CREATE TABLE employees (

department\_id INT,

employee\_id INT,

salary INT

);

INSERT INTO employees (department\_id, employee\_id, salary) VALUES

(1, 101, 80000),

(1, 102, 75000),

(1, 103, 75000),

(1, 104, 60000),

(2, 201, 90000),

(2, 202, 90000),

(2, 203, 85000),

(2, 204, 80000);

--The ROW\_NUMBER() function assigns a unique number to each row within a partition of a result set,

--starting at 1 for the first row in each partition. Think of it as giving a unique serial number to each row.

--The RANK() function assigns a rank to each row within a partition of a result set.

--If two rows have the same value, they receive the same rank, and the next rank(s) are skipped.

--The DENSE\_RANK() function is similar to RANK(), but it does not skip any ranks after ties.

SELECT department\_id, employee\_id, salary,

ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS row\_number,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS dense\_rank

FROM employees;

SELECT employee\_id, employee\_name, salary,

NTILE(4) OVER (ORDER BY salary ASC) AS quartile

FROM employees;

CREATE TABLE employees2 (

employee\_id INT,

employee\_name VARCHAR(50),

salary INT

);

INSERT INTO empl (employee\_id, employee\_name, salary) VALUES

(1, 'Alice', 50000),

(2, 'Bob', 60000),

(3, 'Charlie', 70000),

(4, 'Diana', 80000),

(5, 'Edward', 90000),

(6, 'Fiona', 100000),

(7, 'George', 110000),

(8, 'Hannah', 120000);

----------------------------------------------------------------------------------------------------------------------

CREATE TABLE employees (

department\_id INT,

employee\_id INT,

salary INT

);

INSERT INTO employees (department\_id, employee\_id, salary) VALUES

(1, 101, 80000),

(1, 102, 75000),

(1, 103, 75000),

(1, 104, 70000),

(2, 201, 90000),

(2, 202, 85000),

(2, 203, 85000),

(2, 204, 85000),

(2, 205, 80000);

SELECT department\_id, employee\_id, salary,

ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS row\_number,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS dense\_rank

FROM employees;

ROW\_NUMBER: Unique row numbers for each row.

RANK: Gaps in ranks for tied values.

DENSE\_RANK: No gaps in ranks, even with ties.

SELECT employee\_id, employee\_name, salary,

NTILE(4) OVER (ORDER BY salary ASC) AS quartile

FROM employees;

**NTILE(4)**: Divides the rows into 4 approximately equal groups (quartiles).

**OVER (ORDER BY salary ASC)**: Assigns the quartile number based on the salary in ascending order.

—------------------

CREATE TABLE employees (

department\_id INT,

employee\_id INT,

salary INT

);

INSERT INTO employees (department\_id, employee\_id, salary) VALUES

(1, 101, 80000),

(1, 102, 75000),

(1, 103, 75000),

(1, 104, 70000),

(2, 201, 90000),

(2, 202, 85000),

(2, 203, 85000),

(2, 204, 85000),

(2, 205, 80000);

SELECT department\_id, employee\_id, salary,

ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS row\_number,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS DENSE\_RANK,

-- SUM(salary) OVER (PARTITION BY department\_id ORDER BY employee\_id) AS running\_total

FROM employees;

SELECT employee\_id, salary,

NTILE(4) OVER (ORDER BY salary ASC) AS quartile

FROM employees;

--------------------------

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

department\_id INT,

salary DECIMAL(10, 2)

);

CREATE TABLE sales\_data (

sale\_id INT PRIMARY KEY,

employee\_id INT,

sales DECIMAL(10, 2),

FOREIGN KEY (employee\_id) REFERENCES employees(employee\_id)

);

INSERT INTO employees (employee\_id, first\_name, last\_name, department\_id, salary) VALUES

(1, 'John', 'Doe', 1, 60000),

(2, 'Jane', 'Smith', 2, 80000),

(3, 'Jim', 'Brown', 3, 90000),

(4, 'Jake', 'White', 4, 70000),

(5, 'Jill', 'Green', 5, 75000),

(6, 'Jack', 'Black', 3, 95000),

(7, 'Jerry', 'Gray', 2, 82000);

INSERT INTO sales\_data (sale\_id, employee\_id, sales) VALUES

(1, 1, 1000),

(2, 2, 1500),

(3, 3, 2000),

(4, 4, 700),

(5, 5, 1300),

(6, 6, 1750),

(7, 7, 1200);

SELECT e.employee\_id, e.first\_name, e.last\_name, e.department\_id, s.sales,

RANK() OVER (PARTITION BY e.department\_id ORDER BY s.sales DESC) AS sales\_rank,

DENSE\_RANK() OVER (PARTITION BY e.department\_id ORDER BY s.sales DESC) AS dense\_sales\_rank

FROM employees e

JOIN sales\_data s ON e.employee\_id = s.employee\_id;

SELECT employee\_id, first\_name, last\_name, department\_id, salary,

NTILE(4) OVER (PARTITION BY department\_id ORDER BY salary DESC) AS salary\_quartile

FROM employees;

SELECT e.employee\_id, e.first\_name, e.last\_name, e.department\_id, s.sales,

SUM(s.sales) OVER (PARTITION BY e.department\_id ORDER BY e.employee\_id) AS running\_total

FROM employees e

JOIN sales\_data s ON e.employee\_id = s.employee\_id;

-----------------

--Task 1.1

SELECT department\_id, first\_name, last\_name, salary,

ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS salary\_rank

FROM employees;

--Task 1.2

SELECT department\_id, first\_name, last\_name, salary,

SUM(salary) OVER (PARTITION BY department\_id ORDER BY salary DESC) AS cumulative\_salary

FROM employees;

--Task 2.1

SELECT e.department\_id, e.first\_name, e.last\_name, s.sales,

DENSE\_RANK() OVER (PARTITION BY e.department\_id ORDER BY s.sales DESC) AS sales\_rank

FROM employees e

JOIN sales\_data s ON e.employee\_id = s.employee\_id;

--Task 2.2

SELECT e.employee\_id, e.first\_name, e.last\_name, e.department\_id, s.sales,

SUM(s.sales) OVER (ORDER BY e.employee\_id) AS running\_total\_sales

FROM employees e

JOIN sales\_data s ON e.employee\_id = s.employee\_id;

SELECT employee\_id, first\_name, last\_name, salary, department\_id,

MAX(salary) OVER (PARTITION BY department\_id ORDER BY salary) AS max\_salary

FROM employees;

--------------------

-- Create the employees table with a consistent employee\_id and an additional month column

CREATE TABLE employees (

employee\_id INT,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

salary NUMERIC(10, 2),

department\_id INT,

MONTH VARCHAR(20),

PRIMARY KEY (employee\_id, month)

);

-- Insert data into the employees table with consistent employee\_id and monthly salary variations

INSERT INTO employees (employee\_id, first\_name, last\_name, salary, department\_id, month) VALUES

-- Department 1: John Doe

(1, 'John', 'Doe', 60000, 1, 'January'),

(1, 'John', 'Doe', 61000, 1, 'February'),

(1, 'John', 'Doe', 60500, 1, 'March'),

-- Department 1: Jane Smith

(2, 'Jane', 'Smith', 80000, 1, 'January'),

(2, 'Jane', 'Smith', 82000, 1, 'February'),

(2, 'Jane', 'Smith', 81500, 1, 'March'),

-- Department 1: Jerry Gray

(3, 'Jerry', 'Gray', 82000, 1, 'January'),

(3, 'Jerry', 'Gray', 83000, 1, 'February'),

(3, 'Jerry', 'Gray', 82500, 1, 'March'),

-- Department 2: Jim Brown

(4, 'Jim', 'Brown', 90000, 2, 'January'),

(4, 'Jim', 'Brown', 91000, 2, 'February'),

(4, 'Jim', 'Brown', 90500, 2, 'March'),

-- Department 2: Jack Black

(5, 'Jack', 'Black', 95000, 2, 'January'),

(5, 'Jack', 'Black', 96000, 2, 'February'),

(5, 'Jack', 'Black', 95500, 2, 'March'),

-- Department 3: Jake White

(6, 'Jake', 'White', 70000, 3, 'January'),

(6, 'Jake', 'White', 71000, 3, 'February'),

(6, 'Jake', 'White', 70500, 3, 'March'),

-- Department 3: Jill Green

(7, 'Jill', 'Green', 75000, 3, 'January'),

(7, 'Jill', 'Green', 76000, 3, 'February'),

(7, 'Jill', 'Green', 75500, 3, 'March');

-- Query 1: Calculate the maximum salary for each employee

SELECT employee\_id, first\_name, department\_id, last\_name, month, salary,

MAX(salary) OVER (PARTITION BY employee\_id ORDER BY salary desc) AS max\_salary

FROM employees;

-- Query 2: Calculate the minimum salary for each employee

SELECT employee\_id, first\_name, last\_name, department\_id, month, salary,

MIN(salary) OVER (PARTITION BY employee\_id ORDER BY salary) AS min\_salary

FROM employees;

-- Query 3: Calculate the running total of salaries for each employee ordered by month

SELECT employee\_id, first\_name, last\_name, department\_id, month, salary,

SUM(salary) OVER (PARTITION BY employee\_id ORDER BY e.month ) AS running\_total

FROM employees e;

SELECT employee\_id, first\_name, last\_name, month, salary,

SUM(salary) OVER (PARTITION BY employee\_id) AS running\_total

FROM employees

ORDER BY employee\_id, month;

-- Query 4: Calculate the average salary within each department

SELECT employee\_id, first\_name, last\_name, department\_id, salary,

AVG(salary) OVER (PARTITION BY department\_id) AS avg\_salary

FROM employees

ORDER BY department\_id, salary;

-- Query 5: Count the number of employees within each department

SELECT employee\_id, first\_name, last\_name, department\_id, salary,

COUNT(\*) OVER (PARTITION BY department\_id, employee\_id) AS employee\_count

FROM employees

ORDER BY department\_id, employee\_id;

SELECT department\_id, COUNT (DISTINCT employee\_id)

FROM employees

GROUP BY department\_id

Frame Specifications

When we talk about selecting specific rows for our window function, we can specify that we want the window to be based on certain rows around the current row. For example, we can use the ROWS BETWEEN clause and specify the range as "one preceding and one following."

### **What does "one preceding and one following" mean?**

It means that for each row, the window will include:

* The current row.
* The row immediately before it.
* The row immediately after it.

Essentially, the window for each row will consist of three rows: one preceding, the current row, and one following.

### **Let’s visualize this with an example:**

Imagine we have a set of rows with salaries. For each row:

* The current row is included.
* One row before the current row is also included.
* One row after the current row is included.

If we are at the first row, there’s no preceding row, so the window will include just the current row and the row after it.

### **As we move to the next row:**

The window shifts forward:

* Now the current row becomes the new row in the middle.
* The previous current row becomes the preceding row.
* The next row becomes the following row.

This is what we mean by defining a "moving window" that updates as we move through each row.

#### **ROWS**

* The **ROWS** option operates on a strict row-by-row basis.
* When you specify, for example, **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING**, it includes exactly one row before and one row after the current row, regardless of whether these rows have the same value in the ordering column.
* This approach is precise and doesn't account for duplicates in the ordering column.

#### **RANGE**

* The **RANGE** option, on the other hand, considers all rows with the same value in the ordering column as being part of the same "range."
* For instance, if you're ordering by salary and there are duplicates, **RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING** will include all rows with the same salary value as the row before or after the current row.
* This method is less granular but useful for calculations involving logical groupings (e.g., treating all employees with the same salary as a single group).

### **When to Use ROWS vs. RANGE**

* Use **ROWS** when precision is necessary, such as when calculating metrics specific to exact positions in a dataset.
* Use **RANGE** when you want to group rows logically by their values in the ordering column, especially when duplicates need to be included.

### **Example Use Cases**

1. **ROWS**: Calculating a moving average or sum that strictly considers a fixed number of rows around the current one.
2. **RANGE**: Summing or averaging values grouped by logical equivalence (e.g., all employees with the same salary).

### **Summary**

* **ROWS** is exact and operates strictly on physical rows.
* **RANGE** groups rows logically based on their value in the ordering column.
* Both options provide flexibility, allowing you to tailor window calculations to specific needs.

-- Step 1: Create the sales table

CREATE TABLE sales (

sale\_id SERIAL PRIMARY KEY,

product VARCHAR(50),

sale\_amount NUMERIC

);

-- Step 2: Insert data into the sales table, including duplicates

INSERT INTO sales (product, sale\_amount)

VALUES

('ProductA', 100),

('ProductB', 200),

('ProductC', 200),

('ProductD', 300),

('ProductE', 300),

('ProductF', 400);

-- Verify the data

SELECT \* FROM sales;

--RANGE AND ROWS - DIFFERENCE

SELECT

product,

sale\_amount,

SUM(sale\_amount) OVER (

ORDER BY sale\_amount ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

) AS running\_sum\_rows

FROM sales;

SELECT

product,

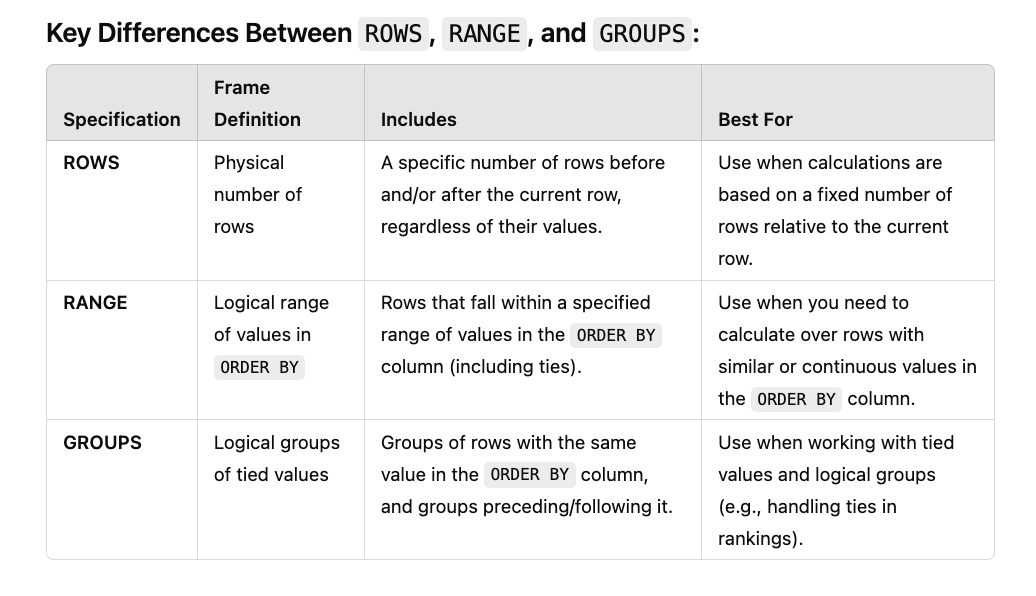
sale\_amount,

SUM(sale\_amount) OVER (

ORDER BY sale\_amount RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING

) AS running\_sum\_range

FROM sales;



**Combining Window Functions with CTEs**

Let's say we want to combine window functions with Common Table Expressions (CTEs). For example, we create a CTE that includes the employee ID, first name, last name, department ID, and salary. Then, we add a rank function to partition the data by department and rank employees' salaries within each department.

Give me the rank of the salaries within each department, where the highest salary gets rank 1. From this, I can later use the CTE to pull out specific information like the best or third-best salary in the department."

Next, if we want to work with subqueries combined with window functions, we can create another table that calculates, for example, the average salary across all departments. Then we compare each employee's salary to the department average.

WITH RankedEmployees AS ( SELECT employee\_id, first\_name, department\_id, salary, RANK() OVER (PARTITION BY department\_id OORDER BY salary DESC) AS salary\_rank FROM employees ) SELECT employee\_id, first\_name, department\_id, salary, salary\_rank FROM RankedEmployees WHERE salary\_rank = 1;

Query with Nested Subqueries:



**Inner Window Function:**

* The first subquery calculates the total sales for each salesperson using SUM(sales\_amount) OVER (PARTITION BY salesperson\_id).

**Nested Subquery for Average:**

* The second subquery calculates the **average total sales** across all salespersons.

**Outer Query Filter:**

* The outer query selects only the salespersons whose total sales exceed the average total sales calculated in the nested subquery.

**Performance Optimizations**

1. **Indexing:**
   * Indexing can be applied to columns used in window functions. For large-scale applications where the window function is queried repeatedly, indexing the columns can significantly improve performance.
2. **Filters and Conditions:**
   * Use filters to limit the number of rows processed by the query. This reduces computational overhead and speeds up query execution.
3. **Avoid Overlapping Windows:**
   * If multiple window functions are used on overlapping rows, consolidate them to minimize redundant calculations.