UNIT –III

TRANSACTION MANAGEMENT

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control –Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery – Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery – Backup and Recovery System – SQL – DCL – TCL commands.

3.1 TRANSACTION CONCEPTS:

The term transaction refers to a collection of operations that form a single logical unit of work.

Example: Transfer of money from one account to another is a transaction.

A transaction is initiated by a user program written in a high-level programming language with embedded database accesses in JDBC or ODBC.

A transaction is delimited by statements of the form **begin transaction and end transaction**.

3.2 ACID PROPERTIES:

- **Atomicity:**
  Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency:**
  With no other transaction executing concurrently preserves the consistency of the database.
- **Isolation:**
  Each transaction is unaware of other transactions executing concurrently in the system.
- **Durability:**
  After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties.**

Transactions access data using two operations:

- **read(*X*)**, which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory.
- **write(*X*)**, which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the write to the data item *X* in the database.

Example:

Let *Ti* be a transaction that transfers $50 from account *A* to account *B*. This transaction can be defined as:

Ti : read(*A*);
A := *A* − 50;
write(*A*);
read(*B*);
B := *B* + 50;
write(*B*).

Let us now consider each of the ACID properties.

• **Consistency**: The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction. This task may be facilitated by **automatic testing of integrity constraints**.

Atomicity:The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the log. If the transaction does not complete its execution, the database

system restores the old values from the log to make it appear as though the transaction never executed.

Suppose that, just before the execution of transaction *Ti*, the values of accounts *A* and *B* are $1000 and $2000, respectively. Suppose that the failure happened after the write(*A*) operation but before the write(*B*) operation. In this case, the values of accounts *A* and *B* reflected in the database are $950 and $2000.

The system destroyed $50 as a result of this failure. In particular, we note that the sum *A* + *B* is no longer preserved.

Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system.**

Durability: The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We can guarantee durability by ensuring that either:

1. The updates carried out by the transaction have been written to disk before the transaction completes.

2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

The **recovery system** of the database is responsible for ensuring durability.

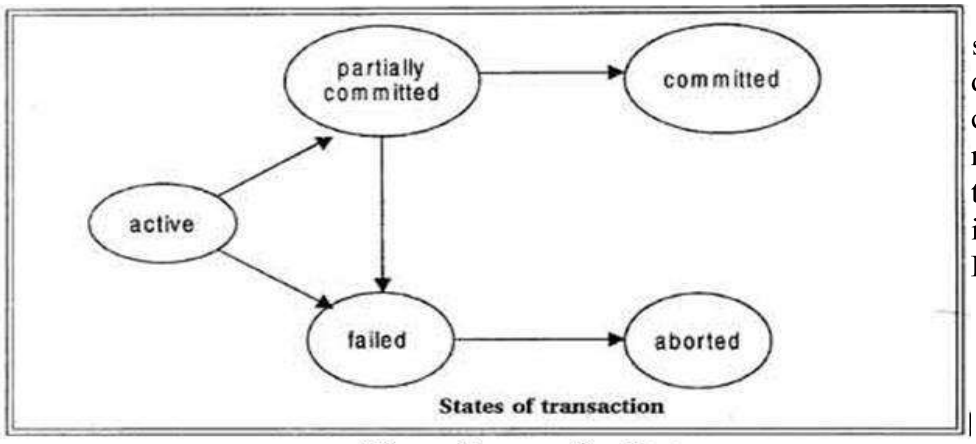
Isolation: If several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, the database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to *A* and the increased total yet to be written to *B*. If a second concurrently running transaction reads *A* and *B* at this intermediate point and computes *A*+*B*, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on *A* and *B* based on the inconsistent values that it read, the database may be left in an inconsistent
state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other.

Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system.**

States of Transaction



The state diagram corresponding to a transaction is shown in Figure.

Fig Transaction States

A transaction must be in one of the following states:

- **Active:** the initial state, the transaction stays in this state while it is executing.
- **Partially committed:** after the final statement has been executed.
- **Failed:** when the normal execution can no longer proceed.
- **Aborted:** after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed:** after successful completion.

3.3 SCHEDULES:

Schedule is defined as a sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed.

A schedule is serializable if it is equivalent to a serial schedule.

A schedule where the operations of each transaction are executed consecutively without any interference from other transactions is called serial sechedule.

Types of serializability are

1. Conflict Serializability
2. View Serializability

3.4 SERIALIZABILITY:

Conflict Serializability:

Instructions Ii and Ij, of transactions Ti and Tj respectively, conflict if and only if there exists some item Q accessed by both Ii and Ij, and at least one of these instructions wrote Q.

1.Ii = read(Q), Ij = read(Q). Ii and Ij don't conflict.
2.Ii = read(Q), Ij = write(Q). They conflict.
3.Ii = write(Q), Ij = read(Q). They conflict.
4.Ii = write(Q), Ij = write(Q). They conflict.

If Ii and Ij are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Consider following schedule 3.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule-3 - showing only the read and write operations

The write (A) of T_1 conflicts with read (A) of T_2. However, write (A) of T_2 does not conflict

with read (B) of T_1, because, the two instructions access different data items.

Because of no conflict, we can swap write (A) and read (B) instructions to generate a new schedule 5. Regardless of initial system state, schedule 3 and 5 generates same result.

T_1	T_2
read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	
	read (B)
	write (B)

Schedule 5-schedule 3 after swapping a pair of instructions

We can continue to swap non-conflicting instructions:

- Swap the **read (*B*)** instruction of T_1 with **read (*A*)** instruction of T_2.
- Swap the **write (*B*)** instruction of T_1 with **write (*A*)** instruction of T_2.
- Swap the **write (*B*)** instruction of T_1 with the **read (*A*)** instruction of T_2.

The final result of these swaps - is shown below, which is serial schedule.

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6-a serial schedule that is equivalent to schedule 3

If a schedule S can be transformed into a schedule S_1 by a series of swaps of

non-conflicting instructions, we say that *S* and *S*1 are **conflict equivalent**.

The concept of conflict equivalence loads to the concept of conflict serializability. We say that a schedule *S* is **conflict serializable**, if it is conflict equivalent to a serial schedule. Thus schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Consider schedule 7. It consists of two transactions T_3 and T_4. This schedule is not conflict serializable, since it is not equivalent to either the serial schedule $<T_3, T_4>$ or the serial schedule $<T_4, T_3>$.

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

Schedule 7

View Serializability:

A schedule S is view serializable if it is view equivalent to a serial schedule.

Let S and S*0* be two schedules with the same set of transactions. S and S*0* are view equivalent if the following three conditions are met:

1. For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S*0*, also read the initial value of Q.
2. For each data item Q, if transaction Ti executes read(Q) in schedule S, and that value was produced by transaction Tj (if any), then transaction Ti must in schedule S*0* also read the value of Q that was produced by transaction Tj.
3. For each data item Q, the transaction (if any) that performs the final write (Q) operation in schedule S must perform the final write (Q) operation in schedule S*0*.

Every conflict serializable schedule is also view serializable.

The following schedule is view-serializable but not conflict serializable

T_3	T_4	T_6
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Schedule 8-a view serializable schedule

In the above schedule, transactions T4 and T5 performs write(Q) operations without having performed a read(Q) operation. Writes of this sort are called **blind writes**. View serializable schedule with blind writes is not conflict serializable.

Testing for Serializability:

Testing for Serializability is done by using a directed graph called **precedence graph**, constructed from schedule.

This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.

The set of vertices consists of all the transactions participating in the schedule.

The set of edges consists of all edges Ti→ Tj for which one of three conditions holds:

1. Ti executes write(Q) before Tj executes read(Q).
2. Ti executes read(Q) before Tj executes write(Q).
3. Ti executes write(Q) before Tj executes write(Q).

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Figure Schedule 1 — a serial schedule in which T_1 is followed by T_2.

The precedence graph for schedule 1 and schedule 2 are shown in figure given below.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	

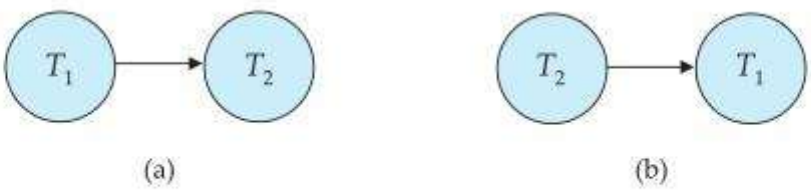


Figure Precedence graph for (a) schedule 1 and (b) schedule 2.

The precedence graph for schedule 1 contains a single edge T1→T2,since all the instructions of T1 are executed before the first instruction of T2 is executed.

The precedence graph for schedule 2 contains a single edge T2→T1,since all the instructions of T2 are executed before the first instruction of T1 is executed.

Consider the following schedule 4.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Figure Schedule 4 — a concurrent schedule resulting in an inconsistent state.

The precedence graph for schedule 4 is shown below.

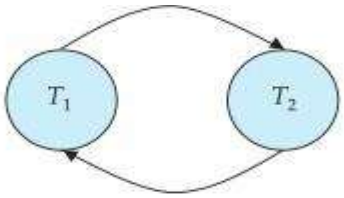


Figure Precedence graph for schedule 4.

To test conflict serializability construct a precedence graph for given schedule. If the graph contains cycle, the schedule is not conflict serializable. If the graph contains no cycle, the schedule is conflict serializable.

Schedule 1 and schedule 2 are conflict serializable as the precedence graph for both schedules does not contain any cycle. While the schedule 4 is not conflict serializable as the precedence graph for it contains cycle.

A serializability order of the transactions can be obtained through **topological sorting**, which determines a linear order consistent with the partial order of the precedence graph.

(a)Test for Conflict Serializability

To test conflict serializability, construct a **precedence graph** for given schedule. If graph contains cycle, the schedule is not conflict serializable. If the graph contains no cycle, then the schedule is conflict serializable.

Schedule 1 and schedule 2 are conflict serializable, as the precedence graph for both schedules does not contain any cycle. However the schedule 9 is not conflict serializable, as precedence graph for it contains cycle.

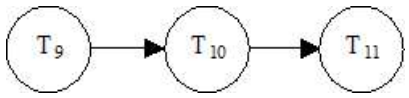
Example: Consider the schedule given in Fig. 5.14. Find out whether that schedule is conflict serializable or not?

T_9	T_{10}	T_{11}
read (A) $A := f_1(A)$ write (A)		
	read (A) $A := f_2(A)$ write (A) read (B) $B := f_3(B)$ write (B)	
		read (B) $B := f_4(B)$ write (B)

Concurrent schedule

Solution:

The precedence graph for given schedule is



As the graph is acyclic, so the schedule is conflict serializable.

3.5 CONCURRENCY CONTROL:

The system must control the interaction among the concurrent transactions. This control is achieve through one of concurrency control schemes. The concurrency control schemes are based on the serializability property.

Different types of protocols/schemes used to control concurrent execution of transactions.

3.6 LOCKING PROTOCOLS:

Locking is a protocol used to control access to data when one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results. Locking is one of the most widely used mechanisms to ensure serializability.

To ensure serializability, it is required that data items should be accessed in mutual exclusive manner; if one transaction is accessing a data item, no other transaction can modify that data item. A transaction is allowed to access a data item only if it is currently holding a lock on that item.

Locks:

The two modes of locks are:

1. **Shared**. If a transaction *Ti* has obtained a **shared-mode lock** (denoted by S) on

item Q, then Ti can read, but cannot write, Q.

2. Exclusive. If a transaction Ti has obtained an **exclusive-mode lock** (denoted by X) on item Q, then Ti can both read and write Q.

We require that every transaction **request** a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q.

The transaction makes the request to the concurrency-control manager.

The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.

The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

To state this more generally, given a set of lock modes, we can define a **compatibility function** on them as follows:

Let A and B represent arbitrary lock modes. Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj ($Ti = Tj$) currently holds a lock of mode B.

If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is **compatible** with mode B. Such a function can be represented conveniently by a matrix.

	S	X
S	true	false
X	false	false

Figure Lock-compatibility matrix comp.

An element comp(A, B) of the matrix has the value *true* if and only if mode A is compatible with mode B.

A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction.

Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction. A transaction can unlock a data item Q by the unlock(Q) instruction.

To access a data item, transaction Ti must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.

Thus, Ti is made to **wait** until all incompatible locks held by other transactions have been released.

Transaction Ti may unlock a data item that it had locked at some earlier point.

Example:

Let A and B be two accounts that are accessed by transactions $T1$ and $T2$. Transaction $T1$ transfers \$50 from account B to account A

$T1$: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);

$A := A + 50$;
write(A);
unlock(A).

Transaction *T*1.

Transaction $T2$ displays the total amount of money in accounts A and B—that is, the sum $A + B$

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

Transaction T2.

Suppose that the values of accounts A and B are \$100 and \$200, respectively.

If these two transactions are executed serially, either in the order $T1$, $T2$ or the order $T2$, $T1$, then transaction $T2$ will display the value \$300.

If, however, these transactions are executed concurrently, then schedule 1is possible.

In this case, transaction $T2$ displays \$250, which is incorrect.

The reason for this mistake is that the transaction $T1$ unlocked data item B too early, as a result of which $T2$ saw an inconsistent state.

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Figure Schedule 1.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks.

The transaction making a lock request cannot execute its next action until the concurrency control

manager grants the lock.

Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction.

3.7 TWO PHASE LOCKING:

Two Phase Locking protocol requires that each transaction issue lock and unlock requests in two phases.

1. **Growing phase:** A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction aquires locks as needed. Once a transaction releases a lock, it enters in the shrinking phase, and it cannot issue more lock requests.

For example, transactions T_3 is two phase. But transactions T_1 and T_2 are not two phase.

T_3: lock-X (B);
read (B); B: = B-50; write (B); lock-X (A); read (A); A: = A=50; write (A); unlock (B); unlock (A);

Advantages:

The two-phase locking protocol ensures conflict serializability.

Consider any transaction, the point in the schedule where the transaction has obtained its final lock is called the **lock-point** of the transaction. Now, the transactions can be ordered according to their lock points. This ordering is the serializability ordering for the transactions.

Disadvantages

1. It does not ensure freedom from deadlock.

2. Cascading rollbacks may occur under two-phase locking.

Consider schedule 2

T_5	T_6	T_7
lock-X(A) read(A) lock-S(B) read(B) write(A) unlock(A)		
	lock-X(A) read(A) write(A)	

Fig.Partial schedule under two-phase locking

Here, the transactions T_5, T_6 and T_7 are two phase, but failure of T_5 after the read (A) instruction of T_7 leads to cascading rollback of T_6 and T_7.

Cascading rollbacks can be avoided by a modification of two-phase locking-called the **strict two-phase locking protocol**.

Types:

1. Strict two phase locking protocol

This protocol requires that locking should be two phase, and all exclusive-mode locks taken by a transaction should be held until the transaction. This requirement prevents any transaction from reading the data written by any uncommitted transaction under exclusive mode until the transaction commits, preventing any other transaction from reading the data.

2. Rigorous two phase locking protocol

This protocol requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.Most database systems implement either strict or rigorous two-phase locking.

Two-phase locking protocol allows lock conversions. There is a mechanism for upgrading a shared lock to an exclusive lock, downgrading an exclusive lock to a shared lock. We donote the conversion from shared to exclusive modes by upgrade, and from exclusive to shared by downgrade. Lock conversion cannot be allowed arbitrarily.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read (Q) operation, the system issues a lock-*S(Q)* instruction followed by the read (Q) instruction.
- When T_i issues a write (Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade (Q) instruction, followed by the write (Q) instruction. Otherwise, the system issues a lock - *X(Q)* instruction, followed by the write (Q) instruction.

- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

3.8 DEADLOCK:

Deadlock refers to a particular situation where two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource.

Some computers, usually those intended for the time-sharing and/or real-time markets, are often equipped with a hardware lock, or hard lock, which guarantees exclusive access to processes, forcing serialization.

Deadlocks are particularly disconcerting because there is no general solution to avoid them.

Livelock:

Livelock is a special case of resource starvation. A livelock is similar to a deadlock, except that the states of the processes involved constantly change with regard to one another wile never progressing.

The general definition only states that a specific process is not progressing.

For example, the system keeps selecting the same transaction for rollback causing the transaction to never finish executing.

Another livelock situation can come about when the system is deciding which transaction gets a lock and which waits in a conflict situation.

Deadlock Prevention:

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations which transactions are about to execute.

DBMS inspects operations and analyze if they can create a deadlock situation. If it finds that a deadlock situation might occur then that transaction is never allowed to be executed.

There are deadlock prevention schemes, which uses time-stamp ordering mechanism of transactions in order to pre-decide a deadlock situation.

Wait-Die Scheme:

In this scheme, if a transaction request to lock a resource (data item), which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

- If TS(Ti) < TS(Tj), that is Ti, which is requesting a conflicting lock, is older than Tj, Ti is allowed to wait until the data-item is available.

- If TS(Ti) > TS(tj), that is Ti is younger than Tj, Ti dies. Ti is restarted later with random delay but with same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

Wound-Wait Scheme:

In this scheme, if a transaction request to lock a resource (data item), which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

- If TS(Ti) < TS(Tj), that is Ti, which is requesting a conflicting lock, is older than Tj, Ti forces Tj to be rolled back, that is Ti wounds Tj. Tj is restarted later with random delay but with same timestamp.
- If TS(Ti) > TS(Tj), that is Ti is younger than Tj, Ti is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait but when an older transaction request an item held by younger one, the older transaction forces the younger one to abort and release the item. In both cases, transaction, which enters late in the system, is aborted.

Deadlock Avoidance:

Aborting a transaction is not always a practical approach. Instead deadlock avoidance mechanisms can be used to detect any deadlock situation in advance.

Methods like "wait-for graph" are available but for the system where transactions are light in weight and have hold on fewer instances of resource. In a bulky system deadlock prevention techniques may work well.

Wait-for Graph:

This is a simple method available to track if any deadlock situation may arise.

For each transaction entering in the system, a node is created.

When transaction Ti requests for a lock on item, say X, which is held by some other transaction Tj, a directed edge is created from Ti to Tj. If Tj releases item X, the edge between them is dropped and Ti locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. System keeps checking if there's any cycle in the graph.

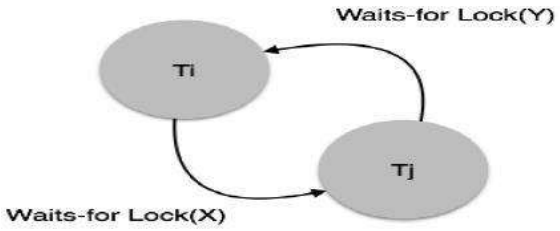


Fig Wait-for Graph

Two approaches can be used, first not to allow any request for an item, which is already locked by some other transaction.

This is not always feasible and may cause starvation, where a transaction indefinitely waits for data item and can never acquire it. Second option is to roll back one of the transactions.

It is not feasible to always roll back the younger transaction, as it may be important than the older one.

With help of some relative algorithm a transaction is chosen, which is to be aborted, this transaction is called victim and the process is known as victim selection.

3.9 TRANSACTION RECOVERY:

- Protocols that obey this are referred to as **non-blocking** protocols. In the following two sections, we consider two common commit protocols suitable for distributed DBMSs: two-phase commit (2PC) and three-phase commit (3PC), a non-blocking protocol.
- Assume that every global transaction has one site that acts as **coordinator** (or **transaction manager**) for that transaction, which is generally the site at which the transaction was initiated. Sites at which the global transaction has agents are called **participants** (or **resource managers**).
- Assume that the coordinator knows the identity of all participants and that each participant knows the identity of the coordinator but not necessarily of the other participants.

Two-Phase Commit (2PC)

- 2PC operates in two phases: a **voting phase** and a **decision phase**.
- The basic idea is that the coordinator asks all participants whether they are prepared to commit the transaction. If one participant votes to abort, or fails to respond within a timeout

period, then the coordinator instructs all participants to abort the transaction.

- If all vote to commit, then the coordinator instructs all participants to commit the transaction. The global decision must be adopted by all participants.
- If a participant votes to abort, then it is free to abort the transaction immediately; in fact, any site is free to abort a transaction at any time up until it votes to commit. This type of abort is known as a **unilateral abort**.
- If a participant votes to commit, then it must wait for the coordinator to broadcast either the *global commit* or *global abort* message.
- This protocol assumes that each site has its own local log, and can therefore rollback or commit the transaction reliably. Two-phase commit involves processes waiting for messages from other sites. To avoid processes being blocked unnecessarily, a system of timeouts is used. The procedure for the coordinator at commit is as follows:

Phase 1

(1) Write a *begin_commit* record to the log file and force-write it to stable storage.
- ✓ Send a PREPARE message to all participants.
- ✓ Wait for participants to respond within a timeout period.

Phase 2

(2) If a participant returns an ABORT vote,
- ✓ Write an abort record to the log file and force write it to stable storage.
- ✓ Send a GLOBAL_ABORT message to all participants.
- ✓ Wait for participants to acknowledge within a timeout period.

(3) If a participant returns a READY_COMMIT vote,
- ✓ Write a commit record to the log file and force-write it to stable storage.
- ✓ Send a GLOBAL_COMMIT message to all participants.
- ✓ Wait for participants to acknowledge within a timeout period.

(4) Once all acknowledgements have been received,
- ✓ Write an end_transaction message to the log file.

2PC Protocol for voting Commit:

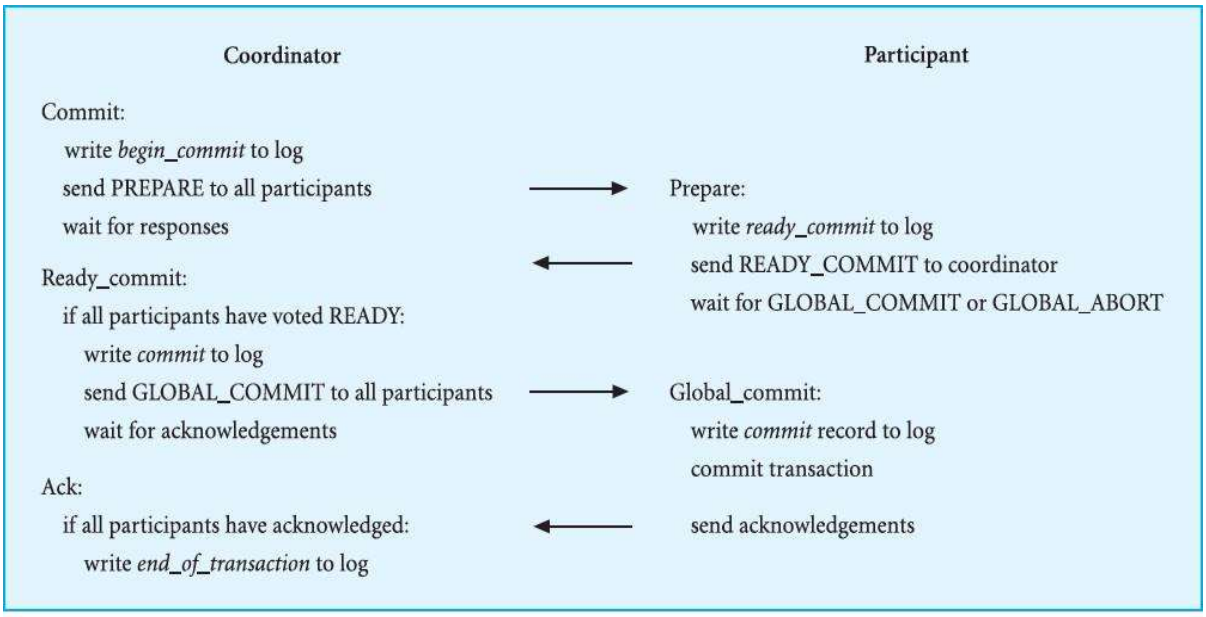


Fig 2 PC Protocol for voting Commit

2PC Protocol for voting Abort:

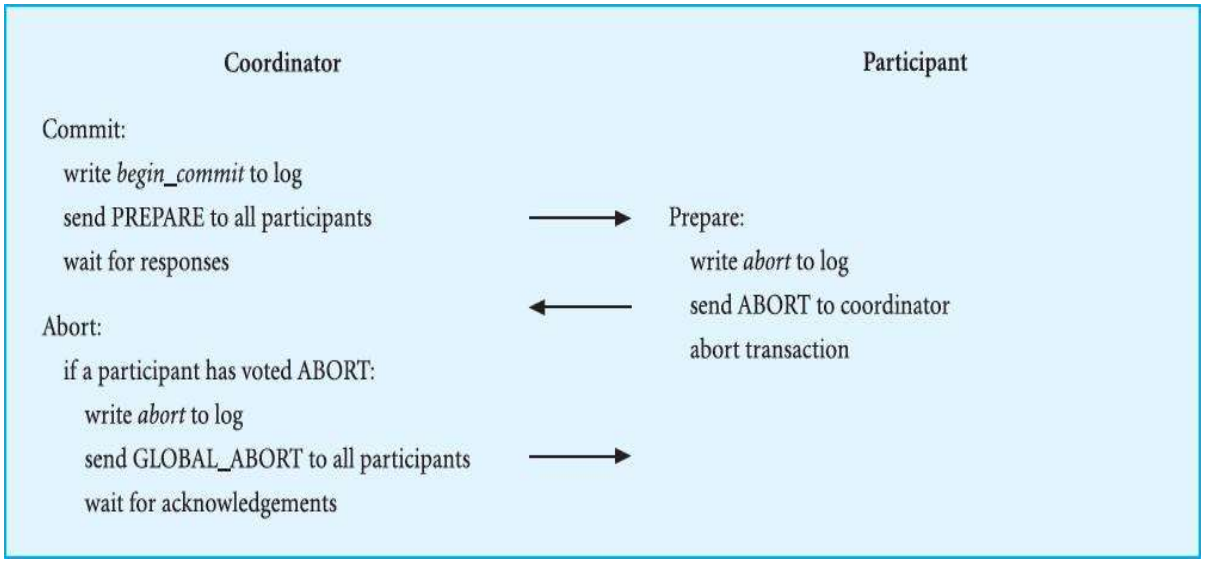


Fig. 2PC Protocol for voting Abort

Termination protocols for 2PC

A termination protocol is invoked whenever a coordinator or participant fails to receive an expected message and times out. The action to be taken depends on whether the coordinator or participant has timed out and on when the timeout occurred.

(i) **Coordinator**

The coordinator can be in one of four states during the commit process:

- ❖ INITIAL
- ❖ WAITING
- ❖ DECIDED
- ❖ COMPLETED

as shown in the state transition diagram in Figure, but can time out only in the middle two states. The actions to be taken are as follows:

Timeout in the WAITING state -The coordinator is waiting for all participants to acknowledge whether they wish to commit or abort the transaction. In this case, the coordinator cannot commit the transaction because it has not received all votes. However, it can decide to globally abort the transaction.

Timeout in the DECIDED state -The coordinator is waiting for all participants to acknowledge whether they have successfully aborted or committed the transaction. In this case, the coordinator simply sends the global decision again to sites that have not acknowledged.

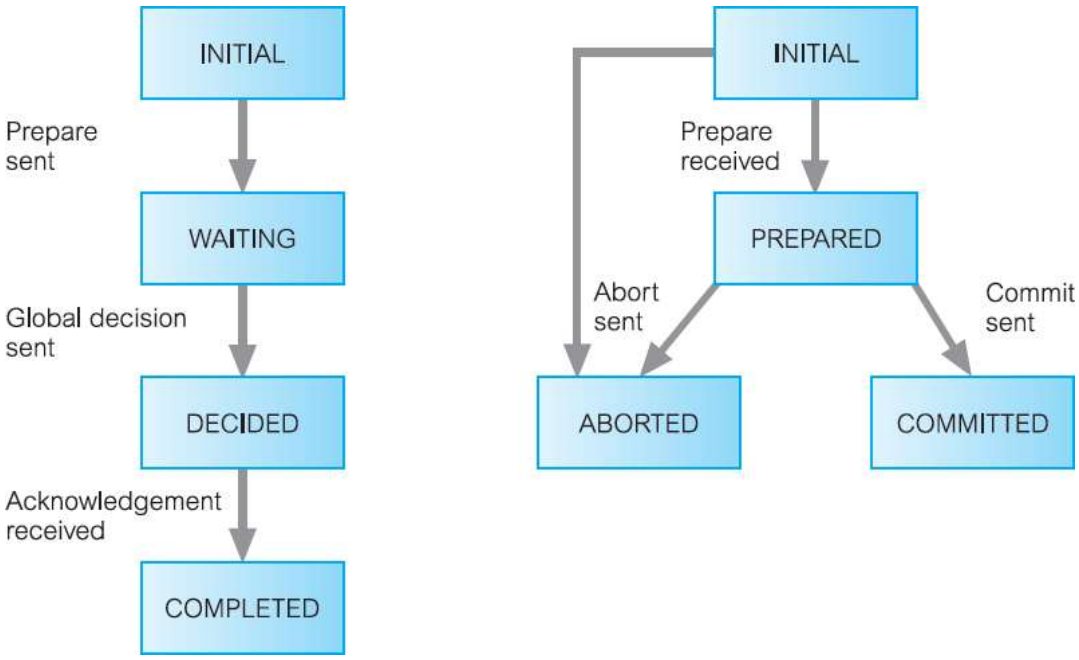


Fig.Termination protocols for 2PC

(ii) **Participant**

A participant can be in one of four states during the commit process:

- ❖ INITIAL
- ❖ PREPARED
- ❖ ABORTED
- ❖ COMMITTED

as shown in the state transition diagram in Figure. However, a participant may time out only in the first two states as follows:

Timeout in the INITIAL state-The participant is waiting for a PREPARE message from the coordinator, which implies that the coordinator must have failed while in the INITIAL state. In this case, the participant can unilaterally abort the transaction. If it subsequently receives a PREPARE message, it can either ignore it, in which case the coordinator times out and aborts the global transaction, or it can send an ABORT message to the coordinator.

Timeout in the PREPARED state-The participant is waiting for an instruction to globally commit or abort the transaction. The participant must have voted to commit the transaction, so it cannot change its vote and abort the transaction. Equally well, it cannot go ahead and commit the transaction, as the global decision may be to abort.

Recovery protocols for 2PC

(i) **Coordinator failure**

Consider three different stages for failure of the coordinator:

- ***Failure in INITIAL state***-The coordinator has not yet started the commit procedure. Recovery in this case starts the commit procedure.
- ***Failure in WAITING state***-The coordinator has sent the PREPARE message and although it has not received all responses, it has not received an abort response. In this case, recovery restarts the commit procedure.
- ***Failure in DECIDED state***-The coordinator has instructed the participants to globally abort or commit the transaction. On restart, if the coordinator has received all acknowledgements, it can complete successfully.

(ii) **Participant failure**

Consider three different stages for failure of a participant:

- ***Failure in INITIAL state***-The participant has not yet voted on the transaction. Therefore, on recovery it can unilaterally abort the transaction, as it would have been impossible for the coordinator to have reached a global commit decision without this participant's vote.
- ***Failure in PREPARED state***-The participant has sent its vote to the coordinator. In this case, recovery is via the termination protocol discussed above.
- ***Failure in ABORTED/COMMITTED states***-The participant has completed the transaction. Therefore, on restart, no further action is necessary.

3.10 SAVE POINTS:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for a SAVEPOINT command is as shown below.

SAVEPOINT SAVEPOINT_NAME;

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as shown below.

ROLLBACK TO SAVEPOINT_NAME;

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

Example

Consider the CUSTOMERS table having the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Suresh	23	Chennai	5400
2	Mano	22	Cochin	6500
3	Subhiksha	23	Newyork	4500
4	Malar	24	Bangalore	4300
5	Sarath	22	Trichy	7500
6	Krishna	32	Coimbatore	6600
7	Ranchana	21	Hyderabad	5500

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;

1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone −

```
SQL> ROLLBACK TO SP2;

Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2.

ID	NAME	AGE	ADDRESS	SALARY
2	Mano	22	Cochin	6500
3	Subhiksha	23	Newyork	4500
4	Malar	24	Bangalore	4300
5	Sarath	22	Trichy	7500
6	Krishna	32	Coimbatore	6600

7	Ranchana	21	Hyderabad	5500

6 rows selected

RELEASE SAVEPOINT Command

RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

RELEASE SAVEPOINT SAVEPOINT_NAME;

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

3.11 ISOLATION LEVELS:

Isolation levels determine the type of phenomena that can occur during the execution of concurrent transactions. Developer sets this property only for the following databases: MSSQL, Informix and DB2.

Three phenomena define SQL Isolation levels for a transaction:

Dirty Reads returns different results within a single transaction when an SQL operation an uncommitted or modified record created by another transaction. Dirty Reads increases concurrency, but reduces consistency.

Non-Repeatable Reads returns different results within a single transaction when an SQL operation reads the same row in a table twice. Non-Repeatable Reads can occur when another transaction modifies and commits a change to the row between transaction reads. Non-repeatable reads increases consistency, but reduces concurrency.

Phantoms returns different results within a single transaction when an SQL operation retrieves a range of data values twice. Phantoms can occur if another transaction inserted a new record and committed the insertion between executions of the range retrieval.

Each Isolation level differs in the phenomena it allows:

Transaction isolation level	Dirty reads	Nonrepeatable reads	Phantoms
Read uncommitted	X	X	X
Read committed	--	X	X
Repeatable read	--	--	X
Serializable	--	--	--

- **Read Uncommitted** – Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet commited changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

- **Read Committed –** This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction hold a read or write lock on the current row, and thus prevent other rows from reading, updating or deleting it.

- **Repeatable Read –** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and write locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non repeatable read.
- **Serializable –** This is the Highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

3.12 SQL FACILITIES FOR CONCURRENCY AND RECOVERY:

Crash Recovery

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

Failure Classification

To see where the problem has occurred, we generalize a failure into various categories, as follows −

a) Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be −

- **Logical errors** − Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors** − Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

System Crash

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently. Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories −

- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile storage** − These memories are made to survive system crashes. They are huge in data storage capacity, but slower in accessibility. Examples may include hard-disks, magnetic tapes, flash memory, and non-volatile (battery backed up) RAM.

Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows −

- ✓ The log file is kept on a stable storage media.
- ✓ When a transaction enters the system and starts execution, it writes a log about it.
- ✓ **<Tn, Start>**
- ✓ When the transaction modifies an item X, it write logs as follows −
- ✓ **<Tn, X, V1, V2>**
- ✓ It reads T_n has changed the value of X, from V_1 to V_2.
- ✓ When the transaction finishes, it logs −
- ✓ **<Tn, commit>**

The database can be modified using two approaches −

- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Recovery with Concurrent Transactions

When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

1. Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

2. Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −

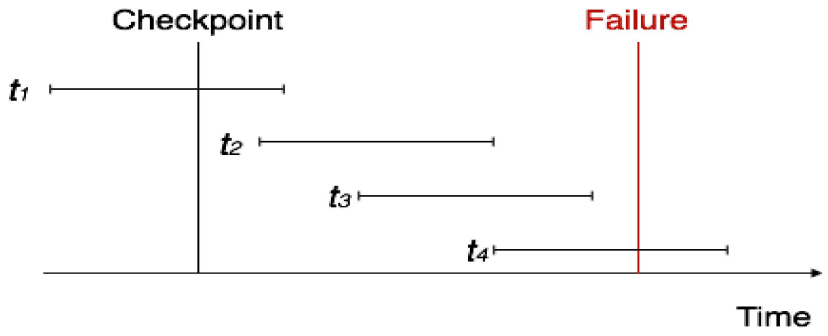


Fig 5.15 Checkpoint versus Failure

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

SQL Commands: Constraints Used In Database

Constraints are used in a database to specify the rules for data in a table. The following are the different types of constraints:

- NOT NULL
- UNIQUE

- INDEX

NOT NULL

This constraint ensures that a column cannot have a NULL value.

Example

```
--NOT NULL on Create Table

CREATE TABLE Employee_Info
(
EmployeeID int NOT NULL,
EmployeeName varchar(255) NOT NULL,
Emergency ContactName varchar(255),
PhoneNumber int NOT NULL,
Address varchar(255),
City varchar(255),
Country varchar(255)
);

--NOT NULL on ALTER TABLE

ALTER TABLE Employee_Info
MODIFY PhoneNumber int NOT NULL;
```

UNIQUE

This constraint ensures that all the values in a column are unique.

Example

```
--UNIQUE on Create Table

CREATE TABLE Employee_Info
(
EmployeeID int NOT NULL UNIQUE,
EmployeeName varchar(255) NOT NULL,
Emergency ContactName varchar(255),
PhoneNumber int NOT NULL,
Address varchar(255),
City varchar(255),
Country varchar(255)
);

--UNIQUE on Multiple Columns

CREATE TABLE Employee_Info
(
EmployeeID int NOT NULL,
EmployeeName varchar(255) NOT NULL,
Emergency ContactName varchar(255),
PhoneNumber int NOT NULL,
Address varchar(255),
City varchar(255),
```

```
27
28      --UNIQUE on ALTER TABLE
29
30      ALTER TABLE Employee_Info
31      ADD UNIQUE (Employee_ID);
32
33      --To drop a UNIQUE constraint
34
35      ALTER TABLE  Employee_Info
36      DROP CONSTRAINT UC_Employee_Info;
```

CHECK

This constraint ensures that all the values in a column satisfy a specific condition.

Example

```
1       --CHECK Constraint on CREATE TABLE
2
3       CREATE TABLE Employee_Info
4       (
5       EmployeeID int NOT NULL,
6       EmployeeName varchar(255),
7       Emergency ContactName varchar(255),
8       PhoneNumber int,
9       Address varchar(255),
10      City varchar(255),
11      Country varchar(255) CHECK (Country=='India')
12      );
13
14      --CHECK Constraint on multiple columns
15
16      CREATE TABLE Employee_Info
17      (
18      EmployeeID int NOT NULL,
19      EmployeeName varchar(255),
20      Emergency ContactName varchar(255),
21      PhoneNumber int,
22      Address varchar(255),
23      City varchar(255),
24      Country varchar(255) CHECK (Country = 'India' AND Cite = 'Hyderabad')
25      );
26
27      --CHECK Constraint on ALTER TABLE
28
29      ALTER TABLE Employee_Info
30      ADD CHECK (Country=='India');
31
32      --To give a name to the CHECK Constraint
33
34      ALTER TABLE Employee_Info
35      ADD CONSTRAINT CheckConstraintName CHECK (Country=='India');
36
37      --To drop a CHECK Constraint
38
39      ALTER TABLE Employee_Info
```

DEFAULT

This constraint consists of a set of default values for a column when no value is specified.

Example

```
1       --DEFAULT Constraint on CREATE TABLE
2
3       CREATE TABLE Employee_Info
4       (
5       EmployeeID int NOT NULL,
6       EmployeeName varchar(255),
7       Emergency ContactName varchar(255),
8       PhoneNumber int,
9       Address varchar(255),
10      City varchar(255),
11      Country varchar(255) DEFAULT 'India'
12      );
13
14      --DEFAULT Constraint on ALTER TABLE
15
16      ALTER TABLE Employee_Info
17      ADD CONSTRAINT defau_Country
18      DEFAULT 'India' FOR Country;
19
20      --To drop the Default Constraint
21
22      ALTER TABLE Employee_Info
23      ALTER COLUMN Country DROP DEFAULT;
```

INDEX

This constraint is used to create indexes in the table, through which you can create and retrieve data from the database very quickly.

Syntax

```
--Create an Index where duplicate values are allowed
CREATE INDEX IndexName
ON TableName (Column1, Column2, ...ColumnN);

--Create an Index where duplicate values are not allowed
CREATE UNIQUE INDEX IndexName
ON TableName (Column1, Column2, ...ColumnN);
```

Example

```
1       CREATE INDEX idex_EmployeeName
2       ON Persons (EmployeeName);
3
4       --To delete an index in a table
5
6       DROP INDEX Employee_Info.idex_EmployeeName;
```

DCL Commands in SQL

This section of the article will give you an insight into the commands which aDMLre used to enforce database security in multiple user database environments. The commands are as follows:

- REVOKE

GRANT

This command is used to provide access or privileges on the database and its objects to the users.

Syntax
GRANT PrivilegeName
ON ObjectName
TO {UserName |PUBLIC |RoleName}
[WITH GRANT OPTION];
where,

- **PrivilegeName –** Is the privilege/right/access granted to the user.
- **ObjectName –** Name of a database object like TABLE/VIEW/STORED PROC.
- **UserName –** Name of the user who is given the access/rights/privileges.
- **PUBLIC –** To grant access rights to all users.
- **RoleName –** The name of a set of privileges grouped together.
- **WITH GRANT OPTION –** To give the user access to grant other users with rights.

Example

```
1  -- To grant SELECT permission to Employee_Info table to user1
2  GRANT SELECT ON Employee_Info TO user1;
```

REVOKE

This command is used to withdraw the user's access privileges given by using the GRANT command.

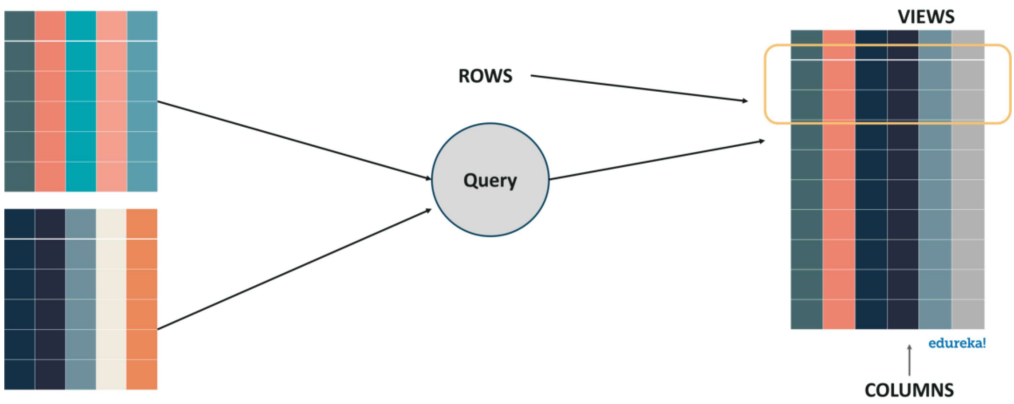
Syntax
REVOKE PrivilegeName
ON ObjectName
FROM {UserName |PUBLIC |RoleName}
Example

```
1  -- To revoke the granted permission from user1
2  REVOKE SELECT ON Employee_Info TO user1;
```

Now, next in this article on SQL Commands, I will discuss Views, Stored Procedures, and Triggers.

Views

A view in SQL is a single table, which is derived from other tables. So, a view contains rows and columns similar to a real table and has fields from one or more table.



The 'CREATE VIEW' statement

This statement is used to create a view, from a table.

Syntax

```
CREATE VIEW ViewName AS
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition;
```

Example

```
1    CREATE VIEW [Kolkata Employees] AS
2    SELECT EmployeeName, PhoneNumber
3    FROM Employee_Info
4    WHERE City = "Kolkata";
```

The 'CREATE OR REPLACE VIEW' statement

This statement is used to update a view.

Syntax

```
CREATE VIEW OR REPLACE ViewName AS
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition;
```

Example

```
1    CREATE VIEW OR REPLACE [Kolkata Employees] AS
2    SELECT EmployeeName, PhoneNumber
3    FROM Employee_Info
4    WHERE City = "Kolkata";
```

The 'DROP VIEW' statement

This statement is used to delete a view.

Syntax

```
DROP VIEW ViewName;
```

Example

```
1  DROP VIEW [Kolkata Employees];
```

Stored Procedures

A code which you can save and reuse it again is known as StoredProcedures.

Syntax

```
CREATE PROCEDURE ProcedureName
AS
SQLStatement
GO;
```

Example

```
EXEC ProcedureName;
```

TCL Commands (SavePoint, RollBack, and Commit)

In SQL, SAVEPOINT, ROLLBACK, and COMMIT are essential components of Transaction Control Language (TCL).

TCL in SQL helps in managing transactions within a database effectively.

SQL SAVEPOINT

The SAVEPOINT command in SQL allows us to set a point within a transaction to which we can roll back without affecting the entire transaction.

This is particularly useful in managing long or complex transactions.

Consider a scenario where we are updating records in the Customers table and want to create a savepoint after each update. Here's how we do it:

```
-- start transaction
BEGIN TRANSACTION;

-- update customer 1's age
UPDATE Customers SET age = 32 WHERE customer_id = 1;

-- create a savepoint named SP1
SAVEPOINT SP1;

-- update customer 2's country
UPDATE Customers SET country = 'Canada' WHERE customer_id = 2;

-- create a savepoint named SP2
SAVEPOINT SP2;

-- If an error occurs, we can rollback to SP1 or SP2
```

Here, SAVEPOINT SP1 and SAVEPOINT SP2 allow us to roll back to those points without undoing all previous changes.

ROLLBACK in SQL

Imagine a scenario where we need to delete a record from the Orders table but decide to rollback the transaction.

```
-- start transaction
BEGIN TRANSACTION;

-- delete order with order_id 5
DELETE FROM Orders WHERE order_id = 5;

-- after some operations, decide to rollback
ROLLBACK;
```

Here, the ROLLBACK statement here undoes the deletion of the order.

COMMIT in SQL

The COMMIT command in SQL is used to save all changes made during the current transaction permanently.

Let's update a record in the Shippings table and then commit the transaction. Here's the query:

```
-- start transaction
BEGIN TRANSACTION;

-- update the status of shipping_id 1
UPDATE Shippings SET status = 'Shipped' WHERE shipping_id = 1;

-- commit the transaction
COMMIT;
```