

MySQL Restrictions and Limitations

Abstract

This is the MySQL Restrictions and Limitations extract from the MySQL 8.0 Reference Manual.

For legal information, see the [Legal Notices](#).

For help with using MySQL, please visit the [MySQL Forums](#), where you can discuss your issues with other MySQL users.

Document generated on: 2024-06-20 (revision: 78872)

Table of Contents

Preface and Legal Notices	v
1 Restrictions on Stored Programs	1
2 Restrictions on Views	7
3 Restrictions on Condition Handling	9
4 Restrictions on Server-Side Cursors	11
5 Restrictions on Subqueries	13
6 Restrictions on XA Transactions	15
7 Restrictions on Character Sets	17
8 Restrictions on Performance Schema	19
9 Restrictions on Pluggable Authentication	21
10 Restrictions and Limitations on Partitioning	23
10.1 Partitioning Keys, Primary Keys, and Unique Keys	29
10.2 Partitioning Limitations Relating to Storage Engines	32
10.3 Partitioning Limitations Relating to Functions	33
11 Windows Platform Restrictions	37
12 Limits in MySQL	39
12.1 Identifier Length Limits	39
12.2 Grant Table Scope Column Properties	40
12.3 Limits on Number of Databases and Tables	40
12.4 Limits on Table Size	40
12.5 Limits on Table Column Count and Row Size	41
13 MySQL Differences from Standard SQL	45
13.1 SELECT INTO TABLE Differences	45
13.2 UPDATE Differences	45
13.3 FOREIGN KEY Constraint Differences	45
13.4 '--' as the Start of a Comment	48
14 Known Issues in MySQL	49

Preface and Legal Notices

This is the MySQL Restrictions and Limitations extract from the MySQL 8.0 Reference Manual.

Licensing information—MySQL 8.0. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL 8.0, see the [MySQL 8.0 Commercial Release License Information User Manual](#) for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of MySQL 8.0, see the [MySQL 8.0 Community Release License Information User Manual](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Legal Notices

Copyright © 1997, 2024, Oracle and/or its affiliates.

License Restrictions

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

Warranty Disclaimer

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

Restricted Rights Notice

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

Hazardous Applications Notice

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other

measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Trademark Notice

Oracle, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

Third-Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Use of This Documentation

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Chapter 1 Restrictions on Stored Programs

- [SQL Statements Not Permitted in Stored Routines](#)
- [Restrictions for Stored Functions](#)
- [Restrictions for Triggers](#)
- [Name Conflicts within Stored Routines](#)
- [Replication Considerations](#)
- [Debugging Considerations](#)
- [Unsupported Syntax from the SQL:2003 Standard](#)
- [Stored Routine Concurrency Considerations](#)
- [Event Scheduler Restrictions](#)
- [Stored routines and triggers in NDB Cluster](#)

These restrictions apply to the features described in [Stored Objects](#).

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](#) but not to stored procedures.

The restrictions for stored functions also apply to triggers. There are also some [restrictions specific to triggers](#).

The restrictions for stored procedures also apply to the `DO` clause of Event Scheduler event definitions. There are also some [restrictions specific to events](#).

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` and `UNLOCK TABLES`.
- `ALTER VIEW`.
- `LOAD DATA` and `LOAD XML`.
- SQL prepared statements (`PREPARE`, `EXECUTE`, `DEALLOCATE PREPARE`) 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 [Prepared Statements](#). Exceptions are `SIGNAL`, `RESIGNAL`, and `GET DIAGNOSTICS`, 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. See [PREPARE Statement](#).

- Within all stored programs (stored procedures and functions, triggers, and events), the parser treats `BEGIN [WORK]` as the beginning of a `BEGIN ... END` block. To begin a transaction in this context, use `START TRANSACTION` 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` 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` statements that do not have an `INTO var_list` clause and other statements such as `SHOW`, `EXPLAIN`, and `CHECK TABLE`. A function can process a result set either with `SELECT ... INTO var_list` or by using a cursor and `FETCH` statements. See [SELECT ... INTO Statement](#), and [Cursors](#).
- `FLUSH` 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` 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. For more information, see [Replication and Triggers](#).
- The `RETURN` statement is not permitted in triggers, which cannot return a value. To exit a trigger immediately, use the `LEAVE` 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. This issue is discussed further in [Stored Program Binary Logging](#).

The `--replicate-wild-do-table=db_name.tbl_name` 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. See [Stored Program 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 from within a stored program. An event may not be 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` 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` 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`. 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, loadable functions, and tables in the `ON SCHEDULE` clauses of `CREATE EVENT` and `ALTER EVENT` 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` 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`, `CREATE FUNCTION`, or `CREATE TRIGGER` 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` storage engine. *Altering the system tables in the `mysql` database is not supported* and is very likely to produce undesirable results.

Chapter 2 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. See [Chapter 5, Restrictions on Subqueries](#).

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` or `ALTER TABLE` 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` 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. Many theoretically updatable views can be updated now, but limitations still exist. For details, see [Updatable and Insertable Views](#).

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` and `SELECT` privileges), that user cannot call `SHOW CREATE VIEW` on that object unless the user is also granted the `SHOW VIEW` privilege.

That shortcoming can lead to problems backing up a database with `mysqldump`, 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` privilege to users who are granted `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` 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` 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` 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` 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` and `CREATE VIEW`, or replace the definition with `CREATE OR REPLACE VIEW`.

For problems that occur when reloading view definitions in dump files, another workaround is to edit the dump file to modify its `CREATE VIEW` statements. However, this does not change the original view definitions, which may cause problems for subsequent dump operations.

Chapter 3 Restrictions on Condition Handling

`SIGNAL`, `RESIGNAL`, and `GET DIAGNOSTICS` 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.

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;
```

Chapter 4 Restrictions on Server-Side Cursors

Server-side cursors are implemented in the C API using the `mysql_stmt_attr_set()` function. The same implementation is used for cursors in stored routines. A server-side cursor enables a result set to be generated on the server side, but not transferred to the client except for those rows that the client requests. For example, if a client executes a query but is only interested in the first row, the remaining rows are not transferred.

In MySQL, a server-side cursor is materialized into an internal temporary table. Initially, this is a `MEMORY` table, but is converted to a `MyISAM` table when its size exceeds the minimum value of the `max_heap_table_size` and `tmp_table_size` system variables. The same restrictions apply to internal temporary tables created to hold the result set for a cursor as for other uses of internal temporary tables. See [Internal Temporary Table Use in MySQL](#). One limitation of the implementation is that for a large result set, retrieving its rows through a cursor might be slow.

Cursors are read only; you cannot use a cursor to update rows.

`UPDATE WHERE CURRENT OF` and `DELETE WHERE CURRENT OF` are not implemented, because updatable cursors are not supported.

Cursors are nonholdable (not held open after a commit).

Cursors are asensitive.

Cursors are nonscrollable.

Cursors are not named. The statement handler acts as the cursor ID.

You can have open only a single cursor per prepared statement. If you need several cursors, you must prepare several statements.

You cannot use a cursor for a statement that generates a result set if the statement is not supported in prepared mode. This includes statements such as `CHECK TABLE`, `HANDLER READ`, and `SHOW BINLOG EVENTS`.

Chapter 5 Restrictions on Subqueries

- In general, you cannot modify a table and select from the same table in a subquery. For example, this limitation applies to statements of the following forms:

```
DELETE FROM t WHERE ... (SELECT ... FROM t ...);
UPDATE t ... WHERE col = (SELECT ... FROM t ...);
{INSERT|REPLACE} INTO t (SELECT ... FROM t ...);
```

Exception: The preceding prohibition does not apply if for the modified table you are using a derived table and that derived table is materialized rather than merged into the outer query. (See [Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization.](#))

Example:

```
UPDATE t ... WHERE col = (SELECT * FROM (SELECT ... FROM t...) AS dt ...);
```

Here the result from the derived table is materialized as a temporary table, so the relevant rows in `t` have already been selected by the time the update to `t` takes place.

In general, you may be able to influence the optimizer to materialize a derived table by adding a `NO_MERGE` optimizer hint. See [Optimizer Hints](#).

- Row comparison operations are only partially supported:
 - For `expr [NOT] IN subquery`, `expr` can be an n -tuple (specified using row constructor syntax) and the subquery can return rows of n -tuples. The permitted syntax is therefore more specifically expressed as `row_constructor [NOT] IN table_subquery`
 - For `expr op {ALL|ANY|SOME} subquery`, `expr` must be a scalar value and the subquery must be a column subquery; it cannot return multiple-column rows.

In other words, for a subquery that returns rows of n -tuples, this is supported:

```
(expr_1, ..., expr_n) [NOT] IN table_subquery
```

But this is not supported:

```
(expr_1, ..., expr_n) op {ALL|ANY|SOME} subquery
```

The reason for supporting row comparisons for `IN` but not for the others is that `IN` is implemented by rewriting it as a sequence of `=` comparisons and `AND` operations. This approach cannot be used for `ALL`, `ANY`, or `SOME`.

- Prior to MySQL 8.0.14, subqueries in the `FROM` clause cannot be correlated subqueries. They are materialized in whole (evaluated to produce a result set) during query execution, so they cannot be evaluated per row of the outer query. The optimizer delays materialization until the result is needed, which may permit materialization to be avoided. See [Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization.](#)
- MySQL does not support `LIMIT` in subqueries for certain subquery operators:

```
mysql> SELECT * FROM t1
      WHERE s1 IN (SELECT s2 FROM t2 ORDER BY s1 LIMIT 1);
ERROR 1235 (42000): This version of MySQL doesn't yet support
'LIMIT & IN/ALL/ANY/SOME subquery'
```

See [Subquery Errors](#).

-
- MySQL permits a subquery to refer to a stored function that has data-modifying side effects such as inserting rows into a table. For example, if `f()` inserts rows, the following query can modify data:

```
SELECT ... WHERE x IN (SELECT f() ...);
```

This behavior is an extension to the SQL standard. In MySQL, it can produce nondeterministic results because `f()` might be executed a different number of times for different executions of a given query depending on how the optimizer chooses to handle it.

For statement-based or mixed-format replication, one implication of this indeterminism is that such a query can produce different results on the source and its replicas.

Chapter 6 Restrictions on XA Transactions

XA transaction support is limited to the [InnoDB](#) storage engine.

For “external XA,” a MySQL server acts as a Resource Manager and client programs act as Transaction Managers. For “Internal XA”, storage engines within a MySQL server act as RMs, and the server itself acts as a TM. Internal XA support is limited by the capabilities of individual storage engines. Internal XA is required for handling XA transactions that involve more than one storage engine. The implementation of internal XA requires that a storage engine support two-phase commit at the table handler level, and currently this is true only for [InnoDB](#).

For [XA START](#), the [JOIN](#) and [RESUME](#) clauses are recognized but have no effect.

For [XA END](#) the [SUSPEND \[FOR MIGRATE\]](#) clause is recognized but has no effect.

The requirement that the [bqual](#) part of the [xid](#) value be different for each XA transaction within a global transaction is a limitation of the current MySQL XA implementation. It is not part of the XA specification.

An XA transaction is written to the binary log in two parts. When [XA PREPARE](#) is issued, the first part of the transaction up to [XA PREPARE](#) is written using an initial GTID. A [XA_prepare_log_event](#) is used to identify such transactions in the binary log. When [XA COMMIT](#) or [XA ROLLBACK](#) is issued, a second part of the transaction containing only the [XA COMMIT](#) or [XA ROLLBACK](#) statement is written using a second GTID. Note that the initial part of the transaction, identified by [XA_prepare_log_event](#), is not necessarily followed by its [XA COMMIT](#) or [XA ROLLBACK](#), which can cause interleaved binary logging of any two XA transactions. The two parts of the XA transaction can even appear in different binary log files. This means that an XA transaction in [PREPARED](#) state is now persistent until an explicit [XA COMMIT](#) or [XA ROLLBACK](#) statement is issued, ensuring that XA transactions are compatible with replication.

On a replica, immediately after the XA transaction is prepared, it is detached from the replication applier thread, and can be committed or rolled back by any thread on the replica. This means that the same XA transaction can appear in the [events_transactions_current](#) table with different states on different threads. The [events_transactions_current](#) table displays the current status of the most recent monitored transaction event on the thread, and does not update this status when the thread is idle. So the XA transaction can still be displayed in the [PREPARED](#) state for the original applier thread, after it has been processed by another thread. To positively identify XA transactions that are still in the [PREPARED](#) state and need to be recovered, use the [XA RECOVER](#) statement rather than the Performance Schema transaction tables.

The following restrictions exist for using XA transactions:

- Prior to MySQL 8.0.30, XA transactions are not fully resilient to an unexpected halt with respect to the binary log. If there is an unexpected halt while the server is in the middle of executing an [XA PREPARE](#), [XA COMMIT](#), [XA ROLLBACK](#), or [XA COMMIT ... ONE PHASE](#) statement, the server might not be able to recover to a correct state, leaving the server and the binary log in an inconsistent state. In this situation, the binary log might either contain extra XA transactions that are not applied, or miss XA transactions that are applied. Also, if GTIDs are enabled, after recovery [@@GLOBAL.GTID_EXECUTED](#) might not correctly describe the transactions that have been applied. Note that if an unexpected halt occurs before [XA PREPARE](#), between [XA PREPARE](#) and [XA COMMIT](#) (or [XA ROLLBACK](#)), or after [XA COMMIT](#) (or [XA ROLLBACK](#)), the server and binary log are correctly recovered and taken to a consistent state.

Beginning with MySQL 8.0.30, this is no longer an issue; the server implements [XA PREPARE](#) as a two-phase operation, which maintains the state of the prepare operation between the storage engine and the server, and imposes order of execution between the storage engine and the binary log, so that state is not broadcast before it is consistent and persistent on the server node.

You should be aware that, when the same transaction XID is used to execute XA transactions sequentially and a break occurs during the processing of `XA COMMIT ... ONE PHASE`, it may no longer be possible to synchronize the state between the binary log and the storage engine. This can occur if the series of events just described takes place after this transaction has been prepared in the storage engine, while the `XA COMMIT` statement is still executing. This is a known issue.

- The use of replication filters or binary log filters in combination with XA transactions is not supported. Filtering of tables could cause an XA transaction to be empty on a replica, and empty XA transactions are not supported. Also, with the replica's connection metadata repository and applier metadata repository stored in `InnoDB` tables, which became the default in MySQL 8.0, the internal state of the data engine transaction is changed following a filtered XA transaction, and can become inconsistent with the replication transaction context state.

The error `ER_XA_REPLICATION_FILTERS` is logged whenever an XA transaction is impacted by a replication filter, whether or not the transaction was empty as a result. If the transaction is not empty, the replica is able to continue running, but you should take steps to discontinue the use of replication filters with XA transactions in order to avoid potential issues. If the transaction is empty, the replica stops. In that event, the replica might be in an undetermined state in which the consistency of the replication process might be compromised. In particular, the `gtid_executed` set on a replica of the replica might be inconsistent with that on the source. To resolve this situation, isolate the source and stop all replication, then check GTID consistency across the replication topology. Undo the XA transaction that generated the error message, then restart replication.

- XA transactions are considered unsafe for statement-based replication. If two XA transactions committed in parallel on the source are being prepared on the replica in the inverse order, locking dependencies can occur that cannot be safely resolved, and it is possible for replication to fail with deadlock on the replica. This situation can occur for a single-threaded or multithreaded replica. When `binlog_format=STATEMENT` is set, a warning is issued for DML statements inside XA transactions. When `binlog_format=MIXED` or `binlog_format=ROW` is set, DML statements inside XA transactions are logged using row-based replication, and the potential issue is not present.

Chapter 7 Restrictions on Character Sets

- Identifiers are stored in `mysql` database tables (`user`, `db`, and so forth) using `utf8mb3`, but identifiers can contain only characters in the Basic Multilingual Plane (BMP). Supplementary characters are not permitted in identifiers.
- The `ucs2`, `utf16`, `utf16le`, and `utf32` character sets have the following restrictions:
 - None of them can be used as the client character set. See [Impermissible Client Character Sets](#).
 - It is currently not possible to use `LOAD DATA` to load data files that use these character sets.
 - `FULLTEXT` indexes cannot be created on a column that uses any of these character sets. However, you can perform `IN BOOLEAN MODE` searches on the column without an index.
- The `REGEXP` and `RLIKE` operators work in byte-wise fashion, so they are not multibyte safe and may produce unexpected results with multibyte character sets. In addition, these operators compare characters by their byte values and accented characters may not compare as equal even if a given collation treats them as equal.

Chapter 8 Restrictions on Performance Schema

The Performance Schema avoids using mutexes to collect or produce data, so there are no guarantees of consistency and results can sometimes be incorrect. Event values in `performance_schema` tables are nondeterministic and nonrepeatable.

If you save event information in another table, you should not assume that the original events remain available later. For example, if you select events from a `performance_schema` table into a temporary table, intending to join that table with the original table later, there might be no matches.

`mysqldump` and `BACKUP DATABASE` ignore tables in the `performance_schema` database.

Tables in the `performance_schema` database cannot be locked with `LOCK TABLES`, except the `setup_XXX` tables.

Tables in the `performance_schema` database cannot be indexed.

Tables in the `performance_schema` database are not replicated.

The types of timers might vary per platform. The `performance_timers` table shows which event timers are available. If the values in this table for a given timer name are `NULL`, that timer is not supported on your platform.

Instruments that apply to storage engines might not be implemented for all storage engines. Instrumentation of each third-party engine is the responsibility of the engine maintainer.

Chapter 9 Restrictions on Pluggable Authentication

The first part of this section describes general restrictions on the applicability of the pluggable authentication framework described at [Pluggable Authentication](#). The second part describes how third-party connector developers can determine the extent to which a connector can take advantage of pluggable authentication capabilities and what steps to take to become more compliant.

The term “native authentication” used here refers to authentication against passwords stored in the `mysql.user` system table. This is the same authentication method provided by older MySQL servers, before pluggable authentication was implemented. “Windows native authentication” refers to authentication using the credentials of a user who has already logged in to Windows, as implemented by the Windows Native Authentication plugin (“Windows plugin” for short).

- [General Pluggable Authentication Restrictions](#)
- [Pluggable Authentication and Third-Party Connectors](#)

General Pluggable Authentication Restrictions

- **Connector/C++:** Clients that use this connector can connect to the server only through accounts that use native authentication.

Exception: A connector supports pluggable authentication if it was built to link to `libmysqlclient` dynamically (rather than statically) and it loads the current version of `libmysqlclient` if that version is installed, or if the connector is recompiled from source to link against the current `libmysqlclient`.

For information about writing connectors to handle information from the server about the default server-side authentication plugin, see [Authentication Plugin Connector-Writing Considerations](#).

- **Connector/NET:** Clients that use Connector/NET can connect to the server through accounts that use native authentication or Windows native authentication.
- **Connector/PHP:** Clients that use this connector can connect to the server only through accounts that use native authentication, when compiled using the MySQL native driver for PHP (`mysqlnd`).
- **Windows native authentication:** Connecting through an account that uses the Windows plugin requires Windows Domain setup. Without it, NTLM authentication is used and then only local connections are possible; that is, the client and server must run on the same computer.
- **Proxy users:** Proxy user support is available to the extent that clients can connect through accounts authenticated with plugins that implement proxy user capability (that is, plugins that can return a user name different from that of the connecting user). For example, the PAM and Windows plugins support proxy users. The `mysql_native_password` and `sha256_password` authentication plugins do not support proxy users by default, but can be configured to do so; see [Server Support for Proxy User Mapping](#).
- **Replication:** Replicas can not only employ replication user accounts using native authentication, but can also connect through replication user accounts that use nonnative authentication if the required client-side plugin is available. If the plugin is built into `libmysqlclient`, it is available by default. Otherwise, the plugin must be installed on the replica side in the directory named by the replica's `plugin_dir` system variable.
- **FEDERATED tables:** A `FEDERATED` table can access the remote table only through accounts on the remote server that use native authentication.

Pluggable Authentication and Third-Party Connectors

Third-party connector developers can use the following guidelines to determine readiness of a connector to take advantage of pluggable authentication capabilities and what steps to take to become more compliant:

- An existing connector to which no changes have been made uses native authentication and clients that use the connector can connect to the server only through accounts that use native authentication. *However, you should test the connector against a recent version of the server to verify that such connections still work without problem.*

Exception: A connector might work with pluggable authentication without any changes if it links to `libmysqlclient` dynamically (rather than statically) and it loads the current version of `libmysqlclient` if that version is installed.

- To take advantage of pluggable authentication capabilities, a connector that is `libmysqlclient`-based should be relinked against the current version of `libmysqlclient`. This enables the connector to support connections through accounts that require client-side plugins now built into `libmysqlclient` (such as the cleartext plugin needed for PAM authentication and the Windows plugin needed for Windows native authentication). Linking with a current `libmysqlclient` also enables the connector to access client-side plugins installed in the default MySQL plugin directory (typically the directory named by the default value of the local server's `plugin_dir` system variable).

If a connector links to `libmysqlclient` dynamically, it must be ensured that the newer version of `libmysqlclient` is installed on the client host and that the connector loads it at runtime.

- Another way for a connector to support a given authentication method is to implement it directly in the client/server protocol. Connector/NET uses this approach to provide support for Windows native authentication.
- If a connector should be able to load client-side plugins from a directory different from the default plugin directory, it must implement some means for client users to specify the directory. Possibilities for this include a command-line option or environment variable from which the connector can obtain the directory name. Standard MySQL client programs such as `mysql` and `mysqladmin` implement a `--plugin-dir` option. See also [C API Client Plugin Interface](#).
- Proxy user support by a connector depends, as described earlier in this section, on whether the authentication methods that it supports permit proxy users.

Chapter 10 Restrictions and Limitations on Partitioning

Table of Contents

10.1 Partitioning Keys, Primary Keys, and Unique Keys	29
10.2 Partitioning Limitations Relating to Storage Engines	32
10.3 Partitioning Limitations Relating to Functions	33

This section discusses current restrictions and limitations on MySQL partitioning support.

Prohibited constructs. The following constructs are not permitted in partitioning expressions:

- Stored procedures, stored functions, loadable functions, or plugins.
- Declared variables or user variables.

For a list of SQL functions which are permitted in partitioning expressions, see [Section 10.3, “Partitioning Limitations Relating to Functions”](#).

Arithmetic and logical operators. Use of the arithmetic operators `+`, `-`, and `*` is permitted in partitioning expressions. However, the result must be an integer value or `NULL` (except in the case of `[LINEAR] KEY` partitioning, as discussed elsewhere in this chapter; see [Partitioning Types](#), for more information).

The `DIV` operator is also supported; the `/` operator is not permitted.

The bit operators `|`, `&`, `^`, `<<`, `>>`, and `~` are not permitted in partitioning expressions.

Server SQL mode. Tables employing user-defined partitioning do not preserve the SQL mode in effect at the time that they were created. As discussed elsewhere in this Manual (see [Server SQL Modes](#)), the results of many MySQL functions and operators may change according to the server SQL mode. Therefore, a change in the SQL mode at any time after the creation of partitioned tables may lead to major changes in the behavior of such tables, and could easily lead to corruption or loss of data. For these reasons, *it is strongly recommended that you never change the server SQL mode after creating partitioned tables.*

For one such change in the server SQL mode making a partitioned tables unusable, consider the following `CREATE TABLE` statement, which can be executed successfully only if the `NO_UNSIGNED_SUBTRACTION` mode is in effect:

```
mysql> SELECT @@sql_mode;
+-----+
| @@sql_mode |
+-----+
|             |
+-----+
1 row in set (0.00 sec)
mysql> CREATE TABLE tu (c1 BIGINT UNSIGNED)
-> PARTITION BY RANGE(c1 - 10) (
-> PARTITION p0 VALUES LESS THAN (-5),
-> PARTITION p1 VALUES LESS THAN (0),
-> PARTITION p2 VALUES LESS THAN (5),
-> PARTITION p3 VALUES LESS THAN (10),
-> PARTITION p4 VALUES LESS THAN (MAXVALUE)
-> );
ERROR 1563 (HY000): Partition constant is out of partition function domain
```

```
mysql> SET sql_mode='NO_UNSIGNED_SUBTRACTION';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @@sql_mode;
+-----+
| @@sql_mode |
+-----+
| NO_UNSIGNED_SUBTRACTION |
+-----+
1 row in set (0.00 sec)
mysql> CREATE TABLE tu (c1 BIGINT UNSIGNED)
-> PARTITION BY RANGE(c1 - 10) (
-> PARTITION p0 VALUES LESS THAN (-5),
-> PARTITION p1 VALUES LESS THAN (0),
-> PARTITION p2 VALUES LESS THAN (5),
-> PARTITION p3 VALUES LESS THAN (10),
-> PARTITION p4 VALUES LESS THAN (MAXVALUE)
-> );
Query OK, 0 rows affected (0.05 sec)
```

If you remove the `NO_UNSIGNED_SUBTRACTION` server SQL mode after creating `tu`, you may no longer be able to access this table:

```
mysql> SET sql_mode='';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM tu;
ERROR 1563 (HY000): Partition constant is out of partition function domain
mysql> INSERT INTO tu VALUES (20);
ERROR 1563 (HY000): Partition constant is out of partition function domain
```

See also [Server SQL Modes](#).

Server SQL modes also impact replication of partitioned tables. Disparate SQL modes on source and replica can lead to partitioning expressions being evaluated differently; this can cause the distribution of data among partitions to be different in the source's and replica's copies of a given table, and may even cause inserts into partitioned tables that succeed on the source to fail on the replica. For best results, you should always use the same server SQL mode on the source and on the replica.

Performance considerations. Some effects of partitioning operations on performance are given in the following list:

- **File system operations.** Partitioning and repartitioning operations (such as `ALTER TABLE` with `PARTITION BY ...`, `REORGANIZE PARTITION`, or `REMOVE PARTITIONING`) depend on file system operations for their implementation. This means that the speed of these operations is affected by such factors as file system type and characteristics, disk speed, swap space, file handling efficiency of the operating system, and MySQL server options and variables that relate to file handling. In particular, you should make sure that `large_files_support` is enabled and that `open_files_limit` is set properly. Partitioning and repartitioning operations involving `InnoDB` tables may be made more efficient by enabling `innodb_file_per_table`.

See also [Maximum number of partitions](#).

- **Table locks.** Generally, the process executing a partitioning operation on a table takes a write lock on the table. Reads from such tables are relatively unaffected; pending `INSERT` and `UPDATE` operations are performed as soon as the partitioning operation has completed. For `InnoDB`-specific exceptions to this limitation, see [Partitioning Operations](#).
- **Indexes; partition pruning.** As with nonpartitioned tables, proper use of indexes can speed up queries on partitioned tables significantly. In addition, designing partitioned tables and queries on these tables to take advantage of *partition pruning* can improve performance dramatically. See [Partition Pruning](#), for more information.

Index condition pushdown is supported for partitioned tables. See [Index Condition Pushdown Optimization](#).

- **Performance with LOAD DATA.** In MySQL 8.0, `LOAD DATA` uses buffering to improve performance. You should be aware that the buffer uses 130 KB memory per partition to achieve this.

Maximum number of partitions.

The maximum possible number of partitions for a given table not using the `NDB` storage engine is 8192. This number includes subpartitions.

The maximum possible number of user-defined partitions for a table using the `NDB` storage engine is determined according to the version of the `NDB` Cluster software being used, the number of data nodes, and other factors. See [NDB and user-defined partitioning](#), for more information.

If, when creating tables with a large number of partitions (but less than the maximum), you encounter an error message such as `Got error ... from storage engine: Out of resources when opening file`, you may be able to address the issue by increasing the value of the `open_files_limit` system variable. However, this is dependent on the operating system, and may not be possible or advisable on all platforms; see [File Not Found and Similar Errors](#), for more information. In some cases, using large numbers (hundreds) of partitions may also not be advisable due to other concerns, so using more partitions does not automatically lead to better results.

See also [File system operations](#).

Foreign keys not supported for partitioned InnoDB tables.

Partitioned tables using the `InnoDB` storage engine do not support foreign keys. More specifically, this means that the following two statements are true:

1. No definition of an `InnoDB` table employing user-defined partitioning may contain foreign key references; no `InnoDB` table whose definition contains foreign key references may be partitioned.
2. No `InnoDB` table definition may contain a foreign key reference to a user-partitioned table; no `InnoDB` table with user-defined partitioning may contain columns referenced by foreign keys.

The scope of the restrictions just listed includes all tables that use the `InnoDB` storage engine. `CREATE TABLE` and `ALTER TABLE` statements that would result in tables violating these restrictions are not allowed.

ALTER TABLE ... ORDER BY. An `ALTER TABLE ... ORDER BY column` statement run against a partitioned table causes ordering of rows only within each partition.

ADD COLUMN ... ALGORITHM=INSTANT. Once you perform `ALTER TABLE ... ADD COLUMN ... ALGORITHM=INSTANT` on a partitioned table, it is no longer possible to exchange partitions with this table.

Effects on REPLACE statements by modification of primary keys. It can be desirable in some cases (see [Section 10.1, “Partitioning Keys, Primary Keys, and Unique Keys”](#)) to modify a table's primary key. Be aware that, if your application uses `REPLACE` statements and you do this, the results of these statements can be drastically altered. See [REPLACE Statement](#), for more information and an example.

FULLTEXT indexes.

Partitioned tables do not support `FULLTEXT` indexes or searches.

Spatial columns. Columns with spatial data types such as `POINT` or `GEOMETRY` cannot be used in partitioned tables.

Temporary tables.

Temporary tables cannot be partitioned.

Log tables. It is not possible to partition the log tables; an `ALTER TABLE ... PARTITION BY ...` statement on such a table fails with an error.

Data type of partitioning key.

A partitioning key must be either an integer column or an expression that resolves to an integer. Expressions employing `ENUM` columns cannot be used. The column or expression value may also be `NULL`; see [How MySQL Partitioning Handles NULL](#).

There are two exceptions to this restriction:

1. When partitioning by `[LINEAR] KEY`, it is possible to use columns of any valid MySQL data type other than `TEXT` or `BLOB` as partitioning keys, because the internal key-hashing functions produce the correct data type from these types. For example, the following two `CREATE TABLE` statements are valid:

```
CREATE TABLE tkc (c1 CHAR)
PARTITION BY KEY(c1)
PARTITIONS 4;
CREATE TABLE tke
  ( c1 ENUM('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet') )
PARTITION BY LINEAR KEY(c1)
PARTITIONS 6;
```

2. When partitioning by `RANGE COLUMNS` or `LIST COLUMNS`, it is possible to use string, `DATE`, and `DATETIME` columns. For example, each of the following `CREATE TABLE` statements is valid:

```
CREATE TABLE rc (c1 INT, c2 DATE)
PARTITION BY RANGE COLUMNS(c2) (
  PARTITION p0 VALUES LESS THAN('1990-01-01'),
  PARTITION p1 VALUES LESS THAN('1995-01-01'),
  PARTITION p2 VALUES LESS THAN('2000-01-01'),
  PARTITION p3 VALUES LESS THAN('2005-01-01'),
  PARTITION p4 VALUES LESS THAN(MAXVALUE)
);
CREATE TABLE lc (c1 INT, c2 CHAR(1))
PARTITION BY LIST COLUMNS(c2) (
  PARTITION p0 VALUES IN('a', 'd', 'g', 'j', 'm', 'p', 's', 'v', 'y'),
  PARTITION p1 VALUES IN('b', 'e', 'h', 'k', 'n', 'q', 't', 'w', 'z'),
  PARTITION p2 VALUES IN('c', 'f', 'i', 'l', 'o', 'r', 'u', 'x', NULL)
);
```

Neither of the preceding exceptions applies to `BLOB` or `TEXT` column types.

Subqueries.

A partitioning key may not be a subquery, even if that subquery resolves to an integer value or `NULL`.

Column index prefixes not supported for key partitioning. When creating a table that is partitioned by key, any columns in the partitioning key which use column prefixes are not used in the table's partitioning function. Consider the following `CREATE TABLE` statement, which has three `VARCHAR` columns, and whose primary key uses all three columns and specifies prefixes for two of them:

```
CREATE TABLE t1 (
  a VARCHAR(10000),
  b VARCHAR(25),
  c VARCHAR(10),
  PRIMARY KEY (a(10), b, c(2))
) PARTITION BY KEY() PARTITIONS 2;
```

This statement is accepted, but the resulting table is actually created as if you had issued the following statement, using only the primary key column which does not include a prefix (column `b`) for the partitioning key:


```
CREATE TABLE t1 (
  a VARCHAR(10000),
  b VARCHAR(25),
  c VARCHAR(10),
  PRIMARY KEY (a(10), b, c(2))
) PARTITION BY KEY(b) PARTITIONS 2;
```

Prior to MySQL 8.0.21, no warning was issued or any other indication provided that this occurred, except in the event that all columns specified for the partitioning key used prefixes, in which case the statement failed, but with a misleading error message, as shown here:

```
mysql> CREATE TABLE t2 (
->   a VARCHAR(10000),
->   b VARCHAR(25),
->   c VARCHAR(10),
->   PRIMARY KEY (a(10), b(5), c(2))
-> ) PARTITION BY KEY() PARTITIONS 2;
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the
table's partitioning function
```

This also occurred when performing `ALTER TABLE` or when upgrading such tables.

This permissive behavior is deprecated as of MySQL 8.0.21 (and is subject to removal in a future version of MySQL). Beginning with MySQL 8.0.21, using one or more columns having a prefix in the partitioning key results in a warning for each such column, as shown here:

```
mysql> CREATE TABLE t1 (
->   a VARCHAR(10000),
->   b VARCHAR(25),
->   c VARCHAR(10),
->   PRIMARY KEY (a(10), b, c(2))
-> ) PARTITION BY KEY() PARTITIONS 2;
Query OK, 0 rows affected, 2 warnings (1.25 sec)
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1681
Message: Column 'test.t1.a' having prefix key part 'a(10)' is ignored by the
partitioning function. Use of prefixed columns in the PARTITION BY KEY() clause
is deprecated and will be removed in a future release.
***** 2. row *****
Level: Warning
Code: 1681
Message: Column 'test.t1.c' having prefix key part 'c(2)' is ignored by the
partitioning function. Use of prefixed columns in the PARTITION BY KEY() clause
is deprecated and will be removed in a future release.
2 rows in set (0.00 sec)
```

This includes cases in which the columns used in the partitioning function are defined implicitly as those in the table's primary key by employing an empty `PARTITION BY KEY()` clause.

In MySQL 8.0.21 and later, if all columns specified for the partitioning key employ prefixes, the `CREATE TABLE` statement used fails with an error message that identifies the issue correctly:

```
mysql> CREATE TABLE t1 (
->   a VARCHAR(10000),
->   b VARCHAR(25),
->   c VARCHAR(10),
->   PRIMARY KEY (a(10), b(5), c(2))
-> ) PARTITION BY KEY() PARTITIONS 2;
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's
partitioning function (prefixed columns are not considered).
```

For general information about partitioning tables by key, see [KEY Partitioning](#).

Issues with subpartitions.

Subpartitions must use [HASH](#) or [KEY](#) partitioning. Only [RANGE](#) and [LIST](#) partitions may be subpartitioned; [HASH](#) and [KEY](#) partitions cannot be subpartitioned.

[SUBPARTITION BY KEY](#) requires that the subpartitioning column or columns be specified explicitly, unlike the case with [PARTITION BY KEY](#), where it can be omitted (in which case the table's primary key column is used by default). Consider the table created by this statement:

```
CREATE TABLE ts (  
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(30)  
);
```

You can create a table having the same columns, partitioned by [KEY](#), using a statement such as this one:

```
CREATE TABLE ts (  
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(30)  
)  
PARTITION BY KEY()  
PARTITIONS 4;
```

The previous statement is treated as though it had been written like this, with the table's primary key column used as the partitioning column:

```
CREATE TABLE ts (  
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(30)  
)  
PARTITION BY KEY(id)  
PARTITIONS 4;
```

However, the following statement that attempts to create a subpartitioned table using the default column as the subpartitioning column fails, and the column must be specified for the statement to succeed, as shown here:

```
mysql> CREATE TABLE ts (  
->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
->   name VARCHAR(30)  
-> )  
-> PARTITION BY RANGE(id)  
-> SUBPARTITION BY KEY()  
-> SUBPARTITIONS 4  
-> (  
->   PARTITION p0 VALUES LESS THAN (100),  
->   PARTITION p1 VALUES LESS THAN (MAXVALUE)  
-> );  
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that  
corresponds to your MySQL server version for the right syntax to use near '  
mysql> CREATE TABLE ts (  
->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
->   name VARCHAR(30)  
-> )  
-> PARTITION BY RANGE(id)  
-> SUBPARTITION BY KEY(id)  
-> SUBPARTITIONS 4  
-> (  
->   PARTITION p0 VALUES LESS THAN (100),  
->   PARTITION p1 VALUES LESS THAN (MAXVALUE)  
-> );  
Query OK, 0 rows affected (0.07 sec)
```

This is a known issue (see [Bug #51470](#)).

DATA DIRECTORY and INDEX DIRECTORY options. Table-level `DATA DIRECTORY` and `INDEX DIRECTORY` options are ignored (see Bug #32091). You can employ these options for individual partitions or subpartitions of InnoDB tables. As of MySQL 8.0.21, the directory specified in a `DATA DIRECTORY` clause must be known to InnoDB. For more information, see [Using the DATA DIRECTORY Clause](#).

Repairing and rebuilding partitioned tables. The statements `CHECK TABLE`, `OPTIMIZE TABLE`, `ANALYZE TABLE`, and `REPAIR TABLE` are supported for partitioned tables.

In addition, you can use `ALTER TABLE ... REBUILD PARTITION` to rebuild one or more partitions of a partitioned table; `ALTER TABLE ... REORGANIZE PARTITION` also causes partitions to be rebuilt. See [ALTER TABLE Statement](#), for more information about these two statements.

`ANALYZE`, `CHECK`, `OPTIMIZE`, `REPAIR`, and `TRUNCATE` operations are supported with subpartitions. See [ALTER TABLE Partition Operations](#).

File name delimiters for partitions and subpartitions. Table partition and subpartition file names include generated delimiters such as `#P#` and `#SP#`. The lettercase of such delimiters can vary and should not be depended upon.

10.1 Partitioning Keys, Primary Keys, and Unique Keys

This section discusses the relationship of partitioning keys with primary keys and unique keys. The rule governing this relationship can be expressed as follows: All columns used in the partitioning expression for a partitioned table must be part of every unique key that the table may have.

In other words, *every unique key on the table must use every column in the table's partitioning expression*. (This also includes the table's primary key, since it is by definition a unique key. This particular case is discussed later in this section.) For example, each of the following table creation statements is invalid:

```
CREATE TABLE t1 (
  col1 INT NOT NULL,
  col2 DATE NOT NULL,
  col3 INT NOT NULL,
  col4 INT NOT NULL,
  UNIQUE KEY (col1, col2)
)
PARTITION BY HASH(col3)
PARTITIONS 4;
CREATE TABLE t2 (
  col1 INT NOT NULL,
  col2 DATE NOT NULL,
  col3 INT NOT NULL,
  col4 INT NOT NULL,
  UNIQUE KEY (col1),
  UNIQUE KEY (col3)
)
PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

In each case, the proposed table would have at least one unique key that does not include all columns used in the partitioning expression.

Each of the following statements is valid, and represents one way in which the corresponding invalid table creation statement could be made to work:

```
CREATE TABLE t1 (
  col1 INT NOT NULL,
  col2 DATE NOT NULL,
  col3 INT NOT NULL,
```

```

    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3)
)
PARTITION BY HASH(col3)
PARTITIONS 4;
CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3)
)
PARTITION BY HASH(col1 + col3)
PARTITIONS 4;

```

This example shows the error produced in such cases:

```

mysql> CREATE TABLE t3 (
->     col1 INT NOT NULL,
->     col2 DATE NOT NULL,
->     col3 INT NOT NULL,
->     col4 INT NOT NULL,
->     UNIQUE KEY (col1, col2),
->     UNIQUE KEY (col3)
-> )
-> PARTITION BY HASH(col1 + col3)
-> PARTITIONS 4;
ERROR 1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function

```

The `CREATE TABLE` statement fails because both `col1` and `col3` are included in the proposed partitioning key, but neither of these columns is part of both of unique keys on the table. This shows one possible fix for the invalid table definition:

```

mysql> CREATE TABLE t3 (
->     col1 INT NOT NULL,
->     col2 DATE NOT NULL,
->     col3 INT NOT NULL,
->     col4 INT NOT NULL,
->     UNIQUE KEY (col1, col2, col3),
->     UNIQUE KEY (col3)
-> )
-> PARTITION BY HASH(col3)
-> PARTITIONS 4;
Query OK, 0 rows affected (0.05 sec)

```

In this case, the proposed partitioning key `col3` is part of both unique keys, and the table creation statement succeeds.

The following table cannot be partitioned at all, because there is no way to include in a partitioning key any columns that belong to both unique keys:

```

CREATE TABLE t4 (
    col1 INT NOT NULL,
    col2 INT NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3),
    UNIQUE KEY (col2, col4)
);

```

Since every primary key is by definition a unique key, this restriction also includes the table's primary key, if it has one. For example, the next two statements are invalid:

```

CREATE TABLE t5 (

```

```

    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)
PARTITION BY HASH(col3)
PARTITIONS 4;
CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col3),
    UNIQUE KEY(col2)
)
PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;

```

In both cases, the primary key does not include all columns referenced in the partitioning expression. However, both of the next two statements are valid:

```

CREATE TABLE t7 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)
PARTITION BY HASH(col1 + YEAR(col2))
PARTITIONS 4;
CREATE TABLE t8 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col4),
    UNIQUE KEY(col2, col1)
)
PARTITION BY HASH(col1 + YEAR(col2))
PARTITIONS 4;

```

If a table has no unique keys—this includes having no primary key—then this restriction does not apply, and you may use any column or columns in the partitioning expression as long as the column type is compatible with the partitioning type.

For the same reason, you cannot later add a unique key to a partitioned table unless the key includes all columns used by the table's partitioning expression. Consider the partitioned table created as shown here:

```

mysql> CREATE TABLE t_no_pk (c1 INT, c2 INT)
->     PARTITION BY RANGE(c1) (
->         PARTITION p0 VALUES LESS THAN (10),
->         PARTITION p1 VALUES LESS THAN (20),
->         PARTITION p2 VALUES LESS THAN (30),
->         PARTITION p3 VALUES LESS THAN (40)
->     );
Query OK, 0 rows affected (0.12 sec)

```

It is possible to add a primary key to `t_no_pk` using either of these `ALTER TABLE` statements:

```

# possible PK
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c1);
Query OK, 0 rows affected (0.13 sec)
Records: 0 Duplicates: 0 Warnings: 0
# drop this PK

```

```
mysql> ALTER TABLE t_no_pk DROP PRIMARY KEY;
Query OK, 0 rows affected (0.10 sec)
Records: 0 Duplicates: 0 Warnings: 0
# use another possible PK
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c1, c2);
Query OK, 0 rows affected (0.12 sec)
Records: 0 Duplicates: 0 Warnings: 0
# drop this PK
mysql> ALTER TABLE t_no_pk DROP PRIMARY KEY;
Query OK, 0 rows affected (0.09 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

However, the next statement fails, because `c1` is part of the partitioning key, but is not part of the proposed primary key:

```
# fails with error 1503
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c2);
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

Since `t_no_pk` has only `c1` in its partitioning expression, attempting to adding a unique key on `c2` alone fails. However, you can add a unique key that uses both `c1` and `c2`.

These rules also apply to existing nonpartitioned tables that you wish to partition using `ALTER TABLE ... PARTITION BY`. Consider a table `np_pk` created as shown here:

```
mysql> CREATE TABLE np_pk (
->   id INT NOT NULL AUTO_INCREMENT,
->   name VARCHAR(50),
->   added DATE,
->   PRIMARY KEY (id)
-> );
Query OK, 0 rows affected (0.08 sec)
```

The following `ALTER TABLE` statement fails with an error, because the `added` column is not part of any unique key in the table:

```
mysql> ALTER TABLE np_pk
->   PARTITION BY HASH( TO_DAYS(added) )
->   PARTITIONS 4;
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

However, this statement using the `id` column for the partitioning column is valid, as shown here:

```
mysql> ALTER TABLE np_pk
->   PARTITION BY HASH(id)
->   PARTITIONS 4;
Query OK, 0 rows affected (0.11 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

In the case of `np_pk`, the only column that may be used as part of a partitioning expression is `id`; if you wish to partition this table using any other column or columns in the partitioning expression, you must first modify the table, either by adding the desired column or columns to the primary key, or by dropping the primary key altogether.

10.2 Partitioning Limitations Relating to Storage Engines

In MySQL 8.0, partitioning support is not actually provided by the MySQL Server, but rather by a table storage engine's own or native partitioning handler. In MySQL 8.0, only the `InnoDB` and `NDB` storage engines provide native partitioning handlers. This means that partitioned tables cannot be created using any other storage engine than these. (You must be using MySQL NDB Cluster with the `NDB` storage engine to create `NDB` tables.)

InnoDB storage engine. InnoDB foreign keys and MySQL partitioning are not compatible. Partitioned InnoDB tables cannot have foreign key references, nor can they have columns referenced by foreign keys. InnoDB tables which have or which are referenced by foreign keys cannot be partitioned.

ALTER TABLE ... OPTIMIZE PARTITION does not work correctly with partitioned tables that use InnoDB. Use ALTER TABLE ... REBUILD PARTITION and ALTER TABLE ... ANALYZE PARTITION, instead, for such tables. For more information, see ALTER TABLE Partition Operations.

User-defined partitioning and the NDB storage engine (NDB Cluster). Partitioning by KEY (including LINEAR KEY) is the only type of partitioning supported for the NDB storage engine. It is not possible under normal circumstances in NDB Cluster to create an NDB Cluster table using any partitioning type other than [LINEAR] KEY, and attempting to do so fails with an error.

Exception (not for production): It is possible to override this restriction by setting the new system variable on NDB Cluster SQL nodes to ON. If you choose to do this, you should be aware that tables using partitioning types other than [LINEAR] KEY are not supported in production. *In such cases, you can create and use tables with partitioning types other than KEY or LINEAR KEY, but you do this entirely at your own risk.* You should also be aware that this functionality is now deprecated and subject to removal without further notice in a future release of NDB Cluster.

The maximum number of partitions that can be defined for an NDB table depends on the number of data nodes and node groups in the cluster, the version of the NDB Cluster software in use, and other factors. See NDB and user-defined partitioning, for more information.

The maximum amount of fixed-size data that can be stored per partition in an NDB table is 128 TB. Previously, this was 16 GB.

CREATE TABLE and ALTER TABLE statements that would cause a user-partitioned NDB table not to meet either or both of the following two requirements are not permitted, and fail with an error:

1. The table must have an explicit primary key.
2. All columns listed in the table's partitioning expression must be part of the primary key.

Exception. If a user-partitioned NDB table is created using an empty column-list (that is, using PARTITION BY KEY() or PARTITION BY LINEAR KEY()), then no explicit primary key is required.

Partition selection. Partition selection is not supported for NDB tables. See Partition Selection, for more information.

Upgrading partitioned tables. When performing an upgrade, tables which are partitioned by KEY must be dumped and reloaded. Partitioned tables using storage engines other than InnoDB cannot be upgraded from MySQL 5.7 or earlier to MySQL 8.0 or later; you must either drop the partitioning from such tables with ALTER TABLE ... REMOVE PARTITIONING or convert them to InnoDB using ALTER TABLE ... ENGINE=INNODB prior to the upgrade.

For information about converting MyISAM tables to InnoDB, see Converting Tables from MyISAM to InnoDB.

10.3 Partitioning Limitations Relating to Functions

This section discusses limitations in MySQL Partitioning relating specifically to functions used in partitioning expressions.

Only the MySQL functions shown in the following list are allowed in partitioning expressions:

- ABS()

- `CEILING()` (see [CEILING\(\) and FLOOR\(\)](#))
- `DATEDIFF()`
- `DAY()`
- `DAYOFMONTH()`
- `DAYOFWEEK()`
- `DAYOFYEAR()`
- `EXTRACT()` (see [EXTRACT\(\) function with WEEK specifier](#))
- `FLOOR()` (see [CEILING\(\) and FLOOR\(\)](#))
- `HOUR()`
- `MICROSECOND()`
- `MINUTE()`
- `MOD()`
- `MONTH()`
- `QUARTER()`
- `SECOND()`
- `TIME_TO_SEC()`
- `TO_DAYS()`
- `TO_SECONDS()`
- `UNIX_TIMESTAMP()` (with `TIMESTAMP` columns)
- `WEEKDAY()`
- `YEAR()`
- `YEARWEEK()`

In MySQL 8.0, partition pruning is supported for the `TO_DAYS()`, `TO_SECONDS()`, `YEAR()`, and `UNIX_TIMESTAMP()` functions. See [Partition Pruning](#), for more information.

CEILING() and FLOOR(). Each of these functions returns an integer only if it is passed an argument of an exact numeric type, such as one of the `INT` types or `DECIMAL`. This means, for example, that the following `CREATE TABLE` statement fails with an error, as shown here:

```
mysql> CREATE TABLE t (c FLOAT) PARTITION BY LIST( FLOOR(c) )(
->     PARTITION p0 VALUES IN (1,3,5),
->     PARTITION p1 VALUES IN (2,4,6)
-> );
ERROR 1490 (HY000): The PARTITION function returns the wrong type
```

EXTRACT() function with WEEK specifier. The value returned by the `EXTRACT()` function, when used as `EXTRACT(WEEK FROM col)`, depends on the value of the `default_week_format` system

variable. For this reason, [EXTRACT\(\)](#) is not permitted as a partitioning function when it specifies the unit as [WEEK](#). (Bug #54483)

See [Mathematical Functions](#), for more information about the return types of these functions, as well as [Numeric Data Types](#).

Chapter 11 Windows Platform Restrictions

The following restrictions apply to use of MySQL on the Windows platform:

- **Process memory**

On Windows 32-bit platforms, it is not possible by default to use more than 2GB of RAM within a single process, including MySQL. This is because the physical address limit on Windows 32-bit is 4GB and the default setting within Windows is to split the virtual address space between kernel (2GB) and user/applications (2GB).

Some versions of Windows have a boot time setting to enable larger applications by reducing the kernel application. Alternatively, to use more than 2GB, use a 64-bit version of Windows.

- **File system aliases**

When using `MyISAM` tables, you cannot use aliases within Windows link to the data files on another volume and then link back to the main MySQL `datadir` location.

This facility is often used to move the data and index files to a RAID or other fast solution.

- **Limited number of ports**

Windows systems have about 4,000 ports available for client connections, and after a connection on a port closes, it takes two to four minutes before the port can be reused. In situations where clients connect to and disconnect from the server at a high rate, it is possible for all available ports to be used up before closed ports become available again. If this happens, the MySQL server appears to be unresponsive even though it is running. Ports may be used by other applications running on the machine as well, in which case the number of ports available to MySQL is lower.

For more information about this problem, see <https://support.microsoft.com/kb/196271>.

- **DATA DIRECTORY and INDEX DIRECTORY**

The `DATA DIRECTORY` clause of the `CREATE TABLE` statement is supported on Windows for `InnoDB` tables only, as described in [Creating Tables Externally](#). For `MyISAM` and other storage engines, the `DATA DIRECTORY` and `INDEX DIRECTORY` clauses for `CREATE TABLE` are ignored on Windows and any other platforms with a nonfunctional `realpath()` call.

- **DROP DATABASE**

You cannot drop a database that is in use by another session.

- **Case-insensitive names**

File names are not case-sensitive on Windows, so MySQL database and table names are also not case-sensitive on Windows. The only restriction is that database and table names must be specified using the same case throughout a given statement. See [Identifier Case Sensitivity](#).

- **Directory and file names**

On Windows, MySQL Server supports only directory and file names that are compatible with the current ANSI code pages. For example, the following Japanese directory name does not work in the Western locale (code page 1252):

```
datadir="C:/私たちのプロジェクトのデータ"
```

The same limitation applies to directory and file names referred to in SQL statements, such as the data file path name in `LOAD DATA`.

- **The `\` path name separator character**

Path name components in Windows are separated by the `\` character, which is also the escape character in MySQL. If you are using `LOAD DATA` or `SELECT ... INTO OUTFILE`, use Unix-style file names with `/` characters:

```
mysql> LOAD DATA INFILE 'C:/tmp/skr.txt' INTO TABLE skr;  
mysql> SELECT * INTO OUTFILE 'C:/tmp/skr.txt' FROM skr;
```

Alternatively, you must double the `\` character:

```
mysql> LOAD DATA INFILE 'C:\\tmp\\skr.txt' INTO TABLE skr;  
mysql> SELECT * INTO OUTFILE 'C:\\tmp\\skr.txt' FROM skr;
```

- **Problems with pipes**

Pipes do not work reliably from the Windows command-line prompt. If the pipe includes the character `^Z` / `CHAR(24)`, Windows thinks that it has encountered end-of-file and aborts the program.

This is mainly a problem when you try to apply a binary log as follows:

```
C:\> mysqlbinlog binary_log_file | mysql --user=root
```

If you have a problem applying the log and suspect that it is because of a `^Z` / `CHAR(24)` character, you can use the following workaround:

```
C:\> mysqlbinlog binary_log_file --result-file=/tmp/bin.sql  
C:\> mysql --user=root --execute "source /tmp/bin.sql"
```

The latter command also can be used to reliably read any SQL file that may contain binary data.

Chapter 12 Limits in MySQL

Table of Contents

12.1 Identifier Length Limits	39
12.2 Grant Table Scope Column Properties	40
12.3 Limits on Number of Databases and Tables	40
12.4 Limits on Table Size	40
12.5 Limits on Table Column Count and Row Size	41

This chapter lists current limits in MySQL 8.0.

12.1 Identifier Length Limits

The following table describes the maximum length for each type of identifier.

Identifier Type	Maximum Length (characters)
Database	64 (includes NDB Cluster 8.0.18 and later)
Table	64 (includes NDB Cluster 8.0.18 and later)
Column	64
Index	64
Constraint	64
Stored Program	64
View	64
Tablespace	64
Server	64
Log File Group	64
Alias	256 (see exception following table)
Compound Statement Label	16
User-Defined Variable	64
Resource Group	64

Aliases for column names in `CREATE VIEW` statements are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters).

For constraint definitions that include no constraint name, the server internally generates a name derived from the associated table name. For example, internally generated foreign key and `CHECK` constraint names consist of the table name plus `_ibfk_` or `_chk_` and a number. If the table name is close to the length limit for constraint names, the additional characters required for the constraint name may cause that name to exceed the limit, resulting in an error.

Identifiers are stored using Unicode (UTF-8). This applies to identifiers in table definitions and to identifiers stored in the grant tables in the `mysql` database. The sizes of the identifier string columns in the grant tables are measured in characters. You can use multibyte characters without reducing the number of characters permitted for values stored in these columns.

Prior to NDB 8.0.18, NDB Cluster imposed a maximum length of 63 characters for names of databases and tables. As of NDB 8.0.18, this limitation is removed. See [Previous NDB Cluster Issues Resolved in NDB Cluster 8.0](#).

Values such as user name and host names in MySQL account names are strings rather than identifiers. For information about the maximum length of such values as stored in grant tables, see [Section 12.2, “Grant Table Scope Column Properties”](#).

12.2 Grant Table Scope Column Properties

Scope columns in the grant tables contain strings. The default value for each is the empty string. The following table shows the number of characters permitted in each column.

Table 12.1 Grant Table Scope Column Lengths

Column Name	Maximum Permitted Characters
<code>Host</code> , <code>Proxied_host</code>	255 (60 prior to MySQL 8.0.17)
<code>User</code> , <code>Proxied_user</code>	32
<code>Db</code>	64
<code>Table_name</code>	64
<code>Column_name</code>	64
<code>Routine_name</code>	64

`Host` and `Proxied_host` values are converted to lowercase before being stored in the grant tables.

For access-checking purposes, comparisons of `User`, `Proxied_user`, `authentication_string`, `Db`, and `Table_name` values are case-sensitive. Comparisons of `Host`, `Proxied_host`, `Column_name`, and `Routine_name` values are not case-sensitive.

12.3 Limits on Number of Databases and Tables

MySQL has no limit on the number of databases. The underlying file system may have a limit on the number of directories.

MySQL has no limit on the number of tables. The underlying file system may have a limit on the number of files that represent tables. Individual storage engines may impose engine-specific constraints. [InnoDB](#) permits up to 4 billion tables.

12.4 Limits on Table Size

The effective maximum table size for MySQL databases is usually determined by operating system constraints on file sizes, not by MySQL internal limits. For up-to-date information operating system file size limits, refer to the documentation specific to your operating system.

Windows users, please note that FAT and VFAT (FAT32) are *not* considered suitable for production use with MySQL. Use NTFS instead.

If you encounter a full-table error, there are several reasons why it might have occurred:

- The disk might be full.
- You are using [InnoDB](#) tables and have run out of room in an [InnoDB](#) tablespace file. The maximum tablespace size is also the maximum size for a table. For tablespace size limits, see [InnoDB Limits](#).

Generally, partitioning of tables into multiple tablespace files is recommended for tables larger than 1TB in size.

- You have hit an operating system file size limit. For example, you are using [MyISAM](#) tables on an operating system that supports files only up to 2GB in size and you have hit this limit for the data file or index file.
- You are using a [MyISAM](#) table and the space required for the table exceeds what is permitted by the internal pointer size. [MyISAM](#) permits data and index files to grow up to 256TB by default, but this limit can be changed up to the maximum permissible size of 65,536TB ($256^7 - 1$ bytes).

If you need a [MyISAM](#) table that is larger than the default limit and your operating system supports large files, the `CREATE TABLE` statement supports `AVG_ROW_LENGTH` and `MAX_ROWS` options. See [CREATE TABLE Statement](#). The server uses these options to determine how large a table to permit.

If the pointer size is too small for an existing table, you can change the options with `ALTER TABLE` to increase a table's maximum permissible size. See [ALTER TABLE Statement](#).

```
ALTER TABLE tbl_name MAX_ROWS=1000000000 AVG_ROW_LENGTH=nnn;
```

You have to specify `AVG_ROW_LENGTH` only for tables with `BLOB` or `TEXT` columns; in this case, MySQL cannot optimize the space required based only on the number of rows.

To change the default size limit for [MyISAM](#) tables, set the `myisam_data_pointer_size`, which sets the number of bytes used for internal row pointers. The value is used to set the pointer size for new tables if you do not specify the `MAX_ROWS` option. The value of `myisam_data_pointer_size` can be from 2 to 7. For example, for tables that use the dynamic storage format, a value of 4 permits tables up to 4GB; a value of 6 permits tables up to 256TB. Tables that use the fixed storage format have a larger maximum data length. For storage format characteristics, see [MyISAM Table Storage Formats](#).

You can check the maximum data and index sizes by using this statement:

```
SHOW TABLE STATUS FROM db_name LIKE 'tbl_name';
```

You also can use `myisamchk -dv /path/to/table-index-file`. See [SHOW Statements](#), or [myisamchk — MyISAM Table-Maintenance Utility](#).

Other ways to work around file-size limits for [MyISAM](#) tables are as follows:

- If your large table is read only, you can use `myisampack` to compress it. `myisampack` usually compresses a table by at least 50%, so you can have, in effect, much bigger tables. `myisampack` also can merge multiple tables into a single table. See [myisampack — Generate Compressed, Read-Only MyISAM Tables](#).
- MySQL includes a `MERGE` library that enables you to handle a collection of [MyISAM](#) tables that have identical structure as a single `MERGE` table. See [The MERGE Storage Engine](#).
- You are using the `MEMORY (HEAP)` storage engine; in this case you need to increase the value of the `max_heap_table_size` system variable. See [Server System Variables](#).

12.5 Limits on Table Column Count and Row Size

This section describes limits on the number of columns in tables and the size of individual rows.

- [Column Count Limits](#)
- [Row Size Limits](#)

Column Count Limits

MySQL has a hard limit of 4096 columns per table, but the effective maximum may be less for a given table. The exact column limit depends on several factors:

- The maximum row size for a table constrains the number (and possibly size) of columns because the total length of all columns cannot exceed this size. See [Row Size Limits](#).
- The storage requirements of individual columns constrain the number of columns that fit within a given maximum row size. Storage requirements for some data types depend on factors such as storage engine, storage format, and character set. See [Data Type Storage Requirements](#).
- Storage engines may impose additional restrictions that limit table column count. For example, [InnoDB](#) has a limit of 1017 columns per table. See [InnoDB Limits](#). For information about other storage engines, see [Alternative Storage Engines](#).
- Functional key parts (see [CREATE INDEX Statement](#)) are implemented as hidden virtual generated stored columns, so each functional key part in a table index counts against the table total column limit.

Row Size Limits

The maximum row size for a given table is determined by several factors:

- The internal representation of a MySQL table has a maximum row size limit of 65,535 bytes, even if the storage engine is capable of supporting larger rows. [BLOB](#) and [TEXT](#) columns only contribute 9 to 12 bytes toward the row size limit because their contents are stored separately from the rest of the row.
- The maximum row size for an [InnoDB](#) table, which applies to data stored locally within a database page, is slightly less than half a page for 4KB, 8KB, 16KB, and 32KB [innodb_page_size](#) settings. For example, the maximum row size is slightly less than 8KB for the default 16KB [InnoDB](#) page size. For 64KB pages, the maximum row size is slightly less than 16KB. See [InnoDB Limits](#).

If a row containing [variable-length columns](#) exceeds the [InnoDB](#) maximum row size, [InnoDB](#) selects variable-length columns for external off-page storage until the row fits within the [InnoDB](#) row size limit. The amount of data stored locally for variable-length columns that are stored off-page differs by row format. For more information, see [InnoDB Row Formats](#).

- Different storage formats use different amounts of page header and trailer data, which affects the amount of storage available for rows.
 - For information about [InnoDB](#) row formats, see [InnoDB Row Formats](#).
 - For information about [MyISAM](#) storage formats, see [MyISAM Table Storage Formats](#).

Row Size Limit Examples

- The MySQL maximum row size limit of 65,535 bytes is demonstrated in the following [InnoDB](#) and [MyISAM](#) examples. The limit is enforced regardless of storage engine, even though the storage engine may be capable of supporting larger rows.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
  c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
  f VARCHAR(10000), g VARCHAR(6000)) ENGINE=InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
  c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
```



```
f VARCHAR(10000), g VARCHAR(6000)) ENGINE=MyISAM CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

In the following [MyISAM](#) example, changing a column to [TEXT](#) avoids the 65,535-byte row size limit and permits the operation to succeed because [BLOB](#) and [TEXT](#) columns only contribute 9 to 12 bytes toward the row size.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
f VARCHAR(10000), g TEXT(6000)) ENGINE=MyISAM CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

The operation succeeds for an [InnoDB](#) table because changing a column to [TEXT](#) avoids the MySQL 65,535-byte row size limit, and [InnoDB](#) off-page storage of variable-length columns avoids the [InnoDB](#) row size limit.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
f VARCHAR(10000), g TEXT(6000)) ENGINE=InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

- Storage for variable-length columns includes length bytes, which are counted toward the row size. For example, a [VARCHAR\(255\) CHARACTER SET utf8mb3](#) column takes two bytes to store the length of the value, so each value can take up to 767 bytes.

The statement to create table `t1` succeeds because the columns require 32,765 + 2 bytes and 32,766 + 2 bytes, which falls within the maximum row size of 65,535 bytes:

```
mysql> CREATE TABLE t1
(c1 VARCHAR(32765) NOT NULL, c2 VARCHAR(32766) NOT NULL)
ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

The statement to create table `t2` fails because, although the column length is within the maximum length of 65,535 bytes, two additional bytes are required to record the length, which causes the row size to exceed 65,535 bytes:

```
mysql> CREATE TABLE t2
(c1 VARCHAR(65535) NOT NULL)
ENGINE = InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

Reducing the column length to 65,533 or less permits the statement to succeed.

```
mysql> CREATE TABLE t2
(c1 VARCHAR(65533) NOT NULL)
ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.01 sec)
```

- For [MyISAM](#) tables, [NULL](#) columns require additional space in the row to record whether their values are [NULL](#). Each [NULL](#) column takes one bit extra, rounded up to the nearest byte.

The statement to create table `t3` fails because [MyISAM](#) requires space for [NULL](#) columns in addition to the space required for variable-length column length bytes, causing the row size to exceed 65,535 bytes:

```
mysql> CREATE TABLE t3
(c1 VARCHAR(32765) NULL, c2 VARCHAR(32766) NULL)
ENGINE = MyISAM CHARACTER SET latin1;
```

```
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. This includes storage overhead, check the manual. You have to change some columns to TEXT or BLOBs
```

For information about [InnoDB NULL](#) column storage, see [InnoDB Row Formats](#).

- [InnoDB](#) restricts row size (for data stored locally within the database page) to slightly less than half a database page for 4KB, 8KB, 16KB, and 32KB `innodb_page_size` settings, and to slightly less than 16KB for 64KB pages.

The statement to create table `t4` fails because the defined columns exceed the row size limit for a 16KB [InnoDB](#) page.

```
mysql> CREATE TABLE t4 (  
  c1 CHAR(255),c2 CHAR(255),c3 CHAR(255),  
  c4 CHAR(255),c5 CHAR(255),c6 CHAR(255),  
  c7 CHAR(255),c8 CHAR(255),c9 CHAR(255),  
  c10 CHAR(255),c11 CHAR(255),c12 CHAR(255),  
  c13 CHAR(255),c14 CHAR(255),c15 CHAR(255),  
  c16 CHAR(255),c17 CHAR(255),c18 CHAR(255),  
  c19 CHAR(255),c20 CHAR(255),c21 CHAR(255),  
  c22 CHAR(255),c23 CHAR(255),c24 CHAR(255),  
  c25 CHAR(255),c26 CHAR(255),c27 CHAR(255),  
  c28 CHAR(255),c29 CHAR(255),c30 CHAR(255),  
  c31 CHAR(255),c32 CHAR(255),c33 CHAR(255)  
  ) ENGINE=InnoDB ROW_FORMAT=DYNAMIC DEFAULT CHARSET latin1;  
ERROR 1118 (42000): Row size too large (> 8126). Changing some columns to TEXT or BLOB may help.  
In current row format, BLOB prefix of 0 bytes is stored inline.
```

Chapter 13 MySQL Differences from Standard SQL

Table of Contents

13.1 SELECT INTO TABLE Differences	45
13.2 UPDATE Differences	45
13.3 FOREIGN KEY Constraint Differences	45
13.4 '--' as the Start of a Comment	48

We try to make MySQL Server follow the ANSI SQL standard and the ODBC SQL standard, but MySQL Server performs operations differently in some cases:

- There are several differences between the MySQL and standard SQL privilege systems. For example, in MySQL, privileges for a table are not automatically revoked when you delete a table. You must explicitly issue a [REVOKE](#) statement to revoke privileges for a table. For more information, see [REVOKE Statement](#).
- The `CAST()` function does not support cast to `REAL` or `BIGINT`. See [Cast Functions and Operators](#).

13.1 SELECT INTO TABLE Differences

MySQL Server doesn't support the `SELECT ... INTO TABLE` Sybase SQL extension. Instead, MySQL Server supports the `INSERT INTO ... SELECT` standard SQL syntax, which is basically the same thing. See [INSERT ... SELECT Statement](#). For example:

```
INSERT INTO tbl_temp2 (fld_id)
  SELECT tbl_temp1.fld_order_id
  FROM tbl_temp1 WHERE tbl_temp1.fld_order_id > 100;
```

Alternatively, you can use `SELECT ... INTO OUTFILE` or `CREATE TABLE ... SELECT`.

You can use `SELECT ... INTO` with user-defined variables. The same syntax can also be used inside stored routines using cursors and local variables. See [SELECT ... INTO Statement](#).

13.2 UPDATE Differences

If you access a column from the table to be updated in an expression, `UPDATE` uses the current value of the column. The second assignment in the following statement sets `col2` to the current (updated) `col1` value, not the original `col1` value. The result is that `col1` and `col2` have the same value. This behavior differs from standard SQL.

```
UPDATE t1 SET col1 = col1 + 1, col2 = col1;
```

13.3 FOREIGN KEY Constraint Differences

The MySQL implementation of foreign key constraints differs from the SQL standard in the following key respects:

- If there are several rows in the parent table with the same referenced key value, `InnoDB` performs a foreign key check as if the other parent rows with the same key value do not exist. For example, if you define a `RESTRICT` type constraint, and there is a child row with several parent rows, `InnoDB` does not permit the deletion of any of the parent rows.
- If `ON UPDATE CASCADE` or `ON UPDATE SET NULL` recurses to update the *same table* it has previously updated during the same cascade, it acts like `RESTRICT`. This means that you cannot use self-

referential `ON UPDATE CASCADE` or `ON UPDATE SET NULL` operations. This is to prevent infinite loops resulting from cascaded updates. A self-referential `ON DELETE SET NULL`, on the other hand, is possible, as is a self-referential `ON DELETE CASCADE`. Cascading operations may not be nested more than 15 levels deep.

- In an SQL statement that inserts, deletes, or updates many rows, foreign key constraints (like unique constraints) are checked row-by-row. When performing foreign key checks, `InnoDB` sets shared row-level locks on child or parent records that it must examine. MySQL checks foreign key constraints immediately; the check is not deferred to transaction commit. According to the SQL standard, the default behavior should be deferred checking. That is, constraints are only checked after the *entire SQL statement* has been processed. This means that it is not possible to delete a row that refers to itself using a foreign key.
- No storage engine, including `InnoDB`, recognizes or enforces the `MATCH` clause used in referential-integrity constraint definitions. Use of an explicit `MATCH` clause does not have the specified effect, and it causes `ON DELETE` and `ON UPDATE` clauses to be ignored. Specifying the `MATCH` should be avoided.

The `MATCH` clause in the SQL standard controls how `NULL` values in a composite (multiple-column) foreign key are handled when comparing to a primary key in the referenced table. MySQL essentially implements the semantics defined by `MATCH SIMPLE`, which permits a foreign key to be all or partially `NULL`. In that case, a (child table) row containing such a foreign key can be inserted even though it does not match any row in the referenced (parent) table. (It is possible to implement other semantics using triggers.)

- MySQL requires that the referenced columns be indexed for performance reasons. However, MySQL does not enforce a requirement that the referenced columns be `UNIQUE` or be declared `NOT NULL`.

A `FOREIGN KEY` constraint that references a non-`UNIQUE` key is not standard SQL but rather an `InnoDB` extension. The `NDB` storage engine, on the other hand, requires an explicit unique key (or primary key) on any column referenced as a foreign key.

The handling of foreign key references to nonunique keys or keys that contain `NULL` values is not well defined for operations such as `UPDATE` or `DELETE CASCADE`. You are advised to use foreign keys that reference only `UNIQUE` (including `PRIMARY`) and `NOT NULL` keys.

- For storage engines that do not support foreign keys (such as `MyISAM`), MySQL Server parses and ignores foreign key specifications.
- MySQL parses but ignores “inline `REFERENCES` specifications” (as defined in the SQL standard) where the references are defined as part of the column specification. MySQL accepts `REFERENCES` clauses only when specified as part of a separate `FOREIGN KEY` specification.

Defining a column to use a `REFERENCES tbl_name(col_name)` clause has no actual effect and *serves only as a memo or comment to you that the column which you are currently defining is intended to refer to a column in another table*. It is important to realize when using this syntax that:

- MySQL does not perform any sort of check to make sure that `col_name` actually exists in `tbl_name` (or even that `tbl_name` itself exists).
- MySQL does not perform any sort of action on `tbl_name` such as deleting rows in response to actions taken on rows in the table which you are defining; in other words, this syntax induces no `ON DELETE` or `ON UPDATE` behavior whatsoever. (Although you can write an `ON DELETE` or `ON UPDATE` clause as part of the `REFERENCES` clause, it is also ignored.)
- This syntax creates a *column*; it does **not** create any sort of index or key.

You can use a column so created as a join column, as shown here:

```

CREATE TABLE person (
  id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  name CHAR(60) NOT NULL,
  PRIMARY KEY (id)
);
CREATE TABLE shirt (
  id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  style ENUM('t-shirt', 'polo', 'dress') NOT NULL,
  color ENUM('red', 'blue', 'orange', 'white', 'black') NOT NULL,
  owner SMALLINT UNSIGNED NOT NULL REFERENCES person(id),
  PRIMARY KEY (id)
);
INSERT INTO person VALUES (NULL, 'Antonio Paz');
SELECT @last := LAST_INSERT_ID();
INSERT INTO shirt VALUES
(NULL, 'polo', 'blue', @last),
(NULL, 'dress', 'white', @last),
(NULL, 't-shirt', 'blue', @last);
INSERT INTO person VALUES (NULL, 'Lilliana Angelovska');
SELECT @last := LAST_INSERT_ID();
INSERT INTO shirt VALUES
(NULL, 'dress', 'orange', @last),
(NULL, 'polo', 'red', @last),
(NULL, 'dress', 'blue', @last),
(NULL, 't-shirt', 'white', @last);
SELECT * FROM person;
+-----+-----+
| id | name |
+-----+-----+
| 1 | Antonio Paz |
| 2 | Lilliana Angelovska |
+-----+-----+
SELECT * FROM shirt;
+-----+-----+-----+-----+
| id | style | color | owner |
+-----+-----+-----+-----+
| 1 | polo | blue | 1 |
| 2 | dress | white | 1 |
| 3 | t-shirt | blue | 1 |
| 4 | dress | orange | 2 |
| 5 | polo | red | 2 |
| 6 | dress | blue | 2 |
| 7 | t-shirt | white | 2 |
+-----+-----+-----+-----+
SELECT s.* FROM person p INNER JOIN shirt s
  ON s.owner = p.id
  WHERE p.name LIKE 'Lilliana%'
  AND s.color <> 'white';
+-----+-----+-----+-----+
| id | style | color | owner |
+-----+-----+-----+-----+
| 4 | dress | orange | 2 |
| 5 | polo | red | 2 |
| 6 | dress | blue | 2 |
+-----+-----+-----+-----+

```

When used in this fashion, the [REFERENCES](#) clause is not displayed in the output of [SHOW CREATE TABLE](#) or [DESCRIBE](#):

```

SHOW CREATE TABLE shirt\G
***** 1. row *****
Table: shirt
Create Table: CREATE TABLE `shirt` (
  `id` smallint(5) unsigned NOT NULL auto_increment,
  `style` enum('t-shirt','polo','dress') NOT NULL,

```

```
`color` enum('red','blue','orange','white','black') NOT NULL,  
`owner` smallint(5) unsigned NOT NULL,  
PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

For information about foreign key constraints, see [FOREIGN KEY Constraints](#).

13.4 '--' as the Start of a Comment

Standard SQL uses the C syntax `/* this is a comment */` for comments, and MySQL Server supports this syntax as well. MySQL also support extensions to this syntax that enable MySQL-specific SQL to be embedded in the comment; see [Comments](#).

MySQL Server also uses `#` as the start comment character. This is nonstandard.

Standard SQL also uses “`--`” as a start-comment sequence. MySQL Server supports a variant of the `--` comment style; the `--` start-comment sequence is accepted as such, but must be followed by a whitespace character such as a space or newline. The space is intended to prevent problems with generated SQL queries that use constructs such as the following, which updates the balance to reflect a charge:

```
UPDATE account SET balance=balance-charge  
WHERE account_id=user_id
```

Consider what happens when `charge` has a negative value such as `-1`, which might be the case when an amount is credited to the account. In this case, the generated statement looks like this:

```
UPDATE account SET balance=balance--1  
WHERE account_id=5752;
```

`balance--1` is valid standard SQL, but `--` is interpreted as the start of a comment, and part of the expression is discarded. The result is a statement that has a completely different meaning than intended:

```
UPDATE account SET balance=balance  
WHERE account_id=5752;
```

This statement produces no change in value at all. To keep this from happening, MySQL requires a whitespace character following the `--` for it to be recognized as a start-comment sequence in MySQL Server, so that an expression such as `balance-- 1` is always safe to use.

Chapter 14 Known Issues in MySQL

This section lists known issues in recent versions of MySQL.

For information about platform-specific issues, see the installation and debugging instructions in [General Installation Guidance](#), and [Debugging MySQL](#).

The following problems are known:

- Subquery optimization for `IN` is not as effective as for `=`.
- Even if you use `lower_case_table_names=2` (which enables MySQL to remember the case used for databases and table names), MySQL does not remember the case used for database names for the function `DATABASE()` or within the various logs (on case-insensitive systems).
- Dropping a `FOREIGN KEY` constraint does not work in replication because the constraint may have another name on the replica.
- `REPLACE` (and `LOAD DATA` with the `REPLACE` option) does not trigger `ON DELETE CASCADE`.
- `DISTINCT` with `ORDER BY` does not work inside `GROUP_CONCAT()` if you do not use all and only those columns that are in the `DISTINCT` list.
- When inserting a big integer value (between 2^{63} and $2^{64}-1$) into a decimal or string column, it is inserted as a negative value because the number is evaluated in signed integer context.
- With statement-based binary logging, the source server writes the executed queries to the binary log. This is a very fast, compact, and efficient logging method that works perfectly in most cases. However, it is possible for the data on the source and replica to become different if a query is designed in such a way that the data modification is nondeterministic (generally not a recommended practice, even outside of replication).

For example:

- `CREATE TABLE ... SELECT` or `INSERT ... SELECT` statements that insert zero or `NULL` values into an `AUTO_INCREMENT` column.
- `DELETE` if you are deleting rows from a table that has foreign keys with `ON DELETE CASCADE` properties.
- `REPLACE ... SELECT, INSERT IGNORE ... SELECT` if you have duplicate key values in the inserted data.

If and only if the preceding queries have no `ORDER BY` clause guaranteeing a deterministic order.

For example, for `INSERT ... SELECT` with no `ORDER BY`, the `SELECT` may return rows in a different order (which results in a row having different ranks, hence getting a different number in the `AUTO_INCREMENT` column), depending on the choices made by the optimizers on the source and replica.

A query is optimized differently on the source and replica only if:

- The table is stored using a different storage engine on the source than on the replica. (It is possible to use different storage engines on the source and replica. For example, you can use `InnoDB` on the source, but `MyISAM` on the replica if the replica has less available disk space.)
- MySQL buffer sizes (`key_buffer_size`, and so on) are different on the source and replica.

-
- The source and replica run different MySQL versions, and the optimizer code differs between these versions.

This problem may also affect database restoration using `mysqlbinlog|mysql`.

The easiest way to avoid this problem is to add an `ORDER BY` clause to the aforementioned nondeterministic queries to ensure that the rows are always stored or modified in the same order. Using row-based or mixed logging format also avoids the problem.

- Log file names are based on the server host name if you do not specify a file name with the startup option. To retain the same log file names if you change your host name to something else, you must explicitly use options such as `--log-bin=old_host_name-bin`. See [Server Command Options](#). Alternatively, rename the old files to reflect your host name change. If these are binary logs, you must edit the binary log index file and fix the binary log file names there as well. (The same is true for the relay logs on a replica.)
- `mysqlbinlog` does not delete temporary files left after a `LOAD DATA` statement. See [mysqlbinlog — Utility for Processing Binary Log Files](#).
- `RENAME` does not work with `TEMPORARY` tables or tables used in a `MERGE` table.
- When using `SET CHARACTER SET`, you cannot use translated characters in database, table, and column names.
- Prior to MySQL 8.0.17, you cannot use `_` or `%` with `ESCAPE` in `LIKE ... ESCAPE`.
- The server uses only the first `max_sort_length` bytes when comparing data values. This means that values cannot reliably be used in `GROUP BY`, `ORDER BY`, or `DISTINCT` if they differ only after the first `max_sort_length` bytes. To work around this, increase the variable value. The default value of `max_sort_length` is 1024 and can be changed at server startup time or at runtime.
- Numeric calculations are done with `BIGINT` or `DOUBLE` (both are normally 64 bits long). Which precision you get depends on the function. The general rule is that bit functions are performed with `BIGINT` precision, `IF()` and `ELT()` with `BIGINT` or `DOUBLE` precision, and the rest with `DOUBLE` precision. You should try to avoid using unsigned long long values if they resolve to be larger than 63 bits (9223372036854775807) for anything other than bit fields.
- You can have up to 255 `ENUM` and `SET` columns in one table.
- In `MIN()`, `MAX()`, and other aggregate functions, MySQL currently compares `ENUM` and `SET` columns by their string value rather than by the string's relative position in the set.
- In an `UPDATE` statement, columns are updated from left to right. If you refer to an updated column, you get the updated value instead of the original value. For example, the following statement increments `KEY` by 2, **not** 1:

```
mysql> UPDATE tbl_name SET KEY=KEY+1,KEY=KEY+1;
```
- You can refer to multiple temporary tables in the same query, but you cannot refer to any given temporary table more than once. For example, the following does not work:

```
mysql> SELECT * FROM temp_table, temp_table AS t2;  
ERROR 1137: Can't reopen table: 'temp_table'
```
- The optimizer may handle `DISTINCT` differently when you are using “hidden” columns in a join than when you are not. In a join, hidden columns are counted as part of the result (even if they are not shown), whereas in normal queries, hidden columns do not participate in the `DISTINCT` comparison.

An example of this is:

```
SELECT DISTINCT mp3id FROM band_downloads
WHERE userid = 9 ORDER BY id DESC;
```

and

```
SELECT DISTINCT band_downloads.mp3id
FROM band_downloads,band_mp3
WHERE band_downloads.userid = 9
AND band_mp3.id = band_downloads.mp3id
ORDER BY band_downloads.id DESC;
```

In the second case, you may get two identical rows in the result set (because the values in the hidden `id` column may differ).

Note that this happens only for queries that do not have the `ORDER BY` columns in the result.

- If you execute a `PROCEDURE` on a query that returns an empty set, in some cases the `PROCEDURE` does not transform the columns.
- Creation of a table of type `MERGE` does not check whether the underlying tables are compatible types.
- If you use `ALTER TABLE` to add a `UNIQUE` index to a table used in a `MERGE` table and then add a normal index on the `MERGE` table, the key order is different for the tables if there was an old, non-`UNIQUE` key in the table. This is because `ALTER TABLE` puts `UNIQUE` indexes before normal indexes to be able to detect duplicate keys as early as possible.

