Lab 1 — Part 1: Getting started with PostgreSQL

(On Windows)

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 12:44:03

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

Creative Commons License — BY-NC-SA 4.0 International

Outline

1. What is PostgreSQL?
2. Installing PostgreSQL
3. Installation
4. PostgreSQL concepts
5. Tutorial (1/4): preparing the environment
6. Tutorial (2/4): creating the database
7. Tutorial (3/4): querying the database
8. Tutorial (4/4): modifying and administering
9. Summary

What is PostgreSQL?

PostgreSQL is an **Object-Relational Database Management System (ORDBMS)**, derived from the **POSTGRES (v4.2)** university project developed at the University of California, Berkeley.

Main characteristics:

- Open-source heir to the original code of the Berkeley academic project.
- Compliant with a large part of the **SQL** standard (ISO/IEC 9075).
- Offers advanced features:
  - Complex queries, foreign keys, *triggers*.
  - Updatable views and full transactional integrity.
  - Multiversion concurrency control (**MVCC**).

Extensibility: PostgreSQL makes it possible to add new **data types**, **functions**, **operators**, **index methods**, as well as custom **procedural languages** (PL/pgSQL, PL/Python, etc.).

PostgreSQL license

Distributed under a **free and permissive** license, PostgreSQL can be used, modified and redistributed without restriction, for *private*, *academic* or *commercial* purposes.

PostgreSQL client/server architecture

PostgreSQL follows a **client/server** model: the client sends SQL queries, the server interprets them, performs the requested operations and returns the results.

Client side: Clients can be interactive tools (`psql`, **pgAdmin**), web applications or scripts. They communicate with the server over the **TCP/IP** protocol, using a default port (5432).

Server side: The main `postgres` process listens for incoming connections. When a client connects, it creates a **dedicated child process** that handles that session exclusively. This process receives the query and passes it to the **parser**, which turns it into a syntax tree, then to the **planner/optimizer**, which determines the most efficient execution strategy. Finally, the **executor** applies this plan and returns the results to the client.

Advantage: This architecture allows **concurrent**, stable and secure operation, where each session is isolated, but all of them share the same data files managed by the main server.

Installing PostgreSQL 16

Download `postgresql-16.x-windows-x64.exe` from https://www.postgresql.org/download/windows, then right-click → **Run as administrator**.

Wizard screen	**What to do**
Installation Directory	keep `C:\Program Files\PostgreSQL\16`
Select Components	PostgreSQL Server, pgAdmin 4, Command Line Tools
Data Directory	keep `...\16\data`
Password	password of `postgres`: **write it down**
Port	5432
Completing	uncheck "Stack Builder", then **Finish**

The installer creates the Windows service `postgresql-x64-16`.

Checking that the server is running

What is it?

- **Service**: a program that runs in the background, without a window, started when Windows boots. The PostgreSQL server is the service `postgresql-x64-16`.
- **Command Prompt** (`cmd`): a window where commands are typed on the keyboard. To open it: Windows key + `R`, type `cmd`, Enter.
- `sc`: a Windows command that displays the state of a service.

In the Command Prompt, type:

```
C:\Users\ahmed> sc query postgresql-x64-16
        STATE              : 4  RUNNING
```

If the `STATE` line shows `RUNNING`, the server is running: everything is fine.

If the server is stopped: start it

If the `STATE` line shows `STOPPED`, the service must be started. This requires Windows **administrator** rights:

1. Windows key, type `cmd`.
2. Right-click on **Command Prompt** → **Run as administrator**.
3. In this window, type:

```
C:\WINDOWS\system32> net start postgresql-x64-16
The postgresql-x64-16 service was started successfully.
```

What is it?

`net start`: a Windows command that starts a service.

Making psql available from the Command Prompt

What is it?

PATH: the list of folders where Windows looks for a program when its name is typed. The psql folder is not in it at first.

```
C:\Users\ahmed> psql --version
'psql' is not recognized as an internal or external command...
```

Add the psql folder to the PATH with the setx command (only once):

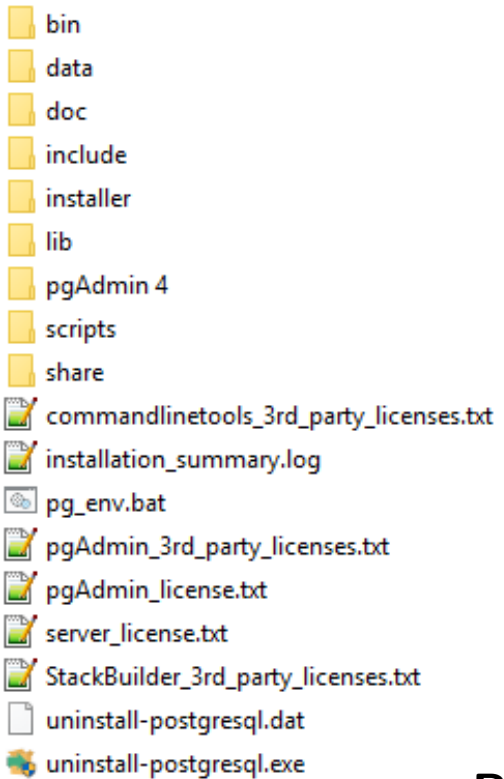
```
C:\Users\ahmed> setx PATH "%PATH%;C:\Program Files\PostgreSQL\16\bin"
SUCCESS: Specified value was saved.
```

Close the Command Prompt, open a **new** one, then check:

```
C:\Users\ahmed> psql --version
psql (PostgreSQL) 16.4
```

Installation contents

The contents of the installation look like this:



- **bin/**: main executables.
- **data/**: data directory (PostgreSQL cluster).
- **doc/**: local documentation.
- **include/**: header files for development.
- **lib/**: dynamic libraries (DLL).
- **pgAdmin 4/**: graphical administration interface.
- **scripts/**: utility scripts.
- **share/**: shared files (init, dictionaries, templates).
- **StackBuilder/**: installer for additional extensions.

Related files: `installation_summary.log`, `pg_env.bat`, `server_license.txt`, `uninstall-postgresql.exe`, etc.

The bin folder — Main executables

The **bin/** directory gathers the essential programs used to **start, administer and interact** with the PostgreSQL server.

- **postgres.exe** — starts the main server.
- **psql.exe** — command-line client.
- **pg_ctl.exe** — starts, stops or restarts the server.
- **initdb.exe** — initializes a database cluster.
- **pg_dump(.exe)** / **pg_dumpall(.exe)** — export and backup.
- **pg_restore.exe** — restore from a backup.
- **createdb(.exe)** / **dropdb(.exe)** — create and drop databases.
- **createuser(.exe)** / **dropuser(.exe)** — manage user roles.
- **vacuumdb.exe** — cleaning and optimization of tables.
- **pgAdmin4.exe** — graphical administration interface.

These executables are the basis of all PostgreSQL administration on Windows.

The data/ folder: the cluster

`C:\Program Files\PostgreSQL\16\data` contains **all** the databases and the server configuration. Never touch it by hand.

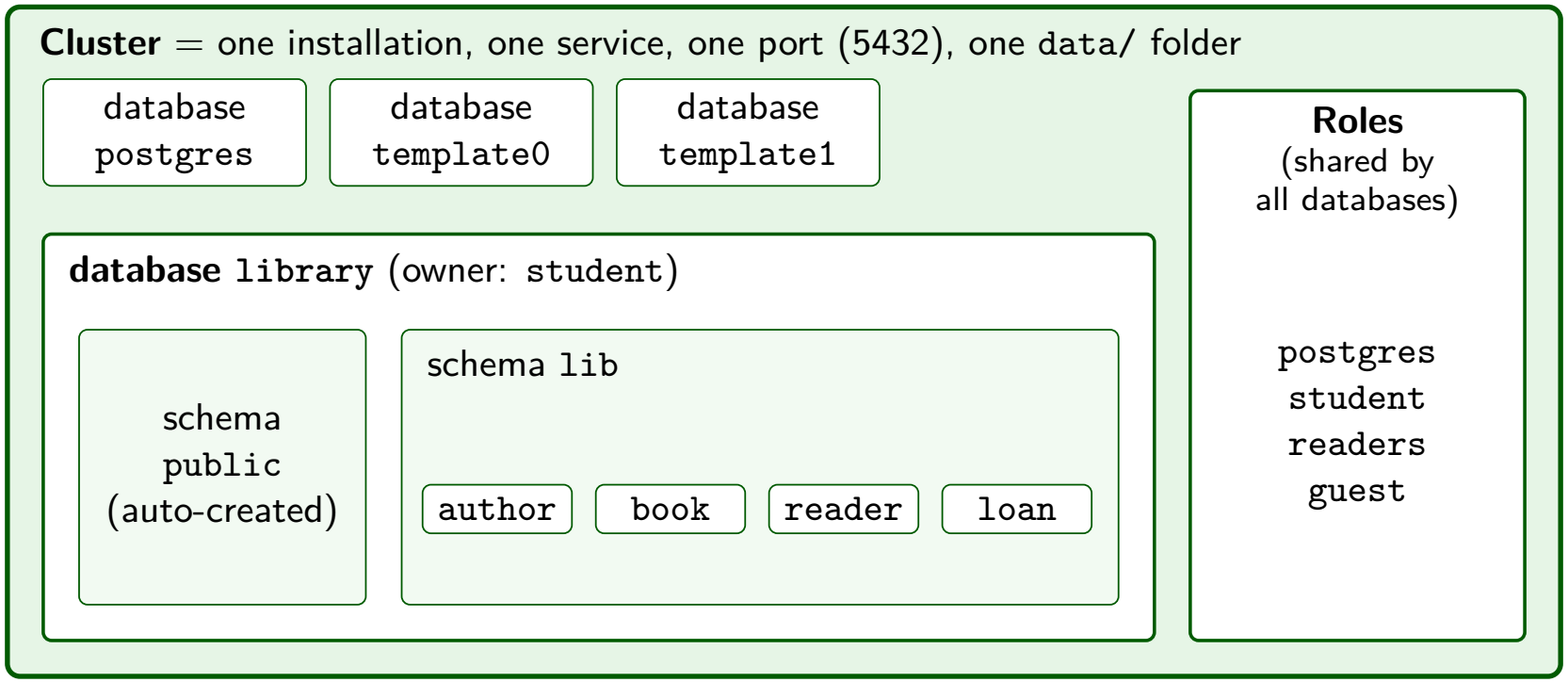
Subfolders

- `base/`: the data of the databases;
- `global/`: the roles;
- `pg_wal/`: the transaction log;
- `log/`: the server logs.

Configuration files

- `postgresql.conf`: settings (port, memory...);
- `pg_hba.conf`: who may connect, and how.

How PostgreSQL organizes data



- **Cluster** ⊃ **databases** ⊃ **schemas** ⊃ **objects** (tables, views. . .).
- **Roles** are shared by all the databases of the cluster.

(Example: the database built in the "Tutorial".)

The postgres database

Created automatically during installation.

- It is the database you connect to **by default**, to **administer**: create roles and databases.
- You do **not** store your tables in it: each project has its own database.
- Never drop it: it always lets you connect to the server.

The postgres role

Created automatically during installation: it is the **superuser**.

- It has **all privileges** on the whole server.
- It is used **only for administration**: creating roles and databases.
- To work, you use an ordinary role that owns its database (Tutorial, step 3).

The template databases `template0` and `template1`

A new database is always a **copy** of a template database:

- `template1`: the template used **by default**;
- `template0`: a pristine template, never modified.

```
CREATE DATABASE db_name;                          -- copy of template1
CREATE DATABASE db_name TEMPLATE template0;       -- copy of template0
```

You never work in these two databases.

Roles in PostgreSQL

In **PostgreSQL**, there is no distinction between **user** and **group**: everything is represented by a **role** (`ROLE`).

Fundamental principle

A role can:

- connect to the server (like a user);
- own objects: databases, tables, views, functions, etc.;
- receive privileges or grant them to other roles;
- group other roles (group function).

Login roles (LOGIN)

A **login role** is a role allowed to connect to the **PostgreSQL** server. It corresponds to a **regular user** of the system.

```
-- Creating a login role (with a password)
CREATE ROLE student LOGIN PASSWORD 'student';

-- Equivalent: CREATE USER = CREATE ROLE ... LOGIN
CREATE USER student PASSWORD 'student';
```

- Can open a session through `psql`, `pgAdmin`, etc.
- Has its own privileges and objects.
- Generally used for application connections.

Example

The tutorial role `student`, owner of the `library` database, is a login role.

Roles without login (NOLOGIN)

A **role without login** does not have the LOGIN attribute. It is used to **group several users** in order to manage their privileges collectively.

```
-- Creating a "group" role
CREATE ROLE readers NOLOGIN;

-- The users guest and visitor become members of the group
GRANT readers TO guest, visitor;
```

- Cannot connect directly to the server.
- Simplifies privilege management (inheritance principle).
- All members inherit the privileges granted to the group.

Example

If the role readers can read a table, then guest and visitor can too.

[1/2] Schemas — Concept and purpose

Definition

A **schema** is a **logical namespace** inside a database. It groups objects — **tables**, **views**, **functions**, **sequences** — to structure and isolate the components of a single project.

A schema makes it possible to:

- **Organize** the database by functional area (for example: *application* (`app`), *reference data* (`ref`), *audit* (`audit`), *log* (`log`), etc.).
- **Separate** privileges and responsibilities according to roles or users.
- **Simplify** management and avoid object name conflicts.

Key idea

A schema acts as a **logical folder** inside a single database: it brings **clarity**, **structure** and **security**, without requiring several separate databases.

[2/2] The default schema — `public`

When a database is created, PostgreSQL automatically adds a schema named **public** to it. Since PostgreSQL 15, only the owner of the database (role `pg_database_owner`) can create objects in it; the other roles only have `USAGE`.

Example

```
-- Implicitly creates the table in the "public" schema
CREATE TABLE clients (id SERIAL PRIMARY KEY, name TEXT);

-- Explicit equivalent
CREATE TABLE public.clients (id SERIAL PRIMARY KEY, name TEXT);
```

Note

If no schema is specified, PostgreSQL uses **public**. For a structured project, it is advisable to create your own application schemas.

The tutorial: a "library" database

We build, step by step, the database of a small **library**: **authors** write **books**, which **readers borrow**.

Table	Columns
`author`	author_id, last_name, first_name, country, birth_year
`book`	book_id, title, *author_id*, genre, pub_year, pages, price
`reader`	reader_id, last_name, first_name, city, email, registered, active
`loan`	loan_id, *book_id*, *reader_id*, loan_date, return_date

Underlined: primary key. *Italics*: foreign key.

Business rules and working method

Business rules

- A book has a single author; an author has written several books, or none.
- A reader borrows several books; a book is borrowed several times.
- A loan without a return date is **ongoing**.

Method

- Type the commands **one at a time**.
- Read PostgreSQL's answer and compare it with the expected result.
- Fix any error **before** going on.

1 Opening the Command Prompt

Windows key + `R`, type `cmd`, Enter. Then:

```
C:\Users\ahmed> chcp 1252
Active code page: 1252
```

What is it?

- **Command Prompt** (`cmd`): a window where commands are typed on the keyboard. The text before > (`C:\Users\ahmed`) is the current folder.
- `chcp 1252`: sets the character encoding of the window, so that **accented characters** (é, è, à, ç) are displayed correctly.

To be done again in every new window.

1 Connecting with psql

```
C:\Users\ahmed> psql -h localhost -p 5432 -U postgres -d postgres
Password for user postgres:
psql (16.4)
Type "help" for help.

postgres=#
```

What is it?

`psql`: the program that connects to the PostgreSQL server and sends it our commands.

- `-h` server (`localhost` = this PC), `-p` port, `-U` role, `-d` database.
- The password is not displayed while you type it: this is normal.

1 Checking the connection

An SQL command **always** ends with a semicolon:

```
postgres=# SHOW client_encoding;
 client_encoding
-----------------
 WIN1252
(1 row)

postgres=# \conninfo
You are connected to database "postgres" as user
"postgres" on host "localhost" at port "5432".
```

- `postgres=#`: database `postgres`, superuser (`#`).
- `\conninfo`: at any time, "where am I connected?".

1 The psql meta-commands

They start with \ and have **no** semicolon:

\l	list the databases	\du	list the roles
\dn	list the schemas	\dt	list the tables
\d t	describe table t	\dv	list the views
\c b r	go to database b as role r	\dp	privileges on tables
\i f	run the script f	\x	vertical display
\h SELECT	help on an SQL command	\?	psql help
\q	quit psql		

1 Reading the psql prompt

The prompt shows the state psql is in:

`postgres=#`	ready, as superuser
`library=>`	ready, with an ordinary role
`postgres-#`	command **not finished**: the ; is missing
`postgres'#`	a quote is not closed
`library=*>`	a transaction is in progress

psql no longer responds?

Look at the prompt. Type ; then Enter, or **Ctrl+C** to abort.

2 Cleaning up before starting

To be able to **redo the tutorial** from the beginning, we remove whatever a previous session may have left behind (as `postgres`):

```
DROP DATABASE IF EXISTS library;
DROP ROLE IF EXISTS guest;
DROP ROLE IF EXISTS readers;
DROP ROLE IF EXISTS student;
```

The first time, nothing exists: just a notice, not an error.

```
NOTICE:  database "library" does not exist, skipping
```

Warning

`DROP` is **permanent**: there is no recycle bin.

3 The right way to work on a database

The superuser `postgres` prepares the ground, then we no longer use it:

1. `postgres` **creates the user** `student`;
2. `postgres` **creates the database** `library` and gives it to `student`;
3. `postgres` gives `student` **all privileges** on the database;
4. `postgres` **revokes the privileges** of all other roles;
5. from then on, we work **only** as `student`.

Result

Only the owner, `student`, works on the database. A handling mistake cannot damage the rest of the server.

3 Creating the user

Connected as `postgres`:

```
CREATE ROLE student LOGIN PASSWORD 'student';
```

Answer: `CREATE ROLE`.

- `LOGIN`: this role can connect (it is a user).
- `PASSWORD`: its password.

```
postgres=# \du
 Role name |                         Attributes
-----------+------------------------------------------------------------
 postgres  | Superuser, Create role, ...
 student   |
```

`student` has **no** special power: this is intentional.

3 Creating the database and granting all privileges

Still as `postgres`:

```
CREATE DATABASE library OWNER student;
GRANT ALL PRIVILEGES ON DATABASE library TO student;
```

Answers: `CREATE DATABASE`, then `GRANT`.

- `OWNER student`: `student` is the **owner** of the database.
- `GRANT ALL PRIVILEGES`: it receives all privileges on the database (connecting to it, creating schemas in it. . .).

3 Revoking the privileges of the other roles

```
REVOKE ALL ON DATABASE library FROM PUBLIC;

\c library postgres
ALTER SCHEMA public OWNER TO student;
REVOKE ALL ON SCHEMA public FROM PUBLIC;
```

What is it?

`PUBLIC`: **all roles**. By default, everyone can connect to a new database; `REVOKE ... FROM PUBLIC` removes this privilege.

- The `public` schema of the database now belongs to `student`.
- Nobody else can connect to the database any more, nor use `public`.

3 Checking the privileges on the database

```
postgres=# \l library
  Name   |  Owner  | ... |  Access privileges
---------+---------+-----+---------------------
 library | student | ... | student=CTc/student
```

- `student=CTc`: `student` can create (`C`), create temporary tables (`T`) and connect (`c`).
- No other role appears: the others no longer have any privilege.

What about `postgres`?

A superuser **always** keeps all privileges: they cannot be revoked. That is why we **no longer use it** to work on the database.

3 Working as the owner

Switch role with `\c database role`:

```
postgres=# \c library student
Password for user student:
You are now connected to database "library" as user
"student".
library=>
```

- The prompt ends with >: you are no longer a superuser.
- In the following sessions, directly: `psql -U student -d library`.
- **The rest of the tutorial is done as `student`**, unless stated otherwise.

4 Creating a schema

The application objects are stored in a **dedicated schema** rather than in `public`:

```
CREATE SCHEMA lib;
```

```
library=> \dn
  List of schemas
  Name   |  Owner
---------+---------
 lib     | student
 public  | student
(2 rows)
```

The `lib` schema belongs to `student`, who created it.

4 Setting the search path (search_path)

```
-- for future connections
ALTER DATABASE library SET search_path TO lib, public;
-- for the current session
SET search_path TO lib, public;
SHOW search_path;                          -- lib, public
```

The search_path is the list of schemas searched for a name **without a schema**:

- CREATE TABLE book creates lib.book;
- FROM book reads lib.book;
- the full name lib.book always works.

5 Basic types: text and numbers

Type	Use	Example
`VARCHAR(n)`	text of at most `n` characters	`'Kateb'`
`CHAR(n)`	fixed-length text	`'F'`
`TEXT`	text without limit	`'Once upon...'`
`SMALLINT`	small integer	`1956`
`INTEGER`	integer	`1488`
`NUMERIC(p,s)`	exact number (price)	`950.00`
`REAL`	approximate number (measurement)	`3.14`

- Text between **single quotes**; a quote inside a text: `'L"Incendie'`.
- Decimal separator: the **dot**. For money: `NUMERIC`.

5 Basic types: dates, booleans, identifiers

Type	Use	Example
`DATE`	a date	`'2026-10-15'`
`TIMESTAMP`	a date and a time	`'2026-10-15 08:30'`
`INTERVAL`	a duration	`INTERVAL '14 days'`
`BOOLEAN`	true or false	`TRUE`, `FALSE`
`... GENERATED ALWAYS AS IDENTITY`	identifier numbered 1, 2, 3...	(automatic)

- Dates in **YYYY-MM-DD** format; `CURRENT_DATE` = today.
- `NULL` (without quotes) = **unknown** value, for all types.

5 Integrity constraints

Constraint	Effect
`NOT NULL`	mandatory value
`UNIQUE`	never the same value twice
`PRIMARY KEY`	identifies each row (unique and not null)
`FOREIGN KEY ... REFERENCES`	the value must exist in the other table
`CHECK (condition)`	the condition must be true
`DEFAULT value`	value used if none is given

5 Writing and naming a constraint

Two possible forms:

```
-- 1) on the column
price NUMERIC(8,2) CHECK (price >= 0)

-- 2) separately, named with CONSTRAINT (at the end of the table)
CONSTRAINT book_price_ck CHECK (price >= 0)
```

The second form is preferred: the name of the constraint appears in error messages.

Naming convention of the tutorial

`table_pk`, `table_xxx_fk`, `table_xxx_uq`, `table_xxx_ck`

6 Creating the author table

```
CREATE TABLE author (
  author_id   INTEGER GENERATED ALWAYS AS IDENTITY,
  last_name   VARCHAR(40) NOT NULL,
  first_name  VARCHAR(40),
  country     VARCHAR(30) NOT NULL DEFAULT 'Algeria',
  birth_year  SMALLINT,
  CONSTRAINT author_pk PRIMARY KEY (author_id),
  CONSTRAINT author_birth_ck
    CHECK (birth_year BETWEEN 1000 AND 2100)
);
```

Expected answer: `CREATE TABLE`.

- `author_id`: numbered automatically, primary key.
- `country`: equals `'Algeria'` if it is not specified.
- Comma between lines, **not** after the last one.

6 Creating the book table

```
CREATE TABLE book (
  book_id    INTEGER GENERATED ALWAYS AS IDENTITY,
  title      VARCHAR(120) NOT NULL,
  author_id  INTEGER NOT NULL,
  genre      VARCHAR(20) NOT NULL,
  pub_year   SMALLINT,
  pages      INTEGER,
  price      NUMERIC(8,2),
  CONSTRAINT book_pk PRIMARY KEY (book_id),
  CONSTRAINT book_author_fk FOREIGN KEY (author_id)
    REFERENCES author (author_id),
  CONSTRAINT book_genre_ck
    CHECK (genre IN ('novel', 'poetry', 'drama', 'essay')),
  CONSTRAINT book_pages_ck CHECK (pages > 0),
  CONSTRAINT book_price_ck CHECK (price >= 0)
);
```

book_author_fk: a book refers to an **existing** author; author is therefore created **first**.

6 Creating the reader table

```
CREATE TABLE reader (
  reader_id    INTEGER GENERATED ALWAYS AS IDENTITY,
  last_name    VARCHAR(40) NOT NULL,
  first_name   VARCHAR(40) NOT NULL,
  city         VARCHAR(30),
  email        VARCHAR(80),
  registered   DATE NOT NULL DEFAULT CURRENT_DATE,
  active       BOOLEAN NOT NULL DEFAULT TRUE,
  CONSTRAINT reader_pk PRIMARY KEY (reader_id),
  CONSTRAINT reader_email_uq UNIQUE (email)
);
```

- `registered` defaults to the current date.
- Two readers cannot have the same email; a reader may have none.

6 Creating the loan table

```
CREATE TABLE loan (
  loan_id      INTEGER GENERATED ALWAYS AS IDENTITY,
  book_id      INTEGER NOT NULL,
  reader_id    INTEGER NOT NULL,
  loan_date    DATE NOT NULL DEFAULT CURRENT_DATE,
  return_date  DATE,
  CONSTRAINT loan_pk PRIMARY KEY (loan_id),
  CONSTRAINT loan_book_fk FOREIGN KEY (book_id)
    REFERENCES book (book_id),
  CONSTRAINT loan_reader_fk FOREIGN KEY (reader_id)
    REFERENCES reader (reader_id) ON DELETE CASCADE,
  CONSTRAINT loan_dates_ck CHECK (return_date >= loan_date)
);
```

- `ON DELETE CASCADE`: deleting a reader deletes their loans.
- Empty `return_date` (`NULL`) = ongoing loan.

6 Checking a table: \d

```
library=> \d loan
                    Table "lib.loan"
   Column    |  Type   | ... | Nullable |     Default
-------------+---------+-----+----------+-----------------
 loan_id     | integer |     | not null | generated always
 book_id     | integer |     | not null |   as identity
 reader_id   | integer |     | not null |
 loan_date   | date    |     | not null | CURRENT_DATE
 return_date | date    |     |          |
Indexes:
    "loan_pk" PRIMARY KEY, btree (loan_id)
Check constraints:
    "loan_dates_ck" CHECK (return_date >= loan_date)
Foreign-key constraints:
    ...
```

\dt lists the 4 tables; \d name details columns and constraints.

7 Inserting the authors

```
INSERT INTO author (last_name, first_name, birth_year) VALUES
  ('Dib', 'Mohammed', 1920),
  ('Kateb', 'Yacine', 1929),
  ('Djebar', 'Assia', 1936),
  ('Feraoun', 'Mouloud', 1913);                    -- INSERT 0 4

INSERT INTO author (last_name, first_name, country, birth_year) VALUES
  ('Camus', 'Albert', 'France', 1913),
  ('Hugo', 'Victor', 'France', 1802),
  ('Darwich', 'Mahmoud', 'Palestine', 1941);       -- INSERT 0 3

INSERT INTO author (last_name, first_name, birth_year)
VALUES ('Mammeri', 'Mouloud', 1917);               -- INSERT 0 1
```

- List the columns, then the values **in the same order**.
- Missing columns take their default value.

7 Checking: SELECT * FROM author

```
library=> SELECT * FROM author;
 author_id | last_name | first_name |  country   | birth_year
-----------+-----------+------------+------------+------------
         1 | Dib       | Mohammed   | Algeria    |       1920
         2 | Kateb     | Yacine     | Algeria    |       1929
         3 | Djebar    | Assia      | Algeria    |       1936
         4 | Feraoun   | Mouloud    | Algeria    |       1913
         5 | Camus     | Albert     | France     |       1913
         6 | Hugo      | Victor     | France     |       1802
         7 | Darwich   | Mahmoud    | Palestine  |       1941
         8 | Mammeri   | Mouloud    | Algeria    |       1917
(8 rows)
```

The identifiers follow the insertion order: they are reused in the foreign keys (Dib = 1, Kateb = 2...).

7 Inserting the books

```
INSERT INTO book (title, author_id, genre, pub_year, pages, price)
VALUES
  ('La Grande Maison', 1, 'novel', 1952, 190, 850.00),
  ('L''Incendie', 1, 'novel', 1954, 210, 800.00),
  ('Qui se souvient de la mer', 1, 'novel', 1962, 190, 900.00),
  ('Nedjma', 2, 'novel', 1956, 256, 950.00),
  ('Le Cadavre encerclé', 2, 'drama', 1959, 120, 600.00),
  ('Les Enfants du nouveau monde', 3, 'novel', 1962, 312, 1100.00),
  ('L''Amour, la fantasia', 3, 'novel', 1985, 302, 1250.00),
  ('Le Fils du pauvre', 4, 'novel', 1950, 210, 700.00),
  ('La Peste', 5, 'novel', 1947, 336, 1200.00),
  ('Les Justes', 5, 'drama', 1949, 150, 650.00),
  ('Les Misérables', 6, 'novel', 1862, 1488, 2500.00),
  ('Les Contemplations', 6, 'poetry', 1856, 420, 900.00),
  ('Une mémoire pour l''oubli', 7, 'essay', 1987, 180, NULL);
```

Answer: `INSERT 0 13`. The last price is unknown: `NULL`.

7 Inserting the readers

```
INSERT INTO reader (last_name, first_name, city, email, registered)
VALUES
  ('Bouzid', 'Amina', 'Skikda', 'amina@mail.dz', '2026-09-20'),
  ('Benali', 'Karim', 'Skikda', 'karim@mail.dz', '2026-09-21'),
  ('Saadi', 'Yasmine', 'Annaba', NULL, '2026-09-25'),
  ('Haddad', 'Rami', 'Constantine', 'rami@mail.dz', '2026-10-01'),
  ('Mansouri', 'Lina', 'Skikda', 'lina@mail.dz', '2026-10-02');

INSERT INTO reader (last_name, first_name, city, registered, active)
VALUES ('Cherif', 'Omar', 'Annaba', '2026-10-05', FALSE);
```

- Answers: `INSERT 0 5`, then `INSERT 0 1`.
- Readers numbered 1 to 6; Omar (6) is inactive.

7 Inserting the loans

```
INSERT INTO loan (book_id, reader_id, loan_date, return_date)
VALUES
  ( 4, 1, '2026-09-22', '2026-10-01'),
  ( 9, 1, '2026-10-02', NULL),
  ( 1, 2, '2026-09-23', '2026-09-30'),
  ( 4, 2, '2026-10-03', NULL),
  (11, 3, '2026-09-26', '2026-10-10'),
  ( 6, 4, '2026-10-02', '2026-10-09'),
  (12, 4, '2026-10-05', NULL),
  ( 8, 5, '2026-10-03', '2026-10-08'),
  ( 1, 5, '2026-10-09', NULL),
  ( 5, 1, '2026-10-10', NULL);                    -- INSERT 0 10
```

- Books and readers must exist **before** their loans.
- 5 loans are ongoing (`return_date NULL`).

7 Constraints in action (1/3)

These commands must **fail**: nothing is stored.

```
=> INSERT INTO book (title, author_id, genre)
   VALUES ('X', 99, 'novel');
ERROR:  insert or update on table "book" violates foreign key
constraint "book_author_fk"
DETAIL:  Key (author_id)=(99) is not present in table
"author".
```

```
=> INSERT INTO book (title, author_id, genre)
   VALUES ('X', 1, 'comic');
ERROR:  new row for relation "book" violates check
constraint "book_genre_ck"
```

Read the message **to the end**: it names the violated constraint.

7 Constraints in action (2/3)

```
=> INSERT INTO book (author_id, genre) VALUES (1, 'novel');
ERROR:  null value in column "title" of relation "book"
violates not-null constraint
```

```
=> INSERT INTO reader (last_name, first_name, email)
   VALUES ('Aouadi', 'Sami', 'karim@mail.dz');
ERROR:  duplicate key value violates unique constraint
"reader_email_uq"
DETAIL:  Key (email)=(karim@mail.dz) already exists.
```

7 Constraints in action (3/3)

```
=> INSERT INTO loan (book_id, reader_id, loan_date,
   return_date) VALUES (1, 1, '2026-10-10', '2026-10-01');
ERROR:  new row for relation "loan" violates check
constraint "loan_dates_ck"
```

```
=> INSERT INTO author (author_id, last_name) VALUES (50, 'Haddad');
ERROR:  cannot insert a non-DEFAULT value into column
"author_id"
```

The exact wording of the messages may vary with the version and the language of Windows; the name of the constraint, however, is always the one you chose.

8 Why write your queries in a script?

What is it?

A **script** is a plain text file, with the `.sql` extension, that contains SQL commands. `psql` runs them all, in order.

Typing each query in `psql` is fine for a quick test. With a script:

- you **fix** a mistake without retyping everything;
- you **replay** all the queries with one command;
- you **keep** your work from one session to the next.

From now on

Write the queries of steps 9 to 13 in a script, then run it.

8 Creating the script file

In a **second** Command Prompt (leave `psql` open in the first one):

```
C:\Users\ahmed> mkdir C:\Lab
C:\Users\ahmed> notepad C:\Lab\queries.sql
```

- `mkdir` creates the folder `C:\Lab` (only once).
- `notepad` opens Notepad; answer **Yes** to "Do you want to create a new file?".

Notepad++ or VS Code are also fine, and they highlight SQL syntax.

8 Writing and saving the queries

In Notepad, type:

```
-- queries.sql: my queries on the library database
-- (a line that starts with -- is a comment)

SELECT title, pub_year FROM book WHERE genre = 'drama';

SELECT COUNT(*) AS num_readers FROM reader;
```

Then **File** → **Save As**:

- **Encoding**: choose **ANSI** (same encoding as the console set by `chcp 1252`);
- **Save**.

Each query ends with a semicolon, as in psql.

8 Running the script in psql

In the psql window, the \i (*include*) command runs a file:

```
library=> \i C:/Lab/queries.sql
        title         | pub_year
----------------------+----------
 Le Cadavre encerclé |     1959
 Les Justes          |     1949
(2 rows)

 num_readers
-------------
           6
(1 row)
```

Mind the path

In psql, write the path with forward slashes /, not \: C:/Lab/queries.sql.

8 Fix, extend, run again

You then work in a loop:

1. add or fix a query in Notepad;
2. save (**Ctrl+S**);
3. run it again in `psql`: **up** arrow to recall `\i ...`, then Enter.

To avoid typing the full path every time:

```
library=> \cd C:/Lab
library=> \i queries.sql
```

`\cd` changes the current folder of `psql`.

8 Running a script without opening psql

From the Command Prompt, the `-f` (*file*) option runs the script then quits:

```
C:\Lab> psql -U student -d library -f queries.sql
```

Stopping at the first error

```
C:\Lab> psql -U student -d library -v ON_ERROR_STOP=1 -f queries.sql
```

Without this option, `psql` carries on after an error. Always fix the **first** error: the following ones often result from it.

8 A script that rebuilds the whole database

Same principle for the creation: a file `C:\Lab\create.sql` gathers the commands of steps 4 to 7:

```
DROP SCHEMA IF EXISTS lib CASCADE;      -- start from scratch
CREATE SCHEMA lib;
SET search_path TO lib, public;

CREATE TABLE author ( ... );            -- step 6
...
INSERT INTO author ... ;                -- step 7
...
```

`\i C:/Lab/create.sql` then rebuilds the entire database, as many times as you like.

9 Reading a table and sorting: ORDER BY

```
SELECT title, pub_year, price  -- the desired columns (* = all)
FROM   book
ORDER  BY price DESC, title;   -- price descending, then title
```

```
            title              | pub_year |  price
-------------------------------+----------+---------
 Une mémoire pour l'oubli      |     1987 |
 Les Misérables                |     1862 | 2500.00
 L'Amour, la fantasia          |     1985 | 1250.00
 La Peste                      |     1947 | 1200.00
 ...
(13 rows)
```

- ASC ascending (default), DESC descending.
- The NULL price comes first; to avoid this: DESC NULLS LAST.

9 Filtering rows: WHERE

```
SELECT title, price FROM book
WHERE  genre = 'novel' AND price < 900
ORDER  BY price;
```

```
       title       | price
-------------------+--------
 Le Fils du pauvre | 700.00
 L'Incendie        | 800.00
 La Grande Maison  | 850.00
(3 rows)
```

- Operators: =, <>, <, <=, >, >=, AND, OR, NOT.
- 'novel': a value. "novel": a column name!

9 A range: BETWEEN

```
SELECT title, pub_year FROM book
WHERE  pub_year BETWEEN 1950 AND 1959  -- bounds included
ORDER  BY pub_year;
```

```
        title         | pub_year
----------------------+----------
 Le Fils du pauvre    |     1950
 La Grande Maison     |     1952
 L'Incendie           |     1954
 Nedjma               |     1956
 Le Cadavre encerclé  |     1959
(5 rows)
```

Equivalent to `pub_year >= 1950 AND pub_year <= 1959`.

9 A list: IN; a pattern: LIKE

```
SELECT title, genre
FROM   book
WHERE  genre IN ('drama',
                 'poetry');
```

```
        title         | genre
----------------------+--------
 Le Cadavre encerclé | drama
 Les Justes          | drama
 Les Contemplations  | poetry
```

```
SELECT title
FROM   book
WHERE  title LIKE 'Les %';
```

```
             title
-----------------------------
 Les Enfants du nouveau monde
 Les Justes
 Les Misérables
 Les Contemplations
```

- LIKE: % = any sequence of characters, _ = one character.
- ILIKE: the same, without distinguishing upper and lower case.

9 Unknown values: IS NULL

```
SELECT last_name, first_name FROM reader
WHERE  email IS NULL;
```

```
 last_name | first_name
-----------+------------
 Saadi     | Yasmine
 Cherif    | Omar
(2 rows)
```

Never = NULL

`WHERE email = NULL` returns **no** rows. Write `IS NULL` or `IS NOT NULL`.

`COALESCE(price, 0)` replaces a `NULL` with 0.

9 Removing duplicates: DISTINCT

```
SELECT city FROM reader
ORDER  BY city;
```

```
    city
------------
 Annaba
 Annaba
 Constantine
 Skikda
 Skikda
 Skikda
(6 rows)
```

```
SELECT DISTINCT city FROM reader
ORDER  BY city;
```

```
    city
------------
 Annaba
 Constantine
 Skikda
(3 rows)
```

DISTINCT keeps each **different** row only once.

9 Computing in the SELECT

```
SELECT title, price,
       ROUND(price * 1.19, 2) AS price_incl_tax  -- AS: names the column
FROM   book
WHERE  genre = 'drama';
```

```
        title         | price  | price_incl_tax
----------------------+--------+----------------
 Le Cadavre encerclé | 600.00 |         714.00
 Les Justes          | 650.00 |         773.50
(2 rows)
```

`+ - * /` on numbers; `ROUND(x, 2)` rounds to 2 decimal places.

9 Functions on text and dates

```
SELECT UPPER(last_name) || ' ' || first_name AS reader,
       EXTRACT(MONTH FROM registered) AS month
FROM   reader
ORDER  BY registered
LIMIT  3;                          -- only the first 3 rows
```

```
    reader      | month
----------------+-------
 BOUZID Amina   |     9
 BENALI Karim   |     9
 SAADI Yasmine  |     9
(3 rows)
```

- || joins two texts end to end.
- Others: `LOWER`, `LENGTH`, `CURRENT_DATE - registered` (days).

9 Choosing a value: CASE

```
SELECT title, pages,
       CASE WHEN pages < 200 THEN 'short'
            WHEN pages < 400 THEN 'medium'
            ELSE 'long'
       END AS length
FROM   book
WHERE  genre <> 'novel';
```

```
          title            | pages | length
---------------------------+-------+--------
 Le Cadavre encerclé       |   120 | short
 Les Justes                |   150 | short
 Les Contemplations        |   420 | long
 Une mémoire pour l'oubli  |   180 | short
(4 rows)
```

CASE returns the value of the **first** true condition.

10 Aggregate functions

They summarize **several rows into a single value**:

```
SELECT COUNT(*) AS num_books, COUNT(price) AS num_prices,
       MIN(price) AS lowest, MAX(price) AS highest,
       ROUND(AVG(price), 2) AS average
FROM   book;
```

```
 num_books | num_prices | lowest | highest | average
-----------+------------+--------+---------+---------
        13 |         12 | 600.00 | 2500.00 | 1033.33
(1 row)
```

- `COUNT(*)` counts the rows; `COUNT(price)` the known prices.
- `AVG`, `SUM`, `MIN`, `MAX` **ignore** `NULL` values.

10 Grouping: GROUP BY

```
SELECT genre, COUNT(*) AS num
FROM   book
GROUP  BY genre              -- one result row per genre
ORDER  BY num DESC;
```

```
 genre   | num
---------+-----
 novel   |   9
 drama   |   2
 essay   |   1
 poetry  |   1
(4 rows)
```

Rule

A column of the SELECT that is not aggregated must appear in the GROUP BY.

10 Filtering groups: HAVING

```sql
SELECT author_id, COUNT(*) AS num
FROM   book
WHERE  genre = 'novel'       -- 1) filters the ROWS
GROUP  BY author_id          -- 2) forms the groups
HAVING COUNT(*) >= 2;        -- 3) filters the GROUPS
```

Result: author 1 (3 novels) and author 3 (2 novels).

Evaluation order of a query

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT

11 Joining two tables: JOIN

```
SELECT b.title, a.last_name
FROM   book b                                    -- b, a: aliases
JOIN   author a ON a.author_id = b.author_id     -- the condition
WHERE  a.country = 'France';
```

```
       title         | last_name
---------------------+-----------
 La Peste            | Camus
 Les Justes          | Camus
 Les Misérables      | Hugo
 Les Contemplations  | Hugo
(4 rows)
```

- The `ON` condition links the foreign key to the primary key.
- Without `ON`: every book with **every** author!

11 Joining three tables

The ongoing loans, with the reader and the title:

```
SELECT r.first_name, b.title, l.loan_date
FROM   loan l
JOIN   reader r ON r.reader_id = l.reader_id
JOIN   book   b ON b.book_id   = l.book_id
WHERE  l.return_date IS NULL
ORDER  BY l.loan_date;
```

```
 first_name |        title         | loan_date
------------+----------------------+------------
 Amina      | La Peste             | 2026-10-02
 Karim      | Nedjma               | 2026-10-03
 Rami       | Les Contemplations   | 2026-10-05
 Lina       | La Grande Maison     | 2026-10-09
 Amina      | Le Cadavre encerclé  | 2026-10-10
(5 rows)
```

11 Outer join: LEFT JOIN

```
SELECT a.last_name, COUNT(b.book_id) AS num_books
FROM   author a
LEFT   JOIN book b ON b.author_id = a.author_id
GROUP  BY a.last_name
ORDER  BY num_books DESC;
```

```
 last_name | num_books
-----------+-----------
 Dib       |         3
 ...
 Mammeri   |         0
(8 rows)
```

- LEFT JOIN keeps **all** the authors, even those without a book.
- With JOIN, Mammeri would disappear.

11 Finding what has no match

The books **never borrowed**:

```
SELECT b.title
FROM   book b
LEFT   JOIN loan l ON l.book_id = b.book_id
WHERE  l.loan_id IS NULL;
```

```
           title
-------------------------
 L'Incendie
 Qui se souvient de la mer
 L'Amour, la fantasia
 Les Justes
 Une mémoire pour l'oubli
(5 rows)
```

For these books, the "loan" part of the join is empty (`NULL`).

12 A subquery that returns one value

A **subquery** is a SELECT in parentheses, inside another query.

```
SELECT title, price FROM book
WHERE  price > (SELECT AVG(price) FROM book)   -- 1033.33
ORDER  BY price DESC;
```

```
            title              |  price
-------------------------------+---------
 Les Misérables                | 2500.00
 L'Amour, la fantasia          | 1250.00
 La Peste                      | 1200.00
 Les Enfants du nouveau monde  | 1100.00
(4 rows)
```

Test the subquery on its own first, then insert it.

12 A subquery that returns a list: IN

The readers who borrowed a book by Kateb (author 2):

```
SELECT first_name, last_name FROM reader
WHERE  reader_id IN (SELECT l.reader_id
                     FROM   loan l
                     JOIN   book b ON b.book_id = l.book_id
                     WHERE  b.author_id = 2);
```

```
 first_name | last_name
------------+-----------
 Amina      | Bouzid
 Karim      | Benali
(2 rows)
```

12 Testing existence: NOT EXISTS

The readers who have **never** borrowed anything:

```
SELECT first_name, last_name FROM reader r
WHERE  NOT EXISTS (SELECT 1
                   FROM   loan l
                   WHERE  l.reader_id = r.reader_id);
```

```
 first_name | last_name
------------+-----------
 Omar       | Cherif
(1 row)
```

- EXISTS is true if the subquery returns at least one row.
- The subquery uses `r.reader_id`: it is re-evaluated for each reader.

12 Combining two results: UNION

The cities of the active readers, plus Algiers:

```
SELECT city FROM reader WHERE active
UNION
SELECT 'Algiers'
ORDER  BY city;
```

```
    city
-------------
 Algiers
 Annaba
 Constantine
 Skikda
(4 rows)
```

- Both SELECTs must have the same columns.
- UNION removes duplicates; UNION ALL keeps them.

12 INTERSECT and EXCEPT

```sql
-- INTERSECT: one AND the other
-- readers with an ongoing loan AND living in Skikda
SELECT reader_id FROM loan WHERE return_date IS NULL
INTERSECT
SELECT reader_id FROM reader WHERE city = 'Skikda';
                                        -- 1, 2, 5

-- EXCEPT: one EXCEPT the other
-- novels never borrowed
SELECT book_id FROM book WHERE genre = 'novel'
EXCEPT
SELECT book_id FROM loan;                -- 2, 3, 7
```

Union, intersection, difference: the set operations of relational algebra.

13 Creating a view

A **view** is a query saved under a name:

```
CREATE VIEW current_loans AS
SELECT l.loan_id,
       r.first_name || ' ' || r.last_name AS reader,
       b.title, l.loan_date
FROM   loan l
JOIN   reader r ON r.reader_id = l.reader_id
JOIN   book   b ON b.book_id   = l.book_id
WHERE  l.return_date IS NULL;
```

Answer: `CREATE VIEW`. `\dv` lists the views.

13 Using a view

It is queried like a table:

```
SELECT reader, title FROM current_loans;
```

```
    reader     |        title
---------------+---------------------
 Amina Bouzid  | La Peste
 Karim Benali  | Nedjma
 Rami Haddad   | Les Contemplations
 Lina Mansouri | La Grande Maison
 Amina Bouzid  | Le Cadavre encerclé
(5 rows)
```

The view stores no data: its result is **always up to date**.

14 Modifying rows: UPDATE

```sql
-- Amina returns "La Peste" (loan 2)
UPDATE loan SET return_date = '2026-10-14'
WHERE  loan_id = 2;                              -- UPDATE 1

-- Poetry books go up by 5 %
UPDATE book SET price = price * 1.05
WHERE  genre = 'poetry';                         -- UPDATE 1
```

`SELECT * FROM current_loans;` now shows only 4 loans.

Always a `WHERE` clause

Without `WHERE`, **all** rows are modified. Check first with a `SELECT` using the same `WHERE`.

14 Deleting rows: DELETE

```
DELETE FROM reader WHERE active = FALSE;        -- DELETE 1
```

A foreign key **protects** the referenced rows:

```
=> DELETE FROM author WHERE last_name = 'Dib';
ERROR:  update or delete on table "author" violates foreign
key constraint "book_author_fk" on table "book"
```

- Dib has books: deleting him is **refused**.
- With `ON DELETE CASCADE` (`loan_reader_fk`), deleting a reader also deletes their loans.

14 Modifying a table: ALTER TABLE

A table evolves **without losing its data**:

```
ALTER TABLE book ADD COLUMN isbn VARCHAR(17);
ALTER TABLE book ADD CONSTRAINT book_isbn_uq UNIQUE (isbn);
ALTER TABLE book ALTER COLUMN title TYPE VARCHAR(150);
ALTER TABLE book RENAME COLUMN pub_year TO published;
ALTER TABLE book RENAME COLUMN published TO pub_year;
ALTER TABLE book DROP CONSTRAINT book_isbn_uq;
ALTER TABLE book DROP COLUMN isbn;
```

- Answer: `ALTER TABLE`; check with `\d book`.
- `DROP TABLE t;` drops the whole table, data included.

15 Transactions

```
library=> BEGIN;
library=*> DELETE
library-*> FROM loan;
DELETE 10
library=*> SELECT COUNT(*)
library-*> FROM loan;
 count
-------
     0
library=*> ROLLBACK;
library=> SELECT COUNT(*)
library-> FROM loan;
 count
-------
    10
```

- `BEGIN` starts a transaction.
- `ROLLBACK` **cancels** everything since `BEGIN`: the loans are back.
- `COMMIT` **validates** permanently.
- Without `BEGIN`, each command is committed immediately.

16 Creating a group and a user

Goal: **readers** who browse the catalog without modifying it. Only `postgres` can create roles:

```
-- as postgres:  \c library postgres
CREATE ROLE readers NOLOGIN;                    -- a group
CREATE ROLE guest LOGIN PASSWORD 'guest';       -- a user
GRANT readers TO guest;                         -- guest is a member
```

Login roles and group roles: see the “Concepts” section.

16 Granting privileges: GRANT

Privileges are granted by the **owner** of the objects:

```
-- as student:  \c library student
GRANT CONNECT ON DATABASE library TO readers;
GRANT USAGE ON SCHEMA lib TO readers;
GRANT SELECT ON ALL TABLES IN SCHEMA lib TO readers;
```

Level	Privileges
Database	`CONNECT` (revoked from `PUBLIC` in step 3)
Schema	`USAGE`, `CREATE`
Table, view	`SELECT`, `INSERT`, `UPDATE`, `DELETE`

All levels are needed: without `USAGE` on the schema, no access to the tables.

16 Testing the privileges

```
library=> \c library guest
library=> SELECT title FROM book WHERE pub_year = 1956;
 title
--------
 Nedjma
(1 row)

library=> DELETE FROM book;
ERROR:  permission denied for table book
```

guest can read, but not delete: this is the intended result.

16 Viewing and revoking privileges

Back as `student`:

```
\dp lib.*                                          -- who has which privileges?
REVOKE SELECT ON lib.reader FROM readers;
ALTER DEFAULT PRIVILEGES IN SCHEMA lib
  GRANT SELECT ON TABLES TO readers;               -- future tables
```

Pitfall

`GRANT ... ON ALL TABLES` only applies to **existing** tables. For tables created later: `ALTER DEFAULT PRIVILEGES`.

17 Backing up a database

From the **Command Prompt** (not in psql):

```
C:\Lab> pg_dump -U student -d library -F c -f library.dump
```

```
C:\Lab> pg_dump -U student -d library -f library.sql
```

- `-F c`: compressed format, read back by `pg_restore`.
- Without `-F`: a readable SQL script (open it!).

Good habit

Keep your script and your backup on a USB stick or in the cloud.

17 Restoring a database

Into a **new** database (the original one remains intact):

```
C:\Lab> createdb -U postgres -O student library_copy
C:\Lab> pg_restore -U student -d library_copy library.dump
C:\Lab> psql -U student -d library_copy
library_copy=> SELECT COUNT(*) FROM lib.book;
 count
-------
    13
(1 row)
```

The copy does contain the 13 books. To drop it afterwards (as `postgres`):

```
DROP DATABASE library_copy;
```

Common errors

Message or symptom	Solution
`password authentication failed`	check the password and the role (`-U`)
`role "ahmed" does not exist`	always specify `-U`
`relation "book" does not exist`	check the database (`\conninfo`) and the `search_path`
`syntax error at or near`	look at the indicated word and the one before it
`current transaction is aborted...`	type `ROLLBACK;`
unreadable accented characters	`chcp 1252`; script saved in ANSI

Cheat sheet (1/2): creating

```
CREATE ROLE r LOGIN PASSWORD '...';
CREATE DATABASE b OWNER r;
CREATE SCHEMA s;
SET search_path TO s, public;

CREATE TABLE t (
  id   INTEGER GENERATED ALWAYS AS IDENTITY,
  col  TYPE [NOT NULL] [DEFAULT value],
  CONSTRAINT t_pk PRIMARY KEY (id),
  CONSTRAINT t_col_uq UNIQUE (col),
  CONSTRAINT t_col_ck CHECK (condition),
  CONSTRAINT t_col_fk FOREIGN KEY (col) REFERENCES t2 (col2)
);

ALTER TABLE t ADD COLUMN ... ;
CREATE VIEW v AS SELECT ... ;
```

Cheat sheet (2/2): manipulating and querying

```
INSERT INTO t (col1, col2) VALUES (v1, v2), (v1, v2);
UPDATE t SET col = value WHERE condition;
DELETE FROM t WHERE condition;

SELECT [DISTINCT] columns
FROM   t1 [LEFT] JOIN t2 ON condition
WHERE  condition
GROUP  BY columns
HAVING condition
ORDER  BY column [DESC]
LIMIT  n;

BEGIN; ... COMMIT;     -- or ROLLBACK;
GRANT SELECT ON t TO r;
```

Self-assessment exercises (1/2)

On the `library` database:

1. Add a reader without specifying their registration date; check the value obtained.
2. List the books of more than 300 pages, from the longest to the shortest.
3. Give, for each country, the number of authors.
4. Give the number of loans of each reader, **including** 0.
5. List the titles borrowed at least twice.

Self-assessment exercises (2/2)

6. Display the duration in days of each completed loan.
7. List the authors who have written no novel.
8. Add to `book` a column `language` (`'French'` by default).
9. In a transaction, delete the completed loans, then cancel.
10. Create a view `catalog` (title, author, price), readable by `readers`.

Success criterion

Your whole script runs **without errors** with `ON_ERROR_STOP`.

What next: Part 2

In **Part 2**, you will apply the same approach **on your own** to the `pizzerias` database:

1. role, database and schema (steps 2 to 4);
2. tables and constraints (steps 5 and 6);
3. data from the assignment (step 7);
4. queries (steps 9 to 12);
5. a script that runs without errors (step 8).

Resources

`www.al-moualime.com/adb`
https://www.postgresql.org/docs/16/