Unidad 10 Programación de bases de datos Parte I

1 INTRODUCCIÓN. PROCEDIMIENTOS Y FUNCIONES

Los procedimientos almacenados y funciones son nuevas funcionalidades de la versión de MySQL 5.0. Un procedimiento almacenado es un conjunto de comandos SQL que pueden almacenarse en el servidor. Es un programa que se almacena físicamente en una tabla dentro del sistema de bases de datos. Este programa esta hecho con un lenguaje propio de cada Gestor de BD y esta compilado, por lo que la velocidad de ejecución será muy rápida. Una vez que se hace, los clientes no necesitan relanzar los comandos individuales pero pueden en su lugar referirse al procedimiento almacenado.

Algunas situaciones en que los procedimientos almacenados pueden ser particularmente útiles:

- Cuando múltiples aplicaciones cliente se escriben en distintos lenguajes o funcionan en distintas plataformas, pero necesitan realizar la misma operación en la base de datos.
- Cuando la seguridad es muy importante. Los bancos, por ejemplo, usan procedimientos almacenados para todas las operaciones comunes. Esto proporciona un entorno seguro y consistente, y los procedimientos pueden asegurar que cada operación se loguea apropiadamente. En tal entorno, las aplicaciones y los usuarios no obtendrían ningún acceso directo a las tablas de la base de datos, sólo pueden ejecutar algunos procedimientos almacenados.

Los procedimientos almacenados pueden mejorar el rendimiento ya que se necesita enviar menos información entre el servidor y el cliente. Esto hace que se aumente la carga del servidor de la base de datos ya que la mayoría del trabajo se realiza en la parte del servidor y no en el cliente.

2 PROCEDIMIENTOS ALMACENADOS

Desde MySQL 5.0.3, el sistema de permisos se ha modificado para tener en cuenta los procedimientos almacenados como sigue:

- El permiso `CREATE ROUTINE` se necesita para crear procedimientos almacenados.
- El permiso `ALTER ROUTINE` se necesita para alterar o borrar procedimientos almacenados. Este permiso se da automáticamente al creador de una rutina.
- El permiso `EXECUTE` se requiere para ejecutar procedimientos almacenados. Sin embargo, este permiso se da automáticamente al creador de la rutina. También, la característica `SQL SECURITY` por defecto para una rutina es `DEFINER`, lo que permite a los usuarios que tienen acceso a la base de datos ejecutar la rutina asociada.

Vamos a ver un ejemplo :

Abrimos una conexión en mysql, creamos una base de datos de nombre personas y seleccionamos esa base de datos:

```
CREATE DATABASE PERSONAS;
USE PERSONAS;
```

Vamos a **crear dos tablas** en una almacenaremos las personas mayores de 18 años y en otra las personas menores.

```
CREATE TABLE ninos(edad INT, nombre VARCHAR(50));
CREATE TABLE adultos(edad INT, nombre VARCHAR(50));
```

Queremos introducir personas en las tablas pero dependiendo de la edad queremos que se introduzcan en una tabla, para eso construimos el siguiente procedimiento almacenado:

```
delimiter //
CREATE PROCEDURE introducePersona(IN edad INT,IN nombre VARCHAR(50))
```

```
BEGIN
IF edad <18 THEN
  INSERT INTO ninos VALUES(edad,nombre);
ELSE
  INSERT INTO adultos VALUES(edad,nombre);
END IF;
END;

//
```

Ya tenemos nuestro procedimiento , vamos a por una visión más detallada:

La primera linea es para decirle a MySQL que a partir de ahora hasta que no introduzcamos // no se acaba la sentencia, esto lo hacemos así por que en nuestro procedimiento almacenado tendremos que introducir el carácter ";" para las sentencias, y si pulsamos enter MySQL pensará que ya hemos acabado la consulta y dará error.

Con create procedure empezamos la definición de procedimiento con nombre introducePersona. En un procedimiento almacenado existen parámetros de entrada y de salida, los de entrada (precedidos de "in") son los que le pasamos para usar dentro del procedimiento y los de salida (precedidos de "out")

Para hacer una llamada a nuestro procedimiento almacenado usaremos la sentencia call:

```
CALL introducePersona(25,'Maria Jose');
```

La **sintaxis de un procedimiento almacenado** es la siguiente:

```
CREATE PROCEDURE nombre_procedim ([parametros[,...]])
    [characteristic ...] routine_body
```

donde:

```
parametros
    [ IN | OUT | INOUT ] nombre_param type
      type:
      Es cualquier tipo de datos valido en MySQL
characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

routine_body:
    procedimientos almacenados o comandos SQL válidos
```

La lista de parámetros entre paréntesis debe estar siempre presente. Si no hay parámetros, se debe usar una lista de parámetros vacía () . Cada parámetro es un parámetro IN por defecto. Para especificar otro tipo de parámetro, use la palabra clave OUT o INOUT antes del nombre del parámetro. Especificando IN, OUT, o INOUT sólo es valido para una PROCEDURE.

Uso de las variables en procedimientos:

Veamos un ejemplo:

```
delimiter //
CREATE PROCEDURE miProc(IN p1 INT)     /* Parámetro de entrada */
    BEGIN
        DECLARE miVar INT;        /* se declara variable local */
        SET miVar = p1 +1 ;        /* se establece la variable */
        IF miVar = 12 THEN
            INSERT INTO lista VALUES(55555);
        ELSE
            INSERT INTO lista VALUES(7665);
        END IF;
    END;
//
```

Veamos otro ejemplo con un parámetro de salida:

```
DELIMITER //
CREATE PROCEDURE contar_pedidos (OUT total_pedidos INT)
     BEGIN
          SELECT COUNT(*) INTO total_pedidos FROM pedidos;
     END //

DELIMITER ; -- Llamar al procedimiento
CALL contar_pedidos(@num_pedidos);
-- Ver el valor del parámetro de salida
SELECT @num_pedidos;
```

Veamos un último ejemplo con un parámetro de entrada-salida IN OUT

Queremos crear un procedimiento almacenado que reciba una cadena de texto y un carácter espaciador El procedimiento dividirá la cadena utilizando el delimitador y devolverá la primera parte de la cadena modificada (en mayúsculas) y la cadena original completa (también modificada, con un prefijo).

```
DELIMITER //

CREATE PROCEDURE manipular_cadena (
    INOUT cadena_entrada_salida VARCHAR(255),
    IN caracter_entrada CHAR(1)
)
BEGIN
    -- Declarar una variable local para almacenar la primera parte de la cadena
    DECLARE primera_parte VARCHAR(255);

    -- Encontrar la posición del carácter espaciador
    SET @posicion = LOCATE(caracter_entrada, cadena_entrada_salida);

    -- Si se encuentra el delimitador
    IF @posicion > 0 THEN
        -- Extraer la primera parte de la cadena
        SET primera_parte = SUBSTR(cadena_entrada_salida, 1, @posicion - 1);
```

```
        -- Convertir la primera parte a mayúsculas y actualizar el parámetro INOUT
        SET cadena_entrada_salida = UPPER(primera_parte);
    ELSE
        -- Si no se encuentra el delimitador, dejar la cadena como está (en
mayúsculas)
        SET cadena_entrada_salida = UPPER(cadena_entrada_salida);
    END IF;

    -- Agregar un prefijo a la cadena original (ahora modificada)
    SET cadena_entrada_salida = CONCAT('Resultado: ', cadena_entrada_salida);

END //

DELIMITER ;
```

Este ejemplo demuestra cómo un parámetro INOUT permite que un procedimiento reciba un valor, lo modifique internamente y luego devuelva el valor modificado a la variable que se utilizó en la llamada al procedimiento.

3 FUNCIONES

Así como existen los procedimientos, también existen las funciones. Para crear una función, MySQL nos ofrece la directiva `CREATE FUNCTION`.

La diferencia entre una función y un procedimiento es que la función devuelve valores. Estos valores pueden ser utilizados como argumentos para instrucciones SQL, tal como lo hacemos normalmente con otras funciones como son, por ejemplo, MAX() o COUNT().

Utilizar la cláusula RETURNS es obligatorio al momento de definir una función y sirve para especificar el tipo de dato que será devuelto (sólo el tipo de dato, no el dato).

Su **sintaxis** es:

```
CREATE FUNCTION nombre_funcion ([parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body
```

Vamos a ver cómo crear funciones en MySQL:

```
DELIMITER //
CREATE FUNCTION holaMundo() RETURNS VARCHAR(20)
BEGIN
    RETURN 'HolaMundo';
END
//
```

Para comprobar que funciona **tecleamos lo siguiente:**

```
SELECT holaMundo();
```

Para **borrar la función** que acabamos de crear :

```
DROP FUNCTION IF EXISTS holaMundo;
```

Uso de variables en funciones

Las variables en las funciones se usan de igual manera que en los procedimientos almacenados, se declaran con la sentencia DECLARE, y se asignan valores con la sentencia SET.

```
DELIMITER //
CREATE FUNCTION holaMundo2() RETURNS VARCHAR(30)
BEGIN
  DECLARE salida VARCHAR(30) DEFAULT 'Hola mundo';
  SET salida = 'Hola mundo con VARIABLES';
  RETURN salida;
END;
//
```

Uso de parámetros en funciones

Veamos un ejemplo sencillo, que suma dos números que le pasamos como parámetro.

```
delimiter //
CREATE FUNCTION suma (a INT, b INT)
RETURNS INT
BEGIN
  DECLARE vsuma INT;
  SET vsuma=a+b;
  RETURN vsuma;
END;
//
```

Para usarla llamamos a la función así:

```
SELECT suma(2,2);
```

4 PARÁMETROS Y VARIABLES

Igual que en otros lenguajes de programación los procedimientos y funciones usan variables y parámetros que determinan la salida del algortimo.

Veamos un pequeño ejemplo:

```
DELIMITER //
DROP PROCEDURE IF EXITS proc1 //
CREATE PROCEDURE proc1 /*nombre del procedimiento*/
(IN parametro1 INTEGER) /*parámetros*/
BEGIN
      DECLARE variable1 INTEGER; /*declaración de variables*/
      DECLARE variable2 INTEGER;
      IF parametro1=17 THEN
            SET variable1=parametro1; /*asignación*/
      ELSE
            SET variable2=30;
      END IF;
      INSERT INTO t VALUES (variable1, variable2);
```

```
END //
```

Encontramos dos clausulas para el manejo de variables:

DECLARE: crea una nueva variable con su nombre y tipo. Los tipos son los usuales de MySQL como char, varchar, int, float, etc. Esta cláusula puede incluir una opción para indicar valores por defecto. Si no se indica, dichos valores serán NULL. Veamos un ejemplo:

```
DECLARE a, b INT DEFAULT 5; /*Crea dos variables enteras con valor por defecto 5*/
```

SET: permite asignar valores a las variables usando el operador de igualdad.

Tipos de parámetros

Existen tres tipos:

IN: Es el tipo por defecto y sirve para incluir parámetros de entrada que usara el procedimiento. En este caso no se mantienen las modificaciones.

OUT: Parámetros de salida. El procedimiento puede asignar valores a dichos parámetros que son devueltos en la llamada.

INOUT: Permite pasar valores al procedimiento que serán modificados y devueltos en la llamada.

Alcance de las variables

Las variables tienen un alcance que está determinado por el bloque `BEGIN/END` en el que se encuentran.

5 INSTRUCCIONES CONDICIONALES

IF-THEN-ELSE

La sintaxis general es:

```
IF expr1 THEN
….
ELSEIF expr2 THEN
….
ELSE
….
END IF;
```

Ejemplo:

```
DELIMITER //
CREATE FUNCTION determinar_tipo_numero (numero INT)
RETURNS VARCHAR(20) DETERMINISTIC
BEGIN
      IF numero > 0 THEN
            RETURN 'Positivo';
      ELSEIF numero < 0 THEN
            RETURN 'Negativo';
      ELSE
            RETURN 'Cero';
END IF;
END //
```

CASE

La sintaxis general es:

```
CASE expresion
    WHEN value1 THEN statement_list
    [WHEN value2 THEN statement_list] ...
    [ELSE statement_list]
END CASE;
```

Donde expresión es una expresión cuyo valor puede coincidir con uno de los posibles value1, value2, etc. En otro caso se ejecutan las instrucciones seguidas por ELSE.

Ejemplo:

```
DELIMITER //
CREATE PROCEDURE aplicar_descuento_producto (IN producto_id INT, IN categoria VARCHAR(50))
BEGIN
    DECLARE precio_actual DECIMAL(10, 2);
    DECLARE descuento DECIMAL(5, 2);
    DECLARE nuevo_precio DECIMAL(10, 2);
    -- Obtener el precio actual del producto
    SELECT precio INTO precio_actual FROM productos WHERE id = producto_id;
    CASE categoria
        WHEN 'Electrónica' THEN
            SET descuento = 0.10; -- 10% de descuento
        WHEN 'Libros' THEN
            SET descuento = 0.05; -- 5% de descuento
        WHEN 'Ropa' THEN
            SET descuento = 0.15; -- 15% de descuento
        ELSE SET descuento = 0.00; -- Sin descuento para otras categorías
    END CASE;
    SET nuevo_precio = precio_actual * (1 - descuento);
    SELECT CONCAT('El nuevo precio del producto ', producto_id, ' (categoría: ', categoria, ') es: ', nuevo_precio) AS mensaje;
END //
```

6 INSTRUCCIONES REPETITIVAS O LOOPS

SIMPLE LOOP

Su sintaxis es:

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

LOOP implementa un constructor de bucle simple que permite ejecución repetida de comandos particulares. El comando dentro del bucle se repite hasta que acaba el bucle, usualmente con un comando LEAVE .

Un comando LOOP puede etiquetarse. end_label no puede darse hasta que esté presente begin_label , y si

ambos lo están, deben ser el mismo

Veamos un ejemplo: Un contador que va hasta desde 0 hasta 4

```
DELIMITER //
CREATE PROCEDURE proc2()
BEGIN
      DECLARE cont INT;
      SET cont = 0;
      loop_label: LOOP
            INSERT INTO t VALUES (cont);
            SET cont=cont+1;
            IF cont>=5 THEN
                  LEAVE loop_label; /*El bucle se termina*/
            END IF;
      END LOOP;
END; //
```

REPEAT UNTIL LOOP

Su sintaxis es:

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

El comando/s dentro de un comando REPEAT se repite hasta que la condición search_condition es cierta.

Un comando REPEAT puede etiquetarse. end_label no puede darse a no ser que begin_label esté presente, y si lo están, deben ser el mismo.

Veamos un ejemplo, mostramos los números impares desde 1 a 10

```
      DELIMITER //
CREATE PROCEDURE proc3()
BEGIN
      DECLARE i INT;
      SET i = 0;
      loopl: REPEAT
            SET i=i+1;
            IF MOD(i,2)<>0 THEN /*numero impar*/
                  SELECT CONCAT(i, " es impar");
            END IF;
      UNTIL i>=10
      END REPEAT loopl;
      END; //
```

WHILE LOOP

Su sintaxis es:

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

El comado/s dentro de un comando WHILE se repite mientras la condición search_condition es cierta.

Un comando WHILE puede etiquetarse. end_label no puede darse a no ser que begin_label también esté presente, y si lo están, deben ser el mismo.

Veamos el anterior ejemplo usando While loop

```
DELIMITER //
CREATE PROCEDURE proc4()
BEGIN
      DECLARE i INT;
      SET i = 1;
      loopl: WHILE i<=10 DO
            IF MOD(i,2)<>0 THEN /*numero impar*/
                  SELECT CONCAT(i, " es impar");
            END IF;
            SET i=i+1;
      END WHILE loopl;
      END; //
```

7 CURSORES EN MySQL

Se utilizan para trabajar con consultas que pueden devolver más de una fila.

El cursor es un identificador, no es una variable. Solamente se puede usar para hacer referencia a una consulta. No se le pueden asignar valores ni utilizar en expresiones. No obstante, el cursor tiene, al igual que las variables, su ámbito.

Los cursores representan consultas SELECT de SQL que devuelven más de un resultado y que permiten el acceso a cada fila de dicha consulta. Lo cual significa que el cursor siempre tiene un puntero señalando a una de las filas del SELECT que representa el cursor.

Se puede recorrer el cursor haciendo que el puntero se mueva por las filas.

Se soportan cursores simples dentro de procedimientos y funciones almacenadas. La sintaxis es la de SQL empotrado.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar cursores o handlers.

7.1 Declaración de cursores

Para declarar un cursor usamos la siguiente sentencia:

```
DECLARE cursor_name CURSOR FOR select_statement
```

Pueden definirse varios cursores en una rutina, pero cada cursor en un bloque debe tener un nombre único.

El comando `SELECT` no puede tener una cláusula `INTO` .

7.2 Comandos relacionados con cursores

Para manipular los cursores disponemos de una serie de comandos:

- `OPEN cursor_name` → Este comando abre un cursor declarado previamente.

- `FETCH cursor_name INTO variable_list` → extrae la siguiente fila de valores del conjunto de resultados del cursor moviendo su puntero interno una posición.
- `CLOSE cursor_name` → Cierra el cursor liberando la memoria que ocupa y haciendo imposible el acceso a cualquiera de sus datos.

Veamos un ejemplo: Tenemos una tabla libros, para almacenar una serie de libros que nos interesan.

La creación de la tabla sería con:

```
CREATE TABLE libros (id INT, titulo VARCHAR(50), autor VARCHAR(30), fecha DATE);
```

Vamos a crear un procedimiento que nos muestre el número de libros, utilizando cursores:

```
DELIMITER //
CREATE PROCEDURE contar_libros ()
BEGIN
      DECLARE tmp VARCHAR(200);
      DECLARE cursor1 CURSOR FOR SELECT titulo FROM libros;
      OPEN cursor1;
      l_cursor: LOOP
            FETCH cursor1 INTO tmp;
      END LOOP l_cursor;
      CLOSE cursor1;
END; //
```

En el ejemplo anterior se produce un error similar al siguiente:

```
call contar_libros(); → ERROR 1329 (02000): No data to FETCH
```

Dado que cuando llegamos a la última fila no hay más datos que obtener, necesitamos de algún modo detectar ese momento. Para ello usaremos un manejador de errores o handler (que explicamos en el siguiente apartado) Esto se realiza con la siguiente instrucción:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
```

Esto hace dos cosas:

- Establece la variable v_ultima_fila a 1
- Permite al programa continuar su ejecución

El procedimiento quedaría entonces de la siguiente manera:

```
DELIMITER //
CREATE PROCEDURE contar_libros2 ()
BEGIN
      DECLARE tmp VARCHAR(200);
      DECLARE v_ultima_fila BOOL;
      DECLARE nn INT;
      DECLARE cursor2 CURSOR FOR SELECT titulo FROM libros;
      DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
      SET v_ultima_fila=0, nn=0;
      OPEN cursor2;
      /*Utilizamos ahora un bucle pare recorrer el cursor*/
```

```
        l_cursor2: LOOP
              FETCH cursor2 INTO tmp;
                    IF v_ultima_fila=1 THEN
                          LEAVE l_cursor2;
                    END IF;
              SET nn=nn+1;
              END LOOP l_cursor2;
        CLOSE cursor2;
        SELECT nn;
END; //
```

Observar que hemos declarado una variable v_ultima_fila, booleana, con posibles valores 0 y 1, indicando si hemos llegado o no a la última fila del cursor. La variable nn almacena el número de libros o registros contenidos en el cursor. Gracias a la sentencia LEAVE, podemos terminar el bucle cuando v_ultima_fila adquiera el valor 1, o lo que es lo mismo, se alcanza el final del cursor.

7.3 Recorrer un cursor

Veamos otras maneras de recorrer el cursor, utilizaremos el ejemplo anterior.

Cursor con repeat until

```
DELIMITER //
CREATE PROCEDURE contar_libros3 ()
BEGIN
      DECLARE tmp VARCHAR(200);
      DECLARE v_ultima_fila BOOL;
      DECLARE nn INT;
      DECLARE cursor3 CURSOR FOR SELECT titulo FROM libros;
      DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
      SET v_ultima_fila=1, nn=0;
      OPEN cursor3;
      /*Utilizamos ahora un bucle pare recorrer el cursor*/
      l_cursor3: REPEAT
            FETCH cursor3 INTO tmp;
            IF v_ultima_fila=1 THEN
                  LEAVE l_cursor3;
            END IF;
            SET nn=nn+1;
            UNTIL v_ultima_fila=1
      END REPEAT l_cursor3;
      CLOSE cursor3;
      SELECT nn;
END; //
```

Cursor con while

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `contar_libros4`()
BEGIN
      DECLARE tmp VARCHAR(200);
      DECLARE v_ultima_fila BOOL;
      DECLARE nn INT;
```

```
        DECLARE cursor4 CURSOR FOR SELECT titulo FROM libros;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
        SET v_ultima_fila=0, nn=0;
        OPEN cursor4;
        /*Utilizamos ahora un bucle pare recorrer el cursor*/
        FETCH cursor4 INTO tmp;
        l_cursor4: WHILE(v_ultima_fila=0) DO
              SET nn=nn+1;
              FETCH cursor4 INTO tmp;

              /*IF v_ultima_fila=1 THEN
                    LEAVE l_cursor4;
              END IF;*/

        END WHILE l_cursor4;
        CLOSE cursor4;
        SELECT nn;
END
```

Posiblemente esta última es la construcción más usada ya que, a diferencia de las anteriores, se evalúa la condición antes de leer un registro del cursor.

8 GESTIÓN DE RUTINAS ALMACENADAS

Eliminación de rutinas

Para eliminar procedimientos o funciones usamos el comando SQL DROP con la siguiente sintaxis:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] nombre_pr;
```

Consultar rutinas

Podemos ver información más o menos detallada de nuestras rutinas usando los comandos:

```
SHOW CREATE {PROCEDURE | FUNCTION} nombre_pr
SHOW {PROCEDURE | FUNCTION}  STATUS [LIKE 'pattern']
```

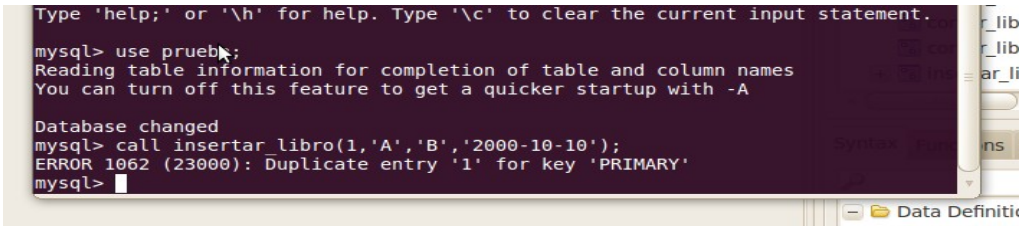
9 MANEJO DE ERRORES

Cuando un programa almacenado encuentra una condición de error, la ejecución se detiene y se devuelve un error a la aplicación que llama. Este es el comportamiento predeterminado. Pero podemos cambiar este comportamiento, es decir, queremos controlarlo. Para hacer esto tenemos que definir controladores de excepciones en nuestros programas. Esto lo hemos visto en los cursores.

Veamos un ejemplo de procedimiento sin manejo de errores:

```
DELIMITER //
CREATE PROCEDURE insertar_libro
(id_p INT, titulo_p  VARCHAR(30), autor_p VARCHAR(30), fecha_p DATE)
BEGIN
INSERT INTO libros VALUES (id_p, titulo_p,autor_p,fecha_p);
END;//
```

Funciona bien cuando no existe el registro, pero si intentamos insertar un libro ya existente, MySQL genera un error.



10

Indica la existencia de una clave repetida en el campo id.

El mismo ejemplo con manejo de errores sería:

```
DELIMITER //
CREATE PROCEDURE insertar_libro
(id_p INT, titulo_p  VARCHAR(30), autor_p VARCHAR(30), fecha_p DATE)
BEGIN
      DECLARE estado VARCHAR(20);
      DECLARE CONTINUE HANDLER FOR 1062 SET estado='Duplicate Entry';
      SET estado='OK';
      INSERT INTO libros VALUES (id_p, titulo_p,autor_p,fecha_p);
END;//
```

Pero si queremos hacer algo con el error debemos usar la variable estado. Veamos en el siguiente ejemplo como usar la salida del valor de la variable estado.

```
DELIMITER //
CREATE PROCEDURE insertar_libro2
(id_p INT, titulo_p  VARCHAR(30), autor_p VARCHAR(30), fecha_p DATE)
BEGIN
      DECLARE estado VARCHAR(20);
      DECLARE CONTINUE HANDLER FOR 1062 SET estado='Duplicate Entry';
      SET estado='OK';
      INSERT INTO libros VALUES (id_p, titulo_p,autor_p,fecha_p);
      IF estado='Duplicate Entry' THEN
            SELECT CONCAT (titulo_p,' es un ibro repetido');
      ELSE
            SELECT 'Insercion correcta';
      END IF;
END;//
```

10.1 Sintaxis del manejador

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
         {SQLSTATE sqlstate_code | MySQL error code | condition name}
```

- **Tipo de manejador**: EXIT o CONTINUE
- **Condición del manejador**: estado SQL (SQLSTATE), error propio de MySQL,o código de error definido por

el usuario.

- Acciones del manejador: acciones a tomar cuando se active el manejador.

Tipos de manejador

- **EXIT:** Cuando se encuentra un error el bloque que se está ejecutando actualmente se termina. Si ese bloque es el bloque principal el procedimiento termina, y el control se devuelve al procedimiento o programa externo que invocó el procedimiento. Si el bloque está encerrado en un bloque externo dentro del mismo programa almacenado, el control se devuelve al bloque exterior.
- **CONTINUE:** para el caso de CONTINUE, la ejecución continúa en la declaración siguiente a la que ocasionó el error

Veamos algunos ejemplos de ambos tipos de controladores.

En el primer ejemplo el procedimiento inserta un registro en libros. Para manejar la posibilidad de que el libro ya exista se crea el manejador EXIT que en caso de activarse establecerá el valor de la variable duplicate_key a 1 y devolverá el control al bloque BEGIN/END exterior.

```
DELIMITER //
CREATE PROCEDURE insertar_libro3
(id_p INT, titulo_p  VARCHAR(30), autor_p VARCHAR(30), fecha_p DATE)
BEGIN
      DECLARE duplicate_key INT DEFAULT 0;
      BEGIN
            DECLARE EXIT HANDLER FOR 1062 SET duplicate_key=1;
            INSERT INTO libros VALUES (id_p, titulo_p,autor_p,fecha_p);
      END;

      IF duplicate_key=1 THEN
            SELECT CONCAT (titulo_p,' es un ibro repetido');
      ELSE
            SELECT 'Insercion correcta';
      END IF;
END;//
```

Un ejemplo de la misma funcionalidad implementado con un controlador CONTINUE sería:

```
DELIMITER //
CREATE PROCEDURE insertar_libro4
(id_p INT, titulo_p  VARCHAR(30), autor_p VARCHAR(30), fecha_p DATE)
BEGIN
      DECLARE duplicate_key INT DEFAULT 0;
      DECLARE CONTINUE HANDLER FOR 1062 SET duplicate_key=1;
      INSERT INTO libros VALUES (id_p, titulo_p,autor_p,fecha_p);

      IF duplicate_key=1 THEN
            SELECT CONCAT (titulo_p,' es un ibro repetido');
      ELSE
            SELECT 'Insercion correcta';
      END IF;
END;//
```

Un controlador EXIT va mejor con los errores catastróficos ya que no se permite ninguna forma de continuación de la tramitación.

Un controlador de CONTINUE es más adecuado cuando se tiene un procedimiento alternativo que se ejecutará si la excepción se produce.