21.2.2. `CREATE PROCEDURE` and `CREATE FUNCTION` Syntax

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

CREATE
    [DEFINER = { user | CURRENT_USER }]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic:
    LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'

routine_body:
    Valid SQL procedure statement
```

These statements create stored routines. By default, a routine is associated with the default database. To associate the routine explicitly with a given database, specify the name as *`db_name.sp_name`* when you create it.

As of MySQL 5.0.3, to execute these statements, it is necessary to have the `CREATE ROUTINE` privilege. MySQL automatically grants the `ALTER ROUTINE` and `EXECUTE` privileges to the routine creator.

The `DEFINER` and `SQL SECURITY` clauses specify the security context to be used when checking access privileges at routine execution time, as described later.

If the routine name is the same as the name of a built-in SQL function, you must use a space between the name and the following parenthesis when defining the routine, or a syntax error occurs. This is also true when you invoke the routine later. For this reason, we suggest that it is better to avoid re-using the names of existing SQL functions for your own stored routines.

The `IGNORE_SPACE` SQL mode applies to built-in functions, not to stored routines. It is always allowable to have spaces after a routine name, regardless of whether `IGNORE_SPACE` is enabled.

The parameter list enclosed within parentheses must always be present. If there are no parameters, an empty parameter list of `()` should be used.

Each parameter can be declared to use any valid data type, except that the `COLLATE` attribute cannot be used.

Each parameter is an `IN` parameter by default. To specify otherwise for a parameter, use the keyword `OUT` or `INOUT` before the parameter name.

Note

Specifying a parameter as `IN`, `OUT`, or `INOUT` is valid only for a `PROCEDURE`. (`FUNCTION` parameters are always regarded as `IN` parameters.)

An `IN` parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns. An `OUT` parameter passes a value from the procedure back to the caller. Its initial value is `NULL` within the procedure, and its value is visible to the caller when the procedure returns. An `INOUT` parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.

For each `OUT` or `INOUT` parameter, pass a user-defined variable so that you can obtain its value when the procedure returns. If you are calling the procedure from within another stored procedure or function, you can also pass a routine parameter or local routine variable as an `IN` or `INOUT` parameter.

The `RETURNS` clause may be specified only for a `FUNCTION`, for which it is mandatory. It indicates the return type of the function, and the function body must contain a `RETURN` *`value`* statement. If the `RETURN` statement returns a value of a different type, the value is coerced to the proper type. For example, if a function specifies an `ENUM` or `SET` value in the `RETURNS` clause, but the `RETURN` statement returns an integer, the value returned from the function is the string for the corresponding `ENUM` member of set of `SET` members.

The *`routine_body`* consists of a valid SQL procedure statement. This can be a simple statement such as `SELECT` or `INSERT`, or it can be a compound statement written using `BEGIN` and `END`.. Compound statements can contain declarations, loops, and other control structure statements.

MySQL stores the `sql_mode` system variable setting that is in effect at the time a routine is created, and always executes the routine with this setting in force, *regardless of the current server SQL mode*.

The `CREATE FUNCTION` statement was used in earlier versions of MySQL to support UDFs (user-defined functions). UDFs continue to be supported, even with the existence of stored functions. A UDF can be regarded as an external stored function. However, do note that stored functions share their namespace with UDFs. A procedure or function is considered “deterministic” if it always produces the same result for the same input parameters, and “not deterministic”

otherwise. If neither DETERMINISTIC nor NOT DETERMINISTIC is given in the routine definition, the default is NOT DETERMINISTIC.

A routine that contains the NOW() function (or its synonyms) or RAND() is non-deterministic, but it might still be replication-safe. For NOW(), the binary log includes the timestamp and replicates correctly. RAND() also replicates correctly as long as it is invoked only once within a routine. (You can consider the routine execution timestamp and random number seed as implicit inputs that are identical on the master and slave.)

In versions prior to 5.0.44-sp1, the DETERMINISTIC characteristic is accepted, but not used by the optimizer. However, if binary logging is enabled, this characteristic always affects which routine definitions MySQL accepts. Several characteristics provide information about the nature of data use by the routine. In MySQL, these characteristics are advisory only. The server does not use them to constrain what kinds of statements a routine will be allowed to execute.

- CONTAINS SQL indicates that the routine does not contain statements that read or write data. This is the default if none of these characteristics is given explicitly. Examples of such statements are SET @x = 1 or DO RELEASE_LOCK('abc'), which execute but neither read nor write data.
- NO SQL indicates that the routine contains no SQL statements.
- READS SQL DATA indicates that the routine contains statements that read data (for example, SELECT), but not statements that write data.
- MODIFIES SQL DATA indicates that the routine contains statements that may write data (for example, INSERT or DELETE).

The SQL SECURITY characteristic can be used to specify whether the routine should be executed using the permissions of the user who creates the routine or the user who invokes it. The default value is DEFINER. This feature is new in SQL:2003. The creator or invoker must have permission to access the database with which the routine is associated. As of MySQL 5.0.3, it is necessary to have the EXECUTE privilege to be able to execute the routine. The user that must have this privilege is either the definer or invoker, depending on how the SQL SECURITY characteristic is set.

The optional DEFINER clause specifies the MySQL account to be used when checking access privileges at routine execution time for routines that have the SQL SECURITY DEFINER characteristic. The DEFINER clause was added in MySQL 5.0.20.

If a *user* value is given for the DEFINER clause, it should be a MySQL account in '*user_name*'@'*host_name*' format (the same format used in the GRANT statement). The *user_name* and *host_name* values both are required. The definer can also be given as CURRENT_USER or CURRENT_USER(). The default DEFINER value is the user who executes the CREATE PROCEDURE or CREATE FUNCTION or statement. (This is the same as DEFINER = CURRENT_USER.)

If you specify the DEFINER clause, these rules determine the legal DEFINER user values:

- If you do not have the SUPER privilege, the only legal *user* value is your own account, either specified literally or by using CURRENT_USER. You cannot set the definer to some other account.
- If you have the SUPER privilege, you can specify any syntactically legal account name. If the account does not actually exist, a warning is generated.

  Although it is possible to create routines with a non-existent DEFINER value, an error occurs if the routine executes with definer privileges but the definer does not exist at execution time.

When the routine is invoked, an implicit USE *db_name* is performed (and undone when the routine terminates). USE statements within stored routines are disallowed.

As of MySQL 5.0.18, the server uses the data type of a routine parameter or function return value as follows. These rules also apply to local routine variables created with the DECLARE statement. Assignments are checked for data type mismatches and overflow. Conversion and overflow problems result in warnings, or errors in strict mode.

- Only scalar values can be assigned to parameters or variables. For example, a statement such as SET x = (SELECT 1, 2) is invalid.
- For character data types, if there is a CHARACTER SET clause in the declaration, the specified character set and its default collation are used. If there is no such clause, as of MySQL 5.0.25, the database character set and collation that are in effect at the time the server loads the routine into the routine cache are used. (These are given by the values of the character_set_database and collation_database system variables.) If the database character set or collation change while the routine is in the cache, routine execution is unaffected by the change until the next time the server reloads the routine into the cache. The COLLATE attribute is not supported. (This includes use of BINARY, because in this context BINARY specifies the binary collation of the character set.)

  In MySQL 5.1, the database character set and collation in effect at the time the routine is created are used. Subsequent changes to the database character set or collation do not affect routine execution.

Before MySQL 5.0.18, parameters, return values, and local variables are treated as items in expressions, and are subject to automatic (silent) conversion and truncation. Stored functions ignore the sql_mode setting.

The COMMENT clause is a MySQL extension, and may be used to describe the stored routine. This information is displayed by the SHOW CREATE PROCEDURE and SHOW CREATE FUNCTION statements.

MySQL allows routines to contain DDL statements, such as CREATE and DROP. MySQL also allows stored procedures (but not stored functions) to contain SQL transaction statements such as COMMIT. Stored functions may not contain statements that do explicit or implicit commit or

rollback. Support for these statements is not required by the SQL standard, which states that each DBMS vendor may decide whether to allow them.

Statements that return a result set cannot be used within a stored function. This includes `SELECT` statements that do not use `INTO` to fetch column values into variables, `SHOW` statements, and other statements such as `EXPLAIN`. For statements that can be determined at function definition time to return a result set, a `Not allowed to return a result set from a function` error occurs (`ER_SP_NO_RETSET`). For statements that can be determined only at runtime to return a result set, a `PROCEDURE %s can't return a result set in the given context` error occurs (`ER_SP_BADSELECT`).

Note

Before MySQL 5.0.10, stored functions created with `CREATE FUNCTION` must not contain references to tables, with limited exceptions. They may include some `SET` statements that contain table references, for example `SET a:= (SELECT MAX(id) FROM t)`, and `SELECT` statements that fetch values directly into variables, for example `SELECT i INTO var1 FROM t`.

The following is an example of a simple stored procedure that uses an `OUT` parameter. The example uses the **mysql** client `delimiter` command to change the statement delimiter from `;` to `//` while the procedure is being defined. This allows the `;` delimiter used in the procedure body to be passed through to the server rather than being interpreted by **mysql** itself.

```
mysql> delimiter //

mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
    -> BEGIN
    ->   SELECT COUNT(*) INTO param1 FROM t;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;

mysql> CALL simpleproc(@a);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @a;
+------+
| @a   |
+------+
| 3    |
+------+
1 row in set (0.00 sec)
```

When using the `delimiter` command, you should avoid the use of the backslash (“\”) character because that is the escape character for MySQL.

The following is an example of a function that takes a parameter, performs an operation using an SQL function, and returns the result. In this case, it is unnecessary to use `delimiter` because the function definition contains no internal `;` statement delimiters:

```
mysql> CREATE FUNCTION hello (s CHAR(20))
mysql> RETURNS CHAR(50) DETERMINISTIC
    -> RETURN CONCAT('Hello, ',s,'!');
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT hello('world');
+----------------+
| hello('world') |
+----------------+
| Hello, world!  |
+----------------+
1 row in set (0.00 sec)
```

For information about invoking stored procedures from within programs written in a language that has a MySQL interface, see Section 21.2.6, "`CALL` Statement Syntax".

21.2.3. `ALTER PROCEDURE` and `ALTER FUNCTION` Syntax

```
ALTER {PROCEDURE | FUNCTION} sp_name [characteristic ...]

characteristic:
    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
  | COMMENT 'string'
```

This statement can be used to change the characteristics of a stored procedure or function. As of MySQL 5.0.3, you must have the `ALTER ROUTINE` privilege for the routine. (That privilege is granted automatically to the routine creator.) More than one change may be specified in an `ALTER PROCEDURE` or `ALTER FUNCTION` statement.

21.2.4. `DROP FUNCTION` Syntax

The `DROP FUNCTION` statement is used to drop stored functions and user-defined functions (UDFs):

- For information about dropping stored functions,
- For information about dropping user-defined functions,
- **`DROP PROCEDURE` and `DROP FUNCTION` Syntax**

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

This statement is used to drop a stored procedure or function. That is, the specified routine is removed from the server. As of MySQL 5.0.3, you must have the `ALTER ROUTINE` privilege for the routine. (That privilege is granted automatically to the routine creator.)

The `IF EXISTS` clause is a MySQL extension. It prevents an error from occurring if the procedure or function does not exist. A warning is produced that can be viewed with `SHOW WARNINGS`.

`DROP FUNCTION` is also used to drop user-defined functions (see Section 12.5.3.2, "`DROP FUNCTION` Syntax").

21.2.6. `CALL` Statement Syntax

```
CALL sp_name([parameter[,...]])
CALL sp_name[()]
```

The `CALL` statement invokes a procedure that was defined previously with `CREATE PROCEDURE`.

`CALL` can pass back values to its caller using parameters that are declared as `OUT` or `INOUT` parameters. It also "returns" the number of rows affected, which a client program can obtain at the SQL level by calling the `ROW_COUNT()` function and from C by calling the `mysql_affected_rows()` C API function.

As of MySQL 5.0.30, stored procedures that take no arguments can be invoked without parentheses. That is, `CALL p()` and `CALL p` are equivalent.

To get back a value from a procedure using an `OUT` or `INOUT` parameter, pass the parameter by means of a user variable, and then check the value of the variable after the procedure returns. (If you are calling the procedure from within another stored procedure or function, you can also pass a routine parameter or local routine variable as an `IN` or `INOUT` parameter.) For an `INOUT` parameter, initialize its value before passing it to the procedure. The following procedure has an `OUT` parameter that the procedure sets to the current server version, and an `INOUT` value that the procedure increments by one from its current value:

```
CREATE PROCEDURE p (OUT ver_param VARCHAR(25), INOUT incr_param INT)
BEGIN
  # Set value of OUT parameter
  SELECT VERSION() INTO ver_param;
  # Increment value of INOUT parameter
  SET incr_param = incr_param + 1;
END;
```

Before calling the procedure, initialize the variable to be passed as the `INOUT` parameter. After calling the procedure, the values of the two variables will have been set or modified:

```
mysql> SET @increment = 10;
mysql> CALL p(@version, @increment);
mysql> SELECT @version, @increment;
+------------+------------+
| @version   | @increment |
+------------+------------+
| 5.0.25-log | 11         |
+------------+------------+
```

If you write C programs that use the CALL SQL statement to execute stored procedures that produce result sets, you *must* set the CLIENT_MULTI_RESULTS flag, either explicitly, or implicitly by setting CLIENT_MULTI_STATEMENTS when you call mysql_real_connect(). This is because each such stored procedure produces multiple results: the result sets returned by statements executed within the procedure, as well as a result to indicate the call status. To process the result of a CALL statement, use a loop that calls mysql_next_result() to determine whether there are more results. For an example, see Section 26.2.9, "C API Handling of Multiple Statement Execution".

For programs written in a language that provides a MySQL interface, there is no native method for directly retrieving the results of OUT or INOUT parameters from CALL statements. To get the parameter values, pass user-defined variables to the procedure in the CALL statement and then execute a SELECT statement to produce a result set containing the variable values. The following example illustrates the technique (without error checking) for a stored procedure p1 that has two OUT parameters.

```
mysql_query(mysql, "CALL p1(@param1, @param2)");
mysql_query(mysql, "SELECT @param1, @param2");
result = mysql_store_result(mysql);
row = mysql_fetch_row(result);
mysql_free_result(result);
```

After the preceding code executes, row[0] and row[1] contain the values of @param1 and @param2, respectively.

To handle INOUT parameters, execute a statement prior to the CALL that sets the user variables to the values to be passed to the procedure.

21.2.7. BEGIN ... END Compound Statement Syntax

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

BEGIN ... END syntax is used for writing compound statements, which can appear within stored routines and triggers. A compound statement can contain multiple statements, enclosed by the BEGIN and END keywords. *statement_list* represents a list of one or more statements. Each statement within *statement_list* must be terminated by a semicolon (;) statement delimiter. Note that *statement_list* is optional, which means that the empty compound statement (BEGIN END) is legal.

Use of multiple statements requires that a client is able to send statement strings containing the ; statement delimiter. This is handled in the **mysql** command-line client with the delimiter command. Changing the ; end-of-statement delimiter (for example, to //) allows ; to be used in a routine body. For an example, see Section 21.2.2, "CREATE PROCEDURE and CREATE FUNCTION Syntax".

A compound statement can be labeled. *end_label* cannot be given unless *begin_label* also is present. If both are present, they must be the same.

The optional [NOT] ATOMIC clause is not yet supported. This means that no transactional savepoint is set at the start of the instruction block and the BEGIN clause used in this context has no effect on the current transaction.

21.2.8. DECLARE Statement Syntax

The DECLARE statement is used to define various items local to a routine:

- Local variables. See Section 21.2.9, "Variables in Stored Routines".
- Conditions and handlers. See Section 21.2.10, "Conditions and Handlers".
- Cursors. See Section 21.2.11, "Cursors".

The SIGNAL and RESIGNAL statements are not currently supported.

DECLARE is allowed only inside a BEGIN ... END compound statement and must be at its start, before any other statements.

Declarations must follow a certain order. Cursors must be declared before declaring handlers, and variables and conditions must be declared before declaring either cursors or handlers.

21.2.9. Variables in Stored Routines

You may declare and use variables within a routine.

21.2.9.1. DECLARE Local Variables

```
DECLARE var_name[,...] type [DEFAULT value]
```

This statement is used to declare local variables. To provide a default value for the variable, include a DEFAULT clause. The value can be specified as an expression; it need not be a constant. If the DEFAULT clause is missing, the initial value is NULL.

Local variables are treated like routine parameters with respect to data type and overflow checking. See Section 21.2.2, "CREATE PROCEDURE and CREATE FUNCTION Syntax".

The scope of a local variable is within the BEGIN ... END block where it is declared. The variable can be referred to in blocks nested within the declaring block, except those blocks that declare a variable with the same name.

21.2.9.2. Variable `SET` Statement

```
SET var_name = expr [, var_name = expr] ...
```

The `SET` statement in stored routines is an extended version of the general `SET` statement. Referenced variables may be ones declared inside a routine, or global system variables.

The `SET` statement in stored routines is implemented as part of the pre-existing `SET` syntax. This allows an extended syntax of `SET a=x, b=y, ...` where different variable types (locally declared variables and global and session server variables) can be mixed. This also allows combinations of local variables and some options that make sense only for system variables; in that case, the options are recognized but ignored.

21.2.9.3. `SELECT ... INTO` Statement

```
SELECT col_name[,...] INTO var_name[,...] table_expr
```

This `SELECT` syntax stores selected columns directly into variables. Therefore, only a single row may be retrieved.

```
SELECT id,data INTO x,y FROM test.t1 LIMIT 1;
```

User variable names are not case sensitive. See Section 8.4, “User-Defined Variables”.

Important

SQL variable names should not be the same as column names. If an SQL statement, such as a `SELECT ... INTO` statement, contains a reference to a column and a declared local variable with the same name, MySQL currently interprets the reference as the name of a variable. For example, in the following statement, `xname` is interpreted as a reference to the `xname` *variable* rather than the `xname` *column*:

```
CREATE PROCEDURE sp1 (x VARCHAR(5))
  BEGIN
    DECLARE xname VARCHAR(5) DEFAULT 'bob';
    DECLARE newname VARCHAR(5);
    DECLARE xid INT;

    SELECT xname,id INTO newname,xid
      FROM table1 WHERE xname = xname;
    SELECT newname;
  END;
```

When this procedure is called, the `newname` variable returns the value `'bob'` regardless of the value of the `table1.xname` column.

See also Section F.1, “Restrictions on Stored Routines and Triggers”.

21.2.10. Conditions and Handlers

Certain conditions may require specific handling. These conditions can relate to errors, as well as to general flow control inside a routine.

21.2.10.1. DECLARE Conditions

```
DECLARE condition_name CONDITION FOR condition_value

condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | mysql_error_code
```

This statement specifies conditions that need specific handling. It associates a name with a specified error condition. The name can subsequently be used in a `DECLARE HANDLER` statement. See Section 21.2.10.2, "DECLARE Handlers".

A *condition_value* can be an SQLSTATE value or a MySQL error code. For a list of SQLSTATE and error values, see Section B.2, "Server Error Codes and Messages".

21.2.10.2. DECLARE Handlers

```
DECLARE handler_type HANDLER FOR condition_value[,...] statement

handler_type:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
  | mysql_error_code
```

The `DECLARE ... HANDLER` statement specifies handlers that each may deal with one or more conditions. If one of these conditions occurs, the specified *statement* is executed. *statement* can be a simple statement (for example, `SET var_name = value`), or it can be a compound statement written using `BEGIN` and `END` (see Section 21.2.7, "BEGIN ... END Compound Statement Syntax").

For a `CONTINUE` handler, execution of the current routine continues after execution of the handler statement. For an `EXIT` handler, execution terminates for the `BEGIN ... END` compound

statement in which the handler is declared. (This is true even if the condition occurs in an inner block.) The `UNDO` handler type statement is not yet supported.

If a condition occurs for which no handler has been declared, the default action is `EXIT`.

A *`condition_value`* can be any of the following values:

- An SQLSTATE value or a MySQL error code. You should not use SQLSTATE value `'00000'` or error code 0, because those indicate sucess rather than an error condition. For a list of SQLSTATE and error values, see Section B.2, "Server Error Codes and Messages".
- A condition name previously specified with `DECLARE ... CONDITION`. See Section 21.2.10.1, "`DECLARE` Conditions".
- `SQLWARNING` is shorthand for all SQLSTATE codes that begin with `01`.
- `NOT FOUND` is shorthand for all SQLSTATE codes that begin with `02`. This is relevant only within the context of cursors and is used to control what happens when a cursor reaches the end of a data set.
- `SQLEXCEPTION` is shorthand for all SQLSTATE codes not caught by `SQLWARNING` or `NOT FOUND`.

The example associates a handler with SQLSTATE 23000, which occurs for a duplicate-key error. Notice that `@x` is `3`, which shows that MySQL executed to the end of the procedure. If the line `DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;` had not been present, MySQL would have taken the default path (`EXIT`) after the second `INSERT` failed due to the `PRIMARY KEY` constraint, and `SELECT @x` would have returned `2`.

If you want to ignore a condition, you can declare a `CONTINUE` handler for it and associate it with an empty block. For example:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;
```

The statement associated with a handler cannot use `ITERATE` or `LEAVE` to refer to labels for blocks that enclose the handler declaration. That is, the scope of a block label does not include the code for handlers declared within the block. Consider the following example, where the `REPEAT` block has a label of `retry`:

```
CREATE PROCEDURE p ()
BEGIN
  DECLARE i INT DEFAULT 3;
  retry:
    REPEAT
      BEGIN
        DECLARE CONTINUE HANDLER FOR SQLWARNING
          BEGIN
            ITERATE retry;  # illegal
          END;
      END;
      IF i < 0 THEN
```

```
        LEAVE retry;        # legal
      END IF;
      SET i = i - 1;
    UNTIL FALSE END REPEAT;
END;
```

The label is in scope for the `IF` statement within the block. It is not in scope for the `CONTINUE` handler, so the reference there is invalid and results in an error:

```
ERROR 1308 (42000): LEAVE with no matching label: retry
```

To avoid using references to outer labels in handlers, you can use different strategies:

- If you want to leave the block, you can use an `EXIT` handler:
- `DECLARE EXIT HANDLER FOR SQLWARNING BEGIN END;`
- If you want to iterate, you can set a status variable in the handler that can be checked in the enclosing block to determine whether the handler was invoked. The following example uses the variable `done` for this purpose:
- `CREATE PROCEDURE p ()`
- `BEGIN`
- `  DECLARE i INT DEFAULT 3;`
- `  DECLARE done INT DEFAULT FALSE;`
- `  retry:`
- `    REPEAT`
- `      BEGIN`
- `        DECLARE CONTINUE HANDLER FOR SQLWARNING`
- `          BEGIN`
- `            SET done = TRUE;`
- `          END;`
- `      END;`
- `      IF NOT done AND i < 0 THEN`
- `        LEAVE retry;`
- `      END IF;`
- `      SET i = i - 1;`
- `    UNTIL FALSE END REPEAT;`
- `END;`

21.2.11. Cursors

Cursors are supported inside stored procedures and functions and triggers. The syntax is as in embedded SQL. Cursors currently have these properties:

- Asensitive: The server may or may not make a copy of its result table

- Read only: Not updatable
- Non-scrollable: Can be traversed only in one direction and cannot skip rows

Cursors must be declared before declaring handlers. Variables and conditions must be declared before declaring either cursors or handlers.

Example:

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT 0;
  DECLARE a CHAR(16);
  DECLARE b,c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

  OPEN cur1;
  OPEN cur2;

  REPEAT
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF NOT done THEN
       IF b < c THEN
          INSERT INTO test.t3 VALUES (a,b);
       ELSE
          INSERT INTO test.t3 VALUES (a,c);
       END IF;
    END IF;
  UNTIL done END REPEAT;

  CLOSE cur1;
  CLOSE cur2;
END
```

21.2.11.1. Declaring Cursors

```
DECLARE cursor_name CURSOR FOR select_statement
```

This statement declares a cursor. Multiple cursors may be declared in a routine, but each cursor in a given block must have a unique name.

The `SELECT` statement cannot have an `INTO` clause.

21.2.11.2. Cursor `OPEN` Statement

```
OPEN cursor_name
```

This statement opens a previously declared cursor.

21.2.11.3. Cursor FETCH Statement

```
FETCH cursor_name INTO var_name [, var_name] ...
```

This statement fetches the next row (if a row exists) using the specified open cursor, and advances the cursor pointer.

If no more rows are available, a No Data condition occurs with SQLSTATE value 02000. To detect this condition, you can set up a handler for it (or for a NOT FOUND condition). An example is shown in Section 21.2.11, "Cursors".

21.2.11.4. Cursor CLOSE Statement

```
CLOSE cursor_name
```

This statement closes a previously opened cursor.

If not closed explicitly, a cursor is closed at the end of the compound statement in which it was declared.

21.2.12. Flow Control Constructs

The IF, CASE, ITERATE, LEAVE LOOP, WHILE, and REPEAT constructs are fully implemented.

Many of these constructs contain other statements, as indicated by the grammar specifications in the following sections. Such constructs may be nested. For example, an IF statement might contain a WHILE loop, which itself contains a CASE statement.

FOR loops are not currently supported.

21.2.12.1. IF Statement

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

`IF` implements a basic conditional construct. If the *`search_condition`* evaluates to true, the corresponding SQL statement list is executed. If no *`search_condition`* matches, the statement list in the `ELSE` clause is executed. Each *`statement_list`* consists of one or more statements.

Note

There is also an `IF()` *function*, which differs from the `IF` *statement* described here. See Section 11.3, "Control Flow Functions".

An `IF ... END IF` block — like all other flow-control blocks used within stored routines — must be terminated with a semicolon, as shown in this example:

```
DELIMITER //

CREATE FUNCTION SimpleCompare(n INT, m INT)
  RETURNS VARCHAR(20)

  BEGIN
    DECLARE s VARCHAR(20);

    IF n > m THEN SET s = '>';
    ELSEIF n = m THEN SET s = '=';
    ELSE SET s = '<';
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m);

    RETURN s;
  END //

DELIMITER ;
```

As with other flow-control constructs, `IF ... END IF` blocks may be nested within other flow-control constructs, including other `IF` statements. Each `IF` must be terminated by its own `END IF` followed by a semicolon. You can use indentation to make nested flow-control blocks more easily readable by humans (although this is not required by MySQL), as shown here:

```
DELIMITER //

CREATE FUNCTION VerboseCompare (n INT, m INT)
  RETURNS VARCHAR(50)

  BEGIN
    DECLARE s VARCHAR(50);

    IF n = m THEN SET s = 'equals';
    ELSE
      IF n > m THEN SET s = 'greater';
      ELSE SET s = 'less';
      END IF;

      SET s = CONCAT('is ', s, ' than');
```

```
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m, '.');

    RETURN s;
  END //

DELIMITER ;
```

In this example, the inner `IF` is evaluated only if `n` is not equal to `m`.

21.2.12.2. `CASE` Statement

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

Or:

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

The `CASE` statement for stored routines implements a complex conditional construct. If a *`search_condition`* evaluates to true, the corresponding SQL statement list is executed. If no search condition matches, the statement list in the `ELSE` clause is executed. Each *`statement_list`* consists of one or more statements.

Note

If no search condition matches the value tested, and the `CASE` statement contains no `ELSE` clause, a Case not found for CASE statement error results.

Each *`statement_list`* consists of one or more statements; an empty *`statement_list`* is not allowed. To handle situations where no value is matched by any `WHEN` clause, use an `ELSE` containing an empty `BEGIN ... END` block, as shown in this example:

```
DELIMITER |

CREATE PROCEDURE p()
  BEGIN
    DECLARE v INT DEFAULT 1;

    CASE v
      WHEN 2 THEN SELECT v;
      WHEN 3 THEN SELECT 0;
      ELSE
```

```
        BEGIN
        END;
    END CASE;
  END;
  |
```

(The indentation used here in the `ELSE` clause is for purposes of clarity only, and is not otherwise significant.)

The syntax of the `CASE` *statement* used inside stored routines differs slightly from that of the SQL `CASE` *expression* described in Section 11.3, "Control Flow Functions". The `CASE` statement cannot have an `ELSE NULL` clause, and it is terminated with `END CASE` instead of `END`.

21.2.12.3. `LOOP` Statement

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

`LOOP` implements a simple loop construct, enabling repeated execution of the statement list, which consists of one or more statements. The statements within the loop are repeated until the loop is exited; usually this is accomplished with a `LEAVE` statement.

A `LOOP` statement can be labeled. *`end_label`* cannot be given unless *`begin_label`* also is present. If both are present, they must be the same.

21.2.12.4. `LEAVE` Statement

```
LEAVE label
```

This statement is used to exit any labeled flow control construct. It can be used within `BEGIN ... END` or loop constructs (`LOOP`, `REPEAT`, `WHILE`).

21.2.12.5. `ITERATE` Statement

```
ITERATE label
```

`ITERATE` can appear only within `LOOP`, `REPEAT`, and `WHILE` statements. `ITERATE` means "do the loop again."

Example:

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

```
  SET @x = p1;
END
```

21.2.12.6. `REPEAT` Statement

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

The statement list within a `REPEAT` statement is repeated until the *`search_condition`* is true. Thus, a `REPEAT` always enters the loop at least once. *`statement_list`* consists of one or more statements.

A `REPEAT` statement can be labeled. *`end_label`* cannot be given unless *`begin_label`* also is present. If both are present, they must be the same.

Example:

```
mysql> delimiter //

mysql> CREATE PROCEDURE dorepeat(p1 INT)
    -> BEGIN
    ->   SET @x = 0;
    ->   REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
    -> END
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL dorepeat(1000)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+------+
| @x   |
+------+
| 1001 |
+------+
1 row in set (0.00 sec)
```

21.2.12.7. `WHILE` Statement

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

The statement list within a `WHILE` statement is repeated as long as the *`search_condition`* is true. *`statement_list`* consists of one or more statements.

A `WHILE` statement can be labeled. *`end_label`* cannot be given unless *`begin_label`* also is present. If both are present, they must be the same.

Example:

```
CREATE PROCEDURE dowhile()
BEGIN
  DECLARE v1 INT DEFAULT 5;

  WHILE v1 > 0 DO
    ...
    SET v1 = v1 - 1;
  END WHILE;
END
```

21.2.13. `RETURN` Statement Syntax

```
RETURN expr
```

The `RETURN` statement terminates execution of a stored function and returns the value *`expr`* to the function caller. There must be at least one `RETURN` statement in a stored function. There may be more than one if the function has multiple exit points.

This statement is not used in stored procedures or triggers.

21.3. Stored Procedures, Functions, Triggers, and `LAST_INSERT_ID()`

Within the body of a stored routine (procedure or function) or a trigger, the value of `LAST_INSERT_ID()` changes the same way as for statements executed outside the body of these kinds of objects (see Section 11.10.3, “Information Functions”). The effect of a stored routine or trigger upon the value of `LAST_INSERT_ID()` that is seen by following statements depends on the kind of routine:

- If a stored procedure executes statements that change the value of `LAST_INSERT_ID()`, the changed value will be seen by statements that follow the procedure call.
- For stored functions and triggers that change the value, the value is restored when the function or trigger ends, so following statements will not see a changed value.

21.4. Binary Logging of Stored Routines and Triggers

The binary log contains information about SQL statements that modify database contents. This information is stored in the form of “events” that describe the modifications. The binary log has two important purposes:

- For replication, the binary log is used on master replication servers as a record of the statements to be sent to slave servers. The master server sends the events contained in its binary log to its slaves, which execute those events to make the same data changes that were made on the master. See Section 18.4, “Replication Implementation Overview”.

- Certain data recovery operations require use of the binary log. After a backup file has been restored, the events in the binary log that were recorded after the backup was made are re-executed. These events bring databases up to date from the point of the backup. See Section 6.2.2, "Using Backups for Recovery".

However, there are certain binary logging issues that apply with respect to stored routines (procedures and functions) and triggers:

- Logging occurs at the statement level. In some cases, it is possible that a statement will affect different sets of rows on a master and a slave.
- Replicated statements executed on a slave are processed by the slave SQL thread, which has full privileges. It is possible for a procedure to follow different execution paths on master and slave servers, so a user can write a routine containing a dangerous statement that will execute only on the slave where it is processed by a thread that has full privileges.
- If a routine that modifies data is non-deterministic, it is not repeatable. This can result in different data on a master and slave, or cause restored data to differ from the original data.

This section describes how MySQL 5.0 handles binary logging for stored routines and triggers. The discussion first states the current conditions that the implementation places on the use of stored routines, and what you can do to avoid problems. Then it summarizes the changes that have taken place in the logging implementation. Finally, implementation details are given that provide information about when and why various changes were made. These details show how several aspects of the current logging behavior were implemented in response to shortcomings identified in earlier versions of MySQL.

In general, the issues described here occur due to the fact that binary logging occurs at the SQL statement level. MySQL 5.1 implements row-level binary logging, which solves or alleviates these issues because the log contains changes made to individual rows as a result of executing SQL statements.

Unless noted otherwise, the remarks here assume that you have enabled binary logging by starting the server with the `--log-bin` option. (See Section 5.2.3, "The Binary Log".) If the binary log is not enabled, replication is not possible, nor is the binary log available for data recovery.

The current conditions on the use of stored functions in MySQL 5.0 can be summarized as follows. These conditions do not apply to stored procedures and they do not apply unless binary logging is enabled.

- To create or alter a stored function, you must have the `SUPER` privilege, in addition to the `CREATE ROUTINE` or `ALTER ROUTINE` privilege that is normally required.
- When you create a stored function, you must declare either that it is deterministic or that it does not modify data. Otherwise, it may be unsafe for data recovery or replication.

By default, for a `CREATE FUNCTION` statement to be accepted, at least one of `DETERMINISTIC`, `NO SQL`, or `READS SQL DATA` must be specified explicitly. Otherwise an error occurs:

```
ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL,
or READS SQL DATA in its declaration and binary logging is enabled
(you *might* want to use the less safe log_bin_trust_function_creators
variable)
```

This function is deterministic (and does not modify data), so it is safe:

```
CREATE FUNCTION f1(i INT)
RETURNS INT
DETERMINISTIC
READS SQL DATA
BEGIN
  RETURN i;
END;
```

This function uses `UUID()`, which is not deterministic, so the function also is not deterministic and is not safe:

```
CREATE FUNCTION f2()
RETURNS CHAR(36) CHARACTER SET utf8
BEGIN
  RETURN UUID();
END;
```

This function modifies data, so it may not be safe:

```
CREATE FUNCTION f3(p_id INT)
RETURNS INT
BEGIN
  UPDATE t SET modtime = NOW() WHERE id = p_id;
  RETURN ROW_COUNT();
END;
```

Assessment of the nature of a function is based on the "honesty" of the creator: MySQL does not check that a function declared `DETERMINISTIC` is free of statements that produce non-deterministic results.

- To relax the preceding conditions on function creation (that you must have the `SUPER` privilege and that a function must be declared deterministic or to not modify data), set the global `log_bin_trust_function_creators` system variable to 1. By default, this variable has a value of 0, but you can change it like this:
- `mysql> **SET GLOBAL log_bin_trust_function_creators = 1;**`

You can also set this variable by using the `--log-bin-trust-function-creators=1` option when starting the server.

If binary logging is not enabled, `log_bin_trust_function_creators` does not apply and `SUPER` is not required for routine creation.

For information about built-in functions that may be unsafe for replication (and thus cause stored functions that use them to be unsafe as well), see Section 18.3.1, "Replication Features and Issues".

Triggers are similar to stored functions, so the preceding remarks regarding functions also apply to triggers with the following exception: `CREATE TRIGGER` does not have an optional `DETERMINISTIC` characteristic, so triggers are assumed to be always deterministic. However, this assumption might in some cases be invalid. For example, the `UUID()` function is non-deterministic (and does not replicate). You should be careful about using such functions in triggers.

Triggers can update tables, so error messages similar to those for stored functions occur with `CREATE TRIGGER` if you do not have the required privileges. On the slave side, the slave uses the trigger `DEFINER` attribute to determine which user is considered to be the creator of the trigger.

The rest of this section provides details on the development of stored routine logging. You need not read it unless you are interested in the background on the rationale for the current logging-related conditions on stored routine use.

The development of stored routine logging in MySQL 5.0 can be summarized as follows:

- Before MySQL 5.0.6: In the initial implementation of stored routine logging, statements that create stored routines and `CALL` statements are not logged. These omissions can cause problems for replication and data recovery.
- MySQL 5.0.6: Statements that create stored routines and `CALL` statements are logged. Stored function invocations are logged when they occur in statements that update data (because those statements are logged). However, function invocations are not logged when they occur in statements such as `SELECT` that do not change data, even if a data change occurs within a function itself; this can cause problems. Under some circumstances, functions and procedures can have different effects if executed at different times or on different (master and slave) machines, and thus can be unsafe for data recovery or replication. To handle this, measures are implemented to allow identification of safe routines and to prevent creation of unsafe routines except by users with sufficient privileges.
- MySQL 5.0.12: For stored functions, when a function invocation that changes data occurs within a non-logged statement such as `SELECT`, the server logs a `DO` *`func_name`*`()` statement that invokes the function so that the function gets executed during data recovery or replication to slave servers. For stored procedures, the server does not log `CALL` statements. Instead, it logs individual statements within a procedure that are executed as a result of a `CALL`. This eliminates problems that may occur when a procedure would follow a different execution path on a slave than on the master.
- MySQL 5.0.16: The procedure logging changes made in 5.0.12 allow the conditions on unsafe routines to be relaxed for stored procedures. Consequently, the user interface for

controlling these conditions is revised to apply only to functions. Procedure creators are no longer bound by them.

- MySQL 5.0.17: Logging of stored functions as `DO func_name()` statements (per the changes made in 5.0.12) are logged as `SELECT func_name()` statements instead for better control over error checking.

Routine logging before MySQL 5.0.6: Statements that create and use stored routines are not written to the binary log, but statements invoked within stored routines are logged. Suppose that you issue the following statements:

```
CREATE PROCEDURE mysp INSERT INTO t VALUES(1);
CALL mysp();
```

For this example, only the `INSERT` statement appears in the binary log. The `CREATE PROCEDURE` and `CALL` statements do not appear. The absence of routine-related statements in the binary log means that stored routines are not replicated correctly. It also means that for a data recovery operation, re-executing events in the binary log does not recover stored routines.

Routine logging changes in MySQL 5.0.6: To address the absence of logging for stored routine creation and `CALL` statements (and the consequent replication and data recovery concerns), the characteristics of binary logging for stored routines were changed as described here. (Some of the items in the following list point out issues that are dealt with in later versions.)

- The server writes `CREATE PROCEDURE`, `CREATE FUNCTION`, `ALTER PROCEDURE`, `ALTER FUNCTION`, `DROP PROCEDURE`, and `DROP FUNCTION` statements to the binary log. Also, the server logs `CALL` statements, not the statements executed within procedures. Suppose that you issue the following statements:
- `CREATE PROCEDURE mysp INSERT INTO t VALUES(1);`
- `CALL mysp();`

  For this example, the `CREATE PROCEDURE` and `CALL` statements appear in the binary log, but the `INSERT` statement does not appear. This corrects the problem that occurred before MySQL 5.0.6 such that only the `INSERT` was logged.

- Logging `CALL` statements has a security implication for replication, which arises from two factors:
  - Statements executed on a slave are processed by the slave SQL thread which has full privileges.
  - It is possible for a procedure to follow different execution paths on master and slave servers.

  The implication is that although a user must have the `CREATE ROUTINE` privilege to create a routine, the user can write a routine containing a dangerous statement that will execute only on the slave where it is processed by a thread that has full privileges. For example, if the master and slave servers have server ID values of 1 and 2, respectively, a user on the master server could create and invoke an unsafe procedure `unsafe_sp()` as follows:

```
mysql> delimiter //
mysql> CREATE PROCEDURE unsafe_sp ()
    -> BEGIN
    ->   IF @@server_id=2 THEN DROP DATABASE accounting; END IF;
    -> END;
    -> //
mysql> delimiter ;
mysql> CALL unsafe_sp();
```

The `CREATE PROCEDURE` and `CALL` statements are written to the binary log, so the slave will execute them. Because the slave SQL thread has full privileges, it will execute the `DROP DATABASE` statement that drops the `accounting` database. Thus, the `CALL` statement has different effects on the master and slave and is not replication-safe.

The preceding example uses a stored procedure, but similar problems can occur for stored functions that are invoked within statements that are written to the binary log: Function invocation has different effects on the master and slave.

To guard against this danger for servers that have binary logging enabled, MySQL 5.0.6 introduces the requirement that stored procedure and function creators must have the `SUPER` privilege, in addition to the usual `CREATE ROUTINE` privilege that is required. Similarly, to use `ALTER PROCEDURE` or `ALTER FUNCTION`, you must have the `SUPER` privilege in addition to the `ALTER ROUTINE` privilege. Without the `SUPER` privilege, an error will occur:

```
ERROR 1419 (HY000): You do not have the SUPER privilege and
binary logging is enabled (you *might* want to use the less safe
log_bin_trust_routine_creators variable)
```

If you do not want to require routine creators to have the `SUPER` privilege (for example, if all users with the `CREATE ROUTINE` privilege on your system are experienced application developers), set the global `log_bin_trust_routine_creators` system variable to 1. You can also set this variable by using the `--log-bin-trust-routine-creators=1` option when starting the server. If binary logging is not enabled, `log_bin_trust_routine_creators` does not apply and `SUPER` is not required for routine creation.

- If a routine that performs updates is non-deterministic, it is not repeatable. This can have two undesirable effects:
  - It will make a slave different from the master.
  - Restored data will be different from the original data.

  To deal with these problems, MySQL enforces the following requirement: On a master server, creation and alteration of a routine is refused unless you declare the routine to be deterministic or to not modify data. Two sets of routine characteristics apply here:

  - The `DETERMINISTIC` and `NOT DETERMINISTIC` characteristics indicate whether a routine always produces the same result for given inputs. The default is `NOT`

DETERMINISTIC if neither characteristic is given. To declare that a routine is deterministic, you must specify DETERMINISTIC explicitly.

- The CONTAINS SQL, NO SQL, READS SQL DATA, and MODIFIES SQL DATA characteristics provide information about whether the routine reads or writes data. Either NO SQL or READS SQL DATA indicates that a routine does not change data, but you must specify one of these explicitly because the default is CONTAINS SQL if no characteristic is given.

By default, for a CREATE PROCEDURE or CREATE FUNCTION statement to be accepted, at least one of DETERMINISTIC, NO SQL, or READS SQL DATA must be specified explicitly. Otherwise an error occurs:

```
ERROR 1418 (HY000): This routine has none of DETERMINISTIC, NO SQL,
or READS SQL DATA in its declaration and binary logging is enabled
(you *might* want to use the less safe log_bin_trust_routine_creators
variable)
```

If you set log_bin_trust_routine_creators to 1, the requirement that routines be deterministic or not modify data is dropped.

- A CALL statement is written to the binary log if the routine returns no error, but not otherwise. When a routine that modifies data fails, you get this warning:

```
ERROR 1417 (HY000): A routine failed and has neither NO SQL nor
READS SQL DATA in its declaration and binary logging is enabled; if
non-transactional tables were updated, the binary log will miss their
changes
```

This logging behavior has the potential to cause problems. If a routine partly modifies a non-transactional table (such as a MyISAM table) and returns an error, the binary log will not reflect these changes. To protect against this, you should use transactional tables in the routine and modify the tables within transactions.

If you use the IGNORE keyword with INSERT, DELETE, or UPDATE to ignore errors within a routine, a partial update might occur but no error will result. Such statements are logged and they replicate normally.

- Although statements normally are not written to the binary log if they are rolled back, CALL statements are logged even when they occur within a rolled-back transaction. This can result in a CALL being rolled back on the master but executed on slaves.
- If a stored function is invoked within a statement such as SELECT that does not modify data, execution of the function is not written to the binary log, even if the function itself modifies data. This logging behavior has the potential to cause problems. Suppose that a function myfunc() is defined as follows:

```
CREATE FUNCTION myfunc () RETURNS INT DETERMINISTIC
BEGIN
  INSERT INTO t (i) VALUES(1);
  RETURN 0;
```

- `END;`

  Given that definition, the following statement is not written to the binary log because it is a `SELECT`. Nevertheless, it modifies the table `t` because `myfunc()` modifies `t`:

  ```
  SELECT myfunc();
  ```

  A workaround for this problem is to invoke functions that do updates only within statements that do updates (and which therefore are written to the binary log). Note that although the `DO` statement sometimes is executed for the side effect of evaluating an expression, `DO` is not a workaround here because it is not written to the binary log.

- On slave servers, `--replicate-*-table` rules do not apply to `CALL` statements or to statements within stored routines. These statements are always replicated. If such statements contain references to tables that do not exist on the slave, they could have undesirable effects when executed on the slave.

Routine logging changes in MySQL 5.0.12: The changes in 5.0.12 address several problems that were present in earlier versions:

- Stored function invocations in non-logged statements such as `SELECT` were not being logged, even when a function itself changed data.
- Stored procedure logging at the `CALL` level could cause different effects on a master and slave if a procedure took different execution paths on the two machines.
- `CALL` statements were logged even when they occurred within a rolled-back transaction.

To deal with these issues, MySQL 5.0.12 implements the following changes to function and procedure logging:

- A stored function invocation is logged as a `DO` statement if the function changes data and occurs within a statement that would not otherwise be logged. This corrects the problem of non-replication of data changes that result from use of stored functions in non-logged statements. For example, `SELECT` statements are not written to the binary log, but a `SELECT` might invoke a stored function that makes changes. To handle this, a `DO` *`func_name`*`()` statement is written to the binary log when the given function makes a change. Suppose that the following statements are executed on the master:

```
CREATE FUNCTION f1(a INT) RETURNS INT
BEGIN
  IF (a < 3) THEN
    INSERT INTO t2 VALUES (a);
  END IF;
  RETURN 0;
END;

CREATE TABLE t1 (a INT);
INSERT INTO t1 VALUES (1),(2),(3);
```

- `SELECT f1(a) FROM t1;`

  When the `SELECT` statement executes, the function `f1()` is invoked three times. Two of those invocations insert a row, and MySQL logs a `DO` statement for each of them. That is, MySQL writes the following statements to the binary log:

  ```
  DO f1(1);
  DO f1(2);
  ```

  The server also logs a `DO` statement for a stored function invocation when the function invokes a stored procedure that causes an error. In this case, the server writes the `DO` statement to the log along with the expected error code. On the slave, if the same error occurs, that is the expected result and replication continues. Otherwise, replication stops.

  Note: See later in this section for changes made in MySQL 5.0.19: These logged `DO` *`func_name`*`()` statements are logged as `SELECT` *`func_name`*`()` statements instead.

- Stored procedure calls are logged at the statement level rather than at the `CALL` level. That is, the server does not log the `CALL` statement, it logs those statements within the procedure that actually execute. As a result, the same changes that occur on the master will be observed on slave servers. This eliminates the problems that could result from a procedure having different execution paths on different machines. For example, the `DROP DATABASE` problem shown earlier for the `unsafe_sp()` procedure does not occur and the routine is no longer replication-unsafe because it has the same effect on master and slave servers.

  In general, statements executed within a stored procedure are written to the binary log using the same rules that would apply were the statements to be executed in standalone fashion. Some special care is taken when logging procedure statements because statement execution within procedures is not quite the same as in non-procedure context:

  - A statement to be logged might contain references to local procedure variables. These variables do not exist outside of stored procedure context, so a statement that refers to such a variable cannot be logged literally. Instead, each reference to a local variable is replaced by this construct for logging purposes:
  - `NAME_CONST(`*`var_name`*`, `*`var_value`*`)`

    `var_name` is the local variable name, and *`var_value`* is a constant indicating the value that the variable has at the time the statement is logged. `NAME_CONST()` has a value of *`var_value`*, and a "name" of *`var_name`*. Thus, if you invoke this function directly, you get a result like this:

    ```
    mysql> SELECT NAME_CONST('myname', 14);
    +--------+
    | myname |
    +--------+
    |     14 |
    +--------+
    ```

NAME_CONST() allows a logged standalone statement to be executed on a slave with the same effect as the original statement that was executed on the master within a stored procedure.

- A statement to be logged might contain references to user-defined variables. To handle this, MySQL writes a SET statement to the binary log to make sure that the variable exists on the slave with the same value as on the master. For example, if a statement refers to a variable @my_var, that statement will be preceded in the binary log by the following statement, where *value* is the value of @my_var on the master:
- SET @my_var = *value*;
- Procedure calls can occur within a committed or rolled-back transaction. Previously, CALL statements were logged even if they occurred within a rolled-back transaction. As of MySQL 5.0.12, transactional context is accounted for so that the transactional aspects of procedure execution are replicated correctly. That is, the server logs those statements within the procedure that actually execute and modify data, and also logs BEGIN, COMMIT, and ROLLBACK statements as necessary. For example, if a procedure updates only transactional tables and is executed within a transaction that is rolled back, those updates are not logged. If the procedure occurs within a committed transaction, BEGIN and COMMIT statements are logged with the updates. For a procedure that executes within a rolled-back transaction, its statements are logged using the same rules that would apply if the statements were executed in standalone fashion:
  - Updates to transactional tables are not logged.
  - Updates to non-transactional tables are logged because rollback does not cancel them.
  - Updates to a mix of transactional and non-transactional tables are logged surrounded by BEGIN and ROLLBACK so that slaves will make the same changes and rollbacks as on the master.

- A stored procedure call is *not* written to the binary log at the statement level if the procedure is invoked from within a stored function. In that case, the only thing logged is the statement that invokes the function (if it occurs within a statement that is logged) or a DO statement (if it occurs within a statement that is not logged). For this reason, care still should be exercised in the use of stored functions that invoke a procedure, even if the procedure is otherwise safe in itself.
- Because procedure logging occurs at the statement level rather than at the CALL level, interpretation of the --replicate-*-table options is revised to apply only to stored functions. They no longer apply to stored procedures, except those procedures that are invoked from within functions.

Routine logging changes in MySQL 5.0.16: In 5.0.12, a change was introduced to log stored procedure calls at the statement level rather than at the CALL level. This change eliminates the requirement that procedures be identified as safe. The requirement now exists only for stored functions, because they still appear in the binary log as function invocations rather than as the statements executed within the function. To reflect the lifting of the restriction on stored procedures, the log_bin_trust_routine_creators system variable is renamed to

`log_bin_trust_function_creators` and the `--log-bin-trust-routine-creators` server option is renamed to `--log-bin-trust-function-creators`. (For backward compatibility, the old names are recognized but result in a warning.) Error messages that now apply only to functions and not to routines in general are re-worded.

Routine logging changes in MySQL 5.0.19: In 5.0.12, a change was introduced to log a stored function invocation as `DO func_name()` if the invocation changes data and occurs within a non-logged statement, or if the function invokes a stored procedure that produces an error. In 5.0.19, these invocations are logged as `SELECT func_name()` instead. The change to `SELECT` was made because use of `DO` was found to yield insufficient control over error code checking.
