University of Windsor

09 Transaction-Dead Lock

Kalyani Selvarajah
School of Computer Science
University of Windsor

Advanced Database Topics
COMP 8157 01/02
FALL 2023

Today's Agenda

2PL

Deadlock

Concurrency Control: Timestamping

https://domains.upperlink.ng/elementor-947/

Introductory Questions

What is the meaning of deadlock and how it can be resolved?

What is the purpose of creating checkpoints during transaction logging?

Recoverability

Serializability identifies schedules that maintain **the consistency of the database**, assuming that none of the transactions in the schedule fails.

- ✓ An alternative perspective examines the **recoverability of transactions** within a schedule.

If a transaction fails,

- ✓ The atomicity property requires that we undo the effects of the transaction.
- ✓ The durability property states that once a transaction commits, its changes cannot be undone (without running another, compensating, transaction).

Recoverable Schedule

Time	T_1	T_2
t_1	BEGIN	
t_2	READ(X)	
t_3	X=X+100	
t_4	WRITE(X)	BEGIN
t_5		READ(X)
t_6		X=X*1.1
t_7		WRITE(X)
t_8		READ(Y)
t_9		Y=Y*1.1
t_{10}		WRITE(Y)
t_{11}	READ(Y)	COMMIT
t_{12}	Y=Y-100	
t_{13}	WRITE(Y)	
t_{14}	ROLL BACK	

T_2 has read the update to X performed by T_1, and has itself updated X and committed the change.

Let's say, we should undo transaction T_2, because it has used a value for X that has been undone. The durability property does not allow this.

This schedule is a *nonrecoverable schedule* which should not be allowed. This leads to the definition of a **recoverable schedule**.

Recoverable Schedule:

A schedule where, for each pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then the commit operation of T_i precedes the commit operation of T_j.

Concurrency Control

The purpose of concurrency control is to prevent two different users (or two different connections by the same user) from trying to update the same data at the same time.

Concurrency control can also prevent one user from seeing out-of-date data while another user is updating the same data.

Concurrency Control Techniques

Serializability can be achieved in several ways;
However, the two main concurrency control techniques that allow transactions to execute safely in parallel.

- ✓ **Locking,**
- ✓ **Timestamping**

Both are conservative approaches (or pessimistic): **delay transactions** in case they conflict with other transactions.

Optimistic methods assume conflict is rare and allow transactions to proceed unsynchronized and check for conflicts only at the end, when a transaction commits.

Locking Methods

Lock: A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.

- ✓ Most widely used approach to ensure serializability.
- ✓ Generally, a transaction must claim a **shared** (read) or **exclusive** (write) lock on a data item before read or write.
- ✓ Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking - Basic Rules

Shared lock: If a transaction has a shared lock on a data item, it can read the item but not update it.

Exclusive lock: If a transaction has an exclusive lock on a data item, it can both read and update the item.

- ✓ Because read operations cannot conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same item.
- ✓ Exclusive lock gives transaction exclusive access to that item.
  - As long as a transaction holds the exclusive lock on the item, no other transactions can read or update that data item.

Compatibility Matrix

	Shared	Exclusive
Shared	✓	✗
Exclusive	✗	✗

Locking - Basic Rules

Locks are used in the following way:

- ✓ Any transaction that needs to access a data item must **first lock the item**, requesting a shared lock for read-only access or an exclusive lock for both read and write access.
- ✓ If the item is not already locked by another transaction, the lock will be **granted**.
- ✓ If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has a shared lock on it, the request will be granted; otherwise, the transaction must **wait** until the existing lock is released.
- ✓ A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions

Some systems allow transaction to **upgrade** read lock to an exclusive lock, or **downgrade** exclusive lock to a shared lock.

Locks

Schedule A

Time	T_1	T_2
t_1	LOCK(A)	
t_2	Read(A)	
t_3		LOCK(A)
t_4	Write(A)	
t_5	Read(A)	
t_6	UNLOCK(A)	
t_7	Read(C)	Read(A)
t_8		Write(A)
t_9	Commit	UNLOCK(A)
t_{10}		Commit

Lock Manager

Granted

Locks

Schedule A

Time	T_1	T_2
t_1	X_LOCK(A)	
t_2	Read(A)	
t_3		X_LOCK(A)
t_4	Write(A)	
t_5	Read(A)	
t_6	UNLOCK(A)	
t_7	Read(C)	Read(A)
t_8		Write(A)
t_9	Commit	UNLOCK(A)
t_{10}		Commit

Lock Manager

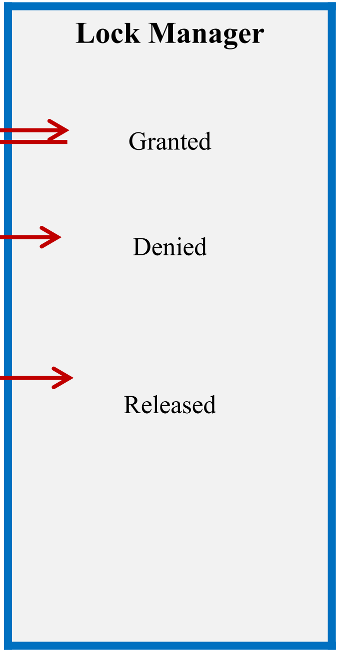
Granted

Denied

Locks

Schedule A

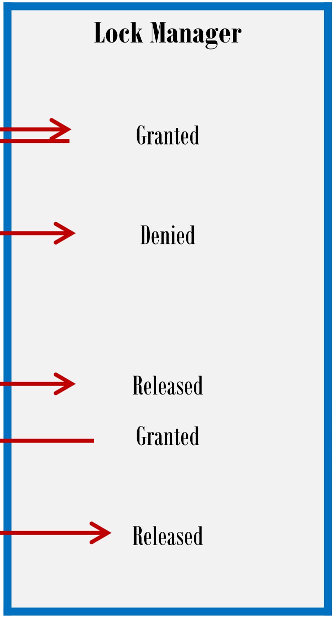
Time	T_1	T_2
t_1	X_LOCK(A)	
t_2	Read(A)	
t_3		X_LOCK(A)
t_4	Write(A)	
t_5	Read(A)	
t_6	UNLOCK(A)	
t_7	Read(C)	Read(A)
t_8		Write(A)
t_9	Commit	UNLOCK(A)
t_{10}		Commit



Locks

Schedule A

Time	T_1	T_2
t_1	X_LOCK(A)	
t_2	Read(A)	
t_3		X_LOCK(A)
t_4	Write(A)	
t_5	Read(A)	
t_6	UNLOCK(A)	
t_7	Read(C)	Read(A)
t_8		Write(A)
t_9	Commit	UNLOCK(A)
t_{10}		Commit



Locks: Problem

Time	T_1	T_2
t_1	BEGIN	
t_2	READ(X)	
t_3	X=X+100	
t_4	WRITE(X)	BEGIN
t_5		READ(X)
t_6		X=X*1.1
t_7		WRITE(X)
t_8		READ(Y)
t_9		Y=Y*1.1
t_{10}		WRITE(Y)
t_{11}	READ(Y)	COMMIT
t_{12}	Y=Y-100	
t_{13}	WRITE(Y)	
t_{14}	COMMIT	

Schedule S1

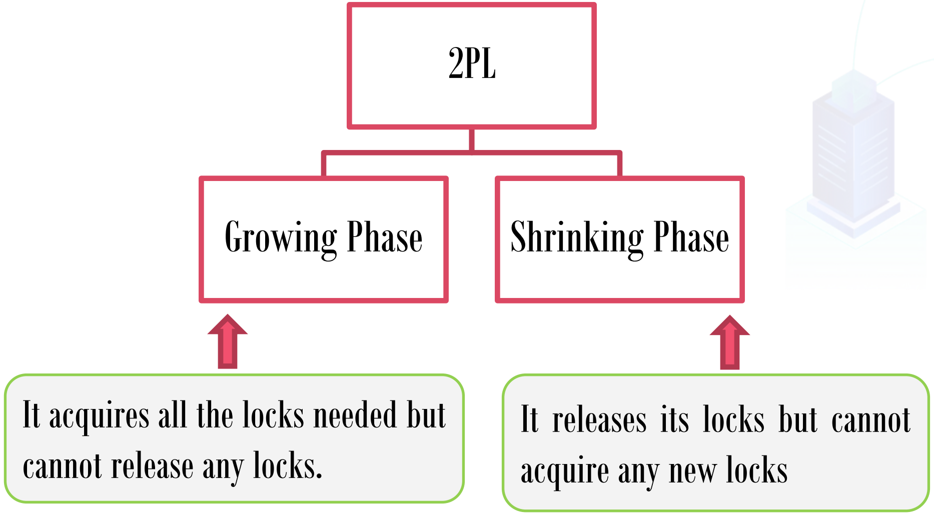
Time	T_1	T_2
t_1	X-LOCK(X)	
t_2	Read(X)	
t_3	Write(X)	
t_4	UNLOCK(X)	
t_5		X-LOCK(X)
t_6		Read(X)
t_7		Write(X)
t_8		UNLOCK(X)
t_9	X-LOCK(Y)	
t_{10}	Read(Y)	
t_{11}	Write(Y)	
t_{12}	UNLOCK(Y)	
t_{13}	Commit	X-LOCK(Y)
t_{14}		Read(Y)
t_{15}		Write(Y)
t_{16}		UNLOCK(Y)
t_{17}		Commit

Schedule S2

Time	T_1	T_2
t_1	X-LOCK(X)	
t_2	Read(X)	
t_3	Write(X)	
t_4	UNLOCK(X)	
t_5		X-LOCK(X)
t_6		Read(X)
t_7		Write(X)
t_8		UNLOCK(X)
t_9		X-LOCK(Y)
t_{10}		Read(Y)
t_{11}		Write(Y)
t_{12}		UNLOCK(Y)
t_{13}	X-LOCK(Y)	Commit
t_{14}	Read(Y)	
t_{15}	Write(Y)	
t_{16}	UNLOCK(Y)	
t_{17}	Commit	

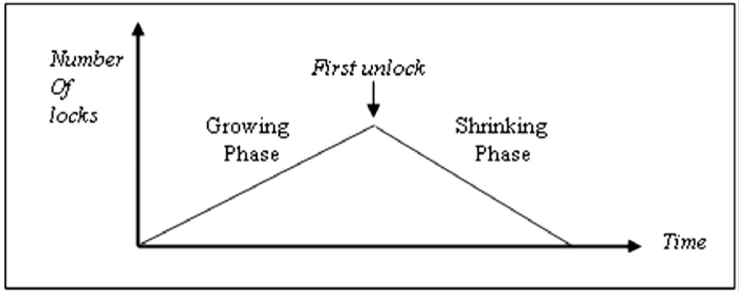
Two-phase locking (2PL)

2PL: A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.

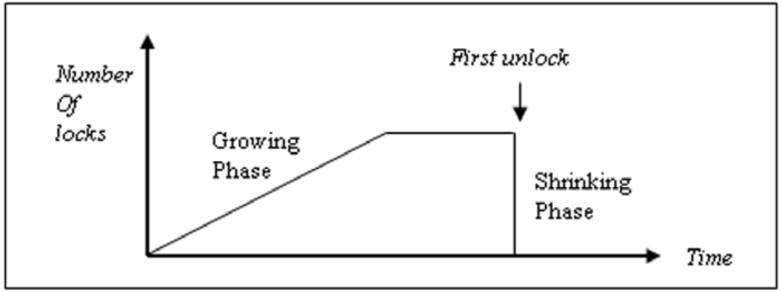


Two-phase locking (2PL)

- There is no requirement that all locks be obtained simultaneously.
- Normally, the transaction acquires some locks, does some processing, and goes on to acquire additional locks as needed.
- However, it never releases any lock until it has reached a stage where no new locks are needed.
- The rules are:
  - A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
  - Once the transaction releases a lock, it can never acquire any new locks.



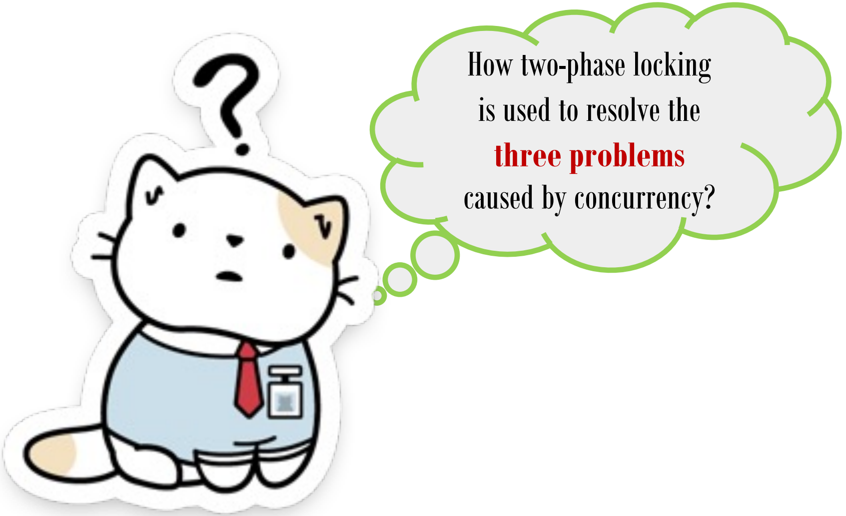
(a)



(b)

Two-phase locking (2PL)

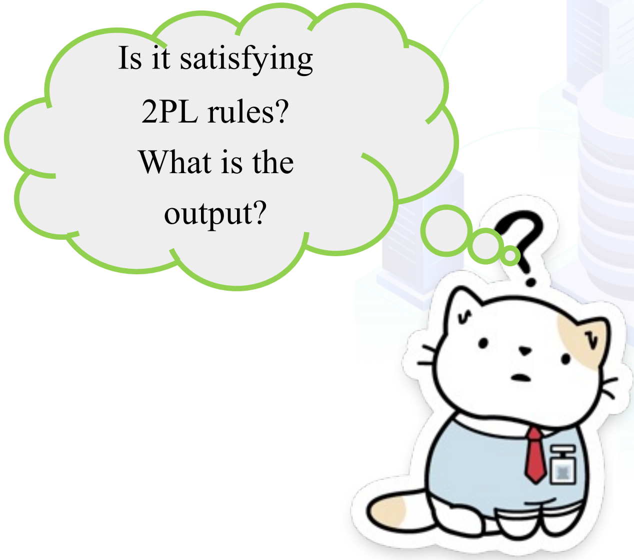
- ✓ If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item.
- ✓ Downgrading can take place only during the shrinking phase.



Two-phase locking (2PL)

Time	T_1	T_2
t_1	X-LOCK(A)	
t_2	Read(A)	
t_3		S-LOCK(A)
t_4	A=A-100	
t_5	Write(A)	
t_6	UNLOCK(A)	
t_7		Read(A)
t_8		UNLOCK(A)
t_9		S-LOCK(B)
t_{10}	X-LOCK (B)	
t_{11}		Read(B)
t_{12}		UNLOCK (B)
t_{13}	Read(B)	PRINT A+B
t_{14}	B=B+100	Commit
t_{15}	Write(B)	
t_{16}	UNLOCK (B)	
t_{17}	Commit	

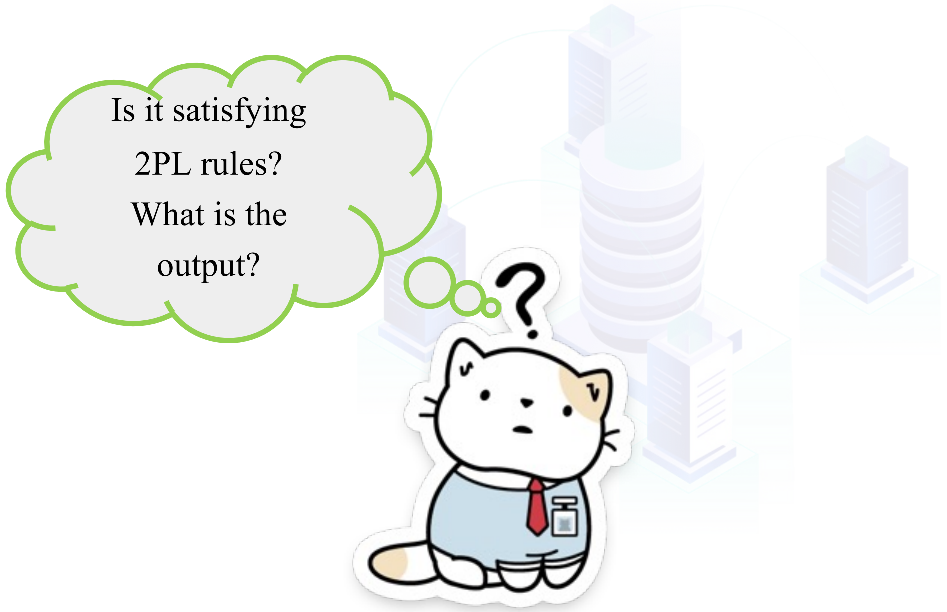
Initial Database state: A=1000, B= 1000



Two-phase locking (2PL)

Time	T_1	T_2
t_1	X-LOCK(A)	
t_2	Read(A)	
t_3		S-LOCK(A)
t_4	A=A-100	
t_5	Write(A)	
t_6	X-LOCK (B)	
t_7	UNLOCK(A)	Read(A)
t_8		S-LOCK(B)
t_9	Read(B)	
t_{10}	B=B+100	
t_{11}	Write(B)	Read(B)
t_{12}	UNLOCK (B)	UNLOCK (B)
t_{13}	Commit	UNLOCK(A)
t_{14}		PRINT A+B
t_{15}		Commit

Initial Database state: A=1000, B= 1000



Preventing the lost update problem using 2PL

Problem:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2	BEGIN	READ(X)	100
t_3	READ(X)	X= X+100	100
t_4	X= X-10	WRITE(X)	200
t_5	WRITE(X)	COMMIT	90
t_6	COMMIT		90

Solution:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2	BEGIN	**X_LOCK(X)**	100
t_3	**X_LOCK(X)**	READ(X)	100
t_4		X= X+100	100
t_5		WRITE(X)	200
t_6		COMMIT/**UNLOCK(X)**	200
t_7	READ(X)		200
t_8	X= X-10		200
t_9	WRITE(X)		190
t_{10}	COMMIT/**UNLOCK (X)**		190

Preventing the uncommitted dependency problem using 2PL

Problem:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2		READ(X)	100
t_3		X= X+100	100
t_4	BEGIN	WRITE(X)	200
t_5	READ(X)	…	200
t_6	X= X-10	ROLLBACK	100
t_7	WRITE(X)		190
t_8	COMMIT		190

Solution:

Time	T_1	T_2	X
t_1		BEGIN	100
t_2		**X_LOCK(X)**	100
t_3		READ(X)	100
t_4	BEGIN	X= X+100	200
t_5	**X_LOCK(X)**	WRITE(X)	200
t_6		ROLLBACK/**UNLOCK(X)**	100
t_7	READ(X)		100
t_8	X= X-10		100
	WRITE(X)		90
	COMMIT/**UNLOCK(X)**		90

Preventing the inconsistent analysis problem using 2PL

Problem:

Time	T_1	T_2	X	Y	Z	SUM
t_1		BEGIN	100	50	25	
t_2	BEGIN	SUM=0	100	50	25	0
t_3	READ(X)	READ(X)	100	50	25	0
t_4	X= X-10	SUM=SUM+X	100	50	25	100
t_5	WRITE(X)	READ(Y)	90	50	25	100
t_6	READ(Z)	SUM=SUM+Y	90	50	25	150
t_7	Z=Z+10		90	50	25	150
t_8	WRITE(Z)		90	50	35	150
t_9	COMMIT	READ(Z)	90	50	35	150
t_{10}		SUM=SUM+Z	90	50	35	185
t_{11}		COMMIT	90	50	35	185

Solution:

Time	T_1	T_2	X	Y	Z	SUM
t_1		BEGIN	100	50	25	
t_2	BEGIN	SUM=0	100	50	25	0
t_3	**X_LOCK(X)**		100	50	25	0
t_4	READ(X)	**S_LOCK(X)**	100	50	25	0
t_5	X= X-10		90	50	25	0
t_6	WRITE(X)		90	50	25	0
t_7	**X_LOCK(Z)**		90	50	25	0
t_8	READ(Z)		90	50	25	0
t_9	Z=Z+10		90	50	25	0
t_{10}	WRITE(Z)		90	50	35	0
t_{11}	COMMIT/**UNLOCK(X,Z)**		90	50	35	0
t_{12}		READ(X)	90	50	35	0
t_{13}		SUM=SUM+X	90	50	35	90
t_{14}		**S_LOCK(Y)**	90	50	35	90
t_{15}		READ(Y)	90	50	35	90
t_{16}		SUM=SUM+Y	90	50	35	140
t_{17}		**S_LOCK(Z)**	90	50	35	140
t_{18}		READ(Z)	90	50	35	140
t_{19}		SUM=SUM+Z	90	50	35	175
t_{20}		COMMIT/ **UNLOCK(X,Y,Z)**	90	50	35	175

Cascading Rollback

- Transaction T_1 obtains an exclusive lock on X and then updates it using Y, which has been obtained with a shared lock, and writes the value of X back to the database before releasing the lock on X.
- Transaction T_2 then obtains an exclusive lock on X, reads the value of X from the database, updates it, and writes the new value back to the database before releasing the lock.
- T_3 share locks X and reads it from the database.
- By now, T_1 has failed and has been rolled back. However, because T_2 is dependent on T_1 (it has read an item that has been updated by T_1), T_2 must also be rolled back.
- Similarly, T_3 is dependent on T_2, so it too must be rolled back. This situation, in which a single transaction leads to a series of rollbacks, is called **Cascading rollback**.

Time	T_1	T_2	T_3
t_1			
t_2	BEGIN		
t_3	**X_LOCK(X)**		
t_4	READ(X)		
t_5	**S_LOCK(Y)**		
t_6	READ (Y)		
t_7	X=Y+X		
t_8	WRITE(X)		
t_9	**UNLOCK(X)**	BEGIN	
t_{10}		**X_LOCK(X)**	
t_{11}		READ(X)	
t_{12}		X=X+100	
t_{13}		WRITE(X)	
t_{14}		**UNLOCK(X)**	
t_{15}	ROLLBACK		
t_{16}			BEGIN
t_{17}			**X_LOCK(X)**
t_{18}		ROLLBACK	
t_{19}			ROLLBACK

Cascading rollback

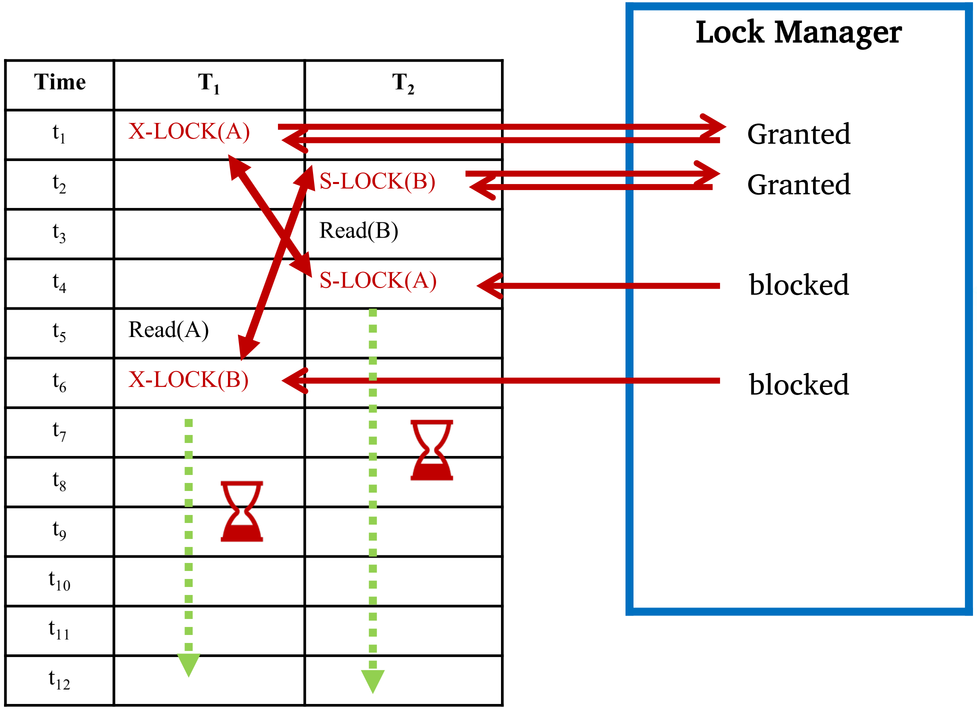
Design protocols that prevent cascading rollbacks. (Cascadeless Schedules)

Solution:

- ✓ **Rigorous 2PL**: hold the release of all locks until the end of the transaction.
- ✓ **Strict 2PL**: holds only exclusive locks until the end of the transaction

Deadlock

Problem with two-phase locking, which applies to all locking-based schemes as transactions can wait for locks on data items.



Deadlock

Three general techniques for handling deadlock:

- ✓ Timeouts
- ✓ Deadlock Detection and Recovery.
- ✓ Deadlock Prevention

1. Timeout

- ✓ A simple approach to deadlock prevention is based on *lock timeouts*.
- ✓ Transaction that requests lock will only wait for a system-defined period of time.
- ✓ If lock has not been granted within this period, lock request times out.
- ✓ In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
- ✓ This is a very simple and practical solution to deadlock prevention that is used by several commercial DBMSs.

2. Deadlock Prevention

- ✓ DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
- ✓ When a transaction tries to acquire a lock that is held by another transaction, the DBMS kills one of them to prevent a deadlock.

2. Deadlock Prevention

- ✓ Assign priorities based on timestamps:

  Older Timestamp = Higher Priority (e.g., $T_1 > T_2$)

- ✓ **Wait-Die ("Old Waits for Young")**
  - If TS(T_i) > TS(T_j), then (T_i older than T_j) T_i is allowed to wait;
  - otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp.*

- ✓ **Wound-Wait ("Young Waits for Old")**
  - If TS(T_i) > TS(T_j), then (T_i older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;*
  - otherwise (T_i younger than T_j) T_i is allowed to wait.

2. Deadlock Prevention (con.)

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		X-LOCK(A)
t_4	X-LOCK(A)	
t_5		
t_6		

$T_1 > T_2$

Time	T_1	T_2
t_1	**BEGIN**	
t_2	X-LOCK(A)	
t_3		**BEGIN**
t_4		X-LOCK(A)
t_5		
t_6		

$T_1 > T_2$

Wait-Die	T_1 Waits	T_2 Aborts
Wound-Wait	T_2 Aborts	T_2 Waits

3. Deadlock Detection and Recovery

- ✓ More practical approach to dealing with deadlock is deadlock detection, where the system checks if a state of deadlock actually exists.

- ✓ Usually handled by construction of **wait-for graph (WFG)** showing transaction dependencies:
  - Create a node for each transaction.
  - Create edge $T_i \rightarrow T_j$, if T_i waiting to lock item locked by T_j.

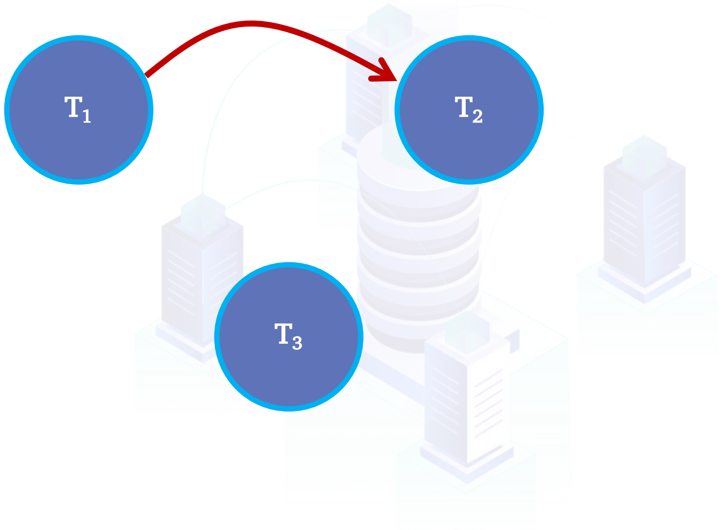
- ✓ Deadlock exists if and only if WFG contains **cycle**.
- ✓ WFG is created at regular intervals.
  - The system will periodically check for cycles in waits-for graph and then make a decision on how to break it.

3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	**BEGIN**	**BEGIN**	**BEGIN**
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			

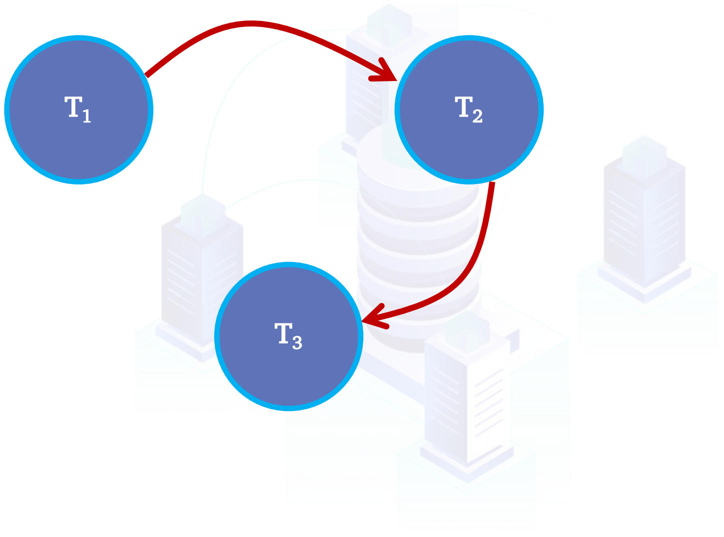
3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	**BEGIN**	**BEGIN**	**BEGIN**
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



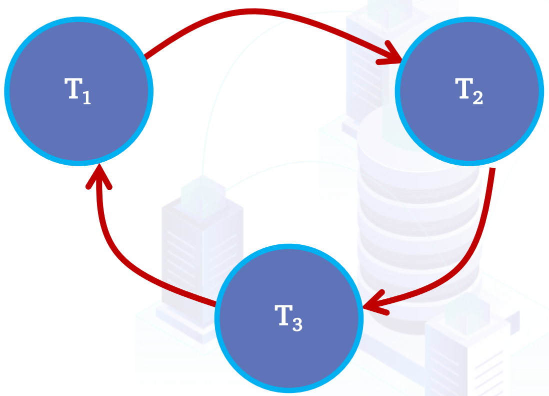
3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	**BEGIN**	**BEGIN**	**BEGIN**
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



3. Deadlock Detection and Recovery

Time	T_1	T_2	T_3
t_1	**BEGIN**	**BEGIN**	**BEGIN**
t_2	S-LOCK(A)		
t_3		X-LOCK(B)	
t_4			S-LOCK(C)
t_5	S-LOCK(B)		
t_6		X-LOCK(C)	
t_7			X-LOCK(A)
t_8			
t_9			



Clearly, the graph has a cycle in it ($T_1 \rightarrow T_2 \rightarrow T_3$), so we can conclude that the system is in deadlock.

Recovery from Deadlock Detection

- Once deadlock has been detected the DBMS needs to abort one or more of the transactions.
- There are several issues that need to be considered:
  - Choice of deadlock victim;
  - How far to roll a transaction back;
  - Avoiding starvation

Choice of deadlock victim

- ✓ In some circumstances, the choice of transactions to abort may be obvious.
- ✓ However, in other situations, the choice may not be so clear.
- ✓ In such cases, we would want to abort the transactions that incur the minimum costs.
- ✓ This may take into consideration:
  - how long the transaction has been running
  - how many data items have been updated by the transaction
  - how many data items the transaction is still to update

How far to roll a transaction back

- ✓ Having decided to abort a particular transaction, we have to decide how far to roll the transaction back.
- ✓ Clearly, undoing all the changes made by a transaction is the simplest solution, although not necessarily the most efficient.
- ✓ It may be possible to resolve the deadlock by rolling back only part of the transaction.

Avoiding starvation

- ✓ Starvation occurs when the same transaction is always chosen as the victim, and the transaction can never complete.
- ✓ Starvation occurs when the concurrency control protocol never selects a particular transaction that is waiting for a lock.
- ✓ Solution:
  - ✓ The DBMS can avoid starvation by storing a count of the number of times a transaction has been selected as the victim and using a different selection criterion once this count reaches some upper limit.

Timestamping Methods

- ✓ Timestamp methods for concurrency control are quite different from locking methods.
- ✓ No locks are involved and therefore there can be **no deadlock**.
- ✓ With timestamp methods, there is no waiting: transactions involved in conflict are simply rolled back and restarted.

Timestamp: A unique identifier created by the DBMS that indicates the relative starting time of a transaction.

- ✓ Timestamps can be generated by simply using the system clock at the time the transaction started, or, more normally, by incrementing a logical counter every time a new transaction starts.
  - ✓ A transaction created at 00:02 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 00:04 is two seconds younger and the priority would be given to the older one.

Timestamping Methods

Timestamping: A concurrency control protocol that orders transactions in such a way that older transactions, transactions with smaller timestamps, get priority in the event of conflict.

The **timestamp-ordering/timestamping protocol** ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- If a transaction attempts to read/write a data item, then the read/write is **only allowed to proceed if last update** on that data item was carried out by an **older transaction.**
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.

Also timestamps for data items:

read-timestamp – giving the timestamp of the last transaction to read the item
write-timestamp – giving the timestamp of the last transaction to write (update) the item

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(T).

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Write (X)
t_4	Read (X)	
t_5		
t_6		

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ **If $TS(T_i) <$ W-timestamp(X)**
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(T).

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Write (X)
t_4	Read (X)	
t_5		
t_6		

$TS(T_1) <$ W-timestamp(X)

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ **If $TS(T_i) <$ W-timestamp(X)**
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(T).

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Write (X)
t_4	Read (X)	
t_5		
t_6		

$TS(T_1) <$ W-timestamp(X)

Read is too late.

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ **If $TS(T_i) <$ W-timestamp(X)**
     - **Operation rejected.**
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(T).

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Write (X)
t_4	Read (X)	
t_5		
t_6		

$TS(T_1) <$ W-timestamp(X)

Read is too late → **Rejected and T_1 rollback**

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - **Operation executed.**
     - R-timestamp(x) = max(TS(Ti), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(Ti).

$TS(T_1) > TS(T_2)$

Time	T_1	T_2
t_1		
t_2		**BEGIN**
t_3		Write (X)
t_4	**BEGIN**	
t_5	Read (X)	
t_6		

$TS(T_1) >=$ W-timestamp(X)

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - **Operation rejected.**
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(T).

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Read (X)
t_4	Write (X)	
t_5		
t_6		

$TS(T_1) <$ R-timestamp(X)

Write is too late → **Rejected**

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ Otherwise, operation executed.
     - W-timestamp(X)= TS(T).

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Write (X)
t_4	Write (X)	
t_5		
t_6		

$TS(T_1) <$ W-timestamp(X)

Write is too late → **Rejected**

Timestamp ordering protocol

- ✓ The timestamp of transaction T_i → $TS(T_i)$.
- ✓ Read time-stamp of data-item X → R-timestamp(X).
- ✓ Write time-stamp of data-item X → W-timestamp(X).

Timestamp ordering protocol works as follows:

1. **If a transaction T_i issues a read(X) operation :**
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) >=$ W-timestamp(X)
     - Operation executed.
     - R-timestamp(x) = max(TS(T), R-timestamp(x)).
2. **If a transaction T_i issues a write(X) operation:**
   - ✓ If $TS(T_i) <$ R-timestamp(X)
     - Operation rejected.
   - ✓ If $TS(T_i) <$ W-timestamp(X)
     - Operation rejected and T_i rolled back.
   - ✓ **Otherwise, operation executed.**
     - W-timestamp(X)= TS(T).

$TS(T_1) > TS(T_2)$

Time	T_1	T_2
t_1		
t_2		**BEGIN**
t_3		Read (X)/ Write(X)
t_4	**BEGIN**	
t_5	Write (X)	
t_6		

$TS(T_1) >$ W-timestamp(X)

Thomas's write rule

- ✓ It can provide **greater concurrency** by rejecting obsolete write operations.
- ✓ It modifies the checks for a **write operation** by transaction T as follows;

If a transaction T_i issues a write(X) operation TS(T) < read_timestamp(x). :

- roll back transaction Ti and restart it using a later timestamp.

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Read (X)
t_4	Write (X)	
t_5		
t_6		

Thomas's write rule

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Read (X)
t_4	Write (X)	
t_5		commit
t_6	**BEGIN**	
t_7		
t_8		
t_9	Write (X)	

Thomas's write rule

- ✓ It can provide **greater concurrency** by rejecting obsolete write operations.
- ✓ It modifies the checks for a **write operation** by transaction T as follows;

If a transaction T_i issues a write(X) operation TS(T)<write_timestamp(x).:

$TS(T_1) < TS(T_2)$

Time	T_1	T_2
t_1	**BEGIN**	
t_2		**BEGIN**
t_3		Write (X)
t_4	Write (X)	
t_5		
t_6		

obsolete value of the item

the write operation can safely be ignored

Summary

We discussed the solution for concurrency control problems: 2PL.

2PL leads to Deadlock: We defined Deadlock and discussed the solutions for deadlock.

We finally discussed another solution for concurrency control problems: Timestamping.