Chapter 2 — Advanced SQL

Declarative

Mr Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)



University of Skikda — Department of Computer Science

1st Year Master NDS/AI

Advanced DataBases (ADB)

Updated on 2026-10-30 at 10:08:11



© 2026 Mr Laïdi FOUGHALI — www.al-moualime.com/adb

Creative Commons License — BY-NC-SA 4.0 International

Course conventions

Each page has a **colour** that shows its role:

Blue: the lecture, to read and to run.

Orange: an exercise or a recall; look for the answer.

Green: the solution of the exercise, on the next page.

- **Method**: answer the exercise **before** turning the page.
- **Review**: this link takes you to the lecture page that holds the answer; the **"↩ Back to slide N"** button (or the R key) brings you back to the exercise (in a PDF: Alt+←).
- **Interactive version**: the **Run** button runs the SQL example in the browser; "Reset database" restores the initial data.

Outline

1. Introduction
2. Formatting results
3. Joins
4. Subqueries
5. Aggregation
6. Views
7. Window functions
8. Conclusion

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 on a relational DBMS, in this case **PostgreSQL**.
- **Objective of this chapter:** go beyond declarative SQL to explore the **advanced and procedural** mechanisms of the standardized language (the **SQL/PSM — Persistent Stored Modules** standard).
  - **Complex queries:** multiple joins, subqueries (simple and correlated), views, aggregations, CTEs.
  - **Procedural programming (preview of the next part):** stored procedures, functions, variables, control structures.
  - **Automation and robustness (preview):** triggers, transactions, error handling.

Purpose

Master complete SQL, both **declarative and procedural**, to develop reliable processing that complies with international standards.

References

- **SQL standard (ISO/IEC 9075:2023)** — relevant parts cited in this chapter:
  Foundation (9075-1), PSM (9075-4), SQL Routines and Types (9075-4/11), and analytical extensions (windows, SQL:2003).
- **ISO record**: *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 NDS/AI curriculum**, it forms the theoretical and pedagogical foundation of this chapter's content.

Important

- The **practical examples** are based on **PostgreSQL**, an **open-source** DBMS compliant with the **SQL standards (ISO/IEC 9075)** and relevant in a university context.
- Other DBMSs (**Oracle**, **SQL Server**, **MySQL**) are mentioned to point out **deviations from the standard** or **implementation specifics**.

The data of the chapter (1/3)

All the examples and exercises of the chapter use two tables: **departments** and their **employees**.

```
CREATE TABLE Departments (
    id        INT            PRIMARY KEY,
    name      VARCHAR(40)    NOT NULL UNIQUE,
    region    VARCHAR(20)    NOT NULL
);

CREATE TABLE Employees (
    emp_id    INT            PRIMARY KEY,
    name      VARCHAR(40)    NOT NULL,
    salary    NUMERIC(8,2)   NOT NULL,
    dept_id   INT            REFERENCES Departments(id)    -- NULL: no department
);
```

The data of the chapter (2/3)

```sql
INSERT INTO Departments (id, name, region) VALUES
    (10, 'Human Resources',  'South'),
    (20, 'Computer Science', 'North'),
    (30, 'Finance',          'North'),
    (40, 'Sales',            'East'),
    (50, 'Logistics',        'West');
```

Logistics has no employee: it will be used for outer joins.

The data of the chapter (3/3)

```sql
INSERT INTO Employees (emp_id, name, salary, dept_id) VALUES
    (1,  'Amine',  5200, 20),
    (2,  'Nadia',  4800, 20),
    (3,  'Yacine', 3900, 20),
    (4,  'Sonia',  3000, 10),
    (5,  'Karim',  2700, 10),
    (6,  'Lina',   4500, 30),
    (7,  'Omar',   4500, 30),
    (8,  'Rania',  3900, 40),
    (9,  'Walid',  2700, 40),
    (10, 'Selma',  3200, NULL);
```

- **Selma** has no department; some salaries are **equal** (4500, 3900, 2700).
- The examples sometimes modify the data: “Reset database” restores this state.

ORDER BY clause

The `ORDER BY` clause sorts the result set on one or more columns. In accordance with the SQL standard, sorting always takes place at the **end of logical processing**, after the `WHERE`, `GROUP BY` and `HAVING` clauses.

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

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

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

- The SQL standard allows using the **column name** or its **ordinal position**.
- Ascending order (`ASC`) is implicit; descending order (`DESC`) must be stated explicitly.
- Using column numbers can hurt code readability and maintainability.

FETCH FIRST / LIMIT clause (1/2)

To restrict the number of rows returned, the **SQL:2008** standard introduced the `FETCH FIRST n ROWS ONLY` clause. PostgreSQL implements this standard syntax while keeping the historical `LIMIT` command, one of its non-standard extensions.

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

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

- **ORDER BY** is essential to guarantee a deterministic result.
- The `FETCH FIRST` syntax should be preferred for cross-DBMS portability.

FETCH FIRST ... WITH TIES clause (2/2)

The `WITH TIES` option includes all rows whose values on the **sort columns** are identical to those of the last row kept by the limit. If several records share these same sort values, they are all returned.

```sql
-- Standard SQL:2008 + PostgreSQL (>= 13)
SELECT name, salary
FROM Employees
ORDER BY salary DESC
FETCH FIRST 5 ROWS WITH TIES;  -- may return more than 5 rows if salaries are
     tied

-- [SQL Standard] ties on (salary, emp_id)
SELECT name, salary, emp_id
FROM Employees
ORDER BY salary DESC, emp_id DESC
FETCH FIRST 5 ROWS WITH TIES;
```

- **ORDER BY** is mandatory: the comparison covers all its columns, in the given order.
- `LIMIT` does not support `WITH TIES` (use `FETCH`).
- Behavior compliant with the **SQL:2008** standard, implemented in PostgreSQL since version 13.

OFFSET / FETCH clauses (1/3)

To display results as **successive pages**, the **SQL:2008** standard combines the `OFFSET` and `FETCH` clauses. PostgreSQL implements `OFFSET/FETCH` (standard). The `WITH TIES` option is available since v13.

```
-- Page 2: rows 6 to 10 (standard)
SELECT name, salary
FROM Employees
ORDER BY salary DESC
OFFSET 5 ROWS              -- skip the first 5 rows
FETCH NEXT 5 ROWS ONLY;    -- return the next 5
```

- `FIRST` and `NEXT` are equivalent; `NEXT` reads more naturally for subsequent pages.
- **ORDER BY** is required for stable, reproducible pagination.

LIMIT / OFFSET (extension) (2/3)

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

- Same logic as `OFFSET/FETCH`, but with a non-standard syntax.
- To include ties, prefer `FETCH ... WITH TIES`.
- Compatible with older versions of PostgreSQL.

Pagination performance (3/3)

Keyset pagination

The next page is computed from the last row displayed. Reading resumes **from a known key**, without scanning the previous rows.

```sql
SELECT emp_id, name, salary FROM Employees
WHERE (salary, emp_id) < (2700, 1523) -- Continue after (2700,1523)
ORDER BY salary DESC, emp_id DESC -- Lexicographic comparison (row value)
FETCH FIRST 20 ROWS ONLY;

-- (3000,2001) < (2700,1523) → FALSE  (higher salary)
-- (2700,1600) < (2700,1523) → FALSE  (higher id)
-- (2700,1500) < (2700,1523) → TRUE   (lower id)
-- (2600,1800) < (2700,1523) → TRUE   (lower salary)

-- Recommended index (portability):
CREATE INDEX idx_employees_salary_id ON Employees (salary, emp_id);
```

- `OFFSET`: reads then discards the previous rows → increasing time.
- `Keyset`: resumes directly after the last key → constant time.
- Index `(salary, id)` required to exploit the sort without memory overhead.
- `id` guarantees a unique and stable order (avoids duplicates).

Exercise 1: which names?

Data: the six highest salaries are Amine (5200), Nadia (4800), Lina (4500), Omar (4500), Rania (3900) and Yacine (3900).

Question: which names does this query display?

```
SELECT name
FROM   Employees
ORDER  BY salary DESC, name
FETCH  FIRST 3 ROWS ONLY;
```

a) Amine, Nadia, Lina
b) Amine, Nadia, Omar
c) Amine, Nadia, Lina, Omar

Choose, then run the query to check. Review: ORDER BY and FETCH FIRST

Solution of exercise 1

Answer: a) Amine, Nadia, Lina.

- The sort is first on `salary DESC`: Amine (5200), Nadia (4800), then Lina and Omar (4500).
- With equal salaries, the second key `name` decides: Lina comes before Omar.
- `FETCH FIRST 3 ROWS ONLY` stops at exactly 3 rows: Omar is not displayed.

Exercise 2: how many rows?

Data: the same salaries as in exercise 1.

Question: how many rows does this query display?

```
SELECT name, salary
FROM   Employees
ORDER  BY salary DESC
FETCH  FIRST 3 ROWS WITH TIES;
```

a) 3 **b)** 4 **c)** 6

Choose, then run the query to check. Review: WITH TIES

Solution of exercise 2

Answer: b) 4 rows.

- The first 3 rows are Amine (5200), Nadia (4800) and one employee at 4500.
- `WITH TIES` adds every row **tied** with the 3rd on the sort column: Lina and Omar both earn 4500.
- Rania and Yacine (3900) are not tied with the 3rd row: they are not added.

Recall: formatting results

At the start of each section, three questions on the previous section.
Answer **from memory**, then click "Review" to check.

1. Why is an `ORDER BY` needed with `FETCH FIRST`?
   Review: FETCH FIRST / LIMIT
2. What does the `WITH TIES` option do?
   Review: WITH TIES
3. Why is keyset pagination faster than `OFFSET`?
   Review: Pagination performance

Why use joins?

When several tables are combined without a matching condition, the DBMS produces a **Cartesian product**: each row of the first table is paired with every row of the second.

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

This result is useless in most cases, because no logical relationship is defined between the tables.
To establish this correspondence, a **join condition** must be added.

Role of aliases

Aliases (`e`, `d`) are **short names** given to tables in order to:

- shorten column references: `e.name` instead of `Employees.name`;
- distinguish tables when several have columns with the same name;
- make queries more readable, especially with multiple joins.

Inner join (INNER JOIN)

An **inner join** (`INNER JOIN`) returns only the rows that have a match in both joined tables. In other words, only the combinations for which the matching condition is true are kept.

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

Here, each employee is paired with their department through the `dept_id` key. Employees without a department, or departments without employees, do not appear in the result.

Logical interpretation

An inner join acts as an **intersection** between two data sets: only the rows present in both tables are displayed.

The `INNER` keyword is optional: `JOIN` and `INNER JOIN` produce the same result.

Outer joins (OUTER JOIN) [SQL-92]

An **outer join** (`OUTER JOIN`) displays not only the matches between two tables, but also the **unmatched** rows that have no counterpart in the other table.

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

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

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

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

Self joins (SELF JOIN) (1/2)

A **self join** links a table to itself when an internal relationship exists, for example to represent married people in the person table.

```
CREATE TABLE person (
  person_id  INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  name       VARCHAR(80) NOT NULL,
  sex        CHAR(1) CHECK (sex IN ('M','F')),
  spouse_id  INTEGER,  -- NULL if single
  parent_id  INTEGER,  -- NULL if parent unknown

  CONSTRAINT ck_person_spouse_self CHECK (spouse_id IS NULL OR spouse_id <>
     person_id),
  FOREIGN KEY (spouse_id) REFERENCES person(person_id)
    DEFERRABLE INITIALLY DEFERRED,
  FOREIGN KEY (parent_id) REFERENCES person(person_id)
    DEFERRABLE INITIALLY DEFERRED
);
```

The columns spouse_id and parent_id are **self-referencing foreign keys** pointing to the same table. The DEFERRABLE INITIALLY DEFERRED option postpones constraint checking until the end of the transaction: this advanced feature is supported by **PostgreSQL** — but not by **MySQL**, **SQL Server** or **Firebird**.

[PostgreSQL documentation — CREATE TABLE (DEFERRABLE) and SET CONSTRAINTS]

Self joins (SELF JOIN) (2/2)

The self join pairs each person with their spouse using a single table.

```
-- person_id assigned in order: Ahmed 1, Leila 2, Ali 3, Sara 4, Yasmine 5
INSERT INTO person (name, sex, spouse_id, parent_id) VALUES
    ('Ahmed',   'M', 2,    NULL),
    ('Leila',   'F', 1,    NULL),
    ('Ali',     'M', 4,    NULL),
    ('Sara',    'F', 3,    NULL),
    ('Yasmine', 'F', NULL, 1);
```

```
SELECT p1.name AS person, p2.name AS spouse
FROM person p1
JOIN person p2 ON p1.spouse_id = p2.person_id;
```

The aliases p1 and p2 represent two distinct occurrences of the same person table.

Exercise 3: how many rows per join?

Data: 10 employees, including Selma **with no department**; 5 departments, including Logistics **with no employee**.

Question: how many rows does each query count? Write down your four numbers.

```
SELECT COUNT(*) FROM Employees e JOIN       Departments d ON e.dept_id = d.id;
SELECT COUNT(*) FROM Employees e LEFT JOIN  Departments d ON e.dept_id = d.id;
SELECT COUNT(*) FROM Employees e RIGHT JOIN Departments d ON e.dept_id = d.id;
SELECT COUNT(*) FROM Employees e FULL JOIN  Departments d ON e.dept_id = d.id;
```

Write down your answers, then run the four queries to check. Review: INNER JOIN and OUTER JOIN

Solution of exercise 3

Join	**Rows**	**Why**
`JOIN`	9	the 9 employees who have a department
`LEFT JOIN`	10	+ Selma, with a `NULL` department
`RIGHT JOIN`	10	+ Logistics, with a `NULL` employee
`FULL JOIN`	11	+ Selma and + Logistics

Exercise 4: which departments have no employee?

Question: which query displays the departments **with no employee**?

```sql
-- a)
SELECT d.name FROM Departments d
JOIN Employees e ON e.dept_id = d.id;
-- b)
SELECT d.name FROM Departments d
LEFT JOIN Employees e ON e.dept_id = d.id
WHERE e.emp_id IS NULL;
-- c)
SELECT d.name FROM Departments d
LEFT JOIN Employees e ON e.dept_id = d.id
WHERE e.emp_id = NULL;
```

Choose, then run the three queries to check. Review: OUTER JOIN

Solution of exercise 4

Answer: b) it displays Logistics.

a) The inner join keeps only the departments **that have** employees (one name per employee: 9 rows).

b) `LEFT JOIN` also keeps Logistics, with `NULL` employee columns; `IS NULL` keeps only it.

c) No row: a comparison `= NULL` is **never true**; you must write `IS NULL`.

Recall: joins

Answer **from memory**, then click "Review" to check.

1. What does a query on two tables produce **without** a join condition?
   Review: Why use joins?
2. What is the difference between `JOIN` and `LEFT JOIN`?
   Review: Outer joins
3. What is a self join?
   Review: Self joins

Definition and purpose

A **subquery** (or **nested query**) is a query placed inside another one. It breaks a complex query down into logical sub-steps.

```sql
-- Example: employees of the 'Computer Science' department
SELECT name
FROM Employees
WHERE dept_id = (
  SELECT id
  FROM Departments
  WHERE name = 'Computer Science'
);
```

- **Usefulness:** make SQL easier to read and maintain.
- **Principle:** the subquery runs first, and its result is passed to the main query.
- **Possible result:** a single value, one row, or a whole table.

Possible positions of subqueries

A subquery can appear in several clauses of an SQL query: `WHERE`, `FROM`, `SELECT` or `HAVING`.

```sql
-- Subquery in SELECT: computing an overall average
SELECT name,
       (SELECT AVG(salary) FROM Employees) AS overall_average
FROM Employees;

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

- `WHERE`: conditional filtering (the most common case, example on the previous slide).
- `FROM`: creation of a local virtual table (see “Derived tables”).
- `SELECT`: adding a computed value to each row.
- `HAVING`: filtering on an aggregate.
- To be optimized: some subqueries can be rewritten with `JOIN`s.

Multi-row subqueries: IN, ANY, ALL

Subqueries returning several values are used with the `IN`, `ANY` and `ALL` operators.

```sql
-- Employees belonging to a department in the North
SELECT name
FROM Employees
WHERE dept_id IN (
  SELECT id FROM Departments WHERE region = 'North'
);

-- Salary > all those of department 10
SELECT name
FROM Employees
WHERE salary > ALL (
  SELECT salary FROM Employees WHERE dept_id = 10
);
```

- `IN`: membership of a set.
- `ANY`: condition true for at least one value.
- `ALL`: condition true for all values.

Correlated subqueries

A **correlated subquery** depends on a value from the main query.

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

- Executed for each row of the outer query.
- Very expressive but often costly.
- Equivalent rewrite, often more efficient: a join with a **CTE** (see "Named subqueries WITH").

Subqueries in FROM (derived tables)

A **derived table** is a subquery placed in the `FROM` clause. It is treated as a temporary table local to the main query.

```sql
-- Average salary per department
SELECT d.name, s.average
FROM Departments d
JOIN (
  SELECT dept_id, AVG(salary) AS average
  FROM Employees
  GROUP BY dept_id
) AS s ON d.id = s.dept_id;
```

- Creates a temporary virtual table.
- The alias after the parenthesis is **mandatory**.

Named subqueries WITH (CTE)

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

```sql
-- Employees paid above the average of their department
-- (rewrite of the correlated subquery seen earlier)
WITH Averages AS (
  SELECT dept_id, AVG(salary) AS average
  FROM Employees
  GROUP BY dept_id
)
SELECT e.name, e.salary, m.average
FROM Employees e
JOIN Averages m ON e.dept_id = m.dept_id
WHERE e.salary > m.average;
```

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

Exercise 5: how many employees?

Data: the 10 salaries are 5200, 4800, 4500, 4500, 3900, 3900, 3200, 3000, 2700 and 2700.

Question: how many employees does this query display?

```
SELECT name
FROM   Employees
WHERE  salary > (SELECT AVG(salary) FROM Employees);
```

a) 4 **b)** 5 **c)** 6 **d)** 10

Choose, then run the query to check. Review: Definition and purpose

Solution of exercise 5

Answer: c) 6 employees.

- The subquery runs first: average $= 38\,400/10 = 3840$.
- Salaries above 3840: 5200, 4800, 4500, 4500, 3900 and 3900.
- That is Amine, Nadia, Lina, Omar, Yacine and Rania.

Exercise 6: what happens?

Data: two departments are in the North region, Computer Science and Finance.

Question: what does this query give?

```
SELECT name
FROM   Employees
WHERE  dept_id = (SELECT id FROM Departments WHERE region = 'North');
```

a) the 5 employees of the North
b) no row
c) an error

Choose, then run the query to check. Review: IN, ANY, ALL

Solution of exercise 6

Answer: c) an error: the subquery returns **two** rows (20 and 30).

- With =, the subquery must return **a single** value.
- To compare with a set of values, use `IN`:

```
SELECT name
FROM   Employees
WHERE  dept_id IN (SELECT id FROM Departments WHERE region = 'North');
```

Recall: subqueries

Answer **from memory**, then click "Review" to check.

1. A subquery returns several values: which operator replaces =?
   Review: IN, ANY, ALL
2. What is a **correlated** subquery?
   Review: Correlated subqueries
3. What must be written after a subquery placed in `FROM`?
   Review: Derived tables

The notion of aggregation

An **aggregate function** combines several rows to obtain *summarized* information: sum, average, minimum, maximum, etc. These functions turn a set of values into a single one — useful for any statistical analysis.

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

- Aggregates apply to a set of rows and produce a single value.
- Standard functions: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.
- `NULL` values are ignored (rule defined by the ISO SQL standard).
- The alias (`AS average_salary`) is optional: it only serves to name the result.

Grouping rows (GROUP BY)

To compute aggregates per category, rows are grouped on one or more columns using `GROUP BY`.

```sql
-- Average salary per department
SELECT dept_id, AVG(salary) AS average
FROM Employees
GROUP BY dept_id
ORDER BY average DESC;  -- or ORDER BY AVG(salary) DESC
```

- Each distinct value of `dept_id` forms a group.
- Each group returns a single row with its aggregated values.
- Only aggregated or grouped columns are allowed in `SELECT`.
- The alias is optional; one can also write `ORDER BY AVG(salary)`.

Filtering groups (HAVING)

Once the groups are formed, the `HAVING` clause filters the results according to a condition on the aggregated values. It runs *after* aggregation, unlike `WHERE`, which acts *before*.

```sql
-- Departments whose average salary exceeds 4000 €
SELECT dept_id, AVG(salary)
FROM Employees
GROUP BY dept_id
HAVING AVG(salary) > 4000;
```

- `WHERE` filters the source rows before grouping.
- `HAVING` filters the groups after aggregation.
- An alias defined in `SELECT` is not yet known in `HAVING`.

Counting (COUNT, DISTINCT)

The `COUNT` function counts rows or unique values.

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

-- Number of distinct departments
SELECT COUNT(DISTINCT dept_id) AS nb_departments FROM Employees;
```

- `COUNT(*)`: counts all rows, including those containing `NULL`s.
- `COUNT(col)`: ignores `NULL`s.
- Distinct pairs: `COUNT(DISTINCT (col1, col2))` in PostgreSQL (row constructor); the form `COUNT(DISTINCT col1, col2)` is a MySQL extension, rejected by PostgreSQL.
- The alias is optional but recommended to name the result.

Combining clauses

A complete query often combines several clauses: grouping, filtering and sorting. The execution order defined by the SQL standard is strict: it guarantees consistent results.

```sql
-- Top 3 departments with an average salary > 4000 €
SELECT dept_id, AVG(salary) AS average
FROM Employees
GROUP BY dept_id
HAVING AVG(salary) > 4000
ORDER BY average DESC
FETCH FIRST 3 ROWS ONLY;  -- SQL:2008 (PostgreSQL >= 8.4; WITH TIES >= 13)
```

- Logical execution order: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `FETCH`.
- PostgreSQL follows this order as defined by the ISO SQL standard.
- `ORDER BY` sorts the groups by the aggregated value.
- The alias is optional; it simply improves the readability of the code and of the result.

Exercise 7: what do the three COUNTs count?

Data: 10 employees; Selma has no department; the 9 others are in 4 departments.

Question: which three numbers are displayed?

```
SELECT COUNT(*), COUNT(dept_id), COUNT(DISTINCT dept_id)
FROM   Employees;
```

a) 10, 10, 5 **b)** 10, 9, 4 **c)** 10, 9, 5 **d)** 9, 9, 4

Choose, then run the query to check. Review: COUNT and DISTINCT

Solution of exercise 7

Answer: b) 10, 9, 4.

- `COUNT(*)` counts every row: 10.
- `COUNT(dept_id)` ignores `NULL`s: 9 (Selma is not counted).
- `COUNT(DISTINCT dept_id)` counts the different values, without `NULL`: 10, 20, 30 and 40, so 4.

Exercise 8: how many groups?

Data: the same as in exercise 7.

Question: how many rows does this query display?

```
SELECT dept_id, COUNT(*)
FROM   Employees
GROUP  BY dept_id;
```

a) 4 **b)** 5 **c)** 10

Choose, then run the query to check. Review: GROUP BY

Solution of exercise 8

Answer: b) 5 rows.

- One row per value of `dept_id`: 10, 20, 30 and 40.
- **Plus** one group for `NULL`: `NULL`s are grouped together (Selma, 1 employee).
- `GROUP BY` does not drop `NULL`s, unlike `COUNT(dept_id)`.

Recall: aggregation

Answer **from memory**, then click "Review" to check.

1. What is the difference between `WHERE` and `HAVING`?
   Review: HAVING
2. What is the difference between `COUNT(*)` and `COUNT(column)`?
   Review: COUNT and DISTINCT
3. In which logical order do `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY` run?
   Review: Combining clauses

Views (definition and creation)

A **view** is a **virtual table** defined from a query. It does not store data, but keeps the structure and logic of the query.

```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.dept_id = d.id;

-- Using 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 standard SQL share the same syntax.

Updatable and non-updatable views

A view can be **updatable** if the DBMS can determine which underlying table(s) the changes apply to. Otherwise, it is **non-updatable** (read-only).

```sql
-- Updatable view: simple projection over a single table
CREATE VIEW EmployeesSimple AS
SELECT emp_id, name, salary, dept_id
FROM Employees
WITH CHECK OPTION;  -- Standard: any row inserted or modified through
                    -- the view must remain visible through it after the
     operation.

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

-- Non-updatable view: EmployeesDept (previous slide),
-- defined by a join → ambiguity
UPDATE EmployeesDept SET department = 'HR';  -- rejected
```

- Updatable: a single table, no aggregates, no GROUP BY.
- Non-updatable: joins, aggregates, functions, DISTINCT, etc.
- PostgreSQL strictly applies the rules of the SQL standard.

Uses and good practices

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

```sql
-- Example: restricting access to sensitive data
CREATE VIEW employees_public AS
SELECT name, dept_id
FROM Employees;
```

```sql
GRANT SELECT ON employees_public TO interns;    -- interns: an existing role
```

- **Security:** restrict access to columns or rows depending on roles.
- **Abstraction:** hide the complexity of queries or joins.
- **Simplification:** easily reuse queries defined once.
- **Metadata:** the standard defines `information_schema` as a standardized view for accessing metadata (implemented by several DBMSs, including PostgreSQL).

Exercise 9: what does the view contain?

Data: 4 employees earn more than 4000 (Amine, Nadia, Lina, Omar); Rania earns 3900.

Question: which numbers do the two `SELECT COUNT(*)` display?

```sql
CREATE OR REPLACE VIEW HighEarners AS
SELECT name, salary FROM Employees WHERE salary > 4000;

SELECT COUNT(*) FROM HighEarners;
UPDATE Employees SET salary = 4200 WHERE name = 'Rania';
SELECT COUNT(*) FROM HighEarners;

UPDATE Employees SET salary = 3900 WHERE name = 'Rania';   -- restores the data
```

a) 4 then 4 **b)** 4 then 5 **c)** 5 then 5

Choose, then run the script to check. Review: Views

Solution of exercise 9

Answer: b) 4 then 5.

- A view **does not store** rows: it keeps the **query**.
- Each read of the view runs this query again on the **current** data.
- After the `UPDATE`, Rania (4200) earns more than 4000: she appears in the view.

Exercise 10: does the update go through?

Question: what does the `UPDATE` on this view do?

```
CREATE OR REPLACE VIEW ComputerScientists AS
SELECT emp_id, name, dept_id
FROM   Employees
WHERE  dept_id = 20                    -- Computer Science
WITH CHECK OPTION;

UPDATE ComputerScientists SET dept_id = 30 WHERE name = 'Yacine';
```

a) Yacine moves to Finance (30)
b) Yacine moves to Finance and leaves the view, without a message
c) an error

Choose, then run the script to check. Review: Updatable views

Solution of exercise 10

Answer: c) an error: the new row violates the check option of the view.

- With `dept_id = 30`, Yacine would no longer satisfy `WHERE dept_id = 20`: he would leave the view.
- `WITH CHECK OPTION` **rejects** any change that would make a row leave the view.
- Without this option, the answer would be b).

Recall: views

Answer **from memory**, then click "Review" to check.

1. Does a view store rows?
   Review: Views (definition and creation)

2. Why is a view defined by a join not updatable?
   Review: Updatable and non-updatable views

3. How does a view help protect sensitive columns?
   Review: Uses and good practices

Window functions

`GROUP BY` **collapses** rows and loses their detail: one row per group in the output. **Window functions** (SQL:2003) also compute aggregates or ranks over a set of related rows — but **without reducing the number of rows returned**.

```sql
-- GROUP BY: one row per department, detail lost
SELECT dept_id, AVG(salary) FROM Employees GROUP BY dept_id;

-- Window function: one row per employee, with the
-- department average available on each row
SELECT name, dept_id, salary,
       AVG(salary) OVER (PARTITION BY dept_id) AS dept_average
FROM Employees;
```

- General syntax: `function(...) OVER (PARTITION BY ... ORDER BY ...)`.
- `PARTITION BY` splits the rows into groups (like `GROUP BY`), but each row remains individually visible.
- `ORDER BY` (inside the `OVER`) orders the rows *within* each partition — independently of any final `ORDER BY` of the query.

Ranking: ROW_NUMBER, RANK, DENSE_RANK

```
SELECT name, dept_id, salary,
       ROW_NUMBER() OVER (PARTITION BY dept_id
                          ORDER BY salary DESC) AS row_no,
       RANK()       OVER (PARTITION BY dept_id
                          ORDER BY salary DESC) AS rnk,
       DENSE_RANK() OVER (PARTITION BY dept_id
                          ORDER BY salary DESC) AS dense_rnk
FROM Employees;
```

- `ROW_NUMBER()`: numbers 1, 2, 3, ... without exception — two tied rows get different numbers (arbitrary order between them).
- `RANK()`: tied rows get the same rank; the next rank **skips** (1, 1, 3, ...).
- `DENSE_RANK()`: tied rows get the same rank; the next rank **does not skip** (1, 1, 2, ...).
- `PARTITION BY` restarts the counter at 1 for each department.

Window aggregates: comparing and accumulating

```sql
-- Difference between each salary and its department's average
SELECT name, dept_id, salary,
       salary - AVG(salary) OVER (PARTITION BY dept_id)
         AS gap_to_average
FROM Employees;

-- Running total of salaries by seniority (ascending emp_id)
SELECT name, emp_id, salary,
       SUM(salary) OVER (ORDER BY emp_id) AS running_payroll
FROM Employees;
```

- Any standard aggregate function (`SUM`, `AVG`, `COUNT`, `MIN`, `MAX`) can be used as a window function.
- Without `PARTITION BY`, the window covers the whole table.
- With `ORDER BY` and no explicit frame, PostgreSQL accumulates by default from the start of the partition up to the current row — hence the running total in the second example.

Accessing the previous/next row

`LAG`/`LEAD` read a value from a **neighboring row**, without a self join.

```
SELECT name, salary,
       LAG(salary)  OVER (ORDER BY salary DESC) AS higher_salary,
       LEAD(salary) OVER (ORDER BY salary DESC) AS lower_salary,
       salary - LAG(salary) OVER (ORDER BY salary DESC) AS gap
FROM Employees;
```

- `LAG(expr, n)`: value of the row n positions **before** (default $n = 1$).
- `LEAD(expr, n)`: value of the row n positions **after**.
- First/last row of the partition: the result is `NULL` (no neighbor), unless a 3rd argument supplies a default value.

Practical case: Top-N per group

A classic problem: *"the 2 highest salaries in each department"*. Impossible with a simple `GROUP BY` + `LIMIT` (which limits the whole query, not each group). Solution: `ROW_NUMBER()` in a CTE, filtered afterwards.

```
WITH ranking AS (
  SELECT name, dept_id, salary,
         ROW_NUMBER() OVER (PARTITION BY dept_id
                            ORDER BY salary DESC) AS rnk
  FROM Employees
)
SELECT name, dept_id, salary
FROM ranking
WHERE rnk <= 2;
```

- A window function is **not allowed** directly in a `WHERE` clause (it is evaluated after it): a CTE or a subquery is used to filter on its result.
- A very common pattern in practice: rankings, deduplication (keeping the most recent row per key), advanced pagination.

Exercise 11: which ranks?

Data: departments 20 and 30: Amine (5200), Nadia (4800), Lina (4500), Omar (4500), Yacine (3900).

Question: which ranks does Yacine get (`rnk`, then `dense_rnk`)?

```
SELECT name, salary,
       RANK()       OVER (ORDER BY salary DESC) AS rnk,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rnk
FROM   Employees
WHERE  dept_id IN (20, 30);
```

a) 4 and 4 **b)** 5 and 4 **c)** 5 and 5 **d)** 4 and 5

Choose, then run the query to check. Review: ROW_NUMBER, RANK, DENSE_RANK

Solution of exercise 11

Answer: b) 5 and 4.

Name	**Salary**	`RANK`	`DENSE_RANK`
Amine	5200	1	1
Nadia	4800	2	2
Lina	4500	3	3
Omar	4500	3	3
Yacine	3900	**5**	**4**

After two ties at rank 3, `RANK` skips rank 4; `DENSE_RANK` does not skip.

Exercise 12: what happens?

Question: we want the 3 highest salaries. What does this query give?

```
SELECT name
FROM   Employees
WHERE  ROW_NUMBER() OVER (ORDER BY salary DESC) <= 3;
```

a) the 3 highest salaries
b) no row
c) an error

Choose, then run the query to check. Review: Top-N per group

Solution of exercise 12

Answer: c) an error: a window function is not allowed in `WHERE`.

`WHERE` is evaluated **before** window functions. Compute the rank in a CTE, then filter:

```
WITH ranking AS (
  SELECT name, ROW_NUMBER() OVER (ORDER BY salary DESC) AS rnk
  FROM   Employees
)
SELECT name FROM ranking WHERE rnk <= 3;
```

Conclusion (advanced declarative SQL)

- **Portability and compliance:** Systematic use of standardized constructs (`ORDER BY`, `FETCH FIRST`, `JOIN`, `GROUP BY`, `HAVING`) ensures **cross-DBMS compatibility** and the **longevity** of queries.
- **Power of the declarative approach:** The mechanisms studied — sorting, grouping, joins, subqueries, CTEs, views and **window functions** — demonstrate SQL's ability to express **complex and analytical** processing without resorting to procedural programming.
- **Structure and security: Views** are an essential tool to structure access, simplify analyses and protect sensitive data.

Next section

The next section covers **procedural SQL programming (SQL/PSM)**, extending declarative logic with **variables, procedures and triggers**.