# DBMS Lecture 8- Indexing:

Introducton:

As we all know, Database store the data in Disk.

Let's consider a Students table, id, name, batch_id.

If we run a simple query, `select * from students where batch_id=2;`

Internally, It will be brought into the RAM, then the query will run.

CPU cannot directly talk to disk. - OS cannot directly operate on disk. - Content from disk is brought into memory and then application works on that data.

If a CPU were to directly work on disk, it will be wasting a lot of it's time. Speed of disk is very slow compared to CPU. - CPU will make an I/O call to bring data into memory. While data is being brought into memory it does something else.

Let's say we have a CPU, cache is almost directly siting on the CPU, small in the size. Because it is close to CPU, therefore very fast. Then RAM, it is volatile, it is bit far away from CPU then cache, and Then comes the disk, they are a persistent storage, tend to have a higher size but not as fast as RAM.

16GB of disk or 16GB of RAM, which is cheaper? Disk is cheaper.

Disk working:

By the nature of how disk works, it will be slower. Because we have to rotate a particular pointer, have to go to the correct place where data is stored then read the data and then you have to send to the CPU.

When you will try to read the data, it will give you the data of complete section and if the closer things are stored together you will get the data faster.

- A table is stored in the disk sorted by primary key.

and if I write a query,

```
select *
from students
where batch_id=2;
```

The very first operation that will happen is the query executor will go to the disk will try to read the first row, bring row1 to memory after this it checks if it matches the condition. Then bring row2 to memory similarly check the condition. Then bring row3 to memory check the condition and do the work.

If there were multiple students here that did not have the batch_id =3, then there will be many disk access that will be very slow and will not end up giving me any data also. let's say there are 1M rows, and there are only 20 students with batch_id=3, How many disk accesses are waste for us? 1M-20 will be waste for us, But we are wasting a lot of memory, time in that.

How do indexes work?

```
select *
from students
where id=3;
```

By default, data is sorted by primary key. Let's say I start going into the disl from first row, I bring the first row into the memory check it not relevant, Then next row, the third row, check it and relevant to me, So will I go to 4^th^ row? Because in query filter is there on the primary key where the id is 3, and if I get the id 3 in my table, now anything after this will not have id=3.

In previous example, we are filtering on non primary key, so It can be found anywhere in the table, so have to check the whole table..

Now, We can use this idea: Let's say in this students table, and ofcourse all of these information is stored on the disk, so have address. So if someone has given this information in memory like, Then how will this query works,

```
select *
from students
where id=3;
```

1. Find the address of the id from memory.
2. Go to the address and fetch data.
3. Done.

Here in this case, You even don't have to excess first 2 rows, I know the address and i can go to that address and will just do 1 disk access and it's done.

Let's say there are 1M rows, I need to get a student with id=15. Case1: No sorting and No table. Worst case is 1M rows.

Case2: Have sorting but No table. Then 15 rows to access.

Case3: Have sorting and Have table. Then only 1.

Here I am optimizing number of disk accesses. This is where indexes come into picture. It say, can we use this idea to speed up the query we have.

Let's say I have students table,

```
select *
from students
where p-No= 44667;
```

Can I apply the similar operation, Can I sort this data on the disk. Not really. Since data is already sorted on the disk by the primary key.

**V1:** Let me create a table, and this table is sorted by p-No.

So, 1. Find the address of the p-No from memory. 2. Go to the address and fetch data. 3. Done.

How will I get the p-No 4 from the table. HashMap

I binary search on the memory to find the p-No 5 then I go to that address and do the work.

**V2:**

```
Select *
from students
where batch_id=3;
```

What is different from V2 and V1? There is something that is fundamentally different, There will be multiple students with batch_id=3, there cannot be just 1 single row that has batch_id=3 and the disk is divided in 2 sections. Now, if I have to find all the students with batch_id=3. 1. Go to the table in memory, find all of those disk address.

Will I still get rows that don't have batch_id=3? Yes, But I will not access sections that don't have batch_id=3. I have saved irrelevant access of disk.

**V3:**

```
select *
from students
where psp between 60 and 80;
```

Case1: Don't have any helper table, 1. Go through every section 2. Fetch that section into memory 3. Compare the condition

Case2: Table

Is this going to be good? No. I will only have integer rows.

and I store the address like lets say there is a psp of 0 in address 1,4,7,8.

If I have the query, BETWEEN 10 AND 35;

First I will go to psp 10, Then take section 20 and above then 30 section. These tables are Index table or Index.

**Purpose of Index:** To prevent unnecessary disk fetches which leads to faster queries.

**When does an Index needs to be updated?** - When there is a Create/ Update/ Delete on the table, Index needs to be updated as well. - Index will needs to be updated whenever any write happens on a table. - Havind indexes will slow down writes but make reads faster. - Index also gets stored in disk=> storage requirement of DB will increase.

Table is something like a: <Key, Value> pair. Key=> column, Value=> address in disk. Table is also sorted by column.

Data Structure: - Sorted by key. - It allows fast access by key.

DS for indexes: BTree/ B+Tree.

Difference between BTree and Normal Binary Tree: A normal Binary Tree has 1 node having 2 children. In BTree: □ Now because there are multiple children,so height is reduced that means the query is going to become faster.

**When to create indexes?** - Don't create indexes when you create table. - Create indexes when you have a query that needs to be speed up. - Create indexes because of access patterns and not by predicting. - Do performance testing=> Do see how it's going to affect your writes, what is the performance impact that is happening.

So, Indexes make reads faster by reducing the number of irrelevant disk accesses.

In MYSQL Workbench:

```
 Explain select *
from sql_store.customers
where points=2273;
```

EXPLAIN command is basically going to tell you different things that are going to happen behind the scenes when you execute this query. After running this: Rows are basically the number of rows that you will have to fetch to execute this particular query: 22

So let's create index on points:

create index {idx_tablename_columnname} on {table_name}(column name)

```
 create index idx_customers_points on
sql_store.customers(points);
```

Now again run this:

```
 Explain select *
from sql_store.customers
where points=2273;
```

Now, rows will be just 1.

See all the indexes: `show indexes in sql_store.customers;` By default a table is always indexed with primary key.

Q- Find a customer with a particular address. Now, I will create index in address column.

But is that really going to be advantageous to us?? No, Because 2 customers are not going to have same address so in the index table I will end up having 1 row per address, so size of index become huge...

In scaler's codebase,

```
select *
from students
where name="naman";
```

1. Create index on students name. Index table is huge.

Now, If I would have indexed just on 'N'=[2,5,8,9] Then size of index will be 26 but again a lot of irrelevant accesses.

Now, index on 'NA'=[2,8,9] Chance of irrelevant access is bit less.

Now, index on 'NAM'=[8,9] After this it's going to be almost same.

- Why should I even create index on size 6 if 2, 3 is also almost giving me the same result.

So rather than creating index on complete column, just create index on first 3 characters.

---

### FULL TEXT INDEXES:

Let's say you have a blogging website where there is a table called blogs. 1 popular query on this table would be on contents.

Find all blogs that have react in their title or in content. And also advanced_search=> Find all blogs that contain react but doesn't contain redux.

So there comes Full Text Indexes: It basically helps in retrieval queries.

```
select *
from sql_blog.posts
where bosy like '%react%'
or title like '%react%';
```

Q- find the blog post that contain the word react and redux?

```
select *
from sql_blog.posts
where bosy like '%react%'
or title like '%react%'
and
(body like '%reedux%'
or title like '%redux%');
```

Now there can be many sophisticated things that I want to do. Solution: Full Text Index.

```
Create fulltext index idx_posts_body_title on
sql_blog.posts(bosy,title);
```

Now index is created.

```
select *
from posts
where match(title,body) against ("react redux" in boolean mode);
```

Here both are optional, If any title or any body contains either react or redux it will be returned.

Now,

```
 select *
from posts
where match(title,body) against ("+react redux" in boolean mode);
```

Here, It must contains react but redux is optional.

```
 select *
from posts
where match(title,body) against ("+react +redux" in boolean mode);
```

Here it must contain both.

```
 select *
from posts
where match(title,body) against ("+react redux -form" in boolean mode);
```

Here, it will not contain the form word.

So this type of advanced search capability is also there in MYSQL using a Full Text Index.

---

## COMPOSITE INDEXS:

students(name, phone number) - Here index table is first sorted by name then if two row have same name, tie breaker will be phone number.

- If two people will have same name and same phone number, then primary key will be the tie breaker.
- So if no tie breaker works, final tie breaker is primary key.

```
 select *
from students
where phone number=124;
```

Now is this be faster? Not be faster.

In a Composite Index, - Put the higher cardinality column first=> with more distinct values.

---