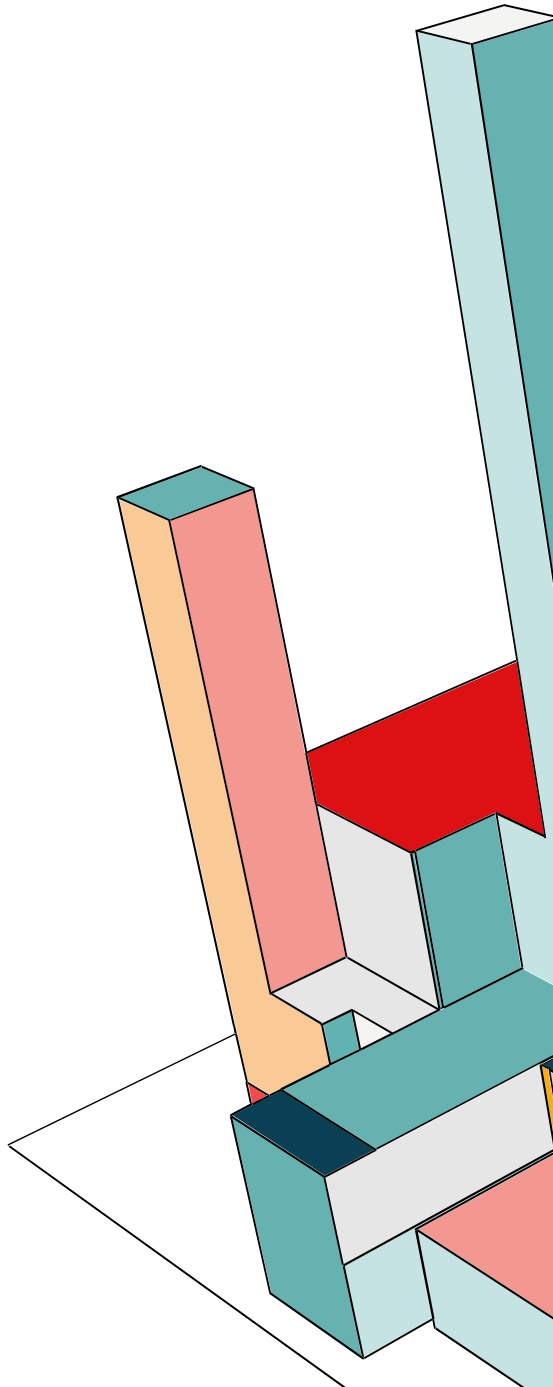
UNDERSTANDING INDEXING IN SQL

WHAT IS INDEXING IN SQL?

- Indexing is a database optimization technique used to speed up the retrieval of rows from a database table.
- It is similar to an index in a book, allowing you to quickly locate information without scanning every page.
- Purpose: To improve the performance of SELECT queries.



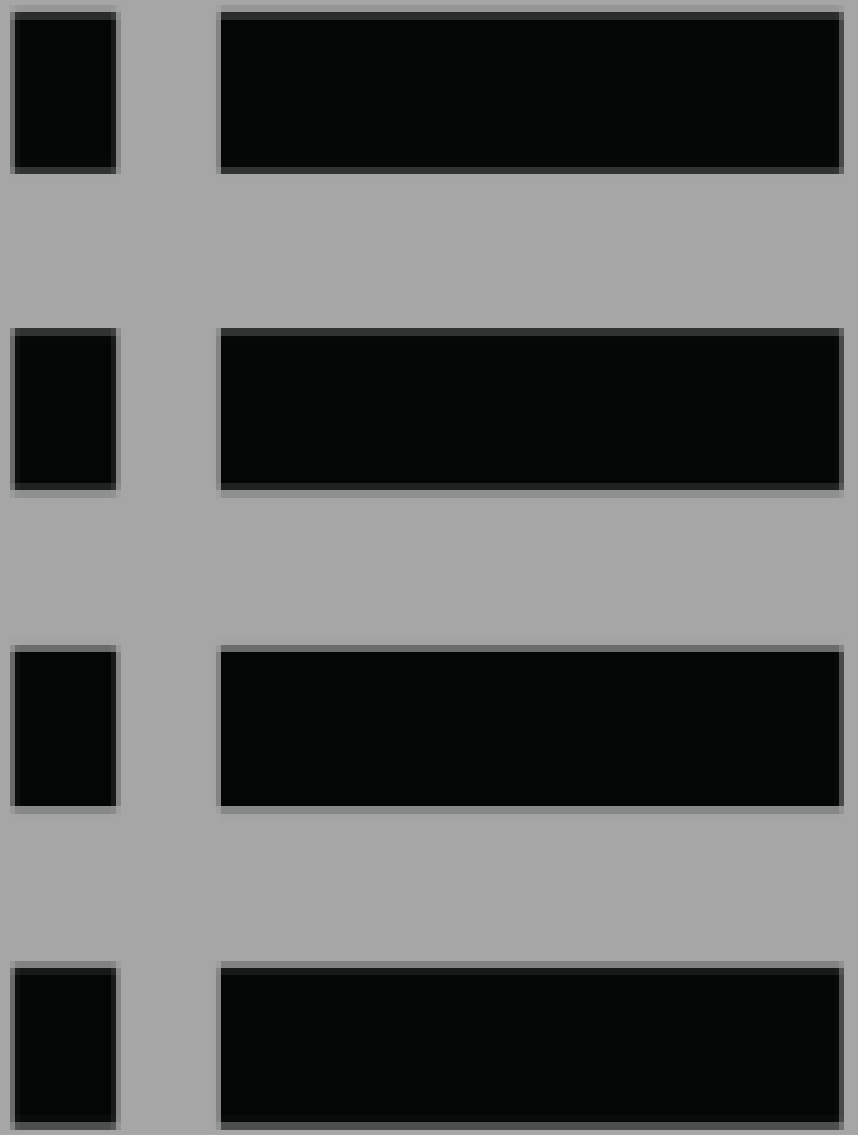
BENEFITS OF INDEXING

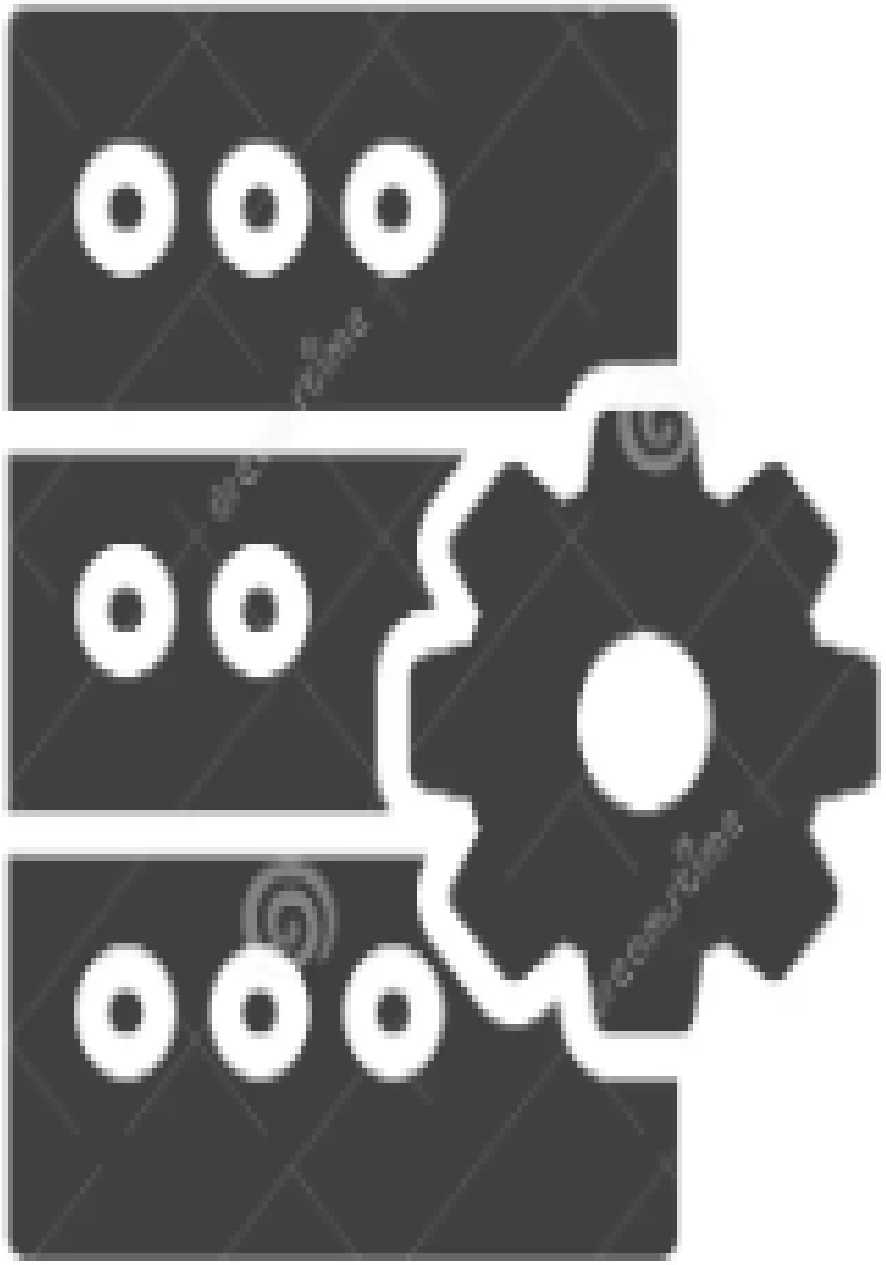
Faster Query Execution: Reduces the number of rows scanned.

Improved Search Performance: Especially for large datasets.

Efficient Sorting: Indexes help in sorting query results faster.

Reduced I/O Operations: Minimizes disk reads during queries.





HOW DO INDEXES WORK?

- Indexes create a sorted list of values from one or more columns.
- They store pointers (references) to the actual data rows in the table.
- When a query is executed, SQL uses the index to quickly locate the relevant rows, avoiding a full table scan. add more

REAL-WORLD EXAMPLE: QUERY OPTIMIZATION WITH INDEXING

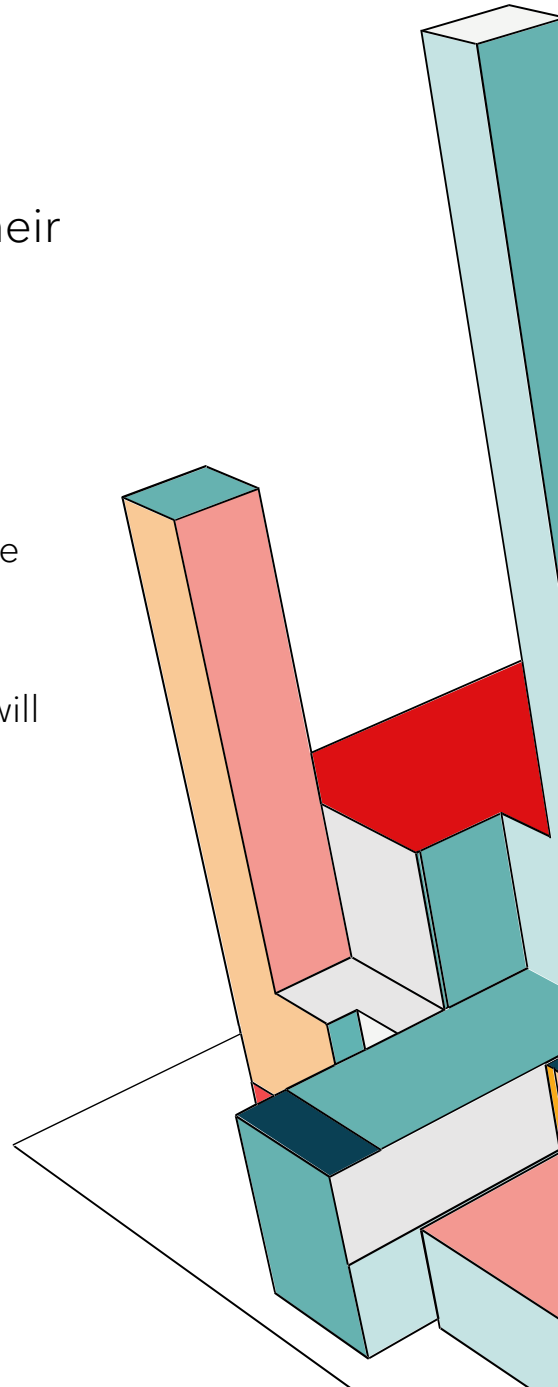
- Let's say a customer service representative needs to search for a specific customer by their email address. The database has a Customers table that contains millions of records. Here's what could happen without and with an index on the email column:

Without Indexing (Full Table Scan)

- Problem: The Customers table has millions of rows, and the representative needs to find a customer based on the email address.
- This process is slow and inefficient, especially when there are many queries running simultaneously. The system will take longer to respond, which frustrates customers and employees.
- Example Query:

```
SELECT * FROM Customers WHERE email = 'john.doe@example.com';
```

- Execution: This is called a **full table scan**. For large tables, this can be **extremely slow**, taking seconds or even minutes, depending on the table size.



REAL-WORLD EXAMPLE: QUERY OPTIMIZATION WITH INDEXING

- Let's say a customer service representative needs to search for a specific customer by their email address. The database has a Customers table that contains millions of records. Here's what could happen without and with an index on the email column:

With Indexing (Faster Lookup)

Solution: To solve this, an index is created on the email column of the Customers table.

SQL Command to Create Index:

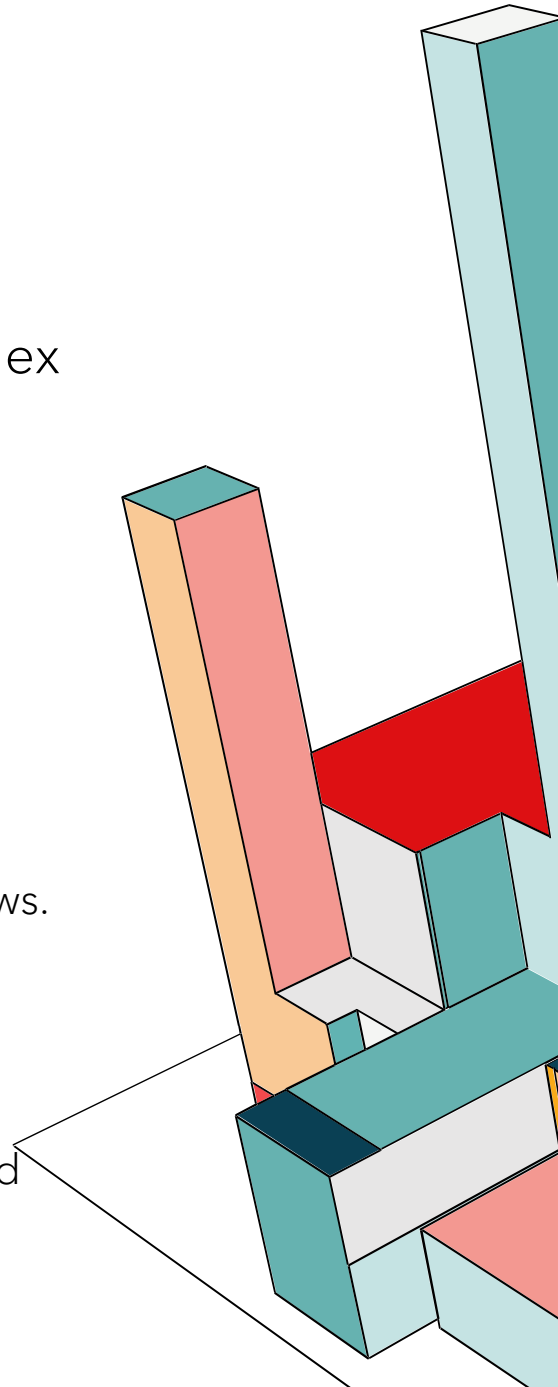
```
CREATE INDEX idx_email ON Customers (email);
```

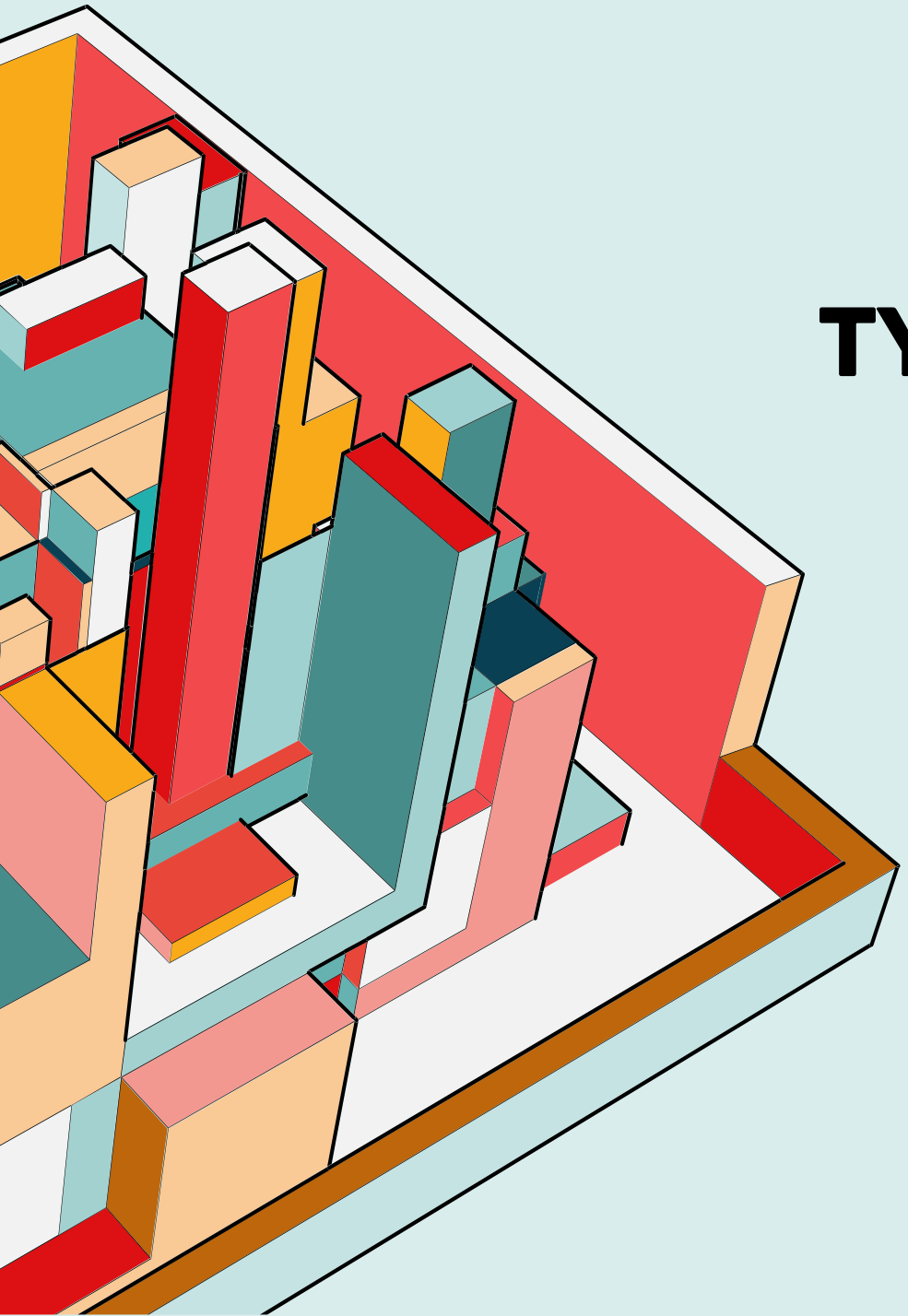
- The index allows SQL to **jump directly** to the location of the email address, skipping unnecessary rows.

Example Query:

```
SELECT * FROM Customers WHERE email = 'john.doe@example.com';
```

- **Execution:** The result is **significantly faster** query performance, as the index makes it possible to find the matching row **in constant time**.





TYPES OF INDEXES IN SQL

- Clustered Index
- Non-Clustered Index
- Unique Index
- Composite Index
- Full-Text Index
- Bitmap Index

CLUSTERED INDEX:

The data in the table is stored in the same order as the index.

When a clustered index is created, it reorganizes the table's data based on the index column(s). As a result, the rows are physically ordered on the disk.

Example: In a table of employee records, if a clustered index is created on the employee_id, the rows will be physically stored in ascending order of employee_id.

Clustered Index Example:

•Scenario: A table storing employee data, where the employee ID is a unique identifier for each employee.

•SYNTAX:

```
CREATE CLUSTERED INDEX idx_employee_id ON employees(employee_id);
```

Since the table is stored physically in the order of employee_id, range queries like BETWEEN or ORDER BY on employee_id will be faste

NON-CLUSTERED INDEX:

The data in the table is stored separately from the index. The table's physical data order remains unchanged, and the index is stored as a separate structure that points to the data.

A non-clustered index contains a pointer to the actual table rows, allowing SQL to quickly find the data without scanning the entire table.

Example: If a non-clustered index is created on the last_name column, the data in the employees table remains unordered, but the index stores a sorted list of last_name values and references the corresponding rows.

Non-Clustered Index Example:

Scenario: You frequently search for employees based on last_name or department_id, but don't need to sort by last_name or department_id.

```
CREATE NONCLUSTERED INDEX idx_lastname ON employees(last_name);
CREATE NONCLUSTERED INDEX idx_department ON employees(department_id);
```

•These non-clustered indexes will speed up lookups, but the table's data order won't change.

UNIQUE INDEX:

A unique index ensures that all values in the indexed column(s) are unique across the table. It prevents the insertion of duplicate values into the indexed column(s), thereby ensuring data integrity.

Example: If you want to ensure that email addresses in a users table are unique, you would create a unique index on the email column:

SYNTAX:

```
CREATE UNIQUE INDEX idx_email ON users(email);
```

COMPOSITE INDEX (MULTI-COLUMN INDEX)

A **composite index** (also called a **multi-column index**) is an index that is created on **two or more columns** of a table. It improves query performance for operations that filter or sort based on multiple columns.

Example: If you want to frequently search for employees by both department_id and last_name, you can create a composite index on both columns:

SYNTAX:

```
CREATE INDEX idx_dept_lastname ON employees(department_id, last_name);
```

FULL TEXT INDEX:

A **Full-Text Index** is designed to optimize searches in large text-based columns. It is primarily used to perform **text searches** that involve large amounts of unstructured or semi-structured data, such as articles, descriptions, or any column containing text.

Example: Searching for specific terms in a column containing long text such as product descriptions or blog posts.

SYNTAX:

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

This index will enable fast searches for terms like

```
SELECT * FROM products WHERE CONTAINS(description, 'keyword');.
```

BIT MAP INDEX:

A **Bitmap Index** is an index that uses a **bitmap (bit array)** to represent the presence or absence of values in a column, especially suited for columns that have a **low cardinality** (i.e., a small number of distinct values). Bitmap indexes are highly efficient for columns with **categorical data**.

Example: In a gender column with values Male and Female, two bitmaps will be created:

One bitmap for Male (1 for Male, 0 for others)

One bitmap for Female (1 for Female, 0 for others)

SYNTAX:

```
CREATE BITMAP INDEX idx_gender ON employees(gender);
```

This index would efficiently represent the gender column (with Male, Female, etc.), enabling fast Boolean operations like filtering AND/OR across different columns.

LIMITATION OF INDEXING

- Performance Overhead: Indexes slow down INSERT, UPDATE, and DELETE operations.
- Disk Space: Indexes consume additional disk space.
- Too Many Indexes: Over-indexing can cause performance degradation, as the database has to manage multiple indexes.

WHEN SHOULD YOU USE INDEXES IN SQL?

Indexes improve query performance by allowing the database to quickly locate data, but they should be used wisely.

- 1. High-Volume Lookup Operations
  - Use when searching for specific values in frequently queried columns (e.g., customer_id, email).
  - Speeds up SELECT queries with WHERE clauses.
- 2. Range Queries
  - Use for columns with numeric or date/time data when filtering by a range (e.g., BETWEEN, >, <).
  - Optimizes queries involving range searches, such as date or price ranges.
- 3. Sorting Operations
  - Use when your query involves ORDER BY to sort data.
  - Reduces the need to sort data manually, improving performance.
- 4. Join Operations
  - Use for columns used in JOIN conditions (e.g., primary key and foreign key relationships).
  - Speeds up JOIN operations by quickly finding matching rows.
- 5. DISTINCT Queries
  - Use when using DISTINCT to remove duplicate records.
  - Fastens queries that return unique values.
- 6. Aggregate Functions
  - Use for columns used in aggregate functions (e.g., SUM(), COUNT(), MAX()).
  - Improves the performance of aggregation queries.
- 7. Full-Text Search
  - Use when you need to perform advanced text-based searches in large text columns.
  - Speeds up searches for specific keywords or phrases.

THANK YOU