Introduction

Optimizing database queries is a key practice for improving performance, especially when dealing with large datasets. By employing strategies such as understanding execution plans, using indexes, and avoiding common anti-patterns, queries can be optimized for speed, efficiency, and resource utilization. This guide provides an in-depth look at how to optimize SQL queries by understanding performance, improving execution plans, and using indexes effectively.

Section 1: Understanding Performance and Optimisation

When optimizing a query, choosing the right approach can make a significant difference in performance. Consider the following example to find the names of departments with an average salary greater than 2500 in an HR employee schema:

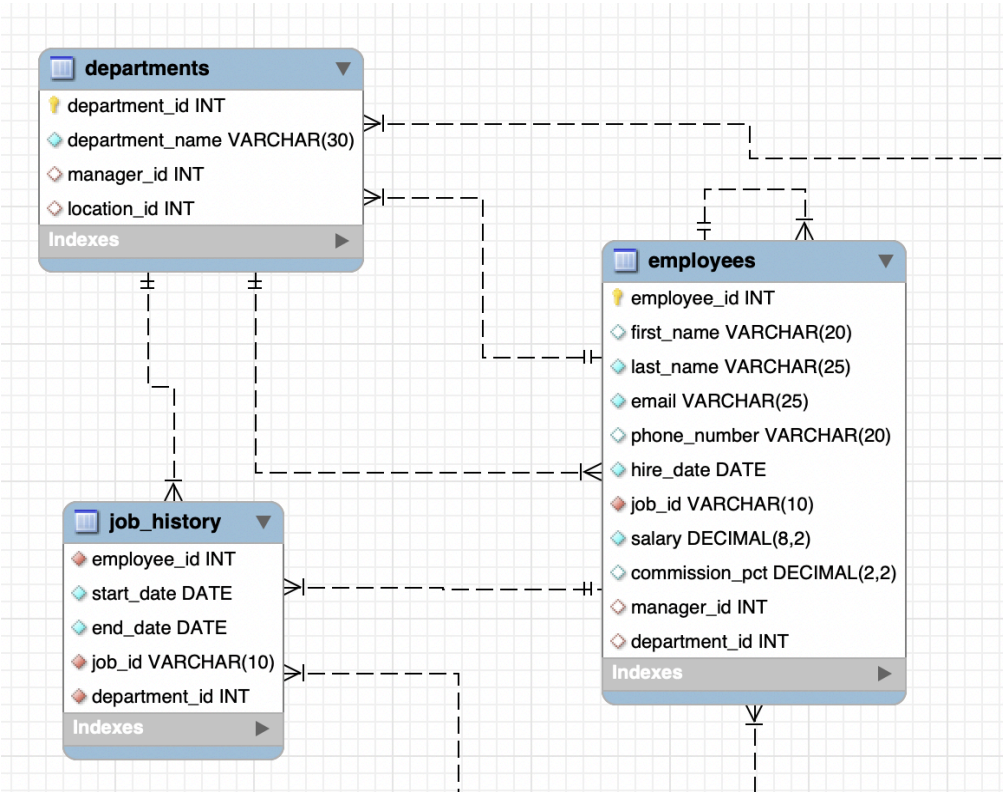
Query 1 (Join First, Then Group)

```
SELECT d.department_name
FROM department d
JOIN employee e ON d.department_id = e.department_id
GROUP BY d.department_id, d.department_name
HAVING AVG(e.salary) > 2500;
```

Query 2 (Group First, Then Join)

```
SELECT d.department_name
FROM (
   SELECT department_id, AVG(salary) as avg_salary
   FROM employee
   GROUP BY department_id
```

```
    HAVING AVG(salary) > 2500
) avg_dept
JOIN department d ON avg_dept.department_id = d.department_id;
```



Query 2 is typically more efficient for performance. By performing the aggregation first, you reduce the number of rows involved in the join, thus minimizing the computational cost of the join operation.

- **Aggregate First**: Performing aggregation before joining reduces the number of rows that need to be processed, improving performance.
- **Early Filtering**: Filtering the data using `HAVING` before the join helps eliminate non-relevant data early in the process, which optimizes the query.

Section 2: Query Scan and Execution Plan

Understanding Execution Plan

To optimize queries, understanding how databases execute them is essential. The `EXPLAIN` statement in MySQL and PostgreSQL provides insights into how a query will be executed, including details on row scans, index usage, and join order.

EXPLAIN Statement

The `EXPLAIN` command helps you visualize how the database plans to execute a query. It provides details about access type, indexes used, and the number of rows scanned.

```
EXPLAIN SELECT * FROM film WHERE title = 'ACADEMY DINOSAUR';
```

This command shows whether the database is performing a **full table scan** or using an index.

Example: Anti-Pattern - Join First, Then Group

Performing a join before aggregation can result in large intermediate datasets that slow down query execution.

Inefficient Query:

```
SELECT
   f.title,
   COUNT(r.rental_id) AS total_rentals
FROM
   film f
INNER JOIN
   inventory i ON f.film_id = i.film_id
INNER JOIN
   rental r ON i.inventory_id = r.inventory_id
GROUP BY
   f.title
ORDER BY
   total_rentals DESC;
```

Optimized Query: Group First, Then Join (Eager Aggregation)

```
WITH RentalCounts AS (
    SELECT
        i.film_id,
        COUNT(r.rental_id) AS film_rental_count
    FROM
        rental r
    INNER JOIN
        inventory i ON r.inventory_id = i.inventory_id
    GROUP BY
        i.film_id
)
SELECT
    f.title,
    rc.film_rental_count AS total_rentals
FROM
    film f
INNER JOIN
    RentalCounts rc ON f.film_id = rc.film_id
ORDER BY
    total_rentals DESC;
```

Validation: To compare performance, use `EXPLAIN ANALYZE` for both queries. The optimized query should show reduced row scans, especially during the join phase.

Section 3: Common Anti-patterns and Optimisation Strategies

Query Logic & Cardinality Anti-Patterns

1. The Correlated Subquery Trap

Principle Violated: Set-Based Processing
Correlated subqueries cause the inner query to execute once for each row of the outer query, which is inefficient.

Anti-pattern Code:

```
-- Bad: Executes inner query N times
SELECT *
FROM Customers C
WHERE C.last_order_date = (
   SELECT MAX(order_date) FROM Orders O
   WHERE O.customer_id = C.customer_id
);
```

Fix:

Replace the correlated subquery with a `JOIN` or a `ROW_NUMBER()` window function.

```
SELECT *
FROM Customers C
JOIN (
   SELECT customer_id, MAX(order_date) AS last_order_date
   FROM Orders
   GROUP BY customer_id
```

```
) O ON C.customer_id = O.customer_id AND C.last_order_date = O.last_order_date;
```

2. Join Before Aggregate (The Big Shuffle)

Principle Violated: Filter Early (Eager Aggregation)
Joining large tables before performing aggregation can lead to inefficient execution with large intermediate result sets.

Anti-pattern Code:

```
SELECT T1.customer_id, COUNT(*)

FROM Orders T1

JOIN Line_Items T2 ON T1.order_id = T2.order_id

GROUP BY T1.customer_id;
```

Fix:

Aggregate first, then join, to reduce the number of rows in the join.

3. Using Window Functions to Replace Self-Joins

Self-joins to calculate metrics like averages are inefficient.

Fix:

Use window functions to calculate metrics without needing a self-join.

```
SELECT employee_id, salary,

   AVG(salary) OVER (PARTITION BY department_id) AS avg_salary

FROM employees;
```

Section 4: Syntax & I/O Efficiency Anti-Patterns

These patterns directly prevent the database's optimizer from generating an efficient plan or force unnecessary data reads, wasting cloud resources.

1. `SELECT *` in Production Pipelines

Principle Violated: Column Pruning and I/O Efficiency

- **Anti-Pattern Code:** Using `SELECT *` in any query that is part of a data pipeline or view definition.
- **The Problem:** It forces the database to read and transmit **every single column** (including large, unused fields), which directly increases **I/O latency** and **cloud compute costs** (bytes scanned). It also makes pipelines susceptible to schema changes.
- **The Fix: Always explicitly list every column** needed. This is mandatory for performance and schema stability.

Section 5: Checking for Existence in Large Tables (The Anti-Join/Semi-Join)

When you are filtering a large table based on whether its key appears in another large table, `EXISTS` is often faster because it performs a **semi-join**.

Scenario Detail

When MySQL processes an `INNER JOIN`, it must find *all* matching rows, combine the columns, and potentially create a large intermediate result set. If the join is one-to-many (e.g., one customer has many orders), the `INNER JOIN` explodes the row count, forcing an expensive sort and distinct operation if you only wanted the unique customers.

The `EXISTS` operator, conversely, returns `TRUE` as soon as it finds **one** row that satisfies the correlated condition, and it **stops searching** that table immediately, dramatically reducing I/O and intermediate data processing.

Sakila Example: Finding Customers with Rentals

We want to find all customers who have ever rented a film.

Inefficient (JOIN with DISTINCT):

```
SELECT DISTINCT
C.customer_id,
```

```
C.first_name
FROM
customer c
INNER JOIN
rental r ON c.customer_id = r.customer_id;
```

Issue: The join generates one row for *every single rental* the customer ever made, and then the database must use `DISTINCT` to collapse these rows back to unique customers.

Efficient (`EXISTS` / Semi-Join):

```
SELECT
C.customer_id,
C.first_name
FROM customer c
WHERE EXISTS (

SELECT 1 -- Any constant is sufficient, as we don't care about the value
FROM rental r
WHERE r.customer_id = c.customer_id
);
```

Section 6: Filtering Based on Absence (Anti-Join)

This is the logical opposite of the first case. When you want to find records in one table that **do not exist** in another, `NOT EXISTS` is the clearest and often most optimized solution.

Scenario Detail

While you can achieve the same result using a `LEFT JOIN` and checking for `IS NULL`, `NOT EXISTS` expresses the intent directly. The optimizer often finds the `NOT EXISTS` construct easier to translate into an efficient **Anti-Join** physical operation than a complex join/filter sequence, leading to predictable performance.

Sakila Example: Finding Customers Who Have Never Rented

We want to find all customers who are in the system but have never placed a rental.

Inefficient (LEFT JOIN with IS NULL):

```
SELECT
C.customer_id,
c.first_name

FROM customer c

LEFT JOIN rental r
ON c.customer_id = r.customer_id

WHERE r.rental_id IS NULL; -- Requires joining the full tables first
```

Efficient (`NOT EXISTS` / Anti-Join):

```
SELECT
C.customer_id,
c.first_name

FROM customer c
WHERE

NOT EXISTS (
SELECT 1
FROM rental r
WHERE r.customer_id = c.customer_id    );
```

Benefit: This tells the database: "Find a customer, then check the `rental` table. If any matching rental exists, discard the customer and move to the next one."

Join Related Anti-patterns

An **Anti-Join** is a set operation used to find records in one table that **do not have a matching record** in another table.

Think of it as the SQL way of answering the question: "Show me everything from List A that is *not* in List B."

Anti-Join Example: Finding Unrented Films

Let's use the Sakila database tables:

1. **`film`** (List A): Contains all films ever made (`film_id`, `title`).

2. **`inventory`** (List B): Contains all copies of films currently in stock (`inventory_id`, `film_id`).

Goal: Find all films that currently have **no copies in stock** (i.e., films in the `film` table that are **not** present in the `inventory` table).

The SQL Anti-Join Pattern

The most common and optimized way to execute an Anti-Join in SQL is by combining a **`LEFT JOIN`** with a specific **`WHERE`** clause:

1. **`LEFT JOIN`:** Start with the main table (`film`) and bring in matching rows from the second table (`inventory`). If no match exists, the columns from the second table will be filled with `NULL`.
2. **`WHERE ... IS NULL`:** Filter the result to keep only the rows where the columns from the second table (`inventory`) are `NULL`. These are the rows where the match failed.

Example Query:

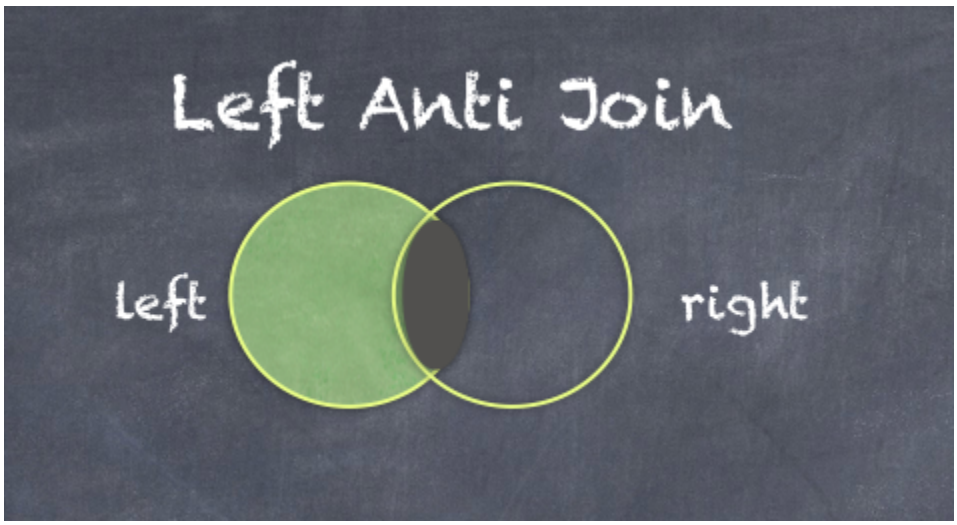
```
SELECT
F.film_id,
F.title
FROM film f
LEFT JOIN inventory i
ON f.film_id = i.film_id -- 1. Bring in matching inventory rows
WHERE i.inventory_id IS NULL; -- 2. Keep only the rows where the match failed (i.e., no inventory exists)
```

Why it's Efficient (The Optimizer's View)

When the database optimizer encounters this `LEFT JOIN/IS NULL` pattern, it often executes a physical operation called an **Anti-Join** (or sometimes an **Anti-Semi-Join**).

***Important** : Unlike a regular `INNER JOIN`, which must find and combine every single matching row, the Anti-Join query allows the database to check the `inventory` table and **stop immediately** if it finds even one matching row. If a*

match is found, the film is discarded, and the optimizer moves to the next film, saving significant I/O and processing time.



Excessive `ORDER BY` without `LIMIT`

Principle Violated: Local vs. Global Sorting

- **Anti-Pattern Code:** Applying an `ORDER BY` clause to a massive dataset when the order is not strictly required for the final output.
- **The Problem:** The `ORDER BY` operation forces a **global sort**, meaning all parallel worker nodes must shuffle and collect their data to a single coordinating node. This is a severe **bottleneck** and causes high latency and memory spill.
- **The Fix:** Only use `ORDER BY` when displaying the final, limited result set (`ORDER BY ... LIMIT 100`). For downstream processing, use efficient alternatives like **`SORT BY`** or **`CLUSTER BY`** (if available) for local sorting within partitions.

Section 7: Indexing Deep Dive

Basics of Indexing

Indexes significantly improve the speed of data retrieval operations by creating data structures that allow the database to quickly locate specific rows, bypassing full table scans.

- **Full Table Scan**: If there's no index, the database must read every row to find the desired data.
- **Indexes**: Improve retrieval speed by using data structures such as B-trees, which allow fast lookups.

Hash Indexes and Limitations

Hash indexes are effective for exact matches but have limitations:

- **No support for range queries**: Hash indexes cannot handle queries like `BETWEEN` or `>=`.
- **No support for partial matches**: Queries like `WHERE email LIKE 'john.doe@%'` won't benefit from hash indexes.

  Example of Hash Index Limitation:

  -- A hash index can't help with this range query

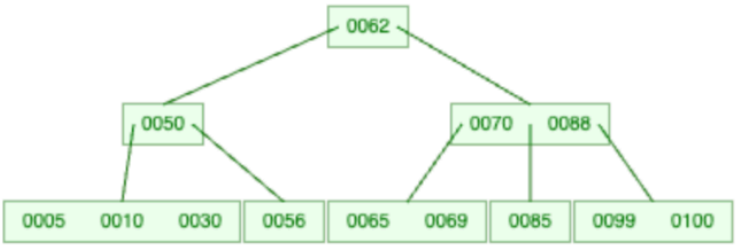
  SELECT * FROM employees WHERE salary BETWEEN 5000 AND 10000;

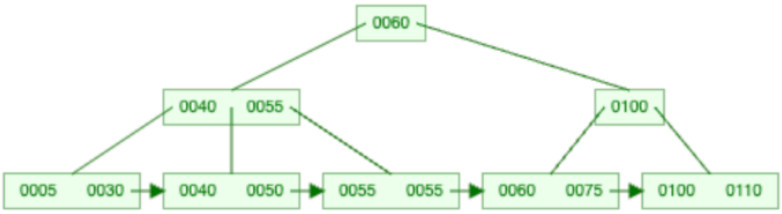
B-Trees and B+ Trees

- **B-Trees**: These balanced trees are good for fast lookups but can be inefficient for large datasets due to high disk access.
- **B+ Trees**: A variant of B-Trees used in databases. Only leaf nodes contain actual data, making them more efficient for lookups and range queries.

Example: B+ Tree Query Navigation

SELECT * FROM students WHERE id = 3;





B+ Trees allow fast lookups by navigating from the root to the leaf nodes, reducing the need for a full table scan.

Section 8: Clustered vs Non-Clustered Indexes

Clustered Index

A clustered index sorts the table's data physically based on the indexed column. Each table can have only one clustered index.

- **Example**: If you create a clustered index on the `id` column, the rows in the table will be ordered by `id`.

Non-Clustered Index

A non-clustered index stores pointers to the data rows rather than physically rearranging the data. A table can have multiple non-clustered indexes.

- **Example**: A non-clustered index on the `name` column points to the rows with corresponding `name` values but does not affect the physical order of the data in the table.

Section 9: How to Create Indexes

Creating an Index on a Single Column

CREATE INDEX idx_film_title ON film(title);

- **Before Indexing**: Without an index, querying the `title` column will result in a full table scan.

- **After Indexing**: With the index, the database can quickly locate titles without scanning the entire table.

Creating Composite Indexes

Composite indexes allow multiple columns to be indexed together, optimizing queries that filter on multiple columns.

CREATE INDEX idx_customer_date ON Orders(customer_id, order_date);

Important: The order of columns in a composite index matters. Queries filtering on the first column benefit from the index.

Section 10: Indexes on Multiple Columns

In MySQL, although usually one index is used, MySQL can sometimes perform **Index Merge** by combining multiple single-column indexes.

Consider this schema:

```
CREATE TABLE Orders (

    order_id BIGINT PRIMARY KEY,

    customer_id BIGINT,

    order_date DATE,

    amount DECIMAL

);
```

Query 1:

```
SELECT * FROM Orders WHERE customer_id = 101;
```

Query 2:

```
SELECT * FROM Orders WHERE customer_id = 101 AND order_date >=
'2025-01-01';
```

The index created on `customer_id` will help narrow down the searches in Query 2, but a scan on the filtered rows for `order_date` will still be necessary.

To further optimize this query, a **composite index** can be created on both `customer_id` and `order_date`:

```
CREATE INDEX idx_customer_date

ON Orders(customer_id, order_date);
```

This index allows the query to be answered directly from the index.

A **composite index** first groups by `customer_id`, and inside each `customer_id`, it sorts by `order_date`.

This means queries like:

```
SELECT * FROM Orders WHERE customer_id = 101 AND order_date >=
'2025-01-01';
```

can be answered efficiently by the composite index.

Analogy:
Imagine organizing a library where books are grouped by author, and then within each author's section, books are sorted by publication year. You can quickly find the section for an author and then scan the books chronologically, without having to go through the entire library.

Prefix Rule

The **prefix rule** determines which queries can efficiently use a composite index.

- A composite index on `(customer_id, order_date)` can handle:
    - `WHERE customer_id = 101`
    - `WHERE customer_id = 101 AND order_date >= '2025-01-01'`
- However, **querying only on `order_date`** will **not** use the index efficiently because the index starts with `customer_id`.

ORDER of Columns Matters

The **order of columns** in a composite index affects its usage by queries.

Example:

```
CREATE INDEX idx_customer_date ON Orders(customer_id, order_date);
```

This creates a **single B+ tree** where:

- Entries are first sorted by `customer_id`.
- Within each `customer_id`, they are sorted by `order_date`.

Query Scenarios:

Query 1:

```
WHERE customer_id = 101
```

This query uses the index effectively.

Query 2:

```
WHERE customer_id = 101 AND order_date >= '2025-01-01'
```

This query fully uses the index because both `customer_id` and `order_date` are part of the index.

Query 3:

```
WHERE order_date >= '2025-01-01'
```

This query **cannot** use the index effectively because `order_date` is not the first column, resulting in a **full table scan**.

Real-World Engineering Scenarios

Composite Index Use Cases:

Analytics Queries:

```
WHERE region = 'US' AND product_id = 42 AND sale_date >= '2025-01-01'
```

An index on `(region, product_id, sale_date)` would optimize the query.

E-commerce Search:

```
WHERE category = 'Shoes' AND brand = 'Nike' ORDER BY price
```

An index on `(category, brand, price)` would improve performance for filtering and sorting.

Logging Systems:

```
WHERE level = 'ERROR' AND created_at > NOW() - interval '1 day'
```

An index on `(level, created_at)` would optimize queries for logs based on severity and time.

Section 11: The Cons of Indexing

Extra Storage

Indexes require additional storage space, which can increase disk space requirements for the database. As more indexes are created, the storage footprint grows, especially for large tables.

- **Example**: A table with millions of rows and several indexes can double or triple the storage requirements.

Slower Writes

Indexes slow down write operations. Every time data is inserted, updated, or deleted, relevant indexes must also be updated, adding overhead to the write process.

- **Insertions**: Each new row requires updating the associated indexes, which can slow down insertion performance.
- **Updates/Deletes**: Similarly, updates or deletes require modifying the associated indexes, leading to further overhead.

Maintenance Overhead

Indexes need periodic maintenance to remain efficient. Over time, as data is modified, indexes can become fragmented, resulting in decreased performance.

- **Rebuilding Indexes**: Periodically rebuilding or reorganizing indexes is necessary to keep them optimized and prevent performance degradation.
- **Performance Degradation**: Fragmented indexes cause slower queries since the database needs to perform more work to access data.

Wrong Indexes Can Mislead

Even though indexes are created to improve performance, they might not always help if the query pattern doesn't align with the index design.

- **Non-matching Query Patterns**: If a query doesn't use the indexed columns correctly, the index might not be used, and the database might fall back on a full table scan.
- **Low Selectivity**: Indexes on columns with many duplicate values might not offer much performance improvement. For example, indexing a column with only two possible values (`true/false`) won't significantly speed up the query.

Section 12: Indexing on Strings

Indexing strings, such as email addresses, names, or product descriptions, is more complex than indexing numbers or dates. There are two main reasons for this:

- **String comparisons** are computationally heavier than numeric comparisons. For example, comparing two 50-character email addresses takes more effort than comparing two integers.

- **Storage consumption** is higher for strings since each index entry may store long text values, leading to greater storage requirements.

Example: Email Index

Imagine a customer table with 10 million records:

- 9 million emails end with `@gmail.com`.

- 1 million have other providers.

If we create an index directly on the email column, MySQL will maintain entries for all 10 million email addresses, many of which are long strings. This increases storage and makes comparisons slower.

Solution:
Instead of indexing the entire email, create an index on just the **prefix** of the email:

```
CREATE INDEX idx_email_prefix ON customers(email(7));
```

This stores only the first 7 characters of the email, which significantly reduces the index size and improves speed, while still distinguishing most users. If any collisions occur (emails starting with the same 7 characters), MySQL will check the full value to confirm.

Analogy:
It's like storing only the first 7 letters of every last name in your index card system. You save space, and it's still enough to quickly locate people.

Prefix Selectivity

The shorter the prefix, the higher the chance of **collisions**, meaning multiple strings could have the same prefix. The trade-off is that shorter prefixes might not be selective enough for large datasets.

- **Mathematics of Prefix Length**:
    - 2-letter prefix → $26^2 = 676$ possibilities.
    - 3-letter prefix → $26^3 = 17{,}576$ possibilities.
    - 4-letter prefix → $26^4 \approx 456{,}976$ possibilities.

As the prefix length increases, the index becomes more selective, especially for larger datasets. For large tables (hundreds of millions of rows), you may need 5–7 characters to make the index selective enough.

Full Text Search Problem

Consider queries like:

```
SELECT * FROM Products WHERE description LIKE '%running shoes%';
```

In this case, the string isn't just a **prefix match**; it's a **substring match** anywhere inside the text. A standard B+ Tree index can't help because it doesn't know where in the string to start looking.

Fulltext Indexes

To address the issue of searching for substrings or specific words within larger text fields, **FULLTEXT indexes** are used in MySQL. A FULLTEXT index breaks the text into tokens (words) and indexes these tokens individually.

- **FULLTEXT Indexing Process**:
    - It ignores common **stop words** (e.g., "a", "an", "the").
    - It applies **stemming**, treating similar words as equivalent (e.g., "run", "runs", "running" are all indexed as "run").

For example:

```
CREATE FULLTEXT INDEX idx_description ON Products(description);
```

You can then use the following query:

```
SELECT * FROM Products WHERE MATCH(description) AGAINST ('running shoes');
```

This allows the database to efficiently find matches for the search term, even if it appears anywhere in the text.

Use Case:
FULLTEXT indexes are ideal for product descriptions, blog posts, or documents where you want to search for words within the text, not just prefixes.

Conclusion

Key Insights

- Optimizing queries involves balancing readability and performance, using techniques like aggregation before join and filtering early.
- **Indexes** are essential for speeding up read operations but must be used wisely to avoid excessive storage and maintenance costs.

- **B+ Trees** are preferred for database indexing because they reduce disk access and provide efficient range queries.
- **Clustered and Non-Clustered Indexes** serve different needs, with clustered indexes physically organizing the data, while non-clustered indexes provide additional lookup structures.

Key Terms

- **Indexing**: A technique to improve query performance by creating data structures that allow fast lookups.
- **Clustered Index**: An index that sorts the table's data in the order of the index.
- **Non-Clustered Index**: An index that stores pointers to rows rather than reordering the data.
- **B+ Tree**: A balanced tree structure used in databases for indexing, with efficient leaf-node-based data retrieval.

Material:

Database Design 39 - Indexes (Clustered, Nonclustered, Composite Index)
How do indexes make databases read faster?
Why do databases store data in B+ trees?