Pre-Joining Topics

Week 2: MYSQL ADVANCED

Stored Procedures

Trigger:

In MySQL, a trigger is a special type of stored program that automatically executes a set of SQL statements in response to specific events that occur on a database table. These events can be INSERT, UPDATE, or DELETE operations.

Syntax:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
  -- SQL statements to be executed
END;
```

Explanation of syntax elements:

- trigger_name: The unique name given to the trigger.
- BEFORE | AFTER: Specifies whether the trigger executes before or after the triggering event.
- INSERT | UPDATE | DELETE: Specifies the type of DML operation that activates the trigger.
- ON table_name: Specifies the table on which the trigger is defined.
- FOR EACH ROW: Indicates that the trigger will execute for each row affected by the operation.
- BEGIN ... END: Delimits the body of the trigger, containing the SQL statements to be executed.

Stored Procedures:

A stored procedure in MySQL is a set of SQL statements stored on the server that can be executed as a single unit. It helps reduce repetitive code, improve performance, and secure access by encapsulating logic inside the database.

In MySQL, a stored procedure is a prepared SQL code that you can save and reuse over and over again. It is a group of one or more SQL statements stored in the database and can be executed as a single unit.

Syntax

To create a stored procedure, you need to change the delimiter (so it doesn't conflict with ; inside the body).

```
DELIMITER $$


CREATE PROCEDURE procedure_name (parameter_list)

BEGIN

  -- SQL statements

END$$


DELIMITER ;
```

To execute it:

```
CALL procedure_name(arguments);
```

Parameters

Stored procedures can take parameters to make them dynamic.

- **IN**: value is passed into the procedure.
- **OUT**: value is returned from the procedure.
- **INOUT**: can pass in a value and return it after modification.

```
CREATE PROCEDURE sample_proc(IN in_param INT, OUT out_param INT, INOUT
inout_param INT)

BEGIN
```

```
  SET out_param = in_param * 2;
  SET inout_param = inout_param + 100;
END;
```

Variables

Variables can be declared inside a procedure:

```
DECLARE var_name datatype DEFAULT value;
SET var_name = new_value;
```

They are useful for intermediate calculations or conditional logic.

Control Structures

Stored procedures support programming-like flow controls:

- **IF…ELSE** for conditional branching
- **CASE** for multiple-choice conditions
- **LOOP**, **WHILE**, and **REPEAT** for iterations

Example:

```
IF total > 1000 THEN
  SET status = 'VIP';
ELSE
  SET status = 'Regular';
END IF;
```

Cursors

Cursors let you process query results row by row. Useful when set-based operations are not enough.

```
DECLARE done INT DEFAULT 0;
DECLARE emp_name VARCHAR(50);
DECLARE cur CURSOR FOR SELECT name FROM employees;
```

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;


OPEN cur;
read_loop: LOOP
  FETCH cur INTO emp_name;
  IF done THEN
    LEAVE read_loop;
  END IF;
  -- use emp_name
END LOOP;
CLOSE cur;
```

Error Handling

Handlers allow you to manage errors gracefully.

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
  -- actions when error occurs
END;
```

Two types are common:

- EXIT → stop execution when error happens
- CONTINUE → continue execution after handling

Advantages

Stored procedures improve performance because they are precompiled and cached. They allow code reuse and centralize business rules in one place. Security is improved since you can grant EXECUTE privilege without giving table access. They also reduce network traffic by grouping multiple SQL statements into a single call.

Limitations

Debugging stored procedures can be difficult. They may put extra load on the database server if they are too complex. Portability across different RDBMS is limited. Complex business logic is often better placed in the application layer rather than inside the database.

Best Practices

Keep procedures modular and readable with clear names. Favor set-based operations instead of cursors whenever possible. Handle exceptions properly and use transactions for data consistency. Restrict user access by granting only EXECUTE permission on procedures.

Example Use Case

A procedure to calculate the total sales for a customer:

```
DELIMITER $$

CREATE PROCEDURE GetCustomerSales(IN cust_id INT, OUT total_sales DECIMAL(10,2))
BEGIN
  SELECT SUM(amount) INTO total_sales
  FROM orders
  WHERE customer_id = cust_id;
END$$

DELIMITER ;
```

Call it like this:

```
CALL GetCustomerSales(101, @sales);
SELECT @sales;
```

```
-- Remove the procedure
DROP PROCEDURE get_car_count;
```

Stored Functions:

A stored function is a specialized type of stored program designed to return a single value. Typically, you use stored functions to encapsulate common formulas or business rules, making them reusable across SQL statements or other stored programs.

Unlike a stored procedure, you can use a stored function in SQL statements wherever you use an expression. This enhances the readability and maintainability of the procedural code.

To create a stored function, you use the CREATE FUNCTION statement. The following illustrates the basic syntax for creating a new stored function:

```
DELIMITER $$


CREATE FUNCTION function_name(
   param1,
   param2,…
)
RETURNS datatype
[NOT] DETERMINISTIC
BEGIN
 -- statements
END $$


DELIMITER ;Code language: SQL (Structured Query Language) (sql)
```

In this syntax:

First, specify the name of the stored function that you want to create after CREATE FUNCTION keywords.

Second, list all parameters of the stored function inside the parentheses followed by the function name.

By default, stored functions consider all parameters as IN parameters. You cannot specify IN , OUT or INOUT modifiers to parameters

Third, specify the data type of the return value in the RETURNS statement, which can be any valid MySQL data types.

Fourth, determine whether a function is deterministic or not using the DETERMINISTIC keyword.

A deterministic function always returns the same result for the same input parameters, while a non-deterministic function produces different results for the same input parameters.

If you don't use DETERMINISTIC or NOT DETERMINISTIC, MySQL defaults to the NOT DETERMINISTIC option.

Finally, write the code in the body of the stored function in the BEGIN...END block.

Inside the body section, you need to include at least one RETURN statement. The RETURN statement sends a value to the calling programs.

Upon reaching the RETURN statement, the stored function terminates the execution of the stored function immediately.