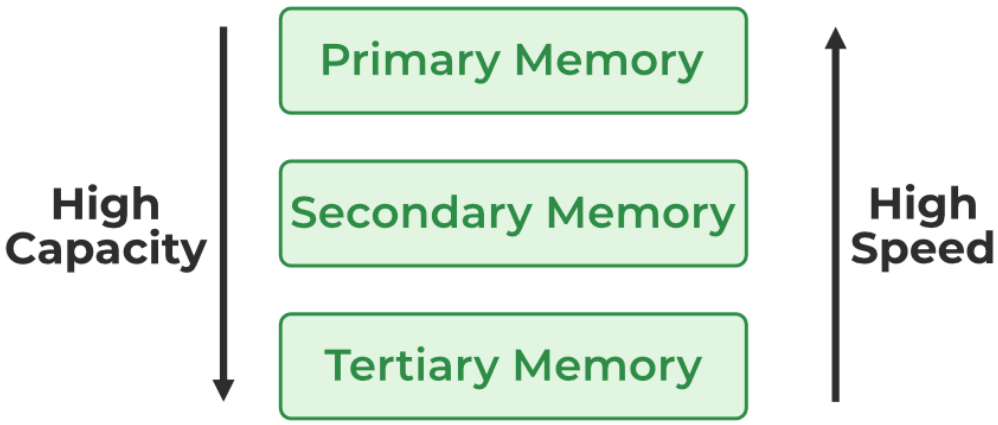
Storage Types in DBMS

Last Updated : 16 Nov, 2023

The records in databases are stored in file formats. Physically, the data is stored in electromagnetic format on a device. The electromagnetic devices used in database systems for data storage are classified as follows:

1. Primary Memory
2. Secondary Memory
3. Tertiary Memory



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 mod[illegible]ystem 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.

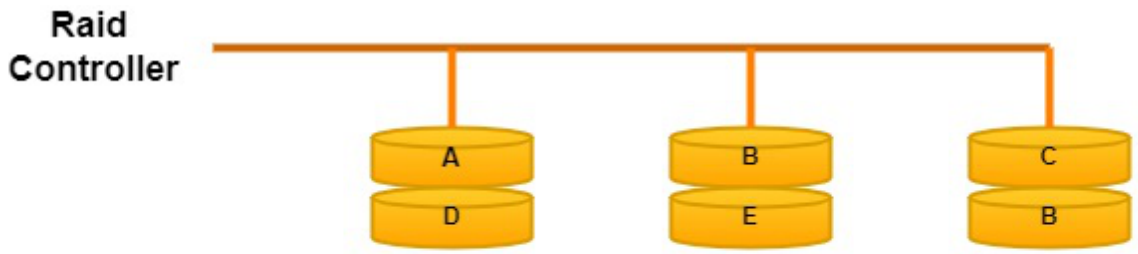
1. Magnetic Disks

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)

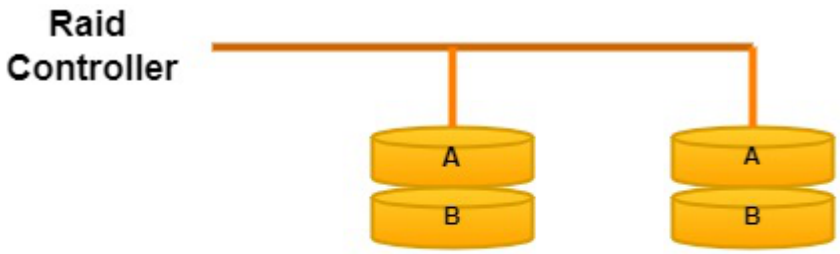
In the Redundant Array of Independent Disks technology, two or more secondary storage devices are connected so that the devices operate as one storage medium. A RAID array consists of several disks linked together for a variety of purposes. Disk arrays are categorized by their RAID levels.

- **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.



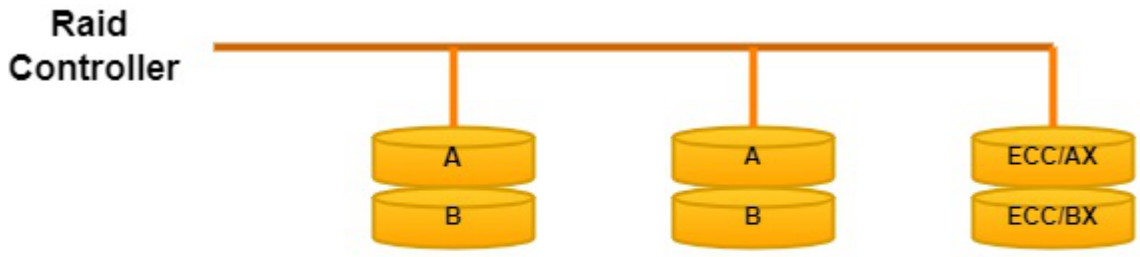
Raid-0

- **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.



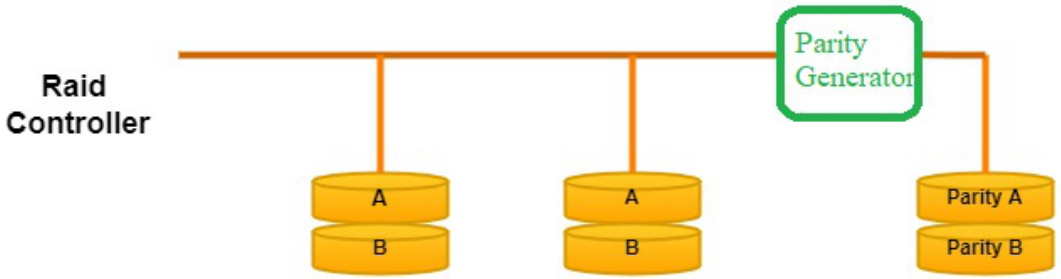
Raid-1

- **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.



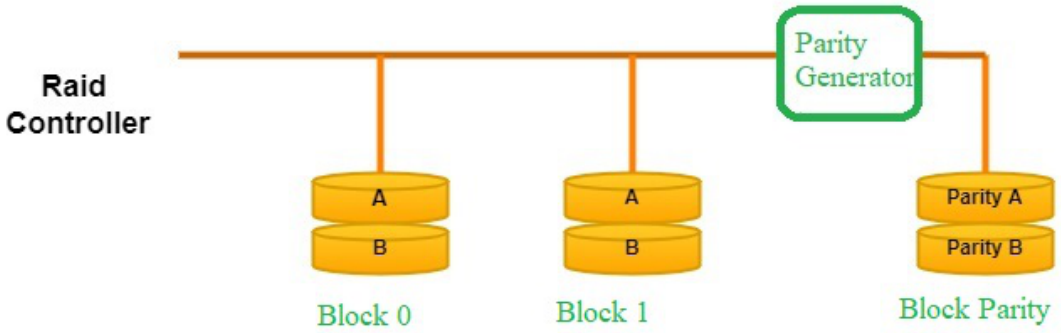
Raid-2

- **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.



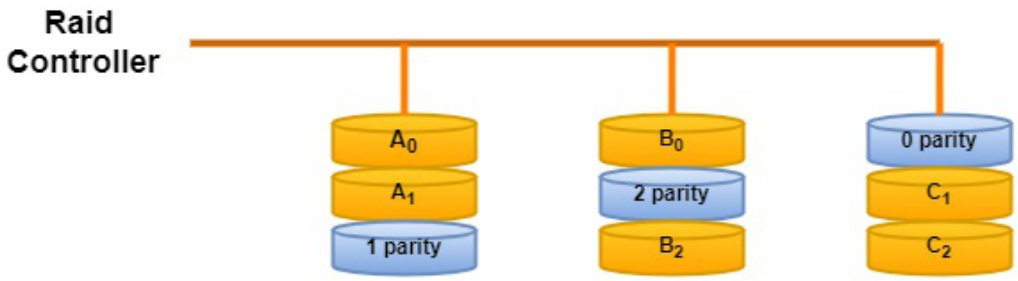
Raid-3

- **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.



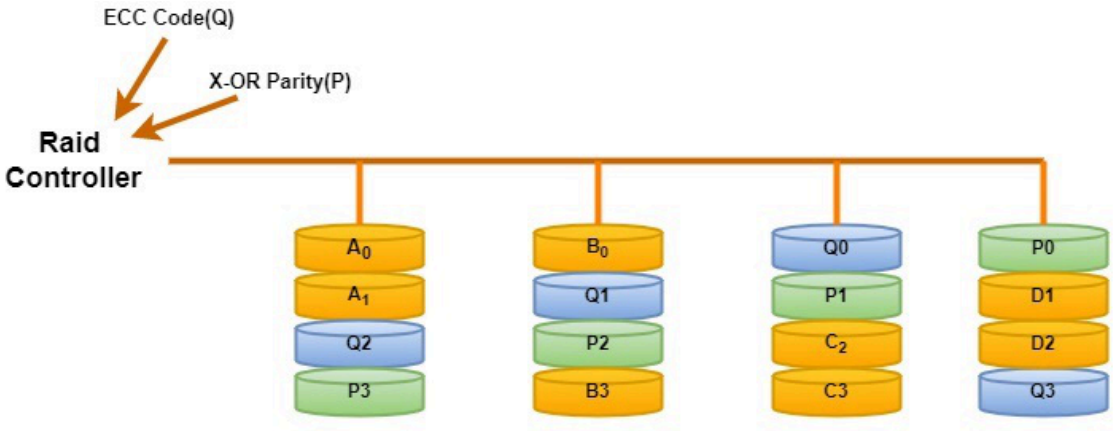
Raid-4

- **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

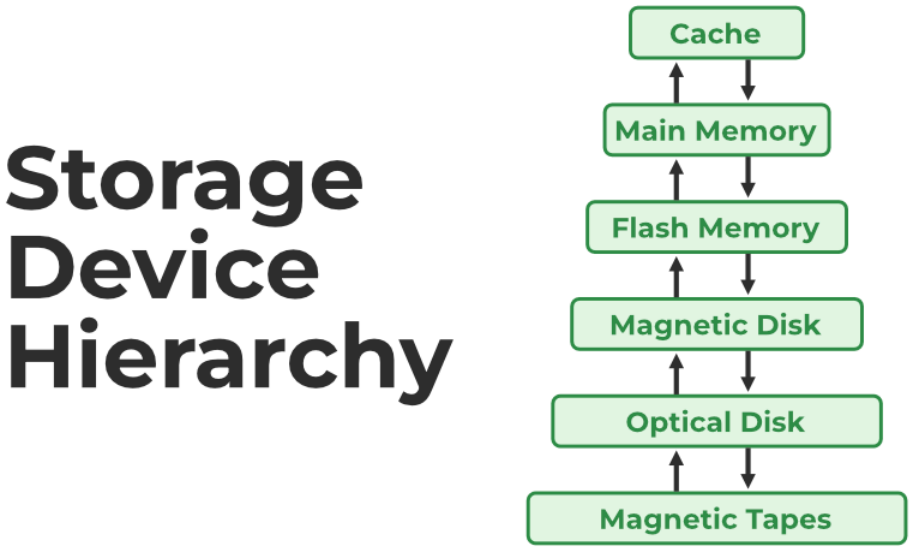
- **RAID 6:** The RAID 6 level extends the level 5 concept. A pair of independent parities are generated and stored on multiple disks at this level. A pair of independent parities are generated and stored on multiple disks at this level. Ideally, you need four disk drives for this level.



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

Conclusion

A DBMS must balance the utilization of primary, secondary, and tertiary memory. Secondary memory meets long-term storage demands, tertiary memory can be used for archiving, and primary memory guarantees quick access for active data. Using various storage types strategically in accordance with needs and patterns of data access is essential for optimal database performance.

Next Article

How to Test Internet Speed using Node.js ?

Similar Reads

Data Storage and Querying in DBMS

Database Management System is the collection of interrelated data/information or a set of programs that manages controls, and...

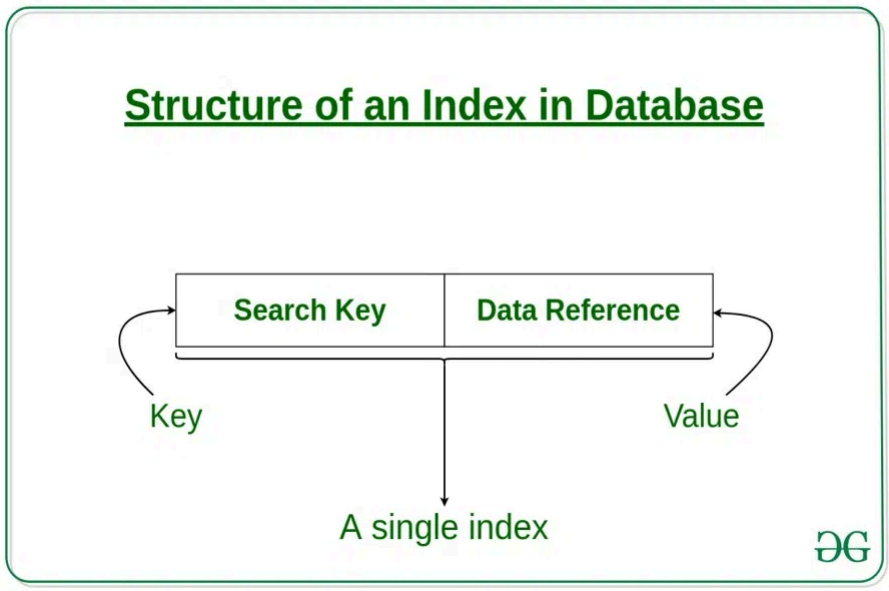
15+ min read

File Organization in DBMS - Set 1

Indexing in Databases – Set 1

Last Updated : 07 Nov, 2023

Indexing improves database performance by minimizing the number of disc visits required to fulfill a query. It is a data structure technique used to locate and quickly access data in databases. Several database fields are used to generate indexes. 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.

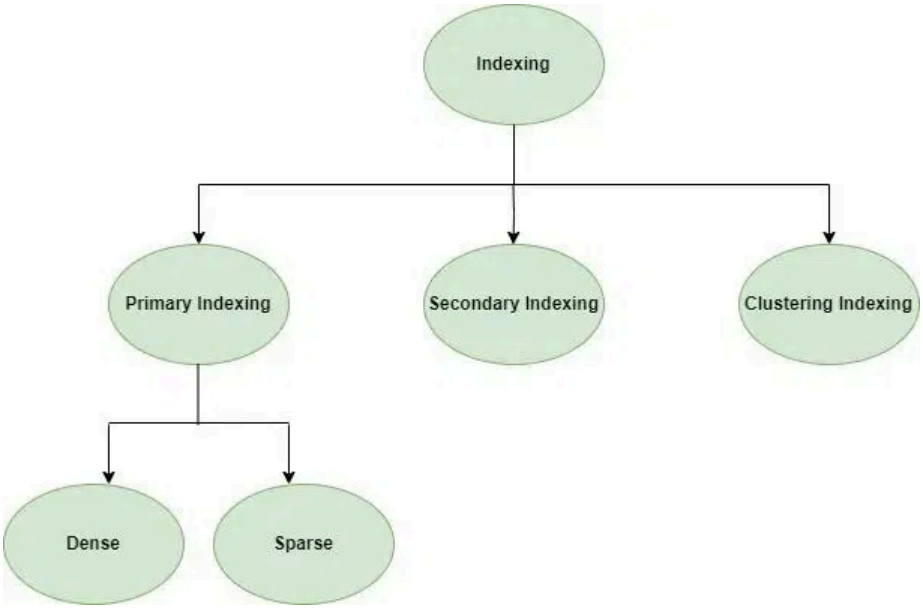


Structure of Index in Database

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.



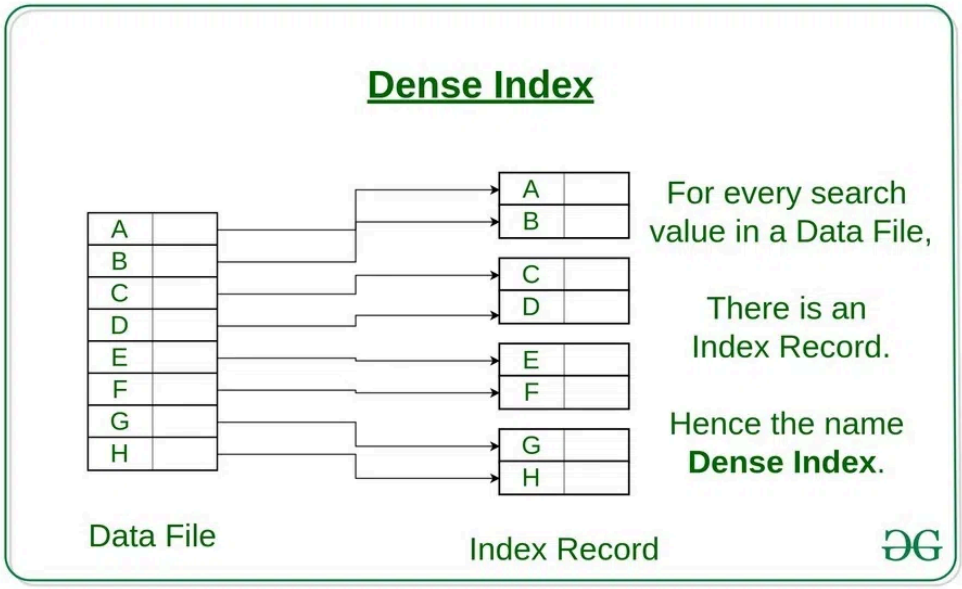
Structure of Index in Database

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

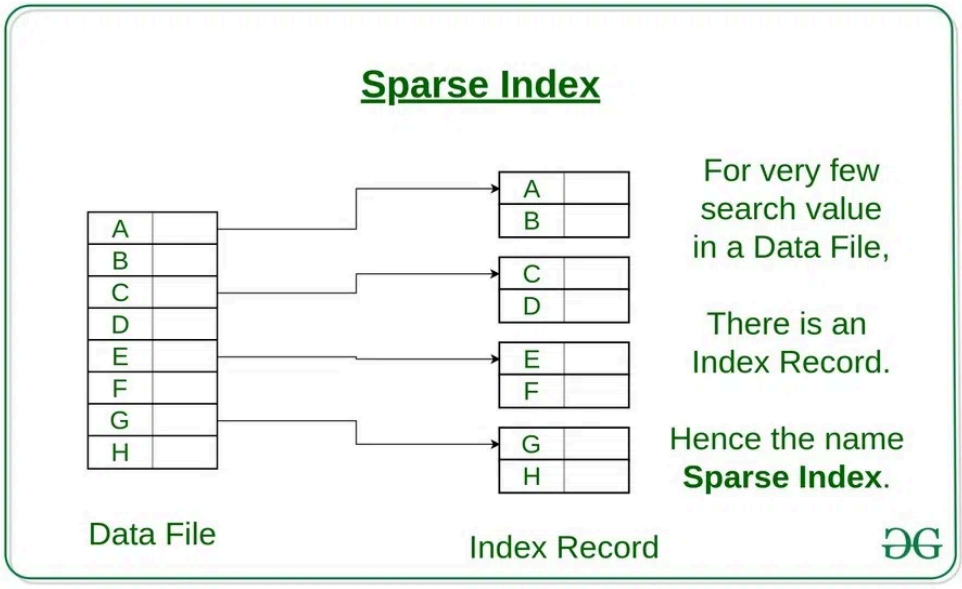
In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organizations might store the data in a dense or sparse format.

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



Dense Index

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



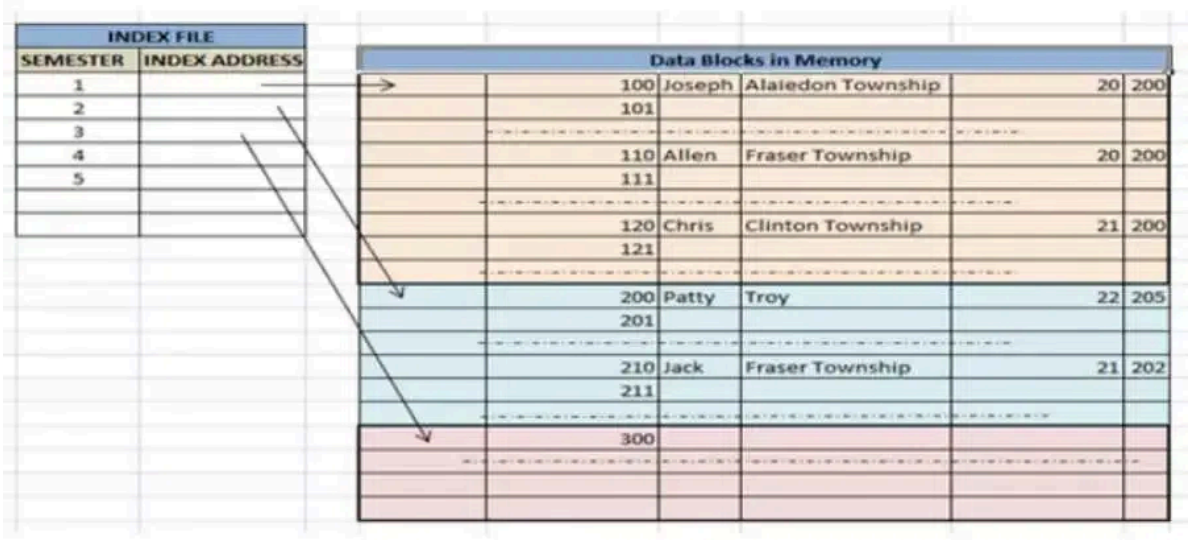
Hash File Organization

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.

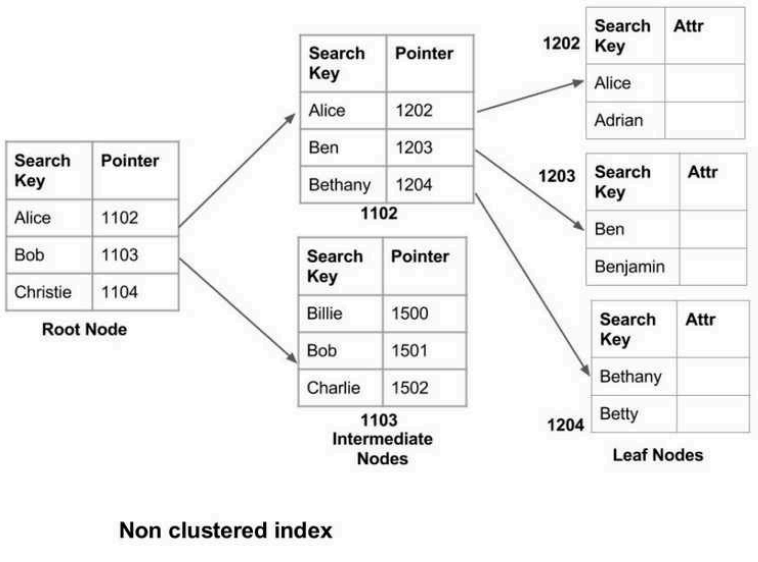
  Students studying each semester, for example, are grouped together. First-semester students, second-semester students, third-semester students, and so on are categorized.



Clustered Indexing

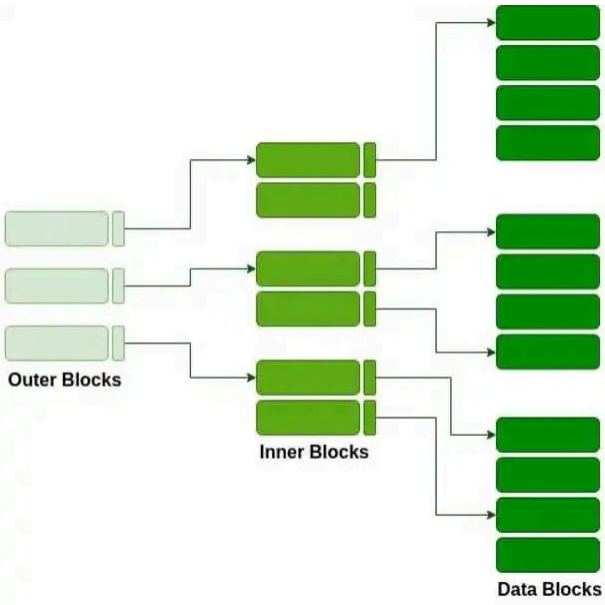
- **Primary Indexing:** This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are 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. In the case of a clustered index, data is directly present in front of the index.



Non Clustered Indexing

- **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 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 [illegible] retrieval.

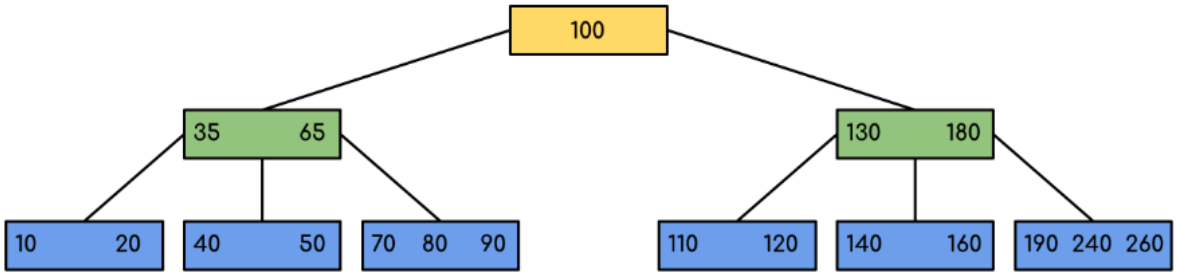
Introduction of B-Tree

Last Updated : 29 Jan, 2025

A B-Tree is a specialized m-way tree designed to optimize data access, especially on disk-based storage systems.

- In a B-Tree of order **m**, each node can have up to m children and m-1 keys, allowing it to efficiently manage large datasets.
- The value of m is decided based on disk block and key sizes.
- One of the standout features of a B-Tree is its ability to store a significant number of keys within a single node, including large key values. It significantly reduces the tree's height, hence reducing costly disk operations.
- B Trees allow faster data retrieval and updates, making them an ideal choice for systems requiring efficient and scalable data management. By maintaining a balanced structure at all times,
- B-Trees deliver consistent and efficient performance for critical operations such as search, insertion, and deletion.

Following is an example of a B-Tree of order 5 .



Properties of a B-Tree

A `B Tree` of order m can be defined as an `m-way` search tree which satisfies the following properties:

1. All leaf nodes of a `B-tree` are at the same level, i.e. they have the same depth (height of the tree).

2. The keys of each node of a `B tree` (in case of multiple keys), should be stored in the ascending order.
3. In a `B tree`, all non-leaf nodes (except root node) should have at least `m/2` children.
4. All nodes (except root node) should have at least `m/2 - 1` keys.
5. If the root node is a leaf node (only node in the tree), then it will have no children and will have at least one key. If the root node is a non-leaf node, then it will have at least `2 children` and at least one key.
6. A non-leaf node with n-1 key values should have n non NULL children.

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

Interesting Facts about B-Tree

- *The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:* $h_{min} = \lceil \log_m(n+1) \rceil - 1$
- *The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is:* $h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$ *and* $t = \lceil \frac{m}{2} \rceil$

Need of a B-Tree

The B-Tree data structure is essential for several reasons:

- **Improved Performance Over M-way Trees**:
  While M-way trees can be either balanced or skewed, B-Trees are always self-balanced. This self-balancing property ensures fewer levels in the tree, significantly reducing access time compared to M-way trees. This makes B-Trees particularly suitable for external storage systems where faster data retrieval is crucial.

- **Optimized for Large Datasets**:
  B-Trees are designed to handle millions of records efficiently. Their

reduced height and balanced structure enable faster sequential access to data and simplify operations like insertion and deletion. This ensures efficient management of large datasets while maintaining an ordered structure.

Operations on B-Tree

B-Trees support various operations that make them highly efficient for managing large datasets. Below are the key operations:

Sr. No.	Operation	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)
4.	Traverse	O(n)

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

Search Operation in B-Tree

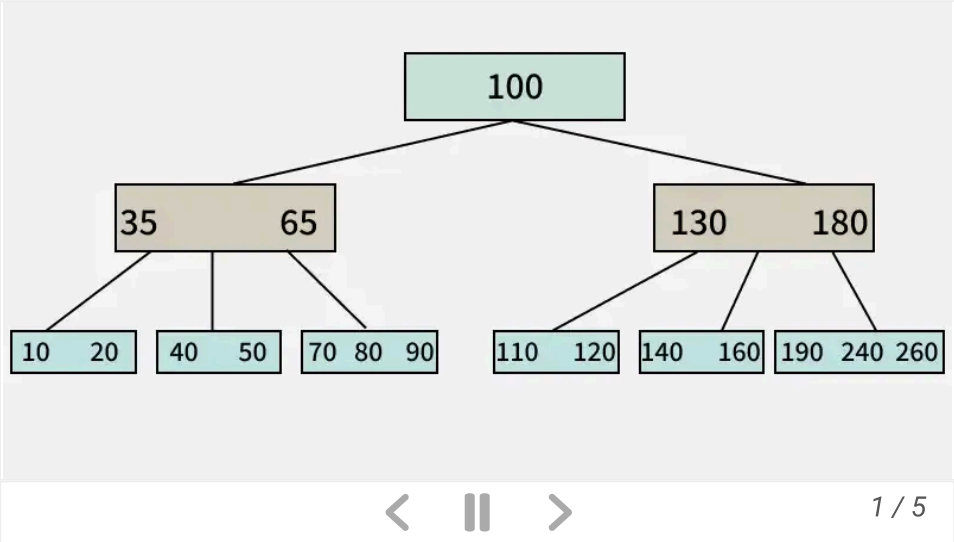
Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

- Start from the root and recursively traverse down.
- For every visited non-leaf node
    - If the current node contains k, return the node.
    - Otherwise, determine the appropriate child to traverse. This is the child just before the first key greater than k.
- If we reach a leaf node and don’t find k in the leaf node, then return NULL.

Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized

as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

Input: *Search 120 in the given B-Tree.*



The key 120 is located in the leaf node containing 110 and 120. The search process is complete.

Algorithm for Searching an Element in a B-Tree

C++ C Java Python C# JavaScript

```
struct Node {
    int n;
    int key[MAX_KEYS];
    Node* child[MAX_CHILDREN];
    bool leaf;
};

Node* BtreeSearch(Node* x, int k) {
    int i = 0;
    while (i < x->n && k > x->key[i]) {
        i++;
    }
    if (i < x->n && k == x->key[i]) {
        return x;
    }
    if (x->leaf) {
        return nullptr;
    }
```

```
    return BtreeSearch(x->child[i], k);
}
```

To read more about various operations on B-Tree refer below links:

- Insert Operation in B-Tree
- Delete Operation in B-Tree

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.
- For small datasets, the search time in a B-Tree might be slower compared to a binary search tree, as each node may contain multiple keys.

Introduction of B+ Tree

Last Updated : 16 Apr, 2025

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 this 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.

Difference Between B+ Tree and B Tree

Some differences between B+ Tree and B Tree are stated below.

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

Parameters	B+ Tree	B Tree
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

Implementation of B+ Tree

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of the B-tree used for indexing, however, is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record. B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their

corresponding data pointers to the disk file block, in order to access them.

Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record. From the above discussion, it is apparent that a B+ tree, unlike a B-tree, has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes.

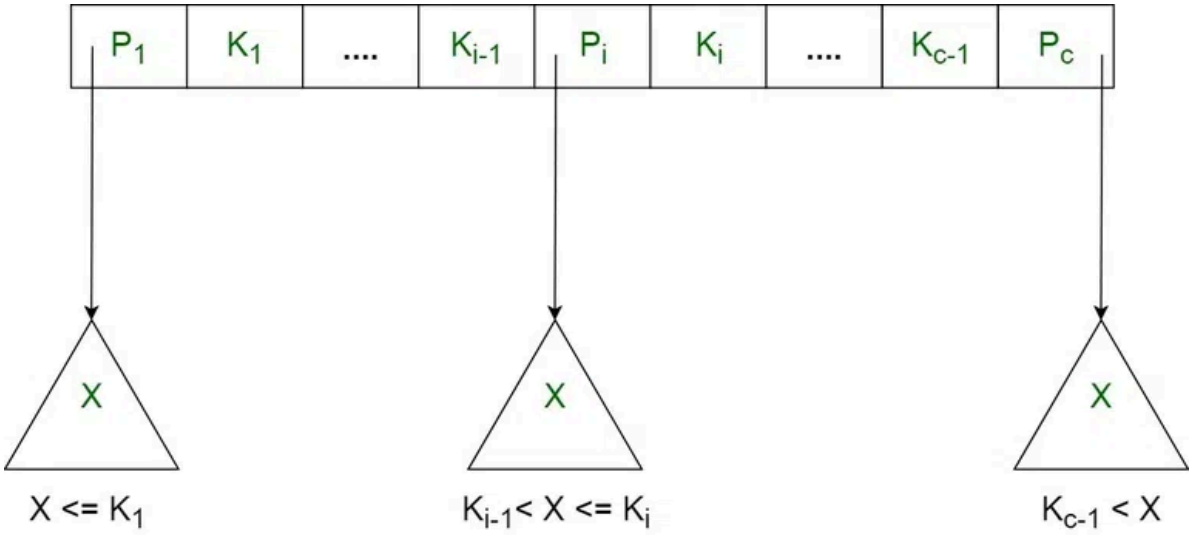
Structure of B+ Trees

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.

The Structure of the Internal Nodes of a B+ Tree of Order 'a' is as Follows

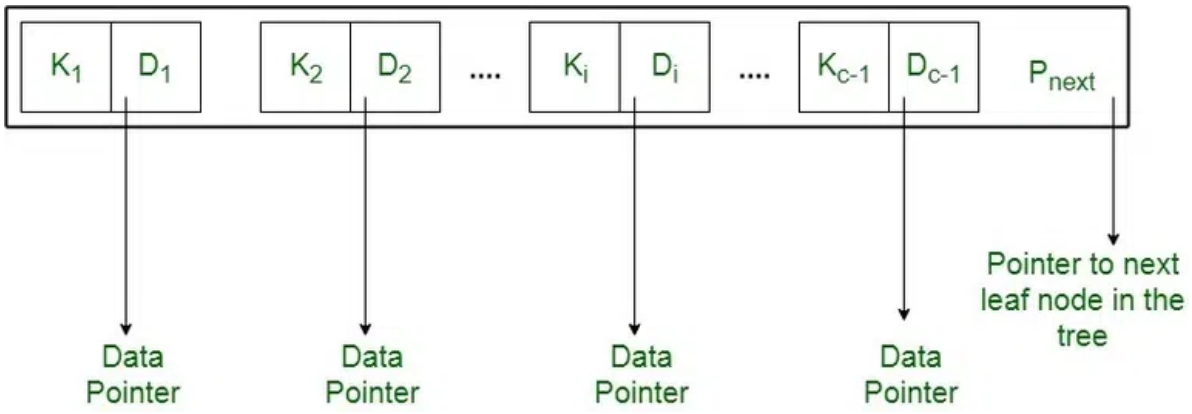
- Each internal node is of the form: <P_1, K_1, P_2, K_2, ….., P_{c-1}, K_{c-1}, P_c> where c <= a and each **P_i is a tree pointer (i.e points to another node of the tree)** and, each **K_i is a key-value** (see diagram-I for reference).
- Every internal node has : $K_1 < K_2 < …. < K_{c-1}$
- For each search field value 'X' in the sub-tree pointed at by P_i, the following condition holds: $K_{i-1} < X <= K_i$, for 1 < I < c and, $K_{i-1} < X$, for i = c (See diagram I for reference)
- Each internal node has at most 'aa tree pointers.
- The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
- If an internal node has 'c' pointers, c <= a, then it has 'c – 1' key values.



Structure of Internal Node

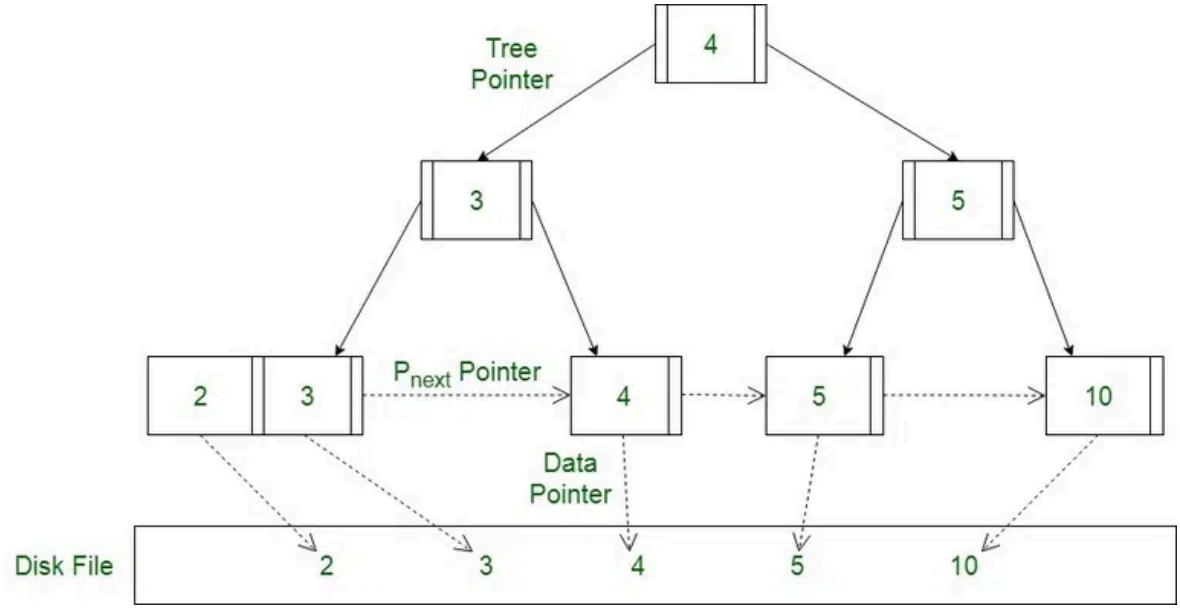
The Structure of the Leaf Nodes of a B+ Tree of Order 'b' is as Follows

- Each leaf node is of the form: <<K_1, D_1>, <K_2, D_2>, ….., <K_{c-1}, D_{c-1}>, P_{next}> where c <= b and each **D_i is a data pointer (i.e points to actual record in the disk whose key value is K_i or to a disk file block containing that record)** and, each **K_i is a key value** and, **P_{next} points to next leaf node in the B+ tree** (see diagram II for reference).
- Every leaf node has : K_1 < K_2 < …. < K_{c-1}, c <= b
- Each leaf node has at least \ceil(b/2) values.
- All leaf nodes are at the same level.



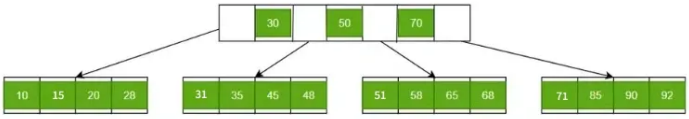
Structure of Lead Node

Diagram-II Using the P_{next} pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk



Tree Pointer

Searching a Record in B+ Trees



B+ tree

Let us suppose we have to find 58 in the B+ Tree. We will start by fetching from the root node then we will move to the leaf node, which might contain a record of 58. In the image given above, we will get 58 between 50 and 70. Therefore, we will we are getting a leaf node in the third leaf node and get 58 there. If we are unable to find that node, we will return that 'record not founded' message.

Insertion in B+ Trees

Insertion in B+ Trees is done via the following steps.

- Every element in the tree has to be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.
- Insert the key into the leaf node in increasing order if there is no overflow.

For more, refer to Insertion in a B+ Trees.

Deletion in B+Trees

Deletion in B+ Trees is just not deletion but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

For more, refer to Deletion in B+ Trees.

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 P_{next} 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.

Application of B+ Trees

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

Next Article

Insertion in a B+ tree

Hashing in DBMS

Last Updated : 03 Jul, 2024

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 data retrieval time are minimized. So, to counter this problem hashing technique is used. In this article, we will learn about various hashing techniques.

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.

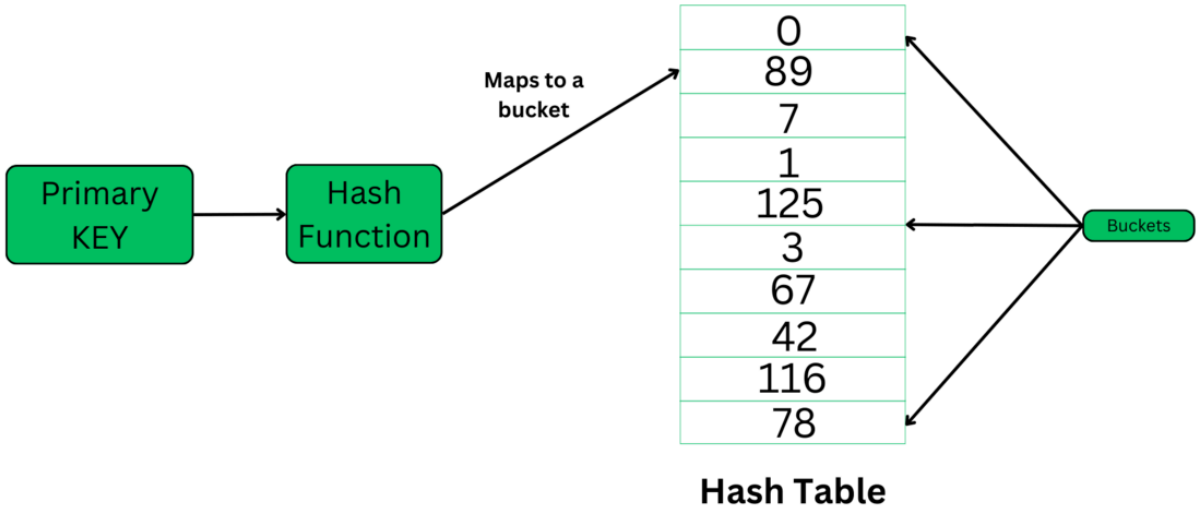
Now, there are 2 possibilities:

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

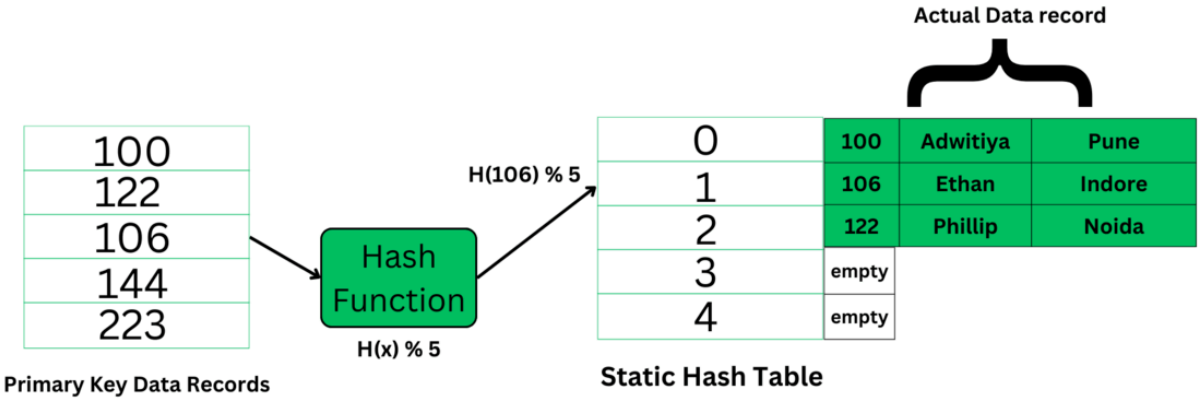
There are two primary hashing techniques in DBMS.

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:

1. Chaining

Chaining 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.

2. 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, 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

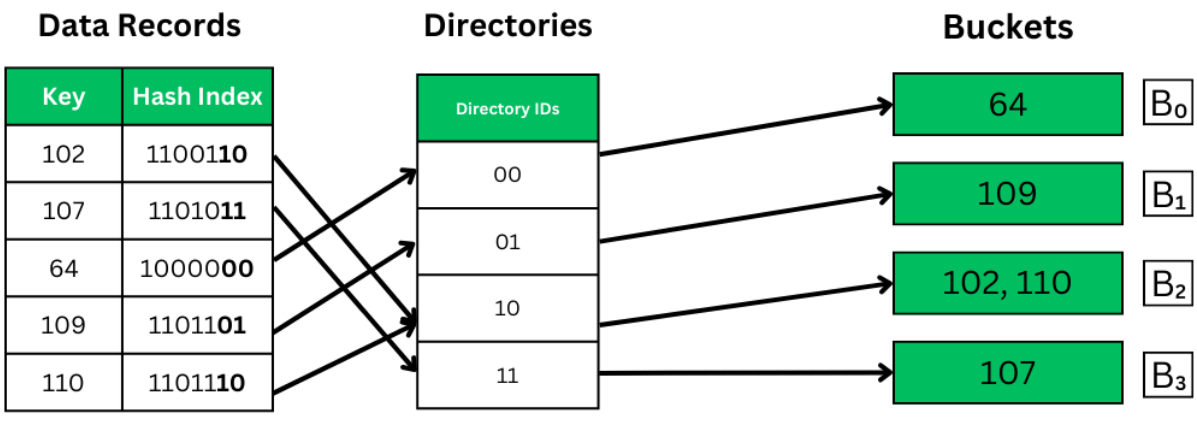
as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.

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.



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.

Next Article

When to Use Agile Model?

Similar Reads

Dynamic Hashing in DBMS

In this article, we will learn about dynamic hashing in DBMS. Hashing in DBMS is used for searching the needed data on the disc. As static...

15+ min read

Static Hashing in DBMS

Static hashing refers to a hashing technique that allows the user to search over a pre-processed dictionary (all elements present in the...

15+ min read

Serializability in DBMS

In this article, we are going to explain the serializability concept and how this concept affects the DBMS deeply, we also understand the concept ...

15+ min read

Structure of Database Management System

A Database Management System (DBMS) is software that allows users to define, store, maintain, and manage data in a structured and efficient...

15+ min read