FREE SAMPLE

A sample from *PostgreSQL Problem Solver*

The Friday 4:45 Fix

Top-N, Running Totals & Period-over-Period — the 3 SQL recipes that save your Friday

3 complete recipes • Tested on PostgreSQL 17

M.Z. Ahmad

leanpub.com/pg-problem-solver

Contents

Read This First (60 Seconds)

Your boss walks over at 4:45 PM on a Friday.

> *"Quick question — what are the top three products in each category by revenue? I need it for the Monday leadership review."*

You know it's possible. You also know `ORDER BY ... LIMIT 3` won't cut it — that gives you three products *total*, not three *per category*. You could write one query per category, but you have a dozen categories and zero desire to spend your evening copy-pasting SQL.

This little book hands you three copy-paste-ready recipes for the three "quick questions" that eat analysts' Friday afternoons:

1. **Top-N per group** — "top 3 products in each category"
2. **Running totals** — "cumulative sales by day"
3. **This period vs. last** — "how did this month compare to last month?"

Each recipe is complete: the problem, the SQL, the exact output, and the mistake everyone makes. Copy the code. Adapt it to your data. Be out the door by 5:00.

Who this is for: analysts who know `SELECT`, `WHERE`, `JOIN`, and `GROUP BY` but hit a wall on medium-hard reporting problems. If that's you, keep reading.

Who this is *not* for: DBAs (nothing here about vacuum or replication) and com-

plete beginners (you need to already know what a JOIN is).

> ***Want to run these yourself?*** *The three-table sample dataset is tiny and free. Loading instructions are in the appendix at the end. But you can also just read the outputs — every result is printed inline.*

Recipe 1 — Top 3 in Each Category

This is the #1 most-Googled data analysis problem. Here it is, solved, in one query.

The Solution (Skip Here If You're in a Hurry)

```
WITH product_totals AS (
    SELECT category_name, product_name, SUM(net_sale) AS total_revenue
    FROM ecomm_sales
    GROUP BY category_name, product_name
),
ranked AS (
    SELECT *,
           DENSE_RANK() OVER (
               PARTITION BY category_name
               ORDER BY total_revenue DESC
           ) AS rank
    FROM product_totals
)
SELECT * FROM ranked WHERE rank <= 3 ORDER BY category_name, rank;
```

Three steps: aggregate, rank *within each category*, filter to the top 3. That's it. Now let's make sure you understand it well enough to adapt it — because next Friday the question will be "top 5 reps per region," and you'll want to change one line, not start over.

The One Idea That Unlocks This: PARTITION BY

A `GROUP BY` collapses rows — you lose the detail. A **window function** partitions the data into sets *just like* `GROUP BY`, but keeps every row intact. That's exactly what "top N per group" needs: rank inside each group without throwing rows away.

There are three ranking functions. The difference is how they treat ties:

Function	Behavior with ties	Example
`ROW_NUMBER()`	Never ties; unique sequential numbers	1, 2, 3, 4
`RANK()`	Ties share a rank; next rank **skips**	1, 1, 3, 4
`DENSE_RANK()`	Ties share a rank; next rank does **not** skip	1, 1, 2, 3

Memory trick: **RANK** rhymes with **GAP** — it skips numbers after a tie. **DENSE_RANK** is *dense* — no gaps. **ROW_NUMBER** gives every row its own number, period.

For "top N per group," `DENSE_RANK()` usually gives the most intuitive result. If two products tie for first, you probably want both, plus the next product — not a gap.

Building It in Three Steps

Step 1 — Aggregate sales per product. No window function yet, just a plain total wrapped in a CTE (a Common Table Expression — a temporary named table that exists only for this query):

```
WITH product_totals AS (
    SELECT category_name, product_name, SUM(net_sale) AS total_revenue
    FROM ecomm_sales
    GROUP BY category_name, product_name
)
SELECT * FROM product_totals ORDER BY category_name, total_revenue DESC;
```

Step 2 — Rank within each category. Now the window function does the work:

```
DENSE_RANK() OVER (
    PARTITION BY category_name   -- rank separately inside each category
    ORDER BY total_revenue DESC  -- higher revenue = rank 1
) AS rank
```

Output:

category_name	product_name	total_revenue	rank
Books	SQL for Data Analysis	239.94	1

category_name	product_name	total_revenue	rank
Books	The Data Warehouse Toolkit	194.96	2
Books	Fundamentals of Data Engineering	134.97	3
Books	Storytelling with Data	104.97	4
Clothing	Waterproof Rain Jacket	264.97	1
Clothing	Running Shorts 5-Pack	199.95	2
Clothing	Denim Jacket Classic	179.97	3

Step 3 — Filter to the top 3. Here's the trap: **window functions can't go in a `WHERE` clause.** That's the #2 mistake people make with them (right after forgetting `PARTITION BY`). The fix is to wrap the ranking query in another CTE, then filter:

```
SELECT * FROM ranked WHERE rank <= 3 ORDER BY category_name, rank;
```

And there it is. Top 3 per category, one query, no copy-pasting.

> ***PRO TIP*** *— When two products tie for first, `DENSE_RANK()` returns* four *rows for a "top 3." That's usually what you want. But if your stakeholder insists on exactly 3, swap in `ROW_NUMBER()`, which breaks ties arbitrarily. Better yet, tell them: "I'm showing 4 because 2 tied for first — want me to trim it?" That 30-second conversation builds trust. Silently hiding ties erodes it.*

Adapt It in One Line

- **Bottom 3 (underperformers to cut):** change `ORDER BY total_revenue DESC` to `ASC`.
- **Top 5 instead of 3:** change `WHERE rank <= 3` to `<= 5`.
- **Top 3 per region / per rep / per month:** change `PARTITION BY category_name` to `PARTITION BY region` (or `sales_rep`, or a truncated date).

One idea — partition, order, filter — answers a whole family of Friday questions.

Recipe 2 — Cumulative Sales by Day (Running Totals)

"Show me a running total of revenue so I can see when we crossed $10K."

A running total is the same partition-and-order idea, minus the ranking. You sum

a value **ordered by date**, and the window quietly accumulates:

```
WITH daily_sales AS (
    SELECT sale_date, SUM(net_sale) AS daily_revenue
    FROM ecomm_sales
    GROUP BY sale_date
)
SELECT
    sale_date,
    daily_revenue,
    SUM(daily_revenue) OVER (ORDER BY sale_date) AS cumulative_revenue
FROM daily_sales
ORDER BY sale_date;
```

Output (first rows):

sale_date	daily_revenue	cumulative_revenue
2024-01-05	159.98	159.98
2024-01-08	39.99	199.97
2024-01-10	79.99	279.96
2024-01-12	34.99	314.95
2024-01-15	89.98	404.93

Why does it accumulate? Because the moment you add `ORDER BY` to `SUM() OVER (...)`, PostgreSQL applies a default *frame*: `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. In plain English: "sum everything from the start of the partition up to and including this row." That's a running total.

> ***PRO TIP*** *— `ORDER BY` inside a window does two jobs at once: it sets the sort order* and *switches on the cumulative frame. Drop the `ORDER BY` and you get the grand total repeated on every row instead of a running total. If your "running total" looks suspiciously flat, that's why.*

Adapt it: want a 7-day moving average instead of an ever-growing total? Same query, but add an explicit frame — `AVG(daily_revenue) OVER (ORDER BY sale_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)`. The full book walks through moving averages, year-to-date, and quarter-to-date totals step by step.

Recipe 3 — This Month vs. Last Month

"How did this month's revenue compare to last month's? And give me the percentage change."

To compare a row to the row before it, you don't need a self-join. You need `LAG()` — it reaches back to a previous row in the same window:

```
WITH monthly AS (
    SELECT DATE_TRUNC('month', sale_date) AS month,
           SUM(net_sale) AS monthly_revenue
    FROM ecomm_sales
    GROUP BY DATE_TRUNC('month', sale_date)
)
SELECT
    month,
    monthly_revenue,
    LAG(monthly_revenue, 1) OVER (ORDER BY month) AS prev_month_revenue
FROM monthly
ORDER BY month;
```

Output:

month	monthly_revenue	prev_month_revenue
2024-01	1129.84	*(null)*
2024-02	1282.80	1129.84
2024-03	1654.78	1282.80
2024-04	780.83	1654.78
2024-05	974.87	780.83

The first row is `NULL` because there's no month before it — that's expected, not a bug.

Once last month's value sits on the same row, the percentage change is just arithmetic. Compute `LAG()` once in a CTE so you don't repeat yourself:

```
WITH monthly AS (
    SELECT DATE_TRUNC('month', sale_date) AS month,
           SUM(net_sale) AS monthly_revenue
```

```
    FROM ecomm_sales
    GROUP BY DATE_TRUNC('month', sale_date)
),
with_prev AS (
    SELECT month, monthly_revenue,
           LAG(monthly_revenue, 1) OVER (ORDER BY month) AS
           prev_month_revenue
    FROM monthly
)
SELECT
    month,
    monthly_revenue,
    prev_month_revenue,
    monthly_revenue - prev_month_revenue AS absolute_change,
    ROUND((monthly_revenue - prev_month_revenue) / prev_month_revenue *
    100, 1) AS pct_change
FROM with_prev
ORDER BY month;
```

Adapt it: `LAG(x, 12)` compares to 12 rows back — that's year-over-year on monthly data. `LEAD()` looks *forward* instead of back. The full book covers year-over-year with multiple lags and how to survive missing months (which quietly break naive period-over-period math).

> ***WATCH OUT*** *— That `pct_change` formula divides by `prev_month_revenue`. If any previous value is `0`, you get a division-by-zero error that kills the whole query. The book's Chapter 4 shows the one-line `NULLIF` guard that makes this safe in production.*

You Just Learned the Pattern. Here's the Map.

Notice that all three recipes are the *same shape*: **partition, order, frame.** Once that clicks, window functions stop being mysterious. That's the entire mental model behind the full book — and it scales to problems that look nothing alike on the surface:

If your boss asks…	The book chapter	The tool
Top / bottom N per group	Ch 2 □ *(you have it)*	`DENSE_RANK`, `PARTITION BY`
Running totals, moving averages	Ch 3 □ *(you have it)*	window frames
This period vs. last, % change, YoY	Ch 4 □ *(you have it)*	`LAG`, `LEAD`
Each customer's first & last purchase	Ch 5	`FIRST_VALUE`, `LAST_VALUE`
Split customers into spending tiers	Ch 6	`NTILE`
Cohort retention matrix	Ch 7	conditional aggregation
Where users drop off in a funnel	Ch 8	`FILTER`, `BOOL_OR`
RFM customer segmentation	Ch 9	`NTILE` + `CASE`
Reorder alerts & dead stock	Ch 10	running totals + `NTILE`
Fill gaps in time-series data	Ch 11	`generate_series`
"Find products like this one"	Ch 12–14	pgvector similarity search
Make the slow query fast	Ch 15	indexing & `EXPLAIN`

Seventeen chapters. Every one follows the same format you just used: **the problem, the SQL, the output, the mistakes, the adaptations.** No theory dumps. No cover-to-cover reading. Find your problem, steal the code, get back to work.

Get the Full Book

PostgreSQL Problem Solver: Stop Googling. Start Solving. *Advanced SQL Recipes for Ranking, Retention, Funnels, Segmentation, and Vector Similarity with pgvector.*

- 17 problem-first chapters + 5 appendices (cheat sheet, dataset schemas, troubleshooting, setup)
- Every query tested against PostgreSQL 17
- The three sample datasets so you can run everything yourself
- A SQL cheat sheet you'll keep open in a second monitor

Keep the momentum — get the full book

On Leanpub you name your own price above the minimum, and every purchase is backed by a 60-day, one-click, 100% refund. Zero risk. Get all 17 chapters, keep your weekends.

Get the full book leanpub.com/pg-problem-solver

Appendix — Run These Queries Yourself in 5 Minutes

Every query above runs against one table: `ecomm_sales`. Here's the shape of it so you can point the recipes at your own data — or load the free sample dataset from the companion site.

Table: `ecomm_sales` (Amazon-style retail transactions, 79 rows, Jan–Jun 2024)

Column	Type	Meaning
`sale_id`	`SERIAL`	Unique transaction ID
`sale_date`	`DATE`	Date of sale
`category_name`	`VARCHAR`	Electronics, Books, Home & Kitchen, Sports & Outdoors, Clothing
`product_name`	`VARCHAR`	Product name
`customer_id`	`INTEGER`	Customer identifier (1001–1010)
`customer_state`	`VARCHAR(2)`	US state code
`quantity`	`INTEGER`	Units purchased
`unit_price`	`DECIMAL(10,2)`	Price per unit
`discount`	`DECIMAL(4,2)`	Discount applied
`net_sale`	`DECIMAL(10,2)`	Total after discount

To run everything in this sample, you need **PostgreSQL 17** and this one table loaded. The full book's Chapter 1 + Appendix E get you from zero to loaded in

about 15 minutes, including the other two datasets (SaaS events and a product catalog) used in the later chapters.

Already have Postgres running? Point these three recipes at any sales table with a date, a category, a product, and an amount. Change the column names, keep the pattern.

Enjoyed this? The full book turns "I'll figure it out eventually" into "I already have the query." **leanpub.com/pg-problem-solver**