STORED OBJECT

[Compound Statements 3](#_Toc57736881)

[BEGIN ... END Compound Statement 3](#_Toc57736882)

[Statement Labels 3](#_Toc57736883)

[DECLARE Statement 4](#_Toc57736884)

[Variables in Stored Programs 4](#_Toc57736885)

[Local Variable DECLARE Statement 5](#_Toc57736886)

[Local Variable Scope and Resolution 5](#_Toc57736887)

[Flow Control Statements 6](#_Toc57736888)

[CASE Statement 6](#_Toc57736889)

[IF Statement 7](#_Toc57736890)

[ITERATE Statement 8](#_Toc57736891)

[13.6.5.4 LEAVE Statement 8](#_Toc57736892)

[LOOP Statement 8](#_Toc57736893)

[REPEAT Statement 8](#_Toc57736894)

[RETURN Statement 9](#_Toc57736895)

[WHILE Statement 9](#_Toc57736896)

[Condition Handling 10](#_Toc57736897)

[DECLARE ... CONDITION Statement 10](#_Toc57736898)

[DECLARE ... HANDLER Statement 11](#_Toc57736899)

[GET DIAGNOSTICS Statement 14](#_Toc57736900)

[RESIGNAL Statement 17](#_Toc57736901)

[**RESIGNAL Overview** 17](#_Toc57736902)

[**RESIGNAL with New Signal Information** 18](#_Toc57736903)

[**RESIGNAL with a Condition Value and Optional New Signal Information** 19](#_Toc57736904)

[**RESIGNAL Requires Condition Handler Context** 20](#_Toc57736905)

[SIGNAL Statement 20](#_Toc57736906)

[**SIGNAL Overview** 21](#_Toc57736907)

[Signal Condition Information Items 23](#_Toc57736908)

[**Effect of Signals on Handlers, Cursors, and Statements** 24](#_Toc57736909)

[Scope Rules for Handlers 24](#_Toc57736910)

[The MySQL Diagnostics Area 27](#_Toc57736911)

[**Diagnostics Area Structure** 27](#_Toc57736912)

[**Diagnostics Area Information Items** 27](#_Toc57736913)

[How the Diagnostics Area is Cleared and Populated 28](#_Toc57736914)

[**Diagnostics Area-Related System Variables** 29](#_Toc57736915)

[Condition Handling and OUT or INOUT Parameters 30](#_Toc57736916)

[Restrictions on Condition Handling 30](#_Toc57736917)

[Stored Objects 31](#_Toc57736918)

[Defining Stored Programs 31](#_Toc57736919)

[Using Stored Routines 32](#_Toc57736920)

[25.2.1 Stored Routine Syntax 33](#_Toc57736921)

[Stored Routines and MySQL Privileges 33](#_Toc57736922)

[25.2.4 Stored Procedures, Functions, Triggers, and LAST\_INSERT\_ID() 34](#_Toc57736923)

[Using Triggers 34](#_Toc57736924)

[Trigger Syntax and Examples 34](#_Toc57736925)

[Using the Event Scheduler 38](#_Toc57736926)

[Event Scheduler Overview 38](#_Toc57736927)

[Event Scheduler Configuration 39](#_Toc57736928)

[Event Syntax 41](#_Toc57736929)

[Event Metadata 41](#_Toc57736930)

[Event Scheduler Status 41](#_Toc57736931)

[The Event Scheduler and MySQL Privileges 42](#_Toc57736932)

[Using Views 44](#_Toc57736933)

[View Syntax 44](#_Toc57736934)

[View Processing Algorithms 45](#_Toc57736935)

[Updatable and Insertable Views 46](#_Toc57736936)

[The View WITH CHECK OPTION Clause 48](#_Toc57736937)

[Stored Object Access Control 48](#_Toc57736938)

[The DEFINER Attribute 49](#_Toc57736939)

[The SQL SECURITY Characteristic 49](#_Toc57736940)

[Examples 49](#_Toc57736941)

[Orphan Stored Objects 50](#_Toc57736942)

[Risk-Minimization Guidelines 51](#_Toc57736943)

[Stored Program Binary Logging 52](#_Toc57736944)

[Restrictions on Stored Programs 57](#_Toc57736945)

[SQL Statements Not Permitted in Stored Routines 57](#_Toc57736946)

[Restrictions for Stored Functions 58](#_Toc57736947)

[Restrictions for Triggers 58](#_Toc57736948)

[Name Conflicts within Stored Routines 58](#_Toc57736949)

[Replication Considerations 59](#_Toc57736950)

[Debugging Considerations 59](#_Toc57736951)

[Unsupported Syntax from the SQL:2003 Standard 59](#_Toc57736952)

[Stored Routine Concurrency Considerations 59](#_Toc57736953)

[Event Scheduler Restrictions 59](#_Toc57736954)

[Stored routines and triggers in NDB Cluster 60](#_Toc57736955)

[Restrictions on Views 60](#_Toc57736956)

## Compound Statements

This section describes the syntax for the [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) compound statement and other statements that can be used in the body of stored programs: Stored procedures and functions, triggers, and events. These objects are defined in terms of SQL code that is stored on the server for later invocation.

A compound statement is a block that can contain other blocks; declarations for variables, condition handlers, and cursors; and flow control constructs such as loops and conditional tests.

### BEGIN ... END Compound Statement

[begin\_label:] BEGIN

[statement\_list]

END [end\_label]

[BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) syntax is used for writing compound statements, which can appear within stored programs (stored procedures and functions, triggers, and events). 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 terminated by a semicolon (;) statement delimiter. The ***statement\_list*** itself is optional, so the empty compound statement (BEGIN END) is legal.

[BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) blocks can be nested.

Use of multiple statements requires that a client is able to send statement strings containing the ; statement delimiter. In the [**mysql**](https://dev.mysql.com/doc/refman/5.6/en/mysql.html) command-line client, this is handled with the delimiter command. Changing the ; end-of-statement delimiter (for example, to //) permit ; to be used in a program body. For an example, see [Section 20.1, “Defining Stored Programs”](https://dev.mysql.com/doc/refman/5.6/en/stored-programs-defining.html).

A [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block can be labeled. See [Section 13.6.2, “Statement Labels”](https://dev.mysql.com/doc/refman/5.6/en/statement-labels.html).

The optional [NOT] ATOMIC clause is not 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.

**Note**

Within all stored programs, the parser treats [BEGIN [WORK]](https://dev.mysql.com/doc/refman/5.6/en/commit.html) as the beginning of a [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block. To begin a transaction in this context, use [START TRANSACTION](https://dev.mysql.com/doc/refman/5.6/en/commit.html) instead.

### Statement Labels

[begin\_label:] BEGIN

[statement\_list]

END [end\_label]

[begin\_label:] LOOP

statement\_list

END LOOP [end\_label]

[begin\_label:] REPEAT

statement\_list

UNTIL search\_condition

END REPEAT [end\_label]

[begin\_label:] WHILE search\_condition DO

statement\_list

END WHILE [end\_label]

Labels are permitted for [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) blocks and for the [LOOP](https://dev.mysql.com/doc/refman/5.6/en/loop.html), [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html), and [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html) statements. Label use for those statements follows these rules:

* ***begin\_label*** must be followed by a colon.
* ***begin\_label*** can be given without ***end\_label***. If ***end\_label*** is present, it must be the same as ***begin\_label***.
* ***end\_label*** cannot be given without ***begin\_label***.
* Labels at the same nesting level must be distinct.
* Labels can be up to 16 characters long.

To refer to a label within the labeled construct, use an [ITERATE](https://dev.mysql.com/doc/refman/5.6/en/iterate.html) or [LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) statement. The following example uses those statements to continue iterating or terminate the loop:

CREATE PROCEDURE doiterate(p1 INT)

BEGIN

label1: LOOP

SET p1 = p1 + 1;

IF p1 < 10 THEN ITERATE label1; END IF;

LEAVE label1;

END LOOP label1;

END;

The scope of a block label does not include the code for handlers declared within the block. For details, see [Section 13.6.7.2, “DECLARE ... HANDLER Statement”](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html).

### DECLARE Statement

The [DECLARE](https://dev.mysql.com/doc/refman/5.6/en/declare.html) statement is used to define various items local to a program:

* Local variables.
* Conditions and handlers.
* Cursors.

[DECLARE](https://dev.mysql.com/doc/refman/5.6/en/declare.html) is permitted only inside a [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) compound statement and must be at its start, before any other statements.

Declarations must follow a certain order. Cursor declarations must appear before handler declarations. Variable and condition declarations must appear before cursor or handler declarations.

### Variables in Stored Programs

System variables and user-defined variables can be used in stored programs, just as they can be used outside stored-program context. In addition, stored programs can use DECLARE to define local variables, and stored routines (procedures and functions) can be declared to take parameters that communicate values between the routine and its caller.

* To declare local variables, use the [DECLARE](https://dev.mysql.com/doc/refman/5.6/en/declare-local-variable.html) statement, as described in [Section 13.6.4.1, “Local Variable DECLARE Statement”](https://dev.mysql.com/doc/refman/5.6/en/declare-local-variable.html).
* Variables can be set directly with the [SET](https://dev.mysql.com/doc/refman/5.6/en/set-variable.html) statement. See [Section 13.7.4.1, “SET Syntax for Variable Assignment”](https://dev.mysql.com/doc/refman/5.6/en/set-variable.html).
* Results from queries can be retrieved into local variables using [SELECT ... INTO ***var\_list***](https://dev.mysql.com/doc/refman/5.6/en/select-into.html) or by opening a cursor and using [FETCH ... INTO ***var\_list***](https://dev.mysql.com/doc/refman/5.6/en/fetch.html). See [Section 13.2.9.1, “SELECT ... INTO Statement”](https://dev.mysql.com/doc/refman/5.6/en/select-into.html), and [Section 13.6.6, “Cursors”](https://dev.mysql.com/doc/refman/5.6/en/cursors.html).

For information about the scope of local variables and how MySQL resolves ambiguous names, see [Section 13.6.4.2, “Local Variable Scope and Resolution”](https://dev.mysql.com/doc/refman/5.6/en/local-variable-scope.html).

It is not permitted to assign the value DEFAULT to stored procedure or function parameters or stored program local variables (for example with a SET ***var\_name*** = DEFAULT statement). As of MySQL 5.6.6, this results in a syntax error.

#### Local Variable DECLARE Statement

DECLARE var\_name [, var\_name] ... type [DEFAULT value]

This statement declares local variables within stored programs. To provide a default value for a 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 stored routine parameters with respect to data type and overflow checking.

Variable declarations must appear before cursor or handler declarations.

Local variable names are not case-sensitive. Permissible characters and quoting rules are the same as for other identifiers.

The scope of a local variable is the [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block within which 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.

#### Local Variable Scope and Resolution

The scope of a local variable is the [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block within which 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.

Because local variables are in scope only during stored program execution, references to them are not permitted in prepared statements created within a stored program. Prepared statement scope is the current session, not the stored program, so the statement could be executed after the program ends, at which point the variables would no longer be in scope. For example, SELECT ... INTO ***local\_var*** cannot be used as a prepared statement. This restriction also applies to stored procedure and function parameters.

A local variable should not have the same name as a table column. If an SQL statement, such as a [SELECT ... INTO](https://dev.mysql.com/doc/refman/5.6/en/select.html) 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. Consider the following procedure definition:

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;

MySQL interprets xname in the [SELECT](https://dev.mysql.com/doc/refman/5.6/en/select.html) statement as a reference to the xname variable rather than the xname column. Consequently, when the procedure sp1()is called, the newname variable returns the value 'bob' regardless of the value of the table1.xname column.

Similarly, the cursor definition in the following procedure contains a [SELECT](https://dev.mysql.com/doc/refman/5.6/en/select.html) statement that refers to xname. MySQL interprets this as a reference to the variable of that name rather than a column reference.

CREATE PROCEDURE sp2 (x VARCHAR(5))

BEGIN

DECLARE xname VARCHAR(5) DEFAULT 'bob';

DECLARE newname VARCHAR(5);

DECLARE xid INT;

DECLARE done TINYINT DEFAULT 0;

DECLARE cur1 CURSOR FOR SELECT xname, id FROM table1;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN cur1;

read\_loop: LOOP

FETCH FROM cur1 INTO newname, xid;

IF done THEN LEAVE read\_loop; END IF;

SELECT newname;

END LOOP;

CLOSE cur1;

END;

### Flow Control Statements

MySQL supports the [IF](https://dev.mysql.com/doc/refman/5.6/en/if.html), [CASE](https://dev.mysql.com/doc/refman/5.6/en/case.html), [ITERATE](https://dev.mysql.com/doc/refman/5.6/en/iterate.html), [LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) [LOOP](https://dev.mysql.com/doc/refman/5.6/en/loop.html), [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html), and [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html) constructs for flow control within stored programs. It also supports [RETURN](https://dev.mysql.com/doc/refman/5.6/en/return.html) within stored functions.

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](https://dev.mysql.com/doc/refman/5.6/en/if.html) statement might contain a [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html) loop, which itself contains a [CASE](https://dev.mysql.com/doc/refman/5.6/en/case.html) statement.

MySQL does not support FOR loops.

#### 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](https://dev.mysql.com/doc/refman/5.6/en/case.html) statement for stored programs implements a complex conditional construct.

**Note**

There is also a [CASE](https://dev.mysql.com/doc/refman/5.6/en/flow-control-functions.html#operator_case) operator, which differs from the [CASE](https://dev.mysql.com/doc/refman/5.6/en/case.html) statement described here. The [CASE](https://dev.mysql.com/doc/refman/5.6/en/case.html) statement cannot have an ELSE NULL clause, and it is terminated with END CASE instead of END.

For the first syntax, ***case\_value*** is an expression. This value is compared to the ***when\_value*** expression in each WHEN clause until one of them is equal. When an equal ***when\_value*** is found, the corresponding THEN clause ***statement\_list*** executes. If no ***when\_value*** is equal, the ELSE clause ***statement\_list*** executes, if there is one.

This syntax cannot be used to test for equality with NULL because NULL = NULL is false.

For the second syntax, each WHEN clause ***search\_condition*** expression is evaluated until one is true, at which point its corresponding THEN clause ***statement\_list*** executes. If no ***search\_condition*** is equal, the ELSE clause ***statement\_list*** executes, if there is one.

If no ***when\_value*** or ***search\_condition*** matches the value tested and the [CASE](https://dev.mysql.com/doc/refman/5.6/en/case.html) statement contains no ELSE clause, a Case not found for CASE statement error results.

Each ***statement\_list*** consists of one or more SQL statements; an empty ***statement\_list*** is not permitted.

To handle situations where no value is matched by any WHEN clause, use an ELSE containing an empty [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block, as shown in this example. (The indentation used here in the ELSE clause is for purposes of clarity only, and is not otherwise significant.)

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;

#### IF Statement

IF search\_condition THEN statement\_list

[ELSEIF search\_condition THEN statement\_list] ...

[ELSE statement\_list]

END IF

The [IF](https://dev.mysql.com/doc/refman/5.6/en/if.html) statement for stored programs implements a basic conditional construct.

**Note**

There is also an [IF()](https://dev.mysql.com/doc/refman/5.6/en/flow-control-functions.html#function_if) function, which differs from the [IF](https://dev.mysql.com/doc/refman/5.6/en/if.html) statement described here. See [Section 12.5, “Flow Control Functions”](https://dev.mysql.com/doc/refman/5.6/en/flow-control-functions.html). The [IF](https://dev.mysql.com/doc/refman/5.6/en/if.html) statement can have THEN, ELSE, and ELSEIF clauses, and it is terminated with END IF.

If a given ***search\_condition*** evaluates to true, the corresponding THEN or ELSEIF clause ***statement\_list*** executes. If no ***search\_condition*** matches, the ELSE clause ***statement\_list*** executes.

Each ***statement\_list*** consists of one or more SQL statements; an empty ***statement\_list*** is not permitted.

An IF ... END IF block, like all other flow-control blocks used within stored programs, 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](https://dev.mysql.com/doc/refman/5.6/en/if.html) statements. Each [IF](https://dev.mysql.com/doc/refman/5.6/en/if.html) 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](https://dev.mysql.com/doc/refman/5.6/en/if.html) is evaluated only if n is not equal to m.

#### ITERATE Statement

ITERATE label

[ITERATE](https://dev.mysql.com/doc/refman/5.6/en/iterate.html) can appear only within [LOOP](https://dev.mysql.com/doc/refman/5.6/en/loop.html), [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html), and [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html) statements. [ITERATE](https://dev.mysql.com/doc/refman/5.6/en/iterate.html) means “start the loop again.”

#### 13.6.5.4 LEAVE Statement

LEAVE label

This statement is used to exit the flow control construct that has the given label. If the label is for the outermost stored program block, [LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) exits the program.

[LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) can be used within [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) or loop constructs ([LOOP](https://dev.mysql.com/doc/refman/5.6/en/loop.html), [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html), [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html)).

For an example, see [Section 13.6.5.5, “LOOP Statement”](https://dev.mysql.com/doc/refman/5.6/en/loop.html).

#### LOOP Statement

[begin\_label:] LOOP

statement\_list

END LOOP [end\_label]

[LOOP](https://dev.mysql.com/doc/refman/5.6/en/loop.html) implements a simple loop construct, enabling repeated execution of the statement list, which consists of one or more statements, each terminated by a semicolon (;) statement delimiter. The statements within the loop are repeated until the loop is terminated. Usually, this is accomplished with a [LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) statement. Within a stored function, [RETURN](https://dev.mysql.com/doc/refman/5.6/en/return.html) can also be used, which exits the function entirely.

Neglecting to include a loop-termination statement results in an infinite loop.

A [LOOP](https://dev.mysql.com/doc/refman/5.6/en/loop.html) statement can be labeled. For the rules regarding label use, see [Section 13.6.2, “Statement Labels”](https://dev.mysql.com/doc/refman/5.6/en/statement-labels.html).

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;

#### REPEAT Statement

[begin\_label:] REPEAT

statement\_list

UNTIL search\_condition

END REPEAT [end\_label]

The statement list within a [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html) statement is repeated until the ***search\_condition*** expression is true. Thus, a [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html) always enters the loop at least once. ***statement\_list*** consists of one or more statements, each terminated by a semicolon (;) statement delimiter.

A [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html) statement can be labeled..

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)

#### RETURN Statement

RETURN expr

The [RETURN](https://dev.mysql.com/doc/refman/5.6/en/return.html) statement terminates execution of a stored function and returns the value ***expr*** to the function caller. There must be at least one [RETURN](https://dev.mysql.com/doc/refman/5.6/en/return.html) 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, triggers, or events. The [LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) statement can be used to exit a stored program of those types.

#### WHILE Statement

[begin\_label:] WHILE search\_condition DO

statement\_list

END WHILE [end\_label]

The statement list within a [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html) statement is repeated as long as the ***search\_condition*** expression is true. ***statement\_list*** consists of one or more SQL statements, each terminated by a semicolon (;) statement delimiter.

A [WHILE](https://dev.mysql.com/doc/refman/5.6/en/while.html) statement can be labeled. For the rules regarding label use, see [Section 13.6.2, “Statement Labels”](https://dev.mysql.com/doc/refman/5.6/en/statement-labels.html).

Example:

CREATE PROCEDURE dowhile()

BEGIN

DECLARE v1 INT DEFAULT 5;

WHILE v1 > 0 DO

...

SET v1 = v1 - 1;

END WHILE;

END;

### Condition Handling

Conditions may arise during stored program execution that require special handling, such as exiting the current program block or continuing execution. Handlers can be defined for general conditions such as warnings or exceptions, or for specific conditions such as a particular error code. Specific conditions can be assigned names and referred to that way in handlers.

To name a condition, use the [DECLARE ... CONDITION](https://dev.mysql.com/doc/refman/5.6/en/declare-condition.html) statement. To declare a handler, use the [DECLARE ... HANDLER](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html) statement

To raise a condition, use the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement. To modify condition information within a condition handler, use [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html).

To retrieve information from the diagnostics area, use the [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) statement.

#### DECLARE ... CONDITION Statement

DECLARE condition\_name CONDITION FOR condition\_value

condition\_value: {

mysql\_error\_code

| SQLSTATE [VALUE] sqlstate\_value

}

The [DECLARE ... CONDITION](https://dev.mysql.com/doc/refman/5.6/en/declare-condition.html) statement declares a named error condition, associating a name with a condition that needs specific handling. The name can be referred to in a subsequent [DECLARE ... HANDLER](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html) statement.

Condition declarations must appear before cursor or handler declarations.

The ***condition\_value*** for [DECLARE ... CONDITION](https://dev.mysql.com/doc/refman/5.6/en/declare-condition.html) indicates the specific condition or class of conditions to associate with the condition name. It can take the following forms:

* ***mysql\_error\_code***: An integer literal indicating a MySQL error code.

Do not use MySQL error code 0 because that indicates success rather than an error condition. For a list of MySQL error codes, see [Server Error Message Reference](https://dev.mysql.com/doc/mysql-errors/5.6/en/server-error-reference.html).

* SQLSTATE [VALUE] ***sqlstate\_value***: A 5-character string literal indicating an SQLSTATE value.

Do not use SQLSTATE values that begin with '00' because those indicate success rather than an error condition. For a list of SQLSTATE values, see [Server Error Message Reference](https://dev.mysql.com/doc/mysql-errors/5.6/en/server-error-reference.html).

Condition names referred to in [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) or use [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statements must be associated with SQLSTATE values, not MySQL error codes.

Using names for conditions can help make stored program code clearer. For example, this handler applies to attempts to drop a nonexistent table, but that is apparent only if you know that 1051 is the MySQL error code for “unknown table”:

DECLARE CONTINUE HANDLER FOR 1051

BEGIN

-- body of handler

END;

By declaring a name for the condition, the purpose of the handler is more readily seen:

DECLARE no\_such\_table CONDITION FOR 1051;

DECLARE CONTINUE HANDLER FOR no\_such\_table

BEGIN

-- body of handler

END;

Here is a named condition for the same condition, but based on the corresponding SQLSTATE value rather than the MySQL error code:

DECLARE no\_such\_table CONDITION FOR SQLSTATE '42S02';

DECLARE CONTINUE HANDLER FOR no\_such\_table

BEGIN

-- body of handler

END;

#### DECLARE ... HANDLER Statement

DECLARE handler\_action HANDLER

FOR condition\_value [, condition\_value] ...

statement

handler\_action: {

CONTINUE

| EXIT

| UNDO

}

condition\_value: {

mysql\_error\_code

| SQLSTATE [VALUE] sqlstate\_value

| condition\_name

| SQLWARNING

| NOT FOUND

| SQLEXCEPTION

}

The [DECLARE ... HANDLER](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html) statement specifies a handler that deals with one or more conditions. If one of these conditions occurs, the specified ***statement*** executes. ***statement*** can be a simple statement such as SET ***var\_name*** = ***value***, or a compound statement written using BEGIN and END .

Handler declarations must appear after variable or condition declarations.

The ***handler\_action*** value indicates what action the handler takes after execution of the handler statement:

* CONTINUE: Execution of the current program continues.
* EXIT: Execution terminates for the [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) compound statement in which the handler is declared. This is true even if the condition occurs in an inner block.
* UNDO: Not supported.

The ***condition\_value*** for [DECLARE ... HANDLER](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html) indicates the specific condition or class of conditions that activates the handler. It can take the following forms:

* ***mysql\_error\_code***: An integer literal indicating a MySQL error code, such as 1051 to specify “unknown table”:
* DECLARE CONTINUE HANDLER FOR 1051
* BEGIN
* -- body of handler

END;

Do not use MySQL error code 0 because that indicates success rather than an error condition. For a list of MySQL error codes, see [Server Error Message Reference](https://dev.mysql.com/doc/mysql-errors/5.6/en/server-error-reference.html).

* SQLSTATE [VALUE] ***sqlstate\_value***: A 5-character string literal indicating an SQLSTATE value, such as '42S01' to specify “unknown table”:
* DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
* BEGIN
* -- body of handler

END;

Do not use SQLSTATE values that begin with '00' because those indicate success rather than an error condition. For a list of SQLSTATE values, see [Server Error Message Reference](https://dev.mysql.com/doc/mysql-errors/5.6/en/server-error-reference.html).

* ***condition\_name***: A condition name previously specified with [DECLARE ... CONDITION](https://dev.mysql.com/doc/refman/5.6/en/declare-condition.html). A condition name can be associated with a MySQL error code or SQLSTATE value.
* SQLWARNING: Shorthand for the class of SQLSTATE values that begin with '01'.
* DECLARE CONTINUE HANDLER FOR SQLWARNING
* BEGIN
* -- body of handler

END;

* NOT FOUND: Shorthand for the class of SQLSTATE values that begin with '02'. This is relevant within the context of cursors and is used to control what happens when a cursor reaches the end of a data set. 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.
* DECLARE CONTINUE HANDLER FOR NOT FOUND
* BEGIN
* -- body of handler

END;

The NOT FOUND condition also occurs for SELECT ... INTO ***var\_list*** statements that retrieve no rows.

* SQLEXCEPTION: Shorthand for the class of SQLSTATE values that do not begin with '00', '01', or '02'.
* DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
* BEGIN
* -- body of handler

END;

For information about how the server chooses handlers when a condition occurs.

If a condition occurs for which no handler has been declared, the action taken depends on the condition class:

* For SQLEXCEPTION conditions, the stored program terminates at the statement that raised the condition, as if there were an EXIT handler. If the program was called by another stored program, the calling program handles the condition using the handler selection rules applied to its own handlers.
* For SQLWARNING conditions, the program continues executing, as if there were a CONTINUE handler.
* For NOT FOUND conditions, if the condition was raised normally, the action is CONTINUE. If it was raised by [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) or [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html), the action is EXIT.

The following example uses a handler for SQLSTATE '23000', which occurs for a duplicate-key error:

mysql> CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));

Query OK, 0 rows affected (0.00 sec)

mysql> delimiter //

mysql> CREATE PROCEDURE handlerdemo ()

BEGIN

DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;

SET @x = 1;

INSERT INTO test.t VALUES (1);

SET @x = 2;

INSERT INTO test.t VALUES (1);

SET @x = 3;

END;

//

Query OK, 0 rows affected (0.00 sec)

mysql> CALL handlerdemo()//

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//

+------+

| @x |

+------+

| 3 |

+------+

1 row in set (0.00 sec)

Notice that @x is 3 after the procedure executes, which shows that execution continued to the end of the procedure after the error occurred. If the [DECLARE ... HANDLER](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html) statement had not been present, MySQL would have taken the default action (EXIT) after the second [INSERT](https://dev.mysql.com/doc/refman/5.6/en/insert.html) failed due to the PRIMARY KEY constraint, and SELECT @x would have returned 2.

To ignore a condition, declare a CONTINUE handler for it and associate it with an empty block. For example:

DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;

The scope of a block label does not include the code for handlers declared within the block. Therefore, the statement associated with a handler cannot use [ITERATE](https://dev.mysql.com/doc/refman/5.6/en/iterate.html) or [LEAVE](https://dev.mysql.com/doc/refman/5.6/en/leave.html) to refer to labels for blocks that enclose the handler declaration. Consider the following example, where the [REPEAT](https://dev.mysql.com/doc/refman/5.6/en/repeat.html) 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;

IF i < 0 THEN

LEAVE retry; # legal

END IF;

SET i = i - 1;

END;

UNTIL FALSE END REPEAT;

END;

The retry label is in scope for the [IF](https://dev.mysql.com/doc/refman/5.6/en/if.html) 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 references to outer labels in handlers, use one of these strategies:

* To leave the block, use an EXIT handler. If no block cleanup is required, the [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) handler body can be empty:

DECLARE EXIT HANDLER FOR SQLWARNING BEGIN END;

Otherwise, put the cleanup statements in the handler body:

DECLARE EXIT HANDLER FOR SQLWARNING

BEGIN

*block* cleanup statements

END;

* To continue execution, set a status variable in a CONTINUE 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;
* IF done OR i < 0 THEN
* LEAVE retry;
* END IF;
* SET i = i - 1;
* END;
* UNTIL FALSE END REPEAT;

END;

#### GET DIAGNOSTICS Statement

GET [CURRENT] DIAGNOSTICS {

statement\_information\_item

[, statement\_information\_item] ...

| CONDITION condition\_number

condition\_information\_item

[, condition\_information\_item] ...

}

statement\_information\_item:

target = statement\_information\_item\_name

condition\_information\_item:

target = condition\_information\_item\_name

statement\_information\_item\_name: {

NUMBER

| ROW\_COUNT

}

condition\_information\_item\_name: {

CLASS\_ORIGIN

| SUBCLASS\_ORIGIN

| RETURNED\_SQLSTATE

| MESSAGE\_TEXT

| MYSQL\_ERRNO

| CONSTRAINT\_CATALOG

| CONSTRAINT\_SCHEMA

| CONSTRAINT\_NAME

| CATALOG\_NAME

| SCHEMA\_NAME

| TABLE\_NAME

| COLUMN\_NAME

| CURSOR\_NAME

}

condition\_number, target:

(see following discussion)

SQL statements produce diagnostic information that populates the diagnostics area. The [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) statement enables applications to inspect this information. It is available as of MySQL 5.6.4. (You can also use [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html) or [SHOW ERRORS](https://dev.mysql.com/doc/refman/5.6/en/show-errors.html) to see conditions or errors.)

No special privileges are required to execute [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html).

The keyword CURRENT means to retrieve information from the current diagnostics area. In MySQL, it has no effect because that is the default behavior.

[GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) is typically used in a handler within a stored program, but it is a MySQL extension that it is permitted outside handler context to check the execution of any SQL statement. For example, if you invoke the [**mysql**](https://dev.mysql.com/doc/refman/5.6/en/mysql.html) client program, you can enter these statements at the prompt:

mysql> DROP TABLE test.no\_such\_table;

ERROR 1051 (42S02): Unknown table 'test.no\_such\_table'

mysql> GET DIAGNOSTICS CONDITION 1

@p1 = RETURNED\_SQLSTATE, @p2 = MESSAGE\_TEXT;

mysql> SELECT @p1, @p2;

+-------+------------------------------------+

| @p1 | @p2 |

+-------+------------------------------------+

| 42S02 | Unknown table 'test.no\_such\_table' |

+-------+------------------------------------+

Briefly, it contains two kinds of information:

* Statement information, such as the number of conditions that occurred or the affected-rows count.
* Condition information, such as the error code and message. If a statement raises multiple conditions, this part of the diagnostics area has a condition area for each one. If a statement raises no conditions, this part of the diagnostics area is empty.

For a statement that produces three conditions, the diagnostics area contains statement and condition information like this:

Statement information:

row count

... other statement information items ...

Condition area list:

Condition area 1:

error code for condition 1

error message for condition 1

... other condition information items ...

Condition area 2:

error code for condition 2:

error message for condition 2

... other condition information items ...

Condition area 3:

error code for condition 3

error message for condition 3

... other condition information items ...

[GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) can obtain either statement or condition information, but not both in the same statement:

* To obtain statement information, retrieve the desired statement items into target variables. This instance of [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) assigns the number of available conditions and the rows-affected count to the user variables @p1 and @p2:

GET DIAGNOSTICS @p1 = NUMBER, @p2 = ROW\_COUNT;

* To obtain condition information, specify the condition number and retrieve the desired condition items into target variables. This instance of [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) assigns the SQLSTATE value and error message to the user variables @p3 and @p4:
* GET DIAGNOSTICS CONDITION 1

@p3 = RETURNED\_SQLSTATE, @p4 = MESSAGE\_TEXT;

The retrieval list specifies one or more ***target*** = ***item\_name*** assignments, separated by commas. Each assignment names a target variable and either a ***statement\_information\_item\_name*** or ***condition\_information\_item\_name*** designator, depending on whether the statement retrieves statement or condition information.

Valid ***target*** designators for storing item information can be stored procedure or function parameters, stored program local variables declared with [DECLARE](https://dev.mysql.com/doc/refman/5.6/en/declare.html), or user-defined variables.

Valid ***condition\_number*** designators can be stored procedure or function parameters, stored program local variables declared with [DECLARE](https://dev.mysql.com/doc/refman/5.6/en/declare.html), user-defined variables, system variables, or literals. A character literal may include a ***\_charset*** introducer. A warning occurs if the condition number is not in the range from 1 to the number of condition areas that have information. In this case, the warning is added to the diagnostics area without clearing it.

When a condition occurs, MySQL does not populate all condition items recognized by [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html). For example:

mysql> GET DIAGNOSTICS CONDITION 1

@p5 = SCHEMA\_NAME, @p6 = TABLE\_NAME;

mysql> SELECT @p5, @p6;

+------+------+

| @p5 | @p6 |

+------+------+

| | |

+------+------+

In standard SQL, if there are multiple conditions, the first condition relates to the SQLSTATE value returned for the previous SQL statement. In MySQL, this is not guaranteed. To get the main error, you cannot do this:

GET DIAGNOSTICS CONDITION 1 @errno = MYSQL\_ERRNO;

Instead, retrieve the condition count first, then use it to specify which condition number to inspect:

GET DIAGNOSTICS @cno = NUMBER;

GET DIAGNOSTICS CONDITION @cno @errno = MYSQL\_ERRNO;

For information about permissible statement and condition information items, and which ones are populated when a condition occurs, see [Diagnostics Area Information Items](https://dev.mysql.com/doc/refman/5.6/en/diagnostics-area.html#diagnostics-area-information-items).

Here is an example that uses [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) and an exception handler in stored procedure context to assess the outcome of an insert operation. If the insert was successful, the procedure uses [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) to get the rows-affected count. This shows that you can use [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) multiple times to retrieve information about a statement as long as the diagnostics area has not been cleared.

CREATE PROCEDURE do\_insert(value INT)

BEGIN

-- Declare variables to hold diagnostics area information

DECLARE code CHAR(5) DEFAULT '00000';

DECLARE msg TEXT;

DECLARE nrows INT;

DECLARE result TEXT;

-- Declare exception handler for failed insert

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION

BEGIN

GET DIAGNOSTICS CONDITION 1

code = RETURNED\_SQLSTATE, msg = MESSAGE\_TEXT;

END;

-- Perform the insert

INSERT INTO t1 (int\_col) VALUES(value);

-- Check whether the insert was successful

IF code = '00000' THEN

GET DIAGNOSTICS nrows = ROW\_COUNT;

SET result = CONCAT('insert succeeded, row count = ',nrows);

ELSE

SET result = CONCAT('insert failed, error = ',code,', message = ',msg);

END IF;

-- Say what happened

SELECT result;

END;

Suppose that t1.int\_col is an integer column that is declared as NOT NULL. The procedure produces these results when invoked to insert non-NULL and NULL values, respectively:

mysql> CALL do\_insert(1);

+---------------------------------+

| result |

+---------------------------------+

| insert succeeded, row count = 1 |

+---------------------------------+

mysql> CALL do\_insert(NULL);

+-------------------------------------------------------------------------+

| result |

+-------------------------------------------------------------------------+

| insert failed, error = 23000, message = Column 'int\_col' cannot be null |

+-------------------------------------------------------------------------+

Within a condition handler, [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) should be used before other statements that might clear the diagnostics area and cause information to be lost about the condition that activated the handler.

#### RESIGNAL Statement

RESIGNAL [condition\_value]

[SET signal\_information\_item

[, signal\_information\_item] ...]

condition\_value: {

SQLSTATE [VALUE] sqlstate\_value

| condition\_name

}

signal\_information\_item:

condition\_information\_item\_name = simple\_value\_specification

condition\_information\_item\_name: {

CLASS\_ORIGIN

| SUBCLASS\_ORIGIN

| MESSAGE\_TEXT

| MYSQL\_ERRNO

| CONSTRAINT\_CATALOG

| CONSTRAINT\_SCHEMA

| CONSTRAINT\_NAME

| CATALOG\_NAME

| SCHEMA\_NAME

| TABLE\_NAME

| COLUMN\_NAME

| CURSOR\_NAME

}

condition\_name, simple\_value\_specification:

(see following discussion)

[RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) passes on the error condition information that is available during execution of a condition handler within a compound statement inside a stored procedure or function, trigger, or event. [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) may change some or all information before passing it on. [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) is related to [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html), but instead of originating a condition as [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) does, [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) relays existing condition information, possibly after modifying it.

[RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) makes it possible to both handle an error and return the error information. Otherwise, by executing an SQL statement within the handler, information that caused the handler's activation is destroyed. [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) also can make some procedures shorter if a given handler can handle part of a situation, then pass the condition “up the line” to another handler.

No privileges are required to execute the [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statement.

All forms of [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) require that the current context be a condition handler. Otherwise, [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) is illegal and a RESIGNAL when handler not active error occurs.

To retrieve information from the diagnostics area, use the [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) statement.

##### **RESIGNAL Overview**

For ***condition\_value*** and ***signal\_information\_item***, the definitions and rules are the same for [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) as for [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html). For example, the ***condition\_value*** can be an SQLSTATE value, and the value can indicate errors, warnings, or “not found.”

The [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statement takes ***condition\_value*** and SET clauses, both of which are optional. This leads to several possible uses:

* [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone:

RESIGNAL;

* [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) with new signal information:

RESIGNAL SET signal\_information\_item [, signal\_information\_item] ...;

* [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) with a condition value and possibly new signal information:
* RESIGNAL condition\_value

[SET signal\_information\_item [, signal\_information\_item] ...];

These use cases all cause changes to the diagnostics and condition areas:

* A diagnostics area contains one or more condition areas.
* A condition area contains condition information items, such as the SQLSTATE value, MYSQL\_ERRNO, or MESSAGE\_TEXT.

The maximum number of condition areas in a diagnostics area is determined by the value of the [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count) system variable.

A simple [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone means “pass on the error with no change.” It restores the last diagnostics area and makes it the current diagnostics area. That is, it “pops” the diagnostics area stack.

Within a condition handler that catches a condition, one use for [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone is to perform some other actions, and then pass on without change the original condition information (the information that existed before entry into the handler).

Example:

DROP TABLE IF EXISTS xx;

delimiter //

CREATE PROCEDURE p ()

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

SET @error\_count = @error\_count + 1;

IF @a = 0 THEN RESIGNAL; END IF;

END;

DROP TABLE xx;

END//

delimiter ;

SET @error\_count = 0;

SET @a = 0;

CALL p();

Suppose that the DROP TABLE xx statement fails. The diagnostics area stack looks like this:

DA 1. ERROR 1051 (42S02): Unknown table 'xx'

Then execution enters the EXIT handler. It starts by pushing a diagnostics area to the top of the stack, which now looks like this:

DA 1. ERROR 1051 (42S02): Unknown table 'xx'

DA 2. ERROR 1051 (42S02): Unknown table 'xx'

At this point, the contents of the first (current) and second (stacked) diagnostics areas are the same. The first diagnostics area may be modified by statements executing subsequently within the handler.

Usually a procedure statement clears the first diagnostics area. BEGIN is an exception, it does not clear, it does nothing. SET is not an exception, it clears, performs the operation, and produces a result of “success.” The diagnostics area stack now looks like this:

DA 1. ERROR 0000 (00000): Successful operation

DA 2. ERROR 1051 (42S02): Unknown table 'xx'

At this point, if @a = 0, [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) pops the diagnostics area stack, which now looks like this:

DA 1. ERROR 1051 (42S02): Unknown table 'xx'

And that is what the caller sees.

If @a is not 0, the handler simply ends, which means that there is no more use for the current diagnostics area (it has been “handled”), so it can be thrown away, causing the stacked diagnostics area to become the current diagnostics area again. The diagnostics area stack looks like this:

DA 1. ERROR 0000 (00000): Successful operation

The details make it look complex, but the end result is quite useful: Handlers can execute without destroying information about the condition that caused activation of the handler.

##### **RESIGNAL with New Signal Information**

[RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) with a SET clause provides new signal information, so the statement means “pass on the error with changes”:

RESIGNAL SET signal\_information\_item [, signal\_information\_item] ...;

As with [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone, the idea is to pop the diagnostics area stack so that the original information goes out. Unlike [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone, anything specified in the SET clause changes.

Example:

DROP TABLE IF EXISTS xx;

delimiter //

CREATE PROCEDURE p ()

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

SET @error\_count = @error\_count + 1;

IF @a = 0 THEN RESIGNAL SET MYSQL\_ERRNO = 5; END IF;

END;

DROP TABLE xx;

END//

delimiter ;

SET @error\_count = 0;

SET @a = 0;

CALL p();

Remember from the previous discussion that [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone results in a diagnostics area stack like this:

DA 1. ERROR 1051 (42S02): Unknown table 'xx'

The RESIGNAL SET MYSQL\_ERRNO = 5 statement results in this stack instead, which is what the caller sees:

DA 1. ERROR 5 (42S02): Unknown table 'xx'

In other words, it changes the error number, and nothing else.

The [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statement can change any or all of the signal information items, making the first condition area of the diagnostics area look quite different.

##### **RESIGNAL with a Condition Value and Optional New Signal Information**

[RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) with a condition value means “push a condition into the current diagnostics area.” If the SET clause is present, it also changes the error information.

RESIGNAL condition\_value

[SET signal\_information\_item [, signal\_information\_item] ...];

This form of [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) restores the last diagnostics area and makes it the current diagnostics area. That is, it “pops” the diagnostics area stack, which is the same as what a simple [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) alone would do. However, it also changes the diagnostics area depending on the condition value or signal information.

Example:

DROP TABLE IF EXISTS xx;

delimiter //

CREATE PROCEDURE p ()

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

SET @error\_count = @error\_count + 1;

IF @a = 0 THEN RESIGNAL SQLSTATE '45000' SET MYSQL\_ERRNO=5; END IF;

END;

DROP TABLE xx;

END//

delimiter ;

SET @error\_count = 0;

SET @a = 0;

SET @@max\_error\_count = 2;

CALL p();

SHOW ERRORS;

This is similar to the previous example, and the effects are the same, except that if [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) happens, the current condition area looks different at the end. (The reason the condition adds to rather than replaces the existing condition is the use of a condition value.)

The [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statement includes a condition value (SQLSTATE '45000'), so it adds a new condition area, resulting in a diagnostics area stack that looks like this:

DA 1. (condition 2) ERROR 1051 (42S02): Unknown table 'xx'

(condition 1) ERROR 5 (45000) Unknown table 'xx'

The result of [CALL p()](https://dev.mysql.com/doc/refman/5.6/en/call.html) and [SHOW ERRORS](https://dev.mysql.com/doc/refman/5.6/en/show-errors.html) for this example is:

mysql> CALL p();

ERROR 5 (45000): Unknown table 'xx'

mysql> SHOW ERRORS;

+-------+------+----------------------------------+

| Level | Code | Message |

+-------+------+----------------------------------+

| Error | 1051 | Unknown table 'xx' |

| Error | 5 | Unknown table 'xx' |

+-------+------+----------------------------------+

##### **RESIGNAL Requires Condition Handler Context**

All forms of [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) require that the current context be a condition handler. Otherwise, [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) is illegal and a RESIGNAL when handler not active error occurs. For example:

mysql> CREATE PROCEDURE p () RESIGNAL;

Query OK, 0 rows affected (0.00 sec)

mysql> CALL p();

ERROR 1645 (0K000): RESIGNAL when handler not active

Here is a more difficult example:

delimiter //

CREATE FUNCTION f () RETURNS INT

BEGIN

RESIGNAL;

RETURN 5;

END//

CREATE PROCEDURE p ()

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION SET @a=f();

SIGNAL SQLSTATE '55555';

END//

delimiter ;

CALL p();

[RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) occurs within the stored function f(). Although f() itself is invoked within the context of the EXIT handler, execution within f() has its own context, which is not handler context. Thus, RESIGNAL within f() results in a “handler not active” error.

In MySQL 5.5, handler scope rules are less developed. f() is considered to execute within handler context and [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) within f() is legal.

#### SIGNAL Statement

SIGNAL condition\_value

[SET signal\_information\_item

[, signal\_information\_item] ...]

condition\_value: {

SQLSTATE [VALUE] sqlstate\_value

| condition\_name

}

signal\_information\_item:

condition\_information\_item\_name = simple\_value\_specification

condition\_information\_item\_name: {

CLASS\_ORIGIN

| SUBCLASS\_ORIGIN

| MESSAGE\_TEXT

| MYSQL\_ERRNO

| CONSTRAINT\_CATALOG

| CONSTRAINT\_SCHEMA

| CONSTRAINT\_NAME

| CATALOG\_NAME

| SCHEMA\_NAME

| TABLE\_NAME

| COLUMN\_NAME

| CURSOR\_NAME

}

condition\_name, simple\_value\_specification:

(see following discussion)

[SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) is the way to “return” an error. [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) provides error information to a handler, to an outer portion of the application, or to the client. Also, it provides control over the error's characteristics (error number, SQLSTATE value, message). Without [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html), it is necessary to resort to workarounds such as deliberately referring to a nonexistent table to cause a routine to return an error.

No privileges are required to execute the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement.

To retrieve information from the diagnostics area, use the [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) statement.

##### **SIGNAL Overview**

The ***condition\_value*** in a [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement indicates the error value to be returned. It can be an SQLSTATE value (a 5-character string literal) or a ***condition\_name*** that refers to a named condition previously defined with [DECLARE ... CONDITION](https://dev.mysql.com/doc/refman/5.6/en/declare-condition.html) .

An SQLSTATE value can indicate errors, warnings, or “not found.” The first two characters of the value indicate its error class, as discussed in [Signal Condition Information Items](https://dev.mysql.com/doc/refman/5.6/en/signal.html#signal-condition-information-items). Some signal values cause statement termination; see [Effect of Signals on Handlers, Cursors, and Statements](https://dev.mysql.com/doc/refman/5.6/en/signal.html#signal-effects).

The SQLSTATE value for a [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement should not start with '00' because such values indicate success and are not valid for signaling an error. This is true whether the SQLSTATE value is specified directly in the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement or in a named condition referred to in the statement. If the value is invalid, a Bad SQLSTATE error occurs.

To signal a generic SQLSTATE value, use '45000', which means “unhandled user-defined exception.”

The [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement optionally includes a SET clause that contains multiple signal items, in a list of ***condition\_information\_item\_name*** = ***simple\_value\_specification*** assignments, separated by commas.

Each ***condition\_information\_item\_name*** may be specified only once in the SET clause. Otherwise, a Duplicate condition information item error occurs.

Valid ***simple\_value\_specification*** designators can be specified using stored procedure or function parameters, stored program local variables declared with [DECLARE](https://dev.mysql.com/doc/refman/5.6/en/declare.html), user-defined variables, system variables, or literals. A character literal may include a ***\_charset*** introducer.

For information about permissible ***condition\_information\_item\_name*** values, see [Signal Condition Information Items](https://dev.mysql.com/doc/refman/5.6/en/signal.html#signal-condition-information-items).

The following procedure signals an error or warning depending on the value of pval, its input parameter:

CREATE PROCEDURE p (pval INT)

BEGIN

DECLARE specialty CONDITION FOR SQLSTATE '45000';

IF pval = 0 THEN

SIGNAL SQLSTATE '01000';

ELSEIF pval = 1 THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'An error occurred';

ELSEIF pval = 2 THEN

SIGNAL specialty

SET MESSAGE\_TEXT = 'An error occurred';

ELSE

SIGNAL SQLSTATE '01000'

SET MESSAGE\_TEXT = 'A warning occurred', MYSQL\_ERRNO = 1000;

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'An error occurred', MYSQL\_ERRNO = 1001;

END IF;

END;

If pval is 0, p() signals a warning because SQLSTATE values that begin with '01' are signals in the warning class. The warning does not terminate the procedure, and can be seen with [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html) after the procedure returns.

If pval is 1, p() signals an error and sets the MESSAGE\_TEXT condition information item. The error terminates the procedure, and the text is returned with the error information.

If pval is 2, the same error is signaled, although the SQLSTATE value is specified using a named condition in this case.

If pval is anything else, p() first signals a warning and sets the message text and error number condition information items. This warning does not terminate the procedure, so execution continues and p() then signals an error. The error does terminate the procedure. The message text and error number set by the warning are replaced by the values set by the error, which are returned with the error information.

[SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) is typically used within stored programs, but it is a MySQL extension that it is permitted outside handler context. For example, if you invoke the [**mysql**](https://dev.mysql.com/doc/refman/5.6/en/mysql.html) client program, you can enter any of these statements at the prompt:

SIGNAL SQLSTATE '77777';

CREATE TRIGGER t\_bi BEFORE INSERT ON t

FOR EACH ROW SIGNAL SQLSTATE '77777';

CREATE EVENT e ON SCHEDULE EVERY 1 SECOND

DO SIGNAL SQLSTATE '77777';

[SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) executes according to the following rules:

If the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement indicates a particular SQLSTATE value, that value is used to signal the condition specified. Example:

CREATE PROCEDURE p (divisor INT)

BEGIN

IF divisor = 0 THEN

SIGNAL SQLSTATE '22012';

END IF;

END;

If the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement uses a named condition, the condition must be declared in some scope that applies to the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement, and must be defined using an SQLSTATE value, not a MySQL error number. Example:

CREATE PROCEDURE p (divisor INT)

BEGIN

DECLARE divide\_by\_zero CONDITION FOR SQLSTATE '22012';

IF divisor = 0 THEN

SIGNAL divide\_by\_zero;

END IF;

END;

If the named condition does not exist in the scope of the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement, an Undefined CONDITION error occurs.

If [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) refers to a named condition that is defined with a MySQL error number rather than an SQLSTATE value, a SIGNAL/RESIGNAL can only use a CONDITION defined with SQLSTATE error occurs. The following statements cause that error because the named condition is associated with a MySQL error number:

DECLARE no\_such\_table CONDITION FOR 1051;

SIGNAL no\_such\_table;

If a condition with a given name is declared multiple times in different scopes, the declaration with the most local scope applies. Consider the following procedure:

CREATE PROCEDURE p (divisor INT)

BEGIN

DECLARE my\_error CONDITION FOR SQLSTATE '45000';

IF divisor = 0 THEN

BEGIN

DECLARE my\_error CONDITION FOR SQLSTATE '22012';

SIGNAL my\_error;

END;

END IF;

SIGNAL my\_error;

END;

If divisor is 0, the first [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement executes. The innermost my\_error condition declaration applies, raising SQLSTATE '22012'.

If divisor is not 0, the second [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement executes. The outermost my\_error condition declaration applies, raising SQLSTATE '45000'.

Signals can be raised within exception handlers:

CREATE PROCEDURE p ()

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

SIGNAL SQLSTATE VALUE '99999'

SET MESSAGE\_TEXT = 'An error occurred';

END;

DROP TABLE no\_such\_table;

END;

CALL p() reaches the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html) statement. There is no table named no\_such\_table, so the error handler is activated. The error handler destroys the original error (“no such table”) and makes a new error with SQLSTATE '99999' and message An error occurred.

##### Signal Condition Information Items

The following table lists the names of diagnostics area condition information items that can be set in a [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) (or [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html)) statement. All items are standard SQL except MYSQL\_ERRNO, which is a MySQL extension.

Item Name Definition

--------- ----------

CLASS\_ORIGIN VARCHAR(64)

SUBCLASS\_ORIGIN VARCHAR(64)

CONSTRAINT\_CATALOG VARCHAR(64)

CONSTRAINT\_SCHEMA VARCHAR(64)

CONSTRAINT\_NAME VARCHAR(64)

CATALOG\_NAME VARCHAR(64)

SCHEMA\_NAME VARCHAR(64)

TABLE\_NAME VARCHAR(64)

COLUMN\_NAME VARCHAR(64)

CURSOR\_NAME VARCHAR(64)

MESSAGE\_TEXT VARCHAR(128)

MYSQL\_ERRNO SMALLINT UNSIGNED

The character set for character items is UTF-8.

It is illegal to assign NULL to a condition information item in a [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement.

A [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement always specifies an SQLSTATE value, either directly, or indirectly by referring to a named condition defined with an SQLSTATE value. The first two characters of an SQLSTATE value are its class, and the class determines the default value for the condition information items:

* Class = '00' (success)

Illegal. SQLSTATE values that begin with '00' indicate success and are not valid for [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html).

* Class = '01' (warning)
* MESSAGE\_TEXT = 'Unhandled user-defined warning condition';

MYSQL\_ERRNO = ER\_SIGNAL\_WARN

* Class = '02' (not found)
* MESSAGE\_TEXT = 'Unhandled user-defined not found condition';

MYSQL\_ERRNO = ER\_SIGNAL\_NOT\_FOUND

* Class > '02' (exception)
* MESSAGE\_TEXT = 'Unhandled user-defined exception condition';

MYSQL\_ERRNO = ER\_SIGNAL\_EXCEPTION

For legal classes, the other condition information items are set as follows:

CLASS\_ORIGIN = SUBCLASS\_ORIGIN = '';

CONSTRAINT\_CATALOG = CONSTRAINT\_SCHEMA = CONSTRAINT\_NAME = '';

CATALOG\_NAME = SCHEMA\_NAME = TABLE\_NAME = COLUMN\_NAME = '';

CURSOR\_NAME = '';

The error values that are accessible after [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) executes are the SQLSTATE value raised by the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement and the MESSAGE\_TEXT and MYSQL\_ERRNO items. These values are available from the C API:

* [mysql\_sqlstate()](https://dev.mysql.com/doc/c-api/5.6/en/mysql-sqlstate.html) returns the SQLSTATE value.
* [mysql\_errno()](https://dev.mysql.com/doc/c-api/5.6/en/mysql-errno.html) returns the MYSQL\_ERRNO value.
* [mysql\_error()](https://dev.mysql.com/doc/c-api/5.6/en/mysql-error.html) returns the MESSAGE\_TEXT value.

At the SQL level, the output from [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html) and [SHOW ERRORS](https://dev.mysql.com/doc/refman/5.6/en/show-errors.html) indicates the MYSQL\_ERRNO and MESSAGE\_TEXT values in the Code and Message columns.

To retrieve information from the diagnostics area, use the [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) statement.

##### **Effect of Signals on Handlers, Cursors, and Statements**

Signals have different effects on statement execution depending on the signal class. The class determines how severe an error is. MySQL ignores the value of the [sql\_mode](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_sql_mode) system variable; in particular, strict SQL mode does not matter. MySQL also ignores IGNORE: The intent of [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) is to raise a user-generated error explicitly, so a signal is never ignored.

In the following descriptions, “unhandled” means that no handler for the signaled SQLSTATE value has been defined with [DECLARE ... HANDLER](https://dev.mysql.com/doc/refman/5.6/en/declare-handler.html).

* Class = '00' (success)

Illegal. SQLSTATE values that begin with '00' indicate success and are not valid for [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html).

* Class = '01' (warning)

The value of the [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count) system variable goes up. [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html) shows the signal. SQLWARNING handlers catch the signal.

Warnings cannot be returned from stored functions because the [RETURN](https://dev.mysql.com/doc/refman/5.6/en/return.html) statement that causes the function to return clears the diagnostic area. The statement thus clears any warnings that may have been present there (and resets [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count) to 0).

* Class = '02' (not found)

NOT FOUND handlers catch the signal. There is no effect on cursors. If the signal is unhandled in a stored function, statements end.

* Class > '02' (exception)

SQLEXCEPTION handlers catch the signal. If the signal is unhandled in a stored function, statements end.

* Class = '40'

Treated as an ordinary exception.

#### Scope Rules for Handlers

A stored program may include handlers to be invoked when certain conditions occur within the program. The applicability of each handler depends on its location within the program definition and on the condition or conditions that it handles:

* A handler declared in a [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block is in scope only for the SQL statements following the handler declarations in the block. If the handler itself raises a condition, it cannot handle that condition, nor can any other handlers declared in the block. In the following example, handlers H1 and H2 are in scope for conditions raised by statements ***stmt1*** and ***stmt2***. But neither H1 nor H2 are in scope for conditions raised in the body of H1 or H2.
* BEGIN -- outer block
* DECLARE EXIT HANDLER FOR ...; -- handler H1
* DECLARE EXIT HANDLER FOR ...; -- handler H2
* stmt1;
* stmt2;

END;

* A handler is in scope only for the block in which it is declared, and cannot be activated for conditions occurring outside that block. In the following example, handler H1 is in scope for ***stmt1*** in the inner block, but not for ***stmt2*** in the outer block:
* BEGIN -- outer block
* BEGIN -- inner block
* DECLARE EXIT HANDLER FOR ...; -- handler H1
* stmt1;
* END;
* stmt2;

END;

* A handler can be specific or general. A specific handler is for a MySQL error code, SQLSTATE value, or condition name. A general handler is for a condition in the SQLWARNING, SQLEXCEPTION, or NOT FOUND class. Condition specificity is related to condition precedence, as described later.

Multiple handlers can be declared in different scopes and with different specificities. For example, there might be a specific MySQL error code handler in an outer block, and a general SQLWARNING handler in an inner block. Or there might be handlers for a specific MySQL error code and the general SQLWARNING class in the same block.

Whether a handler is activated depends not only on its own scope and condition value, but on what other handlers are present. When a condition occurs in a stored program, the server searches for applicable handlers in the current scope (current [BEGIN ... END](https://dev.mysql.com/doc/refman/5.6/en/begin-end.html) block). If there are no applicable handlers, the search continues outward with the handlers in each successive containing scope (block). When the server finds one or more applicable handlers at a given scope, it chooses among them based on condition precedence:

* A MySQL error code handler takes precedence over an SQLSTATE value handler.
* An SQLSTATE value handler takes precedence over general SQLWARNING, SQLEXCEPTION, or NOT FOUND handlers.
* An SQLEXCEPTION handler takes precedence over an SQLWARNING handler.
* It is possible to have several applicable handlers with the same precedence. For example, a statement could generate multiple warnings with different error codes, for each of which an error-specific handler exists. In this case, the choice of which handler the server activates is nondeterministic, and may change depending on the circumstances under which the condition occurs.

One implication of the handler selection rules is that if multiple applicable handlers occur in different scopes, handlers with the most local scope take precedence over handlers in outer scopes, even over those for more specific conditions.

If there is no appropriate handler when a condition occurs, the action taken depends on the class of the condition:

* For SQLEXCEPTION conditions, the stored program terminates at the statement that raised the condition, as if there were an EXIT handler. If the program was called by another stored program, the calling program handles the condition using the handler selection rules applied to its own handlers.
* For SQLWARNING conditions, the program continues executing, as if there were a CONTINUE handler.
* For NOT FOUND conditions, if the condition was raised normally, the action is CONTINUE. If it was raised by [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) or [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html), the action is EXIT.

The following examples demonstrate how MySQL applies the handler selection rules.

This procedure contains two handlers, one for the specific SQLSTATE value ('42S02') that occurs for attempts to drop a nonexistent table, and one for the general SQLEXCEPTION class:

CREATE PROCEDURE p1()

BEGIN

DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'

SELECT 'SQLSTATE handler was activated' AS msg;

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION

SELECT 'SQLEXCEPTION handler was activated' AS msg;

DROP TABLE test.t;

END;

Both handlers are declared in the same block and have the same scope. However, SQLSTATE handlers take precedence over SQLEXCEPTION handlers, so if the table t is nonexistent, the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html) statement raises a condition that activates the SQLSTATE handler:

mysql> CALL p1();

+--------------------------------+

| msg |

+--------------------------------+

| SQLSTATE handler was activated |

+--------------------------------+

This procedure contains the same two handlers. But this time, the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html) statement and SQLEXCEPTION handler are in an inner block relative to the SQLSTATE handler:

CREATE PROCEDURE p2()

BEGIN -- outer block

DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'

SELECT 'SQLSTATE handler was activated' AS msg;

BEGIN -- inner block

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION

SELECT 'SQLEXCEPTION handler was activated' AS msg;

DROP TABLE test.t; -- occurs within inner block

END;

END;

In this case, the handler that is more local to where the condition occurs takes precedence. The SQLEXCEPTION handler activates, even though it is more general than the SQLSTATE handler:

mysql> CALL p2();

+------------------------------------+

| msg |

+------------------------------------+

| SQLEXCEPTION handler was activated |

+------------------------------------+

In this procedure, one of the handlers is declared in a block inner to the scope of the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html) statement:

CREATE PROCEDURE p3()

BEGIN -- outer block

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION

SELECT 'SQLEXCEPTION handler was activated' AS msg;

BEGIN -- inner block

DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'

SELECT 'SQLSTATE handler was activated' AS msg;

END;

DROP TABLE test.t; -- occurs within outer block

END;

Only the SQLEXCEPTION handler applies because the other one is not in scope for the condition raised by the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html):

mysql> CALL p3();

+------------------------------------+

| msg |

+------------------------------------+

| SQLEXCEPTION handler was activated |

+------------------------------------+

In this procedure, both handlers are declared in a block inner to the scope of the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html) statement:

CREATE PROCEDURE p4()

BEGIN -- outer block

BEGIN -- inner block

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION

SELECT 'SQLEXCEPTION handler was activated' AS msg;

DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'

SELECT 'SQLSTATE handler was activated' AS msg;

END;

DROP TABLE test.t; -- occurs within outer block

END;

Neither handler applies because they are not in scope for the [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html). The condition raised by the statement goes unhandled and terminates the procedure with an error:

mysql> CALL p4();

ERROR 1051 (42S02): Unknown table 'test.t'

#### The MySQL Diagnostics Area

SQL statements produce diagnostic information that populates the diagnostics area. Standard SQL has a diagnostics area stack, containing a diagnostics area for each nested execution context. Standard SQL also supports GET STACKED DIAGNOSTICS syntax for referring to the second diagnostics area during condition handler execution. MySQL does not support the STACKED keyword until MySQL 5.7. In MySQL 5.6, there is a single diagnostics area containing information from the most recent statement that wrote to it.

The following discussion describes the structure of the diagnostics area in MySQL, the information items recognized by MySQL and how statements clear and set the diagnostics area.

##### **Diagnostics Area Structure**

The diagnostics area contains two kinds of information:

* Statement information, such as the number of conditions that occurred or the affected-rows count.
* Condition information, such as the error code and message. If a statement raises multiple conditions, this part of the diagnostics area has a condition area for each one. If a statement raises no conditions, this part of the diagnostics area is empty.

For a statement that produces three conditions, the diagnostics area contains statement and condition information like this:

Statement information:

row count

... other statement information items ...

Condition area list:

Condition area 1:

error code for condition 1

error message for condition 1

... other condition information items ...

Condition area 2:

error code for condition 2:

error message for condition 2

... other condition information items ...

Condition area 3:

error code for condition 3

error message for condition 3

... other condition information items ...

##### **Diagnostics Area Information Items**

The diagnostics area contains statement and condition information items. Numeric items are integers. The character set for character items is UTF-8. No item can be NULL. If a statement or condition item is not set by a statement that populates the diagnostics area, its value is 0 or the empty string, depending on the item data type.

The statement information part of the diagnostics area contains these items:

* NUMBER: An integer indicating the number of condition areas that have information.
* ROW\_COUNT: An integer indicating the number of rows affected by the statement. ROW\_COUNT has the same value as the [ROW\_COUNT()](https://dev.mysql.com/doc/refman/5.6/en/information-functions.html#function_row-count) .

The condition information part of the diagnostics area contains a condition area for each condition. Condition areas are numbered from 1 to the value of the NUMBER statement condition item. If NUMBER is 0, there are no condition areas.

Each condition area contains the items in the following list. All items are standard SQL except MYSQL\_ERRNO, which is a MySQL extension. The definitions apply for conditions generated other than by a signal (that is, by a [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) or [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statement). For nonsignal conditions, MySQL populates only those condition items not described as always empty. The effects of signals on the condition area are described later.

* CLASS\_ORIGIN: A string containing the class of the RETURNED\_SQLSTATE value. If the RETURNED\_SQLSTATE value begins with a class value defined in SQL standards document ISO 9075-2 (section 24.1, SQLSTATE), CLASS\_ORIGIN is 'ISO 9075'. Otherwise, CLASS\_ORIGIN is 'MySQL'.
* SUBCLASS\_ORIGIN: A string containing the subclass of the RETURNED\_SQLSTATE value. If CLASS\_ORIGIN is 'ISO 9075' or RETURNED\_SQLSTATE ends with '000', SUBCLASS\_ORIGIN is 'ISO 9075'. Otherwise, SUBCLASS\_ORIGIN is 'MySQL'.
* RETURNED\_SQLSTATE: A string that indicates the SQLSTATE value for the condition.
* MESSAGE\_TEXT: A string that indicates the error message for the condition.
* MYSQL\_ERRNO: An integer that indicates the MySQL error code for the condition.
* CONSTRAINT\_CATALOG, CONSTRAINT\_SCHEMA, CONSTRAINT\_NAME: Strings that indicate the catalog, schema, and name for a violated constraint. They are always empty.
* CATALOG\_NAME, SCHEMA\_NAME, TABLE\_NAME, COLUMN\_NAME: Strings that indicate the catalog, schema, table, and column related to the condition. They are always empty.
* CURSOR\_NAME: A string that indicates the cursor name. This is always empty.

For the RETURNED\_SQLSTATE, MESSAGE\_TEXT, and MYSQL\_ERRNO values for particular errors, see [Server Error Message Reference](https://dev.mysql.com/doc/mysql-errors/5.6/en/server-error-reference.html).

If a [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) (or [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html)) statement populates the diagnostics area, its SET clause can assign to any condition information item except RETURNED\_SQLSTATE any value that is legal for the item data type. [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) also sets the RETURNED\_SQLSTATE value, but not directly in its SET clause. That value comes from the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) statement SQLSTATE argument.

[SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) also sets statement information items. It sets NUMBER to 1. It sets ROW\_COUNT to −1 for errors and 0 otherwise.

##### How the Diagnostics Area is Cleared and Populated

Most nondiagnostic SQL statements populate the diagnostics area automatically, and its contents can be set explicitly with the [SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html) and [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statements. The diagnostics area can be examined with [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) to extract specific items, or with [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html) or [SHOW ERRORS](https://dev.mysql.com/doc/refman/5.6/en/show-errors.html) to see conditions or errors.

SQL statements clear and set the diagnostics area as follows:

* When the server starts executing a statement after parsing it, it clears the diagnostics area for nondiagnostic statements that use tables. Diagnostic statements do not clear the diagnostics area. These statements are diagnostic:
  + [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html)
  + [SHOW ERRORS](https://dev.mysql.com/doc/refman/5.6/en/show-errors.html)
  + [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html)
* If a statement raises a condition, the diagnostics area is cleared of conditions that belong to earlier statements. The exception is that conditions raised by [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) and [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) are added to the diagnostics area without clearing it.

Thus, even a statement that does not normally clear the diagnostics area when it begins executing clears it if the statement raises a condition.

The following example shows the effect of various statements on the diagnostics area, using [SHOW WARNINGS](https://dev.mysql.com/doc/refman/5.6/en/show-warnings.html) to display information about conditions stored there.

This [DROP TABLE](https://dev.mysql.com/doc/refman/5.6/en/drop-table.html) statement uses a table, so it clears the diagnostics area and populates it when the condition occurs:

mysql> DROP TABLE IF EXISTS test.no\_such\_table;

Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;

+-------+------+------------------------------------+

| Level | Code | Message |

+-------+------+------------------------------------+

| Note | 1051 | Unknown table 'test.no\_such\_table' |

+-------+------+------------------------------------+

1 row in set (0.00 sec)

This [SET](https://dev.mysql.com/doc/refman/5.6/en/set-variable.html) statement does not use tables and does not generate warnings, so it leaves the diagnostics area unchanged:

mysql> SET @x = 1;

Query OK, 0 rows affected (0.00 sec)

mysql> SHOW WARNINGS;

+-------+------+------------------------------------+

| Level | Code | Message |

+-------+------+------------------------------------+

| Note | 1051 | Unknown table 'test.no\_such\_table' |

+-------+------+------------------------------------+

1 row in set (0.00 sec)

This [SET](https://dev.mysql.com/doc/refman/5.6/en/set-variable.html) statement generates an error, so it clears and populates the diagnostics area:

mysql> SET @x = @@x;

ERROR 1193 (HY000): Unknown system variable 'x'

mysql> SHOW WARNINGS;

+-------+------+-----------------------------+

| Level | Code | Message |

+-------+------+-----------------------------+

| Error | 1193 | Unknown system variable 'x' |

+-------+------+-----------------------------+

1 row in set (0.00 sec)

The previous [SET](https://dev.mysql.com/doc/refman/5.6/en/set-variable.html) statement produced a single condition, so 1 is the only valid condition number for [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) at this point. The following statement uses a condition number of 2, which produces a warning that is added to the diagnostics area without clearing it:

mysql> GET DIAGNOSTICS CONDITION 2 @p = MESSAGE\_TEXT;

Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;

+-------+------+------------------------------+

| Level | Code | Message |

+-------+------+------------------------------+

| Error | 1193 | Unknown system variable 'xx' |

| Error | 1753 | Invalid condition number |

+-------+------+------------------------------+

2 rows in set (0.00 sec)

Now there are two conditions in the diagnostics area, so the same [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) statement succeeds:

mysql> GET DIAGNOSTICS CONDITION 2 @p = MESSAGE\_TEXT;

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @p;

+--------------------------+

| @p |

+--------------------------+

| Invalid condition number |

+--------------------------+

1 row in set (0.01 sec)

##### **Diagnostics Area-Related System Variables**

Certain system variables control or are related to some aspects of the diagnostics area:

* [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count) controls the number of condition areas in the diagnostics area. If more conditions than this occur, MySQL silently discards information for the excess conditions. (Conditions added by [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) are always added, with older conditions being discarded as necessary to make room.)
* [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count) indicates the number of conditions that occurred. This includes errors, warnings, and notes. Normally, NUMBER and [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count) are the same. However, as the number of conditions generated exceeds [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count), the value of [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count) continues to rise whereas NUMBER remains capped at [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count) because no additional conditions are stored in the diagnostics area.
* [error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_error_count) indicates the number of errors that occurred. This value includes “not found” and exception conditions, but excludes warnings and notes. Like [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count), its value can exceed [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count).
* If the [sql\_notes](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_sql_notes) system variable is set to 0, notes are not stored and do not increment [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count).

Example: If [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count) is 10, the diagnostics area can contain a maximum of 10 condition areas. Suppose that a statement raises 20 conditions, 12 of which are errors. In that case, the diagnostics area contains the first 10 conditions, NUMBER is 10, [warning\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_warning_count) is 20, and [error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_error_count) is 12.

Changes to the value of [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count) have no effect until the next attempt to modify the diagnostics area. If the diagnostics area contains 10 condition areas and [max\_error\_count](https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_error_count) is set to 5, that has no immediate effect on the size or content of the diagnostics area.

Before MySQL 5.6, statement information items are not available directly. ROW\_COUNT can be obtained by calling the ROW\_COUNT() function. NUMBER is approximated by the value of the warning\_count system variable. However, whereas NUMBER is capped to the value of max\_error\_count, warning\_count is not.

#### Condition Handling and OUT or INOUT Parameters

If a stored procedure exits with an unhandled exception, modified values of OUT and INOUT parameters are not propogated back to the caller.

If an exception is handled by a CONTINUE or EXIT handler that contains a [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) statement, execution of [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html) pops the Diagnostics Area stack, thus signalling the exception (that is, the information that existed before entry into the handler). If the exception is an error, the values of OUT and INOUT parameters are not propogated back to the caller.

#### Restrictions on Condition Handling

[SIGNAL](https://dev.mysql.com/doc/refman/5.6/en/signal.html" \o "13.6.7.5 SIGNAL Statement), [RESIGNAL](https://dev.mysql.com/doc/refman/5.6/en/resignal.html), and [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/5.6/en/get-diagnostics.html) are not permissible as prepared statements. For example, this statement is invalid:

PREPARE stmt1 FROM 'SIGNAL SQLSTATE "02000"';

SQLSTATE values in class '04' are not treated specially. They are handled the same as other exceptions.

Standard SQL has a diagnostics area stack, containing a diagnostics area for each nested execution context. Standard SQL syntax includes GET STACKED DIAGNOSTICS for referring to stacked areas. MySQL does not support the STACKED keyword because there is a single diagnostics area containing information from the most recent statement that wrote to it.

In standard SQL, the first condition relates to the SQLSTATE value returned for the previous SQL statement. In MySQL, this is not guaranteed, so to get the main error, you cannot do this:

GET DIAGNOSTICS CONDITION 1 @errno = MYSQL\_ERRNO;

Instead, do this:

GET DIAGNOSTICS @cno = NUMBER;

GET DIAGNOSTICS CONDITION @cno @errno = MYSQL\_ERRNO;

# Stored Objects

This chapter discusses stored database objects that are defined in terms of SQL code that is stored on the server for later execution.

Stored objects include these object types:

* Stored procedure: An object created with [CREATE PROCEDURE](https://dev.mysql.com/doc/refman/5.6/en/create-procedure.html) and invoked using the [CALL](https://dev.mysql.com/doc/refman/5.6/en/call.html) statement. A procedure does not have a return value but can modify its parameters for later inspection by the caller. It can also generate result sets to be returned to the client program.
* Stored function: An object created with [CREATE FUNCTION](https://dev.mysql.com/doc/refman/5.6/en/create-function.html) and used much like a built-in function. You invoke it in an expression and it returns a value during expression evaluation.
* Trigger: An object created with [CREATE TRIGGER](https://dev.mysql.com/doc/refman/5.6/en/create-trigger.html) that is associated with a table. A trigger is activated when a particular event occurs for the table, such as an insert or update.
* Event: An object created with [CREATE EVENT](https://dev.mysql.com/doc/refman/5.6/en/create-event.html) and invoked by the server according to schedule.
* View: An object created with [CREATE VIEW](https://dev.mysql.com/doc/refman/5.6/en/create-view.html) that when referenced produces a result set. A view acts as a virtual table.

Terminology used in this document reflects the stored object hierarchy:

* Stored routines include stored procedures and functions.
* Stored programs include stored routines, triggers, and events.
* Stored objects include stored programs and views.

This chapter describes how to use stored objects. The following sections provide additional information about SQL syntax for statements related to these objects, and about object processing:

* For each object type, there are CREATE, ALTER, and DROP statements that control which objects exist and how they are defined..
* The [CALL](https://dev.mysql.com/doc/refman/5.6/en/call.html) statement is used to invoke stored procedures..
* Stored program definitions include a body that may use compound statements, loops, conditionals, and declared variables.
* Metadata changes to objects referred to by stored programs are detected and cause automatic reparsing of the affected statements when the program is next executed..

## Defining Stored Programs

Each stored program contains a body that consists of an SQL statement. This statement may be a compound statement made up of several statements separated by semicolon (;) characters. For example, the following stored procedure has a body made up of a [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) block that contains a [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html) statement and a [REPEAT](https://dev.mysql.com/doc/refman/8.0/en/repeat.html) loop that itself contains another [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html) statement:

CREATE PROCEDURE dorepeat(p1 INT)

BEGIN

SET @x = 0;

REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;

END;

If you use the [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) client program to define a stored program containing semicolon characters, a problem arises. By default, [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) itself recognizes the semicolon as a statement delimiter, so you must redefine the delimiter temporarily to cause [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) to pass the entire stored program definition to the server.

To redefine the [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) delimiter, use the delimiter command. The following example shows how to do this for the dorepeat() procedure just shown. The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement, and then restored to ; before invoking the procedure. This enables the ; delimiter used in the procedure body to be passed through to the server rather than being interpreted by [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) itself.

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> delimiter ;

mysql> CALL dorepeat(1000);

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x;

+------+

| @x |

+------+

| 1001 |

+------+

1 row in set (0.00 sec)

You can redefine the delimiter to a string other than //, and the delimiter can consist of a single character or multiple characters. 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)

## Using Stored Routines

MySQL supports stored routines (procedures and functions). A stored routine is a set of SQL statements that can be stored in the server. Once this has been done, clients don't need to keep reissuing the individual statements but can refer to the stored routine instead.

Stored routines can be particularly useful in certain situations:

* When multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.
* When security is paramount. Banks, for example, use stored procedures and functions for all common operations. This provides a consistent and secure environment, and routines can ensure that each operation is properly logged. In such a setup, applications and users would have no access to the database tables directly, but can only execute specific stored routines.

Stored routines can provide improved performance because less information needs to be sent between the server and the client. The tradeoff is that this does increase the load on the database server because more of the work is done on the server side and less is done on the client (application) side. Consider this if many client machines (such as Web servers) are serviced by only one or a few database servers.

Stored routines also enable you to have libraries of functions in the database server. This is a feature shared by modern application languages that enable such design internally (for example, by using classes). Using these client application language features is beneficial for the programmer even outside the scope of database use.

MySQL follows the SQL:2003 syntax for stored routines, which is also used by IBM's DB2. All syntax described here is supported and any limitations and extensions are documented where appropriate.

### 25.2.1 Stored Routine Syntax

A stored routine is either a procedure or a function. Stored routines are created with the [CREATE PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html) and [CREATE FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/create-function.html) statements. A procedure is invoked using a [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statement, and can only pass back values using output variables. A function can be called from inside a statement just like any other function (that is, by invoking the function's name), and can return a scalar value. The body of a stored routine can use compound statements.

Stored routines can be dropped with the [DROP PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/drop-procedure.html) and [DROP FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/drop-function.html) statements , and altered with the [ALTER PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/alter-procedure.html) and [ALTER FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/alter-function.html) statements.

A stored procedure or function is associated with a particular database. This has several implications:

* When the routine is invoked, an implicit USE ***db\_name*** is performed (and undone when the routine terminates). [USE](https://dev.mysql.com/doc/refman/8.0/en/use.html) statements within stored routines are not permitted.
* You can qualify routine names with the database name. This can be used to refer to a routine that is not in the current database. For example, to invoke a stored procedure p or function f that is associated with the test database, you can say CALL test.p() or test.f().
* When a database is dropped, all stored routines associated with it are dropped as well.

Stored functions cannot be recursive.

Recursion in stored procedures is permitted but disabled by default. To enable recursion, set the [max\_sp\_recursion\_depth](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_sp_recursion_depth) server system variable to a value greater than zero. Stored procedure recursion increases the demand on thread stack space. If you increase the value of [max\_sp\_recursion\_depth](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_sp_recursion_depth), it may be necessary to increase thread stack size by increasing the value of [thread\_stack](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_thread_stack) at server startup..

MySQL supports a very useful extension that enables the use of regular [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statements (that is, without using cursors or local variables) inside a stored procedure. The result set of such a query is simply sent directly to the client. Multiple [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statements generate multiple result sets, so the client must use a MySQL client library that supports multiple result sets. This means the client must use a client library from a version of MySQL at least as recent as 4.1. The client should also specify the CLIENT\_MULTI\_RESULTS option when it connects. For C programs, this can be done with the [mysql\_real\_connect()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-connect.html) C API function.

In MySQL 8.0.22 and later, a user variable referenced by a statement in a stored procedure has its type determined the first time the procedure is invoked, and retains this type each time the procedure is invoked thereafter.

### Stored Routines and MySQL Privileges

The MySQL grant system takes stored routines into account as follows:

* The [CREATE ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-routine) privilege is needed to create stored routines.
* The [ALTER ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_alter-routine) privilege is needed to alter or drop stored routines. This privilege is granted automatically to the creator of a routine if necessary, and dropped from the creator when the routine is dropped.
* The [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) privilege is required to execute stored routines. However, this privilege is granted automatically to the creator of a routine if necessary (and dropped from the creator when the routine is dropped). Also, the default SQL SECURITY characteristic for a routine is DEFINER, which enables users who have access to the database with which the routine is associated to execute the routine.
* If the [automatic\_sp\_privileges](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_automatic_sp_privileges) system variable is 0, the [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) and [ALTER ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_alter-routine) privileges are not automatically granted to and dropped from the routine creator.
* The creator of a routine is the account used to execute the CREATE statement for it. This might not be the same as the account named as the DEFINER in the routine definition.
* The account named as a routine DEFINER can see all routine properties, including its definition. The account thus has full access to the routine output as produced by:
  + The contents of the [INFORMATION\_SCHEMA.ROUTINES](https://dev.mysql.com/doc/refman/8.0/en/information-schema-routines-table.html) table.
  + The [SHOW CREATE FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/show-create-function.html) and [SHOW CREATE PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/show-create-procedure.html) statements.
  + The [SHOW FUNCTION CODE](https://dev.mysql.com/doc/refman/8.0/en/show-function-code.html) and [SHOW PROCEDURE CODE](https://dev.mysql.com/doc/refman/8.0/en/show-procedure-code.html) statements.
  + The [SHOW FUNCTION STATUS](https://dev.mysql.com/doc/refman/8.0/en/show-function-status.html) and [SHOW PROCEDURE STATUS](https://dev.mysql.com/doc/refman/8.0/en/show-procedure-status.html) statements.
* For an account other than the account named as the routine DEFINER, access to routine properties depends on the privileges granted to the account:
  + With the [SHOW\_ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_show-routine) privilege or the global [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) privilege, the account can see all routine properties, including its definition.
  + With the [CREATE ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-routine), [ALTER ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_alter-routine) or [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) privilege granted at a scope that includes the routine, the account can see all routine properties except its definition.

### 25.2.4 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()](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_last-insert-id) changes the same way as for statements executed outside the body of these kinds of objects. The effect of a stored routine or trigger upon the value of [LAST\_INSERT\_ID()](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_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()](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_last-insert-id), the changed value is 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 statments do not see a changed value.

## Using Triggers

A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. Some uses for triggers are to perform checks of values to be inserted into a table or to perform calculations on values involved in an update.

A trigger is defined to activate when a statement inserts, updates, or deletes rows in the associated table. These row operations are trigger events. For example, rows can be inserted by [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) or [LOAD DATA](https://dev.mysql.com/doc/refman/8.0/en/load-data.html) statements, and an insert trigger activates for each inserted row. A trigger can be set to activate either before or after the trigger event. For example, you can have a trigger activate before each row that is inserted into a table or after each row that is updated.

**Important**

MySQL triggers activate only for changes made to tables by SQL statements. This includes changes to base tables that underlie updatable views. Triggers do not activate for changes to tables made by APIs that do not transmit SQL statements to the MySQL Server. This means that triggers are not activated by updates made using the [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html) API.

Triggers are not activated by changes in INFORMATION\_SCHEMA or performance\_schema tables. Those tables are actually views and triggers are not permitted on views.

The following sections describe the syntax for creating and dropping triggers, show some examples of how to use them, and indicate how to obtain trigger metadata.

### Trigger Syntax and Examples

To create a trigger or drop a trigger, use the [CREATE TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html) or [DROP TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/drop-trigger.html) statement

Here is a simple example that associates a trigger with a table, to activate for [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) operations. The trigger acts as an accumulator, summing the values inserted into one of the columns of the table.

mysql> CREATE TABLE account (acct\_num INT, amount DECIMAL(10,2));

Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TRIGGER ins\_sum BEFORE INSERT ON account

FOR EACH ROW SET @sum = @sum + NEW.amount;

Query OK, 0 rows affected (0.01 sec)

The [CREATE TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html) statement creates a trigger named ins\_sum that is associated with the account table. It also includes clauses that specify the trigger action time, the triggering event, and what to do when the trigger activates:

* The keyword BEFORE indicates the trigger action time. In this case, the trigger activates before each row inserted into the table. The other permitted keyword here is AFTER.
* The keyword INSERT indicates the trigger event; that is, the type of operation that activates the trigger. In the example, [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) operations cause trigger activation. You can also create triggers for [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html) and [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) operations.
* The statement following FOR EACH ROW defines the trigger body; that is, the statement to execute each time the trigger activates, which occurs once for each row affected by the triggering event. In the example, the trigger body is a simple [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html) that accumulates into a user variable the values inserted into the amount column. The statement refers to the column as NEW.amount which means “the value of the amount column to be inserted into the new row.”

To use the trigger, set the accumulator variable to zero, execute an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, and then see what value the variable has afterward:

mysql> SET @sum = 0;

mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);

mysql> SELECT @sum AS 'Total amount inserted';

+-----------------------+

| Total amount inserted |

+-----------------------+

| 1852.48 |

+-----------------------+

In this case, the value of @sum after the [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement has executed is 14.98 + 1937.50 - 100, or 1852.48.

To destroy the trigger, use a [DROP TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/drop-trigger.html) statement. You must specify the schema name if the trigger is not in the default schema:

mysql> DROP TRIGGER test.ins\_sum;

If you drop a table, any triggers for the table are also dropped.

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name.

It is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two BEFORE UPDATE triggers for a table. By default, triggers that have the same trigger event and action time activate in the order they were created. To affect trigger order, specify a clause after FOR EACH ROW that indicates FOLLOWS or PRECEDES and the name of an existing trigger that also has the same trigger event and action time. With FOLLOWS, the new trigger activates after the existing trigger. With PRECEDES, the new trigger activates before the existing trigger.

For example, the following trigger definition defines another BEFORE INSERT trigger for the account table:

mysql> CREATE TRIGGER ins\_transaction BEFORE INSERT ON account

FOR EACH ROW PRECEDES ins\_sum

SET

@deposits = @deposits + IF(NEW.amount>0,NEW.amount,0),

@withdrawals = @withdrawals + IF(NEW.amount<0,-NEW.amount,0);

Query OK, 0 rows affected (0.01 sec)

This trigger, ins\_transaction, is similar to ins\_sum but accumulates deposits and withdrawals separately. It has a PRECEDES clause that causes it to activate before ins\_sum; without that clause, it would activate after ins\_sum because it is created after ins\_sum.

Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger. OLD and NEW are MySQL extensions to triggers; they are not case-sensitive.

In an INSERT trigger, only NEW.***col\_name*** can be used; there is no old row. In a DELETE trigger, only OLD.***col\_name*** can be used; there is no new row. In an UPDATE trigger, you can use OLD.***col\_name*** to refer to the columns of a row before it is updated and NEW.***col\_name*** to refer to the columns of the row after it is updated.

A column named with OLD is read only. You can refer to it (if you have the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) privilege), but not modify it. You can refer to a column named with NEW if you have the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) privilege for it. In a BEFORE trigger, you can also change its value with SET NEW.***col\_name*** = ***value*** if you have the [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_update) privilege for it. This means you can use a trigger to modify the values to be inserted into a new row or used to update a row. (Such a SET statement has no effect in an AFTER trigger because the row change has already occurred.)

In a BEFORE trigger, the NEW value for an AUTO\_INCREMENT column is 0, not the sequence number that is generated automatically when the new row actually is inserted.

By using the [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) construct, you can define a trigger that executes multiple statements. Within the BEGIN block, you also can use other syntax that is permitted within stored routines such as conditionals and loops. However, just as for stored routines, if you use the [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) program to define a trigger that executes multiple statements, it is necessary to redefine the [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) statement delimiter so that you can use the ; statement delimiter within the trigger definition. The following example illustrates these points. It defines an UPDATE trigger that checks the new value to be used for updating each row, and modifies the value to be within the range from 0 to 100. This must be a BEFORE trigger because the value must be checked before it is used to update the row:

mysql> delimiter //

mysql> CREATE TRIGGER upd\_check BEFORE UPDATE ON account

FOR EACH ROW

BEGIN

IF NEW.amount < 0 THEN

SET NEW.amount = 0;

ELSEIF NEW.amount > 100 THEN

SET NEW.amount = 100;

END IF;

END;//

mysql> delimiter ;

It can be easier to define a stored procedure separately and then invoke it from the trigger using a simple [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statement. This is also advantageous if you want to execute the same code from within several triggers.

There are limitations on what can appear in statements that a trigger executes when activated:

* The trigger cannot use the [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statement to invoke stored procedures that return data to the client or that use dynamic SQL. (Stored procedures are permitted to return data to the trigger through OUT or INOUT parameters.)
* The trigger cannot use statements that explicitly or implicitly begin or end a transaction, such as [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html), [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html), or [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html). ([ROLLBACK to SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/commit.html) is permitted because it does not end a transaction.).

MySQL handles errors during trigger execution as follows:

* If a BEFORE trigger fails, the operation on the corresponding row is not performed.
* A BEFORE trigger is activated by the attempt to insert or modify the row, regardless of whether the attempt subsequently succeeds.
* An AFTER trigger is executed only if any BEFORE triggers and the row operation execute successfully.
* An error during either a BEFORE or AFTER trigger results in failure of the entire statement that caused trigger invocation.
* For transactional tables, failure of a statement should cause rollback of all changes performed by the statement. Failure of a trigger causes the statement to fail, so trigger failure also causes rollback. For nontransactional tables, such rollback cannot be done, so although the statement fails, any changes performed prior to the point of the error remain in effect.

Triggers can contain direct references to tables by name, such as the trigger named testref shown in this example:

CREATE TABLE test1(a1 INT);

CREATE TABLE test2(a2 INT);

CREATE TABLE test3(a3 INT NOT NULL AUTO\_INCREMENT PRIMARY KEY);

CREATE TABLE test4(

a4 INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

b4 INT DEFAULT 0

);

delimiter |

CREATE TRIGGER testref BEFORE INSERT ON test1

FOR EACH ROW

BEGIN

INSERT INTO test2 SET a2 = NEW.a1;

DELETE FROM test3 WHERE a3 = NEW.a1;

UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;

END;

|

delimiter ;

INSERT INTO test3 (a3) VALUES

(NULL), (NULL), (NULL), (NULL), (NULL),

(NULL), (NULL), (NULL), (NULL), (NULL);

INSERT INTO test4 (a4) VALUES

(0), (0), (0), (0), (0), (0), (0), (0), (0), (0);

Suppose that you insert the following values into table test1 as shown here:

mysql> INSERT INTO test1 VALUES

(1), (3), (1), (7), (1), (8), (4), (4);

Query OK, 8 rows affected (0.01 sec)

Records: 8 Duplicates: 0 Warnings: 0

As a result, the four tables contain the following data:

mysql> SELECT \* FROM test1;

+------+

| a1 |

+------+

| 1 |

| 3 |

| 1 |

| 7 |

| 1 |

| 8 |

| 4 |

| 4 |

+------+

8 rows in set (0.00 sec)

mysql> SELECT \* FROM test2;

+------+

| a2 |

+------+

| 1 |

| 3 |

| 1 |

| 7 |

| 1 |

| 8 |

| 4 |

| 4 |

+------+

8 rows in set (0.00 sec)

mysql> SELECT \* FROM test3;

+----+

| a3 |

+----+

| 2 |

| 5 |

| 6 |

| 9 |

| 10 |

+----+

5 rows in set (0.00 sec)

mysql> SELECT \* FROM test4;

+----+------+

| a4 | b4 |

+----+------+

| 1 | 3 |

| 2 | 0 |

| 3 | 1 |

| 4 | 2 |

| 5 | 0 |

| 6 | 0 |

| 7 | 1 |

| 8 | 1 |

| 9 | 0 |

| 10 | 0 |

+----+------+

10 rows in set (0.00 sec)

.

## Using the Event Scheduler

The MySQL Event Scheduler manages the scheduling and execution of events, that is, tasks that run according to a schedule.

Stored routines require the events data dictionary table in the mysql system database. This table is created during the MySQL 8.0 installation procedure. If you are upgrading to MySQL 8.0 from an earlier version, be sure to perform the upgrade procedure to make sure that your system database is up to date.

### Event Scheduler Overview

MySQL Events are tasks that run according to a schedule. Therefore, we sometimes refer to them as scheduled events. When you create an event, you are creating a named database object containing one or more SQL statements to be executed at one or more regular intervals, beginning and ending at a specific date and time. Conceptually, this is similar to the idea of the Unix crontab (also known as a “cron job”) or the Windows Task Scheduler.

Scheduled tasks of this type are also sometimes known as “temporal triggers”, implying that these are objects that are triggered by the passage of time. While this is essentially correct, we prefer to use the term events to avoid confusion with triggers of the type discussed in  [“Using Triggers”](https://dev.mysql.com/doc/refman/8.0/en/triggers.html). Events should more specifically not be confused with “temporary triggers”. Whereas a trigger is a database object whose statements are executed in response to a specific type of event that occurs on a given table, a (scheduled) event is an object whose statements are executed in response to the passage of a specified time interval.

While there is no provision in the SQL Standard for event scheduling, there are precedents in other database systems, and you may notice some similarities between these implementations and that found in the MySQL Server.

MySQL Events have the following major features and properties:

* In MySQL, an event is uniquely identified by its name and the schema to which it is assigned.
* An event performs a specific action according to a schedule. This action consists of an SQL statement, which can be a compound statement in a [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) block if desired . An event's timing can be either one-tie or recurrent. A one-time event executes one time only. A recurrent event repeats its action at a regular interval, and the schedule for a recurring event can be assigned a specific start day and time, end day and time, both, or neither. (By default, a recurring event's schedule begins as soon as it is created, and continues indefinitely, until it is disabled or dropped.)

If a repeating event does not terminate within its scheduling interval, the result may be multiple instances of the event executing simultaneously. If this is undesirable, you should institute a mechanism to prevent simultaneous instances. For example, you could use the [GET\_LOCK()](https://dev.mysql.com/doc/refman/8.0/en/locking-functions.html#function_get-lock) function, or row or table locking.

* Users can create, modify, and drop scheduled events using SQL statements intended for these purposes. Syntactically invalid event creation and modification statements fail with an appropriate error message. A user may include statements in an event's action which require privileges that the user does not actually have. The event creation or modification statement succeeds but the event's action fails..
* Many of the properties of an event can be set or modified using SQL statements. These properties include the event's name, timing, persistence (that is, whether it is preserved following the expiration of its schedule), status (enabled or disabled), action to be performed, and the schema to which it is assigned..

The default definer of an event is the user who created the event, unless the event has been altered, in which case the definer is the user who issued the last [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html) statement affecting that event. An event can be modified by any user having the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege on the database for which the event is defined..

* An event's action statement may include most SQL statements permitted within stored routines..

### Event Scheduler Configuration

Events are executed by a special event scheduler thread; when we refer to the Event Scheduler, we actually refer to this thread. When running, the event scheduler thread and its current state can be seen by users having the [PROCESS](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_process) privilege in the output of [SHOW PROCESSLIST](https://dev.mysql.com/doc/refman/8.0/en/show-processlist.html), as shown in the discussion that follows.

The global [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) system variable determines whether the Event Scheduler is enabled and running on the server. It has one of these 3 values, which affect event scheduling as described here. The default is ON.

* ON: The Event Scheduler is started; the event scheduler thread runs and executes all scheduled events.

When the Event Scheduler is ON, the event scheduler thread is listed in the output of [SHOW PROCESSLIST](https://dev.mysql.com/doc/refman/8.0/en/show-processlist.html) as a daemon process, and its state is represented as shown here:

mysql> SHOW PROCESSLIST\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Id: 1

User: root

Host: localhost

db: NULL

Command: Query

Time: 0

State: NULL

Info: show processlist

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Id: 2

User: event\_scheduler

Host: localhost

db: NULL

Command: Daemon

Time: 3

State: Waiting for next activation

Info: NULL

2 rows in set (0.00 sec)

Event scheduling can be stopped by setting the value of [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) to OFF.

* OFF: The Event Scheduler is stopped. The event scheduler thread does not run, is not shown in the output of [SHOW PROCESSLIST](https://dev.mysql.com/doc/refman/8.0/en/show-processlist.html), and no scheduled events are executed.

When the Event Scheduler is stopped ([event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) is OFF), it can be started by setting the value of [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) to ON. (See next item.)

* DISABLED: This value renders the Event Scheduler nonoperational. When the Event Scheduler is DISABLED, the event scheduler thread does not run (and so does not appear in the output of [SHOW PROCESSLIST](https://dev.mysql.com/doc/refman/8.0/en/show-processlist.html)). In addition, the Event Scheduler state cannot be changed at runtime.

If the Event Scheduler status has not been set to DISABLED, [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) can be toggled between ON and OFF (using [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html)). It is also possible to use 0 for OFF, and 1 for ON when setting this variable. Thus, any of the following 4 statements can be used in the [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) client to turn on the Event Scheduler:

SET GLOBAL event\_scheduler = ON;

SET @@GLOBAL.event\_scheduler = ON;

SET GLOBAL event\_scheduler = 1;

SET @@GLOBAL.event\_scheduler = 1;

Similarly, any of these 4 statements can be used to turn off the Event Scheduler:

SET GLOBAL event\_scheduler = OFF;

SET @@GLOBAL.event\_scheduler = OFF;

SET GLOBAL event\_scheduler = 0;

SET @@GLOBAL.event\_scheduler = 0;

Although ON and OFF have numeric equivalents, the value displayed for [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) by [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) or [SHOW VARIABLES](https://dev.mysql.com/doc/refman/8.0/en/show-variables.html) is always one of OFF, ON, or DISABLED. *DISABLED* has no numeric equivalent. For this reason, ON and OFF are usually preferred over 1 and 0 when setting this variable.

Note that attempting to set [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) without specifying it as a global variable causes an error:

mysql< SET @@event\_scheduler = OFF;

ERROR 1229 (HY000): Variable 'event\_scheduler' is a GLOBAL

variable and should be set with SET GLOBAL

**Important**

It is possible to set the Event Scheduler to DISABLED only at server startup. If [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) is ON or OFF, you cannot set it to DISABLED at runtime. Also, if the Event Scheduler is set to DISABLED at startup, you cannot change the value of [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) at runtime.

To disable the event scheduler, use one of the following two methods:

* As a command-line option when starting the server:

--event-scheduler=DISABLED

* In the server configuration file (my.cnf, or my.ini on Windows systems), include the line where it can be read by the server (for example, in a [mysqld] section):

event\_scheduler=DISABLED

To enable the Event Scheduler, restart the server without the [--event-scheduler=DISABLED](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) command-line option, or after removing or commenting out the line containing [event-scheduler=DISABLED](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) in the server configuration file, as appropriate. Alternatively, you can use ON (or 1) or OFF (or 0) in place of the DISABLED value when starting the server.

**Note**

You can issue event-manipulation statements when [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) is set to DISABLED. No warnings or errors are generated in such cases (provided that the statements are themselves valid). However, scheduled events cannot execute until this variable is set to ON (or 1). Once this has been done, the event scheduler thread executes all events whose scheduling conditions are satisfied.

Starting the MySQL server with the [--skip-grant-tables](https://dev.mysql.com/doc/refman/8.0/en/server-options.html#option_mysqld_skip-grant-tables) option causes [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) to be set to DISABLED, overriding any other value set either on the command line or in the my.cnf or my.ini file (Bug #26807).

### Event Syntax

MySQL provides several SQL statements for working with scheduled events:

* New events are defined using the [CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/create-event.html) statement.
* The definition of an existing event can be changed by means of the [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html) statement..
* When a scheduled event is no longer wanted or needed, it can be deleted from the server by its definer using the [DROP EVENT](https://dev.mysql.com/doc/refman/8.0/en/drop-event.html) statement.. Whether an event persists past the end of its schedule also depends on its ON COMPLETION clause, if it has one.

An event can be dropped by any user having the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege for the database on which the event is defined..

### Event Metadata

To obtain metadata about events:

* Query the [EVENTS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-events-table.html) table of the INFORMATION\_SCHEMA database..
* Use the [SHOW CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/show-create-event.html) statement..
* Use the [SHOW EVENTS](https://dev.mysql.com/doc/refman/8.0/en/show-events.html) statement..

***Event Scheduler Time Representation***

Each session in MySQL has a session time zone (STZ). This is the session [time\_zone](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_time_zone) value that is initialized from the server's global [time\_zone](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_time_zone) value when the session begins but may be changed during the session.

The session time zone that is current when a [CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/create-event.html) or [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html) statement executes is used to interpret times specified in the event definition. This becomes the event time zone (ETZ); that is, the time zone that is used for event scheduling and is in effect within the event as it executes.

For representation of event information in the data dictionary, the execute\_at, starts, and ends times are converted to UTC and stored along with the event time zone. This enables event execution to proceed as defined regardless of any subsequent changes to the server time zone or daylight saving time effects. The last\_executed time is also stored in UTC.

Event times can be obtained by selecting from the [INFORMATION\_SCHEMA.EVENTS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-events-table.html) table or from [SHOW EVENTS](https://dev.mysql.com/doc/refman/8.0/en/show-events.html), but they are reported as ETZ or STZ values. The following table summarizes representation of event times.

| **Value** | [INFORMATION\_SCHEMA.EVENTS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-events-table.html) | [SHOW EVENTS](https://dev.mysql.com/doc/refman/8.0/en/show-events.html) |
| --- | --- | --- |
| **Execute at** | ETZ | ETZ |
| **Starts** | ETZ | ETZ |
| **Ends** | ETZ | ETZ |
| **Last executed** | ETZ | n/a |
| **Created** | STZ | n/a |
| **Last altered** | STZ | n/a |

### Event Scheduler Status

The Event Scheduler writes information about event execution that terminates with an error or warning to the MySQL Server's error log..

To obtain information about the state of the Event Scheduler for debugging and troubleshooting purposes, run [**mysqladmin debug**](https://dev.mysql.com/doc/refman/8.0/en/mysqladmin.html) ; after running this command, the server's error log contains output relating to the Event Scheduler, similar to what is shown here:

Events status:

LLA = Last Locked At LUA = Last Unlocked At

WOC = Waiting On Condition DL = Data Locked

Event scheduler status:

State : INITIALIZED

Thread id : 0

LLA : n/a:0

LUA : n/a:0

WOC : NO

Workers : 0

Executed : 0

Data locked: NO

Event queue status:

Element count : 0

Data locked : NO

Attempting lock : NO

LLA : init\_queue:95

LUA : init\_queue:103

WOC : NO

Next activation : never

In statements that occur as part of events executed by the Event Scheduler, diagnostics messages (not only errors, but also warnings) are written to the error log, and, on Windows, to the application event log. For frequently executed events, it is possible for this to result in many logged messages. For example, for SELECT ... INTO ***var\_list*** statements, if the query returns no rows, a warning with error code 1329 occurs (No data), and the variable values remain unchanged. If the query returns multiple rows, error 1172 occurs (Result consisted of more than one row).. For statements that may retrieve multiple rows, another strategy is to use LIMIT 1 to limit the result set to a single row.

### The Event Scheduler and MySQL Privileges

To enable or disable the execution of scheduled events, it is necessary to set the value of the global [event\_scheduler](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_event_scheduler) system variable. This requires privileges sufficient to set global system variables..

The [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege governs the creation, modification, and deletion of events. This privilege can be bestowed using [GRANT](https://dev.mysql.com/doc/refman/8.0/en/grant.html). For example, this [GRANT](https://dev.mysql.com/doc/refman/8.0/en/grant.html) statement confers the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege for the schema named myschema on the user jon@ghidora:

GRANT EVENT ON myschema.\* TO jon@ghidora;

(We assume that this user account already exists, and that we wish for it to remain unchanged otherwise.)

To grant this same user the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege on all schemas, use the following statement:

GRANT EVENT ON \*.\* TO jon@ghidora;

The [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege has global or schema-level scope. Therefore, trying to grant it on a single table results in an error as shown:

mysql> GRANT EVENT ON myschema.mytable TO jon@ghidora;

ERROR 1144 (42000): Illegal GRANT/REVOKE command; please

consult the manual to see which privileges can be used

It is important to understand that an event is executed with the privileges of its definer, and that it cannot perform any actions for which its definer does not have the requisite privileges. For example, suppose that jon@ghidora has the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege for myschema. Suppose also that this user has the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) privilege for myschema, but no other privileges for this schema. It is possible for jon@ghidora to create a new event such as this one:

CREATE EVENT e\_store\_ts

ON SCHEDULE

EVERY 10 SECOND

DO

INSERT INTO myschema.mytable VALUES (UNIX\_TIMESTAMP());

The user waits for a minute or so, and then performs a SELECT \* FROM mytable; query, expecting to see several new rows in the table. Instead, the table is empty. Since the user does not have the [INSERT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_insert) privilege for the table in question, the event has no effect.

If you inspect the MySQL error log (***hostname***.err), you can see that the event is executing, but the action it is attempting to perform fails:

2013-09-24T12:41:31.261992Z 25 [ERROR] Event Scheduler:

[jon@ghidora][cookbook.e\_store\_ts] INSERT command denied to user

'jon'@'ghidora' for table 'mytable'

2013-09-24T12:41:31.262022Z 25 [Note] Event Scheduler:

[jon@ghidora].[myschema.e\_store\_ts] event execution failed.

2013-09-24T12:41:41.271796Z 26 [ERROR] Event Scheduler:

[jon@ghidora][cookbook.e\_store\_ts] INSERT command denied to user

'jon'@'ghidora' for table 'mytable'

2013-09-24T12:41:41.272761Z 26 [Note] Event Scheduler:

[jon@ghidora].[myschema.e\_store\_ts] event execution failed.

Since this user very likely does not have access to the error log, it is possible to verify whether the event's action statement is valid by executing it directly:

mysql> INSERT INTO myschema.mytable VALUES (UNIX\_TIMESTAMP());

ERROR 1142 (42000): INSERT command denied to user

'jon'@'ghidora' for table 'mytable'

Inspection of the [INFORMATION\_SCHEMA.EVENTS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-events-table.html) table shows that e\_store\_ts exists and is enabled, but its LAST\_EXECUTED column is NULL:

mysql> SELECT \* FROM INFORMATION\_SCHEMA.EVENTS

> WHERE EVENT\_NAME='e\_store\_ts'

> AND EVENT\_SCHEMA='myschema'\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EVENT\_CATALOG: NULL

EVENT\_SCHEMA: myschema

EVENT\_NAME: e\_store\_ts

DEFINER: jon@ghidora

EVENT\_BODY: SQL

EVENT\_DEFINITION: INSERT INTO myschema.mytable VALUES (UNIX\_TIMESTAMP())

EVENT\_TYPE: RECURRING

EXECUTE\_AT: NULL

INTERVAL\_VALUE: 5

INTERVAL\_FIELD: SECOND

SQL\_MODE: NULL

STARTS: 0000-00-00 00:00:00

ENDS: 0000-00-00 00:00:00

STATUS: ENABLED

ON\_COMPLETION: NOT PRESERVE

CREATED: 2006-02-09 22:36:06

LAST\_ALTERED: 2006-02-09 22:36:06

LAST\_EXECUTED: NULL

EVENT\_COMMENT:

1 row in set (0.00 sec)

To rescind the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege, use the [REVOKE](https://dev.mysql.com/doc/refman/8.0/en/revoke.html) statement. In this example, the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege on the schema myschema is removed from the jon@ghidora user account:

REVOKE EVENT ON myschema.\* FROM jon@ghidora;

**Important**

Revoking the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege from a user does not delete or disable any events that may have been created by that user.

An event is not migrated or dropped as a result of renaming or dropping the user who created it.

Suppose that the user jon@ghidora has been granted the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) and [INSERT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_insert) privileges on the myschema schema. This user then creates the following event:

CREATE EVENT e\_insert

ON SCHEDULE

EVERY 7 SECOND

DO

INSERT INTO myschema.mytable;

After this event has been created, root revokes the [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege for jon@ghidora. However, e\_insert continues to execute, inserting a new row into mytable each seven seconds. The same would be true if root had issued either of these statements:

* DROP USER jon@ghidora;
* RENAME USER jon@ghidora TO someotherguy@ghidora;

You can verify that this is true by examining the [INFORMATION\_SCHEMA.EVENTS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-events-table.html) table before and after issuing a [DROP USER](https://dev.mysql.com/doc/refman/8.0/en/drop-user.html) or [RENAME USER](https://dev.mysql.com/doc/refman/8.0/en/rename-user.html) statement.

Event definitions are stored in the data dictionary. To drop an event created by another user account, you must be the MySQL root user or another user with the necessary privileges.

Users' [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privileges are stored in the Event\_priv columns of the mysql.user and mysql.db tables. In both cases, this column holds one of the values 'Y' or 'N'. 'N' is the default. mysql.user.Event\_priv is set to 'Y' for a given user only if that user has the global [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege (that is, if the privilege was bestowed using GRANT EVENT ON \*.\*). For a schema-level [EVENT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_event) privilege, [GRANT](https://dev.mysql.com/doc/refman/8.0/en/grant.html) creates a row in mysql.db and sets that row's Db column to the name of the schema, the User column to the name of the user, and the Event\_priv column to 'Y'. There should never be any need to manipulate these tables directly, since the [GRANT EVENT](https://dev.mysql.com/doc/refman/8.0/en/grant.html) and REVOKE EVENT statements perform the required operations on them.

Five status variables provide counts of event-related operations. These are:

* Com\_create\_event: The number of [CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/create-event.html) statements executed since the last server restart.
* Com\_alter\_event: The number of [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html) statements executed since the last server restart.
* Com\_drop\_event: The number of [DROP EVENT](https://dev.mysql.com/doc/refman/8.0/en/drop-event.html) statements executed since the last server restart.
* Com\_show\_create\_event: The number of [SHOW CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/show-create-event.html) statements executed since the last server restart.
* Com\_show\_events: The number of [SHOW EVENTS](https://dev.mysql.com/doc/refman/8.0/en/show-events.html) statements executed since the last server restart.

You can view current values for all of these at one time by running the statement SHOW STATUS LIKE '%event%';.

## Using Views

MySQL supports views, including updatable views. Views are stored queries that when invoked produce a result set. A view acts as a virtual table.

The following discussion describes the syntax for creating and dropping views, and shows some examples of how to use them.

### View Syntax

The [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) statement creates a new view. To alter the definition of a view or drop a view, use [ALTER VIEW](https://dev.mysql.com/doc/refman/8.0/en/alter-view.html) , or [DROP VIEW](https://dev.mysql.com/doc/refman/8.0/en/drop-view.html) .

A view can be created from many kinds of [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statements. It can refer to base tables or other views. It can use joins, [UNION](https://dev.mysql.com/doc/refman/8.0/en/union.html), and subqueries. The [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) need not even refer to any tables. The following example defines a view that selects two columns from another table, as well as an expression calculated from those columns:

mysql> CREATE TABLE t (qty INT, price INT);

mysql> INSERT INTO t VALUES(3, 50), (5, 60);

mysql> CREATE VIEW v AS SELECT qty, price, qty\*price AS value FROM t;

mysql> SELECT \* FROM v;

+------+-------+-------+

| qty | price | value |

+------+-------+-------+

| 3 | 50 | 150 |

| 5 | 60 | 300 |

+------+-------+-------+

mysql> SELECT \* FROM v WHERE qty = 5;

+------+-------+-------+

| qty | price | value |

+------+-------+-------+

| 5 | 60 | 300 |

+------+-------+-------+

### View Processing Algorithms

The optional ALGORITHM clause for [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) or [ALTER VIEW](https://dev.mysql.com/doc/refman/8.0/en/alter-view.html) is a MySQL extension to standard SQL. It affects how MySQL processes the view. ALGORITHM takes three values: MERGE, TEMPTABLE, or UNDEFINED.

* For MERGE, the text of a statement that refers to the view and the view definition are merged such that parts of the view definition replace corresponding parts of the statement.
* For TEMPTABLE, the results from the view are retrieved into a temporary table, which then is used to execute the statement.
* For UNDEFINED, MySQL chooses which algorithm to use. It prefers MERGE over TEMPTABLE if possible, because MERGE is usually more efficient and because a view cannot be updatable if a temporary table is used.
* If no ALGORITHM clause is present, the default algorithm is determined by the value of the [derived\_merge](https://dev.mysql.com/doc/refman/8.0/en/switchable-optimizations.html#optflag_derived-merge) flag of the [optimizer\_switch](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_optimizer_switch) system variable..

A reason to specify TEMPTABLE explicitly is that locks can be released on underlying tables after the temporary table has been created and before it is used to finish processing the statement. This might result in quicker lock release than the MERGE algorithm so that other clients that use the view are not blocked as long.

A view algorithm can be UNDEFINED for three reasons:

* No ALGORITHM clause is present in the [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) statement.
* The [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) statement has an explicit ALGORITHM = UNDEFINED clause.
* ALGORITHM = MERGE is specified for a view that can be processed only with a temporary table. In this case, MySQL generates a warning and sets the algorithm to UNDEFINED.

As mentioned earlier, MERGE is handled by merging corresponding parts of a view definition into the statement that refers to the view. The following examples briefly illustrate how the MERGE algorithm works. The examples assume that there is a view v\_merge that has this definition:

CREATE ALGORITHM = MERGE VIEW v\_merge (vc1, vc2) AS

SELECT c1, c2 FROM t WHERE c3 > 100;

Example 1: Suppose that we issue this statement:

SELECT \* FROM v\_merge;

MySQL handles the statement as follows:

* v\_merge becomes t
* \* becomes vc1, vc2, which corresponds to c1, c2
* The view WHERE clause is added

The resulting statement to be executed becomes:

SELECT c1, c2 FROM t WHERE c3 > 100;

Example 2: Suppose that we issue this statement:

SELECT \* FROM v\_merge WHERE vc1 < 100;

This statement is handled similarly to the previous one, except that vc1 < 100 becomes c1 < 100 and the view WHERE clause is added to the statement WHERE clause using an [AND](https://dev.mysql.com/doc/refman/8.0/en/logical-operators.html#operator_and) connective (and parentheses are added to make sure the parts of the clause are executed with correct precedence). The resulting statement to be executed becomes:

SELECT c1, c2 FROM t WHERE (c3 > 100) AND (c1 < 100);

Effectively, the statement to be executed has a WHERE clause of this form:

WHERE (select WHERE) AND (view WHERE)

If the MERGE algorithm cannot be used, a temporary table must be used instead. Constructs that prevent merging are the same as those that prevent merging in derived tables and common table expressions. Examples are SELECT DISTINCT or LIMIT in the subquery..

### Updatable and Insertable Views

Some views are updatable and references to them can be used to specify tables to be updated in data change statements. That is, you can use them in statements such as [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html), [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html), or [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) to update the contents of the underlying table. Derived tables and common table expressions can also be specified in multiple-table [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) and [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html) statements, but can only be used for reading data to specify rows to be updated or deleted. Generally, the view references must be updatable, meaning that they may be merged and not materialized. Composite views have more complex rules.

For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view nonupdatable. To be more specific, a view is not updatable if it contains any of the following:

* Aggregate functions or window functions ([SUM()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_sum), [MIN()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_min), [MAX()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_max), [COUNT()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_count), and so forth)
* DISTINCT
* GROUP BY
* HAVING
* [UNION](https://dev.mysql.com/doc/refman/8.0/en/union.html) or [UNION ALL](https://dev.mysql.com/doc/refman/8.0/en/union.html)
* Subquery in the select list

Nondependent subqueries in the select list fail for [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html), but are okay for [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html), [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html). For dependent subqueries in the select list, no data change statements are permitted.

* Certain joins (see additional join discussion later in this section)
* Reference to nonupdatable view in the FROM clause
* Subquery in the WHERE clause that refers to a table in the FROM clause
* Refers only to literal values (in this case, there is no underlying table to update)
* ALGORITHM = TEMPTABLE (use of a temporary table always makes a view nonupdatable)
* Multiple references to any column of a base table (fails for [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html), okay for [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html), [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html))

A generated column in a view is considered updatable because it is possible to assign to it. However, if such a column is updated explicitly, the only permitted value is DEFAULT..

It is sometimes possible for a multiple-table view to be updatable, assuming that it can be processed with the MERGE algorithm. For this to work, the view must use an inner join (not an outer join or a [UNION](https://dev.mysql.com/doc/refman/8.0/en/union.html)). Also, only a single table in the view definition can be updated, so the SET clause must name only columns from one of the tables in the view. Views that use [UNION ALL](https://dev.mysql.com/doc/refman/8.0/en/union.html) are not permitted even though they might be theoretically updatable.

With respect to insertability (being updatable with [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statements), an updatable view is insertable if it also satisfies these additional requirements for the view columns:

* There must be no duplicate view column names.
* The view must contain all columns in the base table that do not have a default value.
* The view columns must be simple column references. They must not be expressions, such as these:
* 3.14159
* col1 + 3
* UPPER(col2)
* col3 / col4

(subquery)

MySQL sets a flag, called the view updatability flag, at [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) time. The flag is set to YES (true) if [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) and [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html) (and similar operations) are legal for the view. Otherwise, the flag is set to NO (false). The IS\_UPDATABLE column in the [INFORMATION\_SCHEMA.VIEWS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-views-table.html) table displays the status of this flag. It means that the server always knows whether a view is updatable.

If a view is not updatable, statements such [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html), [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html), and [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) are illegal and are rejected. (Even if a view is updatable, it might not be possible to insert into it, as described elsewhere in this section.)

The updatability of views may be affected by the value of the [updatable\_views\_with\_limit](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_updatable_views_with_limit) system variable. See [Section 5.1.8, “Server System Variables”](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html).

For the following discussion, suppose that these tables and views exist:

CREATE TABLE t1 (x INTEGER);

CREATE TABLE t2 (c INTEGER);

CREATE VIEW vmat AS SELECT SUM(x) AS s FROM t1;

CREATE VIEW vup AS SELECT \* FROM t2;

CREATE VIEW vjoin AS SELECT \* FROM vmat JOIN vup ON vmat.s=vup.c;

[INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html), [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html), and [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html) statements are permitted as follows:

* [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html): The insert table of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement may be a view reference that is merged. If the view is a join view, all components of the view must be updatable (not materialized). For a multiple-table updatable view, [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) can work if it inserts into a single table.

This statement is invalid because one component of the join view is nonupdatable:

INSERT INTO vjoin (c) VALUES (1);

This statement is valid; the view contains no materialized components:

INSERT INTO vup (c) VALUES (1);

* [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html): The table or tables to be updated in an [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) statement may be view references that are merged. If a view is a join view, at least one component of the view must be updatable (this differs from [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html)).

In a multiple-table [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) statement, the updated table references of the statement must be base tables or updatable view references. Nonupdated table references may be materialized views or derived tables.

This statement is valid; column c is from the updatable part of the join view:

UPDATE vjoin SET c=c+1;

This statement is invalid; column x is from the nonupdatable part:

UPDATE vjoin SET x=x+1;

This statement is valid; the updated table reference of the multiple-table [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) is an updatable view (vup):

UPDATE vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...

SET c=c+1;

This statement is invalid; it tries to update a materialized derived table:

UPDATE vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...

SET s=s+1;

* [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html): The table or tables to be deleted from in a [DELETE](https://dev.mysql.com/doc/refman/8.0/en/delete.html) statement must be merged views. Join views are not allowed (this differs from [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) and [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html)).

This statement is invalid because the view is a join view:

DELETE vjoin WHERE ...;

This statement is valid because the view is a merged (updatable) view:

DELETE vup WHERE ...;

This statement is valid because it deletes from a merged (updatable) view:

DELETE vup FROM vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...;

Additional discussion and examples follow.

Earlier discussion in this section pointed out that a view is not insertable if not all columns are simple column references (for example, if it contains columns that are expressions or composite expressions). Although such a view is not insertable, it can be updatable if you update only columns that are not expressions. Consider this view:

CREATE VIEW v AS SELECT col1, 1 AS col2 FROM t;

This view is not insertable because col2 is an expression. But it is updatable if the update does not try to update col2. This update is permissible:

UPDATE v SET col1 = 0;

This update is not permissible because it attempts to update an expression column:

UPDATE v SET col2 = 0;

If a table contains an AUTO\_INCREMENT column, inserting into an insertable view on the table that does not include the AUTO\_INCREMENT column does not change the value of [LAST\_INSERT\_ID()](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_last-insert-id), because the side effects of inserting default values into columns not part of the view should not be visible.

### The View WITH CHECK OPTION Clause

The WITH CHECK OPTION clause can be given for an updatable view to prevent inserts to rows for which the WHERE clause in the ***select\_statement*** is not true. It also prevents updates to rows for which the WHERE clause is true but the update would cause it to be not true (in other words, it prevents visible rows from being updated to nonvisible rows).

In a WITH CHECK OPTION clause for an updatable view, the LOCAL and CASCADED keywords determine the scope of check testing when the view is defined in terms of another view. When neither keyword is given, the default is CASCADED.

WITH CHECK OPTION testing is standard-compliant:

* With LOCAL, the view WHERE clause is checked, then checking recurses to underlying views and applies the same rules.
* With CASCADED, the view WHERE clause is checked, then checking recurses to underlying views, adds WITH CASCADED CHECK OPTION to them (for purposes of the check; their definitions remain unchanged), and applies the same rules.
* With no check option, the view WHERE clause is not checked, then checking recurses to underlying views, and applies the same rules.

Consider the definitions for the following table and set of views:

CREATE TABLE t1 (a INT);

CREATE VIEW v1 AS SELECT \* FROM t1 WHERE a < 2

WITH CHECK OPTION;

CREATE VIEW v2 AS SELECT \* FROM v1 WHERE a > 0

WITH LOCAL CHECK OPTION;

CREATE VIEW v3 AS SELECT \* FROM v1 WHERE a > 0

WITH CASCADED CHECK OPTION;

Here the v2 and v3 views are defined in terms of another view, v1.

Inserts for v2 are checked against its LOCAL check option, then the check recurses to v1 and the rules are applied again. The rules for v1 cause a check failure. The check for v3 also fails:

mysql> INSERT INTO v2 VALUES (2);

ERROR 1369 (HY000): CHECK OPTION failed 'test.v2'

mysql> INSERT INTO v3 VALUES (2);

ERROR 1369 (HY000): CHECK OPTION failed 'test.v3'

## Stored Object Access Control

Stored programs (procedures, functions, triggers, and events) and views are defined prior to use and, when referenced, execute within a security context that determines their privileges. The privileges applicable to execution of a stored object are controlled by its DEFINER attribute and SQL SECURITY characteristic.

### The DEFINER Attribute

A stored object definition can include a DEFINER attribute that names a MySQL account. If a definition omits the DEFINER attribute, the default object definer is the user who creates it.

The following rules determine which accounts you can specify as the DEFINER attribute for a stored object:

* If you have the [SET\_USER\_ID](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_set-user-id) privilege (or the deprecated [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege), you can specify any account as the DEFINER attribute. If the account does not exist, a warning is generated. Additionally, to set a stored object DEFINER attribute to an account that has the [SYSTEM\_USER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_system-user) privilege, you must have the [SYSTEM\_USER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_system-user) privilege.
* Otherwise, the only permitted account is your own, specified either literally or as [CURRENT\_USER](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_current-user) or [CURRENT\_USER()](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_current-user). You cannot set the definer to any other account.

Creating a stored object with a nonexistent DEFINER account creates an orphan object, which may have negative consequences; see [Orphan Stored Objects](https://dev.mysql.com/doc/refman/8.0/en/stored-objects-security.html#stored-objects-security-orphan-objects).

### The SQL SECURITY Characteristic

For stored routines (procedures and functions) and views, the object definition can include an SQL SECURITY characteristic with a value of DEFINER or INVOKER to specify whether the object executes in definer or invoker context. If the definition omits the SQL SECURITY characteristic, the default is definer context.

Triggers and events have no SQL SECURITY characteristic and always execute in definer context. The server invokes these objects automatically as necessary, so there is no invoking user.

Definer and invoker security contexts differ as follows:

* A stored object that executes in definer security context executes with the privileges of the account named by its DEFINER attribute. These privileges may be entirely different from those of the invoking user. The invoker must have appropriate privileges to reference the object (for example, [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) to call a stored procedure or [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) to select from a view), but during object execution, the invoker's privileges are ignored and only the DEFINER account privileges matter. If the DEFINER account has few privileges, the object is correspondingly limited in the operations it can perform. If the DEFINER account is highly privileged (such as an administrative account), the object can perform powerful operations no matter who invokes it.
* A stored routine or view that executes in invoker security context can perform only operations for which the invoker has privileges. The DEFINER attribute has no effect on object execution.

### Examples

Consider the following stored procedure, which is declared with SQL SECURITY DEFINER to execute in definer security context:

CREATE DEFINER = 'admin'@'localhost' PROCEDURE p1()

SQL SECURITY DEFINER

BEGIN

UPDATE t1 SET counter = counter + 1;

END;

Any user who has the [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) privilege for p1 can invoke it with a [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statement. However, when p1 executes, it does so in definer security context and thus executes with the privileges of 'admin'@'localhost', the account named as its DEFINER attribute. This account must have the [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) privilege for p1 as well as the [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_update) privilege for the table t1 referenced within the object body. Otherwise, the procedure fails.

Now consider this stored procedure, which is identical to p1 except that its SQL SECURITY characteristic is INVOKER:

CREATE DEFINER = 'admin'@'localhost' PROCEDURE p2()

SQL SECURITY INVOKER

BEGIN

UPDATE t1 SET counter = counter + 1;

END;

Unlike p1, p2 executes in invoker security context and thus with the privileges of the invoking user regardless of the DEFINER attribute value. p2 fails if the invoker lacks the [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) privilege for p2 or the [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_update) privilege for the table t1.

### Orphan Stored Objects

An orphan stored object is one for which its DEFINER attribute names a nonexistent account:

* An orphan stored object can be created by specifying a nonexistent DEFINER account at object-creation time.
* An existing stored object can become orphaned through execution of a [DROP USER](https://dev.mysql.com/doc/refman/8.0/en/drop-user.html) statement that drops the object DEFINER account, or a [RENAME USER](https://dev.mysql.com/doc/refman/8.0/en/rename-user.html) statement that renames the object DEFINER account.

An orphan stored object may be problematic in these ways:

* Because the DEFINER account does not exist, the object may not work as expected if it executes in definer security context:
  + For a stored routine, an error occurs at routine execution time if the SQL SECURITY value is DEFINER but the definer account does not exist.
  + For a trigger, it is not a good idea for trigger activation to occur until the account actually does exist. Otherwise, the behavior with respect to privilege checking is undefined.
  + For an event, an error occurs at event execution time if the account does not exist.
  + For a view, an error occurs when the view is referenced if the SQL SECURITY value is DEFINER but the definer account does not exist.
* The object may present a security risk if the nonexistent DEFINER account is subsequently re-created for a purpose unrelated to the object. In this case, the account “adopts” the object and, with the appropriate privileges, is able to execute it even if that is not intended.

As of MySQL 8.0.22, the server imposes additional account-management security checks designed to prevent operations that (perhaps inadvertently) cause stored objects to become orphaned or that cause adoption of stored objects that are currently orphaned:

* [DROP USER](https://dev.mysql.com/doc/refman/8.0/en/drop-user.html) fails with an error if any account to be dropped is named as the DEFINER attribute for any stored object. (That is, the statement fails if dropping an account would cause a stored object to become orphaned.)
* [RENAME USER](https://dev.mysql.com/doc/refman/8.0/en/rename-user.html) fails with an error if any account to be renamed is named as the DEFINER attribute for any stored object. (That is, the statement fails if renaming an account would cause a stored object to become orphaned.)
* [CREATE USER](https://dev.mysql.com/doc/refman/8.0/en/create-user.html) fails with an error if any account to be created is named as the DEFINER attribute for any stored object. (That is, the statement fails if creating an account would cause the account to adopt a currently orphaned stored object.)

In certain situations, it may be necessary to deliberately execute those account-management statements even when they would otherwise fail. To make this possible, if a user has the [SET\_USER\_ID](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_set-user-id) privilege, that privilege overrides the orphan object security checks and the statements succeed with a warning rather than failing with an error.

To obtain information about the accounts used as stored object definers in a MySQL installation, query the INFORMATION\_SCHEMA.

This query identifies which INFORMATION\_SCHEMA tables describe objects that have a DEFINER attribute:

mysql> SELECT TABLE\_SCHEMA, TABLE\_NAME FROM INFORMATION\_SCHEMA.COLUMNS

WHERE COLUMN\_NAME = 'DEFINER';

+--------------------+------------+

| TABLE\_SCHEMA | TABLE\_NAME |

+--------------------+------------+

| information\_schema | EVENTS |

| information\_schema | ROUTINES |

| information\_schema | TRIGGERS |

| information\_schema | VIEWS |

+--------------------+------------+

The result tells you which tables to query to discover which stored object DEFINER values exist and which objects have a particular DEFINER value:

* To identify which DEFINER values exist in each table, use these queries:
* SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.EVENTS;
* SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.ROUTINES;
* SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.TRIGGERS;

SELECT DISTINCT DEFINER FROM INFORMATION\_SCHEMA.VIEWS;

The query results are significant for any account displayed as follows:

* + If the account exists, dropping or renaming it causes stored objects to become orphaned. If you plan to drop or rename the account, consider first dropping its associated stored objects or redefining them to have a different definer.
  + If the account does not exist, creating it causes it to adopt currently orphaned stored objects. If you plan to create the account, consider whether the orphaned objects should be associated with it. If not, redefine them to have a different definer.

To redefine an object with a different definer, you can use [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html) or [ALTER VIEW](https://dev.mysql.com/doc/refman/8.0/en/alter-view.html) to directly modify the DEFINER account of events and views. For stored procedures and functions and for triggers, you must drop the object and re-create it to assign a different DEFINER account

* To identify which objects have a given DEFINER account, use these queries, substituting the account of interest for ***user\_name***@***host\_name***:
* SELECT EVENT\_SCHEMA, EVENT\_NAME FROM INFORMATION\_SCHEMA.EVENTS
* WHERE DEFINER = 'user\_name@host\_name';
* SELECT ROUTINE\_SCHEMA, ROUTINE\_NAME, ROUTINE\_TYPE
* FROM INFORMATION\_SCHEMA.ROUTINES
* WHERE DEFINER = 'user\_name@host\_name';
* SELECT TRIGGER\_SCHEMA, TRIGGER\_NAME FROM INFORMATION\_SCHEMA.TRIGGERS
* WHERE DEFINER = 'user\_name@host\_name';
* SELECT TABLE\_SCHEMA, TABLE\_NAME FROM INFORMATION\_SCHEMA.VIEWS

WHERE DEFINER = 'user\_name@host\_name';

For the [ROUTINES](https://dev.mysql.com/doc/refman/8.0/en/information-schema-routines-table.html) table, the query includes the ROUTINE\_TYPE column so that output rows distinguish whether the DEFINER is for a stored procedure or stored function.

If the account you are searching for does not exist, any objects displayed by those queries are orphan objects.

### Risk-Minimization Guidelines

To minimize the risk potential for stored object creation and use, follow these guidelines:

* Do not create orphan stored objects; that is, objects for which the DEFINER attribute names a nonexistent account. Do not cause stored objects to become orphaned by dropping or renaming an account named by the DEFINER attribute of any existing object.
* For a stored routine or view, use SQL SECURITY INVOKER in the object definition when possible so that it can be used only by users with permissions appropriate for the operations performed by the object.
* If you create definer-context stored objects while using an account that has the [SET\_USER\_ID](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_set-user-id) privilege (or the deprecated [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege), specify an explicit DEFINER attribute that names an account possessing only the privileges required for the operations performed by the object. Specify a highly privileged DEFINER account only when absolutely necessary.
* Administrators can prevent users from creating stored objects that specify highly privileged DEFINER accounts by not granting them the [SET\_USER\_ID](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_set-user-id) privilege (or the deprecated [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege).
* Definer-context objects should be written keeping in mind that they may be able to access data for which the invoking user has no privileges. In some cases, you can prevent references to these objects by not granting unauthorized users particular privileges:
  + A stored routine cannot be referenced by a user who does not have the [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_execute) privilege for it.
  + A view cannot be referenced by a user who does not have the appropriate privilege for it ([SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) to select from it, [INSERT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_insert) to insert into it, and so forth).

However, no such control exists for triggers and events because they always execute in definer context. The server invokes these objects automatically as necessary, and users do not reference them directly:

* + A trigger is activated by access to the table with which it is associated, even ordinary table accesses by users with no special privileges.
  + An event is executed by the server on a scheduled basis.

In both cases, if the DEFINER account is highly privileged, the object may be able to perform sensitive or dangerous operations. This remains true if the privileges needed to create the object are revoked from the account of the user who created it. Administrators should be especially careful about granting users object-creation privileges.

## Stored Program Binary Logging

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. (Binary log events differ from scheduled event stored objects.) The binary log has two important purposes:

* For replication, the binary log is used on source replication servers as a record of the statements to be sent to replica servers. The source sends the events contained in its binary log to its replicas, which execute those events to make the same data changes that were made on the source. See [Section 17.2, “Replication Implementation”](https://dev.mysql.com/doc/refman/8.0/en/replication-implementation.html).
* 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 7.3.2, “Using Backups for Recovery”](https://dev.mysql.com/doc/refman/8.0/en/recovery-from-backups.html).

However, if logging occurs at the statement level, there are certain binary logging issues with respect to stored programs (stored procedures and functions, triggers, and events):

* In some cases, a statement might affect different sets of rows on source and replica.
* Replicated statements executed on a replica are processed by the replica SQL thread, which has full privileges. It is possible for a procedure to follow different execution paths on source and replica servers, so a user can write a routine containing a dangerous statement that executes only on the replica where it is processed by a thread that has full privileges.
* If a stored program that modifies data is nondeterministic, it is not repeatable. This can result in different data on source and replica, or cause restored data to differ from the original data.

This section describes how MySQL handles binary logging for stored programs. It states the current conditions that the implementation places on the use of stored programs, and what you can do to avoid logging problems. It also provides additional information about the reasons for these conditions.

In general, the issues described here result when binary logging occurs at the SQL statement level (statement-based binary logging). If you use row-based binary logging, the log contains changes made to individual rows as a result of executing SQL statements. When routines or triggers execute, row changes are logged, not the statements that make the changes. For stored procedures, this means that the [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statement is not logged. For stored functions, row changes made within the function are logged, not the function invocation. For triggers, row changes made by the trigger are logged. On the replica side, only the row changes are seen, not the stored program invocation.

Mixed format binary logging ([binlog\_format=MIXED](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_binlog_format)) uses statement-based binary logging, except for cases where only row-based binary logging is guaranteed to lead to proper results. With mixed format, when a stored function, stored procedure, trigger, event, or prepared statement contains anything that is not safe for statement-based binary logging, the entire statement is marked as unsafe and logged in row format. The statements used to create and drop procedures, functions, triggers, and events are always safe, and are logged in statement format..

Unless noted otherwise, the remarks here assume that binary logging is enabled on the server If the binary log is not enabled, replication is not possible, nor is the binary log available for data recovery.

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

* To create or alter a stored function, you must have the [SET\_USER\_ID](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_set-user-id) privilege (or the deprecated [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege), in addition to the [CREATE ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-routine) or [ALTER ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_alter-routine) privilege that is normally required. (Depending on the DEFINER value in the function definition, [SET\_USER\_ID](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_set-user-id) or [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) might be required regardless of whether binary logging is enabled.
* 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](https://dev.mysql.com/doc/refman/8.0/en/create-function.html) 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()](https://dev.mysql.com/doc/refman/8.0/en/miscellaneous-functions.html#function_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 nondeterministic results.

* When you attempt to execute a stored function, if [binlog\_format=STATEMENT](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_binlog_format) is set, the DETERMINISTIC keyword must be specified in the function definition. If this is not the case, an error is generated and the function does not run, unless [log\_bin\_trust\_function\_creators=1](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_log_bin_trust_function_creators) is specified to override this check (see below). For recursive function calls, the DETERMINISTIC keyword is required on the outermost call only. If row-based or mixed binary logging is in use, the statement is accepted and replicated even if the function was defined without the DETERMINISTIC keyword.
* Because MySQL does not check if a function really is deterministic at creation time, the invocation of a stored function with the DETERMINISTIC keyword might carry out an action that is unsafe for statement-based logging, or invoke a function or procedure containing unsafe statements. If this occurs when [binlog\_format=STATEMENT](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_binlog_format) is set, a warning message is issued. If row-based or mixed binary logging is in use, no warning is issued, and the statement is replicated in row-based format.
* To relax the preceding conditions on function creation (that you must have the [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege and that a function must be declared deterministic or to not modify data), set the global [log\_bin\_trust\_function\_creators](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_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 at server startup.

If binary logging is not enabled, [log\_bin\_trust\_function\_creators](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_log_bin_trust_function_creators) does not apply. [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) is not required for function creation unless, as described previously, the DEFINER value in the function definition requires it.

* 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 17.5.1, “Replication Features and Issues”](https://dev.mysql.com/doc/refman/8.0/en/replication-features.html).

Triggers are similar to stored functions, so the preceding remarks regarding functions also apply to triggers with the following exception: [CREATE TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html) does not have an optional DETERMINISTIC characteristic, so triggers are assumed to be always deterministic. However, this assumption might be invalid in some cases. For example, the [UUID()](https://dev.mysql.com/doc/refman/8.0/en/miscellaneous-functions.html#function_uuid) function is nondeterministic (and does not replicate). 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](https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html) if you do not have the required privileges. On the replica side, the replica uses the trigger DEFINER attribute to determine which user is considered to be the creator of the trigger.

The rest of this section provides additional detail about the logging implementation and its implications. 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. This discussion applies only for statement-based logging, and not for row-based logging, with the exception of the first item: CREATE and DROP statements are logged as statements regardless of the logging mode.

* The server writes [CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/create-event.html), [CREATE PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html), [CREATE FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/create-function.html), [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html), [ALTER PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/alter-procedure.html), [ALTER FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/alter-function.html), [DROP EVENT](https://dev.mysql.com/doc/refman/8.0/en/drop-event.html), [DROP PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/drop-procedure.html), and [DROP FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/drop-function.html) statements to the binary log.
* A stored function invocation is logged as a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement if the function changes data and occurs within a statement that would not otherwise be logged. This prevents nonreplication of data changes that result from use of stored functions in nonlogged statements. For example, [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statements are not written to the binary log, but a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) might invoke a stored function that makes changes. To handle this, a SELECT ***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 source server:
* 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](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement executes, the function f1() is invoked three times. Two of those invocations insert a row, and MySQL logs a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement for each of them. That is, MySQL writes the following statements to the binary log:

SELECT f1(1);

SELECT f1(2);

The server also logs a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement for a stored function invocation when the function invokes a stored procedure that causes an error. In this case, the server writes the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement to the log along with the expected error code. On the replica, if the same error occurs, that is the expected result and replication continues. Otherwise, replication stops.

* Logging stored function invocations rather than the statements executed by a function has a security implication for replication, which arises from two factors:
  + It is possible for a function to follow different execution paths on source and replica servers.
  + Statements executed on a replica are processed by the replica SQL thread which has full privileges.

The implication is that although a user must have the [CREATE ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-routine) privilege to create a function, the user can write a function containing a dangerous statement that executes only on the replica where it is processed by a thread that has full privileges. For example, if the source and replica servers have server ID values of 1 and 2, respectively, a user on the source server could create and invoke an unsafe function unsafe\_func() as follows:

mysql> delimiter //

mysql> CREATE FUNCTION unsafe\_func () RETURNS INT

-> BEGIN

-> IF @@server\_id=2 THEN dangerous\_statement; END IF;

-> RETURN 1;

-> END;

-> //

mysql> delimiter ;

mysql> INSERT INTO t VALUES(unsafe\_func());

The [CREATE FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/create-function.html) and [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statements are written to the binary log, so the replica executes them. Because the replica SQL thread has full privileges, it executes the dangerous statement. Thus, the function invocation has different effects on the source and replica and is not replication-safe.

To guard against this danger for servers that have binary logging enabled, stored function creators must have the [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege, in addition to the usual [CREATE ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-routine) privilege that is required. Similarly, to use [ALTER FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/alter-function.html), you must have the [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege in addition to the [ALTER ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_alter-routine) privilege. Without the [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege, an error occurs:

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\_function\_creators variable)

If you do not want to require function creators to have the [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege (for example, if all users with the [CREATE ROUTINE](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-routine) privilege on your system are experienced application developers), set the global [log\_bin\_trust\_function\_creators](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_log_bin_trust_function_creators) system variable to 1. You can also set this variable at server startup. If binary logging is not enabled, [log\_bin\_trust\_function\_creators](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_log_bin_trust_function_creators) does not apply. [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) is not required for function creation unless, as described previously, the DEFINER value in the function definition requires it.

* If a function that performs updates is nondeterministic, it is not repeatable. This can have two undesirable effects:
  + It causes a replica to differ from the source.
  + Restored data does not match the original data.

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

* + The DETERMINISTIC and NOT DETERMINISTIC characteristics indicate whether a function always produces the same result for given inputs. The default is NOT DETERMINISTIC if neither characteristic is given. To declare that a function 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 function reads or writes data. Either NO SQL or READS SQL DATA indicates that a function 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 FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/create-function.html) 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)

If you set [log\_bin\_trust\_function\_creators](https://dev.mysql.com/doc/refman/8.0/en/replication-options-binary-log.html#sysvar_log_bin_trust_function_creators) to 1, the requirement that functions be deterministic or not modify data is dropped.

* Stored procedure calls are logged at the statement level rather than at the [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) level. That is, the server does not log the [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statement, it logs those statements within the procedure that actually execute. As a result, the same changes that occur on the source server also occur on replicas. This prevents problems that could result from a procedure having different execution paths on different machines.

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 nonprocedure 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()](https://dev.mysql.com/doc/refman/8.0/en/miscellaneous-functions.html#function_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()](https://dev.mysql.com/doc/refman/8.0/en/miscellaneous-functions.html#function_name-const) enables a logged standalone statement to be executed on a replica with the same effect as the original statement that was executed on the source within a stored procedure.

The use of [NAME\_CONST()](https://dev.mysql.com/doc/refman/8.0/en/miscellaneous-functions.html#function_name-const) can result in a problem for [CREATE TABLE ... SELECT](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) statements when the source column expressions refer to local variables. Converting these references to [NAME\_CONST()](https://dev.mysql.com/doc/refman/8.0/en/miscellaneous-functions.html#function_name-const) expressions can result in column names that are different on the source and replica servers, or names that are too long to be legal column identifiers. A workaround is to supply aliases for columns that refer to local variables. Consider this statement when myvar has a value of 1:

CREATE TABLE t1 SELECT myvar;

This is rewritten as follows:

CREATE TABLE t1 SELECT NAME\_CONST(myvar, 1);

To ensure that the source and replica tables have the same column names, write the statement like this:

CREATE TABLE t1 SELECT myvar AS myvar;

The rewritten statement becomes:

CREATE TABLE t1 SELECT NAME\_CONST(myvar, 1) AS myvar;

* + A statement to be logged might contain references to user-defined variables. To handle this, MySQL writes a [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html) statement to the binary log to make sure that the variable exists on the replica with the same value as on the source. For example, if a statement refers to a variable @my\_var, that statement is preceded in the binary log by the following statement, where ***value*** is the value of @my\_var on the source:

SET @my\_var = value;

* + Procedure calls can occur within a committed or rolled-back transaction. 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](https://dev.mysql.com/doc/refman/8.0/en/commit.html), [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html), and [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) 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](https://dev.mysql.com/doc/refman/8.0/en/commit.html) and [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html) 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 nontransactional tables are logged because rollback does not cancel them.
    - Updates to a mix of transactional and nontransactional tables are logged surrounded by [BEGIN](https://dev.mysql.com/doc/refman/8.0/en/commit.html) and [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) so that replicas make the same changes and rollbacks as on the source.
* 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](https://dev.mysql.com/doc/refman/8.0/en/do.html) statement (if it occurs within a statement that is not logged). For this reason, care should be exercised in the use of stored functions that invoke a procedure, even if the procedure is otherwise safe in itself.

## Restrictions on Stored Programs

Some of the restrictions noted here apply to all stored routines; that is, both to stored procedures and stored functions. There are also some [restrictions specific to stored functions](https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html#stored-routines-function-restrictions) but not to stored procedures.

The restrictions for stored functions also apply to triggers. There are also some [restrictions specific to triggers](https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html#stored-routines-trigger-restrictions).

The restrictions for stored procedures also apply to the [DO](https://dev.mysql.com/doc/refman/8.0/en/do.html) clause of Event Scheduler event definitions. There are also some [restrictions specific to events](https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html#stored-routines-event-restrictions).

### SQL Statements Not Permitted in Stored Routines

Stored routines cannot contain arbitrary SQL statements. The following statements are not permitted:

* The locking statements [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) and [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html).
* [ALTER VIEW](https://dev.mysql.com/doc/refman/8.0/en/alter-view.html).
* [LOAD DATA](https://dev.mysql.com/doc/refman/8.0/en/load-data.html).
* SQL prepared statements ([PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html)) can be used in stored procedures, but not stored functions or triggers. Thus, stored functions and triggers cannot use dynamic SQL (where you construct statements as strings and then execute them).
* Generally, statements not permitted in SQL prepared statements are also not permitted in stored programs. For a list of statements supported as prepared statements, see [Section 13.5, “Prepared Statements”](https://dev.mysql.com/doc/refman/8.0/en/sql-prepared-statements.html). Exceptions are [SIGNAL](https://dev.mysql.com/doc/refman/8.0/en/signal.html), [RESIGNAL](https://dev.mysql.com/doc/refman/8.0/en/resignal.html), and [GET DIAGNOSTICS](https://dev.mysql.com/doc/refman/8.0/en/get-diagnostics.html), which are not permissible as prepared statements but are permitted in stored programs.
* Because local variables are in scope only during stored program execution, references to them are not permitted in prepared statements created within a stored program. Prepared statement scope is the current session, not the stored program, so the statement could be executed after the program ends, at which point the variables would no longer be in scope. For example, SELECT ... INTO ***local\_var*** cannot be used as a prepared statement. This restriction also applies to stored procedure and function parameters..
* Within all stored programs (stored procedures and functions, triggers, and events), the parser treats [BEGIN [WORK]](https://dev.mysql.com/doc/refman/8.0/en/commit.html) as the beginning of a [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) block. To begin a transaction in this context, use [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html) instead.

### Restrictions for Stored Functions

The following additional statements or operations are not permitted within stored functions. They are permitted within stored procedures, except stored procedures that are invoked from within a stored function or trigger. For example, if you use [FLUSH](https://dev.mysql.com/doc/refman/8.0/en/flush.html) in a stored procedure, that stored procedure cannot be called from a stored function or trigger.

* Statements that perform 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 permit them.
* Statements that return a result set. This includes [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statements that do not have an INTO ***var\_list*** clause and other statements such as [SHOW](https://dev.mysql.com/doc/refman/8.0/en/show.html), [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html), and [CHECK TABLE](https://dev.mysql.com/doc/refman/8.0/en/check-table.html). A function can process a result set either with [SELECT ... INTO ***var\_list***](https://dev.mysql.com/doc/refman/8.0/en/select-into.html) or by using a cursor and [FETCH](https://dev.mysql.com/doc/refman/8.0/en/fetch.html) statements..
* [FLUSH](https://dev.mysql.com/doc/refman/8.0/en/flush.html) statements.
* Stored functions cannot be used recursively.
* A stored function or trigger cannot modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger.
* If you refer to a temporary table multiple times in a stored function under different aliases, a Can't reopen table: '***tbl\_name***' error occurs, even if the references occur in different statements within the function.
* [HANDLER ... READ](https://dev.mysql.com/doc/refman/8.0/en/handler.html) statements that invoke stored functions can cause replication errors and are disallowed.

### Restrictions for Triggers

For triggers, the following additional restrictions apply:

* Triggers are not activated by foreign key actions.
* When using row-based replication, triggers on the replica are not activated by statements originating on the source. The triggers on the replica are activated when using statement-based replication.
* The [RETURN](https://dev.mysql.com/doc/refman/8.0/en/return.html) statement is not permitted in triggers, which cannot return a value. To exit a trigger immediately, use the [LEAVE](https://dev.mysql.com/doc/refman/8.0/en/leave.html) statement.
* Triggers are not permitted on tables in the mysql database. Nor are they permitted on INFORMATION\_SCHEMA or performance\_schema tables. Those tables are actually views and triggers are not permitted on views.
* The trigger cache does not detect when metadata of the underlying objects has changed. If a trigger uses a table and the table has changed since the trigger was loaded into the cache, the trigger operates using the outdated metadata.

### Name Conflicts within Stored Routines

The same identifier might be used for a routine parameter, a local variable, and a table column. Also, the same local variable name can be used in nested blocks. For example:

CREATE PROCEDURE p (i INT)

BEGIN

DECLARE i INT DEFAULT 0;

SELECT i FROM t;

BEGIN

DECLARE i INT DEFAULT 1;

SELECT i FROM t;

END;

END;

In such cases, the identifier is ambiguous and the following precedence rules apply:

* A local variable takes precedence over a routine parameter or table column.
* A routine parameter takes precedence over a table column.
* A local variable in an inner block takes precedence over a local variable in an outer block.

The behavior that variables take precedence over table columns is nonstandard.

### Replication Considerations

Use of stored routines can cause replication problems..

The [--replicate-wild-do-table=***db\_name.tbl\_name***](https://dev.mysql.com/doc/refman/8.0/en/replication-options-replica.html#option_mysqld_replicate-wild-do-table) option applies to tables, views, and triggers. It does not apply to stored procedures and functions, or events. To filter statements operating on the latter objects, use one or more of the --replicate-\*-db options.

### Debugging Considerations

There are no stored routine debugging facilities.

### Unsupported Syntax from the SQL:2003 Standard

The MySQL stored routine syntax is based on the SQL:2003 standard. The following items from that standard are not currently supported:

* UNDO handlers
* FOR loops

### Stored Routine Concurrency Considerations

To prevent problems of interaction between sessions, when a client issues a statement, the server uses a snapshot of routines and triggers available for execution of the statement. That is, the server calculates a list of procedures, functions, and triggers that may be used during execution of the statement, loads them, and then proceeds to execute the statement. While the statement executes, it does not see changes to routines performed by other sessions.

For maximum concurrency, stored functions should minimize their side-effects; in particular, updating a table within a stored function can reduce concurrent operations on that table. A stored function acquires table locks before executing, to avoid inconsistency in the binary log due to mismatch of the order in which statements execute and when they appear in the log. When statement-based binary logging is used, statements that invoke a function are recorded rather than the statements executed within the function. Consequently, stored functions that update the same underlying tables do not execute in parallel. In contrast, stored procedures do not acquire table-level locks. All statements executed within stored procedures are written to the binary log, even for statement-based binary logging.

### Event Scheduler Restrictions

The following limitations are specific to the Event Scheduler:

* Event names are handled in case-insensitive fashion. For example, you cannot have two events in the same database with the names anEvent and AnEvent.
* An event may not be created, altered, or dropped from within a stored program, if the event name is specified by means of a variable. An event also may not create, alter, or drop stored routines or triggers.
* DDL statements on events are prohibited while a [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement is in effect.
* Event timings using the intervals YEAR, QUARTER, MONTH, and YEAR\_MONTH are resolved in months; those using any other interval are resolved in seconds. There is no way to cause events scheduled to occur at the same second to execute in a given order. In addition—due to rounding, the nature of threaded applications, and the fact that a nonzero length of time is required to create events and to signal their execution—events may be delayed by as much as 1 or 2 seconds. However, the time shown in the [INFORMATION\_SCHEMA.EVENTS](https://dev.mysql.com/doc/refman/8.0/en/information-schema-events-table.html) table's LAST\_EXECUTED column is always accurate to within one second of the actual event execution time. (See also Bug #16522.)
* Each execution of the statements contained in the body of an event takes place in a new connection; thus, these statements have no effect in a given user session on the server's statement counts such as Com\_select and Com\_insert that are displayed by [SHOW STATUS](https://dev.mysql.com/doc/refman/8.0/en/show-status.html). However, such counts are updated in the global scope. (Bug #16422)
* Events do not support times later than the end of the Unix Epoch; this is approximately the beginning of the year 2038. Such dates are specifically not permitted by the Event Scheduler. (Bug #16396)
* References to stored functions, user-defined functions, and tables in the ON SCHEDULE clauses of [CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/create-event.html) and [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html) statements are not supported. These sorts of references are not permitted. (See Bug #22830 for more information.)

### Stored routines and triggers in NDB Cluster

While stored procedures, stored functions, triggers, and scheduled events are all supported by tables using the [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html) storage engine, you must keep in mind that these do not propagate automatically between MySQL Servers acting as Cluster SQL nodes. This is because stored routine and trigger definitions are stored in tables in the mysql system database using InnoDB tables, which are not copied between Cluster nodes.

Any stored routine or trigger that interacts with MySQL Cluster tables must be re-created by running the appropriate [CREATE PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html), [CREATE FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/create-function.html), or [CREATE TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html) statements on each MySQL Server that participates in the cluster where you wish to use the stored routine or trigger. Similarly, any changes to existing stored routines or triggers must be carried out explicitly on all Cluster SQL nodes, using the appropriate ALTER or DROP statements on each MySQL Server accessing the cluster.

**Warning**

Do not attempt to work around the issue just described by converting any mysql database tables to use the [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html) storage engine. Altering the system tables in the *mysql* database is not supported and is very likely to produce undesirable results.

## Restrictions on Views

The maximum number of tables that can be referenced in the definition of a view is 61.

View processing is not optimized:

* It is not possible to create an index on a view.
* Indexes can be used for views processed using the merge algorithm. However, a view that is processed with the temptable algorithm is unable to take advantage of indexes on its underlying tables (although indexes can be used during generation of the temporary tables).

There is a general principle that you cannot modify a table and select from the same table in a subquery.

The same principle also applies if you select from a view that selects from the table, if the view selects from the table in a subquery and the view is evaluated using the merge algorithm. Example:

CREATE VIEW v1 AS

SELECT \* FROM t2 WHERE EXISTS (SELECT 1 FROM t1 WHERE t1.a = t2.a);

UPDATE t1, v2 SET t1.a = 1 WHERE t1.b = v2.b;

If the view is evaluated using a temporary table, you can select from the table in the view subquery and still modify that table in the outer query. In this case, the view is stored in a temporary table and thus you are not really selecting from the table in a subquery and modifying it at the same time. (This is another reason you might wish to force MySQL to use the temptable algorithm by specifying ALGORITHM = TEMPTABLE in the view definition.)

You can use [DROP TABLE](https://dev.mysql.com/doc/refman/8.0/en/drop-table.html) or [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html) to drop or alter a table that is used in a view definition. No warning results from the DROP or ALTER operation, even though this invalidates the view. Instead, an error occurs later, when the view is used. [CHECK TABLE](https://dev.mysql.com/doc/refman/8.0/en/check-table.html) can be used to check for views that have been invalidated by DROP or ALTER operations.

With regard to view updatability, the overall goal for views is that if any view is theoretically updatable, it should be updatable in practice. MySQL as quickly as possible. Many theoretically updatable views can be updated now, but limitations still exist..

There exists a shortcoming with the current implementation of views. If a user is granted the basic privileges necessary to create a view (the [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-view) and [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) privileges), that user cannot call [SHOW CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/show-create-view.html) on that object unless the user is also granted the [SHOW VIEW](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_show-view) privilege.

That shortcoming can lead to problems backing up a database with [**mysqldump**](https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html), which may fail due to insufficient privileges. This problem is described in Bug #22062.

The workaround to the problem is for the administrator to manually grant the [SHOW VIEW](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_show-view) privilege to users who are granted [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_create-view), since MySQL doesn't grant it implicitly when views are created.

Views do not have indexes, so index hints do not apply. Use of index hints when selecting from a view is not permitted.

[SHOW CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/show-create-view.html) displays view definitions using an AS ***alias\_name*** clause for each column. If a column is created from an expression, the default alias is the expression text, which can be quite long. Aliases for column names in [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) statements are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters). As a result, views created from the output of [SHOW CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/show-create-view.html) fail if any column alias exceeds 64 characters. This can cause problems in the following circumstances for views with too-long aliases:

* View definitions fail to replicate to newer replicas that enforce the column-length restriction.
* Dump files created with [**mysqldump**](https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html) cannot be loaded into servers that enforce the column-length restriction.

A workaround for either problem is to modify each problematic view definition to use aliases that provide shorter column names. Then the view replicates properly, and can be dumped and reloaded without causing an error. To modify the definition, drop and create the view again with [DROP VIEW](https://dev.mysql.com/doc/refman/8.0/en/drop-view.html) and [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html), or replace the definition with [CREATE OR REPLACE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html).

For problems that occur when reloading view definitions in dump files, another workaround is to edit the dump file to modify its [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html) statements. However, this does not change the original view definitions, which may cause problems for subsequent dump operations.