UNIDAD DIDÁCTICA 14 PROCEDIMIENTOS ALMACENADOS

Desde MySQL 5.0.0 se pueden definir **procedimientos** almacenados en el servidor. Un **procedimiento** almacenado es un conjunto de comandos SQL que pueden almacenarse en el servidor. 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. El intercambio que hay es que aumenta 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. Considere esto sí muchas máquinas cliente (como servidores Web) se sirven a sólo uno o pocos servidores de bases de datos.

Las **funciones** almacenadas devuelven un resultado que se puede incluir en expresiones como si fuese una función incorporada.

Los **procedimientos** almacenados no devuelven un resultado directamente, pero soportan tipos de parámetros que pueden tener sus conjuntos de valores en el cuerpo del procedimiento, de forma que el procedimiento de llamada pueda acceder a dichos valores después de que el procedimiento finalice.

Normalmente, se utiliza una función para calcular un valor que necesitamos devolver al procedimiento de llamada para utilizarlo en alguna expresión. Se utiliza un procedimiento si necesitamos llamar a una rutina para producir un efecto sin devolver ningún valor.

Ventajas de las rutinas almacenadas:

- Sintaxis SQL extendida para incluir bucles y ramificaciones.
- Proporcionan un mecanismo de gestión de errores.
- Como se almacenan en el servidor, el código necesario para definirlos solo se envía una vez, cuando se crea y no cada vez que lo utilizamos.

- Proporcionan un medio para estandarizar código creando librerías que pueden ser utilizadas por distintas aplicaciones.

PROCEDIMIENTOS ALMACENADOS Y LAS TABLAS DE PERMISOS

Los procedimientos almacenados necesitan la tabla **proc** en la base de datos **mysql**. Esta tabla se crea durante la instalación de MySQL..

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.

Sintaxis de procedimientos almacenados

Los procedimientos almacenados y rutinas se crean con los comandos **CREATE PROCEDURE** y **CREATE FUNCTION**. Una rutina es un procedimiento o una función. Un procedimiento se invoca usando un comando **CALL**, y sólo puede pasar valores usando variables de salida. Una función puede llamarse desde dentro de un comando como cualquier otra función (esto es, invocando el nombre de la función), y puede retornar un valor escalar. Las rutinas almacenadas pueden llamar otras rutinas almacenadas.

Desde MySQL 5.0.1, los procedimientos almacenados o funciones se asocian con una base de datos. Esto tiene varias implicaciones:

Cuando se invoca la rutina, se realiza implícitamente **USE** *db_name* (y se deshace cuando acaba la rutina). Los comandos **USE** dentro de procedimientos almacenados no se permiten.

Se pueden calificar los nombres de rutina con el nombre de la base de datos. Esto puede usarse para referirse a una rutina que no esté en la base de datos actual. Por ejemplo, para invocar procedimientos almacenados **p** o funciones **f** de la base de datos **test**, se puede decir **CALL test.p()o test.f().**

Cuando se borra una base de datos, todos los procedimientos almacenados asociados con ella también se borran.

En MySQL 5.0.0, los procedimientos almacenados son globales y no asociados con una base de datos. Heredan la base de datos por defecto del llamador. Si se ejecuta

USE db_name desde la rutina, la base de datos por defecto original se restaura a la salida de la rutina.

MySQL soporta la extensión muy útil que permite el uso de comandos regulares **SELECT** (esto es, sin usar cursores o variables locales) dentro de los procedimientos almacenados. El conjunto de resultados de estas consultas se envía directamente al cliente. Comandos **SELECT** múltiples generan varios conjuntos de resultados, así que el cliente debe usar una biblioteca cliente de MySQL que soporte conjuntos de resultados múltiples. Esto significa que el cliente debe usar una biblioteca cliente de MySQL como mínimos desde 4.1.

CREATE PROCEDURE y CREATE FUNCTION

Para crear una función se utiliza **CREATE FUNCTION**, para crear un procedimiento **CREATE PROCEDURE.**

Estos comandos crean una rutina almacenada. Desde MySQL 5.0.3, para crear una rutina, es necesario tener el permiso **CREATE ROUTINE**, y los permisos **ALTER ROUTINE y EXECUTE** se asignan automáticamente a su creador.

Por defecto, la rutina se asocia con la base de datos actual. Para asociar la rutina explícitamente con una base de datos, especifique el nombre como **NombreBD.NombreProcedimiento** al crearlo.

Si el nombre de rutina es el mismo que el nombre de una función de SQL, necesita usar un espacio entre el nombre y el siguiente paréntesis al definir la rutina, o hay un error de sintaxis. Esto también es cierto cuando invoca la rutina posteriormente.

La sintaxis para **crear** un procedimiento almacenado es:

```
CREATE PROCEDURE NombreProcedimiento ([parametros, ….....])
[características.....] CuerpoProcedimiento
```

Dónde:

```
parámetros:
     [IN | OUT | INOUT] NombreParametro tipo
     tipo: cualquier tipo válido MySQL

características:
     LANGUAGE SQL |
     [NOT] DETERMINISTIC
     | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL
DATA}
     |SQL SECURITY {DEFINER |INVOKER} | COMMENT 'cadena'

CuerpoProcedimiento: cualquier secuencia de instrucciones SQL
válida para procedimientos almacenados.
```

La sintaxis para **crear** una función es:

```
CREATE FUNCTION NombreFuncion ([parametros[,...]])
RETURNS typo
[características ...] CuerpoFuncion
```

Parámetros: La lista de parámetros siempre debe estar presente y si no se necesita ninguno los paréntesis son obligatorios. Los parámetros son de entrada (IN) por defecto, pero pueden ser de salida que son devueltos en la llamada (OUT) o de entrada salida (INOUT) permite pasar valores al procedimiento que serán modificados y devueltos en la llamada.

Características:

- **DETERMINISTIC**: indica si es determinista o no, es decir, si siempre produce el mismo resultado.

  Si no se indica DETERMINISTIC ni NOT DETERMINISTIC por defecto es NOT DETERMINISTIC.

```
Ejemplo de sentencias DETERMINISTIC y NOT DETERMINISTIC

SELECT RAND(1); COMMENT produce siempre el mismo resultado

SELECT RAND(); COMMENT no produce el mismo resultado
```

- **CONTAINS SQL | NO SQL**: especifica si contiene sentencias SQL o no.
- **SQL DATA | MODIFIES SQL DATA**: indica si las sentencias modifican o no los datos.
- **SQL SECURITY** permite indicar si en la ejecución del procedimiento almacenado se utilizan los privilegios del creador del procedimiento DEFINER (por defecto) o del usuario del procedimiento, es decir, el que lo invoca INVOKER.

CONTAINS SQL es el valor por defecto si no se dan explícitamente ninguna de estas características.

RETURNS puede especificarse sólo con **FUNCTION**, donde es obligatorio. Se usa para indicar el tipo de retorno de la función, y el cuerpo de la función debe contener un comando **RETURN value**.

COMMENT es una extensión de MySQL, y puede usarse para describir el procedimiento almacenado. Esta información se muestra con los comandos **SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION**.

Ejemplo: vamos a crear un procedimiento que muestre por pantalla el mensaje "Buenos días, comienza la clase".

```
1.- DELIMITER $
2.- DROP PROCEDURE IF EXISTS BuenosDias$
3.- CREATE PROCEDURE BuenosDias()
4.- BEGIN
5.- SELECT 'Buenos días, comienza la clase';
6.- END$
7.- DELIMITER ;
```

Línea 1: la palabra clave DELIMITER indica el carácter de comienzo y fin del procedimiento. Normalmente se usaría el ";" pero como se necesita un ";" para cada sentencia SQL, dentro del procedimiento es conveniente usar otro carácter delimitador que no se utiliza en el procedimiento. (Normalmente $, $$, /, //)

Línea 2: Eliminamos el procedimiento si es que existe. Esto evita errores cuando queremos modificar un procedimiento existente.

Línea 3: Indica el comienzo de la definición de un procedimiento donde debe aparecer el nombre seguido por los paréntesis entre los que se pondrán los parámetros en caso de haberlos. El nombre del procedimiento puede ir precedido por el nombre de la base de datos.

```
CREATE PROCEDURE UD14Procedimientos.BuenosDias()
```

Línea 4: **BEGIN** indica el comienzo de una serie de bloques de sentencias sql que forman el cuerpo del procedimiento.

Línea 5: Conjunto de sentencias SQL, en el ejemplo **SELECT** que imprime el mensaje.

Línea 6: Fin de la definición del procedimiento seguido de un $ indicando que ya se termina el procedimiento.

Para crear el procedimiento ejecutamos las sentencias, para ejecutar el procedimiento:

```
CALL BuenosDias();
```

Las sentencias BEGIN y END solo son necesarias en caso de tener más de una sentencia.

```
CREATE PROCEDURE Version()
SELECT VERSION();
CALL VERSION();
```

Ejemplo: vamos a crear un procedimiento que muestre por pantalla la fecha y un número aleatorio.

```
DELIMITER $
CREATE PROCEDURE Fecha()
LANGUAGE SQL
NOT DETERMINISTIC
COMMENT 'Procedimiento que nos da la fecha y un número aleatorio'
SELECT CURDATE(), RAND(); $
CALL Fecha();
```

Las líneas **LANGUAGE** para indicar el lenguaje, **NOT DETERMINISTIC** que indica que el algoritmo no siempre produce el mismo resultado cada vez que es llamado y **COMMENT** para documentar el procedimiento.

El siguiente es un ejemplo de un procedimiento almacenado que usa dos parámetros uno IN y el otro **OUT**. El ejemplo usa el comando **delimiter** para cambiar el delimitador de final de **";"** a **"$"** mientras se define el procedimiento. Esto permite pasar el delimitador **";"** usado en el cuerpo del procedimiento a través del servidor en lugar de ser interpretado por el mismo **mysql**.

Ejemplo: Se va a crear un procedimiento para sumar los sueldos de los empleados de una categoría que introducimos como parámetro.

```
delimiter $$
CREATE PROCEDURE TotalSalarios(Categ VARCHAR(20))
BEGIN
     SELECT Sum(Sueldo) FROM Empleados WHERE Categ = Categoria;
END$$
delimiter ;
```

Un procedimiento almacenado se puede modificar con ALTER PROCEDURE y se puede borrar con DROP PROCEDURE.

Invocación del procedimiento almacenado

Para ejecutar un procedimiento ya creado en el servidor se utiliza la sentencia **CALL**.

```
CALL NombreProcedimiento ([parametros, ….....])
```

El procedimiento almacenado puede devolver resultados mediante los parámetros de tipo **OUT** o **INOUT**.

Por ejemplo, para utilizar el procedimiento creado:

```
CALL TotalSalarios('Administrativo');
CALL TotalSalarios('Representante');
```

Mismo procedimiento con un parámetro de salida

```
DROP PROCEDURE IF EXISTS TotalSalariosSalida;
DELIMITER $
CREATE PROCEDURE TotalSalariosSalida(Categ VARCHAR(20), OUT TOTAL
DOUBLE)
BEGIN
     SELECT Sum(Sueldo) INTO Total FROM Empleados
     WHERE Categ = Categoria;
END$;
CALL TotalSalariosSalida('Representante', @var);
SELECT @var;
```

Ejemplo Crear un procedimiento PrIntercambio(),que nos permite intercambiar el valor de 2 variables,

1. Creamos las variables

   ```
   SET @VAR1 = 7;
   SET @VAR2 = 25;
   ```

2. Comprobamos el valor de las variables

   ```
   SELECT @VAR1, @VAR2;
   ```

3. Creamos el procedimiento

   ```
   DELIMITER $
   DROP PROCEDURE IF EXISTS PrIntercambio $
   CREATE PROCEDURE PrIntercambio(INOUT A INT, INOUT B INT)
   BEGIN
      DECLARE Aux INT;
      SELECT A, B;
      SET Aux = A;
      SET A = B;
      SET B = Aux;
   END $$
   DELIMITER ;
   ```

4. Ejecutamos

   ```
   CALL PrIntercambio(@VAR1, @VAR2);
   ```

5. Comprobamos el valor de las variables **SELECT @VAR1, @VAR2;**

Ejemplo de una función que toma un parámetro, y realiza una operación con una función SQL (CONCAT), y retorna el resultado:

```
DROP FUNCTION IF EXISTS Saludo;
delimiter $$
```

```
CREATE FUNCTION Saludo (Nombre CHAR(20)) RETURNS CHAR(50)
RETURN CONCAT('Hola, ',Nombre,'!')$$
delimiter ;

SELECT Saludo('Maria');
```

Podemos ejecutar un procedimiento o función desde otra base de datos:

```
SELECT UD14Procedimientos.Saludo(Nombre)
FROM Empleados;
```

Si el comando **RETURN** en un procedimiento almacenado retorna un valor con un tipo distinto al especificado en la cláusula **RETURNS** de la función, el valor de retorno se convierte al tipo apropiado. Por ejemplo, si una función retorna un valor **ENUM** o **SET**, pero el comando **RETURN** retorna un entero, el valor retornado por la función es la cadena para el miembro de **ENUM** correspondiente de un conjunto de miembros **SET**.

Ejemplo de función que toma un parámetro de entrada y comprueba su valor. Según cuál sea se asignará con el comando SET el valor adecuado a la variable estado que es devuelta.

```
DROP FUNCTION IF EXISTS Estado;

DELIMITER $

CREATE FUNCTION Estado(InEstado CHAR(1)) RETURNS VARCHAR(20)
BEGIN
   DECLARE EstadoActual VARCHAR(20);
   IF InEstado = 'P' THEN SET EstadoActual = 'Caducado';
   ELSEIF InEstado = 'O' THEN SET EstadoActual = 'Activo';
   ELSEIF InEstado = 'N' THEN SET EstadoActual = 'Nuevo';
   END IF;
   RETURN (EstadoActual);
END$

SELECT Estado('P'), Estado('N'), Estado('O');
```

Ejemplo de función que toma un número como parámetro de entrada y comprueba si es par o impar. Si es par devuelve TRUE y si es impar FALSE.

```
DROP FUNCTION IF EXISTS EsImpar;
DELIMITER $
CREATE FUNCTION EsImpar(Numero INT) RETURNS INT
BEGIN
   DECLARE Impar INT;
   IF MOD(numero, 2) = 0 THEN SET Impar = FALSE;
   ELSE SET Impar = TRUE;
   END IF;
   RETURN (Impar);
END$

SELECT EsImpar(25), EsImpar(32);
```

Una función puede ser llamada desde la línea de comandos, como en el ejemplo anterior o lo más usual es desde otra función o procedimiento como en el siguiente ejemplo.

```
DELIMITER $
DROP PROCEDURE IF EXISTS MuestraEstado $
CREATE PROCEDURE MuestraEstado(IN Numero INT)
BEGIN
     IF (EsImpar(Numero)) THEN SELECT CONCAT(Numero, ' es Impar');
     ELSE SELECT CONCAT(Numero, ' es Par');
     END IF;
END$
delimiter ;
CALL MuestraEstado(21);
CALL MuestraEstado(22);
```

Las funciones permiten reducir la complejidad del código encapsulándolo y simplificando el mantenimiento y la legibilidad.

En las funciones no se puede diferenciar en el tipo de parámetros.

ALTER PROCEDURE y ALTER FUNCTION

```
ALTER {PROCEDURE | FUNCTION} NombreProcedimiento |
NombreFuncion [Caracteristicas ...]
Caracteristicas:
{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
| COMMENT 'Cadena'
```

Este comando puede usarse para cambiar las características de un procedimiento o función almacenada. Debe tener el permiso **ALTER ROUTINE** para la rutina desde MySQL 5.0.3. El permiso se otorga automáticamente al creador de la rutina.

DROP PROCEDURE y DROP FUNCTION

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] NombreProcedimiento |
NombreFuncion
```

Este comando se usa para borrar un procedimiento o función almacenados. Es decir, la rutina especificada se borra del servidor. Debe tener el permiso **ALTER ROUTINE** para las rutinas desde MySQL 5.0.3. Este permiso se otorga automáticamente al creador de la rutina.

La cláusula **IF EXISTS** es una extensión de MySQL. Evita que ocurra un error si la función o procedimiento no existe. Se genera una advertencia que puede verse con **SHOW WARNINGS**.

SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION

```
SHOW CREATE {PROCEDURE | FUNCTION} NombreProcedimiento |
NombreFuncion
```

Este comando es una extensión de MySQL . Similar a **SHOW CREATE TABLE**, retorna la cadena exacta que puede usarse para recrear la rutina nombrada.

```
SHOW CREATE FUNCTION EmpresasAbc.Saludo;
```

SHOW PROCEDURE STATUS y SHOW FUNCTION STATUS

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

Este comando es una extensión de MySQL . Retorna características de rutinas, como el nombre de la base de datos, nombre, tipo, creador y fechas de creación y modificación. Si no se especifica un patrón, le lista la información para todos los procedimientos almacenados, en función del comando que use.

```
SHOW FUNCTION STATUS LIKE 'Saludo';
```

También puede obtener información de rutinas almacenadas de la tabla **ROUTINES** en **INFORMATION_SCHEMA**.

Ejemplo: Crear un procedimiento para conocer los empleados que nacieron en un determinado año.

```
DROP PROCEDURE IF EXISTS AnhoNacimiento;
DELIMITER $
CREATE PROCEDURE AnhoNacimiento(Anho INT)
BEGIN
   SELECT Nombre, FecNacimiento FROM Empleados
   WHERE Anho = YEAR(FecNacimiento);
END $
DELIMITER ;
```

Para ejecutar un procedimiento se llama con la sentencia **CALL**

```
CALL AnhoNacimiento(1956)
CALL AnhoNacimiento(1966)
```

Ejemplo: Crear una función que calcule la edad

```
DROP FUNCTION CalcularEdad;
delimiter $
CREATE FUNCTION CalcularEdad(fecha1 DATE, fecha2 DATE) RETURNS INT
BEGIN
     DECLARE Edad INT;
   SET Edad = (YEAR(fecha2) - YEAR(fecha1)) - IF(RIGHT(fecha1,5) <
   RIGHT(fecha2, 5),0,1);
   RETURN Edad;
END$
delimiter ;
```

Ejemplo: ejecutar la función

```
SELECT CalcularEdad('1988-08-16', CURDATE());
```

Ejemplo: vamos a calcular la edad de todos los empleados utilizando la función CalcularEdad().

```
SELECT Nombre, CalcularEdad(FecNacimiento, CURDATE()) AS Edad FROM
Empleados;
```

La función se puede utilizar las veces que sean necesarias y en cualquier claúsula.

```
SELECT Nombre, CalcularEdad(FecNacimiento, CURDATE()) AS Edad FROM
Empleados
WHERE CalcularEdad(FecNacimiento, CURDATE()) >= 65;
```

Múltiples sentencias SQL

Sentencia compuesta BEGIN ... END

En un procedimiento almacenado se pueden ejecutar múltiples sentencias SQL, que se definen dentro de un bloque **BEGIN** … **END**.

```
[EtiquetaInicio] BEGIN
sentencias SQL válidas dentro de un procedimiento almacenado
END [EtiquetaFin]
```

Un comando compuesto puede etiquetarse. *EtiquetaFin* no puede existir a no ser que también esté presente *EtiquetaInicio*, y si ambos lo están, deben coincidir.

Declaraciones

En una declaración se pueden crear:

- Variables locales
- Condiciones
- Cursores
- Manejadores

La declaración de una variable local, una condición, un cursor o un manejador solamente puede aparecer al principio de un bloque BEGIN … END. Si se necesitan hacer diferentes declaraciones, éstas deben hacerse en el siguiente orden:

- Declaración de variables y condiciones
- Declaración de cursores
- Declaración de manejadores

Sentencias DECLARE

La sentencia DECLARE se utiliza para definir las variables del procedimiento almacenado y las declaraciones deben incluirse en la parte superior del procedimiento almacenado, después de la sentencia BEGIN. Estas variables sólo tienen ámbito de validez dentro del procedimiento almacenado.

Además de las variables definidas con la sentencia DECLARE, en el procedimiento almacenado se puede utilizar los parámetros recibidos en la llamada (sin necesidad de declararlos) y las variables del sistema.

La sintaxis es:

```
DECLARE NombreVariable [,....] tipo [DEFAULT valor]
tipo cualquier tipo válido MySQL
```

Por ejemplo:

```
DECLARE Codigo INT;
DECLARE HOY DATETIME;
DECLARE A, B INT DEFAULT 5;
DECLARE CONTADOR INT;
```

Las variables locales se pueden declarar dentro de alguna rutina en la misma línea (siempre y cuando sean del mismo tipo), separando cada una por una coma. Para darle un valor a éstas o para inicializarlas, se utilizará la instrucción SET.

Sentencias SET

La sentencia SET permite hacer referencia a variables locales declaradas dentro del procedimiento o a variables globales del servidor para asignarles valores o el resultado de una expresión o función.

La sintaxis es:

```
SET NombreVariable1 = expresión1 [,.NombreVariable2 =
expresión2, ...]
```

Por ejemplo:

```
SET Codigo = 0;
SET HOY = NOW();
```

Ejemplo. Crear una función que calcule el Importe de IVA, pasándole como parámetro el importe y el % a aplicar.

```
DROP FUNCTION IF EXISTS FImporteIVA;
delimiter $
CREATE FUNCTION FImporteIVA(Importe DOUBLE, Porcentaje DOUBLE)
RETURNS DOUBLE
BEGIN
   DECLARE IVA DOUBLE;
   SET IVA = Importe * Porcentaje;
   RETURN IVA;
END$
delimiter ;
SELECT FImporteIVA(1589, 0.21)
```

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 una variable que se encuentra fuera de un procedimiento salvo que la asignemos a un parámetro out o a una variable de sesión (usando la @)

```
DELIMITER $
DROP PROCEDURE IF EXISTS AlcanceVariables $
CREATE PROCEDURE AlcanceVariables()
BEGIN
   DECLARE X1 CHAR(10) DEFAULT 'Primero';
   BEGIN
        DECLARE X1 CHAR(10) DEFAULT 'Segundo';
     SELECT X1;
   END;
   SELECT X1;
END$
DELIMITER ;
CALL AlcanceVariables();
```

Estructuras de control de flujo en Procedimientos almacenados

En los procedimientos almacenados se pueden utilizar estas sentencias de control de flujo:

- IF
- CASE
- WHILE
- ITERATE
- LEAVE.

Sentencia IF

Sintaxis básica:

```
IF Condicion THEN ListaSentencias
[ELSEIF Condicion THEN ListaSentencias]
…….
[ELSE ListaSentencias]
END IF
```

Por ejemplo:

```
DROP FUNCTION IF EXISTS Notas;
delimiter $
CREATE FUNCTION Notas (Nota INT) RETURNS CHAR (50)
BEGIN
     DECLARE Calificacion CHAR(50);
     IF Nota < 5 THEN SET Calificacion = "Suspenso";
     ELSEIF NOTA < 7 THEN SET Calificacion = "Aprobado";
          ELSEIF Nota < 9 THEN SET Calificacion = "Notable";
               ELSE SET Calificacion = "Sobresaliente";
     END IF;
     RETURN Calificacion;
END $
Select Notas(4), Notas(6), Notas(8), Notas(10);
```

Ejemplo: Crear un procedimiento que inserte en la tabla Prueba según sea el valor de entrada.

Creamos la tabla Prueba.

```
CREATE TABLE Prueba
( Id INT AUTO_INCREMENT PRIMARY KEY,
Columna1 INT);
```

Creamos el procedimiento.

```
DROP PROCEDURE IF EXISTS PrPrueba ;
DELIMITER $
CREATE PROCEDURE PrPrueba(IN Numero INT) LANGUAGE SQL MODIFIES SQL
DATA
BEGIN
   IF Numero > 0 THEN INSERT INTO Prueba (Columna1) VALUES (25);
   END IF;
   IF Numero = 0 THEN UPDATE Prueba SET Columna1 = Columna1 + 1;
   ELSE UPDATE Prueba SET Columna1 = Columna1 + 2;
   END IF;
END $
```

Probamos el procedimiento.

```
CALL(12);
SELECT * FROM prueba p;
```

```
CALL PrPrueba(0);
SELECT * FROM prueba p;
CALL PrPrueba(-1);
SELECT * FROM prueba p;
CALL PrPrueba(1);
SELECT * FROM prueba p;
```

Sentencia CASE

Sintaxis básica:

```
CASE CasoValor
[WHEN Valor THEN Sentencia]
[ELSE Sentencia]
END CASE
```

Por ejemplo: Incrementar el precio de los productos en un 2% si la opción es 1, en un 5% si es 2 y en un 7% si opción vale 3.

```
DROP PROCEDURE IF EXISTS IncrementarPrecio;
delimiter $
CREATE PROCEDURE IncrementarPrecio (Opcion INT)
BEGIN
   CASE Opcion
      WHEN 1 THEN UPDATE Productos SET Precio = Precio + Precio *
   0.02;
      WHEN 2 THEN UPDATE Productos SET Precio = Precio + Precio *
   0.05;
      WHEN 3 THEN UPDATE Productos SET Precio = Precio + Precio *
   0.07;
      ELSE UPDATE Productos SET Precio = Precio + Precio * 0.01;
   END CASE;
END $
```

Comprobamos los resultados:

```
Select precio from productos;
call IncrementarPrecio(3);
Select precio from productos;
```

Actualiza la tabla Numeros según el valor de opción. Creamos la tabla

```
CREATE TABLE Numeros ( Numero INT NOT NULL DEFAULT '0', PRIMARY
KEY (Numero) ) ENGINE=InnoDB;
```

Creamos el procedimiento

```
DROP PROCEDURE IF EXISTS PrNumeros;
DELIMITER $
CREATE PROCEDURE PrNumeros (Opcion INT)
BEGIN
```

```
CASE Opcion
WHEN 0 THEN INSERT INTO Numeros (Numero) VALUES(32), (25), (18);
WHEN 1 THEN UPDATE Numeros SET Numero = Numero + Numero * 2;
WHEN 2 THEN UPDATE Numeros SET Numero = Numero + Numero * 5;
WHEN 3 THEN UPDATE Numeros SET Numero = Numero + Numero * 7;
ELSE UPDATE Numeros SET Numero = Numero + Numero * 10;
END CASE;
END $
delimiter ;
```

Ejecutamos el procedimiento

```
CALL PrNumeros(0);
SELECT * FROM numeros n;
CALL PrNumeros(1);
SELECT * FROM numeros n;
```

Sentencia LOOP

La sentencia LOOP nos permite definir un bucle para repetir una serie de sentencias hasta que decidamos salir del bucle con la sentencia LEAVE.

Sintaxis básica:

```
[EtiquetaInicioLOOP: ] LOOP
Sentencias
END LOOP [EtiquetaFinLOOP]
```

Si se utilizan las dos etiquetas, los nombres deben ser iguales.

Ejemplo: Insertamos valores en la tabla Prueba mientras no lleguemos a una condición.

```
DROP PROCEDURE IF EXISTS PrLoop;
DELIMITER $
CREATE PROCEDURE PrLoop()
BEGIN
DECLARE Contador INT;
SET Contador = 0;
Etiqueta: LOOP
INSERT INTO Prueba (Columna1) VALUES(Contador);
SET Contador = Contador + 1;
IF Contador >= 5 THEN LEAVE Etiqueta;
END IF;
END LOOP;
END $
DELIMITER ;
```

Comprobación

```
SELECT * FROM Prueba;
```

```
CALL PrLoop();
SELECT * FROM Prueba;
```

Sentencia LEAVE

La sentencia LEAVE nos permite salir de un bucle.

Sintaxis básica:

```
LEAVE Etiqueta
```

Sale de cualquier bloque de control del flujo.

Sentencia ITERATE

Esta sentencia indica el reinicio de un bucle. Puede aparecer dentro de bloques LOOP, REPEAT y WHILE.

Sintaxis básica:

```
ITERATE Etiqueta
```

Por ejemplo:

```
Paso1: LOOP
SET x = x * (x + 1);
INSERT INTO Tabla1 VALUES (x);
IF x < 1000 THEN ITERATE Paso1;
END IF;
END LOOP Paso1;
```

Ejemplo:

```
DROP PROCEDURE IF EXISTS PrLoopLeaveIterate;

delimiter $

CREATE PROCEDURE PrLoopLeaveIterate(valor INT)
BEGIN
Etiqueta: LOOP
INSERT INTO Prueba (Columna1) VALUES(valor);
SET valor = valor + 1;
IF valor < 10 THEN ITERATE Etiqueta;
END IF;
LEAVE Etiqueta;
END LOOP Etiqueta;
SELECT valor;
END$
DELIMITER ;

delete from prueba;

select * from prueba;

CALL PrLoopLeaveIterate(1);
```

Sentencia REPEAT

Esta sentencia repite el bloque hasta que se cumpla la condición.

Sintaxis básica:

```
[EtiquetaInicio: ] REPEAT
Sentencias
…...
UNTIL Condicion
END REPEAT [EtiquetaFin]
```

Si se utilizan las dos etiquetas, los nombres deben ser iguales.

Por ejemplo: Insertar tantos valores en la tabla Prueba, como se lo indicamos en el parámetro.

```
DROP PROCEDURE IF EXISTS PrInsertarNumeros;

DELIMITER $
CREATE PROCEDURE PrInsertarNumeros(IN Limite INT)
BEGIN
     DECLARE x INT DEFAULT 1;
     DELETE FROM Prueba;
     Paso1: REPEAT
          SET x = x + 1;
          INSERT INTO Prueba (Columna1) VALUES (x);
          UNTIL x > Limite
     END REPEAT Paso1;
     SELECT * FROM Prueba;
END $
DELIMITER ;

CALL PrInsertarNumeros(15);
```

Ejemplo: Crear una función que calcule el factorial de un número.

```
DROP FUNCTION IF EXISTS FuFactorialRepeat;
DELIMITER $$
CREATE FUNCTION FuFactorialRepeat(Numero INT) RETURNS int(11)
BEGIN
   DECLARE Factorial INT DEFAULT 1;
   DECLARE Contador INT DEFAULT 1;
   REPEAT
        SET Factorial = Factorial * Contador;
        SET Contador = Contador + 1;
        UNTIL Contador > Numero
   END REPEAT;
   RETURN Factorial;
END $$
DELIMITER ;
SELECT FuFactorialRepeat(5);
```

Sentencia WHILE

Esta sentencia repite el bloque mientras que se cumpla la condición.

Sintaxis básica:

```
[EtiquetaInicio: ] WHILE Condicion DO
Sentencias
END WHILE [EtiquetaFin]
```

Indica la repetición de un bucle. Puede aparecer dentro de bloques LOOP, REPEAT y WHILE.

Si se utilizan las dos etiquetas, los nombres deben ser iguales.

Por ejemplo:

```
DELIMITER $
CREATE PROCEDURE PInsertarNumerosWhile(IN Limite INT)
BEGIN
   DECLARE x INT DEFAULT 1;
   DELETE FROM Numeros;
   Paso1: WHILE x < Limite DO
        INSERT INTO Numeros VALUES (x);
        SET x = x + 1;
   END WHILE Paso1;
   SELECT * FROM Numeros;
END $
DELIMITER ;
CALL PInsertarNumerosWhile(15);
```

Ejemplo: Crear una función que calcule el factorial de un número.

```
DROP FUNCTION IF EXISTS Factorial;
delimiter $
CREATE FUNCTION Factorial (Numero INT) RETURNS INT
BEGIN
   DECLARE Factorial INT DEFAULT 1;
   DECLARE Contador INT DEFAULT 1;
   WHILE Contador <= Numero DO
        SET Factorial = Factorial * Contador;
        SET Contador = Contador + 1;
  END WHILE;
  RETURN Factorial;
END $
DELIMITER ;
SELECT Factorial(6);
```

Ejemplos de Procedimientos almacenados

En los procedimientos se pueden utilizar cualquier tipo de sentencias SQL de definición (drop, create), de manipulación (insert, update y delete) y de selección (select).

```
DROP PROCEDURE IF EXISTS PrDefinicion ;
DELIMITER $
CREATE PROCEDURE PrDefinicion()
BEGIN
  DECLARE Contador INT DEFAULT 1;
  /* INSTRUCCIONES DDL , de definición */
  DROP TABLE IF EXISTS Definicion;
  CREATE TABLE Definicion (Id INT PRIMARY KEY,
  Columna1 VARCHAR(30))ENGINE = InnoDB;
  /* INSERT usando una variable de procedimiento */
  WHILE (Contador < 10) DO
    INSERT INTO Definicion (Id, Columna1) VALUES (Contador,
    CONCAT('Registro ', Contador));
    SET Contador = Contador + 1;
  END WHILE;
/* Ejemplo de actualización usando variables del procedimiento*/
  SET Contador = 5;
  UPDATE Definicion SET Columna1 = CONCAT('Modificada fila ',
  Contador) WHERE Id = Contador;
/* DELETE usando una variable de procedimiento */
  DELETE FROM Definicion WHERE Id > Contador;
END $
```

```
DELIMITER ;
CALL PrDefinicion();
SELECT * FROM Definicion;
```

Sentencias SELECT.....INTO

La sentencia SELECT …..INTO permite almacenar el resultado de una fila de una consulta en variables del programa.

La sintaxis es:

```
SELECT NombreColumna [,....] INTO NombreVariable FROM......
```

Ejemplo

```
SELECT COUNT(*) INTO Contador
```

Ejemplo crear una función que nos devuelva el número de empleados que han nacido en un determinado año que se lo pasamos como parámetro.

```
DROP FUNCTION IF EXISTS FuEmpladosAnho;
DELIMITER $
CREATE FUNCTION FuEmpladosAnho (Anho INT) RETURNS INT
BEGIN
   DECLARE contadorEmpleados INT DEFAULT 0;
   SELECT COUNT(*) INTO contadorEmpleados FROM Empleados
   WHERE Anho = YEAR(FecNacimiento);
   RETURN contadorEmpleados;
END $
DELIMITER ;

SELECT FuEmpladosAnho(1966);
```

Ejemplo que utiliza la propiedad de la sentencia SELECT de enviar valores a variables usando INTO.

```
delimiter $

CREATE PROCEDURE PrMiProcedimiento(Codigo CHAR(5))
BEGIN
  DECLARE PNombre VARCHAR(30);
  DECLARE PCategoria VARCHAR(30);
  DECLARE PVentas DOUBLE;
  SELECT Nombre, Categoria, Ventas INTO PNombre, PCategoria, PVentas
  FROM RepuestosHard.Empleados
  WHERE CodEmpleado = Codigo;
  SELECT Pnombre, Pcategoria, PVentas;
END$

DELIMITER ;

CALL PrMiProcedimiento(101);
```

Por ejemplo: Crear un procedimiento que busque la categoría de un empleado y después liste todos los empleados de esa categoría.

```
DROP PROCEDURE IF EXISTS PrCategoriaEmpleados;

DELIMITER $

CREATE PROCEDURE PrCategoriaEmpleados(Codigo CHAR(5))
BEGIN
  DECLARE PNombre VARCHAR(30);
  DECLARE PCategoria VARCHAR(30);
  DECLARE PVentas DOUBLE;
  SELECT Nombre, Categoria, Ventas INTO PNombre, PCategoria, PVentas
  FROM RepuestosHard.Empleados
  WHERE CodEmpleado = Codigo;
  SELECT Nombre FROM RepuestosHard.Empleados WHERE PCategoria =
  Categoria;
END$

delimiter ;

CALL PrCategoriaEmpleados(101);
```

En estos ejemplos podemos observar que la sentencia SELECT asigna los valores de la fila seleccionada para asignarlos a su vez a nuevas variables internas del procedimiento. Puede ocurrir que muchas veces necesitemos recuperar más de una fila para manipular sus datos, en estos casos no nos sirve la sentencia SELECT …. INTO y se necesitan los cursores.

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 comportamiento es el predeterminado, cuando necesitamos que la rutina almacenada se comporte de diferente forma tenemos que definir controladores de excepciones.

Ejemplo de un procedimiento sin manejo de errores. Insertamos clientes en la BD EmpresasAbc.

```
delimiter $

CREATE PROCEDURE PrInsertarClientes(PrCodCliente CHAR(5), PrNombre
VARCHAR(50), PrCodRepCliente CHAR(5),
PrLimiteCredito DOUBLE)
MODIFIES SQL DATA
BEGIN
  INSERT INTO EmpresasAbc.Clientes(CodCliente, Nombre,
  CodRepCliente, LimiteCredito) VALUES (PrCodCliente, PrNombre,
  PrCodRepCliente, PrLimiteCredito);
  SELECT * FROM EmpresasAbc.Clientes;
END $

Delimiter ;
```

Funciona bien cuando no existe el cliente que insertamos

```
CALL PrInsertarClientes('586', 'Isabel García', '101', 2000);
SELECT * FROM EmpresasAbc.Clientes;
```

Sin embargo, si volvemos a ejecutar el mismo procedimiento, el cliente ya existe.

```
CALL PrInsertarClientes('586', 'Isabel García', '101', 2000);
```

Se produce un error ERROR 1062 Duplicate Key…,.que nos indica que hay una clave repetida y el SELECT no se ejecuta.

En general los errores deben ser prevenidos y tratados.

El mismo procedimiento con manejo de errores

```
delimiter $
CREATE PROCEDURE PrInsertarClientesManejoErrores(PrCodCliente
CHAR(5), PrNombre VARCHAR(50), PrCodRepCliente CHAR(5),
PrLimiteCredito DOUBLE, OUT Estado VARCHAR(45))
MODIFIES SQL DATA
BEGIN
     DECLARE CONTINUE HANDLER FOR 1062 SET Estado = 'Clave
     Duplicada';
     SET Estado = 'OK';
     INSERT INTO EmpresasAbc.Clientes(CodCliente, Nombre,
     CodRepCliente, LimiteCredito) VALUES (PrCodCliente, PrNombre,
     PrCodRepCliente, PrLimiteCredito);
     SELECT * FROM EmpresasAbc.Clientes;
END $
Delimiter ;
CALL PrInsertarClientesManejoErrores('586', 'Isabel García', '101',
2000, @Estado);
```

Se produce un error, si el, que es un aviso, en caso de que el cliente exista, pero el procedimiento sigue su ejecución. Comprobamos que ejecuta la sentencia SELECT y lista los clientes.

Si queremos hacer algo con el error debemos usar la variable @Estado. En el siguiente ejemplo vamos a llamar al procedimiento desde otro procedimiento condicionando la salida al valor de la variable Estado de tipo OUT.

Modificamos el procedimiento PrInsertarClientesManejoErrores, eliminado el SELECT.

Creamos el procedimiento siguiente

```
delimiter $
DROP PROCEDURE PrCrearComentarioError$
CREATE PROCEDURE PrCrearComentarioError(PrCodCliente CHAR(5),
PrNombre VARCHAR(50), PrCodRepCliente CHAR(5),
PrLimiteCredito DOUBLE, OUT Estado VARCHAR(45))
MODIFIES SQL DATA
BEGIN
     DECLARE Estado VARCHAR(20);
     CALL PrInsertarClientesManejoErrores(PrCodCliente, PrNombre,
     PrCodRepCliente, PrLimiteCredito, Estado);
     IF Estado = 'Clave Duplicada' THEN
          SELECT CONCAT('Warning: cliente ya existe ', PrCodCliente)
          AS Mensaje;
     END IF;
END $
delimiter ;
```

Ejecutamos

```
CALL PrCrearComentarioError('586', 'Isabel García', '101', 2000,
@Estado);
```

Declaración de condiciones

Para declarar condiciones se utiliza la siguiente sintaxis:

```
DECLARE NombreCondicion CONDITION FOR ValorCondicion
ValorCondicion puede ser:
SQLSTATE [VALUE] valor_sqlstate | código_error_mysql
```

Esta sentencia especifica condiciones de error sobre las que se precisa un tratamiento especial. El nombre de la condición se puede usar en una sentencia DECLARE HANDLER.

Este comando especifica condiciones que necesitan tratamiento específico. Asocia un nombre con una condición de error específica. El nombre puede usarse en un comando **DECLARE HANDLER**.

Por ejemplo:

```
DECLARE sin_acceso CONDITION FOR SQLSTATE “28000”;
```

Declaración de manipuladores

Para declarar condiciones que pueden tratar una o más condiciones y, cuando se cumple una de las condiciones, se ejecuta la sentencia especificada.

La instrucción DECLARE … HANDLER asocia una o más condiciones con una instrucción a ser ejecutada cuando alguna de las condiciones ocurre. El valor del *Sentencia* indica qué ocurre cuando la condición se ejecuta. Con la instrucción CONTINUE, la ejecución de la instrucción continúa, con la instrucción EXIT el bloque BEGIN actual terminará.

Se utiliza la siguiente **sintaxis**:

```
DECLARE TipoManipulador HANDLER FOR ValorCondicion [,...]
Sentencia

TipoManipulador:

CONTINUE: la ejecución de la rutina vigente continúa después
          de ejecutar el manipulador

EXIT: la sentencia BEGIN | END finaliza

ValorCondicion:

SQLSTATE [VALUE] valor_sqlstate

NombreCondicion:

SQLWARNING: representa a todos los códigos SQLSTATE que
          comienzan por 01.

NOT FOUND: representa a todos los códigos SQLSTATE que
          comienzan por 02.

SQLEXCEPTION: representa a todos los códigos SQLSTATE no
          contemplados en SQLWARNING y NOT FOUND.

Código_error_mysql
```

Tipos de Manipulador

Exit: cuando se encuentra un error el bloque que se está ejecutando actualmente se termina. Si el bloque es el bloque principal el procedimiento se termina y, el controla se devuelve al procedimiento programa externo que llamó al procedimiento donde se ha producido la excepción. Si el bloque está encerrado en un bloque externo dentro del mismo programa almacenado, el control se devuelve a ese bloque exterior.

Continue: la ejecución continúa en la declaración siguiente a la que ocasionó el error.

Por ejemplo, cuando se produzca el SQLSTATE "2800" (denegación de acceso) se ejecutará la asignación de la variable @var, mientras no se produzca ese tipo de error la sentencia no se ejecutará

```
DECLARE EXIT HANDLER FOR SQLSTATE "28000" SET @var = "acceso
denegado";
```

Ejemplo: Crear un procedimiento PrInsertarClientes, va insertar nuevos clientes, en el caso de que el cliente exista añade 1000 al código del cliente.

```
DROP PROCEDURE IF EXISTS PrInsertarClientes01;
DELIMITER $
CREATE PROCEDURE PrInsertarClientes01 (IN Cod INT, IN Nom
VARCHAR(30), IN Repre INT, IN lim DOUBLE)
BEGIN
     DECLARE codigo INT;
     DECLARE claveDuplicada CONDITION for SQLSTATE '23000';
     DECLARE CONTINUE HANDLER FOR claveDuplicada
          INSERT INTO Clientes VALUES (cod+1000, Nom, Repre, lim);
          INSERT INTO Clientes VALUES (cod, Nom, Repre, lim);
          SELECT * FROM Clientes;
END$
DELIMITER ;
CALL PrInsertarClientes('106', 'Maria Jose Rodriguez Perez', 105,
5600);
SELECT Nombre FROM Empleados WHERE CodEmpleado = '105';
CALL PrInsertarClientes('106', 'Pedro García', 105, 5600);
```

Otro ejemplo

```
delimiter $
CREATE PROCEDURE Ejemplo ()
BEGIN
     DECLARE NombreCondicion CONDITION FOR SQLSTATE '23000';
     DECLARE EXIT HANDLER FOR NombreCondicion ROLLBACK;
     START TRANSACTION;
     INSERT INTO Numeros VALUES (5);
     INSERT INTO Numeros VALUES (1);
     COMMIT;
END$
CALL Ejemplo() #produce error ejecuta ROLLBACK
```

Otro ejemplo

```
delimiter $
CREATE PROCEDURE Ejemplo1 ()
BEGIN
     DECLARE NombreCondicion CONDITION FOR SQLSTATE '23000';
     DECLARE EXIT HANDLER FOR NombreCondicion ROLLBACK;
     START TRANSACTION;
     INSERT INTO Numeros VALUES (20);
     INSERT INTO Numeros VALUES (30);
     COMMIT;
END$
CALL Ejemplo2() #no produce error ejecuta COMMIT
SELECT * FROM numeros n;
```

Ejemplos con ambos controladores. Crear un procedimiento para insertar un registro en la tabla de empleados. Para controlar la posibilidad de que el empleado ya exista vamos a crear un manejador de tipo Exit que en caso activarse establecerá el valor de la variable ClaveDuplicada a 1 y devolverá el control al bloque BEGIN/END exterior, por eso se utilizan dos bloques

```
DELIMITER $
DROP PROCEDURE PrInsertarEmpleados$
CREATE PROCEDURE PrInsertarEmpleados(PrCodEmpleado CHAR(5), PrNombre VARCHAR(30), PrFecNacimiento DATE, PrOficina CHAR(3),
PrCategoria VARCHAR(20), PrContrato DATE, PrCodJefe CHAR(5), PrCuota DOUBLE, PrVentas DOUBLE)
MODIFIES SQL DATA
BEGIN
     DECLARE ClaveDuplicada INT DEFAULT 0;
     DECLARE EXIT HANDLER FOR 1062 SET ClaveDuplicada = 1;
     INSERT INTO EmpresasAbc.Empleados(CodEmpleado, Nombre, FecNacimiento, Oficina, Categoria, Contrato, CodJefe, Cuota, Ventas)
     VALUES (PrCodEmpleado, PrNombre, PrFecNacimiento, PrOficina, PrCategoria, PrContrato, PrCodJefe, PrCuota, PrVentas);
     IF ClaveDuplicada = 1 THEN SELECT CONCAT('Error al insertar del registro', PrCodEmpleado, ' clave duplicada') AS Resultado;
     ELSE SELECT CONCAT(PrCodEmpleado,' ', PrNombre, ' creado') AS Resultado;
     END IF;
END $
CALL PrInsertarEmpleados('523', 'Luis Ibañez', '1978/12/31', 30, 'Jefe Departamento', '2000/07/01', '108', 6000, 7800);
```

Al ejecutar el procedimiento la primera vez todo es correcto y aparece el mensaje

Si volvemos a ejecutar el procedimiento podemos comprobar que se visualiza el mensaje generado por el manejador, pero no el que hemos creado.

```
CALL PrInsertarEmpleados('523', 'Luis Ibañez', '1978/12/31', 30, 'Jefe Departamento', '2000/07/01', '108', 6000, 7800);
```

Mismo ejemplo con la funcionalidad CONTINUE

```
DELIMITER $
DROP PROCEDURE PrInsertarEmpleadosContinue$
CREATE PROCEDURE PrInsertarEmpleadosContinue(PrCodEmpleado CHAR(5),
PrNombre VARCHAR(30), PrFecNacimiento DATE, PrOficina CHAR(3),
PrCategoria VARCHAR(20), PrContrato DATE, PrCodJefe CHAR(5), PrCuota
DOUBLE, PrVentas DOUBLE)
MODIFIES SQL DATA
BEGIN
     DECLARE ClaveDuplicada INT DEFAULT 0;
     DECLARE CONTINUE HANDLER FOR 1062 SET ClaveDuplicada = 1;
     INSERT INTO RepuestosHard.Empleados(CodEmpleado, Nombre,
     FecNacimiento, Oficina, Categoria, Contrato, CodJefe, Cuota,
     Ventas)
     VALUES (PrCodEmpleado, PrNombre, PrFecNacimiento, PrOficina,
     PrCategoria, PrContrato, PrCodJefe, PrCuota, PrVentas);

     IF ClaveDuplicada = 1 THEN SELECT CONCAT('Error al insertar del
     registro', PrCodEmpleado, ' clave duplicada') AS Resultado;
     ELSE SELECT CONCAT(PrCodEmpleado,' ', PrNombre, ' creado') AS
     Resultado;
     END IF;
END $
```

Ejecutamos el procedimiento

```
CALL PrInsertarEmpleadosContinue('524', 'Maria Ibañez',
'1978/12/31', 30, 'Jefe Departamento', '2000/07/01', '108', 6000,
7800);
```

Todo Correcto

Volvemos a ejecutar el procedimiento

```
CALL PrInsertarEmpleadosContinue('524', 'Maria Ibañez',
'1978/12/31', 30, 'Jefe Departamento', '2000/07/01', '108', 6000,
7800);
```

Un controlador EXIT es más adecuado para los errores graves, porque no permite ninguna forma de continuar.

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 activan un manejador. Puede ser por un error de código, un error de SQL (SQLSTATE) o por una circunstancia definida por el usuario. Por defecto al indicar un error numérico se refiere a un error SQL.

Por ejemplo:

```
DECLARE CONTINUE HANDLER FOR 1062 SET ClaveDuplicada = 1;
```

Significa que cuando se produzca el error 1062 de MySQL la variable ClaveDuplicada se pondrá a 1.

El mismo ejemplo usando un error estándar o SQLSTATE sería:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET ClaveDuplicada = 1;
```

Los códigos de error los podemos ver:

http://dev.mysql.com/doc/refman/5.7/en/error-messages-server.html

http://manuales.guebs.com/mysql-5.0/error-handling.html

Cursores

Un cursor se asocia con un conjunto de filas o una consulta sobre una tabla de una base de datos.

Se utiliza la siguiente sintaxis:

```
DECLARE NombreCursor CURSOR FOR Sentencia_SQL
```

La instrucción DECLARE … CURSOR declara un cursor para ser asociado a algún SELECT, el cual no deberá contener la instrucción INTO. El cursor se utiliza cuando la sentencia SELECT nos puede devolver más de una fila de resultados y queremos trabajar sobre ellos. Un cursor es un gestor del conjunto

Un cursor se debe declarar (DECLARE), se debe abrir (OPEN) y se debe cerrar (CLOSE). Para navegar por el conjunto de resultados se utiliza la sentencia FETCH, es decir, FETCH se utiliza para obtener las filas resultantes del SELECT

Los cursores sólo se pueden mover hacia adelante con FETCH y sólo son de lectura.

Por ejemplo:

```
DECLARE Cursor1 CURSOR FOR SELECT CodEmpleado,Nombre FROM Empleados
WHERE Nombre LIKE 'M%';
```

Ejemplo declarar un cursor dentro de un procedimiento.

```
delimiter $
CREATE PROCEDURE PrCursor(CuCategoria CHAR(30))
BEGIN
     DECLARE PNombre VARCHAR(30);
     DECLARE PCategoria VARCHAR(30);
     DECLARE PVentas DOUBLE;
     DECLARE Cursor1 CURSOR FOR SELECT Nombre, Categoria, Ventas
     FROM Empleados WHERE Categoria = CuCategoria;
END$
delimiter ;
```

Comandos relacionados con los Cursores

Abrir un cursor

Open: inicializa el conjunto de resultados asociados con el cursor. Un cursor ya declarado se abre con la siguiente sintaxis:

```
OPEN NombreCursor
```

Por ejemplo:

```
OPEN Cursor1;
```

Obtención de un conjunto de datos-FETCH

Esta sentencia obtiene la siguiente fila del conjunto de resultados y avanza el puntero del cursor. Cuando ya no existen más filas, se produce un SQLSTATE "02000" que podemos gestionar con la declaración de una condición de error para dar por finalizado el bucle por el conjunto de resultados.

La sintaxis es:

```
FETCH NombreCursor INTO NombreVariable [, NombreVariable....]
```

Por ejemplo:

```
DECLARE Cursor1 CURSOR FOR SELECT Fecha FROM Empleados WHERE
CodEmpleado = Codigo;

OPEN Cursor1;

REPEAT

FETCH Cursor1 INTO a;
```

Cerrar un cursor

Cierra el cursor liberando la memoria que ocupa y haciendo imposible el acceso a cualquiera de sus datos. Un cursor abierto se cierra con la siguiente sintaxis:

```
CLOSE NombreCursor
```

Por ejemplo:

```
CLOSE Cursor1;
```

Ejemplo:

```
OPEN Cursor1;
FETCH Cursor1 INTO PNombre, PCategoria;
CLOSE Cursor1;
```

Ejemplo de cómo extraer una fila de una tabla:

```
DROP PROCEDURE PrCursorCategoria;
delimiter $
CREATE PROCEDURE PrCursorCategoria(CuCategoria CHAR(30))
BEGIN
  DECLARE PNombre VARCHAR(30);
  DECLARE PCategoria VARCHAR(30);
  DECLARE PVentas DOUBLE;
  DECLARE CuCategoria CURSOR FOR SELECT Nombre, Categoria, Ventas
  FROM RepuestosHard.Empleados WHERE Categoria = CuCategoria;
  OPEN CuCategoria;
  Etiqueta: LOOP
  FETCH CuCategoria INTO PNombre, PCategoria, PVentas;
  END LOOP Etiqueta;
  CLOSE CuCategoria;
END$
delimiter ;
CALL PrCursorCategoria('Representante');
```

Al ejecutar el procedimiento se produce un error.

Cuando se llega a la última fila no hay más datos que obtener por lo que se necesita alguna forma de detectar esta circunstancia. Para ello se utiliza un manejador de errores o HANDLER, que se necesita declararlo con la siguiente instrucción:

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

Que hace dos cosas:

1ª) Establecer la variable UltimaFila = 1.

2ª) Permitir al programa continuar su ejecución.

El procedimiento resultante sería:

```
DROP PROCEDURE PrCursorCategoria;
delimiter $
CREATE PROCEDURE PrCursorCategoria(CuCategoria CHAR(30))
BEGIN
     DECLARE PNombre VARCHAR(30);
     DECLARE PCategoria VARCHAR(30);
     DECLARE PVentas DOUBLE;
     DECLARE UltimaFila BOOL;
     DECLARE Contador INT;
     DECLARE CuCategoria CURSOR FOR SELECT Nombre, Categoria, Ventas
     FROM Empleados WHERE Categoria = CuCategoria;
     DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimaFila = 1;
```

```
    SET UltimaFila = 0;
    SET Contador = 0;
    OPEN CuCategoria;
    Etiqueta: LOOP
        FETCH CuCategoria INTO PNombre, PCategoria, PVentas;
        INSERT INTO empleadoscategoria (nombre, categoria, ventas)
        VALUES (PNombre, PCategoria, PVentas);
        SET Contador = Contador + 1;
        IF UltimaFila = 1 THEN LEAVE Etiqueta;
        END IF;
    END LOOP Etiqueta;
    CLOSE CuCategoria;
    SELECT Contador;
END$
delimiter ;
CALL PrCursorCategoria('Representante');
select * from empleadoscategoria;
```

Casi todos los cursores necesitan un manejador de NOT FOUND.

Se han declarado las variables UltimaFila de tipo BOOL que indica si hemos llegado a la última fila del cursor, y la variable Contador que nos indica el número de empleados de la categoría que pasamos como parámetro. Con la sentencia LEAVE se termina el bucle cuando UltimaFila tome el valor 1 al alcanzar el final del cursor.

Mismo procedimiento con **REPEAT UNTIL**

```
DROP PROCEDURE PrCursorCategoriaRepeat;
delimiter $
CREATE PROCEDURE PrCursorCategoriaRepeat(CuCategoria CHAR(30))
BEGIN
DECLARE PNombre VARCHAR(30);
DECLARE PCategoria VARCHAR(30);
DECLARE PVentas DOUBLE;
DECLARE UltimaFila BOOL;
DECLARE Contador INT;
DECLARE CuCategoria1 CURSOR FOR SELECT Nombre, Categoria, Ventas
FROM Empleados WHERE Categoria = CuCategoria;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimaFila = 1;
SET UltimaFila = 0;
SET Contador = 0;
OPEN CuCategoria1;
Etiqueta: REPEAT
```

```
FETCH CuCategoria1 INTO PNombre, PCategoria, PVentas;
INSERT INTO empleadoscategoria (nombre, categoria, ventas) VALUES
(PNombre, PCategoria, PVentas);
SET Contador = Contador + 1;
UNTIL UltimaFila
END REPEAT Etiqueta;
CLOSE CuCategoria1;
SELECT Contador;
END$
delimiter ;
CALL PrCursorCategoriaRepeat('Director Ventas');
select * from empleadoscategoria
```

Mismo procedimiento con **WHILE**

```
DROP PROCEDURE PrCursorCategoriaWhile;
delimiter $
CREATE PROCEDURE PrCursorCategoriaWhile(CuCategoria CHAR(30))
BEGIN
  DECLARE PNombre VARCHAR(30);
  DECLARE PCategoria VARCHAR(30);
  DECLARE PVentas DOUBLE;
  DECLARE UltimaFila BOOL;
  DECLARE Contador INT;
  DECLARE CuCategoria1 CURSOR FOR SELECT Nombre, Categoria, Ventas
  FROM RepuestosHard.Empleados WHERE Categoria = CuCategoria;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimaFila = 1;
  SET UltimaFila = 0;
  SET Contador = 0;
  OPEN CuCategoria1;
  Etiqueta: WHILE (UltimaFila = 0) DO
    FETCH CuCategoria1 INTO PNombre, PCategoria, PVentas;
    SET Contador = Contador + 1;
  END WHILE Etiqueta;
  CLOSE CuCategoria1;
  SELECT Contador;
END$
delimiter ;
CALL PrCursorCategoriaWhile('Director General');
```

La estructura WHILE es la más utilizada, ya que la condición se evalúa antes de leer un registro del cursor.

Mismo procedimiento con **WHILE**

```
DROP PROCEDURE PrCursorCategoria;
delimiter $
CREATE PROCEDURE PrCursorCategoria(CuCategoria CHAR(30))
BEGIN
   DECLARE PNombre VARCHAR(30);
   DECLARE PCategoria VARCHAR(30);
   DECLARE PVentas DOUBLE;
   DECLARE UltimaFila BOOL;
   DECLARE Contador INT;
   DECLARE CuCategoria CURSOR FOR SELECT Nombre, Categoria, Ventas
   FROM Empleados WHERE Categoria = CuCategoria;
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimaFila = 1;
   SET UltimaFila = 0;
   SET Contador = 0;
   OPEN CuCategoria;
   Etiqueta: WHILE UltimaFila != 1 DO
         FETCH CuCategoria INTO PNombre, PCategoria, PVentas;
         INSERT INTO empleadoscategoria (nombre, categoria, ventas)
         VALUES (PNombre, PCategoria, PVentas);
         SET Contador = Contador + 1;
   END WHILE;
   CLOSE CuCategoria;
   SELECT Contador;
END$
delimiter ;
CALL PrCursorCategoria('Representante');
select * from empleadoscategoria;
```

Ejemplo en el que se obtienen y se insertan en una tabla el número de pedidos de cada cliente y el total vendido a cada cliente.

Creamos la tabla

```
CREATE TABLE PedidosClientes(
Id INT PRIMARY KEY AUTO_INCREMENT,
CodCliente CHAR(5),
Nombre VARCHAR(50),
NumeroPedidos INT,
TotalImporte DOUBLE);
```

Creamos el procedimiento utilizando cursores

```
DROP PROCEDURE IF EXISTS PrPedidosCliente;
delimiter $
CREATE PROCEDURE PrPedidosCliente()
```

```
BEGIN
  DECLARE PCodCliente CHAR(5);
  DECLARE PNombre VARCHAR(30);
  DECLARE PTotalImporte DOUBLE DEFAULT 0;
  DECLARE PContador INT DEFAULT 0;
  DECLARE UltimaFila BOOL DEFAULT 0;
  DECLARE CuClientes CURSOR FOR SELECT CodCliente, Nombre FROM
EmpresasAbc.Clientes;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimaFila = 1;
  OPEN CuClientes;
  WHILE (UltimaFila = 0) DO
    IF UltimaFila = 0 THEN
        FETCH CuClientes INTO PCodCliente, PNombre;
        SELECT COUNT(*), SUM(Importe) INTO PContador, PTotalImporte
        FROM Pedidos JOIN LineasPedido USING (CodPedido)
        WHERE CodCliente = PCodCliente;
        INSERT INTO PedidosClientes (CodCliente, Nombre,
NumeroPedidos, TotalImporte) VALUES
    (PCodCliente, PNombre, PContador, PTotalImporte);
    END IF;
END WHILE ;
CLOSE CuClientes;
END$
```

```
delimiter ;
```

Utilizamos el IF UltimaFila = 0 para que no repita la inserción del último registro

Ejecutamos el procedimiento y comprobamos resultados

```
CALL PrPedidosCliente();
```

```
SELECT * from pedidosclientes;
```

Se pueden utilizar cursores anidados

Creamos la tabla

```
DROP TABLE NumeroPedidos;
CREATE TABLE NumeroPedidos(
id INT PRIMARY KEY AUTO_INCREMENT,
cliente VARCHAR(50),
numPedidos INT);
```

Creamos el procedimiento

```
DROP PROCEDURE IF EXISTS PrPedidosCliente01;
delimiter $
CREATE PROCEDURE PrPedidosCliente01() READS SQL DATA
BEGIN
DECLARE PCodCliente CHAR(5);
DECLARE PNombre VARCHAR(30);
DECLARE PCodPedido INT;
DECLARE PContador INT DEFAULT 0;
DECLARE UltimaFila BOOL DEFAULT 0;
DECLARE CuClientes CURSOR FOR SELECT CodCliente, Nombre FROM
Clientes;
DECLARE CuPedidos CURSOR FOR SELECT CodPedido FROM Pedidos WHERE
CodCliente = PCodCliente;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimaFila = 1;
OPEN CuClientes;
     EtiquetaClientes: WHILE (UltimaFila = 0) DO
          FETCH CuClientes INTO PCodCliente, PNombre;
          IF UltimaFila = 1 THEN
               LEAVE EtiquetaClientes;
          END IF;
          OPEN CuPedidos;
               SET UltimaFila = 0;
               SET PContador = 0;
               EtiquetaPedidos: WHILE (UltimaFila = 0) DO
                    FETCH CuPedidos INTO PCodPedido;
                    IF UltimaFila = 1 THEN
                         LEAVE EtiquetaPedidos;
                    END IF;
                    SET PContador = PContador + 1;
               END WHILE EtiquetaPedidos;
          CLOSE CuPedidos;
          SET UltimaFila = 0;
          INSERT INTO NumeroPedidos (Cliente, NumPedidos) VALUES
(PNombre, PContador);
     END WHILE EtiquetaClientes;
CLOSE CuClientes;
END$
delimiter ;

CALL PrPedidosCliente01();

SELECT * FROM NumeroPedidos;
```

Para cada cliente de la tabla Clientes se obtiene un cursor para los pedidos, que se utiliza para contar el número de pedidos y devolver el resultado usando un INSERT para cada cliente.

Lanzadores de Eventos. TRIGGERS

Se han incorporado desde la versión 5.0. de MySQL. Los Triggers o disparadores son un tipo especial de rutinas almacenadas que se activa o ejecuta cuando en una tabla ocurre un evento del tipo INSERT, UPDATE O DELETE. Estos eventos que modifican los datos dentro de la tabla a la que está asociado el trigger y pueden ser disparados antes (BEFORE) y/o después (AFTER) de que la fila es modificada.

Sólo puede haber un TRIGGER para una acción y una tabla.

Las instrucciones para gestionar TRIGGERS son CREATE TRIGGER, SHOW TRIGGER y DROP TRIGGER.

Crear TRIGGER

```
CREATE TRIGGER NombreDisparador MomentoDisparador EventoDisparador
ON NombreTabla FOR EACH ROW SentenciaDisparador
```

MomentoDisparador:

Es el momento en que el disparador entra en acción. Puede tomar los valores BEFORE (antes) o AFTER (después), para indicar que el disparador se ejecuta antes o después que la sentencia que lo activa.

EventoDisparador

Indica la clase de sentencia que activa el disparador. Puede ser INSERT, UPDATE o DELETE. Por ejemplo, un disparador BEFORE para sentencia INSERT se puede utilizar para validar los valores a insertar.

FOR EACH ROW:

Hace referencia a las acciones a llevar a cabo sobre cada fila de la tabla indicada.

SentenciaDisparador

Es la sentencia que se ejecuta cuando se activa el disparador. Si se desean ejecutar múltiples sentencias deben codificarse dentro de un bloque BEGIN…END, el constructor de sentencias compuestas. Lo que permite que se puedan utilizar las mismas sentencias que los procedimientos y funciones.

Las columnas de la tabla asociada con el disparador pueden referenciarse empleando los alias **OLD** y **NEW**, que se refieren respectivamente a los valores de las columnas antes y después de que la sentencia fue procesada.

La sintaxis NEW.nombrecolumna se utiliza en el trigger para hacer referencia a las columnas de la nueva fila que queremos insertar o en una fila existente después actualizar. Se utilizan en un Trigger INSERT o UPDATE.

Se utiliza OLD.nombrecolumna para hacer referencia a las columnas de la fila original que va a ser eliminada o actualizada por un Trigger DELETE o UPDATE.

Las palabras claves OLD y NEW permiten acceder a columnas en los registros afectados por el disparador. En un disparador para INSERT, solamente puede utilizarse NEW.NombreColumna ya que no hay una versión anterior del registro. En un disparador DELETE sólo puede emplearse OLD.nombreColumna porque no hay un registro nuevo. En un disparador UPDATE se puede emplear OLD.NombreColumna para referirse a las columnas de un registro antes de ser actualizado y NEWNombreColumna para referirse a las columnas del registro después de actualizarlo.

Los Triggers pueden hacer referencia a variables de usuario de forma que se pueda definir una variable dentro de un trigger para pasar un resultado fuera del trigger. Esto nos permite verificar los efectos del trigger.

Ejemplo de un Trigger BEFORE que realiza las siguientes acciones:

El lanzador comprueba si se intenta insertar un valor negativo en la columna de enteros y los convertirá en ceros.

El trigger proporcionará automáticamente un valor NOW() para la columna DATETIME. Esto nos permite eliminar la limitación de que el valor por defecto de una columna tiene que ser una constante e implementa el método de iniciación automática de tipo TIMESTAMP para la columna DATETIME.

```
CREATE TABLE TablaN
(i INT, fe DATETIME);

DELIMITER $
CREATE TRIGGER TInsertar BEFORE INSERT ON TablaN
FOR EACH ROW
BEGIN
     SET NEW.fe = CURRENT_TIMESTAMP;
     IF new.i < 0 THEN
          SET NEW.i = 0;
     END IF;
END $

DELIMITER ;

INSERT INTO TablaN (i) VALUES(-2), (10), (2);

SELECT * FROM TablaN;
```

Dentro de los Triggers se puede llamar a un procedimiento (procedures).

Ejemplo de un Trigger vamos a crear una tabla para el registro de todas las ventas, un procedimiento para fijar el porcentaje de comisión para los empleados basada en las ventas, y un trigger que llama al procedimiento y fija la comisión después de la inserción a la tabla.

```
/* Creamos la tabla*/
CREATE TABLE IF NOT EXISTS Ventas
(id INT NOT NULL AUTO_INCREMENT,
producto VARCHAR(10),
ventas DECIMAL(9,2),
empleado VARCHAR(10),
comision DECIMAL(7,2),
PRIMARY KEY(id));

/* Creamos el Procedimiento*/
DROP PROCEDURE IF EXISTS PComision;
DELIMITER $
CREATE PROCEDURE PComision(Ventas DECIMAL(9,2))
BEGIN
     SET @comi := Ventas/10;
END$

/* Creamos el Trigger*/
DELIMITER $
CREATE TRIGGER TVentasComision BEFORE INSERT ON Ventas
FOR EACH ROW
BEGIN
     CALL PComision(NEW.ventas);
     SET NEW.Comision = @comi;
END$
DELIMITER ;

/*Insertamos un registro y se ejecuta el TRIGGER */
INSERT INTO ventas (producto, ventas, empleado) VALUES
('Peras', 3000, 'Maria'),
('Limones', 2500, 'Beatriz');

SELECT * FROM ventas;
```

Consultar TRIGGER

Para obtener información sobre los Triggers se puede utilizar la sentencia SHOW TRIGGERS, nos da una lista de todos los triggers que hayamos creado.

```
SHOW TRIGGERS [{FROM | IN NbBaseDatos] [LIKE ‘patron’ | WHERE expression]

SHOW TRIGGERS IN ud14procedimientos;

SHOW TRIGGERS FROM ud14procedimientos;

SHOW TRIGGERS IN ud14procedimientos LIKE 'T%';
```

Para obtener una información más detallada podemos utilizar la tabla INFORMATION_SCHEMA.TRIGGERS.

```
SELECT * FROM INFORMATION_SCHEMA.TRIGGERS;

SELECT trigger_name, action_statement FROM
INFORMATION_SCHEMA.TRIGGERS WHERE trigger_name = 'TVentasComision';
```

Eliminación TRIGGER

Para eliminar el disparador o Trigger se puede utilizar la sentencia DROP TRIGGERS. El nombre del disparador debe incluir el nombre de la tabla.

```
DROP TRIGGERS [IF EXISTS] [NbTabla.]NbTriggers;
```

Uso de TRIGGER

Aunque su uso puede ser muy variado y depende del tipo de aplicación o base de datos con la que trabajemos, se puede hacer una clasificación más o menos general.

Control de sesiones

En algunos casos nos puede interesar guardar ciertos valores en variables de sesión creadas por el usuario que al final nos permiten ver un resumen de lo realizado en dicha sesión.

Ejemplo necesitamos saber el importe total de las ventas insertadas en una sesión. Para ello vamos a crear un TRIGGER que acumule las ventas cada vez que insertemos un nuevo registro en la tabla Ventas.

```
/* Creamos la tabla*/

CREATE TABLE Ventas1
(id INT NOT NULL AUTO_INCREMENT,
producto VARCHAR(10),
ventas DECIMAL(9,2),
empleado VARCHAR(10),
comision DECIMAL(7,2),
PRIMARY KEY(id));

/* Creamos el TRIGGER */

DELIMITER $

CREATE TRIGGER TInsertarVentas BEFORE INSERT ON Ventas1
FOR EACH ROW SET @TotalVentas = @TotalVentas + New.Ventas;
```

En el ejemplo vemos como antes de insertar uno o varios registros en la tabla Ventas1 se acumulan las Ventas en @TotalVentas. Para poder utilizarlo hay que establecer el valor de dicha variable acumulador a 0, ejecutar una o varias sentencias de inserción y comprobar cuánto vale la variable.

```
SET @TotalVentas = 0;
INSERT INTO Ventas1 (Producto, Ventas, Empleado)VALUES
('Peras', 3000, 'Maria'), ('Limones', 2500, 'Beatriz');
SELECT @TotalVentas;
```

Control de valores de entrada

Los disparadores o triggers nos permiten controlar los valores insertado o actualizados en las tablas.

En el ejemplo vemos a crear un disparador para Ventas1 para UPDATE que verifica los valores utilizados para actualizar cada columna y modifica el valor para que se encuentre en un rango de 0 a 100. Esto debe hacerse en un TRIGGER BEFORE porque los valores deben verificarse antes de utilizarse para actualizar el registro.

```
/* Creamos el TRIGGER de comprobación*/
DELIMITER $
CREATE TRIGGER TComprobacionVentas BEFORE UPDATE ON Ventas1
FOR EACH ROW
BEGIN
     IF NEW.Ventas < 0 THEN SET New.Ventas = 0;
     ELSEIF New.Ventas > 100 THEN SET NEW.Ventas = 100;
     END IF;
END $
```

Cada vez que se actualice la tabla se controlará el valor de las ventas para que no puedan ser negativas ni mayores que 100.

```
UPDATE Ventas1 SET Ventas = -5 WHERE Producto = 'Limones';
SELECT * FROM Ventas1;
UPDATE Ventas1 SET Ventas = 150 WHERE Producto = 'Limones';
SELECT * FROM Ventas1;
```

Mantenimiento de campos derivados

Los TRIGGERS nos permiten controlar los campos derivados o redundantes, es decir, campos que puedan calcularse a partir de otros

Ejemplo: Vamos a suponer que tenemos una tabla de TotalesVentas, con las columnas Nombreproducto y Total. Cada vez que insertemos un registro en la tabla Ventas2, se actualiza la tabla TotalesVentas.

```
/*Creamos las tablas */

CREATE TABLE Ventas2
(id INT NOT NULL AUTO_INCREMENT,
producto VARCHAR(10),
ventas DECIMAL(9,2),
empleado VARCHAR(10),
comision DECIMAL(7,2),
PRIMARY KEY(id));

CREATE TABLE TotalesVentas(
NombreProducto VARCHAR(10),
Total DECIMAL(9,2),
PRIMARY KEY(NombreProducto));

INSERT INTO TotalesVentas VALUES
('Peras', 0), ('Limones', 0);
```

Creamos el trigger

```
DROP TRIGGER TrCamposDerivados;

DELIMITER $

CREATE TRIGGER TrCamposDerivados AFTER INSERT ON Ventas2
FOR EACH ROW
BEGIN
     INSERT INTO TOTALESVENTAS (NOMBREPRODUCTO, TOTAL) VALUES
     (NEW.PRODUCTO, NEW.VENTAS);
END $

DELIMITER ;
```

Insertamos un valor en la tabla y comprobamos

```
INSERT INTO VENTAS2 (PRODUCTO, VENTAS, EMPLEADO, COMISION) VALUES
('Peras', 5000, 'Marta', 500);

SELECT * FROM VENTAS2;

SELECT * FROM TOTALESVENTAS;
```

Volvemos a insertar un producto existente se produce un error de clave duplicada, en el caso de que el producto ya existe debemos de realizar una actualización, no una inserción.

```
INSERT INTO Ventas2 (Producto, Ventas, Empleado)VALUES
('Peras', 3000, 'Maria'), ('Limones', 2500, 'Beatriz');

SELECT * FROM TotalesVentas;

DROP FUNCTION FuExisteProducto;
DELIMITER $
CREATE FUNCTION FuExisteProducto(NbPr VARCHAR(10)) RETURNS BOOL
```

```
BEGIN
  DECLARE UltimoRegistro BOOL DEFAULT 0;
  DECLARE ExisteRegistro BOOL DEFAULT 0;
  DECLARE NbProducto VARCHAR(10);
   DECLARE CuExisteRegistro CURSOR FOR SELECT NombreProducto FROM
TotalesVentas;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET UltimoRegistro = 1;
  OPEN CuExisteRegistro;
    WHILE UltimoRegistro = 0 AND ExisteRegistro = 0 DO
      FETCH CuExisteRegistro INTO NbProducto;
      IF NbProducto = NbPr THEN SET ExisteRegistro = 1;
      END IF;
    END WHILE;
  CLOSE CuExisteRegistro;
  RETURN ExisteRegistro;
END $
DELIMITER ;

SELECT FuExisteProducto('mANZANAS');


DELIMITER $
DROP TRIGGER TrCamposDerivados $
CREATE TRIGGER TrCamposDerivados AFTER INSERT ON Ventas2 FOR EACH
ROW
BEGIN
  IF FuExisteProducto(NEW.Producto) = 0 THEN
     INSERT INTO TotalesVentas (NombreProducto, Total) VALUES
     (New.Producto, NEW.Ventas);
  ELSE
     UPDATE TotalesVentas SET Total = Total + NEW.Ventas
     WHERE NombreProducto = New.Producto;
  END IF;
END $
DELIMITER ;

INSERT INTO VENTAS2 (PRODUCTO, VENTAS, EMPLEADO, COMISION) VALUES
('Peras', 15000, 'Marta', 500);
SELECT * FROM VENTAS2;
SELECT * FROM TOTALESVENTAS;
INSERT INTO VENTAS2 (PRODUCTO, VENTAS, EMPLEADO, COMISION) VALUES
('Naranjas', 5000, 'Pedro', 500);
```

Estadísticas

Se pueden registrar estadísticas de operaciones o valores de las bases de datos en tiempo real usando triggers.

Por ejemplo, Podemos contabilizar cuantos clientes se hacen cada mes en una tabla ClientesMes(Mes, Anho, Contador).

```
/*Creamos la tabla*/
```

```
CREATE TABLE ClientesMes (
Mes Integer,
Anho Integer,
Contador Integer)ENGINE = InnoDB;

/* Creamos la funcion FExiste que devuelve 1 ó 0 si existe o
no el registro para un mes de un año*/

DELIMITER $

DROP FUNCTION FExiste $

CREATE FUNCTION FExiste() RETURNS BOOL
BEGIN
DECLARE FUltimoRegistro BOOL;
DECLARE FExisteRegistro BOOL;
DECLARE CuMes INT;
DECLARE CuAnho INT;
DECLARE CuFExiste CURSOR FOR SELECT Mes, Anho FROM ClientesMes;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET FUltimoRegistro = 1;
SET FUltimoRegistro = 0;
SET FExisteRegistro = 0;
OPEN CuFExiste;
WHILE FUltimoRegistro = 0 DO FETCH CuFExiste INTO CuMes, CuAnho;
      IF CuMes = MONTH(CURDATE()) AND CuAnho = YEAR(CURDATE()) THEN
            SET FExisteRegistro = 1;
            SET FUltimoRegistro = 1;
      END IF;
END WHILE;
CLOSE CuFExiste;
RETURN FExisteRegistro;
END $

DELIMITER ;

/* Creamos el TRIGGER */

DELIMITER $

CREATE TRIGGER TInsertarClientes AFTER INSERT ON Clientes
FOR EACH ROW
BEGIN
IF FExiste() = 0 THEN
INSERT      INTO      ClientesMes      (Mes,      Anho,      Contador)
VALUES(MONTH(CURDATE()), YEAR(CURDATE()), 1);
ELSE  UPDATE  ClientesMes  SET  Contador  =  Contador  +  1  WHERE
MONTH(CURDATE()) = Mes AND YEAR(CURDATE()) = Anho;
END IF;
END $

DELIMITER ;

/* Insertamos un cliente */

INSERT   INTO   Clientes   (CodCliente,   Nombre,   CodRepCliente,
LimiteCredito) VALUES ('625', 'Maria Rodriguez', '101', 5000);
```

```
/* Comprobamos el resultado */

SELECT * FROM clientesmes c;
```

Registro y auditoría

Cuando muchos usuarios acceden a las bases de datos puede ser que el registro de log no sea suficiente para conocer quién ha hecho que operación y a qué hora. Para ello existen soluciones que permiten filtrar los ficheros de registro para obtener la información que necesitamos. Pero también podemos utilizar triggers que nos facilitan la tarea. Podemos crear un trigger a una tabla que se dispare después (AFTER) de una sentencia INSERT; DELETE o UPDATE que guarde los valores del registro, así como alguna otra información de utilidad en una tabla.

Por ejemplo, Queremos conocer quiénes han sido los usuarios que han realizado actualizaciones en la tabla Ventas3, la fecha y la hora.

```
/* Creamos las tablas */

CREATE TABLE Ventas3
(id INT NOT NULL AUTO_INCREMENT,
producto VARCHAR(10),
ventas DECIMAL(9,2),
empleado VARCHAR(10),
comision DECIMAL(7,2),
PRIMARY KEY(id));

/* Creamos la table de Usuarios e Ventas */

CREATE TABLE UsuariosVentas
(usuario VARCHAR(50),
Fecha DATETIME);

/*Creamos el TRIGGER */

DELIMITER $

CREATE TRIGGER TInsertarVentasControlEstadísticas AFTER INSERT ON
Ventas3
FOR EACH ROW
BEGIN
INSERT UsuariosVentas (usuario, Fecha) VALUES (CURRENT_USER(),
CURRENT_DATE);
END $

DELIMITER ;

/* Insertamos valores */

INSERT INTO Ventas3 (Producto, Ventas, Empleado)VALUES
('Peras', 3000, 'Maria'), ('Limones', 2500, 'Beatriz');

/*Comprobamos */

SELECT * FROM UsuariosVentas;
```

Diferencias entre TRIGGERS y PROCEDIMIENTOS

Hay una diferencia muy básica entre triggers y un procedures: Los triggers son procedimientos que se ejecutan automáticamente, cuando se produce un evento sobre el que se quiere trabajar. Existen tres tipos de eventos que pueden disparar un trigger: INSERT, DELETE y UPDATE.

El trigger se programa para realizar una tarea determinada que se deba hacer siempre que se produzca el evento. No requiere intervención humana o programática y no se puede detener.

Tiene algunas características:

1) No recibe parámetros de entrada o salida.
2) Los únicos valores de entrada son los correspondientes a los de las columnas que se insertan, y sólo son accesibles por medio de ciertas pseudovariables (NEW y OLD).
3) No se puede ejecutar una operación INSERT/UPDATE/DELETE sobre la misma tabla donde el TRIGGER se está ejecutando.
4) No se puede ejecutar una tarea sobre otra tabla, si la segunda tiene un trigger que afecte a la tabla del primer trigger en ejecución (circularidad).
5) No se puede invocar procedures desde un TRIGGER.
6) No se puede invocar un SELECT que devuelva una tabla resultado en el TRIGGER.

Un stored procedure es un procedimiento almacenado que debe ser invocado para ejecutarse.

1) Puede recibir parámetros y devolver parámetros.
2) Puede manejar cualquier tabla, realizar operaciones con ellas y realizar iteraciones de lectura/escritura.
3) Puede devolver una tabla como resultado. también valores dentro de los parámetros del prototipo si los mismos son también de salida.
4) Existen en la base donde se crean, pero no dependen de ninguna tabla.
5) Pueden aceptar recursividad (pero no es recomendable).

Eventos

En MySQL los eventos son tareas que se ejecutan de acuerdo a un horario. Por lo tanto, por lo que también se conocen como eventos programados o triggers temporales, ya que conceptualmente son similares y se diferencian en que los triggers se activan con un evento sobre la base de datos, mientras que los eventos según 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.

Se pueden distinguir 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 **DISABLE** si queremos imposibilitar la activación (ponerla a ON) en tiempo de ejecución.

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

```
SHOW PROCESSLIST;
```

Si ***global_event_scheduler*** no se ha establecido en DISABLED podemos activar el programador con el siguiente comando:

```
SET GLOBAL EVENT_SCHEDULER = ON;
```

Los comandos para la gestión de eventos son: **CREATE EVENT, ALTER EVENT, SHOW EVENT, y DROP EVENT.**

Creación de Eventos

Un evento se define mediante la instrucción CREATE EVENT

```
CREATE [DEFINER = { user | CURRENT_USER}] EVENT [IF NOT EXISTS] NbEvento
ON SCHEDULE schedule
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comentario'
DO cuerpo del evento;
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 }
```

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

La cláusula ON SCHEDULE permite establecer cómo y cuándo 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.

Cuerpo del evento es el contenido del evento que se va a ejecutar.

Las cláusulas COMPLETION permiten mantener el evento aunque haya expirado mientras DISABLE permita 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, suponemos que tenemos una tabla Cuentas en la que se va a establecer una bonificación de 100 € a las cuentas dadas de alta en el intervalo de un mes.

```
/* Creamos la tabla */

CREATE TABLE Cuentas1
( NumeroCuenta INT PRIMARY KEY,
Saldo INT,
FechaCreacion DATE)ENGINE = InnoDB;

/* Insertamos registros */

INSERT INTO Cuentas1 (NumeroCuenta, Saldo, FechaCreacion) VALUES
(1, 1000, '2018/01/23'),
(2, 1100, '2018/01/25'),
(3, 1200, '2018/02/05'),
(4, 1350, '2018/02/24'),
(5, 1400, '2018/03/12'),
(6, 1500, '2018/03/25'),
(7, 2400, '2018/04/07'),
(8, 2400, '2018/04/07'),
(9,. 1400, '2018/04/12'),
(10, 1500, '2018/04/25'),
(11, 2400, '2018/05/07');

/*Comprobamos los datos */

SELECT * FROM Cuentas;

SET GLOBAL EVENT_SCHEDULER = ON;

/* Creamos el evento */

DELIMITER $

CREATE EVENT EvBonificacion
ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 2 MINUTE
DO
UPDATE Cuentas SET Saldo = Saldo + 100 WHERE FechaCreacion BETWEEN
SUBDATE(NOW(), INTERVAL 1 MONTH) AND NOW();
$

DELIMITER ;

/* Pasado el tiempo Comprobamos los datos */

SELECT * FROM Cuentas;
```

Ejemplo, Cada mes se eliminan de la tabla Cuentas, las cuentas que tengan la fecha de creación hace más de un mes. Previamente al borrado se almacenan en una tabla de históricos. El evento comienza el 01/01/2016

```
/* Creamos la tabla HistoricoCuentas*/

CREATE TABLE HistoricoCuentas
( NumeroCuenta INT PRIMARY KEY,
Saldo INT,
FechaCreacion DATE)ENGINE = InnoDB;

SET GLOBAL EVENT_SCHEDULER = ON;

/* Creamos el evento */

DELIMITER $

DROP EVENT ArchivarCuentas$

CREATE EVENT ArchivarCuentas ON SCHEDULE EVERY 1 MINUTE STARTS '2016-01-01 00:00:00' ENABLE
DO
BEGIN
INSERT INTO HistoricoCuentas (NumeroCuenta, Saldo, FechaCreacion)
SELECT * FROM Cuentas
WHERE FechaCreacion < DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
DELETE FROM Cuentas WHERE FechaCreacion < DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
END;

DELIMITER ;
```

Modificación de Eventos

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

```
ALTER [DEFINER = { user | CURRENT_USER}] EVENT NbEvento
ON SCHEDULE schedule]
[ON COMPLETION [NOT] PRESERVE]
[RENAME TO NuevoNombreEvento]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comentario'
[DO cuerpo del evento]
```

Consulta de Eventos

Para la información asociada a un evento usamos SHOW EVENT, con el formato:

```
SHOW EVENT [{FROM | IN ] NombreBasedeDatos]
[LIKE 'patrón' | WHERE expression]
```

Muestra información de los eventos asociados a una base de datos y filtrado según el patrón que determinemos o una clausula WHERE.

Ejemplos de Procedimientos almacenados

Crear un procedimiento que nos diga cuantos empleados han sido contratados en un determinado año.

```
delimiter $

CREATE PROCEDURE ContarNacidosAnho(AnhoNacimiento INT, OUT contador INT)

BEGIN
DECLARE c CURSOR FOR SELECT COUNT(*) FROM Empleados WHERE AnhoNacimiento = YEAR(FecNacimiento);
OPEN c;
FETCH c INTO contador;
Select Contador;
CLOSE c;
END$

delimiter ;
```

Ejecutamos el procedimiento

```
CALL ContarNacidosAnho(1958, @count);

SELECT @count;

CALL ContarNacidosAnho(1958, @count);

SELECT @count;
```

Crear un procedimiento para insertar datos en la tabla empleados y genera un mensaje de error en caso de que la clave del empleado exista y otro mensaje si la inserción ha sido correcta.

```
Delimiter $$

CREATE PROCEDURE ProcedimientoInsertarEmpleados(NuevoCodigo CHAR(5), NuevoNombre VARCHAR(30), NuevaCategoria VARCHAR(30))
MODIFIES SQL DATA
BEGIN
DECLARE ClaveDuplicada INT DEFAULT 0;
BEGIN
     DECLARE EXIT HANDLER FOR 1062 /* Duplicate key*/ SET ClaveDuplicada = 1;
     INSERT INTO Empleados (CodEmpleado, Nombre, Categoria) VALUES(NuevoCodigo,NuevoNombre,NuevaCategoria);
     SELECT CONCAT('Empleado ',NuevoNombre,' creado') as "Resultado";
     END;
     IF ClaveDuplicada = 1 THEN
     SELECT CONCAT('Error al Insertar ', NuevoNombre, ': Clave Duplicada') as "Resultado";
     END IF;
END$$

Delimiter ;
```

```
CALL ProcedimientoInsertarEmpleados('356', 'Ernesto Pereira', 'Director Financiero');
CALL ProcedimientoInsertarEmpleados('356', 'Ernesto Pereira', 'Director Financiero');
Delimiter $$

CREATE PROCEDURE ProcedimientoInsertarEmpleadosContinuar(NuevoCodigo CHAR(5), NuevoNombre VARCHAR(30), NuevaCategoria VARCHAR(30))
MODIFIES SQL DATA
BEGIN
DECLARE ClaveDuplicada INT DEFAULT 0;
BEGIN
DECLARE CONTINUE HANDLER FOR 1062 /* Duplicate key*/ SET ClaveDuplicada = 1;
INSERT INTO Empleados (CodEmpleado, Nombre, Categoria) VALUES(NuevoCodigo,NuevoNombre,NuevaCategoria);
END;
IF ClaveDuplicada = 1 THEN
SELECT CONCAT('Error al Insertar ', NuevoNombre, ': Clave Duplicada') as "Resultado";
ELSE
SELECT CONCAT('Empleado ',NuevoNombre,' creado') as "Resultado";
END IF;
END$$

CALL ProcedimientoInsertarEmpleadosContunuar('125', 'Juan Pedro Martinez', 'Representante')
```

Ejemplo de Cursor

```
DELIMITER $$

CREATE PROCEDURE ProcedimientoEmpleados (Codigo CHAR(5))
BEGIN
DECLARE PrCodigo CHAR(5);
DECLARE PrNombre VARCHAR(30);
DECLARE PrCategoria VARCHAR(30);
DECLARE PrContador INT;
DECLARE NoHayOficinas INT;
DECLARE CursorOficinas CURSOR FOR SELECT CodEmpleado, Nombre, Categoria FROM Empleados;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET NoHayOficinas = 1;
SET NoHayOficinas = 0;
OPEN CursorOficinas;
EtiquetaLoop:WHILE(NoHayOficinas = 0) DO
FETCH CursorOficinas INTO PrCodigo, PrNombre, PrCategoria;
IF NoHayOficinas = 1 THEN
LEAVE EtiquetaLoop;
END IF;
SET PrContador = PrContador + 1;
```

```
SELECT PrCodigo, PrNombre, PrCategoria;
END WHILE EtiquetaLoop;
CLOSE CursorOficinas;
SET NoHayOficinas = 0;
END$$

DELIMITER ;
```

Por ejemplo: Crear un procedimiento en el que si se produce un error, en este caso por clave duplicado, el procedimiento continúa su ejecución

```
delimiter $
CREATE PROCEDURE PHandlerDemostracion ()
BEGIN
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
SET @x = 1;
INSERT INTO Numeros VALUES (3);
SET @x = 2;
INSERT INTO Numeros VALUES (3);
SET @x = 3;
END$
delimiter ;
CALL HandlerDemostracion();
SELECT @x
```

Ejemplo: Crear un procedimiento en el que si se produce un error, en este caso por clave duplicado, el procedimiento detiene su ejecución

```
delimiter $

CREATE PROCEDURE PHandlerDemostracion1 ()
BEGIN
DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
SET @x = 1;
INSERT INTO Numeros VALUES (2);
SET @x = 2;
INSERT INTO Numeros VALUES (2);
SET @x = 3;
END$
delimiter ;
CALL HandlerDemostracion1();

SELECT @x
```