1. Introduction to PostgreSQL Functions

1.1 What is a Function?

A function in PostgreSQL is a reusable block of SQL statements that performs a specific task.

You can create a function once and use it whenever you need it.

For example:

- Calculate the total price of products.
- Calculate the age of a student.
- Find the total number of students.
- Get the details of a particular student.
- Calculate the average marks of students.

Real-life example

Suppose a university needs to calculate the total marks of a student. Instead of writing the same calculation repeatedly, we can create a function and call it whenever required.

1.2 Why Do We Use Functions?

Functions are useful because they:

1. Reduce repeated code.
2. Make SQL queries easier to manage.
3. Improve code reusability.
4. Perform calculations.
5. Return single values or multiple rows.
6. Help organize database operations.

1.3 How Does a Function Work?

- Call the Function
- Provide required input values
- Execute Function
- Perform the required operations
- Return Result
- Return a value or a table

1.4 Function vs. Normal SQL Query

Normal SQL Query	PostgreSQL Function
Written when needed	Created once
Code may be repeated	Code can be reused
Executes a query	Executes predefined logic

Normal SQL Query	PostgreSQL Function
Usually written directly	Called by its name

1.5 Example

Suppose we want to calculate the total price of a product.

```
SELECT 500 * 3 AS total_price;
```

Output:

total_price
1500

We can create a function to perform the same calculation and reuse it whenever needed.

2. Creating a Simple Function

2.1 What is a Simple Function?

A simple function is a function that performs a specific task and returns a result.

For example, we can create a function that adds two numbers.

2.2 Syntax to Create a Function

```
CREATE FUNCTION function_name()
RETURNS data_type
LANGUAGE plpgsql
AS $$
BEGIN
    -- SQL statements
    RETURN value;
END;
$$;
```

2.3 Understanding the Syntax

Keyword	Description
CREATE FUNCTION	Creates a new function
function_name	Name of the function
RETURNS	Specifies the return data type
LANGUAGE plpgsql	Specifies the programming language
AS $$	Starts the function body

Keyword	Description
BEGIN	Starts the execution block
RETURN	Returns a result
END	Ends the execution block
$$	Ends the function body

Important: `plpgsql` is PostgreSQL's procedural language. It allows us to write variables, conditions, loops, and SQL statements inside functions.

2.4 Example: Create a Simple Function

Let's create a function that returns a welcome message.

```
CREATE OR REPLACE FUNCTION get_welcome_message()
RETURNS TEXT
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN 'Welcome to PostgreSQL!';
END;
$$;
```

Understanding the code:

- `CREATE OR REPLACE FUNCTION` creates a new function or replaces an existing function with the same name and argument types.
- `get_welcome_message()` is the function name.
- `RETURNS TEXT` means the function returns text.
- `BEGIN` starts the function body.
- `RETURN` sends the result back.
- `END` finishes the function.

2.5 Calling a Function

After creating a function, we can call it using `SELECT`.

```
SELECT get_welcome_message();
```

Output:

get_welcome_message
Welcome to PostgreSQL!

2.6 Another Example: Calculate Square

```
CREATE OR REPLACE FUNCTION calculate_square()
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN 5 * 5;
END;
$$;
```

Call the function:

```
SELECT calculate_square();
```

Output:

calculate_square
25

Remember: A function must be called to execute its logic and get its returned result.

3. Function Parameters

3.1 What are Function Parameters?

Parameters allow us to pass values to a function.

They make functions more flexible because we can use different input values without changing the function code.

Real-life example

Suppose a university wants to calculate the total fee of a student.

Different students may have different semester fees. Instead of creating a separate function for every student, we can create one function that accepts the fee as a parameter.

3.2 Syntax

```
CREATE OR REPLACE FUNCTION function_name(
    parameter_name data_type
)
RETURNS data_type
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN value;
END;
$$;
```

3.3 Example: Function with One Parameter

Let's create a function that calculates the square of a number.

```
CREATE OR REPLACE FUNCTION calculate_square(
    p_number INTEGER
)
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_number * p_number;
END;
$$;
```

Understanding the parameter:

- `p_number` is the parameter name.
- `INTEGER` is its data type.
- The user provides a number when calling the function.
- The function multiplies the number by itself.

3.4 Calling the Function

```
SELECT calculate_square(5);
```

Output:

calculate_square
25

Try another number:

```
SELECT calculate_square(8);
```

Output:

calculate_square
64

3.5 Example: Function with Two Parameters

Let's create a function that adds two numbers.

```
CREATE OR REPLACE FUNCTION add_numbers(
    p_num1 INTEGER,
```

```
    p_num2 INTEGER
)
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_num1 + p_num2;
END;
$$;
```

Call the function:

```
SELECT add_numbers(10, 20);
```

Output:

add_numbers
30

3.6 Function with Multiple Parameters

We can pass multiple parameters to a function.

For example, calculate the total price using quantity and unit price.

```
CREATE OR REPLACE FUNCTION calculate_total_price(
    p_quantity INTEGER,
    p_unit_price NUMERIC
)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_quantity * p_unit_price;
END;
$$;
```

Call the function:

```
SELECT calculate_total_price(5, 250.50);
```

Output:

calculate_total_price
1252.50

3.7 Named Parameters

PostgreSQL also allows us to pass parameters by name.

```
SELECT calculate_total_price(
    p_unit_price => 250.50,
    p_quantity => 5
);
```

This makes function calls easier to understand, especially when there are several parameters.

Important: Parameters allow us to reuse the same function with different input values.

4. Returning Values from Functions

4.1 What Does Returning a Value Mean?

A function can perform calculations or operations and return a result to the user.

For example:

- Return a student's total marks.
- Return the average marks.
- Return the total number of students.
- Return the name of a student.

The `RETURN` statement is used to send a result back from a function.

4.2 Common Return Data Types

Data Type	Description	Example
INTEGER	Whole numbers	100
NUMERIC	Decimal numbers	250.50
TEXT	Text values	'Ali'
BOOLEAN	True or false	TRUE
DATE	Date values	'2026-11-20'

4.3 Example: Return Student Marks

Suppose we want to create a function that returns the total marks of a student.

```
CREATE OR REPLACE FUNCTION calculate_total_marks(
    p_subject1 INTEGER,
    p_subject2 INTEGER,
    p_subject3 INTEGER
)
RETURNS INTEGER
```

```
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_subject1 + p_subject2 + p_subject3;
END;
$$;
```

Call the function:

```
SELECT calculate_total_marks(80, 75, 90);
```

Output:

calculate_total_marks
245

4.4 Example: Return Average Marks

```
CREATE OR REPLACE FUNCTION calculate_average(
    p_total_marks NUMERIC,
    p_total_subjects INTEGER
)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_total_marks / p_total_subjects;
END;
$$;
```

Call the function:

```
SELECT calculate_average(300, 3);
```

Output:

calculate_average
100.00

4.5 Example: Return TRUE or FALSE

A function can also return a Boolean value.

Let's create a function that checks whether a student has passed or failed.

Assume that 50 marks are required to pass.

```
CREATE OR REPLACE FUNCTION check_pass(
    p_marks INTEGER
)
RETURNS BOOLEAN
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_marks >= 50;
END;
$$;
```

Call the function:

```
SELECT check_pass(75);
```

Output:

check_pass
true

Try another example:

```
SELECT check_pass(40);
```

Output:

check_pass
false

4.6. Returning Values from Table Data

Functions can also retrieve information from database tables.

Suppose we have a table named `students`.

```
CREATE TABLE students (
    student_id INTEGER PRIMARY KEY,
    student_name VARCHAR(100),
    marks INTEGER
);
```

Insert sample data:

```
INSERT INTO students VALUES
(1, 'Ali', 85),
```

```
(2, 'Ahmed', 45),
(3, 'Sara', 90);
```

Create a function to return a student's marks using their ID.

```
CREATE OR REPLACE FUNCTION get_student_marks(
    p_student_id INTEGER
)
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_marks INTEGER;
BEGIN
    SELECT marks
    INTO v_marks
    FROM students
    WHERE student_id = p_student_id;

    RETURN v_marks;
END;
$$;
```

Call the function:

```
SELECT get_student_marks(1);
```

Output:

get_student_marks
85

New concept:

`DECLARE` is used to define variables inside a PL/pgSQL function.

`SELECT ... INTO` stores the result of a query in a variable.

5. Functions Returning Tables

5.1 What is a Table-Returning Function?

A table-returning function is a function that returns multiple rows and columns instead of a single value.

For example, a university may need a function that returns:

- All students.
- Students who passed.
- Students of a particular department.

- Students whose marks are above a specific value.

Instead of returning one value, the function returns a complete result set.

5.2 Difference Between Single Value and Table Return

Single Value Function	Table-Returning Function
Returns one value	Returns multiple rows
Uses `RETURNS INTEGER`, `TEXT`, etc.	Uses `RETURNS TABLE`
Example: Student marks	Example: Student records
Called using `SELECT function()`	Can be queried using `SELECT * FROM function()`

5.3 Syntax

```
CREATE OR REPLACE FUNCTION function_name()
RETURNS TABLE (
    column1 data_type,
    column2 data_type
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT column1, column2
    FROM table_name;
END;
$$;
```

Important keywords:

Keyword	Description
RETURNS TABLE	Defines the columns returned by the function
RETURN QUERY	Executes a query and returns its result rows
SELECT	Retrieves records from a table

5.4 Example: Return All Students

We will use the previously created `students` table.

Create the function:

```
CREATE OR REPLACE FUNCTION get_all_students()
RETURNS TABLE (
    student_id INTEGER,
    student_name VARCHAR(100),
```

```
    marks INTEGER
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        s.student_id,
        s.student_name,
        s.marks
    FROM students AS s;
END;
$$;
```

5.5. Calling the Function

```
SELECT * FROM get_all_students();
```

Output:

student_id	student_name	marks
1	Ali	85
2	Ahmed	45
3	Sara	90

5.6. Example: Return Students Who Passed

Let's create a function that returns students who obtained 50 or more marks.

```
CREATE OR REPLACE FUNCTION get_passed_students()
RETURNS TABLE (
    student_id INTEGER,
    student_name VARCHAR(100),
    marks INTEGER
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        s.student_id,
        s.student_name,
        s.marks
    FROM students AS s
    WHERE s.marks >= 50;
END;
$$;
```

Call the function:

```
SELECT * FROM get_passed_students();
```

Output:

student_id	student_name	marks
1	Ali	85
3	Sara	90

5.7. Example: Table-Returning Function with Parameters

We can also pass parameters to table-returning functions.

Let's create a function that returns students who obtained marks greater than or equal to a given value.

```
CREATE OR REPLACE FUNCTION get_students_by_marks(
    p_min_marks INTEGER
)
RETURNS TABLE (
    student_id INTEGER,
    student_name VARCHAR(100),
    marks INTEGER
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        s.student_id,
        s.student_name,
        s.marks
    FROM students AS s
    WHERE s.marks >= p_min_marks;
END;
$$;
```

Call the function:

```
SELECT * FROM get_students_by_marks(80);
```

Output:

student_id	student_name	marks
1	Ali	85
3	Sara	90

Try another value:

```
SELECT * FROM get_students_by_marks(50);
```

Output:

student_id	student_name	marks
1	Ali	85
3	Sara	90

Remember: The function returns only those students whose marks meet the minimum marks provided as a parameter.

6. Important PostgreSQL Function Commands

Command	Purpose
`CREATE FUNCTION`	Creates a function
`CREATE OR REPLACE FUNCTION`	Creates or replaces a function
`SELECT function_name()`	Calls a function
`RETURNS`	Defines the return data type
`RETURNS TABLE`	Defines a table result
`RETURN`	Returns a single result
`RETURN QUERY`	Returns rows from a query
`DROP FUNCTION`	Deletes a function

7. How to Delete a Function

Syntax:

```
DROP FUNCTION function_name();
```

Example:

```
DROP FUNCTION get_welcome_message();
```

For a function with parameters:

```
DROP FUNCTION calculate_square(INTEGER);
```

8. Examples

Example: Normal SQL Query

Find all films with a rental rate greater than 3.

```
SELECT
    film_id,
    title,
    rental_rate
FROM film
WHERE rental_rate > 3;
```

We can create functions to reuse similar queries and calculations.

Example 1: Return a Welcome Message

Create a function that returns a welcome message for the DVD Rental database.

```
CREATE OR REPLACE FUNCTION get_welcome_message()
RETURNS TEXT
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN 'Welcome to DVD Rental Database!';
END;
$$;
```

Call the function:

```
SELECT get_welcome_message();
```

Expected output:

get_welcome_message
Welcome to DVD Rental Database!

Example 2: Count Total Films

Create a function that returns the total number of films in the database.

```
CREATE OR REPLACE FUNCTION get_total_films()
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_total INTEGER;
```

```
BEGIN
    SELECT COUNT(*)
    INTO v_total
    FROM film;

    RETURN v_total;
END;
$$;
```

Call the function:

```
SELECT get_total_films();
```

Explanation:

- `COUNT(*)` counts all film records.
- `INTO v_total` stores the count in a variable.
- `RETURN v_total` returns the total number of films.

Example 3: Get Customer Details by Customer ID

Create a function that accepts a customer ID and returns the customer's full name.

```
CREATE OR REPLACE FUNCTION get_customer_name(
    p_customer_id INTEGER
)
RETURNS TEXT
LANGUAGE plpgsql
AS $$
DECLARE
    v_customer_name TEXT;
BEGIN
    SELECT
        first_name || ' ' || last_name
    INTO v_customer_name
    FROM customer
    WHERE customer_id = p_customer_id;

    RETURN v_customer_name;
END;
$$;
```

Call the function:

```
SELECT get_customer_name(1);
```

Example 4: Calculate Total Payments of a Customer

Create a function that accepts a customer ID and returns the total amount paid by that customer.

```
CREATE OR REPLACE FUNCTION get_customer_total_payment(
    p_customer_id INTEGER
)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    v_total_payment NUMERIC;
BEGIN
    SELECT COALESCE(SUM(amount), 0)
    INTO v_total_payment
    FROM payment
    WHERE customer_id = p_customer_id;

    RETURN v_total_payment;
END;
$$;
```

Call the function:

```
SELECT get_customer_total_payment(1);
```

Explanation:

- `p_customer_id` receives the customer ID.
- `SUM(amount)` calculates total payments.
- `COALESCE()` returns zero if no payment records are found.
- The function returns the total payment.

Example 5: Calculate Film Rental Amount

Create a function that calculates the rental amount based on the rental rate and number of days.

```
CREATE OR REPLACE FUNCTION calculate_rental_amount(
    p_rental_rate NUMERIC,
    p_days INTEGER
)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN p_rental_rate * p_days;
END;
$$;
```

Call the function:

```
SELECT calculate_rental_amount(2.99, 5);
```

Expected output:

calculate_rental_amount
14.95

Example 6: Return Total Rentals of a Customer

Create a function that returns the total number of rentals made by a customer.

```
CREATE OR REPLACE FUNCTION get_customer_rental_count(
    p_customer_id INTEGER
)
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_total_rentals INTEGER;
BEGIN
    SELECT COUNT(*)
    INTO v_total_rentals
    FROM rental
    WHERE customer_id = p_customer_id;

    RETURN v_total_rentals;
END;
$$;
```

Call the function:

```
SELECT get_customer_rental_count(1);
```

Example 3: Return Film Rental Duration

Create a function that returns the rental duration of a film in days.

```
CREATE OR REPLACE FUNCTION get_film_rental_duration(
    p_film_id INTEGER
)
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
DECLARE
    v_duration INTEGER;
BEGIN
    SELECT rental_duration
```

```
    INTO v_duration
    FROM film
    WHERE film_id = p_film_id;

    RETURN v_duration;
END;
$$;
```

Call the function:

```
SELECT get_film_rental_duration(1);
```

Example 4: Check Whether a Film Is Available for Rental

Create a function that checks whether at least one copy of a film is currently available.

```
CREATE OR REPLACE FUNCTION is_film_available(
    p_film_id INTEGER
)
RETURNS BOOLEAN
LANGUAGE plpgsql
AS $$
DECLARE
    v_available BOOLEAN;
BEGIN
    SELECT EXISTS (
        SELECT 1
        FROM inventory AS i
        WHERE i.film_id = p_film_id
        AND NOT EXISTS (
            SELECT 1
            FROM rental AS r
            WHERE r.inventory_id = i.inventory_id
            AND r.return_date IS NULL
        )
    )
    INTO v_available;

    RETURN v_available;
END;
$$;
```

Call the function:

```
SELECT is_film_available(1);
```

Output: `true` or `false`.

Explanation:

- `inventory` contains copies of films available at stores.
- `rental` contains rental transactions.
- `return_date IS NULL` identifies a copy that has not yet been returned.
- `EXISTS` checks whether at least one available copy exists.

Example 5: Get All Films

Create a function that returns film details.

```
CREATE OR REPLACE FUNCTION get_all_films()
RETURNS TABLE (
    film_id INTEGER,
    title VARCHAR,
    rental_rate NUMERIC,
    rating MPAA_RATING
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        f.film_id,
        f.title,
        f.rental_rate,
        f.rating
    FROM film AS f;
END;
$$;
```

Call the function:

```
SELECT * FROM get_all_films();
```

Example 6: Get Customer Payment History

Create a function that accepts a customer ID and returns all payment records for that customer.

```
CREATE OR REPLACE FUNCTION get_customer_payments(
    p_customer_id INTEGER
)
RETURNS TABLE (
    payment_id INTEGER,
    payment_date TIMESTAMP WITH TIME ZONE,
    amount NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
```

```
    SELECT
        p.payment_id,
        p.payment_date,
        p.amount
    FROM payment AS p
    WHERE p.customer_id = p_customer_id
    ORDER BY p.payment_date DESC;
END;
$$;
```

Call the function:

```
SELECT * FROM get_customer_payments(1);
```

Example 7: Get Films by Category

Create a function that accepts a category ID and returns all films belonging to that category.

```
CREATE OR REPLACE FUNCTION get_films_by_category(
    p_category_id INTEGER
)
RETURNS TABLE (
    film_id INTEGER,
    title VARCHAR,
    rental_rate NUMERIC,
    category_name VARCHAR
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        f.film_id,
        f.title,
        f.rental_rate,
        c.name
    FROM film AS f
    INNER JOIN film_category AS fc
        ON f.film_id = fc.film_id
    INNER JOIN category AS c
        ON fc.category_id = c.category_id
    WHERE c.category_id = p_category_id
    ORDER BY f.title;
END;
$$;
```

Call the function:

```
SELECT * FROM get_films_by_category(1);
```

Example 8: Get Customer Rental History

Create a function that returns the rental history of a particular customer.

```
CREATE OR REPLACE FUNCTION get_customer_rental_history(
    p_customer_id INTEGER
)
RETURNS TABLE (
    rental_id INTEGER,
    customer_name TEXT,
    film_title VARCHAR,
    rental_date TIMESTAMP WITH TIME ZONE,
    return_date TIMESTAMP WITH TIME ZONE
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        r.rental_id,
        c.first_name || ' ' || c.last_name,
        f.title,
        r.rental_date,
        r.return_date
    FROM rental AS r
    INNER JOIN customer AS c
        ON r.customer_id = c.customer_id
    INNER JOIN inventory AS i
        ON r.inventory_id = i.inventory_id
    INNER JOIN film AS f
        ON i.film_id = f.film_id
    WHERE r.customer_id = p_customer_id
    ORDER BY r.rental_date DESC;
END;
$$;
```

Call the function:

```
SELECT * FROM get_customer_rental_history(1);
```

9. Practice Exercises

Students should complete these exercises using PostgreSQL.

Exercise 1: Create a Simple Function

Create a function named `get_university_name()` that returns:

`Ghazi University, Dera Ghazi Khan`

Exercise 2: Function with Parameters

Create a function named `multiply_numbers()` that accepts two integers and returns their product.

Exercise 3: Calculate Student Percentage

Create a function named `calculate_percentage()` that accepts obtained marks and total marks and returns the percentage.

Example: 450 obtained marks out of 500 should return 90%.

Exercise 4: Check Even or Odd

Create a function named `check_even_odd()` that accepts an integer and returns whether the number is even or odd.

Exercise 5: Get Student Information

Create a function named `get_student_details()` that accepts a student ID and returns the student's name and marks.

Exercise 6: Return All Students

Create a table-returning function named `get_all_students()` that returns all student records.

Exercise 7: Filter Students by Marks

Create a function named `get_students_by_marks()` that accepts minimum marks and returns all students who obtained equal or higher marks.