What is a Function in Pl/SQL?

A function in MySQL is a reusable SQL code block designed to perform a specific task and return a single value. Functions can simplify complex queries, improve code reusability, and be used in various SQL statements like SELECT, WHERE, and others.

Pl/SQL function is similar to the PL/SQL Procedure.

The main difference between procedure and function is , function must return the value, and on other hand procedure may or may not return the value.

CREATE DEFINER=`root`@`localhost` FUNCTION `FunName`( parameters ………… ) RETURNS int

    READS SQL DATA

    DETERMINISTIC

BEGIN

           - - queries

END

READS SQL DATA :

This attribute indicates that the function reads data from SQL tables but does not modify any data.

DETERMINISTIC  :

This attribute indicates that the function always produces the same output for the same set of input parameters.

This property is useful for MySQL optimizers to make certain optimizations based on the assumption that the function result is predictable and won’t be changed during the execution of the query.

BEGIN :  Indicate the starting of Function.

END : Indicate the ending of Function.

To Execute the function :

Syntax :

SELECT Function\_Name() AS Alise\_Name;

AS is optional.

## Steps to Create and Use Functions

1. Create a Database and Table

-- Create and use the 'Company' database

CREATE DATABASE IF NOT EXISTS Company;

USE Company;

-- Create the 'employee' table

CREATE TABLE employee (

    id INT,

    name VARCHAR(30) NOT NULL,

    age INT,

    salary INT

);

-- Insert data into the 'employee' table

INSERT INTO employee VALUES

    (1, 'Diviksha', 28, 50000),

    (2, 'Mayank', 35, 75000),

    (3, 'Lalit', 40, 65000),

    (4, 'Kshitija', 25, 40000),

    (5, 'Prabhakar', 30, 90000);

-- Display the contents of the 'employee' table

SELECT \* FROM employee;

2. Create Functions

2.1. Create the AverageSalary Function

DELIMITER //

CREATE DEFINER=`root`@`localhost` FUNCTION `AverageSalary`() RETURNS FLOAT

    READS SQL DATA

    DETERMINISTIC

BEGIN

    DECLARE avg\_salary FLOAT;

    SELECT AVG(salary) INTO avg\_salary FROM employee;

    RETURN avg\_salary;

END; //

DELIMITER ;

## Explanation of Functions

AverageSalary Function

Purpose: Calculates the average salary in the employee table.

Steps:

Declares a variable avg\_salary to store the result.

Uses AVG(salary) to compute the average salary.

Returns the value of avg\_salary.

2.2. Create the EmployeesAboveSalary Function

DELIMITER //

CREATE DEFINER=`root`@`localhost` FUNCTION `EmployeesAboveSalary`(threshold INT) RETURNS INT

    READS SQL DATA

    DETERMINISTIC

BEGIN

    DECLARE emp\_count INT;

    SELECT COUNT(\*) INTO emp\_count FROM employee WHERE salary > threshold;

    RETURN emp\_count;

END; //

DELIMITER ;

3. Call the Functions

3.1. Call the AverageSalary Function

SELECT AverageSalary() AS Average\_Salary;

3.2. Call the EmployeesAboveSalary Function

SELECT EmployeesAboveSalary(60000) AS Employees\_Above\_60000;

4. Output

Updated Table: employee

Function Results

Average Salary:  
Average\_Salary

--------------

64000

Employees Above Salary 60000:  
Employees\_Above\_60000

----------------------

3

## Explanation of Functions

EmployeesAboveSalary Function

Purpose: Counts the number of employees earning above the given threshold.

Steps:

Accepts a parameter threshold to dynamically filter salaries.

Declares a variable emp\_count to store the count.

Uses COUNT(\*) with a WHERE condition to count employees above the threshold.

Returns the value of emp\_count.

## Difference Between Stored Procedure and Functions

## Stored Procedure in MySQL

A Stored Procedure is a precompiled collection of SQL statements stored in the database. It allows you to execute a sequence of queries and logic multiple times without writing the same SQL code repeatedly. Stored procedures are particularly useful for encapsulating business logic and improving performance by reducing network traffic between applications and the database.

Table Creation

Example Table: employee

The employee table contains employee details such as ID, Name, Age, and Salary.

-- Switch to the 'World' database

USE World;

-- Drop the 'employee' table if it already exists

DROP TABLE IF EXISTS employee;

-- Create the 'employee' table

CREATE TABLE employee (

    id INT,

    name VARCHAR(20),

    age INT,

    salary INT

);

-- Insert data into the 'employee' table

INSERT INTO employee (id, name, age, salary) VALUES

    (1, 'Aishwarya', 30, 70000),

    (2, 'Shraddha', 30, 80000),

    (3, 'Alia', 40, 90000),

    (4, 'Kriti', 50, 10000);

Stored Procedure Examples

Example 1: Finding the Maximum Salary

-- Drop the 'GetMaxSalary' procedure if it already exists

DROP PROCEDURE IF EXISTS GetMaxSalary;

-- Create a stored procedure to find the maximum salary

DELIMITER //

CREATE PROCEDURE GetMaxSalary()

BEGIN

    DECLARE max\_salary INT;

    SELECT MAX(salary) INTO max\_salary FROM employee;

    SELECT max\_salary AS Maximum\_Salary;

END //

DELIMITER ;

Example 2: Calculating the Sum of All Salaries

-- Drop the 'GetSumOfSalaries' procedure if it already exists

DROP PROCEDURE IF EXISTS GetSumOfSalaries;

-- Create a stored procedure to calculate the total salary

DELIMITER //

CREATE PROCEDURE GetSumOfSalaries()

BEGIN

    DECLARE total\_salary INT;

    SELECT SUM(salary) INTO total\_salary FROM employee;

    SELECT total\_salary AS Total\_Salary;

END //

DELIMITER ;

Explanation of the Code

Table Creation:

The employee table is created with four columns: id, name, age, and salary. Data is inserted to demonstrate the stored procedures.

Stored Procedure: GetMaxSalary:

The procedure declares a variable max\_salary.

It uses the SELECT MAX(salary) query to calculate the highest salary in the employee table and stores it in the variable.

The result is displayed using a SELECT statement.

Stored Procedure: GetSumOfSalaries:

The procedure declares a variable total\_salary.

It calculates the total salary using the SUM(salary) query and stores it in the variable.

The result is displayed using a SELECT statement.

DELIMITER:

The DELIMITER // command changes the delimiter to avoid conflicts with semicolons (;) used in the procedure body.

Execution:

After creating the procedures, you can execute them using the CALL statement.

Execution and Output

Execute Procedures

-- Call the 'GetMaxSalary' procedure

CALL GetMaxSalary();

-- Call the 'GetSumOfSalaries' procedure

CALL GetSumOfSalaries();

Output

Output for CALL GetMaxSalary();:  
  
+----------------+

| Maximum\_Salary |

+----------------+

|         90000  |

+----------------+

Output for CALL GetSumOfSalaries();:  
  
+--------------+

| Total\_Salary |

+--------------+

|       250000 |

+--------------+

Benefits of Stored Procedures

Reusability: Once created, a stored procedure can be used multiple times.

Reduced Network Traffic: Only procedure calls are sent over the network, not the actual SQL statements.

Encapsulation: Business logic is encapsulated within the database.

Security: Access can be controlled through database permissions.

## Trigger in MySQL

Triggers in MySQL

What are Triggers?  
A trigger is a set of SQL statements automatically executed in response to a specific event on a particular table. Triggers can be executed before or after an insert, update, or delete operation.

Triggers are useful for maintaining integrity, enforcing business rules, auditing, and automating processes in a database.

Database and Table Creation

1. Creating the Triggers\_Example Database and employee Table

-- Create the database

CREATE DATABASE Triggers\_Example;

-- Switch to the database

USE Triggers\_Example;

-- Drop the 'employee' table if it exists

DROP TABLE IF EXISTS employee;

-- Create the 'employee' table

CREATE TABLE employee (

    id INT,

    name VARCHAR(45) NOT NULL,

    occupation VARCHAR(35) NOT NULL,

    working\_date DATE,

    working\_hours INT

);

-- Insert initial data into the 'employee' table

INSERT INTO employee VALUES

    (1, 'Guru', 'Scientist', '2024-10-04', 4),

    (2, 'Tony', 'Business', '2024-10-04', 5),

    (3, 'Nami', 'Painter', '2024-10-04', 6),

    (4, 'Marco', 'Doctor', '2024-10-04', 9),

    (5, 'Robin', 'Singer', '2024-10-04', 10),

    (6, 'Nil', 'Engineer', '2024-10-04', 6);

Trigger Examples

Example 1: BEFORE INSERT Trigger

Ensures working\_hours is not negative before inserting data.

-- Define the trigger to validate 'working\_hours' before insert

DELIMITER //

CREATE TRIGGER before\_insert\_employee

BEFORE INSERT ON employee

FOR EACH ROW

BEGIN

    IF NEW.working\_hours < 0 THEN

        SET NEW.working\_hours = 0;

    END IF;

END; //

DELIMITER ;

-- Insert a row with negative working\_hours

INSERT INTO employee VALUES (7, 'Sneha', 'Cricketer', '2024-11-04', -2);

-- Verify the data

SELECT \* FROM employee;

Output for SELECT:

Example 2: AFTER INSERT Trigger

Keeps track of the total number of employees in a separate table.

-- Create a new table to store the total number of employees

CREATE TABLE employee\_count (

    total\_employees INT DEFAULT 0

);

-- Insert initial row in 'employee\_count'

INSERT INTO employee\_count (total\_employees) VALUES (0);

-- Define the trigger to update the count after an insert

DELIMITER //

CREATE TRIGGER after\_insert\_employee

AFTER INSERT ON employee

FOR EACH ROW

BEGIN

    UPDATE employee\_count SET total\_employees = total\_employees + 1;

END; //

DELIMITER ;

-- Insert new data to trigger the count update

INSERT INTO employee VALUES (8, 'Riya', 'Developer', '2024-03-12', 9);

-- Verify the data and the count

SELECT \* FROM employee;

SELECT \* FROM employee\_count;

Output for employee\_count:

Example 3: BEFORE INSERT Trigger with Conditional Modification

Changes the occupation to 'Dancer' if it is 'Scientist' during insertion.

-- Define a trigger to modify occupation before insert

DELIMITER //

CREATE TRIGGER before\_insert\_role3

BEFORE INSERT ON employee

FOR EACH ROW

BEGIN

    IF NEW.occupation = 'Scientist' THEN

        SET NEW.occupation = 'Dancer';

    END IF;

END; //

DELIMITER ;

-- Insert a row with occupation as 'Scientist'

INSERT INTO employee VALUES (9, 'Priya', 'Scientist', '2024-12-12', 8);

-- Verify the modified data

SELECT \* FROM employee;

Output for SELECT:

Managing Triggers

List All Triggers:   
SHOW TRIGGERS;

Drop a Trigger:  
DROP TRIGGER Triggers\_Example.before\_insert\_employee;

Explanation of Code

BEFORE INSERT Trigger:

Ensures working\_hours is non-negative.

Validates and modifies the data before insertion into the table.

AFTER INSERT Trigger:

Automatically updates the employee\_count table whenever a new row is added to the employee table.

Useful for keeping derived data consistent.

Conditional BEFORE INSERT Trigger:

Checks the occupation value during insertion and modifies it based on business rules.

Benefits of Triggers

Data Validation: Enforces rules and constraints automatically.

Automation: Reduces manual tasks like updating summary tables.

Audit and Logging: Tracks changes or events in the database.

## Cursor in PL/SQL

What is a Cursor in MySQL?

A cursor in MySQL is a database object used to retrieve data row-by-row from a result set. It provides a mechanism to perform operations on individual rows when a query retrieves multiple rows.

Features of Cursors:

Cursors are used in stored procedures, functions, or triggers when row-by-row processing is needed.

A cursor must be declared and opened to retrieve the data.

It must be explicitly closed after its use.

Steps to Use a Cursor

Declare a cursor to define the result set.

Open the cursor to initialize it and prepare to fetch rows.

Fetch rows from the cursor into variables.

Perform operations using the fetched data.

Close the cursor after processing all rows.

MySQL Cursor Syntax

DECLARE cursor\_name CURSOR FOR query;

OPEN cursor\_name;

FETCH cursor\_name INTO variable\_list;

CLOSE cursor\_name;

Example: Using a Cursor

Scenario: Calculate the total salary of employees whose age is greater than 30, using a cursor.

-- Use the 'Company' database

USE Company;

-- Create the 'employee' table

CREATE TABLE employee (

    id INT,

    name VARCHAR(30) NOT NULL,

    age INT,

    salary INT

);

-- Insert data into the 'employee' table

INSERT INTO employee VALUES

    (1, 'Diviksha', 28, 50000),

    (2, 'Mayank', 35, 75000),

    (3, 'Lalit', 40, 65000),

    (4, 'Kshitija', 25, 40000),

    (5, 'Prabhakar', 30, 90000);

-- Create a stored procedure to calculate the total salary using a cursor

DELIMITER //

CREATE PROCEDURE CalculateTotalSalary()

BEGIN

    DECLARE done INT DEFAULT 0;              -- Flag to indicate the end of the cursor

    DECLARE emp\_salary INT;                  -- Variable to hold the fetched salary

    DECLARE total\_salary INT DEFAULT 0;      -- Variable to store the total salary

    DECLARE cur CURSOR FOR SELECT salary FROM employee WHERE age > 30;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    -- Open the cursor

    OPEN cur;

    -- Loop through each row in the result set

    fetch\_loop: LOOP

        FETCH cur INTO emp\_salary;           -- Fetch the next row

        IF done THEN

            LEAVE fetch\_loop;                -- Exit the loop when all rows are processed

        END IF;

        SET total\_salary = total\_salary + emp\_salary; -- Accumulate the salary

    END LOOP;

    -- Close the cursor

    CLOSE cur;

    -- Display the total salary

    SELECT total\_salary AS Total\_Salary\_Above\_30;

END; //

DELIMITER ;

-- Call the stored procedure

CALL CalculateTotalSalary();

Output

Updated Table: employee

Result

Total\_Salary\_Above\_30

---------------------

140000

Explanation of the Code

Cursor Declaration:

DECLARE cur CURSOR FOR SELECT salary FROM employee WHERE age > 30;  
This cursor retrieves the salary of employees whose age is greater than 30.

Handler Declaration:

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;  
This handler sets the done flag to 1 when no more rows are found.

Cursor Workflow:

OPEN cur: Initializes the cursor.

FETCH cur INTO emp\_salary: Retrieves the salary column value from the current row.

LOOP: Iterates through each row until the done flag is set.

CLOSE cur: Releases the cursor after processing is complete.

Accumulation:

SET total\_salary = total\_salary + emp\_salary;  
Adds the salary of each employee fetched by the cursor to total\_salary.

Output:

The total\_salary is displayed as the result using SELECT.