# DBMS Lecture 4

## CRUD and Joins.

**Delete:**

Syntax: delete from [table_name] where [condition] (optional) *If we don't put where, all rows will be deleted.*

In customers table, We want to delete the customer whose id is 17.

`delete from sql_store.customers where customer_id=17;`

---

**Select:**

Syntax: select {columns} from {table_name} where {conditions} order by {columns}

In students table, We only want to get those students whose id >10 and their first_name and batch.

```
 select f_name,batch
from students
where id>10;
```

Now, We want the students should should be ordered by their f_name.

```
 select f_name,batch
from students
where id>10
order by f_name;
```

If we don't write order by there is no guarantee of the order. It might happen that the very first time we do the SQL query and we will get D first and C second then the next time we might get C first and D second. So if we want the order to be consistent, we must use order by.

Order is not certain because, Table is analogous to set, In set order is not guaranteed. Set is not ordered, it's just the collection of things.

==Only select is mandatory.==

`select 1;` select is like read, so read 1.

`select 1,2;` From where we are getting these numbers? so 1,2 is just the value, select is just printing the value.

`select 1,2,'naman';`

These are just literals. The point is that we can write select statement without from, where,order by.

Now, select {col_names} //can have more than 1 column. from {table_name} //singular: only 1 table.

- If we want to get all the columns:

```
 select *
from students;
```

It will give all the students and all the columns of every student.

Now, In customers table, we want all the customers.

```
 select * from sql_store.customers;
```

Now I only want their first_name, city and points.

```
 select first_name, city, points
from sql_store.customers;
```

It will go through all the customers but will not filter the rows, instead only get the values of these 3 columns we mentioned.

Now, We can do operations on them as well. Example: Print the points with points +10.

```
 select first_name, city, points+10
from sql_store.customers;
```

Here, The name of first column is first_name, second is city and the name of third column is points+10.

So if we want a particular column to have a specific name, we can write **"as"**, it's basically an **alias.**

```
 select first_name as fName, city, points+10 as increasedPoints
from sql_store.customers;
```

It only affected the output table not the original customers table.

```
 select first_name
from customers;
```

Here duplicates will be there.

==A select statement basically goes through every row of the table , for every row if the condition exists, it will try to match it against the condition and if the condition is true it will return those rows.==

So in above query, it will got through all the customers, there is no condition here so no concept of filtering the rows, it will just get the value of first_name. Now there are many customers who have same first_name, that's why same value is going to get repeated.

If we want distinct values, Put **distinct** keyword.

```
 select distinct first_name
from customers;
```

So for all the columns it will only going to get those which are distinct values.

Now,

```
 select distinct first_name, points
from customers;
```

In this query also, there will be duplicates.

**Algorithm behind these queries:** for all rows (r) in table: if r matches the condition: list.add(columns in select of the row). if distinct: return distinct values of list else return all values of list.

---

**Where:** select ___ from ___ where {conditions}

*Example:* Students table: Now, Scaler only want those students whose psp is greater than 70.

```
 select *
from students
where psp>70;
```

These are called comparison operators.

> =: greater than equal to
> greater than <: smaller than <= : smaller than equal to = : Is this different from programming language, in SQL we have single equal to. !=, <> : Not equal to.

Now,

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

- In MYSQL, strings are case insensitive. naman=Naman=namAn all are same. So we will get all 4 rows as output.
- Strings are stored with the correct case but by default operations on strings are case insensitve.

Now, Get the customers whose square of points are less than 1M.

```
 select *
from sql_store.customers
where points*points<=1000000;
```

**Multiple conditions:** We want to get students of a particular batch whose psp>80 There are 2 conditions here, batch and psp.

**AND,OR,NOT** allow to combine multiple conditions.

**AND: batch_id=1 AND psp>80** This will go through every row, check if batch_id is greater thsn 1 and psp is greater than 80, if both the conditions are true then only output it.

In customers table, we want those customers who are born after 1985 and their points > 2000

```
 Select * from sql_store.customers
where birth_date >= "1985-01-01"
and points>=2000;
```

Now, there can be a condition when we don't want all the conditions to be true, we only want for eg, 1 condition to be true. Example: People who either born after 1995 or whose points is greater than 4000.

```
 select * from sql_store.customers
where birth_date>="1995-01-01"
OR points>=4000.
```

**NOT**: If (!(a==b){

} It reverses the condition=> (a!=b)

```
 select * from students
where NOT (psp>80
or batch_id =1);
```

Get those customers whose neither psp is greater than 80 nor batch_id=1.

```
 Select * from sql_store.customers
where NOT (birth_date >= "1985-01-01"
and points>=2000);
```

*Now, One thing to remember:*

==when we have multiple conditions to combine, please make use of parenthesis, because without them it might become confusing.==

Example:

```
 Select * from sql_store.customers
where (birth_date >= "1985-01-01"
or points<4000)
and state="HR";
```

This is understandable, either 1^st^ condition is true and 2^nd^ is true.

```
 Select * from sql_store.customers
where (birth_date >= "1985-01-01")
or (points<4000
and state="HR");
```

Get the customers whose either birth_date>1995 or points<4000 and belong to HR.

---

In Students table, We had their f_name, l_name and batch_id.

Get the students whose batch_id is either = 1 or 2 or 3 or 5.

```
 select * from students where batch_id =1
or batch_id =2
or batch_id =3
or batch_id =5;
```

This is really inconvenient.

**IN operator.**

```
 select * from students
where batch_id IN (1,2,3,5);
```

- If we have to compare a column to have one of the given values, use IN operator rather than writing conditions multiple times.

Get those customers who are in either VA or MA or HR.

```
 select * from sql_store.customers
where state IN (VA,MA,HR);
```

Now, Again we have students table, in that we have f_name, l_name, batch_id and psp.

Get all students of batch 2 where psp >=60 and psp<=90.

Whenever we have statements like, a>=x and a<=y We can use **BETWEEN operator:**

a BETWEEN x and y.

```
 select * from students
where batch_id =2
and
(psp between 60 and 90);
```

So here in customers table, Get those customers whose points are between 2000 and 4000.

```
 select * from sql_store.customers
where points between 2000 and 4000;
```

*One small thing to keep in mind here, it's greater than equal to not just greater than. Both sides are inclusive*

---

**LIKE operator:**

whenever we compare strings we have diverse use cases. Example: In batches table: id , name. □

Get all the batches that started in august. Now, we don't have start month of a batch anywhere but we see in name there is start month like august, may, june. Probably we can use name of the batch to get the info. But we can't do any equality check here. We need partial match. so here is LIKE.

% (percent): any number of any character.(>=0) _ (underscore): any single character.

```
 Select * from batches
where name like 'hello';
```

Get all the batches whose name is hello.

```
 select * from batches
where name like '%hello%'
```

If a batch with name 'amahelloam' is going to match, 'amhellobwdw' yes. 'hello' is also going to satisfy because 0 characters before then hello then again 0 characters. The name should have a pattern where there are some characters then hello then again some characters.

```
 select * from batches
where name like '_hello%';
```

Now, 'helloIam' is not going to satisfy the query because it doesn't have a single character before. 'abhelloIam' No, again 2 characters before hello. 'ahelloa' yes. - Find all containing x. - for example, you want to find all strings containing hello.

Now, Let's say we want those customers who belong to a city that contains the character 'o'.

```
 select * from sql_store.customers
where city like '%o%';
```

Now, we want to get the customers whose state name is anything like 'A'.

```
select * from sql_store.customers
where state like '_a';
```

---

Now, Let's say we want those customers whose phone_no is not there. **IS NULL**:

```
select * from sql_store.customers
where phone IS NULL.
```

```
select * from sql_store.customers
where phone = NULL.
```

This will not give correct answer, just completely null row.

- If we have to check the value of a column to be equal to null, don't use x=null, use x IS NULL.

---

**Order By**: - By default in most of the SQL, the order of output is going to be sorted in ascending order by the primary key. - By default sorting happens in ascending order.

In scaler codebase we see the list of students in leaderboard, but they are sorted by psp. Scaler at that particular time is saying, get all the students whose batch_id is 2 and sort them by psp in descending order.

```
select * from students
order by (psp, first_name);
```

This means sort students by psp, if 2 students have same psp sort them by first_name.

In this table, we want 3^rd^ student,

```
select * from students
order by (psp, first_name);
```

Answer of this query, E.

Now, Who is the 3^rd^ student.

```
select * from students
order by (psp, first_name,last_name);
```

So EH is the answer.

Now, What if after last name as well there was a tie? Then it orders the tie by primary key. ==Primary key is the last tie breaker always.==

If we want to sort them in descending order.

```
select * from students
order by (psp, first_name,last_name) DESC;
```

In MYSQL Workbench, We want to order the customers by the name of their state

```
 select * from customers
order by state;
```

Here customers who are having same state are sorted by primary key.

Now, the user want to sort them in descending order.

```
 select * from customers
order by state DESC;
```

But by primary key they are sorted in ascending order.

```
 select * from customers
order by state, customer_id DESC;
```

First sort by state, but we didn't tell the order so by default it sort by ascending order. So this is not going to sort the customers in descending order of their state.

so,

```
 select * from customers
order by state desc, customer_id desc;
```

First we sorted all the customers in descendiing order by state, if they have tie, sort those customers by their customer_id in descending order.

```
 select * from customers
order by state desc, customer_id;
```

Now, customer_id wil be sorted in ascending order.

---

**LIMIT:** The table might have 5000 rows, but the user only care of 10 rows, then can have limit.

```
 select * from blogs
order by publish_date
limit 5;
```

Get 5 recent blogs.

```
 select * from students
limit 5;
```

This query will give the first 5 students based on their primary key.

```
 Select * from customers
order by customer_id
limit 5;
```

---