Agenda

- Introduction to Indexing
- How Indexes Work
- Indexes and Range Queries
    - Data structures used for indexing
- Cons of Indexes
- Indexes on Multiple Columns
- Indexing on Strings
- How to create index

Introduction to Indexing

Hello Everyone

Till now, we had been discussing majorly about how to write SQL queries to fetch data we want to fetch. While discussing those queries, we also often wrote pseudocode talking about how at a higher level that query might work behind the scenes.

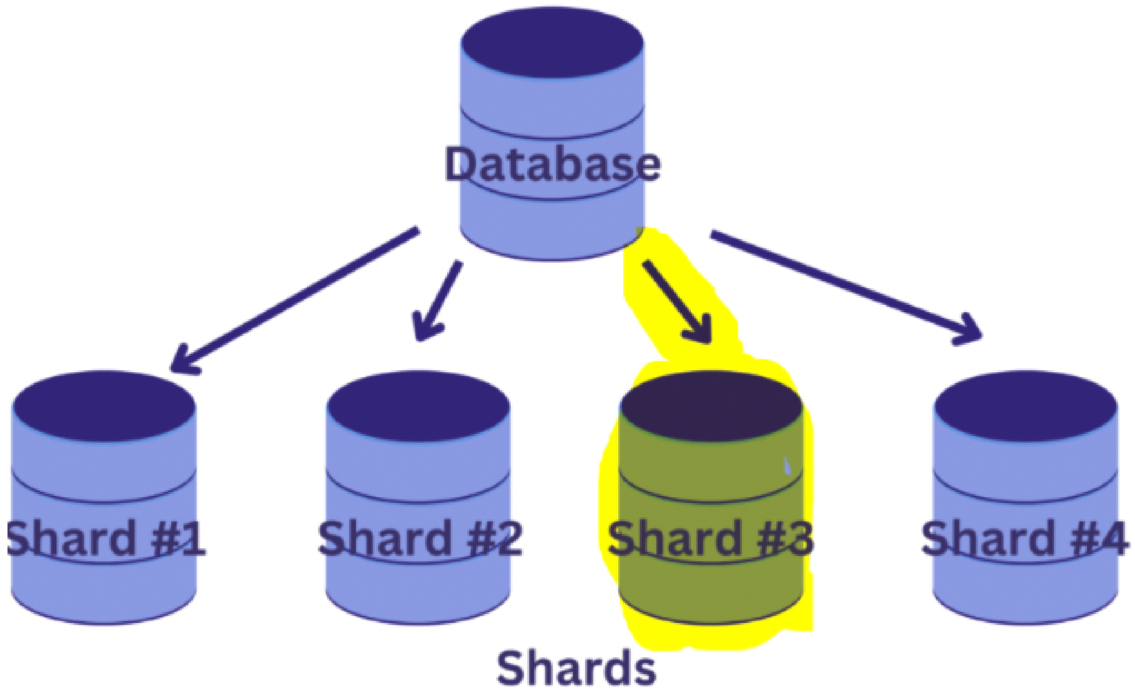
Let us go back to that pseudocode. What do you think are some of the problems you see a user of DB will face if the DB really worked exactly how the pseudocode mentioned it worked?

Correct! In the pseudocode we had, for loops iterated over each row of the database to retrieve the desired rows. This resulted in a minimum time complexity of O(N) for every query. When joins or other operations are involved, the complexity further increases.

Adding to this, in which hardware medium is the data stored in a database?

Yes. A database stores its data in disk. Now, one of the biggest problems with disk is that accessing data from disk is very slow. Much slower than accessing data from RAM. For reference, read https://gist.github.com/jboner/2841832 Reading data from disk is 80x slower than reading from RAM! Let's talk about how data is fetched from the disk. We all know that on disk DB stores data of each row one after other. When data is fetched from disk, OS fetches data in forms of blocks. That means, it reads not just the location that you want to read, but also locations nearby.

> *Inside the database, data is organised in memory blocks known as shards.*



First OS fetches data from disk to memory, then CPU reads from memory. Now imagine a table with 100 M rows and you have to first get the data for each row from disk into RAM, then read it. It will be very slow. Imagine you have a query like:

```
select * from students where id = 100;
```

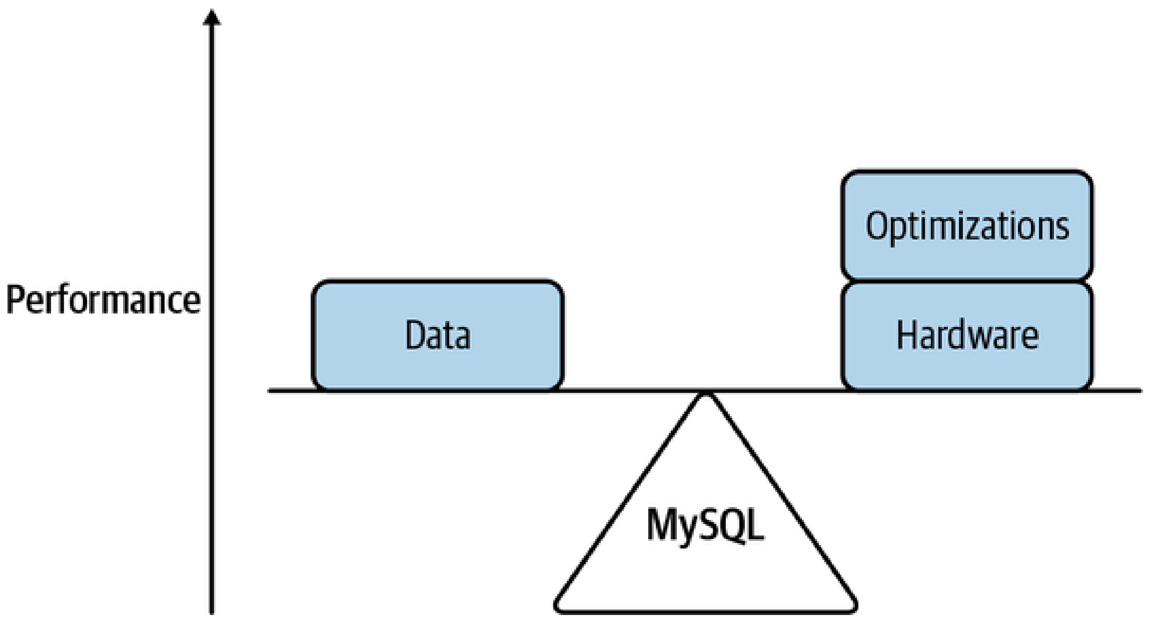
To execute above, you will have to go through literally each row on the disk, and access even the **memory blocks/shards** where this row doesn't exist. Don't you think this is a massive issue and can lead to performance problems?

To understand this better, let's take an example of a book. Imagine a big book covering a lot of topics. Now, if you want to find a particular topic in the book, what will you do? Will you start reading the book from the first page? No, right? You will go to the index of the book, find the page number of the topic you want to read, and then go to that page. This is exactly what indexing is. As index of a book helps go to the correct page of the book fast, the index of a database helps go to the correct block of the disk fast.

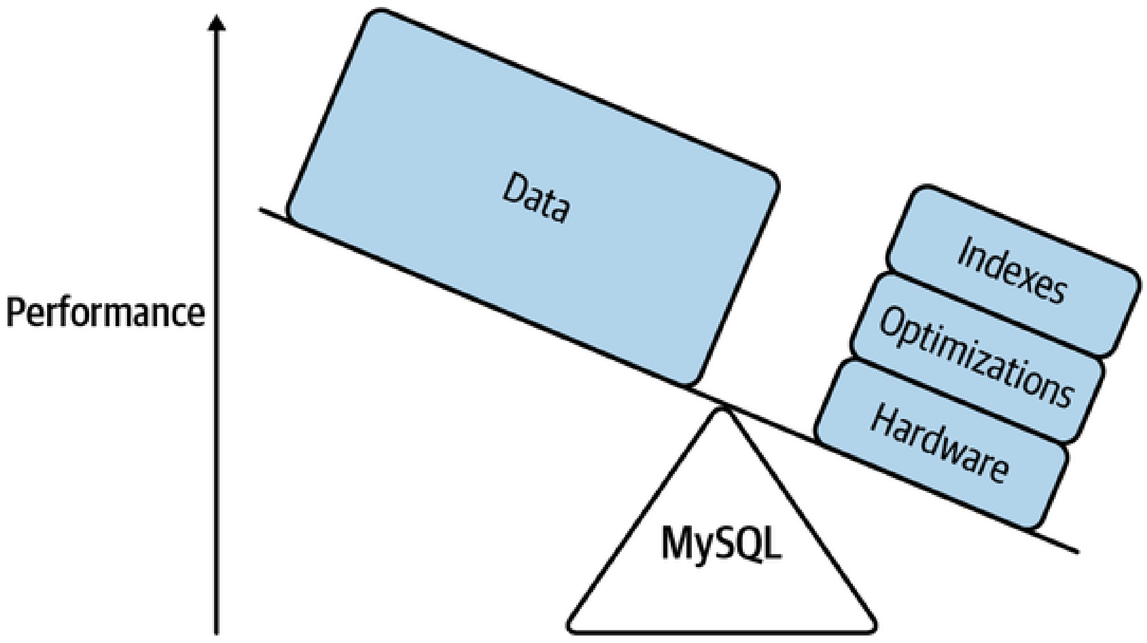
Now this is a very important line. Many people say that an index sorts a table. Nope. It has nothing to do with sorting. We will go over this a bit later in today's class. The major problem statement that indexes solve is to reduce the number of disk block accesses to be done. By preventing wastefull disk block accesses, indexes are able to increase performance of queries.

How is performance related with Inexing:

`Performance of SQL queries with powerful hardware and good optimizations but without indexing:`



`Performance of SQL queries with indexing:`



> *Pic credit: Efficient MySQL Performance: By Daniel Nichter*

MySQL leverages hardware, **optimizations**, and **indexes** to achieve performance when accessing data. Hardware is an obvious leverage because MySQL runs on hardware: the **faster the hardware, the better the performance**. `Less obvious and perhaps more surprising is that hardware provides the least leverage`. Will explain why in a moment. Optimizations refer to the numerous techniques, algorithms, and data structures that enable MySQL to utilize hardware efficiently. **Optimizations** `bring the power of hardware into focus`. And focus is the difference between a light bulb and a laser. Consequently, optimizations provide more leverage than hardware. If databases were small, hardware and optimizations would be sufficient. But increasing data size deleverages the benefits of hardware and optimizations. **Without indexes, performance is severely limited.**

> ***Note: Indexes provide the most and the best leverage. They are required for any nontrivial amount of data.***

How Indexes Work

While we have talked about the problem statement that indexes help solve, let's talk about how indexes work behind the scenes to optimize the queries. Let's try to build indexes ourselves. Let's imagine a huge table with 100s of millions of rows in table spread across 100s of **disk blocks also known as Shards.** We have a query like:

```
select * from students where id = 100;
```

We want to somehow avoid going to any of the disk block that is definitely not going to have the student with id 100. We need something that can help me directly know that hey, **the row with id 100** is present in **this block**. Are you familiar with a data structure that can be stored in memory and can quickly provide the block information for each ID? Key value pairs?

Correct. A map or a hashtable, whatever you call it can help us. If we maintain a **hashmap** where **key** is the id of the student and **value** is the **disk block** where the **row containing that ID is present**, is it going to solve our problem? Yes! That will help. Now, we can directly go to the block where the row is present and fetch the row. **This is exactly how indexes work.** They use some other data structure, which we will come to later.

Here we had queries on id. An important thing about `id` is that id is?

Yes. ID is unique. Will the same approach work if the column on which we are querying may have duplicates? Like multiple rows with same value of that column? Let's see. Let's imagine we have an SQL query as follows:

```
select * from students where name = 'Rahul';
```

How will you modify your map to be able to accomodate multiple rows with name 'Rahul'?

> *NOTE: We can maintain a `list<blocks>` for each key.*

What if we modify our map a bit. Now our **keys will be String (name)** and **values will be a list of blocks** that contain that name. Now, for a query, we will first go to the list of blocks for that name, and then go to each block and fetch the rows. **This way as well, have we avoided fetching the blocks from the disk that were useless?** Yes! Again, this will ensure our performance speeds up.

Indexes and Range Queries

So, is this all that is there about indexes? Are they that simple? Well, no. Are the SQL queries you write always like `x = y`? What other kind of queries you often have to do in DB?

> *Range queries?*

If a HashMap is how an index works, do you think it will be able to take care of range queries? Let's say we have a query like:

```
select * from students where psp between 40.1 and 90.1;
```

Will you be able to use hashmap to get the blocks that contain these students? Nope. A hashmap allows you to get a value in O(1) but if you have to check all the values in the range, you will have to check them 1 by 1 and potentially it will take O(N) time.

Even if we run a loop for a certian range then also we will not be able to get all the numbers since there can be many decimal numbers as well. Ex: 40.3, 40.32, 40.35.

`Since, data is not sorted in case of hashmaps hence we have to exclusively check for every number in range query which in turn makes overall performance slow.`

What are other issues?

- For finding k-th minimum element and supporting updates (Hashmap would be like bruteforce because it does not keep elements sorted, so we need something like Balanced binary search tree)
- Data structure for finding minimum element in any range of an array with updates (Once again hashmap is just too slow for this, we need something like segment tree)

So how will we solve it. Is there any other type of Map you know? Something that allow you to iterate over the values in a sorted way?

> *NOTE: Hint: TreeMap?*

Correct. There is another type of Map called TreeMap.

TreeMap

Let's in brief talk about the working of a TreeMap. For more detailed discussion, revise your DSA classes. A TreeMap uses a Balanced Binary Search Tree (often AVL Tree or Red Black Tree) to store data. Here, each node contains data and the pointers to left and right node.

> *NOTE: Link for TreeMap: https://www.scaler.com/topics/treemap-in-java/*

Now, how will a TreeMap help us in our case? A TreeMap allows us to get the node we are trying to query in O(log N). From there, we can move to the next biggest value in O(log N). **Thus, queries on range can also be solved.**

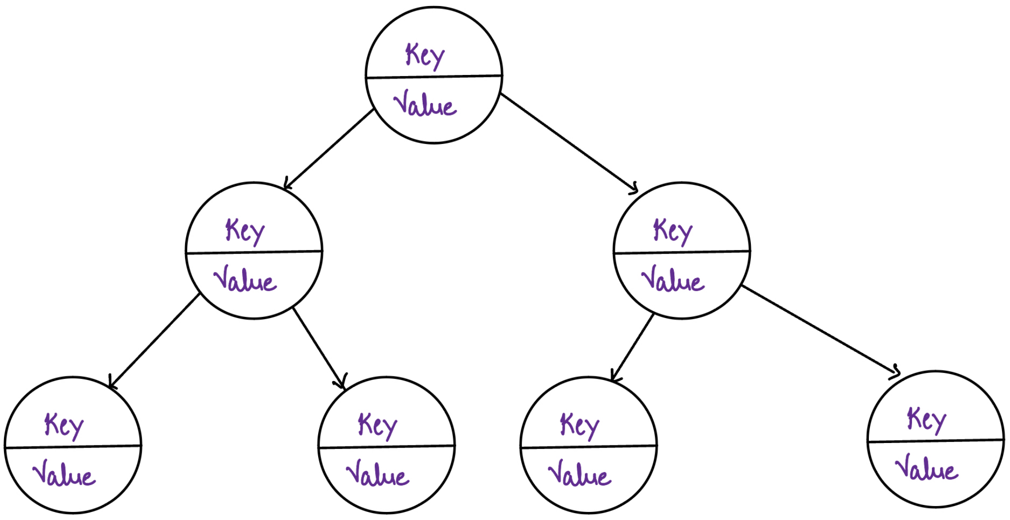
Internal Working of TreeMap in Java:

TreeMap internally uses a Red-Black tree, a **self-balancing binary search tree** containing an extra bit for the color (either red or black). The sole purpose of the colors is to make sure at every step of insertion and removal that the tree remains balanced.

In case of TreeMap:

- **Each Nod**e contains a **Key-Value** pair storing key and their **corresponding Memory Address** as **value**. Along witht that reference to left node and right node.
- Each Node have 2 children.

`Diagram:`



> *Activity: Can we further reduce the complexity of this tree? What is complexity of a TreeMap: log(Height). Can we reduce complexity further by reducing the height of Tree? We will discuss it further in B+Trees.*

B and B+ Trees

Databases also use a Tree like data structure to store indexes. But they don't use a TreeMap. They use a B Tree or a B+ Tree. Here, each node can have multiple children. This helps further reduce the height of the tree ultimately reducing the time complexity, making queries faster.

Properties of B-tree

Following are some of the properties of B-tree in DBMS:

- A non-leaf node's number of keys is one less than the number of its children.
- The number of keys in the root ranges from one to (m-1) maximum. Therefore, root has a minimum of two and a maximum of m children.
- The keys range from min([m/2]-1) to max(m-1) for all nodes (non-leaf nodes) besides the root. Thus, they can have between m and [m/2] children.
- The level of each leaf node is the same.

Need of B-tree

- For having optimized searching we cannot increase a tree's height. Therefore, we want the tree to be as short as possible in height.
- Use of B-tree in DBMS, which has more branches and hence shorter height, is the solution to this problem. Access time decreases as branching and depth grow.
- Hence, use of B-tree is needed for storing data as searching and accessing time is decreased.
- The cost of accessing the disc is high when searching tables Therefore, minimising disc access is our goal.

- So to decrease time and cost, we use B-tree for storing data as it makes the Index Fast.

How Database B-Tree Indexing Works:

- When B-tree is used for database indexing, it becomes a little more complex because it has both a key and a value. The value serves as a reference to the particular data record. A payload is the collective term for the key and value.
- For index data to particular key and value, the database first constructs a unique random index or a primary key for each of the supplied records. The keys and record byte streams are then all stored on a B+ tree. The random index that is generated is used for indexing of the data.
- So this indexing helps to decrease the searching time of data. In a B-tree, all the data is stored on the leaf nodes, now for accessing a particular data index, database can make use of binary search on the leaf nodes as the data is stored in the sorted order.
- If indexing is not used, the database reads each and every records to locate the requested record and it increases time and cost for searching the records, so B-tree indexing is very efficient.

How Searching Happens in Indexed Database?:

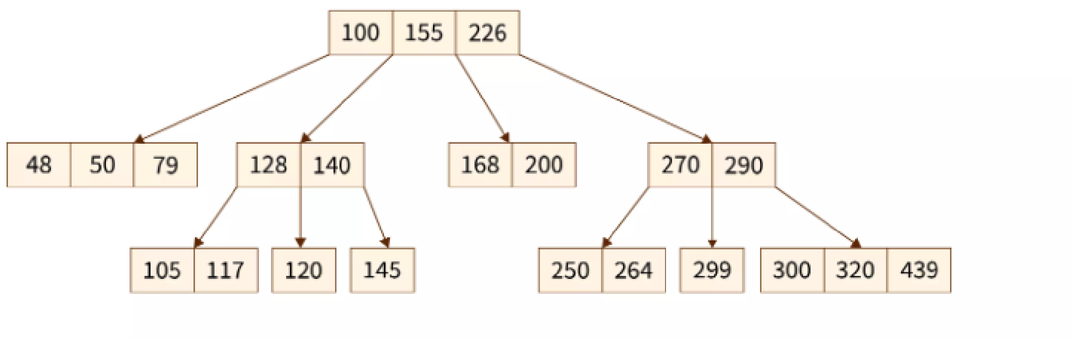
The database does a search in the B-tree for a given key and returns the index in O(log(n)) time. The record is then obtained by running a second B+tree search in O(log(n)) time using the discovered index. So overall approx time taken for searching a record in a B-tree in DBMS Indexed databases is O(log(n)).

Examples of B-Tree:

Suppose there are some numbers that need to be stored in a database, so if we store them in a B-tree in DBMS, they will be stored in a sorted order so that the searching time can be logarithmic.

Lets take a look at an example:

`B+Tree`



The above data is stored in sorted order according to the values, if we want to search for the node containing the value 48, so the following steps will be applied:

- First, the parent node with key having data 100 is checked, as 48 is less than 100 so the left children node of 100 is checked.

- In left children, there are 3 keys, so it will check from the leftmost key as the data is stored in sorted order.
- Leftmost element is having key value as 48 which match the element to be searched, so thats how we the element we wanted to search.

> *Learn more about B+Trees here: https://www.scaler.com/topics/data-structures/b-tree-in-data-structure/*

Cons of Indexes

While we have seen how indexes help make the read queries faster, like everything in engineering, they also have their cons. Let's try to think of those. What are the 4 types of operations we can do on a database? Out of these, which operations may require us to do extra work because of indexes?

Yes, whenever we update data, we may also have to update the corresponding nodes in the index. This will require us to do extra work and thus slow down those operations.

Also, do you think we can store index only on memory? Well technically yes, but memory is volatile. If something goes wrong, we may have to recreate complete index again. Thus, often a copy of index is also stored on disk. This also requires extra space. There are two big problems that can arise with the use of index:

1. Writes will be slower
2. Extra storage

Thus, it is recommended to use index if and only if you see the need for it. Don't create indexes prematurely.

Indexes on Multiple Columns

How do you decide on which columns to create an index? Let's revisit how an index works. If I create an index on the `id` column, the tree map used for storing data will allow for faster retrieval based on that column. However, a query like:

```sql
select * from students where psp = 90.1;
```

will not be faster with this index. The index on `id` has no relevance to the `psp` column, and the query will perform just as slowly as before. Therefore, we need to create an index on the column that we are querying.

We can also create index on 2 columns. Imagine a students table with columns like:

`id | name | email | batch_id | psp |`

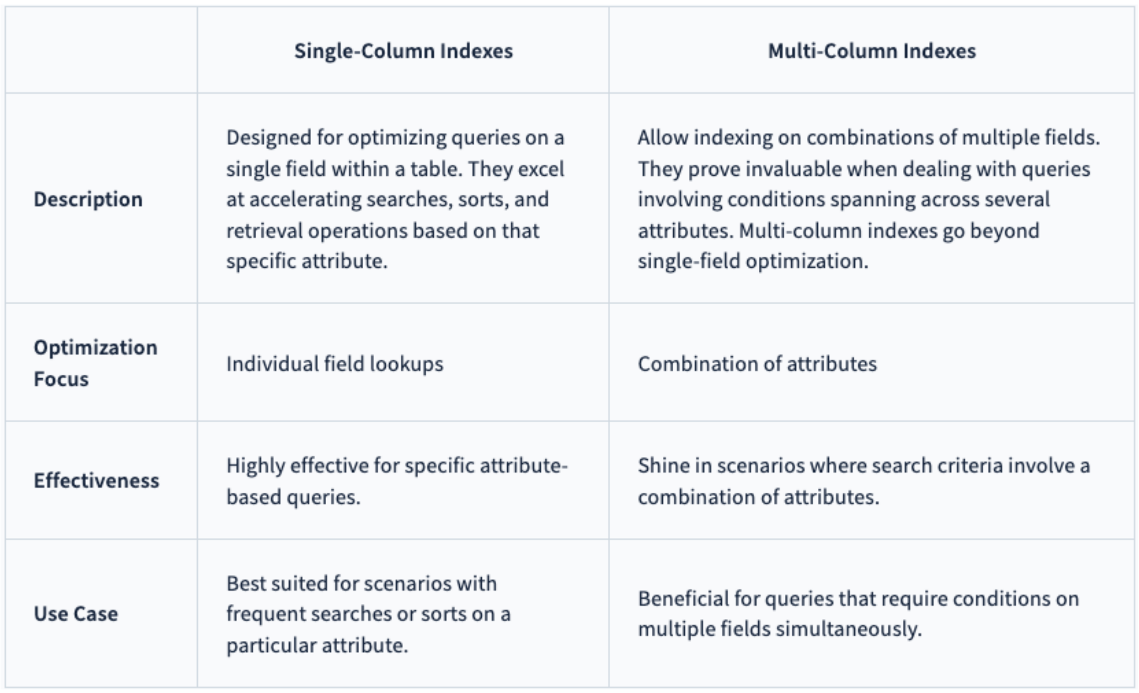
We are writing a query like this:

```sql
select * from students where name = 'Naman';
```

Let's say we create an index on (id, name).

When create index on these 2 columns, it is indexed according to id first and then if there is a tie it, will be indexed on name. So, there can be a name with different ids and we will not be able to filter it, as name is just a tie breaker here. The index is being created on first column i.e `id` and then the second column acts as a `tie breaker`.

Thus, if we create an index on `(id, name)`, it will actually not help us on the filter of name column.



	Single-Column Indexes	Multi-Column Indexes
Description	Designed for optimizing queries on a single field within a table. They excel at accelerating searches, sorts, and retrieval operations based on that specific attribute.	Allow indexing on combinations of multiple fields. They prove invaluable when dealing with queries involving conditions spanning across several attributes. Multi-column indexes go beyond single-field optimization.
Optimization Focus	Individual field lookups	Combination of attributes
Effectiveness	Highly effective for specific attribute-based queries.	Shine in scenarios where search criteria involve a combination of attributes.
Use Case	Best suited for scenarios with frequent searches or sorts on a particular attribute.	Beneficial for queries that require conditions on multiple fields simultaneously.

> *Read more here: https://www.scaler.com/topics/postgresql/multi-column-index-in-postgresql/*

Indexing on Strings

Now let's think of a scenario. How often do we need to use a query like this:

```sql
SELECT * FROM user WHERE email = 'abc@scaler.com';
```

But this query is very slow, so we will definitely create an index on the email column. So, the map that is created behind the scenes using indexing will have email mapped to the corresponding block in the memory.

Now, instead of creating index on whole email, we can create an index for the first part of the email (text before @) and have list of blocks (for more than one email having same first part) mapped to it. Hence, the space is saved.

Typically, with string columns, index is created on prefix of the column instead of the whole column. It gives enough increase in performance.

> *Note: Explain this with example.*

Consider the query:

```sql
SELECT * FROM user
WHERE address LIKE '%ambala%';
```

We can see that indexing will not help in such queries for pattern matching. In such cases, we use Full-Text Index.

> *More on Full-Text Indexing: https://www.scaler.com/topics/mysql-fulltext-search/*

How to create index

Let's look at the syntax using `film` table:

```
CREATE INDEX idx_film_title_release
ON film(title, release_year);
```

Good practices for creating index:

1. Prefix the index name by 'idx'
2. Format for index name - idx<underscore><table name><underscore><attribute name1><underscore><attribute name2>...

Now, let's use the index in a query:

```
EXPLAIN ANALYZE SELECT * FROM film
WHERE title = 'Shawshank Redemption';
```

`Output:`

```
-> Index lookup on film using idx_title (title='Shawshank Redemption')
(cost=0.35 rows=1) (actual time=0.0383..0.0383 rows=0 loops=1)
```

`Without Indexing:`

```
drop index idx_title on film;

EXPLAIN ANALYZE SELECT * FROM film
WHERE title = 'Shawshank Redemption';
```

`Output:`

```
-> Table scan on film  (cost=103 rows=1000)
(actual time=0.0773..1.01 rows=1005 loops=1))
```

We can clearly see `using indexing` the numeber of rows our query have to search to find answer is very less i.e just **1 row**. Meanwhile `without Indexing` the number of rows we have to iterate is `1000` which is very expensive.

If you look at the log of this query, "Index lookup on film using idx_film_title_release" is printed. If we remove the index and run the above query again, we can see that the time in executing the query is different. In case where indexing is not used, it takes more time to execute and more rows are searched to find the title.

> *More on how to create indexes: https://www.scaler.com/topics/how-to-create-index-in-sql/*

Clustered and Non-clustered Index?

Nowadays, we use databases to store records, and to fetch records much more efficiently, we use indexes. The index is a unique key made up of one or more columns. There are two types of indexes:

1. Clustered Index
2. Non-Clustered Index

Clustered Index:

A clustered index is a special type of index that rearranges the physical order of the table rows based on the index key values. This means that the data in the table is stored in the same order as the index. Clustered indexes can only be created on one column.

Non-Clustered Index:

A non-clustered index is a type of index that does not physically reorder the table data. Instead, it creates a separate structure that contains the index key values and pointers to the corresponding rows in the table.

Characteristics of the Clustered Index:

- **Advantages**: Faster queries for returning data in a specific order.
- **Disadvantages**: Only one clustered index can be created per table.
- **When to use**: When you need to return the data in a specific order regularly.

The characteristics of a clustered index are as follows:

- Default Indexing Methodology
- Can use a single or more than one column for indexing.
- Indexes are stored in the same table as actual records.

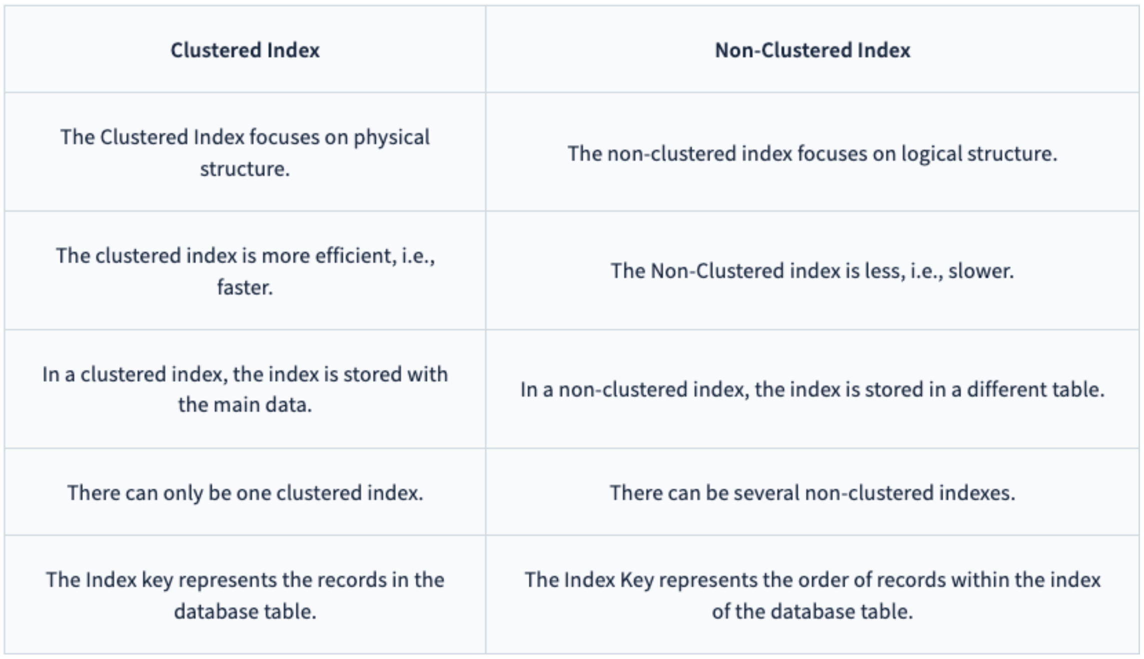
Characteristics of the Non-Clustered Index:

- **Advantages**: Multiple non-clustered indexes can be created per table.
- **Disadvantages**: Queries that use non-clustered indexes may be slower than queries that use clustered indexes, especially for queries that need to return a large amount of data.
- **When to use**: When you need to return data in a specific order, but you don't need to do this regularly.

The characteristics of the non-clustered index are as follows:

- Table data is stored in the form of key-value pairs.
- Tables or Views can have indexes created for them.
- It provides secondary access to records.
- A non-clustered key-value pair and a row identifier are stored in each index row of the non-clustered index.

Differences between Clustered and Non-clustered Index:



Clustered Index	Non-Clustered Index
The Clustered Index focuses on physical structure.	The non-clustered index focuses on logical structure.
The clustered index is more efficient, i.e., faster.	The Non-Clustered index is less, i.e., slower.
In a clustered index, the index is stored with the main data.	In a non-clustered index, the index is stored in a different table.
There can only be one clustered index.	There can be several non-clustered indexes.
The Index key represents the records in the database table.	The Index Key represents the order of records within the index of the database table.

> *Learn more about it here: https://www.scaler.com/topics/clustered-and-non-clustered-index/*

That's all for today. Thanks!