**Where**

**groupby**

**having**

**orderby**

**Group By Clause**

* GROUP BY clause is used with the SELECT statement.
* In the query, GROUP BY clause is placed after the **WHERE clause.**
* In the query, GROUP BY clause is placed **before ORDER BY clause** if used any.

## syntax

The syntax for the GROUP BY clause in SQL Server (Transact-SQL) is:

SELECT expression1, expression2, ... expression\_n,

aggregate\_function (expression)

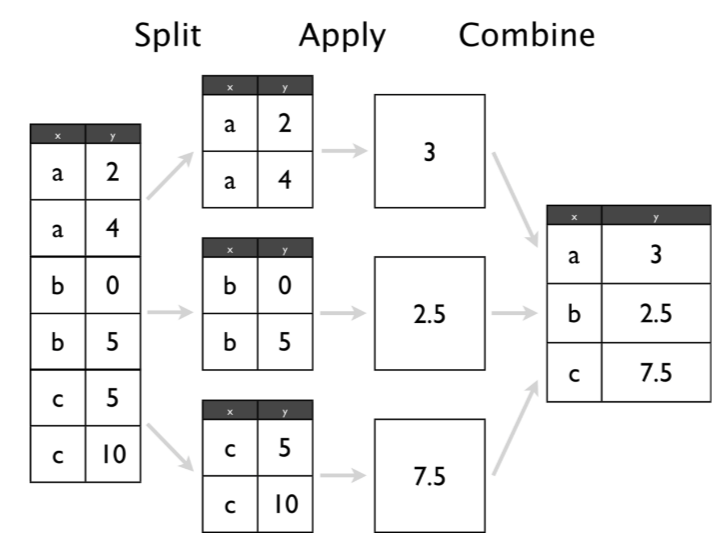
FROM tables

[WHERE conditions]

GROUP BY expression1, expression2, ... expression\_n;

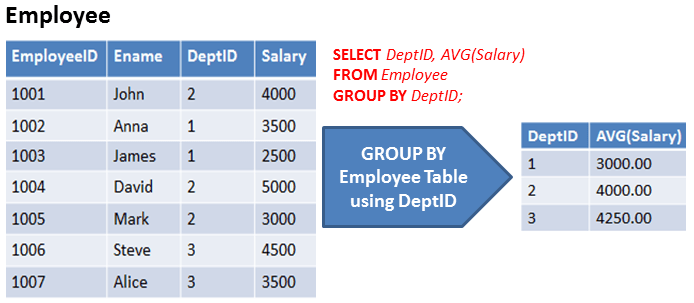
The GROUP BY Clause is utilized in SQL with the SELECT statement to organize similar data into groups. It combines the multiple records in single or more columns using some functions. Generally, these functions are aggregate functions such as **min(),max(),avg(), count(), and sum()** to combine into single or multiple columns. It uses the **split-apply-combine** strategy for data analysis.

* In the split phase, It divides the groups with its values.
* In the apply phase, It applies the aggregate function and generates a single value.
* In the combiner phase, It combines the groups with single values into a single value.

[Image source](https://medium.com/@sean.turner026/applying-custom-functions-to-groupby-objects-in-pandas-61af58955569)

**Points to Remember:**

* GROUP BY Clause is utilized with the SELECT statement.
* GROUP BY aggregates the results on the basis of selected column: COUNT, MAX, MIN, SUM, AVG, etc.
* GROUP BY returns only one result per group of data.
* GROUP BY Clause always follows the WHERE Clause.
* GROUP BY Clause always precedes the ORDER BY



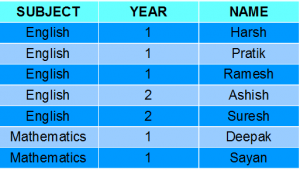
In above example, Table is grouped based on the DeptID column and Salary is aggregated department-wise.

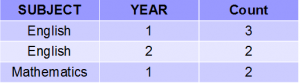
* E**Group By single column**: Group By single column means, to place all the rows with same value of only that particular column in one group. Consider the query as shown below:
* SELECT NAME, SUM(SALARY) FROM Employee
* GROUP BY NAME;
* Its is not mandatory that group by is used always with where we can use use group by without where clause also
* For example
* select deptid ,MAX(salary) from Employee group by deptid

--waq to find number of Employees working in each dept

* select empname, deptid,count(\*) from Employee group by deptid
* We have to select the column which is in aggregate function
* for example
* Msg 8120, Level 16, State 1, Line 14
* Column 'Employee.empname' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

The above query will produce the below output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/table_out.png)  
As you can see in the above output, the rows with duplicate NAMEs are grouped under same NAME and their corresponding SALARY is the sum of the SALARY of duplicate rows. The SUM() function of SQL is used here to calculate the sum.

* **Group By multiple columns**: Group by multiple column is say for example, **GROUP BY column1, column2**. This means to place all the rows with same values of both the columns **column1** and **column2** in one group. Consider the below query:
* 
* SELECT SUBJECT, YEAR, Count(\*)
* FROM Student
* GROUP BY SUBJECT, YEAR;

**Output**:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-55.png)  
As you can see in the above output the students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

**HAVING Clause**

We know that WHERE clause is used to place conditions on columns but what if we want to place conditions on groups?

This is where HAVING clause comes into use. We can use HAVING clause to place conditions to decide which group will be the part of final result-set. Also we can not use the aggregate functions like SUM(), COUNT() etc. with WHERE clause. So we have to use HAVING clause if we want to use any of these functions in the conditions.

**Syntax**:

SELECT column1, function\_name(column2)

FROM table\_name

WHERE condition

GROUP BY column1, column2

HAVING condition

ORDER BY column1, column2;

**function\_name**: Name of the function used for example, SUM() , AVG().

**table\_name**: Name of the table.

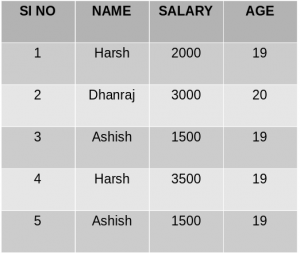
**condition**: Condition used.

**Example**:

SELECT NAME, SUM(SALARY) FROM Employee

GROUP BY NAME

HAVING SUM(SALARY)>3000;

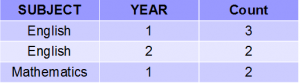


**Output**:  
[Screenshot (56)](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-56.png)  
As you can see in the above output only one group out of the three groups appears in the result-set as it is the only group where sum of SALARY is greater than 3000. So we have used HAVING clause here to place this condition as the condition is required to be placed on groups not columns.

This article is contributed by [**Harsh Agarwal**](https://www.facebook.com/harsh.agarwal.16752). If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

* **Group By multiple columns**: Group by multiple column is say for example, **GROUP BY column1, column2**. This means to place all the rows with same values of both the columns **column1** and **column2** in one group. Consider the below query:
* SELECT SUBJECT, YEAR, Count(\*)
* FROM Student
* GROUP BY SUBJECT, YEAR;

**Output**:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-55.png)  
As you can see in the above output the students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

We use cookies to ensure you have the best browsing experience on our website. B

## xample - Using SUM function

Let's look at a SQL Server GROUP BY query example that uses the [SUM function.](https://www.techonthenet.com/sql_server/functions/sum.php)

For example:

SELECT product\_name, SUM(quantity) AS "Total quantity"

FROM products

GROUP BY product\_name;

This SQL Server GROUP BY example uses the SUM function to return the product\_name and the total quantity (for the product\_name).

Because you have listed one column (the product\_name field) in your SELECT statement that is not encapsulated in the SUM function, you must use the GROUP BY clause. The product\_name field must, therefore, be listed in the GROUP BY clause.

## Example - Using COUNT function

Let's look at how we could use the GROUP BY clause with the [COUNT function](https://www.techonthenet.com/sql_server/functions/count.php).

For example:

SELECT manager\_id, COUNT(\*) AS "Number of employees"

FROM employees

WHERE last\_name = 'Anderson'

GROUP BY manager\_id;

This GROUP BY example uses the COUNT function to return the manager\_id and the number of employees whose last\_name is 'Anderson'.

## Example - Using MIN function

Let's next look at how we could use the GROUP BY clause with the [MIN function](https://www.techonthenet.com/sql_server/functions/min.php).

For example:

SELECT product\_type, MIN(quantity) AS "Lowest quantity"

FROM products

GROUP BY product\_type;

This GROUP BY example uses the MIN function to return the product\_type and the minimum quantity for that product\_type.

## Example - Using MAX function

Finally, let's look at how we could use the GROUP BY clause with the [MAX function](https://www.techonthenet.com/sql_server/functions/max.php).

For example:

SELECT department, MAX(salary) AS "Highest salary"

FROM employees

GROUP BY department;

This GROUP BY example uses the MAX function to return the name of each department and the maximum salary in the department.

### 1. Forgetting GROUP BY with Aggregate Functions

You use SELECT statements with the GROUP BY clause when you want to group and organize rows into specific groups and then perform a specific calculation of each group.

The most common GROUP BY error is forgetting to write GROUP BY inside the SELECT statement.

Here is one example. Imagine that you have the table **recipes**, which contains 100 records and six columns. This table stores the number of views (no\_of\_views) per each recipe published on a famous culinary website:

| **meal\_category** | **name** | **author** | **no\_of\_views** | **no\_of\_views\_lst\_mth** | **author\_id** |
| --- | --- | --- | --- | --- | --- |
| Cold appetizer | Marinated Cheese | Marta | 107104 | 90621 | 1 |
| Soups | Pumpkin soup | John | 68856 | 69377 | 2 |
| desserts | Banana Cheesecake | Ally | 131944 | NULL | 3 |
| drinks | Paloma Picante | Luke | 72027 | 71312 | 4 |
| Bread and pastry | Sour Cream Doughnuts | John | 50935 | 52791 | 2 |
| desserts | Real Strawberry Cupcakes | Lisa | 176268 | 116939 | 11 |
| Soups | potato soup | Mary | 64796 | 64388 | 6 |
| ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... |
| ... | ... | ... | ... | ... | ... |
| Bread and pastry | Cider Doughnuts | Tim | 53896 | 51160 | 8 |

recipe table

Here is a short description of the table’s columns:

* meal\_category – The recipe category (soup, drinks, desserts, etc.).
* name – The recipe’s name.
* author – The author’s name.
* no\_of\_views – The number of views (total pages/recipes viewed) in the current month.
* no\_of\_views\_lst\_mth – The number of views (total pages/recipes viewed) in the previous month.
* author\_id – The author’s unique ID number.

Let's say that you want to count the number of recipes in each meal category. If you write the statement like this (without GROUP BY at the end) ...

|  |
| --- |
| SELECT    meal\_category,    count(\*) AS total\_recipes  FROM recipes ; |

... most SQL engines will give you an error. Some engines, however, will output weird, unwanted results. I’m using MySQL and when I run this statement, I get this:

| **meal\_category** | **total\_recipes** |
| --- | --- |
| Cold appetizer | 100 |

Result without GROUP BY

100 is the total count of all the recipes in the whole data set and the meal category ‘Cold appetizer’ is just one category out of ten. To correct this type of error, you need to add a GROUP BY meal\_category at the end of the statement. (Otherwise, your result in MySQL just doesn't make sense.)

The correct SELECT looks like this:

|  |
| --- |
| SELECT    meal\_category,    count(\*) AS total\_recipes  FROM recipes  GROUP BY meal\_category ; |

Here is a short explanation of what’s going on:

* Records are merged based on meal category. For example, desserts are one group, soups another, main dishes yet another, etc. The column meal\_category is specified after GROUP BY; it is also listed in SELECT.
* For each group, we are using COUNT(\*) to count the total number of recipes in that group.

I'm not going to dive deeply into the syntax here, but I would definitely suggest you read [GROUP BY IN SQL EXPLAINED](https://learnsql.com/blog/group-by-in-sql-explained/) or [USING GROUP BY IN SQL](https://learnsql.com/blog/what-is-group-by-in-sql/) for more details.

As you can see, the result is like we expected:

| **meal\_category** | **total\_recipes** |
| --- | --- |
| Bread and pastry | 7 |
| Cold appetizer | 6 |
| desserts | 20 |
| drinks | 7 |
| Main dishes | 20 |
| Salads | 8 |
| Side dishes | 12 |
| Soups | 17 |
| Warm appetizer | 3 |

Valid GROUP BY result

Tired of fetching data from SQL databases with R or Python to do basic reports? Learn how to do this right in the database with our [Creating Basic SQL Reports](https://learnsql.com/course/sql-basic-reporting?itm_source=lsqlBlog&itm_campaign=_default&itm_medium=text&itm_content=course-sql-basic-reporting-4) course.

### 2. Confusing WHERE and HAVING

Maybe you’d like to see only those meal categories that have more than 10 recipes. A  lot of beginners would write this query:

|  |
| --- |
| SELECT    meal\_category,    count(\*) AS total\_recipes  FROM recipes  WHERE count(\*) > 10  GROUP BY meal\_category ; |

This statement will return an error because you cannot use aggregate functions in a WHERE clause. WHERE is used with GROUP BY when you want to **filter rows before grouping them**.

In our example, we want to filter rows after grouping; in cases like this, we need to use the HAVING clause:

|  |
| --- |
| SELECT    meal\_category,    count(\*) AS total\_recipes  FROM recipes             GROUP BY meal\_category  HAVING count(\*) > 10  ; |

This misunderstanding about the difference between HAVING and WHERE is the second most common error with GROUP BY.

Let's clarify this difference with two more examples.

#### Example 1 -  How to Display Meal Categories with 1M+ Views

A statement that displays only categories with more than 1 million total page views can be written like this:

|  |
| --- |
| SELECT    meal\_category,    sum(no\_of\_views) AS total  FROM recipes  GROUP BY meal\_category  HAVING sum(no\_of\_views) >1000000; |

Here we are using HAVING because we want to filter records after they have been grouped. The result is presented below:

| **meal\_category** | **total** |
| --- | --- |
| desserts | 2969324 |
| Main dishes | 1323981 |
| Side dishes | 1662910 |
| Soups | 1100911 |

Example with HAVING

#### Example 2 – John’s Performance in Each Meal Category

This query extracts only John's recipes and calculates his performance:

|  |
| --- |
| SELECT    meal\_category,    sum(no\_of\_views) AS total  FROM recipes  WHERE author = ‘John’  GROUP BY meal\_category; |

We’re using WHERE because we need to filter records (so we only get John’s data) before we put the records into groups by meal category. Here is how the result looks:

| **meal\_category** | **total** |
| --- | --- |
| Bread and pastry | 50935 |
| desserts | 301869 |
| drinks | 147745 |
| Main dishes | 279934 |
| Salads | 88097 |
| Side dishes | 415864 |
| Soups | 393253 |
| Warm appetizer | 85570 |

John’s KPIs

HAVING and WHERE are nicely described in our articles [WHAT IS THE DIFFERENCE BETWEEN WHERE AND HAVING CLAUSES IN SQL?](https://learnsql.com/blog/difference-between-where-and-having-clauses-in-sql/) and [5 EXAMPLES OF GROUP BY](https://learnsql.com/blog/examples-of-sql-group-by/). If you would like to see more examples on this topic, I suggest starting there.

### 3. Listing a Column Inside SELECT but Not in GROUP BY

Now suppose you want to see the total number of views per meal\_category and author. We can do that? – ?we only need to add the author column to our previous query:

|  |
| --- |
| SELECT    meal\_category,    author,    sum(no\_of\_views) AS total  FROM recipes  GROUP BY meal\_category; |

Does this look okay to you?  Of course not; it will throw an error in most SQL engines. For example, Oracle will tell you[“**ERROR**: NOT A GROUP BY EXPRESSION”](https://learnsql.com/blog/not-a-group-by-expression-error/#:~:text=ORA%2D00979%20%E2%80%9C%20Not%20a%20GROUP,can%20be%20confusing%20to%20beginners.). Why this confusing error? What is missing here?

Well, the SQL engine doesn't know how to calculate the total for each author because we didn't include it in the GROUP BY clause; the attribute author is not listed inside the GROUP BY clause. This is another common error with GROUP BY.

Let's fix this query and run it one more time:

|  |
| --- |
| SELECT    meal\_category,    author,    sum(no\_of\_views) AS total  FROM recipes  GROUP BY meal\_category, author; |

The result is:

| **meal\_category** | **author** | **total** |
| --- | --- | --- |
| Bread and pastry | Dino | 53789 |
| Bread and pastry | John | 50935 |
| Bread and pastry | Marta | 52998 |
| Bread and pastry | Mary | 52904 |
| Bread and pastry | Patricia | 51451 |
| Bread and pastry | Tim | 106226 |
| ... | ... | ... |
| ... | ... | ... |
| ... | ... | ... |
| Soups | Mary | 125731 |
| Soups | Monte | 128356 |
| Soups | Patricia | 255574 |
| Soups | Tim | 132532 |
| Warm appetizer | John | 85570 |
| Warm appetizer | Lisa | 82960 |
| Warm appetizer | Mary | 87560 |

Now this looks okay. Remember, **unaggregated columns that are listed in**SELECT**must also be listed in**GROUP BY.  In our case, the unaggregated columns are meal\_category and author, which are now in SELECT and GROUP BY.

You don't list columns that are inside aggregate functions in GROUP BY. In our example, the column no\_of\_views is used in the aggregate function SUM() and thus is not listed in the GROUP BY clause.

 If you would like to learn more about this topic, check out our article [GROUP BY CLAUSE: HOW WELL DO YOU KNOW IT?](https://learnsql.com/blog/group-by-clause-how-well-do-you-know-it-2/). It explains why SELECTed columns need to appear in the GROUP BY clause. Also, [HOW TO FIX A ‘NOT A GROUP BY EXPRESSION’ ERROR](https://learnsql.com/blog/not-a-group-by-expression-error/) gives more examples related to this type of error.

### 4. Not Grouping by a Unique Key

Now let’s try something else. Suppose we want to get the average number of page views for each recipe author. The following query calculates the average total number of page views for each author using the author name:

|  |
| --- |
| SELECT    author,    avg(no\_of\_views)  FROM recipes  GROUP BY author; |

When you look at the result, you will notice that Lisa averages 116101.5 page views:

| **author** | **avg(NO\_OF\_VIEWS)** |
| --- | --- |
| Ally | 106545 |
| Dino | 94667.9091 |
| John | 88163.35 |
| Lisa | 116101.5 |
| Luke | 104591 |
| Marta | 119789.1667 |
| Mary | 101040.0588 |
| Monte | 84794 |
| Patricia | 81911.1333 |
| Tim | 76185.375 |

GROUP BY author – but names are not unique

However, we actually have two authors named Lisa in our table. When we group the results by the author column, both Lisas are averaged together. Why? Because we are using a non-unique column in the GROUP BY. This means that not all the grouping values have to be unique. If we want to see each Lisa’s average separately, we should add author\_id (a unique column) to the GROUP BY list:

|  |
| --- |
| SELECT    author, author\_id    avg(no\_of\_views)  FROM recipes  GROUP BY author, author\_id; |

Now we see how recipes from Lisa (id=11) are much more viewed than recipes by Lisa (id=5):

| **author** | **author\_id** | **avg(no\_of\_views)** |
| --- | --- | --- |
| Ally | 3 | 106545 |
| Dino | 7 | 94667.9091 |
| John | 2 | 88163.35 |
| Lisa | 5 | 85798 |
| Lisa | 11 | 146405 |
| Luke | 4 | 104591 |
| Marta | 1 | 119789.1667 |
| Mary | 6 | 101040.0588 |
| Monte | 9 | 84794 |
| Patricia | 10 | 81911.1333 |
| Tim | 8 | 76185.375 |

GROUP BY with author and author\_id

It is important to always think about grouping keys. Grouping values should be unique and must represent each group in the desired way. Otherwise, you’ll get inaccurate, confusing results and possibly a GROUP BY error.

### 5. Confusing COUNT(distinct) and COUNT(\*)

If you’re curious to see the total number of authors for each meal category, you can write a GROUP BY statement to calculate that. Let's use COUNT(\*) and retrieve the number of authors in each category:

|  |
| --- |
| SELECT    meal\_category,    count(\*)  FROM recipes  GROUP BY meal\_category; |

Here’s the result – but it’s not what you expected, is it?

| **meal\_category** | **count(\*)** |
| --- | --- |
| Bread and pastry | 7 |
| Cold appetizer | 6 |
| desserts | 20 |
| drinks | 7 |
| Main dishes | 20 |
| Salads | 8 |
| Side dishes | 12 |
| Soups | 17 |
| Warm appetizer | 3 |

This is the total number of recipes in each category, **not the total number of authors**. Why is that? Well, COUNT(\*) counts all the rows in each group. The table **recipe**contains information on a **recipe** level -  each record is one recipe. This query counts the recipes (rows) in each category, not the recipe authors.

One author can have many recipes in each category, so to get the information you want, you must **count distinct authors** (using COUNT(distinct author) instead of COUNT(\*))  inside each group. This is a very common GROUP BY error.

So, when should you use COUNT(\*), COUNT(expression) and COUNT(distinct expression)?

Let's take a look at an example:

|  |  |  |  |
| --- | --- | --- | --- |
| SELECT    meal\_category,    count(distinct author),    count(author),    count(\*)  FROM recipes  GROUP BY meal\_category; | | | |
| **meal\_category** | **count(distinct author)** | **count(author)** | **count(\*)** |
| Bread and pastry | 6 | 7 | 7 |
| Cold appetizer | 2 | 6 | 6 |
| desserts | 8 | 20 | 20 |
| drinks | 5 | 7 | 7 |
| Main dishes | 9 | 20 | 20 |
| Salads | 6 | 8 | 8 |
| Side dishes | 8 | 12 | 12 |
| Soups | 6 | 17 | 17 |
| Warm appetizer | 3 | 3 | 3 |

The difference between COUNT(\*) and COUNT(expression) is visible if we are doing calculations on a column that has some missing values. When missing values are present,  COUNT(\*) will count all the records in a group and COUNT(expression) will count only non-null values.

In the above example, COUNT(\*) and COUNT(author) give the exact same result because the author column doesn’t have any NULL values.

COUNT(distinct author) gives us the number of distinct authors for each category, which is not the same as COUNT(\*). For example, the cold appetizer meal category contains six recipes from two distinct authors. COUNT(\*) counts the number of recipes (records) in each category, while COUNT(distinct author) counts the number of distinct authors.

So, if you would like to display the total number of distinct authors per each meal category, use COUNT(distinct author). Here is the correct query:

|  |
| --- |
| SELECT    meal\_category,    count(distinct author)  FROM recipes  GROUP BY meal\_category;  GROUP BY meal\_category; |

For a more detailed explanation, see [WHAT IS THE DIFFERENCE BETWEEN COUNT(\*), COUNT(1), COUNT(COLUMN NAME), AND COUNT(DISTINCT COLUMN NAME)?](https://learnsql.com/blog/difference-between-count-distinct/)

### 6. Problems Using Aggregate Functions With NULLs

This is another ‘missing value’ problem. Let's say that you want to calculate the average total number of views from the previous month for each category. Your colleague calculated those figures, but they’d like you to double check the result.

Here is your query:

|  |
| --- |
| SELECT        meal\_category,        avg(no\_of\_views\_lst\_mth) as average,    FROM recipes  GROUP BY meal\_category; |

And what you get is ...

| **meal\_category** | **average** |
| --- | --- |
| Bread and pastry | 52274.8571 |
| Cold appetizer | 95584.2 |
| desserts | 144349.7222 |
| drinks | 72551.7143 |
| Main dishes | 61350.8889 |
| Salads | 90798.875 |
| Side dishes | 139765.25 |
| Soups | 64978.8824 |
| Warm appetizer | 78390.6667 |

The result looks okay and you are confident when it comes to the correctness of your query. However, your colleague got slightly different figures:

| **meal\_category** | **average** |
| --- | --- |
| Bread and pastry | 52274.8571 |
| Cold appetizer | 79653.5 |
| desserts | 129914.75 |
| drinks | 72551.7143 |
| Main dishes | 55215.8 |
| Salads | 90798.875 |
| Side dishes | 139765.25 |
| Soups | 64978.8824 |
| Warm appetizer | 78390.6667 |

What just happened? Why the different results?

 In a nutshell, the differing results arise from different interpretations of missing values.

The column no\_of\_views\_lst\_mth represents the number of total page views in the previous month. If a recipe was created in the current month, this column will be NULL for that row.

For example, Ally’s Banana Cheesecake recipe was written in the current month, so there are no statistics for the previous month:

| **meal\_category** | **name** | **author** | **no\_of\_views** | **no\_of\_views\_lst\_mth** | **author\_id** |
| --- | --- | --- | --- | --- | --- |
| desserts | Banana Cheesecake | Ally | 131944 | NULL | 3 |

Banana cheesecake was published in current month

Now, let's get back to those averages and their different results. Averages are calculated as the total sum of no\_of\_views\_lst\_mth divided by the total number of records. If you use the AVG() function and NULLs are present, the engine just ignores the NULLs and does calculations without them. This is what happened when you ran your query – the NULLs were omitted. In some cases, you’ll want to replace NULLs with 0 (because business logic dictates); this is what your colleague did, which produced slightly different figures. Here is your colleague’s query:

|  |
| --- |
| SELECT        meal\_category,             avg(CASE WHEN no\_of\_views\_lst\_mth is null              THEN 0              ELSE no\_of\_views\_lst\_mth END) AS average  FROM recipes  GROUP BY meal\_category; |

Notice how the averages from these two queries treat NULLs differently. For example, the ‘desserts’ category contains NULLs. Thus, the first query omits these rows and does not count them towards the total number of rows; this gives the value 144349.72. The second query replaces all NULLs with zero and counts these rows in the average, giving a smaller value  of 129914.75.

I would say that both queries could be valid, depending on how you want to calculate averages.

Let’s find out how to build basic SQL reports! Here’s our [Creating Basic SQL Reports](https://learnsql.com/course/sql-basic-reporting?itm_source=lsqlBlog&itm_campaign=_default&itm_medium=text&itm_content=course-sql-basic-reporting-1) interactive course.

## 7. Using COUNT(\*) with GROUP BY and a LEFT JOIN

Using GROUP BY with a LEFT JOIN statement can be quite confusing – especially with COUNT(). Let's see how COUNT(\*) and COUNT(expression) function in a LEFT JOIN.

Let’s suppose that someone in marketing has the following table, recipes\_campaigns. It contains information about the number of campaigns run on each meal category in the current month:

| **meal\_category** | **campaigns** |
| --- | --- |
| Bread and pastry | 2 |
| Cold appetizer | 1 |
| desserts | 3 |
| drinks | 0 |
| Main dishes | 3 |
| Salads | 1 |
| Side dishes | 2 |
| Soups | 3 |
| Warm appetizer | 0 |
| brunch | 1 |
| sandwiches | 0 |

recipes\_campaign

In addition to the data in **recipes\_campaigns**, the marketer also wants to see the number of recipes for each meal category. For that, we’ll need information from the **recipes** table. So let’s left join these two tables and calculate the number of recipes using COUNT(\*), like so:

|  |
| --- |
| SELECT        a.meal\_category,        count(\*),       FROM recipes\_campaigns a  LEFT JOIN recipes b ON a.meal\_category=b.meal\_category  GROUP BY a.meal\_category; |

Here’s the result:

| **meal\_category** | **count(\*)** |
| --- | --- |
| Bread and pastry | 7 |
| brunch | 1 |
| Cold appetizer | 6 |
| desserts | 20 |
| drinks | 7 |
| Main dishes | 20 |
| Salads | 8 |
| sandwiches | 1 |
| Side dishes | 12 |
| Soups | 17 |
| Warm appetizer | 3 |

This isn’t what we expected. The table **recipe** does not contain any recipes in the ‘brunch’ category, so why then did we get that 1 in the result? This happens because COUNT() is applied to the LEFT JOIN result! When you LEFT JOIN two tables, the ‘brunch’ category will be present in the output – even if there are no matching recipes or categories in the **recipe** table.

How can we fix this? If we use COUNT(expression) instead of COUNT(\*), we’ll get the result we want:

|  |
| --- |
| SELECT        a.meal\_category,        count(author\_id),  FROM recipes\_campaigns a  LEFT JOIN recipes b ON a.meal\_category=b.meal\_category  GROUP BY a.meal\_category; |

Once you run this, you get:

| **meal\_category** | **count(author\_id)** |
| --- | --- |
| Bread and pastry | 7 |
| brunch | 0 |
| Cold appetizer | 6 |
| desserts | 20 |
| drinks | 7 |
| Main dishes | 20 |
| Salads | 8 |
| sandwiches | 0 |
| Side dishes | 12 |
| Soups | 17 |
| Warm appetizer | 3 |

Here, COUNT(author\_id) counts only the non-NULL values in author\_id **after** the LEFT JOIN is performed. There is no author\_id value for the ‘brunch’ category; in other words, it’s NULL and the result for that category is 0.

## You Can Solve GROUP BY Errors!

Through several examples, we’ve explored [GROUP BY](https://learnsql.com/blog/what-is-group-by-in-sql/) and the most common errors that beginners often make. I hope that now you have a better sense of how GROUP BY works and what’s causing those weird errors or confusing results.