PLSQL

Overview

- PL/SQL is a procedural extension to SQL.
  - The “data manipulation” capabilities of “SQL” are combined with the “processing capabilities” of a “procedural language”.
  - PL/SQL provides features like conditional execution, looping and branching.
  - PL/SQL supports subroutines, as well.
  - PL/SQL program is of block type, which can be “sequential” or “nested” (one inside the other).

Salient Features

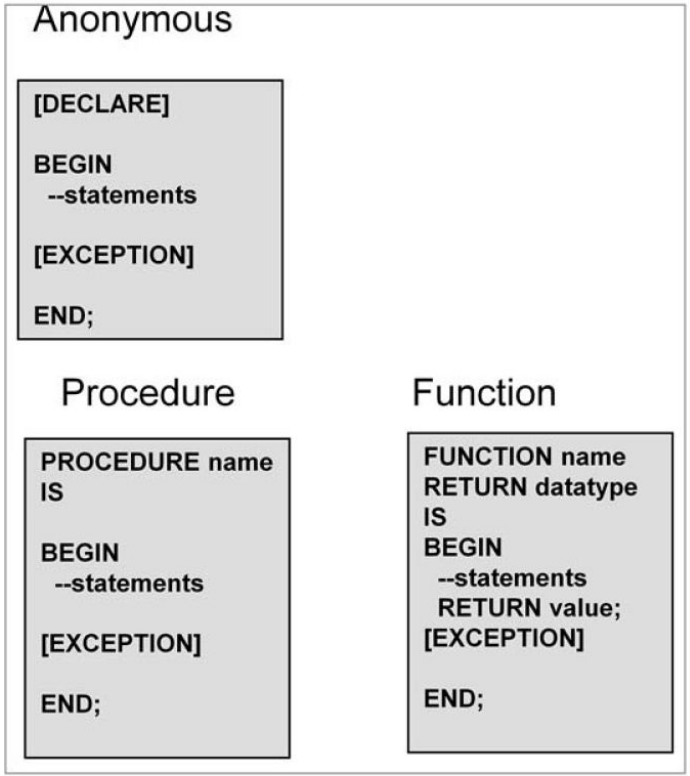
- PL/SQL provides the following features:
  - Tight Integration with SQL
  - Better performance
    - Several SQL statements can be bundled together into one PL/SQL block and sent to the server as a single unit.
  - Standard and portable language
    - Although there are a number of alternatives when it comes to writing software to run against the Oracle Database, it is easier to run highly efficient code in PL/SQL, to access the Oracle Database, than in any other language.

PL/SQL Block Structure

- A PL/SQL block comprises of the following structures:
  - DECLARE – Optional
    - Variables, cursors, user-defined exceptions
  - BEGIN – Mandatory
    - SQL statements
    - PL/SQL statements
  - EXCEPTION – Optional
    - Actions to perform when errors occur
  - END; – Mandatory

Block Types

- There are three types of blocks in PL/SQL:
  - Anonymous
  - Named:
    - Procedure
    - Function



Representation of a PL/SQL block:

- The notations used in a PL/SQL block are given below:
    1. -- is a single line comment.
    2. /* */ is a multi-line comment.
    3. Every statement must be terminated by a semicolon (;).
    4. PL/SQL block is terminated by a slash (/) on a line by itself.
- A PL/SQL block must have an “Execution section”.
- It can optionally have a “Declaration section” and an Exception section, as well.

Guidelines for declaring variables

- Given below are a few guidelines for declaring variables:
  - follow the naming conventions
  - initialize the variables designated as NOT NULL
  - initialize the identifiers by using the assignment operator (:=) or by using the DEFAULT reserved word
  - declare at most one Identifier per line

Types of Variables

- PL/SQL variables
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- Non-PL/SQL variables
  - Bind and host variables

📘 PL/pgSQL Data Types & Variable Declarations (PostgreSQL)

1️⃣ General Syntax

sql

```
identifier [CONSTANT] data_type [NOT NULL] [:= expression];
```

- **identifier** → variable name
- **CONSTANT** → optional, makes variable read-only
- **data_type** → any valid PostgreSQL data type
- **NOT NULL** → ensures variable cannot hold NULL
- **:= expression** → assigns an initial value

2️⃣ Common Data Types in PL/pgSQL

🔹 Numbers

PostgreSQL Type	Description	Oracle Equivalent
`SMALLINT`	-32768 to 32767	`NUMBER(5)`
`INTEGER` / `INT`	-2B to 2B	`NUMBER(10)`
`BIGINT`	-9 quintillion	`NUMBER(19)`
`NUMERIC(p,s)`	Arbitrary precision	`NUMBER(p,s)`
`REAL`	4-byte floating-point	`BINARY_FLOAT`
`DOUBLE PRECISION`	8-byte floating-point	`BINARY_DOUBLE`

Examples

sql

```sql
DECLARE
  v_count INTEGER := 100;
  v_price NUMERIC(10,2) := 4599.99;
  v_ratio REAL := 0.75;
```

🔹 Character Strings

PostgreSQL Type	Description	Oracle Equivalent
`VARCHAR(n)`	Variable length, max `n`	`VARCHAR2(n)`
`CHAR(n)`	Fixed length	`CHAR(n)`
`TEXT`	Unlimited length string	`CLOB`

```
DECLARE
  v_name VARCHAR(50) := 'Alice';
  v_code CHAR(5) := 'P1234';
  v_description TEXT := 'This is a long string value';
```

🔹 Dates & Time

PostgreSQL Type	Description	Oracle Equivalent
`DATE`	Calendar date (YYYY-MM-DD)	`DATE`
`TIME`	Time of day	`DATE` (time part only)
`TIMESTAMP`	Date + time (no TZ)	`TIMESTAMP`
`TIMESTAMPTZ`	Timestamp with time zone	`TIMESTAMP WITH TIME ZONE`
`INTERVAL`	Time span	`INTERVAL`

Examples

sql

```sql
DECLARE
  v_start_date DATE := CURRENT_DATE;
  v_start_time TIME := LOCALTIME;
  v_full_time TIMESTAMP := NOW();
  v_gap INTERVAL := '2 days 3 hours';
```

🔹 Boolean

PostgreSQL Type	Description	Oracle Equivalent
`BOOLEAN`	TRUE / FALSE / NULL	`BOOLEAN` (only in PL/SQL, not in Oracle tables)

Example

sql

Copy Edit

```sql
DECLARE
  v_active BOOLEAN := TRUE;
```

1. Assignment operator

- In PostgreSQL PL/pgSQL you **cannot use `=` for assignment**.
- Use `:=` instead (just like you did in `DECLARE`).

✅ Correct:

```plpgsql
ab := ab + 20;
```

4️⃣ Using Variables

Inside the block, use variables directly. To **print**, use `RAISE NOTICE`:

```sql
DO $$
DECLARE
  v_name VARCHAR(20) := 'Alice';
  v_salary NUMERIC(10,2) := 2500.75;
BEGIN
  RAISE NOTICE 'Employee: %, Salary: %', v_name, v_salary;
END;
$$;
```

```
DO $$
BEGIN
   RAISE NOTICE 'Hello, this is an anonymous block!';
END;
$$;
```

- `DO` → tells Postgres you want to execute a code block.
- `$$ ... $$` → wraps your code (instead of quotes).
- `BEGIN ... END;` → the body of the block.
- `RAISE NOTICE` → just prints a message.

Lab 1

Declaring Datatype by using %TYPE Attribute

- While using the %TYPE Attribute:
  - Declare a variable according to:
    - a database column definition
    - another previously declared variable
  - Prefix %TYPE with:
    - the database table and column
    - the previously declared variable name

Declaring Datatype by using %TYPE Attribute

- Example:

```
...
v_name          staff_master.staff_name%TYPE;
v_balance       NUMBER(7,2);
v_min_balance   v_balance%TYPE := 10;
...
```

Declaring Datatype by using %ROWTYPE

- Example:

```
DECLARE
      nRecord staff_master%rowtype;
BEGIN
      SELECT * into nrecord
                  FROM staff_master
                  WHERE staff_code = 100001;

      UPDATE staff_master
                  SET staff_sal = staff_sal + 101
                  WHERE emp_code = 100001;
END;
```

Lab 2 & 3

Composite Data Types

- Composite Datatypes in PL/SQL:
  - Composite datatype available in PL/SQL:
    - records
  - A composite type contains components within it. A variable of a composite type contains one or more scalar variables.

1.3: Handling Variables in PL/SQL

Record Data Types

- Record Datatype:
  - A record is a collection of individual fields that represents a row in the table.
  - They are unique and each has its own name and datatype.
  - The record as a whole does not have value.
- Defining and declaring records:
  - Define a RECORD type, then declare records of that type.
  - Define in the declarative part of any block, subprogram, or package.

`RECORD` is **not tied to a specific table** — it is more flexible, but its structure is determined only **at runtime**.

3. RECORD vs `%ROWTYPE`

Feature	RECORD	%ROWTYPE
Structure	Determined at runtime from query result	Fixed at compile time (from a table/view)
Flexibility	Can store any row shape (different queries)	Only stores row matching a specific table
Example use	Iterating over arbitrary SELECT queries	When working with all columns of a table

```
DO $$
DECLARE
   rec RECORD;
BEGIN
   SELECT id, name INTO rec FROM employees WHERE id = 1;
   RAISE NOTICE 'Employee: %, %', rec.id, rec.name;

   -- Now use a different query, same record variable
   SELECT department, salary INTO rec FROM employees WHERE id = 1;
   RAISE NOTICE 'Dept: %, Salary: %', rec.department, rec.salary;
END;
$$;
```

- Same `rec` variable works with different queries (structure changes dynamically).

3. Creating a custom composite type

If you want something like Oracle's custom `TYPE`, you can define it once and reuse it:

```sql
CREATE TYPE recname AS (
   customer_id INT,
   customer_name VARCHAR(20)
);

DO $$
DECLARE
   var_rec recname;
BEGIN
   var_rec.customer_id := 20;
   var_rec.customer_name := 'Smith';

   RAISE NOTICE 'ID: %, Name: %', var_rec.customer_id, var_rec.customer_name;
END;
$$;
```

🔹 Scope & Visibility of Variables in PostgreSQL PL/pgSQL

1. **Variables live inside the `DECLARE` section** of a block (`DO`, function, or procedure).
2. **Nested blocks can declare their own variables** (using another `DECLARE` before `BEGIN`).
3. If a nested block **redeclares the same variable name**, the *inner declaration shadows the outer one*.
    - Outer variables are still accessible, but you must use **block labels** to reference them.
4. When a block ends, its variables go out of scope.

```sql
DO $outer$
DECLARE
    v_available_flag BOOLEAN := TRUE;
    v_ssn NUMERIC(9) := 123456789;  -- outer variable
BEGIN
    RAISE NOTICE 'Outer Block: v_ssn = %', v_ssn;

    -- Inner block starts
    DECLARE
        v_ssn CHAR(11) := 'ABC12345678';  -- inner variable shadows outer
        v_startdate DATE := CURRENT_DATE;
    BEGIN
        RAISE NOTICE 'Inner Block: v_ssn = %', v_ssn;  -- refers to inner
        RAISE NOTICE 'Inner Block: v_startdate = %', v_startdate;
        RAISE NOTICE 'Inner Block: outer v_ssn = %', outer.v_ssn; -- refer to outer explicitly
    END;

    -- Back to outer block
    RAISE NOTICE 'Outer Block Again: v_ssn = %', v_ssn;
END;
$outer$;
```

IF Construct

- Given below is a list of Programmatic Constructs which are used in PL/SQL:
  - Conditional Execution:
    - This construct is used to execute a set of statements only if a particular condition is TRUE or FALSE.
    - Syntax:

```
IF Condition_Expr
THEN
        PL/SQL_Statements
END IF;
```

Example in PostgreSQL

sql

```sql
DO $$
DECLARE
    v_num INT := 10;
BEGIN
    IF v_num > 0 THEN
        RAISE NOTICE 'Positive number';
    ELSIF v_num = 0 THEN
        RAISE NOTICE 'Zero';
    ELSE
        RAISE NOTICE 'Negative number';
    END IF;
END;
$$;
```

Simple Loop

- Looping
  - A LOOP is used to execute a set of statements more than once.
  - Syntax:

```
LOOP

      PL/SQL_Statements;
END LOOP ;
```

Simple Loop – EXIT statement

- EXIT
  - Exit path is provided by using EXIT or EXIT WHEN commands.
  - EXIT is an unconditional exit. Control is transferred to the statement following END LOOP, when the execution flow reaches the EXIT statement.

contd.

Simple Loop – EXIT statement

- Syntax:

```
BEGIN
        .....
        .....
        LOOP
                        IF <Condition> THEN
        . . . . .
                        EXIT ;                          -- Exits loop immediately
                        END IF ;

        END LOOP;
        LOOP
                        ..........
                        ..........
                        EXIT WHEN <condition>
        END LOOP;

        . . . . .                                                   -- Control resumes here
        COMMIT ;
        END ;
```

1️⃣ Numeric `FOR` loop

Loops from a start number to an end number.

```sql
DO $$
DECLARE
    i integer;
BEGIN
    FOR i IN 1..5 LOOP   -- i goes from 1 to 5
        RAISE NOTICE 'Iteration %', i;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

Notes:

- `1..5` means inclusive range.
- You can use `REVERSE` to loop backward:

2️⃣ Loop over a query result

You can iterate over rows returned by a `SELECT` query.

```sql
DO $$
DECLARE
    rec RECORD;
BEGIN
    FOR rec IN SELECT id, name FROM users LOOP
        RAISE NOTICE 'User ID: %, Name: %', rec.id, rec.name;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

3️⃣ Loop over an array

sql

```
DO $$
DECLARE
    val integer;
    arr integer[] := ARRAY[10, 20, 30];
BEGIN
    FOR val IN ARRAY arr LOOP
        RAISE NOTICE 'Value: %', val;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

LAB 4

Understanding Exception Handling in PL/SQL

- Error Handling:
  - In PL/SQL, a warning or error condition is called an “exception”.
    - Exceptions can be internally defined (by the run-time system) or user defined.
    - Examples of internally defined exceptions:
      - division by zero
      - out of memory
    - Some common internal exceptions have predefined names, namely:
      - ZERO_DIVIDE
      - STORAGE_ERROR
    - The other exceptions can be given user-defined names.
    - Exceptions can be defined in the declarative part of any PL/SQL block, subprogram, or package. These are user-defined exceptions.

1️⃣ Common predefined exceptions in PostgreSQL

Exception Name	Description
`DIVISION_BY_ZERO`	Raised when dividing by zero.
`NUMERIC_VALUE_OUT_OF_RANGE`	Raised when a numeric operation overflows.
`STRING_DATA_RIGHT_TRUNCATION`	Raised when inserting a string too long for the column.
`NULL_VALUE_NOT_ALLOWED`	Raised when `NOT NULL` constraint is violated.
`RAISE_EXCEPTION`	Generic exception raised with `RAISE EXCEPTION`.
`CHECK_VIOLATION`	When a `CHECK` constraint fails.
`UNIQUE_VIOLATION`	When a `UNIQUE` constraint fails.
`FOREIGN_KEY_VIOLATION`	When a foreign key constraint is violated.
`INSUFFICIENT_PRIVILEGE`	Raised for permission issues.
`SYNTAX_ERROR`	Raised when there is a syntax error in SQL execution.

2️⃣ How to use exceptions

sql

```sql
DO $$
BEGIN
    -- Example: division by zero
    RAISE NOTICE 'Result: %', 10 / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Caught division by zero!';
END
$$ LANGUAGE plpgsql;
```

- In PostgreSQL PL/pgSQL, the same situation **raises a predefined exception called `NO_DATA_FOUND`, but only if you use `SELECT INTO`.**

Example:

sql

Copy Edit

```
DO $$
DECLARE
    myval integer;
BEGIN
    -- Suppose table 'users' has no id = 999
    SELECT id INTO myval FROM users WHERE id = 999;

    RAISE NOTICE 'Value: %', myval;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE NOTICE 'No row found!';
END
$$ LANGUAGE plpgsql;
```

✅ Key points:

1. Only triggers when `SELECT INTO` finds **no rows.**
2. If multiple rows are returned → raises `TOO_MANY_ROWS` (similar to Oracle).

User-defined Exception

- User-defined Exceptions are:
  - declared in the Declaration section,
  - raised in the Executable section, and
  - handled in the Exception section

User-defined Exception - Example

- Here is an example of User Defined Exception:

```
DECLARE
      E_Balance_Not_Sufficient EXCEPTION;
      E_Comm_Too_Large EXCEPTION;
      …
BEGIN
      NULL;
END;
```

Raising Exceptions

- Raising Exceptions:
  - Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that are associated with an Oracle error number using EXCEPTION_INIT.
  - Other user-defined exceptions must be raised explicitly by RAISE statements.
    - The syntax is:

PostgreSQL equivalent:

```plpgsql
RAISE EXCEPTION 'Dept: % already exists', v_department;
```

```
DO $$
DECLARE
    v_counter integer;
    v_department smallint := 50;
BEGIN
    SELECT count(*) INTO v_counter
    FROM department_master
    WHERE dept_code = 50;

    IF v_counter > 0 THEN
        RAISE EXCEPTION 'Dept: % already exists', v_department;
    END IF;

    INSERT INTO department_master (dept_code, dept_name)
    VALUES (v_department, 'new name');

EXCEPTION
    WHEN OTHERS THEN
        INSERT INTO error_log (message)
        VALUES (SQLERRM);
```

`SQLERRM` gives the error message in the `EXCEPTION` block.

```
DO $$
DECLARE
    my_custom_exception EXCEPTION;
BEGIN
    RAISE NOTICE 'Before exception';

    -- Raise custom exception
    RAISE my_custom_exception USING MESSAGE = 'This is my custom exception!';

    RAISE NOTICE 'This will not run because exception is raised';
END;
$$;
```

```
DO $$
DECLARE
    my_custom_exception EXCEPTION;
BEGIN
    RAISE NOTICE 'Before exception';

    -- Raise custom exception
    RAISE my_custom_exception USING MESSAGE = 'This is my custom exception!';

    RAISE NOTICE 'This will not run because exception is raised';
END;
$$;
```

2.4: OTHERS Exception Handler

OTHERS Exception Handler (contd..)

✅ Available Variables in PL/pgSQL Exception Handling

- `SQLSTATE` → The 5-character error code (ANSI standard).
- `SQLERRM` → The text message of the exception.

```
WHEN OTHERS THEN
```

Copy Edit

This will **catch any exception type** (whether system-defined or custom-defined) that you didn't explicitly handle.

```
DO $$
BEGIN
    -- Example: divide by zero
    PERFORM 10 / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Caught division by zero!';
        RAISE NOTICE 'SQLSTATE: %, SQLERRM: %', SQLSTATE, SQLERRM;

    WHEN OTHERS THEN
        RAISE NOTICE 'Unhandled error - SQLSTATE: %, Message: %', SQLSTATE, SQLER
END;
$$;
```

Introduction

- A subprogram is a named block of PL/SQL
- There are two types of subprograms in PL/SQL, namely: Procedures and Functions
- Each subprogram has:
  - A declarative part
  - An executable part or body, and
  - An exception handling part (which is optional)
- A function is used to perform an action and return a single value

Anonymous Blocks & Stored Subprograms Comparison

Anonymous Blocks	Stored Subprograms/Named Blocks
1. Anonymous Blocks do not have names.	1. Stored subprograms are named PL/SQL blocks.
2. They are interactively executed. The block needs to be compiled every time it is run.	2. They are compiled at the time of creation and stored in the database itself. Source code is also stored in the database.
3. Only the user who created the block can use the block.	3. Necessary privileges are required to execute the block.

Procedures

- A procedure is used to perform an action.
- It is illegal to constrain datatypes.
- Syntax:

```
CREATE PROCEDURE  Proc_Name
   (Parameter {IN | OUT | IN OUT} datatype := value,...) AS

   Variable_Declaration ;
   Cursor_Declaration ;
   Exception_Declaration ;
BEGIN
    PL/SQL_Statements ;
EXCEPTION
    Exception_Definition ;
END Proc_Name ;
```

Basic Syntax

sql

```
CREATE [OR REPLACE] PROCEDURE procedure_name(
    param1 datatype [IN | OUT | INOUT],
    param2 datatype [IN | OUT | INOUT],
    ...
)
LANGUAGE plpgsql
[AS $$
DECLARE
    -- local variables (optional)
BEGIN
    -- procedural statements
END;
$$];
```

Subprogram Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Used to pass values to the procedure.	Used to return values to the caller.	Used to pass initial values to the procedure and return updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an uninitialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression, but should be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, literal, initialized variable, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.
Actual parameter is passed by reference (a pointer to the value is passed in).	Actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified.	Actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified.

Simple Example

sql

```sql
CREATE OR REPLACE PROCEDURE insert_student(
    p_name TEXT,
    p_email TEXT
)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO students (name, email) VALUES (p_name, p_email);
    RAISE NOTICE 'Student % with email % inserted!', p_name, p_email;
END;
$$;
```

Calling the Procedure:

sql

```sql
CALL insert_student('Eve', 'eve@example.com');
```

Example with INOUT Parameter

sql

```sql
CREATE OR REPLACE PROCEDURE add_numbers(
    IN a INT,
    IN b INT,
    INOUT sum_result INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    sum_result := a + b;
END;
$$;
```

```
-- Call
DO $$
DECLARE
    total INT := 0;
BEGIN
    CALL add_numbers(10, 20, total);
    RAISE NOTICE 'Sum = %', total;  -- prints 30
END;
$$;
```

Functions

- A function is similar to a procedure.
- A function is used to compute a value.
  - A function accepts one or more parameters, and returns a single value by using a return value.
  - A function can return multiple values by using OUT parameters.
  - A function is used as part of an expression, and can be called as Lvalue = Function_Name(Param1, Param2, …….).
  - Functions returning a single value for a row can be used with SQL statements.

Basic Syntax for a Function

sql

```sql
CREATE [OR REPLACE] FUNCTION function_name(
    param1 datatype [DEFAULT value],
    param2 datatype [DEFAULT value],
    ...
)
RETURNS return_datatype
LANGUAGE plpgsql
AS $$
DECLARE
    -- local variables (optional)
BEGIN
    -- procedural statements
    RETURN some_value;  -- must return a value matching RETURNS
END;
$$;
```

- `OR REPLACE` → optional, replaces existing function with same name.
- `RETURNS` → type of value function will return (`INT`, `TEXT`, `TABLE(...)`, `VOID`, etc.)
- `DECLARE` → optional local variables.
- `RETURN` → must return a value if type is not `VOID`.

Example 1: Function Returning a Scalar Value

```sql
CREATE OR REPLACE FUNCTION get_student_email(p_id INT)
RETURNS TEXT AS $$
DECLARE
    v_email TEXT;
BEGIN
    SELECT email INTO v_email
    FROM students
    WHERE id = p_id;

    RETURN v_email;
END;
$$ LANGUAGE plpgsql;
```

Usage:

```sql
SELECT get_student_email(1);
```

🔹 Example 2: Function Returning a Table

```sql
CREATE OR REPLACE FUNCTION get_students_above_age(min_age INT)
RETURNS TABLE(id INT, name TEXT, email TEXT) AS $$
BEGIN
    RETURN QUERY
    SELECT id, name, email
    FROM students
    WHERE EXTRACT(YEAR FROM AGE(current_date, birth_date)) > min_age;
END;
$$ LANGUAGE plpgsql;
```

Usage:

```sql
SELECT * FROM get_students_above_age(18);
```

Example 3: Function with INOUT Parameters

sql

```sql
CREATE OR REPLACE FUNCTION add_numbers(a INT, b INT, OUT sum_result INT)
AS $$
BEGIN
    sum_result := a + b;
END;
$$ LANGUAGE plpgsql;

-- Call
SELECT add_numbers(10, 20);  -- returns 30
```

🔹 Key Difference

Feature	Function	Procedure
Return value	Must return something	Does not return a value
Call in SQL	`SELECT function_name(...)`	❌ Cannot use `SELECT`
Call syntax	`SELECT function_name(...)`	`CALL procedure_name(...)`
Transaction control	❌ Cannot commit/rollback	✅ Can commit/rollback

🔹 Functions in PostgreSQL

- **Functions do NOT allow transaction control** (no `COMMIT` or `ROLLBACK`).
- They **run inside the calling transaction**.
- **Effect on data depends on the transaction that called the function**:
  - If the caller commits, changes made by the function are committed.
  - If the caller rolls back, all changes in the function are rolled back too.
- **Functions are not auto-commit** — they inherit the transaction context.

Example:

```sql
BEGIN;
SELECT insert_student_function('Alice', 'alice@example.com');
-- still not committed
ROLLBACK;  -- Alice will NOT be inserted
```

🔹 Procedures in PostgreSQL (PostgreSQL 11+)

- **Procedures CAN perform transaction control**.
- Inside a procedure, you can explicitly use:

sql Copy Edit

```sql
COMMIT;
ROLLBACK;
SAVEPOINT sp;
```

- **Procedures do not auto-commit by default**, but you can explicitly commit within them.

Example:

sql Copy Edit

```sql
CREATE PROCEDURE add_student_proc(p_name TEXT, p_email TEXT)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO students(name, email) VALUES (p_name, p_email);
    COMMIT; -- explicit commit
END;
$$;
```

🔑 Summary

Feature	Auto-commit?	Transaction control allowed?
Function	No	❌ Cannot COMMIT/ROLLBACK
Procedure	No (but can commit explicitly)	✅ Can COMMIT/ROLLBACK

So in short:

- **Functions:** run in caller's transaction, no auto-commit, no explicit commit.
- **Procedures:** can manage transactions, no auto-commit by default, can commit explicitly.

Concept of Database Triggers

- Database Triggers:
  - Database Triggers are procedures written in PL/SQL, Java, or C that run (fire) implicitly:
    - whenever a table or view is modified, or
    - when some user actions or database system actions occur
  - They are stored subprograms.

Concept of Database Triggers

- You can write triggers that fire whenever one of the following operations occur:
  - User events:
    - DML statements on a particular schema object
    - DDL statements issued within a schema or database
    - user logon or logoff events
  - System events:
    - server errors
    - database startup
    - instance shutdown

Usage of Triggers

- Triggers can be used for:
  - maintaining complex integrity constraints.
  - auditing information, that is the Audit trail.
  - automatically signaling other programs that action needs to take place when changes are made to a table.

🔹 Basic Syntax for a Trigger

1. **Create the trigger function** (must be a function returning `TRIGGER`):

sql

```sql
CREATE OR REPLACE FUNCTION function_name()
RETURNS TRIGGER AS $$
BEGIN
    -- trigger logic here
    RETURN NEW;  -- or RETURN OLD for DELETE triggers
END;
$$ LANGUAGE plpgsql;
```

2. Create the trigger on a table:

sql

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW | FOR EACH STATEMENT]
EXECUTE FUNCTION function_name();
```

🔹 Explanation

- `BEFORE` → Trigger fires **before the operation**.
- `AFTER` → Trigger fires **after the operation**.
- `FOR EACH ROW` → Trigger fires **once per row** affected.
- `FOR EACH STATEMENT` → Trigger fires **once per statement**, regardless of number of rows.
- `RETURN NEW` → Required for `BEFORE` triggers; returns the new row.
- `RETURN OLD` → Used in DELETE triggers if needed.

✅ Case by case

1. INSERT

- Only `NEW` is available.
- If you `RETURN NEW`, the new row is inserted into the table as usual.
- If you `RETURN NULL`, the insert is **skipped** (row not inserted).

2. UPDATE

- Both `OLD` and `NEW` are available.
- If you `RETURN NEW`, the updated row gets stored in the table.
- If you `RETURN NULL`, the update is skipped (row not changed).
- If you modify `NEW` before returning it, you can change the row before it's saved.

3. DELETE

- Only `OLD` is available.
- If you `RETURN OLD`, the delete proceeds normally.
- If you `RETURN NULL`, the delete is skipped (row not deleted).

🔑 Key Notes

1. **Trigger function must return** `TRIGGER`.
2. Use `NEW` and `OLD` to access row data:
    - `NEW.column` → new values (INSERT/UPDATE)
    - `OLD.column` → old values (UPDATE/DELETE)
3. Triggers can be **BEFORE** or **AFTER**.
4. Triggers can be **row-level** or **statement-level**.

Database Triggers (contd.):

Parts of a Trigger

A trigger has three basic parts:

- A triggering event or statement
- A trigger restriction
- A trigger action

Triggering Event or Statement

- A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:
    - An INSERT, UPDATE, or DELETE statement on a specific table (or view, in some cases)
    - A CREATE, ALTER, or DROP statement on any schema object
    - A database startup or instance shutdown
    - A specific error message or any error message
    - A user logon or logoff

Trigger Restriction

- A trigger restriction specifies a Boolean expression that must be true for the trigger to fire. The trigger action is not run if the trigger restriction evaluates to false or unknown.

Trigger Action

- A trigger action is the procedure (PL/SQL block, Java program, or C callout) that contains the SQL statements and code to be run when the following events occur:
    - a triggering statement is issued
    - the trigger restriction evaluates to true
- Like stored procedures, a trigger action can:
    - contain SQL, PL/SQL, or Java statements
    - define PL/SQL language constructs such as variables, constants, cursors, exceptions
    - define Java language constructs
    - call stored procedures

Disabling and Dropping Triggers

- To disable a trigger:

```
ALTER  TRIGGER Trigger_Name DISABLE/ENABLE
```

- To drop a trigger (by using drop trigger command):

```
DROP TRIGGER Trigger_Name
```

Using :Old & :New values in Triggers

- Note: They are valid only within row level triggers and not in statement level triggers.

Triggering statement	:Old	:New
INSERT	**Undefined – all fields are null.**	**Values that will be inserted when the statement is complete.**
UPDATE	**Original values for the row before the update.**	**New values that will be updated when the statement is complete.**
DELETE	**Original values before the row is deleted.**	**Undefined – all fields are NULL.**

Thank You!