DATA STORAGE & INDEXING

Define the terms Primary Index and Secondary Index. Differentiate between them on basis of the Evaluation Criteria for indices

Primary Index:

- A Primary Index is an index that is created on the primary key of a table. It is a type of index that is used to uniquely identify each record in a database table.
- The primary index is typically a clustered index, meaning that the physical order of the rows in the table is the same as the order of the index.
- It provides fast access to data based on the primary key values.
- 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.

Secondary Index:

- A Secondary Index is an index that is created on a non-primary key column of a table to improve the performance of queries that do not use the primary key.
- Unlike the primary index, a secondary index does not dictate the physical order of the rows in the table.
- It allows for fast access to data based on the values of columns other than the primary key.

Differentiation based on Evaluation Criteria:

1. **Access Time**:
    - Primary Index: Provides fast access to records based on the primary key values.

- Secondary Index: Offers fast access to records based on non-primary key values, which may require additional steps compared to the primary index.

2. **Insertion Time**:
    - Primary Index: Insertions in a primary index may involve more overhead due to maintaining the physical order of the rows.
    - Secondary Index: Insertions in a secondary index may have less overhead compared to a primary index, especially for non-primary key values.
3. **Deletion Time**:
    - Primary Index: Deletions in a primary index may involve more complexity due to maintaining the clustered order of the rows.
    - Secondary Index: Deletions in a secondary index may be less complex compared to a primary index, especially for non-primary key values.
4. **Space Overhead**:
    - Primary Index: The primary index may have a higher space overhead due to maintaining the physical order of the rows.
    - Secondary Index: The secondary index may have a lower space overhead compared to a primary index, especially for non-primary key values.

When does a collision occur in hashing? Illustrate various collision resolution techniques.

A collision occurs in hashing when two different keys map to the same index in the hash table. This can happen due to the inherent nature of hash functions, which are designed to map a large key space to a smaller index space.
Collisions can occur due to various reasons such as:

- **Insufficient buckets**: If the number of buckets is too low, it can lead to collisions.
- **Skew in distribution of records**: If the distribution of keys is not uniform, it can lead to collisions.
- **Chosen hash function**: A poorly designed hash function can produce non-uniform distributions, leading to collisions.
-

To resolve collisions, various techniques are employed:

1. **Chaining**:
   In this technique, when a collision occurs, the colliding keys are stored in a linked list at the index where the collision occurred. This is known as a "bucket" or "chain". Each bucket can hold a fixed number of keys, and if a bucket is full, the system provides an overflow bucket, and the process continues until all keys are stored.

2. **Open Addressing**:
   In this technique, when a collision occurs, the system searches for the next available slot in the table. This is known as "linear probing". The search continues until an empty slot is found, and the key is stored there.

3. **Quadratic Probing**:
   This is a variation of linear probing where the search is done in a quadratic pattern. The search continues until an empty slot is found, and the key is stored there.

4. **Double Hashing**:
   In this technique, two hash functions are used. The first hash function maps the key to an index, and the second hash function maps the key to an offset within the bucket. This technique is used to reduce collisions.

5. **Cuckoo Hashing**:

In this technique, two arrays are used. Each key is stored in one of the arrays, and when a collision occurs, the key is moved to the other array. This technique is used to reduce collisions.

6. **Robin Hood Hashing**:
   In this technique, when a collision occurs, the key is moved to the next available slot in the table. This technique is used to reduce collisions.

7. **Linear Probing with a Twist**:
   In this technique, when a collision occurs, the system searches for the next available slot in the table, but with a twist. The search is done in a circular pattern, and the key is stored at the next available slot.

8. **Binary Search**:
   In this technique, when a collision occurs, the system performs a binary search to find the next available slot in the table. This technique is used to reduce collisions.

9. **Hashing with a Bloom Filter**:
   In this technique, a Bloom filter is used to quickly determine if a key is present in the hash table. If the key is not present, the system can quickly determine that it is not present, reducing the number of collisions.

10. **Cuckoo Hashing with a Twist**:
    In this technique, two arrays are used, and when a collision occurs, the key is moved to the other array. But with a twist, the key is moved to the next available slot in the other array. This technique is used to reduce collisions.

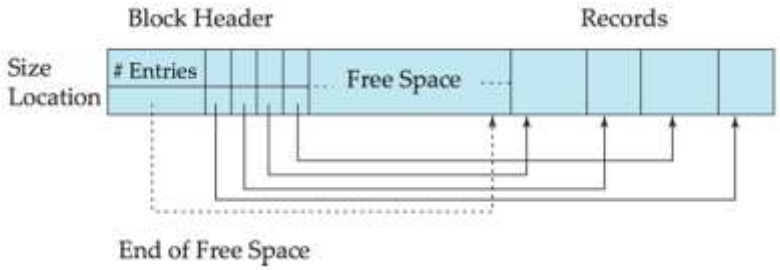
Explain how Variable Length records are Represented in file.

Variable-length records are represented in files using the following approach:

1. **Attribute Representation**:
    - Variable-length attributes are represented using a fixed-size pair of (offset, length) values.
    - The offset indicates the starting position of the variable-length data within the record.
    - The length indicates the number of bytes occupied by the variable-length data.
    - This allows the database system to locate and access the variable-length data within the record.

2. **Null Bitmap**:
    - A null bitmap is used to indicate which attributes in the record have a null value.
    - Each bit in the null bitmap corresponds to an attribute in the record.
    - If a bit is set to 1, it indicates that the corresponding attribute has a null value.
    - This allows the database system to efficiently handle null values without wasting space for storing the actual data.

3. **Slotted Page Structure**:
    - The slotted page structure is commonly used to organize variable-length records within a block or page.
    - The page header contains information such as the number of record entries, the end of free space in the block, and the location and size of each record.
    - The actual records are allocated contiguously in the block, starting from the end of the block.
    - The free space in the block is contiguous, between the final entry in the header array and the first record



The slotted-page structure is commonly used for organizing records within a block.

Slotted page header contains:

- number of record entries
- end of free space in the block
- location and size of each record

4. **Record Insertion and Deletion**:
   - When a new record is inserted, space is allocated for it at the end of the free space, and an entry containing its size and location is added to the header.
   - When a record is deleted, the space it occupied is freed, and its entry in the header is marked as deleted. The records before the deleted record are then moved to occupy the freed space, and the free space is again located between the final entry in the header and the first record.

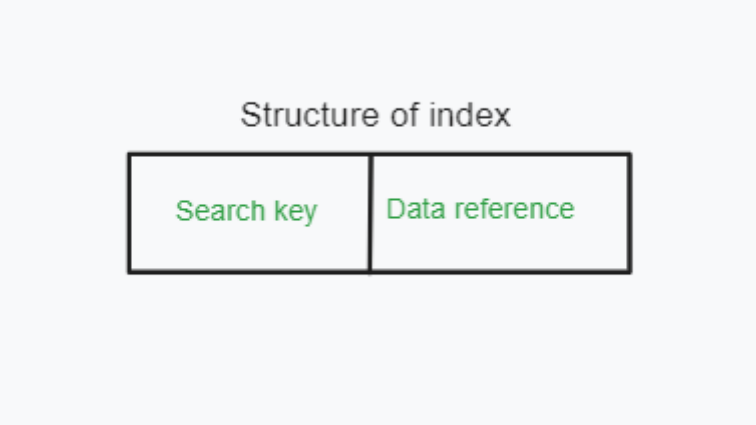
List and explain in brief the ways of Organization of Records in Files.

The ways of organizing records in files are crucial for efficient data storage and retrieval in a database system. Here are some common methods of organizing records in files:

1. **Heap Organization**:
    - In heap organization, records can be placed anywhere in the file where there is available space.
    - There is no specific ordering of records, and new records are inserted wherever space is available.
    - This method is simple but can lead to fragmentation and slower retrieval times.
2. **Sequential Organization**:
    - Sequential organization stores records in sequential order based on the value of a search key.
    - Records are chained together using pointers to allow fast retrieval in search-key order.
    - Deletion and insertion operations may require reorganizing the file to maintain sequential order.
3. **Hashing Organization**:
    - Hashing involves computing a hash function on an attribute of each record to determine the block in which the record should be placed.
    - The hash function result dictates the location of the record in the file.
    - This method is efficient for direct access to records based on a specific attribute.
4. **Multitable Clustering File Organization**:
    - Multitable clustering allows storing multiple relations in one file, enabling efficient retrieval for queries involving related records.
    - It facilitates reading records satisfying join conditions with a single block read.
    - This method is beneficial for queries involving multiple related tables.

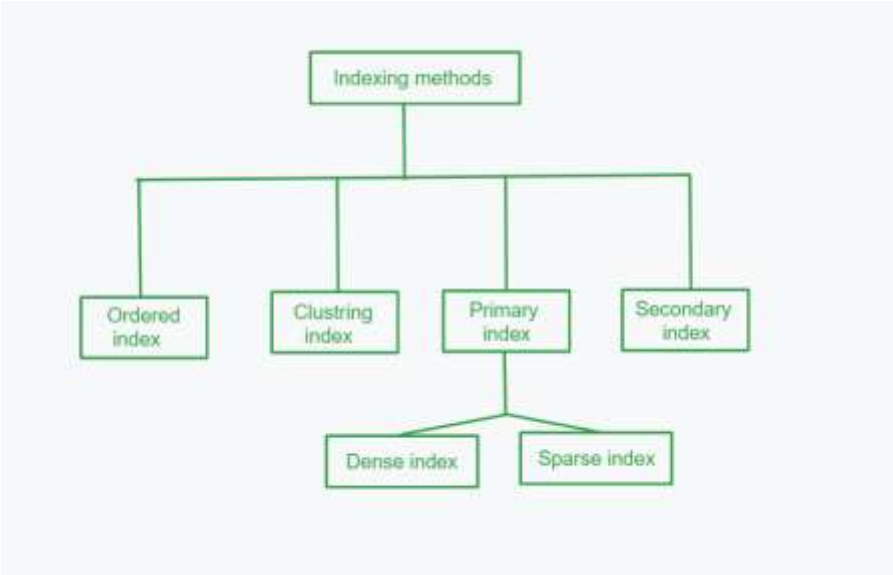
Define the terms Dense Index and Sparse Index. Differentiate between them on basis of the Evaluation Criteria for indice.

Indexing is a technique in **DBMS** that is used to optimize the performance of a database by reducing the number of disk access required. An index is a type of data structure. With the help of an index, we can locate and access data in database tables faster. The dense index and Sparse index are two different approaches to organizing and accessing data in the data structure. These are commonly used in databases and information retrieval systems.



Different Types of Indexing Methods

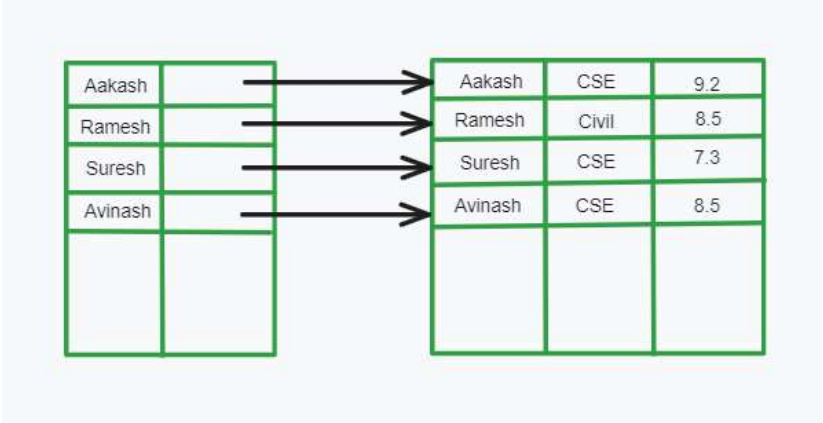
Indexing methods in a database management system (DBMS) can be classified as dense or sparse indexing methods, depending on the number of index entries in the database. Let's take a look at the differences between the two types of indexing methods



Dense indexing and Sparse indexing are types of primary indexing. Now let's take an overview of these terms:

Dense Index

It contains an index record for every search key value in the file. This will result in making searching faster. The total number of records in the index table and main table are the same. It will result in the requirement for more space to store the index of records itself.



Advantages

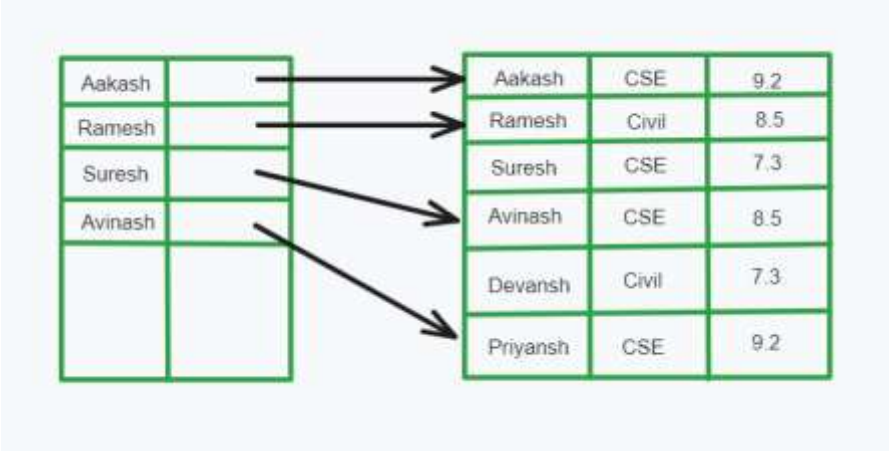
1. Gives quick access to records, particularly for Small datasets.
2. Effective for range searches since each key value has an entry.

Disadvantages

1. Can be memory-intensive and may require a significant amount of storage space.
2. Insertions and deletions result in a higher maintenance overhead because the index must be updated more frequently.

Sparse Index

Sparse index contains an index entry only for some records. In the place of pointing to all the records in the main table index points records in a specific gap. This indexing helps you to overcome the issues of dense indexing in DBMS.



Advantages

1. Uses less storage space than thick indexes, particularly for large datasets.
2. Lessens the effect that insertions and deletions have from index maintenance operations.

Disadvantages

1. Since there may not be an index entry for every key value, access may involve additional steps.
2. might not be as effective as dense indexes for range queries.

Difference Between Dense Index and Sparse Index

Dense index	Sparse index
The index size is larger in dense index.	In sparse index, the index size is smaller.
Time to locate data in index table is less.	Time to locate data in index table is more.
There is more overhead for insertions and deletions in dense index.	Sparse indexing have less overhead for insertions and deletions.
Records in dense index need not to be clustered.	In case of sparse index, records need to be clustered.
Computing time in RAM (Random access memory) is less with dense index.	In sparse index, computing time in RAM is more.
Data pointers in dense index point to each record in the data file.	In sparse index, data pointers point to fewer records in data file.
Search performance is generally faster in dense index.	In sparse index, search performance may require additional steps, which

Dense index	Sparse index
	will result in slowing down the process.

Illustrate Multiple Key Access with appropriate example.

Multiple-Key Access Example:
Suppose we have a table named **instructor** with attributes **ID**, **dept_name**, and **salary**. To illustrate multiple-key access, let's consider the following query:

```sql
SELECT ID
FROM instructor
WHERE dept_name = 'Finance' AND salary = 80000;
```

Possible Strategies for Processing Query:

1. **Using Index on dept_name:**
   - First, use the index on **dept_name** to find instructors with the department name 'Finance'.
   - Then, examine each such record to check if the salary is 80000.
2. **Using Index on salary:**
   - First, use the index on **salary** to find instructors with a salary greater than 80000.
   - Then, examine each such record to see if the department name is 'Finance'.
3. **Using Indices on Both dept_name and salary:**
   - Utilize the index on **dept_name** to find pointers to all records pertaining to the 'Finance' department.
   - Also, use the index on **salary** to find pointers to all records with a salary of 80000.
   - Fetch only the records that satisfy both conditions efficiently.

Indices on Multiple Keys:

- Composite search keys contain more than one attribute, such as (dept_name, salary).
- Lexicographic ordering is used, where (a1, a2) < (b1, b2) if either a1 < b1 or if a1 = b1 and a2 < b2.

Indices on Multiple Attributes:
Suppose we have an index on the combined search key (dept_name, salary).

- With the **WHERE** clause **WHERE dept_name = 'Finance' AND salary = 80000**, the index on (dept_name, salary) can efficiently fetch records satisfying both conditions.
- It can also handle queries like **WHERE dept_name = 'Finance' AND salary < 80000** efficiently.

- An ordered index on (dept_name, salary) can efficiently handle queries on only one attribute, like **SELECT ID FROM instructor WHERE dept_name = 'Finance'**.
- However, it cannot efficiently handle queries like **WHERE dept_name < 'Finance' AND balance = 80000**, which may fetch many records satisfying the first condition but not the second.

Elaborate Deficiencies of Static Hashing and probable solutions to those deficiencies.

Deficiencies of Static Hashing:

1. **Fixed Bucket Addresses**:
    - In static hashing, a hash function maps search-key values to a fixed set of bucket addresses.
    - As databases grow or shrink over time, the fixed bucket addresses may become inadequate to efficiently distribute data.
2. **Performance Degradation**:
    - Choosing a hash function based on the current file size can lead to performance degradation as the database grows.
    - The hash function may not be optimized for the increased data volume, resulting in inefficient data distribution.
3. **Limited Scalability**:
    - Static hashing lacks scalability as it does not dynamically adjust to changing database sizes.
    - It may lead to uneven data distribution and increased bucket overflows as the database size fluctuates.

Probable Solutions:

1. **Dynamic Hashing**:
    - Implement dynamic hashing techniques that can adapt to changes in database size.
    - Dynamic hashing allows for the dynamic allocation of buckets based on the current data volume, ensuring efficient data distribution.
2. **Extendible Hashing**:
    - Use extendible hashing, a dynamic hashing technique that dynamically adjusts the directory structure and bucket sizes as the database grows.
    - This approach allows for efficient data distribution and minimizes the impact of database size changes on performance.
3. **Linear Hashing**:

- Consider linear hashing, another dynamic hashing technique that dynamically expands the hash table as needed.
- Linear hashing provides a flexible and scalable approach to handle database growth without sacrificing performance.

4. **Overflow Handling**:
    - Implement effective strategies for handling bucket overflows, such as overflow chaining or open hashing.
    - Proper overflow management can mitigate the impact of static hashing deficiencies related to insufficient buckets and skewed data distribution.

List and explain the Index Evaluation Metrics.

1. **Access Time**:
    - The time it takes to find a particular data item or set of items using the index.
    - This is a crucial metric, as the primary purpose of an index is to provide fast access to data.
    - Indices that offer faster access times are generally preferred.
2. **Insertion Time**:
    - The time it takes to insert a new data item into the index structure.
    - This includes the time to find the correct place to insert the new item and update the index structure accordingly.
    - Indices that can handle insertions efficiently are desirable, especially in dynamic databases.
3. **Deletion Time**:
    - The time it takes to delete a data item from the index structure.
    - This includes the time to find the item to be deleted and update the index structure.
    - Efficient deletion is important for maintaining the index's integrity and performance.
4. **Space Overhead**:
    - The additional storage space occupied by the index structure itself.
    - Indices that have a lower space overhead are generally preferred, as they consume less storage resources.
    - However, this metric needs to be balanced with the access time and other performance considerations.