
Unidad 12. Triggers, procedimientos y funciones en MySQL

JOSÉ JUAN SÁNCHEZ HERNÁNDEZ

Índice general

1	Triggers, procedimientos y funciones en MySQL	3
1.1	Procedimientos	3
1.2	Funciones	4
1.3	Estructuras de control	4
1.3.1	Instrucciones condicionales	4
1.3.1.1	IF-THEN-ELSE	4
1.3.1.2	CASE	5
1.3.2	Instrucciones repetitivas o bucles	5
1.3.2.1	LOOP	5
1.3.2.2	REPEAT	6
1.3.2.3	WHILE	7
1.4	Manejo de errores	8
1.4.1	DECLARE ... HANDLER	8
1.4.2	Ejemplo 1 - DECLARE CONTINUE HANDLER	9
1.4.3	Ejemplo 2 - DECLARE EXIT HANDLER	10
1.5	Cómo realizar transacciones con procedimientos almacenados	11
1.6	Cursores	12
1.6.1	Operaciones con cursores	12
1.6.1.1	DECLARE	12
1.6.1.2	OPEN	12
1.6.1.3	FETCH	13
1.6.1.4	CLOSE	13
1.7	Triggers	15
1.8	Ejercicios	17
1.8.1	Procedimientos sin sentencias SQL	18
1.8.2	Procedimientos con sentencias SQL	18
1.8.3	Funciones sin sentencias SQL	20
1.8.4	Funciones con sentencias SQL	20
1.8.5	Manejo de errores en MySQL	21
1.8.6	Transacciones con procedimientos almacenados	21
1.8.7	Cursores	22
1.8.8	Triggers	23
1.9	Ejercicios de repaso	25
1.10	Recursos	31

2 Licencia

32

Capítulo 1

Triggers, procedimientos y funciones en MySQL

1.1 Procedimientos

```
CREATE
  [DEFINER = { user | CURRENT_USER }]
  PROCEDURE sp_name ([proc_parameter[,...]])
  [characteristic ...] routine_body

proc_parameter:
  [ IN | OUT | INOUT ] param_name type

func_parameter:
  param_name type

type:
  Any valid MySQL data type

characteristic:
  COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }

routine_body:
  Valid SQL routine statement
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.2 Funciones

```
CREATE
  [DEFINER = { user | CURRENT_USER }]
  FUNCTION sp_name ([func_parameter[,...]])
  RETURNS type
  [characteristic ...] routine_body

proc_parameter:
  [ IN | OUT | INOUT ] param_name type

func_parameter:
  param_name type

type:
  Any valid MySQL data type

characteristic:
  COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }

routine_body:
  Valid SQL routine statement
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.3 Estructuras de control

1.3.1 Instrucciones condicionales

1.3.1.1 IF-THEN-ELSE

```
IF search_condition THEN statement_list
  [ELSEIF search_condition THEN statement_list] ...
  [ELSE statement_list]
END IF
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.3.1.2 CASE

Existen dos formas de utilizar CASE:

```
CASE case_value
  WHEN when_value THEN statement_list
  [WHEN when_value THEN statement_list] ...
  [ELSE statement_list]
END CASE
```

o

```
CASE
  WHEN search_condition THEN statement_list
  [WHEN search_condition THEN statement_list] ...
  [ELSE statement_list]
END CASE
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.3.2 Instrucciones repetitivas o bucles

1.3.2.1 LOOP

```
[begin_label:] LOOP
  statement_list
END LOOP [end_label]
```

Ejemplo:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN
      ITERATE label1;
    END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END;
```

Ejemplo:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_loop$$
CREATE PROCEDURE ejemplo_bucle_loop(IN tope INT, OUT suma INT)
BEGIN
    DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    bucle: LOOP
        IF contador > tope THEN
            LEAVE bucle;
        END IF;

        SET suma = suma + contador;
        SET contador = contador + 1;
    END LOOP;
END
$$

DELIMITER ;
CALL ejemplo_bucle_loop(10, @resultado);
SELECT @resultado;
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.3.2.2 REPEAT

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

Ejemplo:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_repeat$$
CREATE PROCEDURE ejemplo_bucle_repeat(IN tope INT, OUT suma INT)
BEGIN
    DECLARE contador INT;

    SET contador = 1;
```

```
SET suma = 0;

REPEAT
SET suma = suma + contador;
SET contador = contador + 1;
UNTIL contador > tope
END REPEAT;
END
$$

DELIMITER ;
CALL ejemplo_bucle_repeat(10, @resultado);
SELECT @resultado;
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.3.2.3 WHILE

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

Ejemplo:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS ejemplo_bucle_while$$
CREATE PROCEDURE ejemplo_bucle_while(IN tope INT, OUT suma INT)
BEGIN
    DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    WHILE contador <= tope DO
        SET suma = suma + contador;
        SET contador = contador + 1;
    END WHILE;
END
$$

DELIMITER ;
CALL ejemplo_bucle_while(10, @resultado);
SELECT @resultado;
```

Puede encontrar más información en la [documentación oficial de MySQL](#).

1.4 Manejo de errores

1.4.1 DECLARE ... HANDLER

```
DECLARE handler_action HANDLER
    FOR condition_value [, condition_value] ...
    statement

handler_action:
    CONTINUE
| EXIT
| UNDO

condition_value:
    mysql_error_code
| SQLSTATE [VALUE] sqlstate_value
| condition_name
| SQLWARNING
| NOT FOUND
| SQLEXCEPTION
```

Las acciones posibles que podemos seleccionar como *handler_action* son:

- **CONTINUE**: La ejecución del programa continúa.
- **EXIT**: Termina la ejecución del programa.
- **UNDO**: No está soportado en MySQL.

Puede encontrar más información en la [documentación oficial de MySQL](#).

Ejemplo indicando el número de error de MySQL:

En este ejemplo estamos declarando un *handler* que se ejecutará cuando se produzca el error 1051 de MySQL, que ocurre cuando se intenta acceder a una tabla que no existe en la base de datos. En este caso la acción del *handler* es **CONTINUE** lo que quiere decir que después de ejecutar las instrucciones especificadas en el cuerpo del *handler* el procedimiento almacenado continuará su ejecución.

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
    -- body of handler
END;
```

Ejemplo para **SQLSTATE**:

También podemos indicar el valor de la variable **SQLSTATE**. Por ejemplo, cuando se intenta acceder a una tabla que no existe en la base de datos, el valor de la variable **SQLSTATE** es 42S02.

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'  
BEGIN  
    -- body of handler  
END;
```

Ejemplo para **SQLWARNING**:

Es equivalente a indicar todos los valores de **SQLSTATE** que empiezan con 01.

```
DECLARE CONTINUE HANDLER FOR SQLWARNING  
BEGIN  
    -- body of handler  
END;
```

Ejemplo para **NOT FOUND**:

Es equivalente a indicar todos los valores de **SQLSTATE** que empiezan con 02. Lo usaremos cuando estemos trabajando con cursores para controlar qué ocurre cuando un cursor alcanza el final del *data set*. Si no hay más filas disponibles en el cursor, entonces ocurre una condición de **NO DATA** con un valor de **SQLSTATE** igual a 02000. Para detectar esta condición podemos usar un *handler* para controlarlo.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND  
BEGIN  
    -- body of handler  
END;
```

Ejemplo para **SQLException::**

Es equivalente a indicar todos los valores de **SQLSTATE** que empiezan por 00, 01 y 02.

```
DECLARE CONTINUE HANDLER FOR SQLException  
BEGIN  
    -- body of handler  
END;
```

1.4.2 Ejemplo 1 - **DECLARE CONTINUE HANDLER**

```
-- Paso 1  
DROP DATABASE IF EXISTS test;  
CREATE DATABASE test;  
USE test;
```

```
-- Paso 2
CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));

-- Paso 3
DELIMITER $$
CREATE PROCEDURE handlerdemo ()
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
    SET @x = 1;
    INSERT INTO test.t VALUES (1);
    SET @x = 2;
    INSERT INTO test.t VALUES (1);
    SET @x = 3;
END
$$

DELIMITER ;
CALL handlerdemo();
SELECT @x;
```

¿Qué valor devolvería la sentencia `SELECT @x`?

1.4.3 Ejemplo 2 - DECLARE EXIT HANDLER

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));

-- Paso 3
DELIMITER $$
CREATE PROCEDURE handlerdemo ()
BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET @x = 1;
    SET @x = 1;
    INSERT INTO test.t VALUES (1);
    SET @x = 2;
    INSERT INTO test.t VALUES (1);
    SET @x = 3;
END
$$
```

```
DELIMITER ;  
CALL handlerdemo();  
SELECT @x;
```

¿Qué valor devolvería la sentencia `SELECT @x`?

1.5 Cómo realizar transacciones con procedimientos almacenados

Podemos utilizar el manejo de errores para decidir si hacemos `ROLLBACK` de una transacción. En el siguiente ejemplo vamos a capturar los errores que se produzcan de tipo `SQLEXCEPTION` y `SQLWARNING`.

Ejemplo:

```
DELIMITER $$  
CREATE PROCEDURE transaccion_en_mysql()  
BEGIN  
  DECLARE EXIT HANDLER FOR SQLEXCEPTION  
  BEGIN  
    -- ERROR  
    ROLLBACK;  
  END;  
  
  DECLARE EXIT HANDLER FOR SQLWARNING  
  BEGIN  
    -- WARNING  
    ROLLBACK;  
  END;  
  
  START TRANSACTION;  
  -- Sentencias SQL  
  COMMIT;  
END  
$$
```

En lugar de tener un manejador para cada tipo de error, podemos tener uno común para todos los casos.

```
DELIMITER $$  
CREATE PROCEDURE transaccion_en_mysql()  
BEGIN  
  DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING  
  BEGIN  
    -- ERROR, WARNING  
    ROLLBACK;  
  END;  
  
  START TRANSACTION;  
  -- Sentencias SQL  
  COMMIT;  
END  
$$
```

```
START TRANSACTION;  
-- Sentencias SQL  
COMMIT;  
END  
$$
```

1.6 Cursores

Los cursores nos permiten almacenar un conjunto de filas de una tabla en una estructura de datos que podemos ir recorriendo de forma secuencial.

Los cursores tienen las siguientes propiedades:

- *Asensitive*: The server may or may not make a copy of its result table.
- *Read only*: son de sólo lectura. No permiten actualizar los datos.
- *Nonscrollable*: sólo pueden ser recorridos en una dirección y no podemos saltarnos filas.

Cuando declaramos un cursor dentro de un procedimiento almacenado debe aparecer antes de las declaraciones de los manejadores de errores ([HANDLER](#)) y después de la declaración de variables locales.

1.6.1 Operaciones con cursores

Las operaciones que podemos hacer con los cursores son las siguientes:

1.6.1.1 DECLARE

El primer paso que tenemos que hacer para trabajar con cursores es declararlo. La sintaxis para declarar un cursor es:

```
DECLARE cursor_name CURSOR FOR select_statement
```

1.6.1.2 OPEN

Una vez que hemos declarado un cursor tenemos que abrirlo con [OPEN](#).

```
OPEN cursor_name
```

1.6.1.3 FETCH

Una vez que el cursor está abierto podemos ir obteniendo cada una de las filas con **FETCH**. La sintaxis es la siguiente:

```
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
```

Cuando se está recorriendo un cursor y no quedan filas por recorrer se lanza el error **NOT FOUND**, que se corresponde con el valor **SQLSTATE** «02000». Por eso cuando estemos trabajando con cursores será necesario declarar un *handler* para manejar este error.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND ...
```

1.6.1.4 CLOSE

Cuando hemos terminado de trabajar con un cursor tenemos que cerrarlo.

```
CLOSE cursor_name
```

Ejemplo:

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE t1 (
  id INT UNSIGNED PRIMARY KEY,
  data VARCHAR(16)
);

CREATE TABLE t2 (
  i INT UNSIGNED
);

CREATE TABLE t3 (
  data VARCHAR(16),
  i INT UNSIGNED
);

INSERT INTO t1 VALUES (1, 'A');
```

```
INSERT INTO t1 VALUES (2, 'B');

INSERT INTO t2 VALUES (10);
INSERT INTO t2 VALUES (20);

-- Paso 3
DELIMITER $$
DROP PROCEDURE IF EXISTS curdemo$$
CREATE PROCEDURE curdemo()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE a CHAR(16);
    DECLARE b, c INT;
    DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
    DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN cur1;
    OPEN cur2;

    read_loop: LOOP
        FETCH cur1 INTO b, a;
        FETCH cur2 INTO c;
        IF done THEN
            LEAVE read_loop;
        END IF;
        IF b < c THEN
            INSERT INTO test.t3 VALUES (a,b);
        ELSE
            INSERT INTO test.t3 VALUES (a,c);
        END IF;
    END LOOP;

    CLOSE cur1;
    CLOSE cur2;
END

-- Paso 4
DELIMITER ;
CALL curdemo();

SELECT * FROM t3;
```

Solución utilizando un bucle **WHILE**:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS curdemo$$
CREATE PROCEDURE curdemo()
```

```
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a CHAR(16);
  DECLARE b, c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur1;
  OPEN cur2;

  WHILE done = FALSE DO
    FETCH cur1 INTO b, a;
    FETCH cur2 INTO c;

    IF done = FALSE THEN
      IF b < c THEN
        INSERT INTO test.t3 VALUES (a,b);
      ELSE
        INSERT INTO test.t3 VALUES (a,c);
      END IF;
    END IF;
  END WHILE;

  CLOSE cur1;
  CLOSE cur2;
END;
```

1.7 Triggers

```
CREATE
  [DEFINER = { user | CURRENT_USER }]
  TRIGGER trigger_name
  trigger_time trigger_event
  ON tbl_name FOR EACH ROW
  [trigger_order]
  trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```


Un *trigger* es un objeto de la base de datos que está asociado con una tabla y que se activa cuando ocurre un evento sobre la tabla.

Los eventos que pueden ocurrir sobre la tabla son:

- **INSERT**: El *trigger* se activa cuando se inserta una nueva fila sobre la tabla asociada.
- **UPDATE**: El *trigger* se activa cuando se actualiza una fila sobre la tabla asociada.
- **DELETE**: El *trigger* se activa cuando se elimina una fila sobre la tabla asociada.

Ejemplo:

Crea una **base de datos** llamada `test` que contenga **una tabla** llamada `alumnos` con las siguientes columnas.

Tabla `alumnos`:

- `id` (entero sin signo)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres)
- `nota` (número real)

Una vez creada la tabla escriba **dos triggers** con las siguientes características:

- Trigger 1: `trigger_check_nota_before_insert`
 - Se ejecuta sobre la tabla `alumnos`.
 - Se ejecuta *antes* de una operación de *inserción*.
 - Si el nuevo valor de la nota que se quiere insertar es negativo, se guarda como 0.
 - Si el nuevo valor de la nota que se quiere insertar es mayor que 10, se guarda como 10.
- Trigger2: `trigger_check_nota_before_update`
 - Se ejecuta sobre la tabla `alumnos`.
 - Se ejecuta *antes* de una operación de *actualización*.
 - Si el nuevo valor de la nota que se quiere actualizar es negativo, se guarda como 0.
 - Si el nuevo valor de la nota que se quiere actualizar es mayor que 10, se guarda como 10.

Una vez creados los triggers escriba varias sentencias de inserción y actualización sobre la tabla `alumnos` y verifica que los *triggers* se están ejecutando correctamente.

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE alumnos (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(50) NOT NULL,
  apellido1 VARCHAR(50) NOT NULL,
  apellido2 VARCHAR(50),
  nota FLOAT
);
```

```
-- Paso 3
DELIMITER $$
DROP TRIGGER IF EXISTS trigger_check_nota_before_insert$$
CREATE TRIGGER trigger_check_nota_before_insert
BEFORE INSERT
ON alumnos FOR EACH ROW
BEGIN
    IF NEW.nota < 0 THEN
        set NEW.nota = 0;
    ELSEIF NEW.nota > 10 THEN
        set NEW.nota = 10;
    END IF;
END

DELIMITER $$
DROP TRIGGER IF EXISTS trigger_check_nota_before_update$$
CREATE TRIGGER trigger_check_nota_before_update
BEFORE UPDATE
ON alumnos FOR EACH ROW
BEGIN
    IF NEW.nota < 0 THEN
        set NEW.nota = 0;
    ELSEIF NEW.nota > 10 THEN
        set NEW.nota = 10;
    END IF;
END

-- Paso 4
DELIMITER ;
INSERT INTO alumnos VALUES (1, 'Pepe', 'López', 'López', -1);
INSERT INTO alumnos VALUES (2, 'María', 'Sánchez', 'Sánchez', 11);
INSERT INTO alumnos VALUES (3, 'Juan', 'Pérez', 'Pérez', 8.5);

-- Paso 5
SELECT * FROM alumnos;

-- Paso 6
UPDATE alumnos SET nota = -4 WHERE id = 3;
UPDATE alumnos SET nota = 14 WHERE id = 3;
UPDATE alumnos SET nota = 9.5 WHERE id = 3;

-- Paso 7
SELECT * FROM alumnos;
```

1.8 Ejercicios