Procedimientos almacenados y funciones

Los procedimientos almacenados y funciones son nuevas funcionalidades de la versión de MySQL 5.0. Un procedimiento almacenado es un conjunto de comandos SQL que pueden almacenarse en el servidor. Una vez que se hace, los clientes no necesitan relanzar los comandos individuales pero pueden en su lugar referirse al procedimiento almacenado.

Algunas situaciones en que los procedimientos almacenados pueden ser particularmente útiles:

- Cuando múltiples aplicaciones cliente se escriben en distintos lenguajes o funcionan en distintas plataformas, pero necesitan realizar la misma operación en la base de datos.

- Cuando la seguridad es muy importante. Los bancos, por ejemplo, usan procedimientos almacenados para todas las oparaciones comunes. Esto proporciona un entorno seguro y consistente, y los procedimientos pueden asegurar que cada operación se loguea apropiadamente. En tal entorno, las aplicaciones y los usuarios no obtendrían ningún acceso directo a las tablas de la base de datos, sólo pueden ejectuar algunos procedimientos almacenados.

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 si 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, que también usa IBM DB2.

La implementación de MySQL de procedimientos almacenados está en progreso. Toda la sintaxis descrita en este capítulo se soporta y cualquier limitación y extensión se documenta apropiadamente.

Logueo binario para procedimientos almacenados se hace como se describe en Sección: “Registro binario de procedimientos almacenados y disparadores”.

Los Procedimientos almacenados y las tablas de permisos

Los procedimientos almacenados requieren la tabla proc en la base de datos mysql. Esta tabla se crea durante la isntalación de MySQL 5.0. Si está actualizando a MySQL 5.0 desde una versión anterior, asegúrese de actualizar sus tablas de permisos para asegurar que la tabla proc existe.

Desde MySQL 5.0.3, el sistema de permisos se ha modificado para tener en cuenta los procedimientos almacenados como sigue:

El permiso CREATE ROUTINE se necesita para crear procedimientos almacenados.

El permiso ALTER ROUTINE se necesita para alterar o borrar procedimientos almacenados. Este permiso se da automáticamente al creador de una rutina.

El permiso EXECUTE se requiere para ejectuar procedimientos almacenados. Sin embargo, este permiso se da automáticamente al creador de la rutina. También, la característica SQL SECURITY por defecto para una rutina es DEFINER, lo que permite a los usuarios que tienen acceso a la base de datos ejecutar la rutina asociada.

Sintaxis de procedimientos almacenados.

Los procedimientos almacenados y funciones se crean con los 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.

Se puede calificar el nombre 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 el procedimientos almacenados p o la funcione f asociados a la base de datos test, puede hacerse mediante CALL test.p() o test.f().

Cuando se borra una base de datos, todos los procedimientos almacenados asociados con ella también se borran.

(En MySQL 5.0.0, los procedimientos almacenados son globales y no están asociados con una base de datos. Heredan la base de datos por defecto del llamador. Si se ejecuta USE db_name desde la rutina, la base de datos por defecto original se restaura a la salida de la rutina.)

CREATE PROCEDURE y CREATE FUNCTION

Se describe a continuación la sintaxis usada para crear, alterar, borrar, y consultar procedimientos almacenados y funciones:

```
CREATE PROCEDURE nombre_procedimiento ([parametro[,...]])
    [característica ...] cuerpo_de_rutina

CREATE FUNCTION nombre_funcion ([parametro[,...]])
    RETURNS tipo
    [característica...] cuerpo_de_rutina

parametro:
    [ IN | OUT | INOUT ] nombre_parametro  tipo

tipo:
    Cualquier tipo de dato válido en MySQL

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

cuerpo_de_rutina:
    cualquier procedimiento almacenados o comando SQL válido
```

Las instrucciones anteriores son las encargadas de crear un procedimiento o función almacenada.
Desde MySQL 5.0.3, para crear una rutina, es necesario tener el permiso CREATE ROUTINE, y los permisos ALTER ROUTINE y EXECUTE se asignan automáticamente al creador. Si se permite logueo binario necesita también el permisos SUPER como se describe en “Registro binario de procedimientos almacenados y disparadores”.
Por defecto, la rutina se asocia con la base de datos actual. Para asociar la rutina explícitamente a una base de datos, hay que utilizar db_name.sp_name para crearla.

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 se producirá un error de sintaxis. También debe usarlo cuando invoca la rutina posteriormente.

La cláusula RETURNS solo se utiliza con FUNCTION, es obligatoria en ese caso. Sirve para indicar el tipo de retorno de la función, y el cuerpo de la función debe contener un comando RETURN que devuelva un valor de ese tipo.

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, se usa la palabra clave OUT o INOUT antes del nombre del parámetro. Especificando IN, OUT, o INOUT sólo es valido para una PROCEDURE.

El comando CREATE FUNCTION se usa en versiones anteriores de MySQL para soportar UDFs (User Defined Functions) (Funciones Definidas por el Usuario). UDFs se soportan, incluso con la existencia de procedimientos almacenados. Un UDF puede tratarse como una función almacenada externa. Sin embargo, tenga en cuenta que los procedimientos almacenados comparten su espacio de nombres con UDFs.

El fabricante promete introducir en el futuro, un marco para procedimientos almacenados externos. Esto permitira escribir procedimientos almacenados en lenguajes distintos a SQL. Uno de los primeros lenguajes a soportar será PHP ya que el motor central de PHP es pequeño, con flujos seguros y puede empotrarse fácilmente. Como el marco es de desarrollo comunitario, se espera que soportarte muchos otros lenguajes.

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 lo especificamos el valor por defecto es NOT DETERMINISTIC.

Para replicación, use la función NOW(). El uso de RAND() no hace que una rutina sea no determinista. Para NOW(), el log binario incluye el tiempo y hora y replica correctamente. RAND() también replica correctamente mientras se invoque sólo una vez dentro de una rutina. (Puede considerar el tiempo y hora de ejecución de la rutina y una semilla de número aleatorio como entradas implícitas que son idénticas en el maestro y el esclavo.)

Actualmente, la característica DETERMINISTIC se acepta, pero no la usa el optimizador. Sin embargo, si se permite el logueo binario, esta característica afecta si MySQL acepta definición de rutinas.

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 elminará 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 con 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 ;
```

La llamada al procedimiento se hace con CALL

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

Para consultar el valor del parámetro a debemos hacer:

```
mysql> SELECT @a;
```

El resultado suponiendo que la tabla t tiene 3 registros, será el siguente:

@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 ;
```

El llamado podemos hacerlo desde un select:

```
mysql> SELECT hello('world');
```

Y el resultado será el siguiente:

hello('world')
Hello, world!
1 row in set (0.00 sec)

Si el comando RETURN en un procedimiento almacenado retorna un valor de tipo distinto al especificado en RETURNS, el valor de retorno se convertirá al tipo apropiado. Por ejemplo, si una función tiene en RETURNS un valor tipo ENUM o SET, pero el comando RETURN devuelve un entero, el valor retornado por la función es la cadena para un miembro de ENUM o SET.

ALTER PROCEDURE y ALTER FUNCTION

```
ALTER {PROCEDURE | FUNCTION} nombr_rutina [caracteristicas ...]

caracteristicas:
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'cadena de caracteres'
```

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. El permiso se otorga automáticamente al creador de la rutina. Si está activado el logueo binario, necesitará el permiso SUPER, como se describe en “Registro binario de procedimientos almacenados y disparadores”.

Pueden especificarse varios cambios con ALTER PROCEDURE o ALTER FUNCTION.

DROP PROCEDURE y DROP FUNCTION

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

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. Se genera una advertencia que puede verse con SHOW WARNINGS.

SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION

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

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
*************************** 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 'nombre']
```

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 nombre, 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:
```

También puede obtener información de rutinas almacenadas de la tabla ROUTINES en INFORMATION_SCHEMA.

La sentencia CALL

```
CALL nombre_rutina([parametros[,...]])
```

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

CALL puede pasar valores al llamador usando parámetros declarados como OUT o INOUT . También “retorna” el número de registros afectados, que con un programa cliente puede obtenerse a nivel SQL llamando la función

ROW_COUNT() y desde C llamando a la función de la API C mysql_affected_rows().

Sentencia compuesta BEGIN ... END

```
[etiqueta_inicio:] BEGIN
    [lista_sentencias]
END [etiqueta_fin]
```

La sintaxis BEGIN ... END se utiliza para escribir sentencias compuestas que pueden aparecer en el interior de procedimientos almacenados y triggers. Una sentencia compuesta puede contener múltiples sentencias, encerradas por las palabras BEGIN y END. La lista_sentencias es una lista de una o más instrucciones válidas. Cada sentencia dentro de lista_sentencias debe terminar con un punto y coma (;) que es el delimitador de sentencias. La lista_sentencias es opcional, lo que significa que la sentencia compuesta vacía (BEGIN END es correcta.

El uso de múltiples sentencias requiere que el cliente pueda enviar cadenas de sentencias que contengan el delimitador ;. Esto se gestiona en el cliente de línea de comandos mysql con el comando delimiter. Cambiar el delimitador de fin de sentencia ; (por ejemplo con //) permite utilizar ; en el cuerpo de una rutina.
Un comando compuesto puede etiquetarse. No se puede poner etiqueta_fin a no ser que también esté presente etiqueta_inicio, y si ambos están, deben ser iguales.

La cláusula opcional [NOT] ATOMIC no está soportada todavía. 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.

Sentencia DECLARE

El comando DECLARE se usa para definir elementos locales de una rutina: las variables locales, condiciones y handlers, y cursores. Los comandos SIGNAL y RESIGNAL no pueden usarse todavía.

DECLARE puede usarse sólo dentro de comandos compuestos BEGIN ... END y deben ser su inicio, antes de cualquier otro comando.

Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar los cursores o handlers.

Declarar variables locales con DECLARE

```
DECLARE var_name[,...] tipo [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[,...] sql_table_expr
```

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

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

Conditions and Handlers

DECLARE Condiciones - DECLARE handlers

Ciertas condiciones pueden requerir un tratamiento específico. Estas condiciones pueden estar relacionadas con errores, así como control de flujo general dentro de una rutina.

DECLARE Condiciones

```
DECLARE condition_name CONDITION FOR condition_value
```

```
condition_value:
    SQLSTATE [VALUE] sqlstate_value | mysql_error_code
```

Este comando especifica condiciones que necesitan tratamiento específico. Asocia un nombre con una condición de error específica. El nombre puede usarse subsecuentemente en un comando DECLARE HANDLER.
Además de valores SQLSTATE, también soporta códigos de error MySQL.

DECLARE handlers

```
DECLARE handler_type HANDLER FOR condition_value[,...] sp_statement

handler_type:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
  | mysql_error_code
```

Este comando especifica handlers que pueden tratar una o varias condiciones. Si ocurre una de estas condiciones, se ejecuta el comando especificado.

El handler CONTINUE, continúa la rutina actual tras la ejecución del comando del handler. Un handler EXIT, termina la ejecución del comando compuesto BEGIN...END actual. El handler UNDO no se soporta a la fecha.

SQLWARNING es una abreviatura de los códigos SQLSTATE que comienzan con 01.

NOT FOUND abrevia los códigos SQLSTATE que comienzan con 02.

SQLEXCEPTION abrevia los códigos SQLSTATE no tratados por SQLWARNING o NOT FOUND.

Además de los valores SQLSTATE , también soporta los códigos de error MySQL.

Veamos un ejemplo:

```
mysql> CREATE TABLE test.t (s1 int, primary key (s1));
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> delimiter //

mysql> CREATE PROCEDURE handlerdemo ()
    -> BEGIN
    ->   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
    ->   SET @x = 1;
    ->   INSERT INTO test.t VALUES (1);
    ->   SET @x = 2;
    ->   INSERT INTO test.t VALUES (1);
    ->   SET @x = 3;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL handlerdemo()//
Query OK, 0 rows affected (0.00 sec)
```

mysql> SELECT @x//

@x
3
1 row in set (0.00 sec)

Tenga en cuenta que @x llegó a 3, lo que muestra que el prodecimiento handlerdemo() se ha ejecutado hasta el final. Sin la línea DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1; el procedimiento MySQL habría salido por error -ruta por defecto (EXIT)- tras el segundo INSERT fallido debido a la restricción PRIMARY KEY, y entonces SELECT @x nos hubiera devuelto el valor 2.

Cursores

Declarar cursores - OPEN cursor – FETCH cursor y CLOSE cursor.

Pueden usarse cursores simples dentro de procedimientos y funciones almacenadas. La sintaxis es la de SQL empotrado. Los cursores no son sensibles, son de sólo lectura, y no permiten scrolling. No sensible significa que el servidor puede o no hacer una copia de su tabla de resultados.
Los cursores deben declararse antes de declarar los handlers, y las variables y condiciones deben declararse antes de declarar cursores o handlers.

Veamos el siguiente ejemplo:

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE a CHAR(16);
  DECLARE b,c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
```

```
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

  OPEN cur1;
  OPEN cur2;

  REPEAT
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF NOT done THEN
      IF b < c THEN
        INSERT INTO test.t3 VALUES (a,b);
      ELSE
        INSERT INTO test.t3 VALUES (a,c);
      END IF;
    END IF;
  UNTIL done END REPEAT;

  CLOSE cur1;
  CLOSE cur2;
END
```

Declarar cursores

```
DECLARE cursor_name CURSOR FOR select_statement;
```

Este comando declara un cursor. Pueden definirse varios cursores en una rutina, pero cada cursor del bloque debe tener un nombre único.
El comando SELECT no puede tener una cláusula INTO.

Sentencia OPEN cursor

```
OPEN cursor_name
```

Este comando abre un cursor declarado previamente.

Sentencia FETCH cursor

```
FETCH cursor_name INTO var_name [, var_name] ...
```

Este comando toma el siguiente registro (si existe) usando el cursor abierto especificado, y avanza el puntero del cursor en uno.

Sentencia CLOSE cursor

```
CLOSE cursor_name
```

Este comando cierra un cursor abierto previamente.

Si no se cierra explícitamente, un cursor se cierra al final del comando compuesto en que se declara.

Constructores de control de flujo.

Sentencias IF – CASE – LOOP – LEAVE – ITERATE – REPEAT - WHILE

Estos constructores IF, CASE, LOOP, WHILE, ITERATE, y LEAVE están completamente implementados en MYSQL.

Estos constructores pueden estar en un comando simple, o constituir un bloque de comandos usando BEGIN ... END. Los constructores pueden estar anidados.

Los bucles FOR no están soportados todavía en MYSQL.

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 ejectua. Si no coincide ninguna search_condition se ejecuta el comando listado en la cláusula ELSE. La lista 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

Tiene dos implementaciones:

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

O bien:

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 una sentencia condicional compleja. Si la condición search_condition se evalúa a cierto, se ejecuta el comando SQL correspondiente. Si no coincide ninguna condición de búsqueda, se ejecutará el comando en la cláusula ELSE.

Nota: La sitaxis de un comando CASE mostrado aquí para usar dentro de procedimientos almacenados difiere ligeramente de la expresión CASE SQL correspondiente a Funciones de control de flujo.
El comando CASE no puede tener una cláusula ELSE NULL y termina directamente con END CASE en lugar de END.

Sentencia LOOP

```
[rotulo_lazo:] LOOP
   statement_list
END LOOP [rotulo_lazo]
```

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 .

El comando LOOP puede etiquetarse. Tanto el rótulo de inicio, como el de final 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

```
[rotulo_proc:] REPEAT
   statement_list
UNTIL search_condition
END REPEAT [rotulo_proc]
```

La instrucciones dentro de un lazo REPEAT se repiten hasta que la condición search_condition sea cierta.
El lazo REPEAT puede etiquetarse. El rótulo rotulo_proc si está presente, debe ser el mismo al principio y al final.

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

```
[rotulo_proc:] WHILE search_condition DO
   statement_list
END WHILE [rotulo_proc]
```

Las instrucciones dentro de un lazo WHILE se repiten mientras la condición search_condition sea cierta.

Puede usarse etiquetas para un lazo WHILE. Tanto la etiquta al inicio como al final deben ser iguales.

Por ejemplo:

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

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

Registro (log) binario de procedimientos almacenados y disparadores.

En esta sección se describe cómo MySQL trata los procedimientos almacenados (procedimientos o funciones) con respecto al logueo binario. La sección también se aplica a disparadores.

El log binario contiene información sobre comandos SQL que modifican contenidos de base de datos. Esta información se almacena en la forma de "eventos" que describen esas modificaciones.

El log binario tiene dos propósitos importantes:

Es la base de la replicación, ya que la base que cumple la función de maestro envía los eventos contenidos en su log binario a las bases que hacen de esclavos, el esclavo ejecuta estos eventos para lograr los mismos cambios de datos que se hacen en el maestro.

La otra función importante es en la recuperación de datos que utilizan los log binarios. Luego de una restauración de un fichero de copia de seguridad, los eventos en los log binario que se guardaron después de dicha copia de seguridad se re-ejecutan. Estos eventos actualizan la base de datos desde el punto de la copia de seguridad, hasta el momento actual.

El logueo de procedimientos almacenados difieren a partir de la versión MySQL 5.0.6. Antes de esta versión, los comandos que crean y usan procedimientos almacenados no se escriben en el log binario, pero los comandos invocados desde procedimientos almacenados se loguean. Suponga que ejecuta los siguientes comandos:

```
CREATE PROCEDURE mysp
INSERT INTO t VALUES(1);
CALL mysp;
```

En este ejemplo, sólo el comando INSERT aparecerá en el log binario. Los comandos CREATE PROCEDURE y CALL no aparecerán. La ausencia de comandos relacionados con rutinas en el log binario significa que los procedimientos almacenados no se replicarán. También significa que para operaciones de recuperación de datos, al re-ejectuar los eventos del log binario no recuperamos los procedimientos almacenados.

Para lograr la replicación y recuperación de datos, se cambió el logueo de procedimientos almacenados en MySQL versión 5.0.6 y posteriores. Sin embargo, estos cambios generan otros problemas que discutiremos a continuación.

Si no se expresa lo contrario, supondremos que no se ha activado el logueo binario al arrancar el servidor con la opción --log-bin. (Si no activamos el log binario, no es posible la replicación, y tampoco está disponible el log binario para replicación de datos.)

Las características de logueo binario de procedimientos almacenados se describen en la siguiente lista. Algunos son problemas que debemos conocer, pero en algunos casos, hay inicializaciones del servidor pueden usarse para solucionar esos problemas.

Los comandos CREATE PROCEDURE, CREATE FUNCTION, ALTER PROCEDURE, y ALTER FUNCTION se escriben en el log binario, como así también CALL, DROP PROCEDURE, y DROP FUNCTION.

Sin embargo, tenemos asuntos de seguridad concernientes a la replicación: para crear una rutina, el usuario debe tener permiso de CREATE ROUTINE, pero un usuario con este permiso puede escribir una rutina para realizar cualquier acción en un servidor esclavo ya que el flujo SQL de réplica, en el esclavo corre con todos los permisos. Por ejemplo, si el maestro y el esclavo tienen valores de ID de servidor de 1 y 2 respectivamente, un usuario en el maestro puede crear e invocar procedimientos de la siguiente forma:

```
mysql> delimiter //
mysql> CREATE PROCEDURE mysp ()
    -> BEGIN
    ->   IF @@server_id=2 THEN DROP DATABASE accounting; END IF;
    -> END;
    -> //
mysql> delimiter ;
mysql> CALL mysp();
```

Los comandos CREATE PROCEDURE y CALL se escriben en el log binario, así que los ejecutará el esclavo. Ya que el flujo SQL del esclavo tiene todos los permisos, borra la base de datos accounting.

Para evitar este peligro en servidores con logueo binario activado, MySQL 5.0.6 introduce el requerimiento de que los creadores de funciones y procedimientos almacenados deben tener el permiso SUPER, además del permiso CREATE ROUTINE requerido. Del mismo modo, para usar ALTER PROCEDURE o ALTER FUNCTION, debe tener el permiso SUPER además del permiso ALTER ROUTINE . Sin el permiso SUPER ocurre un error:

```
ERROR 1419 (HY000): You do not have the SUPER privilege and
binary logging is enabled (you *might* want to use the less safe
log_bin_trust_routine_creators variable)
```

Quizás no quiera forzar el requerimiento de tener el privilegio SUPER para los creadores de rutinas. Por ejemplo, si todos los usuarios con el permiso CREATE ROUTINE en su sistema son desarrolladores de aplicaciones con experiencia. Para desactivar el requerimiento del privilegio SUPER, podra realizarlo cambiando la variable global de sistema global log_bin_trust_routine_creators a 1. Por defecto, vale 0, pero puede cambiarse así:

```
mysql> SET GLOBAL log_bin_trust_routine_creators = 1;
```

También se puede activar esta variable con la opción --log-bin-trust-routine-creators al arrancar el servidor.

Si el logueo binario no está activado, log_bin_trust_routine_creators no se aplica y no se necesita el permiso SUPER para creación de rutinas.

Una rutina no-determinista que realiza actualizaciones no es repetible, y puede tener dos efectos no deseables:

El esclavo será distinto al maestro.
Los datos restaurados serán distintos a los originales.

Para tratar estos problemas, MySQL fuerza los siguientes requerimientos: En un servidor maestro, se impide la creación y alteración de una rutina que no sea determinista o que no modifique datos. Esto significa que cuando crea una rutina, se debe declarar que es determinista o que no cambia datos. Estas dos características de las rutinas explican a continuación:

DETERMINISTIC, indica que la rutina produce el mismo resultado ante las mismas entradas y NOT DETERMINISTIC indica lo contrario. El valor por defecto es NOT DETERMINISTIC, por lo tanto, se debe especificar DETERMINISTIC explícitamente para declarar que la rutina es determinista.

El uso de las funciones NOW() (o sus sinónimos) o RAND() no hacen que una rutina sea no-determinista necesariamente. Para NOW(), el log binario incluye la fecha y hora por tanto replica correctamente. En el caso de RAND() también replica correctamente mientras se invoque sólo una vez dentro de una rutina.

(Se puede considerar la fecha y hora de ejecución de la rutina y la semilla de números aleatorios como entradas implícitas que son idénticas en el maestro y el esclavo.)

Los modificadores CONTAINS SQL, NO SQL, READS SQL DATA, y MODIFIES SQL proporcionan información acerca de si la rutina lee o escribe datos. Tanto NO SQL o READS SQL DATA indican que una rutina no cambia datos, pero debe especificar uno de estos explícitamente ya que por defecto la opción es CONTAINS SQL si no se expresa otro modificador.

Para que una instrucción de CREATE PROCEDURE o CREATE FUNCTION sea aceptado, se deben especificar explícitamente DETERMINISTIC o NO SQL y READS SQL DATA. De otro modo se produce el siguiente error:

> **ERROR 1418 (HY000)**: This routine has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_routine_creators variable)

Si asigna a log_bin_trust_routine_creators 1, el requerimiento de que la rutina sea determinista o que no modifique datos no hace falta.

Tenga en cuenta que la naturaleza de una rutina se basa en la "honestidad" del creador: MySQL no comprueba que una rutina declarada DETERMINISTIC no contenga comandos que produzcan productos no deterministas.

Un comando CALL se escribe en el log binario si la rutina no retorna error, de otro modo no. Cuando una rutina que modifica datos falla, nos va a dar las siguiente advertencia:

> **ERROR 1417** (HY000): A routine failed and has neither NO SQL nor READS SQL DATA in its declaration and binary logging is enabled; if non-transactional tables were updated, the binary log will miss their changes

Este logueo del comportamiento también tiene su potencial para causar problemas. Si una rutina modifica parcialmente una tabla no transaccional (tal como una tabla MyISAM) y retorna un error, el log binario no refleja estos cambios. Para protegerse de esto, se debe usar trablas transaccionales en la rutina y modificar las tablas dentro de transacciones.

Si se usa la palabra clave IGNORE con INSERT, DELETE, o UPDATE para ignorar errores dentro de una rutina, cuando se produce una actualización parcial puede realizarse sin producir error. Estos comandos se loguean y se replican normalmente.
Si una función almacenada se invoca desde dentro de un comando tal como SELECT que no modifica datos, la ejecución de la función no se escribe en el

log binario, incluso si la función misma modifica datos. Este comportamiento de logueo tiene potencial para causar problemas. Suponga que una función myfunc() se define así:

```
CREATE FUNCTION myfunc () RETURNS INT
BEGIN
    INSERT INTO t(i) VALUES(1);
    RETURN 0;
END;
```

Con esta definición, el siguiente comando modifica la tabla t porque myfunc() modifica t, pero el comando no se escribe en el log binario porque es un SELECT que llama a una función:

```
SELECT myfunc();
```

Una solución a este problema es no utilizar funciones que actualizan dentro de comandos que hacen actualizaciones. Debemos tener en cuenta que aunque el comando DO a veces se ejecuta al evaluar una expresión, como efecto colateral, DO no es una solución a este problema, porque no será escrito en el log binario.
Los comandos que se ejecutan dentro de una rutina no se escriben en el log binario. Supongamos que ejectua los siguientes comandos:

```
CREATE PROCEDURE mysp
   INSERT INTO t VALUES(1);
CALL mysp;
```

En este ejemplo, los comandos CREATE PROCEDURE y CALL aparecerán en el log binario. Pero el comando INSERT no aparecerá. Esto soluciona el problema que ocurría antes de la versión 5.0.6 de MySQL, en las que los comandos CREATE PROCEDURE y CALL no se lograban pero el INSERT sí.

En servidores esclavos, se aplica la siguiente limitación cuando se determina qué eventos del maestro se replican:
reglas --replicate-*-table no se aplican a comandos CALL o a comandos dentro de rutinas --> Las reglas en estos casos siempre retornan “replica!”

Los disparadores o trigger, son similares a los procedimientos almacenados, así que las notas precedentes también se aplican a los disparadores, solo habrá que tener en cuenta que un trigger es deterministico. La sentencia CREATE TRIGGER, no admite como opcional la característica DETERMINISTIC. Sin embargo, esta hipótesis no ser válida en algunos casos. Por ejemplo, la función UUID() no es determinista (y no se replica). Debemos tener cuidado al utilizar tales funciones y disparadores.

En las versiones actuales de MySQL, los triggers no actualizan tablas, pero lo

harán en el futuro. Por esta razón, se producen mensajes de error similares a los que provocan los procedimientos almacenados, con CREATE TRIGGER si no se tiene el permiso SUPER y log_bin_trust_routine_creators es 0.

Esto que hemos tratado anteriormente, se debe a que el logueo binario se hace a nivel de comandos SQL. En las versiones MySQL 5.1 y posteriores, se introducirá logueo binario a nivel de registro, que se produce en un nivel más granular, ya que especifica qué cambios hacer a registros individuales como resultado de ejecutar los comandos SQL.

Programación Python con la base de datos MySQL

Para trabajar con estos ejemplos, supondremos que tienen instalado PyCharm y se ha instalado la librería pymysql.
Consideraremos también que tenemos acceso a una base de datos MySQL, en donde existe un usuario pruebas con clave de acceso Pru-12345 y que dentro del mismo, hemos creado la base de datos DEMO y donde exite una tabla PERSONAL con datos.
Comenzaremos con un ejemplo simple de código que define una función para contar registros:

```
#!/usr/bin/python3
def cuenta_registros01():
    import pymysql
# Abrir Conexión a la Base de Datos
    db = pymysql.connect("localhost","pruebas","Pru-12345","DEMO" )
# Preparar objeto cursor con el método cursor()
    cursor = db.cursor()
# ejecutamos una consulta SQL query con el método execute().
    cursor.execute("SELECT count(*) from PERSONAL")
# Obtenemos resultado de una fila con el método fetchone().
    data = cursor.fetchone()
    print ("Nro de Registros en Personal: %s " % data)
# nos desconectamos del servidor
    db.close()
```

Para utilizar esta función, simplemente la convocamos mediante:

```
cuenta_registros01()
```

Ahora supondremos que la tabla PERSONAL, que posee los siguientes campos: legajo, apellido, nombre y fechaingreso.
Definimos una función para listar el contenido de la tabla:

```
def mostrar_datos():
    import pymysql
    # Abrimos conexión a la Base de Datos
    db = pymysql.connect("localhost","pruebas","Pru-12345","DEMO" )
    # se prepara un objeto cursor mediatne el método cursor()
    cursor = db.cursor()
    # Preparamos la consulta SQL a ejecutar en la Base de datos.
    sql = "SELECT * FROM PERSONAL"
    try:
        # Ejecutamos la consulta SQL
        cursor.execute(sql)
        # Guardamos las filas a una lista de listas.
        results = cursor.fetchall()
        for row in results:
                legajo = row[0]
                apellido = row[1]
```

```
            nombre = row[2]
            fechaingreso = row[3]
            # Se imprimen los resultados fila a fila
            print ("legajo = %s,apellido = %s,nombre = %s,fechaingreso = %s"
% \
            (legajo, apellido, nombre, fechaingreso))
    except:
        print ("Error: no se encontraron registros")
# nos desconectamos del servidor
        db.close()
```

Para utilizar esta función, simplemente la convocamos mediante:

```
mostrar_datos()
```

Creación de una Tabla

Una vez que hemos establecido una conexión, estamos en condiciones de crear tablas o hacer registros en ellas mediante el método cursor().
En este ejemplo, vemos la creación de la tabla PERSONAL:

```
#!/usr/bin/python3
import pymysql
# Abrir la conexión a la base de datos
db = pymysql.connect("localhost","pruebas","Pru-12345","DEMO" )
# Prepararmos un objeto cursor mediante el método cursor()
cursor = db.cursor()
# Se borra (drop) la tabla si existe mediante el método execute().
cursor.execute("DROP TABLE IF EXISTS PERSONAL")
# Creamos la tabla de acuerdo a los requerimientos
sql = """CREATE TABLE PERSONAL(
   NOMBRE  CHAR(20) NOT NULL,
   APELLIDO  CHAR(20),
   EDAD INT,
   SEXO CHAR(1),
   SALARIO FLOAT )"""
cursor.execute(sql)
# Nos desconectamos del servidor
db.close()
```

Operación agregar registros en una tabla: INSERT

Esta instrucción permite agregar registros a una tabla de la base de datos. El siguiente ejemplo usa la instrucción sql INSERT para agregar registros a la tabla PERSONAL:

```
#!/usr/bin/python3
import pymysql
# Abrir la conexión a la base de datos
```

```
db = pymysql.connect("localhost","pruebas","Pru-12345","DEMO" )
# Preparamos un objeto cursor mediante el método cursor()
cursor = db.cursor()
# Preparamos una consulta SQL para agregar, INSERT un registro a la tabla.
sql = """INSERT INTO PERSONAL(NOMBRE,
   APELLIDO, EDAD, SEXO, SALARIO)
   VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # Ejecutamos el comando SQL
   cursor.execute(sql)
   # Confirmamos los cambios con Commit
   db.commit()
except:
   # Desacemos Rollback en caso de un error
   db.rollback()
# Nos desconectamos del servidor
db.close()
```

El ejemplo anterior, tambien puede escribirse de modo que la carga se haga en forma dinámica:

```
#!/usr/bin/python3
import pymysql
# Abrir la conexión a la base de datos
db_host = "localhost"
db_user = "pruebas"
db_pass = "Pru-12345"
db_name = "DEMO"
db = pymysql.connect(db_host,db_user,db_pass,db_name)
# Preparamos un objeto cursor mediante el método cursor()
cursor = db.cursor()
# Preparamos la consutla SQL para agregar INSERT un registro
# a la tabla de la base de datos.
sql = "INSERT INTO PERSONAL(NOMBRE, \
   APELLIDO, EDAD, SEXO, SALARIO) \
   VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
   ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # Ejecutamos el comando SQL
   cursor.execute(sql)
   # Se confirman Commit los cambios a la base de dato
   db.commit()
except:
   # Se deshacen Rollback los cambios si hay errores
   db.rollback()
# nos desconectamos del servidor
db.close()
```

Operación de consulta mediante LECTURA de datos.

La consulta mediante lectura de cualquier tabla de la base de datos, implica la recolección de toda información útil disponible de dicha tabla.

Luego de establecer la conexión con la base de datos, estamos en condiciones de hacerle consultas. Podemos utilizar tanto el método fetchone() para recuperar un solo registro, o el método fetchall() para recuperar varios registros de una tabla.

fetchone() – es el método que nos devuelve la próxima fila del conjunto de resultados de la consulta. El conjunto de resultados, es el objeto que devuelve la consulta cuando se utiliza un objeto cursor para consultar una tabla.

fetchall() – este método devuelve todas la filas en el conjunto de resultados. En el caso de que se hayan extraído algunas filas del conjunto de resultados, este método recupera las filas restantes del conjunto de resultados.

rowcount – este es un atributo de solo lectura que devuelve el número de filas que se ven afectadas por el método execute().

En el siguiente ejemplo, el procedimiento busca todos los registros de la tabla PERSONAL que tengan un salario mayor a $1000 -

```
#!/usr/bin/python3
import pymysql
# Abrir la conexión a la base de datos
db_host = "localhost"
db_user = "pruebas"
db_pass = "Pru-12345"
db_name = "DEMO"
db = pymysql.connect(db_host,db_user,db_pass,db_name)
# Preparamos un objeto cursor mediante el método cursor()
cursor = db.cursor()
# Preparamos la consulta SQL a realizar.
sql = "SELECT * FROM PERSONAL \
      WHERE SALARIO > '%d'" % (1000)
try:
   # Ejecutamos el comando SQL
   cursor.execute(sql)
   # Cargamos el resultado en una lista de listas.
   results = cursor.fetchall()
   for row in results:
      v_nombre   = row[0]
      v_apellido = row[1]
      v_edad     = row[2]
      v_sexo     = row[3]
      v_salario  = row[4]
      # Now print fetched result
      print ("Nombre = %s,Apellido = %s,Edad = %d,Sexo = %s,Salario = %d"
% \
          (v_nombre, v_apellido, v_edad, v_sexo, v_salario ))
except:
   print ("Error: no se encontraron datos")
# Nos desconectamos del servidor
```

```
db.close()
```

La salida de la consulta anterior es la siguiente:

```
Nombre = Mac, Apellido = Mohan, Edad = 20, Sexo = M, Salario = 2000
```

Actualización de registros, operación Update.

La operación UPDATE en cualquier tabla de la base de datos, permite actualizar uno o más campos de registros, que están ya cargados en la base de datos. El ejemplo siguiente, muestra un procedimiento que actualiza todos los registros que tengan el campo SEXO en valor 'M'. Se hará de esta manera aumentar en uno la edad de todos los varones de la tabla PERSONAL.

```
#!/usr/bin/python3
import pymysql
# Abrir la conexión a la base de datos
db_host = "localhost"
db_user = "pruebas"
db_pass = "Pru-12345"
db_name = "DEMO"
db = pymysql.connect(db_host,db_user,db_pass,db_name)
# Preparamos un objeto cursor mediante el método cursor()
cursor = db.cursor()
# Preparamos la consulta SQL para actualizar los registros
# que cumplan la condición requerida
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')
try:
   # Ejecutamos el comando SQL
   cursor.execute(sql)
   # Confirmamos los cambios mediante Commit
   db.commit()
except:
   # Deshacemos los cambios meiante Rollback si hubo errores
   db.rollback()
# Nos desconectamos del servidor
db.close()
```

Borrado de registros, operación DELETE.

La operación de borrado, DELETE, es necesaria para borrar registros de una tabla de la base de datos. En el ejemplo que se da a continuación, se muestra un procedimiento que va a borrar todos los registros de la tabla PERSONAL, cuya edad sea superior a 20 años.

```
#!/usr/bin/python3
import pymysql
# Abrir la conexión a la base de datos
```

```
db_host = "localhost"
db_user = "pruebas"
db_pass = "Pru-12345"
db_name = "DEMO"
db = pymysql.connect(db_host,db_user,db_pass,db_name)
# Preparamos un objeto cursor mediante el método cursor()
cursor = db.cursor()
# Preparamos la consulta SQL necesaria para borrar
# los registros que cumplen la condición requerida
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Ejecutar el comando SQL
   cursor.execute(sql)
   # Confirmar mediante Commit los cambios realizados
   db.commit()
except:
   # Deshacemos los cambios si se producen errores
   db.rollback()
# Nos desconectamos del servidor
db.close()
```

Operativa Transaccional, el uso de las transacciones.

El uso de transacciones es un mecanismo que nos pertite asegurar la consistencia. Las transacciones cumplen cuatro propiedades fundamentales:

Atomicidad	La transacción se debe completar, de lo contrario no debe suceder nada.
Consistencia	Una transacción debe comenzar en un estado consistente y dejar el sistema en un estado consistente.
Aislamiento	Los estados intermedios de la actual transacción, no deben tener visibilidad fuera de esta misma transacción.
Durabilidad	Una vez confirmada la transacción (commit), sus efectos deben permanecer (ser persistentes) incluso luego de una falla del sistema.

La DB API 2.0 de Python nos proporciona dos métodos para confirmar o deshacer transacciones, tal como se ha expuesto en los ejemplos presentados anteriormente.

La operación confirmar transaccion - COMMIT

La operación Commit es la forma de darle a la base de datos una señal de luz verde para que guarde todos los cambios, de forma que luego no se pueden revertir o deshacer.
Par utilizar este método se lo hace mediante:

```
db.commit()
```

Operación dehacer o revertir transacción – ROLLBACK.

Si no satisface uno o más cambios de realizados durante la transacción, se pueden deshacer o revertir completamente mediante el método rollback().
Para utilizar este método se lo hace mediante:

```
db.rollback()
```

Desconectarse de la base de Datos

Para desconectarnos de la base de datos lo hacemos mediante el método close(). Se lo utiliza de la siguiente manera:

```
db.close()
```

Si cerramos la conexión a la base de datos mediante el método close(), cualquier transacción en progreso será revertida por el gestor de Base de Datos. Sin embargo, a fin de no depender de detalles de bajo nivel de la implementación del Gestor de Base de Datos, es preferible que nuestra aplicación realice de forma explícita la confirmación o reversión de las transacciones previo a la desconexión.

Manejo de Errores

Existen numerosas fuentes de errores. Algunos ejemplos pueden ser debido a errores de sintaxis en la instrucción SQL ejecutada, una falla de conexión, o una llamado al método fetch en medio de una instrucción ya cancelada o finalizada.
La API DB define un listado de errores posibles de producirse en cada módulo de base de datos. La siguiente tabla nos brinda un listado de dichas excepciones.

Nro de Excepción	Tipo de excepción y descripción
1 Warning	*Se usa para cuestiones no fatales. Conforma una subclase de los errores estándard (StandardError).*
2 Error	*Clase base para errores. Conforma una subclase de los errores estándard (StandardError.)*
3 InterfaceError	*Se usa para errores en el módulo de base de datos, no en la base de datos en sí. Debe conformar una subclase de errores.*
4 DatabaseError	*Se usa para errores en la base de datos. Debe conformar una subclase de errores.*
5 DataError	*Es una subclase de error de la Base de Datos, que se refiere a errores en los datos.*

6 OperationalError	*Es una Subclase de errores de Base de Datos que se refiere a errores como pérdidas de conexión a la base de datos. Este tipo de errores queda fuera del control del scripter de Python.*
7 IntegrityError	*Es una Subclase de error de la Base de Datos para situaciones que podrían dañar la integridad referencial, tales como restricciones de unicidad o claves foráneas.*
8 InternalError	*Es una Subclase de error de la Base de Datos que se refiere a errores internos del módulo de base de datos, tales como un cursor que ha dejado de estar activo.*
9 ProgrammingError	*Es una Subclase de error de Base de Datos que se refiere a errores tales como error de nombre de tabla y temas por el estilo que pueden deberse directamente al programador.*
10 NotSupportedError	*Es una subclase de error de Base de datos que se refiere a llamadas a funcionalidades que no están soportadas.*

Nuestros scripts de Python deben manejar estos errores, pero antes de utilizar las excepciones anteriores, asegúrate de que tu MySQLdb posee soporte para dicha excepción. Puedes obtener mayor información acerca de estos temas en las especificaciones de la DB API 2.0.