# Indexing and Hashing

## 1. Introduction to Indexing and Hashing

Indexing and hashing are essential techniques used in database management to enhance data retrieval efficiency. These methods organize data in a way that minimizes access time, helping databases to handle queries quickly and effectively. For example, indexing in a library's author catalog allows users to find books by a particular author easily, while hashing can be compared to a postal system where addresses (hashes) lead to specific locations (data buckets). This document explores these key concepts in detail, providing definitions, types, advantages, techniques, and examples.

## 2. Indexing Types and Techniques

### 2.1 Ordered Indices

Ordered indices store index entries in sorted order based on the search key values, aiding quick lookup. Examples include phone directories, where entries are sorted alphabetically by name, and databases that use primary keys to organize records. Types of ordered indices include primary indices, where the key defines the sequential file order, and clustering indices, useful when records are frequently accessed in a specific order.

### 2.2 Dense and Sparse Indices

Dense indices have an index record for every search-key value, offering high accuracy but occupying more storage space. In contrast, sparse indices only record some search keys, leading to a smaller index with less maintenance but slower searches. For example, a dense index might be useful for customer IDs in a customer database, whereas a sparse index could be used for non-unique attributes like location in a large database of stores, reducing storage demands.

### 2.3 Multilevel Indexing

When the primary index is too large to fit in memory, a multilevel index structure is often applied. A multilevel index, or hierarchical index, builds secondary indices on top of the primary index. This structure is common in large databases, such as bank transaction logs or e-commerce customer databases. Multilevel indexing enhances efficiency by creating additional levels that aid in locating entries quickly, even in large datasets.

### 2.4 Secondary Indexing

Secondary indices support searching on attributes other than the primary key, allowing for flexible search criteria. For instance, in a student database sorted by student ID, a secondary index might be used on the 'department' field to quickly retrieve all students within a specific department. Secondary indices are generally dense, ensuring that every value in the secondary attribute has an index entry.

## 3. Hashing Types and Techniques

### 3.1 Static Hashing

Static hashing assigns a fixed number of buckets and relies on a hash function to determine each record's location. For example, a banking database might use static hashing to locate accounts, where account numbers are hashed to specific buckets. While efficient, static hashing is limited by its inability to adapt to changing data sizes without significant rehashing.

### 3.2 Dynamic Hashing

Dynamic hashing is a more adaptable form of hashing that grows and shrinks with the data. Extendable hashing, a type of dynamic hashing, uses a flexible hash function that can adjust the number of bits used based on data volume. For example, a growing database of customer orders might use dynamic hashing to manage fluctuating data without the need for constant rehashing or restructuring.

### 3.3 Hash Functions and Bucket Management

Hash functions are crucial in determining which bucket a search key is mapped to. Ideally, a hash function distributes keys uniformly across all buckets, but when collisions occur, methods such as chaining (linking overflow records) or open addressing (using alternative slots within the bucket) are used. Bucket overflow is managed through strategies like linear probing or double hashing, ensuring that all records can still be efficiently retrieved.

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

B-Trees and B+-Trees are balanced tree structures that store search keys in a way that facilitates efficient querying, insertion, and deletion. While B-Trees minimize redundant key storage, B+-Trees allow for in-order traversal of all records, which is advantageous for range queries. For instance, an inventory management system may use B+-Trees to manage items sorted by categories or prices. B+-Trees self-organize as new records are added or removed, maintaining their balanced nature.

## 5. Comparison of Indexing and Hashing

While both indexing and hashing aim to improve data retrieval, they are suited to different scenarios. Indexing works best for ordered data and supports range-based searches, while hashing is faster for equality-based searches. Hashing is generally less suitable for range queries, as it does not maintain order. In practice, a B+-Tree index may be preferred for an application requiring ordered access, while hashing might be favored in situations requiring frequent point queries, like username-based logins in a user database.

## 6. Practical Examples and Use Cases

In real-world applications, choosing between indexing and hashing depends on the types of queries and data access patterns. Examples include:  
  
- \*\*Banking Systems\*\*: Often use B+-Trees to index accounts by account number and support range-based queries like listing accounts in a balance range.  
  
- \*\*Retail Databases\*\*: For order or product lookups, hash tables are used for quick key-based access to product IDs or order numbers.  
  
- \*\*Library Management\*\*: Libraries frequently use multilevel indexing to manage large collections, with primary indices on titles and secondary indices on authors.  
  
The choice of data structure greatly impacts the performance, flexibility, and storage efficiency of these systems.