

# CSE 444: Database Internals

## Section 6: Transactions

# Today

- Serializability and Conflict Serializability
  - Precedence graph
- Two-Phase Locking
  - Strict two phase locking

# Problem 1: Serializability and Locking

- Is this schedule conflict serializab

What is

- Serializability
- Conflict Serializability?

$T_0$	$T_1$
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	$C_1$
$R_0(B)$	
$W_0(B)$	
$C_0$	

# Review: (Conflict) Serializable Schedule

- A schedule is **serializable** if it is equivalent to a serial schedule
- A schedule is **conflict serializable** if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

# Example:

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



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



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



....

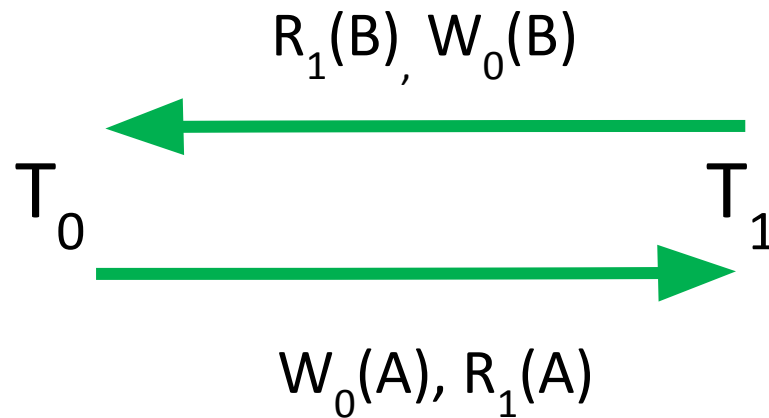
$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

# Problem 1: Serializability and Locking

- Is this schedule conflict serializable?

$T_0$	$T_1$
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	$C_1$
$R_0(B)$	
$W_0(B)$	
$C_0$	

- No.
- The **precedence graph** contains a cycle



- So, use 2PL ...
  - ❑ Original schedule below

$T_0$	$T_1$
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	$C_1$
$R_0(B)$	
$W_0(B)$	
$C_0$	

- So, use 2PL ...

- ❑ Original schedule below

What is

- Two Phase Locking
- Strict Two Phase Locking?

$T_0$	$T_1$
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	$C_1$
$R_0(B)$	
$W_0(B)$	
$C_0$	

# Review:

## (Strict) Two Phase Locking (2PL)

### The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

### Strict 2PL:

All locks held by a transaction are released when the transaction is completed

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit
- Avoids cascading rollbacks

- How can 2PL ensure a conflict-serializable schedule?

□ Original schedule below

$T_0$	$T_1$
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	$C_1$
$R_0(B)$	
$W_0(B)$	
$C_0$	







$T_0$	$T_1$
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
	<b><math>L_1(A) : \text{Block}</math></b>
$L_0(B)$	
$R_0(B)$	
$W_0(B)$	
$U_0(A)$	
$U_0(B)$	
$C_0$	
	<b><math>L_1(A) : \text{Granted}</math></b>
	$R_1(A)$
	<b><math>L_1(B)</math></b>
	$R_1(B)$
	<b><math>U_1(A)</math></b>
	<b><math>U_1(B)</math></b>
	$C_1$

$T_0$	$T_1$
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
	<b><math>L_1(A) : \text{Block}</math></b>
$L_0(B)$	Is this strict 2PL?
$R_0(B)$	
$W_0(B)$	No, release locks after commit
$U_0(A)$	
$U_0(B)$	
$C_0$	
	<b><math>L_1(A) : \text{Granted}</math></b>
	$R_1(A)$
	<b><math>L_1(B)</math></b>
	$R_1(B)$
	<b><math>U_1(A)</math></b>
	<b><math>U_1(B)</math></b>
	$C_1$

- That example ended in a serial schedule, 2PL doesn't necessarily require that
  - Here's an example that doesn't become serial

$T_0$	$T_1$
$R_0(A)$	
$W_0(A)$	
	$R_1(A)$
	$R_1(B)$
	$C_1$
$R_0(B)$	
$W_0(B)$	
$C_0$	







$T_0$	$T_1$
$L_0(A)$	
$R_0(A)$	
$W_0(A)$	
$L_0(B)$	
$U_0(A)$	
	$R_1(A)$
	<b><math>L_1(B)</math>: Block</b>
$R_0(B)$	
$W_0(B)$	
$U_0(B)$	
$C_0$	
	<b><math>L_1(B)</math>: Granted</b>
	$R_1(B)$
	$U_1(A)$
	$U_1(B)$
	$C_1$

We have unlocks in a transaction before committing, so this isn't strict 2 PL

# Common 2PL Misconceptions

- Remember, only one transaction can hold a lock for an element at once
- In 2PL, within each transaction there must be a growing (lock acquiring phase) followed by a shrinking (unlocking) phase
  - Before unlocking, all locks must be granted, not just requested
- Both 2PL and Strict 2PL ensure conflict serializability, neither guarantee a serial schedule or prevent deadlocks

# Lab 3 - Transactions

- NO STEAL / FORCE buffer management policy

- you shouldn't evict dirty (updated) pages from the buffer pool to disk if they are locked by an uncommitted transaction. (this is **NO STEAL**)

- on transaction commit, you should force dirty pages to disk. (e.g., write the pages out) (this is **FORCE**)

- Recommend - locking at page level

- you can acquire locks in BufferPool.getPageO, instead of adding calls to each of your operators. Since we are implementing strict 2PL, we release locks after transaction completes.

- Might have to change previous implementations to access pages using BufferPool.getPageO

# Lab 3 - Transactions (contd.)

- You need to implement shared and exclusive locks
  - Before read, it must have a shared lock or exclusive lock
  - Before write, it must have an exclusive lock
  - Multiple transactions can have a shared lock
  - Only one transaction may have an exclusive lock on an object
  - If transaction  $t$  is the only transaction holding a shared lock on an object  $o$ ,  $t$  may upgrade its lock on  $o$  to an exclusive lock when it needs to write!
- You need to implement strict two-phase locking
  - transactions should acquire the appropriate type of lock on any object before accessing that object
  - transaction shouldn't release any locks until after the transaction commits.

# Lab 3 - Transactions (contd.)

- You will need to implement a LockManager class that will hold data structures to keep track of which locks each transaction holds and that check to see if a lock should be granted to a transaction when it is requested.
- Read about Synchronization in Java, and use the synchronized keyword in appropriate places in LockManager
- You will have to also throw appropriate exceptions like TransactionAbortedException when transaction aborts

# Lab 3 - Transactions (contd.)

## Synchronized method :

Only one thread executing that method per instance

```
public synchronized void releaseLock()
```

```
public void acquireLock() {  
    // wait for lock  
    synchronized (this) {  
        // update state  
    }  
}
```

## Synchronized block :

Only one thread executing that block of code at one time

Usually used to update that object's state synchronously

# Lab 3 - Transactions (contd.)

- Handling deadlocks
  - implement a simple timeout policy that aborts a transaction if it has not completed after a given period of time
  - implement a cycle-detection in a dependency graph data structure, if cycle exists when granting a new lock abort something.
- Design Choices:
  - Locking Granularity: page-level vs tuple-level (our tests assume page-level)
  - Deadlock Detection: timeout vs dependency graphs
  - Deadlock Resolution: aborting yourself vs aborting others
- Read the spec carefully for more details about various methods and edge cases .

# Problem 2: Timestamp-based Concurrency Control

# Timestamp-based Concurrency Control

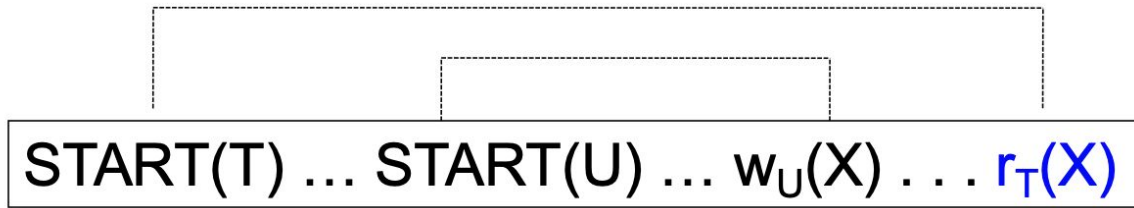
- Some transaction,  $T$ .
- Some element (tuple/page),  $X$ .
- **TS**( $T$ ) - timestamp for transaction  $T$ 
  - Stays constant for all of  $T$ 's operations
- **WT**( $X$ ) – latest write timestamp for  $X$ 
  - Set  $WT(X) = TS(T)$
- **RT**( $X$ ) – latest read timestamp for  $X$ 
  - Set  $RT(X) = TS(T)$
- **C**( $X$ ) –  $X$ 's value has been committed
  - 1 if true, 0 if not

# Timestamp-based Concurrency Control

- **Actions for transaction T**
  - **Grant** a read/write request for a transaction
  - **Abort** (in case T violates physical reality – late actions)
  - **Delay** (make the Grant or Abort decision later)
    - When writing, the change is always tentative until we decide to commit. For this, we use a commit bit C to keep track if the transaction that last wrote X has committed
  - **Ignore *Thomas Write Rule*** – ignore outdated writes

# Timestamp-based Concurrency Control - Four Rules

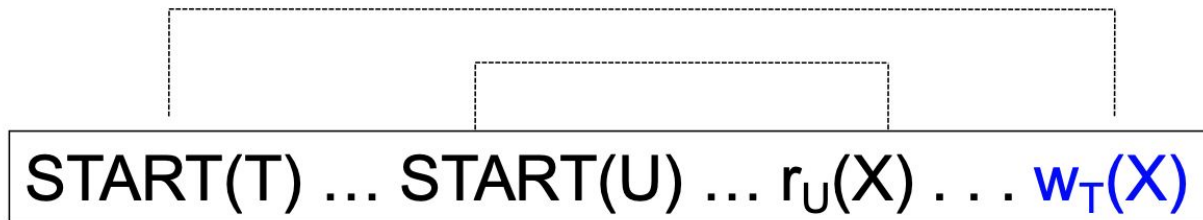
- **Rule 1:** **Read** request on **X** by **T**



- $TS(T) < WT(X)$ , **abort**, (read too late)
- $TS(T) \geq WT(X)$ , physically realizable
  - If  $C = 1$ , **grant**, update  $RT(X)$
  - If  $C = 0$ , **delay** T

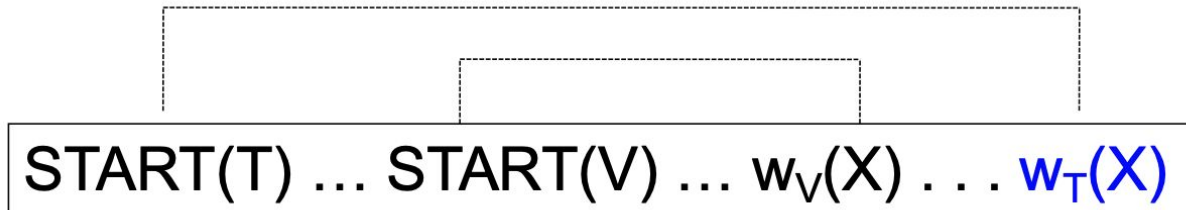
# Timestamp-based Concurrency Control - Four Rules

- Rule 2: Write request on X by T



- $TS(T) < RT(X)$  (write too late)
  - Abort

- $TS(T) \geq RT(X)$ , physically realizable
  - $TS(T) \geq WT(X)$ 
    - then **grant**, update  $WT(X)$ , set  $C = 0$  (as it's not committed yet)



- $TS(T) < WT(X)$ 
  - If  $C = 1$ , **don't write X at all!** (*Thomas Write Rule* – ignore outdated writes)
  - If  $C = 0$ , **delay**

# Timestamp-based Concurrency Control - Four Rules

- **Rule 3: Commit** request by **T**
  - Set  $C = 1$  for all **X** written by **T**
  - Allow waiting transactions to proceed
- **Rule 4: Abort** transaction **T**
  - Check if the waiting transactions can proceed now.

# Summary

## Transaction wants to READ element X

If  $WT(X) > TS(T)$  then ROLLBACK

Else If  $C(X) = \text{false}$ , then WAIT

Else READ and update  $RT(X)$  to larger of  $TS(T)$  or  $RT(X)$

## Transaction wants to WRITE element X

If  $RT(X) > TS(T)$  then ROLLBACK

Else if  $WT(X) > TS(T)$

Then If  $C(X) = \text{false}$  then WAIT

else IGNORE write (**Thomas Write Rule**)

Otherwise, WRITE, and update  $WT(X)=TS(T)$ ,  $C(X)=\text{false}$

# Timestamp-based Concurrency Control

Two transactions get started.

- $\text{Start}(T_1) \rightarrow \text{Start}(T_2)$

# Timestamp-based Concurrency Control

What will happen at the last request?

- $\text{Start}(T_1) \rightarrow \text{Start}(T_2) \rightarrow R_{T_1}(A) \rightarrow R_{T_2}(A) \rightarrow W_{T_1}(B) \rightarrow \mathbf{W_{T_2}(B)}$

# Timestamp-based Concurrency Control

What will happen at the last request?

- $\text{Start}(T_1) \rightarrow \text{Start}(T_2) \rightarrow R_{T_1}(A) \rightarrow R_{T_2}(A) \rightarrow W_{T_1}(B) \rightarrow \mathbf{W_{T_2}(B)}$   
– **ACCEPTED**

# Timestamp-based Concurrency Control

What will happen at the last request?

- $\text{Start}(T_1) \rightarrow \text{Start}(T_2) \rightarrow R_{T_1}(A) \rightarrow R_{T_2}(A) \rightarrow W_{T_1}(B) \rightarrow \mathbf{W_{T_2}(B)}$   
– **ACCEPTED**
- $\text{Start}(T_1) \rightarrow \text{Start}(T_2) \rightarrow R_{T_2}(A) \rightarrow \text{Commit}_{T_2} \rightarrow R_{T_1}(A) \rightarrow \mathbf{W_{T_1}(A)}$

# Timestamp-based Concurrency Control

What will happen at the last request?

- $\text{Start}(T_1) \rightarrow \text{Start}(T_2) \rightarrow R_{T_1}(A) \rightarrow R_{T_2}(A) \rightarrow W_{T_1}(B) \rightarrow \mathbf{W_{T_2}(B)}$   
– **ACCEPTED**
- $\text{Start}(T_1) \rightarrow \text{Start}(T_2) \rightarrow R_{T_2}(A) \rightarrow \text{Commit}_{T_2} \rightarrow R_{T_1}(A) \rightarrow \mathbf{W_{T_1}(A)}$   
– **ABORT**  $T_1$  because  $R_{T_2}(A)$  precedes

# Problem 2: Timestamp-based Concurrency Control

























T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
$R_1(X)$				RT=1		
	$R_2(X)$			RT=2		
	$W_2(X)$			WT=2, C=0		
$W_1(X)$ : abort						
		$W_3(Y)$			WT=3, C=0	
	$W_2(Y)$ : delay					

1. Physically realizable:

$TS(T_2) \geq RT(Y)$  although  $TS(T_2) < WT(Y)$

2. We could not apply Thomas' write rule (**ignore  $W_2(Y)$** ) since  $C=0$





T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
$R_1(X)$				RT=1		
	$R_2(X)$			RT=2		
	$W_2(X)$			WT=2, C=0		
$W_1(X)$ : <b>abort</b>						
		$W_3(Y)$			WT=3, C=0	
	$W_2(Y)$ : <b>delay</b>					
		$C_3$			C=1	

A later write by  $T_3$  has been committed!

T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
R <sub>1</sub> (X)				RT=1		
	R <sub>2</sub> (X)			RT=2		
	W <sub>2</sub> (X)			WT=2, C=0		
W <sub>1</sub> (X): <b>abort</b>						
		W <sub>3</sub> (Y)			WT=3, C=0	
	W <sub>2</sub> (Y): <b>delay</b>					
		C <sub>3</sub>			C=1	
	<b>Ignore W<sub>2</sub>(Y) and proceed</b>					

T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
	Ignore $W_2(Y)$ and <b>proceed</b>					
			$W_4(Z)$			

T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
	Ignore $W_2(Y)$ and <b>proceed</b>					
			$W_4(Z)$			WT=4, C = 0

1. Physically realizable:

$$TS(T_4) \geq RT(Z) \text{ and } TS(T_4) \geq WT(Z)$$

2. Update WT and C (not committed yet)

T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
	Ignore $W_2(Y)$ and <b>proceed</b>					
			$W_4(Z)$			WT=4, C = 0
			$C_4$			C=1

T1	T2	T3	T4	X	Y	Z
1	2	3	4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
	Ignore $W_2(Y)$ and <b>proceed</b>					
			$W_4(Z)$			WT=4, C = 0
			$C_4$			C=1
	$R_2(Z)$					

1. **NOT** Physically realizable:

$$TS(T_2) < WT(Z)$$

Abort/rollback

T4	X	Y	Z
4	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
$W_4(Z)$			WT=4, C = 0
$C_4$			C=1

and proceed

$R_2(Z)$ : abort

# Timestamp-based Concurrency Control

Questions?

# Multiversion Concurrency Control

- Maintains **old** versions of database elements in addition the current version in the database itself.
- The idea is to allow reads that would otherwise result in an abort (as the current version was written by future transaction)

# Problem with Timestamp-Based Scheduling

T1	T2	T3	T4	A
150	200	175	225	RT = 0 WT = 0
$R_1(A)$				RT = 150
$W_1(A)$				WT = 150
	$R_2(A)$			RT = 200
	$W_2(A)$			WT = 200
		$R_3(A)$		
		<b>Abort</b>		
			$R_4(A)$	RT = 225

Had to abort because  
WT(A) is greater than  
my own timestamp

Would have been useful if I  
had access to an old version  
of A (from 150)...

# Multiversion Timestamps

T1	T2	T3	T4	A <sub>0</sub>	A <sub>150</sub>	A <sub>225</sub>
150	200	175	225	RT = 0 WT = 0		
R <sub>1</sub> (A)				RT = 150		
W <sub>1</sub> (A)					Create	
	R <sub>2</sub> (A)				RT=200	
	W <sub>2</sub> (A)					Create
		R <sub>3</sub> (A)			RT=175	
			R <sub>4</sub> (A)			RT=225

Don't have to abort

Just read a previous value of A