Capítulo 19. Procedimientos almacenados y funciones

Tabla de contenidos

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. Más discusión o restricciones de uso de procedimientos almacenados se da en Apéndice H, *Restricciones en características de MySQL*.

Logueo binario para procedimientos almacenados se hace como se describe en Sección 19.3, “Registro binario de procedimientos almacenados y disparadores”.

19.1. 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. Consulte Sección 2.10.2, “Aumentar la versión de las tablas de privilegios”.

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.

19.2. 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 almacenadoso funciones se asocian con una base de datos. Esto tiene varias implicaciones:

- Cunado 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 diractamente 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.

19.2.1. 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`** como se describe en Sección 19.3, “Registro binario de procedimientos almacenados y disparadores”.

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 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). Consulte Sección 27.2, “Añadir nuevas funciones a MySQL”. 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 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 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. Consulte Sección 19.3, “Registro binario de procedimientos almacenados y disparadores”.

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 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)
```

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`** .

19.2.2. 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** , como se describe en Sección 19.3, “Registro binario de procedimientos almacenados y disparadores”.

Pueden especificarse varios cambios con **ALTER PROCEDURE** o **ALTER FUNCTION**.

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

19.2.4. 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,'!')
```

19.2.5. 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:
```

También puede obtener información de rutinas almacenadas de la tabla **ROUTINES** en **INFORMATION_SCHEMA**. Consulte Sección 22.1.14, “La tabla **INFORMATION_SCHEMA ROUTINES**”.

19.2.6. 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()** .

19.2.7. 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.

19.2.8. Sentencia DECLARE

El comando **DECLARE** se usa para definir varios iconos locales de una rutina: las variables locales (consulte Sección 19.2.9, “Variables en procedimientos almacenados”), condiciones y handlers (consulte Sección 19.2.10, “Conditions and Handlers”) y cursores (consulte Sección 19.2.11, “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.

19.2.9. Variables en procedimientos almacenados

Pude declarar y usar una variable dentro de una rutina.

19.2.9.1. 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.

19.2.9.2. 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.

19.2.9.3. 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;
```

19.2.10. Conditions and 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.

19.2.10.1. Condiciones DECLARE

```
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** . Consulte Sección 19.2.10.2, “**DECLARE** handlers”.

Además de valores SQLSTATE , los códigos de error MySQL se soportan.

19.2.10.2. 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 una de estas condiciones ocurren, el comando especificado se ejecuta.

Para un handler **CONTINUE** , continúa la rutina actual tras la ejecución del comando del handler. Para un handler **EXIT** , termina la ejecución del comando compuesto **BEGIN...END** actual. El handler de tipo **UNDO** todavía no se soporta.

- **SQLWARNING** es una abreviación para todos los códigos SQLSTATE que comienzan con **01**.
- **NOT FOUND** es una abreviación para todos los códigos SQLSTATE que comienzan con **02**.

- **SQLEXCEPTION** es una abreviación para todos los códigos SQLSTATE no tratados por **SQLWARNING** o **NOT FOUND**.

Además de los valores SQLSTATE , los códigos de error MySQL se soportan.

Por 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** es **3**, lo que muestra que MySQL se ha ejecutado al final del procedimiento. Si la línea **DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;** no está presente, MySQL habría tomado la ruta por defecto (**EXIT**) tras el segundo **INSERT** fallido debido a la restricción **PRIMARY KEY** , y **SELECT @x** habría retornado **2**.

19.2.11. Cursores

Se soportan 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.

Por 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
```

19.2.11.1. Declarar cursores

```
DECLARE cursor_name CURSOR FOR select_statement
```

Este comando declara un cursor. Pueden definirse varios cursores en una rutina, pero cada cursor en un bloque debe tener un nombre único.

El comando **SELECT** no puede tener una cláusula **INTO** .

19.2.11.2. Sentencia OPEN del cursor

```
OPEN cursor_name
```

Este comando abre un cursor declarado préviamente.

19.2.11.3. Sentencia de cursor FETCH

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

Este comando trata el siguiente registro (si existe) usando el cursor abierto especificado, y avanza el puntero del cursro.

19.2.11.4. Sentencia de cursor CLOSE

```
CLOSE cursor_name
```

Este comando cierra un cursor abierto préviamente.

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

19.2.12. 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.

19.2.12.1. 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**. *statement_list* puede consistir en varios comandos.

Tenga en cuenta que también hay una *función* **IF()**, que difiere del *commando* **IF** descrito aquí. Consulte Sección 12.2, “Funciones de control de flujo”.

19.2.12.2. 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.

Nota: La sitaxis de un *comando* **CASE** mostrado aquí para uso dentro de procedimientos almacenados difiere ligeramente de la *expresión* **CASE** SQL descrita en Sección 12.2, "Funciones de control de flujo". El comando **CASE** no puede tener una cláusula **ELSE NULL** y termina con **END CASE** en lugar de **END**.

19.2.12.3. 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.

19.2.12.4. Sentencia LEAVE

```
LEAVE label
```

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

19.2.12.5. 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
```

19.2.12.6. 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)
```

19.2.12.7. 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
```

19.3. Registro binario de procedimientos almacenados y disparadores

Esta sección describe cómo MySQL trata procedimientos almacenados (procedimientos o funciones) con respecto a 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 las modificaciones.

El log binario tiene dos propósitos importantes:

- La base de replicación es que el maestro envía los eventos contenidos en su log binario a sus esclavos, que ejecuta estos eventos para hacer los mismos cambios de datos que se hacen en el maestro. Consulte Sección 6.2, “Panorámica de la implementación de la replicación”.
- Ciertas operaciones de recuperación de datos necesitan usar el log binario. Tras hacer una restauración de un fichero de copia de seguridad, los eventos en el log binario que se guardaron tras hacer la copia de seguridad se re-ejecutan. Estos eventos actualizan la base de datos desde el punto de la copia de seguridad. Consulte Sección 5.8.2.2, “Usar copias de seguridad para una recuperación”.

El logueo de procedimientos almacenados difiere antes y después de MySQL 5.0.6. Antes de MySQL 5.0.6, 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;
```

Para 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 procedimientos almacenados no se replican correctamente. También significa que para operaciones de recuperación de datos, re-ejectuar eventos en el log binario no recupera procedimientos almacenados.

Para tratar estos temas de replicación y recuperación de datos, se cambió el logueo de procedimientos almacenados en MySQL 5.0.6. Sin embargo, este cambio provoca nuevos temas, que se presentan en la siguiente discusión.

A no ser que se diga lo contrario, estas notas asumen que no ha activado el logueo binario arrancando el servidor con la opción `--log-bin` . (Si el log binario no se activa, la replicación no es posible, ni está disponible el log binario para replicación de datos.) Consulte Sección 5.10.3, “El registro binario (Binary Log)”.

Las características de logueo binario para comandos de procedimientos almacenados se describen en la siguiente lista. Algunos de los iconos indican problemas que debería

conocer, pero en algunos casos, hay inicializaciones de servidor que puede modificar o soluciones que puede usar.

- Los comandos **CREATE PROCEDURE**, **CREATE FUNCTION**, **ALTER PROCEDURE**, y **ALTER FUNCTION** se escriben en el log binario, como lo son **CALL**, **DROP PROCEDURE**, y **DROP FUNCTION**.

  Sin embargo, hay implicaciones de seguridad para replicación: para crear una rutina, un usuario debe tener el permiso **CREATE ROUTINE** , pero un usuario que tenga este permiso puede escribir una rutina para realizar cualquier acción en un servidor esclavo ya que el flujo SQL en el esclavo corre con todos los permisos. Por ejemplo, si el maestro y el esclavo tienen valores de ID del servidor de 1 y 2 respectivamente, un usuario en el maestro puede crear e invocar procedimientos como sigue:

  ```
  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 que los creadores de procedimientos almacenados y funciones deben tener el permiso **SUPER** , además del permiso **CREATE ROUTINE** requerido. Similarmente, 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)
  ```

  Puede no querer forzar el requerimiento en los creadores de rutinas que deben tener el permiso **SUPER**. Por ejemplo, todos los usuarios con el permiso **CREATE ROUTINE** en su sistema pueden ser desarrolladores de aplicaciones con experiencia. Para desactivar el requerimiento de **SUPER**, cambie la variable de sistema global **log_bin_trust_routine_creators** a 1. Por defecto, esta variable vale 0, pero puede cambiarla así:

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

  Puede activar esta variable usando 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 ncesita el permiso **SUPER** para creación de rutinas.

- Una rutina no-determinista que realiza actualizaciones no es repetible, que 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, la creación y alteración de una rutina se rehúsa a no ser que la rutina se declare como determinista o que no modifique datos. Esto significa que cuando crea una rutina, debe declarar que es determinista o que no cambia datos. Dos conjuntos de características de rutinas se aplica aquí:

- **`DETERMINISTIC`** y **`NOT DETERMINISTIC`** indican si una rutina siempre produce el mismo resultado para entradas dadas. Por defecto es **`NOT DETERMINISTIC`** si no se da ninguna característica, así que debe especificar **`DETERMINISTIC`** explícitamente para declarar que la rutina es determinista.

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

- **`CONTAINS SQL`**, **`NO SQL`**, **`READS SQL DATA`**, y **`MODIFIES SQL`** proporciona 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 es **`CONTAINS SQL`** si ninguna de estas características se da.

Por defecto, para que un comando **`CREATE PROCEDURE`** o **`CREATE FUNCTION`** sea aceptado, **`DETERMINISTIC`** o **`NO SQL`** y **`READS SQL DATA`** deben especificarse explícitamente. De otro modo ocurre un 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 que la rutina sea determinista o que no modifique datos se elimina.

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, pero no en otro caso. Cuando una rutina que modifica datos falla, obtiene esta 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 tiene un 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, debe usar trablas transaccionales en la rutina y modificar las tablas dentro de transacciones.

Si usa la palabra clave **IGNORE** con **INSERT**, **DELETE**, o **UPDATE** para ignorar errores dentro de una rutina, una actualización parcial puede ocurrir sin producir error. Tales comandos se loguean y se replican normalmente.

- Si una función almacenada se invoca 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;
```

Dada esta definición, el comando siguiente modifica la tabla **t** porque **myfunc()** modifica **t**, pero el comando no se escribe en el log binario porque es un **SELECT**:

```
SELECT myfunc();
```

Una solución de este problema es invocar funciones que actualizan dentro de comandos que hacen actualizaciones. Tenga en cuenta que aunque el comando **DO** a veces se ejecuta como efecto colateral de evaluar una expresión, **DO** no es una solución aquí porque no está escrito en el log binario.

- Los comandos ejecutados dentro de una rutina no se escriben en el log binario. Suponga que ejectua los siguientes comandos:

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

Para este ejemplo, los comandos **CREATE PROCEDURE** y **CALL** aparecerán en el log binario. El comando **INSERT** no aparecerá. Esto arregla el problema que ocurre antes de MySQL 5.0.6 en que los comandos **CREATE PROCEDURE** y **CALL** no se loguearon y **INSERT** se logueó.

- En servidores esclavos, la siguiente limitación se aplica 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 son similares a los procedimientos almacenados, así que las notas precedentes también se aplican a disparadores con las siguientes excepciones: **CREATE TRIGGER** no tiene una característica **DETERMINISTIC** opcional, así que los disparadores se asumen como deterministas. Sin embargo, esta hipótesis puede ser inválida en algunos casos . Por ejemplo, la función **UUID()** no es determinista (y no replica). Debe ser cuidadoso acerca de usar tales funciones y disparadores.

Los disparadores actualmente no actualizan tablas, pero lo harán en el futuro. Por esta

razón, los mensajes de error similares a los de los procedimientos almacenados ocurren con **`CREATE TRIGGER`** si no tiene el permiso **`SUPER`** y **`log_bin_trust_routine_creators`** es 0.

Los temas tratados en esta sección son resultado del hecho que el logueo binario se hace a nivel de comandos SQL. MySQL 5.1 introducirá logueo binario a nivel de registro, lo que ocurre en un nivel más granular que especifica qué cambios hacer a registros individuales como resultado de ejecutar comandos SQL.

Anterior		Siguiente
Capítulo 18. Extensiones espaciales de MySQL	Inicio	Capítulo 20. Disparadores (triggers)