

UNIT-5 Transaction Concept

Transaction Concept: Transaction State, ACID properties, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, lockbased, time stamp based ,optimistic, concurrency protocols ,Deadlocks, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm.

Introduction to Indexing Techniques: B+ Trees, operations on B+ Trees, Hash Based Indexing.

Transaction: Transaction in DBMS is a set of logically related operations executed as a single unit. These logic are followed to perform modification on data while maintaining integrity and consistency.

Transactions are performed in a way that concurrent actions from different users don't malfunction the database.

Transfer of money from one account to another in a bank management system is the best example of Transaction.

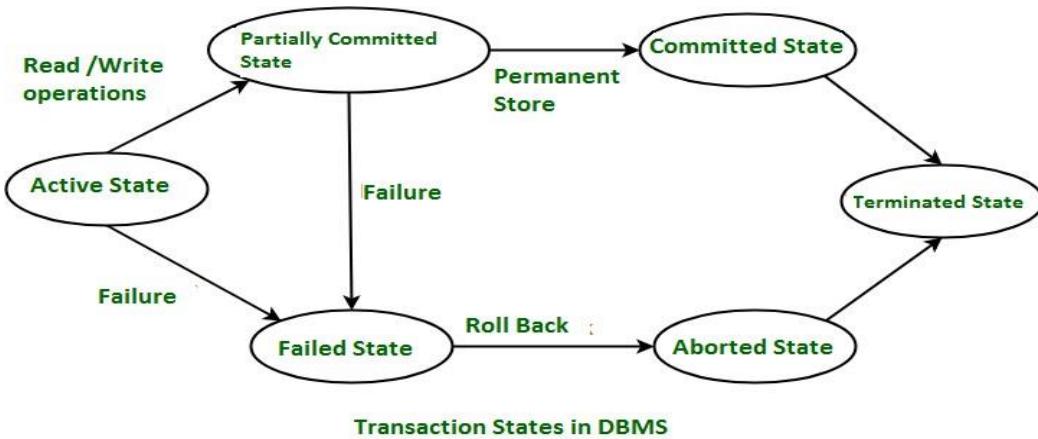
1. R(A);
2. A=A-1000;
3. W(A);
4. R(B);
5. B=B+1000;
6. W(B);

The main operations of a transaction are:

- **Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in the main memory.
- **Write (A):** Write operation Write(A) or W(A) writes the value back to the database from the buffer.

Transaction State:

A Transaction log is a file maintained by the recovery management component to record all the activities of the transaction. After the commit is done transaction log file is removed.



In DBMS, a transaction passes through various states such as active, partially committed, failed, and aborted.

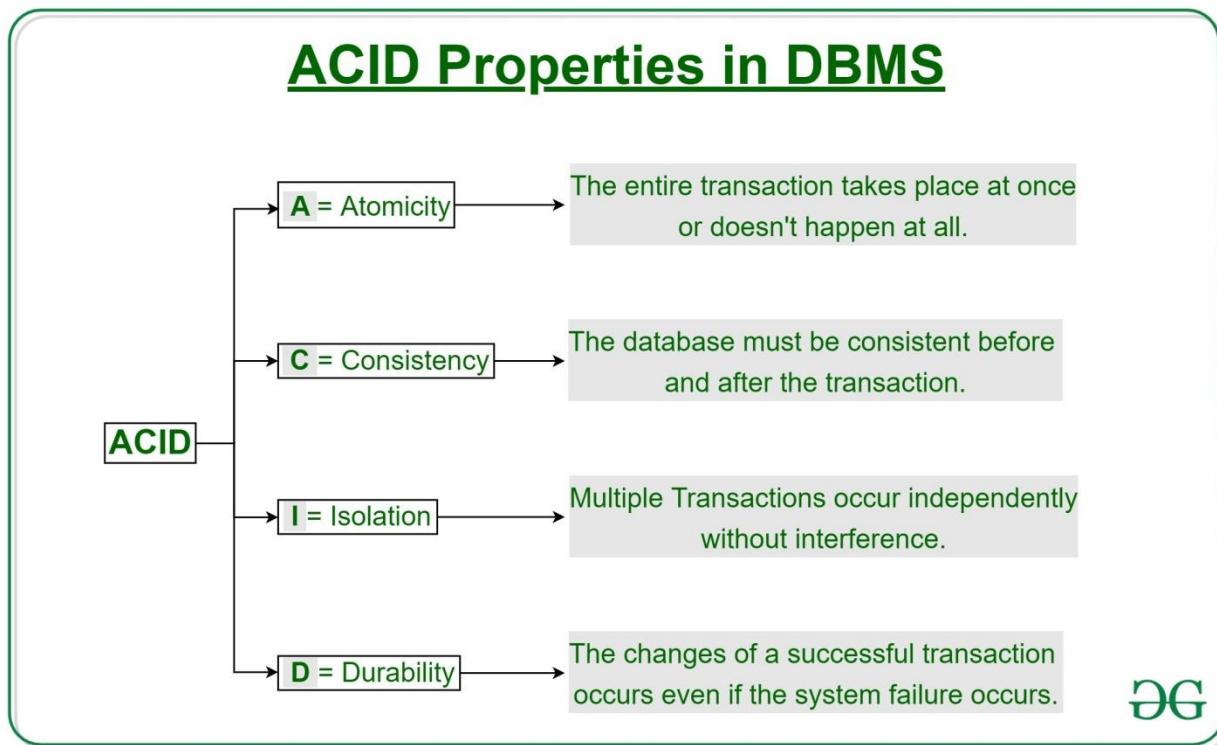
These are different types of Transaction States :

- 1. 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”.
- 2. 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”.
- 3. 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 Database.
- 4. 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.
- 5. 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”.

6. 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.

ACID properties:

A Transaction is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read-and-write operations. To maintain consistency in a database, before and after the transaction, certain properties are followed. These are called properties.



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 the database are not visible.

— Commit : If a transaction commits, changes made are visible.
Atomicity is also known as the ‘All or nothing rule’.

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 = $400 + 300 = 700$.

Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference.

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.

Concurrent Executions:

In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database.

Concurrent execution allows many transactions to access the same data concurrently, which can have numerous benefits, such as higher system throughput and reaction time.

Concurrent execution might offer a number of issues that need to be resolved in order to guarantee accurate and dependable database operation.

Some of the issues with concurrent execution in DBMS include the following.

Lost Update Problem(W - W Conflict)

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

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

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

LOST UPDATE PROBLEM

At time t_6 , transaction T_x writes the value of account A that will be updated as \$250 only, as T_y didn't update the value yet.

Similarly, at time t_7 , transaction T_y writes the values of account A, so it will write as done at time t_4 that will be \$400. It means the value written by T_x is lost, i.e., \$250 is lost.

Dirty Read Problems (W-R Conflict)

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

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

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

DIRTY READ PROBLEM

- At time t_3 , transaction T_x writes the updated value in account A, i.e., \$350.
- Then at time t_4 , transaction T_y reads account A that will be read as \$350.
- Then at time t_5 , transaction T_x rollbacks due to server problem, and the value changes back to \$300 (as initially).
- But the value for account A remains \$350 for transaction T_y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict)

The unrepeatable problem occurs when two or more read operations of the same transaction read different values of the same variable.

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

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

UNREPEATABLE READ PROBLEM

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

Phantom Read

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.

| T1 | T2 |
|--------------------------|------------------------|
| Read(X) Delete(X) | Read(X) Read(X) |

In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction 2's knowledge. Thus, when transaction 2 tries to read X, it is not able to do it.

Deadlock

In a DBMS, a deadlock happens when many transactions are held up as the wait for one another to release the resources they are holding. Deadlocks can happen when resources are not released properly or are acquired by system performance or even system crashes.

Starvation

In a DBMS, starvation happens when one transaction is perpetually blocked from using a resource or finishing a job because that resource has been allocated to another transaction. When resources are not equitably distributed among transactions or when priorities are not correctly controlled, starvation may result.

Serializability:

Serializability is a crucial concept in database management systems, particularly in the context of concurrent transaction processing. It refers to the property of a schedule (a sequence of operations from multiple transactions) that ensures the outcome of executing concurrent transactions is equivalent to some serial execution of those transactions. In other words, even though transactions may run concurrently, their combined effect should be the same as if they were executed one after another in some order.

Ex:

Types of Serializability

1. **Conflict Serializability:** A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are considered conflicting if:

- They belong to different transactions.
- They access the same data item.
- At least one of them is a write operation.

If a schedule can be rearranged without affecting the results, it is conflict serializable.

Ex:

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

Before swapping

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

After swapping

2. View Serializability: A schedule is view serializable if it produces the same final state as a serial schedule. It is based on the concept of views rather than the specific order of operations.

- o View serializability considers three main aspects:
 - The initial reads of each transaction.
 - The final writes of each transaction.
 - The reads of data items in between.

If a schedule can be rearranged to match the views of a serial execution, it is considered view serializable.

Ex:

| T1 | T2 |
|---------|----------|
| Read(A) | Write(A) |

Schedule S1

| T1 | T2 |
|---------|----------|
| Read(A) | Write(A) |

Schedule S2

Recoverability:

Recoverability is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state. Recoverability guarantees that all committed transactions are durable and that their effects are permanently stored in the database, while the effects of uncommitted transactions are undone to maintain data consistency.

There are several levels of recoverability that can be supported by a database system:

No-undo logging: This level of recoverability only guarantees that committed transactions are durable, but does not provide the ability to undo the effects of uncommitted transactions.

Undo logging: This level of recoverability provides the ability to undo the effects of uncommitted transactions but may result in the loss of updates made by committed transactions that occur after the failed transaction.

Redo logging: This level of recoverability provides the ability to redo the effects of committed transactions, ensuring that all committed updates are durable and can be recovered in the event of failure.

Undo-redo logging: This level of recoverability provides both undo and redo capabilities, ensuring that the system can recover to a consistent state regardless of whether a transaction has been committed or not.

Types of Recoverability of Schedules:

1.Recoverable Schedules:

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules.

Example: Consider the following schedule involving two transactions T₁ and T₂.

| T ₁ | T ₂ |
|----------------|----------------|
| R(A) | |
| W(A) | |
| | W(A) |
| | R(A) |
| commit | |
| | commit |

This is a recoverable schedule since T₁ commits before T₂, that makes the value read by T₂ correct.

2.Irrecoverable Schedule: The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that. So this schedule is irrecoverable schedule. When Tj is reading the value updated by Ti and Tj is committed before committing of Ti, the schedule will be irrecoverable.

| T1 | T1's buffer space | T2 | T2's Buffer Space | Database |
|---------------|-------------------|----------|-------------------|----------|
| | | | | A=5000 |
| R(A); | A=5000 | | | A=5000 |
| A=A-100; | A=4000 | | | A=5000 |
| W(A); | A=4000 | | | A=4000 |
| | | R(A); | A=4000 | A=4000 |
| | | A=A+500; | A=4500 | A=4000 |
| | | W(A); | A=4500 | A=4500 |
| | | Commit; | | |
| Failure Point | | | | |
| Commit; | | | | |

3.Recoverable with Cascading Rollback: The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rolled back. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.

| T1 | T1's buffer space | T2 | T2's Buffer Space | Database |
|---------------|-------------------|----------|-------------------|----------|
| | | | | A=5000 |
| R(A); | A=5000 | | | A=5000 |
| A=A-100; | A=4000 | | | A=5000 |
| W(A); | A=4000 | | | A=4000 |
| | | R(A); | A=4000 | A=4000 |
| | | A=A+500; | A=4500 | A=4000 |
| | | W(A); | A=4500 | A=4500 |
| Failure Point | | | | |
| Commit; | | | | |
| | | Commit; | | |

4.Cascadeless Recoverable Rollback: The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.

| T1 | T1's buffer space | T2 | T2's Buffer Space | Database |
|----------|-------------------|----------|-------------------|----------|
| | | | | A=5000 |
| R(A); | A=5000 | | | A=5000 |
| A=A-100; | A=4000 | | | A=5000 |
| W(A); | A=4000 | | | A=4000 |
| Commit; | | | | |
| | | R(A); | A=4000 | A=4000 |
| | | A=A+500; | A=4500 | A=4000 |
| | | W(A); | A=4500 | A=4500 |
| | | Commit; | | |

5.Strict Schedule

A schedule is strict if it is both recoverable and cascades. If the schedule contains no read or write before commit then it is known as a strict schedule. A strict schedule is strict in nature.

Implementation of Isolation:

The four standard isolation levels are:

1. **Read Uncommitted:** This is the lowest level of isolation where a transaction can see uncommitted changes made by other transactions. This can result in dirty reads, non-repeatable reads, and phantom reads.
2. **Read Committed:** In this isolation level, a transaction can only see changes made by other committed transactions. This eliminates dirty reads but can still result in non-repeatable reads and phantom reads.
3. **Repeatable Read:** This isolation level guarantees that a transaction will see the same data throughout its duration, even if other transactions commit changes to the data. However, phantom reads are still possible.
4. **Serializable:** This is the highest isolation level where a transaction is executed as if it were the only transaction in the system. All transactions must be executed sequentially, which ensures that there are no dirty reads, non-repeatable reads, or phantom reads.

Implementation of Isolation

Implementing isolation typically involves concurrency control mechanisms. Here are common mechanisms used:

1. Locking Mechanisms:

Locking ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

- **Shared Lock (S-lock):** Allows a transaction to read an item but not write to it.
- **Exclusive Lock (X-lock):** Allows a transaction to read and write an item. No other transaction can read or write until the lock is released.
- **Two-phase Locking (2PL):** This protocol ensures that a transaction acquires all the locks before it releases any. This results in a growing phase (acquiring locks and not releasing any) and a shrinking phase (releasing locks and not acquiring any).

2. Timestamp-based Protocols:

Every transaction is assigned a unique timestamp when it starts. This timestamp determines the order of transactions. Transactions can only access the database if they respect the timestamp order, ensuring older transactions get priority.

Testing for 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 $T_i \rightarrow T_j$ for which one of the three conditions holds:

1. Create a node $T_i \rightarrow T_j$ if T_i executes write (Q) before T_j executes read (Q).
2. Create a node $T_i \rightarrow T_j$ if T_i executes read (Q) before T_j executes write (Q).
3. Create a node $T_i \rightarrow T_j$ if T_i executes write (Q) before T_j executes write (Q).

Precedence graph for Schedule S



- If a precedence graph contains a single edge $T_i \rightarrow T_j$, then all the instructions of T_i are executed before the first instruction of T_j 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.

For example:

| | T1 | T2 | T3 |
|-------------|-------------------------|-------------------------|-------------------------|
| Time | Read(A) | Read(B) | Read(C) |
| | A := f ₁ (A) | B := f ₂ (B) | C := f ₃ (C) |
| | Write(A) | Write(B) | Write(C) |
| | Read(C) | Read(A) | Read(B) |
| | C := f ₅ (C) | A := f ₄ (A) | B := f ₆ (B) |
| | Write(C) | Write(A) | Write(B) |

Schedule S1

Explanation:

- Read(A):** In T1, no subsequent writes to A, so no new edges
- Read(B):** In T2, no subsequent writes to B, so no new edges
- Read(C):** In T3, no subsequent writes to C, so no new edges
- Write(B):** B is subsequently read by T3, so add edge T2 → T3
- Write(C):** C is subsequently read by T1, so add edge T3 → T1

Write(A): A is subsequently read by T2, so add edge T1 → T2

Write(A): In T2, no subsequent reads to A, so no new edges

Write(C): In T1, no subsequent reads to C, so no new edges

Write(B): In T3, no subsequent reads to B, so no new edges

Lock based:

In locked based protocol, each transaction needs to acquire locks before they start accessing or modifying the data items. There are two types of locks used in databases.

- **Shared Lock :** Shared lock is also known as read lock which allows multiple transactions to read the data simultaneously. The transaction which is holding a shared lock can only read the data item but it cannot modify the data item.
- **Exclusive Lock :** Exclusive lock is also known as the write lock. Exclusive lock allows a transaction to update a data item. Only one transaction can hold the exclusive lock on a data item at a time. While a transaction is holding an exclusive lock on a data item, no other transaction is allowed to acquire a shared/exclusive lock on the same data item.

There are two kind of lock based protocol mostly used in database:

Two Phase Locking Protocol : Two phase locking is a widely used technique which ensures strict ordering of lock acquisition and release. Two phase locking protocol works in two phases.

1.Growing Phase : In this phase, the transaction starts acquiring locks before performing any modification on the data items. Once a transaction acquires a lock, that lock cannot be released until the transaction reaches the end of the execution.

2.Shrinking Phase : In this phase, the transaction releases all the acquired locks once it performs all the modifications on the data item. Once the transaction starts releasing the locks, it cannot acquire any locks further.

Strict Two Phase Locking Protocol : It is almost similar to the two phase locking protocol the only difference is that in two phase locking the transaction can release its locks before it commits, but in case of strict two phase locking the transactions are only allowed to release the locks only when they performs commits.

Time stamp based:

- In this protocol each transaction has a timestamp attached to it. Timestamp is nothing but the time in which a transaction enters into the system.

- The conflicting pairs of operations can be resolved by the timestamp ordering protocol through the utilization of the timestamp values of the transactions. Therefore, guaranteeing that the transactions take place in the correct order.

Optimistic:

Validation Based Protocol is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs, therefore, there is no need for checking while the transaction is executed.

In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database. All updates are applied to local copies of data items kept for the transaction. At the end of transaction execution, while execution of the transaction, a validation phase checks whether any of transaction updates violate serializability.

If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is updated and then restarted.

Optimistic Concurrency Control is a three-phase protocol. The three phases for validation control protocol are:

1. **Read Phase:** Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.
2. **Validation Phase:** Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to the database.
3. **Write Phase:** On the success of the validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

In order to perform the Validation test, each transaction should go through the various phases as described above. Then, we must know about the following three time-stamps that we assigned to transaction T_i , to check its validity:

1. **Start(T_i):** It is the time when T_i started its execution.
2. **Validation(T_i):** It is the time when T_i just finished its read phase and begin its validation phase.
3. **Finish(T_i):** the time when T_i end it's all writing operations in the database under write-phase.

Two more terms that we need to know are:

1. Write_set: of a transaction contains all the write operations that T_i performs.

2. Read_set: of a transaction contains all the read operations that T_i performs.

Concurrency protocols:

Locked based Protocol:

In locked based protocol, each transaction needs to acquire locks before they start accessing or modifying the data items. There are two types of locks used in databases.

- **Shared Lock :** Shared lock is also known as read lock which allows multiple transactions to read the data simultaneously. The transaction which is holding a shared lock can only read the data item but it cannot modify the data item.
- **Exclusive Lock :** Exclusive lock is also known as the write lock. Exclusive lock allows a transaction to update a data item. Only one transaction can hold the exclusive lock on a data item at a time. While a transaction is holding an exclusive lock on a data item, no other transaction is allowed to acquire a shared/exclusive lock on the same data item.

There are two kind of lock based protocol mostly used in database:

Two Phase Locking Protocol : Two phase locking is a widely used technique which ensures strict ordering of lock acquisition and release. Two phase locking protocol works in two phases.

1.Growing Phase : In this phase, the transaction starts acquiring locks before performing any modification on the data items. Once a transaction acquires a lock, that lock can not be released until the transaction reaches the end of the execution.

2.Shrinking Phase : In this phase, the transaction releases all the acquired locks once it performs all the modifications on the data item. Once the transaction starts releasing the locks, it can not acquire any locks further.

Strict Two Phase Locking Protocol : It is almost similar to the two phase locking protocol the only difference is that in two phase locking the transaction can release its locks before it commits, but in case of strict two phase locking the transactions are only allowed to release the locks only when they performs commits.

Timestamp based Protocol:

Time stamp is a unique identifier created by DBMS that indicates relative starting time of a transaction. Whatever transaction we are doing it stores the starting time of the transaction and denotes a specific time.

This can be generated using a system clock or logical counter. This can be started whenever a transaction is started. Here, the logical counter is incremented after a new timestamp has been assigned.

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.

The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.

The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction T_i issues a $\text{Read}(X)$ operation:

- o If $W_{\text{TS}}(X) > TS(T_i)$ then the operation is rejected.
- o If $W_{\text{TS}}(X) \leq TS(T_i)$ then the operation is executed.
- o Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction T_i issues a $\text{Write}(X)$ operation:

If $TS(T_i) < R_{\text{TS}}(X)$ then the operation is rejected.

If $TS(T_i) < W_TS(X)$ then the operation is rejected and T_i is rolled back otherwise the operation is executed.

Where

$TS(T_i)$ denotes the timestamp of the transaction T_i .

$R_TS(X)$ denotes the Read time-stamp of data-item X .

$W_TS(X)$ denotes the Write time-stamp of data-item X .

Optimistic :

Optimistic concurrency control assumes that conflicts between transactions are rare. It allows transactions to execute without acquiring locks:

Phases:

- 1. Read Phase:** Transactions read data and make changes in a local workspace.
- 2. Validation Phase:** Before committing, the system checks if any other transaction has modified the data read by the current transaction.
- 3. Write Phase:** If validation is successful, changes are applied; otherwise, the transaction is rolled back

Deadlocks:

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

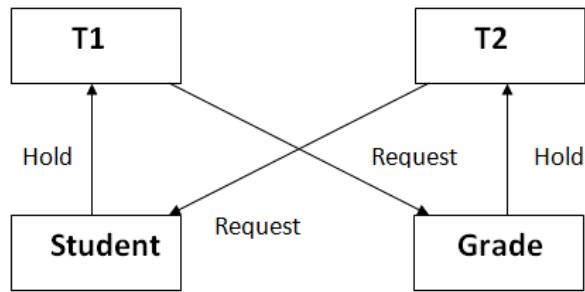


Figure: Deadlock in DBMS

Characteristics of Deadlock

- **Mutual Exclusion:** Only one transaction can hold a particular resource at a time.
- **Hold and Wait:** The Transactions holding resources may request additional resources held by others.
- **No Preemption:** The Resources cannot be forcibly taken from the transaction holding them.
- **Circular Wait:** A cycle of transactions exists where each transaction is waiting for the resource held by the next transaction in the cycle.

Deadlock Avoidance

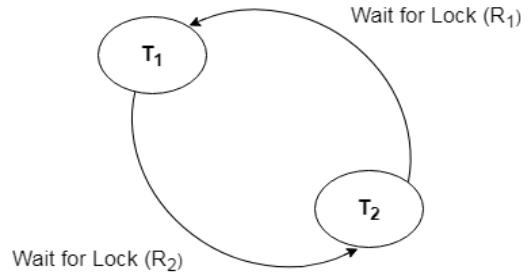
- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for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

Deadlock Detection

- When a transaction waits indefinitely to obtain a lock, The database management system should detect whether the transaction is involved in a deadlock or not.
- **Wait-for-graph** is one of the methods for detecting the deadlock situation. This method is suitable for smaller databases. In this method, a graph is

drawn based on the transaction and its lock on the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

For the above-mentioned scenario, the Wait-For graph is drawn below:



Deadlock Prevention

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

Wait-Die scheme

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

Let's assume there are two transactions T_i and T_j and let $TS(T)$ is a timestamp of any transaction T . If T_2 holds a lock by some other transaction and T_1 is requesting for resources held by T_2 then the following actions are performed by DBMS:

1. Check if $TS(T_i) < TS(T_j)$ - If T_i is the older transaction and T_j has held some resource, then T_i is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.

2. Check if $TS(T_i) < TS(T_j)$ - If T_i is older transaction and has held some resource and if T_j is waiting for it, then T_j is killed and restarted later with the random delay but with the same timestamp.

Wound wait scheme

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

Failure Classification:

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

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.

Reasons for a transaction failure could be -

Logical errors: If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.

Syntax error: It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it.

For example, The system aborts an active transaction, in case of deadlock or resource unavailability.

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.

Storage, Recovery and Atomicity:

Storage:

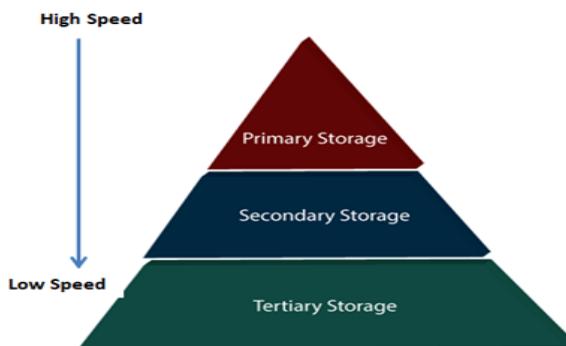
A database system provides an ultimate view of the stored data. However, data in the form of bits, bytes get stored in different storage devices.

In this section, we will take an overview of various types of storage devices that are used for accessing and storing data.

Types of Data Storage

For storing the data, there are different types of storage options available. These storage types differ from one another as per the speed and accessibility. There are the following types of storage devices used for storing the data:

- Primary Storage
- Secondary Storage
- Tertiary Storage



Primary Storage

It is the primary area that offers quick access to the stored data. We also know the primary storage as volatile storage. It is because this type of memory does not permanently store the data. As soon as the system leads to a power cut or a crash, the data also get lost. Main memory and cache are the types of primary storage.

Main Memory: It is the one that is responsible for operating the data that is available by the storage medium. The main memory handles each instruction of a computer machine. This type of memory can store gigabytes of data on a system but is small enough to carry the entire database. At last, the main memory loses the whole content if the system shuts down because of power failure or other reasons.

Cache: It is one of the costly storage media. On the other hand, it is the fastest one. A cache is a tiny storage media which is maintained by the computer hardware usually.

Secondary Storage

Secondary storage is also called as Online storage. It is the storage area that allows the user to save and store data permanently. This type of memory does not lose the data due to any power failure or system crash. That's why we also call it non-volatile storage.

There are some commonly described secondary storage media which are available in almost every type of computer system:

Flash Memory: A flash memory stores data in USB (Universal Serial Bus) keys which are further plugged into the USB slots of a computer system. These USB keys help transfer data to a computer system, but it varies in size limits. This leads the systems towards high performance and is capable of storing large amounts of databases than the main memory.

Magnetic Disk Storage: This type of storage media is also known as online storage media. A magnetic disk is used for storing the data for a long time. It is capable of storing an entire database. It is the responsibility of the computer system to make availability of the data from a disk to the main memory for further accessing.

Tertiary Storage

It is the storage type that is external from the computer system. It has the slowest speed. But it is capable of storing a large amount of data. It is also known as Offline storage.

Optical Storage: An optical storage can store megabytes or gigabytes of data. A Compact Disk (CD) can store 700 megabytes of data with a playtime of around 80 minutes. On the other hand, a Digital Video Disk or a DVD can store 4.7 or 8.5 gigabytes of data on each side of the disk.

Tape Storage: It is the cheapest storage medium than disks. Generally, tapes are used for archiving or backing up the data. It provides slow access to data as it accesses data sequentially from the start.

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.
- 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.
- Database recovery means recovering the data when it get deleted, hacked or damaged accidentally.
- Atomicity is must whether is transaction is over or not it should reflect in the database permanently or it should not effect the database at all.

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-based recovery is a widely used approach in database management systems to recover from system failures and maintain atomicity and durability of transactions. The fundamental idea behind log-based recovery is to keep a log of all changes made to the database, so that after a failure, the system can use the log to restore the database to a consistent state.

1. Transaction Logging:

For every transaction that modifies the database, an entry is made in the log. This entry typically includes:

- **Transaction ID:** A unique identifier for the transaction.
- **Data item identifier:** Identifier for the specific item being modified.
- **OLD value:** The value of the data item before the modification.
- **NEW value:** The value of the data item after the modification.

We represent an update log record as $\langle \backslash(T_i) , \backslash(X_j) , \backslash(V_1), \backslash(V_2) \rangle$, indicating that transaction $\backslash(T_i)$ has performed a write on data item $\backslash(X_j)$. $\backslash(X_j)$ had value $\backslash(V_1)$ before the write, and has value $\backslash(V_2)$ after the write..

Among the types of log records are:

- $\langle \backslash(T_i) \text{ start} \rangle$. Transaction Ti has started.
- $\langle \backslash(T_i) \text{ commit} \rangle$. Transaction Ti has committed.
- $\langle \backslash(T_i) \text{ abort} \rangle$. Transaction Ti has aborted.

2. Writing to the Log

Before any change is written to the actual database (on disk), the corresponding log entry is stored. This is called the **Write-Ahead Logging (WAL)** principle. By ensuring that the log is written first, the system can later recover and apply or undo any changes.

3. Checkpointing

Periodically, the DBMS might decide to take a checkpoint. A checkpoint is a point of synchronization between the database and its log. At the time of a checkpoint:

- All the changes in main memory (buffer) up to that point are written to disk.
- A special entry is made in the log indicating a checkpoint. This helps in reducing the amount of log that needs to be scanned during recovery.

4. Recovery Process

- **Redo:** If a transaction is identified (from the log) as having committed but its changes have not been reflected in the database (due to a crash before the changes could be written to disk), then the changes are reapplied using the 'After Image' from the log.
- **Undo:** If a transaction is identified as not having committed at the time of the crash, any changes it made are reversed using the 'Before Image' in the log to ensure atomicity.

5. Commit/Rollback

Once a transaction is fully complete, a commit record is written to the log. If a transaction is aborted, a rollback record is written, and using the log, the system undoes any changes made by this transaction.

Benefits of Log-Based Recovery

- **Atomicity:** Guarantees that even if a system fails in the middle of a transaction, the transaction can be rolled back using the log.
- **Durability:** Ensures that once a transaction is committed, its effects are permanent and can be reconstructed even after a system failure.
- **Efficiency:** Since logging typically involves sequential writes, it is generally faster than random access writes to a database.

Shadow paging - Its Working principle

Shadow Paging is an alternative disk recovery technique to the more common logging mechanisms. It's particularly suitable for database systems. The fundamental concept behind shadow paging is to maintain two page tables during the lifetime of a transaction: the current page table and the shadow page table.

Here's a step-by-step breakdown of the working principle of shadow paging:

Initialization

When the transaction begins, the database system creates a copy of the current page table. This copy is called the shadow page table.

The actual data pages on disk are not duplicated; only the page table entries are. This means both the current and shadow page tables point to the same data pages initially.

During Transaction Execution

When a transaction modifies a page for the first time, a copy of the page is made. The current page table is updated to point to this new page.

Importantly, the shadow page table remains unaltered and continues pointing to the original, unmodified page.

Any subsequent changes by the same transaction are made to the copied page, and the current page table continues to point to this copied page.

On Transaction Commit

Once the transaction reaches a commit point, the shadow page table is discarded, and the current page table becomes the new "truth" for the database state.

The old data pages that were modified during the transaction (and which the shadow page table pointed to) can be reclaimed.

Recovery after a Crash

If a crash occurs before the transaction commits, recovery is straightforward. Since the original data pages (those referenced by the shadow page table) were never modified, they still represent a consistent database state.

The system simply discards the changes made during the transaction (i.e., discards the current page table) and reverts to the shadow page table.

Recovery algorithm:

Types of Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss.

There are mainly two types of recovery techniques used in DBMS

- **Rollback/Undo Recovery Technique**
- **Commit/Redo Recovery Technique**
- **CheckPoint Recovery Technique**

Database recovery techniques ensure data integrity in case of system failures. Understanding how these techniques work is crucial for managing databases effectively.

Rollback/Undo Recovery Technique

The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not been completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.

Commit/Redo Recovery Technique

The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.

Checkpoint Recovery Technique

Checkpoint Recovery is a technique used to improve data integrity and system stability, especially in databases and distributed systems. It entails preserving the system's state at regular intervals, known as checkpoints, at which all ongoing transactions are either completed or not initiated. This saved state, which includes memory and CPU registers, is kept in stable, non-volatile storage so that it can withstand system crashes. In the event of a breakdown, the system can be restored to the most recent checkpoint, which reduces data loss and downtime. The frequency of checkpoint formation is carefully regulated to decrease system overhead while ensuring that recent data may be restored quickly.

ARIES Algorithm

- ARIES is a recovery algorithm.
- It stands for Algorithm for Recovery and Isolation Exploiting Semantics.
- It is based on Write Ahead Log (WAL) protocol.
- When database crashes during some transaction processing, we have the logs that got saved to the disk.

ARIES has 3 phases that occur in the following order –

1. Analysis Phase:

The Analysis Phase identifies the database's state and determines which log entries need to be processed for recovery.

Example: Suppose we have a log with the following entries

1. <T1, Start>
2. <T1, Write(A, 10)>
3. <T1, Commit>
4. <T2, Start>
5. <T2, Write(B, 20)>
6. <T2, Abort>

During the Analysis Phase:

- The system reads the log to identify active transactions (T2) and committed transactions (T1).
- It also records the state of database pages that were modified by active transactions.

Steps:

1. Scan the log to identify transactions and their states.
2. Create a list of dirty pages (pages that have been modified but not yet written to disk).

2. Redo Phase:

The Redo Phase re-applies changes recorded in the log to ensure that all committed transactions are properly reflected in the database.

Example: Using the log from the previous example, the Redo Phase will:

- Reapply the changes made by T1 and T2. Since T1 committed, its changes to page A will be redone.
- T2's changes are ignored because the transaction was aborted.

Steps:

1. Replay every action taken during a committed transaction.
2. Ensure all modifications are applied to the database pages as per the log.

3. Undo Phase:

The Undo Phase reverses changes made by aborted transactions, restoring the database to a consistent state by undoing any modifications from these transactions.

Example: From the log, we need to undo changes made by T2.

Steps:

1. Identify all operations from aborted transactions.
2. Apply undo operations to revert changes made by these transactions.

Introduction to Indexing Techniques

B+ Trees:

The B+ tree is a balanced binary search tree. It follows a multi-level index format.

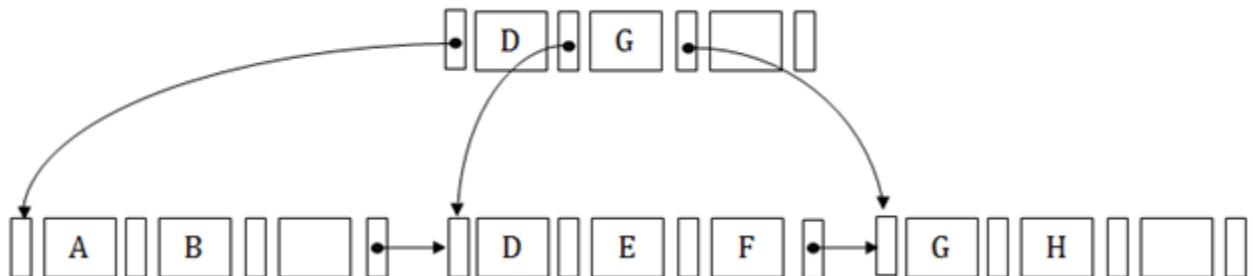
In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.

In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.

It contains an internal node and leaf node.



Internal node

An internal node of the B+ tree can contain at least $n/2$ record pointers except the root node.

At most, an internal node of the tree contains n pointers.

Leaf node

The leaf node of the B+ tree can contain at least $n/2$ record pointers and $n/2$ key values.

At most, a leaf node contains n record pointer and n key values.

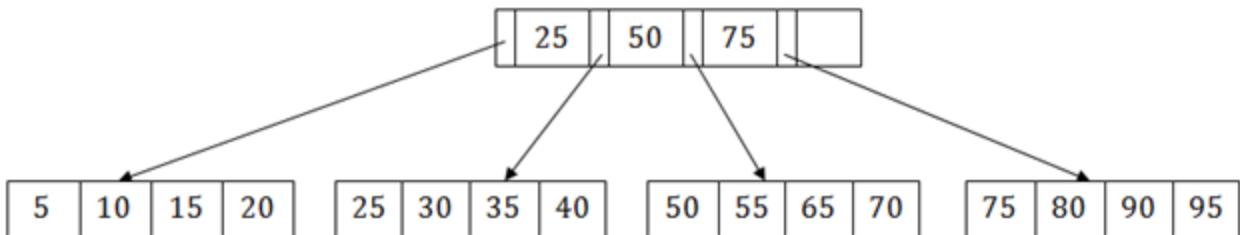
Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

Operations on B+ Trees:

Searching a record in B+ Tree

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

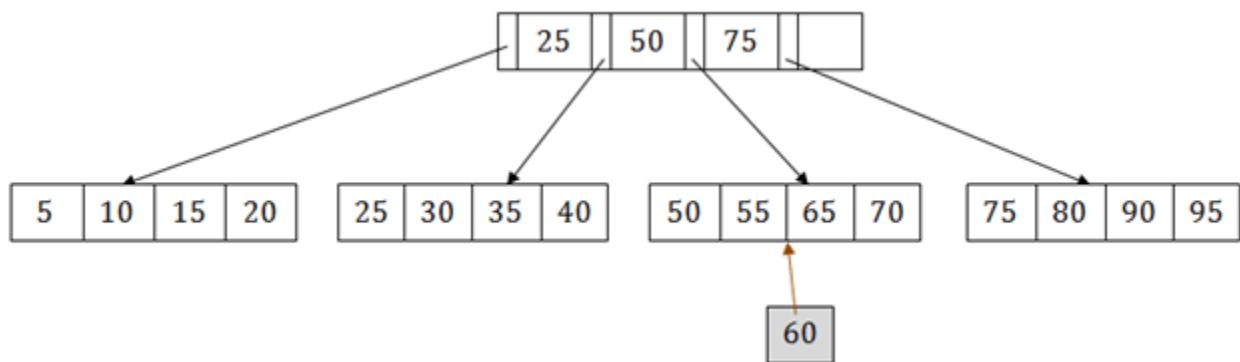
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

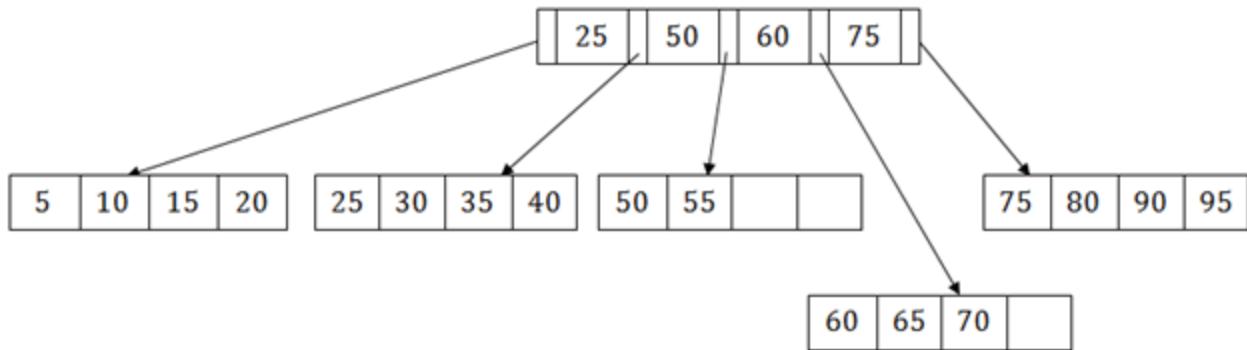
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

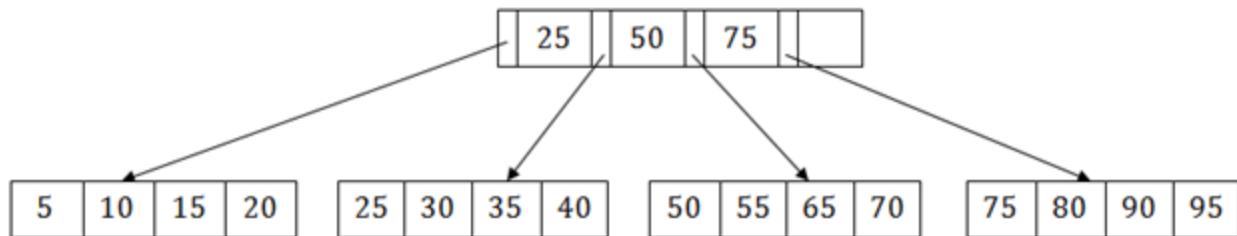


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



Hash Based Indexing:

Hashing in DBMS is a technique to quickly locate a data record in a database irrespective of the size of the database.

The hashing technique utilizes an auxiliary hash table to store the data records using a hash function. There are 2 key components in hashing:

- **Hash Table:** A hash table is an array or data structure and its size is determined by the total volume of data records present in the database. Each memory location in a hash table is called a '*bucket*' or hash indice and stores a data record's exact location and can be accessed through a hash function.
- **Bucket:** A bucket is a memory location (index) in the hash table that stores the data record. These buckets generally store a disk block which further stores multiple records. It is also known as the hash index.
- **Hash Function:** A hash function is a mathematical equation or algorithm that takes one data record's primary key as input and computes the hash index as output.

Hash Function

A hash function is a mathematical algorithm that computes the index or the location where the current data record is to be stored in the hash table so that it can be accessed efficiently later. This hash function is the most crucial component that determines the speed of fetching data.

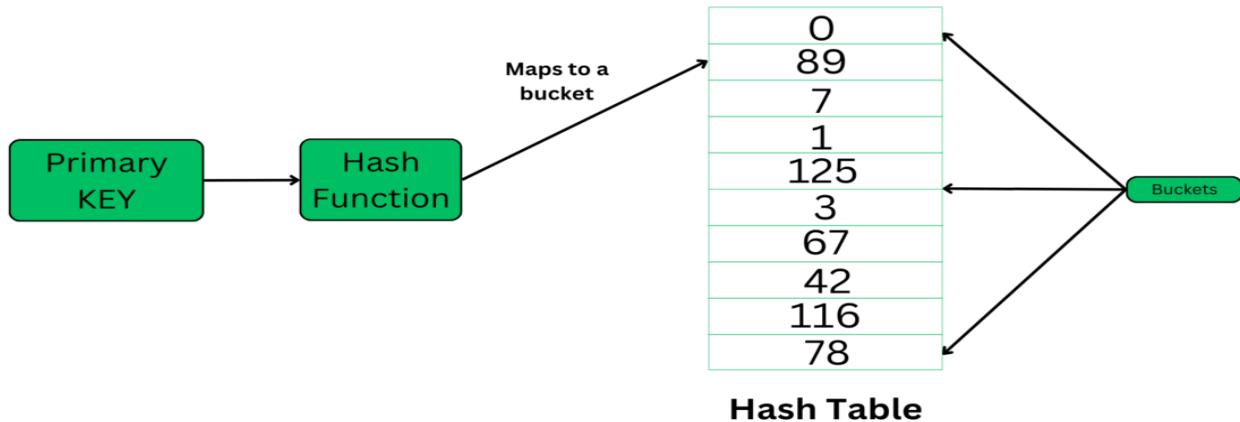
Working of Hash Function

The hash function generates a hash index through the primary key of the data record.

Now, there are 2 possibilities:

1. The hash index generated isn't already occupied by any other value. So, the address of the data record will be stored here.
2. The hash index generated is already occupied by some other value. This is called collision so to counter this, a collision resolution technique will be applied.
3. Now whenever we query a specific record, the hash function will be applied and returns the data record comparatively faster than indexing because we can directly reach the exact location of the data record through the hash function rather than searching through indices one by one.

Example:



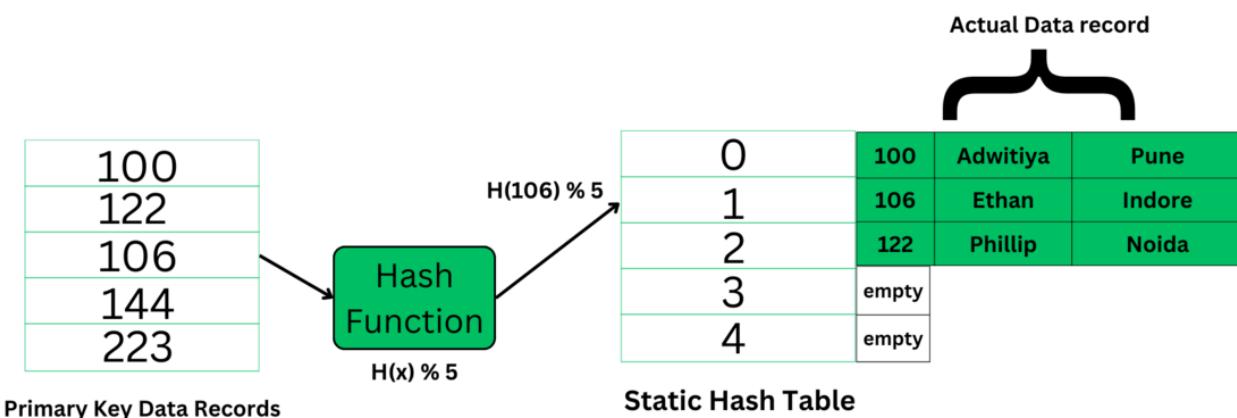
Types of Hashing in DBMS

There are two primary hashing techniques in DBMS.

1. Static Hashing:

In static hashing, the hash function always generates the same bucket's address. For example, if we have a data record for employee_id = 107, the hash function is mod-5 which is $- H(x) \% 5$, where $x = \text{id}$.

Then the operation will take place like this: $H(106) \% 5 = 1$.



The primary key is used as the input to the hash function and the hash function generates the output as the hash index (bucket's address) which contains the address of the actual data record on the disk block.

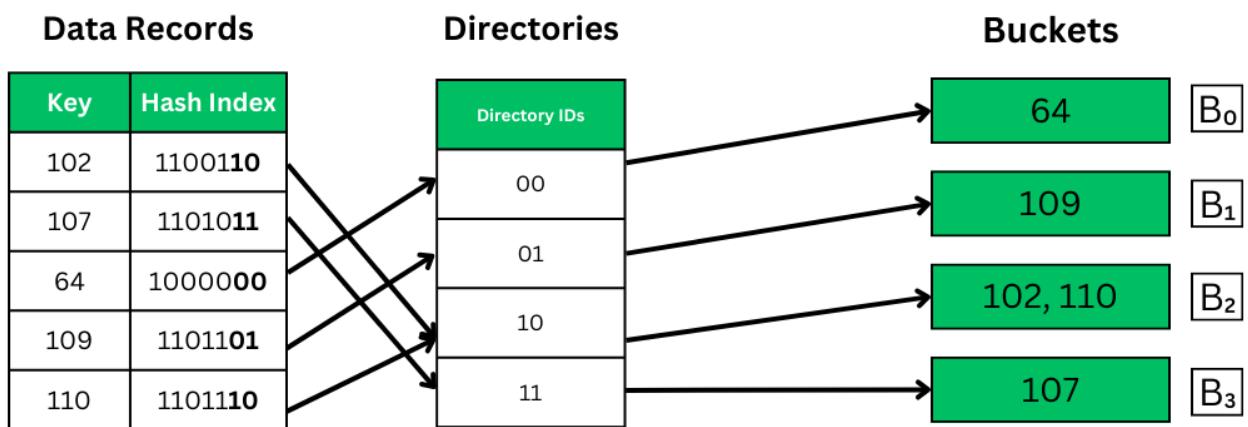
2. Dynamic Hashing:

Dynamic hashing is also known as extendible hashing, used to handle database that frequently changes data sets. This method offers us a way to add and remove

data buckets on demand dynamically. This way as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.

Working of Dynamic Hashing:

Example: If global depth: $k = 2$, the keys will be mapped accordingly to the hash index. K bits starting from LSB will be taken to map a key to the buckets. That leaves us with the following 4 possibilities: 00, 11, 10, 01.



As we can see in the above image, the k bits from LSBs are taken in the hash index to map to their appropriate buckets through directory IDs. The hash indices point to the directories, and the k bits are taken from the directories' IDs and then mapped to the buckets. Each bucket holds the value corresponding to the IDs converted in binary.