

Raushan Kumar

SQL

Indexing in SQL

- ✓ **Primary Index (Primary Key):**
- ✓ **Unique Index:**
- ✓ **Composite Index:**
- ✓ **Non-Clustered Index:**
- ✓ **Clustered Index:**
- ✓ **Full-Text Index:**
- ✓ **Spatial Index:**

Raushan Kumar

<https://www.linkedin.com/in/raushan-kumar-553154297/>

INDEXING IN SQL

What is Indexing ?

In SQL, **indexing** is a technique used to improve the speed of data retrieval operations on a database table. It works like the index in a book—where you can quickly locate the page number associated with a particular topic. Similarly, an index in SQL helps the database quickly locate the rows that match a query, reducing the need to scan the entire table.

An **index** is essentially a data structure (often a B-tree or hash table) that enables quick search operations on a database table. By creating an index on one or more columns of a table, you can speed up data retrieval, which is particularly beneficial when dealing with large datasets.

Why Do We Need Indexes?

Indexes are necessary for several reasons:

- **Faster Query Execution:** When querying large tables, indexes can drastically reduce the amount of data the database engine has to scan.
- **Efficient Searching:** Indexes allow SQL queries that involve WHERE, JOIN, ORDER BY, or GROUP BY operations to find the required data without scanning every row in the table.
- **Improved Sorting:** Indexes can help in faster sorting of data, especially for large result sets.

However, indexes do come with some trade-offs:

- **Disk Space:** Indexes consume extra disk space.
- **Performance Overhead on Data Modifications:** When data is inserted, updated, or deleted, the indexes also need to be updated, which may slow down write operations.

Types of Indexes in SQL

There are several types of indexes you can create in SQL, each serving different purposes.

1. Primary Index (Primary Key):

- Automatically created when a PRIMARY KEY constraint is defined.
- Ensures uniqueness and speeds up searches by the primary key.
- Example: id in a user table.

2. Unique Index:

- Ensures that all values in the indexed column(s) are unique.
- Example: email in a user table.

3. Composite Index:

- An index on multiple columns.
- Useful when queries filter using more than one column.
- Example: An index on both first_name and last_name if you often search for both.

4. Non-Clustered Index:

- Most common type of index.
- Created on columns that are frequently queried to speed up search.
- It doesn't change the physical order of rows in the table but creates a separate structure.

5. Clustered Index:

- A clustered index determines the physical order of rows in a table.
- A table can have only one clustered index, as the rows can only be sorted one way.
- The primary key is typically the clustered index.

6. Full-Text Index:

- Used for full-text searches on large text-based columns.
- Example: Searching for a word or phrase in an article or blog.

7. Spatial Index:

- Used for spatial data types, such as geometry or geographical data.
- Example: Indexing geographical coordinates for fast spatial queries.

How Indexing Works

Let's say you have a large database table with millions of records. Without an index, when you run a query like `SELECT * FROM users WHERE last_name = 'Smith';`, the database engine has to scan each and every row of the table to find the rows where `last_name` is 'Smith'. This process is called a **full table scan**.

However, with an index on the `last_name` column, the database engine can quickly locate the positions of the relevant rows without scanning the entire table. Instead, the index allows the system to directly jump to the locations that contain 'Smith' in the `last_name` column, which is far faster than scanning all rows.

Example: Indexing in SQL

Let's create a real-world example based on a **users** table.

1. Creating the Users Table

```
CREATE TABLE users (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  first_name VARCHAR(50),  
  last_name VARCHAR(50),  
  email VARCHAR(100) UNIQUE,  
  created_at DATETIME);
```

2. Inserting Sample Data

```
INSERT INTO users (first_name, last_name, email, created_at) VALUES  
( 'John', 'Doe', 'john.doe@example.com', '2025-02-01'),  
( 'Jane', 'Smith', 'jane.smith@example.com', '2025-01-15'),  
( 'Sam', 'Brown', 'sam.brown@example.com', '2025-02-10'),  
( 'Alice', 'Johnson', 'alice.johnson@example.com', '2025-01-20'),  
( 'Bob', 'Davis', 'bob.davis@example.com', '2025-02-03');
```

3. Querying Without an Index

If you execute the following query without any indexes on `last_name` or `email`:

```
SELECT * FROM users WHERE last_name = 'Smith';
```

The database engine will scan every row of the `users` table (a **full table scan**), checking if `last_name = 'Smith'` for each record. This can be time-consuming if the table has a large number of records.

4. Creating Indexes

1. Index on `last_name`:

```
CREATE INDEX idx_last_name ON users(last_name);
```

2. Unique Index on `email`:

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

Now, when you execute the same query:

```
SELECT * FROM users WHERE last_name = 'Smith';
```

The database will use the **index on `last_name`** to quickly find the matching rows without scanning the entire table.

5. Viewing Indexes

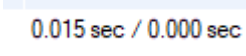
You can view the indexes on the `users` table with the following command:

```
SHOW INDEXES FROM users;
```

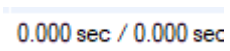
Performance Comparison

Let's analyze the **performance** before and after creating the indexes:

1. Without Index:

- For queries that involve last_name, MySQL needs to perform a **full table scan**.
- This can take a **significant amount of time** for large tables.
-  0.015 sec / 0.000 sec

2. With Index:

- After creating the index, MySQL uses the **index** to directly locate the rows with last_name = 'Smith'.
- This reduces query time from **linear** (full scan) to **logarithmic** (faster lookup) time.
-  0.000 sec / 0.000 sec

Considerations for Indexing

While indexes improve **read performance**, there are some trade-offs to consider:

1. **Write Performance:** When data is inserted, updated, or deleted, the index must also be updated. This can slow down these operations.
2. **Disk Space:** Indexes take up additional disk space. Depending on the number of indexes and the size of the table, this can be significant.
3. **Choosing the Right Columns:** Indexes should be created on columns that are frequently queried using WHERE, JOIN, or ORDER BY clauses.
4. **Over-Indexing:** Having too many indexes can negatively impact **insert, update, and delete performance** because MySQL has to maintain all the indexes during these operations.

Primary Index

A **Primary Index** is an index that is automatically created on the column defined as the **Primary Key** in a database table. The primary key is a constraint that uniquely identifies each record in the table, and the primary index enforces this uniqueness. The primary index in SQL ensures that there are no duplicate values in the primary key column and helps speed up query execution by enabling efficient lookups, especially for the primary key.

A primary index typically organizes the data **physically** on the disk. For this reason, it's also known as a **clustered index**. In fact, in many databases like MySQL or SQL Server, when you define a primary key, the primary key automatically creates a primary index (clustered index), and the data is stored on disk in the order of the primary key.

Key Points about Primary Index:

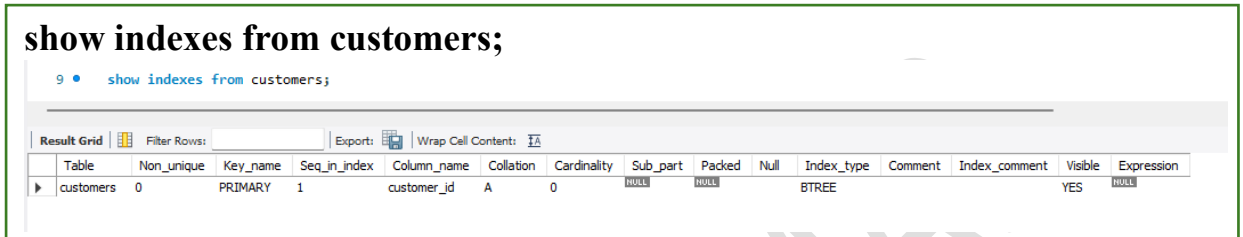
1. **Uniqueness:** The primary index ensures that the values in the column(s) it is created on are unique. This is because the column(s) that define the primary key cannot have NULL values.
2. **Single Index:** A table can only have **one primary index**, as the primary index defines the physical order of data in the table.
3. **Automatic Creation:** When you define a **primary key**, the database automatically creates a primary index on the column(s) involved.
4. **Clustered Nature:** The data in the table is organized (clustered) based on the primary key values, meaning the rows are stored in this order.

Example about Primary Index:

```
CREATE TABLE customers (  
    customer_id INT PRIMARY KEY,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    email VARCHAR(100),  
    created_at DATETIME);
```

In the example above:

- We create a table customers with five columns: customer_id, first_name, last_name, email, and created_at.
- The customer_id column is defined as the **primary key** for this table. As a result, SQL will automatically create a **primary index** on customer_id.
- **show indexes from customers;**



The screenshot shows the SQL query 'show indexes from customers;' and its results in a table format. The table has columns: Table, Non_unique, Key_name, Seq_in_index, Column_name, Collation, Cardinality, Sub_part, Packed, Null, Index_type, Comment, Index_comment, Visible, and Expression. The results show a primary index on the customer_id column.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
customers	0	PRIMARY	1	customer_id	A	0			YES	BTREE			YES	

Inserting Sample Data into the customers Table

INSERT INTO customers (customer_id, first_name, last_name, email, created_at)

VALUES

(5, 'John', 'Doe', 'john.doe@example.com', '2025-02-01'),

(3, 'Jane', 'Smith', 'jane.smith@example.com', '2025-01-15'),

(4, 'Sam', 'Brown', 'sam.brown@example.com', '2025-02-10'),

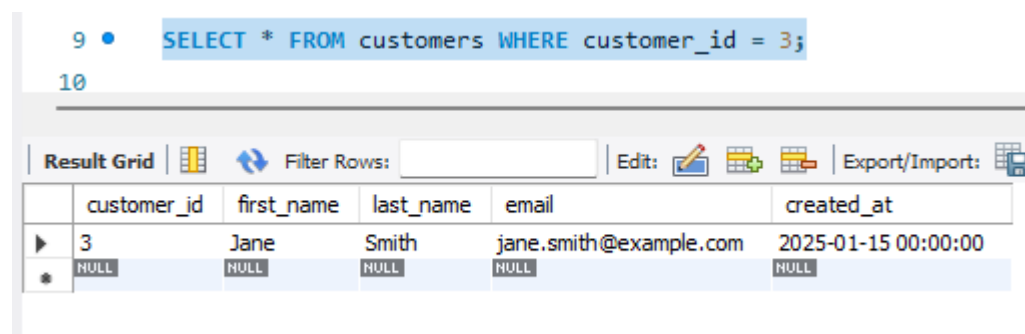
(1, 'Alice', 'Johnson', 'alice.johnson@example.com', '2025-01-20'),

(2, 'Bob', 'Davis', 'bob.davis@example.com', '2025-02-03');

Here, we're inserting five rows of customer data into the customers table. The customer_id column serves as the primary key, ensuring that no two rows have the same customer_id.

Querying the Table Using the Primary Index

```
SELECT * FROM customers WHERE customer_id = 3;
```



The screenshot shows a database query interface. At the top, the query `SELECT * FROM customers WHERE customer_id = 3;` is entered. Below the query, a 'Result Grid' is displayed. The grid has columns: `customer_id`, `first_name`, `last_name`, `email`, and `created_at`. The first row of data shows `customer_id` as 3, `first_name` as Jane, `last_name` as Smith, `email` as jane.smith@example.com, and `created_at` as 2025-01-15 00:00:00. A second row is visible with all values as NULL.

	customer_id	first_name	last_name	email	created_at
▶	3	Jane	Smith	jane.smith@example.com	2025-01-15 00:00:00
*	NULL	NULL	NULL	NULL	NULL

Primary Indexes and Data Organization

In databases like **MySQL** (InnoDB) or **SQL Server**, the data is physically sorted on disk based on the **primary key**. This means that the rows in the table are stored in the same order as the primary key values.

For instance, if you insert customers in the following order of `customer_id` values:

- 5, 2, 3, 1, 4

The database will store them in the following physical order on disk based on the primary index:

- 1, 2, 3, 4, 5

This means that all queries filtering by `customer_id` (such as `SELECT * FROM customers WHERE customer_id = 4`) will be faster, because the data is stored in a sorted order.

Unique Index

What is a Unique Index ?

A **Unique Index** in SQL is an index that ensures the values in the indexed column(s) are unique across all the rows in the table. It allows fast access to data while enforcing **uniqueness** on the indexed columns.

While a **primary index** automatically enforces uniqueness, a **unique index** is not limited to the primary key column but can be applied to any column or set of columns in a table.

Key characteristics of a Unique Index:

- **Uniqueness:** Ensures that all values in the indexed column(s) are unique, meaning no two rows can have the same value for the indexed column(s).
- **Allows NULLs:** Unlike primary indexes, a unique index can allow NULL values. However, depending on the database system, it may allow multiple NULL values in the indexed column(s).
- **Improved Query Performance:** It improves query performance by making it easier to find rows based on the indexed column(s).
- **Non-clustered Index:** A unique index is typically **non-clustered**, meaning it does not affect the physical storage order of the rows but instead creates a separate structure to speed up lookups.

How a Unique Index Works:

When you define a **unique index** on one or more columns, the database will create an internal structure (like a B-tree) that allows for efficient searches based on those columns. The index also ensures that no two rows in the table have the same value for the indexed column(s).

Syntax to Create a Unique Index:

```
CREATE UNIQUE INDEX index_name  
ON table_name (column1, column2, ...);
```

Alternatively, you can create a unique index automatically by using the UNIQUE constraint when defining a column:

```
CREATE TABLE table_name (  
    column1 INT,  
    column2 VARCHAR(255) UNIQUE  
);
```

Real-Time Example:

Let's use a real-world example to understand the application of unique indexes.

Scenario:

Suppose we have a table called **users** where we store the details of users who sign up for an application. The email column should be unique since no two users can have the same email address.

Table Structure:

- **user_id** (Primary Key)
- **email** (Unique)
- **first_name**
- **last_name**
- **signup_date**

1. Create the users Table with a Unique Index on the email Column:

```
CREATE TABLE users (  
    user_id INT PRIMARY KEY,  
    email VARCHAR(255) UNIQUE,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50), signup_date DATETIME);
```

- The **primary key** is set on the `user_id` column, and the **unique constraint** is set on the `email` column. This means the email must be unique across all rows in the table.
- A **unique index** will be created automatically on the `email` column to enforce this constraint.

2. Inserting Sample Data into the users Table:

```
INSERT INTO users (user_id, email, first_name, last_name, signup_date)
VALUES
```

```
(1, 'john.doe@example.com', 'John', 'Doe', '2025-02-01'),
(2, 'jane.smith@example.com', 'Jane', 'Smith', '2025-01-15'),
(3, 'sam.brown@example.com', 'Sam', 'Brown', '2025-02-10'),
(4, 'alice.johnson@example.com', 'Alice', 'Johnson', '2025-01-20');
```

Here, we have inserted four users into the table, each with a unique email address.

3. Attempting to Insert Duplicate Email Address (Should Fail):

```
INSERT INTO users (user_id, email, first_name, last_name, signup_date)
VALUES
```

```
(5, 'john.doe@example.com', 'Mike', 'White', '2025-02-03');
```

This **insert** will fail because the email `'john.doe@example.com'` already exists in the table, and the **unique index** enforces uniqueness on the `email` column. SQL will throw an error:

ERROR: Duplicate entry `'john.doe@example.com'` for key `'email'`

4. Creating a Unique Index Manually:

Although the `email` column already has a **unique constraint**, we can create a **unique index** explicitly:

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

This will create a **unique index** named `idx_email` on the `email` column, ensuring that all values in this column are unique, even if the **unique constraint** wasn't defined.

Benefits of Using Unique Indexes:

1. Data Integrity and Uniqueness:

- A **unique index** ensures that no two rows have the same value in the indexed column(s). In our example, it ensures that each email is unique across all users, preventing duplicate email addresses from being inserted.

2. Improved Query Performance:

- The unique index allows for faster lookups when querying by the indexed column. For example, querying a user by email will be more efficient:
- `SELECT * FROM users WHERE email = 'john.doe@example.com';`

3. Efficient Data Validation:

- Unique indexes help in **data validation**. They automatically ensure that duplicate values do not exist in columns that should hold unique data (e.g., email, username, etc.).

4. Preventing Data Anomalies:

- A unique index prevents **data anomalies** in the database, such as inserting duplicate records that could lead to inconsistencies and errors in reports and business logic.

Real-World Use Cases for Unique Indexes:

1. User Management Systems:

- For applications like CRM systems, online stores, or social media platforms, you often need to ensure that each **email address** is unique to avoid multiple users registering with the same email address.

2. Inventory Systems:

- In an **inventory management system**, product SKUs (Stock Keeping Units) need to be unique. You would create a unique index on the sku column to prevent duplicate product entries.

3. E-commerce Platforms:

- On e-commerce websites, the **order ID** or **transaction ID** should be unique to ensure each purchase is recorded accurately. A unique index can be used on these fields to maintain data integrity.

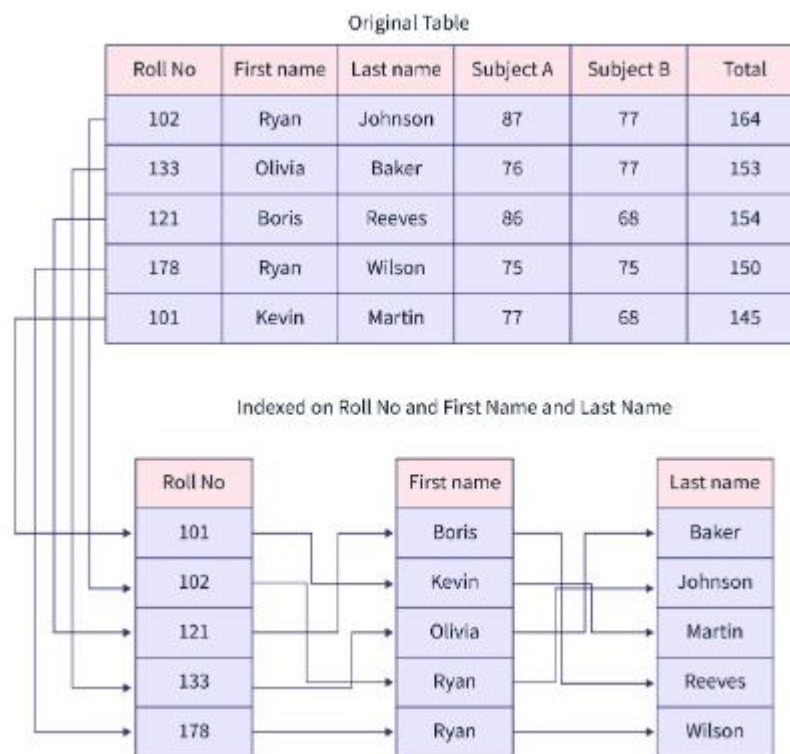
Primary Difference Between Primary Key and Unique Index:

- **Primary Key:**
 - A table can have **only one primary key**.
 - A primary key automatically creates a **clustered index**.
 - The primary key column(s) **cannot be NULL**.
- **Unique Index:**
 - A table can have **multiple unique indexes** (on different columns).
 - A unique index creates a **non-clustered index**.
 - The unique index allows **NULL** values, depending on the database system.

Composite Index

What is a Composite Index?

A **composite index** in SQL (also called a **multi-column index**) is an index that is created on two or more columns of a table. Unlike a single-column index, which is created on just one column, a **composite index** helps to improve the performance of queries that filter, sort, or join on multiple columns.



Why Use a Composite Index?

- **Query Optimization:** Composite indexes are particularly useful when queries involve **multiple columns** in the WHERE, ORDER BY, GROUP BY, or JOIN clauses.
- **Faster Searches:** When queries filter based on combinations of columns, composite indexes provide a way to speed up those searches by storing the combined values of those columns.
- **Reduced Disk I/O:** Using composite indexes can reduce the amount of data scanned and increase query performance.

How Composite Index Works:

A composite index stores a **sorted representation** of the values in the indexed columns. The database system uses this index to quickly locate rows in the table that match specific values in these columns.

Consider a table with three columns:

1. **first_name**
2. **last_name**
3. **birthdate**

If you create a composite index on these three columns, the index will store the combination of **first_name**, **last_name**, and **birthdate** in a sorted order. The database can use this index to quickly find records based on all or a subset of these columns.

Composite Index Syntax:

```
CREATE INDEX index_name  
ON table_name (column1, column2, column3, ...);
```

Alternatively, if you need to create the index when defining the table, you can specify it using the CREATE TABLE statement.

Real-Time Example:

Let's consider a real-world example of a **sales database** for an online store. The table records customer orders.

Table Structure:

We have a table called **orders** with the following columns:

- **order_id** (Primary Key)
- **customer_id**
- **order_date**
- **status**
- **amount**

Table Data:

order_id	customer_id	order_date	status	amount
1	101	2025-01-01	shipped	250.00
2	102	2025-01-01	pending	500.00
3	101	2025-01-02	shipped	100.00
4	103	2025-01-03	shipped	200.00
5	102	2025-01-04	shipped	150.00
6	101	2025-01-05	pending	350.00

Use Case for Composite Index:

Let's assume we frequently query the orders table to find orders placed by a customer (customer_id) in a particular **date range** (order_date), filtered by their **order status** (status).

For example, the query:

```
SELECT *  
FROM orders  
WHERE customer_id = 101  
      AND order_date BETWEEN '2025-01-01' AND '2025-01-05'  
      AND status = 'shipped';
```

This query involves three columns:

1. **customer_id**
2. **order_date**
3. **status**

Creating a Composite Index:

To optimize this query, we can create a composite index on the combination of customer_id, order_date, and status columns:

```
CREATE INDEX idx_customer_order_status  
ON orders (customer_id, order_date, status);
```

Why this composite index helps:

- The composite index allows the database to efficiently locate rows where the `customer_id` is 101, `order_date` is within the range 2025-01-01 to 2025-01-05, and the status is 'shipped'.
- Since the index is sorted by `customer_id`, then `order_date`, and then status, it speeds up this **multi-column query**.

How It Improves Query Performance:

Without an index, the database would need to perform a **full table scan** to find matching rows, which is slow for large tables. With the composite index, the database can use the index to **directly jump** to the relevant data, reducing the amount of data it needs to process.

Order of Columns in a Composite Index:

The order of the columns in the composite index matters. In the above example, the index is created on (`customer_id`, `order_date`, `status`).

- **Queries that filter on all three columns** can benefit from the index. For example, a query with `customer_id`, `order_date`, and `status` will be **optimized**.
- **Queries that filter on only the first column (e.g., `customer_id`)** can still benefit from the index because the index is sorted by `customer_id` first. However, **queries that only use `status` or `order_date` without `customer_id`** may not fully benefit from the index, depending on the database system.

For instance:

```
SELECT * FROM orders
```

```
WHERE order_date BETWEEN '2025-01-01' AND '2025-01-05';
```

This query does **not use the first column (`customer_id`)**. Therefore, it may not be fully optimized by the composite index, and the database might have to resort to scanning a larger part of the index.

When to Use Composite Indexes:

1. Queries with multiple filtering conditions:

- When your queries frequently use multiple columns in the WHERE clause, a composite index can significantly improve performance.

2. Complex joins:

- If you're joining multiple tables on multiple columns, a composite index on those columns can improve join performance. For example, if the orders table is being joined with a customers table based on customer_id, and the query also involves filtering on order_date and status, a composite index on (customer_id, order_date, status) can help.

3. Sorting and Grouping:

- Composite indexes can also speed up **ORDER BY** and **GROUP BY** operations. For instance, if your queries sort the results by customer_id and order_date, then a composite index on (customer_id, order_date) can help.

Pros and Cons of Composite Indexes:

Advantages:

1. Improved Query Performance:

- Composite indexes are very useful in improving the speed of complex queries that involve multiple columns in filtering, sorting, and joining.

2. Efficient Storage:

- By combining multiple indexes into a single composite index, you can save storage space compared to creating individual indexes for each column.

3. Reduced Query Time:

- With composite indexes, queries that would normally require scanning the entire table can be executed much faster.

Disadvantages:

1. Slower Inserts/Updates:

- Inserting or updating data in a table with a composite index can be slower since the database must maintain the index. Every time a row is added, deleted, or modified, the index also needs to be updated.

2. Index Size:

- Composite indexes can be large, especially if they involve multiple columns with many distinct values. Larger indexes can consume more storage space and require more memory.

3. Order of Columns Matters:

- If you often query using a subset of the columns in the composite index, the index may not be as effective unless the first columns of the index are used in the query's WHERE clause.

Additional Example:

For a **sales report** query that groups by status and sorts by order_date:

```
SELECT status, COUNT(*), SUM(amount) FROM orders
WHERE customer_id = 101
GROUP BY status
ORDER BY order_date;
```

In this case, creating a composite index on (customer_id, status, order_date) can improve the performance of both the WHERE, GROUP BY, and ORDER BY clauses.

```
CREATE INDEX idx_customer_status_order_date
ON orders (customer_id, status, order_date);
```

clustered vs non-clustered composite index

In SQL, a **composite index** can be either **clustered** or **non-clustered**, depending on how it is created and the underlying database system. Here's a breakdown of the two concepts:

1. Clustered Composite Index:

- A **clustered index** determines the physical order of the data rows in a table. When a table has a clustered index, the rows are stored on the disk in the same order as the index.
- A **composite clustered index** is created on multiple columns. It means the data in the table is sorted and stored on the disk according to the order of the columns in the composite index.
- A table can have only **one clustered index**, and if a composite index is defined as clustered, the data rows will be physically organized according to the composite index's key columns.

Example:

If you create a composite clustered index on (column1, column2) for a table, the data will be stored in the order of column1 first, then column2. The physical storage of the rows will reflect this sorting.

```
CREATE CLUSTERED INDEX idx_composite_clustered  
ON your_table (column1, column2);
```

2. Non-clustered Composite Index:

- A **non-clustered index** does not affect the physical order of the data rows in the table. Instead, the index is stored separately from the data rows, and it contains pointers (references) to the actual data rows.
- A **composite non-clustered index** is an index on multiple columns, but it does not alter the physical layout of the table. The data is stored in the original order, and the index just stores a mapping to those rows.

Example:

If you create a composite non-clustered index on (column1, column2), the index will store the values of those two columns, and the pointer will direct to the location of the actual row in the table.

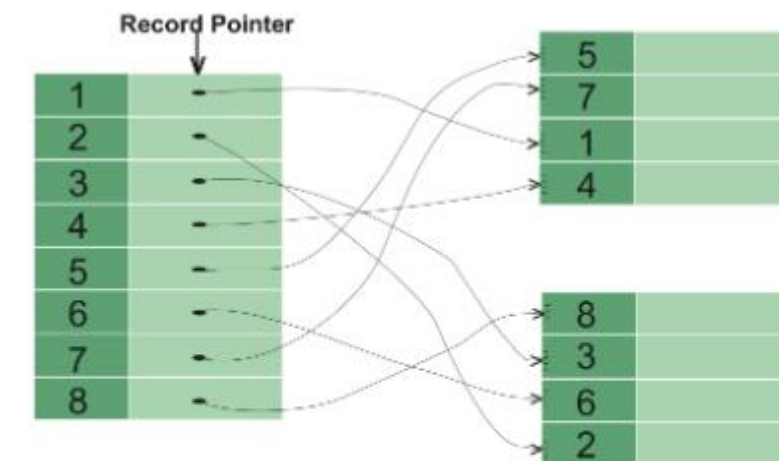
```
CREATE NONCLUSTERED INDEX idx_composite_non_clustered  
ON your_table (column1, column2);
```

Summary:

- **Clustered Composite Index:** The physical order of the rows in the table is affected, and there can only be one clustered index per table.
- **Non-clustered Composite Index:** The physical order of the rows is not affected, and multiple non-clustered indexes can be created on a table.

The choice between clustered and non-clustered composite indexes depends on the query patterns and how you want to optimize data retrieval for specific operations.

NON-CLUSTERED INDEX



EXAMPLE OF NON-CLUSTERED INDEX

A **non-clustered index** is an index that is stored separately from the actual data rows in a table. The index itself contains a sorted list of key values along with pointers (or references) to the actual data rows where the corresponding values exist.

Characteristics of Non-Clustered Index:

1. **Separate Structure:** The non-clustered index does not affect the physical order of the data in the table. It is stored in a separate structure that holds the indexed columns and pointers to the corresponding data rows.
2. **Multiple Indexes:** A table can have multiple non-clustered indexes. In fact, non-clustered indexes are often used for optimizing performance on different query patterns or columns that are frequently queried.
3. **Improved Query Performance:** Non-clustered indexes speed up query performance by allowing SQL Server (or any RDBMS) to locate data more efficiently.
4. **Use of Pointers:** The index stores pointers to the data rows, meaning the index structure itself does not contain the actual data.

5. **Support for Range Queries:** Non-clustered indexes are also very useful for range-based queries (e.g., retrieving rows within a specific date range).

How Non-Clustered Index Works:

- When a **non-clustered index** is created on a table, it creates a **logical structure** (like a B-tree) that holds the index key values and pointers to the actual rows.
- **Index Structure:**
 - **Leaf level:** Contains pointers to the actual data rows.
 - **Non-leaf level:** Contains pointers to the next level of the index until the root level is reached.

Syntax to Create Non-Clustered Index:

```
CREATE NONCLUSTERED INDEX index_name  
ON table_name (column1, column2, ...);
```

Real-Time Example:

Scenario:

You work at an e-commerce company, and you have a table that stores **orders** made by customers. The table contains the following columns:

order_id	customer_id	order_date	status	amount
1	101	2025-01-01	shipped	250.00
2	102	2025-01-01	pending	500.00
3	101	2025-01-02	shipped	100.00
4	103	2025-01-03	shipped	200.00
5	102	2025-01-04	shipped	150.00
6	101	2025-01-05	pending	350.00

Now, suppose you frequently query the table to find all orders placed by a specific `customer_id`. You might create a **non-clustered index** on the `customer_id` column to speed up this type of query.

Step 1: Create a Non-Clustered Index on `customer_id`

```
CREATE NONCLUSTERED INDEX idx_customer_id  
ON orders (customer_id);
```

This creates a non-clustered index on the `customer_id` column.

Step 2: How the Index Works

- **Index Structure:** A separate index structure is created with `customer_id` values and pointers to the corresponding rows in the orders table.

Example of Index Structure:

customer_id	Pointer to Row Location
101	Row 1, Row 3, Row 6
102	Row 2, Row 5
103	Row 4

In the index, each `customer_id` is mapped to the location of the data rows in the orders table. For instance, `customer_id` 101 corresponds to **Row 1**, **Row 3**, and **Row 6**.

Step 3: Query Using the Non-Clustered Index

Now, when you run a query like the following, the non-clustered index on `customer_id` will speed up the retrieval:

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

- Instead of scanning the entire orders table row by row, the database can use the index to quickly locate the relevant rows.
- It will use the pointers from the index to fetch rows with `customer_id = 101` from **Row 1**, **Row 3**, and **Row 6** directly.

Advantages of Non-Clustered Index:

1. **Faster Retrieval:** Queries that search for specific values or ranges in the indexed columns can be executed faster because the database engine can look up the index instead of scanning the entire table.
2. **Multiple Indexes:** Unlike the clustered index (which can only be one per table), a table can have multiple non-clustered indexes. For example, you can create indexes on `customer_id`, `order_date`, and `status` if needed.
3. **No Impact on Data Storage:** Since the non-clustered index is stored separately from the table data, the physical order of the rows in the table does not change.

Disadvantages of Non-Clustered Index:

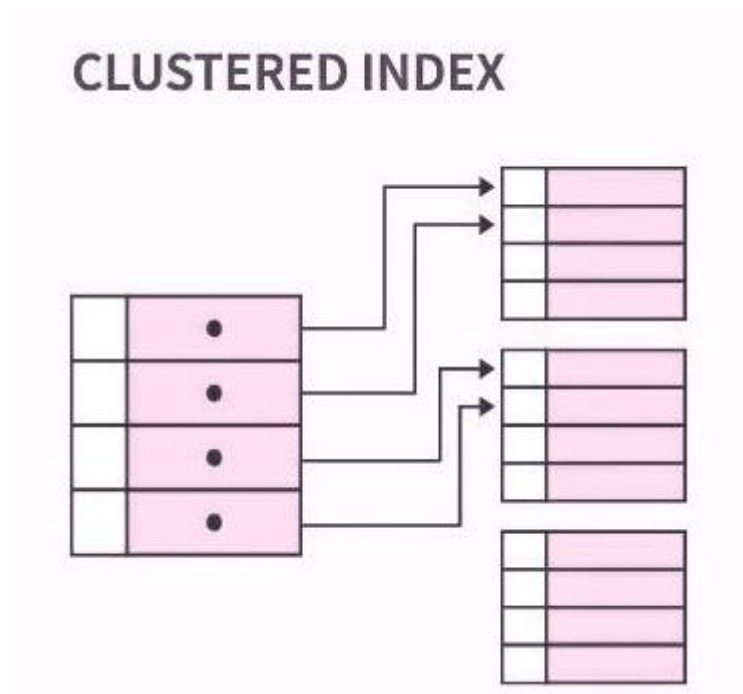
1. **Storage Overhead:** Non-clustered indexes require additional storage to maintain the index structure.
 2. **Slower Inserts/Updates/Deletes:** Any modification to the data (inserts, updates, deletes) will require the non-clustered index to be updated as well. This can lead to some performance overhead when dealing with large numbers of updates.
 3. **Index Maintenance:** Over time, non-clustered indexes might need to be rebuilt or reorganized to ensure optimal performance.
-

CLUSTERED INDEX

What is a Clustered Index?

A **clustered index** is an index where the order of the rows in the database table matches the order of the index. In other words, the table data is physically stored in the order of the clustered index. This means that a **clustered index** defines the physical sorting of the rows within the table, and there can be only one clustered index per table.

When a clustered index is created on a table, the actual table's rows are reordered to match the index. The data is stored in the **B-tree** structure, with the root and leaf nodes of the tree pointing to the actual rows in the table.



Key Characteristics of Clustered Index:

1. **Single Clustered Index per Table:** A table can have only one clustered index because the table's rows can only be sorted in one order.
2. **Physical Sorting:** The data rows are physically stored in the order of the clustered index. This means the data is sorted based on the indexed column.

3. **Automatically Created for Primary Key:** When a primary key constraint is defined on a table, a clustered index is automatically created on that column.
4. **Speed for Range Queries:** Clustered indexes are particularly useful when performing range queries because the data is already sorted in the index's order, allowing fast retrieval of continuous data ranges.

Syntax to Create a Clustered Index:

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

Real-Time Example:

Let's consider a real-world scenario where we have a **Customers** table in an e-commerce application, and we want to create a clustered index to speed up searches based on customer ID.

Step 1: Create the Customers Table

```
CREATE TABLE Customers (  
    customer_id INT PRIMARY KEY,    -- This automatically creates a  
    clustered index on customer_id.  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    email VARCHAR(100),  
    created_at DATETIME  
);
```

- The `customer_id` column is defined as the **PRIMARY KEY**, and SQL Server will automatically create a clustered index on this column, as the primary key is always associated with a clustered index.
- The **physical storage** of the rows in the table will be sorted based on the `customer_id` column.

Step 2: Insert Data into the Customers Table

```
INSERT INTO Customers (customer_id, first_name, last_name, email,  
created_at)
```

```
VALUES
```

```
(1, 'John', 'Doe', 'john.doe@example.com', '2025-01-01'),
```

```
(2, 'Jane', 'Smith', 'jane.smith@example.com', '2025-02-01'),
```

```
(3, 'Sam', 'Brown', 'sam.brown@example.com', '2025-01-15'),
```

```
(4, 'Lucy', 'Black', 'lucy.black@example.com', '2025-02-10');
```

At this point, the data in the Customers table will be physically sorted in the order of customer_id since we have a clustered index on that column.

Step 3: How Clustered Index Works

When you query the table, the database engine can quickly access the rows based on the clustered index.

For example, consider a query where you retrieve a customer based on the customer_id:

```
SELECT * FROM Customers
```

```
WHERE customer_id = 3;
```

Since the table rows are sorted by customer_id due to the clustered index, the database engine can quickly navigate to the **location of the row** with customer_id = 3 without scanning the entire table.

Step 4: Example of Range Query with Clustered Index

Clustered indexes are very efficient when querying for ranges of data, as the data is already sorted. For example, if you want to find all customers with `customer_id` between 2 and 4, the query will be executed efficiently using the clustered index:

```
SELECT * FROM Customers
```

```
WHERE customer_id BETWEEN 2 AND 4;
```

Since the rows are physically sorted by `customer_id`, the database can read the rows from the index structure in a sequential manner, which is much faster than scanning the entire table.

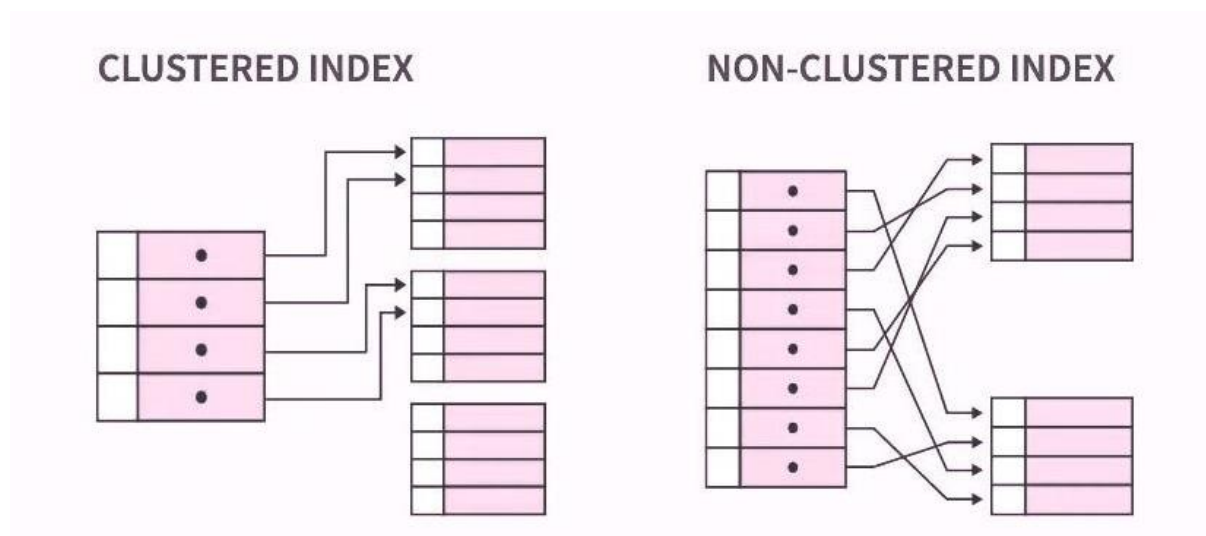
Advantages of Clustered Index:

1. **Efficient Range Queries:** Clustered indexes are excellent for queries that involve ranges of values, as the data is stored sequentially.
2. **Faster Reads for Sorted Data:** Since the data is stored in sorted order, queries that retrieve data in a sorted manner (e.g., `ORDER BY customer_id`) will be much faster.
3. **Smaller Index Size:** The clustered index does not require additional storage for the data since the actual data is the index.

Disadvantages of Clustered Index:

1. **Insert/Update Performance:** Inserting or updating data can be slower with clustered indexes because the data might need to be physically reordered to maintain the order of the index.
2. **Only One per Table:** Since the data is physically stored in the order of the clustered index, only one clustered index can be created on a table. You cannot create a second clustered index on a different column.
3. **Overhead on Deletions:** When rows are deleted, it might cause fragmentation in the clustered index, which could lead to slower query performance unless periodic maintenance like rebuilding the index is done.

Clustered Index vs. Non-Clustered Index:



1. Clustered Index:

- Defines the physical order of the rows in the table.
- Only one per table.
- Excellent for range queries or queries that involve sorting on the indexed column.

2. Non-Clustered Index:

- Does not define the physical order of rows.
- Can have multiple indexes on a table.
- Useful for improving performance on specific queries that do not involve range-based or ordered retrieval.

By: Raushan Kumar

Please follow for more such content:

<https://www.linkedin.com/in/raushan-kumar-553154297/>