MySQL Stored Procedures

By Peter Gulutzan

Book 1 in the “MySQL 5.0 New Features” Series
Second Edition – Revised for MySQL 5.1

Contents

Technical Matters
Introduction
A definition and an example
Why stored procedures?
Setup with MySQL 5.0
CREATE PROCEDURE example
What statements are legal in a procedure body
Call the procedure
Characteristics clauses
Digressions
Exercise
Parameters
Compound statements
Variables
Conditions and IF-THEN-ELSE
CASE
Loops
Error Handling
Cursors
Security
Dynamic PREPARE and EXECUTE
Functions
Metadata
Details
Oracle Comparison
SQL Server Comparison
DB2 Comparison
Standard Comparison
Style
Stored Procedure example: tables_concat()
Function example: rno()
Function example: running_total()
Procedure example: MyISAM "foreign key" insertion
Procedure example: Error propagation
Procedure example: Library
Procedure example: Hierarchy
Tips when writing long routines
Bugs
Feature requests
Resources
Books
The End

Acknowledgments

For design of stored procedures: the ANSI/ISO “SQL Standards” committee.

For the original code: Per-Erik Martin of Uppsala, Sweden.

For enhancing and fixing the code: Antony Curtis, Sergey Glukhov, Dmitri Lenev.

For many technical suggestions and reviewing of this book: Trudy Pelzer.

About the Author

Peter Gulutzan, MySQL AB Software Architect, worked with stored procedures for a different DBMS for several years. He is co-author of three SQL books. His job at MySQL AB involves looking for any deviations from orthodox SQL, finding test procedures, and being the pioneer who tries to use the new code with real databases. He lives in Edmonton, Canada.

Technical Production Matters

Originally written in December 2004 with OpenOffice, using these fonts:
Regular text: Times, regular, 12
Computer output: Courier, bold, 12
Title: Times, bold, 36
Part header: Times, bold, 20
Sub-part header: Times, bold, 16

Date of last revision: June 1 2006

Introduction 4

This book is for long-time MySQL users who want to know "what's new" in version 5. The short answer is "stored procedures, triggers, views, information_schema". The long answer is the “MySQL 5.0 New Features” series, and this book is the first in the series. This is the revised edition, the book is up-to-date for MySQL 5.1.

What I'm hoping to do is make this look like a hands-on session where you, as if you're working it out yourself on your keyboard, can walk through sample problems.

I'll go through each little item, building up slowly. By the end, I'll be showing larger routines that do something useful, something that you might have thought was tough.

Conventions and Styles

Whenever I want to show actual code, such as something that comes directly from the screen of my mysql client program, I switch to a Courier font, which looks different from the regular text font. For example:

```
mysql> DROP FUNCTION f;
Query OK, 0 rows affected (0.00 sec)
```

When the example is large and I want to draw attention to a particular line or phrase, I highlight with a double underline and a small arrow on the right of the page. For example:

```
mysql> CREATE PROCEDURE p ()
    -> BEGIN
    ->   /* This procedure does nothing */              <--
    -> END;//
Query OK, 0 rows affected (0.00 sec)
```

Sometimes I will leave out the “mysql>” and “->” prompts so that you can cut the examples and paste them into your copy of the mysql client program. (If you aren't reading the text of this book in a machine-readable form, try looking for the script on the mysql.com web site.)

I tested all the examples with the publicly-available beta version of MySQL 5.1.12 on Linux, SUSE 10.0. By the time you read this, the MySQL version number will be higher. The available operating systems include Windows, Linux, Solaris, BSD, AIX, NetWare, Mac OS X, and HP-UX. So I'm confident that you'll be able to run every example on your computer. But if not, well, as an experienced MySQL user you know that help and support is always available.

A definition and an example 5

A stored procedure is a *procedure* (like a subprogram in a regular computing language) that is s*tored* (in the database). Correctly speaking, MySQL supports "routines" and there are two kinds of routines: stored procedures which you call, or functions whose return values you use in other SQL statements the same way that you use pre-installed MySQL functions like `PI()`. I'll use the word "stored procedures" more frequently than "routines" because it's what we've used in the past, and what people expect us to use.

A stored procedure has a name, a (possibly empty) parameter list, and an SQL statement, which can contain many more SQL statements. There is new syntax for local variables, error handling, loop control, and IF conditions. Here is an example of a statement that creates a stored procedure.

```
CREATE PROCEDURE procedure1           /* name */
(IN parameter1 INTEGER)               /* parameters */
BEGIN                                 /* start of block */
  DECLARE variable1 CHAR(10);           /* variables */
  IF parameter1 = 17 THEN               /* start of IF */
    SET variable1 = 'birds';                /* assignment */
  ELSE
    SET variable1 = 'beasts';               /* assignment */
  END IF;                               /* end of IF */
  INSERT INTO table1 VALUES (variable1);/* statement */
END                                   /* end of block */
```

What I'm going to do is explain in detail all the things you can do with stored procedures. We'll also get into another new database object, triggers, because there is a tendency to associate triggers with stored procedures.

Why Stored Procedures? 6

Stored procedures have been part of the officially-supported MySQL release for a year, so now there's a track record and a large body of user experience that proves they're the way to go. Understandably, you'll still be cautious. Nevertheless, you should start to think now about moving your code out of where it is now (an application host program, a UDF, a script), and into a stored procedure. The reasons for using procedures are compelling.

Stored procedures are proven technology! Yes, they are new in MySQL, but the same functionality exists in other DBMSs, and often precisely the same syntax too. So there are concepts that you can steal, there are people with experience whom you can consult or hire, there is third-party documentation (books or web pages) that you can read.

Stored procedures are fast! Well, we can't prove that for MySQL yet, and everyone's experience will vary. What we can say is that the MySQL server takes some advantage of caching, just as prepared statements do. 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.

Stored procedures are components! Suppose that you change your host language -- no problem, the logic is in the database not the application.

Stored procedures are portable! When you write your stored procedure in SQL, you know that it will run on every platform that MySQL runs on, without obliging you to install an additional runtime-environment package, or set permissions for program execution in the operating system, or deploy different packages if you have different computer types. That's the advantage of writing in SQL rather than in an external language like Java or C or PHP. Don't get me wrong about this: I know there are sometimes excellent reasons to support external-language routines, they just lack this particular advantage.

Stored procedures are stored! If you write a procedure with the right naming conventions, for example saying chequing_withdrawal for a bank transaction, then people who want to know about chequing can find your procedure. It's always available as 'source code' in the database itself. And it makes sense to link the data with the processes that operate on the data, as you might have heard in your programming-theory classes.

Stored procedures are migratory! MySQL adheres fairly closely to the SQL:2003 standard. Others (DB2, Mimer) also adhere. Others (Oracle, SQL Server) don't adhere but I'll be providing tips and tools that make it easier to take code written for another DBMS and plunking it into MySQL.

```
mysql_fix_privilege_tables
```

or

```
~/mysql-5.1/scripts/mysql_install_db
```

As part of the preparation for our exercises, I'll assume MySQL 5.0 or MySQL 5.1 is installed. If you're not working in a place where database administrators install software and databases for you, you'll have to install it yourself. One thing that's easy to forget is that you need to have a table named mysql.proc. After you've installed the latest version, you should run either mysql_fix_privilege_tables or mysql_install_db -- or stored procedures won't work for you. I also run an undocumented SQL script after starting as --user=root.

Starting the mysql client

This is how I start the mysql client. You might do it differently, for example you might run from a different subdirectory if you have a binary version or a Windows computer.

```
pgulutzan@mysqlcom:~> /usr/local/mysql/bin/mysql --user=root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.1.12-
beta-debug

Type 'help;' or '\h' for help. Type '\c' to clear the
buffer.
```

During this demonstration, I will be showing you what the results look like from the mysql client, which I started after establishing myself as the root user. This means I have lots of privileges.

Check for the Correct Version

To ensure that I'm using the right MySQL version, I display the version in use. I want to be sure that I have version 5. I have two ways to check this:

```
SHOW VARIABLES LIKE 'version';
```

or

```
SELECT VERSION();
```

For example:

```
mysql> SHOW VARIABLES LIKE 'version';
+---------------+-------------------+
| Variable_name | Value             |
+---------------+-------------------+
| version       | 5.1.12-beta-debug |
+---------------+-------------------+
1 row in set (0.03 sec)

mysql> SELECT VERSION();
+-------------------+
| VERSION()         |
+-------------------+
| 5.1.12-beta-debug |
+-------------------+
1 row in set (0.01 sec)
```

When I see '5.0.x' or '5.1.x', I know that stored procedures will work. The differences between 5.0 and 5.1, for stored procedures at least, are slight and rare.

The Sample "Database"

The first thing I do is create a new database and make it my default database.
The SQL statements that I need for this are:

```
CREATE DATABASE db5;
USE db5;
```

For example:

```
mysql> CREATE DATABASE db5;
Query OK, 1 row affected (0.00 sec)

mysql> USE db5;
Database changed
```

I avoid using a real database that might have important data in it.

And now I'm making a simple table to work with. The statements that I use for this are:

```
mysql> CREATE DATABASE db5;
Query OK, 1 row affected (0.01 sec)

mysql> USE db5;
Database changed

mysql> CREATE TABLE t (s1 INT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO t VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

You'll notice that I'm only inserting one row into the table. I want my sample table to be simple. I'm not showing off table manipulation here, I'm showing off stored procedures, and they're complex enough without worrying about big tables.

So the sample database that we'll start with is a table named t with one row in it.

Pick a delimiter

Now I need a delimiter. The statement I'll use is

```
DELIMITER //.
```

For example:

```
mysql> DELIMITER //
```

The delimiter is the character or string of characters that you'll use to tell the mysql client that you've finished typing in an SQL statement. For ages, the delimiter has always been a semicolon. That causes a problem because, in a stored procedure, one can have many statements, and every one must end with a semicolon.

So for your delimiter, pick a string which is very unlikely to occur within any statement, or within any procedure. I've used slash slash. Others use a vertical bar. I've seen '@' in DB2 procedures but I don't like it. You can use whatever you want (except '\'), but during this session it's probably easier to copy me. To resume using “;” as a delimiter later say
`DELIMITER ;`

CREATE PROCEDURE example 10

```
CREATE PROCEDURE p1 () SELECT * FROM t; //
```

Perhaps this is the first stored procedure that you've ever made with MySQL. If so, be sure to mark this event in your diary.

```
CREATE PROCEDURE p1 () SELECT * FROM t; //              <--
```

The first part of the SQL statement that creates a stored procedure is the words "CREATE PROCEDURE".

```
CREATE PROCEDURE p1 () SELECT * FROM t; //              <--
```

The second part is the procedure name. The name of this new procedure will be p1.

DIGRESSION: Legal Identifiers

Procedure Names are not case sensitive, so 'p1' and 'P1' are the same name.

You cannot use two procedures with the same name in the same database. That would be *overloading*. Some DBMSs allow overloading. MySQL doesn't. You can use qualified names of the form “database-name . procedure-name” , for example “db5.p1”.

Procedure names can be delimited. If the name is delimited, it can contain spaces. The maximum name length is 64 characters. But avoid using using names which are already names of built-in MySQL functions. Here's what can happen if you do:

```
mysql> CALL pi();
Error 1064 (42000): You have a syntax error.
mysql> CALL pi ();
Error 1305 (42000): PROCEDURE does not exist.
```

In the first example, I used a function named pi. It's legal, but you must put a space after the name when you're calling it, as I did in the second example.

CREATE PROCEDURE example (continued) 11

```
CREATE PROCEDURE p1 () SELECT * FROM t; //                <--
```

“()” is the 'parameter list'.

The third part of the CREATE PROCEDURE statement is the parameter list. It's always necessary to have a parenthesized parameter list here. In this procedure, there are no parameters, so the parameter list is empty -- so all I had to type is the parentheses. But they are compulsory.

```
CREATE PROCEDURE p1 () SELECT * FROM t; //                <--
```

“SELECT * FROM t;” is the body.

And finally, the last part of the statement is the procedure body, which is an SQL statement. Here the body is “SELECT * FROM t;” which includes a semicolon “;”. The semicolon is optional when there is a real statement-ender (the “//”) following it.

It's probably good if you remember that I'm calling this part the body of the procedure, because body is a technical term that everybody uses.

Ordinarily, it's not normal to put SELECT statements in stored procedures, this is for illustration. I decided that some procedure should simply select from our table, so that when you call the procedure it will be obvious that it's working.

What SQL Statements Are Legal In A Procedure Body 12

What SQL statements are legal in the body of a MySQL procedure? You can make a procedure that contains pretty well anything, including INSERT, UPDATE, DELETE, SELECT, DROP, CREATE, REPLACE, and so on. The only important point to remember is that your code won't be portable if you include a MySQL extension.

As in Standard SQL: any database manipulation language statements are legal:

```
CREATE PROCEDURE p () DELETE FROM t; //
```

Also, SET and COMMIT and ROLLBACK are legal:

```
CREATE PROCEDURE p () SET @x = 5; //
```

MySQL extra feature: any database definition language statements are legal:

```
CREATE PROCEDURE p () DROP TABLE t; //
```

MySQL extension: a direct SELECT is legal:

```
CREATE PROCEDURE p () SELECT 'a'; //
```

By the way, I call the ability to include DDL statements in a procedure a MySQL "extra feature" because the SQL Standard says that this is “non-core” which means it's optional.

One restriction is that, inside a procedure body, you can't put database-definition statements that affect a routine. For example this is illegal:

```
CREATE PROCEDURE p1 ()
 CREATE PROCEDURE p2 () DELETE FROM t; //
```

So these statements, which are all new in MySQL 5.0, are illegal in a procedure body: CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE, CREATE FUNCTION, DROP FUNCTION, CREATE TRIGGER. Soon CREATE EVENT, which is new in MySQL 5.1, will be illegal too. On the other hand, you can say “CREATE PROCEDURE db5.p1 () DROP DATABASE db5//”.

Statements like “USE database” are also illegal. MySQL assumes that the default database is the one that the procedure is in.

And LOAD DATA INFILE is illegal. And LOCK TABLES is illegal. And CHECK is illegal.

Call the procedure 13

(1)

Now, to call a procedure, all you have to do is enter the word CALL and then the name of the procedure and then the parentheses. In MySQL I can omit the parentheses but I never do.

When you do this for our example procedure p1, the result will be that you see the contents of the table t on your screen.

```
mysql> CALL p1() //
+------+
| s1   |
+------+
|    5 |
+------+
1 row in set (0.03 sec)
Query OK, 0 rows affected (0.03 sec)
```

That's reasonable. The body of the procedure is a "SELECT * FROM t;" statement.

(2)

Let me say that again, another way.

```
mysql> CALL p1() //
```

does the same as:

```
mysql> SELECT * FROM t; //
```

So, when you call the procedure, it's exactly as if you'd executed "SELECT * FROM t;".

Okay, the fundamental point – that you can CREATE a procedure then CALL it – is clear now, eh? I hope that you are even saying to yourself that this is rather simple. But now we're going to do a series of exercises where we add one clause at a time, or vary one of the existing clauses. There will be many such clauses, even before we get to complex parts.

Characteristics Clauses 14

(1)

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL                                          <--
NOT DETERMINISTIC                                     <--
CONTAINS SQL                                          <--
SQL SECURITY DEFINER                                  <--
COMMENT 'A Procedure'                                 <--
SELECT CURRENT_DATE, RAND() FROM t //
```

Let's make a procedure that has some clauses which describe the characteristics of the procedure. The clauses come after the parentheses, but before the body. These clauses are all optional. What good are they?

(2)

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL                                          <--
NOT DETERMINISTIC
CONTAINS SQL
SQL SECURITY DEFINER
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

Well, the LANGUAGE SQL clause is no good at all. It simply means that the body of the procedure is written in SQL. And that's always true. But it's not a bad idea to put this clause in, because there's another DBMS which requires it -- namely IBM DB2. So use this if you care about DB2 compatibility. Besides, in the future, MySQL may support calling procedures in other languages besides SQL.

(3)

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC                                          <--
CONTAINS SQL
SQL SECURITY DEFINER
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

The next clause, NOT DETERMINISTIC, is informational. A deterministic procedure, something like the religious concept of predestination, is a procedure which will always return the same outputs given the same data inputs. In this case, since the body of the procedure has a SELECT whose results are not predictable, I've called it NOT DETERMINISTIC. But the MySQL optimizer does not care, at least at this time.

(4)

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC                                          <--
CONTAINS SQL
SQL SECURITY DEFINER
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

The next clause, CONTAINS SQL, is informational. In fact all procedures contain SQL statements, but if they also read databases (SELECT) then it would be proper to say READS SQL DATA, and if they also write databases (UPDATE etc.) then it would be proper to say MODIFIES SQL DATA. It is never proper to say NO SQL, although MySQL allows that too.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC
CONTAINS SQL
SQL SECURITY DEFINER                                          <--
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

The next clause, SQL SECURITY, can be either SQL SECURITY DEFINER or SQL SECURITY INVOKER. This gets us into the realm of privileges. We're going to have an exercise that tests privileges later on.

SQL SECURITY DEFINER means 'at CALL time, check privileges of user who created the procedure' (the other choice is SQL SECURITY INVOKER)

For the moment, it's enough to note that SQL SECURITY DEFINER is an instruction that tells the MySQL server to check the privileges of the user who created the procedure -- at the time that the procedure is called, regardless of which user is doing the CALL. The other option tells the server to check the caller's privileges instead.

Characteristics Clauses (continued) 17

(5)

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC
CONTAINS SQL
SQL SECURITY DEFINER
COMMENT 'A Procedure'                                    <--
SELECT CURRENT_DATE, RAND() FROM t //
```

COMMENT 'A procedure' is an optional remark.

And finally, the comment clause is something that will be stored along with the procedure definition. It is non-standard. I'll point out anything non-standard as I go along, but luckily that will be rare.

(6)

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC
CONTAINS SQL
SQL SECURITY DEFINER
COMMENT ''
SELECT CURRENT_DATE, RAND() FROM t //
```

is the same as:

```
CREATE PROCEDURE p2 ()
SELECT CURRENT_DATE, RAND() FROM t //
```

The characteristics clauses have defaults. If I omit them all, that's the same as saying LANGUAGE SQL NOT DETERMINISTIC CONTAINS SQL SQL SECURITY DEFINER COMMENT ''.

Digression: The effect of CALL p2() //

```
mysql> call p2() //
+--------------+-----------------+
| CURRENT_DATE | RAND()          |
+--------------+-----------------+
| 2006-06-01   | 0.7822275075896 |
+--------------+-----------------+
1 row in set (0.26 sec)
Query OK, 0 rows affected (0.26 sec)
```

And when we call procedure p2, a SELECT is executed which returns the current date and a random number, which is what we expect.

Digression: sql_mode unchanging

```
mysql> set sql_mode='ansi' //
mysql> create procedure p3()select'a'||'b'//
mysql> set sql_mode=''//
mysql> call p3()//
+------------+
| 'a' || 'b' |
+------------+
| ab         |
+------------+
```

MySQL silently preserves the environment at the time that the procedure is created too. For example, suppose we use two bars when we are concatenating strings. That is only legal while the sql mode is ansi. If we change the sql mode to non-ansi, that doesn't matter. The call will still work, as if the original setting is still true.

Exercise 19

Question

If you don't mind exercise, ask yourself whether you could handle this request without peeking at the Answer below.

Create a procedure. The procedure should display 'Hello, world'.

Please take about five seconds to contemplate the problem.

Given what you have read so far, this should be easy. While you think about it, I'll just review some random choice things that I've said already: The DETERMINISTIC clause is a characteristics clause that means the input determines the output reliably ... The statement for calling a procedure is CALL procedure_name (parameter list). Well, I guess that's about enough time.

Answer

Okay. The answer is that you'd make a procedure with a body that contains a a "SELECT 'Hello, world'" statement.

```
mysql> CREATE PROCEDURE p4 () SELECT 'Hello, world' //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p4()//
+--------------+
| Hello, world |
+--------------+
| Hello, world |
+--------------+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

Parameters 20

Let's look more closely at how you can define parameters in a stored procedure.

1. **CREATE PROCEDURE p5**
   () ...

2. **CREATE PROCEDURE p5**
   ([IN] name data-type) ...

3. **CREATE PROCEDURE p5**
   (OUT name data-type) ...

4. **CREATE PROCEDURE p5**
   (INOUT name data-type) ...

Recall that the parameter list is what's between the parentheses just after the stored procedure name.

In the first example the parameter list is empty.

In the second example there is one input parameter. The word IN is optional because parameters are IN (input) by default.

In the third example there is one output parameter.

In the fourth example there is one parameter which is both input and output.

IN example

```
mysql> CREATE PROCEDURE p5(p INT) SET @x = p //
Query OK, 0 rows affected (0.00 sec)
mysql> CALL p5(12345)//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x//
+-------+
| @x    |
+-------+
| 12345 |
+-------+
1 row in set (0.00 sec)
```

The IN example shows a procedure with an input parameter. In the body of the procedure, I say that I want to set a session variable named x to the value of whatever the value of the parameter p is. Then I call the procedure and pass 12345 as the argument for parameter p. Then I select the session variable, to prove that it got the value that I passed, 12345.

OUT example

```
mysql> CREATE PROCEDURE p6 (OUT p INT)
    -> SET p = -5 //
mysql> CALL p6(@y)//
mysql> SELECT @y//
+------+
| @y   |
+------+
| -5   |
+------+
```

Now here's an example where I go the other way. This time p is the name of an output parameter and I'm passing its value to a session variable named @y, in the CALL statement.

In the body of the procedure, I say that the parameter will get the value minus 5. And after I call the procedure, I see that is what happened. So the word OUT tells the DBMS that the value goes out from the procedure.

The effect is the same as if I had executed the statement “SET @y = -5;”.

Let's expand what's in the procedure body now.

```
CREATE PROCEDURE p7 ()
BEGIN
  SET @a = 5;
  SET @b = 5;
  INSERT INTO t VALUES (@a);
  SELECT s1 * @a FROM t WHERE s1 >= @b;
END; //       /* I won't CALL this. */
```

The construct you need, in order to do this, is a BEGIN/END block. This is similar to the BEGIN/END blocks that appear in languages like Pascal, or to the braces that appear in languages like C. You use the block to enclose multiple statements. In this example, I've put in a couple of statements that change the value of some session variables, and then I've done some insert and select statements.

You need a BEGIN/END block when you have more than one statement in the procedure. But that's not all. The BEGIN/END block, also called a compound statement, is the place where you can define variables and flow-of-control.

The New SQL Statements

The new statements:
 * are part of the ANSI/ISO “Persistent Stored Modules” (PSM) language
 * are legal within compound (BEGIN ... END) statements
 * include a DECLARE statement for variable declaration
 * include IF, LOOP, WHILE, REPEAT, etc. for flow of control.

There are several new statements that you can use within a BEGIN END block. They were defined by the SQL standard document called Persistent Stored Modules, or PSM. The new statements let you define variables and loops, just as you can do in any ordinary procedural programming language.

In the next pages I'll talk more about the new statements.

The DECLARE statement is used to define local variables in a BEGIN..END statement.

(1) Example with two DECLARE statements

```
CREATE PROCEDURE p8 ()
BEGIN
  DECLARE a INT;
  DECLARE b INT;
  SET a = 5;
  SET b = 5;
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //       /* I won't CALL this */
```

You don't really define variables within the stored procedure. You define them within the BEGIN/END block.

Notice that these variables are not like session variables:

You don't start them with an at sign (@).

You must declare them explicitly at the start of the BEGIN/END block, along with their data types.

Once you've declared a variable, you can use it anywhere that you would otherwise use any session variable, or literal, or column name.

(2) Example with no DEFAULT clause and SET statement

```
CREATE PROCEDURE p9 ()
BEGIN
  DECLARE a INT /* there is no DEFAULT clause */;
  DECLARE b INT /* there is no DEFAULT clause */;
  SET a = 5;    /* there is a SET statement */
  SET b = 5;    /* there is a SET statement */
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //       /* I won't CALL this */
```

There are several ways to initialize a variable. When declared without a DEFAULT clause, the initial value of a variable is always NULL. You can use the SET statement to assign another value to a variable.

(3) Example with DEFAULT clause

```
CREATE PROCEDURE p10 ()
BEGIN
  DECLARE a, b INT DEFAULT 5;
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //
```

Here's a variation that does the same thing. This time I'm putting both variable declarations on the same line and using a DEFAULT clause to set the initial value, rather than doing two separate DECLARE and SET statements. Both a and b are DEFAULT 5.

(4) Example of CALL

```
mysql> CALL p10() //
+--------+
| s1 * a |
+--------+
|     25 |
|     25 |
+--------+
2 rows in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

Once again, I'm just calling the procedure to show you that it works.

(5) Scope

```
CREATE PROCEDURE p11 ()
BEGIN
  DECLARE x1 CHAR(5) DEFAULT 'outer';
  BEGIN
    DECLARE x1 CHAR(5) DEFAULT 'inner';
    SELECT x1;
  END;
  SELECT x1;
END; //
```

Now let's think about scope. This Scope Example procedure has a BEGIN/END block within another BEGIN/END block. That's fine, perfectly legal.

It also has two variables, both named x1. That's still legal. In this case, the inner variable declaration takes precedence as long as it's in scope.

The point is that the inner variable disappears when the first END is reached. That's what "out of scope" means. The variable ceases to be visible at this point. Therefore you can never see a declared variable outside a stored procedure; however, you can assign it to an OUT parameter or assign it to a session variable.

So now I'll call the Scope Example procedure.

```
mysql> CALL p11()//
+-------+
| x1    |
+-------+
| inner |
+-------+
+-------+
| x1    |
+-------+
| outer |
+-------+
```

What you see in the output is: that the first SELECT retrieves the inner variable, and the second SELECT retrieves the outer variable.

(1)

Now we can do something that involves conditions.

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

Here's a stored procedure that contains an IF statement. Well, it contains two, actually. One is just IF dah-dah-dah END IF. The other is IF dah-dah-dah ELSE dah-dah-dah END IF.

I'm trying to show that it's possible to have something complex, but at the same time I'm trying to keep it simple enough that you can figure out what's happening.

(2)

```
CALL p12 (0) //
```

Suppose we call this procedure and pass a zero. So parameter1 will be zero.

(3)

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;                    <--
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

The first thing that happens is that variable1 gets set to parameter1 plus one, which means it gets set to zero plus one -- so it will be one.

(4)

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN                              <--
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

The next thing that happens is nothing. Since variable1 is one, the condition "variable1 = 0" isn't true. Therefore everything between the first IF and the matching END IF gets skipped.

(5)

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN                                    <--
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

So now we go on to the second IF statement. And for this statement, the condition is true, because we know that parameter1 equals zero. That's what we passed.

(6)

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;                               <--
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

And since it's true that parameter1 equals zero, this UPDATE is executed. If it had been false, or if parameter1 had been null, the other UPDATE would have been executed instead.

There are currently two rows in table t, and they both contain the value 5, so now if we call p12, both rows should contain 6.

(7)

```
mysql> CALL p12(0)//
Query OK, 2 rows affected (0.28 sec)

mysql> SELECT * FROM t//
+------+
| s1   |
+------+
|    6 |
|    6 |
+------+
2 rows in set (0.01 sec)
```

And wow, they do.

CASE 30

(1)

```
CREATE PROCEDURE p13 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  CASE variable1
    WHEN 0 THEN INSERT INTO t VALUES (17);
    WHEN 1 THEN INSERT INTO t VALUES (18);
    ELSE INSERT INTO t VALUES (19);
  END CASE;
END; //
```

In addition to the IF statement, there is another way to check for conditions and take paths according to whether expressions are true or non-true. It's the CASE statement.

This CASE statement could just as easily have been expressed with IF statements. If the value of variable1 is zero, then insert seventeen into table t. If it's one, then insert eighteen. If it's neither zero nor one, then insert nineteen.

(2)

```
mysql> CALL p13(1)//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t//
+------+
| s1   |
+------+
|    6 |
|    6 |
|   19 |
+------+
3 rows in set (0.00 sec)
```

So we execute the procedure, passing a one. We expect that the result is that a value of nineteen will be inserted into table t.

And there it is. It's nice to see how predictable procedures are.

Question

QUESTION: WHAT DOES CALL p13(NULL) // do?

And now, another assignment. The question is: what will this CALL statement do? You can either find out by executing it and then seeing what the SELECT does, or you can figure it out by looking at the code. Give yourself five seconds to figure it out.

Answer

```
mysql> CALL p13(NULL)//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t//
+------+
| s1   |
+------+
|    6 |
|    6 |
|   19 |
|   19 |
+------+
4 rows in set (0.00 sec)
```

Okay, the answer is that when you call procedure p13, MySQL inserts one more row containing 19. The reason is, of course, that since the value of variable1 is NULL, the ELSE portion of the CASE statement is processed. I am hoping that makes sense to everyone. Remember that if you can't understand an answer, it's okay, we'll just keep going forward as if you do.

```
WHILE ... END WHILE
LOOP ... END LOOP
REPEAT ... END REPEAT
```

Now we're going to make some loops. There are actually three standard ways to loop: WHILE loops, LOOP loops, and REPEAT loops. We used to have a non-standard way, GOTO, but it's gone now.

WHILE ... END WHILE

```
CREATE PROCEDURE p14 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  WHILE v < 5 DO
    INSERT INTO t VALUES (v);
    SET v = v + 1;
  END WHILE;
END; //
```

Here's one way, the WHILE loop. It's my favourite because it looks much like an IF statement, so there's less new syntax to learn.

The INSERT and the SET statements here, which are between the WHILE and the END WHILE, happen over and over until the value in variable v becomes greater than or equal to five. I said “SET v = 0;” first to avoid a common trap: if I don't initialize a variable with DEFAULT or SET, then it's NULL, and adding 1 to NULL always yields NULL.

WHILE ... END WHILE example

```
mysql> CALL p14()//
Query OK, 1 row affected (0.00 sec)
```

Here's what happens when we CALL procedure p14.

Try not to worry that it says "one row affected" instead of "five rows affected". The count applies only for the last INSERT.

WHILE ... END WHILE example: CALL

```
mysql> select * from t; //
+------+
| s1   |
+------+
....
|    0 |
|    1 |
|    2 |
|    3 |
|    4 |
+------+
9 rows in set (0.00 sec)
```

And here you see that five inserts happened when we did the CALL.

REPEAT ... END REPEAT

```
CREATE PROCEDURE p15 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  REPEAT
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    UNTIL v >= 5
  END REPEAT;
END; //
```

Here's another way of doing a loop, the REPEAT loop. It's doing the same thing as the WHILE loop did, except that it checks the condition after doing the statements, instead of checking the condition before doing the statements, as the WHILE loop did.

REPEAT ... END REPEAT: look at the UNTIL

```
CREATE PROCEDURE p15 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  REPEAT
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    UNTIL v >= 5                                          <--
  END REPEAT;
END; //
```

Note that there is no semicolon after the UNTIL clause in this procedure statement. This is one of the few times that you can have trouble with semicolon rules. Normally it's okay if you put an extra semicolon in somewhere, which is what I tend to do.

REPEAT ... END REPEAT: calling

```
mysql> CALL p15()//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT COUNT(*) FROM t//
+----------+
| COUNT(*) |
+----------+
|       14 |
+----------+
1 row in set (0.00 sec)
```

And here you see that five more inserts happened when we called procedure p15.

LOOP ... END LOOP

```
CREATE PROCEDURE p16 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
    END IF;
  END LOOP;
END; //
```

Here's another way to write a loop: the LOOP loop. The LOOP loop doesn't have to have a condition at the start, like a WHILE loop does. And it doesn't have to have a condition at the end, like a REPEAT loop does.

LOOP ... END LOOP: with IF and LEAVE

```
CREATE PROCEDURE p16 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN                                    <--
      LEAVE loop_label;
    END IF;
  END LOOP;
END; //
```

Instead, you put an IF statement somewhere inside the loop.

And inside that IF statement, you put a LEAVE statement. The LEAVE statement means "exit the loop". The actual syntax of the LEAVE statement is the word LEAVE and a statement label. I'll talk more about this statement label in a moment.

LOOP ... END LOOP: calling

```
mysql> CALL p16()//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT COUNT(*) FROM t//
+----------+
| COUNT(*) |
+----------+
|       19 |
+----------+
1 row in set (0.00 sec)
```

Let's call procedure p16. The result is that another 5 rows are inserted into table t.

Labels

```
CREATE PROCEDURE p17 ()
label_1: BEGIN
  label_2: WHILE 0 = 1 DO LEAVE label_2; END
WHILE;
  label_3: REPEAT LEAVE label_3; UNTIL 0 =0
END REPEAT;
  label_4: LOOP LEAVE label_4; END LOOP;
END; //
```

In the last LOOP example, I used a statement label. Now here is an example of a procedure with four statement labels. It's possible to use a statement label before BEGIN, before WHILE, before REPEAT, or before LOOP. Those are the only places where a statement label is a legal start to a statement.

Thus the statement “LEAVE label_3 doesn't mean “jump to label_3”. Rather, it means “leave the (compound or looping) statement which is identified by label_3”.

End Labels

```
CREATE PROCEDURE p18 ()
label_1: BEGIN
  label_2: WHILE 0 = 1 DO LEAVE label_2; END
WHILE label_2;
  label_3: REPEAT LEAVE label_3; UNTIL 0 =0
END REPEAT label_3 ;
  label_4: LOOP LEAVE label_4; END LOOP
label_4 ;
END label_1 ; //
```

One can also have statement labels at the end, as well as at the start, of these same statements. These ending labels aren't terribly useful. They're optional. If you put one in, it has to be the same as the starting label. In a long piece of code, the end label can prevent confusion -- it's easier to see what end goes with what start.

LEAVE and Labels

```
CREATE PROCEDURE p19 (parameter1 CHAR)
label_1: BEGIN
  label_2: BEGIN
    label_3: BEGIN
      IF parameter1 IS NOT NULL THEN
        IF parameter1 = 'a' THEN
          LEAVE label_1;
        ELSE BEGIN
          IF parameter1 = 'b' THEN
            LEAVE label_2;
          ELSE
            LEAVE label_3;
            END IF;
          END;
        END IF;
      END IF;
    END;
  END;
END;//
```

LEAVE statements get me out of deeply-nested compound statements, too.

ITERATE

There is one other thing that labels are necessary for, as targets for ITERATE statements.

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

The ITERATE statement, like the LEAVE statement, appears within loops and refers to loop labels. In that case, ITERATE means "start the loop again".

ITERATE is a bit like “continue” in the C language.

(Also: The ITERATE statement, like the LEAVE statement, appears within compound statements and refers to compound-statement labels. In that case, ITERATE means “start the compound statement again”. But I haven't illustrated that here.)

So, for this example, let's start walking through this loop. We'll do it a few times.

ITERATE: Walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP                                          <--
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

We're looping through a loop labelled with loop_label.

ITERATE: Walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN                                           <--
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

Eventually the value of v will be three. Then we'll increase it to four.

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;                                    <--
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

Then we'll do the ITERATE.

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP                                           <--
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

Which causes execution to start again at the beginning of the loop.

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;                                    <--
      END IF;
    END LOOP;
END; //
```

Then variable v becomes five, so we reach the LEAVE statement.

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //                                                     <--
```

As a result of the LEAVE, we exit the loop and reach the end of the compound statement.

```
CREATE PROCEDURE p21
(IN parameter_1 INT, OUT parameter_2 INT)
LANGUAGE SQL DETERMINISTIC SQL SECURITY INVOKER
BEGIN
  DECLARE v INT;
  start_label: LOOP
    IF v = v THEN LEAVE start_label;
    ELSE ITERATE start_label;
    END IF;
  END LOOP start_label;
  REPEAT
    WHILE 1 = 0 DO BEGIN END;
    END WHILE;
    UNTIL v = v END REPEAT;
END;//
```

Here's a statement that has everything in it that you've seen so far. This is a procedure with parameters, characteristics, a BEGIN/END block, statements within statements, a variable declaration, a label, an IF, a WHILE, a LOOP, a REPEAT, a LEAVE, and an ITERATE.

It's a ridiculous procedure and I wouldn't run it because it contains infinite loops. But it does contain all legal syntax.

These are all the new flow-of-control and variable declaration statements. Now, I'm going to move on to a different thing.

Error Handling 44

Summary of the next few pages:
Sample Problem
Handlers
Conditions

Okay, the thing we're moving on to now is error handling.

Sample Problem: Log Of Failures (1)

When INSERT fails, I want to record the failure in a log file.

The sample problem we'll use to show error handling is a common one. I want to have a log of errors. When an INSERT fails, I want to put a notice into a separate file somewhere that the insert failed, when it failed, and why it failed. Let's say that I'm particularly interested in inserts because of violations of foreign-key constraints.

Sample Problem: Log Of Failures (2)

```
mysql> CREATE TABLE t2
       s1 INT, PRIMARY KEY (s1))
       engine=innodb;//
mysql> CREATE TABLE t3 (s1 INT, KEY (s1),
       FOREIGN KEY (s1) REFERENCES t2 (s1))
       engine=innodb;//
mysql> INSERT INTO t3 VALUES (5);//
...
ERROR 1216 (23000): Cannot add or update a child row: a
foreign key constraint fails
```

I start off by making a primary-key table, and a foreign-key table. I'm using InnoDB, so foreign key checks should work. Then I try to insert a value in the foreign-key table that isn't in the primary-key table. That will fail, I know. This is just a quick way to find out what the error number is for the condition. It's Error Number one two one six.

Sample Problem: Log Of Failures (3)

```
CREATE TABLE error_log (error_message
CHAR(80))//
```

Next, I make a table in which I am going to store errors that occur when I'm trying to do my inserts.

Sample Problem: Log Of Errors (4)

```
CREATE PROCEDURE p22 (parameter1 INT)
BEGIN
  DECLARE EXIT HANDLER FOR 1216
    INSERT INTO error_log VALUES
    (CONCAT('Time: ',current_date,
     '. Foreign Key Reference Failure For
Value = ',parameter1));
  INSERT INTO t3 VALUES (parameter1);
END;//
```

Here's my procedure. The first statement here, DECLARE EXIT HANDLER, is what handles exceptions. This is saying that if error one two one six happens, then the procedure will insert a row into the error log table. The word EXIT means "we'll exit from the compound statement when we're done".

Sample Problem: Log Of Errors (5)

```
CALL p22 (5) //
```

Now let's call the procedure. It will fail, of course, because the value five doesn't appear in the primary key table. But there is no error return, because the error is handled inside the procedure.

When I look at the contents of table t3, I see that nothing got inserted into table t3. But when I look at the contents of table error_log, I see that something did get added into table error_log. This tells me that an INSERT into table t3 failed.

DECLARE HANDLER syntax

```
DECLARE
{ EXIT | CONTINUE }
HANDLER FOR
{ error-number | { SQLSTATE error-string } | condition }
SQL statement
```

Okay, so that's how a handler works. It's a piece of code that gets triggered when an error happens. MySQL in fact allows two kinds of handlers. One is the EXIT handler, which we just saw.

Next we'll look at the other kind of handler, the CONTINUE handler. It's similar to the EXIT handler, but after it executes, the play continues. There's no exit from the compound statement.

DECLARE CONTINUE HANDLER example (1)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

There's an example of a CONTINUE handler in the MySQL Reference Manual, and it's so good that I'll just copy it here.

This will demonstrate how a continue handler works.

DECLARE CONTINUE HANDLER example (2)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;                    <--
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

I'll define the handler for an SQLSTATE value this time. Remember that I used a MySQL error code last time, error one two one six. Actually, SQLSTATE two three zero zero zero is a little more general. It happens for either a foreign-key violation or a primary-key violation.

DECLARE CONTINUE HANDLER example (3)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;                                           <--
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

The first statement of this procedure that gets executed is "SET @x = 1".

DECLARE CONTINUE HANDLER example (4)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);                                    <--
 SET @x = 3;
 END;//
```

Then the value one gets inserted into the primary-key table.

DECLARE CONTINUE HANDLER example (5)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;                                                   <--
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

Then the value of @x becomes two.

DECLARE CONTINUE HANDLER example (6)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);                                    <--
 SET @x = 3;
 END;//
```

Then an attempt is made to insert the same value into the primary-key table again. But it fails, because primary keys must be unique.

DECLARE CONTINUE HANDLER example (7)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;                            <--
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

Now, because the insert fails, the handler takes over. The next statement that gets executed is the handler's statement, so @x2 gets set to one.

DECLARE CONTINUE HANDLER example (8)

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR  SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;                                                    <--
 END;//
```

But that's not the end. This is a CONTINUE handler. So now the procedure goes back to where we were, after the failed insert statement. And it sets @x to three.

DECLARE CONTINUE HANDLER example (9)

```
mysql> CALL p23()//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x, @x2//
+------+------+
| @x   | @x2  |
+------+------+
| 3    | 1    |
+------+------+
1 row in set (0.00 sec)
```

Run the procedure.

Here we look at the value of @x. And sure enough, it is three.

Next we look at the value of @x2. And sure enough, it is one.

So that's our proof that the operation was processed in the way that I described.

People can take a bit of time to adjust to handlers. The check is at the start of the statement block, instead of right after the statement that might cause the error, so it looks as if we are jumping around. But it's quite a safe and clean way to code.

DECLARE CONDITION (1)

```
CREATE PROCEDURE p24 ()
BEGIN
  DECLARE `Constraint Violation`
   CONDITION FOR SQLSTATE '23000';
  DECLARE EXIT HANDLER FOR
   `Constraint Violation` ROLLBACK;
  START TRANSACTION;
  INSERT INTO t2 VALUES (1);
  INSERT INTO t2 VALUES (1);
  COMMIT;
  END; //
```

There's another variation on the error handler theme. You can, in effect, give the SQLSTATE or the error code another name. And then you can use that name in a handler. Let's see how that works.

Recall that I defined table t2 as an InnoDB table, so the INSERTs that I make into this table will be subject to ROLLBACK. And a ROLLBACK is precisely what will happen. Because inserting the same value twice into a primary key will cause SQLSTATE '23000' to happen, and I say here that SQLSTATE '23000' is a constraint violation. (The formal syntax description is “CONDITION FOR SQLSTATE [VALUE] 'sqlstate-string'” or “CONDITION FOR mysql-error-number” so some variations exist.)

DECLARE CONDITION (2)

```
CREATE PROCEDURE p24 ()
BEGIN
  DECLARE `Constraint Violation`
   CONDITION FOR SQLSTATE '23000';
  DECLARE EXIT HANDLER FOR
   `Constraint Violation` ROLLBACK;
  START TRANSACTION;
  INSERT INTO t2 VALUES (1);
  INSERT INTO t2 VALUES (1);
  COMMIT;
  END; //
```

And I say here that a constraint violation causes ROLLBACK.

DECLARE CONDITION (3)

```
mysql> CALL p24()//
Query OK, 0 rows affected (0.28 sec)

mysql> SELECT * FROM t2//
Empty set (0.00 sec)
```

So let's call the procedure. And let's see what the result was.

Yes, we see that nothing got INSERTed into table t2. The entire transaction was rolled back. That's the result I wanted.

DECLARE CONDITION (4)

```
mysql> CREATE PROCEDURE p9 ()
    -> BEGIN
    ->   DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;
    ->   DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN END;
    ->   DECLARE EXIT HANDLER FOR SQLWARNING BEGIN END;
    -> END;//
Query OK, 0 rows affected (0.00 sec)
```

There are three predeclared conditions: NOT FOUND (no more rows), SQLEXCEPTION (error), SQLWARNING (warning or note). Since they're predefined, you can use them without saying DECLARE CONDITION. Indeed, if you tried to say something like “DECLARE SQLEXCEPTION CONDITION ...” you'd get an error message.

Summary of things that we can do with cursors:

```
DECLARE cursor-name CURSOR FOR SELECT ...;
OPEN cursor-name;
FETCH cursor-name INTO variable [, variable];
CLOSE cursor-name;
```

Now let's look at cursors. We have a somewhat incomplete implementation of the syntax for cursors inside stored procedures. But we can do the main things. We can declare a cursor, we can open it, we can fetch from it, and we can close it.

Cursor Example

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

Let's walk through a new example with a procedure that contains cursor statements.

Cursor Example (2)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;                                          <--
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

The procedure starts with three DECLARE statements. Incidentally, the order is important. First declare variables. Then declare conditions. Then declare cursors. Then, declare handlers. If you put them in the wrong order, you will get an error message.

Cursor Example (3)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;                <--
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

We next have a declaration of a cursor. This looks almost exactly like what's in embedded SQL, for anyone who has worked with that.

Cursor Example (4)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND                    <--
    SET b = 1;                                              <--
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

The final declaration, as I said, is of a handler. This continue handler does not refer to a MySQL error number or to an SQLSTATE value. Instead it uses NOT FOUND. This happens to be the same as error one three one six or SQLSTATE '02000'.

Cursor Example (5)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;                                               <--
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

The first executable statement is OPEN cur_1. Since cur_1 is associated with "SELECT s1 FROM t", the procedure will SELECT s1 FROM t, producing a result set.

Cursor Example (6)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;                                    <--
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

The first FETCH statement here will cause a single row to be retrieved from the result set that the SELECT produced. Since we know that there are several rows in table t, we know that this statement will happen several times -- that is, it is in a loop.

Cursor Example (7)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;                                             <--
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

Eventually, MySQL will have no more rows to fetch. At that point, we will reach the statement that's in the CONTINUE handler. That is, set variable b to one.

Cursor Example (8)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;                                          <--
  SET return_val = a;
END;//
```

Therefore the condition UNTIL b = 1 is now true. Therefore the loop ends. So now I close the cursor. If I didn't, it would be closed automatically when the compound statement ends. But I think depending on automatic behaviour looks a tad sloppy.

Cursor Example (9)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;                                   <--
END;//
```

Now we'll assign our local variable to an output parameter, so that the results will be visible after the procedure finishes.

Cursor Example (10)

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//

mysql> CALL p25(@return_val)//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @return_val//
+-------------+
| @return_val |
+-------------+
| 5           |
+-------------+
1 row in set (0.00 sec)
```

Let's call this procedure.

Here you see that the return_val parameter got a value of 5, which is right, because that is what is in the last row of table t. So we know that the cursor worked, and we know that the handler worked.

Cursor Characteristics

Summary:

- **READ ONLY**
- **NOT SCROLLABLE**
- **ASENSITIVE**

In version five of MySQL, you can only fetch from cursors, you cannot update them. That is, cursors are READ ONLY. You cannot say:

```
FETCH cursor1 INTO variable1;
UPDATE t1 SET column1 = 'value1' WHERE CURRENT OF cursor1;
```

Cursors are also NOT SCROLLABLE. That is, you can only fetch the next row, you cannot move back and forth in the result set. You cannot say:

```
FETCH PRIOR cursor1 INTO variable1;
FETCH ABSOLUTE 55 cursor1 INTO variable1;
```

And you should avoid doing updates on a table while you have a cursor open on the same table, because cursors are ASENSITIVE. That is, if you don't avoid doing updates, we don't guarantee what the results will be. I know that they can be different if you use the InnoDB storage engine instead of the MyISAM storage engine.

```
Privileges (1) CREATE ROUTINE
Privileges (2) EXECUTE
Privileges (3) INVOKERS AND DEFINERS
```

I'm going to say a few things about privileges and security now. But this section will have to be short, since there's not much to it. We just use familiar GRANTs and REVOKEs.

Privileges (1) CREATE ROUTINE

```
GRANT CREATE ROUTINE
ON database-name . *
TO user(s)
[WITH GRANT OPTION];
* But right now, just use 'root'
```

There is a privilege for creating procedures or functions, just as there is a privilege for creating views or tables. It's the CREATE ROUTINE privilege. The root user has it.

Privileges (2) ALTER ROUTINE

```
GRANT ALTER ROUTINE
{ ON PROCEDURE procedure-name } | { ON database-name . *}
TO user(s)
[WITH GRANT OPTION];
```

There is also an ALTER ROUTINE privilege.

Privileges (3) EXECUTE

```
GRANT EXECUTE
{ ON PROCEDURE procedure-name } | { ON database-name . *}
TO user(s)
[WITH GRANT OPTION];
```

Also there is a special privilege that says whether you are allowed to use, or execute, a procedure.

A procedure's creator has ALTER ROUTINE and EXECUTE privileges automatically.

Privileges (4) Invokers and Definers

```
CREATE PROCEDURE p26 ()
  SQL SECURITY INVOKER
  SELECT COUNT(*) FROM t //
CREATE PROCEDURE p27 ()
  SQL SECURITY DEFINER
  SELECT COUNT(*) FROM t //
GRANT EXECUTE ON db5.* TO peter; //
```

Now let's try out the SQL SECURITY clause. Security is one of the characteristics of a procedure that I mentioned at the start.

You are 'root'. Grant execute rights on your database (db5) to peter. Then start up a new job as peter, and let's see what peter can do with stored procedures.

Notice: peter does not have the right to select from t. Only root has the right to select from t.

Privileges (5) Invokers and Definers

```
/* Logged on with current_user = peter */

mysql> CALL p26();
ERROR 1142 (42000): select command denied to user
 'peter'@'localhost' for table 't'

mysql> CALL p27();
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

So when peter tries to call procedure p26, the one with INVOKER security, he fails. That's because peter can't select from the table. However, when Peter tries to call procedure p27, the one with DEFINER security, he succeeds. That's because root can select from the table, and Peter has root's rights as long as the procedure is being executed.

Hmm, I had better qualify that last remark. I said “Peter has root's rights as long as the procedure is being executed”. But it would be better to say that Peter “impersonates root” or “acts as root”. You can see this by selecting CURRENT_USER inside and outside an SQL SECURITY DEFINER procedure.

```
/* Logged on with current_user = root */

mysql> CREATE PROCEDURE p27a () /* DEFINER is default */
    -> SELECT CURRENT_USER //
Query OK, 0 rows affected (0.00 sec)

/* Logged on with current_user = peter */

mysql> SELECT CURRENT_USER;
+-----------------+
| CURRENT_USER    |
+-----------------+
| peter@localhost |
+-----------------+
1 row in set (0.00 sec)

mysql> CALL p27a();
+----------------+
| CURRENT_USER   |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.01 sec)
```

Dynamic PREPARE and EXECUTE 64

This is a relatively new feature. You can manipulate a string which contains an SQL statement, then you can PREPARE and EXECUTE that statement. This on-the-fly generation is called “Dynamic SQL” in some texts.

```
mysql> CREATE PROCEDURE p30 (search_value INT)
    -> BEGIN
    ->   SET @sql = CONCAT(
    ->     'SELECT * FROM t WHERE s1 = ',search_value);
    ->   PREPARE stmt1 FROM @sql;
    ->   EXECUTE stmt1;
    -> END; //
Query OK, 0 rows affected (0.01 sec)

mysql> CALL p30(0)//
+------+
| s1   |
+------+
|    0 |
+------+
1 row in set (0.00 sec)
```

The example shows how flexible our dynamic SQL is. Build a string, prepare it, execute it. Ta-da.

In theory, you could build any statement a user types in, or any statement that an algorithm can generate. The limitations are:

- Some statements are not preparable. (The number is decreasing rapidly so I won't list them here.)
- You lose a bit of speed because MySQL has to parse the statements you create.
- It's harder to test and debug a stored procedure which can change drastically.
- You can't use a declared variable for the statement, you have to use a session @variable. Side effects can be unfortunate for recursive procedures (procedures which call themselves), or for two different procedures which by chance use the same variable names.

Summary:
CREATE FUNCTION
Limitations of functions

You've now seen pretty well everything that can be in a stored procedure. What you haven't seen, what I've left unmentioned up till now, is what a stored FUNCTION looks like. So it's function time.

(1) CREATE FUNCTION

```
CREATE FUNCTION factorial (n DECIMAL(3,0))
    RETURNS DECIMAL(20,0)
    DETERMINISTIC
  BEGIN
    DECLARE factorial DECIMAL(20,0) DEFAULT 1;
    DECLARE counter   DECIMAL(3,0);
    SET counter = n;
    factorial_loop: REPEAT
      SET factorial = factorial * counter;
      SET counter = counter - 1;
    UNTIL counter = 1
    END REPEAT;
    RETURN factorial;
  END //
```

(Source: “Understanding SQL's stored procedures”. Used for example purposes only.)

Functions look a lot like procedures. The only noticeable syntax differences are that we say create function instead of create procedure, and we need a RETURNS clause to say what the data type of the function is, and we need a RETURN statement because a function must return a value.

This example comes courtesy of a book by Jim Melton, a member of the SQL standard committee. The book is called "Understanding SQL's stored procedures". This example is on page 223 of that book. I decided to use it because MySQL uses standard SQL, and so the example in the book works out of the box. I only had to clear up two typos.

(2) Examples

```
INSERT INTO t VALUES (factorial(pi())) //
SELECT s1, factorial (s1) FROM t //
UPDATE t SET s1 = factorial(s1)
WHERE factorial(s1) < 5 //
```

Now that I've got my function, I can just plop it into any SQL statement, and it will look like any other function.

To put it mildly, this is beautiful. Now I can expand hugely the set of built-in functions that come with MySQL. The number of things I can do with this is limited only by my imagination. And, regrettably, limited because there a few little things MySQL won't allow.

(3): Limitations

Some statements cannot be in stored procedures, and others won't run.

These statements will cause a “not allowed” message at CREATE time:

```
ALTER ANALYZE BACKUP 'CACHE INDEX' CHECKSUM COMMIT CREATE
DEALLOCATE DROP EXECUTE EXPLAIN 'FLUSH PRIVILEGES' LOAD
LOCK OPTIMIZE PREPARE RENAME REPAIR RESTORE ROLLBACK
'SELECT [without INTO]' SHOW 'START TRANSACTION' TRUNCATE
USE
```

These statements will cause a “not locked” message at CALL time:

```
GRANT REVOKE
```

In general, MySQL tries to stop functions from doing commits (explicitly or implicitly), or from returning result sets.

If you refer to a table from both inside and outside a function, you get an error. Example:

```
mysql> CREATE FUNCTION f3 ()
    -> RETURNS INT
    -> BEGIN
    ->   INSERT INTO t VALUES (0);
    ->   RETURN 5;
    -> END; //
Query OK, 0 rows affected (0.01 sec)

mysql> UPDATE t SET s1 = f3()//
ERROR 1442 (HY000): Can't update table 't' in stored
function/trigger because it is already used by statement
which invoked this stored function/trigger.
```

(4): Non-Limitations

Legal:

```
'BEGIN END' CLOSE DECLARE DELETE FETCH HANDLER IF INSERT
ITERATE LEAVE LOOP OPEN REPEAT REPLACE RETURN 'SET declared
variable' UPDATE WHILE
```

You can set variables, and use them with the new flow of control statements. You can use cursors. You can use most data manipulation statements. This is actually quite powerful, and is about what reasonable people expect.

Summary:
SHOW CREATE PROCEDURE / SHOW CREATE FUNCTION
SHOW PROCEDURE STATUS / SHOW FUNCTION STATUS
SELECT from mysql.proc
SELECT from information_schema

At this point, we've made many procedures. They're stored in a MySQL database. Let's look at the information MySQL has actually stored. There are currently four ways to do this. Two of them involve SHOW statements. Two of them involve SELECT statements.

(1) SHOW

```
mysql> show create procedure p6//
+-----------+----------+--------------------+
| Procedure | sql_mode | Create Procedure   |
+-----------+----------+--------------------+
| p6        |          | CREATE PROCEDURE   |
|           |          | `db5`.`p6`(out p   |
|           |          | int) set p = -5    |
+-----------+----------+--------------------+
1 row in set (0.00 sec)
```

The first way to get the metadata information is to execute SHOW CREATE PROCEDURE. It's like SHOW CREATE TABLE and other similar MySQL statements. It doesn't return exactly what you typed in when you made the procedure, but it's close enough for most purposes.

(2) SHOW

```
mysql> SHOW PROCEDURE STATUS LIKE 'p6'//
+------+------+-----------+----------------+
| Db   | Name | Type      | Definer        | ...
+------+------+-----------+----------------+
| db5  | p6   | PROCEDURE | root@localhost | ...
+------+------+-----------+----------------+
1 row in set (0.01 sec)
```

The second way to get metadata information is to execute SHOW PROCEDURE STATUS. This gives you a bit more detail.

(3) SELECT from mysql.proc

```
SELECT * FROM mysql.proc WHERE name = 'p6'//
+------+------+-----------+---------------+
| db   | name | type      | specific_name | ...
+------+------+-----------+---------------+
| db5  | p6   | PROCEDURE | p6            | ...
+------+------+-----------+---------------+
1 row in set (0.00 sec)
```

The third way to get metadata information is to execute a SELECT statement. This method provides the most detail, and it's done with SELECT instead of with SHOW.

(4) SELECT from information_schema: My Favourite

The fourth way to get metadata is with “SELECT ... FROM information_schema. ...”.
I have a strong bias towards doing things the “ANSI/ISO standard” way, so I believe that this is the only good way – the other ways are design errors. Of course, you might have guessed that there are other MySQL people who strongly hold other opinions. So you can take these reasons seriously, or skeptically:

1. Other DBMSs, e.g. SQL Server 2005, use information_schema. MySQL's SHOW is proprietary.
2. There is no guarantee that you have privileges to access mysql.proc, while there is a guarantee that you can access information_schema views. Everybody always has an implicit SELECT privilege on all tables in the information_schema database.
3. SELECT has a multitude of options like getting expressions, grouping, ordering, and of course producing a result set that you can FETCH from. SHOW has no such things.

Well, now that you know that I like it, you are (I hope) asking: okay, so what is it?

As usual, I'll try to answer that with a few examples.

First, I'll use an information_schema SELECT to show what columns are in information_schema.routines.

```
mysql> SELECT TABLE_NAME, COLUMN_NAME, COLUMN_TYPE FROM
INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_NAME = 'ROUTINES';//
+------------+--------------------+--------------+
| TABLE_NAME | COLUMN_NAME        | COLUMN_TYPE  |
+------------+--------------------+--------------+
| ROUTINES   | SPECIFIC_NAME      | varchar(64)  |
| ROUTINES   | ROUTINE_CATALOG    | longtext     |
| ROUTINES   | ROUTINE_SCHEMA     | varchar(64)  |
| ROUTINES   | ROUTINE_NAME       | varchar(64)  |
| ROUTINES   | ROUTINE_TYPE       | varchar(9)   |
| ROUTINES   | DTD_IDENTIFIER     | varchar(64)  |
| ROUTINES   | ROUTINE_BODY       | varchar(8)   |
| ROUTINES   | ROUTINE_DEFINITION | longtext     |
| ROUTINES   | EXTERNAL_NAME      | varchar(64)  |
| ROUTINES   | EXTERNAL_LANGUAGE  | varchar(64)  |
| ROUTINES   | PARAMETER_STYLE    | varchar(8)   |
| ROUTINES   | IS_DETERMINISTIC   | varchar(3)   |
| ROUTINES   | SQL_DATA_ACCESS    | varchar(64)  |
| ROUTINES   | SQL_PATH           | varchar(64)  |
| ROUTINES   | SECURITY_TYPE      | varchar(7)   |
| ROUTINES   | CREATED            | varbinary(19)|
| ROUTINES   | LAST_ALTERED       | varbinary(19)|
| ROUTINES   | SQL_MODE           | longtext     |
| ROUTINES   | ROUTINE_COMMENT    | varchar(64)  |
| ROUTINES   | DEFINER            | varchar(77)  |
+------------+--------------------+--------------+
20 rows in set (0.01 sec)
```

Nifty, eh? Whenever I want to know about an information_schema view, I select from another information_schema view such as TABLES or COLUMNS. It's metadata of the metadata.

So here we'll see how many procedures I have defined in database db5:

```
mysql> SELECT COUNT(*) FROM INFORMATION_SCHEMA.ROUTINES
    -> WHERE ROUTINE_SCHEMA = 'db5';//
+----------+
| COUNT(*) |
+----------+
|       28 |
+----------+
1 row in set (0.02 sec)
```

And now we'll have a closer look at procedure 'p1' which was the first procedure I made. I reformatted the mysql client display, but the information is what mysql says.

```
| SPECIFIC_NAME | ROUTINE_CATALOG | ROUTINE_SCHEMA
+---------------+-----------------+--------------
• p19           | NULL            | p19

| ROUTINE_NAME  | ROUTINE_TYPE    | DTD_IDENTIFIER
+---------------+-----------------+--------------
| p19           | PROCEDURE       | NULL

| ROUTINE_BODY | ROUTINE_DEFINITION | EXTERNAL_NAME
+--------------+--------------------+-------------
| SQL          | select * from t    | NULL

| EXTERNAL_LANGUAGE | PARAMETER_STYLE | IS_DETERMINISTIC
|-------------------+-----------------+-----------------
| NULL              | SQL             | NO

| SQL_DATA_ACCESS | SQL_PATH | SECURITY_TYPE | CREATED
+-----------------+----------+---------------+--------
| CONTAINS SQL    | NULL     | DEFINER       | 2006-06-01

| CREATED             | LAST_ALTERED        | SQL_MODE
+---------------------+---------------------+---------
| 2006-06-01 15:00:26 | 2006-06-01 15:00:26 |

| ROUTINE_COMMENT | DEFINER
+-----------------+--------------
|                 | root@localhost
```

Access control for the ROUTINE_DEFINITION column

The ROUTINE_DEFINITION column in INFORMATION_SCHEMA.ROUTINES has the body of the procedure or function, for example “select * from t”. That might be sensitive information, so the column value is invisible to everyone but the creator.

That is: if the person doing the SELECT isn't the same as the the person who created – if CURRENT_USER <> INFORMATION_SCHEMA.ROUTINES.DEFINER – then MySQL returns a blank for the ROUTINE_DEFINITION column.

Additional clause in SHOW PROCEDURE STATUS

Now that I've listed what the columns are in INFORMATION_SCHEMA.ROUTINES, I can go back and explain a new detail about SHOW PROCEDURE STATUS. Syntax:

```
SHOW PROCEDURE STATUS [WHERE condition];
```

The condition works like a condition in a SELECT statement: if it's true, the row appears in the output of SHOW PROCEDURE STATUS. But here's the weird part: in the WHERE clause you must use INFORMATION_SCHEMA column names, but in the display you'll see SHOW PROCEDURE STATUS field names. For example:

```
mysql> SHOW PROCEDURE STATUS WHERE Db = 'db6';//
Empty set (0.03 sec)

mysql> SHOW PROCEDURE STATUS WHERE ROUTINE_NAME = 'p1';//
+------+------+-----------+----------------+
| Db   | Name | Type      | Definer        | ...
+------+------+-----------+----------------+
| db5  | p    | PROCEDURE | root@localhost | ...
+------+------+-----------+----------------+
1 row in set (0.00 sec)
```

Perhaps, knowing this, I can score higher on the MySQL Certification Exam. I would never consider putting this feature to practical use, though.

Summary of next pages:
ALTER and DROP
Recursive
Definer
Oracle / SQL Server / DB2 / ANSI comparison
Style
Bugs
Feature Requests
Resources

And now I've crammed a lot of details at the end. We won't look at all of these with the degree of attention that they deserve, but that's okay, trial and error will reveal their secrets.

ALTER and DROP

```
ALTER PROCEDURE p6 COMMENT 'Unfinished' //
DROP PROCEDURE p6 //
```

DEFINER

```
CREATE DEFINER=user_name PROCEDURE p31 () BEGIN END; //
```

Recursive

Suppose, inside procedure p1, we have a statement “CALL p1();”. That is, the procedure calls itself. This is called recursion. MySQL limits recursion so that errors will be less severe. So before using recursion, check and adjust the limit.

```
mysql> SHOW VARIABLES LIKE '%recur%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| max_sp_recursion_depth | 0     |
+-------------------------+-------+
1 row in set (0.00 sec)

mysql> SET @@max_sp_recursion_depth = 100;
Query OK, 0 rows affected (0.00 sec)
```

Summary:

- Oracle allows DECLARE after OPEN
  MySQL requires DECLARE be at the start

- Oracle allows “CURSOR cursorname IS”
  MySQL requires “DECLARE cursorname CURSOR”

- Oracle doesn't require '()'
  MySQL requires '()'

- Oracle allows accessing the same tables in and out of functions
  MySQL does not allow accessing the same tables in and out of functions

- Oracle supports “packages”.
  MySQL does not support “packages”.

If you have used Oracle's PL/SQL stored procedures in the past, you'll have noticed that there are some differences between Oracle's and MySQL's stored procedures. There are, in fact, quite a few differences. I've only noted the common ones.

See also: http://www.mysql.com/products/tools/migration-toolkit/ and http://www.ispirer.com. These are web sites that describe tools for converting Oracle stored procedures to MySQL stored procedures. I do not say that these products are good. I have not tried them. I just thought it is interesting that somebody has already made a package for migrating stored procedures from other DBMSs to MySQL.

Tips for migration ...

Change assignment statements like “a:=b” to SET a=b”.

Change RETURN statements inside procedures with “LEAVE label_at_start” where label_at_start is the first token in the stored procedure. For example:
[Oracle procedure]

```
CREATE PROCEDURE ... RETURN; ...
```

[MySQL procedure]

```
CREATE PROCEDURE () label_at_start: BEGIN ... LEAVE
label_at_start; END
```

This is only necessary with procedures, since MySQL supports RETURN in a function.

Side-By-Side

Oracle

```
CREATE PROCEDURE
sp_name
AS
variable1 INTEGER


variable1 := 55
END
```

MySQL

```
CREATE PROCEDURE
sp_name


BEGIN
DECLARE variable1 INTEGER;
SET variable1 = 55;
END
```

Summary:

- SQL Server parameter names begin with '@'
  MySQL parameter names are regular identifiers

- SQL Server can say "DECLARE v1 data-type, v2 data_type"
  MySQL can only say “DECLARE v1 data_type; DECLARE v2 data_type”

- SQL Server doesn't have BEGIN ... END for the procedure body
  MySQL requires BEGIN ... END to surround multiple statements

- SQL Server doesn't require that statements end with ';'
  MySQL requires that statements end with ';', except last statement

- SQL Server has "SET NOCOUNT" and "IF @@ROWCOUNT"
  MySQL does not have those things, but FOUND_ROWS() works

- SQL Server has “WHILE ... BEGIN” statements
  MySQL has “WHILE ... DO” statements

- SQL Server allows “SELECT” for assignment
  MySQL uses SET” for assignment, exclusively

- "WHILE ... BEGIN" not "WHILE ... DO"

- SELECT instead of SET (optionally)

- SQL Server allows accessing the same table table in and out of functions
  MySQL does not allow allow accessing the same table in and out of functions

With Microsoft SQL Server, the list of differences is considerably longer. Converting Microsoft or Sybase programs to MySQL is going to be tedious. But the differences are mostly syntax, it requires no special intelligence to make the changes.

Some migration tips ...

If SQL Server has a variable named @xxx, you have to change it because in MySQL a variable that starts with @ is not a stored-procedure variable, it's a session variable. Don't just change it to xxx, because then it might become ambiguous – for example, there might be a column named xxx in one of the database tables. Use a conventional prefix of your own choice, for example change @xxx to var_xxx.

Side-By-Side

SQL Server

```
CREATE PROCEDURE
sp_procedure1
AS


DECLARE @x VARCHAR(100)
EXECUTE  sp_procedure2 @x

DECLARE c CURSOR FOR
SELECT * FROM t


END
```

MySQL

```
CREATE PROCEDURE
sp_procedure1

()
BEGIN
DECLARE v__x VARCHAR(100);
CALL sp_procedure2(v__x);

DECLARE c CURSOR FOR
SELECT * FROM t;


END
```

- DB2 allows PATH statement
  MySQL does not allow PATH statements

- DB2 allows SIGNAL statement
  MySQL does not allow SIGNAL statement

- DB2 allows overloading of routine names so it has SPECIFIC NAME clause
  MySQL does not allow overloading of routine names

- DB2 has “label_x: ... GOTO label_x” syntax
  MySQL does not have “label label_x; ... GOTO label_x” syntax (it's gone)

- DB2 allows accessing the same table in and out of a function
  MySQL does not allow accessing the same table in and out of a function

DB2 stored procedures look almost exactly the same as MySQL stored procedures. The only things that you have to watch for are a few statements that we have not implemented yet, and also the fact that DB2 allows overloading. That is, with DB2 you can have two routines with the same name, and DB2 distinguishes between them by looking at their parameters, or by using a specific name clause. DB2 stored procedures are upward compatible with MySQL stored procedures.

Migration tips: if you're going to migrate, you need hardly any tips at all. For MySQL's lack of a SIGNAL statement, we elsewhere discuss a temporary workaround. For DB2's GOTO, we used to have a similar MySQL GOTO to make the migration easier, but we decided it's inappropriate. For PATH (which determines where the DBMS looks for the routine in the database catalog), just avoid the problem by adding a prefix to the routine name. For the difficulty with accessing tables within functions, you can often write procedures instead and let the OUT parameter of the procedure substitute for a function result.

Side-By-Side

DB2

```
CREATE PROCEDURE
sp_name
(parameter1 INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v INTEGER;
  IF parameter1 >=5 THEN
    CALL p26();
    SET v = 2;
  END IF;
  INSERT INTO t VALUES (v);
END
@
```

MySQL

```
CREATE PROCEDURE
sp_name
(parameter1 INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v INTEGER;
  IF parameter1 >=5 THEN
    CALL p26();
    SET v = 2;
  END IF;
INSERT INTO t VALUES (v);
END
//
```

Standard Comparison 80

Summary:

Standard SQL requires:
[same things as in DB2]

MySQL's goal is to support these two standard SQL features:
Feature P001 "Stored Modules"
Feature P002 "Computational completeness"

The reason that DB2 is so much like MySQL is that both DBMSs support stored procedures as defined by the SQL Standard. So MySQL's differences with DB2 are pretty well the same as our departures from ANSI/ISO standard syntax. But we are much, much closer to the standard than Oracle or SQL Server.

MySQL's goal is to support these two features of standard SQL: Feature P001 "Stored Modules", Feature P002 "Computational completeness".

```
CREATE PROCEDURE p ()
BEGIN
  /* Use comments! */
  UPDATE t SET s1 = 5;
  CREATE TRIGGER t2_ai ...
END;//
```

In my examples I've used a certain style to write my stored procedures.

Keywords are in upper case.

For naming conventions, I believe it's better in a manual to say that a table is “t something” and a column is “s something”. But in real life, with my DBA hat on, I try to go with the guidelines hinted at in the article “SQL Naming Conventions”: http://dbazine.com/gulutzan5.shtml

Comments always look like C comments. Don't use #comment or –comment. Incidentally comments are not preserved in the metadata (the mysql client strips them), but you can insert remarks by adding a string in a non-executable statement, such as DECLARE.

I indent after the word BEGIN.

I indent back before the word END. Actually I dislike this detail. I'd prefer to say:

```
CREATE PROCEDURE p ()
BEGIN
  /* Use comments! */
  UPDATE t SET s1 = 5;
  CREATE TRIGGER t2_ai ...
__END;//                                                    <--
```

That is, the indenting should end after END, not before it. But I try to curb my abnormal impulses.

I don't suggest that you follow my style, but I do suggest that you choose a style of your own, and stick to it when writing your own stored procedures. It makes the code so much easier to follow.

This is a procedure that lists all the table names in a single string. Compare the MySQL built-in function group_concat().

```
CREATE PROCEDURE tables_concat
(OUT parameter1 VARCHAR(1000))
BEGIN
  DECLARE variable2 CHAR(100);
  DECLARE c CURSOR FOR
  SELECT table_name FROM information_schema.tables;
  DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;    /* 1 */
  SET sql_mode='ansi';                             /* 2 */
  SET parameter1 = '';
  OPEN c;
  LOOP
    FETCH c INTO variable2;                        /* 3 */
    SET parameter1 = parameter1 || variable2 || '.';
    END LOOP;
  CLOSE c;
  END;
```

/* 1 */: “BEGIN END” is a statement that does nothing at all, like the NULL statement in other DBMS languages.
/* 2 */: Changing the sql_mode setting within the procedure so that concatenation with || will work. The setting will still be = 'ansi' after exit from the stored procedure.
/* 3 */: Another way to get out of a LOOP: by declaring an EXIT handler that will eventually catch the failure of a FETCH when there are no more rows.

Here's what happens if we call this procedure later:

```
mysql> CALL tables_concat(@x);
Query OK, 0 rows affected (0.05 sec)

mysql> SELECT @x;
+-------------------------------------------------+
| SCHEMATA.TABLES.COLUMNS.CHARACTER_SETS.COLLATIONS.C
OLLATION_CHARACTER_SET_APPLICABILITY.ROUTINES.STATIST
ICS.VIEWS.USER_PRIVILEGES.SCHEMA_PRIVILEGES.TABLE_PRI
VILEGES.COLUMN_PRIVILEGES.TABLE_CONSTRAINTS.KEY_COLUM
N_USAGE.TABLE_NAMES.columns_priv.db.fn.func.help_cate
gory.help_keyword.help_relation.help_topic.host.proc.
tables_priv.time_zone.time_zone_leap_second.time_zone
1 row in set (0.00 sec)
```

Function Example: rno()

The objective is to get a “row number within result set” integer, similar to the ROWNUM() in other DBMSs.

We need a user variable to store the value after each invocation of rno(). We'll call the variable @rno.

```
CREATE FUNCTION rno ()
RETURNS INT
BEGIN
  SET @rno = @rno + 1;
  RETURN @rno;
  END;
```

We get the row number by just using rno() in a SELECT. Here's what happens if we call the procedure later:

```
mysql> SET @rno = 0;//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT rno(),s1,s2 FROM t;//
+-------+------+------+
| rno() | s1   | s2   |
+-------+------+------+
|     1 |    1 | a    |
|     2 |    2 | b    |
|     3 |    3 | c    |
|     4 |    4 | d    |
|     5 |    5 | e    |
+-------+------+------+
5 rows in set (0.00 sec)
```

Here's a trick that resets @rno to zero within the SELECT. It depends on the fact that WHERE is evaluated first. It is slow and depends on behaviour that MySQL won't guarantee will work in future versions.

```
CREATE FUNCTION rno_reset ()
RETURNS INTEGER
BEGIN
  SET @rno = 0;
  RETURN 1;
  END;
SELECT rno(),s1,s2 FROM t WHERE rno_reset()=1;//
```

Function Example: running_total() 84

This accumulator function builds on what rno() does. The only novelty is that we have to pass a value in the parameter in order to add to the running total with each invocation.

```
CREATE FUNCTION running_total (IN adder INT)
RETURNS INT
 BEGIN
 SET @running_total = @running_total + adder;
 RETURN @running_total;
 END;
```

Here's what happens if we use the running_total() function later:

```
mysql> SET @running_total = 0;//
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT s1,running_total(s1),s2 FROM t ORDER BY s1;//
+------+-------------------+------+
| s1   | running_total(s1) | s2   |
+------+-------------------+------+
|    1 |                 1 | a    |
|    2 |                 3 | b    |
|    3 |                 6 | c    |
|    4 |                10 | d    |
|    5 |                15 | e    |
+------+-------------------+------+
5 rows in set (0.01 sec)
```

The running_total() function works because it's called after ORDER BY is done. This is not standard or portable.

Procedure Example: MyISAM “Foreign Key” insertion 85

The MyISAM storage engine doesn't support foreign keys, but you can put the logic for foreign key checking in a stored procedure.

```
CREATE PROCEDURE fk_insert (p_fk INT, p_animal VARCHAR(10))
BEGIN
  DECLARE v INT;
  BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, NOT FOUND
    SET v = 0;
    IF p_fk IS NOT NULL THEN
      SELECT 1 INTO v FROM tpk WHERE cpk = p_fk LIMIT 1;
      INSERT INTO tfk VALUES (p_fk, p_animal);
    ELSE
      SET v = 1;
    END IF;
  END;
  IF v <> 1 THEN
    DROP TABLE `The insertion failed`;
  END IF;
END;
```

Notes: Either an SQLEXCEPTION or a NOT FOUND condition will cause v to be zero, but otherwise v will be one, because the SELECT will put one into it and the EXIT HANDLER won't set it back to zero. And here's what happens when we call ...

```
mysql> CREATE TABLE tpk (cpk INT PRIMARY KEY);//
Query OK, 0 rows affected (0.01 sec)
mysql> CREATE TABLE tfk (cfk INT, canimal VARCHAR(10));//
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tpk VALUES (1),(7),(10);//
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> CALL fk_insert(1,'wombat');//
Query OK, 1 row affected (0.02 sec)

mysql> CALL fk_insert(NULL,'wallaby');//
Query OK, 0 rows affected (0.00 sec)

mysql> CALL fk_insert(17,'wendigo');//
ERROR 1051 (42S02): Unknown table 'The insertion failed'
```

Suppose that procedure1 calls procedure2 calls procedure3. An SQL error in procedure3 “propagates”, that is, unless some handler catches it, it keeps going up the line, causing procedure2 to fail, which causes procedure1 to fail, and ultimately the caller (the mysql client in this case) sees the error. Sometimes this property is desirable so standard SQL has a SIGNAL statement to force errors to happen, and other DBMSs have something similar (RAISERROR). MySQL doesn't support SIGNAL yet. Until it does, force errors by doing something harmless but erroneous.

```
CREATE PROCEDURE procedure1 ()
BEGIN
  CALL procedure2();
  SET @x = 1;
END;

CREATE PROCEDURE procedure2 ()
BEGIN
  CALL procedure3();
  SET @x = 2;
END;

CREATE PROCEDURE procedure3 ()
  BEGIN
  DROP TABLE error.`error #7815`;
  SET @x = 3;
END;
```

Here's what happens if we call procedure1 later:

```
mysql> CALL procedure1()//
ERROR 1051 (42S02): Unknown table 'error #7815'
```

And @x is unchanged, because none of the “SET @x = ...” statements got executed. Using DROP allows us to get a bit of user-generated diagnostic information into the error message. However, we have to depend on the assumption that there is no database named `error`.

Procedure Example: Library 87

These are the specifications for a library application.

We want anybody to be able to call the procedures if they have the right privileges, which we'll set later with
GRANT ALL ON database-name.* TO user-name;
But the other users should only be able to access using the procedures. That's no problem. It means we want SQL SECURITY DEFINER characteristic. That's the default but we'll specify it anyway, as documentation.

We want a procedure to add books. There should be a test that the book's id isn't positive and the book's title isn't blank. This is a substitute for a CHECK constraint, which MySQL doesn't support.

```
CREATE PROCEDURE add_book
(p_book_id INT, p_book_title VARCHAR(100))
SQL SECURITY DEFINER
BEGIN
 IF p_book_id < 0 OR p_book_title='' THEN
   SELECT 'Warning: Bad parameters';
   END IF;
 INSERT INTO books VALUES (p_book_id, p_book_title);
 END;
```

We want a procedure to add patrons. There should be a test to warn that there are already more than 2 patrons. It's possible to test such a thing with a subquery, e.g.
IF (SELECT COUNT(*) FROM table-name) > 2) THEN ... END IF;
But at the time I originally wrote, there were bugs with subqueries. So instead we'll use "SELECT COUNT(*) INTO variable-name".

```
CREATE PROCEDURE add_patron
(p_patron_id INT, p_patron_name VARCHAR(100))
SQL SECURITY DEFINER
BEGIN
  DECLARE v INT DEFAULT 0;
  SELECT COUNT(*) FROM patrons INTO v;
  IF v > 2 THEN
    SELECT 'warning: already there are ',v,'patrons!';
  END IF;
  INSERT INTO patrons VALUES (p_patron_id,p_patron_name);
END;
```

Procedure Example: Library (continued) 88

We want a procedure to check out books. During the transaction we'd like to display the patrons who already have this book, and the books that this patron already has. We'll find that information with cursor fetching, and we'll use two different ways to test whether there's no more data to fetch: first by checking whether the variable retains its original NULL value after a fetch, second by catching fetch failure with a NOT FOUND handler.

It's cleaner looking if the two cursors are in separate BEGIN/END blocks. But we'll declare variables in the main BEGIN/END block, so that their scope is the whole procedure.

```
CREATE PROCEDURE checkout (p_patron_id INT, p_book_id INT)
SQL SECURITY DEFINER
BEGIN
  DECLARE v_patron_id, v_book_id INT;
  DECLARE no_more BOOLEAN default FALSE;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more=TRUE;
  BEGIN
    DECLARE c1 CURSOR FOR SELECT patron_id
    FROM transactions WHERE book_id = p_book_id;
    OPEN c1;
    SET v_patron_id=NULL;
    FETCH c1 INTO v_patron_id;
    IF v_patron_id IS NOT NULL THEN
      SELECT 'Book is already out to this patron:',
      v_patron_id;
      END IF;
    CLOSE c1;
    END;
  BEGIN
    DECLARE c2 CURSOR FOR SELECT book_id
    FROM transactions WHERE patron_id = p_patron_id;
    OPEN c2;
    book_loop: LOOP
      FETCH c2 INTO v_book_id;
      IF no_more THEN
        LEAVE book_loop;
         END IF;
      SELECT 'Patron already has this book:', v_book_id;
      END LOOP;
    END;
  INSERT INTO transactions VALUES (p_patron_id, p_book_id);
  END;
```

Procedure Example: Hierarchy (I) 89

The hierarchy() procedure does part of what CONNECT BY would do with another DBMS. We have a Persons table where descendants are linked to ancestors via a column person_id. We pass parameter start_with = person to begin the list with. We display the ancestors and descendants in order.

The procedure (well actually two procedures) listing is on the next two pages. I expect that since you've gotten this far, you'll be able to follow the perambulations if you read carefully. Still, I'd better prepare you with a few explanatory notes first.

The hierarchy() procedure receives a person's name as an input parameter. It performs some initializations – most importantly, it makes a temporary table where each result row will be stored as it's found. Then it calls another procedure, hierarchy2(). Now, hierarchy2() is recursive, which means that it calls itself. This wouldn't be necessary if every father had only zero or one sons, but that's not how things are, so as we reach each branch, we have to keep going down the tree with recursion, then return to the branching point and go down the next branch.

I decided to make the exception handler set a flag (error) which gets checked after any SQL statement. If a statement fails, I use “SELECT 'string'” to output a diagnostic, and then “LEAVE proc”. Since proc is a label at the start of the procedure, that effectively means “leave the procedure”.

I made some of the compound statements nested (BEGIN END within BEGIN END) so that I'd be able to do some DECLAREs of variables and handlers just for the specific statements that they are relevant to. Remember that any DECLAREs in the outer compound statement are still in effect for the inner compound statement unless they're overridden, and remember that when an inner compound statement ends, its declarations go out of scope (cease to be noticeable).

The CREATE PROCEDURE for hierarchy() is on the next page. On the page after that is the CREATE PROCEDURE for hierarchy2(). On the page after that is a listing that shows what happens when I call hierarchy() ... if all goes well!

```
CREATE PROCEDURE hierarchy (start_with CHAR(10))
proc:
BEGIN
  DECLARE temporary_table_exists BOOLEAN;
  BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN END;
    DROP TABLE IF EXISTS Temporary_Table;
  END;
  BEGIN
    DECLARE v_person_id, v_father_id INT;
    DECLARE v_person_name CHAR(20);
    DECLARE done, error BOOLEAN DEFAULT FALSE;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET error = TRUE;
    CREATE TEMPORARY TABLE Temporary_Table
    (person_id INT, person_name CHAR(20), father_id INT);
    IF error THEN
    SELECT 'CREATE TEMPORARY failed'; LEAVE proc; END IF;
    SET temporary_table_exists=TRUE;
    SELECT person_id, person_name
    INTO v_person_id, v_person_name FROM Persons
    WHERE person_name = start_with limit 1;
    IF error THEN
      SELECT 'First SELECT failed'; LEAVE proc; END IF;
    IF v_person_id IS NOT NULL THEN
      INSERT INTO Temporary_Table VALUES
      (v_person_id, v_person_name, v_father_id);
      IF error THEN
      SELECT 'First INSERT failed'; LEAVE proc; END IF;
      CALL hierarchy2(v_person_id);
      IF error THEN
      SELECT 'First CALL hierarchy2() failed'; END IF;
      END IF;
    SELECT person_id, person_name, father_id
    FROM Temporary_Table;
    IF error THEN
    SELECT 'Temporary SELECT failed'; LEAVE proc; END IF;
  END;
  IF temporary_table_exists THEN
    DROP TEMPORARY TABLE Temporary_Table;
    END IF;
END;//
```

```
CREATE PROCEDURE hierarchy2 (start_with INT)
proc:
BEGIN
  DECLARE v_person_id INT;
  DECLARE v_father_id INT;
  DECLARE v_person_name CHAR(20);
  DECLARE done, error BOOLEAN DEFAULT FALSE;
  DECLARE c CURSOR FOR
  SELECT person_id, person_name, father_id
  FROM Persons WHERE father_id = start_with;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  SET error = TRUE;
  OPEN c;
  IF error THEN
  SELECT 'OPEN failed'; LEAVE proc; END IF;
  REPEAT
    SET v_person_id=NULL;
    FETCH c INTO v_person_id, v_person_name, v_father_id;
    IF error THEN
    SELECT 'FETCH failed'; LEAVE proc; END IF;
    IF done=FALSE THEN
      INSERT INTO Temporary_Table VALUES
      (v_person_id, v_person_name, v_father_id);
      IF error THEN
      SELECT 'INSERT in hierarchy2() failed'; END IF;
    CALL hierarchy2(v_person_id);
    IF error THEN
    SELECT 'Recursive CALL hierarchy2() failed'; END IF;
    END IF;
    UNTIL done = TRUE
  END REPEAT;
  CLOSE c;
  IF error THEN
  SELECT 'CLOSE failed'; END IF;
END;
//
```

Procedure Example: Hierarchy (continued) 92

What happens when I make some sample data and call hierarchy() ...

```
mysql> CREATE TABLE Persons (person_id INT,
    ->  person_name CHAR(20), father_id INT);//
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO Persons VALUES
    -> (1,'Grandpa',NULL);//
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Persons VALUES
    ->   (2,'Pa-1',1), (3,'Pa-2',1);//
Query OK, 2 rows affected (0.00 sec)

mysql> INSERT INTO Persons VALUES
    ->   (4,'Grandson-1',2),(5,'Grandson-2',2);//
Query OK, 2 rows affected (0.00 sec)

mysql> SET @@max_sp_recursion_depth = 100//
Query OK, 0 rows affected (0.00 sec)

mysql> CALL hierarchy('Grandpa')//
+-----------+-------------+-----------+
| person_id | person_name | father_id |
+-----------+-------------+-----------+
|         1 | Grandpa     |      NULL |
|         2 | Pa-1        |         1 |
|         4 | Grandson-1  |         2 |
|         5 | Grandson-2  |         2 |
|         3 | Pa-2        |         1 |
+-----------+-------------+-----------+
5 rows in set (0.01 sec)

Query OK, 0 rows affected, 1 warning (0.06 sec)
```

The search goes all the way down to grandsons, then starts again at sons. This is a "depth first" display. A "width first" display would have been ordered by level, that is, first the grandpa, then all sons, then all grandsons.

The hierarchy() procedure is pretty complex, eh? It might have bugs that I haven't thought of. It will probably fail if recursion is too deep, e.g. if there are 100 generations. It will certainly loop if there is a cycle, e.g. if somebody is his own grandpa. But despite all those caveats, I think I've proven that CONNECT BY is possible in MySQL via a recursive stored procedure.

Tips when writing long routines 93

MySQL has no stored-procedure debugger, and MySQL error messages can be unhelpful. Here's what I do when I'm creating a 20-line procedure and I see an error message.

Start an editor, and copy the statement to the editor. I find it's easier to change the statement with the editor and then cut-and-paste on the mysql client.

If it's a syntax error, then remove one statement at a time until the syntax error goes away. This is usually faster than eyeballing the whole CREATE PROCEDURE statement, which of course will often appear to be perfectly all right.

If it's a runtime error, then add “SELECT n;” after every executable statement inside the body, where n is 0, 1, 2, etc. Then call the procedure again. This is a diagnostic for tracing the flow of control.

If it's a data error, then skip over it by adding “DECLARE CONTINUE HANDLER FOR ... BEGIN END;”. Often with a test database the data just isn't there, so while debugging I have these statements that let me carry on.

There are still some bugs in our stored procedure code. They're getting rarer. They're getting obscurer. I counted the relevant bugs on December 14 2004 and again on May 31 2006.

```
Bug Count On December 14 2004: *
Category                              Count
Causes Operating System To Reboot      1
Crashes or Hangs                      23
Returns 'Packets out of order'         6
Fails                                 31
                                      --
                                      61

Bug Count On May 31 2006:
Category                              Count
Causes Operating System To Reboot      0
Crashes or Hangs                       2
Returns 'Packets out of order'         0
Fails                                 35
                                      --
                                      37

* Searched differently due to changes in bugs.mysql.com
```

You can measure our progress by looking at the MySQL bugs web page and searching for stored procedure or trigger. Here are the steps to take:

Go to the web page of our bugs database, http://bugs.mysql.com

Click “Advanced Search”

In the dropdown box that by default says “10 bugs”, select “All”.

In the 'Category” list box select “Stored Procedures”.

Click “Search”.

So, if you do a search now and find more bugs than I found on May 31 2006, will that be a bad sign? Well, yes, but perhaps it will happen because we have a bigger pool of potential bug reporters, perhaps including you.

Summary:
SIGNAL
PATH
Accessing More Tables in Functions or Triggers
LOAD DATA
BEGIN ATOMIC ... END
Encrypted storage
Timeout
Debugger
External Languages
DROP ... CASCADE|RESTRICT
FOR statements which end when a cursor ends

Here's a quick look at some additions we're looking to add.

The most important feature requests are for functions that help us to catch up with the power of the ANSI/ISO standard and of DB2. We've also talked about the need for a way to time out if a function goes into a loop and runs too long.

(I) Web Pages Of MySQL
(II) Downloads From MySQL
(III) Books

For my last act I'll show you where you can go for further information.

Web Pages Of MySQL

MySQL reference manual chapter "Stored Procedures":
http://www.mysql.com/doc/en/Stored_Procedures.html

Newsletter article "Stored Procedures in MySQL":
http://www.mysql.com/newsletter/2004-01/a0000000297.html

Here are some resources you'll be able to access from the Web.

MySQL Downloads

```
> cd ~/mysql-5.1-new/mysql-test/t

> dir sp*
   39409 2006-04-13 05:19 sp-error.test
   16732 2006-05-11 07:26 sp-security.test
    3670 2006-04-13 08:51 sp-threads.test
  125960 2006-05-19 07:18 sp.test

> cd ~/mysql-5.1-new/Docs

> dir sp*
   41909 2006-04-13 11:26 sp-imp-spec.txt
```

For more details, download the MySQL 5.0 source code. I don't just mean the code itself. There are test scripts in the mysql-test directory, and documentation or news files in the Docs directory.

But I also mean the code itself. If you're looking at it for the first time, I recommend starting with a look at the online “Internals” documentation, here:
http://dev.mysql.com/doc/internals/en/index.html

Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM
Author: Jim Melton
Publisher: Morgan Kaufmann
Only available second-hand.

One excellent reference is a book that was written about standard SQL stored procedures -- Jim Melton's "Understanding SQL's stored procedures, a complete guide to SQL/PSM". This is a book about the SQL standard, and MySQL tries to follow the standard, so it's useful in places.

That's a nice thing, to have a book available before the product is ready.

MySQL Stored Procedure Programming
Author: Guy Harrison, with Steven Feuerstein
Publisher: O'Reilly

I'll just quote what I said on the book's front page:
“This is the first book I've seen that really concentrates on MySQL's stored procedures. I found tips here that I'd never seen before.”

The End 98

This is the end of the book. I don't bother with a review or an index, since I'm sure you've had no trouble memorizing it all.

If you enjoyed this book, you should look for more in the “MySQL 5.0 New Features” series. You can find a list on our dev.mysql.com page, under the heading “MySQL Key Features”.

Thank you very much for your attention. If you have any comments about this book, please send them to one of the MySQL forums:
http://forums.mysql.com/