**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.

**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.

**Solutions to Your First Subquery**

1. 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.**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,
2. channel, **COUNT**(\*) **as** **events**
3. **FROM** web\_events
4. **GROUP** **BY** 1,2
5. **ORDER** **BY** 3 **DESC**;
6. 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. **SELECT** \*
7. **FROM** (**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,
8. channel, **COUNT**(\*) **as** **events**
9. **FROM** web\_events
10. **GROUP** **BY** 1,2
11. **ORDER** **BY** 3 **DESC**) sub;
12. Finally, here we are able to get a table that shows the average number of events a day for each channel. **SELECT** channel, **AVG**(**events**) **AS** average\_events
13. **FROM** (**SELECT** DATE\_TRUNC('day',occurred\_at) **AS** **day**,
14. channel, **COUNT**(\*) **as** **events**
15. **FROM** web\_events
16. **GROUP** **BY** 1,2) sub
17. **GROUP** **BY** channel

**ORDER** **BY** 2 **DESC**;

**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;

**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** channel

**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.

react-empty: 6485

**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**

1. Here is the necessary quiz to pull the first month/year combo from the orders table. **SELECT** DATE\_TRUNC('month', **MIN**(occurred\_at))
2. **FROM** orders;
3. 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) =

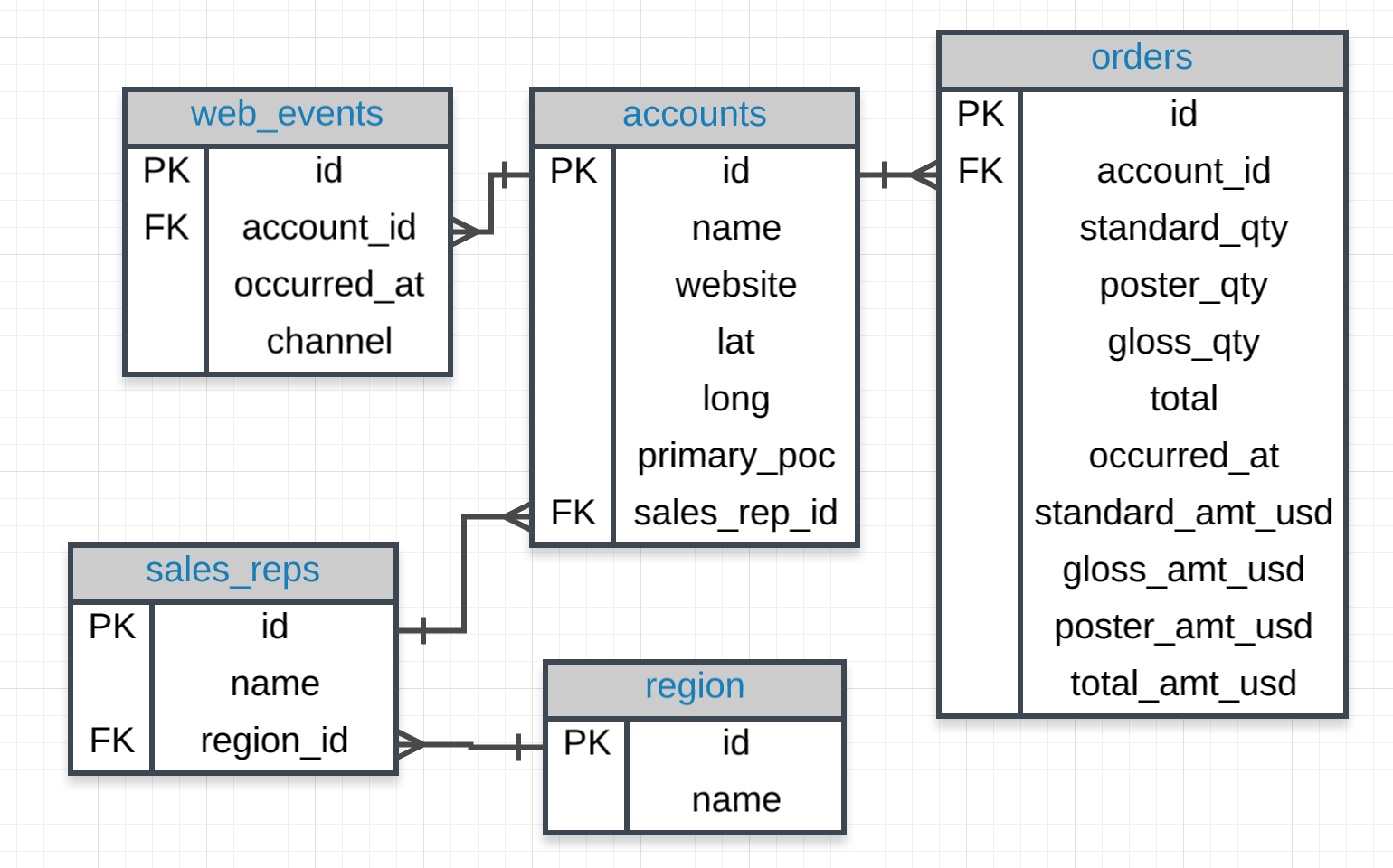
(**SELECT** DATE\_TRUNC('month', **MIN**(occurred\_at)) **FROM** orders);

**SELECT** **SUM**(total\_amt\_usd)

**FROM** orders

**WHERE** DATE\_TRUNC('month', occurred\_at) =

(**SELECT** DATE\_TRUNC('month', **MIN**(occurred\_at)) **FROM** orders);



**More Subqueries Quizzes**

Above is the ERD for the database again - it might come in handy as you tackle the quizzes below. You should write your solution as a subquery, not by finding one solution and copying the output. The importance of this is that it allows your query to be dynamic in answering the question - even if the data changes, you still arrive at the right answer.

1. Provide the **name** of the **sales\_rep** in each **region** with the largest amount of **total\_amt\_usd** sales.
2. For the region with the largest sales **total\_amt\_usd**, how many **total** orders were placed?
3. For the **name** of 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?
4. 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?
5. What is the lifetime average amount spent in terms of **total\_amt\_usd** for the top 10 total spending **accounts**?

What is the lifetime average amount spent in terms of **total\_amt\_usd** for only the companies that spent more than the average of all orders.

**Solution: Subquery Mania**

1. Provide the **name** of the **sales\_rep** in each **region** with the largest amount of **total\_amt\_usd** sales.
2. 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
3. **FROM** sales\_reps s
4. **JOIN** accounts a
5. **ON** a.sales\_rep\_id = s.**id**
6. **JOIN** orders o
7. **ON** o.account\_id = a.**id**
8. **JOIN** region r
9. **ON** r.**id** = s.region\_id
10. **GROUP** **BY** 1,2
11. **ORDER** **BY** 3 **DESC**;
12. 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
13. **FROM**(**SELECT** s.**name** rep\_name, r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
14. **FROM** sales\_reps s
15. **JOIN** accounts a
16. **ON** a.sales\_rep\_id = s.**id**
17. **JOIN** orders o
18. **ON** o.account\_id = a.**id**
19. **JOIN** region r
20. **ON** r.**id** = s.region\_id
21. **GROUP** **BY** 1, 2) t1
22. **GROUP** **BY** 1;
23. 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
24. **FROM**(**SELECT** region\_name, **MAX**(total\_amt) total\_amt
25. **FROM**(**SELECT** s.**name** rep\_name, r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
26. **FROM** sales\_reps s
27. **JOIN** accounts a
28. **ON** a.sales\_rep\_id = s.**id**
29. **JOIN** orders o
30. **ON** o.account\_id = a.**id**
31. **JOIN** region r
32. **ON** r.**id** = s.region\_id
33. **GROUP** **BY** 1, 2) t1
34. **GROUP** **BY** 1) t2
35. **JOIN** (**SELECT** s.**name** rep\_name, r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
36. **FROM** sales\_reps s
37. **JOIN** accounts a
38. **ON** a.sales\_rep\_id = s.**id**
39. **JOIN** orders o
40. **ON** o.account\_id = a.**id**
41. **JOIN** region r
42. **ON** r.**id** = s.region\_id
43. **GROUP** **BY** 1,2
44. **ORDER** **BY** 3 **DESC**) t3
45. **ON** t3.region\_name = t2.region\_name **AND** t3.total\_amt = t2.total\_amt;
46. For the region with the largest sales **total\_amt\_usd**, how many **total** orders were placed?
47. 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
48. **FROM** sales\_reps s
49. **JOIN** accounts a
50. **ON** a.sales\_rep\_id = s.**id**
51. **JOIN** orders o
52. **ON** o.account\_id = a.**id**
53. **JOIN** region r
54. **ON** r.**id** = s.region\_id
55. **GROUP** **BY** r.**name**;
56. 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)
57. **FROM** (**SELECT** r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
58. **FROM** sales\_reps s
59. **JOIN** accounts a
60. **ON** a.sales\_rep\_id = s.**id**
61. **JOIN** orders o
62. **ON** o.account\_id = a.**id**
63. **JOIN** region r
64. **ON** r.**id** = s.region\_id
65. **GROUP** **BY** r.**name**) sub;
66. Finally, we want to pull the total orders for the region with this amount:**SELECT** r.**name**, **SUM**(o.total) total\_orders
67. **FROM** sales\_reps s
68. **JOIN** accounts a
69. **ON** a.sales\_rep\_id = s.**id**
70. **JOIN** orders o
71. **ON** o.account\_id = a.**id**
72. **JOIN** region r
73. **ON** r.**id** = s.region\_id
74. **GROUP** **BY** r.**name**
75. **HAVING** **SUM**(o.total\_amt\_usd) = (
76. **SELECT** **MAX**(total\_amt)
77. **FROM** (**SELECT** r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
78. **FROM** sales\_reps s
79. **JOIN** accounts a
80. **ON** a.sales\_rep\_id = s.**id**
81. **JOIN** orders o
82. **ON** o.account\_id = a.**id**
83. **JOIN** region r
84. **ON** r.**id** = s.region\_id
85. **GROUP** **BY** r.**name**) sub);
86. This provides the **Northeast** with **1230378** orders.
87. 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?   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
88. **FROM** accounts a
89. **JOIN** orders o
90. **ON** o.account\_id = a.**id**
91. **GROUP** **BY** 1
92. **ORDER** **BY** 2 **DESC**
93. **LIMIT** 1;
94. Now, I want to use this to pull all the accounts with more total sales: **SELECT** a.**name**
95. **FROM** orders o
96. **JOIN** accounts a
97. **ON** a.**id** = o.account\_id
98. **GROUP** **BY** 1
99. **HAVING** **SUM**(o.total) > (**SELECT** total
100. **FROM** (**SELECT** a.**name** act\_name, **SUM**(o.standard\_qty) tot\_std, **SUM**(o.total) total
101. **FROM** accounts a
102. **JOIN** orders o
103. **ON** o.account\_id = a.**id**
104. **GROUP** **BY** 1
105. **ORDER** **BY** 2 **DESC**
106. **LIMIT** 1) sub);
107. 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**(\*)
108. **FROM** (**SELECT** a.**name**
109. **FROM** orders o
110. **JOIN** accounts a
111. **ON** a.**id** = o.account\_id
112. **GROUP** **BY** 1
113. **HAVING** **SUM**(o.total) > (**SELECT** total
114. **FROM** (**SELECT** a.**name** act\_name, **SUM**(o.standard\_qty) tot\_std, **SUM**(o.total) total
115. **FROM** accounts a
116. **JOIN** orders o
117. **ON** o.account\_id = a.**id**
118. **GROUP** **BY** 1
119. **ORDER** **BY** 2 **DESC**
120. **LIMIT** 1) inner\_tab)
121. ) counter\_tab;
122. 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
123. **FROM** orders o
124. **JOIN** accounts a
125. **ON** a.**id** = o.account\_id
126. **GROUP** **BY** a.**id**, a.**name**
127. **ORDER** **BY** 3 **DESC**
128. **LIMIT** 1;
129. 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**(\*)
130. **FROM** accounts a
131. **JOIN** web\_events w
132. **ON** a.**id** = w.account\_id **AND** a.**id** = (**SELECT** **id**
133. **FROM** (**SELECT** a.**id**, a.**name**, **SUM**(o.total\_amt\_usd) tot\_spent
134. **FROM** orders o
135. **JOIN** accounts a
136. **ON** a.**id** = o.account\_id
137. **GROUP** **BY** a.**id**, a.**name**
138. **ORDER** **BY** 3 **DESC**
139. **LIMIT** 1) inner\_table)
140. **GROUP** **BY** 1, 2
141. **ORDER** **BY** 3 **DESC**;
142. I added an **ORDER BY** for no real reason, and the account name to assure I was only pulling from one account.
143. What is the lifetime average amount spent in terms of **total\_amt\_usd** for the top 10 total spending **accounts**?
144. 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
145. **FROM** orders o
146. **JOIN** accounts a
147. **ON** a.**id** = o.account\_id
148. **GROUP** **BY** a.**id**, a.**name**
149. **ORDER** **BY** 3 **DESC**
150. **LIMIT** 10;
151. Now, we just want the average of these 10 amounts.**SELECT** **AVG**(tot\_spent)
152. **FROM** (**SELECT** a.**id**, a.**name**, **SUM**(o.total\_amt\_usd) tot\_spent
153. **FROM** orders o
154. **JOIN** accounts a
155. **ON** a.**id** = o.account\_id
156. **GROUP** **BY** a.**id**, a.**name**
157. **ORDER** **BY** 3 **DESC**
158. **LIMIT** 10) temp;
160. What is the lifetime average amount spent in terms of **total\_amt\_usd** for only the companies that spent more than the average of all orders.
161. First, we want to pull the average of all accounts in terms of **total\_amt\_usd**:**SELECT** **AVG**(o.total\_amt\_usd) avg\_all
162. **FROM** orders o
163. **JOIN** accounts a
164. **ON** a.**id** = o.account\_id;
165. Then, we want to only pull the accounts with more than this average amount. **SELECT** o.account\_id, **AVG**(o.total\_amt\_usd)
166. **FROM** orders o
167. **GROUP** **BY** 1
168. **HAVING** **AVG**(o.total\_amt\_usd) > (**SELECT** **AVG**(o.total\_amt\_usd) avg\_all
169. **FROM** orders o
170. **JOIN** accounts a
171. **ON** a.**id** = o.account\_id);
172. Finally, we just want the average of these values. **SELECT** **AVG**(avg\_amt)
173. **FROM** (**SELECT** o.account\_id, **AVG**(o.total\_amt\_usd) avg\_amt
174. **FROM** orders o
175. **GROUP** **BY** 1
176. **HAVING** **AVG**(o.total\_amt\_usd) > (**SELECT** **AVG**(o.total\_amt\_usd) avg\_all
177. **FROM** orders o
178. **JOIN** accounts a
179. **ON** a.**id** = o.account\_id)) temp\_table;

**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.

react-empty: 6895

**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** \*

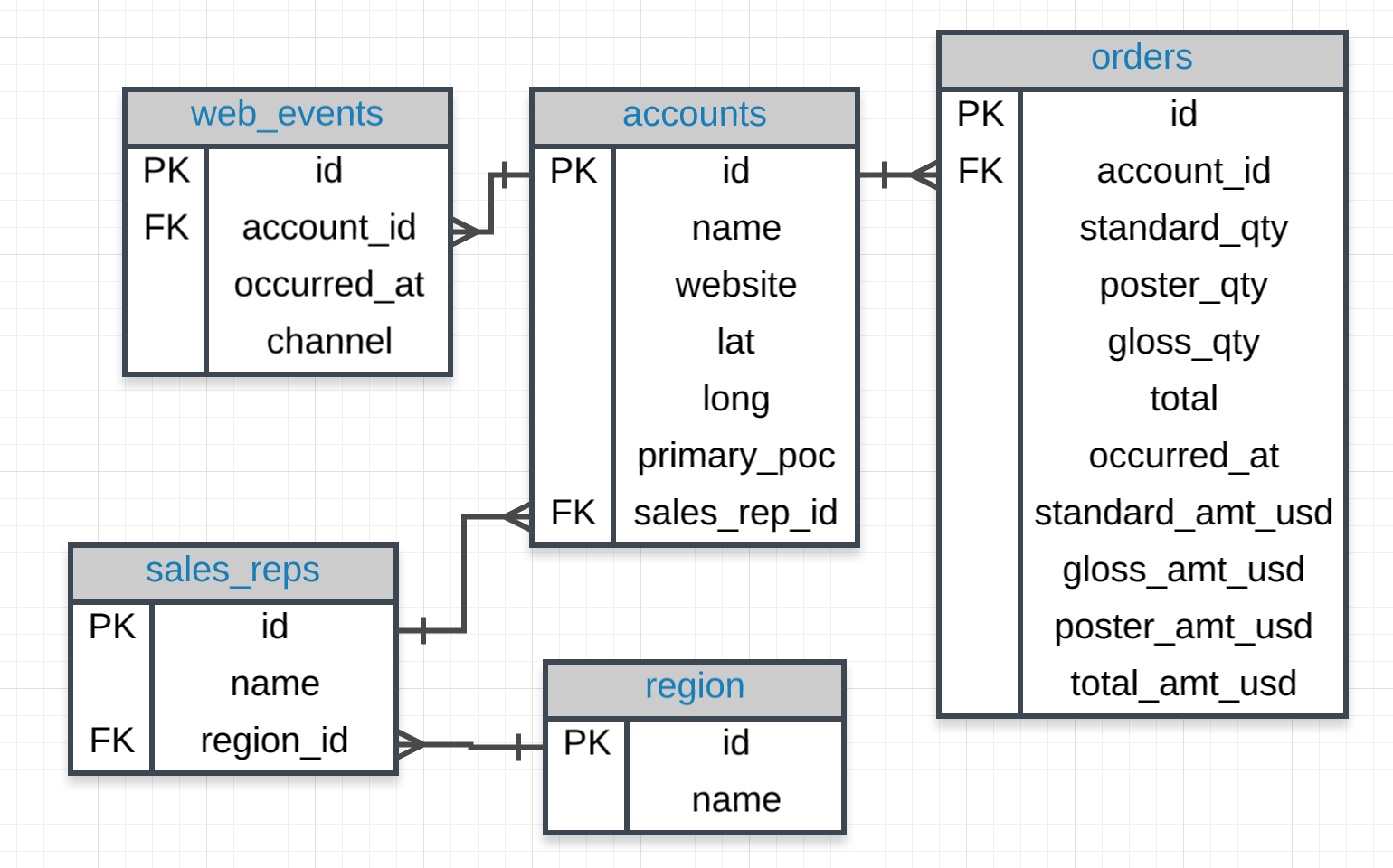
**FROM** table1

**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.



**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.

react-empty: 7040

Above is the ERD for the database again - it might come in handy as you tackle the quizzes below. You should write your solution as using a **WITH** statement, not by finding one solution and copying the output. The importance of this is that it allows your query to be dynamic in answering the question - even if the data changes, you still arrive at the right answer.

1. Provide the **name** of the **sales\_rep** in each **region** with the largest amount of **total\_amt\_usd** sales.
2. For the region with the largest sales **total\_amt\_usd**, how many **total** orders were placed?
3. For the **name** of 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?
4. 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?
5. What is the lifetime average amount spent in terms of **total\_amt\_usd** for the top 10 total spending **accounts**?

What is the lifetime average amount spent in terms of **total\_amt\_usd** for only the companies that spent more than the average of all **accounts**.

**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.

1. Provide the **name** of the **sales\_rep** in each **region** with the largest amount of **total\_amt\_usd** sales.
2. WITH t1 AS (
3. **SELECT** s.**name** rep\_name, r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
4. **FROM** sales\_reps s
5. **JOIN** accounts a
6. **ON** a.sales\_rep\_id = s.**id**
7. **JOIN** orders o
8. **ON** o.account\_id = a.**id**
9. **JOIN** region r
10. **ON** r.**id** = s.region\_id
11. **GROUP** **BY** 1,2
12. **ORDER** **BY** 3 **DESC**),
13. t2 **AS** (
14. **SELECT** region\_name, **MAX**(total\_amt) total\_amt
15. **FROM** t1
16. **GROUP** **BY** 1)
17. **SELECT** t1.rep\_name, t1.region\_name, t1.total\_amt
18. **FROM** t1
19. **JOIN** t2
20. **ON** t1.region\_name = t2.region\_name **AND** t1.total\_amt = t2.total\_amt;
21. For the region with the largest sales **total\_amt\_usd**, how many **total** orders were placed?
22. WITH t1 AS (
23. **SELECT** r.**name** region\_name, **SUM**(o.total\_amt\_usd) total\_amt
24. **FROM** sales\_reps s
25. **JOIN** accounts a
26. **ON** a.sales\_rep\_id = s.**id**
27. **JOIN** orders o
28. **ON** o.account\_id = a.**id**
29. **JOIN** region r
30. **ON** r.**id** = s.region\_id
31. **GROUP** **BY** r.**name**),
32. t2 **AS** (
33. **SELECT** **MAX**(total\_amt)
34. **FROM** t1)
35. **SELECT** r.**name**, **SUM**(o.total) total\_orders
36. **FROM** sales\_reps s
37. **JOIN** accounts a
38. **ON** a.sales\_rep\_id = s.**id**
39. **JOIN** orders o
40. **ON** o.account\_id = a.**id**
41. **JOIN** region r
42. **ON** r.**id** = s.region\_id
43. **GROUP** **BY** r.**name**
44. **HAVING** **SUM**(o.total\_amt\_usd) = (**SELECT** \* **FROM** t2);
45. 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?
46. WITH t1 AS (
47. **SELECT** a.**name** account\_name, **SUM**(o.standard\_qty) total\_std, **SUM**(o.total) total
48. **FROM** accounts a
49. **JOIN** orders o
50. **ON** o.account\_id = a.**id**
51. **GROUP** **BY** 1
52. **ORDER** **BY** 2 **DESC**
53. **LIMIT** 1),
54. t2 **AS** (
55. **SELECT** a.**name**
56. **FROM** orders o
57. **JOIN** accounts a
58. **ON** a.**id** = o.account\_id
59. **GROUP** **BY** 1
60. **HAVING** **SUM**(o.total) > (**SELECT** total **FROM** t1))
61. **SELECT** **COUNT**(\*)
62. **FROM** t2;
63. 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?
64. WITH t1 AS (
65. **SELECT** a.**id**, a.**name**, **SUM**(o.total\_amt\_usd) tot\_spent
66. **FROM** orders o
67. **JOIN** accounts a
68. **ON** a.**id** = o.account\_id
69. **GROUP** **BY** a.**id**, a.**name**
70. **ORDER** **BY** 3 **DESC**
71. **LIMIT** 1)
72. **SELECT** a.**name**, w.channel, **COUNT**(\*)
73. **FROM** accounts a
74. **JOIN** web\_events w
75. **ON** a.**id** = w.account\_id **AND** a.**id** = (**SELECT** **id** **FROM** t1)
76. **GROUP** **BY** 1, 2
77. **ORDER** **BY** 3 **DESC**;
78. What is the lifetime average amount spent in terms of **total\_amt\_usd** for the top 10 total spending **accounts**?
79. WITH t1 AS (
80. **SELECT** a.**id**, a.**name**, **SUM**(o.total\_amt\_usd) tot\_spent
81. **FROM** orders o
82. **JOIN** accounts a
83. **ON** a.**id** = o.account\_id
84. **GROUP** **BY** a.**id**, a.**name**
85. **ORDER** **BY** 3 **DESC**
86. **LIMIT** 10)
87. **SELECT** **AVG**(tot\_spent)
88. **FROM** t1;
90. What is the lifetime average amount spent in terms of **total\_amt\_usd** for only the companies that spent more than the average of all **accounts**.
91. WITH t1 AS (
92. **SELECT** **AVG**(o.total\_amt\_usd) avg\_all
93. **FROM** orders o
94. **JOIN** accounts a
95. **ON** a.**id** = o.account\_id),
96. t2 **AS** (
97. **SELECT** o.account\_id, **AVG**(o.total\_amt\_usd) avg\_amt
98. **FROM** orders o
99. **GROUP** **BY** 1
100. **HAVING** **AVG**(o.total\_amt\_usd) > (**SELECT** \* **FROM** t1))
101. **SELECT** **AVG**(avg\_amt)
102. **FROM** t2;
103. **Wow! That was intense. Nice job if you got these!**

react-empty: 7101

**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.