Database Management Systems

Dr. Santhosh Manikonda
Department of CSE
School of Engineering

Malla Reddy University, Hyderabad

Syllabus
UNIT - IV

- Introduction to Transaction Processing
- Transaction and System Concepts
- ACID Properties
- Characterizing Schedules Based on Recoverability.

Transaction

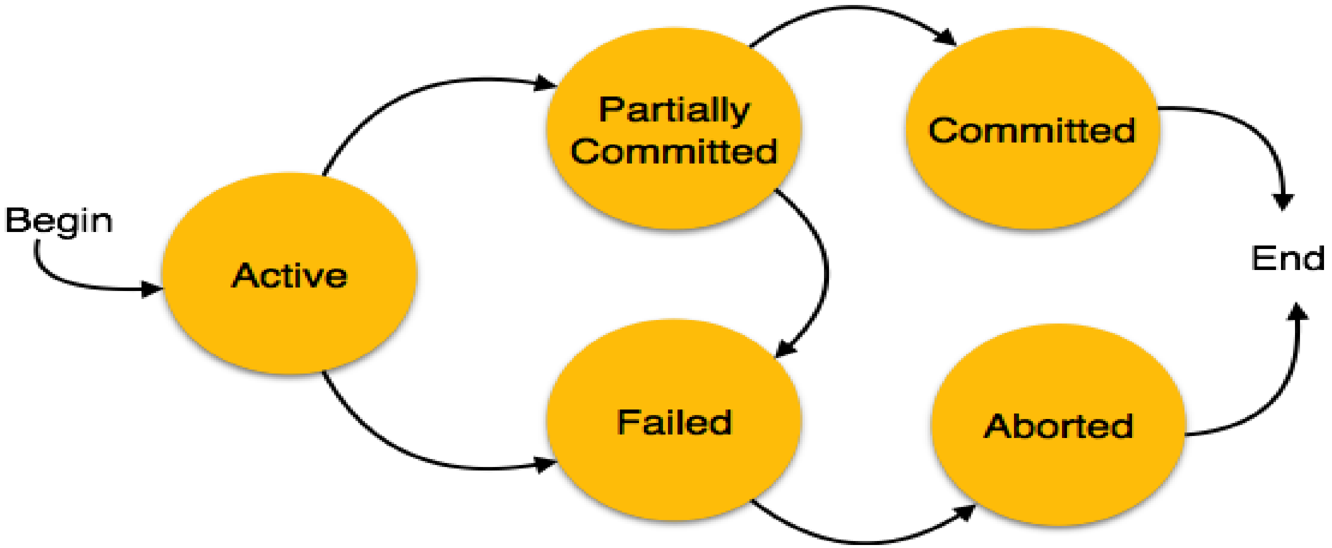
- Transaction: Set of logically related operations to perform a certain task
  - Read Operation – R(A) – Access the values from database
  - Write Operation – W(A) – Modify/Change the values in database

T1	**T2**	**Conflict**
R(A)	R(A)	No
R(A)	W(A)	Yes
W(A)	R(A)	Yes
W(A)	W(A)	Yes

ACID Properties

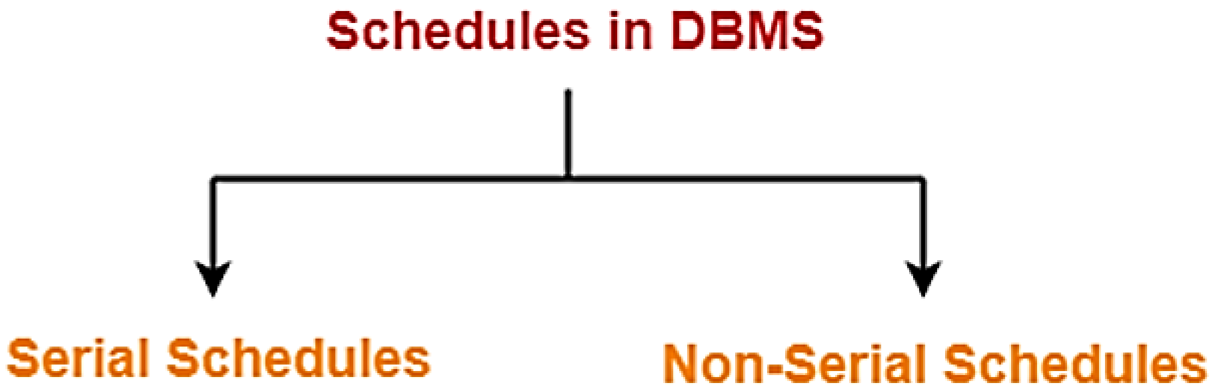
- Atomicity
  - Transaction must be treated as an atomic unit
- Consistency
  - DBMS is in a valid state before and after the transaction – all of the data is valid
- Isolation
  - One transaction should start execution only when the other finished execution - possibility to convert parallel schedule to serial schedule
- Durability
  - Changes made in the database should be permanent even if there is a system failure

Transaction state diagram



Serializability - Schedules

- The order in which the operations of multiple transactions appear for execution is called as a schedule
  - Serial Schedules
  - Parallel/Non-Serial Schedules



Serial and Parallel Schedules

1. Serial schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)

T1 → T2

T1	T2
	R(A)
	W(A)
R(A)	
W(A)	

T1 ← T2

- All the transactions execute serially one after the other.
- When one transaction executes, no other transaction is allowed to execute.
- Serial schedules are always:
  - Consistent
  - Recoverable
  - Low Throughput

Serial and Parallel Schedules

2. Parallel schedule

T1	T2
R(A)	
	R(A)
	W(A)
W(A)	

$T1 \rightarrow T2$

$T1 \leftarrow T2$

T1	T2
	R(A)
R(A)	
W(A)	
	W(A)

$T1 \leftarrow T2$

$T1 \rightarrow T2$

- Multiple transactions execute concurrently.
- Operations of all the transactions are inter leaved or mixed with each other.
- Parallel schedules are always:
  - Not Consistent
  - Not Recoverable
  - High Throughput

Problems with Concurrency

- Dirty Read
- Incorrect summary
- Lost update
- Unrepeatable read
- Phantom read

Problems with Concurrency

- Dirty Read

T1	T2	
R(A) – 100 A-50=50 W(A) – 50 . . . . . Transaction Failed	 R(A) – 50 A+20 = 70 W(A) - 70 Commit	A=100 A=100 A=70

occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

Problems with Concurrency

- Dirty Read

T1	T2	
R(A) – 100 A-50=50 W(A) – 50 . . . . . Transaction Failed	 R(A) – 50 A+20 = 70 W(A) - 70 Commit	A=100 A=100 A=70

Dirty Read

Problems with Concurrency

- Incorrect summary

Arises when a transaction performs aggregate functions in between another transactions

T1	T2	A=1000, B=1000, C=1000
	Sum=0	Sum=0
	Avg=0	Avg=0
	R(C)	T2:R(C)=1000
	Sum=sum+C	Sum=1000
R(A)		T1: R(A)=1000
A-50		
W(A)		T1:W(A):950
	R(B)	T2: R(B)= 1000
	Sum=Sum+B	Sum=1950
	Sum=Sum+C	Sum=2950
	Avg=Sum/3	Avg=983.33
	Commit	
R(B)		T2: R(B)=1000
B=B+50		T2: W(B)=1050
W(B)		
Commit		

Problems with Concurrency

- Incorrect summary

Arises when a transaction performs aggregate functions in between another transactions

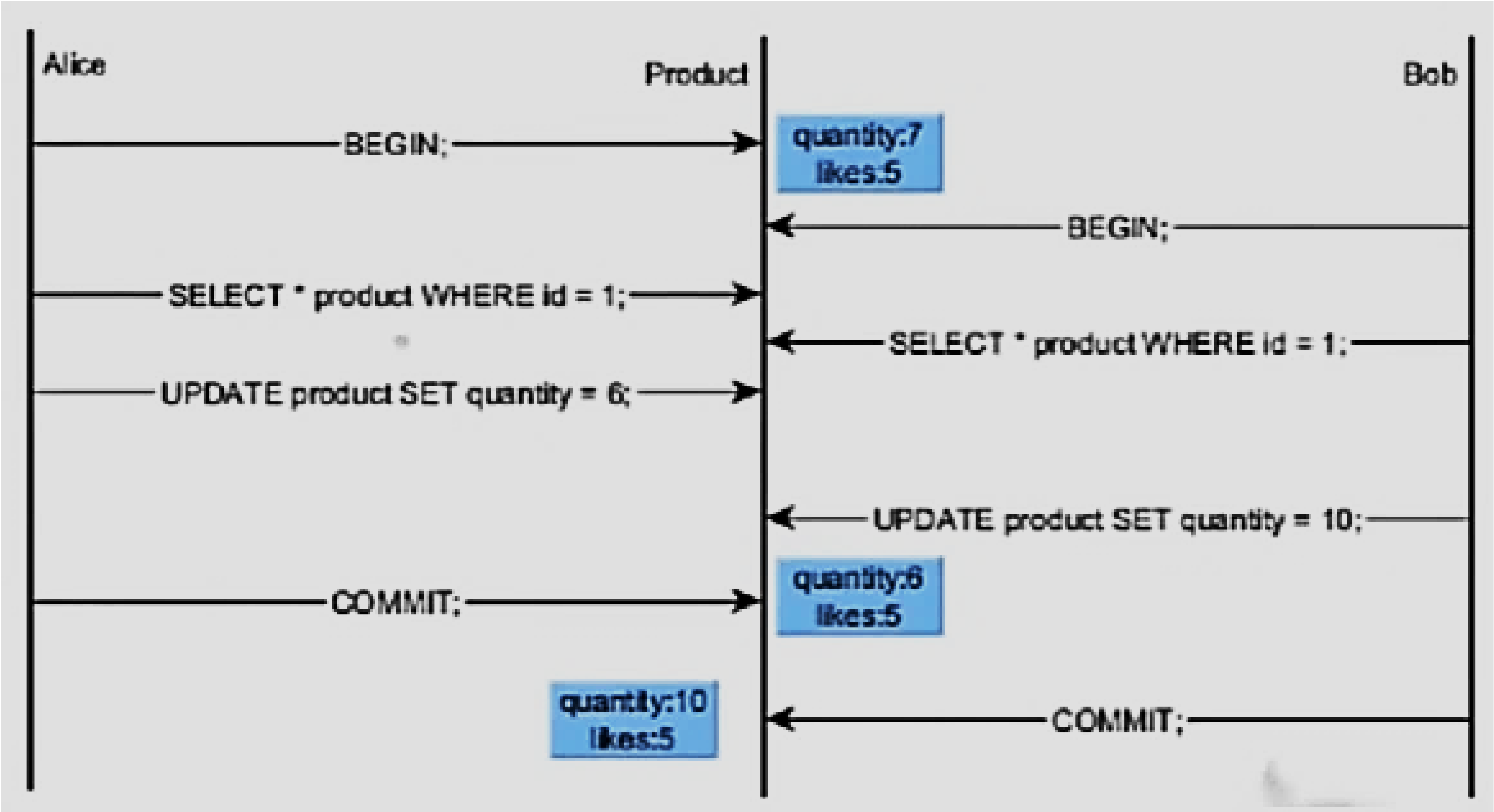
Incorrect Summary

T1	T2	A=1000, B=1000, C=1000
	Sum=0	Sum=0
	Avg=0	Avg=0
	R(C)	T2:R(C)=1000
	Sum=sum+C	Sum=1000
R(A)		T1: R(A)=1000
A-50		
W(A)		T1:W(A):950
	R(B)	T2: R(B)= 1000
	Sum=Sum+B	Sum=1950
	Sum=Sum+C	Sum=2950
	Avg=Sum/3	Avg=983.33
	Commit	
R(B)		T2: R(B)=1000
B=B+50		T2: W(B)=1050
W(B)		
Commit		

Problems with Concurrency

- Lost update

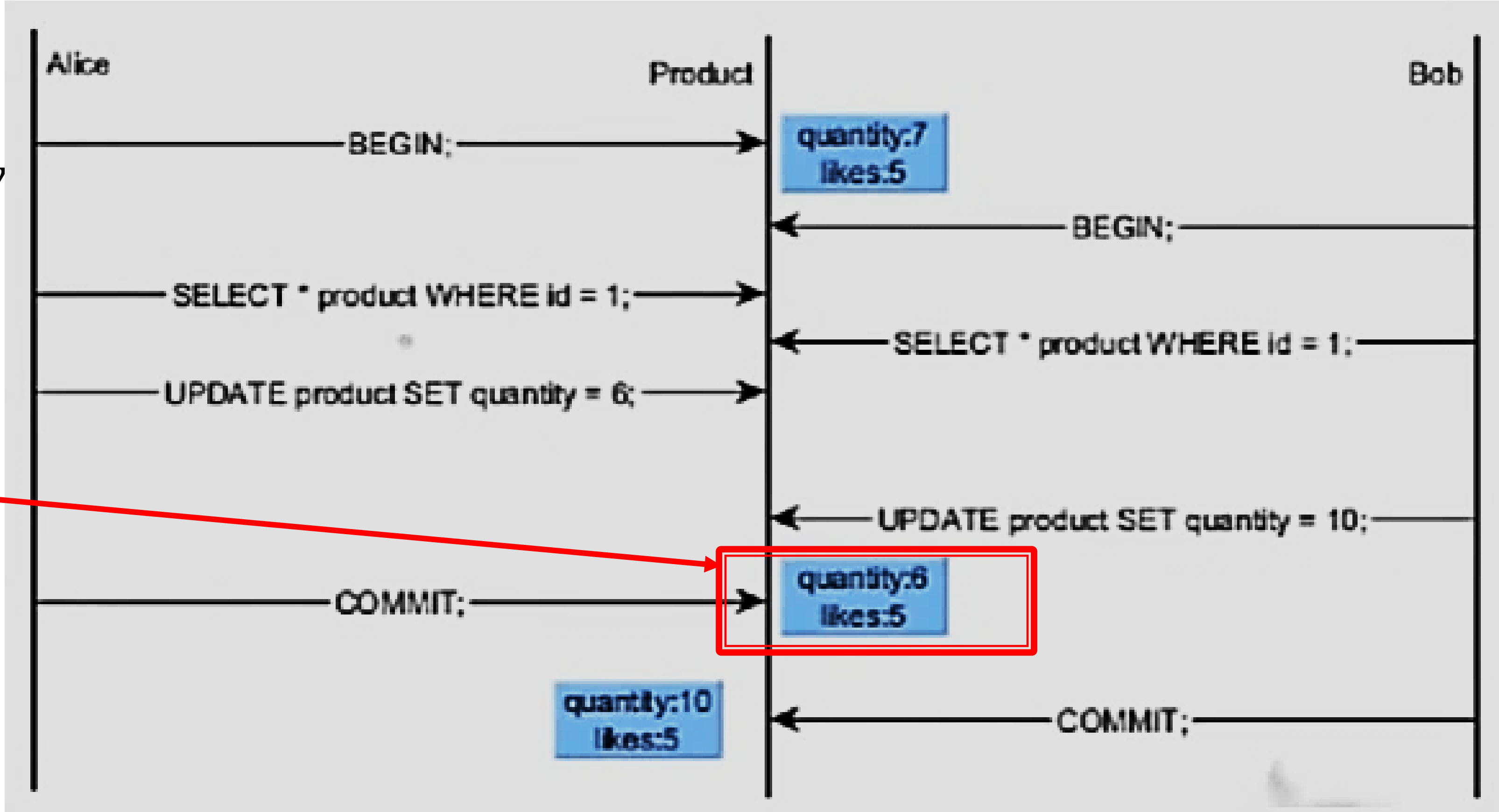
Changes made by one transaction are updated by another transaction.



Problems with Concurrency

- Lost update

Changes made by one transaction are updated by another transaction.



Lost Update

Problems with Concurrency

- Unrepeatable read

Reading before a transaction is committed leads to unrepeatable read

T1	T2	A=100
R(A)		A=100
	R(A)	T2:R(A)=100
A-50		
W(A)		
Commit		
	R(A)	T2:R(A)=50

Problems with Concurrency

- Unrepeatable read

Reading before a transaction is committed leads to unrepeatable read

Unrepeatable Read

T1	T2	A=100
R(A)		A=100
	R(A)	T2:R(A)=100
A-50		
W(A)		
Commit		
	R(A)	T2:R(A)=50

Problems with Concurrency

- Phantom read

Reading after some data is deleted leads to phantom read

T1	T2	A=100
R(A)		A=100
	R(A)	T2:R(A)=100
Delete (A)		
Commit		
	R(A)	T2:R(A)= ------

Problems with Concurrency

- Phantom read

Reading after some data is deleted leads to phantom read

Phantom Read

T1	T2	A=100
R(A)		A=100
	R(A)	T2:R(A)=100
Delete (A)		
Commit		
	$\boxed{\text{R(A)}}$	T2:R(A)= ------

Serializability

- Some non-serial schedules may lead to inconsistency of the database
- Serializability helps to identify which non-serial schedules are correct and will maintain the consistency of the database
- If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a *serializable schedule*.

Serializability

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)

T1	T2
	R(A)
	W(A)
R(A)	
W(A)	

Serializability

T1	T2
R(A)	
	R(A)
	W(A)
W(A)	

- *Parallel Schedule*

- *Finding a Serial schedule for the given parallel schedule is known as serializability*
- *It can be either of these possibilities*
  - *T1 → T2*

    or
  - *T2 → T1*

- Conflict Serializability
- View Serializability

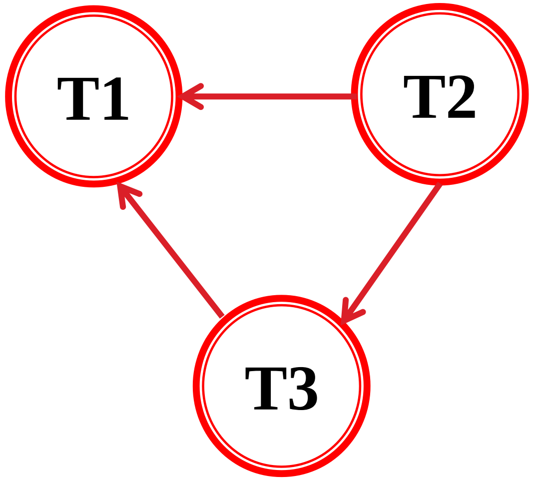
Conflict Serializability

T1	T2	T3
R(A)		
		R(B)
		R(A)
	R(B)	
	R(C)	
		W(B)
	W(C)	
R(C)		
W(A)		
W(C)		

- *Start with the first operation and check for conflict pair in other transactions*
- *If you find conflict pair draw a directed graph*
- *If loop does not exists in the final precedence graph, then it is conflict serializable schedule*

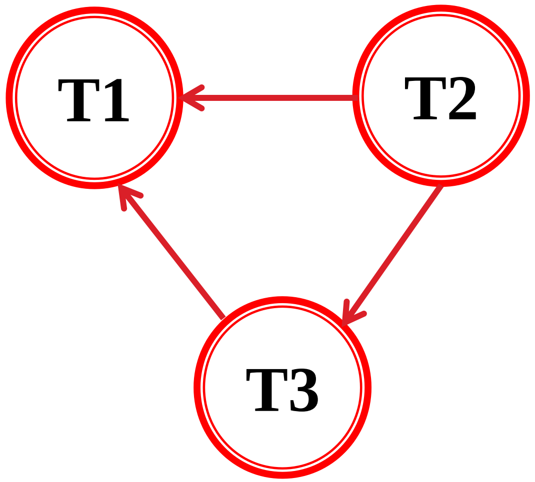
Conflict Serializability

T1	T2	T3
R(A)		
		R(B)
		R(A)
	R(B)	
	R(C)	
		W(B)
	W(C)	
R(C)		
W(A)		
W(C)		



Conflict Serializability

T1	T2	T3
R(A)		
		R(B)
		R(A)
	R(B)	
	R(C)	
		W(B)
	W(C)	
R(C)		
W(A)		
W(C)		



- *T2 (No incoming edge – First transaction)*
- *T3 (No incoming edge after removing T2 - second transaction)*
- *T1 (Last Transaction)*

- *T2 → T3 → T1 is the equivalent serial schedule*

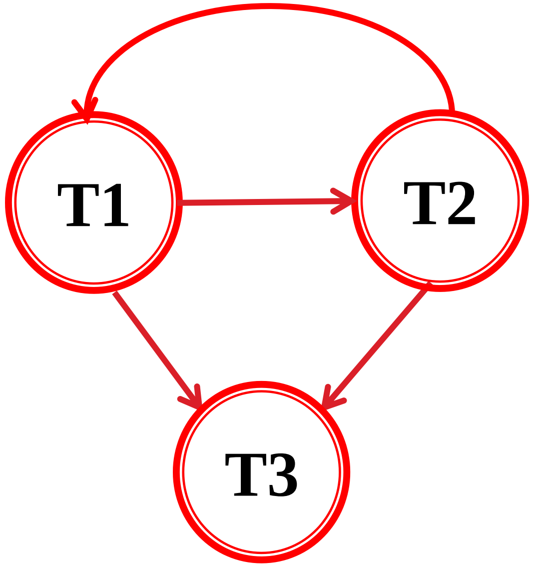
Conflict Serializability

T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)

Check for Conflict serializability

Conflict Serializability

T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)



- *If Loop exists → Cannot say whether the schedule is serializable or not*
    - *We will check for View Serializability*

View Serializability

A = 100

S

T1	T2	T3
R(A)		
	A-30	
	W(A)	
A-30		
W(A)		
		A-30
		W(A)

A = 10

Re-write S as S'

A = 100

S'

T1	T2	T3
R(A)		
A-30		
W(A)		
	A-30	
	W(A)	
		A-30
		W(A)

A = 10

View Serializability

A = 100

S

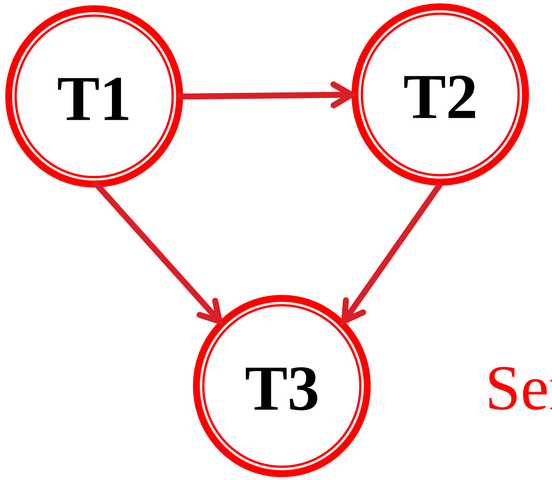
T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)

Re-write S as S’

A = 100

S’

T1	T2	T3
R(A)		
W(A)		
	W(A)	
		W(A)



Serializable

Concurrency Control Protocols

- Primary goal of concurrency protocols is to achieve consistency
- 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.

- This goal is achieved by using different protocols
  - Shared/Exclusive Lock
  - Two – Phase Locking

Concurrency Control Protocols

- Primary goal of concurrency protocols is to achieve consistency
- This goal is achieved by using different protocols
    - Shared/Exclusive Lock
    - Two – Phase Locking

Shared/Exclusive Lock

- Shared Lock (S): In shared lock, a transaction is allowed to only read
  - Shared lock allows another transaction(s) for shared lock only
- Exclusive Lock (X): In exclusive lock, a transaction is allowed to read and write
  - Exclusive lock does not allow any lock until it is released
  - 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.

Shared/Exclusive Lock

		Request	
		S	X
Grant	S	**YES**	**NO**
	X	**NO**	**NO**

Shared/Exclusive Lock

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

Shared/Exclusive Lock

- Problems in Shared Lock/Exclusive Lock
  - May not be sufficient to achieve serializable schedule
  - May not be recoverable
  - May not be free from deadlock
  - May not be free from starvation

Shared/Exclusive Lock

- May not be recoverable

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	X(A)
	R(A)
	W(A)
	U(A)
	Commit
X(B)	
R(B)	
W(B)	
U(B)	
Failed	

Shared/Exclusive Lock

- May not be recoverable

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	X(A)
	R(A)
	W(A)
	U(A)
	Commit
X(B)	
R(B)	
W(B)	
U(B)	
Failed	

Shared/Exclusive Lock

- May not be free from deadlock

	T1	T2	
	X(A)		
	R(A)		
	W(A)		
		X(B)	
		R(B)	
		W(B)	
Wait	X(B)		
	R(B)		
	W(B)		
		X(A)	Wait

Shared/Exclusive Lock

- May not be free from starvation

T1	T2	T3	T4
S(A)			
	X(A)		
		S(A)	
U(A)			
			S(A)
		U(A)	

Two – Phase Lock (2PL)

- Growing Phase:
  - In this phase a transaction may acquire locks
  - Transaction may not release locks

- Shrinking Phase:
  - In this phase Transaction may release locks
  - Transaction may not obtain locks

- The protocol assures serializability.
- It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

Two – Phase Lock (2PL) – Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Two – Phase Lock (2PL)

Growing Phase

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

Shrinking Phase

Two – Phase Lock (2PL)

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

Automatic Acquisition of Locks

- A transaction Ti issues the standard read/write instruction, without explicit locking calls.
- The operation read(D) is processed as:

```
 if Ti has a lock on D
      then
     read(D)
else
      begin
           if necessary wait until no other
           transaction has a lock-X on D
         grant Ti a  lock-S on D;
         read(D)
end
```

Automatic Acquisition of Locks

- The operation write(D) is processed as:

```
if Ti has a  lock-X on D
      then
       write(D)
    else begin
         if necessary wait until no other transaction
      has any lock on D,
         if Ti has a lock-S on D
              then
                 upgrade lock on D  to lock-X
             else
                 grant Ti a lock-X on D
             write(D)
      end;
```

- All locks are released after commit or abort

Two – Phase Lock (2PL)

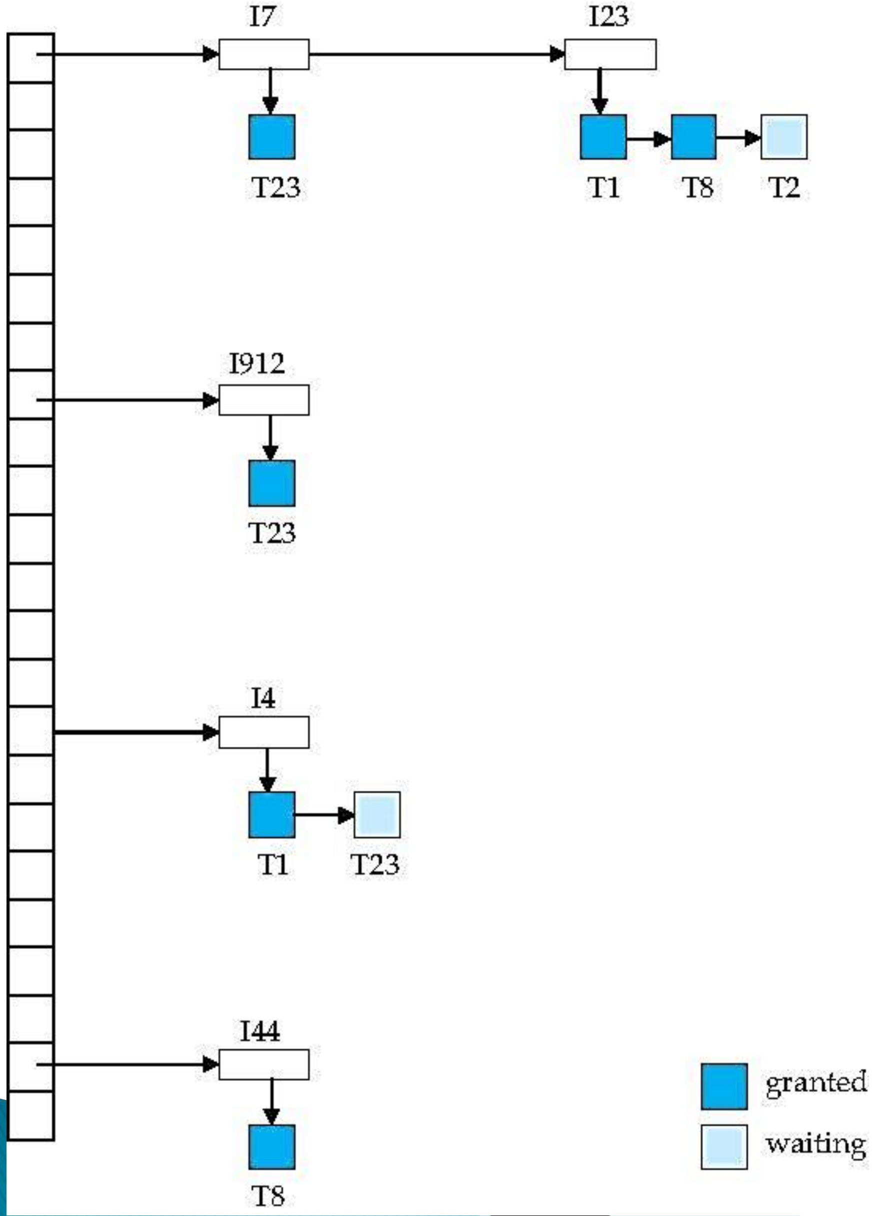
- Strict 2PL
  - Strict 2PL should satisfy basic 2PL
  - All Exclusive Locks should be held until Commit/Abort

- Rigorous 2PL:
  - Rigorous 2PL should satisfy basic 2PL
  - All Exclusive Locks, Shared Locks should be held until Commit/Abort

Implementation of Locking

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Dark blue rectangles indicate granted locks;
- Light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
- Lock manager may keep a list of locks held by each transaction, to implement this efficiently

Timestamp-Based Protocol

- Each transaction is issued a timestamp when it enters the system
  - It is a unique value assigned to every transaction
  - It tells the order in which a transaction has entered the system
    - Transaction: T_i
    - Time stamp: TS(T_i)
- If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j)
- The protocol manages concurrent execution such that the time-stamps determine the serializability order – *Older transactions are executed first*

Timestamp-Based Protocol

- In order to assure such behavior, the protocol maintains for each data A two timestamp values:
- W-timestamp(A)
  - WTS(A) is the largest time-stamp of any transaction that executed write(A) successfully – Last transaction which performed Read successfully
- R-timestamp(A)
  - RTS(A) is the largest time-stamp of any transaction that executed read(A) successfully – Last transaction which performed Write successfully

Timestamp-Based Protocol

- **Time Stamp of Transaction TS(T_i)**

10:00	**10:05**	**10:07**	**(Time of Transaction)**
T1	T2	T3	**Ti**
100	120	134	**TS(T_i)**
Oldest		*Youngest*	

Timestamp-Based Protocol

- **Time stamp of Data Item RTS(A)**

09:00	**09:03**	**09:15**	
T1	T2	T3	**Ti**
10	12	24	**$TS(T_i)$**
R(A)			
	R(A)		
		R(A)	

RTS(A) = 24

Timestamp-Based Protocol

- **Time stamp of Data Item WTS(A)**

09:00	**09:03**	**09:15**	
T1	T2	T3	**Ti**
10	12	24	**$TS(T_i)$**
W(A)	$\boxed{\textbf{W(A)}}$	*W(A)*	

WTS(A) = 12

Timestamp-Based Protocol

- Suppose a transaction T_i issues a Read(A)
- If TS(T_i) <WTS(A), then T_i needs to read a value of A that was already overwritten.
  - Hence, the read operation is rejected, and T_i is rolled back
- If TS(T_i) ≥ WTS(A), then then the read operation is executed, and set RTS(A) = max{RTS(A), TS(T_i)}

Example
TS(T_i) <WTS(A)
900 < 903

09:00	**09:03**
Ti	Tx
	W(A)
	.
R(A)	.
	.
	XXXX

Timestamp-Based Protocol

- Suppose a transaction T_i issues a Read(A)
- If TS(T_i) <WTS(A), then T_i needs to read a value of A that was already overwritten.
  - Hence, the read operation is rejected, and T_i is rolled back
- If TS(T_i) ≥ WTS(A), then then the read operation is executed, and set
  RTS(A) = max{RTS(A), TS(T_i)}

Example
TS(T_i) >WTS(A)
910 > 903

09:10	**09:03**
Ti	Tx
	W(A)
R(A)	

Timestamp-Based Protocol

- Suppose a transaction T_i issues a Write(A)
- If TS(T_i) ≤ WTS(A), then the value of A that T_i is producing was needed previously, and the system assumed that that value would never be produced.
  - Hence, the write operation is rejected, and T_i is rolled back

- If TS(T_i) < WTS(A), then T_i is attempting to write an obsolete value of A
  - Hence, the write operation is rejected, and T_i is rolled back

- Otherwise, the write operation is executed, and set WTS(A) = max{WTS(A), TS(T_i)}

Timestamp-Based Protocol

- Suppose a transaction T_i issues a Write(A)
- If TS(T_i) ≤ RTS(A), then the value of A that T_i is producing was needed previously, and the system assumed that that value would never be produced.
  - Hence, the write operation is rejected, and T_i is rolled back
- If TS(T_i) < WTS(A), then T_i is attempting to write an obsolete value of A
  - Hence, the write operation is rejected, and T_i is rolled back
- Otherwise, the write operation is executed, and set WTS(A) = max{WTS(A), TS(T_i)}

Example
TS(T_i) <RTS(A)
900 < 903

09:00	09:03
Ti	Tx
	R(A)
W(A)	

Timestamp-Based Protocol

- Suppose a transaction T_i issues a Write(A)
- If TS(T_i) ≤ RTS(A), then the value of A that T_i is producing was needed previously, and the system assumed that that value would never be produced.
  - Hence, the write operation is rejected, and T_i is rolled back
- If TS(T_i) < WTS(A), then T_i is attempting to write an obsolete value of A
  - Hence, the write operation is rejected, and T_i is rolled back
- Otherwise, the write operation is executed, and set WTS(A) = max{WTS(A), TS(T_i)}

Example
TS(T_i) <WTS(A)
900 < 903

09:00	**09:03**
Ti	Tx
	W(A)
W(A)	

Timestamp-Based Protocol

- Suppose a transaction T_i issues a Write(A)
- If $TS(T_i) \leq RTS(A)$, then the value of A that T_i is producing was needed previously, and the system assumed that that value would never be produced.
  - Hence, the write operation is rejected, and T_i is rolled back
- If $TS(T_i) < WTS(A)$, then T_i is attempting to write an obsolete value of A
  - Hence, the write operation is rejected, and T_i is rolled back
- Otherwise, the write operation is executed, and set $WTS(A) = \max\{WTS(A), TS(T_i)\}$

Example $TS(T_i) \geq RTS(A)$	Example $TS(T_i) \geq WTS(A)$

Timestamp-Based Protocol - Properties

- It ensures conflict serializability
- It ensures view serializability
- Free from deadlock
- Possibility of dirty read and irrecoverable schedule

Pessimistic concurrency control

- Pessimistic concurrency control (or pessimistic locking) is called "pessimistic" because the system assumes the worst — it assumes that two or more users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are.

- The locks are placed as soon as any piece of the row is accessed, making it impossible for two or more users to update the row at the same time. Depending on the lock mode (shared, exclusive, or update), other users might be able to read the data even though a lock has been placed. For more details on the lock modes, see Lock modes: shared, exclusive, and update.

Optimistic concurrency control

- Optimistic concurrency control (or optimistic locking) assumes that although conflicts are possible, they will be very rare. Instead of locking every record every time that it is used, the system merely looks for indications that two users actually did try to update the same record at the same time. If that evidence is found, then one user's updates are discarded and the user is informed.

Validation-Based Protocol

- Execution of transaction Ti is done in three phases.
  1. Read and execution phase: Transaction Ti writes only to temporary local variables
  2. Validation phase: Transaction Ti performs a "validation test'' to determine if local variables can be written without violating serializability.
  3. Write phase: If Ti is validated, the updates are applied to the database; otherwise, Ti is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
- Assume for simplicity that the validation and write phase occur together, atomically and serially i.e., only one transaction executes validation/write at a time.
- Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation