# Prepared Statements

Content

[Prepared Statements 1](#_Toc57669500)

[Prepared Statements in Application Programs 1](#_Toc57669501)

[Prepared Statements in SQL Scripts 1](#_Toc57669502)

[PREPARE, EXECUTE, and DEALLOCATE PREPARE Statements 2](#_Toc57669503)

[13.5.1 PREPARE Statement 7](#_Toc57669504)

[EXECUTE Statement 11](#_Toc57669505)

[DEALLOCATE PREPARE Statement 11](#_Toc57669506)

MySQL 8.0 provides support for server-side prepared statements. This support takes advantage of the efficient client/server binary protocol. Using prepared statements with placeholders for parameter values has the following benefits:

Less overhead for parsing the statement each time it is executed. Typically, database applications process large volumes of almost-identical statements, with only changes to literal or variable values in clauses such as WHERE for queries and deletes, SET for updates, and VALUES for inserts.

Protection against SQL injection attacks. The parameter values can contain unescaped SQL quote and delimiter characters.

The following sections provide an overview of the characteristics of prepared statements:

## Prepared Statements in Application Programs

You can use server-side prepared statements through client programming interfaces, including the [MySQL C API client library](https://dev.mysql.com/doc/c-api/8.0/en/) for C programs, [MySQL Connector/J](https://dev.mysql.com/doc/connector-j/8.0/en/) for Java programs, and [MySQL Connector/NET](https://dev.mysql.com/doc/connector-net/en/) for programs using .NET technologies. For example, the C API provides a set of function calls that make up its prepared statement API. See [C API Prepared Statements](https://dev.mysql.com/doc/c-api/8.0/en/c-api-prepared-statements.html). Other language interfaces can provide support for prepared statements that use the binary protocol by linking in the C client library, one example being the [mysqli extension](http://php.net/mysqli" \t "_top), available in PHP 5.0 and higher.

## Prepared Statements in SQL Scripts

An alternative SQL interface to prepared statements is available. This interface is not as efficient as using the binary protocol through a prepared statement API, but requires no programming because it is available directly at the SQL level:

You can use it when no programming interface is available to you.

You can use it from any program that can send SQL statements to the server to be executed, such as the **[mysql](https://dev.mysql.com/doc/refman/8.0/en/mysql.html" \o "4.5.1 mysql — The MySQL Command-Line Client)** client program.

You can use it even if the client is using an old version of the client library.

SQL syntax for prepared statements is intended to be used for situations such as these:

To test how prepared statements work in your application before coding it.

To use prepared statements when you do not have access to a programming API that supports them.

To interactively troubleshoot application issues with prepared statements.

To create a test case that reproduces a problem with prepared statements, so that you can file a bug report.

## PREPARE, EXECUTE, and DEALLOCATE PREPARE Statements

SQL syntax for prepared statements is based on three SQL statements:

[EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) executes a prepared.

[DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) releases a prepared statement.

The following examples show two equivalent ways of preparing a statement that computes the hypotenuse of a triangle given the lengths of the two sides.

The first example shows how to create a prepared statement by using a string literal to supply the text of the statement:

mysql> PREPARE stmt1 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';

mysql> SET @a = 3;

mysql> SET @b = 4;

mysql> EXECUTE stmt1 USING @a, @b;

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

| hypotenuse |

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

| 5 |

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

mysql> DEALLOCATE PREPARE stmt1;

The second example is similar, but supplies the text of the statement as a user variable:

mysql> SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';

mysql> PREPARE stmt2 FROM @s;

mysql> SET @a = 6;

mysql> SET @b = 8;

mysql> EXECUTE stmt2 USING @a, @b;

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

| hypotenuse |

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

| 10 |

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

mysql> DEALLOCATE PREPARE stmt2;

Here is an additional example that demonstrates how to choose the table on which to perform a query at runtime, by storing the name of the table as a user variable:

mysql> USE test;

mysql> CREATE TABLE t1 (a INT NOT NULL);

mysql> INSERT INTO t1 VALUES (4), (8), (11), (32), (80);

mysql> SET @table = 't1';

mysql> SET @s = CONCAT('SELECT \* FROM ', @table);

mysql> PREPARE stmt3 FROM @s;

mysql> EXECUTE stmt3;

+----+

| a |

+----+

| 4 |

| 8 |

| 11 |

| 32 |

| 80 |

+----+

mysql> DEALLOCATE PREPARE stmt3;

A prepared statement is specific to the session in which it was created. If you terminate a session without deallocating a previously prepared statement, the server deallocates it automatically.

A prepared statement is also global to the session. If you create a prepared statement within a stored routine, it is not deallocated when the stored routine ends.

To guard against too many prepared statements being created simultaneously, set the [max\_prepared\_stmt\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html" \l "sysvar_max_prepared_stmt_count) system variable. To prevent the use of prepared statements, set the value to 0.

SQL Syntax Permitted in Prepared Statements

The following SQL statements can be used as prepared statements:

ALTER TABLE

ALTER USER

ANALYZE TABLE

CACHE INDEX

CALL

CHANGE MASTER

CHECKSUM {TABLE | TABLES}

COMMIT

{CREATE | DROP} INDEX

{CREATE | RENAME | DROP} DATABASE

{CREATE | DROP} TABLE

{CREATE | RENAME | DROP} USER

{CREATE | DROP} VIEW

DELETE

DO

FLUSH {TABLE | TABLES | TABLES WITH READ LOCK | HOSTS | PRIVILEGES

| LOGS | STATUS | MASTER | SLAVE | USER\_RESOURCES}

GRANT

INSERT

INSTALL PLUGIN

KILL

LOAD INDEX INTO CACHE

OPTIMIZE TABLE

RENAME TABLE

REPAIR TABLE

REPLACE

RESET {MASTER | SLAVE}

REVOKE

SELECT

SET

SHOW {WARNINGS | ERRORS}

SHOW BINLOG EVENTS

SHOW CREATE {PROCEDURE | FUNCTION | EVENT | TABLE | VIEW}

SHOW {MASTER | BINARY} LOGS

SHOW {MASTER | SLAVE} STATUS

SLAVE {START | STOP}

TRUNCATE TABLE

UNINSTALL PLUGIN

UPDATE

Other statements are not supported.

For compliance with the SQL standard, which states that diagnostics statements are not preparable, MySQL does not support the following as prepared statements:

SHOW WARNINGS, SHOW COUNT(\*) WARNINGS

SHOW ERRORS, SHOW COUNT(\*) ERRORS

Statements containing any reference to the [warning\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_warning_count) or [error\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_error_count) system variable.

Generally, statements not permitted in SQL prepared statements are also not permitted in stored programs.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed.

Placeholders can be used for the arguments of the LIMIT clause when using prepared statements.

In prepared [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statements used with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) and [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), placeholder support for OUT and INOUT parameters is available beginning with MySQL 8.0..

SQL syntax for prepared statements cannot be used in nested fashion. That is, a statement passed to [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) cannot itself be a [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), or [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) statement.

SQL syntax for prepared statements is distinct from using prepared statement API calls. For example, you cannot use the [mysql\_stmt\_prepare()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-stmt-prepare.html" \t "_top) C API function to prepare a [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), or [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) statement.

SQL syntax for prepared statements can be used within stored procedures, but not in stored functions or triggers. However, a cursor cannot be used for a dynamic statement that is prepared and executed with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) and [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html). The statement for a cursor is checked at cursor creation time, so the statement cannot be dynamic.

SQL syntax for prepared statements does not support multi-statements (that is, multiple statements within a single string separated by ; characters).

To write C programs that use the [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) SQL statement to execute stored procedures that contain prepared statements, the CLIENT\_MULTI\_RESULTS flag must be enabled. This is because each [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) returns a result to indicate the call status, in addition to any result sets that might be returned by statements executed within the procedure.

CLIENT\_MULTI\_RESULTS can be enabled when you call [mysql\_real\_connect()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-connect.html" \t "_top), either explicitly by passing the CLIENT\_MULTI\_RESULTS flag itself, or implicitly by passing CLIENT\_MULTI\_STATEMENTS (which also enables CLIENT\_MULTI\_RESULTS).

# 13.5.1 PREPARE Statement

PREPARE stmt\_name FROM preparable\_stmt

The [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) statement prepares a SQL statement and assigns it a name, ***stmt\_name***, by which to refer to the statement later. The prepared statement is executed with [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) and released with [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html). For examples, see [Section 13.5, “Prepared Statements”](https://dev.mysql.com/doc/refman/8.0/en/sql-prepared-statements.html).

Statement names are not case-sensitive. ***preparable\_stmt*** is either a string literal or a user variable that contains the text of the SQL statement. The text must represent a single statement, not multiple statements. Within the statement, ? characters can be used as parameter markers to indicate where data values are to be bound to the query later when you execute it. The ? characters should not be enclosed within quotation marks, even if you intend to bind them to string values. Parameter markers can be used only where data values should appear, not for SQL keywords, identifiers, and so forth.

If a prepared statement with the given name already exists, it is deallocated implicitly before the new statement is prepared. This means that if the new statement contains an error and cannot be prepared, an error is returned and no statement with the given name exists.

The scope of a prepared statement is the session within which it is created, which as several implications:

A prepared statement created in one session is not available to other sessions.

When a session ends, whether normally or abnormally, its prepared statements no longer exist. If auto-reconnect is enabled, the client is not notified that the connection was lost. For this reason, clients may wish to disable auto-reconnect..

A prepared statement created within a stored program continues to exist after the program finishes executing and can be executed outside the program later.

A statement prepared in stored program context cannot refer to stored procedure or function parameters or local variables because they go out of scope when the program ends and would be unavailable were the statement to be executed later outside the program. As a workaround, refer instead to user-defined variables, which also have session scope;

Beginning with MySQL 8.0.22, a parameter used in a prepared statement has its type determined when the statement is first prepared, and retains this type whenever [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) is invoked for this prepared statement (unless the statement is reprepared, as explained later in this section). Rules for determining a parameter's type are listed here:

A parameter which is an operand of a binary arithmetic operator has the same data type as the other operand.

If both operands of a binary arithmetic operator are parameters, the type of the parameters is decided by the context of the operator.

If a parameter is the operand of a unary arithmetic operator, the parameter's type is decided by the context of the operator.

If an arithmetic operator has no type-determining context, the derived type for any parameters involved is [DOUBLE PRECISION](https://dev.mysql.com/doc/refman/8.0/en/floating-point-types.html). This can happen, for example, when the parameter is a top-level node in a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) list, or when it is part of a comparison operator.

A parameter which is an operand of a character string operator has the same derived type as the aggregated type of the other operands. If all operands of the operator are parameters, the derived type is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html); its collation is determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html" \l "sysvar_collation_connection).

A parameter which is an operand of a temporal operator has type [DATETIME](https://dev.mysql.com/doc/refman/8.0/en/datetime.html) if the operator returns a DATETIME, [TIME](https://dev.mysql.com/doc/refman/8.0/en/time.html) if the operator returns a TIME, and [DATE](https://dev.mysql.com/doc/refman/8.0/en/datetime.html) if the operator returns a DATE.

A parameter which is an operand of a binary comparison operator has the same derived type as the other operand of the comparison.

A parameter that is an operand of a ternary comparison operator such as [BETWEEN](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#operator_between) has the same derived type as the aggregated type of the other operands.

If all operands of a comparison operator are parameters, the derived type for each of them is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), with collation determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html" \l "sysvar_collation_connection).

A parameter that is an output operand of any of [CASE](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#operator_case), [COALESCE](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_coalesce), [IF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_if), [IFNULL](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_ifnull), or [NULLIF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_nullif) has the same derived type as the aggregated type of the operator's other output operands.

If all output operands of any of [CASE](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#operator_case), [COALESCE](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_coalesce), [IF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_if), [IFNULL](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_ifnull), or [NULLIF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_nullif) are parameters, or they are all NULL, the type of the parameter is decided by the context of the operator.

If the parameter is an operand of any of of [CASE](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#operator_case), [COALESCE()](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_coalesce), [IF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_if), or [IFNULL](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_ifnull), and has no type-determining context, the derived type for each of the parameters involved is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), and its collation is determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html" \l "sysvar_collation_connection).

A parameter which is the operand of a [CAST()](https://dev.mysql.com/doc/refman/8.0/en/cast-functions.html#function_cast) has the same type as specified by the CAST().

If a parameter is an immediate member of a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) list that is not part of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, the derived type of the parameter is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), and its collation is determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html" \l "sysvar_collation_connection).

If a parameter is an immediate member of a SELECT list that is part of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, the derived type of the parameter is the type of the corresponding column into which the parameter is inserted.

If a parameter is used as source for an assignment in a SET clause of an [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) statement or in the ON DUPLICATE KEY UPDATE clause of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, the derived type of the parameter is the type of the corresponding column which is updated by the SET or ON DUPLICATE KEY UPDATE clause.

If a parameter is an argument of a function, the derived type depends on the function's return type.

For some combinations of actual type and derived type, an automatic repreparation of the statement is triggered, to ensure closer compatibility with previous versions of MySQL. Repreparation does not occur if any of the following conditions are true:

NULL is used as the actual parameter value.

A parameter is an operand of a [CAST()](https://dev.mysql.com/doc/refman/8.0/en/cast-functions.html#function_cast). (Instead, a cast to the derived type is attempted, and an exception raised if the cast fails.)

A parameter is a string. (In this case, an implicit CAST(? AS ***derived\_type***) is performed.)

The derived type and actual type of the parameter are both [INTEGER](https://dev.mysql.com/doc/refman/8.0/en/integer-types.html) and have the same sign.

The parameter's derived type is [DECIMAL](https://dev.mysql.com/doc/refman/8.0/en/fixed-point-types.html) and its actual type is either DECIMAL or [INTEGER](https://dev.mysql.com/doc/refman/8.0/en/integer-types.html).

The derived type is [DOUBLE](https://dev.mysql.com/doc/refman/8.0/en/floating-point-types.html) and the actual type is any numeric type.

Both the derived type and the actual type are string types.

If the derived type is temporal and the actual type is temporal. Exceptions: The derived type is [TIME](https://dev.mysql.com/doc/refman/8.0/en/time.html) and the actual type is not TIME; the derived type is [DATE](https://dev.mysql.com/doc/refman/8.0/en/datetime.html) and the actual type is not DATE.

The derived type is temporal and the actual type is numeric.

For cases other than those just listed, the statement is reprepared and the actual parameter types are used instead of the derived parameter types.

These rules also apply to a user variable referenced in a prepared statement.

Using a different data type for a given parameter or user variable within a prepared statement for executions of the statement subsequent to the first execution causes the statement to be reprepared. This is less efficient; it may also lead to the paremeter's (or variable's) actual type to vary, and thus for results to be inconsistent, with subsequent executions of the prepared statement. For these reasons, it is advisable to use the same data type for a given parameter when re-executing a prepared statement.

## EXECUTE Statement

EXECUTE stmt\_name [USING @var\_name [, @var\_name] ...]

After preparing a statement with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), you execute it with an [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) statement that refers to the prepared statement name. If the prepared statement contains any parameter markers, you must supply a USING clause that lists user variables containing the values to be bound to the parameters. Parameter values can be supplied only by user variables, and the USING clause must name exactly as many variables as the number of parameter markers in the statement.

You can execute a given prepared statement multiple times, passing different variables to it or setting the variables to different values before each execution.

## DEALLOCATE PREPARE Statement

{DEALLOCATE | DROP} PREPARE stmt\_name

To deallocate a prepared statement produced with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), use a [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) statement that refers to the prepared statement name. Attempting to execute a prepared statement after deallocating it results in an error. If too many prepared statements are created and not deallocated by either the DEALLOCATE PREPARE statement or the end of the session, you might encounter the upper limit enforced by the [max\_prepared\_stmt\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html" \l "sysvar_max_prepared_stmt_count) system variable.