FUNCIONES

Una de las grandes novedades a partir de la versión 5 de MySQL es el soporte para procesos almacenados. A continuación, vemos los fundamentos teóricos y este tema más algunos ejemplos básicos.

Si hemos usado bases de datos como Oracle, Interbase / Firebird, PostgreSQL, escuchamos hablar de procedimientos almacenados. Sin embargo, en MySQL esto es toda una novedad y un paso enorme para que esta base de datos se convierta en un verdadero sistema gestor de bases de datos.

¿QUÉ ES UN PROCEDIMIENTO ALMACENADO?

Los procedimientos almacenados son un conjunto de instrucciones SQL más una serie de estructuras de control que proveen de cierta lógica al procedimiento. Estos procedimientos están guardados en el servidor y son accedidos a través de llamadas.

Para crear un procedimiento, MySQL ofrece la directiva CREATE PROCEDURE. Al crearlo, éste, es asociado con la base de datos en uso, tal como cuando creamos una tabla.

Para llamar a un procedimiento lo hacemos mediante la instrucción CALL. Desde un procedimiento invocamos a su vez a otros procedimientos o funciones.

Un procedimiento almacenado, al igual cualquiera de los procedimientos que programamos en nuestras aplicaciones utilizando cualquier lenguaje, tiene:

- Un nombre.
- Puede tener una lista de parámetros.
- Tiene un contenido (sección también llamada definición del procedimiento: aquí se especifica qué es lo que va a hacer y cómo).
- Ese contenido puede estar compuesto por instrucciones sql, estructuras de control, declaración de variables locales, control de errores, etcétera.

MySQL sigue la sintaxis SQL:2003 para procedimientos almacenados, que

también usa IBM DB2.

SINTAXIS

En resumen, la sintaxis de un procedimiento almacenado es la siguiente:

```
CREATE PROCEDURE nombre (parámetro)
    [características] definición
```

Puede haber más de un parámetro (se separan con comas) o puede no haber ninguno (en este caso deben seguir presentes los paréntesis, aunque no haya nada dentro).

Los parámetros tienen la siguiente estructura: modo nombre tipo. Donde:

- modo: es opcional y puede ser IN (el valor por defecto, son los parámetros que el procedimiento recibirá), OUT (son los parámetros que el procedimiento podrá modificar) INOUT (mezcla de los dos anteriores).
- nombre: es el nombre del parámetro.
- tipo: es cualquier tipo de dato de los provistos por MySQL.

Dentro de características es posible incluir comentarios o definir si el procedimiento obtendrá los mismos resultados ante entradas iguales, entre otras cosas.

- definición: es el cuerpo del procedimiento y está compuesto por el procedimiento en sí: aquí se define qué hace, cómo lo hace y bajo qué circunstancias lo hace.

Así como existen los procedimientos, también existen las funciones. Para crear una función, MySQL ofrece la directiva CREATE FUNCTION.

La diferencia entre una función y un procedimiento es que la función devuelve valores. Estos valores pueden ser utilizados como argumentos para instrucciones SQL, tal como lo hacemos con otras funciones como: MAX() o COUNT().

Utilizar la cláusula RETURNS es obligatorio al momento de definir una función y sirve para especificar el tipo de dato devuelto (sólo el tipo de dato, no el dato).

Su sintaxis es:

```
CREATE FUNCTION nombre (parámetro)
RETURNS tipo
[características] definición
```

Puede haber más de un parámetro (separados con comas) o ninguno (en este caso deben seguir presentes los paréntesis, aunque no haya nada dentro). Los parámetros tienen la siguiente estructura:nombre tipo.

Donde:

- nombre: es el nombre del parámetro.
- tipo: es cualquier tipo de dato de los provistos por MySQL.

Dentro de características es posible incluir comentarios o definir si la función devolverá los mismos resultados ante entradas iguales, entre otras cosas.

- definición: es el cuerpo del procedimiento y está compuesto por el procedimiento en sí: aquí se define qué hace, cómo lo hace y cuándo lo hace.

Para llamar a una función lo hacemos invocando su nombre, como se hace en muchos lenguajes de programación.

Desde una función podemos invocar a su vez a otras funciones o procedimientos.

```
delimiter //
CREATE PROCEDURE procedimiento (IN cod INT)
BEGIN
SELECT * FROM tabla WHERE cod_t = cod;
END
//
Query OK, 0 rows affected (0.00 sec)
delimiter ;
```

```
CALL procedimento(4);
```

En el código anterior lo primero que hacemos es fijar un delimitador. Al utilizar la línea de comandos de MySQL vemos que el delimitador por defecto es el punto y coma (;): en los procedimientos almacenados lo definimos nosotros.

Lo interesante de esto es que escribimos el delimitador anterior; sin que el procedimiento termine. Más adelante, en este mismo código volveremos al delimitador clásico. Luego, creamos el procedimiento con la sintaxis vista anteriormente y ubicamos el contenido entre las palabras reservadas BEGIN y END.

El procedimiento recibe un parámetro para trabajar con él, por eso ese parámetro es de tipo IN. Definimos el parámetro como OUT cuando en él sale del procedimiento. Si el parámetro hubiese sido de entrada y salida a la vez, sería de tipo denominado INOUT.

El procedimiento termina y es llamado luego mediante la siguiente instrucción:

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

Otro ejemplo:

```
CREATE PROCEDURE procedimiento2 (IN a INTEGER)
BEGIN
DECLARE variable CHAR(20);
IF a > 10 THEN
SET variable = 'mayor a 10';
ELSE
SET variable = 'menor o igual a 10';
END IF;
INSERT INTO tabla VALUES (variable);
END
```

- El procedimiento recibe un parámetro llamado a que es de tipo entero.
- Se declara una variable para uso interno que se llama variable y es de tipo char.
- Se implementa una estructura de control y si a es mayor a 10 se asigna a variable un valor. Si no lo es se le asigna otro.
- Se utiliza el valor final de variable en una instrucción SQL.

Recordemos que para implementar el ultimo ejemplo se deben usar nuevos delimitadores, como se vio anteriormente.

Observemos ahora algunas aplicaciones de funciones:

```
mysql> delimiter //
mysql> CREATE FUNCTION cuadrado (s SMALLINT) RETURNS SMALLINT
RETURN s*s;
//
Query OK, 0 rows affected (0.00 sec)
mysql> delimiter ;
mysql> SELECT cuadrado(2);
```

Otras bases de datos como PostgreSQL implementan procedimientos almacenados y brindan la posibilidad de programarlos utilizando en lenguajes como PHP, C, PGPLSQL o Java.

En MySQL hay intenciones de implementar lo mismo y seguramente en las próximas versiones lo veremos, pero más importante que utilizar un lenguaje u otro es entender para qué podrían servirnos los procedimientos almacenados.

En definitiva hemos dado un recorrido por el mundo de la programación de procedimientos almacenados en MySQL. Es importante que se trata de un mundo que está en pleno desarrollo y que promete evolucionar.

Los procedimientos almacenados son un conjunto de instrucciones en SQL que permiten realizar una tarea determinada. Estos procedimientos se guardan en el servidor y se accede a ellos llamándolos por el nombre dado al momento de crearlos.

Los procedimientos almacenados tienen:

- un nombre
- (pueden tener) una lista de parámetros
- un contenido (lo que hace)

Procedimientos almacenados vs Funciones

A simple vista, parecen ser lo mismo que las funciones, ya que ambos permiten:

- Reusar código
- Esconder detalles de SQL
- Centralizar el mantenimiento

A pesar de eso, existen diferencias entre estas dos estructuras:

- Los procedimientos almacenados se llaman independientemente, mientras que las funciones son llamadas dentro de otra sentencia SQL
- Es posible conceder permiso a los usuarios a un procedimiento almacenado específico, en lugar de permitirle acceder a las tablas.
- Las funciones siempre deben devolver un valor. Los procedimientos pueden retornar un valor escalar, un valor de tabla o simplemente nada.

Procedimientos almacenados y lenguajes de programación

Al momento de acceder a los datos (desde la aplicación), es muy importante utilizar los procedimientos almacenados, ya que esto trae muchos beneficios:

- Reduce la cantidad de información enviada al servidor de bases de datos, ya que no generamos la consulta del lado de la aplicación
- Sólo se requiere compilar una vez (cuando es creado).

- Facilita el reutilizado de código, ya que no hay que escribir la misma consulta en distintos lugares
- Mejora la seguridad

Mostramos cómo crear y usar procedimientos almacenados tanto en SQL Server como en MySQL. Aunque el lenguaje de ambos es SQL, existen varias diferencias como para remarcarlas.

En este ejemplo vamos a:

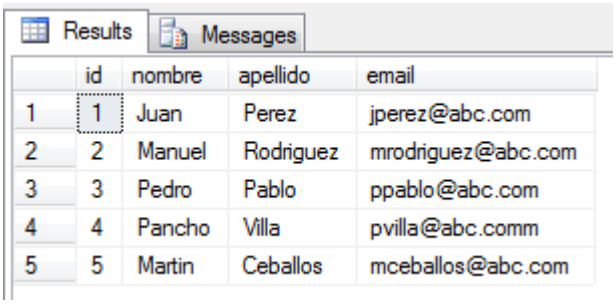
1. crear una tabla llamada estudiantes
2. guardar 5 estudiantes en la tabla
3. crear un procedimiento almacenado que sólo tenga un parámetro de entrada
4. crear un procedimiento almacenado que tenga un parámetro de entrada y otro de salida

Crear y utilizar procedimientos almacenados en MySQL

```
-- creamos la tabla
CREATE TABLE  estudiantes (
    id INT IDENTITY(1,1) NOT NULL,
    nombre VARCHAR(200) NOT  NULL,
    apellido VARCHAR(200)  NULL,
    email VARCHAR(100)  NULL
)


-- agregamos valores
INSERT INTO estudiantes (nombre, apellido, email)
VALUES('Juan', 'Perez', 'jperez@abc.com');
INSERT INTO estudiantes (nombre, apellido, email)
VALUES('Manuel', 'Rodriguez', 'mrodriguez@abc.com');
```

```
INSERT INTO estudiantes (nombre, apellido, email) VALUES('Pedro', 'Pablo', 'ppablo@abc.com');

INSERT INTO estudiantes (nombre, apellido, email) VALUES('Pancho', 'Villa', 'pvilla@abc.comm');

INSERT INTO estudiantes (nombre, apellido, email) VALUES('Martin', 'Ceballos', 'mceballos@abc.com');
```

La tabla queda de la siguiente forma:



Results | Messages

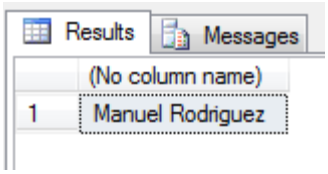
	id	nombre	apellido	email
1	1	Juan	Perez	jperez@abc.com
2	2	Manuel	Rodriguez	mrodriguez@abc.com
3	3	Pedro	Pablo	ppablo@abc.com
4	4	Pancho	Villa	pvilla@abc.comm
5	5	Martin	Ceballos	mceballos@abc.com

Creamos el procedimiento almacenado.

```
CREATE PROCEDURE ObtenerNombreApellido(@idAlumno INT)
AS BEGIN
    SELECT nombre + ' ' + apellido
    FROM estudiantes
    WHERE id=@idAlumno
END
```

Ahora que ya está creado el procedimiento almacenado, podemos llamarlo con la palabra EXECUTE.

```
EXECUTE ObtenerNombreApellido 2
```



Results | Messages

	(No column name)
1	Manuel Rodriguez

Seguimos con el otro procedimiento, para que nos devuelva el nombre y apellido pero en una variable de salida.

```
CREATE PROCEDURE ObtenerNombreApellidoSalida(@idAlumno INT,
    @nombreApellido VARCHAR(200) OUT)
AS
BEGIN
    SELECT @nombreApellido = nombre + ' ' + apellido
    FROM estudiantes
    WHERE id=@idAlumno
END
```

Para llamar a este procedimiento tenemos que hacer unos pasos más. Primero, creamos la variable que contiene el resultado, llamamos al procedimiento y luego mostramos la variable.

```
DECLARE @nombre VARCHAR(200)
EXECUTE ObtenerNombreApellidoSalida 2, @nombre OUTPUT
SELECT @nombre
```

El resultado es el mismo que obtuvimos en el caso anterior.

Crear y utilizar procedimientos almacenados en MySQL

La creación de la tabla en MySQL es igual, la inserción es idéntica.

```
CREATE TABLE  estudiantes (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(200) NOT  NULL,
    apellido VARCHAR(200) NULL,
    email VARCHAR(100) NULL
)
```

-- agregamos valores

```
INSERT INTO estudiantes (nombre, apellido, email)
```

```
VALUES('Juan', 'Perez', 'jperez@abc.com');
INSERT INTO estudiantes (nombre, apellido, email)
VALUES('Manuel', 'Rodriguez', 'mrodriguez@abc.com');
INSERT INTO estudiantes (nombre, apellido, email)
VALUES('Pedro', 'Pablo', 'ppablo@abc.com');
INSERT INTO estudiantes (nombre, apellido, email)
VALUES('Pancho', 'Villa', 'pvilla@abc.comm');
INSERT INTO estudiantes (nombre, apellido, email)
VALUES('Martin', 'Ceballos', 'mceballos@abc.com');
```

La creación del procedimiento almacenado requiere un poco más de explicación.

El carácter ";" se utiliza para terminar una instrucción. Cuando se necesita escribir varias sentencias que forman parte de un mismo bloque (ej: funciones, procedimientos almacenados, triggers), se utiliza un delimitador definido por el usuario. En este caso, el delimitador "//" indica el final del procedimiento. Cabe aclarar que el delimitador "//" que estamos declarando, es totalmente arbitrario, podría ser "$$" o cualquier otro que ayude a distinguir que es un delimitador propio.

```
DELIMITER //
CREATE PROCEDURE ObtenerNombreApellido(idAlumno INT)
   BEGIN
      SELECT CONCAT(nombre, ' ', apellido)
      FROM estudiantes
      WHERE id = idAlumno;
   -- finaliza el procedimiento
   END
//
```

-- se reinicia el delimitador para que vuelva al original

```
DELIMITER;
```

Para llamar al procedimiento, se utiliza la palabra `CALL`

```
CALL ObtenerNombreApellido(2);
```

Ahora, creamos el segundo ejemplo. Al crear un procedimiento con un parámetro de salida, necesitamos pasar el resultado del `SELECT` a la variable con la sentencia INTO.

```
DELIMITER //
CREATE PROCEDURE ObtenerNombreApellidoSalida(
 idAlumno INT,
    OUT nombreApellido VARCHAR(200))
BEGIN
  SELECT CONCAT(nombre, ' ', apellido)
  INTO nombreApellido
  FROM estudiantes
  WHERE id = idAlumno;
END
//
DELIMITER ;
```

Al llamar a este nuevo procedimiento, no es necesario crear la variable @nombre.

```
CALL ObtenerNombreApellidoSalida(2, @nombreApellido);
SELECT @nombreApellido
```

Resumen

Los procedimientos almacenados son una herramienta realmente potente. Su uso es necesario para la seguridad y mejor funcionamiento de nuestros sistemas, permiten rehusar código y optimizan el desempeño de la base de datos reduciendo el tráfico de la red.

Tenemos nuestras funciones y procedimientos en MySQL en lugar de procesar

los datos con algún lenguaje del lado del servidor, como PHP, Además, tiene la ventaja de que transita menos información de la base de datos al servidor web, con el consiguiente aumento del rendimiento y que estas funciones harán que **atacamos la base de datos desde cualquier otro lenguaje, como Java o ASP.NET sin tener que volver a procesar los datos otra vez**.

MySQL tiene muchas funciones que usaremos en nuestros procedimientos almacenados y consultas, pero en ocasiones necesitamos crear **nuestras propias funciones** para hacer cosas especializadas.

Creando funciones en MySQL:

```
DELIMITER //
CREATE FUCNTION holaMundo() RETURNS VARCHAR(20)
BEGIN
    RETURN 'HolaMundo';
END
//
```

Para comprobar que esta operativa escribimos en la consola de MySQL :

```
Select holaMundo();
```

Lo que devuelve el siguiente resultado :

```
mysql> select holaMundo()//

+--------------+
| holaMundo()  |
+--------------+
| Hola Mundo!! |
+--------------+
1 row in set (0.00 sec)
```

Para **borrar la función** que acabamos de crear :

DROP FUNCTION IF EXISTS holaMundo

USO DE LAS VARIABLES EN FUNCIONES

Las variables en las funciones se usan de igual manera que en los procedimientos almacenados, se declaran con la sentencia **DECLARE**, y se asignan valores con la sentencia **SET**.

```
DELIMITER //
CREATE FUNCTION holaMundo() RETURNS VARCHAR(30)
BEGIN
  DECLARE salida VARCHAR(30) DEFAULT 'Hola mundo';
  SET salida = 'Hola mundo con variables';
  RETURN salida;
END
//
```

Esta variable es **local**, y será destruida una vez finalice la función. Cabe destacar el uso de la sentencia DEFAULT en conjunto con DECLARE, que asigna un valor por defecto al declarar la variable.

Uso de parámetros en funciones:

```
DROP FUNCTION IF EXISTS holaMundo
CREATE FUNCTION holaMundo(entrada VARCHAR(20)) RETURNS VARCHAR(20)
BEGIN
  DECLARE salida VARCHAR(20);
  SET salida = entrada;
  RETURN salida;
END
```

Ahora, creamos una función que devuelve el mismo valor que le pasamos como parámetro.

Si escribimos :

```
mysql> select holaMundo("nosolocodigo")//
```

```
+-------------------------+
| holaMundo("nosolocodigo") |
+-------------------------+
| nosolocodigo            |
+-------------------------+
```

1 row in set (0.00 sec)

Obtenemos como resultado lo mismo que hemos pasado como parámetro, en este caso “nosolocodigo”

Para finalizar, algo un poco más complejo, creamos una **función que acepte un dividendo y un divisor y haga una división sin usar el operador división:**

```
create function divide(dividendo int,divisor int) returns int
begin
    declare aux int;
    declare contador int;
    declare resto int;
    set contador = 0;
    set aux = 0;
    while (aux + divisor) <= dividendo do
        set aux = aux + divisor ;
        set contador = contador + 1;
    end while;
    set resto = dividendo - aux ;
return contador;
end;
```

//

Para usarlo, simplemente **llamamos a la función así:**

SELECT divide(20,2)

Lo que devuelve 10.

PROCEDIMIENTOS

Este procedimiento almacenado (PS) esta codificado con un lenguaje propio de cada Gestor de BD y está compilado, por lo que la velocidad de ejecución es rápida.

Principales Ventajas :

•**Seguridad:**Cuando llamamos a un procedimiento almacenado, este deberá realizar todas las comprobaciones pertinentes de seguridad y seleccionará la información lo más precisamente posible, para enviar de vuelta la información justa y necesaria y que por la red corra el mínimo de información, consiguiendo así un aumento del rendimiento de la red considerable.

•**Rendimiento:** el SGBD, en este caso MySQL, es capaz de trabajar más rápido con los datos que cualquier lenguaje del lado del servidor, y llevará a cabo las tareas con más eficiencia. **Solo realizamos una conexión** al servidor y este ya es capaz de realizar todas las comprobaciones sin tener que volver a establecer una conexión. Esto es muy importante, una vez leí que cada conexión con la BD puede tardar hasta medios segundo, imagínate en un ambiente de producción con muchas visitas como puede perjudicar esto a nuestra aplicación… Otra ventaja es la posibilidad de **separar la carga del servidor**, ya que si disponemos de un servidor de base de datos externo estaremos descargando al servidor web de la carga de procesamiento de los datos.

•**Reutilización:** el procedimiento almacenado podrá ser invocado desde cualquier parte del programa, y no tendremos que volver a armar la consulta a la BD cada que vez que queramos obtener unos datos.

Desventajas:

El programa se guarda en la BD, por lo tanto si se corrompe y perdemos la información también perderemos nuestros procedimientos. Esto es fácilmente subsanable llevando a cabo una buena política de respaldos de la BD.

Tener que aprender un nuevo lenguaje. Esto es siempre un problema, sobre todo si no tienes tiempo.

¿Alguna otra?

Por lo tanto es **recomendable usar procedimientos almacenados** siempre que se vaya a hacer una aplicación grande, ya que nos facilitará la tarea bastante y nuestra aplicación será más rápida.

Vamos a ver **un ejemplo** :

Abrimos una consola de MySQL seleccionamos una base de datos y empezamos a escribir:

Creamos **dos tablas** en una almacenamos las personas mayores de 18 años y en otra, las personas menores.

```
create table ninos(edad int, nombre varchar(50));

create table adultos(edad int, nombre varchar(50));
```

Ahora, introducimos personas en las tablas pero dependiendo de la edad queremos que se introduzcan en una tabla u otra, si usamos PHP comprobamos mediante código si la persona es mayor de edad.

Lo hacemos así:

```
$nombre = $_POST['nombre'];

$edad = $_POST['edad'];

if($edad < 18){

        mysql_query('insert into ninos values(' . $edad . ' , "'.
$nombre.'")');

}else{

mysql_query('insert into adultos values(' . $edad . ' , "'.
$nombre.'")');

}
```

Si la consulta es corta como en este caso, esta forma es incluso más rápida que tener que crear un procedimiento almacenado, pero si tienes que hacer esto muchas veces a lo largo de tu aplicación es mejor hacer lo siguiente:

Creamos el procedimiento almacenado:

```
delimiter //
create procedure introducePersona(in edad int,in nombre varchar(50))
begin
if edad < 18 then
insert into ninos values(edad,nombre);
else
insert into adultos values(edad,nombre);
end if;
end;
//
```

Tenemos nuestro procedimiento.

La primera linea es para decirle a MySQL que a partir de ahora hasta que no introduzcamos // no se acaba la sentencia, esto lo hacemos así por que en nuestro procedimiento almacenado tendremos que introducir el carcter ";" para las sentencias, y si pulamos enter MySQL pensará que ya hemos acabado la consulta y dará error.

Con create procedure empezamos la definición de procedimiento con nombre introducePersona. En un procedimiento almacenado existen parámetros de entrada y de salida, los de entrada (precedidos de "in") son los que le pasamos para usar dentro del procedimiento y los de salida (precedidos de "out") son variables que se establecerán a lo largo del procedimiento y una vez esta haya finalizado podremos usar ya que se quedaran en la sesión de MySQL.

En este procedimiento usamos de entrada, más adelante veremos como usar parámetros de salida.

Para **llamar a nuestro procedimiento** almacenado usamos la sentencia call:

call introducePersona(25,"JoseManuel");

Una vez tenemos nuestro procedimiento simplemente lo ejecutamos desde PHP

mediante una llamada como esta:

```
$nombre = $_POST['nombre'];

$edad = $_POST['edad'];

mysql_query('call introducePersona(' . $edad . ' ," '.$nombre.' ");');
```

De ahora en adelante, usamos el procedimiento para introducir personas en nuestra BD de manera que si tenemos 20 scritps PHP que lo usan y un buen día decidimos que la forma de introducir personas no es la correcta, sólo tenemos que modificar el procedimiento introducePersona.

VARIABLES DENTRO DE LOS PROCEDIMIENTOS

Si se declara una variable dentro de un procedimiento mediante el código :

declare miVar int;

Esta tiene

un ámbito local y cuando se acabe el procedimiento no podrá ser accedida. Una vez la variable es declarada, para cambiar su valor usaremos la sentencia SET de este modo :

set miVar = 56 ;

Para acceder a una variable a la terminación de un procedimiento se tiene que usar parámetros de salida.

Veamos algunas aplicaciones para comprobar lo sencillo que es :

IF THEN ELSE

```
delimiter //
create procedure miProc(in p1 int)      /* Parámetro de entrada */
    begin
        declare miVar int;         /* se declara variable local */
        set miVar = p1 +1 ;         /* se establece la variable */
```

```
        if miVar = 12 then
            insert into lista values(55555);
        else
insert into lista values(7665);
        end if;
    end;
//
```

SWITCH

```
delimiter //
create procedure miProc (in p1 int)
    begin
        declare var int ;
        set var = p1 +2 ;
        case var
            when 2 then insert into lista values (66666);
            when 3 then insert into lista values (4545665);
            else insert into lista values (77777777);
        end case;
    end;
//
```

Creo que no hacen falta explicaciones.

COMPARACIÓN DE CADENAS

```
delimiter //
create procedure compara(in cadena varchar(25), in cadena2
varchar(25))
    begin
```

```
        if strcmp(cadena, cadena2) = 0 then
            select "son iguales!";
        else
            select "son diferentes!!";
        end if;
    end;
//
```

La función strcmp devuelve 0 si las cadenas son iguales, si no devuelve 0 es que son diferentes.

USO DE WHILE

```
delimiter //
create procedure p14()
    begin
        declare v int;
        set v = 0;
        while v < 5 do
            insert into lista values (v);
            set v = v +1 ;
        end while;
    end;
//
```

Un while de toda la vida.

USO DEL REPEAT

```
delimiter //
create procedure p15()
    begin
```

```
        declare v int;
        set v = 20;
        repeat
            insert into lista values(v);
            set v = v + 1;
        until v >= 1
        end repeat;
    end;
//
```

El repeat es similar a un “do while” de toda la vida.

LOOP LABEL

```
delimiter //
create procedure p16()
    begin
        declare v int;
        set v = 0;
        loop_label : loop
            insert into lista values (v);
            set v = v + 1;
            if v >= 5 then
            leave loop_label;
            end if;
        end loop;
    end;
//
```

Este es otro tipo de loop, la verdad es que teniendo los anteriores no se me ocurre

aplicación para usar este tipo de loop, pero es bueno saber que existe por si algún día te encuentras algún procedimiento muy antiguo que lo use. El código que haya entre loop_label : loop y end loop; se ejecutara hasta que se encuentre la sentencia leave loop_label; que hemos puesto en la condición, por lo tanto el loop se repetirá hasta que la variable v sea >= que 5.

El loop puede tomar cualquier nombre, es decir puede llamarse miLoop: loop, en cuyo caso se repete hasta que se ejecute la sentencia leave miLoop.

Con esto **empezamos a crear procedimientos complejos y útiles**.

A continuación, **como llevamos el control de flujo de nuestro procedimiento**. También es interesante observar el uso de las variables dentro de los procedimientos. Si se declara una variable dentro de un procedimiento con el código :

declare miVar int;

Tiene un ámbito local y cuando se acabe el procedimiento no será accedida. Una vez la variable es declarada, para cambiar su valor usamos la sentencia SET de este modo :

set miVar = 56 ;

Para acceder a una variable a la terminación de un procedimiento se tiene que usar parámetros de salida.

Vamos a ver unos ejemplos para comprobar lo sencillo que es :

IF THEN ELSE

```
delimiter //
create procedure miProc(in p1 int)     /* Parámetro de entrada */
    begin
        declare miVar int;         /* se declara variable local */
        set miVar = p1 +1 ;         /* se establece la variable */
        if miVar = 12 then
            insert into lista values(55555);
```

```
        else
insert into lista values(7665);
        end if;
    end;
//
```

SWITCH

```
delimiter //
create procedure miProc (in p1 int)
    begin
        declare var int ;
        set var = p1 +2 ;
        case var
            when 2 then insert into lista values (66666);
            when 3 then insert into lista values (4545665);
            else insert into lista values (77777777);
        end case;
    end;
//
```

Creo que no hacen falta explicaciones.

COMPARACIÓN DE CADENAS

```
delimiter //
create procedure compara(in cadena varchar(25), in cadena2
varchar(25))
    begin
        if strcmp(cadena, cadena2) = 0 then
            select "son iguales!";
        else
```

```
            select "son diferentes!!";
        end if;
    end;
//
```

La función strcmp devuelve 0 si las cadenas son iguales, si no devuelve 0 es que son diferentes.

USO DE WHILE

```
delimiter //
create procedure p14()
    begin
        declare v int;
        set v = 0;
        while v < 5 do
            insert into lista values (v);
            set v = v +1 ;
        end while;
    end;
//
Un while de toda la vida.
```

USO DEL REPEAT

```
delimiter //
create procedure p15()
    begin
        declare v int;
        set v = 20;
        repeat
            insert into lista values(v);
```

```
            set v = v + 1;
        until v >= 1
        end repeat;
    end;
//
```

El repeat es similar a un “do while” de toda la vida.

```
LOOP LABEL
delimiter //
create procedure p16()
    begin
        declare v int;
        set v = 0;
        loop_label : loop
            insert into lista values (v);
            set v = v + 1;
            if v >= 5 then
            leave loop_label;
            end if;
        end loop;
    end;
//
```

Este es otro tipo de loop, la verdad es que teniendo los anteriores no se me ocurre aplicación para usar este tipo de loop, pero es bueno saber que existe por si algún día te encuentras algún procedimiento muy antiguo que lo use. El código que haya entre loop_label : loop y end loop; se ejecutara hasta que se encuentre la sentencia leave loop_label; que hemos puesto en la condición, por lo tanto el loop se repetirá hasta que la variable v sea >= que 5.

El loop puede tomar cualquier nombre, es decir puede llamarse miLoop: loop, en cuyo caso se repetirá hasta que se ejecute la sentencia leave miLoop.