Procedimientos almacenados en MySQL

Un procedimiento almacenado es un conjunto de comandos SQL que pueden almacenarse en el servidor. Una vez que se hace, los clientes no necesitan relanzar los comandos individuales, pero pueden en su lugar referirse al procedimiento almacenado.

La sintaxis :

```
 CREATE PROCEDURE nombre_del_procedimiento ([parámetros[,...]])
[characteristicas ...] cuerpo
```

Especificar los parámetros:

```
[ IN | OUT | INOUT ] nombre_del_parametro el_tipo_de_dato
```

Algunas de las palabras reservadas para la creación de procedimientos almacenados.

- **IN** : Nos indica que el parámetro será de entrada
- **OUT** : Nos indica que el parámetro será de salida
- **INTOUT** : Esto se refiere que nuestro parámetro sera de ambas (Entrada y salida).
- **BEGIN**: Limitador del procedimiento
- **END**: Fin de nuestro procedimiento
- **DELIMITER**: Restablece el punto y coma como delimitador.
- **CALL**: para llamar al procedimiento una ves creado
- DELIMITER : por lo regular el manejador no puede interpretar la diferencia entre un query normal, y la ejecución de un procedimiento almacenado, una función o un trigger, por lo cual es necesario añadir un delimitador que le permita conocer el fin del procedimiento almacenado.
- DROP PROCEDURE IF EXISTS `proc_almacen`.`procedure1` $$ Esta sentencia SQL elimina de la base de datos al procedimiento, antes de crearlo.
- CREATE PROCEDURE `proc_almacen`.`procedure1` (in parameter1 INTEGER) CREATE PROCEDURE ` nombre_base_de_Datos `. ` nombre_proc_almacenado` (in parámetros […]) Se usan las palabras CREATE PROCEDURE para crear el procedimiento, seguido del nombre de la base de datos entre ``, separada de ella por un punto e igual entre `` el nombre del procedimiento creado y entre paréntesis los parámetros que recibe
- DECLARE; esta sentencia sirve para declara una variable local al procedimiento

Vamos a un ejemplo para que nos quede más claro el uso de procedimientos almacenados en MySQL, para ellos vamos a crear una base de datos de ejemplo la cual le llamaremos Personas.

Estructura de la base de datos de ejemplo

```
1 CREATE TABLE peques(
2     nombre VARCHAR(200),
3     edad INT
4 )ENGINE=INNODB;
5
6 CREATE TABLE abuelos(
7       nombre VARCHAR(200),
8       edad INT
9 )ENGINE=INNODB;
```

Bien la idea es con un procedimiento resolver el siguiente ejercicio:

Crear un procedimiento que de pendiendo de la edad de la persona insertada la inserte en su tabla correspondiente si es mayor de 18 la inserte en la tabla abuelos si no en la tabla peques.

El procedimiento que resuelve este ejercicio es el siguiente:

```
#Creamos un delimitador para que MySQL no interprete que se termino
una instrucción al poner ;
delimiter //
#Aquí comeienza nuestro procedimiento

CREATE PROCEDURE Persona.ejemplo2(IN nombre VARCHAR(200), IN edad INT)
BEGIN
    IF edad > 18 THEN
         INSERT INTO abuelos VALUES(nombre,edad);
    ELSE
         INSERT INTO peques VALUES(nombre,edad);
END IF;
#Terminamos nuestro procedimiento con los delimitadores que creamos //
END//
```

Ahora solo nos falta llamar al procedimiento almacenado.

```
CALL ejemplo2(TJuanito GuanavacoaT,20)
#Esta persona se debe insertar en la tabla abuelos...
```

Al crear capas de datos entre la interfaz gráfica de nuestra aplicación es fundamental conocer este tipo de herramientas que nos proporciona MySQL. Y todo depende de un buen diseño de nuestra base de datos.

```
//Sentencias DDL
create database Agenda;
use Agenda;
create table Agenda.contactos(nombre varchar(45),telefono varchar(12),
email varchar(50),id_contacto int primary key)TYPE=MyISAM; //se puede
usar también InnoDB

//Sentencias DML
insert into Agenda.contactos values (TAlminoT,T7221312567T,Tgato.balin
@yahoo.com.mxT,1),(TAnitaT,T7227843213T,Tanitayanoesanita@yahoo.com.mx
T,2);
select *from Agenda.contactos;
update Agenda.contactos set nombre=TAlmaGatoT,telefono=T7221312456T wh
ere id_contacto=1;
delete *from Agenda.contactos where nombre=TAnitaT;


//Insertar
DELIMITER $$
DROP PROCEDURE IF EXIST `agenda`.`Insertar`$$
CREATE PROCEDURE `agenda`.`Insertar`(in nombre varchar(45), telefono v
archar(12), email varchar(50), id_contacto int primary key)
BEGIN
/*DECLARE nombre varchar(45)*/
INSERT INTO contactos VALUES (nombre,telefono,email,id_contacto);
END $$
DELIMITER ;

//Borrar
DELIMITER $$
DROP PROCEDURE IF EXIST `agenda`.`Borrar`$$
CREATE PROCEDURE `agenda`.`Borrar`(in ident integer)
BEGIN
   DELETE FROM contactos id_contacto=ident;
END $$
DELIMITER ;

//Actualizar
DELIMITER $$
DROP PROCEDURE IF EXIST `agenda`.`Actualizar`$$
CREATE PROCEDURE `agenda`.`Actualizar`(in ident integer,ntelefono varc
har(12))
BEGIN
UPDATE contactos SET telefono=ntelefono WHERE id_contacto=ident;
END $$
DELIMITER ;

//Para invocar
CALL Insertar(TFernando Carraro AguirreT,T7221312686T,Tcarraro.fernand
o@gmail.comT,1);
select *from Agenda.contactos;

CALL Borrar(1);
select *from Agenda.contactos;

CALL Actualizar(1,T7221312453T);
select *from Agenda.contactos;
```

```
//más ejemplos
DELIMITER $$
DROP PROCEDURE IF EXIST `agenda`.`tipoUsuarios`$$
CREATE PROCEDURE `agenda`.`tipoUsuarios`(in no_cta integer)
BEGIN
DECLARE tipoUsuario char(15);

 IF no_cta=17 THEN
   SET tipoUsuario=TAdminT;
 ELSE
    SET tipoUsuario=TInvitadoT;
 END IF;

INSERT INTO usuarios(tipoUsuario);
END $$
DELIMITER ;

//Invocar
CALL tipoUsuarios(12);//resultado=TInvitadoT
CALL tipoUsuarios(17);//resultado=TAdminT
```

Ejemplo de procedimiento almacenado, Sirve para renumerar el diario contable de apuntes de una empresa y un ejercicio contable determinados:

XKE --> Código de Empresa.
XEJER --> Ejercicio contable.

```
drop procedure if exists PCRENUM;

DELIMITER //
create procedure PCRENUM (IN XKE CHAR(03), IN XEJER CHAR(04))
begin

   declare l_loop_end INT default 0;
   declare XASIENTO DOUBLE(8,0);
   declare XDOCUMENTO CHAR(32);
   declare TXTASIENTO CHAR(8);

   declare TMPCURSOR cursor for select DOCUMENTO from DCAST
                           WHERE KE=XKE AND EJER=XEJER AND SWDEL<>T1T
                           GROUP BY DOCUMENTO
                           ORDER BY FECHA, TIPO, ASIENTO ;

   declare continue handler for sqlstate T02000T set l_loop_end = 1;

   open TMPCURSOR;

   set XASIENTO = 0;
   set XDOCUMENTO = TT;
   set TXTASIENTO = TT;

   repeat

      fetch TMPCURSOR into XDOCUMENTO;

      if not l_loop_end then
         set XASIENTO=XASIENTO+ 1 ;
         set TXTASIENTO=LPAD(XASIENTO,08,T0T) ;
         UPDATE DCAST SET ASIENTO=TXTASIENTO, FECHAM=CURDATE()
                WHERE DOCUMENTO=XDOCUMENTO AND (LINEA>T0T AND
           LINEA<TzT) AND SWDEL<>T1T ;
      end if;

   until l_loop_end end repeat;

   close TMPCURSOR;

   REPLACE DCONT SET KE=XKE, NAT=T99T, TIPO=TKT, EJER=XEJER,
SERIE=TCCCT, NOMBRE=TCONTABILIDADT, NUMERO=XASIENTO, FECHAM=CURDATE(),
SWDEL=TT ;

end;//

DELIMITER ;
```

Funciones almacenadas en MySQL

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 normalmente con otras funciones como son, por ejemplo, MAX() o COUNT().

Su sintaxis es:

```
CREATE FUNCTION nombre_funcion ([parameter[,...]])
    RETURNS type [characteristic ...]
    function_body
```

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

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: nombre_parámetro tipo_dato (**NO se indica que el parámetro es de entrada, todos lo son**)

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 (DETERMINISTICS,
    - DETERMINISTIC A routine is considered “deterministic” if it always produces the same result for the same input parameters
    - NOT DETERMINISTIC. This is mostly used with string or math processing, but not limited to that. Oposite To "DETERMINISTIC".
    - READS SQL DATA This explicitly tells to MySQL that the function will ONLY read data from databases, thus, it does not contain instructions that modify data, but it contains SQL instructions that read data (e.q. SELECT).
    - MODIFIES SQL DATA This indicates that the routine contains statements that may write data (for example, it contain UPDATE, INSERT, DELETE or ALTER instructions).
    - NO SQL This indicates that the routine contains no SQL statements.
    - CONTAINS SQL This indicates that the routine contains SQL instructions, but does not contain statements that read or write data. This is the default if none of these characteristics is given explicitly. Examples of such statements are SELECT NOW(), SELECT 10+@b, SET @x = 1 or DO RELEASE_LOCK('abc'), which execute but neither read nor write data.
- 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 simplemente invocando su nombre, como se hace en muchos lenguajes de programación.

Observemos ahora algunos ejemplos de funciones:

```
CREATE FUNCTION HolaMundo() RETURNS VARCHAR(30)
NOT DETERMINISTIC
BEGIN
  DECLARE salida1 VARCHAR(30) DEFAULT 'Hola';
  DECLARE salida2 VARCHAR(30);

  SET salida2 = ' mundo';
  RETURN CONCAT(salida1,salida2);
END;
```

Para ver cómo crear una función con parámetros:

```
CREATE FUNCTION HolaMundo(entrada VARCHAR(30)) RETURNS VARCHAR(30) NOT
DETERMINISTIC
BEGIN
          DECLARE salida VARCHAR(30);
          SET salida = entrada;
           RETURN salida
END;
```

Desde una función podemos invocar a su vez a otras funciones o procedimientos.
Las variables en las funciones se declaran con la sentencia DECLARE, y se asignan valores con la sentencia SET. Esta variable es de ámbito local, y será destruida una vez finalice la función. Si se desea asignar un valor por defecto a la variable en el momento de declararla se puede utilizar la sentencia DEFAULT junto con la sentencia DECLARE.

```
mysql> delimiter //

mysql> CREATE PROCEDURE procedimiento (IN cod INT)

BEGIN

       SELECT * FROM tabla WHERE cod_t = cod;

END //

mysql> delimiter ;

mysql> CALL procedimento(4);
```

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

Lo interesante de esto es que podemos escribir 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 luego trabajar con él, por eso ese parámetro es de tipo IN. Definimos el parámetro como OUT cuando en él se va aguardar la salida del procedimiento. Si el parámetro hubiera 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 último ejemplo se deberán usar nuevos delimitadores, como se vio anteriormente.

Observemos ahora un ejemplo 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);
```

Vamos a mostrar un ejemplo de uso de triggers y procedimientos almacenados paso a paso.

Nuestro sistema tendrá dos tablas, una con ventas y la otra con comisiones. En la primera se almacenarán cada venta que se realiza en un comercio, y en la segunda las comisiones que le corresponden a cada vendedor en el momento.

```
CREATE TABLE ventas (
      vendedor int(11),
      producto int(11),
      fecha date,
      importe float
)

CREATE TABLE comisiones (
      vendedor int(11),
      comision float
)
```

Las comisiones se calculan de una forma especial, le corresponde un porcentaje de las ventas según el tipo de producto, y es importante para los vendedores el que se sepa en cada momento qué comisiones lleva ganadas (esto es una justificación para no usar un *cron* o algo parecido).

Para calcular qué comisión le corresponde a un vendedor, calcularemos los porcentajes para cada tipo de producto vendido y luego lo añadiremos/actualizaremos en la tabla de comisiones. Todo se realizará en un procedimiento almacenado.

La comisión varía en función del producto de forma que si el producto es:

- Producto = 1 la comisión es del 15% del importe
- Producto = 2 la comisión es del 10% del importe
- Producto = 3 la comisión es del 20% del importe

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `test`.`sp_comisiones`$$
CREATE PROCEDURE `test`.`sp_comisiones` (IN mivendedor INT)
BEGIN
DECLARE micomision INT DEFAULT 0;
DECLARE suma INT;
DECLARE existe BOOL;

select IFNULL(sum(importe),0) into suma from ventas where producto = 1
and vendedor = mivendedor;
SET micomision = micomision + (suma * 0.15);

select IFNULL(sum(importe),0) into suma from ventas where producto = 2
and vendedor = mivendedor;
SET micomision = micomision + (suma * 0.1);

select IFNULL(sum(importe),0) into suma from ventas where producto = 3
and vendedor = mivendedor;
SET micomision = micomision + (suma * 0.2);

select count(1)>0 into existe from comisiones where vendedor =
mivendedor;

if existe then
UPDATE comisiones set comision = comision+micomision where vendedor =
mivendedor;
else
insert into comisiones (vendedor, comision) values (mivendedor,
micomision);
end if;
END$$
DELIMITER ;
```

Ahora, para actualizar los datos de las comisiones usaremos un trigger, así cuando se haga una venta (insert en la tabla ventas), se llamará al procedimiento almacenado (*sp_comisiones*), que recalculará la comisión para ese vendedor.

```
DELIMITER $$

DROP TRIGGER `test`.`tr_ventas_insert`$$
CREATE TRIGGER `test`.`tr_ventas_insert` AFTER INSERT on
`test`.`ventas`
FOR EACH ROW BEGIN
call sp_comisiones(new.vendedor);
END$$

DELIMITER ;
```