Summary of Key Concepts: Query Optimization (Software Engineer Focus)

This document summarizes essential concepts related to database query optimization and indexing, focusing on aspects relevant for software development.

Query Optimization

- **Definition**: The process of improving SQL query efficiency to make them execute faster and consume fewer system resources (CPU, I/O, memory).
- **Importance**: Crucial for the performance of database applications, especially as data volumes grow. Slow queries can severely impact user experience and system scalability.

Common Query Optimization Techniques

- **Use Indexes Efficiently**: Create indexes on columns frequently used in WHERE clauses, JOIN conditions, ORDER BY, and GROUP BY. Indexes act like a book's index, allowing the database to find specific rows much faster without scanning the entire table.
- **Select Only Necessary Columns**: Avoid using SELECT *. Specify only the columns required by your application. This reduces data transfer over the network and processing load on the database and application.
- **Optimize JOIN Operations**: Choose the appropriate JOIN type (e.g., INNER JOIN, LEFT JOIN). Ensure columns used in JOIN conditions (ON clause) are indexed.
- **Partition Large Tables**: For very large tables, partitioning (splitting a table into smaller, manageable pieces based on certain criteria like date ranges or regions) can improve query performance by allowing the database to scan only relevant partitions.
- **Cache Results**: For frequently executed queries that return relatively static data, consider caching the results in your application or using database caching mechanisms to avoid repeated computation.

Physical Storage & Performance Context

- **Storage Hierarchy**: Databases use different storage types: Primary (RAM - fast, volatile), Secondary (SSD/HDD - slower, non-volatile), Tertiary (Tape - archive).
- **RAM vs. Disk**: Accessing data from RAM is significantly faster (nanoseconds) than from disk (milliseconds) due to mechanical delays in HDDs or I/O limitations.
- **Why Optimization Matters**: Databases try to keep frequently accessed data and indexes in RAM (buffer pool/cache). Optimization techniques, especially indexing, aim to minimize slow disk I/O operations by allowing the database to quickly locate the required data, ideally already in RAM or with minimal disk reads.

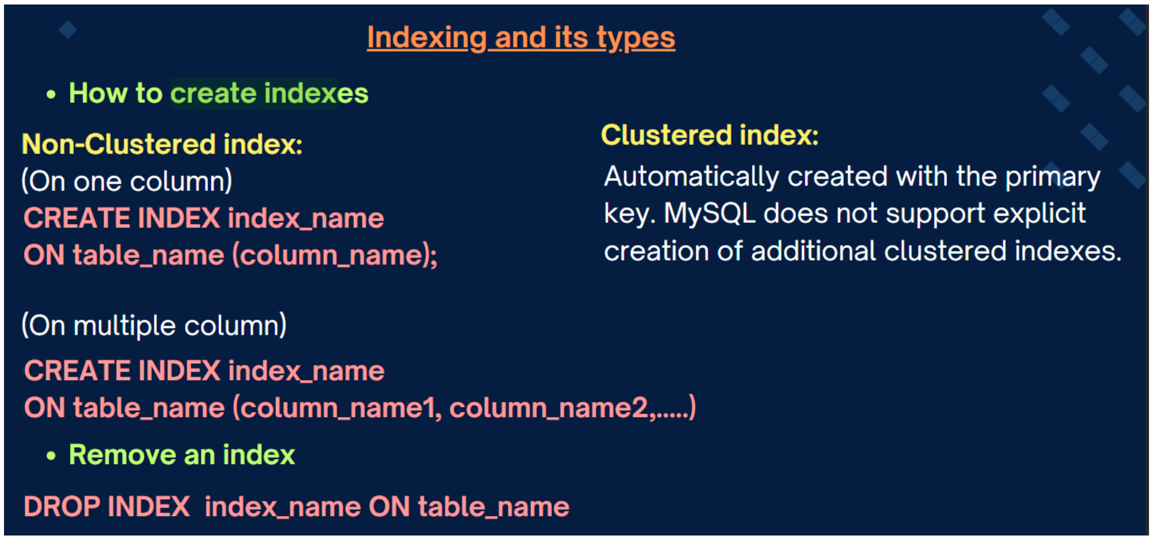
Indexing

- **What is it?**: A data structure (commonly B+ Tree) that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space.
- **How it Works**: An index stores Search Key values (copies of data from one or more columns) and Pointers (references to the location of the actual data rows on disk). When you query based on an indexed column, the database uses the index to quickly find the pointers to the relevant rows, avoiding a full table scan.
  - ***PDF Analogy (Page 7):*** Like using the index at the back of a textbook to find a topic quickly instead of reading every page.
- **Benefits**:
  - Speeds up SELECT queries (especially with WHERE clauses).
  - Can speed up ORDER BY and GROUP BY operations if the index matches the required order.
- **Costs**:
  - Requires additional disk space.
  - Slows down data modification operations (INSERT, UPDATE, DELETE) because the index also needs to be updated.

Indexing Types (Conceptual Overview)

- **Clustered Index**:
    - Determines the physical order of data rows in the table.
    - A table can have only *one* clustered index (usually on the primary key).
    - Very efficient for range queries on the clustered key.
    - *Primary Index (PDF Term)*: Often refers to an index on the primary key, which is typically clustered.
- **Non-Clustered Index (Secondary Index)**:
    - Index structure is separate from the physical data rows. The index contains pointers to the row locations.
    - A table can have multiple non-clustered indexes.
    - Good for searches on non-primary key columns.
    - *Secondary Index (PDF Term)*: An index that is not the primary way data is organized; typically non-clustered. Can be on key or non-key columns.
- **Sparse vs. Dense Indexing (Implementation Detail)**:
    - *Dense*: Index entry for *every* row in the table. Common for non-clustered/secondary indexes.
    - *Sparse*: Index entry only for *some* rows (e.g., first row in each data block). Requires data to be sorted (used with clustered/primary indexes). Uses less space than dense indexes.
- **Multilevel Index**: For very large tables, the index itself can become large. A multilevel index creates an index *on the index* (like chapters in a book index), reducing the search path. B+ trees naturally implement this.

Creating and Dropping Indexes (SQL)



Indexing and its types

- How to create indexes

Non-Clustered index:
(On one column)

```
CREATE INDEX index_name
ON table_name (column_name);
```

(On multiple column)

```
CREATE INDEX index_name
ON table_name (column_name1, column_name2,.....)
```

Clustered index:
Automatically created with the primary key. MySQL does not support explicit creation of additional clustered indexes.

- Remove an index

```
DROP INDEX  index_name ON table_name
```

B-Trees and B+ Trees (The Structure Behind Indexes)

- **Relevance**: Most relational databases use B+ Trees (an evolution of B-Trees) to implement indexes internally. Understanding their basic properties helps understand index performance.
- **Key Properties**:
  - **Balanced Trees**: All leaf nodes are at the same depth. This guarantees consistent search times.
  - **Logarithmic Performance**: Search, insert, delete operations are typically O(log n), which is very efficient for large datasets.
  - **B+ Tree Specifics**:
    - All data (or pointers to data) is stored *only* in the leaf nodes.
    - Internal nodes contain only keys used for navigation.
    - Leaf nodes are linked together in a doubly linked list. This makes range queries (e.g., WHERE age BETWEEN 20 AND 30) very efficient as the database can find the start of the range and then just follow the linked list.
- **Impact**: The balanced, logarithmic nature of B+ trees is why indexed lookups are much faster than full table scans (which are O(n)). The linked-list structure of B+ tree leaves makes indexed range scans efficient.
  - ***PDF Visual (Page 52):*** A table highlights key differences between B-Tree and B+ Tree structure and performance characteristics.