DBMS Concurrency Control

Dr. Geeta Kasana

Computer Science and Engineering Department

TIET

Why use Concurrency method?

Reasons for using Concurrency control method in DBMS:

- To apply Isolation through mutual exclusion between conflicting transactions
- To resolve read-write and write-write conflict issues
- To preserve database consistency through constantly preserving execution obstructions
- The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
- Concurrency control helps to ensure serializability

Database Concurrency Control

- 1 Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.
Following are the Concurrency Control techniques in DBMS:

- Lock-Based Protocols
- Two Phase Locking Protocol (in syllabus)
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock.

Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item.

Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

TYPES OF LOCKS

- **Binary Locks:** A Binary lock on a data item can either locked or unlocked states.
- **Shared/exclusive:** This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

TYPES OF LOCKS cont…

1.**Shared Lock (S):**

A shared lock is also called a **Read-only lock**. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

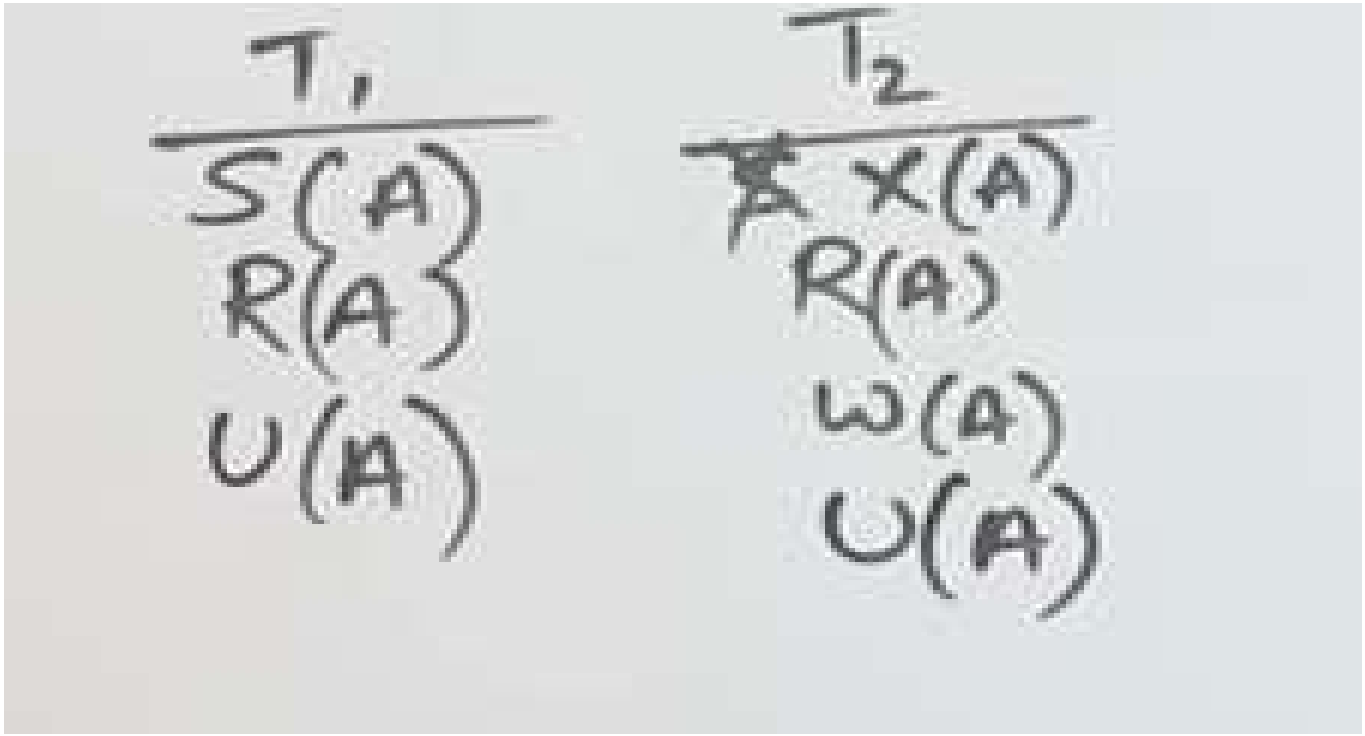
TYPES OF LOCKS cont…

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

TYPES OF LOCKS cont…



Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

- May not sufficient to produce only Serializable schedule
- May not free from Irrecoverability
- May not free from deadlock
- May not free from Starvation

Pitfalls of Lock-Based Protocols(May not sufficient to produce only Serializable schedule)

1.

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
X(B)	
R(B)	
W(B)	
U(B)	

Pitfalls of Lock-Based Protocols(May not free from Irrecoverability)

1.

T1	T2	
X(A)		
R(A)		
W(A)		
U(A)		
	S(A)	
	R(A)	Dirty read problem
	U(A)	
	commit	
Abort		

Pitfalls of Lock-Based Protocols(Deadlock)

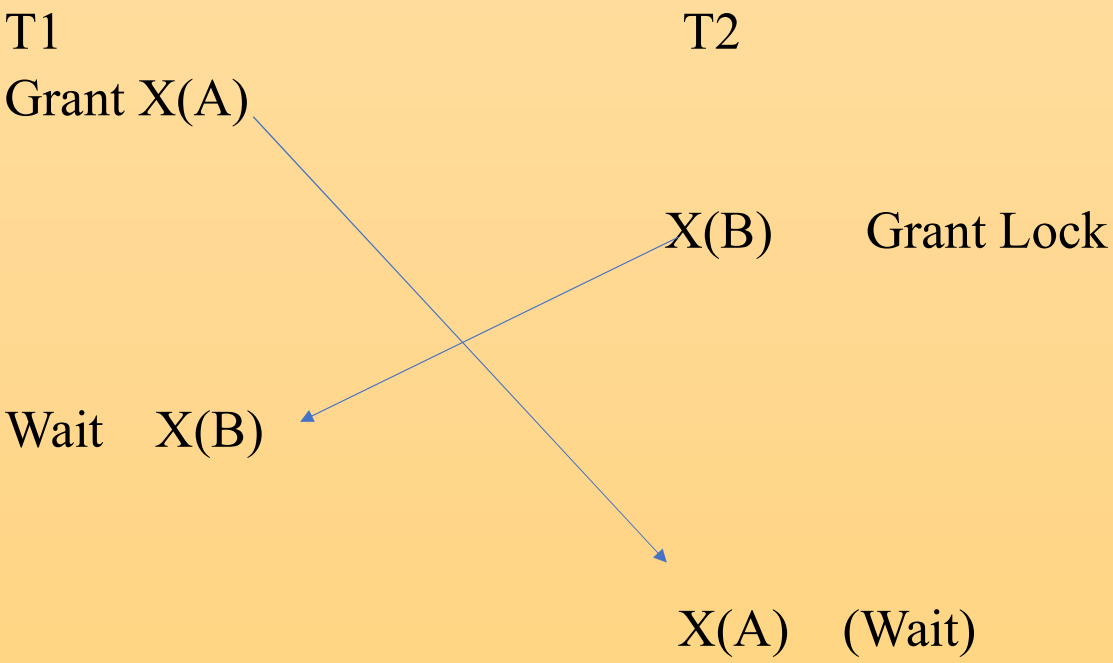
- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols(Deadlock)

- Consider the partial schedule



Deadlock

Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

concepts

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

- When waiting scheme for locked items is not properly managed
- In the case of resource leak
- The same transaction is selected as a victim repeatedly

Two Phase Locking Protocol(2PL)

It is a method of concurrency control in DBMS that ensures serializability by applying a lock to the transaction data which blocks other transactions to access the same data simultaneously. Two Phase Locking protocol helps to eliminate the concurrency problem in DBMS.

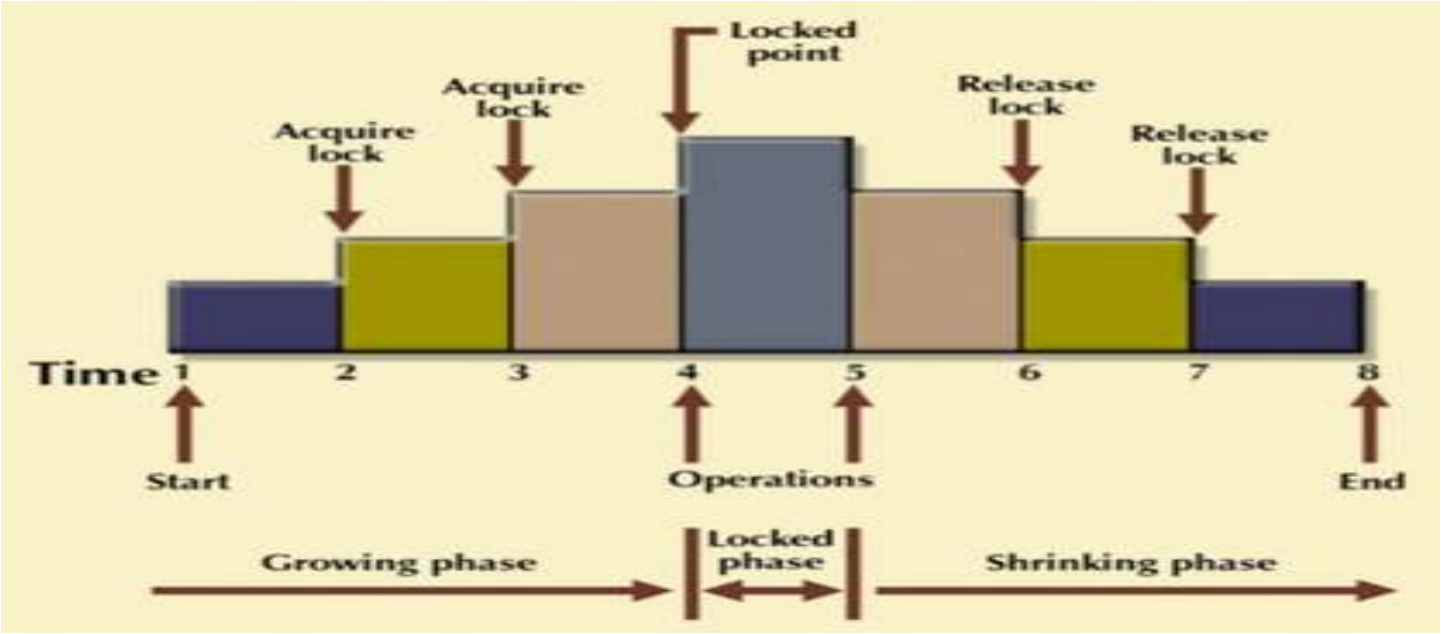
This locking protocol divides the execution phase of a transaction into three different parts.

•In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.

•The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.

•In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

Two Phase Locking Protocol



Two Phase Locking Protocol

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

•**Growing Phase**: In this phase transaction may obtain locks but may not release any locks.

•**Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

Two Phase Locking Protocol

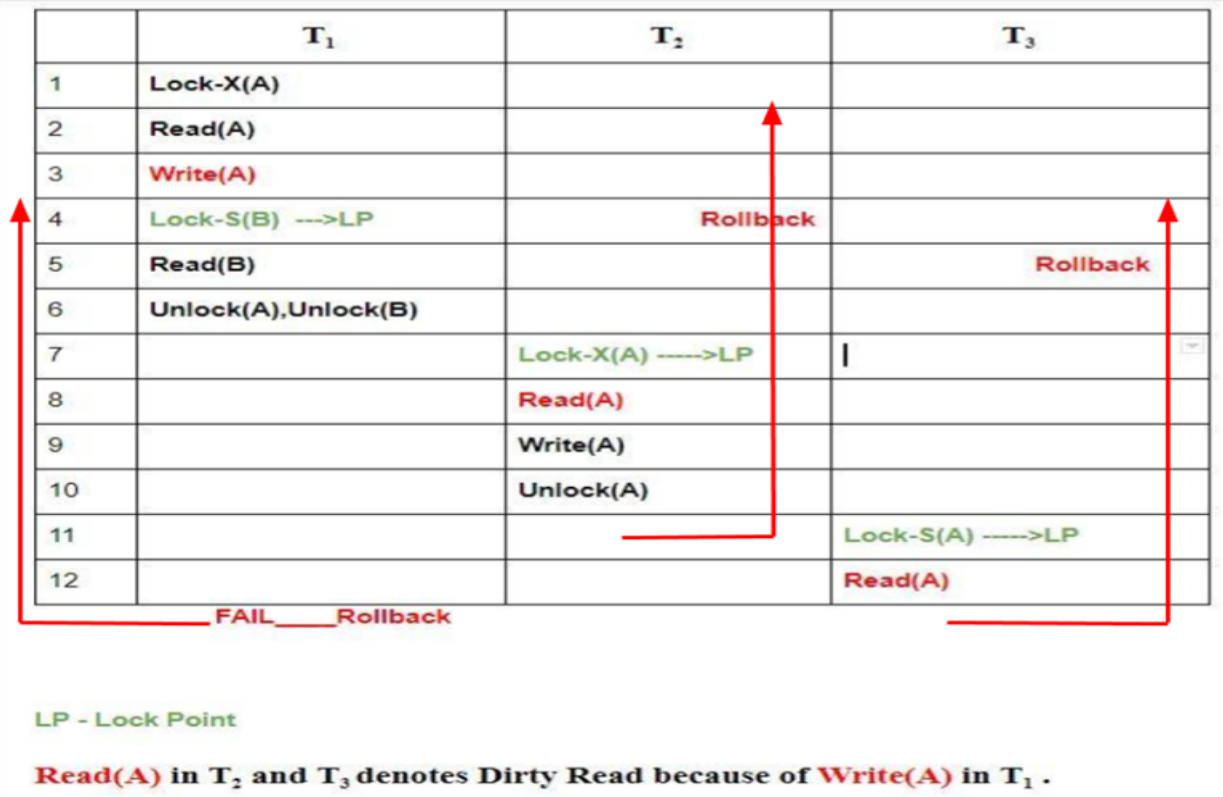
	T_1	T_2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	…….	……
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	…….	……

Transaction T_1:
- The growing Phase is from steps 1-3.
- The shrinking Phase is from steps 5-7.
- Lock Point at 3

Transaction T_2:
- The growing Phase is from steps 2-6.
- The shrinking Phase is from steps 8-9.
- Lock Point at 6

Two Phase Locking Protocol(**Cascading Rollbacks in 2-PL –**

To analyze the schedule. because of Dirty Read in T_2 and T_3 in lines 8 and 12 respectively, when T_1 failed we have to roll back others also. Hence, **Cascading Rollbacks are possible in 2-PL.**



The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Strict Two-Phase Locking

Strict-Two phase locking system is almost similar to 2PL.
The only difference is that Strict-2PL never releases a lock after using it. It holds all the locks until the commit point and releases all the locks at one go when the process is over. **All Exclusive(X) locks** held by the transaction be released until *after* the Transaction Commits.

this protocol requires the transaction to lock all the items it access before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking.

Strict Two-Phase Locking Method

Following Strict 2-PL ensures that our schedule is:

- Recoverable
- Cascadeless

Hence, it gives us freedom from Cascading Abort which was still there in Basic 2-PL and moreover guarantee Strict Schedules but still, *Deadlocks are possible*!

Strict Two-

Strict 2PL - In strict 2PL, all write locks are released at the end of commit of the transaction whereas all read locks are released immediately after the data are consumed.

Transaction T1	*Transaction T2*	*Transaction T3*
Lock-X(x)		
Read(x)		
Lock-S(y)		
Read(y)		
Write(x)		
Unlock(x)	Lock-X(x)	
Unlock(y)	Read(x)	
.	Write(x)	
.	Unlock(x)	Lock-S(x)
.	.	Read(x)
.	.	Unlock(x)
.	.	.
.	.	.
.	.	.
.	.	
.		
.		

For example, let us consider the partial schedule S1 given above with three transactions T1, T2, and T3. In all the three transactions, first all the locks are acquired and then they are released one by one upon consumption. Hence, all the transactions in schedule S1 are two phase transactions.

The Two-Phase Locking Protocol (Cont.)
Example of **Rigorous two-phase locking protocol**

	T_1	T_2
1	Read(A)	
2		Read(A)
3	Read(B)	
4	Write(B)	
5	**Commit**	
6		Read(B)
7		Write(B)
6		**Commit**

The schedule is conflict serializable, so we can try implementing 2-PL.

	T_1	T_2
1	Lock-S(A)	
2	Read(A)	
3		Lock-S(A)
4		Read(A)
5	Lock-X(B)	
6	Read(B)	
7	Write(B)	
8	**Commit**	
9	Unlock(A)	
10	Unlock(B)	
11		Lock-X(B)
12		Read(B)
13		Write(B)
14		**Commit**
15		Unlock(A)
16		Unlock(B)

Solution! We can have different sequence


Thank
You