Capítulo 11. Stored Procedures e Funções

Stored procedures e funções são recursoso novos no MySQL versão 5.0. Uma stored procedure é um conjunto de comandos SQL que podem ser armazenados no servidor. Uma vez que isto tenha sido feito, os clientes não precisam de reenviar os comnados individuais mas pode fazer referência às stored procedures.

Stored procedures podem fornecer um aumento no desempenho já que menos informação precisa ser enviada entre o servidor e o cliente. O lado negativo é que isto aumenta a carga no sistema do servidor de banco de dados, já que a maior parte do trabalho é feita no servidor e menor parte é feita do lado do cliente (aplicação). E geralmente existem muitas máquinas clientes (como servidoes web) mas apenas um ou poucos servidores e banco de dados.

Stored procedures também permitem que você tenha bibliotecas de funções no servidor de banco de dados. No entanto, linguagens de aplicações modernas já permitem que isto seja feito internamente com classes, por exemplo, e usar estes recursos das linguagens de aplicações clientes é benéfico para o programador mesmo fora do escopo do banco de dados usado.

Situações onde stored procedures fazem sentido:

- Quando várias aplicações clientes são escritas em diferentes linguagens ou funcionam em diferentes plataformas, mas precisam realizar as mesmas operações de banco de dados.
- Quando a segurança é prioritária. Bancos, por exemplo, usam stored procedures para todas as operações comuns. Isto fornece um ambiente consistente e seguro, e procedures podem assegurar que cada operação seja registrada de forma apropriada. Neste tipo de condiguração, aplicações e usuários não conseguiriam nenhuma acesso as tabelas do banco de dados diretamente, mas apenas podem executar stored procedures específicas.

O MySQL segue a sintaxe SQL:2003 para stored procedures, que também é usada pelo DB2 da IBM. Suporte para compatibilidade de outras linguagens de stored procedures (PL/SQL, T-SQL) podem ser adicionadas posteriormente.

A implementação do MySQL de stored procedures ainda está em progresso. Todas as sintaxes descritas neste capítulo são suportadas e qualquer limitação e extensão está documentada de forma aprorpiada.

Stored procedures exigem a tabela `proc` na banco de dados `mysql`. Esta tabela é criada durante a instalação do MySQL 5.0. Se você ataulizar para o MySQL 5.0 a partir de uma versão anterior, certifique de atualizar a sua tabela de permissão para ter certeza que a tabela `proc` existe. See Secção 2.5.6, “Atualizando a Tabela de Permissões”.

11.1. Sintaxe de Stored Procedure

Stored procedures e funções são rotinas criadas com as instruções `CREATE PROCEDURE` e `CREATE FUNCTION`. Um procedimento é chamado usando uma instrução `CALL` e só pode passar valores de retorno usadno variáveis de saída. Funções podem retornar um valor escalar e pode ser chamadas de dentro de uma instrução como quqlquer outra função (isto é, chamando o nome da funcao). Rotinas armazenadas podem chamar outras rotinas armazenadas. Uma rotina pode ser tanto um procedimento como uma função.

Atualmentem o MySQL só preserva o contexto para o banco de dados padrão. Isto é, se você usar `USE dbname` dentro de um procedimento, o banco de dados original é restaurado depois da saída da rotina. Uma rotina herda o banco de dados padrão de quem a chama, assim geralmente as rotinas devem utilizar uma instrução `USE dbname`, ou especifique todas as tabelas com uma referência de banco de dados explicita, ex. `dbname.tablename`.

O MySQL suporta uma extensão muito útil que permite o uso da instrução regular `SELECT` (isto é, sem usar cursores ou variáveis locais) dentro de uma stored procedure. O resultado de tal consulta é simplesmente enviado diretamente para o cliente. Várias instruções `SELECT` geram vária resultados, assim o cliente deve usar um biblioteca cliente do MySQL que suporta vários resultados. Isto significa que o cliente deve usar uma biblioteca cliente a partir de uma versão do MySQL mais recente que 4.1, pelo menos.

A seção seguinte descreve a sintaxe usada para criar, alterar, remover e condultar stored procedures e funções.

11.1.1. Manutenção de Stored Procedures

11.1.1.1. `CREATE PROCEDURE` e `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
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT string

routine_body:
  Valid SQL procedure statement(s)
```

A cláusula `RETURNS` pode ser especificada apenas por uma `FUNCTION`. É usada para indicar o tipo de retorno da função, e o corpo da função deve conter uma instrução `RETURN value`.

A lista de parâmetros entre parenteses deve estar sempre presente. Se não houver parâmetros, uma lista de parâmetros vazia de `()` deve ser usada. Cada parâmetro é um parâmetro `IN` por padrão. Para especificar outro tipo de parâmetro, use a palavra chave `OUT` ou `INOUT` antes do nome do parâmetro. Especificar `IN`, `OUT` ou `INOUT` só é valido para uma `PROCEDURE`.

A instrução `CREATE FUNCTION` é usada em versão novas do MySQL para suporte a UDFs (User Defined Functions - Funções Definidas pelo Usuário). See Secção 14.2, “Adicionando Novas Funções ao MySQL”. As UDFs continuam a ser suportadas, mesmo com a existencia de stored functions. Uma UDF pode ser considerada como uma stored function externa. No entanto, note que stored functions compartilham os seus namespace com as `UDF`s.

Um framework para stored procedures externas serão introduzidas em um futuro próxima. Isto permitirá que você escreva stored procedures em outras linguagens além de SQL. Provavelmente, uma das primeiras linguagens a ser suportada sea PHP, já que o mecanismo do PHP é pequeno, seguro com threads e pode facilmente ser embutido. Como o framework será publico, é esperado que muitas outras linguagens também sejam suportadas.

Uma função é considerada ``deterministica'' se ela sempre retorna o mesmo resultado para os mesmos parâmetros de entrada, e ``não deterministica'' caso contrário. O otimizado pode usar este fato. Atualmente, a característica `DETERMINISTIC` é aceita, mas ainda não é usada.

A característica `SQL SECURITY` pode ser usada para especificar se a rotina deve ser executada usando as permissões do usuário que criou a rotina, ou o usuário que a chamou. O valor padrão é `DEFINER`. Este recurso é novo no SQL:2003.

O MySQL ainda não usa o privilégio `GRANT EXECUTE`. Assim ,por enquanto, se um procedimento `p1()` chama a tabela `t1`,o usuário deve ter privilégios na tabela `t1` para chamar o procedimento `p1()` com sucesso.

MySQL stores the `SQL_MODE` settings in effect at the time a routine is created, and will always execute routines with these settings in force.

A cláusula `COMMENT` é uma extensão do MySQL, e pode ser usada para descrever o stored procedure. Esta informação é exibida pelas instruções `SHOW CREATE PROCEDURE` e `SHOW CREATE FUNCTION`.

O MySQL permite rotinas contendo instruções DDL (como `CREATE` e `DROP`) e instruções de transação SQL (como `COMMIT`). Isto não é exigido por padrão e depende de especificações de implementação.

NOTA: Atualmente, stored `FUNCTION`s não podem conter referências às tabelas. Note que isto inclui algumas instruções `SET`, mas exclui algumas instruções `SELECT`. Esta limitação será retirada assim que possível.

A seguir temos um exemplo de uma stored procedure simples que usa um parâmetro `OUT`. O exemplo usa o comando `delimiter` do cliente `mysql` para alterar o delimitador de instrução para antes da definição do procedure. Isto permite que o delimitador ‘`;`’ usado no corpo de procedure seja passado para o servidor em vez de ser interpretado pelo `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> CALL simpleproc(@a)|
Query OK, 0 rows affected (0.00 sec)

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

A seguir esta um exemplo de uma função que utiliza um parametro, realiza uma operação usando uma função SQL e retorna o 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> SELECT hello('world')|
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

11.1.1.2. `ALTER PROCEDURE` e `ALTER FUNCTION`

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

characteristic:
    NAME newname
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT string
```

Este comando pode ser usado para renomear uma stored procedure ou function, e para alterar suas características. Mais de uma mudança pode ser especificada em uma instrução `ALTER PROCEDURE` ou `ALTER FUNCTION`.

11.1.1.3. `DROP PROCEDURE` e `DROP FUNCTION`

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

Este comando é usado para deletar uma stored procedure ou function. Isto é, a rotina especificada é removida do servidor.

A cláusula `IF EXISTS` é uma extensão do MySQL. Ela previne que um erro ocorra se o procedimento ou função não existe. Um aviso é produzido e pode ser vizualizado com `SHOW WARNINGS`.

11.1.1.4. `SHOW CREATE PROCEDURE` e `SHOW CREATE FUNCTION`

```
SHOW CREATE PROCEDURE | FUNCTION sp_name
```

Este comando é uma extensão do MySQL. De forma similar a `SHOW CREATE TABLE`, ele retorna a string exata que pode ser usada para recriar a rotina chamada.

11.1.2. `SHOW PROCEDURE STATUS` e `SHOW FUNCTION STATUS`

```
SHOW PROCEDURE | FUNCTION STATUS [LIKE pattern]
```

Este comando é uma extensão do MySQL. Ele retorna características da rotina, tais como nome, tipo, quem criou, datas de modificação e criação. Se nenhum padrão é especificado, a informação de todas as stored procedures ou todas as stored functions é listado, dependendo de qual instrução você utiliza.

11.1.3. `CALL`

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

O comando `CALL` é usado para chamar uma rotina que foi definida anteriormente com `CREATE PROCEDURE`.

11.1.4. `BEGIN ... END` Compound Statement

```
[begin_label:] BEGIN
 statement(s)
END [end_label]
```

As rotinas armazenadas podem conter várias instruções, usando um instrução `BEGIN ... END`.

`begin_label` e `end_label` devem ser os mesmos, se ambos forem especificados.

Notem que a cláusula opcional `[NOT] ATOMIC` ainda não é suportada. Isto significa que nenhum savepoint de transação é definido no início do bloco da instrução e a cláusula `BEGIN` usada neste contexto não tem nenhum efeito no transação atual.

Várias instruções exigem que um cliente tenha permissão para enviar strings de querys contendo '`;`'. Isto é tratado no cliente mysql

e linha de comando com o comando `delimiter`. Alterando o delimitador '`;`' do final da consulta (por exemplo, para '`|`') permite que '`;`' seja usado no corpo de uma rotina.

11.1.5. Instrução `DECLARE`

A instrução `DECLARE` é usada para definir vários itens locais para uma rotina: variaveis locais (see Secção 11.1.6, "Variables in Stored Procedures"), condições e handlers (see Secção 11.1.7, "Condições e Handlers") e cursors (see Secção 11.1.8, "Cursors"). As instruções `SIGNAL` e `RESIGNAL` ainda não são suportadas.

`DECLARE` só pode ser usada dentro de uma instrução composta `BEGIN ... END` e deve estar no início, antes de qualquer outra instrução.

11.1.6. Variables in Stored Procedures

Você pode declarar e usar variáveis dentro de uma rotina.

11.1.6.1. Variável Local `DECLARE`

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

Este comando é usado para declarar variáveis locais. O escopo de uma variável está dentro do bloco `BEGIN ... END`.

11.1.6.2. Instrução Variável `SET`

```
SET variable = expression [,...]
```

A instrução `SET` em stored procedures é uma versão estendida do comando `SET` geral. As variáveis indicadas podem ser aquelas declaradas dentro de uma rotina, ou variáveis globais do servidor.

A instrução `SET` em stored procedures é implementada como parte da sintaxe pré-existente de `SET`. Isto permite uma sintaxe estendida de `SET a=x, b=y, ...` onde variáveis de tipos diferentes (variaveis declaradas localamente, variáveis do servidor e variáveis globais e de sessão do servidor) podem estar misturadas. Ela também permite combinações de variáveis locais e algumas opções que só fazem sentido para variáveis globais e de sistema; neste caso as opções são aceitas mas ignoradas.

11.1.6.3. Instrução `SELECT ... INTO`

```
SELECT column[,...] INTO variable[,...] table_expression
```

Esta sintaxe de `SELECT` armazena colunas selecionadas diretamente nas variáveis. Por esta razão, apenas uma linha pode ser recuperada. Esta instrução também é extremamente útil quando usada em conjunto com cursores.

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

11.1.7. Condições e Handlers

Certas condições podem exigir tratamento específico. Estas condições podem ser relacionadas a erros, bem como controle de fluxo geral dentro da rotina.

11.1.7.1. `DECLARE` Conditions

```
DECLARE condition_name CONDITION FOR condition_value

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

Esta instrução especifica condições que necessitarão de tratamento especial. Ela associa um nome com uma condição de erro específica. O nome pode ser subsequentemente usado em uma instrução `DECLARE HANDLER`. See Secção 11.1.7.2, "`DECLARE` Handlers".

Além dos valores SQLSTATE, códigos de erro do MySQL também são suportados.

11.1.7.2. `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
```

Esta instrução especifica handlers para lidar com uma ou mais condições. Se uma dessas condições ocorrer, a instrução especificada é executada.

Para um handler `CONTINUE`, a execução das rotinas atuais continuam depois da instrução handler. Para um handler `EXIT`, a execução da rotina atual é terminada. O `handler_type` `UNDO` ainda não é suportado. Atualmente o `UNDO` se comporta como `CONTINUE`.

- `SQLWARNING` is shorthand for all SQLSTATE codes that begin with 01.
- `NOT FOUND` is shorthand for all SQLSTATE codes that begin with 02.
- `EXCEPTION` is shorthand for all SQLSTATE codes not caught by `SQLWARNING` or `NOT FOUND`.

Além dos valores SQLSTATE, códigos de erro do MySQL também são suportados.

Por exemplo:

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

mysql> delimiter |

ysql> CREATE PROCEDURE handlerdemo ()
    -> BEGIN
    ->   DECLARE CONTINUE HANDLER FOR '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)
```

Notice that `@x` is 3, which shows that MySQL executed to the end of the procedure. If the line `DECLARE CONTINUE HANDLER FOR '23000' SET @x2 = 1;` had not been present, MySQL would have taken the default (`EXIT`) path after the second `INSERT` failed due to the `PRIMARY KEY` constraint, and `SELECT @x` would have returned 2.

11.1.8. Cursors

Simple cursors are supported inside stored procedures and functions. The syntax is as in embedded SQL. Cursors are currently asensitive, read-only, and non-scrolling. Asensitive means that the server may or may not make a copy of its result table.

For example:

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE a CHAR(16);
  DECLARE b,c INT;

  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
```

11.1.8.1. Declaring Cursors

```
DECLARE cursor_name CURSOR FOR sql_statement
```

Multiple cursors may be defined in a routine, but each must have a unique name.

11.1.8.2. Cursor `OPEN` Statement

```
OPEN cursor_name
```

This statement opens a previously declared cursor.

11.1.8.3. Cursor `FETCH` Statement

```
FETCH cursor_name
```

This statement fetches the next row (if a row exists) using the specified open cursor, and advances the cursor pointer.

11.1.8.4. Cursor `CLOSE` Statement

```
CLOSE cursor_name
```

This statement closes a previously opened cursor.

11.1.9. Flow Control Constructs

The `IF`, `CASE`, `LOOP`, `WHILE`, `ITERATE`, and `LEAVE` constructs are fully implemented.

These constructs may each contain either a single statement, or a block of statements using the `BEGIN ... END` compound statement. Constructs may be nested.

`FOR` loops are not currently supported.

11.1.9.1. `IF` Statement

```
IF search_condition THEN statement(s)
[ELSEIF search_condition THEN statement(s)]
...
[ELSE statement(s)]
END IF
```

`IF` implements a basic conditional construct. If the `search_condition` evaluates to true, the corresponding SQL statement is executed. If no `search_condition` matches, the statement in the `ELSE` clause is executed.

Please note that there is also an `IF()` function. See Secção 6.3.1.4, “Funções de Fluxo de Controle”.

11.1.9.2. `CASE` Statement

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

or

```
CASE
  WHEN search_condition THEN statement
```

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

`CASE` implements a complex conditional construct. If a `search_condition` evaluates to true, the corresponding SQL statement is executed. If no search condition matches, the statement in the `ELSE` clause is executed.

Please note that the syntax of a `CASE` statement inside a stored procedure differs slightly from that of the SQL `CASE` expression. The `CASE` statement can not have an `ELSE NULL` clause, and the construct is terminated with `END CASE` instead of `END`. See Secção 6.3.1.4, “Funções de Fluxo de Controle”.

11.1.9.3. `LOOP` Statement

```
[begin_label:] LOOP
  statement(s)
END LOOP [end_label]
```

`LOOP` implements a simple loop construct, enabling repeated execution of a particular statement or group of statements. The statements within the loop are repeated until the loop is exited, usually this is accomplished with a `LEAVE` statement.

`begin_label` and `end_label` must be the same, if both are specified.

11.1.9.4. `LEAVE` Statement

```
LEAVE label
```

This statement is used to exit any flow control construct.

11.1.9.5. `ITERATE` Statement

```
ITERATE label
```

`ITERATE` can only appear within `LOOP`, `REPEAT`, and `WHILE` statements. `ITERATE` means ``do the loop iteration again."

For example:

```
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
```

11.1.9.6. `REPEAT` Statement

```
[begin_label:] REPEAT
  statement(s)
UNTIL search_condition
END REPEAT [end_label]
```

The statements within a `REPEAT` statement are repeated until the `search_condition` is true.

`begin_label` and `end_label` must be the same, if both are specified.

For example:

```
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)
```

11.1.9.7. `WHILE` Statement

```
[begin_label:] WHILE search_condition DO
  statement(s)
END WHILE [end_label]
```

The statements within a `WHILE` statement are repeated as long as the `search_condition` is true.

`begin_label` and `end_label` must be the same, if both are specified.

For example:

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

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