MySQL_Exercise_05_Summaries_of_Groups_of_Data

November 30, 2020

1 MySQL Exercise 5: Summaries of Groups of Data

So far you've learned how to select, reformat, manipulate, order, and summarize data from a single table in database. In this lesson, you are going to learn how to summarize multiple subsets of your data in the same query. The method for doing this is to include a "GROUP BY" clause in your SQL query.

1.1 The GROUP BY clause

The GROUP BY clause comes after the WHERE clause, but before ORDER BY or LIMIT:

The GROUP BY clause is easy to incorporate into your queries. In fact, it might be a little too easy to incorporate into MySQL queries, because it can be used incorrectly in MySQL queries even when no error message is displayed. As a consequence, I suggest you adopt a healthy dose of caution every time you use the GROUP BY clause. By the end of this lesson, you will understand why. When used correctly, though, GROUP BY is one of the most useful and efficient parts of an SQL query, and once you are comfortable using it, you will use it very frequently.

To get started, load the SQL library and the Dognition database, and set the dognition database as the default:

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

Let's return to a question from MySQL Exercise 4. How would you query the average rating for each of the 40 tests in the Reviews table? As we discussed, one very inefficient method to do that would be to write 40 separate queries with each one having a different test name in the WHERE conditional clause. Then you could copy or transcribe the results from all 40 queries into one place. But that wouldn't be very pleasant. Here's how you could do the same thing using one query that has a GROUP BY clause:

```
SELECT test_name, AVG(rating) AS AVG_Rating
FROM reviews
GROUP BY test_name
```

This query will output the average rating for each test. More technically, this query will instruct MySQL to average all the rows that have the same value in the test_name column.

Notice that I included test_name in the SELECT statement. As a strong rule of thumb, if you are grouping by a column, you should also include that column in the SELECT statement. If you don't do this, you won't know to which group each row of your output corresponds.

To see what I mean, try the query above without test_name included in the SELECT statement:

```
[ ]: %%sql
     SELECT test_name, AVG(rating) AS AVG_Rating
     FROM reviews
     GROUP BY test_name;
```

You can form groups using derived values as well as original columns. To illustrate this, let's address another question: how many tests were completed during each month of the year?

To answer this question, we need to take advantage of another datetime function described in the website below:

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

MONTH() will return a number representing the month of a date entry. To get the total number of tests completed each month, you could put the MONTH function into the GROUP BY clause, in this case through an alias:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
GROUP BY Month;
```

You can also group by multiple columns or derived fields. If we wanted to determine the total number of each type of test completed each month, you could include both "test_name" and the derived "Month" field in the GROUP BY clause, separated by a comma.

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
GROUP BY test_name, Month;
```

MySQL allows you to use aliases in a GROUP BY clause, but some database systems do not. If you are using a database system that does NOT accept aliases in GROUP BY clauses, you can still group by derived fields, but you have to duplicate the calculation for the derived field in the GROUP BY clause in addition to including the derived field in the SELECT clause:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
GROUP BY test_name, MONTH(created_at);
```

Try the query once with test_name first in the GROUP BY list, and once with Month first in the GROUP BY list below. Inspect the outputs:

```
[ ]: %%sql
```

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS␣
↪Completed_Tests
FROM complete_tests
GROUP BY MONTH(created_at), test_name;
```

Notice that in the first case, the first block of rows share the same test_name, but are broken up into separate months (for those of you who took the "Data Visualization and Communication with Tableau" course of this specialization, this is similar to what would happen if you put test_name first and created_at second on the rows or columns shelf in Tableau).

In the second case, the first block of rows share the same month, but are broken up into separate tests (this is similar to what would happen if you put created_at first and test_name second on the rows or columns shelf in Tableau). If you were to visualize these outputs, they would look like the charts below.

Different database servers might default to ordering the outputs in a certain way, but you shouldn't rely on that being the case. To ensure the output is ordered in a way you intend, add an ORDER BY clause to your grouped query using the syntax you already know and have practiced:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
GROUP BY test_name, Month
ORDER BY test_name ASC, Month ASC;
```

Question 1: Output a table that calculates the number of distinct female and male dogs in each breed group of the Dogs table, sorted by the total number of dogs in descending order (the sex/breed_group pair with the greatest number of dogs should have 8466 unique Dog_Guids):

[ ]:
```
%%sql
SELECT COUNT(DISTINCT(dog_guid)) as number_of_dogs, breed_group, gender
FROM dogs
GROUP BY breed_group, gender
ORDER BY COUNT(DISTINCT(dog_guid)) DESC;
```

Some database servers, including MySQL, allow you to use numbers in place of field names in the GROUP BY or ORDER BY fields to reduce the overall length of the queries. I tend to avoid this abbreviated method of writing queries because I find it challenging to troubleshoot when you are writing complicated queries with many fields, but it does allow you to write queries faster. To use this method, assign each field in your SELECT statement a number acording to the order the field appears in the SELECT statement. In the following statement:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
```

test_name would be #1, Month would be #2, and Num_Completed_Tests would be #3. You could then rewrite the query above to read:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
GROUP BY 1, 2
ORDER BY 1 ASC, 2 ASC;
```

Question 2: Revise the query your wrote in Question 1 so that it uses only numbers in the GROUP BY and ORDER BY fields.

[ ]:
```
%%sql
SELECT COUNT(DISTINCT(dog_guid)) as number_of_dogs, breed_group, gender
FROM dogs
GROUP BY 2, 3
ORDER BY 1 DESC;
```

1.2 The HAVING clause

Just like you can query subsets of rows using the WHERE clause, you can query subsets of aggregated groups using the HAVING clause. However, wheras the expression that follows a WHERE clause has to be applicable to each row of data in a column, the expression that follows a HAVING clause has to be applicable or computable using a group of data.

If you wanted to examine the number of tests completed only during the winter holiday months of November and December, you would need to use a WHERE clause, because the month a test was completed in is recorded in each row. Your query might look like this:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
WHERE MONTH(created_at)=11 OR MONTH(created_at)=12
GROUP BY 1, 2
ORDER BY 3 DESC;
```

If you then wanted to output only the test-month pairs that had at least 20 records in them, you would add a HAVING clause, because the stipulation of at least 20 records only makes sense and is only computable at the aggregated group level:

```
SELECT test_name, MONTH(created_at) AS Month, COUNT(created_at) AS Num_Completed_Tests
FROM complete_tests
WHERE MONTH(created_at)=11 OR MONTH(created_at)=12
GROUP BY 1, 2
HAVING COUNT(created_at)>=20
ORDER BY 3 DESC;
```

Question 3: Revise the query your wrote in Question 2 so that it (1) excludes the NULL and empty string entries in the breed_group field, and (2) excludes any groups that don't have at least 1,000 distinct Dog_Guids in them. Your result should contain 8 rows. (HINT: sometimes empty strings are registered as non-NULL values. You might want to include the following line somewhere in your query to exclude these values as well):

```
breed_group!=""
```

[ ]:
```
%%sql
SELECT COUNT(DISTINCT(dog_guid)) as number_of_dogs, breed_group, gender
FROM dogs
WHERE breed_group IS NOT NULL AND breed_group <> ""
```

```
GROUP BY 2, 3
HAVING number_of_dogs >= 1000
ORDER BY 1 DESC;
```

We will review several issues that can be tricky about using GROUP BY in your queries in the next lesson, but those issues will make more sense once you are sure you are comfortable with the basic functionality of the GROUP BY and HAVING clauses.

1.3 Practice incorporating GROUP BY and HAVING into your own queries.

Question 4: Write a query that outputs the average number of tests completed and average mean inter-test-interval for every breed type, sorted by the average number of completed tests in descending order (popular hybrid should be the first row in your output).

[ ]:
```
%%sql
SELECT breed_type, AVG(total_tests_completed) AS AVG_tests, AVG(mean_iti_days)␣
 ↪AS AVG_days
FROM dogs
GROUP BY breed_type
ORDER BY AVG_tests DESC;
```

Question 5: Write a query that outputs the average amount of time it took customers to complete each type of test where any individual reaction times over 6000 hours are excluded and only average reaction times that are greater than 0 are included (your output should end up with 58 rows).

// COMMENT : it says 6000 hours but answer sheet compares with minutes

[ ]:
```
%%sql
SELECT test_name, AVG(TIMESTAMPDIFF(HOUR, start_time, end_time)) AS AVG_Time
FROM exam_answers
WHERE TIMESTAMPDIFF(HOUR, start_time, end_time) < 6000
GROUP BY test_name
HAVING AVG(TIMESTAMPDIFF(HOUR, start_time, end_time)) > 0
ORDER BY AVG_Time DESC;
```

Question 6: Write a query that outputs the total number of unique User_Guids in each combination of State and ZIP code (postal code) in the United States, sorted first by state name in ascending alphabetical order, and second by total number of unique User_Guids in descending order (your first state should be AE and there should be 5043 rows in total in your output).

[ ]:
```
%%sql
SELECT state, zip, COUNT(DISTINCT user_guid) as guid_nums
FROM users
WHERE country ="US"
GROUP BY state, zip
```

```
ORDER BY state ASC, guid_nums DESC;
```

Question 7: Write a query that outputs the total number of unique User_Guids in each combination of State and ZIP code in the United States *that have at least 5 users*, sorted first by state name in ascending alphabetical order, and second by total number of unique User_Guids in descending order (your first state/ZIP code combination should be AZ/86303).

[ ]:
```
%%sql
SELECT state, zip, COUNT(DISTINCT user_guid) as users
FROM users
WHERE country = "US"
GROUP BY state, zip
HAVING users >= 5
ORDER BY state ASC, users DESC;
```

Be sure to watch the next video before beginning Exercise 6, the next set of MySQL exercises, and feel free to practice any other queries you wish below!

[ ]:
```
```