**What are window functions?**

**Window functions** perform calculations on a set of rows that are related together. But, unlike the aggregate functions, windowing functions do not collapse the result of the rows into a single value. Instead, all the rows maintain their original identity and the calculated result is returned for every row.

## **1. Sum**

One of the most common use cases for the SUM window function is calculating a running sum. Let’s look at an example:

SELECT o.occurred\_at,  
 SUM(o.gloss\_qty) OVER(ORDER BY o.occurred\_at) as running\_gloss\_orders  
 FROM demo.orders o

To break down the syntax here, SUM(o.gloss\_qty) defines the aggregation—we’re going to be taking a sum of gloss\_qty. Next, the OVER(ORDER BY o.occurred\_at) clause tells your SQL engine to go row-by-row and take the sum of every record of gloss\_qty in order by occurred\_at up until the row it’s currently looking at.

We can also create separate running sums for different groups within our dataset by adding a PARTITION BY as seen below:

SELECT o.occurred\_at,  
 o.account\_id,  
 SUM(o.gloss\_qty) OVER(PARTITION BY o.account\_id   
 ORDER BY o.occurred\_at) as running\_gloss\_orders  
 FROM demo.orders o

## **2. Row Number**

Ah, row number. This is my favorite window function. I commonly use ROW\_NUMBER when I want to return the most recent record of a table I’m working with. ROW\_NUMBER simply returns the number of the given row starting with 1, which is the first record as defined by the ORDER BY. No need to specify a variable in the parenthesis here!

WITH  
 order\_ranks as (  
 SELECT o.id,  
 o.account\_id,  
 o.gloss\_qty,  
 o.gloss\_amt\_usd,  
 ROW\_NUMBER() OVER(PARTITION BY o.account\_id   
 ORDER BY o.occurred\_at DESC) as acct\_order\_rank  
 FROM demo.orders o  
)  
  
 SELECT \*  
 FROM order\_ranks  
 WHERE acct\_order\_rank = 1

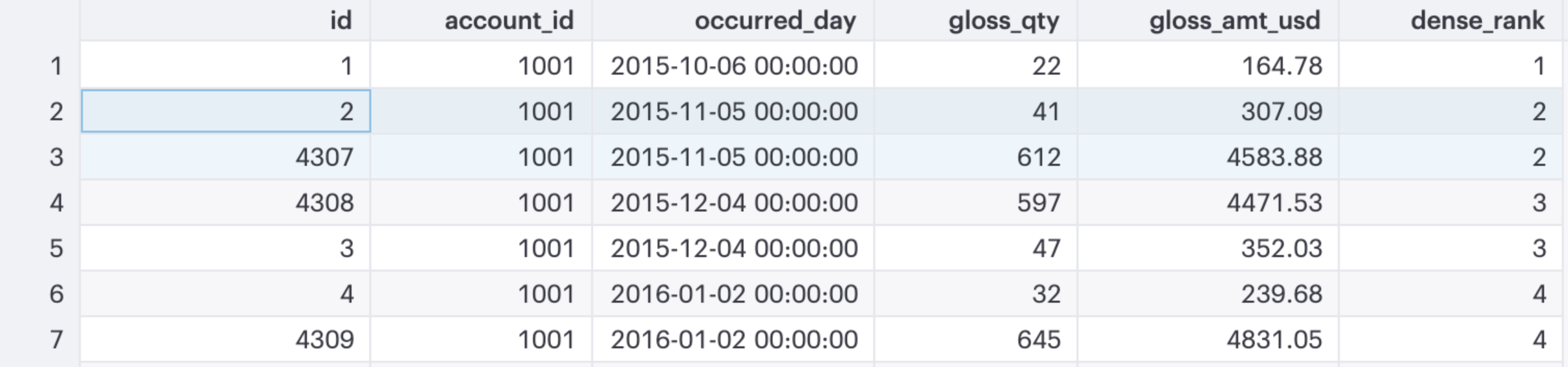
Note the DESC in the ORDER BY —this tells the SQL engine to order rows by occurred\_at in descending order (i.e. latest record first.) The query above returns the most recent record in the orders table for each account. This query also makes use of CTEs (if those are a foreign concept, read [our latest post on them](https://mode.com/blog/use-common-table-expressions-to-keep-your-sql-clean/)).

## **3. Dense Rank**

DENSE\_RANK works similar to ROW\_NUMBER but differs in how it handles cases where two rows for the field specified in your ORDER BY have identical values. DENSE\_RANK will assign rows with identical ORDER\_BY values the same number, whereas ROW\_NUMBER will produce distinct values.

SELECT o.id,  
 o.account\_id,  
 DATE\_TRUNC('day',o.occurred\_at) as occurred\_day,  
 o.gloss\_qty,  
 o.gloss\_amt\_usd,  
 DENSE\_RANK() OVER(PARTITION BY o.account\_id   
 ORDER BY DATE\_TRUNC('day',o.occurred\_at)) as dense\_rank  
 FROM demo.orders o

You can see in the results below that the 2nd and 3rd rows have the same dense\_rank value of 2. Oh, and you’ll notice that the window function above uses a value derived from another SQL function DATE\_TRUNC —nesting SQL functions inside a window function is fair game.



## **4. Lag**

The LAG function is a favorite among time series junkies. This function allows you to compare a row to any of the rows preceding it. So for instance, if you wanted to find out how one order of gloss\_qty compared to the previous orders, this is the function to use.

SELECT o.id,  
 o.occurred\_at,  
 o.gloss\_qty,  
 LAG(gloss\_qty,1) OVER(ORDER BY o.occurred\_at) as order\_lag\_1,  
 LAG(gloss\_qty,2) OVER(ORDER BY o.occurred\_at) as order\_lag\_2,  
 LAG(gloss\_qty,3) OVER(ORDER BY o.occurred\_at) as order\_lag\_3  
 FROM demo.orders o

Keep in mind that the first n rows of your data will be NULL (where n is the number of “lags” you specify). You can see this in the results of our query:



## **5. Max**

And finally, we can top off this list with MAX. MAX works in the same way as SUM so I’m going to spice up the example with some more advanced syntax. Let’s say that I want to see how a current order stacks up against the highest order quantity among the previous 5 orders for gloss\_qty. We can do this by adding in some additional syntax in our ORDER BY clause.

SELECT o.id,  
 o.occurred\_at,  
 o.gloss\_qty,  
 MAX(gloss\_qty) OVER(ORDER BY o.occurred\_at   
 ROWS BETWEEN 5 PRECEDING AND 1 PRECEDING) as max\_order  
FROM demo.orders o

After we specify the field to order by, we add a definition for our window size: ROWS BETWEEN 5 PRECEDING AND 1 PRECEDING. This clause is essentially saying, "look back across the previous 5 orders (not including the current order) and take the maximum value."

This syntax is flexible and can define any window across your dataset. The syntax typically looks like:

ORDER BY [order\_var] ROWS BETWEEN window start AND  
 window\_end

where **window\_star**t and **windown\_end** take on one of the following values:

* UNBOUNDED PRECEDING (i.e. all rows before the current row)
* [VALUE] PRECEDING (where [VALUE] = # of rows behind the current row to consider)
* CURRENT ROW
* [VALUE] FOLLOWING (where [VALUE] = # of rows ahead of the current row to consider)
* UNBOUNDED FOLLOWING (i.e. all rows after the current row)

Window functions are incredibly powerful tools that enable quick and flexible analysis, all in the context of the SQL editor. Now, go find your favorite window function and analyze away!