U.T.6: PROGRAMACIÓN EN BASES DE DATOS.

Los procedimientos almacenados (*Stored Procedures, SP*) suponen una importante innovación introducida en MySQL en la versión 5.0. Son funciones SQL definidas por el usuario que se ejecutan directamente en el servidor. Con los SP es posible guardar parte de la lógica de la aplicación cliente-servidor en el servidor.

En este capítulo veremos porqué los SP son una buena idea (más velocidad, más seguridad para los datos, menos duplicación de código, ...).

En general, las razones para usar SP son:

- **Velocidad**. En muchas ocasiones, determinadas operaciones de una aplicación pueden suponer que muchos datos sean enviados del servidor al cliente y viceversa. El cliente ejecuta un SELECT que produce muchos resultados, que son enviados desde el servidor al cliente. El cliente los procesa y envía una gran lista de UPDATEs de vuelta al servidor. Si todos estos pasos se pudieran ejecutar en el servidor nos evitaríamos una gran sobrecarga de la comunicación cliente/servidor. Además, MySQL puede preprocesar el código de los SP, evitando tener que compilar el código de un SP cada vez que se ejecuta. Sin embargo, MySQL no documenta qué tipo de optimizaciones realiza cuando precompila los SPs.
  De todas formas, usar SP no garantiza una mejora en la velocidad de ejecución de nuestras aplicaciones. Sólo se consiguen mejoras si el código del SP es eficiente. Y como SQL es un lenguaje mucho más primitivo que otros, no siempre es posible escribir código eficiente.
  Otro aspecto a tener en cuenta es que los SP aumentan la carga del servidor, mientras que la reducen en la parte del cliente. El resultado de este balance dependerá mucho de cómo esté configurado nuestro sistema y de cuales sean los cuellos de botella de nuestra aplicación.

- **Reutilización de código**. Ocurre frecuentemente que diferentes aplicaciones, o diferentes usuarios de una misma aplicación realizan la misma operación una y otra vez. Si la lógica de estas operaciones puede ser trasladada al servidor en forma de SPs entonces se consigue reducir la redundancia de código, y además, y muy importante, el sistema es mucho más fácil de mantener.

- **Seguridad de los datos**. Hay muchas aplicaciones en las que la integridad de los datos es crítica, como las relacionadas con el sector financiero. En estos casos, es deseable que los usuarios no puedan acceder directamente a determinadas tablas. La manera de obtener los datos es a través de SPs que se encargan de realizar los SELECT, UPTATE o INSERT a través de los SPs que son visibles a los usuarios. Un beneficio adicional es que los administradores pueden monitorizar los accesos a la base de datos mucho más fácilmente.

La gran desventaja de los SP es que normalmente su portabilidad es muy baja, con lo que las funciones escritas para un sistema concreto no se ejecutaran directamente en otro sistema. La razón es que cada sistema de bases de datos usa su propia sintaxis y sus propias extensiones para los SP, de forma que estos no están estandarizados.

Definir un SP

La mejor manera de ilustrar como funciona un SP es con un ejemplo.

El primer detalle a tener en cuenta es que en MySQL las sentencias se ejecutan después de escribir el carácter punto y coma (;), por esta razón antes de escribir el procedimiento almacenado la función del punto y coma se asigna a otros caracteres usando la sentencia *DELIMITER* seguida de un carácter tal como |, de esta manera el procedimiento puede ser escrito usando los puntos y comas sin que se ejecute mientras se escribe; una vez escrito el procedimiento, se escribe nuevamente la sentencia DELIMITER ; para asignar al punto y coma su función habitual.

Ahora el ejemplo. Supongamos que tenemos una tabla en la que almacenamos en forma de texto comentarios de los usuarios, de un máximo de 255 caracteres de longitud. Además, tenemos una columna en la que guardamos un resumen de n<255 caracteres del comentario. A la hora de guardar el resumen podemos hacer dos cosas:

a) ignorar el límite de n y MySQL truncará la cadena y se quedará con el principio de comentario, o
b) hacer algo más complicado como guardar el principio y el final del comentario, colocando la cadena '...' en medio. Para hacer esto definiremos un SP como este:

```
mysql> delimiter |
mysql> CREATE FUNCTION acortar(s VARCHAR(255), n INT)
       RETURNS VARCHAR(255)
       DETERMINISTIC
       BEGIN
              IF ISNULL(s) THEN
                     RETURN '';
              ELSEIF n<15 THEN
                     RETURN LEFT(s, n);
              ELSE
                     IF CHAR_LENGTH(s) <= n THEN
                            RETURN s;
                     ELSE
                            RETURN CONCAT(LEFT(s, n-10), ' ... ', RIGHT(s, 5));
                     END IF;
              END IF;
       END|
mysql> delimiter ;
```

Este procedimiento lo que hace es recibir una cadena de caracteres ***s*** (tipo VARCHAR(255)) y un entero ***n*** (tipo INT) y retorna una cadena de caracteres (VARCHAR(255)). El entero n indica a qué longitud queremos acortar la cadena inicial s.

Si n<15, consideramos que son muy pocos caracteres como para guardar el principio y el final y simplemente guardamos el principio. Si n>=15 y la cadena de entrada es mayor que n entonces generamos el resumen del comentario de la manera que hemos explicado anteriormente.

Para probar si funciona:

```
mysql> SELECT acortar("Este comentario no significa nada y solo sirve de
ejemplo", 15);

+-------------------------------------------------------------------------+
| acortar("Este comentario no significa nada y solo sirve de ejemplo", 15) |
+-------------------------------------------------------------------------+
| Este ... emplo |
+-------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT titulo, acortar(titulo, 20) FROM titulos LIMIT 10;

+----------------------------------------+----------------------+
| titulo | acortar(titulo, 20) |
+----------------------------------------+----------------------+
| Linux | Linux |
| The Definitive Guide to Excel VBA | The Defini ... l VBA |
| Client/Server Survival Guide | Client/Ser ... Guide |
| Web Application Development with PHP 4.0 | Web Applic ... P 4.0 |
| MySQL | MySQL |
| MySQL & mSQL | MySQL & mSQL |
| A Guide to the SQL Standard | A Guide to ... ndard |
| Visual Basic 6 | Visual Basic 6 |
| Excel 2000 programmieren | Excel 2000 ... ieren |
| PHP - Introducción | PHP - Introducción |
+----------------------------------------+----------------------+
10 rows in set (0.00 sec)
```

Consultar los SPs.

El almacenamiento interno de los SP se hace en la tabla ***mysql.proc***. En las columnas de esta tabla están almacenados todos los datos relativos al SP. Si no tenemos acceso a la base de datos *mysql*, podemos recuperar la misma información con ***INFORMATION_SCHEMA.ROUTINES***.

De esta manera si lo que queremos ver son todas las rutinas creadas en nuestras bases de datos, hacemos una consulta como esta:

SELECT specific_name FROM information_schema.routines;

Si lo que quieremos ver es el código de alguna rutina, para este ejemplo le vamos a llamar 'nombre_rutina':

SELECT routine_definition FROM information_schema.routines WHERE specific_name = 'nombre_rutina';

Si queremos saber de qué tipo es, es decir, si es función o procedimiento:

SELECT routine_type FROM information_schema.routines WHERE specific_name = 'nombre_rutina';

Para más información basta con echar un vistazo a las columnas de la tabla 'routines' del esquema 'information_schema':

SHOW columns FROM information_schema.routines;

Implementación de SP

Afortunadamente los SP de MySQL siguen el estándar SQL:2003. De esta manera, los SP de MySQL se pueden portar fácilmente al sistema DB/2 de IBM. Sin embargo, no se pueden portar a Oracle o Microsoft SQL Server ya que éstos no siguen el estándar.

Crear, editar y borrar SPs

La sintaxis para crear un SP es:

```
CREATE FUNCTION nombre ([parametro[,...]) RETURNS tipo_de_datos
[opciones] codigo_sql

CREATE PROCEDURE nombre ([parametro[,...])
[opciones] codigo_sql

parametro:
[IN | OUT | INOUT] nombre_parametro


SHOW PROCEDURE STATUS;
```

La única diferencia entre los dos es que ***FUNCTION*** retorna un valor y ***PROCEDURE*** no. El SP creado se asocia a la base de datos actual. Puede haber una FUNCTION y un PROCEDURE con el mismo nombre.

Los parámetros de FUNCTION son todos ***IN*** (parámetro de entrada) por defecto.
Los parámetros que son ***OUT*** (parámetro de salida) en un PROCEDURE son valores que se retornan y que se inicializan a NULL. Los ***INOUT*** son parámetros de PROCEDURE que son tanto de entrada como de salida.

Algunas de las ***opciones*** que se puede especificar son:

- LANGUAGE SQL: El único valor posible hasta la versión 8.0.

- [NOT] DETERMINISTIC: Un SP está considerado determinista cuando siempre retorna el mismo resultado con los mismos parámetros. Los SP que dependen de una tabla de la base de datos son no deterministas obviamente. Por defecto los SP son no deterministas. Sin embargo, los SP que si lo son pueden ejecutarse de una manera mucho más eficiente ya que, por ejemplo, los resultados se pueden almacenar en una cache. De cualquier manera, actualmente es obligatorio incluir el tipo DETERMINISTIC para la definición de funciones almacenadas si está activado el registro binario (Binary Logging) utilizado para realizar réplicas de la base de datos.

    - https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html
    - https://dev.mysql.com/doc/refman/8.0/en/stored-programs-logging.html
    - https://www.adictosaltrabajo.com/2009/12/08/mysql-replicacion/

- SQL SECURITY DEFINER o INVOKER: El modo SQL SECURITY especifica los privilegios de acceso al SP.

- COMMENT 'text': El comentario se almacena con el SP a modo de documentación.

Para borrar un SP:

```
DROP FUNCTION [IF EXISTS] nombre

DROP PROCEDURE [IF EXISTS] nombre
```

La opción ***IF EXISTS*** hace que si el SP no existe no se produzca ningún error.

Los SP pueden modificarse con ALTER. Se necesita el privilegio ALTER ROUTINE para hacerlo. Sólo se pueden modificar ciertas características de su definición pero no la lista de parámetros o el cuerpo del procedimiento o la función. Para ello tendríamos que borrarlo y volver a crearlo. La sintaxis es:

```
ALTER  PROCEDURE  nombre_procedimiento/function  [características ...]

características:
   COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
```

Un ejemplo de modificación de procedimiento para añadir un comentario:

```
ALTER  FUNCTION acortar COMMENT 'Función que resume un texto';
```

Los SP están compuestos principalmente de un conjunto de instrucciones SQL ordinarias. Sin embargo hay una serie de instrucciones adicionales que permiten control de flujo, alternativas, cursores,... Primero veremos cuáles son las diferencias entre FUNCTION y PROCEDURE, que están resumidas a continuación:

	PROCEDURE	**FUNCTION**
Llamada	Solo con CALL	Posible en todas las instrucciones SQL (SELECT, UPDATE, ...)
Retorno	Puede retornar uno o más SELECT	Retorna un valor único de un tipo determinado.
Parámetros	Permite por valor y por referencia (IN, OUT, INOUT)	Sólo parámetros por valor.
Instrucciones permitidas	Todas las SQL	No se puede acceder a tablas
Llamadas a otras funciones o procedimientos	Puede llamar a otros procedimientos y/o funciones	Sólo puede llamar otras Funciones

Tabla: Diferencias entre FUNCTION y PROCEDURE

Las reglas generales para escribir SP son:

- Punto y coma. Para separar las diferentes instrucciones que componen un SP se utiliza el carácter ';'.
- BEGIN-END. Las instrucciones que no se encuentran entre dos palabras reservadas (p.e. IF, THEN, ELSE) han de ponerse entre un BEGIN y un END.
- Salto de línea. Los saltos de línea son considerados como espacios en blanco.
- Variables. Las variables locales y los parámetros se acceden sin un carácter @ al principio. El resto de variables SQL deben comenzar con @.
- Mayúsculas/minúsculas. MySQL no las distingue al definir el nombre del SP.
- Caracteres especiales. Evitar el uso de caracteres fuera del código ASCII (acentos, n,...). A pesar de estar permitidos, pueden aparecer problemas al administrar la base de datos.
- Comentarios. Los comentarios se introducen con '--' y se extienden hasta el final de la línea.

6.2 Llamadas

Las funciones y los procedimientos tienen diferentes maneras de invocarse. Además, los SP están asociados a una base de datos con lo que si queremos ejecutar un SP de otra base de datos debemos hacerlo poniendo el nombre de la base de datos delante seguido por un punto (p.e. nombre_bd.nombre_sp())

Funciones

Las funciones, como las predefinidas por SQL, se pueden integrar en comandos SQL. Por ejemplo, la función acortar que hemos definido antes la hemos usado dentro de un SELECT. Por ejemplo:

```
SELECT acortar(titulo, 30), acortar(titulo, 20) FROM titulos

UPDATE titulos SET titulo = acortar(titulo, 70) WHERE CHAR_LENGHT(title)>70
```

Además, también se puede almacenar el resultado de una función en una variable SQL:

```
mysql> SET @s = " una cadena de caracteres my larga ";
Query OK, 0 rows affected (0.00 sec)

mysql> SET @a = acortar(@s, 15);
Query OK, 0 rows affected (0.00 sec)

mysql> select acortar(@s, 10) INTO @b;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @s, @a,@b;
+-----------------------------------+-----------------+------------+
| @s | @a | @b |
+-----------------------------------+-----------------+------------+
| una cadena de caracteres my larga | una ... arga | una caden |
1 row in set (0.00 sec)
```

Procedimientos

Los procedimientos deben ser llamados con ***CALL***. Se puede retornar como resultado una tabla (igual que con un comando SELECT). No se pueden usar procedimientos dentro de sentencias SQL.
Por ejemplo, podemos usar un procedimiento para obtener el titulo, subtitulo y editorial de cada libro:

```
CREATE PROCEDURE  biblioteca.obtener_titulo(IN id INT)
BEGIN
        SELECT titulo, subtitulo, nombreEdit
        FROM titulos, editoriales
        WHERE titulos.tituloID = id
        AND titulos.editID = editoriales.editID;
END
```

Y para usarlo:

```
mysql> CALL obtener_titulo(1);
+--------+-----------------------------------------+-------------+
| titulo | subtitulo | nombreEdit |
+--------+-----------------------------------------+-------------+
| Linux | Instalacion, Configuracion, Administracion | Grupo Anaya |
+--------+-----------------------------------------+-------------+
```

En el caso de procedimientos con variables de salida, el resultado sólo puede ser evaluado si se pasa una variable SQL:

```
mysql> delimiter |

mysql> CREATE PROCEDURE mitad(IN a INT, OUT b INT)
        BEGIN SET b = a/2;
        END |
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> CALL mitad(10, @resultado);
Query OK, 0 rows affected (0.04 sec)

mysql> SELECT @resultado;
+------------+
| @resultado |
+------------+
| 5 |
+------------+
```

Recursividad

En el código de un procedimiento se pueden llamar a otros procedimientos o funciones, mientras que en el código de una función sólo se pueden llamar a otras funciones. Además, funciones y procedimientos pueden llamarse a ellos mismos, con lo cual se pueden implementar algoritmos recursivos. Por ejemplo, podemos implementar el típico ejemplo del cálculo del factorial con una función:

```
CREATE FUNCTION factorial (n BIGINT) RETURNS BIGINT
BEGIN
  IF n>=2 THEN
    RETURN n * factorial(n-1);
 END IF;
 RETURN n;
END
```

Al instalar el servidor de MySQL la recursión está desactivada por defecto. Para poder activarla hay que cambiar la variable max_sp_recursion_depth a un valor mayor que 0. El valor que coloquemos será el nivel máximo de recursión permitido. Hay que tener cuidado con esta variable ya que al activar la recursión, y sobre todo con un valor alto, podemos sobrecargar el servidor si nuestras funciones o procedimientos están mal diseñados.
Para ver un listado de las variables de servidor:

mysql> SHOW VARIABLES;

6.3 Parámetros y valores de retorno

Hay una serie de detalles importantes acerca de cómo MySQL gestiona los parámetros que se pasan a funciones y procedimientos.

Procedimientos
Recordar que la sintaxis para crear un procedimiento es:

```
CREATE PROCEDURE nombre ([[IN OUT INOUT] nombre_parametro  tipo_parametro[,...])
```

Los atributos IN, OUT e INOUT determinan si el parámetro será usado sólo como entrada, sólo como salida, o en las dos direcciones. Se pueden usar todos los tipos de datos de MySQL. Sin embargo, no se pueden usar atributos para los tipos, como NULL, NOT NULL. Actualmente MySQL no hace ningún tipo de comprobación de tipos de los parámetros. A pesar de que los procedimientos no pueden retornar ningún valor, sí que se pueden usar instrucciones SELECT en su interior. Incluso se pueden ejecutar varios SELECT en secuencia. En ese caso, el procedimiento retorna el resultado en forma de varias tablas. Sin embargo, sólo los lenguajes

de programación que soportan el atributo MULTI_RESULT pueden acceder a esos resultados múltiples.

Funciones

La sintaxis para crear una función es:

```
CREATE FUNCTION nombre ([nombre_parametro tipo_datos[,...]) RETURNS tipo_datos
```

En este caso todos los parámetros son de entrada (por valor). Las funciones retornan un sólo valor del tipo especificado con la sentencia RETURN, que además termina la ejecución de la función.

6.4 Variables

Declaración
Se pueden declarar variables locales dentro de los procedimientos y funciones. La sintaxis es:

```
DECLARE nombre_var1, nombre_var2, .... tipo_datos [DEFAULT valor];
```

Se debe definir el tipo de datos para todas las variables. Se inicializan con NULL, a menos que se les asigne un valor explícito.
Hay que tener cuidado de no dar a las variables los mismos nombres que columnas o tablas de la base de datos ya que, a pesar de estar permitido, puede dar lugar a errores muy difíciles de rastrear.

Asignación

Las asignaciones de tipo *x = x + 1* no están permitidas en SQL. Hay que usar SET o SELECT INTO. Esta última es una variante de SELECT que acaba en INTO nombre_variable. Esta variante sólo se puede usar cuando la instrucción SELECT retorna un sólo registro. En las funciones sólo se puede usar SET, y no SELECT INTO.

Ejemplos:

```
SET var1=valor1, var2=valor2, ...

SELECT var = valor

SELECT 2*7 INTO var

SELECT COUNT(*) FROM tabla WHERE condicion INTO var

SELECT titulo, subtitulo FROM titulos WHERE tituloID=3 INTO mititulo, misubtitulo
```

6.5 Encapsulación de instrucciones

Cada función o procedimiento consiste en una o más instrucciones SQL que comienzan con un BEGIN y terminan con un END. Una construcción BEGIN-END puede incluirse dentro de otra si tenemos que declarar determinadas variables que sólo van a ser utilizadas en un determinado ámbito. Dentro de una construcción BEGIN-END hay que seguir un determinado orden:

```
BEGIN
   DECLARE variables;
   DECLARE cursores;
   DECLARE condiciones;
   DECLARE handlers;
   instrucciones  SQL;
END;
```

Antes de BEGIN puede añadirse una etiqueta, que se puede usar en conjunción con la instrucción LEAVE que sirve para abandonar un determinado ámbito definido por un BEGIN-END. Sintaxis:

```
nombre_bloque: BEGIN
    instrucciones;
    IF condicion THEN LEAVE nombre_bloque; ENDE IF;
    instrucciones;
END nombre_bloque;
```

De esta manera, con el comando LEAVE podemos escoger qué ámbito queremos abandonar.

6.6 Control de flujo

Hay básicamente dos maneras de controlar el flujo en SPs.

IF-THEN-ELSE

La sintaxis de esta construcción es:

```
IF condicion THEN
    instrucciones;
[ELSEIF condicion THEN
    instrucciones;]
[ELSE
    instrucciones;]
END IF;
```

No es necesario usar BEGIN-END dentro de estas estructuras de control. La condición puede formularse usando consultas con WHERE o HAVING.

CASE

Es una variante de IF que es útil cuando todas las condiciones dependen del valor único de una expresión. La sintaxis es:

```
CASE expresion
  WHEN valor1 THEN
        instrucciones;
  [WHEN valor2 THEN
        instrucciones;]
  [ELSE
        instrucciones;]
END CASE;
```

6.7 Bucles

MySQL ofrece una serie de opciones para construir bucles. Curiosamente, el bucle FOR no existe.

REPEAT-UNTIL

Las instrucciones dentro de esta estructura se ejecutan hasta que se cumple una determinada condición. Como la condición se evalúa al final del bucle, éste siempre se ejecuta al menos una vez.
El bucle puede llevar opcionalmente una etiqueta, que puede ser usada para abandona el bucle usando la instrucción LEAVE, o repetir una iteración adicional con ITERATE (ver más adelante).

```
[nombre_bucle:] REPEAT
        instrucciones;
UNTIL condicion1
END REPEAT [nombre_bucle];
```

Por ejemplo, ahora mostraremos una función que retorna una cadena de caracteres compuesta por n caracteres '*':

```
CREATE FUNCTION asteriscos (n INT) RETURNS TEXT
BEGIN
        DECLARE i INT DEFAULT 0;
        DECLARE s TEXT DEFAULT '  ';
        bucle1: REPEAT
                SET i = i + 1;
                SET s = CONCAT(s ,  "*");
        UNTIL i >= n END REPEAT;
        RETURN s;
END;
```

WHILE-DO

Las instrucciones entre WHILE y DO se ejecutan siempre que la condición correspondiente sea cierta. Como la condición se evalúa al principio del bucle, es posible que no se produzca ninguna iteración. Sintaxis:

```
[nombre_bucle:] WHILE condicion DO
        instrucciones;
END WHILE [nombre_bucle];
```

LOOP

Con esta estructura se define un bucle que se ejecuta hasta que se sale de el mediante una instrucción LEAVE. Sintaxis:

```
nombre_bucle: LOOP
        instrucciones;
END LOOP nombre_bucle;
```

LEAVE e ITERATE

LEAVE nombre_bucle aborta la ejecución de un bucle o de una estructura BEGIN-END.
ITERATE nombre_bucle sirve para ejecutar las instrucciones del bucle nombre_bucle una vez. ITERATE no puede usarse con BEGIN-END.

6.8 Gestión de errores (handlers)

Durante la ejecución de instrucciones SQL dentro de un SP pueden producirse errores. SQL define un mecanismo de handlers para gestionar esos errores. Un handler debe ser definido después de la declaración de variables, cursores y condiciones, pero antes del bloque BEGIN-END. Sintaxis:

```
DECLARE tipo HANDLER FOR condicion1, condicion2, ... instrucción;
```

Los elementos que intervienen en esta declaración son:

- ***tipo***: puede ser CONTINUE o EXIT. El primero significa que la ejecución del programa continuará a pesar del error. EXIT significa salir del bloque BEGIN-END donde se ha declarado el handler. Hay prevista una tercera opción que será UNDO y que servirá para deshacer los cambios producidos hasta ese momento por el SP.
- ***condición***: indica bajo qué condiciones debería activarse el handler. Hay varias opciones:
    - SQLSTATE 'codigo_error': especifica un error concreto mediante su código.

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

- SQLWARNING: comprende todos los estados SQLSTATE 01nnn.

```
DECLARE CONTINUE HANDLER FOR SQLWARNING
 BEGIN
  -- body of handler
 END;
```

- NOT FOUND: nombre de condición que comprende los errores que comienzan con un SQLSTATE 02nnn

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
 BEGIN
  -- body of handler
 END;
```

- codigo_error_mysql: se especifica el número de error MySQL.

```
DECLARE CONTINUE HANDLER FOR 1051
 BEGIN
  -- body of handler
 END;
```

- nombre_condicion: se refiera a una condicion que ha sido declarada entes con DECLARE CONDITION.

```
DECLARE no_such_table CONDITION FOR 1051;
DECLARE CONTINUE HANDLER FOR no_such_table
 BEGIN
  -- body of handler
 END;
```

- ***instrucción***: esta instrucción es ejecutada cuando se produce el error.

Puedes encontrar una lista de los códigos de error de MySQL y SQLSTATE en Section B.3, "Server Error Codes and Messages".

Las condiciones hacen posible dar a cierto grupo de errores un nombre claro. La sintaxis para declarar una condición es:

```
DECLARE nombre CONDITION FOR condición;
```

La condición se puede formular con *SQLSTATE 'codigo_error'* o *num_error_mysql*. Por ejemplo, si queremos gestionar el error que se produce al insertar un registro con una clave duplicada, haremos esto:

```
DECLARE clavedup VARCHAR(100);
DECLARE clave_duplicada CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR clave_duplicada SET mi_error='clavedup'
```

MySQL no permite que se pueda activar un error en un momento dado. La única manera es ejecutando una sentencia que sabemos que producirá el error deseado.

6.9 Cursores

Un cursor funciona como un puntero a un registro. Con ellos se puede iterar sobre todos los registros de una tabla. Normalmente se usan para facilitar el diseño de funciones y procedimientos que modifican los datos de una tabla, y que si se tuvieran que hacer con UPDATE serían muy complicados. De todas maneras, hay detractores del uso de cursores. La razón que esgrimen es que cualquier algoritmo que use cursores se puede reescribir de manera más elegante, y a veces más eficiente, sin usar cursores.

Sintaxis

El uso de cursores requiere varios pasos. Primero hay que declarar el cursor con:

```
DECLARE nombre_cursor CURSOR FOR SELECT ... ;
```

Se puede usar cualquier comando SELECT. Después hay que activar el cursor con:

```
OPEN nombre_cursor;
```

A partir de ese momento se puede usar el comando FETCH para acceder a cada registro de la consulta:

```
FETCH nombre_cursor INTO var1, var2, ...;
```

De esta manera, la primera columna del registro al que apunta *nombre_cursor* se almacena en la variable var1, la segunda en var2, y así sucesivamente. Estas variables tienen que haber sido declaradas con anterioridad, y han de ser del tipo correcto. Para saber cuándo se han acabado de leer los registros correspondientes al SELECT del cursor, MySQL emite el error 1329 (*no data to fetch*) que corresponde a *SQLSTATE 02000*. Este error no se puede evitar, pero se puede capturar con un handler. De esta manera, siempre que usemos cursores tendremos que definir el handler correspondiente. Normalmente se usa NOT FOUND como condición, donde se engloban todos los SQLSTATE 02nnn.

El cursor se puede cerrar con:

```
CLOSE nombre_cursor;
```

De todas maneras, ésto no se suele hacer ya que el cursor es automáticamente cerrado al final del bloque BEGIN-END.

Limitaciones

Hay tres limitaciones principales en el uso de cursores:

- Los cursores son sólo de lectura, no se pueden modificar los datos a los que apunta.
- Los cursores sólo pueden avanzar, de manera que los datos deben ser procesados en el orden en el que han sido proporcionados por el servidor.
- No se puede cambiar la estructura de las tablas mientras se están leyendo datos con un cursor. Si se hace, MySQL no garantiza un comportamiento consistente.

Como ejemplo haremos un procedimiento que recorre todos los registros de la tabla *Título* y suma el número de caracteres de *Titulo* y *Subtitulo*. Al final, la suma se divide por el número de registros lo que nos da el número medio de caracteres para *Título* más *Subtitulo*:

```
CREATE PROCEDURE biblioteca.testCursor (OUT longitud_media DOUBLE)
BEGIN
        DECLARE t, subt VARCHAR(100);
        DECLARE terminado INT DEFAULT 0;
        DECLARE n BIGINT DEFAULT 0;
        DECLARE cnt INT;
        DECLARE miCursor CURSOR FOR
                SELECT titulo, subtitulo FROM titulos;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET terminado=1;
        SELECT COUNT(*) FROM titulos INTO cnt;
        OPEN miCursor;
        miBucle: LOOP
                FETCH miCursor INTO t, subt;  ----→ handler
                IF terminado=1 THEN LEAVE miBucle; END IF;
                SET n = n + CHAR_LENGTH(t);
                IF NOT ISNULL(subt) THEN
                        SET n = n + CHAR_LENGTH(subt);
                END IF;
        END LOOP miBucle;
        SET longitud_media = n/cnt;
END
```

Y si lo ejecutamos obtendremos algo como:

```
mysql> CALL testcursor(@result);
Query OK, 0 rows affected (0.02 sec)
mysql> SELECT @result;
+--------------+
| @result |
+--------------+
| 31.629629629 |
+--------------+
```

Y para ilustrar como se podría haber hecho lo mismo sin un SP:

```
mysql> SELECT (SUM(CHAR_LENGTH(titulo)) +SUM(CHAR_LENGTH(subtitulo)))/COUNT(*)
AS longitud_media FROM titulos;

+----------------+
| longitud_media |
+----------------+
| 31.6296 |
```

6.10. Triggers

Es un mecanismo muy ligado a los SP. Los triggers son conjuntos de sentencias SQL o SPs que son ejecutados automáticamente antes o después de una modificación de la base de datos (UPDATE, INSERT, DELETE). Se usan para mantener condiciones sobre los datos de la base de datos y para monitorizar las operaciones. Hay que tener en cuenta que un trigger se ejecuta por cada modificación, así que si las operaciones a realizar automáticamente son muy complicadas, puede llevar a una ralentización considerable de nuestra aplicación. Esto se puede dar particularmente cuando se hacen modificaciones a muchos registros al mismo tiempo, ya que se ejecutará un trigger por cada registro modificado. Lo que explicaremos a continuación sobre triggers se aplica solamente a versiones iguales o superiores a 5.1.6.

La sintaxis para la creación de triggers es:

```
CREATE
        [DEFINER = { usuario | CURRENT_USER }]
        TRIGGER nombre_trigger tiempo_trigger evento_trigger
        ON nombre_tabla FOR EACH ROW sentencia_trigger
```

Un trigger es un evento ligado a una base de datos y a la tabla *nombre_tabla*, que ha de ser de tipo permanente, es decir, no puede ser de tipo TEMPORARY ni VIEW. El trigger se ejecuta cuando se produce una determinada operación. Para crear un trigger se necesita el privilegio TRIGGER para la tabla asociada. El atributo *DEFINER* determina el contexto de seguridad que se usará para determinar los privilegios de acceso en el momento de activación. El atributo *tiempo_trigger* determina si la activación se debe de hacer antes (*BEFORE*) o después (*AFTER*) de modificar un registro. El tipo de acción que activa el trigger viene determinado por *evento_trigger*. Los valores posibles son:

- INSERT. El trigger se activa cuando se inserta un registro. Esto incluye las instrucciones SQL INSERT, LOAD DATA y REPLACE.
- UPDATE. El trigger se activa cuando se modifica un registro. Esto corresponde a la instrucción SQL UPDATE.
- DELETE. El trigger se activa cuando se borra un registro. Por ejemplo cuando se usan instrucciones SQL como DELETE y REPLACE. Sin embargo, si se borran registros usando DROP TABLE o TRUCATE no se activan estos triggers ya que estas instrucciones no se transforman en DELETE.

Es importante recalcar que estos tres tipos de eventos no se corresponden con las instrucciones SQL que tienen el mismo nombre, sino que son tipos de eventos que pueden englobar uno o más tipos de instrucciones SQL. Como ejemplo de posible confusión tenemos el siguiente comando:

```
INSERT INTO ... ON DUPLICATE KEY UPDATE ...
```

A pesar de ser un comando INSERT, se pueden producir modificaciones si hay claves duplicadas, con lo que en éste solo comando se pueden activar triggers de tipo INSERT y UPDATE.
No se pueden tener dos triggers definidos para la misma tabla, el mismo tiempo y el mismo evento. Por ejemplo, para una tabla solo puede haber un trigger del tipo BEFORE INSERT.
La acción SQL que se ejecuta si se activa el trigger es *sentencia_trigger*. Si se quieren ejecutar varias instrucciones SQL se puede usar BEGIN-END. Eso quiere decir que se puede usar la misma sintaxis que para los procedimientos almacenados. De todas maneras, hay una serie de restricciones para los triggers en relación a las instrucciones que están permitidas. Las más importantes son:

- LOCK TABLES, UNLOCK TABLES.
- LOAD DATA, LOAD TABLE.
- Las instrucciones PREPARE, EXECUTE, DEALLOCATE PREPARE no se pueden usar.
- No se puede hacer commit ni rollback.

Más información en:
http://dev.mysql.com/doc/refman/5.1/en/routine-restrictions.html

Otra limitación en este momento es que los triggers no se activan cuando se ejecuta un CASCADE en una clave foránea, en vía corregirlo.

Veamos un ejemplo de cómo funcionan los triggers:

```
CREATE TABLE test1(a1 INT);
CREATE TABLE test2(a2 INT);
CREATE TABLE test3(a3 INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE test4(a4 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
                   b4 INT DEFAULT 0);

DELIMITER |

CREATE TRIGGER testref BEFORE INSERT ON test1
    FOR EACH ROW BEGIN
        INSERT INTO  test2  SET  a2  =  NEW.a1;
        DELETE FROM test3 WHERE  a3  =  NEW.a1;
        UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
   END;
|

DELIMITER ;

INSERT INTO test3 (a3) VALUES (NULL), (NULL), (NULL), (NULL), (NULL), (NULL), (NULL),
                              (NULL), (NULL), (NULL);
INSERT INTO test4 (a4) VALUES (0), (0), (0), (0), (0), (0), (0), (0), (0), (0);
```

Ahora realizamos un INSERT en la tabla test1:

```
mysql> INSERT INTO test1 VALUES (1), (3), (1), (7), (1), (8), (4), (4);
Query OK, 8 rows affected (0.04 sec)
Records: 8 Duplicates: 0 Warnings: 0
```

En esta instrucción, al haberse producido INSERTs en la tabla test1, el trigger testref se ha activado para cada uno de los INSERTS. Veamos si se ha ejecutado correctamente:

```
mysql> SELECT * FROM test1;
+------+
| a1 |
+------+
| 1 |
| 3 |
| 1 |
| 7 |
| 1 |
| 8 |
| 4 |
| 4 |
+------+
8 rows in set (0.00 sec)
```

La tabla test1, evidentemente, contiene los valores que hemos insertado directamente. Veamos el resto de tablas afectadas.

La tabla test2 ha quedado así:

```
mysql> SELECT * FROM test2;
+---+
|a2|
+--+
| 1 |
| 3 |
| 1 |
| 7 |
| 1 |
| 8 |
| 4 |
| 4 |
+---+
8 rows in set (0.00 sec)
```

Esto es debido a que el trigger lo único que hace es insertar un registro en test2 con los mismos valores que la columna a1 del nuevo registro (NEW.a1) que está a punto de insertarse.

La tabla test3 ha quedado así:

```
mysql> SELECT * FROM test3;
+----+
| a3 |
+----+
| 2 |
| 5 |
| 6 |
| 9 |
| 10 |
+----+
5 rows in set (0.00 sec)
```

La tabla test3 se había inicializado insertando un registro con valor NULL, pero al ser su columna *a3* de tipo NOT NULL PRIMARY KEY, el sistema inicializó los valores de esta columna con 1,2,3, ... 10. Al ejecutarse el segundo INSERT, el que activa los triggers, cada vez que se inserta un registro en test1, se borran todos los registros de test3 con valores en la columna *a3* igual al valor de la columna *a1* del registro que está a punto de insertarse. Por lo tanto, se borran los registros con valor en a3 igual a 1,3,4,7,8.

Y finalmente veamos cómo ha quedado la tabla test4:

```
mysql> SELECT * FROM test4;
+----+------+
| a4 | b4 |
+----+------+
| 1 | 3 |
| 2 | 0 |
| 3 | 1 |
| 4 | 2 |
| 5 | 0 |
| 6 | 0 |
| 7 | 1 |
| 8 | 1 |
| 9 | 0 |
| 10 | 0 |
+----+------+
10 rows in set (0.00 sec)
```

Si nos fijamos en la instrucción del trigger que afecta a la tabla test4, veremos que lo único que hace es contar cuántos registros insertados contienen el valor 1 en a1, el valor 2, y así sucesivamente.

Para referirnos al registro que se va a insertar o modificar en un trigger BEFORE podemos usar ***NEW***, y para referirnos al registro que se ha borrado o modificado en un trigger AFTER podemos usar ***OLD***.

También podemos alterar un registro que se va a insertar o modificar. Por ejemplo, podemos hacer cosas como ésta:

```
CREATE TRIGGER sdata_insert BEFORE INSERT ON `sometable`
FOR EACH ROW
BEGIN
SET NEW.guid = UUID();
END;
```

Sin embargo, esto sólo está permitido cuando el trigger es de tipo BEFORE. Si lo usamos en un trigger AFTER obtendremos un error.

Un punto importante a tener en cuenta es el uso de triggers BEFORE con tablas InnoDB. Si hemos definido restricciones, puede ser que determinados INSERT fallen debido a dichas restricciones, pero los triggers sí que se activarán !! Por eso es recomendable usar triggers de tipo AFTER siempre que sea posible.

Borrar triggers

Para borrar un trigger existente ejecutaremos ***DROP TRIGGER***. Es necesario especificar la tabla a la que está asociado:

```
DROP TRIGGER nombre_tabla.nombre_trigger;
```

6.11. Signals.

Las señales se han introducido en MySQL 5.5 para permitir lanzar tipos de excepciones. Esto no estaba permitido en las versiones anteriores, dificultando así la programación de procedimientos almacenados sólidos con manejo de excepciones. Este tipo de elementos es especialmente útil para implementar restricciones sobre las tablas.

El resultado es que podemos construir esquemas de BD con reglas de integridad más complejas y realizar el control de excepciones.

Hay casos en los que creamos un trigger con el objetivo de comprobar que los datos modificados en una tabla con las operaciones INSERT, UPDATE o DELETE cumplen algún requisito. En caso de no hacerlo, debemos cancelar la operación, es decir, evitar que se

produzca la operación que desencadenó el trigger. Esta puede ser otra utilidad que MYSQL nos ofrece con la sentencia SIGNAL.

La sintaxis para la utilización de señales es:

```
SIGNAL condition_value [SET signal_information_item
                [signal_information_item] ...]

condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | condition_name

signal_information_item:
    condition_information_item_name = simple_value_specification

condition_information_item_name:
    CLASS_ORIGIN
  | SUBCLASS_ORIGIN
  | MESSAGE_TEXT
  | MYSQL_ERRNO
  | CONSTRAINT_CATALOG
  | CONSTRAINT_SCHEMA
  | CONSTRAINT_NAME
  | CATALOG_NAME
  | SCHEMA_NAME
  | TABLE_NAME
  | COLUMN_NAME
  | CURSOR_NAME
```

En el ejemplo siguiente crearemos las tablas *video*, *audio* y *pdf* que contienen datos sobre recursos audiovisuales. Los recursos deben tener un nombre único en la base de datos, restricción que controlaremos mediante un trigger que evitará la inserción de un nuevo recurso en caso de duplicidad de nombre invocando una señal que aborte dicha inserción.

```
CREATE TABLE audio (id int auto_increment, resource_name varchar(255), primary key (id));
CREATE TABLE video (id int auto_increment, resource_name varchar(255), primary key (id));
CREATE TABLE pdf (id int auto_increment, resource_name varchar(255), primary key (id));


DELIMITER |
CREATE TRIGGER audio01 BEFORE INSERT ON audio
FOR EACH ROW
BEGIN
 DECLARE count INT;
 -- DECLARE CONTINUE HANDLER FOR SQLSTATE '45000' set @error=1;

 SELECT COUNT(1) INTO count FROM video WHERE resource_name = new.resource_name;
```

```
 IF count > 0 THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'El nombre del recurso existe en video';
 END IF;
 SELECT COUNT(1) INTO count FROM pdf WHERE resource_name = new.resource_name;
 IF count > 0 THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'El nombre de recurso existe en pdf';
 END IF;
END; |

DELIMITER ;
```

Lanzamos algunas inserciones para provocar la ejecución del trigger:

```
insert into audio(resource_name) values ('test');
insert into video(resource_name) values ('test1');
insert into audio(resource_name) values ('test1');.
```

El resultado de estas inserciones:

```
mysql> insert into audio(resource_name) values ('test');
Query OK, 1 row affected (0.02 sec)

mysql> select * from audio;
+----+---------------+
| id | resource_name |
+----+---------------+
|  1 | test          |
+----+---------------+
1 row in set (0.00 sec)

mysql> insert into video(resource_name) values ('test1');
Query OK, 1 row affected (0.06 sec)

mysql> insert into audio(resource_name) values ('test1');
ERROR 1644 (45000): El nombre del recurso existe en video.
mysql> select * from audio;
+----+---------------+
| id | resource_name |
+----+---------------+
|  1 | test          |
+----+---------------+
1 row in set (0.00 sec)
```

En este otro ejemplo vamos a controlar que cada vez que se actualice el sueldo de un empleado de la tabla EMPLEADO, sea para aumentarlo. Creamos el siguiente trigger:

```
delimiter |
CREATE TRIGGER comprueba_sueldo BEFORE UPDATE ON Empleado FOR EACH ROW
BEGIN
        IF NEW.sueldo <= OLD.sueldo THEN
                SIGNAL SQLSTATE '45000' SET message_text='El sueldo debe incrementarse';
        END IF;
END |
delimiter ;
```

En el ejemplo anterior se comprueba que el sueldo nuevo de un empleado (guardado en la variable NEW.sueldo) es siempre mayor que el sueldo que tenía (OLD.sueldo). Si no se cumple esta condición, creamos un signal especificando el SQLSTATE junto con el texto(opcional) que queremos que se muestre como error. Al ejecutarse, el signal aborta la actualización que había desencadenado el trigger.