

PARTE II.

PROGRAMAS ALMACENADOS

Tema 2. Programas almacenados en MySQL

2.1. Introducción.....	3
2.2. Ventajas de su utilización.....	3
2.3. Edición de programas.....	4
2.4. Fundamentos básicos del lenguaje	8
2.4.1. Variables, tipos de datos, comentarios y literales.....	8
2.4.2. Variables de usuario	12
2.4.3. Parámetros	13
2.4.4. Principales operadores.....	15
2.4.5. Expresiones.....	16
2.4.6. Funciones incorporadas	17
2.4.7. Bloques de instrucciones	20
2.4.8. El comando IF	22
2.4.9. El comando CASE.....	24
2.4.10. El comando LOOP, LEAVE e ITERATE	25
2.4.11. El comando REPEAT ... UNTIL.....	26
2.4.12. El comando WHILE	27
2.4.13. Bucles anidados	28
2.5. Procedimientos	28
2.5.1. Creación de Procedimientos. Diccionario de datos	28
2.5.2. Modificación de Procedimientos	31
2.5.3. Borrado de Procedimientos	32
2.5.4. Utilización de instrucciones DDL y DML en procedimientos almacenados	32
2.5.5. Utilización de instrucciones de consulta en procedimientos almacenados	32
2.5.6 Almacenar en variables el valor de una fila de una tabla	33
2.5.7 Sentencias preparadas. SQL dinámico	35
2.6. Cursores.....	38
2.6.1. Sentencias utilizadas con cursores. Ejemplos	38
2.6.2. Cursores anidados.....	43
2.7. Manejo de errores.....	44
2.7.1 Introducción a la gestión de errores.....	44
2.7.2 Tipos de manejador	48
2.7.3 La condición del manejador	49
2.7.4 Orden de actuación del manejador	52
2.7.5 Ámbito de actuación del manejador	53
2.7.6 Ejemplo de tratamiento de errores.....	54
2.8 Funciones.....	55
2.8.1 Creación de funciones. Diccionario de datos	55
2.8.2 Ejemplos de utilización de Funciones	56
2.8.3 Modificación de funciones	60
2.9 Triggers.....	60
2.9.1 Creación de triggers. Diccionario de datos.....	60

2.9.2 Borrado de triggers	62
2.9.3 Utilización de triggers	63

Anexo. MySQL desde otras aplicaciones

1. Introducción: Ventajas y desventajas	66
2. Ejecución de procedimientos almacenados. Tratamiento de errores.....	67
2.1 En PHP	67
2.2 En Visual Basic Express 2005.....	69
3. Ejecución de funciones.....	70
3.1 En PHP	70
3.2 En Visual Basic Express 2005.....	71
4. Ejecución de sentencias DDL.....	71
4.1 En PHP	71
4.2 En Visual Basic Express 2005.....	72
5. Ejecución de sentencias preparadas.....	72
4.1 En PHP	72
4.2 En Visual Basic Express 2005.....	74

TEMA 2

Programas almacenados en MySQL

Alberto Carrera Martín

2.1 Introducción

La aparición de los procedimientos almacenados, funciones y triggers (desencadenadores) en MySQL 5 ha supuesto una enorme revolución en este gestor de bases de datos que ya disfrutaba de una gran popularidad. Se ha producido el salto para que MySQL pueda ser utilizado como SGBD empresarial. A ello hay que añadir que la sintaxis de los programas es sencilla lo que facilita su escritura.

Un programa almacenado es un conjunto de instrucciones almacenadas dentro del servidor de bases de datos y que se ejecutan en él. Este conjunto se identifica por un nombre.

Tipos de programas almacenados:

- **Procedimientos almacenados:** El más común de los programas almacenados. Resuelven un determinado problema cuando son llamados y pueden aceptar varios parámetros de entrada y devolver varios de salida.
- **Funciones almacenadas:** Similares a los procedimientos salvo que sólo devuelven un valor como parámetro de salida. La ventaja que presentan las funciones es que pueden ser utilizadas dentro de instrucciones SQL y por tanto aumentan considerablemente las capacidades de este lenguaje.
- **Triggers** o desencadenadores o disparadores: Son programas que se activan (“disparan”) ante un determinado suceso ocurrido dentro de la base de datos.

Por el momento, MySQL a diferencia de otros SGBD comerciales no ofrece la posibilidad de utilizar ni paquetes ni clases.

Los programas almacenados en MySQL cumplen en gran medida el estándar ANSI de especificación de ANSI SQL:2003 SQL/PSM (Persistent Stored Module).

2.2 Ventajas de su utilización

Ventajas de la utilización programas almacenados en comparación con los realizados en un determinado lenguaje y que no residen en el servidor:

- Mayor seguridad y robustez en la base de datos. Al permitir que los usuarios puedan ejecutar diferentes programas (los que estén autorizados), se está limitando e impidiendo el acceso directo a las tablas donde están almacenados los datos evitando la manipulación directa de éstas por parte de los usuarios y por tanto eliminando la posibilidad de pérdida accidental de los datos. Los programas serán los que accederán a las tablas.

- Mejor mantenimiento de las aplicaciones que acceden a los programas almacenados y por tanto disminución de la posibilidad de aparición de errores. En lugar de que cada aplicación cliente disponga de sus propios programas para realizar operaciones de inserción de consulta o actualización, los programas almacenados permiten centralizar los métodos de acceso y actualización anteriores presentando una interfaz común para todos los programas.
- Mayor portabilidad de las aplicaciones (relacionada con el punto anterior) puesto que la lógica de la aplicación ya queda implementada en los programas almacenados permitiendo al programador centrarse sólo en su interfaz.
- Debido a la fuerte integración con el lenguaje SQL, no se necesita de ningún tipo de conector o driver como ODBC (Open DataBase Connectivity) o JDBC (Java DataBase Connectivity) para poder construir y ejecutar sentencias SQL. Bastará agrupar estas últimas bajo un programa almacenado que se llamará en el momento en el que se necesite.
- Reducción del tráfico de red. El cliente llama a un procedimiento del Servidor enviándole unos datos. Éste los recibe y tras procesarlos devuelve unos resultados. Por la red no viajan nada más que los datos. En contrapartida señalar que se produce una carga más elevada en el servidor, mucho más que si las aplicaciones se ejecutaran en los clientes, pero hoy en día no supone mucho inconveniente con la tecnología actual de servidor de que se dispone.

2.3 Edición de programas

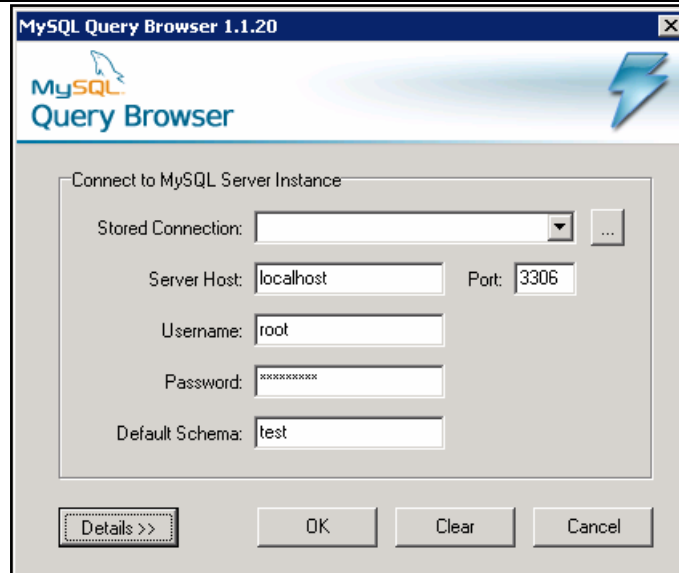
Para editar programas se pueden utilizar diferentes herramientas o métodos:

- A través de la línea de comando del cliente MySQL.
- Herramienta *MySQL Query Browser*.
- Cualquier editor de texto o cualquier otra herramienta de terceros como el *TOAD* para MySQL.

Como hemos hecho con el tema anterior de vistas, utilizaremos la herramienta *MySQL Query Browser* porque presenta mayores ventajas como ayuda incorporada, posibilidad de ejecutar sentencias SQL o la visualización de palabras clave del lenguaje resaltadas en diferentes colores entre otras características.

Pasaremos a continuación a crear nuestro primer procedimiento. Para ello:

1. Arranca la herramienta *MySQL Query Browser* (*Botón Inicio / Todos los Programas / MySQL / MySQL Query Browser* si trabajas bajo el sistema operativo Windows). Aunque el usuario *root* solo debe utilizarse para labores de administración principalmente, para probar los diferentes procedimientos de estos temas puedes conectarte con él y a la base de datos *test* tal y como muestra la siguiente figura:

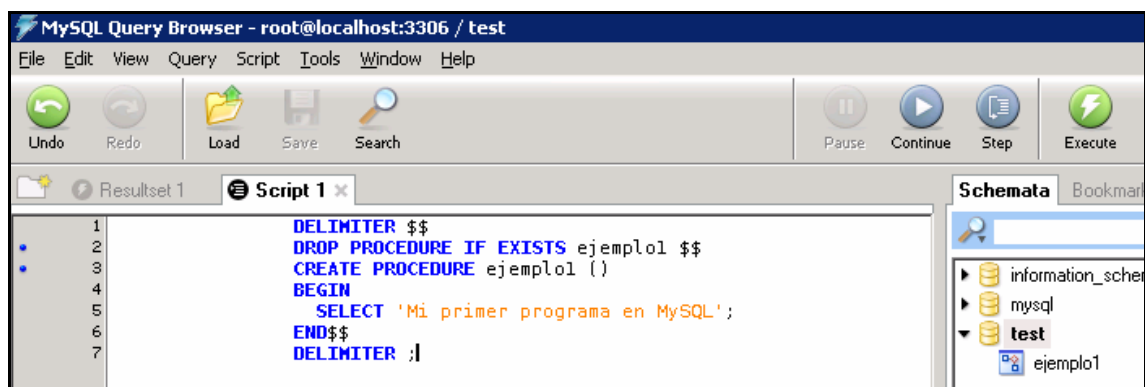
**Ilustración 1. Conexión a la base de datos**

2. Elige el comando u opción de menú *File / New Script Tab*.
3. Introduce las líneas de código que vienen a continuación:

```
DELIMITER $$  
DROP PROCEDURE IF EXISTS ejemplo1 $$  
CREATE PROCEDURE ejemplo1 ()  
BEGIN  
    SELECT 'Mi primer programa en MySQL';  
END$$  
DELIMITER ;
```

Breve explicación: MySQL utiliza el carácter “;” para finalizar cada sentencia SQL. Como dentro del cuerpo del procedimiento las instrucciones van separadas por “;” (5ª línea) para distinguirlas de las sentencias SQL necesitamos utilizar otro carácter delimitador (en la 1ª línea se habilita y en la última se vuelve a dejar el que estaba). La sentencia DROP PROCEDURE.... borra el procedimiento si este estuviera previamente creado (de no ponerlo daría error); en la 3ª línea se crea dicho procedimiento.

La 4ª línea indica el comienzo del cuerpo del procedimiento que finaliza en la penúltima línea.

**Ilustración 2. Nuestro primer procedimiento**

4. Pulsa el botón *Guardar* o comando del menú *File / Save* – Nombre del script: *ejemplo1*.
5. Para compilar el procedimiento anterior en busca de posibles errores (no es el caso si lo has copiado igual que aquí aparece) elige la opción de menú *Script / Execute* o el botón *Execute* (botón verde de la parte superior derecha de la ilustración 2 anterior). Si no te apareciera su nombre (*ejemplo1*) o cualquier otro objeto que crees en adelante en la pestaña *Schemata* tal y como figura en la ilustración 2 anterior, selecciona la base de datos *test* y con el botón derecho la opción *Refresh* (refrescar):

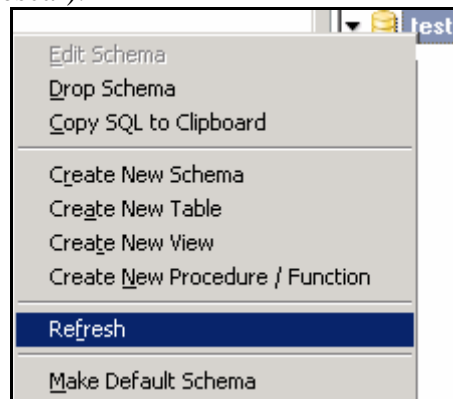


Ilustración 3. Opción Refresh

6. Para ejecutar el procedimiento, haz doble clic sobre su nombre *ejemplo1* que figura dentro de la pestaña *Schemata* de la ilustración 2 anterior (o también en la ilustración 4 siguiente). Esto se traduce en la siguiente sentencia de llamada:

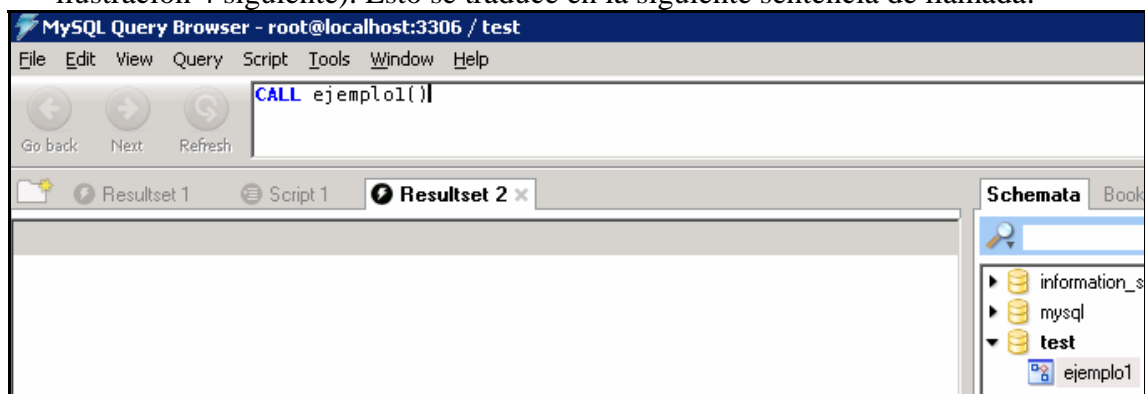


Ilustración 4. Ejecutando el procedimiento

A continuación pulsa el botón *Execute* (si no aparece en pantalla se consigue la misma funcionalidad pulsando la combinación de teclas `<ctrl.>+<enter>` o eligiendo el comando *Query / Execute* del menú). Aparecerá el resultado de la ejecución del programa:

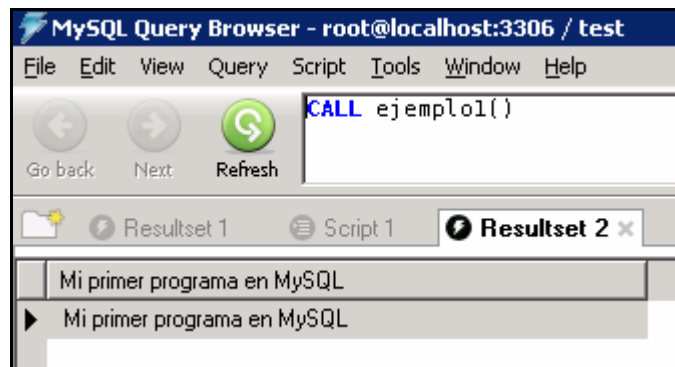


Ilustración 5. Resultado de la ejecución

Puedes también crear procedimientos y funciones de una forma rápida utilizando la opción de menú *Script / Create Stored Procedure – Function*. De esta manera ahorrarás mucho tiempo introduciendo código al insertarse automáticamente las sentencias básicas. Para probar esta última posibilidad selecciona la base de datos *test* y pulsa la opción de menú anterior:

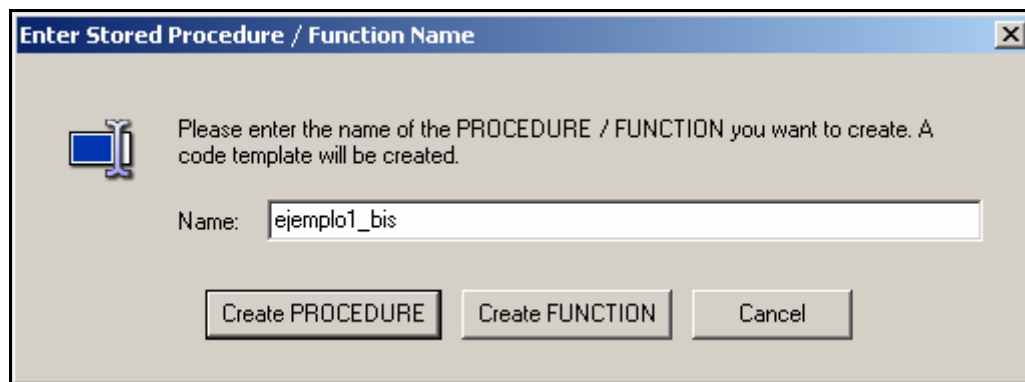


Ilustración 6. Creación de procedimientos mediante comandos de menú

Pulsa el botón *Create PROCEDURE*

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS `test`.`ejemplo1_bis` $$
4 CREATE PROCEDURE `test`.`ejemplo1_bis` ()
5 BEGIN
6
7 END $$
8
9 DELIMITER ;

```

Procedimiento 1 ejemplo1_bis

y verás que sólo tienes que escribir en la línea 6 que está vacía la instrucción:

`SELECT 'Mi primer programa en MySQL';`

El nombre que precede al del procedimiento corresponde al de la base de datos donde queda almacenado dicho procedimiento, que es con la que estás trabajando al tenerla seleccionada.

Para modificar o volver a editar el contenido de cualquier programa puedes hacerlo:

- Si no aparece en pantalla, utiliza el comando *File / Open Script* del menú para abrir el script que contiene el programa almacenado.
- Modifica las líneas de código necesarias.
- Volver a guardarlo con la opción del menú *File / Save*.
- Sigue los mismos pasos vistos para compilarlo y ejecutarlo.

Si todo ha ido bien dispones del programa en dos sitios, tanto en el disco duro de tu ordenador como dentro de la base de datos.

Podrías también modificar el contenido del programa almacenado seleccionándolo de la pestaña *Schemata*, después hacer clic con el botón derecho y comando *Edit Procedure* tal y como muestra la ilustración 7.

Después de modificarlo puedes guardarlo, compilarlo y ejecutarlo como se ha detallado anteriormente.

Quizás sea más interesante utilizar el primero de estos dos métodos, modificar directamente el script almacenado en el disco por razones de seguridad (disponer de una copia fuera de la base de datos), por poder migrar el procedimiento a otra base de datos distinta (observa en este último método en la línea 3 como este procedimiento *ejemplo1bis* está asociado a la base de datos test) y siempre se pueden guardar diferentes versiones en disco (ejemplo1, ejemplo 1_1, ejemplo1_2...) que contengan los diferentes cambios que se han ido produciendo en la mejora del programa.

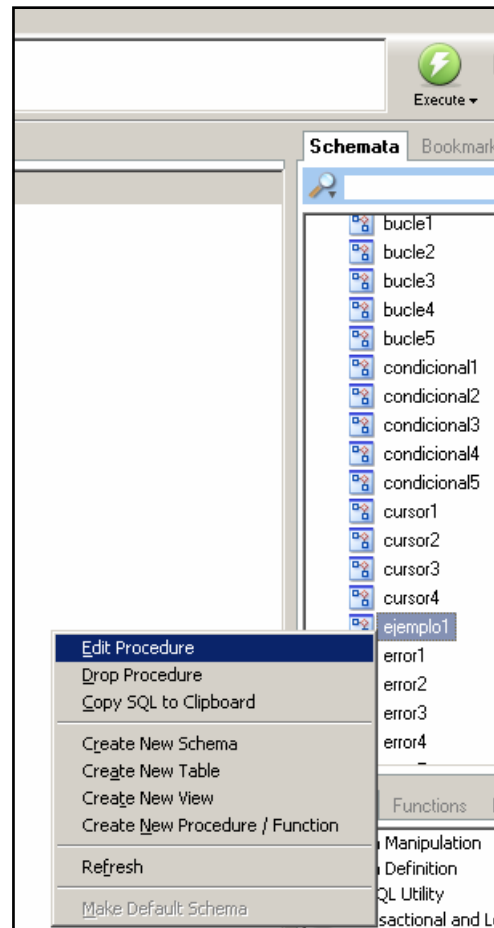


Ilustración 7. Otra forma de editar procedimientos

2.4 Fundamentos básicos del lenguaje

2.4.1 Variables, tipos de datos, comentarios y literales

Las variables son elementos de datos con un nombre cuyo valor puede ir cambiando a lo largo de la ejecución del programa.

SINTAXIS:

```
DECLARE nombre_variable1 [,nombre_variable2...] tipo [DEFAULT valor];
```

La declaración de variables se realiza antes del comienzo de las instrucciones (y antes de los cursores y manejadores de errores que se verá también en este curso)

De la sintaxis anterior se deduce que se pueden declarar varias variables del mismo tipo seguidas y separadas por comas. Al mismo tiempo que se declaran las variables se pueden definir asignándoles un valor inicial mediante la cláusula DEFAULT; si no se les asigna ninguno entonces las variables quedan definidas al valor NULL. En el momento de la declaración hay que indicar el tipo de datos de la variable; pueden utilizarse cualquiera de los que se emplean en el momento de crear tablas mediante la instrucción CREATE TABLE. Veremos los que se utilizan más frecuentemente.

TIPOS DE DATOS NUMÉRICOS

INT, INTEGER	Entero. Los valores pueden ir desde -2147483648 a 2147483647 para enteros con signo o desde 0 a 4294967295 para enteros sin signo
TINYINT	El más pequeño de los enteros. Rango entre -128 y -127 con signo, o de 0 a 255 sin signo
SMALLINT	Entero entre -32768 y 32767 (valores con signo) o entre 0 y 65535 (valores con signo)
MEDIUMINT	Entero de valores con signo que van de -8388608 a 8388607 o para valores sin signo de 0 a 16777215
BIGINT	Entero grande. Con signo puede tomar valores desde -9223372036854775808 a 9223372036854775807 y sin signo de 0 a 18446744073709551615
FLOAT	Real de precisión simple. Permite almacenar números de -1.7E38 a 1.7E38 con signo o de 0 a 3.4E38 para valores sin signo
DOUBLE	Real de precisión doble. Puede llegar a alcanzar valores de 0 a 1.7E308 para números sin signo
DECIMAL(precisión, escala)	Equivalen al tipo DOUBLE pero se diferencian en que ocupan bastante mayor espacio (por almacenar valores exactos y no aproximados). Si el número de decimales es importante (cantidades monetarias) es mejor utilizar el tipo NUMERIC.
NUMERIC(precisión, escala)	Precisión indica el número de dígitos totales, escala es el número de decimales a la derecha de la coma del total de dígitos que viene expresado en la precisión

TIPOS DE DATOS DE TEXTO

CHAR(longitud)	Cadenas de texto de longitud fija hasta un máximo de longitud de 255 caracteres. Si el valor a almacenar es más corto que la longitud de la variable el resto de caracteres se rellenan a blancos
VARCHAR(longitud)	Cadenas de texto de longitud variable hasta un máximo de 64 KB. A diferencia del tipo CHAR, si el valor a almacenar es más corto, el tamaño real de la variable es el número de caracteres que ocupa el valor pues no rellena a blancos. Como almacena la longitud junto con los caracteres, su utilización en los programas hace que la ejecución de éstos sea un poco más lenta que si se utiliza el tipo CHAR
ENUM	Almacena un valor concreto de un conjunto posible de valores
SET	Similar a ENUM pero permite guardar más de un valor
TEXT	Texto de hasta 64 KB. de tamaño
LONGTEXT	Texto de hasta 4 GB. de tamaño

TIPOS DE DATOS DE FECHA Y HORA

DATE	Fechas con el formato AAAA-MM-DD entre 1000-01-01 y 9999-12-31
DATETIME	Fecha y hora con el formato AAAA-MM-DD hh:mm:ss. Para la parte de la hora el rango debe estar entre 00:00:00 y 23:59:59

OTROS TIPOS DE DATOS

BLOB	Hasta 64KB. de datos binarios
LOB	Hasta 4GB. de datos binarios

A continuación un ejemplo de declaración de variables, el contenido de algunas de ellas se visualiza mediante las líneas de código 16 a 18

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS variables1 $$
3 CREATE PROCEDURE variables1 ()
4 BEGIN
5     DECLARE v_entero1, v_entero2, v_entero3 INT;
6     DECLARE v_entero4 INT DEFAULT -40000;
7     /* v_entero5 ENTERO SIN SIGNO */
8     DECLARE v_entero5 INT UNSIGNED DEFAULT 400000000;
9     DECLARE v_real1 FLOAT DEFAULT 344.67;
10    DECLARE v_real2 FLOAT DEFAULT 1.5e14;
11    DECLARE v_real3 NUMERIC(7,2) DEFAULT 4561.44;
12    DECLARE v_caracter1 CHAR(1) DEFAULT 'Y';
13    DECLARE v_caracter2 VARCHAR(20);
14    DECLARE v_fecha1 DATE DEFAULT '1966-11-03';
15    DECLARE v_fecha2 DATE DEFAULT CURRENT_DATE;
16    SELECT v_entero1; -- NULL
17    SELECT v_real3; -- 4561.44
18    SELECT v_fecha2; -- Visualizará la fecha actual
19 END $$
20 DELIMITER ;

```

Procedimiento 2 variables1

Otro ejemplo de declaración de variables. Utilizamos la sentencia de asignación SET para asignar valores a variables como se verá más adelante:

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS variables2 $$
3 CREATE PROCEDURE variables2 ()
4 BEGIN
5     DECLARE v_caracter1 CHAR(1);
6     DECLARE v_forma_pagol ENUM('metálico', 'tarjeta', 'transferencia');
7     --
8     -- SET v_caracter1='SI'; ---> Error por exceder el tamaño
9     SET v_forma_pagol = 1; -- metálico
10    SET v_forma_pagol='tarjeta';
11    -- SET v_forma_pagol='cheque'; ---> Error por no existir dicho valor
12    -- SET v_forma_pagol = 4; ---> Error por no existir dicho índice
13    SET v_forma_pagol='TARJETA'; -- ---> tarjeta
14 END $$
15 DELIMITER ;

```

Procedimiento 3 variables2

De las figuras anteriores se observa:

- Aparecen dos tipos de COMENTARIOS:
 - o Comentarios de una sola línea, precedidos de --
 - o Comentarios de varias líneas, entre /* */, aunque en este caso sólo se extiende el comentario a lo largo de una sola línea.
- Los LITERALES de texto y fecha van encerradas entre ' '.

Las reglas para nombrar variables son bastante flexibles pues a diferencia de otros lenguajes se permite:

1. Nombres largos (más de 255 caracteres).
2. Caracteres especiales.
3. Pueden comenzar con caracteres numéricos.

Todas las variables que se pueden utilizar deben ser escalares, es decir, un solo valor, a diferencia de otros lenguajes que permiten definir variables basadas en tipos de datos compuestos como son los registros, arrays...

Para asignar valores a variables se utiliza la siguiente sintaxis:

`SET nombre_variable1 = expresión1 [, nombre_variable2 = expresión2 ...]`

En este caso y a diferencia de otros lenguajes es necesario especificar la sentencia SET para asignar valores a las variables. Se puede en una sola instrucción realizar varias asignaciones:

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS asigna1 $$
3 CREATE PROCEDURE asigna1 ()
4 BEGIN
5   DECLARE v_entero1, v_entero2, v_entero3 INT;
6   DECLARE v_entero4 INT DEFAULT -40000;
7   DECLARE v_real2 FLOAT DEFAULT 1.5e14;
8   DECLARE v_real3 NUMERIC(7,2) DEFAULT 4561.44;
9   DECLARE v_caracter1 CHAR(1) DEFAULT 'Y';
10  DECLARE v_fecha1 DATE DEFAULT '1966-11-03';
11  DECLARE v_fecha2 DATE DEFAULT CURRENT_DATE;
12  --
13  SET v_caracter1='N';
14  SET v_entero1 = v_entero4 + 10000;
15  SET v_real2 = v_entero4 + v_real3;
16  SET v_fecha1 = '2006/05/29', v_fecha2=v_fecha2+1;
17  --
18  SELECT v_caracter1; -- N
19  SELECT v_entero1; -- -30000
20  SELECT v_real2; -- -35438.6
21  SELECT v_fecha1; -- 2006/05/29
22  SELECT v_fecha2; -- Visualiza la fecha de mañana
23 END $$
24 DELIMITER ;

```

Procedimiento 4 asigna1

2.4.2 Variables de usuario

Un tipo especial de variables son las variables de usuario ya que pueden ser manipuladas dentro y fuera de los programas almacenados. Son una característica de MySQL desde la versión 3. Ejemplo de utilización:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS variables3 $$
3 CREATE PROCEDURE variables3 ()
4 BEGIN
5     SET @v1=@v1 * 2;
6 END $$
7 DELIMITER ;
```

Procedimiento 5 variables3

Puedes probar el procedimiento *variables3* anterior desde el cliente de línea de comandos de MySQL. Accede a él desde dentro de la herramienta *MySQL Query Browser* mediante el comando de menú *Tools / MySQL Command Line Client*.

Este tipo de variables no necesitan declaración y van precedidas del carácter @. Son de un tipo de datos variant y pueden almacenar texto, fechas y números. En las siguientes líneas se crea una variable de usuario de nombre *v1*, se inicializa a 20 y a continuación se llama (CALL) al programa *variables3* que lo que hace es doblar el valor de la variable como se puede comprobar tanto en las líneas de procedimiento como en la ventana de ejecución desde la línea de comandos de MySQL:

```
mysql> USE TEST
Database changed
mysql> SET @v1=20;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL variables3();
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @v1;
+-----+
| @v1 |
+-----+
| 40 |
+-----+
1 row in set (0.00 sec)
```

Su alcance es de una sesión y por tanto son accesibles desde cualquier programa que se ejecuta durante esa sesión, lo que las asemeja al concepto de variables globales como se muestra en el siguiente ejemplo:

```
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS variables4 $$
4 CREATE PROCEDURE variables4 ()
5 BEGIN
6     SET @v1='Raquel';
7     CALL variables5();
8     SELECT @v1;
9 END $$
10 DELIMITER ;
```

Procedimiento 6 variables4

```
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS variables5 $$
4 CREATE PROCEDURE variables5 ()
5 BEGIN
6     SET @v1= CONCAT(@v1, ' y Mario Carrera');
7 END $$
8 DELIMITER ;
```

Procedimiento 7 variables5

Probando los dos procedimientos anteriores:

```
mysql> USE TEST
Database changed
mysql> CALL variables4();
+-----+
| @v1 |
+-----+
| Raquel y Mario Carrera |
+-----+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.02 sec)
```

2.4.3 Parámetros

Los parámetros son variables que se envían y reciben de los programas a los que se llaman.

Se definen en la cláusula CREATE de creación de los procedimientos de la siguiente forma:

```
CREATE PROCEDURE([[IN |OUT |INOUT] nombre_parámetro tipo de datos...])
```

Los tipos de parámetros, según el modo en que se pasan al procedimiento llamado son:

1. **IN:** Opción por defecto. En otros lenguajes representa el modo de paso de parámetros por valor, es decir, el procedimiento almacenado trabaja con una “copia” del parámetro que recibe y por tanto no modifica nada el valor del parámetro que se le pasa al programa almacenado tal y como se puede ver en el siguiente ejemplo y posterior ejecución:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS parametrol $$
3 CREATE PROCEDURE parametrol (IN p_p1 INTEGER )
4 --      EQUIVALE A parametrol (p_p1 INTEGER )
5 BEGIN
6     SET p_p1 = 25;
7 END $$
8 DELIMITER ;
```

Procedimiento 8 parametrol

Ejecutando el procedimiento parametrol:

```
mysql> USE TEST
Database changed
mysql> SET @p1 =10;
Query OK, 0 rows affected (0.02 sec)

mysql> CALL parametrol (@p1);
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT @p1;
+-----+
| @p1 |
+-----+
| 10 |
+-----+
1 row in set (0.00 sec)
```

2. **OUT**: Es una forma de paso de parámetros por variable, es decir, las modificaciones del parámetro dentro del programa almacenado modifican directamente el parámetro pasado como argumento. Hasta que no se le asigne un valor determinado dentro del programa, su valor dentro de él será nulo. Se suelen utilizar como flags o indicadores de cómo ha ido la ejecución de un programa como veremos en el apartado de tratamiento de errores. Ejemplo:

```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS parametro2 $$
4 CREATE PROCEDURE parametro2 (OUT p_p1 INTEGER )
5 BEGIN
6     DECLARE v_v1 INTEGER DEFAULT 300;
7     SET v_v1 = p_p1;
8     SELECT v_v1, p_p1;
9     /* mostrará NULL en ambas columnas puesto que
10    un parámetro OUT no puede ser accedido
11    desde el programa almacenado */
12     SET p_p1 = 33;
13     SET p_p1=p_p1 - 5;
14 END $$
15 DELIMITER ;

```

Procedimiento 9 parametro2

Probando el procedimiento parametro2: Este procedimiento trabajará directamente sobre la variable que se le pase como argumento.

```

mysql> USE TEST
Database changed

mysql> SET @p1=100;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL parametro2(@p1);
+-----+-----+
| v_v1 | p_p1 |
+-----+-----+
| NULL | NULL |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.05 sec)

mysql> SELECT @p1;
+-----+
| @p1 |
+-----+
| 28  |
+-----+
1 row in set (0.00 sec)

```

3. **INOUT**: Otra forma de paso de parámetros por variable pero con la característica de que se le puede pasar un valor inicial que el programa llamado tendrá en cuenta (y no lo considerará NULL como en caso del parámetro OUT). Ejemplo:

```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS parametro3 $$
4 CREATE PROCEDURE parametro3 (INOUT p_pl DECIMAL(7,2) )
5 BEGIN
6     SET p_pl=p_pl / 166.386;
7 END $$
8 DELIMITER ;

```

Procedimiento 10 parametro3

Ejecutando el procedimiento parametro3:

```

mysql> SET @p1=1000;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL parametro3(@p1);
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT @p1;
+-----+
| @p1   |
+-----+
| 6.01  |
+-----+
1 row in set (0.00 sec)

```

2.4.4 Principales Operadores

Similares a los de otros lenguajes de programación. Se utilizan en la mayoría de los casos junto con la sentencia SET para asignar valores a variables, formando parte de expresiones de comparación y en bucles.

OPERADORES MATEMÁTICOS

+	Suma
-	Resta
*	Multiplicación
/	División
DIV	División entera
%	Resto

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS operadores1 $$
4 CREATE PROCEDURE operadores1 ()
5 BEGIN
6     DECLARE a INT DEFAULT 2 ;
7     DECLARE b INT DEFAULT 256 ;
8     DECLARE c FLOAT DEFAULT 440.56 ;
9     --
10    SET c = c + b / a ;
11    --
12    SELECT c;          -- 568.56
13    SELECT c/2*3;      -- 852.83999633789
14    SELECT b%a;        -- 0
15 END $$
16 DELIMITER ;

```

Procedimiento 11 operadores1

OPERADORES DE COMPARACIÓN.

Comparan dos valores y devuelven como resultado **CIERTO** (1), **FALSO** (0) O **NULO** (NULL)

OPERADOR	SIGNIFICADO	EJEMPLO Y RESULTADO
>	Mayor que	3 > 2 → Cierto
<	Menor que	4 < 6 → Cierto
<=	Menor o igual que	4 <= 3 → Falso
>=	Mayor o igual que	4 >= 3 → Cierto
=	Igual a	4 = 4 → Cierto
<> !=	Distinto de	4 <> 4 → Falso
<=>	Comparación de valores nulos. Devuelve cierto si ambos valores son nulos	NULL <=> NULL → Cierto
BETWEEN	Comprendido entre dos valores	45 BETWEEN 25 AND 50 → Cierto
IS NULL	Si valor nulo	3 IS NULL → Falso
IS NOT NULL	Si valor no nulo	3 IS NOT NULL → Cierto
NOT BETWEEN	No comprendido entre dos valores	45 NOT BETWEEN 25 AND 50 → Falso
IN	Pertenencia al conjunto o lista	45 IN (44, 45, 46) → Cierto
NOT IN	No pertenencia al conjunto o lista	45 NOT IN (44, 45, 46) → Falso
LIKE	Coincidencia con patrón de búsqueda	"ALBERTO CARRERA" LIKE "%CARRERA" → Cierto

TABLA DE VERDAD DEL OPERADOR LÓGICO AND

AND (&&)	CIERTO	FALSO	NULL
CIERTO	CIERTO	FALSO	NULL
FALSO	FALSO	FALSO	NULL
NULL	NULL	NULL	NULL

TABLA DE VERDAD DEL OPERADOR LÓGICO OR

OR ()	CIERTO	FALSO	NULL
CIERTO	CIERTO	CIERTO	CIERTO
FALSO	CIERTO	FALSO	NULL
NULL	CIERTO	NULL	NULL

TABLA DE VERDAD DEL OPERADOR LÓGICO NOT

NOT (!)	CIERTO / TRUE (1)	FALSO / FALSE (0)	NULL
	FALSO (0)	CIERTO (1)	NULL

2.4.5 Expresiones

Se trata de una combinación de literales, variables y operadores que se evalúan para devolver un valor.

Ver líneas 12 a 14 del procedimiento anterior *operadores1*.

2.4.6 Funciones incorporadas

En los programas almacenados pueden seguirse utilizando la mayoría de las funciones incluidas en MySQL y que se utilizan para formar las sentencias SQL, excepto las que trabajan con grupos de datos (cláusula GROUP BY) puesto que las variables en los programas almacenados son escalares y almacenan un solo valor. Por eso funciones como SUM, COUNT, MIN, MAX y AVG pueden emplearse en programas almacenados siempre y cuando devuelvan una fila y no varias (como consecuencia p.ej. en este último caso de utilizar la cláusula GROUP BY).

A continuación se detallarán las más importantes. Para más información consultar el manual.

FUNCIONES MATEMÁTICAS		
FUNCIÓN	DEVUELVE	EJEMPLO Y RESULTADO
ABS(num)	Valor absoluto de num	SELECT ABS(-3) → 3
SIGN(num)	-1, 0 o 1 en función del valor de num	SELECT SIGN(2), SIGN(-2), SIGN(0) → 1, -1, 0
MOD(num1, num2)	Resto de la división de num1 por num2	SELECT MOD (5,2) → 1
FLOOR(num)	Mayor valor entero inferior a num	SELECT FLOOR(23.9) → 23
CEILING(num)	Menor valor entero superior a num	SELECT CEILING(23.9) → 24
ROUND(num)	Redondeo entero más próximo	SELECT ROUND(23.5), ROUND(23.4); → 24 23
ROUND(num,d)	Redondeo a d decimales más próximo	SELECT ROUND(23.545,2), ROUND(23.44,1) → 23,55 23,4
TRUNCATE (num, d)	Num truncado a d decimales	SELECT TRUNCATE (22.89, 1), TRUNCATE (15326,-3) → 22,8 5000
POW(num1, num2)	Num1 elevado a la num2 potencia	SELECT POW(2,5) → 32
SQRT (num)	Raíz cuadrada de num	SELECT SQRT(36) → 6

FUNCIONES DE CADENA		
FUNCIÓN	DEVUELVE	EJEMPLO Y RESULTADO
LIKE(plantilla)	Resultado de comparar una cadena con una plantilla	SELECT 'ALBERTO' LIKE 'ALBER%' → 1 (cierto)
NOT LIKE (plantilla)	Lo contrario a la fila anterior	SELECT 'ALBERTO' NOT LIKE 'ABIERTO' → 1
_ (subrayado)	Se trata de un comodín que reemplaza un carácter en una cadena	SELECT 'ALBERTO' LIKE 'ALBERT_' → 1
%	Como el caso anterior pero para uno o más caracteres	SELECT 'ALBERTO' LIKE 'ALBER%' → 1 (cierto)
\	Como en otros lenguajes se trata del carácter de escape, si precede al comodín elimina su función y lo trata como un carácter más	SELECT '30%' LIKE '30\%' → 1

FUNCIONES DE CADENA (Continuación)

BINARY	Por defecto en las comparaciones entre cadenas no se distingue mayúsculas de minúsculas salvo que se indique esta opción	<code>SELECT 'ALBERTO' LIKE BINARY 'Alberto' → 0</code> (falso)
STRCMP(cad1, cad2)	-1 si cad1 < cad2, 0 si cad1=cad2 o 1 si cad1 > cad2	<code>SELECT STRCMP('ALBERTO', 'ABIERTO') → 1</code>
UPPER(cad)	La cadena cad en mayúsculas	<code>SELECT UPPER('Alberto') → 'ALBERTO'</code>
LOWER(cad)	La cadena cad en minúsculas	<code>SELECT LOWER('Alberto') → 'alberto'</code>

FUNCIONES DE FECHA

FUNCIÓN	DEVUELVE	EJEMPLO Y RESULTADO
NOW()	Fecha y hora según el formato 'aaaa-mm-dd hh:mm:ss'	<code>SELECT NOW() → 2006-08-01 00:40:25</code>
DAYOFWEEK(fecha)	Cifra que representa el día de la semana (1 – domingo, 2 –lunes...)	<code>SELECT DAYOFWEEK('1966-11-03') → 5</code>
WEEKDAY(fecha)	Ídem de DAYOFWEEK pero con otros valores: 0 – lunes, 1 – martes...	<code>SELECT WEEKDAY('1966-11-03') → 3</code>
DAYOFMONTH(fecha)	Día del mes (entre 1 y 31)	<code>SELECT DAYOFMONTH('1966-11-03') → 3</code>
DAYOFYEAR(fecha)	Día del año (entre 1 y 366)	<code>SELECT DAYOFYEAR('1966-11-03') → 307</code>
MONTH(fecha);	Mes del año (entre 1 y 12)	<code>SELECT MONTH('1966-11-03') → 11</code>
DAYNAME(fecha)	Nombre del día de la fecha	<code>SELECT DAYNAME('1966-11-03') → 'Thursday'</code>
MONTHNAME(fecha)	Nombre del mes	<code>SELECT MONTHNAME('1966-11-03') → 'November'</code>
QUARTER(fecha)	Trimestre del año (entre 1 y 4)	<code>SELECT QUARTER('1966-11-03') → 4</code>
WEEK(fecha [,inicio])	Semana del año (entre 1 y 52). Inicio especifica el comienzo de la semana. Si no se especifica vale 0 (domingo). Para empezar el lunes utilizar el 1	<code>SELECT WEEK('2006-12-20',1) → 51</code>
YEAR(fecha)	Año (entre 1000 y 9999)	<code>SELECT YEAR('2006-12-20') → 2006</code>
HOUR(fecha)	La hora	<code>SELECT HOUR(NOW()) → 1</code>
MINUTE(fecha)	Los minutos	<code>SELECT MINUTE(NOW()) → 5</code>
SECOND(fecha)	Los segundos	<code>SELECT SECOND(NOW()) → 58</code>

FUNCIONES DE FECHA *(Continuación)*

TO_DAYS(fecha)	Número de días transcurridos desde el año 0 hasta la fecha	SELECT TO_DAYS('2006-08-01') → 732889
DATE_ADD(fecha, INTERVAL valor tipo de intervalo)	La fecha sumado el intervalo especificado	SELECT DATE_ADD('2006-08-01', INTERVAL 1 MONTH) → '2006-09-01'
DATEDIFF(fecha1, fecha2)	El número de días transcurridos entre fecha1 y fecha2	SELECT DATEDIFF('2006-08-01', '2006-07-26') → 6
CURDATE()	Fecha actual según el formato 'aaaa-mm-dd'	SELECT CURRENT_DATE() → '2006-08-01'
CURRENT_DATE()	Fecha actual según el formato 'hh:mm:ss'	SELECT CURRENT_TIME() → '01:12:43'
CURTIME()	Fecha actual según el formato 'hh:mm:ss'	SELECT CURRENT_TIME() → '01:12:43'
CURRENT_TIME()	Devuelve la fecha en el formato especificado. Consultar el manual para las posibilidades de la opción formato.	SELECT DATE_FORMAT(NOW(), 'Hoy es %d de %M de %Y') → 'Hoy es 01 de August de 2006'
DATE_FORMAT (fecha, formato)		

FUNCIONES DE CONTROL

FUNCIÓN	DESCRIPCIÓN	EJEMPLO	Y RESULTADO
IF(expr1, expr2, expr3)	Si la expresión expr1 es cierta, devuelve expr2, sino expr3	SET @A=20; SET @B=15; SELECT IF(@A<@B, @A+@B, @A - @B); → 5	
IFNULL(expr1, expr2)	Si la expresión expr1 es NULL devuelve expr2, sino expr1	SET @A=20; SELECT IFNULL(@A, 0); → 20	
NULLIF(expr1, expr2)	Si la expresión expr1 es igual a expr2, devuelve NULL sino expr1	SET @A=20; SET @B=15; SELECT NULLIF(@B, @A); → 15	
CASE valor WHEN comp1 THEN res1 [WHEN comp2 THEN res2] [ELSE reselse] END	Compara el valor con cada una de las expresiones comp. Si se verifica la igualdad entonces devuelve el valor res asociado, en cualquier otro caso devuelve reselse	SELECT CASE WEEKDAY(NOW()) WHEN 5 THEN 'Fin de semana' WHEN 6 THEN 'Fin de semana' ELSE 'No es fin de semana' END;	

FUNCIONES DE AGREGACIÓN			
FUNCIÓN	DEVUELVE	EJEMPLO	Y RESULTADO
AVG(columna)	Media de los valores de la columna especificada	SELECT AVG(salario) FROM empleados	→ 302.9412
COUNT (columna *)	Número de valores no nulos de la columna (si esta se especifica como argumento). Utilizando el carácter * devuelve el número total de valores incluyendo los nulos	SELECT COUNT(comision) FROM empleados	→ 14 SELECT COUNT(*) FROM empleados → 34 (luego 20 trabajadores no tienen comisión)
MIN(columna)	Valor mínimo de la columna	SELECT MIN(salario) FROM empleados	→ 100
MAX(columna)	Valor máximo de la columna	SELECT MAX(salario) FROM empleados	→ 720
SUM(columna)	Suma de valores contenidos en la columna	SELECT SUM(salario) FROM empleados	→ 10300

OTRAS FUNCIONES			
FUNCIÓN	DESCRIPCIÓN	EJEMPLO	Y RESULTADO
CAST (expresión AS tipo) CONVERT (expresión, tipo)	Convierte la expresión al tipo indicado	SELECT CONVERT(20060802, DATE)	→ '2006-08-02'
LAST_INSERT_ID()	Devuelve el valor creado por una columna de tipo AUTO_INCREMENT en la última inserción	SELECT LAST_INSERT_ID()	→ 0
VERSION()	Devuelve la versión del servidor MySQL	SELECT VERSION()	→ '5.0.20a-nt'
CONNECTION_ID()	Devuelve el identificador de conexión	SELECT CONNECTION_ID()	→ 4
DATABASE()	Devuelve la base de datos actual	SELECT DATABASE()	→ 'test'
USER()	Devuelve el usuario actual	SELECT USER()	→ root@localhost

2.4.7 Bloques de instrucciones

Hasta ahora hemos estado trabajando con procedimientos con un solo bloque de instrucciones, comenzando con la sentencia BEGIN y terminando con la sentencia END:

```
CREATE {PROCEDURE | FUNCTION | TRIGGER} nombre_del_programa
BEGIN
    Instrucciones
END;
```

Este es el caso más sencillo pues muchos programas almacenados MySQL contienen varios bloques agrupando un conjunto de instrucciones y comenzando cada uno de ellos con la instrucción BEGIN y finalizando con la END. Con la estructura de bloques se consigue reunir las instrucciones en agrupaciones lógicas que realizan una determinada función, como por ejemplo los bloques de los manejadores de errores que se tratan en este curso. También se consigue delimitar el ámbito de las variables, declarándolas y definiéndolas dentro de un bloque interno, de este modo no son visibles fuera de él. En cambio una variable externa al bloque interno será accesible también desde dentro de este último. En el caso en que la variable externa al bloque y la interna tuvieran el mismo nombre, dentro del bloque interno se estará referenciando a la variable interna.

Un bloque no solo agrupa instrucciones sino también otros elementos que se tratan en este curso como:

1. Declaración de variables y condiciones.
2. Declaración de cursores.
3. Declaración de manejadores de error.
4. Código de programa.

En el momento de declararse, para evitar mensajes de error, debe seguirse el orden anterior, comenzando en primer lugar por la declaración de variables.

Como ocurre con otros lenguajes de programación, los bloques pueden etiquetarse. Esta forma de actuar garantiza una fácil lectura del procedimiento y por tanto de su mantenimiento y permite abandonar el bloque antes de que este concluya si así fuera necesario (sentencia LEAVE). Sintaxis:

```
[etiqueta:] BEGIN
Declaración de variables y condiciones.
Declaración de cursores.
Declaración de manejadores de error.
Código de programa.
END [etiqueta];
```

Ejemplos de aplicación:

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bloque1 $$
4 CREATE PROCEDURE bloque1 ()
5 BEGIN
6     DECLARE v_externa VARCHAR(45) DEFAULT 'Variable visible en todo el procedimiento';
7     BEGIN
8         DECLARE v_interna VARCHAR(45);
9         SET v_interna='Sólo soy accesible en las líneas 7 a 11';
10        SELECT v_externa; -- Correcto
11    END;
12    SELECT v_interna; -- Error
13 END $$
14 DELIMITER ;
```

Procedimiento 12 bloque1

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bloque2 $$
4 CREATE PROCEDURE bloque2 ()
5 BEGIN
6     DECLARE v VARCHAR(65) DEFAULT 'Variable visible en todo el procedimiento';
7     BEGIN
8         SET v='Cambio del valor de la variable en el bloque interno';
9     END;
10    SELECT v; -- 'Cambio del valor de la variable en el bloque interno'
11 END $$
12 DELIMITER ;

```

Procedimiento 13 bloque2

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS bloque3 $$
3 CREATE PROCEDURE bloque3 ()
4 BEGIN
5     DECLARE v INT DEFAULT 500;
6     BEGIN
7         DECLARE v INT;
8         SET v = 200;
9         SELECT v; -- 200
10    END;
11    SELECT v; -- 500
12 END $$
13 DELIMITER ;

```

Procedimiento 14 bloque3

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS bloque4 $$
3 CREATE PROCEDURE bloque4 ()
4     bloque_externo: BEGIN
5         DECLARE v INT DEFAULT 100;
6         bloque_interno: BEGIN
7             IF v < 500 THEN
8                 LEAVE bloque_interno;
9             END IF;
10            SELECT 'No llega a ejecutarse esta instrucción';
11        END bloque_interno;
12        SELECT 'Fin del bloque externo';
13    END bloque_externo $$
14 DELIMITER ;

```

Procedimiento 15 bloque4

2.4.8 El comando IF

Igual en su funcionamiento que otros lenguajes de programación. Ejecuta la acción cuya expresión (condición) es cierta (de no ser cierta ninguna entonces ejecuta la acción asociada a la sentencia ELSE).

Sintaxis:

```

IF condición1 THEN instrucciones
    [ELSEIF condición2 THEN instrucciones ....]
    [ELSE instrucciones]
END IF;

```

Ejemplo:

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS condicional1 $$
4 CREATE PROCEDURE condicional1 ()
5 BEGIN
6     DECLARE v_edad TINYINT UNSIGNED DEFAULT 47;
7     IF v_edad <= 12 THEN
8         SELECT 'Niño';
9     ELSEIF v_edad <= 30 THEN
10        SELECT 'Joven';
11    ELSE
12        SELECT 'Adulto';
13    END IF;
14 END $$
15
16 DELIMITER ;
```

Procedimiento 16 condicional1

El valor inicial (línea 6) de la variable `v_edad` conduce a que la instrucción que se ejecute sea la de la línea 12 al no cumplir ninguna de las dos condiciones anteriores.

La forma de actuar es la siguiente: Si la expresión de la línea 7 es cierta, se ejecutará la instrucción de la línea 8, sino si la expresión de la línea 9 es cierta entonces se ejecutará la instrucción de la línea 10; si no son ciertas ninguna de las dos expresiones entonces se ejecutará la instrucción de la línea 12.

El ejemplo que viene a continuación nos sirve para ver una versión más “abreviada” de la sentencia condicional que la del ejemplo anterior así como para señalar el comportamiento de la sentencia condicional ante valores nulos.

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS condicional2 $$
4 CREATE PROCEDURE condicional2 ()
5 BEGIN
6     DECLARE v_edad TINYINT UNSIGNED DEFAULT NULL;
7     IF v_edad <= 30 THEN
8         SELECT 'Joven';
9     ELSE
10        SELECT 'Adulto';
11    END IF;
12 END $$
13
14 DELIMITER ;
```

Procedimiento 17 condicional2

En este caso se ejecutará la instrucción 10 puesto que la edad contiene un valor nulo que no hace cierta la expresión de la línea 7. Hay que tener cuidado pues se está asignando el calificativo de “adulto” a una persona de la que se desconoce su edad. Para estos casos puede utilizarse la forma más simple de la sentencia condicional:

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS condicional3 $$
4 CREATE PROCEDURE condicional3 ()
5 BEGIN
6     DECLARE v_edad TINYINT UNSIGNED DEFAULT NULL;
7     IF v_edad <= 30 THEN
8         SELECT 'Joven';
9     END IF;
10    IF v_edad > 30 THEN
11        SELECT 'Adulto';
12    END IF;
13 END $$
14
15 DELIMITER ;

```

Procedimiento 18 condicional3

2.4.9 El comando CASE

Similar a la sentencia condicional IF anterior. Todo lo que se puede expresar con la sentencia IF se puede expresar con la sentencia CASE. Esta última se suele utilizar cuando el número de expresiones o condiciones a evaluar es elevado y por tanto la lectura del código se hace más fácil y agradable.

Existen 2 formas posibles si lo que se evalúa es una expresión o una condición:

```

CASE expresión
    WHEN valor1 THEN
        instrucciones
    [WHEN valor2 THEN
        instrucciones ...]
    [ELSE
        instrucciones]
END CASE;

```

En el ejemplo siguiente, tras evaluar la expresión de la línea 8 se ejecutará la instrucción de la línea 10. El comportamiento ante una expresión nula es exactamente el mismo que para la sentencia IF. Si el valor en la línea 7 de la variable fuera NULL, el resultado del procedimiento sería el de la línea 14 siendo que no se corresponde por tanto con la opción de pago elegida.

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS condicional4 $$
4 CREATE PROCEDURE condicional4 ()
5 BEGIN
6     DECLARE v_forma_pago ENUM ('metalico', 'tarjeta', 'transferencia');
7     SET v_forma_pago = 'metalico';
8     CASE v_forma_pago
9         WHEN 'metalico' THEN
10            SELECT 'Forma de pago elegida: metalico';
11        WHEN 'tarjeta' THEN
12            SELECT 'Forma de pago elegida: tarjeta';
13        ELSE
14            SELECT 'Forma de pago elegida: transferencia';
15        END CASE;
16
17 END $$
18
19 DELIMITER ;

```

Procedimiento 19 condicional4

La otra sintaxis es similar a la anterior pero con condiciones en lugar de expresiones:

```
CASE
    WHEN condición1 THEN
        instrucciones
    [WHEN condición2 THEN
        instrucciones ...]
    [ELSE
        instrucciones]
END CASE;
```

Ejemplo:

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS condicional5 $$
4 CREATE PROCEDURE condicional5 ()
5 BEGIN
6     DECLARE v_edad TINYINT UNSIGNED DEFAULT 7;
7     CASE
8         WHEN v_edad <= 12 THEN
9             SELECT 'Niño';
10        WHEN v_edad <= 30 THEN
11            SELECT 'Joven';
12        ELSE
13            SELECT 'Adulto';
14        END CASE;
15    END $$
16
17 DELIMITER ;
```

Procedimiento 20 condicional5

En el ejemplo anterior se ejecutaría la instrucción 8.

Sólo se ejecuta la instrucción o instrucciones asociadas a una expresión o condición (no hace falta por tanto una instrucción de salida o de ruptura como ocurre en otros lenguajes). Una vez que se ejecuta se finaliza la instrucción CASE y la ejecución del programa continua por la línea siguiente.

2.4.10 El comando LOOP, LEAVE e ITERATE

Sintaxis

```
[etiqueta:] LOOP
    instrucciones
END LOOP [etiqueta];
```

Todas las instrucciones comprendidas entre las palabras reservadas LOOP Y END LOOP (bucle), se ejecutan un número de veces hasta que la ejecución del bucle se encuentra con la instrucción:

```
LEAVE etiqueta
```

en ese momento se abandona el bucle.

En el siguiente ejemplo la instrucción 10 se ejecutará 4 veces:

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bucle1 $$
4 CREATE PROCEDURE bucle1 ()
5 BEGIN
6     DECLARE i TINYINT UNSIGNED;
7     SET i=0;
8     mibucle: LOOP
9         SET i=i+1;
10        SELECT 'Valor de i= ' + i AS i;
11        IF i=4 THEN
12            LEAVE mibucle;
13        END IF;
14    END LOOP mibucle;
15 END $$
16
17 DELIMITER ;

```

Procedimiento 21 bucle1

La sentencia:

ITERATE *etiqueta*

se utiliza para forzar que la ejecución del bucle termine en el momento donde se encuentra con esta instrucción y continúe por el principio del bucle. De esta manera en el ejemplo que viene a continuación se ejecutará 3 veces la instrucción de la línea 13 (para los valores de i del 1 al 4 excepto el 3)

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bucle2 $$
4 CREATE PROCEDURE bucle2 ()
5 BEGIN
6     DECLARE i TINYINT UNSIGNED;
7     SET i=0;
8     mibucle: LOOP
9         SET i=i+1;
10        IF i=3 THEN
11            ITERATE mibucle;
12        END IF;
13        SELECT 'Valor de i= ' + i AS i;
14        IF i=4 THEN
15            LEAVE mibucle;
16        END IF;
17    END LOOP mibucle;
18
19 END $$
20
21 DELIMITER ;

```

Procedimiento 22 bucle2

2.4.11 El comando REPEAT ... UNTIL

Sintaxis:

```

[etiqueta:] REPEAT
    instrucciones
UNTIL expresión
END REPEAT [etiqueta]

```

Las instrucciones se ejecutarán hasta que sea cierta la expresión. Por lo menos el conjunto de instrucciones se ejecuta una vez pues la evaluación de la expresión se hace posterior a la ejecución de las instrucciones. El mismo procedimiento *bucle1* anterior pero utilizando esta otra instrucción quedaría:

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bucle3 $$
4 CREATE PROCEDURE bucle3 ()
5 BEGIN
6     DECLARE i TINYINT UNSIGNED;
7     SET i=0;
8     mibucle: REPEAT
9         SET i=i+1;
10        SELECT 'Valor de i= ' + i AS i;
11    UNTIL i=4
12    END REPEAT mibucle;
13 END $$
14
15 DELIMITER ;

```

Procedimiento 23 bucle3

Equivale a la sentencia iterativa vista anteriormente

```

etiqueta: LOOP
    instrucciones
    IF expresión THEN LEAVE etiqueta; END IF;
END LOOP etiqueta;

```

Podría utilizarse la sentencia *ITERATE* pero puede llevar a situaciones contradictorias si llegara a ejecutar otra vez el conjunto de instrucciones aun habiendo hecha cierta la expresión de finalización del bucle.

La sentencia *LEAVE* también puede utilizarse.

2.4.12 El comando WHILE

Sintaxis:

```

[etiqueta:] WHILE expresión DO
    instrucciones
END WHILE [etiqueta]

```

Ejecuta el conjunto de instrucciones mientras sea cierta la expresión. Ejemplo:

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bucle4 $$
4 CREATE PROCEDURE bucle4 ()
5 BEGIN
6     DECLARE i TINYINT UNSIGNED;
7     SET i=0;
8     mibucle: WHILE i < 4 DO
9         SET i=i+1;
10        SELECT 'Valor de i= ' + i AS i;
11    END WHILE mibucle;
12 END $$
13
14 DELIMITER ;

```

Procedimiento 24 bucle4

Equivale a la instrucción iterativa anterior:

```
etiqueta: LOOP
    IF !expresión THEN LEAVE etiqueta; END IF;
    resto de instrucciones;
END LOOP etiqueta;
```

2.4.13 Bucles anidados

Consiste en utilizar comandos de repetición dentro de otros. Ejemplo de utilización:

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS bucle5 $$
4 CREATE PROCEDURE bucle5 ()
5 BEGIN
6     DECLARE i,j TINYINT UNSIGNED DEFAULT 1;
7
8     bucle_externo: LOOP
9         SET j=1;
10        bucle_interno: LOOP
11            SELECT CONCAT("Valor de i y j: ",i, "-", j) AS i_j;
12            SET j=j+1;
13            IF j>2 THEN
14                LEAVE bucle_interno;
15            END IF;
16        END LOOP bucle_interno;
17        SET i=i+1;
18        IF i>2 THEN
19            LEAVE bucle_externo;
20        END IF;
21    END LOOP bucle_externo;
22
23 END $$
24
25 DELIMITER ;
```

Procedimiento 25 bucle5

Al ejecutar el ejemplo anterior se visualizarán los valores i-j: 1-1, 1-2, 2-1, 2-2

Como se ha indicado en el apartado anterior 2.4.7 de bloques de instrucciones, es importante etiquetar el comienzo y final del bucle no solo por la instrucción LEAVE o ITERATE sino por claridad en el seguimiento del código.

2.5 Procedimientos

2.5.1 Creación de procedimientos. Diccionario de Datos

La sintaxis completa de creación de procedimientos es:

```
CREATE PROCEDURE nombre_procedimiento ([parametro1[,...]])
    [LANGUAGE SQL]
    [ [NOT] DETERMINISTIC]
    [ {CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA | NO SQL} ]
    [SQL SECURITY {DEFINER | INVOKER} ]
    [COMMENT comentario]
    bloque_de_instrucciones_del_procedimiento
```

Para crear un procedimiento o función el usuario debe disponer del privilegio `CREATE ROUTINE` y el de `ALTER ROUTINE` para modificarlo o borrarlo; este último privilegio se asigna automáticamente al creador del procedimiento o función. Para poder ejecutar un procedimiento o función debe disponerse del privilegio `EXECUTE`.

Aspectos a tener en cuenta:

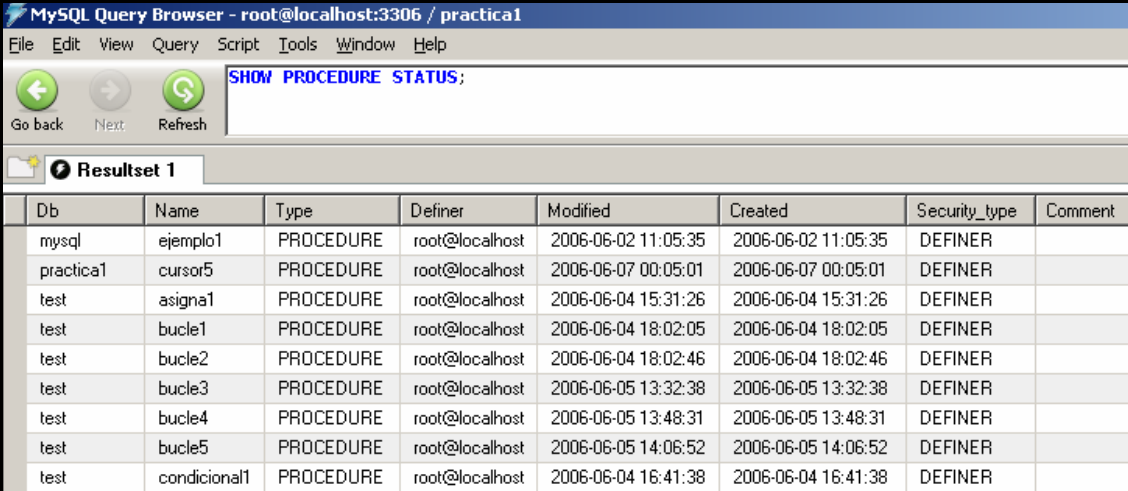
1. El nombre del procedimiento cumple con las normas vistas en este capítulo referidas a nombrar variables. La lista de argumentos suministrados al procedimiento también ha sido estudiada.
2. La cláusula `LANGUAGE SQL` indica que el lenguaje utilizado en los procedimientos cumple el estándar `SQL:PSM`. Es una cláusula innecesaria en este momento pues los procedimientos en MySQL sólo soportan este estándar. Si MySQL en un futuro aceptara la escritura de procedimientos almacenados en otros lenguajes como C o Java entonces ya sería necesario indicar el lenguaje de programación utilizado en el procedimiento.
3. `[NOT] DETERMINISTIC`. Referido al comportamiento del procedimiento. Si un procedimiento es `DETERMINISTIC`, siempre ante una misma entrada devuelve una misma salida. Funciones numéricas como el valor absoluto, cuadrado, raíz cuadrada son `DETERMINISTIC` pues siempre devuelven el mismo resultado ante un mismo valor. Por otro lado, una función que devolviera el número de días transcurridos desde 1900 hasta la fecha sería `NOT DETERMINISTIC` pues cambia según el día que se ejecuta. Por defecto la opción es `NOT DETERMINISTIC`. Al igual que la anterior se puede prescindir de su utilización debido a que por el momento no es tenida en cuenta por el servidor. Más adelante podrá ser utilizada para la optimización de consultas.
4. `[{CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA | NO SQL}]` Indica el tipo de acceso que realizará el procedimiento a la base de datos, si sólo se va a leer datos se especificará la cláusula `READS SQL DATA`, si además de ello los modifica entonces la cláusula `MODIFIES SQL DATA` será la que se emplee. Si el procedimiento no realiza ningún tipo de acceso a la base de datos puede utilizarse la cláusula `NO SQL`. Por defecto se utiliza la opción `CONTAINS SQL` que indica que el procedimiento contiene consultas SQL. Estos parámetros se utilizan para mejorar el rendimiento.
5. `[SQL SECURITY {DEFINER | INVOKER}]` Indica si el procedimiento almacenado se ejecutará con los permisos del usuario que lo creó (`DEFINER`) o con los permisos del usuario que llama al procedimiento (`INVOKER`). La opción por defecto es `DEFINER`. Hay que tener muy en cuenta que con la opción `DEFINER` un usuario que lance el procedimiento podrá acceder a los datos aunque no posea los privilegios sobre las tablas que almacenan dichos datos; se trata de un mecanismo de acceso a los procedimientos sin dar directamente acceso a los datos.
6. `[COMMENT comentario]` Comentario sobre el procedimiento que puede ayudar al usuario a conocer/recordar el funcionamiento del procedimiento. Esta información puede consultarse como se ha visto anteriormente en el diccionario de datos. Puede prescindirse de dicha cláusula y realizar la escritura de los comentarios aclaratorios al principio del bloque de instrucciones del procedimiento utilizando los caracteres `/* */` o `--`

Para conocer toda la información sobre los procedimientos almacenados se procederá a consultar el diccionario de datos de manera similar a cómo se ha hecho anteriormente con otros objetos como las vistas. Se incluye también información para las funciones que se tratan más adelante.

Sintaxis:

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE patrón]
```

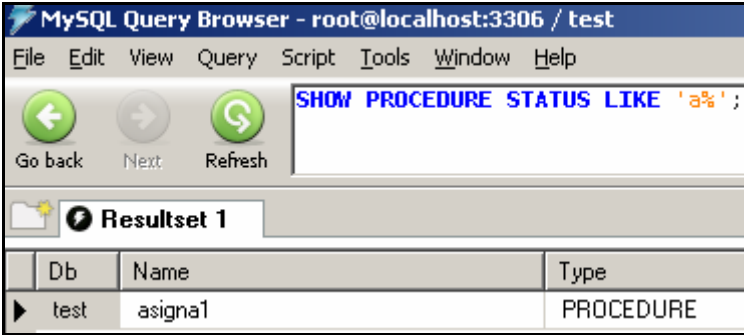
Ejemplo:



Db	Name	Type	Definer	Modified	Created	Security_type	Comment
mysql	ejemplo1	PROCEDURE	root@localhost	2006-06-02 11:05:35	2006-06-02 11:05:35	DEFINER	
practical1	cursor5	PROCEDURE	root@localhost	2006-06-07 00:05:01	2006-06-07 00:05:01	DEFINER	
test	asigna1	PROCEDURE	root@localhost	2006-06-04 15:31:26	2006-06-04 15:31:26	DEFINER	
test	bucle1	PROCEDURE	root@localhost	2006-06-04 18:02:05	2006-06-04 18:02:05	DEFINER	
test	bucle2	PROCEDURE	root@localhost	2006-06-04 18:02:46	2006-06-04 18:02:46	DEFINER	
test	bucle3	PROCEDURE	root@localhost	2006-06-05 13:32:38	2006-06-05 13:32:38	DEFINER	
test	bucle4	PROCEDURE	root@localhost	2006-06-05 13:48:31	2006-06-05 13:48:31	DEFINER	
test	bucle5	PROCEDURE	root@localhost	2006-06-05 14:06:52	2006-06-05 14:06:52	DEFINER	
test	condicional1	PROCEDURE	root@localhost	2006-06-04 16:41:38	2006-06-04 16:41:38	DEFINER	

Ilustración 8. Información sobre los procedimientos creados

Utilizando un patrón de búsqueda:



Db	Name	Type
test	asigna1	PROCEDURE

Ilustración 9. Información sobre determinados procedimientos

El comando:

```
SHOW CREATE [PROCEDURE | FUNCTION] nombre
```

permite ver información de los procedimientos y funciones así como las líneas de código (al igual que con vistas seleccionar columna *Create Procedure* – botón derecho del ratón – Comando *View Field in Popup Editor*):

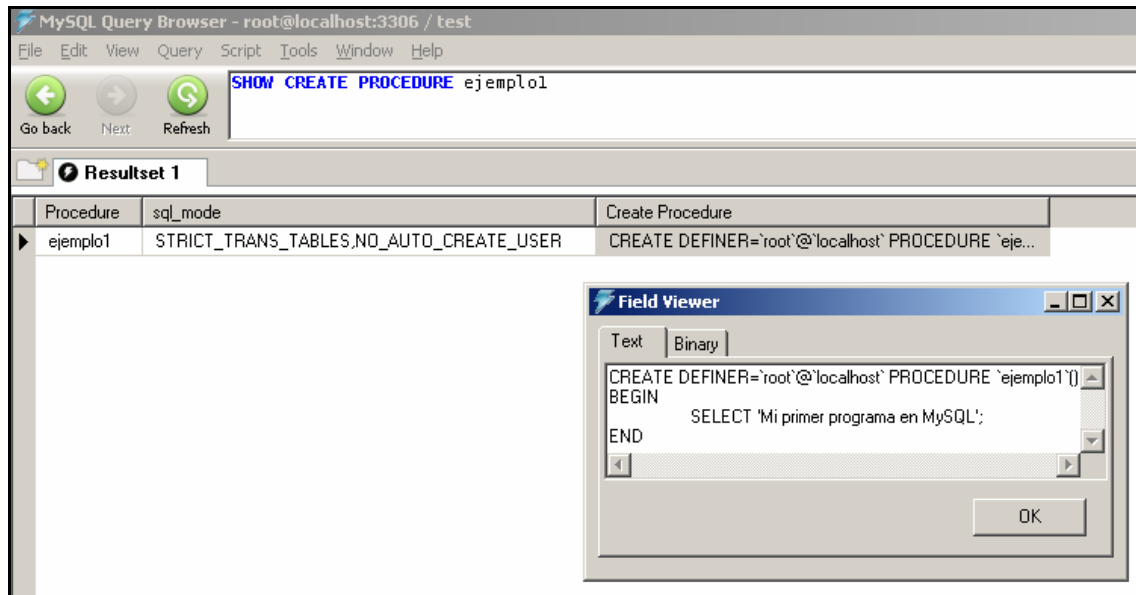


Ilustración 10. Información sobre el código de los procedimientos

La otra forma de ver toda la información de un procedimiento es consultando directamente el DICCIONARIO DE DATOS.

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES;
```

Instrucción 1

La tabla (vista) ROUTINES aporta más información que la suministrada por los comandos SHOW PROCEDURE STATUS y SHOW CREATE.

2.5.2 Modificación de procedimientos

Sintaxis:

```
ALTER PROCEDURE nombre_procedimiento
    {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
    SQL SECURITY {DEFINER|INVOKER}
    COMMENT comentario
```

Se debe poseer el privilegio de ALTER ROUTINE para poder modificar procedimientos y funciones. El uso de las distintas opciones se ha expuesto en el apartado de creación.

2.5.3 Borrado de procedimientos

Sintaxis:

```
DROP PROCEDURE [IF EXISTS] nombre_procedimiento
```

Al igual que para la modificación, se debe poseer el privilegio de ALTER ROUTINE para poder borrar procedimientos y funciones.

2.5.4 Utilización de instrucciones DDL y DML en procedimientos almacenados

Junto con la recuperación de datos (se expone en el siguiente punto), se trata de una de las más importantes ventajas del empleo de programas almacenados (en este caso procedimientos).

En el ejemplo siguiente, en las líneas 7 a 11 nos encontramos con instrucciones DDL (lenguaje definición de datos) cuya finalidad es la creación de una tabla. Por otro lado, la instrucción DML (lenguaje de manipulación de datos) de la línea 14 inserta un alumno de código *i* y nombre “alumno *i*” cada vez que se ejecuta dicha instrucción dentro del bucle (5 veces en total).

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS procedimientol $$
4 CREATE PROCEDURE procedimientol ()
5 BEGIN
6     DECLARE i TINYINT UNSIGNED DEFAULT 1;
7     DROP TABLE IF EXISTS alumnos ;
8     CREATE TABLE alumnos
9         (id          INT PRIMARY KEY,
10          alumno     VARCHAR(30))
11     ENGINE=innodb;
12 --
13     WHILE (i<=5) DO
14         INSERT INTO alumnos VALUES(i,CONCAT("alumno ",i));
15         SET i=i+1;
16     END WHILE;
17
18 END $$
19
20 DELIMITER ;

```

Procedimiento 26 – “procedimientol”

Puede verse el resultado de la ejecución del procedimiento y la tabla creada con sus 5 filas refrescando la base de datos *test* de la pestaña *Schemata* (clic con el botón derecho del ratón más opción *Refresh*) y haciendo *doble clic* sobre el nombre de la tabla dos veces consecutivas (ilustración de la derecha)

id	alumno
1	alumno 1
2	alumno 2
3	alumno 3
4	alumno 4
5	alumno 5

Ilustración 11. Tabla alumnos

2.5.5 Utilización de instrucciones de consulta en procedimientos almacenados

A diferencia de otros gestores de bases de datos, en MySQL se puede (sólo en procedimientos) devolver como resultado de la ejecución del procedimiento un conjunto de filas. La funcionalidad de esta forma de recuperación se asemeja a la que se puede conseguir con el empleo de vistas.

En el siguiente procedimiento se recupera el conjunto de filas de los departamentos que pertenecen al centro que se le pasa como argumento:

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS resultset1 $$
3 CREATE PROCEDURE resultset1 (IN p_numce INTEGER)
4 BEGIN
5     SELECT numde, nomde, presu
6     FROM departamentos
7     WHERE numce = p_numce;
8 END $$
9 DELIMITER ;

```

Procedimiento 27 resultset1

Ejemplo de ejecución:

```
mysql> CALL test.resultset1(20);
```

numde	nomde	presu
110	DIRECCION COMERCIAL	15000000
111	SECTOR INDUSTRIAL	11000000
112	SECTOR SERVICIOS	9000000

3 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

El inconveniente es que el programa que recibe el conjunto de filas resultantes no puede ser un procedimiento MySQL sino otro programa escrito en otro lenguaje como Java o PHP (ver anexo de este tema). Para salvar este problema y poder enviar a un procedimiento MySQL un conjunto de registros se utilizan las tablas temporales:

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS resultset2 $$
3 CREATE PROCEDURE resultset2 (IN p_numce INTEGER)
4 BEGIN
5     DROP TEMPORARY TABLE IF EXISTS tmp_departamentos;
6     CREATE TEMPORARY TABLE tmp_departamentos AS
7     SELECT numde, nomde, presu
8     FROM departamentos
9     WHERE numce = p_numce;
10 END $$
11 DELIMITER ;

```

Procedimiento 28 resultset2

El tiempo de vida de las tablas temporales es el tiempo de duración de la sesión.

2.5.6 Almacenar en variables el valor de una fila de una tabla

No sólo las instrucciones SQL de definición y manipulación de datos (DDL y DML) pueden intercalarse en los procedimientos almacenados, la recuperación de los datos almacenados en la base de datos permitirá su posterior proceso o tratamiento.

El siguiente ejemplo almacena una fila entera de la tabla creada en el procedimiento anterior (tabla *alumnos*) en las variables declaradas en las líneas 6 y 7, mostrando su contenido en la línea 12. En la línea 8 se señalan las columnas a recuperar (en este caso

las dos) y en la línea 9 se indica donde (variables) se guardarán los valores de dichas columnas según el orden en que aparecen (el valor de la columna id se guardará en la variable v_id, el de la columna alumno en la variable v_alumno).

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS procedimiento2 $$
4 CREATE PROCEDURE procedimiento2 ()
5 BEGIN
6     DECLARE v_id INT;
7     DECLARE v_alumno VARCHAR(30);
8     SELECT id, alumno
9         INTO v_id, v_alumno
10        FROM alumnos
11       WHERE id = 4;
12     SELECT v_id, v_alumno;
13 END $$
14
15 DELIMITER ;

```

Procedimiento 29 - “procedimiento2”

En el anterior ejemplo no ha habido ningún problema de ejecución pues cada una de las dos variables almacena sólo un ítem (el de la correspondiente columna), pues la consulta sólo devuelve una fila.

v_id	v_alumno
4	alumno 4

Ilustración 12. Resultado de la ejecución del procedimiento anterior

Pero ¿Qué ocurriría si la consulta devolviera más de una fila?

Puedes probarlo a partir del anterior procedimiento, creando otro denominado procedimiento3: Opción de menú *File / Save As...*, cambia “procedimiento2” por “procedimiento3” en las filas 3 y 4, elimina la línea 11 y coloca un punto y coma al final de la línea 10 (por lo que la consulta pasará a devolver las 5 filas de la tabla al no existir cláusula WHERE); a continuación ejecuta el *procedimiento3*. Te encontrarás en la parte inferior de la pantalla de resultado con este error:

!	Description	ErrorNr.
!	Result consisted of more than one row	1172

Ilustración 13. Error: La consulta sólo debería devolver una fila

recordándote que no se puede obtener como resultado de la ejecución de una instrucción SELECT ... INTO más de una fila. Dicho error provoca que la ejecución del procedimiento finalice y no continúe por lo que las siguientes instrucciones a la que provocó el error (SELECT... INTO) no se ejecutarían (en este caso solo una, la instrucción que visualiza las dos variables).

Otra cuestión importante ¿Qué ocurriría si no devolviera ningún valor la instrucción SELECT como es el caso del *procedimiento4* que viene a continuación

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS procedimiento4 $$
4 CREATE PROCEDURE procedimiento4 ()
5 BEGIN
6     DECLARE v_id INT;
7     DECLARE v_alumno VARCHAR(30);
8     SELECT id, alumno
9         INTO v_id, v_alumno
10        FROM alumnos
11       WHERE id = 100;
12
13     SELECT v_id, v_alumno;
14 END $$
15
16 DELIMITER ;

```

Procedimiento 30 - "procedimiento4"

A diferencia de otros lenguajes de programación de bases de datos no provocaría situación de error severa similar a la anterior del *procedimiento3*. El *procedimiento4* finalizaría mostrando los contenidos de las variables *v_id* y *v_alumno*, nulo, puesto que no existe el alumno de id 100 y por tanto la sentencia SQL no recupera ninguna fila.

v_id	v_alumno
NULL	NULL

Ilustración 14. La consulta no devuelve filas de la tabla

2.5.7 Sentencias preparadas. SQL dinámico

De la misma manera que otros grandes gestores de bases de datos, MySQL soporta la capacidad de preparar sentencias SQL para que estas pueden ser ejecutadas varias veces de manera eficiente y segura debido a que disminuye considerablemente el tiempo de análisis y preparación de la instrucción que va a ser ejecutada así como puede prevenir el problema de seguridad del SQL inyectado mediante la utilización de los parámetros o variables BIND (ver ejemplos en PHP en anexo). La sintaxis para crear una sentencia preparada es:

```
PREPARE nombre_sentencia FROM texto_sql
```

donde el *texto_sql* contiene marcadores (carácter ?) para representar los valores que se utilizarán en el momento de ejecutar la instrucción SQL.

La sentencia preparada anterior se ejecuta mediante:

```
EXECUTE nombre_sentencia [USING variable [,variable...]]
```

donde la cláusula USING se encarga de suministrar los valores para los marcadores especificados en la sentencia PREPARE. La forma de especificar estos valores es en forma de variables de usuario (prefijo @).

La sentencia preparada se puede eliminar mediante:

```
DEALLOCATE PREPARE sentencia_preparada
```

Ejemplo de sentencias preparadas

```
mysql> USE TEST
Database changed
mysql> PREPARE alumnos_insert_dinamic FROM "INSERT INTO alumnos
VALUES(?,?)";
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> SET @id='1000';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @nombre='Alumno 1000';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE alumnos_insert_dinamic USING @id,@nombre;
Query OK, 1 row affected (0.02 sec)

mysql> SET @id='1001';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @nombre='Alumno 1001';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE alumnos_insert_dinamic USING @id,@nombre;
Query OK, 1 row affected (0.02 sec)

mysql> DEALLOCATE PREPARE alumnos_insert_dinamic;
Query OK, 0 rows affected (0.00 sec)
```

Los procedimientos almacenados no necesitan el mecanismo de sentencias preparadas puesto que las instrucciones que se contienen ya se encuentran listas y preparadas para su ejecución. De todas formas su utilización dentro de procedimientos puede tener sentido si se desea ejecutar SQL dinámico dentro de ellos (no puede hacerse ni dentro de funciones ni de triggers). Una instrucción SQL es dinámica si es construida en tiempo de ejecución a diferencia de las instrucciones “estáticas” que se construyen cuando se compila el procedimiento. El empleo por tanto de SQL dinámico tiene sentido cuando no se conoce por completo la sentencia SQL en el momento de la compilación y necesita ser completada mediante algún dato procedente de una entrada del usuario o de otra aplicación.

Al siguiente procedimiento, que utiliza SQL dinámico, se le puede pasar cualquier instrucción SQL como argumento:

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS sql_dinamico1 $$
4 CREATE PROCEDURE sql_dinamico1(instruccion_sql VARCHAR(4000))
5 BEGIN
6     SET @tmp_sql=instruccion_sql;
7     PREPARE instruccion FROM @tmp_sql;
8     EXECUTE instruccion;
9     DEALLOCATE PREPARE instruccion;
10 END $$
11 DELIMITER ;
```

Procedimiento 31 sql_dinamico1

La ejecución del procedimiento anterior eliminará los dos alumnos introducidos:

```
CALL sql_dinamico1('delete from alumnos where id >=1000')
```

Ilustración 15. Ejecutando el procedimiento sql_dinamico1

Nota: Cuidado con el ejemplo del procedimiento anterior, su finalidad es totalmente ilustrativa pero no debe utilizarse y mucho menos ser llamado desde otras aplicaciones PHP, ASP, Visual Basic... pues garantiza al 100% que ocurra el grave problema del SQL inyectado, permitiendo al que invoca el procedimiento cualquier operación de cualquier tipo sobre la base de datos.

El SQL dinámico no se usa muy a menudo y debe utilizarse sólo en los casos en que sea necesario debido a que es más complejo y menos eficiente que el SQL estático. Debe emplearse para realizar tareas o implementar utilidades que no pueden realizarse de otra manera. Un ejemplo muy característico de su empleo por su grado de optimización y rapidez es para encontrar filas a partir de múltiples criterios de búsqueda:

```
1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS sql_dinamico2 $$
3 CREATE PROCEDURE sql_dinamico2(
4     p_numde INTEGER,
5     p_fecha_ingreso DATE,
6     p_salario INTEGER)
7 BEGIN
8     DECLARE v_clausula_where VARCHAR(1000) DEFAULT 'WHERE';
9     IF p_numde IS NOT NULL THEN
10         SET v_clausula_where=CONCAT(v_clausula_where,
11             ' numde=',p_numde,'');
12     END IF;
13     IF p_fecha_ingreso IS NOT NULL THEN
14         IF v_clausula_where<>'WHERE' THEN
15             SET v_clausula_where=CONCAT(v_clausula_where, ' AND ');
16         END IF;
17         SET v_clausula_where=CONCAT(v_clausula_where,
18             ' fecin < ',p_fecha_ingreso,'');
19     END IF;
20     IF p_salario IS NOT NULL THEN
21         IF v_clausula_where<>'WHERE' THEN
22             SET v_clausula_where=CONCAT(v_clausula_where, ' AND ');
23         END IF;
24         SET v_clausula_where=CONCAT(v_clausula_where,
25             ' salario >=',p_salario,'');
26     END IF;
27     SET @sentencia_sql=CONCAT('SELECT * FROM empleados ',
28         v_clausula_where);
29     PREPARE instruccion FROM @sentencia_sql;
30     EXECUTE instruccion;
31     DEALLOCATE PREPARE instruccion;
32 END $$
33 DELIMITER ;
```

Procedimiento 32 sql_dinamico2

Para averiguar los empleados del departamento 121 contratados antes de la década de los 60 cuyo salario sea superior a 300:

```
CALL sql_dinamico2(121, '1960-01-01', 300)
```

Ilustración 16. Ejecución del procedimiento sql_dinamico2

o la misma consulta anterior pero sin importar la fecha de ingreso:

```
CALL sql_dinamico2(121, null, 300)
```

Ilustración 17. Ejecución del procedimiento sql_dinamico2

Observa cómo el procedimiento anterior espera tres parámetros, dos de tipo numérico y otro de tipo fecha aunque alguno de ellos pueda ser nulo. La posibilidad de poder inyectar una instrucción SQL utilizando los procedimientos de esta manera es prácticamente nula.

2.6 Cursores

Es el instrumento que se utiliza cuando la sentencia SQL dentro del programa devuelve más de una fila como hemos visto en el apartado anterior.

De esta manera, un cursor es una zona de memoria que contiene un conjunto de filas resultantes de una sentencia SQL con la ventaja de que podremos recorrer, visualizar y manipular una a una cada una de esas filas.

Sintaxis:

```
DECLARE nombre_del_cursor CURSOR FOR sentencia_select;
```

Un aspecto muy importante a tener en cuenta es que los cursores se declaran en los procedimientos u otros programas almacenados después de la declaración de variables, no hacerlo así producirá una situación de error.

2.6.1 Sentencias utilizadas con cursores. Ejemplos

Veamos antes el siguiente ejemplo:

```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS cursor1 $$
4 CREATE PROCEDURE cursor1 ()
5 BEGIN
6     DECLARE v_id INT;
7     DECLARE v_alumno VARCHAR(30);
8     DECLARE c_alumnos CURSOR FOR
9     SELECT id, alumno -- SELECT *
10    FROM alumnos
11   WHERE id <= 2; -- Sólo las 2 primeras filas
12   OPEN c_alumnos;
13   alumnos_cursor: LOOP
14       FETCH c_alumnos INTO v_id, v_alumno;
15       /* Aquí iría el tratamiento de los datos recuperados */
16       /* Por el momento solo visualizarlos */
17       SELECT v_id, v_alumno;
18   END LOOP alumnos_cursor;
19   CLOSE c_alumnos;
20
21 END $$
22
23 DELIMITER ;
```

Procedimiento 33 cursor1

Las sentencias asociadas al cursor se describen a continuación. Más adelante encontrarás una explicación gráfica que te ayudará a comprender mejor el funcionamiento de los cursores así como del procedimiento anterior:

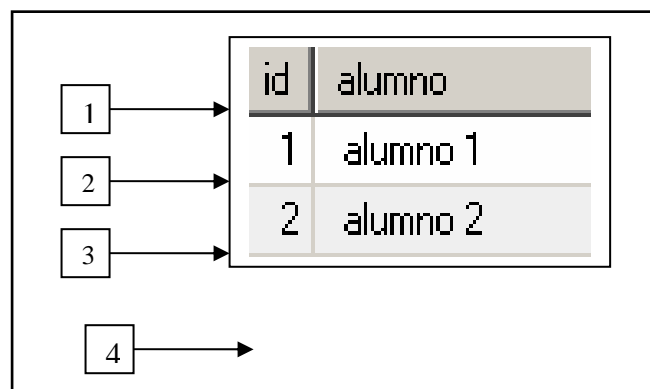
- **Declaración del cursor:** En las líneas 8 a 11 se declara el cursor.
- **Apertura del cursor:** Cláusula OPEN (línea 12). El cursor se inicializa trayendo a memoria las filas (conjunto activo) que cumplen la condición del cursor, en este caso sólo 2 filas por la cláusula WHERE de la línea 11. El puntero del cursor se queda apuntando al comienzo del conjunto de filas recuperadas.
- **Recuperación de las filas del cursor.** Cláusula FETCH (línea 14): Cada vez que se ejecuta recupera la siguiente fila a la que apunta el cursor y avanza el puntero una posición. Cada fila recuperada la va almacenando en las variables *v_id* y *v_alumno*.
- **Cierre del cursor:** Cláusula CLOSE (línea 19). Desactiva el cursor liberando la zona de memoria ocupada por el cursor.

Puedes ver una ejecución más clara del procedimiento anterior abriendo una ventana cliente sin salir de la herramienta *MySQL Query Browser* mediante la opción de menú *Tools / MySQL Command Line Client*:

```
mysql> call cursor1();
+-----+-----+
| v_id | v_alumno |
+-----+-----+
| 1    | alumno 1 |
+-----+-----+
1 row in set (0.00 sec)

+-----+-----+
| v_id | v_alumno |
+-----+-----+
| 2    | alumno 2 |
+-----+-----+
1 row in set (0.01 sec)
ERROR 1329 (02000): No data - zero rows fetched, selected, or
processed
```

¿Qué ha ocurrido realmente?



Si observas en el esquema anterior, el recorrido de un cursor se asemeja mucho al recorrido de un fichero secuencial, produciéndose error cuando se intenta leer un registro y previamente se ha alcanzado el final del fichero (*“ERROR 1329 (02000): No data - zero rows fetched, selected, or processed”*)

La ejecución, reflejada paso a paso en el esquema, es la siguiente:

Recuadro 1: Apertura del cursor (línea 12 del procedimiento anterior). Se traen a memoria las dos filas que cumplen la declaración del cursor. El puntero (flecha) del cursor apunta al comienzo de la primera fila.

Recuadro 2: Primera vez que se ejecuta en el bucle la sentencia FETCH (línea 14). Se recupera la siguiente fila apuntada por el cursor, en este caso la primera, almacenando el valor de la primera columna (1) en la variable *v_id* y el valor de la segunda columna (alumno1) en la variable *v_alumno*. El puntero del cursor avanza una posición. Antes de finalizar el bucle se muestran los valores de las variables (línea 17)

<i>v_id</i>	<i>v_alumno</i>
1	alumno 1

Ilustración 18. Visualización de la primera fila del cursor

Recuadro 3: Segunda vez que se ejecuta en el bucle la sentencia FETCH (línea 14). Se recupera la siguiente fila apuntada por el cursor, en este caso la segunda, almacenando el valor de la primera columna (2) en la variable *v_id* y el valor de la segunda columna (alumno2) en la variable *v_alumno*. El puntero del cursor avanza una posición. Antes de finalizar el bucle se muestran los valores de las variables (línea 17):

<i>v_id</i>	<i>v_alumno</i>
2	alumno 2

Ilustración 19. Visualización de la segunda fila del cursor

Recuadro 4: Tercera vez que se ejecuta en el bucle la sentencia FETCH (línea 14). Se intenta recuperar la siguiente fila apuntada por el cursor pero no hay ninguna fila que recuperar pues el cursor ha finalizado, por tanto se produce la situación de error:

ERROR 1329 (02000): No data - zero rows fetched, selected, or processed

Ilustración 20. Error en el cursor

Para tratar este error, el intento de recuperar una fila habiendo llegado ya al final del cursor, definiremos un manejador de error; los manejadores de error y el tratamiento de errores se estudiarán en un apartado próximo pero ya se avanza una pequeña parte del mismo para poder trabajar este punto:

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



```
1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS cursor2 $$
4 CREATE PROCEDURE cursor2 ()
5 BEGIN
6     DECLARE v_id INT;
7     DECLARE v_alumno VARCHAR(30);
8     DECLARE v_ultima_fila INT DEFAULT 0;
9     DECLARE c_alumnos CURSOR FOR
10     SELECT id, alumno
11     FROM alumnos
12     WHERE id <= 2;
13     DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
14     OPEN c_alumnos;
15     alumnos_cursor: LOOP
16         FETCH c_alumnos INTO v_id, v_alumno;
17         IF v_ultima_fila=1 THEN
18             LEAVE alumnos_cursor;
19         END IF;
20         SELECT v_id, v_alumno;
21     END LOOP alumnos_cursor;
22     CLOSE c_alumnos;
23
24 END $$
25
26 DELIMITER ;
```

Procedimiento 34 cursor2

Explicación del procedimiento *cursor2*:

1. Con la instrucción de la línea 13 estamos declarando un manejador de error para el error NOT FOUND, que es el que se produce cuando se intenta recuperar una fila habiendo llegado al final del cursor. Se utiliza una variable (*v_ultima_fila*) que tomará el valor 1 en el momento que se produzca el error. Dentro de nuestro programa podremos preguntar por su valor.
2. Cuando se produzca la situación de error, la ejecución del programa continuará (debido a la cláusula CONTINUE de la línea 13) y no finalizará en el punto donde se encuentra el error. En el momento en que la instrucción de la línea 16 (FETCH ... INTO) falle por no poder leer ninguna fila del cursor, la variable *v_ultima_fila* tomará el valor 1 y la ejecución del procedimiento continuará por la línea siguiente al error (en este caso por la línea 17) para abandonar el bucle.

Puedes ejecutar el programa desde la línea de comandos y comprobarás que ya no se finaliza el programa tras el error como en el procedimiento *cursor1*, puesto que en el momento en que se intente leer una fila del cursor posterior a la última, la condición de la línea 17 será cierta y por tanto la ejecución del programa abandonará el bucle (línea 18) y continuará por la línea 22 (recuerda que si el error no se hubiera tratado el programa hubiera finalizado y terminaría en el mismo punto en que se produjera el error).

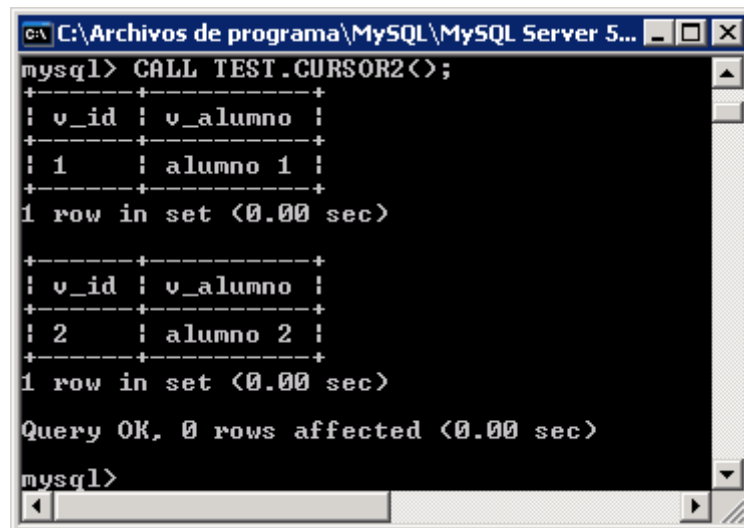


Ilustración 21. Ejecución correcta del cursor

El mismo recorrido del cursor pero con la instrucción iterativa REPEAT... UNTIL:

```

13  DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
14  OPEN c_alumnos;
15      alumnos_cursor: REPEAT
16          FETCH c_alumnos INTO v_id, v_alumno;
17          IF v_ultima_fila=1 THEN
18              LEAVE alumnos_cursor;
19          END IF;
20          SELECT v_id, v_alumno;
21          UNTIL v_ultima_fila -- v_ultima_fila=1
22          END REPEAT alumnos_cursor;
23  CLOSE c_alumnos;
24
25  END $$

```

Procedimiento 35 cursor2 utilizando la instrucción REPEAT UNTIL

y con la instrucción WHILE:

```

13  DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
14  OPEN c_alumnos;
15      alumnos_cursor: WHILE (v_ultima_fila=0) DO
16          FETCH c_alumnos INTO v_id, v_alumno;
17          IF v_ultima_fila=1 THEN
18              LEAVE alumnos_cursor;
19          END IF;
20          SELECT v_id, v_alumno;
21          END WHILE alumnos_cursor;
22  CLOSE c_alumnos;
23
24  END $$

```

Procedimiento 36 cursor2 utilizando la instrucción WHILE

Es muy habitual que las filas a recuperar del cursor dependan de un dato que se pasa como argumento. El procedimiento siguiente es el mismo que el procedimiento *cursor2* salvo que el número de filas a recuperar viene determinado por el parámetro *p_id* que se le pasa al procedimiento (línea 4) y por tanto también se modifica la línea 12:

```

1 DELIMITER $$
2
3 DROP PROCEDURE IF EXISTS cursor2_1 $$
4 CREATE PROCEDURE cursor2_1 (IN p_id INT)
5 BEGIN
6     DECLARE v_id INT;
7     DECLARE v_alumno VARCHAR(30);
8     DECLARE v_ultima_fila INT DEFAULT 0;
9     DECLARE c_alumnos CURSOR FOR
10     SELECT id, alumno
11     FROM alumnos
12     WHERE id <= p_id;
13     DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
14     OPEN c_alumnos;
15     alumnos_cursor: LOOP
16         FETCH c_alumnos INTO v_id, v_alumno;
17         IF v_ultima_fila=1 THEN
18             LEAVE alumnos_cursor;
19         END IF;
20         SELECT v_id, v_alumno;
21     END LOOP alumnos_cursor;
22     CLOSE c_alumnos;
23
24 END $$
25
26 DELIMITER ;

```

Procedimiento 37 cursor2_1

La llamada al procedimiento:

```
mysql> CALL cursor2_1(4);
```

producirá como resultado la visualización de las 4 primeras filas de la tabla alumnos.

2.6.2 Cursores anidados

En algunas ocasiones, del mismo modo que ocurre con bloques de instrucciones, será necesario incluir un cursor dentro de otro cursor. P.ej un cursor que recorriera las filas de todos los Centros y a su vez, por cada centro, otro cursor que recorriera todos los departamentos del mismo:

```

Centro: 10 - SEDE CENTRAL
Departamento: DIRECCION GENERAL
Departamento: ORGANIZACION
Departamento: PERSONAL
Departamento: PROCESO DE DATOS
Departamento: FINANZAS
+-----+
TOTAL DEPARTAMENTOS DEL CENTRO
+-----+
| 5                                     |
+-----+
Centro: 20 - RELACION CON CLIENTES
Departamento: DIRECCION COMERCIAL
Departamento: SECTOR INDUSTRIAL
Departamento: SECTOR SERVICIOS
+-----+
| TOTAL DEPARTAMENTOS DEL CENTRO      |
+-----+
| 3                                     |
+-----+

```

El procedimiento quedaría:

```

1 DELIMITER $$
2 DROP PROCEDURE IF EXISTS cursor5 $$
3 CREATE PROCEDURE cursor5 ()
4 BEGIN
5     DECLARE v_numce INTEGER;
6     DECLARE v_nomce VARCHAR(25);
7     DECLARE v_nomde VARCHAR(20);
8     DECLARE v_ultima_fila INT DEFAULT 0;
9     DECLARE v_total_departamentos INTEGER;
10    DECLARE c_centros CURSOR FOR
11    SELECT numce, nomce
12    FROM centros
13    ORDER BY numce;
14    DECLARE c_departamentos CURSOR FOR
15    SELECT nomde
16    FROM departamentos
17    WHERE departamentos.numce = v_numce;
18    DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_ultima_fila=1;
19    OPEN c_centros;
20    centros_cursor: LOOP
21        FETCH c_centros INTO v_numce, v_nomce;
22        IF v_ultima_fila=1 THEN
23            LEAVE centros_cursor;
24        END IF;
25        SELECT CONCAT('Centro: ', v_numce, ' - ', v_nomce) AS 'CENTRO';
26        SET v_total_departamentos = 0;
27        OPEN c_departamentos;
28        departamentos_cursor: LOOP
29            FETCH c_departamentos INTO v_nomde;
30            IF v_ultima_fila=1 THEN
31                LEAVE departamentos_cursor;
32            END IF;
33            SELECT CONCAT('Departamento: ', v_nomde) AS 'Departamento';
34            SET v_total_departamentos = v_total_departamentos + 1;
35        END LOOP departamentos_cursor;
36        CLOSE c_departamentos;
37        SELECT v_total_departamentos AS 'TOTAL DEPARTAMENTOS DEL CENTRO';
38        SET v_ultima_fila=0;
39    END LOOP centros_cursor;
40    CLOSE c_centros;
41
42 END $$
43 DELIMITER ;

```

Procedimiento 38 cursor5

Es muy importante señalar en el procedimiento anterior y cuando se utilicen cursores anidados que antes de leer una nueva fila del cursor externo, hay que resetear la variable utilizada para controlar el final del cursor (línea 38) para que el final del cursor interno no provoque un final del cursor externo sin haber terminado este último.

Antes de terminar este apartado indicar que existe un tipo especial de cursores de actualización que se tratan en el tema siguiente de transacciones.

2.7 Manejo de errores

2.7.1 Introducción a la gestión de errores

En general, si una sentencia SQL falla dentro de un programa almacenado se produce una situación de error, se interrumpe la ejecución del programa en ese punto y finaliza salvo en el caso de que el programa que falla hubiera sido llamado por otro; en ese caso la ejecución continua por el programa que llamó a este programa que ha causado el

error; ocurriría lo mismo si en lugar de un programa que es llamado desde otro programa nos encontráramos con un bloque interno dentro de otro bloque más externo; si se produjera error en el bloque anidado interno, la ejecución del programa se interrumpiría en el bloque interno para continuar por el externo.

Como hemos visto anteriormente, este comportamiento se puede controlar definiendo los manejadores de error o handlers.

Una handler es un bloque de instrucciones SQL que se ejecuta cuando se verifica una condición tras una excepción (error) generada por el servidor.

Sintaxis:

```
DECLARE {CONTINUE | EXIT | UNDO} HANDLER FOR
        {SQLSTATE sqlstate_code | MySQL error code | nombre_condición}
        instrucciones_del_manejador
```

Deben declararse después de las declaraciones de variables y cursores ya que referencian a estos en su declaración.

El manejador puede ser de tres tipos:

- CONTINUE: La excepción o error generado no interrumpe el código del procedimiento.
- EXIT: Permite poner fin al bloque de instrucciones o programa en el que se genera la excepción.
- UNDO: No está soportada por el momento.

La condición de error del manejador puede expresarse también de 3 formas:

- Con un código estándar ANSI SQLSTATE.
- Con un código de error MySQL.
- Una expresión.

El manejador de error indica mediante un conjunto de instrucciones lo que hay que hacer en caso de que se produzca ese error.

Ejemplos de situaciones de error:

```
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error1 $$
4 CREATE PROCEDURE error1 (p_id INT, p_alumno VARCHAR(30))
5 MODIFIES SQL DATA
6 BEGIN
7     INSERT INTO alumnos VALUES(p_id,p_alumno);
8 END $$
```

Procedimiento 39 error1

El anterior procedimiento recibe dos argumentos, identificador de alumno y nombre del alumno y los inserta en la base de datos. Como modifica la base de datos se intercala la cláusula de la línea 5.

Si hacemos una llamada al anterior procedimiento desde la ventana cliente o desde la propia herramienta *MySQL Query Browser*:

```
CALL error1(6, 'Alberto Carrera');
```

Instrucción 2. Llamando al procedimiento error1

se puede comprobar viendo la tabla o realizando una consulta que añada al anterior alumno.

¿Qué ocurre si intentamos insertar un alumno de clave primaria repetida?

```
CALL error1(6, 'Raquel M. Carrera');
```

Aparecerá un error con un número de error (1062) y una descripción (“entrada duplicada”) advirtiéndonos de tal situación; evidentemente no se realiza la inserción y la ejecución del programa se detiene y finaliza (pues no retorna a ningún otro programa ya que no hay ningún otro que lo llamó).

Tomando nota del número de error y modificando el anterior procedimiento para añadir dicho control de error:

```
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error2 $$
4 CREATE PROCEDURE error2 (p_id INT, p_alumno VARCHAR(30))
5 MODIFIES SQL DATA
6 BEGIN
7     DECLARE CONTINUE HANDLER FOR 1062
8         SELECT CONCAT('Nº de matrícula ', p_id, ' ya existente') AS 'Aviso de error';
9     INSERT INTO alumnos VALUES(p_id,p_alumno);
10 END $$
```

Procedimiento 40 error2

Una llamada al programa como:

```
CALL error2(6, 'Raquel M. Carrera');
```

Aviso de error

Producirá la siguiente salida: Nº de matrícula 6 ya existente

Resaltar que el error ha sido tratado y por tanto no finaliza la ejecución del programa en el momento en que se produce y si hubiera más líneas de código detrás de la línea 9, éstas se ejecutarían debido al tipo de manejador, CONTINUE; esto no ocurriría si el error no fuera tratado como hemos comprobado en el procedimiento *error1* anterior

¿Qué ocurre si intentamos introducir un alumno de clave primaria nula?

```
CALL error2(NULL, 'Mario A. Carrera');
```

Habrá que considerar y tratar también el error 1048 que se produce:

```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error3 $$
4 CREATE PROCEDURE error3 (p_id INT, p_alumno VARCHAR(30))
5 MODIFIES SQL DATA
6 BEGIN
7     DECLARE CONTINUE HANDLER FOR 1062
8         SELECT CONCAT('Nº de matrícula ', p_id, ' ya existente') AS 'Aviso de error';
9     DECLARE CONTINUE HANDLER FOR 1048
10        SELECT CONCAT('El nº de matrícula no puede ser nulo') AS 'Aviso de error';
11    INSERT INTO alumnos VALUES(p_id,p_alumno);
12 END $$

```

Procedimiento 41 error3

Llegados a este punto podemos utilizar un tercer argumento en el procedimiento, un parámetro de tipo OUT, para que actúe como flag o indicador de cómo ha ido la ejecución del procedimiento:

```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error4 $$
4 CREATE PROCEDURE error4 (p_id INT, p_alumno VARCHAR(30), OUT p_resultado VARCHAR(40))
5 MODIFIES SQL DATA
6 BEGIN
7     DECLARE CONTINUE HANDLER FOR 1062
8         SET p_resultado = CONCAT('Nº de matrícula ', p_id, ' ya existente');
9     DECLARE CONTINUE HANDLER FOR 1048
10        SET p_resultado = 'Nº de matrícula nulo';
11    SET p_resultado = 'correcto';
12    INSERT INTO alumnos VALUES(p_id,p_alumno);
13 END $$

```

Procedimiento 42 error4

Llamando al anterior procedimiento *error4* desde el interior de otro (línea 7 de *error5*):

```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error5 $$
4 CREATE PROCEDURE error5 ()
5 BEGIN
6     DECLARE v_resultado VARCHAR(40);
7     CALL error4 (7, 'Mariano Carrera', v_resultado);
8     IF v_resultado = 'correcto' THEN
9         SELECT 'Alumno dado de alta' AS 'Insercion de alumnos';
10    else
11        SELECT v_resultado AS 'Aviso de error';
12    END IF;
13 END $$

```

Procedimiento 43 error5

La primera ejecución del procedimiento: *CALL error5()* funcionará correctamente, apareciendo como fila/columna resultante : 'Alumno dado de alta'.

La segunda ejecución del mismo procedimiento anterior también funcionará correctamente pues la situación de error de código repetido es tratada y por tanto el programa finaliza correctamente y visualiza por pantalla que el número de matrícula ya existe.

Volviendo otra vez a la declaración de un manejador, hemos visto que tiene 3 partes que pasaremos a ver a continuación más en detalle:

1. Tipo de manejador: CONTINUE o EXIT.
2. Condición del manejador: SQLSTATE *sqlstate_code* | MySQL error code | *nombre_condición*.
3. Acciones o instrucciones que realiza el manejador.

2.7.2 Tipos de manejador

- **EXIT:** El programa que causa el error finaliza, devolviendo el control al programa que le llamó. Si se produce en un bloque interno, entonces el control de la ejecución pasa al externo.
- **CONTINUE:** La ejecución continúa por la siguiente línea de código que causó el error.

En ambos casos el error es tratado por lo que la ejecución del programa puede considerarse correcta y antes de realizarse la opción CONTINUE o EXIT se ejecuta el conjunto de instrucciones asociados al manejador.

Ejemplo de utilización del manejador EXIT:

```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error6 $$
4 CREATE PROCEDURE error6 (p_id INT, p_alumno VARCHAR(30))
5 MODIFIES SQL DATA
6 BEGIN
7     DECLARE v_clave_repetida TINYINT DEFAULT 0;
8     BEGIN
9         DECLARE EXIT HANDLER FOR 1062
10             SET v_clave_repetida=1;
11         INSERT INTO alumnos VALUES(p_id,p_alumno);
12         SELECT CONCAT('Alumno dado de alta: ', p_alumno, ' con matrícula nº: ', p_id )
13             AS 'Inserción de alumnos';
14     END;
15     IF v_clave_repetida=1 THEN
16         SELECT CONCAT('Nº de matrícula: ', p_id, ' ya existente') as 'Aviso de error';
17     END IF;
18 END $$

```

Procedimiento 44. error6

Observa como en este caso hay dos bloques de instrucciones BEGIN...END. El manejador de errores con la opción EXIT está integrado con el bloque más interno.

La llamada al procedimiento anterior:

```
CALL error6(8, 'Fernando Carrera');
```

Dará como resultado la inserción del alumno y su mensaje correspondiente de alumno dado de alta.

```
CALL error6(8, 'Conchita Martín');
```

dará como resultado un aviso indicando que el número de matrícula 8 ya existe. La instrucción de la línea 11 falla (no se lleva a cabo por tanto la inserción) puesto que no puede haber dos filas distintas con el mismo valor en el campo id (al ser clave primaria) provocando el error de clave repetida que hace que se ejecute la instrucción asociada al manejador (línea 10). Como el tipo de manejador es de tipo EXIT, se interrumpe la ejecución del programa y se sale del bloque interno (si sólo hubiera habido un bloque se hubiera salido del programa); por tanto la instrucción de la línea 12 no llega a ejecutarse y la ejecución del programa sigue por la línea 15.

Ejemplo de utilización del manejador CONTINUE. Observa que a diferencia del anterior caso, en este procedimiento no hay más que un bloque:


```
2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS error7 $$
4 CREATE PROCEDURE error7 (p_id INT, p_alumno VARCHAR(30))
5 MODIFIES SQL DATA
6 BEGIN
7     DECLARE v_clave_repetida TINYINT DEFAULT 0;
8     DECLARE CONTINUE HANDLER FOR 1062
9         SET v_clave_repetida=1;
10    INSERT INTO alumnos VALUES(p_id,p_alumno);
11    IF v_clave_repetida=1 THEN
12        SELECT CONCAT('Nº de matrícula: ', p_id, ' ya existente') as 'Aviso de error';
13    ELSE
14        SELECT CONCAT('Alumno dado de alta: ', p_alumno, ' con matrícula nº: ', p_id )
15            AS 'Inserción de alumnos';
16    END IF;
17 END $$
```

Procedimiento 45. error7

`CALL error7(9, 'Pablo Martínez');`

dará como resultado la inserción del alumno Pablo y su mensaje correspondiente de alumno dado de alta. Como no falla la instrucción la expresión de la línea 11 es falsa y se ejecutará la línea 14.

`CALL error7(9, 'Luis Hueso');`

dará como resultado una aviso indicando que ese número de matrícula ya existe. Al intentar hacer la inserción de la línea 10 el procedimiento provoca una situación de error (igual que para el caso anterior de la cláusula EXIT) que es tratada inmediatamente después en el manejador de error poniendo a 1 el valor de la variable `v_clave_repetida`; por tanto, al ser tratado el error y de forma continua (cláusula CONTINUE línea 8) la ejecución del procedimiento continua por la línea 11 haciendo cierta la condición y avisando de la duplicidad de dicha matrícula.

¿Qué tipo de manejador usar? ¿EXIT o CONTINUE? Algunos lenguajes de programación de otros sistemas gestores de bases de datos incluyen por defecto sólo la opción EXIT (de todas maneras en estos casos se puede simular una opción CONTINUE colocando la instrucción propensa a causar error dentro de un bloque BEGIN...END).

Si la lógica del programa obliga a abandonar la ejecución del mismo si se produce un error entonces se puede utilizar EXIT. En caso de que esté contemplado en la lógica del programa que se pueden ejecutar otras alternativas si se produce error podemos utilizar la cláusula CONTINUE.

2.7.2 La condición del manejador

Has 3 formas posibles de indicar cuando debe ser invocado el conjunto de instrucciones del manejador de error para tratarlo.

- 1) Número de error de MySQL.
- 2) Código SQLSTATE estándar ANSI.
- 3) Condiciones de error con nombre.

1) **Número de error de MySQL**, propio de MySQL, que dispone de un conjunto particular de números de error. Ej.:

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

El número de error 1062 está asociado en MySQL al intento de almacenar un registro de clave duplicada.

2) **Código SQLSTATE estándar ANSI**: A diferencia del anterior, SQLSTATE no es definido por MySQL sino que es un estándar y por tanto el mismo error tiene asociado el mismo SQLSTATE independientemente del gestor de bases de datos utilizado: MySQL, Oracle, DB2, Microsoft SQL Server. En los anteriores gestores de bases de datos el SQLSTATE 23000 está asociado al error de clave duplicada. Por tanto en MySQL el anterior manejador de error y el siguiente son similares:

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

¿Cuál utilizar de los dos anteriores? En teoría el segundo permitiría una mayor portabilidad a otros gestores de bases de datos pero en la práctica los lenguajes de programación para Oracle y Microsoft SQL Server son incompatibles con el de MySQL por lo que no hace muy atractivo el uso de esta segunda opción (DB2 de IBM es “algo” compatible pues tanto DB2 como MySQL están basados en el estándar SQL:2003). Además hay códigos SQLSTATE que no corresponden con un código de error de MySQL sino a varios (para estos casos se utiliza el SQLSTATE genérico ‘HY000’)

Por lo anteriormente expuesto seguiremos utilizando en este curso los códigos propios de error de MySQL.

Para saber el código de un error, se puede averiguar de varias formas:

1. Como en otros lenguajes, provocarlo para obtener su número de error que luego utilizaremos en el manejador del error. Ej.: Hemos visto en páginas anteriores de este curso el error que se produce cuando se intenta leer una fila de un cursor y ya existen más:

```
ERROR 1329 (02000): No data - zero rows fetched, selected, or processed
```

El 1329 indica el número de error de MySQL y entre paréntesis, en este caso 02000, el número SQLSTATE correspondiente.

2. Mirando el manual, suele venir una tabla detallada de errores con su número y descripción en un apéndice dedicado.

3) **Condiciones de error con nombre**

3.1) **Predefinidas de MySQL**

```
SQLEXCEPTION
SQLWARNING
NOT FOUND
```

Ejemplos:

```
/* Si ocurre cualquier condición de error (salvo la excepción NOT FOUND tratada en el apartado de cursores) la variable v_error valdrá 1 y continuará la ejecución del programa */
```

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION  
SET v_error=1;
```

```
/* Si ocurre cualquier condición de error (excepto la condición NOT FOUND) el procedimiento finaliza ejecutando antes la instrucción ROLLBACK (deshacer operaciones que se hubieran realizado) y otra de advertencia de la situación de error */
```

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION  
BEGIN  
    ROLLBACK;  
    SELECT 'Ocurrió un error. Procedimiento terminado';  
END;
```

```
/* Si la instrucción FETCH en un cursor no recupera ninguna fila*/
```

```
/* De 3 formas distintas: Condición de error con nombre, SQLSTATE y código de error de MySQL */
```

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

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'  
SET v_ultima_fila=1;
```

```
DECLARE CONTINUE HANDLER FOR 1329  
SET v_ultima_fila=1;
```

- SQLWARNING es un subconjunto de los códigos SQLSTATE, representando todos los códigos que empiezan por 01.
- NOT FOUND es un subconjunto de los códigos SQLSTATE, representando todos los códigos que comienzan por 02.
- SQLEXCEPTION es un subconjunto de los códigos SQLSTATE, representando todos los códigos que no comienzan ni por 01 ni por 02.

3.2) *Definidos por el usuario*

Facilitan lectura del código y por tanto el fácil mantenimiento de la aplicación.

Consiste en “bautizar” o darle un nombre a un código de error MySQL o SQLSTATE.

Si intentas ejecutar el siguiente código que forma parte de un procedimiento:

```

5 BEGIN
6 CREATE TABLE alumnos (campol INTEGER );
7 END $$

```

Bloque 1

Verás que produce una situación de error pues la tabla ya existe:

```

ERROR 1050 (42S01): Table 'alumnos' already exists
mysql> _

```

En la imagen siguiente, al mismo tiempo que tratamos el error MySQL anterior le asociamos un nombre (línea 6) para facilitar su comprensión a la hora de realizar tareas como la de mantenimiento de la aplicación como se ha señalado. De esta manera ya se puede realizar la declaración del manejador de error de la línea 7.

```

5 BEGIN
6 DECLARE tabla_existente_error CONDITION FOR 1050;
7 DECLARE EXIT HANDLER FOR tabla_existente_error
8 SELECT ('Fin del programa. La tabla que intentas crear ya existe')
9 AS 'Aviso de error';
10 CREATE TABLE alumnos (campol INTEGER );
11 END $$

```

Bloque 2

2.7.4 Orden de actuación del manejador

Ante varias posibilidades de ejecutarse un manejador ¿Quién se ejecuta?

Siempre el manejador asociado al error MySQL en primer lugar (en este caso líneas 11-12 del bloque 3 siguiente). Si éste no estuviera declarado y definido entonces se ejecutaría el manejador asociado al código SQLSTATE. En último lugar el manejador asociado al manejador SQLEXCEPTION.

```

7 /*.....*/
8 BEGIN
9 DECLARE EXIT HANDLER FOR SQLEXCEPTION
10 SELECT 'Ocurrió un error. Fin del programa';
11 DECLARE EXIT HANDLER FOR 1062
12 SELECT 'Clave Repetida. Código MySQL 1062';
13 DECLARE EXIT HANDLER FOR SQLSTATE '23000'
14 SELECT 'Ocurrió error SQLSTATE 23000';
15 INSERT INTO alumnos VALUES(1,'MARIO A. CARRERA');
16 END $$

```

Bloque 3

El orden de actuación de los manejadores puede facilitarnos una forma de trabajo en nuestros programas. Los errores de código MySQL más habituales podrán ser tratados en manejadores específicos y aquéllos que no sean considerados podrán ser atrapados en un manejador SQLEXCEPTION (equivaldría a la parte ELSE de una sentencia CASE que atraparía todo lo que no ha sido “filtrado”).

En el siguiente ejemplo bloque 4 se declara un manejador para el tratar la excepción de clave duplicada (líneas 9 a 10), lo que ocurre es que el error producido es el código de

error MySQL 1048 (el primer campo de la tabla no puede ser nulo) y no el 1062 como consecuencia de la línea 13.

En esta situación se ejecuta el manejador SQLEXCEPTION (líneas 11-12)

```

7  /*.....*/
8  BEGIN
9      DECLARE EXIT HANDLER FOR 1062
10         SELECT 'Clave Repetida. Código MySQL 1062';
11      DECLARE EXIT HANDLER FOR SQLEXCEPTION
12         SELECT 'Ocurrió un error. Fin del programa';
13      INSERT INTO alumnos VALUES(NULL, 'MARIO A. CARRERA');
14  END $$

```

Bloque 4

Se podría hacer uso de una función del tipo “*err_code()*” que indique el código de error que se ha producido o una variable que almacenara el código de error y su mensaje. Esta función está incluida en los lenguajes de otros gestores de bases de datos y en otros (p.ej PHP) pero todavía no en MySQL (se espera para la versión 5.2, la especificación SQL:2003 si que lo incluye). También se echa en falta por el momento (aparecerá en la versión 5.2) la posibilidad de que el usuario provoque intencionadamente situaciones de error (sentencias del tipo SIGNAL, RAISE...) y poder atenderlas dentro de manejadores de error propios.

2.7.5 Ámbito de actuación del manejador

Como hemos visto los manejadores actúan en todos aquellos bloques donde se han declarado; su alcance llega también a los bloques anidados.

Ej.:

```

7  /*.....*/
8  BEGIN
9      DECLARE EXIT HANDLER FOR 1062
10         SELECT 'Clave Repetida. Código MySQL 1062';
11      DECLARE EXIT HANDLER FOR SQLEXCEPTION
12         SELECT 'Ocurrió un error. Fin del programa'
13         AS 'Aviso de error';
14      BEGIN
15         INSERT INTO alumnos VALUES(NULL, 'MARIO A. CARRERA');
16      END;
17  END $$

```

Bloque 5

Aunque la excepción se produce en el bloque interno (líneas 14 a 16), será atrapada por el manejador declarado en un nivel superior (bloque externo):

Aviso de error
Ocurrió un error. Fin del programa

Si se atrapa en un bloque interno, ya no se propaga a la del bloque superior externo:

```

7  /*.....*/
8  BEGIN
9      DECLARE EXIT HANDLER FOR 1062
10         SELECT 'Clave Repetida. Código MySQL 1062';
11      DECLARE EXIT HANDLER FOR SQLEXCEPTION
12         SELECT 'Ocurrió un error. Fin del programa'
13         AS 'Aviso de error';
14      BEGIN
15         DECLARE EXIT HANDLER FOR 1062
16            SELECT 'Bloque interno. Clave Repetida. Código MySQL 1062';
17         DECLARE EXIT HANDLER FOR SQLEXCEPTION
18            SELECT 'Bloque interno. Ocurrió un error. Fin del programa'
19            AS 'Aviso de error';
20         INSERT INTO alumnos VALUES(NULL, 'MARIO A. CARRERA');
21      END;
22 END $$

```

Bloque 6

Aviso de error
Bloque interno. Ocurrió un error. Fin del programa

El comportamiento de los manejadores de error entre bloques internos y externos es similar al comportamiento de un procedimiento que llama a otro (el primer procedimiento actuaría como “externo” y el llamado como “interno”)

2.7.6 Ejemplo de tratamiento de errores

El siguiente ejemplo resume todo lo tratado en este apartado 2.7. No se incluye el manejador de errores NOT FOUND pues en el procedimiento no se utilizan cursores.

```

2  DELIMITER $$
3  DROP PROCEDURE IF EXISTS error8 $$
4  CREATE PROCEDURE error8 (p_id INT, p_alumno VARCHAR(30),
5                          OUT p_error_num INT, OUT p_error_text VARCHAR(100))
6  MODIFIES SQL DATA
7  BEGIN
8      DECLARE clave_repetida_error CONDITION FOR 1062;
9      DECLARE clave_nula_error CONDITION FOR 1048;
10     DECLARE CONTINUE HANDLER FOR clave_repetida_error
11     BEGIN
12         SET p_error_num=1062;
13         SET p_error_text='Clave duplicada';
14     END;
15     DECLARE CONTINUE HANDLER FOR clave_nula_error
16     BEGIN
17         SET p_error_num=1048;
18         SET p_error_text='Clave nula';
19     END;
20     DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
21     BEGIN
22         SET p_error_num=-1;
23         SET p_error_text='Ocurrió un error';
24     END;
25     SET p_error_num=0;
26     INSERT INTO alumnos VALUES(p_id,p_alumno);
27     IF p_error_num=0 THEN
28         SET p_error_text='Alta de alumno realizada';
29     END IF;
30 END $$

```

Procedimiento 46 error8

```

mysql> CALL test.error8(1, 'ALBERTO CARRERA', @ENUM_ERROR, @TEXT_ERROR);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @ENUM_ERROR, @TEXT_ERROR;
+-----+-----+
| @ENUM_ERROR | @TEXT_ERROR |
+-----+-----+
| 1062       | Clave duplicada |
+-----+-----+
1 row in set (0.00 sec)

mysql> CALL test.error8(NULL, 'ALBERTO CARRERA', @ENUM_ERROR, @TEXT_ERROR);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @ENUM_ERROR, @TEXT_ERROR;
+-----+-----+
| @ENUM_ERROR | @TEXT_ERROR |
+-----+-----+
| 1048       | Clave nula |
+-----+-----+
1 row in set (0.00 sec)

mysql> CALL test.error8(15, 'ALBERTO CARRERA', @ENUM_ERROR, @TEXT_ERROR);
Query OK, 1 row affected (0.02 sec)

mysql> SELECT @ENUM_ERROR, @TEXT_ERROR;
+-----+-----+
| @ENUM_ERROR | @TEXT_ERROR |
+-----+-----+
| 0          | Alta de alumno realizada |
+-----+-----+
1 row in set (0.00 sec)

mysql> /* Borrado de tabla alumno */
mysql> CALL test.error8(16, 'CARMEN BAILIN', @ENUM_ERROR, @TEXT_ERROR);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @ENUM_ERROR, @TEXT_ERROR;
+-----+-----+
| @ENUM_ERROR | @TEXT_ERROR |
+-----+-----+
| -1         | Ocurri-| un error |
+-----+-----+
1 row in set (0.00 sec)

mysql> _

```

Ilustración 22. Probando el procedimiento 44 anterior

2.8 Funciones

2.8.1 Creación de funciones. Diccionario de Datos

La sintaxis completa de creación de funciones es:

```

CREATE FUNCTION nombre_función ([parametro1[,...]])
  RETURNS tipo_de_datos
  [LANGUAGE SQL]
  [ [NOT] DETERMINISTIC]
  [ {CONTAINS SQL | MODIFIES SQL DATA | READS SQL DATA | NO SQL} ]
  [SQL SECURITY {DEFINER | INVOKER} ]
  [COMMENT comentario]
  bloque_de_instrucciones_de_la_función

```

Como se ha visto en el tema de procedimientos, las funciones almacenadas son similares a los procedimientos salvo que sólo devuelven un valor. Por tanto gran parte de lo que se ha expuesto en el apartado de procedimientos es válido aquí. Los permisos que hay que poseer para crear funciones así como la información que proporciona el diccionario de datos se han expuesto en el apartado de procedimientos.

La devolución del valor se produce utilizando la cláusula RETURN y no mediante variables OUT o INOUT.

La ventaja que presentan las funciones es que pueden ser utilizadas dentro de instrucciones SQL y por tanto aumentan considerablemente las capacidades de este lenguaje.

La utilización de funciones posibilita realizar una código más fácil de leer y mantener, pues agrupa en una función un conjunto de operaciones relacionadas lógicamente entre sí (fórmulas, reglas de negocio...). Además consultas SQL complejas pueden abreviarse con el empleo de funciones.

La mayoría de las opciones de la sintaxis anterior son idénticas a las vistas en el tema de procedimientos. Además de ello:

- La cláusula RETURN es obligatoria e indica el tipo de dato que devuelve la función.
- No aparece en la sintaxis, pues no se puede utilizar, los parámetros IN, OUT, INOUT. Todos los parámetros pasados a la función se definen implícitamente del tipo IN.
- Dentro del cuerpo de la función debe aparecer por lo menos una instrucción RETURN, que devuelve el resultado de la función al programa llamado y finaliza su ejecución. Si la ejecución de la función llegara al final de la misma sin haber encontrado una instrucción RETURN, causaría un error. El seguimiento y mantenimiento de la función será mucho más simple si sólo aparece una sentencia RETURN dentro de la función; en ese caso habrá que utilizar una variable para conseguirlo.

2.8.2 Ejemplos de utilización de funciones

La función funcion1 devuelve en euros las pesetas que se le pasan como argumento.

```
2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion1 $$
4 CREATE FUNCTION funcion1 (p_pesetas DECIMAL(12,2))
5 /* CONVERSION DE PESETAS A EUROS */
6 RETURNS DECIMAL(10,2)
7 BEGIN
8 RETURN (p_pesetas/166.386);
9 END $$
10 DELIMITER ;
```

Función 1 "funcion1"

A continuación se indica cómo llamar a la función anterior creada en la base de datos *test*:


```
mysql> SELECT test.funcion1(10000);
+-----+
| test.funcion1(10000) |
+-----+
| 60.10                |
+-----+
1 row in set, 1 warning (0.00 sec)
```

Ídem de antes pero dejando el resultado de la función en una variable:

```
mysql> SET @v_euros=test.funcion1(10000);
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT @v_euros;
+-----+
| @v_euros |
+-----+
| 60.10    |
+-----+
1 row in set (0.00 sec)
```

En el siguiente ejemplo la función realiza la conversión de pesetas a euros y viceversa: Si el primer argumento es un 1 entonces convertirá a euros, si es un 2 convertirá a pesetas:

```
2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion2 $$
4 CREATE FUNCTION funcion2 (p_moneda TINYINT, p_cantidad DECIMAL(12,2))
5 /* CONVERSOR DE PESETAS <--> EUROS */
6 RETURNS DECIMAL(12,2)
7 BEGIN
8 /* 1--> De pesetas a euros, 2--> De euros a pesetas */
9 IF p_moneda = 1 THEN
10 RETURN (p_cantidad/166.386);
11 ELSE
12 RETURN (p_cantidad * 166.386);
13 END IF;
14 END $$
15 DELIMITER ;
```

Función 2 "funcion2"

Ejemplo de ejecución de la función anterior;

```
mysql> SELECT test.funcion2(2, 6);
+-----+
| test.funcion2(2, 6) |
+-----+
| 998.32              |
+-----+
1 row in set, 1 warning (0.00 sec)
```

La función 2 puede mejorarse dejando un solo RETURN (se ejecutará siempre al ser la última instrucción de la función) y controlando que los dos únicos parámetros que se le pasen son 1 o 2 (si no es así la función devolverá NULL):

```

2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion3 $$
4 CREATE FUNCTION funcion3 (p_moneda TINYINT, p_cantidad DECIMAL(12,2))
5 /* CONVERSION DE PESETAS <--> EUROS */
6 RETURNS DECIMAL(12,2)
7 BEGIN
8 /* 1--> De pesetas a euros, 2--> De euros a pesetas */
9 DECLARE v_resultado DECIMAL (12,2);
10 IF p_moneda = 1 THEN
11     SET v_resultado = (p_cantidad/166.386);
12 END IF;
13 IF p_moneda = 2 THEN
14     SET v_resultado = (p_cantidad * 166.386);
15 END IF;
16 RETURN v_resultado;
17 END $$
18 DELIMITER ;

```

Función 3 "funcion3"

Llamando a la función anterior desde dentro de un procedimiento:

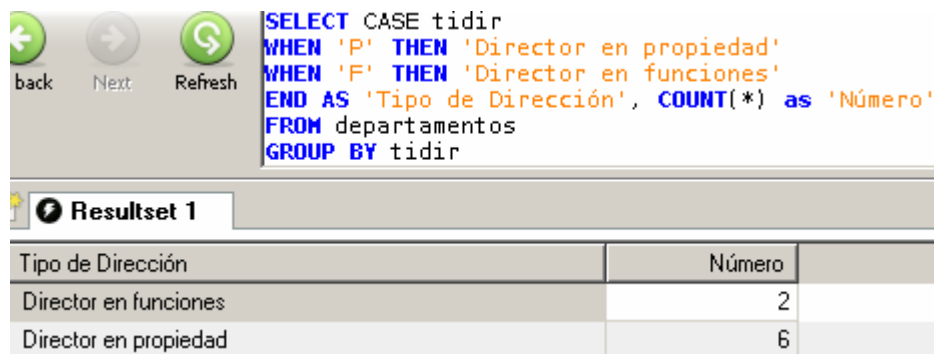
```

2 DELIMITER $$
3 DROP PROCEDURE IF EXISTS f1 $$
4 CREATE PROCEDURE f1 (INOUT p_num DECIMAL(12,2))
5 BEGIN
6     IF p_num >= 0 THEN
7         SET p_num=funcion3 (2, p_num);
8     END IF;
9 END $$
10 DELIMITER ;

```

Procedimiento 47 f1

Si en lugar del cuerpo de un procedimiento se utiliza una instrucción SELECT para llamar a la función:



The screenshot shows a database client interface with a SQL query editor and a results pane. The query is a CASE statement that counts the number of directors in different departments. The results pane shows a table with two columns: 'Tipo de Dirección' and 'Número'.

```

SELECT CASE tidir
WHEN 'P' THEN 'Director en propiedad'
WHEN 'F' THEN 'Director en funciones'
END AS 'Tipo de Dirección', COUNT(*) as 'Número'
FROM departamentos
GROUP BY tidir

```

Tipo de Dirección	Número
Director en funciones	2
Director en propiedad	6

Instrucción 3

La anterior sentencia se puede simplificar a partir de la siguiente función:

```

2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion4 $$
4 CREATE FUNCTION funcion4 (p_tipo CHAR(1))
5 RETURNS VARCHAR(25)
6 BEGIN
7     DECLARE v_tipo_director VARCHAR(25);
8     IF p_tipo = 'P' THEN
9         SET v_tipo_director='Director en propiedad';
10     ELSE
11         SET v_tipo_director='Director en funciones';
12     END IF;
13     RETURN(v_tipo_director);
14 END $$
15 DELIMITER ;

```

Función 4 "funcion4"

Por lo que quedaría reducida a:

```
SELECT funcion4(tidir) AS 'Tipo de Dirección', COUNT(*) as 'Número'
FROM departamentos
GROUP BY tidir
```

Instrucción 4

Si se desea obtener el número de empleados de un departamento que se le pasa como argumento:

```
2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion5 $$
4 CREATE FUNCTION funcion5 (p_departamento INTEGER)
5     RETURNS INTEGER
6     READS SQL DATA
7 BEGIN
8     DECLARE v_num_total INTEGER;
9     SELECT COUNT(*)
10    INTO v_num_total
11   FROM empleados
12  WHERE numde=p_departamento;
13     RETURN(v_num_total);
14 END $$
15 DELIMITER ;
```

Función 5 "funcion5"

La función 5 anterior más simplificada:

```
2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion6 $$
4 CREATE FUNCTION funcion6 (p_departamento INTEGER)
5     RETURNS INTEGER
6     READS SQL DATA
7 BEGIN
8     RETURN (SELECT COUNT(*) FROM empleados WHERE numde=p_departamento);
9 END $$
10 DELIMITER ;
```

Función 6 "funcion6"

Para obtener el nombre de un alumno cuyo número de identificación se pasa como parámetro (devuelve nulo si este no existe) puede emplearse la siguiente función:

```
2 DELIMITER $$
3 DROP FUNCTION IF EXISTS funcion7 $$
4 CREATE FUNCTION funcion7 (p_id INTEGER)
5     RETURNS VARCHAR(30)
6     READS SQL DATA
7 BEGIN
8     DECLARE v_nombre VARCHAR(30);
9     SELECT alumno
10    INTO v_nombre
11   FROM alumnos
12  WHERE id =p_id;
13     RETURN (v_nombre);
14 END $$
15 DELIMITER ;
```

Función 7 "funcion7"

2.8.3 Modificación de funciones

Sintaxis:

```
ALTER FUNCTION nombre_función
    {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
    | SQL SECURITY {DEFINER|INVOKER}
    | COMMENT comentario
```

Se debe poseer el privilegio de ALTER ROUTINE para poder modificar procedimientos y funciones. El uso de las distintas opciones se ha expuesto en el apartado de creación de procedimientos.

2.8.4 Borrado de funciones

Sintaxis:

```
DROP FUNCTION [IF EXISTS] nombre_función
```

Al igual que ocurre para la modificación, se debe poseer el privilegio de ALTER ROUTINE para poder borrar procedimientos y funciones.

2.9 Triggers

2.9.1 Creación de triggers. Diccionario de Datos

Los triggers, también llamados desencadenadores o disparadores, son programas almacenados que se ejecutan (“disparan”) automáticamente en respuesta a algún suceso que ocurre en la base de datos. En MySQL ese tipo de suceso se corresponde con alguna instrucción DML (INSERT, UPDATE, DELETE) sobre alguna tabla.

Suponen un mecanismo para asegurar la integridad de los datos. Se emplean también como un método para realizar operaciones de auditoría sobre la base de datos. No hay que abusar de su utilización pues ello puede traer consigo una sobrecarga del sistema y por tanto un bajo rendimiento del mismo. Al final de este apartado verás ejemplos de su utilización que ayudarán a comprender su utilización.

Sintaxis:

```
CREATE [DEFINER={cuenta_usuario | CURRENT_USER}] TRIGGER
nombre_trigger
    {BEFORE | AFTER}
    {UPDATE | INSERT | DELETE}
    ON tabla
    FOR EACH ROW
    cuerpo_del_trigger
```

Para poder crear triggers, en versiones anteriores a la 5.16 se necesita el privilegio SUPER. A partir de esta el privilegio es el de TRIGGER. No se pueden crear ni sobre una tabla temporal ni sobre una vista, solamente sobre tablas.

Aspectos a comentar de la sintaxis anterior:

- DEFINER = {cuenta_usuario | CURRENT_USER } indica con qué privilegios se ejecutan las instrucciones del trigger. La opción por defecto es CURRENT_USER, que indica que las instrucciones se ejecutan con los privilegios del usuario que lanzó la instrucción de creación del trigger. La opción cuenta_usuario por otro lado hace que el trigger se ejecute con los privilegios de dicha cuenta.
- Nombre del trigger. Sigue las mismas normas que para nombrar cualquier objeto de la base de datos.
- BEFORE | AFTER. Señala cuando se ejecuta el trigger, antes (before) o después (after) de la instrucción DML que lo provocó.
- UPDATE | INSERT | DELETE. Define la operación DML asociada al trigger.
- ON tabla. Define la tabla base asociada al trigger.
- FOR EACH ROW. Indica que el trigger se ejecutará por cada fila de la tabla afectada por la operación DML. Esto es, si tenemos asociado un trigger a la operación de borrado de una tabla y se eliminan con una sola instrucción 6 filas de ésta última, el trigger se ejecutará 6 veces, una por cada fila eliminada. Otros gestores de bases de datos (así como futuras implementaciones de MySQL) consideran también el otro estándar de ANSI, la cláusula FOR EACH STATEMENT. Con esta segunda opción, el trigger se ejecutaría por cada operación DML realizada; en el ejemplo anterior, la instrucción de borrado daría lugar a que sólo se ejecutara el trigger una sola vez en lugar de 6 (filas afectadas) con esta futura cláusula.
- Cuerpo del trigger: El conjunto de instrucciones que forman este programa almacenado. Si son más de una irán en un bloque BEGIN ... END. No pueden contener una instrucción CALL de llamada a un procedimiento almacenado.

Referencias a las columnas afectadas

Dentro del cuerpo del trigger se puede hacer referencia a los valores de las columnas que están siendo modificadas con una sentencia DML (la que provocó que se disparara el trigger), incluso pueden cambiarse si así se considerara. Para ello se utilizan los objetos NEW y OLD. De esta manera, en un trigger de tipo BEFORE UPDATE afectando a una columna *micolumna*, se utilizará la expresión *OLD.micolumna* para conocer el valor de la columna antes de ser modificado y *NEW.micolumna* será el nuevo valor de la columna después de la modificación. Estos dos valores sólo tienen sentido los dos juntos en una modificación, pues ante una inserción (INSERT) no existe valor antiguo (OLD) y ante un borrado (DELETE) no existe un valor nuevo (NEW) pues el que existe se elimina. Ver ejemplo *trigger1* del apartado 2.9.3.

Dentro de un trigger de tipo BEFORE, puede cambiarse el nuevo valor mediante una sentencia de asignación SET por lo que anularía por completo el efecto de la instrucción DML que provocó el trigger.

Eventos de disparo

Como se ha indicado, un trigger se ejecuta automáticamente (dispara) antes o después de una instrucción DML: INSERT, UPDATE o DELETE.

Además de la forma explícita anterior, pueden también ejecutarse las instrucciones del cuerpo del trigger si la modificación se produce de forma implícita, como sería el caso de una instrucción REPLACE pues en realidad equivale a una instrucción DELETE seguida de una INSERT (por lo tanto aquí se ejecutarían los triggers asociados a estas dos operaciones).

¿Triggers de tipo before o after?

Prácticamente no hay diferencia. La ventaja que puede suponer utilizar los triggers de tipo BEFORE es que pueden cambiarse los valores modificados inicialmente por una instrucción UPDATE o INSERT mientras que con los triggers de tipo AFTER daría un error de ejecución.

Teniendo en cuenta los 2 tipos de triggers y las tres operaciones DML distintas, podríamos tener tener 6 triggers por tabla:

BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE, BEFORE DELETE y AFTER DELETE.

La vista TRIGGERS del diccionario de datos contendrá toda la información sobre cada trigger, existiendo una entrada (fila) en esta vista por cada trigger en el servidor; la información que aparece en las columnas es la que se ha tenido en cuenta a la hora de crearlo.

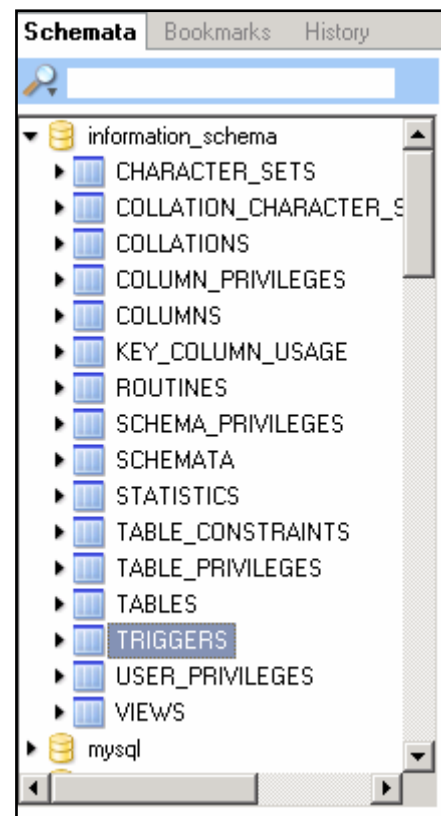


Ilustración 193. Vista triggers del diccionario de datos

2.9.2 Borrado de triggers

Sintaxis:

```
DROP TRIGGER nombre_trigger
```

Se necesita el privilegio SUPER para borrar triggers.

2.9.3 Utilización de triggers

Ejemplos de utilización de triggers:

Copias o réplicas de datos:

Finalidad: Mantener sincronizada una copia de seguridad de unos datos.

Para probar el ejemplo que viene a continuación lanzar antes la tabla:

```
CREATE TABLE alumnos_replica
(id      INT PRIMARY KEY,
 alumno VARCHAR(30))
ENGINE=innodb;
```

Puedes volver a crear si así lo deseas la tabla *alumnos* mediante el *procedimiento1.sql*.

Si creas el siguiente trigger:

```
2 DELIMITER $$
3 CREATE TRIGGER trigger1
4 BEFORE INSERT ON alumnos
5 FOR EACH ROW
6 BEGIN
7 INSERT INTO alumnos_replica VALUES (NEW.id, NEW.alumno);
8 END$$
9 DELIMITER ;
10
```

Trigger 1 "trigger1"

podrás comprobar que la siguiente instrucción:

```
INSERT INTO ALUMNOS VALUES (20, 'Alberto Carrera');
```

Instrucción 5

crea una fila nueva en la tabla *alumnos* como consecuencia de la *instrucción 5* anterior y otra también idéntica en la tabla *alumnos_replica* debido a la ejecución automática del conjunto de instrucciones del trigger.

Otras situaciones en las que se pueden emplear triggers podrían ser para llevar el mantenimiento del stock de artículos: En el momento en que se haga un pedido, se decrementa automáticamente el número de unidades pedidas de las existencias de ese artículo.

Auditoría:

Finalidad: Auditar las operaciones que se realizan sobre una tabla.

Antes de lanzar el siguiente ejemplo, utilizaremos la tabla:

```
CREATE TABLE AUDITA (mensaje VARCHAR(200));
```

Si editas el siguiente trigger:

```
2 DELIMITER $$
3 CREATE TRIGGER trigger2
4   AFTER UPDATE ON alumnos
5   FOR EACH ROW
6 BEGIN
7   INSERT INTO audita
8     VALUES (CONCAT('Modificacion realizada por ', USER(), ' el dia ', NOW(),
9                   ' Valores antiguos: ', OLD.id, ' y ', OLD.alumno,
10                  ' Valores nuevos: ', NEW.id, ' y ', NEW.alumno));
11 END$$
12 DELIMITER ;
```

Trigger 2 trigger 2

la instrucción siguiente:

```
UPDATE ALUMNOS
SET alumno = 'Mario Carrera'
WHERE id=20;
```

Instrucción 6

modificará la columna alumno de la tabla *alumnos* dejando a Mario como nuevo nombre y por efecto del trigger creará la siguiente entrada en la tabla audita:

mensaje
Modificacion realizada por root@localhost el dia 2006-06-27 17:49:29 Valores antiguos: 20 y Alberto Carrera Valores nuevos: 20 y Mario Carrera

Ilustración 204. Utilización de triggers para auditar las operaciones en la tabla

En relación a este último trigger ¿Qué pasaría si el nombre del alumno fuera nulo?

Validación de datos

Además de las constraints o restricciones que se definen en el momento de crear las tablas, podemos utilizar los triggers para validar los datos a almacenar en una tabla y de esta manera mantener la consistencia de los mismos.

Ejemplos de control de entrada de datos:

- El valor de una columna no debe ser negativo y estar comprendido entre 1 y 80.
- Un empleado que no sea vendedor no puede tener comisión.
- Un empleado no puede ser jefe de si mismo.
- ...

Siempre que nos encontremos con alguna situación similar a las anteriores no debemos dejar que se realice la modificación de la tabla.

Un aspecto importantísimo a tener en cuenta es que la combinación operación DML + trigger asociado o trigger asociado + operación DML. Si el trigger falla, entonces falla la operación DML que intenta modificar la tabla de la base de datos. Esto puede ser útil para evitar entradas indeseadas en las tablas de la base de datos. De esta manera si en el

cuerpo del trigger se detecta que el dato no es correcto podría provocarse una situación de error que anulara y abortara la operación DML. El problema es que hasta la versión 5.2 por lo menos no aparecerá la instrucción SIGNAL (RAISE) que permita provocar errores como en otros gestores de bases de datos.

No se puede utilizar SQL dinámico para simular una instrucción SIGNAL pues los triggers no admiten SQL dinámico. Otra estrategia a utilizar en este caso para forzar una situación de error y echar atrás el resultado de la sentencia DML que intenta modificar los datos es realizar una SELECT en el cuerpo del trigger que no recupere ninguna fila. Al ocurrir un error en el trigger, la combinación trigger + operación DML falla y por tanto aborta la ejecución de la operación DML (ídem si la combinación es operación DML + trigger).

Supongamos que no se permite altas de alumnos con identificador negativo o cero como primera columna de la tabla alumnos. Bastará con que codifiquemos un trigger del tipo:

```

2 DELIMITER $$
3 CREATE TRIGGER trigger3
4   BEFORE INSERT ON alumnos
5   FOR EACH ROW
6 BEGIN
7   DECLARE v_valor_inexistente VARCHAR(15);
8   IF NEW.id <= 0 THEN
9     SELECT 'Provocar error'
10    INTO v_valor_inexistente
11    FROM alumnos
12    WHERE id = -20000;
13   END IF;
14 END $$
15 DELIMITER ;

```

Trigger 3 "trigger3"

Si intentamos realizar primera de las dos siguientes inserciones que vienen a continuación, la de una alumna con clave negativa, el trigger fallará y por tanto no se llevará a cabo la operación DML que se ejecuta después (la inserción de la fila) como puede comprobarse después de listar la tabla alumnos. En cambio una inserción de un id superior a 0 hará que el trigger finalice correctamente no provocando situación de error y por tanto permitiendo después lanzar la operación DML de inserción:

```

C:\Archivos de programa\MySQL\MySQL Server 5.0\bin\mysql.exe
mysql> USE TEST
Database changed
mysql> INSERT INTO ALUMNOS VALUES (-1, 'Blanca Bailín');
ERROR 1329 (02000): No data - zero rows fetched, selected, or processed
mysql> INSERT INTO ALUMNOS VALUES (10, 'Blanca Bailín');
Query OK, 1 row affected (0.02 sec)

mysql> SELECT * FROM alumnos;
+----+-----+
| id | alumno |
+----+-----+
| 1  | alumno 1 |
| 2  | alumno 2 |
| 3  | alumno 3 |
| 4  | alumno 4 |
| 5  | alumno 5 |
| 10 | Blanca Bailín |
| 20 | Mario Carrera |
| 21 | Raquel Carrera |
+----+-----+
8 rows in set (0.00 sec)

mysql>

```

Ilustración 25. Prueba del comportamiento de los triggers

Anexo: MySQL desde otras aplicaciones

1 Introducción: Ventajas y desventajas

Aunque sea de manera demostrativa, no se podía terminar este tema sin exponer cómo MySQL puede ser utilizado desde otros entornos y lenguajes de programación externos a él como son Access, PHP, Perl, Java, Python, C, C++ o .NET (C#, Visual Basic .NET) entre otros. Para ilustrar alguno de ellos se ha utilizado PHP 5.14 y la versión de Visual Basic Express 2005, por ser ampliamente utilizados entre la comunidad de programadores y por poderse obtener y utilizar de manera gratuita. La instalación y utilización de estos lenguajes así como los controladores de datos queda fuera del ámbito de este curso por lo que al alumno nunca se le va a exigir ninguno de los conceptos aquí expuestos en ningún tipo de ejercicio o prueba.

Utilizar los elementos de MySQL desde otros entornos externos facilita:

- En entornos cliente / servidor, la carga en el lado del cliente se reduce considerablemente.
- Mantener actualizadas constantemente las versiones de las aplicaciones cliente es mucho más costoso que hacerlo si estas se encuentran centralizadas.
- Las aplicaciones cliente son más pequeñas al encontrarse desarrolladas muchas de las tareas en forma de programas almacenados.
- Una sola unidad de programa puede ser empleada por entornos totalmente diferentes como Java o .NET.
- Más facilidad en la portabilidad / escalabilidad de las aplicaciones cambiando la lógica solamente desde dentro de los programas almacenados.
- Disminución del tráfico de red al ejecutarse las operaciones en el servidor.
- Mecanismos de mayor seguridad evitando que el usuario acceda a la base de datos salvo a aquellos programas almacenados para los que esté autorizado.

Por otro lado, también puede dar lugar a la aparición de las siguientes desventajas:

- El rendimiento algunas operaciones (búsquedas con patrón, manejo de cadenas...) puede llegar a ser menor si se utilizan programas almacenados en lugar de hacerlo en otros lenguajes como PHP, Java o Perl.
- Pueden provocar una fragmentación lógica si una parte de la tarea se implementa con programas almacenados y la otra reside dentro de los programas cliente. Además depurar una aplicación puede llegar a ser dificultoso pues no existe la posibilidad de un solo depurador entre distintos entornos.
- Gran dificultad a la hora de portar o migrar los programas almacenados a otros gestores de bases de datos (sólo DB2 de IBM y MySQL cumplen el estándar ANSI para programas almacenados).

2 Ejecución de procedimientos almacenados. Tratamiento de errores

Los ejemplos que vienen a continuación realizan una llamada al procedimiento *resultset1* visto en el apartado 2.5.5 de este tema (este último envía número de departamento, nombre y presupuesto de los departamentos cuyo centro se le pasa como argumento).

2.1 En PHP

El script de este apartado recoge un número de Departamento mediante un formulario de entrada y manda el dato (*id_centro*) al procedimiento almacenado *resultset1* para que este le envíe el número de departamento, nombre y presupuesto de los departamentos. Ej. de ejecución:

Ilustración 216 (dcha.). Resultado de llamar al procedimiento *resultset1* desde PHP



Nº Departamento	Nombre	Presupuesto
110	DIRECCION COMERCIAL	15000000
111	SECTOR INDUSTRIAL	11000000
112	SECTOR SERVICIOS	9000000

```

1 <html><head><title>Departamentos de un Centro</title><head><body>
2 <h1>Departamentos del Centro</h1>
3 <form method="post" >
4 <p>Introduzca número de centro:
5 <input type="text" name="id_centro" size="4">
6 <input type="submit" name="submit" value="Consultar"><p>
7 </form>
8
9 <?php
10 /* Llamada al procedimiento almacenado resultset1, recuperando
11 los Departamentos del Centro que se le envía en el formulario */
12
13 $hostname = "localhost";
14 $username = "root";
15 $password = "-----";
16 $database = "test";
17
18 if (isset ($_POST['submit']) and !empty ($_POST['id_centro'])) {
19
20 /*Creamos un objeto conexión a partir del servidor, usuario y base datos*/
21 $dbh = new mysqli($hostname, $username, $password, $database);
22
23 /* Comprobar conexión */
24 if (mysqli_connect_errno()) {
25     printf("Conexión fallida: %s\n", mysqli_connect_error());
26     exit ();
27 }
28 $id_centro = $_POST['id_centro'];
29 /* Llamada al procedimiento almacenado resultset */
30 if ($result_set = $dbh->query("call resultset1( $id_centro )")) {
31     print ('<table border="1" width="30%"> <tr> '.
32         '<td>Nº Departamento</td><td>Nombre</td><td>Presupuesto</td></tr>');
33     /*Recorrido del conjunto de filas devueltas */
34     while ($row = $result_set->fetch_object()) {
35         /* Impresión de cada uno de los 3 campos:numde, nomde y presu*/
36         printf("<tr><td>%s</td><td>%s</td><td>%s</td></tr>\n",
37             $row->numde, $row->nomde, $row->presu);
38     }
39 }
40 /* En caso de error en la llamada se imprime el código de error MySQL,
41 el SQLSTATE y el texto del error*/
42 else {
43     printf("<p>Error:%d (%s) %s\n", mysqli_errno($dbh),
44         mysqli_sqlstate($dbh), mysqli_error($dbh));
45 }
46 print ("</table> ");
47 /*Cerrando la conexión */
48 $dbh->close();
49 }
50 ?>
51 </body></html>

```

Ilustración 227. LLamando a un procedimiento desde PHP

El mismo ejemplo pero sin hacer uso del procedimiento y ejecutando la sentencia Select directamente:

Ilustración 238 (dcha). El mismo resultado pero sin hacer uso del procedimiento

```

29 /*.....
30     $id_centro = $_POST['id_centro'];
31
32
33     $sql="SELECT numde, nomde, presu
34         FROM departamentos
35         WHERE numce = '$id_centro' " ;
36
37     if ($result_set = $dbh->query($sql)) {
38 /*.....

```

2.2 En Visual Basic Express 2005

La llamada al procedimiento *vbnet1* que viene a continuación se realiza desde la ventana *inmediato*, los resultados se visualizan por la ventana de *resultados*:

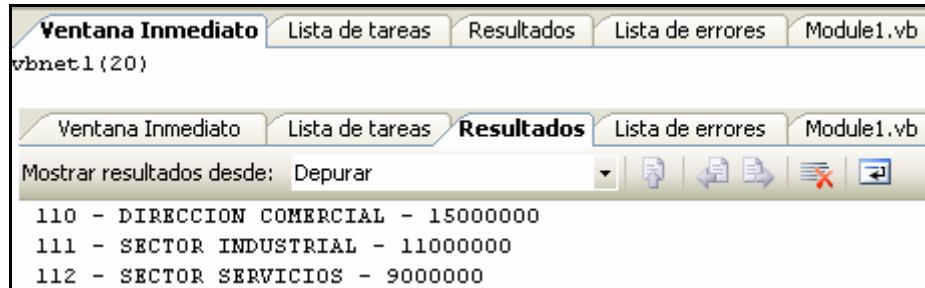


Ilustración 249. Ejecución del procedimiento y resultados

```
Imports MySql.Data.MySqlClient
Module Module1

Sub vbnet1(ByVal centro As Integer)
    'Procedimiento que llama al procedimiento MySQL resultset1
    'enviándole el argumento centro que recibe
    Dim host As String = "192.168.0.5"
    Dim usuario As String = "root"
    Dim clave As String = "-----"
    Dim base_datos As String = "test"
    Dim cadena_conexion As String = "Database=" & base_datos & _
        " ;Data Source=" & host & _
        ";User Id=" & usuario & ";Password=" & clave
    'Creación de la conexión
    Dim MiConexion As New MySqlConnection(cadena_conexion)
    Try
        MiConexion.Open()
        'Creación de un nuevo comando
        Dim instruccion As MySqlCommand = New MySqlCommand("resultset1", MiConexion)
        'El nuevo comando es de tipo procedimiento almacenado
        instruccion.CommandType = CommandType.StoredProcedure
        'Creación del parámetro que se enviará como argumento
        Dim parametrol As MySqlParameter
        'Asociando el parámetro creado de VB 2005 al argumento p_numce
        'del procedimiento resultset1 de MySQL y definiendo su tipo (entero)
        parametrol = instruccion.Parameters.Add("p_numce", MySqlDbType.Int32)
        'Asignando al parámetro el valor recibido en el procedimiento VB 2005
        parametrol.Value = centro
        'El conjunto de filas y columnas resultantes de llamar
        'al procedimiento se almacenará en un objeto de tipo DataReader
        Dim MyReader As MySqlDataReader = instruccion.ExecuteReader
        'Recorrido de las filas del DataReader
        While MyReader.Read
            'Impresión de cada una de sus columnas
            Console.WriteLine(MyReader.GetInt32(0))
            Console.WriteLine(" - " & MyReader.GetString(1))
            Console.WriteLine(" - " & MyReader.GetInt32(2))
        End While
        MyReader.Close()
    Catch Exception As MySqlException
        Console.WriteLine("Ocurrió un error: ")
        Console.WriteLine(Exception.Message)
    End Try
    MiConexion.Close()
End Sub
```

El mismo ejemplo pero utilizando directamente la instrucción Select

```

'.....
Try
    MiConexion.Open()
    Dim cadenasselect As String
    cadenasselect = "Select numde, nomde, presu from departamentos where numce= " & centro
    Dim instruccion As MySqlCommand = New MySqlCommand(cadenasselect, MiConexion)
    Dim MyReader As MySqlDataReader = instruccion.ExecuteReader
    While MyReader.Read
        Console.WriteLine(MyReader.GetInt32(0))
        Console.WriteLine(" - " & MyReader.GetString(1))
        Console.WriteLine(" - " & MyReader.GetInt32(2))
    End While
    MyReader.Close()
Catch Exception As MySqlException
'.....

```

3 Ejecución de funciones

Los dos ejemplos que vienen a continuación realizan una llamada a la función *funcion7* vista en el apartado 2.8.2 de este tema (devuelve el nombre de un alumno a partir de un identificador de alumno que se le pasa como parámetro). Se utilizan sentencias preparadas tanto en este apartado como en el 2.10.5.

3.1 En PHP

El fragmento de script de este apartado recoge un número matrícula de alumno mediante un formulario de entrada y envía el dato (*alumno_id*) a la función *funcion7* para que este devuelva el nombre. Ej. de ejecución

Nombre del alumno

Introduzca número de matrícula del alumno:

El nombre del alumno es alumno 2

Ilustración 30. Resultado de llamar a la función “funcion7” desde PHP

```

26 /*.....
27     $alumno_id = $_POST['alumno_id'];
28     /* Preparación de la instrucción de llamada
29     con el parámetro ? que se le envía a la función */
30     $stmt=$dbh->prepare("SELECT funcion7(?)") or die($my_db->error);
31     /* Asociar la variable PHP con el parámetro ? SQL
32     El primer argumento 'i' indica el tipo: i--> integer*/
33     $stmt->bind_param('i',$alumno_id) or die($stmt->error);
34     /* Ejecución de la instrucción preparada */
35     $stmt->execute( ) or die($stmt->error);
36     /* Resultado de la ejecución de la función en la variable nombre */
37     $stmt->bind_result($nombre);
38     /* Recuperación del valor devuelto por la función*/
39     $stmt->fetch( );
40     if (is_null ($nombre))
41         printf ("No existe alumno con número de matrícula %d", $alumno_id);
42     else
43         printf ("El nombre del alumno es %s", $nombre);
44 /*.....

```

Ilustración 25. Llamando a una función desde PHP

3.2 En Visual Basic Express 2005

```

Sub vbnet2(ByVal alumno As Integer)
    'Llamar a la función MySQL enviando como argumento el alumno
    'y recibiendo (return) de ésta en el objeto parametro_devuelto el nombre del mismo
    Dim host As String = "192.168.0.5"
    Dim usuario As String = "root"
    Dim clave As String = "-----"
    Dim base_datos As String = "test"
    Dim cadena_conexion As String = "Database=" & base_datos & _
        " ;Data Source=" & host & _
        " ;User Id=" & usuario & " ;Password=" & clave
    Dim MiConexion As New MySqlConnection(cadena_conexion)
    Try
        MiConexion.Open()
        Dim instruccion As MySqlCommand = New MySqlCommand("funcion7", MiConexion)
        instruccion.CommandType = CommandType.StoredProcedure
        Dim parametro_enviado As MySqlParameter
        parametro_enviado = instruccion.Parameters.Add("p_id", MySqlDbType.Int32)
        parametro_enviado.Value = alumno
        Dim parametro_devuelto As MySqlParameter = _
        instruccion.Parameters.Add("parametro_devuelto", MySqlDbType.String)
        parametro_devuelto.Direction = ParameterDirection.ReturnValue
        instruccion.ExecuteNonQuery()
        Console.WriteLine("Nombre del alumno =" & parametro_devuelto.Value)
    Catch Exception As MySqlException
        '.....
    End Try
End Sub

```

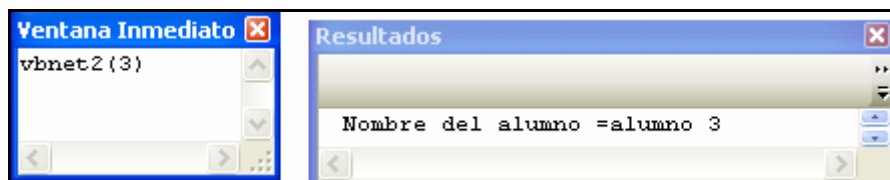


Ilustración 32. Ejecución del procedimiento y resultados

4 Ejecución de sentencias DDL

Los dos ejemplos que vienen a continuación crean una sencilla tabla en MySQL.

4.1 En PHP

Nota: En el siguiente apartado 5.1 también se utilizan instrucciones DDL dentro del script PHP.

El resultado de realizar dos llamadas al script PHP de creación de la tabla viene a continuación en las dos próximas ilustraciones:

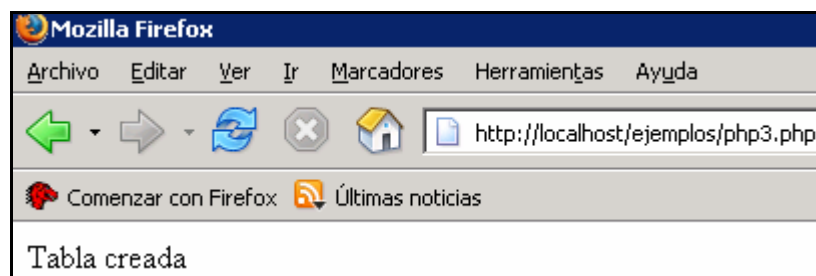
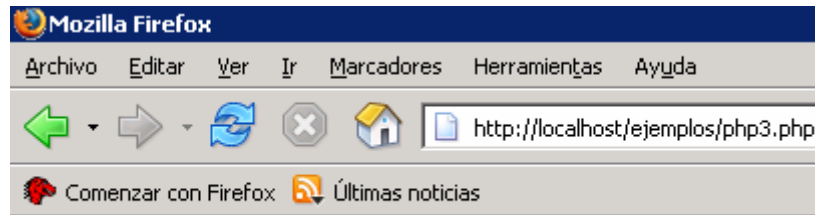


Ilustración 263. Resultado de ejecutar el script PHP de creación de una tabla



Fallo al crear la tabla 1050: (42S01) Table 'tabla1' already exists

Ilustración 34. Resultado de ejecutar por segunda vez el script PHP de creación de una tabla

```

15 /*.....
16 /*Sobraría especificar la base de datos test
17  pues ya se ha hecho en la conexión */
18 $dbh->query("CREATE TABLE test.tabla1 (id INTEGER)");
19 if ($dbh->errno <> 0 ) {
20     printf("Fallo al crear la tabla %d: (%s) %s\n"
21         , $dbh->errno, $dbh->sqlstate, $dbh->error);
22 }
23 else
24     printf("Tabla creada\n");
25 /*.....
  
```

Ilustración 275. Creando una tabla desde PHP

4.2 En Visual Basic Express 2005

```

'.....
Dim MiConexion As New MySqlConnection(cadena_conexion)
Dim cadena_sql As String = "CREATE TABLE tabla1 (id INTEGER)"
Dim instruccion As MySqlCommand = New MySqlCommand(cadena_sql, MiConexion)
Try
    MiConexion.Open()
    instruccion.ExecuteNonQuery()
    Console.WriteLine("Tabla creada")
Catch Exception As MySqlException
    Console.WriteLine("Error en la creación: ")
    Console.WriteLine(Exception.Message)
End Try
'.....
  
```

5 Ejecución de sentencias preparadas

Relacionado con el apartado 2.5.7 de este tema.

5.1 En PHP

El siguiente script tiene la misma funcionalidad que el *procedimiento1* visto en este tema (apartado 2.5.4) en el que se creaba la tabla *alumnos* y se rellenaba con 5 de ellos.


```

1 <?php
2 /* Sentencias preparadas
3 /*****
4 $hostname = "localhost";
5 $username = "root";
6 $password = "-----";
7 $database = "test";
8
9 $dbh = new mysqli($hostname, $username, $password, $database);
10 /* Comprobar conexión */
11 if (mysqli_connect_errno()) {
12     printf("Conexión fallida: %s\n", mysqli_connect_error());
13     exit ();
14 }
15
16 $dbh->query("DROP TABLE IF EXISTS alumnos");
17 if ($dbh->errno <> 0 ) {
18     printf("Fallo al borrar la tabla alumnos %d: (%s) %s\n"
19         , $dbh->errno, $dbh->sqlstate, $dbh->error);
20 }
21 else
22     printf("Tabla alumnos borrada\n");
23
24 $dbh->query("CREATE TABLE alumnos
25         (id INT PRIMARY KEY,
26         alumno VARCHAR(30))
27         ENGINE=innodb");
28 if ($dbh->errno <> 0 ) {
29     printf("Fallo al crear la tabla %d: (%s) %s\n"
30         , $dbh->errno, $dbh->sqlstate, $dbh->error);
31 }
32 else
33     printf("Tabla alumnos creada\n");
34
35 #Preparando la instrucción
36 $insert_stmt=$dbh->prepare("INSERT INTO alumnos VALUES(?,?)")
37     or die($dbh->error);
38 #Asociando las variables a los parametros ?, #i= parametro entero #s=cadena
39 $insert_stmt->bind_param("is", $contador, $nombre);
40 for ($contador = 1; $contador <= 5; $contador++) {
41     $nombre="alumno ".$contador;
42     #Ejecución de la instrucción preparada
43     $insert_stmt->execute( ) or die ($insert_stmt->error);
44 }
45 $insert_stmt->close( );
46
47 $dbh->close();
48 ?>

```

Ilustración 286. Sentencias preparadas en PHP

Llamando directamente al procedimiento 1:

```

32 /*.....
33     printf("Tabla alumnos creada\n");
34
35
36 $sql = 'call procedimiento1( )';
37 $dbh->query($sql);
38 if ($dbh->errno) {
39     die("Ejecución procedimiento1 fallida: ".$dbh->errno." : ".$dbh->error);
40 }
41 else {
42     printf("Procedimiento 1 ejecutado correctamente\n");
43 }
44 /*.....

```

Ilustración 297. Inserción de datos en la tabla sin utilizar sentencias preparadas

5.2 En Visual Basic Express 2005

```

'Inserción en la tabla alumnos de los alumnos:
'alumno 10, alumno 11... alumno 20 mediante SENTENCIAS PREPARADAS
'.....
Dim MiConexion As New MySqlConnection(cadena_conexion)
Dim cadena_sql As String = "INSERT INTO ALUMNOS VALUES (?nuevo_id,?nuevo_alumno)"
'Creando un nuevo comando asociado a la cadena Select anterior
Dim instruccion As MySqlCommand = New MySqlCommand(cadena_sql, MiConexion)
Try
    MiConexion.Open()
    'Creando el parámetro parametro_id
    Dim parametro_id As MySqlParameter
    'Asociándolo con el comando creado
    parametro_id = instruccion.Parameters.Add("?nuevo_id", MySqlDbType.Int32)
    'Creando el segundo parámetro parametro_alumno
    Dim parametro_alumno As MySqlParameter
    'Asociándolo con el comando creado
    parametro_alumno = instruccion.Parameters.Add("?nuevo_alumno", MySqlDbType.String)
    Dim contador As Integer
    For contador = 10 To 20
        parametro_id.Value = contador
        parametro_alumno.Value = "alumno " + contador.ToString
        'En cada iteración, inserción del alumno "contador"
        instruccion.ExecuteNonQuery()
    Next
Catch Exception As MySqlException
    Console.WriteLine("Error en la inserción: ")
    Console.WriteLine(Exception.Message)
End Try
'.....

```