**INDEXING IN DBMS**

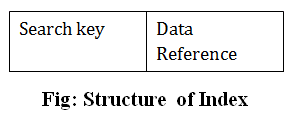
What is Indexing in DBMS

* The index is a type of data structure. It is used to locate and access the data in a database table quickly.
* Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

## 

## Index structure:

Indexes can be created using some database columns.



* The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
* The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

The indexing has various attributes:

* **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 particular data element or set of elements.
* **Insertion Time**: It refers to the time taken to find the appropriate space and insert a 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.

## What Does Indexing Do?

Indexing is the way to get an unordered table into an order that will maximize the query’s efficiency while searching

How Does Indexing Work ?

When a table is un-indexed, the order of the rows will likely not be discernible by the query as optimized in any way, and your query will therefore have to search through the rows linearly. In other words, the queries will have to search through every row to find the rows matching the conditions. As you can imagine, this can take a long time. Looking through every single row is not very efficient.

For example, the table below represents a table in a fictional data source, that is completely unordered.

If we were to run the following query:

SELECT

| **COMPANY\_ID** | **UNIT** | **UNIT\_COST** |
| --- | --- | --- |
| 10 | 12 | 1.15 |
| 12 | 12 | 1.05 |
| 14 | 18 | 1.31 |
| 18 | 18 | 1.34 |
| 11 | 24 | 1.15 |
| 16 | 12 | 1.31 |
| 10 | 12 | 1.15 |
| 12 | 24 | 1.3 |
| 18 | 6 | 1.34 |
| 18 | 12 | 1.35 |
| 14 | 12 | 1.95 |
| 21 | 18 | 1.36 |
| 12 | 12 | 1.05 |
| 20 | 6 | 1.31 |
| 18 | 18 | 1.34 |
| 11 | 24 | 1.15 |
| 14 | 24 | 1.05 |

company\_id,

units,

unit\_cost

FROM

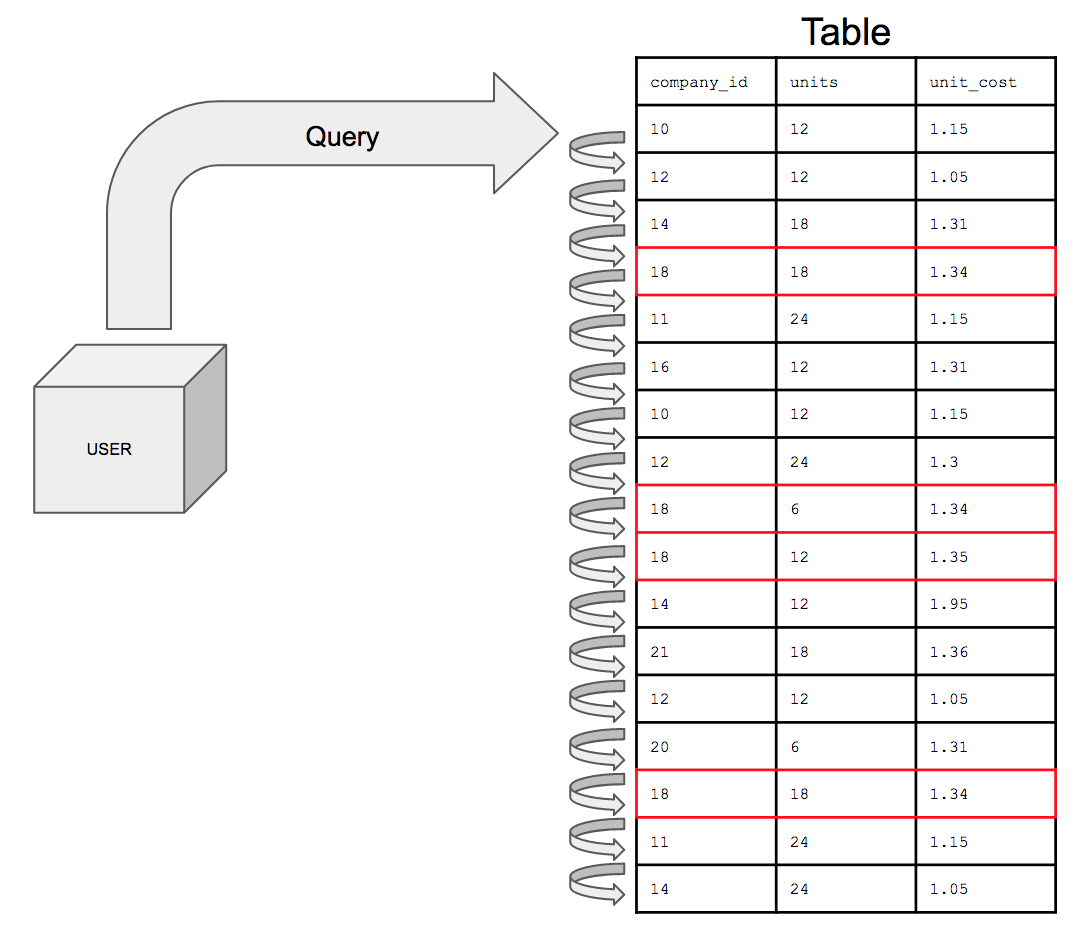
index\_test

WHERE

company\_id = 18

The database would have to search through all rows in the order they appear in the table, from top to bottom, one at a time.

This will only get more and more time consuming as the size of the table increases.

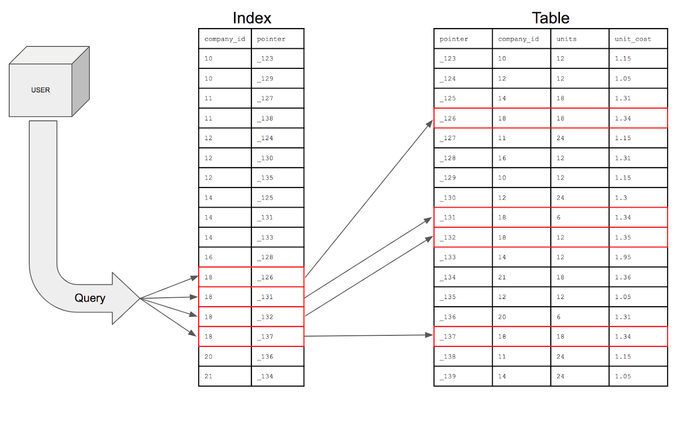


What indexing does is sets up the column you’re search conditions are on in a sorted order to assist in optimizing query performance.

With an index on the company\_id column, the table would, essentially, “look” like this:

| **COMPANY\_ID** | **UNIT** | **UNIT\_COST** |
| --- | --- | --- |
| 10 | 12 | 1.15 |
| 10 | 12 | 1.15 |
| 11 | 24 | 1.15 |
| 11 | 24 | 1.15 |
| 12 | 12 | 1.05 |
| 12 | 24 | 1.3 |
| 12 | 12 | 1.05 |
| 14 | 18 | 1.31 |
| 14 | 12 | 1.95 |
| 14 | 24 | 1.05 |
| 16 | 12 | 1.31 |
| 18 | 18 | 1.34 |
| 18 | 6 | 1.34 |
| 18 | 12 | 1.35 |
| 18 | 18 | 1.34 |
| 20 | 6 | 1.31 |
| 21 | 18 | 1.36 |

Now, the database can search for company\_id number 18 and return all the requested columns for that row then move on to the next row. If the next row’s comapny\_id number is also 18 then it will return the all the columns requested in the query. If the next row’s company\_id is 20, the query knows to stop searching and the query will finish.



## Why Is Indexing Used in the Database?

Imagine you need to store a list of numbers in a file and search a given number on that list. The simplest solution is to store data in an array and append values when new values come. But if you need to check if a given value exists in the array, then you need to search through all of the array elements one by one and check whether the given value exists. If you are lucky enough, you can find the given value in the first element. In the worst case, the value can be the last element in the array. We can denote this worst case as O(n) in asymptotic notation. This means if your array size is “n,” at most, you need to do “n” number of searches to find a given value in an array.

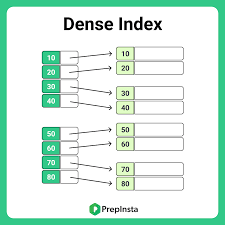
Types of indexing methods

**Primary Indexing**

* Primary Index is***an ordered file which has fixed length size with two fields.***
* The ***first field*** is the same a***primary key*** and ***second field*** is ***pointed to that specific data block***.
* In the primary index, there is always ***one to one relationship*** between the entries in the index table.
* Primary Indexing is ***further divided into two types***.
  + **Dense Index**
  + **Sparse Index**

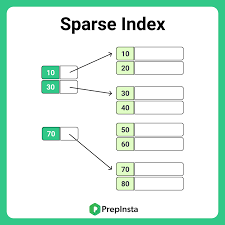
#### **Dense Index**

* In a dense index, a ***record is created for every search key*** valued in the database.
* Dense indexing helps you to***search faster but needs more space*** to store index records.
* In dense indexing, records contain search key value and ***points to the real record on the disk***.



#### **Sparse Index**

* The sparse index is an index record that ***appears for only some of the values*** in the file.
* Sparse Index helps you to ***resolve the issues of dense indexing.***
* In sparse indexing technique, ***a range of index columns stores the same data block address***, and when data needs to be retrieved, this block address will be fetched.
* Sparse indexing method ***stores index records for only some search key values***.
* It needs***less space, less maintenance overhead for insertion, and deletions*** but it is ***slower compared to the dense index for locating records.***



**Cluster Indexing**

* In a clustered index, ***records themselves are stored in the index and not pointers.***
* Sometimes the***index is created on non-primary key*** columns which might not be unique for each record. In such a situation, you ***can group two or more columns to get the unique values and create an index*** which is called clustered Index.
* This also ***helps you to identify the record faster***.

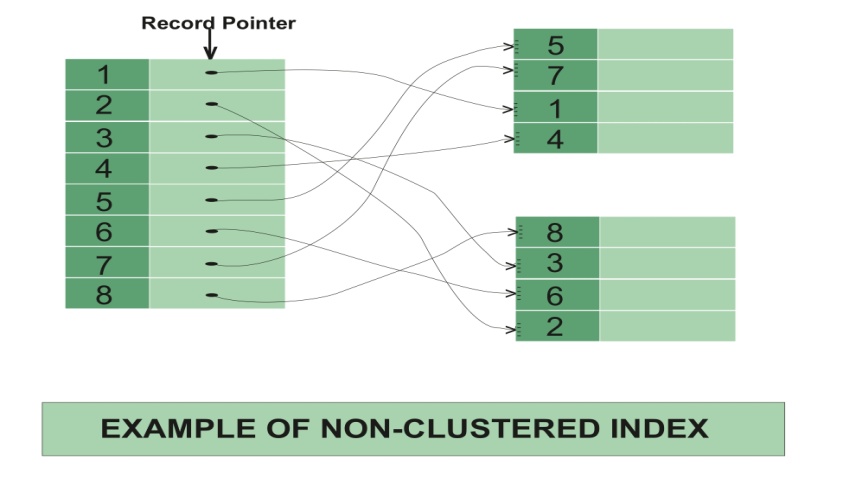
#### Difference between Clustered and Non-clustered index - GeeksforGeeks

#### ****Example****

* Consider a company recruited many employees in various departments. In this case, ***clustering index should be created for all employees who belong to the same dept.***
* In a single cluster it is considered that an index points to the cluster as a whole.

**Secondary Indexing**

* The secondary index can be generated by ***a field which has a unique value for each record***.
* It is also known as a ***non-clustering index.***
* This two-level database indexing technique is ***used to reduce the mapping size of the first level.***
* For the first level, a large range of numbers is selected, because of this mapping size always remains small.

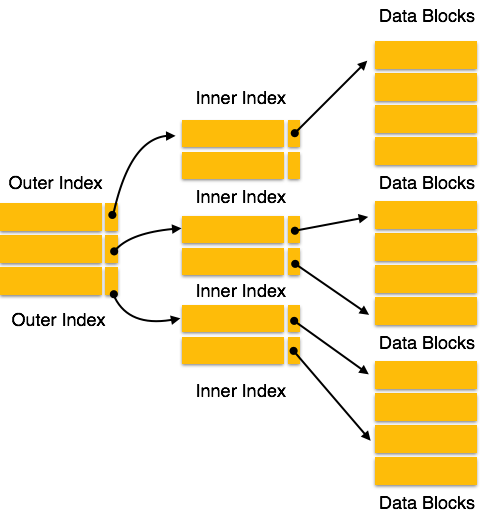


#### ****Example****

* In a bank account database, data is stored sequentially by Account\_No,  we may want to find all accounts in of a specific branch of some bank.
* In this case, we ***can have a secondary index for every search key.***
* Index record is ***a record pointing to a bucket that contains pointers to all the records*** with their specific search key value.

##### Multilevel Indexing

* Multilevel Indexing is created when a primary index does not fit in memory.
* In this type of indexing method, you ***can reduce the number of disk accesses to short any record and kept on a disk*** as a sequential file and create a sparse base on that file.



### What is the B tree?

[**B tree**](https://www.javatpoint.com/b-tree) is a self-balancing tree, and it is a m-way tree where m defines the order of the tree. **Btree** is a generalization of the [Binary Search tree](https://www.javatpoint.com/binary-search-tree) in which a node can have more than one key and more than two children depending upon the value of **m**. In the B tree, the data is specified in a sorted order having lower values on the left subtree and higher values in the right subtree.

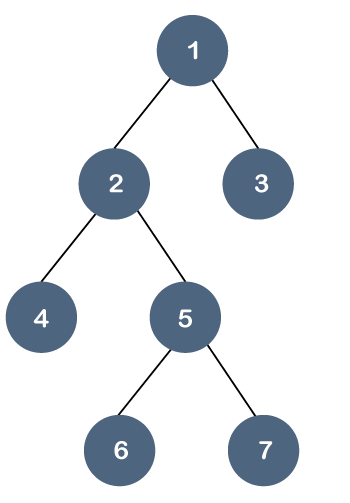
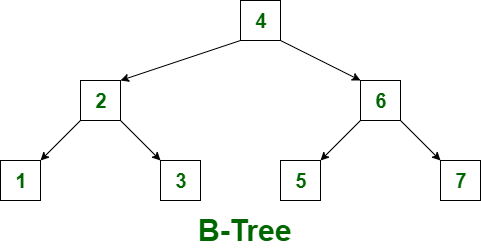
**Properties of B tree**

**The following are the properties of the B tree:**90

Hello Java Program for Beginners

* In the B tree, all the leaf nodes must be at the same level, whereas, in the case of a binary tree, the leaf nodes can be at different levels.

**Let's understand this property through an example.**

In the above tree, all the leaf nodes are not at the same level, but they have the utmost two children. Therefore, we can say that the above tree is a [binary tree](https://www.javatpoint.com/binary-tree) but not a B tree.

* If the Btree has an order of m, then each node can have a maximum of **m** In the case of minimum children, the leaf nodes have zero children, the root node has two children, and the internal nodes have a ceiling of m/2.
* Each node can have maximum (m-1) keys. For example, if the value of m is 5 then the maximum value of keys is 4.
* The root node has minimum one key, whereas all the other nodes except the root node have (ceiling of m/2 minus - 1) minimum keys.
* If we perform insertion in the B tree, then the node is always inserted in the leaf node.

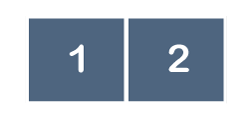
##### Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

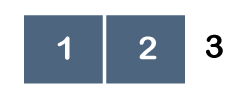
**Step 1:** First, we create a node with 1 value as shown below:



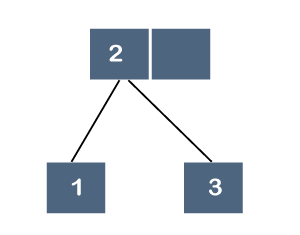
**Step 2:** The next element is 2, which comes after 1 as shown below:



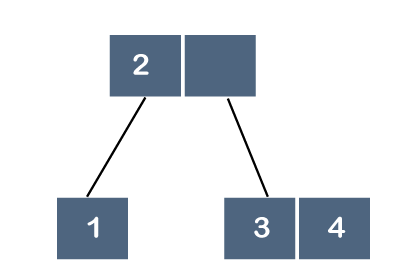
**Step 3:** The next element is 3, and it is inserted after 2 as shown below:



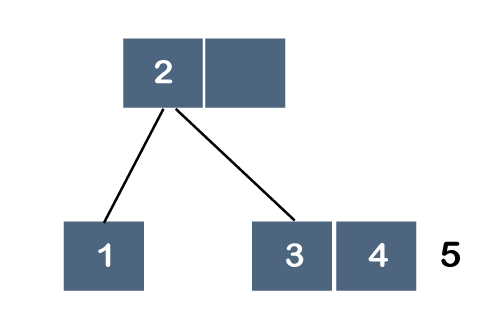
As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 2, so it moves to its parent. The node 2 does not have any parent, so it will become the root node as shown below:



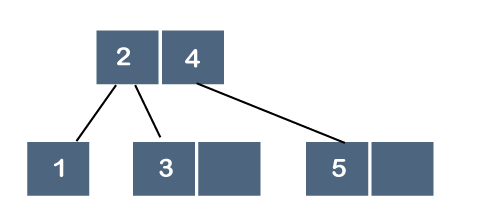
**Step 4:** The next element is 4. Since 4 is greater than 2 and 3, so it will be added after the 3 as shown below:



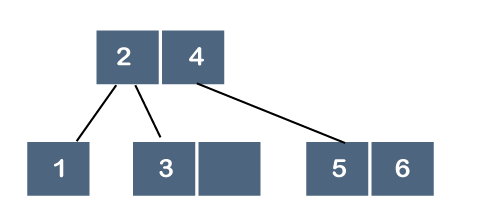
**Step 5:** The next element is 5. Since 5 is greater than 2, 3 and 4 so it will be added after 4 as shown below:



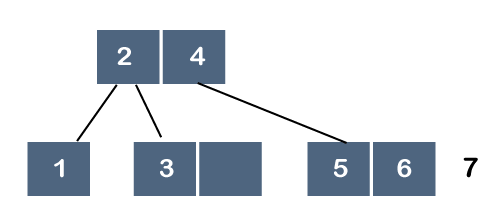
As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 4, so it moves to its parent. The parent is node 2; therefore, 4 will be added after 2 as shown below:



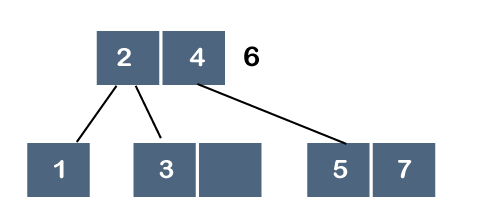
**Step 6:** The next element is 6. Since 6 is greater than 2, 4 and 5, so 6 will come after 5 as shown below:



**Step 7:** The next element is 7. Since 7 is greater than 2, 4, 5 and 6, so 7 will come after 6 as shown below:



As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 6, so it moves to its parent as shown below:



But, 6 cannot be added after 4 because the node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 4, so it moves to its parent. As node 4 does not have any parent, node 4 will become a root node as shown below:

###### B tree vs B+ tree

### What is a B+ tree?

The [B+ tree](https://www.javatpoint.com/b-plus-tree) is also known as an advanced self-balanced tree because every path from the root of the tree to the leaf of the tree has the same length. Here, the same length means that all the leaf nodes occur at the same level. It will not happen that some of the leaf nodes occur at the third level and some of them at the second level.

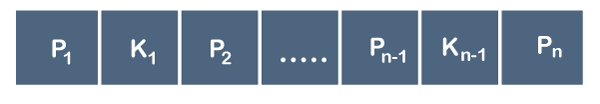
A B+ tree index is considered a multi-level index, but the B+ tree structure is not similar to the multi-level index sequential files.

**Why is the B+ tree used?**

A B+ tree is used to store the records very efficiently by storing the records in an indexed manner using the B+ tree indexed structure. Due to the multi-level indexing, the data accessing becomes faster and easier.

**B+ tree Node Structure**

The node structure of the B+ tree contains pointers and key values shown in the below figure:



As we can observe in the above B+ tree node structure that it contains n-1 key values (k1 to kn-1) and n pointers (p1 to pn).

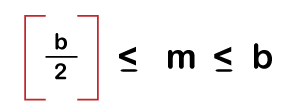
The search key values which are placed in the node are kept in sorted order. Thus, if i<j then ki<kj.

**Constraint on various types of nodes**

Let 'b' be the order of the B+ tree.

**Non-Leaf node**

Let 'm' represents the number of children of a node, then the relation between the order of the tree and the number of children can be represented as:



Let k represents the search key values. The relation between the order of the tree and search key can be represented as:

As we know that the number of pointers is equal to the search key values plus 1, so mathematically, it can be written as:

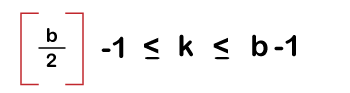
**Number of Pointers (or children) = Number of Search keys + 1**

Therefore, the maximum number of pointers would be 'b', and the minimum number of pointers would be the ceiling function of b/2.

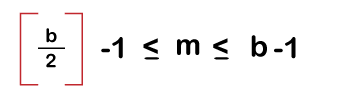
**Leaf Node**

A leaf node is a node that occurs at the last level of the B+ tree, and each leaf node uses only one pointer to connect with each other to provide the sequential access at the leaf level.

In leaf node, the maximum number of children is:



The maximum number of search keys is:



**Root Node**

The maximum number of children in the case of the root node is: b

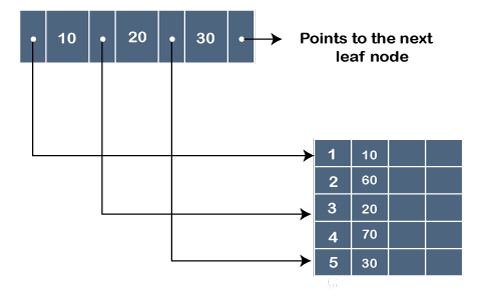
The minimum number of children is: 2

**Special cases in B+ tree**

**Case 1:** If the root node is the only node in the tree. In this case, the root node becomes the leaf node.

In this case, the maximum number of children is 1, i.e., the root node itself, whereas, the minimum number of children is b-1, which is the same as that of a leaf node.

**Representation of a leaf node in B+ tree**



In the above figure, '.' represents the pointer, whereas the 10, 20 and 30 are the key values. The pointer contains the address at which the key value is stored, as shown in the above figure.

**Example of B+ tree**



In the above figure, the node contains three key values, i.e., 9, 16, and 25. The pointer that appears before 9, contains the key values less than 9 represented by ki. The pointer that appears before 16, contains the key values greater than or equal to 9 but less than 16 represented by kj. The pointer that appears before 25, contains the key values greater than or equal to 16 but less than 25 represented by kn.

### Differences between B tree and B+ tree

|  |  |
| --- | --- |
| **B tree** | **B+ tree** |
| In the B tree, all the keys and records are stored in both internal as well as leaf nodes. | In the B+ tree, keys are the indexes stored in the internal nodes and records are stored in the leaf nodes. |
| In B tree, keys cannot be repeatedly stored, which means that there is no duplication of keys or records. | In the B+ tree, there can be redundancy in the occurrence of the keys. In this case, the records are stored in the leaf nodes, whereas the keys are stored in the internal nodes, so redundant keys can be present in the internal nodes. |
| In the Btree, leaf nodes are not linked to each other. | In B+ tree, the leaf nodes are linked to each other to provide the sequential access. |
| In Btree, searching is not very efficient because the records are either stored in leaf or internal nodes. | In B+ tree, searching is very efficient or quicker because all the records are stored in the leaf nodes. |
| Deletion of internal nodes is very slow and a time-consuming process as we need to consider the child of the deleted key also. | Deletion in B+ tree is very fast because all the records are stored in the leaf nodes so we do not have to consider the child of the node. |
| In Btree, sequential access is not possible. | In the B+ tree, all the leaf nodes are connected to each other through a pointer, so sequential access is possible. |
| In Btree, the more number of splitting operations are performed due to which height increases compared to width, | B+ tree has more width as compared to height. |
| In Btree, each node has atleast two branches and each node contains some records, so we do not need to traverse till the leaf nodes to get the data. | In B+ tree, internal nodes contain only pointers and leaf nodes contain records. All the leaf nodes are at the same level, so we need to traverse till the leaf nodes to get the data. |
| The root node contains atleast 2 to m children where m is the order of the tree. | The root node contains atleast 2 to m children where m is the order of the tree. |

### Advantages and dis-advantages of indexing:

#### **Advantages of Indexing**

* It helps you to ***reduce the total number of I/O operations*** needed to retrieve the data, so you don’t need to access a row in the database from an index structure.
* Offers ***faster search and retrieval of data*** to users.
* Indexing also helps you to***reduce table space as you don’t need to link to a row in a table,*** as there is no need to store the ROWID in the Index. Thus you will able to reduce the table space.
* You don’t need to sort data in the lead nodes as the value of the primary key classifies it.

### **Disadvantages of Indexing**

* To perform the indexing database management system, you ***need a primary key on the table with a unique value.***
* You are ***not allowed to partition an index-organized table***.
* SQL indexing ***decrease performance in INSERT, DELETE, and UPDATE query***.