PROGRAMACIÓN DE BASE DE DATOS

Primero DAW/DAM

Descripción breve

Primera toma de contacto con la programación de las Bases de Datos: Cursores, Gestión de errores, Triggers y Procedimientos almacenados

Antº Javier Miras Llamas

Tabla de contenido

Programación MySQL

La programación de la base de datos MySQL amplía la funcionalidad de hemos visto hasta ahora sobre la manipulación de estas mediante sentencias INSERT, UPDATE, DELETE, etc.

Una de las grandes ventajas que nos ofrece la programación de la base de datos es un mejor rendimiento en entornos de red cliente-servidor, reducir el tráfico y aumentar la productividad.

Conceptos básicos

Como para cualquier otro lenguaje de programación, debemos conocer las reglas de sintaxis que podemos utilizar, los diferentes elementos de que consta, los tipos de datos de los que disponemos, las estructuras de control que nos ofrece (tanto iterativas como condicionales) y cómo se realiza el manejo de los errores.

Unidad léxica

La programación en MySQL es no sensible a las mayúsculas, por lo que será equivalente escribir en mayúsculas o minúsculas, excepto cuando hablemos de literales.

Cada unidad léxica debe estar separada por espacios, por saltos de línea o por tabuladores para aumentar la legibilidad del código escrito.

```
IF (usuario = 'Javier') THEN
    SET correcto := TRUE;
ELSE
    SET correcto := FALSE;
END IF;
```

Las unidades léxicas se clasifican en:

- **Operadores[1]:** Conjunto de símbolos utilizados para realizar operaciones, delimitar comentarios, etc. En la siguiente tabla se muestra un resumen de estos.

Símbolo	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
%, MOD	Módulo
(	Apertura de lista
)	Cierre de lista
:	Variable de host
,	Separador de elementos
;	Terminado de sentencias
:=	Asignación
=	Asignación (Como parte de la sentencia SET)
&&, AND	Y lógica
||, OR	O lógica
!, NOT	Negación

Símbolo	Significado
LIKE	Coincidencia de patrón simple
NOT LIKE	Negación de coincidencia de patrón simple
IN ()	Valor dentro de un conjunto
IS NULL	Comprobación de valor nulo
IS NOT NULL	Comprobación de valor no nulo
=	Igualdad
<>, !=	Distinto
!=	Distinto
<	Menor
<=	Menor o igual
>	Mayor
>=	Mayor o igual
--	Comentario de una línea
/* */	Comentario de varias líneas

- **Identificadores:** Se utilizan para nombrar los elementos de los programas. Un identificador puede ser una palabra reservada: IF, ELSE, THEN, etc.; o uno creado por nosotros para dar nombre, por ejemplo, a una variable o una constante. En este caso se debemos tener en cuenta los siguientes aspectos:
  - Debe empezar por una letra seguido por letras, números, $, _ o #
  - No puede utilizarse una palabra reservada.

[1] MySQL :: MySQL 8.0 Reference Manual :: 12.4 Operators

- **Literales:** Son valores concretos que se utilizarán para asignarlos a las variables o constantes. Dentro de los literales tenemos:
  - Números: Enteros y reales. Por ejemplo: 3, 2.75, 8.1e35.
  - Hexadecimales: Valores en hexadecimal, va precedido por una equis (x). Por ejemplo: 0x36.
  - Cadena de caracteres: Secuencia de caracteres. Puede ir entre comillas simples (')o comillas dobles ("). Por ejemplo: 'Esto es una cadena de caracteres', "Esto es otra cadena de caracteres".
  - Boleanos: Valor booleano TRUE o FALSE. El valor verdadero se evalúa como 1, mientras que el valor falso se evalúa como 0.
  - NULL: El valor null significa "No hay dato". No confundir el valor null con un 0 o cadena vacía. Recordad que la gestión de este tipo de dato se debe hacer con "IS NULL", "IS NOT NULL" o la función IFNULL().
- **Comentarios:** Para los comentarios de una única línea utilizaremos el doble guión (--), mientras que los comentarios de más de una línea se acotarán entre los delimitadores /* y */.

Tipos de datos

Los tipos de datos[2] que podemos utilizar en la programación de la base de datos son los mismo que hemos visto en la manipulación de esta.

- **Numéricos:** BIT, TINYINT, SMALLINT, MEDIUMINT, INT, INTEGER, BIGINT, DECIMAL, etc.
- **Alfanuméricos:** CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM y SET.
- **Booleanos:** Recordad que este valor es sinónimo de TINYINT(1). El valor falso es 0 y cualquier otro valor será verdadero .
- **Fecha/hora:** DATE, TIME, DATETIME, TIMESTAMP y YEAR.

[2] MySQL :: MySQL 8.0 Reference Manual :: 11 Data Types

Unidad básica: El bloque

La unidad básica de programación es el bloque, que está compuesto por los siguientes elementos de código:

- BEGIN.. END: Bloque de sentencias ejecutables.
- DECLARE: Declaración de variables y constantes.

```
BEGIN
    DECLARE contador INT DEFAULT 1;

    WHILE (Contador <= n) DO
        SET contador := contador + 1;
    END WHILE;
END;
```

Ejercicios

Ejercicio 1. Dado el siguiente código descomponlo en sus diferente unidades léxicas.

```
IF (a = b) THEN
    SET iguales := TRUE; -- Iguales
ELSE
    SET iguales := FALSE; -- Diferentes
END IF;
```

Ejercicio 2. Indica si son correctas o no las siguientes sentencias. Razona la respuesta.

```
DECLARE entero INT;
DECLARE entero1 INT;
DECLARE entero FLOAT;
DECLARE real FLOAT;
DECLARE cadena VARCHAR(20);
DECLARE cp CHAR(5);
DECLARE cadena CHAR(5);
```

Estructuras de control

IF .. THEN

La sentencia IF..THEN tiene la siguiente estructura:

```
IF SEARCH_CONDITION THEN
        STATEMENT_LIST
[ELSEIF SEARCH_CONDITION THEN
        STATEMENT_LIST] ...
[ELSE
        STATEMENT_LIST]
END IF
```

CASE

La sentencia CASE es equivalente a switch y sigue la siguiente estructura:

```
CASE CASE_VALUE
        WHEN WHEN_VALUE THEN STATEMENT_LIST
        [WHEN WHEN_VALUE THEN STATEMENT_LIST] ...
        [ELSE STATEMENT_LIST]
END CASE
```

O bien:

```
CASE
        WHEN SEARCH_CONDITION THEN STATEMENT_LIST
        [WHEN SEARCH_CONDITION THEN STATEMENT_LIST] ...
```

```
        [ELSE STATEMENT_LIST]
    END CASE
```

LOOP

La sentencia LOOP es equivalente a while. Tiene la siguiente estructura:

```
    [BEGIN_LABEL:] LOOP
        STATEMENT_LIST
    END LOOP [END_LABEL]
```

WHILE

La sentencia WHILE tiene la siguiente estructura:

```
    [BEGIN_LABEL:] WHILE SEARCH_CONDITION DO
        STATEMENT_LIST
    END WHILE [END_LABEL]
```

REPEAT

La sentencia REPEAT es equivalente a do..while y tiene la siguiente estructura:

```
    [BEGIN_LABEL:] REPEAT
        STATEMENT_LIST
    UNTIL SEARCH_CONDITION
    END REPEAT [END_LABEL]
```

Ejemplos

Ejemplo 1. IF..THEN

```
DECLARE n INT DEFAULT 4;
DECLARE m INT DEFAULT 5;
DECLARE s VARCHAR(20);

IF (n > m) THEN
    SET s := '>';
ELSEIF (n = m) THEN
    SET s := '=';
ELSE
    SET s := '<';
END IF;

SET s := CONCAT(n, ' ', s, ' ', m);  -- 4 < 5
```

Ejemplo 2. CASE

```
DECLARE valor INT DEFAULT 1;

CASE valor
    WHEN 2 THEN
        SELECT valor;

    WHEN 3 THEN
        SELECT 0;

    ELSE
        BEGIN
        /*
            Resto de instrucciones
        */
        END;
END CASE;
```

Ejemplo 3. LOOP

```
DECLARE contador INT DEFAULT 0;

repetir: LOOP
    SET contador := contador + 1;

    IF (contador < 10) THEN
        ITERATE repetir;              -- Reiteramos
    END IF;

    LEAVE repetir;                    -- Salimos
END LOOP repetir;
```

Ejemplo 4. WHILE

```
DECLARE contador INT DEFAULT 5;

WHILE (contador > 0) DO

    /*
        Resto de las sentencias
    */

    SET contador := contador - 1;
END WHILE;
```

Ejemplo 5. REPEAT

```
DECLARE contador INT DEFAULT 0;

REPEAT

    /*
        Resto de las sentencias
    */

    SET contador := contador + 1;
UNTIL (contador > 5)
END REPEAT;
```

Ejercicios

Ejercicio 3. ¿Qué bucle es el más apropiado para ejecutar sentencias por lo menos una vez?

Ejercicio 4. ¿Cuál de los siguientes bloques es correcto?

a)
```
LOOP bucle:
     -- Sentencias
END bucle:
```

b)
```
bucle: LOOP
    -- Sentencias
END LOOP bucle;
```

c)
```
bucle: LOOP bucle;
    -- Sentencias
END bucle LOOP;
```

Ejercicio 5. Crea una estructura de código que sume los diez primeros número utilizando WHILE.

Ejercicio 6. Crea una estructura de código que sume los diez primeros número utilizando REPEAT.

Ejercicio 7. ¿Cuál de los siguientes bucles no está soportado por MySQL?

a) LOOP
b) REPEAT
c) FOR
d) WHILE

GESTIÓN DE ERRORES

Para gestionar los errores y las excepciones que se nos puedan plantear a la hora de ejecutar un procedimiento y disparar un trigger utilizaremos la sentencia DECLARE.. HANDLER.

DECLARE .. HANDLER

La sentencia DECLARE.. HANDLER especifica un controlador que se encarga de una o más condiciones. Si ocurriese una de las condiciones especificadas, se ejecutaría las instrucciones indicadas.

Su estructura es la siguiente:

```
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
  }
```

Veamos cada una de las partes de la sentencia de creación:

- HANDLER_ACTION. Indica qué acción se llevará a cabo.
  - CONTINUE. La ejecución del programa actual continúa.
  - EXIT. La ejecución del programa termina.
  - UNDO. No está soportada por MySQL.
- CONDITION_VALUE. Indica la condición o conjunto de condiciones que activan el controlador.
  - MYSQL_ERROR_CODE[3]. Entero que indica el código de error producido. No se debe utilizar el código 0 porque indica la ausencia de error.
  - SQLSTATE [VALUE] SQLSTATE_VALUE[3]. Cadena de 5 caracteres que indica un valor de SQLSTATE. No se debe utilizar los valores que empiezan por '00' porque indican la ausencia de error.
  - CONDITION_NAME. Condición previamente especificada con DECLARE .. CONDITION.
  - SQLWARNING. Alias para el valor de las clases SQLSTATE que empiezan por '01'.
  - NOT FOUND. Alias para el valor de las clases SQLSTATE que empiezan por '02'. Se utilizará, sobre todos, con los cursores para indicar que se ha llegado al final y no hay más filas disponibles.
  - SQLEXCEPTION. Alias para el valor de las clases que no empiezan por '00', '01' y '02'.

Ejemplos

Ejemplo 1. El siguiente controlador se ejecutará cuando no exista la tabla (número de error 1051) a la que se intenta acceder. Se interrumpirá la ejecución del procedimiento almacenado.

```
DECLARE EXIT HANDLER FOR 1051
```

[3] https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html

```
BEGIN
  -- Sentencias del controlador
END;
```

Ejemplo 2. El siguiente controlador se ejecutará cuando no exista la tabla (SQLSTATE 42S02) a la que se intenta acceder. Se interrumpirá la ejecución del procedimiento almacenado.

```
DECLARE EXIT HANDLER FOR SQLSTATE '42S02'
BEGIN
  -- Sentencias del controlador
END;
```

Ejemplo 3. El siguiente controlador se ejecutará cuando se produzca un SQLWARNING. Se continuará la ejecución del procedimiento almacenado.

```
DECLARE CONTINUE HANDLER FOR SQLWARNING
BEGIN
  -- Sentencias del controlador
END;
```

Ejemplo 4. El siguiente ejemplo utiliza un controlador para gestionar el SQLSTATE '23000', que ocurre cuando se produce el error de clave duplicada.

```
CREATE PROCEDURE DemoControlador()
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'
    BEGIN
        SET @x := 1;
    END;

    SET @x := 1;

    INSERT INTO categorias VALUES (1, 'Error');

    SET @x := 2;
END;
```

Al hacer la inserción se produce un error. Saltará la excepción y se captura por el controlador, como la condición es de continuar (CONTINUE) se sigue ejecutando el procedimiento. El valor final de @x es 2.

Ejemplo 5. El siguiente ejemplo utiliza un controlador para gestionar el SQLSTATE '23000', que ocurre cuando se produce el error de clave duplicada.

```
CREATE PROCEDURE DemoControlador()
BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE '23000'
    BEGIN
        SET @x := -1;
    END;

    SET @x := 1;

    INSERT INTO categorias VALUES (1, 'Error');

    SET @x := 2;
END;
```

Al hacer la inserción se produce un error. Saltará la excepción y se captura por el controlador, como la condición es de terminar (EXIT) se interrumpe la ejecutando el procedimiento. El valor final de @x es -1.

CURSORES

Los cursores es una estructura que almacena un conjunto de filas devuelto por una sentencia SELECT. Los cursores tienen las siguientes características:

- Asensitive: El servidor puede o no hacer una copia de los resultados.
- Read only: Es de sólo lectura y no se puede actualizar.
- Nonscrollable: Sólo se puede desplazar en una dirección y no pueden saltarse filas.

DECLARE

Para definir un cursor utilizaremos la sentencia DECLARE. Tienen la siguiente estructura:

```
DECLARE CURSOR_NAME CURSOR FOR SELECT_STATEMENT
```

Consideraciones sobre los cursores:

- La sentencia SELECT no puede tener la cláusula INTO.
- Los cursores deben aparecer después de las declaraciones de las variables y antes de la declaración de gestión del cursor.
- El nombre de cada cursor debe ser único. En un mismo bloque de código no puede haber dos cursores con el mismo nombre.
- El número de columnas de la sentencia SELECT tiene que ser el mismo que el de la sentencia FETCH.

OPEN

La sentencia OPEN abre un cursor previamente declarado. Su estructura es la siguiente:

```
OPEN CURSOR_NAME
```

FETCH

La sentencia FETCH obtiene una fila de la sentencia SELECT del cursor y avanza a la siguiente fila. Su estructura es la siguiente:

```
FETCH [[NEXT] FROM] CURSOR_NAME INTO VAR_NAME [, VAR_NAME] ...
```

Si existe la fila las columnas se almacenan en las variables indicadas, por este motivo el número de columnas de la sentencia SELECT tiene que ser igual a las columnas extraídas.

Si no hay mas filas, saltará la condición No Data (SQLSTATE con valor '02000'). Para gestionar esta situación utilizaremos la condición NOT FOUND.

CLOSE

La sentencia CLOSE cierra un cursor abierto. Su estructura es la siguiente:

```
CLOSE CURSOR_NAME
```

En el caso de que el cursor no estuviese abierto se produciría un error.

Si no se cierra un cursor, este se cerrará automáticamente cuando se finalice el bloque donde fue declarado.

Ejemplo

```
CREATE PROCEDURE CursorDemo()
BEGIN
    DECLARE fin INT DEFAULT FALSE;
    DECLARE cadena CHAR(16);
    DECLARE entero_1, entero_2 BIGINT;

    -- Declaramos los cursores
    DECLARE cursor1 CURSOR FOR SELECT entero, dato FROM tabla1;
    DECLARE cursor2 CURSOR FOR SELECT entero FROM tabla2;
```

```
    -- Declaramos el final de la búsqueda en los cursores
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    BEGIN
       SET fin = TRUE;
    END;

    -- Abrimos los cursores
    OPEN cursor1;
    OPEN cursor2;

    bucle_lectura: LOOP

        -- Leemos una fila y posicionamos el cursor en la siguiente fila.
        FETCH cursor1 INTO entero_1, cadena;
        FETCH cursor2 INTO entero_2;

        IF fin THEN
            LEAVE bucle_lectura;
        END IF;

        IF (entero_1 >= entero_2) THEN
            INSERT INTO tabla3 VALUES (entero_1, cadena);
        ELSE
            INSERT INTO tabla3 VALUES (entero_2, cadena);
        END IF;
    END LOOP;

    -- Cerramos los cursores
    CLOSE cursor1;
    CLOSE cursor2;
END;
```

Ejercicios

Ejercicio 8. ¿Se puede modificar un dato obtenido de un cursor?

Ejercicio 9. ¿Se puede acceder aleatoriamente a los datos de un cursor?

TRIGGERS (Disparadores)

Un TRIGGER es un programa almacenado en la base de datos que se ejecuta automáticamente (dispara) en respuesta a los comandos INSERT, UPDATE y DELETE.

Los TRIGGERS se utilizan, normalmente, para:

- Mantener la integridad de los datos. Por ejemplo, cuando se hace una inserción o actualización de un artículo verificar que una columna tenga un valor determinado: cantidad negativa, no tenga precio, etc.
- Ejecutar acciones de forma implícita. Por ejemplo, descontar el stock de un producto cuando se ha hecho una venta.
- Obtener valores de otras tablas. Por ejemplo, verificar si el tipo de IVA es el que actualmente está vigente.

CREATE

La sentencia de creación de un TRIGGER es la siguiente:

```
CREATE [DEFINER = USER | CURRENT_USER] TRIGGER TRIGGER_NAME
    { BEFORE | AFTER }
    { INSERT | UPDATE | DELETE }
    ON TABLE_NAME FOR EACH ROW
    [{ FOLLOWS | PRECEDES } OTHER_ TRIGGER_NAME]
    TRIGGER_BODY
```

Veamos cada una de las partes de la sentencia de creación:

- DEFINER = USER | CURRENT_USER. Este parámetro es opcional. Con él indicamos que usuario tiene privilegios para que se ejecute el trigger en respuesta a un evento. Por defecto, tendrá el valor CURRENT_USER.
- TRIGGER_NAME. Nombre del trigger. Es aconsejable utilizar la siguiente nomenclatura: nombredelatabla_operacion_trigger, por ejemplo: productos_BI_trigger, clientes_AI_trigger, etc.

- BEFORE | AFTER. Se indica si se ejecuta antes o después del evento que dispara el trigger.
- INSERT | UPDATE | DELETE. Evento que dispara el trigger.
- TABLE_NAME. Nombre de la tabla asociada al trigger.
- { FOLLOWS | PRECEDES } OTHER_ TRIGGER_NAME. Este parámetro es opcional. Indica en qué orden se ejecuta este trigger en relación a otro trigger asociado al mismo evento de la tabla. Con , el nuevo trigger se disparará después del trigger OTHER_ TRIGGER_NAME. Con PRECEDES, se disparará antes que OTHER_ TRIGGER_NAME.
- TRIGGER_BODY. Secuencia de instrucciones que se ejecutarán cuando se dispare el trigger.

OLD y NEW

Los identificadores OLD y NEW los utilizaremos para acceder a los valores de una columna.

- OLD indica el valor antiguo de la columna.
- NEW el nuevo valor que puede tomar la columna.

En función del evento que dispara el trigger podremos utilizar uno u otro:

- INSERT. Sólo se puede utilizar ONEW.
- UPDATE. Se utilizará tanto OLD como NEW.
- DELETE. Sólo se puede utilizar OLD.

DROP

Para eliminar un trigger de nuestra base de datos utilizaremos el comando DROP. Su estructura es la siguiente:

```
DROP TRIGGER [IF EXISTS] TRIGGER_NAME
```

SHOW

Podemos ver los triggers que tenemos en nuestra base de datos utilizando el comando SHOW.

```
SHOW TRIGGERS
```

Si se quisiese obtener información de un trigger en concreto la sentencia SHOW sería la siguiente:

```
SHOW CREATE TRIGGER TRIGGER_NAME
```

Ejemplos

Ejemplo 1. El siguiente trigger verifica si un precio es negativo, en cuyo caso lo pondrá a cero.

```
CREATE DEFINER=CURRENT_USER TRIGGER productos_BI_trigger BEFORE INSERT ON productos FOR EACH ROW
BEGIN
    -- Instrucciones asociadas al trigger
    IF (NEW.precio < 0) THEN
        SET New.precio := 0;
    END IF;
END
```

Ejemplo 2. El siguiente trigger almacena la tabla log los cambios producidos en la actualización de la tabla productos.

```
CREATE DEFINER=CURRENT_USER TRIGGER productos_AU_trigger AFTER UPDATE ON productos FOR EACH ROW
BEGIN
    INSERT INTO log (fecha, usuario, descripcion)
    VALUES (NOW(),
            USER(),
            CONCAT('UPDATE PRODUCTOS: ',
                   ' OLD (', OLD.producto, ', ' , OLD.precio, ')',
                   ' NEW (', NEW.producto, ', ' , NEW.precio,')'));
END
```

Ejercicios

Ejercicio 10. Indica dos utilizades de un trigger.

Ejercicio 11. Crea un trigger en la tabla artículos de la base de datos Q3ERP para que cuando se inserte un artículo verifique si el precio es positivo. En caso contrario pondrá el precio a cero.

Ejercicio 12. Crea un trigger en la tabla artículos de la base de datos Q3ERP para que cuando se modifique un artículo verifique si el precio es positivo. En caso contrario pondrá el precio a cero.

Ejercicio 13. Crea un trigger en todas las tablas de la base de datos Q3ERP que cada vez que se modifique algo se guarde la modificación en la tabla log.

Ejercicio 14 Crea un trigger en todas las tablas de la base de datos Q3ERP para que cada vez que se borre un registro se guarde la información en la tabla log.

PROCEDURES (Procedimientos)

Un procedimiento es un conjunto de instrucciones que guardaremos en nuestra base de datos para su posterior ejecución. Los procedimientos, a diferencia de las funciones, no devuelven ningún valor.

CREATE

La sentencia de creación de un PROCEDURE es la siguiente:

```
CREATE [DEFINER = USER] PROCEDURE SP_NAME ([{ IN | OUT | INOUT} PARAM_NAME TYPE [, ...]])
[{ COMMENT 'STRING'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }]
ROUTINE_BODY
```

Veamos cada una de las partes de la sentencia de creación:

- DEFINER = USER. Este parámetro es opcional. Con él indicamos que usuario tiene privilegios para que se ejecute el trigger en respuesta a un evento.
- SP_NAME. Nombre del procedimiento.
- IN | OUT | INOUT. Clase de parámetro, pude ser de entrada (IN), salida (OUT) o entrada y salida (INOUT). Si no se indica nada el parámetro será de entrada.
- PARAM_NAME. Nombre del parámetro.
- TYPE. Cualquier tipo de dato válido en MySQL.
- COMMENT 'STRING'. Comentario sobre el procedimiento.

- LANGUAGE SQL. Indica que el lenguaje utilizado para el procedimiento es SQL. En un futuro se podrían utilizar otro tipo de lenguajes como PHP, Java, etc.
- [NOT] DETERMINISTIC. Un procedimiento es determinista (DETERMINISTIC) si siempre produce el mismo resultado para los mismos parámetros de entrada, por el contrario, será no determinista (NOT DETERMINISTIC) si para los mismos parámetros de entrada da distinto resultado. El valor por defecto es no determinista.
- CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA. Determina la estructura del procedimiento. CONTAINS SQL es el tipo por defecto e indique que contiene sentencias SQL. NO SQL indica que no contiene sentencias SQL. READS SQL DATA especifica que el procedimiento lee datos, pero no los modifica. MODIFIES SQL DATA especifica que el procedimiento escribe, modifica o borra datos.
- SQL SECURITY { DEFINER | INVOKER }. Indica el nivel de seguridad del procedimiento. SQL SECURITY DEFINER indica que el procedimiento se ejecutará con los permisos del usuario que lo creó. Si se indica SQL SECURITY INVOKER, se ejecutará con los permisos del usuario que ejecuta el procedimiento. El valor por defecto es SQL SECURITY DEFINER.
- ROUTINE_BODY. Sentencias que ejecutará el procedimiento.

CALL

Para invocar un procedimiento se utilizará el comando CALL. Su estructura es la siguiente:

```
CALL SP_NAME([PARAM_NAME [, ...]])
```

DROP

Para borrar un procedimiento se utilizará el comando DROP. Su estructura es la siguiente:

```
DROP PROCEDURE [IF EXISTS] SP_NAME
```

Ejemplos

Ejemplo 1. El siguiente procedimiento efectúa la suma de dos números enteros que se le pasan como parámetro de entrada, a y b, y el resultado lo guarda en el parámetro de salida c.

```
CREATE DEFINER=CURRENT_USER PROCEDURE Sumar(IN a INT, IN b INT, OUT c INT)
    NO SQL
    DETERMINISTIC
    COMMENT 'Suma dos valores enteros'
BEGIN
    SET c := a + b;
END
```

Ejemplo 2. El siguiente procedimiento realiza una secuencia de transacciones gestiona los errores que se pudiesen producir.

```
CREATE PROCEDURE TransaccionEnMySQL()
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN

        -- ERROR, WARNING

        ROLLBACK;
    END;

    START TRANSACTION;

        -- Sentencias SQL

    COMMIT;
END
```

Ejercicios

Ejercicio 15. Crea un procedimiento que tenga como parámetro de entrada un número real y como parámetro de salida una cadena de caracteres. El parámetro de salida indicará si el número es positivo, negativo o cero.

Ejercicio 16. Escribe un procedimiento que reciba como entrada la nota de un alumno (numérico real) y un parámetro de salida (cadena de caracteres) con las siguientes condiciones:

- [0, 5): Insuficiente
- [5, 6): Aprobado
- [6, 7): Bien
- [7, 9): Notable
- [9, 10]: Sobresaliente
- En cualquier otro caso la nota no será válida.

Ejercicio 17. Resuelva el procedimiento diseñado en el ejercicio anterior haciendo uso de la estructura de control CASE.

Ejercicio 18. Escribe un procedimiento que reciba como parámetro de entrada un valor numérico que represente un día de la semana y que devuelva una cadena de caracteres con el nombre del día de la semana correspondiente. Por ejemplo, para el valor de entrada 1 devolverá lunes, para el 2 devolverá martes, etc.

Ejercicio 19. Crea una tabla llamada Fibonacci. Inserta los 50 primeros términos de la serie mediante un procedimiento almacenado. Utiliza la instrucción REPEAT.

Ejercicio 20. Repite el ejercicio anterior, pero esta vez utiliza la instrucción WHILE.

Ejercicio 21. Repite el ejercicio anterior, pero modifícalo para pasarle por parámetro la cantidad de términos de la serie.

FUNCTION (Funcciones)

Una función es un conjunto de instrucciones que guardaremos en nuestra base de datos para su posterior ejecución. Las funciones, a diferencia de los procedimientos, devuelven un valor.

CREATE

La sentencia de creación de una FUNCTION es la siguiente:

```
CREATE [DEFINER = USER] FUNCTION SP_NAME ([{ IN | OUT | INOUT} PARAM_NAME TYPE [, ...]]) RETURNS TYPE
[{ COMMENT 'STRING'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }]
ROUTINE_BODY
```

Las partes de la sentencia de creación son las mismas que para el procedimiento, excepto que se añade RETURNS indicando el tipo de dato que se devuelve.

DROP

Para borrar una FUNCTION se utilizará el comando DROP. Su estructura es la siguiente:

```
DROP FUNCTION [IF EXISTS] SP_NAME
```

Ejemplos

Ejemplo 1. La siguiente función devuelve una cadena indicando la comparación de los parámetros introducidos.

```
CREATE FUNCTION Comparacion(n INT, m INT) RETURNS VARCHAR(20)
BEGIN
    DECLARE s VARCHAR(20);

    IF (n > m) THEN
        SET s := '>';
    ELSEIF (n = m) THEN
        SET s := '=';
    ELSE
        SET s := '<';
    END IF;

    SET s := CONCAT(n, ' ', s, ' ', m);

    RETURN s;
END
```

Ejemplo 2. La siguiente función Estrellas crea una cadena de asteriscos, para simular la calidad de un producto.

```
CREATE DEFINER=CURRENT_USER FUNCTION Estrellas(calidad tinyint) RETURNS char(5) CHARSET utf8mb4
    DETERMINISTIC
BEGIN
    DECLARE est CHAR(5) DEFAULT '';
    DECLARE i INT DEFAULT 0;
    DECLARE saveCalidad INT DEFAULT calidad;

    IF (saveCalidad > 5) THEN
        SET saveCalidad := 5;
    END IF;

    IF (saveCalidad < 0) THEN
        SET saveCalidad := 0;
    END IF;

    WHILE (i < saveCalidad) DO
        SET est := CONCAT(est, '*');
        SET I := I + 1;
    END WHILE;

    RETURN est;
```

```
END
```

La llamada a esta función se puede ver en el siguiente ejemplo.

```
CREATE DEFINER=CURREN_USER TRIGGER productos_BU_trigger BEFORE UPDATE ON productos FOR EACH ROW
BEGIN
    SET NEW.estrellas := Estrellas(NEW.calidad);

    -- Resto de sentencias.
END
```

Ejercicios

Ejercicio 22. Crea una función que tenga como parámetro de entrada un número real y devuelva una cadena de caracteres indicando si el número es positivo, negativo o cero.

Ejercicio 23. Crea una función que devuelva la raíz cuadrada de un valor real introducido como parámetro. Si la no se calcular la raíz cuadrada devolverá -1.

Ejercicio 24. Crea una función que resuelva una ecuación de segundo grado.

Ejercicio 25. Crea una función que devuelva el número de años transcurridos entre dos fechas.

Ejercicio 26. Escribe una función para la base de datos Q3ERP que devuelva el número total de productos que hay en la tabla productos.

Ejercicio 27. Escribe una función para la base de datos Q3ERP que devuelva el valor medio del precio de los productos de una determinada marca que se recibirá como parámetro de entrada.

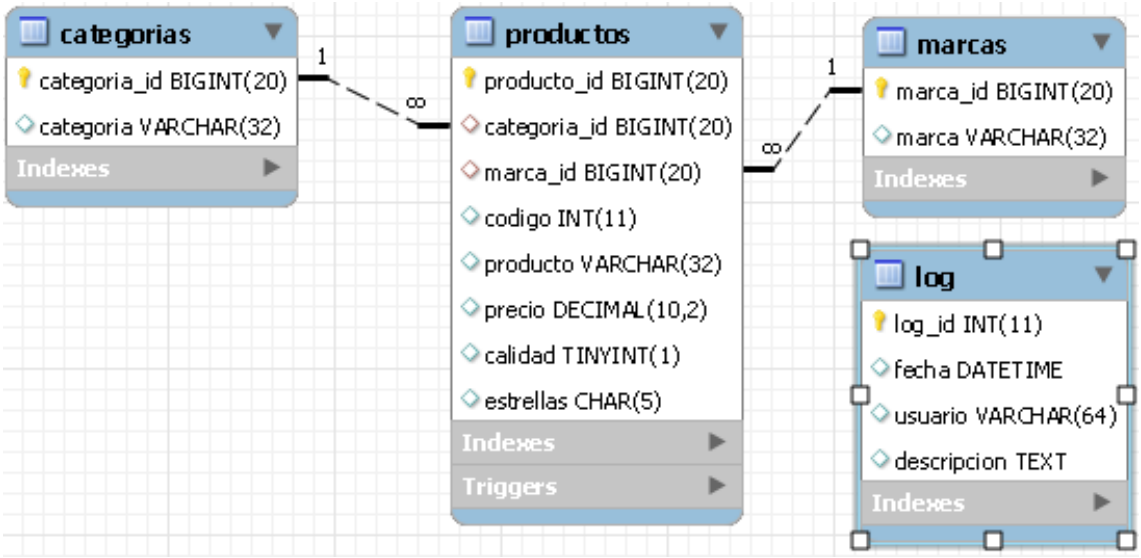
Ejercicio 28. Escribe una función para la base de datos Q3ERP que devuelva el valor mínimo del precio de los productos de una determinada marca que se recibirá como parámetro de entrada.

Base de datos

La base de datos la vamos a crear en MySQL. Se llamará Q3ERP y tendrá las siguientes tablas:

Tabla	Campos	Tipo	Descripción
categorias	categoria_id	Entero (PK)	Clave primaria
	categoria	Cadena (32)	Nombre de la categoría
marcas	marca_id	Entero (PK)	Clave primaria
	marca	Cadena (32)	Nombre de la marca
productos	producto_id	Entero (PK)	Clave primaria
	categoria_id	Entero (FK)	Clave ajena a la tabla categorías
	marca_id	Entero (FK)	Clave ajena a la tabla marcas
	producto	Cadena (32)	Nombre del producto
	precio	Decimal (10,2)	Precio del producto
	calidad	Tinyint(1)	Calidad del producto (1..5)
	estrellas	Char(5)	Simulación de la calidad
log	log_id	Entero (PK)	Clave primaria
	fecha	Fecha/hora	Fecha de la inserción
	usuario	Cadena(64)	Usuario que hace la inserción
	descripcion	Texto	Texto con la acción realizada

El esquema entidad/relación es de la siguiente forma:



Ejecutamos el siguiente script con las instrucciones SQL necesarias para la creación de la base de datos.

```
-- Creación de la base de datos

CREATE DATABASE Q3ERP;

USE Q3ERP;

-- Creación de las tablas

CREATE TABLE `categorias` (
  `categoria_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `categoria` varchar(32) CHARACTER SET utf8 DEFAULT NULL,

  PRIMARY KEY (`categoria_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

CREATE TABLE `marcas` (
  `marca_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `marca` varchar(32) CHARACTER SET utf8 DEFAULT NULL,

  PRIMARY KEY (`marca_id`)
```

```
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

CREATE TABLE `productos` (
  `producto_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `categoria_id` bigint(20) DEFAULT NULL,
  `marca_id` bigint(20) DEFAULT NULL,
  `codigo` int(11) DEFAULT NULL,
  `producto` varchar(32) CHARACTER SET utf8 DEFAULT NULL,
  `precio` decimal(10,2) DEFAULT NULL,
  `calidad` tinyint(1) DEFAULT NULL,
  `estrellas` char(5) DEFAULT NULL,

  PRIMARY KEY (`producto_id`),
  KEY `fk_productos_categorias_idx` (`categoria_id`),
  KEY `fk_productos_marcas_idx` (`marca_id`),

  CONSTRAINT `fk_productos_categorias` FOREIGN KEY (`categoria_id`) REFERENCES `categorias` (`categoria_id`) ON UPDATE
CASCADE,
  CONSTRAINT `fk_productos_marcas` FOREIGN KEY (`marca_id`) REFERENCES `marcas` (`marca_id`) ON UPDATE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

CREATE TABLE `log` (
  `log_id` int(11) NOT NULL AUTO_INCREMENT,
  `fecha` datetime DEFAULT NULL,
  `usuario` varchar(64) DEFAULT NULL,
  `descripcion` text,
  PRIMARY KEY (`log_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

-- Inserción de datos
INSERT INTO categorias VALUES
  (''),
  ('Portátiles'),
  ('PCs escritorio'),
  ('Impresoras'),
  ('Monitores'),
  ('Teclados'),
  ('Tarjetas vídeo'),
  ('Altavoces'),
  ('Micrófonos');
GO

INSERT INTO marcas VALUES
```

```
  (''),
  ('Brother'),
  ('HP'),
  ('LG'),
  ('Logitech'),
  ('Lenovo'),
  ('Asus'),
  ('Dell'),
  ('Sansumg'),
  ('Gygabyte'),
  ('Epson'),
  ('Nvidia');
GO
```