Lecture 4. PostgreSQL DQL

SELECT

Core idea

> `SELECT` is the main **query** statement in PostgreSQL.

It can be very complex, but the basic form for a single table is:

```
SELECT select_list
FROM table_name;
```

- `select_list` – what to show (columns, expressions, , etc.).
- `table_name` – from which table to read rows.

Basic SELECT examples

One column

```
SELECT first_name
FROM customer;
```

Returns only the `first_name` column of all rows.

Multiple columns

```
SELECT first_name, last_name, email
FROM customer;
```

Returns several columns from the same table.

All columns

```
SELECT *
FROM customer;
```

`*` = “all columns” of the table.

SELECT with expressions

```
SELECT first_name || ' ' || last_name, email
FROM customer;
```

`||` is string concatenation: builds full name.

You can also select **expressions without a table**:

```
SELECT 5 * 3;
SELECT now();
```

- `5 * 3` → arithmetic expression.
- `now()` → function returning current timestamp.

Column Aliases

> A **column alias** is a temporary name for a column or expression in the result.

```
SELECT column_or_expression AS alias_name
FROM table_name;
```

- Alias exists **only during that query**.
- Useful for:
  - readability
  - giving names to expressions
  - using names in client code / tools.

Table Aliases

> **Table aliases** give a table a temporary shorter name in a query.

Syntax:

```
FROM table_name AS alias_name
```

or just:

```
FROM table_name alias_name
```

Use cases:

- Short names (`c` instead of `customer`) – especially useful in joins.
- Required when you use the same table more than once in a query.

SELECT DISTINCT

> `DISTINCT` removes **duplicate rows** from the result set.

```
SELECT DISTINCT column1
FROM table_name;
```

Result will contain each distinct `column1` value at most once.

With multiple columns, `DISTINCT` works on **combinations** of values.

PostgreSQL Operators

Operators make queries more powerful and flexible.

- Arithmetic operators
- Comparison operators
- Logical operators
- Pattern-matching operators
- NULL-related operators
- Other operators (concatenation, `BETWEEN`, `IN`, etc.)

Arithmetic Operators

Used on numeric data:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo (remainder)

Comparison Operators

Compare two values and return **boolean** (`TRUE`, `FALSE`, or `NULL`):

Typical SQL comparison operators include:

- `=` equal
- `<>` not equal
- `<` less than
- `>` greater than
- `<=` less or equal
- `>=` greater or equal

(Used in `WHERE`, `JOIN`, etc.)

Logical Operators

Combine conditions in `WHERE`, `HAVING`, etc.:

- `AND` – TRUE only if **all** conditions are true.
- `OR` – TRUE if **at least one** condition is true.
- `NOT` – inverts the result of a condition.

Pattern Matching Operators

Used to search for patterns inside text values.

- `LIKE` – SQL pattern:
  - `%` – any sequence of characters (0 or more)
  - `_` – exactly one character

- `ILIKE` – case-insensitive version of `LIKE`.
- `~` – matches POSIX regular expression (PostgreSQL-specific).

NULL-related Operators

Check for `NULL` explicitly:

- `IS NULL`
- `IS NOT NULL`

Because `= NULL` and `<> NULL` do **not** work as expected.

Other Operators

- Concatenation: `||` – join two strings.
- `BETWEEN` – check if value is in a **range** (inclusive).
- `IN` – check if value is in a **list** of values.

SELECT List – Column Names

If you **don't specify a column alias**:

- If the expression is a **simple column reference**, its result column uses that column's name.
- If the expression is more complex (functions, operators, etc.):
  - PostgreSQL may use a **function or type name**, or
  - fall back to a generated name like `?column?`.

So it's usually better to **use `AS` aliases** for expressions.

WHERE Clause

> The `WHERE` clause filters rows **before** they appear in the result.

General form:

```
SELECT ...
FROM table_name
```

```
WHERE condition;
```

- `condition` must evaluate to **boolean**.
- Rows that do **not** satisfy the condition are **removed** from the result.

Example idea:

```
SELECT *
FROM customer
WHERE first_name = 'John';
```

GROUP BY

> `GROUP BY` gathers rows with the same values into **groups**, usually to use **aggregate functions** (like `COUNT`, `SUM`, etc.).

```
SELECT select_list
FROM table_name
GROUP BY grouping_element;
```

- All selected rows that share the same values for the **grouped expressions** are condensed into **one row per group**.
- Grouping expression can be:
  - input column name,
  - name or ordinal number of an output column,
  - or any expression built from input columns.

HAVING

> `HAVING` filters groups **after** `GROUP BY`

```
SELECT select_list
FROM table_name
```

```
GROUP BY grouping_element
HAVING condition;
```

- `condition` is like in `WHERE`, but applied to **group rows**.
- `HAVING` removes groups that **do not** satisfy the condition. Lecture4

Difference from `WHERE`:

- `WHERE` – filters **individual rows** before grouping.
- `HAVING` – filters **groups** after `GROUP BY`.
- Columns in `HAVING` condition must reference:
  - grouping columns, or
  - be inside aggregate functions.

Set Operations: UNION, INTERSECT, EXCEPT

> These operators combine result sets from multiple `SELECT` statements. Each `select_statement` must **not** have its own `ORDER BY` or `LIMIT` (those belong to the final query).

Also: the combined statements must have:

- The **same number of columns**.
- Corresponding columns of **compatible data types**.

UNION

```
select_statement
UNION [ALL]
select_statement;
```

- Computes the **set union** of rows from both result sets.
- A row is in the union if it appears in **at least one** result set.
- By default, duplicates are **removed** (set semantics).

- With `UNION ALL`, duplicates are **kept** (multiset semantics).

INTERSECT

```
select_statement
INTERSECT [ALL]
select_statement;
```

- Computes the **intersection**: rows that appear in **both** result sets.
- Default: no duplicate rows in the result.
- With `INTERSECT ALL`:
  - if a row appears `m` times in left and `n` times in right,
  - it appears `min(m, n)` times in the result.

Precedence:

- `INTERSECT` binds more tightly than `UNION`.
  - `A UNION B INTERSECT C` is read as:

    `A UNION (B INTERSECT C)`.

EXCEPT

```
select_statement
EXCEPT [ALL]
select_statement;
```

- Computes rows that are in the **left** result, but **not** in the right.
- Default: duplicates removed.
- With `EXCEPT ALL`:
  - row count in result = `max(m – n, 0)` where:
    - `m` = duplicates in left,
    - `n` = duplicates in right. Lecture4

Multiple `EXCEPT` operators:

- Evaluated **left to right**, unless parentheses are used.

- `EXCEPT` binds at the same level as `UNION`.

ORDER BY

> `ORDER BY` sorts the result rows

```
SELECT ...
FROM ...
[WHERE ...]
[GROUP BY ...]
[HAVING ...]
ORDER BY expression [ASC | DESC] [...];
```

- Rows are sorted by the **leftmost** expression, then by the next, etc.
- If rows are equal according to all sort expressions,

  they appear in some **implementation-dependent** order.

Sort key expression can be:

- name or **ordinal number** of an output column,
- or any expression based on input columns.

Options:

- `ASC` – ascending (default).
- `DESC` – descending.
- `USING operator_name` – use a specific sorting operator.

Null handling:

- `NULLS LAST` – nulls come **after** all non-null values.
- `NULLS FIRST` – nulls come **before** all non-null values.

LIMIT and OFFSET

> Used to restrict **how many rows** are returned and **from where** in the result sequence.

LIMIT

`LIMIT` has two independent parts :

- `count` – **maximum number of rows** to return.
- `start` / `OFFSET` – number of rows to **skip** before returning rows.

Typical PostgreSQL syntax:

```
SELECT ...
FROM ...
ORDER BY ...
LIMIT count OFFSET start;
```

Behavior:

- If both provided: skip `start` rows, then return up to `count` rows.
- If `count` evaluates to `NULL` → treated as `LIMIT ALL` (no limit).
- If `start` evaluates to `NULL` → treated as `OFFSET 0`.

OFFSET (conceptually)

Slides describe `OFFSET` as: skip a specified number of rows before starting to return rows. Often used with `LIMIT` for **pagination**.