UNIT IV
Transaction Management

What if System Fails?
What if more than one user is concurrently updating the same data?

- A Transaction is a collection of operations that performs a single logical function in a database application.
  - Each Transaction is a unit of both atomicity and consistency. Thus, we require that transaction do not violate any database consistency constraints.
  - For example, you are transferring money from your bank account to your friend's account, the set of operations would be like this:
    1. Read your account balance
    2. Deduct the amount from your balance
    3. Write the remaining balance to your account
    4. Read your friend's account balance
    5. Add the amount to his account balance
    6. Write the new updated balance to his account
- This whole set of operations can be called a transaction. Although here we have seen you read, write, and update operations in the above example but the transaction can have operations like read, write, insert, update, delete.
- **In DBMS, we write the above 6 steps transaction like this:**
- Lets say your account is A and your friend's account is B, you are transferring 1000 from A to B, the steps of the transaction are:

  1. R(A);
  2. A = A - 1000;
  3. W(A);
  4. R(B);
  5. B = B + 1000;
  6. W(B);

- In the above transaction R refers to the Read operation and W refers to the write operation.
- **Transaction failure in between the operations**
  - Now that we understand what is transaction, we should understand what are the problems associated with it.
  - Transaction can fail before finishing the all the operations in the set. This can happen due to power failure, system crash etc.
  - This is a serious problem that can leave database in an inconsistent state. Assume that transaction fail after third operation (see the example above) then the amount would be deducted from your account, but your friend will not receive it.

To solve this problem, we have the following two operations

Commit: If all the operations in a transaction are completed successfully then commit those changes to the database permanently.

– **Rollback**: If any of the operation fails then rollback all the changes done by previous operations.

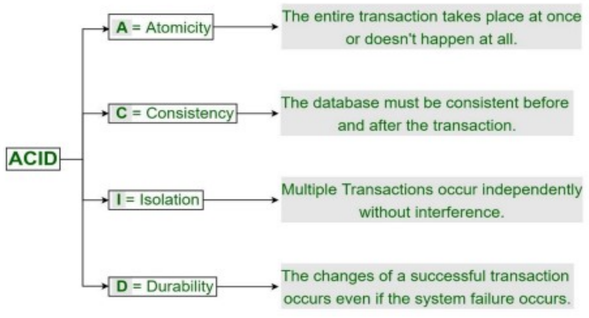
ACID Properties

- Even though these operations can help us avoiding several issues that may arise during transaction but they are not sufficient when two transactions are running concurrently.
- To handle those problems we need to understand database ACID properties.

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ACID properties.

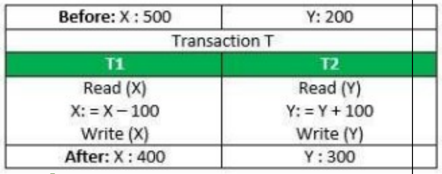
ACID Properties in DBMS


ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

Atomicity

- By this, we mean that either the entire transaction takes place at once or doesn't happen at all.
- There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

Abort: If a transaction aborts, changes made to database are not visible.
Commit: If a transaction commits, changes made are visible.
Atomicity is also known as the 'All or nothing rule'.
Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.



Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

- If the transaction fails after completion of T1 but before completion of T2.(say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y.
- This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

- This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, The total amount before and after the transaction must be maintained.
  - Total before T occurs = 500 + 200 = 700.
  - Total after T occurs = 400 + 300 = 700.
- Therefore, database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete.
- One of the methods to achieve consistency of DB is to define primary and foreign keys. These keys will not restrict unwanted data insert/ delete/update, by checking the integrity of data in DB. Suppose the transaction was to delete a department for which employees are still working. Then the system will not allow deleting the department, unless all its employees are deleted from the system. This is because foreign key is defined on employee table for its department.
- Another way is by maintaining the log for each transaction. It will make sure if there is any failure in middle of any transaction, the data will be recovered by seeing the log.

Isolation

- This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference.
- Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.
- This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
- Let X= 500, Y = 500.
- Consider two transactions T and T".

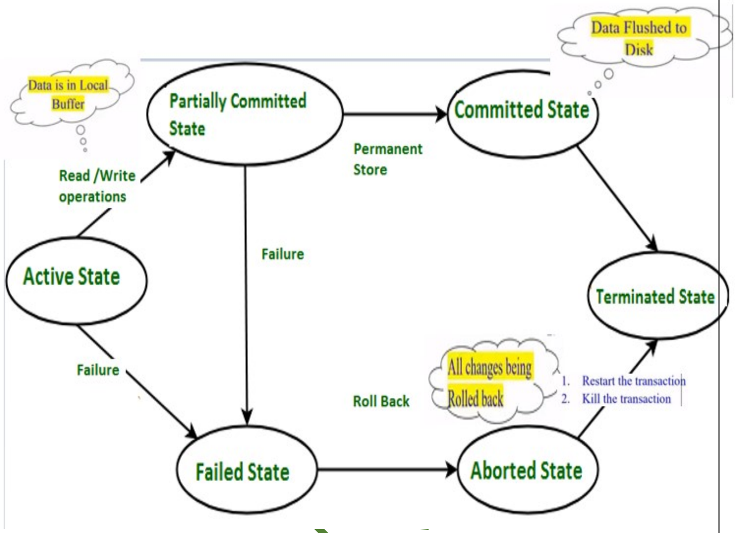
Durability

- This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.
- These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.
- In ATM withdrawal, if the system failure happens after debit or credit operation, the system should be strong enough to update DB with his new balance, after system recovers. It should keep log of each transaction and its failure. So when the system recovers, it should be able to know when a system has failed and if there is any pending transaction, then it should be updated to DB.

The ACID properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored

Transaction States in DBMS

- States through which a transaction goes during its lifetime.
- These are the states which tell about the current state of the Transaction and also tell how we will further do the processing in the transactions.
- These states govern the rules which decide the fate of the transaction whether it will commit or abort.



These are different types of Transaction States :

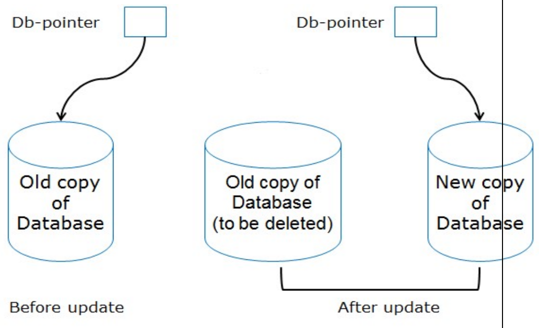
- ➢ **Active State:**
  - ➢ When the instructions of the transaction are running then the transaction is in active state. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".
- ➢ **Partially Committed:**
  - ➢ After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Database then the state will change to "committed state" and in case of failure it will go to the "failed state".
- ➢ **Failed State:**
  - ➢ When any instruction of the transaction fails, it goes to the "failed state" or if failure occurs in making a permanent change of data on Data Base.
- ➢ **Aborted State:**
  - ➢ After having any type of failure the transaction goes from "failed state" to "aborted state" and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.
- ➢ **Committed State:**
  - ➢ It is the state when the changes are made permanent on the Data Base and the transaction is complete and therefore terminated in the "terminated state".
- ➢ **Terminated State:**
  - ➢ If there isn't any roll-back or the transaction comes from the "committed state", then the system is consistent and ready for new transaction and the old transaction is terminated.

Implementation of Atomicity and Durability

- The recovery-management component of a database system can support atomicity and durability by a variety of schemes.

Using shadow copy:

- In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched.
- If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
- This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk; it points to the current copy of the database.



Shadow-copy technique for atomicity and durability

How the technique handles transaction failures:

- If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
- We can abort the transaction by just deleting the new copy of the database.

- Once the transaction has been committed, all the updates that it performed are in the database pointed to by db pointer.
- Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

How the technique handles system failures:

- Suppose that the system fails at any time before the updated db-pointer is written to disk. Then, when the system restarts, it will read db-pointer and will thus see the original contents of the database, and none of the effects of the transaction will be visible on the database.
- Next, suppose that the system fails after db-pointer has been updated on disk. Before the pointer is updated, all updated pages of the new copy of the database were written to disk.
- Again, we assume that, once a file is written to disk, its contents will not be damaged even if there is a system failure. Therefore, when the system restarts, it will read db-pointer and will thus see the contents of the database after all the updates performed by the transaction.

Note: Unfortunately, this implementation is extremely inefficient in the context of large databases, since executing a single transaction requires copying the entire database.

Concurrency Control

- Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.
- A Schedule is a collection of many transactions which is implemented as a unit. depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:
  - Serial: The transactions are executed one after another, in a non-primtive manner.
  - Concurrent: The transactions are executed in a primitive, time shared method.
- <u>Concurrent execution:</u> In the transaction process, a system usually allows executing more than one transaction simultaneously. This process is called a concurrent execution.

Serial Schedule	

T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;
	T2
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;

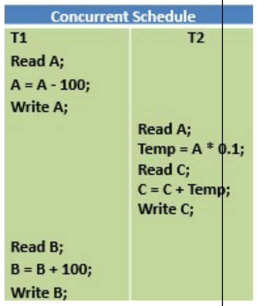
Advantages of concurrent execution of a transaction

- Decrease waiting time or turnaround time.
- Improve response time
- Increased throughput or resource utilization.

- In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time.
- However, a serial schedule is inefficient because of a longer waiting time and response time.

In **Concurrent schedule**,

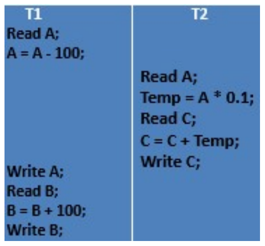
- CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly.



Concurrent Schedule	
T1	T2
Read A;	
A = A - 100;	
Write A;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Read B;	
B = B + 100;	
Write B;	

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:
  - Lost update problem (Write – Write conflict)
  - Dirty read problem (W-R conflict)
  - Unrepeatable read (R-W Conflict)



T1	T2
Read A;	
A = A - 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

Lost update problem (Write – Write conflict)

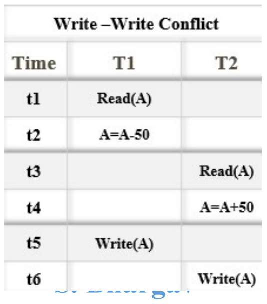
- This type of problem occurs when two transactions in database access the same data item and have their operations in an interleaved manner that makes the value of some database item incorrect.
- If there are two transactions T1 and T2 accessing the same data item value and then update it, then the second record overwrites the first record.

Example: Let's take the value of A is 100

Here,

- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T1 transaction deducts the value of A by 50.
- At t3 time, T2 transactions read the value of **A i.e., 100**.
- At t4 time, T2 transaction adds the value of A by 150.
- At t5 time, T1 transaction writes the value of A data item on

the basis of value seen at time t2 i.e., 50.



Write –Write Conflict		
Time	T1	T2
t1	Read(A)	
t2	A=A-50	
t3		Read(A)
t4		A=A+50
t5	Write(A)	
t6		Write(A)

- At t6 time, T2 transaction writes the value of A based on value seen at time t4 i.e., 150.
- So at time T6, the update of Transaction T1 is lost because Transaction T2 overwrites the value of A without looking at its current value, Such type of problem is known as the Lost Update Problem.

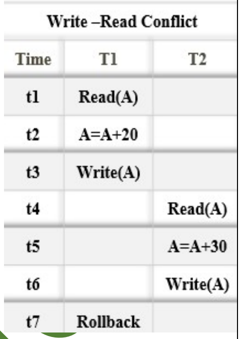
Dirty read problem (W-R conflict)

- This type of problem occurs when one transaction T1 updates a data item of the database, and then that transaction fails due to some reason, but its updates are accessed by some other transaction.

Example: Let's take the value of A is 100

- At t1 time, T1 transaction reads the value of A i.e., 100.
- At t2 time, T1 transaction adds the value of A by 20.
- At t3 time, T1transaction writes the value of A (120) in the database.
- At t4 time, T2 transactions read the value of A data item i.e., 120.
- At t5 time, T2 transaction adds the value of A data item by 30.
- At t6 time, T2transaction writes the value of A (150) in the database.
- At t7 time, a T1 transaction fails due to power failure then it is

rollback according to atomicity property of transaction (either all or none).



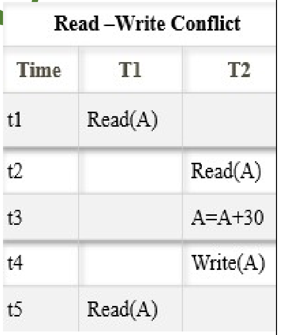
Write –Read Conflict

Time	T1	T2
t1	Read(A)	
t2	A=A+20	
t3	Write(A)	
t4		Read(A)
t5		A=A+30
t6		Write(A)
t7	Rollback	

- So, transaction T2 at t4 time contains a value which has not been committed in the database. The value read by the transaction T2 is known as a dirty read.

Unrepeatable read (R-W Conflict)

- It is also known as an inconsistent retrieval problem. If a transaction T_1 reads a value of data item twice and the data item is changed by another transaction T_2 in between the two read operation. Hence T_1 access two different values for its two read operation of the same data item.
- Example: Let's take the value of A is 100.
  - At t1 time, T1 transaction reads the value of A i.e., 100.
  - At t2 time, T2transaction reads the value of A i.e., 100.
  - At t3 time, T2 transaction adds the value of A data item by 30.
  - At t4 time, T2 transaction writes the value of A (130) in the database.
- Transaction T2 updates the value of A. Thus, when another read statement is performed by transaction T1, it accesses the new value of A, which was updated by T2. Such type of conflict is known as R-W conflict.



Read –Write Conflict

Time	T1	T2
t1	Read(A)	
t2		Read(A)
t3		A=A+30
t4		Write(A)
t5	Read(A)	

Serializability in DBMS:

- Some non-serial schedules may lead to inconsistency of the database.
- Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

Serializable Schedules-

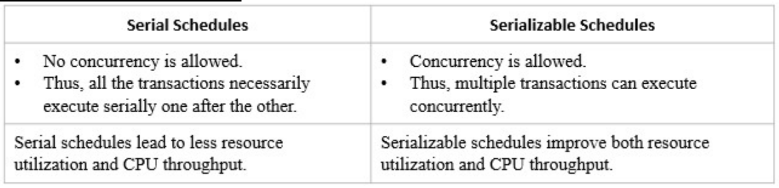
- If a given non-serial schedule of 'n' transactions is equivalent to some serial schedule of 'n' transactions, then it is called as a serializable schedule.

Characteristics-

Serializable schedules behave exactly same as serial schedules.

- Thus, serializable schedules are always-
  - Consistent
  - Recoverable
  - Casacadeless
  - Strict

<u>Serial Schedules Vs Serializable Schedules-</u>



Serial Schedules	Serializable Schedules
• No concurrency is allowed. • Thus, all the transactions necessarily execute serially one after the other.	• Concurrency is allowed. • Thus, multiple transactions can execute concurrently.
Serial schedules lead to less resource utilization and CPU throughput.	Serializable schedules improve both resource utilization and CPU throughput.

Types of Serializability-

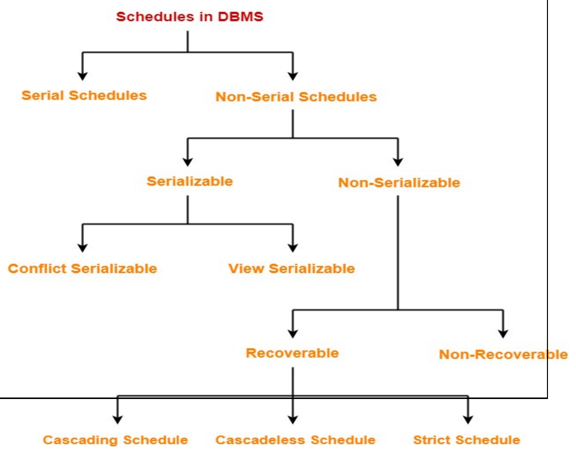
- Conflict Serializability
- View Serializability

Conflict Serializability-

- If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a conflict serializable schedule.

Conflicting Operations-

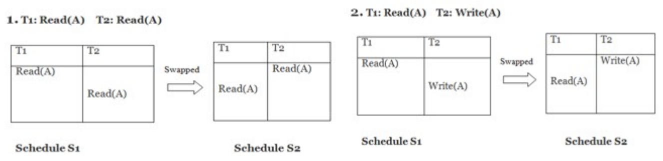
- Two operations are called as conflicting operations if all the following conditions hold true for them-
  - Both the operations belong to different transactions
  - Both the operations are on the same data item



- o At least one of the two operations is a write operation

Example:

- Swapping is possible only if S1 and S2 are logically equal.



T1	T2
Read(A)	
	Read(A)

T1	T2
	Read(A)
Read(A)	

T1	T2
Read(A)	
	Write(A)

T1	T2
	Write(A)
Read(A)	

Here, S1 = S2. That means it is non-conflict. Here, S1 ≠ S2. That means it is conflict.

Conflict Equivalent

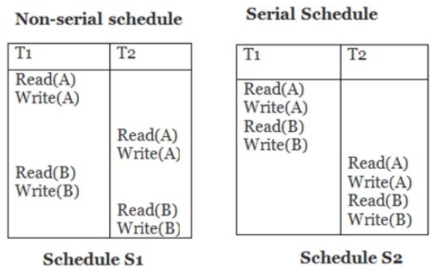
- In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

- o They contain the same set of the transaction.
- o If each pair of conflict operations are ordered in the same way

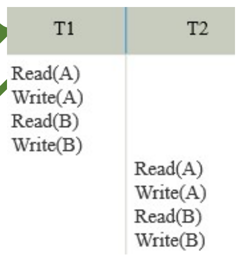
Example:

- Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.
- After swapping of non-conflict operations, the schedule S1 becomes:


Non-serial schedule
Serial Schedule
Schedule S1
Schedule S2

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

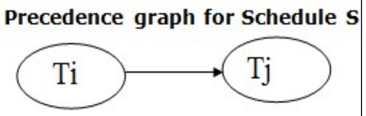


T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Testing of Serializability

Serialization Graph is used to test the Serializability of a schedule.

- Assume a schedule S. For S, we construct a graph known as precedence graph. This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain all edges Ti ->Tj for which one of the three conditions holds:
  - o Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q).
  - o Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q).
  - o Create a node Ti → Tj if Ti executes write (Q) before Tj executes write (Q).



Precedence Graph

- If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
- If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

Precedence Graph

- If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.

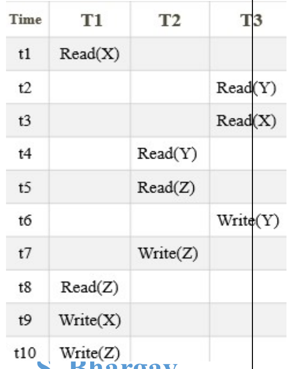
If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

Schedule S2 (Non-Serial Schedule):

Precedence graph for schedule S2:

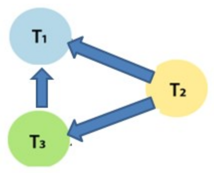
- In the above schedule, there are three transactions: T1, T2, and T3. So, the precedence graph contains three vertices.
- To draw the edges between these nodes or vertices, follow the below steps:

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.

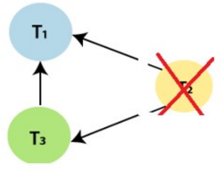


Time	T1	T2	T3
t1	Read(X)		
t2			Read(Y)
t3			Read(X)
t4		Read(Y)	
t5		Read(Z)	
t6			Write(Y)
t7		Write(Z)	
t8	Read(Z)		
t9	Write(X)		
t10	Write(Z)		

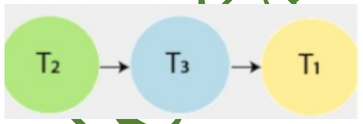
- ➢ After all the steps, the precedence graph will be ready, and it does not contain any cycle or loop, so the above schedule S2 is conflict serializable. And it is equivalent to a serial schedule. Above schedule S2 is transformed into the serial schedule by using the following steps:



- ➢ **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 T1→T3.
- ➢ **Step3:** And at last, take the vertex T1 and connect with T3.



- ➢ Then, **Precedence graph equivalent to schedule S2**



View Serializability

- A schedule will view serializable if it is view equivalent to a serial schedule.
- **If a schedule is conflict serializable, then it will be view serializable.**
- The view serializable which does not conflict serializable contains blind writes.

View Equivalent

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

1.Initial Read

- ➢ The initial read of both the schedules must be in the same transaction.
- ➢ Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.
- ➢ The two schedules S1 and S2 are view equivalent because Initial read operation in S1 is done by T1 and in S2 also it is done by T1.

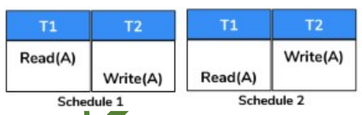
2.Updated Read

- Suppose in schedule S1, if transaction ***Tm is reading A which is updated by transaction Tn then in S2 also, Tm should read A which is updated by Tn.***
- The two schedules are ***not view equal*** because, ***in S1,transaction T3 is reading A updated by transaction T2 and in S2, transaction T3 is reading A which is updated by transaction T1.***

3.Final Write

- A final write must be the ***same in both the schedules.***
- Suppose in schedule ***S1, if a transaction T1 updates A in the last, then in S2 final write operation should also be done by transaction T1.***
- The two schedules is view equal because Final write operation in S1 is done by T3 and in S2 also the final write operation is done by T3.

1. Initial Read



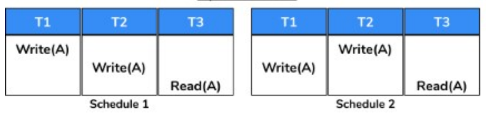
T1	T2
Read(A)	
	Write(A)

Schedule 1

T1	T2
	Write(A)
Read(A)	

Schedule 2

2. Updated Read



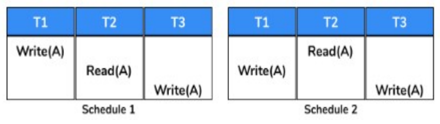
T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule 1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule 2

3. Final Write



T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

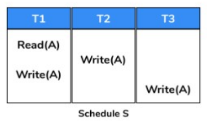
Schedule 1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule 2

- Consider a schedule S with 3 transactions.
- The total number of possible schedules is 3!=6.They are
- S1 =
- S2 =
- S3 =
- S4 =
- S5 =
- S6 =

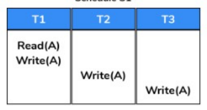
Example



T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

Schedule S

Schedule S1



T1	T2	T3
Read(A)		
Write(A)		
	Write(A)	
		Write(A)

Considering the first schedule

Schedule S1

Step 1: Final updation on data items

- In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read

- The initial read operation in S is done by T1 and in S1, it is also done by T1.

Step 3: Final Write

- The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.
- The first schedule S1 satisfies all three conditions, so we don't need to check another schedule. Hence, view equivalent serial schedule is **T1 → T2 → T3**

Non - Serializable Schedules

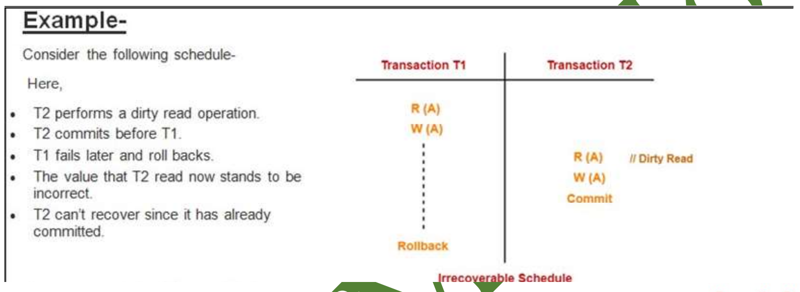
- A non-serial schedule which is not serializable is called as a non-serializable schedule.
- A non - serializable schedule is not guaranteed to produce the same effect as produced by some serial schedule on any consistent database.

Characteristics-

➢ Non - serializable schedules-
  - may or may not be consistent
  - may or may not be recoverable

Irrecoverable Schedules-

➢ If in a schedule,
  - A transaction performs a dirty read operation from an uncommitted transaction
  - And commits before the transaction from which it has read the value then such a schedule is known as an Irrecoverable Schedule.



Example-

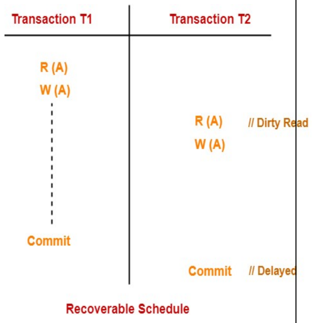
Consider the following schedule-

Here,

- T2 performs a dirty read operation.
- T2 commits before T1.
- T1 fails later and roll backs.
- The value that T2 read now stands to be incorrect.
- T2 can't recover since it has already committed.

Recoverable Schedules-

- If in a schedule,
  - A transaction performs a dirty read operation from an uncommitted transaction
  - And its commit operation is delayed till the uncommitted transaction either commits or roll backs then such a schedule is known as a Recoverable Schedule.
- Here,
  - The commit operation of the transaction that performs the dirty read is delayed.
  - This ensures that it still has a chance to recover if the uncommitted transaction fails later.



Checking Whether a Schedule is Recoverable or Irrecoverable-

Method-01:

- Check whether the given schedule is conflict serializable or not.
  - ✓ If the given schedule is conflict serializable, then it is surely recoverable. Stop and report your answer.
  - ✓ If the given schedule is not conflict serializable, then it may or may not be recoverable. Go and check using other methods.

Method-02:

- Check if there exists any dirty read operation.(Reading from an uncommitted transaction is called as a dirty read)
  - ✓ If there does not exist any dirty read operation, then the schedule is surely recoverable. Stop and report your answer.
  - ✓ If there exists any dirty read operation, then the schedule may or may not be recoverable.

Types of Recoverable Schedules-

A recoverable schedule may be any one of these kinds-

1. Cascading Schedule
2. Cascade less Schedule
3. Strict Schedule

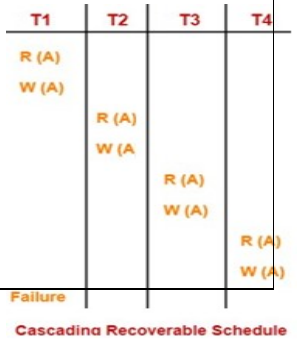
Cascading Schedule-

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 Schedule** or **Cascading Rollback** or **Cascading Abort**.

It simply leads to the wastage of CPU time.

Here,

- Transaction T2 depends on transaction T1.
- Transaction T3 depends on transaction T2.
- Transaction T4 depends on transaction T3.
- In this schedule,
- The failure of transaction T1 causes the transaction T2 to rollback.

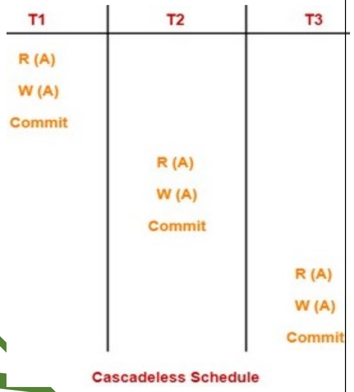


- The rollback of transaction T2 causes the transaction T3 to rollback.
- The rollback of transaction T3 causes the transaction T4 to rollback.
- Such a rollback is called as a **Cascading Rollback**.

NOTE- If the transactions T2, T3 and T4 would have committed before the failure of transaction T1, and then the schedule would have been irrecoverable.

Cascade-less Schedule-

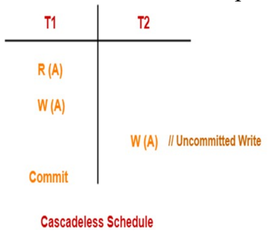
- If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Cascade less Schedule**.
- Cascadeless schedule allows only committed read operations.
- Therefore, it avoids cascading roll back and thus saves CPU time.



NOTE-

- Cascadeless schedule allows only committed read operations.
- However, it allows uncommitted write operations.

Example-

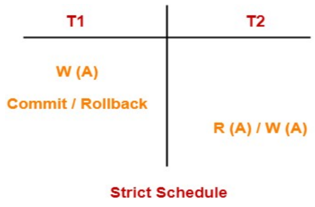


Strict Schedule-

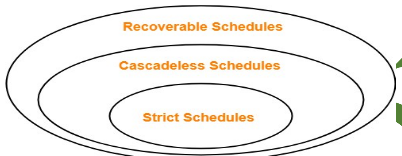
- If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Strict Schedule**.

In other words,

- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.



Example-



Remember-

- Strict schedules are stricter than cascadeless schedules.
- All strict schedules are cascadeless schedules.
- All cascadeless schedules are not strict schedules.

Concurrency Control Protocols

Concurrency Control

- Concurrency control is the way to preserve isolation of transactions while managing concurrent execution.
- We know that serializability ensures the consistency of a database.
- So, concurrency control schemes are mostly based on the serializability property.

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

1.shared (*S*) *mode*. Data item can only be read. S-lock is requested using lock-S instruction.
2.exclusive (*X*) *mode*. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

	S	X
S	True	False
X	False	False

Lock-compatibility matrix →

- 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.

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.

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

```
T1: A→B
lock-X(A);
read (A);
A= A-100;
write(A);
unlock(A);
lock-X(B);
read (B);
B=B+100;
Write(B);
unlock(B);
display(A+B)
```

```
T2:Display (A+B)
lock-S(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display(A+B)
```

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

Deadlock:

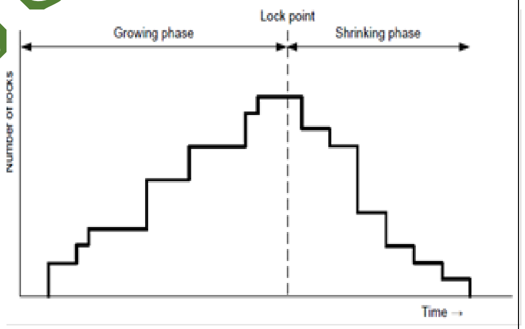
- Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

The Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase
- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking 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).

- Problems with 2PL
  - Unnecessary or early lock
  - Dead lock
  - Cascading rollback

Example: The following way shows how unlocking and locking work with 2-PL.

T1	T2
LOCK-S(A)	
	LOCK-S(A)
LOCK-X(B)	
——	——
UNLOCK(A)	
	LOCK-X(C)
UNLOCK(B)	
	UNLOCK(A)
	UNLOCK(C)
——	——

Transaction T1:
- Growing phase: from step 1-3
- Shrinking phase: from step 5-7
- Lock point: at 3

Transaction T2:
- Growing phase: from step 2-6
- Shrinking phase: from step 8-9
- Lock point: at 6

If lock conversion is allowed then the following phase can happen:
- Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
- Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

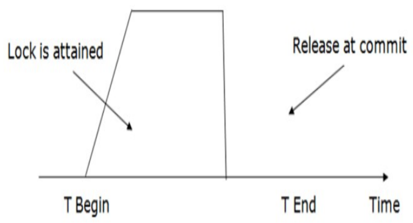
It is true that the 2PL protocol offers serializability.
- – However, it does not ensure that deadlocks do not happen.
- – To avoid this, follow a modified protocols

Conservative 2PL.
- No growing phase (grant lock permissions for all data item at once)
- Reduce deadlocks , but practical implementation is difficult.
- Problem with cascading rollback

Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.

- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock(exclusive) until commit.
- Allow unlocking for shared lock items only.
- No cascading rollback.
- Chance of deadlock.



Rigorous two-phase locking is even stricter:

- All data items unlock after commit or rollback (all locks X and S).
- No Shrinking phase.
- All locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

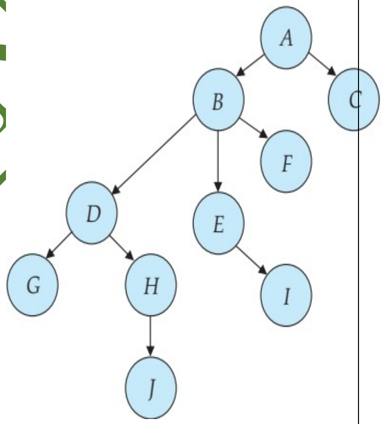
Implementation of Locking

- 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 a deadlock is detected.
- 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

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering → on the set D = {d1, d2 ,..., dh} of all data items.
- One idea is to have prior knowledge on about the order in which the database items will be accessed.
- If di → dj then any transaction accessing both di and dj, must access di before accessing dj
- Implies that the set D may now be viewed as a directed acyclic graph, called a database graph.
- The tree-protocol is a simple kind of graph protocol.



Tree protocol:

1. Only exclusive locks are allowed.
2. The first lock by Ti may be on any data item. Subsequently, a data Q can be locked by Ti only if the parent of Q is currently locked by Ti .
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.

Timestamps

- With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by TS (Ti). This timestamp is assigned by the database system before the Transaction Ti starts execution.
- To implement this scheme we associate with each data item Q, two timestamp values:
- **1. W-timestamp W_TS(Q):** denotes the largest timestamp of any transaction that executed write(Q) successfully.
- **2. R-timestamp R_TS(Q):** denotes the largest timestamp of any transaction that executed read(Q) successfully.

The timestamp ordering protocol

1. Suppose that transaction Ti issues read (Q)

- If TS(Ti) < W_TS(Q), then Ti needs to read value of Q that was already overwritten. Hence read operation is **rejected** and Ti is rolled back.
- if TS(Ti) > W_TS(Q), the read operation is **executed** and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti)

2. Suppose that transaction Ti issues write(Q)

- If TS(Ti) < R_TS(Q), then the value of Q that Ti is producing was needed previously and the system assumed that, that value would never be produced, Hence the system **rejects** the write operation and rolls Ti back.
- If TS(Ti)<W_TS(Q), then Ti is attempting to write an obsolete value of Q, Hence the system **rejects** this write operation and rolls Ti back.
- Otherwise, the system executes the write operation and sets W-timestamp to TS(Ti)

Few Points

- Timestamp Ordering protocol ensures conflict serializability.

- The protocol ensures freedom from the deadlock, since no transaction ever waits. However, there is a possibility of Starvation of long transaction if a sequence of conflicting start transactions caused repeated restarting of the long transaction.
- Protocol generates the schedules that are not recoverable.

Validation Based Protocol

1.Read Phase: During this phase, the system executes transaction Ti. . It reads the values of the various data items and stores them
in variable local to Ti. It performs all the write operations on temporary local variables without update of the actual database.
2.Validation Phase: Transaction Ti performs a validation test to determine whether it can copy to database the temporary local
variables that hold the result of write operations without causing a violation of serializability.
3.Write Phase: If Transaction Ti succeeds in validation, then the system applies the actual updates to the database, otherwise the
system rolls back Ti.

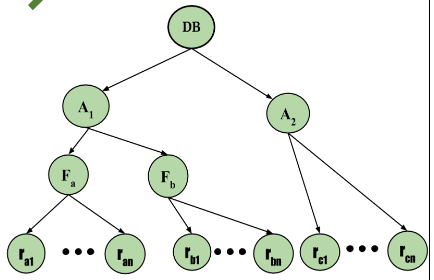
1.Resolves the cascade rollbacks
2.Suffers from the starvation.
3.Optimistic concurrency control.

Multiple Granularity

- Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

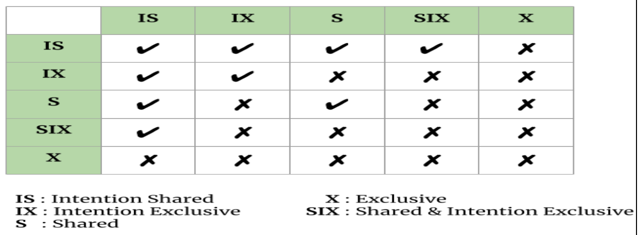
- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.
- Consider a tree which has four levels of nodes.
  - Database
  - Area
  - File
  - Record



Intention Mode Lock –
In addition to S and X lock modes,
there are three additional lock modes with multiple granularities:

- **Intention-Shared (IS):** explicit locking at a lower level of the tree but only with shared locks.
- **Intention-Exclusive (IX):** explicit locking at a lower level with exclusive or shared locks.
- **Shared & Intention-Exclusive (SIX):** the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive mode locks.

Compatibility Matrix with Intention Lock Modes:
The below table describes the compatibility matrix for these lock modes:



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

IS : Intention Shared
IX : Intention Exclusive
S : Shared
X : Exclusive
SIX : Shared & Intention Exclusive