■ Ranking Functions Recap

ROW_NUMBER() → assigns a unique number to each row.

RANK() → assigns rank with gaps when values tie.

DENSE_RANK() → assigns rank without gaps when values tie.

■ But how are these ranks decided?

■ On What Basis Is Rank Given?

Ranking is always based on the ORDER BY inside the OVER() clause.

Example:
SELECT employee_id, salary,
RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM employees;

Explanation:

ORDER BY salary DESC → sorts employees by highest salary first.

The rank is then assigned in that sorted order.

If two employees have the same salary:

RANK() → gives them the same rank but skips the next number (gap).

DENSE_RANK() → gives them the same rank without skipping.

ROW_NUMBER() → still forces a unique number for each row.

■ Is Sorting Required?

■ Yes.

Without ORDER BY, the rank would be meaningless, because there's no basis to assign it.

Think of ranking like a sports leaderboard → you must sort by score/time before assigning positions.

■ Example with Tied Values
SELECT employee_id, salary,
ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
RANK() OVER (ORDER BY salary DESC) AS rank_num,
DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank_num
FROM employees;

employee_id salary ROW_NUMBER RANK DENSE_RANK
E1 90000 1 1 1
E2 90000 2 1 1
E3 85000 3 3 2
E4 80000 4 4 3

■ So in interview answer:

"Ranking functions depend on the ORDER BY clause inside the window function. You must specify the column(s) to sort on, because rank is assigned in that order. Without ORDER BY, ranking has no meaning."

■ Example with Aggregates in Window Functions

```
SELECT new_id, new_cat,
SUM(new_id) OVER (ORDER BY new_id
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Total",
AVG(new_id) OVER (ORDER BY new_id
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Average",
COUNT(new_id) OVER (ORDER BY new_id
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Count",
MIN(new_id) OVER (ORDER BY new_id
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Min",
MAX(new_id) OVER (ORDER BY new_id
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "Max"
FROM test_data;
```

■ What does ROWS BETWEEN ... AND ... mean?

This is the window frame — it defines which rows are considered for each calculation.

UNBOUNDED PRECEDING → from the very first row in the partition.

UNBOUNDED FOLLOWING → till the very last row in the partition.

■ Together: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING means → take the entire partition (all rows) for every calculation.

That’s why in the result:

SUM is the same (2500) for each row,

AVG is the same,

COUNT is the same,

MIN and MAX are constant (100 and 700).

■ How to set a limit instead of entire table?

You can restrict the frame to just a few rows before/after the current row:

Current row only → Current row + 2 next rows

ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING

■ Sliding window (like moving average)

```
AVG(new_id) OVER (
ORDER BY new_id
ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
)
```

■ This will calculate average for the current row considering itself + 2 rows before + 2 rows after.

■ Example: Moving Sum of Last 3 Orders

```
SELECT order_id, amount,
```

```
SUM(amount) OVER (
ORDER BY order_id
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
) AS moving_sum
FROM orders;
```

■ For each row → it takes that row's amount + previous 2 rows, gives a sliding total.

■ So in your screenshot, they chose all rows in the partition (like a static aggregate repeated row-wise).
But by adjusting the frame (PRECEDING, FOLLOWING), you can create sliding / moving calculations.

■ Value Window Functions in SQL

1■■ LAG()

Definition:

Returns the value from a previous row in the result set (based on ORDER BY).

Useful for comparing the current row with the row before it.

Syntax:

LAG(column_name, offset, default_value) OVER (ORDER BY column_name)

offset → how many rows back (default is 1).

default_value → what to return if the previous row doesn't exist.

Example:

```
SELECT order_id, order_date,
LAG(order_date) OVER (ORDER BY order_date) AS prev_order
FROM orders;
```

■ Each order shows the previous order date. The first row will have NULL (or default value).

2■■ LEAD()

Definition:

Opposite of LAG → gets the value from a next row.

Useful for comparing with future rows.

Syntax:

LEAD(column_name, offset, default_value) OVER (ORDER BY column_name)

Example:

```
SELECT order_id, order_date,
LEAD(order_date) OVER (ORDER BY order_date) AS next_order
FROM orders;
```

■ Each row shows the next order date. The last row will have NULL (or default value).

3■■ FIRST_VALUE()

Definition:

Returns the first value in the partition (based on ORDER BY).

Syntax:

FIRST_VALUE(column_name) OVER (PARTITION BY dept_id ORDER BY salary DESC)

Example:

```
SELECT employee_id, dept_id, salary,
FIRST_VALUE(salary) OVER (PARTITION BY dept_id ORDER BY salary DESC) AS
highest_salary_in_dept
FROM employees;
```

■ For each employee, you also see the highest salary in their department.

4■■ LAST_VALUE()

Definition:

Returns the last value in the partition (based on ORDER BY).

■■ Note: By default, LAST_VALUE looks at the frame (ROWS BETWEEN ...).
If you don’t set the frame to include the whole partition, it may just return the current row’s value.
So usually we use:

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

Example:

```
SELECT employee_id, dept_id, salary,
LAST_VALUE(salary) OVER (
PARTITION BY dept_id
ORDER BY salary DESC
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
) AS lowest_salary_in_dept
FROM employees;
```

■ For each employee, you also see the lowest salary in their department.

■ Why Do We Use These Functions?

■ LAG / LEAD:

Compare row-to-row values → e.g., difference between current and previous sales.

Detect trends (e.g., stock prices up/down compared to yesterday).

Time-based calculations (days between two orders).

■ FIRST_VALUE / LAST_VALUE:

Find first/last value in a range (e.g., first purchase date, last transaction).

Useful in partitioned analytics (e.g., department's highest/lowest salary).

Helpful in reporting and BI dashboards (show first/last entry quickly).

■ Example Use Case: Sales Growth

```
SELECT month, sales,
LAG(sales) OVER (ORDER BY month) AS prev_month_sales,
sales - LAG(sales) OVER (ORDER BY month) AS growth
FROM monthly_sales;
```

■ This shows month-over-month growth.

■ Interview Answer (short):

LEAD and LAG let us access next/previous row values without self-joins. FIRST_VALUE and LAST_VALUE return the first and last value in a window/partition. These are used for trend analysis, comparisons, and analytics. For example, finding month-over-month sales growth, or highest/lowest salary in each department.