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.

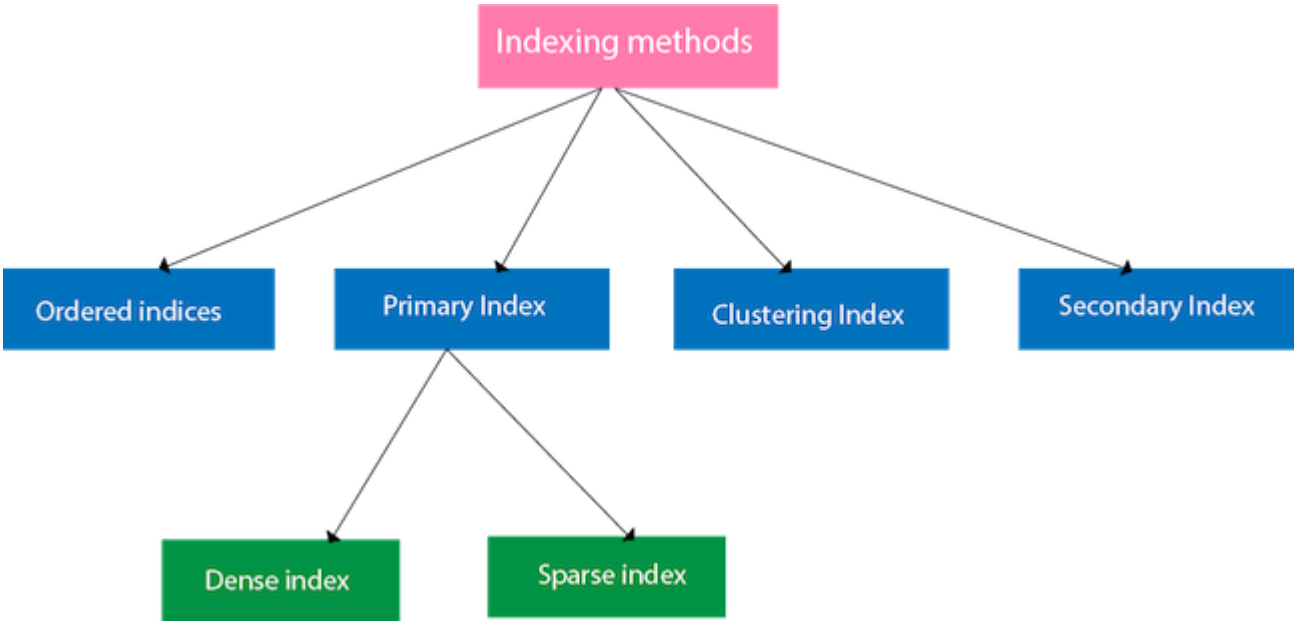
Index structure:

Indexes can be created using some database columns.

Search key	Data Reference

Fig: Structure of Index

Indexing Methods



Ordered indices

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

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

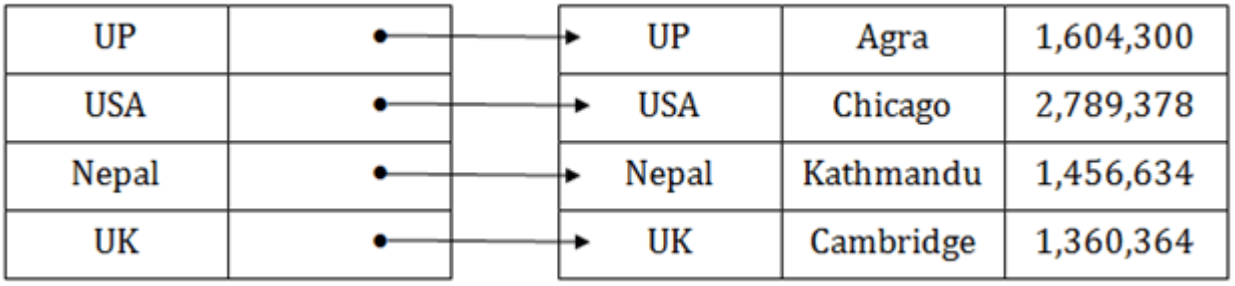
- In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543*10=5430 bytes.
- In the case of an index, we will search using indexes and the DBMS will read the record after reading 542*2= 1084 bytes which are very less compared to the previous case.

Primary Index

- If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
- As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: Dense index and Sparse index.

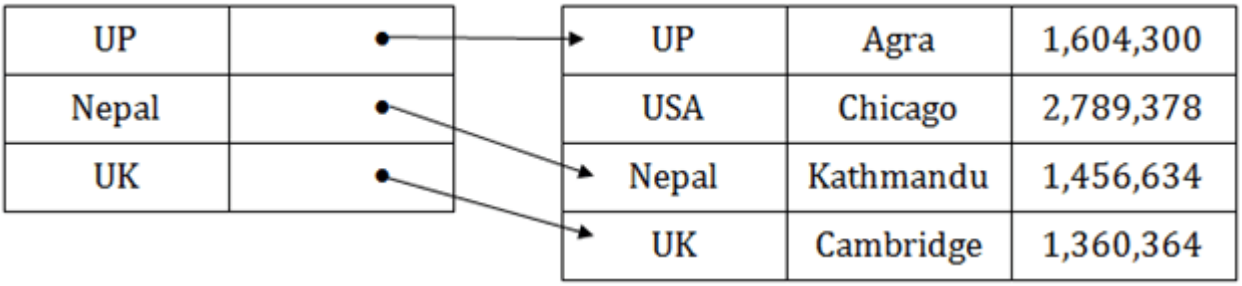
Dense index

- The dense index contains an index record for every search key value in the data file. It makes searching faster.
- In this, the number of records in the index table is same as the number of records in the main table.
- It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.



Sparse index

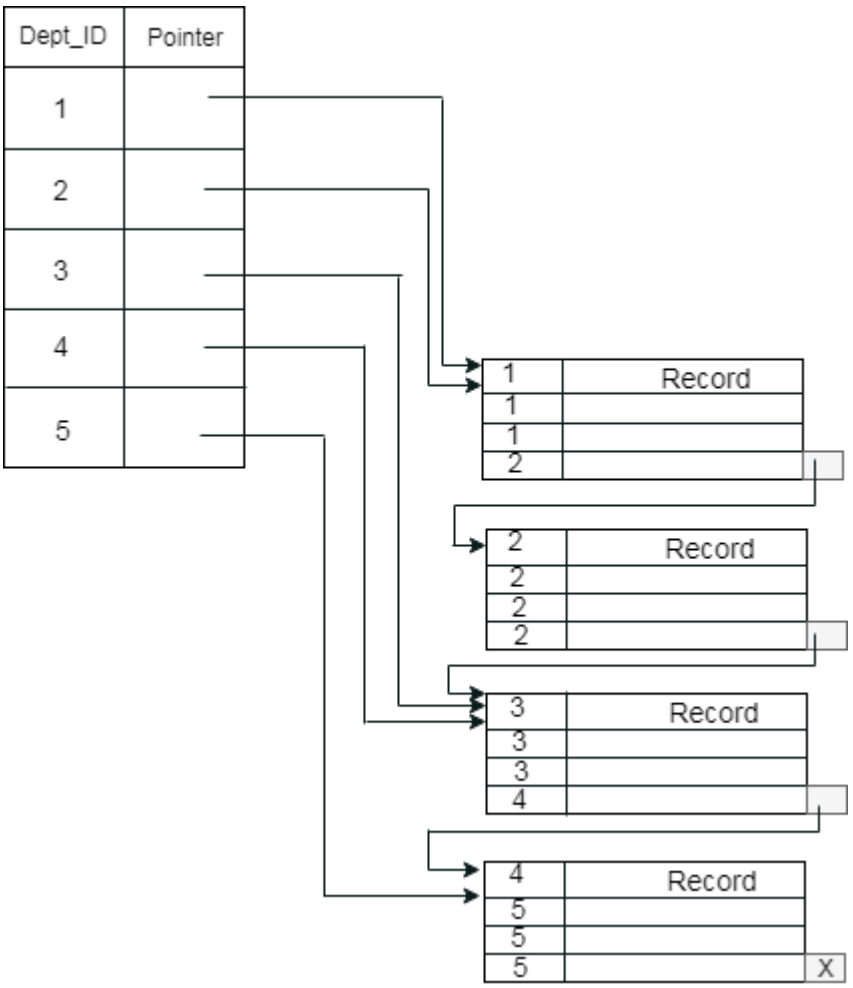
- In the data file, index record appears only for a few items. Each item points to a block.
- In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.



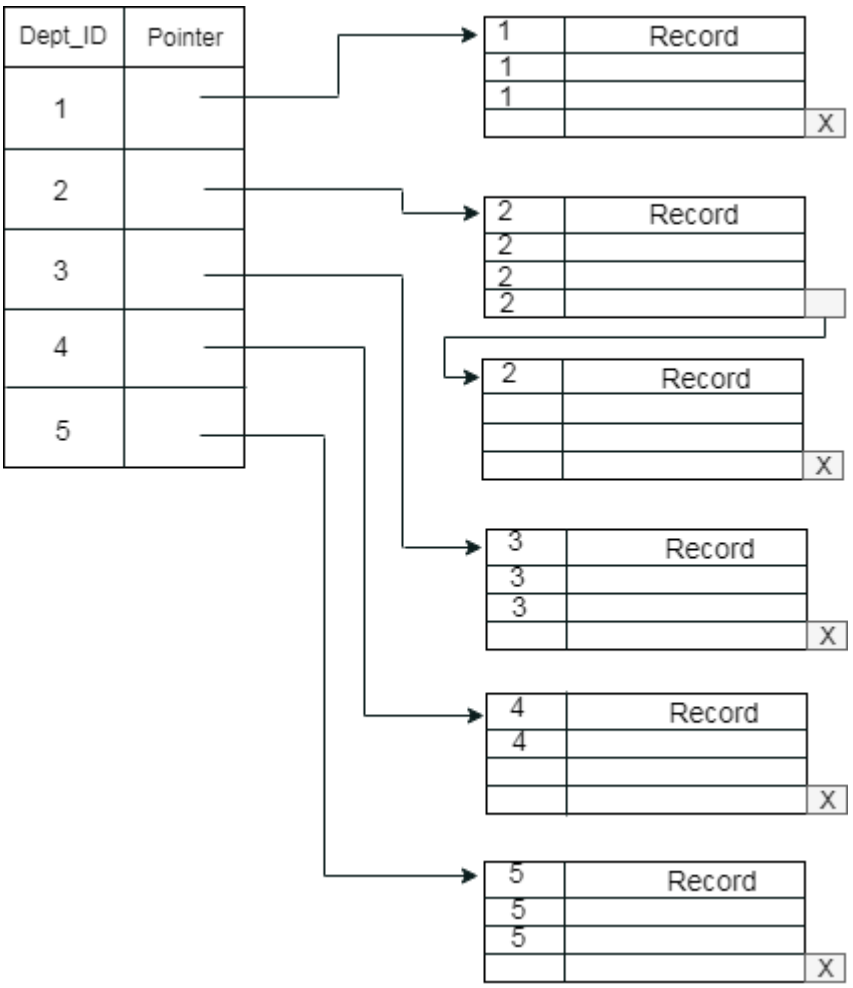
Clustering Index

- 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.



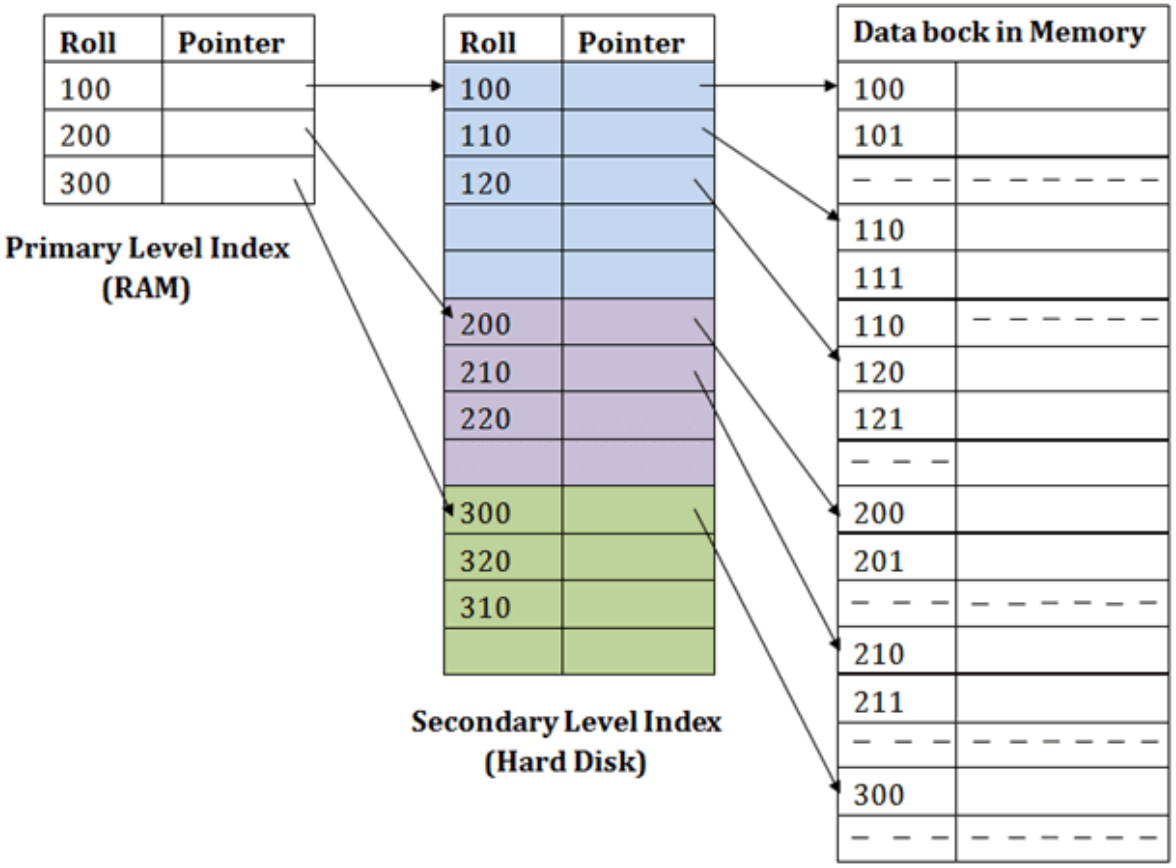
The previous schema is little confusing because one disk block is shared by records which belong to the different cluster. If we use separate disk block for separate clusters, then it is called better technique.



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).



For example:

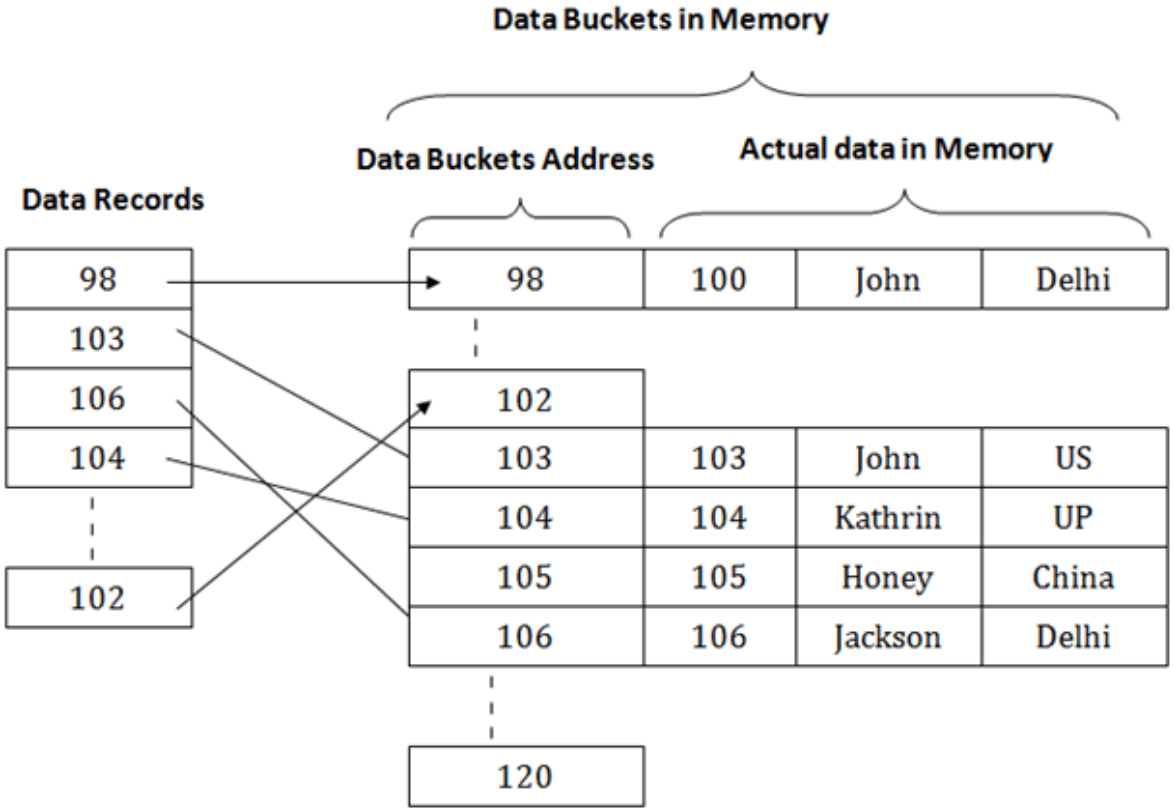
- If you want to find the record of roll 111 in the diagram, then it will search the highest entry which is smaller than or equal to 111 in the first level index. It will get 100 at this level.
- Then in the second index level, again it does max (111) <= 111 and gets 110. Now using the address 110, it 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.

Hashing

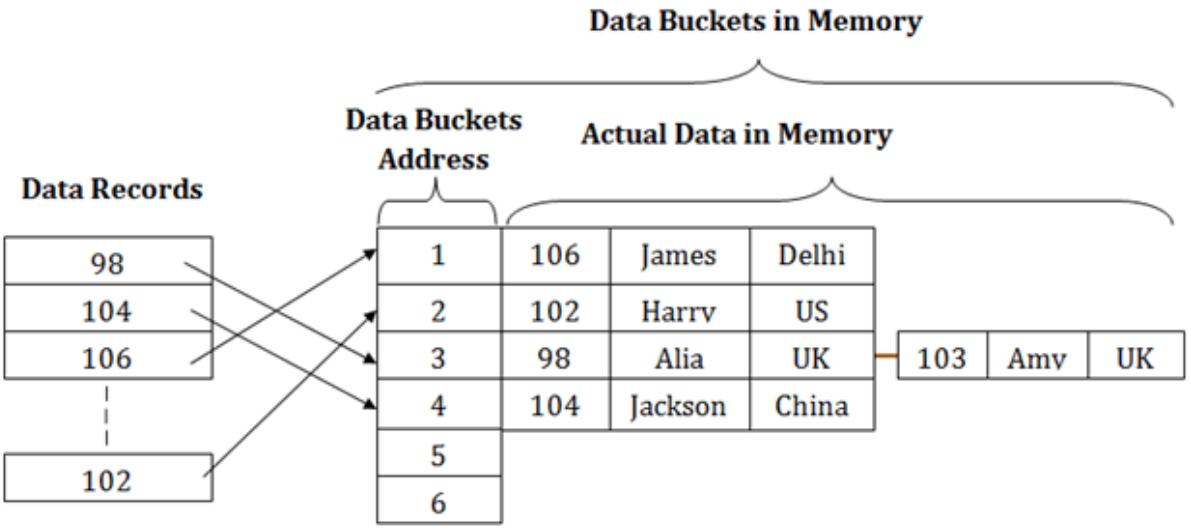
In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.

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.

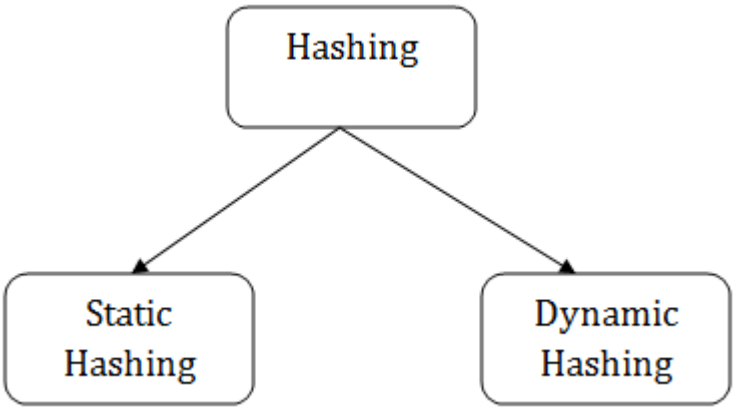
In this, a hash function can choose any of the column value to generate the address. Most of the time, the hash function uses the primary key to generate the address of the data block. A hash function is a simple mathematical function to any complex mathematical function. We can even consider the primary 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.



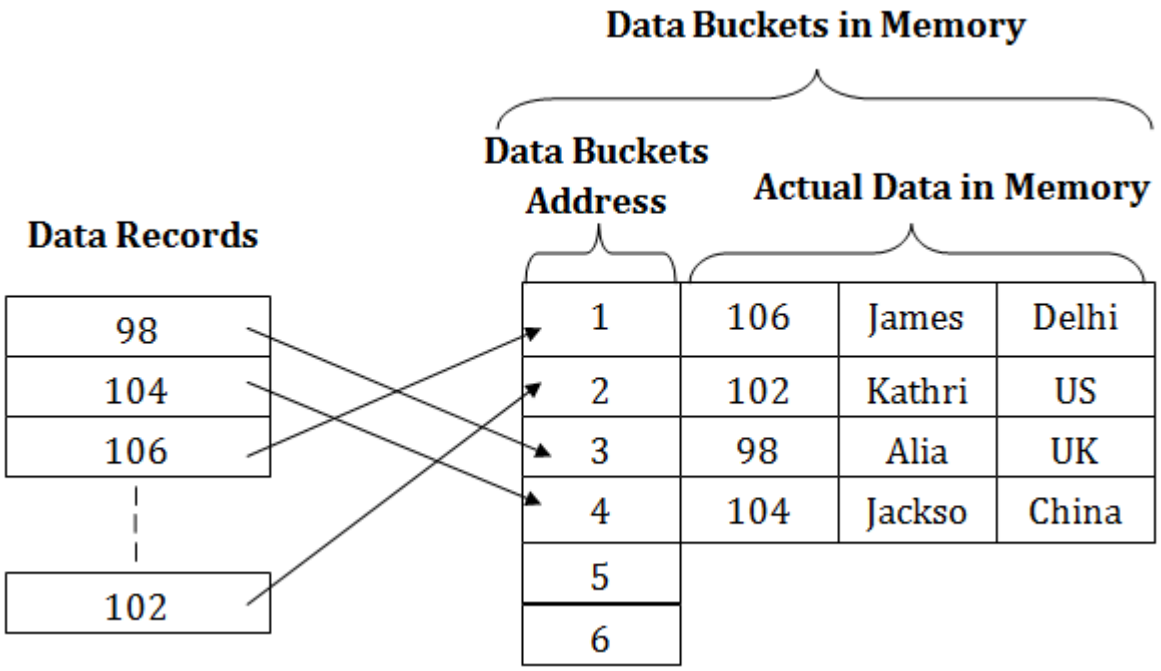
Types of Hashing:



Static Hashing

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.

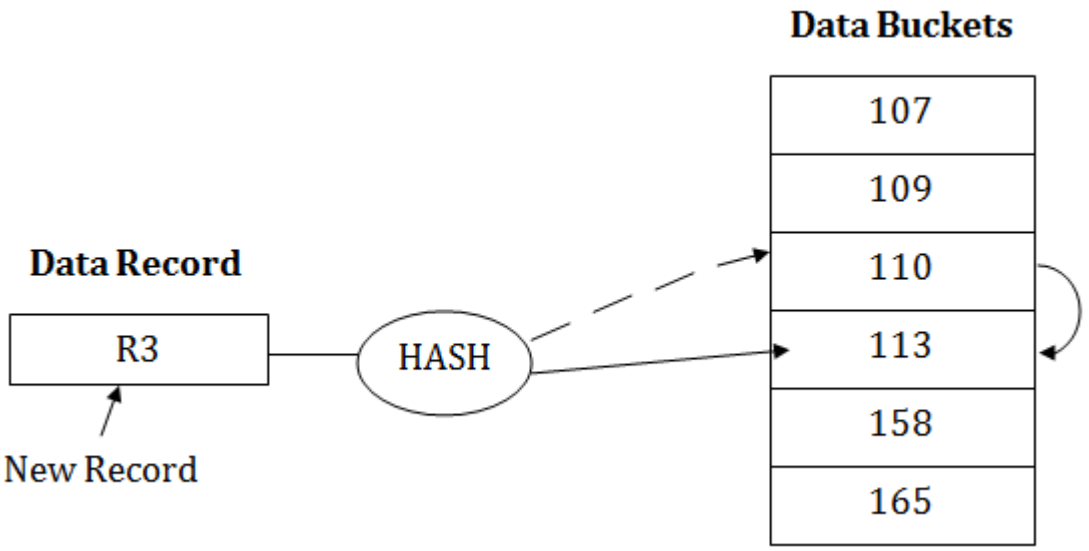
If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.

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**.

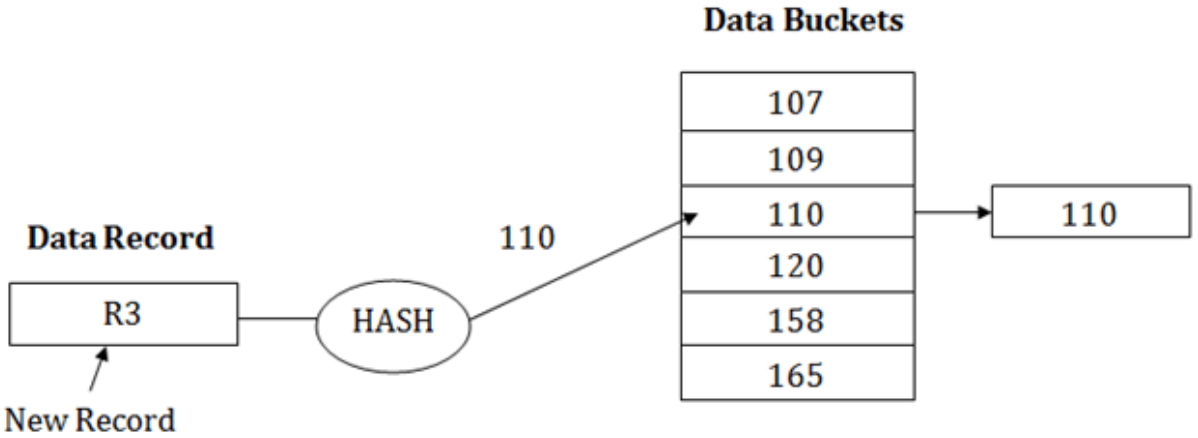
For example: suppose R3 is a new address which needs to be inserted, the hash function generates address as 112 for R3. But the generated address is already full. So the system searches next available data bucket, 113 and assigns R3 to it.



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.



4.3 Query Processing

Query processing is the step-by-step procedure that a database system follows to execute a query efficiently. It involves converting a high-level SQL query into a low-level sequence of operations that the database system understands.

Steps Involved in Query Processing

1. **Parsing and Translation:**
    - The query written in SQL is checked for syntax and semantics.
    - The query is converted into a relational algebra expression for further processing.
2. **Query Optimization:**
    - The system evaluates multiple ways of executing the query and selects the most efficient one based on cost.
3. **Query Execution:**
    - The optimized query plan is executed by the database engine to retrieve the required data.

Measures of Query Cost

The cost of executing a query is measured based on the resources it uses. Key factors include:

1. **Disk I/O (Input/Output):**
    - Reading and writing data to/from the disk.
    - This is the most critical factor as it impacts performance the most.
2. **CPU Time:**
    - Time taken for computations during query execution.
3. **Memory Usage:**
    - The amount of RAM required during query processing.
4. **Communication Costs:**
    - Relevant in distributed databases, where data must be transmitted between different nodes.

Algorithms for SELECT and PROJECT Operations

1. **SELECT Operation (σ):**
   Retrieves rows that satisfy a certain condition.
   Algorithms:
   - **Linear Search:** Scans each row and checks if it satisfies the condition. (Simple but slow)
   - **Binary Search:** Used when the data is sorted. It divides the data and searches efficiently.
   - **Index-Based Search:** Uses an index to directly find rows matching the condition.
2. **PROJECT Operation (π):**
   Retrieves specific columns from a table, removing duplicates.
   Algorithms:
   - **Single Scan:** Reads all rows, extracts required columns, and removes duplicates.
   - **Sorting with Deduplication:** Sorts rows by the selected columns and eliminates duplicates during sorting.

4.4 Query Optimization

Query optimization ensures that a query is executed in the most efficient way, minimizing resource usage and response time.

Overview of Query Optimization

- A high-level query can be executed in multiple ways (different execution plans).
- The **query optimizer** selects the most efficient execution plan by estimating the cost of each option.

Transformation of Relational Expressions

- The relational algebra expression derived from the SQL query can be rewritten into equivalent forms.
- **Equivalent forms:** They produce the same output but may differ in execution costs.
  Example:
  - (A JOIN B) JOIN C
  - A JOIN (B JOIN C)
    Choosing the best order of operations can significantly reduce the cost.

Estimating Statistics

The optimizer uses metadata and statistics about the database to estimate:

1. **Number of rows (cardinality):**
   - How many rows a table or result set is expected to have.
2. **Size of the data:**
   - How much memory or storage the query will use.
3. **Distribution of values:**
   - Helps in estimating the effectiveness of conditions like WHERE age > 30.

Choice of Evaluation Plan

- An **evaluation plan** is the step-by-step procedure for executing the query.
- The optimizer uses:
  1. **Access Paths:**
     Methods like table scans or index lookups to retrieve data.
  2. **Join Algorithms:**
     - **Nested Loop Join:** Simple but slow for large tables.
     - **Merge Join:** Faster for sorted data.
     - **Hash Join:** Efficient for large datasets.
- The final choice depends on which plan has the lowest estimated cost based on disk I/O, CPU time, and memory usage.