DBMS Lecture 6

Aggregation:

==Grouping/Collecting/Combining things.==

Example:

```
 select * from students
where----
```

This query goes through every row of the table, it checks that particular row with the conditon, if that condition is matched it returns the row, if the condition doesn't match it doesn't return the row. But still it's a row by row operation.

Example: In students table, **Q**- Get the student with maximum psp. Now, here we are not actually returning a single row, but probably doing an operation across the row, going to find the maximum number of psp and then will give the answer.

Q- Find the average psp of the batch. Here also returning the average psp is not returning a row of students table.

Q- Find the total number of scaler coins with the students of a batch. Here also we are not returning a single row but doing an operation across the row and then returning the value.

So basically, ==Aggregate functions take a set of values and they convert it to a single value.== Example: sum(1,3,4,7)=> 15.

Common Aggregate Functions:

Max(): from a set of rows, it finds the maximum value. Min(): It will give the minimum value. Avg():It will give the average value. Sum(): It will give the sum total of all the values. Count(): It gives the number of values passed to it. Count(9,6,7,8)=> 4

==Aggregate functions only consider NON NULL values.== Count(5,NULL,6,7)=> 3

Q- Find total nnumber of emails of students in a batch.

```
 select count(email)
from students
where batch=123;
```

It will go to the students table, First it selects the students with batch 123. Then count(abc@gmail, bcf@gmail, fh@gmail)=> 3 What if the email is NULL? Count(abc@gmail, bcf@gmail, fh@gmail, NULL) => Now here are 4 students but count will give 3.

Count(*) Complete row. count(Row1, Row2, Row3, Row4) And complete row can never be NULL because at least primary key can't be NULL.

MYSQL Workbench: In Customers table, **Q**- What is the maximum number of points that any customer have?

```
 Select max(points)
from sql_store.customers;
```

```
 select min(points)
from sql_store.customers;
```

```
 select avg(points)
from sql_store.customers;
```

Q- How many customers have phone_number?

```
 select count(phone_number)
from sql_store.customers;
```

Till now all of aggregate functons are working across the whole table. Let's say In students table,

Q- Find average psp of students.

```
 select avg(psp)
from students;
```

Q- Find average psp of students of every batch. Output table:

Here, We still need to find average, but we don't have to aggregate across the table.

So what to do: 1. Group different rows of the table. 2. On every group, find the average.

GROUP BY:

```
 select {columns that were there in group by or aggregate functions}
from students
group by batch_id
```

```
 select batch_id, avg(psp)
from students
group by batch_id;
```

Is this true?

```
 select student_id, batch_id, avg(psp)
from students
group by batch_id;
```

No, Because student_id is not the column that is included in group by.

Internal working:

1. Group rows by columns in group by.
2. Call aggregate functions on each group.

MYSQL Workbench: In Orders table,

Q- Find how many orders are there in every status that is possible.

```
 select status, count(*)
from orders
group by status;
```

In students table,

Q- Find batches where students have an average psp >= 80.

Option 1:

```
 select batch_id, avg(psp)
from students
where avg(psp)>=80
```

Option 2: ``` select batch_id, avg(psp) from students where psp>=80 group by batch_id.

```
 Not works.

Where is a statement that works on the rows of original tables.

It will go through every row, and find the rows where psp is >=80 and these rows are going to be group

But the question is to find the batches where average psp>=80.
There can be student whose psp is 60, one student whose psp is 100. But as long as the average psp is 
But here we will end up returning all the batches because we are only considering the students whose p
So option 2 is definately incorrect.

In Option1,
`where avg(psp)>=80`

Now, Where woks when an SQL query goes through every row of the original table, then we don't even kno



---
**HAVING:**

- Where can't be used after group by.
- If we have to filter after grouping the rows, we will use HAVING clause..
```

Select batch_id,avg(psp) from students group by batch_id having avg(psp)>=80;

```
 We have students table,

Now, Select something from students but don't have to select something from students as it is. We have

1. This particular table gets grouped by batch_id.
Once we have grouped the rows, now the original rows are no longer there. Now SQL is going to do every
2. Now it will run a query, whatever the rows are find the average psp for that.
3. Consider the groups that satisfy the condition.

![](https://i.imgur.com/MxQwdmr.png)
Output:
![](https://i.imgur.com/ZdJ9q0x.png)

1. From: Which table to dop query
2. where: which rows of table to consider
3. Group by: Combine the rows outputed by 2.
If no where, consider all rows
4. Having: Filtering the groups.
By aggregate functons or by the columns that were there in group by.

Where vs Having:




| Where              | Having               |
| ------------------ | -------------------- |
|  Where filters rows | Having filter groups.                      |
| Comes before group by  | comes after group by       |
| Columns of the table to filter | Aggregate functions or columns via which table was grouped |

**Q-** Find the average psp of every batch that has more than 5 students.

select {After everything}
from students
{Do we have to filter the students? No where needed}
group by batch_id
{Now, Do we have to filter the groups or consider all groups? Filter}
having count(*)>5
```

select batch_id, avg(psp) from students group by batch_id having count(*)>5;

```
**Q-** Find average quantity in stock of products where unit price>=2.
```

select quantity_in_stock, product_id from sql_inventory.products where unit_price>=2;

```
 This is wrong.
1. Filter the products where unit price>=2.
2. On these rows we have to do the average, don't have to group by.

```

select avg(quantity_in_stock) from sql_inventory.products where unit_price>=2;

```


**Q-** Find the invoice_id, total payment with total payment >40.

```

select invoice_id, sum(amount) from payments group by invoice_id having sum(amount)>40;

```


---

## BuiltIn functions:

- MYSQL implementations give a set of functions to perform a common operation.
- Numbers
- Strings
- Dates

**Numbers:**
1. ROUND:(1.91) Round to one decimal point=> 1.9
Takes a number, digits after decimal.
ROUND(2.31789,2)2=>2.32

2. TRUNCATE: Removing the digits.
(number, digits after decimal)
Example: TRUNCATE(2.31789,2)=> 2.31 (Just removing the other digits).

3. CEILING: Next integer >= number
Example: CEILING(3)=> 3
CEILING(2.94)=> 3

4. FLOOR: First integer <= number
Example: FLOOR(3)=> 3
FLOOR(2.94)=> 2

5. ABS: Absolute value of a number(positive)
ABS(2.94)=> 2.94
ABS(-2.94)=> 2.94

6. RAND: Doesn't take any parameter and returns a random value b/w 0 to 1.[0,1]

**Build a function that generates kind of random values b/w [0,10] (Integers)**

**FLOOR(RAND()*10)**
but this function will give 10 very less time.
```

```
0- 0.0999 = 0
0.1- 0.1999= 1
0.2- 0.299 = 2

0.9- 0.999= 9
1= 10
We are being partial against 10.

**ROUND(RAND,1)*10**

0.001 => 0.0*10=> 0

0.00- 0.0499=> 0
0.05- 0.1499=> 1
0.15- 0.2499=> 2

0.85- 0.9499=> 9
0.95- 1=> 10
It's more uniform.



---

Strings:

1. LENGTH:
LENGTH(string) gives the length of string

2. UPPER:
UPPER(delhi)=> DELHI

3. LOWER:
LOWER(DELHI)=> delhi

4. LTRIM: Removes the spaces at the start.
LTRIM("    DELHI")=> "DELHI"

5. RTRIM: Removes the spaces from right side.

6. TRIM: removes spaces from both the sides.

7. LOCATE: Finding if something exist in a string.
LOCATE(toFind, fromString)
Find if 'el' exist in delhi.
LOCATE(el, delhi)
=> 0, doesn't exist.
=> returns index of starting position.
=> Answer=> 2.
==INDEX in MYSQL starts from 1.==
8. Extract a chunk of String.

Let's say we have a string, x='dd/mm/yy'

**Q-** Print the date for every row.
Basically to find first 2 characters.

8.1: LEFT(STRING, LENGTH)=> 1^st^ length characters
LEFT(x,2) => dd

**Q-** Print the year part.

8.2: RIGHT(STRING, LENGTH)
RIGHT(x,2)=> yy

**Q-** Find the month.

8.3: SUBSTRING(STRING, START, LENGTH)
SUBSTRING(x, 4, 2)=> mm

---

**DATE:**
Also case insensitive.
1. NOW(): Prints current date and time.

IN MYSQL Workbench,
select now();

select curdate(): current date

select curtime(): current time.

2. CURDATE()
3. CURTIME()

4. YEAR(date)=> give year part.
YEAR(2022-10-05)=>2022

5. MONTH(date): Month part.
MONTH(2022-10-05)=> october.

6. DAYNAME: It gives the day on which this date happend.
Example: Monday. (2022-10-05 happens on monday let's say.)

Example:
```

select * from customers where year(birth_date)> 1990;

```
 We have to add something to a date or subtract something, find difference between dates.

1. DATE-ADD(date, interval)
interval=> interval {quantity} {unit}
Exanple: interval 1 day, so it's going to add 1 day to this particular date.
interval 1 minute: add 1 minute to this particular date that is given.

**Q-** Print the time after 10 minute.

`select date-add(now(),interval 10 minute);`

2. DATE-SUB(date, interval)

Amazon has a repeated process where every day they check the total amount of orders placed in last 24

In orders table, have columns like, id, amount, placed_at

```

select sum(amount) from orders where placed_at>=date-sub(now(),interval 1 day);

```


3. DATE-DIFF: difference between 2 dates.
DATE-DIFF(date1, date2)=> return the number of days

Orders table, id, amount, placed_at, paymentAt.
And 1 order can be paid any day after order is placed.

**Q-** Find all the orders where payment was done >= 2 days after placing the order.

```

select * from orders where datediff(paymentAT, placed_at) >= 2; ``` It takes only date part.

4. DATE-FORMAT: These functions take a date and then the format which we want the date to be returned in.
