UNIT IV

Transaction

A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

1. Open_Account(X)
2. Old_Balance = X.balance
3. New_Balance = Old_Balance - 800
4. X.balance = New_Balance
5. Close_Account(X)

Y's Account

1. Open_Account(Y)
2. Old_Balance = Y.balance
3. New_Balance = Old_Balance + 800
4. Y.balance = New_Balance
5. Close_Account(Y)

Therefore ,

Transaction Concept

- A transactionis a unit of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Atomicity

- It states that all operations of the transaction take place at once if not, the transaction is aborted.
- There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Abort: If a transaction aborts then all the changes made are not visible.

Commit: If a transaction commits then all the changes made are visible.

Consistency

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

Consistency

- The integrity constraints are maintained so that the database is consistent before and after the transaction.
- The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- The consistent property of database states that every transaction sees a consistent database instance.
- The transaction is used to transform the database from one consistent state to another consistent state.

Durability

- The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
- They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
- The recovery subsystem of the DBMS has the responsibility of Durability property.

 Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.

 Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

 Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

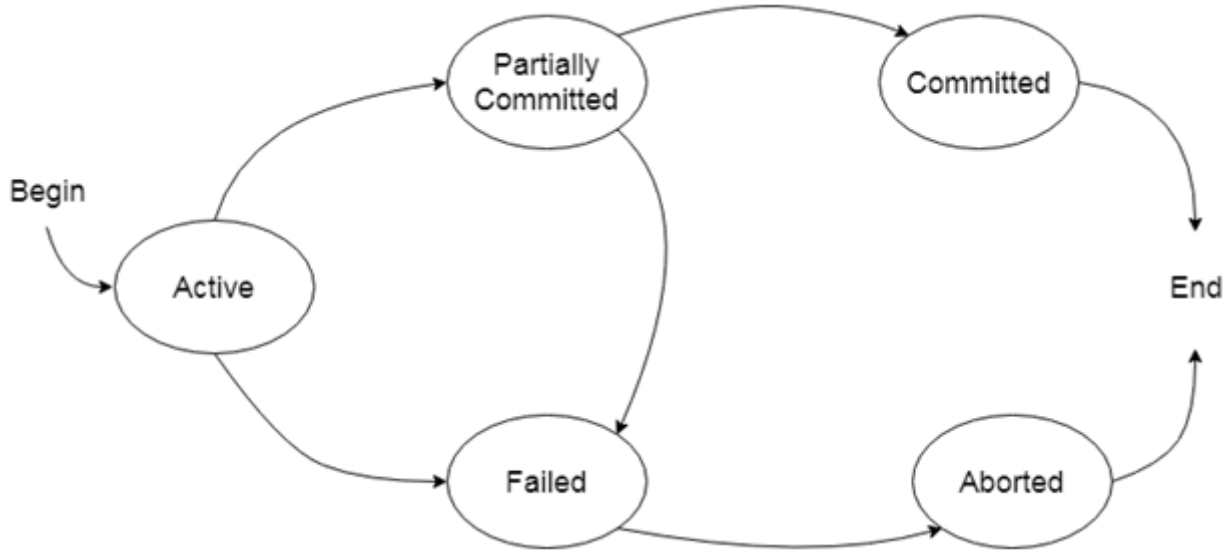
 Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).

Can be ensured trivially by running transactions serially, that is one after the other.However, executing multiple transactions concurrently has significant benefits, as

we will see.

States of Transaction

In a database, the transaction can be in one of the following states -



Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
    1. Re-start the transaction
    2. Kill the transaction

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for

atomicity and durability.

- The shadow-database scheme:

- assume that only one transaction is active at a time.
- a pointer called db_pointer always points to the current consistent copy of the

database.

- all updates are made on a shadow copy of the database, and db_pointer is made

to point to the updated shadow copy only after the transaction reaches partial

commit and all updated pages have been flushed to disk.

- in case transaction fails, old consistent copy pointed to by db_pointer can be

used, and the shadow copy can be deleted.

- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the entire database.

Concurrent Executions

 Multiple transactions are allowed to run concurrently in the system. Advantages are:

 increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk

 reduced average response time for transactions: short transactions need not wait behind long ones.

 Concurrency control schemes – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

 Schedules – sequences that indicate the chronological order in which instructions of concurrent transactions are executed

 a schedule for a set of transactions must consist of all instructions of those transactions

 must preserve the order in which the instructions appear in each individual transaction.

Serializability

 Basic Assumption – Each transaction preserves database consistency.

 Thus serial execution of a set of transactions preserves database consistency.

 A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

Different forms of schedule equivalence give rise to the notions of:

1. conflictserializability

2. viewserializability

 We ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only read and write instructions.

Conflict Serializable Schedule

- A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.

- The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

Conflicting Operations

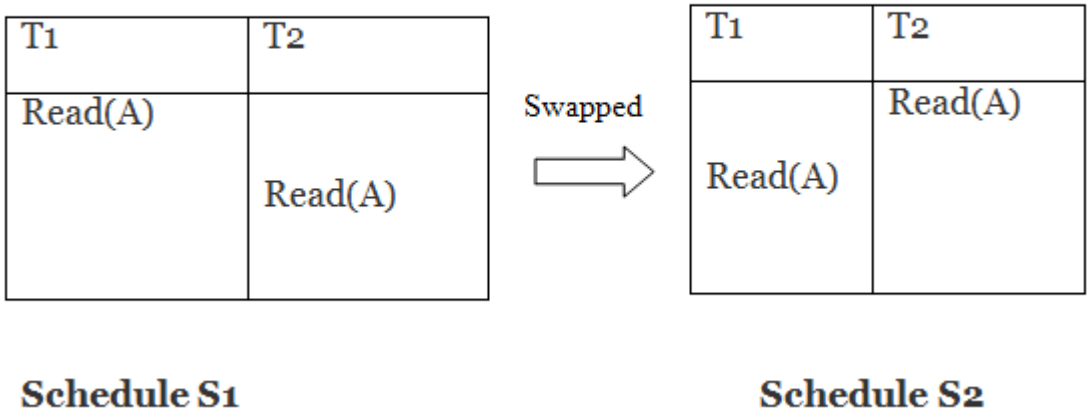
The two operations become conflicting if all conditions satisfy:

1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

Example:

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

1. T1: Read(A) T2: Read(A)



Schedule S1 **Schedule S2**

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

1. They contain the same set of the transaction.

2. If each pair of conflict operations are ordered in the same way.

Example:

Non-serial schedule

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

Schedule S1

Serial Schedule

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

Schedule S2

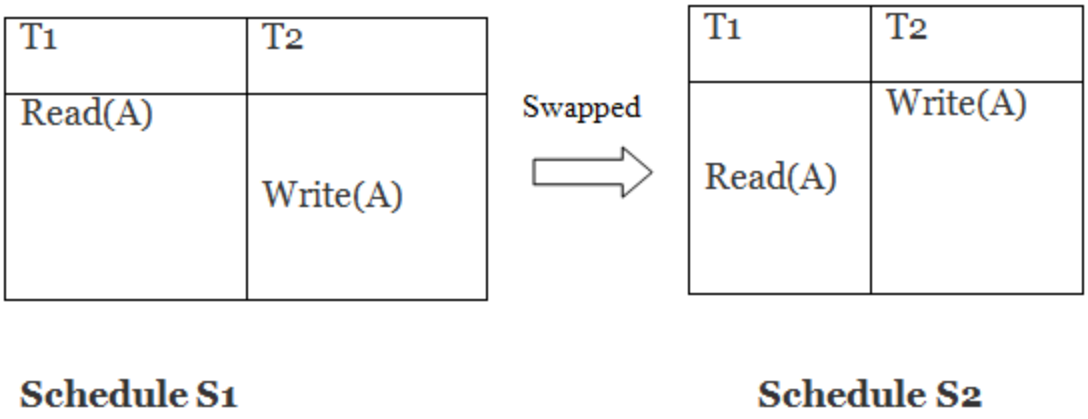
Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

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

Since, S1 is conflict serializable.

2. T1: Read(A) T2: Write(A)



Schedule S1 **Schedule S2**

Here, S1 ≠ S2. That means it is conflict

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

An initial read of both schedules must be the same. 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.

T1	T2
Read(A)	Write(A)

Schedule S1

T1	T2
Read(A)	Write(A)

Schedule S2

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

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

Schedule S1

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

Schedule S2

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

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

Schedule S1

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

Schedule S2

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Example:

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

Schedule S

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>
7. S6 = <T3 T2 T1>

Taking first schedule S1:

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

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

Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not

committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

Implementation of Isolation

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules,
but provides a poor degree of concurrency..
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Transaction Definition in SQL

 Data manipulation language must include a construct for specifying the set of actions

that comprise a transaction.

 In SQL, a transaction begins implicitly.

 A transaction in SQL ends by:

1) Commit work commits current transaction and begins a new one.

2) Rollback work causes current transaction to abort.

3) Levels of consistency specified by SQL-92:

4) Serializable— default

5) Repeatable read

6) Read committed

7) Read uncommitted

Levels of Consistency in SQL-92 (SQL-92 was the third revision of the SQL database query language. Unlike SQL-89, it was a major revision of the standard. Aside from a few minor incompatibilities, the SQL-89 standard is forward-compatible with SQL-92. **) i.e., t**he standard specificationi

 Serializable— default

 Repeatable read —only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.

 Read committed —only committed records can be read, but successive reads of record may return different (but committed) values.

 Read uncommitted —even uncommitted records may be read.

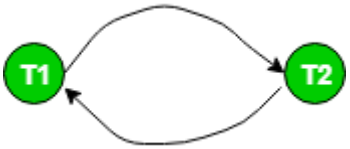
Testing for Serializability

 Consider some schedule of a set of transactions T 1 , T 2 , ..., T n

 Precedence graph— a direct graph where the vertices are the transactions (names).

 We draw an arc from T i to T j if the two transaction conflict, and T i accessed the data item on which the conflict arose earlier.

 We may label the arc by the item that was accessed.



Test for Conflict Serializability

 A schedule is conflict serializable if and only if its precedence graph is acyclic.

 Cycle-detection algorithms exist which take order n 2 time, where n is the number of vertices in the graph. (Better algorithms take order n + e where e is the number of edges.)

 If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. This is a linear order consistent with the partial order of the graph.

For example, a serializability order for Schedule A would be

T 5 →T 1 →T 3 →T 2 →T4.

Test for View Serializability

 The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.

 The problem of checking if a schedule is view serializable falls in the class of NP-complete problems. Thus existence of an efficient algorithm is unlikely.

However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Concurrency Control vs. Serializability Tests

 Testing a schedule for serializabilityafter it has executed is a little too late!

 Goal – to develop concurrency control protocols that will assure serializability. They will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline that avoids nonseralizable schedules.

 Tests for serializability help understand why a concurrency control protocol is correct.

Concurrency Control

 Lock-Based Protocols

 Timestamp-Based Protocols

 Validation-Based Protocols

 Multiple Granularity

Lock-Based Protocols

 A lock is a mechanism to control concurrent access to a data item

 Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.

 Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

 Lock-compatibility matrix

Types of failures:

1 Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

2. System Crash

System failure can occur due to power failure or other hardware or software failure.

Example: Operating system error.

Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.

Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Two phase commit protocol:

A two-phase commit is a standardized protocol that ensures that a database commit is implementing in the situation where a commit operation must be broken into two separate parts.

In database management, saving data changes is known as a commit and undoing changes is known as a rollback. Both can be achieved easily using transaction logging when a single server is involved, but when the data is spread across geographically-diverse servers in distributed computing (i.e., each server being an independent entity with separate log records), the process can become more tricky.

A special object, known as a coordinator, is required in a distributed transaction. As its name implies, the coordinator arranges activities and synchronization between distributed servers. The two-phase commit is implemented as follows:

Phase 1 - Each server that needs to commit data writes its data records to the log. If a server is unsuccessful, it responds with a failure message. If successful, the server replies with an OK message.

Phase 2 - This phase begins after all participants respond OK. Then, the coordinator sends a signal to each server with commit instructions. After committing, each writes the commit as part of its log record for reference and sends the coordinator a message that its commit has been successfully implemented. If a server fails, the coordinator sends instructions to all servers to roll back the transaction. After the servers roll back, each sends feedback that this has been completed.

/* One coordinator responsible for initiating protocol

Other entries called participants

If coordinator or participants unable to commit all parts of transaction are aborted between

those two

Phase 1 the coordinator sends can commit message to all participants

Phase 2 implement that decision at all sites in transaction */.

Recoverable schedules:

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, let us consider a transaction T1 , which did some operations and then transaction T2 also did some operations then transaction T1 should commit first and then transaction T2 . This type of schedules are called as Recoverable schedules.

Different types of locks:

Locks are mechanism used to ensure data integrity. The oracle engine automatically locks table data while executing SQL statements like Select/insert/UPDATE/DELETE. This K type of locking is called implicit locking

There are two types of Locks

1. Shared lock

2. Exclusive lock

Shared lock:

Shared locks are placed on resources whenever a read operation (select) is performed. Multiple shared locks can be simultaneously set on a resource.

Exclusive lock:

Exclusive locks are placed on resources whenever a write operation (INSERT, UPDATE And DELETE) are performed. Only one exclusive lock can be placed on a resource at a time. i.e. the first user who acquires an exclusive lock will continue to have the sole ownership of the resource, and no other user can acquire an exclusive lock on that resource.

Types of Schedules based Recoverability in DBMS:

Generally, there are three types of schedule given as follows:

1. Recoverable Schedule –

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not

committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

3. Strict Schedule –

if schedule contains no read or write before commit then it is known as strict schedule. Strict schedule is strict in nature.

cascadeless schedules:
This schedule avoids all possible Dirty Read Problem.

In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits. That means there must not be Dirty Read. Because Dirty Read Problem can cause Cascading Rollback, which is inefficient.
Cascadeless Schedule avoids cascading aborts/rollbacks (ACA). Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.
In other words, if some transaction Tj wants to read value updated or written by some other
transaction Ti, then the commit of Tj must read it after the commit of Ti.

Note that Cascadeless schedule allows only committed read operations. However, it allows uncommitted write operations.
Also note that Cascadeless Schedules are always recoverable, but all recoverable transactions may not be Cascadeless Schedule.

phases of transactions:

Active state: This phase is divided into two states:
- Initial phase: This phase is achieved when the transaction starts.
- Partially Committed phase: This is achieved when the transactions final statement has been executed. Even though the final statement is finished execution, the transaction may abort due to some failure.
- Failed state: This state is reached when the normal execution fails.
- Aborted state: A transaction is aborted when the system feels it needs to be failed. This state should not have any effect on the system and thus all changes done until it were aborted; are rolled back.
- Committed: After the transaction is successfully executed, it enters the committed state. In this state all changes are committed. These committed changes cannot be undone or aborting.