@Auther: Gabir N. Yousef
@Date: 3rd of Sep 2019

SQL Aggregation

NULLs

Notice that **NULL**s are different than a zero - they are cells where data does not exist.

When identifying **NULL**s in a **WHERE** clause, we write **IS NULL** or **IS NOT NULL**. We don't use `=`, because **NULL** isn't considered a value in SQL. Rather, it is a property of the data.

NULLs - Expert Tip

There are two common ways in which you are likely to encounter **NULL**s:

- **NULL**s frequently occur when performing a **LEFT** or **RIGHT JOIN**. You saw in the last lesson - when some rows in the left table of a left join are not matched with rows in the right table, those rows will contain some **NULL** values in the result set.

- **NULL**s can also occur from simply missing data in our database.

COUNT the Number of Rows in a Table

Try your hand at finding the number of rows in each table. Here is an example of finding all the rows in the **accounts** table.

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

But we could have just as easily chosen a column to drop into the aggregation function:

```
SELECT COUNT(accounts.id)
FROM accounts;
```

These two statements are equivalent, but this isn't always the case, which we will see in the next video.

Notice that **COUNT** does not consider rows that have **NULL** values. Therefore, this can be useful for quickly identifying which rows have missing data. You will learn **GROUP BY** in an upcoming concept, and then each of these aggregators will become much more useful.

SUM

Unlike **COUNT**, you can only use **SUM** on numeric columns. However, **SUM** will ignore **NULL** values, as do the other aggregation functions you will see in the upcoming lessons.

Aggregation Reminder

An important thing to remember: **aggregators only aggregate vertically - the values of a column**.

Notice that **MIN** and **MAX** are aggregators that again ignore **NULL** values. Check the expert tip below for a cool trick with **MAX** & **MIN**.

Expert Tip

Functionally, **MIN** and **MAX** are similar to **COUNT** in that they can be used on non-numerical columns. Depending on the column type, **MIN** will return the lowest number, earliest date, or non-numerical value as early in the alphabet as possible. As you might suspect, **MAX** does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to “Z.”

Similar to other software **AVG** returns the mean of the data - that is the sum of all of the values in the column divided by the number of values in a column. This aggregate function again ignores the **NULL**values in both the numerator and the denominator.

If you want to count **NULL**s as zero, you will need to use **SUM** and **COUNT**. However, this is probably not a good idea if the **NULL** values truly just represent unknown values for a cell.

MEDIAN

One quick note that a median might be a more appropriate measure of center for this data, but finding the median happens to be a pretty difficult thing to get using SQL alone — so difficult that finding a median is occasionally asked as an interview question.

GROUP BY

The key takeaways here:

- **GROUP BY** can be used to aggregate data within subsets of the data. For example, grouping for different accounts, different regions, or different sales representatives.
- Any column in the **SELECT** statement that is not within an aggregator must be in the **GROUP BY** clause.
- The **GROUP BY** always goes between **WHERE** and **ORDER BY**.
- **ORDER BY** works like **SORT** in spreadsheet software.

GROUP BY - Expert Tip

Before we dive deeper into aggregations using **GROUP BY** statements, it is worth noting that SQL evaluates the aggregations before the **LIMIT** clause. If you don't group by any columns, you'll get a 1-row result—no problem there. If you group by a column with enough unique values that it exceeds the **LIMIT** number, the aggregates will be calculated, and then some rows will simply be omitted from the results.

This is actually a nice way to do things because you know you're going to get the correct aggregates. If SQL cuts the table down to 100 rows, then performed the aggregations, your results would be substantially different. The above query's results exceed 100 rows, so it's a perfect example. In the next concept, use the SQL environment to try removing the **LIMIT** and running it again to see what changes.

Key takeaways:

- You can **GROUP BY** multiple columns at once, as we showed here. This is often useful to aggregate across a number of different segments.

- The order of columns listed in the **ORDER BY** clause does make a difference. You are ordering the columns from left to right.

GROUP BY - Expert Tips

- The order of column names in your **GROUP BY** clause doesn't matter—the results will be the same regardless. If we run the same query and reverse the order in the **GROUP BY** clause, you can see we get the same results.

- As with **ORDER BY**, you can substitute numbers for column names in the **GROUP BY** clause. It's generally recommended to do this only when you're grouping many columns, or if something else is causing the text in the GROUP BY clause to be excessively long.

- A reminder here that any column that is not within an aggregation must show up in your GROUP BY statement. If you forget, you will likely get an error. However, in the off chance that your query does work, you might not like the results!

DISTINCT

DISTINCT is always used in **SELECT** statements, and it provides the unique rows for all columns written in the **SELECT** statement. Therefore, you only use **DISTINCT** once in any particular **SELECT**statement.

You could write:

```
SELECT DISTINCT column1, column2, column3
FROM table1;
```

which would return the unique (or **DISTINCT**) rows across all three columns.

You would **not** write:

```
SELECT DISTINCT column1, DISTINCT column2, DISTINCT column3
FROM table1;
```

You can think of **DISTINCT** the same way you might think of the statement "unique".

DISTINCT - Expert Tip

It's worth noting that using **DISTINCT**, particularly in aggregations, can slow your queries down quite a bit.

HAVING

HAVING is the "clean" way to filter a query that has been aggregated, but this is also commonly done using a subquery. Essentially, any time you want to perform a **WHERE** on an element of your query that was created by an aggregate, you need to use **HAVING** instead.

Notes about **WHERE** and **HAVING**.

- **WHERE** subsets the returned data based on a logical condition.
- **WHERE** appears after the **FROM**, **JOIN**, and **ON** clauses, but before **GROUP BY**.
- **HAVING** appears after the **GROUP BY** clause, but before the **ORDER BY** clause.
- **HAVING** is like **WHERE**, but it works on logical statements involving aggregations.

GROUPing **BY** a date column is not usually very useful in SQL, as these columns tend to have transaction data down to a second. Keeping date information at such a granular data is both a blessing and a curse, as it gives really precise information (a blessing), but it makes grouping information together directly difficult (a curse).

Lucky for us, there are a number of built in SQL functions that are aimed at helping us improve our experience in working with dates.

Here we saw that dates are stored in year, month, day, hour, minute, second, which helps us in truncating. In the next concept, you will see a number of functions we can use in SQL to take advantage of this functionality.

DATE_TRUNC.

DATE_TRUNC allows you to truncate your date to a particular part of your date-time column. Common trunctions are `day`, `month`, and `year`. Here is a great blog post by Mode Analytics on the power of this function.

DATE_PART can be useful for pulling a specific portion of a date, but notice pulling `month` or day of the week (`dow`) means that you are no longer keeping the years in order. Rather you are grouping for certain components regardless of which year they belonged in.

For additional functions you can use with dates, check out the documentation here, but the **DATE_TRUNC** and **DATE_PART** functions definitely give you a great start!

You can reference the columns in your select statement in **GROUP BY** and **ORDER BY** clauses with numbers that follow the order they appear in the select statement. For example

```
SELECT standard_qty, COUNT(*)
FROM orders
```

GROUP BY 1 *(this 1 refers to standard_qty since it is the first of the columns included in the select statement)*

ORDER BY 1 *(this 1 refers to standard_qty since it is the first of the columns included in the select statement)*

CASE - Expert Tip

- The CASE statement always goes in the SELECT clause.
- CASE must include the following components: WHEN, THEN, and END. ELSE is an optional component to catch cases that didn't meet any of the other previous CASE conditions.
- You can make any conditional statement using any conditional operator (like WHERE) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.
- You can include multiple WHEN statements, as well as an ELSE statement again, to deal with any unaddressed conditions.

Example

In a quiz question in the previous Basic SQL lesson, you saw this question:

1. Create a column that divides the `standard_amt_usd` by the `standard_qty` to find the unit price for standard paper for each order. Limit the results to the first 10 orders, and include the `id` and `account_id` fields. **NOTE - you will be thrown an error with the correct solution to this question. This is for a division by zero. You will learn how to get a solution without an error to this query when you learn about CASE statements in a later section.**

Let's see how we can use the **CASE** statement to get around this error.

```
SELECT id, account_id, standard_amt_usd/standard_qty AS unit_price
FROM orders
LIMIT 10;
```

Now, let's use a **CASE** statement. This way any time the **standard_qty** is zero, we will return 0, and otherwise we will return the **unit_price**.

```
SELECT account_id, CASE WHEN standard_qty = 0 OR standard_qty IS NULL THEN 0
                        ELSE standard_amt_usd/standard_qty END AS unit_price
FROM orders
LIMIT 10;
```

Now the first part of the statement will catch any of those division by zero values that were causing the error, and the other components will compute the division as necessary. You will notice, we essentially charge all of our accounts 4.99 for standard paper. It makes sense this doesn't fluctuate, and it is more accurate than adding 1 in the denominator like our quick fix might have been in the earlier lesson.

You can try it yourself using the environment below.

This one is pretty tricky. Try running the query yourself to make sure you understand what is happening. The next concept will give you some practice writing **CASE** statements on your own. In this video, we showed that getting the same information using a **WHERE** clause means only being able to get one set of data from the **CASE** at a time.

There are some advantages to separating data into separate columns like this depending on what you want to do, but often this level of separation might be easier to do in another programming language - rather than with SQL.

RECAP

Each of the sections has been labeled to assist if you need to revisit a particular topic. Intentionally, the solutions for a particular section are actually not in the labeled section, because my hope is this will force you to practice if you have a question about a particular topic we covered.

You have now gained a ton of useful skills associated with **SQL**. The combination of **JOINs** and **Aggregations** are one of the reasons **SQL** is such a powerful tool.

If there was a particular topic you struggled with, I suggest coming back and revisiting the questions with a fresh mind. The more you practice the better, but you also don't want to get stuck on the same problem for an extended period of time!