# DBMS

Week 11

# Chapter 14 indexing

Many queries reference only a small proportion of the records in a file. For example, a query like “Find all instructors in the Physics department” or “Find the total number of credits earned by the student with ID 22201” references only a fraction of the instructor or student records. It is inefficient for the system to read every tuple in the instructor relation to check if the dept name value is “Physics”. Likewise, it is inefficient to read the entire student relation just to find the one tuple for the ID “22201”. Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures that we associate with files.

## Basic Concepts

## Index in Databases

- An index in a database system serves a similar purpose to an index in a textbook or book.
- It allows efficient retrieval of specific data items (e.g., records) from the database.
- Indices make it possible to locate data without scanning the entire database, improving query performance.

## Comparison to Book Indices

- Just as a book's index helps readers find specific topics or information within the book, a database index helps users locate specific data within a database.
- Both book indices and database indices are organized to facilitate quick access to relevant information or records.

## Types of Database Indices

- There are two fundamental types of database indices:
    - Ordered Indices: These are based on a sorted ordering of attribute values and support efficient access types like finding specific attribute values or attribute value ranges.
    - Hash Indices: These rely on a hash function to distribute values uniformly across a range of buckets.

## Evaluation Criteria for Indexing Techniques

- When choosing an indexing technique, several factors need to be considered:
  - Access Types: The types of access operations the index can efficiently support (e.g., exact value lookup or range queries).
  - Access Time: The time taken to locate a specific data item using the index.
  - Insertion Time: The time taken to insert a new data item into the indexed structure.
  - Deletion Time: The time taken to delete a data item and update the index structure.
  - Space Overhead: The additional storage space required by the index structure.

## Multiple Indices

- In practice, databases often require multiple indices for a single file or relation.
- Each index corresponds to a specific search key (attribute or set of attributes).
- Having multiple indices allows users to search for data using different criteria, such as searching by author, subject, or title in a library catalog.

## Search Key

- An attribute or set of attributes used to look up records in a file is referred to as a search key.
- Search keys are used to perform efficient data retrieval operations on indexed data.

## Hash Indices

## Hash Function and Buckets

- Hashing involves using a hash function to map search-key values (K) to bucket addresses (B). A bucket is a unit of storage that can store one or more records or index entries.

## In-Memory Hash Indices

- In an in-memory hash index, the set of buckets is typically represented as an array of pointers. Each bucket stores the index entries or records associated with that bucket.

## Hash Function Usage

- To insert a record with a search key Ki, the hash function h(Ki) is computed to determine the bucket for that record. The record's index entry is then added to the appropriate bucket.

## Lookup and Deletion

- For lookups, the hash function is used to find the bucket, and the search key is compared with the entries in that bucket. Deletion involves locating the bucket and removing the record from it.

## Bucket Overflow

- When a bucket becomes full, overflow chaining is used. Overflow buckets are linked together in a linked list, and records that cannot fit in the original bucket are stored in these overflow buckets.

## Disk-Based Hash Indices

- In disk-based hash indices, the concept of buckets and overflow buckets remains, but disk blocks are used to store them. When an overflow occurs, a new overflow bucket is added to the linked list.

## Handling Skew

- Skew in the distribution of records across buckets can lead to bucket overflow. Careful selection of hash functions can help minimize skew. Additionally, the number of buckets is chosen to include some empty space to reduce the probability of overflow.

## Static Hashing

- The number of buckets is fixed when the index is created, which can become a limitation if the number of records exceeds expectations. Static hashing may require rebuilding the index with more buckets.

## Dynamic Hashing

- Dynamic hashing techniques allow for a more incremental increase in the number of buckets as needed, avoiding the need for a complete index rebuild. Linear hashing and extendable hashing are examples of such dynamic techniques.

## Multiple-Key Access

## Using Multiple Single-Key Indices

- In some cases, a database may have multiple single-key indices on different attributes of a relation.
- When executing a query that involves conditions on multiple attributes, different strategies can be applied to utilize these indices.
- For example, if there are indices on 'dept name' and 'salary,' and a query seeks instructors in the Finance department with a salary of $80,000, three strategies are possible:
  - Use the 'dept name' index to find Finance department records and then check each for the salary condition.
  - Use the 'salary' index to find instructors with a salary of $80,000 and then check each for the department condition.
  - Use both the 'dept name' and 'salary' indices simultaneously to find the intersection of records satisfying both conditions.

## Challenges with Multiple Indices

- The choice of strategy depends on factors like the number of records matching each condition.
- The third strategy takes advantage of multiple indices but may still be inefficient if many records need to be scanned.

## Bitmap Indices

- Bitmap indices are introduced as a potential solution to speed up the intersection operation used in the third strategy.
- Bitmap indices can significantly improve the performance of this type of query.

## Indices on Multiple Keys

- Another approach is to create an index on a composite search key, which combines multiple attributes.
- An ordered (B+-tree) index on the composite search key (e.g., 'dept name' and 'salary') can efficiently handle queries with conditions on both attributes.

## Handling Range Queries

- Queries with equality conditions on one attribute and range conditions on another can also be efficiently processed using composite search keys and ordered indices.

## Limitations of Composite Search Keys

- While composite search keys are powerful, they may not be as efficient for certain queries, particularly those involving comparison conditions on non-equality attributes.

## Covering Indices

- Covering indices store extra attributes (beyond the search-key attributes) along with record pointers.
- They allow some queries to be answered using only the index, without accessing the actual records.
- Covering indices can improve query performance and reduce the size of the search key.

## Creation of Indices

## Index Creation Syntax

- While the SQL standard doesn't specify a specific syntax for creating indices, most relational database systems support SQL commands for creating and dropping indices. The common syntax for creating an index is:
  - create index <index-name> on <relation-name> (<attribute-list>);
  - <index-name> is the name of the index.
  - <relation-name> is the name of the relation (table) on which the index is created.
  - <attribute-list> is the list of attributes that form the search key for the index.

## Index Types

- Databases may support various types of indices, and you can specify the index type as part of the index creation command. The available index types depend on the specific database system.

## Unique Indices

- To declare an attribute or list of attributes as a candidate key (a unique key), you can use the create unique index syntax instead of create index. This enforces uniqueness constraints on the indexed attributes.

## Automatic Index Usage

- When users submit SQL queries, the query processor automatically uses available indices to optimize query execution. This allows for faster query performance when indices are appropriately designed.

## Benefits of Indices

- Indices are particularly useful for attributes involved in selection conditions or join conditions in queries. They can significantly reduce the cost of queries, especially when dealing with large datasets.

## Index Maintenance Overhead

- While indices enhance query performance, they come with a cost. They need to be updated whenever there are modifications (inserts, updates, deletes) to the underlying data. Creating too many indices can slow down update operations.

## Importance of Planning

- It's essential to plan and create indices based on the specific needs of your application. Identifying which indices are critical for performance and creating them before deploying the application is crucial.

## Primary Key and Foreign Key Indices

- Most database systems automatically create an index on the primary key of a relation. This index helps ensure the uniqueness of primary key values and speeds up primary key lookups. Creating indices on foreign key attributes can also be beneficial for query performance, especially in join operations.

## Index Tuning Tools

- Database systems often provide tools like index tuning wizards or advisors that help database administrators identify the need for creating or optimizing indices based on query and update patterns.

## Automated Index Creation

- Some cloud-based database systems offer automated index creation to improve query performance without manual intervention by database administrators. These systems analyze query execution patterns and create indices as needed.

## Bitmap Indices

## Sequential Numbering of Records

- To use bitmap indices, records in a relation must be sequentially numbered. This sequential numbering allows for easy retrieval of records based on their record number.

## Fixed Values or Value Ranges

- Bitmap indices are particularly useful when dealing with attributes that have a limited number of distinct values or when values are grouped into ranges. For example, an attribute like "gender" that can have values "male" or "female" is a good candidate for a bitmap index.

## Bitmap Structure

- A bitmap index consists of one bitmap for each distinct value or value range of the indexed attribute. Each bitmap is essentially an array of bits, with one bit corresponding to each record in the relation.

## Setting Bits

- In a bitmap index, the ith bit of a bitmap for a particular value or value range is set to 1 if the record numbered i has that value for the indexed attribute. All other bits in the bitmap are set to 0.

## Bitmap Intersection

- Bitmap indices are particularly useful when performing selections that involve multiple attributes. When you want to select records based on multiple conditions (e.g., gender = 'female' and income level = 'L2'), you can retrieve the corresponding bitmaps for each condition and perform a logical AND operation (intersection) on them to get a new bitmap representing the combined condition.

## Efficiency for Multiple Conditions

- Bitmap indices are highly efficient when multiple conditions need to be checked simultaneously because they allow you to perform bitwise operations on bitmaps, which is very fast.

## Limitations

- Bitmap indices are most effective when the selectivity of each condition is relatively low. If a large portion of the records satisfy a particular condition, it may not be efficient to use a bitmap index for that condition.

## Aggregations

- Bitmap indices can be useful for aggregations as well, such as counting the number of records that satisfy specific conditions. You can count the number of set bits in a bitmap to get the count of matching records.

## Combination with Other Indexes

- In practice, bitmap indices are often used in combination with other index types, such as B+-tree indices, to optimize query performance. This approach allows for efficient filtering of records using bitmap indices and subsequent retrieval of detailed information using B+-tree indices.

## Class activity

- Index type
  - Ordered indices
  - Hash indices
- Evaluation factors
  - Access types
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead
- Search key
- Ordered indices
  - Ordered index
  - Clustering index
  - Primary indices;
  - Nonclustering indices
  - Secondary indices
  - Index-sequential files
- Index entry
- Index record
- Dense index
- Sparse index
- Multilevel indices
- Nonunique search key
- Composite search key
- B+-tree index files
  - Balanced tree
  - Leaf nodes
  - Nonleaf nodes
  - Internal nodes
  - Range queries
  - Node split
  - Node coalesce
  - Redistribute of pointers
  - Uniquifier
- B+-tree extensions
  - Prefix compression
  - Bulk loading
  - Bottom-up B+-tree construction
- B-tree indices
- Hash file organization
  - Hash function
  - Bucket
  - Overflow chaining
  - Closed addressing
  - Closed hashing
  - Bucket overflow
  - Skew
  - Static hashing
  - Dynamic hashing
- Multiple-key access
- Covering indices
- Write-optimized index structure
  - Log-structured merge (LSM) tree
  - Stepped-merge index
  - Buffer tree
- Bitmap index
- Bitmap intersection
- Indexing of spatial data
  - Range queries
  - Nearest neighbor queries
  - k-d tree
  - k-d-B tree
  - Quadtrees
  - R-tree
  - Bounding box
- Temporal indices
- Time interval
- Closed interval
- Open interval