Concurrency Control

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Goals for Schedules

- We want to ensure that schedules of concurrent txns are:
  - *serializable*: equivalent to some serial schedule
  - *recoverable*: ordered so that the system can safely recover from a crash or undo an aborted transaction
  - *cascadeless*: ensure that an abort of one transaction does not produce a series of *cascading rollbacks*
- To achieve these goals, we use some type of *concurrency control mechanism*.
  - *controls* the actions of *concurrent* transactions
  - prevents problematic interleavings

Locking

- Locking is one way to provide concurrency control.
- Involves associating one or more *locks* with each *database element*.
  - each page
  - each record
  - possibly even each collection

Locking Basics

- A transaction must ***request and acquire a lock*** for a data element before it can access it.
- In our initial scheme, every lock can be held by only one txn at a time.

T_1	T_2
l(X)	
r(X)	
	l(X) ***denied; wait for T1***
w(X)	
u(X)	
	l(X) ***granted***
	r(X)
	u(X)

- As necessary, the DBMS:
  - ***denies*** lock requests for elements that are currently locked
  - makes the requesting transaction wait
- A transaction ***unlocks an element*** when it's done with it.
- After the unlock, the DBMS can grant the lock to a waiting txn.
  - we'll show a ***second lock request*** when the lock is granted

Locking and Serializability

- Just having locks isn't enough to guarantee serializability.
- Example: our problematic schedule can still be carried out.

T1

```
read balance1
write(balance1 – 500)



read balance2
write(balance2 + 500)
```

T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 – fee)
```

T_1	T_2
l(bal1);r(bal1) w(bal1); *u(bal1)*	
	l(bal1);r(bal1) *l(bal2)*;r(bal2) w(bal1) *u(bal1);u(bal2)*
l(bal2);r(bal2) w(bal2); *u(bal2)*	

Two-Phase Locking (2PL)

- One way to ensure serializability is *two-phase locking (2PL)*.
- 2PL requires that *all* of a txn's lock actions come before *all* its unlock actions.
- Two phases:
  1. lock-acquisition phase:
     a txn acquires locks, but it doesn't release any
  2. lock-release phase:
     once a txn releases a lock, it can't acquire any new ones
- Reads and writes can occur in both phases.
  - provided that a txn holds the necessary locks
- 2PL is *per-transaction*.
  - one txn could be in its lock-release phase
    while another txn is still in its lock-acquisition phase

Two-Phase Locking (2PL) (cont.)

- In our earlier example, T1 does *not* follow the 2PL rule.

T_1	T_2
l(bal1);r(bal1) w(bal1); ***u(bal1)***	
	l(bal1);r(bal1) *l(bal2)*;r(bal2) w(bal1) *u(bal1);u(bal2)*
l(bal2);r(bal2) w(bal2); *u(bal2)*	

2PL would prevent this interleaving.

- More generally, 2PL produces conflict serializable schedules.

An Informal Argument for 2PL's Correctness

- Consider schedules involving only two transactions.
  To get one that is ***not*** conflict serializable, we need:

  1) at least one conflict that requires T1 → T2
     - T1 operates first on the data item in this conflict
     - T1 must unlock it before T2 can lock it: $u_1(A)$ .. $l_2(A)$

  2) at least one conflict that requires T2 → T1
     - T2 operates first on the data item in this conflict
     - T2 must unlock it before T1 can lock it: $u_2(B)$ .. $l_1(B)$

- Consider all of the ways these pairs of actions could be ordered:

  .. $u_1(A)$ .. $l_2(A)$ .. $u_2(B)$ .. $\boldsymbol{l_1(B)}$ ..
  .. $u_2(B)$ .. $l_1(B)$ .. $u_1(A)$ .. $\boldsymbol{l_2(A)}$ ..
  .. $u_1(A)$ .. $u_2(B)$ .. $\boldsymbol{l_2(A)}$ .. $\boldsymbol{l_1(B)}$ ..
  .. $u_2(B)$ .. $u_1(A)$ .. $\boldsymbol{l_1(B)}$ .. $\boldsymbol{l_2(A)}$ ..
  .. $u_1(A)$ .. $u_2(B)$ .. $\boldsymbol{l_1(B)}$ .. $\boldsymbol{l_2(A)}$ ..
  .. $u_2(B)$ .. $u_1(A)$ .. $\boldsymbol{l_2(A)}$ .. $\boldsymbol{l_1(B)}$ ..

  - none of these are possible under 2PL, because they require at least one txn to lock after unlocking.

The Need for Different Types of Locks

- With only one type of lock, overlapping transactions can't read the same data item, even though two reads don't conflict.
- To get around this, use more than one *mode* of lock.

Exclusive vs. Shared Locks

- An *exclusive lock* allows a transaction to write or read an item.
  - gives the txn exclusive access to that item
  - only one txn can hold it at a given time
  - $xl_i(A)$ = transaction T_i requests an exclusive lock for A
    - if another txn holds *any* lock for A, T_i must wait until that lock is released
- A *shared lock* only allows a transaction to read an item.
  - multiple txns can hold a shared lock for the same data item at the same time
  - $sl_i(A)$ = transaction T_i requests a shared lock for A
    - if another txn holds an *exclusive* lock for A, T_i must wait until that lock is released

Lock Compatibility Matrix

- Used to specify when a lock request for a currently locked item should be granted.

mode of lock requested for item

mode of existing lock for that item (held by a *different* txn)

	shared	**exclusive**
shared	yes	no
exclusive	no	no

Examples of Using Shared and Exclusive Locks

$sl_i(A)$ = transaction T_i requests a shared lock for A

$xl_i(A)$ = transaction T_i requests an exclusive lock for A

- Examples:

T_1	**T_2**
	xl(A); w(A)
sl(B); r(B)	sl(B);r(B)
xl(C); r(C) w(C) u(B); u(C)	u(A); u(B)

without shared locks,T2 would need to wait until T1 unlocked B

Note: T1 acquires an exclusive lock before reading C. ***Why?***

What About Recoverability / Cascadelessness?

- 2PL alone does *not* guarantee either of them.

- Example:

T_1	T_2
	xl(A); w(A)
	sl(C)
	u(A)
xl(A); r(A)	
	r(C); u(C)
w(A); u(A)	
commit	
	commit

2PL?

not recoverable. why not?

not cascadeless. why not?

Strict Locking

- *Strict locking* makes txns hold all *exclusive* locks until they commit or abort.
    - doing so prevents dirty reads, which means schedules will be recoverable and cascadeless

T_1	T_2
	xl(A); w(A)
	sl(C)
	u(A)
xl(A); r(A)	
	r(C); **u(C)**
w(A); ***u(A)***	
commit	
	commit

→

T_1	T_2
	xl(A); w(A)
	sl(C)
xl(A); r(A)	
	r(C); **u(C)**
w(A)	
commit	
u(A)	
	commit
	u(A)

What else needs to change?

Strict Locking

- *Strict locking* makes txns hold all *exclusive* locks until they commit or abort.
  - doing so prevents dirty reads, which means schedules will be recoverable and cascadeless

T_1	T_2
	xl(A); w(A)
	sl(C)
	u(A)
xl(A); r(A)	
	r(C); **u(C)**
w(A); ***u(A)***	
commit	
	commit

→

T_1	T_2
	xl(A); w(A)
	sl(C)
xl(A); wait	
	r(C); **u(C)**
	commit
	u(A)
xl(A); r(A)	
w(A)	
commit	
u(A)	

T1 can't acquire the lock for A until after T2 commits.
Thus, its read of A is not dirty!

- strict + 2PL = strict 2PL

Rigorous Locking

- Under strict locking, it's possible to get something like this:

T_1	T_2	T_3
…		
sl(A); r(A)		
u(A)		
…		
	xl(A); w(A)	
	commit	
	u(A)	
		sl(A); r(A)
		commit
		u(A)
		print A
commit		
print A		

- T3 reports A's new value.
- T1 reports A's old value, even though it commits after T3.
- the ordering of commits (T2,T3,T1) is not same as the equivalent serial ordering (T1,T2,T3)

- *Rigorous locking* requires txns to hold *all* locks until commit/abort.
- It guarantees that transactions commit in the same order as they would in the equivalent serial schedule.
- rigorous + 2PL = rigorous 2PL

Deadlock

- Consider the following schedule:

T_1	T_2
	xl(A);w(A)
sl(B);r(B)	
	xl(B) ***denied; wait for T1***
sl(A) ***denied; wait for T2***	

- This schedule produces *deadlock.*
  - T1 is waiting for T2 to unlock A
  - T2 is waiting for T1 to unlock B
  - neither can make progress!
- We'll see later how to deal with this.

Lock Upgrades

- It can be problematic to acquire an exclusive lock earlier than necessary.
- Instead:
  - acquire a shared lock to read the item
  - *upgrade* to an exclusive lock when you need to write
  - may need to wait to upgrade if others hold shared locks
- Note: we're ***not*** releasing the shared lock before acquiring the exclusive one. why not?

T_1	T_2
xl(A)	
r(A)	
	sl(A) *waits a long time for T1!*
VERY LONG computation	
w(A)	
u(A)	
	r(A) *finally!*

T_1	T_2
sl(A)	
r(A)	
	sl(A)
	r(A) *right away!*
VERY LONG computation	u(A)
xl(A)	
w(A)	
u(A)	

A Problem with Lock Upgrades

- Upgrades can lead to deadlock:
  - two txns each hold a shared lock for an item
  - both txns attempt to upgrade their locks
  - each txn is waiting for the other to release its shared lock
  - *deadlock!*
- Example:

T_1	T_2
	sl(A)
	r(A)
sl(A)	
r(A)	
xl(A)	
denied; wait for T2	
	xl(A)
	denied; wait for T1

Update Locks

- To avoid deadlocks from lock upgrades, some systems provide two different lock modes for reading:
  - shared locks – used if you *only* want to read an item
  - *update locks* – used if you want to read an item *and later update it*

	shared lock	**update lock**
what does holding this type of lock let you do?	read the locked item	read the locked item (in anticipation of updating it later)
can it be upgraded to an exclusive lock?	**no** (not in this locking scheme)	**yes**
how many txns can hold this type of lock for a given item?	an arbitrary number	**only one** *(and thus there can't be a deadlock from two txns trying to upgrade!)*

Different Locks for Different Purposes

- If you only need to read an item, acquire a shared lock.
- If you only need to write an item, acquire an exclusive lock.
- If you need to read and then write an item:
  - acquire an update lock for the read
  - upgrade it to an exclusive lock for the write
  - this sequence of operations is sometimes called *read-modify-write* (RMW)

Compatibility Matrix with Update Locks

mode of existing lock for that item (held by a *different* txn)	mode of lock requested for item: **shared**	**exclusive**	**update**
shared	yes	no	yes
exclusive	no	no	no
update	no	no	no

- When there are one or more shared locks on an item, a txn can still acquire an update lock for that item.
  - allows for concurrency on the read portion of RMW txns
- There can't be more than one update lock on an item.
  - prevents deadlocks when upgrading from update to exclusive
- If a txn holds an update lock on an item, other txns *can't* acquire any *new* locks on that item.
  - prevents the RMW txn from waiting indefinitely to upgrade

Examples of Using Update Locks

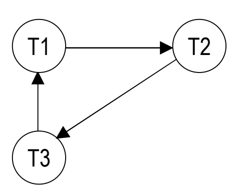
$ul_i(A)$ = T_i requests an update lock for A

T_1	T_2	T_3	
		sl(A) r(A)	
	sl(A) r(A)		
sl(B) r(B) ul(C) r(C)			
	ul(B) r(B)		← **request A?**
		ul(C) r(C)	← **request B**
	xl(A) w(A)		← **request C**
xl(C) w(C) …			← **request D**

Detecting and Handling Deadlocks

- When DBMS detects a deadlock, it roll backs one of the deadlocked transactions.
- Can use a *waits-for graph* to detect the deadlock.
  - the vertices are the transactions
  - an edge from T1 → T2 means T1 is waiting for T2 to release a lock
  - a cycle indicates a deadlock
- Example:

T_1	T_2	T_3
		xl(C)
xl(A)		
		sl(A) ***denied; wait for T1***
	sl(B) sl(C) ***denied; wait for T3***	
xl(B) ***denied; wait for T2***		



cycle – deadlock!

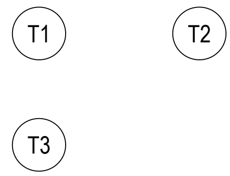
Another Example

- Would the following schedule produce deadlock?

$r_1(B); w_1(B); r_3(A); r_2(C); r_2(B); r_1(A); w_1(A); w_3(C); w_2(A); r_1(C); w_3(A)$

  - assume: no update locks;
    a lock for an item is acquired just before it is first needed

T_1	T_2	T_3
sl(B); r(B)		
xl(B); w(B)		
		sl(A); r(A)
	sl(C); r(C)	



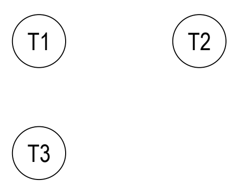
Extra Practice *(try this later on your own!)*

- Would the following schedule produce deadlock?

$w_1(A); w_3(B); r_3(C); r_2(D); r_1(D); w_1(D); w_2(C); r_3(A); w_2(A)$

  - assume: no update locks;
    a lock for an item is acquired just before it is first needed

T_1	T_2	T_3



Optimistic Concurrency Control

- Locking is *pessimistic.*
  - assumes serializability will be violated
  - prevents transactions from performing actions that *might* violate serializability
    - example:

T_1	T_2
	xl(A); w(A)
sl(B); r(B)	
	xl(B)
…	

denied, because T1 *might* read B again

- There are other approaches that are *optimistic*.
  - assume serializability will be maintained
  - only interfere with a transaction if it actually does something that violates serializability
- We'll look at one such approach – one that uses timestamps.

Timestamp-Based Concurrency Control

- In this approach, the DBMS assigns timestamps to txns.
  - TS(T) = the timestamp of transaction T
  - the timestamps must be unique
  - TS(T1) < TS(T2) if and only if T1 started *before* T2
- The system ensures that all operations are consistent with a *serial* ordering based on the timestamps.
  - if TS(T1) < TS(T2), the DBMS only allows actions that are consistent with the serial schedule T1; T2

Timestamp-Based Concurrency Control (cont.)

- Examples of actions that are not allowed:
  - example 1:

actual schedule

T_1	T_2
	TS = 100 r(C)
TS = 102 w(A)	
	r(A)

equivalent serial schedule

T_1	T_2
	TS = 100 r(C) **r(A)** …
TS = 102 **w(A)** …	

not allowed

- T2 starts before T1
- thus, T2 comes before T1 in the equivalent serial schedule (see left)
- in the serial schedule, T2 would *not* see see T1's write
- thus, T2's read should have come *before* T1's write, and we can't allow the read
- we say that ***T2's read is too late***

Timestamp-Based Concurrency Control (cont.)

- Examples of actions that are not allowed:
  - example 2:

actual schedule

T_1	T_2
TS = 205 r(A)	
	TS = 209 r(B)
w(B)	

equivalent serial schedule

T_1	T_2
TS = 205 r(A) **w(B)** …	
	TS = 209 **r(B)** …

not allowed

- T1 starts before T2
- thus, T1 comes before T2 in the equivalent serial schedule (see left)
- in the serial schedule, T2 *would* see T1's write
- thus, T1's write should have come *before* T2's read, and we can't allow the write
- we say that ***T1's write is too late***

Timestamp-Based Concurrency Control (cont.)

- When a txn attempts to perform an action that is inconsistent with a timestamp ordering:
  - the offending txn is rolled back
  - it is restarted with a new, larger timestamp
- With a larger timestamp, the txn comes later in the equivalent serial ordering.
  - allows it to perform the offending operation
- Rolling back the txn ensures that all of its actions correspond to the new timestamp.

Timestamps on Data Elements

- To determine if an action should be allowed, the DBMS associates two timestamps with each data element:
  - a *read timestamp*:
    RTS(A) = the largest timestamp of any txn that has read A
    - the timestamp of the reader that comes latest in the equivalent serial ordering
  - a *write timestamp*:
    WTS(A) = the largest timestamp of any txn that has written A
    - the timestamp of the writer that comes latest in the equivalent serial ordering
    - *the timestamp of the txn that wrote A's current value*

Timestamp Rules for Reads

- When T tries to read A:
  - if TS(T) < WTS(A), roll back T and restart it
    - T comes *before* the txn that wrote A, so T shouldn't be able to see A's current value
    - T's read is too late (see our earlier example 1)
  - else allow the read
    - T comes *after* the txn that wrote A, so the read is OK
    - the system also updates RTS(A):
      RTS(A) = max(TS(T), RTS(A))
      - why can't we just set RTS(A) to T's timestamp?

Timestamp Rules for Reads (cont.)

- Example: assume that T1 wants to read A, and we have the following timestamps:

TS(T1) = 30	WTS(A) = 10
TS(T2) = 50	RTS(A) = 50

- T1 started before T2 (30 < 50)
  - thus T1 comes before T2 in the equivalent serial ordering
- T2 has already read A. How do we know? RTS(A) = TS(T2)
- Despite that, it's okay for T1 to read A.
  - reads don't conflict, so we don't care about the equivalent serial ordering of *two readers* of an item
  - what matters is that T1 comes after the *writer* of A's current value (30 > 10)

Timestamp Rules for Writes

- When T tries to write A:
  - if TS(T) < RTS(A), roll back T and restart it
    - T comes *before* the txn that read A, so that other txn should have read the value T wants to write
    - T's write is too late (see our earlier example 2)
  - else if TS(T) < WTS(A), ignore the write and let T continue
    - T comes *before* the txn that wrote A's current value
    - thus, in the equivalent serial schedule,
      T's write would have been overwritten by A's current value
  - else allow the write
    - how should the system update WTS(A)?

Thomas Write Rule

- The policy of ignoring out-of-date writes is known as the *Thomas Write Rule:*

  …else if TS(T) < WTS(A), ignore the write and let T continue

- What if there is a txn that should have read A *between* the two writes? It's still okay to ignore T's write of A.
  - example:
    - TS(T) = 80, WTS(A) = 100 → we ignore T's write of A
      what if txn U with TS(U) = 90 is supposed to read A?
    - if U had already read A, Thomas write rule wouldn't apply:
      - RTS(A) = 90
      - T would be rolled back because TS(T) < RTS(A)
    - if U tries to read A after we ignore T's write:
      - U will be rolled back because TS(U) < WTS(A)

Example of Using Timestamps

- They prevent our problematic balance-transfer example.

T1

```
read balance1
write(balance1 - 500)


read balance2
write(balance2 + 500)
```

T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

T1	T2	bal1	bal2
		RTS = WTS = 0	RTS = WTS = 0
TS = 350			
r(bal1)		RTS = 350	
w(bal1)		WTS = 350	
	TS = 375		
	r(bal1); r(bal2)	RTS = 375	RTS = 375
	w(bal1)	WTS = 375	
r(bal2)			RTS: no change
w(bal2)			
denied:rollback			

what's the problem here?

Preventing Dirty Reads Using a Commit Bit

- We associate a ***commit bit* c(A)** with each data element A.
  - tells us whether the writer of A's value has committed
  - initially, c(A) is true
- When a txn is allowed to write A:
  - set c(A) to false
  - update WTS(A) as before
- If the timestamps would allow a txn to read A but c(A) is false, the txn is made to wait.
  - preventing a dirty read!
- When A's writer commits, we:
  - set c(A) to true
  - allow waiting txns try again

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
TS = 200		
r(A)		RTS = 200
w(A)		**c = false** WTS = 200
	TS = 210 r(A) ***denied: wait***	
commit		**c = true**
	r(A)?	

Preventing Dirty Reads Using a Commit Bit (cont.)

- If a txn is allowed to write A and c(A) is already false:
  - c(A) remains false
  - update WTS(A) as before
- If the timestamps would cause a txn's write of A to *be ignored* but c(A) is false, the txn must wait.
  - we'll need its write if the writer of A's current value is rolled back

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
	TS = 400 w(A)	**c = false** WTS = 400
TS = 450 w(A)		**c stays false** WTS = 450
	w(A) ***denied: wait***	
commit		**c = true**
	w(A) ***ignored*** …	

Preventing Dirty Reads Using a Commit Bit (cont.)

- Note: c(A) remains false until the writer of the *current value* commits.
- Example: what if T2 had committed after T1's write?

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
	TS = 400 w(A)	**c = false** WTS = 400
TS = 450 w(A)		**c stays false** WTS = 450
	commit	

Preventing Dirty Reads Using a Commit Bit (cont.)

- What happens when a txn T is rolled back?
  - restore the prior state (value and timestamps) of all data elements of which T is the most recent writer
  - set the commit bits of those elements based on whether the writer of the prior value has committed
  - make waiting txns try again
  - in addition, if there were a data element B for which RTS(B) == TS(T), we would restore its old RTS value

T1	T2	A
		RTS = 0 WTS = 0 **c = true**
	TS = 400 w(A)	**c = false** **WTS = 400**
TS = 450 w(A)		**c stays false** WTS = 450
	w(A) ***denied: wait***	
roll back		**WTS = 400** **c = false**
	w(A) ***allowed!***	no changes

Example of Using Timestamps and Commit Bits

- The balance-transfer example would now proceed differently.

T1

```
read balance1
write(balance1 - 500)


read balance2
write(balance2 + 500)
```

T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 - fee)
```

T1	T2	bal1	bal2
		RTS = WTS = 0 c = true	RTS = WTS = 0 c = true
TS = 350 r(bal1) w(bal1)		RTS = 350 WTS = 350; c = false	
	TS = 375 r(bal1) ***denied: wait***		
r(bal2) w(bal2) commit		c = true	RTS = 350 WTS = 350; c = false c = true
	r(bal1) ***and completes***	RTS = 375	

Multiversion Timestamp Protocol

- To reduce the number of rollbacks, the DBMS can keep old versions of data elements, along with the associated timestamps.
- When a txn T tries to read A, it's given the version of A that it should read, based on the timestamps.
  - *the DBMS never needs to roll back a read-only transaction!*

two different versions of A

T1	T2	T3	A(0)	A(105)
	TS = 101		RTS = WTS = 0	
TS = 105			c = true; val = “foo”	
r(A)			RTS = 105	
w(A)				*created*
				RTS = 0; WTS = 105
				c = false; val = “bar”
	r(A): *get A(0)*		no change	
commit				c = true
		TS = 112		
		r(A)		
		get A(105)		RTS = 112

Multiversion Timestamp Protocol (cont.)

- Because each write creates a new version, the WTS of a given version never changes.
- The DBMS maintains RTSs and commit bits for each version, and it updates them using the same rules as before.
- If txn T attempts to write A:
  - find the version of A that T should be overwriting (the one with the largest WTS < TS(T))
  - compare TS(T) with the RTS of that version
  - example: txn T (TS = 50) wants to write A
    - it should be overwriting A(0)
    - show we allow its write and create A(50)?

A(0)	A(105)
RTS = 75	RTS = 0

Multiversion Timestamp Protocol (cont.)

- If T's write of A is *not* too late:
  - create a new version of A with WTS = TS(T)
- Writes are never ignored.
  - there may be active txns that should read that version
- Versions can be discarded as soon as there are no active transactions that could read them.
  - can discard A(t1) if:
    - there is another, later version, A(t2), with t2 > t1

      and
    - there is no active transaction with a TS < t2
  - example: we can discard A(0) as soon as …?

A(0)	**A(105)**
RTS = 75	RTS = 0

Locking vs. Timestamps

- Advantages of timestamps:
  - txns spend less time waiting
  - no deadlocks
- Disadvantages of timestamps:
  - can get more rollbacks, which are expensive
  - may use somewhat more space to keep track of timestamps
- Advantages of locks:
  - only deadlocked txns are rolled back
- Disadvantages of locks:
  - unnecessary waits may occur

The Best of Both Worlds

- Combine 2PL and multiversion timestamping!
- Transactions that perform writes use 2PL.
  - their actions are governed by locks, not timestamps
  - thus, only deadlocked txns are rolled back
- Multiple versions of data elements are maintained.
  - each write creates a new version
  - the WTS of a version is based on when the writer *commits,* not when it started
- Read-only transactions do *not* use 2PL.
  - they are assigned timestamps when they start
  - when T reads A, it gets the version from right before T started
    - will only get a version whose writer has committed
  - read-only txns never need to wait or be rolled back!

Summary: Timestamp Rules for Reads and Writes

when not using commit bits

- When T tries to read A:
  - if TS(T) < WTS(A), roll back T and restart it
    - T's read is too late
  - else allow the read
    - set RTS(A) = max(TS(T), RTS(A))
- When T tries to write A:
  - if TS(T) < RTS(A), roll back T and restart it
    - T's write is too late
  - else if TS(T) < WTS(A), ignore the write and let T continue
    - in the equiv serial sched, T's write would be overwritten
  - else allow the write
    - set WTS(A) = TS(T)

Summary: Timestamp Rules for Reads and Writes

when using commit bits

- When T tries to read A:
  - if TS(T) < WTS(A), roll back T and restart it
    - T's read is too late
  - else allow the read **(but if c(A) == false, make it wait)**
    - set RTS(A) = max(TS(T), RTS(A))

- When T tries to write A:
  - if TS(T) < RTS(A), roll back T and restart it
    - T's write is too late
  - else if TS(T) < WTS(A), ignore the write and let T continue **(but if c(A) == false, make it wait)**
    - in the equiv serial sched, T's write would be overwritten
  - else allow the write
    - set WTS(A) = TS(T) **(and set c(A) to false)**

Summary: Other Details for Commit Bits

- When the writer of *the current value* of data item A commits, we:
  - set c(A) to true
  - allow waiting txns try again

- When a txn T is rolled back, we process:
  - all data elements A for which WTS(A) == TS(T)
    - restore their prior state (value and timestamps)
    - set their commit bits based on whether the writer of the prior value has committed
    - make waiting txns try again
  - all data elements A for which RTS(A) == TS(T)
    - restore their prior RTS

Extra Practice Problem 1

- How will this schedule be executed?

  $w_1(A); w_2(A); r_3(B); w_3(B); r_3(A); r_2(B); w_1(B); r_2(A)$

T1	T2	T3	A	B
			RTS = WTS = 0 c = true	RTS = WTS = 0 c = true

Extra Practice Problem 2

- How will this schedule be executed?

  $r_1(B); r_2(B); w_1(B); w_3(A); w_2(A); w_3(B); commit_3; r_2(A)$

T1	T2	T3	A	B
			RTS = WTS = 0 c = true	RTS = WTS = 0 c = true