
Agenda

- Aggregate Queries
  - Aggregate Functions
    - COUNT
    - * (asterisk)
- Other aggregate functions
- GROUP BY Clause
- HAVING Clause

Aggregate Queries

Hello Everyone, till now whatever SQL queries we had written worked over each row of the table one by one, filtered some rows, and returned the rows. Eg: We have been answering questions like:

- Find the students who ...
- Find the batches who ...
- Find the name of every student.

But now we will be answering questions like:

- What is the average salary of all the employees?
- What was the count of movies released in each year?
- What is the maximum salary of all the employees?

In above questions, we are not interested in the individual rows, but we are interested to get some data by combining/aggregating multiple rows. For example, to find the answer of first, you will have to get the rows for all of the employees, go through their salary column, average that and print.

How to do this is what we are going to learn now.

> *Activity: Search meaning of aggregate on Google.*

Aggregate Functions

SQL provides us with some functions which can be used to aggregate data. These functions are called aggregate functions. Imagine a set of column. With the values of that column across all rows, what all operations would you may want to do?

Correct, to allow for exactly all of these operations, SQL provides us with aggregate functions. Aggregate functions will always output 1 value. Let's go through some of these functions one by one and see how they work.

COUNT

Count function takes the values from a particular column and returns the number of values in that set. Umm, but don't you think it will be exactly same as the number of rows in the table? Nope. Not true. Aggregate functions only take `not null values` into account. So, if there are any `null values` in the column, they will not be counted.

Example: Let's take a students table with data like follows:

> *STUDENTS*

id	name	age	batch_id
1	A	20	1
2	B	21	1
3	C	22	null
4	D	23	2

If you will try to run `COUNT` and give it the values in batch_id column, it will return 3. `Because there are 3 not null values in the column`. This is different from number of rows in the students table.

Let's see how do you use this operation in SQL.

```
SELECT COUNT(batch_id) FROM students;
```

To understand how aggregate functions work via a pseudocode, let's see how SQL query optimizer may execute them.

```
table = []

count = 0

for row in table:
    if row[batch_id]:
        count += 1

print(count)
```

Few things to note here: While printing, do we have access to the values of row? Nope. We only have access to the count variable. So, we can only print the count. Extrapolating this point, when you use aggregate functions, you can only print the result of the aggregate function. You cannot print the values of the rows.

Eg:

```
SELECT COUNT(batch_id), batch_id FROM students;
```

This will be an invalid query. Because, you are trying to print the values of `batch_id` column as well as the count of `batch_id` column. But, you can only print the count of `batch_id` column.

---

* (asterisk) to count number of rows in the table

What if we want to count the number of rows in the table? We can do that by passing a `*` to the count function.

* as we know from earlier, refers to all the columns in the table. So, count(*) will count the number of rows in the table. You may think what if there is a `Null` value in a row. Yes, there can be one/more `Null` values in a row but the whole row can't be `Null` as per rule of MySQL.

```
SELECT COUNT(*) FROM students;
```

The above query will print the number of rows in the table.

Other aggregate functions

We can use multiple aggregation function in the same query as well. For example:

```
SELECT COUNT(batch_id), AVG(age) FROM students;
```

Some aggregate functions are as follows.

1. MAX: Gives Maximum value
2. MIN: Gives minimum value

Note that, values in the column must be comparable for MAX and MIN.

3. AVG: Gives average of non NULL values from the column. For example: AVG(1, 2, 3, NULL) will be 2.
4. SUM: Gives sum, ignoring the null values from the column.

`Max, Min`

```
SELECT MAX(age), MIN(age)
FROM students;

-- SOLUTION:

MAX  MIN
23    20
```

`Sum`

```
SELECT SUM(batch_id)
FROM students;

-- SOLUTION:

SUM
4
```

> *Very Important: We can't use Aggregatets in Nested. It will give us error.*

Many times we have seen learners using Nesting in `Aggregates` during Mock interviews. Please note this for future reference now. Example:

```
SELECT SUM(COUNT(batch_id))
FROM STUDENTS;

-- This above query will not work.
```

However distinct can be used inside an aggregate function as distinct is not an aggregate function.

`Example:`

```
SELECT SUM(DISTINCT(batch_id))
FROM STUDENTS;

-- Here we have only two batch_id which are DISTINCT 1, 2
-- Therefore we will do sum of (1, 2) i.e = 3.
```

> *Learn more about aggregates here: https://www.scaler.com/topics/sql/aggregate-function-in-sql/*

GROUP BY clause

Till now we combined multiple values into a single values by doing some operation on all of them. What if, we want to get the final values in multiple sets? That is, we want to get the set of values as our result in which each value is derived from a group of values from the column.

The way Group By clause works is it allows us to break the table into multiple groups so as to be used by the aggregate function.

For example: `GROUP BY batch_id` will bring all rows with same `batch_id` together in one group

> *Note: Also, GROUP BY always works before aggregate functions. Group By is used to apply aggregate function within groups (collection of rows). The result comes out to be a set of values where each value is derived from its corresponding group.*

Let's take an example.

id	name	age	batch_id
1	A	20	1
2	B	21	3
3	C	22	1
4	D	23	2
5	E	23	1
6	F	25	2
7	G	22	3
8	H	21	2
9	I	20	1

```
SELECT COUNT(*), batch_id FROM students GROUP BY batch_id;
```

The result of above query will be: | COUNT(*) | batch_id | |----------|----------| | 4 | 1 | | 3 | 2 | | 2 | 3 |

Explanation: The query breaks the table into 3 groups each having rows with `batch_id` as 1, 2, 3 respectively. There are 4 rows with `batch_id = 1`, 3 rows with `batch_id = 2` and 2 rows with `batch_id = 3`.

Note that, we can only use the columns in SELECT which are present in Group By because only those columns will have same value across all rows in a group.

Now let's try to understand this using Infographics:

Here's our student table. Notice how the data is mixed and not organised by batch. Normally, this data is just a list. But what if we want to organise these students by their batch?

`Students:`

student_id	name	batch_name
1	John Doe	Batch A
2	Jane Smith	Batch B
3	Alice Johnson	Batch A
4	Bob Brown	Batch C
5	Emily White	Batch B
6	Michael Green	Batch C
7	Sarah Parker	Batch A
8	Ryan Miller	Batch B
9	Olivia Garcia	Batch C
10	William Turner	Batch A

Now, let's see what might happen internally when we apply the 'Group By' clause to this table, organising it by batch names.

student_id	name	batch_name
1	John Doe	Batch A
2	Jane Smith	Batch B
3	Alice Johnson	Batch A
4	Bob Brown	Batch C
5	Emily White	Batch B
6	Michael Green	Batch C
7	Sarah Parker	Batch A
8	Ryan Miller	Batch B
9	Olivia Garcia	Batch C
10	William Turner	Batch A

After grouping, notice how all students from each batch are now grouped together, making it easier to analyse the data where we can apply aggregates as well.

`Final group of students:`

student_id	name	batch_name
1	John Doe	Batch A
3	Alice Johnson	Batch A
7	Sarah Parker	Batch A
10	William Turner	Batch A

student_id	name	batch_name
2	Jane Smith	Batch B
5	Emily White	Batch B
8	Ryan Miller	Batch B

student_id	name	batch_name
4	Bob Brown	Batch C
6	Michael Green	Batch C
9	Olivia Garcia	Batch C

Now we can apply queries like:

```
SELECT count(*), batch_name
FROM students
GROUP BY batch_name;

-- It will give output as count of students for each batch.
-- Also we can print data which is common to a group in our case batch_name.
-- Groups don't have student's name as common. So, there will be an error if you use name in select.
```

HAVING Clause

HAVING clause is used to filter groups. Let's take a question to understand the need of HAVING clause:

There are 2 tables: Students(id, name, age, batch_id) and Batches(id, name). Print the batch names that have more than 100 students along with count of the students in each batch.

```
SELECT COUNT(S.id), B.name
FROM Students S
JOIN Batches B ON S.batch_id = B.id
GROUP BY B.name;
HAVING COUNT(S.id) > 100;

-- Using `WHERE` here instead of `HAVING` can give us error.
```

Here, `GROUP BY B.name` groups the results by the `B.name` column (batch name). It ensures that the count is calculated for each distinct batch name. `HAVING COUNT(S.id) > 100` condition filters the grouped results based on the count of `S.id` (number of students). It retains only the groups where the count is greater than 100.

The sequence in which query executes is:

- Firstly, join of the two tables is done.
- Then is is divided into groups based on `B.name` .
- In the third step, result is filtered using the condition in HAVING clause.
- Lastly, it is printed through SELECT.

`FROM -> WHERE -> GROUP BY -> HAVING -> SELECT`

`WHERE is not build to be able to handle aggregates`. We can not use WHERE after GROUP BY because WHERE clause works on rows and as soon as GROUP BY forms a result, the rows are convereted into groups. So, no individual conditions or actions can be performed on rows after GROUP BY.

> ***Note: WHERE is not build to be able to handle aggregates as 'WHERE' works with rows not groups***

- Differences between Having and Where:

Comparison Basis	WHERE Clause	HAVING Clause
Definition	It is used to perform filtration on individual rows.	It is used to perform filtration on groups.
Basic	It is implemented in row operations.	It is implemented in column operations.
Data fetching	The WHERE clause fetches the specific data from particular rows based on the specified condition	The HAVING clause first fetches the complete data. It then separates them according to the given condition.
Aggregate Functions	The WHERE clause does not allow to work with aggregate functions.	The HAVING clause can work with aggregate functions.
Act as	The WHERE clause acts as a pre-filter.	The HAVING clause acts as a post-filter.
Used with	We can use the WHERE clause with the SELECT, UPDATE, and DELETE statements.	The HAVING clause can only use with the SELECT statement.
GROUP BY	The GROUP BY clause comes after the WHERE clause.	The GROUP BY clause comes before the HAVING clause.

That's all for this class. If there are any doubts feel free to ask now. Thanks!