Jorge Domínguez Chávez

MySQL

Triggers, Funciones y Procedimientos

2015

© 2015 Jorge Domínguez Chávez

Esta obra se distribuye bajo licencia Creative Commons

http://creativecommonsvenezuela.org.ve

Reconocimiento

Atribución: permite a otros copiar, distribuir, exhibir, y realizar su trabajo con derechos de autor y trabajos derivados basados en ella - pero sólo si ellos dan crédito de la manera que usted solicite.

Compartir Igual: permite que otros distribuyan trabajos derivados sólo bajo una licencia idéntica a la licencia que rige el trabajo original.

© 2015 Jorge Domínguez Chávez

ISBN

Publicado por IEASS, Editores.

www.ieass.com.ve

Venezuela, 2015

La portada y contenido de este libro ha sido creado en LibreOffice Versión: 4.4.1.2

PROLOGO

Hoy, mientras leía mensajes del correo antiguo rescaté unos apuntes de mis cursos de programación que dicté en los últimos años. Los apuntes eran de **MySQL**, más específicamente sobre **triggers, funciones y procedimientos almacenados (ps)** y todo eso que MySQL implementa a partir de su versión 5.

MySQL ha sido siempre un motor de bases de datos **muy rápido** y muy utilizado para proyectos open source de código abierto, sobre todo para proyectos web dada su gran velocidad. Se ha convertido en la actualidad en una solución viable y de misión critica para la administración de datos. Antes, MySQL se consideraba como la opción ideal para sitios Web; sin embargo, ahora incorpora muchas de las funciones necesarias para otros entornos y conserva su gran velocidad. MySQL supera desde hace tiempo a muchas soluciones comerciales en velocidad y dispone de un sistema de permisos elegante y potente, y ahora, ademas, desde la versión 4 incluye el motor de almacenamiento InnoDB compatible con ACID.Pero también a sido muy **criticado por la falta de características avanzadas** que otros Sistemas Gestores de Bases de Datos, como Oracle, Informix, SQLServer de Microsoft o PostgreSQL tienen.

MySQL es un sistema gestor de bases de datos (SGBD, DBMS por sus siglas en inglés) muy conocido y ampliamente usado por su simplicidad y notable rendimiento. Además está implementado algunas características avanzadas disponibles en otros SGBD del mercado, es una opción atractiva tanto para aplicaciones comerciales, como académicas y/o de entretenimiento precisamente por su facilidad de uso y tiempo reducido de puesta en marcha. Esto y su libre distribución en Internet bajo licencia GPL le otorgan como beneficios adicionales (no menos importantes) contar con un alto grado de estabilidad y un rápido desarrollo.

MySQL está disponible para múltiples plataformas, la seleccionada para los ejemplos de este libro es GNU/Linux. Sin embargo, las diferencias con cualquier otra plataforma son prácticamente nulas, ya que la herramienta utilizada en este caso es el cliente mysql-client, que permite interactuar con un servidor MySQL (local o

remoto) en modo texto. De este modo es posible realizar los ejercicios sobre un servidor instalado localmente o, a través de Internet, sobre un servidor remoto.

Para la realización de las actividades, es imprescindible que dispongamos de los datos de acceso del usuario administrador de la base de datos. Aunque en algunos de ellos los privilegios necesarios serán menores, para los capítulos que tratan la administración del SGBD será imprescindible disponer de las credenciales de administrador.

Estas características avanzadas son los procedimientos almacenados, triggers, transacciones y demás cosas.

HISTORIA DE MYSQL

Empezamos con la intención de usar mSQL para conectar a nuestras tablas utilizando nuestras propias rutinas rápidas de bajo nivel (ISAM). Sin embargo y tras algunas pruebas, llegamos a la conclusión que mSQL no era lo suficientemente rápido o flexible para nuestras necesidades. Esto provocó la creación de una nueva interfaz SQL para nuestra base de datos pero casi con la misma interfaz API que mSQL. Esta API fue diseñada para facilitar código de terceras partes que fue escrito para poder usarse con mSQL para ser fácilmente portado para el uso con MySQL.

La derivación del nombre MySQL no está clara. Nuestro directorio base y un gran número de nuestras bibliotecas y herramientas han tenido el prefijo "my" por más de 10 años. Sin embargo, la hija del cofundador Monty Widenius también se llama My. Cuál de los dos dió su nombre a MySQL todavía es un misterio, incluso para nosotros.

El nombre del delfín de MySQL (el logo) es "Sakila", que fué elegido por los fundadores de MySQL AB de una gran lista de nombres sugerida por los usuarios en el concurso "Name the Dolphin" (ponle nombre al delfín). El nombre ganador fue enviado por Ambrose Twebaze, un desarrollador de software Open Source de Swaziland, África. Según Ambrose, el nombre femenino de Sakila tiene sus raíces en SiSwate, el idioma local de Swaziland. Sakila también es el nombre de una ciudad en Arusha, Tanzania, cerca del país de origen de Ambrose, Uganda.

Este libro te enseña a emplear los triggers, las funciones y los procedimientos en el gestor de base de datos MySQL; así como a trabajar en línea de comando cliente mysql de MySQL. Antes de comenzar, deberías saber:

1....cómo introducir comandos en la línea de comandos

2....tu contraseña y nombre de usuario de MySQL

3....el nombre de la base de datos MySQL a la que quieres acceder

4....la dirección ip o el hostmane del servidor de la base de datos (si no está en tu local host)

Este libro tiene 4 secciones:

1. **Introducción**- Es lo que estás leyendo ahora mismo.
2. **Triggers**- Cómo programar, ejecutar los triggers y entender su funcionamiento y sintaxis. (insert, update, delete)
3. **Funciones**- Cómo programar, ejecutar las funciones y entender su operación y sintaxis. Ejemplos de SELECT.
4. **Procedimientos**- Cómo programar, ejecutar los procedimientos almacenados (PS) y entender su funcionamiento y sintaxis.

Este libro está diseñado para enseñarte las bases de la obtención de información dentro y fuera de una base de datos MySQL existente, utilizando la línea de comando cliente MySQL.

El libro **NO** es una referencia completa, y no trata ni de lejos ningún tipo de tópicos avanzados. Es simplemente una toma de contacto rápida para la utilización de la línea de comando `mysql` de MySQL.

Este material ha sido preparado de mis clases en las cátedras de base de datos, programación PHP, administración de base de datos.

DEDICATORIA

A las dos grandes mujeres en mi vida: mi Madre Amparo y mi esposa Dianella.

Jorge

INDICE

Índice de contenido

INTRODUCCIÓN

El libro está pensado para ser desarrollado por una persona que tiene un conocimiento intermedio de programación. El objetivo es aprender MySQL en forma sencilla viendo un concepto teórico, luego algunos ejercicios resueltos y por último y lo más importante efectuar una serie de ejercicios. Puede desarrollar los ejercicios en el libro, probarlos y ver los resultados. Un conocimiento profundo de MySQL facilitará el desarrollo de páginas dinámicas con PHP que acceden a una base de datos MySQL.

Mientras que para llevar una contabilidad, una agenda de contactos, facturas, etc., almacenar datos "alfanuméricas" sirve cualquier producto del mercado, el "motor" o "gestor" de un sistema de gestión documental tiene unos requisitos más exigentes. Hace muchos años que la continua evolución del hardware ha resuelto los problemas de capacidad y rendimiento que plantea guardar en un PC corriente un archivo con 40, 50, 300 mil o más registros. Los datos "alfanuméricos", nombres, apellidos, números de teléfono, cuentas contables ocupan unos cuantos bytes y, por muchos que tengamos, siempre hablamos de kilobytes. Unos pocos "Megas" a lo sumo. ¿Qué son 3 o 4 MegaBytes en un PC que tiene un disco duro con 1 TeraByte. Es un problema resuelto.

Ahora un "registro" de la base de datos, cada uno de los elementos que la componen, es un archivo. Puede ser un documento con texto (20, 30, 140Kb...), PDF (50, 100, 200Kb), hoja electrónica de cálculo (otro tanto). Puede que tenga imágenes, logotipos, fotos, documentos escaneados con una buena resolución, con lo que ya el orden de magnitud empieza a ser de Mb, más que Kb. No es difícil que un usuario incorpore 2 o 3 mil documentos al año con lo que nos empezamos a mover (en una PYME de 10 usuarios) en cifras de varios Gb anuales. Esto ya no es fácil de gestionar. Y menos sencillo es garantizar un rendimiento adecuado al introducir los nuevos documentos, y al consultar los existentes.

Ahora que ya ha decidido lo que quiere, la pregunta de siempre: ¿cual elegir? Porque, como pasa con todo en la sociedad actual, la oferta es enorme, variada y con frecuencia contradictoria. Unos comentarios sobre cómo y porqué elegimos

nosotros la base de datos sobre la que “montar” nuestra aplicación. (En realidad, más correcto es decir SGBD, sistema de gestión de base de datos).

Primera decisión: software libre o propietario. ¡Ya estamos otra vez! Y es más complicado aún, porque puedes implementar un programa desarrollado con un lenguaje “propietario” como Delphi, Visual net o Visual Basic, sobre un SGBD libre, o no, y, a su vez, sobre un servidor con Linux o Windows Server o UNIX.

Si solamente nos centramos en la base de datos (SGBD):

“Libres” o gratuitas hay varias: MySQL, PostgreSQL, SQLite3, MariaDB, Firebird, las cuatro primeras son las más conocidas y, sin duda, son válidas para un proyecto, por complicado que sea. Los sistemas GNU/Linux o software libre o de código abierto ya no son una novedad más, son una realidad[1].

“Propietarias” hay más. Dejando de lado las menos “potentes”, dBASE (un clásico, pero muy superada), FileMaker, Interbase, Access. Tenemos las dos más conocidas, Oracle y SQL Server de Microsoft, además de IBM DB2, Informix, Progress.

Y un tercer grupo, las “super-propietarias”. Las creadas por empresas, normalmente pequeñas, a medida para su programa de gestión, documental o no.

¿Cual elegir? La respuesta es muy sencilla: todas sirven. Todas. Esto ya está inventado. Cualquiera de ellas es válida, si se configura y utiliza bien. La decisión no es en función de que sirva o no para mi necesidad. Seguro que sirve. La decisión estará basada en otras características.

Evitamos las “exóticas” (en PYMES, insisto, como Progress, Informix) porque es difícil conseguir ayuda técnica cuando tienes un problema. Un argumento definitivo para los programadores.

En cuanto al software libre, es casi una decisión filosófica. Es como ser usuario de Mac, o tener un iPhone. O como convencer al que tiene una Canon reflex fabulosa de que las Nikon son mejores. El software libre es una decisión válida y técnicamente irreprochable. Y lo es, igualmente, montar Oracle. Hay espacio para todos y hay aplicaciones estupendas en los dos sistemas. No van a ser mejores o peores los programas, ni los usuarios van a trabajar mejor, por el hecho de haber

1 http://www.linuxjournal.com

elegido uno u otro gestor de base de datos. La clave está en el software, en el interfaz de usuario, en la facilidad de uso, y eso es independiente de la tecnología.

Todo depende del uso que haga de MySQL y de nuestro conocimiento y experiencia sobre él.

Por lo tanto, el objetivo de este libro es que aprendamos a programar disparadores, procedimientos, y funciones en MySQL.

Un trigger es un objeto asociado a una tabla que es ejecutado cuando sucede un evento en la tabla propietaria. Son aquellas sentencias (INSERT, UPDATE, DELETE) que modifican los datos, sujetos o no a ciertas condiciones, dentro de una tabla. Sólo puede haber un trigger de cada insert, update y delete por tabla o vista.

Los procedimientos son códigos que se ejecutan directamente en el servidor. Un procedimiento se invoca usando un comando CALL, y sólo puede pasar valores usando variables de salida.

Una función puede llamarse desde dentro de un comando como cualquier otra función (esto es, invocando el nombre de la función), y puede retornar un valor escalar. Las rutinas almacenadas pueden llamar otras rutinas almacenadas.

Tanto los triggers, los procedimientos y las funciones se asocian con la base de datos abierta, en uso.

Y como nada hacen más que la práctica, Para captar mejor la teoría, se harán muchos ejercicios con los alumnos, para probar la teoría y verificar la integración de la materia.

También, el alumno podrá copiar sus códigos en un pendrive (memoria usb) o disco externo o a DropBox al fin del curso para llevarse, con propósito de seguir la práctica en su hogar.

CAPÍTULO 1

TRIGGERS

¿QUÉ ES UN TRIGGER?

Desde la aparición de la versión 5.0.3 de la base de datos MySQL, se ha implementado la posibilidad de desarrollar triggers para las tablas de una base de datos. Recordar que un trigger es un objeto asociado a una tabla que es ejecutado cuando sucede un evento en la tabla propietaria. Son aquellas sentencias (INSERT, UPDATE, DELETE) que modifican los datos dentro de una tabla. Sólo puede haber un trigger de cada insert, update y delete por tabla o vista.

Pero los triggers tienen una ventaja obvia. Es código que se ejecuta en el servidor de la base de datos, y no en la máquina cliente.

Un **Trigger o Disparador** es un **programa almacenado**(*stored programo SP*), creado para ejecutarse automáticamente cuando ocurra un evento en una tabla o vista de la base de datos. Dichos eventos son generados por los comandos INSERT, UPDATE y DELETE, los cuales forman parte del DM[2] (Data Modeling Language) de SQL[1].

Un Disparador nunca se llama directamente, en cambio, cuando una aplicación o usuario intenta insertar, actualizar, o anular una fila en una tabla, la acción definida en el disparador se ejecuta automáticamente (se dispara).

Las ventajas de los triggers son varias:

1. La entrada en vigor automática de restricciones de los datos, hace que los usuarios entren sólo valores válidos.
2. El mantenimiento de la aplicación se reduce, los cambios a un disparador se refleja automáticamente en todas las aplicaciones que tienen que ver con la

2 Los TRIGGERS DDL tiene una particularidad adicional sobre los de tipo DML y es que no tiene mucho sentido las tablas inserted y deleted ya que el tipo de operaciones que disparan los triggers son radicalmente diferentes. Sin embargo, como ellos necesitan recibir información acerca del evento que ha ocasionado que el trigger se dispare, para ello existe la función EVENTDATA()

tabla sin la necesidad de recompilar o enlazar.

3. Logs automáticos de cambios a las tablas. Una aplicación puede guardar un registro corriente de cambios, creando un disparador que se active siempre que una tabla se modifique.
4. La notificación automática de cambios a la Base de Datos con alertas de evento en los disparadores.

Decimos que los **Triggers** se invocan para ejecutar un conjunto de instrucciones que protejan, restrinjan, actualicen o preparen la información de las tablas, al momento de manipular la información. Para crear triggers son necesarios los privilegios `SUPER y TRIGGER`.

SINTAXIS DE UN TRIGGER

Usamos la siguiente sintaxis:

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 conocida para crear nuevos objetos en la base de datos. Explicación de las 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 esta característica tiene el valor `CURRENT_USER` que hace referencia al usuario actual que esta creando el Trigger.
- `nombre_del_trigger`: Indica el nombre del trigger. Por defecto existe una **nomenclatura** práctica para nombrar un trigger, la cual da mejor legibilidad en la administración de la base de datos. Primero, escriba el

nombre de tabla, luego especifique 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 elija que sentencia usa para que se ejecute el Trigger.
- `ON nombre_de_la_tabla`: En esta sección establece 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 ejecuta al ser invocado.

IDENTIFICADORES NEW Y OLD EN TRIGGERS

Si requiere 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 o NEW.idproducto`.

Si usa la sentencia `UPDATE se` refiere a un valor `OLD` y `NEW`, ya que modifica registros existentes por los valores. En cambio, si usa `INSERT` sólo usa `NEW`, ya que su naturaleza es únicamente de insertar nuevos valores a las columnas. Y con `DELETE` usa `OLD` debido a que borra valores existentes.

TRIGGERS BEFORE Y AFTER

Estas clausulas indican si el Trigger se ejecuta BEFORE (**antes)** o AFTER (**después)** del evento DML. Hay ciertos eventos que no son compatibles con estas sentencias.

Sí tiene un **Trigger AFTER** que se ejecuta en una sentencia `UPDATE`, es ilógico editar valores nuevos `NEW`, sabiendo que el evento ya ocurrió. Igual ocurre con la sentencia `INSERT`, el Trigger tampoco podría hacer referencia a 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 implementamos varios casos de uso que mantengan la integridad de la tabla de la base de datos, como *Validar información*, *Calcular atributos derivados*, *Seguimiento de movimientos de datos en tablas de la base de datos*, etc.

Cuando surja una necesidad para que se ejecute una acción implícitamente (sin ejecución manual) sobre los registros de una tabla, considera el uso de un Trigger.

Aunque los **triggers** sirven en su mayoría para mantener la integridad de la base de datos también pueden usarse para:

AUDITORIA

- Los **triggers** se usan para llenar tablas de auditoría, en donde se registren ciertos tipos de transacciones o accesos hacia tablas.

REGLAS DE NEGOCIO

- Cuando ciertas reglas de negocio son muy elaboradoras y necesitan ser expresadas a nivel base de datos, es necesario que además de reglas y restricciones se utilicen **triggers**.

VALIDACIÓN DE DATOS

- Los **triggers** pueden controlar ciertas acciones, por ejemplo: pueden rechazar o modificar ciertos valores que no cumplan determinadas reglas,

para prevenir que datos inválidos sean insertados en la base.

SEGURIDAD

- Refuerzan las reglas de seguridad de una aplicación.
- Ofrecen verificación de seguridad basada en valores.
- Integridad de los datos mejorada.
- Fuerzan restricciones dinámicas de integridad de datos y de integridad referencial.
- Aseguran que las operaciones relacionadas se realizan juntas de forma implícita.
- Respuesta instantánea ante un evento auditado.
- Ofrece un mayor control sobre la Base de Datos.

DESVENTAJAS

- Hay que definir con anticipación la tarea que realizara el trigger
- Peligro de pérdida en Reorganizaciones.
- Hay que programarlos para cada DBMS.
- Un Trigger nunca se llama directamente.
- Los triggers no se desarrollan pensando en un sólo registro, los mismos deben funcionar en conjunto con los datos ya que se disparan por operación y no por registro.
- Por funcionalidad, no hay que poner en uno solo las funciones de INSERT, UPDATE y DELETE.
- Utilizar moderadamente los triggers.
- No se pueden utilizar en tablas temporales.

APLICACIONES

TRIGGER BEFORE EN LA SENTENCIA UPDATE

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

```
DELIMITER //[3]
CREATE TRIGGER cliente_BU_Trigger
BEFORE UPDATE ON cliente FOR EACH ROW
BEGIN
-- La edad es negativa?
IF NEW.edad<0 THEN
SET NEW.edad = NULL;
END IF;
END// DELIMITER;
```

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 asignamos `NULL` a ese campo.

TRIGGER AFTER EN LA SENTENCIA UPDATE

Suponga el siguiente escenario: *Tienda de accesorios para Gamers*. Para la actividad del negocio se ha creado un **sistema de facturación** sencillo, que registra las ventas realizadas dentro de una factura que contiene el detalle de las compras.

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

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

3 Los delimitadores pueden ser // o && o $$.

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* sabemos si algún vendedor "ocioso" esta alterando las facturas, lo que lógicamente sería atentar contra las finanzas del negocio. Cada registro informa el usuario que modificó la tabla `DETALLE_FACTURA` y muestra una descripción sobre los cambios en cada columna.

Trigger BEFORE en la sentencia INSERT

En el siguiente ejemplo se muestra como mantener la integridad de una base de datos con respecto a un atributo derivado.

La *Tienda de electrodomésticos* ha implementado un **sistema de facturación**. En la base de datos que soporta la información del negocio, existen varias tablas, pero nos centramos en la tabla `PEDIDO` y en la tabla `TOTAL_VENTAS`.

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

Pero suponga que elimina el ultimo pedido hecho por este cliente, ¿que pasaría con el registro en TOTAL_VENTAS? quedaría desactualizado.

Usamos tres Triggers para solucionar esta situación. Para que cada vez que use un comando DML en la tabla PEDIDO, no tenga que preocuparse por actualizar manualmente TOTAL_VENTAS.

Solución:

```
-- 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 ellos mantiene el total de ventas de cada cliente actualizado dependiendo del evento realizado sobre un pedido.

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

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

Y si elimina un pedido de un cliente simplemente descuenta del total acumulado el monto que con anterioridad había acumulado.

INFORMACIÓN DE UN TRIGGER

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

SHOW CREATE TRIGGER futbolista_ai_trigger;

También ve los Triggers que hay en su base de datos con:

SHOW TRIGGERS;

ELIMINAR UN TRIGGER

DROP, **DROP** y más **DROP**. Como ya sabe usamos este comando para eliminar casi cualquier cosa en la base de datos:

DROP TRIGGER [IF EXISTS] nombre_trigger

Recuerde que puede adicionar la condición `IF EXISTS` para indica que si el Trigger ya existe, entonces que lo borre.

Veamos el manejo de triggers mediante sencillas aplicaciones, que consisten en una base de datos con 2 tablas. Una de apuntes contables y otra de saldos mensuales que se van actualizando según vayamos insertando, modificando o eliminado apuntes.

Creamos un archivo de texto llamado EJEMPLO1.SQL Que almacena nuestro código SQL que de ahora en adelante estará en negrita.

- Creamos la base de datos:

```
DROP DATABASE IF EXISTS contable;
CREATE DATABASE contable;
USE contable;
```

- Creamos las tablas apuntes y saldos del tipo InnoDB, con una clave primaria para identificar cada registro de la tabla como único.

```
DROP TABLE IF EXISTS APUNTES;
CREATE TABLE APUNTES (
```

```
ASIENTO INT(8) DEFAULT 0,
LINEA SMALLINT(5) DEFAULT 0,
FECHA DATE DEFAULT '2014-01-01',
TEXTO VARCHAR(40) default '',
CUENTA CHAR(10) default '',
DEBE DOUBLE(10,2) DEFAULT 0,
HABER DOUBLE(10,2) DEFAULT 0,
PRIMARY KEY (ASIENTO,LINEA),
KEY K2(CUENTA, FECHA)) ENGINE=InnoDB ROW_FORMAT=DYNAMIC;
```

- Creamos la tabla Saldos, también con una clave primaria única.

```
DROP TABLE IF EXISTS SALDO;
CREATE TABLE SALDO (
CUENTA CHAR(10) NOT NULL default '',
ANO SMALLINT(4) DEFAULT 0,
MES TINYINT(2) DEFAULT 0,
DEBE DOUBLE(10,2) DEFAULT 0,
HABER DOUBLE(10,2) DEFAULT 0,
PRIMARY KEY (CUENTA,ANO,MES))
ENGINE=InnoDB ROW_FORMAT=DYNAMIC;
```

- Procedemos a crear los Triggers, para la tabla apuntes.
- Primero el trigger de inserción de registros. Atención a los delimitadores y al punto y coma de final de sentencia. Remarcar que los saldos se actualizan después de ingresar un registro en la tabla apuntes (AFTER INSERT).

Considere la orden: INSERT INTO … ON DUPLICATE KEY UPDATE.

Esta sentencia inserta un registro en la tabla de saldos, y si éste existe, solamente actualiza las columnas debe y haber. Por eso hemos definido claves primarias

(PRIMARY KEY) en las tablas, requisito indispensable para que esta sentencia trabaje.

```
DELIMITER //
CREATE TRIGGER APTS_I AFTER INSERT ON APUNTES
FOR EACH ROW
BEGIN
INSERT INTO SALDO SET
SALDO.CUENTA=NEW.CUENTA,
SALDO.ANO=YEAR(NEW.FECHA),
SALDO.MES=MONTH(NEW.FECHA),
SALDO.DEBE=NEW.DEBE,
SALDO.HABER=NEW.HABER
ON DUPLICATE KEY UPDATE
SALDO.DEBE=SALDO.DEBE+NEW.DEBE,
SALDO.HABER=SALDO.HABER+NEW.HABER ;
END;//
DELIMITER;
```

- Creamos el trigger de actualización o modificación de asientos:

```
DELIMITER //
CREATE TRIGGER APTS_U AFTER UPDATE ON APUNTES
FOR EACH ROW
BEGIN
INSERT INTO SALDO SET
SALDO.CUENTA=OLD.CUENTA,
SALDO.ANO=YEAR(OLD.FECHA),
SALDO.MES=MONTH(OLD.FECHA),
SALDO.DEBE=OLD.DEBE*(-1),
```

```
SALDO.HABER=OLD.HABER*(-1)
ON DUPLICATE KEY UPDATE
SALDO.DEBE=SALDO.DEBE+(OLD.DEBE*(-1)),
SALDO.HABER=SALDO.HABER+(OLD.HABER*(-1)) ;
INSERT INTO SALDO SET
SALDO.CUENTA=NEW.CUENTA,
SALDO.ANO=YEAR(NEW.FECHA),
SALDO.MES=MONTH(NEW.FECHA),
SALDO.DEBE=NEW.DEBE,
SALDO.HABER=NEW.HABER
ON DUPLICATE KEY UPDATE
SALDO.DEBE=SALDO.DEBE+NEW.DEBE,
SALDO.HABER=SALDO.HABER+NEW.HABER;
END;//
DELIMITER ;
```

- Por último, creamos el trigger de eliminación de la tabla apuntes

```
DELIMITER //
CREATE TRIGGER APTS_D AFTER DELETE ON APUNTES
FOR EACH ROW
BEGIN
INSERT INTO SALDO SET
SALDO.CUENTA=OLD.CUENTA,
SALDO.ANO=YEAR(OLD.FECHA),
SALDO.MES=MONTH(OLD.FECHA),
SALDO.DEBE=OLD.DEBE*(-1),
SALDO.HABER=OLD.HABER*(-1)
ON DUPLICATE KEY UPDATE
```

```
SALDO.DEBE=SALDO.DEBE+(OLD.DEBE*(-1)),
SALDO.HABER=SALDO.HABER+(OLD.HABER*(-1)) ;
END;//
DELIMITER ;
```

- Grabamos el archivo EJEMPLO1.SQL
- Vamos al prompt del MySQL con el usuario root. Para estar seguros de tener los privilegios necesarios, ejecutamos el código.
- Creamos un archivo de código MySQL denominado EJEMPLO2.SQL, que contiene dos (2) asientos contables. Correspondientes a una factura y el cobro de la misma al mes siguiente. Utilizamos transacciones.

```
USE contable;
SET AUTOCOMMIT=0 ;
START TRANSACTION ;
INSERT INTO APUNTES VALUES (1,1,'2014-02-07','Fra.112 PEPE PALO','4300000001',1160,0);
INSERT INTO APUNTES VALUES (1,2,'2014-02-07','Fra.112 PEPE PALO','4770000001',0,160);
INSERT INTO APUNTES VALUES (1,3,'2014-02-07','Fra.112 PEPE PALO','7000000000',0,1000);
INSERT INTO APUNTES VALUES (2,1,'2014-03-20','Cobro Fra.112 PEPE PALO','5700000000',1160,0);
INSERT INTO APUNTES VALUES (2,2,'2014-03-20','Cobro Fra.112 PEPE PALO','4300000001',0,1160);
COMMIT;
```

- Grabamos el archivo EJEMPLO2.SQL

Vamos al prompt de MySQL y ejecutamos el código.

- Ahora, hacer un SELECT de la tabla SALDOS y ver resultado

Otra forma de auditoria sobre una de las tablas de una base de datos es la

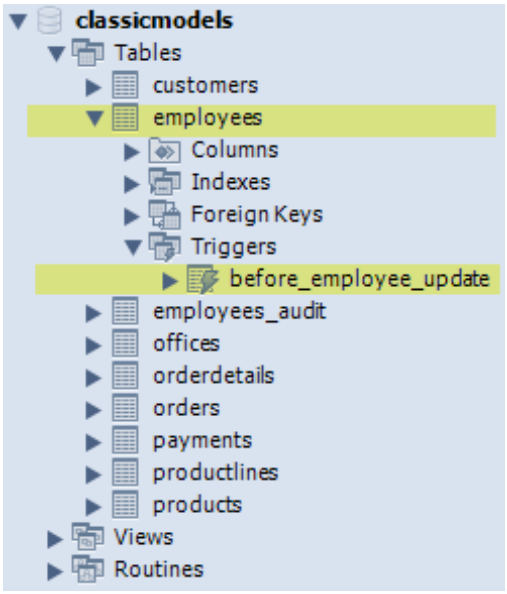
siguiente:

```
CREATE TABLE employees_audit (
id int(11) NOT NULL AUTO_INCREMENT,
employeeNumber int(11) NOT NULL,
lastname varchar(50) NOT NULL,
changedon datetime DEFAULT NULL,
action varchar(50) DEFAULT NULL,
PRIMARY KEY (id)
)
```

Creamos un trigger `BEFORE UPDATE` que es activado antes de que los cambios se realicen en la tabla `employees`.

```
DELIMITER $$
CREATE TRIGGER before_employee_update BEFORE UPDATE ON employees
FOR EACH ROW BEGIN
INSERT INTO employees_audit
SET action = 'update',
employeeNumber = OLD.employeeNumber,
lastname = OLD.lastname,
changedon = NOW();
END$$
DELIMITER;
```

Sí vemos el esquema (schema) de la base de datos, notamos el trigger `before_employee_update` bajo la tabla `empleados` según la siguiente figura:



Actualizamos el registro de un empleado como prueba que el trigger realmente funciona.

```
UPDATE employees
SET lastName = 'Phan'
WHERE employeeNumber = 1056
```

Para verificar que el trigger fue ejecutado por la sentencia UPDATE, hacemos una consulta sobre la tabla `employees_audit`table usando la siguiente query:

```
SELECT * FROM employees_audit
```

Veamos la salida de la consulta:

id	employeeNumber	lastname	changedon	action
1	1056	Phan	2013-01-16 15:59:36	update

El trigger fue realmente ejecutado tal que vemos un nuevo registro en la tabla `employees_audit`.

Hemos explicamos cómo crear triggers o disparadores en MySQL y demostramos qué es un trigger y cómo usarlo en MySQL. Podemos crear triggers para simular un deshacer de una tabla, un trigger que guarda de forma automática los valores de los campos anteriores y los nuevos valores en caso de modificación de un registro.

TIPOS DE TRIGGER

Existen dos tipos de disparadores que se clasifican según la cantidad de ejecuciones a realizar:

- **Row Triggers** (o disparadores de fila): son aquellos que se ejecutaran n-veces si se llaman n-veces desde la tabla asociada al trigger.

- **Statement Triggers** (o disparadores de secuencia): son aquellos que sin importar la cantidad de veces que se cumpla con la condición, su ejecución es única.

En cuanto disponga de la aplicación para ejecutar sentencias SQL y un usuario de MySQL Server con permisos suficientes para crear triggers o disparadores en la base de datos donde queramos, a continuación debemos analizar para qué vamos a usar el trigger, dependiendo de la tarea a realizar necesitaremos, por ejemplo, una tabla auxiliar. En el ejemplo que vamos a usar queremos que mediante un disparador o trigger de MySQL Server se añada un registro a una tabla auxiliar cada vez que un usuario realice una inserción en una de las tablas de MySQL Server. Para ello crearemos la tabla destino del trigger con la sentencia SQL:

```
CREATE TABLE log_accesos(
codigo int(11) NOT NULL AUTO_INCREMENT,
usuario varchar(100),
fecha datetime,
PRIMARY KEY (`codigo`)
)ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
```

En la tabla auxiliar de auditoría almacenamos el nombre del usuario de MySQL

Server y la fecha y hora en la que haya realizado una inserción en una tabla de una base de datos de nuestro servidor. Auditamos la inserción de registros en la tabla "factura".

A continuación creamos el trigger o disparador correspondiente con la sentencia SQL:

```
delimiter $$
CREATE TRIGGER ajsoluciones.tg_auditoria_accesos
BEFORE INSERT ON ajsoluciones.factura
FOR EACH ROW
BEGIN
INSERT INTO ajsoluciones.log_accesos (usuario, fecha) values (CURRENT_USER(), NOW());
END$$
delimiter ;
```

Introducimos la sentencia SQL anterior en MySQL, si todo es correcto (tenemos permisos, existe la tabla "factura", existe la tabla "log_accesos" y existe el catálogo o base de datos "ajsoluciones") el trigger queda almacenado y operativo:

En el caso anterior, el trigger se crea en la base de datos "ajsoluciones", con el nombre "tg_auditoria_accesos" para la tabla "factura" y, a partir de ahora, cuando un usuario cree un registro en la tabla "factura" se crea otro de forma automática en la tabla auxiliar "log_accesos" con el nombre del usuario de MySQL que ha creado el registro y con la fecha y hora en que realizó la inserción.

Para obtener el usuario actual de MySQL usamos la función *CURRENT_USER()* y para obtener la fecha y hora actuales por lo que usamos la función *NOW()*.

La **sintaxis** para crear este trigger es:

```
CREATE
[DEFINER = { user | CURRENT_USER }]
```

TRIGGER trigger_name trigger_time trigger_event

ON tbl_name FOR EACH ROW trigger_body

Otras acciones o tareas a realizar con los triggers o disparadores de MySQL

- Para **consultar** los triggers o disparadores creados en una base de datos ejecutaremos el comando SQL:

  show create triggers;

Muestra un registro por cada trigger creado con los campos: trigger, event, table, statement, timing, created, sql_mode, definer, character_set_client, collation_connection, database_collation. Las opciones importantes son:

- **Trigger**: almacena el nombre del disparador.
- **Event**: indica el tipo de trigger (insert, update, delete).
- **Table**: tabla de la base de datos a la que se asocia el trigger.
- **Statement**: código SQL del trigger.
- **Timing**: tiempo en que se ejecutará el trigger: before (antes), after (después).
- Para **eliminar** un trigger o disparador existente ejecutaremos la siguiente consulta: *drop trigger nombre_trigger*;
- Para **mostrar la consulta SQL** completa de **creación** de un trigger ejecutamos el comando: *show create trigger nombre_trigger;*

ALMACENAMIENTO DE LOS TRIGGER

- los triggers o disparadores se almacenan en la tabla *TRIGGERS* del catálogo del sistema *information_schema*, para verlos ejecute:

  *select * from information_schema.triggers*;

Crear trigger para auditoría de modificaciones en una tabla de una base de datos MySQL

Crear disparador en tabla MySQL para logear todos los cambios de una tabla

Vamos a explicar cómo crear un trigger en MySQL que guarde en una tabla auxiliar las modificaciones realizadas en los campos de una tabla de la base de datos del servidor MySQL. Con este trigger sabemos qué modificaciones se han realizado, con qué usuario, en qué fecha y hora y qué valores había antes de la modificación y después de ella.

Con este caso, implementamos de forma automática el control de cambios de una tabla, si en algún momento implementamos un deshacer o bien recuperar la tabla a un punto anterior en el tiempo.

Realizaremos este trigger para la tabla “factura" con la siguiente estructura:

```
CREATE TABLE factura (
codigo int(10) unsigned NOT NULL AUTO_INCREMENT,
numero varchar(15) DEFAULT NULL,
importetotal float(19,4) DEFAULT NULL,
baseimponible float(19,4) DEFAULT NULL,
porcentajeiva float(19,4) DEFAULT NULL,
importeiva float(19,4) DEFAULT NULL,
porcentajedescuento float(19,4) DEFAULT NULL,
importedescuento float(19,4) DEFAULT NULL,
codigocliente int(10) unsigned NOT NULL DEFAULT '0',
fecha datetime DEFAULT '0000-00-00 00:00:00',
cobrado char(1) DEFAULT NULL,
observacion varchar(255) DEFAULT NULL,
importecobrado float(19,4) DEFAULT NULL,
codusuarioa int(10) unsigned DEFAULT NULL,
codusuariom int(10) unsigned DEFAULT NULL,
fechaa datetime DEFAULT NULL,
fecham datetime DEFAULT NULL,
contabiliza char(1) DEFAULT NULL,
```

```
imprimida char(1) DEFAULT NULL,
enviada char(1) DEFAULT NULL,
fechaenvio datetime DEFAULT NULL,
piefactura text,
fechavencimiento datetime DEFAULT NULL,
serie char(2) NOT NULL DEFAULT '',
PRIMARY KEY (codigo),
UNIQUE KEY Indice_Numero_Factura (numero) USING HASH
)
```

Creamos la tabla auxiliar "factura" con los mismos campos para guardar el valor anterior (añadimos el sufijo "_old") y duplicamos los campos para guardar el nuevo valor (añadimos el sufijo "_new"):

```
CREATE TABLE auditoria_factura (
codigo int(10) unsigned NOT NULL AUTO_INCREMENT,
usuario varchar(100) NOT NULL,
fecha datetime NOT NULL,
numero_old varchar(15),
mportetotal_old float(19,4),
baseimponible_old float(19,4),
porcentajeiva_old float(19,4),
importeiva_old float(19,4),
porcentajedescuento_old float(19,4),
importedescuento_old float(19,4),
codigocliente_old int(10) unsigned,
fecha_old datetime,
cobrado_old char(1),
observacion_old varchar(255),
```

```
importecobrado_old float(19,4),
codusuarioa_old int(10),
codusuariom_old int(10),
fechaa_old datetime,
fecham_old datetime,
contabiliza_old char(1),
imprimida_old char(1),
enviada_old char(1),
fechaenvio_old datetime,
piefactura_old text,
fechavencimiento_old datetime,
serie_old char(2) NOT NULL,
numero_new varchar(15),
importetotal_new float(19,4),
baseimponible_new float(19,4),
porcentajeiva_new float(19,4),
importeiva_new float(19,4),
porcentajedescuento_new float(19,4),
importedescuento_new float(19,4),
codigocliente_new int(10) unsigned,
fecha_new datetime,
cobrado_new char(1),
observacion_new varchar(255),
importecobrado_new float(19,4),
codusuarioa_new int(10) unsigned,
codusuariom_new int(10) unsigned,
fechaa_new datetime,
```

```
fecham_new datetime,
contabiliza_new char(1),
imprimida_new char(1),
enviada_new char(1),
fechaenvio_new datetime,
piefactura_new text,
fechavencimiento_new datetime,
serie_new char(2),
PRIMARY KEY (codigo)
)
```

El código del disparador o trigger completo es:

```
DELIMITER $$
CREATE TRIGGER ajsoluciones.tg_auditoria_factura
AFTER UPDATE ON ajsoluciones.factura
FOR EACH ROW
BEGIN
INSERT INTO ajsoluciones.auditoria_factura
(usuario, fecha, numero_old, importetotal_old,
baseimponible_old, porcentajeiva_old, importeiva_old,
porcentajedescuento_old, importedescuento_old,
codigocliente_old, fecha_old, cobrado_old,
observacion_old, importecobrado_old, codusuarioa_old,
codusuariom_old, fechaa_old, fecham_old, contabiliza_old,
imprimida_old, enviada_old, fechaenvio_old,
piefactura_old, fechavencimiento_old, serie_old,
numero_new, importetotal_new, baseimponible_new,
porcentajeiva_new, importedescuento_new, importeiva_new,
```

```
porcentajedescuento_new, codigocliente_new,
fecha_new, cobrado_new, observacion_new,
importecobrado_new, codusuarioa_new, codusuariom_new,
fechaa_new, fecham_new, contabiliza_new, imprimida_new,
enviada_new, fechaenvio_new, piefactura_new,
fechavencimiento_new, serie_new)
VALUES (CURRENT_USER(), NOW(), OLD.numero, OLD.importetotal,
OLD.baseimponible, OLD.porcentajeiva, OLD.importeiva,
OLD.porcentajedescuento, OLD.importedescuento,
OLD.codigocliente, OLD.fecha, OLD.cobrado,
OLD.observacion, OLD.importecobrado, OLD.codusuarioa,
OLD.codusuariom, OLD.fechaa, OLD.fecham, OLD.contabiliza,
OLD.imprimida, OLD.enviada, OLD.fechaenvio,
OLD.piefactura, OLD.fechavencimiento, OLD.serie,
NEW.numero, NEW.importetotal, NEW.baseimponible,
NEW.porcentajeiva, NEW.importedescuento, NEW.importeiva,
NEW.porcentajedescuento, NEW.codigocliente,
NEW.fecha, NEW.cobrado, NEW.observacion,
NEW.importecobrado, NEW.codusuarioa, NEW.codusuariom,
NEW.fechaa, NEW.fecham, NEW.contabiliza, NEW.imprimida,
NEW.enviada, NEW.fechaenvio, NEW.piefactura,
NEW.fechavencimiento, NEW.serie);
END;
$$
DELIMITER ;
```

A continuación, explicamos el trigger anterior:

1. Por un lado usamos *AFTER UPDATE* para indicar que el trigger se ejecute cada vez que un usuario realice alguna modificación en la tabla *factura* del catálogo o base de datos *ajsoluciones*.
2. El trigger inserta un registro en la tabla auxiliar *ajsoluciones.auditoria_factura*, en dicho registro, el trigger guarda para cada campo el valor anterior y el nuevo valor (si hay modificación). Para ello se usan las cláusulas especiales "**OLD**.nombre_campo" (el trigger obtiene el valor anterior al cambio del campo) y "**NEW**.nombre_campo" (el trigger obtiene el nuevo valor del campo).
3. La función de *CURRENT_USER() obtiene* y almacena en la tabla el usuario actual de MySQL.
4. La función *NOW()* obtiene la fecha y hora en que el usuario realiza el cambio en la tabla.

Cuando el usuario realice un cambio en cualquier campo de la tabla facturas, por ejemplo, si añade a "Observación" el texto "Prueba trigger AjpdSoft" y guarda los cambios:

El trigger establecido para esta tabla crea un registro en la tabla *auditoria_factura* con el valor anterior en el campo "observacion_old" y el valor nuevo en el campo "observacion_new". Para ver esta tabla ejecutaremos la consulta SQL:

```
select fecha, usuario, observacion_old, observacion_new
from ajsoluciones.auditoria_factura
```

El trigger almacena la fecha, la hora, el usuario de MySQL, el valor de los campos anteriores y el valor de los campos modificado.

Con este ejemplo, implementamos una opción muy interesante para logear o auditar los cambios que se produzcan en una tabla MySQL. No se debe abusar de este tipo de trigger pues podría generar tablas con millones de registros y ralentizar la base de datos. Es recomendable usar este tipo de trigger sólo en el caso de tablas importantes para las que se quiera auditar los cambios, vigilando siempre el crecimiento de las tablas auxiliares.

Otras opciones y funcionalidades de los trigger en MySQL

- **Seguridad**: si queremos que un usuario, además del usuario "root" tenga permisos para crear triggers o disparadores en una tabla, ejecutaremos el comando SQL:

  GRANT CREATE TRIGGER ON nombre_tabla TO nombre_usuario

Pero si sólo requerimos un administrador en nuestra aplicación, usamos:

```
DROP TABLE IF EXISTS "usuario";
CREATE TABLE "usuario"(
      "id" Integer PRIMARY KEY AUTO_INCREMENT,
      "login" varchar(10) NOT NULL,
      "clave" varchar(64) NOT NULL,
      "role" enum('Administrador','Usuario'),
      "pregunta" Text NOT NULL,
      "respuesta" Text NOT NULL,
      "status" char(1) enum('A','I'),
      "creado" DateTime DEFAULT current_date,
      "modificado" DateTime,
);

CREATE TRIGGER "actualiza_tiemstamp_usuario"
      AFTER UPDATE
      ON "usuario"
      FOR EACH ROW
BEGIN
UPDATE usuario SET modificado = current_date WHERE id = NEW.id;
END;
```

```
CREATE TRIGGER "unAdmin"
      BEFORE INSERT
      ON "usuario"
      FOR EACH ROW
      WHEN NEW.role = 'Administrador'
BEGIN
SELECT RAISE(ABORT, 'Gracias, registro procesado !!!')
WHERE
EXISTS (SELECT 1 FROM usuario WHERE role = 'Administrador');
END;
```

- **Seguridad**: para dar permisos de creación de triggers para un usuario para todas las tablas ejecutamos el comando SQL:

  *GRANT CREATE TRIGGER ON *.* TO nombre_usuario*

Un **ejemplo** de un disparador o trigger para calcular el importe de comisión de una venta realizada: si tenemos una tabla donde guardamos las ventas realizadas por cada comercial, con la siguiente estructura:

```
CREATE TABLE bdajpdsoft.ventas (
codigo INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
codigocliente INTEGER UNSIGNED NOT NULL,
codigocomercial INTEGER UNSIGNED NOT NULL,
importeventa DECIMAL(9,2),
importecomision DECIMAL(9,2),
PRIMARY KEY (`codigo`)
) ENGINE = InnoDB;
```

Podemos realizar un trigger que, de forma automática, calcule la comisión que corresponde a cada venta, dicha comisión se guardará en el campo "importecomision" y se calcula mediante el siguiente procedimiento almacenado:

```
DELIMITER $$
CREATE PROCEDURE pr_calculo_comision (importe DECIMAL(9,2))
BEGIN
SET @var_global_comision := valor / 5;
END;
$$
DELIMITER ;
```

El trigger para actualizar el valor del campo "importecomision" de forma automática es:

```
DELIMITER $$
CREATE TRIGGER ajsoluciones.tg_actualizar_comision
BEFORE INSERT ON ajsoluciones.ventas
FOR EACH ROW
BEGIN
  CALL ajsoluciones.pr_calculo_comision(NEW.importeventa);
  SET NEW.importecomision = @var_global_comision;
END;
$$
DELIMITER ;
```

TRIGGER CONDICIONADO

Se refiere a deshacer de nuevo a la contraseña anterior (OLD), cuando no se suministra la nueva (NEW).

```
DROP TRIGGER IF EXISTS upd_user;
DELIMITER $$
```

```
CREATE TRIGGER upd_user BEFORE UPDATE ON `user`
FOR EACH ROW BEGIN
IF (NEW.password IS NULL OR NEW.password = '') THEN
SET NEW.password = OLD.password;
ELSE
SET NEW.password = Password(NEW.Password);
END IF;
END$$
DELIMITER ;
```

El trigger muestra que un usuario no puede dejar en blanco su contraseña.

Si el campo contraseña (encriptada) se está enviando de vuelta en la actualización a MySQL, entonces no debe ser nulo o en blanco, y MySQL intentará rehacer la función de contraseña () sobre ella. Para detectar esto, utilice este código en su lugar.

```
DELIMITER $$
CREATE TRIGGER upd_user BEFORE UPDATE ON `user`
FOR EACH ROW BEGIN
IF (NEW.password IS NULL OR NEW.password = '' OR NEW.password =
OLD.password) THEN
SET NEW.password = OLD.password;
ELSE
SET NEW.password = Password(NEW.Password);
END IF;
END$$
DELIMITER ;
```

```
delimiter $$
CREATE TRIGGER attendance_clock
AFTER INSERT ON alldevicerecord
FOR EACH ROW BEGIN
DECLARE `check` INT;
SET `check` = 1;
IF (`check` % 2 = 0) THEN
INSERT into designation(emp_id, clock_date,clock_in) VALUES (new.id, new.time, new.time);
set `check` = `check` + 1;
ELSE
UPDATE designation SET clock_out = new.time WHERE emp_id = NEW.id;
set `check` = `check` + 1;
END IF;
END;
$$
delimiter //
```

Veamos otro caso, donde la cantidad (amount) ingresada sale de los límites, debe estar entre 0 y 100.

```
CREATE TRIGGER upd_check BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
IF NEW.amount < 0 THEN
SET NEW.amount = 0;
ELSEIF NEW.amount > 100 THEN
```

```
SET NEW.amount = 100;
END IF;
END;//
delimiter ;
```

¿CÓMO CONSTRUIR UN TRIGGER CONDICIONAL?

Escribamos un trigger para dos tablas, como las siguientes:

```
Table A-----------------------------Table B

order_id--------sku---------------order_id----order_#----sku_copy

    568---------AAA---------------568---------2345
    567---------BBB---------------567---------6789-------empty column
    566---------CCC---------------566---------1234
```

Cuando un cliente realiza una compra se añade un nuevo registro a cada tabla. He añadido la columna ' sku_copy ' a la Tabla B, que no consigue llenar cuando se crea un nuevo registro.

Cuando se crea un nuevo registro, requiero que el trigger copie el campo 'Código' en la Tabla A para el campo 'sku_copy' en la Tabla B. Sin embargo, el problema es cómo estructurar la siguiente condición en el trigger.

SI: 'order_id' de la tabla A coincide con 'order_id' en la Tabla B. ENTONCES: copia “sku" del registro de la tabla A al registro en la tabla B con el 'ORDER_ID' coincidente. Los datos deben ser añadidos en 'sku_copy' de la Tabla B.

Estoy utilizando el siguiente trigger SQL pero da error cuando se ejecuta :

"# 1363 - No hay fila OLD en el trigger INSERT "

Aquí está el código.

```
DELIMITER $$
CREATE TRIGGER trigger_name
AFTER INSERT ON tableA
```

```
FOR EACH ROW BEGIN
INSERT INTO tableB
SET sku_copy = OLD.sku,
order_id = OLD.order_id,
order = OLD.order;
END $$
DELIMITER ;
```

¿Cómo corregir el código? He aquí la solución:

```
DELIMITER $$
CREATE TRIGGER `sku_after_update` AFTER UPDATE ON `uau3h_virtuemart_order_items`
FOR EACH ROW
BEGIN
IF (old.order_item_sku_copy != new.order_item_sku)
THEN
UPDATE uau3h_virtuemart_orders
SET order_item_sku_copy=new.order_item_sku,
WHERE virtuemart_order_id=new.virtuemart_order_id;
END IF;
END$$
DELIMITER ;
```

MySQL Schema Setup:

```
CREATE TABLE TableA(order_id INT, sku VARCHAR(10));
CREATE TABLE TableB(order_id INT, order_no VARCHAR(10),sku_copy VARCHAR(10));
```

vamos a:

```
CREATE TRIGGER trigger_name
    AFTER INSERT ON TableA
    FOR EACH ROW BEGIN

    UPDATE TableB
    SET sku_copy = NEW.sku
    WHERE order_id = NEW.order_id;
END;
```

vamos a:

```
INSERT INTO TableB(order_id, order_no)VALUES(1,'111');
INSERT INTO TableB(order_id, order_no)VALUES(2,'222');
INSERT INTO TableB(order_id, order_no)VALUES(3,'333');
```

vamos a:

```
INSERT INTO TableA(order_id, sku)VALUES(1,'AAA'),(2,'BBB');
```

Query 1:

```
SELECT * FROM TableB;
```

Resultados:

```
| ORDER_ID | ORDER_NO | SKU_COPY |
-----------------------------------
|        1 |      111 |      AAA |
|        2 |      222 |      BBB |
|        3 |      333 |   (null) |
```

MySQL Schema Setup:

```
CREATE TABLE TableA(order_id INT, sku VARCHAR(10));


CREATE TABLE TableB(order_id INT, order_no
VARCHAR(10),sku_copy VARCHAR(10));


vamos a
CREATE TRIGGER TableA_AfterInsert
    AFTER INSERT ON TableA
    FOR EACH ROW BEGIN

    UPDATE TableB
    SET sku_copy = NEW.sku
    WHERE order_id = NEW.order_id;
END;
vamos a
INSERT INTO TableB(order_id, order_no)VALUES(1,'111');
INSERT INTO TableB(order_id, order_no)VALUES(2,'222');
INSERT INTO TableB(order_id, order_no)VALUES(3,'333');
vamos a
INSERT INTO TableA(order_id, sku)VALUES(1,'AAA'),
(2,'BBB');
vamos a
CREATE TRIGGER TableA_AfterUpdate
```

```
    AFTER UPDATE ON TableA
    FOR EACH ROW BEGIN

    IF (OLD.sku != NEW.sku)
    THEN
      UPDATE TableB
      SET sku_copy = NEW.sku
      WHERE order_id = NEW.order_id;
    END IF;
END;
vamos a
UPDATE TableA
  SET sku = 'NEW'
WHERE order_id = 2;
vamos a
```

Query 1:

```
SELECT * FROM TableB;
```

Resultados:

```
| ORDER_ID | ORDER_NO | SKU_COPY |
-----------------------------------
|        1 |      111 |      AAA |
|        2 |      222 |      NEW |
|        3 |      333 |   (null) |
```

En ambos casos, las tablas virtuales `NEW` y `OLD` se refieren a la tabla donde

está asociado el trigger. `NEW` contiene la nueva versión del registro que fue ingresado o actualizado; mientras que `OLD` contiene la versión precambio del registro `OLD` que sólo está asociado en un trigger UPDATE ya que no hay una versión anterior (update) sobre la inserción (insert).

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:

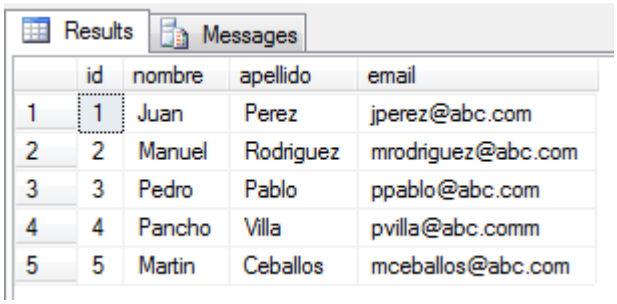
1. crear una tabla llamada estudiantes
2. guardar 5 estudiantes en la tabla
3. crear un procedimiento almacenado que sólo tenga un parámetro de entrada
4. crear un procedimiento almacenado que tenga un parámetro de entrada y otro de salida

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

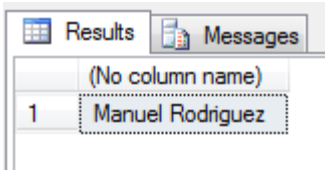
	id	nombre	apellido	email
1	1	Juan	Perez	jperez@abc.com
2	2	Manuel	Rodriguez	mrodriguez@abc.com
3	3	Pedro	Pablo	ppablo@abc.com
4	4	Pancho	Villa	pvilla@abc.comm
5	5	Martin	Ceballos	mceballos@abc.com

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.

EJERCICIOS

Vamos a trabajar con triggers:

a. Creamos una base de datos llamada ejemplo y le decimos que queremos trabajar con ella.

b. Crearemos 2 tablas, en una guardamos datos de las personas y en la otra cuando se ha introducido cada dato.

Persona(codigo, nombre, edad) y Nuevosdatos(codigo,cuando,tipo)

c. Crear un trigger para que se añada un dato en la segunda tabla cada vez que insertemos en la primera. El trigger saltará con un AFTER INSERT, los datos que introduzcamos serán: el código de la persona, la fecha actual y la letra "i" para indicar que el cambio ha sido la inserción de un dato nuevo. Cada vez que insertemos datos, debemos realizar una consulta para que nos muestre los datos insertados.

d. Crear un trigger que cuente los registros de la tabla employees y mantenga ese numero actualizado en una tabla llamada cuenta, este trigger actuara después de una instrucción insert o delete ejemplo si se introduce un registro nuevo en la tabla aumentara en uno el contador, si se elimina un registro se disminuirá en uno el contador.

e. Crear un trigger que cada vez que se inserte un producto, en la tabla productos, este vaya contando en otra tabla el numero de productos insertados y otro trigger para que cada vez que se elimine vaya descontando.

f. Crear un trigger que cada vez que se elimine almacene la información del producto, ver ejercicio e, en otra tabla con la fecha y la hora de la eliminación.

g. Compra y venta Cree una tabla llamada productos(puede utilizar la tabla del caso e) Cree una tabla ventas (codventa, codigodelproducto,fecha, valorunitario, cantidad) Cree una tabla compras (codcompra, codigodelproducto, fecha, valorunitario, cantidad). Cree un trigger que cada vez que se inserte una compra

actualice (aumente) la cantidad en la tabla producto. Cree un trigger que cada vez que se inserte una venta actualice (disminuya) la cantidad en la tabla producto.

h. Auditoria.

a. Cree una tabla clientes con los siguientes campos (documento, nombre, apellido, saldo).

b. Cree una tabla pagos (código, codcliente, fecha, valor).

c. Cree una tabla créditos (código, codcliente, fecha, valor).

d. Cree un trigger que antes de insertar un crédito aumente el saldo del cliente.

e. Cree un trigger que antes de insertar un pago disminuya el saldo del cliente.

f. Cree una trigger que cada vez que se inserte, se actualice o elimine en cualquiera de las tres tablas, guarde información en una tabla que se llame auditoria con los siguientes campos(Evento,nombredetabla, código, fecha, hora) Evento: inserción, actualización o eliminación Nombredelatabla: la tabla que será afectada codigo: la llave primaria de lo que se está afectando. Fecha y hora: son tomados del sistema.

Funciones

Cree dos tablas así:

tabla 1 CASA

Código

descripción

nrcuarto

Largo

Ancho

Area

Valor

Perímetro

tabla 2 valormetro

Valor

1. (20%) Cree una función que calcule el área de la casa teniendo en cuenta que es el área en metros cuadrado de la casa.

2. (20%) Cree una función que calcule el perímetro de la casa.

3. (20%) Cree una función que calcule el valor de la casa teniendo en cuenta que recibe tres parámetros el largo, el ancho y el valor del metro cuadrado. El valor es el área por el valor del metro cuadrado. (10%) Llame al método que calcula el área.

4. (30%) Construya un procedimiento que almacene en la tabla casa ya con la información calculada (llame a las funciones necesarias). Tenga en cuenta lo siguiente para llamar la función que calcula el valor de la casa primero debe buscar el valor guardado en valor metro.

Procedimientos

Dada la siguiente tabla persona (peso, estado) Realice un procedimiento para determinar si la persona puede donar sangre, si el peso el menor a 50 guarde en estado “no admitido”, en caso contrario seria “admitido”. Dada la siguiente tabla clientes (cedula, nombre, apellido) Cree procedimientos para los siguientes casos

1. Que inserte información en la tabla clientes. Ayuda(recibe tres parámetros de entrada)

2. Que actualice el nombre de un cliente. Ayuda (recibe dos parámetros, numero de cedula de quien se va actualizar y el nuevo nombre)

3. Que elimine un cliente. Ayuda (recibe un parámetro, numero de cedula de quien se va eliminar.

4. Investigar procedimientos con paramentaros de salida.

5. Investigar cómo hacer un ciclo (while).

REFERENCIAS

[1] Domínguez Ch. Jorge. (2014) Breve Introducción al SQL, editorial IEASS, Venezuela.

http://lucas.hispalinux.es/Postgresqles/web/navegable/programmer/triggers.html

http://dev.mysql.com/doc/refman/5.0/es/server-side-scripts.html

http://dev.mysql.com/doc/refman/5.0/es/instance-manager.html

http://dev.mysql.com/doc/refman/5.0/es/configuring-mysql.html

http://dev.mysql.com/doc/refman/5.0/es/server-shutdown.html

http://dev.mysql.com/doc/refman/5.0/es/security.html

http://dev.mysql.com/doc/refman/5.0/es/mysql.html

http://dev.mysql.com/doc/refman/5.0/es/mysqladmin.html

http://dev.mysql.com/doc/refman/5.0/es/features.html

http://dev.mysql.com/doc/refman/5.0/es/log-files.html

http://dev.mysql.com/doc/refman/5.0/es/user-account-management.html

http://dev.mysql.com/doc/refman/5.0/es/account-management-sql.html

http://dev.mysql.com/doc/refman/5.0/es/privilege-system.html

http://dev.mysql.com/doc/refman/5.0/es/encryption-functions.html

http://dev.mysql.com/doc/refman/5.0/es/storage-engines.html

http://dev.mysql.com/doc/refman/5.0/es/myisam-storage-engine.html

http://dev.mysql.com/doc/refman/5.0/es/innodb.html

http://dev.mysql.com/doc/refman/5.0/es/data-manipulation.html

http://dev.mysql.com/doc/refman/5.0/es/examples.html

http://dev.mysql.com/doc/refman/5.0/es/query-speed.html

http://dev.mysql.com/doc/refman/5.0/es/disaster-prevention.html

http://dev.mysql.com/doc/refman/5.0/es/mysqlhotcopy.html

http://dev.mysql.com/doc/refman/5.0/es/mysqldump.html

http://dev.mysql.com/doc/refman/5.0/es/mysqlbinlog.html

http://dev.mysql.com/doc/refman/5.0/es/myisampack.html

http://dev.mysql.com/doc/refman/5.0/es/load-data.html

http://dev.mysql.com/doc/refman/5.0/es/select.html

http://dev.mysql.com/doc/refman/5.0/es/mysqlimport.htmlanexos

http://dev.mysql.com/doc/refman/5.0/es/check-table.html

http://dev.mysql.com/doc/refman/5.0/es/repair-table.html

http://dev.mysql.com/doc/refman/5.0/en/myisamchk.html

http://dev.mysql.com/doc/refman/5.0/es/using-mysqlcheck.html

http://dev.mysql.com/doc/refman/5.0/es/query-cache.html

http://www.xtec.net/~acastan/textos/Tuning%20LAMP.pdf

http://dev.mysql.com/doc/refman/5.0/es/optimizing-the-server.html

http://dev.mysql.com/doc/refman/5.0/es/ndbcluster.html

http://dev.mysql.com/doc/refman/5.0/es/connectors.html

http://dev.mysql.com/doc/refman/5.0/es/replication.html

http://dev.mysql.com/doc/refman/5.0/es/replication-sql.html

http://dev.mysql.com/books/hpmysql-excerpts/ch07.html