SQL Ranking Functions:
The Data Deciders! Rank, Order, Partition!

Sequential SQL

1. What Are They?

Meanings & Values

Imagine athletes in a race. You know their finish times (the data rows). An aggregate (like AVG time) tells you the average speed of all runners. But Ranking Functions tell you who came 1st, 2nd, 3rd, and so on! They don't collapse the rows; they just add a new column **to each row** showing its position within a specific order.
Core Idea: Assign a numerical rank or sequential number to rows based on an ordering, often within partitions (groups).
The Deciders:

- `ROW_NUMBER()`: Gives a unique, sequential number to each row starting from 1 within its partition. No ties, no gaps! *Each row gets a number true, 1, 2, 3, all for you!*
- `RANK()`: Assigns a rank based on the ordering. If rows have the same value (a tie), they get the same rank. The next rank *skips* numbers to account for the tied rows. *Ties share a rank, but beware, the next number isn't there!*
- `DENSE_RANK()`: Similar to `RANK()`, ties share the same rank. But `DENSE_RANK()` does *not* skip numbers after a tie. The next rank is just the next integer. *Ties get the same, it's true, but the next rank's right after you!*

Their Value[a]**:** A new column of integers that tells you the relative position of each row based on criteria you define. Essential for finding "top N" items within categories, pagination, or analyzing sequences.

[a] After this lecture read the documentation to know entirely about these functions

Relations

How They Play with Others

Ranking Functions are cool kids at the party, but they hang out in a specific corner: the **Window Function** corner. They use the 'OVER()' clause, which acts like a magnifying glass on parts of your data without collapsing it like 'GROUP BY'.

Playing Nicely (or Differently) with Prereqs:

- `GROUP BY`: **Different Purpose!** 'GROUP BY' squishes many rows into one summary row per group (e.g., total sales per department). Ranking Functions keep *all* original rows and add a rank *within* a group (or the whole set). You can't use Ranking Functions directly in a 'GROUP BY' or apply 'GROUP BY' *after* ranking easily in the same query level.
- `Aggregates` (`SUM`, `AVG`, etc.): These usually aggregate data per group (`GROUP BY`) or the whole set. While Aggregates *can* be used as Window Functions (calculating a running total or moving average), Ranking Functions are *specifically* for numbering rows, not summing or averaging values.
- `WHERE`: Filters rows **BEFORE** Ranking Functions are applied. If a row is filtered out by 'WHERE', it won't be ranked. The rank applies to the rows that 'WHERE' lets through.
- `HAVING`: Filters **GROUPS** *after* aggregation with 'GROUP BY'. Since Ranking Functions don't use 'GROUP BY' in their core logic (they use 'PARTITION BY' within 'OVER()'), 'HAVING' is not directly relevant to filtering based on the ranks themselves at the same query level. You'd typically filter on the calculated rank in a subquery or CTE.
- `ORDER BY` (Main Query): Determines the final order of the result set shown to you **AFTER** the ranks are calculated. It's independent of the 'ORDER BY' *within* the 'OVER()' clause.
- `OVER(ORDER BY ...)`: **Crucial!** This 'ORDER BY' inside the 'OVER()' clause defines the sequence *within the window/partition* that determines the rank. Without it (especially for `RANK` and `DENSE_RANK`), the ranking is non-deterministic!
- `OVER(PARTITION BY ...)`: Acts like 'GROUP BY' for Window Functions. It divides the rows into partitions (subsets). Ranking is then applied independently **within each partition**. If you omit 'PARTITION BY', the entire result set is treated as one partition.
- `Joins`: Joins combine data from multiple tables **BEFORE** Ranking Functions are applied. Ranking works on the result set produced by the join.
- `NULLs`: 'NULL' values in the column specified in the 'ORDER BY' clause *within* 'OVER()' will affect the ranking order according to the database's

rules for sorting ‘NULL‘s (usually first or last). `COALESCE` can be used *before* ranking if you need to treat ‘NULL‘s differently for ordering purposes.

How to Use Them

Structures & Syntax

Ranking Functions always appear in the `SELECT` list and require the `OVER()` clause. The `OVER()` clause is where you define the window (or partition) and the ordering for the ranking. Practice these in your pgAdmin4 query tool!
The Structure: ‘RANK() OVER ([PARTITION BY column_list] ORDER BY column_list [ASC—DESC], ...)‘ (Syntax is the same for `DENSE_RANK()` and `ROW_NUMBER()`)
Let’s See It in Action:
1. Basic Ranking (on the whole set - no PARTITION BY) Rank employees by salary.
SELECT employee_id, salary, ROW_NUMBER() OVER (ORDER BY salary DESC) AS rn_salary, RANK() OVER (ORDER BY salary DESC) AS rank_salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS DENSE_RANK_salary FROM employees ORDER BY salary DESC; – Order the final output by salary *Note the difference in rn_salary, rank_salary, and dense_rank_salary if there are employees with the same salary!*
2. Ranking Within Partitions (like GROUP BY for Windows) Rank employees by salary *within each department*.
SELECT department_id, employee_id, salary, ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rn_dept_salary, RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank_dept_salary, DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS DENSE_RANK_dept_salary FROM employees ORDER BY department_id, salary DESC; – Order final output *The ranking resets to 1 for each new department!*
3. Using Ranks to Filter (finding Top N per group) Find the top 3 highest-paid employees in each department. You can’t use ‘WHERE rank_dept_salary ¡= 3‘ directly because ranking is evaluated *after* ‘WHERE‘. You need a subquery or CTE.
– Using a Common Table Expression (CTE) WITH RankedEmployees AS (SELECT department_id, employee_id, salary, RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank_dept_salary FROM employees) SELECT department_id, employee_id, salary FROM RankedEmployees WHERE rank_dept_salary ¡= 3 – Now you can filter on the rank! ORDER BY department_id, rank_dept_salary; *Using ‘RANK()‘ here means if there’s a tie for 3rd, you’ll get *all* tied employees. Using ‘ROW_NUMBER()‘ would pick an arbitrary 3 if there’s a tie for 3rd.*

Why Use Them?

Advantages

They make complex tasks simple and efficient compared to older SQL methods using just ‘GROUP BY‘ and subqueries, or relying on client-side processing.
Key Benefits:

- **FIND TOP N PER GROUP:** The classic use case! Easily identify the highest/lowest performing, newest/oldest entries, etc., within categories. Trying to do this with ‘GROUP BY‘ alone is a nightmare of subqueries or variables.
- **EFFICIENT:** The database engine is optimized to calculate these ranks. It’s usually much faster than pulling all data to your application and looping through it, or using less standard database-specific features.
- **CONCISE CODE:** A single function call with ‘OVER()‘ replaces many lines of complex, harder-to-read SQL that would attempt to simulate ranking using joins or correlated subqueries.
- **ANALYTICAL POWER:** They are fundamental building blocks for more advanced analytical queries, letting you see data’s position relative to others.
- **PAGINATION SIMPLIFIED:** `ROW_NUMBER()` is perfect for generating sequence numbers for pagination results.

Want a top list? With Rank functions, your queries persist!

Watch Out!

Disadvantages)

Like any powerful tool, Ranking Functions have their quirks.
Potential Pitfalls:

- **NO ORDER IN OVER():** If you omit ‘ORDER BY‘ within the ‘OVER()‘ clause (or if the ordering is not unique), the ranking can be non-deterministic. Rows that are equal according to the ‘ORDER BY‘ might get ranked in a different order each time you run the query (especially true for `ROW_NUMBER()`). Always include ‘ORDER BY‘ for consistent results.
- **CONFUSION BETWEEN RANK & DENSE_RANK:** Understand how ties are handled! `RANK()` leaves gaps, `DENSE_RANK()` doesn’t. Choose the one that matches your requirement for tie handling.
- **INCORRECT PARTITIONING:** Getting the ‘PARTITION BY‘ wrong means your ranks will be calculated across the wrong groups of data. Double-check your partitioning logic!

- **NULL HANDLING:** 'NULL's in the 'ORDER BY' column affect the sort order and thus the ranks. Know how your specific database sorts 'NULL's (first or last) or use `COALESCE` if you need explicit 'NULL' handling for ranking.
- **NOT FILTERABLE BY WHERE:** Remember you can't use the calculated rank directly in a 'WHERE' clause at the same query level. Use a subquery or CTE. *The rank is born, after WHERE is worn!*

Master the ranks, avoid the hacks!

Master Ranking Functions, Master Positional Analysis!