Indexing in Databases

Indexing is a mechanism to optimize the retrieval of records from a database. Instead of scanning the entire table, an index allows the database system to quickly locate and access the desired data.

- **Basic Concepts**
  - **Index:** A data structure that helps in efficient data retrieval.
  - **Search Key:** One or more attributes used to search records in the file. Example: In a student table, the "Roll Number" could be the search key.
  - **Index Entries:** Records in the index file consisting of a search key and a pointer to the actual data.
- **Types of Indices**
  1. **Ordered Indices**: Index entries are stored in sorted order of the search key.
  2. **Hash Indices**: Search keys are mapped to "buckets" using a hash function.
- **Types of Ordered Indices**
  - **Dense Index**
    - **Definition:** Contains an index record for every search key value.
    - **Example:**
      - Consider a table Instructor:
        `ID | Name | Dept_Name | Salary 101 | John | CS | 80000 102 | Alice | EE | 85000 103 | Bob | ME | 75000``
      - A dense index on the `ID` attribute would look like:
        `Search Key (ID) | Pointer to Record 101 | Pointer to John 102 | Pointer to Alice 103 | Pointer to Bob`
  - **Sparse Index**
    - **Definition:** Contains index entries for some search key values.
    - **Example:**
      - Using the same table, if the data is sorted by `ID` and stored in blocks:
        - Block 1: Records with IDs 101–102.
        - Block 2: Records with ID 103.
      - A sparse index might look like:

```
        101             | Pointer to Block 1
        103             | Pointer to Block 2
```

B+-Tree Indexing

- A B+-Tree is a self-balancing tree data structure, commonly used for database indexing. It maintains sorted data and allows for efficient insertion, deletion, and search operations.
- **Properties of B+-Trees**
  1. All paths from the root to leaf nodes are of equal length.
  2. Non-leaf nodes store keys and pointers to child nodes.
  3. Leaf nodes store search key values and pointers to records.
- **Structure of B+-Tree Node**
  - Non-leaf Node:
    - Keys: K1,K2,...,KnK_1, K_2, ..., K_nK1,K2,...,Kn
    - Pointers: P1,P2,...,Pn+1P*1, P_2, ..., P*{n+1}P1,P2,...,Pn+1
  - Leaf Node:
    - Keys: K1,K2,...,KnK_1, K_2, ..., K_nK1,K2,...,Kn
    - Pointers to records: P1,P2,...,PnP_1, P_2, ..., P_nP1,P2,...,Pn
  - Example of B+- tree:
    - **Initial State:**
      - Consider a database storing employee IDs: 10, 20, 30, 40, 50, 60, 70.
      - A B+-Tree of order 3 (maximum 3 keys per node) might look like:
        ```[30]
        /
        [10, 20] [40, 50, 60, 70]
    - Here, the root node stores the key `30` and divides the keys into two leaf nodes.
  - **Insertion of Key 25:**
    - After adding `25`, the second leaf node splits:
    - **Insertion of Key 25:**
      ```[30]
      /
      [10, 20, 25] [40, 50, 60, 70]
- **Hash Indexing**
  - **Definition:** Search keys are distributed uniformly across buckets using a hash function.
  - **Example:**
    - A hash function h(x)=x mod 3h(x) = x mod 3h(x)=x mod 3 is applied to employee IDs:
      - Employee IDs: 101, 102, 103.
      - Buckets:
        `Bucket 0: 102` Bucket 1: 103 Bucket 2: 101`

- Performance Metrics
    1. **Access Time:** Time taken to retrieve a record.
    2. **Insertion Time:** Time to add a new entry to the index.
    3. **Deletion Time:** Time to remove an entry.
    4. **Space Overhead:** Extra space used to store the index.
- Comparison of Index Types

Metric	**Dense Index**	**Sparse Index**	**Hash Index**
Space Usage	High	Low	Medium
Retrieval Speed	Fast	Moderate	Fast (if hash function is good)
Maintenance Overhead	High	Low	Moderate

- Applications
    - **Dense Index:** Suitable for primary keys.
    - **Sparse Index:** Works well for clustered data.
    - **B+-Tree:** Efficient for range queries.
    - **Hashing:** Best for equality searches.
- By selecting the appropriate indexing mechanism, database performance can be optimized significantly.

Complexity of Updates in B+-Tree

- B+-Trees are widely used due to their efficiency in indexing, particularly for dynamic data where insertions, deletions, and updates are common.
- **Cost Analysis of Insertion and Deletion**
    1. **Height-Based Complexity:**
        - The cost of an insertion or deletion in a B+-Tree depends on its height.
        - For a tree with K entries and a maximum fanout of n, the height h is approximately `O(log[n/2](K))`.
        - Each I/O operation (accessing a node from disk) is proportional to the height because the operation must traverse from the root to the affected leaf.
    2. **Practical Considerations:**
        - Internal nodes are often kept in memory (buffer), reducing the number of I/O operations.
        - Splits and merges are rare, as most operations affect only a single leaf node.
- **Node Occupancy:**
    - **Average Occupancy:**
        - **Random Insertions:** Around 2/3 of the node capacity.

        - **Sorted Insertions:** Around 1/2 of the node capacity (because of skewed data distribution).
- **Impact of Non-Unique Search Keys**
    - When search keys are non-unique (e.g., duplicates), additional challenges arise.
        1. **Handling Duplicates:**
            - **Buckets on Separate Blocks:** Stores pointers to records with the same key. However, this approach can lead to:
                - Poor performance due to extra I/O for accessing buckets.
                - Increased complexity for managing these buckets.
            - **List of Tuple Pointers:**
                - Maintains a list of pointers directly within the index.
                - **Downside:** Deleting a record becomes expensive if the search key has many duplicates, potentially leading to linear complexity in the number of duplicates.
            - **Make Search Keys Unique:**
                - Add a **record-identifier (record-id)** to the search key, simplifying insertion and deletion.
        2. **Space and Query Overhead:**
            - Using record-ids increases storage requirements but simplifies index management.
            - There is no additional query cost, as the system can still use the index efficiently.
- **Other Indexing Issues**
    1. **Record Relocation:**
        - When a record moves, all **secondary indices** (which store record pointers) must be updated.
        - In B+-Tree file organizations, node splits can cause record movement, making updates to secondary indices costly.
        - **Solution:** Use the search key of the B+-Tree file organization instead of record pointers for secondary indices.
    2. **Non-Unique Search Keys:**
        - If the search key is non-unique, add a record-id to differentiate entries.
        - This increases storage but simplifies operations like insertion and deletion.
    3. **Query and Node Split Costs:**
        - If record pointers are replaced with keys, finding a record involves an additional traversal to locate the actual data.
        - This adds a minor query overhead but significantly reduces the cost of node splits.

- **Example : Insertion in B+-Tree File Organization**
  - **Initial State:**
    - Suppose a B+- Tree with a maximum of 4 records per leaf stores employee records:

      -
        ```
        Leaf Node 2: [EmpID 40, EmpID 50]
        ```
    - Root points to these two leaf nodes.
  - **Insert `EmpID 35`:**
    - The record is inserted into the second leaf node:

      -
        ```
        Leaf Node 2: [EmpID 35, EmpID 40, EmpID 50]
        ```
  - **Insert `EmpID 25`:**
    - Leaf Node 1 is full, triggering a split:

      -
        ```
        New Leaf Node 2: [EmpID 25, EmpID 30]
        New Leaf Node 3: [EmpID 35, EmpID 40, EmpID 50]`
        ```
    - Root updates to:
      - `Root: [EmpID 25, EmpID 35]`

Indexing Strings in B+-Trees

- String keys in B+-trees require specific techniques to handle their variable lengths efficiently, especially in terms of storage and retrieval.
- **Variable-Length Strings as Keys**
  1. **Variable Fanout:**
     - Since strings have varying lengths, the number of keys (fanout) in each node varies.
     - **Splitting Criterion:** Nodes are split based on space utilization, not the number of pointers, ensuring efficient usage of available space.
  2. **Prefix Compression:**
     - **Internal Nodes:**
       - Key values stored in internal nodes can be prefixes of the full keys.
       - Sufficient characters are retained to distinguish between the subtrees.
         - Example: To separate "Silas" and "Silberschatz," the prefix **"Silb"** is sufficient.

    - **Leaf Nodes:**
      - Keys in leaf nodes can also be compressed by sharing common prefixes.
        - Example: In a leaf storing `["Silas," "Silberschatz"]`, both can be stored as `["Silas," "erschatz"]` to reduce storage.

- **Bulk Loading and Bottom-Up B+-Tree Construction**
  For inserting a large number of entries efficiently, **bulk loading** methods are used instead of inserting entries one by one.
  - **Challenges in One-at-a-Time Insertion:**
    - Each insertion requires at least one I/O operation if the leaf level does not fit in memory.
    - This is inefficient for loading a large number of entries.
  - **Efficient Bulk-Loading Techniques:**
    1. **Sort-First Insertion (Efficient Alternative 1):**
       - **Steps:**
         - Sort all entries using an **external-memory sorting algorithm**.
         - Insert them in sorted order into the B+-tree.
       - **Advantages:**
         - Improved I/O performance since entries are inserted sequentially.
       - **Drawbacks:**
         - Most leaf nodes remain only half full due to random splits.
    2. **Bottom-Up Construction (Efficient Alternative 2):**
       - **Steps:**
         - Sort all entries.
         - Build the tree **layer by layer**, starting from the leaf level.
       - **Advantages:**
         - Optimized space utilization and reduced I/O.
         - Preferred method for bulk-loading utilities in database systems.
- **Flash-Based Indexing**
  Flash storage has unique characteristics that affect B+-tree indexing.
  1. **I/O Costs on Flash:**
     - Random I/O is significantly faster compared to disk (20-100 microseconds).
     - Writes are not in-place, requiring expensive erase operations after repeated writes.
  2. **Optimization Strategies:**
     - Use smaller page sizes to reduce the cost of random writes and erases.
     - **Bulk-Loading:** Minimizes the number of page erases, making it highly effective for flash storage.

        - **Write-Optimized Trees:** Specially adapted tree structures minimize page writes and are better suited for flash-optimized search trees.
- **Indexing in Main Memory**
    - B+-trees designed for main memory indexing optimize for random access speeds and cache efficiency.
        1. **Cost of Random Access:**
            - While memory access is much faster than disk or flash, it is still slower compared to cache reads.
            - Cache misses significantly impact performance.
        2. **Optimizing Cache Usage:**
            - **Small Nodes:** B+-tree nodes are designed to fit within a cache line to minimize cache misses.
            - **Node Design:**
                - Use large nodes to optimize for disk access.
                - Internally structure nodes using small, tree-like structures instead of arrays to improve cache efficiency.

Example Scenarios

- **Prefix Compression Example:**
    - Keys: ["Silas", "Silberschatz", "Silence"]
    - Internal Node Compression:
        - Instead of storing all keys, store **["Sil"]** in the internal node as it distinguishes between subtrees.
    - Leaf Node Compression:
        - Store the following in the leaf:

            -
              ```
              ["ers" (for Silberschatz)]
              ["ence" (for Silence)]```
              ```

    - **Bottom-Up Construction Example:**
        1. **Initial Data:** Unsorted list of keys: [50, 20, 30, 40, 10].
        2. **Step 1:** Sort the data: [10, 20, 30, 40, 50].
        3. **Step 2:** Create the leaf level:
            - Leaf 1: [10, 20]
            - Leaf 2: [30, 40]
            - Leaf 3: [50].
        4. **Step 3:** Build the next layer:

    - Internal Node 1: [20, 40].
    - Root: [40].

Hashing

- **Static Hashing**:
  - Hash function `h` maps search-key values to bucket addresses.
  - **Buckets**: Units of storage containing entries (e.g., disk blocks).
  - Challenges:
    - Bucket overflow due to insufficient buckets or skewed distributions.
    - Addressed using overflow chaining (closed addressing) but can degrade performance over time.
  - **Deficiencies**:
    - Fixed bucket size limits flexibility for database growth or shrinkage.
    - Space may be wasted or overflow may occur as the database changes.
- **Dynamic Hashing**:
  - Solves static hashing limitations by dynamically resizing buckets.
  - Techniques include periodic rehashing (expensive) and incremental rehashing (linear or extendable hashing).
  - Used to handle variable-sized databases without massive disruptions.
- **Comparison of Hashing and Ordered Indexing**
  - **Hashing**:
    - Efficient for exact-match queries.
    - Performs poorly for range queries.
  - **Ordered Indexing**:
    - Better for range queries.
    - Can require reorganization during updates.
- **Multiple Key Access**
  - Strategies for multiple conditions in queries:
    1. Use separate indices for each attribute and filter results.
    2. Use composite indices (e.g., `(dept_name, salary)`) for better efficiency.
  - **Composite Search Keys**:
    - Lexicographic ordering allows efficient handling of multiple conditions (e.g., `dept_name = "Finance" AND salary < 80000`).
- **Indexing Techniques**
  - **Index Creation**:
    - SQL syntax: `CREATE INDEX index_name ON table_name (attribute_list)`
    - Types include clustered and unclustered indices.

    - **Covering Indices**:
        - Additional attributes added to leaf nodes to minimize actual record retrieval.
- **Write-Optimized Indices**
    - **B+-trees**: Efficient for reads but suboptimal for write-heavy workloads.
    - **Log-Structured Merge (LSM) Trees**:
        - Inserts handled in-memory, periodically merged to disk.
        - Benefits: Sequential I/O for inserts, full leaves reduce space wastage.
        - Drawbacks: Increased query costs due to multiple levels.
    - **Buffer Trees**:
        - Buffers at internal nodes optimize bulk inserts.
        - Benefits include less query overhead and adaptability to various tree structures.
- **Bitmap Indices**
    - Suitable for attributes with limited distinct values.
    - Use bitmaps to represent presence/absence of values across records.
    - Efficient for queries on multiple attributes using bitmap operations like AND, OR, and NOT.
    - Compact storage makes bitmap indices efficient.
- **Efficient Query Processing**
    - **Bitmap Operations**:
        - CPU-optimized, leveraging word-level operations for bitmaps.
        - Useful for range queries and matching multiple conditions efficiently.
- Key Concepts from Spatial and Temporal Indices:
- **Spatial Data:**
    - **Data Types and Queries:**
        - Databases store spatial data like lines, polygons, and raster images.
        - Queries can use spatial conditions (e.g., `contains`, `overlaps`) and combine spatial and nonspatial conditions.
        - Example queries:
            - **Nearest Neighbor:** Find the closest object to a given point.
            - **Range Queries:** Find objects within or partially inside a spatial region.
            - **Intersection and Union:** Combine or find commonalities in spatial regions.
            - **Spatial Join:** Combine spatial relations based on their location.
    - **Indexing of Spatial Data:**
        1. **k-d Tree:**
            - Early multi-dimensional indexing structure.
            - Partitions space at each tree level across dimensions cyclically.
            - Balanced splits ensure half the points lie on either side of a node.

        - Stops splitting when fewer than a threshold number of points remain.
    2. **k-d-B Tree:**
        - Extends the k-d tree with multiple child nodes per internal node.
        - Designed for secondary storage and bulkier datasets.
    3. **Quadtrees:**
        - Recursive partitioning of space into quadrants.
        - Each node corresponds to a rectangular region and has four children, representing quadrants.
        - Leaf nodes contain a limited number of points or no points.
    4. **R-Trees:**
        - N-dimensional extension of B+-trees for indexing rectangles and polygons.
        - Nodes hold bounding boxes that encompass child elements' rectangles/polygons.
        - Allows overlap of bounding boxes.
        - Common variants include R+ and R*-trees.
        - **Search Process:**
            - For each node, check children’s bounding boxes intersecting the query region.
            - Recursively traverse overlapping child nodes.
- **Indexing Temporal Data:**
    1. **Temporal Data Characteristics:**
        - Associated with time intervals (start and end times).
        - Active tuples have undefined or infinite end times.
    2. **Query Types:**
        - Search for tuples valid at a specific time or within a time interval.
        - Temporal indices improve query efficiency.
    3. **Temporal Index Design:**
        - Use spatial indices (e.g., R-tree) where time is an additional dimension.
        - Handle active tuples separately (those with infinite end times).
        - Index active tuples on attributes like `(a, start-time)` for efficient searching.
        - Temporal indices help enforce primary key constraints on time intervals.
- **Example Hash Index:**
    - A hash index can be applied to attributes like `ID` in a table (e.g., `instructor` table), enabling efficient lookup for exact matches.

File Organization and Indexing in DBMS

1. Basics of Hard Disks:

- Hard Disks (HD) store all data, including databases, files, and operating system structures (e.g., directories).
- **Components:**
  - **Platters:** Magnetic-coated disks for data storage.
  - **Read/Write (R/W) Head:** Reads/writes data on the platters.
  - **Stepper Motor:** Moves the R/W head with high precision.
  - **Controller:** Manages data transfer between HD and CPU.

2. Data Storage on Hard Disks:

- **Tracks:** Concentric circles on the disk surface where data is stored.
- **Sectors:** Smallest unit of storage, usually 512 bytes.
- **Clusters:** Groups of contiguous sectors, allowing efficient data transfer.
- **Data Writing and Reading:**
  - Files are stored in sectors, and a file's size determines the number of sectors needed:
    - Example: A file of 800 bytes requires 2 sectors (`ceil(800/512) = 2`).
  - Data is optimally stored in contiguous sectors to minimize fragmentation.
- **Fragmentation:**
  - Occurs when a file's sectors are spread across non-contiguous locations.
  - Increases seek time, slowing down R/W operations.

3. Key Metrics in Hard Disk Operations:

- **Block:** Amount of data in a sector (512 bytes in this case).
- **Seek Time:** Time to position the R/W head at the correct sector.
- **Latency:** Time for the desired sector to rotate under the R/W head.

4. DBMS File Organization:

- **Special Needs:**
  - Database tables can be very large, requiring dedicated HD partitions.
  - DBMS directly controls the HD driver to manage sectors and data allocation efficiently.
- **Challenges:**
  - Avoiding fragmentation for large datasets.
  - Optimizing data placement for sequential and random access patterns.

5. Advantages of Proper File Organization:

- Minimizes seek time and latency.
- Reduces fragmentation for large database tables.
- Enhances performance for frequent read/write operations.

6. Importance in DBMS:

- Ensuring contiguous data storage improves performance in:
  - Sequential scans (e.g., querying large tables).

    - Index lookups and joins.
- DBMS-controlled partitions optimize large-scale data storage and retrieval.

Typical DB Operations and Performance Considerations

Importance of Fast Query Response:

1. **Simultaneous Access by Multiple Users:**
    - **Example 1:** An online shopping platform (e.g., Amazon) where multiple users are browsing and purchasing items at the same time.
    - **Example 2:** A social media platform (e.g., Facebook) where users are constantly interacting, posting, and commenting.
    - **Example 3:** A stock trading application (e.g., E*TRADE) where thousands of traders are executing buy/sell orders simultaneously.
2. **Large Tables with Many Rows:**
    - **Example 1:** A customer database for a large retail chain containing millions of customer records.
    - **Example 2:** A national health database with patient information spanning multiple decades.
    - **Example 3:** A university student database with records for tens of thousands of students, including course enrollment and grades.

- **User Tasks in a Database:**
    - Users generally perform operations such as searching, creating, modifying, or deleting records in a table. Each of these tasks requires specific attention in terms of performance optimization.
        1. **Search for a Particular Row:**
            - **Goal:** Retrieve a specific row based on a query condition.
            - Example: Finding a student record by student ID or an employee by name.
        2. **Create a Row in a Table:**
            - **Goal:** Insert a new record into the table.
            - Example: Adding a new product to an inventory system or registering a new student in a school database.
        3. **Modify Data in a Row:**
            - **Goal:** Update the data within a particular record.
            - Example: Updating the contact information of an employee or changing a customer’s order status.
        4. **Delete a Row from a Table:**
            - **Goal:** Remove a record from the table.

            - Example: Deleting a canceled order from a sales database or removing an inactive employee’s record.
- **Storage of Tables on a Hard Disk (HD):**
    - **1. Storing Tables:**
        - Each table is stored as an independent file on the disk, potentially as a series of blocks (sectors).
        - For large databases, the tables may span multiple sectors or even files.
    - **2. Contiguous Storage of Attributes:**
        - **Why Attributes are Stored Together:** Attributes of a record (e.g., name, ID, department) are usually accessed together in queries. Storing them contiguously minimizes disk seeks and reduces read time.
        - **Why Sequence Matters:** Each record in a table must store its attributes in the same order because the DBMS expects this consistent ordering when reading or writing data.
    - **3. Record Storage Sequence:**
        - Records are stored in a sequence chosen by the DBMS. Since there is no inherent order for storing records, the sequence can be optimized for query performance, ensuring that frequently accessed records are grouped or that less-frequent ones are kept separately.
- **Record Storage in Blocks and Sectors:**
    - **Records in a Block:**
        - Each record is typically stored in a block, assuming there is enough space. If the record cannot fit entirely in one block, it will not be split across blocks; instead, it will be placed in the next available block.
    - **Field Separator and Record Separator:**
        - Within each block, records are separated by **field separators** (e.g., commas in CSV files) and **record separators** (e.g., newline characters).
    - **Block Storage Example:**
        - Multiple records are placed in a single block as long as there is enough space. As a result, some blocks may contain a mix of records, while others may have wasted space.
    - **Wasted Disk Space:**
        - Some blocks may contain unutilized space at the end if the last record does not fully fill the block. This unused space can contribute to fragmentation.
- **Optimizing Disk Access:**
    - **Minimizing Fragmentation:**
        - It is important for the DBMS to manage record placement efficiently to reduce fragmentation, where records are spread out across non-contiguous disk areas.

- **Efficient Block Management:**
  - Since the DBMS interacts with the disk in terms of blocks, optimizing how records are stored within blocks and minimizing disk seeks during queries is critical for performance.

Factors Affecting Database Operation Speed

- The speed of a DBMS executing queries is influenced by data transfer between the hard disk (HD) and RAM, as well as the type of operation being performed.
- **Key Operations Impacting Speed:**
  1. **Finding a Row:**
     - Searching is faster with an index (O(log n)) but slower with a full table scan (O(n)).
  2. **Inserting a Row:**
     - Insertion time depends on the table size, index updates, and storage engine write performance.
  3. **Deleting a Row:**
     - Deletion can be fast but may require index updates or table reorganization.
  4. **Modifying a Cell:**
     - Modifications are quick in memory but slower if indexes are updated or data is written to disk.
- **Data Transfer Between HD and RAM:**
  - **Speed Difference:**
    CPU-RAM communication is 100-1000x faster than HD-RAM transfer. For example, reading a block from the HD takes **10-3 seconds**, while accessing it in RAM takes **10-6 seconds**.
  - **Time to Read a Block from HD:**
    - **Sector Address Time:** Time to send the sector address to the HD.
    - **Seek Time:** Time for the read/write head to move to the correct location (dominates access speed).
    - **R/W Time:** Time to read the data and transfer it to the buffer.
    - **Transfer Time:** Time to move data from the buffer to RAM.
- **Disk Access Speed:**
  - **Seek Time:** The time it takes for the R/W head to find the data, which is the most time-consuming factor.
  - **Rotational Latency:** The time it takes for the disk to rotate to the correct sector, significant for HDDs but less so for SSDs.

Heap Files (Pile Storage)

- **Insertion**: When inserting a new record, it is simply added at the end of the last block of the heap file. This method is fast but results in poor search performance.
- **Search**: To find a record, a linear search is used, which can be inefficient for large datasets.
- **Update**: Like search, updating a record involves a linear search and then modifying the record.
- **Delete**: Deletion requires a search to locate the record and may leave unused space in the block. Different strategies exist to handle this space, like marking records as deleted or periodically rewriting the file.
- **Performance**: While heap files offer fast insertions and updates, searching and deleting records are inefficient due to the need for linear scans across blocks.

Sorted Files

- **Sorted by a Key**: A table is organized in sorted order based on an attribute, enabling efficient search via binary search.
- **Search**: The binary search method is significantly faster than linear search, especially for large datasets.
- **Insertion/Update**: Inserting a new record requires finding the correct position and may involve shifting other records, which can be slow. Using an overflow file (a separate heap file) for insertions can help mitigate this.
- **Performance**: Sorted files drastically improve search efficiency compared to heap files, but insertions and updates (especially when the ordering attribute changes) are more costly.

Faster Searching: Hashing Functions

- **Hashing**: A method that distributes records into "buckets" using a hashing function based on a chosen attribute (e.g., a key). The goal is to achieve faster search times by accessing a specific bucket.
- **Hashing Procedure**:
  1. A hashing key is calculated from the attribute value.
  2. The corresponding bucket is identified using the formula `K mod M`.
  3. The record is inserted or searched within the bucket.
- **Performance**: In general, hashing offers near-constant time for searches and insertions, making it very efficient. However, collisions (when multiple records map to the same bucket) can occur, which may require handling overflow records.
- **Overflow Handling**: If a bucket is full, overflow records are stored in subsequent buckets, ensuring that records are eventually written, though this can degrade performance slightly in extreme cases.

Basics of Indexes

- **Indexes**: Indexes are additional structures (like a physical index in a textbook) that help DBMSs locate records quickly based on various attributes.
- **Use of Indexes**: While the document doesn't go into details about index types, it suggests that indexes are used to improve search operations, even when records are stored in a sorted order or using hashing.