https://classroom.udacity.com/courses/ud002-bert/lessons/76a484da-1f2e-4886-ba2c-684bb30e267d/concepts/46f30a9c-fe2e-49d5-89fb-f8984212a176

SQL AGGREGATIONS LN 30 (1-33)

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 :

```
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;
```

Notice that COUNT does not consider rows that have NULL values. Therefore, this can be useful for quickly identifying which rows have missing data.

SUM Solutions

1. **Find the total amount of poster_qty paper ordered in the orders table.**

   ```
   SELECT SUM(poster_qty) AS total_poster_sales
   FROM orders;
   ```

2. **Find the total amount of standard_qty paper ordered in the orders table.**

   ```
   SELECT SUM(standard_qty) AS total_standard_sales
   FROM orders;
   ```

3. **Find the total dollar amount of sales using the total_amt_usd in the orders table.**

   ```
   SELECT SUM(total_amt_usd) AS total_dollar_sales
   FROM orders;
   ```

4. **Find the total amount for each individual order that was spent on standard and gloss paper in the orders table. This should give a dollar amount for each order in the table. Notice, this solution did not use an aggregate.**

   ```
   SELECT standard_amt_usd + gloss_amt_usd AS total_standard_gloss
   FROM orders;
   ```

5. **Though the price/standard_qty paper varies from one order to the next. I would like this ratio across all of the sales made in the orders table.Notice, this solution used both an aggregate and our mathematical operators**

   ```
   SELECT SUM(standard_amt_usd)/SUM(standard_qty) AS
   standard_price_per_unit
   FROM orders;
   ```

MIN & MAX :

Notice that here we were simultaneously obtaining the **MIN** and **MAX** number of orders of each paper type. However, you could run each individually.

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."

AVERAGE :

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 - Expert Tip

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.

Questions: MIN, MAX, & AVERAGE

Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

1. When was the earliest order ever placed? You only need to return the date.

   ```
   SELECT MIN(occurred_at)
   FROM orders;
   ```

2. Try performing the same query as in question 1 without using an aggregation function.

   ```
   SELECT occurred_at
   FROM orders
   ORDER BY occurred_at
   LIMIT 1;
   ```

3. When did the most recent (latest) **web_event** occur?

   ```
   SELECT MAX(occurred_at)
   FROM web_events;
   ```

4. Try to perform the result of the previous query without using an aggregation function.

   ```
   SELECT occurred_at
   FROM web_events
   ORDER BY occurred_at DESC
   LIMIT 1;
   ```

5. Find the mean (**AVERAGE**) amount spent per order on each paper type, as well as the mean amount of each paper type purchased per order. Your final answer should have 6 values - one for each paper type for the average number of sales, as well as the average amount.

```
SELECT AVG(standard_qty) mean_standard, AVG(gloss_qty) mean_gloss,
           AVG(poster_qty) mean_poster, AVG(standard_amt_usd)
mean_standard_usd,
           AVG(gloss_amt_usd) mean_gloss_usd, AVG(poster_amt_usd)
mean_poster_usd
FROM orders;
```

6. Via the video, you might be interested in how to calculate the MEDIAN. Though this is more advanced than what we have covered so far try finding - what is the MEDIAN **total_usd** spent on all **orders**?

```
SELECT *
FROM (SELECT total_amt_usd
      FROM orders
      ORDER BY total_amt_usd
      LIMIT 3457) AS Table1
ORDER BY total_amt_usd DESC
LIMIT 2;
```

`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.

NOTE : GROUP BY is used to group a selected set of rows into a set of summary rows by the values of one or more columns or expressions. It is always used in conjunction with one or more columns.

https://www.youtube.com/watch?v=FKSSOpQe5Jc

Solutions: GROUP BY

1. **Which account (by name) placed the earliest order? Your solution should have the account name and the date of the order.**

```
SELECT a.name, o.occurred_at
FROM accounts a
JOIN orders o
ON a.id = o.account_id
ORDER BY occurred_at
LIMIT 1;
```

2. **Find the total sales in usd for each account. You should include two columns - the total sales for each company's orders in usd and the company name.**

```
SELECT a.name, SUM(total_amt_usd) total_sales
FROM orders o
JOIN accounts a
ON a.id = o.account_id
GROUP BY a.name;
```

3. **Via what channel did the most recent (latest) web_event occur, which account was associated with this web_event? Your query should return only three values - the date, channel, and account name.**

```
SELECT w.occurred_at, w.channel, a.name
FROM web_events w
JOIN accounts a
ON w.account_id = a.id
ORDER BY w.occurred_at DESC
LIMIT 1;
```

4. **Find the total number of times each type of channel from the web_events was used. Your final table should have two columns - the channel and the number of times the channel was used.**

```
SELECT w.channel, COUNT(*)
FROM web_events w
GROUP BY w.channel
```

5. **Who was the primary contact associated with the earliest web_event?**

```
SELECT a.primary_poc
FROM web_events w
JOIN accounts a
ON a.id = w.account_id
ORDER BY w.occurred_at
```

```
LIMIT 1;
```

6. **What was the smallest order placed by each account in terms of total usd. Provide only two columns - the account name and the total usd. Order from smallest dollar amounts to largest.**

```
SELECT a.name, MIN(total_amt_usd) smallest_order
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.name
ORDER BY smallest_order;
```

7. **Find the number of sales reps in each region. Your final table should have two columns - the region and the number of sales_reps. Order from fewest reps to most reps.**

```
SELECT r.name, COUNT(*) num_reps
FROM region r
JOIN sales_reps s
ON r.id = s.region_id
GROUP BY r.name
ORDER BY num_reps;
```

`DISTINCT : If you want to group by some columns but you don’t necessarily want to include any aggregations you can use DISTINCT instead.`

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.

Solutions: DISTINCT

Use DISTINCT to test if there are any accounts associated with more than one region?

The below two queries have the same number of resulting rows (351), so we know that every account is associated with only one region. If each account was associated with more than one region, the first query should have returned more rows than the second query.

```
SELECT a.id as "account id", r.id as "region id",
a.name as "account name", r.name as "region name"
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id;
```

And

```
SELECT DISTINCT id, name
FROM accounts;
```

`HAVING :` https://www.youtube.com/watch?v=15ndzPfa6-E

HAVING is used to get that exact count like

HAVING SUM>1000

Difference bw HAVING and WHERE:

HAVING can be used along with aggregate operators but WHERE cannot be used with aggregate operators.

How many of the sales reps have more than 5 accounts that they manage?

```
SELECT s.id, s.name, COUNT(*) num_accounts
FROM accounts a
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.id, s.name
HAVING COUNT(*) > 5
ORDER BY num_accounts;
```

The first function you are introduced to in working with dates is 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!

Solutions: Working With DATEs

1.Find the sales in terms of total dollars for all orders in each `year`, ordered from greatest to least. Do you notice any trends in the yearly sales totals?

```
SELECT DATE_PART('year', occurred_at) ord_year,  SUM(total_amt_usd)
total_spent
 FROM orders
 GROUP BY 1
 ORDER BY 2 DESC;
```

2.**When we look at the yearly totals, you might notice that 2013 and 2017 have much smaller totals than all other years. If we look further at the monthly data, we see that for `2013` and `2017` there is only one month of sales for each of these years (12 for 2013 and 1 for 2017). Therefore, neither of these are evenly represented. Sales have been increasing year over year, with 2016 being the largest sales to date. At this rate, we might expect 2017 to have the largest sales.**

Which month did Parch & Posey have the greatest sales in terms of total dollars? Are all months evenly represented by the dataset?

In order for this to be 'fair', we should remove the sales from 2013 and 2017. For the same reasons as discussed above.

```
SELECT DATE_PART('month', occurred_at) ord_month, SUM(total_amt_usd)
total_spent
FROM orders
WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
GROUP BY 1
ORDER BY 2 DESC;
```

The greatest sales amounts occur in December (12).

3.Which year did Parch & Posey have the greatest sales in terms of total number of orders? Are all years evenly represented by the dataset?

```
SELECT DATE_PART('year', occurred_at) ord_year,  COUNT(*) total_sales
FROM orders
GROUP BY 1
ORDER BY 2 DESC;
```

Again, 2016 by far has the most amount of orders, but again 2013 and 2017 are not evenly represented to the other years in the dataset.

4.Which month did Parch & Posey have the greatest sales in terms of total number of orders? Are all months evenly represented by the dataset?

```
SELECT DATE_PART('month', occurred_at) ord_month, COUNT(*) total_sales
FROM orders
WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
GROUP BY 1
ORDER BY 2 DESC;
```

December still has the most sales, but interestingly, November has the second most sales (but not the most dollar sales. To make a fair comparison from one month to another 2017 and 2013 data were removed.

5.In which month of which year did `Walmart` spend the most on gloss paper in terms of dollars?

```
SELECT DATE_TRUNC('month', o.occurred_at) ord_date, SUM(o.gloss_amt_usd)
tot_spent
FROM orders o
```

```
JOIN accounts a
ON a.id = o.account_id
WHERE a.name = 'Walmart'
GROUP BY 1
ORDER BY 2 DESC
LIMIT 1;
```

May 2016 was when Walmart spent the most on gloss paper.

6.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.**

WE GO FOR CASE STATEMENTS INORDER TO CREATE A DERIVED COLUMN

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.