SUCCEED WE MUST

DATABASE PROGRAMING

Stored Procedures

rmwavu@must.ac.ug
FACULTY OF COMPUTING AND INFORMATICS
DEPARTMENT OF INFORMATION TECHNOLOGY

SUCCEED WE MUST

What is stored procedure

- What is a stored procedure ? A **stored procedure** (also termed **proc**, **storp**, **sproc**, **StoPro**, **StoredProc**, **StoreProc**, **sp**, or **SP**) is a subroutine available to applications that access a relation database management system.

Stored Procedures: Intro

- **Stored Routines** (Procedures and Functions) are supported in version MySQL 5.0 and latter
- The general syntax of Creating a Stored Procedure is :
  CREATE PROCEDURE proc_name ([proc_parameter[.....]])
  routine_body
- **proc_name :** procedure name
  proc_parameter : [IN | OUT | INOUT] param_name type
  routine_body : Valid SQL procedure statement
- The parameter list is available with in the parentheses. Parameter can be declared to use any valid data type By default each parameter is an IN parameter.
- For specifying other type of parameter used, the OUT or INOUT keyword before the parameter name. An **IN** parameter is used to pass the value into a procedure.

Stored Procedures: Intro

- The stored procedure is created on an active database
- An **OUT** parameter is used to pass the value from the procedure to the **caller** but its visible to the caller.
- An **INOUT** parameter is initialized by the caller and it can be **modified by the procedure**, and any change made by the procedure is visible to the caller.
- The **routine_body** contains the valid SQL procedure statement that can be a simple statement like SELECT or INSERT or they can be a compound statement written using BEGIN and END.
- Compound statement can **consist of declarations, loops or other control structure.**

SUCCEED WE MUST

Creating a stored Procedure

- The example bellow is created for future use, it will not return any values unless it is called

- **Example**: Returning the average product price;
  - delimiter /// - Change the termination from ; **to** /// - do not use key characters like /
  - The termination can automatically be reset to ; if you logout and login again in MySQL but if you lose the connection or if your database crushes, the delimiter will remain the same
  - Delimiter ; -- Resets back termination from /// to ;

```
mysql> delimiter ///
mysql> CREATE PROCEDURE average( )
    -> BEGIN
    -> SELECT Avg(SalesAmmount) AS priceaverage FROM sales;
    -> END ///
Query OK, 0 rows affected (0.00 sec)

mysql> call average()///
+--------------+
| priceaverage |
+--------------+
|  302666.6667 |
+--------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> --this is a statement
```

Stored Procedure: Simple example selecting from more than one table

```
mysql> DELIMITER ///
mysql>
mysql> CREATE PROCEDURE average2( )
    -> BEGIN
    -> SELECT Avg(SalesAmmount) AS priceaverage FROM sales;
    -> SELECT FirstName FROM customers;
    -> END ///
Query OK, 0 rows affected (0.00 sec)

mysql> call average2()
    -> ///
+--------------+
| priceaverage |
+--------------+
|  302666.6667 |
+--------------+
1 row in set (0.00 sec)

+-----------+
| FirstName |
+-----------+
| Kimera    |
| Tugume    |
| Ruhanga   |
| Nyangoma  |
+-----------+
4 rows in set (0.01 sec)

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

Parameters in Stored Procedures

- Typically stored procedures do not display results; rather, they return them into variables that you specify.
- Example with parameters;
- P1, ph and pa are parameters of datatype decimal created to store variables

```
mysql> describe sales//
+---------------+---------+------+-----+---------+-------+
| Field         | Type    | Null | Key | Default | Extra |
+---------------+---------+------+-----+---------+-------+
| SalesId       | int(11) | NO   | PRI | NULL    |       |
| Date          | date    | YES  |     | NULL    |       |
| SalesAmmount  | int(11) | YES  |     | NULL    |       |
| CustomerId    | int(11) | YES  | MUL | NULL    |       |
+---------------+---------+------+-----+---------+-------+
4 rows in set (0.01 sec)

mysql> CREATE PROCEDURE average(
    -> OUT pl DECIMAL(8,2),
    -> OUT ph DECIMAL(8,2),
    -> OUT pa DECIMAL(8,2)
    -> )
    -> BEGIN
    -> SELECT Min(SalesAmmount) INTO pl FROM sales;
    -> SELECT Max(SalesAmmount) INTO ph FROM sales;
    -> SELECT Avg(SalesAmmount) INTO pa FROM sales;
    -> 
    -> END //
Query OK, 0 rows affected (0.00 sec)
```

Example explained

- OUT is used to specify that this parameter is used to send a value out of the stored procedure (back to the caller).
- MySQL supports parameters of types
  - IN (those passed to stored procedures),
  - OUT (those passed from stored procedures, as we've used here), and
  - INOUT (those used to pass parameters to and from stored procedures).
- The stored procedure code itself is enclosed within BEGIN and END
- A series of SELECT statements are performed to retrieve the values that are then saved into the appropriate variables (by specifying the INTO keyword).

Executing Stored Procedures

- **MySQL refers to stored procedure execution as calling**, and so the MySQL statement to execute a stored procedure is simply CALL.
- **CALL takes the name of the stored procedure and any parameters that need to be passed to it.**
- From the example above, three variable names can be specified and exactly three parameters will be passed and three updated values will be stored in these variables:

```
mysql> call average (@pricelow, @pricehigh, @priceaverage)//
Query OK, 0 rows affected, 1 warning (0.02 sec)
```

- All MySQL variable names begin with @
- When called, this statement does not actually display any data. Rather, it returns variables that can then be displayed (or used in other processing).

Executing Stored Procedures

- To display the product average, you would write;

```
mysql> select @priceaverage//
+-----------------+
| @priceaverage   |
+-----------------+
|       302666.67 |
+-----------------+
1 row in set (0.00 sec)
```

- To display the three values, you would write;
  SELECT @pricehigh, @pricelow, @priceaverage;

Example 2:

- using both IN and OUT parameters. **ordertotal** accepts an order number and returns the total for that order:
- CREATE PROCEDURE ordertotal(

```
 IN onumber INT,
OUT ototal DECIMAL(8,2) )
BEGIN
SELECT Sum(item_price*quantity) FROM orderitems WHERE order_num =
onumber INTO ototal;
END //
delimiter ;
```

- **onumber** is defined as IN because the order number is passed in to the stored procedure. **ototal** is defined as OUT because the total is to be returned from the stored procedure.
- The SELECT statement used both of these parameters, the WHERE clause uses onumber to select the right rows, and INTO uses ototal to store the calculated total.

Example 2: Calling and execution

- CALL ordertotal(20005, @total);
  To invoke the stored procedure, you pass two parameters and to display the total, you use;
- SELECT @total;

Dropping Stored Procedure

- After they are created, stored procedures remain on the server, ready for use, until dropped.
- The drop command removes the stored procedure from the server.
  - DROP PROCEDURE productpricing;

Example 3: Real Importance of SP's

- The previous examples actually make things a little more complex but the real use of sp's is when business rules and intelligent processing are included within them.
- **Example 3:**
  You need to obtain order totals as before, but also need to add sales tax to the total, but only for some customers (perhaps the ones in your own district).
- Now you need to do several things;
  - Obtain the total (as before).
  - Conditionally add tax to the total.
  - Return the total (with or without tax).
- This is the perfect job for SP's

SUCCEED WE MUST

Example 3

```
Delimiter //
-- Name: ordertotal
-- Parameters: onumber = order number
-- taxable = 0 if not taxable, 1 if taxable
-- ototal = order total variable
CREATE PROCEDURE ordertotal(
IN onumber INT,
IN taxable BOOLEAN,
OUT ototal DECIMAL(8,2)
) COMMENT 'Obtain order total, optionally adding tax‘

 BEGIN
 -- Declare variable for total
DECLARE total DECIMAL(8,2);
 -- Declare tax percentage
DECLARE taxrate INT DEFAULT 6;
 -- Get the order total
SELECT Sum(item_price*quantity) FROM orderitems WHERE order_num = onumber INTO total;
-- Is this taxable?
IF taxable THEN
 -- Yes, so add taxrate to the total
SELECT total+(total/100*taxrate) INTO total;
END IF;

-- And finally, save to out variable
SELECT total INTO ototal;
END//
delimiter ;
```

Example explained

- Comments have been added throughout (preceded by --). This is extremely important as stored procedures increase in complexity.
- A parameter taxable of type BOOLEAN (specify true if taxable, false if not).
- Within the stored procedure body, two local variables are defined using DECLARE statements.
- DECLARE requires that a variable name and datatype be specified.
- The SELECT has changed so the result is stored in total(the local variable) instead of ototal.
- Then an IF statement checks to see if taxable is true, and if it is, another SELECT statement is used to add the tax to local variable total. And finally, total (which might or might not have had tax added) is saved to ototal using another SELECT statement..

Explanation cont'd

- The COMMENT Keyword
- The stored procedure for this example included a COMMENT value in the CREATE PROCEDURE statement.
- This is not required, but if specified, is displayed in SHOW PROCEDURE STATUS results

Executing and more remarks on SP's

- Call the Procedure and execute it;
  - CALL ordertotal(20005, 0, @total);
  - SELECT @total;
- Boolean values can be specified as **1** for true and **0** for false
- To display the CREATE statement, use
  SHOW CREATE PROCEDURE ordertotal;
- To obtain a list of stored procedures including details on when and who created them, use
  SHOW PROCEDURE STATUS.

Example 4: Using INOUT parameter

The stored procedure capitalizes all words in a string and returns it back to the calling program as bellow

```
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
            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$$
```

Execution

- SET @str = 'mysql stored procedure tutorial';
- CALL Capitalize (@str);
- SELECT @str;

Example 5: Using stored procedures to input data

SUCCEED WE MUST

- You can create a stored procedure to help you input information in a table by just passing values to the procedure and even maniplating your data the way you want

Tow tables used

```
mysql> describe sales;
+--------------+---------+------+-----+---------+-------+
| Field        | Type    | Null | Key | Default | Extra |
+--------------+---------+------+-----+---------+-------+
| SalesId      | int(11) | NO   | PRI | NULL    |       |
| Date         | date    | YES  |     | NULL    |       |
| SalesAmmount | int(11) | YES  |     | NULL    |       |
| CustomerId   | int(11) | YES  | MUL | NULL    |       |
+--------------+---------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

```
mysql> select *from customers!
+------------+-----------+----------+-------------------+------------+------------+
| CostomerId | FirstName | LastName | Email             | DOB        | Mobile     |
+------------+-----------+----------+-------------------+------------+------------+
|          1 | Kimera    | Richard  | rk@gmai.com       | 1999-09-08 | 2147483647 |
|          2 | Kalungi   | Dickson  | lk@gmail.com      | 1956-09-06 |      64646 |
|          3 | Mwavu     | Rogers   | baller@must.ac.ug | 1990-09-02 | 2147483647 |
|          4 | Carlorine | Mukunde  | km@yahoo.com      | 1986-09-09 | 2147483647 |
+------------+-----------+----------+-------------------+------------+------------+
4 rows in set (0.00 sec)
```

SUCCEED WE MUST

Example 5: Using stored procedures to input data

- Creating the procedure

```
mysql> create procedure totalPurchase(
    -> IN salesId int(4), Purchasedate date, amm int (8), id int, custNo int,
    -> OUT total int (10),
    -> OUT CustName varchar(15)
    -> )
    -> BEGIN
    -> insert into sales values (salesId,Purchasedate,amm,id);
    -> select LastName from customers where CostomerId=custNo INTO CustName;
    -> select sum(salesAmmount) from sales where customerId=custNo Into total;
    -> End!
Query OK, 0 rows affected (0.00 sec)
```

- Passing variables to the procedure and selecting the results

```
mysql> call totalpurchase(27,2013-08-08,50000,1,1,@totalamm,@cname)!
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> select @totalamm,@cname!
+-----------+---------+
| @totalamm | @cname  |
+-----------+---------+
|  80200000 | Richard |
+-----------+---------+
1 row in set (0.00 sec)
```

Commit & Rollback

By default most RDBMS systems will be configured to automatically COMMIT an SQL statement, i.e save it in the database.

It is possible however to use Transactions to enact a number of SQL statements and either COMMIT the data to the database or ROLLBACK the data if required

Why do you think we may want to do this?

Commit & Rollback

The Syntax of a Transaction is as follows

START TRANSACTION

[SQL statement(s)]

COMMIT / ROLLBACK

Note: The decision to commit or rollback is typically based up a user decision or if an error has arisen, therefore your SQL should be dynamic i.e. provided by your programming language based on for example user interaction.

SUCCEED WE MUST

Commit & Rollback

For example the following statement deletes all data from tablle called one, alters it to delete column for sex and adds column for Age, and lastly populates new records for the table and commits it:

```
mysql> start transaction;
    -> Delete from one;
    -> alter table one drop COLUMN sex;
    -> alter table one ADD COLUMN Age int;
    -> insert into one (id,name,course,age) values (1,"Kimera","BCS",23);
    -> insert into one (id,name,course,age) values (2,"Dickson","BIT",35);
    -> insert into one (id,name,course,age) values (3,"Richard","BCE",54);
    -> commit//
```

Below is the content that has been committed

```
mysql> select *from one;
    -> //
+------+-------------+-----------+--------+
| id   | name        | course    | Age    |
+------+-------------+-----------+--------+
|    1 | Kimera      | BCS       |     23 |
|    2 | Dickson     | BIT       |     35 |
|    3 | Richard     | BCE       |     54 |
+------+-------------+-----------+--------+
```

Commit & Rollback

In summary we could build up a list of queries and dependant of user action could eventually send the COMMIT command,

if the user wishes to cancel the action we can send the ROLLBACK command instead and all transactions will then ROLLBACK to their previous state

- User defined Variables
- Cursors
- MySql Error Handling

User Defined Variables

- 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.
- ***User-defined variables are connection-specific***. That is, a user variable defined by one client cannot be seen or used by other clients. All variables for a given client connection are automatically freed when that client exits.
- One way to set a user-defined variable is by issuing a **SET** statement: The := symbol is the assignment operator not initialization =
- All variables start with the @ symbol e.g
  SET @year1=120, @year2=220, @year3:=40;

  View results:
  Select @yea1,@year2,@year3, @total=@year1+@year2+@year3;
- **Reference:** **http://dev.mysql.com/doc/refman/5.0/en/user-variables.html**

Cursors

- To handle a result set inside a stored procedure, you use a cursor.
- A cursor allows you to iterate a set of rows returned by a query and process each row accordingly.
- Cursors can be used in **Stored procedures**, **stored functions** and **triggers**

Properties of Cursors

- **Read only**: you cannot update data in the underlying table through the cursor.
- **Non-scrollable**: you can only fetch rows in the order determined by the SELECT statement. You cannot fetch rows in the reversed order. In addition, you cannot skip rows or jump to a specific row in the result set.
- **Asensitive:** cursors are two kinds :asensitive and insensitive
  - An asensitive cursor points to the actual data, whereas an insensitive cursor uses a temporary copy of the data.
  - An asensitive cursor performs faster than an insensitive cursor because it does not have to make a temporary copy of data.
  - However, any change that made to the data from other connections will affect the data that is being used by an asensitive cursor, therefore it is safer if you don't update the data that is being used by an asensitive cursor. MySQL cursor is asensitive.

Structure of Cursors

1.Declaring - **DECLARE** cursor_name CURSOR FOR SELECT_statement;

- A cursor is declared with the DECLARE keyword
- It must always be associated with the SELECT statement

2. **Open** the cursor by using the **OPEN** statement, The **OPEN** statement initializes the result set for the cursor therefore you must call the OPEN statement before fetching rows from the result set.

- OPEN cursor_name;

3. Use the **FETCH** statement to retrieve the next row pointed by the cursor and move the cursor to the next row in the result set.

- FETCH cursor_name INTO variables list;
- After that, you can check to see if there is any row available before fetching it.

- Finally, you call the **CLOSE** statement to deactivate the cursor and release the memory associated with it as follows:

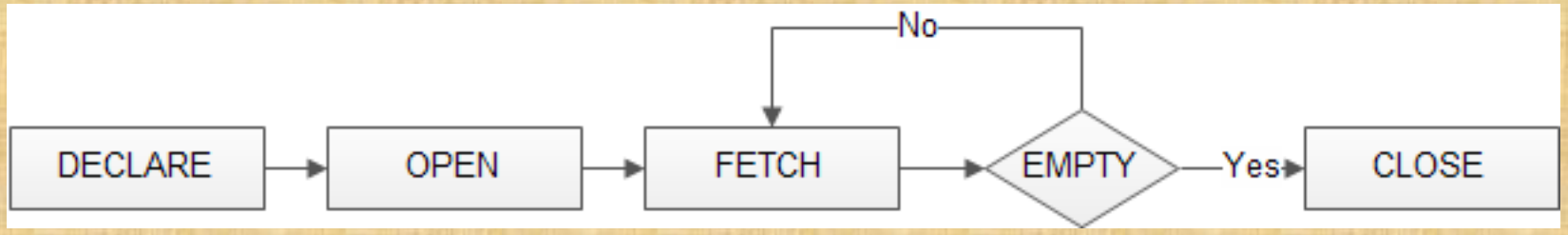
- CLOSE cursor_name;

SUCCEED WE MUST

Structure of Cursors

- When working with MySQL cursor, you must also declare a NOT FOUND handler to handle the situation when the cursor could not find any row.
- Because each time you call the FETCH statement, the cursor attempts to read the next row in the result set.
- To declare a **NOT FOUND** handler, you use the following syntax:
  - **DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;**
  - **finished** is a **variable** to indicate that the cursor has reached the end of the result set.
  - Notice that the handler declaration must appear after variable and cursor declaration inside the stored procedures.

Bellow is structure that shows the flow of a cursor



SUCCEED WE MUST

Example

- We are going to develop a stored procedure that builds an **email list** of all **lecturers** in the **lecturers** table in the database
- **First**, we declare some variables, a cursor for looping over the emails of employees, and a NOT FOUND handler

```
DECLARE finished INTEGER DEFAULT 0;
DECLARE email varchar(255) DEFAULT "";
-- declare cursor for employee email
DECLARE email_cursor CURSOR FOR SELECT email FROM
lecturer;
-- declare NOT FOUND handler
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished
= 1;
```

- Next, Open the e-mail cursor by using the OPEN statement

```
OPEN email_cursor;
```

Example

- Then, we iterate the email list, and concatenate all emails where each email is separated by a semicolon(;):

```
get_email: LOOP
FETCH email_cursor INTO v_email;
        IF v_finished = 1 THEN
        LEAVE get_email;
        END IF;
 -- build email list
        SET email_list = CONCAT(v_email,";",email_list);
        END LOOP get_email;
```

- After that, inside the loop we used the v_finished variable to check if there is any email in the list to terminate the loop
- Finally, we close the cursor, **CLOSE email_cursor;**

Example: Stored procedure with a cursor

```
DELIMITER $$
CREATE PROCEDURE build_email_list (INOUT email_list
    varchar(4000))

BEGIN
    DECLARE v_finished INTEGER DEFAULT 0;
        DECLARE v_email varchar(100) DEFAULT "";

    -- declare cursor for employee email
    DEClARE email_cursor CURSOR FOR
        SELECT email FROM lecturers;

    -- declare NOT FOUND handler
    DECLARE CONTINUE HANDLER
        FOR NOT FOUND SET v_finished = 1;

    OPEN email_cursor;
-- This code is broken down into two
```

```
-- This is a continuation of the code
get_email: LOOP

        FETCH email_cursor INTO v_email;

        IF v_finished = 1 THEN
            LEAVE get_email;
        END IF;

        -- build email list
        SET email_list = CONCAT(v_email,";",email_list);

    END LOOP get_email;

    CLOSE email_cursor;

END$$

DELIMITER ;
```

SUCCEED WE MUST

Example: Stored procedure with a cursor

```
mysql> DELIMITER $$
mysql> DROP PROCEDURE IF EXISTS build_email_list$$
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE PROCEDURE build_email_list (INOUT email_list varchar(4000))
    -> BEGIN
    -> DECLARE v_finished INTEGER DEFAULT 0;
    -> DECLARE v_email varchar(100) DEFAULT " ";
    -> DECLARE email_cursor CURSOR FOR SELECT email FROM lecturer;
    -> DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_finished = 1;
    -> OPEN email_cursor;
    -> emailoop: LOOP
    -> FETCH email_cursor INTO v_email;
    -> IF v_finished = 1 THEN
    -> LEAVE emailoop;
    -> END IF;
    -> SET email_list = CONCAT(v_email,";",email_list);
    -> END LOOP emailoop;
    -> CLOSE email_cursor;
    -> END$$
Query OK, 0 rows affected (0.00 sec)
```

Running the Procedure

- You can test the build_email_list stored procedure using the following script:
- SET @email_list = "";
- CALL build_email_list (@email_list);
- SELECT @email_list;

Running the Procedure

```
mysql> set @emailist="";
Query OK, 0 rows affected (0.00 sec)

mysql> call build_email_list (@emailist);
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> select @emailist;
+----------------------------------------------------------------------------------------------------
+
| @emailist
|
+----------------------------------------------------------------------------------------------------
+
| diddy@gmail.com;ayi@gmail.com;murungi@mail.com;kd@mail.com;kaldixo@must.ac.ug;rkimera@must.ac.ug;
|
+----------------------------------------------------------------------------------------------------
+
1 row in set (0.00 sec)
```

Error Handling in MySQL

- Like Oracle, MySQL has options for the user to specify what actions should be taken when an error occurs in a procedure or program.
- A status variable called **SQLSTATE** is used for communicating error conditions.
- A value of '02000' means no data has been found, this variable can be used to control the loop.
- You can declare a CONTINUE HANDLER that specifies that when the value of SQLSTATE is '02000', the BOOLEAN variable done, which can also be declared earlier, should be set to 1.
- When this occurs, we end the repeat. Besides the CONTINUE handler, which allows the procedure to continue to execute, we can also use EXIT, which ends the procedure. The general form for declaring a handler is
- DECLARE {CONTINUE/EXIT} HANDLER FOR *condition_value* [,*condition_value*….] statement;

Error Handling

- The *condition_value* can be a value of SQLSTATE (go to http://dev.mysql.com/doc/mysql/en/error-handling.html for a complete list),
- SQLWARNING, NOT FOUND, SQLEXCEPTION, or a MySQL_error_code. The same handler can be used for several conditions. If the condition occurs the statement specified in the declaration is executed.
- It can be either a simple statement such as , SET done=1, or a compound statement delimited by BEGIN…END.
- If the handler is a CONTINUE, the procedure continues after that statement is executed. If it is an EXIT handler, the procedure ends.
- The following command is used to create a named condition
  DECLARE condition_name CONDITION FOR condition_value;

Error Handling

- The *condition_value* may be a value of SQLSTATE or a MySQL error code. A complete list of these values and codes can be found in the MySQL documentation.
- For example, we can declare the conditions foreign_key_error and duplicate key values
  - DECLARE foreign_key_error CONDITION FOR 1216;
  - DECLARE duplicate_key CONDITION FOR SQLSTATE '23000';
- Then we can write a handler for these conditions using
  - DECLARE EXIT HANDLER FOR foreign_key_error, duplicate_key;

SUCCEED WE MUST

REFERENCES

- http://dev.mysql.com/doc/refman/5.0/en/cursors.html
- http://www.brainbell.com/tutorials/MySQL/Working_With_Cursors.htm
- http://www.kbedell.com/2009/03/02/a-simple-example-of-a-mysql-stored-procedure-that-uses-a-cursor/
- http://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html
- http://www.mysqltutorial.org/stored-procedures-loop.aspx
- http://sql-info.de/mysql/examples/create-function-examples.html

END