Chapter 2 — Advanced SQL

Procedural SQL programming

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 12:43:21



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

Creative Commons License — BY-NC-SA 4.0 International

Course conventions

Each page has a **colour** that shows its role:

Blue: the lecture, to read and to run.

Orange: an exercise or a recall; look for the answer.

Green: the solution of the exercise, on the next page.

- **Method**: answer the exercise **before** turning the page.
- **Review**: this link takes you to the lecture page that holds the answer; the **"↩ Back to slide N"** button (or the R key) brings you back to the exercise (in a PDF: Alt+←).
- **Interactive version**: the **Run** button runs the SQL example in the browser; "Reset database" restores the initial data.

Outline

1. Introduction
2. A first example: student records (Lab 2)
3. PL/pgSQL basics
4. Control structures
5. Functions
6. Procedures
7. Exceptions
8. Triggers
9. Transactions
10. Cursors
11. Dynamic SQL
12. Conclusion

General objective

- The **SQL language** (first standard: **SQL-86**, ANSI X3.135-1986 then ISO 9075:1987) is **declarative**: we describe the **expected result**, and the DBMS chooses how to obtain it.
- Applications need richer processing: **chain** statements, **decide** (conditions), **repeat** (loops), **react to errors**.
- The **SQL/PSM — Persistent Stored Modules** standard (**ISO/IEC 9075-4:1996**) adds these mechanisms: this is **procedural SQL programming**.
- Programs are **stored and executed in the DBMS**: fewer client-server exchanges, centralized business rules, transaction control and access rights managed by the server.
- **Goal of the chapter**: learn to write, step by step, blocks, functions, procedures and triggers in **PL/pgSQL**.

[ISO/IEC 9075-4:2016 — SQL/PSM; Silberschatz et al., 2019]

Origin of procedural SQL

- **Oracle PL/SQL** (1988) is the first major procedural language built into a DBMS; it strongly influenced what followed.
- The standardization work that followed SQL-92 led to **SQL/PSM** (**ISO/IEC 9075-4:1996**), later included in **SQL:1999**. It defines:
  - `BEGIN ... END` blocks and **local variables**,
  - **control** structures (`IF`, `CASE`, `WHILE`, `LOOP`),
  - **error handling** (`DECLARE ... HANDLER`),
  - stored **functions** and **procedures**.
- Each DBMS has its own dialect:
  - **PL/pgSQL** (PostgreSQL, 1998), very close to PL/SQL,
  - **MySQL** (stored procedures since version 5.0, 2005) and **IBM Db2 SQL PL**, close to SQL/PSM,
  - **Transact-SQL** (Microsoft SQL Server).
- **Consequence**: the principles are shared, but procedural code is **hardly portable** from one DBMS to another.

[ISO/IEC 9075-4:1996; Elmasri & Navathe (2016); Gardarin (2003)]

Philosophy of procedural SQL

- **Principle**: bring the **application logic** closer to the **data** by running the processing directly **inside the DBMS**.
- **Effects**: fewer **network exchanges**, lower latency, better **transactional consistency** and **security**.
- The DBMS becomes an **integrated application engine**, with **stored procedures**, **triggers** and **business rules** applied to all applications.
- **Duality preserved**:
  - **declarative** SQL: describes the *desired result*,
  - **procedural** SQL: defines the *logical sequence to obtain it*.
- **Golden rule**: if a single SQL query is enough, use it; procedural code is for when you must **chain, decide or react**.

[Silberschatz et al. (2019); ISO/IEC 9075-4:2016]

Course positioning

- **Course scope**: all **practical examples** use **PostgreSQL** and its **PL/pgSQL** language; they were tested on PostgreSQL 18.
- **Theoretical reference**: the **SQL standards** (**ISO/IEC 9075**, including **SQL/PSM**); PostgreSQL-specific features are pointed out.
- **Why PostgreSQL?**
  - Reliable **open source**, backed by a large community.
  - **Largely compliant with the SQL standards**.
  - **Industrial-grade system**: robust, efficient and feature-rich (replication, JSONB, FDW...).
- **Worldwide adoption**: PostgreSQL is used in production in critical environments, for example:
  - **Amazon**: *Aurora* and *AWS RDS for PostgreSQL*.
  - **Google**: *Cloud SQL for PostgreSQL*.
  - **Meta (Instagram)**: transactional storage based on PostgreSQL.
- **Method**: we first solve **Lab 2**, presenting the notions informally; we then **formalize** them one by one, each with a short example; run each example and watch the messages.

[ISO/IEC 9075; PostgreSQL documentation; AWS & Google Cloud]

Start of the example: student records

Start of the example

Lab 2 "Student records", solved step by step:
the problem, the constraints, then four programming steps.

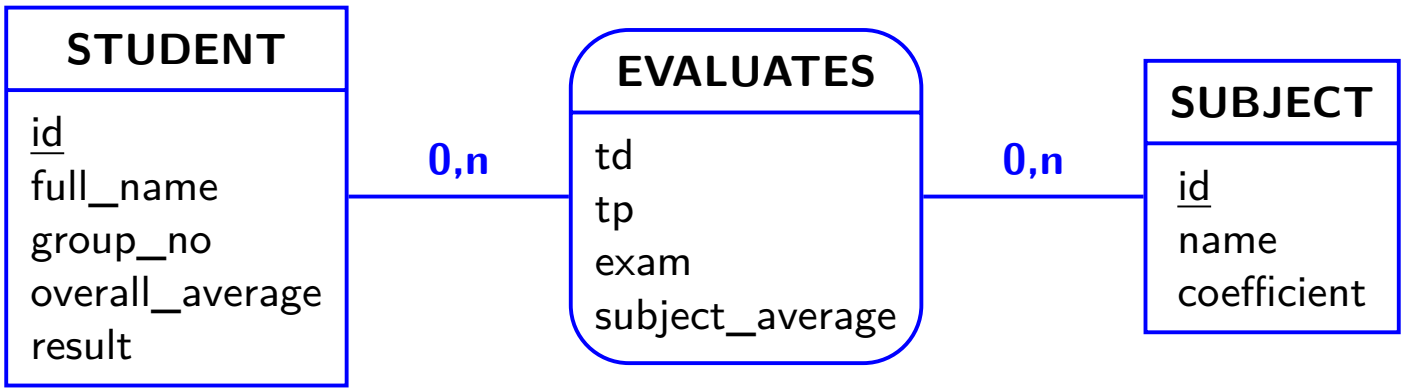
The example ends on the page "End of the example".

The problem (1/3)

Before formalizing, let us solve **Lab 2 "Academic records"** together.

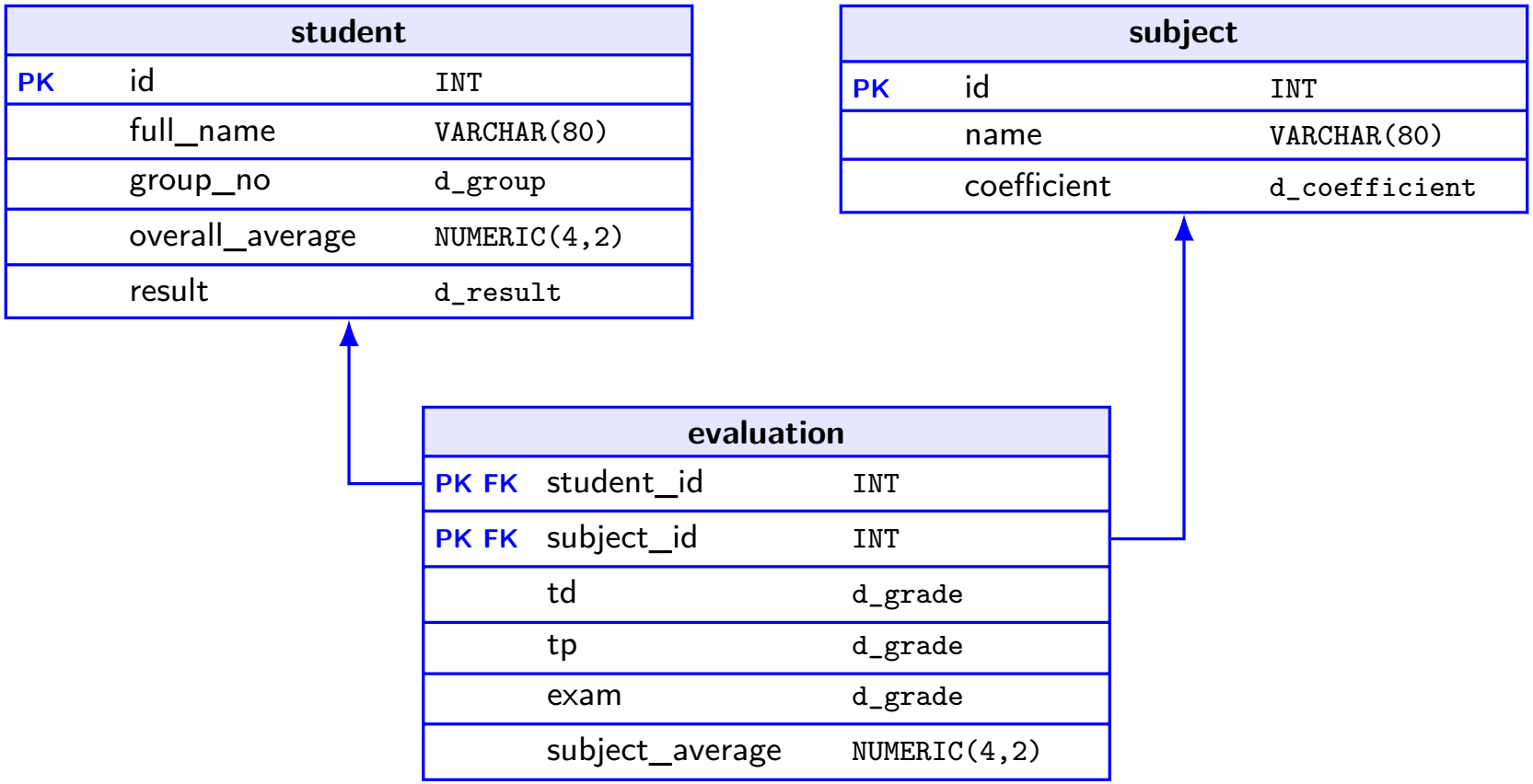
- The department's records office wants a database that stores the **students**, the **subjects** and the **grades** (tutorial, lab, exam) of each student in each subject.
- The database must **compute by itself** the average of each subject, the overall average of each student and the **result** (`PASSED` or `FAILED`).
- These values must stay **correct at all times**: at each entry, correction or deletion of a grade, whatever application is used.

The conceptual model (2/3)



- Two entities, **STUDENT** and **SUBJECT**, linked by the association **EVALUATES**.
- The grades and the subject average belong to the association: they concern **one** student in **one** subject.
- Cardinalities 0,n: a student can be evaluated in several subjects, a subject concerns several students.

The logical model: the database schema (3/3)



The association **EVALUATES** becomes the table **evaluation**, whose primary key (**PK**) is the two foreign keys (**FK**; the arrow leads to the referenced table).

Structural constraints (1/5)

They follow from the **model**: identifiers, links between tables, mandatory values.

Constraint	**Coded in the database**
Each student is uniquely identified	`PRIMARY KEY (id)`
Each subject is uniquely identified; its name is unique	`PRIMARY KEY (id)`, `UNIQUE (name)`
An evaluation concerns a student and a subject that exist	`FOREIGN KEY ... REFERENCES`
Only one evaluation per student and per subject	`PRIMARY KEY (student_id, subject_id)`
The three grades are mandatory	`NOT NULL`

All of them are coded **declaratively**, in the `CREATE TABLE`.

Business constraints: allowed values (2/5)

Business constraints come from the **business rules** of the records office. First family: the **allowed values**.

Business rule	**Coded in the database**
The group is between 1 and 5	domain `d_group` (`CHECK`)
The coefficient is between 1 and 5	domain `d_coefficient` (`CHECK`)
Each grade is between 0 and 20	domain `d_grade` (`CHECK`)
The result is `PASSED` or `FAILED`	domain `d_result` (`CHECK`)

They are coded **declaratively**, with domains.

Business constraints: computation rules (3/5)

Second family: the **computed values**.

$$\texttt{subject_average} = \frac{\frac{td + tp}{2} + exam}{2}$$

$$\texttt{overall_average} = \frac{\sum(subject_average \times coefficient)}{\sum coefficient}$$

$$\texttt{result} = \texttt{PASSED} \text{ if } \texttt{overall_average} \geq 10, \text{ else } \texttt{FAILED}$$

They are coded **procedurally**: a `BEFORE` trigger computes the average of a subject, an `AFTER` trigger recomputes the overall average and the result.

Computation rules: a worked example (4/5)

BENALI KARIM

Subject	td	tp	exam	average	coefficient
Advanced databases	14	15	13	13.75	5
Networks	12	11	10	10.75	3
English	16	15	14	14.75	1

$$overall_average = \frac{13.75 \times 5 + 10.75 \times 3 + 14.75 \times 1}{5 + 3 + 1} = 12.86$$

$$12.86 \geq 10 \quad \rightarrow \quad \texttt{PASSED}$$

Declarative or procedural? (5/5)

- **Declarative**: we **state** the rule, the DBMS checks it at each write.
  - `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `NOT NULL`, `CHECK`, `DOMAIN`
  - to be preferred whenever possible
- **Procedural**: we **program** what must be done.
  - functions, triggers
  - needed when the rule **computes** a value, or depends on **several rows** or **several tables**
- The computation rules are **procedural**: this is the subject of this chapter.

The approach

Four steps:

1. create the domains, the tables and the data (structural constraints and allowed values);
2. write the computation of an average, then turn it into a **function**;
3. have the computation rules applied **automatically**, each time grades are entered (**triggers**);
4. test.

Each notion first appears **in context**; it will be formalized in the rest of the course.

Step 1: the domains (1/3)

```sql
CREATE DOMAIN d_group       AS SMALLINT       -- tutorial group number
    CHECK (VALUE BETWEEN 1 AND 5);

CREATE DOMAIN d_coefficient AS SMALLINT       -- weight of a subject
    CHECK (VALUE BETWEEN 1 AND 5);

CREATE DOMAIN d_grade       AS NUMERIC(4,2)   -- grade out of 20
    CHECK (VALUE BETWEEN 0 AND 20);

CREATE DOMAIN d_result      AS VARCHAR(10)    -- final decision
    CHECK (VALUE IN ('PASSED', 'FAILED'));
```

- A **domain** is a type with a rule attached.
- The rule is written **only once** and applies to all the columns of that type.

Step 1: the tables (2/3)

```
CREATE TABLE student (
    id                INT             GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    full_name         VARCHAR(80)     NOT NULL,
    group_no          d_group         NOT NULL,
    overall_average   NUMERIC(4,2),                   -- computed by the database
    result            d_result                        -- computed by the database
);

CREATE TABLE subject (
    id                INT             GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    name              VARCHAR(80)     NOT NULL UNIQUE,
    coefficient       d_coefficient   NOT NULL
);

CREATE TABLE evaluation (
    student_id        INT             REFERENCES student(id) ON DELETE CASCADE,
    subject_id        INT             REFERENCES subject(id) ON DELETE CASCADE,
    td                d_grade         NOT NULL,
    tp                d_grade         NOT NULL,
    exam              d_grade         NOT NULL,
    subject_average   NUMERIC(4,2),                   -- computed by the database
    PRIMARY KEY (student_id, subject_id)
);
```

Step 1: the data (3/3)

```sql
INSERT INTO subject (name, coefficient) VALUES
    ('Advanced databases',        5),
    ('Networks',                  3),
    ('English',                   1);

INSERT INTO student (full_name, group_no) VALUES
    ('BENALI KARIM',              1),
    ('MANSOURI SARRA',            2),
    ('TOUATI OMAR',               1);

SELECT * FROM student;       -- averages and results still empty
```

- `IDENTITY` numbers the rows automatically: 1, 2, 3.
- The grades will be entered in step 4, once the database knows how to compute.

Step 2: computing an average in a block (1/2)

```
DO $$
DECLARE
    v_td      NUMERIC := 14;
    v_tp      NUMERIC := 15;
    v_exam    NUMERIC := 13;
    v_average NUMERIC;
BEGIN
    v_average := round(((v_td + v_tp) / 2 + v_exam) / 2, 2);
    RAISE NOTICE 'Subject average: %', v_average;
END;
$$;
```

What we see (formalized in the "PL/pgSQL basics" section):

- `DO $$ ... $$`: a **block** of statements executed by the server;
- `DECLARE`: **variables**; `:=` gives them a value;
- `RAISE NOTICE`: displays a message.

Step 2: a reusable function (2/2)

```
CREATE OR REPLACE FUNCTION subject_average(
    p_td     NUMERIC,
    p_tp     NUMERIC,
    p_exam   NUMERIC)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN round(((p_td + p_tp) / 2 + p_exam) / 2, 2);
END;
$$;

SELECT subject_average(14, 15, 13);
SELECT subject_average(6, 7, 5);
```

What we see (formalized in the "Functions" section):

- a **function** keeps the computation in the database, under a name, with **parameters**;
- `RETURN` returns the result; the function is used in a `SELECT`.

Step 3: the average of a subject (1/3)

```
CREATE OR REPLACE FUNCTION compute_subject_average()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    NEW.subject_average := subject_average(NEW.td, NEW.tp, NEW.exam);
    RETURN NEW;
END;
$$;

CREATE TRIGGER trg_compute_subject_average
BEFORE INSERT OR UPDATE OF td, tp, exam ON evaluation
FOR EACH ROW
EXECUTE FUNCTION compute_subject_average();
```

What we see (formalized in the "Triggers" section):

- a **trigger**: "**before** each entry of grades, execute this function";
- `NEW` is the row being written: its average is filled in.

Step 3: the overall average (2/3)

```
CREATE OR REPLACE FUNCTION recompute_overall_average()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_id      INT := COALESCE(NEW.student_id, OLD.student_id);
    v_average NUMERIC;
BEGIN
    -- weighted average of the student concerned
    SELECT SUM(e.subject_average * s.coefficient) / SUM(s.coefficient)
    INTO   v_average
    FROM   evaluation e
    JOIN   subject s ON s.id = e.subject_id
    WHERE  e.student_id = v_id;

    UPDATE student
    SET    overall_average = v_average,
           result = CASE WHEN v_average IS NULL THEN NULL
                         WHEN v_average >= 10   THEN 'PASSED'
                         ELSE                        'FAILED'
                    END
    WHERE  id = v_id;
    RETURN NULL;
END;
$$;
```

Step 3: the second trigger (3/3)

```
CREATE TRIGGER trg_recompute_overall_average
AFTER INSERT OR UPDATE OR DELETE ON evaluation
FOR EACH ROW
EXECUTE FUNCTION recompute_overall_average();
```

What we see (formalized in the following sections):

- `SELECT ... INTO` stores the result of a query in a **variable**;
- `CASE` chooses `PASSED` or `FAILED`;
- `AFTER`: the trigger acts **after** the evaluation is written;
- `OLD`: the old row, the only one available after a **deletion**.

The two triggers work in sequence: an entered grade updates the subject average, then the overall average and the result.

Step 4: entering the grades (1/3)

```
INSERT INTO evaluation (student_id, subject_id, td, tp, exam) VALUES
    (1, 1, 14, 15, 13), (1, 2, 12, 11, 10), (1, 3, 16, 15, 14),   -- Karim
    (2, 1, 10, 10, 10), (2, 2, 11,  9, 10),                       -- Sarra
    (3, 1,  6,  7,  5), (3, 2,  8,  6,  7);                       -- Omar

SELECT full_name, overall_average, result FROM student ORDER BY id;
```

Run it: we entered **only grades**; the averages and results were computed by the database.

Step 4: correcting, deleting (2/3)

```sql
-- Omar gets 20 in the ADB resit exam
UPDATE evaluation SET exam = 20
WHERE  student_id = 3 AND subject_id = 1;

SELECT full_name, overall_average, result
FROM   student WHERE id = 3;

-- His Networks grade is deleted
DELETE FROM evaluation
WHERE  student_id = 3 AND subject_id = 2;

SELECT full_name, overall_average, result
FROM   student WHERE id = 3;
```

Updates and deletions trigger the same recomputations: the data always stay **consistent**.

Step 4: an invalid grade (3/3)

```
INSERT INTO evaluation (student_id, subject_id, td, tp, exam)
VALUES (2, 3, 25, 10, 10);
```

- The grade 25 is rejected by the domain `d_grade`: nothing is saved.
- The rules are **in the database**: they apply to **all** the applications that use it.

Test page: the student records database (1/3)

▶ Delete ▶ Create all ▶ List ▶ List with grades

Suggested order: delete the database, create everything (the full script replays steps 1 to 4), then list. The code of the buttons is given on the next pages; in the interactive course, the server's response is displayed under the buttons.

Test page: looking at the students (2/3)

The students, without their grades:

```
SELECT id, full_name, group_no, overall_average, result
FROM   student
ORDER  BY full_name;
```

The students with their grades:

```
SELECT s.full_name, j.name AS subject, e.td, e.tp, e.exam, e.subject_average
FROM   student s
JOIN   evaluation e ON e.student_id = s.id
JOIN   subject j    ON j.id = e.subject_id
ORDER  BY s.full_name, j.name;
```

Test page: deleting the database (3/3)

```
DROP TABLE    IF EXISTS evaluation, student, subject CASCADE;

DROP FUNCTION IF EXISTS compute_subject_average(),
                        recompute_overall_average(),
                        subject_average(NUMERIC, NUMERIC, NUMERIC);

DROP DOMAIN   IF EXISTS d_group, d_coefficient, d_grade, d_result;
```

- Objects are deleted in the reverse order of creation: the tables (and their triggers), then the functions, then the domains.
- `IF EXISTS` avoids an error when the object does not exist yet.

Summary: what we used

Notion	**Step**	**Formalized in the section**
domain, `CHECK`	1	(SQL reminder)
block, variables, `RAISE`	2	PL/pgSQL basics
function, parameters, `RETURN`	2	Functions
trigger, `NEW`, `OLD`	3	Triggers
`SELECT ... INTO`	3	PL/pgSQL basics
`CASE`	3	Control structures

Rest of the course: each notion is taken up one by one, on a second example (accounts and sales).

End of the example: student records

End of the example

The rest of the chapter takes up each notion, one at a time, on a second example: accounts, products and sales.

Back to the start: "Start of the example".

Second example: the accounts (1/3)

To formalize each notion, a second example: **accounts** and the **log** of their operations.

```
CREATE TABLE accounts (
    id              INT             PRIMARY KEY,
    name            TEXT            NOT NULL,
    balance         NUMERIC(15,2)   NOT NULL DEFAULT 0,
    modified_at     TIMESTAMP
);

CREATE TABLE account_log (
    log_id          SERIAL          PRIMARY KEY,
    account_id      INT,
    operation       TEXT,                   -- INSERT, UPDATE or DELETE
    old_balance     NUMERIC(15,2),
    new_balance     NUMERIC(15,2),
    logged_at       TIMESTAMP       DEFAULT now()
);
```

The log will be filled **automatically** by triggers.

Second example: products and sales (2/3)

```
CREATE TABLE products (
    id              INT             PRIMARY KEY,
    name            TEXT            NOT NULL UNIQUE,
    price           NUMERIC(10,2)   NOT NULL,
    stock           INT             NOT NULL CHECK (stock >= 0)
);

CREATE TABLE sales (
    id              SERIAL          PRIMARY KEY,
    product_id      INT             NOT NULL REFERENCES products(id),
    quantity        INT             NOT NULL CHECK (quantity > 0),
    price           NUMERIC(10,2),              -- unit price
    amount          NUMERIC(12,2)               -- quantity * price
);
```

Each sale reduces the **stock** of the product sold ("Triggers" section).

Second example: the initial data (3/3)

```
INSERT INTO accounts (id, name, balance) VALUES
    (101, 'Amina',  1000),
    (102, 'Karim',   250),
    (103, 'Lina',    -50);

INSERT INTO products (id, name, price, stock) VALUES
    (1,   'Keyboard', 2500, 10),
    (2,   'Mouse',     800, 25),
    (3,   'Monitor', 18000,  4);

INSERT INTO sales (product_id, quantity, price, amount) VALUES
    (1,   2,  2500,  5000),
    (2,   3,   800,  2400),
    (3,   1, 18000, 18000);
```

- Account 103 is overdrawn: it will be used in the condition examples.
- The examples modify the data: “Reset database” puts them back in this state.

Declarative SQL and procedural SQL

Declarative SQL: we say **what we want**; the DBMS decides how to compute it.

```sql
SELECT SUM(amount) FROM sales;
```

Procedural SQL: we describe **the steps** of the computation, executed **on the server**.

```sql
DO $$
DECLARE
    total NUMERIC := 0;
    r     RECORD;
BEGIN
    FOR r IN SELECT amount FROM sales LOOP
        total := total + r.amount;          -- step repeated for each sale
    END LOOP;
    RAISE NOTICE 'Total sales = %', total;
END;
$$;
```

Key points: here, `SELECT SUM` is enough. Procedural code is for when the processing of a row depends on **conditions**, **errors** or **other tables**.

First block: DO and $$ (1/3)

The smallest PL/pgSQL program: an **anonymous block** executed with `DO`.

```
DO $$
BEGIN
    RAISE NOTICE 'Hello from the PostgreSQL server!';
END;
$$;
```

Explanation:

- `DO` immediately executes an **anonymous** block: it is not stored in the database.
- `$$ ... $$` delimits the code of the block (*dollar quoting*): inside, apostrophes are written normally. The delimiter can be named: `$body$ ... $body$`.
- `BEGIN ... END;` contains the statements; each statement ends with `;`.
- `RAISE NOTICE` displays a message (detailed later).

Structure of a block (2/3)

```
[ <<label>> ]
[ DECLARE
    declarations ; ]                -- variables (optional)
BEGIN
    statements ;                    -- the only mandatory part
[ EXCEPTION
    WHEN condition THEN
        error_handling ; ]          -- (optional)
END [ label ] ;
```

Key points:

- Blocks can be **nested**; this `BEGIN` does **not** start a transaction.
- SQL/PSM standard (and MySQL): errors handled by `DECLARE ... HANDLER`; PL/pgSQL: `EXCEPTION` section.

Two kinds of blocks (3/3)

- **Anonymous block** (`DO`): not stored, run once; ideal for testing or for a one-off task.
- **Named block**: body of a **function**, a **procedure** or a **trigger**; stored in the database and reusable.

Declaring and assigning variables

```
DO $$
DECLARE
    v_name    TEXT;                          -- not initialized: NULL
    v_balance NUMERIC(15,2) := 0;            -- initialization
    v_rate    CONSTANT NUMERIC := 0.15;      -- constant: cannot be modified
    v_id      accounts.id%TYPE := 101;       -- type of accounts.id
BEGIN
    v_balance := v_balance + 100;            -- assignment
    v_balance := v_balance * (1 + v_rate);
    RAISE NOTICE 'Account %: computed balance = %', v_id, v_balance;
END;
$$;
```

Key points:

- Declaration: `name type [:= value];` (without a value: `NULL`); `CONSTANT` forbids any modification.
- `table.column%TYPE` reuses the type of a column; prefixes `v_` and `p_`: no confusion with the columns.

Commenting your code

```
-- Single-line comment
v_balance := v_balance + 100;    -- end-of-line comment

/*
  Multi-line comment:
  explains a group of statements
*/
v_balance := v_balance * (1 + v_rate);
```

Good practices:

- Explain the **why** (the business rule), not the **what** (the code already says it).
- Put a comment before each important part of a program.
- Keep end-of-line comments for short remarks.

Displaying messages: RAISE

```
DO $$
DECLARE
    v_balance NUMERIC(15,2) := 120;
BEGIN
    RAISE NOTICE 'Current balance: %', v_balance;
    v_balance := v_balance - 50;
    RAISE NOTICE 'Withdrawal of %: new balance = %', 50, v_balance;
    RAISE WARNING 'Low balance: %', v_balance;
END;
$$;
```

Explanation:

- Each % in the message is replaced by the next argument, **in order**.
- Levels: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`: a simple message, execution **continues**.
- `RAISE EXCEPTION` signals an **error** and **stops** the block (**Exceptions** section).
- Use: follow the execution and fix problems (debugging).

Reading the database: SELECT INTO

```
DO $$
DECLARE
    v_name    accounts.name%TYPE;
    v_balance accounts.balance%TYPE;
BEGIN
    SELECT name, balance INTO v_name, v_balance
    FROM accounts
    WHERE id = 101;
    RAISE NOTICE '% has % DA', v_name, v_balance;
END;
$$;
```

Key points:

- `SELECT ... INTO` copies the row read into the variables, **in order**; no row: `NULL`, **without error**; `INTO STRICT` requires **exactly one** row.
- `PERFORM query` runs a `SELECT` whose result is ignored.

Exercise 1: which message?

Data: three accounts, Amina (1000 DA), Karim (250 DA) and Lina (−50 DA).

Question: which message does this block display?

```
DO $$
DECLARE
    v_count INT;
    v_total NUMERIC;
BEGIN
    SELECT count(*), sum(balance) INTO v_count, v_total
    FROM   accounts
    WHERE  balance > 0;
    RAISE NOTICE '% account(s), total: % DA', v_count, v_total;
END;
$$;
```

a) 3 account(s), total: 1200 DA
b) 2 account(s), total: 1250 DA
c) 1 account(s), total: 1000 DA

Choose, then run the block to check. Review: SELECT ... INTO

Solution of exercise 1

Answer: b) 2 account(s), total: 1250 DA.

- `WHERE balance > 0` keeps Amina (1000) and Karim (250); Lina (-50) is excluded.
- So `count(*)` is 2 and `sum(balance)` is $1000 + 250 = 1250$.
- `SELECT ... INTO` stores these two values in `v_count` and `v_total`, **in order**.

Recall: the basics

At the start of each section, three questions on the previous section.
Answer **from memory**, then click "Review" to check.

1. What is the only mandatory part of a PL/pgSQL block?
   Review: Structure of a block
2. What is the value of a variable declared without an initial value?
   Review: Declaring and assigning variables
3. What is `%TYPE` used for in a declaration?
   Review: Declaring and assigning variables

Condition: IF (1/2)

```
DO $$
DECLARE
    v_balance accounts.balance%TYPE;
BEGIN
    SELECT balance INTO v_balance FROM accounts WHERE id = 102;
    IF v_balance < 0 THEN
        RAISE NOTICE 'Account overdrawn';
    ELSIF v_balance < 500 THEN
        RAISE NOTICE 'Low balance';
    ELSE
        RAISE NOTICE 'Sufficient balance';
    END IF;
END;
$$;
```

Key points:

- Only the branch of the **first** true condition runs; `ELSIF` and `ELSE` are optional.
- **Pitfall:** a `NULL` condition is treated as **false**: for a non-existent account, the program displays "Sufficient balance"! Hence the value of testing `FOUND`.

Condition: CASE (2/2)

```
DO $$
DECLARE
    v_balance NUMERIC := 250;
    v_status  TEXT;
BEGIN
    CASE                                    -- "searched" form: conditions
        WHEN v_balance < 0   THEN v_status := 'overdrawn';
        WHEN v_balance < 500 THEN v_status := 'low';
        ELSE                      v_status := 'sufficient';
    END CASE;
    RAISE NOTICE 'Status: %', v_status;

    CASE v_status                           -- "simple" form: one value
        WHEN 'overdrawn' THEN RAISE NOTICE 'Alert the customer';
        WHEN 'low'       THEN RAISE NOTICE 'Suggest a transfer';
        ELSE                  RAISE NOTICE 'Nothing to do';
    END CASE;
END;
$$;
```

- The `CASE` **statement** ends with `END CASE` (the expression, with `END`).
- Without `ELSE` and no matching branch: error `CASE_NOT_FOUND`.

Testing a query: FOUND and ROW_COUNT

```
DO $$
DECLARE
    v_balance accounts.balance%TYPE;
    v_count   INT;
BEGIN
    SELECT balance INTO v_balance FROM accounts WHERE id = 999;
    IF NOT FOUND THEN                          -- no row read
        RAISE NOTICE 'Account 999 not found';
    END IF;

    UPDATE accounts SET balance = balance + 10 WHERE balance < 500;
    GET DIAGNOSTICS v_count = ROW_COUNT;       -- rows modified
    RAISE NOTICE '% account(s) credited', v_count;
END;
$$;
```

Key points:

- `FOUND` (automatic Boolean) is `TRUE` if the last SQL statement read or modified **at least one** row (`SELECT INTO`, `UPDATE`, `INSERT`, `DELETE`, `FETCH`...).
- `GET DIAGNOSTICS v = ROW_COUNT` gives the **number** of rows processed.

Simple loop: LOOP and EXIT (1/4)

```
DO $$
DECLARE
    i INT := 0;
BEGIN
    LOOP
        i := i + 1;
        EXIT WHEN i > 3;                    -- exit condition
        RAISE NOTICE 'Iteration %', i;
    END LOOP;
    RAISE NOTICE 'End of the loop';
END;
$$;
```

Key points:

- `LOOP ... END LOOP` repeats its content **forever**: an exit statement is required.
- `EXIT WHEN condition` leaves the loop as soon as the condition is true; it is equivalent to `IF condition THEN EXIT; END IF;`.
- Forgetting the `EXIT` gives an infinite loop: the query never ends (it must be cancelled).

Conditional loop: WHILE (2/4)

Problem: how many years does it take to double 1000 DA invested at 5 % per year?

```
DO $$
DECLARE
    v_capital NUMERIC := 1000;
    v_years   INT := 0;
BEGIN
    WHILE v_capital < 2000 LOOP
        v_capital := v_capital * 1.05;
        v_years   := v_years + 1;
    END LOOP;
    RAISE NOTICE 'Capital doubled in % years', v_years;
END;
$$;
```

Key points:

- `WHILE condition LOOP` repeats **as long as** the condition is true.
- The condition is tested **before** each iteration: if it is false at the start, the loop does not run.
- Use it when the number of iterations is **not known** in advance.

Loop over integers: FOR (3/4)

```
DO $$
BEGIN
    FOR i IN 1..3 LOOP
        RAISE NOTICE 'i = %', i;
    END LOOP;

    FOR i IN REVERSE 10..1 BY 3 LOOP
        RAISE NOTICE 'i = %', i;
    END LOOP;
END;
$$;
```

Key points:

- `FOR i IN start..end LOOP`: `i` takes each integer of the range in turn.
- `i` is declared **automatically** (integer) and exists only inside the loop.
- `REVERSE` goes backwards (start $\geq$ end); `BY` sets the step.
- Use it when the number of iterations is **known**; no `EXIT` needed.

Loop over a query: FOR ... IN SELECT (4/4)

```
DO $$
DECLARE
    r RECORD;                              -- will receive one row per iteration
BEGIN
    FOR r IN SELECT id, name, balance FROM accounts ORDER BY id LOOP
        IF r.balance < 0 THEN
            RAISE NOTICE '%: overdrawn by % DA', r.name, -r.balance;
        ELSE
            RAISE NOTICE '%: % DA', r.name, r.balance;
        END IF;
    END LOOP;
END;
$$;
```

Key points:

- The loop runs the query and puts each row into `r`, then stops after the last one.
- `r` must be declared: `RECORD` (any row) or `accounts%ROWTYPE` (a row of `accounts`).
- Each column is accessed with dot notation: `r.name`, `r.balance`.

Skipping an iteration: CONTINUE (1/2)

```
DO $$
BEGIN
    FOR i IN 1..6 LOOP
        CONTINUE WHEN i % 2 = 0;          -- skips even numbers
        RAISE NOTICE 'Odd: %', i;
    END LOOP;
END;
$$;
```

Key points:

- `CONTINUE WHEN condition` abandons the current iteration and moves **to the next one**.
- `EXIT` leaves the loop; `CONTINUE` stays in the loop.
- Useful to ignore some rows (missing value, case already handled...).

Nested loops and labels (2/2)

```
DO $$
BEGIN
    <<outer>>
    FOR i IN 1..3 LOOP
        FOR j IN 1..3 LOOP
            EXIT outer WHEN i * j = 4;          -- leaves BOTH loops
            RAISE NOTICE '% x % = %', i, j, i * j;
        END LOOP;
    END LOOP outer;
    RAISE NOTICE 'End';
END;
$$;
```

Key points:

- <<name>> gives a **label** to a loop or a block.
- `EXIT name` and `CONTINUE name` act on the named loop, not only on the innermost one.
- PL/pgSQL has **no** `GOTO` statement (unlike Oracle PL/SQL).

Exercise 2: how many messages?

Question: how many messages does this block display?

```
DO $$
BEGIN
    FOR i IN 1..10 LOOP
        CONTINUE WHEN i % 3 <> 0;     -- i is not a multiple of 3
        EXIT WHEN i > 7;
        RAISE NOTICE 'i = %', i;
    END LOOP;
END;
$$;
```

a) 10 **b)** 3 **c)** 2 **d)** none

Choose, then run the block to check. Review: CONTINUE and EXIT

Solution of exercise 2

Answer: c) 2 messages: `i = 3`, then `i = 6`.

- $i = 1, 2, 4, 5, 7, 8$: not a multiple of 3, `CONTINUE` moves to the next iteration.
- $i = 3$ and $i = 6$: multiples of 3 and ≤ 7, the message is displayed.
- $i = 9$: `EXIT` (since $9 > 7$) stops the loop before the display.

Exercise 3: which loops stop by themselves?

Question: which loops stop by themselves, **without** `EXIT` or `RETURN`? (several answers possible)

a) `LOOP ... END LOOP`

b) `WHILE v_balance < 2000 LOOP ... END LOOP`

c) `FOR i IN 1..10 LOOP ... END LOOP`

d) `FOR r IN SELECT ... LOOP ... END LOOP`

Review: LOOP and WHILE

Solution of exercise 3

Answer: b), c) and d).

a) `LOOP` runs **forever**: only `EXIT` or `RETURN` stops it.

b) `WHILE` stops as soon as the condition is no longer true.

c) `FOR i` stops at the end of the range (after $i = 10$).

d) `FOR r IN SELECT` stops after the last row of the query.

Recall: control structures

Answer **from memory**, then click "Review" to check.

1. Which loop do you choose when the number of iterations is not known in advance?
   Review: WHILE

2. After a `SELECT ... INTO`, how do you know whether a row was found?
   Review: FOUND and ROW_COUNT

3. An `IF` condition is `NULL`: which branch runs?
   Review: IF

Functions, procedures and triggers

A **named** block **stored** in the database takes one of these three forms:

Construct	Call, result	Description and use
Function	`SELECT`, `RETURN`	Returns a value, a row or a set of rows; usable **in a query**. Cannot execute `COMMIT`/`ROLLBACK`. **Reusable computations and transformations**.
Procedure	`CALL`, `OUT` parameters	Runs a complete task; can commit or roll back (`COMMIT`/`ROLLBACK`). **Encapsulates a business operation**.
Trigger	Automatic	Function fired by `INSERT`, `UPDATE`, `DELETE` (or `TRUNCATE`) on a table. **Validates, completes or logs the data**. Never called directly.

We study them in this order: functions, procedures, then triggers (after exceptions, which they use).

[ISO/IEC 9075-4:2016; PostgreSQL documentation]

What is a function? (1/4)

- A function is a **named** block, **stored** in the database, that receives **parameters** and **returns a result** with `RETURN`.
- It is called in a query: `SELECT function_name(...);`, or from another function.
- It runs **in the caller's transaction**: if it fails, the whole statement is cancelled.
- PostgreSQL accepts other languages: `sql`, PL/Python, PL/Perl, PL/v8 (JavaScript)...

A function in Python: PL/Python (2/4)

AI and PostgreSQL with PL/Python

PL/Python makes it possible to use Python and its libraries directly inside PostgreSQL:

- run machine learning models **on the server side**;
- process large volumes of data without transferring them to the application;
- write predictive or statistical analysis functions usable in a query.

Structure of a function (3/4)

```
CREATE OR REPLACE FUNCTION function_name(p_param1 TYPE, ...)
RETURNS return_type
LANGUAGE plpgsql
AS $$
DECLARE
    v_var TYPE;                -- local variables
BEGIN
    -- statements
    RETURN value;              -- returns the result, ends
END;
$$;

-- removal
DROP FUNCTION function_name(TYPE, ...);
```

Return types (4/4)

Return type	**Result**	**Call**
`INT`, `NUMERIC`, `TEXT...`	one value	`SELECT f(...);`
name of a table (`accounts`)	one row	`SELECT * FROM f(...);`
`TABLE(...)`, `SETOF ...`	several rows	`SELECT * FROM f(...);`
`VOID`	nothing	`SELECT f(...);`
`TRIGGER`	(special case)	called by a trigger

Returning a value (1/5)

```sql
CREATE OR REPLACE FUNCTION get_balance(p_id INT)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    v_balance accounts.balance%TYPE;
BEGIN
    SELECT balance INTO v_balance FROM accounts WHERE id = p_id;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Account % does not exist', p_id;   -- error: stop
    END IF;
    RETURN v_balance;
END;
$$;

SELECT get_balance(101);                          -- one value
SELECT name, get_balance(id) FROM accounts;       -- in a query
```

- The parameter `p_id` is used like a variable.
- `RETURN` returns the result and **ends** the function.
- `RAISE EXCEPTION` stops the function with a clear error message (instead of returning `NULL`).

Returning a row (2/5)

```
CREATE OR REPLACE FUNCTION get_account(p_id INT)
RETURNS accounts                        -- one row of the accounts table
LANGUAGE plpgsql
AS $$
DECLARE
    v_account accounts%ROWTYPE;         -- all the columns of accounts
BEGIN
    SELECT * INTO v_account FROM accounts WHERE id = p_id;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Account % does not exist', p_id;
    END IF;
    RAISE NOTICE 'Name: %, balance: %', v_account.name, v_account.balance;
    RETURN v_account;
END;
$$;

SELECT * FROM get_account(101);         -- one row, with its columns
```

- `accounts%ROWTYPE`: a variable that holds **an entire row** of the table.
- Each field is accessed with dot notation: `v_account.name`.
- `SELECT * FROM f()` presents the result in columns, like a table.

Returning a query: RETURN QUERY (3/5)

```
CREATE OR REPLACE FUNCTION sales_by_product()
RETURNS TABLE(product TEXT, total NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT p.name, SUM(s.amount)
    FROM sales s
    JOIN products p ON p.id = s.product_id
    GROUP BY p.name;
END;
$$;

SELECT * FROM sales_by_product() ORDER BY total DESC;
```

- `RETURNS TABLE(...)` describes the columns of the result.
- `RETURN QUERY` adds **all** the rows of the query to the result.
- **Pitfall:** the columns of `TABLE(...)` are also variables; prefixing the table columns (`p.name`, `s.amount`) avoids the "ambiguous reference" error.

Result row by row: RETURN NEXT (4/5)

```
CREATE OR REPLACE FUNCTION accounts_at_risk(p_threshold NUMERIC)
RETURNS SETOF accounts                    -- rows of accounts
LANGUAGE plpgsql
AS $$
DECLARE
    r accounts%ROWTYPE;
BEGIN
    FOR r IN SELECT * FROM accounts ORDER BY id LOOP
        IF r.balance < p_threshold THEN
            RETURN NEXT r;                -- adds r to the result and continues
        END IF;
    END LOOP;
    RETURN;                               -- end: the result is returned
END;
$$;

SELECT * FROM accounts_at_risk(500);
```

- `RETURN NEXT` **adds** a row to the result **without** ending the function; the final `RETURN` ends it.
- Useful when each row needs processing; otherwise `RETURN QUERY` (or a simple `SELECT`) is enough.

Function without a result: VOID (5/5)

```sql
CREATE OR REPLACE FUNCTION update_stock(p_id INT, p_qty INT)
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE products SET stock = stock + p_qty WHERE id = p_id;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Product % does not exist', p_id;
    END IF;
END;
$$;

SELECT update_stock(1, 10);        -- restocks product 1
```

- `RETURNS VOID`: the function acts on the data but returns nothing; `RETURN` is optional.
- A function cannot commit its own work (`COMMIT`): for a complete business operation, a **procedure** is preferred (next section).

Default parameters and volatility

```sql
CREATE OR REPLACE FUNCTION price_with_vat(
    p_price NUMERIC,
    p_vat   NUMERIC DEFAULT 0.19     -- optional parameter
)
RETURNS NUMERIC
LANGUAGE sql                         -- a single query: SQL is enough
IMMUTABLE                            -- same inputs => same result
AS $$
    SELECT round(p_price * (1 + p_vat), 2);
$$;

SELECT price_with_vat(1000);
SELECT price_with_vat(1000, 0.09);
SELECT price_with_vat(p_price => 500);     -- call naming the parameter
```

- `DEFAULT` makes a parameter optional; `=>` passes a parameter by its name.
- **Volatility** (helps the optimizer): `IMMUTABLE` (does not read the database), `STABLE` (reads without modifying), `VOLATILE` (default, may modify). A wrong declaration can give wrong results.
- `LANGUAGE sql` is enough for a function without variables or loops.

Exercise 4: which rows?

Data: Keyboard (2500 DA), Mouse (800 DA), Monitor (18000 DA); `price_with_vat(p)` returns $p \times 1.19$ (19 % VAT).

Question: which products does this query display?

```
SELECT name, price, price_with_vat(price) AS with_vat
FROM   products
WHERE  price_with_vat(price) > 3000
ORDER  BY with_vat;
```

a) Keyboard and Monitor
b) Monitor only
c) no product
d) an error

Choose, then run the query to check. Review: the price_with_vat function

Solution of exercise 4

Answer: b) Monitor only.

Product	**Price**	**With VAT**	$>$ **3000?**
Keyboard	2500	2975	no
Mouse	800	952	no
Monitor	18000	21420	**yes**

A function can be used in `WHERE` as in the `SELECT` list: no error.

Recall: functions

Answer **from memory**, then click "Review" to check.

1. What is the difference between `RETURN QUERY` and `RETURN NEXT`?
   Review: RETURN NEXT
2. What does a function declared `IMMUTABLE` promise?
   Review: Default parameters and volatility
3. What does a function declared `RETURNS SETOF accounts` return?
   Review: Return types

What is a procedure? (1/2)

- A procedure is a named, stored block, created with `CREATE PROCEDURE` (PostgreSQL 11 and later).
- It is called with `CALL procedure_name(...);`, never in a `SELECT`.
- It does not return a value with `RETURN`; it passes its results through `OUT` or `INOUT` parameters.
- **Key difference** with a function: it can **commit** (`COMMIT`) or **roll back** (`ROLLBACK`) its work (**Transactions** section).
- Use: a **complete business operation** (transfer, end-of-day closing, purging a history...).

Structure of a procedure (2/2)

```
CREATE OR REPLACE PROCEDURE procedure_name(
    IN    p_in    TYPE,          -- read by the procedure (default mode)
    INOUT p_inout TYPE,          -- read, modified and returned
    OUT   p_out   TYPE           -- returned (PostgreSQL 14 and later)
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- statements (no RETURN value)
END;
$$;

CALL procedure_name(value1, value2, NULL);    -- NULL in place of an OUT
```

- `IN`: input value; if modified in the procedure, it is not returned.
- `INOUT`: input value that comes back modified to the caller.
- `OUT`: value produced by the procedure; in the call, write `NULL` (or a variable in a block).

Adjusting the balance of an account (1/2)

```
CREATE OR REPLACE PROCEDURE adjust_balance(
    IN    p_id      INT,
    IN    p_amount  NUMERIC,
    INOUT p_balance NUMERIC DEFAULT NULL    -- will receive the new balance
)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE accounts
    SET    balance = balance + p_amount
    WHERE  id = p_id
    RETURNING balance INTO p_balance;       -- gets the value written
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Account % does not exist', p_id;
    END IF;
END;
$$;
```

- `UPDATE ... RETURNING ... INTO` modifies the row and reads the new value **in a single statement**.
- `DEFAULT NULL` makes the last parameter optional in the call.

Calling the procedure (2/2)

With `CALL`: the `INOUT` parameter is returned as a result row.

```
CALL adjust_balance(101, 200);
```

From a block: pass a **variable**, which receives the returned value.

```
DO $$
DECLARE
    v_balance NUMERIC;
BEGIN
    CALL adjust_balance(101, -200, v_balance);
    RAISE NOTICE 'New balance: %', v_balance;
END;
$$;
```

- A procedure cannot be used in a query: `SELECT adjust_balance(...)` causes an error.
- A negative amount performs a debit: the same procedure serves both operations.

Exercise 5: true or false?

Question: is each statement about **procedures** true or false?

a) A procedure is called with `CALL`.

b) A procedure can be used in a `SELECT`, like a function.

c) Called outside an explicit transaction, a procedure can execute `COMMIT`.

d) A procedure returns its results through `INOUT` or `OUT` parameters.

Review: What is a procedure?

Solution of exercise 5

a) **True**: a procedure is run with `CALL`.

b) **False**: only a **function** can appear in a `SELECT`.

c) **True**: called by `CALL` outside a `BEGIN ... COMMIT` block, it can commit (`COMMIT`) or roll back (`ROLLBACK`).

d) **True**: it has no `RETURN` of a value; its results go through `INOUT` or `OUT` parameters.

Recall: procedures

Answer **from memory**, then click "Review" to check.

1. Which statement calls a procedure?
   Review: Calling the procedure
2. How do you get back a value computed by a procedure?
   Review: Structure of a procedure
3. Can a function execute `COMMIT`? And a procedure?
   Review: Functions, procedures and triggers

What happens when an error occurs? (1/2)

- **Without handling**, an error (division by zero, duplicate key…) **stops** the block and **rolls back** the current transaction.
- The `EXCEPTION` section of a block lets you **catch** the error:
  1. execution immediately leaves the `BEGIN` part;
  2. the changes made **in this block** are **rolled back**;
  3. the first `WHEN` clause that matches the error is executed;
  4. execution resumes **after the** `END` of the block.
- If no clause matches, the error goes up to the enclosing block, then to the caller.
- A condition is designated by its **name** (`division_by_zero`, `unique_violation`…), by its code `SQLSTATE '23505'`, or by `OTHERS` (all errors).

[PostgreSQL documentation, PL/pgSQL — Trapping Errors; Appendix A (error codes)]

Catching an error (2/2)

```
DO $$
DECLARE
    v_total  NUMERIC := 1000;
    v_shares INT := 0;
    v_share  NUMERIC;
BEGIN
    v_share := v_total / v_shares;          -- error: division by zero
    RAISE NOTICE 'Share: %', v_share;       -- never executed
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'No shares: division impossible';
END;
$$;
```

Key points:

- The failing statement is abandoned, as is the rest of the `BEGIN` part.
- The `WHEN division_by_zero` clause runs, then the block ends **normally**: the error is "absorbed", the caller does not see it.
- Remove the `EXCEPTION` section and run it again: the error goes up to the client.

Catching several kinds of errors (1/2)

```
DO $$
BEGIN
    INSERT INTO accounts(id, name, balance) VALUES (101, 'Duplicate', 0);
EXCEPTION
    WHEN unique_violation THEN               -- SQLSTATE 23505
        RAISE NOTICE 'Account 101 already exists';
    WHEN not_null_violation THEN             -- SQLSTATE 23502
        RAISE NOTICE 'A mandatory column is empty';
    WHEN OTHERS THEN                         -- any other error
        RAISE NOTICE 'Unexpected error: %', SQLERRM;
        RAISE;                               -- re-raises the error
END;
$$;
```

- One `WHEN` clause per kind of error; the first one that matches is executed.
- The clauses are tested in order: `OTHERS` always comes **last**.

The most frequent errors (2/2)

Condition	SQLSTATE	Cause
`unique_violation`	23505	duplicate key value
`foreign_key_violation`	23503	referenced row missing
`not_null_violation`	23502	mandatory column empty
`check_violation`	23514	`CHECK` constraint not satisfied
`division_by_zero`	22012	division by zero
`no_data_found`	P0002	`INTO STRICT`: no row
`too_many_rows`	P0003	`INTO STRICT`: too many rows
`raise_exception`	P0001	`RAISE EXCEPTION` without a code

The first two characters of the `SQLSTATE` give the class: 23 constraint violated, 22 invalid data, `P0` PL/pgSQL error.

Signaling your own errors (1/2)

```
-- Simple form: a message with values
RAISE EXCEPTION 'Insufficient balance: % DA', v_balance;

-- Full form: error code, detail and hint
RAISE EXCEPTION 'Insufficient balance'
    USING ERRCODE = 'check_violation',           -- or '23514'
          DETAIL  = format('Balance %s, withdrawal %s', v_balance, p_amount),
          HINT    = 'Reduce the withdrawal amount.';
```

- `RAISE EXCEPTION` stops the block; the caller can catch it like any other error.
- Without `ERRCODE`, the code is `P0001` (`raise_exception`).
- `DETAIL` and `HINT` complete the message displayed by the client.

Re-raising an error, checking an assumption (2/2)

```
-- In a WHEN clause: re-raise the caught error
RAISE;

-- Check an assumption during development
ASSERT v_balance >= 0, 'The balance should never be negative';
```

- `RAISE;` alone, in a `WHEN` clause, re-raises the current error to the caller.
- `ASSERT` is used for debugging: it raises an error if the condition is false.

Message and code: SQLERRM, SQLSTATE (1/3)

```
DO $$
DECLARE
    v_x INT;
BEGIN
    v_x := 1 / 0;
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Message  : %', SQLERRM;
        RAISE NOTICE 'SQLSTATE : %', SQLSTATE;
END;
$$;
```

Key points:

- `SQLERRM`: text of the error message.
- `SQLSTATE`: standardized 5-character code; the first two give the class (22: invalid data, 23: constraint violated, `P0`: PL/pgSQL error).
- These two variables exist **only** in a `WHEN` clause.

More details: GET STACKED DIAGNOSTICS (2/3)

```
DO $$
DECLARE
    v_msg    TEXT;
    v_detail TEXT;
    v_table  TEXT;
BEGIN
    INSERT INTO sales(product_id, quantity, price, amount)
    VALUES (99, 1, 100, 100);                    -- product 99 does not exist
EXCEPTION
    WHEN foreign_key_violation THEN
        GET STACKED DIAGNOSTICS
            v_msg    = MESSAGE_TEXT,
            v_detail = PG_EXCEPTION_DETAIL,
            v_table  = TABLE_NAME;
        RAISE NOTICE 'Error : %', v_msg;
        RAISE NOTICE 'Detail: %', v_detail;
        RAISE NOTICE 'Table : %', v_table;
END;
$$;
```

- `GET STACKED DIAGNOSTICS` copies the information about the current error into variables.
- Other items: `RETURNED_SQLSTATE`, `CONSTRAINT_NAME`, `PG_EXCEPTION_CONTEXT`...

Where did the error occur? (3/3)

```
DO $$
DECLARE
    v_code    TEXT;
    v_msg     TEXT;
    v_context TEXT;
BEGIN
    PERFORM get_balance(999);                    -- non-existent account
EXCEPTION
    WHEN OTHERS THEN
        GET STACKED DIAGNOSTICS
            v_code    = RETURNED_SQLSTATE,
            v_msg     = MESSAGE_TEXT,
            v_context = PG_EXCEPTION_CONTEXT;
        RAISE NOTICE 'Code    : %', v_code;
        RAISE NOTICE 'Message: %', v_msg;
        RAISE NOTICE 'Context: %', v_context;
END;
$$;
```

- The error is raised **inside** the function `get_balance` and caught by the calling block.
- `PG_EXCEPTION_CONTEXT` gives the call stack: function and line number of the statement that failed.

Tips for exception handling (1/2)

- Catch only the errors you know how to **handle**; let the others go up.
- Put the **specific** conditions before `WHEN OTHERS`.
- A `WHEN OTHERS` that only displays a message **hides** the error: end it with `RAISE;` to re-raise it.
- Limit the scope: wrap only the statements concerned in a `BEGIN ... EXCEPTION ... END` block.

Tips for exception handling (2/2)

- Write **useful** messages: which operation, which values (`RAISE EXCEPTION 'Account % does not exist', p_id`).
- Use `GET STACKED DIAGNOSTICS` to log the detail and the context.
- A block with an `EXCEPTION` section costs more (implicit savepoint): do not put one in a loop without need.

Exercise 6: which message?

Question: what does this block display?

```
DO $$
DECLARE
    v_x INT;
BEGIN
    v_x := 10 / 0;
    RAISE NOTICE 'A';
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'B';
    WHEN OTHERS THEN
        RAISE NOTICE 'C';
END;
$$;
```

a) A **b)** A then B **c)** B **d)** B then C

Choose, then run the block to check. Review: What happens when an error occurs?

Solution of exercise 6

Answer: c) B.

- `10 / 0` raises the `division_by_zero` error: the rest of the `BEGIN` part is not run, so `A` is never displayed.
- PostgreSQL runs the **first** `WHEN` clause that matches: `B`.
- `WHEN OTHERS` is not reached: only one clause runs.

Recall: exceptions

Answer **from memory**, then click "Review" to check.

1. An `EXCEPTION` section catches an error: what happens to the work done in the block?
   Review: What happens when an error occurs?

2. Why end a `WHEN OTHERS` with `RAISE;`?
   Review: Tips for exception handling

3. Which variable gives the standard code of the error?
   Review: SQLERRM and SQLSTATE

A trigger: "when… then…" (1/3)

- A **trigger** is a rule: "**when** a given event occurs on a given table, **then** execute a given function".
- It is created in **two steps**:
  1. a **function** of type `RETURNS TRIGGER` (the "what to do");
  2. the `CREATE TRIGGER` statement (the "when"), which attaches it to a table.
- It runs **automatically**, whatever application modifies the table, and in the same transaction.
- Typical uses:
  - **complete** a row (modification date, computed amount),
  - **check** a business rule too complex for a `CHECK` constraint,
  - **log** the changes (audit),
  - **propagate** a change to another table (stock).

When does the trigger run? (2/3)

Timing	
`BEFORE`	before the write: can **modify** or **reject** the row
`AFTER`	after the write: used to **log** or **propagate**
`INSTEAD OF`	on a **view**: replaces the requested operation
Level	
`FOR EACH ROW`	one execution **per row** modified
`FOR EACH STATEMENT`	one execution **per statement**
Event	
`INSERT`, `UPDATE [OF col]`, `DELETE`, `TRUNCATE`	can be combined with `OR`

The syntax of CREATE TRIGGER (3/3)

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF } { INSERT | UPDATE | DELETE } [ OR ... ]
ON table_name
[ FOR EACH { ROW | STATEMENT } ]        -- STATEMENT by default
[ WHEN ( condition ) ]
EXECUTE FUNCTION function_name();
```

- Square brackets [ ] enclose the optional parts; { | } offers a choice.
- The function called must exist before the trigger.

Accessing the row: NEW and OLD (1/2)

In a `FOR EACH ROW` trigger function, two variables represent the row concerned:

Event	`NEW` (row after)	`OLD` (row before)
`INSERT`	yes	`NULL`
`UPDATE`	yes	yes
`DELETE`	`NULL`	yes

```
RAISE NOTICE 'Account %: % -> %', NEW.id, OLD.balance, NEW.balance;
```

The value returned by the trigger function (2/2)

Value returned by the function (`FOR EACH ROW` triggers):

- `BEFORE`: `RETURN NEW` writes the row (possibly modified); `RETURN NULL` **silently cancels** the operation on this row. For `DELETE`: `RETURN OLD`.
- `AFTER`: the value is ignored; write `RETURN NULL`.

Other useful variables: `TG_OP` (`'INSERT'`, `'UPDATE'`...), `TG_TABLE_NAME`, `TG_WHEN`.

First trigger: dating the changes

```sql
-- 1. The function: what to do
CREATE OR REPLACE FUNCTION set_modified_at()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    NEW.modified_at := now();   -- completes the row before it is written
    RETURN NEW;                 -- BEFORE: the row to write
END;
$$;

-- 2. The trigger: when to do it
CREATE TRIGGER trg_set_modified_at
BEFORE INSERT OR UPDATE ON accounts
FOR EACH ROW
EXECUTE FUNCTION set_modified_at();

-- 3. Test
UPDATE accounts SET balance = balance + 1 WHERE id = 101;
SELECT id, balance, modified_at FROM accounts WHERE id = 101;
```

- Only a `BEFORE` trigger can modify `NEW`.
- `modified_at` is filled in by the database, no longer by the application.

Logging insertions (1/5)

```sql
CREATE OR REPLACE FUNCTION log_account_insert()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO account_log(account_id, operation, new_balance)
    VALUES (NEW.id, 'INSERT', NEW.balance);
    RETURN NULL;                    -- AFTER: value ignored
END;
$$;

CREATE TRIGGER trg_log_insert
AFTER INSERT ON accounts
FOR EACH ROW
EXECUTE FUNCTION log_account_insert();

INSERT INTO accounts(id, name, balance) VALUES (104, 'Samir', 300);
SELECT * FROM account_log;          -- one INSERT row for account 104
```

- `AFTER`: we log a row that was **actually** inserted.
- `NEW` contains the inserted row; `logged_at` takes its default value (`now()`).

Logging balance changes (2/5)

```sql
CREATE OR REPLACE FUNCTION log_balance_update()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO account_log(account_id, operation, old_balance, new_balance)
    VALUES (NEW.id, 'UPDATE', OLD.balance, NEW.balance);
    RETURN NULL;
END;
$$;

CREATE TRIGGER trg_log_update
AFTER UPDATE OF balance ON accounts              -- only if SET balance = ...
FOR EACH ROW
WHEN (OLD.balance IS DISTINCT FROM NEW.balance) -- only if it changes
EXECUTE FUNCTION log_balance_update();
```

- `OLD.balance`: value before, `NEW.balance`: value after; `UPDATE OF balance` ignores the other `UPDATE`s.
- `WHEN (...)` filters **before** calling the function; `IS DISTINCT FROM` also compares `NULL`s.

Logging deletions (3/5)

```
CREATE OR REPLACE FUNCTION log_account_delete()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO account_log(account_id, operation, old_balance)
    VALUES (OLD.id, 'DELETE', OLD.balance);
    RETURN NULL;
END;
$$;

CREATE TRIGGER trg_log_delete
AFTER DELETE ON accounts
FOR EACH ROW
EXECUTE FUNCTION log_account_delete();

DELETE FROM accounts WHERE id = 104;
SELECT * FROM account_log ORDER BY log_id;
```

- `OLD` contains the deleted row; `NEW` is `NULL`.
- The log keeps a trace of a row that no longer exists in `accounts`.

One trigger for everything: TG_OP (4/5)

```
CREATE OR REPLACE FUNCTION audit_accounts()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO account_log(account_id, operation, new_balance)
        VALUES (NEW.id, TG_OP, NEW.balance);
    ELSIF TG_OP = 'UPDATE' THEN
        INSERT INTO account_log(account_id, operation, old_balance, new_balance)
        VALUES (NEW.id, TG_OP, OLD.balance, NEW.balance);
    ELSE                                                    -- DELETE
        INSERT INTO account_log(account_id, operation, old_balance)
        VALUES (OLD.id, TG_OP, OLD.balance);
    END IF;
    RETURN NULL;
END;
$$;
```

TG_OP gives the current operation: one function handles the three cases.

Replacing the three triggers with one (5/5)

```
DROP TRIGGER trg_log_insert ON accounts;
DROP TRIGGER trg_log_update ON accounts;
DROP TRIGGER trg_log_delete ON accounts;

CREATE TRIGGER trg_audit_accounts
AFTER INSERT OR UPDATE OR DELETE ON accounts
FOR EACH ROW
EXECUTE FUNCTION audit_accounts();
```

- Without the `DROP TRIGGER`s, each operation would be logged twice.
- `INSERT OR UPDATE OR DELETE`: a single trigger for the three events.

Checking a sale and updating the stock (1/2)

```
CREATE OR REPLACE FUNCTION check_sale()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_product products%ROWTYPE;
BEGIN
    SELECT * INTO v_product FROM products
    WHERE id = NEW.product_id FOR UPDATE;        -- reads and locks the product
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Product % does not exist', NEW.product_id;
    ELSIF v_product.stock < NEW.quantity THEN
        RAISE EXCEPTION 'Insufficient stock for % (% left)',
                        v_product.name, v_product.stock;
    END IF;
    NEW.price  := v_product.price;               -- completes the sale
    NEW.amount := NEW.quantity * v_product.price;
    UPDATE products SET stock = stock - NEW.quantity
    WHERE id = NEW.product_id;                   -- updates the stock
    RETURN NEW;                                  -- the sale is inserted
END;
$$;
```

Creating the trigger and testing it (2/2)

```
CREATE TRIGGER trg_check_sale
BEFORE INSERT ON sales
FOR EACH ROW
EXECUTE FUNCTION check_sale();

-- Sale accepted: price and amount computed, stock reduced
INSERT INTO sales(product_id, quantity) VALUES (2, 4);
SELECT * FROM sales ORDER BY id;
SELECT name, stock FROM products WHERE id = 2;

-- Sale rejected: insufficient stock
INSERT INTO sales(product_id, quantity) VALUES (3, 100);
```

Key points:

- `BEFORE` **completes** the row; `RAISE EXCEPTION` **rejects** it: neither the sale nor the stock changes.
- The rule applies to **all** the applications that insert sales.

Good practices for triggers (1/2)

- Prefer a **constraint** (`CHECK`, `FOREIGN KEY`, `UNIQUE`) when it is enough: simpler and safer.
- `BEFORE` to modify or reject a row; `AFTER` to log or propagate.
- Always return the right value: `NEW` (or `OLD` for `DELETE`) in `BEFORE`, `NULL` in `AFTER`.
- Keep triggers **short**: they run for each row and slow down modifications.

Good practices for triggers (2/2)

- Beware of **cascades**: a trigger that modifies a table can fire others (or even itself).
- Several triggers on the same event run in the **alphabetical order** of their names.
- Management: `DROP TRIGGER name ON table;` `ALTER TABLE table DISABLE TRIGGER name;`
- On a **view**, `INSTEAD OF` makes it possible to update a view that is not updatable.

Exercise 7: which trigger?

Question: for each need, fill in the last two columns.

Need	`BEFORE / AFTER?`	`ROW / STATEMENT?`
1. Compute the amount of a sale before storing it		
2. Log every deleted account		
3. Reject a sale when the stock is insufficient		
4. Log only once an `UPDATE` that modifies a thousand rows		

Review: When does the trigger run? and The syntax of CREATE TRIGGER

Solution of exercise 7

Need	**Timing**	**Level**
1. Compute the amount before storing it	`BEFORE`	`ROW`
2. Log every deleted account	`AFTER`	`ROW`
3. Reject a sale when the stock is insufficient	`BEFORE`	`ROW`
4. Log only once an `UPDATE` of a thousand rows	`AFTER`	`STATEMENT`

1 and 3: only `BEFORE` can modify `NEW` or prevent the write. 4: `STATEMENT` runs once per statement (`BEFORE` would also do).

Exercise 8: how many rows in the log?

Data: the trigger `trg_audit_accounts` (`AFTER INSERT OR UPDATE OR DELETE ... FOR EACH ROW`) adds one row to the log each time it runs.

Question: how many rows does this `UPDATE` add to the log?

```
SELECT count(*) FROM account_log;     -- before

UPDATE accounts SET balance = balance + 10
WHERE  id IN (101, 102);

SELECT count(*) FROM account_log;     -- after
```

a) 0 **b)** 1 **c)** 2

Choose, then run the three queries and compare the two numbers. Review: FOR EACH ROW

Solution of exercise 8

Answer: c) 2 rows: the second number is 2 more than the first.

- The `UPDATE` modifies **two** accounts (101 and 102).
- `FOR EACH ROW`: the trigger runs **once per modified row**, so twice; it writes one row to the log each time.
- With `FOR EACH STATEMENT`, it would run only once.

Recall: triggers

Answer **from memory**, then click "Review" to check.

1. To modify `NEW` before it is written: `BEFORE` or `AFTER` trigger?
   Review: When does the trigger run?
2. What is `OLD` in a trigger fired by an `INSERT`?
   Review: NEW and OLD
3. In which order do several triggers on the same event run?
   Review: Good practices for triggers

What is a transaction?

- A **transaction** is a sequence of operations treated as **a whole**: everything is committed, or nothing.
- Example: a transfer = a debit **and** a credit. A debit without a credit would make money disappear.
- It guarantees the **ACID** properties:
  - **Atomicity**: all the operations succeed, or none is applied;
  - **Consistency**: the database goes from one valid state to another valid state;
  - **Isolation**: concurrent transactions do not see each other's uncommitted changes;
  - **Durability**: a committed change survives failures.
- Commands: `BEGIN` (start), `COMMIT` (commit), `ROLLBACK` (cancel), `SAVEPOINT` (restore point).

[Elmasri & Navathe (2016); PostgreSQL documentation]

Committing or rolling back (1/2)

```
BEGIN;                                      -- or START TRANSACTION
UPDATE accounts SET balance = balance - 500 WHERE id = 101;
UPDATE accounts SET balance = balance + 500 WHERE id = 102;
COMMIT;                                     -- both updates are committed

BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 101;
ROLLBACK;                                   -- cancelled: the balance is unchanged
```

- Without `BEGIN`, each statement is a transaction on its own (*autocommit*).
- Between `BEGIN` and `COMMIT`, the other sessions still see the old values.
- **Caution:** `BEGIN;` (SQL) starts a transaction; the `BEGIN` of a PL/pgSQL block does not.

An error invalidates the transaction (2/2)

```
BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 101;
SELECT 1 / 0;
UPDATE accounts SET balance = balance + 500 WHERE id = 102;
COMMIT;
```

Key points:

- After an error, the transaction is **invalidated**: all the following commands are rejected.
- The final `COMMIT` turns into a `ROLLBACK`: the debit of account 101 is cancelled.
- This is **atomicity**: it is impossible to commit half of a transfer.

A transfer in a function

```
CREATE OR REPLACE FUNCTION transfer_funds(
    p_from INT, p_to INT, p_amount NUMERIC)
RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE
    v_balance accounts.balance%TYPE;
BEGIN
    SELECT balance INTO v_balance FROM accounts
    WHERE id = p_from FOR UPDATE;       -- locks the debited account
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Account % does not exist', p_from;
    ELSIF v_balance < p_amount THEN
        RAISE EXCEPTION 'Insufficient balance (% DA)', v_balance;
    END IF;
    UPDATE accounts SET balance = balance - p_amount WHERE id = p_from;
    UPDATE accounts SET balance = balance + p_amount WHERE id = p_to;
    IF NOT FOUND THEN                   -- debit already done: it will be undone
        RAISE EXCEPTION 'Account % does not exist', p_to;
    END IF;
END;
$$;
SELECT transfer_funds(101, 102, 300);
```

Caller's transaction: an exception cancels **everything**, debit included.

Committing in a procedure: COMMIT

```
-- Purge the log in batches of p_batch rows
CREATE OR REPLACE PROCEDURE purge_log(
    p_before DATE, p_batch INT DEFAULT 1000)
LANGUAGE plpgsql
AS $$
DECLARE
    v_count INT;
BEGIN
    LOOP
        DELETE FROM account_log
        WHERE log_id IN (SELECT log_id FROM account_log
                         WHERE logged_at < p_before LIMIT p_batch);
        GET DIAGNOSTICS v_count = ROW_COUNT;
        EXIT WHEN v_count = 0;
        COMMIT;                              -- commits the batch, frees the locks
        RAISE NOTICE '% rows deleted', v_count;
    END LOOP;
END;
$$;
CALL purge_log('2027-01-01');                -- outside a BEGIN ... COMMIT
```

- Only a **procedure** called **outside an explicit transaction** can `COMMIT`/`ROLLBACK`.
- Benefit: commit as you go, without blocking the table.

Rolling back only part: SAVEPOINT (1/2)

In SQL: a savepoint makes it possible to roll back part of the transaction.

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 101;
SAVEPOINT before_bonus;
UPDATE accounts SET balance = balance + 1000 WHERE id = 102;
ROLLBACK TO SAVEPOINT before_bonus;     -- cancels only the bonus
COMMIT;                                 -- the debit of account 101 is committed
```

The implicit savepoint in PL/pgSQL (2/2)

In PL/pgSQL, you do not write `SAVEPOINT`: a block with an `EXCEPTION` section creates one automatically.

```
DO $$
BEGIN
    BEGIN                                   -- implicit savepoint
        INSERT INTO sales(product_id, quantity) VALUES (3, 100);
    EXCEPTION
        WHEN OTHERS THEN                    -- only this sale is cancelled
            RAISE NOTICE 'Sale rejected: %', SQLERRM;
    END;
    RAISE NOTICE 'Processing continues';
END;
$$;
```

Usage tips

- Group in **one** transaction the operations that must succeed together (debit and credit).
- Keep transactions **short**: the modified rows stay locked until the `COMMIT`.
- Lock before checking (`SELECT ... FOR UPDATE`) when the decision depends on a value read.
- In a function, report problems with `RAISE EXCEPTION`: the rollback is automatic.
- Catch an error (`EXCEPTION`) only to cancel **part** of the work and continue.
- Keep `COMMIT` in code for batch-processing **procedures**.

Exercise 9: by how much does the balance change?

Question: after this script, by how much has the balance of account 102 changed?

```
BEGIN;
UPDATE accounts SET balance = balance + 100 WHERE id = 102;
SAVEPOINT s1;
UPDATE accounts SET balance = balance + 999 WHERE id = 102;
ROLLBACK TO SAVEPOINT s1;
UPDATE accounts SET balance = balance - 100 WHERE id = 102;
COMMIT;
```

a) $+999$ **b)** $+100$ **c)** 0 **d)** -100

Choose, note the balance, run the script, then compare. Review: SAVEPOINT

Solution of exercise 9

Answer: c) 0, the final balance equals the initial balance.

- $+100$: kept (done **before** the savepoint `s1`).
- $+999$: undone by `ROLLBACK TO SAVEPOINT s1`.
- -100: kept.
- Net change committed by `COMMIT`: $+100 - 100 = 0$.

Recall: transactions

Answer **from memory**, then click "Review" to check.

1. Does the `BEGIN` of a PL/pgSQL block start a transaction?
   Review: Committing or rolling back
2. After an error in a transaction, what happens to the `COMMIT`?
   Review: An error invalidates the transaction
3. Why read the balance with `SELECT ... FOR UPDATE` before debiting?
   Review: Usage tips

Why cursors? (1/2)

- A **cursor** is a **pointer** to the result of a query, which reads it **row by row**.
- You have already used one without knowing it: `FOR r IN SELECT ...` `LOOP` creates an **implicit cursor**.
- An **explicit cursor** lets the programmer control each step:

  `DECLARE` → `OPEN` → `FETCH` (repeated) → `CLOSE`

- It is useful to:
  - **stop** reading at any time (without reading the whole result),
  - **reuse** the same query with different **parameters**,
  - **modify the current row** (`WHERE CURRENT OF`),
  - return a result to the application (`refcursor` type).
- Rows are read **as needed**: a large result does not fill the memory.

[PostgreSQL documentation, PL/pgSQL — Cursors]

The four steps of a cursor (2/2)

```
DECLARE
    cursor_name CURSOR FOR SELECT ...;      -- 1. binds the cursor to a query
    v_col1 TYPE;                            --    receiving variables
    v_col2 TYPE;
BEGIN
    OPEN cursor_name;                       -- 2. runs the query
    LOOP
        FETCH cursor_name INTO v_col1, v_col2;   -- 3. reads the next row
        EXIT WHEN NOT FOUND;                --    no more rows: leave
        -- process the row
    END LOOP;
    CLOSE cursor_name;                      -- 4. releases the resources
END;
```

- `OPEN` runs the query; the cursor is positioned **before** the first row.
- `FETCH ... INTO` moves forward one row and copies its columns into the variables.
- After the last row, `FETCH` sets `FOUND` to `FALSE`: `EXIT WHEN NOT FOUND` leaves the loop.
- `CLOSE` closes the cursor; it can then be opened again.

Reading sales and stopping at a threshold (1/2)

```
DO $$
DECLARE
    c_sales  CURSOR FOR SELECT id, amount FROM sales ORDER BY id;
    v_id     sales.id%TYPE;
    v_amount sales.amount%TYPE;
    v_total  NUMERIC := 0;
BEGIN
    OPEN c_sales;
    LOOP
        FETCH c_sales INTO v_id, v_amount;
        EXIT WHEN NOT FOUND;                    -- end of the result
        v_total := v_total + v_amount;
        RAISE NOTICE 'Sale %: % (running total %)', v_id, v_amount, v_total;
        EXIT WHEN v_total >= 7000;              -- early stop
    END LOOP;
    CLOSE c_sales;
END;
$$;
```

- Two possible exits: the **end of the result** or the **threshold** reached.

Parameterized cursor and WHERE CURRENT OF (2/2)

```
DO $$
DECLARE
    -- Parameterized cursor: accounts whose balance is below a threshold
    c_accounts CURSOR (p_threshold NUMERIC) FOR
        SELECT id, name, balance FROM accounts
        WHERE balance < p_threshold
        FOR UPDATE;                              -- locks the rows read
BEGIN
    FOR r IN c_accounts(500) LOOP                -- automatic OPEN, FETCH, CLOSE
        UPDATE accounts SET balance = balance + 50
        WHERE CURRENT OF c_accounts;             -- the current row of the cursor
        RAISE NOTICE '50 DA bonus for %', r.name;
    END LOOP;
END;
$$;
```

- The **parameter** (`p_threshold`) is supplied when opening: `c_accounts(500)`.
- `FOR r IN cursor` opens, reads and closes the cursor by itself; `r` is declared automatically.
- `WHERE CURRENT OF` modifies exactly the row the cursor has just read.

Exercise 10: which name?

Data: Keyboard (2500 DA), Mouse (800 DA), Monitor (18000 DA).

Question: which name does this block display?

```
DO $$
DECLARE
    c      CURSOR FOR SELECT name FROM products ORDER BY price DESC;
    v_name products.name%TYPE;
BEGIN
    OPEN c;
    FETCH c INTO v_name;
    FETCH c INTO v_name;
    CLOSE c;
    RAISE NOTICE '%', v_name;
END;
$$;
```

a) Monitor **b)** Keyboard **c)** Mouse

Choose, then run the block to check. Review: The four steps of a cursor

Solution of exercise 10

Answer: b) Keyboard.

- `ORDER BY price DESC`: Monitor (18000), Keyboard (2500), Mouse (800).
- The first `FETCH` reads Monitor; the second reads the next row, Keyboard, and replaces the value of `v_name`.

Exercise 11: true or false?

Question: is each statement about **cursors** true or false?

a) `OPEN` loads the whole query result into memory.

b) After the last row, `FETCH` sets `FOUND` to `FALSE`.

c) `FOR r IN cursor` opens and closes the cursor by itself.

d) `WHERE CURRENT OF` modifies the row the cursor has just read.

Review: Why cursors? and The four steps of a cursor

Solution of exercise 11

a) **False**: the cursor reads the rows **as they are needed**, which avoids loading a large result into memory.

b) **True**: `FOUND` is `FALSE` when `FETCH` reads no more rows.

c) **True**: the `FOR` loop opens the cursor, then closes it when it exits.

d) **True**: `UPDATE ... WHERE CURRENT OF c` modifies the row the cursor is positioned on.

Recall: cursors

Answer **from memory**, then click "Review" to check.

1. Why is a `SELECT ... INTO` not enough to go through a table?
   Review: Why cursors?
2. What are the four steps of a cursor?
   Review: The four steps of a cursor
3. How do you know that a `FETCH` found no more rows?
   Review: The four steps of a cursor

Building a query at run time: EXECUTE

Problem: the table name is only known at run time.

```
CREATE OR REPLACE FUNCTION count_rows(p_table TEXT)
RETURNS BIGINT
LANGUAGE plpgsql
AS $$
DECLARE
    v_count BIGINT;
BEGIN
    EXECUTE format('SELECT count(*) FROM %I', p_table)   -- %I: identifier
    INTO v_count;
    RETURN v_count;
END;
$$;

SELECT count_rows('accounts');
```

- `EXECUTE text [INTO variables] [USING values]` runs a query built as text.
- `format()`: `%I` inserts an **identifier**, `%L` a **value**; `USING` passes `$1`, `$2`...

Avoiding SQL injection

```
-- DANGEROUS: the input value is pasted into the query text
EXECUTE 'SELECT balance FROM accounts WHERE name = ''' || p_name || ''''
INTO v_balance;
-- with p_name = x' OR '1'='1, the condition becomes always true

-- SAFE: the value is passed as a parameter
EXECUTE 'SELECT balance FROM accounts WHERE name = $1'
INTO v_balance
USING p_name;

-- SAFE: an identifier is protected by format(%I)
EXECUTE format('SELECT count(*) FROM %I', p_table) INTO v_count;
```

- **SQL injection**: a malicious value changes the meaning of the query (reading or destroying data).
- **Rule**: never concatenate a value supplied by the user; `USING` for values, `%I` for identifiers.
- Static PL/pgSQL queries (without `EXECUTE`) are protected: variables in them are always values.

[OWASP Top 10 — Injection; PostgreSQL documentation]

Exercise 12: which queries are safe?

Data: `p_name` contains text typed by the user.

Question: which statements are safe from SQL injection? (several answers possible)

```
-- a)
EXECUTE 'SELECT * FROM accounts WHERE name = ''' || p_name || '''';
-- b)
EXECUTE 'SELECT * FROM accounts WHERE name = $1' USING p_name;
-- c)
EXECUTE format('SELECT * FROM accounts WHERE name = %L', p_name);
-- d)
SELECT * FROM accounts WHERE name = p_name;
```

Review: Avoiding SQL injection and EXECUTE

Solution of exercise 12

Answer: b), c) and d).

a) **Unsafe**: the input is pasted into the query text; `x' OR '1'='1` returns every account.

b) Safe: `USING` passes `p_name` as a **value**, never as code.

c) Safe: `%L` quotes the value and doubles the quotes it contains.

d) Safe: static query, PL/pgSQL passes `p_name` as a parameter.

Summary and outlook

- Procedural SQL complements declarative SQL: it is used when you must **chain, decide, repeat or react to errors**, **on the server side**.
- Progression of the chapter: **a complete example (Lab 2)** → **block and variables** → **conditions and loops** → **functions and procedures** → **exceptions** → **triggers** → **transactions** → **cursors** → **dynamic SQL**.
- Functions, procedures and triggers put the **business rules** in the database: they apply to all applications.
- Transactions guarantee that a business operation is applied **entirely or not at all**.
- Good practices: prefer a query or a constraint when it is enough, report errors with clear messages, never concatenate user input into dynamic SQL.

Outlook: automation, predictive analytics, API integration and development of complex server-side applications.

[Silberschatz et al., 2019; ISO/IEC 9075-4:2016; PostgreSQL documentation]