Got it, Talha! Let’s dive deeper and **explore exactly how each index type works**, with detailed examples, before and after queries, **clustered vs non-clustered**, and much more. I'll make sure you have this topic on your fingertips for interview prep!

**🔍 How Indexes Work: Detailed Examples**

Let’s start with the basic concepts and work through each index type with actual queries to illustrate what happens **before** and **after** indexes are applied.

**1. Primary Index**

A **Primary Index** is automatically created on the **primary key**. The rows are physically stored in **order** based on the primary key value.

**Before Indexing:** You have a table like this:

CREATE TABLE Drivers (

driver\_id INT PRIMARY KEY, -- This will create the primary index

name VARCHAR(100)

);

Data in Drivers:

| **driver\_id** | **name** |
| --- | --- |
| 1 | Ahmed |
| 3 | Bilal |
| 2 | Talha |

**After Applying Primary Index:** The primary index **physically organizes the data** in the table. This means the data in the Drivers table is stored in the order of driver\_id:

SELECT \* FROM Drivers WHERE driver\_id = 2;

* **With Index**: The query will directly access the row with driver\_id = 2 in **O(1)** time (since it’s ordered).
* **Without Index**: It would need to do a **full table scan** to find the driver.

**2. Clustered Index**

A **Clustered Index** is a special type of index where the **data is physically stored in the order of the index**. There can only be **one clustered index per table** because data can be sorted in only one way.

**Before Applying Clustered Index**: Data might be stored in random order in the table.

CREATE TABLE Rides (

ride\_id INT PRIMARY KEY, -- Primary key = Clustered index

passenger\_id INT,

fare INT

);

| **ride\_id** | **passenger\_id** | **fare** |
| --- | --- | --- |
| 3 | P02 | 300 |
| 2 | P01 | 250 |
| 1 | P03 | 400 |

**After Clustered Index (Primary Key)**: The rows will be physically rearranged **based on the ride\_id** (because the primary key is clustered).

| **ride\_id** | **passenger\_id** | **fare** |
| --- | --- | --- |
| 1 | P03 | 400 |
| 2 | P01 | 250 |
| 3 | P02 | 300 |

SELECT \* FROM Rides WHERE ride\_id = 2;

* **With Clustered Index**: The query will fetch the row directly without scanning the table.
* **Without Clustered Index**: The query would scan all rows to find ride\_id = 2.

**Why only one clustered index?**

Because the data is physically ordered by the clustered index. The **primary key** is usually chosen as the clustered index.

**3. Non-Clustered Index**

A **Non-clustered Index** creates a separate structure (B+Tree or hash table) that contains pointers to the actual rows. Unlike clustered indexes, **data is not physically stored in the index order**.

**Before Applying Non-Clustered Index**: Data in Rides:

| **ride\_id** | **passenger\_id** | **fare** |
| --- | --- | --- |
| 3 | P02 | 300 |
| 2 | P01 | 250 |
| 1 | P03 | 400 |

**After Creating Non-Clustered Index**:

CREATE INDEX idx\_fare ON Rides(fare);

The **Non-clustered Index** stores **only fare values and pointers** to the actual rows.

| **fare** | **pointer to row (ride\_id)** |
| --- | --- |
| 250 | row 2 |
| 300 | row 1 |
| 400 | row 3 |

Now when you run a query like this:

SELECT \* FROM Rides WHERE fare = 300;

* **With Non-clustered Index**: The database will **lookup** the index (which is faster) to find the pointer, and then **directly access** the actual row.
* **Without Non-clustered Index**: It would scan every row in the Rides table to find fare = 300.

**4. Composite Index**

A **Composite Index** is an index on **multiple columns**. It’s useful when you frequently run queries involving multiple columns.

**Before Creating Composite Index**:

CREATE INDEX idx\_ride ON Rides(passenger\_id, fare);

Data in Rides:

| **ride\_id** | **passenger\_id** | **fare** |
| --- | --- | --- |
| 3 | P02 | 300 |
| 2 | P01 | 250 |
| 1 | P03 | 400 |

**After Composite Index**: Now, the database creates an index that sorts by both passenger\_id and fare.

| **passenger\_id** | **fare** | **pointer to row (ride\_id)** |
| --- | --- | --- |
| P01 | 250 | row 2 |
| P02 | 300 | row 3 |
| P03 | 400 | row 1 |

Now you can run queries efficiently that use both columns:

SELECT \* FROM Rides WHERE passenger\_id = 'P02' AND fare = 300;

* **With Composite Index**: The query can directly use the index and find the row quickly.
* **Without Composite Index**: The database would need to scan the table and use a **multiple condition filter**.

**Important:**

The **order of columns in the composite index** matters! The left-most column(s) must be used in the query for it to be efficient.

**5. Full-text Index**

A **Full-text Index** is used for **text searching** in large fields (like finding words in descriptions).

CREATE FULLTEXT INDEX idx\_comment ON Rides(comment);

When you do this:

SELECT \* FROM Rides WHERE MATCH(comment) AGAINST('fast');

* **With Full-text Index**: The database uses specialized algorithms to **quickly match** words or phrases in large text fields.
* **Without Full-text Index**: The database would have to scan the entire comment field for each row.

This is extremely **efficient** for text searches.

**6. Bitmap Index**

A **Bitmap Index** is useful for columns with low cardinality (few distinct values), such as gender, status, etc.

Example (assuming status has limited values like ‘active’, ‘inactive’):

CREATE BITMAP INDEX idx\_status ON Rides(status);

Here’s how it works:

* The bitmap index stores a **bitmap** for each distinct value in the status column.
* For example, if status = 'active' and status = 'inactive' have only a few distinct values, a bitmap index will be efficient in filtering rows quickly.

**7. Hash Index**

A **Hash Index** is useful for exact match lookups, i.e., **equality searches**.

For example, if you need to look up passengers by **unique ID**:

CREATE INDEX idx\_passenger\_id ON Rides(passenger\_id);

* **With Hash Index**: The database will quickly find the exact match for passenger\_id.
* **Without Hash Index**: The database would perform a **full scan** for every row.

**⚡ When NOT to Use Indexes**

* **Frequent writes/updates**: Indexes **slow down** insert/update/delete operations.
* **Small tables**: For small tables, full table scans are fast enough, so **no need** for an index.
* **Highly dynamic columns**: Don’t index columns that change frequently, as the index will require constant updates.

**🔄 Clustered vs Non-clustered Index**

* **Clustered Index**: Data is **physically sorted** in the table, and there can be only **one clustered index per table**.
* **Non-clustered Index**: It’s **separate** from the table and stores **pointers** to the actual data. You can have **multiple non-clustered indexes**.

Example:

-- Clustered: Primary key index, data is physically sorted

CREATE TABLE Riders (

rider\_id INT PRIMARY KEY, -- Clustered index

name VARCHAR(100)

);

For the **non-clustered** index, the database will keep a separate structure of pointers:

CREATE INDEX idx\_rider\_name ON Riders(name); -- Non-clustered index

**🔑 Interview Questions for Indexes**

1. What are indexes and how do they improve query performance?
2. Can a table have multiple clustered indexes?
3. When would you use a composite index?
4. How do clustered and non-clustered indexes differ?
5. What is a bitmap index and when would you use it?
6. How does a B-tree work in indexing?
7. What’s the difference between a full-text index and a regular index?
8. What is the impact of indexes on write performance