UNIT V

Transaction Concept: Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm.

Indexing Techniques: B+ Trees: Search, Insert, Delete algorithms, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes , Index data Structures, Hash Based Indexing: Tree base Indexing ,Comparison of File Organizations, Indexes and Performance Tuning

TRANSACTIONS

Transaction is a set of operations which are all logically related.

A **transaction** is a very small unit of a program which contains group of tasks. A transaction accesses the contents of a database by using read and write operations.

Operations in Transaction-

The main operations in a transaction are-
Read Operation
Write Operation

1. Read Operation-

Read operation reads the data from the database and then stores it in the buffer in main memory.
For example- **Read(A)** instruction will read the value of A from the database and will store it in the buffer in main memory.

2. Write Operation-

Write operation writes the updated data value back to the database from the buffer.
For example- Write(A) will write the updated value of A from the buffer to the database.

A **transaction** in a database system must maintain Atomicity, Consistency, Isolation, and Durability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

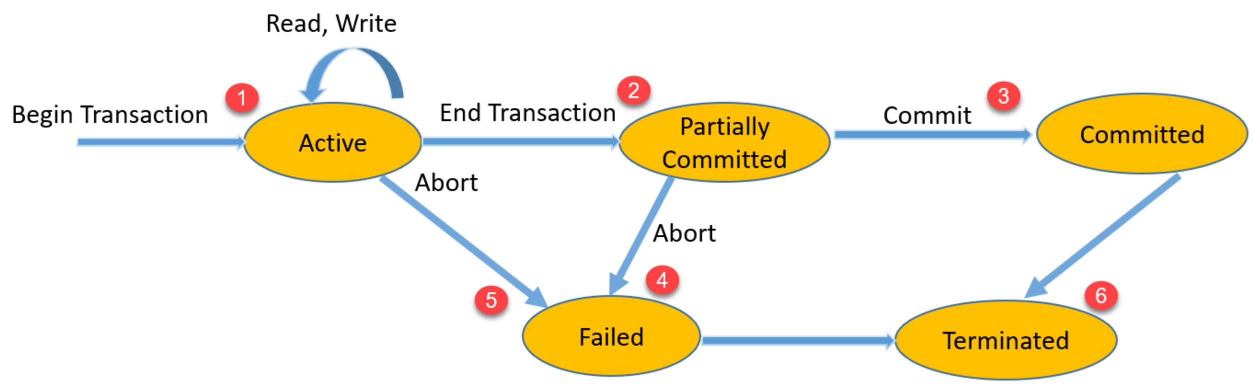
Let’s take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

A’s Account
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)

B’s Account
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)

STATE DIAGRAM OF TRANSACTION:

A transaction goes through many different states throughout its life cycle.
These states are called as **transaction states**.



Transaction states are as follows-

1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Aborted state
6. Terminated state

1. Active State-

This is the first state in the life cycle of a transaction.
A transaction is called in an **active state** as long as its instructions are getting executed.

All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State-

After the last instruction of transaction has executed, it enters into a **partially committed state**. After entering this state, the transaction is considered to be partially committed.

It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State-

After all the changes made by the transaction have been successfully stored into the database, it enters into a committed state.

Now, the transaction is considered to be fully committed.

NOTE-

After a transaction has entered the committed state, it is not possible to roll back the transaction.

In other words, it is not possible to undo the changes that has been made by the transaction.

This is because the system is updated into a new consistent state.

The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

4. Failed State-

When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

5. Aborted State-

After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.

To undo the changes made by the transaction, it becomes necessary to roll back the transaction.

After the transaction has rolled back completely, it enters into an **aborted state**.

6. Terminated State-

This is the last state in the life cycle of a transaction.

After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.

ACID PROPERTIES

A transaction in a database system must maintain **Atomicity, Consistency, Isolation, and Durability** − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none.

  Example: Let’s take an example of banking system to understand this: Suppose Account 'A' has a balance of 400 & B has 700. Account A is transferring 100 to Account B. This is a transaction that has two operations a) Debiting 100 from A’s balance b) Creating 100 to B’s balance. Let’s say first operation passed successfully while second failed, in this case A’s balance would be 300 while B would be having 700 instead of 800. This is unacceptable in a banking system. Either the transaction should fail without executing any of the operation or it should process both the operations. The Atomicity property ensures that.

- **Consistency** − The database must remain in a consistent state after any transaction. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

  Example: Referring to the example above,
  The total amount before and after the transaction must be maintained.
  Total before Transfer from A to B = 400 + 700 = 1100.
  Total after Transfer from A to B = 300 + 800 = 1100.
  Therefore, database is consistent.

- Isolation − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Example: if two transactions T1 and T2 are executed concurrently, the net effect is guaranteed to be equivalent to executing T1 followed by executing T2 or executing T2 followed by executing T1.

- Durability − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a some of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system is on.

TRANSACTION LOG

A DBMS uses a transaction log to keep track of all transactions that update the database. The information stored in this log is used by the DBMS for a recovery requirement triggered by a ROLLBACK statement, a program’s abnormal termination, or a system failure such as a network discrepancy or a disk crash. Some RDBMSs use the transaction log to recover a database forward to a currently consistent state.
After a server failure, for example, Oracle automatically rolls back uncommitted transactions and rolls forward transactions that were committed but not yet written to the physical database.
While the DBMS executes transactions that modify the database, it also automatically updates the transaction log.
The transaction log stores:

- A record for the beginning of the transaction.

- For each transaction component (SQL statement):
- The type of operation being performed (update, delete, insert).
- The names of the objects affected by the transaction (the name of the table).
- The “before” and “after” values for the fields being updated.

- Pointers to the previous and next transaction log entries for the same transaction.
  - The ending (COMMIT) of the transaction.

TRANSACTION MANAGEMENT WITH SQL:

SQL has Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

TCL commands are :

Commit
Rollback
Savepoint

COMMIT command

COMMIT command is used to permanently save any transaction into the database.
When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.
To avoid that, we use the COMMIT command to mark the changes as permanent.
Following is commit command's syntax,

```
Commit;
```

ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.
If we have used the UPDATE command to make some changes into the database, and realize that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.
Following is rollback command's syntax,

```
Rollback ;
or
Rollback to savepoint_name;
```

SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.
Following is savepoint command's syntax,

```
Savepoint savepoint_name;
```

In short, using this command we can name the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.

Using Savepoint and Rollback

Following is the table class,

id	name
1	Abhi
2	Adam
4	Alex

Lets use some SQL queries on the above table and see the results.

```
INSERT INTO class VALUES(5, 'Rahul');
COMMIT;
UPDATE class SET name = 'Abhijit' WHERE id = '5';
SAVEPOINT A;
INSERT INTO class VALUES(6, 'Chris');
SAVEPOINT B;
INSERT INTO class VALUES(7, 'Bravo');
SAVEPOINT C;
SELECT * FROM class;
```

NOTE: SELECT statement is used to show the data stored in the table.

The resultant table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris
7	Bravo

Now let's use the ROLLBACK command to roll back the state of data to the savepoint B.

```
ROLLBACK TO B;
SELECT * FROM class;
```

Now our class table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Now let's again use the ROLLBACK command to roll back the state of data to the savepoint A

```
ROLLBACK TO A;
```

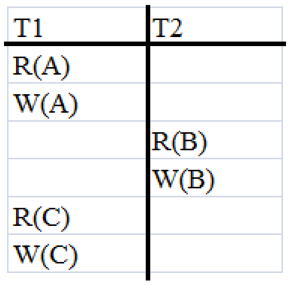
```
SELECT * FROM class;
```

Now the table will look like,
So now you know how the commands COMMIT, ROLLBACK and SAVEPOINT works.

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit

SCHEDULE:

A schedule is a list of actions (reading, writing, aborting, or committing) from a set of transactions and the order of these actions is the same as the order in which they appear.
Example: Consider the following figure which shows the two transactions T1 and T2 and the schedule of actions as R(A),W(A),R(B),W(B),R(C), and W(C).



T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

A schedule that contains either an abort or commit for each transaction whose actions are listed in it is called **complete schedule**.
If the actions of different transactions are not interleaved- that is transactions are executed from start to finish, one by one-we call the schedule is a **serial schedule**.

CONCURRENCY CONTROL

- In the concurrency control, the multiple transactions can be executed simultaneously.
- It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

DBMS Serializability

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which schedules are serializable. **A serializable schedule is the one that always leaves the database in consistent state.**

What is a serializable schedule?

A serial schedule is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.
A non-serial schedule of 'n' number of transactions is said to be serializable, if it is equivalent to the serial schedule of those 'n' transactions. **A serial schedule doesn't allow concurrency, only one transaction executes at a time** and the other starts when the already running transaction finished.

Example: The schedule shown in the following figure is a serialisable schedule.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

Even though the actions of T1 and T2 are interleaved, the result of this schedule is equivalent to running T1(in its entirety) and then running T2.

PROBLEMS(CONFLICTS) OF CONCURRENCY CONTROL

- Several problems can occur when concurrent transactions are executed in an uncontrolled manner. Following are the three problems in concurrency control. Two actions on the same database object **conflict** if at least one of them is a write operation. There are three conflicts: **Write-Read (WR) conflict, Read-Write (RW) conflict, and Write-Write (WW) conflict.**

The following are problems of concurrency execution of transctions:

1. Reading uncommitted data (Dirty read) -(WR conflict)
2. Unrepeatable read –(RW conflict)
3. Lost updates(overwriting uncommitted data)-(WW conflict)

1. Reading uncommitted data -Dirty Read-WR conflict

- A **Dirty read** is the situation when a transaction reads a data that has not yet been committed. For example, Let's say transaction1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.

Example-

Transaction T1	Transaction T2
R(A)	
W(A)	
	R(A) **// Dirty Read**
	W(A)
	Commit
Failure	

Here,

1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back.

In this example,

- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

2. Unrepeatable read(RW conflict):

- Unrepeatable read is also known as Inconsistent Retrievals Problem.
- Unrepeatable read occurs when a transaction reads same row twice, and get a different value each time. For example,

Transaction T1	Transaction T2
R(X)	
	R(X)
W(X)	
	R(X) **//Unrepeatable Read**

Here,

1. T1 reads the value of X (= 10 say).
2. T2 reads the value of X (= 10).
3. T1 updates the value of X (from 10 to 15 say) in the buffer.
4. T2 again reads the value of X (but = 15).

In this example,

- T2 gets to read a different value of X in its second reading.

- T2 wonders how the value of X got changed because according to it, it is running in isolation.

3. Lost updates Problem(overwriting uncommitted data)-(WW conflict)

- A lost update occurs when two different transactions are trying to update the same value within a database at the same time. The result of the first transaction is then "lost", as it is simply overwritten by the second transaction.

Transaction T1	Transaction T2
R(A)	
W(A)	
	W(A)
	commit
commit	

If two transactions T1 and T2 read a record and then update it, then the effect of updating of the first record will be overwritten by the second update.

Here,

1. T1 reads the value of A (= 10 say).
2. T1 updates the value to A (= 15 say) in the buffer.
3. T2 does **blind write** A = 25 (write without read) in the buffer.
4. T2 commits.
5. When T1 commits, it writes A = 25 in the database.

In this example,

1. T1 writes the over written value of X in the database.
2. Thus, update from T1 gets lost.

CONCURRENCY CONTROL PROTOCOL(METHODS)

Concurrency control protocols ensure atomicity, isolation, and serializability of concurrent transactions. The concurrency control protocol can be divided into three categories:

1. Lock based protocol
2. Time-stamp protocol
3. Validation based protocol

1. Locking Methods of Concurrency Control :

"A lock is a variable, associated with the data item, which controls the access of that data item." Locking is the most widely used form of the concurrency control. Locks are further divided into three fields:

1. Lock Granularity
2. Lock Types
3. Deadlocks

1. Lock Granularity:

A database is basically represented as a collection of named data items. The size of the data item chosen as the unit of protection by a concurrency control program is called GRANULARITY. Locking can take place at the following level :

- Database level.
- Table level.
- Page level.
- Row (Tuple) level.
- Attributes (fields) level.

i. Database level Locking :

At database level locking, the entire database is locked. Thus, it prevents the use of any tables in the database by transaction T2 while transaction T1 is being executed. Database level of locking is suitable for batch processes. Being very slow, it is unsuitable for on-line multi-user DBMSs.

ii. Table level Locking :

At table level locking, the entire table is locked. Thus, it prevents the access to any row (tuple) by transaction T2 while transaction T1 is using the table. if a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables. Table level locking is less restrictive than database level. Table level locks are not suitable for multi-user DBMS

iii. Page level Locking :

At page level locking, the entire disk-page (or disk-block) is locked. A page has a fixed size such as 4 K, 8 K, 16 K, 32 K and so on. A table can span several pages, and a page can contain several rows (tuples) of one or more tables. Page level of locking is most suitable for multi-user DBMSs.

iv. Row (Tuple) level Locking :

At row level locking, particular row (or tuple) is locked. A lock exists for each row in each table of the database. The DBMS allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page. The row level lock is much less restrictive than database level, table level, or page level locks. The row level locking improves the availability of data. However, the management of row level locking requires high overhead cost.

v. Attributes (fields) level Locking :

At attribute level locking, particular attribute (or field) is locked. Attribute level locking allows concurrent transactions to access the same row, as long as they require the use of different attributes within the row. The attribute level lock yields the most flexible multi-user data access. It requires a high level of computer overhead.

2. Lock Types :

The DBMS mailnly uses following types of locking techniques.

a. Binary Locking
b. Shared / Exclusive Locking
c. Two - Phase Locking (2PL)

a. Binary Locking :

A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X.

If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested.

We refer to the current value (or state) of the lock associated with item X as LOCK(X).

Two operations, lock_item and unlock_item, are used with binary locking.

Lock_item(X):

A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK(X)

= 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X.

Unlock_item (X):

When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) to 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item ; i.e., at a time only one transaction can hold a lock.

b. Shared / Exclusive Locking :

Shared lock :

These locks are reffered as read locks, and denoted by 'S'.

If a transaction T has obtained Shared-lock on data item X, then T can read X, but cannot write X. Multiple Shared lock can be placed simultaneously on a data item.

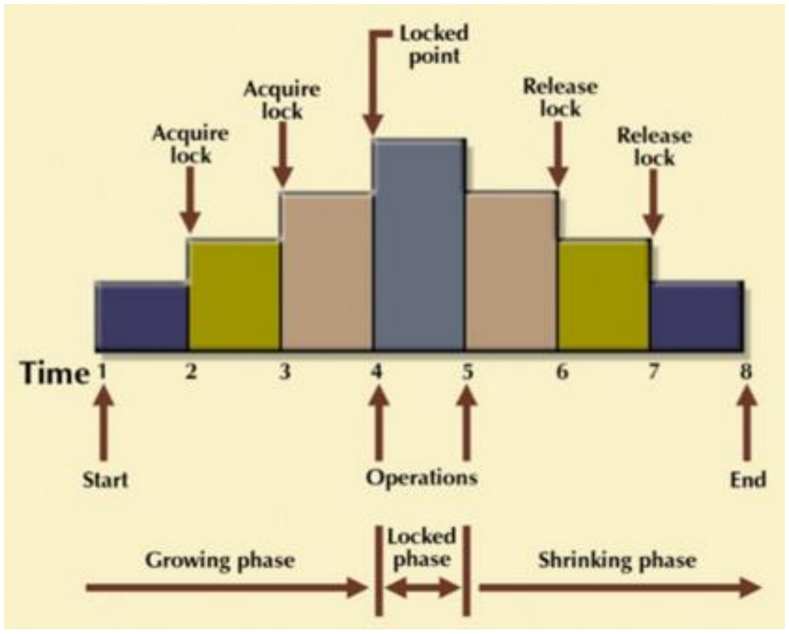
Exclusive lock :

These Locks are referred as Write locks, and denoted by 'X'.
If a transaction T has obtained Exclusive lock on data item X, then T can be read as well as write X. Only one Exclusive lock can be placed on a data item at a time. This means multiple transactions does not modify the same data simultaneously.

c. Two-Phase Locking (2PL) :

Two-phase locking (also called 2PL) is a method or a protocol of controlling concurrent processing in which all locking operations precede the first unlocking operation. Thus, a transaction is said to follow the two-phase locking protocol if all locking operations (such as read_Lock, write_Lock) precede the first unlock operation in the transaction. Two-phase locking is the standard protocol used to maintain level 3 consistency 2PL defines how transactions acquire and relinquish locks. The essential discipline is that after a transaction has released a lock it may not obtain any further locks. 2PL has the following two phases:
This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.



A growing phase, in which a transaction acquires all the required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.

A shrinking phase, in which a transaction releases all locks and cannot obtain any new lock.

A transaction shows Two-Phase Locking lechnique.

Time	Transaction	Remarks
t0	Lock - X (A)	acquire Exclusive lock on A.
t1	Read A	read original value of A
t2	A = A - 100	subtract 100 from A
t3	Write A	write new value of A
t4	Lock - X (B)	acquire Exclusive lock on B.
t5	Read B	read original value of B
t6	B = B + 100	add 100 to B
t7	Write B	write new value of B
t8	Unlock (A)	release lock on A

t9	Unock (B)	release lock on B

3. DEADLOCKS :

A deadlock is a condition in which two (or more) transactions in a set are waiting simultaneously for locks held by some other transaction in the set.

Neither transaction can continue because each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. Thus, a deadlock is an impasse that may result when two or more transactions are each waiting for locks to be released that are held by the other. Transactions whose lock requests have been refused are queued until the lock can be granted.

A deadlock is also called a circular waiting condition where two transactions are waiting (directly or indirectly) for each other. Thus in a deadlock, two transactions are mutually excluded from accessing the next record required to complete their transactions, also called a deadly embrace.

Example:

A deadlock exists two transactions A and B exist in the following example:

Transaction A = access data items X and Y

Transaction B = access data items Y and X

Here, Transaction-A has aquired lock on X and is waiting to acquire lock on y. While, Transaction-B has aquired lock on Y and is waiting to aquire lock on X. But, none of them can execute further.

Transaction-A	Time	Transaction-B
---	t0	---
Lock (X) (acquired lock on X)	t1	---
---	t2	Lock (Y) (acquired
Lock (Y) (request lock on Y)	t3	---
Wait	t4	Lock (X) (request lock
Wait	t5	Wait

Wait	t6	Wait
Wait	t7	Wait

Deadlock Detection and Prevention:

Deadlock detection:

This technique allows deadlock to occur, but then, it detects it and solves it. Here, a database is periodically checked for deadlocks. If a deadlock is detected, one of the transactions, involved in deadlock cycle, is aborted. other transaction continue their execution. An aborted transaction is rolled back and restarted.

Deadlock Prevention:

Deadlock prevention technique avoids the conditions that lead to deadlocking. It requires that every transaction lock all data items it needs in advance. If any of the items cannot be obtained, none of the items are locked. In other words, a transaction requesting a new lock is aborted if there is the possibility that a deadlock can occur. Thus, a timeout may be used to abort transactions that have been idle for too long. This is a simple but indiscriminate approach. If the transaction is aborted, all the changes made by this transaction are rolled back and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention technique is used in two-phase locking.

2. Time-Stamp Methods for Concurrency control :

Timestamp is a unique identifier created by the DBMS to identify the relative starting time of a transaction.

Typically, timestamp values are assigned in the order in which the transactions are submitted to the system. So, a timestamp can be thought of as the transaction start time. Therefore, time stamping is a method of concurrency control in which each transaction is assigned a transaction timestamp. Timestamps must have two properties namely

1. Uniqueness : The uniqueness property assures that no equal timestamp values can exist.
2. monotonicity : monotonicity assures that timestamp values always increase.

Timestamp are divided into further fields :

1. Granule Timestamps
2. Timestamp Ordering
3. Conflict Resolution in Timestamps

1. Granule Timestamps :

Granule timestamp is a record of the timestamp of the last transaction to access it. Each granule accessed by an active transaction must have a granule timestamp.

A separate record of last Read and Write accesses may be kept. Granule timestamp may cause. Additional Write operations for Read accesses if they are stored with the granules. The problem can be avoided by maintaining granule timestamps as an in-memory table. The table may be of limited size, since conflicts may only occur between current transactions. An entry in a granule timestamp table consists of the granule identifier and the transaction timestamp. The record containing the largest (latest) granule timestamp removed from the table is also maintained. A search for a granule timestamp, using the granule identifier, will either be successful or will use the largest removed timestamp.

2. Timestamp Ordering :

Following are the three basic variants of timestamp-based methods of concurrency control:

- Total timestamp ordering
- Partial timestamp ordering
- Multiversion timestamp ordering

(a) Total timestamp ordering :

The total timestamp ordering algorithm depends on maintaining access to granules in timestamp order by aborting one of the transactions involved in any conflicting access. No distinction is made between Read and Write access, so only a single value is required for each granule timestamp .

(b)Partial timestamp ordering :

In a partial timestamp ordering, only non-permutable actions are ordered to improve upon the total timestamp ordering. In this case, both Read and Write granule timestamps are stored.
The algorithm allows the granule to be read by any transaction younger than the last transaction that updated the granule. A transaction is aborted if it tries to update a granule that has previously been accessed by a younger transaction. The partial timestamp ordering algorithm aborts fewer transactions than the total timestamp ordering algorithm, at the cost of extra storage for granule timestamps

(c) Multiversion Timestamp ordering :

The multiversion timestamp ordering algorithm stores several versions of an updated granule, allowing transactions to see a consistent set of versions for all granules it accesses. So, it reduces the conflicts that result in transaction restarts to those where there is a Write-Write conflict.
Each update of a granule creates a new version, with an associated granule timestamp.
A transaction that requires read access to the granule sees the youngest version that is older than the transaction. That is, the version having a timestamp equal to or immediately below the transaction's timestamp.

3. Conflict Resolution in Timestamps :

To deal with conflicts in timestamp algorithms, some transactions involved in conflicts are made to wait and to abort others.
Following are the main strategies of conflict resolution in timestamps:

WAIT-DIE:

- The older transaction waits for the younger if the younger has accessed the granule first.
- The younger transaction is aborted (dies) and restarted if it tries to access a granule after an older concurrent transaction.

WOUND-WAIT:

- The older transaction pre-empts the younger by suspending (wounding) it if the younger transaction tries to access a granule after an older concurrent transaction.

- An older transaction will wait for a younger one to commit if the younger has accessed a granule that both want.

The handling of aborted transactions is an important aspect of conflict resolution algorithm. In the case that the aborted transaction is the one requesting access, the transaction must be restarted with a new (younger) timestamp. It is possible that the transaction can be repeatedly aborted if there are conflicts with other transactions.

An aborted transaction that had prior access to granule where conflict occurred can be restarted with the same timestamp. This will take priority by eliminating the possibility of transaction being continuously locked out.

Drawbacks of Time-stamp

- Each value stored in the database requires two additional timestamp fields, one for the lasttime the field (attribute) was read and one for the last update.
- It increases the memory requirements and the processing overhead of database.

3. Optimistic Methods of Concurrency Control :

The optimistic method of concurrency control is based on the assumption that conflicts of database operations are rare and that it is better to let transactions run to completion and only check for conflicts before they commit.

An optimistic concurrency control method is also known as validation or certification methods. No checking is done while the transaction is executing. The optimistic method does not require locking or timestamping techniques. Instead, a transaction is executed without restrictions until it is committed. In optimistic methods, each transaction moves through the following phases:

a. Read phase.
b. Validation or certification phase.
c. Write phase.

a. Read phase :

In a Read phase, the updates are prepared using private (or local) copies (or versions) of the granule. In this phase, the transaction reads values of committed data from the database, executes the needed computations, and makes the updates to a private copy of the database values. All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.
It is conventional to allocate a timestamp to each transaction at the end of its Read to determine the set of transactions that must be examined by the validation procedure. These set of transactions are those who have finished their Read phases since the start of the transaction being verified

b. Validation or certification phase :

In a validation (or certification) phase, the transaction is validated to assure that the changes made will not affect the integrity and consistency of the database.
If the validation test is positive, the transaction goes to the write phase. If the validation test is negative, the transaction is restarted, and the changes are discarded. Thus, in this phase the list of granules is checked for conflicts. If conflicts are detected in this phase, the transaction is aborted and restarted. The validation algorithm must check that the transaction has :

- Seen all modifications of transactions committed after it starts.
- Not read granules updated by a transaction committed after its start.

c. Write phase :

In a Write phase, the changes are permanently applied to the database and the updated granules are made public. Otherwise, the updates are discarded and the transaction is restarted. This phase is only for the Read-Write transactions and not for Read-only transactions

DATABASE RECOVERY TECHNIQUES IN DBMS

Database systems, like any other computer system, are subject to failures but the data stored in it must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

There are both automatic and non-automatic ways for both, backing up of data and recovery from any failure situations. The techniques used to recover the lost data due to system crash, transaction errors, viruses, catastrophic failure, incorrect commands execution etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred update and immediate update or backing up data can be used.

Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur in the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

- The log is kept on disk start_transaction(T): This log entry records that transaction T starts the execution.
- read_item(T, X): This log entry records that transaction T reads the value of database item X.
- write_item(T, X, old_value, new_value): This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
- commit(T): This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- abort(T): This records that transaction T has been aborted.
- checkpoint: 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.

A transaction T reaches its **commit** point when all its operations that access the database have been executed successfully i.e. the transaction has reached the point at which it will not **abort** (terminate without completing). Once committed, the transaction is permanently recorded in the database. Commitment always involves writing a commit entry to the log and writing the log to disk. At the time of a system crash, item is searched back in the log for all transactions T that have written a start_transaction(T) entry into the log but have not written a commit(T) entry yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process

- **Undoing –** If a transaction crashes, then the recovery manager may undo transactions i.e. reverse the operations of a transaction. This involves examining a transaction for the log entry write_item(T, x, old_value, new_value) and setting the value of item x in the database to old-value.There are two major techniques for recovery from non-catastrophic transaction failures: deferred updates and immediate updates.
- **Deferred update –** This technique does not physically update the database on disk until a transaction has reached its commit point. Before reaching commit, all transaction updates are recorded in the local transaction workspace. If a transaction fails before reaching its commit point, it will not have changed the database in any way so UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the local transaction workspace, because their effect may not yet have been written in the database. Hence, a deferred update is also known as the **No-undo/redo algorithm**
- **Immediate update –** In the immediate update, the database may be updated by some operations of a transaction before the transaction reaches its commit point. However, these operations are recorded in a log on disk before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo. This technique is known as **undo/redo algorithm.**
- **Caching/Buffering –** In this one or more disk pages that include data items to be updated are cached into main memory buffers and then updated in memory before being written back to disk. A collection of in-memory buffers called the DBMS cache is kept under control of DBMS for holding these buffers. A directory is used to keep track of which database items are in the buffer. A dirty bit is associated with each buffer, which is 0 if the buffer is not modified else 1 if modified.
- **Shadow paging –** It provides atomicity and durability. A directory with n entries is constructed, where the ith entry points to the ith database page on the link. When a transaction began executing the current directory is copied into a shadow directory. When a page is to be modified, a shadow page is allocated in which changes are made and when it is ready to become durable, all pages that refer to original are updated to refer new replacement page.

Some of the backup techniques are as follows :

- **Full database backup** – In this full database including data and database, Meta information needed to restore the whole database, including full-text catalogs are backed up in a predefined time series.
- **Differential backup** – It stores only the data changes that have occurred since last full database backup. When same data has changed many times since last full database backup, a differential backup stores the most recent version of changed data. For this first, we need to restore a full database backup.
- **Transaction log backup** – In this, all events that have occurred in the database, like a record of every single statement executed is backed up. It is the backup of transaction log entries and contains all transaction that had happened to the database. Through this, the database can be recovered to a specific point in time. It is even possible to perform a backup from a transaction log if the data files are destroyed and not even a single committed transaction is lost.

Types of Transaction Recovery

Recovery information is divided into two types:

•Undo (or Rollback) Operations

•Redo (or Cache Restore) Operations

Ingres performs both online and offline recovery, as described in Recovery Modes (see page Recovery Modes).

Undo Operation

Undo or transaction backout recovery is performed by the DBMS Server. For example, when a transaction is aborted, transaction log file information is used to roll back all related updates. The DBMS Server writes the Compensation Log Records (CLRs) to record a history of the actions taken during undo operations.

Redo Operation

A Redo recovery operation is database-oriented. Redo recovery is performed after a server or an installation fails. Its main purpose is to recover the contents of the DMF cached data pages that are lost when a fast-commit server fails. Redo recovery is performed by the recovery process. Redo recovery precedes undo recovery.

Redo Operation in a Cluster Environment

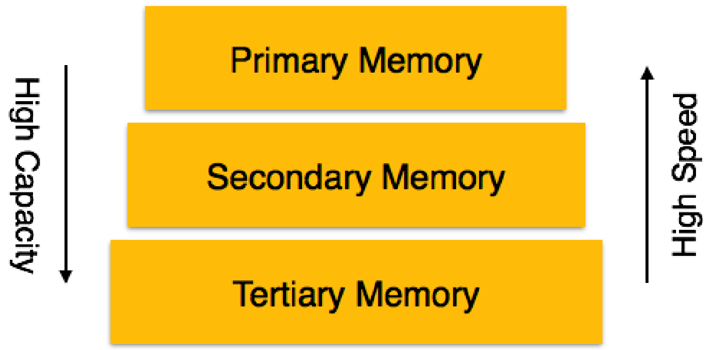
In an Ingres cluster environment where all nodes are active, the local recovery server performs transaction redo/undo for a failed DBMS server on its node, just like in the non-cluster case. The difference in a cluster installation is that if the recovery process (RCP) dies on one node, either because of an Ingres failure, or a general failure of the hardware, an RCP on another node will take responsibility for cleaning up transactions for the failed nodes.

UNIT -6

OVERVIEW OF STORAGES AND INDEXING

DBMS - Storage System

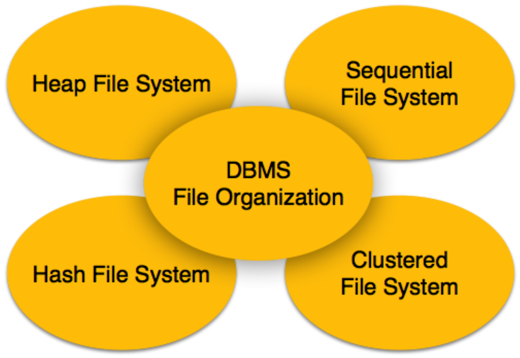
Databases are stored in file formats, which contain records. At physical level, the actual data is stored in electromagnetic format on some device. These storage devices can be broadly categorized into three types −



- **Primary Storage** − The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile. Primary storage requires continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.
- **Secondary Storage** − Secondary storage devices are used to store data for future use or as backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
- **Tertiary Storage** − Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

File Organization

File Organization defines how file records are mapped onto disk blocks. We have four types of File Organization to organize file records −



Heap File Organization

When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records. Heap File does not support any ordering, sequencing, or indexing on its own.

Sequential File Organization

Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization, records are placed in the file in some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

Hash File Organization

Hash File Organization uses Hash function computation on some fields of the records. The output of the hash function determines the location of disk block where the records are to be placed.

Clustered File Organization

Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

DBMS - Indexing

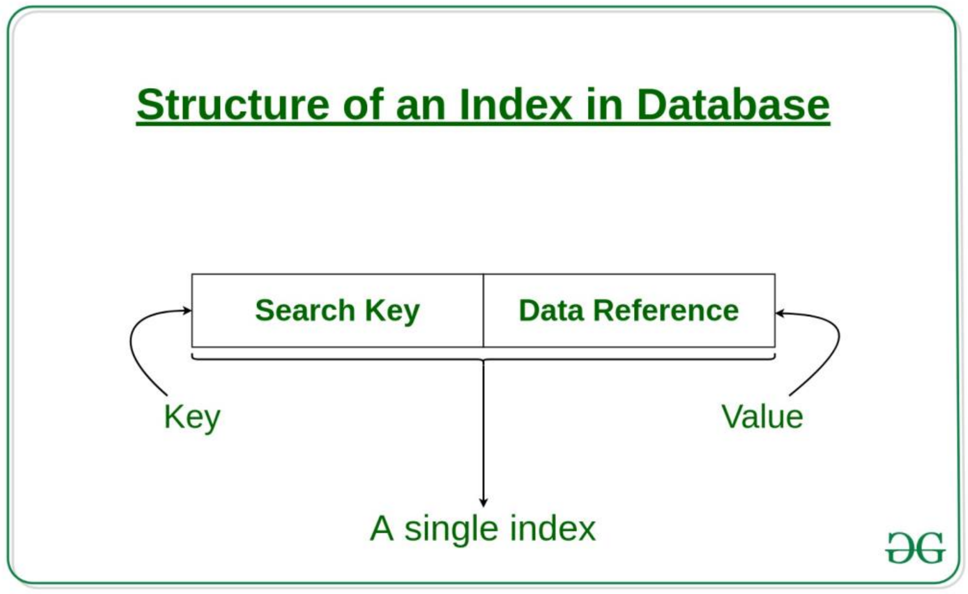
We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

It is a data structure technique which is used to **quickly locate and access** the data in a database.

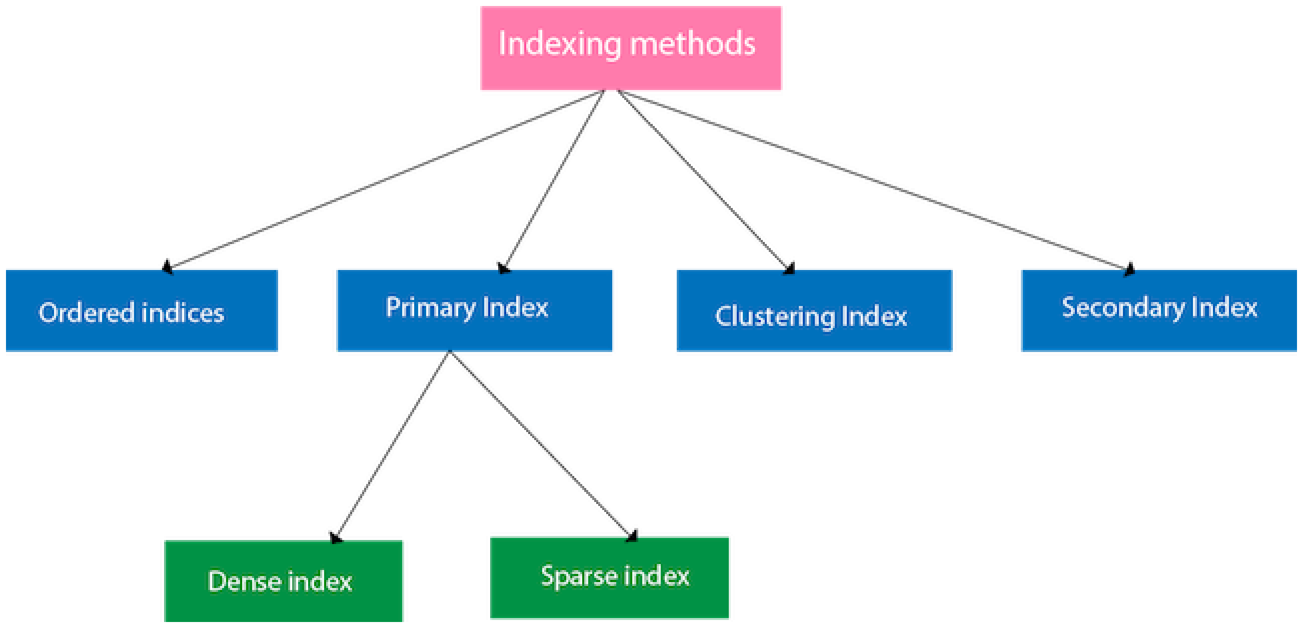
Indexes are created using a few database columns.

- The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.
  Note: The data may or may not be stored in sorted order.
- The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.



In general, there **are two types of file organization mechanism** which are followed by the indexing methods to store the data:

Indexing Methods



Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

Example: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search student with ID-543.

In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543*10=5430 bytes.

In the case of an index, we will search using indexes and the DBMS will read the record after reading 542*2= 1084 bytes which are very less compared to the previous case.

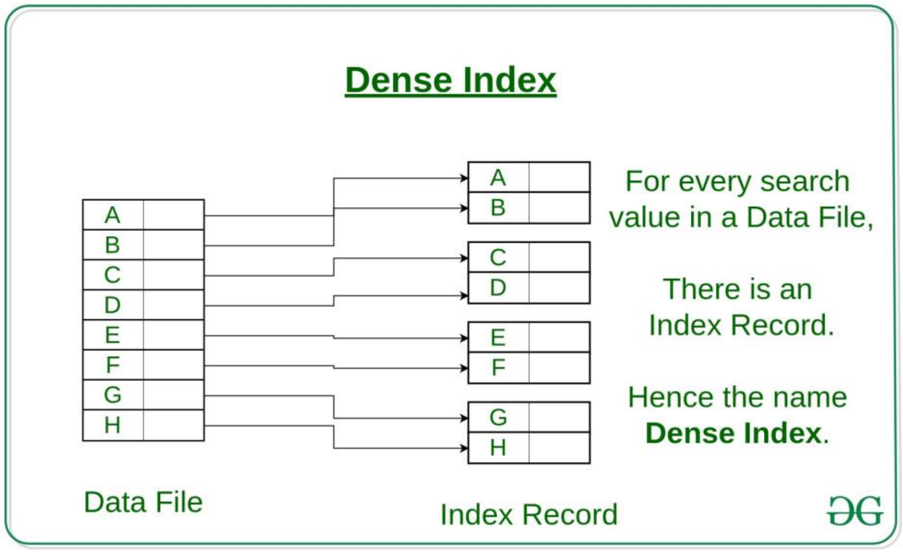
Primary Index

If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.

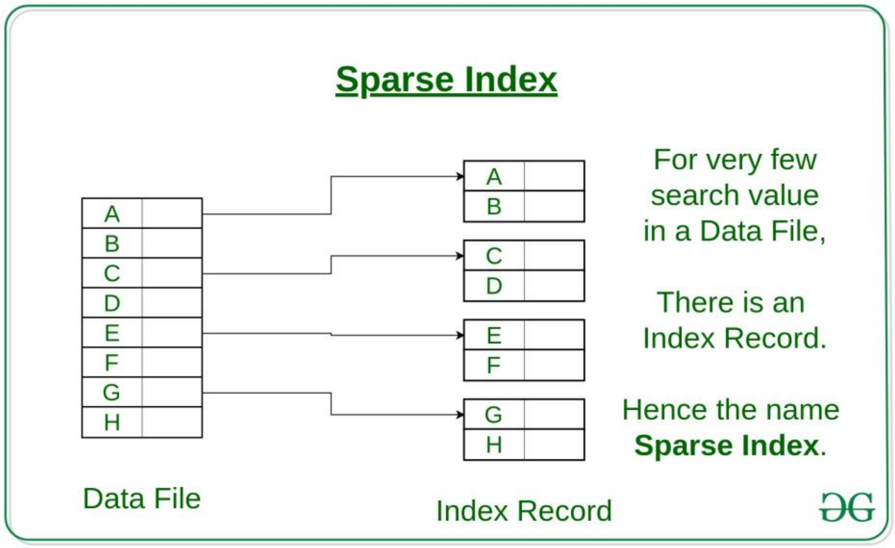
As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.

The primary index can be classified into two types: Dense index and Sparse index.

- **Dense Index:**
  - For every search key value in the data file, there is an index record.
  - This record contains the search key and also a reference to the first data record with that search key value.

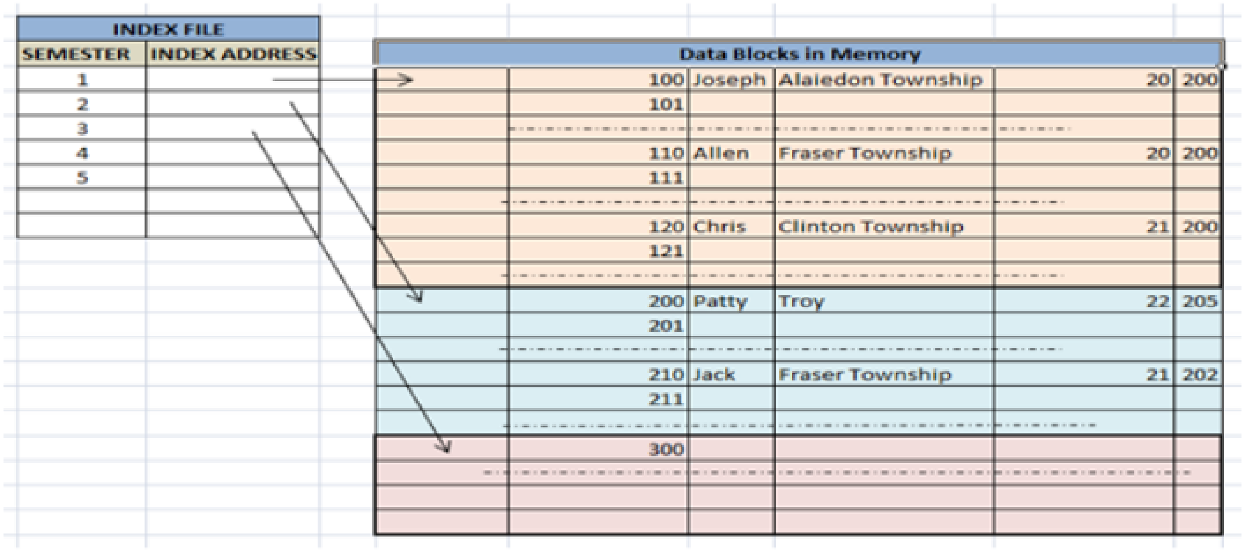

Dense Index
A
B
C
D
E
F
G
H
For every search value in a Data File,
There is an Index Record.
Hence the name Dense Index.
Data File
Index Record

- **Sparse Index:**
  - The index record appears only for a few items in the data file. Each item points to a block as shown.
  - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
  - We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.


Sparse Index
A
B
C
D
E
F
G
H
For very few search value in a Data File,
There is an Index Record.
Hence the name Sparse Index.
Data File
Index Record

Clustered Indexing

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as the clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.
For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.



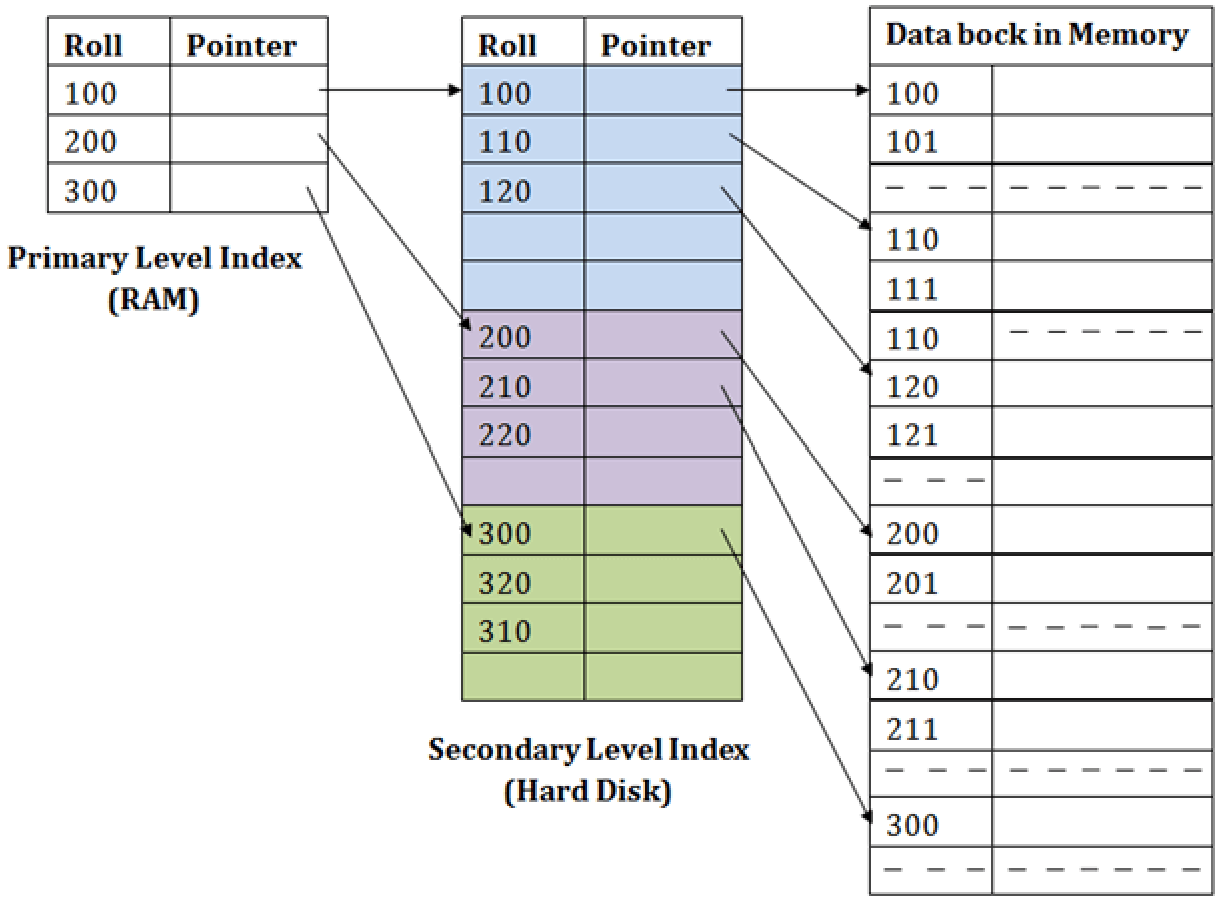
INDEX FILE	
SEMESTER	INDEX ADDRESS
1	
2	
3	
4	
5	

Data Blocks in Memory					
	100	Joseph	Alaiedon Township	20	200
	101				
	110	Allen	Fraser Township	20	200
	111				
	120	Chris	Clinton Township	21	200
	121				
	200	Patty	Troy	22	205
	201				
	210	Jack	Fraser Township	21	202
	211				
	300				

Clustered index sorted according to first name (Search key)

Secondary Indexing:

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced. In this method, the huge range for the columns is selected initially so that the mapping size of the first level becomes small. Then each range is further divided into smaller ranges. The mapping of the first level is stored in the primary memory, so that address fetch is faster. The mapping of the second level and actual data are stored in the secondary memory (hard disk).

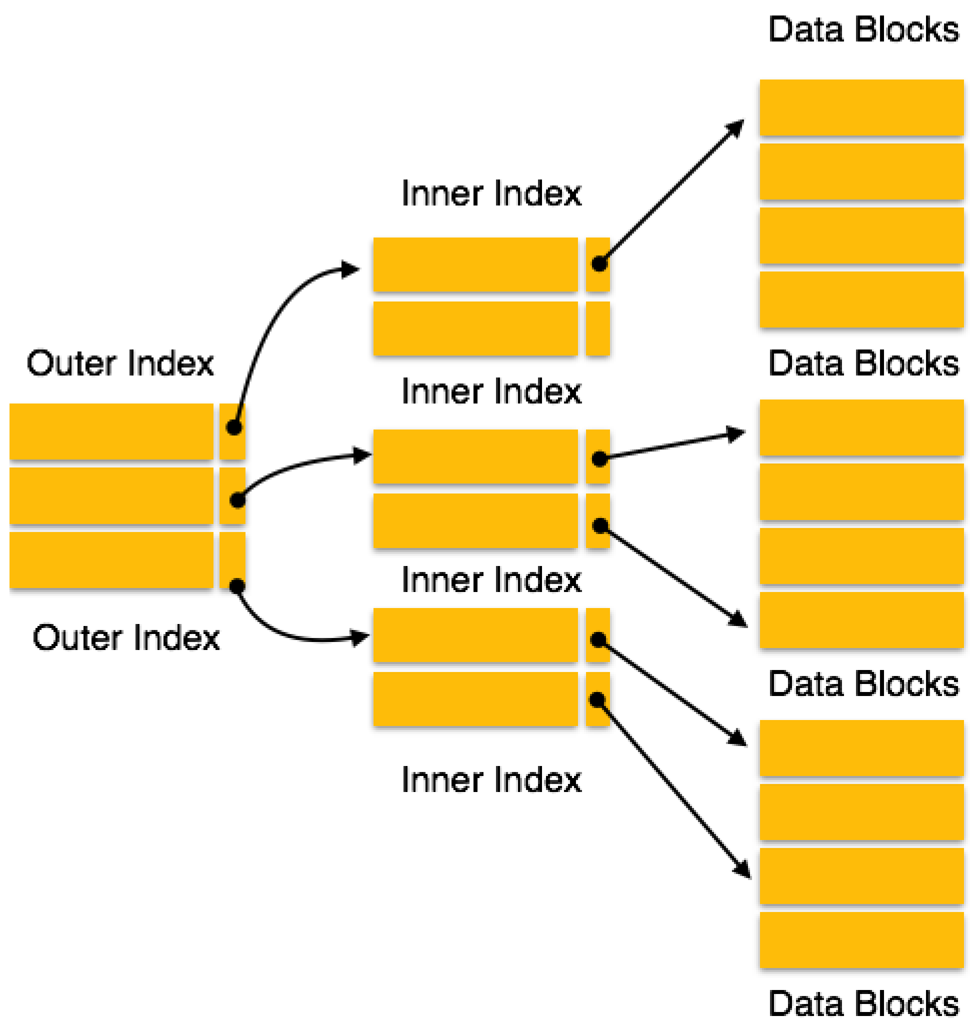


For example:

- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
- Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it goes to the data block and starts searching each record till it gets 111.
- This is how a search is performed in this method. Inserting, updating or deleting is also done in the same manner.

Multilevel Indexing

With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

B Tree

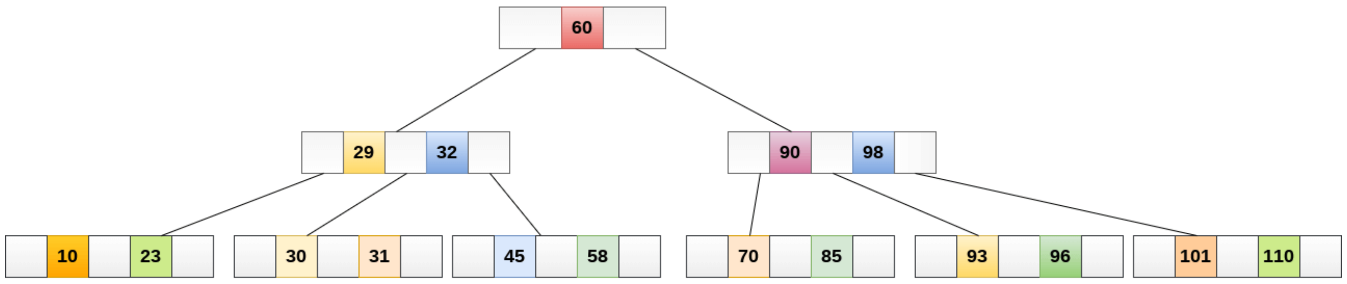
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



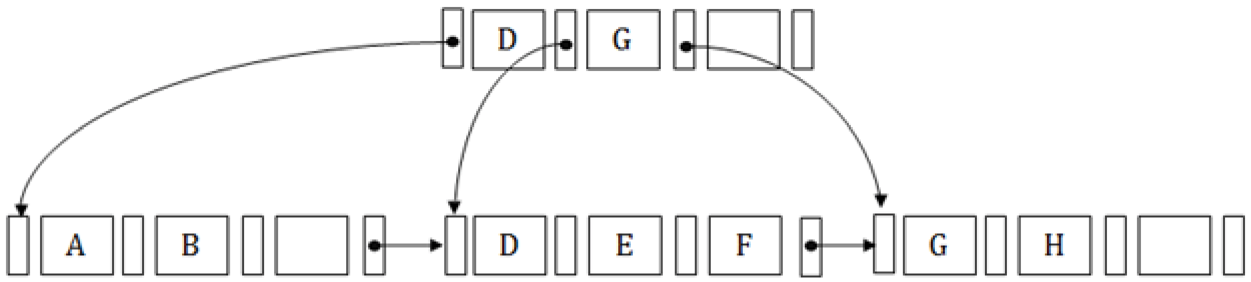
While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

B+ Tree

- 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.

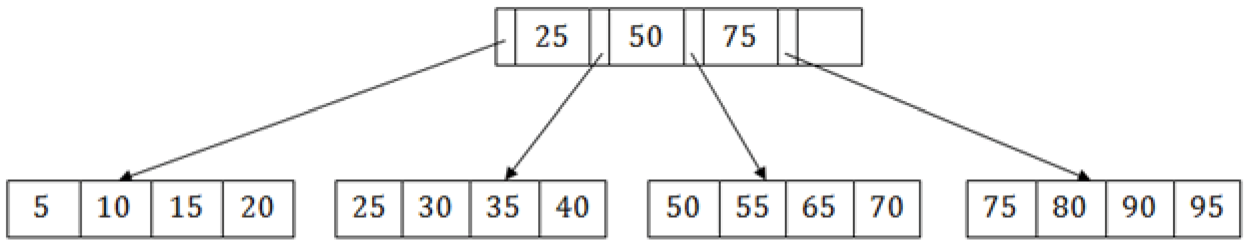
Leaf node

- 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.

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.

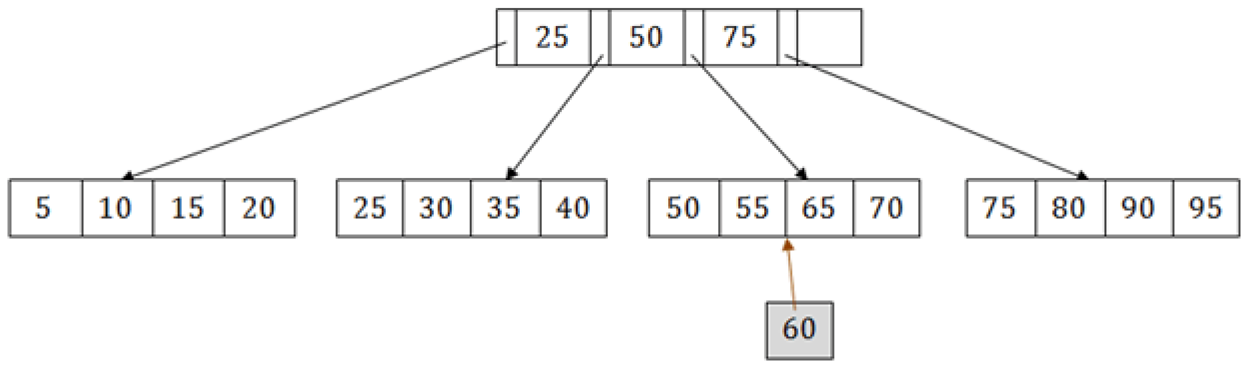
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

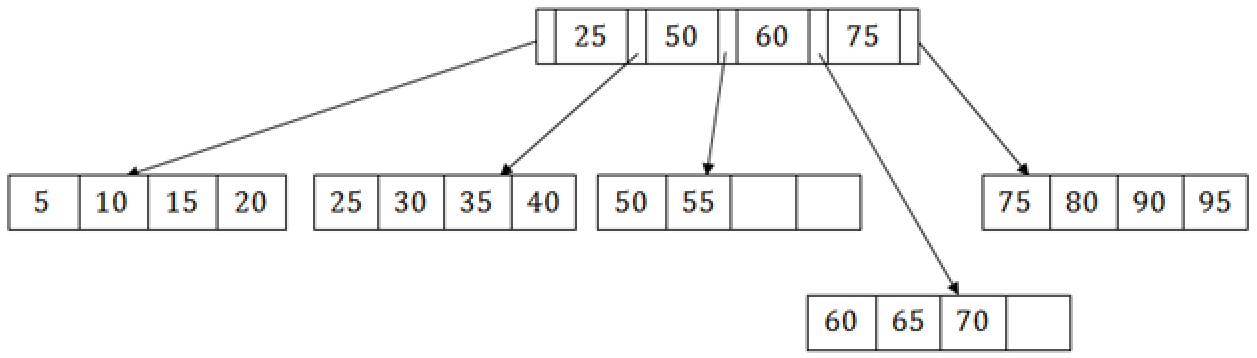
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

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.

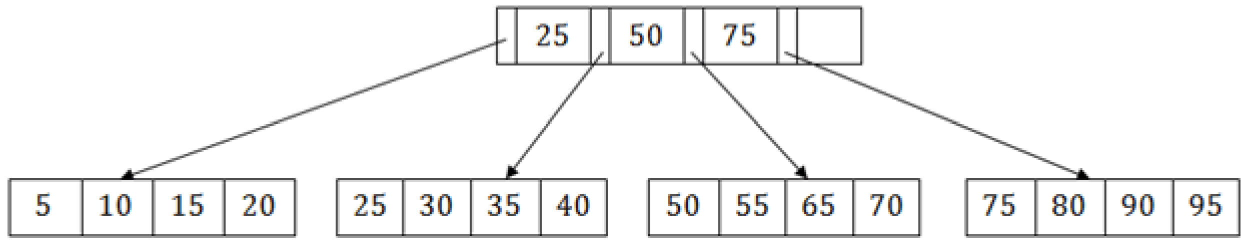


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

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:



DBMS - Hashing

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

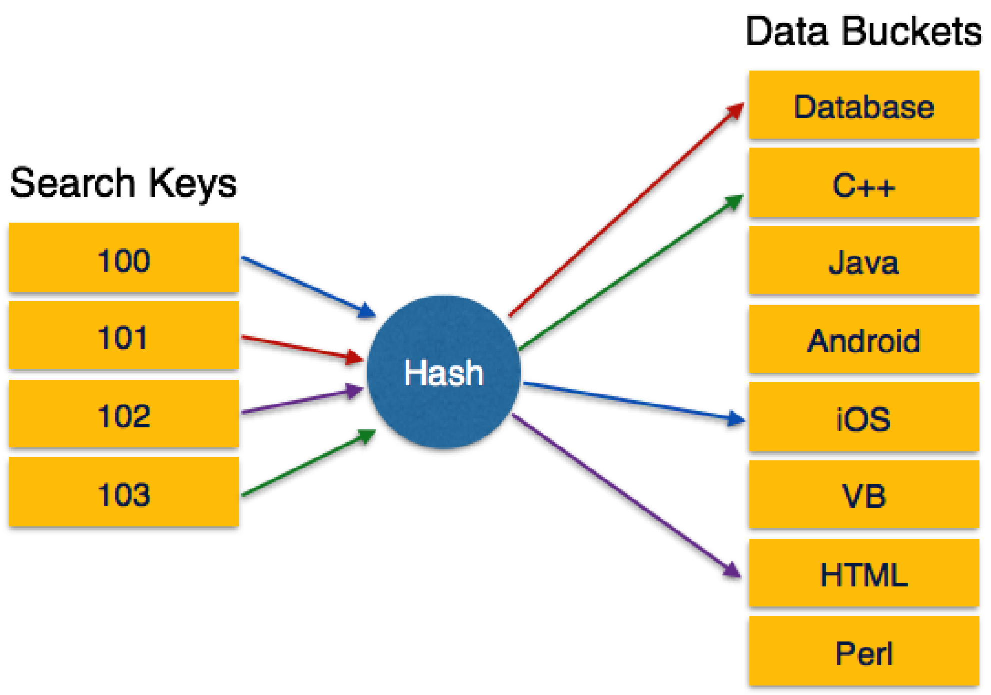
Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

- **Bucket** − A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** − A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



Operation

- **Insertion** − When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

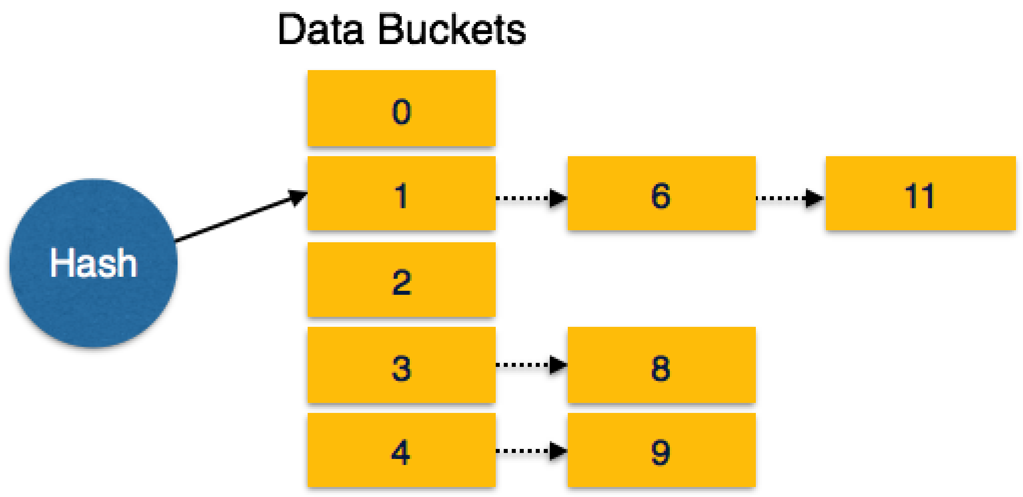
  Bucket address = h(K)

- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** − This is simply a search followed by a deletion operation.

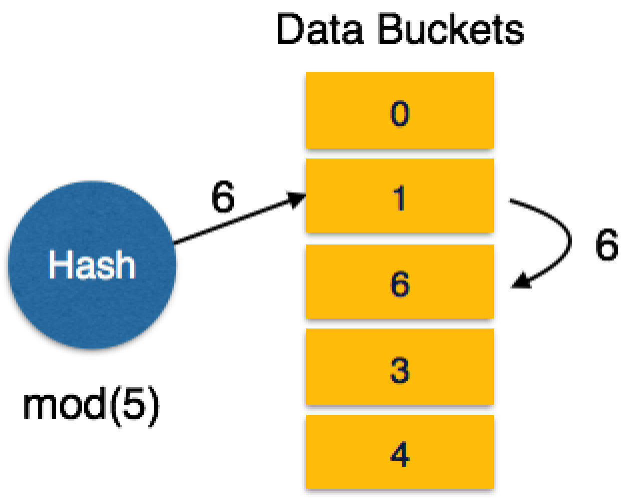
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** − When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



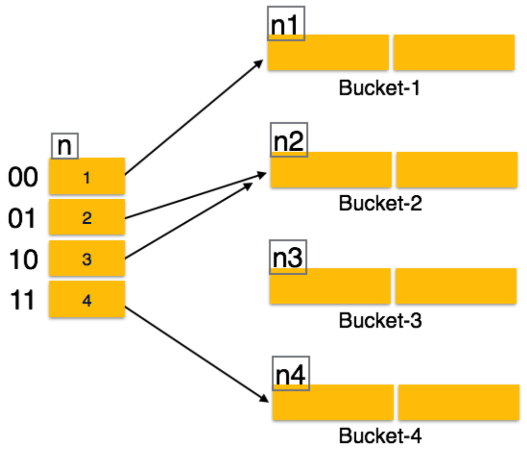
- **Linear Probing** − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.