SQL Stored Programs

Procedures, functions, triggers

What Is a Stored Program?

- A database *stored program* is a computer program that is stored within, and executes within, the database server.
- The source code and any compiled version of the stored program are almost always held within the database server's system tables as well.
- When the program is executed, it is executed within the memory address of a database server process or thread.

Types of stored programs

- *Stored procedures*
  - Most common type of stored program.
  - Generic program unit that is executed on request and that can accept multiple input and output parameters.
- *Stored functions*
  - Similar to stored procedures, but their execution results in the return of a single value.
  - Can be used within a standard SQL statement.
- *Triggers*
  - Stored programs that are activated in response to, or are *triggered* by, an activity within the database.
  - Typically invoked in response to a DML operation (INSERT, UPDATE, DELETE) against a database table.
  - Can be used for data validation or for the automation of denormalization.

Why Use Stored Programs?

- The use of stored programs can lead to a more secure database.
- Stored programs offer a mechanism to abstract data access routines, which can improve the maintainability of your code as underlying data structures evolve.
- Stored programs can reduce network traffic, because the program can work on the data from within the server, rather than having to transfer the data across the network.
- Stored programs can be used to implement common routines accessible from multiple applications—possibly using otherwise incompatible frameworks— executed either within or from outside the database server.
- Database-centric logic can be isolated in stored programs and implemented by programmers with more specialized, database experience.
- The use of stored programs can, under some circumstances, improve the portability of your application.

Stored programs in MySQL

- Implemented as a subset of the ANSI SQL:2003 SQL/PSM (Persistent Stored Module) specification.
- Available since version 5.
- Block-structured language (like Pascal) includes familiar commands for manipulating variables, implementing conditional execution, performing iterative processing, and handling errors.

Creating the Procedure

- Create a stored program with the CREATE PROCEDURE
  - CREATE PROCEDURE *procedure_name* ([*parameter[,...]*])
    [LANGUAGE SQL]
    [[NOT] DETERMINISTIC]
    [{CONTAINS SQL|MODIFIES SQL DATA|READS SQL DATA|NO SQL}] [SQL SECURITY {DEFINER|INVOKER}]
    [COMMENT comment_string]
    procedure_statements

    BEGIN

    …

    END

- LANGUAGE SQL
  - Indicates that the stored procedure uses the SQL:PSM standard stored procedure language. Since MySQL currently supports only those stored procedures written in this language, specifying this keyword is unnecessary at present.
- SQL SECURITY {DEFINER|INVOKER}
  - Determines whether the stored procedure should execute using the permissions of the user who created the stored procedure (DEFINER) or the permissions of the user who is currently executing the stored procedure (INVOKER). The default is DEFINER.

- [NOT] DETERMINISTIC
  - Indicates whether the stored procedure will always return the same results if the same inputs are provided. By default, MySQL will assume that a stored procedure (or function) is NOT DETERMINISTIC.
  - Only time this keyword is critical is when you are creating a stored function
- NO SQL|CONTAINS SQL|READS SQL DATA|MODIFIES SQL DATA
  - Indicates the type of access to database data that the stored procedure will perform. If a program reads data from the database, you may specify the READS SQL DATA keyword. If the program modifies data in the database, you could specify MODIFIES SQL DATA. If the procedure or function performs no database accesses, you may specify NO SQL.
- *COMMENT comment_string*
  - Specifies a comment that is stored in the database along with the procedure definition.

Creating the Procedure

- Delete previously declared procedure to avoid errors.
  - DROP PROCEDURE IF EXISTS *ProcName*
  - DROP {PROCEDURE|FUNCTION|TRIGGER} [IF EXISTS] *program_name*
- Change delimiter before declaring procedure
  - DELIMITER $$ (can be anything)
- Use CALL to invoke procedure

```
DELIMITER $$

DROP PROCEDURE IF EXISTS TestProcedure$$
CREATE PROCEDURE TestProcedure()
BEGIN
    SELECT "HelloWorld";
END$$

DELIMITER ;


CALL TestProcedure();
```

Variables

- Local variables can be declared within stored procedures using the DECLARE statement.
  - DECLARE *variable_name* [,*variable_name...*] *datatype* [DEFAULT *value*];
- Variable names follow the same naming rules as MySQL table column names and can be of any MySQL data type.
- Give variables an initial value with the DEFAULT clause.
  - NULL value assigned if no DEFAULT clause
- Assign new values using the SET command.
  - SET *variable_name* = *expression* [,*variable_name* = *expression ...*]

String Data Types

- MySQL supports two basic string data types: CHAR and VARCHAR
  - Functionally similar, only difference is maximum size
- ENUM
  - Used to store one of a set of permissible values.
  - These values can be accessed as their string value or as their indexed position in the set of possibilities
- SET
  - Similar to the ENUM type, except that multiple values from the list of allowable values can occur in the variables

```
CREATE PROCEDURE sp_enums(in_option ENUM('Yes','No','Maybe'))
BEGIN
  DECLARE position INTEGER;
  SET position=in_option;
  SELECT in_option,position;
END
--------------

Query OK, 0 rows affected (0.01 sec)

--------------
CALL sp_enums('Maybe')
--------------

+-----------+----------+
| in_option | position |
+-----------+----------+
| Maybe     |        3 |
+-----------+----------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

--------------
CALL sp_enums(2)
--------------

+-----------+----------+
| in_option | position |
+-----------+----------+
| No        |        2 |
+-----------+----------+
1 row in set (0.00 sec)
```

```
CREATE PROCEDURE sp_set(in_option SET('Yes','No','Maybe'))
BEGIN

  SELECT in_option;
END
-------------

Query OK, 0 rows affected (0.00 sec)

-------------
CALL sp_set('Yes')
-------------

+-----------+
| in_option |
+-----------+
| Yes       |
+-----------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)

-------------
CALL sp_set('Yes,No,Maybe')
-------------

+--------------+
| in_option    |
+--------------+
| Yes,No,Maybe |
+--------------+
1 row in set (0.00 sec)
```

Table 3-1. Commonly used MySQL data types

Data type	Explanation	Examples of corresponding values
INT, INTEGER	A 32-bit integer (whole number). Values can be from approximately -2.1 billion to +2.1 billion. If unsigned, the value can reach about 4.2 billion, but negative numbers are not allowed.	123,345 -2,000,000,000
BIGINT	A 64-bit integer (whole number). Values can be from approximately -9 million trillion to +9 million trillion or from 0 to 18 million trillion if unsigned.	9,000,000,000,000,000,000 -9,000,000,000,000,000,000
FLOAT	A 32-bit floating-point number. Values can range from about -1.7e38 to 1.7e38 for signed numbers or 0 to 3.4e38 if unsigned.	0.00000000000002 17897.890790 -345.8908770 1.7e21
DOUBLE	A 64-bit floating-point number. The value range is close to infinite (1.7e308).	1.765e203 -1.765e100
DECIMAL(*precision*,*scale*) NUMERIC(*precision*,*scale*)	A fixed-point number. Storage depends on the precision, as do the possible numbers that can be stored. NUMERICs are typically used where the number of decimals is important, such as for currency.	78979.00 -87.50 9.95
DATE	A calendar date, with no specification of time.	'1999-12-31'

Table 3-1. Commonly used MySQL data types (continued)

Data type	Explanation	Examples of corresponding values
DATETIME	A date and time, with resolution to a particular second.	'1999-12-31 23:59:59'
CHAR(*length*)	A fixed-length character string. The value will be right-padded up to the length specified. A maximum of 255 bytes can be specified for the length.	'hello world '
VARCHAR(*length*)	A variable-length string up to 64K in length.	'Hello world'
BLOB, TEXT	Up to 64K of data, binary in the case of BLOB, or text in the case of TEXT.	Almost anything imaginable
LONGBLOB, LONGTEXT	Longer versions of the BLOB and TEXT types, capable of storing up to 4GB of data.	Almost anything imaginable, but a lot more than you would have imagined for BLOB or TEXT

Example 3-1. Examples of variable declarations

```
DECLARE l_int1      INT DEFAULT -2000000;
DECLARE l_int2      INT UNSIGNED DEFAULT 4000000;
DECLARE l_bigint1   BIGINT DEFAULT 4000000000000000;
DECLARE l_float     FLOAT DEFAULT 1.8e8;
DECLARE l_double    DOUBLE DEFAULT 2e45;
DECLARE l_numeric   NUMERIC(8,2) DEFAULT 9.95;

DECLARE l_date      DATE DEFAULT '1999-12-31';
DECLARE l_datetime  DATETIME DEFAULT '1999-12-31 23:59:59';

DECLARE l_char      CHAR(255) DEFAULT 'This will be padded to 255 chars';
DECLARE l_varchar   VARCHAR(255) DEFAULT 'This will not be padded';

DECLARE l_text      TEXT DEFAULT 'This is a really long string.  In stored programs
                    we can use text columns fairly freely, but in tables there are some
                    limitations regarding indexing and use in various expressions.';
```

Literals

- A *literal* is a data value hardcoded into a program.
- Commonly used in variable assignment statements or comparisons, as arguments to procedures or functions, or within SQL statements.
- All variables in MySQL stored programs are *scalars*, there are no equivalents to arrays, records, or structures.
- *Numeric literals*
  - Represents a number and can be defined as a raw number, a hexadecimal value, or in scientific notation.
  - Hexadecimal values are represented by prefixing them with '0x'
  - Scientific notation is represented using the letter 'e'. 2.4e -> $2.4x10^4$

Literals

- Date literals
  - A date literal is a string in the format ‘YYYY-MM-DD’ or —for the DATETIME data type—in the format ‘YYYY-MM-DD HH24:MI:SS’
- *String literals*
  - Any string value surrounded by quotes.
  - If single quotes need to be included within the literal itself delimited by single quotes, they can be represented by two single quotes or prefixed with a back- slash (\').
  - Use escape sequences for special characters (\t for a tab, \n for a new line, \\ for a back- slash, etc.).

```
CREATE PROCEDURE variable_demo()
BEGIN
    DECLARE my_integer     INT;            /* 32-bit integer */
    DECLARE my_big_integer BIGINT;         /* 64-bit integer */
    DECLARE my_currency    NUMERIC(8,2);   /* Number with 2 decimals*/
    DECLARE my_pi          FLOAT           /* Floating point number*/
            DEFAULT 3.1415926;             /* initialized as PI */
    DECLARE my_text        TEXT;           /* huge  text */
    DECLARE my_dob         DATE
            DEFAULT '1960-06-21';          /* My Birthday */
    DECLARE my_varchar     VARCHAR(30)
            DEFAULT 'Hello World!';        /* 30 bytes of text*/

    SET my_integer=20;
    SET my_big_integer=POWER(my_integer,3);

END;$$
```

Parameters

- Place parameters within parentheses that are located immediately after the name of the stored procedure.
  - CREATE PROCEDURE|FUNCTION([[IN|OUT|INOUT] *parameter_name data_type* ...])
- Each parameter has a name, a data type, and, optionally, a mode.
- Valid modes are IN (read-only), INOUT (read-write), and OUT (write-only).

Parameters

- IN: This mode is the default. It indicates that the parameter can be passed into the stored program but that any modifications are not returned to the calling program.

- OUT: This mode means that the stored program can assign a value to the parameter, and that value will be passed back to the calling program. The calling program must supply a variable to receive the output of the OUT parameter, but the stored program itself has no access to whatever might be initially stored in that variable.

- INOUT: This mode means that the stored program can read the parameter and that the calling program can see any modifications that the stored program may make to that parameter.

```
DROP PROCEDURE IF EXISTS testParameters$$
CREATE PROCEDURE testParameters(inputNo INT, OUT outNo FLOAT)
BEGIN
    SET outNo = sqrt(inputNo);
END$$

CALL testParameters(16, @outValue)$$
SELECT @outValue$$
```

Example 3-3. Example of an IN parameter

```
mysql> CREATE PROCEDURE sp_demo_in_parameter(IN p_in INT)
BEGIN
    /* We can see the value of the IN parameter */
    SELECT p_in;
    /* We can modify it*/
    SET p_in=2;
    /* show that the modification took effect */
    select p_in;
END;

/* This output shows that the changes made within the stored program cannot be accessed
from the calling program (in this case, the mysql client):*/
```

```
mysql> SET @p_in=1

Query OK, 0 rows affected (0.00 sec)

mysql> CALL sp_demo_in_parameter(@p_in)

+------+---------------------------------------+
| p_in | We can see the value of the IN parameter |
+------+---------------------------------------+
|    1 | We can see the value of the IN parameter |
+------+---------------------------------------+
1 row in set (0.00 sec)

+------+-----------------------------------+
| p_in | IN parameter value has been changed |
+------+-----------------------------------+

|    2 | IN parameter value has been changed |
+------+-----------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @p_in,'We can''t see the changed value from the calling program'

+------+-----------------------------------------------------+
| @p_in | We can't see the changed value from the calling program |
+------+-----------------------------------------------------+
| 1    | We can't see the changed value from the calling program |
+------+-----------------------------------------------------+
1 row in set (0.00 sec)
```

Example 3-4. Example of an OUT parameter

```
mysql> CREATE PROCEDURE sp_demo_out_parameter(OUT p_out INT)

BEGIN
    /* We can't see the value of the OUT parameter */
    SELECT p_out,'We can''t see the value of the OUT parameter';
    /* We can modify it*/
    SET p_out=2;
    SELECT p_out,'OUT parameter value has been changed';

END;
```

```
mysql> SET @p_out=1

Query OK, 0 rows affected (0.00 sec)

mysql> CALL sp_demo_out_parameter(@p_out)

+-------+-----------------------------------------------------------------+
| p_out | We can't see the value of the OUT parameter in the stored program |
+-------+-----------------------------------------------------------------+
|  NULL | We can't see the value of the OUT parameter in the stored program |
+-------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

+-------+--------------------------------------+
| p_out | OUT parameter value has been changed |
+-------+--------------------------------------+
|     2 | OUT parameter value has been changed |

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

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @p_out,"Calling program can see the value of the changed OUT parameter"

+-----------------------------------------------------------+
| Calling program can see the value of the changed OUT parameter |
+-----------------------------------------------------------+
| 2                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

```
mysql> CREATE PROCEDURE sp_demo_inout_parameter(INOUT p_inout INT)

BEGIN

    SELECT p_inout,'We can see the value of the INOUT parameter in the stored program';

    SET p_inout=2;
    SELECT p_inout,'INOUT parameter value has been changed';

END;
//
Query OK, 0 rows affected (0.00 sec)
```

```
SET @p_inout=1
//

Query OK, 0 rows affected (0.00 sec)

CALL sp_demo_inout_parameter(@p_inout) //

+---------+-----------------------------------------------------------------+
| p_inout | We can see the value of the INOUT parameter in the stored program |
+---------+-----------------------------------------------------------------+
|       1 | We can see the value of the INOUT parameter in the stored program |
+---------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

+---------+-----------------------------------------+
| p_inout | INOUT parameter value has been changed |
+---------+-----------------------------------------+
|       2 | INOUT parameter value has been changed |
+---------+-----------------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

SELECT @p_inout ,"Calling program can see the value of the changed INOUT parameter"
//


+---------+-------------------------------------------------------------+
| @p_inout | Calling program can see the value of the changed INOUT parameter |
+---------+-------------------------------------------------------------+
| 2       | Calling program can see the value of the changed INOUT parameter |
+---------+-------------------------------------------------------------+
1 row in set (0.00 sec)
```

User Variables

- *User variables* are special MySQL variables that can be defined and manipulated inside or outside stored programs.
- A feature of the MySQL base product, not the stored program language
- Scope is outside of individual stored programs (can be used like global variables)
- Provide an alternative method of passing information to stored programs
- User variables remain in existence for the duration of a MySQL session and can be accessed by any program or statement running within that session

Example 3-6. Manipulating user variables in the MySQL client

```
mysql> SELECT 'Hello World' into @x ;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @x;
+-------------+
| @x          |
+-------------+
| Hello World |
+-------------+
1 row in set (0.03 sec)
```

Example 3-8. Using a user variable as a "global variable" across stored programs

```
mysql> CREATE PROCEDURE p1()
    ->  SET @last_procedure='p1';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE PROCEDURE p2()
    ->  SELECT CONCAT('Last procedure was ',@last_procedure);
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p1();
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p2();
+-----------------------------------------------+
| CONCAT('Last procedure was ',@last_procedure) |
+-----------------------------------------------+
| Last procedure was p1                         |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

Comments

- Two styles of comments are supported in MySQL stored programs:
  - Two dashes -- followed by a space create a comment that continues until the end of the current line. We'll call these *single-line comments*.
  - C-style comments commence with /* and terminate with */. We'll call these *multiline comments*.

Example 3-9. Example of stored program comments

```
CREATE PROCEDURE comment_demo
    (IN p_input_parameter INT  -- Dummy parameter to illustrate styles
    )
/*
|    Program: comment_demo
|    Purpose: demonstrate comment styles
|    Author:  Guy Harrison
|    Change History:
|        2005-09-21 - Initial
|
*/
```

Operators

- MySQL operators include the familiar operators common to most programming languages, although C-style operators (++, --, +=, etc.) are not supported.

Table 3-2. MySQL mathematical operators

Operator	Description	Example
+	Addition	SET var1=2+2; → 4
-	Subtraction	SET var2=3-2; → 1
*	Multiplication	SET var3=3*2; → 6
/	Division	SET var4=10/3; → 3.3333
DIV	Integer division	SET var5=10 DIV 3; → 3
%	Modulus	SET var6=10%3 ; → 1

Table 3-3. Comparison operators

Operator	Description	Example	Example result
>	Is greater than	1>2	FALSE
<	Is less than	2<1	FALSE
<=	Is less than or equal to	2<=2	TRUE
>=	Is greater than or equal to	3>=2	TRUE
BETWEEN	Value is between two values	5 BETWEEN 1 AND 10	TRUE
NOT BETWEEN	Value is not between two values	5 NOT BETWEEN 1 AND 10	FALSE
IN	Value is in a list	5 IN (1,2,3,4)	FALSE
NOT IN	Value is not in a list	5 NOT IN (1,2,3,4)	TRUE
=	Is equal to	2=3	FALSE
<>, !=	Is not equal to	2<>3	FALSE
<=>	Null safe equal (returns TRUE if both arguments are NULL)	NULL<=>NULL	TRUE
LIKE	Matches a simple pattern	"Guy Harrison" LIKE "Guy%"	TRUE
REGEXP	Matches an extended regular expression	"Guy Harrison" REGEXP "[Gg]reg"	FALSE
IS NULL	Value is NULL	0 IS NULL	FALSE
IS NOT NULL	Value is not NULL	0 IS NOT NULL	TRUE

Table 3-4. Truth table for AND operator

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

Table 3-5. Truth table for the OR operator

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Table 3-6. Truth table for the XOR operator

XOR	TRUE	FALSE	NULL
TRUE	FALSE	TRUE	NULL
FALSE	TRUE	FALSE	NULL
NULL	NULL	NULL	NULL

Built-in Functions

- Most of the functions that MySQL makes available for use in SQL statements can be used within stored programs.
- Fully documented in the MySQL reference manual
- Functions involved in group (multiple-row) operators may be used in SQL but not in stored programs

Function	Description
ABS(*number*)	Returns the absolute value of the number supplied. For instance, ABS(-2.3)=2.3.
CEILING(*number*)	Returns the next highest integer. For instance, CEILING(2.3)=3.
CONCAT(*string1*[,*string2*,*string3*,...])	Returns a string comprised of all the supplied strings joined (concatenated) together.
CURDATE	Returns the current date (without the time portion).
DATE_ADD(*date*,INTERVAL *amount_type*)	Adds the specified interval to the specified date and returns a new date. Valid types include SECOND, MINUTE, HOUR, DAY, MONTH, and YEAR.
DATE_SUB(*date*,INTERVAL *interval_type*)	Subtracts the specified interval from the specified date and returns a new date. Valid types include SECOND, MINUTE, HOUR, DAY, MONTH, and YEAR.
FORMAT(*number*,*decimals*)	Returns a number with a specified number of decimal places and with 1000 separators (usually ",",).
GREATEST(*num1*,*num2*[,*num3*, ...])	Returns the greatest number from all the numbers supplied as arguments.
IF(*test*, *value1*,*value2*)	Tests a logical condition. If TRUE, returns *value1*; otherwise, returns *value2*.
IFNULL(*value*,*value2*)	Returns the value of the first argument, unless that argument is NULL; in that case, it returns the value of the second argument.
INSERT(*string*,*position*,*length*,*new*)	Inserts a string into the middle of another string.
INSTR(*string*,*substring*)	Finds the location of a substring within a string.
ISNULL(*expression*)	Returns 1 if the argument is NULL, 0 otherwise.
LEAST(*num1*,*num2*[,*num3*, ...])	Returns the smallest number from the list of arguments.
LEFT(*string*,*length*)	Returns the leftmost portion of a string.

LENGTH(*string*)	Returns the length of a string in bytes. CHAR_LENGTH can be used if you want to return the number of characters (which could be different if you are using a multibyte character set).
LOCATE(*substring*,*string*[,*number*])	Returns the location of the substring within the string, optionally starting the search at the position given by the third argument.
LOWER(*string*)	Translates the given string into lowercase.
LPAD(*string*,*length*,*padding*)	Left-pads the string to the given length, using the third argument as the pad character.
LTRIM(*string*)	Removes all leading whitespace from a string.
MOD(*num1*,*num2*)	Returns the modulo (remainder) returned by the division of the first number by the second number.
NOW	Returns the current date and time.
POWER(*num1*,*num2*)	Raises *num1* to the power *num2*.

Function	Description
RAND([*seed*])	Returns a random number. The *seed* may be used to initialize the random number generator.
REPEAT(*string*,*number*)	Returns a string consisting of *number* repetitions of the given *string*.
REPLACE(*string*,*old*,*new*)	Replaces all occurrences of *old* with *new* in the given *string*.
ROUND(*number*[,*decimal*])	Rounds a numeric value to the specified number of decimal places.
RPAD(*string*,*length*,*padding*)	Right-pads *string* to the specified *length* using the specified *padding* character.
RTRIM(*string*)	Removes all trailing blanks from *string*.
SIGN(*number*)	Returns -1 if the number is less than 0, 1 if the number is greater than 0, or 0 if the number is equal to 0.
SQRT(*number*)	Returns the square root of the given number.
STRCMP(*string1*,*string2*)	Returns 0 if the two strings are identical, -1 if the first string would sort earlier than the second string, or 1 otherwise.
SUBSTRING(*string*,*position*,*length*)	Extracts *length* characters from *string* starting at the specified *position*.
UPPER(*string*)	Returns the specified string converted to uppercase.
VERSION	Returns a string containing version information for the current MySQL server.

Block Structure

- Most MySQL stored programs consist of one or more blocks
  - CREATE {PROCEDURE|FUNCTION|TRIGGER} *program_name*
    BEGIN
      program_statements
    END;
- The purpose of a block is twofold:
  - *To logically group related code segments*
  - *To control the scope of variables and other objects*

Block Structure

- A block consists of various types of declarations and program code. The order in which these can occur is as follows:
  - Variable and condition declarations
  - Cursor declarations
  - Handler declarations
  - Program code
- Violating this order will produce an error
- *[label:]* BEGIN

  variable and condition declarations
  cursor declarations
  handler declarations

  program code

  END *[label];*

Example 4-1. Declarations within a block are not visible outside the block

```
mysql> CREATE PROCEDURE nested_blocks1( )
BEGIN
        DECLARE outer_variable VARCHAR(20);
        BEGIN
                DECLARE inner_variable VARCHAR(20);
                SET inner_variable='This is my private data';
        END;
        SELECT inner_variable,' This statement causes an error ';
END;
$$

Query OK, 0 rows affected (0.00 sec)

mysql> CALL nested_blocks1( )
-------------

ERROR 1054 (42S22): Unknown column 'inner_variable' in 'field list'
```

Conditional Execution

- IF
  - The IF statement allows you to test the truth of an expression
  - ELSEIF clause is used for all conditional branches after the initial IF
  - The ELSE clause is executed if the Boolean expressions in the IF and ELSEIF clauses all evaluate to false
- IF expression THEN commands
  [ELSEIF expression THEN commands ....]
  [ELSE commands]
  END IF;
- Remember, due to NULL values, NOT TRUE does not always mean FALSE

Example 4-14. Using nested IF to avoid redundant evaluations

```
IF (sale_value > 200) THEN
    CALL free_shipping(sale_id);    /*Free shipping*/

    IF (customer_status='PLATINUM') THEN
        CALL apply_discount(sale_id,20); /* 20% discount */

    ELSEIF (customer_status='GOLD') THEN
        CALL apply_discount(sale_id,15); /* 15% discount */

    ELSEIF (customer_status='SILVER') THEN
        CALL apply_discount(sale_id,10); /* 10% discount */

    ELSEIF (customer_status='BRONZE') THEN
        CALL apply_discount(sale_id,5); /* 5% discount*/
    END IF;

END IF;
```

```
CREATE PROCEDURE discounted_price
     (normal_price NUMERIC(8,2),
      OUT discount_price NUMERIC(8,2))
     NO SQL
BEGIN
     IF (normal_price>500) THEN
        SET discount_price=normal_price*.8;

     ELSEIF (normal_price>100) THEN
        SET discount_price=normal_price*.9;

     ELSE
        SET discount_price=normal_price;

     END IF;

END$$
```

Conditional Execution

- CASE
  - Alternative conditional execution or flow control statement.
- Simple CASE statement
  - CASE *expression*
    WHEN *value* THEN
    statements
    [WHEN value THEN
    statements ...]
    [ELSE
    statements]
    END CASE;
- A CASE statement will raise an exception if none of the conditions apply

Example 4-15. Example of a simple CASE statement

```
CASE customer_status
    WHEN 'PLATINUM'  THEN
        CALL apply_discount(sale_id,20); /* 20% discount */

    WHEN 'GOLD' THEN
        CALL apply_discount(sale_id,15); /* 15% discount */

    WHEN 'SILVER' THEN
        CALL apply_discount(sale_id,10); /* 10% discount */

    WHEN 'BRONZE' THEN
        CALL apply_discount(sale_id,5); /* 5% discount*/
END CASE;
```

Conditional Execution

- Searched CASE statement
  - Functionally equivalent to an IF-ELSEIF-ELSE-END IF block.
- CASE

  WHEN *condition* THEN

  statements

  [WHEN *condition* THEN

  statements...]

  [ELSE

  statements]

  END CASE;

```
CASE
    WHEN  (sale_value >200 AND customer_status='PLATINUM') THEN
          CALL free_shipping(sale_id);     /* Free shipping*/
          CALL apply_discount(sale_id,20); /* 20% discount */

    WHEN (sale_value >200 AND customer_status='GOLD') THEN
         CALL free_shipping(sale_id);     /* Free shipping*/
         CALL apply_discount(sale_id,15); /* 15% discount */

    WHEN (sale_value >200 AND customer_status='SILVER') THEN
         CALL free_shipping(sale_id);     /* Free shipping*/
         CALL apply_discount(sale_id,10); /* 10% discount */

    WHEN (sale_value >200 AND customer_status='BRONZE') THEN
         CALL free_shipping(sale_id);    /* Free shipping*/
         CALL apply_discount(sale_id,5); /* 5% discount*/

    WHEN (sale_value>200) THEN
         CALL free_shipping(sale_id);    /* Free shipping*/
    ELSE
         SET dummy=dummy;

END CASE;
```

Loops

- Allow stored programs to execute statements repetitively
- [*label*:] LOOP
  statements
  END LOOP [*label*];
- 3 types of loops in MySQL:
  - Simple loops using the LOOP and END LOOP clauses
  - Loops that continue while a condition is true, using the WHILE and END WHILE clauses
  - Loops that continue until a condition is true, using the REPEAT and UNTIL clauses

LEAVE Statement

- Allows to terminate a loop
  - LEAVE *label*;
- LEAVE causes the current loop to be terminated. The label matches the loop to be terminated, so if a loop is enclosed within another loop, we can break out of both loops with a single statement.

Example 4-20. Using LEAVE to terminate a loop

```
SET i=1;
myloop: LOOP
    SET i=i+1;
    IF i=10 then
            LEAVE myloop;
    END IF;
END LOOP myloop;
SELECT 'I can count to 10';
```

ITERATE Statement

- The ITERATE statement is used to restart execution at the beginning of a loop, with- out executing any of the remaining statements in the loop.
  - ITERATE *label*;

Example 4-21. Using ITERATE to return to the start of a loop

```
SET i=0;
loop1: LOOP
    SET i=i+1;
    IF i>=10 THEN          /*Last number - exit loop*/
         LEAVE loop1;
    ELSEIF MOD(i,2)=0 THEN /*Even number - try again*/
         ITERATE loop1;
    END IF;

    SELECT CONCAT(i," is an odd number");

END LOOP loop1;
```

REPEAT…UNTIL Loop

- The REPEAT and UNTIL statements can be used to create a loop that continues until some logical condition is met.
  - [*label*:] REPEAT
    statements
    UNTIL *expression*
    END REPEAT [*label*]

Example 4-22. Example of a REPEAT loop

```
SET i=0;
loop1: REPEAT
    SET i=i+1;
    IF MOD(i,2)<>0 THEN /*Even number - try again*/
       Select concat(i," is an odd number");
    END IF;
UNTIL i >= 10
END REPEAT;
```

WHILE Loop

- A WHILE loop executes as long as a condition is true. If the condition is not true to begin with, then the loop will never execute
  - [*label*:] WHILE *expression* DO
    statements
    END WHILE [*label*]

Example 4-24. Odd numbers less than 10 implemented as a WHILE loop

```
SET i=1;
loop1: WHILE i<=10 DO
    IF MOD(i,2)<>0 THEN /*Even number - try again*/
       SELECT CONCAT(i," is an odd number");
    END IF;
    SET i=i+1;
END WHILE loop1;
```

```
DROP PROCEDURE IF EXISTS simple_loop$$
CREATE PROCEDURE simple_loop()
  DETERMINISTIC
BEGIN

  DECLARE counter INT DEFAULT 0;

  simple_loop: LOOP
    SET counter=counter+1;
    IF counter=10 THEN
       LEAVE simple_loop;
    END IF;
  END LOOP simple_loop;
  SELECT 'I can count to 10';
END;$$
```

Error handling

- When an error occurs in a stored program, the default behavior of MySQL is to terminate execution of the program and pass the error out to the calling program.

- If you need a different kind of response to an error, you create an *error handler* that defines the way in which the stored program should respond to one or more error conditions.
  - DECLARE {CONTINUE | EXIT} HANDLER FOR {SQLSTATE *sqlstate_code*| *MySQL error code*| *condition_name*} *stored_program_statement*

- Handlers must be defined after any variable or cursor declarations, but before executing statements

- Examples of common scenarios that call for the definition of error handlers:
  - An embedded SQL statement might return no rows, a NOT FOUND error handler will prevent the stored program from terminating prematurely.
  - A SQL statement might return an error (a constraint violation, for instance), create a handler to prevent program termination. The handler will allow you to process the error and continue program execution.

Types of Handlers

- EXIT
  - When an EXIT handler fires, the currently executing block is terminated. If this block is the main block for the stored program, the procedure terminates, and control is returned to the procedure or external program that invoked the procedure.
  - If the block is enclosed within an outer block inside of the same stored program, control is returned to that outer block.
- CONTINUE
  - With a CONTINUE handler, execution continues with the statement following the one that caused the error to occur.
- In either case, any statements defined within the hander (the *handler actions*) are run before either the EXIT or CONTINUE takes place.

Example 6-6. Example of an EXIT handler

```
1  CREATE PROCEDURE add_department
2         (in_dept_name VARCHAR(30),
3          in_location VARCHAR(30),
4          in_manager_id INT)
5      MODIFIES SQL DATA
6  BEGIN
7      DECLARE duplicate_key INT DEFAULT 0;
8      BEGIN
9           DECLARE EXIT HANDLER FOR 1062 /* Duplicate key*/ SET duplicate_key=1;
10
11          INSERT INTO departments (department_name,location,manager_id)
12          VALUES(in_dept_name,in_location,in_manager_id);
13
14          SELECT CONCAT('Department ',in_dept_name,' created') as "Result";
15     END;
16
17     IF duplicate_key=1 THEN
18          SELECT CONCAT('Failed to insert ',in_dept_name,
19                        ': duplicate key') as "Result";
20     END IF;
21 END$$
```

Example 6-7. Example of a CONTINUE handler

```
CREATE PROCEDURE add_department
       (in_dept_name VARCHAR(30),
        in_location VARCHAR(30),
        in_manager_id INT)
  MODIFIES SQL DATA
BEGIN
    DECLARE duplicate_key INT DEFAULT 0;

    DECLARE CONTINUE HANDLER FOR 1062 /* Duplicate key*/
            SET duplicate_key=1;

    INSERT INTO departments (department_name,location,manager_id)
    VALUES(in_dept_name,in_location,in_manager_id);

    IF duplicate_key=1 THEN
         SELECT CONCAT('Failed to insert ',in_dept_name,
                              ': duplicate key') as "Result";
    ELSE
              SELECT CONCAT('Department ',in_dept_name,' created') as "Result";
    END IF;
END$$
```

Handler Conditions

- The handler condition defines the circumstances under which the handler will be invoked. The circumstance is always associated with an error condition, but you have three choices as to how you define that error:
  - As a MySQL error code (unique to the MySQL server)

```
DECLARE CONTINUE HANDLER FOR 1062 SET duplicate_key=1;
```

  - As an ANSI-standard SQLSTATE code (defined by the ANSI standard, database-independent)

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET duplicate_key=1;
```

  - As a named condition. You may define your own named conditions or use one of the built-in conditions SQLEXCEPTION, SQLWARNING, and NOT FOUND.

```
mysql> CALL nosuch_sp();

ERROR 1305 (42000): PROCEDURE sqltune.nosuch_sp does not exist
```

Handler Examples

- If any error condition arises (other than a NOT FOUND), continue execution after setting l_error=1:

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET l_error=1;
```

- If any error condition arises (other than a NOT FOUND), exit the current block or stored program after issuing a ROLLBACK statement and issuing an error message:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
    SELECT 'Error occurred - terminating';
END;
```

- If MySQL error 1062 (duplicate key value) is encountered, continue execution after executing the SELECT statement (which generates a message for the calling program):

```
DECLARE CONTINUE HANDER FOR 1062
    SELECT 'Duplicate key in index';
```

- If SQLSTATE 23000 (duplicate key value) is encountered, continue execution after executing the SELECT statement (which generates a message for the calling program):

```
DECLARE CONTINUE HANDER FOR SQLSTATE '23000'
    SELECT 'Duplicate key in index';
```

Handler Precedence

- Only one handler executes for each error
- If more than one handler is applicable to a situation, only the most specific handler will execute
  - Handlers based on MySQL error codes are the most specific type of handler, since an error condition will always correspond to a single MySQL error code
  - SQLSTATE codes can sometimes map to many MySQL error codes, so they are less specific
  - General conditions such as SQLEXCEPTION and SQLWARNING are not at all specific
- This strictly defined precedence allows us to define a general-purpose handler for unexpected conditions, while creating a specific handler for those circumstances that we can easily anticipate

Named Conditions

- Improve the readability of handlers by defining a condition declaration, which associates a MySQL error code or SQLSTATE code with a meaningful name
  - DECLARE *condition_name* CONDITION FOR {SQLSTATE *sqlstate_code* | *MySQL_error_code*};

```
DECLARE foreign_key_error CONDITION FOR 1216;

DECLARE CONTINUE HANDLER FOR foreign_key_error MySQL_statements;
```

Interacting with the Database

- Most stored programs involve some kind of interaction with database tables. There are four main types of interactions:
  - Store the results of a SQL statement that returns a single row into local variables.
  - Create a “cursor” that allows the stored program to iterate through the rows returned by a SQL statement.
  - Execute a SQL statement, returning the result set(s) to the calling program.
  - Embed a SQL statement that does not return a result set, such as INSERT, UPDATE, DELETE, etc.

SELECTing INTO Local Variables

- Use the SELECT INTO syntax when querying information from a single row of data (whether retrieved from a single row, an aggregate of many rows, or a join of multiple tables).

- Include an INTO clause “inside” the SELECT statement that tells MySQL where to put the data retrieved by the query.

- SELECT *expression1* [, *expression2* ....]
  INTO *variable1* [, *variable2* ...]
  other SELECT statement clauses

Example 5-2. Using a SELECT-INTO statement

```
CREATE PROCEDURE get_customer_details(in_customer_id INT)
BEGIN
    DECLARE l_customer_name     VARCHAR(30);
    DECLARE l_contact_surname   VARCHAR(30);
    DECLARE l_contact_firstname VARCHAR(30);

    SELECT customer_name, contact_surname,contact_firstname
      INTO l_customer_name,l_contact_surname,l_contact_firstname
      FROM customers
     WHERE customer_id=in_customer_id;

    /* Do something with the customer record */

END;
```

```
DROP PROCEDURE IF EXISTS customer_sales$$
CREATE PROCEDURE customer_sales
        (in_customer_id INT)
   READS SQL DATA
BEGIN
    DECLARE total_sales NUMERIC(8,2);

    SELECT SUM(sale_value)
      INTO total_sales
      FROM sales
     WHERE customer_id=in_customer_id;

    SELECT CONCAT('Total sales for ',in_customer_id,' is ',total_sales);
END;$$
```

Using Cursors

- SELECT INTO is fine for single-row queries, but many applications require the querying of multiple rows of data.
- Use a *cursor* in MySQL to fetch one or more rows from a SQL result set into stored program variables, usually with the intention of performing some row-by-row processing on the result set.
- Define a cursor with the DECLARE statement
  - DECLARE *cursor_name* CURSOR FOR *SELECT_statement*;

Example 5-5. Cursor definition including a stored procedure variable

```
CREATE PROCEDURE cursor_demo (in_customer_id INT)
BEGIN
  DECLARE v_customer_id   INT;
  DECLARE v_customer_name VARCHAR(30);
  DECLARE c1 CURSOR FOR
      SELECT in_customer_id,customer_name
        FROM customers
       WHERE customer_id=in_customer_id;
```

Cursor Statements

- The MySQL stored program language supports three statements for performing operations on cursors:
- OPEN
  - Initializes the result set for the cursor. We must open a cursor before fetching any rows from that cursor.
  - OPEN *cursor_name*;
- FETCH
  - Retrieves the next row from the cursor and moves the cursor "pointer" to the following row in the result set. FETCH *cursor_name* INTO *variable list*;
  - The variable list must contain one variable of a compatible data type for each column returned by the SELECT statement contained in the cursor declaration.
- CLOSE
  - Deactivates the cursor and releases the memory associated with that cursor.
  - CLOSE *cursor_name*;
  - We should close a cursor when we have finished fetching from it, or when we need to open that cursor again after changing a variable that affects the cursor's result set.

Example 5-6. Fetching a single row from a cursor

```
OPEN cursor1;
FETCH cursor1 INTO l_customer_name,l_contact_surname,l_contact_firstname;
CLOSE cursor1;
```

Example 5-7. Simple (flawed) cursor loop

```
DECLARE c_dept CURSOR FOR
        SELECT department_id
          FROM departments;

OPEN c_dept;
dept_cursor: LOOP
    FETCH c_dept INTO l_dept_id;
END LOOP dept_cursor;
CLOSE c_dept;
```

Example 5-11. Cursor loop with REPEAT UNTIL loop

```
DECLARE dept_csr CURSOR FOR
    SELECT department_id,department_name, location
      FROM departments;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
REPEAT
    FETCH dept_csr INTO l_department_id,l_department_name,l_location;
UNTIL no_more_departments
END REPEAT;
CLOSE dept_csr;
SET no_more_departments=0;
```

Example 5-10. A LOOP-LEAVE-END LOOP cursor loop

```
OPEN dept_csr;
dept_loop1:LOOP
    FETCH dept_csr INTO l_department_id,l_department_name,l_location;
    IF no_more_departments=1 THEN
        LEAVE dept_loop1;
    END IF;
    SET l_department_count=l_department_count+1;
END LOOP;
CLOSE dept_csr;
SET no_more_departments=0;
```

Example 5-12. Most REPEAT UNTIL loops also need a LEAVE statement

```
DECLARE dept_csr CURSOR FOR
    SELECT department_id,department_name, location
      FROM departments;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
dept_loop:REPEAT
     FETCH dept_csr INTO l_department_id,l_department_name,l_location;
          IF no_more_departments THEN
             LEAVE dept_loop;
          END IF;
          SET l_department_count=l_department_count+1;
UNTIL no_more_departments
END REPEAT dept_loop;
CLOSE dept_csr;
SET no_more_departments=0;
```

Example 5-14. A cursor WHILE loop

```
DECLARE dept_csr CURSOR FOR
    SELECT department_id,department_name, location
      FROM departments;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
dept_loop:WHILE(no_more_departments=0) DO
     FETCH dept_csr INTO l_department_id,l_department_name,l_location;
     IF no_more_departments=1 THEN
          LEAVE dept_loop;
     END IF;
     SET l_department_count=l_department_count+1;
END WHILE dept_loop;
CLOSE dept_csr;
SET no_more_departments=0;
```

```
DROP PROCEDURE IF EXISTS cursor_example1$$
CREATE PROCEDURE cursor_example(in_department_id INT)
BEGIN
    DECLARE l_employee_id INT;
    DECLARE l_salary      NUMERIC(8,2);
    DECLARE l_department_id INT;
    DECLARE l_new_salary  NUMERIC(8,2);

    DECLARE done          INT DEFAULT 0;

    DECLARE cur1 CURSOR FOR
            SELECT employee_id, salary, department_id
              FROM employees
             WHERE department_id=in_department_id
               FOR UPDATE ;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;

    OPEN cur1;
    emp_loop: LOOP

      FETCH cur1 INTO l_employee_id, l_salary, l_department_id;

      IF done=1 THEN       /* No more rows*/
         LEAVE emp_loop;
      END IF;

    END LOOP emp_loop;
    CLOSE cur1;
END;
```

Returning Result Sets from Stored Procedures

- An *unbounded* SELECT statement—one not associated with an INTO clause or a cursor—returns its result set to the calling program.
- When executing such a stored procedure from the MySQL command line, the results are returned in the same way as if when executing a SELECT or SHOW statement.
- Similar to a view, but allowing more flexibility
- A stored program call can return more than one result set.

Example 5-20. Using unbounded SELECTs to return data to the calling program

```
CREATE PROCEDURE emps_in_dept(in_department_id INT)
BEGIN
    SELECT department_name, location
      FROM departments
     WHERE department_id=in_department_id;

    SELECT employee_id,surname,firstname
      FROM employees
     WHERE department_id=in_department_id;
END;
```

Embedding Non-SELECTs

- “Simple” SQL statements that do not return results can also be embedded in stored programs.

- These statements include DML statements such as UPDATE, INSERT, and DELETE and may also include certain DDL statements such as CREATE TABLE.

Example 5-1. Embedding non-SELECT statements in stored programs

```
CREATE PROCEDURE simple_sqls( )
BEGIN
    DECLARE i INT DEFAULT 1;

    /* Example of a utility statement */
    SET autocommit=0;

    /* Example of DDL statements */
    DROP TABLE IF EXISTS test_table ;
    CREATE TABLE test_table
         (id         INT PRIMARY KEY,
          some_data VARCHAR(30))
      ENGINE=innodb;

    /* Example of an INSERT using a procedure variable */
    WHILE (i<=10) DO
         INSERT INTO TEST_TABLE VALUES(i,CONCAT("record ",i));
         SET i=i+1;
    END WHILE;

    /* Example of an UPDATE using procedure variables*/
    SET i=5;
    UPDATE test_table
       SET some_data=CONCAT("I updated row ",i)
     WHERE id=i;

    /* DELETE with a procedure variable */
    DELETE FROM test_table
     WHERE id>i;

END;
```

```
CREATE PROCEDURE sp_update_salary
    (in_employee_id INT, in_new_salary NUMERIC(8,2))
    DETERMINISTIC
BEGIN
    IF in_new_salary <5000 OR in_new_salary > 500000 THEN
        SELECT 'Illegal salary; salary must be between $5,000 and $500,000';
    ELSE
        UPDATE employees
                        SET salary=in_new_salary
                WHERE employee_id=in_employee_id;
    END IF;
END$$
```

Calling Stored Programs from Stored Programs

- Simply use the CALL statement as if outside a procedure

```
CREATE PROCEDURE call_example
    (employee_id INT, employee_type VARCHAR(20))
    NO SQL
BEGIN
    DECLARE l_bonus_amount NUMERIC(8,2);

    IF employee_type='MANAGER' THEN
       CALL calc_manager_bonus(employee_id ,l_bonus_amount);
    ELSE
       CALL calc_minion_bonus(employee_id,l_bonus_amount);
    END IF;
    CALL grant_bonus(employee_id,l_bonus_amount);
END;
```

```
CREATE PROCEDURE putting_it_all_together(in_department_id INT)
    DETERMINISTIC MODIFIES SQL DATA
BEGIN
    DECLARE l_employee_id INT;
    DECLARE l_salary      NUMERIC(8,2);
    DECLARE l_department_id INT;
    DECLARE l_new_salary  NUMERIC(8,2);
    DECLARE done          INT DEFAULT 0;

    DECLARE cur1 CURSOR FOR
            SELECT employee_id, salary, department_id
              FROM employees
             WHERE department_id=in_department_id
               FOR UPDATE ;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;

    CREATE TEMPORARY TABLE IF NOT EXISTS emp_raises
      (employee_id INT, department_id INT, new_salary NUMERIC(8,2));

    OPEN cur1;
    emp_loop: LOOP

      FETCH cur1 INTO l_employee_id, l_salary, l_department_id;

      IF done=1 THEN       /* No more rows*/
         LEAVE emp_loop;
      END IF;

      CALL new_salary(l_employee_id,l_new_salary); /*get new salary*/

      IF (l_new_salary<>l_salary) THEN             /*Salary changed*/

         UPDATE employees
            SET salary=l_new_salary
          WHERE employee_id=l_employee_id;
         /* Keep track of changed salaries*/
         INSERT INTO emp_raises (employee_id,department_id,new_salary)
          VALUES (l_employee_id,l_department_id,l_new_salary);
      END IF;

    END LOOP emp_loop;
    CLOSE cur1;
    /* Print out the changed salaries*/
    SELECT employee_id,department_id,new_salary from emp_raises;
    COMMIT;
END;
```