TEMA 5: AUTOMATIZACIÓN DE TAREAS II.

5.1. Excepciones.

Se considera una excepción cualquier error que pueda ocurrir a lo largo de la ejecución de un programa. Si una excepción no es tratada, provocará la terminación anormal del programa en el que se produzca. Sin embargo, si lo que se desea es que se informe del error o excepción al usuario y que se pueda continuar con la ejecución del programa, entonces será necesario tratar la excepción correspondiente mediante un manejador de errores o *handler*.

Veamos un ejemplo al respecto. Creemos un procedimiento que se encargue de añadir un nuevo pedido a la tabla *Pedido* de nuestra base de datos *Pedidos*. Este procedimiento recibirá como parámetros la referencia del pedido que se desea añadir y su fecha:

```
mysql> create procedure InsertarPedido (refe char(5), fecha date)
    -> begin
    ->     insert into Pedido values (refe, fecha);
    -> end;//
Query OK, 0 rows affected (0.00 sec)
```

Ejecutemos a continuación este procedimiento dos veces:

```
mysql> call InsertarPedido ('P0005', '2014-11-09');//
Query OK, 1 row affected (0.05 sec)

mysql> call InsertarPedido ('P0005', '2014-11-09');//
ERROR 1062 (23000): Duplicate entry 'P0005' for key 'PRIMARY'
```

La primera vez que ejecutamos el procedimiento funciona correctamente, por lo que nos añade el pedido con referencia P0005 a la tabla *Pedido*. En la segunda llamada al procedimiento, como intentamos añadir de nuevo un pedido con la misma referencia, se produce un error o excepción y el programa finaliza anormalmente mostrándonos una descripción del error en pantalla. Si queremos que cuando se produzca la excepción, en lugar de terminar anormalmente el programa y mostrarnos una descripción del error predeterminada en inglés, el programa termine normalmente y muestre un mensaje confeccionado por nosotros, entonces deberemos tratar la excepción creando un manejador o *handler* para ella.

Podemos crear un manejador para la excepción 1062, que es la que se produce cuando se incumple una restricción de clave primaria o de unicidad. Vamos a borrar el procedimiento que acabamos de crear y crear uno con el mismo nombre que en caso de que se produzca la excepción 1062 muestre un mensaje de aviso al usuario indicando que ya existe un pedido con la referencia pasada como parámetro. En este procedimiento crearemos un manejador o *handler* para la excepción 1062, de manera que si se produce dicha excepción, se asigne a la

variable *duplicado* el valor 1. Esta es una variable booleana que debemos declarar anteriormente con valor inicial 0, ya que suponemos optimistamente que no se va a producir un duplicado. En caso de que se produzca dicha excepción, se asignará a la variable *duplicado* el valor 1. Por ello, en este caso, mostramos un mensaje de error al usuario. En caso contrario, indicaremos que el pedido ha sido añadido a la base de datos.

```
mysql> drop procedure InsertarPedido;//
Query OK, 0 rows affected (0.00 sec)

mysql> create procedure InsertarPedido (refe char(5), fecha date)
    -> begin
    -> declare duplicado bool default 0;
    -> declare continue handler for 1062 set duplicado = 1;
    -> insert into Pedido values (refe, fecha);
    -> if duplicado = 1 then
    ->     select concat ('Ya existe un pedido con la referencia ',refe) Error;
    -> else
    ->     select concat ('Añadido pedido con referencia ',refe) Mensaje;
    -> end if;
    -> end//
Query OK, 0 rows affected (0.00 sec)
```

Ahora, si llamamos a este procedimiento solicitando la inserción de un pedido con una referencia no repetida, no habrá ningún problema y no aparecerá ningún mensaje de error. En caso contrario, se mostrará el mensaje de error diseñado por nosotros. Probémoslo a continuación:

```
mysql> call InsertarPedido ('P0006', '2014-11-11');//
+--------------------------------------+
| Mensaje                              |
+--------------------------------------+
| Añadido pedido con referencia P0006  |
+--------------------------------------+
1 row in set (0.04 sec)

Query OK, 0 rows affected (0.05 sec)

mysql> call InsertarPedido ('P0005', '2014-11-14');//
+-------------------------------------------+
| Error                                     |
+-------------------------------------------+
| Ya existe un pedido con la referencia P0005 |
+-------------------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

La instrucción empleada para crear un manejador o *handler* de excepciones presenta el siguiente formato:

```
DECLARE {CONTINUE|EXIT} HANDLER FOR
Condición acciones_manejador;

Condición:
{SQLSTATE 'valor_estado_SQL' | código_error_MySQL | NOT FOUND}
```

Como se puede observar, después de la palabra DECLARE se debe indicar si se trata de un manejador de tipo CONTINUE o EXIT:

- Con un manejador del tipo CONTINUE, tras el levantamiento de la excepción, la ejecución del programa continúa en la siguiente instrucción.
- Con un manejador del tipo EXIT, después de producirse la excepción, la ejecución del bloque en el que se ha producido la excepción finaliza, pasando la ejecución al bloque externo dentro del mismo programa, o bien, si es el bloque principal, se devuelve la ejecución al programa externo que invocó al procedimiento o función.

Se debe indicar después HANDLER FOR y a continuación la condición por la que se produce la excepción, que puede tomar varios formatos, entre ellos:

- SQLSTATE 'valor_estado_SQL': Es un código alfanumérico de 5 caracteres que lleva asociado cada posible error en MySQL.
- código_error_MySQL: Es un código numérico de 4 cifras que lleva asociado cada posible error en MySQL.
- NOT FOUND: Hace referencia a estados SQL que comienzan por '02'.

Se muestran en la siguiente tabla algunas de las excepciones que se pueden producir en MySQL, indicando por cada una de ellas su código de error, su valor de estado y una descripción. Están resaltadas en amarillo las excepciones más relevantes.

Código de error	Valor de estado	Descripción
1005	HY000	No se puede crear la tabla indicada
1006	HY000	No se puede crear la base de datos indicada
1007	HY000	No se puede crear la base de datos indicada. La base de datos ya existe.
1008	HY000	No se puede borrar la base de datos indicada. La base de datos no existe.
1009	HY000	Error eliminando base de datos.
1040	08004	Demasiadas conexiones
1044	42000	Acceso denegado para el usuario indicado sobre la base de datos indicada.
1045	28000	Acceso denegado para el usuario indicado con la contraseña indicada.
1046	3D000	No seleccionada base de datos.
1047	08S01	Comando desconocido
1048	23000	La columna indicada no puede tomar valor nulo
1049	42000	Base de datos desconocida
1050	42S01	La tabla indicada ya existe
1051	42S02	Tabla desconocida
1052	23000	Columna ambigua

1054	42S22	Columna desconocida en la tabla indicada
1055	42000	La columna indicada no está en GROUP BY
1056	42000	No se puede agrupar sobre el campo indicado
1057	42000	La misma sentencia contiene funciones de resumen y atributos
1062	23000	Entrada duplicada para el atributo clave indicado
1068	42000	Definida clave primaria duplicada
1090	42000	No se pueden eliminar todas las columnas con ALTER TABLE; use DROP TABLE
1102	42000	Nombre de base datos incorrecta
1103	42000	Nombre de tabla incorrecta
1106	42000	Nombre de procedimiento desconocido
1107	42000	Incorrecto número de parámetros para el procedimiento indicado.
1108	42000	Parámetros incorrectos para el procedimiento indicado.
1215	HY000	No se puede añadir una restricción de clave ajena.
1216	HY000	No se puede añadir o eliminar una fila hija porque falla una restricción de clave ajena
1217	HY000	No se puede añadir o eliminar una fila padre porque falla una restricción de clave ajena
1231	42000	A la variable indicada no se le puede asignar el valor indicado
1242	21000	La subconsulta devuelve más de una fila.
1280	42000	Nombre de índice indicado incorrecto.
1329	02000	Cero filas (ningún dato) recibido, seleccionado o procesado
1348	HY000	La columna indicada no es modificable
1451	23000	No se puede eliminar o modificar una fila padre porque falla una restricción de clave ajena
1452	23000	No se puede insertar o modificar una fila hija porque falla una restricción de clave ajena

En el ejemplo expuesto con anterioridad, en el que se creó el procedimiento *InsertarPedido*, se creó un manejador para la excepción con código de error 1062, excepción que se produce cuando se intenta asignar un valor duplicado a un atributo con valor único. También se podría haber especificado en la instrucción de declaración del manejador en vez del código de error 1062 el valor de estado '23000', que es el que corresponde a esta excepción.

Creemos otro ejemplo de procedimiento con manejo de excepciones. Se trata de un procedimiento que recibe el código de un artículo y nos mostrará su descripción en caso de que exista un artículo con el código recibido como parámetro. En caso de que no exista

ningún artículo con dicho código, se mostrará un mensaje como el siguiente: 'No existe ningún artículo con el código XXXXX'. Hemos de tener en cuenta que para obtener la descripción de un artículo a partir de su código tendremos que emplear una instrucción SELECT … INTO, de manera que si esta instrucción no nos devuelve ninguna fila por no haber ningún artículo con el código buscado, se producirá la excepción con código de error 1329 y valor de estado '02000'. Podremos especificar cualquiera de estos dos valores en la instrucción de declaración del manejador. Optamos por especificar el valor de estado. El procedimiento nos quedará como sigue:

```
mysql> create procedure MostrarDescri (codar char(5))
    -> begin
    -> declare encontrado bool default 1;
    -> declare descri varchar(30);
    -> declare continue handler for sqlstate '02000' set encontrado = 0;
    -> select DesArt into descri from Articulo where codart = codar;
    -> if encontrado = 0 then
    ->     select concat ('No existe ningún artículo con el código ', codar) ERROR;
    -> else
    ->     select descri Descripción;
    -> end if;
    -> end;
    -> //
Query OK, 0 rows affected (0.05 sec)
```

Ahora si llamamos a este procedimiento pasándole el código de un artículo existente, nos mostrará su descripción; en caso contrario, se nos mostrará el mensaje de error que hemos especificado. Probémoslo:

```
mysql> call MostrarDescri ('A0043');//
+----------------+
| Descripción    |
+----------------+
| Bolígrafo azul |
+----------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call MostrarDescri ('A0099');//
+-----------------------------------------------+
| ERROR                                         |
+-----------------------------------------------+
| No existe ningún artículo con el código A0099 |
+-----------------------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

5.2. Cursores.

Hasta ahora todas las consultas que hemos creado en nuestros procedimientos y funciones nos devolvían una sola fila y esto no era casualidad. Esto se debe a que hemos empleado lo que se llaman cursores implícitos. Y es que estos cursores permiten solo manejar una fila. De hecho, si en los programas que hemos creado hasta ahora una consulta nos devolviera varias filas, se produciría una excepción que, al no ser tratada, provocaría la terminación anormal del programa.

Por todo esto, si queremos ejecutar consultas que devuelvan varias filas, debemos usar cursores explícitos, que vamos a tratar a continuación.

Para poder utilizar un cursor explícito, al igual que para poder utilizar una variable, es necesario declararlo. Un cursor se declara de acuerdo con la siguiente sintaxis:

```
DECLARE Nombre_cursor CURSOR FOR Sentencia_select;
```

Por su parte, para utilizar un cursor, será necesario, en primer lugar, abrirlo. Para ello, se utilizará la siguiente instrucción:

```
OPEN Nombre_cursor;
```

Al abrir un cursor, se ejecuta la sentencia *select* que se asignó al mismo en la declaración y se almacenan los resultados de la ejecución en estructuras internas de memoria.

Para acceder a la información almacenada en el cursor, es decir, a los datos resultantes de la ejecución de la sentencia select, es necesario usar la instrucción siguiente:

```
FETCH Nombre_cursor into Lista_variables;
```

donde *Lista_variables* puede ser una sola variable o varias variables separadas por comas. Habrá que escribir tantas variables separadas por comas como elementos aparezcan en la cláusula *select*. Esta/s variable/s tendrá/n que estar previamente declarada/s.

La instrucción *fetch* recupera una de las filas y pasa automáticamente a la siguiente fila de las resultantes de la ejecución de la sentencia *select*. Si al ejecutar una orden *fetch*, esta no devuelve datos, o lo que es lo mismo, si una vez leídas todas las filas del cursor, se pretende leer otra fila, se produce el error o excepción NOT FOUND (no encontrado). Si este error o excepción no es tratado, se producirá la terminación anormal del subprograma ejecutado.

Una vez empleado el cursor, es necesario cerrarlo con la siguiente instrucción:

```
CLOSE Nombre_cursor;
```

Para probar todo esto comencemos creando una tabla *Pedido2* copia de *Pedido*, pero sin datos. Para ello, ejecutemos las siguientes órdenes:

```
mysql> create table pedido2 as select * from pedido;
Query OK, 4 rows affected (0.34 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> delete from pedido2;
Query OK, 4 rows affected (0.05 sec)
```

Creemos a continuación un procedimiento que muestre por cada pedido su referencia y fecha mediante el empleo de un cursor. Primero lo aplicaremos sobre la tabla *Pedido* y luego sobre la tabla *Pedido2*. Comenzaremos creando un procedimiento que recupere solo una de las filas de la tabla *Pedido*. Para ello, declararemos un cursor, a continuación lo abriremos, seleccionaremos una fila y mostraremos su contenido. Finalmente cerraremos el cursor.

```
mysql> delimiter //
mysql> Create procedure VerPedido()
    -> Begin
    -> Declare refe char(5);
    -> Declare fecha date;
    -> Declare c cursor for select refped, fecped from pedido;
    -> Open c;
    -> Fetch c into refe, fecha;
    -> Select concat ('Referencia: ', refe, ' Fecha: ', fecha) "Datos pedidos";
    -> Close c;
    -> End; //
Query OK, 0 rows affected (0.00 sec)
```

Si ejecutamos este procedimiento, obtendremos el siguiente resultado.

```
mysql> call VerPedido();//
+-----------------------------------+
| Datos pedidos                     |
+-----------------------------------+
| Referencia: P0001 Fecha: 2014-02-16 |
+-----------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)
```

Como podemos observar, nos muestra los datos del primer pedido almacenado en la tabla *Pedido*. Ahora eliminemos este procedimiento y creemos uno con el mismo nombre, pero que consulte la tabla *Pedido2*, copia de *Pedido* pero sin datos. Una vez creado, ejecutémoslo:

```
mysql> drop procedure VerPedido; //
Query OK, 0 rows affected (0.00 sec)

mysql> Create procedure VerPedido()
    -> Begin
    -> Declare refe char(5);
    -> Declare fecha date;
    -> Declare c cursor for select refped, fecped from pedido2;
    -> Open c;
    -> Fetch c into refe, fecha;
    -> Select concat ('Referencia: ', refe, ' Fecha: ', fecha) "Datos pedidos";
```

```
    -> Close c;
    -> End; //
Query OK, 0 rows affected (0.00 sec)

mysql> call VerPedido();//
ERROR 1329 (02000): No data - zero rows fetched, selected, or processed
```

Como podemos observar, el procedimiento ha finalizado anormalmente pues se ha producido un error o excepción, concretamente la excepción con el número 1329, que indica que no se han encontrado datos. Esto es lo que ocurrirá cuando ejecutemos una orden *fetch* que no devuelva datos.

Pues bien, si trabajamos con cursores es porque deseamos que se recorran varias filas de una o varias tablas y que se muestre información referente a esas filas. Dado que con una orden *fetch* recuperamos los datos de una fila de un cursor, deberemos emplear varias órdenes *fetch* para recorrer las diversas filas resultado de ejecutar una consulta. Pues bien, utilizaremos un bucle *while* para ejecutar varias órdenes *fetch*. Siempre que empleemos un cursor, después de abrirlo, recuperaremos la primera fila del mismo con una orden *fetch* y luego realizaremos un cierto tratamiento sobre las filas recuperadas dentro de un bucle *while* que tendrá como condición el que la orden *fetch* que se acaba de ejecutar no haya provocado la excepción NOT FOUND.

Pues bien, para poder trabajar con excepciones tenemos que declarar lo que se llama un manejador de errores o *handler*. Para declarar un manejador para la excepción NOT FOUND debemos emplear la siguiente sintaxis:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET NombreVariable = valor;
```

De esta manera, si al ejecutar una orden *fetch* se produce la excepción NOT FOUND, se asignará a la variable *NombreVariable* el valor indicado. Esta variable es necesario haberla declarado previamente. Normalmente lo que haremos es declarar una variable booleana llamada *fin* con valor inicial 0 y obtendremos los datos correspondientes a cada una de las filas del resultado de la ejecución de la sentencia select con una orden fetch dentro del bucle mientras que *fin* sea 0. Al declarar el manejador, indicaremos que cuando se produzca la excepción NOT FOUND, se asigne a la variable *fin* el valor 1. De esta manera, cuando una orden *fetch* no encuentre datos, se asignará a la variable *fin* el valor 1 y ya nos saldremos del bucle porque han sido tratadas todas las filas.

De esta manera, el procedimiento *VerPedido* correcto nos quedará con el siguiente código:

```
mysql> drop procedure VerPedido;//
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> Create procedure VerPedido()
    -> Begin
    -> Declare refe char(5);
    -> Declare fecha date;
    -> Declare fin bool default 0;
    -> Declare c cursor for select refped, fecped from pedido;
    -> Declare continue handler for not found set fin = 1;
    -> Open c;
    -> Fetch c into refe, fecha;
    -> while fin = 0 do
     ->   Select concat ('Referencia: ', refe, ' Fecha: ', fecha) "Datos pedidos";
    ->   Fetch c into refe, fecha;
    -> end while;
    -> Close c;
    -> End; //
Query OK, 0 rows affected (0.00 sec)

mysql> call verpedido();//
+-------------------------------------+
| Datos pedidos                       |
+-------------------------------------+
| Referencia: P0001 Fecha: 2014-02-16 |
+-------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------+
| Datos pedidos                       |
+-------------------------------------+
| Referencia: P0002 Fecha: 2014-02-18 |
+-------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------+
| Datos pedidos                       |
+-------------------------------------+
| Referencia: P0003 Fecha: 2014-02-23 |
+-------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------+
| Datos pedidos                       |
+-------------------------------------+
| Referencia: P0004 Fecha: 2014-02-25 |
+-------------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)
```

Al trabajar con cursores puede ocurrir que no nos sea posible indicar en la sentencia *select* asociada al cursor los términos exactos de la sentencia, sino que tengamos que utilizar una o varias variables. Pues bien, estas variables reciben el nombre de variables de acoplamiento. Para usarlas, será necesario declararlas como cualquier otra variable y luego utilizarlas en la sentencia *select* asociada al cursor. Si estas variables aparecen definidas como parámetros del procedimiento o función, entonces no es necesario declararlas explícitamente con *declare*.

Por ejemplo, para mostrar para los pedidos en los que se solicita un determinado artículo identificado por su código (parámetro del procedimiento), el código del pedido y el número de unidades solicitadas del artículo, podemos crear el siguiente procedimiento:

```
mysql> Create procedure VerArticuloPedido (art CHAR(5))
    -> begin
    -> Declare refe char(5);
    -> Declare cant int(4);
    -> declare fin bool default 0;
    -> Declare c cursor for select refped, cantart from lineapedido where
                                codart = art;
    -> declare continue handler for not found set fin = 1;
    -> open c;
    -> fetch c into refe, cant;
    -> while fin = 0 do
    ->   Select concat ('Pedido: ', refe, ' Unidades: ', cant) "Datos ";
    ->   fetch c into refe,cant;
    -> end while;
    -> close c;
    -> end;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> call VerArticuloPedido('A0043');//
+----------------------------+
| Datos                      |
+----------------------------+
| Pedido: P0001 Unidades: 10 |
+----------------------------+
1 row in set (0.00 sec)

+---------------------------+
| Datos                     |
+---------------------------+
| Pedido: P0002 Unidades: 5 |
+---------------------------+
1 row in set (0.00 sec)

+---------------------------+
| Datos                     |
+---------------------------+
| Pedido: P0004 Unidades: 5 |
+---------------------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

Como vemos, en la declaración del cursor se ha incluido una variable de acoplamiento (*art*) que es un parámetro del procedimiento. De esta manera, el resultado de la ejecución del procedimiento será diferente en función del artículo que reciba como parámetro el procedimiento.

5.3. Disparadores o triggers.

Un disparador es un tipo especial de rutina almacenada asociada a una tabla que se ejecuta o dispara automáticamente cuando ocurre una inserción (INSERT), borrado (DELETE) o modificación (UPDATE) sobre una tabla. Los disparadores se pueden emplear para diferentes propósitos, entre los que se encuentran:

- Implementar restricciones complejas de seguridad o de integridad.
- Impedir transacciones erróneas.
- Generar automáticamente valores derivados.
- Auditar actualizaciones, incluso enviando alertas.

La orden para crear un disparador es CREATE TRIGGER y requiere de la siguiente sintaxis:

```
CREATE TRIGGER nombre_disparador momento_disparo evento_disparo
ON nombre_tabla FOR EACH ROW
[orden_disparador]
sentencia_disparador
```

Como se puede fácilmente deducir, después de la palabra TRIGGER hay que escribir el nombre que se desea asignar al disparador. Después se debe indicar obligatoriamente el momento del disparo, que puede ser BEFORE (antes) o AFTER (después). A continuación se debe indicar el evento del disparo, que puede ser INSERT, UPDATE o DELETE. Luego se debe poner la palabra ON y el nombre de una tabla. Con toda esta información estaremos indicando si el disparador se tiene que ejecutar antes o después de realizar una inserción, borrado o modificación sobre una tabla.

Después se escribirá FOR EACH ROW, lo que significa que la sentencia del disparador se ejecutará por cada fila que se inserte, borre o actualice sobre la tabla especificada tras la palabra ON.

Debe tenerse en cuenta que puede haber varios disparadores que correspondan al mismo momento y evento de disparo sobre la misma tabla, por ejemplo, dos disparadores para ejecutarse antes de insertar en una determinada tabla. En ese caso, los dos disparadores se activarían en el mismo orden en el que fueron creados. Si se desea especificar el orden de ejecución de varios disparadores creados para el mismo momento y evento de disparo, se puede usar la sintaxis:

```
{follows | precedes} nombre_disparador_existente
```

Si se especifica *follows nombre_disparador_existente*, se está indicando que el disparador que se está creando se activará después del disparador existente cuyo nombre se indica después de la palabra *follows*. Por el contrario, si se especifica *precedes nombre_disparador_existente*, se está indicando que el disparador que se está creando se activará antes del disparador existente cuyo nombre se indica después de la palabra *precedes*.

Al final, se indicará la sentencia del disparador, esto es, la sentencia que queremos que se ejecute cuando se active el disparador. Si se desean ejecutar varias sentencias, deben colocarse entre las palabras BEGIN y END, que permiten construir sentencias complejas.

A las columnas de la tabla asociada al disparador nos podemos referir con los alias OLD y NEW. Con OLD.nombre_atributo nos referimos al valor de un atributo de una fila existente antes de ser borrada o modificada. Con NEW.nombre_atributo nos referimos al valor de un atributo en una nueva fila que va a ser insertada o después de ser modificada una fila existente.

Para trabajar con disparadores vamos a crear dos tablas similares a *Emple* y *Depart*. Comenzaremos creando una tabla llamada *Dep* conteniendo por cada departamento su número, nombre y el número de empleados que tiene:

```
mysql> create table Dep
    -> (numdep int(2) unsigned primary key,
    -> nomdep varchar(30) not null,
    -> numemple int(3) unsigned not null default 0);
Query OK, 0 rows affected (0.33 sec)
```

Creamos la tabla *Emp*, conteniendo por cada empleado, su número, nombre, salario y número del departamento en el que trabaja, que es una clave ajena a la tabla *Dep*:

```
mysql> create table Emp
    -> (numemp int(4) unsigned primary key,
    -> nomemp varchar(40) not null,
    -> salemp float(7,2) unsigned not null,
    -> numdep int(2) unsigned not null,
    -> foreign key(numdep) references Dep(numdep) on update cascade);
Query OK, 0 rows affected (0.11 sec)
```

Queremos que el atributo *numemple* de la tabla *Dep* se mantenga siempre actualizado, de manera que refleje de manera fiel el número de empleados de cada departamento. He asignado a este atributo el valor por defecto cero para que cuando se cree un departamento, si no se indica valor en la correspondiente sentencia INSERT, se le asigne un 0 a este atributo, como debe ser. Para mantener actualizado este atributo hemos de considerar las siguientes situaciones:

- Cada vez que se añada un nuevo empleado a la tabla *Emp*, se debe incrementar en una unidad el número de empleados (atributo *numemple*) para la fila correspondiente al departamento del empleado en la tabla *Dep*. Por ello, deberemos crear un disparador BEFORE o AFTER INSERT sobre la tabla *Emp*. En la sentencia del disparador deberemos indicar que se incremente el valor del atributo *numemple* de la tabla *Dep* en 1 para la fila cuyo número de departamento (*numdep*) sea el del departamento del nuevo empleado que se acaba de añadir.

```
mysql> create trigger NuevoEmpleado
    -> after insert on Emp for each row
    -> update Dep set numemple=numemple+1 where numdep=NEW.numdep;
Query OK, 0 rows affected (0.10 sec)
```

Para probar que este disparador funciona correctamente vamos a insertar primero dos departamentos en la tabla Dep y luego vamos a añadir un empleado al departamento número 1.

```
mysql> insert into Dep (numdep,nomdep)
    -> values (1, 'Compras'), (2, 'Ventas');
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> insert into Emp values (1, 'Jose Gil', 1250.45, 1);
Query OK, 1 row affected (0.07 sec)
```

Para comprobar si el disparador funciona correctamente, veamos el contenido de la tabla *Dep*.

```
mysql> select * from Dep;
+--------+---------+----------+
| numdep | nomdep  | numemple |
+--------+---------+----------+
|      1 | Compras |        1 |
|      2 | Ventas  |        0 |
+--------+---------+----------+
2 rows in set (0.00 sec)
```

Podemos observar como el número de empleados del departamento número 1 es 1, lo que quiere decir que el disparador se ha ejecutado y ha incrementado el número de empleados de la tabla *Dep* de cero a uno.

- Cada vez que se elimine a un empleado de la tabla *Emp*, se debe decrementar en una unidad el número de empleados (atributo *numemple*) para la fila correspondiente al departamento del empleado en la tabla *Dep*. Por ello, deberemos crear un disparador BEFORE o AFTER DELETE sobre la tabla *Emp*. En la sentencia del disparador deberemos indicar que se decremente el valor del atributo *numemple* de la tabla *Dep* en 1 para la fila cuyo número de departamento (*numdep*) sea el del departamento del empleado que se acaba de borrar.

```
mysql> create trigger BajaEmpleado
    -> after delete on Emp for each row
    -> update Dep set numemple=numemple-1 where numdep=OLD.numdep;
Query OK, 0 rows affected (0.08 sec)
```

Para probar si el disparador funciona correctamente eliminemos al empleado 1 creado con anterioridad, con lo que el número de empleados de su departamento (el número 1) deberá volver a tomar valor cero.

```
mysql> delete from Emp where numemp=1;
Query OK, 1 row affected (0.10 sec)

mysql> select * from Dep;
+--------+---------+----------+
| numdep | nomdep  | numemple |
+--------+---------+----------+
|      1 | Compras |        0 |
|      2 | Ventas  |        0 |
+--------+---------+----------+
2 rows in set (0.00 sec)
```

- Para terminar, cada vez que se cambie en la tabla *Emp* el número del departamento en el que trabaja un empleado, se debe decrementar en una unidad el número de empleados (atributo *numemple*) para la fila correspondiente al departamento en el que trabajaba el empleado en la tabla *Dep* e incrementar en una unidad el número de empleados (atributo *numemple*) para la fila correspondiente al nuevo departamento en el que trabaja el empleado. Por ello, deberemos crear un disparador BEFORE o AFTER UPDATE sobre la tabla *Emp*. Deberemos efectuar las dos actualizaciones solo en el caso de que el número de departamento después de efectuar la modificación (NEW) sea diferente del número de departamento antes de llevar a cabo el cambio (OLD). Deberemos incluir varias sentencias en el disparador, las cuales deberán ir, por tanto, entre las palabras BEGIN y END. En este caso, para poder delimitar las sentencias con el símbolo punto y coma (;) dentro del disparador, deberemos cambiar el delimitador de la sentencia que crea el disparador (en este caso se ha empleado el delimitador //). Por un lado, deberemos indicar que se decremente el valor del atributo *numemple* de la tabla *Dep* en 1 para la fila cuyo número de departamento (*numdep*) sea el del antiguo departamento del empleado. Por otro lado, deberemos indicar que se incremente el valor del atributo *numemple* de la tabla *Dep* en 1 para la fila cuyo número de departamento (*numdep*) sea el del nuevo departamento del empleado.

```
mysql> create trigger CambioDep
    -> after update on Emp
    -> for each row
    -> begin
    -> /*Si se cambia al empleado de nº de departamento, resto 1 al nº
       de empleados del departamento viejo y sumo 1 al nº de empleados
       del departamento nuevo*/
```

```
    -> if NEW.numdep != OLD.numdep then
    ->  update Dep set numemple = numemple - 1 where numdep = OLD.numdep;
    ->  update Dep set numemple = numemple + 1 where numdep = NEW.numdep;
    -> end if;
    -> end;
    -> //
Query OK, 0 rows affected (0.08 sec)
```

Para probar el funcionamiento de este disparador añadamos dos empleados a la tabla *Emp*, los cuales trabajen en el departamento número 1.

```
mysql> insert into Emp
    -> values (1, 'Ana Gil', 1350, 1), (2, 'Luisa Gómez', 2156.34, 1);
Query OK, 2 rows affected (0.09 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

Como podemos ver a continuación, el contenido de la tabla *Dep* en cuanto al número de empleados del departamento número 1 es correcto, ya que le acabamos de asignar dos empleados.

```
mysql> select * from Dep;
+--------+---------+----------+
| numdep | nomdep  | numemple |
+--------+---------+----------+
|      1 | Compras |        2 |
|      2 | Ventas  |        0 |
+--------+---------+----------+
2 rows in set (0.00 sec)
```

Ahora vamos a cambiar el número de departamento en el que trabaja la empleada número 1, asignándole el departamento número 2:

```
mysql> update Emp
    -> set numdep=2
    -> where numemp=1;
Query OK, 1 row affected (0.07 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Tras esta modificación el número de empleados del departamento 1 debe ser 1 y el número de empleados del departamento número 2 debe ser 1 también. Comprobémoslo:

```
mysql> select * from Dep;
+--------+---------+----------+
| numdep | nomdep  | numemple |
+--------+---------+----------+
|      1 | Compras |        1 |
|      2 | Ventas  |        1 |
+--------+---------+----------+
2 rows in set (0.00 sec)
```

Para eliminar un disparador se emplea la orden DROP TRIGGER, cuya sintaxis es la siguiente:

DROP TRIGGER [IF EXISTS] nombre_disparador

Por ejemplo, para borrar el disparador *BajaEmpleado* creado anteriormente escribiremos:

```
mysql> drop trigger BajaEmpleado;
Query OK, 0 rows affected (0.05 sec)
```

No existe ninguna sentencia SQL tipo ALTER TRIGGER para modificar un disparador. En caso de que se desee modificar un disparador, será necesario eliminarlo con DROP TRIGGER y volverlo a crear con CREATE TRIGGER.

Los disparadores se almacenan en la tabla TRIGGERS de la base de datos *information_schema*, como estudiamos en el tema 2. Además de poder consultar información sobre los disparadores creados en esta tabla, se puede hacer uso de la instrucción SHOW TRIGGERS, que presenta el formato:

SHOW TRIGGERS [[FROM | IN] nombre_BD];

Así, por ejemplo, para visualizar los disparadores existentes en la base de datos *Empresa*, escribiremos:

```
mysql> show triggers from Empresa;
```

5.4. Eventos.

Los eventos son tareas que se ejecutan en un momento determinado (día y hora) que se especifica al crear el evento. Por este motivo, a veces nos referimos a ellos como eventos programados.

Un evento se identifica por un nombre y el esquema o base de datos al que se asigna. Lleva a cabo la acción o acciones especificadas de acuerdo con un horario. Si son varias las acciones, se deben especificar entre BEGIN y END.

Podemos hablar de 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 (activado), OFF (desactivado) y DISABLED, en cuyo caso no se puede habilitar la activación en tiempo de ejecución.

Cuando el programador de eventos se detiene (variable global *event_scheduler* está a OFF), puede ser iniciado estableciendo la variable *event_scheduler* a ON. Podemos ver si está activo o no de dos maneras:

- Consultando la variable del sistema event_scheduler:

```
mysql> show variables like 'event%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| event_scheduler | ON    |
+-----------------+-------+
1 row in set, 1 warning (0.04 sec)
```

- Observando la salida del comando siguiente, ya que si está activo, el programador de eventos se ejecuta como un hilo más del servidor:

```
mysql> show processlist;
+----+-----------------+-----------------+---------+---------+--------+-----------------------+------------------+
| Id | User            | Host            | db      | Command | Time   | State                 | Info             |
+----+-----------------+-----------------+---------+---------+--------+-----------------------+------------------+
|  4 | event_scheduler | localhost       | NULL    | Daemon  | 285871 | Waiting on empty queue | NULL             |
|  9 | root            | localhost:51181 | pedidos | Query   |      0 | starting              | show processlist |
+----+-----------------+-----------------+---------+---------+--------+-----------------------+------------------+
2 rows in set (0.00 sec)
```

Vemos que el programador de eventos está activo. Si en algún momento estuviese inactivo (OFF), si *event_scheduler* no toma el valor DISABLED, podemos activarlo con el siguiente comando:

```
mysql> set GLOBAL event_scheduler = ON;
Query OK, 0 rows affected (0.04 sec)
```

Para la creación de eventos se usa la orden CREATE EVENT, cuya sintaxis se muestra a continuación:

CREATE EVENT [IF NOT EXISTS] nombre_evento ON SCHEDULE momento
[ENABLE|DISABLE] [COMMENT 'comentarios']
DO cuerpo_evento;

momento:
AT momento [+INTERVAL intervalo] | EVERY intervalo
[STARTS momento [+INTERVAL intervalo]]
[ENDS momento [+INTERVAL intervalo]]

intervalo:
número [YEAR | QUARTER | MONTH | WEEK | DAY |HOUR | MINUTE | SECOND]

Como se puede observar, después de CREATE EVENT es necesario escribir el nombre del evento y, a continuación, indicar después de ON SCHEDULE AT el momento en el que se desea que este se ejecute. Este momento debe ser un día y hora especificado en el formato 'AAAA-MM-DD HH:MM:SS' o CURRENT_TIMESTAMP, que indica ahora mismo. También se puede especificar un momento consistente en sumar un intervalo de tiempo a otro momento, o bien, que se ejecute cada cierto intervalo de tiempo (EVERY intervalo). En este último caso, se debe especificar cuándo debe comenzar a ejecutarse el evento (después de STARTS) y se puede indicar opcionalmente además cuándo debe finalizar su ejecución. En caso de que se desee especificar un intervalo de tiempo, se indicará un número y a continuación la palabra YEAR (año), QUARTER (trimestre)….o SECOND

(segundo). Así, si se indica EVERY 1 WEEK STARTS '2019-01-01 00:00:00' quiere decir que se debe ejecutar a las 0 horas del 1 de enero de 2019 y periódicamente cada semana.

Se puede crear un evento, pero dejarlo inactivo escribiendo la palabra DISABLE. Si se especifica ENABLE en vez de DISABLE, se indicará que se desea que esté activo. Además, recordemos que para que los eventos puedan funcionar la variable *event_scheduler* debe toma el valor ON.

Por último, después de la palabra DO se debe indicar obligatoriamente la sentencia o sentencias que deseamos que se ejecuten como parte del evento. Si se trata de varias sentencias, deberán especificarse entre BEGIN y END.

Vamos a crear un evento por medio del cual se calcule en el instante de crearlo y cada semana la antigüedad (número de años en la empresa) de los empleados de la tabla *Emple*. Para ello, vamos a crear una tabla *Emple2* copia de *Emple*, pero le vamos a añadir un campo entero de dos cifras no negativo llamado *antigüedad*, al que vamos a asignar un 0 como valor por defecto:

```
mysql> create table emple2 as select * from emple;
Query OK, 11 rows affected (0.41 sec)
Records: 11  Duplicates: 0  Warnings: 0

mysql> alter table emple2
    -> add antigüedad int(2) unsigned default 0;
Query OK, 0 rows affected (0.68 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Veamos ahora el contenido de la tabla *Emple2*:

```
mysql> select * from emple2;
+--------+----------+------------+------+------------+---------+----------+---------+-------------+
| emp_no | apellido | oficio     | dir  | fecha_alt  | salario | comision | dept_no | antigüedad  |
+--------+----------+------------+------+------------+---------+----------+---------+-------------+
|   7499 | ARROYO   | VENDEDOR   | 7698 | 2009-02-20 | 1200.00 |    30.00 |      30 |           0 |
|   7521 | SALA     | VENDEDOR   | 7698 | 2010-02-22 |  960.00 |    48.00 |      30 |           0 |
|   7566 | JIMÉNEZ  | DIRECTOR   | 7839 | 2010-02-04 | 2300.00 |     0.00 |      20 |           0 |
|   7654 | MARTÍN   | VENDEDOR   | 7698 | 2010-09-29 |  965.00 |  1000.00 |      30 |           0 |
|   7698 | NEGRO    | DIRECTOR   | 7839 | 2010-05-01 | 2200.00 |     0.00 |      30 |           0 |
|   7738 | CEREZO   | DIRECTOR   | 7839 | 2010-09-06 | 2210.00 |     0.00 |      10 |           0 |
|   7788 | GILA     | ANALISTA   | 7566 | 2013-04-23 | 2350.00 |     0.00 |      20 |           0 |
|   7839 | REY      | PRESIDENTE | NULL | 2010-11-17 | 3900.00 |     0.00 |      10 |           0 |
|   7876 | Alonso   | EMPLEADO   | 7788 | 2013-08-09 |  860.00 |     0.00 |      20 |           0 |
|   7900 | JIMENO   | EMPLEADO   | 7698 | 2010-12-03 |  725.00 |     0.00 |      30 |           0 |
+--------+----------+------------+------+------------+---------+----------+---------+-------------+
10 rows in set (0.00 sec)
```

Pues bien, vamos a crear el evento con el nombre *CalcularAntigüedad*. La antigüedad se calcula como el número de años transcurridos entre la fecha del día de hoy (fecha que nos devuelve la función *now()*) y la fecha de alta del empleado en la empresa (atributo *fecha_alt*). Para llevar a cabo este cálculo, MySQL nos proporciona la función *timestampdiff (unidad_tiempo, fecha1, fecha2)* que recibe como parámetro una unidad de tiempo y dos fechas tal que fecha2 es superior o igual a fecha1. Esta función nos devuelve la diferencia

entre esas dos fechas en la unidad de tiempo indicada como primer parámetro. La unidad de tiempo puede ser *year*, *month*, *week*, *day*, *hour*, *minute*, *second*… Pues bien, como lo que a nosotros nos interesa es el número de años entre las dos fechas, pondremos *year* como unidad de medida. El evento nos quedará como sigue:

```
mysql> create event CalcularAntigüedad
    -> on schedule every 1 week starts current_timestamp
    -> do
    -> update emple2
    -> set antigüedad=timestampdiff(year, fecha_alt, now());
Query OK, 0 rows affected (0.00 sec)
```

Si ahora consultamos el contenido de la tabla Emple2, veremos como el atributo antigüedad toma el valor correcto:

```
mysql> select * from emple2;
+--------+----------+------------+------+------------+---------+----------+---------+------------+
| emp_no | apellido | oficio     | dir  | fecha_alt  | salario | comision | dept_no | antigüedad |
+--------+----------+------------+------+------------+---------+----------+---------+------------+
|   7499 | ARROYO   | VENDEDOR   | 7698 | 2009-02-20 | 2000.00 |     0.00 |      30 |          9 |
|   7521 | SALA     | VENDEDOR   | 7698 | 2010-02-22 |  998.40 |    49.92 |      30 |          8 |
|   7566 | JIMÉNEZ  | DIRECTOR   | 7839 | 2010-02-04 | 2436.00 |     0.00 |      20 |          8 |
|   7654 | MARTÍN   | VENDEDOR   | 7698 | 2010-09-29 |  965.00 |  1000.00 |      30 |          8 |
|   7698 | NEGRO    | DIRECTOR   | 7839 | 2010-05-01 | 2200.00 |     0.00 |      30 |          8 |
|   7738 | CEREZO   | DIRECTOR   | 7839 | 2010-09-06 | 2210.00 |     0.00 |      10 |          8 |
|   7788 | GIL      | ANALISTA   | 7566 | 2013-04-23 | 2385.25 |     0.00 |      20 |          5 |
|   7839 | REY      | PRESIDENTE | NULL | 2010-11-17 | 3900.00 |     0.00 |      10 |          7 |
|   7844 | TOVAR    | VENDEDOR   | 7698 | 2010-09-08 | 1100.00 |     0.00 |      30 |          8 |
|   7876 | Alonso   | EMPLEADO   | 7788 | 2013-08-09 |  872.90 |     0.00 |      20 |          5 |
|   7900 | JIMENO   | EMPLEADO   | 7698 | 2010-12-03 |  725.00 |     0.00 |      30 |          7 |
+--------+----------+------------+------+------------+---------+----------+---------+------------+
11 rows in set (0.00 sec)
```

Los eventos se pueden modificar haciendo uso de la sentencia ALTER EVENT, cuya sintaxis es la siguiente:

ALTER EVENT nombre_evento
[ON SCHEDULE momento]
[RENAME TO nuevo_nombre_evento]
[ENABLE|DISABLE]
[COMMENT ‘comentarios’]
[DO cuerpo_evento];

Como se puede observar, se pueden modificar todas las características del evento, incluso su cuerpo y además se le puede cambiar de nombre.

Por su parte, para eliminar un evento se usará el comando DROP EVENT con la siguiente sintaxis:

DROP EVENT [IF EXISTS] nombre_evento

Los eventos se almacenan en la tabla *Events* de la base de datos *information_schema*. La estructura de esta tabla es la que se muestra a continuación:

```
mysql> desc events;
+----------------------+---------------+------+-----+---------------------+-------+
| Field                | Type          | Null | Key | Default             | Extra |
+----------------------+---------------+------+-----+---------------------+-------+
| EVENT_CATALOG        | varchar(64)   | NO   |     |                     |       |
| EVENT_SCHEMA         | varchar(64)   | NO   |     |                     |       |
| EVENT_NAME           | varchar(64)   | NO   |     |                     |       |
| DEFINER              | varchar(93)   | NO   |     |                     |       |
| TIME_ZONE            | varchar(64)   | NO   |     |                     |       |
| EVENT_BODY           | varchar(8)    | NO   |     |                     |       |
| EVENT_DEFINITION     | longtext      | NO   |     | NULL                |       |
| EVENT_TYPE           | varchar(9)    | NO   |     |                     |       |
| EXECUTE_AT           | datetime      | YES  |     | NULL                |       |
| INTERVAL_VALUE       | varchar(256)  | YES  |     | NULL                |       |
| INTERVAL_FIELD       | varchar(18)   | YES  |     | NULL                |       |
| SQL_MODE             | varchar(8192) | NO   |     |                     |       |
| STARTS               | datetime      | YES  |     | NULL                |       |
| ENDS                 | datetime      | YES  |     | NULL                |       |
| STATUS               | varchar(18)   | NO   |     |                     |       |
| ON_COMPLETION        | varchar(12)   | NO   |     |                     |       |
| CREATED              | datetime      | NO   |     | 0000-00-00 00:00:00 |       |
| LAST_ALTERED         | datetime      | NO   |     | 0000-00-00 00:00:00 |       |
| LAST_EXECUTED        | datetime      | YES  |     | NULL                |       |
| EVENT_COMMENT        | varchar(64)   | NO   |     |                     |       |
| ORIGINATOR           | bigint(10)    | NO   |     | 0                   |       |
| CHARACTER_SET_CLIENT | varchar(32)   | NO   |     |                     |       |
| COLLATION_CONNECTION | varchar(32)   | NO   |     |                     |       |
| DATABASE_COLLATION   | varchar(32)   | NO   |     |                     |       |
+----------------------+---------------+------+-----+---------------------+-------+
24 rows in set (0.00 sec)
```

Se almacena en esta tabla por cada evento toda la información referente al mismo, siendo relevante mencionar la base de datos en la que se ha creado el evento (EVENT_SCHEMA), el nombre del evento (EVENT_NAME), el código del cuerpo del evento (EVENT_DEFINITION), el tipo del evento (si es recurrente/repetitivo o no), el intervalo de tiempo en que se ejecuta (atributos INTERVAL_VALUE e INTERVAL_FIELD), cuándo empieza (STARTS) y cuándo finaliza (ENDS).

Así, mediante el siguiente comando, se puede ver información relevante acerca del evento *CalcularAntigüedad* que acabamos de crear:

```
mysql> select event_schema, event_name, event_type, interval_value, interval_field, starts
    -> from events
    -> where event_name = 'CalcularAntigüedad';
+--------------+--------------------+------------+----------------+----------------+---------------------+
| event_schema | event_name         | event_type | interval_value | interval_field | starts              |
+--------------+--------------------+------------+----------------+----------------+---------------------+
| empresa      | CalcularAntigüedad | RECURRING  |              7 | WEEK           | 2018-11-12 20:55:30 |
+--------------+--------------------+------------+----------------+----------------+---------------------+
1 row in set (0.00 sec)
```

Podríamos ver mediante el siguiente comando el código del mismo evento:

```
mysql> select event_definition
    -> from events
    -> where event_name = 'CalcularAntigüedad';
+-------------------------------------------------------------------------------+
| event_definition                                                              |
+-------------------------------------------------------------------------------+
| update emple2
set antigüedad=timestampdiff(year, fecha_alt now())                           |
+-------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

También se puede hacer uso del comando SHOW EVENTS, cuya sintaxis es la misma que la del comando SHOW TRIGGERS:

SHOW EVENTS [[FROM | IN] nombre_BD];