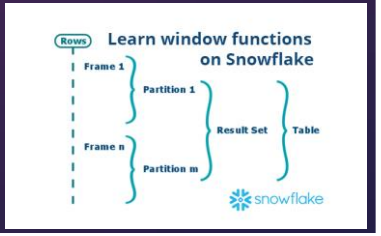
JAI SHREE GANESH

WELCOME TO WORLD OF ANALYTICS

Window Functions

- Window functions in Snowflake are a way to compute values over a group of rows.
- They return a single value for each row, in contrast to aggregate functions which return a single value for a group of rows.
- Window functions are the base of data warehousing workloads for many reasons.
- First of all they are very similar to the GROUP BY clause as they run on a set of rows instead of a single one but, but unlike the GROUP BY clause we do not lose the individual rows.
- To remedy this we could perform a self join with the result of GROUP BY operation but window functions implement the same functionality in a more efficient way that gives better performance.
- Window functions also provide complex functions that are based on the (requested) order of rows such as DENSE_RANK which would be hard to implement with traditional SQL.
- Using window functions you can avoid writing procedural SQL code.



OVERVIEW

What is a Window?

- A window is a group of related rows.
- For example, a window might be defined based on timestamps, with all rows in the same month grouped in the same window.
- Or a window might be defined based on location, with all rows from a particular city grouped in the same window.
- A window can consist of zero, one, or multiple rows.
- For simplicity, Snowflake documentation usually says that a window contains multiple rows.

snowflake

OVERVIEW

Some window functions are order-sensitive. There are two main types of order-sensitive window functions:

- Rank-related functions.
- Window frame functions.

Rank-related functions list information based on the “rank” of a row.
For example, if you rank stores in descending order by profit per year, the store with the most profit will be ranked 1; the second-most profitable store will be ranked 2, etc.

Window frame functions allow you to perform rolling operations, such as calculating a running total or a moving average, on a subset of the rows in the window.

snowflake

What is a Window Function?

A window function is any function that operates over a window of rows.

A window function is generally passed two parameters:

- A row. More precisely, a window function is passed 0 or more expressions. In almost all cases, at least one of those expressions references a column in that row. (Most window functions require at least one column or expression, but a few window functions, such as some rank-related functions, do not required an explicit column or expression.)
- A window of related rows that includes that row. The window can be the entire table, or a subset of the rows in the table.

snowflake

What is a Window Function?

For non-window functions, all arguments are usually passed explicitly to the function, for example:

```
MY_FUNCTION(argument1, argument2, …)
```

Window functions behave differently; although the current row is passed as an argument the normal way, the window is passed through a separate clause, called an `OVER` clause. The syntax of the `OVER` clause is documented later.

snowflake

Why use window functions?

Window functions are very powerful, and once you've gotten your head around how to use them, you'll be surprised at what they allow you to do.

Some common use cases are:

- Running/Cumulative Total
- Moving Average
- Rank rows by custom criteria and groupings
- Finding the year-over-year % Change

snowflake

SYNTAX

```
<function> ( [ <argument1> [ , ... , <argumentN> ] )
  OVER (
        [ PARTITION BY <expr1> ]
        [ ORDER BY <expr2> ]
        [ frame_definition ] )
```

where the PARTITION BY denotes how to GROUP rows into partitions, ORDER BY how to order the rows in those partitions, and FRAME which determines which rows to consider in those ordered partitions.

- Each time a window function is called, it is passed a row (the current row in the window) and the window of rows that contain the current row.
- The window function returns one output row for each input row.
- The output depends on the individual row passed to the function and the values of the other rows in the window passed to the function.

snowflake

TYPES

- In general, window functions can be grouped into 3 types:
- Navigation functions: Return the value given a specific location criteria (e.g. first_value)
- Numbering functions: Assign a number (e.g. rank) to each row based on their position in the specified window
- Analytic functions: Perform a calculation on a set of values (e.g. sum)

Aggregate and sequencing functions on Snowflake

- Aggregate functions perform operations that take into account all values across a window.
  Sequencing functions produce output based on the position of the row in the window.

Aggregate functions supported by Snowflake are:

- AVG
- COUNT
- MAX
- MIN
- SUM

Sequencing and ranking functions in Snowflake are:

- ROW_NUMBER
- LAG
- LEAD
- FIRST_VALUE
- LAST_VALUE
- NTH_VALUE
- RANK
- PERCENT_RANK
- DENSE_RANK
- CUME_DIST
- NTILE
- WIDTH_BUCKET

snowflake

Navigation functions

- `FIRST_VALUE` : Returns the value_expression for the **first** row in the current window frame.

```
FIRST_VALUE( <expr> ) [ { IGNORE | RESPECT } NULLS ]
                      OVER ( [ PARTITION BY <expr1> ] ORDER BY <expr2>  [ { ASC | DESC } ] [ <window_
```

- `LAST_VALUE` : Returns the value_expression for the **last** row in the current window frame.

```
LAST_VALUE( <expr> ) [ { IGNORE | RESPECT } NULLS ]
                     OVER ( [ PARTITION BY <expr1> ] ORDER BY <expr2> [ { ASC | DESC } ] [ <window_fr
```

snowflake

Numbering functions

- `RANK` : Returns the rank of each row in the ordered partition (starts at 1).

```
RANK() OVER ( [ PARTITION BY <expr1> ] ORDER BY <expr2> [ { ASC | DESC } ] [ <window_frame> ] )
```

- `DENSE_RANK` : Returns the rank, but values of the same value get the same rank (starts at 1).

```
DENSE_RANK() OVER ( [ PARTITION BY <expr1> ] ORDER BY <expr2> [ ASC | DESC ] [ <window_frame> ] )
```

snowflake

ROW_NUMBER()

The ROW_NUMBER() is an analytic function that generates a non-persistent sequence of temporary values which are calculated dynamically when the query is executed.

The ROW_NUMBER() function assigns a unique incrementing number for each row within a partition of a result set.

The row number starts at 1 and continues up sequentially, to the end of the table.

snowflake

ROW_NUMBER() : SYNTAX

- `ROW_NUMBER` : Returns the sequential row number for each ordered partition.

```
ROW_NUMBER() OVER (
  [ PARTITION BY <expr1> [, <expr2> ... ] ]
  ORDER BY <expr3> [ , <expr4> ... ] [ { ASC | DESC } ]
  )
```

snowflake

ROW_NUMBER() : SYNTAX

Snowflake Row Number Syntax: OVER

- The OVER clause defines the window or set of rows that the ROW_NUMBER() function operates on.
- The possible components of the OVER clause are ORDER BY (required), and PARTITION BY (optional).

Snowflake Row Number Syntax: PARTITION BY

- The PARTITION BY clause divides the rows into partitions (groups of rows) to which the function is applied.
- The PARTITION BY clause is optional. When you omit it, the ROW_NUMBER() function will treat the whole result set as a single partition/group.

snowflake

ROW_NUM

- `ROW_NUMBER` : Returns the sequential row number for each ordered partition.

```
ROW_NUMBER() OVER (
  [ PARTITION BY <expr1> [, <expr2> ... ] ]
  ORDER BY <expr3> [ , <expr4> ... ] [ { ASC | DESC } ]
  )
```

Snowflake Row Number Syntax: ORDER BY

- The ORDER BY clause defines the sequential order of the rows within each partition of the result set.
- The ORDER BY clause is required, you must include it because the ROW_NUMBER() function is order sensitive.

Snowflake Row Number Syntax: Expression1 and Expression2

- expression1 and expression2 specify the column(s) or expression(s) to partition by. You can partition by one or more columns or expressions.
  For example, suppose that you are selecting data across multiple states (or provinces) and you want row numbers from 1 to N within each state; in that case, you can partition by the state.

Snowflake Row Number Syntax: expression3 and expression4

- expression3 and expression4 are a component of the Snowflake Row Number specify the column(s) or expression(s) to use to determine the order of the rows. You can order by 1 or more expressions.

snowflake

LEAD FUNCTIONS

Let me summarize the Lead in the picture below. Order the data first by the column Sale_Date and after the data sorts, then begin with row one and ask, "Can we get the Daily_Sales value from the next row, and add it to the current line?" If the answer is "Yes," then do it, but if the answer is "No," then put in a Null.

```
SELECT Product_ID, Sale_Date, Daily_Sales,
LEAD (Daily_Sales, 1) OVER (ORDER BY Sale_Date) AS Next_Value
FROM Sales_Table
WHERE Product_ID = 1000
```

Product_ID	Sale_Date	Daily_Sales	Next_Value
1000	2000-09-28	48850.40	54500.22
1000	2000-09-29	54500.22	36000.07
1000	2000-09-30	36000.07	40200.43
1000	2000-10-01	40200.43	32800.50
1000	2000-10-02	32800.50	64300.00
1000	2000-10-03	64300.00	54553.10
1000	2000-10-04	54553.10	?

The LEAD analytic puts the value of the next row on the current line.

The LEAD allows you to see the Daily_Sales for today next to tomorrow's value.

LEAD FUNCTIONS

The LEAD above allows you to see the Daily_Sales for today next to tomorrow's value.

Check out the next picture below. Notice we have two leads, and one has a lead of one row, and the other has a lead of two rows. Now, row one has the Daily_Sales for today and the Daily_Sales for the next two days.

For the Next_Value column, the moving window is one. For the Two_Days column, the moving window is two. The moving window in Lead establishes by the numbers after LEAD (Daily_Sales, 1) or LEAD(Daily_Sales,2). You can put any number (n) in the moving window, and that tells Snowflake to get the value n rows down.

LEAD FUNCTIONS

```
SELECT Product_ID, Sale_Date, Daily_Sales,
LEAD (Daily_Sales, 1) OVER (ORDER BY Sale_Date) AS Next_Value,
LEAD (Daily_Sales, 2) OVER (ORDER BY Sale_Date) AS Two_Days
FROM Sales_Table WHERE Product_ID = 1000
```

Product_ID	Sale_Date	Daily_Sales	Next_Value	Two_Days
1000	2000-09-28	48850.40	54500.22	36000.07
1000	2000-09-29	54500.22	36000.07	40200.43
1000	2000-09-30	36000.07	40200.43	32800.50
1000	2000-10-01	40200.43	32800.50	64300.00
1000	2000-10-02	32800.50	64300.00	54553.10
1000	2000-10-03	64300.00	54553.10	?
1000	2000-10-04	54553.10	?	?

Null because there is no next row after

Null because there is no row two rows after

LEAD FUNCTIONS

Now, we are going to take the next example even farther by adding a third day to the answer set, and we are also going to add the PARTITION BY statement and change our WHERE clause.

The PARTITION BY resets the calculation and acts much like a GROUP BY statement.

Notice that each Product_ID calculates within the Product_ID only. When a new Product_ID appears, the calculation starts over.

LEAD FUNCTIONS

```
SELECT Product_ID, Sale_Date, Daily_Sales,
LEAD (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date)  AS Next_Value,
LEAD (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Two_Days
FROM Sales_Table
```

Product_ID	Sale_Date	Daily_Sales	Next_Value	Two_Days
3000	2000-09-28	61301.77	34509.13	43868.86
3000	2000-09-29	34509.13	43868.86	28000.00
3000	2000-09-30	43868.86	28000.00	19678.94
3000	2000-10-01	28000.00	19678.94	21553.79
3000	2000-10-02	19678.94	21553.79	15675.33
3000	2000-10-03	21553.79	15675.33	?
3000	2000-10-04	15675.33	?	?
1000	2000-09-28	48850.40	54500.22	36000.07
1000	2000-09-29	54500.22	36000.07	40200.43
1000	2000-09-30	36000.07	40200.43	32800.50
1000	2000-10-01	40200.43	32800.50	64300.00
1000	2000-10-02	32800.50	64300.00	54553.10
1000	2000-10-03	64300.00	54553.10	?
1000	2000-10-04	54553.10	?	?

snowflake

LEAD FUNCTIONS

Now, we are going to take the next example even farther by adding a third day to the answer set and change our WHERE clause.

The PARTITION BY resets the calculation and acts much like a GROUP BY statement.

Notice that each Product_ID calculates within the Product_ID only.

When a new Product_ID appears, the calculation starts over.

LEAD FUNCTIONS

```
SELECT Product_ID as PROD, Sale_Date Sale_Date, Daily_Sales,
LEAD (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date)  AS Next,
LEAD (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Two,
LEAD (Daily_Sales, 3) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Three
FROM Sales_Table WHERE Sale_Date < '2000-10-02'
```

Prod	Sale_Date	Daily_Sales	One	Two	Three
1000	2000-09-28	48850.40	54500.22	36000.07	40200.43
1000	2000-09-29	54500.22	36000.07	40200.43	?
1000	2000-09-30	36000.07	40200.43	?	?
1000	2000-10-01	40200.43	?	?	?
2000	2000-09-28	41888.88	48000.00	49850.03	54850.29
2000	2000-09-29	48000.00	49850.03	54850.29	?
2000	2000-09-30	49850.03	54850.29	?	?
2000	2000-10-01	54850.29	?	?	?
3000	2000-09-28	61301.77	34509.13	43868.86	28000.00
3000	2000-09-29	34509.13	43868.86	28000.00	?
3000	2000-09-30	43868.86	28000.00	?	?
3000	2000-10-01	28000.00	?	?	?

snowflake

LEAD FUNCTIONS

In the picture above, notice the null values. They appear because there are no rows after available. We have run out of future row values to display. In another sense, we have all of the values needed on the first row of each Product_ID in the answer set. The first row in the answer set for each Product_ID gives us the full picture. Watch how clever this report gets when you add the QUALIFY statement. Compare the picture above with the picture below. Notice that the QUALIFY statement removes the additional rows not needed.

Prod	Sale_Date	Daily_Sales	One	Two	Three
1000	2000-09-28	48850.40	54500.22	36000.07	40200.43
1000	2000-09-29	54500.22	36000.07	40200.43	?
1000	2000-09-30	36000.07	40200.43	?	?
1000	2000-10-01	40200.43	?	?	?
2000	2000-09-28	41888.88	48000.00	49850.03	54850.29
2000	2000-09-29	48000.00	49850.03	54850.29	?
2000	2000-09-30	49850.03	54850.29	?	?
2000	2000-10-01	54850.29	?	?	?
3000	2000-09-28	61301.77	34509.13	43868.86	28000.00
3000	2000-09-29	34509.13	43868.86	28000.00	?
3000	2000-09-30	43868.86	28000.00	?	?
3000	2000-10-01	28000.00	?	?	?

snowflake

LEAD FUNCTIONS

```
SELECT Product_ID as PROD, Sale_Date Sale_Date, Daily_Sales,
LEAD (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date)  AS Next,
LEAD (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Two,
LEAD (Daily_Sales, 3) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Three
FROM Sales_Table WHERE Sale_Date < '2000-10-02'
QUALIFY Three IS NOT NULL
```

snowflake

LEAD FUNCTIONS

```
SELECT Product_ID as PROD, Sale_Date Sale_Date, Daily_Sales,
LEAD (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date)  AS Next,
LEAD (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Two,
LEAD (Daily_Sales, 3) OVER (PARTITION BY Product_ID
                            ORDER BY Sale_Date) AS Three
FROM Sales_Table WHERE Sale_Date < '2000-10-02'
QUALIFY Three IS NOT NULL
```

The **QUALIFY** statement is to Ordered Analytics much like a HAVING statement is to aggregation.

QUALIFY is a filter that works on the calculations after they calculate.

Prod	Sale_Date	Daily_Sales	One	Two	Three
1000	2000-09-28	48850.40	54500.22	36000.07	40200.43
2000	2000-09-28	41888.88	48000.00	49850.03	54850.29
3000	2000-09-28	61301.77	34509.13	43868.86	28000.00

snowflake

LAG FUNCTIONS

```
SELECT Product_ID, Sale_Date, Daily_Sales,
LAG (Daily_Sales, 1) OVER (ORDER BY Sale_Date) AS Prev_Value
FROM Sales_Table
WHERE Product_ID = 1000
```

Product_ID	Sale_Date	Daily_Sales	Prev_Value
1000	2000-09-28	48850.40	?
1000	2000-09-29	54500.22	48850.40
1000	2000-09-30	36000.07	54500.22
1000	2000-10-01	40200.43	36000.07
1000	2000-10-02	32800.50	40200.43
1000	2000-10-03	64300.00	32800.50
1000	2000-10-04	54553.10	64300.00

The LAG analytic puts the value of the previous row on the current line.

The LAG allows you to see the Daily_Sales for today next to yesterday's value.

snowflake

LAG FUNCTIONS

The LAG above allows you to see the Daily_Sales for today next to yesterday's value.

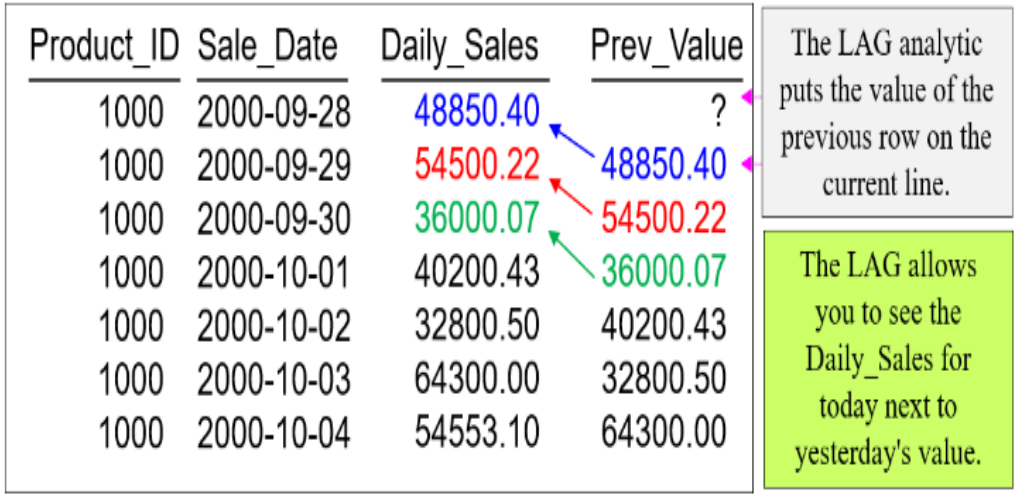
Notice the example where it has LAG (Daily_Sales, 1).

The one represents the moving window.

We want to get the value of Daily_Sales one row up. I

f there were a two instead of a one, we would want to get the Daily_Sales value two rows above.

Sometimes you will see only LAG (Daily_Sales), which defaults to LAG (Daily_Sales, 1).



Product_ID	Sale_Date	Daily_Sales	Prev_Value
1000	2000-09-28	48850.40	?
1000	2000-09-29	54500.22	48850.40
1000	2000-09-30	36000.07	54500.22
1000	2000-10-01	40200.43	36000.07
1000	2000-10-02	32800.50	40200.43
1000	2000-10-03	64300.00	32800.50
1000	2000-10-04	54553.10	64300.00

snowflake

LAG FUNCTIONS

In the example below, we are using two LAG window functions. We alias the first LAG as Prev_Value because it is capturing the Daily_Sales value one row above. We give the second Lag an alias of Prev_2_Rows because it is capturing the value two rows above. On the current line of the answer set, we see today's Daily_Sales, yesterday's Daily_Sales, and the Daily_Sales from two-days previous.

```
SELECT Product_ID, Sale_Date, Daily_Sales,
LAG (Daily_Sales, 1) OVER (ORDER BY Sale_Date) AS Prev_Value,
LAG (Daily_Sales, 2) OVER (ORDER BY Sale_Date) AS Prev_2_Rows
FROM Sales_Table WHERE Product_ID = 1000
```

Product_ID	Sale_Date	Daily_Sales	Prev_Value	Prev_2_Rows
1000	2000-09-28	48850.40	?	?
1000	2000-09-29	54500.22	48850.40	?
1000	2000-09-30	36000.07	54500.22	48850.40
1000	2000-10-01	40200.43	36000.07	54500.22
1000	2000-10-02	32800.50	40200.43	36000.07
1000	2000-10-03	64300.00	32800.50	40200.43
1000	2000-10-04	54553.10	64300.00	32800.50

snowflake

LAG FUNCTIONS

Now, we are going to take the next example even farther by adding a PARTITION BY statement. The PARTITION BY resets the calculation and acts much like a GROUP BY statement. Notice that each Product_ID calculates within the Product_ID only. When a new Product_ID appears, the LAG calculation starts over.

```
SELECT Product_ID, Sale_Date, Daily_Sales,
LAG (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date)  AS Prev_Value,
LAG (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date) AS Prev_2_Days
FROM Sales_Table
```

Product_ID	Sale_Date	Daily_Sales	Prev_Value	Prev_2_Days
3000	2000-09-28	61301.77	?	?
3000	2000-09-29	34509.13	61301.77	?
3000	2000-09-30	43868.86	34509.13	61301.77
3000	2000-10-01	28000.00	43868.86	34509.13
3000	2000-10-02	19678.94	28000.00	43868.86
3000	2000-10-03	21553.79	19678.94	28000.00
3000	2000-10-04	15675.33	21553.79	19678.94
1000	2000-09-28	48850.40	?	?
1000	2000-09-29	54500.22	48850.40	?
1000	2000-09-30	36000.07	54500.22	48850.40
1000	2000-10-01	40200.43	36000.07	54500.22
1000	2000-10-02	32800.50	40200.43	36000.07
1000	2000-10-03	64300.00	32800.50	40200.43
1000	2000-10-04	54553.10	64300.00	32800.50

snowflake

LAG FUNCTIONS

Now, in our next example, we will use PARTITION BY again, but we will have three lag statements with the last one to lag three rows back. I am doing this for a reason. I want to show you how to use the QUALIFY statement in the final example. In the picture below, notice the null values. They appear because there are no rows above available. We have run out of past values to display.

```
SELECT Product_ID as PROD, Sale_Date Sale_Date, Daily_Sales,
LAG (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date)  AS Prev,
LAG (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date) AS Prev2,
LAG (Daily_Sales, 3) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date) AS Prev3
FROM Sales_Table WHERE Sale_Date < '2000-10-02'
```

Prod	Sale_Date	Daily_Sales	Prev	Prev2	Prev3
1000	2000-09-28	48850.40	?	?	?
1000	2000-09-29	54500.22	48850.40	?	?
1000	2000-09-30	36000.07	54500.22	48850.40	?
1000	2000-10-01	40200.43	36000.07	54500.22	48850.40
2000	2000-09-28	41888.88	?	?	?
2000	2000-09-29	48000.00	41888.88	?	?
2000	2000-09-30	49850.03	48000.00	41888.88	?
2000	2000-10-01	54850.29	49850.03	48000.00	41888.88
3000	2000-09-28	61301.77	?	?	?
3000	2000-09-29	34509.13	61301.77	?	?
3000	2000-09-30	43868.86	34509.13	61301.77	?
3000	2000-10-01	28000.00	43868.86	34509.13	61301.77

snowflake

LAG FUNCTIONS

```
SELECT Product_ID as PROD, Sale_Date Sale_Date, Daily_Sales,
LAG (Daily_Sales, 1) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date)  AS Prev,
LAG (Daily_Sales, 2) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date) AS Prev2,
LAG (Daily_Sales, 3) OVER (PARTITION BY Product_ID
                           ORDER BY Sale_Date) AS Prev3
FROM Sales_Table WHERE Sale_Date < '2000-10-02'
QUALIFY Prev3 IS NOT NULL
```

Prod	Sale_Date	Daily_Sales	Prev	Prev2	Prev3
1000	2000-10-01	40200.43	36000.07	54500.22	48850.40
2000	2000-10-01	54850.29	49850.03	48000.00	41888.88
3000	2000-10-01	28000.00	43868.86	34509.13	61301.77

In the picture above, notice the null values. They appear because there are no rows above available. The last row in the answer set for each Product_ID gives us the full picture. Watch how clever this report gets when you add the QUALIFY statement. Compare the picture above with the picture below. Notice that the QUALIFY statement removes the additional rows not needed. Qualify is an additional filter that works on the analytic totals after they calculate.

The QUALIFY statement is to Ordered Analytics, much like a HAVING statement is to aggregation. Only Teradata and Snowflake have the QUALIFY statement available.

snowflake


THANK YOU