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.

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**. If you want to perform a calculation across rows, you would do this with simple arithmetic.

We saw this in the first lesson if you need a refresher, but the quiz in the next concept should assure you still remember how to aggregate across rows.

Aggregation Questions

Use the **SQL** environment below to find the solution for each of the following questions. If you get stuck or want to check your answers, you can find the answers at the top of the next concept.

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

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

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

4. Find the total amount spent on **standard_amt_usd** and **gloss_amt_usd** paper for each order in the orders table. This should give a dollar amount for each order in the table.

5. Find the **standard_amt_usd** per unit of **standard_qty** paper. Your solution should use both an aggregation and a mathematical operator.

SUM Solutions

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

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

1.

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

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

2.

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;
```

3.

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;
```

4.

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
```

5. `FROM orders;`

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.”
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.

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

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

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

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.

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**?

Solutions: MIN, MAX, & AVG

SEND FEEDBACK

Solutions: MIN, MAX, and AVERAGE

When was the earliest order ever placed?

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

1.

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;
```

2.

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

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

3.

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;
```

4.

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;
```

5.

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**? Note, this is more advanced than the topics we have covered thus far to build a general solution, but we can hard code a solution in the following way.

```
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;
```

6.

   Since there are 6912 orders - we want the average of the 3457 and 3456 order amounts when ordered. This is the average of 2483.16 and 2482.55. This gives the median of **2482.855**. This obviously isn't an ideal way to compute. If we obtain new orders, we would have to change the limit. SQL didn't even calculate the median for us. The above used a SUBQUERY, but you could use any method to find the two necessary values, and then you just need the average of them.

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.

GROUP BY Note

Now that you have been introduced to **JOIN**s, **GROUP BY**, and aggregate functions, the real power of **SQL** starts to come to life. Try some of the below to put your skills to the test!

Questions: GROUP BY

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.

One part that can be difficult to recognize is when it might be easiest to use an aggregate or one of the other SQL functionalities. Try some of the below to see if you can differentiate to find the easiest solution.

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

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**.

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**.

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.

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

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.

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.

Solutions: GROUP BY

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;
```

1.

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;
```

2. 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;
```

3. 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
```

4. 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;
```

5.

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;
```

6.
   Sort of strange we have a bunch of orders with no dollars. We might want to look into those.

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
```

7.
```
ORDER BY num_reps;
```

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!

Questions: GROUP BY Part II

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. For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four columns - one for the account **name** and one for the average quantity purchased for each of the paper types for each account.

2. For each account, determine the average amount spent per order on each paper type. Your result should have four columns - one for the account **name** and one for the average amount spent on each paper type.

3. Determine the number of times a particular **channel** was used in the **web_events** table for each **sales rep**. Your final table should have three columns - the **name of the sales rep**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

4. Determine the number of times a particular **channel** was used in the **web_events** table for each **region**. Your final table should have three columns - the **region name**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

Solutions: GROUP BY Part II

For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four columns - one for the account **name** and one for the average spent on each of the paper types.

```
SELECT a.name, AVG(o.standard_qty) avg_stand, AVG(o.gloss_qty) avg_gloss,
AVG(o.poster_qty) avg_post
FROM accounts a
JOIN orders o
```

```
ON a.id = o.account_id
GROUP BY a.name;
```

1.

For each account, determine the average amount spent per order on each paper type. Your result should have four columns - one for the account **name** and one for the average amount spent on each paper type.

```
SELECT a.name, AVG(o.standard_amt_usd) avg_stand, AVG(o.gloss_amt_usd)
avg_gloss, AVG(o.poster_amt_usd) avg_post
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.name;
```

2.

Determine the number of times a particular **channel** was used in the **web_events** table for each **sales rep**. Your final table should have three columns - the **name of the sales rep**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

```
SELECT s.name, w.channel, COUNT(*) num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.name, w.channel
ORDER BY num_events DESC;
```

3.

Determine the number of times a particular **channel** was used in the **web_events** table for each **region**. Your final table should have three columns - the **region name**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.

```
SELECT r.name, w.channel, COUNT(*) num_events
FROM accounts a
JOIN web_events w
ON a.id = w.account_id
JOIN sales_reps s
ON s.id = a.sales_rep_id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name, w.channel
```

4. `ORDER BY num_events DESC;`

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.

Questions: DISTINCT

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

2. Have any **sales reps** worked on more than one account?

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;
```

1.

Have any **sales reps** worked on more than one account?

Actually all of the sales reps have worked on more than one account. The fewest number of accounts any sales rep works on is 3. There are 50 sales reps, and they all have more than one account. Using **DISTINCT** in the second query assures that all of the sales reps are accounted for in the first query.

```
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
ORDER BY num_accounts;
```

and

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

2.

HAVING - Expert Tip

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.

QUIZ QUESTION

Often there is confusion about the difference between **WHERE** and **HAVING**. Select all the statements that are true regarding **HAVING** and **WHERE** statements.

- **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.

SUBMIT

Questions: HAVING

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

2. How many **accounts** have more than 20 orders?

3. Which account has the most orders?

4. Which accounts spent more than 30,000 usd total across all orders?
5. Which accounts spent less than 1,000 usd total across all orders?
6. Which account has spent the most with us?
7. Which account has spent the least with us?
8. Which accounts used `facebook` as a **channel** to contact customers more than 6 times?
9. Which account used `facebook` most as a **channel**?
10. Which channel was most frequently used by most accounts?

Solutions: HAVING

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;
```

and technically, we can get this using a **SUBQUERY** as shown below. This same logic can be used for the other queries, but this will not be shown.

```
SELECT COUNT(*) num_reps_above5

FROM(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) AS Table1;
```

1.

How many **accounts** have more than 20 orders?

```
SELECT a.id, a.name, COUNT(*) num_orders

FROM accounts a

JOIN orders o

ON a.id = o.account_id

GROUP BY a.id, a.name
```

```
HAVING COUNT(*) > 20

ORDER BY num_orders;
```

2.

Which account has the most orders?

```
SELECT a.id, a.name, COUNT(*) num_orders

FROM accounts a

JOIN orders o

ON a.id = o.account_id

GROUP BY a.id, a.name

ORDER BY num_orders DESC

LIMIT 1;
```

3.

How many accounts spent more than 30,000 usd total across all orders?

```
SELECT a.id, a.name, SUM(o.total_amt_usd) total_spent

FROM accounts a

JOIN orders o

ON a.id = o.account_id

GROUP BY a.id, a.name

HAVING SUM(o.total_amt_usd) > 30000

ORDER BY total_spent;
```

4.

How many accounts spent less than 1,000 usd total across all orders?

```
SELECT a.id, a.name, SUM(o.total_amt_usd) total_spent
```

```
FROM accounts a

JOIN orders o

ON a.id = o.account_id

GROUP BY a.id, a.name

HAVING SUM(o.total_amt_usd) < 1000

ORDER BY total_spent;
```

5.

Which account has spent the most with us?

```
SELECT a.id, a.name, SUM(o.total_amt_usd) total_spent

FROM accounts a

JOIN orders o
```

```
ON a.id = o.account_id

GROUP BY a.id, a.name

ORDER BY total_spent DESC

LIMIT 1;
```

6.

Which account has spent the least with us?

```
SELECT a.id, a.name, SUM(o.total_amt_usd) total_spent

FROM accounts a

JOIN orders o

ON a.id = o.account_id

GROUP BY a.id, a.name
```

```
ORDER BY total_spent
```

```
LIMIT 1;
```

7.

Which accounts used `facebook` as a **channel** to contact customers more than 6 times?

```
SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel
```

```
FROM accounts a
```

```
JOIN web_events w
```

```
ON a.id = w.account_id
```

```
GROUP BY a.id, a.name, w.channel
```

```
HAVING COUNT(*) > 6 AND w.channel = 'facebook'
```

```
ORDER BY use_of_channel;
```

8.

Which account used `facebook` most as a **channel**?

```
SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel

FROM accounts a

JOIN web_events w

ON a.id = w.account_id

WHERE w.channel = 'facebook'

GROUP BY a.id, a.name, w.channel

ORDER BY use_of_channel DESC

LIMIT 1;
```

9.

Note: This query above only works if there are no ties for the account that used facebook the most. It is a best practice to use a larger limit number first such as 3 or 5 to see if there are ties before using LIMIT 1.

Which channel was most frequently used by most accounts?

```
SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel

FROM accounts a

JOIN web_events w

ON a.id = w.account_id

GROUP BY a.id, a.name, w.channel

ORDER BY use_of_channel DESC

LIMIT 10;
```

10. All of the top 10 are `direct`.

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.

In this link you can find the formatting of dates around the world, as referenced in the video.
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!

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)*

Questions: Working With DATEs

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. 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?

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

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?

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?

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

Solutions: Working With DATEs

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;
```

1.

   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;
```

2. The greatest sales amounts occur in December (12).

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;
```

3. 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.

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;
```

4.

   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.

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;
```

5.

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

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.

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.

Solutions: CASE

Write a query to display for each order, the account ID, total amount of the order, and the level of the order - ‘Large’ or ’Small’ - depending on if the order is $3000 or more, or less than $3000.

```
SELECT account_id, total_amt_usd,

CASE WHEN total_amt_usd > 3000 THEN 'Large'

ELSE 'Small' END AS order_level

FROM orders;
```

1.

Write a query to display the number of orders in each of three categories, based on the `total` number of items in each order. The three categories are: 'At Least 2000', 'Between 1000 and 2000' and 'Less than 1000'.

```
SELECT CASE WHEN total >= 2000 THEN 'At Least 2000'

   WHEN total >= 1000 AND total < 2000 THEN 'Between 1000 and 2000'

   ELSE 'Less than 1000' END AS order_category,

COUNT(*) AS order_count

FROM orders

GROUP BY 1;
```

2.

We would like to understand 3 different branches of customers based on the amount associated with their purchases. The top branch includes anyone with a Lifetime Value (total sales of all orders) `greater than 200,000` usd. The second branch is between `200,000 and 100,000` usd. The lowest branch is anyone `under 100,000` usd. Provide a table that includes the **level** associated with each **account**. You should provide the **account name**, the **total sales of all orders** for the customer, and the

level. Order with the top spending customers listed first.

```
SELECT a.name, SUM(total_amt_usd) total_spent,

     CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'

     WHEN  SUM(total_amt_usd) > 100000 THEN 'middle'

     ELSE 'low' END AS customer_level

FROM orders o

JOIN accounts a

ON o.account_id = a.id

GROUP BY a.name

ORDER BY 2 DESC;
```

3.

We would now like to perform a similar calculation to the first, but we want to obtain the total amount spent by customers only in `2016` and `2017`. Keep the same **level**s as in the previous question. Order with the top spending customers listed first.

```
SELECT a.name, SUM(total_amt_usd) total_spent,

     CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'

     WHEN  SUM(total_amt_usd) > 100000 THEN 'middle'

     ELSE 'low' END AS customer_level

FROM orders o

JOIN accounts a

ON o.account_id = a.id

WHERE occurred_at > '2015-12-31'

GROUP BY 1
```

```
ORDER BY 2 DESC;
```

4.

We would like to identify top performing **sales reps**, which are sales reps associated with more than 200 orders. Create a table with the **sales rep name**, the total number of orders, and a column with `top` or `not` depending on if they have more than 200 orders. Place the top sales people first in your final table.

```
SELECT s.name, COUNT(*) num_ords,

     CASE WHEN COUNT(*) > 200 THEN 'top'

     ELSE 'not' END AS sales_rep_level

FROM orders o

JOIN accounts a

ON o.account_id = a.id

JOIN sales_reps s
```

```
ON s.id = a.sales_rep_id
```

```
GROUP BY s.name
```

```
ORDER BY 2 DESC;
```

5.

   It is worth mentioning that this assumes each name is unique - which has been done a few times. We otherwise would want to break by the name and the id of the table.

The previous didn't account for the middle, nor the dollar amount associated with the sales. Management decides they want to see these characteristics represented as well. We would like to identify top performing **sales reps**, which are sales reps associated with more than `200` orders or more than `750000` in total sales. The `middle` group has any **rep** with more than 150 orders or `500000` in sales. Create a table with the **sales rep name**, the total number of orders, total sales across all orders, and a column with `top`, `middle`, or `low` depending on this criteria. Place the top sales people based on dollar amount of sales first in your final table.

```
SELECT s.name, COUNT(*), SUM(o.total_amt_usd) total_spent,
     CASE WHEN COUNT(*) > 200 OR SUM(o.total_amt_usd) > 750000 THEN 'top'

     WHEN COUNT(*) > 150 OR SUM(o.total_amt_usd) > 500000 THEN 'middle'

     ELSE 'low' END AS sales_rep_level

FROM orders o

JOIN accounts a

ON o.account_id = a.id

JOIN sales_reps s

ON s.id = a.sales_rep_id

GROUP BY s.name

ORDER BY 3 DESC;
```

6.

You might see a few upset sales people by this criteria!

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!