SQL Window Functions In Five Easy Steps

David Stokes
DBeaver.com
https://www.slideshare.net/davidmstokes

Description

Structured Query Language Window Functions are a powerful tool for analytics.

They let you get more granular insight than a GROUP BY clause. But the syntax is obtuse, the terms used are nebulous (unbounded previous anyone?), and the results can be much less insightful than expected.

This session is a quick introduction to and explanation of how to use Window Functions efficiently to better investigate your data.

DBeaver

Big Revelation #1

Structured Query Language is NOT everyone's favorite

Structured Query Language has been around since the 1970s mainly because of it's powerful way it deals with data.

The relational model is fantastic for storing and retrieving data.

But it has query syntax …

… wrapping that syntax within a transaction can be very hard …

… and SQL gives back what you ask from it.

Analytics in SQL can be frustrating

There are a limited number of aggregate ‘built in functions’ like SUM() and AVG()

Grouping data can be frustrating

Why They Exist

- Window functions provide granular insights that `GROUP BY` can’t match.
- The syntax is complex but manageable with practice and clear examples.

Practicing Perfectly Brings Perfection:

- Practicing with sample datasets like DVDRENTAL for PostgreSQL or Sakila & World for MySQL/MariaDB
- Resources:Documentation, SQL blogs, or tutorials

Window functions perform calculations across a set of rows (a "window") related to the current row, without collapsing the result set like `GROUP BY`.

- Unlike `GROUP BY`, which aggregates rows into a single output, window functions retain individual rows while adding calculated values (e.g., running totals, rankings).

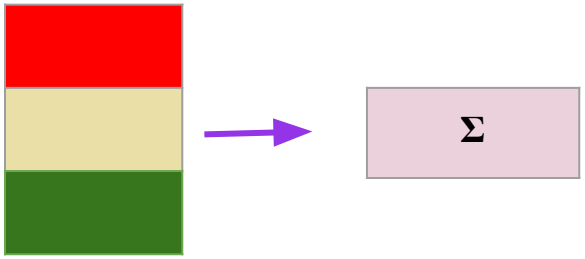
Syntax: **FUNCTION() OVER (PARTITION BY column ORDER BY column)**.

Why They Matter

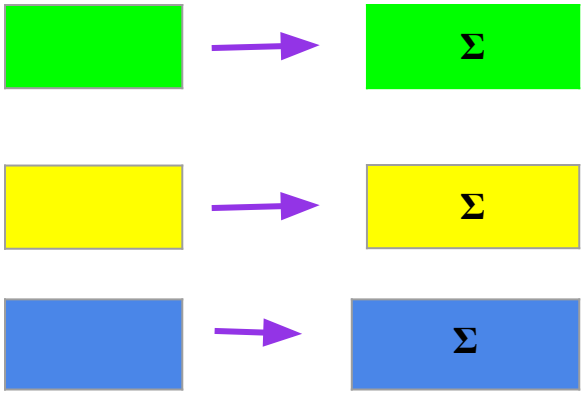
- Enable granular analytics (e.g., compare a row to its group's average without losing row-level detail).
- Ideal for ranking, running totals, moving averages, and year-over-year comparisons.
- Comparison to GROUP BY
  - `GROUP BY` reduces rows; window functions preserve them.

Example: Calculate total sales per region vs. sales contribution of each order to its region's total.

You Use Aggregates On A Table, Window Functions on Groups of Rows



Aggregate



Window Function

```sql
SELECT
    order_id,
    region,
    sale_amount,
    SUM(sale_amount) OVER (PARTITION BY region) AS total_region_sales,
     (sale_amount * 100.0 / SUM(sale_amount) OVER (PARTITION BY region))
AS order_contribution_percentage
FROM
    orders
ORDER BY
    region, order_id;
```

```
SELECT
    employee_id,
    department,
    salary,
    SUM(salary) OVER (PARTITION BY department) AS total_dept_salary
FROM employees;
```

Explanation: Adds a column showing the total salary for each department, keeping each employee's row intact.

```
window_function_name(arguments) OVER (
    [PARTITION BY partition_expression [, ...]]
    [ORDER BY sort_expression [ASC | DESC | USING operator] [NULLS {FIRST | LAST}] [, ...]]
    [frame_clause]
)
```

```
SELECT
   employee_id,
   department,
   salary,
   RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank_in_department,
   SUM(salary) OVER (PARTITION BY department ORDER BY salary ASC ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS running_total_salary
FROM
   employees;
```

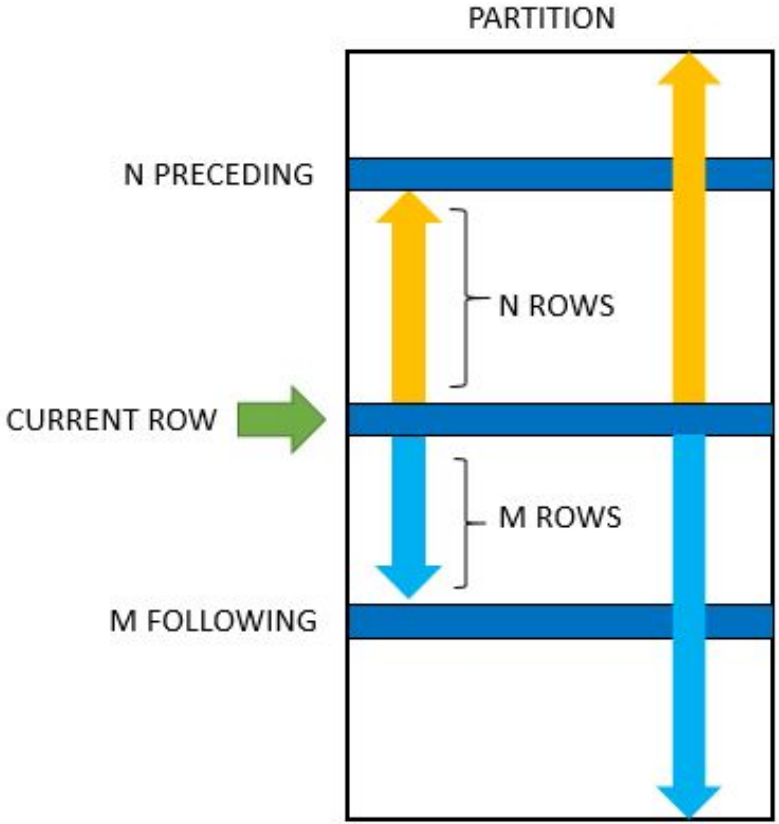
- Function:
  - Aggregate (e.g., `SUM`, `AVG`, `COUNT`)
  - Ranking (e.g., `RANK`, `ROW_NUMBER`)
  - Value functions (e.g., `LAG`, `LEAD`).

- OVER Clause:
  - Defines the window.

- PARTITION BY:
  - Groups rows into partitions (like `GROUP BY`, but rows remain).

- ORDER BY:
  - Defines row order within the window, critical for running totals or rankings.

- Frame Specification:
  - Defines the subset of rows in the window for calculation (e.g., `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).



UNBOUNDED PRECEDING: From the first row of the partition to the current row (or another specified row).

CURRENT ROW: The row being processed.

UNBOUNDED FOLLOWING: To the last row of the partition.

ROWS vs. RANGE:

- `ROWS` counts physical rows;
- `RANGE` considers value ranges (e.g., for dates or numbers)

```sql
SELECT
    order_id,
    order_date,
    sales,
    SUM(sales) OVER (
        PARTITION BY EXTRACT(YEAR FROM order_date)
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM orders;
```

UNBOUNDED PRECEDING AND CURRENT ROW means the sum includes all rows from the start of the year to the current row.

- Aggregate Functions:
  - SUM`, `AVG`, `COUNT`, `MAX`, `MIN`.
- Ranking Functions:
  - `ROW_NUMBER`, `RANK`, `DENSE_RANK`, `NTILE`.
- Value Functions:
  - `LAG`, `LEAD`, `FIRST_VALUE`, `LAST_VALUE`, `NTH_VALUE`.
- Analytic Functions:
  - `CUME_DIST`, `PERCENT_RANK`, `PERCENTILE_CONT`.

```
SELECT
  employee_id,
  department,
  salary,
  RANK() OVER (PARTITION BY department ORDER BY salary DESC)
AS salary_rank
     FROM employees;
```

```
SELECT
  order_id,
  customer_id,
  sales,
  SUM(sales) OVER (
             PARTITION BY customer_id
             ORDER BY order_date
             ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
         ) AS cumulative_sales
  FROM orders;
```

The previous sale for each customer, useful for calculating differences.

```
SELECT
  order_id,
  order_date,
  sales,
  LAG(sales) OVER (PARTITION BY customer_id ORDER BY order_date)
    AS previous_sale
  FROM orders;
```

- Misunderstanding Frame Specification:
  - Problem: Forgetting `ORDER BY` in running totals leads to summing the entire partition.
  - Solution: Always include `ORDER BY` for sequential calculations and specify the frame (e.g., `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`).
- Performance Issues:
  - Problem: Large partitions or poorly defined windows can slow queries.
  - Solution: Use indexes on `PARTITION BY` and `ORDER BY` columns; limit partitions with `WHERE` clauses.
- Confusing Results:
  - Problem: `RANK` vs. `DENSE_RANK` or `ROWS` vs. `RANGE` confusion.
  - Solution: Test with small datasets and verify results step-by-step.
- Obtuse Syntax:
  - Problem: Syntax like `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` is hard to parse.
  - Solution: Break down queries into smaller parts and use CTEs for readability.

- Start with simple windows (e.g., `PARTITION BY` without `ORDER BY`) and build complexity.
- Use meaningful aliases (e.g., `running_total` instead of `sum`).
- Test with `EXPLAIN ANALYZE` to check query performance.
- Visualize the window with small datasets to confirm logic.

```
CREATE INDEX idx_orders_customer_date
ON orders (customer_id, order_date);

SELECT
    order_id,
    customer_id,
    sales,
    SUM(sales) OVER (
        PARTITION BY customer_id
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_sales
FROM orders
WHERE order_date >= '2025-01-01';
```

4. Practice! Practice! Practice!

Window Functions Are *NOT* Easy!

You do need to work with them and use them to become competent with their use.

```
SELECT
  order_date,
  sales,
  AVG(sales) OVER (
  PARTITION BY EXTRACT(MONTH FROM order_date)
  ORDER BY order_date
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
  ) AS moving_avg
FROM orders;
```

```
SELECT
  order_date,
  sales,
  LAG(sales) OVER (
   PARTITION BY EXTRACT(MONTH FROM order_date), EXTRACT(DAY
FROM order_date)
   ORDER BY EXTRACT(YEAR FROM order_date)
  ) AS last_year_sales
FROM orders;
```

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

```
WITH sales_data AS (
 SELECT
   order_id,
   customer_id,
   sales,
   SUM(sales) OVER (PARTITION BY customer_id) AS total_sales
      FROM orders
  )
  SELECT
      order_id,
      customer_id,
      sales,
      (sales::float / total_sales) * 100 AS percent_of_total
  FROM sales_data;
```

5. Be creative

Just asking you to be more creative will not make you so. Buit look for common requested information and develop your Window Functions

Key Takeaways:

- Window functions provide granular insights that `GROUP BY` can’t match.
- The syntax is complex but manageable with practice and clear examples

DBeaver PRO AI Capabilities Out of the Box

DBeaver Lite

- AI chat panel
- AI Query Explanation
- Explain and Fix SQL Code
- Smart Metadata Descriptions
- AI Query Suggestions
- AI Command

DBeaver Enterprise

- AI chat panel
- AI Query Explanation
- Explain and Fix SQL Code
- Smart Metadata Descriptions
- AI Query Suggestions
- AI Command

DBeaver Ultimate

- AI chat panel
- AI Query Explanation
- Explain and Fix SQL Code
- Smart Metadata Descriptions
- AI Query Suggestions
- AI Command

Team Edition

- AI chat panel
- AI Query Explanation
- Explain and Fix SQL Code
- Smart Metadata Descriptions
- AI Query Suggestions
- AI Command
- AI Assistant

CloudBeaver Enterprise

- AI Command
- AI Assistant

Supported AI providers:

- OpenAI
- Azure OpenAI
- Google Gemini
- Ollama
- GitHub Copilot

Two week free evals

Q/A

https://www.slideshare.net/davidmstokes

david.stokes@DBeaver.com