MySQL Exercise 9: Subqueries and Derived Tables

Now that you understand how joins work, in this lesson we are going to learn how to incorporate subqueries and derived tables into our queries.

Subqueries, which are also sometimes called inner queries or nested queries, are queries that are embedded within the context of another query. The output of a subquery is incorporated into the queries that surround it. Subqueries can be used in SELECT, WHERE, and FROM clauses. When they are used in FROM clauses they create what are called derived tables.

The main reasons to use subqueries are:

- Sometimes they are the most logical way to retrieve the information you want
- They can be used to isolate each logical part of a statement, which can be helpful for troubleshooting long and complicated queries
- Sometimes they run faster than joins

Some people find subqueries easier to read than joins. However, that is often a result of not feeling comfortable with the concepts behind joins in the first place (I prefer join syntax, so admittedly, that is my preference).

Subqueries must be enclosed in parentheses. Subqueries have a couple of rules that joins don't:

- ORDER BY phrases cannot be used in subqueries (although ORDER BY phrases can still be used in outer queries that contain subqueries).
- Subqueries in SELECT or WHERE clauses that return more than one row must be used in combination with operators that are explicitly designed to handle multiple values, such as the IN operator. Otherwise, subqueries in SELECT or WHERE statements can output no more than 1 row.

So why would you use subqueries?

Let's look at some examples.

Start by loading the sql library and database, and making the Dognition database your default database :

In [1]:

```
%load_ext sql
%sql mysql://studentuser:studentpw@localhost/dognitiondb
%sql USE dognitiondb
```

```
 * mysql://studentuser:***@localhost/dognitiondb
0 rows affected.
```

Out[1]:

```
[]
```

1) "On the fly calculations" (or, doing calculations as you need them)

One of the main uses of subqueries is to calculate values as you need them. This allows you to use a summary calculation in your query without having to enter the value outputted by the calculation explicitly. A situation when this capability would be useful is if you wanted to see all the records that were greater than the average value of a subset of your data.

Recall one of the queries we wrote in "MySQL Exercise 4: Summarizing your Data" to calculate the average amount of time it took customers to complete all of the tests in the exam_answers table (we had to exclude negative durations from the calculation due to some abnormalities in the data):

```
SELECT AVG(TIMESTAMPDIFF(minute,start_time,end_time)) AS AvgDuration
FROM exam_answers
WHERE TIMESTAMPDIFF(minute,start_time,end_time)>0;
```

What if we wanted to look at just the data from rows whose durations were greater than the average, so that we could determine whether there are any features that seem to correlate with dogs taking a longer time to finish their tests? We could use a subquery to calculate the average duration, and then indicate in our SELECT and WHERE clauses that we only wanted to retrieve the rows whose durations were greater than the average. Here's what the query would look like:

```
SELECT *
FROM exam_answers
WHERE TIMESTAMPDIFF(minute,start_time,end_time) >
    (SELECT AVG(TIMESTAMPDIFF(minute,start_time,end_time)) AS AvgDuration
     FROM exam_answers
     WHERE TIMESTAMPDIFF(minute,start_time,end_time)>0);
```

You can see that TIMESTAMPDIFF gets compared to the singular average value outputted by the subquery surrounded by parentheses. You can also see that it's easier to read the query as a whole if you indent and align all the clauses associated with the subquery, relative to the main query.

Question 1: How could you use a subquery to extract all the data from exam_answers that had test durations that were greater than the average duration for the "Yawn Warm-Up" game? Start by writing the query that gives you the average duration for the "Yawn Warm-Up" game by itself (and don't forget to exclude negative values; your average duration should be about 9934):

In [22]:

```
%%sql
SELECT AVG(TIMESTAMPDIFF(MINUTE,start_time,end_time)) FROM exam_answers
WHERE test_name = "Yawn Warm-Up"
AND TIMESTAMPDIFF(MINUTE,start_time,end_time) > 0;
```

```
 * mysql://studentuser:***@localhost/dognitiondb
1 rows affected.
```

Out[22]:

AVG(TIMESTAMPDIFF(MINUTE,start_time,end_time))
9933.5197

Question 2: Once you've verified that your subquery is written correctly on its own, incorporate it into a main query to extract all the data from exam_answers that had test durations that were greater than the average duration for the "Yawn Warm-Up" game (you will get 11059 rows):

In [ ]:

```
%%sql
SELECT subcategory_name, test_name, start_time, end_time, dog_guid,
TIMESTAMPDIFF(MINUTE,start_time,end_time) as Time_diff FROM exam_answers
WHERE TIMESTAMPDIFF(MINUTE,start_time,end_time) >
    (SELECT AVG(TIMESTAMPDIFF(MINUTE,start_time,end_time)) FROM exam_answers
    WHERE test_name = "Yawn Warm-Up"
    AND TIMESTAMPDIFF(MINUTE,start_time,end_time) > 0)
    ORDER BY Time_diff;
```

Now double check the results you just retrieved by replacing the subquery with "9934"; you should get the same results. It is helpful to get into the habit of including these kinds of quality checks into your query-writing process.

This example shows you how subqueries allow you retrieve information dynamically, rather than having to hard code in specific numbers or names. This capability is particularly useful when you need to build the output of your queries into reports or dashboards that are supposed to display real-time information.

2) Testing membership

Subqueries can also be useful for assessing whether groups of rows are members of other groups of rows. To use them in this capacity, we need to know about and practice the IN, NOT IN, EXISTS, and NOT EXISTS operators.

Recall from MySQL Exercise 2: Selecting Data Subsets Using WHERE that the IN operator allows you to use a WHERE clause to say how you want your results to relate to a list of multiple values. It's basically a condensed way of writing a sequence of OR statements. The following query would select all the users who live in the state of North Carolina (abbreviated "NC") or New York (abbreviated "NY"):

```
SELECT *
FROM users
WHERE state IN ('NC','NY');
```

Notice the quotation marks around the members of the list referred to by the IN statement. These quotation marks are required since the state names are strings of text.

A query that would give an equivalent result would be:

```
SELECT *
FROM users
WHERE state ='NC' OR state ='NY';
```

A query that would select all the users who do NOT live in the state of North Carolina or New York would be:

```
SELECT *
FROM users
WHERE state NOT IN ('NC','NY');
```

Question 3: Use an IN operator to determine how many entries in the exam_answers tables are from the "Puzzles", "Numerosity", or "Bark Game" tests. You should get a count of 163022.

In [3]:

```
%%sql
SELECT COUNT(*)
FROM exam_answers
WHERE subcategory_name IN ('Puzzles','Numerosity','Bark Game');
```

```
 * mysql://studentuser:***@localhost/dognitiondb
1 rows affected.
```

Out[3]:

COUNT(*)
163022

Question 4: Use a NOT IN operator to determine how many unique dogs in the dog table are NOT in the "Working", "Sporting", or "Herding" breeding groups. You should get an answer of 7961.

In [25]:

```
%%sql
SELECT COUNT(*)
FROM dogs
WHERE breed_group NOT IN ('Working', 'Sporting', 'Herding');
```

```
 * mysql://studentuser:***@localhost/dognitiondb
1 rows affected.
```

Out[25]:

COUNT(*)
7961

EXISTS and NOT EXISTS perform similar functions to IN and NOT IN, but EXISTS and NOT EXISTS can only be used in subqueries. The syntax for EXISTS and NOT EXISTS statements is a little different than that of IN statements because EXISTS is not preceded by a column name or any other expression. The most important difference between EXISTS/NOT EXISTS and IN/NOT IN statements, though, is that unlike IN/NOT IN statements, EXISTS/NOT EXISTS are logical statements. Rather than returning raw data, per se, EXISTS/NOT EXISTS statements return a value of TRUE or FALSE. As a practical consequence, EXISTS statements are often written using an asterisk after the SELECT clause rather than explicit column names. The asterisk is faster to write, and since the output is just going to be a logical true/false either way, it does not matter whether you use an asterisk or explicit column names.

We can use EXISTS and a subquery to compare the users who are in the users table and dogs table, similar to what we practiced previously using joins. If we wanted to retrieve a list of all the users in the users table who were also in the dogs table, we could write:

```
SELECT DISTINCT u.user_guid AS uUserID
FROM users u
WHERE EXISTS (SELECT d.user_guid
              FROM dogs d
              WHERE u.user_guid =d.user_guid);
```

You would get the same result if you wrote:

```
SELECT DISTINCT u.user_guid AS uUserID
FROM users u
WHERE EXISTS (SELECT *
              FROM dogs d
              WHERE u.user_guid =d.user_guid);
```

Essentially, both of these queries say give me all the distinct user_guids from the users table that have a value of "TRUE" in my EXISTS clause. The results would be equivalent to an inner join with GROUP BY query. Now...

Question 5: How could you determine the number of unique users in the users table who were NOT in the dogs table using a NOT EXISTS clause? You should get the 2226, the same result as you got in Question 10 of MySQL Exercise 8: Joining Tables with Outer Joins.

In [36]:

```
%%sql
SELECT COUNT(DISTINCT user_guid)
FROM users u
WHERE NOT EXISTS (SELECT *
                  FROM dogs d
                  WHERE d.user_guid = u.user_guid);
```

```
 * mysql://studentuser:***@localhost/dognitiondb
1 rows affected.
```

Out[36]:

COUNT(DISTINCT user_guid)
2226

3) Accurate logical representations of desired output and Derived Tables

A third situation in which subqueries can be useful is when they simply represent the logic of what you want better than joins.

We saw an example of this in our last MySQL Exercise. We wanted a list of each dog a user in the users table owns, with its accompanying breed information whenever possible. To achieve this, we wrote this query in Question 6:

```
SELECT u.user_guid AS uUserID, d.user_guid AS dUserID, d.dog_guid AS dDogID, d.breed
FROM users u LEFT JOIN dogs d
   ON u.user_guid=d.user_guid
```

Once we saw the "exploding rows" phenomenon due to duplicate rows, we wrote a follow-up query in Question 7 to assess how many rows would be outputted per user_id when we left joined the users table on the dogs table:

```
SELECT u.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM users u LEFT JOIN dogs d
   ON u.user_guid=d.user_guid
GROUP BY u.user_guid
ORDER BY numrows DESC
```

This same general query without the COUNT function could have been used to output a complete list of all the distinct users in the users table, their dogs, and their dogs' breed information. However, the method we used to arrive at this was not very pretty or logically satisfying. Rather than joining many duplicated rows and fixing the results later with the GROUP BY clause, it would be much more elegant if we could simply join the distinct UserIDs in the first place. There is no way to do that with join syntax, on its own. However, you can use subqueries in combination with joins to achieve this goal.

To complete the join on ONLY distinct UserIDs from the users table, we could write:

```
SELECT DistinctUUsersID.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
   ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC
```

Try it yourself:

In [ ]:

```
%%sql
SELECT DistinctUUsersID.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
   ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC;
```

Queries that include subqueries always run the innermost subquery first, and then run subsequent queries sequentially in order from the innermost query to the outermost query.

Therefore, the query we just wrote extracts the distinct user_guids from the users table *first*, and then left joins that reduced subset of user_guids on the dogs table. As mentioned at the beginning of the lesson, since the subquery is in the FROM statement, it actually creates a temporary table, called a derived table, that is then incorporated into the rest of the query.

There are several important points to notice about the syntax of this subquery. First, an alias of "DistinctUUsersID" is used to name the results of the subquery. *We are required to give an alias to any derived table we create in subqueries within FROM statements.* Otherwise there would be no way for the database to refer to the multiple columns within the temporary results we create.

Second, *we need to use this alias every time we want to execute a function that uses the derived table.* Remember that the results in which we are interested require a join between the dogs table and the temporary table, not the dogs table and the original users table with duplicates. That means we need to make sure we reference the temporary table alias in the ON, GROUP BY, and SELECT clauses.

Third, relatedly, aliases used within subqueries can refer to tables outside of the subqueries. However, *outer queries cannot refer to aliases created within subqueries unless those aliases are explicitly part of the subquery output*. In other words, if you wrote the first line of the query above as:

```
SELECT u.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
...
```

the query would not execute because the alias "u" is contained inside the subquery, but is not included in the output. **Go ahead and try it to see what the error message looks like:**

In [39]:

```
%%sql
SELECT u.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
   ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC;
```

```
 * mysql://studentuser:***@localhost/dognitiondb
(MySQLdb._exceptions.OperationalError) (1054, "Unknown column 'u.user_guid' in 'field list'")
[SQL: SELECT u.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
   ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC;]
(Background on this error at: http://sqlalche.me/e/e3q8)
```

```
(Background on this error at: http://sqlalche.me/e/e3q8)
```

A similar thing would happen if you tried to use the alias u in the GROUP BY statement.

Another thing to take note of is that when you use subqueries in FROM statements, the temporary table you create can have multiple columns in the output (unlike when you use subqueries in outside SELECT statements). But for that same reason, subqueries in FROM statements can be very computationally intensive. Therefore, it's a good idea to use them sparingly, especially when you have very large data sets.

Overall, subqueries and joins can often be used interchangeably. Some people strongly prefer one approach over another, but there is no consensus about which approach is best. When you are analyzing very large datasets, it's a good idea to test which approach will likely be faster or easier to troubleshoot for your particular application.

Let's practice some more subqueries!

Question 6: Write a query using an IN clause and equijoin syntax that outputs the dog_guid, breed group, state of the owner, and zip or the owner for each distinct dog in the Working, Sporting, and Herding breed groups. (You should get 10,254 rows; the query will be a little slower than some of the others we have practiced)

In [ ]:

```
%%sql
SELECT DISTINCT d.dog_guid, d.breed_group, u.state, u.zip
FROM dogs d, users u
WHERE d.breed_group IN ('Working', 'Sporting', 'Herding');
```

```
 * mysql://studentuser:***@localhost/dognitiondb
```

Question 7: Write the same query as in Question 6 using traditional join syntax.

In [ ]:

```
%%sql
SELECT DISTINCT d.dog_guid, d.breed_group, u.state, u.zip
FROM dogs d
JOIN users u
ON d.user_guid = u.user_guid
WHERE d.breed_group IN ('Working', 'Sporting', 'Herding');
```

Question 8: Earlier we examined unique users in the users table who were NOT in the dogs table. Use a NOT EXISTS clause to examine all the users in the dogs table that are not in the users table (you should get 2 rows in your output).

In [12]:

```
%%sql
SELECT d.user_guid as UserGuid, d.dog_guid as DogGuid
FROM dogs d
WHERE NOT EXISTS(SELECT DISTINCT u.user_guid
                 FROM users u
                 WHERE u.user_guid = d.user_guid);
```

```
 * mysql://studentuser:***@localhost/dognitiondb
2 rows affected.
```

Out[12]:

UserGuid	DogGuid
None	fd7c0a66-7144-11e5-ba71-058fbc01cf0b
None	fdbb6b7a-7144-11e5-ba71-058fbc01cf0b

Question 9: We saw earlier that user_guid 'ce7b75bc-7144-11e5-ba71-058fbc01cf0b' still ends up with 1819 rows of output after a left outer join with the dogs table. If you investigate why, you'll find out that's because there are duplicate user_guids in the dogs table as well. How would you adapt the query we wrote earlier (copied below) to only join unique UserIDs from the users table with unique UserIDs from the dog table?

Join we wrote earlier:

```
SELECT DistinctUUsersID.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
  ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC;
```

Let's build our way up to the correct query. To troubleshoot, let's only examine the rows related to user_guid 'ce7b75bc-7144-11e5-ba71-058fbc01cf0b', since that's the userID that is causing most of the trouble. Rewrite the query above to only LEFT JOIN *distinct* user(s) from the user table whose user_guid='ce7b75bc-7144-11e5-ba71-058fbc01cf0b'. The first two output columns should have matching user_guids, and the numrows column should have one row with a value of 1819:

In [13]:

```
%%sql
SELECT DistinctUUsersID.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u
      WHERE u.user_guid = "ce7b75bc-7144-11e5-ba71-058fbc01cf0b") AS DistinctUUsersID
LEFT JOIN dogs d
  ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC;
```

```
 * mysql://studentuser:***@localhost/dognitiondb
1 rows affected.
```

Out[13]:

uUserID	dUserID	numrows
ce7b75bc-7144-11e5-ba71-058fbc01cf0b	ce7b75bc-7144-11e5-ba71-058fbc01cf0b	1819

Question 10: Now let's prepare and test the inner query for the right half of the join. Give the dogs table an alias, and write a query that would select the distinct user_guids from the dogs table (we will use this query as a inner subquery in subsequent questions, so you will need an alias to differentiate the user_guid column of the dogs table from the user_guid column of the users table).

In [ ]:

```
%%sql
SELECT DISTINCT d.user_guid AS dUserGuid
FROM dogs d;
```

Question 11: Now insert the query you wrote in Question 9 as a subquery on the right part of the join you wrote in question 8. The output should return columns that should have matching user_guids, and 1 row in the numrows column with a value of 1. If you are getting errors, make sure you have given an alias to the derived table you made to extract the distinct user_guids from the dogs table, and double-check that your aliases are referenced correctly in the SELECT and ON statements.

In [19]:

```
%%sql
SELECT DistinctUUsersID.user_guid AS uUserID, dogsGuids.dUserGuid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid
      FROM users u
      WHERE u.user_guid = "ce7b75bc-7144-11e5-ba71-058fbc01cf0b") AS DistinctUUsersID
LEFT JOIN (SELECT DISTINCT d.user_guid AS dUserGuid
            FROM dogs d) as dogsGuids
  ON DistinctUUsersID.user_guid=dogsGuids.dUserGuid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC;
```

```
 * mysql://studentuser:***@localhost/dognitiondb
1 rows affected.
```

Out[19]:

uUserID	dUserID	numrows
ce7b75bc-7144-11e5-ba71-058fbc01cf0b	ce7b75bc-7144-11e5-ba71-058fbc01cf0b	1

Question 12: Adapt the query from Question 10 so that, in theory, you would retrieve a full list of all the DogIDs a user in the users table owns, with its accompagnying breed information whenever possible. HOWEVER, BEFORE YOU RUN THE QUERY MAKE SURE TO LIMIT YOUR OUTPUT TO 100 ROWS *WITHIN* THE SUBQUERY TO THE LEFT OF YOUR JOIN. If you run the query without imposing limits it will take a *very* long time. If you try to limit the output by just putting a limit clause at the end of the outermost query, the database will still have to hold the entire derived tables in memory and join each row of the derived tables before limiting the output. If you put the limit clause in the subquery to the left of the join, the database will only have to join 100 rows of data.

In [ ]:

```
%%sql
SELECT DistinctUUsersID.user_guid AS uUserID, DistictDUsersID.user_guid AS dUserID,
       DistictDUsersID.dog_guid AS DogID, DistictDUsersID.breed AS breed
FROM (SELECT DISTINCT u.user_guid
      FROM users u
      LIMIT 100) AS DistinctUUsersID
LEFT JOIN (SELECT DISTINCT d.user_guid, d.dog_guid, d.breed
           FROM dogs d) AS DistictDUsersID
ON DistinctUUsersID.user_guid=DistictDUsersID.user_guid
ORDER BY DistinctUUsersID.user_guid;
```

Question 13: You might have a good guess by now about why there are duplicate rows in the dogs table and users table, even though most corporate databases are configured to prevent duplicate rows from ever being accepted. To be sure, though, let's adapt this query we wrote above:

```
SELECT DistinctUUsersID.user_guid AS uUserID, d.user_guid AS dUserID, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
  ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
ORDER BY numrows DESC
```

Add dog breed and dog weight to the columns that will be included in the final output of your query. In addition, use a HAVING clause to include only UserIDs who would have more than 10 rows in the output of the left join (your output should contain 5 rows).

In [23]:

```
%%sql
SELECT DistinctUUsersID.user_guid AS uUserID, d.breed, d.weight, count(*) AS numrows
FROM (SELECT DISTINCT u.user_guid FROM users u) AS DistinctUUsersID
LEFT JOIN dogs d
  ON DistinctUUsersID.user_guid=d.user_guid
GROUP BY DistinctUUsersID.user_guid
HAVING numrows > 10
ORDER BY numrows DESC;
```

```
 * mysql://studentuser:***@localhost/dognitiondb
5 rows affected.
```

Out[23]:

uUserID	breed	weight	numrows
ce7b75bc-7144-11e5-ba71-058fbc01cf0b	Shih Tzu	190	1819
ce225842-7144-11e5-ba71-058fbc01cf0b	Shih Tzu	190	26
ce2258a6-7144-11e5-ba71-058fbc01cf0b	Shih Tzu	190	20
ce135e14-7144-11e5-ba71-058fbc01cf0b	Shih Tzu	190	13
ce29675e-7144-11e5-ba71-058fbc01cf0b	Labrador Retriever- Mix	60	11

You can see that almost all of the UserIDs that are causing problems are Shih Tzus that weigh 190 pounds. As we learned in earlier lessons, Dognition used this combination of breed and weight to code for testing accounts. These UserIDs do not represent real data. These types of testing entries would likely be cleaned out of databases used in large established companies, but could certainly still be present in either new databases that are still being prepared and configured, or in small companies which have not had time or resources to perfect their data storage.

There are not very many incorrect entries in the Dognition database and most of the time these entries will not appreciably affect your queries or analyses. However, you have now seen the effects such entries can have in the rare cases when you need to implement outer joins on tables that have duplicate rows or linking columns with many to many relationships. Hopefully, understanding these rare cases has helped you understand more deeply the fundamental concepts behind joining tables in relational databases.

Feel free to practice more subqueries below!

In [ ]:

```

```