Ranking Functions in SQL: Solutions

Generated Examples

May 21, 2025

Contents

Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. (Same as in exercises document).

```sql
DROP TABLE IF EXISTS product\_sales;
DROP TABLE IF EXISTS employees;

CREATE TABLE employees (
    employee\_id INT PRIMARY KEY,
    first\_name VARCHAR(50),
    last\_name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10, 2),
    hire\_date DATE
);

INSERT INTO employees (employee\_id, first\_name, last\_name, department, salary, hire\_date) VALUES
(1, 'Alice', 'Smith', 'HR', 60000.00, '2020-01-15'),
(2, 'Bob', 'Johnson', 'HR', 65000.00, '2019-07-01'),
(3, 'Charlie', 'Williams', 'IT', 80000.00, '2021-03-10'),
(4, 'David', 'Brown', 'IT', 90000.00, '2018-05-20'),
(5, 'Eve', 'Jones', 'IT', 80000.00, '2022-01-05'),
(6, 'Frank', 'Garcia', 'Finance', 75000.00, '2020-11-01'),
(7, 'Grace', 'Miller', 'Finance', 75000.00, '2021-06-15'),
(8, 'Henry', 'Davis', 'Marketing', 70000.00, '2019-02-28'),
(9, 'Ivy', 'Rodriguez', 'Marketing', 72000.00, '2023-01-10'),
(10, 'Jack', 'Wilson', 'Marketing', 70000.00, '2020-08-15'),
(11, 'Karen', 'Moore', 'HR', 60000.00, '2021-09-01'),
(12, 'Liam', 'Taylor', 'IT', 95000.00, '2023-03-01');

CREATE TABLE product\_sales (
    sale\_id INT PRIMARY KEY,
    product\_name VARCHAR(100),
    category VARCHAR(50),
    sale\_date DATE,
    sale\_amount DECIMAL(10, 2),
    quantity\_sold INT
);

INSERT INTO product\_sales (sale\_id, product\_name, category, sale\_date, sale\_amount, quantity\_sold) VALUES
(1, 'Laptop Pro', 'Electronics', '2023-01-10', 1200.00, 5),
(2, 'Smartphone X', 'Electronics', '2023-01-12', 800.00, 10),
(3, 'Office Chair', 'Furniture', '2023-01-15', 150.00, 20),
(4, 'Desk Lamp', 'Furniture', '2023-01-18', 40.00, 30),
(5, 'Laptop Pro', 'Electronics', '2023-02-05', 1200.00, 3),
(6, 'Gaming Mouse', 'Electronics', '2023-02-10', 75.00, 50),
(7, 'Smartphone X', 'Electronics', '2023-02-15', 780.00, 8),
(8, 'Bookshelf', 'Furniture', '2023-02-20', 200.00, 10),
(9, 'Laptop Pro', 'Electronics', '2023-03-01', 1150.00, 4),
(10, 'External HDD', 'Electronics', '2023-03-05', 100.00, 25),
(11, 'Office Chair', 'Furniture', '2023-03-10', 140.00, 15),
(12, 'Desk Lamp', 'Furniture', '2023-03-15', 35.00, 40),
(13, 'Smartphone Y', 'Electronics', '2023-03-20', 900.00, 12),
(14, 'Coffee Maker', 'Appliances', '2023-01-20', 60.00, 10),
(15, 'Toaster', 'Appliances', '2023-02-25', 30.00, 15),
(16, 'Blender', 'Appliances', '2023-03-01', 50.00, 8);
```

Listing 1: Dataset for Employees and Product Sales Tables

1 Practice Meanings, Values, Relations, and Advantages

1.1 Basic ROW_NUMBER()

Problem: Assign a unique sequential number to each employee based on their hire date (oldest first).

Solution:

```sql
SELECT
    employee\_id,
    first\_name,
    last\_name,
    hire\_date,
    ROW\_NUMBER() OVER (ORDER BY hire\_date ASC) as hiring\_sequence
FROM
    employees;
```

Listing 2: Solution for i.1

1.2 Basic RANK()

Problem: Rank employees based on their salary in descending order. Show how ties are handled (gaps in rank).

Solution:

```sql
SELECT
    employee\_id,
    first\_name,
    last\_name,
    salary,
    RANK() OVER (ORDER BY salary DESC) as salary\_rank
FROM
    employees;
```

Listing 3: Solution for i.2

1.3 Basic DENSE_RANK()

Problem: Rank employees based on their salary in descending order using dense ranking. Show how ties are handled differently from `RANK()` (no gaps in rank).

Solution:

```sql
SELECT
    employee\_id,
    first\_name,
    last\_name,
    salary,
    DENSE\_RANK() OVER (ORDER BY salary DESC) as dense\_salary\_rank
FROM
    employees;
```

Listing 4: Solution for i.3

1.4 Comparing ROW_NUMBER(), RANK(), DENSE_RANK() within partitions

Problem: For each department, assign a unique row number, rank (with gaps), and dense rank (no gaps) to employees based on their salary in descending order.

Solution:

```sql
SELECT
    employee\_id,
    first\_name,
    last\_name,
    department,
    salary,
    ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as dept\_row\_num,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dept\_salary\_rank,
    DENSE\_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dept\_dense\_salary\_rank
FROM
    employees
ORDER BY
    department, salary DESC;
```

Listing 5: Solution for i.4

1.5 Advantage - Top N per group

Problem: Identify the top 2 highest-paid employees in each department.

Solution:

```sql
WITH RankedEmployees AS (
    SELECT
        employee\_id,
        first\_name,
        last\_name,
        department,
        salary,
        DENSE\_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dept\_dense\_salary\_rank
    FROM
        employees
)
SELECT
    employee\_id,
    first\_name,
    last\_name,
    department,
    salary,
    dept\_dense\_salary\_rank
FROM
    RankedEmployees
WHERE
    dept\_dense\_salary\_rank <= 2
ORDER BY
    department, dept\_dense\_salary\_rank;
```

Listing 6: Solution for i.5

2 Practice Disadvantages of Technical Concepts

2.1 Misinterpretation of RANK() vs DENSE_RANK() for Nth distinct value

Problem: A manager wants to identify all employees who are in the top 2 distinct salary tiers within the company. Demonstrate this by comparing results from `RANK()` and `DENSE_RANK()`. Specifically, if the requirement is "employees in the 4th highest salary tier company-wide", show how using `RANK() = 4` might yield no results, while `DENSE_RANK() = 4` would correctly identify them.

Solution: First, let's display company-wide ranks using both functions:

```sql
SELECT
    employee\_id, first\_name, salary,
    RANK() OVER (ORDER BY salary DESC) as company\_rank,
    DENSE\_RANK() OVER (ORDER BY salary DESC) as company\_dense\_rank
FROM employees
ORDER BY salary DESC;
```

Listing 7: Solution for ii.1 - Display Ranks

Observation from the above query output:

- Distinct salaries are: 95k, 90k, 80k, 75k, 72k, 70k, 65k, 60k.
- `DENSE_RANK` correctly numbers these distinct salary tiers from 1 to 8.
- `RANK` results:
  - Liam (95k) -> `company_rank`=1, `company_dense_rank`=1
  - David (90k) -> `company_rank`=2, `company_dense_rank`=2
  - Charlie (80k), Eve (80k) -> Both `company_rank`=3, `company_dense_rank`=3
  - Frank (75k), Grace (75k) -> Both `company_rank`=5, `company_dense_rank`=4
  - Ivy (72k) -> `company_rank`=7, `company_dense_rank`=5
  - Henry (70k), Jack (70k) -> Both `company_rank`=8, `company_dense_rank`=6
  - Bob (65k) -> `company_rank`=10, `company_dense_rank`=7
  - Alice (60k), Karen (60k) -> Both `company_rank`=11, `company_dense_rank`=8

If a manager asks for "employees in the 4th highest salary tier", using `company_rank = 4` would yield no results from the above. Using `company_dense_rank = 4` correctly identifies Frank and Grace (salary 75000.00).

Query to find employees in the 4th distinct salary tier using `DENSE_RANK()`:

```sql
SELECT
    employee\_id, first\_name, salary,
    DENSE\_RANK() OVER (ORDER BY salary DESC) as company\_dense\_rank
FROM employees
QUALIFY DENSE\_RANK() OVER (ORDER BY salary DESC) = 4;
-- Using subquery for broader compatibility:
-- WITH RankedSalaries AS (
-- SELECT employee\_id, first\_name, salary,
-- DENSE\_RANK() OVER (ORDER BY salary DESC) as company\_dense\_rank
-- FROM employees
```

```
-- )
-- SELECT * FROM RankedSalaries WHERE company\_dense\_rank = 4;
```

Listing 8: Solution for ii.1 - Correct identification

This demonstrates a disadvantage if `RANK()` is chosen when `DENSE_RANK()` is appropriate for the business question regarding distinct value tiers.

2.2 Potential for confusion with complex ORDER BY in window definition

Problem: Employees are ranked within their department by salary (descending). For employees with the same salary, their secondary sort key is hire date (ascending, earlier hire gets better rank). Show this ranking using `ROW_NUMBER()`. Then, demonstrate how if the secondary sort key (`hire_date`) was mistakenly ordered descending, it would change the row numbers for tied-salary employees, potentially leading to incorrect "top" employee identification if the secondary sort was critical for tie-breaking. Focus on employees in departments and salary groups where ties exist.

Solution:

```
-- Correct secondary sort (hire\_date ASC for tie-breaking)
SELECT
    department, first\_name, salary, hire\_date,
    ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC, hire\_date ASC) as rn\_hire\_asc
FROM employees
WHERE salary IN (80000, 75000, 70000, 60000) -- Focus on employees with tied salaries
ORDER BY department, salary DESC, hire\_date ASC;

-- Incorrect/different secondary sort (hire\_date DESC for tie-breaking)
SELECT
    department, first\_name, salary, hire\_date,
    ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC, hire\_date DESC) as rn\_hire\_desc
FROM employees
WHERE salary IN (80000, 75000, 70000, 60000) -- Focus on employees with tied salaries
ORDER BY department, salary DESC, hire\_date DESC;
```

Listing 9: Solution for ii.2 - Comparing ROW_NUMBER() with different secondary sort

Observation:

- In IT, for salary 80000.00: Charlie (hired 2021-03-10), Eve (hired 2022-01-05).
  - With `rn_hire_asc`: Charlie is 1st among 80k, Eve is 2nd. (After David 90k and Liam 95k in IT, so actual ranks are 3rd and 4th if considering top salary). The problem stated "salary (descending)", so within IT: Liam (95k) is 1, David (90k) is 2. Then for 80k: Charlie (hire 2021) is 3, Eve (hire 2022) is 4.
  - With `rn_hire_desc`: For 80k in IT: Eve (hire 2022, later) becomes 3, Charlie (hire 2021, earlier) becomes 4.
- Similar changes occur in HR for 60000.00 (Alice vs Karen) and Finance for 75000.00 (Frank vs Grace), and Marketing for 70000.00 (Henry vs Jack).

This shows how a subtle change in the `ORDER BY` clause within the window definition can significantly alter rankings for records that are tied on the primary sort key. This can be a source of confusion or error if the tie-breaking rule is critical and not correctly implemented or understood.

2.3 Conceptual disadvantage - Readability with many window functions

Problem: Display each employee's salary, their salary rank within their department, their overall salary rank in the company (dense), and a row number based on alphabetical order of their full name (`last_name`, then `first_name`). The query itself will demonstrate how multiple window functions can make the `SELECT` clause dense.

Solution:

```
SELECT
    e.employee\_id,
    e.first\_name,
    e.last\_name,
    e.department,
    e.salary,
    RANK() OVER (PARTITION BY e.department ORDER BY e.salary DESC) as dept\_salary\_rank,
    DENSE\_RANK() OVER (ORDER BY e.salary DESC) as overall\_company\_dense\_salary\_rank,
    ROW\_NUMBER() OVER (ORDER BY e.last\_name ASC, e.first\_name ASC) as alphabetical\_row\_num
FROM
    employees e
ORDER BY
    e.department, dept\_salary\_rank;
```

Listing 10: Solution for ii.3

Observation: The `SELECT` clause, while powerful, becomes somewhat lengthy with multiple window functions, each with its own `OVER()` clause. In much more complex queries involving joins, aggregations, and even more window functions with varying partitions and orderings, this part of the query can become very difficult to read, debug, and maintain. This highlights a "soft" disadvantage related to code complexity and readability. Using CTEs can help mitigate this but doesn't entirely remove the conceptual load of tracking multiple window contexts.

3 Practice Cases of Inefficient Alternatives

3.1 Find the employee(s) with the highest salary in each department

Problem: List the employee(s) with the highest salary in each department. If there are ties, list all. Provide both an inefficient solution (e.g., using a subquery with `MAX` for each department and joining back) and an efficient solution using ranking functions.

Solution: *Inefficient Solution (using subquery with MAX for each department):*

```sql
SELECT e.department, e.first\_name, e.last\_name, e.salary
FROM employees e
JOIN (
    SELECT department, MAX(salary) AS max\_salary
    FROM employees
    GROUP BY department
) AS dept\_max\_salary
ON e.department = dept\_max\_salary.department AND e.salary = dept\_max\_salary.max\_salary
ORDER BY e.department, e.salary DESC;
```

Listing 11: Inefficient Solution for iii.1

Efficient Solution (using DENSE_RANK()):

```sql
WITH RankedSalaries AS (
    SELECT
        department,
        first\_name,
        last\_name,
        salary,
        DENSE\_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as rn
    FROM employees
)
SELECT department, first\_name, last\_name, salary
FROM RankedSalaries
WHERE rn = 1
ORDER BY department, salary DESC;
```

Listing 12: Efficient Solution for iii.1

Note: The efficient solution scans the table once to compute ranks, while the inefficient one typically involves an aggregation and then a join, which can be less performant on large datasets, especially without proper indexing.

3.2 Assign sequential numbers to records

Problem: Provide a unique sequential number for each product sale, ordered by `sale_date` then by `sale_id`. Provide both an inefficient solution (e.g., using a correlated subquery to count preceding records) and an efficient solution using ranking functions.

Solution: *Inefficient Solution (using a correlated subquery):*

```sql
SELECT
    s1.sale\_id,
    s1.product\_name,
    s1.sale\_date,
    (SELECT COUNT(*)
```

```
     FROM product\_sales s2
     WHERE (s2.sale\_date < s1.sale\_date)
        OR (s2.sale\_date = s1.sale\_date AND s2.sale\_id <= s1.sale\
    _id)
    ) as inefficient\_row\_num
FROM product\_sales s1
ORDER BY inefficient\_row\_num;
```

Listing 13: Inefficient Solution for iii.2

Efficient Solution (using ROW_NUMBER()):

```
SELECT
    sale\_id,
    product\_name,
    sale\_date,
    ROW\_NUMBER() OVER (ORDER BY sale\_date ASC, sale\_id ASC) as
    efficient\_row\_num
FROM product\_sales
ORDER BY efficient\_row\_num;
```

Listing 14: Efficient Solution for iii.2

Note: Correlated subqueries like the one in the inefficient solution are often very slow because the subquery is executed for each row of the outer query. `ROW_NUMBER()` is optimized for this task.

3.3 Ranking based on an aggregate

Problem: Rank product categories by their total sales amount in descending order. Provide both an inefficient solution (e.g., aggregating in a subquery/CTE, then using another correlated subquery or complex logic for ranking) and an efficient solution using ranking functions on the aggregated results.

Solution: *Inefficient Solution (aggregating then using a correlated subquery for ranking):*

```
WITH CategorySales AS (
    SELECT
        category,
        SUM(sale\_amount) AS total\_sales
    FROM product\_sales
    GROUP BY category
)
SELECT
    cs.category,
    cs.total\_sales,
    (SELECT COUNT(DISTINCT cs2.total\_sales)
     FROM CategorySales cs2
     WHERE cs2.total\_sales >= cs.total\_sales) AS inefficient\_dense\
    _rank
FROM CategorySales cs
ORDER BY inefficient\_dense\_rank;
```

Listing 15: Inefficient Solution for iii.3

Efficient Solution (aggregating then using a ranking function):

```
WITH CategorySales AS (
    SELECT
```

```sql
        category,
        SUM(sale\_amount) AS total\_sales
    FROM product\_sales
    GROUP BY category
)
SELECT
    category,
    total\_sales,
    DENSE\_RANK() OVER (ORDER BY total\_sales DESC) as efficient\_sales\_rank
FROM CategorySales
ORDER BY efficient\_sales\_rank;
```

Listing 16: Efficient Solution for iii.3

Note: Similar to the previous example, the correlated subquery for ranking in the inefficient solution performs poorly compared to the direct application of a window ranking function on the aggregated results.

4 Practice Hardcore Combined Problem

4.1 Comprehensive Employee Analysis

Problem: For each department:

1. Identify the employee(s) with the highest salary in that department.
2. For these top-paid employees, determine their salary rank across the entire company (use `RANK()` for ties).
3. Show the average salary of their department.
4. Calculate the difference between their salary and their department's average salary.
5. Identify the `employee_id` and full name (`first_name || ' ' || last_name`) of the employee who was hired immediately *after* them within the same department, along with that next hire's date. If they are the last one hired in their department (among the top-paid, or overall if simpler for this part), this should be `NULL` for the next hire's details. (Clarification: "last one hired in their department" should be interpreted as the one with latest hire date in the entire department for identifying the "next hire").

Present a single SQL query that provides all this information.

Solution:

```sql
WITH DepartmentTopSalary AS (
    SELECT
        employee\_id,
        first\_name,
        last\_name,
        department,
        salary,
        hire\_date,
        DENSE\_RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dept\_salary\_rank\_val, -- Renamed to avoid conflict
        AVG(salary) OVER (PARTITION BY department) as avg\_dept\_salary
    FROM employees
),
TopPaidInDepartment AS (
    SELECT
        employee\_id,
        first\_name,
        last\_name,
        department,
        salary,
        hire\_date,
        avg\_dept\_salary,
        RANK() OVER (ORDER BY salary DESC) as overall\_company\_salary\_rank
    FROM DepartmentTopSalary
    WHERE dept\_salary\_rank\_val = 1 -- Used renamed column
),
EmployeeHireSequence AS (
    SELECT
        employee\_id,
        first\_name, -- Added for next\_hire full name
        last\_name,  -- Added for next\_hire full name
        department,
        hire\_date,
        -- This ROW\_NUMBER is for all employees in the department by hire\_date
        -- to correctly identify the "next hire" overall in the department.
        ROW\_NUMBER() OVER (PARTITION BY department ORDER BY hire\_date ASC) as hire\_sequence\_in\_dept
    FROM employees
)
SELECT
    tpid.department,
    tpid.first\_name AS top\_emp\_first\_name,
    tpid.last\_name AS top\_emp\_last\_name,
    tpid.salary AS top\_emp\_salary,
    tpid.overall\_company\_salary\_rank,
    ROUND(tpid.avg\_dept\_salary, 2) AS department\_avg\_salary,
    ROUND(tpid.salary - tpid.avg\_dept\_salary, 2) AS salary\_diff\_from\_dept\_avg,
    tpid.hire\_date AS top\_emp\_hire\_date,
    next\_hire.employee\_id AS next\_hired\_employee\_id,
    next\_hire.first\_name || ' ' || next\_hire.last\_name AS next\_hired\_full\_name,
    next\_hire.hire\_date AS next\_hired\_hire\_date
FROM TopPaidInDepartment tpid
-- Join with EmployeeHireSequence to get the hire\_sequence\_in\_dept for the top paid employee
JOIN EmployeeHireSequence current\_emp\_hire\_seq
```

```
    ON tpid.employee\_id = current\_emp\_hire\_seq.employee\_id
-- LEFT JOIN to find the next employee in the sequence within the same department
LEFT JOIN EmployeeHireSequence next\_hire
    ON tpid.department = next\_hire.department -- Ensure same department
    AND current\_emp\_hire\_seq.hire\_sequence\_in\_dept + 1 = next\_hire.hire\_sequence\_in\_dept
ORDER BY
    tpid.department, tpid.salary DESC, tpid.employee\_id;
```

Listing 17: Solution for iv.1

Explanation of the Hardcore Problem Solution Logic:

- `DepartmentTopSalary` CTE:
  - Uses `DENSE_RANK()` partitioned by `department` and ordered by `salary DESC` to find the rank of each employee's salary within their department.
  - Uses `AVG(salary)` as a window aggregate function partitioned by `department` to calculate the average salary for each department and associate it with every employee in that department.
- `TopPaidInDepartment` CTE:
  - Filters `DepartmentTopSalary` for employees where `dept_salary_rank_val = 1`, effectively selecting the highest-paid employee(s) in each department.
  - Calculates `overall_company_salary_rank` for these top-paid employees using `RANK()` ordered by `salary DESC` across the entire company.
- `EmployeeHireSequence` CTE:
  - Assigns a `ROW_NUMBER()` to each employee within their department based on their `hire_date` (earliest first). This sequence is crucial for identifying the "next hired" employee.
- Final `SELECT` Statement:
  - Starts with the `TopPaidInDepartment` employees.
  - Joins with `EmployeeHireSequence` (aliased as `current_emp_hire_seq`) on `employee_id` to get the `hire_sequence_in_dept` of the current top-paid employee.
  - `LEFT JOIN`s again with `EmployeeHireSequence` (aliased as `next_hire`) on two conditions:
    1. Same `department`.
    2. The `hire_sequence_in_dept` of `next_hire` is exactly one greater than that of `current_emp_hire_seq`. This finds the employee hired immediately after the current top-paid employee in that department.

    A `LEFT JOIN` is used because if a top-paid employee is the last one hired in their department, there will be no "next hire," and their details should be `NULL`.
  - Selects the required fields, including calculated difference and formatting the average salary.
  - Orders the results for clarity.