Unidad 3 – 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 operaciones 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 ejecutar 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.
No está permitido la utilización de símbolos en el nombre de los procedimientos, al igual que en tablas, base de datos, trigger, cursores, etc. No es conveniente usar ñ, punto, acentos, barras, guion medio. El guion bajo está permitido.

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 instalación 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 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.

En MySQL 5.0.0, los procedimientos almacenados son globales y no 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.
MySQL soporta la extensión muy útil que permite el uso de comandos regulares SELECT (esto es, sin usar cursores o variables locales) 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. Esto significa que el cliente debe usar una biblioteca cliente de MySQL como mínimos desde 4.1.
La siguiente sección describe la sintaxis usada para crear, alterar, borrar, y consultar procedimientos almacenados y funciones.

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
```

Estos comandos crean una rutina 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 a su creador. Si se permite logueo binario necesita también el permisos SUPER
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.
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.
Un marco para procedimientos almacenados externos se introducirá en el futuro. Esto permitirá 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 público, se espera soportar 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 se da ni DETERMINISTIC ni NOT DETERMINISTIC por defecto es NOT DETERMINISTIC.
Para replicación, use la función NOW() (o su sinónimo) o RAND() no hace una rutina no determinista necesariamente. 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.

En MySQL los procedimientos almacenados no retornan valores. Las funciones si retornan valores.

Para que MySQL te devuelva una tabla en una rutina almacenada, debe ser forzosamente en un Stored Procedure. Una Stored function no puede hacerlo porque sólo puede devolver un único valor, y no una tabla. La ventaja de una SF es que puedes usarla para obtener un valor dado desde una consulta como si fuese una función nativa.

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.

```
delimiter //
CREATE PROCEDURE prueba (OUT parametro1 INT)
BEGIN
        SELECT COUNT(*)
        INTO parametro1
        FROM clientes;
END
//

delimiter ;
CALL prueba(@a);
```

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:

```
delimiter //
CREATE FUNCTION saludo (s CHAR(20))
RETURNS CHAR(50)
RETURN CONCAT(Hola, ',s,'!');
//

delimiter ;
SELECT saludo('Jesus');
```

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

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. El permiso se otorga automáticamente al creador de la rutina. Si está activado el logueo binario, necesitará el permiso SUPER.
Pueden especificarse varios cambios con ALTER PROCEDURE o ALTER FUNCTION.

DROP PROCEDURE y DROP FUNCTION

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

Este comando se usa para borrar un procedimiento almacenado o función. 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} 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.

```
SHOW CREATE FUNCTION test.hello\G
*************************** 1. row ***************************
Function: hello
sql_mode:
Create Function: CREATE FUNCTION `test`.`saludo`(s CHAR(20)) RETURNS CHAR(50)
RETURN CONCAT(Hola, ',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. 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 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. *lista_sentencias* es una lista de una o más sentencias. Cada sentencia dentro de *lista_sentencias* debe terminar con un punto y como (;) delimitador de sentencias. *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 *end_label* a no ser que también esté presente *begin_label* , 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 varios iconos locales de una rutina: las variables locales, condiciones y handlers y cursores. Los comandos SIGNAL y RESIGNAL no se soportan en la actualidad.
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.

Variables en procedimientos almacenados

Es posible declarar y usar una variable dentro de una rutina.

```
DECLARE b,c int;
DECLARE a char(16);
DECLARE d boolean default true
BEGIN
        DECLARE con tint default 2;
END
```

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. Se utiliza para asignar valores.
Las variables referenciadas pueden ser las declaradas dentro de una rutina, o variables de servidor globales.
El comando SET en procedimientos almacenados se implementa 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;
```

Parámetros de salida OUT

```
delimiter //
CREATE PROCEDURE devolver_barrio(
        IN pk int,
        OUT nombre varchar(50)
)
BEGIN
        SELECT barrio into nombre
        FROM barrios
        WHERE pk_barrio=pk;
END
//

call devolver_barrio ('1',@nombre);
select (@nombre)
```

Ejemplos:

```
DELIMITER //
CREATE PROCEDURE consulta_barrios()
        BEGIN
                Select * from barrios;
        END
//
```

```
DELIMITER //
CREATE PROCEDURE insertar_barrio(IN cod int, nombre varchar(50))
        BEGIN
                INSERT INTO barrios(pk_barrio,barrio)
                VALUES(cod,nombre);
        END
//

CALL insertar_barrio(16,"Parque Luro");
```

ACLARACION: si el campo cod está definido como autoincremental, el procedimiento generará un error.

```
DELIMITER //
CREATE PROCEDURE buscar_barrio(IN cod int)
        BEGIN
                SELECT *
                FROM barrios
                WHERE pk_barrio=cod;
        END
//

CALL buscar_barrio(16);
```

```
DELIMITER //
CREATE FUNTION retornar_cant_barrios()
RETURNS int
BEGIN
        DECLARE barrio int;
                SELECT count(*)
                INTO barrio
                FROM barrios;
        RETURN barrio;
END
//

SELECT retornar_cant_barrios();
```

```
DELIMITER $$
CREATE FUNCTION cantidadDepto(depto int)
RETURNS int
BEGIN
        DECLARE total int;
        SET total = (   SELECT COUNT(a.idAfiliado)
                        FROM afiliados a INNER JOIN gruposFamiliares gf ON
                        a.grupoFamiliar=gf.idGrupoFamiliar
                        WHERE gf.codpostal=depto);
RETURN total;
END$$
DELIMITER;
```

```
DELIMITER $$
DROP PROCEDURE IF EXISTS procedimiento1$$
CREATE PROCEDURE procedimiento1()
BEGIN
        SELECT codPostal, count(codPostal)
        FROM afiliados a  INNER JOIN gruposfamiliares gf ON
                a.grupoFamiliar=gf.idGrupoFamiliar
        WHERE estadoAfiliado(a.idAfiliado)=1
        GROUP BY codPostal;
END$$
DELIMITER;
```

Procedimiento Almacenado y Estructuras de Control en MySQL WHILE, IF THEN ELSE, SWITCH

```
DECLARE edad INT;
```

Esta variable tendrá un ámbito local y cuando se acabe el procedimiento no podrá ser accedida. Una vez que la variable es declarada, para cambiar su valor usaremos la sentencia SET como en el siguiente ejemplo:

```
SET edad = 56 ;
```

Para poder acceder a una variable a la finalización de un procedimiento se tiene que usar parámetros de salida como en el siguiente Código:

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;
//
```

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;
//
```