Module - 3

Storage Strategies in DBMS

Storage System Hierarchy in DBMS

The storage hierarchy typically has multiple levels, each with its specific characteristics. Here's a typical hierarchy from fastest (and usually most expensive per byte) to slowest (and usually least expensive per byte):

1. Registers

- Located within the CPU.
- Smallest and fastest type of storage.
- Used to hold data currently being processed.

2. Cache Memory (L1, L2, L3 caches)

- On or very close to the CPU.
- Extremely fast but small in size.
- Acts as a buffer for frequently used data.

3. Main Memory (RAM)

- Data that's actively being used or processed is loaded here.
- Faster than secondary storage.
- Volatile in nature (i.e., data is lost when power is turned off).

4. Flash Storage (Solid State Drives - SSD)

- No moving parts.
- Faster than traditional hard drives.
- More durable and reliable.

5. Magnetic Disks (Hard Disk Drives - HDD)

- Primary secondary storage medium.
- Non-volatile, persistent storage.
- Data is stored in tracks, sectors, and cylinders.
- Slower than main memory but offers a large storage capacity at a lower cost.

6. Optical Disks (CD, DVD, Blu-Ray)

- Data is read using lasers.
- Slower than magnetic disks and usually have less storage capacity.
- Portable and commonly used for media and software distribution.

7. Magnetic Tapes

- Sequential access storage, unlike disks which are random access.
- Often used for backups and archiving due to their high capacity and low cost.
- Much slower access times compared to magnetic disks.

8. Remote Storage/Cloud Storage

- Data stored in remote servers and accessed over the internet.
- Provides scalability, availability, and fault-tolerance.
- Latency depends on network speed and distance to servers.

Types of Storage:

1. **Primary Storage:** Includes registers, cache memory, and main memory (RAM). It's the main memory where the operating system, application programs, and data in current use are kept for quick access by the computer's processor.
2. **Secondary Storage:** Encompasses data storage devices like HDDs, SSDs, CDs, and USB drives. It is non-volatile and retains data even when the computer is turned off.
3. **Tertiary Storage or Off-line Storage:** Often involves magnetic tape systems or optical disk archives. This is slower than secondary storage and is used for data archiving and backup.

4. **Quaternary Storage:** Refers to methods like cloud storage or other remote storage techniques where data is stored in remote servers and is fetched over the internet or other networks.

Types of Memory

1. Primary Memory

The primary memory of a server is the type of data storage that is directly accessible by the central processing unit, meaning that it doesn't require any other devices to read from it. The primary memory must, in general, function flawlessly with equal contributions from the electric power supply, the hardware backup system, the supporting devices, the coolant that moderates the system temperature, etc.

- The size of these devices is considerably smaller and they are volatile.
- According to performance and speed, the primary memory devices are the fastest devices, and this feature is in direct correlation with their capacity.
- These primary memory devices are usually more expensive due to their increased speed and performance.

The cache is one of the types of Primary Memory.

- **Cache Memory:** Cache Memory is a special very high-speed memory. It is used to speed up and synchronize with a high-speed CPU. Cache memory is costlier than main memory or disk memory but more economical than CPU registers. Cache memory is an

extremely fast memory type that acts as a buffer between RAM and the CPU.

2. Secondary Memory

Data storage devices known as secondary storage, as the name suggests, are devices that can be accessed for storing data that will be needed at a later point in time for various purposes or database actions. Therefore, these types of storage systems are sometimes called backup units as well. Devices that are plugged or connected externally fall under this memory category, unlike primary memory, which is part of the CPU. The size of this group of devices is noticeably larger than the primary devices and smaller than the tertiary devices.

- It is also regarded as a temporary storage system since it can hold data when needed and delete it when the user is done with it. Compared to primary storage devices as well as tertiary devices, these secondary storage devices are slower with respect to actions and pace.
- It usually has a higher capacity than primary storage systems, but it changes with the technological world, which is expanding every day.

Some commonly used Secondary Memory types that are present in almost every system are:

- **Flash Memory:** Flash memory, also known as flash storage, is a type of nonvolatile memory that erases data in units called blocks and rewrites data at the byte level. Flash memory is widely used for storage and data transfer in consumer devices, enterprise systems, and industrial applications. Unlike traditional hard drives, flash memories are able to retain data even after the power has been turned off
- **Magnetic Disk Storage:** A Magnetic Disk is a type of secondary memory that is a flat disc covered with a magnetic coating to hold information. It is used to store various programs and files. The polarized information in one direction is represented by 1, and vice versa. The direction is indicated by 0.

3. Tertiary Memory

For data storage, Tertiary Memory refers to devices that can hold a large amount of data without being constantly connected to the server or the peripherals. A device of this type is connected either to a server or to a device where the database is stored from the outside.

- Due to the fact that tertiary storage provides more space than other types of device memory but is most slowly performing, the cost of tertiary storage is lower than primary and secondary. As a means to make a backup of data, this type of storage is commonly used for making copies from servers and databases.
- The ability to use secondary devices and to delete the contents of the tertiary devices is similar.

Some commonly used Tertiary Memory types that are almost present in every system are:

- **Optical Storage:** It is a type of storage where reading and writing are to be performed with the help of a laser. Typically data written on CDs and DVDs are examples of Optical Storage.
- **Tape Storage:** Tape Storage is a type of storage data where we use magnetic tape to store data. It is used to store data for a long time and also helps in the backup of data in case of data loss.

Memory Hierarchy

A computer system has a hierarchy of memory. Direct access to a CPU's main memory and inbuilt registers is available. Accessing the main memory takes less time than running a CPU. Cache memory is introduced to minimize this difference in speed. Data that is most frequently accessed by the CPU resides in cache memory, which provides the fastest access time to data. Fastest-accessing memory is the most expensive. Although large storage devices are slower and less expensive than CPU registers and cache memory, they can store a greater amount of data.

1. Magnetic Disks

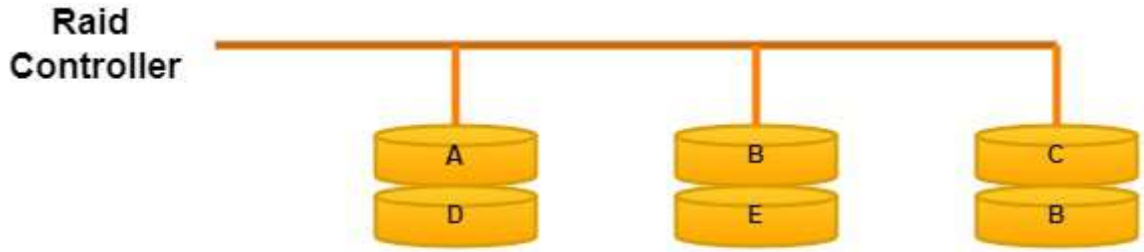
Present-day computer systems use hard disk drives as secondary storage devices. Magnetic disks store information using the concept of magnetism. Metal disks are coated with magnetizable material to create hard disks. Spindles hold these disks vertically. As the read/write head moves between the disks, it de-magnetizes or magnetizes the spots under it. There are two magnetized spots: 0 (zero) and 1 (one). Formatted hard disks store data efficiently by storing them in a defined order. The hard disk plate is divided into many concentric circles, called tracks. Each track contains a number of sectors. Data on a hard disk is typically stored in sectors of 512 bytes.

2. Redundant Array of Independent Disks(RAID)

In the Redundant Array of Independent Disks technology, two or more secondary storage devices are connected so that the devices operate as one

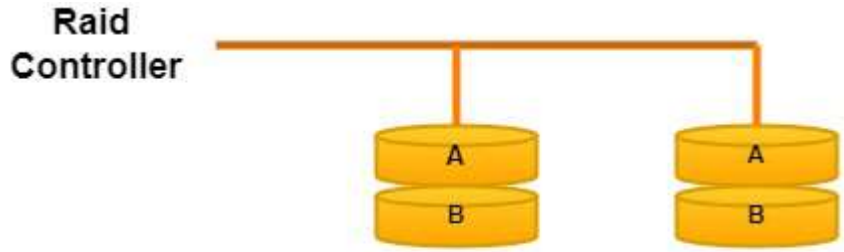
storage medium. A RAID array consists of several disks linked together for a variety of purposes. Disk arrays are categorized by their RAID levels.

- **RAID 0:** At this level, disks are organized in a striped array. Blocks of data are divided into disks and distributed over disks. Parallel writing and reading of data occur on each disk. This improves performance and speed. Level 0 does not support parity and backup.



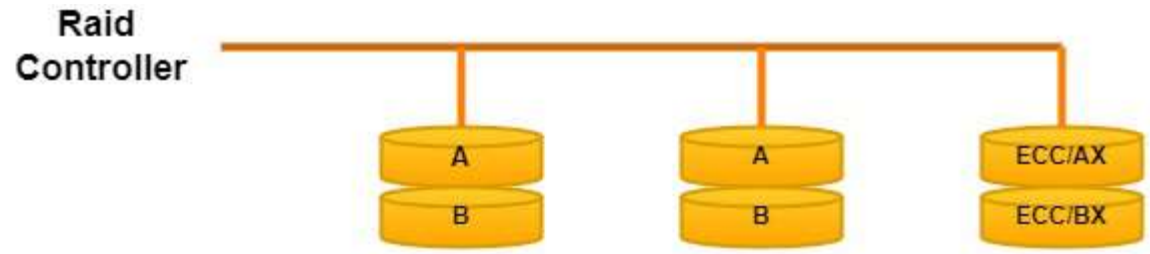
Raid-0

- **RAID 1:** Mirroring is used in RAID 1. A RAID controller copies data across all disks in an array when data is sent to it. In case of failure, RAID level 1 provides 100% redundancy.



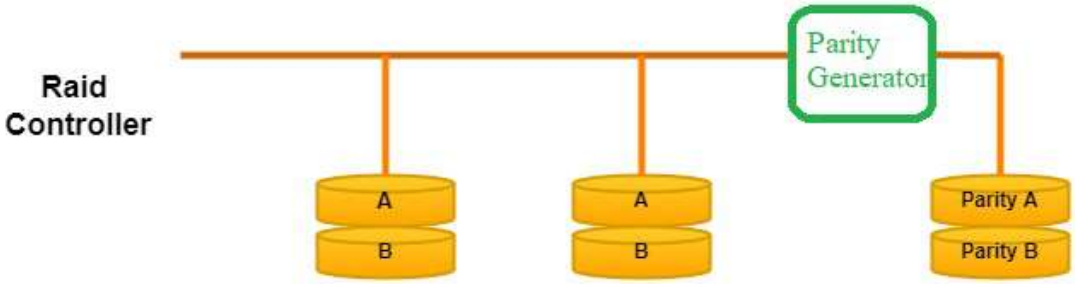
Raid-1

- **RAID 2:** The data in RAID 2 is striped on different disks, and the Error Correction Code is recorded using Hamming distance. Similarly to level 0, each bit within a word is stored on a separate disk, and ECC codes for the data words are saved on a separate set of disks. As a result of its complex structure and high cost, RAID 2 cannot be commercially deployed.



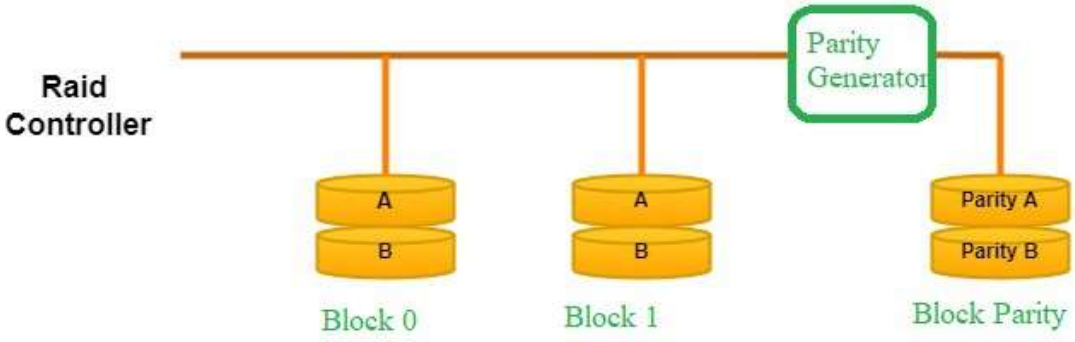
Raid-2

- **RAID 3:** Data is striped across multiple disks in RAID 3. Data words are parsed to generate a parity bit. It is stored on a different disk. Thus, single-disk failures can be avoided.



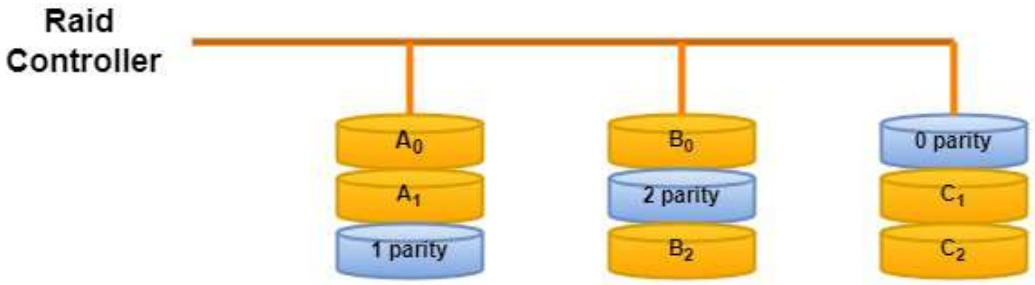
Raid-3

- **RAID 4:** This level involves writing an entire block of data onto data disks, and then generating the parity and storing it somewhere else. At level 3, bytes are striped, while at level 4, blocks are striped. Both levels 3 and 4 require a minimum of three disks.



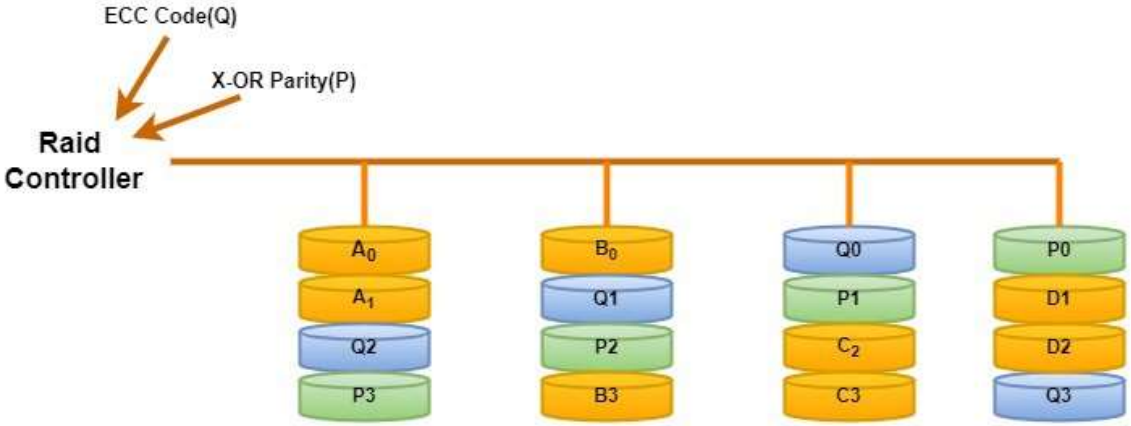
Raid-4

- **RAID 5:** The data blocks in RAID 5 are written to different disks, but the parity bits are spread out across all the data disks rather than being stored on a separate disk.



Raid-5

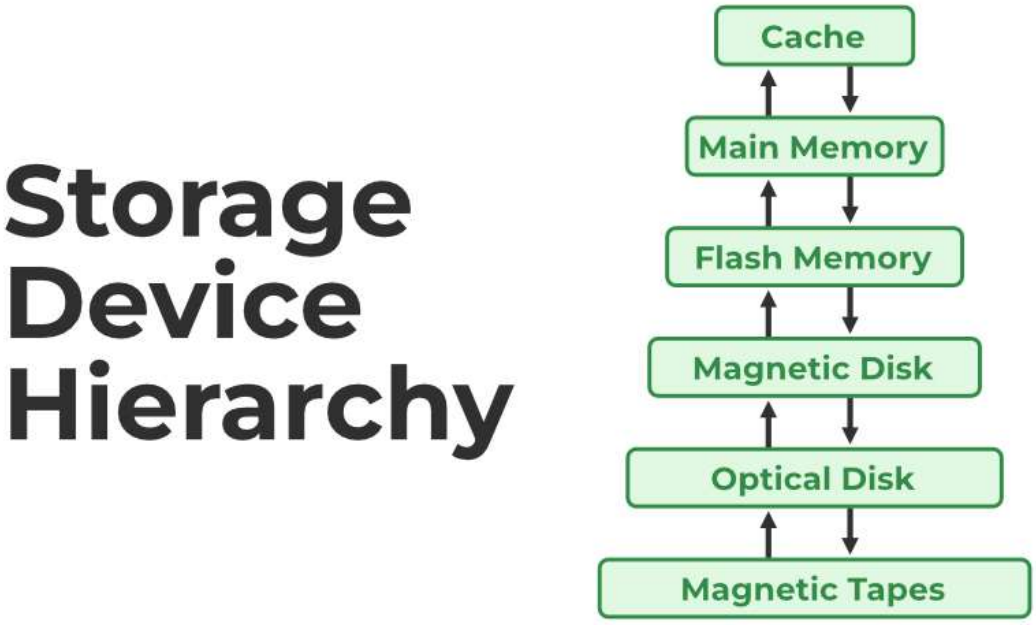
- **RAID 6:** The RAID 6 level extends the level 5 concept. A pair of independent parities are generated and stored on multiple disks at this level. A pair of independent parities are generated and stored on multiple disks at this level. Ideally, you need four disk drives for this level.



Raid-6

Storage Hierarchy

Rather than the storage devices mentioned above, there are also other devices that are also used in day-to-day life. These are mentioned below in the form of faster speed to lower speed from top to down.



Storage Hierarchy

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.

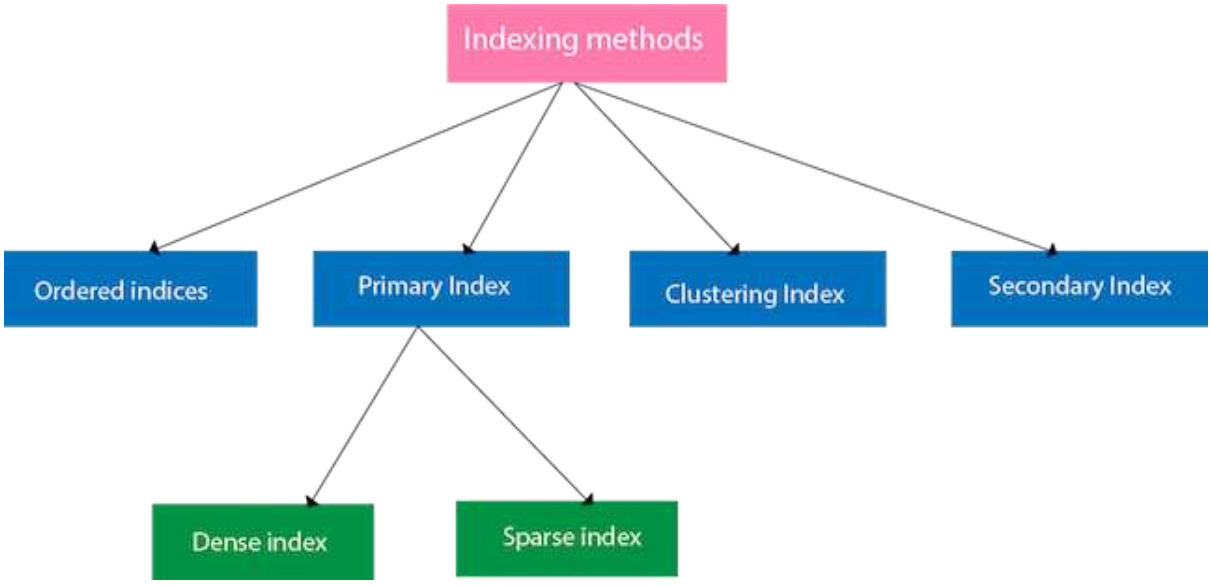
Search key	Data Reference

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.

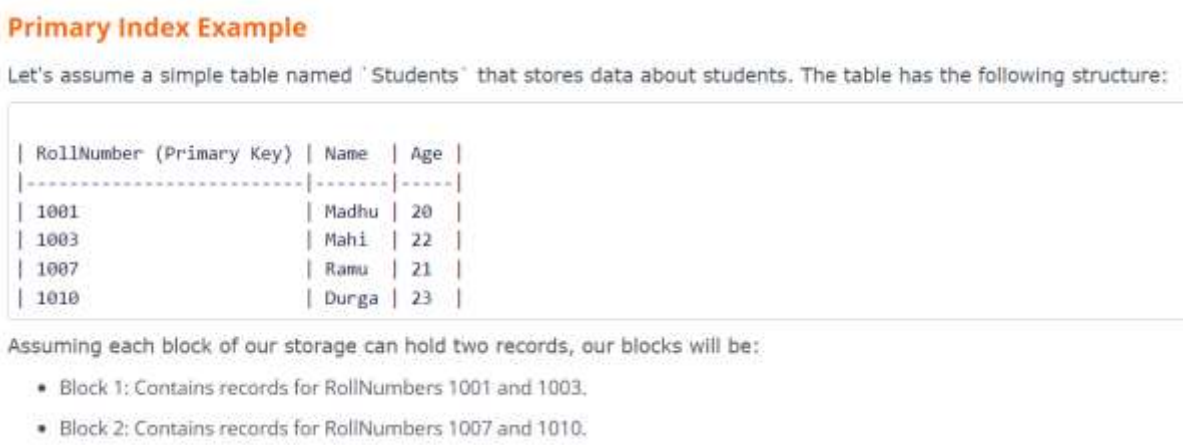
Example: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search student with ID-543.

- 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.



Primary Index Example

Let's assume a simple table named `Students` that stores data about students. The table has the following structure:

RollNumber (Primary Key)	Name	Age
1001	Madhu	20
1003	Mahi	22
1007	Ramu	21
1010	Durga	23

Assuming each block of our storage can hold two records, our blocks will be:

- Block 1: Contains records for RollNumbers 1001 and 1003.
- Block 2: Contains records for RollNumbers 1007 and 1010.

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

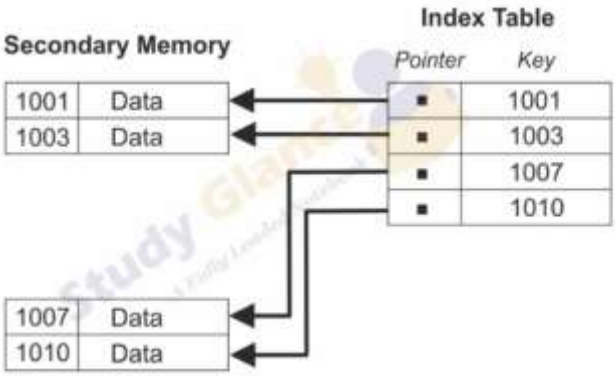
The primary index can be classified into two types

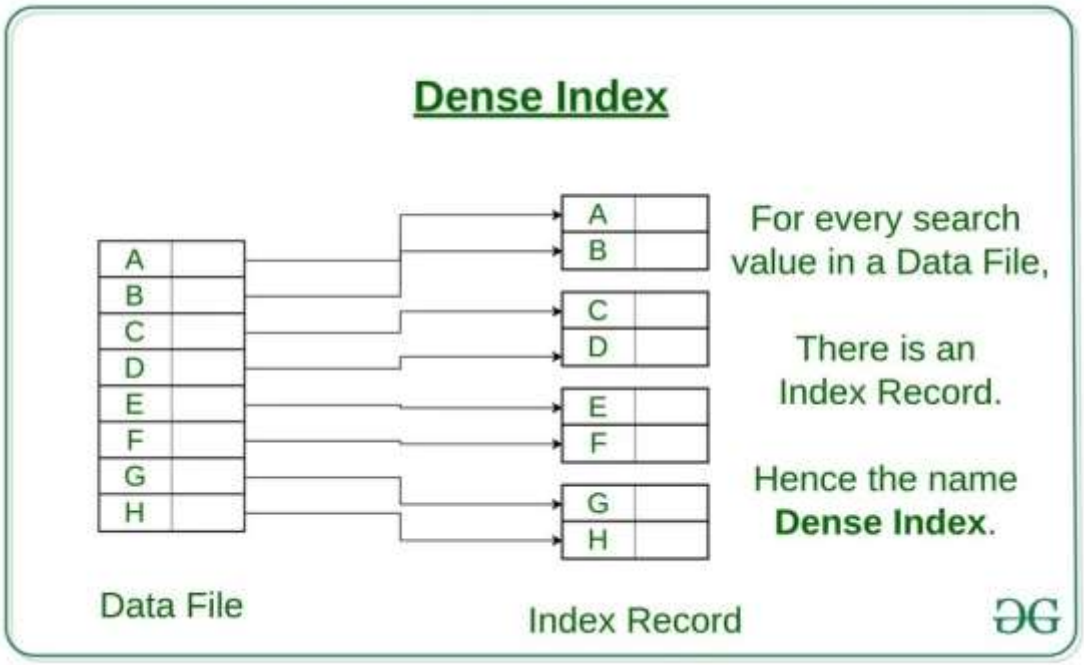
1. Dense Index: In this, an index entry appears for every search key value in the data file.
2. Sparse (or Non-Dense) Index: Here, index records are created only for some of the search key values. A sparse index reduces the size of the index file.

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.

Key (RollNumber)	Pointer to Block
1001	Block 1
1003	Block 1
1007	Block 2
1010	Block 2



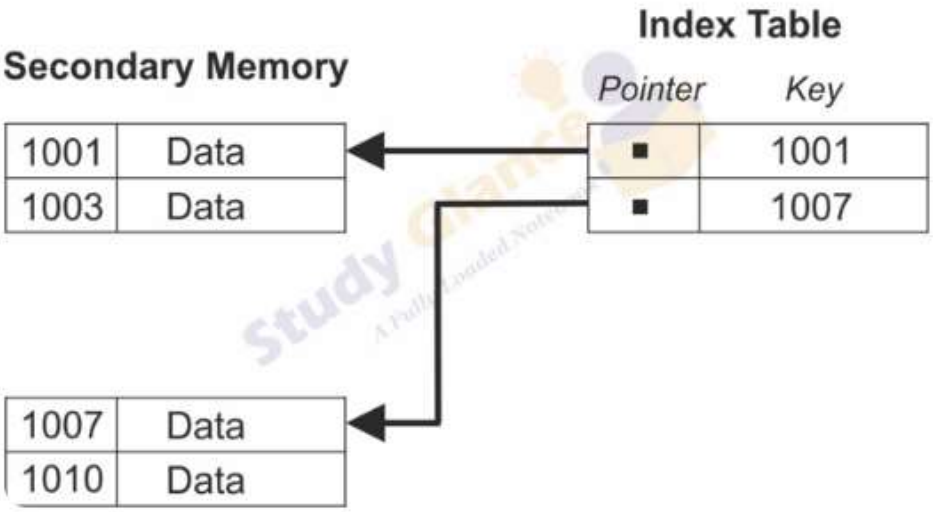


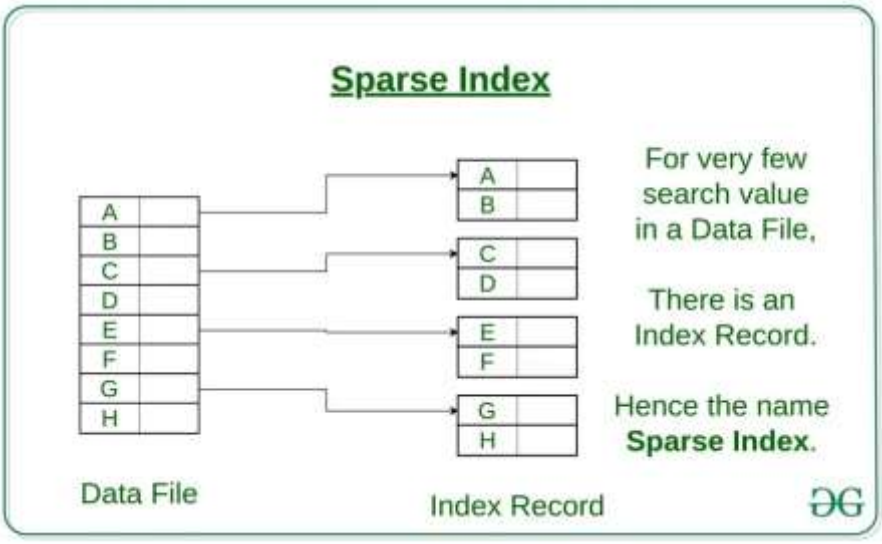
Sparse index

- 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.
- 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=$\log_2(n)+1$, (here n=number of blocks acquired by index file)
-

Key (RollNumber)	Pointer to Block
1001	Block 1
1007	Block 2





In the dense primary index, there's an entry for every key value, whereas in the sparse primary index, only the first key value of each block (the block's anchor record) gets an entry in the index. In this way, sparse indexes are more compact but might require more sequential block reads if a search key isn't an anchor for a block.

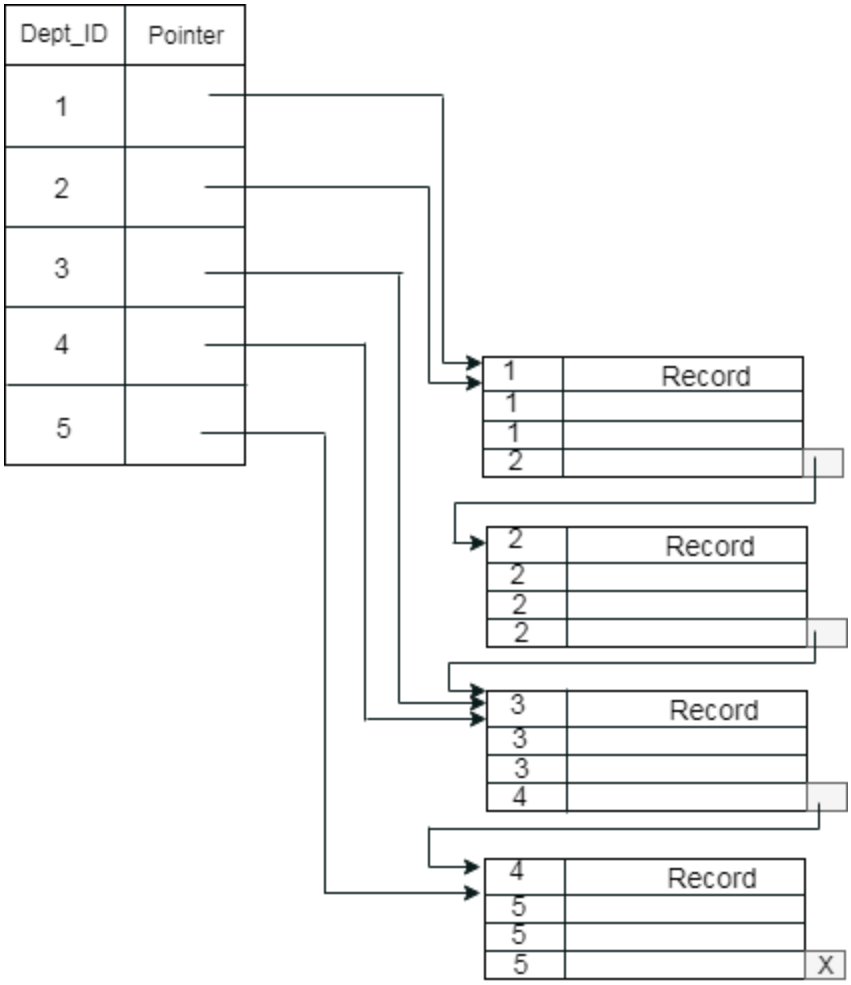
Hash File Organization

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:

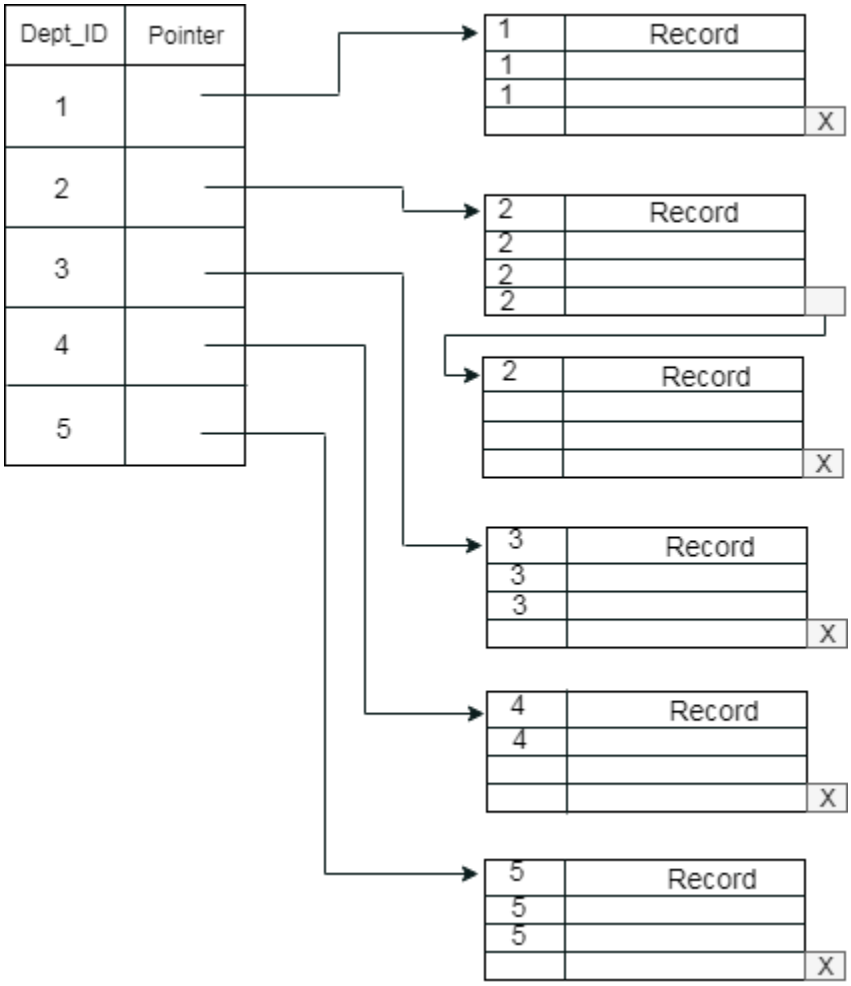
Clustering Index

- 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.
- 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.
- 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.



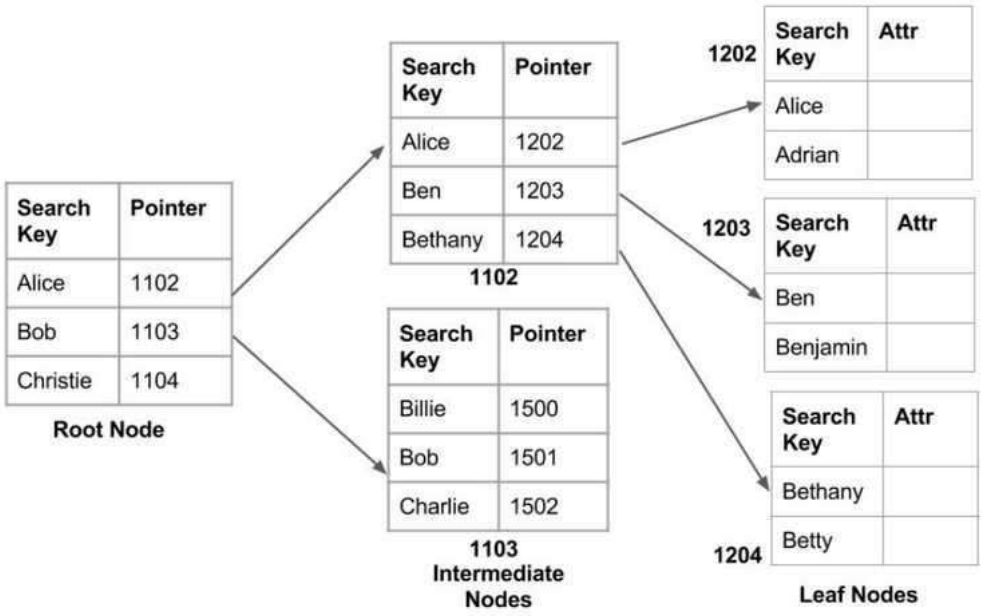
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



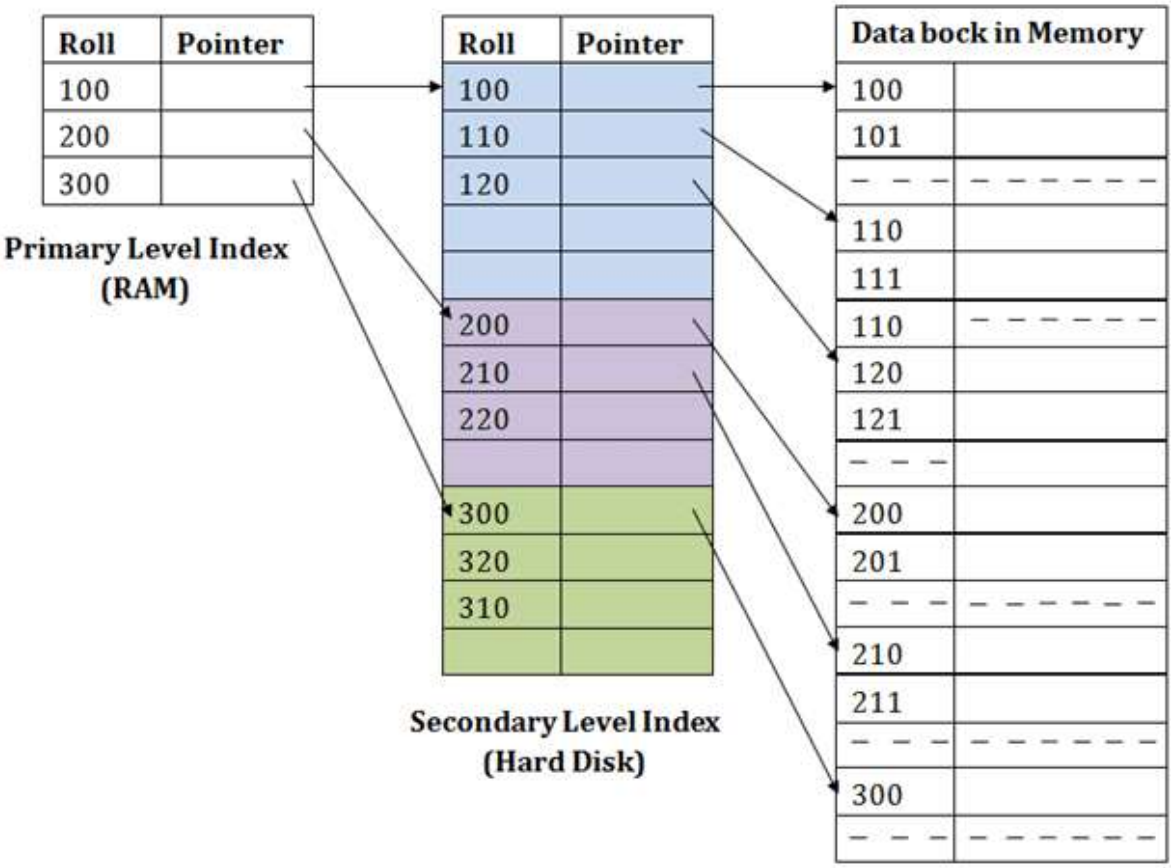
Secondary Index

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).



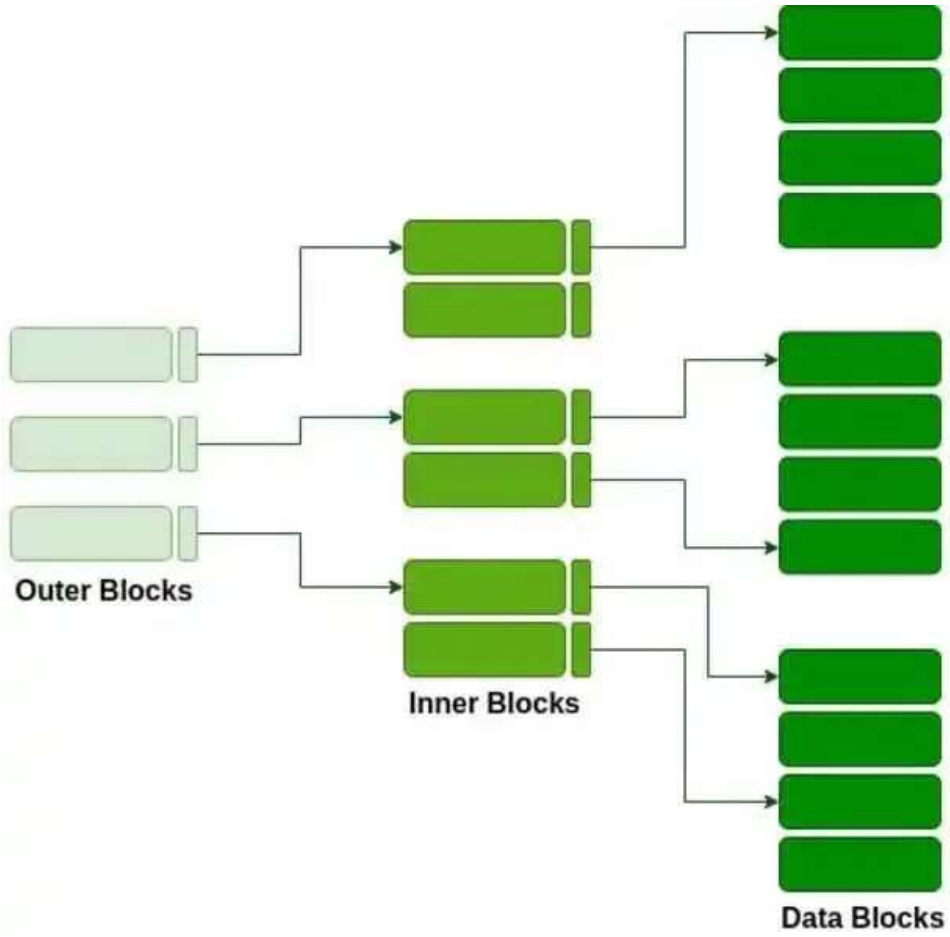
Non clustered index



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 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.



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 assist 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.
- 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.

Overall, indexing in databases provides significant benefits for improving query performance, efficient data access, optimized data sorting, consistent data performance, and enforced data integrity

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 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.

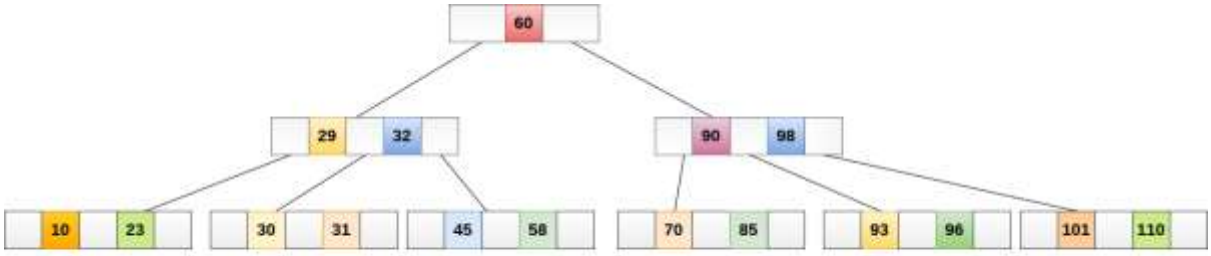
B Tree

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.

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.

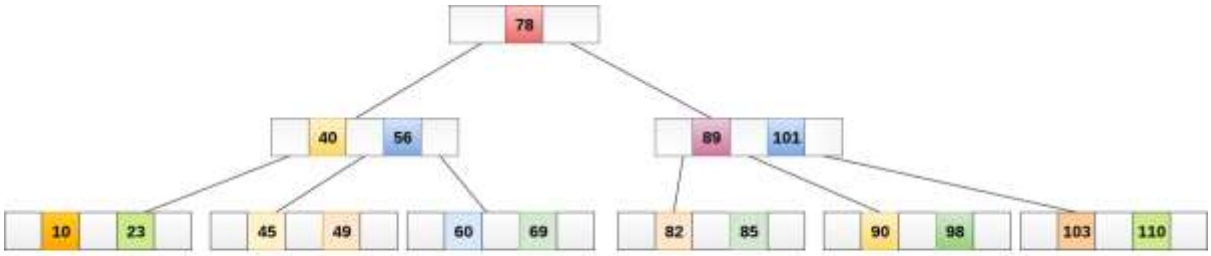
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree



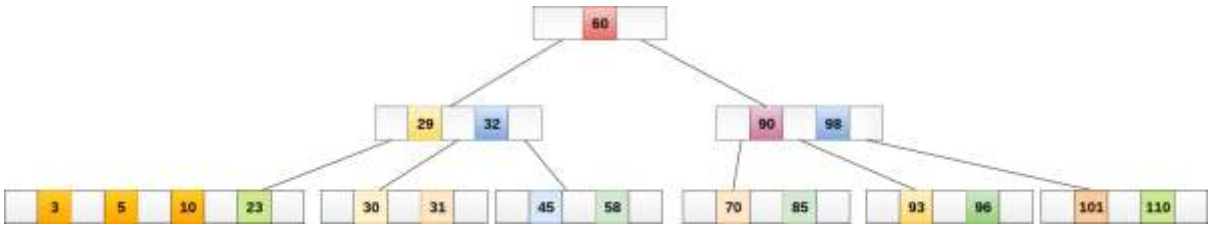
Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

Insert the node 8 into the B Tree of order 5 shown in the following image.



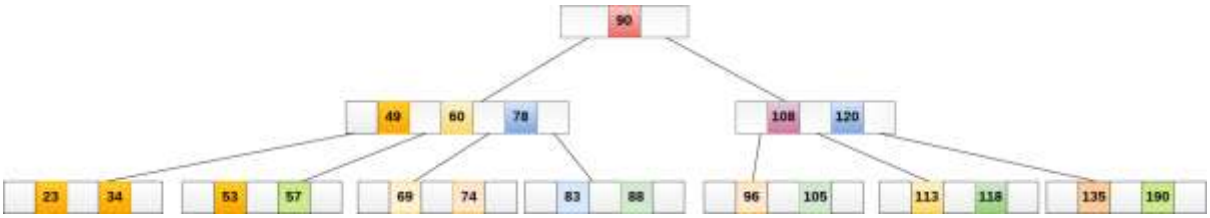
8 will be inserted to the right of 5, therefore insert 8.

Deletion

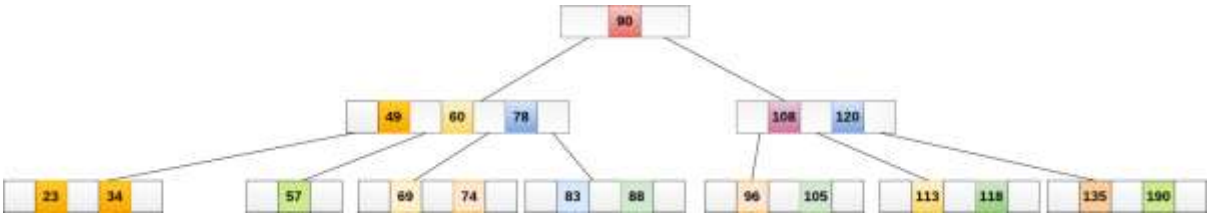
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

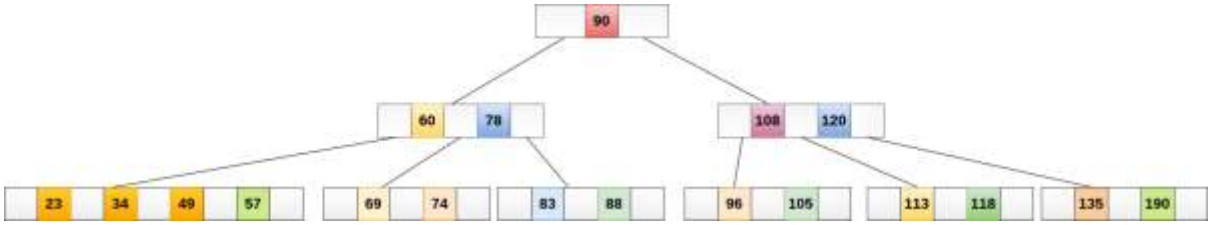
Example 1
Delete the node 53 from the B Tree of order 5 shown in the following figure.



53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

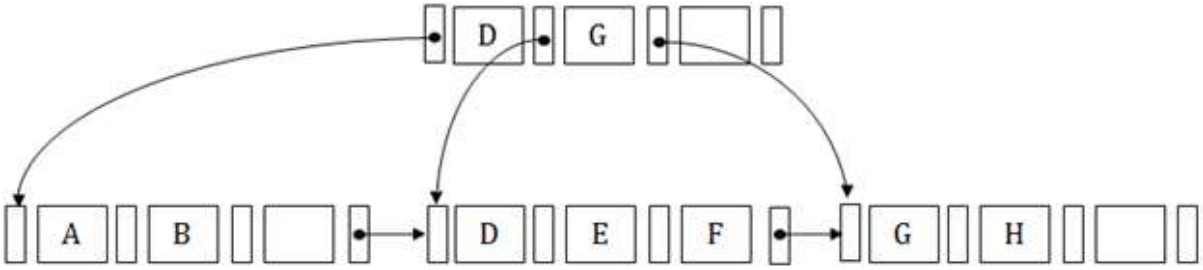
B+ Tree

- 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.

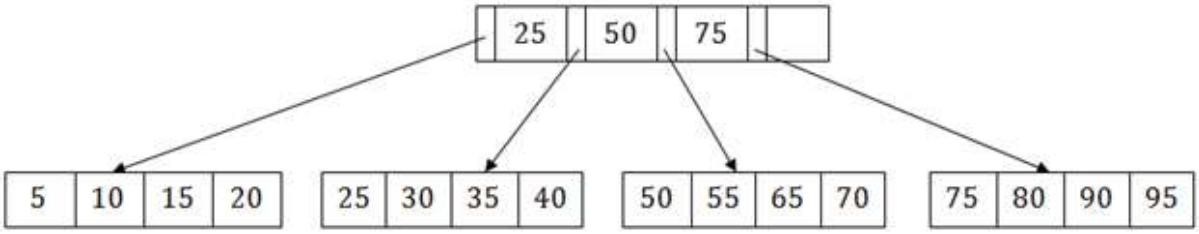
Leaf node

- 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.

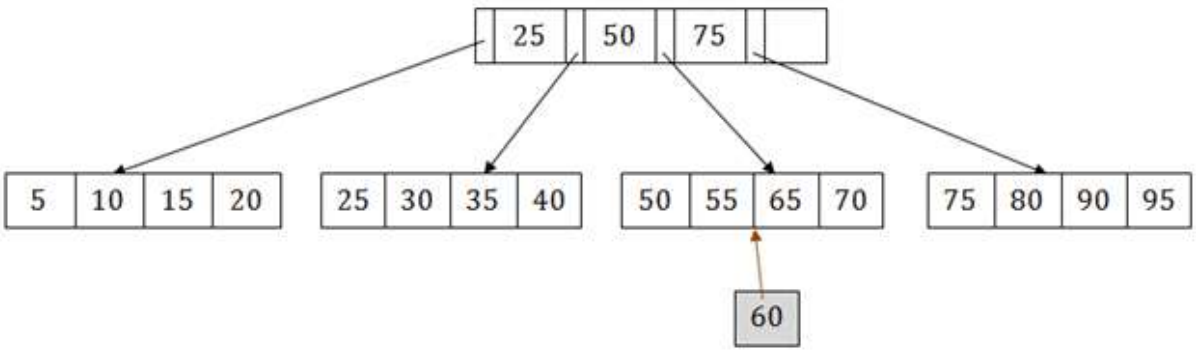
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

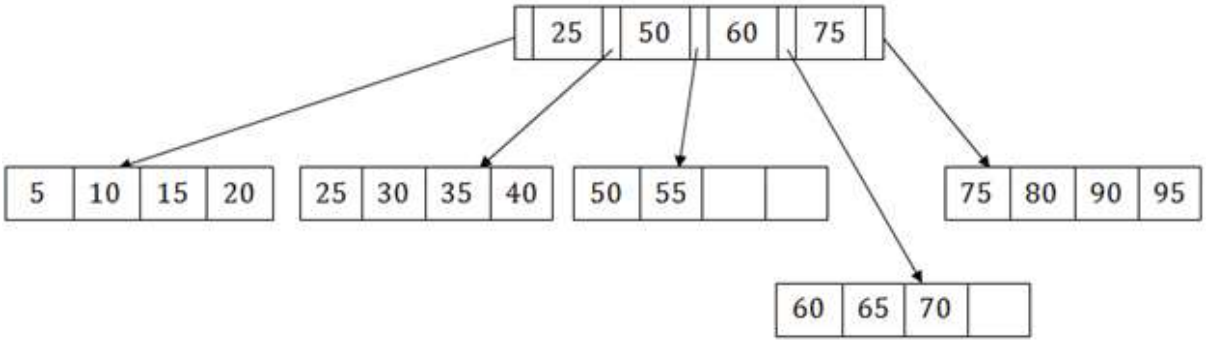
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

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.

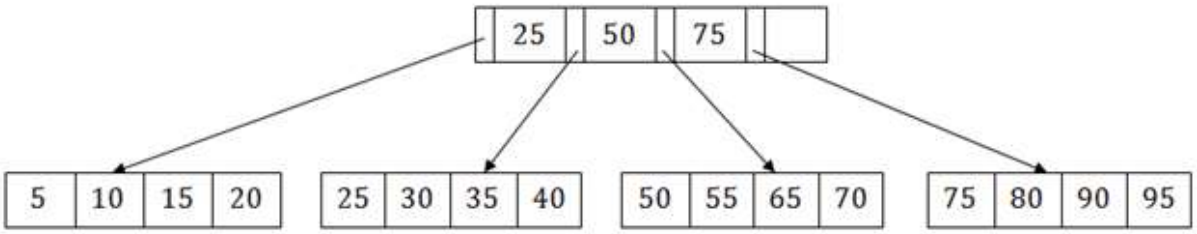


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

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:

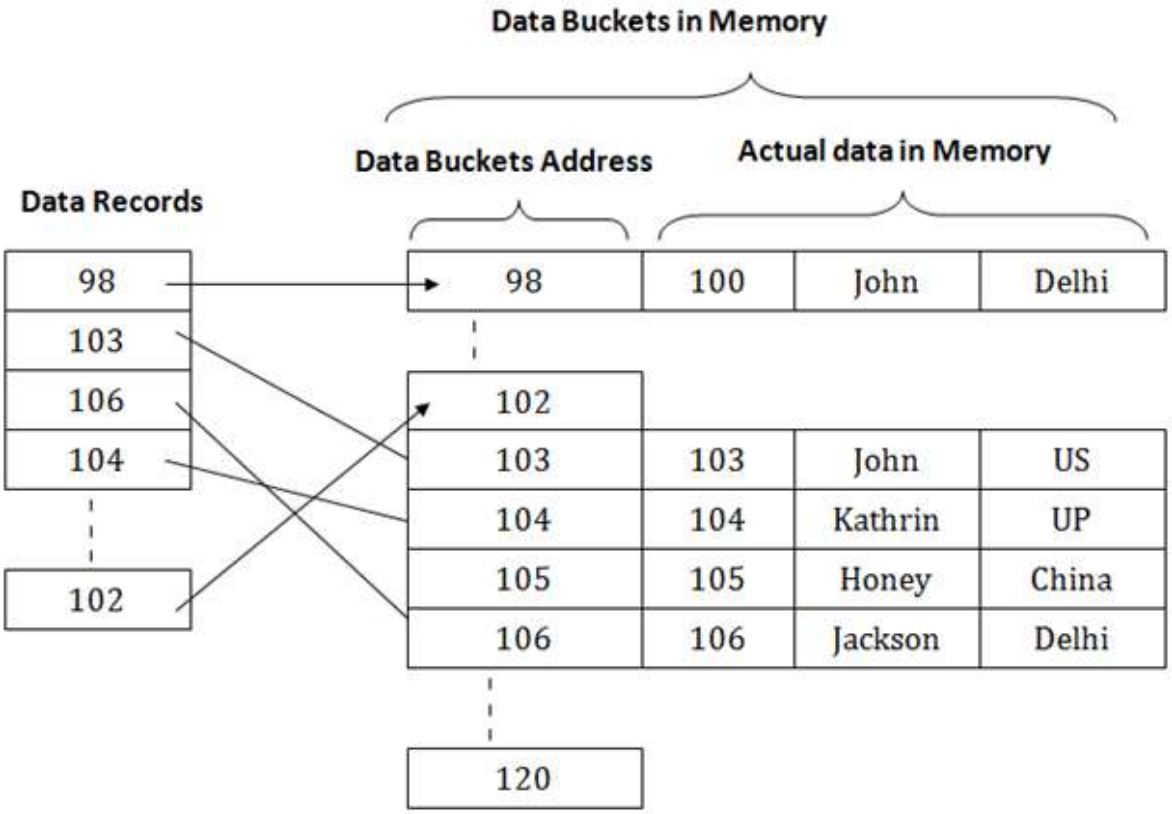


Hashing in DBMS

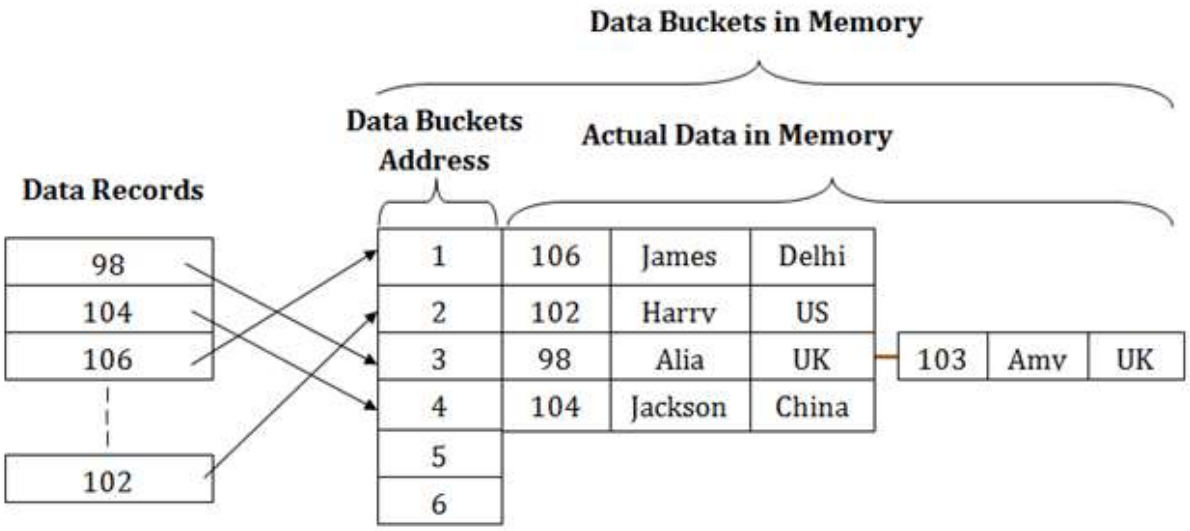
In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

In this technique, data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data bucket or data blocks.

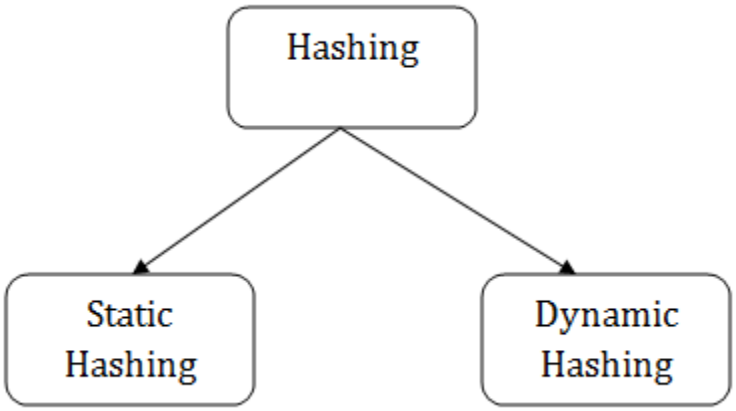
In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary key itself as the address of the data block. That means each row whose address will be the same as a primary key stored in the data block.



The above diagram shows data block addresses same as primary key value. This hash function can also be a simple mathematical function like exponential, mod, cos, sin, etc. Suppose we have mod (5) hash function to determine the address of the data block. In this case, it applies mod (5) hash function on the primary keys and generates 3, 3, 1, 4 and 2 respectively, and records are stored in those data block addresses.



Types of Hashing:

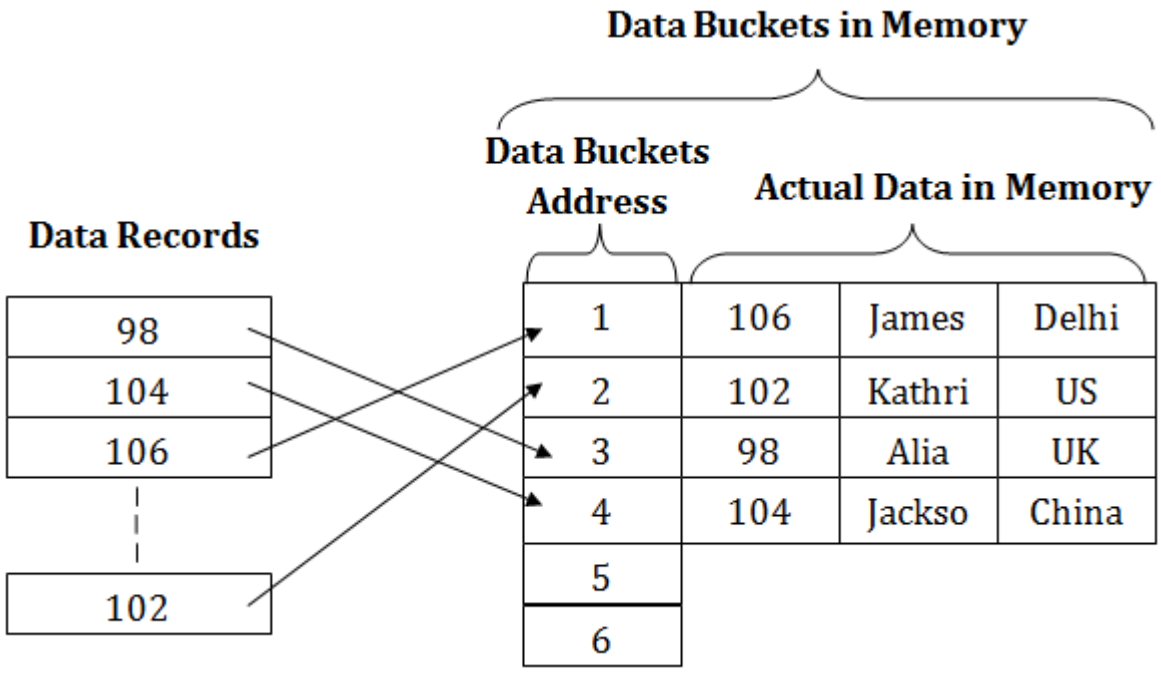


- Static Hashing
- Dynamic Hashing

A. Static Hashing

In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP_ID =103 using the hash function mod (5) then it will always result in same bucket address 3. Here, there will be no change in the bucket address.

Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.



Operations of Static Hashing

- **Searching a record**

When a record needs to be searched, then the same hash function retrieves the address of the bucket where the data is stored.

- **Insert a Record**

When a new record is inserted into the table, then we will generate an address for a new record based on the hash key and record is stored in that location.

- **Delete a Record**

To delete a record, we will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

- **Update a Record**

To update a record, we will first search it using a hash function, and then the data record is updated.

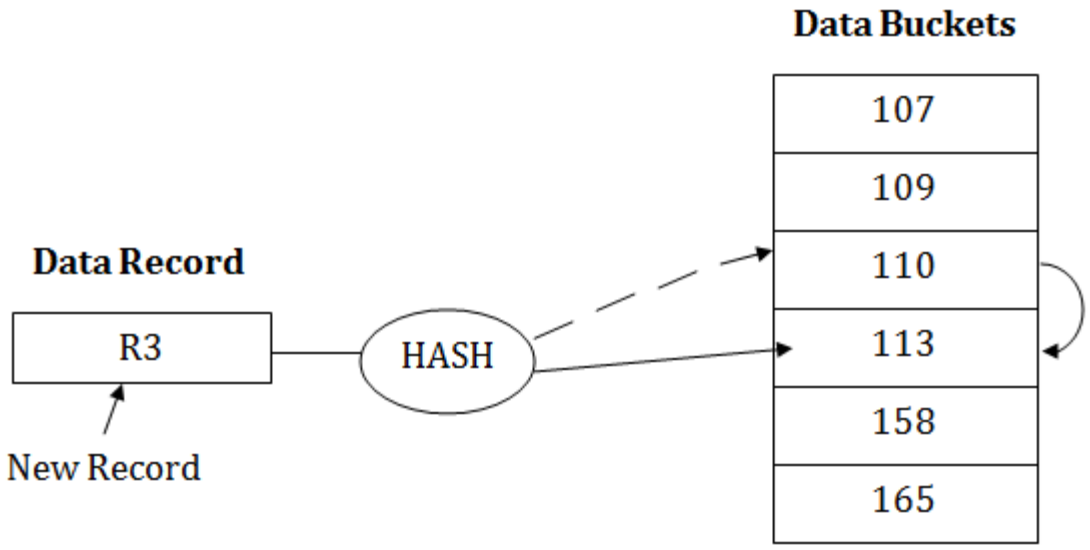
If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

To overcome this situation, there are various methods. Some commonly used methods are as follows:

1. Open Hashing

When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.

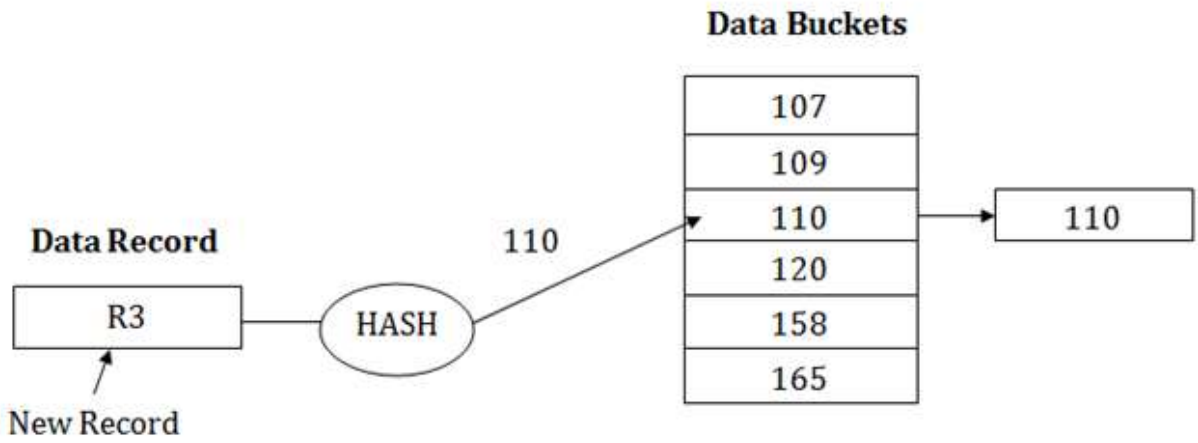
For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



2. Close Hashing

When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**.

For example: Suppose R3 is a new address which needs to be inserted into the table, the hash function generates address as 110 for it. But this bucket is full to store the new data. In this case, a new bucket is inserted at the end of 110 buckets and is linked to it.



B. Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.

- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

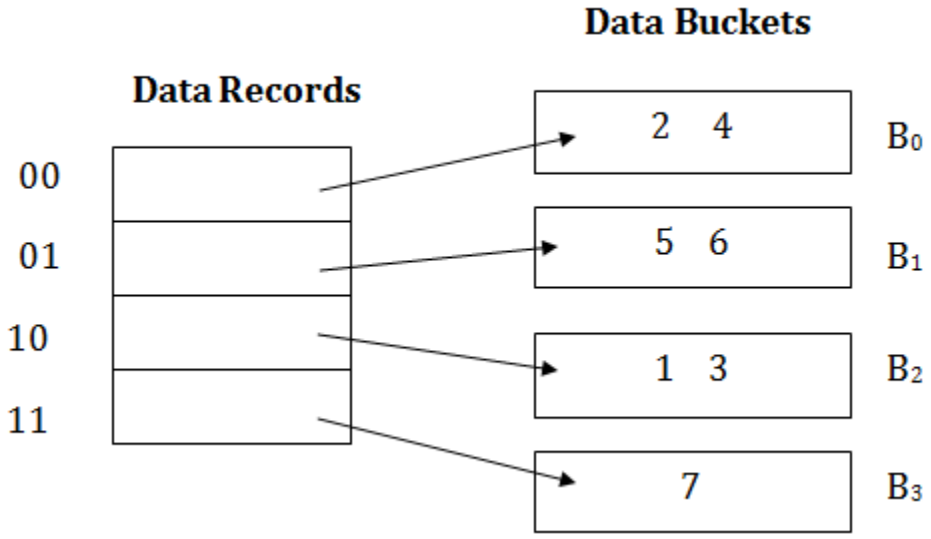
- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

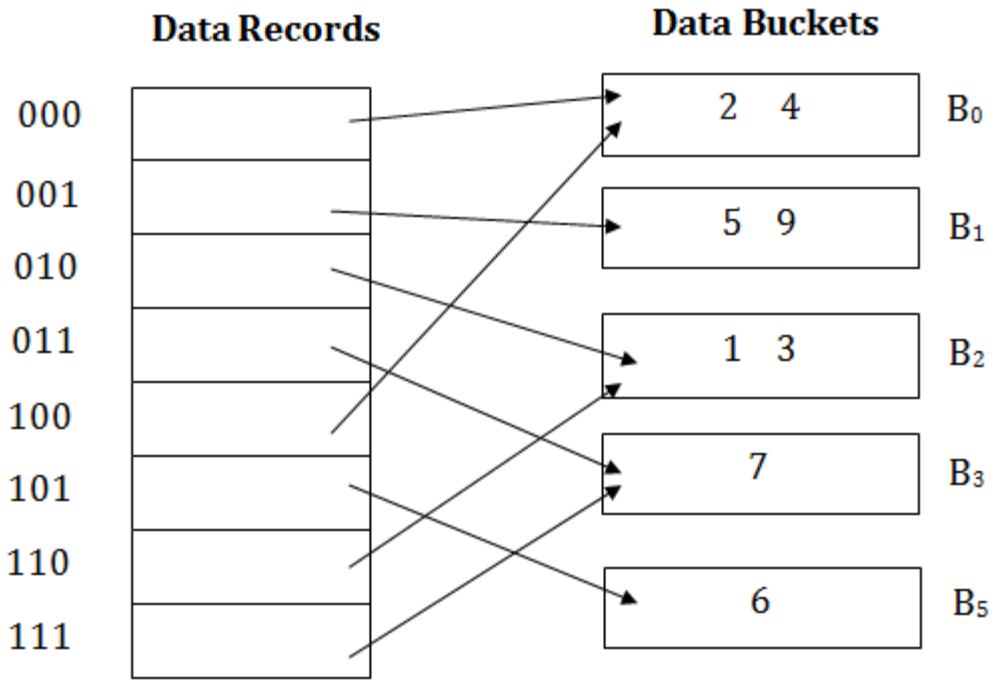
Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

The last two bits of 2 and 4 are 00. So it will go into bucket B0. The last two bits of 5 and 6 are 01, so it will go into bucket B1. The last two bits of 1 and 3 are 10, so it will go into bucket B2. The last two bits of 7 are 11, so it will go into B3.



Insert key 9 with hash address 10001 into the above structure:

- Since key 9 has hash address 10001, it must go into the first bucket. But bucket B1 is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.
- Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.