SQL Window Functions

Version SQL:2023 · PostgreSQL 18 · verified 2026-11-14

Ranking, top N per group, LAG/LEAD, running totals, moving averages and frames — and why the default frame surprises you.

Contents

Mental model 3

The OVER clause 3

Ranking rows 5

Top N per group 6

LAG and LEAD 8

Running totals and frames 9

Moving averages 11

FIRST_VALUE, LAST_VALUE, NTH_VALUE 13

Named windows and FILTER 14

Performance 16
Further reading 16

Mental model

A window function computes a value for each row from a set of related rows — its window — without collapsing them the way `GROUP BY` does. The window is built in three steps: `PARTITION BY` splits the rows into groups, `ORDER BY` sorts each group, and the frame picks which sorted rows the function sees from the current row. Window functions run after `WHERE`, `GROUP BY` and `HAVING`, which is why you cannot filter on them in the same query.

The OVER clause

Clause	Effect
`OVER ()`	Every row sees the whole result
`PARTITION BY col`	Restart the window per group
`ORDER BY col`	Sort rows inside each partition
frame (`ROWS ...`)	Narrow which sorted rows count

`OVER` is what turns a call into a window function. Any ordinary aggregate — `sum`, `avg`, `count`, `max` — becomes

one when you add it, and returns the aggregate of the window beside every row instead of one row per group. That makes “share of the group total” a single query with no self-join.

```
SELECT region, day, amount,
  100.0 * amount / SUM(amount) OVER (
    PARTITION BY region
  ) AS pct_of_region
FROM revenue;
```

> **Gotcha:** Window functions are allowed only in the `SELECT` list and `ORDER BY`. `WHERE rn <= 3` fails — PostgreSQL reports `window functions are not allowed in WHERE` — because `WHERE` runs before the windows exist. Filter in an outer query.

Ranking rows

Function	Ties 100, 100, 90 give
`ROW_NUMBER()`	1, 2, 3 — arbitrary order inside ties
`RANK()`	1, 1, 3 — gap after a tie
`DENSE_RANK()`	1, 1, 2 — no gap
`NTILE(4)`	Bucket 1–4, sizes as equal as possible
`PERCENT_RANK()`	`(rank - 1) / (rows - 1)`, 0 to 1

Ranking functions number rows in the window's `ORDER BY` order and ignore the frame. Pick by what a tie should mean: `ROW_NUMBER` when you need exactly one row per position, `RANK` for competition-style standings, `DENSE_RANK` when you want "the second-highest value". `NTILE(4)` gives quartile buckets; with 10 rows the bucket sizes are 3, 3, 2, 2.

```
SELECT name, dept, salary,
  ROW_NUMBER() OVER w AS rn,
  RANK()       OVER w AS rnk,
```

```
  DENSE_RANK() OVER w AS dense
FROM employees
WINDOW w AS (
  PARTITION BY dept ORDER BY salary DESC
);
-- Alice 1 1 1 · Bob 2 1 1 · Carol 3 3 2
```

> **Gotcha:** `ROW_NUMBER()` over tied values is not deterministic — Alice and Bob can swap between runs, so "row 1" changes and a paginated report shows a row twice. Add a unique tiebreaker: `ORDER BY salary DESC, id`.

Top N per group

Want	Rank with
Exactly N rows per group	`ROW_NUMBER()`
N, plus anything tied with row N	`RANK()`
The top N distinct values	`DENSE_RANK()`

Number the rows in a CTE or subquery, then filter on the number in the outer query. The same shape deduplicates a table: keep `rn = 1` per natural key, ordered by the newest timestamp, and you have the latest version of each record.

```
WITH ranked AS (
  SELECT name, dept, salary,
    ROW_NUMBER() OVER (
      PARTITION BY dept
      ORDER BY salary DESC, name
    ) AS rn
  FROM employees
)
SELECT name, dept, salary
FROM ranked
WHERE rn <= 3;
```

> **Note:** `QUALIFY rn <= 3` filters window results without the wrapper in some engines (Snowflake, BigQuery, DuckDB). PostgreSQL 18 does not support it; use the CTE.

LAG and LEAD

Call	Returns
`LAG(x)`	`x` from the previous row, else `NULL`
`LEAD(x)`	`x` from the next row, else `NULL`
`LAG(x, 7)`	`x` from 7 rows back
`LAG(x, 1, 0)`	Previous `x`, or `0` on the first row

`LAG` and `LEAD` read another row of the same partition by position, so “change since yesterday” needs no self-join. The first row has no predecessor and gets `NULL` unless you pass a default as the third argument.

```
SELECT day, price,
  price - LAG(price) OVER (ORDER BY day)
```

```
    AS change,
  LEAD(price) OVER (ORDER BY day)
    AS next_price
FROM prices;
-- first row: change is NULL
```

> **Gotcha:** `LAG` counts rows, not calendar days. If a day is missing from the table, "previous row" silently means two days ago. Generate a complete date series (`generate_series` in PostgreSQL) and outer-join to it first when gaps are possible.

Running totals and frames

Frame mode	Unit of `n PRECEDING`
`ROWS`	Physical rows
`RANGE`	Distance in the `ORDER BY` value
`GROUPS`	Peer groups — sets of tied rows

With an `ORDER BY` and no frame, the default frame is `RANGE UNBOUNDED PRECEDING`: from the partition start through the current row's last **peer**, where peers are rows that tie on the `ORDER BY`. That gives a running total, but tied rows all receive the total of the whole tie. Write `ROWS` when you need one distinct value per row. Without `ORDER BY`, every row is a peer, so the frame is the whole partition.

```
SELECT day, amount,
  SUM(amount) OVER (ORDER BY day)
    AS by_range,
  SUM(amount) OVER (
    ORDER BY day ROWS BETWEEN
      UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS by_rows
FROM revenue
WHERE region = 'north';
-- two rows on one day: by_range repeats
```

> **Gotcha:** The default frame is invisible in the query, so the bug is too. A running total looks right in testing and then jumps by two rows at once in production, the first time two events share a timestamp.

Moving averages

Frame	Covers
`ROWS BETWEEN 6 PRECEDING AND CURRENT ROW`	Last 7 rows
`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`	Centred 3 rows
`RANGE BETWEEN '6 days' PRECEDING AND CURRENT ROW`	Last 7 calendar days
`... EXCLUDE CURRENT ROW`	Same frame, minus this row

A frame with a fixed offset slides with the current row, which turns `AVG` into a moving average and `MAX` into a rolling peak. `ROWS` counts rows, so it has the same missing-day problem as `LAG`; in PostgreSQL, `RANGE` with an `interval` offset counts calendar time instead, but needs exactly one `ORDER BY` column.

```
SELECT day, amount,
  AVG(amount) OVER (
    ORDER BY day
    ROWS BETWEEN 6 PRECEDING
      AND CURRENT ROW
  ) AS avg_7
FROM revenue
WHERE region = 'north';
```

> **Gotcha:** The frame stops at the partition edge, so the first six rows average fewer than seven values and look smoother than they are. Hide them, or return `COUNT(*) OVER` the same frame and keep only rows where it reaches 7.

FIRST_VALUE, LAST_VALUE, NTH_VALUE

Function	Reads
`FIRST_VALUE(x)`	First row of the frame
`LAST_VALUE(x)`	Last row of the frame
`NTH_VALUE(x, n)`	Row `n` of the frame, else `NULL`

Unlike ranking functions and `LAG`, these read the **frame**, not the whole partition. `FIRST_VALUE` works with the default frame because the frame always starts at the partition's first row.

```sql
SELECT name, dept, salary,
  LAST_VALUE(name) OVER (
    PARTITION BY dept
    ORDER BY salary DESC
    ROWS BETWEEN UNBOUNDED PRECEDING
      AND UNBOUNDED FOLLOWING
  ) AS lowest_paid
FROM employees;
```

> **Gotcha:** With the default frame, `LAST_VALUE(x)` returns the current row's own value (or its last peer's), because the frame ends at the current row. Spell out `UNBOUNDED FOLLOWING`, or flip the sort and use `FIRST_VALUE`.

Named windows and FILTER

Syntax	Use for
`WINDOW w AS (...)`	Define a window once per query

Syntax	Use for
`OVER w`	Use it unchanged
`OVER (w ROWS ...)`	Extend it with a frame
`SUM(x) FILTER (WHERE ...) OVER`	Aggregate a subset of the window

A `WINDOW` clause names a definition that several calls share, which keeps them consistent and lets the planner sort once. `FILTER` works only on aggregates used as window functions, and feeds them only rows where its condition is true.

```
SELECT day, region, amount,
  SUM(amount) OVER w AS total,
  COUNT(*) FILTER (
    WHERE amount > 100
  ) OVER w AS big_sales
FROM revenue
WINDOW w AS (PARTITION BY region);
```

> **Note:** `OVER (w ...)` copies and modifies `w`, and PostgreSQL rejects it if `w` already has a frame clause. Put the frame in the `OVER`, not in the shared definition.

Performance

Each distinct `PARTITION BY` plus `ORDER BY` combination needs rows in that order, so the planner adds a `Sort` below each `WindowAgg` node in `EXPLAIN`. Windows that share a definition share the sort. A B-tree index on the partition and order columns, such as `(region, day)`, can return rows already sorted, and the planner drops the `Sort` when that is cheaper — typically with a selective filter or a `LIMIT`. Window functions see only rows that survived `WHERE`, so filter as early as you can — see the indexing and EXPLAIN sheet.

Further reading

- PostgreSQL — Window function calls

- PostgreSQL — Window functions reference
- PostgreSQL — Window functions tutorial
- PostgreSQL — The WINDOW clause
- SQLite — Window functions