Module 4:

Concept of transaction, ACID properties, serializability, states of transaction, Concurrency control, Locking techniques, Time stamp-based protocols, Granularity of data items, Deadlock, Failure classifications, storage structure, Recovery & atomicity, Log base recovery, Recovery with concurrent transactions, Database backup & recovery, Remote Backup System, Database security issues.

What is a Database Transaction?

A **Database Transaction** is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise.

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

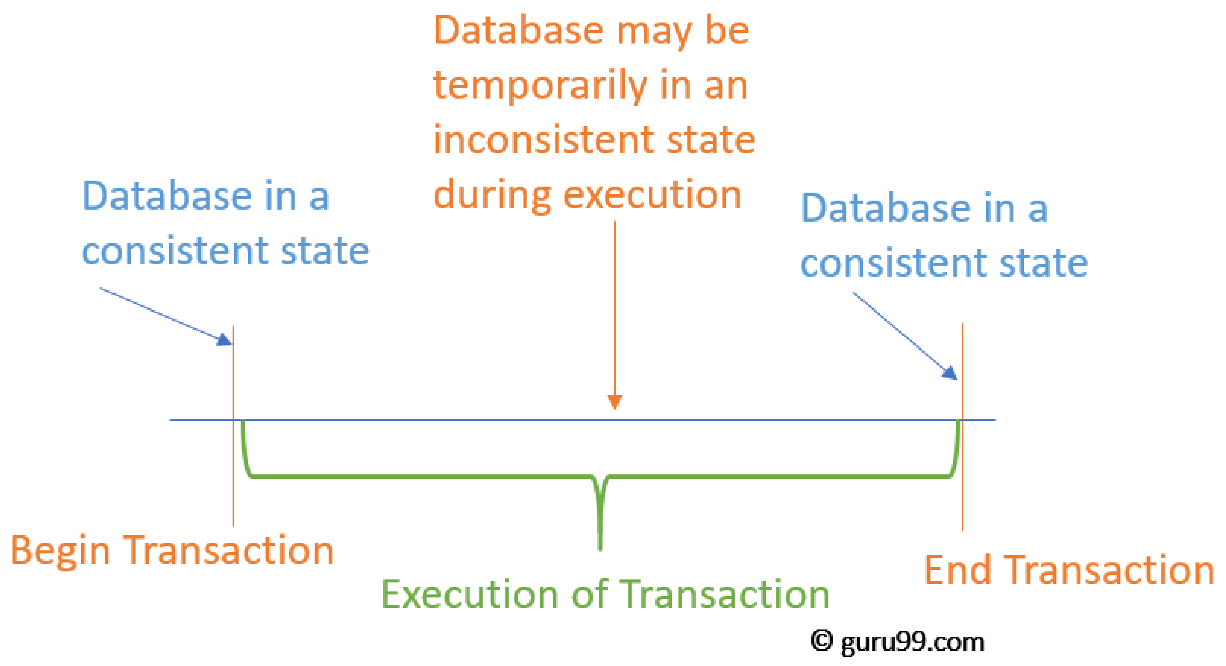
A's Account

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

B's Account

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction in DBMS. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.


Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Database in a consistent state
Begin Transaction
Execution of Transaction
End Transaction
© guru99.com

Facts about Database Transactions

- A transaction is a program unit whose execution may or may not change the contents of a database.
- The transaction concept in DBMS is executed as a single unit.
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- A successful transaction can change the database from one CONSISTENT STATE to another
- DBMS transactions must be atomic, consistent, isolated and durable
- If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

Why do you need concurrency in Transactions?

A database is a shared resource accessed. It is used by many users and processes concurrently. For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.

Not managing concurrent access may create issues like:

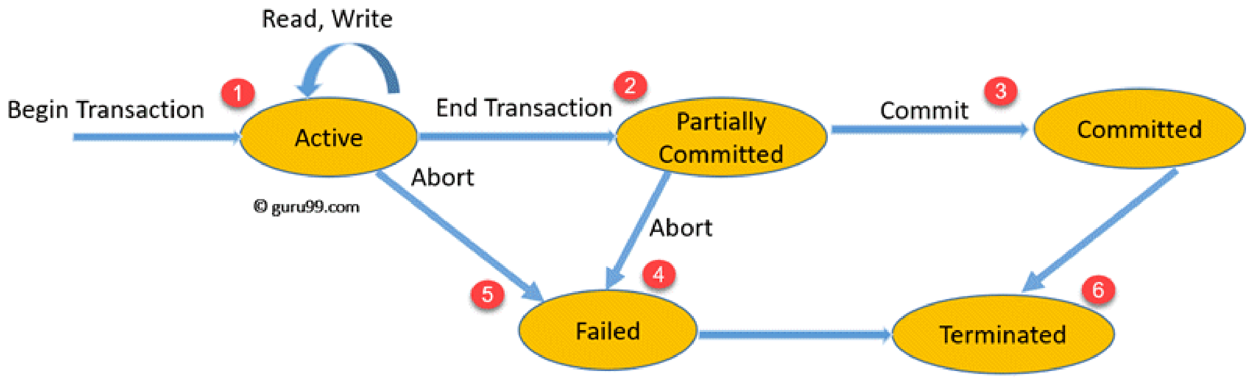
- Hardware failure and system crashes
- Concurrent execution of the same transaction, deadlock, or slow performance

States of Transactions

The various states of a transaction concept in DBMS are listed below:

State	**Transaction types**
Active State	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
Partially Committed	A transaction goes into the partially committed state after the end of a transaction.
Committed State	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
Failed State	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
Terminated State	State of transaction reaches terminated state when certain transactions which are leaving the

system can't be restarted.



State Transition Diagram for a Database Transaction

Let's study a state transition diagram that highlights how a transaction moves between these various states.

1. Once a transaction states execution, it becomes active. It can issue READ or WRITE operation.
2. Once the READ and WRITE operations complete, the transactions becomes partially committed state.
3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a success, the transaction commits and enters into the committed state.
4. If the check is a fail, the transaction goes to the Failed state.
5. If the transaction is aborted while it's in the active state, it goes to the failed state. The transaction should be rolled back to undo the effect of its write operations on the database.
6. The terminated state refers to the transaction leaving the system.

What are ACID Properties?

ACID Properties are used for maintaining the integrity of database during transaction processing. ACID in DBMS stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

- **Atomicity:** A transaction is a single unit of operation. You either execute it entirely or do not execute it at all. There cannot be partial execution.
- **Consistency:** Once the transaction is executed, it should move from one consistent state to another.
- **Isolation:** Transaction should be executed in isolation from other transactions (no Locks). During concurrent transaction execution, intermediate transaction results from simultaneously executed transactions should not be made available to each other. (Level 0,1,2,3)
- **Durability: ·** After successful completion of a transaction, the changes in the database should persist. Even in the case of system failures.

ACID Property in DBMS with example:

Below is an example of ACID property in DBMS:

```
Transaction 1: Begin X = X - 50, Y = Y + 50 END
Transaction 2: Begin X = 1.1 * X, Y = 1.1 * Y END
```

Transaction 1 is transferring $50 from account X to account Y.

Transaction 2 is crediting each account with a 10% interest payment.

If both transactions are submitted together, there is no guarantee that the Transaction 1 will execute before Transaction 2 or vice versa. Irrespective of the order, the result must be as if the transactions take place serially one after the other.

DBMS Schedule

A schedule is a process of combining the multiple transactions into one and executing the operations of these transactions in a predefined order. A schedule can have multiple transactions in it, each transaction comprising of several tasks or operations or instructions. A schedule can also be defined as “a sequence of operations of multiple transactions that appears for execution”. Or we can say that when several transactions are executed concurrently in the database, then the order of execution of these transactions is known as a schedule. A specific sequence of operations of a set of instructions is called a schedule.

Schedule Example

Schedule 1:

Time	**Transaction T1**	**Transaction T2**
t1	Read(A)	
t2	A=A+50	
t3	Write(A)	
t4		Read(A)
t5		A+A+100
t6		Write(A)
t7	Read(B)	
t8	B=B+100	

t9	Write(B)	
t10		Read(B)
t11		B=B+50
t12		Write(B)

Types of Schedule

Schedule is of two types:

- Serial schedule
- Non-serial schedule
-

Serial Schedule

The serial schedule is a type of schedule in which the transactions are executed one after other without interleaving. It is a schedule in which multiple transactions are organized in such a way that one transaction is executed first. When the operations of first transaction complete, then next transaction is executed.

In a serial schedule, when one transaction executes its operation, then no other transaction is allowed to execute its operations.

For example:

Suppose a schedule S with two transactions T1 and T2. If all the instructions of T1 are executed before T2 or all the instructions of T2 are executed before T1, then S is said to be a serial schedule.

Below tables shows the example of a serial schedule, in which the operations of first transaction are executed first before starting another transactionT2:

Time	Transaction T1	Transaction T2
t1	Read(A)	
t2	A=A+50	
t3	Write(A)	
t4	Read(B)	
t5	B=B+100	
t6	Write(B)	
t7		Read(A)
t8		A+A+100
t9		Write(A)
t10		Read(B)
t11		B=B+50
t12		Write(B)

Non-Serial Schedule

The non-serial schedule is a type of schedule where the operations of multiple transactions are interleaved. Unlike the serial schedule, this schedule proceeds without waiting for the execution of the previous transaction to complete. This schedule may give rise to the problem of concurrency. In a non-serial schedule multiple transactions are executed concurrently.

For example:

Time	Transaction T1	Transaction T2
t1	Read(A)	
t2		Read(A)
t3	A=A+50	
t4		A+A+100
t5	Write(A)	
t6		Write(A)

In this schedule S, there are two transaction T1 and T2. If T1 and T2 transactions are executed concurrently, and the operations of T1 and T2 are interleaved. So, this schedule is an example of a non-serial schedule.

Serializability in DBMS

Serializability is the concept in a transaction that helps to identify which non-serial schedule is correct and will maintain the database consistency. It relates to the isolation property of transaction in the database.

Serializability is the concurrency scheme where the execution of concurrent transactions is equivalent to the transactions which execute serially.

Serializable Schedule

A serial schedule is always a serializable schedule because any transaction only starts its execution when another transaction has already completed its execution. However, a non-serial schedule of transactions needs to be checked for Serializability.

Note: If a schedule of concurrent 'n' transactions can be converted into an equivalent serial schedule. Then we can say that the schedule is serializable. And this property is known as serializability.

Testing of Serializability

To test the serializability of a schedule, we can use the serialization graph.
Suppose, a schedule S. For schedule S, construct a graph called as a precedence graph. It has a pair G = (V, E), where E consists of a set of edges, and V consists of a set of vertices. The set of vertices contain all the transactions participating in the S schedule. The set of edges contains all edges Ti ->Tj for which one of the following three conditions satisfy:

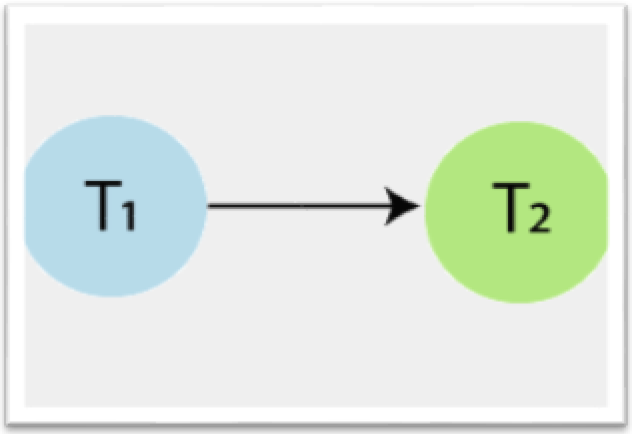
1. Create a node Ti ? Tj if Ti transaction executes write (Q) before Tj transaction executes read (Q).
2. Create a node Ti ? Tj if Ti transaction executes read (Q) before Tj transaction executes write (Q).
3. Create a node Ti ? Tj if Ti transaction executes write (Q) before Tj transaction executes write (Q).

Schedule S:

Time	Transaction T1	Transaction T2
t1	Read(A)	

t2	A=A+50	
t3	Write(A)	
t4		Read(A)
t5		A+A+100
t6		Write(A)

Precedence graph of Schedule S



In above precedence graph of schedule S, contains two vertices T1 and T2, and a single edge T1? T2, because all the instructions of T1 are executed before the first instruction of T2 is executed.

If a precedence graph for any schedule contains a cycle, then that schedule is non-serializable. If the precedence graph has no cycle, then the schedule is serializable.

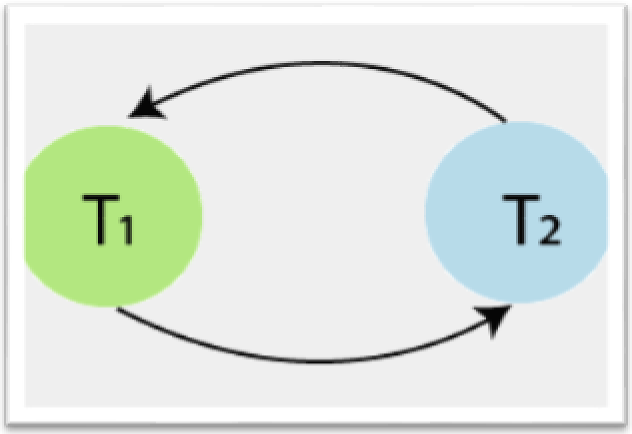
So, schedule S is serializable (i.e., serial schedule) because the precedence graph has no cycle.

Schedule S1:

Time	**Transaction T1**	**Transaction T2**
t1	Read(A)	
t2		Read(A)
t3		Write(A)

t4	A=A+50	
t5	Write(A)	

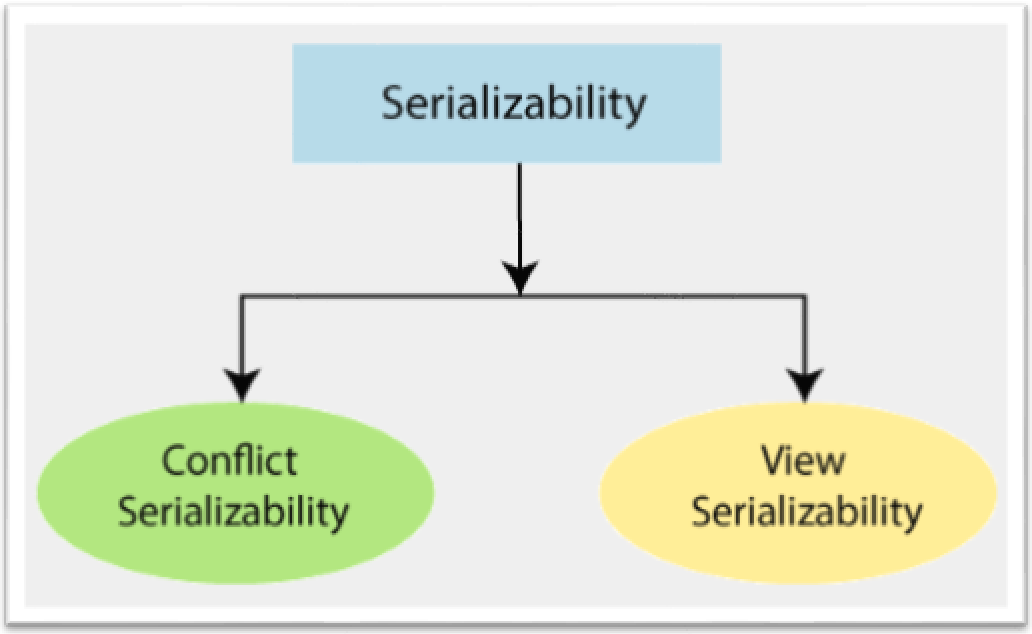
Precedence graph of Schedule S1



In above precedence graph of schedule S1, contains two vertices T1 and T2, and edges T1? T2 and T2? T1. In this Schedule S1, operations of T1 and T2 transaction are present in an interleaved manner.
The precedence graph contains a cycle, that's why schedule S1 is non-serializable.

Types of Serializability

1. Conflict Serializability
2. View Serializability



Conflict Serializability in DBMS

A schedule is said to be conflict serializable if it can transform into a serial schedule after swapping of non-conflicting operations. It is a type of serializability that can be used to check whether the non-serial schedule is conflict serializable or not.

Conflicting operations

The two operations are called conflicting operations, if all the following three conditions are satisfied:

- Both the operation belongs to separate transactions.
- Both works on the same data item.
- At least one of them contains one write operation.

Time	**Transaction T1**	**Transaction T2**	**Transaction T3**
t1	Read(X)		
t2			Read(Y)
t3			Read(X)

t4		Read(Y)	
t5		Read(Z)	
t6			Write(Y)
t7		Write(Z)	
t8	Read(Z)		
t9	Write(X)		
t10	Write(Z)		

Note: Conflict pairs for the same data item are:
Read-Write
Write-Write
Write-Read

Conflict Equivalent Schedule

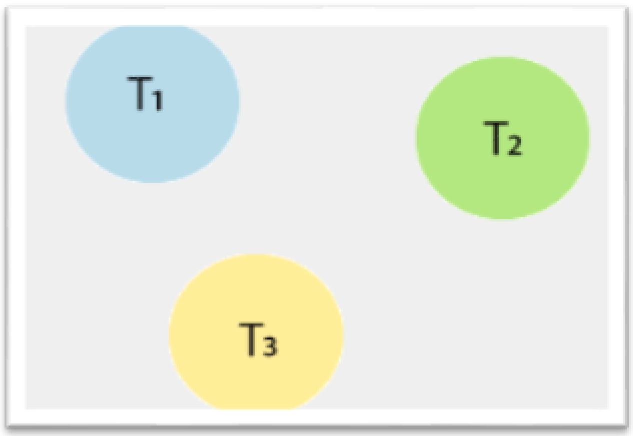
Two schedules are called as a conflict equivalent schedule if one schedule can be transformed into another schedule by swapping non-conflicting operations.

Example of conflict serializability:

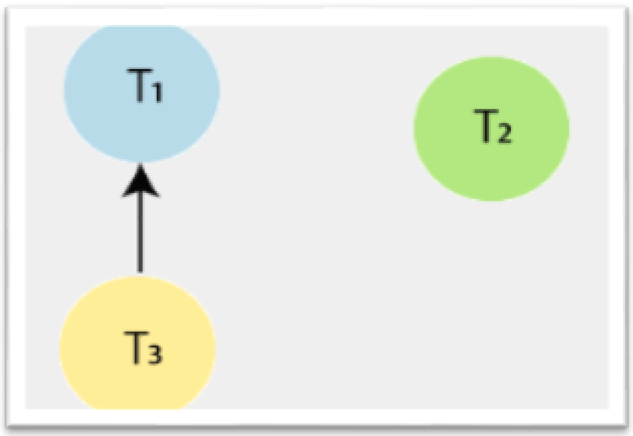
Schedule S2 (Non-Serial Schedule):

Precedence graph for schedule S2:

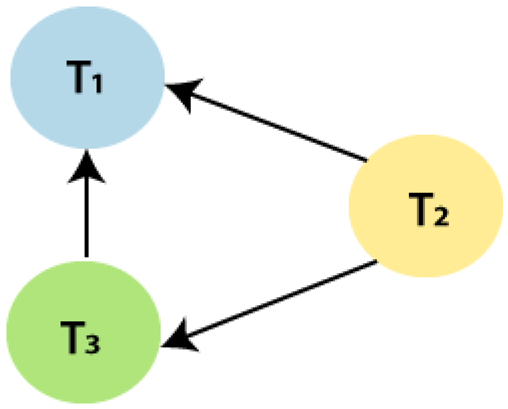
In the above schedule, there are three transactions: T1, T2, and T3. So, the precedence graph contains three vertices.



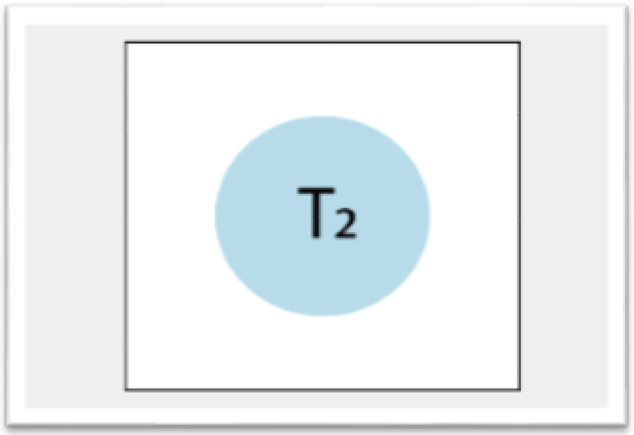
To draw the edges between these nodes or vertices, follow the below steps:
Step1: At time t1, there is no conflicting operation for **read(X)** of Transaction T1.
Step2: At time t2, there is no conflicting operation for **read(Y)** of Transaction T3.
Step3: At time t3, there exists a conflicting operation **Write(X)** in transaction T1 for **read(X)** of Transaction T3. So, draw an edge from T3?T1.



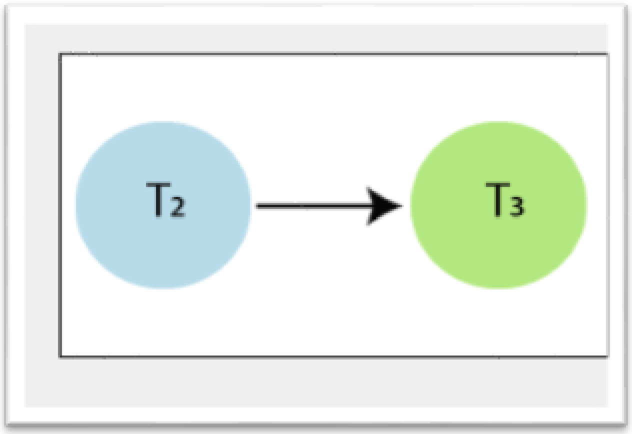
Step4: At time t4, there exists a conflicting operation **Write(Y)** in transaction T3 for **read(Y)** of Transaction T2. So, draw an edge from T2->T3.
Step5: At time t5, there exists a conflicting operation **Write (Z)** in transaction T1 for **read (Z)** of Transaction T2. So, draw an edge from T2->T1.



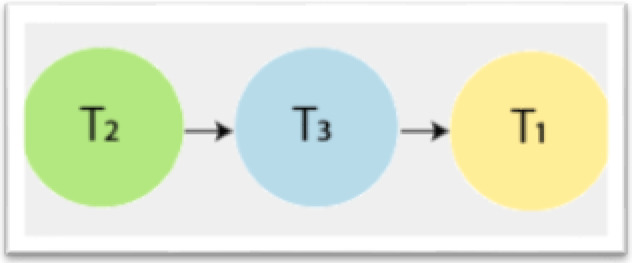
Step6: At time t6, there is no conflicting operation for **Write(Y)** of Transaction T3.
Step7: At time t7, there exists a conflicting operation **Write (Z)** in transaction T1 for **Write (Z)** of Transaction T2. So, draw an edge from T2?T1, but it is already drawn.
After all the steps, the precedence graph will be ready, and it does not contain any cycle or loop, so the above schedule S2 is conflict serializable. And it is equivalent to a serial schedule. Above schedule S2 is transformed into the serial schedule by using the following steps:
Step1: Check the vertex in the precedence graph where **indegree=0.** So, take the vertex T2 from the graph and remove it from the graph.



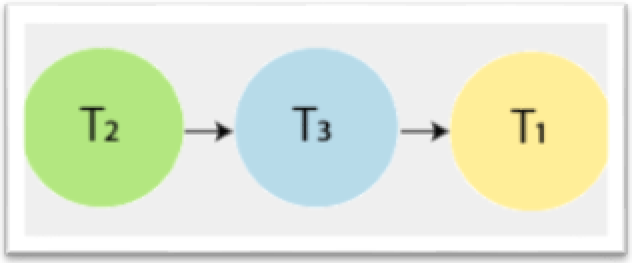
Step 2: Again check the vertex in the left precedence graph where **indegree=0.** So, take the vertex T3 from the graph and remove it from the graph. And draw the edge from T2?T3.



Step3: And at last, take the vertex T1 and connect with T3.



Precedence graph equivalent to schedule S2



Schedule S2 (Serial Schedule):

Time	Transaction T1	Transaction T2	Transaction T3
t1		Read(Y)	
t2		Read(Z)	

t3		Write(Z)	
t4			Read(Y)
t5			Read(X)
t6			Write(Y)
t7	Read(X)		
t8	Read(Z)		
t9	Write(X)		
t10	Write(Z)		

Schedule S3 (Non-Serial Schedule):

Time	Transaction T1	Transaction T2
t1	Read(X)	
t2		Read(X)
t3		Read (Y)
t4		Write(Y)

t5	Read(Y)	
t6	Write(X)	

To convert this schedule into a serial schedule, swap the non- conflicting operations of T1 and T2.

Time	Transaction T1	Transaction T2
t1		Read(X)
t2		Read (Y)
t3		Write(Y)
t4	Read(X)	
t5	Read(Y)	
t6	Write(X)	

Then, finally get a serial schedule after swapping all the non-conflicting operations, so this schedule is **conflict serializable.**

View Serializability in DBMS

It is a type of serializability that can be used to check whether the given schedule is view serializable or not. A schedule called as a view serializable if it is view equivalent to a serial schedule.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if both satisfy the following conditions:

1. Initial read
An initial read of the data item in both the schedule must be same. For example, lets two schedule S1 and S2. If transaction T1 reads the data item X in schedule S1, then in schedule S2 transaction T1 also reads X.
Schedule S1:

Time	Transaction T1	Transaction T2
t1	Read(X)	
t2		Write(X)

Schedule S2:

Time	Transaction T1	Transaction T2
t1		Write(X)
t2	Read(X)	

Above two schedules, S1 and S2 are view equivalent, because initial read instruction in S1 is done by T1 transaction and in schedule S2 is also done by transaction T2.

2. Updated Read
In schedule S1, if the transaction T_i is reading the data item A which is updated by transaction T_j, then in schedule S2 also, T_i should read data item A which is updated by T_j.

Schedule S1:

Time	Transaction T1	Transaction T2
t1	Write(X)	
t2		Read(X)

Schedule S2:

Time	Transaction T1	Transaction T2
t1	Write(X)	
t2		Read(X)

Above two schedules S1 and S2 are view equivalent because in schedule S1 transaction T2 reads the data item A which is updated by T1 and in schedule S2 T2 also reads the data item A which is updated by T1.

3. Final write

The final write operation on each data item in both the schedule must be same. In a schedule S1, if a transaction T1 updates data item A at last then in schedule S2, final writes operations should also be done by T1 transaction.

Schedule S1:

Time	Transaction T1	Transaction T2	Transaction T3
t1			Write(X)
t2		Read(X)	
t3	Write(X)		

Schedule S2:

Time	Transaction T1	Transaction T2	Transaction T3
t1			Write(X)
t2		Read(X)	
t3	Write(X)		

Above two schedules, S1 and S2 are view equivalent because final write operation in schedule S1 is done by T1 and in S2, T1 also does the final write operation.

View Serializable

A schedule is said to be a view serializable if that schedule is view equivalent to a serial schedule.

View Serializable example

Schedule S1 (Non-Serial Schedule):

Time	Transaction T1	Transaction T2
t1	Read(X)	
t2	Write(X)	
t3		Read(X)
t4		Write(X)
t5	Read(Y)	
t6	Write(Y)	

t7		Read(Y)
t8		Write(Y)

Schedule S2 (Serial Schedule):

Time	Transaction T1	Transaction T2
t1	Read(X)	
t2	Write(X)	
t3	Read(Y)	
t4	Write(Y)	
t5		Read(X)
t6		Write(X)
t7		Read(Y)
t8		Write(Y)

Note: S2 is the serial schedule of S1. If we can prove that both the schedule is view equivalent, then we can say that S1 schedule is a view serializable schedule.
Now, check the three conditions of view serializability for this example:

1. Initial Read
In S1 schedule, T1 transaction first reads the data item X. In Schedule S2 also transaction T1 first reads the data item X.
Now, check for Y. In schedule S1, T1 transaction first reads the data item Y. In schedule S2 also the first read operation on data item Y is performed by T1.
We checked for both data items X and Y, and the **initial read** condition is satisfied in schedule S1 & S2.

2. Updated Read
In Schedule S1, transaction T2 reads the value of X, which is written by transaction T1. In Schedule S2, the same transaction T2 reads the data item X after T1 updates it.
Now check for Y. In Schedule S1, transaction T2 reads the value of Y, which is written by T1. In S2, the same transaction T2 reads the value of data item Y after T1 writes it.
The **update read condition** is also satisfied for both the schedules S1 and S2.

3. Final Write
In schedule S1, the **final write operation** on data item X is done by transaction T2. In schedule S2 also transaction T2 performs the final write operation on X.
Now, check for data item Y. In schedule S1, the final write operation on Y is done by T2 transaction. In schedule S2, a final write operation on Y is done by T2.
We checked for both data items X and Y, and the **final write** condition is also satisfied for both the schedule S1 & S2.

Conclusion: Hence, all the three conditions are satisfied in this example, which means Schedule S1 and S2 are view equivalent. Also, it is proved that schedule S2 is the serial schedule of S1. Thus we can say that the S1 schedule is a view serializable schedule.

DBMS Concurrency Control

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

But before knowing about concurrency control, we should know about concurrent execution.

Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

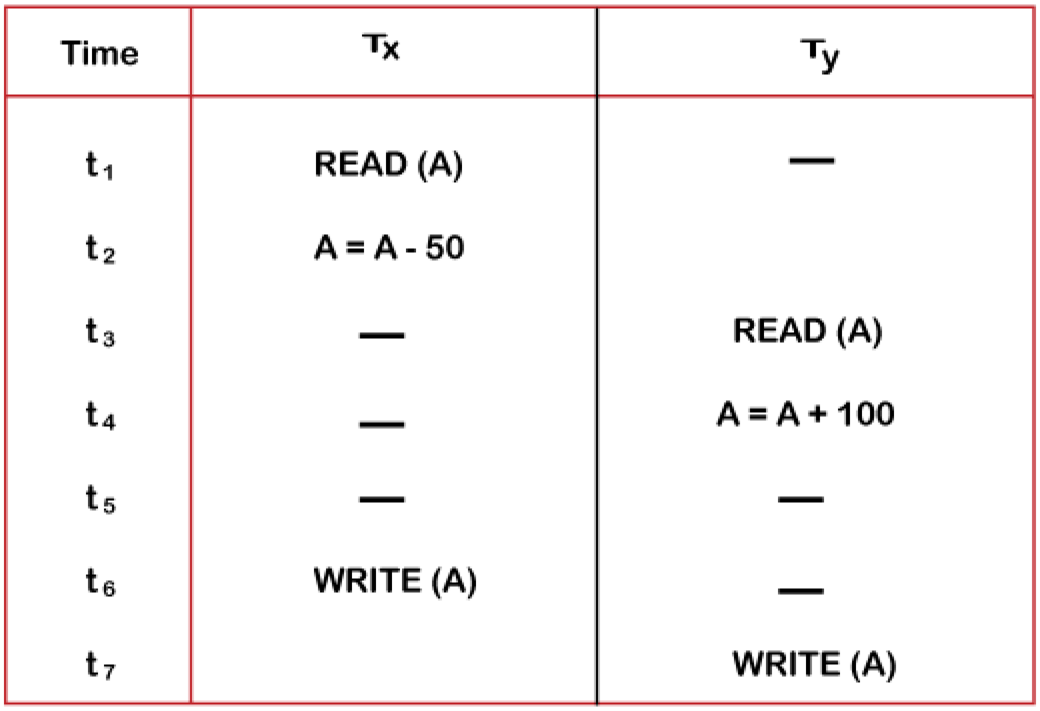
In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent*.

For example:

Consider the below diagram where two transactions T_X and T_Y, are performed on the same account A where the balance of account A is $300.



Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_X deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction T_X writes the value of account A that will be updated as $250 only, as T_Y didn't update the value yet.
- Similarly, at time t7, transaction T_Y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_X is lost, i.e., $250 is lost.

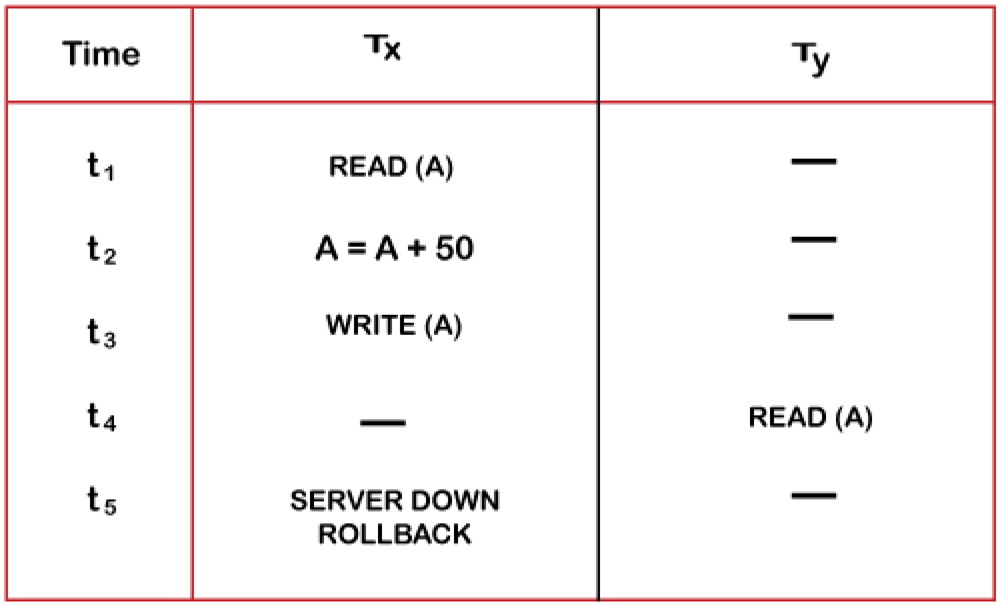
Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict)

The dirty read problem occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated* *database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

For example:

Consider two transactions T_X and T_Y in the below diagram performing read/write operations on account A where the available balance in account A is $300:



Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

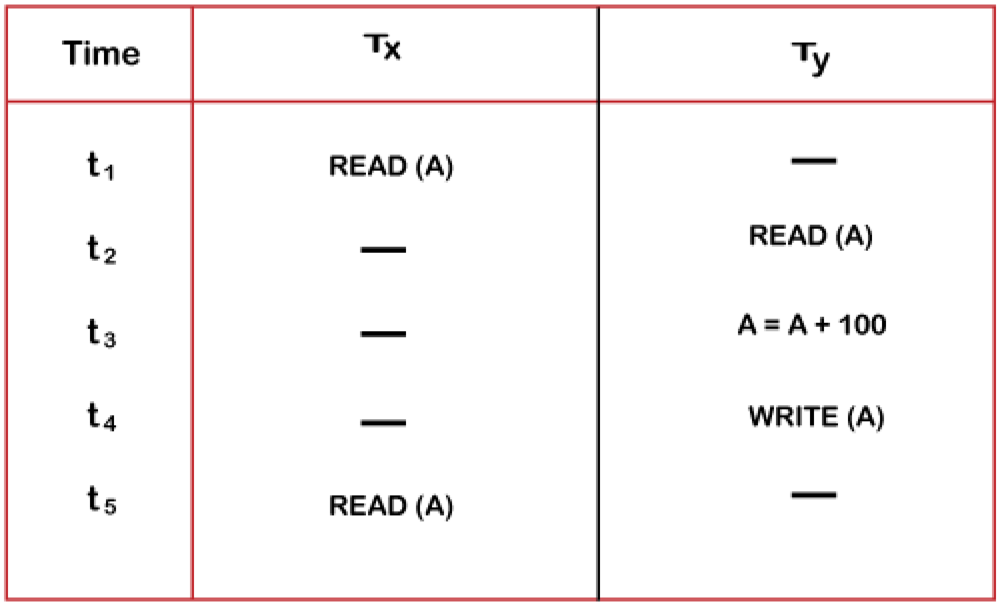
- At time t1, transaction T_X reads the value of account A, i.e., $300.
- At time t2, transaction T_X adds $50 to account A that becomes $350.
- At time t3, transaction T_X writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_Y reads account A that will be read as $350.
- Then at time t5, transaction T_X rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_Y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (R-W Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, T_X and T_Y, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:



Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction T_X reads the value from account A, i.e., $300.
- At time t2, transaction T_Y reads the value from account A, i.e., $300.
- At time t3, transaction T_Y updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction T_Y writes the updated value, i.e., $400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as $400.
- It means that within the same transaction T_X, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction T_Y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

It is the process of managing simultaneous execution of transaction in a shared database.

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Two Phase Locking Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

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 those 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.

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.

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.

- Shared lock is also called read lock, used for reading data items only.
- Shared locks support read integrity. They ensure that a record is not in process of being updated during a read-only request.
- Shared locks can also be used to prevent any kind of updates of record.
- It is denoted by Lock-S.
- S-lock is requested using Lock-S instruction.

For **example**, consider a case where initially A=100 and there are two transactions which are reading A. If one of transaction wants to update A, in that case other transaction would be reading wrong value. However, Shared lock prevents it from updating until it has finished reading.

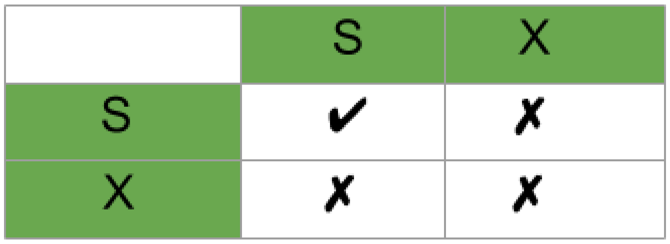
2. Exclusive Lock (X):

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.

- With the Exclusive Lock, a data item can be read as well as written. Also called write lock.
- An exclusive lock prevents any other locker from obtaining any sort of a lock on the object.
- They can be owned by only one transaction at a time.
- It is denoted as Lock-X.
- X-lock is requested using Lock-X instruction.

For **example**, consider a case where initially A=100 when a transaction needs to deduct 50 from A. We can allow this transaction by placing X lock on it. Therefore, when the any other transaction wants to read or write, exclusive lock prevent it.

Lock Compatibility Matrix :



	S	X
S	✔	✗
X	✗	✗

Compatibility matrix for locks

- If the transaction T1 is holding a shared lock in data item A, then the control manager can grant the shared lock to transaction T2 as compatibility is TRUE, but it cannot grant the exclusive lock as compatibility is FALSE.
- In simple words if transaction T1 is reading a data item A, then same data item A can be read by another transaction T2 but cannot be written by another transaction.
- Similarly if an exclusive lock (i.e. lock for read and write operations) is hold on the data item in some transaction then no other transaction can acquire Shared or Exclusive lock as the compatibility function denoted FALSE.

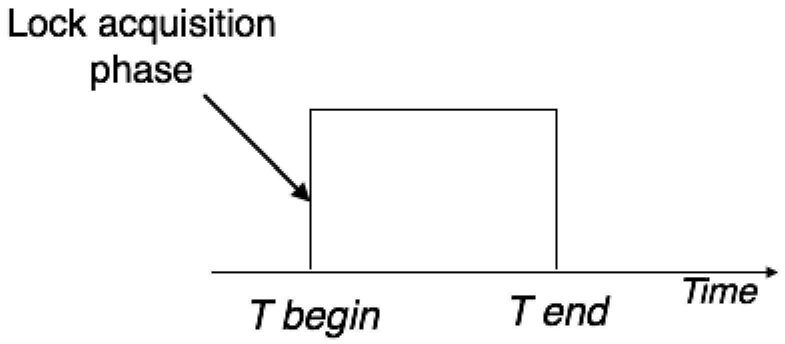
For example, when a transaction needs to update the account balance of a person. You can allow this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevents this operation.

3. Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the 'write' operation.

4. Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.



Difference between Shared Lock and Exclusive Lock :

S.No.	Shared Lock	Exclusive Lock
1.	Lock mode is read only operation.	Lock mode is read as well as write operation.
2.	Shared lock can be placed on objects that do not have an exclusive lock already placed on them.	Exclusive lock can only be placed on objects that do no have any other kind of lock.
3.	Prevents others from updating the data.	Prevents others from reading or updating the data.
4.	Issued when transaction wants to read item that do not have an exclusive lock.	Issued when transaction wants to update unlocked item.
5.	Any number of transaction can hold shared lock on an item.	Exclusive lock can be hold by only one transaction.
6.	S-lock is requested using lock-S instruction.	X-lock is requested using lock-X instruction.

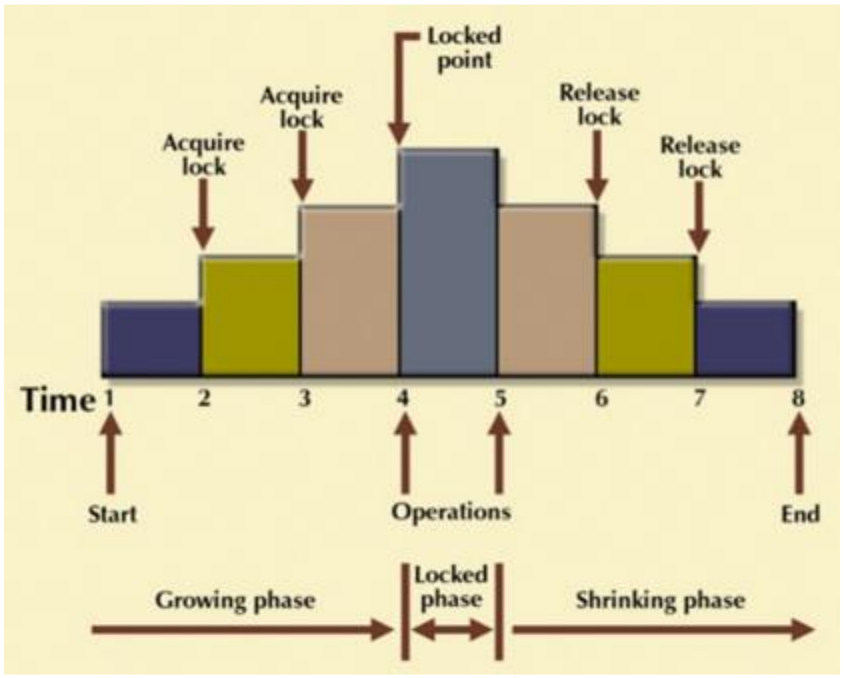
Two Phase Locking Protocol (2PL)

There are two types of Locks available **Shared S(a)** and **Exclusive X(a)**. Implementing this lock system without any restrictions gives us the Simple Lock-based protocol (or *Binary Locking*), but it has its own disadvantages, they do **not guarantee Serializability**. Schedules may follow the preceding rules but a non-serializable schedule may result.
To guarantee serializability, we must follow some additional protocol *concerning the positioning of locking and unlocking operations* in every transaction. This is where the concept of Two-Phase Locking(2-PL) comes into the picture, 2-PL ensures serializability.

Two Phase Locking Protocol also known as 2PL protocol 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.



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

  If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

  Upgrade / Downgrade locks : A transaction that holds a lock on an item **A** is allowed under certain condition to change the lock state from one state to another.
  Upgrade: A S(A) can be upgraded to X(A) if T_i is the only transaction holding the S-lock on element A.
  Downgrade: We may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions.

  Note – If **lock conversion** is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

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

Let’s see a transaction implementing 2-PL.

	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	…….	……

This is just a skeleton transaction that shows how unlocking and locking work with 2-PL. Note for:

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

What is **LOCK POINT?**

The Point at which the growing phase ends, i.e., when a transaction takes the final lock it needs to carry on its work.

That is 2-PL ensures serializability, but there are still some drawbacks of 2-PL:

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation are possible.

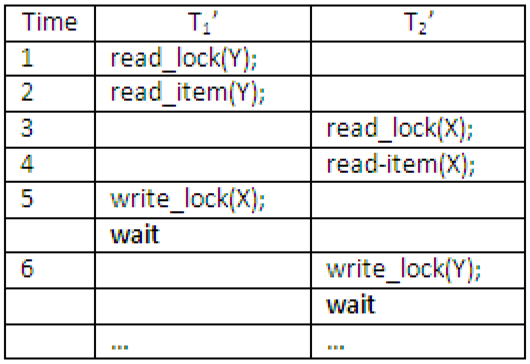
Problem With Simple Locking…

Consider the Partial Schedule:

	T_1	**T_2**
1	lock-X(B)	
2	read(B)	
3	B:=B-50	
4	write(B)	
5		lock-S(A)
6		read(A)
7		lock-S(B)
8	lock-X(A)	
9	……	……

Deadlock – consider the above execution phase. Now, **T_1** holds an Exclusive lock over B, and **T_2** holds a Shared lock over A. Consider Statement 7, **T_2** requests for lock on B, while in Statement 8 **T_1** requests lock on A. This as you may notice imposes a **Deadlock** as none can proceed with their execution.

`Example 2`



Time	T_1'	T_2'
1	read_lock(Y);	
2	read_item(Y);	
3		read_lock(X);
4		read-item(X);
5	write_lock(X);	
	wait	
6		write_lock(Y);
		wait
	…	…

- Drawing the precedence graph, you may detect the loop. So Deadlock is also possible in 2-PL.

- Two-phase locking may also limit the amount of concurrency that occurs in a schedule because a Transaction may not be able to release an item after it has used it. This may be because of the protocols and other restrictions we may put on the schedule to ensure serializability, deadlock freedom, and other factors. This is the price we have to pay to ensure serializability and other factors, hence it can be considered as a bargain between concurrency and maintaining the ACID properties.

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. This may be avoided if the concurrency control manager is properly designed.

Starvation or Livelock is the situation when a transaction has to wait for a indefinite period of time to acquire a lock.

Reasons of Starvation –

- If waiting scheme for locked items is unfair. (priority queue)
- Victim selection.(same transaction is selected as a victim repeatedly)
- Resource leak.
- Via denial-of-service attack.

Starvation can be best explained with the help of an example – Suppose there are 3 transactions namely T1, T2, and T3 in a database that are trying to acquire a lock on data item ‘ I ‘. Now, suppose the scheduler grants the lock to T1(maybe due to some priority), and the other two transactions are waiting for the lock. As soon as the execution of T1 is over, another transaction T4 also come over and request unlock on data item I. Now, this time the scheduler grants lock to T4, and T2, T3 has to wait again. In this way if new transactions keep on requesting the lock, T2 and T3 may have to wait for an indefinite period of time, which leads to **Starvation**.

What are the solutions to starvation –

1. **Increasing Priority –**
   Starvation occurs when a transaction has to wait for an indefinite time, In this situation, we can increase the priority of that particular transaction/s. But the drawback with this solution is that it may happen that the other transaction may have to wait longer until the highest priority transaction comes and proceeds.
2. **Modification in Victim Selection algorithm –**
   If a transaction has been a victim of repeated selections, then the

algorithm can be modified by lowering its priority over other transactions.

3. **First Come First Serve approach –**
   A fair scheduling approach i.e FCFS can be adopted, In which the transaction can acquire a lock on an item in the order, in which the requested the lock.
4. **Wait die and wound wait scheme –**
   These are the schemes that use the timestamp ordering mechanism of transaction.

Deadlock in DBMS

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.

For example: In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.

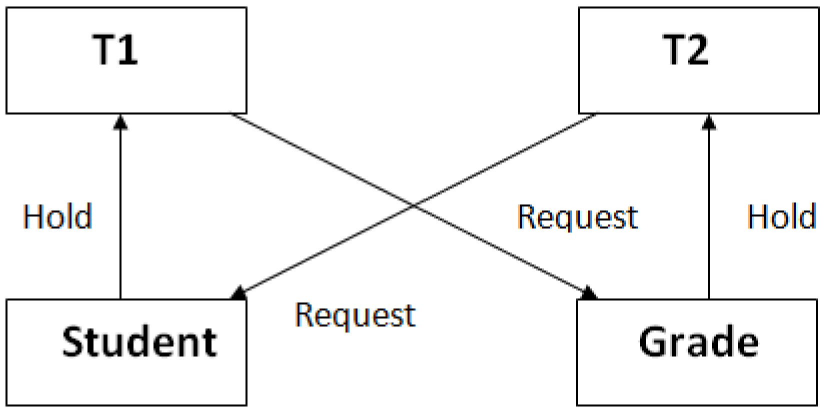


Figure: Deadlock in DBMS

Deadlock Avoidance

- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection

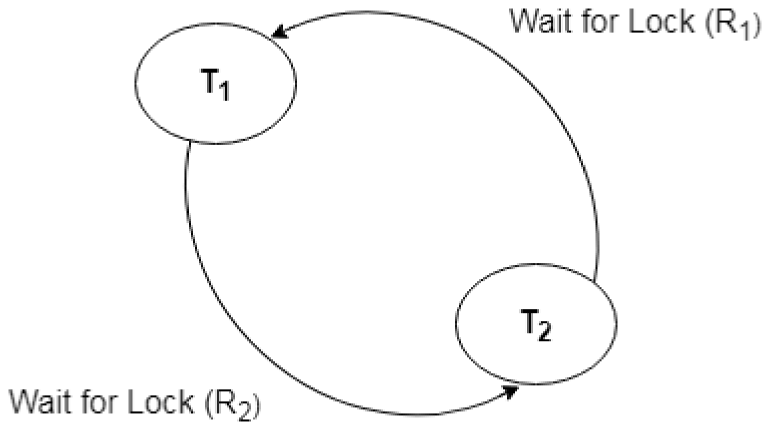
In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Wait for Graph

- This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.

- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



Deadlock Prevention

- Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.
- The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

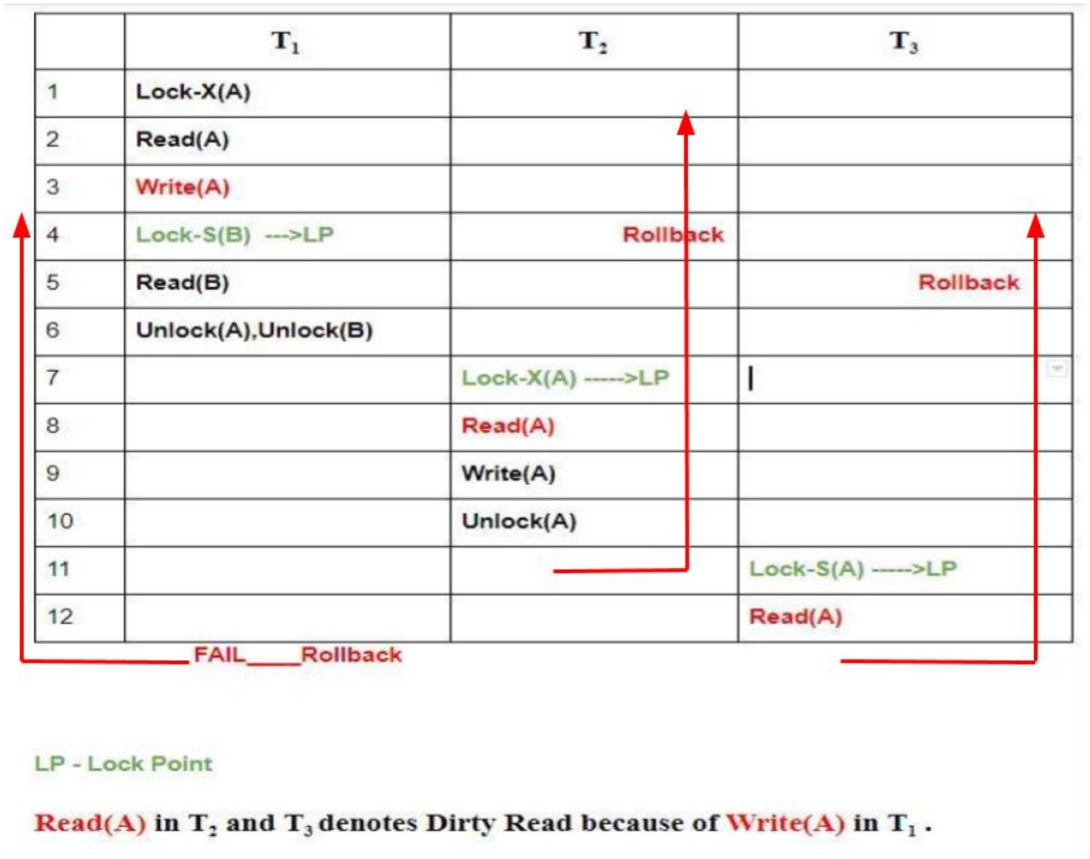
1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.
2. Check if TS(T_i) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

Cascading Rollbacks in 2-PL –

Let’s see the following Schedule:



	T_1	T_2	T_3
1	Lock-X(A)		
2	Read(A)		
3	Write(A)		
4	Lock-S(B) --->LP	Rollback	
5	Read(B)		Rollback
6	Unlock(A),Unlock(B)		
7		Lock-X(A) ----->LP	
8		Read(A)	
9		Write(A)	
10		Unlock(A)	
11			Lock-S(A) ----->LP
12			Read(A)

FAIL____Rollback

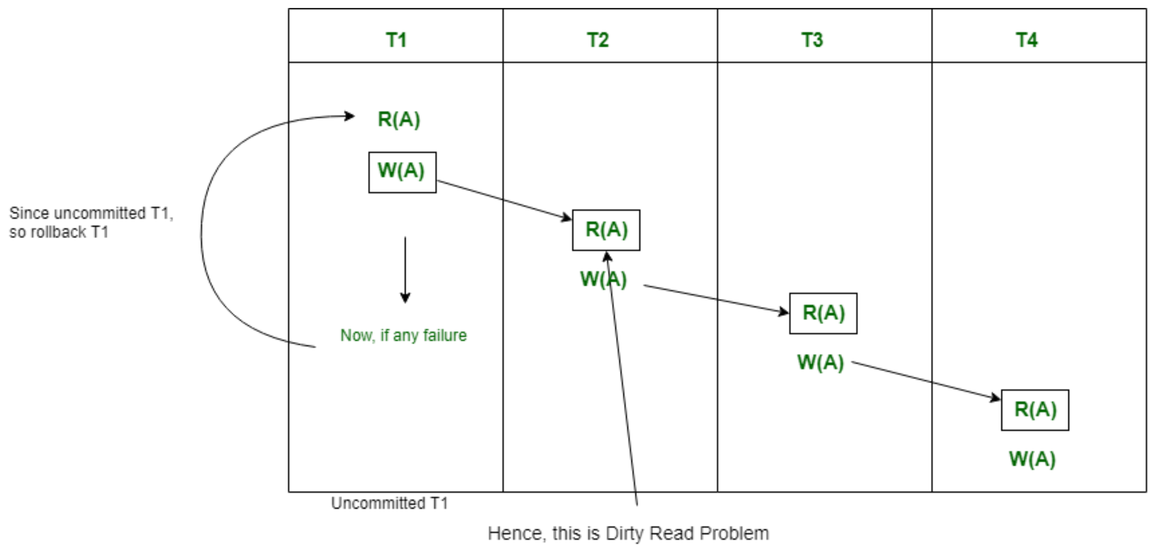
LP - Lock Point

Read(A) in T_2 and T_3 denotes Dirty Read because of Write(A) in T_1 .

- Take a moment 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.**

- If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Rollback or Cascading Abort or Cascading Schedule. It simply leads to the wastage of CPU time.
  These Cascading Rollbacks occur because of **Dirty Read problems**.

- For example, transaction T1 writes uncommitted x that is read by Transaction T2. Transaction T2 writes uncommitted x that is read by Transaction T3. Suppose at this point T1 fails. T1 must be rolled back, since T2 is dependent on T1, T2 must be rolled back, and since T3 is dependent on T2, T3 must be rolled back.



Because of T1 rollback, all T2, T3, and T4 should also be rollback (Cascading dirty read problem).

- This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.