MySQL: Procedural Programming

Stored Procedures, Functions, and Triggers

Procedural SQL Overview

- SQL is **declarative** - you specify *what* you want
- SQL is NOT procedural - you don't specify *how* to get it
- **Persistent Stored Modules (PSM)** - SQL-99 standard
  - Block of code with SQL statements + procedural extensions
  - Stored and executed at the DBMS server
- **Benefits:**
  - Access-controlled sharing across users
  - Better performance (server-side execution)
  - Code reuse

MySQL Procedural SQL Capabilities

You can create:

1. **Stored Procedures** - named collection of SQL + procedural statements
2. **Triggers** - automatically executed in response to events
3. **User-Defined Functions (UDFs)** - return values like built-in functions

Note: Don't confuse UDFs with SQL's built-in functions (MIN, COUNT, etc.)

- Built-in functions: only in SQL statements
- UDFs: mainly in triggers and stored procedures

Stored Procedures

> A **stored procedure** is a named collection of procedural and SQL statements, stored in the database

Why useful?

- ✅ Better performance (stored & executed in database)
- ✅ Facilitates code reuse
- ✅ Encapsulates business logic
- ✅ Centralized maintenance

Creating a Stored Procedure

Example: Get all overdue tasks from a GTD (Getting Things Done) database

```
DELIMITER //

CREATE PROCEDURE GetOverdueTasks()
BEGIN
    SELECT task_name, task_due
    FROM task
    WHERE task_due IS NOT NULL AND task_due < CURDATE();
END //

DELIMITER ;
```

How it works:

1. Change delimiter to allow `;` inside procedure
2. Use `CREATE PROCEDURE` statement
3. Change delimiter back to default

Executing and Dropping Procedures

Execute:

```
CALL GetOverdueTasks();
```

Drop:

```
DROP PROCEDURE procedure_name;
```

List procedures:

```
SHOW PROCEDURE STATUS WHERE db = 'gtd_db';
```

View procedure definition:

```
SHOW CREATE PROCEDURE GetOverdueTasks;
```

Procedure Parameters

Three types of parameters:

Type	Description	Caller sees changes?	Procedure sees initial value?
IN	Input only	❌	✅
OUT	Output only	✅	❌
INOUT	Input and output	✅	✅

Example: IN Parameters

Create a new task with a status_id:

```
DELIMITER //

CREATE PROCEDURE CreateTask(
    IN task_name VARCHAR(255),
    IN status_id INT UNSIGNED
)
BEGIN
    INSERT INTO task(task_name, status_id)
    VALUES (task_name, status_id);
END //

DELIMITER ;
```

Usage:

```
CALL CreateTask('Write report', 1);
```

Variables in Stored Procedures

Two types:

1. **Local variables** - used within procedure, lost when done
    - Declared with `DECLARE` (just after `BEGIN`)
    - Initialized with `DEFAULT` keyword (otherwise NULL)
2. **User-defined variables** - persist throughout session
    - Prefixed with `@`
    - Persist across procedures

Assignment:

```
SET variable_name = value;
```

Example: Variables

Create task with status name instead of status_id:

```
DELIMITER //
CREATE PROCEDURE CreateTask(
    IN in_task VARCHAR(255),
    IN in_status VARCHAR(255)
)
BEGIN
    DECLARE v_status_id INT UNSIGNED;

    SELECT status_id INTO v_status_id
    FROM status
    WHERE status_name = in_status;

    INSERT INTO task(task_name, status_id)
    VALUES (in_task, v_status_id);
END //
DELIMITER ;
```

Control Flow: IF Statement

```
IF condition1 THEN
    statements;
ELSEIF condition2 THEN
    statements;
ELSE
    statements;
END IF;
```

Example: IF with Default Values

Create task with optional project and default status:

```
DELIMITER //
CREATE PROCEDURE CreateTask(
    IN in_task VARCHAR(255),
    IN in_status VARCHAR(255),
    IN in_project VARCHAR(255)
)
BEGIN
    DECLARE v_status_id INT UNSIGNED;
    DECLARE v_proj_id INT UNSIGNED;

    # Default to 'Next' status
    IF in_status IS NULL THEN
        SET in_status = 'Next';
    END IF;
    SELECT status_id INTO v_status_id
    FROM status WHERE status_name = in_status;

    # Handle optional project
    IF in_project IS NOT NULL THEN
        SELECT proj_id INTO v_proj_id
        FROM project WHERE proj_name = in_project;
    END IF;

    INSERT INTO task(task_name, status_id, proj_id)
    VALUES (in_task, v_status_id, v_proj_id);
END //
```

🎯 Active Learning: Quick Check

Question: What's the difference between an IN parameter and a local variable?

Take 1 minute to discuss with a neighbor

Control Flow: LOOP

```
[begin_label:] LOOP
    statements;
END LOOP [end_label]
```

Exit the loop:

```
IF condition THEN
    LEAVE loop_label;
END IF;
```

Note: MySQL also has `WHILE` and `REPEAT` loops

Example: LOOP with Calendar Table

Setup:

```
CREATE TABLE calendar (
    date DATE PRIMARY KEY,
    month INT NOT NULL,
    quarter INT NOT NULL,
    year INT NOT NULL
);
```

Example: Fill Calendar Dates

```
DELIMITER //
CREATE PROCEDURE fillDates(
    IN startDate DATE,
    IN endDate DATE
)
BEGIN
    DECLARE currentDate DATE DEFAULT startDate;

    insert_date: LOOP
        -- exit if done
        IF currentDate > endDate THEN
            LEAVE insert_date;
        END IF;

        -- insert date
        INSERT INTO calendar(date, month, quarter, year)
        VALUES(currentDate, MONTH(currentDate),
               QUARTER(currentDate), YEAR(currentDate));

        -- increment
        SET currentDate = DATE_ADD(currentDate, INTERVAL 1 DAY);

    END LOOP;
END //
DELIMITER ;
```

Cursors

> **Cursor**: Database object for iterating through SELECT results

Use when: You need to process individual rows one at a time

Basic workflow:

1. Declare cursor
2. Declare NOT FOUND handler
3. Open cursor
4. Fetch and process rows
5. Close cursor

Cursor Syntax

```
-- Declare cursor
DECLARE cursor_name CURSOR FOR
SELECT column1, column2
FROM your_table
WHERE your_condition;

-- Declare NOT FOUND handler
DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = true;

-- Open cursor
OPEN cursor_name;

-- Fetch data
FETCH cursor_name INTO variable1, variable2;
-- Process the data

-- Close cursor
CLOSE cursor_name;
```

Example: List Tasks with Names

Goal: Replace status_id and proj_id with status_name and proj_name

```
DELIMITER //
CREATE PROCEDURE ListTasks()
BEGIN
    DECLARE v_done BOOL DEFAULT false;
    DECLARE v_task_name VARCHAR(255);
    DECLARE v_status_id INT UNSIGNED;
    DECLARE v_status_name VARCHAR(255);
    DECLARE v_proj_id INT UNSIGNED;
    DECLARE v_proj_name VARCHAR(255);

    -- Declare cursor
    DECLARE cursor_all_tasks CURSOR FOR
        SELECT task_name, status_id, proj_id
        FROM task;

    -- NOT FOUND handler
    DECLARE CONTINUE HANDLER
        FOR NOT FOUND SET v_done = true;
```

Example: List Tasks (continued)

```
    -- Create temp table for output
    CREATE TEMPORARY TABLE result(
        task_name VARCHAR(255),
        status_name VARCHAR(255),
        proj_name VARCHAR(255)
    );

    OPEN cursor_all_tasks;

    process_task: LOOP
        FETCH cursor_all_tasks INTO v_task_name, v_status_id, v_proj_id;
        IF v_done = true THEN
            LEAVE process_task;
        END IF;

        -- Fetch status name
        IF v_status_id IS NOT NULL THEN
            SELECT status_name INTO v_status_name
            FROM status WHERE status_id = v_status_id;
        ELSE
            SET v_status_name = NULL;
        END IF;
```

Example: List Tasks (continued)

```
        -- Fetch project name
        IF v_proj_id IS NOT NULL THEN
           SELECT proj_name INTO v_proj_name
           FROM project WHERE proj_id = v_proj_id;
        ELSE
            SET v_proj_name = NULL;
        END IF;

        INSERT INTO result
        VALUES (v_task_name, v_status_name, v_proj_name);

    END LOOP;

    CLOSE cursor_all_tasks;

    SELECT * FROM result;
    DROP TEMPORARY TABLE result;
END //

DELIMITER ;
```

User-Defined Functions (UDFs)

- Like a stored procedure but **returns a value**
- Can be invoked from:
  - Stored procedures ✅
  - Triggers ✅
  - SQL statements ✅

UDF Syntax

```
CREATE FUNCTION function_name (
    IN argument data-type, …
)
RETURNS data-type
BEGIN
    Procedure SQL statements;
    …
    RETURN (value or expression);
END;
```

Triggers

> **Triggers advance stored procedures by having the DBMS run them automatically under specified conditions**

Key Properties:

1. Invoked **BEFORE** or **AFTER** data mutation
    - MySQL: row-level triggers only
    - SQL standard: also statement-level triggers
2. Associated with **one table**
3. A table can have **many triggers**
4. Executed as **part of the transaction** that triggered it

Trigger Use Cases

- **Enforce constraints** beyond what declarative constraints allow
- **Enforce referential integrity** with custom logic
- **Auditing** - track mutation history
- **Derived data maintenance** - update statistics/aggregates
- **Validation** - abort invalid operations

Trigger Syntax

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- Trigger body (SQL statements)
END;
```

Important modifiers:

- `OLD` - values before mutation
- `NEW` - values after mutation

BEFORE vs AFTER Triggers

Type	When to Use	Common Operations
BEFORE	Abort mutation based on logic	Validation, setting derived values
AFTER	React to successful mutation	Auditing, updating dependent data

Multiple triggers:

```
{FOLLOWS | PRECEDES} existing_trigger_name
```

(after `FOR EACH ROW`)

Calling Procedures from Triggers

Use the `CALL` statement:

```
BEGIN
    CALL stored_procedure_name(params);
END;
```

Restriction: Stored procedure must have **no OUT or INOUT params**

Example Setup: Items Table

```
CREATE TABLE items (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);

INSERT INTO items(id, name, price)
VALUES (1, 'Item', 50.00);
```

Audit table (1:M relationship):

```
CREATE TABLE item_changes (
    change_id INT PRIMARY KEY AUTO_INCREMENT,
    item_id INT,
    change_type VARCHAR(10),
    change_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (item_id) REFERENCES items(id)
);
```

Example: Audit Trigger

```
DELIMITER //

CREATE TRIGGER update_items_trigger
AFTER UPDATE
ON items
FOR EACH ROW
BEGIN
    INSERT INTO item_changes (item_id, change_type)
    VALUES (NEW.id, 'UPDATE');
END;
//

DELIMITER ;
```

Test it:

```
UPDATE items SET price = 60.00 WHERE id = 1;
SELECT * FROM item_changes;
```

🎯 Active Learning: Design a Trigger

Scenario: You have a `CUSTOMER` table with:

- `cust_balance` (current balance)
- `cust_total_purchases` (lifetime total)

Task: Design a trigger that updates `cust_total_purchases` when a new purchase is recorded in the `PURCHASE` table.

Discuss with your neighbor: Should this be BEFORE or AFTER? Why?

Trigger Caution: Concurrency Issues

Problem: Triggers susceptible to multiple SQL commands operating on the same rows simultaneously in the same transaction

Example scenario:

- PRODUCT table with: `P_QOH`, `P_MIN`, `P_REORDER`
- Want to set `P_REORDER` when inventory is low

What DOESN'T Work

```
DELIMITER //
CREATE TRIGGER TRG_PRODUCT_REORDER
AFTER UPDATE ON PRODUCT
FOR EACH ROW
BEGIN
    UPDATE PRODUCT
    SET P_REORDER = 1
    WHERE P_QOH <= P_MIN;
END
//
DELIMITER ;
```

Why it fails:

1. Triggering statement updates row (e.g., P_CODE = '6')
2. Pauses to run trigger
3. Trigger attempts to update **same row**
4. ❌ **Blocked!** Only one UPDATE can operate on a row at a time

What DOES Work: Solution 1

```
DELIMITER //
CREATE TRIGGER TRG_PRODUCT_REORDER
BEFORE UPDATE ON PRODUCT
FOR EACH ROW
BEGIN
    IF NEW.P_QOH <= NEW.P_MIN THEN
        SET NEW.P_REORDER = 1;
    END IF;
END
//
DELIMITER ;
```

Why it works:

- Uses `BEFORE` instead of `AFTER`
- Modifies `NEW` values in memory (before disk write)
- No DML statement needed
- No concurrent row access conflict

Understanding OLD and NEW

DBMS creates two copies of every row being changed:

- `OLD` - values **before** any changes
- `NEW` - values **after** changes (in memory)

Can only be referenced within trigger actions

BEFORE trigger: Modifies NEW before the change is written to disk

Solution Refinement: Reset Flag

```
BEGIN
    IF NEW.P_QOH <= NEW.P_MIN THEN
        SET NEW.P_REORDER = 1;
    ELSE
        SET NEW.P_REORDER = 0;
    END IF;
END
```

But wait! What about INSERT operations with inconsistent P_REORDER?

Final Solution: DRY with Procedure

Create reusable procedure:

```
DELIMITER //
CREATE PROCEDURE PRC_PRODUCT_REORDER(
    IN PQOH INT,
    IN PMIN INT,
    OUT PREORDER INT
)
BEGIN
    IF PQOH <= PMIN THEN
        SET PREORDER = 1;
    ELSE
        SET PREORDER = 0;
    END IF;
END;
//
DELIMITER ;
```


Note: Procedure uses variables, cannot reference NEW/OLD

Final Solution: UPDATE Trigger

```
DELIMITER //
CREATE TRIGGER TRG_UPDATE_PRODUCT_REORDER
BEFORE UPDATE ON PRODUCT
FOR EACH ROW
BEGIN
    CALL PRC_PRODUCT_REORDER(
        NEW.P_QOH,
        NEW.P_MIN,
        NEW.P_REORDER
    );
END;
//
DELIMITER ;
```

Final Solution: INSERT Trigger

```
DELIMITER //
CREATE TRIGGER TRG_INSERT_PRODUCT_REORDER
BEFORE INSERT ON PRODUCT
FOR EACH ROW
BEGIN
    CALL PRC_PRODUCT_REORDER(
        NEW.P_QOH,
        NEW.P_MIN,
        NEW.P_REORDER
    );
END;
//
DELIMITER ;
```

Result: No code duplication! ✅

Other Trigger Statements

Drop trigger:

```
DROP TRIGGER trigger_name;
```

List all triggers:

```
SHOW TRIGGERS;
```

🎯 Active Learning: Trigger Debugging

Bug Hunt: What's wrong with this trigger?

```
CREATE TRIGGER validate_price
AFTER INSERT ON products
FOR EACH ROW
BEGIN
    IF NEW.price < 0 THEN
        SET NEW.price = 0;
    END IF;
END;
```

Take 2 minutes to identify the issue and propose a fix

Key Takeaways

✅ **Stored Procedures:** Reusable, performant server-side code
✅ **Parameters:** IN, OUT, INOUT for flexible interfaces
✅ **Control Flow:** IF, LOOP, cursors for complex logic
✅ **UDFs:** Return values, usable in SQL and procedures
✅ **Triggers:** Automatic execution for data integrity and auditing
✅ **BEFORE vs AFTER:** Choose based on use case
✅ **Avoid concurrency issues:** Use BEFORE triggers with NEW/OLD

Best Practices

1. **Use meaningful names** for procedures, functions, triggers
2. **Keep procedures focused** - single responsibility principle
3. **Prefer BEFORE triggers** when modifying the same row
4. **Use procedures to eliminate code duplication** in triggers
5. **Document complex logic** with comments
6. **Test edge cases** thoroughly, especially with triggers
7. **Consider performance** - triggers run on every mutation