Lecture 11. Stored Procedures & PL/pgSQL

What is a Stored Procedure?

> A **stored procedure** is a **named block of SQL + procedural logic** stored in the database and executed on the server.

It can contain:

- Multiple SQL statements
- Variables
- Loops, conditions
- Error handling

Main goals: **encapsulate logic**, **reuse code**, **improve performance**, **centralize business rules**.

Advantages of Stored Procedures

- **Fewer round trips**
  - Instead of sending many separate SQL statements from the application to the DB, the app calls **one procedure**.
  - DB executes all internal SQL on the server side.
- **Better performance**
  - User-defined procedures are **precompiled / cached** on the server.
  - Execution plan can be reused → less overhead.
- **Reuse across applications**
  - Once written, the procedure can be called from **any app** that connects to that database

---

Disadvantages of Stored Procedures

  - **Slower development**:
    - Requires DB-specific skills (PL/pgSQL, SQL inside DB).
  - **Harder versioning & debugging**:
    - Logic sits inside database, not in normal application codebase.
  - **Portability issues**:
    - Procedures are DB-specific (PostgreSQL vs MySQL vs SQL Server).
    - Migration to another DBMS may require rewriting.

Procedural Languages in PostgreSQL

PostgreSQL supports several languages for writing stored procedures / functions:

- Built-in:
  - **SQL**
  - **PL/pgSQL**
  - **C**
- Via extensions:
  - **PL/Perl**, **PL/Python**, **PL/JavaScript**, **PL/Ruby**, **PL/Java**, **PL/Tcl**, etc.

In this lecture, focus is on **PL/pgSQL**.

PL/pgSQL Overview

PL/pgSQL (Procedural Language / PostgreSQL) is PostgreSQL's main procedural language.

It allows:

- Writing **functions**, **procedures**, **triggers**

- Using control-flow logic:
    - `IF`, `CASE`, `LOOP`, `WHILE`, `FOR`
- Using **variables, parameters, error handling**
- Executing **dynamic SQL** at runtime
- Deep integration with SQL (easy to call queries inside code)

Key Features:

- **Control Structures**: `IF`, `CASE`, `LOOP`, `WHILE`, `FOR`
- **Error Handling**: `EXCEPTION` blocks
- **Dynamic SQL**: `EXECUTE`
- **Variables and Parameters**: `DECLARE` block
- **Integration**: runs inside PostgreSQL server, close to the data.

Basic PL/pgSQL Block Structure

Generic shape:

```
CREATE OR REPLACE PROCEDURE proc_name(arg_list)
LANGUAGE plpgsql
AS $$
DECLARE
    -- variable declarations
    total_sales numeric := 0;
BEGIN
    -- main code
    -- SQL statements, IF, loops, etc.

    -- optional error handling
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Error: division by zero';
```

```
END;
$$;
```

Important elements:

- `LANGUAGE plpgsql` – tells PostgreSQL to use PL/pgSQL.
- `$$ ... $$` – delimiters for the procedure body.
- `DECLARE` – section where you declare variables (optional).
- `BEGIN ... END` – main block.
- `EXCEPTION ...` – handles runtime errors (optional).

Variables (DECLARE Section)

In the `DECLARE` block you define local variables:

```
DECLARE
    counter      integer := 0;
    customer_id  integer;
    total_amount numeric;
```

- You can set default values with `:=`.
- Variables can be used in SQL statements or expressions inside the procedure.

Control Structure (Logic):

PL/pgSQL supports classic control-flow:

- `IF ... THEN ... ELSE ... END IF;`
- `CASE ... WHEN ... THEN ... END CASE;`
- Loops:
    - `LOOP ... END LOOP;`

    - `WHILE condition LOOP ... END LOOP;`
    - `FOR var IN 1..10 LOOP ... END LOOP;`

Used to implement complex business rules directly in the DB.

Error Handling with EXCEPTION

`EXCEPTION` blocks allow you to **catch and handle errors**:

```
BEGIN
   -- risky operations
   UPDATE accounts
   SET balance = balance - 100
   WHERE id = 1;

EXCEPTION
   WHEN foreign_key_violation THEN
      RAISE NOTICE 'Error: foreign key violation';
END;
```

- Prevents the whole transaction from failing uncontrolled.
- You can log errors, revert part of logic, or raise custom messages.

Parameters in Stored Procedures

Parameters allow passing values **into** and **out of** a procedure.

PostgreSQL supports:

1. **IN**
    - Default type if not specified (for procedures).
    - Passes a value **into** the procedure.
    - Cannot be modified to return something back.
2. **OUT**

   - Used to **return values** from the procedure.
   - Act like extra result columns.
3. **INOUT**
   - The parameter is passed in and can be **modified**.
   - Final value is returned to the caller.

Conceptually:

```
CREATE PROCEDURE adjust_salary(IN emp_id int, IN delta numeric)
...

CREATE PROCEDURE get_salary(IN emp_id int, OUT salary numeric)
...

CREATE PROCEDURE change_and_report(INOUT amount numeric)
...
```

Function vs Stored Procedures

Functions

- **Must return** a value (scalar, row, table, or `void`).
- Typically used in:
  - `SELECT`, `WHERE`, `JOIN`, etc.
- **No transaction control** inside:
  - Cannot `COMMIT` or `ROLLBACK` within a function.

Stored Procedures

- May **return no value**.
- Invoked with `CALL procedure_name(...)`.
- **Support transaction control inside**:
  - Can use `COMMIT`, `ROLLBACK`, `SAVEPOINT` etc. in certain patterns.

- Better suited for **multi-step workflows** that need control over transactions.

Dynamic SQL in Stored Procedures

> **Dynamic SQL** = constructing SQL text at runtime and executing it.

Usage pattern:

```
EXECUTE 'DELETE FROM ' || quote_ident(table_name) ||
      ' WHERE created_at < $1'
USING cutoff_date;
```

- Allows building queries where table name, columns, or conditions are dynamic.
- Very powerful but must be used carefully (SQL injection, complexity).