CSCI E-59 Designing & Developing Relational and Non-Relational Databases

Harvard University Extension, Spring 2022

Greg Misicko

Lecture 9 - Stored Routines, Transactions

@GregMisicko

Agenda

- Stored Procedures
    - What are they? What are their advantages/disadvantages?
    - Input and Output
    - Conditionals (What I want to do depends on the data I’m seeing…)
    - Loops (I want to do the same thing multiple times)
    - Error handling (Because we always make mistakes)

Next Section

Monday, April 4, 8pm ET - Anthony
Monday, April 11, 8pm ET - Anthony
Saturday, April 23, 10am ET - Marina
Saturday, April 30, 10am ET - Marina

Guest Lecturer is tentative for the second to last week of class

Homework 5 is due next week, April 8th

I’m still working on the final project details

Stored Procedures

We talked earlier in the semester about how the Structured Query Language (SQL) was designed to be simple, and it is often suggested that you describe what you want your query to do using plain English as a first step before converting your plain English statement into SQL. However, by now you've probably learned that it's not always always such a simple matter to state your query, execute it, and get your expected results back. Sometimes it takes some effort to revise and fine tune your query before it works as expected. Sometimes queries can grow very large and complex with nested queries, unions and joins adding to the complexity.

Stored Procedures can be thought of as reusable queries, invoked through their name. In programming terms you can think of them as a method, or function that you've defined.

In addition to writing up reusable queries through stored procedures, you can implement logic in them to perform conditional operations. You have the ability to use IF statements, CASE, and loops.

https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx/

Stored Procedures Advantages

Reduce network traffic
Stored procedures help reduce the network traffic between applications and MySQL Server. Because instead of sending multiple lengthy SQL statements, applications have to send only the name and parameters of stored procedures.

Centralize business logic in the database
You can use the stored procedures to implement business logic that is reusable by multiple applications. The stored procedures help reduce the efforts of duplicating the same logic in many applications and make your database more consistent.

Make database more secure
The database administrator can grant appropriate privileges to applications that only access specific stored procedures without giving any privileges on the underlying tables.

https://www.mysqltutorial.org/introduction-to-sql-stored-procedures.aspx

Stored Procedures Disadvantages

Resource usage
If you use many stored procedures, the memory usage of every connection will increase substantially.

Overusing a large number of logical operations in the stored procedures will increase the CPU usage because MySQL is not well-designed for logical operations.

Troubleshooting
It's difficult to debug stored procedures. Unfortunately, MySQL does not provide any facilities to debug stored procedures like other enterprise database products such as Oracle and SQL Server.

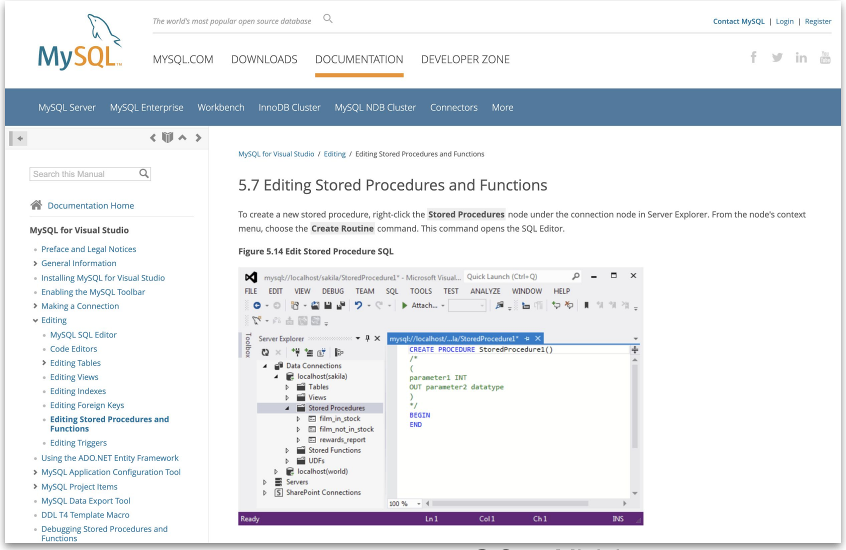
Maintenance
Developing and maintaining stored procedures often requires a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance.

Lack of Portability
You won't be able to easily move queries from one database server to another, but migrating stored procedures could be arguably more challenging than updating external code.
https://www.mysqltutorial.org/introduction-to-sql-stored-procedures.aspx

Stored Procedures Disadvantages

Writing code outside of an IDE is primitive and tedious, but if you are going to get serious about it, you can find options..

One of the various options:
https://dev.mysql.com/doc/visual-studio/en/visual-studio-editing-stored-procedures-and-functions.html



MySQL for Visual Studio

- Preface and Legal Notices
- General Information
- Installing MySQL for Visual Studio
- Enabling the MySQL Toolbar
- Making a Connection
- Editing
  - MySQL SQL Editor
  - Code Editors
  - Editing Tables
  - Editing Views
  - Editing Indexes
  - Editing Foreign Keys
  - **Editing Stored Procedures and Functions**
  - Editing Triggers
- Using the ADO.NET Entity Framework
- MySQL Application Configuration Tool
- MySQL Project Items
- MySQL Data Export Tool
- DDL T4 Template Macro
- Debugging Stored Procedures and Functions

MySQL for Visual Studio / Editing / Editing Stored Procedures and Functions

5.7 Editing Stored Procedures and Functions

To create a new stored procedure, right-click the **Stored Procedures** node under the connection node in Server Explorer. From the node's context menu, choose the **Create Routine** command. This command opens the SQL Editor.

Figure 5.14 Edit Stored Procedure SQL

```
CREATE PROCEDURE StoredProcedure1()
/*
(
parameter1 INT
OUT parameter2 datatype
)
*/
BEGIN
END
```

@GregMisicko

Stored Procedures: A Simple Query

How do we create a simple stored procedure? Let’s create one out of one of our earlier SQL statements:

```
select * from customers where first_name='Mark';
```

Not exactly a complex query or one that we’d desperately want to replicate, but it will serve as an example.

```
DELIMITER //

CREATE PROCEDURE GetCustomerNamedMark()
BEGIN
    SELECT *  FROM customers WHERE first_name='Mark';
END //

DELIMITER ;
```

Stored Procedures: How to Invoke

We've defined a stored procedure, now how do we call it? By using CALL, of course:

```
CALL GetCustomerNamedMark();
```

Like magic - we get our results as expected. The syntax for our stored procedure was pretty intuitive, except - what's up with the 'delimiter' piece?

```
DELIMITER //

CREATE PROCEDURE GetCustomerNamedMark()
BEGIN
    SELECT *  FROM customers WHERE first_name='Mark';
END //

DELIMITER ;
```

A stored procedure may consist of multiple statements, for example we could have executed more than one SELECT statement instead of just one. Each of those statements will be separated by our standard delimiter, the semi-colon. We identify a new delimiter to mark the end of our stored procedure definition.

Stored Procedures

To remove a stored procedure we can use our familiar DROP command:

```
DROP PROCEDURE GetCustomerNamedMark;
```

And to list all of our stored procedures:

```
SHOW PROCEDURE STATUS WHERE db = 'bike_stores';
```

You do not need to specify which db to list stored procedures for, but if you don't you may end up with a much longer list than you wanted (or expected).

To show the full details of the stored procedure:

```
SHOW CREATE PROCEDURE GetCustomerNamedMark;
```

Stored Procedures: Variables

As you would in a program, you can define variables to hold values for you in your stored procedures. You define a variable using DECLARE, specifying a name, and its type:

```
DECLARE <variable name> <data type> DEFAULT <default value>;
```

You feed data into your variable using INTO:

```
INTO <variable name>
```

And you can also SET a value on your variable using:

```
SET <variable name>  = <value>;
```

Stored Procedures: Declaring Variables

For example, let's say we wanted to count the unique last names in our customers table. We can declare a variable to store this value in, and then retrieve it with a SELECT statement once we want to return the value:

```
DELIMITER //

CREATE PROCEDURE countLastNames()
BEGIN
      DECLARE uniqueLastNames INT DEFAULT 0;

      select COUNT(DISTINCT(last_name))
            INTO uniqueLastNames
      FROM customers;

      SELECT uniqueLastNames;
END //

DELIMITER ;
```

Stored Procedures: Inputs

To make our stored procedures behave in a more dynamic way, we can pass them input values. We can also define output values that we'll be able to fetch after execution. Finally, we can define an input value that can be used as either.

What would we want to pass values in for? Let's say we want to identify some bit of information to help form a query. For example, we have multiple states in our customers table - what if we wanted to select customers by state?

First of all, what would this query look like on it's own? If we were searching for all customers in the state of California, it would look like this:

```
SELECT *
FROM customers
WHERE state = 'CA';
```

How can we turn this into a stored procedure where the state can be identified when calling it?

Stored Procedures: Inputs

```
DELIMITER //

CREATE PROCEDURE GetCustomersByState(
    IN targetState VARCHAR(2)
)
BEGIN
    SELECT *
    FROM customers
    WHERE state = targetState;
END //

DELIMITER ;
```

We're now stating that the user much identify a two character input string...

...which is to be used in the body of our SQL statement.

Stored Procedures: Outputs

To set an output variable:

```
DELIMITER //

CREATE PROCEDURE countCustomersByState(
    IN targetState VARCHAR(2),
    OUT totalInState INT
)
BEGIN
    SELECT *
    FROM customers
    WHERE state = targetState;

    SELECT COUNT(*)
    INTO totalInState
    FROM customers
    WHERE state = targetState;
END //

DELIMITER ;
```

And we run this using:

```
call GetCustomersByState('CA',@totalInState);
select @totalInState;
```

Stored Procedures: Conditionals

Just as with any programming language, we can use IF, THEN, ELSE to make decisions within our procedures.

IF statements are defined within a block. You identify what it is you want to check for (for example: IF something is true) and then identify what should happen if that condition is met.

```
IF condition THEN
   statements;
ELSE
   statements;
END IF;
```

If your condition is not met, you can identify what ELSE should happen, or try another IF evaluation.

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

Stored Procedures: Conditionals

Let's try to do the following:
Identify whether or not one specific student from the students table in our school database has a GPA which is greater than the average GPA of all students.

How can we break this problem down?

- We will need to take student ID as an input
- identify the average GPA of all students
- check to see if our specified student has a GPA that is greater than the average
    - if so, say so
    - else, say not

Stored Procedures: Conditional Statements

Here is a little procedure to declare whether a student from our students table in our school database has an above average grade, or below average grade:

```
DELIMITER $$

CREATE PROCEDURE RateStudent(
    IN  studentID INT,
    OUT studentRating  VARCHAR(25))
BEGIN
    DECLARE averageGPA DECIMAL(2,1) DEFAULT 0.0;
    DECLARE studentGPA DECIMAL(2,1) DEFAULT 0.0;

    SELECT avg(gpa) FROM students INTO averageGPA;

    SELECT gpa INTO studentGPA
    FROM students
    WHERE student_id = studentID;

    IF studentGPA > averageGPA THEN
        SET studentRating = 'Above average student';
    ELSE
        SET studentRating = 'Below average student';
    END IF;

    SELECT studentRating;
END$$

DELIMITER ;
```

These variables only live within the running procedure, they disappear after execution

@GregMisicko

Stored Procedures: CASE Statements

CASE statements are another type of conditional which can be used to identify which course of action to take on certain conditions:

```
DELIMITER //

CREATE PROCEDURE RateStudentByCase(
    IN  studentID INT,
    OUT studentRating  VARCHAR(25))
BEGIN
    DECLARE averageGPA DECIMAL(2,1) DEFAULT 0.0;
    DECLARE studentGPA DECIMAL(2,1) DEFAULT 0.0;

    SELECT avg(gpa) FROM students INTO averageGPA;

    SELECT gpa INTO studentGPA
    FROM students
    WHERE student_id = studentID;

    CASE studentGPA
        WHEN studentGPA > averageGPA THEN
           SET studentRating = 'Above average student';
        WHEN studentGPA = averageGPA THEN
           SET studentRating = 'Perfectly average student';
        ELSE
           SET studentRating = 'Below average student';
    END CASE;

    SELECT studentRating;
END //

DELIMITER ;
```

@GregMisicko

Stored Procedures: CASE Statements

The code on the previous slide is **wrong**, so don't expect it to work (the CASE statement is used incorrectly, you can fix it by deleting a single word in the procedure).

How do we go about identifying the issues and correcting them?

Without something like an IDE (integrated development environment) it can be very difficult to identify bugs and perform debug operations. There are a few simple things we can do:

- We can include the equivalent of 'print' statements in our code to get some insight into the values of our variables
  - for example: we could add '`SELECT studentGPA`' if we wanted to verify what value was being stored in this variable
- Use essential reference material such as https://dev.mysql.com/doc/refman/8.0/en/

Stored Procedures: CASE Statements

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

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Or:

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Knowing this, our previous code can be made to work by using the correct version of CASE statement, which is the second one.

Stored Procedure: ALTER

I want to edit my stored procedure for some reason, can I use ALTER?

https://dev.mysql.com/doc/refman/8.0/en/alter-procedure.html

Yes, but it doesn’t really allow you to make a useful modification. If you want enhanced edit features, you WILL need to use an editing tool.

Stored Procedures: String Manipulation

It’s worth mentioning that if you are going to work with strings, you can manipulate them using Functions that were covered in Lecture 7. For example, if we wanted to customize text using the results of a query:

```
SELECT first_name INTO studentName from students WHERE student_id = studentID;

IF studentGPA > averageGPA THEN
    SET studentRating = concat(studentName, ': Above average student');
ELSE
    SET studentRating = concat(studentName, ' is a below average student';
END IF;
```

Also, always remember that we can effectively print to the console by using a SELECT statement which retrieves a string as a result:

```
SELECT concat(studentName, ': Above average student') AS message;
```

would simply write the string to our console for us. Remember: this can be useful when performing basic debugging!

Stored Procedures: Loops

There are various ways to repeat actions in a Stored Procedure. Different RDBMS's provide different options, MySQL offers you:
LOOP/LEAVE
WHILE
REPEAT

LEAVE is a break statement which will allow you to exit any loop type you wish to.

All of these will provide you the ability to perform repetitive actions, and all of them will allow you to stop the loop based on some condition. We'll take a look at a few of the options available.

Stored Procedures: WHILE

WHILE tells our procedure to do something for as long as a specified condition is true. The basic format looks like this:

```
DELIMITER //

CREATE PROCEDURE DoStuff()
BEGIN

    WHILE <condition> DO
        <do stuff>
    END WHILE;

END//

DELIMITER ;
```

Stored Procedures: LOOP

LOOP tells our procedure to do something forever. The basic format looks like this:

```
LOOP
    statement_list
END LOOP
```

and can optionally provide labels for the loops:

```
[loop_label:] LOOP
    statement_list
END LOOP [loop_label]
```

Stored Procedures: LOOP

Let's take a look at a LOOP example and the key parts of it:
https://dev.mysql.com/doc/refman/8.0/en/statement-labels.html

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN ITERATE label1; END IF;
    LEAVE label1;
  END LOOP label1;
END;
```

We have a few new things standing out here, in particular the ITERATE and LEAVE.
ITERATE basically tells our procedure to start the loop over again. LEAVE will break from the loop.

Stored Procedures: Handling Errors

We often forget certain scenarios when writing our code, which leads to errors in execution. It's essential in programming to think through all the things which might go wrong, and implement a backup plan for when they do.

The declaration of a handler is not complicated, but there are details that need to be understood about its parts:

```
DECLARE action HANDLER FOR condition_value statement;
```

The action has two options:
CONTINUE: if this condition is realized, even if it is an error of some sort, the execution of the stored procedure will continue.
EXIT: if this condition is realized, execution is stopped.

The `condition_value` identifies which kind of problem we are handling:
SQLWARNING: there is a problem with what you tried to do, but it is not necessarily an error
NOTFOUND: something you tried to fetch is not there, or you reached the end of a data set
SQLEXCEPTION: an error has occurred

Stored Procedures: Handling Errors

Note that there are specific codes that are recognized by MySQL that you can build handlers for. For example, a duplicate key will result in error code 1062 and could be handled with something like:

```
DECLARE EXIT HANDLER FOR 1062
BEGIN
    SELECT CONCAT('Duplicate key (',supplierId,',',productId,') occurred') AS message;
END;
```

How is the above doing anything to “handle” the error? No specific action is being performed, but an error message is being displayed.

Using this approach you can handle individual error types, rather than have a single way to handle all SQLEXCEPTION ‘s
Error Code ranges:
https://dev.mysql.com/doc/refman/8.0/en/error-message-elements.html#error-code-ranges
Full listing of error codes:
https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html
Example code:
https://www.mysqltutorial.org/mysql-error-handling-in-stored-procedures/

Stored Procedures: Handling Errors

Note that there are specific codes that are recognized by MySQL that you can build handlers for. For example, a duplicate key will result in error code 1062 and could be handled with something like:

```
DECLARE EXIT HANDLER FOR 1062
BEGIN
    SELECT CONCAT('Duplicate key (',supplierId,',',productId,') occurred') AS message;
END;
```

Using this approach you can handle individual error types, rather than have a single way to handle all SQLEXCEPTION s For example:

```
    DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate keys error encountered' Message;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'SQLException encountered' Message;
```

Stored Procedures: Cursors

So far, all of the statements we've used inside of stored procedures have dealt with the return of a single value. If those SQL statements were to return multiple values, we would get an error. If you want to use a SQL statement in your stored procedure which returns more than one value, you use a CURSOR.

Technically, our single value returns were an **implicit cursor**. An **explicit cursor** is one which returns more than one value.

Once you have defined a CURSOR you work with it using the following commands:
OPEN: populates the cursor with data
FETCH: retrieves individual rows of data
CLOSE: closes the cursor for processing

Stored Procedures: Cursors Example

```
CREATE PROCEDURE cursordemo()
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;
```

Used to track whether or not we've reached the end of our result set (MySQL doesn't have a BOOLEAN data type)

Some data

Cursors which will fetch 1 or more values per record

Handler for when we've run out of data in our cursor

If 'done' has been set to TRUE, break out of the loop

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