Indexing in DBMS

- Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

- The index is a type of data structure. It is used to locate and access the data in a database table quickly.

Index structure:

Indexes can be created using some database columns.

Search key	Data Reference

Fig: Structure of Index

- 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.

Indexing Methods
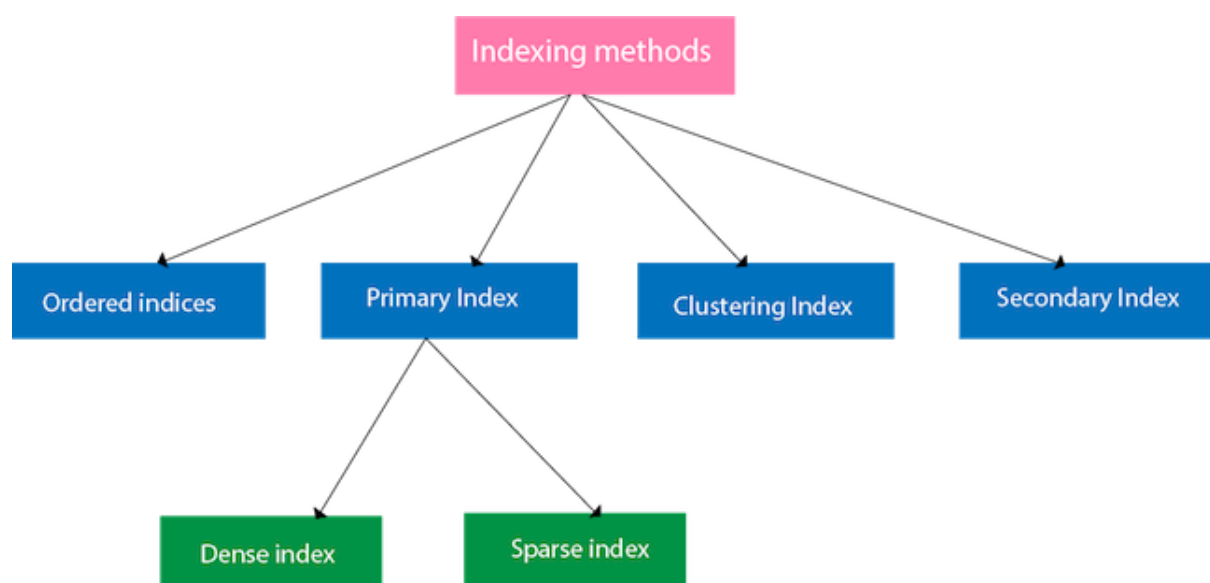
Ordered indices

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

- The dense index contains an index record for every search key value in the data file. It makes searching faster.

- In this, the number of records in the index table is same as the number of records in the main table.

- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

| UP			UP	Agra	1,604,300
USA			USA	Chicago	2,789,378
Nepal			Nepal	Kathmandu	1,456,634
UK			UK	Cambridge	1,360,364

Sparse index

- In the data file, index record appears only for a few items. Each item points to a block.

- In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.

|			UP	Agra	1,604,300
UP	•		USA	Chicago	2,789,378
Nepal	•		Nepal	Kathmandu	1,456,634
UK	•		UK	Cambridge	1,360,364

Clustering Index

- o A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.

- o In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.

- o The records which have similar characteristics are grouped, and indexes are created for these group.

Example: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept_ID are considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept_Id is a non-unique key.
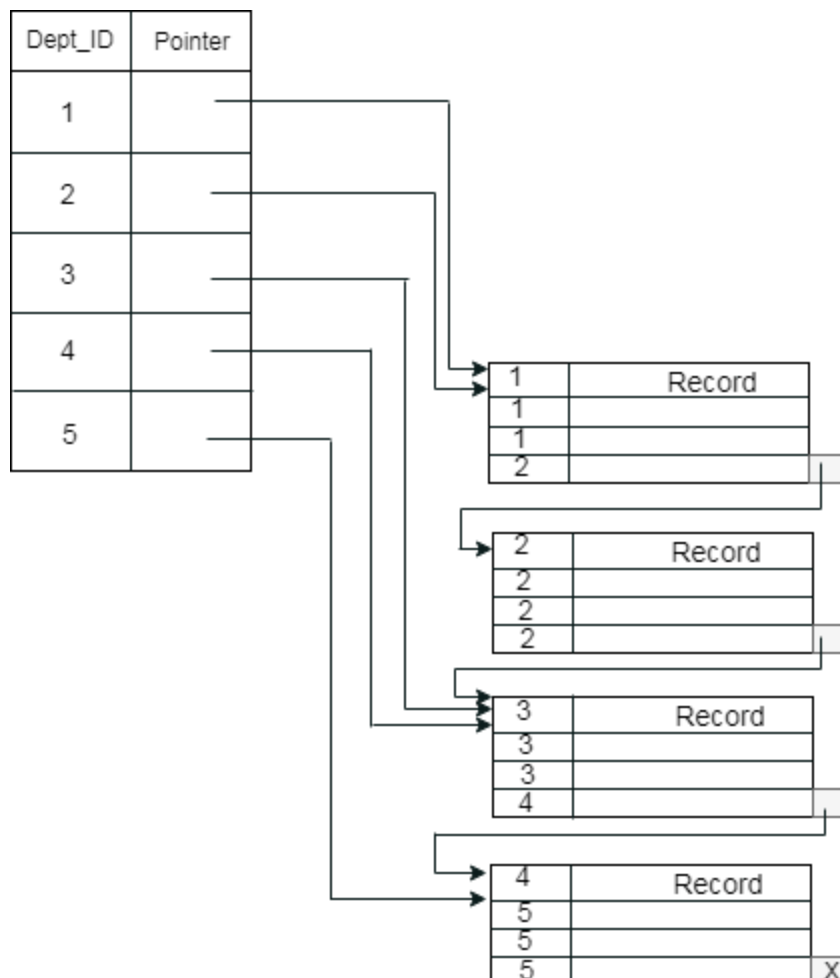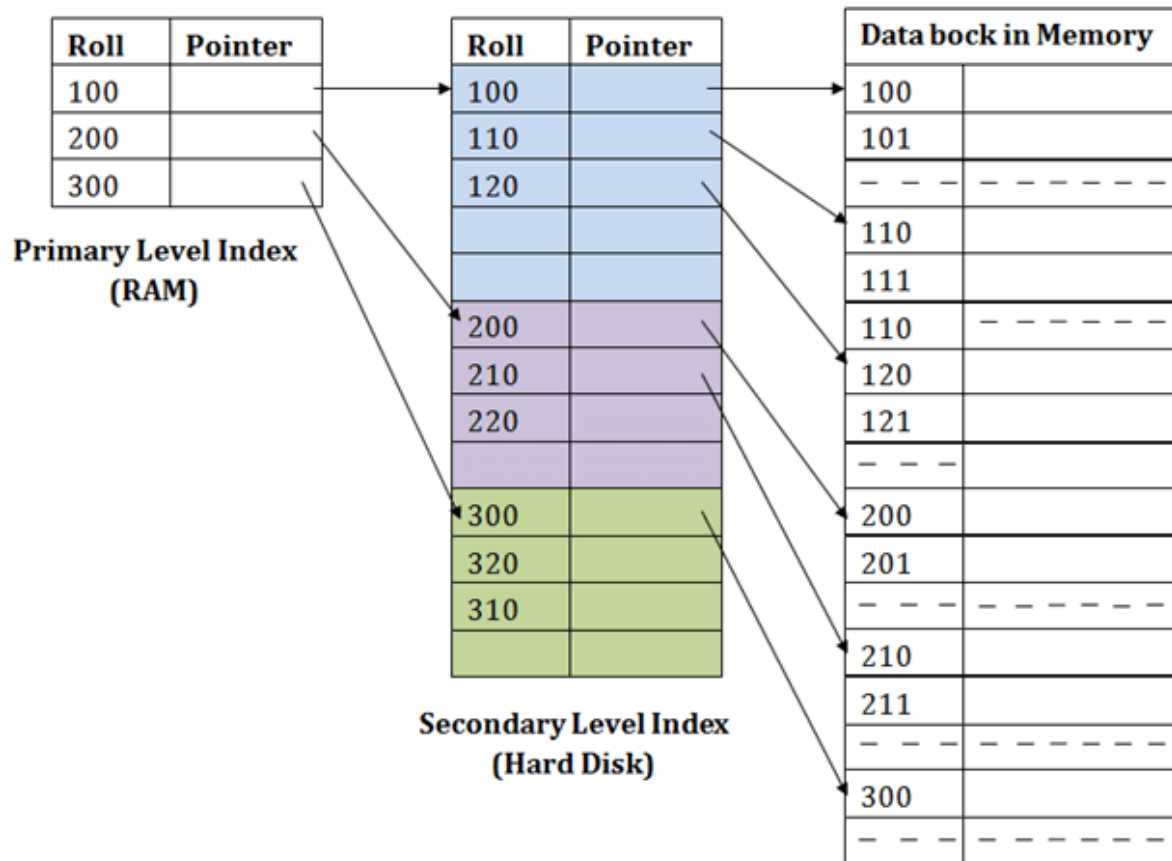
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.
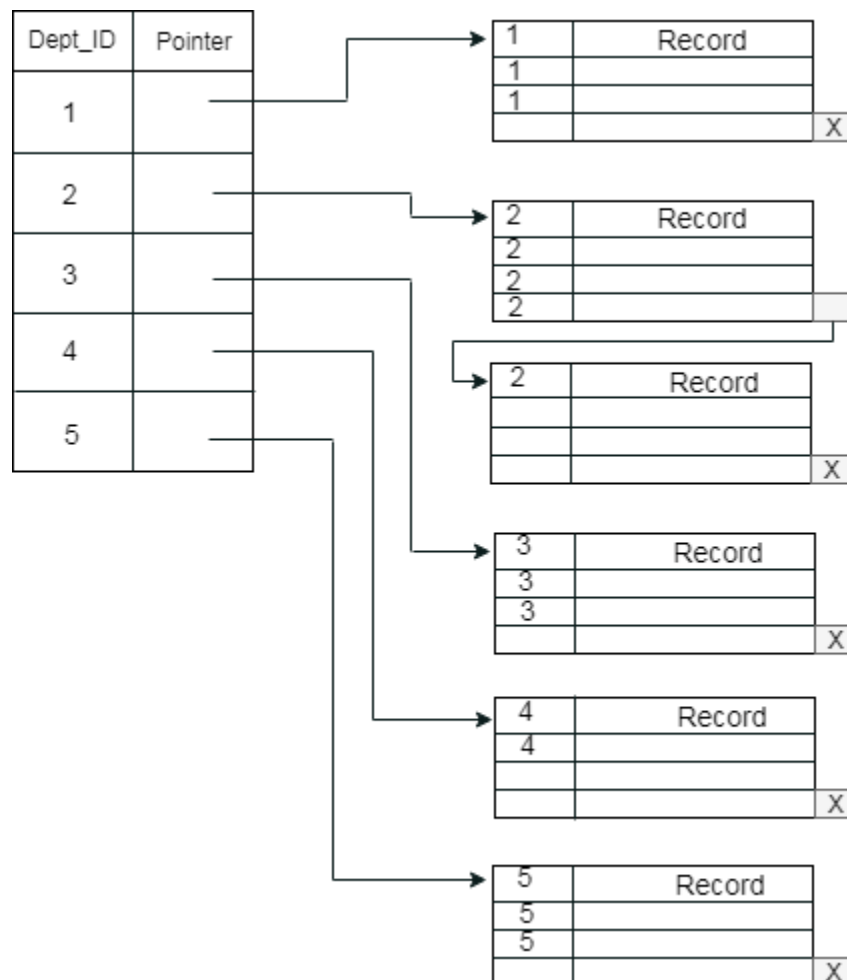
Secondary Index

In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

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).

Roll	Pointer
100	
200	
300	

Primary Level Index (RAM)

Roll	Pointer
100	
110	
120	
200	
210	
220	
300	
320	
310	

Secondary Level Index (Hard Disk)

| Data bock in Memory	
100	
101	
− − −	− − − − − −
110	
111	
110	− − − − − −
120	
121	
− − −	
200	
201	
− − −	− − − − − −
210	
211	
− − −	− − − − − −
300	
− − −	− − − − − −

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.

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.
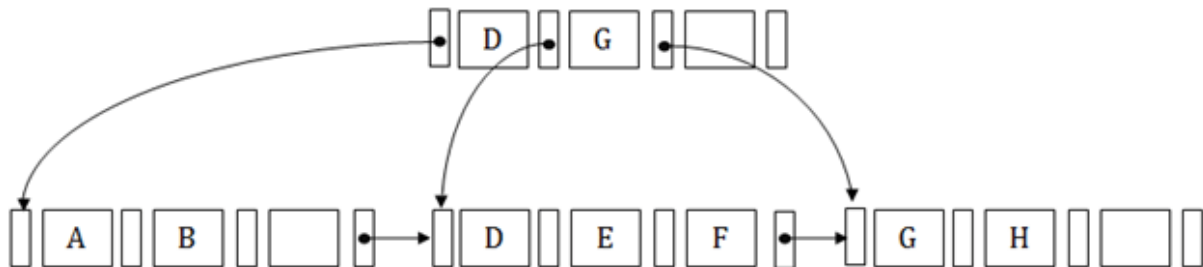
Internal node

- 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.
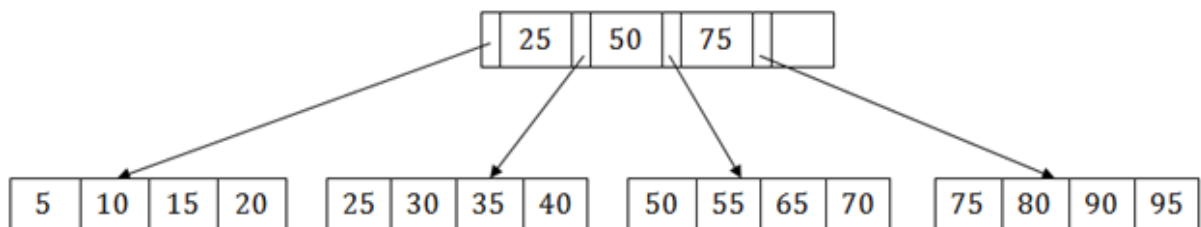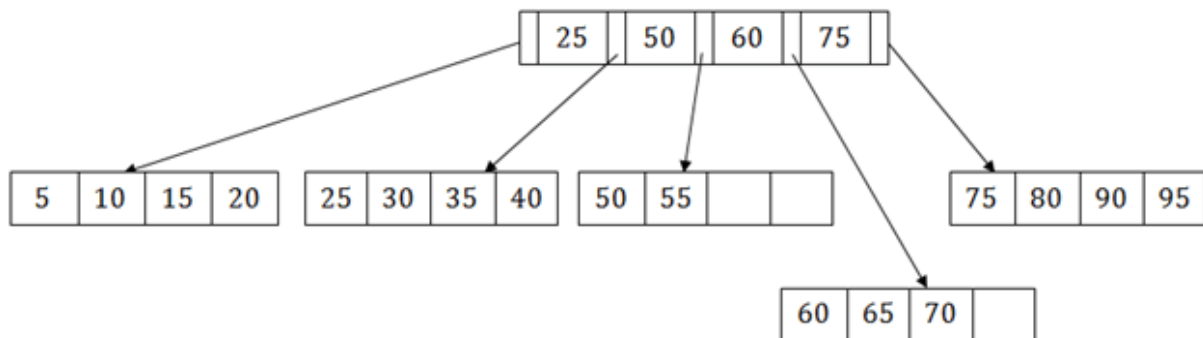

B+ Tree Insertion

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.

| 25 | | 50 | | 75 | | |

| 5 | 10 | 15 | 20 |

| 25 | 30 | 35 | 40 |

| 50 | 55 | 65 | 70 |

| 75 | 80 | 90 | 95 |

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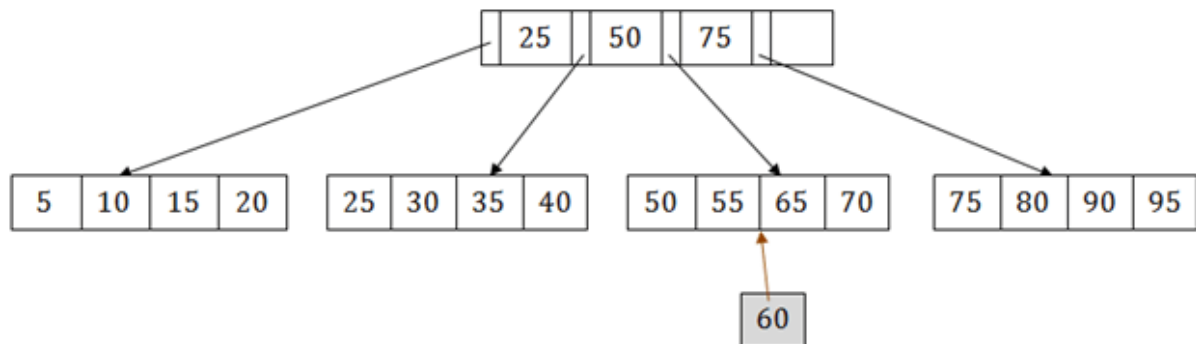
The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

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.

| 25 | | 50 | | 60 | | 75 | |

| 5 | 10 | 15 | 20 |

| 25 | 30 | 35 | 40 |

| 50 | 55 | | |

| 75 | 80 | 90 | 95 |

| 60 | 65 | 70 | |

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: