Extra 4 SQL Tricks Every Data Scientist Should Know

Part 2 of getting more out of SQL to step up your analytics work

Yi Li Follow
Apr 20 · 6 min read ★

In my previous blog, 6 SQL Tricks Every Data Scientist Should Know, I discussed the most useful SQL functionalities in my daily work as a data scientist. This blog, as Part 2 of this mini-series, I will continue to share my SQL tips to help you level up your analytics/reporting efficiency.

Today, we will be working with a new toy table shown below containing multiple data types, and for demonstration purpose, this exercise will be implemented in MS SQL server 2017,

ID_Var	Num_Var	Gen_Var	Date_Var
ID, used as the join key	float number	gender (coded as string)	date
0155	14.35	Female	6/3/2018
0155	60.79	Female	6/1/2018
0155	60.79	Female	6/2/2018
0155	79.77	Female	9/29/2018
0155	122.82	Female	9/30/2018
0180	1810.47	Female	6/30/2016
0188	2732.62	Female	6/30/2016
0188	2782.89	Female	7/31/2016
0188	2989.38	Female	8/31/2016
0792	721.36	Male	6/30/2016
0792	817.94	Male	7/1/2016
0792	886.17	Male	8/31/2016
0792	954.71	Male	9/30/2016
0792	1048.69	Male	10/31/2016

Toy data table (with variable definitions)

1. **ROW_NUMBER() to return a subset of rows**

The ROW_NUMBER() function in SQL creates a unique incrementing integer value to each row of the result. This column of values is considered a **pseudo-column** as it does not inherently exist in our data table, and because of which, the result is returned in the order determined by the analysts in the ORDER BY clause.

With this pseudo-column, we can solve the "top N" questions using a simple query, and the following demonstrates how to select the top 3 records with the highest Num_Var value for each ID,

Here is the returned subset (and of course, you can remove the column RowNumber from the output),

RowNumber	ID_Var	Num_Var
1	0155	122.82
2	0155	79.77
3	0155	60.79
1	0180	1810.47
1	0188	2989.38
2	0188	2782.89
3	0188	2732.62
1	0792	1048.69
2	0792	954.71
3	0792	886.17

Output: top 3 records with the highest Num_Var value

Noticed that in spite of being only one row, ID 0180 was returned correctly just as other IDs (with more than 3 rows). Hence, no extra caution needed in regard to whether the raw data have enough rows for the number N specified. Straightforward!

2. Compute the length of consecutive days with the ROW_NUMBER()(advanced)

Knowing the concept of the ROW_NUMBER(), you must be thinking about another possible application in time series, where we calculate the consecutive time the event of interest lasts.

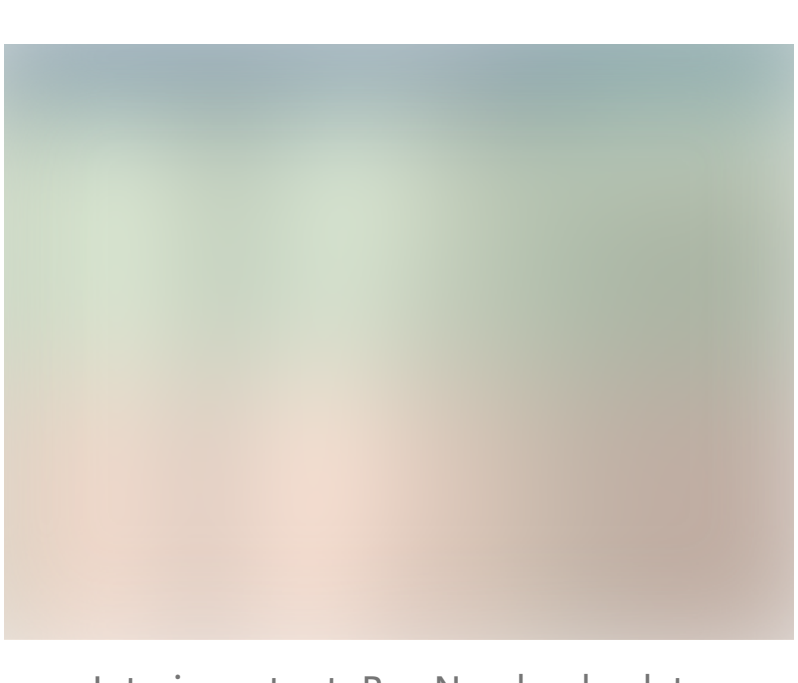
What do I mean? Well, let's consider our toy data being customers logging in to our website. Customer ID 0155 logged in on 06/01, 06/02, and 06/03 (i.e., 3 consecutive days), and then there were another 2 consecutive logins on 09/29 and 09/30 each. Visually it is not that simple for us to do this calculation if the dates are not sorted. How a SQL query could help?

An intuitive way would be first using the ROW_NUMBER() to assign row numbers ordered by dates within each customer ID. Because the integer row numbers happen to be incremental by 1 also, the length of consecutive days is just the difference between the last date and the first date! Unfortunately, it won't work because it would give us a 5, rather than 3 and 2 (for customer 0155).

This brings up an important note on the ROW_NUMBER() function: **it has no gaps!** It creates an ever-increasing value that always starts with 1 for each partition, whereas our data has gaps (e.g., 06/03 and 09/29 are not consecutive). Thus, simply applying ROW_NUMBER() cannot get us the desired output. We need some tweaks, and here's how,

Step 1: Create the RowNumber (order by date ASC)

which gives us this interim output, and let's label it ***output_table_with_RowNumber*** for easy reference below,



Interim output: RowNumber by date

Step 2: Date_Var — RowNumber to group the consecutive days (i.e., dates with the same starting points for counting)

and here's the return, which is labeled as ***output_table_with_grouping_var***,

Interim output: Creating a grouping variable

now, don't get hung up on the values of this derived variable ***Starting_Count_DT*** because it's meaningless. It is just a grouping variable to basically tell SQL that these rows are consecutive (i.e., incremental by 1).

Step 3: Calculate the length (of consecutive days) grouped by the Starting_Count_DT

There is our final output!

Output: length of consecutive days

To put everything together,

Phew, problem solved! If you don't get it the first time, no worries, you can put this into your data science code snippets stash, and refer back whenever you need to. *Spoil alert*, don't be surprised to encounter this exact problem in your next data scientist position interview with the big tech companies 😄

3. WITH statement to keep it DRY

The previous query, as we just saw, is a fairly complex one containing 2 nested sub-queries. These sub-queries add more complexity to read and debug our code in the sense that if we were to pass it to other teammates, chances are they would have no idea what this query is doing.

Now the WITH clause (a.k.a., subquery factoring) to the rescue! Let's see how we can re-write the query above to make it more understandable,

Here, (1) by leveraging the WITH clause, we broke down the nested queries into 2 individual temp views for better code readability; (2) no requirement for the relations among the multiple WITH queries, meaning they can be independent or dependent (like in our example).

Another useful scenario of the WITH clause is when the WITH query is referenced multiple times. Without repeatedly writing the same sub-query again and again (i.e., the DRY / Don't Repeat Yourself principle), we can introduce upfront and reuse it later.

4. Concatenating to re-format your data structure

Moving on, our next task is to create an aggregated report for our internal stakeholders, showing the dates each customer logged in. This will be an aggregation of many-to-many information. Also observing that the ID_Var and Gen_Var both contain duplicated values across rows, therefore to make it suitable for human reading, we immediately think of the STRING_AGG() function in MS SQL 2017,

this code snippet yields,

Output: Dates concatenated

As we can see, (1) ordering the elements in the concatenated list is allowed by simply specifying the ORDER BY statement (e.g., ID 0155); (2) despite of being more commonly used for string concatenation, the STRING_AGG() function also works for other data types (e.g., the date and number variables in our example).

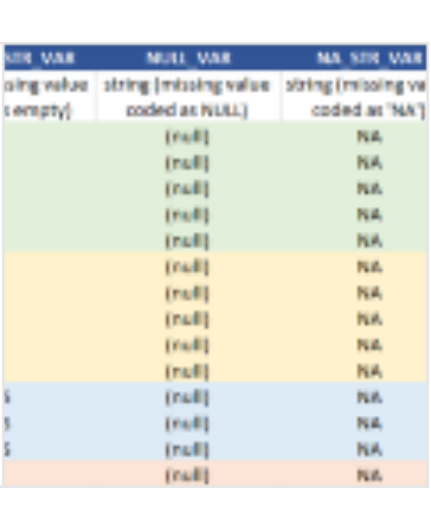
Now, a couple of ***callouts*** regarding this aggregation functionality:

(1) although it is available in major SQL databases, the specific function name varies. In Oracle/PLSQL (11g), it's the **LISTAGG()**, in MySQL along with IBM Netezza, it's the **GROUP_CONCAT()**, in MS SQL server 2017 and PostgresSQL, it's the **STRING_AGG()**.

(2) besides making the data view simpler, I find it also useful in scenarios where we need to pull data into R or Python for in-depth analysis (by the way, check out my previous post for tips of extracting data from SQL database to Python if you haven't done so). Instead of transferring the entire raw data table as is with most columns having duplicates, we can first concatenate the field with varying values into one row per ID to reduce the data size and then parse this field in R or Python. In this concatenation, Null values will be excluded, and thus, we don't have to worry about missing value handling at this point.

So there you have it — 4 additional SQL tips for data analytics! I hope you this mini-series inspiring, and enjoyed reading it as much as me writing it. As always, all the code snippets together with the toy data is available here in my Github 😀

6 SQL Tricks Every Data Scientist Should Know
SQL tricks to make your analytics work more efficient
towardsdatascience.com



WRITTEN BY
Yi Li
Data Scientist
Follow