Unidad 12. Triggers, procedimientos y funciones en MySQL

ASIR / DAW - (Gestión de) Bases de datos

JOSÉ JUAN SÁNCHEZ HERNÁNDEZ

IES Celia Viñas (Almería) - 2022/2023

Índice general

1 Triggers, procedimientos y funciones en MySQL **3**

1.1 Procedimientos . . . 3
1.1.1 Sintaxis . . . 3
1.1.2 DELIMITER . . . 4
1.1.3 Parámetros de entrada, salida y entrada/salida . . . 4
1.1.4 Ejemplo de un procedimiento con parámetros de entrada . . . 5
1.1.5 Llamada de procedimientos con CALL . . . 6
1.1.6 Ejemplos de procedimientos con parámetros de salida . . . 6
1.2 Funciones . . . 8
1.2.1 Sintaxis . . . 8
1.2.2 Parámetros de entrada . . . 9
1.2.3 Resultado de salida . . . 9
1.2.4 Características de la función . . . 10
1.2.5 Declaración de variables locales con DECLARE . . . 11
1.2.6 Ejemplos . . . 11
1.3 Estructuras de control . . . 12
1.3.1 Instrucciones condicionales . . . 12
1.3.1.1 IF-THEN-ELSE . . . 12
1.3.1.2 CASE . . . 12
1.3.2 Instrucciones repetitivas o bucles . . . 13
1.3.2.1 LOOP . . . 13
1.3.2.2 REPEAT . . . 14
1.3.2.3 WHILE . . . 14
1.4 Manejo de errores . . . 15
1.4.1 DECLARE ... HANDLER . . . 15
1.4.2 Ejemplo 1 - DECLARE CONTINUE HANDLER . . . 17
1.4.3 Ejemplo 2 - DECLARE EXIT HANDLER . . . 18
1.5 Cómo realizar transacciones con procedimientos almacenados . . . 18
1.6 Cursores . . . 19
1.6.1 Operaciones con cursores . . . 20
1.6.1.1 DECLARE . . . 20
1.6.1.2 OPEN . . . 20
1.6.1.3 FETCH . . . 20
1.6.1.4 CLOSE . . . 20

1.7 *Triggers* . . . 23
1.8 Ejercicios . . . 25
1.8.1 Procedimientos sin sentencias SQL . . . 25
1.8.2 Procedimientos con sentencias SQL . . . 26
1.8.3 Funciones sin sentencias SQL . . . 27
1.8.4 Funciones con sentencias SQL . . . 28
1.8.5 Manejo de errores en MySQL . . . 28
1.8.6 Transacciones con procedimientos almacenados . . . 28
1.8.7 Cursores . . . 29
1.8.8 *Triggers* . . . 30
1.9 Ejercicios de repaso . . . 32
1.10 Recursos . . . 38

2 Licencia **39**

Capítulo 1

Triggers, procedimientos y funciones en MySQL

En esta unidad vamos a estudiar los procedimientos, funciones y *triggers* de MySQL, que son objetos que contienen código SQL y se almacenan asociados a una base de datos.

- **Procedimiento almacenado**: Es un objeto que se crea con la sentencia `CREATE PROCEDURE` y se invoca con la sentencia `CALL`. Un procedimiento puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida.
- **Función almacenada**: Es un objeto que se crea con la sentencia `CREATE FUNCTION` y se invoca con la sentencia `SELECT` o dentro de una expresión. Una función puede tener cero o muchos parámetros de entrada y siempre devuelve un valor, asociado al nombre de la función.
- **Trigger**: Es un objeto que se crea con la sentencia `CREATE TRIGGER` y tiene que estar asociado a una tabla. Un *trigger* se activa cuando ocurre un evento de inserción, actualización o borrado, sobre la tabla a la que está asociado.

1.1 Procedimientos

Un procedimiento almacenado es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Es un objeto que se crea con la sentencia `CREATE PROCEDURE` y se invoca con la sentencia `CALL`. Un procedimiento puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida.

1.1.1 Sintaxis

```
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.

1.1.2 DELIMITER

Para definir un procedimiento almacenado es necesario modificar temporalmente el carácter separador que se utiliza para delimitar las sentencias SQL.

El carácter separador que se utiliza por defecto en SQL es el punto y coma (`;`). En los ejemplos que vamos a realizar en esta unidad vamos a utilizar los caracteres `$$` para delimitar las instrucciones SQL, pero es posible utilizar cualquier otro carácter.

Ejemplo:

En este ejemplo estamos configurando los caracteres `$$` como los separadores entre las sentencias SQL.

```
DELIMITER $$
```

En este ejemplo volvemos a configurar que el carácter separador es el punto y coma.

```
DELIMITER ;
```

1.1.3 Parámetros de entrada, salida y entrada/salida

En los procedimientos almacenados podemos tener tres tipos de parámetros:

- **Entrada**: Se indican poniendo la palabra reservada `IN` delante del nombre del parámetro. Estos parámetros no pueden cambiar su valor dentro del procedimiento, es decir, cuando el procedimiento finalice estos parámetros tendrán el mismo valor que tenían cuando se hizo la llamada al procedimiento. En programación sería equivalente al paso por valor de un parámetro.
- **Salida**: Se indican poniendo la palabra reservada `OUT` delante del nombre del parametro. Estos parámetros cambian su valor dentro del procedimiento. Cuando se hace la llamada al procedimiento empiezan con un valor inicial y cuando finaliza la ejecución del procedimiento pueden terminar con otro valor diferente. En programación sería equivalente al paso por referencia de un parámetro.
- **Entrada/Salida**: Es una combinación de los tipos `IN` y `OUT`. Estos parámetros se indican poniendo la palabra reservada `IN/OUT` delante del nombre del parámetro.

Ejemplo 1:

En este ejemplo, se muestra la cabecera de un procedimiento llamado `listar_productos` que sólo tiene el parámetro `gama` que es de entrada (`IN`).

```
CREATE PROCEDURE listar_productos(IN gama VARCHAR(50))
```

Ejemplo 2:

Aquí se muestra la cabecera de un procedimiento llamado `contar_productos` que tiene el parámetro `gama` de entrada (`IN`) y el parámetro `total` de salida (`OUT`).

```
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
```

1.1.4 Ejemplo de un procedimiento con parámetros de entrada

Escriba un procedimiento llamado listar_productos que reciba como entrada el nombre de la gama y muestre un listado de todos los productos que existen dentro de esa gama. Este procedimiento no devuelve ningún parámetro de salida, lo que hace es mostrar el listado de los productos.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS listar_productos$$
CREATE PROCEDURE listar_productos(IN gama VARCHAR(50))
BEGIN
  SELECT *
  FROM producto
  WHERE producto.gama = gama;
END
$$
```

1.1.5 Llamada de procedimientos con CALL

Para hacer la llamada a un procedimiento almacenado se utiliza la palabra reservada `CALL`.

Ejemplo:

```
DELIMITER ;
CALL listar_productos('Herramientas');
SELECT * FROM producto;
```

1.1.6 Ejemplos de procedimientos con parámetros de salida

Ejemplo 1:

Escriba un procedimiento llamado `contar_productos` que reciba como entrada el nombre de la gama y devuelva el número de productos que existen dentro de esa gama. Resuelva el ejercicio de dos formas distintas, utilizando `SET` y `SELECT ... INTO`.

```
-- Solución 1. Utilizando SET
DELIMITER $$
DROP PROCEDURE IF EXISTS contar_productos$$
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
BEGIN
  SET total = (
    SELECT COUNT(*)
    FROM producto
    WHERE producto.gama = gama);
END
$$

DELIMITER ;
CALL contar_productos('Herramientas', @total);
SELECT @total;

-- Solución 2. Utilizando SELECT ... INTO
DELIMITER $$
DROP PROCEDURE IF EXISTS contar_productos$$
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
BEGIN
  SELECT COUNT(*)
  INTO total
  FROM producto
  WHERE producto.gama = gama;
END
$$
```

```
DELIMITER ;
CALL contar_productos('Herramientas', @total);
SELECT @total;
```

Ejemplo 2:

Escribe un procedimiento que se llame `calcular_max_min_media`, que reciba como parámetro de entrada el nombre de la gama de un producto y devuelva como salida tres parámetros. El precio máximo, el precio mínimo y la media de los productos que existen en esa gama. Resuelva el ejercicio de dos formas distintas, utilizando `SET` y `SELECT ... INTO`.

```
-- ńSolucio 1. Utilizando SET
DELIMITER $$
DROP PROCEDURE IF EXISTS calcular_max_min_media$$
CREATE PROCEDURE calcular_max_min_media(
  IN gama VARCHAR(50),
    OUT maximo DECIMAL(15, 2),
  OUT minimo DECIMAL(15, 2),
    OUT media DECIMAL(15, 2)
)
BEGIN
  SET maximo = (
    SELECT MAX(precio_venta)
    FROM producto
        WHERE producto.gama = gama);

  SET minimo = (
    SELECT MIN(precio_venta)
    FROM producto
        WHERE producto.gama = gama);

  SET media = (
    SELECT AVG(precio_venta)
    FROM producto
        WHERE producto.gama = gama);
END
$$

DELIMITER ;
CALL calcular_max_min_media('Herramientas', @maximo, @minimo, @media);
SELECT @maximo, @minimo, @media;

-- ńSolucio 2. Utilizando SELECT ... INTO
DELIMITER $$
DROP PROCEDURE IF EXISTS calcular_max_min_media$$
CREATE PROCEDURE calcular_max_min_media(
  IN gama VARCHAR(50),
```

```
    OUT maximo DECIMAL(15, 2),
  OUT minimo DECIMAL(15, 2),
    OUT media DECIMAL(15, 2)
)
BEGIN
  SELECT
    MAX(precio_venta),
    MIN(precio_venta),
    AVG(precio_venta)
    FROM producto
  WHERE producto.gama = gama
    INTO maximo, minimo, media;
END
$$

DELIMITER ;
CALL calcular_max_min_media('Herramientas', @maximo, @minimo, @media);
SELECT @maximo, @minimo, @media;
```

1.2 Funciones

Una función almacenada es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Es un objeto que se crea con la sentencia `CREATE FUNCTION` y se invoca con la sentencia `SELECT` o dentro de una expresión. Una función puede tener cero o muchos parámetros de entrada y siempre devuelve un valor, asociado al nombre de la función.

1.2.1 Sintaxis

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

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.

1.2.2 Parámetros de entrada

En una función todos los parámetros son de entrada, por lo tanto, **no será necesario** utilizar la palabra reservada `IN` delante del nombre de los parámetros.

Ejemplo:

A continuación se muestra la cabecera de la función `contar_productos` que tiene un parámetro de entrada llamado `gama`.

```
CREATE FUNCTION contar_productos(gama VARCHAR(50))
```

1.2.3 Resultado de salida

Una función siempre devolverá un valor de salida asociado al nombre de la función. En la definición de la cabecera de la función hay que definir el tipo de dato que devuelve con la palabra reservada `RETURNS` y en el cuerpo de la función debemos incluir la palabra reservada `RETURN` para devolver el valor de la función.

Ejemplo:

En este ejemplo se muestra una **defínción incompleta** de una función donde se se puede ver el uso de las palabras reservadas `RETURNS` y `RETURN`.

```
DELIMITER $$
DROP FUNCTION IF EXISTS contar_productos$$
CREATE FUNCTION contar_productos(gama VARCHAR(50))
  RETURNS INT UNSIGNED
  ...
BEGIN
  ...

  RETURN total;
END
$$
```

1.2.4 Características de la función

Después de la definición del tipo de dato que devolverá la función con la palabra reservada `RETURNS`, tenemos que indicar las características de la función. Las opciones disponibles son las siguientes:

- `DETERMINISTIC`: Indica que la función siempre devuelve el mismo resultado cuando se utilizan los mismos parámetros de entrada.
- `NOT DETERMINISTIC`: Indica que la función no siempre devuelve el mismo resultado, aunque se utilicen los mismos parámetros de entrada. Esta es la opción que se selecciona por defecto cuando no se indica una característica de forma explícita.
- `CONTAINS SQL`: Indica que la función contiene sentencias SQL, pero no contiene sentencias de manipulación de datos. Algunos ejemplos de sentencias SQL que pueden aparecer en este caso son operaciones con variables (Ej: `SET @x = 1`) o uso de funciones de MySQL (Ej: `SELECT NOW();`) entre otras. Pero en ningún caso aparecerán sentencias de escritura o lectura de datos.
- `NO SQL`: Indica que la función no contiene sentencias SQL.
- `READS SQL DATA`: Indica que la función no modifica los datos de la base de datos y que contiene sentencias de lectura de datos, como la sentencia `SELECT`.
- `MODIFIES SQL DATA`: Indica que la función sí modifica los datos de la base de datos y que contiene sentencias como `INSERT`, `UPDATE` o `DELETE`.

Para poder crear una función en MySQL es necesario indicar al menos una de estas tres características:

- `DETERMINISTIC`
- `NO SQL`
- `READS SQL DATA`

Si no se indica al menos una de estas características obtendremos el siguiente mensaje de error.

```
ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL,
or READS SQL DATA in its declaration and binary logging is enabled
(you *might* want to use the less safe log_bin_trust_function_creators
variable)
```

Es posible configurar el valor de la varible global `log_bin_trust_function_creators` a `1`, para indicar a MySQL que queremos eliminar la restricción de indicar alguna de las características anteriores cuando definimos una función almacenada. Esta variable está configurada con el valor `0` por defecto y para poder modificarla es necesario contar con el privilegio `SUPER`.

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

En lugar de configurar la variable global en tiempo de ejecución, es posible modificarla en el archivo de configuración de MySQL.

1.2.5 Declaración de variables locales con DECLARE

Tanto en los procedimientos como en las funciones es posible declarar variables locales con la palabra reservada `DECLARE`.

La sintaxis para declarar variables locales con `DECLARE` es la siguiente.

```
DECLARE var_name [, var_name] ... type [DEFAULT value]
```

El ámbito de una variable local será el bloque `BEGIN` y `END` del procedimiento o la función donde ha sido declarada.

Una restricción que hay que tener en cuenta a la hora de trabajar con variables locales, es que se deben declarar antes de los cursores y los *handlers*.

Ejemplo:

En este ejemplo estamos declarando una variable local con el nombre `total` que es de tipo `INT UNSIGNED`.

```
DECLARE total INT UNSIGNED;
```

Puede encontrar más información en la documentación oficial de MySQL.

1.2.6 Ejemplos

Escriba una función llamada `contar_productos` que reciba como entrada el nombre de la gama y devuelva el número de productos que existen dentro de esa gama.

```
DELIMITER $$
DROP FUNCTION IF EXISTS contar_productos$$
CREATE FUNCTION contar_productos(gama VARCHAR(50))
  RETURNS INT UNSIGNED
BEGIN
  -- Paso 1. Declaramos una variable local
  DECLARE total INT UNSIGNED;

  -- Paso 2. Contamos los productos
  SET total = (
    SELECT COUNT(*)
    FROM producto
    WHERE producto.gama = gama);

  -- Paso 3. Devolvemos el resultado
  RETURN total;
END
```

```
$$

DELIMITER ;
SELECT contar_productos('Herramientas');
```

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 utilzar `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
$$
```

1.6 Cursores

Los cursores nos permiten almacenar una 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

1.8.1 Procedimientos sin sentencias SQL

1. Escribe un procedimiento que no tenga ningún parámetro de entrada ni de salida y que muestre el texto `¡Hola mundo!`.
2. Escribe un procedimiento que reciba un número real de entrada y muestre un mensaje indicando si el número es positivo, negativo o cero.
3. Modifique el procedimiento diseñado en el ejercicio anterior para que tenga un parámetro de entrada, con el valor un número real, y un parámetro de salida, con una cadena de caracteres indicando si el número es positivo, negativo o cero.
4. Escribe un procedimiento que reciba un número real de entrada, que representa el valor de la nota de un alumno, y muestre un mensaje indicando qué nota ha obtenido teniendo en cuenta 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.

5. Modifique el procedimiento diseñado en el ejercicio anterior para que tenga un parámetro de entrada, con el valor de la nota en formato numérico y un parámetro de salida, con una cadena de texto indicando la nota correspondiente.

6. Resuelva el procedimiento diseñado en el ejercicio anterior haciendo uso de la estructura de control `CASE`.
7. 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` debería devolver la cadena `lunes`.

1.8.2 Procedimientos con sentencias SQL

1. Escribe un procedimiento que reciba el nombre de un país como parámetro de entrada y realice una consulta sobre la tabla `cliente` para obtener todos los clientes que existen en la tabla de ese país.
2. Escribe un procedimiento que reciba como parámetro de entrada una forma de pago, que será una cadena de caracteres (Ejemplo: `PayPal`, `Transferencia`, etc). Y devuelva como salida el pago de máximo valor realizado para esa forma de pago. Deberá hacer uso de la tabla `pago` de la base de datos `jardineria`.
3. Escribe un procedimiento que reciba como parámetro de entrada una forma de pago, que será una cadena de caracteres (Ejemplo: `PayPal`, `Transferencia`, etc). Y devuelva como salida los siguientes valores teniendo en cuenta la forma de pago seleccionada como parámetro de entrada:

- el pago de máximo valor,
- el pago de mínimo valor,
- el valor medio de los pagos realizados,
- la suma de todos los pagos,
- el número de pagos realizados para esa forma de pago.

Deberá hacer uso de la tabla `pago` de la base de datos `jardineria`.

4. Crea una **base de datos** llamada `procedimientos` que contenga una **tabla** llamada `cuadrados`. La tabla `cuadrados` debe tener dos columnas de tipo `INT UNSIGNED`, una columna llamada `número` y otra columna llamada `cuadrado`.

Una vez creada la base de datos y la tabla deberá **crear un procedimiento** llamado `calcular_cuadrados` con las siguientes características. El procedimiento recibe un parámetro de entrada llamado `tope` de tipo `INT UNSIGNED` y calculará el valor de los cuadrados de los primeros números naturales hasta el valor introducido como parámetro. El valor del números y de sus cuadrados deberán ser almacenados en la tabla `cuadrados` que hemos creado previamente.

Tenga en cuenta que el procedimiento deberá eliminar el contenido actual de la tabla antes de insertar los nuevos valores de los cuadrados que va a calcular.

Utilice un bucle `WHILE` para resolver el procedimiento.

5. Utilice un bucle `REPEAT` para resolver el procedimiento del ejercicio anterior.
6. Utilice un bucle `LOOP` para resolver el procedimiento del ejercicio anterior.
7. Crea una **base de datos** llamada `procedimientos` que contenga una **tabla** llamada `ejercicio`. La tabla debe tener una única columna llamada `número` y el tipo de dato de esta columna debe ser `INT UNSIGNED`.

Una vez creada la base de datos y la tabla deberá **crear un procedimiento** llamado `calcular_números` con las siguientes características. El procedimiento recibe un parámetro de entrada llamado `valor_inicial` de tipo `INT UNSIGNED` y deberá almacenar en la tabla `ejercicio` toda la secuencia de números desde el valor inicial pasado como entrada hasta el 1.

Tenga en cuenta que el procedimiento deberá eliminar el contenido actual de las tablas antes de insertar los nuevos valores.

Utilice un bucle `WHILE` para resolver el procedimiento.

8. Utilice un bucle `REPEAT` para resolver el procedimiento del ejercicio anterior.
9. Utilice un bucle `LOOP` para resolver el procedimiento del ejercicio anterior.
10. Crea una **base de datos** llamada `procedimientos` que contenga una **tabla** llamada `pares` y otra **tabla** llamada `impares`. Las dos tablas deben tener única columna llamada `número` y el tipo de dato de esta columna debe ser `INT UNSIGNED`.

Una vez creada la base de datos y las tablas deberá **crear un procedimiento** llamado `calcular_pares_impares` con las siguientes características. El procedimiento recibe un parámetro de entrada llamado `tope` de tipo `INT UNSIGNED` y deberá almacenar en la tabla `pares` aquellos números pares que existan entre el número 1 el valor introducido como parámetro. Habrá que realizar la misma operación para almacenar los números impares en la tabla `impares`.

Tenga en cuenta que el procedimiento deberá eliminar el contenido actual de las tablas antes de insertar los nuevos valores.

Utilice un bucle `WHILE` para resolver el procedimiento.

11. Utilice un bucle `REPEAT` para resolver el procedimiento del ejercicio anterior.
12. Utilice un bucle `LOOP` para resolver el procedimiento del ejercicio anterior.

1.8.3 Funciones sin sentencias SQL

1. Escribe una función que reciba un número entero de entrada y devuelva `TRUE` si el número es par o `FALSE` en caso contrario.
2. Escribe una función que devuelva el valor de la hipotenusa de un triángulo a partir de los valores de sus lados.
3. Escribe una función 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` debería devolver la cadena `lunes`.
4. Escribe una función que reciba tres números reales como parámetros de entrada y devuelva el mayor de los tres.
5. Escribe una función que devuelva el valor del área de un círculo a partir del valor del radio que se recibirá como parámetro de entrada.
6. Escribe una función que devuelva como salida el número de años que han transcurrido entre dos fechas que se reciben como parámetros de entrada. Por ejemplo, si pasamos como parámetros de entrada las fechas `2018-01-01` y `2008-01-01` la función tiene que devolver que han pasado 10 años.

Para realizar esta función puede hacer uso de las siguientes funciones que nos proporciona MySQL:

- `DATEDIFF`
- `TRUNCATE`

7. Escribe una función que reciba una cadena de entrada y devuelva la misma cadena pero sin acentos. La función tendrá que reemplazar todas las vocales que tengan acento por la misma vocal pero sin acento. Por ejemplo, si la función recibe como parámetro de entrada la cadena `María` la función debe devolver la cadena `Maria`.

1.8.4 Funciones con sentencias SQL

1. Escribe una función para la base de datos `tienda` que devuelva el número total de productos que hay en la tabla `productos`.
2. Escribe una función para la base de datos `tienda` que devuelva el valor medio del precio de los productos de un determinado fabricante que se recibirá como parámetro de entrada. El parámetro de entrada será el nombre del fabricante.
3. Escribe una función para la base de datos `tienda` que devuelva el valor máximo del precio de los productos de un determinado fabricante que se recibirá como parámetro de entrada. El parámetro de entrada será el nombre del fabricante.
4. Escribe una función para la base de datos `tienda` que devuelva el valor mínimo del precio de los productos de un determinado fabricante que se recibirá como parámetro de entrada. El parámetro de entrada será el nombre del fabricante.

1.8.5 Manejo de errores en MySQL

1. Crea una **base de datos** llamada `test` que contenga una **tabla** llamada `alumno`. La tabla debe tener cuatro columnas:

- `id`: entero sin signo (clave primaria).
- `nombre`: cadena de 50 caracteres.
- `apellido1`: cadena de 50 caracteres.
- `apellido2`: cadena de 50 caracteres.

Una vez creada la base de datos y la tabla deberá **crear un procedimiento** llamado `insertar_alumno` con las siguientes características. El procedimiento recibe cuatro parámetros de **entrada** (`id`, `nombre`, `apellido1`, `apellido2`) y los insertará en la tabla `alumno`. El procedimiento devolverá como **salida** un parámetro llamado `error` que tendrá un valor igual a 0 si la operación se ha podido realizar con éxito y un valor igual a 1 en caso contrario.

Deberá manejar los errores que puedan ocurrir cuando se intenta insertar una fila que contiene una clave primaria repetida.

1.8.6 Transacciones con procedimientos almacenados

1. Crea una **base de datos** llamada `cine` que contenga **dos tablas** con las siguientes columnas.

Tabla `cuentas`:

- `id_cuenta`: entero sin signo (clave primaria).
- `saldo`: real sin signo.

Tabla `entradas`:

- `id_butaca`: entero sin signo (clave primaria).
- `nif`: cadena de 9 caracteres.

Una vez creada la base de datos y las tablas deberá **crear un procedimiento** llamado `comprar_entrada` con las siguientes características. El procedimiento recibe 3 parámetros de **entrada** (`nif`, `id_cuenta`, `id_butaca`) y devolverá como **salida** un parámetro llamado `error` que tendrá un valor igual a 0 si la compra de la entrada se ha podido realizar con éxito y un valor igual a 1 en caso contrario.

El procedimiento de compra realiza los siguientes pasos:

- Inicia una transacción.
- Actualiza la columna `saldo` de la tabla `cuentas` cobrando 5 euros a la cuenta con el `id_cuenta` adecuado.
- Inserta una una fila en la tabla `entradas` indicando la butaca (`id_butaca`) que acaba de comprar el usuario (`nif`).
- Comprueba si ha ocurrido algún error en las operaciones anteriores. Si no ocurre ningún error entonces aplica un `COMMIT` a la transacción y si ha ocurrido algún error aplica un `ROLLBACK`.

Deberá manejar los siguientes errores que puedan ocurrir durante el proceso.

- ERROR 1264 (Out of range value)
- ERROR 1062 (Duplicate entry for PRIMARY KEY)

2. ¿Qué ocurre cuando intentamos comprar una entrada y le pasamos como parámetro un número de cuenta que no existe en la tabla `cuentas`? ¿Ocurre algún error o podemos comprar la entrada?

En caso de que exista algún error, ¿cómo podríamos resolverlo?.

1.8.7 Cursores

1. Escribe las sentencias SQL necesarias para **crear una base de datos** llamada `test`, **una tabla** llamada `alumnos` y **4 sentencias de inserción** para inicializar la tabla. La tabla alumnos está formada por las siguientes columnas:

- `id` (entero sin signo y clave primaria)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres
- `fecha_nacimiento` (fecha)

Una vez creada la tabla se decide añadir una nueva columna a la tabla llamada `edad` que será un valor calculado a partir de la columna `fecha_nacimiento`. Escriba la sentencia SQL necesaria para **modificar la tabla y añadir la nueva columna**.

Escriba una **función** llamada `calcular_edad` que reciba una fecha y devuelva el número de años que han pasado desde la fecha actual hasta la fecha pasada como parámetro:

- Función: `calcular_edad`
- Entrada: Fecha
- Salida: Número de años (entero)

Ahora escriba un procedimiento que permita calcular la edad de todos los alumnmos que ya existen en la tabla. Para esto será necesario crear un **procedimiento** llamado `actualizar_columna_edad` que calcule la edad de cada alumno y actualice la tabla. Este procedimiento hará uso de la función `calcular_edad` que hemos creado en el paso anterior.

2. Modifica la tabla `alumnos` del ejercicio anterior para añadir una nueva columna `email`. Una vez que hemos modificado la tabla necesitamos asignarle una dirección de correo electrónico de forma automática.

Escriba un **procedimiento** llamado **crear_email** que dados los parámetros de entrada: `nombre`, `apellido1`, `apellido2` y `dominio`, cree una dirección de email y la devuelva como salida.

- Procedimiento: `crear_email`
- Entrada:
  - `nombre` (cadena de caracteres)
  - `apellido1` (cadena de caracteres)
  - `apellido2` (cadena de caracteres)
  - `dominio` (cadena de caracteres)
- Salida:
  - `email` (cadena de caracteres)

devuelva una dirección de correo electrónico con el siguiente formato:

- El primer carácter del parámetro `nombre`.
- Los tres primeros caracteres del parámetro `apellido1`.
- Los tres primeros caracteres del parámetro `apellido2`.
- El carácter `@`.
- El dominio pasado como parámetro.

Ahora escriba un procedimiento que permita crear un email para todos los alumnmos que ya existen en la tabla. Para esto será necesario crear un **procedimiento** llamado `actualizar_columna_email` que actualice la columna `email` de la tabla `alumnos`. Este procedimiento hará uso del procedimiento `crear_email` que hemos creado en el paso anterior.

3. Escribe un **procedimiento** llamado `crear_lista_emails_alumnos` que devuelva la lista de emails de la tabla `alumnos` separados por un punto y coma. Ejemplo: `juan@iescelia.org;maria@iescelia.org;pepe@iescelia.org;lucia@iescelia.org`.

1.8.8 *Triggers*

1. 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.

2. 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)
- `email` (cadena de caracteres)

Escriba un **procedimiento** llamado **crear_email** que dados los parámetros de entrada: `nombre`, `apellido1`, `apellido2` y `dominio`, cree una dirección de email y la devuelva como salida.

- Procedimiento: `crear_email`
- Entrada:
  - `nombre` (cadena de caracteres)
  - `apellido1` (cadena de caracteres)
  - `apellido2` (cadena de caracteres)
  - `dominio` (cadena de caracteres)
- Salida:
  - `email` (cadena de caracteres)

devuelva una dirección de correo electrónico con el siguiente formato:

- El primer carácter del parámetro `nombre`.
- Los tres primeros caracteres del parámetro `apellido1`.
- Los tres primeros caracteres del parámetro `apellido2`.
- El carácter `@`.
- El dominio pasado como parámetro.

Una vez creada la tabla escriba **un trigger** con las siguientes características:

- Trigger: `trigger_crear_email_before_insert`
  - Se ejecuta sobre la tabla `alumnos`.
  - Se ejecuta *antes* de una operación de *inserción*.
  - Si el nuevo valor del email que se quiere insertar es `NULL`, entonces se le creará automáticamente una dirección de email y se insertará en la tabla.
  - Si el nuevo valor del email no es `NULL` se guardará en la tabla el valor del email.

Nota: Para crear la nueva dirección de email se deberá hacer uso del procedimiento `crear_email`.

3. Modifica el ejercicio anterior y añade un nuevo *trigger* que las siguientes características:

Trigger: `trigger_guardar_email_after_update`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta *después* de una operación de *actualización*.
- Cada vez que un alumno modifique su dirección de email se deberá insertar un nuevo registro en una tabla llamada `log_cambios_email`.

La tabla `log_cambios_email` contiene los siguientes campos:

- `id`: clave primaria (entero autonumérico)
- `id_alumno`: id del alumno (entero)
- `fecha_hora`: marca de tiempo con el instante del cambio (fecha y hora)
- `old_email`: valor anterior del email (cadena de caracteres)
- `new_email`: nuevo valor con el que se ha actualizado

4. Modifica el ejercicio anterior y añade un nuevo *trigger* que tenga las siguientes características:

Trigger: `trigger_guardar_alumnos_eliminados`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta *después* de una operación de *borrado*.
- Cada vez que se elimine un alumno de la tabla `alumnos` se deberá insertar un nuevo registro en una tabla llamada `log_alumnos_eliminados`.

La tabla `log_alumnos_eliminados` contiene los siguientes campos:

- `id`: clave primaria (entero autonumérico)
- `id_alumno`: id del alumno (entero)
- `fecha_hora`: marca de tiempo con el instante del cambio (fecha y hora)
- `nombre`: nombre del alumno eliminado (cadena de caracteres)
- `apellido1`: primer apellido del alumno eliminado (cadena de caracteres)
- `apellido2`: segundo apellido del alumno eliminado (cadena de caracteres)
- `email`: email del alumno eliminado (cadena de caracteres)

1.9 Ejercicios de repaso

1. ¿Qué beneficios nos puede aportar utilizar procedimientos y funciones almacenadas?
2. Según la siguiente sentencia, ¿estamos haciendo una llamada a un procedimiento o a una función?

```
CALL resolver_ejercicio2()
```

3. ¿Cuáles de los siguientes bloques son correctos?

```
1.
LOOP bucle:
  statements
END bucle:

2.
bucle: LOOP
  statements
END bucle;

3.
bucle:
LOOP bucle;
  statements;
END bucle;
```

4. ¿Pueden aparecer las siguientes sentencias en el mismo bloque de código?

```
DECLARE a INT;
DECLARE a INT;
```

5. ¿Pueden aparecer las siguientes sentencias en el mismo bloque de código?

```
DECLARE a INT;
DECLARE a FLOAT;
```

6. ¿Pueden aparecer las siguientes sentencias en el mismo bloque de código?

```
DECLARE b VARCHAR(20);
DECLARE b HANDLER FOR SQLSTATE '02000';
```

7. ¿Para qué podemos utilizar un cursor en MySQL?
8. ¿Puedo actualizar los datos de un cursor en MySQL? Si fuese posible actualizar los datos de un cursor, ¿se actualizarían automáticamente los datos de la tabla?
9. Cuál o cuáles de los siguientes bucles no está soportado en MySQL: `FOR`, `LOOP`, `REPEAT` y `WHILE`.
10. Si el cuerpo del bucle se debe ejecutar al menos una vez, ¿qué bucle sería más apropiado?

11. ¿Qué valor devuelve la sentencia `SELECT value`?

- 0
- 9
- 10
- `NULL`
- El código entra en un bucle infinito y nunca alcanza la sentencia `SELECT value`

```
DELIMITER $$
CREATE PROCEDURE incrementor (OUT i INT)
BEGIN
  REPEAT
    SET i = i + 1;
  UNTIL i > 9
  END REPEAT;
END;

DELIMITER $$
CREATE PROCEDURE test ()
BEGIN
  DECLARE value INT default 0;
  CALL incrementor(value);

  -- ¿Qué valor se muestra en esta sentencia?
  SELECT value;
END;

DELIMITER ;
CALL test();
```

11. ¿Qué valor devuelve la sentencia `SELECT value`?

- 0
- 9
- 10
- `NULL`
- El código entra en un bucle infinito y nunca alcanza la sentencia `SELECT value`

```
DELIMITER $$
CREATE PROCEDURE incrementor (IN i INT)
BEGIN
  REPEAT
    SET i = i + 1;
  UNTIL i > 9
  END REPEAT;
END;
```

```
DELIMITER $$
CREATE PROCEDURE test ()
BEGIN
  DECLARE value INT default 0;
  CALL incrementor(value);

  -- ¿Qué valor se muestra en esta sentencia?
  SELECT value;
END;

DELIMITER ;
CALL test();
```

12. Realice los siguientes procedimientos y funciones sobre la base de datos `jardineria`.

a)

- Función: `calcular_precio_total_pedido`
- Descripción: Dado un código de pedido la función debe calcular la suma total del pedido. Tenga en cuenta que un pedido puede contener varios productos diferentes y varias cantidades de cada producto.
- Parámetros de entrada: `codigo_pedido` (`INT`)
- Parámetros de salida: El precio total del pedido (`FLOAT`)

b)

- Función: `calcular_suma_pedidos_cliente`
- Descripción: Dado un código de cliente la función debe calcular la suma total de todos los pedidos realizados por el cliente. Deberá hacer uso de la función `calcular_precio_total_pedido` que ha desarrollado en el apartado anterior.
- Parámetros de entrada: `codigo_cliente` (`INT`)
- Parámetros de salida: La suma total de todos los pedidos del cliente (`FLOAT`)

c)

- Función: `calcular_suma_pagos_cliente`
- Descripción: Dado un código de cliente la función debe calcular la suma total de los pagos realizados por ese cliente.
- Parámetros de entrada: `codigo_cliente` (`INT`)
- Parámetros de salida: La suma total de todos los pagos del cliente (`FLOAT`)

d)

- Procedimiento: `calcular_pagos_pendientes`
- Descripción: Deberá calcular los pagos pendientes de todos los clientes. Para saber si un cliente tiene algún pago pendiente deberemos calcular cuál es la cantidad de todos los pedidos y los pagos que ha realizado. Si la cantidad de los pedidos es mayor que la de los pagos entonces ese cliente tiene pagos pendientes.

Deberá insertar en una tabla llamada `clientes_con_pagos_pendientes` los siguientes datos:

- `id_cliente`
- `suma_total_pedidos`

- `suma_total_pagos`
- `pendiente_de_pago`

13. Teniendo en cuenta el significado de los siguientes códigos de error:

- Error: 1036 (ER_OPEN_AS_READONLY). Table «%s» is read only
- Error: 1062 (ER_DUP_ENTRY). Duplicate entry «%s» for key %d

```
-- Paso 1
CREATE TABLE t (s1 INT, PRIMARY KEY (s1));

-- Paso 2
DELIMITER $$
CREATE PROCEDURE handlerexam(IN a INT, IN b INT, IN c INT, OUT x INT)
BEGIN
  DECLARE EXIT HANDLER FOR 1036 SET x = 10;
  DECLARE EXIT HANDLER FOR 1062 SET x = 30;

  SET x = 1;
  INSERT INTO t VALUES (a);
  SET x = 2;
  INSERT INTO t VALUES (b);
  SET x = 3;
  INSERT INTO t VALUES (c);
  SET x = 4;
END
$$
```

¿Qué devolvería la última sentencia `SELECT @x` en cada caso (**a** y **b**)? Justifique su respuesta. Sin una justificación válida la respuesta será considerada incorrecta.

```
-- a)
CALL handlerexam(1, 2, 3, @x);
SELECT @x;

-- b)
CALL handlerexam(1, 2, 1, @x);
SELECT @x;
```

14. Dado el siguiente procedimiento:

```
-- Paso 1
CREATE TABLE t (s1 INT, PRIMARY KEY (s1));

-- Paso 2
DELIMITER $$
CREATE PROCEDURE test(IN a INT, OUT b INT)
BEGIN
    SET b = 0;
    WHILE a > b DO
        SET b = b + 1;
        IF b != 2 THEN
            INSERT INTO t VALUES (b);
        END IF
    END WHILE;
END;
```

¿Qué valores tendría la tabla `t` y qué valor devuelve la sentencia `SELECT value` en cada caso (**a** y **b**)? Justifique la respuesta. Sin una justificación válida la respuesta será considerada incorrecta.

```
-- a)
CALL test(-10, @value);
SELECT @value;

-- b)
CALL test(10, @value);
SELECT @value;
```

15. Escriba un **procedimiento** llamado `obtener_numero_empleados` que reciba como parámetro de entrada el código de una oficina y devuelva el número de empleados que tiene.

Escriba una sentencia SQL que realice una llamada al procedimiento realizado para comprobar que se ejecuta correctamente.

16. Escriba una **función** llamada `cantidad_total_de_productos_vendidos` que reciba como parámetro de entrada el código de un producto y devuelva la cantidad total de productos que se han vendido con

ese código.

Escriba una sentencia SQL que realice una llamada a la función realizada para comprobar que se ejecuta correctamente.

17. Crea una tabla que se llame `productos_vendidos` que tenga las siguientes columnas:

- `id` (entero sin signo, auto incremento y clave primaria)
- `codigo_producto` (cadena de caracteres)
- `cantidad_total` (entero)

Escriba un **procedimiento** llamado `estadísticas_productos_vendidos` que para cada uno de los productos de la tabla `producto` calcule la cantidad total de unidades que se han vendido y almacene esta información en la tabla `productos_vendidos`.

El procedimiento tendrá que realizar las siguientes acciones:

- Borrar el contenido de la tabla `productos_vendidos`.
- Recorrer cada uno de los productos de la tabla `producto`. Será necesario usar un **cursor**.
- Calcular la cantidad total de productos vendidos. En este paso será necesario utilizar la función `cantidad_total_de_productos_vendidos` desarrollada en el ejercicio 2.
- Insertar en la tabla `productos_vendidos` los valores del código de producto y la cantidad total de unidades que se han vendido para ese producto en concreto.

18. Crea una tabla que se llame `notificaciones` que tenga las siguientes columnas:

- `id` (entero sin signo, autoincremento y clave primaria)
- `fecha_hora`: marca de tiempo con el instante del pago (fecha y hora)
- `total`: el valor del pago (real)
- `codigo_cliente`: código del cliente que realiza el pago (entero)

Escriba un *trigger* que nos permita llevar un control de los pagos que van realizando los clientes. Los detalles de implementación son los siguientes:

- Nombre: `trigger_notificar_pago`
- Se ejecuta sobre la tabla `pago`.
- Se ejecuta *después* de hacer la inserción de un pago.
- Cada vez que un cliente realice un pago (es decir, se hace una inserción en la tabla `pago`), el *trigger* deberá insertar un nuevo registro en una tabla llamada `notificaciones`.

Escriba algunas sentencias SQL para comprobar que el *trigger* funciona correctamente.

1.10 Recursos

- MySQL Stored Procedures. Peter Gulutzan.
- MySQL Stored Procedures. MySQL Tutorial.

Capítulo 2

Licencia