PRIMARY INDEXING

What is Primary Indexing?

- A primary index is an index on a set of fields that includes the unique primary key for the table. The primary index is automatically created when a primary key constraint is defined. It ensures that the data is stored in a way that allows efficient access to the primary key.

- A Primary Index is an ordered file whose records are of fixed length with two fields. The first field of the index is the primary key of the data file in an ordered manner, and the second field of the ordered file contains a block pointer that points to the data block where a record containing the key is available.

Primary Indexing

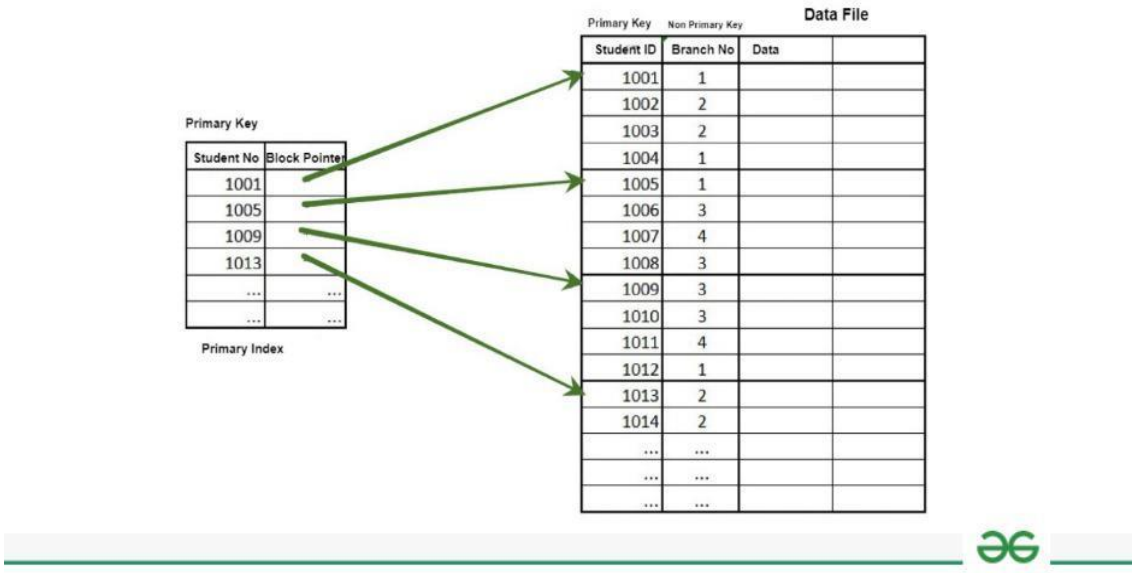
Characteristics of a Primary Index:

1. It is based on the primary key of the table.
2. The primary key must have unique values.
3. The index entries are sorted in the same order as the data file (hence often a clustered index).
4. The index has one entry for each block in a data file, not for each record.

Working of Primary Indexing

- In primary indexing, the data file is sorted or clustered based on the primary key as shown in the below figure.
- An **index file** (also known as the **index table**) is created alongside the data file.
- The index file contains pairs of primary key values and pointers to the corresponding data records.

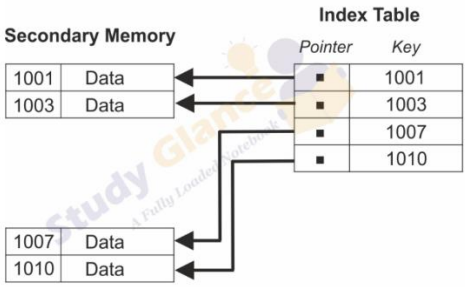
- Each entry in the index file corresponds to a block or page in the data file.



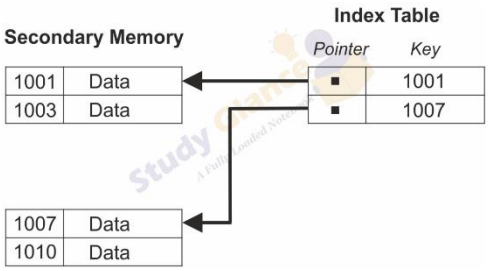
Primary Indexing

Types of Primary Indexing

- **Dense Indexing**: In dense indexing it has an index entry for every search key value in the data file. This approach ensures efficient data retrieval but requires more storage space.
  No of Index Entry = No of DB Record



- **Sparse Indexing**: Sparse indexing involves having fewer index entries than records. The index entries point to blocks of records rather than individual records. While it reduces storage overhead, it may require additional disk accesses during retrieval.
  No of Index Entry = No of Block



Advantages of Primary Indexing

- Primary indexing allows fast retrieval of records based on their primary index values.
- Primary indexing reduces the need for full-table scans or sequential lookups, due to which it reduce disk I/O operations.
- The data file is organized based on the primary key, ensuring that records are stored in a logical order. This organization simplifies range queries and other operations.

Disadvantages of Primary Indexing

- Primary indexing requires additional space to store the index field alongside the actual data records. This extra storage can impact overall system costs, especially for large datasets.
- When records are added or deleted, the data file needs reorganization to maintain the order based on the primary key.
- After record deletion, the space used by the deleted record must be released for reuse by other records. Managing this memory cleanup can be complex.

CLUSTERED INDEXING

Clustered indexing in databases is a type of indexing where the data rows are physically ordered based on the key values. This type of indexing is beneficial for range queries and queries that require accessing multiple rows in a sorted order. When a table is clustered, the rows are stored in the order of the clustering key, which can be a primary key or a unique key.

Characteristics of Clustering Index:

1. **Physical Order**: The rows in a table are stored in the same order as the index key. This means that the data is physically rearranged on the disk to match the index order.
2. **Single Cluster Index per Table**: Since clustering alters the physical order of rows, a table can have only one cluster index.
3. **Efficient Range Queries**: Clustering indexes are particularly efficient for range queries because the rows are stored in sorted order. This reduces the number of I/O operations needed to retrieve a range of values.
4. **Improved Performance**: For queries that retrieve multiple rows or require sorting, clustering indexes can significantly improve performance.

Example:

Imagine a `Books` table with the following records:

BookID	Title	Genre
3	A Tale of Two Cities	Fiction
1	Database Systems	Academic
4	Python Programming	Technical
2	The Great Gatsby	Fiction

If we create a clustered index on `BookID`, the physical order of records would be rearranged based on the ascending order of `BookID`.

BookID	Title	Genre
1	Database Systems	Academic
2	The Great Gatsby	Fiction
3	A Tale of Two Cities	Fiction
4	Python Programming	Technical

Now, when you want to find a book based on its ID, the DBMS can quickly locate the data because the data is stored in the order of the BookID.

Advantages

- **Improved Query Performance:** Clustering indexing results in faster query performance, as the data is stored in a way that makes it easier to retrieve the desired information. This is because the index is built based on the clustered data, reducing the number of disk I/Os required to retrieve the data.
- **Reduced Disk Space Usage:** Clustering indexing reduces the amount of disk space required to store the index. This is because the index contains only the information necessary to retrieve the data, rather than storing a copy of the data itself.
- **Better Handling of Complex Queries:** Clustering indexing provides better performance for complex queries that involve multiple columns. This is because the data is stored in a way that makes it easier to retrieve the relevant information.
- **Improved Insert Performance:** Clustering indexing can result in improved insert performance, as the database does not have to update the index every time a new record is inserted.
- **Improved Data Retrieval:** Clustering indexing can also improve the efficiency of data retrieval operations. In a clustered index, the data is stored in a logical order, which makes it easier to locate and retrieve the data. This can result in faster data retrieval times, particularly for large databases.

Disadvantages

- **Increased Complexity:** Clustering indexing is a more complex technology compared to other indexing mechanisms, such as B-Tree indexing.
- **Reduced Update Performance:** Clustering indexing can result in reduced update performance, as the database must reorganize the data to reflect the changes.
- **Limited to One Clustered Index:** A table can have only one clustered index, as having multiple clustered indexes would result in conflicting physical orderings of the data.

Conclusion

Clustering indexing is a type of indexing mechanism that provides improved query performance, reduced disk space usage, and better handling of complex queries. It is best suited for use in large databases, where query performance is a concern, and the data can be organized in a meaningful way based on a specific column or set of columns. However, clustering indexing is a more complex technology compared to other indexing mechanisms and can result in reduced update performance. As with any technology, the decision to use clustering indexing should be based on a careful evaluation of the specific requirements of your database.

SECONDARY INDEXING

A secondary index in a database is an additional indexing structure that is used to improve the performance of queries that access data based on columns that are not the primary key or clustering key. Secondary indexes allow the database to quickly locate rows in a table without having to scan the entire table, thereby improving query performance.

Characteristics of Secondary Index:

1. **Non-Unique Columns**: Secondary indexes are typically created on non-primary key columns or non-unique columns to facilitate faster searches.
2. **Multiple Secondary Indexes**: A table can have multiple secondary indexes, each on different columns, to support various types of queries.
3. **Logical Order**: Unlike clustering indexes, secondary indexes do not affect the physical order of the rows in the table. They maintain a logical order that points to the physical location of the data.

How Secondary Index Works:

1. **Index Structure**: A secondary index typically uses a data structure like a B-tree or a hash table. Each entry in the index contains the indexed column value and a pointer to the corresponding row in the table.
2. **Query Optimization**: When a query is executed that involves the indexed column, the database engine can use the secondary index to quickly locate the rows that match the query criteria.
3. **Logical Mapping**: The index maintains a logical mapping from the indexed column values to the physical rows, allowing for efficient data retrieval without scanning the entire table.

Advantages of Secondary Index:

1. **Improved Query Performance**: Secondary indexes significantly improve the performance of queries that involve columns other than the primary key or clustering key.
2. **Support for Multiple Columns**: Multiple secondary indexes can be created on different columns to optimize various types of queries.
3. **Flexibility**: Secondary indexes provide flexibility in querying and can be added or removed without affecting the physical structure of the table.

Disadvantages of Secondary Index:

1. **Additional Storage**: Secondary indexes require additional storage space to maintain the index structure.
2. **Maintenance Overhead**: Insert, update, and delete operations may require additional processing to maintain the secondary indexes, leading to increased overhead.
3. **Potential for Fragmentation**: Frequent updates and deletions can cause fragmentation in the index, which may degrade performance over time.

Example:

Let's continue with the `Students` table:

RollNumber (Primary Key)	Name	Age
1001	John	20
1003	Alice	22
1007	Bob	21
1010	Clara	23

Assuming we want to create a secondary index on the `**Age**` column:

Dense Secondary Index on Age:

Age	Pointer to Record
20	Record 1
21	Record 3
22	Record 2
23	Record 4

Here, each age value has a direct pointer to the corresponding record.

If another student with an age of 22 is added:

RollNumber	Name	Age
1012	David	22

The dense secondary index would then be:

Age	Pointer to Record
20	Record 1
21	Record 3
22	Record 2, Record 5
23	Record 4

Conclusion

Secondary indexing is a crucial technique in databases that enhances the performance of queries involving non-primary key columns. By creating secondary indexes, databases can quickly locate rows based on various columns, reducing the need for full table scans and improving overall query efficiency. However, secondary indexes also introduce additional storage and maintenance overhead, which should be carefully managed to maintain optimal performance.

B+ TREES

- 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.

B+ trees are a type of self-balancing tree data structure commonly used in databases and file systems for efficient storage and retrieval of data. Here's a detailed overview of B+ trees in the context of database management systems (DBMS):

Structure of B+ Trees:

1. **Nodes:**
   - **Internal Nodes:** Contain keys and pointers to child nodes. These nodes facilitate navigation to locate the leaf node containing a specific key.
   - **Leaf Nodes:** Contain key-value pairs sorted by key. They also contain a pointer to the next leaf node, allowing sequential access.
2. **Keys and Pointers:**
   - Internal nodes store keys to guide searches, while leaf nodes store keys along with corresponding data (or pointers to data).
3. **Order:**
   - B+ trees have a parameter called the **order (m)**, which determines the maximum number of children each internal node can have.
   - Leaf nodes can store up to m-1 key-value pairs.

Advantages in DBMS:

1. **Efficient for Range Queries:**
   - B+ trees are particularly efficient for range queries due to their balanced structure and sequential access capability through leaf nodes.
2. **Disk I/O Efficiency:**

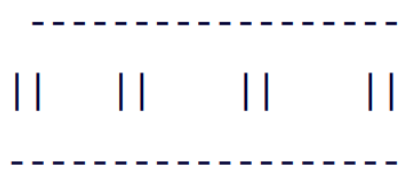
- Since B+ trees are balanced, the depth of the tree remains shallow, leading to fewer disk accesses for operations like search, insert, and delete.

3. **Ordered Storage:**
   - Data is stored in order of keys, making range queries faster as they can sequentially scan leaf nodes.
4. **Support for Equality and Range Searches:**
   - B+ trees support both equality searches (finding exact matches) and range queries (finding keys within a range), making them versatile for various query types encountered in databases.

Operations on B+ Trees:

1. **Search:**
   - Begins at the root and recursively searches down to the appropriate leaf node using key comparisons.
2. **Insertion:**
   - Follows the search path to find the correct leaf node. Inserts the new key-value pair while maintaining the tree's balance by possibly splitting nodes.
3. **Deletion:**
   - Similar to insertion, deletion maintains the balance of the tree by merging nodes if necessary, after removing a key.

Let's say we have a B⁺ Tree of order 4, and we want to insert the keys `[10, 20, 5, 6, 12, 30, 7, 17]` into an initially empty tree.



Insertion

1. Insert 10:
- The tree is empty, so 10 becomes the root.

2. Insert 20:
- There's room in the leaf node for 20.

```
[10, 20]
```

3. Insert 5:
- Still room in the leaf node for 5.

```
[5, 10, 20]
```

4. Insert 6:
- The leaf node is full; split it and promote the smallest key in the right node to be the new root.

```
    [10]
   /    \
[5, 6]  [10, 20]
```

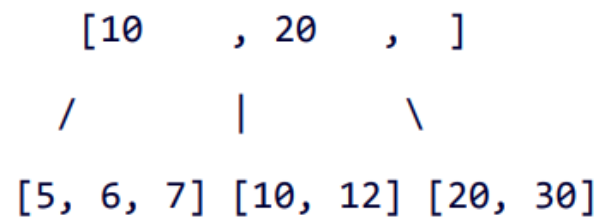
5. Insert 12:
- Insert into the appropriate leaf node.

```
    [10, , ]
   /    \
[5, 6]  [10, 12, 20]
```

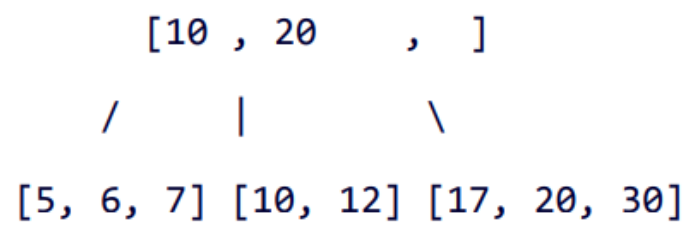
6. Insert 30:
- Need to split the right leaf node, promote 20.

```
    [10 , 20    , ]
   /    |        \
[5, 6] [10, 12] [20, 30]
```

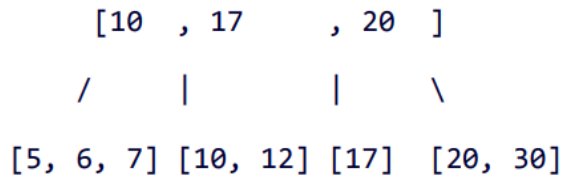
7. Insert 7:
- Insert into the appropriate leaf node.



8. Insert 17:
- Insert into the appropriate leaf node and split.



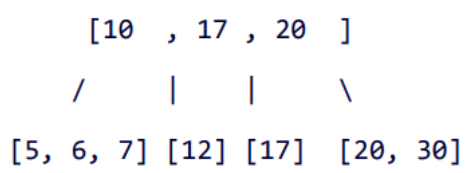
Here, the middle internal node gets split, and 17 is promoted.



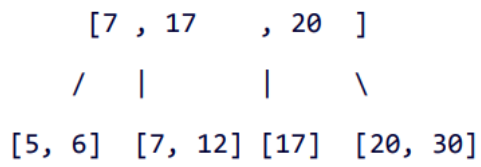
Search (for 12):
- Start at the root, go down the second child because 12 > 10 and 12 < 17, and find 12 in the corresponding leaf node.

Deletion (of 10):

1. Remove 10 from the leaf node.



2. Since the key 10 is also present in the internal node, we replace it with its in-order predecessor (or successor based on design), which is 7.



HASHING

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.

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

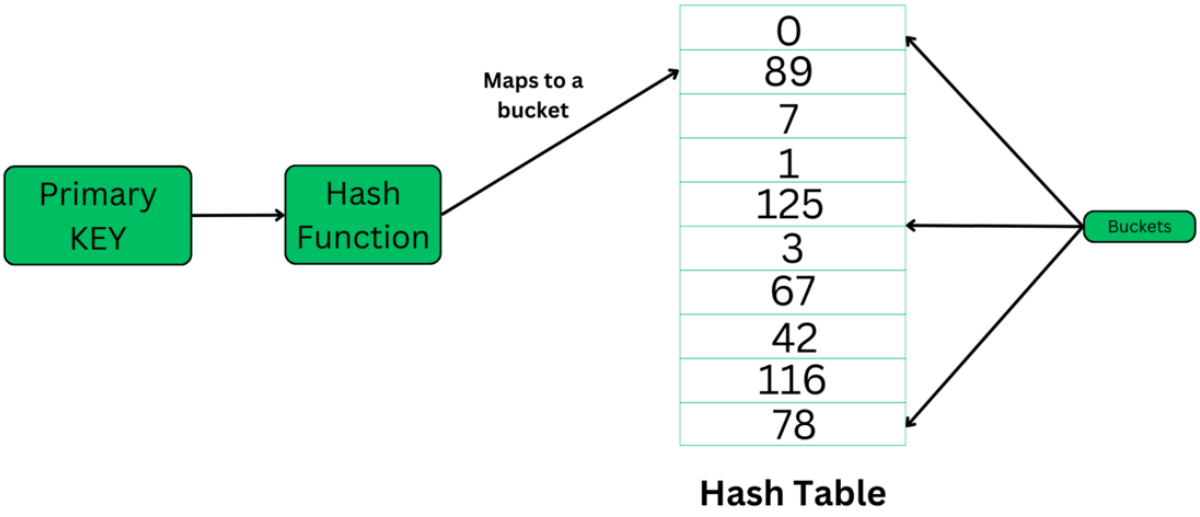
The hash function generates a hash index through the primary key of the data record.

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

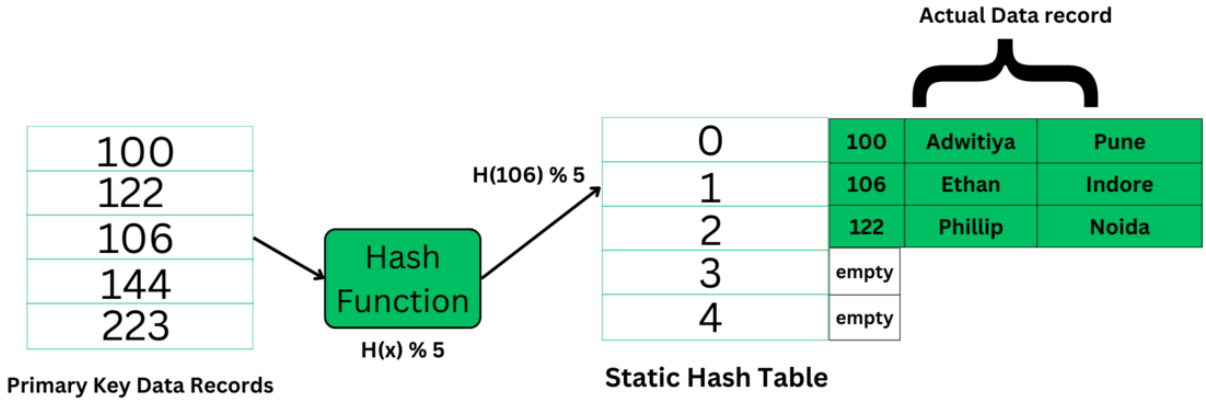
There are two primary hashing techniques in DBMS.

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

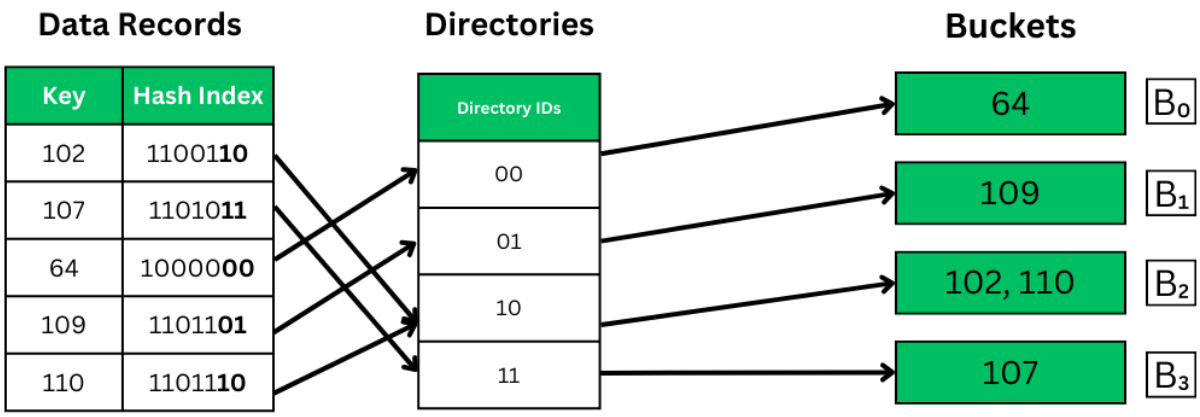
same value for any data record it can still be stored in a bucket by adding a new node.
However, this will give rise to the problem bucket skew that is, if the hash function keeps generating the same value again and again then the hashing will become inefficient as the remaining data buckets will stay 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.



Dynamic Hashing – mapping

As we can see in the above image, the k bits from LSBs are taken in the hash index to map to their appropriate buckets through directory IDs. The hash indices point to the directories, and the k bits are taken from the directories' IDs and then mapped to the buckets. Each bucket holds the value corresponding to the IDs converted in binary.