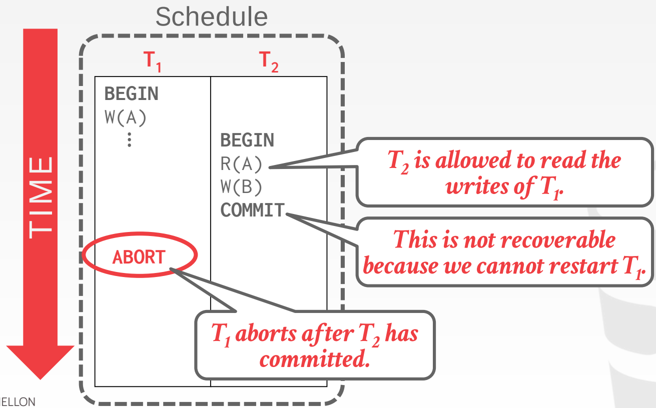
Schedule

TIME

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

T_2 is allowed to read the writes of T_1.

RECOVERABLE SCHEDULES



Thomas' Write Rule (I)

- This TSO modification is motivated by a wish for more concurrency than permitted by vanilla TSO
- Key idea: allow obsolete write operations under certain circumstances
- Recall: when tx_i attempts to write X, if $\text{TS}(tx_i) < \text{W-TS}(X)$, then tx_i is attempting to write an obsolete value of X
- Thomas' Write rule says: instead of rolling back tx_i, ignore this write operation – if no other tx is going to read its value, and allow tx_i to continue
- Implication: will allow some view-serializable schedules that are not conflict serializable

Thomas' Write Rule (II)

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

- Under vanilla TSO, because TS $(tx_{27}) <$ TS (tx_{28}), read(Q) operation of tx_{27} succeeds, as does the write(Q) operation of tx_{28}
- But: when tx_{27} attempts its write(Q) operation, the DBMS rejects it, because TS $(tx_{27}) <$ W-TS(Q)
- Observation: we don't really have to rollback tx_{27}!
- tx_{28} has already written Q, and the value that tx_{27} is attempting to write will never be read (by a concurrent transaction)
- Note: any transaction tx_i with TS$(tx_i) <$ TS(tx_{28}) that attempts a read(Q) will be rolled back anyway,
  - Because TS$(tx_i) <$ W-TS(Q)
- And: any transaction tx_i with TS$(tx_i) >$ TS(tx_{28}) must read the value of Q written by tx_{28}, rather than the value that tx_{27} is attempting to write
- Thomas: safe to ignore write(Q) operation of tx_{27}
- The result is a schedule that is view equivalent to the serial schedule $< tx_{27}, tx_{28} >$

Optimistic Concurrency Control: Validation-Based Protocols

Optimistic Concurrency Control: Key Insight

- If conflicts between txs are rare …
- And most tx durations are short-lived …
- Then: forcing txs to wait while they acquire locks, adds considerable (<u>unnecessary</u>) overhead
- Better approach: optimize for "no-conflict" scenarios

Optimistic Concurrency Control: Approach

- DBMS creates a "private workspace" (aka our "shadow db") for each transaction
  - Tx reads data? DBMS first copies data to shadow db
  - Tx writes data? Modifications are applied to the copy
- At commit time, DBMS compares shadow db state to see whether this state conflicts with state of other txs
  - Meaning: the state of their "shadow dbs"
- If there are no conflicts, the "write set" is copied back to the "real db"
  - Otherwise: abort the transaction
- Summary: three phases
  1. Read phase
  2. Validation phase
  3. Write phase
- Major benefits: tx execution can be interleaved without locking
- Assume that the validation and write phases are performed atomically and serially

OCC – EXAMPLE

Schedule

TIME

T1	T2
BEGIN	BEGIN
READ	
R(A)	*READ*
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

T1	T2
BEGIN	BEGIN
READ	
R(A)	*READ*
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ R(A)	
	READ R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
-	-	-

T_1 Workspace

Object	Value	W-TS
-	-	-
-	-	-

OCC – EXAMPLE

TIME

Schedule

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

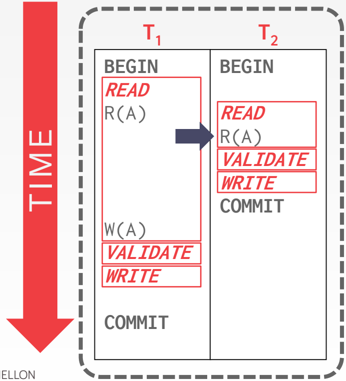
Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
–	–	–
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

$TS(T_2)=1$

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

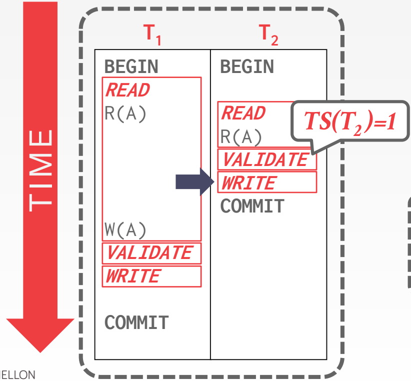
Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule



Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

T_2 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	*READ*
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

$TS(T_2)=1$

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	123	0
–	–	–

OCC – EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

TS(T_2)=1

Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
	READ
	R(A)
	VALIDATE
	WRITE
W(A)	
VALIDATE	
WRITE	
COMMIT	

$TS(T_2)=1$

$TS(T_1)=2$

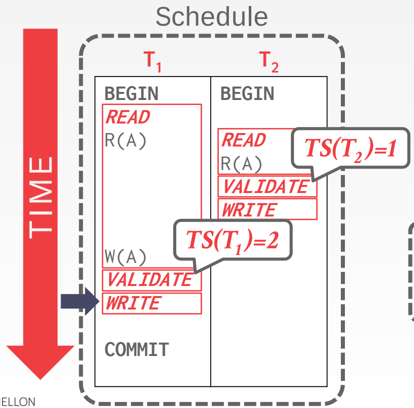
Database

Object	Value	W-TS
A	123	0
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

OCC – EXAMPLE



Database

Object	Value	W-TS
A	456	2
–	–	–

T_1 Workspace

Object	Value	W-TS
A	456	∞
–	–	–

Optimistic Concurrency Control: Towards An Implementation

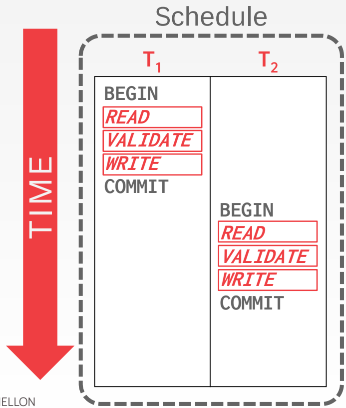
Key idea: use commit time as serialization order

- Associate each transaction tx_i with three timestamps
  - startTS: the time that a tx started its execution
  - validationTS: the time that a tx exited its read phase and entered its validation phase
  - finishTS: the time that a tx finished its write phase
- Using TSO notation, we set $\text{TS}(tx_i)$ to its validationTS
- If $\text{TS}(tx_i) < \text{TS}(tx_j)$, all valid schedules produced by this protocol must be equivalent to a serial schedule in which tx_i appears before tx_j
- Now let's drill down into the validation test ...

Validation Test For tx_j #1

- Consider any two transactions such that $\textsc{ts}(tx_i) < \textsc{ts}(tx_j)$
- The DBMS determines that $\underline{finish}TS(tx_i) < \underline{start}TS(tx_j)$
- This condition applies when the two txs are <u>not</u> executing concurrently
- Then the writes of tx_j do <u>not affect</u> the reads of tx_i because they occur <u>after</u> tx_i has finished its reads
- Conclusion: OK to commit tx_j

OCC – VALIDATION STEP #1



Validation Test For tx_j #2

- Consider any two transactions such that $\text{TS}(tx_i) < \text{TS}(tx_j)$
- The DBMS determines that $\underline{start}TS(tx_j) < \underline{finish}TS(tx_i) < \underline{validation}TS(tx_j) \ldots$
  - Key point: tx_i completes its write phase before tx_j begins its validation phase …
- And: the set of data items written by tx_i has no overlap with the set of data items read by tx_j
- This condition applies when the two txs are executing concurrently but the writes of tx_i do not affect the reads of tx_j
  - And: tx_j cannot affect the reads of tx_i …
- Conclusion: OK to commit tx_j

If both of these validation tests fail, the DBMS must abort tx_j

Reviewing Optimistic Concurrency Control

- Produces cascadeless schedules ☺
  - All "actual" writes occur after a tx has successfully committed
- By reviewing the validation test, you can see why optimistic concurrency control performs well when there are relatively feew "inter-tx" conflicts
  - Either txs are read-only (or "read-mostly")
  - Or: txs access disjoint subsets of data
- Performance issues
  - Pay overhead for copying data to/from "shadow db"
  - The validation & write phases are bottlenecks because the DBMS has to serialize (in effect "lock") the database during these activities
    - Implication: may not perform well when lots of concurrent activity ☹
  - Also: when the DBMS does abort a tx, the results are more costly than in 2PL
    - Because they occur after a txn has already executed to ("almost") completion ☹

Isolation Levels

Introduction (I)

- We've devoted several lectures to explain the concept (and implementation strategies for) of serializability
- We've established that serializability is extremely useful because it allows programmers to ignore concurrency issues
- Yet, as we're about to discuss, many database either provide, or allow programmers to specify weaker isolation levels than provided by serializability!
  - Meaning: tx_1 can "see" the effects of tx_2 execution during its own execution

Introduction (II)

- Q: why would databases (and programmers) want to use anything other than “serializable isolation”?
- A: because there is a fundamental tradeoff between “strict isolation” on the one hand and “good performance” on the other ☹
- You can already see why this is so: serializability is about restricting the set of possible schedules to those that meet the proper criteria
    - By relaxing (“weakening”) the isolation criteria, the DBMS can increase its level of concurrency

(Some) Applications Can Live With Weaker Isolation

Examples:

- A read-only transaction that wants to get an approximate total balance of all accounts
  - Doesn't matter that other transactions are modifying existing accounts, inserting new accounts, or removing accounts
  - You expect to get the "essentially the same answer" in all cases
- When the DBMS computes internal statistics (for query optimization purposes), its analysis can similarly be approximate

Relaxing Isolation Levels: More Pervasive Examples

> Human beings are so flexible that they can resolve inconsistency on the fly ☺
>
> Alternatively: we make so many mistakes that enterprises provide work-flow mechanisms that enable humans to override or accommodate database inconsistency

- Example: Amazon doesn't lock its stock of widgets simply because you added a widget to your cart
  - Q: how does Amazon deal with possible application consistency issues?
  - A: have you ever received an email after you placed an order informing you that "we're temporarily out of stock" ☺
- Example: airline reservation system or movie theater seat booking system
  - Customer's transaction does a tentative lock
  - Tentative lock expires after "no activity"
  - Customer understand that she'll have to start over within a given amount of time

Categorizing "Weaker Isolation Levels"

- Dirty read: T_2 read value which never "really existed" in the database
  1. T_1 modifies x
  2. x is then read by T_2
  3. T_1 aborts
- Non-repeatable ("fuzzy") read: the value "did exist", but (visibly) changes during the same transaction
  1. T_1 reads x
  2. T_2 then modifies or deletes x and commits
  3. T_1 tries to read x again and sees a new value
  4. Should (and with what semantics) T_1 commit?
- Phantoms
  1. T_1 retrieves data based on some predicate ("criteria")
  2. T_2 concurrently inserts new tuples (or updates existing tuples) such that these tuples also satisfy T_1's predicate
  3. T_1 repeats the query, this time observes the phantoms that were not retrieved previously

Nobody Allows "Lost Updates"

$$S = \begin{bmatrix} T1 & T2 \\ W(A) & \\ & W(B) \\ W(B) & \\ Com. & \\ & W(A) \\ & Com. \end{bmatrix}$$

- It's impossible to serialize this schedule ☹
  - A serial schedule results either in committing T_1 values for A and B or in committing T_2 values
- This schedule causes a lost update
- The DBMS has allowed txs to overwrite uncommitted data

Nobody Allows “Dirty Writes”

$$S = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ & R(B) \\ & W(B) \\ & Com. \\ R(B) & \\ W(B) & \\ Com. & \end{bmatrix}$$

“Dirty Read” Example

- We’ve already mentioned the “dirty read” phenomenon in which T_2 reads A, which has been modified by uncommitted T_1
- May not be “safe” to use that value ☹

- “dirty writes” are even worse from an isolation perspective
  1. T_1 modifies x
  2. T_2 also modifies x
  3. But: T_1 is still active!
- Easy to break database consistency with dirty write scenarios involving more than one datum
- Also makes it hard (impossible?) for the DBMS to automatically rollback modifies by using “pre-tx” versions of the data

SQL-92 Isolation Levels (I)

- The SQL standard has given official names to different isolation levels
  - The levels are a hierarchy: higher-level isolation provides additional benefits over lower-levels
- Unfortunately: some important isolation levels are ignored by the standard
- Also: (as usual) not all levels are implemented by all databases ☹
  - Example: Oracle (and POSTGRESQL prior to version 9) by default provide an isolation level called snapshot isolation which is not part of the SQL standard
- In SQL, a transaction begins implicitly
- In SQL, a transaction ends when code issues a `commit` or `rollback`
- POSTGRESQL: explicitly start a tx by issuing `begin`
  - Otherwise: POSTGRESQL uses auto-commit mode in which *"each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done)"*

SQL-92 Isolation Levels (II)

(The figure uses an "X" to say *"the specified phenomena can occur under this isolation level"*)

Transaction isolation level	Dirty reads	Nonrepeatable reads	Phantoms
Read uncommitted	X	X	X
Read committed	--	X	X
Repeatable read	--	--	X
Serializable	--	--	--

Source

Lock-Based Protocols For Achieving SQL-92 Isolation Levels

- SERIALIZABLE: Obtain all locks first; plus index locks, plus strict 2PL
- REPEATABLE READS: Same as above, but no index locks (see Textbook for discussion)
- READ COMMITTED: Same as above, but S locks are released immediately
- READ UNCOMMITTED: Same as above, but allows dirty reads (no S locks)

Today's Lecture: Wrapping it Up

Timestamp Ordering: The Basic Idea

Optimistic Concurrency Control: Validation-Based Protocols

Isolation Levels

Readings

- The textbook discusses *timestamp-based protocols* in Chapter 18.5
- The textbook discusses *validation-based protocols* in Chapter 18.6
- The textbook discusses *transaction isolation levels* in Chapter 17.8 and Chapter 17.9