Chapter 2 — Advanced SQL

Declarative

Mr. Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Course materials ⇒ al-moualime.com)

University of Skikda — Department of Computer Science

1^{st} Year Master RSD/AI

Advanced Databases (ADB)

Octobre 5, 2025

Version 1.0 (Initial) — 2025-10-25 à 07:55:49

Plan

Introduction

- The fundamental SQL commands (DDL and DML) were **reviewed** in Chapter 1 of the course and **practiced** in Lab 1. They cover the basic operations of a relational DBMS, here **PostgreSQL**.
- **Objective of this chapter :** go beyond declarative SQL to explore the **advanced and procedural** mechanisms of the standardized language (standard **SQL/PSM — Persistent Stored Modules**).
- **Focus areas :**
  - **Complex queries :** multiple joins, correlated subqueries, views, aggregations, and analytic functions.
  - **Procedural programming :** stored procedures, functions, variables, and control structures.
  - **Automation and robustness :** triggers, transactions, and error handling.

Goal

Master a complete SQL, both **declarative and procedural**, to develop reliable processes compliant with international standards.

References

- **SQL Standard (ISO/IEC 9075 :2023) :** *Information technology — Database languages — SQL* https://www.iso.org/standard/76583.html
- **PostgreSQL (official documentation) :** https://www.postgresql.org/docs/
- **Course bibliography :** established according to the **official Master 1 RSD/AI program** ; it constitutes the theoretical and pedagogical basis of this chapter.

Important

- The **practical examples** rely on **PostgreSQL**, an **open-source** DBMS compliant with the **SQL standard (ISO/IEC 9075)** and suitable for academic contexts.
- Other DBMSs (**Oracle**, **SQL Server**, **MySQL**) are cited to highlight **non-standard deviations** or **implementation specifics**.

ORDER BY Clause

The `ORDER BY` clause sorts the result set by one or more columns. According to the SQL standard, sorting occurs **at the end of logical processing**, after `WHERE`, `GROUP BY`, and `HAVING`.

```
-- Alphabetical sort (ascending by default)
SELECT name, salary FROM Employees
ORDER BY name;

-- Descending by salary
SELECT name, salary FROM Employees
ORDER BY salary DESC;

-- Sort by column ordinal (1 = name, 2 = salary)
SELECT name, salary FROM Employees
ORDER BY 2 DESC, 1 ASC;
```

- The standard allows either the **column name** or its **ordinal position**.
- Ascending (`ASC`) is implicit ; descending (`DESC`) must be explicit.
- Using ordinals can harm readability.

FETCH FIRST / LIMIT Clause

To restrict the number of returned rows, **SQL :2008** introduced `FETCH FIRST n ROWS ONLY`. PostgreSQL implements this standard syntax while keeping the historical `LIMIT` (non-standard extension).

```
-- Standard syntax (SQL:2008)
SELECT name, salary
FROM Employees
ORDER BY salary DESC
FETCH FIRST 5 ROWS ONLY;

-- PostgreSQL equivalent (non-standard extension)
SELECT name, salary
FROM Employees
ORDER BY salary DESC
LIMIT 5;
```

- PostgreSQL supports both **FETCH FIRST** (standard) and **LIMIT** (extension).
- `FETCH FIRST` is not universally available across DBMSs.
- **Recommendation :** prefer the standard form for **portability**.

OFFSET / FETCH Clause

For **paged** results, **SQL :2008** introduced the combination `OFFSET` + `FETCH FIRST`. PostgreSQL provides a **standards-compliant** implementation, while also supporting the historical `LIMIT ... OFFSET ...` variant for backward compatibility.

```sql
-- Standard pagination (page 2: rows 6-10)
SELECT name, salary FROM Employees
ORDER BY salary DESC
OFFSET 5 ROWS
FETCH FIRST 5 ROWS ONLY;

-- PostgreSQL variant (non-standard extension, equivalent)
SELECT name, salary FROM Employees
ORDER BY salary DESC
LIMIT 5 OFFSET 5;
```

- `OFFSET` sets the **initial displacement**.
- `FETCH FIRST` specifies the **maximum row count** after the offset.
- PostgreSQL follows the **SQL :2008** standard semantics.

Why use joins ?

Combining tables without an association condition yields a **Cartesian product** : each row of the first table is paired with every row of the second.

```
-- Cartesian product: 10 employees × 5 departments → 50 rows
SELECT e.name, d.name
FROM Employees e, Departments d;
```

This result is rarely useful. To establish logical correspondence between tables, use a **join**.

Inner join (INNER JOIN)

An **inner join** (`INNER JOIN`) returns only rows where the join condition is satisfied on both sides.

```
SELECT e.name, d.name
FROM Employees e
JOIN Departments d ON e.dep_id = d.dep_id;
```

The keyword `INNER` is optional. An inner join corresponds to the logical intersection of matching rows.

Outer joins (OUTER JOIN)

An **outer join** also keeps rows that do not find a match on the other side, yielding a complete result.

```sql
-- LEFT JOIN: all employees, even without a department
SELECT e.name, d.name
FROM Employees e
LEFT JOIN Departments d ON e.dep_id = d.dep_id;

-- RIGHT JOIN: all departments, even without employees
SELECT e.name, d.name
FROM Employees e
RIGHT JOIN Departments d ON e.dep_id = d.dep_id;

-- FULL OUTER JOIN: all rows from both sides
SELECT e.name, d.name
FROM Employees e
FULL JOIN Departments d ON e.dep_id = d.dep_id;
```

These forms (`LEFT`, `RIGHT`, `FULL`) are part of **SQL-92** and are fully supported by **PostgreSQL**.

Self joins (SELF JOIN)

A **self join** relates a table to itself, useful for hierarchical or dependency relations within one table.

```
SELECT e1.name AS employee, e2.name AS manager
FROM Employees e1
JOIN Employees e2 ON e1.manager_id = e2.emp_id;
```

Aliases (e1, e2) are necessary to distinguish the two instances.

Best practices and common mistakes

Avoid patterns that harm readability and portability. Prefer standard-compliant, maintainable SQL.

```sql
-- Obsolete practice (SQL-89): implicit join via commas
SELECT e.name, d.name
FROM Employees e, Departments d
WHERE e.dep_id = d.dep_id;

-- Good practice: explicit join (SQL-92)
SELECT e.name, d.name
FROM Employees e
JOIN Departments d ON e.dep_id = d.dep_id;
```

- Prefer `JOIN ... ON` (SQL-92) over implicit comma joins (older SQL-89).
- Avoid `NATURAL JOIN` except in tightly controlled schemas.
- Use `USING(...)` for simple keys ; `ON(...)` for composite keys.
- Always use clear table aliases.

Definition and purpose

A **subquery** (nested query) is a query inside another query. It helps decompose complex logic into smaller steps.

```
-- Example: employees in the 'IT' department
SELECT name
FROM Employees
WHERE dep_id = (
  SELECT id
  FROM Departments
  WHERE name = 'IT'
);
```

- **Use case :** improve readability and maintainability.
- **Principle :** the subquery runs first ; its result feeds the outer query.
- **Result types :** single value, single row, or full table.

Where subqueries can appear

Subqueries can appear in `WHERE`, `FROM`, `SELECT`, or `HAVING`.

```sql
-- In WHERE: filter by a computed result
SELECT name
FROM Employees
WHERE dep_id = (SELECT id FROM Departments WHERE name = 'IT');

-- In FROM: create a derived table
SELECT d.name, s.avg_salary
FROM Departments d
JOIN (SELECT dep_id, AVG(salary) AS avg_salary
      FROM Employees GROUP BY dep_id) AS s
ON d.id = s.dep_id;
```

- `WHERE` : conditional filtering (most common).
- `FROM` : local virtual table.
- `SELECT` : computed value per row.
- `HAVING` : filtering over aggregates.

Examples by position

Examples of the main subquery positions :

```sql
-- Subquery in SELECT: global average
SELECT name,
       (SELECT AVG(salary) FROM Employees) AS global_avg
FROM Employees;

-- Subquery in HAVING: compare an aggregate
SELECT dep_id, AVG(salary)
FROM Employees
GROUP BY dep_id
HAVING AVG(salary) > (
  SELECT AVG(salary) FROM Employees
);
```

- Subqueries may yield a scalar, a row, or a table.
- They often improve clarity and reduce deeply nested joins.
- Consider equivalent rewrites with `JOIN` for performance.

Multi-value subqueries : IN, ANY, ALL

For subqueries returning multiple values, use `IN`, `ANY`, and `ALL`.

```sql
-- Employees belonging to any 'North' region department
SELECT name
FROM Employees
WHERE dep_id IN (
  SELECT id FROM Departments WHERE region = 'North'
);

-- Salary greater than all salaries in department 10
SELECT name
FROM Employees
WHERE salary > ALL (
  SELECT salary FROM Employees WHERE dep_id = 10
);
```

- `IN` : membership in a set.
- `ANY` : condition true for at least one value.
- `ALL` : condition true for every value.

Correlated subqueries

A **correlated subquery** depends on values from the outer query.

```sql
-- Employees whose salary exceeds their department average
SELECT e.name, e.salary
FROM Employees e
WHERE e.salary > (
  SELECT AVG(salary)
  FROM Employees
  WHERE dep_id = e.dep_id
);
```

- Executed once per outer row (expressive but often expensive).
- Often replaceable by an equivalent join.

Subqueries in FROM (derived tables)

A **derived table** is a subquery placed in `FROM` and treated as a local temporary table.

```
-- Average salary by department
SELECT d.name, s.avg_salary
FROM Departments d
JOIN (
  SELECT dep_id, AVG(salary) AS avg_salary
  FROM Employees
  GROUP BY dep_id
) AS s ON d.id = s.dep_id;
```

- Creates a temporary virtual table.
- An alias after the closing parenthesis is **required**.
- Not reusable elsewhere (unlike CTEs).

Named subqueries WITH (CTE)

A **CTE (Common Table Expression)** is a named subquery defined with `WITH`. It improves readability and can be reused multiple times within the same query.

```sql
-- Average salary by department
WITH Averages AS (
  SELECT dep_id, AVG(salary) AS avg_salary
  FROM Employees
  GROUP BY dep_id
)
SELECT e.name, e.salary, a.avg_salary
FROM Employees e
JOIN Averages a ON e.dep_id = a.dep_id;
```

- **Readable :** structures the query into logical blocks.
- **Reusable :** in several parts of the same query.
- **Standard SQL** — supported by PostgreSQL and MySQL $\geq$ 8.0.

Aggregation concept

An **aggregate function** combines multiple rows into a single summary value : sum, average, maximum, etc. This is a cornerstone of relational analysis.

```
-- Example: average salary of all employees
SELECT AVG(salary) AS avg_salary
FROM Employees;
```

- Main functions : `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.
- `NULL` values are ignored by aggregates (standard behavior) — PostgreSQL and MySQL conform.

Grouping rows (GROUP BY)

The `GROUP BY` clause applies aggregate functions on groups of rows sharing the same value in one or more columns.

```
-- Average salary by department
SELECT dep_id, AVG(salary) AS avg_salary
FROM Employees
GROUP BY dep_id
ORDER BY avg_salary DESC
FETCH FIRST 3 ROWS ONLY;  -- Top 3 departments (standard SQL:2008
    supported by PostgreSQL)
```

- Each group produces one output row.
- Only grouped or aggregated columns may appear in `SELECT`.
- PostgreSQL enforces this SQL standard rule strictly (MySQL relaxed it before v5.7).

Filtering groups (HAVING)

`HAVING` filters *after* aggregation, whereas `WHERE` filters *before* aggregation.

```
-- Departments with average salary > 4000
SELECT dep_id, AVG(salary) AS avg_salary
FROM Employees
GROUP BY dep_id
HAVING AVG(salary) > 4000
ORDER BY avg_salary DESC;
```

- `WHERE` $\rightarrow$ filters source rows.
- `HAVING` $\rightarrow$ filters groups after aggregation.
- PostgreSQL complies with the SQL standard, including nested aggregates.

Accurate counts (COUNT, DISTINCT)

COUNT counts rows or distinct values.

```
-- Total number of employees
SELECT COUNT(*) AS total_count FROM Employees;

-- Number of distinct departments
SELECT COUNT(DISTINCT dep_id) AS dept_count FROM Employees;
```

- `COUNT(*)` : counts all rows, including those with `NULL`s.
- `COUNT(col)` : ignores `NULL`s.
- `COUNT(DISTINCT col1, col2, ...)` : unique tuples across multiple columns (standard SQL ; supported by PostgreSQL and MySQL).

Combining clauses

These clauses can be combined to build complete analytical queries.

```
-- Top 3 departments with avg salary > 4000
SELECT dep_id, AVG(salary) AS avg_salary
FROM Employees
GROUP BY dep_id
HAVING AVG(salary) > 4000
ORDER BY avg_salary DESC
FETCH FIRST 3 ROWS ONLY;  -- Standard SQL:2008 (supported by
     PostgreSQL)
```

- Logical evaluation order : `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `DISTINCT` → `ORDER BY` → `OFFSET/FETCH`.
- PostgreSQL adheres to the ISO SQL standard ordering.

Analytic functions (windows)

Analytic functions (SQL :2003) extend classical aggregates : they compute values over dynamic windows of rows without collapsing them.

```
-- Department average per row, without grouping rows
SELECT dep_id, name, salary,
       AVG(salary) OVER (PARTITION BY dep_id) AS dep_avg
FROM Employees
ORDER BY dep_id, salary DESC
FETCH FIRST 10 ROWS ONLY;
```

- `OVER(...)` defines the analysis window.
- `PARTITION BY` groups logically without merging rows.
- PostgreSQL implements the main features from **SQL :2003 (windows)**.
- MySQL supports them since version 8.0 (2018).

Definition and creation

A **view** is a **virtual table** defined from a query. It does not store data ; it preserves the query's structure and logic.

```sql
-- View listing employees with their department
CREATE VIEW EmployeesDept AS
SELECT e.name, e.salary, d.name AS department
FROM Employees e
JOIN Departments d ON e.dep_id = d.dep_id;

-- Use a view like a table
SELECT * FROM EmployeesDept;
```

- **CREATE VIEW** creates a reusable logical table.
- **ALTER VIEW** modifies it ; **DROP VIEW** removes it.
- PostgreSQL and the SQL standard share the same basic syntax.

Updatable and non-updatable views

A view can be **updatable** if the DBMS can determine the target base table(s) for modifications ; otherwise it is **read-only**.

```
-- Updatable view: simple projection on a single table
CREATE VIEW EmployeesSimple AS
SELECT emp_id, name, salary FROM Employees;

UPDATE EmployeesSimple SET salary = salary * 1.05
WHERE emp_id = 10;  -- valid

-- Non-updatable view: join → ambiguity
CREATE VIEW EmployeesDept AS
SELECT e.name, d.name AS department FROM Employees e
JOIN Departments d ON e.dep_id = d.dep_id;

UPDATE EmployeesDept SET department = 'HR';  -- rejected
```

- Updatable : single table, no aggregates, no GROUP BY.
- Non-updatable : joins, aggregates, functions, DISTINCT, etc.
- PostgreSQL enforces SQL-standard rules.
- PostgreSQL can make certain read-only views **writable** via `INSTEAD OF TRIGGER` (DBMS-specific, non-standard).

Usage and best practices

Views are essential for **security**, **simplification**, and **abstraction**.

```
-- Example: restrict access to sensitive data
CREATE VIEW EmployeesPublic AS
SELECT name, role, dep_id
FROM Employees;

GRANT SELECT ON EmployeesPublic TO interns;
```

- **Security :** restrict visible columns or rows per role.
- **Abstraction :** hide join or calculation complexity.
- **Simplification :** factor frequent queries.
- **Performance :** possible with *materialized views* (non-standard ; available in PostgreSQL, Oracle). *SQL Server* offers *indexed views* (similar goal).

Conclusion (Declarative SQL)

- **Standard compliance :** Systematically adopt normalized SQL forms (`JOIN ... ON`, `GROUP BY`, `FETCH FIRST`, `OVER(...)`) to ensure **portability** and **conformance** to **ISO/IEC 9075**. This discipline yields durable, DBMS-agnostic queries.

- **Power of the declarative paradigm :** Joins, subqueries, CTEs, aggregates, and analytic functions enable **complex analytical workloads** without procedural programming.

- **Structure and security : Views** provide abstraction and protection, ensuring readability, reuse, and consistency.

Next section

The next section presents **concrete case studies** demonstrating the **power of declarative SQL** for **advanced statistical analyses**. All queries will be expressed in **standard SQL**, without procedural programming.

Global and per-group statistics

```sql
-- Global: mean, variance, and standard deviation (standard SQL:2003
   )
-- VAR_POP and STDDEV_POP use N (population)
-- VAR_SAMP uses N-1 (sample)

SELECT
  AVG(salary)        AS mean_value,
  VAR_POP(salary)    AS variance,
  STDDEV_POP(salary) AS std_dev
FROM Employees;

-- By department: grouped mean and standard deviation
SELECT dep_id,
       AVG(salary)        AS mean_value,
       STDDEV_POP(salary) AS std_dev
FROM Employees
GROUP BY dep_id
ORDER BY mean_value DESC;
```

Median and percentiles (ordered-set)

```
-- Quartiles (Q1, median, Q3) from the salary distribution
-- SQL:2003 (ORDERED-SET aggregates)
SELECT dep_id,
       PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY salary) AS q1,
                 -- first quartile (25%)
       PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY salary) AS
           median,  -- median (50%)
       PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY salary) AS q3
                  -- third quartile (75%)
FROM Employees
GROUP BY dep_id
ORDER BY dep_id;
```

The `PERCENTILE_CONT` function is an **ordered aggregate**. It computes the value below which a given percentage of observations falls, after sorting the data with `ORDER BY`.

If the computed position does not match an exact existing value, PostgreSQL performs **continuous interpolation** between neighboring values to obtain a smooth estimate.

Note that there is another variant : `PERCENTILE_DISC`.

Rankings and Top-N

```sql
-- Rank departments by average salary
-- RANK() OVER(...) assigns ranks with ties (different from
     ROW_NUMBER).
WITH DepAvg AS (
  SELECT dep_id, AVG(salary) AS mean_value
  FROM Employees
  GROUP BY dep_id
)
SELECT dep_id, mean_value,
       RANK() OVER (ORDER BY mean_value DESC) AS rank_no
FROM DepAvg
ORDER BY rank_no;

-- Top 5 per department: ROW_NUMBER() + filter rk <= 5
SELECT *
FROM (
  SELECT e.dep_id, e.name, e.salary,
         ROW_NUMBER() OVER (
           PARTITION BY e.dep_id
           ORDER BY e.salary DESC
         ) AS rk
  FROM Employees e
) t
WHERE rk <= 5
ORDER BY dep_id, rk;
```

Running totals and shares

```sql
-- Cumulative salary per department by salary order
SELECT dep_id, name, salary,
       SUM(salary) OVER (
         PARTITION BY dep_id
         ORDER BY salary
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS dept_cumsum
FROM Employees
ORDER BY dep_id, salary;

-- Relative share of a salary within its department total
SELECT dep_id, name, salary,
       salary::NUMERIC
       / NULLIF(SUM(salary) OVER (PARTITION BY dep_id), 0) AS
          dept_share
FROM Employees
ORDER BY dep_id, salary DESC;

-- Relative position (0..1): CUME_DIST() and PERCENT_RANK()
SELECT name, salary,
       CUME_DIST()    OVER (ORDER BY salary) AS cume_dist,
       PERCENT_RANK() OVER (ORDER BY salary) AS percent_rank
FROM Employees
ORDER BY salary;
```

Normalization and outliers (z-score)

```
-- z-score: (value - dept_mean) / dept_stddev
-- Compares deviations from the mean on different scales.
WITH Stats AS (
  SELECT e.*,
         AVG(salary)         OVER (PARTITION BY dep_id) AS mu,
         STDDEV_POP(salary) OVER (PARTITION BY dep_id) AS sigma
  FROM Employees e
)
SELECT dep_id, name, salary,
       (salary - mu) / NULLIF(sigma, 0) AS zscore
FROM Stats
ORDER BY dep_id, zscore DESC;

-- Outlier detection: |z| > 3 (empirical three-sigma rule)
WITH Stats AS (
  SELECT e.*,
         AVG(salary)         OVER (PARTITION BY dep_id) AS mu,
         STDDEV_POP(salary) OVER (PARTITION BY dep_id) AS sigma
  FROM Employees e
)
SELECT dep_id, name, salary,
       (salary - mu) / NULLIF(sigma, 0) AS zscore
FROM Stats
WHERE ABS((salary - mu) / NULLIF(sigma, 0)) > 3
ORDER BY dep_id, zscore DESC;
```