6 Stored Procedures

Learning Goals:
- Understand what Stored Procedures are and where they are used
- Understand and be able to discuss advantages and disadvantages of using Stored Procedures
- Be able to read and understand the code of Stored Procedures
- Be able to define and call Stored Procedures in MySQL

6.1 Why and How to Use Stored Procedures

Many commercial DBMSs provide functionality to store procedures and functions inside the database. The traditional name for this is "stored procedures". Hence, when people refer to the term "stored procedures" this implies that functions can also be used.
The SQL standard refers to this concept as "persistent stored modules".

They are useful in the following situations:
- If the same routine is used by several applications, it is better to store it centrally and have the application programs refer to it so they don't have to reimplement it.
- Often, the application programs run on client machines while the database is located on a different server machine. In some cases, running a routine on the server is faster because it saves communication time over the network between server and client.
- Routines that are associated with the database server can be used to enhance the modelling power of the system, by providing more ways of deriving data for views and by implementing more complex constraints than SQL can provide.

Disadvantage:
- Stored procedures make application programs less portable.

Stored procedures can be written in SQL or in some other general-purpose programming language.

A Note about Performance

Quotation from Peter Gulutzan's White Paper about Stored Procedures in MySQL:
"There is no compilation, so an SQL stored procedure won't work as quickly as a procedure written with an external language such as C. The main speed gain comes from reduction of network traffic. If you have a repetitive task that requires checking, looping, multiple statements, and no user interaction, do it with a single call to a procedure that's stored on the server. Then there won't be messages going back and forth between server and client, for every step of the task."

You find a complete version of this paper in the Moodle.

Standardization:

SQL/PSM (persistant stored modules) is the part of the newer SQL versions that standardizes the syntax for Stored Procedures.

Different vendors support the standard more or less in a similar way.
Oracle provides the functionality of Stored Procedures written in its procedural extension to SQL, called PL/SQL (= Procedural Language/SQL). A PL/SQL program consists of blocks which can be nested within each other. Each block contains three parts:

- a declarative part,
- an executable part,
- and an exception-building part.

It allows grouping of SQL statements before sending them to Oracle for execution. The PL/SQL blocks can be compiled once and stored in executable form in the database as a stored procedure. They can also be used to implement triggers.

A small PL/SQL tutorial can be found at
http://www-db.stanford.edu/~ullman/fcdb/oracle/or-plsql.html

Looking up the Defined Stored Procedures in MySQL:

It is useful to sometimes look up which Stored Procedures are already defined.
For this purpose, the database called information_schema (the data dictionary) that is generated automatically by MySQL can be used. The table name ROUTINES contains information about Stored Procedures.

Example:
SELECT routine_name, routine_type, definer FROM information_schema.routines
WHERE routine_schema LIKE 'test';
(test is the name of the database in this example)
A possible result is the following:

```
mysql> SELECT routine_name, routine_type, definer FROM
information_schema.routines WHERE routine_schema LIKE 'test'//
+--------------------+--------------+----------------+
| routine_name       | routine_type | definer        |
+--------------------+--------------+----------------+
| case_example       | PROCEDURE    | root@localhost |
| create_random_table | PROCEDURE    | root@localhost |
| create_triggers    | FUNCTION     | root@localhost |
| create_triggers    | PROCEDURE    | root@localhost |
| drop_table         | PROCEDURE    | root@localhost |
| get_random_name    | PROCEDURE    | root@localhost |
| if_example         | PROCEDURE    | root@localhost |
| listTAs            | PROCEDURE    | root@localhost |
| loop_example       | PROCEDURE    | root@localhost |
| raise_error        | PROCEDURE    | root@localhost |
| rec_iteration      | PROCEDURE    | root@localhost |
```

```
| repeat_example      | PROCEDURE    | root@localhost |
| show_table          | PROCEDURE    | root@localhost |
| SortGoodAndBad      | PROCEDURE    | root@localhost |
| testt               | FUNCTION     | root@localhost |
| tt                  | PROCEDURE    | root@localhost |
| while_example       | PROCEDURE    | root@localhost |
| wrapper             | PROCEDURE    | root@localhost |
+---------------------+--------------+----------------+
18 rows in set (0.01 sec)
```

If you want to see the code of the routine, you can for instance write:
SELECT routine_name, routine_type, definer, routine_definition FROM information_schema.routines WHERE routine_schema LIKE 'test';

Alternative:
MySQL provides the statement
SHOW CREATE PROCEDURE procedurename;
or
SHOW CREATE FUNCTION functionname;

Note:
To make output that is very broad more readable, you can end an SQL statement with \G instead of ;
This lists the tuples in vertical instead of horizontal position.

6.2 Creating a Stored Procedure

The general syntax for creating a Stored Procedure is:

CREATE PROCEDURE procedureName (parameters)
local declarations
procedure body;

and for a stored function:

CREATE FUNCTION functionName (parameters)
RETURNS returnType
local declarations
function body;

The difference between procedures and functions is here as in many programming languages: functions return a value, while procedures do not.

For stored routines that are written in a general-purpose programming language, it is customary to specify the name of the language and the filename where the code is stored:

CREATE PROCEDURE procedureName (parameters)
LANGUAGE languageName

EXTERNAL NAME filePathName;

Parameters and local declarations are only used where necessary.

Examples:

1. In a command line interface

```
DROP PROCEDURE IF EXISTS listTAs;
/* This should be executed before redefining a procedure in a new way */

CREATE PROCEDURE listTAs ()
SELECT DISTINCT Student.matNr, sName
FROM Student, TA
WHERE Student.matNr = TA.matNr;
```

Note:
1. When dropping a procedure, it is necessary to name it without the (). However, when defining it, it is necessary to give the () to indicate where the procedure body starts.
2. In a command line interface, usually a semicolon (`;`) is used as a delimiter between two separate SQL statements. Within a stored procedure, it is however possible that several semicolons appear. Therefore, the semicolon cannot be used as the delimiter. You can define a different delimiter, which should be a character or sequence of characters that is not likely to appear otherwise in the statements, for instance `//`. In MySQL you define it simply by saying to the MySQL prompt:
   `delimiter //`
   After completing the definition of the stored procedures, one can redefine the delimiter to a semicolon again by typing
   `delimiter ;`
   to the prompt.

2. Using Java and JDBC

```
String listTAProcedure = "create procedure listTAs ()" +
                  "select distinct Student.matNr, sName " +
                  "from Student, TA " +
                  "where Student.matNr = TA.matNr; // ";

Statement stmt = conn.createStatement();
               //conn is of type Connection
               //It is assumed here that a connection exists
stmt.executeUpdate(listTAProcedure);
```

The procedure listTAs will be compiled and stored in the database as a database object that can be called like a method.

6.3 Calling a Stored Procedure

A stored routine can be called from everywhere where one can use SQL statements (interactive interface, embedded SQL, SQLJ, JDBC, etc.).

The syntax for the call is
CALL routineName (argumentList);

If it is a function, the call is made by using the function name in a place where a value can be used.

Examples:

1. Calling a Stored Procedure in a command line interface

CALL procedureName (parameters);

For instance:
CALL listTAs();
or
CALL listTAs;

Calls may also be used from within an SQL statement:

For instance
SELECT *
FROM MyTable
WHERE someAttribute = functionName (param1, param2);

2. Calling a stored procedure in JDBC

The part that is enclosed in curly braces is the escape syntax for stored procedures. When the driver encounters "{call myProcedureName (parameters)}" it will translate this escape syntax into the native SQL used by the database to call the stored procedure named myProcedureName (parameters) .

CallableStatement is an interface that inherits from PreparedStatement.

The general syntax of calling a stored procedure is as follows:
{?= CALL procedureName(param1,param2,...)}

The Stored procedure call is placed within braces ({}). If the Stored Procedure returns a value, you need to add the question mark and equal (?=) before the call keyword.
If it does not return any values, there is no ?= sign.
If the Stored Procedure accepts any parameters, you list them inside the parentheses after the Stored Procedure's name.

Here are some examples of using the syntax for calling stored procedures in different contexts:

- {CALL procedure_name() }
  Accepts no parameters and returns no value
- {CALL procedure_name(?,?) }
  Accepts two parameters and returns no value
- {?= CALL procedure_name() }
  Accepts no parameter and returns a value
- {?= call procedure_name(?) }
  Accepts one parameter and returns a value

The question mark placeholder (?) can be used for all types of parameters (IN ,OUT, and INOUT).

Example of calling a Stored Procedure from within a Java program:

```
CallableStatement cs = conn.prepareCall(
                       "{CALL listTAs ()}");
ResultSet rs = cs.executeQuery();
```

More details - especially on how to pass parameter values - can be found for instance here:
http://www.mysqltutorial.org/calling-mysql-stored-procedures-from-jdbc/

6.4 Basics of the Syntax of Stored Procedures in MySQL

In the following, some parts of the Stored Procedure syntax are discussed.
Control structures like if, case, while, etc. are not discussed in this section. These are explained by examples in the subsequent section.

Identifiers

In MySQL, the names of Stored Procedures are not case-sensitive. Within the same database, procedure names must be unique (i.e. you cannot overload).
The maximum name length is 64 characters.
Do not use predefined MySQL identifiers as names for stored routines!

Legal and illegal statements inside the procedure body

You can use data manipulation and data definition statements, like:
INSERT, UPDATE, DELETE, SELECT, DROP TABLE, DROP DATABASE, CREATE TABLE, REPLACE, SET, COMMIT, ROLLBACK
These are all legal.

You may use MySQL extensions (i.e. MySQL-specific SQL syntax that may not be understood by other DBMSs) in the procedure, but your code won't be portable if you do so.

Manipulating a routine from inside a routine, however, is not allowed.
The following statements are illegal in a procedure body:

CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE, CREATE FUNCTION, DROP FUNCTION, CREATE TRIGGER, DROP TRIGGER.

Statements like “USE database” are also illegal. MySQL assumes that the default database is the one that the procedure is in.
If you want to refer to a different database inside a statement, you need to put the database name before all table names from that database.

Variables

There are local variables and global variables.

Local variables:
- can only be defined inside a procedure body (i.e. for instance not in the command line)
- are only visibles within the scope of the block where they are defined, but not inside a block that is nested within this block.
- must be declared with a DECLARE clause and may include a default clause
  DECLARE variableName dataType DEFAULT defaultValue;
  Example: DECLARE a INT DEFAULT 0;
  The DEFAULT clause is optional.

Global variables, aka session variables:
- can be defined outside a procedure body.
- are visibles within the scope of a user session
- have names that must start with @, for example @matNr.
- don’t need to be declared with a DECLARE clause, but may be simply used. The type is assigned dynamically to match the assigned value in a reasonable way.
  Example: SET @matNr = 1234;

Parameters

When you create a procedure, you must give a parameter list in ( ) after the procedure name. The parameter list may be empty, but the ( ) must always be there.
The parameters that are listed can be IN, OUT, or INOUT parameters.
- IN parameters will be passed from the caller to the procedure.
- OUT parameters are assigned some value inside the procedure, which can be used by the caller after the call has been completed.
- INOUT parameters will be passed from the caller to the procedure and are assigned some value inside the procedure, which can be used by the caller after the call has been completed.

Cursors

Cursors are used to iterate through a result set of tuples returned by a select query in a procedure body.

Cursors are visible only within the scope of the Stored Procedure in which they are defined. They cannot be passed down to a Stored Procedure that is called within this Stored Procedure. You can define several cursors within the same Stored Procedure.

They must have different names.
Cursors are automatically closed at the end of the block (begin ... end) in which they have been defined if they were not closed explicitly by a close cursor statement. It is good style to close cursors explicitly.

A table on which a cursor is open should not be updated, or the results may be inconsistent in MySQL.

Note:
Cursors can only advance forward, not backwards, so you cannot go back to a previous tuple.
An example of how cursors are used, is described further down in this chapter.

Exception Handling

In the case that an SQL statement is not successful, error conditions ("error codes") are set, for instance "NOT FOUND". Other error codes simply have numbers.
An "exception handler" can be defined to catch this error condition and define some meaningful behaviour how to react to it.

You can check the error code numbers by provoking the error and seeing what message the system gives you.

Example:
`ERROR 1216 (23000): Cannot add or update a child row: a foreign key constraint fails`
In case of an error, MySQL prints an ERROR message composed of three parts:
- an error code (1216 in the above example), which is MySQL specific
- an SQLSTATE value (23000 in the above example) which is normed in ANSI SQL
- and a message string ("Cannot add or update..." in the above example).

You can define an exception handler for this for instance like:
DECLARE EXIT HANDLER FOR 1216, followed by a block of code describing what should be done in the case of this error.

The general syntax for defining exception handlers in MySQL is:
DECLARE
{ EXIT | CONTINUE }
HANDLER FOR
{ error-number | { SQLSTATE error-string } | condition }
SQL statement

If some error code of a database operation is thrown, the DBMS checks whether there is a handler defined for this condition. If so, it executes the code defined for this handler. If there is no handler defined for the exit code, the DBMS interrupts the execution of the procedure with a suitable error message (e.g. "table not found" or similar.)

There are CONTINUE, EXIT and UNDO handlers. UNDO is not yet supported in MySQL.

Behaviour of the procedure according to the handler:
- If the handler is a "continue handler", the execution of the procedure is continued.
- If the handler is an exit handler, the execution of the procedure is stopped.
- An undo handler is planned but at the moment not yet implemented.

It is not possible to have several different handlers for the same error code. That means that if a Stored Procedure contains several different SQL statements that might return "not found", then in every case the same handler code is executed.
However, the necessary treatment of the "not found" event might be different in the different cases. In order to solve this, either the different SQL statements should be treated in different Stored Procedures, or one can define different cases in the code block of the handler.

6.5 Examples of Stored Procedures in MySQL

In this section, a number of examples is explained to show what may be done with Stored Procedures in MySQL.

Example 1: Assign an IN Parameter Value to Variable

```
DELIMITER //

DROP PROCEDURE IF EXISTS text_output//
CREATE PROCEDURE text_output (IN someText VARCHAR(20))
/* This procedure passes a parameter to a local variable and outputs the value of it. */
BEGIN
  DECLARE myText VARCHAR(20);     /*declare a local variable*/
  SET myText = someText;       /* assign the parameter value to the variable myText */
  SELECT myText;               /* output the value of myText as a string to stdout */
END//

DELIMITER ;
CALL text_output ('Hello World!');
```

```
mysql> CALL text_output ('Hello World!');
+-------------+
| myText      |
+-------------+
| Hello World!|
+-------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.06 sec)
```

Note:
If an IN parameter is assigned a new value inside the procedure, this new value cannot be seen from outside of the procedure. If this is needed, an OUT parameter must be used.

Example 2: Visibility of an OUT Parameter

```
DELIMITER //

DROP PROCEDURE IF EXISTS visibility_test//
CREATE PROCEDURE visibility_test (OUT myParam INT)
/* This procedure sets the value of the OUT parameter myParam to -5. This value can
then be used outside the procedure. */
  SET myParam = -5;//

DELIMITER ;
SELECT @y;
CALL visibility_test (@y);
SELECT @y;
```

```
mysql> select @y;
+------+
| @y   |
+------+
| NULL |
+------+
1 row in set (0.00 sec)

mysql> call visibility_test (@y);
Query OK, 0 rows affected (0.00 sec)

mysql> select @y;
+------+
| @y   |
+------+
|   -5 |
+------+
1 row in set (0.00 sec)
```

Example 3: Using a Global Variable

```
DELIMITER //

DROP PROCEDURE IF EXISTS create_global_var//
CREATE PROCEDURE create_global_var( IN myInput INT)
BEGIN
  DECLARE myLocal INT;       /* create local variable myLocal */
  SET @myGlobal = myInput;    /* create global variable @myGlobal */
  SET myLocal = 13;
END//

DELIMITER ;
```

```
CALL create_global_var(100);
SELECT @myGlobal;
SELECT myLocal;
/*This value is unknown because it is only local!*/
```

```
mysql> CALL create_global_var(100);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @myGlobal;
+-----------+
| @myGlobal |
+-----------+
|       100 |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT myLocal;
ERROR 1054 (42S22): Unknown column 'myLocal' in 'field list'
```

Example 4: Using IF

```
DELIMITER //

DROP PROCEDURE IF EXISTS if_test//
CREATE PROCEDURE if_test (IN a INT)
/*An IN parameter is passed whose value is incremented by 1 and assigned to a local
variable b. */
BEGIN
   DECLARE b INT DEFAULT a + 1;
   /* creating a local variable b with default value a + 1 */
   SELECT a, b;
   IF a >= 0 THEN SELECT 'Positive value';
   ELSE SELECT 'Negative value';
   END IF;
END//

DELIMITER ;
CALL if_test (1);
CALL if_test (-1);
```

```
mysql> CALL if_test (1);
+------+------+
| a    | b    |
+------+------+
|    1 |    2 |
+------+------+
1 row in set (0.00 sec)

+----------------+
| Positive value |
```

```
+----------------+
| Positive value |
+----------------+
1 row in set (0.02 sec)

Query OK, 0 rows affected (0.02 sec)

mysql> call if_test (-1);
+------+------+
| a    | b    |
+------+------+
|   -1 |    0 |
+------+------+
1 row in set (0.00 sec)

+----------------+
| Negative value |
+----------------+
| Negative value |
+----------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Example 5: Using CASE

```
DELIMITER //

DROP PROCEDURE IF EXISTS case_test//
CREATE PROCEDURE case_test (IN myInput INT)
BEGIN
   CASE myInput
   WHEN 1 THEN SELECT 1;
   WHEN 2 THEN SELECT 2;
   WHEN 3 THEN SELECT 3;
   ELSE SELECT 'a different value';
   END CASE;
END//

DELIMITER ;

CALL case_test(1);
CALL case_test(2);
CALL case_test(3);
CALL case_test(4);
```

```
mysql> CALL case_test(1);
+---+
| 1 |
```

```
+---+
| 1 |
+---+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALL case_test(2);
+---+
| 2 |
+---+
| 2 |
+---+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALL case_test(3);
+---+
| 3 |
+---+
| 3 |
+---+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALL case_test(4);
+-------------------+
| a different value |
+-------------------+
| a different value |
+-------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)
```

Example 5: Using LOOP

```
DELIMITER //
DROP PROCEDURE IF EXISTS loop_test//
CREATE PROCEDURE loop_test (IN myInput INT)
BEGIN
   DECLARE a INT DEFAULT 0;

   myLoop: LOOP
   /* declare a label name to identify this loop */
      IF a = myInput THEN
         LEAVE myLoop;
```

```
        /* The leave statement means: exit the loop */
      END IF;
      SELECT a;
      SET a = a + 1;
   END LOOP myLoop;    /* specify which loop ends here */
END//

DELIMITER ;
CALL loop_test(3);
```

```
mysql> CALL loop_test(3);
+------+
| a    |
+------+
|    0 |
+------+
1 row in set (0.00 sec)

+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

+------+
| a    |
+------+
|    2 |
+------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Note:
Please also look up the ITERATE clause, described in Gulutzan's paper.

Example 6: Using WHILE

```
DELIMITER //

DROP PROCEDURE IF EXISTS while_test//
CREATE PROCEDURE while_test (IN myInput INT)
BEGIN
   DECLARE n INT DEFAULT 0;
   WHILE n < myInput DO
      SELECT n;
      SET n = n + 1;
   END WHILE;
END//
```

```
DELIMITER ;
CALL while_test(3);
```

```
mysql> CALL while_test(3);
+------+
| n    |
+------+
|    0 |
+------+
1 row in set (0.00 sec)

+------+
| n    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

+------+
| n    |
+------+
|    2 |
+------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

Example 7: Using REPEAT

```
DELIMITER //

DROP PROCEDURE IF EXISTS repeat_test//
CREATE PROCEDURE repeat_test (IN myInput INT)
BEGIN
   DECLARE n INT DEFAULT 0;

   REPEAT
      SELECT n;
      SET n = n+1;
      UNTIL n = myInput
      /* no semicolon between UNTIL condition and  END REPEAT! */
   END REPEAT;

END//

DELIMITER ;
CALL repeat_test(3);
```

The following causes an endless loop:
CALL repeat_test(-1);
(You can stop it by hitting CTRL and C together).

```
mysql> CALL repeat_test(3);
+------+
| n    |
+------+
|    0 |
+------+
1 row in set (0.00 sec)

+------+
| n    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

+------+
| n    |
+------+
|    2 |
+------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Recursion Example

Here starts a larger and more meaningful example, based on the earlier example of the recursive query discussed in the SQL chapter.
As a reminder, the problem was the following:
Given is the table
Employee (empNr, deptNr, mgrNr, salary, bonus) with mgrNr referring to empNr as a foreign key. mgrNr represents the immediate supervisor of the employee empNr.

Query:
Find all employees, to whom employee 7 is supervisor (either direct or higher up in the hierarchy) and whose salary is greater than 100.

First, we create the table Employee and the values used in the example in that chapter:

CREATE TABLE Employee
(empNr INT PRIMARY KEY,
deptNr INT,
mgrNr INT,
salary DECIMAL(8,2),

```
bonus DECIMAL(8,2),
FOREIGN KEY (mgrNr) REFERENCES Employee (empNr)
ON DELETE SET NULL ON UPDATE CASCADE);

INSERT INTO Employee VALUES
(9, NULL, NULL, 180, 100),
(8, 1, 9, 110, 50),
(7, 2, 9, 110, 50),
(6, 2, 7, 120, 20),
(5, 2, 7, 50, 30),
(4, 2, 6, 150, 0),
(3, 2, 5, 90, 0),
(2, 2, 5, 50, 0),
(1, 2, 3, 110, 0);
```

Next, we define the necessary Stored Procedures:

```
DELIMITER //

/***********************************************
rec_iteration
***********************************************/
DROP PROCEDURE IF EXISTS rec_iteration//
CREATE PROCEDURE rec_iteration(IN id INT, IN tablename VARCHAR(128) )
BEGIN

   DECLARE done INT DEFAULT 0;

   DECLARE empId INT;
/* empId is used to store an employee id coming back from the db in the procedure*/

   DECLARE cur1 CURSOR FOR SELECT empNr FROM Employee WHERE mgrNr
= id;
/* The SQL statement for which this cursor is defined delivers a list of the supervisees
of the manager with mgrNr = id.*/

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
   /* handler for not found condition, when there are no more rows left to fetch*/

   OPEN cur1;
/*execute the select statement for which the cursor has been defined*/

   REPEAT
      FETCH cur1 INTO empId;
     /* now empId contains the empNr of an employee whose manager has the empNr
id */
      IF NOT done THEN
         SELECT CONCAT("INSERT INTO ", tablename, " VALUES (?)")
                  INTO @stmt_body;
```

```
/* In the global variable @stmt_body, we collect the form of an INSERT statement into
a string variable. The statement will later be executed. It cannot be executed directly
because it contains the table name as a string variable and not as a constant. If the
tablename was given as a constant, this workaround would not be necessary. The
questionmark (?) will later be filled with the value of the respective empId*/
        PREPARE stmt FROM @stmt_body;
/* From the (string) contents of the variable @stmt_body, an SQL statement that can
be executed, is prepared by the PREPARE ... FROM clause. This clause accepts a
variable defined by the programmer (in this case stmt) that does not need to be
declared beforehand. stmt is now called a "prepared statement" because it is stored in
the variable and ready to be executed, but it has not been executed yet. In order to do
this, an EXECUTE statement is necessary.*/
        SET @globalEmpId = empId;
/* This simply assigns the contents of the local variable to the global variable @glo-
balEmpId. It is only necessary because of the execute statement which does not work
with local variables. */
        EXECUTE stmt USING @globalEmpId;
/* The SQL statement that is contained in the variable stmt is now executed. The con-
tents of @globalEmpId are assigned to the questionmark (?) in the prepared state-
ment. In our example here that means the mgrNr is passed to the prepared statement
so that this statement can insert the value into the result table.*/
        DEALLOCATE PREPARE stmt;
/* It is good style to deallocate a prepared statement after it is not needed any longer.
The memory space and the pointer to it is freed then. It will be freed anyway after the
connection to the database is ended.*/
        CALL rec_iteration(empId, tablename);
/* This is the place where the recursion happens! For the current supervisee whose
empNr is stored in empId, the procedure that finds this person's supervisees is exe-
cuted again. The results are collected within the same table.*/
      END IF;
   UNTIL done END REPEAT;

   CLOSE cur1;
/* It is good style to close a cursor after it is not needed any longer. */
END//

/***********************************************
get_random_name
***********************************************/
/*returns a random result table name based on the current unix_timestamp concate-
nated with a number between 0 and 100. Unix_Timestamp() is a MySQL built-in func-
tion that delivers a time stamp (also under Windows :-) ).
The reason why a random number is used to create the table name is to make the
table names unique even if several users run the stored procedure at the same time. It
may be possible that two calls running in parallel by chance get the same table name
for their results but it is very unlikely.*/

DROP PROCEDURE IF EXISTS get_random_name//
CREATE PROCEDURE get_random_name (OUT output VARCHAR(128))
```

```
/* This is written as a procedure that delivers just one out parameter. Actually, it would
be better style to write this as a function.*/
BEGIN
   SET output = CONCAT("RESULT_TABLE_", UNIX_TIMESTAMP(), '_',
                                    FLOOR(RAND()*100));
/* FLOOR(): a function that cuts off the digits after the decimal point. RAND() is a func-
tion that returns a random number between 0 and 1.*/
END//

/**********************************************
create_random_table
**********************************************/
DROP PROCEDURE IF EXISTS create_random_table//
CREATE PROCEDURE create_random_table (OUT random_table_name VAR-
CHAR(128))
BEGIN
   DECLARE done INT DEFAULT 1;
   /*1050 mysql error is thrown when a table already existts*/
   DECLARE CONTINUE HANDLER FOR 1050 SET done=0;

   REPEAT
/* we repeat the creation of a table with the name random_table_name until a table
with this name does not exist yet. It might exist if the same procedure is being called in
parallel by another process that has received by chance the same random number
from the procedure get_random_name*/
   CALL get_random_name(random_table_name);
   SET done = 1;

  SET @stmt_body = CONCAT("CREATE TABLE ", random_table_name, " (empNr
INT)" );
 /* Alternatively:  SELECT CONCAT("CREATE TABLE ", random_table_name, "
(empNr INT)" ) INTO @stmt_body; */
   /* You can not use local variables for prepared stmts =>> syntax error */
   PREPARE stmt FROM @stmt_body;
   EXECUTE stmt;
   DEALLOCATE PREPARE stmt;
   UNTIL done=1
   END REPEAT;
END//

/**********************************************
drop_table
**********************************************/
DROP PROCEDURE IF EXISTS drop_table//
CREATE PROCEDURE drop_table (IN table_name VARCHAR(128))
/* This procedure accepts a table_name in a parameter and prepares an SQL state-
ment that drops this table. If the table_name were known already, one could directly
state drop table_name, but because we don't know it at programming time, it must be
done with a prepared statement. */
BEGIN
```

```
    SELECT CONCAT("DROP TABLE ", table_name) INTO @stmt_body;
    /* alternatively: SET @stmt_body = CONCAT("DROP TABLE ", table_name); */
    PREPARE stmt from @stmt_body;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
END//

/***********************************************
show_table
***********************************************/

DROP PROCEDURE IF EXISTS show_table//
CREATE PROCEDURE show_table (IN table_name VARCHAR(128))
/* This procedure accepts a table_name in a parameter and prepares an SQL state-
ment that lists the contents of this table. If the table_name were known already, one
could directly state select * from table_name, but because we don't know it at pro-
gramming time, it must be done with a prepared statement. */

BEGIN
   SELECT CONCAT("SELECT * from ", table_name) INTO @stmt_body;
   PREPARE stmt from @stmt_body;
   EXECUTE stmt;
   DEALLOCATE PREPARE stmt;
END//

/***********************************************
wrapper
***********************************************/
DROP PROCEDURE IF EXISTS wrapper//
CREATE PROCEDURE wrapper (IN mgrNr INT)
/* A "wrapper" is a routine that is "wrapped around" some other code to achieve some
kind of transformation, in our case just to combine the different steps that we want to
perform. */
BEGIN
   DECLARE random_table_name VARCHAR(128);

   CALL create_random_table(random_table_name);
    /* Create a table with a random name for the result */

   CALL rec_iteration( mgrNr, random_table_name);
   /* Execute the recursive search for supervisees*/

   CALL show_table(random_table_name);
   /*output the results*/

   CALL drop_table(random_table_name);
   /* drop the result table because it is needed no longer */

END//
```

```
DELIMITER ;

/***********************************************
execute the procedure
***********************************************/
SET MAX_SP_RECURSION_DEPTH = 10;
/* setting maximum recursion depth to 10 */

SELECT 'Manager 7 supervises the following Employees';
/* This is just a comment written to stdout to indicate what the result means */

CALL wrapper(7);
/*call the procedure for employee with empNr 7 */
```

And here is the same code once more, only without the comments, for better readability:

```
DELIMITER //

/***********************************************
rec_iteration
***********************************************/
DROP PROCEDURE IF EXISTS rec_iteration//
CREATE PROCEDURE rec_iteration(IN id INT, IN tablename VARCHAR(128) )
BEGIN

   DECLARE done INT DEFAULT 0;
   DECLARE empId INT;
   DECLARE cur1 CURSOR FOR SELECT empNr from Employee where mgrNr  = id;
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

   OPEN cur1;
   REPEAT
      FETCH cur1 INTO empId;
      IF NOT done THEN
         SELECT CONCAT("INSERT INTO ", tablename, " VALUES (?)")
                  INTO @stmt_body;
         PREPARE stmt FROM @stmt_body;
         SET @globalEmpId = empId;
         EXECUTE stmt USING @globalEmpId;
         DEALLOCATE PREPARE stmt;
         call rec_iteration(empId, tablename);
      end IF;
   UNTIL done END REPEAT;

   CLOSE cur1;
END//
```

```
/***********************************************
get_random_name
***********************************************/

DROP PROCEDURE IF EXISTS get_random_name//
CREATE PROCEDURE get_random_name (OUT output VARCHAR(128))

BEGIN
   SET output = CONCAT("RESULT_TABLE_", UNIX_TIMESTAMP(), '_',
                                  FLOOR(RAND()*100));
END//

/***********************************************
create_random_table
***********************************************/
DROP PROCEDURE IF EXISTS create_random_table//
CREATE PROCEDURE create_random_table (OUT random_table_name VAR-
CHAR(128))
BEGIN
   DECLARE done INT DEFAULT 1;
   DECLARE CONTINUE HANDLER FOR 1050 SET done=0;

   REPEAT

   CALL get_random_name(random_table_name);
   SET done = 1;

  SET @stmt_body = CONCAT("CREATE TABLE ", random_table_name, " (empNr
INT)" );
   PREPARE stmt FROM @stmt_body;
   EXECUTE stmt;
   DEALLOCATE PREPARE stmt;
   UNTIL done=1
   END REPEAT;
END//

/***********************************************
drop_table
***********************************************/
DROP PROCEDURE IF EXISTS drop_table//
CREATE PROCEDURE drop_table (IN table_name VARCHAR(128))

BEGIN
      SELECT CONCAT("DROP TABLE ", table_name) INTO @stmt_body;
      /* alternatively: SET @stmt_body = CONCAT("DROP TABLE ", table_name); */
      PREPARE stmt from @stmt_body;
      EXECUTE stmt;
      DEALLOCATE PREPARE stmt;
END//
```

```
/***********************************************
show_table
***********************************************/

DROP PROCEDURE IF EXISTS show_table//
CREATE PROCEDURE show_table (IN table_name VARCHAR(128))

BEGIN
   SELECT CONCAT("SELECT * from ", table_name) INTO @stmt_body;
   PREPARE stmt from @stmt_body;
   EXECUTE stmt;
   DEALLOCATE PREPARE stmt;
END//

/***********************************************
wrapper
***********************************************/
DROP PROCEDURE IF EXISTS wrapper//
CREATE PROCEDURE wrapper (IN mgrNr INT)

BEGIN
   DECLARE random_table_name VARCHAR(128);

   CALL create_random_table(random_table_name);
   CALL rec_iteration( mgrNr, random_table_name);
   CALL show_table(random_table_name);
   CALL drop_table(random_table_name);

END//

DELIMITER ;

SET MAX_SP_RECURSION_DEPTH = 10;

SELECT 'Manager 7 supervises the following Employees: ';
CALL wrapper(7);
```

6.6 Evaluation Summary

Advantages of Stored Procedures:

- In a client/server architecture, more of the processing takes place on the server, thus reducing network traffic. Also, the client becomes thinner and the server fatter.
- When applications access the stored procedures instead of the data directly, this increases data security.
- Data integrity is improved because all application use the same functionality.

Disadvantages of Stored Procedures:

- Writing stored procedures takes more time than just using a development framework for the application.

- Stored procedures are written specifically for a given DBMS. This makes it difficult to replace the DBMS (the procedures may have to be reimplemented).
- An application using stored procedures is less portable to another platform with another DBMS.
- In a client/server architecture, performance of the server goes down with increasing number of users who may all use the same stored procedures.