Tema 8 – Programación Base de Datos

1º DAW – Bases de Datos

J. Carlos Moreno

Curso 2022/2023

Contenido

8 Programación de Bases de Datos.

Toda aplicación informática consta de dos partes fundamentales y bien diferenciadas:

- Bases de Datos de tipo relacional u orientada a objetos
- El código de programa o las funciones que manipulan dichos datos para conseguir la funcionalidad deseada

Normalmente esto último se logra mediante lenguajes de programación com C, php, perl, java, etc…, para bases de datos relacionales o, de modo más avanzado C++, Java o Visual .NET para bases objetos relacionales (bases relacionales con características de objetos) u orientadas a objetos. Sin embargo cada vez más los SGBD incorporan lenguajes propios que permiten integrar datos y funcionalidad dentro de la misma base de datos. Esto tiene varias ventajas:

- Independencia del Sistema Operativo. No requiere librerías o programas especiales para usar el lenguaje de programación.
- Aplicaciones más ligeras. Parte de la carga del proceso se incorpora en el servidor con lo que las aplicaciones requieren menos desarrollo y código.
- Facilidad de mantenimiento. Solamente se requiere actualizar el propio gestor sin necesidad de ajustes ajenos a él. Además, las modificaciones en las aplicaciones son menores al tener repartida la funcionalidad en el SGBD.

En este tema veremos una panorámica de la tecnología actual respecto a los lenguajes de programación en varios SGBD para después centrarnos en la programación dentro de MySQL.

8.1 Lenguajes de Programación y Bases de Datos.

Tradicionalmente las bases de datos se han limitado a servir de repositorios de datos organizados según ciertas relaciones en tablas. Dichos datos eran accedidos mediante interfaces de lenguajes de programación. Algunos de estos lenguajes permitían la inclusión de sentencias SQL como parte del código de aplicación, es lo que se conoce lenguajes de tipo anfitrión como Java, PL/I o C. Esa características se ha modificado para dar paso a API, funciones incorporadas por distintos lenguajes para acceder a servidores de datos. Así cada vez más lenguajes (perl, python, Ruby, php, etc.) amplían sus posibilidades en cuanto al acceso a bases de datos.

Al mismo tiempo, los propios SGBD van incorporando cada vez más potentes lenguajes propios integrados en el software y que minimizan el desarrollo de aplicaciones en la parte de acceso a datos. Como ejemplos podemos citar los siguientes:

- MySQL. Permite la definición de rutinas, disparadores, vistas y eventos mediante un lenguaje propio sin nombre específico, por lo que le llamaremos lenguaje MySQL
- Oracle. incorpora el llamado PL/SQL para la programación de objetos de la base de datos.
- SQL Server. incorpora el llamado Transact-SQL o T-SQL para la implementación de sentencias SQL, así como para programación de rutinas, disparadores y otros objetos.

- PostgreSQL. Permite mediante el uso de módulos ser compilado para usar lenguajes diversos, el más habitual es PL/PGSQL, pero existen muchos otros como PL/PHP, PL/R o PL/Java.

En la actualidad se están dando cambios significativos ya que se están incorporando características de la programación orientada a objetos a las bases de datos relacionales. Esto es así debido al auge de estos lenguajes y al uso de los mismos de tipos avanzados de datos (como objetos, métodos, herencia, arrays, etc.) no soportado por las bases de datos tradicionales.

8.2 Lenguaje de Programación MySQL

Has ahora hemos venido trabajando con MySQL a modo comando, hemos escrito un comando SQL en un entorno de desarrollo en nuestro caso MySQL Workbench lo hemos ejecutado y a continuación nos ha mostrado los resultados.

Este enfoque está bien como aprendizaje pero no nos servirá para crear grandes aplicaciones que dependan de MySQL como recurso de datos. Necesitamos un herramienta que de alguna manera pueda automatizar las consultas que queremos ejecutar cumpliendo con todos los requisitos de gestión y seguridad requeridos, afortunadamente hoy día MySQL al igual que la mayoría de sistemas gestores de bases de datos ofrece la posibilidad de cubrir estas necesidades mediante su lenguaje de programación MySQL a través del cual podremos crear mecanizar nuestras tareas para grandes aplicaciones mediante las **rutinas almacenadas**.

Las rutinas almacenadas se crean usando el lenguaje de programación nativo de MySQL y pueden ser de tres tipos:

- Procedimientos
- Funciones
- Triggers (disparadores)

En apartados posteriores estudiaremos con detalle cada uno de los apartados anteriores.

8.3 Variables en MySQL

En esta sección veremos cómo trabajar con variables en nuestro nuevo lenguaje de programación.

Cuando trabajamos con cualquier lenguaje de programación necesitamos declarar variables que servirán com recipientes donde voy a almacenar un valor o un conjunto de valores. Las variables se caracterizan por:

- Nombre
- Tipo de dato que almacenan

Una variable podrá cambiar de valor a lo largo de la ejecución de un programa, el valor que devolverá una variable será el último que se le haya asignado.

En MySQL es un lenguaje NO TIPIFICADO es decir, que no no hay que especificar el tipo de dato que va a almacenar una variable, el tipo de dato lo asume al partir del valor que se le ha asignado.

La sintaxis para declarar una variable en MySQL es la siguiente:

```
SET @variable = valor;
```

Veamos ahora los siguientes ejemplos:

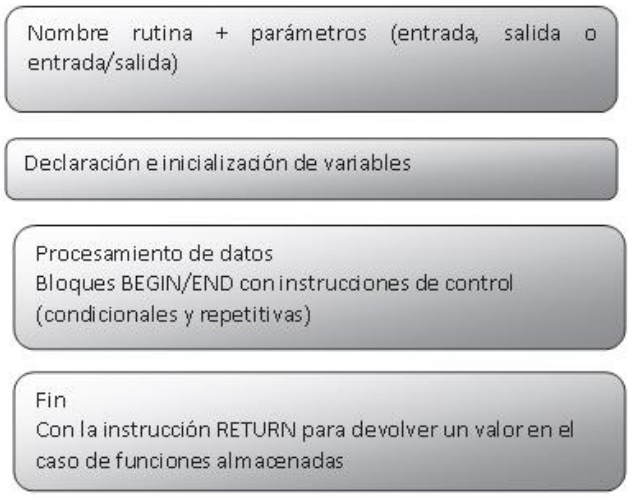
```
-- Declaramos una variable tipo VARCHAR
SET @nombre = 'Juan Carlos';

-- Mostramos el valor de dicha variable
SELECT @nombre;

-- Podremos asignar a una variable el resultado de expresiones
SET @suma = 35 + 45;

-- Podremos asignar el resultado de una consulta
SET @fecha_nacimiento = '1991/12/3'
SET @edad = TIMESTAMPDIFF(YEAR, @fecha_nacimiento, NOW());

-- Podemos ahora hacer que muestre varios valores con el SELECT
SELECT @fecha_nacimiento, @edad;

-- Podemos hacer que una variable tome el resultado de una consulta
SET @corredor_veterano_id = (SELECT (id) FROM maratoon.corredores
          WHERE edad = (SELECT MAX(edad) from maratoon.corredores)
          LIMIT 1);

-- Ahora esta variable la podemos emplear en una sentencia SQL.
SELECT * FROM maratoon.corredores WHERE id = @corredor_veterano_id;
```

Un aspecto importante es que a las variables MySQL se les antepone el simbolo '**@**', lo cual indicará que son variables **globales** y que las podremos usar dentro de cualquiera de nuestras rutinas almacenas, tanto procedimientos como funciones o triggers.

8.4 Procedicientos en MySQL

En MySQL podemos definir un procedimento de la misma forma y con las mismas características que con cualquier otro lenguaje de programación.

Los procedimientos almacenados aportan enormes beneficios a nuestras aplicaciones. Puede incluir SQL anidados, complejos, con subconsultas, funciones de agregado, agrupación de registros y devolverá un resultado consistente cada vez que lo ejecutemos. Podrán tanto devolver resultados, como añadir, actualizar y eliminar datos de nuestras tablas. Además dentro de un procedimiento almacenado podremos llamar a otro procedimiento almacenada.



Características de los Procedimientos MySQL:

- Normalmente un procedimiento almacenado se crea para que ejecute un comando SQL ya sea DCL, DML o DDL.
- Un procedimiento pude incluir tres tipos de parámetros:
  - IN – Parámetro de entrada
  - OUT – Parámetro de salida
  - INOUT – Parámetro de entrada salida
- Cuando defino un parámetro dentro de un procedimiento e de especificar el tipo de parámetro (IN, OUT, INOUT) y el tipo de dato. Si no se especifica el tipo de parámetro por defecto se considera IN.
- Para declarar una variable dentro de un procedimiento debo usar la cláusula DECLARE e indicar el tipo de dato que va a contener dicha variable
- Los tipos de datos que puedo usar son los mismos que utilizo para definir una columna con CREATE (INTEGER, FLOAT(), DECIMAL(10,2), VARCHAR(), CHAR(), TEXT, TIMESTAMP, DATETIME, TIME, …
- Para asignar un varlor a una variable debo usar la cláusula SET

La sintáxis para declarar un procedimiento almacenado es la siguiente:

```
-- Sintáxis PROCEDMIENTO

CREATE

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

proc_parameter:
    [ IN | OUT | INOUT ] 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
```

Donde:

- **Sp_name**: es el nombre de la rutina
- **Parameter**: entrada (IN), salida (OUT), entrada/salida (INOUT).
- **Type:** cualquier tipo de datos admitido por MYSQL
- **Routine_body**. Es el cuerpo de la rutina formado general mente por sentencias SQL. En caso de haber más de una debe ir delimitado por sentencias BEGIN y END.
- **Deterministic**. Indica si siempre produce el mismo resultado.
- **Contains SQL/no SQL**. Especifica si contiene sentencias SQL.
- **Modifies SQL data/Reads SQL data**. Indica si las sentencias modican los datos.
- **SQL security**. Determina si debe ejecutarse con permisos del creador (define) o del que lo invoca (invoker)

8.4.1 Ejemplo 1. Hola mundo

Normalmente un procedimiento almacenado se crea para que ejecute un comando SQL ya sea DCL, DML o DDL. Empezaremos por el típico programa Hola Mundo que se crea cada vez que nos iniciamos en un nuevo lenguaje de programación.

```
-- Ejemplo 1. Hola mundo

DELIMITER $$
DROP PROCEDURE IF EXISTS hola_mundo $$
CREATE PROCEDURE test.hola_mundo()
BEGIN
          SELECT 'HOLA MUNDO PL/SQL';
END $$
```

Comentamos línea a línea este pequeño procedimiento:

- La palabra clave DELIMITER indica el carácter de comienzo y fin del procedimiento. Típicamente sería “;” pero dado que necesitamos ; para cada sentencia SQL dentro del procedimiento es conveniente usar otro carácter (normalmente $$ o //).
- Eliminamos el procedimiento si es que existe. Esto evita errores cuando queremos modificar un procedimiento existente.
- Indica el comienzo de la definición de un procedimiento donde debe aparecer el nombre seguido por paréntesis entre los que pondremos los parámetros en caso de haberlos. En este caso procedemos al nombre con la base de datos test, a la que pertenecerá dicho procedimiento.
- BEGIN indica el comienzo de una sería de bloques de sentencias SQL que componen el cuerpo del procedimiento cuando hay más de una.
- Conjunto de sentencias SQL, en este caso un SELECT que imprime la cadena por pantalla.
- Fin de la definición del procedimiento seguido de un doble $ indicando que ya hemos terminado.

Para ejecutar este procedimiento desde otra pestaña de Worbench

```
CALL test.hola_mundo();
```

8.4.2 Ejemplo 2. Clasificación

Vamos ahora a crear un procedimiento algo más complejo usando la base de datos **maratoon**.

Este nuevo procedimiento mostrará la **clasificación general** de una determinada carrera. En este caso necesitaremos definir un parámetro de entrada que se corresponderá con el **id** de la carrera de la cual queremos obtener la clasificación general.

```
-- Ejemplo 2.
-- Clasificación general de una carrera
-- Base de datos maratoon

DELIMITER $$
DROP PROCEDURE IF EXISTS maratoon.clasificacion $$
CREATE PROCEDURE maratoon.clasificacion(IN id_carrera INT UNSIGNED)
BEGIN
	SELECT
		r.id Reg,
		co.id,
		co.apellidos,
		co.nombre,
		co.Club_id,
		cl.NombreCorto Club,
		co.Categoria_id,
		ca.NombreCorto Categoria,
		r.TiempoInvertido
	FROM
		maratoon.corredores co
			INNER JOIN
		maratoon.clubs cl ON co.Club_id = cl.id
			INNER JOIN
		maratoon.categorias ca ON co.categoria_id = ca.id
			INNER JOIN
		maratoon.registros r ON co.id = r.corredor_id
	WHERE
		r.carrera_id = id_carrera
	ORDER BY r.TiempoInvertido ASC;
END
```

Ahora desde otra pestaña de Workbench para hacer una llamada a este procedimiento

```
CALL maratoon.clasificacion(1);
```

8.4.3 Ejemplo 3. Actualizar edad.

Ahora vamos a crear un procedimiento para la base de datos maratoon que permita actualizar la edad de todos los corredores. En esta ocasión este nuevo procedimiento no necesitará ningún parámetro.

```
-- Ejemplo 3.
-- Base de datos maratoon
-- Actualizar la edad de todos los corredores

DELIMITER $$
DROP PROCEDURE IF EXISTS maratoon.ActualizarEdad $$
CREATE PROCEDURE maratoon.ActualizarEdad()
BEGIN
	UPDATE Corredores SET Edad = TIMESTAMPDIFF(YEAR,FechaNacimiento,NOW());
END
```

Para ejecutar el procedimiento

```
CALL maratoon.ActualizarEdad();
```

8.4.4 Ejemplo 4. Actualizar categoria_id de la tabla corredores

Este nuevo procedimiento va a contener una operación que ya hemos realizado en temas anteriores y es actaulizar la columna categoria_id en base a la edad de un corredor.

Para complicar un poco el ejercicio, vamos a suponer que la tabla corredores no dispone de la columna edad, sabemos que esta columna es de tipo derivada, es decir, podemos prescindir perfectamente de ella puesto que su valor se puede calcular a partir de la fecha de nacimiento, y eso es lo que vamos a hacer en este ejemplo.

En este procedimiento vamos a necesitar como parámetro:

- Id del corredor del cual queremos actualizar la categoría a la que pertenece

También vamos a necesitar las siguietes variables:

- Fecha_nacimiento. Obtendremos la fecha de nacimiento del corredor a partir del id
- Edad. Calcularemos la edad a partir de la fecha de nacimiento
- Categoria. Almacenaremos el id de categoria con el que tendremos que actualizar el registro de dicho corredor.

La sintáxis para declarar una variable en MySQL dentro de un procedimiento es la siguiente:

```
 DECLARE variable tipo_dato
```

También vamos a usar estructura condicional que se llama estructura CASE la cual estudiaremos más adelante pero que es fácil de interpretar dentro del siguiente ejemplo.

```
-- Ejemplo 4. Actualizar categoria
-- Base de datos maratoon
-- Actualizar la columna categoria_id de corredores
-- Parámetro: id del corredor

DELIMITER $$
DROP PROCEDURE IF EXISTS maratoon.ActaulizarCategoria $$
CREATE PROCEDURE maratoon.ActualizarCategoria(IN corredor_id INT UNSIGNED)
BEGIN

    DECLARE fecha_nacimiento DATE;
    DECLARE edad TINYINT UNSIGNED;
    DECLARE categoria TINYINT UNSIGNED;

    -- Obtenemos la fecha de nacimiento del corredor id
    SET fecha_nacimiento =
        (SELECT FechaNacimiento from maratoon.corredores WHERE id = corredor_id);
```

```
    -- Calculamos la edad a partir de la fecha de nacimiento
    SET edad = TIMESTAMPDIFF(YEAR,fecha_nacimiento,NOW());

    -- Calculamos el id de categoria a partir de la edad
    CASE
        WHEN edad< 12 THEN SET categoria = 1;
        WHEN edad< 14 THEN SET categoria = 2;
        WHEN edad< 17 THEN SET categoria = 3;
        WHEN edad< 29 THEN SET categoria = 4;
        WHEN edad< 39 THEN SET categoria = 5;
        WHEN edad< 49 THEN SET categoria = 6;
        WHEN edad< 60 THEN SET categoria = 7;
        ELSE SET categoria = 8;
    END CASE;

    -- Actualizamos el registro
    UPDATE maratoon.corredores SET categoria_id = categoria WHERE id = corredor_id;
END
```

8.5 Funciones en MySQL

Una función simplifica software, y hace que el software que creamos sea mucho más fácil de mantener, reparar y mejorar. Si un programa tiene el mismo bloque de código idéntico repetido cien veces, el mismo cambio requerido para todos esos bloques requeriría el mismo trabajo y pruebas, cien veces. Esto nos daría un increíblemente enorme ahorro de tiempo, esfuerzo y gastos.

En el capítulo anterior vimos que MySQL ofrece funciones integradas que podemos llamar directamente a las ventanas de consulta de Workbench y en los procedimientos almacenados que podamos desarrollar.

Los productos MySQL y SQL en general, incluyen funciones integradas que incluyen siguientes categorías:

- Fecha y hora
- Cadena
- Matemáticas

MySQL nos ofrece una forma de construir nuestras propias funciones personalizadas. Por ejemplo, podríamos construir fácilmente funciones MySQL que calcular el número de horas entre el tiempo de ejecución de la función y el primer segundo del 1 de enero de ese año. Estas funciones podrían tener un gran valor para las finanzas, seguros y aplicaciones de gestión de inventario.

Las funciones SQL ofrecen muchas de las ventajas de los procedimientos almacenados de SQL como se describió anteriormente, pero una función no ofrece exactamente las mismas ventajas y flexibilidad:

- Las funciones siempre devolverán un valor mediante la cláusula RETURN
- Las funciones a diferencia de los procedimientos sólo admiten parámeros de entrada, mientras que los procedimientos permiten parametros de entrada, salida y entrada/salida (IN, OUT, INOUT)
- Las funciones no pueden actualizar o eliminar datos de la tabla.
- Las funciones no pueden llamar a procedimientos almacenados.

- No se puede llamar a una función SQL tan fácilmente.

Sintáxis para crear una **Función almacenada en MySQL**.

```
-- Sintáxis FUNCIÓN

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
```

Donde:

- **Sp_name**: es el nombre de la función
- **Type:** cualquier tipo de datos admitido por MYSQL (INTEGER, VARCHAR(), CHAR, DATE(), TEXT, …)
- **Routine_body**. Es el cuerpo de la rutina formado generalmente por sentencias SQL. En caso de haber más de una debe ir delimitado por sentencias BEGIN y END.
- **Deterministic**. Indica si siempre produce el mismo resultado.
- **Contains SQL/no SQL**. Especifica si contiene sentencias SQL.
- **Modifies SQL data/Reads SQL data**. Indica si las sentencias modican los datos.
- **SQL security**. Determina si debe ejecutarse con permisos del creador (define) o del que lo invoca (invoker)

Los corchetes indican parámetros opcionales y las palabras en mayúsculas son reservadas de SQL y el resto son opciones que se explican detalladamente en el manual de referencia. Todos los procedimientos o funciones se crean asociadas a una base de datos, que será la activa en ese momento o la que pongamos como prefijo en el nombre del mismo.

En lo sucesivo iremos explicando las cláusulas más típicas con distintos ejemplos.

8.5.1 Función Estado()

Vamos a construir una función que recibe como parámetro uno de los siguientes caracteres P, O y N. La función devolverá respectivamente las cadenas Caducado, Activo y Nuevo.

```
-- Ejmplo04: Función que devuelve el estado de un alimento
-- Parametro es un carácter de tipo CHAR(1)
-- RETURNS – el valor que devuelve esta función es de tipo VARCHAR(20)

DELIMITER $$
DROP FUNCTION IF EXISTS test.estado$$
CREATE FUNCTION test.estado(in_estado CHAR(1))
        RETURNS VARCHAR(20)
BEGIN
        DECLARE estado VARCHAR(20);
        IF in_estado='P' THEN
                SET estado='caducado';
        ELSEIF in_estado='O' THEN
                SET estado='activo';
        ELSEIF in_estado='N' THEN
                SET estado='nuevo';
        END IF;
        RETURN(estado);
END$$
```

Para usar esta función lo podríamos hacer con el siguiente código desde otra pestaña de Workbench

```
SET @EstadoArticulo='N';
SELECT test.estado(@Estadoarticulo);
```

Con el comando *SET @Nombre[=valor]* creamos una variable que se llama de sesión, a la que le podemos asignar incluso un valor. Luego las funciones no pueden ser llamadas con *CALL* como los procedimientos, deben estar incluidas siempre en una expresión del tipo *SET @EstadoArticulo=estado('N')* o bien incluirla en una expresión en la cláusula *SELECT* como se muestra en este ejemplo.

También podríamos haber usado directamente la función estado() de esta forma

```
SELECT test.estado('N');
```

8.5.2 Función EsImpar()

Esta función me devolverá TRUE (1) en caso de que un número sea Impar, en caso contrario devolverá FALSE (0).

```
-- Ejmplo05: Función EsImpar()
-- Parámetro de entrada será un valor de tipo INT
-- RETURNS – Devolverá un valor de tipo BOOLEAN

DELIMITER $$
DROP FUNCTION IF EXISTS test.esimpar$$
CREATE FUNCTION test.esimpar(numero INT)
        RETURNS boolean
BEGIN
        DECLARE impar boolean;
        IF MOD(numero,2)=0 THEN
                SET impar=FALSE;
        ELSE
                SET impar=TRUE;
        END IF;
        RETURN(impar);
END$$
```

Para usar esta función desde otra pestaña de Workbench

```
SET @valor=41;
SELECT esimpar(@valor);
```

8.5.3 Procedimiento Muestra_Estado

En este apartado vamos a definir un procedimiento, aunque no sea su sección, para mostrar el uso de funciones dentro de este tipo de rutinas:

```
-- Ejmplo06: Procedimiento Muestra_Estado
-- Procedimiento muestra_estado donde se usa la función esimpar()

DELIMITER $$
DROP PROCEDURE IF EXISTS test.muestra_estado$$
CREATE PROCEDURE test.muestra_estado(in numero int)
BEGIN
          IF (esimpar(numero)) THEN
                    SELECT CONCAT(numero, " es impar");
          ELSE
                    SELECT CONCAT(numero, " es par");
          END IF;
END$$
```

8.5.4 Función importe total venta

Vamos a desarrollar una función algo más compleja basada en la base de datos **geslibros**.

La función debe devolver el importe total de una venta a partir de los importes parciales incluidos en la tabla lineasventas.

```
-- Ejemplo07: importeventa
-- RETURNS: el importe total de una venta a partir de los registros lineeasventa asociados
-- Parémtros: id de la venta

DELIMITER $$
DROP FUNCTION IF EXISTS geslibros.importeventa$$
CREATE FUNCTION geslibros.importeventa(id_venta INT UNSIGNED)
          RETURNS DECIMAL(10,2)
BEGIN
          DECLARE importe_venta DECIMAL(10,2);
          SET importe_venta =
             (SELECT SUM(importe) FROM geslibros.lineasventas WHERE venta_id = id_venta);

          RETURN(importe_venta);
END$$
```

Una vez desarrollada esta función sería mucho más sencillo la actualización de una venta en mi base de datos geslibros. Creamos para ello el siguiente procedimiento

```
-- Ejemplo08: actualizar_importe_ventas
-- Parémtros: id venta
-- Descripción: usando la función importeventa para actualizar la columna
--              importetotal de la tabla ventas.

DELIMITER $$
DROP PROCEDURE IF EXISTS geslibros.actualizar_importe_ventas $$
CREATE PROCEDURE geslibros.actualizar_importe_ventas(IN venta_id INT UNSIGNED)
BEGIN
          UPDATE geslibros.ventas SET importe_total = geslibros.importeventa(venta_id) WHERE
id = venta_id;
END $$
```

En definitiva las funciones permiten reducir la complejidad aparente del código encapsulándola y simplificando, por tanto, su mantenimiento y legibilidad.

En estos ejemplos ya hemos incluido instrucciones de control como *IF*. A continuación explicaremos detalles de sintaxis más importantes.

8.6 Parámetros y variables.

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

Veamos el siguiente ejemplo

```
-- Ejmplo06: Parámetros y variables
-- Creamos la tabla valores en la base de datos Test
USE TEST;
create table valores (
CodValor INT unsigned auto_increment PRIMARY KEY,
VALOR INT
);

-- Creamos el procedimiento Proc01 en la base de datos test
DELIMITER $$
DROP PROCEDURE IF EXISTS test.proc01 $$
CREATE PROCEDURE test.proc01(IN parametro01 INTEGER)
BEGIN
      DECLARE variable01 INTEGER;
      DECLARE variable02 INTEGER;
      IF parametro01 = 17 THEN
            SET variable01=parametro01;
      else
            SET variable02=30;
      END IF;
      INSERT INTO valores VALUES
      (null, variable01),
      (null, variable02);
END$$

-- Hacemos una llamada al procedimiento
DELIMITER ;
SET @valor=17;
CALL test.proc01(@valor);

-- Obviamente el ejemplo es incoherente y solo tiene propósitos didácticos
```

Encontramos dos nuevas cláusulas para el manejo de variables:

- **DECLARE**. Crea una nueva variable con su nombre y tipo. Los tipos son los usados en MySQL como CHAR, VARCHAR, INT (INTEGER), FLOAT, … Esta cláusula puede incluir una opción para indicar valores por defecto. Si no se indica, dichos valores serán NULL. Por ejemplo: *DECLARE a, b INT DEFAULT 5*, donde crea dos variables enteras con valor 5 por defecto.
- **SET.** Permite asignar valores a las variables usando el operador de igualdad. Veamos los siguientes ejemplos: *SET a = 12; SET b = 1; SET Vnombre = 'Pedro'*. Fuera del

algoritmo podemos usar también variables: *SET @apellidos='García Pérez'; CALL asignarapellido(@apellidos)*

8.6.1 Tipos de Parámetros.

Dependiendo del tipo de rutina almacenada podremos usar los siguientes tipos de parámetros:

- Procedimiento: En los procedimientos podremos usar tres tipos de parámetros entrada, salida y entrada salida (IN, OUT, INOUT).
- Funciones. En las funciones sin embargo todos los parámetros son entrada (IN).

Veamos los distintos tipos de parámetros:

- **IN**. Es el tipo por defecto y sirve para incluir parámetros de entrada que usará el procedimiento o función. En el ejemplo06 se ha declarado el parámetro de entrada *(IN) parametro01* de tipo entero *(INTEGUER*), al ser el valor por defecto no hubiese sido necesario especificar la cláusula *IN*.
- **OUT.** Permite especificar los parámetros de salida. El procedimiento puede asignar valores a dichos parámetros que son devueltos en la salida.

```
-- Ejmplo07: Parámetros de salida OUT
DELIMITER $$
DROP PROCEDURE IF EXISTS test.proc04 $$
CREATE PROCEDURE test.proc04(OUT p INT)
BEGIN
SET p = - 5 ;
END$$

-- Cambiamos el delimitador
DELIMITER ;
CALL test.proc04(@valor);
SELECT @valor;

-- Siendo el resultado -5
```

- **INOUT**. Permite especificar parámetros de entrada/salida. Pasa valores al procedimiento que serán modificados y devueltos en la llamada.

```
-- Ejmplo08: Parámetros de salida OUT
DELIMITER $$
DROP PROCEDURE IF EXISTS test.proc05 $$
CREATE PROCEDURE test.proc05(INOUT p INT)
BEGIN
SET p = p - 5 ;
END$$

-- se cambia el delimitador a ;
DELIMITER ;
SET @valor=20;
CALL test.proc05(@valor);
SELECT @valor;

-- Siendo el resultado 15
```

8.6.2 Alcance de las variables

Las variables tienen un alcance que está determinado por el bloque BEGIN/END en el que se encuentran. Es decir, no podemos ver o usar una variable que se encuentra fuera de un procedimiento salvo que le asignemos un parámetro OUT o INOUT o a una variable de sesión (usando la @).

Veamos el siguiente ejemplo:

```
-- Ejmplo09: Alcance de variables
DELIMITER $$
DROP PROCEDURE IF EXISTS test.proc06 $$
CREATE PROCEDURE test.proc06()
BEGIN
    DECLARE x1 varchar(10) default 'fuera';
    BEGIN
        DECLARE x1 varchar(10) default 'dentro';
        select x1;
    END;
    select x1;
END;$$

-- se cambia el delimitador a ;
DELIMITER ;
CALL test.proc06();
-- Siendo result1 'dentro'
-- Siendo result2 'fuera'
```

Las variables x1 del primer y segundo bloque BEGIN/END son distintas, sólo tienen validez dentro del bloque como se muestra en la llamada al procedimiento.

8.7 Instrucciones Condicionales

En muchas ocasiones el valor de una o más variables o parámetros determinará el proceso de las mismas. Cuando esto ocurre debemos usar instrucciones condicionales:

- Simple IF cuando sólo hay una condición
- IF THEN ELSE cuando hay dos alternativas
- CASE o IF THEN ELSEIF cuando hay múltiples alternativas.

8.7.1 IF-THEN-ELSE

Podemos incluir instrucciones condicionales usando:

- IF
- IF THEN ELSE
- IF THEN ELSEIF

La sintaxis general para esta construcción es:

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

Veamos los siguientes ejemplos

```
-- Ejmplo10: Construcción IF e IF-THEN
DELIMITER $$
DROP PROCEDURE IF EXISTS test.condicionales $$
CREATE PROCEDURE test.condicionales(IN par1 INT)
BEGIN
      DECLARE var1 INT;
      SET var1 = par1 +1;
      IF var1>10 THEN
            INSERT INTO valores VALUES (null, var1);
      END IF;
      IF par1=0 THEN
            UPDATE valores set valor = valor +1;
      ELSE
            UPDATE valores set valor = valor +2;
      END IF;
END$$

DELIMITER ;
SET @valor=17;
CALL condicionales(@valor);
select * from valores;
-- El resultado es un registro con valor igual a 20
```

En el siguiente ejemplo hemos creado una función que nos devuelve una categoría a partir de la edad de una persona:

```
-- Ejmplo11: Construcción IF-THEN-ELSEIF
DELIMITER $$
DROP FUNCTION IF EXISTS test.categoria $$
CREATE FUNCTION test.categoria(Edad INT)
RETURNS CHAR(3)
BEGIN
      DECLARE CATEGORIA CHAR(3);
            IF Edad <12 THEN SET CATEGORIA="INF";
            ELSEIF Edad < 18 THEN SET CATEGORIA="JUV";
            ELSEIF Edad < 30 THEN SET CATEGORIA="SNA";
            ELSEIF Edad < 40 THEN SET CATEGORIA="SNB";
            ELSEIF Edad < 50 THEN SET CATEGORIA="VTA";
            ELSE SET CATEGORIA="VTB";
    END IF;
    RETURN CATEGORIA;
END$$

DELIMITER ;
SET @Edad=28;
SELECT categoria(@Edad);

-- Devolverá la categoría SNA
```

8.7.2 CASE

Cuando hay muchas condiciones el uso de la anterior estructura IF-THEN-ELSEIF genera confusión en el código. Para estos casos es más apropiado el uso de la instrucción CASE.

Se puede usar con dos posibles sintaxis:

Evalua casos de una variable

Sintaxis

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

Ejemplo

```
-- Ejmplo12: Construcción CASE con casos según valor de una variable

DELIMITER $$
DROP PROCEDURE IF EXISTS test.estructuracase $$
CREATE PROCEDURE test.estructuracase(par1 INT)
BEGIN
         CASE par1
                  WHEN 0 THEN INSERT INTO valores VALUES   (null, par1+1);
                  WHEN 1 THEN INSERT INTO valores VALUES   (null, par1+3);
                  WHEN 2 THEN INSERT INTO valores VALUES   (null, par1+5);
                  ELSE INSERT INTO valores VALUES (null, par1);
         END CASE;
END$$

-- Hacemos uso de ese procedimiento

SET @valor=2;
CALL estructuracase(@valor);
select * from valores;
```

Evalua condiciones

Sintaxis

```
CASE

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

Ejemplo

```
-- A partir de la base de datos maratoon
-- Esta función devuelve el id de categoría que le corresponde a un corredor
-- en función de la edad

DELIMITER $$
DROP FUNCTION IF EXISTS maratoon.categoria $$
CREATE FUNCTION maratoon.categoria(Edad INT)
         RETURNS INT UNSIGNED
BEGIN
    DECLARE categoria INT UNSIGNED;
    CASE
        WHEN edad< 12 THEN SET categoria = 1;
        WHEN edad< 14 THEN SET categoria = 2;
        WHEN edad< 17 THEN SET categoria = 3;
        WHEN edad< 29 THEN SET categoria = 4;
        WHEN edad< 39 THEN SET categoria = 5;
        WHEN edad< 49 THEN SET categoria = 6;
        WHEN edad< 60 THEN SET categoria = 7;
        ELSE SET categoria = 8;
    END CASE;
    RETURN categoria;
END
```

Si nos fijamos en este ejemplo la cláusula CASE no le acompaña ninguna variable, en ese caso el WHEN lo que evalúa son condicioines, en caso que cumpla su condición se ejecuta el bloque de instrucciones que tiene asociado.

8.8 Instrucciones Repetitivas o Loops

Permiten iterar un conjunto de instrucciones un número determinado de veces. Estas instrucciones también son conocidas como bucles. Para ello MySQL provee tres tipos de instrucciones:

- Simple loop
- Repeat until
- While loop

8.8.1 SIMPLE LOOP

Implementa un constructor de bucle simple que permite la ejecución repetida de comandos.

Su sintaxis básica es la siguiente:

```
[label:] LOOP
    statement_list
END LOOP [label]
```

Donde la palabra opcional ***label*** permite etiquetar el *loop* para podernos referir a él dentro del bloque.

El siguiente ejemplo muestra un **bucle infinito** dentro de un procedimiento:

```
-- Ejmplo: Loop Infinito

DELIMITER $$
DROP PROCEDURE IF EXISTS infinite_loop$$
CREATE PROCEDURE infinite_loop()
BEGIN
        LOOP
                SELECT 'Este bucle es infinito !!!';
        END LOOP;
END$$
```

En el siguiente ejemplo etiquetamos el *loop* con el nombre *loop_label*. El loop o el bucle se ejecuta mientras no lleguemos a la condición de la línea 8. En caso de que se cumpla la orden *LEAVE* termina el *loop* etiquetado como *loop_label*.

```
-- Ejemplo: LOOP con condición de salida mediatne LEAVE

DELIMITER $$
DROP PROCEDURE IF EXISTS test.simpleloop $$
CREATE PROCEDURE test.simpleloop()
BEGIN
        DECLARE cont INT;
        SET cont=0;
        loop_label: LOOP
                INSERT INTO valores values (null, cont);
                SET cont=cont+1;
                IF cont>= 5 THEN
                        LEAVE loop_label;
                END IF;
        END LOOP;
END $$

-- Comprobación del procedimiento
-- Añadirá 5 registros a la tabla valores
-- Valores: 0 al 4.
DELIMITER ;
CALL simpleloop;
```

8.8.2 REPEAT UNTIL

Se repiten los comandos incluidos en el bucle, hasta que la condición es cierta.

Su sintaxis es la siguiente:

```
[label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [label]
```

Con el siguiente ejemplo se muestran los números impares del 0 al 10.

```
-- Ejmplo15: Repeat Until

DELIMITER $$
DROP PROCEDURE IF EXISTS test.untilloop $$
CREATE PROCEDURE test.untilloop()
BEGIN
          DECLARE i INT;
          SET i=0;
          loop1: REPEAT
                    SET i=i+1;
                    IF MOD(i,2)<>0 THEN /* número impar */
                              SELECT concat (i, " es impar");
                    END IF;
          UNTIL i>= 10
          END REPEAT;
END; $$

-- Muestra los números impares del 0 al 10
DELIMITER ;
CALL untilloop;
```

8.8.3 WHILE LOOP

Las instrucciones incluidas en el bucle se repiten mientras la condición del *while* sea cierta.

Veamos la sintaxis:

```
[label:] WHILE search_condition DO
    statement_list
END WHILE [label]
```

En el siguiente ejemplo se muestran los números pares del 0 al 10.

```
-- Ejmplo16: While

DELIMITER $$
DROP PROCEDURE IF EXISTS test.whileloop $$
CREATE PROCEDURE test.whileloop()
BEGIN
          DECLARE i INT;
          SET i=1;
          WHILE i<=10 DO
                    IF MOD(i,2)=0 THEN /* número par */
                              SELECT concat (i, " es par");
                    END IF;
                    SET i=i+1;
          END WHILE;
END$$

-- Muestra los números pares del 0 al 10
DELIMITER ;
CALL whileloop;
```

8.9 SQL en Rutinas: CURSORES

Hasta ahora la mayoría de los ejemplos vistos contenían instrucciones o expresiones referidas a cálculos matemáticos o de cadenas sencillas sin implicar el uso de datos de una base de datos. Normalmente sin embargo el uso de procedimientos implica manipular datos de tablas

de bases de datos lo que implica usar instrucciones SQL. En esta sección veremos ejemplos diversos de procedimientos que acceden a bases de datos haciendo uso de las instrucciones explicadas en apartados anteriores.

En general podemos usar cualquier instrucción de SQL, tanto perteneciente al DDL, DML o DCL.

Ejemplo

Veamos el siguiente ejemplo en el que usamos sentencias SQL de definición (DROP y CREATE) y sentencias SQL de manipulación (INSERT, UPDATE y DELETE). El ejemplo incluye comentarios que lo explican.

```
-- Ejemplo17: SQL en Rutinas
DELIMITER $$
DROP PROCEDURE IF EXISTS simple_sql $$
CREATE PROCEDURE simple_sql()
BEGIN
          DECLARE i INT DEFAULT 1;
          DROP TABLE IF EXISTS test_table;
          CREATE TABLE IF NOT EXISTS test_table (
                    id INT AUTO_INCREMENT PRIMARY KEY,
                    Nombre VARCHAR(30)
          );
          -- USO INSERT
          WHILE (i<10) DO
                    INSERT INTO test_table VALUES (NULL, CONCAT('Registro-', i));
                    SET i=i+1;
          END WHILE;
          -- USO UPDATE
          SET i=5;
          UPDATE test_table SET Nombre=CONCAT('Actualizando-', i) WHERE id=i;
          -- USO DELETE
          SET i=7;
          DELETE FROM test_table WHERE id=i;
          -- USO SELECT
          SELECT * FROM test_table;
END;$$

DELIMITER ;
CALL simple_sql();
```

Ejemplo

En el siguiente ejemplo usamos la propiedad de las sentencias SELECT de enviar valores a variables usando la cláusula ***INTO***.

```
-- Ejemplo18: SELECT … INTO variables
-- Uso Base de Datos Maratoon
DELIMITER $$
DROP PROCEDURE IF EXISTS maratoon.ObtenerCorredor $$
CREATE PROCEDURE maratoon.ObtenerCorredor(PIdCorredor INT)
BEGIN
DECLARE Vid INT UNSIGNED;
DECLARE VNombre VARCHAR(20);
DECLARE VApellidos VARCHAR(45);
DECLARE VEdad TINYINT UNSIGNED;
DECLARE VCategoria CHAR(3);
DECLARE VClub VARCHAR(30);
SELECT c.Id, c.Nombre, c.Apellidos, c.Edad, ca.Nombrecorto, cl.Nombre INTO Vid, VNombre,
Vapellidos, VEdad, VCategoria, VClub FROM corredor AS c JOIN categoria AS ca
USING(CodCategoria) JOIN club AS cl USING(CodClub) WHERE id=PIdCorredor;

SELECT Vid, VNombre, Vapellidos, VEdad, VCategoria, VClub;
END$$

DELIMITER ;
SET @NumCorredor=3;
CALL maratoon.ObtenerCorredor(@NumCorredor);
```

En el anterior ejemplo observamos que la sentencia SELECT asigna los valores de la fila seleccionada para asignarlos a su vez a nuevas variables internas del procedimiento.

CURSOR

No obstante muchas veces querremos recuperar más de una fila para manipular sus datos, en estos casos no sirve la construcción anterior y requerimos el uso de **CURSORES**. Conceptualmente un cursor se asocia con un conjunto de filas y columnas sobre una o más tabla de una base de datos.

Un **CURSOR** se crea usando la siguiente sintaxis:

```
DECLARE cursor_name CURSOR FOR sentencia_select
```

Ejemplo

Veamos el siguiente ejemplo de declaración de un CURSOR

```
-- Ejemplo: Declaración CURSOR

DECLARE cursor01 CURSOR FOR SELECT Id, Nombre, Apellidos FOR corredores;
```

Y debe hacerse después de declarar todas las variables necesarias para el procedimiento.

8.9.1 Comandos relacionados con los cursores

Para manipular los cursores disponemos de una serie de comandos:

- ***OPEN***: inicializa el conjunto de resultados asociados con el cursor.
  OPEN cursor_name;
- ***FETCH***: extrae la siguiente fila de valores del conjunto de resultados del cursor moviendo su puntero interno una posición.
  FETCH cursor_name INTO variables_list;
- ***CLOSE****:* cierra el cursor liberando la memoria que ocupa y haciendo imposible el acceso a cualquiera de sus datos.
  CLOSE cursor_name;

Veamos el siguiente ejemplo donde declaramos un cursor que extraiga de la tabla Corredores de la Base de Datos Maratoon, todos los corredores que sean de Ubrique.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS corredores_poblacion $$
CREATE PROCEDURE Corredores_Poblacion(PCiudad VARCHAR(20))
BEGIN
    DECLARE Vid INT UNSIGNED;
    DECLARE VNombre VARCHAR(20);
    DECLARE VApellidos VARCHAR(40);
    DECLARE VCiudad VARCHAR(20);
    DECLARE Cursor01 CURSOR FOR SELECT id, Nombre, Apellidos, Ciudad FROM Corredores WHERE
Ciudad=PCiudad;
    OPEN Cursor01;
         l_cursor: LOOP
                   FETCH Cursor01 INTO Vid, VNombre, VApellidos, VCiudad;
    END LOOP l_cursor;
    CLOSE Cursor01;
END $$
--
DELIMITER ;
CALL Corredores_Poblacion('Ubrique');
```

Este procedimiento como vemos da **error**, dado que cuando llegamos a la última fila no hay más datos que obtener así que necesitamos de algún modo detectar ese momento. Para ello usaremos un manejador de errores o *handler* necesitando para ello la siguiente instrucción:

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

Que hace dos cosas:

- Establecer la variable *ultima_fila=1*.
- Permitir al programa continuar su ejecución.

Veamos el ejemplo con **LOOP**

```
-- Cursores con estructura LOOP

DELIMITER $$
DROP PROCEDURE IF EXISTS Corredores_Poblacion$$
CREATE procedure Corredores_Poblacion(PCiudad VARCHAR(20))
BEGIN
    DECLARE Vid INT;
    DECLARE VNombre VARCHAR(20);
    DECLARE VApellidos VARCHAR(40);
    DECLARE VCiudad VARCHAR(20);
    DECLARE lrf BOOLEAN;
    DECLARE nreg INT;
    DECLARE Cursor01 CURSOR FOR SELECT id, Nombre, Apellidos, Ciudad FROM Corredores WHERE
Ciudad=PCiudad;
    DECLARE CONTINUE HANDLER  FOR NOT FOUND SET lrf=1;
    SET lrf=0, nreg=0;
    OPEN Cursor01;
    l_cursor: LOOP
        FETCH Cursor01 INTO Vid, VNombre, VApellidos, VCiudad;
        SELECT Vid, VNombre, VApellidos, VCiudad;
        IF lrf=1 THEN
                    LEAVE l_cursor;
        END IF;
        SET nreg=nreg+1;
    END LOOP l_cursor;
    CLOSE Cursor01;
    SELECT nreg;
END $$
```

En este caso hemos declarado dos nuevas variables: *lrf* (*last row fetched* o última fila extraída), que es una variable booleana con posibles 0 o 1 indicando si hemos llegado o no al final del cursor, y por otra parte la variable *nreg* que mostrará el número de corredores de esa población. Gracias a la sentencia *LEAVE* podremos terminar el bucle cuando *lrf* adquiere el valor 1, o lo que es lo mismo se alcanza el final del cursor.

Veamos el mismo ejemplo anterior con ***REPEAT UNTIL***

```
-- Cursores con estructura REPEAT UNTIL

DELIMITER $$
DROP PROCEDURE IF EXISTS Corredores_Poblacion$$
CREATE procedure Corredores_Poblacion(PCiudad VARCHAR(20))
BEGIN
         DECLARE Vid INT;
    DECLARE VNombre VARCHAR(20);
    DECLARE VApellidos VARCHAR(40);
    DECLARE VCiudad VARCHAR(20);
    DECLARE lrf BOOLEAN;
    DECLARE nreg INT;
    DECLARE Cursor01 CURSOR FOR SELECT id, Nombre, Apellidos, Ciudad FROM Corredores
    WHERE Ciudad=PCiudad;
    DECLARE CONTINUE HANDLER  FOR NOT FOUND SET lrf=1;
    SET lrf=0, nreg=0;
    OPEN Cursor01;
    REPEAT
                    FETCH Cursor01 INTO Vid, VNombre, VApellidos, VCiudad;
                    SELECT 'Registro: ', nreg, Vid, VNombre, VApellidos, VCiudad;
                    SET nreg=nreg+1;
```

```
    UNTIL lrf
    END REPEAT;
    CLOSE Cursor01;
    SELECT nreg;
END $$
```

Cursor con ***while***

```
DELIMITER $$
DROP PROCEDURE IF EXISTS Corredores_Poblacion$$
CREATE procedure Corredores_Poblacion(PCiudad VARCHAR(20))
BEGIN
        DECLARE Vid INT;
    DECLARE VNombre VARCHAR(20);
    DECLARE VApellidos VARCHAR(40);
    DECLARE VCiudad VARCHAR(20);
    DECLARE lrf BOOLEAN;
    DECLARE nreg INT;
    DECLARE Cursor01 CURSOR FOR SELECT id, Nombre, Apellidos, Ciudad FROM Corredores
    WHERE Ciudad=PCiudad;
    DECLARE CONTINUE HANDLER  FOR NOT FOUND SET lrf=1;
    SET lrf=0, nreg=0;
    OPEN Cursor01;
    WHILE (NOT lrf) DO
        FETCH Cursor01 INTO Vid, VNombre, VApellidos, VCiudad;
        SELECT 'Registro: ', nreg, Vid, VNombre, VApellidos, VCiudad;
        SET nreg=nreg+1;
    END WHILE;
    CLOSE Cursor01;
    SELECT nreg;
END $$
```

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

8.10 Gestión de Rutinas Almacenadas

Las rutinas se manipulan con los comandos CREATE, DROP y SHOW.

8.10.1 Eliminación de rutinas

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

```
DROP [PROCEDURE | FUNCTION] [IF EXISTS] sp_name
```

8.10.2 Consulta de rutinas

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

```
SHOW CREATE [PROCEDURE | FUNCTION] sp_name
SHOW [PROCEDURE | FUNCTION] STATUS [LIKE 'patron']
```

En el segundo caso podemos hacer un filtro de patrones

Estos comandos y en general todos los de tipo *SHOW*, se nutren del diccionario de datos gracias a la tabla *INFORMATION_SCHEMA.ROUTINES* que también podemos consultar con la sentencia *SELECT*.

8.11 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. Ese es el comportamiento predeterminado. ¿Qué pasa si necesitamos otro tipo de comportamiento? ¿Qué pasa si por ejemplo, queremos que la trampa de error, log, o informar al respecto y luego continuar con la ejecución de

nuestra aplicación? Para este tipo de control tenemos que definir controladores de excepciones en nuestros programas.

Por ejemplo si queremos insertar un corredor con una clave existente devolvería un mensaje de error parecido al siguiente

```
ERROR 1062 (23000): Duplicate entry 'IdCorredor' for key 1
```

Indica la existencia de clave repetida en el campo IdCorredor. En genera los errores deben ser prevenidos y tratados o manejados. Un procedimiento con manejo de dicho error sería de la siguiente forma:

```
-- Ejemplo: Manejo de errores

DELIMITER $$
DROP PROCEDURE IF EXISTS maratoon.Insertar_Corredor$$
CREATE procedure maratoon.Insertar_Corredor(PNCorredor INT, PNombre VARCHAR(20),
PApellidos VARCHAR(45), PCiudad VARCHAR(30), PFecNac DATE, OUT PEstado VARCHAR(45))
MODIFIES SQL DATA
BEGIN
    DECLARE CONTINUE HANDLER FOR 1062 SET PEstado='Duplicate Entry';
    SET PEstado='OK';
    INSERT INTO maratoon.corredores(NCorredor, Nombre, Apellidos, Ciudad, FechaNacimiento)
    VALUES (PNCorredor, PNombre, PApellidos, PCiudad, PFecNac);
END; $$
```

Pero si queremos hacer algo con el error debemos usar la variable *out_status*. En el siguiente ejemplo llamamos al procedimiento dentro de otro procedimiento, condicionando la salida al valor de la variable @estado tipo *out*

```
-- Ejemplo: Manejo de errores
DELIMITER ;
set @Estado=null;
CALL Insertar_Corredor (10, 'Carlos', 'García Bragado', 'Arcos', '1967-07-20', @Estado);
select @Estado;
```

8.11.1 Sintaxis de manejador

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
        {SQLSTATE sqlstate_code | MySQL error code | condition_name}
        Handler_action
```

- Tipo manejador: *EXIT o CONTINUE*
- Condición del manejador: estado *SQL (SQLSTATE)*, error propio *MySQL* o código de error definido por el usuario.
- Acciones del manejador: acciones a tomar cuando se active el manejador

8.11.2 Tipo de manejador

- EXIT. Cuando se encuentra un error el bloque que se está ejecutando actualmente se termina. Si este 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 CONTINUE la ejecución continúa en la declaración siguiente a la que ocasionó el error.

Veamos ahora ejemplos de ambos tipos de controladores. En el siguiente ejemplo el procedimiento crea un registro de corredor. Para manejar la posibilidad de que el autor ya

exista se crea el manejador de tipo EXIT que en caso de activarse establecerá el valor de la variable *duplícate_key* a 1 y devolverá el control al bloque *BEGIN/END* exterior.

```
-- Ejemplo: HANDLER de tipo EXIT

DELIMITER $$
USE maratoon$$
DROP PROCEDURE IF EXISTS maratoon.Insertar_Corredor$$
CREATE procedure maratoon.Insertar_Corredor(PNCorredor INT, PNombre VARCHAR(20),
PApellidos VARCHAR(45), PCiudad VARCHAR(30), PFecNac DATE)
MODIFIES SQL DATA
BEGIN
    DECLARE duplicate_key INT DEFAULT 0;
    BEGIN
        DECLARE EXIT HANDLER FOR 1062 SET duplicate_key=1;
        INSERT INTO maratoon.corredores(NCorredor, Nombre, Apellidos, Ciudad,
FechaNacimiento)
        VALUES (PNCorredor, PNombre, PApellidos, PCiudad, PFecNac);
    END;
    IF duplicate_key=1 THEN
        SELECT 'ERROR INSERCION: Clave Duplicada' AS Resultado;
    ELSE
        SELECT CONCAT('Corredor ', PNCorredor, PNombre, 'Creado') AS Resultado;
    END IF;
END; $$

--
DELIMITER ;
CALL Insertar_Corredor (10, 'Carlos', 'García Bragado', 'Arcos', '1967-07-20');
```

Un controlador de tipo EXIT es más adecuado para los errores catastróficos ya que no permite ninguna forma de continuación de la tramitación.

Un controlador CONTINUE es más adecuado cuando se tiene algún procesamiento alternativo que se ejecutará si la excepción se produce.

Un desencadenador de manejador define las circunstancias que activa un manejador. Pueden ser por un error de código, un error de SQL o por una circunstancia definida por el usuario. Por defecto al indicar un error numérico nos referimos a un error de SQL.

8.12 TRIGGERS

Un trigger o **disparador** es un tipo especial de rutina almacenada que se activa o ejecuta cuando en una tabla ocurre un evento de tipo: INSERT, DELETE o UPDATE. Los disparadores implementan una funcionalidad asociada a cualquier cambio en una tabla.

8.12.1 Gestión de los Disparadores

Las instrucciones para gestionar los disparadores son:

- *CREATE TRIGGER*
- *SHOW TRIGGER*
- *DROP TRIGGER*

8.12.1.1 Crear trigger. CREATE TRIGGER

La sintaxis de dicho comando es la siguiente

```
CREATE TRIGGER  nonmbre_disp momento_disp evento_disp
ON nombre_tabla FOR EACH ROW sentencia_disp
```

Donde:

- Nombre_disp: Nombre del disparador lo más descriptivo posible.
- Momento_disp: momento en el que el disparador se activa: BEFORE, AFTER.
- Evento_disp: clase de sentencia que activa el disparador: INSERT, UPDATE, DELETE. No puede haber dos disparadores en una misma tabla que se activen en el mismo momento y sentencia
- Nombre_tabla: Nombre de la tabla asociada al disparador
- FOR EACH ROW: acciones a llevar a cabo sobre cada fila de la tabla
- Setencia_dis. Sentencia que se ejecuta cuando se activa el disparador. Si se desean ejecutar múltiples sentencias deben colocarse entre BEGIN … END. También pueden ejecutarse procedimientos almacenados en la Base de Datos

Las columnas de las tablas asociadas pueden referenciarse empleando los prefijos OLD y NEW:

- OLD.nom_col: valor de la columna antes de ser modificada o borrada.
- NEW.nom_col: valor de la columna antes de ser insertada o valor de la columna después de ser modificada.

Las palabras claves OLD y NEW permiten acceder a columnas en los registros afectadas por un disparador.

- INSERT. Sólo puede usar NEW.nom_col ya que no hay una versión anterior del registro
- DELETE. Sólo puede usar OLD.nom_col ya que no hay una versión nueva.
- UPDATE. Se puede usar OLD.nom_col valor antes de la actualización NEW.nom_col valor ya actualizado

Veamos el siguiente ejemplo:

Ejemplo 1. TRIGGERS asociado a INSERT

En el siguiente ejemplo, sobre la base de datos BANCOS, se crea el TRIGGERS llamado **actualizar_saldo** que permita actualizar la columna saldo de la tabla cuentas, después (**AFTER**) de insertar (**INSERT**) un nuevo registro en la tabla **movimientos**.

```
-- Trigger: actualizar_saldo
-- Momento: AFTER (después de)
-- Evento: INSERT
-- Tabla: movimientos
-- Descripción: Actualiza la columna saldo de la tabla cuentas, cada vez que se inserte
-- un registro en la tabla Movimientos.

DELIMITER $$
DROP TRIGGER IF EXISTS bancos.actualizar_saldo$$
CREATE TRIGGER bancos.actualizar_saldo AFTER INSERT ON bancos.movimientos
FOR EACH ROW
BEGIN

         UPDATE bancos.cuentas SET saldo = saldo + NEW.cantidad WHERE id=NEW.cuenta_id;

END$$
```

Ahora si insertamos un registro en la tabla movimientos:

```
INSERT movimientos VALUE (NULL, 1, NULL, 'Transferencia','R', -50.10)

-- Al estar asociado a un evento de tipo INSERT solo dispongo de los prefijos NEW.
-- NEW.cantidad = -50.10
-- NEW.cuenta_id = 1
-- Por lo tanto el UPDATE asociado al TRIGGERS sería
-- UPDATE bancos.cuentas SET saldo = saldo – 50.10 WHERE id=1;
```

```
INSERT movimientos VALUE (NULL, 1, NULL, 'Ingreso','I', 100.00)

-- Si inserto el registro anterior entonces
-- NEW.cantidad = 100.00
-- NEW.cuenta_id = 1
-- UPDATE bancos.cuentas SET saldo = saldo + 100.00 WHERE id=1;
```

Ejemplo 2. TRIGGERS asociado a UPDATE

En esta ocasión vamos a crear un TRIGGERS asociado a un evento de tipo UPDATE.

Supongamos que es posible actualizar un movimiento, debido a que la cantidad anotada ha sido errónea, en este caso una vez actualizado dicho movimiento se ha de actualizar además la columna saldo de la tabla cuentas, al igual que ocurría cuando insertámos un nuevo movimiento. Solamente podremos modificar la columna cantidad, así que el resto de columnas no se permite su modificación.

```
-- Trigger: act_mov_act_saldo
-- Momento: AFTER (después de)
-- Evento: UPDATE
-- Tabla: movimientos
-- Descripción: Actualiza la columna saldo de la tabla cuentas, cada vez que se ACTUALICE
-- un registro en la tabla Movimientos.

DELIMITER $$
DROP TRIGGER IF EXISTS bancos.act_mov_act_saldo$$
CREATE TRIGGER bancos.act_mov_act_saldo AFTER UPDATE ON bancos.movimientos
FOR EACH ROW
BEGIN
          DECLARE act_saldo DECIMAL(10,2);
          SET act_saldo =  NEW.cantidad - OLD.cantidad;

          UPDATE bancos.cuentas SET saldo = saldo + act_saldo WHERE id=OLD.cuenta_id;

END$$

-- Ejemplos
UPDATE movimientos SET cantidad = 100 WHERE id = 24;
UPDATE movimientos SET cantidad = -20.10 WHERE id = 25;
```

Ejemplo 3. TRIGGERS asociado a DELETE

En una base de datos relacional hay que tener mucho cuidado con el comando DELETE. En la base de datos **bancos.sql** se establece que cuando se elimine un movimiento, es decir, un registro en la tabla movimientos, se ha de actualizar la columna saldo de la tabla cuentas**.**

```
-- Trigger: delete_mov_act_saldo
-- Momento: AFTER (después de)
-- Evento: DELETE
-- Tabla: movimientos
-- Descripción: Actualiza la columna saldo de la tabla cuentas, cada vez que se elimine
-- un registro en la tabla Movimientos.

DELIMITER $$
DROP TRIGGER IF EXISTS bancos.delete_mov_act_saldo$$
CREATE TRIGGER bancos.delete_mov_act_saldo AFTER DELETE ON bancos.movimientos
FOR EACH ROW
BEGIN
          UPDATE bancos.cuentas SET saldo = saldo - OLD.cantidad WHERE id = OLD.cuenta_id;
END$$

-- Ejemplos
DELETE FROM movimientos WHERE id=10;
```

8.12.1.2 Eliminar Triggers. DROP TRIGGER.

Para el eliminar el disparador se emplea una sentencia *DROP TRIGGER*. El nombre del disparador debe incluir el nombre de la base de datos.

```
DROP TRIGGER  [IF EXISTS] [db_name.]nonmbre_disp
```

8.12.1.3 Consultar Triggers. SHOW TRIGGER.

Podemos obtener información de los *Trigger* creados con *SHOW TRIGGER*

```
SHOW TRIGGER  [[FROM | IN] db_name] [LIKE 'pattern' | WHERE expr]
```

Este comando nos permite mostrar trigger de una base de datos filtrándolo con un patrón o cláusula *WHERE*.

Cuando creamos un *triggers* se crea un nuevo registro en la tabla *INFORMATION_SCHEMA* llamada *INFORMATION_SCHEMA.TRIGGERS*, que podemos mostrar con el siguiente comando:

```
SELECT * FROM Information_schema.triggers
```

8.12.2 Uso de los disparadores

Aunque el uso de los disparadores es muy variado y depende mucho de la aplicación o base de datos con que trabajemos podemos hacer una clasificación más o menos general.

8.12.2.1 Control de Sesiones.

En ocasiones puede ser interesante recoger ciertos valores en variables de sesión creadas por el usuario que nos permitan ver un resumen de lo realizado en dicha sesión.

En el ejemplo acumulamos la cantidad del movimiento a insertar en la variable de sesión @sum.

```
USE test;
DROP TABLE IF EXISTS movimientos;
CREATE TABLE IF NOT EXISTS movimientos (
    num INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    cantidad DECIMAL(8 , 2 )
);

-- Creamos el disparador
DROP TRIGGER IF EXISTS test.insertar_mov;
CREATE TRIGGER test.insertar_mov BEFORE INSERT ON movimientos
FOR EACH ROW SET @SUM = @SUM + NEW.Cantidad;

-- Comprobamos que el disparador se ha creado
SELECT * FROM INFORMATION_SCHEMA.TRIGGERS;

-- Uso disparador
SET @SUM=0;
INSERT INTO movimientos VALUES
(NULL, 100.20),
(NULL, 200.20),
(NULL, 300.20),
(NULL, 400.20);

-- Mostramos el valor total ingresado en la sesión
SELECT @SUM AS 'Total Ingresado';

-- Resultado
-- 1000.80
```

8.12.2.2 Validación de datos de entrada

Supongamos que en la base de datos bancos se establece la restricción de que un usuario no puede sacar más de 900 € en un solo reintegro (R)

```
-- Trigger: validar_reintegro
-- Momento: BEFORE
-- Evento: INSERT
-- Tabla: movimientos
-- Descripción: Valida la cantidad de un movimiento en caso de ser un reintegro R ya que no
-- está permitido sacar más de 900 €

DELIMITER $$
DROP TRIGGER IF EXISTS bancos.validar_reintegro$$
CREATE TRIGGER bancos.validar_reintegro BEFORE INSERT ON bancos.movimientos
FOR EACH ROW
BEGIN
         IF NEW.tipo = 'R' Then
                  IF NEW.cantidad < - 900.00 THEN
                           SET NEW.cantidad = - 900;
                  END IF;
         END IF;
END$$
DELIMITER ;

-- Ejemplo
INSERT INTO movimientos VALUES
(null, 1, DEFAULT, 'Apertura', 'R', -912.00);
```

8.12.2.3 Registro y auditorías

Cuando muchos usuarios acceden a la base de datos, puede ser que el registro de log no sea suficiente.

Podemos asignar un triggers a una tabla que se dispare después de (AFTER) de una sentencia DELETE o UPDATE, que guarde los valores del registro, así como información de utilidad en una tabla log.

En nuestro ejemplo queremos saber quién y a que hora modificó la tabla movimientos. Para ello crearemos un triggers que registre dichas actualizaciones incluyendo los datos antiguos y los nuevos, por cada registro modificado.

Para ello vamos a usar la base de datos Bancos.sql del apartamdo 8.3.1 a la que le vamos a insertar una nueva tabla llamada auditoría.

```
-- Triggers Registro Auditorias
USE bancos;
-- Tabla auditoria
DROP TABLE IF EXISTS auditoria_movimientos;
CREATE TABLE IF NOT EXISTS auditoria_movimientos(
 id INT AUTO_INCREMENT PRIMARY KEY,
 evento VARCHAR(15),
 cuenta_id_ant INT UNSIGNED,
 fechahora_ant TIMESTAMP,
 tipo_ant CHAR(1),
 cantidad_ant DECIMAL(8, 2 ),
 cuenta_id_nuv INT UNSIGNED,
 fechahora_nuv TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 tipo_nuv CHAR(1),
 cantidad_nuv DECIMAL(10, 2 ),
 usuario VARCHAR(40),
 fecharegsitro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Una vez creada la tabla anterior creamos el siguiente TRIGGERS

```
-- Creamos el disparador que inserte un registro en auditoria
delimiter $$
DROP TRIGGER IF EXISTS auditoria_mov$$
CREATE TRIGGER auditoria_mov AFTER UPDATE ON movimientos
FOR EACH ROW
BEGIN
    INSERT INTO auditoria VALUES
    (NULL, 'UPDATE', OLD.cuenta_id, OLD.fechahora, OLD.tipo, OLD.cantidad,
     NEW.cuenta_id, NEW.fechahora, NEW.tipo, NEW.cantidad, CURRENT_USER(), NOW());

END;$$

-- Uso disparador
DELIMITER ;
UPDATE movimientos SET
cantidad = 200 WHERE num=2;
-- Se ha añadido el registro correspondiente en la tabla
-- AUDITORIA.
```

8.13 Eventos

En MySQL los eventos son **tareas que se ejecutan de acuerdo a un horario**. Por lo tanto, a veces nos referimos a ellos como los eventos programados. Conceptualmente, esto es similar a la idea del programa *Crontab* de Linux o el programador de tareas de Windows. También se conocen como ***triggers* temporales** ya que conceptualmente son similares diferenciándose en que el *trigger* se activa por una acción sobre la base de datos mientras que el evento se activa a partir de una marca de tiempo.

Un **evento** se identifica por su **nombre** y el **esquema** o base de datos al que se le asigna. Lleva a cabo una acción específica de acuerdo a un horario. Esta acción consiste en una o varias instrucciones SQL dentro de un bloque ***BEGIN/END***.

Distinguimos dos tipos de eventos, los que se programan para una **única ocasión** y los que ocurren **periódicamente** cada cierto tiempo.

La variable ***global event_scheduler*** determina si el programador de eventos está habilitado y en ejecución en el servidor. Esta variable puede tomar los valores ***ON*** para activarlo, ***OFF*** para desactivarlo y ***DISABLED*** si queremos imposibilitar la activación.

Cuando el Programador de eventos (*Scheduler*) se detiene (variable *global_event_scheduler* está *OFF*), puede ser iniciado por establecer el valor de *global_event_scheduler* en *ON*.

```
-- Activación del programador de eventos
SET GLOBAL event_scheduler=ON;
SHOW variables like 'event_scheduler';
```

Los comandos para la gestión de eventos son:

- *CREATE EVENT*
- *ALTER EVENT*
- *SHOW EVENT*
- *DROP EVENT*

8.13.1.1 Creación de Eventos

Un evento se define mediante la instrucción ***CREATE EVENT***:

```
CREATE [DEFINER = { user | CURRENT_USER }] EVENT [IF NOT EXISTS] event_name

    ON SCHEDULE schedule
    [ON COMPLETION [NOT] PRESERVE]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'comment']
    DO event_body;

schedule:
    AT timestamp [+ INTERVAL interval] ...
  | EVERY interval
    [STARTS timestamp [+ INTERVAL interval] ...]
    [ENDS timestamp [+ INTERVAL interval] ...]

interval:
    quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
              WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
              DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

Donde se crea un evento con un nombre asociado a una base de datos o esquema determinado.

En esta instrucción distinguimos las siguientes cláusulas:

- *ON SCHEDULE*: permite establecer cómo y cuando se ejecutará el evento. Una sola vez, durante un intervalo, cada cierto tiempo o en una fecha hora de inicio y fin determinadas.
- *DEFINER*: especifica el usuario cuyos permisos se tendrán en cuenta en la ejecución del evento.
- *Event_body*: es el contenido o código del evento que se va a ejecutar
- *COMPLETION*: permite mantener el evento aunque haya expirado mientras que *DISABLE* permite crear el evento en estado inactivo.
- *DISABLE ON SLAVE*: sirve para indicar que el evento se creó en el master de una replicación y que, por tanto, no se ejecutará en el esclavo.

Ejemplo 1.

Sobre la base de *datos* bancos se realiza una exportación diaria de la tabla clientes en formato CSV hacia la unidad F: del sistema.

```
-- Ejemplo de Evento

DELIMITER $$
DROP EVENT IF EXISTS bancos.ObtenerClientes $$
CREATE EVENT IF NOT EXISTS bancos.ObtenerClientes ON SCHEDULE EVERY 1 DAY STARTS '2020-05-16
00:00:00' ENABLE
DO
BEGIN
          SELECT
                    *
          FROM
                    bancos.clientes INTO OUTFILE 'f:clientes.csv' FIELDS TERMINATED BY ';'
                    OPTIONALLY ENCLOSED BY '\"' LINES TERMINATED BY '\n';
END$$

-- Para mostrar los eventos
DELIMITER ;
SHOW EVENTS IN bancos;
SELECT * FROM INFORMATION_SCHEMA.EVENTS;
```

Ejemplo 2.

En este ejemplo programaremos un evento en MySQL para que al cabo de una hora se verifique el saldo de las cuentas de la base de datos bancos.

Para ello usaremos el procedmiento **actualizar_saldo** realizado en la practica 8.4

```
-- Procedimiento actualizar_saldo - Práctica 8.4

CREATE DEFINER=`root`@`localhost` PROCEDURE `actualizar_saldo`()
BEGIN

    DECLARE v_id INT UNSIGNED;
    DECLARE v_iban CHAR(24);
    DECLARE v_saldo, f_saldo DECIMAL(10,2);

    DECLARE lrf BOOLEAN DEFAULT FALSE;

    DECLARE cuentas CURSOR FOR SELECT id, iban, saldo FROM Cuentas ORDER BY id;

    DECLARE CONTINUE HANDLER  FOR NOT FOUND SET lrf=1;

    OPEN cuentas;
    FETCH cuentas INTO v_id, v_iban, v_saldo;
         WHILE (NOT lrf) DO
        SET f_saldo = bancos.saldo_cuenta(v_id);
        IF v_saldo <> f_saldo THEN
                            UPDATE cuentas SET saldo = f_saldo WHERE id = v_id;
            SELECT 'OJO DESCUADRE: 'v_id, v_iban, v_saldo, f_saldo;
                    END IF;
        FETCH cuentas INTO v_id, v_iban, v_saldo;
    END WHILE;
    CLOSE cuentas;
END
```

Que como vemos también usa la función **saldo_cuenta()** realizada en la practica 8.3

```
-- Función saldu_cuenta() en la Práctica 8.2

DROP FUNCTION IF EXISTS bancos.saldo_cuenta $$
CREATE FUNCTION bancos.saldo_cuenta(id_cuenta INT UNSIGNED)
         RETURNS DECIMAL(10,2)
BEGIN
         DECLARE vsaldo DECIMAL(10,2);
    SET vsaldo =
    (SELECT
                    SUM(cantidad)
                    FROM
                    movimientos
                    WHERE
                    cuenta_id = id_cuenta
         );
    RETURN vsaldo;
END$$
```

El evento para que se ejecute **dentro de una hora**

```
-- Evento que actuliza los saldos dentro de una hora

DELIMITER $$
DROP EVENT IF EXISTS bancos.verificarSaldos $$
CREATE EVENT IF NOT EXISTS bancos.verificarSaldos
ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 HOUR ENABLE
DO CALL actualizar_saldo;
```

Evento que se ejecute cada semana

```
-- Cada semana a partir del 2020-05-16
```

```
DELIMITER $$
DROP EVENT IF EXISTS bancos.verificarSaldos $$
CREATE EVENT IF NOT EXISTS bancos.verificarSaldos
ON SCHEDULE EVERY 1 WEEK STARTS '2020-05-16 00:00:00' ENABLE
DO CALL actualizar_saldo;
```

Evento que se ejecute cada semana durante 12 meses

```
-- Cada semana durante 12 meses

DELIMITER $$
DROP EVENT IF EXISTS bancos.verificarSaldos $$
CREATE EVENT IF NOT EXISTS bancos.verificarSaldos
ON SCHEDULE EVERY 1 WEEK STARTS '2020-05-16 00:00:00'
ENDS '2020-05-16 00:00:00' + INTERVAL 1 YEAR ENABLE
DO CALL actualizar_saldo;
```

8.13.1.2 Modificar un evento

Para modificar un evento usamos la orden ALTER EVENT con la siguiente sintaxis:

```
ALTER

    [DEFINER = { user | CURRENT_USER }]
    EVENT event_name
    [ON SCHEDULE schedule]
    [ON COMPLETION [NOT] PRESERVE]
    [RENAME TO new_event_name]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'comment']
    [DO event_body]
```

Las cláusulas son similares a las de CRAEATE EVENT

8.13.1.3 Consulta de eventos

Para la información asociada a un evento usamos SHOW EVENT:

```
SHOW EVENTS [{FROM | IN} schema_name]
    [LIKE 'pattern' | WHERE expr]
```

Que muestra información de eventos asociados a un esquema o bases de datos y filtrado según el patrón que determinemos o una cláusula WHERE.

Como siempre podemos recurrir al diccionario de datos consultado la tabla ***INFORMATION_SCHEMA.EVENTS***