Funciones en MySql

Una **función** en **MySQL** es una **rutina** creada para tomar unos parámetros, procesarlos y retornar en un salida.

Se diferencian de los **procedimientos** en las siguientes características:

- Solamente pueden tener parámetros de entrada **IN** y no parámetros de salida **OUT** o **INOUT**
- Deben retornar en un valor con algún **tipo de dato** definido
- Pueden usarse en el contexto de una sentencia **SQL**
- Solo retornan un valor individual, no un conjunto de registros.

Como creo una función?

Debes usar la sentencia **CREATE FUNCTION**. La sintaxis para crear una función es casi idéntica a la de **crear un procedimiento**, veamos:

```
CREATE FUNCTION nombre_función (parametro1,parametro2,...)
RETURNS tipoDato
[atributos de la rutina]
<bloque de instruccciones>
```

La única diferencia entre la creación de un procedimiento y una función es que la sintaxis de una función contiene la palabra reservada **RETURNS** para indicar que tipo de dato se retornará.

Podemos ver un ejemplo?

Claro! Fíjate como obtenemos el factorial de un numero x ingresado como parámetro:

```
DELIMITER //

CREATE FUNCTION factorial(x INT) RETURNS INT(11)
BEGIN
        DECLARE factorial INT;

        -- Guardamos el valor de x
        SET factorial = x ;

        -- Caso en que x sea menor o igual a 0
        IF x <= 0 THEN
        RETURN 1;
        END IF;

        -- Iteramos para obtener multiplicaciones
        consecutivas

        bucle: LOOP

        -- Cada iteracion reducimos en 1 a x
        SET x = x - 1 ;

        -- Condición de parada del bucle
```

```
        IF x<1 THEN
        LEAVE bucle;
        END IF;

        -- Factorial parcial
        SET factorial = factorial * x ;

        END LOOP bucle;

        -- Retornamos en el factorial
        RETURN factorial;

END//
DELIMITER ;
```

No confundas RETURNS con RETURN. La primera es para indicar el tipo de dato de retorno de la función y la segunda es para retornar el valor en el cuerpo de la función.

Puedes mostrar como usar una función en un SELECT?

A continuación crearemos un función que retorne en el nombre completo de la prioridad de un cliente, introduciendo como parámetro el campo prioridad.

Creación de la función:

```
DELIMITER //
CREATE FUNCTION EXT_PRIORIDAD (cliente_prioridad VARCHAR(5)) RETURNS
VARCHAR(20)
BEGIN
        CASE cliente_prioridad

        WHEN 'A' THEN
            RETURN 'Alto';
        WHEN 'M' THEN
            RETURN 'Medio';
        WHEN 'B' THEN
            RETURN 'Bajo';
        ELSE
            RETURN 'NN';
        END CASE;
END//
DELIMITER ;
```

Con ella podremos consultar de la siguiente forma a los clientes de la base de datos:

```
SELECT NOMBRE, APELLIDO, EXT_PRIORIDAD(PRIORIDAD)
FROM CLIENTE;
```

De esta manera hemos usado nuestra función en un contexto de consulta. También podemos usar funciones en las sentencias **DELETE** y **UPDATE**, siempre y cuando el valor retornado sea acorde con las necesidades.

Como actualizar una función?

Para actualizar una función usamos el comando **ALTER FUNCTION**. Con esta sentencia podemos cambiar los atributos de la función, pero no podremos cambiar el cuerpo. Veamos la sintaxis:

```
ALTER FUNCTION nombre_funcion
[SQL SECURITY {DEFINER|INVOKER}]
[COMMENT descripción ]
```

Si quisiéramos añadir una descripción a una función que calcula el promedio de huéspedes diario con respecto a una fecha llamada **promedio_huespedes,** hacemos lo siguiente:

```
ALTER FUNCTION promedio_huespedes
COMMENT 'Calculo del promedio diario de huéspedes entre una fecha inicial
y una fecha final';
```

Como borrar una función?

Usando el comando **DROP FUNCTION**. Simplemente especificamos el nombre de la función y esta se borrará de la **base de datos**. Su sintaxis esta definida de la siguiente forma:

```
DROP FUNCTION nombre_funcion
```

Por ejemplo, para borrar una función que retorna en el **ingreso neto** con respecto a todos los **tiquetes aéreos** comprados en una sucursal de una **aerolínea**, llamada **ingreso_neto_sucursal**:

```
DROP FUNCTION ingreso_neto_sucursal;
```

Disparadores en MySql

Un **Trigger** en **MySQL** es un **programa almacenado**(***stored program***), creado para ejecutarse automaticamente cuando ocurra un evento en nuestra **base de datos**. Dichos eventos son generados por los comandos **INSERT**, **UPDATE** y **DELETE**, los cuales hacen parte del **DML(Data Modeling Languaje) de SQL**.

Esto significa que invocaremos nuestros **Triggers** para ejecutar un **bloque de instrucciones** que proteja, restrinja o preparen la información de nuestras tablas, al momento de manipular nuestra información. Para crear triggers en MySQL necesitas los privilegios SUPER Y TRIGGER.

Crear un Trigger en MySQL

Usaremos una sintaxis similar a la creación de **Procedimientos** y **Funciones** en MySQL. Observemos:

```
CREATE [DEFINER={usuario|CURRENT_USER}]
TRIGGER nombre_del_trigger {BEFORE|AFTER} {UPDATE|INSERT|DELETE}
ON nombre_de_la_tabla
FOR EACH ROW
<bloque_de_instrucciones>
```

Obviamente la sentencia CREATE es conocidisima para crear nuevos objetos en la base de datos. Eso ya lo tienes claro. Enfoquemos nuestra atención en las otras partes de la definición:

- DEFINER={usuario|CURRENT_USER}
  : Indica al **gestor de bases de datos** qué usuario tiene privilegios en su cuenta, para la invocación de los triggers cuando surjan los eventos **DML**. Por defecto este característica tiene el valor CURRENT_USER que hace referencia al usuario actual que esta creando el Trigger.
- nombre_del_trigger:
  Indica el nombre de nuestro trigger. Existe una **nomenclatura** muy práctica para nombrar un trigger, la cual nos da mejor legibilidad en la administracion de la base de datos. Primero ponemos el nombre de tabla, luego especificamos con la inicial de la operación DML y seguido usamos la inicial del momento de ejecución(AFTER o BEFORE). Por ejemplo:

```
 -- BEFORE INSERT
clientes_BI_TRIGGER
```

- BEFORE|AFTER: Especifica si el Trigger se ejecuta antes o después del evento DML.
- UPDATE|INSERT|DELETE:
  Aquí eliges que sentencia usarás para que se ejecute el Trigger.

- ON nombre_de_la_tabla:
  En esta sección estableces el nombre de la tabla asociada.
- FOR EACH ROW: Establece que el Trigger se ejecute por cada fila en la tabla asociada.
- <bloque_de_instrucciones>: Define el bloque de sentencias que el Trigger ejecutará al ser invocado.

Identificadores NEW y OLD en Triggers

Si queremos relacionar el trigger con columnas especificas de una tabla debemos usar los identificadores OLD y NEW.
OLD indica el valor antiguo de la columna y NEW el valor nuevo que pudiese tomar. Por ejemplo: OLD.idproducto ó NEW.idproducto.
Si usamos la sentencia UPDATE podremos referirnos a un valor OLD y NEW, ya que modificaremos registros existentes por nuevos valores. En cambio si usamos INSERT solo usaremos NEW, ya que su naturaleza es únicamente de insertar nuevos valores a las columnas. Y si usamos DELETE usaremos OLD debido a que borraremos valores que existen con anterioridad.

Triggers BEFORE y AFTER

Estas clausulas indican si el Trigger se ejecuta **antes** o **después** del evento DML. Hay ciertos eventos que no son compatibles con estas sentencias.
Por ejemplo, si tuvieras un **Trigger AFTER** que se ejecuta en una sentencia UPDATE, sería ilógico editar valores nuevos NEW, sabiendo que el evento ya ocurrió. Igual sucedería con la sentencia INSERT, el Trigger tampoco podría referenciar valores NEW, ya que los valores que en algún momento fueron NEW, han pasado a ser OLD.

¿Qué utilidades tienen los Triggers?

Con los Triggers podemos implementar varios casos de uso que mantengan la integridad de la base de datos, como ***Validar información***, ***Calcular atributos derivados***, ***Seguimientos de movimientos en la base de datos***, etc.
Cuando surja una necesidad en donde veas que necesitas que se ejecute una acción implícitamente(sin que la ejecutes manualmente) sobre los registros de una tabla, entonces puedes considerar el uso de un Trigger.

Ejemplo de Trigger BEFORE en la sentencia UPDATE

A continuación veremos un Trigger que valida la edad de un cliente antes de una sentencia `UPDATE`. Si por casualidad el nuevo valor es negativo, entonces asignaremos `NULL` a este atributo.

```
DELIMITER //
CREATE TRIGGER cliente_BU_Trigger
```

```
BEFORE UPDATE ON cliente FOR EACH ROW
BEGIN
-- La edad es negativa?
IF NEW.edad
```

Este **Trigger** se ejecuta antes de haber insertado el registro, lo que nos da el poder de verificar primero si el nuevo valor de la edad esta en el rango apropiado, si no es así entonces asignaremos `NULL` a ese campo. **Grandes los Triggers!**

Ejemplo de Trigger AFTER en la sentencia UPDATE

Supongamos que tenemos una ***Tienda de accesorios para Gamers***. Para la actividad de nuestro negocio hemos creado un **sistema de facturación** muy sencillo, que registra las ventas realizadas dentro de una factura que contiene el detalle de las compras.

Nuestra tienda tiene 4 vendedores de turno, los cuales se encargan de registrar las compras de los clientes en el horario de funcionamiento.

Implementaremos un Trigger que guarde los cambios realizados sobre la tabla `DETALLE_FACTURA` de la base de datos realizados por los vendedores.

Veamos la solución:

```
DELIMITER //
CREATE TRIGGER detalle_factura_AU_Trigger
AFTER UPDATE ON detalle_factura FOR EACH ROW
BEGIN
INSERT INTO log_updates
(idusuario, descripcion)
VALUES (user( ),
CONCAT('Se modificó el registro ','(',
OLD.iddetalle,',', OLD.idfactura,',',OLD.idproducto,',',
OLD.precio,',', OLD.unidades,') por ',
'(', NEW.iddetalle,',', NEW.idfactura,',',NEW.idproducto,',',
NEW.precio,',', NEW.unidades,')'));

END//

DELIMITER ;
```

Con este registro de ***logs*** podremos saber si algún vendedor “ocioso” esta alterando las facturas, lo que lógicamente sería atentar contra las finanzas de nuestro negocio. Cada registro nos informa el usuario que modificó la tabla `DETALLE_FACTURA` y muestra una descripción sobre los cambios en cada columna.

Ejemplo de Trigger BEFORE en al sentencia INSERT

El siguiente ejemplo que te voy a mostrar **¡me encanta!**, ya que muestra como mantener la integridad de una base de datos con respecto a una atributo derivado.

Supón que tienes una ***Tienda de electrodomésticos*** y que has implementado un **sistema de facturación**. En la base de datos que soporta la información de tu negocio, existen varias tablas, pero nos vamos a centrar en la tabla `PEDIDO` y la tabla `TOTAL_VENTAS`.

`TOTAL_VENTAS` almacena las ventas totales que se le han hecho a cada cliente del negocio. Es decir, si el cliente **Armado Barreras** en una ocasión compró 1000 dolares, luego compró 1250 dolares y hace poco ha vuelto a comprar 2000 dolares, entonces el total vendido a este cliente es de 4250 dolares.

Pero supongamos que eliminamos el ultimo pedido hecho por este cliente, ¿que pasaría con el registro en `TOTAL_VENTAS` ?,…**¡exacto!**, quedaría desactualizado.

Usaremos tres Triggers para solucionar esta situación. Para que cada vez que usemos un comando DML en la tabla `PEDIDO`, no tengamos que preocuparnos por actualizar manualmente `TOTAL_VENTAS`.

Veamos:

```
-- TRIGGER PARA INSERT
DELIMITER //
CREATE TRIGGER PEDIDO_BI_TRIGGER
BEFORE INSERT ON PEDIDO
FOR EACH ROW
BEGIN
DECLARE cantidad_filas INT;
SELECT COUNT(*)
INTO cantidad_filas
FROM TOTAL_VENTAS
WHERE idcliente=NEW.idcliente;
IF cantidad_filas > 0 THEN
UPDATE TOTAL_VENTAS
SET total=total+NEW.total
WHERE idcliente=NEW.idcliente;
ELSE
INSERT INTO TOTAL_VENTAS
(idcliente,total)
VALUES(NEW.idcliente,NEW.total);
END IF;
END//


-- TRIGGER PARA UPDATE
CREATE TRIGGER PEDIDO_BU_TRIGGER
BEFORE UPDATE ON PEDIDO
FOR EACH ROW
BEGIN
UPDATE TOTAL_VENTAS
SET total=total+(NEW.total-OLD.total)
WHERE idcliente=NEW.idcliente;
END//


-- TRIGGER PARA DELETE
```

```
CREATE TRIGGER PEDIDO_BD_TRIGGER
BEFORE DELETE ON PEDIDO
FOR EACH ROW
BEGIN
UPDATE TOTAL_VENTAS
SET total=total-OLD.total
WHERE idcliente=OLD.idcliente;
END//
```

Con todos ellos mantendremos el total de ventas de cada cliente actualizado dependiendo del evento realizado sobre un pedido.

Si insertamos un nuevo pedido generado por un cliente existente, entonces vamos rápidamente a la tabla `TOTAL_VENTAS` y actualizamos el total comprado por ese cliente con una sencilla suma acumulativa.

Ahora, si cambiamos el monto de un pedido, entonces vamos a `TOTAL_VENTAS` para descontar el monto anterior y adicionar el nuevo monto.

Y si eliminamos un pedido de un cliente simplemente descontamos del total acumulado el monto que con anterioridad habíamos acumulado. **¿Práctico cierto?**

Ver la información de un Trigger en MySQL

Si!, usa el comando SHOW CREATE TRIGGER y rápidamente estarás viéndolas especificaciones de tu Trigger creado. Observa el siguiente ejemplo:

```
SHOW CREATE TRIGGER futbolista_ai_trigger;
```

También puedes ver los Triggers que hay en tu base de datos con:

```
SHOW TRIGGERS;
```

Eliminar un Trigger en MySQL

DROP, **DROP** y mas **DROP**. Como ya sabes usamos este comando para eliminar casi cualquier cosa en nuestra base de datos:

```
DROP TRIGGER [IF EXISTS] nombre_trigger
```

Recuerda que podemos adicionar la condicion IF EXISTS para indica que si el Trigger ya existe, entonces que lo borre.