What this lesson is about...

Up to this point you have learned a lot about working with data using SQL. This lesson will focus on three topics:

1. Subqueries
2. Table Expressions
3. Persistent Derived Tables

Both **subqueries** and **table expressions** are methods for being able to write a query that creates a table, and then write a query that interacts with this newly created table. Sometimes the question you are trying to answer doesn't have an answer when working directly with existing tables in database.

However, if we were able to create new tables from the existing tables, we know we could query these new tables to answer our question. This is where the queries of this lesson come to the rescue.

If you can't yet think of a question that might require such a query, don't worry because you are about to see a whole bunch of them!

Whenever we need to use existing tables to create a new table that we then want to query again, this is an indication that we will need to use some sort of **subquery**. In the next couple of concepts, we will walk through an example together. Then you will get some practice tackling some additional problems on your own.

NEXT

Your First Subquery

The first time you write a subquery it might seem really complex. Let's try breaking it down into its different parts.

If you get stuck look again at the video above. We want to find the average number of events for each day for each channel. The first table will provide us the number of events for each day and channel, and then we will need to average these values together using a second query.

You try solving this yourself.

Tasks to complete.

Task List

- Use the test environment below to find the number of events that occur for each day for each channel.
- Quiz 1
- Now create a subquery that simply provides all of the data from your first query.
- Quiz 2
- Now find the average number of events for each channel. Since you broke out by day earlier, this is giving you an average per day.
- Quiz 3

Solutions to Your First Subquery

First, we needed to group by the day and channel. Then ordering by the number of events (the third column) gave us a quick way to answer the first question.

```sql
SELECT DATE_TRUNC('day',occurred_at) AS day,
   channel, COUNT(*) as events
FROM web_events
GROUP BY 1,2
ORDER BY 3 DESC;
```

1.

Here you can see that to get the entire table in question 1 back, we included an `*` in our **SELECT** statement. You will need to be sure to alias your table.

```sql
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
           channel, COUNT(*) as events
     FROM web_events
     GROUP BY 1,2
     ORDER BY 3 DESC) sub;
```

2.

Finally, here we are able to get a table that shows the average number of events a day for each channel.

```sql
SELECT channel, AVG(events) AS average_events
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
             channel, COUNT(*) as events
      FROM web_events
      GROUP BY 1,2) sub
GROUP BY channel
ORDER BY 2 DESC;
```

3.

Subquery Formatting

When writing **Subqueries**, it is easy for your query to look incredibly complex. In order to assist your reader, which is often just yourself at a future date, formatting SQL will help with understanding your code.

The important thing to remember when using subqueries is to provide some way for the reader to easily determine which parts of the query will be executed together. Most people do this by indenting the subquery in some way - you saw this with the solution blocks in the previous concept.

The examples in this class are indented quite far—all the way to the parentheses. This isn't practical if you nest many subqueries, but in general, be thinking about how to write your queries in a readable way. Examples of the same query written multiple different ways is provided below. You will see that some are much easier to read than others.

Badly Formatted Queries

Though these poorly formatted examples will execute the same way as the well formatted examples, they just aren't very friendly for understanding what is happening!

Here is the first, where it is impossible to decipher what is going on:

```
SELECT * FROM (SELECT DATE_TRUNC('day',occurred_at) AS day, channel, COUNT(*)
as events FROM web_events GROUP BY 1,2 ORDER BY 3 DESC) sub;
```

This second version, which includes some helpful line breaks, is easier to read than that previous version, but it is still not as easy to read as the queries in the **Well Formatted Query** section.

```
SELECT *
FROM (
SELECT DATE_TRUNC('day',occurred_at) AS day,
channel, COUNT(*) as events
FROM web_events
GROUP BY 1,2
ORDER BY 3 DESC) sub;
```

Well Formatted Query

Now for a well formatted example, you can see the table we are pulling from much easier than in the previous queries.

```
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
                channel, COUNT(*) as events
      FROM web_events
      GROUP BY 1,2
      ORDER BY 3 DESC) sub;
```

Additionally, if we have a **GROUP BY**, **ORDER BY**, **WHERE**, **HAVING**, or any other statement following our subquery, we would then indent it at the same level as our outer query.

The query below is similar to the above, but it is applying additional statements to the outer query, so you can see there are **GROUP BY** and **ORDER BY** statements used on the output are not tabbed. The inner query **GROUP BY** and **ORDER BY** statements are indented to match the inner table.

```
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
                channel, COUNT(*) as events
      FROM web_events
      GROUP BY 1,2
      ORDER BY 3 DESC) sub
GROUP BY day, channel, events
ORDER BY 2 DESC;
```

These final two queries are so much easier to read!

Subqueries Part II

In the first subquery you wrote, you created a table that you could then query again in the **FROM** statement. However, if you are only returning a single value, you might use that value in a logical statement like **WHERE**, **HAVING**, or even **SELECT** - the value could be nested within a **CASE** statement.

On the next concept, we will work through this example, and then you will get some practice on answering some questions on your own.

Expert Tip

Note that you should not include an alias when you write a subquery in a conditional statement. This is because the subquery is treated as an individual value (or set of values in the **IN** case) rather than as a table.

Also, notice the query here compared a single value. If we returned an entire column **IN** would need to be used to perform a logical argument. If we are returning an entire table, then we must use an **ALIAS** for the table, and perform additional logic on the entire table.

Queries Needed to Find the Solutions to the Previous Quiz

Here is the necessary quiz to pull the first month/year combo from the orders table.

```
SELECT DATE_TRUNC('month', MIN(occurred_at))
```

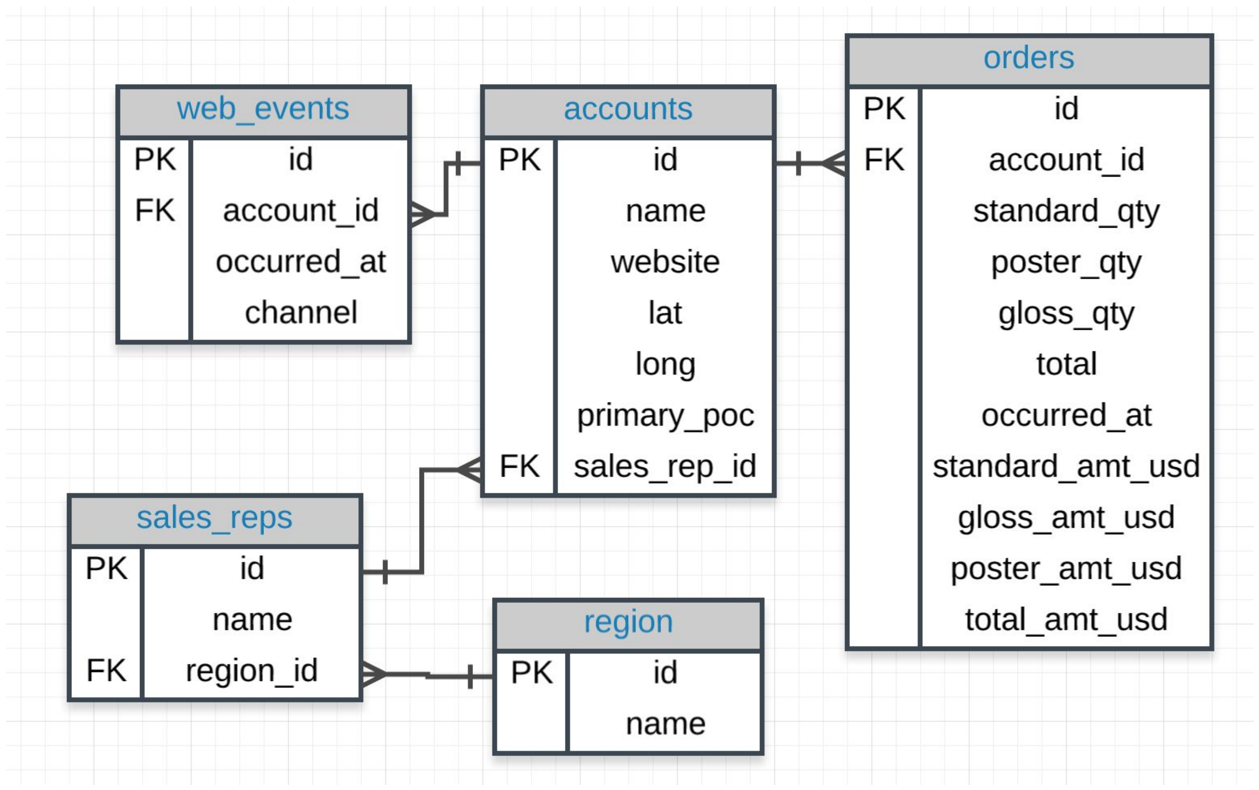
```
FROM orders;
```

1.
2. Then to pull the average for each, we could do this all in one query, but for readability, I provided two queries below to perform each separately.

```
SELECT AVG(standard_qty) avg_std, AVG(gloss_qty) avg_gls, AVG(poster_qty)
avg_pst
FROM orders
WHERE DATE_TRUNC('month', occurred_at) =
     (SELECT DATE_TRUNC('month', MIN(occurred_at)) FROM orders);

SELECT SUM(total_amt_usd)
FROM orders
WHERE DATE_TRUNC('month', occurred_at) =
      (SELECT DATE_TRUNC('month', MIN(occurred_at)) FROM orders);
```

Notice the tables referenced in the video below are those that were shown on the previous page. Sorry they are not shown at the beginning of the video. Skipping to 1:30 will move past the information on the previous page.



Below is a video showcasing the workflow for solving the first problem. Each of the problem solutions is shown in text below as well.

Solution: Subquery Mania

Provide the **name** of the **sales_rep** in each **region** with the largest amount of **total_amt_usd** sales.

First, I wanted to find the **total_amt_usd** totals associated with each **sales rep**, and I also wanted the region in which they were located. The query below provided this information.

```
SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt

FROM sales_reps s
```

```
JOIN accounts a

ON a.sales_rep_id = s.id

JOIN orders o

ON o.account_id = a.id

JOIN region r

ON r.id = s.region_id

GROUP BY 1,2

ORDER BY 3 DESC;
```

Next, I pulled the max for each region, and then we can use this to pull those rows in our final result.

```
SELECT region_name, MAX(total_amt) total_amt

     FROM(SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd)
total_amt

             FROM sales_reps s

             JOIN accounts a

             ON a.sales_rep_id = s.id

             JOIN orders o
```

```
                ON o.account_id = a.id

                JOIN region r

                ON r.id = s.region_id

                GROUP BY 1, 2) t1

     GROUP BY 1;
```

Essentially, this is a **JOIN** of these two tables, where the region and amount match.

```
SELECT t3.rep_name, t3.region_name, t3.total_amt

FROM(SELECT region_name, MAX(total_amt) total_amt

     FROM(SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd)
total_amt

                FROM sales_reps s

                JOIN accounts a

                ON a.sales_rep_id = s.id

                JOIN orders o

                ON o.account_id = a.id

                JOIN region r
                ON r.id = s.region_id

                GROUP BY 1, 2) t1

     GROUP BY 1) t2

JOIN (SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd)
total_amt

     FROM sales_reps s

     JOIN accounts a

     ON a.sales_rep_id = s.id

     JOIN orders o

     ON o.account_id = a.id

     JOIN region r

     ON r.id = s.region_id

     GROUP BY 1,2

     ORDER BY 3 DESC) t3

ON t3.region_name = t2.region_name AND t3.total_amt = t2.total_amt;
```

1.

For the region with the largest sales **total_amt_usd**, how many **total** orders were placed?

The first query I wrote was to pull the **total_amt_usd** for each **region**.

```
SELECT r.name region_name, SUM(o.total_amt_usd) total_amt

FROM sales_reps s

JOIN accounts a

ON a.sales_rep_id = s.id

JOIN orders o

ON o.account_id = a.id

JOIN region r

ON r.id = s.region_id

GROUP BY r.name;
```

Then we just want the region with the max amount from this table. There are two ways I considered getting this amount. One was to pull the max using a subquery. Another way is to order descending and just pull the top value.

```
SELECT MAX(total_amt)

FROM (SELECT r.name region_name, SUM(o.total_amt_usd) total_amt

             FROM sales_reps s
```

```
                JOIN accounts a

                ON a.sales_rep_id = s.id

                JOIN orders o

                ON o.account_id = a.id

                JOIN region r

                ON r.id = s.region_id

                GROUP BY r.name) sub;
```

Finally, we want to pull the total orders for the region with this amount:

```
SELECT r.name, COUNT(o.total) total_orders

FROM sales_reps s

JOIN accounts a

ON a.sales_rep_id = s.id

JOIN orders o

ON o.account_id = a.id

JOIN region r

ON r.id = s.region_id
```

```
GROUP BY r.name

HAVING SUM(o.total_amt_usd) = (

      SELECT MAX(total_amt)

      FROM (SELECT r.name region_name, SUM(o.total_amt_usd) total_amt

              FROM sales_reps s

              JOIN accounts a

              ON a.sales_rep_id = s.id

              JOIN orders o

              ON o.account_id = a.id

              JOIN region r

              ON r.id = s.region_id

              GROUP BY r.name) sub);
```

2. This provides the **Northeast** with **2357** orders.

How many accounts had more **total** purchases than the account **name** which has bought the most **standard_qty** paper throughout their lifetime as a customer?

First, we want to find the account that had the most **standard_qty** paper. The query here pulls that account, as well as the total amount:

```
SELECT a.name account_name, SUM(o.standard_qty) total_std, SUM(o.total) total

FROM accounts a

JOIN orders o

ON o.account_id = a.id

GROUP BY 1

ORDER BY 2 DESC

LIMIT 1;
```

Now, I want to use this to pull all the accounts with more total sales:

```
SELECT a.name

FROM orders o

JOIN accounts a

ON a.id = o.account_id

GROUP BY 1

HAVING SUM(o.total) > (SELECT total
                   FROM (SELECT a.name act_name, SUM(o.standard_qty) tot_std,
SUM(o.total) total

                         FROM accounts a

                         JOIN orders o

                         ON o.account_id = a.id

                         GROUP BY 1

                         ORDER BY 2 DESC

                         LIMIT 1) sub);
```

This is now a list of all the accounts with more total orders. We can get the count with just another simple subquery.

```
SELECT COUNT(*)

FROM (SELECT a.name

       FROM orders o

       JOIN accounts a

       ON a.id = o.account_id

       GROUP BY 1

       HAVING SUM(o.total) > (SELECT total
                   FROM (SELECT a.name act_name, SUM(o.standard_qty) tot_std,
SUM(o.total) total

                         FROM accounts a

                         JOIN orders o

                         ON o.account_id = a.id

                         GROUP BY 1

                         ORDER BY 2 DESC

                         LIMIT 1) inner_tab)

             ) counter_tab;
```

3.

For the customer that spent the most (in total over their lifetime as a customer) **total_amt_usd**, how many **web_events** did they have for each channel?

Here, we first want to pull the customer with the most spent in lifetime value.

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

FROM orders o

JOIN accounts a
```

```
ON a.id = o.account_id

GROUP BY a.id, a.name

ORDER BY 3 DESC

LIMIT 1;
```

Now, we want to look at the number of events on each channel this company had, which we can match with just the **id**.

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

FROM accounts a

JOIN web_events w

ON a.id = w.account_id AND a.id =  (SELECT id

                     FROM (SELECT a.id, a.name, SUM(o.total_amt_usd) tot_spent

                           FROM orders o

                           JOIN accounts a

                           ON a.id = o.account_id

                           GROUP BY a.id, a.name

                           ORDER BY 3 DESC
                           LIMIT 1) inner_table)

GROUP BY 1, 2

ORDER BY 3 DESC;
```

4. I added an **ORDER BY** for no real reason, and the account name to assure I was only pulling from one account.

What is the lifetime average amount spent in terms of **total_amt_usd** for the top 10 total spending **accounts**?

First, we just want to find the top 10 accounts in terms of highest **total_amt_usd**.

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

FROM orders o

JOIN accounts a

ON a.id = o.account_id

GROUP BY a.id, a.name

ORDER BY 3 DESC

LIMIT 10;
```

Now, we just want the average of these 10 amounts.

```
SELECT AVG(tot_spent)

FROM (SELECT a.id, a.name, SUM(o.total_amt_usd) tot_spent

      FROM orders o

      JOIN accounts a

      ON a.id = o.account_id

      GROUP BY a.id, a.name

      ORDER BY 3 DESC

       LIMIT 10) temp;
```

5.

What is the lifetime average amount spent in terms of **total_amt_usd**, including only the companies that spent more per order, on average, than the average of all orders.

First, we want to pull the average of all accounts in terms of **total_amt_usd**:

```
SELECT AVG(o.total_amt_usd) avg_all

FROM orders o
```

Then, we want to only pull the accounts with more than this average amount.

```
SELECT o.account_id, AVG(o.total_amt_usd)

FROM orders o

GROUP BY 1

HAVING AVG(o.total_amt_usd) > (SELECT AVG(o.total_amt_usd) avg_all

                               FROM orders o);
```

Finally, we just want the average of these values.

```
SELECT AVG(avg_amt)

FROM (SELECT o.account_id, AVG(o.total_amt_usd) avg_amt

    FROM orders o

    GROUP BY 1

    HAVING AVG(o.total_amt_usd) > (SELECT AVG(o.total_amt_usd) avg_all

                                   FROM orders o)) temp_table;
```

6.

Wow! That was intense. Nice job if you got these!

The **WITH** statement is often called a **Common Table Expression** or **CTE**. Though these expressions serve the exact same purpose as subqueries, they are more common in practice, as they tend to be cleaner for a future reader to follow the logic.

In the next concept, we will walk through this example a bit more slowly to make sure you have all the similarities between subqueries and these expressions down for you to use in practice! If you are already feeling comfortable skip ahead to practice the quiz section.

On the next page, you will see the details of this query. This might take some back and forth to see how the syntax works.

Your First WITH (CTE)

The same question as you saw in `your first subquery` is provided here along with the solution.

QUESTION: You need to find the average number of events for each channel per day.

SOLUTION:

```
SELECT channel, AVG(events) AS average_events

FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,

             channel, COUNT(*) as events
```

```
      FROM web_events

      GROUP BY 1,2) sub

GROUP BY channel

ORDER BY 2 DESC;
```

Let's try this again using a **WITH** statement.

Notice, you can pull the inner query:

```
SELECT DATE_TRUNC('day',occurred_at) AS day,

       channel, COUNT(*) as events

FROM web_events

GROUP BY 1,2
```

This is the part we put in the **WITH** statement. Notice, we are aliasing the table as `events` below:

```
WITH events AS (
          SELECT DATE_TRUNC('day',occurred_at) AS day,
                         channel, COUNT(*) as events
          FROM web_events
          GROUP BY 1,2)
```

Now, we can use this newly created `events` table as if it is any other table in our database:

```
WITH events AS (
          SELECT DATE_TRUNC('day',occurred_at) AS day,
                         channel, COUNT(*) as events
          FROM web_events
          GROUP BY 1,2)
```

```
SELECT channel, AVG(events) AS average_events

FROM events

GROUP BY channel

ORDER BY 2 DESC;
```

For the above example, we don't need anymore than the one additional table, but imagine we needed to create a second table to pull from. We can create an additional table to pull from in the following way:

```
WITH table1 AS (

          SELECT *

          FROM web_events),



     table2 AS (

          SELECT *

          FROM accounts)
```

```
SELECT *

FROM table1

JOIN table2

ON table1.account_id = table2.id;
```

You can add more and more tables using the **WITH** statement in the same way. The quiz at the bottom will assure you are catching all of the necessary components of these new queries.

Feel free to explore how this works with the environment below.

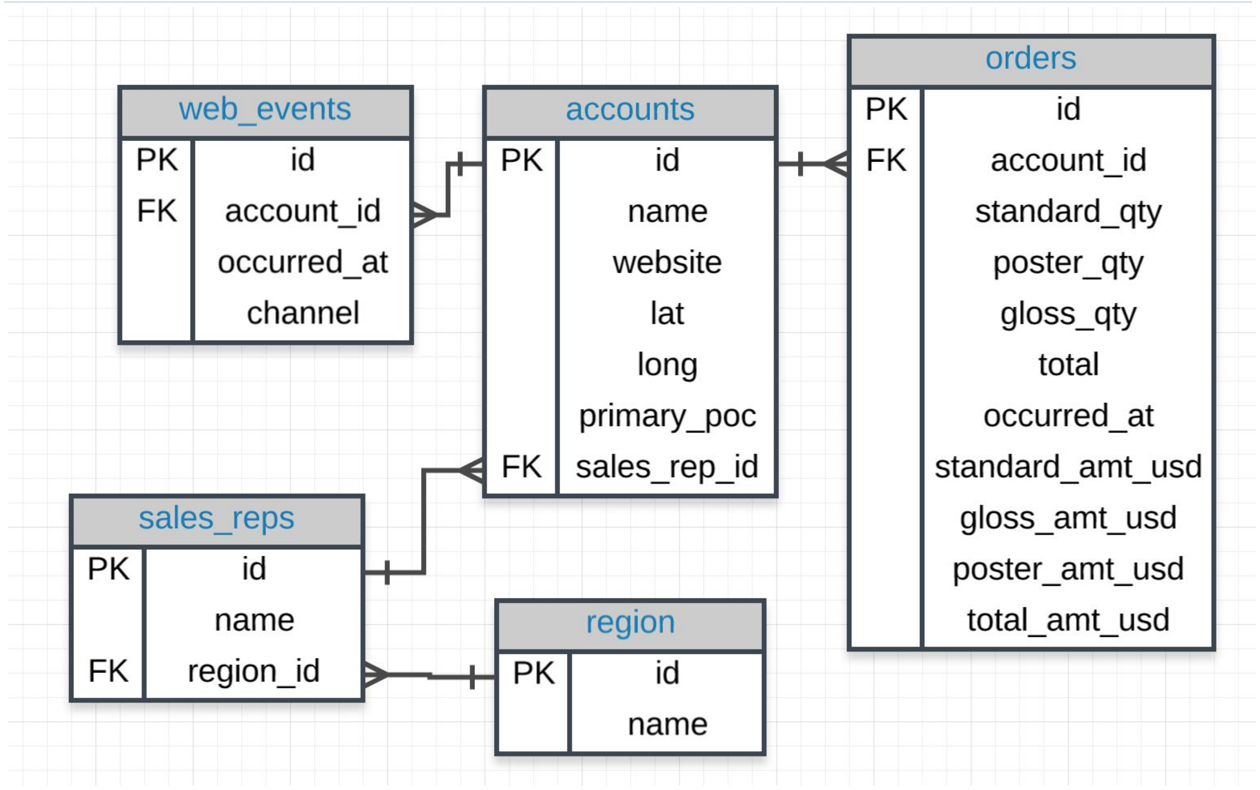
Menu

Expand

Select all of the below that are true regarding **WITH** statements.

- When creating multiple tables using **WITH**, you add a comma after every table leading to your final query.

- When creating multiple tables using **WITH**, you add a comma after every table except the last table leading to your final query.
- The new table name is always aliased using `table_name AS`, which is followed by your query nested between parentheses.
- You begin each new table using a **WITH** statement.



WITH Quizzes

Essentially a **WITH** statement performs the same task as a **Subquery**. Therefore, you can write any of the queries we worked with in the "Subquery Mania" using a **WITH**. That's what you'll do here. Try to perform each of the earlier queries again, but using a **WITH** instead of a subquery.

WITH Solutions

Below, you will see each of the previous solutions restructured using the **WITH** clause.

This is often an easier way to read a query.

Provide the **name** of the **sales_rep** in each **region** with the largest amount of **total_amt_usd** sales.

```
WITH t1 AS (

  SELECT s.name rep_name, r.name region_name, SUM(o.total_amt_usd) total_amt

   FROM sales_reps s

   JOIN accounts a

   ON a.sales_rep_id = s.id

   JOIN orders o

   ON o.account_id = a.id

   JOIN region r

   ON r.id = s.region_id

   GROUP BY 1,2

   ORDER BY 3 DESC),
t2 AS (
    SELECT region_name, MAX(total_amt) total_amt
    FROM t1
    GROUP BY 1)
SELECT t1.rep_name, t1.region_name, t1.total_amt
FROM t1
JOIN t2
ON t1.region_name = t2.region_name AND t1.total_amt = t2.total_amt;
```

1.

For the region with the largest sales **total_amt_usd**, how many **total** orders were placed?

```
WITH t1 AS (
    SELECT r.name region_name, SUM(o.total_amt_usd) total_amt
    FROM sales_reps s
    JOIN accounts a
```

```
   ON a.sales_rep_id = s.id

   JOIN orders o

   ON o.account_id = a.id

   JOIN region r

   ON r.id = s.region_id

   GROUP BY r.name),

t2 AS (

   SELECT MAX(total_amt)

   FROM t1)

SELECT r.name, COUNT(o.total) total_orders

FROM sales_reps s

JOIN accounts a

ON a.sales_rep_id = s.id

JOIN orders o

ON o.account_id = a.id

JOIN region r
```

```
ON r.id = s.region_id

GROUP BY r.name

HAVING SUM(o.total_amt_usd) = (SELECT * FROM t2);
```

2.

For the account that purchased the most (in total over their lifetime as a customer) **standard_qty** paper, **how many accounts** still had more in **total** purchases?

```
WITH t1 AS (

  SELECT a.name account_name, SUM(o.standard_qty) total_std, SUM(o.total) total

  FROM accounts a

  JOIN orders o

  ON o.account_id = a.id

  GROUP BY 1

  ORDER BY 2 DESC

  LIMIT 1),

t2 AS (
  SELECT a.name

  FROM orders o

  JOIN accounts a

  ON a.id = o.account_id

  GROUP BY 1

  HAVING SUM(o.total) > (SELECT total FROM t1))

SELECT COUNT(*)

FROM t2;
```

3.

For the customer that spent the most (in total over their lifetime as a customer) **total_amt_usd**, how many **web_events** did they have for each channel?

```
WITH t1 AS (

   SELECT a.id, a.name, SUM(o.total_amt_usd) tot_spent

   FROM orders o

   JOIN accounts a
   ON a.id = o.account_id

   GROUP BY a.id, a.name

   ORDER BY 3 DESC

   LIMIT 1)

SELECT a.name, w.channel, COUNT(*)

FROM accounts a

JOIN web_events w

ON a.id = w.account_id AND a.id =  (SELECT id FROM t1)

GROUP BY 1, 2

ORDER BY 3 DESC;
```

4.

What is the lifetime average amount spent in terms of **total_amt_usd** for the top 10 total spending **accounts**?

```
WITH t1 AS (

   SELECT a.id, a.name, SUM(o.total_amt_usd) tot_spent
   FROM orders o

   JOIN accounts a

   ON a.id = o.account_id

   GROUP BY a.id, a.name

   ORDER BY 3 DESC

   LIMIT 10)

SELECT AVG(tot_spent)

FROM t1;
```

5.

What is the lifetime average amount spent in terms of **total_amt_usd**, including only the companies that spent more per order, on average, than the average of all orders.

```
WITH t1 AS (

   SELECT AVG(o.total_amt_usd) avg_all

   FROM orders o

   JOIN accounts a
   ON a.id = o.account_id),

t2 AS (

   SELECT o.account_id, AVG(o.total_amt_usd) avg_amt

   FROM orders o

   GROUP BY 1

   HAVING AVG(o.total_amt_usd) > (SELECT * FROM t1))

SELECT AVG(avg_amt)

FROM t2;
```

6.

Wow! That was intense. Nice job if you got these!

Recap

This lesson was the first of the more advanced sequence in writing SQL. Arguably, the advanced features of **Subqueries** and **CTEs** are the most widely used in an analytics role within a company. Being able to break a problem down into the necessary tables and finding a solution using the resulting table is very useful in practice.

If you didn't get the solutions to these queries on the first pass, don't be afraid to come back another time and give them another try. Additionally, you might try coming up with some questions of your own to see if you can find the solution.

The remaining portions of this course may be key to certain analytics roles, but you have now covered all of the main SQL topics you are likely to use on a day to day basis.