

SQL Aggregate Functions: The Data Crunchers!

Summarize, Analyze, Conquer! Sequential SQL

1. What Are They? (Meanings & Values)

Imagine turning piles of paper into a single report summary. That's aggregates! They take **many rows**, perform a calculation, and give back just **one value**. Many rows, single show!

The Stars:

- **COUNT()**: How many? Or how many non-null?
- **SUM()**: Total up the numbers.
- **AVG()**: Find the middle number (mean).
- **MIN()**: Smallest value.
- **MAX()**: Largest value.

PostgreSQL Power-ups: `STRING_AGG`, `MODE`, `VARIANCE`, `STDDEV`, `PERCENTILE_CONT` (for things like median!). Their **Value** is a single piece of info summarizing the group or the whole set.

2. Relations: How They Play with Others

Aggregates don't work alone; they join the party with other clauses you know and love!

- **WHERE:** Filters rows **BEFORE** aggregation. The gatekeeper for individual rows. Aggregates never live here!
- **GROUP BY:** Your aggregate's **best friend**. It splits rows into bunches. Aggregates run **ON EACH BUNCH**. The rule? If a column isn't aggregated in **SELECT**, it **MUST** be in **GROUP BY**. Don't break the code of conduct!
- **HAVING:** Filters **GROUPS AFTER** aggregation. Aggregates live happily here! Use it for conditions on `SUM()` > 1000 or `COUNT(*)` < 5.
- **NULLs:** Aggregates usually **ignore NULLs**. Especially `AVG()` and `COUNT(column_name)`. If NULL means zero, use `COALESCE` first! Watch out for the quiet NULL!
- **Window Functions (`OVER()`, `PARTITION BY`, `ORDER BY`):** The next level! Aggregates run **PER ROW** based on a window. `PARTITION BY` is like a `GROUP BY` that **doesn't collapse rows**. `ORDER BY` sets the sequence within the window (hello, running totals!). Ranking functions (`RANK`, `DENSE_RANK`, `ROW_NUMBER`) are cool kids in this club too! See the forest, keep the trees!

3. How to Use Them: Structures & Syntax

Different functions, different places! Let's see how they look in action. Practice these in your pgAdmin4 query tool!

Simple Aggregates (COUNT, SUM, MIN, MAX)

- **On the Whole Set:** Summarize everything selected by FROM and WHERE. One row out!

```
SELECT
    COUNT(*) AS total_rows,
    SUM(salary) AS total_salary
FROM employees
WHERE hire_date >= '2020-01-01';
```

- **With GROUP BY:** Summarize for each distinct group. Many rows out (one per group)!

```
SELECT
    department_id,
    AVG(salary) AS avg_dept_salary -- AVG is here, but structure applies
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5; -- Filter groups using aggregate
```

- **As a Window Function (OVER):** Calculate aggregate per row, based on a window (partition). All original rows out, with a new aggregate column!

```
SELECT
    employee_id,
    salary,
    department_id,
    SUM(salary) OVER (PARTITION BY department_id) AS dept_total_salary,
    MIN(salary) OVER (ORDER BY hire_date) AS lowest_salary_to_date
FROM employees;
```

Basic Statistics (AVG, VARIANCE, STDDEV, MODE, PERCENTILE_CONT)

- **On the Whole Set:** Calculations over the full result set.

```
SELECT
    AVG(performance_rating) AS overall_avg_rating,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary) AS median_salary
FROM employees;
```

- **With GROUP BY:** Calculations per group. MODE and PERCENTILE_CONT use the special WITHIN GROUP (ORDER BY ...) syntax here.

```
SELECT
    department_id,
    AVG(performance_rating) AS avg_dept_rating,
    MODE() WITHIN GROUP (ORDER BY performance_rating) AS dept_mode_rating
FROM employees
GROUP BY department_id;
```

- **As a Window Function (OVER):** Per-row statistics over a window. Note: MODE and PERCENTILE_CONT are generally **not** used in the standard OVER(PARTITION BY ... ORDER BY ...) format like AVG or SUM.

```

SELECT
    employee_id,
    salary,
    department_id,
    AVG(salary) OVER (PARTITION BY department_id) AS dept_avg_salary,
    STDDEV(salary) OVER (PARTITION BY department_id) AS dept_salary_stddev
FROM employees;

```

Alphanumeric (STRING_AGG)

- **On the Whole Set:** Concatenate strings from all relevant rows into one (less common solo).

```

SELECT
    STRING_AGG(first_name, ', ') AS all_first_names
FROM employees
WHERE department_id = 1;

```

- **With GROUP BY:** Concatenate strings within each group. Ordering inside **STRING_AGG** is key for predictable results!

```

SELECT
    d.department_name,
    STRING_AGG(e.last_name, ' | ' ORDER BY e.last_name) AS dept_last_names
FROM departments d JOIN employees e ON d.department_id = e.department_id
GROUP BY d.department_name;

```

- **As a Window Function (OVER):** Per-row cumulative string concatenation over a window (requires **ORDER BY** in the window frame).

just for SQL Server, variation for Oracle and DB2
with LISTAGG(). NOT POSSIBLE with PostgreSQL

```

SELECT
    employee_id,
    first_name,
    hire_date,
    STRING_AGG(first_name, ', ' ORDER BY hire_date)
        OVER (ORDER BY hire_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS names_hired_up_to_this_date
FROM employees
ORDER BY hire_date;

```

Run these examples in pgAdmin4 to see the structures in action!

4. Why Use Them? (Advantages)

They make your SQL sing!

- **SUPER EFFICIENT:** The database engine is built for this. Much faster than pulling all data to an app and doing it there. Saves you from writing those loop-de-loops!

- **SIMPLE & CONCISE:** One function call beats a lot of manual coding.
- **DATA SHINKERS:** Turn massive tables into small, useful summaries.
- **REPORTING GOLD:** Get key insights quickly (totals, averages) for decision-makers.
- **ANALYTICAL POWER:** With `OVER()`, perform complex per-row analysis easily.

5. Watch Out! (Disadvantages)

Even stars have shadows!

- **LOSE DETAIL:** The big one. Without `OVER()`, you see the summary, not the individual rows that made it. Collapsing rows can be a tough ask for those details!
- **EASY TO MISUSE:** Confusing `WHERE` and `HAVING`, or messing up `GROUP BY` rules are common trips.
- **NULL SURPRISES:** They ignore `NULLs` by default. Make sure you understand what that means for your data's story.

Master Aggregates, Master Your Data Summaries!