This lecture

We continue our coverage of the fundamentals of SQL/MySQL with stored routines.

Stored Routines

A stored routine is a named set of SQL statements that is stored on the server and which can be initiated by a single call.

Stored routines are further subdivided into procedures which do not return anything (although they can assign values to variables) and functions that return values to the caller.

In MySQL 5.0.x, functions have a major limitation in that they cannot access tables, so they are essentially just a way to add special-purpose calculations to MySQL.

Rationale for stored routines

A stored routine is maintained on the server which has various consequences both positive and negative:

- A complex sequence of SQL statements can be prepared once by a professional DBA and then made available to all client programs
- Stored routines can access confidential or sensitive tables without exposing them to client programs
- Processing becomes more centralized with the server taking on a greater computational load
**Basic Syntax**

The basic syntax for creating the simplest possible procedure, one with no parameters and consisting of a single SQL statement is as follows:

```sql
CREATE PROCEDURE myproc()
    /* An SQL statement */
```

For example, in the world database we could issue the command:

```sql
CREATE PROCEDURE listCapitals()
    SELECT C.name, T.name
    FROM country C, city T
    WHERE C.capital = T.id;
```

(GF Royle 2006-8, N Spadaccini 2008)

SQL IV 5 / 25

---

**Calling a user-defined procedure**

```sql
CALL listCapitals();
```

+---------------------------+---------------+
<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>Kabul</td>
</tr>
<tr>
<td>Netherlands</td>
<td>Amsterdam</td>
</tr>
<tr>
<td>Netherlands Antilles</td>
<td>Willemstad</td>
</tr>
</tbody>
</table>

A stored procedure “belongs” to a specific database (the one in use when the `CREATE PROCEDURE` command was issued, and so this procedure belongs to the world database.

(GF Royle 2006-8, N Spadaccini 2008)

SQL IV 6 / 25

---

**Procedure parameters**

In order to do anything more useful than pure textual replacement, a procedure needs to have parameters that the user can specify on calling.

```sql
CREATE PROCEDURE listOneCapital(cntry VARCHAR(50))
    SELECT C.name, T.name
    FROM country C, city T
    WHERE C.capital = T.id
    AND C.name = cntry;
```

This procedure has one parameter called `cntry` which is of type `VARCHAR(50)`.

(GF Royle 2006-8, N Spadaccini 2008)

SQL IV 7 / 25

---

**Calling the procedure**

As usual, when the procedure is called the caller specifies an actual argument which is used in place of the formal parameter.

```sql
CALL listOneCapital('Australia');
```

+-----------+----------+
<table>
<thead>
<tr>
<th>name</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>Canberra</td>
</tr>
</tbody>
</table>

(GF Royle 2006-8, N Spadaccini 2008)

SQL IV 8 / 25
Output Parameters

A procedure has no RETURN statement and hence cannot return a value to the caller. However the caller can specify a user-variable in the parameter list to which the procedure can assign a value.

CREATE PROCEDURE regionPop(rgn TEXT, OUT rpop INT)
    SELECT SUM(population)
    FROM country C
    WHERE C.region = rgn
    INTO rpop;

The output parameter rpop is indicated by the keyword OUT and the SELECT statement performs the selection INTO the variable.

(GF Royle 2006-8, N Spadaccini 2008)

Using output parameters

When this procedure is called the user must give a variable name for the second argument

CALL regionPop(’North America’, @napop);

Nothing appears on the terminal, but the variable @napop has had a value assigned to it, which can subsequently be used.

SELECT @napop;
+-----------+
| @napop    |
| 309632000 |
+-----------+

(GF Royle 2006-8, N Spadaccini 2008)

Multiple statements

To enhance our procedures further we need to be able to perform a sequence of SQL statements inside a procedure, not just a single statement.

This can be done by putting the statements between BEGIN and END.

CREATE PROCEDURE myproc()
BEGIN
    /* A whole bunch of MySQL statements */
END

One problem that immediately arises is how to terminate each of the statements inside the BEGIN/END area — if we just use the semicolon then MySQL will think that the procedure definition has terminated.

(GF Royle 2006-8, N Spadaccini 2008)

Temporarily change delimiters

The solution to this is to temporarily change the delimeter so that we can enter the entire procedure.

DELIMITER ++
CREATE PROCEDURE myproc()
BEGIN
    /* A whole bunch of MySQL statements */
    /* each terminated with the usual semicolon */
END++
DELIMITER ;

The first line temporarily changes the delimiter to ++, then the entire procedure is entered, and finally the delimiter is changed back again.

(GF Royle 2006-8, N Spadaccini 2008)
Procedure Variables

Of course, in order to use multiple statements effectively it helps to be able to use “local variables” within the procedure.1

CREATE PROCEDURE regionSummary(rgn TEXT)
BEGIN
    DECLARE rp INT;
    CALL regionPop(rgn, rp);
END

This fragment creates a local variable called \( rp \) and then calls the previously defined procedure to assign the total population of the specified region to that variable.

1Henceforth I will not include the DELIMITER statements

Multiple statements

We can complete this procedure fragment by using the variable that we have just evaluated in a subsequent SQL statement.

CREATE PROCEDURE regionSummary(rgn TEXT)
BEGIN
    DECLARE rp INT;
    CALL regionPop(rgn,rp);
    SELECT C.name, C.population,
           C.population / rp * 100 as perc
    FROM country C
    WHERE C.region = rgn
    ORDER BY C.population
    DESC LIMIT 5;
END

This has simply added one more SELECT statement that performs another query to list the five most populous countries in that region.

Calling this procedure

mysql> CALL regionSummary("Caribbean");
+--------------------+------------+---------+
| name               | population | perc    |
|--------------------+------------+---------|
| Cuba               | 11201000   | 29.3681 |
| Dominican Republic | 8495000    | 22.2732 |
| Haiti              | 8222000    | 21.5574 |
| Puerto Rico        | 3869000    | 10.1442 |
| Jamaica            | 2583000    | 6.7724  |
+--------------------+------------+---------+

Other constructs

In addition to this basic functionality, stored procedures can also perform rudimentary selection and repetition with constructs such as

- IF-THEN-ELSE
- WHILE...END WHILE
- REPEAT...END REPEAT
- LOOP...END LOOP
Largest and Smallest

Suppose that instead of the top five countries for the specified region, we wanted to list the **most populous** and **least populous** countries. We could do this with three `SELECT` statements — one to find the minimum and maximum country populations in that region, then one each to find which country has the minimum and the maximum population. However we could do this with just one `SELECT` statement provided we could process the results afterwards.

---

Cursors

A cursor is essentially a mechanism to store the results of a query, and to process the results row-by-row. In MySQL 5.0.x, cursor support is very limited and currently cursors can only be created in procedures and they can only be processed row-by-row from start to finish. Cursors essentially support only four statements:

- `DECLARE...CURSOR` for declares a cursor
- `OPEN...` opens the cursor
- `FETCH...INTO` fetches the current row for processing
- `CLOSE...` closes the cursor

---

Cursor control

```
CREATE PROCEDURE regionLimits(rgn TEXT)
BEGIN
    DECLARE regionOnly CURSOR FOR
        SELECT C.name, C.population
        FROM country C
        WHERE region = rgn;
    OPEN regionOnly;
    /* process the rows */
    CLOSE regionOnly;
END
```

---

How many rows?

We will use a loop to process each row, and so we need to know how many rows the cursor contains; this can be found from the MySQL function `FOUND_ROWS()` which returns the number of rows that the last query found.

```
DECLARE numRows INT;
DECLARE numDone INT;
/* Declare cursor */
OPEN regionOnly;
SELECT FOUND_ROWS() INTO numRows;
WHILE numDone < numRows DO
    /* Process a row */
    SET numDone = numDone + 1;
END WHILE;
```
Storing max and min

In order to use the cursor to process each row, we need to have variables to store the name and population of the most populous and least populous countries and variables for the contents of each row. So the declaration section will need to have the following added to it:

DECLARE minP INT;
DECLARE maxP INT;
DECLARE minC VARCHAR(50);
DECLARE maxC VARCHAR(50);
DECLARE cname VARCHAR(50);
DECLARE cpop INT;

Initializing

These variables need to be initialized to have the values of the first city in the list. So immediately after the `SELECT FOUND_ROWS()` INTO `numRows` we put

```sql
FETCH regionOnly INTO cname, cpop;
SET minP = cpop;
SET maxP = cpop;
SET minC = cname;
SET maxC = cname;
SET numDone = 1;
```

Inside the loop

In the loop, we fetch the contents of the next row and compare them to the existing minimum/maximum values:

```sql
WHILE numDone < numRows DO
    FETCH regionOnly INTO cname, cpop;
    IF (cpop < minP) THEN
        SET minP = cpop;
        SET minC = cname;
    END IF;
    IF (cpop > maxP) THEN
        SET maxP = cpop;
        SET maxC = cname;
    END IF;
    SET numDone = numDone + 1;
END WHILE;
```

Finally

And finally after the loop we “print” the output.

```sql
SELECT minC as smallest,
minP as smallestPop,
maxC as largest,
maxP as largestPop;
```

The output from the whole procedure is then something like

```
<table>
<thead>
<tr>
<th>smallest</th>
<th>smallestPop</th>
<th>largest</th>
<th>largestPop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anguilla</td>
<td>8000</td>
<td>Cuba</td>
<td>11201000</td>
</tr>
</tbody>
</table>
```

(GF Royle 2006-8, N Spadaccini 2008)
Conclusion

Although MySQL supports stored procedures and functions, the programming tools available are very rudimentary and awkward compared to a general-purpose programming language.

Therefore while stored routines are extremely useful when they consist of things that can be expressed easily in SQL, they become very awkward when performing general processing.

Therefore in the absence of a compelling reason (e.g. security) to use stored routines, most non-SQL processing should be performed at the client, and not on the server.