Advanced SQL Stored Procedures

Variables are memory locations used to store data values. Variables have two primary characteristics: 1) **Scope**: where they can be seen by the program; and 2) **Lifespan**: how long they exist in memory.

Declaring variables (outside of stored programs):

You can store a value in a user-defined variable in one statement and then refer to it later in another statement. This enables you to pass values from one statement to another. All variables for a given client connection are automatically freed when that client exits.

User variables are written as @var_name, where the variable name var_name consists of alphanumeric characters. **Programming tip:** begin identifier names with alpha characters, using "camel" case syntax. For example, **myVar**

User variable names are not case sensitive in MySQL 5.0 and up, but are case sensitive before MySQL 5.0. **Programming tip:** regardless of whether a particular environment permits case-sensitive identity names, do ***not*** name different identifiers with the same name!

One way to set a user-defined variable is by issuing a SET statement:

```
SET @var_name = expr [, @var_name = expr] ...
```

For **SET**, **either = or :=** can be used as the assignment operator.

You can also assign a value to a user variable in **statements other than SET**. In this case, the assignment operator **must be := and not =** because the latter is treated as the comparison operator = in non-SET statements:

```
SET @t1=1, @t2=2, @t3:=4;

SELECT @t1, @t2, @t3, @t4 := @t1+@t2+@t3;
```

Examples:

Which presidents were born after President Reagan (*not* SET must use :=):

```
-- retrieve birth date into variable
select @reaganBirth := birth
from president
where last_name='Reagan';

-- use variable's value
select last_name, first_name, birth
from president
where birth > @reaganBirth
order by birth;
```

Display the number of members (*not* SET must use :=):

```
select @count := count(*) from member;
```

Display today's date, and the date a week ago (SET can use both = and :=):

```
set @today = curdate();
set @weekAgo := DATE_SUB(@today, INTERVAL 7 DAY);
select @today, @weekAgo;
```

Declaring variables (within stored programs - declare statement used to declare local variables):

Variables are used in stored procedures to store the immediate result. You can declare a variable by the following syntax:

```
DECLARE variableName datatype(size) DEFAULT default_value;
```

Following DECLARE statement is the variable name. **The variable name should follow the naming convention and should not be the same name of table or column in a database**. Next you can specify the data type of the variable, it can be any primitive type which MySQL supports such as INT, VARCHAR and DATETIME…along with the data type is the value of the variable. When you declare a variable, its initial value is NULL. You can also assign the default value for the variable by using DEFAULT statement. For example, we can define a variable name total_sale with the data type INT and default value is 0 as follows:

```
DECLARE totalSale INT DEFAULT 0;
```

To declare two or more variables with the same data type we can use only just one DECLARE such as:

```
DECLARE x, y INT DEFAULT 0
```

We declared two variables x and y with the same data type INT and their default value is zero.

Assigning variables:

Once you declared a variable, you can start using it. To assign other values to a variable you can use SET statement, for example:

```
DECLARE totalCount INT DEFAULT 0
SET totalCount = 10;
```

The total_count variable's value now is ten (10).

Beside SET statement, we can use SELECT … INTO to assign a query result to a variable.

```
DECLARE totalProducts INT DEFAULT 0
SELECT COUNT(*) INTO totalProducts
FROM products
```

In the example above, we declare a variable **totalProducts** and initialize its value to zero. Then we use SELECT … INTO statement to assign the variable **totalProducts** with the total number of products in the products table.

MySQL session variables vs. procedure variables

Variable scope:

A variable has its own scope (where it can be "seen"). If you declare a variable inside a stored procedure, it will be out of scope when the end of the stored procedure is reached. If you defined a variable inside block BEGIN/END inside a stored procedure it will be out of scope when END is reached. **Programming tip: *always*** use BEGIN/END. Also, you *could* declare two or more variables with the same name in different scopes (though, ***not*** a good practice!); a variable is only effective w/in its scope.

User-defined variables: User variables are written as **@varName**, where the variable name varName consists of alphanumeric characters

http://dev.mysql.com/doc/refman/5.0/en/user-variables.html

MySQL session variables are loosely typed variables that may be initialized somewhere in a session and keep their value until the session ends.

A variable with the '@' at the beginning is a session variable. It exists until the session ends. (That is, while your connection with the database exists, the variable can still be used.)

They are prepended with an @ sign, like this: @var
You can initialize this variable with a SET statement or inside in a query:

```
SET @var := 1

SELECT @var2 := 2
```

When you develop a stored procedure in MySQL, you can pass the input parameters and declare the local variables:

```
DROP PROCEDURE if exists mySp;
DELIMITER //
CREATE PROCEDURE mySp (var INT)
BEGIN
  DECLARE var2 INT;
  SET var2 = 1;

  SELECT  var;
  SELECT  var2;
END//

DELIMITER ;

CALL mySp(10);
```

Note: These variables are not prepended with any prefixes.

The difference between a procedure variable and a session variable is that procedure variables are reinitialized to NULL each time the procedure is called, while the session variable is not:

```
DROP PROCEDURE if exists mySp;
DELIMITER //
CREATE PROCEDURE mySp()
BEGIN
  DECLARE myVar INT DEFAULT 1;
  SET myVar := myVar + 1;
  SET @yourVar := @yourVar + 1;
  SELECT  myVar, @yourVar;
END//

DELIMITER ;

SET @yourVar = 1;

CALL mySp();
myVar  @yourVar
---   ---
2     2
CALL mySp();
myVar  @yourVar
---   ---
2     3
CALL mySp();
myVar  @yourVar
---   ---
2     4
```

Note: myVar (procedure variable) is reinitialized each time the procedure is called, while @yourVar (session variable) is not.

Also take note, that there are global variables available: for example, SELECT @@version; This is why using DELIMITER @@ is *not* a good idea.

Stored Procedure Parameters:

Most stored procedures you develop require parameters. Parameters make the stored procedure more flexible and useful. In MySQL, a parameter has one of three modes **IN**, **OUT** and **INOUT**.

Note:
Specifying a parameter as IN, OUT, or INOUT is valid only for a PROCEDURE. For a FUNCTION, parameters are always regarded as IN parameters.
IN this is the default mode. IN indicates that a parameter can be passed into a stored procedure but any modification inside the stored procedure does not change the parameter value. Example, suppose you pass argument myArg, which holds a value of 10, into stored procedure getAll(id), after executing the stored procedure, the argument myArg still has the value of 10, even though the getAll stored procedure can change the value of the parameter id inside the stored procedure. **Example:**

```
Call getAll(myArg);
```

OUT: Any value the parameter has when passed is ignored by the procedure, and its initial value within the procedure is **NULL**

INOUT: value passed by the caller is the parameter's initial value within the procedure. It can be modified inside the stored procedure, then passed back.

Therefore, an **IN** parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns. An **OUT** parameter passes a value from the procedure back to the caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns. An **INOUT** parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.

For each **OUT** or **INOUT** parameter, you must pass a user-defined variable in the CALL statement that invokes the procedure in order to obtain its value when the procedure returns. If calling the procedure from within another stored procedure or function, you can also pass a routine parameter or local routine variable as an IN or INOUT parameter.

The syntax of defining a parameter in a stored procedure is as follows. MODE could be IN, OUT or INOUT depending upon the purpose of the parameter specified.:

```
Create procedure procedureName(MODE paramName param_type(param_size))
```

paramName is the name of the parameter. The **name must not be the same as the column name of tables** and adheres to naming conventions. Following the parameter name is the type of parameter and its size.

Each parameter is separated by a comma, if the stored procedure has more than one parameter.

Differences between **IN** (no change to caller), **OUT** (does not accept caller value – NULL, and change to caller), and **INOUT** (accepts caller value, and change to caller):

IN:

```
DROP PROCEDURE IF EXISTS spIN;
DELIMITER //
CREATE PROCEDURE spIN(IN p INT)
BEGIN
  select p as 'inside sp';
END //
DELIMITER ;

SET @a=2;
select @a as 'before call';

CALL spIN(@a);
select @a as 'after call';

DROP PROCEDURE IF EXISTS spIN;
```

OUT:

```
DROP PROCEDURE IF EXISTS spOUT;
DELIMITER //
CREATE PROCEDURE spOUT(OUT p INT)
BEGIN
  /* must set par value, otherwise, NULL */
  select p as 'inside sp';
END //
DELIMITER ;

SET @a=2;
select @a as 'before call';

CALL spOUT(@a);
select @a as 'after call';

DROP PROCEDURE IF EXISTS spOUT;
```

INOUT:

```
DROP PROCEDURE IF EXISTS spINOUT;
DELIMITER //
CREATE PROCEDURE spINOUT(INOUT p INT)
BEGIN
  select p as 'inside sp';

  /* changes to par, will return to caller*/
  set p = p*2;
END //
DELIMITER ;

SET @a=2;
select @a as 'before call';

CALL spINOUT(@a);
select @a as 'after call';

DROP PROCEDURE IF EXISTS spINOUT;
```

IN (default) example: stored procedure to get all offices in a country.

```
DROP PROCEDURE IF EXISTS getOfficeByCountry;
DELIMITER //
CREATE PROCEDURE getOfficeByCountry(IN countryName VARCHAR(255))
BEGIN
  SELECT city, phone
  FROM offices
  WHERE country = countryName;
END //
DELIMITER ;
```

As you can see, **countryName** is the **IN** parameter, its data type is varchar, and its size is 255. In the body part of the stored procedure, we retrieve all offices, from where the country name value is passed to the parameter **countryName**. Likewise, if you wanted to retrieve all offices in the USA, pass the following value:

```
CALL getOfficeByCountry('USA')
```

The stored procedure will return all offices in the USA. To get all offices in France, pass France to the stored procedure:

```
CALL getOfficeByCountry('France')
```

The following example shows a simple stored procedure that uses an **OUT** parameter:

```
DROP PROCEDURE IF EXISTS myProc;
DELIMITER //

CREATE PROCEDURE myProc(OUT par1 INT)
BEGIN
  SELECT COUNT(*) INTO par1 FROM customer;
END//

DELIMITER ;

CALL myProc(@a);

SELECT @a;
```

```
+------+
| @a   |
+------+
|    5 |
+------+
```

IN and **OUT** (used together) example: write a store procedure to count the order in a specific order status such as shipped, resolved, cancelled, on hold, disputed or in process.

```
DELIMITER $$
CREATE PROCEDURE countOrderByStatus(IN orderStatus VARCHAR(25),OUT total INT)
BEGIN
  SELECT count(orderNumber)
  INTO total
  FROM orders
  WHERE status = orderStatus;
END$$
DELIMITER ;
```

The countOrderByStatus stored procedure has two parameters:

1. The orderStatus parameter is an **IN** parameter; we pass order status values such as "shipped" or "in process" to the stored procedure to get the number returned.

2. The total parameter is the **OUT** parameter which we use to get the total order by a specified status back.

So to get the number of shipped orders, use the following statements:

```
CALL CountOrderByStatus('shipped',@total);
SELECT @total AS total_shipped;
```

To get the number of in process, do the same:

```
CALL CountOrderByStatus('in process',@total);
SELECT @total AS total_in_process;
```

To get back a value from a procedure using an **OUT** or **INOUT** parameter, pass the parameter by means of a user variable, and then check the value of the variable after the procedure returns. For an **INOUT** parameter, initialize its value before passing it to the procedure. The following procedure has an OUT parameter that the procedure sets to the current server version, and an INOUT parameter that the procedure increments by one from its current value:

```
CREATE PROCEDURE p (OUT versionPar VARCHAR(25), INOUT incrementPar INT)
BEGIN
  /* Set value of OUT parameter */
  SELECT VERSION() INTO versionPar;

  /* Increment value of INOUT parameter */
  SET incrementPar = incrementPar + 1;
END;
```

Before calling the procedure, initialize the variable to be passed to the **INOUT** parameter. After calling the procedure, the values of the two variables will have been set or modified:

```
SET @increment = 10;
CALL p(@version, @increment);
SELECT @version, @increment;
```

MODE Characteristics

IN:

1. Default mode
2. Value is passed into subprogram
3. Actual parameter can be a literal, expression, constant, or initialized variable
4. Formal parameter acts as a constant
5. Can be assigned a default value

OUT:

1. Must be specified
2. Returned to calling environment
3. Uninitialized variable
4. Must be a variable
5. Cannot be assigned a default value

INOUT:

1. Must be specified
2. Passed into subprogram; returned to calling environment
3. Initialized variable
4. Must be a variable
5. Cannot be assigned default value

More Challenging Stored Procedures

Conditional Control in Stored Procedure:

Conditional control enables you to execute the code based on the value of an expression or a combination of an expression using logical operators. MySQL supports two conditional control statements such as **IF** and **CASE**.

IF statement syntax:

```
IF expression THEN commands
[ELSEIF expression THEN commands]
[ELSE commands]
END IF;
```

The commands associated with IF or ELSEIF or ELSE only execute when the expression is evaluated as TRUE. One of the common traps of the IF statement is the **NULL** value. When the expression is evaluated as NULL it is neither TRUE nor FALSE. Here are several combinations of **IF** statements:

```
IF expression THEN commands
END IF;
```

```
IF expression THEN commands
ELSE commands
END IF;
```

```
IF expression THEN commands
ELSEIF expression THEN commands
ELSE commands
END IF;
```

You can have IF statementS nested with other IF statements.

The **CASE** Statement:

When multiple conditions are used with the IF statement the code may not be easy to read. the CASE statement can be used to make the code clearer:

```
CASE
WHEN expression THEN commands
…
WHEN expression THEN commands
ELSE commands
END CASE;
```

Loops in Stored Procedures:

MySQL stored programming language supports loop which allows you to process commands iteratively. The standard loops are discuss as follows

WHILE loop

while loop syntax:

```
WHILE expression DO
  Statements
END WHILE
```

First, the while loop checks the expression, if it evaluates to true it will execute the statement(s) until the expression becomes false. Because the while loop checks the expression before the statements execute, it is known as a pretest loop.

```
DROP PROCEDURE IF EXISTS whileProc;
DELIMITER $$
CREATE PROCEDURE whileProc()
BEGIN
  DECLARE x INT;
  DECLARE str VARCHAR(255);
  SET x = 1;
  SET str = '';
  WHILE x <= 5 DO
    SET str = CONCAT(str,x,',');
    SET x = x + 1;
  END WHILE;
  SELECT str;
END$$
DELIMITER ;

Call whileProc();
DROP PROCEDURE IF EXISTS whileProc;
```

In the stored procedure above, the string is built (by looping) until the variable **x** is greater than 5, then the output of the concatenated string is displayed using the SELECT statement. A common trap for developers is to forget that if variable **x** is not initialized, its default value is NULL. So, the condition in the while loop is always true; the code block inside the while loop is executed indefinitely, or until your server crashes (modern DBMSs may see the indefinite condition, and prevent a crash—that is, breaking out of the loop).

REPEAT loop

repeat loop syntax:

```
REPEAT
  Statements;
UNTIL expression
END REPEAT
```

First, the statements are executed, and then the expression is evaluated. If the expression is evaluated as true, the statements are executed again, until its value becomes false. Because the repeat loop checks the expression after the execution of statements, it is known as a post-test loop.

```
DROP PROCEDURE IF EXISTS repeatProc;
DELIMITER $$
CREATE PROCEDURE repeatProc()
BEGIN
  DECLARE x INT;
  DECLARE str VARCHAR(255);
  SET x = 1;
  SET str = '';
  REPEAT
  SET str = CONCAT(str,x,',');
  SET x = x + 1;
  UNTIL x > 5
  END REPEAT;
  SELECT str;
END$$
DELIMITER ;

Call repeatProc();
DROP PROCEDURE IF EXISTS repeatProc;
```

Note: there is no delimiter (;) after UNTIL expression (only after END REPEAT)

LOOP loop, **LEAVE** and **ITERATE**
Leave statement allows you to leave the loop. (Similar to **break** in other languages.)
Iterate statement allows a start of the loop again. (Similar to **continue** in Java or C#.)

MySQL also supports a **LOOP** loop, which allows statements to execute repeatedly, and with more flexibility:

```
DROP PROCEDURE IF EXISTS loopProc;
DELIMITER $$
CREATE PROCEDURE loopProc()
BEGIN
  DECLARE x INT;
  DECLARE str VARCHAR(255);
  SET x = 1;
  SET str = '';
  loop_label: LOOP
    IF x > 10 THEN
      LEAVE loop_label;
    END IF;

  SET x = x + 1;
    IF (x mod 2) THEN
      ITERATE loop_label;
    ELSE
      SET str = CONCAT(str,x,',');
    END IF;
  END LOOP;
  SELECT str;
END$$

DELIMITER ;

Call loopProc();
DROP PROCEDURE IF EXISTS loopProc;
```

The stored procedure above only constructs a string with even numbers. First, define a loop label, if a variable **x** is greater than 10 the loop ends, because of the leave statement. Otherwise, if the variable **x** is odd, ITERATE ignores everything below it and continues, if the variable x is even, the block after ELSE constructs strings with even numbers.

What's happening here is that **x** is initialized to 1. Once the value of x is greater than 10 it leaves the loop. The set statement updates the loop by one. If x mod 2 is evaluated to zero (0), which means that it is an even number, then the else (false) path is taken—continuing to concatenate the values.

More complex use of **INOUT** mode parameter example: capitalize all words in a string and return the string back to the calling program, will work with or without back ticks (`):

```
DROP PROCEDURE IF EXISTS `capitalize`;
DELIMITER $$
CREATE PROCEDURE `capitalize`(INOUT str VARCHAR(1024))
BEGIN
  DECLARE i INT DEFAULT 1;
  DECLARE myc, pc CHAR(1);
  DECLARE outstr VARCHAR(1000) DEFAULT str;
  WHILE i <= CHAR_LENGTH(str) DO
    SET myc = SUBSTRING(str, i, 1);
    SET pc = CASE WHEN i = 1 THEN ' '
    ELSE SUBSTRING(str, i - 1, 1)
END;
IF pc IN (' ', '&', '''', '_', '?', ';', ':', '!', ',', '-', '/', '(', '.')
THEN
SET outstr = INSERT(outstr, i, 1, UPPER(myc));
END IF;
SET i = i + 1;
END WHILE;
SET str = outstr;
END$$
DELIMITER ;
```

Call **capitalize** stored procedure:

```
SET @str = 'mysql stored procedure tutorial';
CALL capitalize(@str);
SELECT @str;

DROP PROCEDURE IF EXISTS `capitalize`;
```

The @str value is 'Mysql Stored Procedure Tutorial'

References

Stored Programs and Views:
http://dev.mysql.com/doc/refman/5.0/en/stored-programs-views.html

CREATE PROCEDURE and CREATE FUNCTION Syntax:
http://dev.mysql.com/doc/refman/5.0/en/create-procedure.html

Stored Routines:
http://dev.mysql.com/doc/refman/5.0/en/stored-routines.html

Variables:
http://dev.mysql.com/doc/refman/5.0/en/user-variables.html

http://dev.mysql.com/doc/refman/5.0/en/call.html

http://www.codeproject.com/Articles/36484/Working-C-code-for-MySql-Stored-Procedures-IN-OUT

Use:

http://www.mysqltutorial.org/stored-procedures-parameters.aspx

Cursors:

http://dev.mysql.com/doc/refman/5.0/en/cursors.html

http://forge.mysql.com/wiki/Cursors

Why NOT to use cursors:

http://www.oreillynet.com/databases/blog/2006/09/refactoring_mysql_cursors.html

Other:

http://dev.mysql.com/doc/refman/5.0/en/faqs-stored-procs.html

http://dev.mysql.com/doc/refman/5.0/en/declare-local-variable.html

Modes:
http://www.geekinterview.com/question_details/29083