

TEMA 2: AUTOMATIZACIÓN DE TAREAS: CONTRUCCIÓN DE GUIONES DE ADMINISTRACIÓN

1. HERRAMIENTAS PARA AUTOMATIZAR TAREAS

Las rutinas (procedimientos y funciones) almacenadas son un conjunto de comandos SQL que pueden almacenarse en el servidor.

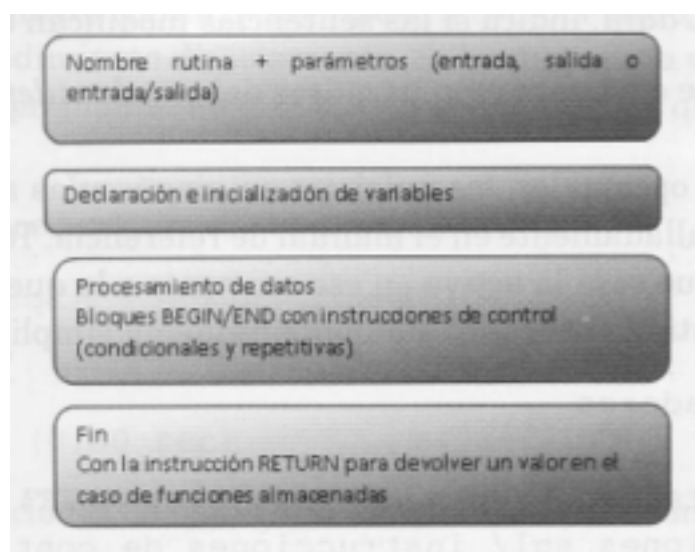
Una vez que se hace, los clientes no necesitan lanzar cada comando individual sino que pueden en su lugar llamar al procedimiento almacenado como un único comando. Las rutinas almacenadas pueden mejorar el rendimiento ya que se necesita enviar menos información entre el servidor y el cliente.

- Una función almacenada es un programa almacenado que devuelve un valor a través de cero o un único valor de retorno.
- Los procedimientos almacenados pueden devolver valores a través de parámetros OUT o INOUT.

A diferencia de los procedimientos almacenados, las funciones almacenadas se pueden utilizar en expresiones y pueden incluirse en otras funciones o procedimientos así como en el interior de sentencias SQL como SELECT, UPDATE, DELETE e INSERT.

1.1.SINTAXIS DE RUTINAS ALMACENADAS.

El esquema general de una rutina almacenada se resume en la siguiente imagen:



- **Sintaxis general para la creación de un procedimiento:**

```
CREATE PROCEDURE nom_procednto ([IN | OUT | INOUT] nom_param] tipo)
BEGIN
Routine_body
END;
```

- **Sintaxis general para la creación de una función:**

```
CREATE FUNCTION nom_funcion ([ [nom_param_Entrada tipo] , ...])
RETURNS tipo_param_Salida
BEGIN
DECLARE nombre_param_Salida tipo_param_Salida;
Routine_body
RETURN (nom_param_Salida);
END;
```

- routine_body: es el cuerpo de la rutina formado generalmente por sentencias SQL (terminadas en ;)

Todos los procedimientos o funciones se crean asociados a una base de datos que será la activa en ese momento o la que pongamos como prefijo en el nombre del mismo.

Si queremos comprobar las rutinas existentes:

```
mysql> SELECT * from Information_Schema.Routines;
```

Si queremos ver una rutina en concreto:

```
mysql> SHOW CREATE PROCEDURE|FUNCTION nombre_rutina;
```

EJEMPLO 1.

```
DELIMITER $$
```

```
DROP PROCEDURE IF EXISTS hola_mundo$$
```

```
CREATE PROCEDURE prueba.hola_mundo()
```

```
BEGIN
```

```
SELECT 'hola mundo';
```

```
END $$
```

```
DELIMITER ;
```

Es un procedimiento muy simple que imprime por pantalla la cadena 'hola mundo'. Este y otros ejemplos más complejos servirán de base para explicar su sintaxis. A continuación lo explicamos detalladamente por líneas:

- Línea 1. La palabra clave DELIMITER indica el carácter de comienzo y fin del procedimiento. Típicamente sería un ; pero dado que necesitamos un ; para cada sentencia SQL dentro del procedimiento es conveniente usar otro carácter (normalmente \$\$ o //).
- Línea 2. Eliminamos el procedimiento si es que existe. Esto evita errores cuando queremos modificar un procedimiento existente.
- Línea 3. Indica el comienzo de la definición de un procedimiento donde debe aparecer el nombre seguido por paréntesis entre los que pondremos los parámetros en caso de haberlos. En este caso precedemos al nombre con la base de datos test a la que pertenecerá el procedimiento.
- Línea 4. Begin indica el comienzo de una serie de bloques de sentencias sql que componen el cuerpo del procedimiento cuando hay mas de una.
- Línea 5. Conjunto de sentencias SQL, en este caso un select que imprime la cadena por pantalla.
- Línea 6. Fin de la definición del procedimiento seguido de un doble \$ indicando que ya hemos terminado.

En caso de encontrarnos en un cliente podemos ejecutar el código del procedimiento directamente desde la consola. Si hemos usado un editor guardaremos el código en un fichero con un nombre y extensión apropiados, en este caso ejemplo1.sql que ejecutaremos desde el cliente con el comando [*source*](#).

Una vez creado estamos en condiciones de ejecutarlo llamándolo con el comando **CALL**:

```
mysql> call hola_mundo() $$
+-----+
| Hola mundo |
+-----+
| Hola mundo|
+-----+
1 row in set (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```

EJEMPLO 2.

```
CREATE PROCEDURE ejem_version()
```

```
SELECT versión(); $$
```

En este ejemplo mostramos directamente la versión de MySQL usando la función versión. Las sentencias BEGIN y END solo son necesarias en caso de tener más de una sentencia.

EJEMPLO 3.

```
DELIMITER $$
```

```
CREATE PROCEDURE fecha()
```

```
SELECT CURRENT_DATE;
```

```
$$
```

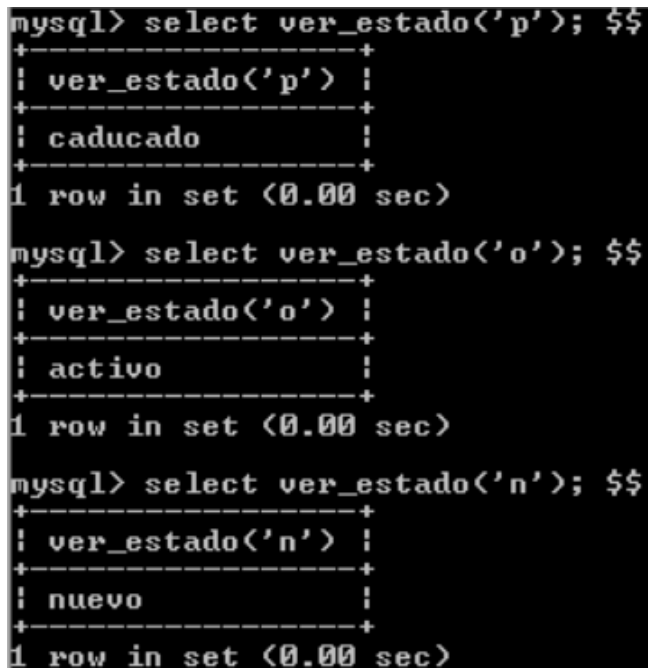
En este ejemplo obtenemos la fecha actual.

EJEMPLO 4.

```
DELIMITER $$
CREATE FUNCTION ver_estado(estado_entrada CHAR(1))
RETURNS VARCHAR(20)
BEGIN
    DECLARE estado VARCHAR(20);

    IF estado_entrada = 'P' THEN
        SET estado = 'caducado';
    ELSEIF estado_entrada = 'O' THEN
        SET estado = 'activo';
    ELSEIF estado_entrada = 'N' THEN
        SET estado = 'nuevo';
    END IF;
    RETURN(estado);
END;$$
DELIMITER ;
```

En este ejemplo la función recibe un valor de estado como entrada y comprueba su valor. Según cual sea se asignará con el comando SET el valor abreviado a la variable estado que es devuelta. [NOTA 1](#)



```
mysql> select ver_estado('p'); $$
+-----+
| ver_estado('p') |
+-----+
| caducado        |
+-----+
1 row in set (0.00 sec)

mysql> select ver_estado('o'); $$
+-----+
| ver_estado('o') |
+-----+
| activo          |
+-----+
1 row in set (0.00 sec)

mysql> select ver_estado('n'); $$
+-----+
| ver_estado('n') |
+-----+
| nuevo           |
+-----+
1 row in set (0.00 sec)
```

EJEMPLO 5.

```

DELIMITER $$
CREATE FUNCTION esimpar(numero int)
    RETURNS int
BEGIN
    DECLARE impar INT;
    IF MOD(numero,2)=0 THEN SET impar=FALSE;
    ELSE SET impar=TRUE;
    END IF;
    RETURN(impar);
END ;$$

```

En este caso recibimos un número como entrada devolviendo TRUE si es par y FALSE en caso de que sea impar.

EJEMPLO 6.

Una función puede ser llamada con su nombre y una lista de parámetros con el tipo de dato apropiado. Veamos el siguiente ejemplo en el que llamamos a una función desde la línea de comandos:

```

mysql> SET @x=impar(42);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT impar(42);

```

```

mysql> set @x=esimpar(42);
Query OK, 0 rows affected (0.00 sec)

mysql> select @x;
+-----+
| @x    |
+-----+
| 0     |
+-----+
1 row in set (0.00 sec)

mysql> select esimpar(42);
+-----+
| esimpar(42) |
+-----+
| 0           |
+-----+
1 row in set (0.00 sec)

```

La función impar devuelve un 0 o un 1 si la variable pasada como parámetro es o no par. Dicho valor se asigna a una variable de sesión @x que podemos mostrar con un SELECT.

Otra opción es usar directamente la función como una expresión en la cláusula SELECT tal como se ve en el siguiente ejemplo.

EJEMPLO 7.

Sin embargo es más usual llamar a las funciones desde otras funciones o procedimientos como en el siguiente ejemplo:

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

Ahora la nueva función nos muestra un mensaje indicando la paridad del parámetro.

De este modo las funciones permiten reducir la complejidad aparente del código encapsulando el código y simplificando por tanto su mantenimiento y legibilidad.

1.2.PARÁMETROS Y VARIABLES.

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

1.2.1 VARIABLES

Encontramos dos nuevas cláusulas para el **manejo de variables**:

DECLARE: crea una nueva variable con su nombre y tipo. Los tipos son los usuales de MySQL como char, varchar, int, float, etc... Esta cláusula puede incluir una opción para indicar valores por defecto. Si no se indica, dichos valores serán NULL.

Por ejemplo: **DECLARE a, b INT DEFAULT 5;**

Crea dos variables enteras con valor 5 por defecto.

SET: permite asignar valores a las variables usando el operador de igualdad.

Por ejemplo: **SET a = 25;**

Alcance de las variables

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

EJEMPLO 8.

```
DELIMITER $$
CREATE PROCEDURE proc5()
BEGIN
  DECLARE x1 CHAR(5) DEFAULT 'fuera';
  BEGIN
    DECLARE x1 CHAR(5) DEFAULT 'dentro';
    SELECT x1;
  END;
  SELECT x1;
END; $$
```

Las variables x1 del primer y segundo bloque Begin / End son distintas, solo tienen validez dentro del bloque como se demuestra en la llamada al procedimiento.

```
mysql> CALL proc5()$$
+-----+
| x1 |
+-----+
| dentro |
+-----+
+-----+
| x1 |
+-----+
| fuera |
+-----+
```


1.2.2 PARÁMETROS

Tipos de parámetros

También observamos la posibilidad de incluir un parámetro. Existen tres tipos:

- **IN:** Es el tipo por defecto y sirve para incluir parámetros de entrada que usará el procedimiento. En este caso no se mantienen las modificaciones.

```
DELIMITER $$
CREATE PROCEDURE proc2(IN p INT) SET @x = p
$$

mysql> CALL proc2(12345)$$
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x$$
+-----+
| @x |
+-----+
| 12345 |
+-----+
1 row in set (0.00 sec)
```

Se establece el valor de una variable de sesión (precedida por @) al valor de entrada p.

- **OUT:** Parámetros de salida. El procedimiento puede asignar valores a dichos parámetros que son devueltos en la llamada.

```
CREATE PROCEDURE proc3(OUT p INT) SET p = -5 $$
mysql> CALL proc3(@y)$$
mysql> SELECT @y$$
+-----+
| @y |
+-----+
| -5 |
+-----+
```

En este caso hemos creado una nueva variable @y al llamar al procedimiento cuyo valor se actualiza dentro del mismo por ser ésta de tipo OUT.

- **INOUT:** Permite pasar valores al procedimiento que serán modificados y devueltos en la llamada.

```
CREATE PROCEDURE proc4(INOUT p INT) SET p = p-5 $$
mysql> SET @y=0$$
mysql> CALL proc4(@y)$$
mysql> SELECT @y$$
+-----+
| @y |
+-----+
| -5 |
+-----+
```

Esta vez usamos el mismo valor del parámetro para disminuir su valor previamente asignado con SET.

EJERCICIO 1.

Crea un procedimiento *proc1* que recibe una variable entera de entrada llamada *parámetro1*. A continuación se declaran sendas variables *variable1* y *variable2* de tipo entero y se testea el valor del *parámetro*, en caso de que sea 17 se asigna su valor a la *variable1* y si no a la *variable2* se le asigna el valor 30.

Posteriormente se introduce una fila en la tabla *t* (previamente creada) rellenando sus dos campos con los valores de las dos variables.

1.3.TIPOS DE INSTRUCCIONES.**1.3.1. INSTRUCCIONES CONDICIONALES**

En muchas ocasiones el valor de una o más variables o parámetros determinará el proceso de las mismas. Cuando esto ocurre debemos usar instrucciones condicionales de tipo simple o IF cuando solamente hay una condición, alternativas o IF THEN ELSE cuando hay dos posibilidades o múltiple, o CASE cuando tenemos un conjunto de condiciones distintas.

IF-THEN-ELSE

La sintaxis general es:

```
IF expr1 THEN
...
ELSEIF expr2 THEN
...
ELSE
...
END IF
```

EJERCICIO 2.

Se debe insertar o actualizar la tabla (con un solo campo *s1*) según el valor de entrada de un procedimiento.

Utilizamos una variable *var1* que se inicializa con el valor del *parámetro* de entrada más 1.

Cuando el valor de la *variable1* es 1 entonces hacemos una inserción del valor de entrada dado. En caso de que sea 0 el *parámetro* de entrada actualizamos sumando 1 al valor actual y si no sumamos 2.

CASE

Cuando hay muchas condiciones es más apropiado el uso de esta instrucción.

Su sintaxis general es:

```
CASE expression
  WHEN value THEN statements
  [WHEN value THEN statements ...]
  [ELSE statements]
END CASE;
```

Donde *expression* es una expresión cuyo valor puede coincidir con uno de los posibles val1, val2, etc. En otro caso se ejecutan las instrucciones seguidas por ELSE.

EJERCICIO 3

Se debe insertar un registro en la tabla con un valor en función del parámetro de entrada. Si el parámetro es 1 se deberá insertar primero, si es 2 segundo y si es 3 tercero. Para cualquier otro valor se insertará otro.

1.3.2. INSTRUCCIONES REPETITIVAS O LOOPS.

Los loops permiten iterar un conjunto de instrucciones un número determinado de veces. Para ello MySQL provee tres tipos de instrucciones:

SIMPLE LOOP

La sintaxis general es:

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

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

El siguiente ejemplo muestra un bucle infinito que no se recomienda probar:

```
Infinite_loop: LOOP
SELECT 'Esto no acaba nunca !';
END LOOP infinite_loop;
```

EJEMPLO 9.

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

```
0. DELIMITER $$
1. CREATE PROCEDURE proc9()
2. BEGIN
3.     DECLARE cont INT;
4.     SET cont = 0;
5.     loop_label: LOOP
6.         INSERT INTO t VALUES (cont);
7.         SET cont = cont + 1;
8.         IF cont >= 5 THEN
9.             LEAVE loop_label;
10.        END IF;
11.    END LOOP;
12. END; $$
```

Como vemos todo el proceso queda delimitado en un bloque `BEGIN/END` el cual incluye un bucle que comienza en la línea 5 y termina en la línea 11. A su vez éste bucle realiza la inserción de una fila con el valor del contador `cont` en la tabla `t` (si estamos en la base de datos `test` deberíamos crearla) en la línea 6, incrementa el valor del contador `cont` en una unidad con `SET` en la línea 7 e incluye una instrucción condicional simple que comprueba el valor del contador `cont` de manera que cuando éste supere el valor 5 se producirá la salida del bucle con la instrucción `LEAVE` de la línea 9. Finalmente termina el `IF`.

REPEAT UNTIL LOOP

La sintaxis general es:

```
[etiqueta:] REPEAT
instrucciones
UNTIL expresion
END REPEAT [etiqueta]
```

EJERCICIO 4.

Se muestran los números impares desde 0 a 10

WHILE LOOP

La sintaxis general es:

```
[etiqueta:] WHILE expression DO
instrucciones
END WHILE [etiqueta]
```

EJERCICIO 5

Se muestran los números impares desde 0 a 10

EJERCICIO 6.

- Crear una tabla con 2 campos: id y datos.
- Insertar en los 10 primeros registros: 1 registro 1.
- Actualizar los cinco últimos registros cambiándolos a “fila actualizada” (con repeat)

En este ejemplo usamos sentencias sql de definición (drop y create) y sentencias sql de manipulación (insert, update).

El resultado debe quedar así:

id	datos
1	registro 1
2	registro 2
3	registro 3
4	registro 4
5	registro 5
6	fila actualizada
7	fila actualizada
8	fila actualizada
9	fila actualizada
10	fila actualizada

1.4.GESTIÓN DE RUTINAS ALMACENADAS.

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

- **Eliminación rutinas:** Para eliminar procedimientos o función usamos el comando SQL DROP con la siguiente sintaxis:

DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name

- **Consulta rutinas:** Podemos ver información más o menos detalladas de nuestras rutinas usando los comandos:

SHOW CREATE {PROCEDURE | FUNCTION} sp_name

SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'patron']

Donde en el segundo caso podemos hacer un filtro por patrones.

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

ACTIVIDAD 1: Ejercicios 1 al 5.

ANEXO:

NOTA 1:

Puede que al crear una función en el Wamp de ERROR 1418 (HY000): This routine has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_function_creators variable)

Debemos activar la variable global log_bin_trust_function_creators que por defecto está desactivada.

Mysql> show variables like 'log%';

Mysql> set GLOBAL log_bin_trust_function_creators = ON;