Cal Poly
Spring 2013

CPE/CSC 365
Introduction to Database Systems

Eriq Augustine

Stored Routines
Procedures and Functions

Stored Routines

Stored routines are pieces of code that are stored in the database. They can be used to execute code in the database. Stored routines can be used as a validation or access control mechanism. They can also be used to consolidate and centralize logic that would otherwise appear in the application layer, moving the logic closer to the data.
This handout will cover a subset of all the functionality available to stored routines.

Procedures vs. Functions

Stored routines come in two different flavors: procedures and functions. There are two primary differences between them:

1. Functions can only return one value while procedures can return many through the use of out parameters and implicit SELECTs.
2. Functions can be invoked inside SELECTs just like any other MySQL function (like RAND() or NOW()), while procedures need to be invoked separately.

Pros & Cons

Pros

- Shares logic between all applications that use the database.
- Provides an abstraction layer between the database and other applications. Using stored routines, the database can change under the application without the application having to change.
- Less network traffic because additional logic can be handled on the server.

Cons

- Increased load on server.
- More difficult to maintain program logic because logic is now in the application and the database.
- Migration to another DBMS (not MySQL) can be more difficult because much of the syntax and functionality is MySQL exclusive.

Delimiters

In stored routines, you are allowed to have multiple SQL statements. However, declaring the stored routine itself is a statement. Therefore, you will need to change the delimiter while defining routines.

You will want to choose a delimiter that is unlikely to occur within the routine. I like using "$$". "//" and "|" is also popular. For the rest of this lecture, assume "$$" is the delimiter.

Delimiters can be set with the DELIMITER statement:

```
DELIMITER $$
```

After you finish your routine, make sure to set the delimiter back to semicolon:

```
DELIMITER ;
```

Allowable Statements

Only a subset of SQL statements are allowable in stored routines. The allowable subset differs between SQL and MySQL, you will need to be aware of the difference if you want to write portable code.
Allowable in both SQL and MySQL:

- DML
- Transaction statements: START TRANSACTION, COMMIT, ROLLBACK.
- SET

MySQL only:

- DDL
- Direct SELECT:

```
SELECT 'a';
```

Explicitly disallowed in MySQL:

- Any DDL statement that modifies a routine.

Creating Routines

CREATE PROCEDURE

```
CREATE PROCEDURE <name>([[IN | OUT | INOUT] <parameter name> <parameter type> [, ...]])
[caracteristic ...]
BEGIN
  <routine body>
END$$
```

CREATE FUNCTION

```
CREATE FUNCTION <name>(<parameter name> <parameter type> [, ...]]) RETURNS <type>
[caracteristic ...]
BEGIN
  <routine body>
END$$
```

Parameter Types

Procedures support three different types of parameters.

- IN: The parameter is only used to pass data to the routine. This is the classical parameter type, the default type for procedures, and the type used by functions.
- OUT: The parameter is only used to pass information back to the caller. The initial value for an OUT parameter is always NULL.
- INOUT: The parameter may be used to pass a value to the function and back to the caller. The initial value is whatever the callers passes and any modifications to it are seen by the caller.

Characteristics

- COMMENT 'string': Allows a comment to be stored along with the routine.
- LANGUAGE SQL: Indicates the language that the routine is written in. Ignored in MySQL, only SQL is supported.
- [NOT] DETERMINISTIC]: A DETERMINISTIC routine will always return the same result for the same input data. The validity of declaring a routine deterministic is not assessed by the DBMS the judgement of the user is trusted.

  Doing a SELECT or using NOW() or RAND() can make a routine non-deterministic.

  Mistakenly declaring a non-deterministic routine as DETERMINISTIC can result in unexpected outputs because of optimizations made by the DBMS. While mistakenly declaring a deterministic routine as NOT DETERMINISTIC can result in a decrease in performance because optimizations are not taken.
- CONTAINS SQL: Indicates that the routine contains SQL statements. (Advisory only, does not affect the evaluation of the routine).
- NO SQL: Indicates that the routine contains no SQL statements. (Advisory only, does not affect the evaluation of the routine).
- READS SQL DATA: Indicates that the routine is read only. (Advisory only, does not affect the evaluation of the routine).
- MODIFIES SQL DATA: Indicates that the routine may write data. (Advisory only, does not affect the evaluation of the routine).

Returning Values

Function

To return values from a function, the RETURN statement is used.

```
RETURN 5;
RETURN myVariable;
-- Note that the following is a relational expression, not an assignment.
RETURN myVariable = 5;
```

Procedure

Since procedures do not directly return, getting values from them is slightly more complicated.

OUT Parameters

Out parameters can be used to return a result from a stored procedure:

```
CREATE PROCEDURE outParam(OUT myParam INT)
CONTAINS SQL
BEGIN
    SET myParam = 5;
END$$
```

Implicit SELECT

In addition to parameters, procedures also have a different method for “returning” values. Any SELECT statement in a procedures that is not captured by a variable or a cursor will have its result set emitted from the procedure. These result sets cannot be captured by a variable, but various database connectors can capture the result sets as they capture the output from a standard SELECT.

Invoking the following procedure:

```
CREATE PROCEDURE selectProc1()
CONTAINS SQL
BEGIN
    SELECT * FROM List LIMIT 2;
END$$
```

May result in one result set being emitted:

```
MariaDB [test]> CALL selectProc1();
+------------+-----------+-------+-----------+
| lastName   | firstName | grade | classroom |
+------------+-----------+-------+-----------+
| LEAPER     | ADRIAN    |     4 |       111 |
| GERSTEIN   | AL        |     5 |       109 |
+------------+-----------+-------+-----------+
2 rows in set (0.00 sec)
```

However, a procedure may have as many un-captured SELECTs as it wants. Therefore, it may emit more than one result set:

```
CREATE PROCEDURE selectProc2()
CONTAINS SQL
BEGIN
    SELECT * FROM List LIMIT 2;
    SELECT * FROM List LIMIT 3;
END$$
```

```
MariaDB [test]> CALL selectProc2();
+------------+-----------+-------+-----------+
| lastName   | firstName | grade | classroom |
+------------+-----------+-------+-----------+
| LEAPER     | ADRIAN    |     4 |       111 |
| GERSTEIN   | AL        |     5 |       109 |
+------------+-----------+-------+-----------+
2 rows in set (0.00 sec)
```

```
+------------+-----------+-------+-----------+
| lastName   | firstName | grade | classroom |
+------------+-----------+-------+-----------+
| LEAPER     | ADRIAN    |     4 |       111 |
| GERSTEIN   | AL        |     5 |       109 |
| YUEN       | ANIKA     |     1 |       103 |
+------------+-----------+-------+-----------+
3 rows in set (0.00 sec)
```

Variables

One for the most useful things about stored routines is the ability to use variables.

Declaring Variables

Variables in stored routines must appear at the top before cursor or handler declarations.

```
DECLARE <variable name> [, <variable name>] ... <variable type> [DEFAULT <value>];
```

Examples:

```
DECLARE myBool BOOLEAN;
DECLARE myStr1, myStr2 VARCHAR(32);
DECLARE myInt INTEGER DEFAULT 0;
```

Setting

Variables can be set in a few different ways.

SET

The simplest way to set a variable is with the SET statement:

```
SET myInt = 5;
SET myStr1 = 'Dogs', myStr2 = 'Cats';
```

SELECT INTO

You can use a SELECT statement **that only returns one row** to set the values of variables by using the INTO clause. The INTO clause comes after the SELECT and before the FROM clauses. Besides assigning the input into variables, the INTO clause does not affect how the query operates.

```
SELECT first, last
INTO myStr1, myStr2
FROM Teachers
WHERE last = 'COVIN';
```

FETCH

When using a cursor, the current values from the cursor can be captured using the FETCH statement. Just like a SELECT statement, a cursor can select multiple attributes.

Assume there is a cursor called `teacherCursor` for the following SELECT statement:

```
SELECT first, last
FROM Teachers;
```

Then the following FETCH statement can get the values from it:

```
FETCH teacherCursor INTO myStr1, myStr2;
```

Cursors

Cursors can be used to iterate through a pre-defined query.

The typical work flow with a cursor is:

1. Declare
2. Open
3. Iterate
4. Close

Declare

Cursors must be declared after variables and before handlers.

The syntax for declaring a cursor is:

```
DECLARE <cursor name> CURSOR FOR
    <query>
```

Cursors may return as many columns and tuples as desired.

Examples

```
DECLARE goodsPriceCursor CURSOR FOR
    SELECT pRICE
    FROM Goods
    WHERE pRICE > 5;

DECLARE teacherCursor CURSOR FOR
    SELECT first, last
    FROM Teachers;

DECLARE purchasesCursor CURSOR FOR
    SELECT C.firstName, C.lastName, R.saleDate, COUNT(*)
    FROM Customers C, Receipts R, Items I
    WHERE R.customer = C.cId
            AND R.rNumber = I.receipt
    GROUP BY R.rNumber;
```

Open

After you declare a cursor but before you use it, you must open the cursor. The OPEN statement is what will actually execute the query.

```
OPEN myCursor;
```

Iterate

A cursor is iterated using the FETCH statement. Each subsequent call to FETCH will return the next tuple.

If FETCH is used with an empty cursor (when no more rows are available), a "No Data" condition occurs. This must be detected with a handler (discussed later).

```
DECLARE teacherCursor CURSOR FOR
   SELECT first, last
   FROM Teachers;

OPEN teacherCursor;

-- First tuple
FETCH teacherCursor INTO myStr1, myStr2;

-- Second tuple
FETCH teacherCursor INTO myStr1, myStr2;

-- Third tuple
FETCH teacherCursor INTO myStr1, myStr2;
```

Close

After you are done with a cursor, it should be closed so its resources can be freed.

```
CLOSE myCursor;
```

Control Flow

Stored routines offer basic control flow constructs.

Conditionals

IF

Stored routines supports the standard IF/ELSEIF/ELSE conditional construct:

```
IF myInt > 0 THEN
   < do something >
ELSEIF myInt = 0 THEN
   < do something else >
ELSE
   < another thing >
END IF;
```

Like most IF constructs, the ELSE is optional and may only occur once. While the ELSEIF can occur zero or more times.

CASE

Stored routines also support case (switch) statements.

```
CASE myInt
    WHEN -1 THEN
        < do something>
    WHEN 0 THEN
        < do something else >
    ELSE
        < another thing >
END CASE;
```

Labels

Labels are used to name a block of code. Labels can be used with a generic block (defined by BEGIN ... END), or with any looping structure (LOOP, REPEAT, or WHILE).

Labels can be associated with general blocks like so (see the LOOP section for how to associate a label with a loop):

```
<label name>: BEGIN
    <loop body>
END <label name>;
```

The label at the end of the block or loop is optional.
Labels allows the program to immediately jump out of blocks using the LEAVE statement:

```
LEAVE <label>;
```

When the LEAVE statement is used in a loop, it is the equivalent of the `break` statement in Java.

Handlers

Handlers are short callbacks that are invoked when one or more conditions occur. They are most often used to observe the end of a cursor.
The syntax for declaring a handler is:

```
DECLARE <handler action> HANDLER FOR <condition> [, <condition> ...] <statement>
```

Handler Actions:

- CONTINUE: Execution of the current program continues.
- EXIT: Execution terminates for the BEGIN block in which the handler is declared. This is true even if the condition occurs in an inner block.
- UNDO: Not supported in MySQL.

Condition Values:

- <MySQL error code>: An actual MySQL error code that is raised for MySQL errors like bad syntax or a constraint violation.
- SQLSTATE [VALUE] <sqlstate value>: Similar to MySQL error codes.
- <condition name>: Used to catch a condition explicitly declared using the DECLARE CONDITION statement.
- SQLWARNING: Shorthand for the class of SQLSTATE values that begin with '01'.

- NOT FOUND: shorthand for the class of SQLSTATE values that begin with '02'. This is the type of condition that rises when a cursor runs out of data.
- SQLEXCEPTION: shorthand for the class of SQLSTATE values that do not begin with '00', '01', or '02'.

Loops

Stored routines come with three different types of loops: LOOP, REPEAT, and WHILE.

LOOP

This LOOP construct is the simplest looping construct. Once entered the loop will continue to execute until it is terminated with either a LEAVE or RETURN.

```
[<label>:] LOOP
    <loop body>
END LOOP [<label>];
```

The LOOP construct works well with cursors and handlers:

```
DECLARE done BOOLEAN DEFAULT FALSE;
DECLARE firstName VARCHAR(32);
DECLARE lastName VARCHAR(32);

DECLARE teacherCursor CURSOR FOR
    SELECT first, last
    FROM Teachers;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN teacherCursor;

teacherLoop: LOOP
    FETCH teacherCursor INTO firstName, lastName;

    IF done = TRUE THEN
        LEAVE teacherLoop;
    END IF;

    SELECT firstName, lastName;
END LOOP;
```

WHILE

The WHILE looping construct in stored routines is similar to the classic while loop. The loop will continue to run until it is either terminated (LEAVE or RETURN) or when the guard becomes FALSE.

```
[<label>:] WHILE <guard expression> DO
    <loop body>
END WHILE [<label>];
```

Since there are no for loops in stored routines, you can easily use the WHILE looping construct instead:

```
CREATE PROCEDURE whileProc(IN num INT)
CONTAINS SQL
BEGIN
    DECLARE count INT DEFAULT 0;

    WHILE count < num DO
        SELECT count;

        SET count = count + 1;
    END WHILE;
END$$
```

REPEAT

The REPEAT looping construct differs from the WHILE lopping structure in two ways:

1. The loop body will always be executed at least once (like a do-while loop).
2. The loop will continue to execute until the guard becomes TRUE.

```
[<label>:] REPEAT
    <loop body>
UNTIL <guard expression>
END REPEAT [<label>];

CREATE PROCEDURE repeatProc(IN num INT)
CONTAINS SQL
BEGIN
    DECLARE count INT DEFAULT 0;

    REPEAT
        SELECT count;

        SET count = count + 1;
    UNTIL count = num
    END REPEAT;
END$$
```

Invoking Routines

Procedures

Procedures need to be explicitly invoked using the CALL statement. You cannot invoke procedures within a SELECT statement.

```
CALL selectProc1();

CALL whileProc(5);

CALL outParam(@val);
-- outputs 5
SELECT @val;
```

Functions

Functions can be invoked within a SELECT statement. They can be invoked the same as other MySQL functions (like NOW() and RAND()).

```
SELECT myFunction();

SELECT *
FROM Teachers
WHERE lastName = someFunction(firstName);
```

Alter Routines

You can alter the characteristics of a routine by using the ALTER PROCEDURE/ROUTINE statements. Only the characteristics can be altered. If you want to edit the function body, you will need to drop it and create it again.

```
ALTER {PROCEDURE | FUNCTION} <routine name> <characteristic> [, <characteristic> ...];
```

Dropping Routines

You can drop a routine just like you can drop a table:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] <routine name>;
```

The IF EXISTS qualifier will only try to execute the statement if the routine exists.

Viewing Stored Routines

You can view all routines using the SHOW STATUS statement:

```
SHOW {PROCEDURE | FUNCTION} STATUS;
```

You can view the statement used to create a procedure using the SHOW CREATE statement:

```
SHOW CREATE {PROCEDURE | FUNCTION} <routine name>;
```

Hint: you can use '\G' in place of a semicolon to change how the output of a query is displayed. This can be useful for long strings such as stored routines.

References

1. http://www.cs.jhu.edu/~nikhil/mysql-storedprocedures.pdf
2. http://net.tutsplus.com/tutorials/an-introduction-to-stored-procedures/
3. http://dev.mysql.com/doc/refman/5.0/en/stored-routines.html
4. http://dev.mysql.com/doc/refman/5.0/en/create-procedure.html
5. http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx