Procedimientos almacenados

Es un segmento de código declarativo SQL, que se almacena en la base de datos. Un procedimiento almacenado puede ser invocado por un programa, un disparador o incluso otro procedimiento almacenado.

Los procedimientos almacenados ayudan a aumentar el rendimiento de la aplicación al usarlos. Una vez creados, se compilan y almacenan en la base de datos. Un procedimiento normalmente se ejecuta más rápido que un conjunto de sentencias sin compilar.

Los procedimientos almacenados pueden ser reutilizados por otras aplicaciones.

Los procedimientos almacenados están asegurados ya que el administrador de la base de datos debe dar el permiso correspondiente para poder tener acceso a ellos.

Estructura de un procedimiento almacenado

```
DELIMITER //
CREATE PROCEDURE mostrarproductos()
  BEGIN
  SELECT *  FROM productos;
  END //
DELIMITER ;
```

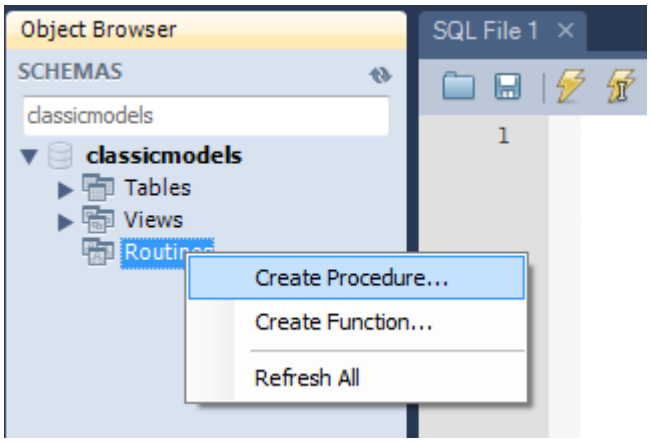
El primer comando es DELIMITER / /. Esta orden no está relacionada con la sintaxis del procedimiento almacenado. La declaración DELIMITER cambia el delimitador estándar que es el punto y coma por otro cualquiera. Tenemos que cambiar el delimitador porque queremos pasar el procedimiento almacenado al servidor como un todo en lugar

de dejar que Mysql interprete cada instrucción en el momento en que se escribe.

Después de la palabra clave END se coloca el delimitador / / para indicar el final del procedimiento almacenado y la última orden (DELIMITER ;) vuelve a cambiar el delimitador para que éste sea el punto y coma.

En este caso, el nombre del procedimiento almacenado es mostrarproductos y no se pueden olvidar los paréntesis después del nombre del mismo que se usan para los parámetros.

Las palabras claves BEGIN y END encierran el cuerpo del procedimiento almacenado.



Para invocar un procedimiento almacenado:

```
CALL NOMBRE_PROCEDIMIENTO();
```

```
CALL Mostrarproductos();
```

Variables en procedimientos almacenados

Declaración de variables

```
DECLARE nomvariable tipodedato(tamaño) DEFAULT valor por defecto;
```

El tipo de datos de la variable puede ser cualquier tipo primitivo que MySQL

soporte como INT, VARCHAR y DATETIME. Cuando se declara una variable su valor inicial es NULL y se puede asignar un valor predeterminado con default. La sección de declaraciones se hace dentro del bloque BEGIN END.

```
DECLARE total INT DEFAULT 0;

DECLARE x,y INT(4) DEFAULT 0;
```

Asignación de valores a las variables

```
SET nombre_var = expr [, nombre_var = expr] …;

SET A=1, B=2;
```

Además de la instrucción SET, se utiliza SELECT INTO para asignar el resultado de una consulta a una variable.

```
DECLARE total_productos INT DEFAULT 0;

SELECT COUNT (*) INTO total_productos

FROM productos;
```

Para mostrar el valor de una variable, se usa SELECT nombre_var.

Alcance y Tipos de variables

Una variable tiene su propio ámbito. Si se declara una variable dentro de un procedimiento almacenado, estará fuera de alcance cuando finalice el mismo. Se puede declarar dos o más variables con el mismo nombre en diferentes ámbitos porque una variable sólo es eficaz en su propio ámbito.

MySql proporciona acceso a muchas variables de sistema y de conexión. Muchas variables pueden modificarse dinámicamente mientras el servidor se está ejecutando, ésto permite variar la operación del servidor sin tener que detenerlo y reiniciarlo.

El servidor MySql tiene dos clases de variables. Las variables **globales** que afectan

la operación general del servidor y las variables de **Sesión o Locales** actúan sobre la operación en conexiones de clientes individuales. Para poder ver los valores de todas las variables globales y de sesión:

```
SHOW GLOBAL VARIABLES;

SHOW SESSION VARIABLES;      – O también omitiéndo la palabra SESSION
```

Cuando el servidor arranca, inicializa todas las variables globales a sus valores predeterminados. Estos valores pueden ser modificados por opciones especificadas en ficheros de opciones o en la línea de comandos. Después de que el servidor se inicie, las variables globales pueden ser modificadas con la sentencia SET GLOBAL @@ nombre_var pero solamente cuando se tiene el privilegio SUPER.

Para recuperar el valor de una variable GLOBAL debe utilizarse una de las siguientes sentencias:

```
SELECT @@global.sort_buffer_size;

SHOW GLOBAL VARIABLES like 'sort_buffer_size';
```

El servidor también permite un conjunto de variables de sesión para cada cliente que se conecta. Las variables de sesión de cliente se inicializan en el momento de conectarse, empleando el valor actual de la correspondiente variable global. Las variables de sesión dinámicas pueden ser modificadas por el cliente mediante una sentencia SET SESSION @@nombre_var. No se requieren privilegios especiales para establecer el valor una variable de sesión, pero un cliente puede modificar solamente sus propias variables, no las de otros clientes.

```
SET GLOBAL sort_buffer_size=valor;

SET @@global.sort_buffer_size=valor;
```

Para establecer el valor de una variable SESSION, debe emplearse una de las siguientes sintaxis:

```
SET SESSION sort_buffer_size=valor;

SET @@session.sort_buffer_size=valor;

SET sort_buffer_size=valor;
```

Si al establecer el valor de una variable no se utiliza GLOBAL, SESSION o LOCAL por defecto se asume SESSION.

Para recuperar el valor de una variable SESSION debe utilizarse una de las siguientes sentencias:

```
SELECT @@sort_buffer_size;

SELECT @@session.sort_buffer_size;

SHOW SESSION VARIABLES like 'sort_buffer_size';
```

Cuando se recupera una variable con SELECT @@nombre_var sin especificar si es global., session., o local. MySQL devuelve el valor de SESSION si existe y el valor GLOBAL en otro caso.

Por otro lado, un usuario puede definir variables propias de su sesión anteponiéndoles el símbolo "@" al principio. El ámbito de estas variables es la sesión del usuario. No se declaran, sólo se les asigna un valor con set y ya pueden usarse. La asignación puede hacerse bien en un procedimiento, función o en la línea de comandos.

```
set @h="1998-02-09";        – crea la variable de sesión del usuario h y le asigna un valor
select @h;                  – muestra el contenido de esta variable de sesión
```

```
DELIMITER $$
CREATE  PROCEDURE variables_de_sesión()
BEGIN
        set @var="23";      – Una vez ejecutado el procedimiento, la variable existe durante
                            – toda la sesión del usuario
        select @var;
end $$
DELIMITER ;
```

Estructuras de control en Procedimientos Almacenados

Instrucción IF

```
IF condición/es THEN instrucción/es
   [ELSEIF condición/es THEN instrucción/es]
   [ELSE instrucción/es]
END IF;
```

Instrucción CASE

```
CASE  expresión
   WHEN valor1 THEN instrucción/es
   WHEN valor2 THEN instrucción/es
   ...
   ELSE instrucción/es
END CASE;
```

Estructuras repetitivas en procedimientos almacenados

Bucle WHILE

```
WHILE expresión DO
   Instrucción/es;
END WHILE;
```

Bucle REPEAT

```
REPEAT
   Instrucción/es;
UNTIL expresión              – IMPORTANTE NO LLEVA ;
END REPEAT;
```

Bucle LOOP más instrucciones LEAVE e ITERATE

La sentencia LEAVE permite salir del bucle, es similar a la sentencia BREAK de Java y se debe usar lo menos posible.

La declaración ITERATE permite saltarse el código completo debajo de ella y comenzar una nueva iteración. La declaración ITERATE es como el CONTINUE en Java.

```
Etiqueta: LOOP
     Instrucciones;   /* LEAVE etiqueta;  y ITERATE etiqueta; */
END LOOP;
```

Este procedimiento almacenado construye una cadena con los números impares comprendidos entre 1 y 100.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS LoopProc$$    /* BORRADO DE PROCEDIMIENTO */
CREATE PROCEDURE LoopProc()
       BEGIN
          DECLARE x INT;
          DECLARE str VARCHAR(255);
          SET x=1;
          SET str='';
          loop_label:  LOOP
               IF  x > 100 THEN
                 LEAVE  loop_label;
               END  IF;
```

```
            SET  x = x + 1;
            IF  (x mod 2) THEN     /* FALSE=0 TRUE<>0 */
                ITERATE  loop_label;
            ELSE
                SET  str = CONCAT(str,x,',');
            END  IF;
        END LOOP;
        SELECT str;
    END$$
DELIMITER ;
```

Procedimientos almacenados con Parámetros

Casi todos los procedimientos almacenados que se desarrollan requieren parámetros que hacen el procedimiento almacenado más flexible y útil. En MySQL, un parámetro puede ser de uno de estos tres tipos IN, OUT e INOUT.

- IN es el modo por defecto, indica que el parámetro es de entrada y no puede ser modificado por lo que siempre estará a la derecha del igual en las asignaciones.

- OUT este modo indica que el parámetro es de salida por lo que dentro del procedimiento almacenado siempre estará a la izquierda en las asignaciones.

- INOUT obviamente este modo se combina los modos de entrada y salida.

La sintaxis de definición de un parámetro en el procedimiento almacenado es la siguiente:

MODO nombre_param tipo_param (tamaño_param)

El modo puede ser IN, OUT o INOUT en función de la finalidad del parámetro especificado. Los parámetros se separan con comas si el procedimiento almacenado tiene más de un parámetro.

Este primer ejemplo es un procedimiento que obtiene todas las oficinas de una determinada región.

```
DELIMITER //
 CREATE PROCEDURE mostraroficinas(IN p_region VARCHAR(6))
    BEGIN
       SELECT oficina, ciudad
       FROM oficinas
       WHERE region = p_region;
    END //
 DELIMITER ;

CALL mostraroficinas('NORTE');
```

Este procedimiento almacenado nos devuelve cuántos pedidos ha hecho un determinado cliente.

```
DELIMITER $$
CREATE PROCEDURE pedidosporcliente(
       IN p_cliente INT(4),
       OUT total INT)
    BEGIN
       SELECT count(*)
       INTO total
       FROM pedidos
       WHERE cliente=p_cliente;
    END$$
 DELIMITER ;
```

Para ejecutar este procedimiento sería:

```
CALL pedidosporcliente( 1235, @total);
SELECT @total;
```

```
DELIMITER $$
CREATE PROCEDURE GetCustomerShipping(
        in  p_customerNumber int(11),
        out p_shiping        varchar(50))
BEGIN
    DECLARE customerCountry varchar(50);

    SELECT country INTO customerCountry
    FROM customers
    WHERE customerNumber = p_customerNumber;

    CASE customerCountry
        WHEN  'USA' THEN
           SET p_shiping = '2-day Shipping';
        WHEN 'Canada' THEN
           SET p_shiping = '3-day Shipping';
```

```
        ELSE
            SET p_shiping = '5-day Shipping';
    END CASE;
END$$
DELIMITER ;


SET @customerNo = 112;
SELECT country into @country
FROM customers
WHERE customernumber = @customerNo;

CALL GetCustomerShipping(@customerNo,@shipping);

SELECT @customerNo AS Customer,
       @country   AS Country,
       @shipping  AS Shipping;
```

Customer	Country	Shipping
112	USA	2-day Shipping

```
DELIMITER $$
DROP PROCEDURE IF EXISTS WhileLoopProc$$
CREATE PROCEDURE WhileLoopProc()
BEGIN
      DECLARE x  INT;
      DECLARE str  VARCHAR(255);
      SET x = 1;
      SET str =  '';
      WHILE x  <= 5 DO
         SET  str = CONCAT(str,x,',');
         SET  x = x + 1;
      END WHILE;
      SELECT str;
END$$
DELIMITER ;


DELIMITER $$
DROP PROCEDURE IF EXISTS RepeatLoopProc$$
CREATE PROCEDURE RepeatLoopProc()
BEGIN
       DECLARE x  INT;
       DECLARE str  VARCHAR(255);
       SET x = 1;
       SET str = '';
       REPEAT
```

```
            SET  str = CONCAT(str,x,',');
            SET  x = x + 1;
        UNTIL x  > 5
        END REPEAT;
        SELECT str;
END$$
DELIMITER ;

DELIMITER $$
DROP PROCEDURE IF EXISTS LOOPLoopProc$$
CREATE PROCEDURE LOOPLoopProc()
BEGIN
     DECLARE x  INT;
     DECLARE str  VARCHAR(255);
     SET x = 1;
     SET str =  '';
     loop_label:  LOOP
                  IF  x > 10 THEN
                              LEAVE  loop_label;
                  END  IF;
                  SET  x = x + 1;
                  IF  (x mod 2) THEN
                        ITERATE  loop_label;
                  ELSE
                        SET  str = CONCAT(str,x,',');
                  END  IF;

                END LOOP;
                SELECT str;
END$$
DELIMITER ;
```

Cursores SQL en Procedimientos Almacenados

MySQL soporta cursores en procedimientos almacenados, funciones y triggers. El cursor se utiliza para poder recuperar las filas que devuelve una consulta y **procesarlas**. Si únicamente se quieren visualizar las filas, con la orden Select basta. En Oracle no es así, ya que se necesita un cursor siempre que la Select devuelva más de una fila aunque éstas no vayan a ser procesadas y sólo se vayan a mostrar.

Tener en cuenta que:

- Es de sólo lectura es decir no se puede actualizar el cursor.
- Únicamente puede recorrerse en una dirección y nunca hacia atrás.
- Debe evitarse mientras una tabla se está actualizando abrir un cursor de la misma porque pueden obtenerse resultados inesperados.

```
DECLARE nombre_cursor CURSOR FOR select....;
```

Una vez declarado, hay que abrir el cursor mediante la sentencia OPEN.

```
OPEN nombre_cursor;
```

A continuación, se pueden recuperar una a una las filas del cursor mediante la sentencia FETCH.

```
FETCH nombre_cursor INTO lista de variables;
```

Y, por último, debe cerrarse el cursor para desactivarlo y liberar la memoria asociada a ese cursor.

```
CLOSE nombre_cursor;
```

Si no existen más registros disponibles, ocurrirá una condición de Sin Datos con el valor SQLSTATE 02000. Entonces se configura un manejador (handler) para detectar esta condición del fin de cursor (en Oracle se controla con NOT FOUND).

Los cursores deben declararse antes de declarar los handlers y las variables deben declararse antes de declarar cursores y handlers.

```
DELIMITER \\
CREATE PROCEDURE curdemo()
BEGIN
        DECLARE done INT DEFAULT 0;
        DECLARE a CHAR(16);
        DECLARE b,c INT;
        DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
        DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
        DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
```

```
        OPEN cur1;
        OPEN cur2;
        REPEAT
          FETCH cur1 INTO a, b;
          FETCH cur2 INTO c;
          IF NOT done THEN
             IF b < c THEN
                  INSERT INTO test.t3 VALUES (a,b);
             ELSE
                  INSERT INTO test.t3 VALUES (a,c);
             END IF;
           END IF;
         UNTIL done         /* done=1 => true   done=0 => false  */
         END REPEAT;
         CLOSE cur1;
         CLOSE cur2;
END\\
DELIMITER ;



DELIMITER ;;
DROP PROCEDURE IF EXISTS `micursor`;;
CREATE PROCEDURE `micursor`()
BEGIN

      DECLARE done BOOLEAN DEFAULT FALSE;
      DECLARE uid integer;
      DECLARE newdate date;

      DECLARE c1 cursor for SELECT id,timestamp from employers ORDER BY id ASC;
      DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = TRUE;

      open c1;
      c1_loop: LOOP
             fetch c1 into uid,newdate;
             IF `done` THEN LEAVE c1_loop; END IF;
             UPDATE calendar SET timestamp = newdate  WHERE id=uid;
      END LOOP c1_loop;
      CLOSE c1;
END ;;
DELIMITER ;



DELIMITER $$
DROP PROCEDURE IF EXISTS CursorProc$$
CREATE PROCEDURE CursorProc()
BEGIN
       DECLARE  no_more_products, quantity_in_stock INT DEFAULT 0;
       DECLARE  prd_code VARCHAR(6);
       DECLARE  cur_products CURSOR FOR
                SELECT  idproducto FROM products;
       DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
                SET no_more_products = TRUE;
```

```
        OPEN  cur_products;
        FETCH  cur_products INTO prd_code; /*considerar dónde poner los
                                          fetch*/
        IF  no_more_products THEN SELECT “LISTADO VACÍO”;
        ELSE
              REPEAT
                  SELECT  quantityInStock INTO quantity_in_stock
                  FROM products WHERE  productCode = prd_code;
                  IF  quantity_in_stock < 100 THEN
                       INSERT  INTO infologs(msg) VALUES (prd_code);
                  END  IF;
                  FETCH  cur_products INTO prd_code;
              UNTIL  no_more_products = 1
              END REPEAT;
        END IF;
        CLOSE  cur_products;
      SELECT *  FROM infologs;
      DROP TABLE  infologs;
END$$
DELIMITER;
```

En este código se ha usado la orden DROP. MySQL permite que los procedimientos y funciones contengan comandos DDL (tales como `CREATE` y `DROP`) Ésto no lo requiere el estándar, de hecho Oracle no lo permite.

Este comando muestra todos los procedimientos de la base de datos pudiendo filtrar también el resultado.

```
SHOW PROCEDURE STATUS [LIKE 'patrón' | WHERE expr]

SHOW PROCEDURE STATUS

SHOW PROCEDURE STATUS WHERE db = 'classicmodels'

SHOW PROCEDURE STATUS WHERE name LIKE '%product%'
```

Para ver el contenido del procedimiento.

```
SHOW CREATE PROCEDURE nombre_procedimiento

SHOW CREATE PROCEDURE Mostrar_productos
```

Funciones

Las funciones se diferencian de los procedimientos en que devuelven un valor.

```
DELIMITER //
CREATE FUNCTION HOLA(S CHAR(20)) RETURNS CHAR(50)
BEGIN
RETURN CONCAT('hola, ',s,'!');
END //
```

Podemos omitir la longitud en el RETURN y sólo indicar el tipo.

Para ejecutar la función anterior: SELECT HOLA('PEPE');

Si la función no tuviera parámetros: SELECT HOLA();

Los comandos DROP y ALTER de los procedimientos se aplican también a las funciones de la misma forma pero sustituyendo la palabra clave PROCEDURE por FUNCTION.

Las funciones pueden contener instrucciones SELECT INTO y también cursores.