Practical 7

Introduction to stored-procedures

Learning objectives

1. Write a simple stored-procedure without parameters or compound statements
2. Write stored procedures with IN, OUT and INOUT parameters
3. CALL stored procedures and use them appropriately
4. Use user-defined variables in stored procedures
5. Use IF..THEN .. ELSE control structure in stored procedures
6. Use LOOP control structure in stored procedure
7. Use CURSORs in a simple stored procedure

1. Setting-up

- Do this practical is DBS/Prac07 directory. If Prac07 directory is not there, create it.
- **Task 1:** Same .sql files used in **Practical 02** will be used in this task.
    - If you have downloaded them earlier, please use them. If not, download and copy the given three sql files from the practical 02 link to your Prac07 directory.
- **Task 2:** Download the createTable_t1t2t3.sql and inst1t2.sql files from the Prac07 link for this task.
- **Task 3:** Same .sql files used in **Practical 05** will be used in this task.
    - **Create a sub-directory, 'task3' inside Prac07 directory for this task.**
    - If you have downloaded the .sql files earlier, please use them. If not, download and copy the given sql files from the practical 05 link to your Prac07/task3 directory.
- Go to Prac07 directory, open a Terminal and connect to MySQL server using correct username and password.
- Use a suitable command record all the commands and query results to *Prac07Workings.out* file
- Open another Terminal while you are in the same directory and open a text file with name *Prac07Commands* in Vim (> vim *Prac07Commands*) or using any text editor you wish to use. Create your commands in this file before pasting them on MySQL prompt. Use a comment line with the question number before starting each task.
- Comment each command/ query you type.

2. Task 1: Stored procedure examples

This activity use employees (Emp) and department (Dept) tables which have already used in the Practical02. This activity is based on the examples in the Lecture 7.

Following the instructions given in Practical02, create both tables in 'dswork' database and then to insert the given set of values into the created tables. Please refer practical 02 for description of the tables.

1. Use an ALTER TABLE statement and create a column named "newBonus' with the same datatype as bonus in the Emp table.

Hint: Refer Lecture 6, Slide 34 for a way to add a new column to an existing table.

2. Write a procedure named *'resetEmpNewBonus'* to set the new bonus of all employees to zero. Write this procedure in a .sql file with name *reset1.sql*.
   You may look at Lecture 7, slide 11 for sample code. Please type the code instead of copy and paste to avoid errors.
   Execute (source) the *reset1.sql*. file to create the procedure in the MySQL server.

   Run the procedure using the following command.

```
CALL resetEmpNewBonus();
```

   After calling the procedure, use a SELECT statement to check the values of the *newBonus* column.

> NOTE: You can write any stored procedure in .sql files or copy and paste them to the command window. Writing them in .sql file helps to organize them well and reuse them when required.

3. You can see list of stored procedures created in the '*dswork'* database using the following command:

```
SHOW PROCEDURE STATUS WHERE Db='dswork';
```

   Above will show all procedures in 'dswork' database.

4. You can drop any procedure using the following command:

```
DROP PROCEDURE <procedure_name>;
```

   Drop the previously created procedure. Verify that it is deleted by using a suitable command.

5. Create a copy of the *reset1.sql* file as *reset2.sql*. Rename the procedure name in the new file as *resetEmpNewBonus2*(). Revise the procedure to accept a bonus value as an input (IN parameter) and assign it to all employee's new bonus value.

   HINT: Look at lecture 7, Slide 12-16 for use of IN parameter.
   Create the *resetEmpNewBonus2*() procedure and call it with new bonus values of:
   - 200.00
   - 400.00.

   Look at the how the *Emp* table values change after calling the procedure.

6. In a new .sql file named *insert1.sql*, write a stored procedure named *insEmpDetails()* to insert new employee details to *Emp* table. The procedure should accept *empno, fname, midinit ,lastname, workdept* and *salary* as IN parameters. For all new employees, bonus would be zero.
   Run the insert1.sql file to create the stored procedure.
   Call the stored procedure to insert two new employees to the *Emp* table with suitable values. (Use a SELECT statement and look at *Dept* table to see valid department numbers)

   HINT: Look at Lecture 7, Slide 16 for a similar example.

7. Copy the *insert1.sql*, as insert2.sql and revise the insEmpDetails*()* procedure to auto increment the *empno* of new employees by increasing the largest employee number by one. *(empno* should not be a parameter now).

   HINT : Look at Lecture 7, Slide 23 for a similar example.

   Make sure you DROP the previous procedure with the same name and set the DELIMITER properly before and after compound statements.

8. A stored procedure to find the number of employees in a particular department is given in the Lecture 7, slide 25. Write this stored procedure in a .sql file , create the stored procedure and use it to find the number of employees of the departments :
   - A00
   - C01

9. Revise the *countNumEmp* stored procedure as *countNumEmp2 ,* to use a user defined variable to store the count of employees in a particular department, which can be then used outside the stored procedure (without using OUT parameter)

   HINT: Following statement shows how to use a variable to do the required task. '*dep'* is a IN parameter.

```
SET @num= (SELECT COUNT(*) FROM Emp WHERE workdept= dep);
```

   Note that this approach is not a good practise; using OUT parameters is a better way as it is clear and users can access the results easily without knowing the details of the procedure.

10. Write the LOOP and IF examples in Lecture 7, slide 29-30 as a stored procedures and run each with different p1 values such as 5, 10, and 15.

11. In a new sql file named *cursorList.sql*, create the *createProgList* stored procedure given in lecture 7, slide 34. Run it and see if you can get the programmers' name list.

3. Task 2: Stored procedure with two cursors example

You will create the "procedure with two cursors" example in the Lecture 7, slide 35 in this task. Then you will use it to insert values to a table by comparing values in another two tables.

1. Open the *createTable _t1t2t3.sql* file and be familiar with the table structure. Source this file to create three tables *t1,t2* and *t3*. Insert the sample values to tables *t1* and *t2* using the inst1t2.sql file.
   Show all the values of all rows of tables *t1* and *t2.*

2. In a new .sql file named *cursorInsert.sql*, copy and paste the stored procedure given in the lecture 7, Slide 36. Then revise the procedure carefully to:
   - Add suitable DELIMITER statements
   - Change the datatype of variable '*a*', as per the corresponding data type in the given tables.
   - Remove or change the database name when referring to tables.

3. Create the *curdemo()* stored procedure by sourcing the *cursorInsert.sql* file. Then call the stored procedure.
   After calling the stored procedure successfully, use SELECT * statements and look at the content of the tables *t1, t2, t3* .

4. Change the data in tables *t1* and *t2* as you wish, and then again create the *curdemo()* and see how the output is changed.

4. Task 3: More Stored Procedures

Follow the instructions given in Practical05, task 2 to create tables *Emp, Dept, Proj* and *Pworks* in 'dswork' database and then to insert the given set of values into the created tables. Please refer practical 05 for description of the tables. (You will do this activity in Prac07/Task 3 directory. You may need to create a new out file when doing this task)

1. Write a stored procedure to help you to assign an employee to a project. Your procedure should:
   a. accept the three attributes of *Pworks* as input
   b. check that the number of hours (if non-null) is not negative. If it is, display an error message and set the number of hours to NULL.
   c. insert the data to Pworks table.

Test the procedure using some sample data as parameters.

2. Revise the stored procedure written above to check the following condition also:
    a. same as (1) a above
    b. check the two foreign key references to ensure that an appropriate entry exists in that table (e.g. that the *empno* exists in *Emp*). If it doesn't, you should display an error message and exit without attempting to insert the tuple. HINT: You can use SELECT to display a message.
    c. same as (1)b above
    d. same as (1)c above

    Test the procedure using some sample data as parameters.

3. Write a procedure to change the total hours an employee works on projects, taking employee number and the total hours as parameters. After executing the procedure, the total hours that the employee works on all his/her projects together should be equal to the given parameter value.

    For example, suppose an employee with *empno* of 012010 is to be assigned 38 hours per week on all his projects together. Then the parameters to the procedure are '012010' and 38. If the employee currently works 28 hours in total on 5 projects, then he should be assigned an extra 2 hours on each of the projects. On the other hand, if he currently works 43 hours on 5 projects, then his hours on each project should be reduced by 1 hour.

    HINT1: In writing the body of the procedure, first get the total hours and number of projects of the employee and store them in variables. Then update the hours the employee is assigned to each project.

    HINT2: You may assume that the number of hours changed per project is a whole number. You may also assume that the new number of hours specified won't take any project under 0 hours. When testing this procedure, make sure that both of these conditions are met.
    (In a real-world situation, you could not make these assumptions, making the procedure far more complicated.)

    Test the procedure using some sample data as parameters.

NOTE 1: **Errors and Warnings**

- When defining procedures and functions, it's not uncommon to get a message saying that there was an error or warning (or several of both), but no other explanation. MySQL does not display many types of errors and warning directly, but you can still access them.
- You can use the SHOW WARNINGS command to see a list of warnings from the last statement executed.
- Every time you execute a statement, the warnings are reset. This means that if you entered multiple statements (perhaps the last is changing the delimiter back) then you may have lost the warning.
- In general, if SHOW WARNINGS displays an empty table then you will have to enter the statement that caused the actual error.
- This command does give a good amount of information about the problem, but if you need more help you can search for the details of the error message online.
- For errors, the command is SHOW ERRORS, which works in exactly the same way.

5. Submitting your work

Your prac07 directory should have *Prac07Workings,out* and Prac07Commands.sql files together with all given .sql files and set of new .sql files you have created. Zip your Prac07 directory and upload it to Blackboard under 'Assessments/In Class Practical Submissions'

Check whether you have achieved learning outcomes:

I am confident that I can,

Write a simple stored procedure without parameters or compound statements	✓
Call a simple stored procedure (CALL < Procedure_name>)	
Write stored procedure with IN parameters and call it	
Write stored procedure with IN and OUT parameters	
Call a stored procedure with OUT parameter and use the output	
Use compound statements (BEGIN…END) in a stored procedure	
Use IF, THEN ELSE constructs in a stored procedure	
Use LOOP constructs in a stored procedure	
Use Cursors in a simple stored procedure	

Please refer lecture slides, reading materials, and online resources and attempt again, if all the learning outcomes were not achieved. Ask your tutor and get help if you need any clarification.

It's always a good practise to try to finish the practical of a particular week, before attempting the next practical worksheet as your work will be building upon the previous week's tasks.