MySQL_Exercise_04_Summarizing_Your_Data

November 29, 2020

1 MySQL Exercise 4: Summarizing your Data

Last week you practiced retrieving and formatting selected subsets of raw data from individual tables in a database. In this lesson we are going to learn how to use SQL to run calculations that summarize your data without having to output all the raw rows or entries. These calculations will serve as building blocks for the queries that will address our business questions about how to improve Dognition test completion rates.

These are the five most common aggregate functions used to summarize information stored in tables:

You will use COUNT and SUM very frequently.

COUNT is the only aggregate function that can work on any type of variable. The other four aggregate functions are only appropriate for numerical data.

All aggregate functions require you to enter either a column name or a "*" in the parentheses after the function word.

Let's begin by exploring the COUNT function.

1.1 1. The COUNT function

First, load the sql library and the Dognition database, and set dognition as the default database.

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

The Jupyter interface conveniently tells us how many rows are in our query output, so we can compare the results of the COUNT function to the results of our SELECT function. If you run:

```
SELECT breed
FROM dogs
```

Jupyter tells that 35050 rows are "affected", meaning there are 35050 rows in the output of the query (although, of course, we have limited the display to only 1000 rows at a time).

Now try running:

```
SELECT COUNT(breed)
FROM dogs
```

[ ]:
```
%%sql
SELECT COUNT(breed)
FROM dogs;
```

COUNT is reporting how many rows are in the breed column in total. COUNT should give you the same output as Jupyter's output without displaying the actual rows of data that are being aggregated.

You can use DISTINCT (which you learned about in MySQL Exercise 3) with COUNT to count all the unique values in a column, but it must be placed inside the parentheses, immediately before the column that is being counted. For example, to count the number of distinct breed names contained within all the entries in the breed column you could query:

```
SELECT COUNT(DISTINCT breed)
  FROM dogs
```

What if you wanted to know how many indivdual dogs successfully completed at least one test?

Since every row in the complete_tests table represents a completed test and we learned earlier that there are no NULL values in the created_at column of the complete_tests table, any non-null Dog_Guid in the complete_tests table will have completed at least one test. When a column is included in the parentheses, null values are automatically ignored. Therefore, you could use:

```
SELECT COUNT(DISTINCT Dog_Guid)
  FROM complete_tests
```

Question 1: Try combining this query with a WHERE clause to find how many individual dogs completed tests after March 1, 2014 (the answer should be 13,289):

[ ]:
```
%%sql
SELECT COUNT(DISTINCT dog_guid)
FROM complete_tests
WHERE DATE(updated_at) >= '2014-3-1';
```

You can use the "*" in the parentheses of a COUNT function to count how many rows are in the entire table (or subtable). There are two fundamental difference between COUNT(*) and COUNT(column_name), though.

The first difference is that you cannot use DISTINCT with COUNT(*).

Question 2: To observe the second difference yourself first, count the number of rows in the dogs table using COUNT(*):

[ ]:
```
%%sql
SELECT COUNT(*)
FROM dogs;
```

Question 3: Now count the number of rows in the exclude column of the dogs table:

[ ]:
```
%%sql
SELECT COUNT(exclude)
FROM dogs;
```

The output of the second query should return a much smaller number than the output of the first query. That's because:

> When a column is included in a count function, null values are ignored in the count.
> When an asterisk is included in a count function, nulls are included in the count.

This will be both useful and important to remember in future queries where you might want to use SELECT(*) to count items in multiple groups at once.

Question 4: How many distinct dogs have an exclude flag in the dogs table (value will be "1")? (the answer should be 853)

[ ]:
```
%%sql
SELECT COUNT(DISTINCT dog_guid)
FROM dogs
WHERE exclude = 1;
```

1.2 2. The SUM Function

The fact that the output of:

```
SELECT COUNT(exclude)
FROM dogs
```

was so much lower than:

```
SELECT COUNT(*)
FROM dogs
```

suggests that there must be many NULL values in the exclude column. Conveniently, we can combine the SUM function with ISNULL to count exactly how many NULL values there are. Look up "ISNULL" at this link to MySQL functions I included in an earlier lesson:

http://www.w3resource.com/mysql/mysql-functions-and-operators.php

You will see that ISNULL is a logical function that returns a 1 for every row that has a NULL value in the specified column, and a 0 for everything else. If we sum up the number of 1s outputted by ISNULL(exclude), then, we should get the total number of NULL values in the column. Here's what that query would look like:

```
SELECT SUM(ISNULL(exclude))
FROM dogs
```

It might be tempting to treat SQL like a calculator and leave out the SELECT statement, but you will quickly see that doesn't work.

> *Every SQL query that extracts data from a database MUST contain a SELECT statement.* <mark>

Try counting the number of NULL values in the exclude column:

[ ]:
```
%%sql
SELECT SUM(ISNULL(exclude))
FROM dogs;
```

The output should return a value of 34,025. When you add that number to the 1025 entries that have an exclude flag, you get a total of 35,050, which is the number of rows reported by SELECT COUNT(*) from dogs.

1.3 3. The AVG, MIN, and MAX Functions

AVG, MIN, and MAX all work very similarly to SUM.

During the Dognition test, customers were asked the question: "How surprising were [your dog's name]'s choices?" after completing a test. Users could choose any number between 1 (not surprising) to 9 (very surprising). We could retrieve the average, minimum, and maximum rating customers gave to this question after completing the "Eye Contact Game" with the following query:

```
SELECT test_name,
AVG(rating) AS AVG_Rating,
MIN(rating) AS MIN_Rating,
MAX(rating) AS MAX_Rating
FROM reviews
WHERE test_name="Eye Contact Game";
```

This would give us an output with 4 columns. The last three columns would have titles reflecting the names inputted after the AS clauses. Recall that if you want to title a column with a string of text that contains a space, that string will need to be enclosed in quotation marks after the AS clause in your query.

Question 5: What is the average, minimum, and maximum ratings given to "Memory versus Pointing" game? (Your answer should be 3.5584, 0, and 9, respectively)

[ ]:
```
%%sql
SELECT test_name,
AVG(rating) AS AVG_Rating,
MIN(rating) AS MIN_Rating,
MAX(rating) AS MAX_Rating,
COUNT(rating) AS TOTAL_Ratings
FROM reviews
WHERE test_name = "Memory versus Pointing";
```

What if you wanted the average rating for each of the 40 tests in the Reviews table? One way to do that with the tools you know already is to write 40 separate queries like the ones you wrote above for each test, and then copy or transcribe the results into a separate table in another program like Excel to assemble all the results in one place. That would be a very tedious and time-consuming exercise. Fortunately, there is a very simple way to produce the results you want within one query. That's what we will learn how to do in MySQL Exercise 5. However, it is important that you

feel comfortable with the syntax we have learned thus far before we start taking advantage of that functionality. Practice is the best way to become comfortable!

1.4 Practice incorporating aggregate functions with everything else you've learned so far in your own queries.

Question 6: How would you query how much time it took to complete each test provided in the exam_answers table, in minutes? Title the column that represents this data "Duration." Note that the exam_answers table has over 2 million rows, so if you don't limit your output, it will take longer than usual to run this query. (HINT: use the TIMESTAMPDIFF function described at: http://www.w3resource.com/mysql/date-and-time-functions/date-and-time-functions.php. It might seem unkind of me to keep suggesting you look up and use new functions I haven't demonstrated for you, but I really want you to become confident that you know how to look up and use new functions when you need them! It will give you a very competative edge in the business world.)

[ ]:
```
%%sql
SELECT test_name, AVG(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS AVG_Time
FROM exam_answers
GROUP BY test_name;
```

Question 7: Include a column for Dog_Guid, start_time, and end_time in your query, and examine the output. Do you notice anything strange?

[ ]:
```
%%sql
SELECT dog_guid, start_time, end_time, AVG(TIMESTAMPDIFF(MINUTE, start_time,␣
 ↪end_time)) AS AVG_Time
FROM exam_answers
GROUP BY dog_guid
LIMIT 200;
```

If you explore your output you will find that some of your calculated durations appear to be "0." In some cases, you will see many entries from the same Dog_ID with the same start time and end time. That should be impossible. These types of entries probably represent tests run by the Dognition team rather than real customer data. In other cases, though, a "0" is entered in the Duration column even though the start_time and end_time are different. This is because we instructed the function to output the time difference in minutes; unless you change your settings, it will output "0" for any time differences less than the integer 1. If you change your function to output the time difference in seconds, the duration in most of these columns will have a non-zero number.

Question 8: What is the average amount of time it took customers to complete all of the tests in the exam_answers table, if you do not exclude any data (the answer will be approximately 587 minutes)?

[ ]:
```
%%sql
SELECT AVG(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS AVG_Time
FROM exam_answers;
```

Question 9: What is the average amount of time it took customers to complete the "Treat Warm-Up" test, according to the exam_answers table (about 165 minutes, if no data is excluded)?

[ ]:
```
%%sql
SELECT test_name, AVG(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS AVG_Time
FROM exam_answers
WHERE test_name = "Treat Warm-Up";
```

Question 10: How many possible test names are there in the exam_answers table?

[ ]:
```
%%sql
SELECT COUNT(DISTINCT test_name)
FROM exam_answers;
```

You should have discovered that the exam_answers table has many more test names than the completed_tests table. It turns out that this table has information about experimental tests that Dognition has not yet made available to its customers.

Question 11: What is the minimum and maximum value in the Duration column of your query that included the data from the entire table?

[ ]:
```
%%sql
SELECT MIN(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS MIN_Time,
MAX(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS MAX_Time
FROM exam_answers;
```

The minimum Duration value is *negative*! The end_times entered in rows with negative Duration values are earlier than the start_times. Unless Dognition has created a time machine, that's impossible and these entries must be mistakes.

Question 12: How many of these negative Duration entries are there? (the answer should be 620)

[ ]:
```
%%sql
SELECT COUNT(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS NEG_Durations
FROM exam_answers
WHERE TIMESTAMPDIFF(MINUTE, start_time, end_time) < 0 ;
```

Question 13: How would you query all the columns of all the rows that have negative durations so that you could examine whether they share any features that might give you clues about what caused the entry mistake?

[ ]:
```
%%sql
SELECT *
FROM exam_answers
WHERE TIMESTAMPDIFF(MINUTE, start_time, end_time) < 0 ;
```

Question 14: What is the average amount of time it took customers to complete all of the tests in the exam_answers table when the negative durations are excluded from

your calculation (you should get 11233 minutes)?

[ ]:
```
%%sql
SELECT AVG(TIMESTAMPDIFF(MINUTE, start_time, end_time)) AS AVG_Time
FROM exam_answers
WHERE TIMESTAMPDIFF(MINUTE, start_time, end_time) > 0 ;
```

You have just seen another first-hand example of how messy real-world data can be, and how easy it can be to miss the "mess" when your data sets are too large to examine thoroughly by eye. Before continuing on to the next SQL lesson, make sure to watch the video about how you as a data analyst can practice building habits that will prevent you from being fooled by messy data.

And, as always, feel free to practice more queries here!

[ ]:
```

```