Stored Procedures and Functions: Implementation Guide

Executive Summary

This document provides a comprehensive guide to creating reusable SQL blocks through stored procedures, user-defined functions, and triggers. These database objects encapsulate complex logic, improve security, enhance performance, and enable code reuse across applications.

1. Introduction to Reusable SQL Blocks

What Are They?

Stored Procedures are pre-compiled SQL statements that perform specific tasks. They accept parameters, execute SQL logic, and can return multiple values through output parameters.

User-Defined Functions are similar but more restrictive—they accept input parameters and MUST return exactly one value. They are typically used for calculations and transformations within SELECT statements or WHERE clauses.

Triggers are special database objects that automatically execute (or "fire") in response to specific events (INSERT, UPDATE, DELETE) on a particular table.

Why Use Them?

- **Security**: Restrict direct table access; users execute procedures instead
- **Performance**: Compiled once, executed many times
- **Reusability**: Write once, use everywhere
- **Reduced Network Traffic**: Send fewer bytes between application and database
- **Data Consistency**: Enforce business rules at database level
- **Maintainability**: Centralize logic; easier to update

2. MySQL Stored Procedures

Syntax and Structure

```
DELIMITER //

CREATE PROCEDURE procedure_name(
    [IN|OUT|INOUT] parameter_name datatype,
    ...
)
```

```
BEGIN
    -- Declarations
    DECLARE variable_name datatype DEFAULT value;

    -- SQL statements
    -- Control flow (IF, CASE, LOOP)

    -- Return via OUT parameters
    SET parameter_name = value;
END //

DELIMITER ;
```

Parameter Types Explained

Parameter Type	Behavior	Example
IN (default)	Read-only input; original value unchanged	`IN age INT`
OUT	Write-only output; returns value to caller	`OUT result VARCHAR(50)`
INOUT	Read/write; input can be modified and returned	`INOUT balance DECIMAL(10,2)`

Execution

```
-- For procedures with no output
CALL procedure_name(arg1, arg2);

-- For procedures with output parameters
CALL procedure_name(arg1, @output_var);
SELECT @output_var;
```

Example 1: GetCustomerPurchaseStatus

This procedure demonstrates:

- Multiple IN parameters
- Multiple OUT parameters
- Variable declaration
- IF...ELSEIF...ELSE conditional logic
- JOIN queries with aggregation

```
DELIMITER //

CREATE PROCEDURE GetCustomerPurchaseStatus(
    IN p_customer_id INT,
    OUT p_customer_name VARCHAR(100),
    OUT p_total_spent DECIMAL(10, 2),
    OUT p_purchase_status VARCHAR(20)
)
```

```
BEGIN
    DECLARE v_total_amount DECIMAL(10, 2);

    SELECT customer_name, COALESCE(SUM(order_total), 0)
    INTO p_customer_name, v_total_amount
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    WHERE c.customer_id = p_customer_id
    GROUP BY c.customer_id;

    SET p_total_spent = v_total_amount;

    IF v_total_amount &gt; 5000 THEN
        SET p_purchase_status = 'VIP';
    ELSEIF v_total_amount &gt; 2000 THEN
        SET p_purchase_status = 'LOYAL';
    ELSEIF v_total_amount &gt; 500 THEN
        SET p_purchase_status = 'REGULAR';
    ELSE
        SET p_purchase_status = 'NEW';
    END IF;
END //

DELIMITER ;
```

Execution:

```
CALL GetCustomerPurchaseStatus(101, @name, @spent, @status);
SELECT @name, @spent, @status;
```

Output Example:

```
@name: John Smith
@spent: 3500.00
@status: LOYAL
```

Example 2: AddProductToInventory

This procedure demonstrates:

- Input validation
- Error handling with DECLARE...HANDLER
- Multiple conditional paths
- Transaction logging

```
DELIMITER //

CREATE PROCEDURE AddProductToInventory(
    IN p_product_id INT,
    IN p_quantity INT,
    OUT p_success BOOLEAN,
    OUT p_message VARCHAR(200)
)
BEGIN
    DECLARE v_current_quantity INT;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        SET p_success = FALSE;
        SET p_message = 'Error: Database exception occurred';
    END;

    IF p_quantity &lt;= 0 THEN
        SET p_success = FALSE;
        SET p_message = 'Error: Quantity must be greater than zero';
    ELSE
        SELECT quantity INTO v_current_quantity
        FROM inventory
        WHERE product_id = p_product_id FOR UPDATE;

        IF v_current_quantity IS NULL THEN
            SET p_success = FALSE;
            SET p_message = 'Error: Product not found';
        ELSE
            UPDATE inventory
            SET quantity = quantity + p_quantity,
                last_updated = NOW()
            WHERE product_id = p_product_id;

            INSERT INTO inventory_log(product_id, quantity_changed, action, timestamp)
            VALUES(p_product_id, p_quantity, 'ADD', NOW());

            SET p_success = TRUE;
            SET p_message = CONCAT('Success: Added ', p_quantity, ' units');
        END IF;
    END IF;
END //

DELIMITER ;
```

3. MySQL User-Defined Functions

Syntax and Structure

```
DELIMITER //

CREATE FUNCTION function_name(
    param_name datatype,
    ...
)
RETURNS return_datatype
[DETERMINISTIC | NOT DETERMINISTIC]
[READS SQL DATA | MODIFIES SQL DATA | NO SQL | CONTAINS SQL]
BEGIN
```

```
    DECLARE variable_name datatype;

    -- Logic

    RETURN expression;  -- Must have exactly one RETURN
END //

DELIMITER ;
```

Function Characteristics

- **DETERMINISTIC**: Same input always produces same output (enables optimization)
- **NON DETERMINISTIC** (default): May return different results
- **READS SQL DATA**: Reads database (most common)
- **MODIFIES SQL DATA**: May modify database (rare)
- **Parameters**: Always treated as IN (read-only)
- **RETURN**: Mandatory, exactly one per function

Usage

```
-- In SELECT
SELECT GetCustomerCreditScore(customer_id) FROM customers;

-- In WHERE clause
SELECT * FROM customers WHERE GetCustomerCreditScore(customer_id) &gt; 700;

-- In other functions
SELECT CONCAT(first_name, ' (Score: ', GetCustomerCreditScore(customer_id), ')')
FROM customers;
```

Example 1: CalculateAge

Simple age calculation from date of birth:

```
DELIMITER //

CREATE FUNCTION CalculateAge(p_dob DATE)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE v_age INT;
    SET v_age = YEAR(CURDATE()) - YEAR(p_dob)
                - (DATE_FORMAT(CURDATE(), '%m%d') &lt; DATE_FORMAT(p_dob, '%m%d'));
    RETURN v_age;
END //

DELIMITER ;
```

Usage:

```
SELECT CalculateAge('1990-05-15');  -- Returns: 34
SELECT employee_name, CalculateAge(date_of_birth) AS age FROM employees;
```

Example 2: GetCustomerCreditScore

Complex function with multiple calculations and conditional logic:

```
DELIMITER //

CREATE FUNCTION GetCustomerCreditScore(p_customer_id INT)
RETURNS INT
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE v_score INT DEFAULT 0;
    DECLARE v_total_orders INT;
    DECLARE v_on_time_payments INT;
    DECLARE v_late_payments INT;
    DECLARE v_total_spent DECIMAL(10, 2);

    SELECT COUNT(*) INTO v_total_orders
    FROM orders WHERE customer_id = p_customer_id;

    SELECT COUNT(*) INTO v_on_time_payments
    FROM order_payments WHERE customer_id = p_customer_id AND payment_status = 'ON_TIME';

    SELECT COUNT(*) INTO v_late_payments
    FROM order_payments WHERE customer_id = p_customer_id AND payment_status = 'LATE';

    SELECT COALESCE(SUM(order_total), 0) INTO v_total_spent
    FROM orders WHERE customer_id = p_customer_id;

    SET v_score = 500;

    IF v_total_orders &gt; 0 THEN
        SET v_score = v_score + (v_on_time_payments * 300 / v_total_orders);
    END IF;

    IF v_late_payments &gt; 0 THEN
        SET v_score = v_score - (v_late_payments * 200 / GREATEST(v_total_orders, 1));
    END IF;

    IF v_total_spent &gt; 10000 THEN
        SET v_score = v_score + 200;
    ELSEIF v_total_spent &gt; 5000 THEN
        SET v_score = v_score + 100;
    END IF;

    RETURN CASE
        WHEN v_score &lt; 0 THEN 0
        WHEN v_score &gt; 1000 THEN 1000
        ELSE v_score
    END;
END //

DELIMITER ;
```

4. SQLite Triggers (Alternative to Procedures/Functions)

Important Note

SQLite does NOT support traditional stored procedures or user-defined functions. Instead, use **TRIGGERS** for automated database logic.

Syntax

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
[BEFORE|AFTER|INSTEAD OF] [INSERT|UPDATE|DELETE]
ON table_name
[WHEN condition]
FOR EACH ROW
BEGIN
    SQL_statements;
END;
```

Special Variables

- **NEW.column_name**: New value (for INSERT and UPDATE)
- **OLD.column_name**: Previous value (for UPDATE and DELETE)

Example 1: UpdateProductTimestamp

Auto-update last_modified timestamp on any product update:

```
CREATE TRIGGER UpdateProductTimestamp
AFTER UPDATE ON products
FOR EACH ROW
BEGIN
    UPDATE products SET last_updated = CURRENT_TIMESTAMP WHERE product_id = NEW.product_i
END;
```

Example 2: LogInventoryChanges

Audit trail for inventory modifications:

```
CREATE TRIGGER LogInventoryChanges
AFTER UPDATE ON inventory
FOR EACH ROW
BEGIN
```

```
    INSERT INTO inventory_audit_log(product_id, old_quantity, new_quantity, changed_on)
    VALUES(NEW.product_id, OLD.quantity, NEW.quantity, CURRENT_TIMESTAMP);
END;
```

Example 3: PreventNegativeInventory

Validation trigger using RAISE:

```
CREATE TRIGGER PreventNegativeInventory
BEFORE UPDATE ON inventory
FOR EACH ROW
WHEN NEW.quantity &lt; 0
BEGIN
    SELECT RAISE(ABORT, 'Inventory cannot be negative');
END;
```

5. Best Practices and Tips

1. **Naming Conventions**
   - Procedures: `sp_GetCustomers`, `sp_UpdateInventory`
   - Functions: `fn_CalculateAge`, `fn_GetCreditScore`
   - Parameters: `p_customerID`, `p_quantity`
   - Variables: `v_totalAmount`, `v_counter`
2. **Error Handling**
   - Always validate input parameters
   - Use DECLARE...HANDLER for exception management
   - Return meaningful error messages
3. **Performance**
   - Mark functions DETERMINISTIC when applicable
   - Index columns used in WHERE clauses
   - Minimize cursor usage; prefer JOINs
4. **Security**
   - Use procedures to prevent SQL injection
   - Restrict user privileges; grant EXECUTE only
   - Parameterize all inputs
5. **Documentation**
   - Include comments for purpose and parameters
   - Provide example execution statements
   - Document any prerequisites

6. Execution in Tools

MySQL Workbench

1. Open new SQL editor tab
2. Paste CREATE PROCEDURE or CREATE FUNCTION code
3. Execute (Ctrl+Enter or Execute button)
4. To call: Type CALL procedure_name(...);
5. View in Navigator → Schemas → Stored Procedures

DB Browser for SQLite

1. Select "Execute SQL" tab
2. Paste CREATE TRIGGER code
3. Execute to create trigger
4. Trigger fires automatically on matching events
5. View triggers in Database Structure panel

Conclusion

Stored procedures, functions, and triggers are powerful tools for building robust, secure, and maintainable database systems. By mastering these concepts, developers can create reusable logic, enforce business rules at the database level, and significantly improve application performance and security.

Use the provided examples as templates for your own implementations, and always follow best practices for naming, documentation, and error handling.

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29]

⁂

1. https://www.sqlshack.com/learn-mysql-the-basics-of-mysql-stored-procedures/
2. https://stackoverflow.com/questions/5039324/creating-a-procedure-in-mysql-with-parameters
3. https://www.mysqltutorial.org/mysql-stored-procedure/mysql-if-statement/
4. https://www.mysqltutorial.org/mysql-stored-procedure/stored-procedures-parameters-in-mysql/
5. https://www.youtube.com/watch?v=oagHZwY9JJY
6. https://www.oreilly.com/library/view/mysql-stored-procedure/0596100892/ch04s02.html
7. https://www.tutorialsteacher.com/sqlserver/stored-procedure-parameters
8. https://200oksolutions.com/blog/mysql-stored-procedures-vs-functions/
9. https://stackoverflow.com/questions/6326082/using-if-and-else-in-mysql-stored-procedures
10. https://www.ibm.com/docs/en/data-studio/4.1.1?topic=statements-out-inout-parameters-in-storedprocedureresult-objects

11. https://dev.mysql.com/doc/refman/8.2/en/stored-programs-defining.html
12. https://sqlite.org/forum/info/3354e0eabd81f2a39a338820f4616a23e4a2b2974454bbfb4cf8ec922a221ddd
13. https://www.tutorialspoint.com/how-mysql-if-else-statement-can-be-used-in-a-stored-procedure
14. https://system.data.sqlite.org/home/doc/0a3d6229a7425242/Doc/Extra/Core/lang_createtrigger.html
15. https://sqlitebrowser.org
16. https://www.mysqltutorial.org/mysql-stored-procedure/getting-started-with-mysql-stored-procedures/
17. https://www.sqlitetutorial.net/sqlite-trigger/
18. https://datacarpentry.github.io/sql-socialsci/02-db-browser.html
19. https://phoenixnap.com/kb/mysql-stored-procedure
20. https://sqlite.org/lang_createtrigger.html
21. https://www.youtube.com/watch?v=fPWiZhVjvIU
22. https://stackoverflow.com/questions/71960525/sqlite-conditionals-in-creation-of-a-trigger
23. https://dotnettutorials.net/lesson/user-defined-functions-in-mysql/
24. https://dev.mysql.com/doc/refman/9.2/en/create-procedure.html
25. https://stackoverflow.com/questions/2088905/pros-and-cons-of-triggers-vs-stored-procedures-for-denormalization
26. https://mariadb.com/docs/server/server-usage/user-defined-functions/create-function-udf
27. https://dev.mysql.com/doc/refman/8.3/en/create-procedure.html
28. https://www.geeksforgeeks.org/dbms/difference-between-trigger-and-procedure-in-dbms/
29. https://www.geeksforgeeks.org/mysql/mysql-creating-stored-function/