Window Functions

In the last lecture, we discussed a few unique features of HiveQL, in this lecture, we will discuss about window functions. Window functions are also used in MySQL. Let's understand window functions with the help of a case study.

Problem Statement

You are a Data Analyst at the Food Corporation of India (FCI). You have been tasked to study the Farmer's market - *Mandis*.

Dataset: Farmer's Market Database

Create a Table and data using this

CREATE TABLE vendor_inventory (vendor_id INT, market_date DATE, product_id INT, original_price DECIMAL(10, 2), product_name VARCHAR(100));

INSERT INTO vendor_inventory (vendor_id, market_date, product_id, original_price, product_name) VALUES (1, '2024-10-15', 101, 150.00, 'Tomatoes'), (1, '2024-10-15', 102, 250.00, 'Potatoes'), (1, '2024-10-15', 103, 200.00, 'Carrots'), (2, '2024-10-15', 201, 100.00, 'Spinach'), (2, '2024-10-15', 202, 50.00, 'Cabbage'), (2, '2024-10-15', 203, 120.00, 'Onions'), (3, '2024-10-15', 301, 180.00, 'Apples'), (3, '2024-10-15', 302, 220.00, 'Oranges'), (3, '2024-10-15', 303, 210.00, 'Bananas'), (4, '2024-10-15', 401, 95.00, 'Peppers'), (4, '2024-10-15', 402, 180.00, 'Broccoli'), (4, '2024-10-15', 403, 75.00, 'Lettuce');

Use case 1

Get the price of the most expensive item per vendor?

This is pretty simple:

- Group records by vendor_id in the vendor_inventory table.
- Return the MAX original_price value

```
SELECT
        vendor_id,
        MAX(original_price) AS highest_price
FROM vendor_inventory
GROUP BY vendor_id
```

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

Use case 2

Rank the products in each vendor's inventory. Expensive products get a lower rank

In this usecase:

- 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. This technique is enabled by something called window functions.

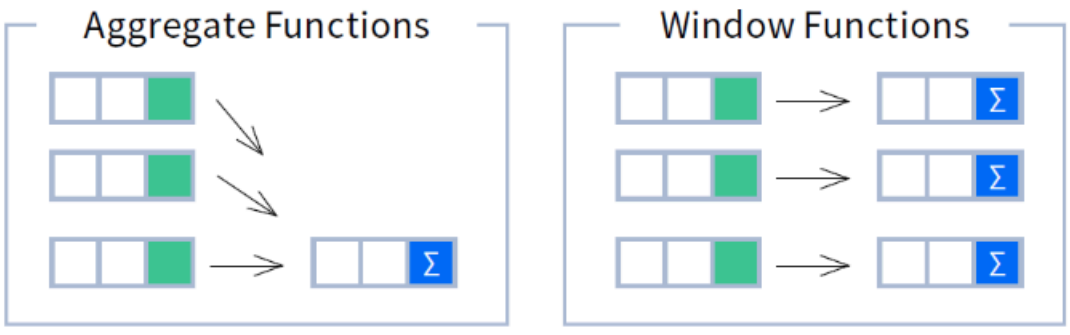
Definition of Window Function

The 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 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 Use case 2,

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.

Use Case 3

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,
- **WHERE** - the data are filtered with conditions specified in the WHERE clause,
- **GROUP BY** - the data are grouped by conditions specified in the WHERE clause,
- **Aggregate functions** - the aggregate functions are applied to the groups created in the GROUP BY phase,
- **HAVING** - the groups are filtered with the given condition,
- **Window functions,**
- **SELECT** - the database selects the given columns,
- **DISTINCT** - repeated values are removed,
- **UNION**/**INTERSECT**/**EXCEPT** - the database applies set operations,
- **ORDER BY** - the results are sorted,
- **OFFSET** - the first rows are skipped,
- **LIMIT/FETCH/TOP** - only the first rows are selected

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).

Use Case 4

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 same rank to be assigned to same values.

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).

Use Case 5

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
```

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
```

Use Case 6

Count how many different products each vendor brought to market on each date and display that count on 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 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.

Window Frames

Let's consider a dataset like this

Sales Data

	sale_id	product_id	sale_date	amount
1	1	101	2023-01-01	200
2	2	101	2023-01-02	300
3	3	102	2023-01-03	150
4	4	102	2023-01-04	350
5	5	103	2023-01-05	100
6	6	103	2023-01-06	250
7	7	101	2023-01-07	220
8	8	101	2023-01-08	320
9	9	102	2023-01-09	180
10	10	102	2023-01-10	400

We will use this to explain window functions such as `LEAD`, `LAG`, `FIRST_VALUE`, and `NTH_VALUE`.

This dataset contains:

- **sale_id**: Unique identifier for each sale.
- **product_id**: The ID of the product being sold.
- **sale_date**: The date of the sale.
- **amount**: The sales amount on that day.

Window frames allow you to define a subset of rows over which the window function operates, providing more granular control. While window functions like `LEAD`, `LAG`, `FIRST_VALUE`, and `NTH_VALUE` operate over a set of rows defined by a **partition** and **order**, **window frames** further narrow down the range of rows considered for each individual calculation.

A **window frame** defines a moving range of rows relative to the current row within the ordered set. This range specifies how many preceding or following rows are included in the calculation. In know this is not making much sense now, but lets try to understand this with examples.

Examples of Window Frames

1. LEAD() Function

What is LEAD?

The **LEAD** function allows you to access data from the following rows. This is helpful when you want to compare the current row with future rows. Imagine looking at today's sales and asking: "What was the sales amount the next day?"

How it works

The syntax for **LEAD** looks like this.

```
LEAD(column_name, offset, default_value)

    OVER (PARTITION BY column_name ORDER BY column_name)
```

- `column_name`: The column from which you want to fetch data.
- `offset`: How many rows ahead you want to look. For example, `1` means "the next row," `2` means "two rows ahead."
- `default_value`: This is optional, and it defines what value should be returned if the function goes beyond the available rows.

Example:

You want to compare today's sales to the next day’s sales. Let's look at the following dataset.

sale_id	product_id	sale_date	amount

1	101	2023-01-01	200
2	101	2023-01-02	300
3	102	2023-01-03	150

Now, if we apply **LEAD** to look at the next day's sale amount for each product, it would work like this:

```
SELECT sale_id, product_id, sale_date, amount,

       LEAD(amount, 1) OVER (PARTITION BY product_id ORDER BY
sale_date) AS next_day_sales

FROM sales_data;
```

Output:

sale_id	product_id	sale_date	amount	next_day_sales
1	101	2023-01-01	200	300
2	101	2023-01-02	300	(NULL)

3	102	2023-01-03	150	(NULL)

Explanation:

- For the first row (2023-01-01), the next day's sale amount is `300`.
- For the second row (2023-01-02), there is no next day sale, so it returns `NULL`.

---

2. LAG() Function

What is LAG?

The **LAG** function is the opposite of LEAD. It retrieves data from previous rows. It's handy when you want to see how today's value compares with yesterday's value.

How it works

The syntax is very similar to LEAD:

```
LAG(column_name, offset, default_value)

    OVER (PARTITION BY column_name ORDER BY column_name)
```

- `column_name`: The column from which you want to fetch data.
- `offset`: How many rows back you want to look (e.g., `1` for the previous row).
- `default_value`: The value to return if no previous row exists.

Example:

Let's take the same data and look at how **LAG** works when we want to compare today's sale with the previous day's sale:

```
SELECT sale_id, product_id, sale_date, amount,

       LAG(amount, 1) OVER (PARTITION BY product_id ORDER BY
sale_date) AS prev_day_sales

FROM sales_data;
```

Output:

sale_id	product_id	sale_date	amount	prev_day_sales
1	101	2023-01-01	200	(NULL)
2	101	2023-01-02	300	200
3	102	2023-01-03	150	(NULL)

Explanation:

- For the first row (2023-01-01), there is no previous day, so it returns `NULL`.
- For the second row (2023-01-02), the previous day's sale amount is `200`.

---

3. FIRST_VALUE() Function

What is FIRST_VALUE?

The **FIRST_VALUE** function returns the first value in a defined window (ordered by some column). This is great when you want to compare the current value to the first value in the dataset or the first value in a group.

How it works

Here's the syntax:

```
FIRST_VALUE(column_name)
```

```
    OVER (PARTITION BY column_name ORDER BY column_name)
```

Example:

Let's say you want to find the very first sale amount for each product and compare it with today's sales.

```
SELECT sale_id, product_id, sale_date, amount,

       FIRST_VALUE(amount) OVER (PARTITION BY product_id ORDER BY
sale_date) AS first_sale_amount

FROM sales_data;
```

Output:

sale_id	product_id	sale_date	amount	first_sale_amount
1	101	2023-01-01	200	200
2	101	2023-01-02	300	200
3	102	2023-01-03	150	150

Explanation:

- For each row, it shows the first sale amount for that product. For product `101`, the first sale amount is `200`.

---

4. NTH_VALUE() Function

What is NTH_VALUE?

The **NTH_VALUE** function allows you to fetch the Nth row in an ordered set. For instance, if you want to see the 2nd or 3rd sale amount in the time series, this function is perfect.

How it works

Here's the syntax:

```
NTH_VALUE(column_name, N)

    OVER (PARTITION BY column_name ORDER BY column_name)
```

Example:

If you want to find the 2nd sale amount for each product:

```
SELECT sale_id, product_id, sale_date, amount,

       NTH_VALUE(amount, 2) OVER (PARTITION BY product_id ORDER BY
sale_date) AS second_sale_amount

FROM sales_data;
```

Output:

sale_id	product_id	sale_date	amount	second_sale_amount
1	101	2023-01-01	200	300

2	101	2023-01-02	300	300
3	102	2023-01-03	150	(NULL)

Explanation:

- For product `101`, the 2nd sale amount is `300`, so every row for this product shows `300`.
- For product `102`, there is no second sale, so it returns `NULL`.