Table of Contents

1. SQL queries versus procedural programming

So far in this course, we have used traditional SQL queries to work with the data in our tables. Traditional SQL is a non-procedural language; you write statements that describe what you want the system to do, and the dbms and the database engine handle the details. This does not mean that SQL is a simplistic language or that it is always easy to write good SQL statements. (You have certainly seen that by now!) What non-procedural means is that you do not need to supply the step by step directions to "open " a table, get the first row, see if it passes the tests you have in the row filter, if so- pull out the columns you want and display them and then go to the next row and repeat until the table is empty and then close the table. In a procedural language you would have to code all of those steps yourself.

The dbms also allows for procedural programming to work with the data in the tables. The developer can create routines that are stored with the other database objects. These routine can be used to display data, or perform update actions. There are two main types of routines- functions and procedures. In this unit we discuss creating and using functions. Procedures are mentioned briefly in an optional document in this unit.

1.1. MySQL programming

In the MySQL system, the programming language does not have a separate name; it is just called MySQL programming. The functions you will create are part of the database system. You have seen aspects of MySQL programming when you used variables. The amount of support the various dbms have for programming techniques varies widely. MySQL did not support programming features into version 5 so its support of programming is somewhat limited.

The first thing to know about this section of the course is that we are going to cover very little of the MySQL programming language. In chapter 31 and 32 of the VanLans book, the author quickly covers a lot of MySQL concepts. For this class, we are going to work with one major concept only- creating MySQL functions. There will be some basic syntax rules we have to cover and some logic structures- but this is a list of topics that are **not** included in our discussion: cursors, error handling, coding loops, output parameters, anything specific to procedures. Procedures are discussed briefly but are not needed for the assignments.

Of course if you want to learn more about MySQL as a programming language you can enroll in CS 155P MySQL Programming.

2. Functions

Since we are going to create functions, it will help to first explain what a function is. A function is a named unit of code that may accept parameters, and that computes and returns a value. You have used many of MySQL's intrinsic functions (Upper, Round, Coalesce). MySQL gives you the ability to create your own functions. The

code for the function is stored in the database along with your other database objects. When you create your own functions, you can use them in your SQL statements anywhere the value returned would be allowed.

3. Example of a Function

Before we start with syntax rules, I'll give you an example of a pretty simple function so that we have some code to talk about. Then we will go though more examples of functions, adding features as we go.

We will start with a function that determines a bonus amount for each of our employees. Everyone gets a bonus of 15% of their salary. For the function to do its job we have to "tell" the function what the salary amount is and the function will "tell" us the amount of the bonus.

Before we create the function we have to change the end of sql delimiter. Currently it is the semicolon. When you enter a query and then type a semicolon, the MySQL client runs the query. The individual statements inside a Create Function query will end with a semicolon and we don't want to execute the create function until we have all of the code written. So we use a different delimiter.

You can change the delimiter by using the delimiter statement at the MySQL prompt including the delimiter to be used. I am using a # as the delimiter.

```
MySQL> delimiter #
```

Note that I now have to use that for the delimiter for a regular sql query also.

```
MySQL> select  emp_id, salary from a_emp.employees limit 3#
+--------+----------+
| emp_id | salary   |
+--------+----------+
|    100 | 24000.00 |
|    101 | 98005.00 |
|    102 | 30300.00 |
+--------+----------+
```

If I forget and just use the semicolon I get another prompt and can use the # now.

```
MySQL> select  emp_id, salary from a_emp.employees limit 3;
    -> #
+--------+----------+
| emp_id | salary   |
+--------+----------+
|    100 | 24000.00 |
|    101 | 98005.00 |
|    102 | 30300.00 |
+--------+----------+
```

You can change the delimiter back to a semicolon if you want; it will return to a semicolon the next time you log into mysql.

The delimiter rule applies to command such as the use command. If you want to switch the database and the delimiter is set to #, the command is

```
Use a_vets#
```

3.1. Managing your function routines

Your function code will be stored on the server. If you need to see the code that was used to create a function you use the "show create function" command. You can end this command with the \G statement terminator to get a slightly better display. Don't worry about the first few lines of the output. You can see the code after Create Function line

Demo 01: Show my function code (run this after you have created a function)

```
mysql> show create function bonus_1\G
*************************** 1. row ***************************
       Function: bonus_1
       sql_mode: STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER
Create Function: CREATE DEFINER=`rose`@`localhost` FUNCTION `bonus_1`(
   in_salary decimal(9,2)
) RETURNS decimal(9,2)
BEGIN

   declare c_bonus_rate decimal(9,2);
   declare v_bonus decimal(9,2);
   set c_bonus_rate := 0.15;

   set v_bonus := in_salary * c_bonus_rate;
   RETURN v_bonus;

END
1 row in set (0.00 sec)
```

When you create and test your functions, you will probably find that you have to correct them. In MySQL you do this by dropping the function and creating it again. You cannot edit the function code. So you want to keep a copy of your function code in a text file someplace while you work on it and you need the statement to drop a function.

Demo 02: Drop function statement

```
Drop function if exists Bonus_1#
```

3.2. Our first function

Demo 03: Create the function in your a_emp database

```
create FUNCTION Bonus_1 (
   in_salary decimal(9,2)
)
   RETURNS decimal(9,2)
BEGIN
/* local variable declaration */
   declare c_bonus_rate decimal(9,2);
   declare v_bonus decimal(9,2);
   set c_bonus_rate := 0.15;

   set v_bonus := in_salary * c_bonus_rate;
   RETURN v_bonus;
END;
#
```

You can copy this into your client and run this and create a function. We can use this function in a query. We pass the function the salary column and the function returns the bonus amount. We created the function in the testbed database but we can use it with tables in other database if we use the two-part table name.

Demo 04: Using the function

```
select emp_id
,      salary
,      Bonus_1(salary)
from   a_emp.employees
limit 4
```

```
#
+--------+----------+-----------------+
| emp_id | salary   | Bonus_1(salary) |
+--------+----------+-----------------+
|    100 | 24000.00 |         3600.00 |
|    101 | 98005.00 |        14700.75 |
|    102 | 30300.00 |         4545.00 |
|    103 |  9000.00 |         1350.00 |
+--------+----------+-----------------+
```

We could also switch to a different database and use the two part name of our function.

```
select emp_id
,      salary
,      a_emp.Bonus_1(salary)
from   a_emp.employees
limit 4
#
```

Demo 05: We could also use this function without a table which will be useful for testing.

```
select Bonus_1(300), Bonus_1(5000)#
+--------------+---------------+
| Bonus_1(300) | Bonus_1(5000) |
+--------------+---------------+
|        45.00 |        750.00 |
+--------------+---------------+
```

3.3. Going through the code

Let's go through the function code one line at a time.

We have seen the Create syntax when we created views; here we are creating a function. We give the function a name and open a parenthesis for the parameters.

```
create FUNCTION Bonus_1 (
```

We have one parameter; I need a name for it – here I am using the name in_salary. The data type decimal (9, 2) is the same as for a table column. Since there is only one parameter, I close the parentheses.

```
   in_salary decimal(9,2)
)
```

The next line says that the value this function returns will be a decimal (9, 2)

```
   RETURNS decimal(9,2)
```

The word Begins signals the start of a block of code.

```
BEGIN
```

This is a comment; you can use the /* */ style comment or the -- style comment.

```
/* local variable declaration */
```

The next two lines provide identifiers for local variables. These statements define variables with a data type. The initial value of a variable is null. (Those nulls always seem to show up- don't they?)

```
   declare c_bonus_rate decimal(9,2);
   declare v_bonus decimal(9,2);
```

Do all of your declarations of variables before you write other statements.

This is an assignment statement that sets an initial value. You can use either: = or = for the assignment operator.

```
set c_bonus_rate := 0.15;
```

This is an assignment statement with a calculation.

```
set v_bonus := in_salary * c_bonus_rate;
```

Since we have calculated the answer we can return it- this is how the function can tell us the answer.

```
RETURN v_bonus;
```

Indicate that we are done with the code block

```
END;
```

And don't forget the # delimiter to run this chunk of code and create our function.

3.4. Errors

Sometimes you make a typing error or get confused about the syntax. The MySQL translator shows you error messages. As with many computer systems, the error messages may not make a lot of sense.

- Hint 1- look for missing semicolons
- Hint 2- look for extra semicolons; semicolons end statements, not lines
- Hint 3- check the spelling of the variables.
- Hint 4- include Set with assignments

If that does not help- post a request for help in the discussion; other people usually see our typing errors before we do.

4. MySQL Basics

MySQL programming statements are an extension to regular MySQL. MySQL included stored routines such as procedures and functions in version 5.0. We will start with some minimal techniques needed to create your own functions that you can use in your SQL queries.

4.1. Syntax rules

Identifiers- user-created names for variables, constants, functions, procedures, etc. The identifier starts with a letter and may include letters, digits and underscores. Identifiers are not case-specific. Do not use reserved words as identifiers. Do not start the identifier with an @ character. That is for session variables, not local variables.

Variable- variables are named memory locations used to store data values. A variable must be declared with a name and a data type before it is used. Variables are initialized to null by default.

Local variables- variables that are declared and used inside the function but that have no meaning outside of the function. Starting the identifier for a local variable with v_ helps to avoid naming conflicts but it is not a requirement of the language. Declare each variable in a separate statement.

Data types- MySQL recognizes the database data types- such as CHAR, VARCHAR, DATE and DECIMAL (9,2). MySQL also has the BOOLEAN data type

Declaring variables

```
Declare v_new_salary  decimal ((9,2)
Declare v_name         varchar(25);
```

Assignment- One way for a variable to get a value is with an assignment statement.

```
Set v_new_salary  :=  25000;
Set v_name        :=  'Pops';
```

Statement- a statement in MySQL ends with a semicolon. A statement is not the same as a physical line on the screen. Some things that might look like a statement are not really a complete statement and therefore do not end with a semicolon. Follow the models I provide for statements.

Comments- You can use the same single line comments and multi-line comments you did with SQL.

4.2. This is a model for a function

Function Template

```
01|  CREATE FUNCTION myFunction (
02|     in_para1 datatype, in_para2 datatype, …)
03|     RETURNS return_datatype
04|  BEGIN
05|  /* local variable declaration */
06|      Declare v_return  return_datatype;
07|      Declare v_varb    datatype;
08|
09|  /* assignments and other statements to do the calculations */
10|      RETURN v_return;
11|  END;
```

Return value- MySQL uses the `RETURN` keyword to specify the value to be returned. The value can be a literal, a variable, or an expression. MySQL allows multiple return statements; structured programming requires one exit point. The data type of the return value is indicated by return data type after the parameter list. (line 03)

Parameters- values that are passed into the function. Consider the Substr function, Substr(prod_name, 1, 10). The actual parameters are prod_name, 1, and 10. In the code that is used to define how the function works, there are formal parameters corresponding to these actual parameters. When you create a function that accepts parameters, you will need to define the formal parameters.

- The actual parameters must match the formal parameters in decimal (9,2) and have compatible data types.
- Since a function should not change its input parameters, all of the formal parameters will be `IN` parameters. (If you want to change the value of parameters, you should create a procedure.)
- The use of `in_` tags in parameter identifiers are a convention; not a syntax rule.
- If the function has no arguments, do not include the empty () in the function definition.

Begin End- The major chunk of code has a `BEGIN` and an `END` keyword. The executable part of the code happens in this block.

5. Examples of Functions

We are going to work with the data from the employees table in the a_emp database and create functions to provide new salary values. For the queries that test these functions I am going to stick with Select statement, but you could use this in update statement to change the data in the table. If you want to do that, add a few new employees to play with and limit the update to those employees. If you do an update and then create or recreate a function you will not be able to roll back those changes.

One thing to note here is that our functions do not mention the database or tables. We can write functions that do that but our functions will be simpler.

5.1. newsalary_1 and newsalary_2

What would the salary be if everyone got a $300 raise? These functions do not actually change the data in the tables; they return values that reflect new salaries based on the values passed to the function.

Demo 06: NewSalary_1 The function returns a value that is 300 more than the input.

```
Drop function if exists newsalary_1 #

create function newsalary_1 (
   in_salary   decimal (9,2))
   returns decimal (10,2)
begin
  declare  c_increase_amount decimal (10,2) default 300;
  declare  v_new_salary      decimal (10,2);

  set v_new_salary  := in_salary + c_increase_amount;
   return v_new_salary ;
end;
#
```

Demo 07: Using the function.

```
select   emp_id, dept_id, hire_date, salary
,        newsalary_1(salary) Sal_1
from     a_emp.employees
limit 4
#
+--------+---------+------------+----------+----------+
| emp_id | dept_id | hire_date  | salary   | Sal_1    |
+--------+---------+------------+----------+----------+
|    100 |      10 | 1989-06-17 | 24000.00 | 24300.00 |
|    101 |      30 | 2008-06-17 | 98005.00 | 98305.00 |
|    102 |     215 | 2010-06-12 | 30300.00 | 30600.00 |
|    103 |     210 | 2010-08-01 |  9000.00 |  9300.00 |
+--------+---------+------------+----------+----------+
```

Demo 08: NewSalary_2 Everyone gets a 5% raise

```
Drop function if exists newsalary_2 #

create function newsalary_2 (
   in_salary   decimal (10,2) )
   returns decimal (11,2)
begin
  declare  c_increase_rate decimal (4,2) default 0.05;
  declare  v_new_salary             decimal (11,2);

   set v_new_salary:= in_salary *(1+ c_increase_rate);
   return v_new_salary ;

end;
#
```

Demo 09: Using the function.

```
select   emp_id, dept_id, hire_date, salary
,        newsalary_2 (salary) Sal_2
from     a_emp.employees
```

```
limit 4
#
+--------+---------+------------+----------+-----------+
| emp_id | dept_id | hire_date  | salary   | Sal_2     |
+--------+---------+------------+----------+-----------+
|    100 |      10 | 1989-06-17 | 24000.00 |  25200.00 |
|    101 |      30 | 2008-06-17 | 98005.00 | 102905.25 |
|    102 |     215 | 2010-06-12 | 30300.00 |  31815.00 |
|    103 |     210 | 2010-08-01 |  9000.00 |   9450.00 |
+--------+---------+------------+----------+-----------+
```