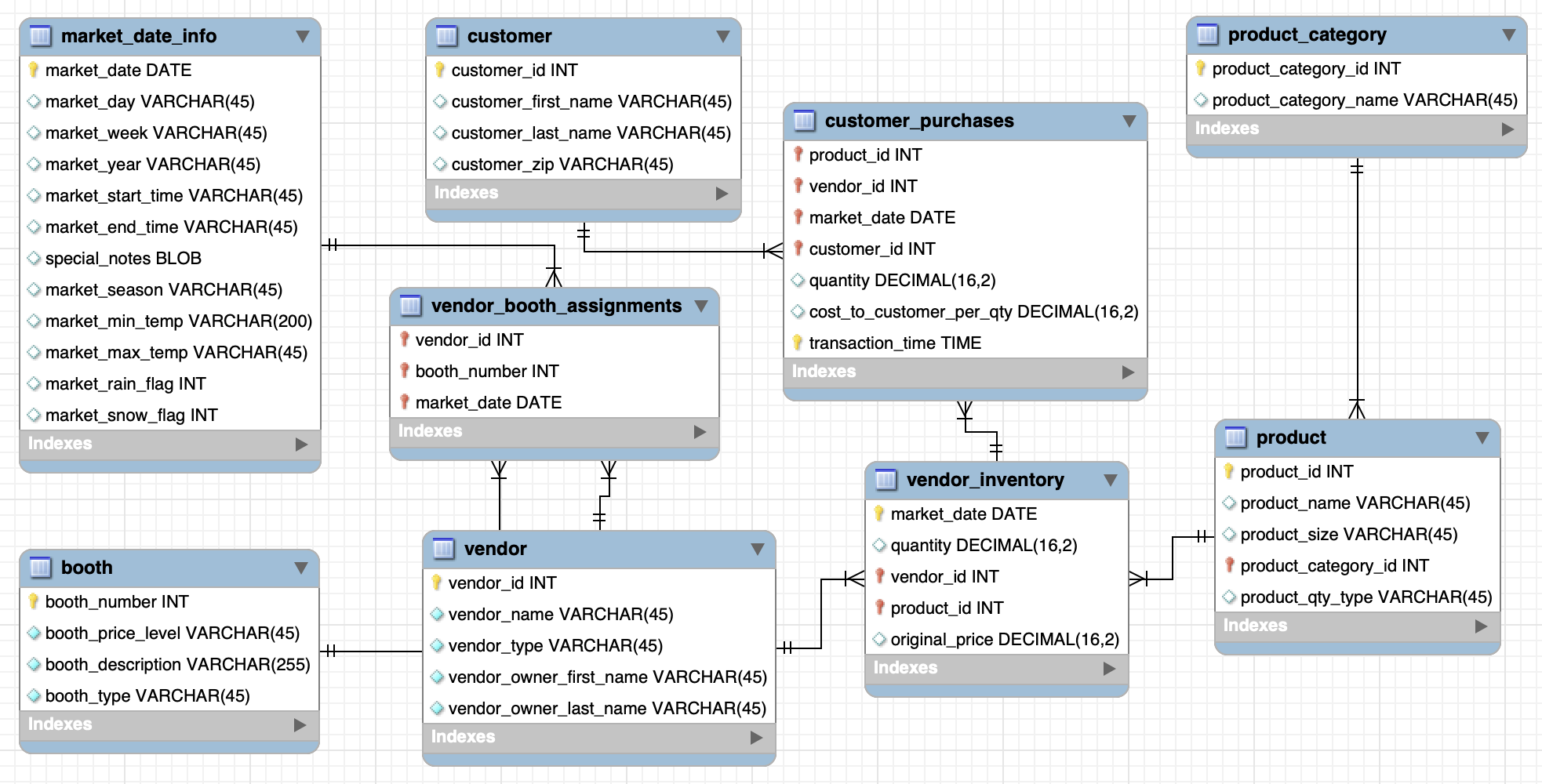
Window Functions

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement:

You are a Data Analyst at Amazon Fresh. You have been tasked to study the Farmer’s Market.

Dataset: Farmer’s Market database



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# What does Group By do to the output? - Groups / Collapses rows.

* All the functions covered, like ROUND(), **return one value in each row** of the results dataset.
* When **GROUP BY is used, the functions operate on multiple values in an aggregated group of records**, summarizing across multiple rows in the underlying dataset, like AVG(), but each value returned is associated with a single row in the results.
* The output is always grouped.

**What if we want to operate on multiple rows but don’t want the records to be grouped in the output?**

Let’s look at an example use case:

# Question: Get the price of the most expensive item per vendor?

This is pretty simple:

1. Group records by ***vendor\_id*** in the vendor\_inventory table.
2. Return the MAX ***original\_price*** value

SELECT

vendor\_id,

MAX(original\_price) AS highest\_price

FROM farmers\_market.vendor\_inventory

GROUP BY vendor\_id

Now, here you’ll get the most expensive item per vendor.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What if the question changes to:

# New Question: Rank the products in each vendor’s inventory. Expensive products get a lower rank.

In this question:

* You don’t want to group the rows by vendor here as we want to rank all the products on each date.
* So, we need a technique to maintain the row-level information you would otherwise lose by using Group By.

# 

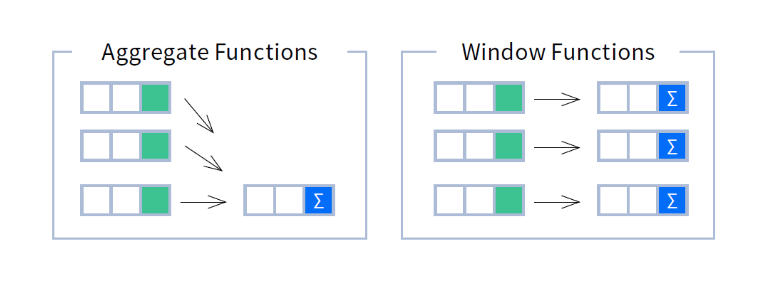
# **Window Functions**

Window function gives the ability to put the values from one row of data into context compared to a group of rows or partitions.

**We can answer questions like:**

* Where would this row land in the results if the dataset were sorted? - Rank
* How does a value in this row compare to the prior row? - Accessing the preceding / following row.
* How does a current row value compare to the group or partition(in window function’s context) average value?

So, window functions **return group aggregate calculations alongside individual row-level** information for items in that group or partition.



So, in our question,

We need a function that allows us to **rank rows by a value** - in our case, **ranking products per vendor by price** - called **ROW\_NUMBER**().

**Syntax** : ROW\_NUMBER() OVER (<partition\_definition> <order\_definition>)

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

**ROW\_NUMBER() OVER (PARTITION BY vendor\_id ORDER BY original\_price** **DESC) AS price\_rank**

FROM farmers\_market.vendor\_inventory

**Syntax Breakdown:**

* I would interpret the **ROW\_NUMBER**() line as “**number the inventory rows per vendor, sorted by original price, in descending order.**”
* **OVER() -** tells the DBMS to apply the function over a window of rows.
* The part inside the parentheses says how to apply the ROW\_NUMBER() function.
* We’re going to **PARTITION** **BY** ***vendor\_id*** (you can think of this like a GROUP BY without actually combining the rows, so we’re telling it how to split the rows into groups without aggregating).
* The **ORDER BY** indicates how to sort each partition of rows. So, we’ll sort the rows by price, high to low, within each vendor\_id partition, and then number each row as per their price.
* The highest-priced item per vendor will be first assigned **row number 1**.

**Output Explanation:**

* For each vendor, the products are **sorted by original\_price**, **high to low, and the row numbering column is called price\_rank**.
* The row numbering starts when you get to the next ***vendor\_id***, so the most expensive item per vendor has a ***price\_rank*** of 1.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Follow-up question: Get me all the products per vendor that have a price rank of 1.

SELECT \* FROM

(

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

ROW\_NUMBER() OVER (PARTITION BY vendor\_id ORDER BY original\_price DESC) AS price\_rank

FROM farmers\_market.vendor\_inventory ORDER BY vendor\_id) x

WHERE x.price\_rank = 1;

**Query Breakdown:**

* You’ll notice that the preceding query has a different structure than the queries we have written so far.
* The concept of subqueries comes again. There is one query embedded inside the other! This is also called “**querying from a derived table**,”.
* We’re treating the results of the “inner” SELECT statement like a table, here given the table alias x, selecting all columns from it, and filtering to only the rows with a particular ROW\_NUMBER.
* Our ROW\_NUMBER column is aliased ***price\_rank***, and we’re filtering to ***price\_rank = 1***, because we numbered the rows by ***original\_price*** in descending order, so the most expensive item will have the lowest row number.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Why not put the WHERE clause in the main query itself? - Execution Order**

* If we didn’t use a subquery and had attempted to filter based on the values in the ***price\_rank*** field by adding a WHERE clause to the first query with the **ROW\_ NUMBER** function, **we would get an error.**
* The ***price\_rank*** **value is unknown at the time the WHERE clause conditions are evaluated per row** because the window functions have not yet had a chance to check the entire dataset to determine the ranking.
* If we tried to put the **ROW\_NUMBER** function in the **WHERE** clause, instead of referencing the ***price\_rank*** alias, we would get a different error, but for the same reason.

**All query elements are processed in a very strict order:**

* **FROM** - The database gets the data from tables in FROM clause and if necessary, performs the JOINs.
* **JOIN** - Depending on the type of JOIN used in the query and conditions specified for joining the tables in the **ON** clause, the database engine matches rows from the virtual table created in the FROM clause.
* **WHERE** - After the JOIN operation, the data is filtered based on the conditions specified in the WHERE clause. Rows that do not meet the criteria are excluded.
* **GROUP** **BY** - If the query includes a GROUP BY clause, the rows are grouped based on the specified columns.
* **Aggregate** **Functions** - The aggregate functions are applied to the groups created in the GROUP BY clause.
* **HAVING** - The HAVING clause filters the groups of rows based on the specified conditions
* **Window Functions**
* **SELECT** - After grouping and filtering is done, the SELECT statement determines which columns to include in the final result set.
* **DISTINCT** - The DISTINCT keyword is applied within the SELECT clause to ensure that only unique values are returned for the specified columns.
* **UNION**/**INTERSECT**/**MINUS** - After generating the result sets from individual SELECT queries, the database applies Set Operations.
* **ORDER BY** - It allows you to sort the result set based on one or more columns, either in ascending or descending order.
* **OFFSET** - The specified number of rows are skipped from the beginning of the result set.
* **LIMIT** - After skipping the rows, the LIMIT clause is applied to restrict the number of rows returned.

**Note:**

You can also use ROW\_NUMBER without a PARTITION BY clause to number every record across the whole result (instead of numbering per partition).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Transition:**

The problem with the output in ROW\_NUMBER() is that even for the same values, we are getting different numbers or ranks but what if you want the same rank to be assigned to the same values.

# **RANK and DENSE\_RANK**

To return all products with the highest price per vendor when there is more than one with the same price, use the **RANK function**.

**The RANK function numbers the results just like ROW\_NUMBER does, but gives rows with the same value the same ranking**.

Replace ROW\_NUMBER with RANK in the original query.

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

RANK() OVER (PARTITION BY vendor\_id ORDER BY original\_price DESC) AS price\_rank

FROM farmers\_market.vendor\_inventory

ORDER BY

vendor\_id, original\_price DESC;

**Output Breakdown:**

* Notice that the ranking for **vendor\_id 1** goes from **1 to 2 to 4**, **skipping 3.** That’s because there’s a tie for second place, so there’s no third place.
* If you don’t want to skip numbers like this in your ranking when there is a tie use the **DENSE\_RANK** function..
* And if you don’t want the ties at all, use the ROW\_NUMBER function.

The ROW\_NUMBER() and RANK() functions can help answer a question that asks something like

* “What are the top 10 items sold at the farmer’s market, by price?” (by filtering the results to rows numbered less than or equal to 10).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Question: Divide the rows into a number of buckets based on a NTILE function

The SQL NTILE() is a window function that distributes rows of an ordered partition into a specified number of approximately equal groups, or buckets.

It assigns each group a bucket number starting from one. For each row in a group, the NTILE() function assigns a bucket number representing the group to which the row belongs.

**The dynamic solution is to use the NTILE function.**

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

NTILE(10) OVER (ORDER BY original\_price DESC) AS price\_ntile

FROM farmers\_market.vendor\_inventory

ORDER BY original\_price DESC;

**Important points about NTILE**

* If the number of rows in the results set can be divided evenly, the results will be broken up into **n equally sized groups,** **labeled 1 to n**.
* If they can’t be divided up evenly, some groups will end up with one more row than others.
* Note that the **NTILE is only using the count of rows to split the groups**, and is not using a field value to determine where to make the splits.
* Therefore, **it’s possible that two rows with the same value specified in the ORDER BY clause** will end up in two different **NTILE groups.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Transition:**

But what if you were asked to return the “**top** **tenth**” of the inventory when sorted by price?

* You could start by running a query that used the **COUNT**() function,
* dividing the number returned by 10,
* then writing another query that numbers the rows, and
* filtering to those with a row number less than or equal to the number you just determined.

**But that isn’t a dynamic solution; you’d have to modify it as the number of rows in the database changed.**

# **Aggregated Window Functions**

## Question: As a farmer, you want to figure out which of your products were above the average price on each market date?

We can use the **AVG**() function as a window function, partitioned by ***market\_date***, and compare each product’s price to that value.

First, let’s try using AVG() as a window function.

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

AVG(original\_price) OVER (PARTITION BY market\_date) AS average\_cost\_product\_by\_market\_date

FROM farmers\_market.vendor\_inventory;

**Query Breakdown**

* The AVG() function in this query is structured as a window function, meaning it has “OVER (PARTITION BY \_\_ ORDER BY \_\_)” syntax, so instead of returning a single row per group with the average for that group, like you would get with GROUP BY, this function displays the average for each partition in every row within the partition.
* When you get to a new ***market\_date*** value in the results dataset, the ***average\_cost\_product\_by\_market\_date*** value changes.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Follow-up Question: Extract the farmer’s products with prices above the market date’s average product cost for vendor id 8?

* Using a **subquery**, we can filter the results to a single vendor, with **vendor\_id** **8**, and
* only **display products that have prices above the market date’s average product** cost.
* Here we will also format the ***average\_cost\_product\_by\_market\_*** date to two digits after the decimal point using the **ROUND**() function:

SELECT \* FROM

(

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

ROUND(AVG(original\_price) OVER (PARTITION BY market\_date ORDER BY market\_date), 2) AS average\_cost\_product\_by\_market\_date

FROM farmers\_market.vendor\_inventory )x

WHERE x.vendor\_id = 8

AND x.original\_price > x.average\_cost\_product\_by\_market\_date

ORDER BY x.market\_date, x.original\_price DESC;

* Another use of an aggregate window function is to count how many items are in each partition.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Question: Count how many different products each vendor brought to market on each date and display the count for each row.

The answer to this question would help you identify that the row you’re looking at represents just one of the products in a counted set:

SELECT

vendor\_id,

market\_date,

product\_id,

original\_price,

COUNT(product\_id) OVER (PARTITION BY market\_date, vendor\_id)

vendor\_product\_count\_per\_market\_date

FROM farmers\_market.vendor\_inventory

ORDER BY vendor\_id, market\_date, original\_price DESC;

**Output:**

* You can see that even if I’m only looking at one row for vendor 7 on July 6, 2019, I would know that it is one of **4 products** that the vendor had in their inventory on that market date.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Concept of Window Frames** ([ref resource](https://dev.mysql.com/blog-archive/mysql-8-0-2-introducing-window-functions/))

Let’s look at the [**sales**](https://drive.google.com/file/d/1ZGMaJBZ3rs63UJ0ZMjbefZTRYiOedHdz/view?usp=share_link) table.

Tell me what will the output of each query look like:

Select \* FROM sales;

1. SELECT SUM(sale) FROM sales;
2. SELECT SUM(sale) OVER() AS sum

FROM sales;

1. SELECT employee,

date,

SUM(sale) OVER(PARTITION BY employee) AS sum

FROM sales;

1. SELECT employee,

sale,

date,

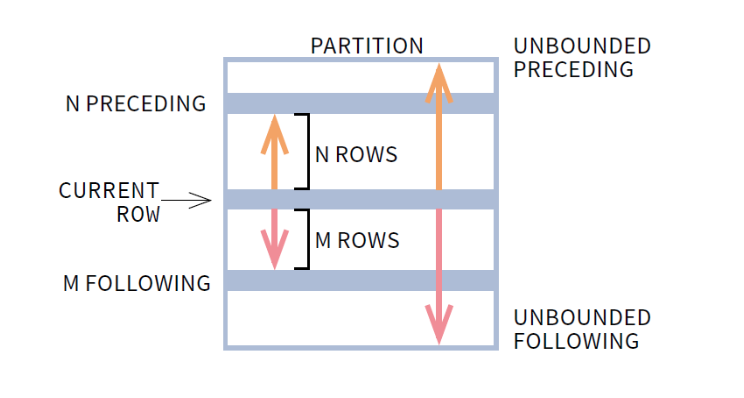
SUM(sale) OVER (ORDER BY date) AS cum\_sales

FROM sales;

The ORDER BY clause in the window gives us a cumulative sum. Now, why is that? - **Window Frames**

**Query Breakdown:**

This query internally is represented as: (ORDER BY date RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)



**In the output,**

* **UNBOUNDED PRECEDING**: It indicates that the window starts at the first row of the partition, UNBOUNDED PRECEDING is the default.
* **CURRENT ROW** indicates the window begins or ends at the current row.
* **UNBOUNDED FOLLOWING** indicates that the window ends at the last row of the partition.

**Query:**

SELECT

employee, sale, date,

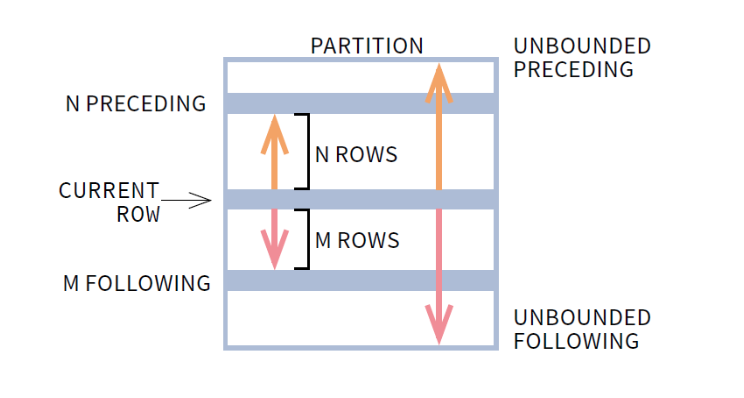
SUM(sale) OVER (ORDER BY date ROWS UNBOUNDED PRECEDING) AS cum\_sales

FROM sales;

Both ROWS and RANGE clauses in SQL limit the rows considered by the window function within a partition.

The ROWS clause does that quite literally. It **specifies a fixed number of rows** that precede or follow the current row regardless of their value. These rows are used in the window function.

On the other hand, the RANGE clause logically limits the rows. That means it considers the rows **based on their value compared to the current row**.



## \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Question: Calculate the moving average on a window frame of 1 preceding and 1 following.

SELECT MONTH(date), SUM(sale),

AVG(SUM(sale)) OVER (ORDER BY MONTH(date)

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS sliding\_avg

FROM sales GROUP BY MONTH(date);

**Code for BigQuery:**

SELECT EXTRACT(MONTH FROM date),

SUM(sale), AVG(SUM(sale)) OVER (ORDER BY MIN(EXTRACT(MONTH FROM date))

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS sliding\_avg

FROM farmers\_market.sales

GROUP BY 1;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

How will you calculate 5 - DAY moving average for Stock prices?

SELECT

date, sale,

AVG(sale) OVER (ORDER BY date RANGE 4 PRECEDING) AS sliding\_avg

FROM sales;

**Code for BigQuery:**

SELECT

date, sale,

AVG(sale) OVER (ORDER BY unix\_date(date) RANGE 4 PRECEDING) AS sliding\_avg

FROM farmers\_market.sales;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**LAG and LEAD**

Now we can see how SQL can calculate the changes over time.

## Question: Using the **vendor\_booth\_assignments** table in the Farmer’s Market database, display each vendor’s booth assignment for each ***market\_date*** alongside their previous booth assignments.

For this, we are going to use the LAG() function.

* LAG retrieves data from a row that is a selected number of rows back in the dataset. You can set the number of rows (offset) to any integer value **x** to count x rows backward, following the sort order specified in the ORDER BY section of the window function.
* Partition by vendor\_id.

**Syntax of LEAD / LAG:**

LEAD(expr, N, default)

OVER (Window\_specification | Window\_name)

**Parameters used:**

* **expr**: It can be a column or any built-in function.
* **N**: It is a positive value which determines the number of rows preceding/succeeding the current row. If it is omitted in the query then its default value is 1.
* **default**: It is the default value return by function in-case no row precedes/succeeds the current row by N rows. If it is missing then it is by default NULL.
* **OVER()**: It defines how rows are partitioned into groups. If OVER() is empty then the function computes the result using all rows.
* **Window\_specification**: It consists of a query partition clause which determines how the query rows are partitioned and ordered.
* **Window\_name**: If window is specified elsewhere in the query then it is referenced using this Window\_name.

SELECT

market\_date,

vendor\_id,

booth\_number,

LAG(booth\_number,1) OVER (PARTITION BY vendor\_id ORDER BY market\_date, vendor\_id) AS previous\_booth\_number

FROM farmers\_market.vendor\_booth\_assignments

ORDER BY market\_date, vendor\_id, booth\_number;

**The LAG() function is used to get a value from a row that precedes the current row.**

**Output:**

* In this case, for each ***vendor\_id*** for each ***market\_date***, we’re pulling the ***booth\_number*** the vendor had **1 market date in the past.**
* The values are all NULL for the first market date because there is no prior market date to pull values from.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Using this as a Subquery -**

Question: The Market manager may want to filter these query results to a specific market date to determine which vendors are new or changing booths that day, so we can contact them and ensure setup goes smoothly. Check it for date: 2019-04-10

**Breakdown:**

* We will create this report by wrapping the query with the LAG function in another query,
* we can use this inner query results to filter the results to a ***market\_date*** and vendors whose current ***booth\_number*** is different from their ***previous\_booth\_number***:

SELECT \* FROM

(

SELECT

market\_date,

vendor\_id,

booth\_number,

LAG(booth\_number,1) OVER (PARTITION BY vendor\_id ORDER BY market\_date, vendor\_id) AS previous\_booth\_number

FROM farmers\_market.vendor\_booth\_assignments

ORDER BY market\_date, vendor\_id, booth\_number

) x

WHERE x.market\_date = '2019-04-10'

AND (x.booth\_number <> x.previous\_booth\_number OR x.previous\_booth\_number IS NULL);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Question: Let’s say you want to find out if the total sales on each market date are higher or lower than they were on the previous market date.

**Breakdown**

(Crux - they will have to use both GroupBy(total sales for the day) and Window function(LAG)):

* We are going to use the customer\_purchases table from the Farmer’s Market database, and also add a GROUP BY function, which the previous examples did not include.
* The window functions are calculated after the grouping and aggregation occur.
* First, we need to get the total sales per market date, using a GROUP BY and regular aggregate SUM.

SELECT

market\_date,

SUM(quantity \* cost\_to\_customer\_per\_qty) AS market\_date\_total\_sales

FROM farmers\_market.customer\_purchases

GROUP BY market\_date;

Then, we can add the **LAG**() window function to output the previous ***market\_date’s*** calculated sum on each row.   
  
We ORDER BY market\_date in the window function to ensure it’s the previous market date we’re comparing to and not another date.

SELECT

market\_date,

SUM(quantity \* cost\_to\_customer\_per\_qty) AS market\_date\_total\_sales,

LAG(SUM(quantity \* cost\_to\_customer\_per\_qty), 1) OVER (ORDER BY

market\_date) AS previous\_market\_date\_total\_sales

FROM farmers\_market.customer\_purchases

GROUP BY market\_date;

**LEAD** works the same way as LAG, but it gets the value from the next row instead of the previous row (assuming the offset integer is 1). You can set the offset integer to any value x to count x rows forward, following the sort order specified in the ORDER BY section of the window function.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Nth\_Value() / First\_Value() - Window Frames**

The **NTH\_VALUE**() is a window function that allows you to retrieve the value of an expression or column from the Nth row within a window frame when the rows are ordered according to a specific column.

**Syntax of the NTH\_VALUE() function:**

NTH\_VALUE(column\_name, n) OVER (

[PARTITION BY partition\_expression, ... ]

ORDER BY sort\_expression [ASC | DESC]

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS alias

* If that Nth row does not exist, the function returns **NULL**.
* N must be a positive integer e.g., 1, 2, and 3.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The **FIRST\_VALUE**() is a window function that is used to return the value from the first row within a specified window frame when the rows are ordered according to a specific column.

**Syntax of the FIRST\_VALUE() function:**

FIRST\_VALUE(column\_name) OVER (

[PARTITION BY partition\_expression, ... ]

ORDER BY sort\_expression [ASC | DESC]

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS alias

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Let’s look at the [**employee**](https://drive.google.com/file/d/18vIoUPVTF7Z16JBSOmO2doMpazIFidca/view?usp=share_link) table.

## Question: Find the employee with the second highest salary in each department.

SELECT

employee\_name,

department,

salary,

NTH\_VALUE(employee\_name, 2) OVER (

PARTITION BY department

ORDER BY salary DESC

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) second\_highest\_salary

FROM

farmers\_market.employee;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Use subqueries to find the 2nd highest and 3rd highest item/value.

**Query 1:**

SELECT vendor\_id,

MAX(original\_price) AS salary

FROM farmers\_market.vendor\_inventory

WHERE original\_price <= (SELECT MAX(original\_price)

FROM farmers\_market.vendor\_inventory);

**Query 2:**

SELECT original\_price

FROM

(SELECT original\_price

FROM farmers\_market.vendor\_inventory

ORDER BY original\_price

LIMIT 3) AS Comp

ORDER BY original\_price

LIMIT 1;