Databases - Stored Routines

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

This lecture

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

(This material will appear in the remaining lab sessions, but will not be on the final exam.)

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.

Normally, we imagine the stored routine as being *written* by the DBA and *called* by the client programs.

Stored routines are further subdivided into:

- Stored *procedures* do not return anything
  Stored procedures can however assign values to variables etc.
- Stored *functions* that return *values* to the client

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:

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

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

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

So the procedure's *name* is `listCapitals` and its *body* is the single `SELECT` statement.

Calling a user-defined procedure

```
CALL listCapitals();
+-----------------------------+-----------------------------------+
| name                        | name                              |
+-----------------------------+-----------------------------------+
| Aruba                       | Oranjestad                        |
| Afghanistan                 | Kabul                             |
| Angola                      | Luanda                            |
| Anguilla                    | The Valley                        |
```

A stored procedure “belongs” to a specific database, namely the one that was being used when `CREATE PROCEDURE` command was issued.

Procedure parameters

In order to do anything more useful than just saving typing, a procedure will have *parameters* that the user will *specify* on calling the procedure

```
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)`.

Calling the procedure

When the procedure is *called* the caller specifies an *actual value*, known as the *argument* to the procedure.

```
CALL listOneCapital(’Uganda’);
+--------+---------+
| name   | name    |
+--------+---------+
| Uganda | Kampala |
+--------+---------+
```

Inside the procedure, the argument — in this case `’Uganda’` — will be used wherever the variable `cntry` occurs.

Output Parameters

A procedure has no `RETURN` statement and doesn't *return* a value to the caller.

However the caller can specify a *user-variable* in the parameter list and the procedure can assign a value to that variable.

```
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.

Using output parameters

When this procedure is *called* the user must

- Specify an actual value for each *input* variable
- Specify a variable name for each *output* variable

```
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 |
+-----------+
```

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.

Temporarily change delimiters

The solution to this is to temporarily *change* the delimiter 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.

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.

[1] Henceforth 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*.

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.

The algorithm

The basic idea is simple:

Suppose I were to read out a sequence of 750 numbers, and then ask you what the biggest number was? How would you approach this without remembering all the 750 numbers?

The algorithm

The basic idea is simple:

Suppose I were to read out a sequence of 750 numbers, and then ask you what the biggest number was? How would you approach this without remembering all the 750 numbers?

Always remember the *“biggest so far”*, and as each new number comes along, compare it to the number that is being remembered and update the remembered number only if the new number is bigger.

A stored procedure

```
DELIMITER ++
CREATE PROCEDURE regionLimits(rgn TEXT)
BEGIN

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

  DECLARE regionOnly CURSOR FOR
    SELECT C.name, C.population
    FROM country C
    WHERE region = rgn;

  OPEN regionOnly;
  SELECT FOUND_ROWS() INTO numRows;

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

  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;

  CLOSE regionOnly;

  SELECT minC as smallest,
      minP as smallestPop,
      maxC as largest,
      maxP as largestPop;
END++
DELIMITER ;
```

Cursors

A *cursor* is essentially a mechanism to *store* the results of a query, and then process the results row-by-row.

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

Cursors are also the primary mechanism by which *client programs* interact with the database.

Lifecycle of a cursor — conception

Think of a cursor as a machine that *runs a SQL query* and returns the results of this query to you *one row at a time* as you ask for them.

A cursor springs into existence when execution of the stored procedure reaches a `DECLARE...CURSOR` statement — it does not *run* the statement at this point, but just remembers it.

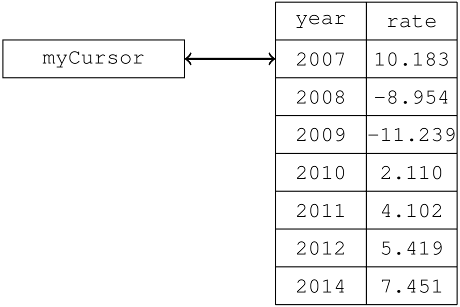
```
DECLARE myCursor CURSOR FOR
  SELECT year, rate
  FROM Investments;
```

myCursor

Lifecycle of a cursor — birth

At some later stage, when the `OPEN` statement is executed, the cursor actually *runs* the query.

```
OPEN myCursor;
```



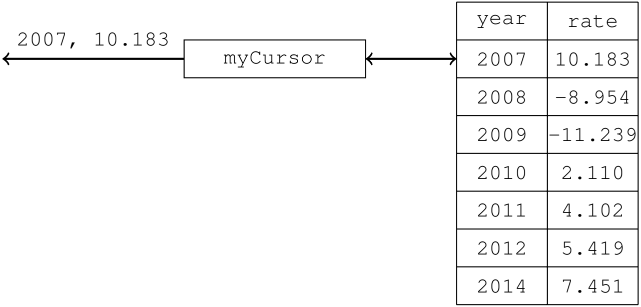
year	rate
2007	10.183
2008	-8.954
2009	-11.239
2010	2.110
2011	4.102
2012	5.419
2014	7.451

It does not return the result of the query, but just *remembers* it.

Lifecycle of a cursor — working life

The data is extracted from the cursor — one row at a time — using `FETCH`. Each `FETCH` call causes it to return *whichever row it is pointing to*

```
FETCH myCursor INTO a, b;
```

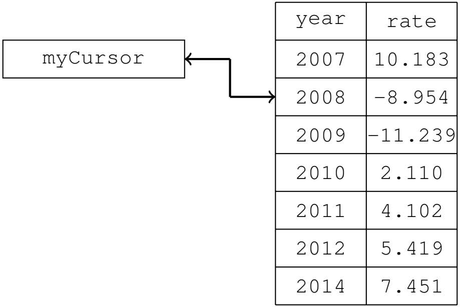


year	rate
2007	10.183
2008	-8.954
2009	-11.239
2010	2.110
2011	4.102
2012	5.419
2014	7.451

Lifecycle of a cursor — working life

The data is extracted from the cursor — one row at a time — using `FETCH`. Each `FETCH` call causes it to return *whichever row it is pointing to* and then shift the pointer to the next row.

```
FETCH myCursor INTO a, b;
```

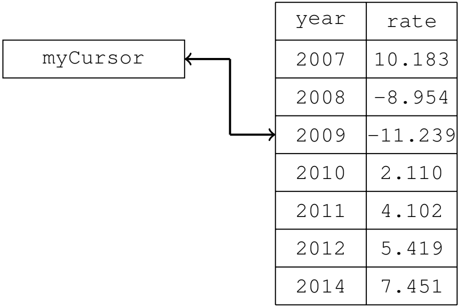


year	rate
2007	10.183
2008	-8.954
2009	-11.239
2010	2.110
2011	4.102
2012	5.419
2014	7.451

Lifecycle of a cursor — working life

The data is extracted from the cursor — one row at a time — using `FETCH`. Each `FETCH` call causes it to return *whichever row it is pointing to* and then shift the pointer to the next row.

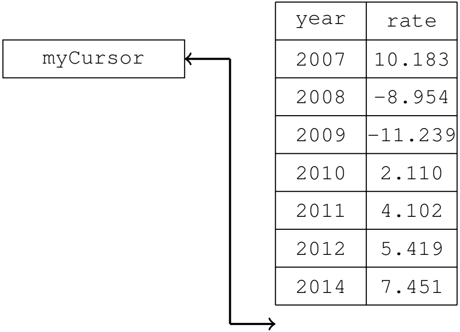
```
FETCH myCursor INTO a, b;
```



year	rate
2007	10.183
2008	-8.954
2009	-11.239
2010	2.110
2011	4.102
2012	5.419
2014	7.451

Lifecycle of a cursor — death

Eventually, after multiple `FETCH` statements, the cursor will have gone through all of the rows and have no more data left to return.



year	rate
2007	10.183
2008	-8.954
2009	-11.239
2010	2.110
2011	4.102
2012	5.419
2014	7.451

`CLOSE myCursor;` closes the cursor and reclaims the memory and/or other resources it is using.

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 and open cursor */


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 found so far” and “least populous found so far”, along with variables for the values extracted from each row as it is processed.

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 are initialized with the values from the *first row* (after we have read just one row, then the name and population are the best-so-far for both maximum and minimum population.)

So immediately after the `SELECT FOUND_ROWS() INTO numRows` we put

```
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:

```
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.

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

The output from the whole procedure is then something like

```
CALL regionLimits("Caribbean");
+----------+-------------+---------+------------+
| smallest | smallestPop | largest | largestPop |
+----------+-------------+---------+------------+
| Anguilla |        8000 | Cuba    |   11201000 |
+----------+-------------+---------+------------+
```

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.