Teaching Mobility at the University of Macedonia

Department of Applied Informatics

Topic : Stored Routines and Triggers in MySQL

Presenter: Dhori Beta
Department of Informatics, Fan S. Noli University
18-22 May 2026

An overview of stored routines

Stored routines are database objects that encapsulate SQL code for repeated execution. They come in two forms:

- **stored procedure**
- **sstored functions**.

Stored procedures perform actions without necessarily returning a value, whereas stored functions are designed to return a value, much like functions in programming languages such as C#, Python, or Java.

These routines efficiently centralize logic within the database, providing reusability and consistency in database operations.

The benefits of stored routines

1 - To perform database operations, a client will send SQL statements to the server and await the results. Based on the results, further instructions may be sent by the client, followed by yet more based on subsequent return values, and so on. If the client is remote from the server, each transaction will introduce a delay due to network latency.

Stored routines eliminate much of this communication by storing the SQL logic on the server. Instead of a repetitive back and forth between the client and server, the client can call the routine and await the final result

The benefits of stored routines

2 - When working with distributed systems like client-server databases, it is important to minimize the number of places where changes to code logic need to be made. Consider, for example, a set of SQL statements common to various clients of a specific database. If this code is stored within the clients instead of the server, any bug fixes or code updates must be applied to every client. If the code is instead stored centrally on the database server, however, the code updates only need to be made in one location, resulting in easier code maintenance.

3 - Stored routines also provide an extra layer of database security. Instead of providing a user with access privileges to a table containing sensitive data, the user can be allowed to run a stored routine that performs only the necessary actions to complete a task.

Creating stored procedures

Stored procedures are created in MySQL using the CREATE PROCEDURE statement, defining the procedure's name, specifying its parameters (if any), and encapsulating the SQL logic within a BEGIN...END block

The syntax for declaring stored procedures is

CREATE PROCEDURE <procedure_name>(IN/OUT/INOUT <param> <type>, <param> <type> ..)
BEGIN
<SQL statements>
END

Syntax for declaring stored procedures

CREATE PROCEDURE <procedure_name>(IN/OUT/INOUT <param> <type>, <param> <type> ..)
BEGIN
<SQL statements>
END

Stored routines support three parameter types: IN, OUT, and INOUT

- *IN parameter allows a value to be passed into the routine*
- *OUT parameter enables the routine to return a value to the caller.*
- *INOUT parameter serves as both an input and an output, allowing the caller to pass in a value and receive an updated value in return.*

Working with MySQL variables

MySQL variables store temporary values that can be referenced and manipulated during a session or within a query.

MySQL supports both user-defined and system variables.

User-defined variable names are prefixed with the '@' character and created using the SET statement:

```
SET @myVar = 150;
```

User-defined variables can also be set from query results using the ***SELECT INTO*** *statement, the syntax for which is as follows:*

```
SELECT <column_1>, <column_2> ...
INTO <variable_1>, <variable_2>, ...
FROM <table>;
```

These variables are session-specific, persist until the session ends and can be inspected using the SELECT statement:

```
SELECT @myVar;
```

Working with MySQL variables

System variables

*control the configuration and behavior of the MySQL server and can be either global or session-specific. They are predefined by MySQL and can be viewed or modified using the **SHOW VARIABLES** and **SET** commands, respectively*

Stored procedure examples

*The following code creates a procedure named **ProductsBySupplier()** that takes a company name as an input parameter and retrieves a list of associated products:*

```
CREATE PROCEDURE ProductsBySupplier(IN company_name varchar(25))
BEGIN
    SELECT product.name
    FROM product
    RIGHT JOIN supplier
    ON product.supplier_id = supplier.id
    WHERE company = company_name;
END
```

Stored procedure examples

To execute this procedure, we use the CALL statement, as shown below:

CALL ProductsBySupplier('Apple');

Stored procedure examples

The procedure below demonstrates the use of IN and OUT parameters in the same routine. The input parameter is the supplier name, while the output parameters will contain the lowest and highest prices of the supplier's products:

```
CREATE PROCEDURE SupplierMinMax( IN company_name varchar(25),
OUT min_price DECIMAL(6,2), OUT max_price DECIMAL(6,2))
BEGIN
    SELECT MIN(price), MAX(price)
    INTO min_price, max_price
    FROM product
    RIGHT JOIN supplier
    ON product.supplier_id = supplier.id
    WHERE company = company_name;
END
```

Stored procedure examples

When we call the procedure, we will pass references to variables for the ***OUT*** *parameters. After the procedure finishes, these variables will hold the minimum and maximum product prices. We assign these values to the variables within the procedure using the following SELECT INTO statement:*

SELECT MIN(price), MAX(price) INTO min_price, max_price ...

We can use the following CALL statement to execute the procedure:

CALL SupplierMinMax('Apple', @min_price, @max_price);

Stored procedure examples

We could also use these variables in a SELECT statement to identify products with minimum and maximum prices:

```
SELECT name, price
FROM product
WHERE price = @min_price OR price = @max_price;
```

Creating stored functions

MySQL stored functions are blocks of SQL code that perform a task and return a result.

A set of keywords are used when writing functions, including

RETURNS,

RETURN

DETERMINISTIC

as well as interaction clauses like

NO SQL

CONTAINS SQL

MODIFIES SQL DATA

READS SQL DATA.

These keywords and clauses define how the function interacts with the database and how MySQL optimizes its execution.

Creating stored functions

The syntax for declaring a function

CREATE FUNCTION *<function_name>(<param> <type>, <param> <type> ..)*
RETURNS *<return_type>*
    NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
    DETERMINISTIC | NOT DETERMINISTIC
BEGIN
    RETURN <expression>;
END

CREATE FUNCTION *<function_name>(<param> <type>, <param> <type> ..)*
RETURNS *<return_type>*
NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
DETERMINISTIC | NOT DETERMINISTIC
BEGIN
RETURN <expression>;
END

- *The **RETURNS** keyword declares the data type of the function's return value, which is mandatory for all stored functions. Declaring the incorrect return type may result in unpredictable behavior.*

- *The **RETURN** keyword is used within a function's body to indicate the value that will be returned to the caller. Each function can have only one RETURN statement for every possible execution path, and the value returned must be consistent with the data type defined in the RETURNS clause*

CREATE FUNCTION *<function_name>(<param> <type>, <param> <type> ..)*
RETURNS *<return_type>*
NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
DETERMINISTIC | NOT DETERMINISTIC
BEGIN
RETURN <expression>;
END

- ***DETERMINISTIC*** *-If a function is declared as deterministic, it must return the same result each time it is called when passed the same input parameters. This predictability allows MySQL to cache return values and return the results from previous matching calls without executing the overhead of rerunning the function.*
- ***NON DETERMINISTIC*** *- The return results of non-deterministic functions vary, even when passed the same input parameters. This variability is usually the result of a dependency on dynamic values within the function's logic, such as the current date or random number generation*

CREATE FUNCTION *<function_name>(<param> <type>, <param> <type> ..)*
RETURNS *<return_type>*
NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
DETERMINISTIC | NOT DETERMINISTIC
BEGIN
RETURN <expression>;
END

- *The **NOSQL** clause indicates to MySQL that the function body does not execute SQL statements. The function relies entirely on its input parameters and internal logic to perform calculations or operations without interacting with the database.*
- *A function declared with **READS SQL DATA** can execute SQL queries that retrieve data from the database but cannot modify any data. This clause is used for functions that depend on database queries to calculate their results*

CREATE FUNCTION *<function_name>(<param> <type>, <param> <type> ..)*
RETURNS *<return_type>*
NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
DETERMINISTIC | NOT DETERMINISTIC
BEGIN
RETURN <expression>;
END

- *The **MODIFIES SQL DATA** clause tells MySQL that the function will perform database updates such as inserting, updating, or deleting data. Stored functions are generally intended to return results without altering data, so this clause **is not frequently use***
- *The **CONTAINS SQL** clause tells MySQL that the function uses SQL statements that neither read from tables nor modify the database. While the function may perform operations such as declaring variables or calling other stored routines, it does not directly interact with table data.*

Stored function examples

```
CREATE FUNCTION CalculateTax(price DECIMAL(10, 2), tax_rate DECIMAL(10, 2))
RETURNS DECIMAL(10, 2)
        NO SQL
        DETERMINISTIC
BEGIN
        RETURN price * (tax_rate / 100);
END
```

The function does not use SQL statements and will return the same value when provided the same input values, so it is declared using the NO SQL and DETERMINISTIC clauses.

After adding the function to the database, execute it as follows:

```
SELECT CalculateTax(220, 5.25);
```

```
CREATE FUNCTION FormatName(first_name VARCHAR(20), last_name
VARCHAR(20))
        RETURNS VARCHAR(40)
        CONTAINS SQL
        DETERMINISTIC
BEGIN
        DECLARE full_name VARCHAR(40);
        SET full_name = CONCAT(first_name, ' ', last_name);
        RETURN full_name;
END
```

This function *FormatName() is deterministic and uses the SQL CONCAT function. Since the function does not access the database, it was declared using the CONTAINS SQL clause.*

```
S       ELECT FormatName('Davy', 'Crockett');
```

Viewing procedures and functions

Information about routines is stored in the routines and functions tables of the built-in MySQL information_schema database and is accessed using the following syntax:

SELECT *<column names>*
FROM *information_schema.routines*
WHERE *routine_type = '[PROCDECURE | FUNCTION]'*
AND *routine_schema = '<database name>'\G*

Deleting stored procedures and functions

Stored routines are deleted using the
DROP FUNCTION and DROP PROCEDURE statement
.
For example:

```
DROP PROCEDURE SupplierMinMax;
DROP PROCEDURE ProductsBySupplier;
DROP FUNCTION CalculateTax;
DROP FUNCTION FormatName;
```

Automation with MySQL
Triggers

An overview of MySQL triggers

Triggers automatically execute blocks of SQL code in response to changes to a database table, such as

deleting rows
inserting rows
updating rows.

These triggers are configured to execute ***befor****e or* ***after*** *an event occurs.*

For instance, you could use a trigger to validate column values before a row is inserted into a table or to add a row to an audit log table after a row has been deleted.

The syntax for creating triggers

Triggers are declared in MySQL using the CREATE TRIGGER statement combined with clauses defining the circumstance under which the trigger will activate.

The CREATE TRIGGER statement uses the following syntax:

CREATE TRIG*GER <trigger_name>*
    [BEFORE | AFTER] [INSERT | UPDATE | DELETE]
***ON** <table_name>*
FOR EACH ROW
BEGIN
    <trigger_body>
***END**;*

CREATE TRIG*GER <trigger_name>*
[BEFORE | AFTER] [INSERT | UPDATE | DELETE]
ON *<table_name>*
FOR EACH ROW
BEGIN
<trigger_body>
END*;*

- ***trigger_name*** *– The trigger name - must be unique within the scope of the database.*
- ***BEFORE | AFTER*** *- Defines whether the trigger execution occurs before or after the event.*
- ***INSERT | UPDATE | DELETE*** *- Specifies the event that will activate the trigger.*
- ***table_name*** *- The table on which the specified event will activate the trigger.*
- ***FOR EACH ROW*** *- Indicates that the trigger operates at the row level.*
- ***trigger_bo****dy - The SQL statements to execute when the trigger is activated.*

Accessing trigger event data

Triggers activate in response to changes in table data and execute the SQL code in the trigger body in response.

A typical trigger body will act based on the data changes that activated the trigger.

- *a trigger to log row deletions might need to include information about the deleted row.*
- *In the case of an UPDATE operation, the trigger body may need access to both the original and replacement data values.*
- *In the trigger code, access to the event data is available via the **OLD** and **NEW** trigger extension variables using dot notation to reference the column name.*

For instance, in an UPDATE event that changes the customer name column of a table row. the old and new customer_name values would be referenced as:

OLD.customer_name

NEW.customer_name

Triggers examples

*As an example, we can create a trigger to log when a row is added to the **supplier** database. The log table will be called **supplier_log** and consist of:*

- *a **primary key,***
- *the name of the database user*
- *the supplier name, and*
- *The event type and date.*

```
CREATE TABLE supplier_log (
        id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(35),
        supplier_name VARCHAR(25),
        event_type VARCHAR(25),
        event_date DATE
);
```

An AFTER INSERT example

```
CREATE TRIGGER after_supplier_insert
        AFTER INSERT ON supplier
        FOR EACH ROW
BEGIN
    INSERT INTO supplier_log (username, supplier_name, event_type,event_date)
    VALUES (USER(), NEW.company, 'INSERT', CURDATE());
END
```

We have used the USER() and CURDATE() functions within the code body to identify the database user's name and the current date, respectively.

A BEFORE INSERT example

*The BEFORE clause performs actions **before** changes are made to a database table. This allows us to perform actions such as validating or transforming the data before it is written to the table or preventing a deletion from proceeding under specific conditions. The following trigger, for example, converts the address field to uppercase when new records are inserted into the supplier table:*

```
CREATE TRIGGER before_supplier_insert
    BEFORE INSERT ON supplier
    FOR EACH ROW
BEGIN
    SET NEW.address = UPPER(NEW.address);
END
```

A BEFORE DELETE example

The BEFORE DELETE trigger is useful for ensuring that certain conditions are met before rows are deleted from a table. The following trigger uses a SIGNAL statement to prevent rows from being deleted from the supplier table:

```
CREATE TRIGGER before_supplier_delete
        BEFORE DELETE ON supplier
        FOR EACH ROW
BEGIN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Deletions are not allowed in the supplier table';
END
```

Displaying trigger information

Information about the triggers defined within a database can be displayed using the SHOW TRIGGERS statement.

The following statement lists all of the active triggers for the current database:

```
SHOW TRIGGERS\G
```

Deleting triggers

Triggers are deleted using the DROP TRIGGER statement, the syntax for which is as follows:

```
DROP TRIGGER <trigger_name>;
```