Networking today with Cisco

2026 Edition

grimy86

Contents

PostgreSQL & SQL

Introduction

DDL & DML

PostgreSQL uses structured query language, which consists of two core categories.

Data Definition Language (DDL) which defines the structure of the database:

- Creating objects
- Altering objects
- Removing objects

Data Manipulation Langue (DML) which works with the data and only manipulates rows, not structure:

- Inserting data
- Updating data
- Deleting data
- Reading data

PostgreSQL 16 (released September 14, 2023)

Latest major version, offering performance improvements, better replication, and enhanced SQL features.

- **(O)RDBMS (Object-Relational Database Management System)**: PostgreSQL stores data in relational tables and also supports advanced object-like features such as custom data types, functions, and inheritance.
- **Open-source**: Freely available and community-driven, with no licensing cost and strong long-term support.
- **ACID-compliant**, ensures reliable transactions through:
  - **Atomicity**, all operations succeed or fail together
  - **Consistency**, data remains valid after transactions
  - **Isolation**, concurrent transactions do not interfere
  - **Durability**, committed data is not lost, even after crashes
- **Scales from small to large databases**: suitable for anything from small local projects to enterprise-scale systems with large datasets and many concurrent users.
- **Extensible**: supports extensions such as PostGIS (geospatial data), pg_trgm (text search), and many others that add powerful functionality.
- Runs as a **system service (daemon)**: a single PostgreSQL **instance** runs in the background and can manage multiple databases at the same time.
- **Supports multiple schemas (namespaces)**: schemas logically organize database objects (tables, views, functions) and help avoid name conflicts within a database.
- **Users are defined per instance, not per database**: a PostgreSQL user (role) can have access to multiple databases within the same instance.
- Normal users and superusers:
  - **Normal users**: limited privileges
  - **Superusers**: full control over the entire instance (similar to root/admin)
- **Local filesystem storage (PGDATA)**: all database files, configuration files, and write-ahead logs are stored in the **PGDATA directory on disk**.

Data types

- Character types:
  - `VARCHAR`, variable-length character strings with an optional maximum length.
  - `CHAR`, fixed-length character strings, padded with spaces if shorter than n.
- Numerical types:
  - `INTEGER` / `SMALLINT` / `BIGINT`, whole numbers with increasing size and range.
  - `NUMERIC` / `DECIMAL`, exact precision numbers, commonly used for

financial data.
    - `FLOAT`, approximate numeric values, used when performance is more important than exact precision.
- Datetime types:
    - `DATE`, stores a calendar date (year, month, day).
    - `TIME`, stores a time of day.
    - `TIMESTAMP`, stores both date and time (can include time zone support).
- Boolean type:
    - `BOOLEAN`, represents logical values: `TRUE`, `FALSE`, or `NULL`.

SQL statement rules

SQL statements are:

- Not case sensitive
- Can be spread across multiple lines
- Keywords like `DISTINCT` should be written in all-uppercase characters
- Keywords also shouldn't be abbreviated
- Statements are closed with a `;`

It's best practice to start clauses like `FROM`, `WHERE`, `ORDER BY`, etc. on a new line. This improves readability.

Selecting data

`SELECT`, `FROM`

A `SELECT` clause returns data from specific columns, it's followed by a `WHERE` clause where we define from which table the data is needed.

It's by far the most important query in SQL. There's also lots of ways to select data, you could join tables togheter for example.

Examples:

```sql
-- This is a SQL comment
-- Select ALL columns
SELECT *
FROM actors;

-- Select specific columns
SELECT actorid, actorname
FROM actors;

-- Select specific columns and rename them for presentation purposes
-- It is possible to not use strings here with the aliasses but this might result in all lo
-- The AS clause is optional here and provided for readability
SELECT
    actorid AS "Actor Number",
```

```
    actorname AS "Actor Name"
FROM actors;
```

Mathematical expressions

We can use basic PEMDAS operations in sql statements. Additionally we can also use mathemathical functions like `MOD`.

Example:

```
SELECT salary * 12 as "Actor Annual Income"
FROM actors;
```

`NULL` values

A `NULL` value is a special type of value meaning that there's no data in the field. This doesn't mean it's equal to 0, it's simply missing a value.

This is important to be mindful about because mathematical operations like `NULL` division is not allowed.

Text concatenation

The concatenation operator `||` adds two text-based columns togheter. When we want to provide a space or other character between the two concatednated columns we do so using single quotation marks. Double quotes won't work.

Example:

```
-- Note the single quotes
-- Note that without the AS clause the server has no idea what this column should be named
-- This results in ?column? as it's name
SELECT firstname || ' ' || lastname AS "Actor Name"
FROM actors;
```

`DISTINCT`

The `DISTINCT` keyword lets us select unique data, this avoids duplicates.

Example:

```
SELECT DISTINCT name AS "Family Name"
FROM actors;
```

The psql query buffer

In psql (the PostgreSQL terminal), there is something called a **query buffer**. It is temporary memory where the current SQL query is stored before execution.

`\e`:

- Opens the default text editor
- Loads the **last saved query** into the editor
- That query is placed into the **query buffer**
- The **path to the buffer file** is shown in the editor title bar

`\e scriptpath`:

- Opens the editor
- Loads contents of the specified **SQL script file**
- The script is placed into the **query buffer**

Note that backslash `\` commands are not stored in the buffer.

Describe

`\d tablename`:

- Shows the table structure
- Columns
- Data Types
- Constraints
- Primary keys
- Foreign key relationships

`WHERE` & comparison operators

With the `WHERE` clause we can add a condition to the `SELECT` clause. This time when we're selecting text values themselves, they are case-sensitive. Also, we select values using comparison operators:

- `=`
- `!=` or `<>`
- `<`
- `<=`
- `>`
- `>=`
- `BETWEEN n AND n`
- `IN (set)`
- `LIKE`
- `IS NULL`
- `IS NOT NULL`

Example:

```
SELECT *
FROM actors
WHERE actorid < 10;
```

Logical operators

- `AND`
- `OR`
- A combination is also possible: `WHERE n OR n AND n`
- `NOT`

Example:

```
SELECT *
FROM actors
WHERE actorid < 10
AND known_from IS NOT NULL
```

`ORDER BY`

With `ORDER BY` we can effective sort the values in a column.

- `ASC`
- `DESC`
- `NULLS FIRST`
- `NULLS LAST`

Example:

```
SELECT
    actorid AS "Actor ID",
    firstname || ' ' || lastname AS "Name",
    known_from AS "Most notable appearance"
FROM actors
WHERE actorid BETWEEN 0 AND 10
AND known_from IS NOT NULL
ORDER BY "Actor ID" DESC;
```

`LIMIT` & `OFFSET`

With `LIMIT` we limit the amount of output and with `OFFSET` we can skip rows based on a certain output.

Example:

```
SELECT
    actorid AS "Actor ID",
    firstname || ' ' || lastname AS "Name",
    known_from AS "Most notable appearance"
FROM actors
WHERE actorid BETWEEN 0 AND 10
AND known_from IS NOT NULL
ORDER BY "Actor ID" DESC;
LIMIT 3;
```

PSQL variables

PSQL variables are limited to the psql-client. They are processed before SQL goes to the server.

Example:

1. `\set table 'actors'`
2. `SELECT * FROM :table;`

Example with user input:

1. `\set table 'actors'`
2. `\prompt 'What's your name?' name`
3. `SELECT * FROM :table WHERE name = :name;`

If we ever want to free up a variable we can use the `\unset variableName` command.

Single-row functions

Single-row functions take data from one singular row as opposed to multiple-row functions.

Character functions

Character conversions:

- `LOWER(arg)`, all lowercase text as a result
- `UPPER(arg)`, all uppercase text as a result
- `INITCAP(arg)`, Uppercases on the first character in every word.

String manipulation:

- `CONCAT(arg1, arg2, ..., argn)`
- `CONCAT_WS(arg1, arg2, ..., argn)`, adds whitespace?
- `LENGTH(arg)`, results in the amount of characters
- `POSITION(substring in string)`, results in the starting position of the substring
- `STRSPOS(string, substring)`, same as possition but flipped?
- `SUBSTR(string, pos1, pos2)`, returns the substring between pos1 and pos2
- `TRIM([LEADING | TRAILING | BOTH] [character(s)] from string)`, returns a character-trimmed string
- `LPAD(string, length, [, paddingchar])`, padds a string with a specified character to the left
- `RPAD(string, length, [, paddingchar])`, padds a string with a specified character to the right
- `REPLACE(string, toReplace, replacementString)`

Numerical functions

- `ROUND(arg [, precision])`
- `TRUNC(arg [, precision])`
- `MOD(divident, divisor)`

Date functions

- `CURRENT_DATE`
- `CURRENT_TIMESTAMP`
- `NOW()`, returns current date and time
- `DATE_TRUNC(unit, timestamp)`
- `AGE(arg)`

Examples

```sql
--
SELECT created_at, AGE(created_at)
FROM users;

--
SELECT
    id,
    name,
    LENGTH(name) AS name_length
FROM users;

--
SELECT *
FROM users
WHERE LOWER(email) = 'admin@example.com';
```

Nested functions

SQL allows for nested functions which are functions operating on data return from functions.

Example:

```sql
-- We make the last name all uppercase
-- We make the first character of the first name uppercase
-- Then we concatenate both with a space between
SELECT
    CONCAT_WS(' ',INITCAP(first_name), UPPER(last_name)) AS name
FROM employees;
```

Type conversion

There's two types of type conversion, implicit or automatic and explicit. If we want to explicitly convert types then we must use functions, otherwise a coversion happens implicitly.

Implicit:

- Integer + float = float
- String + integer = integer

Explicit:

- `TO_CHAR()`
- `TO_NUMBER()`
- `TO_DATE()`
- `TO_TIMESTAMP()`

Date & time calculations

DATE-type default format: `YYYY-MM-DD` TIME-type default format: `HH:MI:SS` TIMESTAMP-type default format: `YYYY-MM-DD HH:MI:SS`

Calculations:

- date 1 - date 2 = numeric amount of days
- Using `EXTRACT` we can get data from a date
  - `EXTRACT(DOW FROM date)`, retursn the day of the week index (sunday = 0, saturday = 6)
  - `EXTRACT(DAY FROM date)`, returns the day index (1-31)
  - `EXTRACT(MONTH FROM date)`, returns the month index (january = 1, december = 12)
  - `EXTRACT(YEAR FROM date)`, return the year

Example:

```
SELECT TO_DATE('03/09/25', 'DD/MM/YY') - TO_DATE('03/08/25','DD/MM/YY')
AS Difference;
```

Functions with NULL

- `COALESCE(arg1, arg2, ..., arg n)`
  - All arguments must have the same type
  - Loops over the argument trying to find a value that isn't null
- `NULLIF(arg1, arg2)`
  - All arguments must have the same type
  - Compares arg1 to arg2
    - If both are equal it returns NULL
    - If both are different it returns arg1

Examples:

```sql
-- Pick the first non-NULL column
SELECT
    COALESCE(middle_name, first_name) AS display_name
FROM users;

-- Multiple fallbacks
SELECT
    COALESCE(phone_mobile, phone_home, 'No phone') AS contact_number
FROM customers;

-- Prevent NULL in calculations
SELECT
    salary + COALESCE(bonus, 0) AS total_income
FROM employees;

-- Avoid division by zero
SELECT sales / NULLIF(quantity, 0)
FROM orders;

-- Convert placeholder values to NULL
SELECT
    NULLIF(email, '') AS email
FROM users;
```

Conditional expressions

We can make our SQL expressions conditional by using the `CASE` structure.

Structure:

```sql
CASE
    WHEN x THEN return_value_x
    WHEN y THEN return_value_y
    ELSE default_return_value
END
```

Examples:

```sql
-- Simple case
SELECT
    CASE
        WHEN 5 > 3 THEN 'yes'
        ELSE 'no'
    END AS result;

-- Case using table data
SELECT
```

```sql
    name,
    score,
    CASE
        WHEN score >= 90 THEN 'A'
        WHEN score >= 80 THEN 'B'
        WHEN score >= 70 THEN 'C'
        ELSE 'F'
    END AS grade
FROM students;

-- Case in a where clause
SELECT *
FROM orders
WHERE
    CASE
        WHEN status = 'urgent' THEN priority = 'high'
        ELSE priority = 'normal'
    END;

-- Case handling NULLs
SELECT
    name,
    CASE
        WHEN phone IS NULL THEN 'No phone'
        ELSE phone
    END AS contact
FROM customers;
```

Group- or Multiple-row functions

Aggregate group functions

- `AVG(column)`, the average
- `SUM(column)`, the sum
- `MIN(column)`, mimimum value
- `MAX(column)`, maximum value
- `COUNT(column)`, NOT NULL count
- `STDDEV(column)`, standard deviance
- `VARIANCE(column)`, variance

`GROUP BY`

When you use `GROUP BY`, the result will contain one row per group. As a result, **every selected value must be uniquely determined for that group**.

`GROUP BY` merges multiple rows into groups, and then you calculate one result per group.

Key rule: every column in `SELECT` must be either in the `GROUP BY` or inside an aggregate function (`SUM`, `COUNT`, `AVG`, `MAX`, `MIN`).

To recap: We group rows by one or more columns, and we can only select values that produce exactly one result per group.

Example:

```sql
-- How much did each customer spend in total?
SELECT
    customer,
    SUM(amount) AS total_spent
FROM orders
GROUP BY customer;
```

Invalid example:

```sql
-- This example is invalid because one customer can have many amounts
-- SQL wouldn't know which one to show
SELECT customer, amount
FROM orders
GROUP BY customer;
```

`HAVING`

`HAVING` filters groups, not rows. `HAVING` exists because `WHERE` cannot use aggregate functions like `SUM` or `COUNT` because aggregates only make sense after grouping.

Example:

```sql
-- Which customers spent more than 60?
SELECT
    customer,
    SUM(amount) AS total_spent
FROM orders
GROUP BY customer
HAVING SUM(amount) > 60;
```

`JOIN` data from seperate tables togheter

A JOIN combines rows from two tables based on a related column. **Usually by primary key(PK) in one table and foreign key(FK) in the other**.

`NATURAL JOIN`

With a natural join you do not specify the join condition yourself. Using natural joins is **considered malpractice**. NATURAL JOIN is just a **special case of `INNER JOIN` with automatic conditions**.

1. Looks for columns with the same name in both tables
2. Automatically joins on all of those columns
3. Automatically removes duplicate join columns from the result

Example:

```
SELECT emp.employee_id, dep.department_name
FROM employees
NATURAL JOIN departments dep;
```

What SQL does internally:

```
-- Because department_id is the only shared column name.
SELECT employee_id, department_name
FROM employees
JOIN departments
  ON employees.department_id = departments.department_id;
```

`USING` vs. `ON` & `INNER JOIN`

Whenever we write `JOIN` we're effectively using the `INNER JOIN`, they are equivalent. More on join types later.

`USING`:

- Shorter syntax
- Column name must be **identical in both tables**
- The **join column appears only once** in the result

`ON`, best practice:

- Most explicit and flexible
- Works even if column names differ
- Required for complex conditions
- Most common in real code

```
-- USING
SELECT department_id, location_id
FROM departments
JOIN locations
USING (location_id);

-- ON Without aliasing
SELECT department_id, location_id
FROM departments
JOIN locations
ON (departments.location_id = locations.location_id);

-- ON with aliases
SELECT d.department_id, l.location_id
```

```sql
FROM departments AS d
JOIN locations AS l
ON d.location_id = l.location_id;

-- Multi-table JOIN
SELECT emp.employee_id, dep.department_id, loc.location_id
FROM employees emp
JOIN departments dep
  ON emp.department_id = dep.department_id
JOIN locations loc
  ON dep.location_id = loc.location_id;

-- Where clause
SELECT employee_id, department_id, location_id
FROM employees AS emp
JOIN departments AS dep
ON (emp.employee_id = dep.department_id)
WHERE department_id IN (20, 50, 70, 80)
ORDER BY loc.city;
```

`JOIN` Types

1. `JOIN` or `INNER JOIN`: Only matching rows from both tables
2. `LEFT JOIN` or `LEFT OUTER JOIN`: All rows from the left table, plus matching rows from the right.
3. `RIGHT JOIN` or `RIGHT OUTER JOIN`: All rows from the right table, plus matching rows from the left.
4. `FULL JOIN` or `FULL OUTER JOIN`: All rows from both tables
5. `SELF JOIN`: A table joined to itself

Subqueries

A **subquery** is a query nested inside another query.
Conceptually, it answers a **follow-up question** whose result is needed by the main query.

Example logic:

1. What department does John Doe work in?
2. Who else works in that department?

The inner query answers (1), and its result is used by the outer query to answer (2).

General Rules for Subqueries

The number of values returned by the subquery determines **which operators** you can use (`=`, `IN`, `ANY`, `ALL`).

- Subqueries are written **to the right of the comparison operator**
- Subqueries are enclosed in **parentheses `()`**
- Subqueries are **logically evaluated before** the outer query
- A subquery returns **one or more values**, which the outer query uses for filtering

Single-row subqueries

A **single-row subquery** returns **exactly one value**.

Because only one value is returned, you can use **single-value comparison operators**: `=`, `>`, `<`, `>=`, `<=`, `<>`

Aggregate functions (`MIN`, `MAX`, `AVG`) return one value, that makes them perfect for single-row subqueries.

Example:

```sql
-- "Who works in the same department as John Doe (employee_id = 142)?"
SELECT last_name
FROM employees
WHERE department_id =
    (SELECT department_id
     FROM employees
     WHERE employee_id = 142);

-- "Who works in the same department and earns more than John?"
SELECT last_name, salary
FROM employees
WHERE department_id =
    (SELECT department_id
     FROM employees
     WHERE employee_id = 142)
AND salary >
    (SELECT salary
     FROM employees
     WHERE employee_id = 195);

-- "Which employees earn the minimum salary?" (Single-row aggregate)
SELECT last_name, salary
FROM employees
WHERE salary =
    (SELECT MIN(salary)
     FROM employees);
```

Multiple-row subqueries

A **multiple-row subquery** returns **more than one value**.

You cannot use `=` with multiple values. Instead, use operators designed for sets:

- `IN`: Match any value in the result set
- `ANY`: Compare against at least one value
- `ALL`: Compare against every value

Examples:

```sql
-- IN
-- Subquery returns all department IDs where someone named Taylor works
-- Outer query returns employees in any of those departments
SELECT last_name, department_id
FROM employees
WHERE department_id IN
    (SELECT department_id
     FROM employees
     WHERE LOWER(last_name) = 'taylor');

-- ANY
-- Subquery returns all salaries in department 20
-- Condition is true if the employee's salary is less than ANY of those salaries
SELECT last_name, salary
FROM employees
WHERE salary < ANY
    (SELECT salary
     FROM employees
     WHERE department_id = 20);

-- ALL
-- Salary must be less than every salary returned by the subquery
-- Less than the minimum salary in department 20
SELECT last_name, salary
FROM employees
WHERE salary < ALL
    (SELECT salary
     FROM employees
     WHERE department_id = 20);

-- NULL value checking
SELECT last_name
FROM employees
WHERE employee_id NOT IN
    (SELECT manager_id
     FROM employees);
```

DML

DML focuses on how data is inserted, updated, deleted, and safely managed using transactions.

Database Transactions

A **transaction** is a sequence of DML operations that must be executed **as a single unit**: either all changes succeed or none do.

Transaction clauses

- **BEGIN**: Starts a transaction.
- **COMMIT**: Permanently saves all changes.
- **ROLLBACK**: Cancels all changes made in the transaction.
- **SAVEPOINT**: Creates intermediate points to partially roll back.

Transaction behavior

- Changes are **only visible to the user** who started the transaction.
- Modified rows are **locked** for other users.
- Other users can still **read old data (SELECT)**.
- The database keeps both the **original** and **modified** versions until commit or rollback.

COMMIT

- Makes all changes permanent.
- Unlocks affected rows.
- Removes all savepoints.
- Ends the transaction.

ROLLBACK

- Restores the database to its original state.
- Unlocks affected rows.
- Removes all savepoints.
- Ends the transaction.

SAVEPOINT and Partial Rollback

Savepoints allow rollback to a specific point instead of cancelling everything.

Example: `sql SAVEPOINT update1; ROLLBACK TO update1;`

Adding data using `INSERT`

Adds a new row to a table. Values must match all columns in the table, in exact order.

It's important to note that we can handle `NULL` values implicitly by simple omitting the nullable columns from the column list. Or explicitly by providing `NULL` as a value.

Another important note is to use `DEFAULT` for:

- Auto-increment (**SERIAL**) columns
- Columns with predefined default values

And yet another important note is to use `BEGIN` before inserting data if you want the option to `ROLLBACK` in casy anything goes wrong or to `COMMIT` to make the transaction permanent.

Key rules:

- Column order must match value order.
- Text and dates use single quotes.
- Dates use `TO_DATE`.

```
-- Explicit column list, best practice
INSERT INTO table (column1, column2)
VALUES (value1, value2);

-- Implicit column list
-- Values must match all columns in the table, in exact order.
INSERT INTO table
VALUES (value1, value2, ...);
```

Modifying data using `UPDATE`

`UPDATE` updates one or more rows in a table. Again, update using transactions.

Key rules:

- Without `WHERE`, all rows are updated.
- `NULL` is used to remove a column value.
- The number of affected rows depends on the condition.

Removing data using `DELETE` & `SET`

Deletes one or more rows from a table.

Key rules:

- Without `WHERE`, all rows are deleted.
- Foreign key constraints may prevent deletion.
- Deletion is permanent only after `COMMIT`.

Example:

```
-- This employee has been terminated
UPDATE employees
```

```
SET salary = 0
WHERE employee_id = 179;
```

DML subqueries

Subqueries allow DML operations based on data from other tables.

`INSERT`:

```
INSERT INTO table (col1, col2)
SELECT col1, col2
FROM other_table;
```

`UPDATE`:

```
UPDATE employees
SET salary = (
    SELECT salary
    FROM employees
    WHERE employee_id = 204
)
WHERE employee_id = 179;
```

`DELETE`:

```
DELETE FROM table
WHERE column IN (SELECT column FROM other_table);
```

DDL

DDL is used to **create, modify, and remove database structures** such as databases, tables, and constraints in PostgreSQL.

DDL defines the **structure** of a relational database, not the data itself.

Main DDL clauses:

- `CREATE DATABASE`
- `CREATE TABLE`
- `ALTER TABLE`
- `DROP TABLE`
- Datatypes
- Constraints

`CREATE DATABASE`

Creates a new database and assigns an owner.

Important: **Only users with `CREATEDB` rights can create databases.**

Example:

```sql
-- Giving the user "student" the CREATEDB rights
ALTER USER student WITH CREATEDB;

-- Creating the "ClassesDB" databases and asssigning the user "student" as its owner
CREATE DATABASE ClassesDB OWNER student;
```

`CREATE TABLE`

Creates a new table in the database.

We can also combine `CREATE TABLE` with DDL subqueries but it's generally not recommended.

Naming Rules

- Must start with a letter or `_`
- Max 30 characters
- Case-insensitive (stored lowercase)
- Allowed characters: letters, digits, `_`, `$`, `#`
- No reserved SQL keywords (unless quoted with `" "`)

Conventions:

- Table names are **plural** by convention.
- Columns are separated by commas.
- Datatype, default values, and constraints are **defined per column**.

`SERIAL`

`SERIAL` is not a real datatype, but shorthand for an **auto-incrementing sequence**. Commonly used with the `PRIMARY KEY` contraint.

`DEFAULT` Values

`DEFAULT` values are automatically used **when no value is provided in an `INSERT`**.

Example:

```sql
CREATE TABLE department (
    departmentnumber SERIAL,
    departmentname VARCHAR(10),
    postalcode VARCHAR(13),
    startdate DATE DEFAULT CURRENT_DATE
);
```

Constraints

Constraints enforce **data integrity** and restrict invalid data.

Allowed Constraints:

- `PRIMARY KEY`: Exactly one per table, automatically `UNIQUE` and `NOT NULL`
- `FOREIGN KEY`: Creates a relationship to another table's primary key.
- `NOT NULL`: Prevents empty values, automatically applied to primary keys.
- `UNIQUE`: Ensures values appear only once, primary keys are always unique.
- `CHECK`: Restricts allowed values using expressions.

Violating Constraints:

- Inserting or updating invalid data raises an error.
- Deleting a row referenced by a foreign key raises an error.
- Constraints protect consistency and relationships.

`DROP TABLE`

Removes tables permanently.

`DROP TABLE` constraints:

- `IF EXISTS`: avoids error if table does not exist
- `RESTRICT` (default): fails if dependencies exist
- `CASCADE`: also removes dependent objects and constraints

Example:

```
DROP TABLE IF EXISTS deparments;
DROP TABLE IF EXISTS deparments CASCADE;
```

`ALTER TABLE`

Used to modify existing tables.

Common Operations:

- Add columns
- Remove columns
- Modify column definitions
- Add or drop constraints
- Rename columns
- Set or remove default values

Examples:

```
-- Adding a column
ALTER TABLE table_name
ADD COLUMN column datatype;

-- Dropping a constraint
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

```sql
-- Adding a constraint
ALTER TABLE table_name
ADD CONSTRAINT pk_name PRIMARY KEY (column_name);
```

DDL CREATE OBJECT: `VIEW`

A `VIEW` is a virtual table based on a query.

- Stores the SQL definition, not the data.
- Always reflects the current state of the underlying tables.
- Simplifies complex queries and improves readability.

Examples:

```sql
-- Simple view
CREATE OR REPLACE VIEW lonen_dep90_vu AS
SELECT employee_id AS id,
       last_name AS naam,
       salary AS maandsalaris
FROM employees
WHERE department_id = 90;

-- Querying a view
SELECT * FROM lonen_dep90_vu;

-- Complex view (Calculated Columns)
CREATE OR REPLACE VIEW jaarlonen_dep90_vu AS
SELECT employee_id AS id,
       last_name AS naam,
       12 * salary AS jaarsalaris
FROM employees
WHERE department_id = 90;

-- Complex view (Aggregation & Join)
CREATE OR REPLACE VIEW job_sum_vu (naam, loontotaal) AS
SELECT j.job_title,
       SUM(e.salary)
FROM employees e
JOIN jobs j USING (job_id)
GROUP BY j.job_title;
```

`DROP VIEW`

Removes a view definition.

Example:

```sql
DROP VIEW view_name;
```