# Indexing and Hashing

## 1. Introduction to Indexing and Hashing

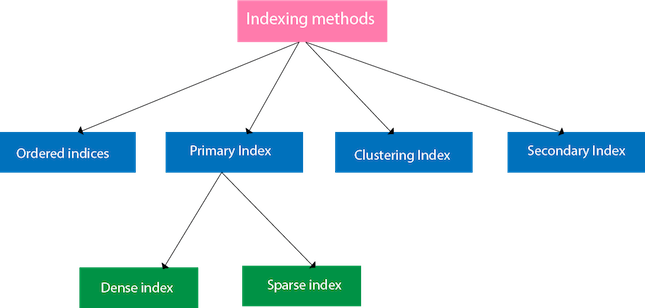
Indexing and hashing are crucial techniques in database management, enhancing the speed and efficiency of data retrieval operations. Each approach is suited to different types of queries, and both significantly reduce data access times.

* **Purpose**: To minimize retrieval times and improve database performance.
* **Indexing**:
  + Creates an ordered structure of data based on a specific attribute (search key).
  + Similar to an index in a book, helping to locate data quickly.
* **Hashing**:
  + Uses a hash function to assign data to specific "buckets" for rapid access.
  + Ideal for equality-based queries (e.g., finding a record with a specific ID).

### Real-World Examples:

* **Library Systems**: Author catalogs are organized alphabetically (indexing).
* **Banking Systems**: Account numbers hashed to specific storage locations for fast lookup.

## 2. Indexing Types and Techniques



### 2.1 Ordered Indices

Ordered indices are a basic type of index where data entries are stored in sorted order based on the search key.

* **Types of Ordered Indices**:
  + **Primary Index**:
    - Organized by the primary key or another unique attribute.
    - Often found in records arranged sequentially, like a list of student IDs.
  + **Clustering Index**:
    - Uses a non-unique attribute to order records.
    - Groups similar data together (e.g., grouping employees by department).
  + **Secondary Index**:
    - Provides alternate access paths for non-primary keys.
    - Useful when data is searched by different attributes frequently.
* **Benefits**:
  + Enables efficient searching and range queries (e.g., finding all records between two dates).
  + Supports fast retrieval, especially in sorted datasets.
* **Use Case**: A university database with a primary index on student ID and a clustering index on major or graduation year.

### 2.2 Dense and Sparse Indices

The choice between dense and sparse indices depends on the trade-off between storage space and access speed.

* **Dense Index**:
  + Contains an index entry for every search-key value.
  + Useful when every key is needed for exact match queries.
  + Common in systems with unique identifiers like customer IDs.
* **Sparse Index**:
  + Contains index entries only for some search keys.
  + Requires sequential scanning between entries, but reduces storage.
  + Efficient for large datasets sorted by search key.
* **Advantages and Disadvantages**:
  + **Dense Index**:
    - **Pros**: Faster search times for exact matches.
    - **Cons**: Higher storage and maintenance costs.
  + **Sparse Index**:
    - **Pros**: Lower storage requirements.
    - **Cons**: Slower searches due to sequential scanning.

### 2.3 Multilevel Indexing

Multilevel indexing is used when a primary index is too large to fit into memory.

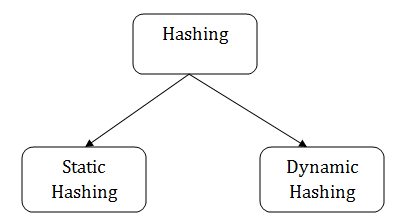
* **Structure**:
  + A hierarchy of indices with outer indices pointing to inner indices.
  + Allows large indices to be broken down into manageable levels.
* **Applications**:
  + Common in large systems, such as bank databases or e-commerce catalogs.
  + Improves access time by reducing the need to scan the entire index.

### 2.4 Secondary Indexing

Secondary indices enable searching based on non-primary attributes.

* **Characteristics**:
  + Dense in nature, as every record in the secondary attribute has an index entry.
  + Allows retrieval based on frequently searched attributes (e.g., department, salary).
* **Use Cases**:
  + **Student Database**: Primary index on ID, secondary index on department.
  + **Employee Database**: Primary index on employee ID, secondary index on department.

## 3. Hashing Types and Techniques



### l

### 3.1 Static Hashing

Static hashing uses a fixed number of buckets determined by a hash function.

* **Key Points**:
  + Each record is mapped to a bucket based on its search key.
  + Effective for stable datasets with predictable storage needs.
  + Inflexible for datasets that grow, as bucket sizes cannot easily be changed.
* **Examples**:
  + A university database hashing student records by ID.
  + Bank account records assigned to fixed buckets based on account number.

### 3.2 Dynamic Hashing

Dynamic hashing expands or shrinks the number of buckets based on data growth.

* **Advantages**:
  + Adapts to changes in database size without requiring rehashing.
  + More flexible than static hashing.
* **Techniques**:
  + **Extendable Hashing**: Uses a prefix-based hash table that expands as needed.
  + **Linear Hashing**: Adds new buckets incrementally, useful in distributed databases.
* **Example**: A retail database growing with new product data that can dynamically adjust its hashing.

### 3.3 Hash Functions and Bucket Management

The effectiveness of a hash function depends on its ability to distribute records uniformly.

* **Characteristics of a Good Hash Function**:
  + Uniformly distributes keys to prevent "hotspots" in buckets.
  + Minimizes collisions by avoiding overlapping buckets.
* **Collision Handling Techniques**:
  + **Chaining**: Uses linked lists within each bucket to handle overflow.
  + **Open Addressing**: Uses empty slots within the hash table to resolve collisions.
* **Bucket Overflow Management**:
  + **Linear Probing**: Looks for the next available slot linearly.
  + **Quadratic Probing**: Uses a quadratic function to find the next slot.

## 4. B-Tree and B+-Tree Structures

### 4.1 B-Tree

B-Trees are balanced trees that store keys in a sorted, hierarchical structure.

* **Characteristics**:
  + Each node contains a sorted list of keys and pointers to children.
  + Provides balanced structure for efficient insertion, deletion, and search.
* **Advantages**:
  + Reduces the number of disk accesses due to the balanced structure.
  + Ideal for read-heavy applications needing fast retrieval.

### 4.2 B+-Tree

B+-Trees are an extension of B-Trees, optimized for range queries and sequential access.

* **Structure**:
  + Keys are stored in internal nodes, while leaf nodes contain pointers to records.
  + Leaf nodes are linked for in-order traversal, ideal for range queries.
* **Advantages Over B-Trees**:
  + Supports fast, in-order traversal.
  + More efficient for range queries (e.g., retrieving all employees within a salary range).
* **Applications**:
  + **Inventory Systems**: Indexing products by category for quick retrieval.
  + **Library Databases**: Organizing books by author or genre for efficient access.

## 5. Comparison of Indexing and Hashing

Both indexing and hashing are designed to improve retrieval times but serve different types of queries.

* **Indexing**:
  + Supports ordered data, making it suitable for range queries.
  + Slower for equality-based lookups compared to hashing.
* **Hashing**:
  + Ideal for equality searches due to constant-time access.
  + Less effective for range queries as it lacks an inherent order.
* **Considerations**:
  + **Query Type**: Indexing for range queries, hashing for exact matches.
  + **Data Size**: Indexing scales better with data growth, while hashing may need restructuring.

## 6. Practical Examples and Use Cases

In practical applications, the choice between indexing and hashing depends on data access patterns.

* **Banking Systems**:
  + Use B+-Trees for indexing accounts by number and supporting range-based queries, such as viewing transactions within a specific date range.
* **Retail Databases**:
  + Use hash tables for order lookups by order ID, providing quick key-based access for customer service representatives.
* **Library Management**:
  + Libraries implement multilevel indexing for large collections, with primary indices on titles and secondary indices on authors.

Choosing the right data structure affects the efficiency, flexibility, and storage optimization of databases, impacting both performance and scalability.