SQL – Data Definition and Manipulation Languages (DDL & DML)

PostgreSQL – Practical Work Guidelines

1. SQL: The Need

Modern information systems rely on databases that may contain dozens (or hundreds) of tables and very large volumes of data. They also run in heterogeneous environments (different hardware, operating systems, and application stacks). To ensure portability and maintainability, database systems separate data storage from application processing. This motivates the need for a simple, standard query language that is independent of the underlying environment and can be integrated into complex applications. SQL (Structured Query Language) fulfills this role.

2. SQL: The Relational Language

3. SQL Components

In practice, SQL is commonly presented as four complementary sub-languages:

- DDL (Data Definition Language): define schemas and database objects (tables, constraints, indexes, views, …).
- DML (Data Manipulation Language): insert, update, delete, and query data.
- DCL (Data Control Language): manage privileges and access control (GRANT, REVOKE).
- TCL (Transaction Control Language): manage transactions (BEGIN, COMMIT, ROLLBACK) — central in PostgreSQL practice.

4. Reference Schema for All Examples

To keep the course consistent, we use a simplified relational schema inspired by the classic Oracle HR examples. All DDL and DML illustrations refer to this schema unless stated otherwise.

Tables:

- departments(dept_id, dept_name)
- jobs(job_id, job_title, min_salary, max_salary)
- employees(emp_id, first_name, last_name, birth_date, hire_date, salary, dept_id#, job_id#)

Schema creation (PostgreSQL):

```
CREATE SCHEMA IF NOT EXISTS hr;
SET search_path TO hr;
```

Core table definitions (PostgreSQL):

```
CREATE TABLE departments (
  dept_id   INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  dept_name VARCHAR(50) NOT NULL UNIQUE
```

```
);

CREATE TABLE jobs (
  job_id     VARCHAR(10) PRIMARY KEY,
  job_title  VARCHAR(50) NOT NULL,
  min_salary NUMERIC(10,2) CHECK (min_salary >= 0),
  max_salary NUMERIC(10,2) CHECK (max_salary >= min_salary)
);

CREATE TABLE employees (
  emp_id     INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  first_name VARCHAR(30) NOT NULL,
  last_name  VARCHAR(30) NOT NULL,
  birth_date DATE,
  hire_date  DATE DEFAULT CURRENT_DATE NOT NULL,
  salary     NUMERIC(10,2) CHECK (salary >= 0),
  dept_id    INTEGER REFERENCES departments(dept_id) ON DELETE SET NULL,
  job_id     VARCHAR(10) REFERENCES jobs(job_id) ON DELETE RESTRICT
);
```

PART I — DDL: Data Definition Language

1. Object Naming Rules (Tables and Columns)

In teaching materials, naming rules are often given as strict conventions. PostgreSQL is more flexible, but adopting conventions helps avoid errors and improves readability.

- Use letters, digits, and underscores; avoid spaces and special characters.
- Prefer lowercase names (PostgreSQL folds unquoted identifiers to lowercase).
- Keep names meaningful and consistent (e.g., emp_id, dept_id).
- Avoid reserved keywords (e.g., user, select).
- Use a schema prefix when necessary: hr.employees, hr.departments.
- A fully qualified column reference can be written as table_name.column_name.

2. Common Data Types in PostgreSQL

2.1 Numeric Types

- SMALLINT: 2-byte integer (range approximately −32768 to +32767).
- INTEGER (INT): 4-byte integer.
- NUMERIC(p, d) / DECIMAL(p, d): exact fixed-point number with precision p and scale d.
- REAL: floating-point, single precision.
- DOUBLE PRECISION: floating-point, double precision.
- BOOLEAN: TRUE / FALSE (instead of BIT for boolean logic in PostgreSQL).

2.2 Character Types

- CHAR(n): fixed-length string, padded with spaces.
- VARCHAR(n): variable-length string up to n characters.
- TEXT: variable-length string with no specified limit (practical in PostgreSQL).
- Strings are delimited by single quotes ('). To include a quote inside a string, double it: 'today''s'.

2.3 Date/Time Types

- DATE: calendar date (year-month-day).
- TIME [WITHOUT TIME ZONE]: time of day.
- TIMESTAMP [WITHOUT TIME ZONE]: date and time.
- INTERVAL: duration/period.

2.4 NULL and Unknown Values

A column without a value is NULL. NULL is not zero and is not an empty string. It represents the absence of information. In comparisons, NULL behaves as “unknown.

3. CREATE TABLE

Concept: CREATE TABLE defines a new relation (table), its attributes (columns), and its integrity constraints.

General syntax:

```
CREATE TABLE [schema.]table_name (
  column_name data_type [DEFAULT expression] [NOT NULL],
  ...,
  [CONSTRAINT constraint_name] PRIMARY KEY (...),
  [CONSTRAINT constraint_name] UNIQUE (...),
  [CONSTRAINT constraint_name] FOREIGN KEY (...) REFERENCES ref_table(...)
      [ON DELETE {CASCADE | SET NULL | RESTRICT | NO ACTION}]
      [ON UPDATE {CASCADE | SET NULL | RESTRICT | NO ACTION}],
  [CONSTRAINT constraint_name] CHECK (condition)
);
```

Rules and notes:

- A table must have a unique name within its schema.
- Each column name must be unique within the table.
- DEFAULT defines a value used when INSERT omits the column.
- NOT NULL enforces mandatory data for the column.
- Constraints can be declared at column level or table level.
- PostgreSQL supports identity columns (GENERATED ... AS IDENTITY) as the recommended modern replacement for SERIAL.

Illustrative example (article table):

```
CREATE TABLE article (
  reference VARCHAR(10) CONSTRAINT pk_article PRIMARY KEY,
  name      VARCHAR(30) NOT NULL,
  price     NUMERIC(9,2),
  datemaj   DATE DEFAULT CURRENT_DATE
);
```

4. Integrity Constraints (Summary + Pedagogical Notes)

Integrity constraints are rules enforced by the DBMS to guarantee data consistency. They protect the database against invalid states (duplicate identifiers, broken references, negative salaries, etc.). In PostgreSQL, constraints are validated at INSERT/UPDATE time (and sometimes at transaction COMMIT depending on deferrability).

4.1 PRIMARY KEY

Purpose: uniquely identify each row in a table.

Syntax (column level):

```
col_name data_type PRIMARY KEY
```

Syntax (table level):

```
CONSTRAINT pk_name PRIMARY KEY (col1, col2, ...)
```

Rules and notes:

- A PRIMARY KEY is unique and NOT NULL by definition.
- A table can have only one PRIMARY KEY (possibly composite).
- PostgreSQL automatically creates a unique index to enforce it.

Example:

```
CREATE TABLE departments (
  dept_id   INTEGER GENERATED ALWAYS AS IDENTITY,
  dept_name VARCHAR(50) NOT NULL,
  CONSTRAINT pk_departments PRIMARY KEY (dept_id)
);
```

4.2 UNIQUE

Purpose: prevent duplicate values in a column or set of columns that must remain distinct.

General syntax:

```
CONSTRAINT uq_name UNIQUE (col1, col2, ...)
```

Rules and notes:

- A table can have multiple UNIQUE constraints.

- UNIQUE allows NULLs (multiple NULLs are allowed in PostgreSQL) because NULL means “unknown”.
- A UNIQUE constraint also creates a unique index.

Example:

```
ALTER TABLE hr.departments ADD CONSTRAINT uq_dept_name UNIQUE (dept_name);
```

4.3 FOREIGN KEY (Referential Integrity)

Purpose: ensure that a referencing value exists in the referenced table (or is NULL), maintaining consistency across tables.

General syntax:

```
CONSTRAINT fk_name
  FOREIGN KEY (fk_col1, fk_col2, ...)
  REFERENCES ref_table (ref_col1, ref_col2, ...)
  [ON DELETE action]
  [ON UPDATE action];
```

Common actions:

- ON DELETE CASCADE: deleting a referenced row deletes dependent rows.
- ON DELETE SET NULL: dependent foreign keys become NULL.
- ON DELETE RESTRICT / NO ACTION: deletion is rejected if dependents exist (default behavior).

Rules and notes:

- The referenced columns must be a PRIMARY KEY or UNIQUE.
- FK columns and referenced columns must be type-compatible.
- FOREIGN KEY constraints may require indexes on FK columns for performance (recommended when tables grow).

Example (employees → departments):

```
ALTER TABLE hr.employees
ADD CONSTRAINT fk_emp_dept
FOREIGN KEY (dept_id) REFERENCES hr.departments(dept_id)
ON DELETE SET NULL;
```

4.4 CHECK

Purpose: enforce a boolean condition that must hold for every row.

General syntax:

```
CONSTRAINT ck_name CHECK (condition)
```

Rules and notes:

- Use CHECK for domain constraints (e.g., salary >= 0).

- The condition can reference columns of the same row.
- Be careful with NULL: if the condition evaluates to NULL, the row is accepted (because it is not FALSE).

Example:

```
ALTER TABLE hr.jobs ADD CONSTRAINT ck_salary_range CHECK (max_salary >= min_salary);
```

4.5 NOT NULL (Mandatory Attribute)

Purpose: forbid NULL values for a column when the information is mandatory.

Syntax:

```
col_name data_type NOT NULL
```

Rules and notes:

- NOT NULL is frequently used for essential attributes (names, identifiers, creation dates).
- In PostgreSQL, adding NOT NULL requires that existing rows do not contain NULL for that column.
- NOT NULL can be combined with DEFAULT to simplify inserts.

Example:

```
ALTER TABLE hr.employees ALTER COLUMN last_name SET NOT NULL;
```

5. ALTER TABLE

ALTER TABLE modifies an existing table definition: add or drop columns, change types, and manage constraints.

5.1 Add Column

General syntax:

```
ALTER TABLE table_name ADD COLUMN column_name data_type [DEFAULT expr] [NOT NULL];
```

Rules and notes:

- If NOT NULL is used, either the table must be empty or a DEFAULT must populate existing rows.
- Adding a column is fast in PostgreSQL when the DEFAULT is a constant (optimizations apply).

Example:

```
ALTER TABLE hr.employees ADD COLUMN email VARCHAR(80);
```

5.2 Modify Column Type

General syntax:

```
ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type;
```

Rules and notes:

- The change must be compatible or explicitly castable.
- Large tables may require time to rewrite data depending on the conversion.

Example:

```
ALTER TABLE hr.employees ALTER COLUMN first_name TYPE VARCHAR(40);
```

5.3 Set or Drop DEFAULT

General syntax:

```
ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT expression;
ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;
```

Example:

```
ALTER TABLE hr.employees ALTER COLUMN hire_date SET DEFAULT CURRENT_DATE;
```

5.4 Set or Drop NOT NULL

General syntax:

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;
```

Example:

```
ALTER TABLE hr.employees ALTER COLUMN first_name SET NOT NULL;
```

5.5 Drop Column

General syntax:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Rules and notes:

- The column must not be required by other objects (views, constraints, indexes).
- Use CASCADE carefully if you want to automatically drop dependent objects.

Example:

```
ALTER TABLE hr.employees DROP COLUMN birth_date;
```

6. RENAME Objects

In PostgreSQL, renaming is typically done through ALTER statements.

General syntax:

```
ALTER TABLE old_table_name RENAME TO new_table_name;
ALTER TABLE table_name RENAME COLUMN old_col TO new_col;
```

Example:

```
ALTER TABLE hr.employees RENAME TO staff;
```

7. DROP TABLE

DROP TABLE removes a table definition and all of its rows.

General syntax:

```
DROP TABLE [IF EXISTS] table_name [CASCADE | RESTRICT];
```

Rules and notes:

- RESTRICT (default) refuses the drop if other objects depend on the table.
- CASCADE drops dependent objects (use with caution).
- This action is transactional in PostgreSQL (can be rolled back before COMMIT).

Example:

```
DROP TABLE article;
```

8. TRUNCATE TABLE

TRUNCATE quickly removes all rows from a table. It is faster than DELETE without WHERE because it avoids row-by-row processing.

General syntax:

```
TRUNCATE TABLE table_name [RESTART IDENTITY] [CASCADE];
```

Rules and notes:

- RESTART IDENTITY resets identity/serial counters.
- CASCADE truncates referencing tables as well.
- TRUNCATE is transactional in PostgreSQL, but it acquires stronger locks than DELETE.

Example:

```
TRUNCATE TABLE hr.audit_log;
```

9. COMMENT ON (Documentation inside the Catalog)

COMMENT ON stores documentation strings in PostgreSQL system catalogs. This is useful for data dictionaries and tooling.

General syntax:

```
COMMENT ON TABLE table_name IS 'text';
COMMENT ON COLUMN table_name.column_name IS 'text';
```

Example:

```
COMMENT ON TABLE hr.employees IS 'Employees master data';
COMMENT ON COLUMN hr.employees.salary IS 'Monthly salary in DZD (example)';
```

10. DCL: Privileges (GRANT / REVOKE)

DCL controls who can access or modify database objects. In practical work, you may simulate roles and privileges.

General syntax:

```
GRANT privilege_list ON object TO role;
REVOKE privilege_list ON object FROM role;
```

Example:

```
GRANT SELECT, INSERT, UPDATE ON hr.employees TO hr_user;
REVOKE UPDATE ON hr.employees FROM hr_user;
```

PART II — DML: Data Manipulation Language

1. INSERT

INSERT adds new rows to a table.

General syntax:

```
INSERT INTO table_name (col1, col2, ..., coln)
VALUES (val1, val2, ..., valn);
```

Rules and notes:

- Omitted columns take DEFAULT values if defined, otherwise NULL (unless NOT NULL).
- Values must satisfy all constraints (PK, FK, CHECK, NOT NULL, UNIQUE).
- Use RETURNING to immediately retrieve generated identifiers.

Example:

```
INSERT INTO hr.employees(first_name, last_name, salary, dept_id, job_id)
VALUES ('Ilyes', 'Ait', 105000, 1, 'DEV')
RETURNING emp_id;
```

2. INSERT … SELECT

INSERT can also populate a table using the result of a query.

General syntax:

```
INSERT INTO table_name (col1, col2, ...)
SELECT expr1, expr2, ...
```

```
FROM other_table
WHERE condition;
```

Example (create a table of high salaries):

```
CREATE TABLE IF NOT EXISTS hr.high_salary (
  emp_id INTEGER PRIMARY KEY,
  salary NUMERIC(10,2) NOT NULL
);

INSERT INTO hr.high_salary(emp_id, salary)
SELECT emp_id, salary
FROM hr.employees
WHERE salary >= 110000;
```

3. UPDATE

UPDATE modifies existing rows.

General syntax:

```
UPDATE table_name
SET col1 = expression1,
    col2 = expression2,
    ...
WHERE condition;
```

Rules and notes:

- Without WHERE, all rows are updated (dangerous in practice).
- Expressions can reference old column values (e.g., salary = salary * 1.05).
- Use RETURNING to see modified rows immediately.

Example:

```
UPDATE hr.employees
SET salary = salary * 1.05
WHERE dept_id = 1
RETURNING emp_id, salary;
```

4. DELETE

DELETE removes rows.

General syntax:

```
DELETE FROM table_name
WHERE condition;
```

Rules and notes:

- Without WHERE, all rows are deleted (but the table structure remains).
- Foreign keys and ON DELETE actions determine what happens to dependent rows.
- DELETE is slower than TRUNCATE for full-table removal because it processes rows individually.

Example:

```
DELETE FROM hr.employees
WHERE last_name = 'Meziane';
```