@Auther: Gabir N. Youssef
@Date: 6 September 2019
@Assignments and Solutions: http://bit.ly/2k5b570

SQL Subqueries & Temporary Tables

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 the 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**.

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 in multiple different ways are 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.

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.

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 any more 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.

Related to WITH statement

- 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.