

# **Enhancing Query Proficiency**



Window functions

**Agenda** 

Subqueries



# Let's begin the discussion by answering a few questions



A company wants to rank employees within each department based on salary, ensuring that employees with the same salary receive the same rank How can they achieve this?

- A COUNT() OVER (PARTITION BY department ORDER BY salary DESC)
- B ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC)
- c RANK() OVER (PARTITION BY department ORDER BY salary DESC)
- ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary ASC)



A company wants to rank employees within each department based on salary, ensuring that employees with the same salary receive the same rank How can they achieve this?

- A COUNT() OVER (PARTITION BY department ORDER BY salary DESC)
- B ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC)
- c RANK() OVER (PARTITION BY department ORDER BY salary DESC)
- ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary ASC)

## Window Function: RANK()



#### sales

sale_id	employee	department	sale_amount	sale_date
1	Alice	Electronics	500	2024-02-10
2	Bob	Electronics	700	2024-02-11
3	Charlie	Electronics	700	2024-02-12
4	David	Furniture	400	2024-02-10
5	Emma	Furniture	800	2024-02-11
6	Frank	Furniture	300	2024-02-12
7	Grace	Furniture	800	2024-02-13

RANK assigns a unique rank to each row within a partition, allowing ties by skipping subsequent ranks.

```
SELECT *,
   RANK() OVER(
     PARTITION BY department
     ORDER BY sale_amount DESC) AS rank
FROM sales;
```

sale_id	employee	department	sale_amount	sale_date	rank
2	Bob	Electronics	700	2024-02-11	1
3	Charlie	Electronics	700	2024-02-12	1
1	Alice	Electronics	500	2024-02-10	3
5	Emma	Furniture	800	2024-02-11	1
7	Grace	Furniture	800	2024-02-13	1
4	David	Furniture	400	2024-02-10	3
6	Frank	Furniture	300	2024-02-12	4

#### Window Function: ROW\_NUMBER()



#### sales

sale_id	employee	department	sale_amount	sale_date
1	Alice	Electronics	500	2024-02-10
2	Bob	Electronics	700	2024-02-11
3	Charlie	Electronics	700	2024-02-12
4	David	Furniture	400	2024-02-10
5	Emma	Furniture	800	2024-02-11
6	Frank	Furniture	300	2024-02-12
7	Grace	Furniture	800	2024-02-13

ROW\_NUMBER assigns a unique sequential number to each row within a partition, without skipping numbers, even if there are ties.

```
SELECT *,
  ROW_NUMBER() OVER(
    PARTITION BY department
    ORDER BY sale_amount DESC) AS row
FROM sales;
```

sale_id	employee	department	sale_amount	sale_date	row
2	Bob	Electronics	700	2024-02-11	1
3	Charlie	Electronics	700	2024-02-12	2
1	Alice	Electronics	500	2024-02-10	3
5	Emma	Furniture	800	2024-02-11	1
7	Grace	Furniture	800	2024-02-13	2
4	David	Furniture	400	2024-02-10	3
6	Frank	Furniture	300	2024-02-12	4



# A clinic wants to list each customer's next scheduled appointment. Which function should be used?

- LEAD(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)
- LAG(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)
- RANK(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)
- NEXT(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)



## A clinic wants to list each customer's next scheduled appointment. Which function should be used?

- LEAD(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)
- LAG(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)
- RANK(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)
- NEXT(appointment\_date) OVER (PARTITION BY customer\_id ORDER BY appointment\_date)

#### **Window Function: LEAD()**



#### sales

sale_id	employee	department	sale_amount	sale_date
1	Alice	Electronics	500	2024-02-10
2	Bob	Electronics	700	2024-02-11
3	Charlie	Electronics	700	2024-02-12
4	David	Furniture	400	2024-02-10
5	Emma	Furniture	800	2024-02-11
6	Frank	Furniture	300	2024-02-12
7	Grace	Furniture	800	2024-02-13

**LEAD** returns the value of a column from a **subsequent row within the same partition**, allowing forward-looking comparisons.

```
SELECT *,
  LEAD(sale_date) OVER(
    PARTITION BY department
    ORDER BY sale_date) AS next
FROM sales;
```

sale_id	employee	department	sale_amount	sale_date	next
1	Alice	Electronics	500	2024-02-10	2024-02-11
2	Bob	Electronics	700	2024-02-11	2024-02-12
3	Charlie	Electronics	700	2024-02-12	
4	David	Furniture	400	2024-02-10	2024-02-11
5	Emma	Furniture	800	2024-02-11	2024-02-12
6	Frank	Furniture	300	2024-02-12	2024-02-13
7	Grace	Furniture	800	2024-02-13	

## Window Function: LAG()



#### sales

sale_id	employee	department	sale_amount	sale_date
1	Alice	Electronics	500	2024-02-10
2	Bob	Electronics	700	2024-02-11
3	Charlie	Electronics	700	2024-02-12
4	David	Furniture	400	2024-02-10
5	Emma	Furniture	800	2024-02-11
6	Frank	Furniture	300	2024-02-12
7	Grace	Furniture	800	2024-02-13

LAG returns the value of a column from a preceding row within the same partition, enabling backward-looking comparisons.

```
SELECT *,
  LAG(sale_date) OVER(
    PARTITION BY department
    ORDER BY sale_date) AS prev
FROM sales;
```

sale_id	employee	department	sale_amount	sale_date	prev
1	Alice	Electronics	500	2024-02-10	
2	Bob	Electronics	700	2024-02-11	2024-02-10
3	Charlie	Electronics	700	2024-02-12	2024-02-11
4	David	Furniture	400	2024-02-10	
5	Emma	Furniture	800	2024-02-11	2024-02-10
6	Frank	Furniture	300	2024-02-12	2024-02-11
7	Grace	Furniture	800	2024-02-13	2024-02-12

#### **Enhancing Query Proficiency: Quiz**



A company wants to find the average salary per department and then list only the departments where the average salary is above 50,000.

Α

```
SELECT department_id, AVG(salary)
FROM employees
WHERE AVG(salary) > 50000
GROUP BY department_id;
```

В

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 50000;
```

C

```
FROM (
    SELECT
          department_id, avg_salary
FROM (
          SELECT
          department_id,
          AVG (salary) AS avg_salary
          FROM employees
          GROUP BY department_id)
WHERE avg_salary > 50000;
```

D

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id;
```

#### **Enhancing Query Proficiency: Quiz**



A company wants to find the average salary per department and then list only the departments where the average salary is above 50,000.

A SELECT department\_id, AVG(salary)
FROM employees
WHERE AVG(salary) > 50000
GROUP BY department\_id;

В

SELECT department\_id, AVG (salary)
FROM employees
GROUP BY department\_id
HAVING AVG (salary) > 50000;

С

```
SELECT department_id, avg_salary
FROM (
    SELECT
          department_id,
          AVG (salary) AS avg_salary
    FROM employees
    GROUP BY department_id)
WHERE avg_salary > 50000;
```

D

SELECT department\_id, AVG(salary)
FROM employees
GROUP BY department\_id;

#### **Subqueries in SQL**



#### What is a Subquery?

A subquery is a query nested inside another SQL query

Subqueries can be used anywhere within the main query, they are commonly used within the FROM and WHERE clauses.

FROM clause as a temporary table

WHERE clause as a filter

#### **Subqueries in SQL**



FROM clause as a temporary table:

```
-- Find the third most expensive product

SELECT PRODUCT_NAME

FROM (

SELECT

RANK() OVER(

ORDER BY PRODUCT_PRICE DESC) AS RNK,

PRODUCT_NAME

FROM PRODUCT)

WHERE RNK = 3;
```

**WHERE** clause as a filter:

```
-- Display the names of products whose price is
above the average price for all products
SELECT PRODUCT_NAME
FROM PRODUCT
WHERE PRODUCT_PRICE > (
    SELECT AVG(PRODUCT_PRICE)
    FROM PRODUCT);
```



#### Which is the correct order of execution in SQL?

- A SELECT  $\rightarrow$  FROM  $\rightarrow$  WHERE  $\rightarrow$  GROUP BY  $\rightarrow$  HAVING  $\rightarrow$  ORDER BY
- **B** FROM  $\rightarrow$  GROUP BY  $\rightarrow$  WHERE  $\rightarrow$  HAVING  $\rightarrow$  SELECT  $\rightarrow$  ORDER BY
- c FROM  $\rightarrow$  WHERE  $\rightarrow$  GROUP BY  $\rightarrow$  HAVING  $\rightarrow$  SELECT  $\rightarrow$  ORDER BY
- SELECT  $\rightarrow$  GROUP BY  $\rightarrow$  HAVING  $\rightarrow$  ORDER BY  $\rightarrow$  FROM  $\rightarrow$  WHERE



#### Which is the correct order of execution in SQL?

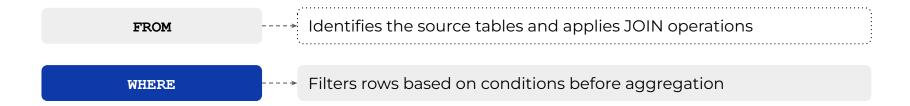
- SELECT  $\rightarrow$  FROM  $\rightarrow$  WHERE  $\rightarrow$  GROUP BY  $\rightarrow$  HAVING  $\rightarrow$  ORDER BY
- **B** FROM  $\rightarrow$  GROUP BY  $\rightarrow$  WHERE  $\rightarrow$  HAVING  $\rightarrow$  SELECT  $\rightarrow$  ORDER BY
- c FROM  $\rightarrow$  WHERE  $\rightarrow$  GROUP BY  $\rightarrow$  HAVING  $\rightarrow$  SELECT  $\rightarrow$  ORDER BY
- SELECT  $\rightarrow$  GROUP BY  $\rightarrow$  HAVING  $\rightarrow$  ORDER BY  $\rightarrow$  FROM  $\rightarrow$  WHERE



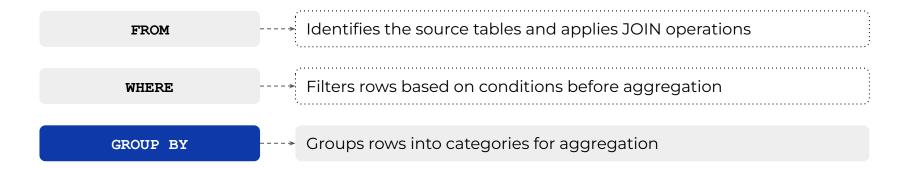
FROM

Identifies the source tables and applies JOIN operations

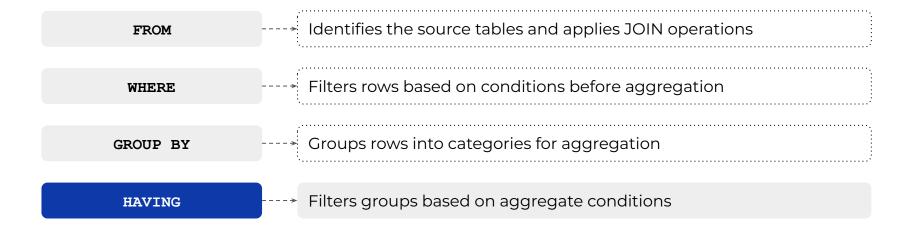














FROM	> Identifies the source tables and applies JOIN operations
WHERE	Filters rows based on conditions before aggregation
GROUP BY	Groups rows into categories for aggregation
HAVING	Filters groups based on aggregate conditions
SELECT	Determines which columns or expressions to return



FROM	Identifies the source tables and applies JOIN operations	
WHERE	> Filters rows based on conditions before aggregation	
GROUP BY	Groups rows into categories for aggregation	
HAVING	> Filters groups based on aggregate conditions	
SELECT	> Determines which columns or expressions to return	
ORDER BY	> Sorts the result set	



FROM	Identifies the source tables and applies JOIN operations	
WHERE	> Filters rows based on conditions before aggregation	
GROUP BY	Groups rows into categories for aggregation	
HAVING	> Filters groups based on aggregate conditions	
SELECT	> Determines which columns or expressions to return	
ORDER BY	> Sorts the result set	
LIMIT	> Restricts the number of rows returned	

**G**Great Learning

**Happy Learning!** 

