TP No. 1 — Part 3: PostgreSQL tutorial

(Pizzerias)

M. Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)

University of Skikda — Department of Computer Science

1st Year Master RSD/AI

Advanced Databases (BDDA)

Updated on 2026-09-22 at 22:27:14

Outline

Working instructions

Objective

- Learn the basics of the SQL language using the `Pizzerias` model.
- Understand the steps involved in creating, manipulating and querying a relational database.

General instructions

- Run each command separately in `psql`.
- Read the message displayed by PostgreSQL:
  - Success → move on to the next step.
  - Error → analyze, fix, then try again.
- Lines starting with `-` are explanatory comments: do not ignore them.

Logical order of work

- `CREATE ROLE` → `CREATE DATABASE` → `CREATE TABLE` → `INSERT INTO` → `SELECT`
- Keep a copy of the corrected script for validation.

Connecting with psql

Connecting to PostgreSQL on Windows

```
Win + R -> cmd
Microsoft Windows [Version 10.0.17763.1697]
(C) 2018 Microsoft Corporation. All rights reserved.

C:\Users\danny>chcp 1252
Active code page: 1252

C:\Users\danny>psql -U postgres
Password for user postgres:
```

Client-side configuration (in psql)

```
psql (16.10)
Type "help" for help.

postgres=# \encoding UTF8
postgres=# SHOW client_encoding;
 client_encoding
-----------------
 UTF8
(1 row)
```

Note: UTF8 makes it possible to display accented characters correctly.

Resetting the state

Cleaning up the environment (in psql)

```
-- First go back to the default database "postgres"
postgres=# \c postgres
You are now connected to database "postgres" as user "postgres".

-- Drop the sample database if an old one remains
postgres=# DROP DATABASE pizzerias;
ERROR:  database "pizzerias" does not exist  -- normal message if already
     dropped

-- Drop the role used for testing
postgres=# DROP ROLE laidi;
ERROR:  role "laidi" does not exist          -- normal message if already
     dropped
```

Note: the `postgres=#` prompt indicates that you are connected as a superuser.

Role, database and privileges

```
-- Create a user role with a password (login allowed)
CREATE ROLE laidi LOGIN PASSWORD '1234';

-- Create the pizzerias database owned by the role "laidi"
CREATE DATABASE pizzerias OWNER laidi;

-- Connect to check
postgres=# \c pizzerias
You are now connected to database "pizzerias" as user "postgres".
```

Since PostgreSQL 15: nothing else to do

The `public` schema of a database now belongs to the predefined role `pg_database_owner`, of which the **database owner** is automatically a member. Since `laidi` owns `pizzerias`, it already has the `CREATE` privilege on `public` — **no `GRANT` or `ALTER SCHEMA` is needed**.

Before PostgreSQL 15, the `public` schema belonged to `postgres` and you had to explicitly run `GRANT ALL ON SCHEMA public TO laidi;` or `ALTER SCHEMA public OWNER TO laidi;` — which is what you will still see in many outdated online tutorials.

Working session

Close the current session and reconnect as the `laidi` user.

1. Leave the superuser session (in `psql`)

```
postgres=# \q
```

2. Open a new session as `laidi` (in `cmd`)

```
C:\> psql -U laidi -d pizzerias
Password for user laidi:
psql (16.10)
Type "help" for help.
```

3. Check the active connection (in `psql`)

```
pizzerias=> \conninfo
You are connected to database "pizzerias" as user "laidi" on host "localhost" (
     address "::1") at port "5432".
```

Note: all the following commands now run as `laidi`.

Character strings

Definition

Standard SQL:

- `CHAR(n)` — fixed-length text
- `VARCHAR(n)` — variable-length text
- `CLOB` — large text (Character Large Object)

Usage

- `CHAR(n)` for short, normalized codes (e.g. countries, categories).
- `VARCHAR(n)` for labels and ordinary text fields.
- `CLOB` for long content (comments, detailed descriptions).

Non-standard (to avoid): `TEXT`, `LONGVARCHAR`.

Tip: set consistent sizes for `VARCHAR(n)` to improve sorting and indexing.

Numeric types

Definition

Main categories:

- **Integers:** `SMALLINT`, `INTEGER`, `BIGINT`.
- **Exact decimals:** `DECIMAL(p,s)`, `NUMERIC(p,s)` — used for amounts or stock.
- **Floating point:** `REAL`, `DOUBLE PRECISION` — for physical measurements or scientific computations.

Tip: prefer `DECIMAL/NUMERIC` for precision in financial calculations.

Dates and times

Definition

Standard temporal types:

- `DATE` — day, month, year
- `TIME` — time of day
- `TIMESTAMP` — combined date and time
- `INTERVAL` — duration or time difference

Usage:

- `TIMESTAMP` to record events.
- `INTERVAL` for duration calculations.

Tip: prefer `INTERVAL` to integers counting "minutes" or "seconds".

Auto-increment (IDENTITY)

Definition

Standard SQL: `GENERATED AS IDENTITY`.

Modes:

- `ALWAYS` — the value is always generated automatically.
- `BY DEFAULT` — the value is generated unless supplied manually.

PostgreSQL extension (non-standard): `SERIAL`.

Declaration examples

```
id INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY
id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY
```

Tip: use `IDENTITY` for portable SQL scripts.

Integrity constraints

Role of constraints

In SQL, **integrity constraints** ensure the validity, consistency and reliability of data. They are checked directly by the DBMS engine during `INSERT`, `UPDATE` or `DELETE` operations, guaranteeing that the rules defined in the schema are never violated.

Types of constraints (standard SQL)

- `NOT NULL` — forbids missing values.
- `UNIQUE` — prevents duplicates.
- `PRIMARY KEY` — uniquely identifies each row.
- `FOREIGN KEY` — creates a dependency link between two tables.
- `CHECK` — imposes a logical condition on the values.
- `CONSTRAINT` — explicitly names a constraint.

Note: constraints are an integral part of the relational model.

Integrity constraints(continued)

Examples of constraints

```
CONSTRAINT uq_email UNIQUE(email)
FOREIGN KEY(client_id) REFERENCES client(id)
CONSTRAINT ck_price_positive CHECK(price >= 0)
```

- Name each constraint clearly to make the schema easier to maintain and read.
- Define `ON DELETE` and `ON UPDATE` behaviors according to the business logic.
- Prefer SQL constraints to checks in application code.
- Place constraints at the end of `CREATE TABLE` statements for readability.

Tip: specify the foreign-key actions — e.g. `ON DELETE CASCADE` or `SET NULL`.

Creating the person table

Version 1 – Direct writing (compact)

```
CREATE TABLE person (
  name   VARCHAR(20) PRIMARY KEY,
  age    SMALLINT CHECK (age BETWEEN 0 AND 120),
  gender CHAR(1) CHECK (gender IN ('M','F'))
);
```

Version 2 – Structured writing (preferred)

```
CREATE TABLE person (
  name    VARCHAR(20),
  age     SMALLINT,
  gender  CHAR(1),

  -- Structural constraints grouped at the end
  CONSTRAINT person_pkey PRIMARY KEY (name),
  CONSTRAINT person_age_ck    CHECK (age BETWEEN 0 AND 120),
  CONSTRAINT person_gender_ck CHECK (gender IN ('M','F'))
);
```

Creating the `frequents` table

Version 1 – Direct writing (compact)

```
CREATE TABLE frequents (
  name      VARCHAR(20) NOT NULL,
  pizzeria  VARCHAR(30) NOT NULL,
  PRIMARY KEY (name, pizzeria),
  FOREIGN KEY (name) REFERENCES person(name)
);
```

Version 2 – Structured writing (preferred)

```
CREATE TABLE frequents (
  name      VARCHAR(20) NOT NULL,
  pizzeria  VARCHAR(30) NOT NULL,

  -- Structural constraints grouped at the end
  CONSTRAINT frequents_pkey PRIMARY KEY (name, pizzeria),
  CONSTRAINT frequents_name_fkey FOREIGN KEY (name)
    REFERENCES person(name)
);
```

Creating the eats table

Version 1 – Direct writing (compact)

```
CREATE TABLE eats (
  name  VARCHAR(20) NOT NULL,
  pizza VARCHAR(60) NOT NULL,
  PRIMARY KEY (name, pizza),
  FOREIGN KEY (name) REFERENCES person(name)
);
```

Version 2 – Structured writing (preferred)

```
CREATE TABLE eats (
  name   VARCHAR(20) NOT NULL,
  pizza  VARCHAR(60) NOT NULL,

  -- Structural constraints grouped at the end
  CONSTRAINT eats_pkey PRIMARY KEY (name, pizza),
  CONSTRAINT eats_name_fkey FOREIGN KEY (name)
    REFERENCES person(name)
);
```

Creating the serves table

Version 1 – Direct writing (compact)

```
CREATE TABLE serves (
  pizzeria VARCHAR(30) NOT NULL,
  pizza    VARCHAR(60) NOT NULL,
  price    NUMERIC(10,2) CHECK (price >= 0),
  PRIMARY KEY (pizzeria, pizza)
);
```

Version 2 – Structured writing (preferred)

```
CREATE TABLE serves (
  pizzeria  VARCHAR(30) NOT NULL,
  pizza     VARCHAR(60) NOT NULL,
  price     NUMERIC(10,2),

  -- Structural constraints grouped at the end
  CONSTRAINT serves_pkey PRIMARY KEY (pizzeria, pizza),
  CONSTRAINT serves_price_ck CHECK (price >= 0)
);
```

Minors and the pizzerias they frequent

Task: return the list of pizzerias frequented by at least one person under 18.

```sql
-- Join `frequents` (name <-> pizzeria) with `person` (name <-> age)
-- then filter on age < 18. DISTINCT avoids duplicate pizzerias.

SELECT DISTINCT f.pizzeria
FROM frequents f
JOIN person p ON p.name = f.name
WHERE p.age < 18;
```

Women eating mushroom or tuna

Task: distinct names of people of gender 'F' who eat 'mushroom' or 'tuna'.

```sql
-- Gender filter on `person`, then IN membership test on `eats`.

SELECT DISTINCT p.name
FROM person p
JOIN eats m ON m.name = p.name
WHERE p.gender = 'F'
  AND m.pizza IN ('mushroom','tuna');
```

Women eating both pizzas

Task: names of the women who eat both 'mushroom' and 'tuna'.

```sql
-- Group by person and require two distinct pizzas from the target set.

SELECT p.name
FROM person p
JOIN eats m ON m.name = p.name
WHERE p.gender = 'F'
  AND m.pizza IN ('mushroom','tuna')
GROUP BY p.name
HAVING COUNT(DISTINCT m.pizza) = 2;
```

Amy's pizzerias at $<$ 1000 DZD

Task: pizzerias serving at least one pizza that Amy eats at a price strictly $<$ 1000 DZD.

```sql
-- `eats` links Amy to her pizzas; `serves` links pizzerias and pizzas with
     prices.

SELECT DISTINCT s.pizzeria
FROM serves s
JOIN eats m ON m.pizza = s.pizza
WHERE m.name = 'Amy'
  AND s.price < 1000;
```

Single-gender pizzerias

Task: pizzerias whose customers are all of the same gender (M or F).

```sql
-- A pizzeria is single-gender if COUNT(DISTINCT p.gender) = 1 within its group.

SELECT f.pizzeria
FROM frequents f
JOIN person p ON p.name = f.name
GROUP BY f.pizzeria
HAVING COUNT(DISTINCT p.gender) = 1;
```

Pizzas not served in their pizzerias

Task: (name, pizza) pairs such that no pizzeria frequented by the person serves that pizza.

```sql
-- NOT EXISTS pattern: keep the pair if no frequented pizzeria offers it.

SELECT m.name, m.pizza
FROM eats m
WHERE NOT EXISTS (
  SELECT 1
  FROM frequents f
  JOIN serves s ON s.pizzeria = f.pizzeria
               AND s.pizza    = m.pizza
  WHERE f.name = m.name
);
```

Consistent pizzerias

Task: people who only frequent pizzerias serving at least one of their pizzas.

```sql
-- For every frequented pizzeria, one of the person's pizzas must be served there
      .

SELECT p.name
FROM person p
WHERE NOT EXISTS (
  SELECT 1
  FROM frequents f
  WHERE f.name = p.name
    AND NOT EXISTS (
      SELECT 1
      FROM serves s
      JOIN eats m ON m.name = p.name AND m.pizza = s.pizza
      WHERE s.pizzeria = f.pizzeria
    )
);
```

Frequent all relevant pizzerias

Task: people who frequent every pizzeria serving at least one pizza they eat.

```sql
-- "For every pizzeria serving one of the person's pizzas, the person frequents
     it" (double NOT EXISTS).

SELECT p.name
FROM person p
WHERE NOT EXISTS (
  SELECT 1
  FROM serves s
  JOIN eats m ON m.name = p.name AND m.pizza = s.pizza
  WHERE NOT EXISTS (
    SELECT 1
    FROM frequents f
    WHERE f.name = p.name
      AND f.pizzeria = s.pizzeria
  )
);
```

Minimum price for 'tuna'

Task: pizzerias offering the 'tuna' pizza at the lowest observed price.

```sql
-- Compare the price of 'tuna' with the overall minimum for that pizza.

SELECT s.pizzeria
FROM serves s
WHERE s.pizza = 'tuna'
  AND s.price  = (
    SELECT MIN(price)
    FROM serves
    WHERE pizza = 'tuna'
  );
```