TEMA 11: CREACIÓN DE GUIONES.

OBJETIVOS:

- Utilizar medidas para mantener la integridad y consistencia de la información
- Diseñar guiones de sentencias para llevar a cabo tareas complejas.
- Anular parcial o totalmente los cambios producidos por una transacción.

Introducción a la programación de bases de datos

Introducción a la programación de bases de datos

Programar una base de datos es la forma en la que el desarrollador o administrador de la base de datos interactúa con ella.

Hay muchas maneras de programar una base de datos, la elección del modo más adecuado depende de con qué tecnología de base de datos se está trabajando (por ejemplo, Oracle), de qué compiladores se tengan instalados en la máquina objeto de ejecución (por ejemplo, *C++)*, de qué ventajas e inconvenientes se persigan, o de la naturaleza de la aplicación de base de datos que se va a desarrollar. La aplicación de base de datos puede ser de tres tipos distintos:

- Codificación en el lado del servidor: La lógica de la aplicación reside enteramente en la base de datos. La aplicación está basada en la implementación de disparadores que se ejecutan automáticamente cuando ocurre algún cambio en los datos almacenados, y en el almacenamiento de procedimientos y funciones que son llamados explícitamente. Una ventaja importante es que, de esta forma, se puede reutilizar el mismo código para muchos clientes.
- Modelo cliente/servidor (también llamado modelo en dos niveles):
  El código de la aplicación corre en otra máquina distinta del servidor donde se encuentra la base de datos. Las llamadas a la base de datos son transmitidas desde la máquina cliente al servidor. Los datos son transmitidos desde el cliente al servidor para las operaciones de inserción y actualización, y desde el servidor al cliente durante las consultas. Los datos son procesados en la máquina cliente. Normalmente, estas aplicaciones son escritas usando precompiladores con las sentencias SQL embebidas en el código.
- Modelo en tres niveles: Al modelo en dos niveles se le añade un servidor de aplicación separado que procesa las peticiones. Éste podría ser, por ejemplo, un servidor Web básico, o un servidor que realizara funciones de cacheo de datos y balanceo de carga. Aumentar la potencia de procesamiento de esta capa intermedia permite disminuir los recursos necesarios para los clientes, pudiendo estos llegar a ser simples navegadores Web.

Para el desarrollo de aplicaciones, además de necesitar un interfaz de programación donde hacer el desarrollo, se precisa de un software de cliente que enlace el código desarrollado con la base de datos. Los interfaces más comunes son:

- SQL incorporado: Las aplicaciones de bases de datos de SQL incorporado se conectan a las bases de datos y ejecutan directamente sentencias de SQL incorporado que se encuentran insertadas dentro de una aplicación escrita en un lenguaje principal (típicamente C, C++ o COBOL). Las sentencias SQL se pueden ejecutar estática o dinámicamente.
- **ODBC:** Microsoft desarrolló una interfaz SQL denominada **Open Database Connectivity** (ODBC) para los sistemas operativos de Microsoft. Los controladores ODBC específicos de la BBDD se cargan dinámicamente en el tiempo de ejecución, mediante un gestor de controladores basado en la fuente de datos (nombre de la base de datos) proporcionada en la petición de conexión. La aplicación se enlaza directamente con una única biblioteca del gestor de controladores, en lugar de con la biblioteca de cada motor de base de datos. El gestor de controladores media entre las llamadas de función de la aplicación en el momento de la ejecución y asegura que éstas se dirijan hasta el

controlador ODBC adecuado que sea específico para el motor. Dado que el gestor del controlador ODBC sólo conoce las funciones específicas de ODBC, no se podrá acceder a las funciones específicas del motor de base de datos. ODBC no está limitado a los sistemas operativos de Microsoft; otras implementaciones están disponibles en varias plataformas. Para el desarrollo de aplicaciones ODBC, debe instalarse un ODBC Software Development Kit.

- **CLI:** Call Level Interface (interfaz a nivel de llamada). Es una interfaz de programación de aplicaciones C y C++ para el acceso a bases de datos relacionales, que utiliza llamadas de función para pasar sentencias de SQL dinámico como argumentos de función. Es una alternativa al SQL dinámico incorporado, pero a diferencia de SQL incorporado, CLI de DB2 no necesita variables del sistema principal o precompilador. Se basa en la especificación ODBC 3,51 y en la Norma Internacional para SQL/CLI. Estas especificaciones se escogieron como la base para proporcionar una curva de aprendizaje más corta para aquellos programadores de aplicaciones, ya familiarizados con cualquiera de estas interfaces de bases de datos .

- **JDBC:** Java Database Connectivity es un API (Applications Programming Interface, interfaz de programación de aplicaciones) que permite enviar sentencias SQL a una base de datos relacional.

- **SQLJ:** Es un estándar ANSI SQL-1999 para incorporar sentencias SQL en código fuente Java. Proporciona una alternativa al JDBC para el acceso a datos desde Java tanto en el lado cliente como en el lado servidor. Un código fuente SQLJ es más breve que su equivalente en código fuente JDBC.

- OCI y OCCI: Oracle Call Interface (OCI) y Oracle *C++* Call Interface (OCCI) son APIs que permiten crear aplicaciones que invocan procedimientos y funciones nativas para acceder y controlar todas las fases de ejecución de sentencias SQL en bases de datos Oracle. Permite desarrollar aplicaciones combinando la potencia de acceso a los datos del SQL, con la capacidad de hacer bucles, crear estructuras de control. ..

- Interfaz de base de datos Perl: Combina la potencia del lenguaje interpretado Perl y el SQL dinámico. Estas propiedades convierten a Perl en un lenguaje perfecto para crear y revisar rápidamente aplicaciones de bases de datos (DB2). El Módulo DBI de Perl utiliza una interfaz que es muy parecida a las interfaces CLI y JDBC, lo cual facilita la traducción de las aplicaciones Perl a aplicaciones CLI y JDBC, y viceversa.

- Hypertext Preprocessor (PHP): Es un lenguaje de programación interpretado. La primera versión de PHP fue creada por Rasmus Lerdorf y recibió contribuciones bajo una licencia de código abierto en 1995. Principalmente pensado para el desarrollo de aplicaciones Web, inicialmente era un motor de plantillas HTML muy sencillo, pero con el tiempo los desarrolladores de PHP han ido añadiendo funciones de acceso a bases de datos, han reescrito el intérprete, han incorporado soporte orientado a objetos y han mejorado el rendimiento. Actualmente, PHP se ha convertido en un lenguaje muy utilizado para el desarrollo de aplicaciones Web porque se centra en soluciones prácticas y da soporte a las funciones más utilizadas en aplicaciones Web.

- OLE DB: Microsoft OLE DB es un conjunto de interfaces OLE/COM que proporciona a las aplicaciones un acceso uniforme a datos almacenados en distintas fuentes de información. La arquitectura OLE DB define a los consumidores de OLE DB y a los

proveedores de OLE DB. Un consumidor de OLE DB puede ser cualquier sistema o aplicación que utiliza interfaces OLE DB; un proveedor de OLE DB es un componente que expone las interfaces OLE DB.

PROCEDIMIENTOS ALMACENADOS EN MYSQL

Mysql desde la version 5.0 soporta procedimientos almacenados

MySQL se puso las pilas y en su versión 5 implementó muchas de estas características, dejando un SGBD muy rápido y además muy bien preparado para implementar bases de datos realmente grandes y mantenibles.

Pero, ***¿qué es realmente un procedimiento almacenado? .*** Pues es un programa que se almacena físicamente en una tabla dentro del sistema de bases de datos. Este programa está hecho con un lenguaje propio de cada Gestor de BD y esta compilado, por lo que la velocidad de ejecución será muy 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 medio 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 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 engorro, sobre todo si no tienes tiempo.

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

Algunas situaciones donde los procedimientos almacenados pueden ser en particular útiles:

• Cuando aplicaciones de cliente múltiples son escritas en lenguas diferentes o trabajo en plataformas diferentes, pero tienen que realizar las mismas operaciones de base de datos.

• Cuando la seguridad es suprema. Los bancos, por ejemplo, usan procedimientos almacenados para todas las operaciones comunes. Este proporciona un ambiente consecuente y seguro, y los procedimientos pueden asegurar que cada operación es correctamente registrada. En tal sistema, las aplicaciones y los usuarios no conseguirían ningún acceso a las TABLAS de base de datos directamente, pero sólo pueden ejecutar procedimientos almacenados específicos.

Los procedimientos almacenados pueden mejorar el rendimiento ya que se necesita enviar menos información entre el servidor y el cliente. El intercambio que hay es que aumenta la carga del servidor de la base de datos ya que la mayoría del trabajo se realiza en la parte del servidor y no en el cliente. Considere esto sí muchas máquinas cliente (como servidores Web) se sirven a sólo uno o pocos servidores de bases de datos.

Los procedimientos almacenados le permiten tener bibliotecas o funciones en el servidor de base de datos. Esta característica es compartida por los lenguajes de programación modernos que permiten este diseño interno, por ejemplo, usando clases. Usando estas características del lenguaje de programación cliente es beneficioso para el programador incluso fuera del entorno de la base de datos.

MySQL sigue la sintaxis SQL:2003 para procedimientos almacenados

CREACIÓN DE PROCEDIMIENTOS ALMACENADOS Y FUNCIONES

Para crear una función o un procedimiento se deben usar las instrucciones CREATE FUNCTION o CREATE PROCEDURE.

Los procedimientos almacenados y rutinas se crean con comandos **CREATE PROCEDURE** y **CREATE FUNCTION**. Una rutina es un procedimiento o una función. 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.

Desde MySQL 5.0.1, los procedimientos almacenados o funciones se asocian con una base de datos. Esto tiene varias implicaciones:

- Cuando se invoca la rutina, se realiza implícitamente **USE** *db_name* (y se deshace cuando acaba la rutina). Los comandos **USE** dentro de procedimientos almacenados no se permiten.
- Puede calificar los nombres de rutina con el nombre de la base de datos. Esto puede usarse para referirse a una rutina que no esté en la base de datos actual. Por ejemplo, para invocar procedimientos almacenados **p** o funciones **f** esto se asocia con la base de datos **test** , puede decir **CALL test.p()** o **test.f()**.
- Cuando se borra una base de datos, todos los procedimientos almacenados asociados con ella también se borran.

MySQL soporta la extensión muy útil que permite el uso de comandos regulares **SELECT** dentro de los procedimientos almacenados. El conjunto de resultados de estas consultas se envía directamente al cliente. Comandos **SELECT** múltiples generan varios conjuntos de resultados, así que el cliente debe usar una biblioteca cliente de MySQL que soporte conjuntos de resultados múltiples.

SINTAXIS CREATE PROCEDURE y CREATE FUNCTION

```
CREATE PROCEDURE sp_name ([parameter[,...]])
    [characteristic ...] routine_body

CREATE FUNCTION sp_name ([parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MySQL data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

routine_body:
    procedimientos almacenados o comandos SQL válidos
```

.

EL SGBD (MysqlWorkbench, Navicat, etc) no puede interpretar la diferencia entre una consulta normal, y la ejecución de un procedimiento almacenado, función o trigger. Por eso debemos añadir un delimitador ($$, //, ||) que nos permita indicar el inicio y final de la sintaxis del procedimiento almacenado. Entonces la sintáxis iniciaria así:

[**DELIMITER**][espacio][$$ ó //]

se desaconseja el uso del operador "**OR**" [||] como signo delimitador.

Por defecto, la rutina se asocia con la base de datos actual. Para asociar la rutina explícitamente con una base de datos, especifique el nombre como *db_name.sp_name* al crearlo.

Si el nombre de rutina es el mismo que el nombre de una función de SQL, necesita usar un espacio entre el nombre y el siguiente paréntesis al definir la rutina, o hay un error de sintaxis. Esto también es cierto cuando invoca la rutina posteriormente.

La cláusula **RETURNS** puede especificarse sólo con **FUNCTION**, donde es obligatorio. Se usa para indicar el tipo de retorno de la función, y el cuerpo de la función debe contener un comando **RETURN value**.

La lista de parámetros entre paréntesis debe estar siempre presente. Si no hay parámetros, se debe usar una lista de parámetros vacía **()** . Cada parámetro es un parámetro **IN** por defecto. Para especificar otro tipo de parámetro, use la palabra clave **OUT** o **INOUT** antes del nombre del parámetro. Especificando **IN**, **OUT**, o **INOUT** sólo es válido para una **PROCEDURE**.

Los parámetros pueden ser de salida **OUT** y/o de Entrada **IN**, por defecto si no lo especificamos será **IN**.

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.

Un procedimiento o función se considera “determinista” si siempre produce el mismo resultado para los mismos parámetros de entrada, y “no determinista” en cualquier otro caso. Si no se da ni **DETERMINISTIC** ni **NOT DETERMINISTIC** por defecto es **NOT DETERMINISTIC**.

Varias características proporcionan información sobre la naturaleza de los datos usados por la rutina. **CONTAINS SQL** indica que la rutina no contiene comandos que leen o escriben datos. **NO SQL** indica que la rutina no contiene comandos SQL . **READS SQL DATA** indica que la rutina contiene comandos que leen datos, pero no comandos que escriben datos. **MODIFIES SQL DATA** indica que la rutina contiene comandos que pueden escribir datos. **CONTAINS SQL** es el valor por defecto si no se dan explícitamente ninguna de estas características.

La característica **SQL SECURITY** puede usarse para especificar si la rutina debe ser ejecutada usando los permisos del usuario que crea la rutina o el usuario que la invoca. El valor por defecto es **DEFINER**. Esta característica es nueva en SQL:2003. El creador o el invocador deben tener permisos para acceder a la base de datos con la que la rutina está asociada. Desde MySQL 5.0.3, es necesario tener el permiso **EXECUTE** para ser capaz de ejecutar la rutina. El usuario que debe tener este permiso es el definidor o el invocador, en función de cómo la característica **SQL SECURITY** .

MySQL almacena la variable de sistema **sql_mode** que está en efecto cuando se crea la rutina, y siempre ejecuta la rutina con esta inicialización.

La cláusula **COMMENT** es una extensión de MySQL, y puede usarse para describir el procedimiento almacenado. Esta información se muestra con los comandos **SHOW CREATE PROCEDURE** y **SHOW CREATE FUNCTION**.

MySQL permite a las rutinas que contengan comandos DDL (tales como **CREATE** y **DROP**) y comandos de transacción SQL (como **COMMIT**). Esto no lo requiere el estándar, y por lo tanto, es específico de la implementación.

Los procedimientos almacenados no pueden usar **LOAD DATA INFILE**.

Nota: Actualmente, los procedimientos almacenados creados con **CREATE FUNCTION** no pueden tener referencias a tablas. (Esto puede incluir algunos comandos **SET** que pueden contener referencias a tablas, por ejemplo **SET a:= (SELECT MAX(id) FROM t)**, y por otra parte no pueden contener comandos **SELECT** , por ejemplo **SELECT 'Hello world!' INTO var1**.) Esta limitación se el minará en breve.

Los comandos que retornan un conjunto de resultados no pueden usarse desde una función almacenada. Esto incluye comandos **SELECT** que no usan **INTO** para tratar valores de columnas en variables, comandos **SHOW** y otros comandos como **EXPLAIN**. Para comandos que pueden determinarse al definir la función para que retornen un conjunto de resultados, aparece un mensaje de error **Not allowed to return a result set from a function** (**ER_SP_NO_RETSET_IN_FUNC**). Para comandos que puede determinarse sólo en tiempo de ejecución si retornan un conjunto de resultados, aparece el error **PROCEDURE %s can't return a result set in the given context** (**ER_SP_BADSELECT**).

El siguiente es un ejemplo de un procedimiento almacenado que use un parámetro **OUT**. El ejemplo usa el cliente `mysql` y el comando **delimiter** para cambiar el delimitador del comando de; a **//** mientras se define el procedimiento. Esto permite pasar el delimitador; usado en el cuerpo del procedimiento a través del servidor en lugar de ser interpretado por el mismo `mysql`.

```
mysql> delimiter //

mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
    -> BEGIN
    ->   SELECT COUNT(*) INTO param1 FROM t;
    -> END
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> CALL simpleproc(@a);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @a;
+------+
| @a   |
+------+
| 3    |
+------+
1 row in set (0.00 sec)
```

Al usar el comando **delimiter**, debe evitar el uso de la antibarra ('****') ya que es el carácter de escape de MySQL.

El siguiente es un ejemplo de función que toma un parámetro, realiza una operación con una función SQL, y retorna el resultado:

```
mysql> delimiter //

mysql> CREATE FUNCTION hello (s CHAR(20)) RETURNS CHAR(50)
    -> RETURN CONCAT('Hello, ',s,'!');
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

EJEMPLO DEL USO DE UN PROCEDIMIENTO ALMACENADO

Ejemplo de un SELECT dentro de un Procedimiento Almacenado.

```
USE bdtienda;


DELIMITER $$ -- inicio

DROP PROCEDURE IF EXISTS sp_productoPorCod$$ -- eliminamos
si existe un procedimiento con el mismo nombre

CREATE  PROCEDURE sp_productoPorCod (IN cod INT) -- creamos
el procedimiento con un parámetro de entrada

BEGIN  -- inicio cuerpo procedimiento almacenado

    DECLARE estadoOfert CHAR(2);  -- declaramos  una
variable local para almacenar el estado de Oferta.

    /* Hacemos una consulta y el resultado lo almacenamos
en la variable declarada*/

    SELECT oferta INTO estadoOfert FROM productos WHERE
oferta = 'SI' AND codproducto = cod;

    IF estadoOfert = 'SI' THEN -- si está en oferta
elegimos precio_oferta

        SELECT codproducto, nombreproduc, precio_oferta
FROM productos WHERE codproducto = cod;

    ELSE -- sino el precio_normal

        SELECT codproducto, nombreproduc,  precio_normal
FROM productos WHERE codproducto = cod;

    END IF;

END $$  -- fin de cuerpo del procedimiento almacenado

DELIMITER ; -- fin


call sp_productoPorCod(2); -- llamamos al procedimiento
```

ALTER PROCEDURE y ALTER FUNCTION

```
ALTER {PROCEDURE | FUNCTION} sp_name [characteristic ...]

characteristic:
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'
```

Este comando puede usarse para cambiar las características de un procedimiento o función almacenada. Debe tener el permiso **ALTER ROUTINE** para la rutina desde MySQL 5.0.3.

DROP PROCEDURE y DROP FUNCTION

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Este comando se usa para borrar un procedimiento o función almacenado. Esto es, la rutina especificada se borra del servidor. Debe tener el permiso **ALTER ROUTINE** para las rutinas desde MySQL 5.0.3. Este permiso se otorga automáticamente al creador de la rutina.

La cláusula **IF EXISTS** es una extensión de MySQL . Evita que ocurra un error si la función o procedimiento no existe.

SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name
```

Este comando es una extensión de MySQL . Similar a **SHOW CREATE TABLE**, retorna la cadena exacta que puede usarse para recrear la rutina nombrada.

```
mysql> SHOW CREATE FUNCTION test.hello\G
*************************** 1. row ***************************
       Function: hello
       sql_mode:
Create Function: CREATE FUNCTION `test`.`hello`(s CHAR(20)) RETURNS CHAR(50)
RETURN CONCAT('Hello, ',s,'!')
```

SHOW PROCEDURE STATUS y SHOW FUNCTION STATUS

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

Este comando es una extensión de MySQL . Retorna características de rutinas, como el nombre de la base de datos, nombre, tipo, creador y fechas de creación y modificación. Si no se especifica un patrón, le lista la información para todos los procedimientos almacenados, en función del comando que use.

```
mysql> SHOW FUNCTION STATUS LIKE 'hello'\G
```

```
*************************** 1. row ***************************
           Db: test
         Name: hello
         Type: FUNCTION
      Definer: testuser@localhost
     Modified: 2004-08-03 15:29:37
      Created: 2004-08-03 15:29:37
Security_type: DEFINER
      Comment:
```

La sentencia `CALL`

```
CALL sp_name([parameter[,...]])
```

El comando **`CALL`** invoca un procedimiento definido préviamente con **`CREATE PROCEDURE`**.

`CALL` puede pasar valores al llamador usando parámetros declarados como **`OUT`** o **`INOUT`** .

Sentencia compuesta `BEGIN ... END`

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

Los procedimientos almacenados pueden contener varios comandos, usnado un comando compuesto **`BEGIN ... END`** .

Un comando compuesto puede etiquetarse. *`end_label`* no puede darse a no ser que también esté presente *`begin_label`* , y si ambos lo están, deben ser el mismo.

Tenga en cuenta que la cláusula opcional **`[NOT] ATOMIC`** no está soportada. Esto significa que no hay un punto transaccional al inicio del bloque de instrucciones y la cláusula **`BEGIN`** usada en este contexto no tiene efecto en la transacción actual.

Usar múltiples comandos requiere que el cliente sea capaz de enviar cadenas de consultas con el delimitador de comando **`;`**. Esto se trata en el cliente de línea de comandos `mysql` con el comando **`delimiter`**. Cambiar el delimitador de final de consulta **`;`** end-of-query (por ejemplo, a **`//`**) permite usar **`;`** en el cuerpo de la rutina.

Variables en procedimientos almacenados

Pude declarar y usar una variable dentro de una rutina.

Declarar variables locales con DECLARE

```
DECLARE var_name[,...] type [DEFAULT value]
```

Este comando se usa para declarar variables locales. Para proporcionar un valor por defecto para la variable, incluya una cláusula **DEFAULT** . El valor puede especificarse como expresión, no necesita ser una constante. Si la cláusula **DEFAULT** no está presente, el valor inicial es **NULL**.

La visibilidad de una variable local es dentro del bloque **BEGIN ... END** donde está declarado. Puede usarse en bloques anidados excepto aquéllos que declaren una variable con el mismo nombre.

Sentencia SET para variables

```
SET var_name = expr [, var_name = expr] ...
```

El comando **SET** en procedimientos almacenados es una versión extendida del comando general **SET**. Las variables referenciadas pueden ser las declaradas dentro de una rutina, o variables de servidor globales.

El comando **SET** en procedimientos almacenados se implemnta como parte de la sintaxis **SET** pre-existente. Esto permite una sintaxis extendida de **SET a=x, b=y,...** donde distintos tipos de variables (variables declaradas local y globalmente y variables de sesión del servidor) pueden mezclarse. Esto permite combinaciones de variables locales y algunas opciones que tienen sentido sólo para variables de sistema; en tal caso, las opciones se reconocen pero se ignoran.

La sentencia SELECT ... INTO

```
SELECT col_name[,...] INTO var_name[,...] table_expr
```

Esta sintaxis **SELECT** almacena columnas seleccionadas directamente en variables. Por lo tanto, sólo un registro puede retornarse.

```
SELECT id,data INTO x,y FROM test.t1 LIMIT 1;
```

Constructores de control de flujo

Los constructores **IF**, **CASE**, **LOOP**, **WHILE**, **ITERATE**, y **LEAVE** están completamente implementados.

Estos constructores pueden contener un comando simple, o un bloque de comandos usando el comando compuesto **BEGIN ... END**. Los constructores pueden estar anidados.

Los bucles **FOR** no están soportados.

Sentencia IF

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

IF implementa un constructor condicional básico. Si *search_condition* se evalúa a cierto, el comando SQL correspondiente listado se ejecuta. Si no coincide ninguna *search_condition* se ejecuta el comando listado en la cláusula **ELSE**. *statement_list* puede consistir en varios comandos.

Tenga en cuenta que también hay una *función* **IF()**, que difiere del *commando* **IF** descrito aquí.

La sentencia CASE

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

O:

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

El comando **CASE** para procedimientos almacenados implementa un constructor condicional complejo. Si una *search_condition* se evalúa a cierto, el comando SQL correspondiente se ejecuta. Si no coincide ninguna condición de búsqueda, el comando en la cláusula **ELSE** se ejecuta.

Sentencia LOOP

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

LOOP implementa un constructor de bucle simple que permite ejecución repetida de comandos particulares. El comando dentro del bucle se repite hasta que acaba el bucle, usualmente con un comando **LEAVE** .

Un comando **LOOP** puede etiquetarse. *end_label* no puede darse hasta que esté presente *begin_label* , y si ambos lo están, deben ser el mismo.

Sentencia LEAVE

```
LEAVE label
```

Este comando se usa para abandonar cualquier control de flujo etiquetado. Puede usarse con **BEGIN ... END** o bucles.

La setencia ITERATE

```
ITERATE label
```

ITERATE sólo puede aparecer en comandos **LOOP**, **REPEAT**, y **WHILE** . **ITERATE** significa “vuelve a hacer el bucle.”

Por ejemplo:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN ITERATE label1; END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END
```

Sentencia REPEAT

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

El comando/s dentro de un comando **REPEAT** se repite hasta que la condición *search_condition* es cierta.

Un comando **REPEAT** puede etiquetarse. *end_label* no puede darse a no ser que *begin_label* esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

```
mysql> delimiter //

mysql> CREATE PROCEDURE dorepeat(p1 INT)
    -> BEGIN
    ->   SET @x = 0;
    ->   REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
    -> END
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL dorepeat(1000)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+------+
| @x   |
+------+
| 1001 |
+------+
```

```
1 row in set (0.00 sec)
```

Sentencia `WHILE`

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

El comado/s dentro de un comando **`WHILE`** se repite mientras la condición *`search_condition`* es cierta.

Un comando **`WHILE`** puede etiquetarse. *`end_label`* no puede darse a no ser que *`begin_label`* también esté presente, y si lo están, deben ser el mismo.

Por ejemplo:

```
CREATE PROCEDURE dowhile()
BEGIN
  DECLARE v1 INT DEFAULT 5;

  WHILE v1 > 0 DO
    ...
    SET v1 = v1 - 1;
  END WHILE;
END
```

Vamos a mostrar un ejemplo de uso de triggers y procedimientos almacenados paso a paso. Quizás el ejemplo no sea el más correcto, o simplemente sea poco útil, pero lo importante es el uso en sí, no el ejemplo.

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

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

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

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

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `test`.`sp_comisiones`$$
CREATE PROCEDURE `test`.`sp_comisiones` (IN mivendedor INT)
BEGIN
DECLARE micomision INT DEFAULT 0;
DECLARE suma INT;
DECLARE existe BOOL;
select IFNULL(sum(importe),0) into suma from ventas where producto = 1 and
vendedor = mivendedor;
SET micomision = micomision + (suma * 0.15);
select IFNULL(sum(importe),0) into suma from ventas where producto = 2 and
vendedor = mivendedor;
SET micomision = micomision + (suma * 0.1);
select IFNULL(sum(importe),0) into suma from ventas where producto = 3 and
vendedor = mivendedor;
SET micomision = micomision + (suma * 0.2);
select count(1)>0 into existe from comisiones where vendedor = mivendedor;
if existe then
UPDATE comisiones set comision = comision+micomision where vendedor =
mivendedor;
else
insert into comisiones (vendedor, comision) values (mivendedor, micomision);
end if;
END$$
DELIMITER ;
```

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

```
DELIMITER $$
DROP TRIGGER `test`.`tr_ventas_insert`$$
CREATE TRIGGER `test`.`tr_ventas_insert` AFTER INSERT on `test`.`ventas`
FOR EACH ROW BEGIN
```

```
call sp_comisiones(new.vendedor);
END$$
DELIMITER ;
```