Window Functions contd.

---

1. Problem Statement
2. Lead(), Lag()
3. Cumulative Sum
4. Window Frames
5. Moving Average
6. ROWS vs. RANGE
7. First_value(), Last_value()
8. Impact of the analysis

Please note that any topics that are not covered in today's lecture will be covered in the next lecture.

Problem Statement: GlobalTech Solutions

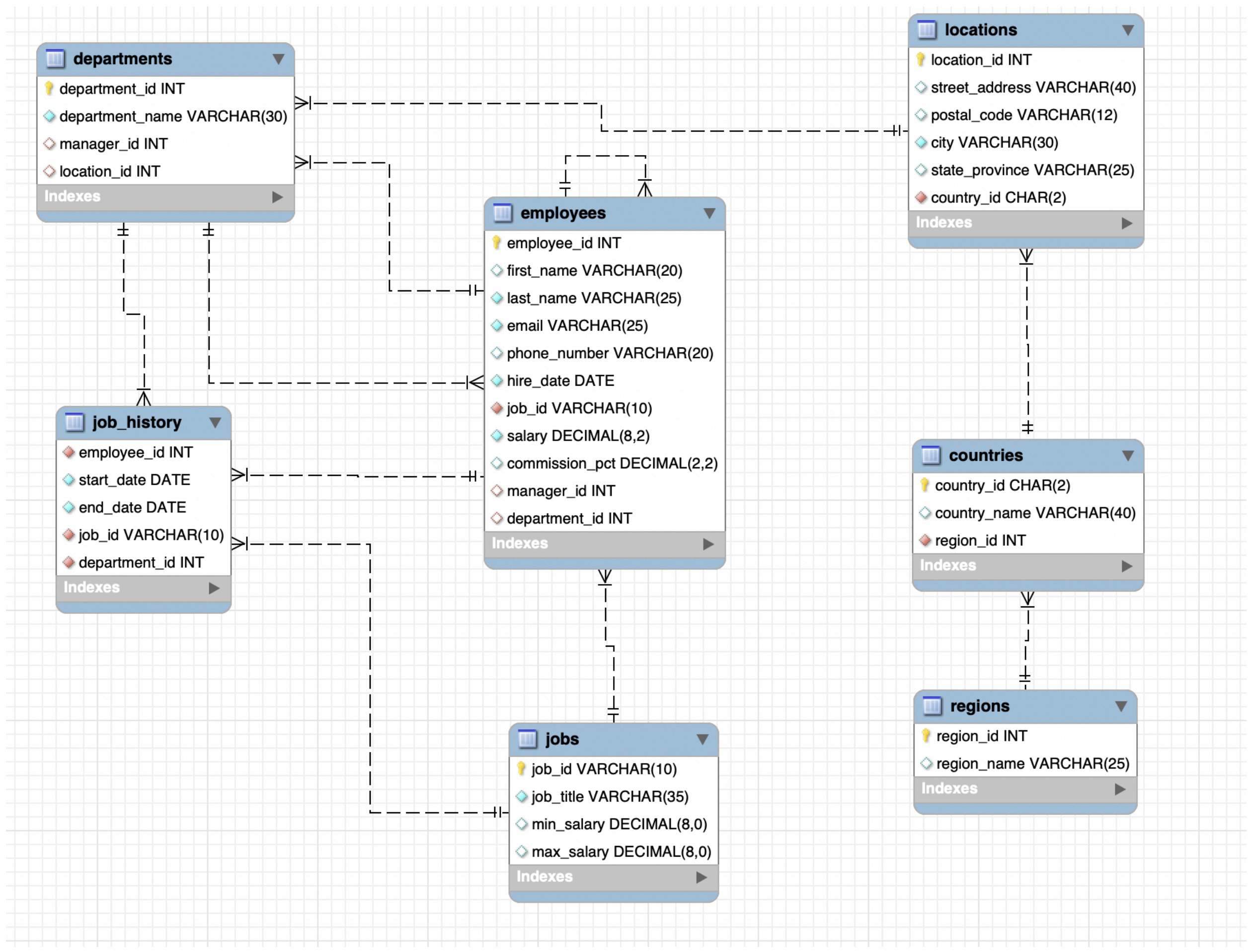
GlobalTech Solutions, a leading technology firm renowned for its innovative solutions and cutting-edge technology, boasts a diverse and talented workforce.

The objective of this project is to conduct an in-depth analysis of employee compensation within the company, using SQL.

This analysis will focus on comparing salaries across various dimensions, including average salaries, salary distributions within departments, and identifying top earners.

The derived insights will help in understanding salary structures, pinpointing any underlying disparities, and assessing salary growth based on job roles within the company.

Dataset: link



Disclaimer: In real-world scenarios, learners might not always provide the exact responses anticipated. However, the instructor should guide the conversation strategically to reach the desired learning outcome and introduce the relevant SQL concepts.

The following is a dialogue template for the instructor's reference. This would help motivate/initiate discussion around the topics to be covered in the session :

—— Dialogue Template starts ——

Instructor: Welcome back, everyone! In our previous session, we explored the power of window functions in SQL. We harnessed functions like

ROW_NUMBER(), RANK(), DENSE_RANK(), and NTILE() to perform calculations and comparisons within result sets. We leveraged these functions to calculate averages, identify top performers, and categorize data into buckets based on specific criteria.

Imagine you want to provide a report for each employee that includes not just their details and current salary, but also something more – the salary of the next highest earner in the company, along with the difference between their salaries.

Learner: So, we need to find out for each employee, who earns more than them in the company, and by how much?

Instructor: Exactly! There's a special window function in SQL called LEAD() that allows us to peek ahead within the data, just like looking at the next person in line.

Learner: Oh so, maybe we can use this LEAD() function to look at the salary of the employee right after the current employee in the list, but do we need to consider sorting everyone by salary in descending order (highest to lowest) right?

Instructor: Precisely! By using LEAD() on the salary data, ordered by salary in descending order, we can compare each employee's salary with the one positioned just above them in the salary hierarchy. We can then calculate the difference between their salaries to reveal the salary gap.

Let’s see how we can achieve this.

—– Dialogue Template ends —–

Formulating questions to be explored based on the data provided:

1. Display detailed information on employees earning above the company-wide average salary.
2. Show details of employees whose salaries exceed the average salary of their respective departments.
3. Identify the top 5 highest-earning employees in the company.
4. Fetch details of employees with the highest (or nth highest) salaries in each department.
5. How can we categorize all employees into N different groups (quartiles) based on their salaries?
6. For each employee, compare their salary to the next highest salary in their department, and add a new column showing the difference.
7. Display the Cumulative / Running sum of salaries in each department, ordered by the employees' joining dates.
8. Display the Moving sum/average of salaries in each department, with a window size of N, ordered by the employees' joining dates.
9. Compare each employee's salary with the salaries of the first and last hired employees in their department.

Note: During the previous session, we addressed some of these questions, and the remaining ones will be covered today.

LEAD() and LAG()

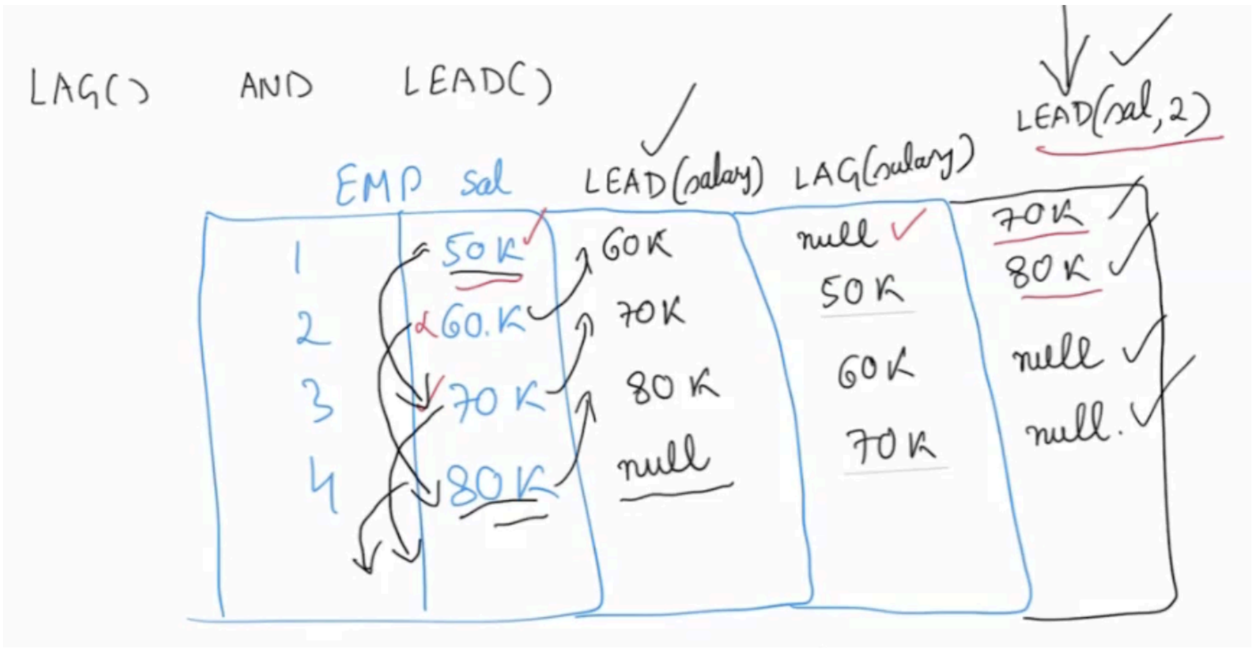
#Q: For each employee, provide their details and current salary. Additionally, identify the salary of the next highest-earning employee within the company. Present the difference in their salaries in a new column labelled `sal_diff`.

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

- LEAD retrieves data from a row that is a selected number of rows ahead in the dataset. You can set the number of rows (offset) to any integer value **x** to count x rows forward, following the sort order specified in the ORDER BY section of the window function.

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

Illustration ->



LAG() AND LEAD()

EMP	sal	LEAD(salary)	LAG(salary)	LEAD(sal,2)
1	50K	60K	null	70K
2	60.K	70K	50K	80K
3	70K	80K	60K	null
4	80K	null	70K	null.

Important Points

1. When we apply **LEAD**(), the last row of the newly created column will have **NULL**.
    a. Because there's nothing **post 80K** to compare it with, that's why we get NULL right next to it.
2. When we apply **LAG**(), the first row of the newly created column will have **NULL**
    a. Because there's nothing **prior to 50K** to compare it with, that's why we get NULL right next to it.
3. **LEAD**() & **LAG**() also give you the option of skipping values and then comparing (as shown in the last column).

Syntax of LEAD / LAG:

```
LEAD/LAG(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 that 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 returned by the 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 that determines how the query rows are partitioned and ordered.
- **Window_name**: If the window is specified elsewhere in the query then it is referenced using this Window_name.

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 LEAD(salary) OVER(ORDER BY salary) AS next_salary,
 LEAD(salary) OVER(ORDER BY salary) - salary AS sal_diff,
FROM hr.employees
ORDER BY salary, next_salary;
```

Solution Query for the above illustration:

```
SELECT
 employee_id,
 department_id,
 salary,
 LEAD(salary) OVER(ORDER BY salary) AS next_salary,
 LAG(salary) OVER(ORDER BY salary) AS prev_salary,
 LEAD(salary, 2) OVER(ORDER BY salary) AS next_next_salary
FROM hr.employees
ORDER BY salary, next_salary;
```

Quiz - 1

Q. Lag and Lead can work even without the order by clause within the window function statement.

a) True
b) False - correct

Quiz - 2

Q. Assume we have 5 rows of dates and sales. How many NULL values will be in the sales_lag column based on this query?
    lag(sales,7) over (order by date) as sales_lag

a) 2
b) 5 - correct
c) 1
d) No Null values

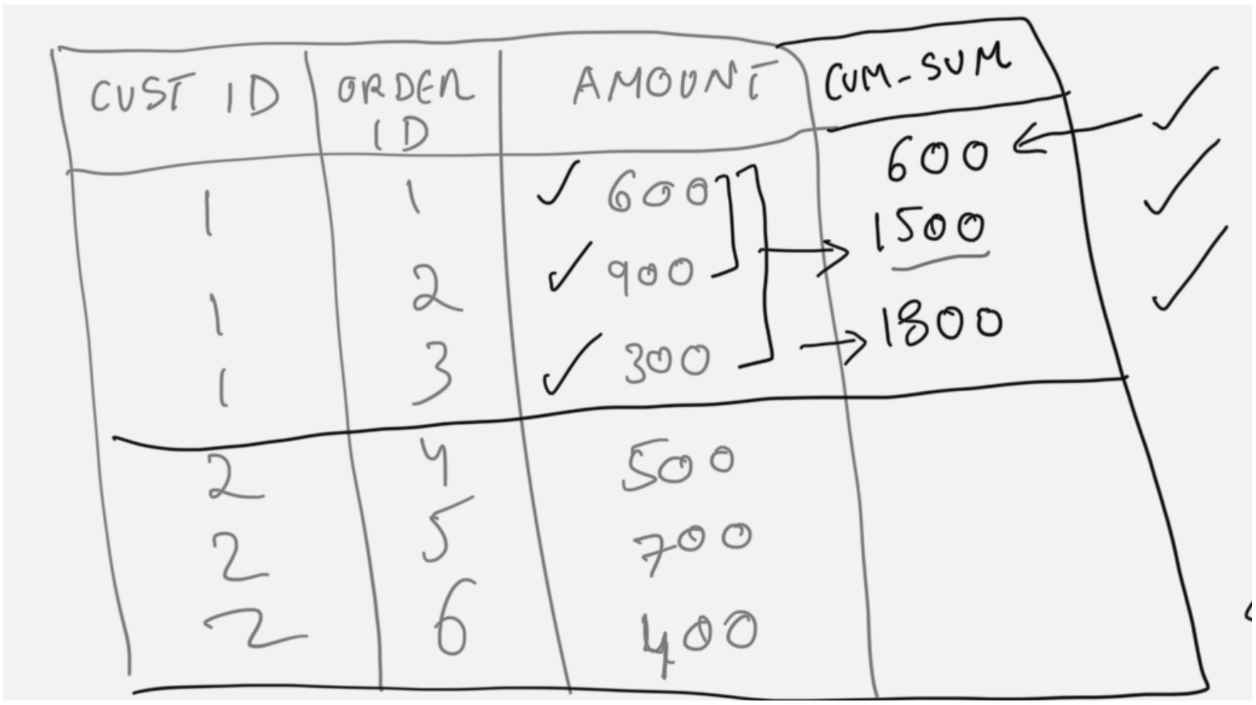
Cumulative / Running Sum

For calculating the **sum on an entire window**, we do something like this -

1. SUM(amount) OVER(PARTITION BY cust_id)
2. SUM(salary) OVER(PARTITION BY department_id)

This will show the same value for all rows within a partition.

Illustration ->



CUST ID	ORDER ID	AMOUNT	CUM-SUM
1	1	✓ 600	600
1	2	✓ 900	1500
1	3	✓ 300	1800
2	4	500	
2	5	700	
2	6	400	

For calculating the **sum on a window frame** i.e., from top till the current row -

1. SUM(amount) OVER(PARTITION BY cust_id ORDER BY order_id)
2. SUM(salary) OVER(PARTITION BY department_id ORDER BY employee_id)

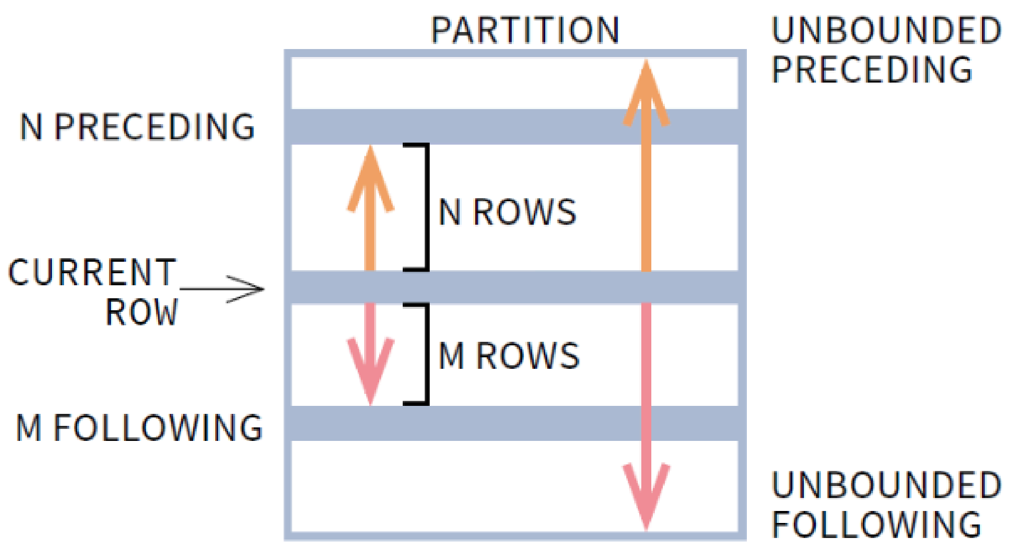
The ORDER BY clause in the window gives us a cumulative/running sum.

Window Frames (ADVANCED CONTROL)

Query Breakdown:

The above query is internally represented as: SUM(salary) OVER(PARTITION BY department_id ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

Please note that this is the **default** window frame for SQL.



- **N PRECEDING**: Specifies the number of rows before the current row that should be included in the window frame.
- **UNBOUNDED PRECEDING**: It indicates that the window starts at the first row of the partition.
- **CURRENT ROW**: It indicates the window begins or ends at the current row.
- **N FOLLOWING**: It includes the rows that follow the current row, up to the specified "N" rows. The current row is not included in the window frame.
- **UNBOUNDED FOLLOWING**: It indicates that the window ends at the last row of the partition.

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 hire_date,
 SUM(salary) OVER(PARTITION BY department_id ORDER BY hire_date)
AS cum_sal1,
 SUM(salary) OVER(PARTITION BY department_id ORDER BY hire_date
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cum_sal2,
FROM hr.employees;
```

In the result set, you can see that both the columns - **cum_sal1** and **cum_sal2** have the exact same data.

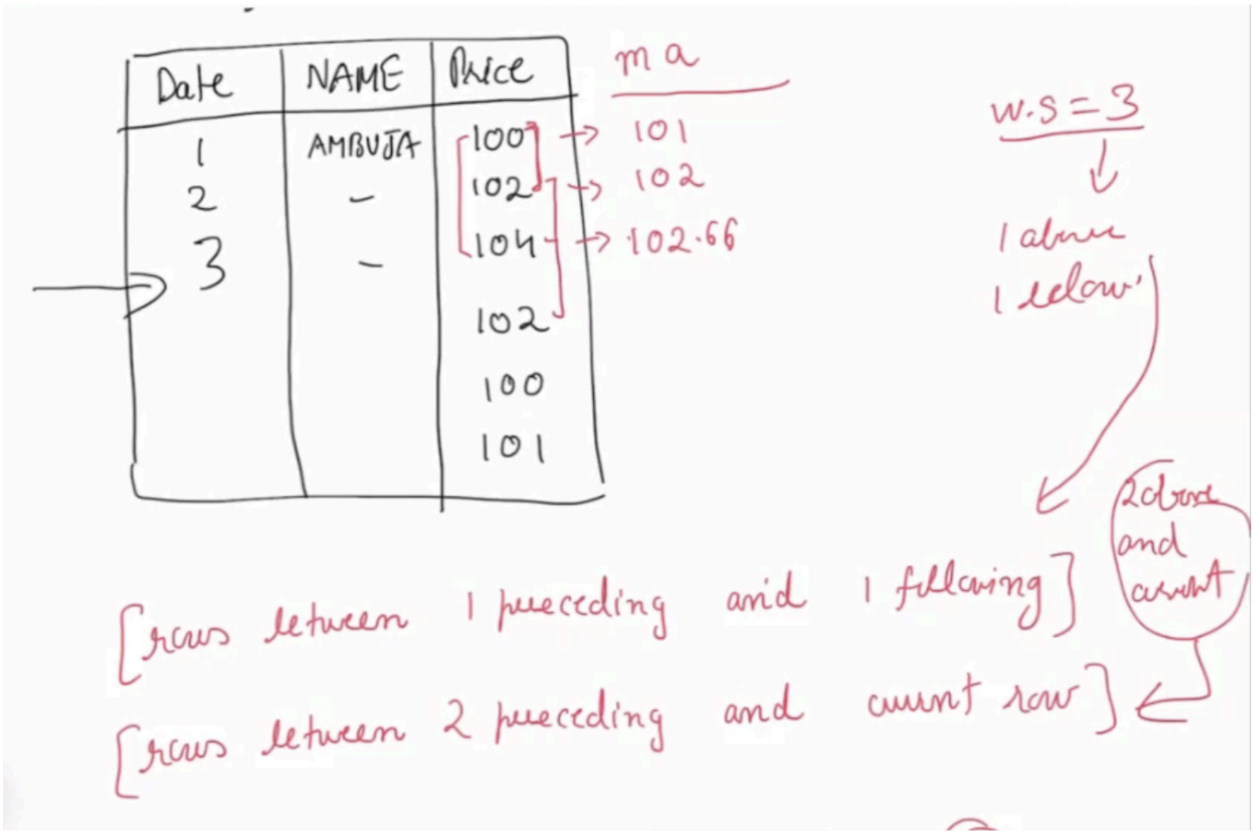
#Q: Display the cumulative / running sum of salaries in each department, ordered by the employees' joining dates.

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 hire_date,
 SUM(salary) OVER(PARTITION BY department_id) AS total_dept_salary,
 SUM(salary) OVER(PARTITION BY department_id ORDER BY hire_date)
AS running_dept_salary
FROM hr.employees;
```

Moving Average/Sum

Illustration ->



#Q: Display the moving sum of salaries in each department, with a window size of N=3, ordered by the employees' joining dates.

Query:

```
SELECT
 employee_id,
 department_id,
 salary,
 hire_date,
 SUM(salary) OVER(PARTITION BY department_id ORDER BY hire_date
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) mov_sum_1,
 SUM(salary) OVER(PARTITION BY department_id ORDER BY hire_date
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) mov_sum_2,
FROM hr.employees;
```

- The window frame specified in the first moving sum (**mov_sum_1**) considers the current row, the row before it, and the row after it.
- The window frame specified in the second moving sum (**mov_sum_2**) considers the current row and the two rows before it.

ROWS vs. RANGE clause

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

The ROWS clause **specifies a fixed number of rows** that precede or follow the current row regardless of their value.

On the other hand, the RANGE clause **includes all rows with values falling within a specified range** relative to the current row.

The distinction between ROWS and RANGE becomes more apparent when dealing with columns that may have **repeated** or **duplicate** values.

Quiz - 3

```
Q. Sum(sales) over(order by date) is the same as sum(sales)
over(order by date range between current row and unbounded following)

   a) True
   b) False - correct
```

Quiz - 4

```
Q. Assume we have monthly sales data. Which is the right query to get
the running average of sales by month?

   a) Avg(sales) over (order by month) - correct
   b) Avg(sales) over (partition by month)
   c) Avg(sales) over (partition by month order by month range
      between unbounded preceding and current row)
   d) Avg(sales) over (order by month range between unbounded
      preceding and unbounded following)
```

Quiz - 5

```
Q. Right function to get the moving average of the past 3
days(including today)

  a) Avg(sales) over (order by date curr row and 3 rows following)
  b) Avg(sales) over (order by date)
  c) Avg(sales) over (order by date 3 rows pred and curr row) -
     correct
  d) Avg(sales) over (order by date 2 rows pred and 1 rows
     following)
```

First_value() and Last_value()

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]
) AS alias
```

Similarly, the **LAST_VALUE**() window function returns the value from the **last row within a specified window frame**.

There's also an **NTH_VALUE**() 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]
) AS alias
```

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

#Q: Compare each employee's salary with the salaries of the first and last hired employees in their department.

Query:

```
SELECT
  department_id, employee_id, hire_date, salary,
  FIRST_VALUE(salary) OVER (PARTITION BY department_id ORDER BY
hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING) AS first_hired_emp_sal,
  LAST_VALUE(salary) OVER (PARTITION BY department_id ORDER BY
hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING) AS last_hired_emp_sal,
FROM hr.employees
ORDER BY department_id;
```

Quiz - 6

Q. Assume a table contains marks, students and scores. When we want to see the students that have scored the 4th highest mark in each subject and the name of the student should be displayed in each row within each subject, then which one to use

```
a) Rank()
b) DenseRank()
c) Nth_value() - correct
d) First_value()
```

How can GlobalTech Solutions benefit from our analysis?

- **Salary Comparison and Equity:** Ensuring fair compensation practices and identifying potential disparities.

- **Performance Evaluation:** Evaluating employee performance and identifying areas where compensation may need adjustment to align with performance.

- **Budgeting and Resource Allocation:** Determining how much each department spends on salaries and whether adjustments are needed.

- **Insights into Employee Satisfaction:** Disparities in salaries may indicate areas where employee morale and satisfaction could be improved.

- **Identifying Trends and Patterns:** Analyzing salary data over time can reveal trends and patterns in compensation practices, such as salary growth rates.

- **Compliance and Risk Management:** Ensuring equitable salary practices and compliance with regulations is crucial for mitigating legal and reputational risks.