Database Systems (ISYS1001/ISYS5008)

Lecture 7

Introduction to database programming : Stored Procedures in MySQL

Updated: 22nd September,2021

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

CRICOS Provide Code: 00301J

Copyright Warning

COMMONWEALTH OF AUSTRALIA
Copyright Regulation 1969

WARNING

This material has been copied and communicated to you by or on behalf of **Curtin University of Technology** pursuant to Part VB of the *Copyright Act 1968* (**the Act**)

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Learning outcomes

- Explain why procedures are important.
- Write a simple procedure without any parameters and call it .
- Write and use simple procedures with IN , OUT, INOUT parameters
- Write and use procedures with compound statements
- Use variables in procedures
- Use control structures such as IF.. ELSE and LOOP
- Use cursors in procedures to iterate through rows

Procedures and functions in databases

- SQL consists of many language components; a data definition language, a query language, a data modification language, and more.
- None of these is a procedural programming language such as those you might be used to.
- SQL provides some functionality like other programming languages stored database objects that are defined in terms of SQL code and stored in the server for later execution.
- Examples:
- **Stored procedures, stored functions, triggers, views..**
- Different SQL implementations offer additional functionalities also via their own programming language constructs:
  - Example : T-SQL in MS SQL, PL-SQL in Oracle
- We will look at basics of procedures and functions in MySQL only

Stored procedure and stored functions in MySQL

- Stored procedure:

An object created with CREATE PROCEDURE and invoked using the CALL statement. A procedure does not have a return value but can modify its parameters for later inspection by the caller. It can also generate result sets to be returned to the client program.

- Stored function:

An object created with CREATE FUNCTION and used much like a built-in function. We nvoke it in an expression and it returns a value during expression evaluation.

Definitions from MySQL reference manual

Why stored procedures and functions

- A stored routine (procedure or function) is a set of SQL statements that can be stored in the server.
- Once this has been done, clients don't need to keep reissuing the individual statements but can refer to the stored routine instead.
- Stored routines can be particularly useful in certain situations:
  - When multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.
  - When security is extremely important.
    - Example : Banks, use stored procedures and functions for all common operations. This provides a consistent and secure environment, and routines can ensure that each operation is properly logged. In such a setup, applications and users would have no access to the database tables directly, but can only execute specific stored routines.
    - Stored procedures/ functions can be granted access rights to the database that users who execute the procedure. (not directly to the database)

Why stored procedures and functions

- Advantages and concerns:
  - Stored procedures/ functions can provide improved performance because less information needs to be sent between the server and the client (reduced network traffic)
  - The trade-off is that this does increase the load on the database server because more of the work is done on the server side and less is done on the client (application) side.
  - They enable to have libraries of functions in the database server (embedding business logic as an API in the database)
  - Support delegating access –rights
  - Reducing security threats

Defining stored procedures and functions

```
CREATE PROCEDURE <proc_name> ([<parameter_list>])
   [COMMENT <quoted_string>]
   <statement>;
```

```
CREATE FUNCTION <proc_name> ([<parameter_list>])
     RETURNS <type>
   [COMMENT <quoted_string>]
   <statement>;
```

Defining stored procedures and functions

- `<proc_name>` is the name of the procedure which should be some sensible name.
- `<parameter_list> :` The parameter list is the arguments being passed in, which will generally be global variables.
- `RETURNS` section indicates what sort of value the function will return. This return value is the only thing that differentiates a procedure and a function.
- Functions always return exactly one value, although procedures can output information through their parameters.
- `COMMENT` section allows you to briefly describe your procedure, which is also good practice. While this section is optional, you should always comment your procedure.
- `<statement>` section contains the statements to execute within the procedure or function.

Stored procedures without parameters

- Consider the **Emp** table from Practical 3
- Assume that the bonus has to be set to zero at the beginning of the year, then at any time during the year, it can be incremented as per requirements.
- *If we are going to keep track of old bonuses, the database design has to be changed to have bonus as a separate table,* which we are not going to consider.

COL NAME	TYPE	SIZE	NULL	DESCRIPTION
empno	CHAR	6	no	Employee number, unique
firstname	VARCHAR	12	no	First name
midinit	CHAR	1	no	Middle initial
lastname	VARCHAR	15	no	Last name
workdept	CHAR	3		Employee's dept number
phoneno	CHAR	4		Employee's telephone number
hiredate	DATE			Date hired
job	CHAR	8		Job held by employee
edlevel	INT	2		No. of years of formal educ.
sex	CHAR	1		M=male, F=female
birthdate	DATE			Date of birth
salary	DECIMAL	(8,2)		Annual salary
bonus	**DECIMAL**	**(8,2)**		**Annual bonus**
comm	DECIMAL	(8,2)		Annual commission

Example: Stored procedures without parameters


```
CREATE PROCEDURE resetEmpBonus()
   COMMENT 'Sets bonus field to zero for all employees
   in the Emp table.'
   UPDATE Emp
   SET bonus = 0;
```

- This is a very simple procedure and can be done with a single statement.
- Executing the procedure :

```
CALL resetEmpBonus();
```

- However, if at any time , the interface to the procedure changes, we have to change where the procedure is called only.
- Similarly, if the company decides to give a bonus of at least $200 to everybody, then the procedure can be changed without having to touch the interface code.

Stored procedure with parameters

- In many cases, some user inputs may be required to be used in procedures and/or the a output needs to be used ou.
- A user interface will allow the user to enter information in one or more ways, and this information needs to be passed to the procedure.
- This requires the use of the parameters in procedures, which is a comma-separated list.
- Each item in the list has the form :

  [<mode>] <name> <type> where the *mode* can be:

  - IN = procedure uses value, does not change value.
  - OUT = procedure changes, does not use.
  - IN OUT = both.
  - **IN is the default when no mode is given for a parameter.**
  - the name and the type are equivalent to the attribute declaration in a table.

Stored procedure with parameters

- An OUT parameter has an initial value of NULL.
- Initial value of an OUT parameter can be changed inside the procedure.
- It will contain the value assigned to it in the procedure after the procedure terminates.
- The new value can be passed back to the calling program,
- An IN OUT attribute contains the value passed in, and also passes a value out. Passing values out requires a user-defined variable, as discussed later.
- Note that the mode is optional; if no mode is specified the parameter is an IN parameter.
- Note that functions can’t specify parameter mode; all function parameters are IN only.
- The parameter list enclosed within parentheses must always be present. If there are no parameters, an empty parameter list of() should be used.
- Parameter names are not case-sensitive.

Example: Stored procedure with IN parameter

- Example: Regarding **Emp** table, when a position is about to be offered to a potential employee, but before it is accepted, details like the salary and date hired may still change.
- In addition, other values such as the bonus and commission may be automatically calculated and thus doesn't need to be inserted manually.
- These values has to be managed when inserting values to Emp table.
- Practically, the person doing the inserting shouldn't be expected to type the INSERT command themselves; they will end up with a selection of an option and a window will pop up that allows them to fill in certain data.
- This window may be the representation of an INSERT operation.
- Smart way to manage the INSERT queries would be to write a procedure.

Example: Stored procedure with IN parameter


```
CREATE PROCEDURE insNewEmp(
   e CHAR(6),
   f VARCHAR(12),
   m CHAR(1),
   l VARCHAR(15),
   w CHAR(3)
)
COMMENT 'Insert new employee into table Emp.'
INSERT INTO Emp(empno, firstname, midinit, lastname,
   workdept, bonus)
   VALUES(e, f, m, l, w, 0);
```

- Note that we're not inserting all of the values of an Employee here. Any fields not specified will be set to NULL.
- We have a default bonus of zero. (since a new employee has not yet earned a bonus when being hired, it is assumed to be zero)

- Executing the procedure :

```
CALL insNewEmp('100001', 'Lin', 'M', 'Soon', 'B01');
```

Stored procedures with Multiple statements

- It is possible to include multiple statements in a procedure or function by defining a **block** using the BEGIN and END keywords.
- We may have as many statements as we like between these key-words.
- Each such statement must be terminated with a delimiter semi-colon (;).
- Such blocks of statements are called compound statements.
- However, entering a semi-colon(;) immediately completes a MySQL command and attempt to parse it, including the one to define a procedure. If this happens after the first statement in a procedure that is supposed to have several of them, an error will occur.
- Work around this by using the DELIMITER command to change the delimiter to something like $$ or // before the procedure starts, and then change it back to semi-colon (;) afterwards.

Stored procedures with Multiple Statements


```
DELIMITER $$
CREATE PROCEDURE <proc_name> ([<parameter_list>])
   [COMMENT <quoted_string>]
   BEGIN
    <statement>;
    …
    <statement>;
   END$$
DELIMITER;
```

- The above sequence of commands first sets the delimiter to $$, then defines the procedure. After the definition is completed (with $$) the delimiter is set back to the semi-colon
- Always remember to change the delimiter back to the semi-colon afterwards!

Example: use of DELIMITER

```
DELIMITER //
CREATE PROCEDURE resetEmpBonus()
   COMMENT 'Sets bonus field to zero for all employees in the
   Emp table.'
   BEGIN
        UPDATE Emp
        SET bonus = 0;
   END//
DELIMITER ;
```

- Above procedure can be written without BEGIN.. END block also.
- Here the BEGIN.. END block is used to demonstrate DELIMITER.

User defined variables

- System variables and user-defined variables can be used in stored programs, just as they can be used outside stored-program context.
- In addition DECLARE can be used to define local variables.

  DECLARE <variable name> <type> [DEFAULT <value>]
- This variable is only valid within the compound statement that it is declared.
- To define variables that persist outside of a procedure also, outside of a compound statement, we can use:

  SET @<var_name> = <expression>;
  - This doesn't allow us to specify a type.
  - Variables defined this way stay in memory and can be passed out of a procedures.

User defined variables

- A variable name may consist of any alpha-numeric string as well as period (.), underscore (_) and the dollar sign($); it always starts with an at (@).
- In addition, we can include other characters (such as -) in a variable name by placing quotes around it. So @this-var is not a legal variable name but @this_var and @'this-var' are both legal.
- A user-defined variable persists (stays defined and keeps its value) for the entire session.
- If we start a new session, all previous user-defined variables are lost.
- This means that we can't use such variables to store information and they can used only to communicate. All information in a database should be stored in a table, or computed as needed.

Example: use of variables and multiple statements


```
DROP PROCEDURE insNewEmp;
DELIMITER //
CREATE PROCEDURE insNewEmp(
   f VARCHAR(12),
   m CHAR(1),
   l VARCHAR(15),
   w CHAR(3)
   )
   COMMENT 'Insert new employee into table Emp.'
   BEGIN
        DECLARE nextid CHAR(6);   # Will contain an empno
        SELECT MAX(empno)+1 FROM Emp INTO nextid;
        /* Now do the insertion. */
        INSERT INTO Emp (empno, firstname, midinit, lastname,
                    workdept)
                VALUES (nextid, f, m, l, w);
   END
DELIMITER ;
```

NOTE: There is a better way to automatically generate IDs. Check out this website for a useful example of that.

Example : user defined variables


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

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

```
+------+------+------+------------------+
| @t1  | @t2  | @t3  | @t4 = @t1+@t2+@t3 |
+------+------+------+------------------+
|   1  |   2  |   4  |                7 |
+------+------+------+------------------+
```

Example taken from http://dev.mysql.com/doc/refman/5.6/en/user-variables.html

- Note : Previous releases of MySQL made it possible to assign a value to a user variable in statements other than SET. This functionality is supported in MySQL 8.0 for backward compatibility but is subject to removal in a future release of MySQL.
- When making an assignment in this way, you must use **:=** as the assignment operator; = is treated as the comparison operator in statements other than SET (version 8).

Example: Procedure with OUT parameter

- Assume we want to count the number of employees in a particular department.
- We can pass the department number *in* with a parameter(IN mode), and pass the count back *out* with another parameter (OUT mode).

```
CREATE PROCEDURE countNumEmp(
   IN dep CHAR(3),
   OUT ecount INT
)
   COMMENT 'Count the number of employees in a given department.'
   SELECT COUNT(*) FROM Emp WHERE workdept = dep INTO ecount;
```

- We can call the procedure as follows:

```
CALL countNumEmp('A00',@result);
```

- In this case, the returned result will be stored in the variable @result afterwards, and can be displayed with a SELECT statement.

```
SELECT @result;
```

Flow control statements

- Loops (repetitions) and if ..else statements (decision making) can be used in compound statements.
- IF, CASE, ITERATE, LEAVE, LOOP , WHILE and REPEAT constructs are supported within stored procedures and functions.
- Additionally. RETURN is supported in within stored functions.

IF Statements

- Simplest form:

```
IF <condition> THEN
   <statement(s)>
END IF;
```

- Add ELSE <statement(s)> if required, as:

```
IF <condition> THEN
   <statement(s)>
   ELSE <condition>
   <statement(s)>
END IF;
```

Remember that a SQL statement always ends with a ;

- Add additional cases by ELSIF <statements(s)>:

```
IF <condition> THEN
   <statement(s)>
   ELSEIF <condition> THEN
   <statement(s)>
   ELSEIF <condition> THEN
   <statement(s)>
   ELSE
END IF;
```

Iterative control

```
[<label>] LOOP
    <statement list>
    LEAVE [<label>];
END LOOP [<label>];
```

- Statements within the loop are repeated until the loop is terminated. Usually, this is accomplished with a LEAVE statement.
- We can use the ITERATE keyword (“start the loop again”) to go back to the start of the loop from inside the loop.
- WHILE and REPEAT loops also exist.
- The label is optional but useful for nested loops.

Example : LOOP and IF Statement (version 1)

```
DELIMITER //
CREATE PROCEDURE doIterate(p1 INTEGER)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 = 10 THEN
        LEAVE label1;
    END IF;
  END LOOP label1;
  SET @x = p1;
END//
DELIMITER ;
```

- x needs the @ since it's not a local variable or parameter.
- What if we called the procedure with p1 = 15?

  P1=15 will lead to an infinite loop.

Example : LOOP and IF Statement (version 2)

```
DELIMITER //
CREATE PROCEDURE doIterate2(p1 INTEGER)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN ITERATE label1;
    END IF;
    LEAVE label1;
    END LOOP label1;
  SET @x = p1;
END//
DELIMITER ;
```

LEAVE is similar to *break* in C/Java and ITERATE is similar to continue in C/Java

Stored procedures and Cursor

- Cursor is a method used to traverse over set of rows of tables of a database.
- In SQL procedures, cursors can be used to iterate through results of a SELECT statement, fetch each row individually and perform complex operations on the data.
- It acts as a pointer to the next row of data.
- In order to use a cursor, in general we have to follow four steps;
  - DECLARE the cursor
  - OPEN the cursor to establish the resulting record set.
  - FETCH data into a local variable , row by row
  - CLOSE the cursor

Loops and cursors

- Cursor can be used with loops to iterate through each tuple of a SELECT statement and do required manipulations.
- Declaring a cursor:

```
DECLARE <name> CURSOR FOR <select>;
```

- Above statement declares a cursor and associates it with a SELECT statement that retrieves the rows to be traversed by the cursor.
- Curser must be open to use it for fetching data. Opening a cursor:

```
OPEN <name>;
```

- To fetch the rows, we can use a FETCH statement. The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement. FETCH read the next tuple of the selection from it.

```
FETCH <name> INTO <var_list>;
```

- After finishing the work, cursor has to be closed: :

```
CLOSE <name>;
```

Loops and cursors

- When iterating through a results set, if no more rows are available, a No Data condition occurs .
- To detect this condition, we can set up a handler for it (or for a NOT FOUND condition).
- Cursor handler can be used to test whether a loop is finished.
- We will not go into details on handlers; the format we will use is:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET <name> = TRUE;
```

- This will create a handler with name given in <name>.
- When a cursor looks for the next tuple and doesn't find one, the handler will be set to a value of TRUE.
- If the cursor does find a tuple, the handler is FALSE.
- MySQL Cursors are read-only; they cannot modify a table.

```
CREATE PROCEDURE curdemo()
BEGIN
      DECLARE done INT DEFAULT FALSE;
      DECLARE a CHAR(16);
      DECLARE b, c INT;
      DECLARE cur1 CURSOR FOR SELECT id, data FROM test.t1;
      DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
      DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

      OPEN cur1;
      OPEN cur2;

      read_loop: LOOP
            FETCH cur1 INTO a, b;
            FETCH cur2 INTO c;
            IF done THEN
                  LEAVE read_loop;
            END IF;
            IF b < c THEN
                  INSERT INTO test.t3 VALUES (a,b);
            ELSE
                  INSERT INTO test.t3 VALUES (a,c);
            END IF;
      END LOOP;
CLOSE cur1;
CLOSE cur2;
END;
```

Detailed Procedure with Cursor

Source: https://dev.mysql.com/doc/refman/8.0/en/cursors.html

References

- Stored procedures and functions:

https://dev.mysql.com/doc/refman/8.0/en/stored-programs-defining.html

- Compound statement syntax

https://dev.mysql.com/doc/refman/8.0/en/sql-compound-statements.html

- Flow control Statements:

https://dev.mysql.com/doc/refman/8.0/en/flow-control-statements.html

- Cursors

https://dev.mysql.com/doc/refman/8.0/en/cursors.html

- Good cursor explanation and an example https://www.mysqltutorial.org/mysql-cursor

Summary

- Stored objects such as stored procedures, stored functions, triggers etc. gives programming capability to databases
- Stored procedures are stored in the server and thereby help to reduce the network traffic, implement better security, control user access and provides programming interface.
- Stored procedure can be implemented with or without parameters
- If no parameter type is defined, IN in the default type.
- Parameters can be IN, OUT or INOUT type.
- Compound statements can be used within stored procedures (with BEGIN… END block) and DELIMITER would be changed when using them.
- User defined variables can be used in stored procedures, and to access the output of them.
- Control structures such as LOOP and IF .. THEN.. ELSE can be used in stored procedures.
- Cursors can be used in stored procedures to iterate through results sets.

Happy Database systems

Next week : Triggers

Practical worksheet 8

There will be no assessment in the next week