Chapter 2 — Advanced SQL

Procedural SQL programming

M. Laïdi FOUGHALI

l.foughali@univ-skikda.dz

(Website ⇒ www.al-moualime.com)

University of Skikda — Department of Computer Science

1st Year Master RSD/AI

Advanced Databases (BDDA)

Updated on 2026-09-22 at 22:26:32

Outline

General objective

- The **standard SQL language** (defined in **SQL-86, ISO/IEC 9075:1986**) is **declarative**: it specifies the expected result without imposing the processing method.
- Modern applications required complex processing (sequences of statements, logical conditions, loops, error handling, etc.), which declarative SQL alone could not handle.
- The **SQL/PSM — Persistent Stored Modules** standard (**ISO/IEC 9075-4:1996**) introduces these mechanisms and **procedural SQL programming**.
- It enables the DBMS to run internal programs that guarantee **transactional consistency**, **security** and **performance**.
- It improves **portability** and reduces **client-server traffic**.

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

Origin of procedural SQL

- The **SQL3 (1992)** project formalized the **SQL/PSM — Persistent Stored Modules** extension, defined in the standard (**ISO/IEC 9075-4:1996**), which **introduces**:
  - `BEGIN ... END` blocks,
  - **local variables**,
  - **control-flow** structures (`IF`, `WHILE`, `LOOP`),
  - **exception handling** (`HANDLER`).
- This standard inspired several procedural languages, **notably**:
  - **PL/pgSQL** (Procedural Language / PostgreSQL SQL — PostgreSQL, 1996),
  - **MySQL SQL/PSM** (partial implementation of the standard, MySQL, 2005).
- **Goal**: allow SQL to run logical programs on the server side, combining declarative power with procedural control.

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

Philosophy of procedural SQL

- **Principle**: bring **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** without external dependencies.
- **Duality preserved**:
  - **declarative** SQL: describes the *desired result*,
  - **procedural** SQL: defines the *logical sequence to obtain it*.
- SQL thus remains **expressive**, **structured** and **operational**, faithful to the relational model and suited to modern needs.

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

Course positioning

- **Course scope**: all **practical examples** use **PostgreSQL**, including procedures, functions, triggers and PL/pgSQL.
- **Theoretical reference**: based on the **SQL standards** (**ISO/IEC 9075**, including **SQL/PSM**); PostgreSQL-specific extensions are clearly flagged.
- **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 widely 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.
- **Pedagogical value**: provides a **standard** foundation and a technology **used in production**, linking theory and practice.

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

New running example: account management

Declarative SQL relied on `Employees`/`Departments`. The procedural part uses a **new** set of tables, better suited to transaction and trigger scenarios (account movements, sales, stock).

```sql
CREATE TABLE accounts (
    id      INT PRIMARY KEY,
    balance NUMERIC NOT NULL
);

CREATE TABLE products (
    id    INT PRIMARY KEY,
    stock INT NOT NULL
);

CREATE TABLE sales (
    id     INT PRIMARY KEY,
    amount NUMERIC NOT NULL
);

-- History fed by the triggers of this chapter
CREATE TABLE accounts_log (
    id           INT,
    old_balance  NUMERIC,
    new_balance  NUMERIC,
    created_at   TIMESTAMP,
    updated_at   TIMESTAMP,
    deleted_at   TIMESTAMP
```

Declarative SQL vs procedural SQL

Declarative SQL: expresses **what we want to obtain**; the **DBMS** decides how to execute it.

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

Procedural SQL: describes **the processing logic step by step** and runs **on the server side**.

```
DO $calc$
DECLARE
    total NUMERIC(15,2) := 0;
BEGIN
    FOR r IN SELECT amount FROM sales
    LOOP
      total := total + r.amount;
    END LOOP;
    RAISE NOTICE 'Total sales = %', total;
END;
$calc$;
```

Main objectives:

- Bring business logic directly to the data;
- Reduce round trips between client and server;
- Enable complex processing (conditions, loops, exceptions);

Procedural block

A **procedural block** is an executable unit grouping several SQL and logical statements. The syntax is similar for all blocks, but the behavior differs depending on whether it is **named** or **anonymous**:

```
[DECLARE
    variable_1 type [DEFAULT value];
    variable_2 type;
    ...
]
BEGIN
    -- Sequential SQL statements
    -- Flow control: IF, CASE, LOOP, WHILE, FOR
EXCEPTION
    -- Error handling (CONTINUE / EXIT HANDLER)
END;
```

Distinction and typical uses:

- **Anonymous block**: not stored, executed immediately; ideal for testing scripts or one-off processing.
- **Named block** (function or procedure): stored and reusable by name; allows transactions to be managed and business logic to be encapsulated.

[ISO/IEC 9075-4:2016 — Persistent Stored Modules]

Functions, Procedures and Triggers

Construct	Return, Call	Description and use
Function	RETURN, SELECT	Returns a simple value, a record or a set of rows; can be used directly in SQL queries. Cannot perform COMMIT/ROLLBACK. **Ideal for reusable computations and transformations**.
Procedure	OUT, CALL	Runs on the server side for several statements. Can manage COMMIT and ROLLBACK and have effects on the data. **Encapsulates business logic**.
Trigger	Automatic	Fired on INSERT, UPDATE or DELETE. **Validates, enriches or logs data automatically**. Cannot be invoked directly.

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

Declaring and assigning variables

```
DECLARE
    v_balance DECIMAL(15,2) DEFAULT 0;
    v_rate    DECIMAL(5,2) := 0.15;
BEGIN
    -- Simple assignments
    v_balance := v_balance + 100;
    v_balance := v_balance * (1 + v_rate);

    -- Assignment from an SQL query
    SELECT balance INTO v_balance
    FROM accounts
    WHERE id = 101;
END;
```

Assignment principles:

- ‘DEFAULT‘ or ‘:=‘ to initialize a variable in ‘DECLARE‘.
- ‘:=‘ to modify a variable within the ‘BEGIN . . . END‘ block.
- ‘SELECT . . . INTO‘ to assign the result of an SQL query to a variable.

[ISO/IEC 9075-4:2016 — Persistent Stored Modules]

How to use comments

```
-- Single-line comment
v_balance := v_balance + 100;  -- increment the balance
```

```
/*
  Multi-line comment
  Explains several statements
*/
v_rate := 0.15;
```

Good practices:

- Explain the **why**, not the **what**.
- Place comments before important blocks.
- Use inline comments for specific statements.

The RAISE statement

```
DECLARE
    v_balance DECIMAL(15,2) := 120;  -- initialization
BEGIN
    RAISE NOTICE 'Current balance: %', v_balance;
    v_balance := v_balance - 50;
    RAISE NOTICE 'New balance after withdrawal: %', v_balance;
END;
```

Explanations:

- 'RAISE NOTICE': sends a message to the user or to the server log.
- Useful for debugging and tracing execution.
- Variables can be inserted with the '
- Execution continues after the message is displayed.

To report an error and stop execution: use 'RAISE EXCEPTION', presented later.

The FOUND variable

FOUND is an implicit Boolean variable in PL/pgSQL. It is ‘TRUE‘ if the last SQL statement affected or returned at least one row, and ‘FALSE‘ otherwise.

It is mainly used after: ‘SELECT INTO‘, ‘UPDATE‘, ‘DELETE‘, or ‘INSERT ... RETURNING‘.

Example: SELECT INTO

```
DO $$
DECLARE
    v_balance NUMERIC(15,2);  -- correct declaration with precision and scale
BEGIN
    SELECT balance INTO v_balance FROM accounts WHERE id = 999;
    IF NOT FOUND THEN
        RAISE NOTICE 'Account not found';
    END IF;
END;
$$;
```

Key points:

- Checks whether a statement produced a result.
- Useful for handling missing data or triggering conditional actions.

What is a function?

- Encapsulates reusable, modular code.
- Returns a value with `RETURN`.
- Contains local variables and `BEGIN...END` blocks.
- Called via `SELECT function_name(...);` or from other functions.
- PostgreSQL also supports other procedural languages: PL/Python, PL/JavaScript (PL/v8), etc.

AI and PostgreSQL with PL/Python

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

- Run AI and machine-learning models on the server side.
- Process large volumes of data without round trips to the application.
- Create advanced functions for predictive analytics or statistics directly in the database.
- Combine the robustness of PostgreSQL with the power of Python for AI.

General structure of a function

```
-- General syntax
CREATE OR REPLACE FUNCTION function_name(param1 TYPE, ...)
RETURNS RETURN_TYPE AS $$
DECLARE
    v_var TYPE;  -- local variables
BEGIN
    -- function body
    RETURN value;
END;
$$ LANGUAGE plpgsql;

-- Call
-- Scalar or VOID: SELECT function_name(...);
-- Row or Table:   SELECT * FROM function_name(...);

-- Possible return types:
-- 1. A scalar type (INTEGER, TEXT, DECIMAL, DATE, etc.)
-- 2. TABLE(col1 TYPE, col2 TYPE, ...): to return a set of rows
-- 3. VOID: if the function returns nothing
-- 4. TRIGGER: if the function is used as a trigger
```

Returning a scalar value

```
-- Returning a single (scalar) value
CREATE OR REPLACE FUNCTION get_balance(p_id INT)
RETURNS NUMERIC(15,2) AS $$
DECLARE
    v_balance NUMERIC(15,2);
BEGIN
    SELECT balance INTO v_balance
    FROM accounts
    WHERE id = p_id;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Account does not exist';
    END IF;
    RETURN v_balance;
END;
$$ LANGUAGE plpgsql;

-- Call
SELECT get_balance(101);
```

Returning a row

```sql
-- %ROWTYPE: special type representing a complete row of a table
CREATE OR REPLACE FUNCTION get_account(p_id INT)
RETURNS accounts AS $$
DECLARE
    v_account accounts%ROWTYPE;  -- holds all the fields of the accounts table
BEGIN
    -- Fetch the row
    SELECT * INTO v_account
    FROM accounts
    WHERE id = p_id;

    -- Handle the case where the account does not exist
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Account does not exist';
    END IF;

    -- Example of accessing individual columns
    RAISE NOTICE 'Name: %, Balance: %', v_account.name, v_account.balance;

    RETURN v_account;
END;
$$ LANGUAGE plpgsql;

-- Calling the function
SELECT * FROM get_account(101);
```

Returning a table

```
-- Using RETURN QUERY to return several rows
CREATE OR REPLACE FUNCTION total_sales()
RETURNS TABLE(product TEXT, total NUMERIC(15,2)) AS $$
BEGIN
    RETURN QUERY
    SELECT product, quantity * price AS total
    FROM sales;
END;
$$ LANGUAGE plpgsql;

-- Call
SELECT * FROM total_sales();
```

- 'RETURN QUERY' executes the query and returns all the rows.
- Retrieves a **set of rows** as from a table.
- The result can be filtered or sorted with 'WHERE' or 'ORDER BY'.

Function without a return value (VOID)

```sql
-- Function without a return value (VOID)
CREATE OR REPLACE FUNCTION update_stock(p_id INT, p_qty INT)
RETURNS VOID AS $$
BEGIN
    UPDATE products
    SET stock = stock + p_qty
    WHERE id = p_id;
END;
$$ LANGUAGE plpgsql;

-- Call
SELECT update_stock(5, 10);
```

Trigger function

```sql
-- Function used as a trigger
CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at := now();   -- update the updated_at column
    RETURN NEW;                -- mandatory for a BEFORE/AFTER trigger
END;
$$ LANGUAGE plpgsql;

-- Creating the trigger
CREATE TRIGGER trg_set_updated
BEFORE INSERT OR UPDATE ON accounts
FOR EACH ROW
EXECUTE FUNCTION set_updated_at();
```

Teaching note

TRIGGER functions are used to run code automatically before or after an operation on a table.
Their behavior is detailed in a later section.

What is a procedure?

- A procedure runs reusable code without necessarily returning a value.
- Called via: `CALL procedure_name(...);`
- INOUT parameters make it possible to retrieve or modify values after execution.
- Useful for sequential or complex processing that does not require a direct return value.

General structure of a procedure

```
-- General declaration
CREATE PROCEDURE procedure_name(
    IN param1 TYPE,       -- input
    INOUT param2 TYPE     -- input and output
)
LANGUAGE plpgsql AS $$
BEGIN
    -- procedure body
END;
$$;

-- Call
CALL procedure_name(val1, val2);
```

- IN: input parameter
- INOUT: parameter that can be modified and retrieved after execution

Adjusting an account balance

```sql
-- Procedure to adjust the balance
CREATE PROCEDURE p_adjust_balance(
    IN p_id INT,
    IN p_amount NUMERIC(15,2),
    INOUT p_balance NUMERIC(15,2)
)
LANGUAGE plpgsql AS $$
BEGIN
    -- Update the balance
    UPDATE accounts SET balance = balance + p_amount
    WHERE id = p_id;

    -- Fetch the new balance
    SELECT balance INTO p_balance
    FROM accounts
    WHERE id = p_id;
END;
$$;

-- Execution
DO $$
DECLARE v_balance NUMERIC(15,2);
BEGIN
    CALL p_adjust_balance(101, 200.00, v_balance);
    RAISE NOTICE 'Final balance: %', v_balance;
END $$;
```

Triggers: automatic execution

- A trigger is a mechanism that automatically runs a function when an operation occurs on a table: INSERT, UPDATE or DELETE.
- A trigger makes it possible to:
  - log changes,
  - validate complex constraints,
  - compute or update other columns automatically.
- Triggers are associated with TRIGGER functions.
- Three main types:
  - BEFORE: before the operation
  - AFTER: after the operation
  - INSTEAD OF: for views

Using NEW and OLD

- In a trigger function, the row values are accessed through two complete records:
  - `NEW`: the row after modification (INSERT or UPDATE)
  - `OLD`: the row before modification (UPDATE or DELETE)
- Each column of the row is accessed with dot notation:

```
RAISE NOTICE 'Old balance: %, New balance: %', OLD.balance, NEW.balance;
```

- In BEFORE triggers, columns can be modified via `NEW.column`.
- Makes it possible to compare or modify values before or after the operation.

Creating a simple trigger

```
-- Trigger function
CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at := now();  -- update a column before the operation
    RETURN NEW;               -- mandatory for BEFORE/AFTER
END;
$$ LANGUAGE plpgsql;

-- Creating the trigger
CREATE TRIGGER trg_set_updated
BEFORE INSERT OR UPDATE ON accounts
FOR EACH ROW
EXECUTE FUNCTION set_updated_at();
```

- BEFORE: runs the function before the operation, allows the row to be modified.
- AFTER: runs after the operation, useful for logging.
- FOR EACH ROW: fires for each modified row.

UPDATE example

```
CREATE OR REPLACE FUNCTION log_update_balance()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO accounts_log(id, old_balance, new_balance, updated_at)
    VALUES (OLD.id, OLD.balance, NEW.balance, now());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_log_update
AFTER UPDATE ON accounts
FOR EACH ROW
EXECUTE FUNCTION log_update_balance();
```

- OLD.balance → value before the modification
- NEW.balance → value after the modification
- Keeps a history of the changes

DELETE example

```
CREATE OR REPLACE FUNCTION log_delete_account()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO accounts_log(id, old_balance, deleted_at)
    VALUES (OLD.id, OLD.balance, now());
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

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

- OLD contains all the columns of the deleted row
- RETURN OLD needed for AFTER DELETE
- Keeps a history of deletions

INSERT example

```
CREATE OR REPLACE FUNCTION log_insert_account()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO accounts_log(id, new_balance, created_at)
    VALUES (NEW.id, NEW.balance, now());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

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

- NEW contains all the columns of the inserted row
- RETURN NEW mandatory for BEFORE/AFTER INSERT
- Tracks all newly created rows

Trigger good practices

- Always specify the type: BEFORE or AFTER as needed.
- Use FOR EACH ROW to handle each row individually.
- NEW and OLD are records representing the whole row.
- RETURN NEW or OLD is mandatory depending on the type of operation.
- Avoid overly heavy logic in triggers so as not to slow down transactions.
- For views: INSTEAD OF intercepts INSERT/UPDATE/DELETE operations.

IF / ELSIF / ELSE and CASE

```
DECLARE v_balance DECIMAL(15,2) := 120;
BEGIN
    IF v_balance < 0 THEN
        RAISE NOTICE 'Negative balance: %', v_balance;
    ELSIF v_balance < 100 THEN
        RAISE NOTICE 'Low balance: %', v_balance;
    ELSE
        RAISE NOTICE 'Sufficient balance: %', v_balance;
    END IF;
END;
```

```
CASE
    WHEN v_balance < 0 THEN RAISE NOTICE 'Overdrawn';
    WHEN v_balance < 100 THEN RAISE NOTICE 'Low';
    ELSE RAISE NOTICE 'Sufficient';
END CASE;
```

Explanations:

- 'IF ... ELSIF ... ELSE ... END IF': classic conditional control.
- 'CASE ... END CASE': compact alternative for several conditions.

LOOP with EXIT

```
DECLARE counter INT := 0;
BEGIN
    LOOP
        counter := counter + 1;
        EXIT WHEN counter > 3;
        RAISE NOTICE 'Counter: %', counter;
    END LOOP;
END;
```

Explanations:

- ‘LOOP‘: infinite loop until ‘EXIT‘.
- ‘EXIT WHEN condition‘: exits when the condition is true.

WHILE loop

```
DECLARE counter INT := 0;
BEGIN
    WHILE counter < 3 LOOP
        counter := counter + 1;
        RAISE NOTICE 'WHILE counter: %', counter;
    END LOOP;
END;
```

Explanations:

- 'WHILE condition LOOP': repeats as long as the condition is true.

FOR loop over a range

```
FOR i IN 1..3 LOOP
    RAISE NOTICE 'FOR i: %', i;
END LOOP;
```

Explanations:

- 'FOR i IN start..end LOOP': i takes each integer in turn.
- The loop is bounded automatically, no need for 'EXIT'.

FOR loop over a SELECT

```
FOR rec IN SELECT amount FROM sales LOOP
    RAISE NOTICE 'Sale amount: %', rec.amount;
END LOOP;
```

Explanations:

- ‘rec‘ represents the current row.
- Each column is accessible via ‘rec.column_name‘.
- The loop stops after the last row.

CONTINUE example

```
FOR i IN 1..5 LOOP
    CONTINUE WHEN i % 2 = 0;
    RAISE NOTICE 'Odd: %', i;
END LOOP;
```

Explanations:

- 'CONTINUE WHEN condition': skips the current iteration if the condition is true.
- Useful to ignore certain values without leaving the loop.

No GOTO in PL/pgSQL

```
BEGIN
    RAISE NOTICE 'Start';

    <<blk>>
    BEGIN
        RAISE NOTICE 'Skipped if condition is true';
        EXIT blk WHEN TRUE;
        RAISE NOTICE 'Never reached';
    END;

    RAISE NOTICE 'End';
END;
```

Explanations:

- Unlike Oracle PL/SQL, **PL/pgSQL has no `GOTO` statement**.
- Labels «`label`» only serve to name a block or a loop, so that it can be targeted with `EXIT`/`CONTINUE`.
- `EXIT label WHEN condition`: leaves the named block/loop if the condition is true — it is the only jump mechanism available.

Returning values from a function

Objective: return a value, a row or a set of rows from a PostgreSQL function.

```
\small
  -- Returns a single value and ends the function
  RETURN v_balance;

  -- Adds a row in a set-returning function
  RETURN NEXT rec;

  -- Runs a query and returns all its rows
  RETURN QUERY SELECT * FROM sales;
```

Explanations:

- 'RETURN': returns a single value and immediately ends the function.
- 'RETURN NEXT': adds a row to the result set (for set-returning functions).
- 'RETURN QUERY': runs the query and returns all the rows directly to the caller.

[1/3] Why use cursors?

- Cursors make it possible to **iterate row by row** over the results of a query, which a simple 'SELECT' cannot do when individual processing is needed.
- They are essential to **process rows one at a time** when each row requires conditional or complex logic.
- They make it possible to **handle large volumes of data efficiently**, without overloading memory, since not all rows are loaded at once.
- Useful in PL/pgSQL functions and procedures for robust sequential processing, for example:
  - Row-by-row financial computations.
  - Sending e-mails or notifications for each record.
  - Conditional updates of large tables.

[2/3] Using an explicit cursor

```
-- General structure of an explicit cursor

DECLARE
    cursor_name CURSOR FOR SELECT ...;  -- defines the cursor on a query
    var1 TYPE;                          -- variables to store the columns
BEGIN
    OPEN cursor_name;                   -- opens the cursor
    LOOP
        FETCH cursor_name INTO var1, var2, ...;  -- reads the next row

        EXIT WHEN NOT FOUND; -- leaves the loop when there are no more rows

        -- processing of each row
    END LOOP;
    CLOSE cursor_name;                  -- closes the cursor
END;
```

- 'OPEN': initializes the cursor.
- 'FETCH ... INTO': retrieves the current row.
- 'EXIT WHEN NOT FOUND': ends only the loop, not the procedure.
- 'CLOSE': releases the cursor's resources.

[3/3] Listing sales with a cursor

```
-- Example: iterate over sales and display each row
DECLARE
    c_sales CURSOR FOR SELECT id, amount FROM sales;
    v_id INT;
    v_amount NUMERIC(15,2);
BEGIN
    OPEN c_sales;
    LOOP
        FETCH c_sales INTO v_id, v_amount;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE 'Sale %: %', v_id, v_amount;
    END LOOP;
    CLOSE c_sales;
END;
```

- ‘OPEN‘: opens the cursor to start reading.
- ‘FETCH ... INTO‘: reads a row and stores the values in the variables.
- ‘EXIT WHEN NOT FOUND‘: ends the loop when all rows have been read.
- ‘CLOSE‘: closes the cursor to release resources.

Catching and handling errors

Objective: intercept and handle errors without interrupting the execution of the block.

```
DECLARE
    msg TEXT;
    divisor INT := 0;
BEGIN
    UPDATE accounts SET balance = balance / divisor WHERE id = 101;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Error: division by zero';
    WHEN others THEN
        GET STACKED DIAGNOSTICS msg = MESSAGE_TEXT;
        RAISE NOTICE 'Unknown error: %', msg;
END;
```

Explanations:

- Here, `divisor = 0` really causes a division by zero: it is indeed the `WHEN division_by_zero` branch that runs (not `WHEN others`).
- 'WHEN division_by_zero': catches a division by zero.
- 'WHEN others': catches all other non-specific errors.
- 'GET STACKED DIAGNOSTICS msg = MESSAGE_TEXT': retrieves the detailed message.

Messages and exceptions

Objective: inform the user or report errors during execution.

```
-- Informational message
RAISE NOTICE 'Balance: %', v_balance;

-- Stops execution and reports a critical error
RAISE EXCEPTION 'Critical error: %', error_code;
```

Explanations:

- 'RAISE NOTICE': informational message, execution continues.
- 'RAISE EXCEPTION': aborts the block or the function (can be caught).
- Useful for logging, tracing and flow control.

Advanced example

Objective: combine exception catching, RAISE and retrieval of detailed information.

```
DECLARE msg TEXT;
BEGIN
    INSERT INTO accounts(id, balance) VALUES(101, 1000);
EXCEPTION
    WHEN unique_violation THEN
        RAISE NOTICE 'Error: the ID already exists';
    WHEN others THEN
        GET STACKED DIAGNOSTICS msg = MESSAGE_TEXT;
        RAISE EXCEPTION 'Critical error: %', msg;
END;
```

Explanations:

- ‘unique_violation‘: catches a uniqueness violation.
- ‘RAISE NOTICE‘: for minor errors.
- ‘RAISE EXCEPTION‘: for critical errors.
- ‘GET STACKED DIAGNOSTICS‘: provides full details of the error.

SQLERRM and SQLSTATE

Objective: quickly obtain the error message and SQL code.

```
BEGIN
    UPDATE accounts SET balance = balance / 0;  -- raises an error
EXCEPTION
    WHEN others THEN
        RAISE NOTICE 'Message: %', SQLERRM;
        RAISE NOTICE 'SQLSTATE code: %', SQLSTATE;
END;
```

Explanations:

- ‘SQLERRM‘: message associated with the exception.
- ‘SQLSTATE‘: standard SQL code of the error.
- These variables are only available inside an ‘EXCEPTION‘ block.

Detailed information about the exception

Objective: obtain all the available information about the exception.

```
DECLARE
    msg TEXT;
    detail TEXT;
    hint TEXT;
BEGIN
    INSERT INTO accounts(id, balance) VALUES (101, 1000);  -- may raise an error
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS
            msg = MESSAGE_TEXT,
            detail = PG_EXCEPTION_DETAIL,
            hint = PG_EXCEPTION_HINT;
        RAISE NOTICE 'Error: %', msg;
        RAISE NOTICE 'Detail: %', detail;
        RAISE NOTICE 'Hint: %', hint;
END;
```

Explanations:

- ‘MESSAGE_TEXT‘: main error message.
- ‘PG_EXCEPTION_DETAIL‘: detailed text provided by PostgreSQL.
- ‘PG_EXCEPTION_HINT‘: additional suggestions or hints.

All methods combined

```
DECLARE
    msg TEXT;
    code TEXT;
    detail TEXT;
    hint TEXT;
    context TEXT;
BEGIN
    UPDATE accounts SET balance = balance / 0;  -- raises an error
EXCEPTION
    WHEN others THEN
        -- Special variables
        msg := SQLERRM;
        code := SQLSTATE;

        -- GET STACKED DIAGNOSTICS
        GET STACKED DIAGNOSTICS
            detail = PG_EXCEPTION_DETAIL,
            hint = PG_EXCEPTION_HINT,
            context = PG_EXCEPTION_CONTEXT;

        RAISE NOTICE 'Message: %', msg;
        RAISE NOTICE 'SQLSTATE code: %', code;
        RAISE NOTICE 'Detail: %', detail;
        RAISE NOTICE 'Hint: %', hint;
        RAISE NOTICE 'Context: %', context;
END;
```

Tips for exception handling

Good practices:

- Always catch specific exceptions before ‘WHEN others‘.
- Use ‘RAISE NOTICE‘ for informational messages and ‘RAISE EXCEPTION‘ for critical errors.
- Limit the scope of ‘EXCEPTION‘ blocks to avoid hiding unexpected errors.
- Add explicit, actionable messages to the execution logs.
- Use ‘GET STACKED DIAGNOSTICS‘ to enrich debugging information.
- Avoid ‘GOTO‘ for error handling; prefer structured blocks.
- Do not overuse ‘EXCEPTION‘ blocks: they have a significant execution cost.

Transaction management

- A transaction is a set of SQL operations executed as a unit.
- It guarantees the ACID properties:
  - Atomicity: all operations succeed or none is applied
  - Consistency: the database remains in a valid state
  - Isolation: changes are only visible after commit
  - Durability: committed changes persist despite failures
- Main commands:
  - COMMIT: commits the changes
  - ROLLBACK: cancels all the changes made

Money transfer

```sql
START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE id = 101;
UPDATE accounts SET balance = balance + 500 WHERE id = 102;

-- Commit the transaction
COMMIT;

-- Roll back if an error is detected
-- ROLLBACK;
```

Transactional block in a function

```
CREATE OR REPLACE FUNCTION transfer_funds(
    debtor_id INT,
    creditor_id INT,
    amount NUMERIC(15,2)
)
RETURNS VOID AS $$
BEGIN
    UPDATE accounts SET balance = balance - amount WHERE id = debtor_id;
    UPDATE accounts SET balance = balance + amount WHERE id = creditor_id;

    IF (SELECT balance FROM accounts WHERE id = debtor_id) < 0 THEN
        RAISE EXCEPTION 'Insufficient balance';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

- PL/pgSQL functions run inside an implicit transaction.
- Any error or exception automatically rolls back the current transaction.
- The final commit or rollback of the transaction is handled by the calling client.

Using BEGIN, COMMIT, ROLLBACK

```
-- Start of the explicit transaction
BEGIN;

DO $$
BEGIN
  -- Call a function that performs a transfer without retrieving a value
  PERFORM transfer_funds(101, 102, 500);
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Error detected: %', SQLERRM;
        RAISE;  -- re-raises the exception: a DO block cannot run
                -- ROLLBACK itself when nested inside an already open
                -- transaction; it must let the error invalidate the transaction
END;
$$;

-- Commit the transaction if everything is correct
COMMIT;
```

- BEGIN starts the explicit transaction.
- PERFORM calls the PL/pgSQL function.
- A `DO` block nested inside an explicit transaction **cannot** call COMMIT/ROLLBACK itself; a re-raised exception (`RAISE`) automatically invalidates the current transaction.

Usage tips

- Always handle errors with an EXCEPTION block to manage rollbacks.
- Keep transactions short to avoid blocking (locks).
- Check constraints before modifying data to reduce exceptions.
- Do not put overly complex logic in a main transaction.

Summary and outlook

- Procedural SQL complements declarative SQL for complex **server-side** processing, with flow control and exception handling.
- PostgreSQL with PL/pgSQL and extensions (PL/Python, PL/v8...) makes it possible to build efficient and secure functions, procedures and triggers.
- Key concepts: **variables, loops, conditions, cursors, exceptions, functions and triggers** bring business logic closer to the data.
- Integrating Python enables **advanced analytics and AI** directly inside the database.
- Good practices: structured code, targeted exceptions, relevant comments, optimized loops and cursors.

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

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