a) Define unclustered index? An unclustered index is a type of database index that maintains a separate structure from the actual data. Instead of arranging the rows in the table itself, it creates a distinct index structure that contains pointers (or references) to the locations of data in the table. This means the physical order of the rows in the table isn't affected by the index.	**1 M**
b) What is Indexing and Hashing? **Indexing**: • Indexing in DBMS is a way to improve the speed of data retrieval operations on a table. • It creates a data structure—usually a tree or a hash table—that stores pointers to the actual data on disk. **Hashing**: • Hashing is another technique to locate data quickly in a database. • It uses a hash function to convert a search key into an index in a hash table.	**1 M**
c) What is an index? Give an example. An index in DBMS is a data structure that helps speed up data retrieval operations on a database table. It's like a shortcut that allows the database to quickly locate the data without scanning the entire table. **Example**: Imagine a library catalog system. If you're looking for a book by its title, instead of going through each book on the shelf, you refer to the catalog (the index) that tells you exactly where to find that book.	**1 M**
d) Discuss about primary indexes? Primary indexes are one of the key indexing mechanisms in DBMS, often used to enhance the performance of query processing. **Primary Index**: • A primary index is created on the primary key of a table. • The primary key is a unique identifier for records in a table, meaning each record has a unique value for the primary key. • This index is usually sparse, meaning it does not store every single record's pointer but just the key pointers.	**1 M**

e) What is meant by secondary index?

A secondary index in DBMS is an additional means to retrieve data, separate from the primary index, aimed at improving search performance.

Secondary Index:

- It's created on non-primary key columns, allowing for quicker access to data based on those columns.
- Unlike a primary index, a secondary index can be dense, meaning it contains pointers to every record in the table.

1 M

a) differences between tree based and Hash based indexes

3 M

b) What are the advantages and disadvantages of B+ trees?

3 M

Advantages of B+ Tree:

1. **Balanced Structure**: B+ Trees maintain a balanced structure, ensuring that all leaf nodes are at the same level. This guarantees consistent and predictable performance.
2. **Efficient Searches**: Because it's a balanced tree, the search operations are efficient, typically logarithmic in complexity.
3. **Range Queries**: B+ Trees are particularly good for range queries. Since all the leaf nodes are linked, traversing a range of values is efficient.
4. **Disk-Friendly**: B+ Trees minimize disk I/O operations as they keep internal nodes smaller, ensuring more nodes can be kept in memory.
5. **Order Preservation**: Since B+ Trees store data in a sorted manner, it's easy to perform ordered operations like finding the smallest or largest value in a range.

Disadvantages of B+ Tree:

1. **Complexity**: B+ Trees are more complex to implement and maintain compared to simpler data structures like hash tables.
2. **Overhead**: There is an overhead associated with maintaining the balanced structure, particularly during insertions and deletions.
3. **Space Usage**: Although efficient, B+ Trees can consume more space due to internal pointers and the linked structure of leaf nodes.
4. **Latency on Sequential Access**: For purely sequential access, B+ Trees can be less efficient than other structures because of the additional traversal required.

c) What is the difference between Indexing and Hashing ? **3 M**

Indexing and **Hashing** both aim to speed up data retrieval in a database, but they achieve this in distinct ways.

Indexing:

- It creates a data structure (like a tree or a hash table) that allows quick search, insertion, and deletion operations.
- Indexes can be on primary keys or other attributes.
- They are sorted, which makes range queries efficient.
- Example: B+ Trees are a common indexing structure, particularly good for range searches and ordered queries.

Hashing:

- Uses a hash function to map search keys to positions in a hash table.
- Ideal for exact match queries.
- It’s an unsorted approach, so range queries are not efficient.
- Example: Hash tables offer constant time complexity for search operations, making them fast for lookups by key.

d) What is the main difference between ISAM and B+ tree indexes? **3 M**

ISAM (Indexed Sequential Access Method):

- **Structure**: ISAM uses a static tree structure.
- **Updates**: Modifications require reorganizing the index, which can be cumbersome.
- **Efficiency**: Good for read-heavy environments since the index is not dynamically adjusted.
- **Storage**: Static, which can lead to inefficient space usage over time.

B+ Tree:

- **Structure**: B+ Trees maintain a dynamic, balanced tree structure.
- **Updates**: Automatically adjusts, splits, or merges nodes to maintain balance.
- **Efficiency**: Suitable for both read and write operations, thanks to dynamic balancing.
- **Storage**: More efficient space usage due to dynamic nature.

e) What are the advantages of using tree structured indexes? **3 M**

Tree-structured indexes, like B+ Trees, come with a host of advantages:

1. **Balanced Structure**: They maintain a balanced form, ensuring consistent performance for

searches, insertions, and deletions.

2. **Efficient Query Performance**: Their logarithmic time complexity makes them efficient, even for large datasets.
3. **Ordered Data**: Since they store data in a sorted order, operations like range queries and ordered traversals are very efficient.
4. **Low I/O Operations**: They reduce disk I/O operations because internal nodes fit into memory, making access faster.
5. **Dynamic Adjustments**: They automatically adjust to maintain balance, which is crucial for maintaining performance as data changes.
6. **Hierarchical Structure**: Allows efficient multi-level indexing, optimizing space and search times.

Tree-structured indexes, especially B+ Trees, offer a balanced, efficient, and dynamic way to handle large amounts of data.

a) Explain the Insertion and deletion Operations in B+ trees with example. **5 M**

Refer material

b) Explain Deletion and insertion operations in ISAM with example. **5 M**

Refer material

c) What are the Pros and Cons of ISAM? **5 M**

Pros of ISAM (Indexed Sequential Access Method):

1. **Fast Read Access**: ISAM is highly efficient for read-heavy operations. It provides quick access to records through its indexed structure.
2. **Simplicity**: The structure is relatively simple and easy to implement, making it straightforward to use.
3. **Predictable Performance**: Because of its static nature, ISAM offers predictable and stable performance for queries.
4. **Efficient for Static Data**: Works well for databases where data doesn't change frequently, providing consistent performance without the need for frequent reorganizations.

Cons of ISAM:

1. **Static Structure**: ISAM's structure is static, meaning once it's created, it doesn't adjust automatically to changes in data. This can lead to inefficiencies over time as the database grows and changes.
2. **Maintenance Overhead**: Requires periodic reorganization and maintenance to handle insertions and deletions efficiently, which can be cumbersome and time-consuming.
3. **Space Utilization**: Can lead to wasted space due to its pre-allocated index structure, especially

if the data distribution changes over time.

4. **Inefficiency with Frequent Updates**: Not ideal for environments with frequent updates, as the static structure can become unbalanced and inefficient, requiring manual intervention to reorganize the data.

ISAM is best suited for environments with static or read-heavy workloads where its predictability and simplicity can shine

d) How the concurrency control is done in B+ trees? Explain **5 M**

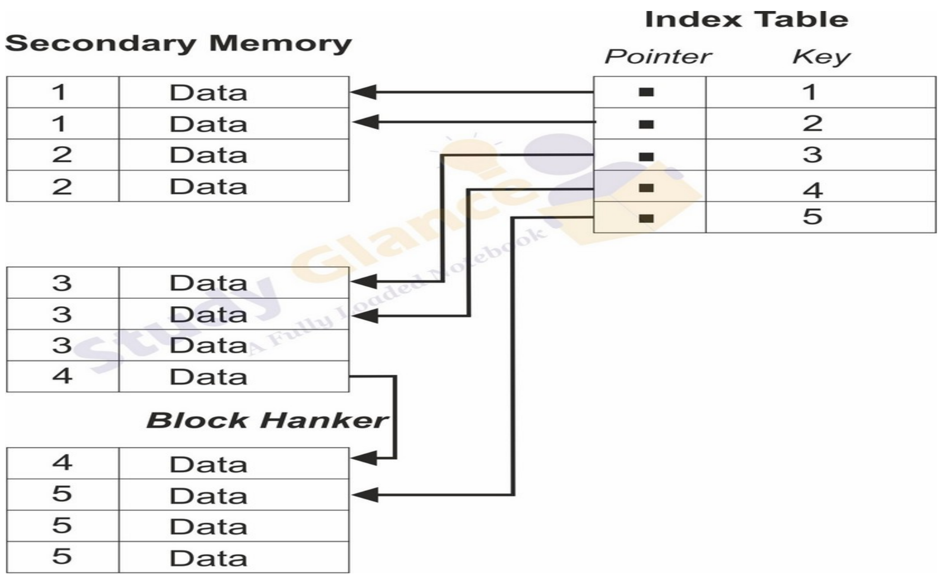
Concurrency control in B+ Trees ensures that multiple transactions can access and modify the tree without interfering with each other, preserving data consistency and integrity. Here's how it's typically managed:

1. **Locking Protocols**:
   - **Shared Locks (S-Locks)**: These allow multiple transactions to read the same data simultaneously.
   - **Exclusive Locks (X-Locks)**: These are used when a transaction needs to modify data, ensuring no other transaction can read or write to that data until the lock is released.
2. **Lock Coupling (Crabbing)**:
   - This method involves acquiring locks gradually as you traverse the B+ Tree. For example, when searching for a node to insert a key:
     - Start at the root, lock it.
     - Move to the next child, lock it, then release the lock on the parent.
     - Continue this pattern down the tree.
   - Ensures that the path from the root to the leaf is protected, reducing the duration any single lock is held.
3. **Optimistic Concurrency Control**:
   - Instead of locking, this approach assumes conflicts are rare and validates the transaction at commit time.
   - Transactions operate on a local copy of the data, then check if the data has changed before committing. If there are no conflicts, the changes are applied; otherwise, the transaction is rolled back and restarted.
4. **Latching**:
   - Fine-grained, short-duration locks used to protect specific data structures or operations within the B+ Tree.
   - Latches are more lightweight than locks, minimizing contention and improving performance for concurrent operations.

Concurrency control in B+ Trees balances efficiency and data integrity, making sure that multiple transactions can coexist without stepping on each other's toes

e) Explain about clustered index organization? **5 M**

A clustered index determines the physical order of data in a table. In other words, the order in which the rows are stored on disk is the same as the order of the index key values. There can be only one clustered index on a table, but the table can have multiple non-clustered (or secondary) indexes.



Characteristics of a Clustered Index:

1. It dictates the physical storage order of the data in the table.
2. There can **only be one clustered index** per table.
3. It can be created on columns with **non-unique values**, but if it's on a non-unique column, the DBMS will usually add a unique identifier to make each entry unique.
4. Lookup based on the clustered index is **fast** because the desired data is directly located without the need for additional lookups (unlike non-clustered indexes which require a second lookup to fetch the data).

Clustered Index 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`.

The table would then look like this:

```
| 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.

Benefits of a Clustered Index:

- **Fast Data Retrieval:** Because the data is stored sequentially in the order of the index key, range queries or ordered queries can be very efficient.
- **Data Pages:** With the data being stored sequentially, the number of data pages that need to be read from the disk is minimized.
- **No Additional Lookups:** Once the key is located using the index, there's no need for additional lookups to fetch the data, as it is stored right there.

a) State and explain various file organization methods. **10 M**

Refer material

b) What are indexed data structures? Explain any one of them. **10 M**

Explain Hash indexing technique (OR) B+ tree

c) Explain indexes and performance tuning. **10 M**

Indexing

1. Objective: To create a data structure that improves the speed of data retrieval operations.

2. Methodologies: Includes clustered, non-clustered, primary, secondary, composite, bitmap, and hash indexes, among others.

3. Implications:

- Clustered indexes are excellent for range-based queries but slow down insert/update operations.
- Non-clustered indexes improve data retrieval speed but can take up additional storage.
- Bitmap indexes are useful for low-cardinality columns.

4. Real-world Examples: Search engines, e-commerce websites, any application that requires fast data retrieval.

Performance Tuning

1. Objective: To optimize the resources used by the database for efficient transaction processing.

2. Methodologies: Query optimization, index tuning, denormalization, database sharding, caching, partitioning, etc.

3. Implications:

- Query optimization can dramatically reduce the resources needed for query processing.
- Proper indexing can mitigate the need for full-table scans.
- Denormalization and caching can improve read operations but may compromise data integrity or consistency.

4. Real-world Examples: Financial trading systems, real-time analytics, high-performance computing.

Points of Comparison on File organization, indexing, and performance tuning

1. Granularity:

- File organization is about how data is stored at the file level.
- Indexing is about improving data access at the table or even column level.
- Performance tuning is a broad set of activities that can encompass both file organization and indexing among many other techniques.

2. Resource Usage:

- File organization techniques aim to use disk space efficiently.
- Indexing aims to use both disk space and memory for fast data retrieval.
- Performance tuning aims to optimize all system resources including CPU, memory, disk, and network bandwidth.

3. Query Efficiency:

- File organization generally impacts how efficiently data can be read or written to disk.
- Indexing significantly impacts how efficiently queries can retrieve data.
- Performance tuning seeks to optimize both read and write operations through a variety of methods.

4. Complexity:

- File organization is relatively straightforward.
- Indexing can become complex depending on the types of indexes and the nature of the queries.
- Performance tuning is usually the most complex as it involves a holistic understanding of hardware, software, data, and queries.