**UNIT - 3**

***Ordered Indices***

Ordered indices in DBMS are **data structures that organize indexed data in a specific order, typically ascending or descending**. These indices **facilitate faster search and retrieval operations** based on the indexed column(s). Common types include B-tree, Binary Search Tree (BST), B+ Tree, Sorted Hash, and Skip List. They **enable efficient querying by maintaining a sorted arrangement of data, improving database performance particularly** **for range queries and sequential access**. Choosing the appropriate ordered index type depends on factors such as dataset size, query patterns, and DBMS capabilities, crucial for optimizing database performance.

There are various types of ordered indices, including:

1. **B-tree Index**: A B-tree (balanced tree) index is a widely used data structure for maintaining sorted data. It allows for efficient searching, insertion, and deletion operations. B-tree indices are commonly used in relational database management systems like MySQL, PostgreSQL, and Oracle.
2. **Binary Search Tree (BST) Index**: A binary search tree is a tree data structure where each node has at most two children, known as the left child and the right child. In a BST index, data is stored in a hierarchical order that enables fast search operations.
3. **B+ Tree Index**: Similar to a B-tree index, but with some variations in structure that optimize range queries and sequential access. B+ trees are commonly used in databases for their efficient range query performance.
4. **Sorted Hash Index**: In this type of index, the keys are hashed and stored in sorted order. It combines the benefits of hash-based and ordered indices, providing fast lookup while maintaining data order.
5. **Skip List**: A skip list is a probabilistic data structure that allows for efficient searching, insertion, and deletion operations in logarithmic time complexity. It's often used for implementing ordered indices in certain databases.

***Static Hashing vs Dynamic Hashing***

| **Aspect** | **Static Hashing** | **Dynamic Hashing** |
| --- | --- | --- |
| 1. Data Distribution | **Fixed number of buckets** determined during **initialization** | **Dynamically** **adjusts the number of buckets as needed** |
| 1. Bucket Overflow | **Challenge of Bucket overflow can arise** often depending upon memory size. | **Bucket overflow can occur very late** or doesn’t occur at all. |
| 1. Bucket Overflow Handling | **Fixed overflow handling mechanism** | **Overflow handling adjusts dynamically** |
| 1. Bucket Splits | **No bucket splits** required | **Bucket splits occur dynamically** based on **load** |
| 1. Performance | **Faster lookup due to fixed structure** | **Slightly slower lookup due to dynamic adjustments** |
| 1. Memory Usage | May lead to **inefficient memory usage** if bucket sizes are not optimized | **Optimizes memory usage** by adjusting bucket sizes dynamically |
| 1. Maintenance | **Requires occasional rehashing** if data distribution becomes uneven | **Requires less maintenance** as bucket sizes adjust dynamically |
| 1. Scalability | **Limited** **scalability** due to fixed bucket structure | **Highly scalable** as it dynamically adjusts bucket sizes |
| 1. Implementation Complexity | **Simpler implementation** with fixed bucket structure | **More complex implementation** due to dynamic adjustments |
| 1. Extensibility | **Limited extensibility**; **difficult to accommodate changes in data distribution** | **More adaptable to changes in data distribution**, accommodating growth more seamlessly |
| 1. Collision Resolution | **Relies on fixed collision resolution methods** | **Adapts collision resolution dynamically based on load** |
| 1. Form of Data | **Fixed-size**, **non-changing** data. | **Variable-size**, **changing** data. |

***Ordered Indexing vs Hashing***

| **Ordered Indexing** | **Hashing** |
| --- | --- |
| It is a technique that allows to quickly retrieve records from database file. | It is a technique that allows to search location of desired data on disk without using index structure. |
| It is generally used to optimize or increase performance of database simply by minimizing number of disk accesses that are required when a query is processed. | It is generally used to index and retrieve items in database as it is faster to search that specific item using shorter hashed key rather than using its original value. |
| It offers faster search and retrieval of data to users, helps to reduce table space, makes it possible to quickly retrieve or fetch data, can be used for sorting, etc. | It is faster than searching arrays and lists, provides more flexible and reliable method of data retrieval rather than any other data structure, can be used for comparing two files for quality, etc. |
| Its main purpose is to provide basis for both rapid random lookups and efficient access of ordered records. | Its main purpose is to use math problem to organize data into easily searchable buckets. |
| It is not considered best for large databases and its good for small databases. | It is considered best for large databases. |
| Types of indexing includes ordered indexing, primary indexing, secondary indexing, clustered indexing. | Types of hashing includes static and dynamic hashing. |
| It uses data reference to hold address of disk block. | It uses mathematical functions known as hash function to calculate direct location of records on disk. |
| It is important because it protects file and documents of large size business organizations, and optimize performance of database. | It is important because it ensures data integrity of files and messages, takes variable length string or messages and compresses and converts it into fixed length value. |

***B-Tree***

# **Introduction of B-Tree**

The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

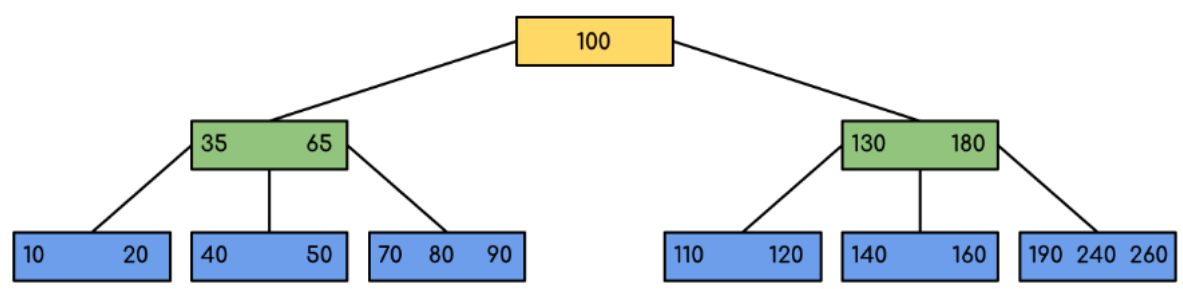
## ****Time Complexity of B-Tree:****

| **S.no.** | **Algorithm** | **Time Complexity** |
| --- | --- | --- |
| 1. | Search | O(log n) |
| 2. | Insert | O(log n) |
| 3. | Delete | O(log n) |

**Note:**“n” is the total number of elements in the B-tree

## ****Properties of B-Tree:****

* All leaves are at the same level.
* B-Tree is defined by the term minimum degree ‘**t**‘. The value of ‘**t**‘ depends upon disk block size.
* Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
* All nodes (including root) may contain at most (**2\*t – 1**) keys.
* Number of children of a node is equal to the number of keys in it plus **1**.
* All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
* B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
* Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
* Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of a B-Tree of minimum order 5   
**Note:** that in practical B-Trees, the value of the minimum order is much more than 5.   
 

We can see in the above diagram that all the leaf nodes are at the same level and all non-leafs have no empty sub-tree and have keys one less than the number of their children.

## ****Applications of B-Trees:****

* It is used in large databases to access data stored on the disk
* Searching for data in a data set can be achieved in significantly less time using the B-Tree
* With the indexing feature, multilevel indexing can be achieved.
* Most of the servers also use the B-tree approach.
* B-Trees are used in CAD systems to organize and search geometric data.
* B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

## ****Advantages of B-Trees:****

* B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
* B-Trees are self-balancing.
* High-concurrency and high-throughput.
* Efficient storage utilization.

## ****Disadvantages of B-Trees:****

* B-Trees are based on disk-based data structures and can have a high disk usage.
* Not the best for all cases.
* Slow in comparison to other data structures.

***B+Tree***

# **Introduction of B+ Tree**

**B + Tree** is a variation of the B-tree data structure. In a B + tree, **data pointers are stored only at the leaf nodes of the tree**. In a **B+ tree structure** of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record). The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

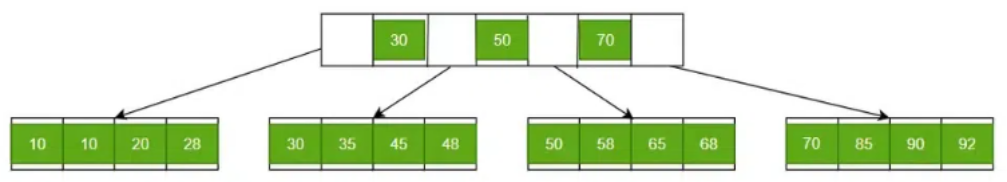
## Features of B+ Trees

* **Balanced:** B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.
* **Multi-level:** B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.
* **Ordered:**B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.
* **Fan-out:** B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.
* **Cache-friendly:** B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.
* **Disk-oriented:** B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

## Why Use B+ Tree?

* B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.
* B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

## Structure of B+ Trees



B+ Trees contain two types of nodes:

* **Internal Nodes:**Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
* **Leaf Nodes:**Leaf Nodes are the nodes that have n pointers.

## ****Application of B+ Trees****

* Multilevel Indexing
* Faster operations on the tree (insertion, deletion, search)
* Database indexing

## ****Advantages of B+Trees****

* A B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.
* Data stored in a B+ tree can be accessed both sequentially and directly.
* It takes an equal number of disk accesses to fetch records.
* B+trees have redundant search keys, and storing search keys repeatedly is not possible.

## ****Disadvantages of B+ Trees****

* The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

***Difference Between B+ Tree and B-Tree***

| **Parameters** | **B+ Tree** | **B-Tree** |
| --- | --- | --- |
| **Structure** | **Separate leaf nodes for data storage** and **internal nodes for indexing** | **Nodes** **store both keys and data values** |
| **Leaf Nodes** | Leaf nodes **form a linked list for efficient range-based queries** | Leaf nodes **do not form a linked list** |
| **Order** | **Higher order** (more keys) | **Lower order** (fewer keys) |
| **Key Duplication** | Typically **allows key duplication** in leaf nodes | Usually **does not allow key duplication** |
| **Disk Access** | **Better disk access due to sequential reads** in a linked list structure | **More disk I/O due to non-sequential reads** in internal nodes |
| **Applications** | **Database systems**, **file systems**, where **range queries** are common | **In-memory data structures, databases**, **general-purpose use** |
| **Performance** | Better performance for **range queries** and **bulk data retrieval** | Balanced performance for **search, insert,** and **delete operations** |
| **Memory Usage** | Requires **more memory for internal nodes** | Requires **less memory as keys and values are stored in the same node** |

***Index Definition in SQL***

An index is **a schema object**. It is **used to** **speed up the process of searching data in the database tables by using pointers**. It can **reduce disk I/O**(input/output) **by using a rapid path access method to locate data quickly**. The INDEX **requires its own space in the hard disk**. The index concept in SQL is **same** as the index concept in the **novel or a book**.

In SQL, an Index is **created on the fields of the tables**. We can **easily build one or more indexes on a table**. The **creation and deletion** **of the Index** **do not affect the data of the database**.

It is the **best SQL technique for improving the performance of queries**. The **drawback** of using indexes is that **they slow down the execution time of UPDATE and INSERT statements. But they have one advantage also as they speed up the execution time of SELECT and WHERE statements**.

***Why SQL Index?***

* SQL Indexes can **search the information of the large database quickly**.
* This data structure **sorts the data values of columns** (fields) either in **ascending or descending order**. And then, it assigns the entry for each value.
* Each Index table **contains only two columns**. The **first column is row\_id, and the other is indexed-column**.
* When indexes are used with **smaller tables**, the **performance** of the index **may not be recognized**.

***Create an INDEX***

**CREATE** **INDEX** Index\_Name **ON** Table\_Name ( Column\_Name);

Here, **Index\_Name** is the name of that index that we want to create, and **Table\_Name** is the name of the table on which the index is to be created. The **Column\_Name** represents the name of the column on which index is to be applied.

If we want to create an index on the combination of two or more columns, then the following syntax can be used in SQL:

**CREATE** **INDEX** Index\_Name **ON** Table\_Name ( column\_name1, column\_name2, ...., column\_nameN);

**Example for creating an Index in SQL:**

Let's take an Employee table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Emp\_Id** | **Emp\_Name** | **Emp\_Salary** | **Emp\_City** | **Emp\_State** |
| 1001 | Akshay | 20000 | Noida | U.P |
| 1002 | Ram | 35000 | Jaipur | Rajasthan |
| 1003 | Shyam | 25000 | Gurgaon | Haryana |
| 1004 | Yatin | 30000 | Lucknow | U.P |

The following SQL query creates an Index **'Index\_state'** on **the Emp\_State** column of the **Employee** table.

**CREATE** **INDEX** index\_state **ON** Employee (Emp\_State);

Suppose we want to create an index on the combination of the **Emp\_city** and the **Emp\_State** column of the above **Employee** table. For this, we have to use the following query:

**CREATE** **INDEX** index\_city\_State **ON** Employee (Emp\_City, Emp\_State);

***Create UNIQUE INDEX***

Unique Index is the **same as the Primary key** in SQL. The unique index **does not allow selecting those columns which contain duplicate values**.

This index is the best way to maintain the data integrity of the SQL tables.

**Syntax for creating the Unique Index is as follows:**

**CREATE** **UNIQUE** **INDEX** Index\_Name **ON** Table\_Name ( Column\_Name);

**Example for creating a Unique Index in SQL:**

Let's take the above Employee table. The following SQL query creates the unique index i**ndex\_salary** on the **Emp\_Salary** column of the **Employee** table.

**CREATE** **UNIQUE** **INDEX** index\_salary **ON** Employee (Emp\_Salary);

***Rename an INDEX***

We can easily rename the index of the table in the relational database using the ALTER command.

**ALTER** **INDEX** old\_Index\_Name RENAME **TO** new\_Index\_Name;

**Example for Renaming the Index in SQL:**

The following SQL query renames the index **'index\_Salary'** to **'index\_Employee\_Salary'** of the above Employee table:

**ALTER** **INDEX** index\_Salary RENAME **TO** index\_Employee\_Salary;

***Remove an INDEX***

An Index of the table can be easily removed from the SQL database using the DROP command. If you want to delete an index from the data dictionary, you must be the owner of the database or have the privileges for removing it.

**Syntaxes for Removing an Index in relational databases are as follows:**

**DROP** **INDEX** Index\_Name;

**Example for removing an Index in SQL:**

Suppose we want to remove the above **'index\_Salary'** from the SQL database. For this, we have to use the following SQL query:

**DROP** **INDEX** index\_salary;

***Alter an INDEX***

An index of the table can be easily modified in the relational database using the ALTER command.

**The basic syntax for modifying the Index in SQL is as follows:**

**ALTER** **INDEX** Index\_Name **ON** Table\_Name REBUILD;

***When should INDEXES not be used in SQL?***

* SQL Indexes can be **avoided when the size of the table is small**.
* When the **table needs to be updated frequently**.
* Indexed should **not be used** on those cases when the **column of a table contains a large number of NULL values**.

***Hashing in DBMS***

Hashing in DBMS is a technique to quickly locate a data record in a database irrespective of the size of the database. For larger databases containing thousands and millions of records, the indexing data structure technique becomes very inefficient because searching a specific record through indexing will consume more time. This doesn’t align with the goals of DBMS, especially when performance and date retrieval time are minimized. So, to counter this problem hashing technique is used.

## What is Hashing?

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.

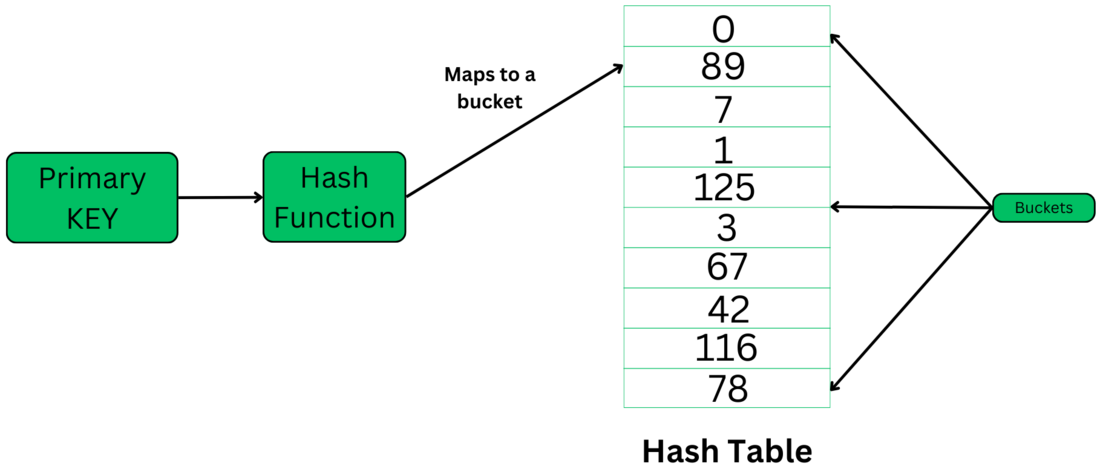
**Now, there are 2 possibilities:**

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:**



***Hashing***

## Types of Hashing in DBMS

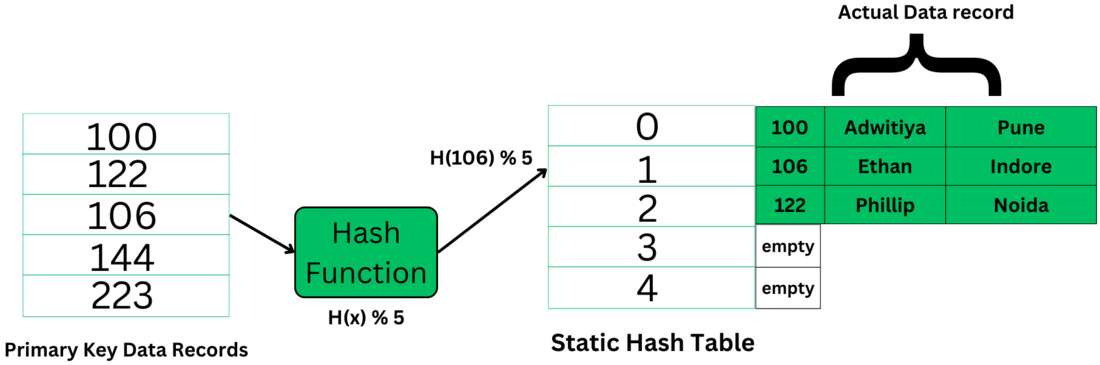
There are two primary hashing techniques in [DBMS](https://www.geeksforgeeks.org/dbms/).

## 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 = id. Then the operation will take place like this:

H(106) % 5 = 1.  
This indicates that the data record should be placed or searched in the 1st bucket (or 1st hash index) in the hash table.

**Example:**



***Static Hashing Technique***

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.

### Static Hashing has the following Properties

* **Data Buckets:** The number of buckets in memory remains constant. The size of the hash table is decided initially and it may also implement chaining that will allow handling some collision issues though, it’s only a slight optimization and may not prove worthy if the database size keeps fluctuating.
* **Hash function:** It uses the simplest hash function to map the data records to its appropriate bucket. It is generally modulo-hash function
* **Efficient for known data size:**It’s very efficient in terms when we know the data size and its distribution in the database.
* It is inefficient and inaccurate when the data size dynamically varies because we have limited space and the hash function always generates the same value for every specific input. When the data size fluctuates very often it’s not at all useful because collision will keep happening and it will result in problems like – bucket skew, insufficient buckets etc.

To resolve this problem of bucket overflow, techniques such as – chaining and open addressing are used. Here’s a brief info on both:

### a.Chaining

[Chaining](https://www.geeksforgeeks.org/separate-chaining-collision-handling-technique-in-hashing/) is a mechanism in which the hash table is implemented using an array of type nodes, where each bucket is of node type and can contain a long chain of linked lists to store the data records. So, even if a hash function generates the same value for any data record it can still be stored in a bucket by adding a new node.

However, this will give rise to the problem bucket skew that is, if the hash function keeps generating the same value again and again then the hashing will become inefficient as the remaining data buckets will stay unoccupied or store minimal data.

### b.Open Addressing/Closed Hashing

This is also called closed hashing this aims to solve the problem of collision by looking out for the next empty slot available which can store data. It uses techniques like **linear probing**,**quadratic probing**,**double hashing,** etc.

## 2.Dynamic Hashing

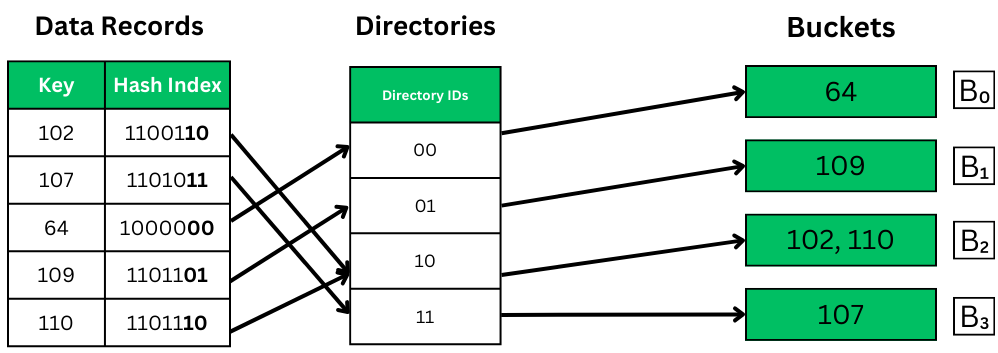
Dynamic hashing is also known as [extendible hashing](https://www.geeksforgeeks.org/extendible-hashing-dynamic-approach-to-dbms/), used to handle database that frequently changes data sets. This method offers us a way to add and remove data buckets on demand dynamically. This way as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.

### Properties of Dynamic Hashing

* The buckets will vary in size dynamically periodically as changes are made offering more flexibility in making any change.
* Dynamic Hashing aids in improving overall performance by minimizing or completely preventing collisions.
* **It has the following major components:**Data bucket, Flexible hash function, and directories
* A flexible hash function means that it will generate more dynamic values and will keep changing periodically asserting to the requirements of the database.
* Directories are containers that store the pointer to buckets. If bucket overflow or bucket skew-like problems happen to occur, then bucket splitting is done to maintain efficient retrieval time of data records. Each directory will have a directory id.
* **Global Depth:**It is defined as the number of bits in each directory id. The more the number of records, the more bits are there.

### 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.

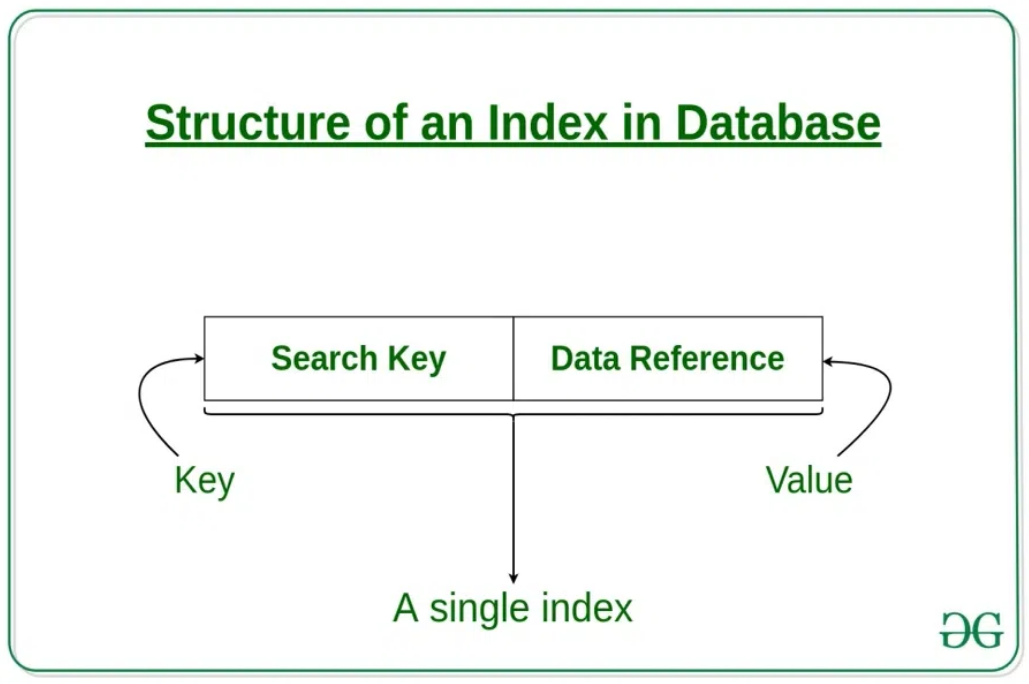


***Dynamic Hashing – mapping***

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.

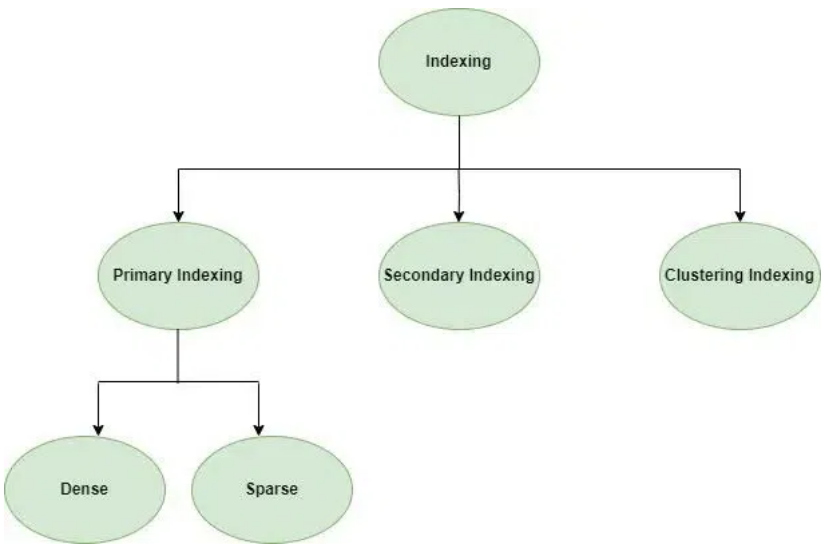
***Indexing in DBMS***

It is a **data structure technique used to locate and quickly access data in databases**. Indexing **improves database performance by minimizing the number of disc visits required to fulfill a query**. The **main key or candidate key** **of the table** is **duplicated in the first column**, which is the **Search key**. To speed up data retrieval, the values are also kept in sorted order. It should be highlighted that **sorting the data is not required**. 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**.



## Attributes of Indexing

* **Access Types:** This refers to the **type of access** such as **value-based search**, **range access**, etc.
* **Access Time:** It refers to **the time needed to find a particular data element or set of elements**.
* **Insertion Time:** It refers to **the time taken to find the appropriate space and insert new data**.
* **Deletion Time:** **Time taken to find an item and delete it** as well as **update the index structure**.
* **Space Overhead:** It refers to **the additional space required by the index**.

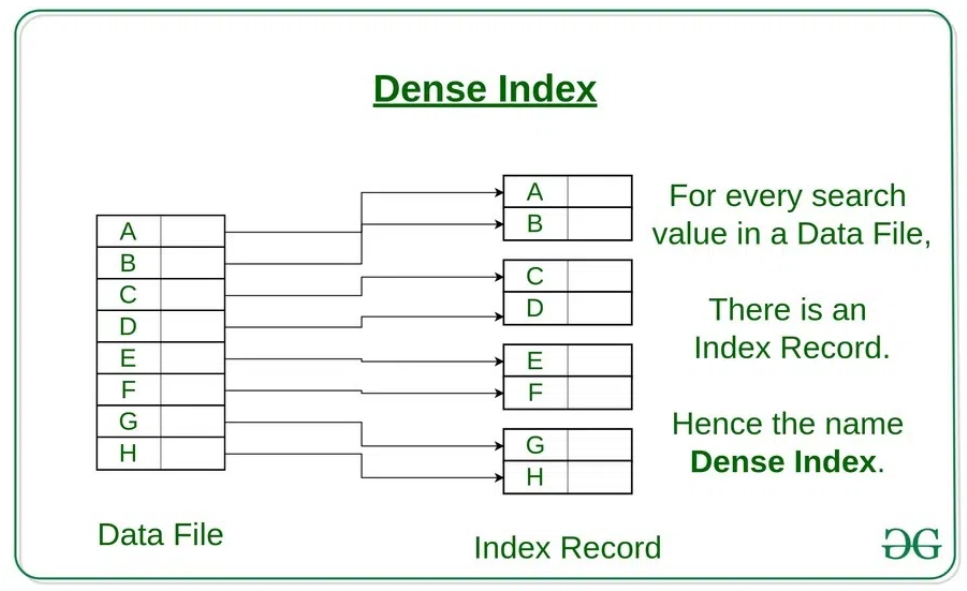


In general, there are **two types of file organization mechanisms** that are **followed by the indexing methods to store the data**:

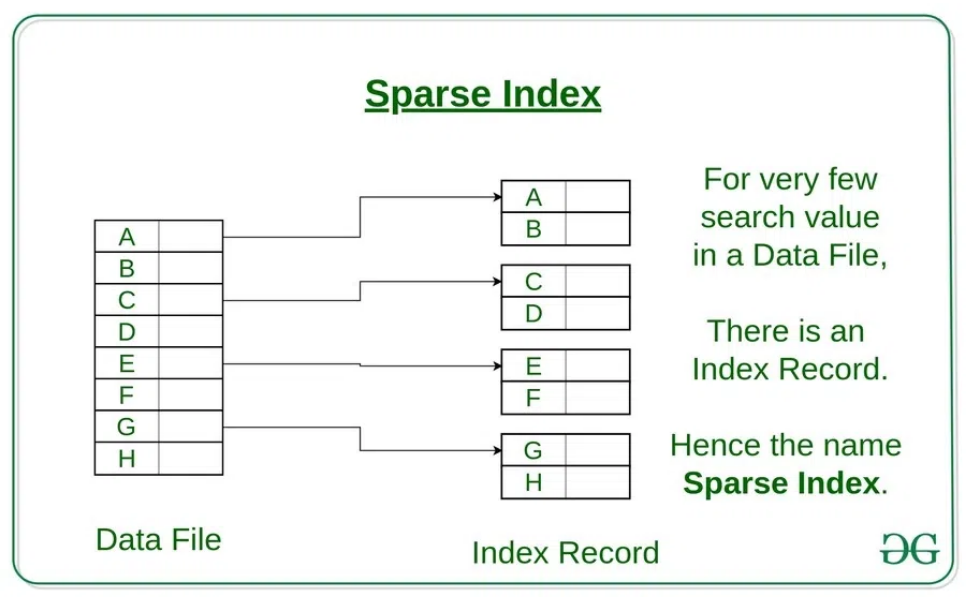
## ****Sequential File Organization or Ordered Index File****

In this, the indices are **based on a sorted ordering of the values**. These are **generally fast and a more traditional type of storing mechanism**. These Ordered or Sequential file organizations might store the data in a **dense or sparse format**.

1. **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.



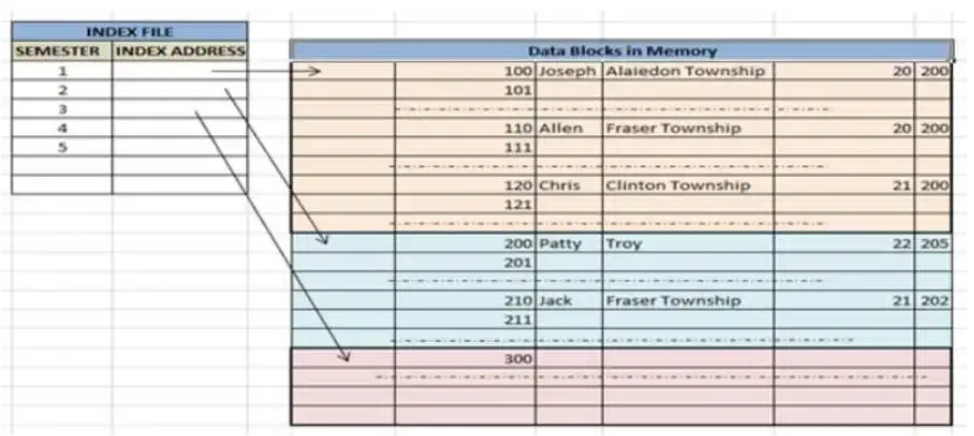
1. **Sparse Index** 
   * The **index record appears only for a few items in the data file**. Each item points to a block as shown.
   * To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
   * We start at that 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.
   * Number of Accesses required=log₂(n)+1, (here n=number of blocks acquired by index file)



## ****Hash File Organization****

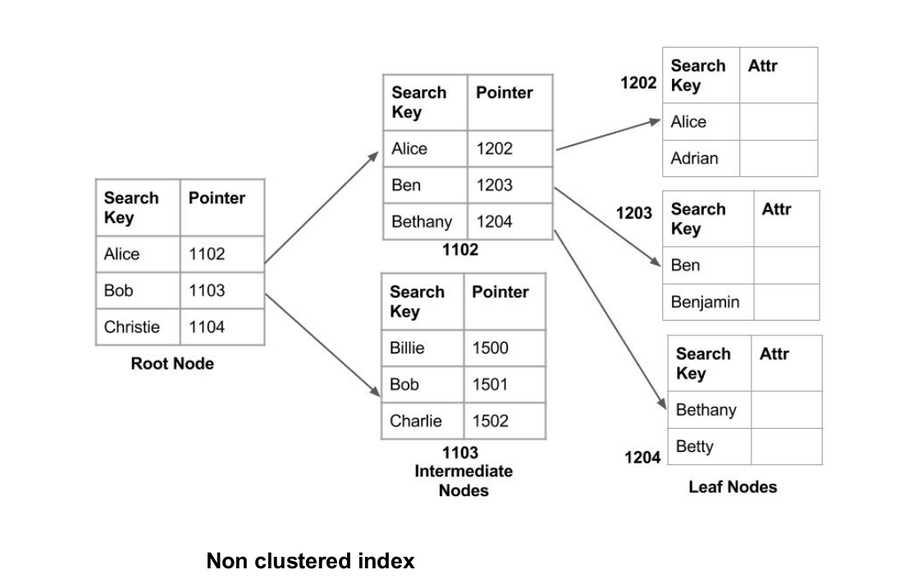
Indices are based on the values being **distributed uniformly across a range of buckets**. The **buckets to which a value is assigned are determined by a function** called a **hash function**. There are primarily three methods of indexing:

* **Clustered Indexing**: When **more than two records are stored in the same file** this type of storing is known as **cluster indexing**. By using cluster indexing we can **reduce the cost of searching** reason being multiple records related to the same thing are stored in one place and it also gives the frequent joining of more than two tables (records).   
  The 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 an index out of them. This method is known as the clustering index. Essentially, records with similar properties are grouped together, and indexes for these groupings are formed.   
  Students studying each semester, for example, are grouped together. First-semester students, second-semester students, third-semester students, and so on are categorized.

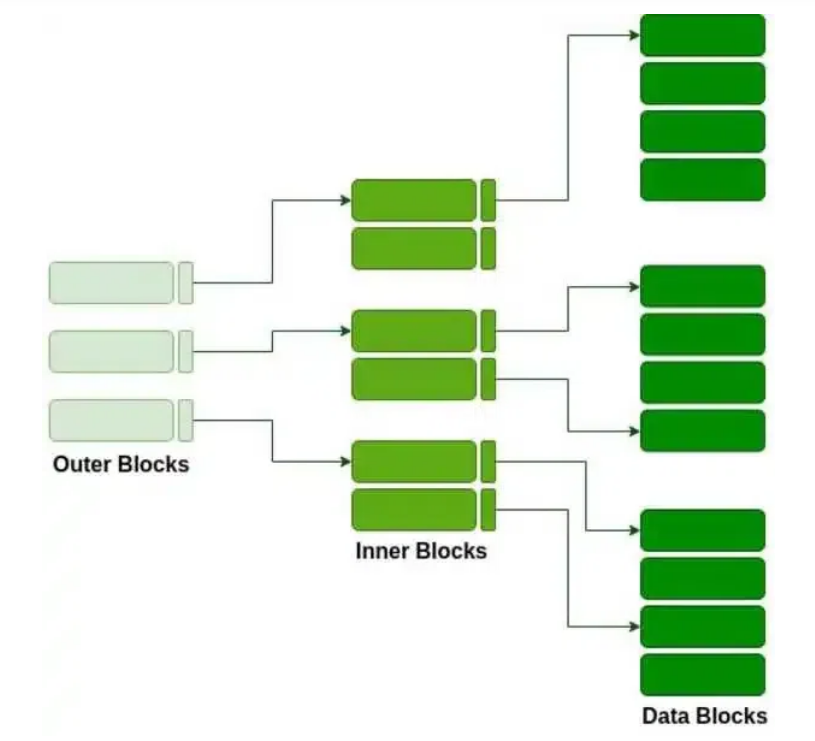
****

***Clustered Indexing***

* **Primary Indexing:**This is a type of Clustered Indexing wherein **the data is sorted according to the search key and the primary key of the database table** is used to create the index. It is a default format of indexing where it induces sequential file organization. **As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient**.
* **Non-clustered or Secondary Indexing**: A non-clustered index just tells us where the data lies, i.e. it **gives us a list of virtual pointers or references to the location where the data is actually stored**. **Data is not physically stored in the order of the index**. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly. It **requires more time as compared to the clustered index** because some amount of extra work is done in order to extract the data by further following the pointer.



* **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 be 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**.



***Multilevel Indexing***

## Advantages of Indexing

* **Improved Query Performance:** Indexing **enables faster data retrieval from the database**. The database may rapidly discover rows that match a specific value or collection of values by generating an index on a column, minimizing the amount of time it takes to perform a query.
* **Efficient Data Access:** Indexing can **enhance data access efficiency by lowering the amount of disk I/O required to retrieve data**. The database can maintain the data pages for frequently visited columns in memory by generating an index on those columns, decreasing the requirement to read from disk.
* **Optimized Data Sorting:**Indexing can also **improve the performance of sorting operations**. By creating an index on the columns used for sorting, the database can avoid sorting the entire table and instead sort only the relevant rows.
* **Consistent Data Performance:** Indexing can **ensure that the database performs consistently even as the amount of data in the database rises**. Without indexing, queries may take longer to run as the number of rows in the table grows, while indexing maintains a roughly consistent speed.
* **Enforced Data Integrity**: By **ensuring that only unique values are inserted into columns that have been indexed as unique**, indexing can also be utilized to ensure the integrity of data. This **avoids storing duplicate data in the database**, which might lead to issues when performing queries or reports.

## Disadvantages of Indexing

* Indexing necessitates **more storage space to hold the index data structure**, which might increase the total size of the database.
* **Increased database maintenance overhead:** Indexes must be maintained as data is added, destroyed, or modified in the table, which might raise database maintenance overhead.
* Indexing can **reduce insert and update performance** since the index data structure must be updated each time when the data is modified.
* **Choosing an index can be difficult:** It can be **challenging to choose the right indexes for a specific query or application** and may call for a detailed examination of the data and access patterns.

## Features of Indexing

* The development of data structures, such as B-trees or hash tables, that provide quick access to certain data items is known as indexing. The data structures themselves are built on the values of the indexed columns, which are utilized to quickly find the data objects.
* The most important columns for indexing columns are selected based on how frequently they are used and the sorts of queries they are subjected to. The cardinality, selectivity, and uniqueness of the indexing columns can be taken into account.
* There are several different index types used by databases, including primary, secondary, clustered, and non-clustered indexes. Based on the particular needs of the database system, each form of index offers benefits and drawbacks.
* For the database system to function at its best, periodic index maintenance is required. According to changes in the data and usage patterns, maintenance work involves building, updating, and removing indexes.
* Database query optimization involves indexing, which is essential. The query optimizer utilizes the indexes to choose the best execution strategy for a particular query based on the cost of accessing the data and the selectivity of the indexing columns.
* Databases make use of a range of indexing strategies, including covering indexes, index-only scans, and partial indexes. These techniques maximize the utilization of indexes for particular types of queries and data access.
* When non-contiguous data blocks are stored in an index, it can result in index fragmentation, which makes the index less effective. Regular index maintenance, such as defragmentation and reorganization, can decrease fragmentation.

## Conclusion

Indexing is a very useful technique that helps in optimizing the search time in database queries. The table of database indexing consists of a search key and pointer. There are four types of indexing: Primary, Secondary, Clustering, and Multivalued Indexing. Primary indexing is divided into two types, dense and sparse. Dense indexing is used when the index table contains records for every search key. Sparse indexing is used when the index table does not use a search key for every record. Multilevel indexing uses B+Tree. The main purpose of indexing is to provide better performance for data retrieval.

***Query Processing & Optimization***

Query Processing is the **activity performed** in extracting data from the database. **Query Processing** includes **translation of high level Queries into low level expressions that can be used at physical level of file system, query optimization and actual execution of query to get the actual result**. In query processing, it takes various steps for fetching the data from the database. The steps involved are:

1. Parsing and translation
2. Optimization
3. Evaluation

## *1. Parsing and Translation*

Initially, the user queries get **translated in high-level database languages** such as SQL. Later, it gets **translated into expressions that can be further used at the physical level of the file system**. After this, the actual evaluation of the queries and a variety of query-optimizing transformations takes place. **SQL is not perfectly** **suitable** for the **internal representation of the query to the system**. Relational algebra is **well suited** for the internal representation of a query. When a user executes any query, for generating the internal form of the query, the **parser performs** the following **checks** :-

* **Syntax check –** **checks** the **syntax of the query**. Example:

SELECT \* FORM employee

Here error of wrong spelling of FROM is given by this check.

* **Semantic check –** **determines** **whether** the **statement is meaningful or not**. Example: query contains a table name which does not exist is checked by this check.
* **Shared Pool check –** Every query **possess** a **hash code during its execution**. So, this check **determines existence of written hash code in shared pool** if code exists in shared pool then database will not take additional steps for optimization and execution.

## *2. Optimization*

* The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
* Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query Optimization.
* For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends on the memory allocations to several operations, execution costs, and so on.

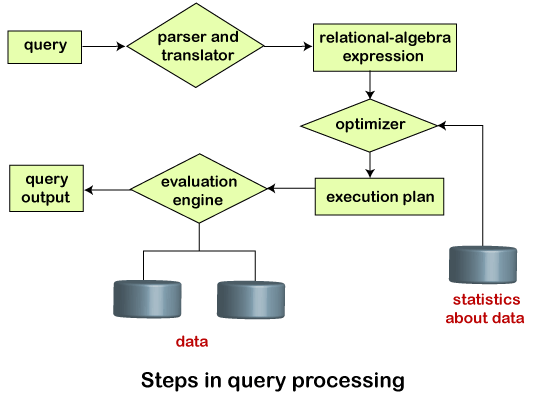
## *3. Evaluation*

For this, with addition to the relational algebra translation, it is required to annotate the translated relational algebra expression with the instructions used for specifying and evaluating each operation. Thus, after translating the user query, the system executes a query evaluation plan.

### Query Evaluation Plan

* In order to fully evaluate a query, the system needs to construct a query evaluation plan.
* The annotations in the evaluation plan may refer to the algorithms to be used for the particular index or the specific operations.
* Such relational algebra with annotations is referred to as **Evaluation Primitives**. The evaluation primitives carry the instructions needed for the evaluation of the operation.
* Thus, a query evaluation plan defines a sequence of primitive operations used for evaluating a query. The query evaluation plan is also referred to as **the query execution plan**.
* A **query execution engine** is responsible for generating the output of the given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

Finally, after selecting an evaluation plan, the system evaluates the query and produces the output of the query.



***Measures of Query Cost in DBMS***

**Cost of query** is the **time taken by the query to hit the database and return the result**. It involves **query processing time** i.e. time taken to parse and translate the query, optimize it, evaluate, execute and return the result to the user. Though it is in fraction of seconds, it includes multiple sub tasks and time taken by each of them.

In other words, Query cost is **what optimizer thinks of how long your query will take** (relative to total batch time). The **optimizer** tries to **choose the optimal query plan by looking at the query and statistics of the data, trying several execution plans and selecting the least costly of them**.

The expense assessment of a query evaluation plan is determined by keeping in mind the following aspects :-

1. The number **of disk accesses**.
2. **Execution time** taken by the CPU to execute a query.
3. The **Communication costs** involved in distributed or parallel database systems.

***Selection Operation***

The SELECT Statement in SQL is **used to retrieve or fetch data from a database**.

We can **fetch either the entire table or according to some specified rules**. The **data returned is stored in a result table**. This result table is **also called the result set**. With the SELECT clause of a SELECT command statement, we **specify the columns that we want to be displayed in the query result** and, optionally, which column headings we prefer to see above the result table.

**CREATE TABLE:**

CREATE TABLE Customer(

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(50),

LastName VARCHAR(50),

Country VARCHAR(50),

Age int(2),

Phone int(10)

);

-- Insert some sample data into the Customers table

INSERT INTO Customer (CustomerID, CustomerName, LastName, Country, Age, Phone)

VALUES (1, 'Shubham', 'Thakur', 'India','23','xxxxxxxxxx'),

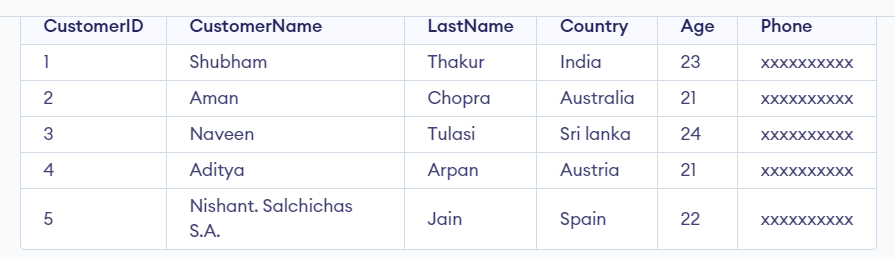
(2, 'Aman ', 'Chopra', 'Australia','21','xxxxxxxxxx'),

(3, 'Naveen', 'Tulasi', 'Sri lanka','24','xxxxxxxxxx'),

(4, 'Aditya', 'Arpan', 'Austria','21','xxxxxxxxxx'),

(5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain','22','xxxxxxxxxx');

**Output:**



To fetch any column in the table.

**Syntax:**

*SELECT column1,column2*

*FROM table\_name*

*column1 , column2: names of the fields of the table*

*table\_name: from where we want to apply query*

This query will return all the rows in the table with fields column1 and column2.

## SELECT Statement in SQL

To fetch the entire table or all the fields in the table:

**Syntax:**

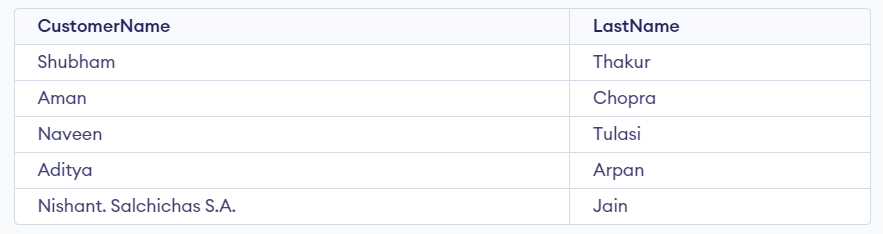
*SELECT \* FROM table\_name;*

*— asterisks represent all attributes of the table*

Query to fetch the fields CustomerName, LastName from the table Customer:

SELECT CustomerName, LastName FROM Customer;

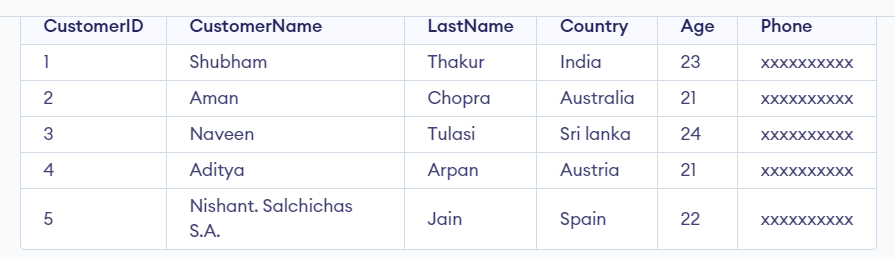
**Output:**



To fetch all the fields from the table Customer:

SELECT \* FROM Customer;

**Output:**



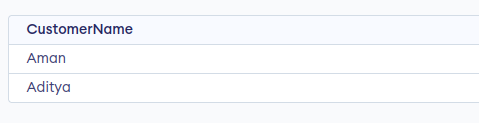
## SELECT Statement with WHERE Clause

Suppose we want to see table values with specific conditions then Where Clause is used with select statement.

**Query:**

SELECT CustomerName FROM Customer where Age = '21';

**Output:**

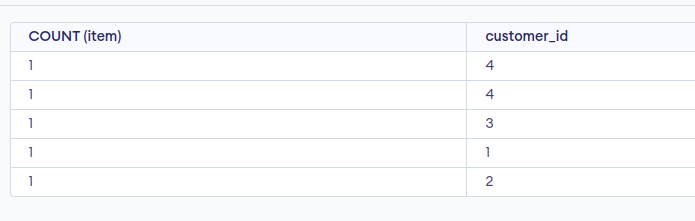


## SQL SELECT Statement with GROUP BY Clause

**Query:**

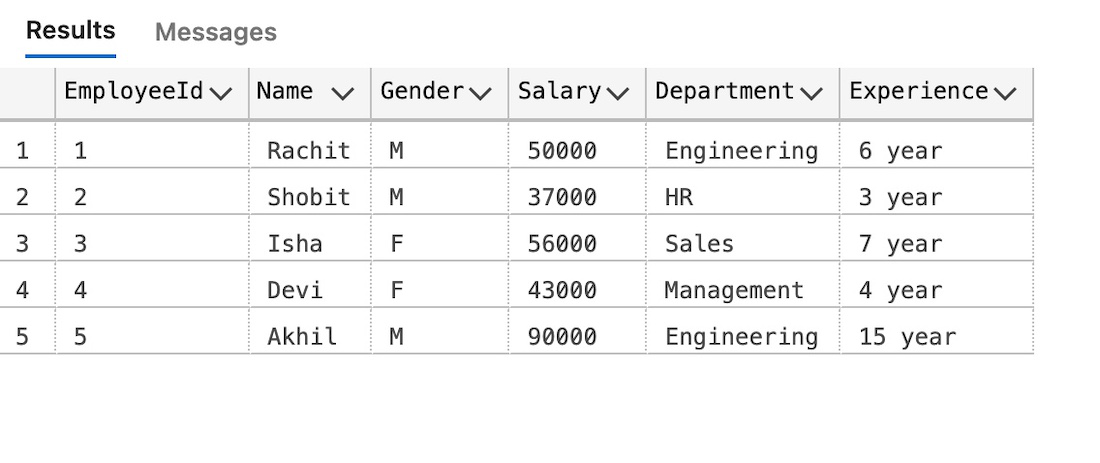
SELECT COUNT (item), Customer\_id FROM Orders GROUP BY order\_id;

**Output:**



## SELECT Statement with HAVING Clause

Consider the following database for Having Clause:



**Query:**

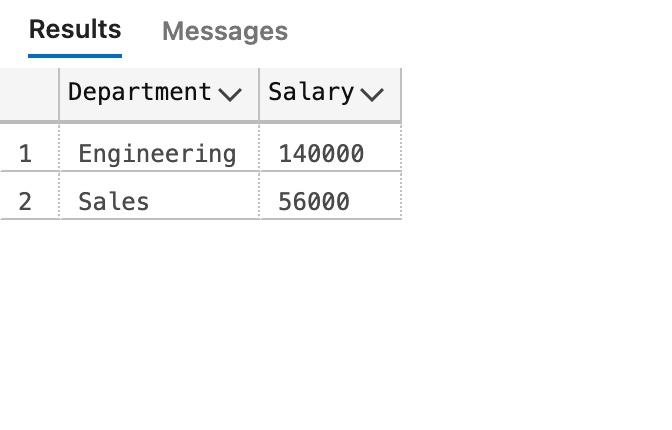
SELECT Department, sum(Salary) as Salary

FROM employee

GROUP BY department

HAVING SUM(Salary) >= 50000;

**Output:**

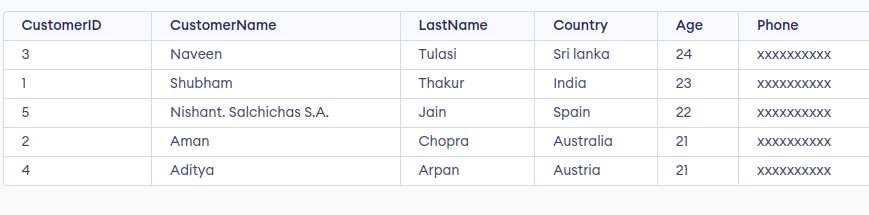


## SELECT Statement with ORDER BY clause in SQL

**Query:**

SELECT \* FROM Customer ORDER BY Age DESC;

**Output:**



***Sorting***

The SQL **ORDER BY** clause is **used to sort the data in ascending or descending order**, **based on one or more columns**. By default, some databases sort the query results in an ascending order.

In addition to that, ORDER BY clause can also sort the data in a database table in a preferred order. This case may not sort the records of a table in any standard order (like alphabetical or lexicographical), but, they could be sorted based on any external condition. For instance, in an ORDERS table containing the list of orders made by various customers of an organization, the details of orders placed can be sorted based on the dates on which those orders are made. This need not be alphabetically sorted, instead, it is based on "first come first serve".

Syntax

The basic syntax of the ORDER BY clause which would be used to sort the result in an ascending or descending order is as follows −

SELECT column-list

FROM table\_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

You can use more than one column in the ORDER BY clause. Make sure that whatever column you are using to sort, that column should be in the column-list.

Sorting Results in Ascending Order

Using Order By Clause in SQL, the records in a database table can be sorted in ascending order, either by default or by specifying the "ASC" keyword in the clause condition. Let us see an example to understand this.

Example

Assume we have created a table named **CUSTOMERS** using the CREATE TABLE statement as shown below −

CREATE TABLE CUSTOMERS (

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Now, insert values into this table using the INSERT statement as follows −

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES

(1, 'Ramesh', 32, 'Ahmedabad', 2000.00 ),

(2, 'Khilan', 25, 'Delhi', 1500.00 ),

(3, 'kaushik', 23, 'Kota', 2000.00 ),

(4, 'Chaitali', 25, 'Mumbai', 6500.00 ),

(5, 'Hardik', 27, 'Bhopal', 8500.00 ),

(6, 'Komal', 22, 'Hyderabad', 4500.00 ),

(7, 'Muffy', 24, 'Indore', 10000.00 );

The table will be created as −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |

Following is an example, which would sort the result in an ascending order by NAME and SALARY.

SELECT \* FROM CUSTOMERS ORDER BY NAME;

Output

This would produce the following result −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

Sorting Results in Descending Order

But, to sort the records in a database table in descending order, we need to specify the "DESC" keyword in the clause condition. Let us see an example to understand this.

Example

The following query sorts the records of the CUSTOMERS tables in descending order based on the column **NAME**.

SELECT \* FROM CUSTOMERS ORDER BY NAME DESC;

Output

This would produce the following result −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
| 6 | Komal | 22 | Hyderabad | 4500.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |

***Join Operations***

**SQL Join** statement is **used to combine data or rows from two or more tables based on a common field between them**. Different types of Joins are as follows:

* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL JOIN
* NATURAL JOIN

Consider the two tables below as follows:

**Student**



**StudentCourse**



### ****A. INNER JOIN****

The INNER JOIN keyword **selects all rows from both the tables as long as the condition is satisfied**. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e **value of the common field will be the same**.

**Syntax**:

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
INNER JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
**table1**: First table.  
**table2**: Second table  
**matching\_column**: Column common to both the tables.

***Note****: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*



**Example Queries(INNER JOIN)**

This query will show the names and age of students enrolled in different courses.

SELECT StudentCourse.COURSE\_ID, Student.NAME, Student.AGE FROM Student  
INNER JOIN StudentCourse  
ON Student.ROLL\_NO = StudentCourse.ROLL\_NO;

**Output**:



### ****B. LEFT JOIN****

This join **returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join**. **For the rows for which there is no matching row on the right side, the result-set will contain null**. LEFT JOIN is also known as LEFT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
LEFT JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
table1: First table.  
table2: Second table  
matching\_column: Column common to both the tables.

***Note****: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*



**Example Queries(LEFT JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID   
FROM Student  
LEFT JOIN StudentCourse   
ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output**:



### ****C. RIGHT JOIN****

RIGHT JOIN is **similar to LEFT JOIN**. This join **returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join**. **For the rows for which there is no matching row on the left side, the result-set will contain null**. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
RIGHT JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
  
table1: First table.  
table2: Second table  
matching\_column: Column common to both the tables.

***Note****: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*



**Example Queries(RIGHT JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID   
FROM Student  
RIGHT JOIN StudentCourse   
ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**



### ****D. FULL JOIN****

FULL JOIN **creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables**. For the rows for which there is no matching, the result-set will contain NULL values.



**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1   
FULL JOIN table2  
ON table1.matching\_column = table2.matching\_column;  
  
table1: First table.  
table2: Second table  
matching\_column: Column common to both the tables.

**Example Queries(FULL JOIN)**:

SELECT Student.NAME,StudentCourse.COURSE\_ID   
FROM Student  
FULL JOIN StudentCourse   
ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**

| **NAME** | **COURSE\_ID** |
| --- | --- |
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | NULL |
| ROHIT | NULL |
| NIRAJ | NULL |
| NULL | 4 |
| NULL | 5 |
| NULL | 4 |

### ****E. Natural join (?)****

Natural join can **join tables based on the common columns** in the tables being joined. A natural join **returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables**.

Both table must have at list one common column with same column name and same data type.

| **Employee** | | |
| --- | --- | --- |
| **Emp\_id** | **Emp\_name** | **Dept\_id** |
| **1** | **Ram** | **10** |
| **2** | **John** | **30** |
| **3** | **Bob** | **50** |

| **Department** | |
| --- | --- |
| **Dept\_id** | **Dept\_name** |
| **10** | **IT** |
| **30** | **HR** |
| **40** | **TIS** |

**Query**: Find all Employees and their respective departments.

**Solution**: (Employee) ? (Department)

| **Emp\_id** | **Emp\_name** | **Dept\_id** | **Dept\_name** |
| --- | --- | --- | --- |
| **1** | **Ram** | **10** | **IT** |
| **2** | **John** | **30** | **HR** |

