# Indexing and Hashing

## 1. Introduction to Indexing and Hashing

Indexing and hashing are two fundamental techniques used in database management to facilitate efficient data retrieval. Both serve to minimize access time and improve performance, though they are suited to different types of search operations. Indexing arranges data records in a way that supports fast retrieval based on search keys, while hashing employs a hash function to map keys to specific buckets or locations. This document provides an overview of the main concepts, types, advantages, and examples of indexing and hashing, with a focus on database applications.

## 2. Indexing Types and Techniques

### 2.1 Ordered Indices

Ordered indices involve sorting index entries based on the search key values. Examples include the author catalog in a library where data is arranged alphabetically. There are several types of ordered indices, such as primary, clustering, and secondary indices.

### 2.2 Dense and Sparse Indices

In dense indices, every search-key value in the file has an entry in the index, providing high accuracy but consuming more space. Sparse indices, on the other hand, only have entries for some of the search-key values, which reduces space but may increase search time as it requires a sequential scan starting from a nearby index entry.

### 2.3 Multilevel Indexing

Multilevel indexing is used when a primary index becomes too large to fit in memory. In this case, the index itself is indexed using additional levels. The outer index is typically sparse, while the inner index provides faster access. Multilevel indices require maintenance across all levels for any insertions or deletions.

### 2.4 Secondary Indexing

Secondary indexing enables access based on non-primary search keys, which helps in finding all records matching a specific field value. This approach is particularly useful when a dataset is sorted by a different attribute than the one being searched. Secondary indices are generally dense.

## 3. Hashing Types and Techniques

### 3.1 Static Hashing

Static hashing maps search-key values to a fixed set of bucket addresses using a hash function. Each bucket can store multiple records, though issues arise when the bucket becomes full. Static hashing works well for databases with stable sizes, but resizing requires rehashing.

### 3.2 Dynamic Hashing

Dynamic hashing, unlike static hashing, adapts to growth and shrinkage in the dataset. The hash function used can be dynamically altered to distribute data better. Examples include extendable hashing and linear hashing. Dynamic hashing enables flexibility but may require extra processing and storage for rehashing.

### 3.3 Hash Functions and Bucket Management

An effective hash function distributes keys evenly across buckets, minimizing collisions. However, when collisions occur, methods such as chaining or open addressing are used. Effective bucket management and collision handling are essential to maintaining performance.

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

B-Trees and B+-Trees are data structures designed for efficient indexing. B+-Trees are often preferred in databases because they provide consistent retrieval times and self-organize as data is added or removed. B-Trees save storage by avoiding duplicate keys, while B+-Trees support fast in-order traversal, beneficial for range queries.

## 5. Comparison of Indexing and Hashing

Both indexing and hashing have unique strengths. Indexing supports ordered data and range-based searches, while hashing provides faster lookups for specific key values. Indexing is generally better for queries needing sorted data, whereas hashing excels in constant-time access for equality-based searches.

## 6. Practical Examples and Use Cases

In practice, indexing and hashing techniques are chosen based on the type of queries a database will handle. For instance, relational databases often implement B+-Tree structures for ordered indexing, while NoSQL databases may rely on hash-based techniques for quick key-based lookups.