Xi'an Jiaotong-Liverpool University
西交利物浦大学

DTS207TC - Database Development and Design: Lab 3 Advanced SQL

Course Leader: Dr.Zhang Di, Co-Teacher: Dr. Affan Yasin

September 23, 2025

Contents

1 PostgreSQL Functions **3**
1.1 What is a Function? 3
1.2 Types of Functions in PostgreSQL 3
1.3 Anatomy of a Function 3
1.4 PostgreSQL Functions: Fill-in-the-Blank Revision 5
1.5 PostgreSQL Functions: Fill-in-the-Blank – Solution – 7
1.6 Some more Examples 8

2 Understanding Database Triggers **10**
2.1 What are Triggers? 10
2.2 A Real-Life Example: E-commerce Website 10
2.3 What If We Don't Use Triggers and perform it manually ? 11
2.4 Summary of the Trigger Example 11
2.5 Run the code 11
2.6 Fill-in-the-Blanks: Test Your Trigger Knowledge 13

3 Recursive Function **13**
3.1 What is a Recursive Function? 13
3.2 Applications in E-commerce/Digital World: 14
3.3 Key Points: 14
3.4 Fill in the Blank - Exercise 15

4 Practice Questions **16**
4.1 Practice Examples for PostgreSQL Functions 16
4.2 Trigger Practice Task 20

Instructions

1. First create the database, tables, and import the data, or run the queries given to create and import the data.
2. The table creation and data import are already performed in the first week. If you have issues, you can contact the teacher in the lab or inother case generate your own data and run the queries for revision.

Message from Teachers

“Every expert was once a beginner.”

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING

1 PostgreSQL Functions

1.1 What is a Function?

In PostgreSQL, a **function** is a reusable piece of code that performs a specific task and can return a value. Think of it like a *mini-program* inside your database. Functions help you avoid writing the same SQL queries repeatedly.

Example:

- Calculate the area of a rectangle
- Convert temperature from Celsius to Fahrenheit
- Find the total marks of a student

1.2 Types of Functions in PostgreSQL

1. **SQL Functions** – Use simple SQL statements.
2. **PL/pgSQL Functions** – PL/pgSQL stands for **Procedural Language/PostgreSQL**. It allows you to write **functions**. It can have loops, conditions, and variables (more powerful). Think of it as SQL + programming features (like a mini programming language inside PostgreSQL)
3. **Other Languages** – e.g., PL/Python, PL/Perl.

1.3 Anatomy of a Function

A function usually has:

- **Name** – What you call the function
- **Parameters** – Inputs the function needs
- **Return type** – What the function gives back
- **Body** – The SQL or PL/pgSQL code it runs

Function Template

```
CREATE OR REPLACE FUNCTION function_name(parameter_name data_type)
RETURNS return_data_type
LANGUAGE plpgsql
AS $$
BEGIN
    -- your SQL/PL code here
    RETURN something;
END;
$$;
```

1.3.1 Example A: Add Two Numbers

Example: Add Two Numbers

```
CREATE OR REPLACE FUNCTION add_numbers(a INT, b INT)
RETURNS INT
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN a + b;
END;
$$;

-- Test it
SELECT add_numbers(5, 3);
-- Result: 8
```

1.3.2 Example B: Check Pass/Fail

Example: Check Pass/Fail

```
CREATE OR REPLACE FUNCTION check_pass(mark INT)
RETURNS TEXT
LANGUAGE plpgsql
AS $$
BEGIN
    IF mark >= 50 THEN
        RETURN 'Pass';
    ELSE
        RETURN 'Fail';
    END IF;
END;
$$;

-- Test it
SELECT check_pass(70);  -- Pass
SELECT check_pass(40);  -- Fail
```

1.3.3 Example C: Function Using Table Data

Example: Get Average Age

```
CREATE OR REPLACE FUNCTION get_total(student_id INT)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    avg_age NUMERIC;
BEGIN
    SELECT AVG(age) INTO avg_age
    FROM students s
    WHERE s.student_id = get_total.student_id;  -- fully qualify the function parameter

    RETURN avg_age;
END;
$$;

-- Test it
SELECT get_total(1);
```

1.4 PostgreSQL Functions: Fill-in-the-Blank Revision

Q1: Basic Function Template

Fill in the blanks:

```
CREATE OR REPLACE FUNCTION ________(parameter_name data_type)
RETURNS ________
LANGUAGE plpgsql
AS $$
BEGIN
    -- your SQL/PL code here
    RETURN ________;
END;
$$;
```

Q2: Add Two Numbers

Fill in the blanks:

```
CREATE OR REPLACE FUNCTION add_numbers(a INT, b INT)
RETURNS ________
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN a ________ b;
END;
$$;
```

Q3: Check Pass/Fail

Fill in the blanks:

```
CREATE OR REPLACE FUNCTION check_pass(mark INT)
RETURNS ________
LANGUAGE plpgsql
AS $$
BEGIN
    IF mark ________ 50 THEN
        RETURN 'Pass';
    ELSE
        RETURN 'Fail';
    END IF;
END;
$$;
```

Q4: Average Age Function

Fill in the blanks:

```
CREATE OR REPLACE FUNCTION get_avg_age(student_id INT)
RETURNS ________
LANGUAGE plpgsql
AS $$
DECLARE
    avg_age ________;
BEGIN
    SELECT AVG(age) INTO avg_age
    FROM students s
    WHERE s.student_id = ________;

    RETURN avg_age;
END;
$$;
```

Q5: Test Function Call

Fill in the blanks:

```
-- Test the function
SELECT ________(5);
```

1.5 PostgreSQL Functions: Fill-in-the-Blank – Solution –

Solution Q1

```
CREATE OR REPLACE FUNCTION function_name(parameter_name data_type)
RETURNS return_data_type
LANGUAGE plpgsql
AS $$
BEGIN
    -- your SQL/PL code here
    RETURN something;
END;
$$;
```

Solution 2

```
CREATE OR REPLACE FUNCTION add_numbers(a INT, b INT)
RETURNS INT
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN a + b;
END;
$$;
```

Solution 3

```
CREATE OR REPLACE FUNCTION check_pass(mark INT)
RETURNS TEXT
LANGUAGE plpgsql
AS $$
BEGIN
    IF mark >= 50 THEN
        RETURN 'Pass';
    ELSE
        RETURN 'Fail';
    END IF;
END;
$$;
```

Solution 4

```
CREATE OR REPLACE FUNCTION get_avg_age(student_id INT)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    avg_age NUMERIC;
BEGIN
    SELECT AVG(age) INTO avg_age
    FROM students s
    WHERE s.student_id = get_avg_age.student_id;

    RETURN avg_age;
END;
$$;
```

Solution 5

```
-- Test the function
SELECT add_numbers(5);
-- or SELECT check_pass(5);
-- or SELECT get_avg_age(5);
```

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING

1.6 Some more Examples

1.6.1 Example A: Simple Function - Get Student Count

Function Code

```
CREATE OR REPLACE FUNCTION get_student_count()
RETURNS INT AS $$
BEGIN
    RETURN (SELECT COUNT(*) FROM students);
END;
$$ LANGUAGE plpgsql;
```

How it works: PostgreSQL runs the COUNT query and returns the integer value.

Run it:

```sql
SELECT get_student_count();
```

Tentative Output:

get_student_count
49

1.6.2 Example B: Function with Parameter - Count Courses for a Student

Function Code

```sql
CREATE OR REPLACE FUNCTION get_course_count(sid INT)
RETURNS INT AS $$
BEGIN
    RETURN (SELECT COUNT(*)
            FROM enrollments
            WHERE student_id = sid);
END;
$$ LANGUAGE plpgsql;
```

Run it:

```sql
SELECT get_course_count(1);
```

Tentative Output:

get_course_count
3

Explanation: Counts the courses student 1 is enrolled in.

1.6.3 Example C: Function with Logic - Calculate Average Grade of a Student

Function Code

```sql
CREATE OR REPLACE FUNCTION get_avg_grade(sid INT)
RETURNS NUMERIC AS $$
DECLARE
    avg_grade NUMERIC;
BEGIN
    SELECT AVG(grade) INTO avg_grade
    FROM enrollments
    WHERE student_id = sid;

    RETURN avg_grade;
END;
$$ LANGUAGE plpgsql;
```

Run it:

```
SELECT get_avg_grade(1);
```

Tentative Output:

get_avg_grade
85.3

Explanation: Calculates the average grade for student 1.

1.6.4 Practice - Fill in the Blanks

Fill the missing keyword in the function below:

```
CREATE OR REPLACE FUNCTION get_total_courses()
RETURNS INT AS $$
BEGIN
    RETURN (SELECT COUNT(*) FROM courses);
END;
$$ LANGUAGE _______;
```

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING

2 Understanding Database Triggers

2.1 What are Triggers?

A **trigger** is like an automatic helper inside the database. Whenever something happens to a table (`INSERT`, `UPDATE`, or `DELETE`), the trigger wakes up and performs an action without asking you. :)

Think of it as a "*watchdog*" that is always alert and ready to react.

2.2 A Real-Life Example: E-commerce Website

Let's imagine an online shopping system (e-commerce website such as JD.com, Taobao.com):

- When a customer places an **order**, a trigger can automatically reduce the **stock quantity**.
- When a **payment** is made, a trigger can add the details into a **payment history log**.
- If a customer account is **deleted**, a trigger can safely move the data into an **archive table** so nothing is lost.
- If someone tries to set a discount higher than 50%, a trigger can **stop it immediately**.

2.3 What If We Don't Use Triggers and perform it manually ?

- Someone might forget to log changes, so we lose track of who did what.
- Business rules might get skipped (for example, deleting important records by mistake).
- We need to write the same code again and again in our application, which is boring and error-prone.
- Data quality may suffer, leading to mistakes in reports and decisions.

Key Message

Triggers are like invisible assistants.They work silently in the background to make sure your database is safe, reliable, and smart.

2.4 Summary of the Trigger Example

Easy Summary

- We first create a **students table** to store student information (ID, name, age).
- Next, we make a **log table** which works like a diary, recording actions such as when a student is added.
- A **trigger function** is written to automatically insert a note into the log table whenever a new student is added.
- A **trigger** is then created to tell the database: "After inserting a student, run the trigger function."
- Finally, when we add students (e.g., Alice and Bob), the log table is automatically updated with their IDs, the action (INSERT), and the exact time.

In simple words: The database is keeping its own automatic diary. Every time a new student is added, it secretly writes a note in the log table, so we always know what happened and when.

2.5 Run the code

The code is uploaded on the Learning Mall. Kindly download and run on PostgreSQL for better understanding.

Trigger Example

```sql
-- Step 1: Create the main table for storing student data
CREATE TABLE studentsss (
    student_id SERIAL PRIMARY KEY,    -- unique ID for each student
    name VARCHAR(50),                 -- student name
    age INT                           -- student age
);

-- Step 2: Create a log table to record actions on students table
CREATE TABLE student_log (
    log_id SERIAL PRIMARY KEY,                -- unique log entry ID
    student_id INT,                           -- student ID related to the
       action
    action VARCHAR(20),                       -- type of action: INSERT,
       UPDATE, DELETE
    action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP  -- time when action
       happened
);

-- Step 3: Create a trigger function to log INSERT operations
CREATE OR REPLACE FUNCTION log_student_insert()
RETURNS TRIGGER AS $$
BEGIN
    -- Insert a record into student_log whenever a new student is added
    INSERT INTO student_log(student_id, action)
    VALUES (NEW.student_id, 'INSERT');
    RETURN NEW; -- return the new row so the INSERT can continue
END;
$$ LANGUAGE plpgsql;

-- Step 4: Create the trigger that calls the function AFTER an INSERT
CREATE TRIGGER after_student_insert
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_insert();

-- Step 5: Test the trigger
-- Insert a new student and check if log table is updated
INSERT INTO students (name, age) VALUES ('Alice', 20);
INSERT INTO students (name, age) VALUES ('Bob', 22);

-- Now run:
-- SELECT * FROM student_log;
-- You should see logs of both INSERT actions.
```

2.6 Fill-in-the-Blanks: Test Your Trigger Knowledge

Fill-in-the-Blanks Exercise

Complete the missing parts of the trigger code:

```sql
-- Step 1: Create the trigger function
CREATE OR REPLACE FUNCTION log_student_insert()
RETURNS TRIGGER AS $$
BEGIN
    -- Insert a record into the log table
    INSERT INTO student_log(student_id, action)
    VALUES (___, 'INSERT');    -- Fill: what refers to the new row?

    RETURN ___;                -- Fill: what should we return to continue
        insert?
END;
$$ LANGUAGE plpgsql;

-- Step 2: Create the trigger
CREATE TRIGGER after_student_insert
___ INSERT ON students        -- Fill: when should the trigger fire (BEFORE/
    AFTER)?
FOR EACH ___                  -- Fill: should it be FOR EACH ROW or STATEMENT?
EXECUTE FUNCTION ___();       -- Fill: which function should be called?
```

Hints for Students:

- 'NEW' is used to refer to the newly inserted row.
- Always 'RETURN NEW' for an INSERT trigger.
- Most logging triggers fire **AFTER INSERT**.
- 'FOR EACH ROW' ensures the trigger runs for every inserted row.
- The function name is the one you created in Step 1.

3 Recursive Function

3.1 What is a Recursive Function?

A recursive function is a function that **calls itself** to solve a problem. Recursive functions are useful for:

- Traversing hierarchical or nested data (e.g., categories and sub-categories in e-commerce)
- Calculations like factorial, Fibonacci series, or cumulative sums

- Generating sequences or performing repeated operations

3.2 Applications in E-commerce/Digital World:

- **Product Categories:** Displaying all sub-categories of a parent category
- **Order Hierarchies:** Summing nested order items or tracking multi-level discounts
- **Recommendation Systems:** Traversing user interactions or social connections recursively

3.3 Key Points:

- Always define a **base case** to avoid infinite recursion.
- Recursive functions can be less efficient than iterative solutions for very large inputs.

Example 1: Factorial

```
CREATE OR REPLACE FUNCTION factorial(n integer)
RETURNS bigint AS $$
BEGIN
    IF n = 0 THEN
        RETURN 1;  -- Base case
    ELSE
        RETURN n * factorial(n - 1);  -- Recursive call
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Usage:
SELECT factorial(5);  -- Returns 120
```

Example 2: Fibonacci Series

```
CREATE OR REPLACE FUNCTION fibonacci(n integer)
RETURNS bigint AS $$
BEGIN
    IF n = 0 THEN
        RETURN 0;
    ELSIF n = 1 THEN
        RETURN 1;
    ELSE
        RETURN fibonacci(n - 1) + fibonacci(n - 2);
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Usage:
SELECT fibonacci(7);  -- Returns 13
```

Example 3: Sum of Natural Numbers

```
CREATE OR REPLACE FUNCTION sum_natural(n integer)
RETURNS integer AS $$
BEGIN
    IF n <= 1 THEN
        RETURN n;
    ELSE
        RETURN n + sum_natural(n - 1);
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Usage:
SELECT sum_natural(10);  -- Returns 55
```

3.4 Fill in the Blank - Exercise

Exercise 1: Factorial Function

Complete the blanks to create a recursive factorial function:

```
CREATE OR REPLACE FUNCTION factorial(n integer)
RETURNS bigint AS $$
BEGIN
    IF n = __BLANK__ THEN
        RETURN __BLANK__;
    ELSE
        RETURN n * factorial(__BLANK__);
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Exercise 2: Fibonacci Function

Fill in the missing parts to create a recursive Fibonacci function:

```
CREATE OR REPLACE FUNCTION fibonacci(n integer)
RETURNS bigint AS $$
BEGIN
    IF n = 0 THEN
        RETURN __BLANK__;
    ELSIF n = 1 THEN
        RETURN __BLANK__;
    ELSE
        RETURN fibonacci(__BLANK__) + fibonacci(__BLANK__);
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Exercise 3: Sum of Natural Numbers

Complete the blanks to create a recursive sum function:

```
CREATE OR REPLACE FUNCTION sum_natural(n integer)
RETURNS integer AS $$
BEGIN
    IF n <= __BLANK__ THEN
        RETURN n;
    ELSE
        RETURN n + sum_natural(__BLANK__);
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Message from your Teachers

“Errors are proof that you are trying.”

Message from your Teachers

“The best way to learn programming is not only by reading it, but by doing it.”

4 Practice Questions

4.1 Practice Examples for PostgreSQL Functions

- **Q1: Double a Number** Write a function to double a given number.
- **Q2: Square of a Number** Write a function to calculate the square of a number.
- **Q3: Greater of Two Numbers** Write a function to return the greater of two numbers.
- **Q4: Total Number of Courses** Write a function to get the total number of courses in the courses table.
- **Q5: Get Student Name by ID** Write a function to get the name of a student given their ID.
- **Q6: Check if Student is Adult** Write a function to check if a student is an Adult (age ¿= 18) or Minor.

Message

“Don’t see the solution. First try by yourself !”

Solution 1

```
CREATE OR REPLACE FUNCTION double_number(n INT)
RETURNS INT AS $$
BEGIN
  RETURN n * 2;
END;
$$ LANGUAGE plpgsql;

-- Test
SELECT double_number(7);  -- 14
```

Solution 2

```
CREATE OR REPLACE FUNCTION square_number(n INT)
RETURNS INT AS $$
BEGIN
  RETURN n * n;
END;
$$ LANGUAGE plpgsql;

-- Test
SELECT square_number(5);  -- 25
```

Solution 3

```
CREATE OR REPLACE FUNCTION max_number(a INT, b INT)
RETURNS INT AS $$
BEGIN
  IF a > b THEN
    RETURN a;
  ELSE
    RETURN b;
  END IF;
END;
$$ LANGUAGE plpgsql;

-- Test
SELECT max_number(10, 15);  -- 15
```

Solution 4

```
CREATE OR REPLACE FUNCTION get_student_name(sid INT)
RETURNS VARCHAR AS $$
DECLARE
  sname VARCHAR;
BEGIN
  SELECT name INTO sname
  FROM students
  WHERE student_id = sid;

  RETURN sname;
END;
$$ LANGUAGE plpgsql;

-- Test
SELECT get_student_name(1);
```

Solution 5

```
CREATE OR REPLACE FUNCTION total_courses()
RETURNS INT AS $$
BEGIN
  RETURN (SELECT COUNT(*) FROM courses);
END;
$$ LANGUAGE plpgsql;

-- Test
SELECT total_courses();
```

Solution 6

```
CREATE OR REPLACE FUNCTION is_adult(sid INT)
RETURNS TEXT AS $$
DECLARE
  age_val INT;
BEGIN
  SELECT age INTO age_val
  FROM students
  WHERE student_id = sid;

  IF age_val >= 18 THEN
    RETURN 'Adult';
  ELSE
    RETURN 'Minor';
  END IF;
END;
$$ LANGUAGE plpgsql;

-- Test
SELECT is_adult(2);
```

4.2 Trigger Practice Task

Complete the SQL statements or triggers wherever indicated. These exercises are designed to help you understand triggers in an e-commerce context. The running code can be downloaded from the learning Mall course page.

1. Tables Setup

```
-- Q1: Create 'products' table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(50),
    stock INT
);

-- Q2: Create 'orders' table
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    customer_name VARCHAR(50),
    product_name VARCHAR(50),
    quantity INT,
    last_modified TIMESTAMP
);

-- Q3: Create 'order_audit' table
CREATE TABLE order_audit (
    audit_id SERIAL PRIMARY KEY,
    customer_name VARCHAR(50),
    product_name VARCHAR(50),
    action_time TIMESTAMP
);
```

2. Trigger Functions

```
-- Q4: Function to log new orders
CREATE OR REPLACE FUNCTION log_new_order()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO order_audit(customer_name, product_name, action_time)
    VALUES (NEW.customer_name, NEW.product_name, NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Q5: Function to prevent order deletion
CREATE OR REPLACE FUNCTION prevent_order_delete()
RETURNS TRIGGER AS $$
BEGIN
    RAISE EXCEPTION 'Cannot delete order % for %', OLD.order_id, OLD.customer_name;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

-- Q6: Function to update 'last_modified' timestamp
CREATE OR REPLACE FUNCTION update_order_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.last_modified = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Q7: Function to check stock before inserting order
CREATE OR REPLACE FUNCTION check_stock_before_order()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.quantity > (SELECT stock FROM products WHERE product_name = NEW.product_name) THEN
        RAISE EXCEPTION 'Not enough stock for %', NEW.product_name;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

3. Trigger Creation & Testing

```
-- Q8: Create trigger to log new orders
CREATE TRIGGER trg_new_order
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE FUNCTION log_new_order();

-- Q9: Create trigger to prevent deletion
CREATE TRIGGER trg_prevent_delete_order
BEFORE DELETE ON orders
FOR EACH ROW
EXECUTE FUNCTION prevent_order_delete();

-- Q10: Create trigger to update timestamp
CREATE TRIGGER trg_update_order_timestamp
BEFORE UPDATE ON orders
FOR EACH ROW
EXECUTE FUNCTION update_order_timestamp();

-- Q11: Create trigger to check stock before order
CREATE TRIGGER trg_check_stock
BEFORE INSERT ON orders
FOR EACH ROW
EXECUTE FUNCTION check_stock_before_order();

-- Q12: Insert sample product
INSERT INTO products(product_name, stock) VALUES ('Laptop', 5);

-- Q13: Insert sample order to test triggers
-- INSERT INTO orders(customer_name, product_name, quantity) VALUES ('John Doe', 'Laptop', 1);

-- Q14: Try deleting an order (should raise exception)
-- DELETE FROM orders WHERE order_id = 1;

-- Q15: Update order to test last_modified
-- UPDATE orders SET quantity = 2 WHERE order_id = 1;

-- Q16: Try placing an order exceeding stock (should raise exception)
-- INSERT INTO orders(customer_name, product_name, quantity)
-- VALUES ('Alice', 'Laptop', 10);
```