
Unidad 12. Triggers, procedimientos y funciones en MySQL

Apuntes de BD para DAW, DAM y ASIR

José Juan Sánchez Hernández

Curso 2023/2024

Índice

| | | |
|----------|--|----------|
| 1 | Triggers, procedimientos y funciones en MySQL | 1 |
| 1.1 | Procedimientos | 1 |
| 1.1.1 | Sintaxis | 1 |
| 1.1.2 | DELIMITER | 2 |
| 1.1.3 | Parámetros de entrada, salida y entrada/salida | 2 |
| 1.1.4 | Ejemplo de un procedimiento con parámetros de entrada | 3 |
| 1.1.5 | Llamada de procedimientos con CALL | 3 |
| 1.1.6 | Tipos de variables en MySQL | 3 |
| 1.1.7 | Declaración de variables locales con DECLARE | 4 |
| 1.1.8 | Ejemplos de procedimientos con parámetros de salida | 4 |
| 1.2 | Funciones | 7 |
| 1.2.1 | Sintaxis | 7 |
| 1.2.2 | Parámetros de entrada | 8 |
| 1.2.3 | Resultado de salida | 8 |
| 1.2.4 | Características de la función | 9 |
| 1.2.5 | Ejemplos | 10 |
| 1.3 | Estructuras de control | 10 |
| 1.3.1 | Instrucciones condicionales | 10 |
| 1.3.1.1 | IF-THEN-ELSE | 10 |
| 1.3.1.2 | CASE | 10 |
| 1.3.2 | Instrucciones repetitivas o bucles | 11 |
| 1.3.2.1 | LOOP | 11 |
| 1.3.2.2 | REPEAT | 12 |
| 1.3.2.3 | WHILE | 12 |
| 1.4 | Manejo de errores | 13 |
| 1.4.1 | DECLARE . . . HANDLER | 13 |
| 1.4.2 | Ejemplo 1 - DECLARE CONTINUE HANDLER | 14 |
| 1.4.3 | Ejemplo 2 - DECLARE EXIT HANDLER | 15 |
| 1.5 | Cómo realizar transacciones con procedimientos almacenados | 16 |
| 1.6 | Cursorres | 16 |
| 1.6.1 | Operaciones con cursores | 17 |
| 1.6.1.1 | DECLARE | 17 |
| 1.6.1.2 | OPEN | 17 |
| 1.6.1.3 | FETCH | 17 |
| 1.6.1.4 | CLOSE | 17 |
| 1.7 | Triggers | 19 |

| | | |
|----------|--|-----------|
| 1.8 | Ejercicios | 22 |
| 1.8.1 | Procedimientos sin sentencias SQL | 22 |
| 1.8.2 | Procedimientos con sentencias SQL | 22 |
| 1.8.3 | Funciones sin sentencias SQL | 24 |
| 1.8.4 | Funciones con sentencias SQL | 25 |
| 1.8.5 | Manejo de errores en MySQL | 25 |
| 1.8.6 | Transacciones con procedimientos almacenados | 25 |
| 1.8.7 | Cursorres | 26 |
| 1.8.8 | <i>Triggers</i> | 27 |
| 1.9 | Ejercicios de repaso | 30 |
| 1.10 | Recursos | 35 |
| 2 | Licencia | 36 |

Índice de figuras

Índice de cuadros

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

```
1 CREATE
2     [DEFINER = { user | CURRENT_USER }]
3     PROCEDURE sp_name ([proc_parameter[,...]])
4     [characteristic ...] routine_body
5
6 proc_parameter:
7     [ IN | OUT | INOUT ] param_name type
8
9 func_parameter:
10    param_name type
11
12 type:
13    Any valid MySQL data type
14
15 characteristic:
16    COMMENT 'string'
```

```
17 | LANGUAGE SQL
18 | [NOT] DETERMINISTIC
19 | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
20 | SQL SECURITY { DEFINER | INVOKER }
21
22 routine_body:
23     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.

```
1 DELIMITER $$
```

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

```
1 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 parámetro. 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**).

```
1 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).

```
1 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.

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS listar_productos$$
3 CREATE PROCEDURE listar_productos(IN gama VARCHAR(50))
4 BEGIN
5     SELECT *
6     FROM producto
7     WHERE producto.gama = gama;
8 END
9 $$
```

1.1.5 Llamada de procedimientos con CALL

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

Ejemplo:

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

1.1.6 Tipos de variables en MySQL

En MySQL podemos utilizar los siguientes tipos de variables:

- **Variables locales.** Se declaran con la palabra reservada `DECLARE` dentro de un procedimiento, una función o un trigger. Su ámbito es local al procedimiento, la función o el trigger donde han sido declaradas.

Ejemplo:

```
1 DECLARE total INT UNSIGNED;
```

- **Variables definida por el usuario en el ámbito de la sesión.** Se declaran precedidas del carácter `@` y tienen validez dentro la sesión donde han sido declaradas. Cuando finaliza la sesión su valor se pierde.

Ejemplo:

```
1 SET @total = 0;
```

- **Variables del sistema.** Estas variables se utilizan para configurar MySQL. Pueden ser globales o de sesión. La diferencia que existe entre ellas es que una variable de sesión pierde su contenido cuando cerramos la sesión con el servidor, mientras que una variable global mantiene su valor hasta que se realiza un reinicio del servicio o se modifica por otro valor.

Ejemplo:

```
1 SET @@GLOBAL.lc_time_names = 'es_ES';
2 SET GLOBAL lc_time_names = 'es_ES';
3
4 SET @@SESSION.lc_time_names = 'es_ES';
5 SET SESSION lc_time_names = 'es_ES';
```

1.1.7 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.

```
1 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`.

```
1 DECLARE total INT UNSIGNED;
```

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

1.1.8 Ejemplos de procedimientos con parámetros de salida

Ejemplo 1:

En este ejemplo vamos a crear dos procedimientos sin sentencias SQL:

- El procedimiento `calcular_area_circulo` recibe como entrada el radio de un círculo y devuelve como salida el área del círculo.

- El procedimiento `calcular_volumen_cilindro` recibe como entrada el radio y la altura, y devuelve como salida el volumen del cilindro. Este procedimiento hará uso del procedimiento `calcular_area_circulo`.

```

1 DROP DATABASE IF EXISTS test;
2 CREATE DATABASE test;
3 USE test;
4
5
6 DELIMITER $$
7 DROP PROCEDURE IF EXISTS calcular_area_circulo$$
8 CREATE PROCEDURE calcular_area_circulo(
9     IN radio DOUBLE,
10    OUT area DOUBLE
11 )
12 BEGIN
13     SET area = PI() * POW(radio, 2);
14 END
15 $$
16
17 DROP PROCEDURE IF EXISTS calcular_volumen_cilindro$$
18 CREATE PROCEDURE calcular_volumen_cilindro(
19     IN radio DOUBLE,
20     IN altura DOUBLE,
21     OUT volumen DOUBLE
22 )
23 BEGIN
24     DECLARE area DOUBLE;
25
26     -- La variable local `area` almacenará el valor de salida del procedimiento
27     CALL calcular_area_circulo(radio, area);
28
29     SET volumen = area * altura;
30 END
31 $$
32
33 DELIMITER ;
34 -- La variable de usuario `@volumen` almacenará el valor de salida del
35 -- procedimiento
36 CALL calcular_volumen_cilindro(4.5, 6, @volumen);
37 SELECT @volumen;

```

Ejemplo 2:

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`.

```

1 -- Solución 1. Utilizando SET
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS contar_productos$$
4 CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
5 BEGIN
6     SET total = (
7         SELECT COUNT(*)
8         FROM producto

```

```
9     WHERE producto.gama = gama);
10 END
11 $$
12
13 DELIMITER ;
14 CALL contar_productos('Herramientas', @total);
15 SELECT @total;
16
17 -- Solución 2. Utilizando SELECT ... INTO
18 DELIMITER $$
19 DROP PROCEDURE IF EXISTS contar_productos$$
20 CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
21 BEGIN
22     SELECT COUNT(*)
23     INTO total
24     FROM producto
25     WHERE producto.gama = gama;
26 END
27 $$
28
29 DELIMITER ;
30 CALL contar_productos('Herramientas', @total);
31 SELECT @total;
```

Nota importante:

En el ejemplo anterior, hemos utilizado la **variable de usuario** `@total` para almacenar el parámetro de salida del procedimiento `contar_productos`.

Tenga en cuenta que el uso del carácter `@` en la variable `@total` es debido a que estamos utilizando una **variable definida por el usuario** en el ámbito de la sesión del usuario y no porque estemos almacenando un parámetro de salida del procedimiento.

Ejemplo 3:

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`.

```
1 -- Solucio 1. Utilizando SET
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS calcular_max_min_media$$
4 CREATE PROCEDURE calcular_max_min_media(
5     IN gama VARCHAR(50),
6     OUT maximo DECIMAL(15, 2),
7     OUT minimo DECIMAL(15, 2),
8     OUT media DECIMAL(15, 2)
9 )
10 BEGIN
11     SET maximo = (
12         SELECT MAX(precio_venta)
13         FROM producto
14         WHERE producto.gama = gama);
15
16     SET minimo = (
```

```
17     SELECT MIN(precio_venta)
18     FROM producto
19     WHERE producto.gama = gama);
20
21     SET media = (
22     SELECT AVG(precio_venta)
23     FROM producto
24     WHERE producto.gama = gama);
25 END
26 $$
27
28 DELIMITER ;
29 CALL calcular_max_min_media('Herramientas', @maximo, @minimo, @media);
30 SELECT @maximo, @minimo, @media;
31
32 -- ñSolucio 2. Utilizando SELECT ... INTO
33 DELIMITER $$
34 DROP PROCEDURE IF EXISTS calcular_max_min_media$$
35 CREATE PROCEDURE calcular_max_min_media(
36     IN gama VARCHAR(50),
37     OUT maximo DECIMAL(15, 2),
38     OUT minimo DECIMAL(15, 2),
39     OUT media DECIMAL(15, 2)
40 )
41 BEGIN
42     SELECT
43         MAX(precio_venta),
44         MIN(precio_venta),
45         AVG(precio_venta)
46     FROM producto
47     WHERE producto.gama = gama
48     INTO maximo, minimo, media;
49 END
50 $$
51
52 DELIMITER ;
53 CALL calcular_max_min_media('Herramientas', @maximo, @minimo, @media);
54 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

```
1 CREATE
2     [DEFINER = { user | CURRENT_USER }]
3     FUNCTION sp_name ([func_parameter[,...]])
```

```
4 RETURNS type
5 [characteristic ...] routine_body
6
7 func_parameter:
8   param_name type
9
10 type:
11   Any valid MySQL data type
12
13 characteristic:
14   COMMENT 'string'
15   | LANGUAGE SQL
16   | [NOT] DETERMINISTIC
17   | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
18   | SQL SECURITY { DEFINER | INVOKER }
19
20 routine_body:
21   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`.

```
1 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 **definición incompleta** de una función donde se se puede ver el uso de las palabras reservadas `RETURNS` y `RETURN`.

```
1 DELIMITER $$
2 DROP FUNCTION IF EXISTS contar_productos$$
3 CREATE FUNCTION contar_productos(gama VARCHAR(50))
4   RETURNS INT UNSIGNED
5   ...
6 BEGIN
7   ...
8
```

```
9 RETURN total;
10 END
11 $$
```

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.

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

Es posible configurar el valor de la variable 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`.

```
1 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 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.

```
1 DELIMITER $$
2 DROP FUNCTION IF EXISTS contar_productos$$
3 CREATE FUNCTION contar_productos(gama VARCHAR(50))
4 RETURNS INT UNSIGNED
5 READS SQL DATA
6 BEGIN
7     -- Paso 1. Declaramos una variable local
8     DECLARE total INT UNSIGNED;
9
10    -- Paso 2. Contamos los productos
11    SET total = (
12        SELECT COUNT(*)
13        FROM producto
14        WHERE producto.gama = gama);
15
16    -- Paso 3. Devolvemos el resultado
17    RETURN total;
18 END
19 $$
20
21 DELIMITER ;
22 SELECT contar_productos('Herramientas');
```

1.3 Estructuras de control

1.3.1 Instrucciones condicionales

1.3.1.1 IF-THEN-ELSE

```
1 IF search_condition THEN statement_list
2     [ELSEIF search_condition THEN statement_list] ...
3     [ELSE statement_list]
4 END IF
```

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

1.3.1.2 CASE

Existen dos formas de utilizar CASE:

```
1 CASE case_value
2     WHEN when_value THEN statement_list
3     [WHEN when_value THEN statement_list] ...
4     [ELSE statement_list]
5 END CASE
```

o

```
1 CASE
2   WHEN search_condition THEN statement_list
3   [WHEN search_condition THEN statement_list] ...
4   [ELSE statement_list]
5 END CASE
```

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

1.3.2 Instrucciones repetitivas o bucles

1.3.2.1 LOOP

```
1 [begin_label:] LOOP
2   statement_list
3 END LOOP [end_label]
```

Ejemplo:

```
1 CREATE PROCEDURE doiterate(p1 INT)
2 BEGIN
3   label1: LOOP
4     SET p1 = p1 + 1;
5     IF p1 < 10 THEN
6       ITERATE label1;
7     END IF;
8     LEAVE label1;
9   END LOOP label1;
10  SET @x = p1;
11 END;
```

Ejemplo:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS ejemplo_bucle_loop$$
3 CREATE PROCEDURE ejemplo_bucle_loop(IN tope INT, OUT suma INT)
4 BEGIN
5   DECLARE contador INT;
6
7   SET contador = 1;
8   SET suma = 0;
9
10  bucle: LOOP
11    IF contador > tope THEN
12      LEAVE bucle;
13    END IF;
14
15    SET suma = suma + contador;
16    SET contador = contador + 1;
17  END LOOP;
18 END
19 $$
20
```



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

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

1.3.2.2 REPEAT

```
1 [begin_label:] REPEAT
2     statement_list
3 UNTIL search_condition
4 END REPEAT [end_label]
```

Ejemplo:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS ejemplo_bucle_repeat$$
3 CREATE PROCEDURE ejemplo_bucle_repeat(IN tope INT, OUT suma INT)
4 BEGIN
5     DECLARE contador INT;
6
7     SET contador = 1;
8     SET suma = 0;
9
10    REPEAT
11        SET suma = suma + contador;
12        SET contador = contador + 1;
13    UNTIL contador > tope
14    END REPEAT;
15 END
16 $$
17
18 DELIMITER ;
19 CALL ejemplo_bucle_repeat(10, @resultado);
20 SELECT @resultado;
```

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

1.3.2.3 WHILE

```
1 [begin_label:] WHILE search_condition DO
2     statement_list
3 END WHILE [end_label]
```

Ejemplo:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS ejemplo_bucle_while$$
3 CREATE PROCEDURE ejemplo_bucle_while(IN tope INT, OUT suma INT)
4 BEGIN
5     DECLARE contador INT;
6
7     SET contador = 1;
```

```
8   SET suma = 0;
9
10  WHILE contador <= tope DO
11      SET suma = suma + contador;
12      SET contador = contador + 1;
13  END WHILE;
14  END
15  $$
16
17  DELIMITER ;
18  CALL ejemplo_bucle_while(10, @resultado);
19  SELECT @resultado;
```

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

1.4 Manejo de errores

1.4.1 DECLARE ... HANDLER

```
1  DECLARE handler_action HANDLER
2      FOR condition_value [, condition_value] ...
3      statement
4
5  handler_action:
6      CONTINUE
7      | EXIT
8      | UNDO
9
10 condition_value:
11     mysql_error_code
12     | SQLSTATE [VALUE] sqlstate_value
13     | condition_name
14     | SQLWARNING
15     | NOT FOUND
16     | 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.

```
1 DECLARE CONTINUE HANDLER FOR 1051
2 BEGIN
3   -- body of handler
4 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**.

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

Ejemplo para **SQLWARNING**:

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

```
1 DECLARE CONTINUE HANDLER FOR SQLWARNING
2 BEGIN
3   -- body of handler
4 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.

```
1 DECLARE CONTINUE HANDLER FOR NOT FOUND
2 BEGIN
3   -- body of handler
4 END;
```

Ejemplo para **SQLEXCEPTION**::

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

```
1 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
2 BEGIN
3   -- body of handler
4 END;
```

1.4.2 Ejemplo 1 - DECLARE CONTINUE HANDLER

```
1 -- Paso 1
2 DROP DATABASE IF EXISTS test;
3 CREATE DATABASE test;
4 USE test;
5
6 -- Paso 2
```

```
7 CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));
8
9 -- Paso 3
10 DELIMITER $$
11 CREATE PROCEDURE handlerdemo ()
12 BEGIN
13     DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
14     SET @x = 1;
15     INSERT INTO test.t VALUES (1);
16     SET @x = 2;
17     INSERT INTO test.t VALUES (1);
18     SET @x = 3;
19 END
20 $$
21
22 DELIMITER ;
23 CALL handlerdemo();
24 SELECT @x;
```

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

1.4.3 Ejemplo 2 - DECLARE EXIT HANDLER

```
1 -- Paso 1
2 DROP DATABASE IF EXISTS test;
3 CREATE DATABASE test;
4 USE test;
5
6 -- Paso 2
7 CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));
8
9 -- Paso 3
10 DELIMITER $$
11 CREATE PROCEDURE handlerdemo ()
12 BEGIN
13     DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET @x = 1;
14     SET @x = 1;
15     INSERT INTO test.t VALUES (1);
16     SET @x = 2;
17     INSERT INTO test.t VALUES (1);
18     SET @x = 3;
19 END
20 $$
21
22 DELIMITER ;
23 CALL handlerdemo();
24 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:

```
1 DELIMITER $$
2 CREATE PROCEDURE transaccion_en_mysql()
3 BEGIN
4     DECLARE EXIT HANDLER FOR SQLEXCEPTION
5         BEGIN
6             -- ERROR
7             ROLLBACK;
8         END;
9
10    DECLARE EXIT HANDLER FOR SQLWARNING
11        BEGIN
12            -- WARNING
13            ROLLBACK;
14        END;
15
16    START TRANSACTION;
17        -- Sentencias SQL
18    COMMIT;
19 END
20 $$
```

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

```
1 DELIMITER $$
2 CREATE PROCEDURE transaccion_en_mysql()
3 BEGIN
4     DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
5         BEGIN
6             -- ERROR, WARNING
7             ROLLBACK;
8         END;
9
10    START TRANSACTION;
11        -- Sentencias SQL
12    COMMIT;
13 END
14 $$
```

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:

```
1 DECLARE cursor_name CURSOR FOR select_statement
```

1.6.1.2 OPEN

Una vez que hemos declarado un cursor tenemos que abrirlo con **OPEN**.

```
1 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:

```
1 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.

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

1.6.1.4 CLOSE

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

```
1 CLOSE cursor_name
```

Ejemplo:

```
1 -- Paso 1
2 DROP DATABASE IF EXISTS test;
3 CREATE DATABASE test;
4 USE test;
5
6 -- Paso 2
7 CREATE TABLE t1 (
8     id INT UNSIGNED PRIMARY KEY,
9     data VARCHAR(16)
10 );
11
12 CREATE TABLE t2 (
13     i INT UNSIGNED
14 );
15
16 CREATE TABLE t3 (
17     data VARCHAR(16),
18     i INT UNSIGNED
19 );
20
21 INSERT INTO t1 VALUES (1, 'A');
22 INSERT INTO t1 VALUES (2, 'B');
23
24 INSERT INTO t2 VALUES (10);
25 INSERT INTO t2 VALUES (20);
26
27 -- Paso 3
28 DELIMITER $$
29 DROP PROCEDURE IF EXISTS curdemo$$
30 CREATE PROCEDURE curdemo()
31 BEGIN
32     DECLARE done INT DEFAULT FALSE;
33     DECLARE a CHAR(16);
34     DECLARE b, c INT;
35     DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
36     DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
37     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
38
39     OPEN cur1;
40     OPEN cur2;
41
42     read_loop: LOOP
43         FETCH cur1 INTO b, a;
44         FETCH cur2 INTO c;
45         IF done THEN
46             LEAVE read_loop;
47         END IF;
48         IF b < c THEN
49             INSERT INTO test.t3 VALUES (a,b);
50         ELSE
51             INSERT INTO test.t3 VALUES (a,c);
52         END IF;
53     END LOOP;
54
55     CLOSE cur1;
56     CLOSE cur2;
```

```
57 END
58
59 -- Paso 4
60 DELIMITER ;
61 CALL curdemo();
62
63 SELECT * FROM t3;
```

Solución utilizando un bucle WHILE:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS curdemo$$
3 CREATE PROCEDURE curdemo()
4 BEGIN
5     DECLARE done INT DEFAULT FALSE;
6     DECLARE a CHAR(16);
7     DECLARE b, c INT;
8     DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
9     DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
10    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
11
12    OPEN cur1;
13    OPEN cur2;
14
15    WHILE done = FALSE DO
16        FETCH cur1 INTO b, a;
17        FETCH cur2 INTO c;
18
19        IF done = FALSE THEN
20            IF b < c THEN
21                INSERT INTO test.t3 VALUES (a,b);
22            ELSE
23                INSERT INTO test.t3 VALUES (a,c);
24            END IF;
25        END IF;
26    END WHILE;
27
28    CLOSE cur1;
29    CLOSE cur2;
30 END;
```

1.7 Triggers

```
1 CREATE
2     [DEFINER = { user | CURRENT_USER }]
3     TRIGGER trigger_name
4     trigger_time trigger_event
5     ON tbl_name FOR EACH ROW
6     [trigger_order]
7     trigger_body
8
9     trigger_time: { BEFORE | AFTER }
```



```
11 trigger_event: { INSERT | UPDATE | DELETE }
12
13 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:

Crema 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.

```
1 -- Paso 1
2 DROP DATABASE IF EXISTS test;
3 CREATE DATABASE test;
4 USE test;
5
6 -- Paso 2
7 CREATE TABLE alumnos (
8     id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
```

```
9     nombre VARCHAR(50) NOT NULL,
10     apellido1 VARCHAR(50) NOT NULL,
11     apellido2 VARCHAR(50),
12     nota FLOAT
13 );
14
15 -- Paso 3
16 DELIMITER $$
17 DROP TRIGGER IF EXISTS trigger_check_nota_before_insert$$
18 CREATE TRIGGER trigger_check_nota_before_insert
19 BEFORE INSERT
20 ON alumnos FOR EACH ROW
21 BEGIN
22     IF NEW.nota < 0 THEN
23         set NEW.nota = 0;
24     ELSEIF NEW.nota > 10 THEN
25         set NEW.nota = 10;
26     END IF;
27 END
28
29 DELIMITER $$
30 DROP TRIGGER IF EXISTS trigger_check_nota_before_update$$
31 CREATE TRIGGER trigger_check_nota_before_update
32 BEFORE UPDATE
33 ON alumnos FOR EACH ROW
34 BEGIN
35     IF NEW.nota < 0 THEN
36         set NEW.nota = 0;
37     ELSEIF NEW.nota > 10 THEN
38         set NEW.nota = 10;
39     END IF;
40 END
41
42 -- Paso 4
43 DELIMITER ;
44 INSERT INTO alumnos VALUES (1, 'Pepe', 'López', 'López', -1);
45 INSERT INTO alumnos VALUES (2, 'María', 'Sánchez', 'Sánchez', 11);
46 INSERT INTO alumnos VALUES (3, 'Juan', 'Pérez', 'Pérez', 8.5);
47
48 -- Paso 5
49 SELECT * FROM alumnos;
50
51 -- Paso 6
52 UPDATE alumnos SET nota = -4 WHERE id = 3;
53 UPDATE alumnos SET nota = 14 WHERE id = 3;
54 UPDATE alumnos SET nota = 9.5 WHERE id = 3;
55
56 -- Paso 7
57 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`. Resuelva el procedimiento haciendo uso de la estructura de control `IF`.
8. Resuelva el procedimiento diseñado en el ejercicio anterior haciendo uso de la estructura de control `CASE`.

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 `María`.

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 alumnos 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 alumnos 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.
- La dirección de email debe estar en minúsculas.

También deberá crear una **función** llamada `eliminar_acentos` que reciba una cadena de caracteres y devuelva la misma cadena 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`.

- Función: `eliminar_acentos`
- Entrada:
 - `cadena` (cadena de caracteres)
- Salida:
 - (cadena de caracteres)

El procedimiento `crear_email` deberá hacer uso de la función `eliminar_acentos`.

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 añada 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?

```
1 CALL resolver_ejercicio2()
```

3. ¿Cuáles de los siguientes bloques son correctos?

```
1 1.  
2 LOOP bucle:  
3   statements  
4 END bucle:  
5  
6 2.  
7 bucle: LOOP  
8   statements  
9 END bucle;  
10  
11 3.  
12 bucle:  
13 LOOP bucle;  
14   statements;  
15 END bucle;
```

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

```
1 DECLARE a INT;  
2 DECLARE a INT;
```

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

```
1 DECLARE a INT;  
2 DECLARE a FLOAT;
```

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

```
1 DECLARE b VARCHAR(20);  
2 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`

```
1 DELIMITER $$  
2 CREATE PROCEDURE incrementor (OUT i INT)  
3 BEGIN  
4 REPEAT  
5 SET i = i + 1;  
6 UNTIL i > 9  
7 END REPEAT;  
8 END;  
9  
10 DELIMITER $$  
11 CREATE PROCEDURE test ()  
12 BEGIN  
13 DECLARE value INT default 0;  
14 CALL incrementor(value);  
15  
16 -- ¿Qué valor se muestra en esta sentencia?  
17 SELECT value;  
18 END;  
19  
20 DELIMITER ;  
21 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`

```
1 DELIMITER $$
2 CREATE PROCEDURE incrementor (IN i INT)
3 BEGIN
4     REPEAT
5         SET i = i + 1;
6     UNTIL i > 9
7     END REPEAT;
8 END;
9
10 DELIMITER $$
11 CREATE PROCEDURE test ()
12 BEGIN
13     DECLARE value INT default 0;
14     CALL incrementor(value);
15
16     -- ¿Qué valor se muestra en esta sentencia?
17     SELECT value;
18 END;
19
20 DELIMITER ;
21 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 (DECIMAL)

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 (DECIMAL)

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 (DECIMAL)

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 utilizar las funciones `calcular_suma_pedidos_cliente` y `calcular_suma_pagos_cliente`, que ha desarrollado en los ejercicios anteriores.

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

- `codigo_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

```
1 -- Paso 1
2 CREATE TABLE t (s1 INT, PRIMARY KEY (s1));
3
4 -- Paso 2
5 DELIMITER $$
6 CREATE PROCEDURE handlerexam(IN a INT, IN b INT, IN c INT, OUT x INT)
7 BEGIN
8     DECLARE EXIT HANDLER FOR 1036 SET x = 10;
9     DECLARE EXIT HANDLER FOR 1062 SET x = 30;
10
11     SET x = 1;
12     INSERT INTO t VALUES (a);
13     SET x = 2;
14     INSERT INTO t VALUES (b);
15     SET x = 3;
16     INSERT INTO t VALUES (c);
17     SET x = 4;
18 END
19 $$
```

¿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.

```
1 -- a)
2 CALL handlerexam(1, 2, 3, @x);
3 SELECT @x;
4
5 -- b)
6 CALL handlerexam(1, 2, 1, @x);
7 SELECT @x;
```

14. Dado el siguiente procedimiento:

```
1 -- Paso 1
2 CREATE TABLE t (s1 INT, PRIMARY KEY (s1));
3
4 -- Paso 2
5 DELIMITER $$
6 CREATE PROCEDURE test(IN a INT, OUT b INT)
7 BEGIN
8     SET b = 0;
9     WHILE a > b DO
10         SET b = b + 1;
11         IF b != 2 THEN
12             INSERT INTO t VALUES (b);
13         END IF
14     END WHILE;
15 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.

```
1 -- a)
2 CALL test(-10, @value);
3 SELECT @value;
4
5 -- b)
6 CALL test(10, @value);
7 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`.
- Calcular la cantidad total de productos vendidos. En este paso será necesario utilizar la función `cantidad_total_de_productos_vendidos` desarrollada en el ejercicio anterior.
- 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 Gultzan.
- [MySQL Stored Procedures](#). MySQL Tutorial.

2 Licencia

Esta página forma parte del curso Bases de Datos de José Juan Sánchez Hernández y su contenido se distribuye bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional.